Correctness and Complexity of BMSSP in Isabelle/HOL

Arthur Freitas Ramos David Barros Hulak
Ruy J. G. B. de Queiroz

July 14, 2026

Abstract

This development formalizes the bounded multi-source shortest path (BMSSP) subproblem underlying the deterministic directed single-source shortest path algorithm of Duan, Mao, Mao, Shu, and Yin [2]. It proves three explicit claims. First, the executable `bmssp_distances` computes the exact reachable shortest-distance map for well-formed finite natural weighted digraphs; this executable instantiates the bucketed work-list with block size one, so it is a verified Dijkstra-style instantiation of the bucketed partition, proved directly against the shortest-path semantics rather than by refining the recursive cost relation. Second, a separate costed BMSSP recurrence—a recursive inductive relation, not runnable code—satisfies the $O(m \log^{2/3} n)$ bound under the abstract Insert/BatchPrepend/Pull operation-cost interface; a cost bridge shows the concrete bucketed operation costs fill the recurrence's cost slots at the logarithmic schedule scales, so the bound holds with the bucketed costs rather than free parameters, and a non-conditional runtime theorem charges those bucketed costs inside the recurrence under exactly the standing hypotheses of the abstract headline. Third, the bucketed partition operations realize the primitive costs required by that interface. The runtime locale is shown non-vacuous not only at one fixed graph but across a size-indexed family of directed paths of every size. The formalization does not state a single generated-code machine-step theorem for the executable, and the executable and the asymptotic analysis are separate checked artifacts that share the partition primitives. The entry also exports executable SML and includes a checked worked shortest-path example whose evaluated output is $[(0,0),(1,3),(2,5),(3,8),(4,6)]$.

AI assistance was used for proof engineering. The final definitions, statements, and proofs are checked by Isabelle.

Contents

1 Background **5**

2 Roadmap of the Formalization **5**

3 Assumptions and Scope **7**

4 The Headline Theorems **7**

5 The Bucketed Partition Data Structure **8**

6 Executable Export and Worked Example **9**

7 Checked Source **10**

8 Correctness Interface for BMSSP **10**

9 Abstract BMSSP Correctness **16**

10 Shortest-Path Prefix Facts **27**

11 Concrete Base Case **51**

12 Partition Data-Structure Interface **59**

13 Bucketed Partition Data Structure **77**

14 Cost Budgets for the Bucketed Structure **188**
14.1 Paper-Facing Costed Operations 255

15 Bucketed Partition **264**

16 Executable Bucketed BMSSP Smoke Test **264**

17 Executable Base-Case Ordering **279**

18 Unique Shortest-Path Tree **298**

19 FindPivots Core Relaxation Lemmas **305**

20 Concrete FindPivots Rounds **311**

21 Pull Minimum **356**

22 Partition Pull to BMSSP Pull Minimum **378**

23 Initial Labels **381**

24 Concrete One-Step BMSSP Assembly **384**

25 Concrete Top-Level Correctness **387**

26 Level-Indexed BMSSP Relation **389**

27 Operational Pull Step **391**

28 Core Cost Accounting **416**

29 Logarithmic Parameter Schedule **436**

30 Range-Synchronised Costed Runs **517**

31 Exact Split Range-Costed Runs **615**

32 Direct-Insert Costed Runs **639**

33 Strict Tie-Breaking Consequences **674**

34 Concrete Partition Data Structure **762**

35 Exact Concrete Cost Runs **779**

36 Top-Level BMSSP Theorems **824**

37 Reduction Contract for the Asymptotic Bound **856**

38 Executable Graph Refinement Bridge **858**

39 Executable Refinement **993**

40 Executable Headline **993**

41 A Concrete Non-Vacuous Instance of the Runtime Locale **996**

42 Bridging the Bucketed Operation Costs to the Schedule Parameters **999**

42.1 Logarithmic Arithmetic 1000
42.2 Bucketed Budgets at the Ratio Bound 1001
42.3 Monotonicity of the Abstract Cost Predicates 1001
42.4 Bucketed Operations Realise the Schedule Cost Parameters . 1002
42.5 The Bucketed Cost Parameters Sit at the Schedule Scales . . 1004
42.6 Connected Headline: Bucketed-Costed BMSSP is $O(m\ log^{2/3}\ n)$1006

43 A Size-Indexed Family of Runtime Instances 1009
43.1 The Path Graph of Size n 1009
43.2 The Path Family Locale 1010
43.2.1 The Only Walk From the Source Is an Initial Segment 1010
43.2.2 Reachability and Uniqueness 1011
43.2.3 Bounded Out-Degree and Positive Weights 1013
43.2.4 The Reduced Positive Instance and Its Running-Time Bound 1013
43.3 The Bound Holds at Every Size 1015

44 Non-Conditional Bucketed Running-Time Bound 1016
44.1 A One-Third-Scale Envelope for the BatchPrepend Cost . . . 1016
44.2 A Decoupled Amortised Bound: Geometry Exponent vs. Operation Cost 1018
44.3 A Costed Run With Decoupled Geometry and Bucketed Costs 1021
44.4 Totality of the Bucketed Cost 1022
44.5 Cost Bound for a Bucketed Run 1023
44.6 The Non-Conditional Bucketed Running-Time Bound 1025

1 Background

The single-source shortest path problem asks for the distance from one source vertex to every reachable vertex of a directed graph with non-negative edge weights. Dijkstra's algorithm [1] is the classical starting point: it maintains tentative labels and repeatedly settles a vertex of minimum tentative distance. With Fibonacci heaps, Fredman and Tarjan [3] obtained the well-known $O(m + n \log n)$ implementation for graphs with n vertices and m edges. For sparse directed graphs this bound became the standard deterministic barrier: the logarithmic sorting-like term is paid by the priority-queue discipline that extracts vertices one by one.

Duan, Mao, Mao, Shu, and Yin [2] break this barrier by changing the recursive unit of work. Their algorithm is not simply Dijkstra with a faster heap. The central object is BMSSP, a bounded multi-source shortest path subproblem. A BMSSP call starts from a set of already relevant sources and an upper bound B. It only has to complete vertices whose shortest distances fall below the bound and whose shortest paths pass through the current source set. Recursive calls therefore work on ranges of distances rather than on one global queue of all unsettled vertices.

The formalization follows that structure. At the correctness level, the algorithm is expressed in terms of finite directed graphs, simple walks, shortest-path distances, and BMSSP pre/postconditions. At the cost level, the recursive proof is parameterized by an abstract partition interface with Insert, BatchPrepend, and Pull operations. The runtime theorem is proved once against that interface and the paper's logarithmic schedule. The concrete bucketed data structure is then shown to realize the interface with the critical operation bounds. This cost statement is separate from the functional-correctness theorem for the exported executable.

The essential improvement is the ratio inside the logarithm. A sorted-list or ordinary balanced-tree representation searches among all N live entries and naturally gives a $\log N$ term. The paper's partition structure searches among bucket boundaries, so Insert pays $O(\max(1, \log(N/M)))$, where M is the block-size parameter. With the schedule used by the recursive BMSSP analysis, this distinction is exactly what preserves the $O(m \log^{2/3} n)$ headline.

2 Roadmap of the Formalization

The formal development is organized by proof layer. The table below gives one or two primary entry points for each layer and the checked surface that a reader should look for there; supporting theories appear nearby in the session order.

Layer	Start here	Checked surface
Semantic target	`BMSSP_Correctness`	Finite weighted digraphs, shortest-path distances, BMSSP pre/postconditions, and root-call correctness for ordinary SSSP.
Recursive BMSSP	`BMSSP_Algorithm_Correctness`; `BMSSP_Recursive`	Correctness of FindPivots, base cases, partition-loop traces, recursive calls, and final assembly of completed vertices.
Partition interface	`BMSSP_Partition_Interface`; `BMSSP_Partition_Pull_Bridge`	Abstract Insert, BatchPrepend, and Pull contracts, with the algebraic cost predicates used by the recurrence proof.
Cost recurrence	`BMSSP_Top_Level_Bounds`	The logarithmic parameter schedule and the cost-model theorem `bmssp_runtime_bigo_target`.
Bucketed data structure	`BMSSP_Bucketed_Partition`	Concrete bucketed operations satisfying `bp_insert_cost_bound`, `bp_batch_prepend_cost_bound`, `bp_pull_cost_bound`, and the interface facts `bp_realises_partition_insert_cost_bound`, `bp_realises_partition_batch_cost_bound`, and `bp_realises_partition_pull_cost_bound`.
Executable artifact	`BMSSP_Executable_Headline`	The executable `bmssp_distances`, the checked example `example_bmssp_correct`, and the headline theorem `bmssp_correct_strong`.
Cost bridge	`BMSSP_Bucketed_Cost_Bridge`	Connects the bucketed operation costs to the recurrence cost slots at the logarithmic schedule scales; capstone `bucketed_refined_cost_bigo_sssp_time_target`.
Runtime family	`BMSSP_Path_Family`	A directed-path runtime instance of every size; the bound `path_family_runtime_bigo_target` with edge count `path_family_edge_count` growing without bound.
Non-conditional runtime	`BMSSP_Bucketed_Runtime`	The hypothesis-free `bmssp_bucketed_runtime_bigo_target`: the bucketed operation costs charged inside the recurrence yield $O(m \log^{2/3} n)$, via the decoupled amortised bound `finite_initial_label_decoupled_amortized`.

Several additional theories separate the mathematical layers used below: shortest-path semantics, abstract BMSSP correctness, costed recurrences, partition-interface costs, bucketed primitive operations, executable code, and the final executable wrapper. The cost-model theorems and the executable correctness theorem are deliberately separate claims.

3 Assumptions and Scope

The entry proves three related, but not identical, layers.

First, the semantic BMSSP development starts from finite weighted digraphs, simple walks, shortest-path distances, and a bounded BMSSP pre/postcondition. The recursive correctness proof is organized through FindPivots, partition loop obligations, complete sources, and bounded distance ranges.

Second, the asymptotic development is a cost-model proof. It solves the costed BMSSP recurrence under the abstract Insert/BatchPrepend/Pull interface, uses the logarithmic parameter schedule from the paper, and proves the `bmssp_runtime_bigo_target` theorem in the corresponding runtime locale. This is not stated as a machine-step bound for generated SML.

Third, the executable development proves functional correctness for the exported finite natural-weight graph representation. The public theorem `bmssp_correct_strong` applies to every G satisfying `nat_graph_well_formed` G and every source src $\in$ `nat_graph_vertices` G. It does not assume unique shortest paths. The well-formedness predicate is a representation-side condition: the directed edge list is duplicate-free and the total natural edge weight is below the executable sentinel `bmssp_infinity`.

4 The Headline Theorems

The primary executable correctness statement is

`bmssp_correct_strong` in `BMSSP_Executable_Headline`.

The theorem fixes a finite directed graph G of natural-numbered vertices with non-negative natural edge weights and a source vertex src. The statement is

> For every G with `nat_graph_well_formed` G and every src in `nat_graph_vertices` G:
>
> - the keys of bmssp_distances G src are distinct;
> - the key set is exactly the set of vertices reachable from src;
> - every pair (v, d) returned by bmssp_distances G src satisfies $d =$ nat_graph_dist G src v after coercion to real numbers.

Thus the executable theorem is not conditional on a unique-shortest-path or tie-breaking assumption. Its displayed hypotheses are the finite natural-graph representation condition and the source-membership condition.

The runtime and bucketed-interface statements are separate public claims. `bmssp_runtime_bigo_target` is the cost-model theorem for the abstract BMSSP recurrence and logarithmic parameter schedule. Its closed assumption-free form is stated inside the runtime locale; interpreting that locale supplies the reduced positive instance, reachability, and bounded outdegree hypotheses. To certify that this locale is inhabited rather than vacuous, the theory `BMSSP_Runtime_Instance` interprets it on a concrete unit-weight directed path $0 \to 1 \to 2 \to 3$: it discharges unique shortest walks (from the verified simple-walk enumerator), positive weights, full reachability, and bounded outdegree, and thereby obtains `path_runtime_bigo_target`, the same logarithmic running-time bound as a closed, assumption-free statement for an explicit non-trivial graph. The corollaries `bp_realises_partition_insert_cost_bound`, `bp_realises_partition_batch_cost_bound`, and `bp_realises_partition_pull_cost_bound` show that the bucketed Insert, BatchPrepend, and Pull operations realize the abstract primitive-cost predicates used by that recurrence. The more concrete theorems `bp_insert_cost_bound`, `bp_batch_prepend_cost_bound`, and `bp_pull_cost_bound` expose the amortised budgets for the bucketed implementation itself.

The session build checks all of these claims. The executable smoke test `example_bmssp_correct` also executes the generated code equations on a worked five-vertex example with bmssp_distances example_graph $0 = [(0,0),(1,3),(2,5),(3,8),(4,6)]$, proved by `by eval` inside the session build itself.

5 The Bucketed Partition Data Structure

The partition data structure is the main concrete contribution of the executable layer. It stores tentative labels as key/value pairs grouped into buckets. Each bucket contains an unsorted list of at most M entries in the regular state, and the bucket markers form a monotone directory of lower bounds. The directory, rather than the full set of entries, is the object searched by Insert. If N entries have been inserted since the last Pull, there are only on the order of N/M buckets, which gives the $\log(N/M)$ search term.

The formal model separates several invariants. The ordinary invariant states that bucket sizes are within the block-size bound, markers are sorted, markers are lower bounds for their bucket entries, keys are distinct, and the functional value memory agrees with the entries stored in buckets. The ordered invariant additionally states that adjacent bucket boundaries separate

the value ranges. The lazy invariant relaxes the size side condition: a bucket may temporarily grow up to the lazy threshold before a split is forced.

The amortised analysis is deliberately close to the paper's intuition. A lazy Insert searches the bucket directory, inserts into one bucket, and only splits when the bucket has accumulated enough excess entries. The potential function records overflow beyond M together with a small slack term for the number of buckets. The proofs use a scaled credit invariant, so the actual cost of a split is paid by the overflow accumulated by previous insertions. After the split the potential drops, and the amortised Insert cost is the directory-search charge plus a constant amount of update work.

BatchPrepend receives a list of new candidates that belong before the currently exposed range. The concrete operation rebuckets that batch and prepends the resulting buckets in one operation, which is why its accounting is per item rather than a repeated ordinary Insert cost. Pull removes a first bucket when the first-bucket side conditions hold, and its scanned work is bounded by the block size. The theorem names reflect these three conclusions: `bp_insert_cost_bound`, `bp_batch_prepend_cost_bound`, and `bp_pull_cost_bound`.

The session also contains a baseline sorted-list partition model used by the exact recurrence and abstraction proofs. Sorted lists refine the same abstract contracts, but they search on N entries and therefore do not supply the paper's N/M ratio. The bucketed structure is the implementation connected to the active partition-interface realization facts.

6 Executable Export and Worked Example

The final theory gives a concrete executable surface. Vertices and edge weights are natural numbers, graphs are finite edge lists, and the work-list used by the executable loop is the bucketed partition from `BMSSP_Bucketed_Partition`. The entry point `bmssp_distances` computes a finite association list of distances for the example graph.

The executable entry point is a finite natural-weight instantiation using the bucketed work-list. Its functional correctness is proved independently against the shortest-path semantics. The asymptotic result applies to the costed BMSSP model and bucketed operation interface; the two layers share the same partition primitives but are distinct artifacts. The executable layer uses a finite upper sentinel for natural labels; the abstract correctness layer uses the `bound = Fin real | Infinity` datatype.

The example has source 0 and five vertices. The hand-computed distances are:

$$d(0) = 0, \quad d(1) = 3, \quad d(2) = 5, \quad d(3) = 8, \quad d(4) = 6.$$

The Isabelle theorem

`example_bmssp_correct`

states that the executable function returns exactly this association list. Its proof is `by eval`. Thus the bundled Poly/ML evaluator executes the generated code equations during the build; if the executable algorithm or the expected map is wrong, the theory does not check.

The same theory contains a `value` command for human inspection and an `export_code` command that emits SML module `BMSSP`. The session ROOT file includes an export directive so the generated code can be materialized as `generated/BMSSP.ML`.

7 Checked Source

The remainder of this document is the checked Isabelle source. Definitions, theorem statements, proof scripts, and document text are all processed by the same session build.

theory *BMSSP-Correctness*
  imports *Complex-Main*
begin

8 Correctness Interface for BMSSP

This is the entry point of the correctness layer. It deliberately avoids committing to the paper's data structure, cost accounting, perturbation reduction, or executable representation. The purpose here is to state the mathematical contract that every later algorithmic refinement has to meet.

The ambient object is a finite directed graph with non-negative real edge weights and a distinguished source vertex. Distances are not imported from a library shortest-path algorithm. Instead, they are defined directly as minima of finite sets of simple-walk weights. This gives the later proofs a small, transparent semantic target: a label is correct exactly when it agrees with the minimum simple-walk distance from the source.

BMSSP means bounded multi-source shortest path. A recursive BMSSP call is not asked to solve all of single-source shortest paths. It receives a set of sources, a bound, and a current label function. It must complete the vertices whose shortest paths pass through the source set and whose distances lie below the bound returned by the call. This bounded contract is the reason the paper can recurse on distance ranges rather than repeatedly extracting one global minimum vertex.

The definitions below therefore introduce three pieces of vocabulary that recur throughout the development. A bound can be finite or infinite; a bound tree is the set of vertices relevant to a source set below such a bound; and the BMSSP postcondition says that the output labels are complete on exactly that tree. Later theories refine how the tree is assembled: FindPivots identifies useful sources, the partition loop splits the distance range, and

the bucketed data structure realizes the required Insert, BatchPrepend, and Pull operations. The final theorems of this file show that a successful root BMSSP call is already enough to establish ordinary single-source shortest-path correctness.

datatype *bound* = *Fin real* | *Infinity*

fun *below-bound* :: *real* ⇒ *bound* ⇒ *bool* **where**
  below-bound x Infinity ⟷ *True*
| *below-bound x* (*Fin b*) ⟷ $x < b$

fun *bound-le* :: *bound* ⇒ *bound* ⇒ *bool* **where**
  bound-le - *Infinity* ⟷ *True*
| *bound-le Infinity* (*Fin* -) ⟷ *False*
| *bound-le* (*Fin a*) (*Fin b*) ⟷ $a \leq b$

The datatype *bound* is intentionally small. The predicate *below-bound* is strict for finite bounds, matching the paper's range convention, while *bound-le* is the preorder used when a recursive call returns a possibly smaller upper bound. Keeping this type separate from raw reals makes the top-level call with *Infinity* explicit and avoids ad hoc sentinel values in the correctness proof.

locale *finite-weighted-digraph* =
  fixes $V :: 'v\ set$
    and $E :: ('v \times 'v)\ set$
    and $w :: 'v \Rightarrow 'v \Rightarrow real$
    and $s :: 'v$
  assumes *finite-V*: *finite V*
    and *source-in-V*: $s \in V$
    and *edge-in-V*: $(u, v) \in E \Longrightarrow u \in V \wedge v \in V$
    and *nonneg-weight*: $(u, v) \in E \Longrightarrow 0 \leq w\ u\ v$
begin

fun *walk* :: *'v list* ⇒ *bool* **where**
  walk [] ⟷ *False*
| *walk* [*x*] ⟷ $x \in V$
| *walk* (x # y # *xs*) ⟷ $x \in V \wedge (x, y) \in E \wedge$ *walk* (y # *xs*)

fun *walk-weight* :: *'v list* ⇒ *real* **where**
  walk-weight [] = *0*
| *walk-weight* [*x*] = *0*
| *walk-weight* (x # y # *xs*) = $w\ x\ y$ + *walk-weight* (y # *xs*)

definition *walk-betw* :: *'v* ⇒ *'v list* ⇒ *'v* ⇒ *bool* **where**
  walk-betw a p b ⟷ $p \neq [] \wedge$ *hd* $p = a \wedge$ *last* $p = b \wedge$ *walk p*

definition *simple-walk-betw* :: *'v* ⇒ *'v list* ⇒ *'v* ⇒ *bool* **where**
  simple-walk-betw a p b ⟷ *walk-betw a p b* ∧ *distinct p*

definition *reachable* :: $'v \Rightarrow 'v \Rightarrow bool$ **where**
reachable a b $\longleftrightarrow$ $(\exists\, p.$ *simple-walk-betw a p b*$)$

definition *simple-walk-weights* :: $'v \Rightarrow 'v \Rightarrow real\ set$ **where**
simple-walk-weights a b = *walk-weight* ‘ $\{p.$ *simple-walk-betw a p b*$\}$

definition *dist* :: $'v \Rightarrow 'v \Rightarrow real$ **where**
dist a b = *Min* (*simple-walk-weights a b*)

definition *shortest-walk* :: $'v \Rightarrow 'v\ list \Rightarrow 'v \Rightarrow bool$ **where**
shortest-walk a p b $\longleftrightarrow$ *simple-walk-betw a p b* $\wedge$ *walk-weight p* = *dist a b*

definition *through* :: $'v\ set \Rightarrow 'v \Rightarrow bool$ **where**
through S v $\longleftrightarrow$ $(\exists\, u{\in}S.\ \exists\, p.$ *shortest-walk s p v* $\wedge$ *u* $\in$ *set p*$)$

definition *bound-tree* :: $'v\ set \Rightarrow bound \Rightarrow 'v\ set$ **where**
bound-tree S B =
$\{v \in V.$ *reachable s v* $\wedge$ *through S v* $\wedge$ *below-bound* (*dist s v*) *B*$\}$

definition *complete-on* :: $('v \Rightarrow real) \Rightarrow 'v\ set \Rightarrow bool$ **where**
complete-on d U $\longleftrightarrow$ $(\forall\, v{\in}U.$ *reachable s v* $\longrightarrow$ *d v* = *dist s v*$)$

definition *sssp-correct* :: $('v \Rightarrow real) \Rightarrow bool$ **where**
sssp-correct d $\longleftrightarrow$ $(\forall\, v{\in}V.$ *reachable s v* $\longrightarrow$ *d v* = *dist s v*$)$

definition *bmssp-pre* :: $('v \Rightarrow real) \Rightarrow 'v\ set \Rightarrow bound \Rightarrow bool$ **where**
bmssp-pre d S B $\longleftrightarrow$
S $\subseteq$ *V* $\wedge$
$(\forall\, v{\in}V.$ *reachable s v* $\longrightarrow$ *below-bound* (*dist s v*) *B* $\longrightarrow$
d v $\neq$ *dist s v* $\longrightarrow$ *through S v*$)$

definition *bmssp-post* ::
$('v \Rightarrow real) \Rightarrow 'v\ set \Rightarrow bound \Rightarrow ('v \Rightarrow real) \Rightarrow bound \Rightarrow 'v\ set \Rightarrow bool$ **where**
bmssp-post d S B d′ B′ U $\longleftrightarrow$
bound-le B′ B $\wedge$ *U* = *bound-tree S B′* $\wedge$ *complete-on d′ U*

The graph locale makes the semantic model finite at the point where finiteness is needed. A walk is a list of vertices following edges; a simple walk is a walk with no repeated vertices; and *local.dist* is the minimum weight over simple walks. Because all edge weights are non-negative, restricting to simple walks preserves shortest distances while making the set minimized by *local.dist* finite.

The central BMSSP contract is expressed by *bmssp-pre* and *bmssp-post*. The precondition says that every not-yet-complete reachable vertex below the input bound must be reachable through the current source set, as formalized by *through*. The postcondition returns a new bound, an output set, and an updated label function; it demands that the output set is exactly *bound-tree* for the returned bound and that labels are complete there via *complete-*

on. This is intentionally weaker than global SSSP correctness, because the recursive algorithm only solves the range assigned to the current call.

lemma *walk-set-subset*:
assumes *walk p*
shows *set p* $\subseteq$ *V*
using *assms edge-in-V*
by (*induction p rule*: *walk.induct*) *auto*

lemma *simple-walk-length-le-card*:
assumes *simple-walk-betw a p b*
shows *length p* $\leq$ *card V*
proof −
have *walk p* **and** *distinct p*
using *assms* **unfolding** *simple-walk-betw-def walk-betw-def* **by** *auto*
then have *set p* $\subseteq$ *V*
using *walk-set-subset* **by** *blast*
with ‹*distinct p*› **have** *length p* = *card* (*set p*)
by (*simp add*: *distinct-card*)
also have ... $\leq$ *card V*
using ‹*set p* $\subseteq$ *V*› *finite-V* **by** (*simp add*: *card-mono*)
finally show *?thesis* **.**
qed

lemma *finite-simple-walks*:
shows *finite* {*p*. *simple-walk-betw a p b*}
proof (*rule finite-subset*)
show {*p*. *simple-walk-betw a p b*}
$\subseteq$ {*p*. *set p* $\subseteq$ *V* $\wedge$ *length p* $\leq$ *card V*}
using *simple-walk-length-le-card walk-set-subset*
unfolding *simple-walk-betw-def walk-betw-def* **by** *auto*
show *finite* {*p*. *set p* $\subseteq$ *V* $\wedge$ *length p* $\leq$ *card V*}
using *finite-lists-length-le*[*OF finite-V*] **.**
qed

lemma *finite-simple-walk-weights*:
shows *finite* (*simple-walk-weights a b*)
unfolding *simple-walk-weights-def*
using *finite-simple-walks* **by** *simp*

lemma *simple-walk-weights-nonempty*:
assumes *reachable a b*
shows *simple-walk-weights a b* $\neq$ {}
using *assms* **unfolding** *reachable-def simple-walk-weights-def* **by** *blast*

lemma *dist-is-simple-walk-weight*:
assumes *reachable a b*
shows *dist a b* $\in$ *simple-walk-weights a b*
unfolding *dist-def*
using *finite-simple-walk-weights simple-walk-weights-nonempty*[*OF assms*] **by**

(*rule Min-in*)

lemma *shortest-walk-exists*:
assumes *reachable a b*
obtains *p* **where** *shortest-walk a p b*
proof −
have *dist a b* ∈ *simple-walk-weights a b*
using *assms* **by** (*rule dist-is-simple-walk-weight*)
then have ∃ *p*. *simple-walk-betw a p b* ∧ *walk-weight p* = *dist a b*
unfolding *simple-walk-weights-def* **by** (*auto simp*: *image-iff*)
then obtain *p* **where** *simple-walk-betw a p b walk-weight p* = *dist a b*
by *blast*
then have *shortest-walk a p b*
unfolding *shortest-walk-def* **by** *blast*
then show *thesis*
using *that* **by** *blast*
qed

lemma *shortest-walk-hd*:
assumes *shortest-walk a p b*
shows *p* ≠ [] *hd p* = *a last p* = *b*
using *assms* **unfolding** *shortest-walk-def simple-walk-betw-def walk-betw-def* **by** *auto*

lemma *reachable-refl*:
assumes *v* ∈ *V*
shows *reachable v v*
using *assms* **unfolding** *reachable-def simple-walk-betw-def walk-betw-def* **by** *auto*

lemma *reachable-source-through*:
assumes *reachable s v*
shows *through* {*s*} *v*
proof −
obtain *p* **where** *p*: *shortest-walk s p v*
using *assms* **by** (*rule shortest-walk-exists*)
then have *p* ≠ [] *hd p* = *s*
using *shortest-walk-hd* **by** *blast*+
then have *s* ∈ *set p*
by (*metis hd-in-set*)
with *p* **show** *?thesis*
unfolding *through-def* **by** *blast*
qed

lemma *top-bmssp-pre*:
assumes ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
shows *bmssp-pre d* {*s*} *Infinity*
using *assms reachable-source-through source-in-V*
unfolding *bmssp-pre-def* **by** *auto*

lemma *bound-tree-source-infinity*:
assumes $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
shows *bound-tree* $\{s\}$ *Infinity* $= V$
using *assms reachable-source-through*
unfolding *bound-tree-def* **by** *auto*

lemma *bound-tree-source-infinity-reachable*:
shows *bound-tree* $\{s\}$ *Infinity* $= \{v \in V.$ *reachable s v*$\}$
using *reachable-source-through*
unfolding *bound-tree-def* **by** *auto*

lemma *bmssp-post-complete-bound-tree*:
assumes *bmssp-post d S B d' B' U*
and $v \in$ *bound-tree S B'*
shows $d'\ v =$ *dist s v*
proof −
have $U =$ *bound-tree S B'* **and** *complete-on d' U*
using *assms*(*1*) **unfolding** *bmssp-post-def* **by** *auto*
then have $v \in U$
using *assms*(*2*) **by** *simp*
moreover have *reachable s v*
using *assms*(*2*) **unfolding** *bound-tree-def* **by** *auto*
then show *?thesis*
using ‹*complete-on d' U*› *calculation* **unfolding** *complete-on-def* **by** *auto*
qed

lemma *bmssp-success-completes-requested-tree*:
assumes *bmssp-post d S B d' B U*
and $v \in$ *bound-tree S B*
shows $d'\ v =$ *dist s v*
using *bmssp-post-complete-bound-tree*[*OF assms*] **.**

lemma *sssp-correctI*:
assumes $\bigwedge v.\ [\![v \in V;$ *reachable s v*$]\!] \Longrightarrow d\ v =$ *dist s v*
shows *sssp-correct d*
using *assms* **unfolding** *sssp-correct-def* **by** *blast*

The remaining lemmas discharge the semantic obligations introduced above. Finiteness of simple walks justifies the use of *Min*; reachability of the source shows that the root source set covers every reachable vertex; and $[\![$*bmssp-post ?d ?S ?B ?d' ?B' ?U*; *?v* ∈ *bound-tree ?S ?B'*$]\!] \Longrightarrow$ *?d' ?v* = *local.dist s ?v* extracts a pointwise distance equality from the BMSSP postcondition. The final two theorems are the bridge from a root BMSSP result to the ordinary SSSP predicate *sssp-correct*. Later files prove BMSSP postconditions for concrete recursive executions, and then reuse these root theorems unchanged.

theorem *successful-root-bmssp-correct*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*

```
    and post: bmssp-post d {s} Infinity d' Infinity U
  shows sssp-correct d'
proof −
  have U = V
    using post bound-tree-source-infinity[OF all-reachable]
    unfolding bmssp-post-def by auto
  moreover have complete-on d' U
    using post unfolding bmssp-post-def by auto
  ultimately show ?thesis
    unfolding complete-on-def sssp-correct-def by auto
qed

theorem successful-root-bmssp-sssp-correct:
  assumes post: bmssp-post d {s} Infinity d' Infinity U
  shows sssp-correct d'
proof (rule sssp-correctI)
  fix v
  assume vV: v ∈ V and reach-v: reachable s v
  have v ∈ bound-tree {s} Infinity
    using vV reach-v reachable-source-through unfolding bound-tree-def by auto
  then show d' v = dist s v
    by (rule bmssp-post-complete-bound-tree[OF post])
qed

end

end
theory BMSSP-Algorithm-Correctness
  imports BMSSP-Correctness
begin
```

9 Abstract BMSSP Correctness

This theory is the correctness skeleton of the BMSSP development. The previous entry-point theory defines distances, bounds, and the basic bounded multi-source postcondition. Here we refine that interface just far enough to express the recursive algorithm described in the paper, while still abstracting from running time and from the concrete partition data structure.

The key strengthening is that sources used by a recursive call are required to be complete sources: their current labels already equal their true shortest-path distances. This matters because BMSSP does not merely carry a set of vertices; it carries a set of vertices whose shortest-path trees may certify the rest of the current bounded range. A vertex that is not solved yet must either be solved directly by the current round or lie on a shortest path through one of those complete sources.

At this level the algorithm is decomposed into three abstract obligations:

- *FindPivots* must either complete a bounded vertex or delegate it to a complete pivot.
- The partition loop driven by *Pull*, recursive BMSSP calls, edge relaxation, and *BatchPrepend* must solve the bounded problem induced by the pivot set.
- The final assembly of recursive-loop vertices with the completed *FindPivots* vertices is then proved correct.

The main theorem of the file, *abstract-bmssp-correct*, says that any implementation satisfying those obligations establishes the full BMSSP postcondition. Later theories instantiate the obligations with a concrete recursive loop, then the bucketed partition structure, and finally the executable example. None of the runtime arguments appear here; this file is the semantic hinge that lets the later cost proofs remain separate from the shortest-path correctness proof.

Two abstractions matter for navigating the rest of the development. First, this layer abstracts from the partition data structure entirely: any implementation satisfying the three obligations above is acceptable, so the later sorted-list and bucketed-partition refinements both plug in here without disturbing the correctness proof. Second, this layer abstracts from running time entirely: the cost predicates and amortized budgets live one layer further down, in `BMSSP_Top_Level_Bounds.thy` and `BMSSP_Bucketed_Partition.thy`, so that a reader interested only in correctness can stop here.

context *finite-weighted-digraph*
begin

definition *complete-vertices* :: $('v \Rightarrow real) \Rightarrow 'v\ set$ **where**
  $complete\text{-}vertices\ d = \{v \in V.\ reachable\ s\ v \wedge d\ v = dist\ s\ v\}$

definition *through-complete* :: $('v \Rightarrow real) \Rightarrow 'v\ set \Rightarrow 'v \Rightarrow bool$ **where**
  $through\text{-}complete\ d\ S\ v \longleftrightarrow through\ (S \cap complete\text{-}vertices\ d)\ v$

definition *bmssp-pre-full* :: $('v \Rightarrow real) \Rightarrow 'v\ set \Rightarrow bound \Rightarrow bool$ **where**
  $bmssp\text{-}pre\text{-}full\ d\ S\ B \longleftrightarrow$
    $S \subseteq V \wedge$
    $(\forall v{\in}V.\ reachable\ s\ v \longrightarrow below\text{-}bound\ (dist\ s\ v)\ B \longrightarrow$
      $d\ v \neq dist\ s\ v \longrightarrow through\text{-}complete\ d\ S\ v)$

definition *bmssp-post-full* ::
  $('v \Rightarrow real) \Rightarrow 'v\ set \Rightarrow bound \Rightarrow ('v \Rightarrow real) \Rightarrow bound \Rightarrow 'v\ set \Rightarrow bool$ **where**
  $bmssp\text{-}post\text{-}full\ d\ S\ B\ d'\ B'\ U \longleftrightarrow$
    $bound\text{-}le\ B'\ B \wedge U = bound\text{-}tree\ S\ B' \wedge complete\text{-}on\ d'\ U$

definition *find-pivots-post* ::
  $('v \Rightarrow real) \Rightarrow 'v\ set \Rightarrow bound \Rightarrow ('v \Rightarrow real) \Rightarrow 'v\ set \Rightarrow 'v\ set \Rightarrow bool$ **where**

find-pivots-post d S B d′ P W ⟷
  $S \subseteq V \wedge$
  $P \subseteq S \wedge$
  $W \subseteq$ *bound-tree S B* $\wedge$
  complete-on d′ W $\wedge$
  $(\forall\, v \in V.$ *reachable s v* $\longrightarrow$ *below-bound* (*dist s v*) *B* $\longrightarrow$
    d v = *dist s v* $\longrightarrow$ *d′ v* = *dist s v*) $\wedge$
  $(\forall\, v \in$ *bound-tree S B*. $v \in W \vee$ *through-complete d′ P v*)

definition *partition-loop-post* ::
  (′*v* ⇒ *real*) ⇒ ′*v set* ⇒ *bound* ⇒ (′*v* ⇒ *real*) ⇒ *bound* ⇒ ′*v set* ⇒ *bool* **where**
  partition-loop-post d P B d′ B′ U ⟷ *bmssp-post-full d P B d′ B′ U*

definition *final-bmssp-assembly* ::
  ′*v set* ⇒ *bound* ⇒ ′*v set* ⇒ ′*v set* ⇒ ′*v set* ⇒ *bool* **where**
  final-bmssp-assembly S B′ W U-loop U ⟷
    U = *U-loop* ∪ {*v* ∈ *W*. *below-bound* (*dist s v*) *B′*}

definition *abstract-bmssp-step* ::
  (′*v* ⇒ *real*) ⇒ ′*v set* ⇒ *bound* ⇒ (′*v* ⇒ *real*) ⇒ *bound* ⇒ ′*v set* ⇒ *bool* **where**
  abstract-bmssp-step d S B d′ B′ U ⟷
    $(\exists$ *d-fp P W U-loop*.
      find-pivots-post d S B d-fp P W $\wedge$
      partition-loop-post d-fp P B d′ B′ U-loop $\wedge$
      complete-on d′ W $\wedge$
      final-bmssp-assembly S B′ W U-loop U)

The definitions above separate the proof into named contracts. The set *complete-vertices* records exactly the vertices solved by a label function, and *through-complete* refines the earlier *through*-predicate by requiring the intermediary source to be solved. The strengthened precondition *bmssp-pre-full* is therefore the recursive invariant: every unresolved bounded vertex is certified by a shortest path through a complete source.

The predicate *find-pivots-post* is intentionally asymmetric. It returns a new label function, a pivot subset, and a set of vertices completed directly by the pivot search. Vertices in the current bounded tree are then either in that completed set or are delegated to the pivot subset via *through-complete*. The partition loop contract *partition-loop-post* is just the full BMSSP postcondition specialized to those pivots. Finally, *final-bmssp-assembly* describes the set union that reconstructs the caller's solved range from the recursive-loop output and the vertices completed by FindPivots.

This decomposition is useful because each later theory can prove one small interface. The algorithmic theory proves an instance of *abstract-bmssp-step*; the data-structure theories prove that the partition operations realize the abstract loop assumptions; the top-level theory only needs the postcondition produced here.

lemma *bmssp-post-full-imp-post*:

assumes *bmssp-post-full d S B d′ B′ U*
shows *bmssp-post d S B d′ B′ U*
using *assms* **unfolding** *bmssp-post-full-def bmssp-post-def* **by** *blast*

lemma *top-bmssp-pre-full*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *source-complete*: *d s* = *dist s s*
shows *bmssp-pre-full d* {*s*} *Infinity*
proof −
have *s* ∈ *complete-vertices d*
using *source-in-V source-complete reachable-refl*
unfolding *complete-vertices-def* **by** *blast*
then have $\bigwedge v.$ *reachable s v* $\Longrightarrow$ *through-complete d* {*s*} *v*
using *reachable-source-through* **unfolding** *through-complete-def* **by** *auto*
then show *?thesis*
using *all-reachable source-in-V* **unfolding** *bmssp-pre-full-def* **by** *auto*
qed

lemma *top-bmssp-pre-full-reachable*:
assumes *source-complete*: *d s* = *dist s s*
shows *bmssp-pre-full d* {*s*} *Infinity*
proof −
have *s* ∈ *complete-vertices d*
using *source-in-V source-complete reachable-refl*
unfolding *complete-vertices-def* **by** *blast*
then have $\bigwedge v.$ *reachable s v* $\Longrightarrow$ *through-complete d* {*s*} *v*
using *reachable-source-through* **unfolding** *through-complete-def* **by** *auto*
then show *?thesis*
using *source-in-V* **unfolding** *bmssp-pre-full-def* **by** *auto*
qed

lemma *through-complete-imp-through*:
assumes *through-complete d S v*
shows *through S v*
using *assms* **unfolding** *through-complete-def through-def* **by** *blast*

lemma *bound-tree-mono*:
assumes *S* ⊆ *T*
shows *bound-tree S B* ⊆ *bound-tree T B*
using *assms* **unfolding** *bound-tree-def through-def* **by** *blast*

lemma *bound-tree-bound-mono*:
assumes *bound-le B′ B*
shows *bound-tree S B′* ⊆ *bound-tree S B*
using *assms* **unfolding** *bound-tree-def*
by (*cases B′*; *cases B*) *auto*

lemma *complete-on-subset*:
assumes *complete-on d T S* ⊆ *T*

shows *complete-on d S*
using *assms* **unfolding** *complete-on-def* **by** *blast*

lemma *source-in-own-bound-tree*:
assumes *xS*: $x \in S$
and *reach-x*: *reachable s x*
and *below-x*: *below-bound* (*dist s x*) *B*
shows $x \in$ *bound-tree S B*
proof −
obtain *p* **where** *p*: *shortest-walk s p x*
using *reach-x* **by** (*rule shortest-walk-exists*)
have *x-in-p*: $x \in$ *set p*
using *shortest-walk-hd*(*1*,*3*)[*OF p*] **by** (*metis last-in-set*)
then have *through S x*
using *xS p* **unfolding** *through-def* **by** *blast*
moreover have $x \in V$
proof −
have *walk p*
using *p* **unfolding** *shortest-walk-def simple-walk-betw-def*
walk-betw-def **by** *blast*
then have *set p* $\subseteq V$
by (*rule walk-set-subset*)
then show *?thesis*
using *x-in-p* **by** *blast*
qed
ultimately show *?thesis*
using *reach-x below-x* **unfolding** *bound-tree-def* **by** *blast*
qed

lemma *sources-subset-bound-tree*:
assumes *reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *below*: $\bigwedge x.\ x \in S \Longrightarrow$ *below-bound* (*dist s x*) *B*
shows *S* $\subseteq$ *bound-tree S B*
using *source-in-own-bound-tree*[*OF - reaches below*] **by** *blast*

lemma *sources-subset-bound-tree-if-label-below*:
assumes *complete*: *complete-on d S*
and *reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *below*: $\bigwedge x.\ x \in S \Longrightarrow$ *below-bound* (*d x*) *B*
shows *S* $\subseteq$ *bound-tree S B*
proof (*rule sources-subset-bound-tree*)
fix *x*
assume *xS*: $x \in S$
then show *reachable s x*
by (*rule reaches*)
have *d x* = *dist s x*
using *complete reaches*[*OF xS*] *xS* **unfolding** *complete-on-def* **by** *blast*
then show *below-bound* (*dist s x*) *B*
using *below*[*OF xS*] **by** *simp*

```
qed

lemma bmssp-post-full-source-card-le-if-label-below-output:
  assumes post: bmssp-post-full d S B d' B' U
    and complete: complete-on d S
    and reaches: ⋀x. x ∈ S ⟹ reachable s x
    and below: ⋀x. x ∈ S ⟹ below-bound (d x) B'
  shows card S ≤ card U
proof −
  have U-eq: U = bound-tree S B'
    using post unfolding bmssp-post-full-def by blast
  have S-subset-U: S ⊆ U
    using sources-subset-bound-tree-if-label-below[OF complete reaches below]
    unfolding U-eq .
  have finite-U: finite U
    unfolding U-eq bound-tree-def using finite-V by auto
  show ?thesis
    by (rule card-mono[OF finite-U S-subset-U])
qed
```

The first group of lemmas is structural. It relates the strengthened complete-source invariant to the base BMSSP vocabulary and proves the monotonicity facts needed when the recursive call returns a smaller bound. The source-subset lemmas are small but important: they show that a complete source whose label is below the returned bound belongs to the returned *bound-tree*. That is the formal reason the recursive call cannot discard the pivots it was asked to solve.

```
lemma find-pivots-establishes-pivot-pre:
  assumes pre: bmssp-pre-full d S B
    and fp: find-pivots-post d S B d' P W
  shows bmssp-pre-full d' P B
proof −
  have P ⊆ V
    using fp unfolding find-pivots-post-def bound-tree-def by auto
  moreover have ⋀v. ⟦v ∈ V; reachable s v; below-bound (dist s v) B; d' v ≠ dist
s v⟧
      ⟹ through-complete d' P v
  proof −
    fix v
    assume v: v ∈ V reachable s v below-bound (dist s v) B d' v ≠ dist s v
    show through-complete d' P v
    proof (cases through S v)
      case True
      then have v ∈ bound-tree S B
        using v unfolding bound-tree-def by blast
      then have v ∈ W ∨ through-complete d' P v
        using fp unfolding find-pivots-post-def by blast
      then show ?thesis
      proof
```

```
      assume v ∈ W
      then have d' v = dist s v
        using fp v unfolding find-pivots-post-def complete-on-def by blast
      with v show ?thesis by blast
    qed blast
  next
    case False
    have d v = dist s v
    proof (rule ccontr)
      assume d v ≠ dist s v
      then have through-complete d S v
        using pre v unfolding bmssp-pre-full-def by blast
      then have through S v
        by (rule through-complete-imp-through)
      with False show False by blast
    qed
    then have d' v = dist s v
      using fp v unfolding find-pivots-post-def by blast
    with v show ?thesis by blast
  qed
 qed
 ultimately show ?thesis
  unfolding bmssp-pre-full-def by blast
qed

lemma conservative-find-pivots-post:
 assumes pre: bmssp-pre-full d S B
 defines W ≡ {v ∈ bound-tree S B. d v = dist s v}
 shows find-pivots-post d S B d S W
proof −
 have S: S ⊆ V
  using pre unfolding bmssp-pre-full-def by blast
 have W-subset: W ⊆ bound-tree S B
  unfolding W-def by blast
 have compW: complete-on d W
  unfolding W-def complete-on-def by blast
 have preserve: ∀ v∈V. reachable s v ⟶ below-bound (dist s v) B ⟶
    d v = dist s v ⟶ d v = dist s v
  by blast
 have cover: ∀ v∈bound-tree S B. v ∈ W ∨ through-complete d S v
 proof
  fix v
  assume v: v ∈ bound-tree S B
  show v ∈ W ∨ through-complete d S v
  proof (cases d v = dist s v)
    case True
    then show ?thesis
      using v unfolding W-def by blast
  next
```

```
    case False
    then show ?thesis
      using pre v unfolding bmssp-pre-full-def bound-tree-def by blast
  qed
 qed
 show ?thesis
  using S W-subset compW preserve cover
  unfolding find-pivots-post-def by blast
qed
```

The next two lemmas explain the handoff from FindPivots to recursion. The theorem $[\![$*bmssp-pre-full ?d ?S ?B*; *find-pivots-post ?d ?S ?B ?d′ ?P ?W*$]\!] \Longrightarrow$ *bmssp-pre-full ?d′ ?P ?B* is the crucial invariant transfer: after FindPivots, the pivot set satisfies *bmssp-pre-full* under the updated labels. If a vertex was already outside the old source tree, the preservation clause of *find-pivots-post* keeps its label correct; if it was inside the tree and not directly completed, the cover clause delegates it through a complete pivot.

The conservative postcondition is a sanity instance of the same interface. It says the abstract FindPivots contract is not empty: keeping the same sources and naming the already-complete part of the tree is sufficient. The executable development uses stronger concrete pivot selection, but this lemma is helpful for understanding the shape of the contract independently of any particular implementation.

```
lemma final-assembly-bound-tree:
 assumes fp: find-pivots-post d S B d-fp P W
  and loop: partition-loop-post d-fp P B d′ B′ U-loop
  and asm: final-bmssp-assembly S B′ W U-loop U
 shows U = bound-tree S B′
proof −
 have loop-post: bmssp-post-full d-fp P B d′ B′ U-loop
  using loop unfolding partition-loop-post-def .
 then have le: bound-le B′ B and Uloop: U-loop = bound-tree P B′
  unfolding bmssp-post-full-def by auto
 have P-subset: P ⊆ S
  using fp unfolding find-pivots-post-def by blast
 have W-subset: W ⊆ bound-tree S B
  using fp unfolding find-pivots-post-def by blast

 show ?thesis
 proof
  show U ⊆ bound-tree S B′
  proof
   fix v
   assume v ∈ U
   then consider v ∈ U-loop | v ∈ W below-bound (dist s v) B′
    using asm unfolding final-bmssp-assembly-def by blast
   then show v ∈ bound-tree S B′
   proof cases
```

```
      case 1
      then have v ∈ bound-tree P B'
        using Uloop by simp
      then show ?thesis
        using bound-tree-mono[OF P-subset, of B'] by blast
    next
      case 2
      then show ?thesis
        using W-subset unfolding bound-tree-def by blast
    qed
  qed
next
  show bound-tree S B' ⊆ U
  proof
    fix v
    assume v: v ∈ bound-tree S B'
    then have v ∈ bound-tree S B
      using bound-tree-bound-mono[OF le, of S] by blast
    then have cover: v ∈ W ∨ through-complete d-fp P v
      using fp unfolding find-pivots-post-def by blast
    then show v ∈ U
    proof
      assume v ∈ W
      moreover have below-bound (dist s v) B'
        using v unfolding bound-tree-def by blast
      ultimately show ?thesis
        using asm unfolding final-bmssp-assembly-def by blast
    next
      assume through-complete d-fp P v
      then have through P v
        by (rule through-complete-imp-through)
      then have v ∈ bound-tree P B'
        using v unfolding bound-tree-def by blast
      then have v ∈ U-loop
        using Uloop by simp
      then show ?thesis
        using asm unfolding final-bmssp-assembly-def by blast
    qed
  qed
qed

lemma final-assembly-complete:
  assumes fp: find-pivots-post d S B d-fp P W
    and loop: partition-loop-post d-fp P B d' B' U-loop
    and compW: complete-on d' W
    and asm: final-bmssp-assembly S B' W U-loop U
  shows complete-on d' U
proof −
```

have *complete-on d′ U-loop*
using *loop* **unfolding** *partition-loop-post-def bmssp-post-full-def* **by** *blast*
moreover have $\{v \in W.\ \textit{below-bound}\ (\textit{dist}\ s\ v)\ B'\} \subseteq W$
by *blast*
then have *complete-on d′* $\{v \in W.\ \textit{below-bound}\ (\textit{dist}\ s\ v)\ B'\}$
using *complete-on-subset*[*OF compW*] **by** *blast*
ultimately show *?thesis*
using *asm* **unfolding** *final-bmssp-assembly-def complete-on-def* **by** *blast*
qed

Final assembly is where the two solved regions are reconciled. The loop has solved *bound-tree* for the pivot set below the returned bound. The FindPivots phase may also have solved vertices in the caller's original tree; only those below the returned bound are retained in the caller's output set. Lemma ⟦*find-pivots-post ?d ?S ?B ?d-fp ?P ?W*; *partition-loop-post ?d-fp ?P ?B ?d′ ?B′ ?U-loop*; *final-bmssp-assembly ?S ?B′ ?W ?U-loop ?U*⟧ ⟹ *?U = bound-tree ?S ?B′* proves that this union is not merely contained in the desired set but exactly equals the caller's *bound-tree*. Lemma ⟦*find-pivots-post ?d ?S ?B ?d-fp ?P ?W*; *partition-loop-post ?d-fp ?P ?B ?d′ ?B′ ?U-loop*; *complete-on ?d′ ?W*; *final-bmssp-assembly ?S ?B′ ?W ?U-loop ?U*⟧ ⟹ *complete-on ?d′ ?U* then proves that labels are complete on that exact set.

theorem *abstract-bmssp-correct*:
assumes *step*: *abstract-bmssp-step d S B d′ B′ U*
shows *bmssp-post-full d S B d′ B′ U*
proof −
obtain *d-fp P W U-loop* **where**
fp: *find-pivots-post d S B d-fp P W* **and**
loop: *partition-loop-post d-fp P B d′ B′ U-loop* **and**
compW: *complete-on d′ W* **and**
asm: *final-bmssp-assembly S B′ W U-loop U*
using *step* **unfolding** *abstract-bmssp-step-def* **by** *blast*
have *le*: *bound-le B′ B*
using *loop* **unfolding** *partition-loop-post-def bmssp-post-full-def* **by** *blast*
have *U*: *U = bound-tree S B′*
using *final-assembly-bound-tree*[*OF fp loop asm*] **.**
have *comp*: *complete-on d′ U*
using *final-assembly-complete*[*OF fp loop compW asm*] **.**
show *?thesis*
using *le U comp* **unfolding** *bmssp-post-full-def* **by** *blast*
qed

theorem *abstract-bmssp-correct-with-pre*:
assumes *pre*: *bmssp-pre-full d S B*
and *step*: *abstract-bmssp-step d S B d′ B′ U*
shows *bmssp-post-full d S B d′ B′ U*
using *abstract-bmssp-correct*[*OF step*] **.**

lemma *abstract-bmssp-step-exposes-pivot-pre*:

assumes *pre*: *bmssp-pre-full d S B*
and *step*: *abstract-bmssp-step d S B d′ B′ U*
obtains *d-fp P W U-loop* **where**
find-pivots-post d S B d-fp P W
bmssp-pre-full d-fp P B
partition-loop-post d-fp P B d′ B′ U-loop
complete-on d′ W
final-bmssp-assembly S B′ W U-loop U
proof −
obtain *d-fp P W U-loop* **where**
fp: *find-pivots-post d S B d-fp P W* **and**
loop: *partition-loop-post d-fp P B d′ B′ U-loop* **and**
compW: *complete-on d′ W* **and**
asm: *final-bmssp-assembly S B′ W U-loop U*
using *step* **unfolding** *abstract-bmssp-step-def* **by** *blast*
have *bmssp-pre-full d-fp P B*
using *find-pivots-establishes-pivot-pre*[*OF pre fp*] **.**
then show *thesis*
using *that fp loop compW asm* **by** *blast*
qed

corollary *abstract-bmssp-correct-weak*:
assumes *abstract-bmssp-step d S B d′ B′ U*
shows *bmssp-post d S B d′ B′ U*
using *bmssp-post-full-imp-post abstract-bmssp-correct assms* **by** *blast*

The theorem *abstract-bmssp-step ?d ?S ?B ?d′ ?B′ ?U* $\Longrightarrow$ *bmssp-post-full ?d ?S ?B ?d′ ?B′ ?U* is the file's main assembly result: from an *abstract-bmssp-step* witness it extracts the FindPivots contract, the recursive partition-loop contract, and the final set assembly, then rebuilds *bmssp-post-full*. The variant ⟦*bmssp-pre-full ?d ?S ?B*; *abstract-bmssp-step ?d ?S ?B ?d′ ?B′ ?U*⟧ $\Longrightarrow$ *bmssp-post-full ?d ?S ?B ?d′ ?B′ ?U* keeps the precondition in its statement for clients whose proof scripts naturally carry it, while ⟦*bmssp-pre-full ?d ?S ?B*; *abstract-bmssp-step ?d ?S ?B ?d′ ?B′ ?U*; ⋀*d-fp P W U-loop*. ⟦*find-pivots-post ?d ?S ?B d-fp P W*; *bmssp-pre-full d-fp P ?B*; *partition-loop-post d-fp P ?B ?d′ ?B′ U-loop*; *complete-on ?d′ W*; *final-bmssp-assembly ?S ?B′ W U-loop ?U*⟧ $\Longrightarrow$ *?thesis*⟧ $\Longrightarrow$ *?thesis* exposes the pivot precondition needed by the recursive implementation.

The final top-level theorems connect this abstract step to ordinary single-source shortest paths. Once the root call uses source set $\{s\}$ and bound *Infinity*, the root correctness lemmas from the entry-point theory turn the BMSSP postcondition into *sssp-correct*.

theorem *abstract-top-level-sssp-correct*:
assumes *all-reachable*: ⋀*v*. *v* ∈ *V* $\Longrightarrow$ *reachable s v*
and *step*: *abstract-bmssp-step d* {*s*} *Infinity d′ Infinity U*
shows *sssp-correct d′*
proof −

have *bmssp-post d* {*s*} *Infinity d′ Infinity U*
using *abstract-bmssp-correct-weak*[*OF step*] **.**
then show *?thesis*
using *successful-root-bmssp-correct*[*OF all-reachable*] **by** *blast*
qed

theorem *abstract-top-level-reachable-sssp-correct*:
assumes *step*: *abstract-bmssp-step d* {*s*} *Infinity d′ Infinity U*
shows *sssp-correct d′*
proof −
have *bmssp-post d* {*s*} *Infinity d′ Infinity U*
using *abstract-bmssp-correct-weak*[*OF step*] **.**
then show *?thesis*
by (*rule successful-root-bmssp-sssp-correct*)
qed

theorem *initialized-abstract-top-level-sssp-correct*:
assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
and *source-complete*: *d s* = *dist s s*
and *step*: *abstract-bmssp-step d* {*s*} *Infinity d′ Infinity U*
shows *sssp-correct d′*
using *abstract-top-level-sssp-correct*[*OF all-reachable step*] **.**

theorem *initialized-abstract-top-level-reachable-sssp-correct*:
assumes *source-complete*: *d s* = *dist s s*
and *step*: *abstract-bmssp-step d* {*s*} *Infinity d′ Infinity U*
shows *sssp-correct d′*
using *abstract-top-level-reachable-sssp-correct*[*OF step*] **.**

end

end
theory *BMSSP-Shortest-Path-Lemmas*
imports *BMSSP-Correctness*
begin

10 Shortest-Path Prefix Facts

The algorithm's tree-based correctness arguments repeatedly use the fact that vertices appearing earlier on a shortest path have no larger true distance. This theory proves that fact from the finite non-negative weighted graph model, without assuming uniqueness of shortest paths.

context *finite-weighted-digraph*
begin

lemma *walk-nonempty*:
assumes *walk p*
shows *p* ≠ []

```
  using assms by (cases p) auto

lemma walk-weight-nonneg:
  assumes walk p
  shows 0 ≤ walk-weight p
  using assms
proof (induction p)
  case Nil
  then show ?case by simp
next
  case (Cons x ps)
  show ?case
  proof (cases ps)
    case Nil
    then show ?thesis by simp
  next
    case (Cons y ys)
    then have edge: (x, y) ∈ E and tail: walk (y # ys)
      using Cons.prems by auto
    have wx: 0 ≤ w x y
      using edge nonneg-weight by auto
    have tail-nonneg: 0 ≤ walk-weight (y # ys)
      using Cons.IH tail Cons by simp
    show ?thesis
      using wx tail-nonneg Cons by simp
  qed
qed

lemma dist-nonneg:
  assumes reachable a b
  shows 0 ≤ dist a b
proof −
  have dist a b ∈ simple-walk-weights a b
    by (rule dist-is-simple-walk-weight[OF assms])
  then obtain p where p: simple-walk-betw a p b
    and dist-eq: walk-weight p = dist a b
    unfolding simple-walk-weights-def by auto
  have walk p
    using p unfolding simple-walk-betw-def walk-betw-def by blast
  then have 0 ≤ walk-weight p
    by (rule walk-weight-nonneg)
  then show ?thesis
    using dist-eq by simp
qed

lemma walk-take-Suc:
  assumes walk p i < length p
  shows walk (take (Suc i) p)
  using assms
```

```
proof (induction p arbitrary: i)
  case Nil
  then show ?case by simp
next
  case (Cons x ps)
  show ?case
  proof (cases ps)
    case Nil
    then show ?thesis
      using Cons by (cases i) auto
  next
    case (Cons y ys)
    then have walk-tail: walk (y # ys)
      using Cons.prems by auto
    show ?thesis
    proof (cases i)
      case 0
      then show ?thesis
        using Cons.prems Cons by auto
    next
      case (Suc j)
      then have j < length (y # ys)
        using Cons.prems Cons by simp
      then have walk (take (Suc j) (y # ys))
        using Cons.IH walk-tail Cons by simp
      then show ?thesis
        using Cons.prems Cons Suc by auto
    qed
  qed
qed

lemma walk-weight-take-Suc-le:
  assumes walk p i < length p
  shows walk-weight (take (Suc i) p) ≤ walk-weight p
  using assms
proof (induction p arbitrary: i)
  case Nil
  then show ?case by simp
next
  case (Cons x ps)
  show ?case
  proof (cases ps)
    case Nil
    then show ?thesis
      using Cons by (cases i) auto
  next
    case (Cons y ys)
    then have edge: (x, y) ∈ E and walk-tail: walk (y # ys)
      using Cons.prems by auto
```

```
    show ?thesis
    proof (cases i)
      case 0
      have 0 ≤ w x y
        using edge nonneg-weight by auto
      moreover have 0 ≤ walk-weight (y # ys)
        using walk-weight-nonneg[OF walk-tail] .
      ultimately show ?thesis
        using Cons 0 by auto
    next
      case (Suc j)
      then have j: j < length (y # ys)
        using Cons.prems Cons by simp
      have walk-weight (take (Suc j) (y # ys)) ≤ walk-weight (y # ys)
        using Cons.IH walk-tail j Cons by simp
      then show ?thesis
        using Cons Suc by auto
    qed
  qed
qed

lemma simple-walk-prefix:
  assumes p: simple-walk-betw a p b
    and i: i < length p
  shows simple-walk-betw a (take (Suc i) p) (p ! i)
proof −
  have nonempty: p ≠ [] and hd: hd p = a and walk-p: walk p and distinct-p:
distinct p
    using p unfolding simple-walk-betw-def walk-betw-def by auto
  have take-nonempty: take (Suc i) p ≠ []
    using i by auto
  have hd (take (Suc i) p) = a
    using hd i by (simp add: hd-take)
  moreover have last (take (Suc i) p) = p ! i
  proof −
    have len-take: length (take (Suc i) p) = Suc i
      using i by simp
    have last (take (Suc i) p) =
        take (Suc i) p ! (length (take (Suc i) p) − 1)
      using take-nonempty by (rule last-conv-nth)
    also have ... = take (Suc i) p ! i
      using len-take by simp
    finally have last (take (Suc i) p) = take (Suc i) p ! i .
    then show ?thesis
      using i by simp
  qed
  moreover have walk (take (Suc i) p)
    using walk-take-Suc[OF walk-p i] .
  moreover have distinct (take (Suc i) p)
```

```
    using distinct-p by simp
  ultimately show ?thesis
    using take-nonempty unfolding simple-walk-betw-def walk-betw-def by auto
qed

lemma dist-le-simple-walk-weight:
  assumes p: simple-walk-betw a p b
  shows dist a b ≤ walk-weight p
proof −
  have mem: walk-weight p ∈ simple-walk-weights a b
    using p unfolding simple-walk-weights-def by blast
  have fin: finite (simple-walk-weights a b)
    using finite-simple-walk-weights .
  show ?thesis
    unfolding dist-def using fin mem by (rule Min-le)
qed

lemma shortest-walk-prefix-dist-le:
  assumes p: shortest-walk a p b
    and u: u ∈ set p
  shows dist a u ≤ dist a b
proof −
  have simple-p: simple-walk-betw a p b
    using p unfolding shortest-walk-def by blast
  then have walk-p: walk p
    unfolding simple-walk-betw-def walk-betw-def by auto
  obtain i where i: i < length p p ! i = u
    using u by (auto simp: in-set-conv-nth)
  let ?q = take (Suc i) p
  have simple-q: simple-walk-betw a ?q u
    using simple-walk-prefix[OF simple-p i(1)] i(2) by simp
  have dist a u ≤ walk-weight ?q
    using dist-le-simple-walk-weight[OF simple-q] .
  also have ... ≤ walk-weight p
    using walk-weight-take-Suc-le[OF walk-p i(1)] .
  also have ... = dist a b
    using p unfolding shortest-walk-def by simp
  finally show ?thesis .
qed

lemma through-witness-dist-le:
  assumes through S v
  obtains u p where u ∈ S shortest-walk s p v u ∈ set p dist s u ≤ dist s v
proof −
  obtain u p where u: u ∈ S shortest-walk s p v u ∈ set p
    using assms unfolding through-def by blast
  then have dist s u ≤ dist s v
    using shortest-walk-prefix-dist-le by blast
  then show thesis
```

```
    using that u by blast
qed

lemma walk-snoc:
  assumes walk-p: walk p
    and nonempty: p ≠ []
    and last-p: last p = u
    and edge: (u, v) ∈ E
  shows walk (p @ [v])
  using walk-p nonempty last-p edge
proof (induction p arbitrary: u)
  case Nil
  then show ?case by simp
next
  case (Cons x ps)
  show ?case
  proof (cases ps)
    case Nil
    then have xu: x = u
      using Cons.prems by simp
    have xv: (x, v) ∈ E
      using Cons.prems xu by simp
    have xV: x ∈ V
      using Cons.prems Nil by simp
    have vV: v ∈ V
      using edge edge-in-V by blast
    show ?thesis
      using xV vV xv Nil by auto
  next
    case (Cons y ys)
    have ps-walk: walk ps
      using Cons.prems Cons by auto
    have ps-nonempty: ps ≠ []
      using Cons by simp
    have ps-last: last ps = u
      using Cons.prems ps-nonempty by simp
    have tail-edge: (u, v) ∈ E
      using Cons.prems by simp
    have walk (ps @ [v])
      using Cons.IH[OF ps-walk ps-nonempty ps-last tail-edge] .
    then show ?thesis
      using Cons.prems Cons by auto
  qed
qed

lemma walk-weight-snoc:
  assumes p ≠ [] last p = u
  shows walk-weight (p @ [v]) = walk-weight p + w u v
  using assms
```

```
proof (induction p arbitrary: u)
  case Nil
  then show ?case by simp
next
  case (Cons x ps)
  show ?case
  proof (cases ps)
    case Nil
    then show ?thesis
      using Cons by simp
  next
    case (Cons y ys)
    have ps-ne: ps ≠ []
      using Cons by simp
    have ps-last: last ps = u
      using Cons.prems ps-ne by simp
    have walk-weight (ps @ [v]) = walk-weight ps + w u v
      using Cons.IH[OF ps-ne ps-last] .
    then show ?thesis
      using Cons by simp
  qed
qed

lemma walk-append-tl:
  assumes walk-p: walk p
    and walk-q: walk q
    and p-ne: p ≠ []
    and q-ne: q ≠ []
    and join: last p = hd q
  shows walk (p @ tl q)
  using walk-p walk-q p-ne q-ne join
proof (induction p arbitrary: q)
  case Nil
  then show ?case by simp
next
  case (Cons x ps)
  show ?case
  proof (cases ps)
    case Nil
    have q-ne': q ≠ []
      using Cons.prems by blast
    have walk-q': walk q
      using Cons.prems by blast
    have x = hd q
      using Cons.prems Nil by simp
    then have x # tl q = q
      using q-ne' by (cases q) auto
    then show ?thesis
      using walk-q' Nil by simp
```

```
  next
    case (Cons y ys)
    have tail-walk: walk (y # ys)
      using Cons.prems Cons by auto
    have tail-walk-ps: walk ps
      using tail-walk Cons by simp
    have tail-ne: y # ys ≠ []
      by simp
    have tail-ne-ps: ps ≠ []
      using Cons by simp
    have tail-join: last (y # ys) = hd q
      using Cons.prems Cons by simp
    have tail-join-ps: last ps = hd q
      using tail-join Cons by simp
    have walk-q': walk q
      using Cons.prems by blast
    have q-ne': q ≠ []
      using Cons.prems by blast
    have walk (ps @ tl q)
      using Cons.IH[OF tail-walk-ps walk-q' tail-ne-ps q-ne' tail-join-ps] .
    then have walk ((y # ys) @ tl q)
      using Cons by simp
    then show ?thesis
      using Cons.prems Cons by auto
  qed
qed

lemma walk-weight-append-tl:
  assumes p-ne: p ≠ []
    and q-ne: q ≠ []
    and join: last p = hd q
  shows walk-weight (p @ tl q) = walk-weight p + walk-weight q
  using p-ne q-ne join
proof (induction p arbitrary: q)
  case Nil
  then show ?case by simp
next
  case (Cons x ps)
  show ?case
  proof (cases ps)
    case Nil
    have q-ne': q ≠ []
      using Cons.prems by blast
    have x = hd q
      using Cons.prems Nil by simp
    then have x # tl q = q
      using q-ne' by (cases q) auto
    then show ?thesis
      using Nil by simp
```

```
  next
    case (Cons y ys)
    have tail-ne: y # ys ≠ []
      by simp
    have tail-ne-ps: ps ≠ []
      using Cons by simp
    have tail-join: last (y # ys) = hd q
      using Cons.prems Cons by simp
    have tail-join-ps: last ps = hd q
      using tail-join Cons by simp
    have q-ne': q ≠ []
      using Cons.prems by blast
    have walk-weight (ps @ tl q) = walk-weight ps + walk-weight q
      using Cons.IH[OF tail-ne-ps q-ne' tail-join-ps] .
    then have walk-weight ((y # ys) @ tl q) =
        walk-weight (y # ys) + walk-weight q
      using Cons by simp
    then show ?thesis
      using Cons by simp
  qed
qed

lemma walk-betw-append-tl:
  assumes p: walk-betw a p b
    and q: walk-betw b q c
  shows walk-betw a (p @ tl q) c
proof −
  have p-ne: p ≠ [] and q-ne: q ≠ []
    and hd-p: hd p = a and last-p: last p = b
    and hd-q: hd q = b and last-q: last q = c
    and walk-p: walk p and walk-q: walk q
    using p q unfolding walk-betw-def by auto
  have join: last p = hd q
    using last-p hd-q by simp
  have walk: walk (p @ tl q)
    using walk-append-tl[OF walk-p walk-q p-ne q-ne join] .
  have nonempty: p @ tl q ≠ []
    using p-ne by simp
  have hd: hd (p @ tl q) = a
    using hd-p p-ne by simp
  have last: last (p @ tl q) = c
  proof (cases tl q = [])
    case True
    then have q = [hd q]
      using q-ne by (cases q) auto
    then show ?thesis
      using last-p hd-q last-q by simp
  next
    case False
```

```
    then show ?thesis
      using last-q q-ne by (cases q) auto
  qed
  show ?thesis
    using nonempty hd last walk unfolding walk-betw-def by blast
qed

lemma walk-weight-append-tl-betw:
  assumes p: walk-betw a p b
    and q: walk-betw b q c
  shows walk-weight (p @ tl q) = walk-weight p + walk-weight q
proof −
  have p ≠ [] q ≠ [] last p = hd q
    using p q unfolding walk-betw-def by auto
  then show ?thesis
    by (rule walk-weight-append-tl)
qed

lemma walk-suffix-append:
  assumes walk (xs @ ys) ys ≠ []
  shows walk ys
  using assms
proof (induction xs)
  case Nil
  then show ?case by simp
next
  case (Cons x xs)
  show ?case
  proof (cases xs)
    case Nil
    then show ?thesis
      using Cons by (cases ys) auto
  next
    case (Cons y zs)
    then have tail-walk: walk ((y # zs) @ ys)
      using Cons.prems by auto
    have tail-walk-ps: walk (xs @ ys)
      using tail-walk Cons by simp
    then show ?thesis
      using Cons.IH[OF tail-walk-ps Cons.prems(2)] by blast
  qed
qed

lemma walk-prefix-append:
  assumes walk (xs @ ys) xs ≠ []
  shows walk xs
  using assms
proof (induction xs arbitrary: ys)
  case Nil
```

```
  then show ?case by simp
next
  case (Cons x xs)
  show ?case
  proof (cases xs)
    case Nil
    then have x ∈ V
      using Cons.prems by (cases ys) auto
    then show ?thesis
      using Nil by simp
  next
    case (Cons y zs)
    then have xV: x ∈ V and edge: (x, y) ∈ E and tail-walk: walk ((y # zs) @
ys)
      using Cons.prems by auto
    have tail-walk-ps: walk (xs @ ys)
      using tail-walk Cons by simp
    have tail-ne: xs ≠ []
      using Cons by simp
    have walk xs
      using Cons.IH[OF tail-walk-ps tail-ne] .
    then have walk (y # zs)
      using Cons by simp
    then show ?thesis
      using Cons xV edge by auto
  qed
qed

lemma walk-remove-cycle:
  assumes walk-p: walk (xs @ y # ys @ y # zs)
  shows walk (xs @ y # zs)
proof −
  let ?pref = xs @ [y]
  let ?cycle-suffix = y # ys @ y # zs
  let ?suffix = y # zs
  have pref-walk: walk ?pref
    using walk-prefix-append[of ?pref ys @ y # zs] walk-p by simp
  have suffix-walk: walk ?suffix
    using walk-suffix-append[of xs @ y # ys ?suffix] walk-p by simp
  have pref-ne: ?pref ≠ []
    by simp
  have suffix-ne: ?suffix ≠ []
    by simp
  have join: last ?pref = hd ?suffix
    by simp
  have walk (?pref @ tl ?suffix)
    using walk-append-tl[OF pref-walk suffix-walk pref-ne suffix-ne join] .
  then show ?thesis
    by simp
```

qed

lemma *walk-weight-remove-cycle-le*:
assumes *walk-p*: *walk* (*xs* @ *y* # *ys* @ *y* # *zs*)
shows *walk-weight* (*xs* @ *y* # *zs*) ≤ *walk-weight* (*xs* @ *y* # *ys* @ *y* # *zs*)
proof −
let *?pref* = *xs* @ [*y*]
let *?cycle* = *y* # *ys* @ [*y*]
let *?suffix* = *y* # *zs*
let *?cycle-suffix* = *y* # *ys* @ *y* # *zs*
have *pref-walk*: *walk ?pref*
using *walk-prefix-append*[*of ?pref ys* @ *y* # *zs*] *walk-p* **by** *simp*
have *cycle-suffix-walk*: *walk ?cycle-suffix*
using *walk-suffix-append*[*of xs ?cycle-suffix*] *walk-p* **by** *simp*
have *cycle-walk*: *walk ?cycle*
using *walk-prefix-append*[*of ?cycle zs*] *cycle-suffix-walk* **by** *simp*
have *suffix-walk*: *walk ?suffix*
using *walk-suffix-append*[*of xs* @ *y* # *ys ?suffix*] *walk-p* **by** *simp*
have *pref-ne*: *?pref* ≠ [] **and** *cycle-suffix-ne*: *?cycle-suffix* ≠ []
and *cycle-ne*: *?cycle* ≠ [] **and** *suffix-ne*: *?suffix* ≠ []
by *simp-all*
have *pref-join-cycle-suffix*: *last ?pref* = *hd ?cycle-suffix*
by *simp*
have *pref-join-suffix*: *last ?pref* = *hd ?suffix*
by *simp*
have *cycle-join-suffix*: *last ?cycle* = *hd ?suffix*
by *simp*
have *new-weight*:
walk-weight (*xs* @ *y* # *zs*) = *walk-weight ?pref* + *walk-weight ?suffix*
using *walk-weight-append-tl*[*OF pref-ne suffix-ne pref-join-suffix*] **by** *simp*
have *original-weight*:
walk-weight (*xs* @ *y* # *ys* @ *y* # *zs*) =
walk-weight ?pref + *walk-weight ?cycle-suffix*
using *walk-weight-append-tl*[*OF pref-ne cycle-suffix-ne pref-join-cycle-suffix*] **by**
simp
have *cycle-suffix-weight*:
walk-weight ?cycle-suffix = *walk-weight ?cycle* + *walk-weight ?suffix*
using *walk-weight-append-tl*[*OF cycle-ne suffix-ne cycle-join-suffix*] **by** *simp*
have *cycle-nonneg*: *0* ≤ *walk-weight ?cycle*
using *walk-weight-nonneg*[*OF cycle-walk*] **.**
show *?thesis*
using *new-weight original-weight cycle-suffix-weight cycle-nonneg* **by** *linarith*
qed

lemma *walk-betw-remove-cycle*:
assumes *p*: *walk-betw a* (*xs* @ *y* # *ys* @ *y* # *zs*) *b*
shows *walk-betw a* (*xs* @ *y* # *zs*) *b*
proof −
let *?p* = *xs* @ *y* # *ys* @ *y* # *zs*

```
  let ?p' = xs @ y # zs
  have walk-p: walk ?p
    using p unfolding walk-betw-def by blast
  have walk-p': walk ?p'
    using walk-remove-cycle[OF walk-p] .
  have nonempty: ?p' ≠ []
    by simp
  have hd-p: hd ?p = a and last-p: last ?p = b
    using p unfolding walk-betw-def by auto
  have hd-p': hd ?p' = a
    using hd-p by (cases xs) auto
  have last-p': last ?p' = b
    using last-p by (cases zs) auto
  show ?thesis
    using nonempty hd-p' last-p' walk-p' unfolding walk-betw-def by blast
qed

lemma walk-betw-to-simple-walk-le-exists:
  assumes p: walk-betw a p b
  shows ∃ q. simple-walk-betw a q b ∧ walk-weight q ≤ walk-weight p
  using p
proof (induction length p arbitrary: p rule: less-induct)
  case less
  show ?case
  proof (cases distinct p)
    case True
    then have simple-walk-betw a p b
      using less.prems unfolding simple-walk-betw-def by blast
    then show ?thesis
      by blast
  next
    case False
    then obtain xs ys zs y where decomp: p = xs @ [y] @ ys @ [y] @ zs
      using not-distinct-decomp by blast
    let ?p' = xs @ y # zs
    have p-form: p = xs @ y # ys @ y # zs
      using decomp by simp
    have p'-betw: walk-betw a ?p' b
      using walk-betw-remove-cycle[of a xs y ys zs b] less.prems p-form by simp
    have p'-shorter: length ?p' < length p
      using p-form by simp
    obtain q where q: simple-walk-betw a q b walk-weight q ≤ walk-weight ?p'
      using less.hyps[OF p'-shorter p'-betw] by blast
    have walk-weight ?p' ≤ walk-weight p
      using walk-weight-remove-cycle-le[of xs y ys zs] less.prems p-form
      unfolding walk-betw-def by simp
    then have walk-weight q ≤ walk-weight p
      using q(2) by linarith
    then show ?thesis
```

```
    using q(1) by blast
  qed
qed

lemma walk-betw-to-simple-walk-le:
  assumes p: walk-betw a p b
  obtains q where simple-walk-betw a q b walk-weight q ≤ walk-weight p
  using walk-betw-to-simple-walk-le-exists[OF p] that by blast

lemma walk-suffix-from-index:
  assumes p: simple-walk-betw a p b
    and i: i < length p
  shows walk-betw (p ! i) (drop i p) b
proof −
  have walk-p: walk p and last-p: last p = b
    using p unfolding simple-walk-betw-def walk-betw-def by auto
  have drop-ne: drop i p ≠ []
    using i by auto
  have hd-drop: hd (drop i p) = p ! i
    using i by (simp add: hd-drop-conv-nth)
  have last-drop: last (drop i p) = b
    using last-p i by (simp add: last-drop)
  have p-decomp: p = take i p @ drop i p
    by simp
  have walk-drop: walk (drop i p)
    using walk-suffix-append[of take i p drop i p] walk-p drop-ne p-decomp by simp
  show ?thesis
    using drop-ne hd-drop last-drop walk-drop unfolding walk-betw-def by blast
qed

lemma take-Suc-append-tl-drop:
  assumes i < length p
  shows take (Suc i) p @ tl (drop i p) = p
proof −
  have drop-i: drop i p = p ! i # drop (Suc i) p
    using Cons-nth-drop-Suc[OF assms] by simp
  have take (Suc i) p @ tl (drop i p) =
      take i p @ [p ! i] @ drop (Suc i) p
    using assms drop-i by (simp add: take-Suc-conv-app-nth)
  also have ... = take i p @ drop i p
    using drop-i by simp
  finally show ?thesis
    by simp
qed

lemma shortest-walk-prefix-shortest:
  assumes sp: shortest-walk a p b
    and i: i < length p
  shows shortest-walk a (take (Suc i) p) (p ! i)
```

proof −
let *?u* = *p* ! *i*
let *?pref* = *take* (*Suc i*) *p*
let *?suf* = *drop i p*
have *simple-p*: *simple-walk-betw a p b*
using *sp* **unfolding** *shortest-walk-def* **by** *blast*
have *weight-p*: *walk-weight p* = *dist a b*
using *sp* **unfolding** *shortest-walk-def* **by** *blast*
have *pref-simple*: *simple-walk-betw a ?pref ?u*
using *simple-walk-prefix*[*OF simple-p i*] **.**
have *pref-walk-betw*: *walk-betw a ?pref ?u*
using *pref-simple* **unfolding** *simple-walk-betw-def* **by** *blast*
have *suf-walk-betw*: *walk-betw ?u ?suf b*
using *walk-suffix-from-index*[*OF simple-p i*] **.**
have *split-p*: *?pref* @ *tl ?suf* = *p*
using *take-Suc-append-tl-drop*[*OF i*] **.**
have *weight-split*: *walk-weight p* = *walk-weight ?pref* + *walk-weight ?suf*
using *walk-weight-append-tl-betw*[*OF pref-walk-betw suf-walk-betw*] *split-p* **by** *simp*
have *dist-pref-le*: *dist a ?u* ≤ *walk-weight ?pref*
using *dist-le-simple-walk-weight*[*OF pref-simple*] **.**
have *no-strict*: ¬ *dist a ?u* < *walk-weight ?pref*
proof
assume *strict*: *dist a ?u* < *walk-weight ?pref*
have *reach-u*: *reachable a ?u*
using *pref-simple* **unfolding** *reachable-def* **by** *blast*
obtain *r* **where** *r-short*: *shortest-walk a r ?u*
using *shortest-walk-exists*[*OF reach-u*] **by** *blast*
have *r-simple*: *simple-walk-betw a r ?u*
using *r-short* **unfolding** *shortest-walk-def* **by** *blast*
have *r-walk-betw*: *walk-betw a r ?u*
using *r-simple* **unfolding** *simple-walk-betw-def* **by** *blast*
have *r-weight*: *walk-weight r* = *dist a ?u*
using *r-short* **unfolding** *shortest-walk-def* **by** *blast*
have *concat-walk*: *walk-betw a* (*r* @ *tl ?suf*) *b*
using *walk-betw-append-tl*[*OF r-walk-betw suf-walk-betw*] **.**
have *concat-weight*: *walk-weight* (*r* @ *tl ?suf*) = *walk-weight r* + *walk-weight ?suf*
using *walk-weight-append-tl-betw*[*OF r-walk-betw suf-walk-betw*] **.**
obtain *q* **where** *q*: *simple-walk-betw a q b*
walk-weight q ≤ *walk-weight* (*r* @ *tl ?suf*)
using *walk-betw-to-simple-walk-le*[*OF concat-walk*] **by** *blast*
have *walk-weight* (*r* @ *tl ?suf*) < *walk-weight p*
using *strict r-weight weight-split concat-weight* **by** *linarith*
then have *walk-weight q* < *dist a b*
using *q*(*2*) *weight-p* **by** *linarith*
moreover have *dist a b* ≤ *walk-weight q*
using *dist-le-simple-walk-weight*[*OF q*(*1*)] **.**
ultimately show *False*

```
    by linarith
  qed
  have walk-weight ?pref = dist a ?u
    using dist-pref-le no-strict by linarith
  then show ?thesis
    using pref-simple unfolding shortest-walk-def by blast
qed

lemma walk-nth-edge:
  assumes walk p
    and Suc i < length p
  shows (p ! i, p ! Suc i) ∈ E
  using assms
proof (induction p arbitrary: i)
  case Nil
  then show ?case by simp
next
  case (Cons x xs)
  show ?case
  proof (cases i)
    case 0
    then show ?thesis
      using Cons.prems by (cases xs) auto
  next
    case (Suc j)
    then have Suc j < length xs
      using Cons.prems by simp
    moreover have walk xs
      using Cons.prems Suc by (cases xs) auto
    ultimately have (xs ! j, xs ! Suc j) ∈ E
      using Cons.IH by blast
    then show ?thesis
      using Suc by simp
  qed
qed

lemma simple-walk-snoc:
  assumes p: simple-walk-betw s p u
    and edge: (u, v) ∈ E
    and fresh: v ∉ set p
  shows simple-walk-betw s (p @ [v]) v
proof −
  have nonempty: p ≠ [] and hd: hd p = s and last-p: last p = u
    and walk-p: walk p and distinct-p: distinct p
    using p unfolding simple-walk-betw-def walk-betw-def by auto
  have walk (p @ [v])
    using walk-snoc[OF walk-p nonempty last-p edge] .
  moreover have p @ [v] ≠ [] hd (p @ [v]) = s last (p @ [v]) = v distinct (p @ [v])
    using nonempty hd fresh distinct-p by auto
```

```
  ultimately show ?thesis
    unfolding simple-walk-betw-def walk-betw-def by auto
qed

lemma dist-triangle-edge:
  assumes edge: (u, v) ∈ E
    and reach-u: reachable s u
  shows reachable s v dist s v ≤ dist s u + w u v
proof −
  obtain p where p: shortest-walk s p u
    using reach-u by (rule shortest-walk-exists)
  have simple-p: simple-walk-betw s p u
    using p unfolding shortest-walk-def by blast
  have weight-p: walk-weight p = dist s u
    using p unfolding shortest-walk-def by blast
  show reach-v: reachable s v
  proof (cases v ∈ set p)
    case True
    then obtain i where i: i < length p p ! i = v
      by (auto simp: in-set-conv-nth)
    have simple-walk-betw s (take (Suc i) p) v
      using simple-walk-prefix[OF simple-p i(1)] i(2) by simp
    then show ?thesis
      unfolding reachable-def by blast
  next
    case False
    have simple-walk-betw s (p @ [v]) v
      using simple-walk-snoc[OF simple-p edge False] .
    then show ?thesis
      unfolding reachable-def by blast
  qed
  show dist s v ≤ dist s u + w u v
  proof (cases v ∈ set p)
    case True
    have dist s v ≤ dist s u
      using shortest-walk-prefix-dist-le[OF p True] .
    moreover have 0 ≤ w u v
      using edge nonneg-weight by auto
    ultimately show ?thesis
      by linarith
  next
    case False
    have simple-q: simple-walk-betw s (p @ [v]) v
      using simple-walk-snoc[OF simple-p edge False] .
    have dist s v ≤ walk-weight (p @ [v])
      using dist-le-simple-walk-weight[OF simple-q] .
    also have ... = dist s u + w u v
      using simple-p weight-p unfolding simple-walk-betw-def walk-betw-def
      by (simp add: walk-weight-snoc)
```

```
    finally show ?thesis .
  qed
qed

definition sound-label where
  sound-label d ⟷ (∀ v∈V. reachable s v ⟶ dist s v ≤ d v)

definition relax-edge where
  relax-edge d u v = (λx. if x = v then min (d v) (d u + w u v) else d x)

definition tight-edge-step where
  tight-edge-step u v ⟷
    (u, v) ∈ E ∧ reachable s u ∧ dist s v = dist s u + w u v

lemma relax-edge-le:
  relax-edge d u v x ≤ d x
  unfolding relax-edge-def by simp

lemma shortest-walk-successively-tight:
  assumes sp: shortest-walk s p v
  shows successively tight-edge-step p
  unfolding successively-conv-nth
proof clarify
  fix i
  assume i: Suc i < length p
  have simple-p: simple-walk-betw s p v
    using sp unfolding shortest-walk-def by blast
  have walk-p: walk p
    using simple-p unfolding simple-walk-betw-def walk-betw-def by blast
  have edge: (p ! i, p ! Suc i) ∈ E
    using walk-nth-edge[OF walk-p i] .
  have pref-i: shortest-walk s (take (Suc i) p) (p ! i)
    using shortest-walk-prefix-shortest[OF sp] i by simp
  have pref-Suc-i: shortest-walk s (take (Suc (Suc i)) p) (p ! Suc i)
    using shortest-walk-prefix-shortest[OF sp i] .
  have reach-i: reachable s (p ! i)
    using pref-i unfolding shortest-walk-def reachable-def by blast
  have pref-i-ne: take (Suc i) p ≠ []
    using i by (cases p) auto
  have last-pref-i: last (take (Suc i) p) = p ! i
  proof −
    have len-take: length (take (Suc i) p) = Suc i
      using i by simp
    have last (take (Suc i) p) = take (Suc i) p ! i
      using last-conv-nth[OF pref-i-ne] len-take by simp
    then show ?thesis
      using i by simp
  qed
  have take-next:
```

```
    take (Suc (Suc i)) p = take (Suc i) p @ [p ! Suc i]
    using i by (simp add: take-Suc-conv-app-nth)
  have weight-step:
    walk-weight (take (Suc (Suc i)) p) =
      walk-weight (take (Suc i) p) + w (p ! i) (p ! Suc i)
    using walk-weight-snoc[OF pref-i-ne last-pref-i, of p ! Suc i] take-next by simp
  have dist-i: walk-weight (take (Suc i) p) = dist s (p ! i)
    using pref-i unfolding shortest-walk-def by blast
  have dist-Suc-i: walk-weight (take (Suc (Suc i)) p) = dist s (p ! Suc i)
    using pref-Suc-i unfolding shortest-walk-def by blast
  show tight-edge-step (p ! i) (p ! Suc i)
    using edge reach-i weight-step dist-i dist-Suc-i
    unfolding tight-edge-step-def by simp
qed

lemma relax-edge-sound:
  assumes sound: sound-label d
    and edge: (u, v) ∈ E
    and reach-u: reachable s u
  shows sound-label (relax-edge d u v)
proof −
  have uV: u ∈ V
    using edge edge-in-V by blast
  have dist-u-le: dist s u ≤ d u
    using sound uV reach-u unfolding sound-label-def by blast
  have triangle: dist s v ≤ dist s u + w u v
    using dist-triangle-edge[OF edge reach-u] by blast
  show ?thesis
    unfolding sound-label-def relax-edge-def
  proof clarify
    fix x
    assume xV: x ∈ V and reach-x: reachable s x
    show dist s x ≤ (if x = v then min (d v) (d u + w u v) else d x)
    proof (cases x = v)
      case True
      have dist s v ≤ d v
        using sound xV reach-x True unfolding sound-label-def by blast
      moreover have dist s v ≤ d u + w u v
        using dist-u-le triangle by linarith
      ultimately show ?thesis
        using True by simp
    next
      case False
      then show ?thesis
      using sound xV reach-x unfolding sound-label-def by simp
    qed
  qed
qed
```

lemma *relax-tight-edge-completes*:
  assumes *sound*: *sound-label d*
    and *complete-u*: *d u* = *dist s u*
    and *tight*: *tight-edge-step u v*
  shows *sound-label* (*relax-edge d u v*)
    and *relax-edge d u v v* = *dist s v*
proof −
  have *edge*: (*u*, *v*) ∈ *E* **and** *reach-u*: *reachable s u*
    and *dist-v*: *dist s v* = *dist s u* + *w u v*
    using *tight* **unfolding** *tight-edge-step-def* **by** *auto*
  show *sound-label* (*relax-edge d u v*)
    using *relax-edge-sound*[*OF sound edge reach-u*] **.**
  have *vV*: *v* ∈ *V*
    using *edge edge-in-V* **by** *blast*
  have *reach-v*: *reachable s v*
    using *dist-triangle-edge*[*OF edge reach-u*] **by** *blast*
  have *dist s v* ≤ *d v*
    using *sound vV reach-v* **unfolding** *sound-label-def* **by** *blast*
  then show *relax-edge d u v v* = *dist s v*
    using *complete-u dist-v* **unfolding** *relax-edge-def* **by** *simp*
qed

fun *relax-path* **where**
  relax-path d [] = *d*
| *relax-path d* [*x*] = *d*
| *relax-path d* (*x* # *y* # *xs*) = *relax-path* (*relax-edge d x y*) (*y* # *xs*)

fun *relax-edges* **where**
  relax-edges d [] = *d*
| *relax-edges d* ((*u*, *v*) # *es*) = *relax-edges* (*relax-edge d u v*) *es*

fun *path-edges* **where**
  path-edges [] = []
| *path-edges* [*x*] = []
| *path-edges* (*x* # *y* # *xs*) = (*x*, *y*) # *path-edges* (*y* # *xs*)

lemma *relax-edges-append*:
  relax-edges d (*xs* @ *ys*) = *relax-edges* (*relax-edges d xs*) *ys*
  by (*induction xs arbitrary*: *d*) (*auto split*: *prod.splits*)

lemma *relax-edges-le*:
  relax-edges d es x ≤ *d x*
proof (*induction es arbitrary*: *d*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons e es*)
  obtain *u v* **where** *e*: *e* = (*u*, *v*)
    by *force*

```
  have relax-edges (relax-edge d u v) es x ≤ relax-edge d u v x
    by (rule Cons.IH)
  also have ... ≤ d x
    by (rule relax-edge-le)
  finally show ?case
    unfolding e by simp
qed

lemma relax-path-eq-relax-edges:
  relax-path d p = relax-edges d (path-edges p)
  by (induction p arbitrary: d rule: relax-path.induct) simp-all

lemma relax-edge-preserves-other:
  assumes x ≠ v
  shows relax-edge d u v x = d x
  using assms unfolding relax-edge-def by simp

lemma relax-edge-preserves-complete-other:
  assumes x ≠ v d x = dist s x
  shows relax-edge d u v x = dist s x
  using assms by (simp add: relax-edge-preserves-other)

lemma relax-edge-preserves-complete-sound:
  assumes sound: sound-label d
    and complete-x: d x = dist s x
    and edge: (u, v) ∈ E
    and reach-u: reachable s u
  shows relax-edge d u v x = dist s x
proof (cases x = v)
  case False
  then show ?thesis
    using complete-x by (simp add: relax-edge-preserves-other)
next
  case True
  have vV: v ∈ V
    using edge edge-in-V by blast
  have reach-v: reachable s v
    using dist-triangle-edge[OF edge reach-u] by blast
  have dist-v-le-dv: dist s v ≤ d v
    using sound vV reach-v unfolding sound-label-def by blast
  have dist-v-le-candidate: dist s v ≤ d u + w u v
  proof −
    have uV: u ∈ V
      using edge edge-in-V by blast
    have dist s u ≤ d u
      using sound uV reach-u unfolding sound-label-def by blast
    moreover have dist s v ≤ dist s u + w u v
      using dist-triangle-edge[OF edge reach-u] by blast
    ultimately show ?thesis
```

```
    by linarith
  qed
  show ?thesis
    using True complete-x dist-v-le-dv dist-v-le-candidate unfolding relax-edge-def
by simp
qed

lemma relax-edges-sound:
  assumes sound: sound-label d
    and edges: ⋀u v. (u, v) ∈ set es ⟹ (u, v) ∈ E
    and reaches: ⋀u v. (u, v) ∈ set es ⟹ reachable s u
  shows sound-label (relax-edges d es)
  using sound edges reaches
proof (induction es arbitrary: d)
  case Nil
  then show ?case by simp
next
  case (Cons e es)
  obtain u v where e: e = (u, v)
    by force
  have edge: (u, v) ∈ E
    using Cons.prems e by auto
  have reach: reachable s u
    using Cons.prems e by auto
  have sound1: sound-label (relax-edge d u v)
    using relax-edge-sound[OF Cons.prems(1) edge reach] .
  have edges-tail: ⋀a b. (a, b) ∈ set es ⟹ (a, b) ∈ E
    using Cons.prems by auto
  have reaches-tail: ⋀a b. (a, b) ∈ set es ⟹ reachable s a
    using Cons.prems by auto
  show ?case
    using Cons.IH[OF sound1 edges-tail reaches-tail] e by simp
qed

lemma relax-edges-preserves-complete-if-not-targeted:
  assumes complete-x: d x = dist s x
    and not-target: ⋀u. (u, x) ∉ set es
  shows relax-edges d es x = dist s x
  using complete-x not-target
proof (induction es arbitrary: d)
  case Nil
  then show ?case by simp
next
  case (Cons e es)
  obtain u v where e: e = (u, v)
    by force
  have x-ne-v: x ≠ v
    using Cons.prems(2)[of u] e by auto
  have complete-after: relax-edge d u v x = dist s x
```

```
    using x-ne-v Cons.prems(1) unfolding relax-edge-def by simp
  have not-target-tail: ⋀a. (a, x) ∉ set es
    using Cons.prems e by auto
  have relax-edges (relax-edge d u v) es x = dist s x
    using Cons.IH[of relax-edge d u v] complete-after not-target-tail by blast
  show ?case
    using ‹relax-edges (relax-edge d u v) es x = dist s x› e by simp
qed

lemma relax-edges-preserves-complete-sound:
  assumes sound: sound-label d
    and complete-x: d x = dist s x
    and edges: ⋀u v. (u, v) ∈ set es ⟹ (u, v) ∈ E
    and reaches: ⋀u v. (u, v) ∈ set es ⟹ reachable s u
  shows relax-edges d es x = dist s x
  using sound complete-x edges reaches
proof (induction es arbitrary: d)
  case Nil
  then show ?case by simp
next
  case (Cons e es)
  obtain u v where e: e = (u, v)
    by force
  have edge: (u, v) ∈ E
    using Cons.prems e by auto
  have reach: reachable s u
    using Cons.prems e by auto
  have sound1: sound-label (relax-edge d u v)
    using relax-edge-sound[OF Cons.prems(1) edge reach] .
  have complete-after: relax-edge d u v x = dist s x
    using relax-edge-preserves-complete-sound[OF Cons.prems(1) Cons.prems(2)
edge reach] .
  have edges-tail: ⋀a b. (a, b) ∈ set es ⟹ (a, b) ∈ E
    using Cons.prems by auto
  have reaches-tail: ⋀a b. (a, b) ∈ set es ⟹ reachable s a
    using Cons.prems by auto
  have relax-edges (relax-edge d u v) es x = dist s x
    using Cons.IH[OF sound1 complete-after edges-tail reaches-tail] .
  then show ?case
    using e by simp
qed

lemma relax-path-tight-sound-complete:
  assumes nonempty: p ≠ []
    and sound: sound-label d
    and complete-hd: d (hd p) = dist s (hd p)
    and tight: successively tight-edge-step p
  shows sound-label (relax-path d p)
    and relax-path d p (last p) = dist s (last p)
```

```
proof −
  have combined: ⋀d. p ≠ [] ⟹ sound-label d ⟹
      d (hd p) = dist s (hd p) ⟹ successively tight-edge-step p ⟹
      sound-label (relax-path d p) ∧
      relax-path d p (last p) = dist s (last p)
  proof (induction p)
    case Nil
    then show ?case by simp
  next
    case (Cons x ps)
    show ?case
    proof (cases ps = [])
      case True
      then show ?thesis
        using Cons.prems by simp
    next
      case False
      then obtain y ys where ps: ps = y # ys
        by (cases ps) auto
      have step: tight-edge-step x y
        using Cons.prems ps by (simp add: successively-Cons)
      have tight-tail: successively tight-edge-step (y # ys)
        using Cons.prems ps by (simp add: successively-Cons)
      let ?d1 = relax-edge d x y
      have complete-x: d x = dist s x
        using Cons.prems by simp
      have sound1: sound-label ?d1
        using relax-tight-edge-completes(1)[OF Cons.prems(2) complete-x step] .
      have complete-y: ?d1 y = dist s y
        using relax-tight-edge-completes(2)[OF Cons.prems(2) complete-x step] .
      have complete-hd-ps: ?d1 (hd ps) = dist s (hd ps)
        using complete-y ps by simp
      have tight-ps: successively tight-edge-step ps
        using tight-tail ps by simp
      have tail: sound-label (relax-path ?d1 ps) ∧
          relax-path ?d1 ps (last ps) = dist s (last ps)
        using Cons.IH[of ?d1] False sound1 complete-hd-ps tight-ps by blast
      then show ?thesis
        using ps by simp
    qed
  qed
  then show sound-label (relax-path d p)
    using assms by blast
  from combined show relax-path d p (last p) = dist s (last p)
    using assms by blast
qed

lemma relax-edges-with-tight-path-prefix-complete:
  assumes nonempty: p ≠ []
```

```
    and sound: sound-label d
    and complete-hd: d (hd p) = dist s (hd p)
    and tight: successively tight-edge-step p
    and es: es = path-edges p @ extra
    and extra-edges: ⋀u v. (u, v) ∈ set extra ⟹ (u, v) ∈ E
    and extra-reaches: ⋀u v. (u, v) ∈ set extra ⟹ reachable s u
  shows sound-label (relax-edges d es)
    and relax-edges d es (last p) = dist s (last p)
proof −
  let ?d1 = relax-edges d (path-edges p)
  have path-eq: ?d1 = relax-path d p
    using relax-path-eq-relax-edges[of d p] by simp
  have sound1: sound-label ?d1
    using relax-path-tight-sound-complete(1)[OF nonempty sound complete-hd tight]
path-eq by simp
  have complete-last: ?d1 (last p) = dist s (last p)
    using relax-path-tight-sound-complete(2)[OF nonempty sound complete-hd tight]
path-eq by simp
  have full: relax-edges d es = relax-edges ?d1 extra
    using es relax-edges-append[of d path-edges p extra] by simp
  show sound-label (relax-edges d es)
    using relax-edges-sound[OF sound1 extra-edges extra-reaches] full by simp
  show relax-edges d es (last p) = dist s (last p)
    using relax-edges-preserves-complete-sound[OF sound1 complete-last extra-edges
extra-reaches] full by simp
qed

end

end
theory BMSSP-Base-Case
  imports BMSSP-Algorithm-Correctness BMSSP-Shortest-Path-Lemmas
begin
```

11 Concrete Base Case

The algorithm's level-0 BMSSP call is a bounded Dijkstra-style search from a singleton source. It explores the bounded tree below the input bound and stops once the next extracted distance would exceed the first k vertices. Operationally, the paper describes this with a priority queue. For correctness, only one property of that queue matters here: vertices are extracted in nondecreasing true distance order.

This theory turns that observation into a compact semantic base case. Since the graph is finite, the bounded tree can be listed in sorted distance order. If the list has at most k vertices, the base case completes the whole bounded tree and returns the original bound. If the list is longer, the k-th distance becomes the returned finite bound and the completed set is exactly

the part of the bounded tree strictly below that bound. The final lemma packages this as a BMSSP postcondition without assuming any recursive postcondition for the base case itself.

context *finite-weighted-digraph*
begin

definition *min-dist-vertex* :: *'v set* ⇒ *'v* **where**
  min-dist-vertex A = (*SOME x*. *x* ∈ *A* ∧ *dist s x* = *Min* (*dist s* ‘ *A*))

lemma *finite-bound-tree* [*simp*]: *finite* (*bound-tree S B*)
  unfolding *bound-tree-def* **using** *finite-V* **by** *auto*

lemma *min-dist-vertex-prop*:
  assumes *finite A A* ≠ {}
  shows *min-dist-vertex A* ∈ *A* ∧ *dist s* (*min-dist-vertex A*) = *Min* (*dist s* ‘ *A*)
proof −
  obtain *a* **where** *a* ∈ *A*
    using *assms* **by** *auto*
  then have *Min* (*dist s* ‘ *A*) ∈ *dist s* ‘ *A*
    using *assms* **by** (*intro Min-in*) *auto*
  then have ∃ *x*. *x* ∈ *A* ∧ *dist s x* = *Min* (*dist s* ‘ *A*)
    by *auto*
  then have (*SOME x*. *x* ∈ *A* ∧ *dist s x* = *Min* (*dist s* ‘ *A*)) ∈ *A* ∧
      dist s (*SOME x*. *x* ∈ *A* ∧ *dist s x* = *Min* (*dist s* ‘ *A*)) = *Min* (*dist s* ‘ *A*)
    by (*rule someI-ex*)
  then show *?thesis*
    unfolding *min-dist-vertex-def* **.**
qed

lemma *min-dist-vertex-in*:
  assumes *finite A A* ≠ {}
  shows *min-dist-vertex A* ∈ *A*
  using *min-dist-vertex-prop*[*OF assms*] **by** *blast*

lemma *min-dist-vertex-min*:
  assumes *finite A A* ≠ {} *y* ∈ *A*
  shows *dist s* (*min-dist-vertex A*) ≤ *dist s y*
proof −
  have *dist s* (*min-dist-vertex A*) = *Min* (*dist s* ‘ *A*)
    using *min-dist-vertex-prop*[*OF assms*(*1*,*2*)] **by** *blast*
  also have ... ≤ *dist s y*
    using *assms* **by** *auto*
  finally show *?thesis* **.**
qed

lemma *ordered-extraction-exists*:
  assumes *finite A*
  shows ∃ *xs*. *set xs* = *A* ∧ *distinct xs* ∧ *sorted-wrt* (λ*u v*. *dist s u* ≤ *dist s v*) *xs*
proof −

```
  obtain xs where xs: set xs = A distinct xs
    using finite-distinct-list[OF assms] by blast
  let ?ys = sort-key (dist s) xs
  have set ?ys = A
    using xs(1) by simp
  moreover have distinct ?ys
    using xs(2) by simp
  moreover have sorted-wrt (λu v. dist s u ≤ dist s v) ?ys
  proof −
    have sorted (map (dist s) ?ys)
      by (rule sorted-sort-key)
    then show ?thesis
      by (simp add: sorted-map)
  qed
  ultimately show ?thesis
    by blast
qed

definition closest-vertices :: 'v set ⇒ 'v list where
  closest-vertices A =
    (SOME xs. set xs = A ∧ distinct xs ∧ sorted-wrt (λu v. dist s u ≤ dist s v) xs)

lemma closest-vertices-properties:
  assumes finite A
  shows set (closest-vertices A) = A
    and distinct (closest-vertices A)
    and sorted-wrt (λu v. dist s u ≤ dist s v) (closest-vertices A)
proof −
  have ∃xs. set xs = A ∧ distinct xs ∧
      sorted-wrt (λu v. dist s u ≤ dist s v) xs
    using ordered-extraction-exists[OF assms] .
  then have cv-props: set (closest-vertices A) = A ∧
     distinct (closest-vertices A) ∧
     sorted-wrt (λu v. dist s u ≤ dist s v) (closest-vertices A)
    unfolding closest-vertices-def by (rule someI-ex)
  then show set (closest-vertices A) = A
    by blast
  from cv-props show distinct (closest-vertices A)
    by blast
  from cv-props show sorted-wrt (λu v. dist s u ≤ dist s v) (closest-vertices A)
    by blast
qed

definition base-case-order :: 'v ⇒ bound ⇒ 'v list where
  base-case-order x B = closest-vertices (bound-tree {x} B)

definition base-case-vertices :: nat ⇒ 'v ⇒ bound ⇒ 'v set where
  base-case-vertices k x B =
    (let xs = base-case-order x B in
```

```
    if length xs ≤ k then set xs
    else {v ∈ set (take (Suc k) xs). dist s v < dist s (xs ! k)})

definition base-case-bound :: nat ⇒ 'v ⇒ bound ⇒ bound where
  base-case-bound k x B =
    (let xs = base-case-order x B in
      if length xs ≤ k then B else Fin (dist s (xs ! k)))

definition base-case-result :: nat ⇒ 'v ⇒ bound ⇒ bound × 'v set where
  base-case-result k x B =
    (base-case-bound k x B, base-case-vertices k x B)
```

The helper *closest-vertices* is a choice of a distinct list sorted by true distance. It is intentionally non-executable: the base-case proof is semantic, and later executable theories provide their own finite procedure. The public base-case result is described by *base-case-result*, whose components are *base-case-bound* and *base-case-vertices*. The strict inequality in *base-case-vertices* mirrors the main BMSSP convention for finite bounds.

```
lemma base-case-order-set:
  set (base-case-order x B) = bound-tree {x} B
  unfolding base-case-order-def
  using closest-vertices-properties(1)[OF finite-bound-tree] .

lemma base-case-order-sorted:
  sorted-wrt (λu v. dist s u ≤ dist s v) (base-case-order x B)
  unfolding base-case-order-def
  using closest-vertices-properties(3)[OF finite-bound-tree] .

lemma base-case-order-distinct:
  distinct (base-case-order x B)
  unfolding base-case-order-def
  using closest-vertices-properties(2)[OF finite-bound-tree] .

lemma in-set-take-dist-lt-nth:
  assumes sorted: sorted-wrt (λu v. dist s u ≤ dist s v) xs
    and y: y ∈ set xs
    and lt: dist s y < dist s (xs ! k)
    and k: k < length xs
  shows y ∈ set (take k xs)
proof −
  obtain i where i: i < length xs xs ! i = y
    using y by (auto simp: in-set-conv-nth)
  have i < k
  proof (rule ccontr)
    assume ¬ i < k
    then have k-le-i: k ≤ i
      by simp
    show False
    proof (cases k = i)
```

```
    case True
    then show ?thesis
      using lt i(2) by simp
  next
    case False
    with k-le-i have k < i
      by simp
    then have dist s (xs ! k) ≤ dist s (xs ! i)
      using sorted-wrt-nth-less[OF sorted, of k i] i(1) by simp
    then show ?thesis
      using lt i(2) by simp
  qed
 qed
 then have i < length (take k xs) take k xs ! i = y
  using i by auto
 then show ?thesis
  using nth-mem by metis
qed

lemma base-case-success:
 assumes length (base-case-order x B) ≤ k
 shows base-case-vertices k x B = bound-tree {x} B
 using assms base-case-order-set[of x B]
 unfolding base-case-vertices-def by (simp add: Let-def)

lemma below-bound-less-trans:
 assumes a < b below-bound b B
 shows below-bound a B
 using assms by (cases B) auto

lemma Fin-below-bound-le:
 assumes below-bound a B
 shows bound-le (Fin a) B
 using assms by (cases B) auto

lemma base-case-partial:
 assumes len: k < length (base-case-order x B)
 shows base-case-vertices k x B = bound-tree {x} (Fin (dist s ((base-case-order x
B) ! k)))
proof −
 let ?xs = base-case-order x B
 let ?b = dist s (?xs ! k)
 have set-xs: set ?xs = bound-tree {x} B
  using base-case-order-set[of x B] .
 have sorted: sorted-wrt (λu v. dist s u ≤ dist s v) ?xs
  using base-case-order-sorted[of x B] .
 have kth-in: ?xs ! k ∈ bound-tree {x} B
  using len set-xs nth-mem by metis
 show ?thesis
```

```
  proof
    show base-case-vertices k x B ⊆ bound-tree {x} (Fin ?b)
    proof
      fix v
      assume v ∈ base-case-vertices k x B
      moreover have base-case-vertices k x B =
          {v ∈ set (take (Suc k) ?xs). dist s v < ?b}
        using len unfolding base-case-vertices-def by (simp add: Let-def)
      ultimately have v: v ∈ set (take (Suc k) ?xs) dist s v < ?b
        by auto
      have v ∈ set ?xs
        using v(1) by (meson in-set-takeD)
      then have v ∈ bound-tree {x} B
        using set-xs by simp
      then show v ∈ bound-tree {x} (Fin ?b)
        using v(2) unfolding bound-tree-def by auto
    qed
  next
    show bound-tree {x} (Fin ?b) ⊆ base-case-vertices k x B
    proof
      fix v
      assume v: v ∈ bound-tree {x} (Fin ?b)
      then have lt: dist s v < ?b
        unfolding bound-tree-def by auto
      have kth-below: below-bound (dist s (?xs ! k)) B
        using kth-in unfolding bound-tree-def by auto
      have below-bound (dist s v) B
        using below-bound-less-trans[OF lt kth-below] .
      then have v ∈ bound-tree {x} B
        using v unfolding bound-tree-def by auto
      then have v ∈ set ?xs
        using set-xs by simp
      then have v ∈ set (take k ?xs)
        using in-set-take-dist-lt-nth[OF sorted - lt len] by blast
      then have v ∈ set (take (Suc k) ?xs)
        using set-take-subset-set-take[of k Suc k ?xs] by auto
      then show v ∈ base-case-vertices k x B
        using len lt unfolding base-case-vertices-def by (simp add: Let-def)
    qed
  qed
qed

lemma base-case-result-correct:
  assumes S = {x}
  shows case base-case-result k x B of (B', U) ⇒
    U = bound-tree S B' ∧ complete-on (λv. if v ∈ U then dist s v else d v) U
proof (cases length (base-case-order x B) ≤ k)
  case True
  then have U: base-case-vertices k x B = bound-tree {x} B
```

```
    using base-case-success by blast
  have B': base-case-bound k x B = B
    using True unfolding base-case-bound-def by (simp add: Let-def)
  have complete-on (λv. if v ∈ base-case-vertices k x B then dist s v else d v)
      (base-case-vertices k x B)
    unfolding complete-on-def by simp
  then show ?thesis
    using assms U B' unfolding base-case-result-def by simp
next
  case False
  then have len: k < length (base-case-order x B)
    by simp
  let ?B = Fin (dist s ((base-case-order x B) ! k))
  have U: base-case-vertices k x B = bound-tree {x} ?B
    using base-case-partial[OF len] .
  have B': base-case-bound k x B = ?B
    using False unfolding base-case-bound-def by (simp add: Let-def)
  have complete-on (λv. if v ∈ base-case-vertices k x B then dist s v else d v)
      (base-case-vertices k x B)
    unfolding complete-on-def by simp
  then show ?thesis
    using assms U B' unfolding base-case-result-def by simp
qed

lemma base-case-bound-le:
  bound-le (base-case-bound k x B) B
proof (cases length (base-case-order x B) ≤ k)
  case True
  then show ?thesis
    unfolding base-case-bound-def by (cases B) (simp-all add: Let-def)
next
  case False
  then have len: k < length (base-case-order x B)
    by simp
  let ?xs = base-case-order x B
  have ?xs ! k ∈ bound-tree {x} B
    using len base-case-order-set[of x B] nth-mem by metis
  then have below-bound (dist s (?xs ! k)) B
    unfolding bound-tree-def by auto
  then show ?thesis
    using False Fin-below-bound-le unfolding base-case-bound-def by (simp add:
Let-def)
qed

lemma base-case-result-bound-le:
  case base-case-result k x B of (B', U) ⇒ bound-le B' B
  unfolding base-case-result-def using base-case-bound-le by simp

lemma base-case-result-bmssp-post:
```

assumes $S = \{x\}$
shows *case base-case-result k x B of* $(B', U) \Rightarrow$
bmssp-post d S B $(\lambda v.$ *if* $v \in U$ *then dist s v else d v)* B' U
proof −
have *corr*: *case base-case-result k x B of* $(B', U) \Rightarrow$
$U =$ *bound-tree S* $B' \wedge$ *complete-on* $(\lambda v.$ *if* $v \in U$ *then dist s v else d v)* U
using *base-case-result-correct*[*OF assms*, **where** $k{=}k$ **and** $B{=}B$ **and** $d{=}d$] **.**
have *le*: *case base-case-result k x B of* $(B', U) \Rightarrow$ *bound-le* B' B
using *base-case-result-bound-le*[**where** $k{=}k$ **and** $x{=}x$ **and** $B{=}B$] **.**
show *?thesis*
using *corr le* **unfolding** *bmssp-post-def* **by** (*cases base-case-result k x B*) *auto*
qed

The proof splits on whether the sorted bounded tree has already fit inside the base-case budget. In the success case, *length* (*base-case-order ?x ?B*) $\leq$ *?k* $\Longrightarrow$ *base-case-vertices ?k ?x ?B* = *bound-tree* $\{?x\}$ *?B* says the returned set is the entire input tree. In the partial case, *?k* $<$ *length* (*base-case-order ?x ?B*) $\Longrightarrow$ *base-case-vertices ?k ?x ?B* = *bound-tree* $\{?x\}$ (*Fin* (*local.dist s* (*base-case-order ?x ?B* ! *?k*))) identifies the returned set with the same tree under the newly returned finite bound. Lemma *?S* = $\{?x\}$ $\Longrightarrow$ *case base-case-result ?k ?x ?B of* $(B', U) \Rightarrow$ *bmssp-post ?d ?S ?B* $(\lambda v.$ *if* $v \in U$ *then local.dist s v else ?d v)* B' U is the result consumed by the recursive relation: updating labels to *local.dist* on the returned set establishes the BMSSP postcondition.

lemma *base-case-label-sound*:
assumes *sound*: *sound-label d*
shows *sound-label* $(\lambda v.$ *if* $v \in$ *base-case-vertices k x B then dist s v else d v)*
using *sound* **unfolding** *sound-label-def* **by** *auto*

lemma *finite-base-case-vertices* [*simp*]:
finite (*base-case-vertices k x B*)
proof (*cases length* (*base-case-order x B*) $\leq k$)
case *True*
then show *?thesis*
using *base-case-success*[*of x B k*] **by** *simp*
next
case *False*
then show *?thesis*
unfolding *base-case-vertices-def* **by** (*simp add*: *Let-def*)
qed

lemma *card-base-case-vertices-le*:
card (*base-case-vertices k x B*) $\leq k$
proof (*cases length* (*base-case-order x B*) $\leq k$)
case *True*
have *card* (*base-case-vertices k x B*) = *length* (*base-case-order x B*)
using *True base-case-order-distinct*[*of x B*]
unfolding *base-case-vertices-def* **by** (*simp add*: *Let-def distinct-card*)

```
  then show ?thesis
    using True by simp
next
  case False
  then have len: k < length (base-case-order x B)
    by simp
  let ?xs = base-case-order x B
  let ?U = base-case-vertices k x B
  have U-subset: ?U ⊆ set (take k ?xs)
  proof
    fix v
    assume v ∈ ?U
    moreover have ?U = {v ∈ set (take (Suc k) ?xs). dist s v < dist s (?xs ! k)}
      using len unfolding base-case-vertices-def by (simp add: Let-def)
    ultimately have v: v ∈ set (take (Suc k) ?xs) dist s v < dist s (?xs ! k)
      by auto
    then have v-set: v ∈ set ?xs
      by (meson in-set-takeD)
    then show v ∈ set (take k ?xs)
      using in-set-take-dist-lt-nth[OF base-case-order-sorted[of x B] v-set v(2) len]
by blast
  qed
  have card-U-le: card ?U ≤ card (set (take k ?xs))
    using U-subset by (simp add: card-mono)
  have card-take-le: card (set (take k ?xs)) ≤ length (take k ?xs)
    by (rule card-length)
  have length (take k ?xs) = k
    using len by simp
  then show ?thesis
    using card-U-le card-take-le by linarith
qed
```

The remaining facts are bookkeeping for the costed and recursive layers. The base-case label update preserves *sound-label*, the returned set is finite, and *card* (*base-case-vertices ?k ?x ?B*) $\leq$ *?k* records the intended cap. The strict cutoff in the partial case is what makes this cardinality bound hold: ties at the k-th distance are not included.

```
end

end
theory BMSSP-Partition-Interface
  imports BMSSP-Correctness
begin
```

12 Partition Data-Structure Interface

The algorithm's non-base BMSSP loop depends on a partition data structure with *Insert*, *BatchPrepend*, and *Pull*. This theory records the mathematical

contract of those operations. It also gives a sorted reference implementation of the functional *Pull* contract. The separate concrete partition-state theory refines this view with explicit active entries, retained value memory, costed operations, and pull-to-split bridge theorems.

The interface is intentionally small. A partition view is just a finite set of keys with a real-valued label for each key. In the algorithm those labels are tentative distances, but the interface does not mention graphs. This separation is what lets the correctness proof talk about the mathematical effect of a pull, while the bucketed refinement later proves that a concrete representation implements the same effect with the desired costs.

Three design choices are worth making explicit. First, inserts are minimum-updates: inserting an existing key keeps the smaller label. Second, batch prepends are admissible only when every prepended label is below every remaining partition label. This is the abstract form of the "prepend a lower block" operation in the paper. Third, a pull does not merely return any small set. It returns a prefix of the partition order, a separating bound, and a residual view whose values are unchanged.

The sorted reference implementation at the end of the file is not the efficient implementation used for the final headline. It is a specification device. Sorting the keys by label makes the pull contract transparent, and the bucketed structure is later shown to realise the same contract while satisfying the paper-tight amortised bounds.

record *'k partition-view* =
  keys-of :: *'k set*
  value-of :: *'k* ⇒ *real*

definition *min-update* **where**
  min-update D x b =
    (| *keys-of* = *insert x* (*keys-of D*),
      value-of = (*value-of D*)(*x* := *if x* ∈ *keys-of D then min* (*value-of D x*) *b else b*) |)

definition *insert-spec* **where**
  insert-spec D x b D' ⟷
    keys-of D' = *insert x* (*keys-of D*) ∧
    value-of D' x = (*if x* ∈ *keys-of D then min* (*value-of D x*) *b else b*) ∧
    (∀ *y*. *y* ≠ *x* ⟶ *value-of D' y* = *value-of D y*)

definition *batch-min-update* **where**
  batch-min-update D xs = *fold* (λ(*x*, *b*) *D'*. *min-update D' x b*) *xs D*

definition *pull-separates* **where**
  pull-separates D M B S x D' ⟷
    S ⊆ *keys-of D* ∧
    card S ≤ *M* ∧
    keys-of D' = *keys-of D* − *S* ∧

value-of $D' =$ *value-of* $D \wedge$
$(\forall\, u{\in}S.\ \forall\, v{\in}$*keys-of* $D'.$ *value-of* $D\ u \leq$ *value-of* $D'\ v) \wedge$
(*if keys-of* $D' = \{\}$ *then* $x = B$
else $(\forall\, u{\in}S.$ *value-of* $D\ u < x) \wedge (\forall\, v{\in}$*keys-of* $D'.\ x \leq$ *value-of* $D'\ v)$)

definition *batch-prepend-admissible* **where**
batch-prepend-admissible $D\ xs \longleftrightarrow$
$(\forall\, (x,\ b){\in}$*set* $xs.\ \forall\, y{\in}$*keys-of* $D.\ b \leq$ *value-of* $D\ y)$

definition *partition-upper-bound* **where**
partition-upper-bound $D\ B \longleftrightarrow (\forall\, u{\in}$*keys-of* $D.$ *value-of* $D\ u < B)$

definition *partition-insert-cost-bound* **where**
partition-insert-cost-bound $c\ t \longleftrightarrow c \leq t$

definition *partition-batch-cost-bound* **where**
partition-batch-cost-bound $c\ t\ xs \longleftrightarrow c \leq t * $ *length* xs

definition *partition-pull-cost-bound* **where**
partition-pull-cost-bound $c\ S \longleftrightarrow c \leq$ *card* S

The three cost predicates are deliberately parametric and algebraic. They do not prescribe an implementation, only the local budget that an implementation must provide to plug into the runtime recurrence. The sorted reference pull has a linear cost, while the bucketed implementation later supplies the stronger logarithmic and amortised bounds required by the paper.

The summation lemmas below are the interface's first payoff: once individual operations satisfy their local predicates, loop-level proofs can sum costs by list induction without opening the implementation of the data structure.

lemma *partition-insert-costs-sum-bound*:
assumes *list-all* $(\lambda c.$ *partition-insert-cost-bound* $c\ t)$ *cs*
shows *sum-list cs* $\leq t *$ *length cs*
using *assms*
proof (*induction cs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons c cs*)
then have *c*: $c \leq t$
unfolding *partition-insert-cost-bound-def* **by** *simp*
have *tail*: *sum-list cs* $\leq t *$ *length cs*
using *Cons.IH Cons.prems* **by** *simp*
show *?case*
using *c tail* **by** *simp*
qed

lemma *partition-batch-costs-sum-bound*:

```
  assumes list-all2 (λc xs. partition-batch-cost-bound c t xs) cs xss
  shows sum-list cs ≤ t * sum-list (map length xss)
  using assms
proof (induction cs arbitrary: xss)
  case Nil
  then show ?case
    by (cases xss) simp-all
next
  case (Cons c cs)
  then obtain xs xss' where xss: xss = xs # xss'
    and c: c ≤ t * length xs
    and rest: list-all2 (λc xs. partition-batch-cost-bound c t xs) cs xss'
    by (cases xss) (auto simp: partition-batch-cost-bound-def)
  have tail: sum-list cs ≤ t * sum-list (map length xss')
    using Cons.IH[OF rest] .
  show ?case
  proof −
    have c + sum-list cs ≤
        t * length xs + t * sum-list (map length xss')
      using c tail by simp
    also have ... = t * (length xs + sum-list (map length xss'))
      by (simp add: algebra-simps)
    finally show ?thesis
      unfolding xss by simp
  qed
qed

lemma partition-pull-costs-sum-bound:
  assumes list-all2 (λc S. partition-pull-cost-bound c S) cs Ss
  shows sum-list cs ≤ sum-list (map card Ss)
  using assms
proof (induction cs arbitrary: Ss)
  case Nil
  then show ?case
    by (cases Ss) simp-all
next
  case (Cons c cs)
  then obtain S Ss' where Ss: Ss = S # Ss'
    and c: c ≤ card S
    and rest: list-all2 (λc S. partition-pull-cost-bound c S) cs Ss'
    by (cases Ss) (auto simp: partition-pull-cost-bound-def)
  have tail: sum-list cs ≤ sum-list (map card Ss')
    using Cons.IH[OF rest] .
  show ?case
    using c tail unfolding Ss by simp
qed

lemma partition-batch-cost-bound-append:
  assumes xs: partition-batch-cost-bound c-x t xs
```

```
    and ys: partition-batch-cost-bound c-y h ys
    and h-le-t: h ≤ t
  shows partition-batch-cost-bound (c-x + c-y) t (xs @ ys)
proof −
  have c-x: c-x ≤ t * length xs
    using xs unfolding partition-batch-cost-bound-def by simp
  have c-y: c-y ≤ h * length ys
    using ys unfolding partition-batch-cost-bound-def by simp
  have h * length ys ≤ t * length ys
    using h-le-t by simp
  then have c-x + c-y ≤ t * length xs + t * length ys
    using c-x c-y by linarith
  also have ... = t * length (xs @ ys)
    by (simp add: algebra-simps)
  finally show ?thesis
    unfolding partition-batch-cost-bound-def .
qed

lemma batch-prepend-admissibleI:
  assumes ⋀x b y. ⟦(x, b) ∈ set xs; y ∈ keys-of D⟧ ⟹
    b ≤ value-of D y
  shows batch-prepend-admissible D xs
  using assms unfolding batch-prepend-admissible-def by auto

lemma batch-prepend-admissible-below-all:
  assumes vals: ⋀x b. (x, b) ∈ set xs ⟹ b ≤ L
    and lower: ⋀y. y ∈ keys-of D ⟹ L ≤ value-of D y
  shows batch-prepend-admissible D xs
proof (rule batch-prepend-admissibleI)
  fix x b y
  assume xb: (x, b) ∈ set xs and y: y ∈ keys-of D
  have b ≤ L
    by (rule vals[OF xb])
  moreover have L ≤ value-of D y
    by (rule lower[OF y])
  then show b ≤ value-of D y
    using calculation by linarith
qed

lemma sorted-value-less-nth-in-take:
  fixes f :: 'a ⇒ 'b::linorder
  assumes sorted: sorted-wrt (λu v. f u ≤ f v) xs
    and y: y ∈ set xs
    and lt: f y < f (xs ! k)
    and k: k < length xs
  shows y ∈ set (take k xs)
proof −
  obtain i where i: i < length xs xs ! i = y
    using y by (auto simp: in-set-conv-nth)
```

```
  have i < k
  proof (rule ccontr)
    assume ¬ i < k
    then have k-le-i: k ≤ i
      by simp
    show False
    proof (cases k = i)
      case True
      have f y < f y
        using lt i(2) True by simp
      then show ?thesis
        using less-irrefl by blast
    next
      case False
      with k-le-i have k < i
        by simp
      then have f (xs ! k) ≤ f (xs ! i)
        using sorted-wrt-nth-less[OF sorted, of k i] i(1) by simp
      then show ?thesis
        using lt i(2) by simp
    qed
  qed
  then have i < length (take k xs) take k xs ! i = y
    using i by auto
  then show ?thesis
    using nth-mem by metis
qed
```

The sorted reference view chooses one canonical order of the partition keys: any distinct list sorted by *value-of*. Isabelle's Hilbert choice is used only to avoid committing to a concrete sorting algorithm in the specification. The theorem *partition-key-order-properties* immediately recovers the facts needed by later proofs: the order contains exactly the keys, has no duplicates, and is sorted by label.

The reference pull then takes the first block of that order. If at most M keys remain, the pull consumes everything and returns the outer bound. Otherwise it uses the M-th key as the separating bound and pulls exactly the keys with smaller labels. This strict comparison matches the BMSSP lower split used by the correctness proof.

```
definition partition-key-order where
  partition-key-order D =
    (SOME xs. set xs = keys-of D ∧ distinct xs ∧
      sorted-wrt (λu v. value-of D u ≤ value-of D v) xs)

definition sorted-pull-cost where
  sorted-pull-cost D = length (partition-key-order D)

lemma partition-key-order-properties:
```

```
  assumes finite-keys: finite (keys-of D)
  shows set (partition-key-order D) = keys-of D
    and distinct (partition-key-order D)
    and sorted-wrt (λu v. value-of D u ≤ value-of D v) (partition-key-order D)
proof −
  obtain xs where xs: set xs = keys-of D distinct xs
    using finite-distinct-list[OF finite-keys] by blast
  let ?ys = sort-key (value-of D) xs
  have exists:
    ∃ ys. set ys = keys-of D ∧ distinct ys ∧
      sorted-wrt (λu v. value-of D u ≤ value-of D v) ys
  proof (intro exI[of - ?ys] conjI)
    show set ?ys = keys-of D
      using xs by simp
    show distinct ?ys
      using xs by simp
    show sorted-wrt (λu v. value-of D u ≤ value-of D v) ?ys
    proof −
      have sorted (map (value-of D) ?ys)
        by (rule sorted-sort-key)
      then show ?thesis
        by (simp add: sorted-map)
    qed
  qed
  have props: set (partition-key-order D) = keys-of D ∧
      distinct (partition-key-order D) ∧
      sorted-wrt (λu v. value-of D u ≤ value-of D v) (partition-key-order D)
    unfolding partition-key-order-def by (rule someI-ex[OF exists])
  then show set (partition-key-order D) = keys-of D
    by blast
  from props show distinct (partition-key-order D)
    by blast
  from props show sorted-wrt (λu v. value-of D u ≤ value-of D v) (partition-key-
order D)
    by blast
qed

definition sorted-pull-set where
  sorted-pull-set M D =
    (let xs = partition-key-order D in
      if length xs ≤ M then set xs
      else {u ∈ keys-of D. value-of D u < value-of D (xs ! M)})

definition sorted-pull-bound where
  sorted-pull-bound M B D =
    (let xs = partition-key-order D in
      if length xs ≤ M then B else value-of D (xs ! M))

definition remove-keys-view where
```

```
  remove-keys-view D S =
    (| keys-of = keys-of D − S, value-of = value-of D |)

lemma min-update-keys [simp]:
  keys-of (min-update D x b) = insert x (keys-of D)
  unfolding min-update-def by simp

lemma min-update-value-same:
  assumes y ≠ x
  shows value-of (min-update D x b) y = value-of D y
  using assms unfolding min-update-def by simp

lemma min-update-value-key-new:
  assumes x ∉ keys-of D
  shows value-of (min-update D x b) x = b
  using assms unfolding min-update-def by simp

lemma min-update-value-key-old:
  assumes x ∈ keys-of D
  shows value-of (min-update D x b) x = min (value-of D x) b
  using assms unfolding min-update-def by simp

lemma min-update-value-le-old:
  assumes x ∈ keys-of D
  shows value-of (min-update D y b) x ≤ value-of D x
proof (cases x = y)
  case True
  then show ?thesis
    using assms unfolding min-update-def by simp
next
  case False
  then show ?thesis
    using min-update-value-same[of x y D b] by simp
qed

lemma min-update-value-le-inserted:
  value-of (min-update D x b) x ≤ b
  unfolding min-update-def by simp

theorem min-update-insert-spec:
  insert-spec D x b (min-update D x b)
  unfolding insert-spec-def min-update-def by auto

lemma min-update-preserves-upper-bound:
  assumes upper: partition-upper-bound D B
    and b-lt: b < B
  shows partition-upper-bound (min-update D x b) B
proof −
  have ⋀u. u ∈ keys-of (min-update D x b) ⟹
```

```
    value-of (min-update D x b) u < B
  proof −
    fix u
    assume u: u ∈ keys-of (min-update D x b)
    show value-of (min-update D x b) u < B
    proof (cases u = x)
      case True
      then show ?thesis
      proof (cases x ∈ keys-of D)
        case True
        then have min (value-of D x) b < B
          using upper b-lt unfolding partition-upper-bound-def by simp
        then show ?thesis
          using ‹u = x› True unfolding min-update-def by simp
      next
        case False
        then show ?thesis
          using ‹u = x› b-lt unfolding min-update-def by simp
      qed
    next
      case False
      then have u ∈ keys-of D
        using u unfolding min-update-def by simp
      then have value-of D u < B
        using upper unfolding partition-upper-bound-def by blast
      then show ?thesis
        using False min-update-value-same[of u x D b] by simp
    qed
  qed
  then show ?thesis
    unfolding partition-upper-bound-def by blast
qed

lemma batch-min-update-keys-subset:
  keys-of (batch-min-update D xs) ⊆ keys-of D ∪ fst ‘ set xs
proof (induction xs arbitrary: D)
  case Nil
  then show ?case
    unfolding batch-min-update-def by simp
next
  case (Cons xb xs)
  obtain x b where xb: xb = (x, b)
    by force
  have keys-of (batch-min-update (min-update D x b) xs)
      ⊆ keys-of (min-update D x b) ∪ fst ‘ set xs
    using Cons.IH[of min-update D x b] .
  also have ... ⊆ keys-of D ∪ fst ‘ set (xb # xs)
    using xb by auto
  finally show ?case
```

```
    unfolding batch-min-update-def xb by simp
qed

lemma batch-min-update-keys:
  keys-of (batch-min-update D xs) = keys-of D ∪ fst ‘ set xs
proof (induction xs arbitrary: D)
  case Nil
  then show ?case
    unfolding batch-min-update-def by simp
next
  case (Cons xb xs)
  obtain x b where xb: xb = (x, b)
    by force
  have keys-of (batch-min-update (min-update D x b) xs) =
      keys-of (min-update D x b) ∪ fst ‘ set xs
    using Cons.IH[of min-update D x b] .
  then show ?case
    unfolding batch-min-update-def xb by auto
qed

lemma batch-min-update-value-le-old:
  assumes x ∈ keys-of D
  shows value-of (batch-min-update D xs) x ≤ value-of D x
  using assms
proof (induction xs arbitrary: D)
  case Nil
  then show ?case
    unfolding batch-min-update-def by simp
next
  case (Cons xb xs)
  obtain y b where xb: xb = (y, b)
    by force
  have x-in-step: x ∈ keys-of (min-update D y b)
    using Cons.prems by simp
  have tail:
    value-of (batch-min-update (min-update D y b) xs) x ≤
      value-of (min-update D y b) x
    by (rule Cons.IH[OF x-in-step])
  have step: value-of (min-update D y b) x ≤ value-of D x
    by (rule min-update-value-le-old[OF Cons.prems])
  have tail-fold:
    value-of (fold (λ(x, b) D'. min-update D' x b) xs (min-update D y b)) x ≤
      value-of (min-update D y b) x
    using tail unfolding batch-min-update-def .
  show ?case
    unfolding batch-min-update-def xb
    using tail-fold step by (simp add: case-prod-beta)
qed
```

```
lemma batch-min-update-value-le-pair:
  assumes (x, b) ∈ set xs
  shows value-of (batch-min-update D xs) x ≤ b
  using assms
proof (induction xs arbitrary: D)
  case Nil
  then show ?case by simp
next
  case (Cons xb xs)
  obtain y c where xb: xb = (y, c)
    by force
  show ?case
  proof (cases (x, b) = (y, c))
    case True
    have x-in-step: x ∈ keys-of (min-update D y c)
      using True by simp
    have tail:
      value-of (batch-min-update (min-update D y c) xs) x ≤
        value-of (min-update D y c) x
      by (rule batch-min-update-value-le-old[OF x-in-step])
    have step: value-of (min-update D y c) x ≤ b
      using True min-update-value-le-inserted[of D y c] by simp
    have tail-fold:
      value-of (fold (λ(x, b) D'. min-update D' x b) xs (min-update D y c)) x ≤
        value-of (min-update D y c) x
      using tail unfolding batch-min-update-def .
    show ?thesis
      unfolding batch-min-update-def xb
      using tail-fold step by (simp add: case-prod-beta)
  next
    case False
    then have (x, b) ∈ set xs
      using Cons.prems xb by auto
    then have tail:
      value-of (batch-min-update (min-update D y c) xs) x ≤ b
      by (rule Cons.IH)
    have tail-fold:
      value-of (fold (λ(x, b) D'. min-update D' x b) xs (min-update D y c)) x ≤ b
      using tail unfolding batch-min-update-def .
    show ?thesis
      unfolding batch-min-update-def xb
      using tail-fold by (simp add: case-prod-beta)
  qed
qed

lemma sorted-pull-cost-linear:
  assumes finite-keys: finite (keys-of D)
  shows sorted-pull-cost D = card (keys-of D)
proof −
```

```
  have length (partition-key-order D) = card (set (partition-key-order D))
    using partition-key-order-properties(2)[OF finite-keys] by (simp add: distinct-
card)
  also have ... = card (keys-of D)
    using partition-key-order-properties(1)[OF finite-keys] by simp
  finally show ?thesis
    unfolding sorted-pull-cost-def .
qed
```

The minimum-update lemmas establish the functional behaviour of inserts and batches. They are intentionally elementary: keys are added, old values are never increased, inserted values are respected, and an upper bound is preserved when every inserted value is below that bound. These facts are used by the operational refresh step after a recursive child call has returned.

```
lemma batch-min-update-preserves-upper-bound:
  assumes upper: partition-upper-bound D B
    and values-lt: ⋀x b. (x, b) ∈ set xs ⟹ b < B
  shows partition-upper-bound (batch-min-update D xs) B
  using assms
proof (induction xs arbitrary: D)
  case Nil
  then show ?case
    unfolding batch-min-update-def by simp
next
  case (Cons xb xs)
  obtain x b where xb: xb = (x, b)
    by force
  have b-lt: b < B
    using Cons.prems(2)[of x b] xb by simp
  have upper-step: partition-upper-bound (min-update D x b) B
    using min-update-preserves-upper-bound[OF Cons.prems(1) b-lt] .
  have tail-lt: ⋀y c. (y, c) ∈ set xs ⟹ c < B
  proof −
    fix y c
    assume (y, c) ∈ set xs
    then show c < B
      using Cons.prems(2)[of y c] by simp
  qed
  have ih: partition-upper-bound (batch-min-update (min-update D x b) xs) B
    by (rule Cons.IH[OF upper-step tail-lt])
  then show ?case
    unfolding batch-min-update-def xb by (simp add: case-prod-beta)
qed

lemma pull-separates-pulled-smallest:
  assumes pull-separates D M B S x D'
    and u ∈ S
    and v ∈ keys-of D'
  shows value-of D u ≤ value-of D' v
```

using *assms* **unfolding** *pull-separates-def* **by** *blast*

lemma *pull-separates-remaining-keys*:
assumes *pull-separates D M B S x D′*
shows *keys-of D′ = keys-of D − S*
using *assms* **unfolding** *pull-separates-def* **by** *blast*

lemma *pull-separates-card*:
assumes *pull-separates D M B S x D′*
shows *card S ≤ M*
using *assms* **unfolding** *pull-separates-def* **by** *blast*

lemma *pull-separates-subset*:
assumes *pull-separates D M B S x D′*
shows *S ⊆ keys-of D*
using *assms* **unfolding** *pull-separates-def* **by** *blast*

lemma *pull-separates-preserves-upper-bound*:
assumes *pull*: *pull-separates D M B0 S x D′*
and *upper*: *partition-upper-bound D B*
shows *partition-upper-bound D′ B*
using *pull upper* **unfolding** *pull-separates-def partition-upper-bound-def* **by** *auto*

The pull lemmas unpack the abstract separator contract. A successful pull gives a source set of size at most M, a residual key set with the pulled keys removed, and a bound that lies between the pulled prefix and the residual suffix. The admissibility lemma for batch prepend is the property needed by the BMSSP loop: if every new batch value is below the separator, then the batch may be prepended in front of the residual partition.

lemma *pull-separates-empty-bound*:
assumes *pull-separates D M B S x D′*
and *keys-of D′ = {}*
shows *x = B*
using *assms* **unfolding** *pull-separates-def* **by** *simp*

lemma *pull-separates-nonempty-bound*:
assumes *pull-separates D M B S x D′*
and *keys-of D′ ≠ {}*
and *v ∈ keys-of D′*
shows *x ≤ value-of D′ v*
using *assms* **unfolding** *pull-separates-def* **by** *auto*

lemma *pull-separates-bound-le*:
assumes *pull*: *pull-separates D M B S x D′*
and *upper*: *partition-upper-bound D B*
shows *x ≤ B*
proof (*cases keys-of D′ = {}*)
case *True*

```
  then show ?thesis
    using pull-separates-empty-bound[OF pull True] by simp
next
  case False
  then obtain v where v: v ∈ keys-of D'
    by blast
  have x-le-v: x ≤ value-of D' v
    by (rule pull-separates-nonempty-bound[OF pull False v])
  have v-old: v ∈ keys-of D
    using pull v unfolding pull-separates-def by auto
  have val-eq: value-of D' = value-of D
    using pull unfolding pull-separates-def by blast
  have value-of D v < B
    using upper v-old unfolding partition-upper-bound-def by blast
  then show ?thesis
    using x-le-v val-eq by auto
qed

lemma pull-separates-batch-prepend-admissible:
  assumes pull: pull-separates D M B S x D'
    and vals: ⋀u b. (u, b) ∈ set xs ⟹ b ≤ x
  shows batch-prepend-admissible D' xs
proof (cases keys-of D' = {})
  case True
  then show ?thesis
    unfolding batch-prepend-admissible-def by simp
next
  case False
  show ?thesis
  proof (rule batch-prepend-admissible-below-all[OF vals])
    fix y
    assume y ∈ keys-of D'
    then show x ≤ value-of D' y
      by (rule pull-separates-nonempty-bound[OF pull False])
  qed
qed

theorem pull-separates-exact-split:
  assumes pull: pull-separates D M B S x D'
    and upper: ⋀u. u ∈ keys-of D ⟹ value-of D u < B
  shows S = {u ∈ keys-of D. value-of D u < x}
proof −
  have S-subset: S ⊆ keys-of D
    using pull-separates-subset[OF pull] .
  have D'-keys: keys-of D' = keys-of D − S
    using pull unfolding pull-separates-def by blast
  show ?thesis
  proof (cases keys-of D' = {})
    case True
```

```
    have x-B: x = B
      using pull True by (rule pull-separates-empty-bound)
    have S-keys: S = keys-of D
      using S-subset D'-keys True by blast
    have {u ∈ keys-of D. value-of D u < x} = keys-of D
      using upper unfolding x-B by blast
    then show ?thesis
      using S-keys by simp
  next
    case False
    have S-to-split: S ⊆ {u ∈ keys-of D. value-of D u < x}
    proof
      fix u
      assume uS: u ∈ S
      then have value-of D u < x
        using pull False unfolding pull-separates-def by auto
      then show u ∈ {u ∈ keys-of D. value-of D u < x}
        using uS S-subset by blast
    qed
    have split-to-S: {u ∈ keys-of D. value-of D u < x} ⊆ S
    proof
      fix u
      assume u: u ∈ {u ∈ keys-of D. value-of D u < x}
      show u ∈ S
      proof (rule ccontr)
        assume u ∉ S
        then have uD': u ∈ keys-of D'
          using u D'-keys by blast
        have x ≤ value-of D' u
          by (rule pull-separates-nonempty-bound[OF pull False uD'])
        moreover have value-of D' u = value-of D u
          using pull unfolding pull-separates-def by simp
        ultimately show False
          using u by simp
      qed
    qed
    show ?thesis
      using S-to-split split-to-S by blast
  qed
qed
```

The final theorems connect the sorted reference operation to the abstract pull contract. *sorted-pull-separates* proves that the reference operation really returns a separating prefix, and *sorted-pull-exact-split* proves that the same prefix is exactly the strict lower split at the returned bound whenever all labels are below the outer bound.

These two facts are specification tools. The executable bucketed partition does not sort all keys on every pull; instead, it proves the same separator and split properties for its bucket state and then plugs those properties into

the generic BMSSP correctness and cost layers.

```
theorem sorted-pull-separates:
  fixes M :: nat
    and B :: real
  assumes finite-keys: finite (keys-of D)
    and S-def: S = sorted-pull-set M D
    and x-def: x = sorted-pull-bound M B D
    and D'-def: D' = remove-keys-view D S
  shows pull-separates D M B S x D'
proof −
  let ?xs = partition-key-order D
  have set-xs: set ?xs = keys-of D
    using partition-key-order-properties(1)[OF finite-keys] .
  have distinct-xs: distinct ?xs
    using partition-key-order-properties(2)[OF finite-keys] .
  have sorted-xs: sorted-wrt (λu v. value-of D u ≤ value-of D v) ?xs
    using partition-key-order-properties(3)[OF finite-keys] .
  show ?thesis
  proof (cases length ?xs ≤ M)
    case True
    have S-all: S = keys-of D
      using True set-xs unfolding S-def sorted-pull-set-def by (simp add: Let-def)
    have D-empty: keys-of D' = {}
      unfolding D'-def remove-keys-view-def S-all by simp
    have x = B
      using True unfolding x-def sorted-pull-bound-def by (simp add: Let-def)
    moreover have card S ≤ M
    proof −
      have card S = card (set ?xs)
        using S-all set-xs by simp
      also have ... = length ?xs
        using distinct-xs by (simp add: distinct-card)
      finally have card S = length ?xs .
      then show ?thesis
        using True by simp
    qed
    ultimately show ?thesis
      unfolding pull-separates-def D'-def remove-keys-view-def
      using S-all D-empty by auto
  next
    case False
    then have M-lt: M < length ?xs
      by simp
    let ?beta = value-of D (?xs ! M)
    have x-beta: x = ?beta
      using False unfolding x-def sorted-pull-bound-def by (simp add: Let-def)
    have S-lt: S = {u ∈ keys-of D. value-of D u < ?beta}
      using False unfolding S-def sorted-pull-set-def by (simp add: Let-def)
    have S-subset: S ⊆ keys-of D
```

unfolding *S-lt* **by** *blast*
have *S-take*: $S \subseteq$ *set* (*take M ?xs*)
proof
fix *u*
assume *uS*: $u \in S$
then have *u-keys*: $u \in$ *keys-of D* **and** *lt*: *value-of D u* $<$ *?beta*
unfolding *S-lt* **by** *auto*
then have $u \in$ *set ?xs*
using *set-xs* **by** *simp*
then show $u \in$ *set* (*take M ?xs*)
using *sorted-value-less-nth-in-take*[*OF sorted-xs - lt M-lt*] **by** *blast*
qed
have *card-S*: *card S* $\leq M$
proof −
have *card S* $\leq$ *card* (*set* (*take M ?xs*))
using *S-take* **by** (*intro card-mono*) *simp-all*
also have $\ldots \leq$ *length* (*take M ?xs*)
by (*rule card-length*)
also have $\ldots = M$
using *M-lt* **by** *simp*
finally show *?thesis* **.**
qed
have *D′-keys*: *keys-of D′* = *keys-of D* − *S*
unfolding *D′-def remove-keys-view-def* **by** *simp*
have *D′-value*: *value-of D′* = *value-of D*
unfolding *D′-def remove-keys-view-def* **by** *simp*
have *remaining-nonempty*: *keys-of D′* $\neq \{\}$
proof −
have *nth-in*: *?xs* ! $M \in$ *keys-of D*
using *set-xs M-lt nth-mem* **by** *metis*
have *?xs* ! $M \notin S$
unfolding *S-lt* **by** *simp*
then have *?xs* ! $M \in$ *keys-of D* − *S*
using *nth-in* **by** *blast*
then show *?thesis*
using *D′-keys* **by** *blast*
qed
have *pulled-before-remaining*:
$\bigwedge u\ v.\ [\![u \in S;\ v \in$ *keys-of D′*$]\!] \Longrightarrow$ *value-of D u* $\leq$ *value-of D′ v*
proof −
fix *u v*
assume *uS*: $u \in S$ **and** *vD*: $v \in$ *keys-of D′*
have *value-of D u* $<$ *?beta*
using *uS* **unfolding** *S-lt* **by** *blast*
moreover have ¬ *value-of D v* $<$ *?beta*
using *vD D′-keys* **unfolding** *S-lt* **by** *blast*
ultimately have *value-of D u* $\leq$ *value-of D v*
by *simp*
then show *value-of D u* $\leq$ *value-of D′ v*

```
      using D'-value by simp
    qed
    have lower-remaining:
      ⋀v. v ∈ keys-of D' ⟹ x ≤ value-of D' v
    proof −
      fix v
      assume vD: v ∈ keys-of D'
      have ¬ value-of D v < ?beta
        using vD D'-keys unfolding S-lt by blast
      then show x ≤ value-of D' v
        using x-beta D'-value by simp
    qed
    have upper-pulled:
      ⋀u. u ∈ S ⟹ value-of D u < x
      using x-beta unfolding S-lt by blast
    show ?thesis
      unfolding pull-separates-def
      using S-subset card-S D'-keys D'-value remaining-nonempty
        pulled-before-remaining lower-remaining upper-pulled
      by auto
  qed
qed

theorem sorted-pull-exact-split:
  assumes finite-keys: finite (keys-of D)
    and upper: ⋀u. u ∈ keys-of D ⟹ value-of D u < B
  shows sorted-pull-set M D =
    {u ∈ keys-of D. value-of D u < sorted-pull-bound M B D}
proof −
  let ?xs = partition-key-order D
  have set-xs: set ?xs = keys-of D
    using partition-key-order-properties(1)[OF finite-keys] .
  show ?thesis
  proof (cases length ?xs ≤ M)
    case True
    then have sorted-pull-set M D = keys-of D
      using set-xs unfolding sorted-pull-set-def by (simp add: Let-def)
    moreover have sorted-pull-bound M B D = B
      using True unfolding sorted-pull-bound-def by (simp add: Let-def)
    ultimately show ?thesis
    proof −
      have {u ∈ keys-of D. value-of D u < sorted-pull-bound M B D} = keys-of D
      proof
        show {u ∈ keys-of D. value-of D u < sorted-pull-bound M B D} ⊆ keys-of D
          by blast
      next
        show keys-of D ⊆ {u ∈ keys-of D. value-of D u < sorted-pull-bound M B D}
        proof
          fix u
```

```
        assume u: u ∈ keys-of D
        then have value-of D u < B
          by (rule upper)
        then show u ∈ {u ∈ keys-of D. value-of D u < sorted-pull-bound M B D}
          using u ‹sorted-pull-bound M B D = B› by simp
      qed
    qed
    then show ?thesis
      using ‹sorted-pull-set M D = keys-of D› by simp
  qed
 next
  case False
  then show ?thesis
    unfolding sorted-pull-set-def sorted-pull-bound-def by (simp add: Let-def)
 qed
qed

end
theory BMSSP-Bucketed-Partition-Internal
 imports BMSSP-Partition-Interface
   HOL−Library.Discrete-Functions
   HOL−Library.Landau-Symbols
begin
```

13 Bucketed Partition Data Structure

This is the main data-structure theory for the paper-faithful partition layer. The BMSSP paper uses a bucketed structure whose operation bounds are expressed in terms of the ratio between the number of inserted elements and the block size. That ratio is the essential distinction from a sorted-list model: Insert must pay for a search over buckets, not for a search over all stored elements.

The formal model is functional. A partition state contains a block size, an upper bound, a list of buckets, and a value memory. Each bucket has a marker and an unsorted list of key/value entries. The marker is a lower bound for the values in the bucket, and adjacent markers delimit the value ranges used by Pull. The list-level operations below first establish a simple exact model, then a lazy model that permits buckets to grow up to the split threshold before they are rebuilt.

The theory has three layers. The first layer defines the functional state, invariants, Insert, BatchPrepend, and Pull, and proves that their views refine the abstract partition interface. The second layer adds primitive step counters and a potential function for lazy splitting. The third layer names the paper-facing operations and proves the three exported cost bounds: Insert at the ratio-log budget, BatchPrepend at the matching batched budget, and Pull at an amortized linear-in-M budget. The baseline sorted-list partition

state used by the recurrence proofs is not used here.

The amortized analysis uses a scaled credit potential $\Phi(P) = 4 \cdot \sum_b$ *max 0* (*length* (*bp-bucket-entries b*) − *bp-block-size P*), summed over the buckets of *P*. The factor of four is the constant that absorbs the per-Insert charge: each Insert raises Φ by at most four credits, which is exactly the budget that pays for the $O(M)$ rebucket work performed during a Pull when a bucket has overflowed past the lazy split threshold. This bookkeeping is what makes the bound paper-tight rather than merely amortized-but-loose: an Insert that does not split the bucket pays only for the bucket walk $O(log\ (N\ /\ M))$, an Insert that triggers a rebucket pays the same physical cost plus a refund of credits to Φ, and Pull discharges the credits accumulated by its bucket's prior Inserts to fund the rebucket step. Without the factor four the per-Insert charge would not cover the rebucket, and the resulting Insert bound would degrade to $O(N\ /\ M)$; with the factor four it stays at the paper rate.

```
record 'k bp-bucket =
  bp-marker :: real
  bp-bucket-entries :: ('k × real) list

record 'k bucketed-partition =
  bp-block-size :: nat
  bp-upper-bound :: real
  bp-buckets :: 'k bp-bucket list
  bp-values :: 'k ⇒ real

definition bp-bucket-entries-flat :: 'k bp-bucket list ⇒ ('k × real) list where
  bp-bucket-entries-flat bs = concat (map bp-bucket-entries bs)

definition bp-entries :: 'k bucketed-partition ⇒ ('k × real) list where
  bp-entries P = bp-bucket-entries-flat (bp-buckets P)

definition bp-entry-keys :: ('k × real) list ⇒ 'k set where
  bp-entry-keys xs = fst ‘ set xs

definition bp-bucket-keys :: 'k bp-bucket ⇒ 'k set where
  bp-bucket-keys b = bp-entry-keys (bp-bucket-entries b)

definition bp-view :: 'k bucketed-partition ⇒ 'k partition-view where
  bp-view P =
    (| keys-of = bp-entry-keys (bp-entries P),
      value-of = bp-values P |)

definition bp-bucket-sizes-ok :: 'k bucketed-partition ⇒ bool where
  bp-bucket-sizes-ok P ⟷
    (∀ b∈set (bp-buckets P).
      length (bp-bucket-entries b) ≤ bp-block-size P)
```

definition *bp-lazy-bucket-sizes-ok* :: *'k bucketed-partition* ⇒ *bool* **where**
bp-lazy-bucket-sizes-ok P ⟷
(∀ *b*∈*set* (*bp-buckets P*).
length (*bp-bucket-entries b*) ≤ *2* ∗ *bp-block-size P*)

definition *bp-bucket-markers-sorted* :: *'k bucketed-partition* ⇒ *bool* **where**
bp-bucket-markers-sorted P ⟷
sorted-wrt (λ*b c*. *bp-marker b* ≤ *bp-marker c*) (*bp-buckets P*)

definition *bp-bucket-markers-lower-bound* :: *'k bucketed-partition* ⇒ *bool* **where**
bp-bucket-markers-lower-bound P ⟷
(∀ *b*∈*set* (*bp-buckets P*).
∀ *p*∈*set* (*bp-bucket-entries b*). *bp-marker b* ≤ *snd p*)

definition *bp-values-consistent* :: *'k bucketed-partition* ⇒ *bool* **where**
bp-values-consistent P ⟷
(∀ *p*∈*set* (*bp-entries P*). *bp-values P* (*fst p*) = *snd p*)

definition *bp-distinct-keys* :: *'k bucketed-partition* ⇒ *bool* **where**
bp-distinct-keys P ⟷ *distinct* (*map fst* (*bp-entries P*))

fun *bp-bucket-boundaries-ok* :: *'k bp-bucket list* ⇒ *bool* **where**
bp-bucket-boundaries-ok [] ⟷ *True*
| *bp-bucket-boundaries-ok* [*b*] ⟷ *True*
| *bp-bucket-boundaries-ok* (*b* # *c* # *bs*) ⟷
(∀ *p*∈*set* (*bp-bucket-entries b*). *snd p* ≤ *bp-marker c*) ∧
bp-bucket-boundaries-ok (*c* # *bs*)

definition *bp-bucket-boundaries-state-ok* ::
'k bucketed-partition ⇒ *bool* **where**
bp-bucket-boundaries-state-ok P ⟷
bp-bucket-boundaries-ok (*bp-buckets P*)

definition *bp-invariant* :: *'k bucketed-partition* ⇒ *bool* **where**
bp-invariant P ⟷
0 < *bp-block-size P* ∧
bp-distinct-keys P ∧
bp-bucket-sizes-ok P ∧
bp-bucket-markers-sorted P ∧
bp-bucket-markers-lower-bound P ∧
bp-values-consistent P

definition *bp-ordered-invariant* :: *'k bucketed-partition* ⇒ *bool* **where**
bp-ordered-invariant P ⟷
bp-invariant P ∧ *bp-bucket-boundaries-state-ok P*

definition *bp-lazy-invariant* :: *'k bucketed-partition* ⇒ *bool* **where**
bp-lazy-invariant P ⟷
0 < *bp-block-size P* ∧

```
    bp-distinct-keys P ∧
    bp-lazy-bucket-sizes-ok P ∧
    bp-bucket-markers-sorted P ∧
    bp-bucket-markers-lower-bound P ∧
    bp-values-consistent P

definition bp-lazy-ordered-invariant :: 'k bucketed-partition ⇒ bool where
  bp-lazy-ordered-invariant P ⟷
    bp-lazy-invariant P ∧ bp-bucket-boundaries-state-ok P
```

The invariant vocabulary separates three concerns. The basic view of a state is *bp-view*: it exposes only the key set and the value memory required by the abstract partition interface. The structural invariant *bp-invariant* says that the block size is positive, keys are unique, strict bucket sizes are at most *bp-block-size*, markers are sorted, markers are lower bounds for their entries, and the stored value memory agrees with the bucket entries. The ordered invariant *bp-ordered-invariant* adds the boundary condition between adjacent buckets, which is what makes a first-bucket Pull sound.

The lazy variants are used for the amortized Insert proof. In *bp-lazy-invariant*, buckets may temporarily contain up to twice the block size. This slack is exactly what lets Insert avoid rebuilding on every operation. Once a bucket crosses the lazy threshold, the state is rebuilt into strict buckets and the potential drops enough to pay for the split.

```
definition bp-empty :: nat ⇒ real ⇒ 'k bucketed-partition where
  bp-empty M B =
    (| bp-block-size = M,
      bp-upper-bound = B,
      bp-buckets = [],
      bp-values = (λ-. B) |)

definition bp-singleton-bucket :: 'k × real ⇒ 'k bp-bucket where
  bp-singleton-bucket p =
    (| bp-marker = snd p, bp-bucket-entries = [p] |)

lemma bp-singleton-bucket-simps [simp]:
  bp-marker (bp-singleton-bucket p) = snd p
  bp-bucket-entries (bp-singleton-bucket p) = [p]
  unfolding bp-singleton-bucket-def by simp-all

definition bp-make-bucket :: ('k × real) list ⇒ 'k bp-bucket where
  bp-make-bucket xs =
    (| bp-marker = snd (hd xs), bp-bucket-entries = xs |)

fun bp-bucketize-sorted-entries-aux ::
  nat ⇒ nat ⇒ ('k × real) list ⇒ 'k bp-bucket list where
  bp-bucketize-sorted-entries-aux 0 M xs = []
| bp-bucketize-sorted-entries-aux (Suc fuel) M xs =
    (if M = 0 ∨ xs = []
```

```
    then []
    else bp-make-bucket (take M xs) #
      bp-bucketize-sorted-entries-aux fuel M (drop M xs))

definition bp-bucketize-sorted-entries ::
  nat ⇒ ('k × real) list ⇒ 'k bp-bucket list where
  bp-bucketize-sorted-entries M xs =
    bp-bucketize-sorted-entries-aux (length xs) M xs

definition bp-bucketize-entries ::
  nat ⇒ ('k × real) list ⇒ 'k bp-bucket list where
  bp-bucketize-entries M xs = bp-bucketize-sorted-entries M (sort-key snd xs)

lemma bp-bucketize-sorted-entries-aux-empty [simp]:
  bp-bucketize-sorted-entries-aux fuel M [] = []
  by (cases fuel) simp-all

definition bp-rebucket :: 'k bucketed-partition ⇒ 'k bucketed-partition where
  bp-rebucket P =
    P(|bp-buckets := bp-bucketize-entries (bp-block-size P) (bp-entries P)|)

fun bp-local-insert-bucket ::
  nat ⇒ 'k × real ⇒ 'k bp-bucket list ⇒ 'k bp-bucket list where
  bp-local-insert-bucket M p [] = bp-bucketize-entries M [p]
| bp-local-insert-bucket M p [b] =
    (if snd p < bp-marker b
     then bp-bucketize-entries M [p] @ [b]
     else bp-bucketize-entries M (p # bp-bucket-entries b))
| bp-local-insert-bucket M p (b # c # bs) =
    (if snd p < bp-marker b
     then bp-bucketize-entries M [p] @ b # c # bs
     else if snd p < bp-marker c
       then bp-bucketize-entries M (p # bp-bucket-entries b) @ c # bs
       else b # bp-local-insert-bucket M p (c # bs))

definition bp-lazy-bucket-insert-entries ::
  nat ⇒ 'k × real ⇒ 'k bp-bucket ⇒ 'k bp-bucket list where
  bp-lazy-bucket-insert-entries M p b =
    (if length (bp-bucket-entries b) < 2 * M
     then [b(|bp-bucket-entries := p # bp-bucket-entries b|)]
     else bp-bucketize-entries M (p # bp-bucket-entries b))

fun bp-lazy-insert-bucket ::
  nat ⇒ 'k × real ⇒ 'k bp-bucket list ⇒ 'k bp-bucket list where
  bp-lazy-insert-bucket M p [] = bp-bucketize-entries M [p]
| bp-lazy-insert-bucket M p [b] =
    (if snd p < bp-marker b
     then bp-bucketize-entries M [p] @ [b]
     else bp-lazy-bucket-insert-entries M p b)
```

```
| bp-lazy-insert-bucket M p (b # c # bs) =
    (if snd p < bp-marker b
     then bp-bucketize-entries M [p] @ b # c # bs
     else if snd p < bp-marker c
       then bp-lazy-bucket-insert-entries M p b @ c # bs
       else b # bp-lazy-insert-bucket M p (c # bs))

fun bp-insert-bucket :: 'k × real ⇒ 'k bp-bucket list ⇒ 'k bp-bucket list where
  bp-insert-bucket p [] = [bp-singleton-bucket p]
| bp-insert-bucket p (b # bs) =
    (if snd p ≤ bp-marker b
     then bp-singleton-bucket p # b # bs
     else b # bp-insert-bucket p bs)

definition bp-delete-key-from-bucket :: 'k ⇒ 'k bp-bucket ⇒ 'k bp-bucket where
  bp-delete-key-from-bucket x b =
    b(|bp-bucket-entries := filter (λp. fst p ≠ x) (bp-bucket-entries b)|)

definition bp-delete-key :: 'k ⇒ 'k bucketed-partition ⇒ 'k bucketed-partition where
  bp-delete-key x P =
    P(|bp-buckets := map (bp-delete-key-from-bucket x) (bp-buckets P)|)

definition bp-insert ::
  'k ⇒ real ⇒ 'k bucketed-partition ⇒ 'k bucketed-partition where
  bp-insert x b P =
    (let b' = (if x ∈ bp-entry-keys (bp-entries P)
         then min (bp-values P x) b else b);
       P0 = bp-delete-key x P
     in P0(|bp-buckets := bp-insert-bucket (x, b') (bp-buckets P0),
             bp-values := (bp-values P)(x := b')|))

definition bp-local-insert-state ::
  'k ⇒ real ⇒ 'k bucketed-partition ⇒ 'k bucketed-partition where
  bp-local-insert-state x b P =
    (let b' = (if x ∈ bp-entry-keys (bp-entries P)
         then min (bp-values P x) b else b);
       P0 = bp-delete-key x P
     in P0(|bp-buckets :=
             bp-local-insert-bucket (bp-block-size P0) (x, b') (bp-buckets P0),
             bp-values := (bp-values P)(x := b')|))

definition bp-lazy-insert-state ::
  'k ⇒ real ⇒ 'k bucketed-partition ⇒ 'k bucketed-partition where
  bp-lazy-insert-state x b P =
    (let b' = (if x ∈ bp-entry-keys (bp-entries P)
         then min (bp-values P x) b else b);
       P0 = bp-delete-key x P
     in P0(|bp-buckets :=
             bp-lazy-insert-bucket (bp-block-size P0) (x, b') (bp-buckets P0),
```

```
      bp-values := (bp-values P)(x := b')⦈)

definition bp-batch-prepend ::
  ('k × real) list ⇒ 'k bucketed-partition ⇒ 'k bucketed-partition where
  bp-batch-prepend xs P =
    fold (λ(x, b) P'. bp-insert x b P') xs P

fun bp-batch-value-update ::
  ('k × real) list ⇒ ('k ⇒ real) ⇒ 'k ⇒ real where
  bp-batch-value-update [] f = f
| bp-batch-value-update ((x, b) # xs) f =
    bp-batch-value-update xs (f(x := b))

fun bp-batch-max-value :: real ⇒ ('k × real) list ⇒ real where
  bp-batch-max-value beta [] = beta
| bp-batch-max-value beta ((x, b) # xs) =
    bp-batch-max-value (max beta b) xs

fun bp-rebase-first-bucket-marker ::
  real ⇒ 'k bp-bucket list ⇒ 'k bp-bucket list where
  bp-rebase-first-bucket-marker beta [] = []
| bp-rebase-first-bucket-marker beta (b # bs) =
    b⦇bp-marker := beta⦈ # bs

fun bp-drop-empty-prefix :: 'k bp-bucket list ⇒ 'k bp-bucket list where
  bp-drop-empty-prefix [] = []
| bp-drop-empty-prefix (b # bs) =
    (if bp-bucket-entries b = [] then bp-drop-empty-prefix bs else b # bs)

definition bp-bucketed-batch-prepend-state ::
  ('k × real) list ⇒ 'k bucketed-partition ⇒
   'k bucketed-partition where
  bp-bucketed-batch-prepend-state xs P =
    (case xs of
       [] ⇒ P
     | p # ps ⇒
         (let beta = bp-batch-max-value (snd p) ps;
            new = bp-bucketize-entries (bp-block-size P) xs;
            old = bp-rebase-first-bucket-marker beta
              (bp-drop-empty-prefix (bp-buckets P))
          in P⦇bp-buckets := new @ old,
                bp-values := bp-batch-value-update xs (bp-values P)⦈))

definition bp-delete-keys-from-bucket :: 'k set ⇒ 'k bp-bucket ⇒ 'k bp-bucket where
  bp-delete-keys-from-bucket S b =
    b⦇bp-bucket-entries := filter (λp. fst p ∉ S) (bp-bucket-entries b)⦈

definition bp-delete-keys :: 'k set ⇒ 'k bucketed-partition ⇒ 'k bucketed-partition
where
```

```
  bp-delete-keys S P =
    P(|bp-buckets := map (bp-delete-keys-from-bucket S) (bp-buckets P)|)

fun bp-min-value :: real ⇒ ('k × real) list ⇒ real where
  bp-min-value B [] = B
| bp-min-value B ((x, b) # xs) = min b (bp-min-value B xs)

definition bp-bucket-below-bound :: 'k bp-bucket ⇒ real ⇒ bool where
  bp-bucket-below-bound b beta ⟷
    (∀ p∈set (bp-bucket-entries b). snd p < beta)

definition bp-first-bucket-pull ::
  nat ⇒ real ⇒ 'k bucketed-partition ⇒
   'k set × real × 'k bucketed-partition where
  bp-first-bucket-pull M B P =
    (case bp-buckets P of
       b # c # bs ⇒
         (let S = bp-bucket-keys b
          in (S, bp-marker c, bp-delete-keys S P))
     | - ⇒ ({}, B, P))

definition bp-pull-set :: nat ⇒ 'k bucketed-partition ⇒ 'k set where
  bp-pull-set M P =
    (if length (bp-entries P) ≤ M
     then bp-entry-keys (bp-entries P)
     else {})

definition bp-pull-bound :: nat ⇒ real ⇒ 'k bucketed-partition ⇒ real where
  bp-pull-bound M B P =
    (if length (bp-entries P) ≤ M
     then B
     else bp-min-value B (bp-entries P))

definition bp-conservative-pull ::
  nat ⇒ real ⇒ 'k bucketed-partition ⇒
   'k set × real × 'k bucketed-partition where
  bp-conservative-pull M B P =
    (let S = bp-pull-set M P;
       beta = bp-pull-bound M B P
     in (S, beta, bp-delete-keys S P))

definition bp-can-first-bucket-pull ::
  nat ⇒ 'k bucketed-partition ⇒ bool where
  bp-can-first-bucket-pull M P ⟷
    (case bp-buckets P of
       b # c # bs ⇒
         length (bp-entries P) > M ∧
         length (bp-bucket-entries b) ≤ M ∧
         bp-bucket-below-bound b (bp-marker c) ∧
```

```
      bp-bucket-entries-flat (c # bs) ≠ []
  | - ⇒ False)

definition bp-pull ::
  nat ⇒ real ⇒ 'k bucketed-partition ⇒
    'k set × real × 'k bucketed-partition where
  bp-pull M B P =
    (if bp-can-first-bucket-pull M P
     then bp-first-bucket-pull M B P
     else bp-conservative-pull M B P)

lemma bp-can-first-bucket-pullE:
  assumes bp-can-first-bucket-pull M P
  obtains b c bs where
    bp-buckets P = b # c # bs
    length (bp-entries P) > M
    length (bp-bucket-entries b) ≤ M
    bp-bucket-below-bound b (bp-marker c)
    bp-bucket-entries-flat (c # bs) ≠ []
  using assms
proof (cases bp-buckets P)
  case Nil
  then show ?thesis
    using assms unfolding bp-can-first-bucket-pull-def by simp
next
  case (Cons b rest)
  note buckets-outer = Cons
  then show ?thesis
  proof (cases rest)
    case Nil
    then show ?thesis
      using assms buckets-outer unfolding bp-can-first-bucket-pull-def by simp
  next
    case (Cons c bs)
    have buckets: bp-buckets P = b # c # bs
      using buckets-outer Cons by simp
    have len-entries: length (bp-entries P) > M
      using assms buckets unfolding bp-can-first-bucket-pull-def by simp
    have len-b: length (bp-bucket-entries b) ≤ M
      using assms buckets unfolding bp-can-first-bucket-pull-def by simp
    have below: bp-bucket-below-bound b (bp-marker c)
      using assms buckets unfolding bp-can-first-bucket-pull-def by simp
    have tail-nonempty: bp-bucket-entries-flat (c # bs) ≠ []
      using assms buckets unfolding bp-can-first-bucket-pull-def by simp
    show ?thesis
      by (rule that[OF buckets len-entries len-b below tail-nonempty])
  qed
qed
```

lemma *bp-entry-keys-filter-neq* [*simp*]:
bp-entry-keys (*filter* (λ*p*. *fst p* ≠ *x*) *xs*) = *bp-entry-keys xs* − {*x*}
unfolding *bp-entry-keys-def* **by** *auto*

lemma *bp-entry-keys-filter-notin* [*simp*]:
bp-entry-keys (*filter* (λ*p*. *fst p* ∉ *S*) *xs*) = *bp-entry-keys xs* − *S*
unfolding *bp-entry-keys-def* **by** *auto*

lemma *bp-bucket-keys-alt* [*simp*]:
bp-bucket-keys b = *fst* ‘ *set* (*bp-bucket-entries b*)
unfolding *bp-bucket-keys-def bp-entry-keys-def* **by** *simp*

lemma *bp-entries-empty* [*simp*]:
bp-entries (*bp-empty M B*) = []
unfolding *bp-entries-def bp-empty-def bp-bucket-entries-flat-def* **by** *simp*

lemma *bp-bucket-entries-flat-append* [*simp*]:
bp-bucket-entries-flat (*bs* @ *cs*) =
bp-bucket-entries-flat bs @ *bp-bucket-entries-flat cs*
unfolding *bp-bucket-entries-flat-def* **by** *simp*

lemma *bp-bucket-entries-flat-rebase-first-bucket-marker* [*simp*]:
bp-bucket-entries-flat (*bp-rebase-first-bucket-marker beta bs*) =
bp-bucket-entries-flat bs
by (*cases bs*) (*simp-all add*: *bp-bucket-entries-flat-def*)

lemma *bp-bucket-entries-flat-drop-empty-prefix* [*simp*]:
bp-bucket-entries-flat (*bp-drop-empty-prefix bs*) =
bp-bucket-entries-flat bs
by (*induction bs*) (*simp-all add*: *bp-bucket-entries-flat-def*)

lemma *length-bp-rebase-first-bucket-marker* [*simp*]:
length (*bp-rebase-first-bucket-marker beta bs*) = *length bs*
by (*cases bs*) *simp-all*

lemma *length-bp-drop-empty-prefix-le* [*simp*]:
length (*bp-drop-empty-prefix bs*) ≤ *length bs*
by (*induction bs*) *auto*

lemma *bp-entry-keys-rebase-first-bucket-marker* [*simp*]:
bp-entry-keys
(*bp-bucket-entries-flat* (*bp-rebase-first-bucket-marker beta bs*)) =
bp-entry-keys (*bp-bucket-entries-flat bs*)
by *simp*

lemma *bp-entry-keys-drop-empty-prefix* [*simp*]:
bp-entry-keys (*bp-bucket-entries-flat* (*bp-drop-empty-prefix bs*)) =
bp-entry-keys (*bp-bucket-entries-flat bs*)
by *simp*

```
lemma bp-drop-empty-prefix-set-subset:
  set (bp-drop-empty-prefix bs) ⊆ set bs
  by (induction bs) auto

lemma bp-drop-empty-prefix-head-nonempty:
  assumes bp-drop-empty-prefix bs = b # bs'
  shows bp-bucket-entries b ≠ []
  using assms
proof (induction bs)
  case Nil
  then show ?case by simp
next
  case (Cons a bs)
  show ?case
  proof (cases bp-bucket-entries a = [])
    case True
    then show ?thesis
      using Cons.IH Cons.prems by simp
  next
    case False
    then have a = b
      using Cons.prems by simp
    then show ?thesis
      using False by simp
  qed
qed

lemma bp-bucket-boundaries-ok-drop-empty-prefix:
  assumes bp-bucket-boundaries-ok bs
  shows bp-bucket-boundaries-ok (bp-drop-empty-prefix bs)
  using assms
proof (induction bs)
  case Nil
  then show ?case by simp
next
  case (Cons b bs)
  show ?case
  proof (cases bp-bucket-entries b = [])
    case True
    have bp-bucket-boundaries-ok bs
      using Cons.prems by (cases bs) simp-all
    then show ?thesis
      using True Cons.IH by simp
  next
    case False
    then show ?thesis
      using Cons.prems by simp
  qed
```

qed

lemma *bp-bucket-markers-sorted-drop-empty-prefix*:
assumes *sorted-wrt* (λ*b c*. *bp-marker b* ≤ *bp-marker c*) *bs*
shows *sorted-wrt* (λ*b c*. *bp-marker b* ≤ *bp-marker c*)
(*bp-drop-empty-prefix bs*)
using *assms*
proof (*induction bs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons b bs*)
show *?case*
proof (*cases bp-bucket-entries b* = [])
case *True*
have *sorted-wrt* (λ*b c*. *bp-marker b* ≤ *bp-marker c*) *bs*
using *Cons.prems* **by** *simp*
then show *?thesis*
using *True Cons.IH* **by** *simp*
next
case *False*
then show *?thesis*
using *Cons.prems* **by** *simp*
qed
qed

lemma *bp-bucket-boundaries-ok-rebase-first-bucket-marker* [*simp*]:
bp-bucket-boundaries-ok (*bp-rebase-first-bucket-marker beta bs*) =
bp-bucket-boundaries-ok bs
proof (*cases bs*)
case *Nil*
then show *?thesis* **by** *simp*
next
case (*Cons b rest*)
then show *?thesis*
by (*cases rest*) *simp-all*
qed

lemma *bp-batch-value-update-notin*:
assumes *x* ∉ *fst* ‘ *set xs*
shows *bp-batch-value-update xs f x* = *f x*
using *assms* **by** (*induction xs arbitrary*: *f*) *auto*

lemma *bp-batch-value-update-in-set-distinct*:
assumes *distinct*: *distinct* (*map fst xs*)
and *member*: (*x*, *b*) ∈ *set xs*
shows *bp-batch-value-update xs f x* = *b*
using *assms*
proof (*induction xs arbitrary*: *f*)

```
  case Nil
  then show ?case by simp
next
  case (Cons p xs)
  obtain y c where p-def: p = (y, c)
    by force
  show ?case
  proof (cases x = y)
    case True
    have x-not-tail: x ∉ fst ‘ set xs
      using True Cons.prems p-def by auto
    then have (x, b) ∉ set xs
      by force
    then have (x, b) = (y, c)
      using Cons.prems p-def by auto
    then have bc: b = c
      by simp
    have y-not-tail: y ∉ fst ‘ set xs
      using x-not-tail True by simp
    have unchanged:
      bp-batch-value-update xs (f(y := c)) y = c
      using bp-batch-value-update-notin[OF y-not-tail, of f(y := c)]
      by simp
    then show ?thesis
      unfolding p-def True bc
      by simp
  next
    case False
    then have (x, b) ∈ set xs
      using Cons.prems p-def by auto
    moreover have distinct (map fst xs)
      using Cons.prems p-def by simp
    ultimately show ?thesis
      unfolding p-def by (simp add: Cons.IH)
  qed
qed

lemma batch-min-update-value-distinct-disjoint:
  assumes distinct: distinct (map fst xs)
    and disjoint: fst ‘ set xs ∩ keys-of D = {}
  shows value-of (batch-min-update D xs) =
    bp-batch-value-update xs (value-of D)
  using distinct disjoint
proof (induction xs arbitrary: D)
  case Nil
  then show ?case
    unfolding batch-min-update-def by simp
next
  case (Cons p xs)
```

```
  obtain x b where p-def: p = (x, b)
    by force
  have distinct-tail: distinct (map fst xs)
    using Cons.prems p-def by simp
  have x-not-tail: x ∉ fst ‘ set xs
    using Cons.prems p-def by auto
  have x-not-D: x ∉ keys-of D
    using Cons.prems p-def by auto
  have disjoint-tail:
    fst ‘ set xs ∩ keys-of (min-update D x b) = {}
    using Cons.prems x-not-tail p-def
    unfolding min-update-def by auto
  have tail:
    value-of (batch-min-update (min-update D x b) xs) =
      bp-batch-value-update xs (value-of (min-update D x b))
    by (rule Cons.IH[OF distinct-tail disjoint-tail])
  have fold-tail:
    value-of
      (fold (λ(x, b) D'. min-update D' x b) xs (min-update D x b)) =
      bp-batch-value-update xs (value-of (min-update D x b))
    using tail unfolding batch-min-update-def .
  have step-values:
    value-of (min-update D x b) = (value-of D)(x := b)
    using x-not-D unfolding min-update-def by simp
  show ?case
    unfolding p-def
    by (simp add: batch-min-update-def fold-tail step-values)
qed

lemma bp-batch-max-value-upper:
  assumes ⋀p. p ∈ set xs ⟹ snd p ≤ gamma
    and beta ≤ gamma
  shows bp-batch-max-value beta xs ≤ gamma
  using assms
proof (induction xs arbitrary: beta)
  case Nil
  then show ?case by simp
next
  case (Cons p xs)
  obtain x b where p-def: p = (x, b)
    by force
  have head-le: b ≤ gamma
    using Cons.prems(1)[of p] p-def by simp
  have max-le: max beta b ≤ gamma
    using Cons.prems(2) head-le by simp
  have bp-batch-max-value (max beta (snd p)) xs ≤ gamma
  proof (rule Cons.IH)
    fix q
    assume q ∈ set xs
```

```
    then show snd q ≤ gamma
      using Cons.prems by simp
  qed (use max-le p-def in simp)
  then show ?case
    unfolding p-def by simp
qed

lemma bp-batch-max-value-ge-initial:
  beta ≤ bp-batch-max-value beta xs
proof (induction xs arbitrary: beta)
  case Nil
  then show ?case by simp
next
  case (Cons p xs)
  obtain x b where p-def: p = (x, b)
    by force
  have beta ≤ max beta b
    by simp
  moreover have max beta b ≤ bp-batch-max-value (max beta b) xs
    by (rule Cons.IH)
  ultimately have beta ≤ bp-batch-max-value (max beta b) xs
    by linarith
  then show ?case
    unfolding p-def by simp
qed

lemma bp-batch-max-value-ge-member:
  assumes p ∈ set xs
  shows snd p ≤ bp-batch-max-value beta xs
  using assms
proof (induction xs arbitrary: beta)
  case Nil
  then show ?case by simp
next
  case (Cons q xs)
  obtain y c where q-def: q = (y, c)
    by force
  show ?case
  proof (cases p = q)
    case True
    have snd p ≤ max beta c
      using True q-def by simp
    moreover have max beta c ≤ bp-batch-max-value (max beta c) xs
      by (rule bp-batch-max-value-ge-initial)
    ultimately have snd p ≤ bp-batch-max-value (max beta c) xs
      by linarith
    then show ?thesis
      unfolding q-def by simp
  next
```

```
    case False
    then have p ∈ set xs
      using Cons.prems by simp
    then show ?thesis
      unfolding q-def by (simp add: Cons.IH)
  qed
qed

lemma bp-batch-max-value-ge-member-Cons:
  assumes q ∈ set (p # ps)
  shows snd q ≤ bp-batch-max-value (snd p) ps
  using assms
  by (cases q = p)
    (simp-all add: bp-batch-max-value-ge-initial bp-batch-max-value-ge-member)

lemma bp-bucket-entries-flat-bucketize-sorted-entries-aux:
  assumes 0 < M
    and length xs ≤ fuel
  shows bp-bucket-entries-flat
      (bp-bucketize-sorted-entries-aux fuel M xs) = xs
  using assms
proof (induction fuel arbitrary: xs)
  case 0
  then have xs = []
    by simp
  then show ?case
    by (simp add: bp-bucket-entries-flat-def)
next
  case (Suc fuel)
  show ?case
  proof (cases xs = [])
    case True
    then show ?thesis
      by (simp add: bp-bucket-entries-flat-def)
  next
    case False
    have len-drop: length (drop M xs) ≤ fuel
      using Suc.prems False by (cases M) auto
    have tail:
      bp-bucket-entries-flat
        (bp-bucketize-sorted-entries-aux fuel M (drop M xs)) =
       drop M xs
      by (rule Suc.IH[OF Suc.prems(1) len-drop])
    show ?thesis
      using Suc.prems False tail
      by (simp add: bp-bucket-entries-flat-def bp-make-bucket-def)
  qed
qed
```

lemma *bp-bucket-entries-flat-bucketize-sorted-entries*:
assumes *0 < M*
shows *bp-bucket-entries-flat (bp-bucketize-sorted-entries M xs) = xs*
using *assms*
unfolding *bp-bucketize-sorted-entries-def*
by (*simp add*: *bp-bucket-entries-flat-bucketize-sorted-entries-aux*)

lemma *bp-bucket-entries-flat-bucketize-entries*:
assumes *0 < M*
shows *bp-bucket-entries-flat (bp-bucketize-entries M xs) = sort-key snd xs*
using *assms*
unfolding *bp-bucketize-entries-def*
by (*simp add*: *bp-bucket-entries-flat-bucketize-sorted-entries*)

lemma *set-bp-bucket-entries-flat-bucketize-entries*:
assumes *0 < M*
shows *set (bp-bucket-entries-flat (bp-bucketize-entries M xs)) = set xs*
using *assms*
by (*simp add*: *bp-bucket-entries-flat-bucketize-entries*)

lemma *bp-entry-keys-bucketize-entries* [*simp*]:
assumes *0 < M*
shows *bp-entry-keys (bp-bucket-entries-flat (bp-bucketize-entries M xs)) =*
bp-entry-keys xs
using *assms*
unfolding *bp-entry-keys-def*
by (*simp add*: *set-bp-bucket-entries-flat-bucketize-entries*)

lemma *bp-bucketize-sorted-entries-aux-sizes-ok*:
assumes *0 < M*
shows ∀ *b*∈*set (bp-bucketize-sorted-entries-aux fuel M xs)*.
length (bp-bucket-entries b) ≤ M
using *assms*
by (*induction fuel arbitrary*: *xs*) (*auto simp*: *bp-make-bucket-def*)

lemma *bp-bucketize-sorted-entries-sizes-ok*:
assumes *0 < M*
shows ∀ *b*∈*set (bp-bucketize-sorted-entries M xs)*.
length (bp-bucket-entries b) ≤ M
using *assms*
unfolding *bp-bucketize-sorted-entries-def*
by (*rule bp-bucketize-sorted-entries-aux-sizes-ok*)

lemma *bp-bucketize-entries-sizes-ok*:
assumes *0 < M*
shows ∀ *b*∈*set (bp-bucketize-entries M xs)*.
length (bp-bucket-entries b) ≤ M
using *assms*
unfolding *bp-bucketize-entries-def*

```
  by (rule bp-bucketize-sorted-entries-sizes-ok)

lemma bp-sorted-wrt-snd-sort-key [simp]:
  sorted-wrt (λp q. snd p ≤ snd q) (sort-key snd xs)
proof −
  have sorted (map snd (sort-key snd xs))
    by (rule sorted-sort-key)
  then show ?thesis
    by (simp add: sorted-map)
qed

lemma sorted-wrt-hd-snd-le:
  fixes xs :: ('k × real) list
  assumes sorted: sorted-wrt (λp q. snd p ≤ snd q) xs
    and xs: xs ≠ []
    and p: p ∈ set xs
  shows snd (hd xs) ≤ snd p
proof (cases xs)
  case Nil
  then show ?thesis
    using xs by simp
next
  case (Cons q qs)
  show ?thesis
  proof (cases p = q)
    case True
    have snd (hd xs) = snd p
      using Cons True by simp
    then show ?thesis
      by simp
  next
    case False
    then have p ∈ set qs
      using p Cons by simp
    moreover have ∀ r∈set qs. snd q ≤ snd r
      using sorted Cons by simp
    ultimately show ?thesis
      using Cons by simp
  qed
qed

lemma sorted-wrt-snd-take-drop-le:
  fixes xs :: ('k × real) list
  assumes sorted: sorted-wrt (λp q. snd p ≤ snd q) xs
    and p: p ∈ set (take n xs)
    and q: q ∈ set (drop n xs)
  shows snd p ≤ snd q
proof −
  have sorted-wrt (λp q. snd p ≤ snd q) (take n xs @ drop n xs)
```

```
    using sorted by simp
  then have ∀ p∈set (take n xs). ∀ q∈set (drop n xs). snd p ≤ snd q
    unfolding sorted-wrt-append by blast
  then show ?thesis
    using p q by blast
qed

lemma bp-bucketize-sorted-entries-aux-marker-in-set:
  assumes b ∈ set (bp-bucketize-sorted-entries-aux fuel M xs)
  shows ∃ p∈set xs. bp-marker b = snd p
  using assms
proof (induction fuel arbitrary: xs)
  case 0
  then show ?case
    by simp
next
  case (Suc fuel)
  show ?case
  proof (cases M = 0 ∨ xs = [])
    case True
    then show ?thesis
      using Suc.prems by simp
  next
    case False
    have M-pos: 0 < M
      using False by simp
    have xs-nonempty: xs ≠ []
      using False by simp
    from Suc.prems have b-cases:
      b = bp-make-bucket (take M xs) ∨
       b ∈ set (bp-bucketize-sorted-entries-aux fuel M (drop M xs))
      using False by simp
    from b-cases show ?thesis
    proof
      assume b-def: b = bp-make-bucket (take M xs)
      have hd (take M xs) ∈ set xs
        using M-pos xs-nonempty by (cases xs) auto
      then show ?thesis
        unfolding b-def bp-make-bucket-def by auto
    next
      assume b-tail:
        b ∈ set (bp-bucketize-sorted-entries-aux fuel M (drop M xs))
      obtain p where p-drop: p ∈ set (drop M xs)
        and marker: bp-marker b = snd p
        using Suc.IH[OF b-tail] by auto
      have p ∈ set xs
        by (meson in-set-dropD p-drop)
      then show ?thesis
        using marker by auto
```

```
    qed
  qed
qed

lemma bp-bucketize-sorted-entries-aux-markers-lower-bound:
  fixes xs :: ('k × real) list
  assumes M-pos: 0 < M
    and sorted: sorted-wrt (λp q. snd p ≤ snd q) xs
  shows ∀ b∈set (bp-bucketize-sorted-entries-aux fuel M xs).
    ∀ p∈set (bp-bucket-entries b). bp-marker b ≤ snd p
  using M-pos sorted
proof (induction fuel arbitrary: xs)
  case 0
  then show ?case
    by simp
next
  case (Suc fuel)
  show ?case
  proof (cases xs = [])
    case True
    then show ?thesis
      by simp
  next
    case False
    have current:
      ∀ p∈set (bp-bucket-entries (bp-make-bucket (take M xs))).
        bp-marker (bp-make-bucket (take M xs)) ≤ snd p
    proof
      fix p
      assume p: p ∈ set (bp-bucket-entries (bp-make-bucket (take M xs)))
      then have p-take: p ∈ set (take M xs)
        unfolding bp-make-bucket-def by simp
      have sorted-take: sorted-wrt (λp q. snd p ≤ snd q) (take M xs)
        using Suc.prems by simp
      have take-nonempty: take M xs ≠ []
        using p-take by (cases take M xs) auto
      have snd (hd (take M xs)) ≤ snd p
        by (rule sorted-wrt-hd-snd-le[OF sorted-take take-nonempty p-take])
      then show bp-marker (bp-make-bucket (take M xs)) ≤ snd p
        unfolding bp-make-bucket-def by simp
    qed
    have tail:
      ∀ b∈set (bp-bucketize-sorted-entries-aux fuel M (drop M xs)).
        ∀ p∈set (bp-bucket-entries b). bp-marker b ≤ snd p
      by (rule Suc.IH) (use Suc.prems in simp-all)
    show ?thesis
      using False current tail Suc.prems by simp
  qed
qed
```

```
lemma bp-bucketize-sorted-entries-markers-lower-bound:
  fixes xs :: ('k × real) list
  assumes 0 < M
    and sorted-wrt (λp q. snd p ≤ snd q) xs
  shows ∀ b∈set (bp-bucketize-sorted-entries M xs).
    ∀ p∈set (bp-bucket-entries b). bp-marker b ≤ snd p
  using assms
  unfolding bp-bucketize-sorted-entries-def
  by (rule bp-bucketize-sorted-entries-aux-markers-lower-bound)

lemma bp-bucketize-entries-markers-lower-bound:
  assumes 0 < M
  shows ∀ b∈set (bp-bucketize-entries M xs).
    ∀ p∈set (bp-bucket-entries b). bp-marker b ≤ snd p
  using assms
  unfolding bp-bucketize-entries-def
  by (rule bp-bucketize-sorted-entries-markers-lower-bound) simp

lemma bp-bucketize-sorted-entries-aux-markers-sorted:
  fixes xs :: ('k × real) list
  assumes M-pos: 0 < M
    and sorted: sorted-wrt (λp q. snd p ≤ snd q) xs
  shows sorted-wrt (λb c. bp-marker b ≤ bp-marker c)
    (bp-bucketize-sorted-entries-aux fuel M xs)
  using M-pos sorted
proof (induction fuel arbitrary: xs)
  case 0
  then show ?case
    by simp
next
  case (Suc fuel)
  show ?case
  proof (cases xs = [])
    case True
    then show ?thesis
      by simp
  next
    case False
    let ?tail = bp-bucketize-sorted-entries-aux fuel M (drop M xs)
    have tail-sorted:
      sorted-wrt (λb c. bp-marker b ≤ bp-marker c) ?tail
      by (rule Suc.IH) (use Suc.prems in simp-all)
    have current-le-tail:
      ∀ c∈set ?tail.
        bp-marker (bp-make-bucket (take M xs)) ≤ bp-marker c
    proof
      fix c
      assume c: c ∈ set ?tail
```

```
      obtain p where p-drop: p ∈ set (drop M xs)
        and c-marker: bp-marker c = snd p
        using bp-bucketize-sorted-entries-aux-marker-in-set[OF c] by auto
      have p-xs: p ∈ set xs
        by (meson in-set-dropD p-drop)
      have head-le: snd (hd xs) ≤ snd p
        by (rule sorted-wrt-hd-snd-le[OF Suc.prems(2) False p-xs])
      have marker-eq: bp-marker (bp-make-bucket (take M xs)) = snd (hd xs)
        using Suc.prems False by (cases xs) (auto simp: bp-make-bucket-def)
      show bp-marker (bp-make-bucket (take M xs)) ≤ bp-marker c
        using head-le marker-eq c-marker by simp
    qed
    show ?thesis
      using False tail-sorted current-le-tail Suc.prems by simp
  qed
qed

lemma bp-bucketize-sorted-entries-markers-sorted:
  fixes xs :: ('k × real) list
  assumes 0 < M
    and sorted-wrt (λp q. snd p ≤ snd q) xs
  shows sorted-wrt (λb c. bp-marker b ≤ bp-marker c)
    (bp-bucketize-sorted-entries M xs)
  using assms
  unfolding bp-bucketize-sorted-entries-def
  by (rule bp-bucketize-sorted-entries-aux-markers-sorted)

lemma bp-bucketize-entries-markers-sorted:
  assumes 0 < M
  shows sorted-wrt (λb c. bp-marker b ≤ bp-marker c)
    (bp-bucketize-entries M xs)
  using assms
  unfolding bp-bucketize-entries-def
  by (rule bp-bucketize-sorted-entries-markers-sorted) simp

lemma bp-bucketize-entries-marker-in-set:
  assumes b ∈ set (bp-bucketize-entries M xs)
  shows ∃ p∈set xs. bp-marker b = snd p
proof −
  obtain p where p-sort: p ∈ set (sort-key snd xs)
    and marker: bp-marker b = snd p
    using assms
    unfolding bp-bucketize-entries-def bp-bucketize-sorted-entries-def
    by (auto dest: bp-bucketize-sorted-entries-aux-marker-in-set)
  then have p ∈ set xs
    by simp
  then show ?thesis
    using marker by auto
qed
```

```
lemma bp-bucketize-sorted-entries-aux-boundaries-ok:
  fixes xs :: ('k × real) list
  assumes M-pos: 0 < M
    and sorted: sorted-wrt (λp q. snd p ≤ snd q) xs
  shows bp-bucket-boundaries-ok
    (bp-bucketize-sorted-entries-aux fuel M xs)
  using M-pos sorted
proof (induction fuel arbitrary: xs)
  case 0
  then show ?case
    by simp
next
  case (Suc fuel)
  show ?case
  proof (cases xs = [])
    case True
    then show ?thesis
      by simp
  next
    case False
    let ?tail = bp-bucketize-sorted-entries-aux fuel M (drop M xs)
    have tail-boundaries: bp-bucket-boundaries-ok ?tail
      by (rule Suc.IH) (use Suc.prems in simp-all)
    show ?thesis
    proof (cases ?tail)
      case Nil
      then show ?thesis
        using False Suc.prems by simp
    next
      case (Cons c cs)
      have c-in: c ∈ set ?tail
        using Cons by simp
      obtain q where q-drop: q ∈ set (drop M xs)
        and c-marker: bp-marker c = snd q
        using bp-bucketize-sorted-entries-aux-marker-in-set[OF c-in] by auto
      have current-before-c:
        ∀ p∈set (bp-bucket-entries (bp-make-bucket (take M xs))).
          snd p ≤ bp-marker c
      proof
        fix p
        assume p:
          p ∈ set (bp-bucket-entries (bp-make-bucket (take M xs)))
        then have p-take: p ∈ set (take M xs)
          unfolding bp-make-bucket-def by simp
        have snd p ≤ snd q
          by (rule sorted-wrt-snd-take-drop-le[OF Suc.prems(2) p-take q-drop])
        then show snd p ≤ bp-marker c
          using c-marker by simp
```

```
    qed
    show ?thesis
      using False Cons tail-boundaries current-before-c Suc.prems by simp
  qed
 qed
qed

lemma bp-bucketize-sorted-entries-boundaries-ok:
 fixes xs :: ('k × real) list
 assumes 0 < M
   and sorted-wrt (λp q. snd p ≤ snd q) xs
 shows bp-bucket-boundaries-ok (bp-bucketize-sorted-entries M xs)
 using assms
 unfolding bp-bucketize-sorted-entries-def
 by (rule bp-bucketize-sorted-entries-aux-boundaries-ok)

lemma bp-bucketize-entries-boundaries-ok:
 assumes 0 < M
 shows bp-bucket-boundaries-ok (bp-bucketize-entries M xs)
 using assms
 unfolding bp-bucketize-entries-def
 by (rule bp-bucketize-sorted-entries-boundaries-ok) simp

lemma distinct-map-fst-sort-key:
 assumes distinct (map fst xs)
 shows distinct (map fst (sort-key f xs))
proof −
 have distinct-sort: distinct (sort-key f xs)
   using assms unfolding distinct-map by simp
 have inj: inj-on fst (set (sort-key f xs))
   using assms unfolding distinct-map by simp
 show ?thesis
   unfolding distinct-map using distinct-sort inj by simp
qed

lemma bp-rebucket-view [simp]:
 assumes 0 < bp-block-size P
 shows bp-view (bp-rebucket P) = bp-view P
 using assms
 unfolding bp-rebucket-def bp-view-def bp-entries-def
 by simp

lemma bp-rebucket-invariant:
 assumes inv: bp-invariant P
 shows bp-invariant (bp-rebucket P)
proof −
 let ?M = bp-block-size P
 let ?xs = bp-entries P
 have M-pos: 0 < ?M
```

```
  using inv unfolding bp-invariant-def by blast
have distinct-xs: distinct (map fst ?xs)
  using inv unfolding bp-invariant-def bp-distinct-keys-def by blast
have vals:
  ∀ p∈set ?xs. bp-values P (fst p) = snd p
  using inv unfolding bp-invariant-def bp-values-consistent-def by blast
have flat:
  bp-bucket-entries-flat (bp-bucketize-entries ?M ?xs) = sort-key snd ?xs
  by (rule bp-bucket-entries-flat-bucketize-entries[OF M-pos])
have flat-rebucket:
  bp-entries (bp-rebucket P) = sort-key snd ?xs
  using flat unfolding bp-rebucket-def bp-entries-def by simp
have sizes-ok:
  ∀ b∈set (bp-bucketize-entries ?M ?xs).
    length (bp-bucket-entries b) ≤ ?M
  by (rule bp-bucketize-entries-sizes-ok[OF M-pos])
have markers-sorted:
  sorted-wrt (λb c. bp-marker b ≤ bp-marker c)
    (bp-bucketize-entries ?M ?xs)
  by (rule bp-bucketize-entries-markers-sorted[OF M-pos])
have markers-lower:
  ∀ b∈set (bp-bucketize-entries ?M ?xs).
    ∀ p∈set (bp-bucket-entries b). bp-marker b ≤ snd p
  by (rule bp-bucketize-entries-markers-lower-bound[OF M-pos])
have set-flat:
  set (bp-bucket-entries-flat (bp-bucketize-entries ?M ?xs)) = set ?xs
  by (rule set-bp-bucket-entries-flat-bucketize-entries[OF M-pos])
show ?thesis
  unfolding bp-invariant-def
proof (intro conjI)
  show 0 < bp-block-size (bp-rebucket P)
    using M-pos unfolding bp-rebucket-def by simp
  show bp-distinct-keys (bp-rebucket P)
    using distinct-map-fst-sort-key[OF distinct-xs, of snd]
    unfolding bp-distinct-keys-def flat-rebucket by simp
  show bp-bucket-sizes-ok (bp-rebucket P)
    using sizes-ok
    unfolding bp-rebucket-def bp-bucket-sizes-ok-def by simp
  show bp-bucket-markers-sorted (bp-rebucket P)
    using markers-sorted
    unfolding bp-rebucket-def bp-bucket-markers-sorted-def by simp
  show bp-bucket-markers-lower-bound (bp-rebucket P)
    using markers-lower
    unfolding bp-rebucket-def bp-bucket-markers-lower-bound-def by simp
  show bp-values-consistent (bp-rebucket P)
    using vals set-flat
    unfolding bp-rebucket-def bp-values-consistent-def bp-entries-def
      bp-entry-keys-def
    by auto
```

```
  qed
qed

lemma bp-rebucket-boundaries-state-ok:
  assumes inv: bp-invariant P
  shows bp-bucket-boundaries-state-ok (bp-rebucket P)
proof −
  have M-pos: 0 < bp-block-size P
    using inv unfolding bp-invariant-def by blast
  show ?thesis
    using bp-bucketize-entries-boundaries-ok[OF M-pos, of bp-entries P]
    unfolding bp-bucket-boundaries-state-ok-def bp-rebucket-def by simp
qed

lemma bp-rebucket-ordered-invariant:
  assumes inv: bp-invariant P
  shows bp-ordered-invariant (bp-rebucket P)
  unfolding bp-ordered-invariant-def
  using bp-rebucket-invariant[OF inv] bp-rebucket-boundaries-state-ok[OF inv]
  by blast
```

Rebucketing is the canonical repair operation. It sorts the flat entry list by value, chunks it into blocks of size *bp-block-size*, and rebuilds markers from the first entry of each block. The view theorem $0 < \textit{bp-block-size}\ ?P \Longrightarrow \textit{bp-view}\ (\textit{bp-rebucket}\ ?P) = \textit{bp-view}\ ?P$ says this repair is observationally invisible at the abstract partition interface: keys and remembered values do not change. The invariant theorems above then show that rebucketing restores strict bucket sizes, sorted markers, lower-bound markers, and adjacent bucket boundaries. Later regularized operations use this fact when a lazy step has accumulated enough debt to warrant a rebuild.

```
lemma bp-ordered-invariant-invariant:
  assumes bp-ordered-invariant P
  shows bp-invariant P
  using assms unfolding bp-ordered-invariant-def by blast

lemma bp-ordered-invariant-boundaries-state-ok:
  assumes bp-ordered-invariant P
  shows bp-bucket-boundaries-state-ok P
  using assms unfolding bp-ordered-invariant-def by blast

lemma bp-invariant-lazy-invariant:
  assumes inv: bp-invariant P
  shows bp-lazy-invariant P
proof −
  have lazy-sizes: bp-lazy-bucket-sizes-ok P
  proof −
    have ∀ b∈set (bp-buckets P).
      length (bp-bucket-entries b) ≤ 2 * bp-block-size P
```

```
    using inv unfolding bp-invariant-def bp-bucket-sizes-ok-def
    by auto
  then show ?thesis
    unfolding bp-lazy-bucket-sizes-ok-def .
 qed
 show ?thesis
  using inv lazy-sizes
  unfolding bp-invariant-def bp-lazy-invariant-def
  by blast
qed

lemma bp-ordered-invariant-lazy-ordered-invariant:
 assumes ord: bp-ordered-invariant P
 shows bp-lazy-ordered-invariant P
 using bp-invariant-lazy-invariant[OF bp-ordered-invariant-invariant[OF ord]]
  bp-ordered-invariant-boundaries-state-ok[OF ord]
 unfolding bp-lazy-ordered-invariant-def by blast

lemma bp-lazy-ordered-invariant-lazy-invariant:
 assumes bp-lazy-ordered-invariant P
 shows bp-lazy-invariant P
 using assms unfolding bp-lazy-ordered-invariant-def by blast

lemma bp-lazy-ordered-invariant-boundaries-state-ok:
 assumes bp-lazy-ordered-invariant P
 shows bp-bucket-boundaries-state-ok P
 using assms unfolding bp-lazy-ordered-invariant-def by blast

lemma bp-rebucket-invariant-from-lazy:
 assumes inv: bp-lazy-invariant P
 shows bp-invariant (bp-rebucket P)
proof −
 let ?M = bp-block-size P
 let ?xs = bp-entries P
 have M-pos: 0 < ?M
  using inv unfolding bp-lazy-invariant-def by blast
 have distinct-xs: distinct (map fst ?xs)
  using inv unfolding bp-lazy-invariant-def bp-distinct-keys-def by blast
 have vals:
  ∀ p∈set ?xs. bp-values P (fst p) = snd p
  using inv unfolding bp-lazy-invariant-def bp-values-consistent-def
  by blast
 have flat:
  bp-bucket-entries-flat (bp-bucketize-entries ?M ?xs) = sort-key snd ?xs
  by (rule bp-bucket-entries-flat-bucketize-entries[OF M-pos])
 have flat-rebucket:
  bp-entries (bp-rebucket P) = sort-key snd ?xs
  using flat unfolding bp-rebucket-def bp-entries-def by simp
 have sizes-ok:
```

```
    ∀ b∈set (bp-bucketize-entries ?M ?xs).
      length (bp-bucket-entries b) ≤ ?M
    by (rule bp-bucketize-entries-sizes-ok[OF M-pos])
  have markers-sorted:
    sorted-wrt (λb c. bp-marker b ≤ bp-marker c)
      (bp-bucketize-entries ?M ?xs)
    by (rule bp-bucketize-entries-markers-sorted[OF M-pos])
  have markers-lower:
    ∀ b∈set (bp-bucketize-entries ?M ?xs).
      ∀ p∈set (bp-bucket-entries b). bp-marker b ≤ snd p
    by (rule bp-bucketize-entries-markers-lower-bound[OF M-pos])
  have set-flat:
    set (bp-bucket-entries-flat (bp-bucketize-entries ?M ?xs)) = set ?xs
    by (rule set-bp-bucket-entries-flat-bucketize-entries[OF M-pos])
  show ?thesis
    unfolding bp-invariant-def
  proof (intro conjI)
    show 0 < bp-block-size (bp-rebucket P)
      using M-pos unfolding bp-rebucket-def by simp
    show bp-distinct-keys (bp-rebucket P)
      using distinct-map-fst-sort-key[OF distinct-xs, of snd]
      unfolding bp-distinct-keys-def flat-rebucket by simp
    show bp-bucket-sizes-ok (bp-rebucket P)
      using sizes-ok
      unfolding bp-rebucket-def bp-bucket-sizes-ok-def by simp
    show bp-bucket-markers-sorted (bp-rebucket P)
      using markers-sorted
      unfolding bp-rebucket-def bp-bucket-markers-sorted-def by simp
    show bp-bucket-markers-lower-bound (bp-rebucket P)
      using markers-lower
      unfolding bp-rebucket-def bp-bucket-markers-lower-bound-def by simp
    show bp-values-consistent (bp-rebucket P)
      using vals set-flat
      unfolding bp-rebucket-def bp-values-consistent-def bp-entries-def
        bp-entry-keys-def
      by auto
  qed
qed

lemma bp-rebucket-boundaries-state-ok-from-lazy:
  assumes inv: bp-lazy-invariant P
  shows bp-bucket-boundaries-state-ok (bp-rebucket P)
proof −
  have M-pos: 0 < bp-block-size P
    using inv unfolding bp-lazy-invariant-def by blast
  show ?thesis
    using bp-bucketize-entries-boundaries-ok[OF M-pos, of bp-entries P]
    unfolding bp-bucket-boundaries-state-ok-def bp-rebucket-def by simp
qed
```

```
lemma bp-rebucket-ordered-invariant-from-lazy:
  assumes inv: bp-lazy-invariant P
  shows bp-ordered-invariant (bp-rebucket P)
  unfolding bp-ordered-invariant-def
  using bp-rebucket-invariant-from-lazy[OF inv]
    bp-rebucket-boundaries-state-ok-from-lazy[OF inv]
  by blast

lemma set-bp-bucket-entries-flat-local-insert-bucket:
  assumes 0 < M
  shows set (bp-bucket-entries-flat (bp-local-insert-bucket M p bs)) =
    insert p (set (bp-bucket-entries-flat bs))
  using assms
proof (induction bs arbitrary: p)
  case Nil
  have set (bp-bucket-entries-flat (bp-bucketize-entries M [p])) = set [p]
    by (rule set-bp-bucket-entries-flat-bucketize-entries[OF Nil.prems])
  then show ?case
    by (simp add: bp-bucket-entries-flat-def)
next
  case (Cons b bs)
  note IH = Cons.IH
  note prems = Cons.prems
  show ?case
  proof (cases bs)
    case Nil
    have single:
      set (bp-bucket-entries-flat (bp-bucketize-entries M [p])) = {p}
      using set-bp-bucket-entries-flat-bucketize-entries[OF prems, of [p]]
      by simp
    have inserted:
      set (bp-bucket-entries-flat
        (bp-bucketize-entries M (p # bp-bucket-entries b))) =
       insert p (set (bp-bucket-entries b))
      using set-bp-bucket-entries-flat-bucketize-entries
        [OF prems, of p # bp-bucket-entries b]
      by simp
    then show ?thesis
      using Nil single inserted
      by (auto simp: bp-bucket-entries-flat-def)
  next
    case (Cons c cs)
    have single:
      set (bp-bucket-entries-flat (bp-bucketize-entries M [p])) = {p}
      using set-bp-bucket-entries-flat-bucketize-entries[OF prems, of [p]]
      by simp
    have inserted:
      set (bp-bucket-entries-flat
```

```
      (bp-bucketize-entries M (p # bp-bucket-entries b))) =
     insert p (set (bp-bucket-entries b))
    using set-bp-bucket-entries-flat-bucketize-entries
     [OF prems, of p # bp-bucket-entries b]
    by simp
  have tail:
    set (bp-bucket-entries-flat (bp-local-insert-bucket M p bs)) =
     insert p (set (bp-bucket-entries-flat bs))
    by (rule IH[OF prems])
  then show ?thesis
    using Cons single inserted tail
    by (auto simp: bp-bucket-entries-flat-def)
 qed
qed

lemma set-bp-bucket-entries-flat-lazy-bucket-insert-entries:
 assumes 0 < M
 shows set (bp-bucket-entries-flat
     (bp-lazy-bucket-insert-entries M p b)) =
   insert p (set (bp-bucket-entries b))
proof (cases length (bp-bucket-entries b) < 2 * M)
 case True
 then show ?thesis
   unfolding bp-lazy-bucket-insert-entries-def bp-bucket-entries-flat-def
   by simp
next
 case False
 have set (bp-bucket-entries-flat
     (bp-bucketize-entries M (p # bp-bucket-entries b))) =
   set (p # bp-bucket-entries b)
   by (rule set-bp-bucket-entries-flat-bucketize-entries[OF assms])
 then show ?thesis
   using False unfolding bp-lazy-bucket-insert-entries-def by simp
qed

lemma set-bp-bucket-entries-flat-lazy-insert-bucket:
 assumes 0 < M
 shows set (bp-bucket-entries-flat (bp-lazy-insert-bucket M p bs)) =
   insert p (set (bp-bucket-entries-flat bs))
 using assms
proof (induction bs arbitrary: p)
 case Nil
 have set (bp-bucket-entries-flat (bp-bucketize-entries M [p])) = set [p]
   by (rule set-bp-bucket-entries-flat-bucketize-entries[OF Nil.prems])
 then show ?case
   by (simp add: bp-bucket-entries-flat-def)
next
 case (Cons b bs)
 note IH = Cons.IH
```

```
  note M-pos = Cons.prems
  show ?case
  proof (cases bs)
    case Nil
    have single:
      set (bp-bucket-entries-flat (bp-bucketize-entries M [p])) = {p}
      using set-bp-bucket-entries-flat-bucketize-entries[OF M-pos, of [p]]
      by simp
    have inserted:
      set (bp-bucket-entries-flat
        (bp-lazy-bucket-insert-entries M p b)) =
       insert p (set (bp-bucket-entries b))
      by (rule set-bp-bucket-entries-flat-lazy-bucket-insert-entries
          [OF M-pos])
    show ?thesis
      using Nil single inserted by (auto simp: bp-bucket-entries-flat-def)
  next
    case (Cons c cs)
    have single:
      set (bp-bucket-entries-flat (bp-bucketize-entries M [p])) = {p}
      using set-bp-bucket-entries-flat-bucketize-entries[OF M-pos, of [p]]
      by simp
    have inserted:
      set (bp-bucket-entries-flat
        (bp-lazy-bucket-insert-entries M p b)) =
       insert p (set (bp-bucket-entries b))
      by (rule set-bp-bucket-entries-flat-lazy-bucket-insert-entries
          [OF M-pos])
    have tail:
      set (bp-bucket-entries-flat (bp-lazy-insert-bucket M p bs)) =
       insert p (set (bp-bucket-entries-flat bs))
      by (rule IH[OF M-pos])
    show ?thesis
      using Cons single inserted tail
      by (auto simp: bp-bucket-entries-flat-def)
  qed
qed

lemma length-bp-bucket-entries-flat-lazy-bucket-insert-entries [simp]:
  assumes 0 < M
  shows length (bp-bucket-entries-flat
      (bp-lazy-bucket-insert-entries M p b)) =
    Suc (length (bp-bucket-entries b))
proof (cases length (bp-bucket-entries b) < 2 * M)
  case True
  then show ?thesis
    unfolding bp-lazy-bucket-insert-entries-def bp-bucket-entries-flat-def
    by simp
next
```

```
  case False
  have length (bp-bucket-entries-flat
      (bp-bucketize-entries M (p # bp-bucket-entries b))) =
    length (p # bp-bucket-entries b)
    using bp-bucket-entries-flat-bucketize-entries
      [OF assms, of p # bp-bucket-entries b]
    by simp
  then show ?thesis
    using False unfolding bp-lazy-bucket-insert-entries-def by simp
qed

lemma length-bp-bucket-entries-flat-lazy-insert-bucket [simp]:
  assumes 0 < M
  shows length (bp-bucket-entries-flat (bp-lazy-insert-bucket M p bs)) =
    Suc (length (bp-bucket-entries-flat bs))
  using assms
proof (induction bs arbitrary: p)
  case Nil
  have len:
    length (bp-bucket-entries-flat (bp-bucketize-entries M [p])) =
      length [p]
    using bp-bucket-entries-flat-bucketize-entries[OF Nil.prems, of [p]]
    by simp
  show ?case
    using len by (simp add: bp-bucket-entries-flat-def)
next
  case (Cons b bs)
  note IH = Cons.IH
  note M-pos = Cons.prems
  show ?case
  proof (cases bs)
    case Nil
    have len-single:
      length (bp-bucket-entries-flat (bp-bucketize-entries M [p])) =
        length [p]
      using bp-bucket-entries-flat-bucketize-entries[OF M-pos, of [p]]
      by simp
    have len-inserted:
      length (bp-bucket-entries-flat
          (bp-lazy-bucket-insert-entries M p b)) =
        Suc (length (bp-bucket-entries b))
      by (rule length-bp-bucket-entries-flat-lazy-bucket-insert-entries
          [OF M-pos])
    show ?thesis
      using Nil len-single len-inserted
      by (auto simp: bp-bucket-entries-flat-def)
  next
    case (Cons c cs)
    have len-single:
```

```
      length (bp-bucket-entries-flat (bp-bucketize-entries M [p])) =
        length [p]
      using bp-bucket-entries-flat-bucketize-entries[OF M-pos, of [p]]
      by simp
    have len-inserted:
      length (bp-bucket-entries-flat
          (bp-lazy-bucket-insert-entries M p b)) =
        Suc (length (bp-bucket-entries b))
      by (rule length-bp-bucket-entries-flat-lazy-bucket-insert-entries
          [OF M-pos])
    have tail-len:
      length (bp-bucket-entries-flat
          (bp-lazy-insert-bucket M p (c # cs))) =
        Suc (length (bp-bucket-entries-flat (c # cs)))
      using IH[OF M-pos, of p] Cons by simp
    show ?thesis
    proof (cases snd p < bp-marker b)
      case True
      then show ?thesis
        using Cons len-single by (simp add: bp-bucket-entries-flat-def)
    next
      case False
      show ?thesis
      proof (cases snd p < bp-marker c)
        case True
        then show ?thesis
          using Cons False len-inserted
          by (simp add: bp-bucket-entries-flat-def)
      next
        case False
        then show ?thesis
          using Cons ‹¬ snd p < bp-marker b› tail-len
          by (simp add: bp-bucket-entries-flat-def)
      qed
    qed
  qed
qed

lemma bp-lazy-insert-bucket-distinct-keys:
  assumes M-pos: 0 < M
    and distinct: distinct (map fst (bp-bucket-entries-flat bs))
    and fresh: fst p ∉ bp-entry-keys (bp-bucket-entries-flat bs)
  shows distinct
    (map fst (bp-bucket-entries-flat (bp-lazy-insert-bucket M p bs)))
proof (rule card-distinct)
  let ?old = map fst (bp-bucket-entries-flat bs)
  let ?new =
    map fst (bp-bucket-entries-flat (bp-lazy-insert-bucket M p bs))
  have set-new: set ?new = insert (fst p) (set ?old)
```

```
    using set-bp-bucket-entries-flat-lazy-insert-bucket[OF M-pos, of p bs]
    by auto
  have fresh-set: fst p ∉ set ?old
    using fresh unfolding bp-entry-keys-def by simp
  have card (set ?new) = Suc (card (set ?old))
    unfolding set-new using fresh-set by simp
  also have ... = Suc (length ?old)
    using distinct-card[OF distinct] by simp
  also have ... = length ?new
    using length-bp-bucket-entries-flat-lazy-insert-bucket[OF M-pos, of p bs]
    by simp
  finally show card (set ?new) = length ?new .
qed

lemma bp-lazy-bucket-insert-entries-sizes-ok:
  assumes M-pos: 0 < M
    and size: length (bp-bucket-entries b) ≤ 2 * M
  shows ∀ c∈set (bp-lazy-bucket-insert-entries M p b).
    length (bp-bucket-entries c) ≤ 2 * M
proof (cases length (bp-bucket-entries b) < 2 * M)
  case True
  then have length (p # bp-bucket-entries b) ≤ 2 * M
    by simp
  then show ?thesis
    using True unfolding bp-lazy-bucket-insert-entries-def by simp
next
  case False
  have strict-sizes:
    ∀ c∈set (bp-bucketize-entries M (p # bp-bucket-entries b)).
      length (bp-bucket-entries c) ≤ M
    by (rule bp-bucketize-entries-sizes-ok[OF M-pos])
  have ∀ c∈set (bp-bucketize-entries M (p # bp-bucket-entries b)).
      length (bp-bucket-entries c) ≤ 2 * M
    using strict-sizes by auto
  then show ?thesis
    using False unfolding bp-lazy-bucket-insert-entries-def by simp
qed

lemma bp-lazy-insert-bucket-sizes-ok:
  assumes M-pos: 0 < M
    and sizes: ∀ b∈set bs. length (bp-bucket-entries b) ≤ 2 * M
  shows ∀ b∈set (bp-lazy-insert-bucket M p bs).
    length (bp-bucket-entries b) ≤ 2 * M
  using sizes
proof (induction bs arbitrary: p)
  case Nil
  have strict:
    ∀ b∈set (bp-bucketize-entries M [p]).
      length (bp-bucket-entries b) ≤ M
```

```
    by (rule bp-bucketize-entries-sizes-ok[OF M-pos])
  then show ?case
    using M-pos by auto
next
  case (Cons b bs)
  note IH = Cons.IH
  note prems = Cons.prems
  show ?case
  proof (cases bs)
    case Nil
    have single:
      ∀ b∈set (bp-bucketize-entries M [p]).
        length (bp-bucket-entries b) ≤ 2 * M
      using bp-bucketize-entries-sizes-ok[OF M-pos, of [p]] by auto
    have inserted:
      ∀ c∈set (bp-lazy-bucket-insert-entries M p b).
        length (bp-bucket-entries c) ≤ 2 * M
      by (rule bp-lazy-bucket-insert-entries-sizes-ok[OF M-pos])
        (use prems Nil in simp)
    have b-size: length (bp-bucket-entries b) ≤ 2 * M
      using prems by simp
    then show ?thesis
      using Nil single inserted b-size by auto
  next
    case (Cons c cs)
    have tail-sizes:
      ∀ b∈set bs. length (bp-bucket-entries b) ≤ 2 * M
      using prems by simp
    have single:
      ∀ b∈set (bp-bucketize-entries M [p]).
        length (bp-bucket-entries b) ≤ 2 * M
      using bp-bucketize-entries-sizes-ok[OF M-pos, of [p]] by auto
    have inserted:
      ∀ d∈set (bp-lazy-bucket-insert-entries M p b).
        length (bp-bucket-entries d) ≤ 2 * M
      by (rule bp-lazy-bucket-insert-entries-sizes-ok[OF M-pos])
        (use prems Cons in simp)
    show ?thesis
      using IH[of p, OF tail-sizes] prems Cons single inserted by auto
  qed
qed

lemma bp-local-insert-bucket-sizes-ok:
  assumes M-pos: 0 < M
    and sizes: ∀ b∈set bs. length (bp-bucket-entries b) ≤ M
  shows ∀ b∈set (bp-local-insert-bucket M p bs).
    length (bp-bucket-entries b) ≤ M
  using sizes
proof (induction bs arbitrary: p)
```

```
  case Nil
  have ∀ b∈set (bp-bucketize-entries M [p]).
    length (bp-bucket-entries b) ≤ M
    by (rule bp-bucketize-entries-sizes-ok[OF M-pos])
  then show ?case
    by simp
next
  case (Cons b bs)
  note IH = Cons.IH
  note prems = Cons.prems
  show ?case
  proof (cases bs)
    case Nil
    then show ?thesis
      using prems bp-bucketize-entries-sizes-ok[OF M-pos, of [p]]
        bp-bucketize-entries-sizes-ok[OF M-pos, of p # bp-bucket-entries b]
      by auto
  next
    case (Cons c cs)
    have tail-sizes:
      ∀ b∈set bs. length (bp-bucket-entries b) ≤ M
      using prems by simp
    show ?thesis
      using IH[of p, OF tail-sizes] prems Cons
        bp-bucketize-entries-sizes-ok[OF M-pos, of [p]]
        bp-bucketize-entries-sizes-ok[OF M-pos, of p # bp-bucket-entries b]
      by auto
  qed
qed

lemma bp-local-insert-bucket-markers-lower-bound:
  assumes M-pos: 0 < M
    and lower: ∀ b∈set bs. ∀ p∈set (bp-bucket-entries b).
      bp-marker b ≤ snd p
  shows ∀ b∈set (bp-local-insert-bucket M p bs).
    ∀ q∈set (bp-bucket-entries b). bp-marker b ≤ snd q
  using lower
proof (induction bs arbitrary: p)
  case Nil
  have ∀ b∈set (bp-bucketize-entries M [p]).
    ∀ q∈set (bp-bucket-entries b). bp-marker b ≤ snd q
    by (rule bp-bucketize-entries-markers-lower-bound[OF M-pos])
  then show ?case
    by simp
next
  case (Cons b bs)
  note IH = Cons.IH
  note prems = Cons.prems
  show ?case
```

```
  proof (cases bs)
    case Nil
    then show ?thesis
      using prems bp-bucketize-entries-markers-lower-bound[OF M-pos, of [p]]
        bp-bucketize-entries-markers-lower-bound
          [OF M-pos, of p # bp-bucket-entries b]
      by auto
  next
    case (Cons c cs)
    have tail-lower:
     ∀ b∈set bs. ∀ p∈set (bp-bucket-entries b).
       bp-marker b ≤ snd p
      using prems by simp
    show ?thesis
      using IH[of p, OF tail-lower] prems Cons
        bp-bucketize-entries-markers-lower-bound[OF M-pos, of [p]]
        bp-bucketize-entries-markers-lower-bound
          [OF M-pos, of p # bp-bucket-entries b]
      by auto
  qed
qed

lemma bp-lazy-bucket-insert-entries-markers-lower-bound:
  assumes M-pos: 0 < M
    and lower: ∀ q∈set (bp-bucket-entries b). bp-marker b ≤ snd q
    and p-ge: bp-marker b ≤ snd p
  shows ∀ c∈set (bp-lazy-bucket-insert-entries M p b).
    ∀ q∈set (bp-bucket-entries c). bp-marker c ≤ snd q
proof (cases length (bp-bucket-entries b) < 2 * M)
  case True
  then show ?thesis
    using lower p-ge unfolding bp-lazy-bucket-insert-entries-def by auto
next
  case False
  have ∀ c∈set (bp-bucketize-entries M (p # bp-bucket-entries b)).
    ∀ q∈set (bp-bucket-entries c). bp-marker c ≤ snd q
    by (rule bp-bucketize-entries-markers-lower-bound[OF M-pos])
  then show ?thesis
    using False unfolding bp-lazy-bucket-insert-entries-def by simp
qed

lemma bp-lazy-insert-bucket-markers-lower-bound:
  assumes M-pos: 0 < M
    and lower: ∀ b∈set bs. ∀ p∈set (bp-bucket-entries b).
      bp-marker b ≤ snd p
  shows ∀ b∈set (bp-lazy-insert-bucket M p bs).
    ∀ q∈set (bp-bucket-entries b). bp-marker b ≤ snd q
  using lower
proof (induction bs arbitrary: p)
```

```
  case Nil
  show ?case
    by (simp add: bp-bucketize-entries-markers-lower-bound[OF M-pos])
next
  case (Cons b bs)
  note IH = Cons.IH
  note prems = Cons.prems
  show ?case
  proof (cases bs)
    case Nil
    have single:
      ∀ b∈set (bp-bucketize-entries M [p]).
        ∀ q∈set (bp-bucket-entries b). bp-marker b ≤ snd q
      by (rule bp-bucketize-entries-markers-lower-bound[OF M-pos])
    have b-lower:
      ∀ q∈set (bp-bucket-entries b). bp-marker b ≤ snd q
      using prems by simp
    have inserted:
      bp-marker b ≤ snd p ⟹
        ∀ c∈set (bp-lazy-bucket-insert-entries M p b).
          ∀ q∈set (bp-bucket-entries c). bp-marker c ≤ snd q
      by (rule bp-lazy-bucket-insert-entries-markers-lower-bound
          [OF M-pos b-lower])
    show ?thesis
    proof (cases snd p < bp-marker b)
      case True
      show ?thesis
        using Nil True single b-lower by auto
    next
      case False
      have p-ge-b: bp-marker b ≤ snd p
        using False by simp
      show ?thesis
        using Nil False inserted[OF p-ge-b] by simp
    qed
  next
    case (Cons c cs)
    have tail-lower:
      ∀ b∈set bs. ∀ p∈set (bp-bucket-entries b).
        bp-marker b ≤ snd p
      using prems by simp
    have single:
      ∀ b∈set (bp-bucketize-entries M [p]).
        ∀ q∈set (bp-bucket-entries b). bp-marker b ≤ snd q
      by (rule bp-bucketize-entries-markers-lower-bound[OF M-pos])
    have b-lower:
      ∀ q∈set (bp-bucket-entries b). bp-marker b ≤ snd q
      using prems by simp
    have inserted:
```

```
      bp-marker b ≤ snd p ⟹
       ∀ d∈set (bp-lazy-bucket-insert-entries M p b).
         ∀ q∈set (bp-bucket-entries d). bp-marker d ≤ snd q
       by (rule bp-lazy-bucket-insert-entries-markers-lower-bound
         [OF M-pos b-lower])
     have tail: ∀ b∈set (bp-lazy-insert-bucket M p bs).
         ∀ q∈set (bp-bucket-entries b). bp-marker b ≤ snd q
       by (rule IH[OF tail-lower])
     show ?thesis
     proof (cases snd p < bp-marker b)
       case True
       have result:
         bp-lazy-insert-bucket M p (b # bs) =
           bp-bucketize-entries M [p] @ b # bs
         using True Cons by simp
       show ?thesis
         unfolding result
         using single prems Cons by auto
     next
       case False
       have p-ge-b: bp-marker b ≤ snd p
         using False by simp
       show ?thesis
       proof (cases snd p < bp-marker c)
         case True
         have result:
         bp-lazy-insert-bucket M p (b # bs) =
             bp-lazy-bucket-insert-entries M p b @ bs
           using False True Cons by simp
         show ?thesis
           unfolding result
           using inserted[OF p-ge-b] prems Cons by auto
       next
         case False
         have result:
           bp-lazy-insert-bucket M p (b # bs) =
             b # bp-lazy-insert-bucket M p bs
           using ‹¬ snd p < bp-marker b› False Cons by simp
         show ?thesis
           unfolding result
           using tail prems Cons by auto
       qed
     qed
   qed
 qed

lemma bp-bucketize-entries-markers-le:
   fixes beta :: real
   assumes ∀ q∈set xs. snd q ≤ beta
```

```
  shows ∀ b∈set (bp-bucketize-entries M xs). bp-marker b ≤ beta
proof
  fix b
  assume b: b ∈ set (bp-bucketize-entries M xs)
  obtain q where q: q ∈ set xs and marker: bp-marker b = snd q
    using bp-bucketize-entries-marker-in-set[OF b] by auto
  have snd q ≤ beta
    using assms q by blast
  show bp-marker b ≤ beta
    using marker ‹snd q ≤ beta› by simp
qed

lemma bp-bucketize-entries-markers-ge:
  fixes alpha :: real
  assumes ∀ q∈set xs. alpha ≤ snd q
  shows ∀ b∈set (bp-bucketize-entries M xs). alpha ≤ bp-marker b
proof
  fix b
  assume b: b ∈ set (bp-bucketize-entries M xs)
  obtain q where q: q ∈ set xs and marker: bp-marker b = snd q
    using bp-bucketize-entries-marker-in-set[OF b] by auto
  have alpha ≤ snd q
    using assms q by blast
  show alpha ≤ bp-marker b
    using marker ‹alpha ≤ snd q› by simp
qed

lemma bp-bucketize-entries-entry-in-source:
  assumes M-pos: 0 < M
    and b: b ∈ set (bp-bucketize-entries M xs)
    and q: q ∈ set (bp-bucket-entries b)
  shows q ∈ set xs
proof −
  have q ∈ set (bp-bucket-entries-flat (bp-bucketize-entries M xs))
    using b q unfolding bp-bucket-entries-flat-def by auto
  then show ?thesis
    using set-bp-bucket-entries-flat-bucketize-entries[OF M-pos, of xs]
    by simp
qed

lemma bp-bucketize-entries-nonempty:
  assumes 0 < M and xs ≠ []
  shows bp-bucketize-entries M xs ≠ []
proof
  assume empty: bp-bucketize-entries M xs = []
  have set (bp-bucket-entries-flat (bp-bucketize-entries M xs)) = set xs
    by (rule set-bp-bucket-entries-flat-bucketize-entries[OF assms(1)])
  then have set xs = {}
    using empty unfolding bp-bucket-entries-flat-def by simp
```

```
  then show False
    using assms(2) by auto
qed

lemma bp-local-insert-bucket-nonempty:
  assumes 0 < M
  shows bp-local-insert-bucket M p bs ≠ []
  using assms
proof (induction bs arbitrary: p)
  case Nil
  then show ?case
    by (simp add: bp-bucketize-entries-nonempty)
next
  case (Cons b bs)
  then show ?case
  proof (cases bs)
    case Nil
    then show ?thesis
      using Cons.prems by (simp add: bp-bucketize-entries-nonempty)
  next
    case (Cons c cs)
    then show ?thesis
      using Cons by (simp add: bp-bucketize-entries-nonempty)
  qed
qed

lemma bp-bucket-boundaries-ok-append:
  assumes left: bp-bucket-boundaries-ok xs
    and right: bp-bucket-boundaries-ok ys
    and cross: ys ≠ [] ⟹
      ∀ b∈set xs. ∀ p∈set (bp-bucket-entries b).
        snd p ≤ bp-marker (hd ys)
  shows bp-bucket-boundaries-ok (xs @ ys)
  using left cross
proof (induction xs)
  case Nil
  then show ?case
    using right by simp
next
  case (Cons b xs)
  note outer-left = Cons.prems(1)
  note outer-cross = Cons.prems(2)
  note outer-IH = Cons.IH
  show ?case
  proof (cases xs)
    case Nil
    show ?thesis
    proof (cases ys)
      case Nil
```

```
      then show ?thesis
        using ‹xs = []› by simp
    next
      case (Cons c cs)
      have cross-all:
        ∀ d∈set (b # xs). ∀ p∈set (bp-bucket-entries d).
          snd p ≤ bp-marker (hd ys)
        using outer-cross Cons by simp
      have ∀ p∈set (bp-bucket-entries b). snd p ≤ bp-marker c
        using cross-all Cons unfolding ‹xs = []› by simp
      then show ?thesis
        using right Cons ‹xs = []› by simp
    qed
  next
    case (Cons c cs)
    have head-boundary:
      ∀ p∈set (bp-bucket-entries b). snd p ≤ bp-marker c
      using outer-left Cons by simp
    have tail-boundaries: bp-bucket-boundaries-ok (xs @ ys)
    proof (rule outer-IH)
      show bp-bucket-boundaries-ok xs
        using outer-left Cons by simp
      show ys ≠ [] ⟹
        ∀ b∈set xs. ∀ p∈set (bp-bucket-entries b).
          snd p ≤ bp-marker (hd ys)
        using outer-cross by simp
    qed
    show ?thesis
      using Cons head-boundary tail-boundaries by simp
  qed
qed

lemma bp-bucket-boundaries-ok-bucketize-append-Cons:
  assumes M-pos: 0 < M
    and tail: bp-bucket-boundaries-ok (c # bs)
    and upper: ∀ q∈set xs. snd q ≤ bp-marker c
  shows bp-bucket-boundaries-ok (bp-bucketize-entries M xs @ c # bs)
proof (rule bp-bucket-boundaries-ok-append)
  show bp-bucket-boundaries-ok (bp-bucketize-entries M xs)
    by (rule bp-bucketize-entries-boundaries-ok[OF M-pos])
  show bp-bucket-boundaries-ok (c # bs)
    by (rule tail)
  show c # bs ≠ [] ⟹
    ∀ b∈set (bp-bucketize-entries M xs).
      ∀ p∈set (bp-bucket-entries b). snd p ≤ bp-marker (hd (c # bs))
  proof
    fix b
    assume b: b ∈ set (bp-bucketize-entries M xs)
    show ∀ p∈set (bp-bucket-entries b). snd p ≤ bp-marker (hd (c # bs))
```

```
    proof
      fix p
      assume p: p ∈ set (bp-bucket-entries b)
      have p ∈ set xs
        by (rule bp-bucketize-entries-entry-in-source[OF M-pos b p])
      then show snd p ≤ bp-marker (hd (c # bs))
        using upper by simp
    qed
  qed
qed

lemma bp-lazy-bucket-insert-entries-nonempty:
  assumes M-pos: 0 < M
  shows bp-lazy-bucket-insert-entries M p b ≠ []
proof (cases length (bp-bucket-entries b) < 2 * M)
  case True
  then show ?thesis
    unfolding bp-lazy-bucket-insert-entries-def by simp
next
  case False
  have bp-bucketize-entries M (p # bp-bucket-entries b) ≠ []
    by (rule bp-bucketize-entries-nonempty[OF M-pos]) simp
  then show ?thesis
    using False unfolding bp-lazy-bucket-insert-entries-def by simp
qed

lemma bp-lazy-insert-bucket-nonempty:
  assumes M-pos: 0 < M
  shows bp-lazy-insert-bucket M p bs ≠ []
  using assms
proof (induction bs arbitrary: p)
  case Nil
  then show ?case
    by (simp add: bp-bucketize-entries-nonempty)
next
  case (Cons b bs)
  note IH = Cons.IH
  note M-pos = Cons.prems
  show ?case
  proof (cases bs)
    case Nil
    show ?thesis
    proof (cases snd p < bp-marker b)
      case True
      then show ?thesis
        using Nil by (simp add: bp-bucketize-entries-nonempty[OF M-pos])
    next
      case False
      then show ?thesis
```

```
      using Nil bp-lazy-bucket-insert-entries-nonempty[OF M-pos, of p b]
      by simp
    qed
  next
    case (Cons c cs)
    show ?thesis
    proof (cases snd p < bp-marker b)
      case True
      then show ?thesis
        using Cons by (simp add: bp-bucketize-entries-nonempty[OF M-pos])
    next
      case False
      show ?thesis
      proof (cases snd p < bp-marker c)
        case True
        then show ?thesis
          using Cons False bp-lazy-bucket-insert-entries-nonempty
            [OF M-pos, of p b]
          by simp
      next
        case False
        then show ?thesis
          using Cons ‹¬ snd p < bp-marker b› IH[OF M-pos, of p]
          by simp
      qed
    qed
  qed
qed

lemma bp-lazy-bucket-insert-entries-markers-sorted:
  assumes M-pos: 0 < M
  shows sorted-wrt (λb c. bp-marker b ≤ bp-marker c)
    (bp-lazy-bucket-insert-entries M p b)
proof (cases length (bp-bucket-entries b) < 2 * M)
  case True
  then show ?thesis
    unfolding bp-lazy-bucket-insert-entries-def by simp
next
  case False
  have sorted-wrt (λb c. bp-marker b ≤ bp-marker c)
      (bp-bucketize-entries M (p # bp-bucket-entries b))
    by (rule bp-bucketize-entries-markers-sorted[OF M-pos])
  then show ?thesis
    using False unfolding bp-lazy-bucket-insert-entries-def by simp
qed

lemma bp-lazy-bucket-insert-entries-markers-ge:
  fixes alpha :: real
  assumes M-pos: 0 < M
```

```
    and lower: ∀ q∈set (bp-bucket-entries b). bp-marker b ≤ snd q
    and p-ge: bp-marker b ≤ snd p
    and alpha-le: alpha ≤ bp-marker b
  shows ∀ c∈set (bp-lazy-bucket-insert-entries M p b).
    alpha ≤ bp-marker c
proof (cases length (bp-bucket-entries b) < 2 * M)
  case True
  then show ?thesis
    using alpha-le unfolding bp-lazy-bucket-insert-entries-def by simp
next
  case False
  have entries-ge:
    ∀ q∈set (p # bp-bucket-entries b). alpha ≤ snd q
    using lower p-ge alpha-le by auto
  have ∀ c∈set (bp-bucketize-entries M (p # bp-bucket-entries b)).
    alpha ≤ bp-marker c
    by (rule bp-bucketize-entries-markers-ge[OF entries-ge])
  then show ?thesis
    using False unfolding bp-lazy-bucket-insert-entries-def by simp
qed

lemma bp-lazy-bucket-insert-entries-markers-le:
  fixes beta :: real
  assumes M-pos: 0 < M
    and marker-le: bp-marker b ≤ beta
    and upper: ∀ q∈set (p # bp-bucket-entries b). snd q ≤ beta
  shows ∀ c∈set (bp-lazy-bucket-insert-entries M p b).
    bp-marker c ≤ beta
proof (cases length (bp-bucket-entries b) < 2 * M)
  case True
  then show ?thesis
    using marker-le unfolding bp-lazy-bucket-insert-entries-def by simp
next
  case False
  have ∀ c∈set (bp-bucketize-entries M (p # bp-bucket-entries b)).
    bp-marker c ≤ beta
    by (rule bp-bucketize-entries-markers-le[OF upper])
  then show ?thesis
    using False unfolding bp-lazy-bucket-insert-entries-def by simp
qed

lemma bp-lazy-bucket-insert-entries-entries-le:
  fixes beta :: real
  assumes M-pos: 0 < M
    and upper: ∀ q∈set (p # bp-bucket-entries b). snd q ≤ beta
  shows ∀ c∈set (bp-lazy-bucket-insert-entries M p b).
    ∀ q∈set (bp-bucket-entries c). snd q ≤ beta
proof (cases length (bp-bucket-entries b) < 2 * M)
  case True
```

```
  show ?thesis
  proof
    fix c
    assume c: c ∈ set (bp-lazy-bucket-insert-entries M p b)
    show ∀ q∈set (bp-bucket-entries c). snd q ≤ beta
    proof
      fix q
      assume q: q ∈ set (bp-bucket-entries c)
      have q ∈ set (p # bp-bucket-entries b)
        using True c q unfolding bp-lazy-bucket-insert-entries-def by auto
      then show snd q ≤ beta
        using upper by blast
    qed
  qed
next
  case False
  show ?thesis
  proof
    fix c
    assume c:
      c ∈ set (bp-lazy-bucket-insert-entries M p b)
    show ∀ q∈set (bp-bucket-entries c). snd q ≤ beta
    proof
      fix q
      assume q: q ∈ set (bp-bucket-entries c)
      have c-bucketize:
        c ∈ set (bp-bucketize-entries M (p # bp-bucket-entries b))
        using c False unfolding bp-lazy-bucket-insert-entries-def by simp
      have q ∈ set (p # bp-bucket-entries b)
        by (rule bp-bucketize-entries-entry-in-source
            [OF M-pos c-bucketize q])
      then show snd q ≤ beta
        using upper by blast
    qed
  qed
qed

lemma bp-lazy-bucket-insert-entries-boundaries-ok:
  assumes M-pos: 0 < M
  shows bp-bucket-boundaries-ok (bp-lazy-bucket-insert-entries M p b)
proof (cases length (bp-bucket-entries b) < 2 * M)
  case True
  then show ?thesis
    unfolding bp-lazy-bucket-insert-entries-def by simp
next
  case False
  have bp-bucket-boundaries-ok
      (bp-bucketize-entries M (p # bp-bucket-entries b))
    by (rule bp-bucketize-entries-boundaries-ok[OF M-pos])
```

```
  then show ?thesis
    using False unfolding bp-lazy-bucket-insert-entries-def by simp
qed

lemma bp-lazy-insert-bucket-markers-ge-hd:
  assumes M-pos: 0 < M
    and sorted: sorted-wrt (λb c. bp-marker b ≤ bp-marker c) bs
    and lower: ∀ b∈set bs. ∀ q∈set (bp-bucket-entries b).
      bp-marker b ≤ snd q
    and nonempty: bs ≠ []
    and p-ge: bp-marker (hd bs) ≤ snd p
  shows ∀ b∈set (bp-lazy-insert-bucket M p bs).
    bp-marker (hd bs) ≤ bp-marker b
  using sorted lower nonempty p-ge
proof (induction bs arbitrary: p)
  case Nil
  then show ?case
    by simp
next
  case (Cons b bs)
  note IH = Cons.IH
  note sorted = Cons.prems(1)
  note lower = Cons.prems(2)
  note p-ge = Cons.prems(4)
  show ?case
  proof (cases bs)
    case Nil
    have p-ge-b: bp-marker b ≤ snd p
      using p-ge by simp
    have inserted-ge:
      ∀ c∈set (bp-lazy-bucket-insert-entries M p b).
        bp-marker b ≤ bp-marker c
    proof (rule bp-lazy-bucket-insert-entries-markers-ge
        [where alpha = bp-marker b, OF M-pos - p-ge-b])
      show ∀ q∈set (bp-bucket-entries b). bp-marker b ≤ snd q
        using lower Nil by simp
      show bp-marker b ≤ bp-marker b
        by simp
    qed
    show ?thesis
    proof (cases snd p < bp-marker b)
      case True
      then show ?thesis
        using p-ge by simp
    next
      case False
      then show ?thesis
        using Nil inserted-ge by simp
    qed
```

```
next
  case (Cons c cs)
  have sorted-tail:
    sorted-wrt (λb c. bp-marker b ≤ bp-marker c) (c # cs)
    using sorted Cons by simp
  have lower-tail:
    ∀ b∈set (c # cs). ∀ q∈set (bp-bucket-entries b).
      bp-marker b ≤ snd q
    using lower Cons by simp
  have b-le-c: bp-marker b ≤ bp-marker c
    using sorted Cons by simp
  show ?thesis
  proof (cases snd p < bp-marker b)
    case True
    then show ?thesis
      using p-ge by simp
  next
    case False
    show ?thesis
    proof (cases snd p < bp-marker c)
      case True
      have p-ge-b: bp-marker b ≤ snd p
        using False by simp
      have inserted-ge:
        ∀ d∈set (bp-lazy-bucket-insert-entries M p b).
          bp-marker b ≤ bp-marker d
      proof (rule bp-lazy-bucket-insert-entries-markers-ge
          [where alpha = bp-marker b, OF M-pos])
        show ∀ q∈set (bp-bucket-entries b). bp-marker b ≤ snd q
          using lower Cons by simp
        show bp-marker b ≤ snd p
          by (rule p-ge-b)
        show bp-marker b ≤ bp-marker b
          by simp
      qed
      have tail-ge:
        ∀ d∈set (c # cs). bp-marker b ≤ bp-marker d
        using sorted Cons by auto
      show ?thesis
        using Cons False True inserted-ge tail-ge by auto
    next
      case False
      have p-ge-c: bp-marker c ≤ snd p
        using False by simp
      have tail-ge-c:
        ∀ d∈set (bp-lazy-insert-bucket M p (c # cs)).
          bp-marker c ≤ bp-marker d
        using IH[of p] sorted-tail lower-tail p-ge-c Cons by auto
      have tail-ge-b:
```

```
        ∀ d∈set (bp-lazy-insert-bucket M p (c # cs)).
          bp-marker b ≤ bp-marker d
        using b-le-c tail-ge-c by force
      show ?thesis
        using Cons ‹¬ snd p < bp-marker b› False tail-ge-b by simp
    qed
  qed
 qed
qed

lemma bp-lazy-insert-bucket-markers-sorted:
 assumes M-pos: 0 < M
   and sorted: sorted-wrt (λb c. bp-marker b ≤ bp-marker c) bs
   and lower: ∀ b∈set bs. ∀ q∈set (bp-bucket-entries b).
     bp-marker b ≤ snd q
   and boundaries: bp-bucket-boundaries-ok bs
 shows sorted-wrt (λb c. bp-marker b ≤ bp-marker c)
   (bp-lazy-insert-bucket M p bs)
 using sorted lower boundaries
proof (induction bs arbitrary: p)
 case Nil
 then show ?case
   by (simp add: bp-bucketize-entries-markers-sorted[OF M-pos])
next
 case (Cons b bs)
 note IH = Cons.IH
 note sorted = Cons.prems(1)
 note lower = Cons.prems(2)
 note boundaries = Cons.prems(3)
 show ?case
 proof (cases bs)
   case Nil
   show ?thesis
   proof (cases snd p < bp-marker b)
     case True
     have left-sorted:
       sorted-wrt (λb c. bp-marker b ≤ bp-marker c)
         (bp-bucketize-entries M [p])
       by (rule bp-bucketize-entries-markers-sorted[OF M-pos])
     have left-le-b:
       ∀ x∈set (bp-bucketize-entries M [p]).
         bp-marker x ≤ bp-marker b
       by (rule bp-bucketize-entries-markers-le) (use True in simp)
     show ?thesis
       using Nil True left-sorted left-le-b
       by (simp add: sorted-wrt-append)
   next
     case False
     show ?thesis
```

```
      using Nil False
        bp-lazy-bucket-insert-entries-markers-sorted[OF M-pos, of p b]
      by simp
  qed
next
  case (Cons c cs)
  have sorted-tail:
    sorted-wrt (λb c. bp-marker b ≤ bp-marker c) (c # cs)
    using sorted Cons by simp
  have lower-tail:
    ∀ b∈set (c # cs). ∀ q∈set (bp-bucket-entries b).
      bp-marker b ≤ snd q
    using lower Cons by simp
  have boundaries-tail: bp-bucket-boundaries-ok (c # cs)
    using boundaries Cons by simp
  have b-le-c: bp-marker b ≤ bp-marker c
    using sorted Cons by simp
  have b-le-all:
    ∀ d∈set (b # c # cs). bp-marker b ≤ bp-marker d
    using sorted Cons by auto
  show ?thesis
  proof (cases snd p < bp-marker b)
    case True
    have left-sorted:
      sorted-wrt (λb c. bp-marker b ≤ bp-marker c)
        (bp-bucketize-entries M [p])
      by (rule bp-bucketize-entries-markers-sorted[OF M-pos])
    have right-sorted:
      sorted-wrt (λb c. bp-marker b ≤ bp-marker c) (b # c # cs)
      using sorted Cons by simp
    have left-le-b:
      ∀ x∈set (bp-bucketize-entries M [p]).
        bp-marker x ≤ bp-marker b
      by (rule bp-bucketize-entries-markers-le) (use True in simp)
    have cross:
      ∀ x∈set (bp-bucketize-entries M [p]).
        ∀ y∈set (b # c # cs). bp-marker x ≤ bp-marker y
      using left-le-b b-le-all by force
    show ?thesis
      using Cons True left-sorted right-sorted cross
      by (simp add: sorted-wrt-append)
  next
    case False
    show ?thesis
    proof (cases snd p < bp-marker c)
      case True
      have left-sorted:
        sorted-wrt (λb c. bp-marker b ≤ bp-marker c)
          (bp-lazy-bucket-insert-entries M p b)
```

```
    by (rule bp-lazy-bucket-insert-entries-markers-sorted[OF M-pos])
  have left-le-c:
    ∀ x∈set (bp-lazy-bucket-insert-entries M p b).
      bp-marker x ≤ bp-marker c
  proof (rule bp-lazy-bucket-insert-entries-markers-le
      [OF M-pos b-le-c])
    show ∀ q∈set (p # bp-bucket-entries b).
      snd q ≤ bp-marker c
      using boundaries Cons True by auto
  qed
  have c-le-all:
    ∀ d∈set (c # cs). bp-marker c ≤ bp-marker d
    using sorted-tail by auto
  have cross:
    ∀ x∈set (bp-lazy-bucket-insert-entries M p b).
      ∀ y∈set (c # cs). bp-marker x ≤ bp-marker y
    using left-le-c c-le-all by force
  show ?thesis
    using Cons False True left-sorted sorted-tail cross
    by (simp add: sorted-wrt-append)
next
  case False
  have p-ge-c: bp-marker c ≤ snd p
    using False by simp
  have tail-sorted-result:
    sorted-wrt (λb c. bp-marker b ≤ bp-marker c)
      (bp-lazy-insert-bucket M p (c # cs))
    using IH[of p] sorted-tail lower-tail boundaries-tail Cons by auto
  have tail-ge-hd:
    ∀ d∈set (bp-lazy-insert-bucket M p (c # cs)).
      bp-marker (hd (c # cs)) ≤ bp-marker d
  proof (rule bp-lazy-insert-bucket-markers-ge-hd
      [where bs = c # cs and p = p and M = M])
    show 0 < M
      by (rule M-pos)
    show sorted-wrt (λb c. bp-marker b ≤ bp-marker c) (c # cs)
      by (rule sorted-tail)
    show ∀ b∈set (c # cs). ∀ q∈set (bp-bucket-entries b).
      bp-marker b ≤ snd q
      by (rule lower-tail)
    show c # cs ≠ []
      by simp
    show bp-marker (hd (c # cs)) ≤ snd p
      using p-ge-c by simp
  qed
  have tail-ge-c:
    ∀ d∈set (bp-lazy-insert-bucket M p (c # cs)).
      bp-marker c ≤ bp-marker d
    using tail-ge-hd by simp
```

```
      have tail-ge-b:
        ∀ d∈set (bp-lazy-insert-bucket M p (c # cs)).
          bp-marker b ≤ bp-marker d
        using b-le-c tail-ge-c by force
      show ?thesis
        using Cons ‹¬ snd p < bp-marker b› False
          tail-sorted-result tail-ge-b by simp
    qed
  qed
 qed
qed

lemma bp-lazy-insert-bucket-boundaries-ok:
  assumes M-pos: 0 < M
    and sorted: sorted-wrt (λb c. bp-marker b ≤ bp-marker c) bs
    and lower: ∀ b∈set bs. ∀ q∈set (bp-bucket-entries b).
      bp-marker b ≤ snd q
    and boundaries: bp-bucket-boundaries-ok bs
  shows bp-bucket-boundaries-ok (bp-lazy-insert-bucket M p bs)
  using sorted lower boundaries
proof (induction bs arbitrary: p)
  case Nil
  then show ?case
    by (simp add: bp-bucketize-entries-boundaries-ok[OF M-pos])
next
  case (Cons b bs)
  note IH = Cons.IH
  note sorted = Cons.prems(1)
  note lower = Cons.prems(2)
  note boundaries = Cons.prems(3)
  show ?case
  proof (cases bs)
    case Nil
    show ?thesis
    proof (cases snd p < bp-marker b)
      case True
      have tail: bp-bucket-boundaries-ok [b]
        by simp
      have upper: ∀ q∈set [p]. snd q ≤ bp-marker b
        using True by simp
      show ?thesis
        using Nil True
        by (simp add: bp-bucket-boundaries-ok-bucketize-append-Cons
            [OF M-pos tail upper])
    next
      case False
      show ?thesis
        using Nil False
        bp-lazy-bucket-insert-entries-boundaries-ok[OF M-pos, of p b]
```

```
    by simp
  qed
next
  case (Cons c cs)
  have sorted-tail:
    sorted-wrt (λb c. bp-marker b ≤ bp-marker c) (c # cs)
    using sorted Cons by simp
  have lower-tail:
    ∀ b∈set (c # cs). ∀ q∈set (bp-bucket-entries b).
      bp-marker b ≤ snd q
    using lower Cons by simp
  have boundaries-tail: bp-bucket-boundaries-ok (c # cs)
    using boundaries Cons by simp
  show ?thesis
  proof (cases snd p < bp-marker b)
    case True
    have tail-all: bp-bucket-boundaries-ok (b # c # cs)
      using boundaries Cons by simp
    have upper: ∀ q∈set [p]. snd q ≤ bp-marker b
      using True by simp
    show ?thesis
      using Cons True
      by (simp add: bp-bucket-boundaries-ok-bucketize-append-Cons
          [where c = b and bs = c # cs and xs = [p],
            OF M-pos tail-all upper])
  next
    case False
    show ?thesis
    proof (cases snd p < bp-marker c)
      case True
      have upper:
        ∀ q∈set (p # bp-bucket-entries b). snd q ≤ bp-marker c
        using boundaries Cons True by auto
      have left-boundaries:
        bp-bucket-boundaries-ok (bp-lazy-bucket-insert-entries M p b)
        by (rule bp-lazy-bucket-insert-entries-boundaries-ok[OF M-pos])
      have cross-entries:
        ∀ d∈set (bp-lazy-bucket-insert-entries M p b).
          ∀ q∈set (bp-bucket-entries d). snd q ≤ bp-marker c
        by (rule bp-lazy-bucket-insert-entries-entries-le
            [OF M-pos upper])
      have append-ok:
        bp-bucket-boundaries-ok
          (bp-lazy-bucket-insert-entries M p b @ c # cs)
      proof (rule bp-bucket-boundaries-ok-append)
        show bp-bucket-boundaries-ok
            (bp-lazy-bucket-insert-entries M p b)
          by (rule left-boundaries)
        show bp-bucket-boundaries-ok (c # cs)
```

```
    by (rule boundaries-tail)
  show c # cs ≠ [] ⟹
    ∀ b∈set (bp-lazy-bucket-insert-entries M p b).
      ∀ p∈set (bp-bucket-entries b).
        snd p ≤ bp-marker (hd (c # cs))
    using cross-entries by simp
 qed
 show ?thesis
  using Cons False True append-ok by simp
next
 case False
 have p-ge-c: bp-marker c ≤ snd p
  using False by simp
 have tail-boundaries:
  bp-bucket-boundaries-ok (bp-lazy-insert-bucket M p (c # cs))
  using IH[of p] sorted-tail lower-tail boundaries-tail Cons by auto
 have tail-ge-hd:
  ∀ d∈set (bp-lazy-insert-bucket M p (c # cs)).
    bp-marker (hd (c # cs)) ≤ bp-marker d
 proof (rule bp-lazy-insert-bucket-markers-ge-hd
    [where bs = c # cs and p = p and M = M])
  show 0 < M
    by (rule M-pos)
  show sorted-wrt (λb c. bp-marker b ≤ bp-marker c) (c # cs)
    by (rule sorted-tail)
  show ∀ b∈set (c # cs). ∀ q∈set (bp-bucket-entries b).
    bp-marker b ≤ snd q
    by (rule lower-tail)
  show c # cs ≠ []
    by simp
  show bp-marker (hd (c # cs)) ≤ snd p
    using p-ge-c by simp
 qed
 have tail-ge-c:
  ∀ d∈set (bp-lazy-insert-bucket M p (c # cs)).
    bp-marker c ≤ bp-marker d
  using tail-ge-hd by simp
 have tail-nonempty-result:
  bp-lazy-insert-bucket M p (c # cs) ≠ []
  by (rule bp-lazy-insert-bucket-nonempty[OF M-pos])
 obtain d ds where tail-eq:
  bp-lazy-insert-bucket M p (c # cs) = d # ds
  using tail-nonempty-result
  by (cases bp-lazy-insert-bucket M p (c # cs)) auto
 have c-le-d: bp-marker c ≤ bp-marker d
  using tail-ge-c tail-eq by simp
 have head-boundary:
  ∀ q∈set (bp-bucket-entries b). snd q ≤ bp-marker d
 proof
```

```
        fix q
        assume q: q ∈ set (bp-bucket-entries b)
        have snd q ≤ bp-marker c
          using boundaries Cons q by auto
        then show snd q ≤ bp-marker d
          using c-le-d by linarith
      qed
      show ?thesis
        using Cons ‹¬ snd p < bp-marker b› False tail-eq
          tail-boundaries head-boundary by simp
    qed
  qed
 qed
qed

lemma bp-lazy-insert-bucket-preserves-bucket-shape:
  assumes M-pos: 0 < M
    and sizes: ∀ b∈set bs. length (bp-bucket-entries b) ≤ 2 * M
    and sorted: sorted-wrt (λb c. bp-marker b ≤ bp-marker c) bs
    and lower: ∀ b∈set bs. ∀ q∈set (bp-bucket-entries b).
      bp-marker b ≤ snd q
    and boundaries: bp-bucket-boundaries-ok bs
  shows (∀ b∈set (bp-lazy-insert-bucket M p bs).
      length (bp-bucket-entries b) ≤ 2 * M) ∧
    sorted-wrt (λb c. bp-marker b ≤ bp-marker c)
      (bp-lazy-insert-bucket M p bs) ∧
    (∀ b∈set (bp-lazy-insert-bucket M p bs).
      ∀ q∈set (bp-bucket-entries b). bp-marker b ≤ snd q) ∧
    bp-bucket-boundaries-ok (bp-lazy-insert-bucket M p bs)
  using bp-lazy-insert-bucket-sizes-ok[OF M-pos sizes, of p]
    bp-lazy-insert-bucket-markers-sorted
      [OF M-pos sorted lower boundaries, of p]
    bp-lazy-insert-bucket-markers-lower-bound[OF M-pos lower, of p]
    bp-lazy-insert-bucket-boundaries-ok
      [OF M-pos sorted lower boundaries, of p]
  by blast

lemma bp-local-insert-bucket-markers-ge-hd:
  assumes M-pos: 0 < M
    and sorted: sorted-wrt (λb c. bp-marker b ≤ bp-marker c) bs
    and lower: ∀ b∈set bs. ∀ q∈set (bp-bucket-entries b).
      bp-marker b ≤ snd q
    and nonempty: bs ≠ []
    and p-ge: bp-marker (hd bs) ≤ snd p
  shows ∀ b∈set (bp-local-insert-bucket M p bs).
    bp-marker (hd bs) ≤ bp-marker b
  using sorted lower nonempty p-ge
proof (induction bs arbitrary: p)
  case Nil
```

```
  then show ?case
    by simp
next
  case (Cons b bs)
  note IH = Cons.IH
  note sorted = Cons.prems(1)
  note lower = Cons.prems(2)
  note p-ge = Cons.prems(4)
  show ?case
  proof (cases bs)
    case Nil
    have markers-ge:
      ∀ c∈set (bp-bucketize-entries M (p # bp-bucket-entries b)).
        bp-marker b ≤ bp-marker c
    proof (rule bp-bucketize-entries-markers-ge)
      show ∀ q∈set (p # bp-bucket-entries b). bp-marker b ≤ snd q
        using lower p-ge by auto
    qed
    show ?thesis
      using Nil p-ge markers-ge by auto
  next
    case (Cons c cs)
    have sorted-tail:
      sorted-wrt (λb c. bp-marker b ≤ bp-marker c) (c # cs)
      using sorted Cons by simp
    have lower-tail:
      ∀ b∈set (c # cs). ∀ q∈set (bp-bucket-entries b).
        bp-marker b ≤ snd q
      using lower Cons by simp
    have b-le-c: bp-marker b ≤ bp-marker c
      using sorted Cons by simp
    show ?thesis
    proof (cases snd p < bp-marker b)
      case True
      then show ?thesis
        using p-ge by simp
    next
      case False
      note not-before-b = False
      show ?thesis
      proof (cases snd p < bp-marker c)
        case True
        have markers-ge:
          ∀ d∈set (bp-bucketize-entries M (p # bp-bucket-entries b)).
            bp-marker b ≤ bp-marker d
        proof (rule bp-bucketize-entries-markers-ge)
          show ∀ q∈set (p # bp-bucket-entries b). bp-marker b ≤ snd q
            using lower p-ge by auto
        qed
```

```
      have tail-ge:
        ∀ d∈set (c # cs). bp-marker b ≤ bp-marker d
        using sorted Cons by auto
      show ?thesis
        using Cons False True markers-ge tail-ge by auto
    next
      case False
      have p-ge-c: bp-marker c ≤ snd p
        using False by simp
      have tail-ge-c:
        ∀ d∈set (bp-local-insert-bucket M p (c # cs)).
          bp-marker c ≤ bp-marker d
        using IH[of p] sorted-tail lower-tail p-ge-c Cons by auto
      show ?thesis
        using Cons False b-le-c tail-ge-c by auto
    qed
  qed
 qed
qed

lemma bp-local-insert-bucket-markers-sorted:
  assumes M-pos: 0 < M
    and sorted: sorted-wrt (λb c. bp-marker b ≤ bp-marker c) bs
    and lower: ∀ b∈set bs. ∀ q∈set (bp-bucket-entries b).
      bp-marker b ≤ snd q
    and boundaries: bp-bucket-boundaries-ok bs
  shows sorted-wrt (λb c. bp-marker b ≤ bp-marker c)
    (bp-local-insert-bucket M p bs)
  using sorted lower boundaries
proof (induction bs arbitrary: p)
  case Nil
  then show ?case
    by (simp add: bp-bucketize-entries-markers-sorted[OF M-pos])
next
  case (Cons b bs)
  note IH = Cons.IH
  note sorted = Cons.prems(1)
  note lower = Cons.prems(2)
  note boundaries = Cons.prems(3)
  show ?case
  proof (cases bs)
    case Nil
    show ?thesis
    proof (cases snd p < bp-marker b)
      case True
      have left-sorted:
        sorted-wrt (λb c. bp-marker b ≤ bp-marker c)
          (bp-bucketize-entries M [p])
        by (rule bp-bucketize-entries-markers-sorted[OF M-pos])
```

```
    have left-le-b:
      ∀ x∈set (bp-bucketize-entries M [p]).
        bp-marker x ≤ bp-marker b
    proof (rule bp-bucketize-entries-markers-le)
      show ∀ q∈set [p]. snd q ≤ bp-marker b
        using True by simp
    qed
    show ?thesis
      using Nil True left-sorted left-le-b
      by (simp add: sorted-wrt-append)
  next
    case False
    show ?thesis
      using Nil False
      by (simp add: bp-bucketize-entries-markers-sorted[OF M-pos])
  qed
next
  case (Cons c cs)
  have sorted-tail:
    sorted-wrt (λb c. bp-marker b ≤ bp-marker c) (c # cs)
    using sorted Cons by simp
  have lower-tail:
    ∀ b∈set (c # cs). ∀ q∈set (bp-bucket-entries b).
      bp-marker b ≤ snd q
    using lower Cons by simp
  have boundaries-tail: bp-bucket-boundaries-ok (c # cs)
    using boundaries Cons by simp
  have b-le-c: bp-marker b ≤ bp-marker c
    using sorted Cons by simp
  have b-le-all-tail:
    ∀ d∈set (b # c # cs). bp-marker b ≤ bp-marker d
    using sorted Cons by auto
  show ?thesis
  proof (cases snd p < bp-marker b)
    case True
    have left-sorted:
      sorted-wrt (λb c. bp-marker b ≤ bp-marker c)
        (bp-bucketize-entries M [p])
      by (rule bp-bucketize-entries-markers-sorted[OF M-pos])
    have right-sorted:
      sorted-wrt (λb c. bp-marker b ≤ bp-marker c) (b # c # cs)
      using sorted Cons by simp
    have left-le-b:
      ∀ x∈set (bp-bucketize-entries M [p]).
        bp-marker x ≤ bp-marker b
    proof (rule bp-bucketize-entries-markers-le)
      show ∀ q∈set [p]. snd q ≤ bp-marker b
        using True by simp
    qed
```

```
  have cross:
    ∀x∈set (bp-bucketize-entries M [p]).
      ∀y∈set (b # c # cs). bp-marker x ≤ bp-marker y
    using left-le-b b-le-all-tail by force
  show ?thesis
    using Cons True left-sorted right-sorted cross
    by (simp add: sorted-wrt-append)
next
  case False
  note not-before-b = False
  show ?thesis
  proof (cases snd p < bp-marker c)
    case True
    have left-sorted:
      sorted-wrt (λb c. bp-marker b ≤ bp-marker c)
        (bp-bucketize-entries M (p # bp-bucket-entries b))
      by (rule bp-bucketize-entries-markers-sorted[OF M-pos])
    have left-le-c:
      ∀x∈set (bp-bucketize-entries M (p # bp-bucket-entries b)).
        bp-marker x ≤ bp-marker c
    proof (rule bp-bucketize-entries-markers-le)
      show ∀q∈set (p # bp-bucket-entries b). snd q ≤ bp-marker c
        using boundaries Cons True by auto
    qed
    have c-le-all:
      ∀d∈set (c # cs). bp-marker c ≤ bp-marker d
      using sorted-tail by auto
    have cross:
      ∀x∈set (bp-bucketize-entries M (p # bp-bucket-entries b)).
        ∀y∈set (c # cs). bp-marker x ≤ bp-marker y
      using left-le-c c-le-all by force
    show ?thesis
      using Cons False True left-sorted sorted-tail cross
      by (simp add: sorted-wrt-append)
  next
    case False
    have p-ge-c: bp-marker c ≤ snd p
      using False by simp
    have p-ge-hd: bp-marker (hd (c # cs)) ≤ snd p
      using p-ge-c by simp
    have tail-nonempty: c # cs ≠ []
      by simp
    have tail-sorted-result:
      sorted-wrt (λb c. bp-marker b ≤ bp-marker c)
        (bp-local-insert-bucket M p (c # cs))
      using IH[of p] sorted-tail lower-tail boundaries-tail Cons by auto
    have tail-ge-hd:
      ∀d∈set (bp-local-insert-bucket M p (c # cs)).
        bp-marker (hd (c # cs)) ≤ bp-marker d
```

```
        by (rule bp-local-insert-bucket-markers-ge-hd
            [where bs = c # cs and p = p and M = M,
              OF M-pos sorted-tail lower-tail tail-nonempty p-ge-hd])
      have tail-ge-c:
        ∀ d∈set (bp-local-insert-bucket M p (c # cs)).
          bp-marker c ≤ bp-marker d
        using tail-ge-hd by simp
      have tail-ge-b:
        ∀ d∈set (bp-local-insert-bucket M p (c # cs)).
          bp-marker b ≤ bp-marker d
        using b-le-c tail-ge-c by force
      show ?thesis
        using Cons not-before-b False tail-sorted-result tail-ge-b by simp
    qed
  qed
 qed
qed

lemma bp-local-insert-bucket-boundaries-ok:
  assumes M-pos: 0 < M
    and sorted: sorted-wrt (λb c. bp-marker b ≤ bp-marker c) bs
    and lower: ∀ b∈set bs. ∀ q∈set (bp-bucket-entries b).
      bp-marker b ≤ snd q
    and boundaries: bp-bucket-boundaries-ok bs
  shows bp-bucket-boundaries-ok (bp-local-insert-bucket M p bs)
  using sorted lower boundaries
proof (induction bs arbitrary: p)
  case Nil
  then show ?case
    by (simp add: bp-bucketize-entries-boundaries-ok[OF M-pos])
next
  case (Cons b bs)
  note IH = Cons.IH
  note sorted = Cons.prems(1)
  note lower = Cons.prems(2)
  note boundaries = Cons.prems(3)
  show ?case
  proof (cases bs)
    case Nil
    show ?thesis
    proof (cases snd p < bp-marker b)
      case True
      have tail: bp-bucket-boundaries-ok [b]
        by simp
      have upper: ∀ q∈set [p]. snd q ≤ bp-marker b
        using True by simp
      show ?thesis
        using Nil True
        by (simp add: bp-bucket-boundaries-ok-bucketize-append-Cons
```

```
        [OF M-pos tail upper])
  next
    case False
    show ?thesis
     using Nil False by (simp add: bp-bucketize-entries-boundaries-ok[OF M-pos])
  qed
next
  case (Cons c cs)
  have sorted-tail:
    sorted-wrt (λb c. bp-marker b ≤ bp-marker c) (c # cs)
    using sorted Cons by simp
  have lower-tail:
    ∀ b∈set (c # cs). ∀ q∈set (bp-bucket-entries b).
      bp-marker b ≤ snd q
    using lower Cons by simp
  have boundaries-tail: bp-bucket-boundaries-ok (c # cs)
    using boundaries Cons by simp
  show ?thesis
  proof (cases snd p < bp-marker b)
    case True
    have tail-all: bp-bucket-boundaries-ok (b # c # cs)
      using boundaries Cons by simp
    have upper: ∀ q∈set [p]. snd q ≤ bp-marker b
      using True by simp
    show ?thesis
      using Cons True
      by (simp add: bp-bucket-boundaries-ok-bucketize-append-Cons
          [where c = b and bs = c # cs and xs = [p],
            OF M-pos tail-all upper])
  next
    case False
    note not-before-b = False
    show ?thesis
    proof (cases snd p < bp-marker c)
      case True
      have upper:
        ∀ q∈set (p # bp-bucket-entries b). snd q ≤ bp-marker c
        using boundaries Cons True by auto
      show ?thesis
        using Cons not-before-b True
        by (simp add: bp-bucket-boundaries-ok-bucketize-append-Cons
            [OF M-pos boundaries-tail upper])
    next
      case False
      have p-ge-c: bp-marker c ≤ snd p
        using False by simp
      have p-ge-hd: bp-marker (hd (c # cs)) ≤ snd p
        using p-ge-c by simp
      have tail-nonempty: c # cs ≠ []
```

```
      by simp
    have tail-boundaries:
      bp-bucket-boundaries-ok (bp-local-insert-bucket M p (c # cs))
      using IH[of p] sorted-tail lower-tail boundaries-tail Cons by auto
    have tail-ge-hd:
      ∀ d∈set (bp-local-insert-bucket M p (c # cs)).
        bp-marker (hd (c # cs)) ≤ bp-marker d
      by (rule bp-local-insert-bucket-markers-ge-hd
          [where bs = c # cs and p = p and M = M,
            OF M-pos sorted-tail lower-tail tail-nonempty p-ge-hd])
    have tail-nonempty-result:
      bp-local-insert-bucket M p (c # cs) ≠ []
      by (rule bp-local-insert-bucket-nonempty[OF M-pos])
    obtain d ds where tail-eq:
      bp-local-insert-bucket M p (c # cs) = d # ds
      using tail-nonempty-result by (cases bp-local-insert-bucket M p (c # cs))
auto
    have c-le-d: bp-marker c ≤ bp-marker d
      using tail-ge-hd tail-eq by simp
    have head-boundary:
      ∀ q∈set (bp-bucket-entries b). snd q ≤ bp-marker d
    proof
      fix q
      assume q: q ∈ set (bp-bucket-entries b)
      have snd q ≤ bp-marker c
        using boundaries Cons q by auto
      then show snd q ≤ bp-marker d
        using c-le-d by linarith
    qed
    show ?thesis
     using Cons not-before-b False tail-eq tail-boundaries head-boundary by simp
  qed
 qed
qed

lemma bp-local-insert-bucket-preserves-bucket-shape:
  assumes M-pos: 0 < M
    and sizes: ∀ b∈set bs. length (bp-bucket-entries b) ≤ M
    and sorted: sorted-wrt (λb c. bp-marker b ≤ bp-marker c) bs
    and lower: ∀ b∈set bs. ∀ q∈set (bp-bucket-entries b).
      bp-marker b ≤ snd q
    and boundaries: bp-bucket-boundaries-ok bs
  shows (∀ b∈set (bp-local-insert-bucket M p bs).
      length (bp-bucket-entries b) ≤ M) ∧
    sorted-wrt (λb c. bp-marker b ≤ bp-marker c)
      (bp-local-insert-bucket M p bs) ∧
    (∀ b∈set (bp-local-insert-bucket M p bs).
      ∀ q∈set (bp-bucket-entries b). bp-marker b ≤ snd q) ∧
```

bp-bucket-boundaries-ok (*bp-local-insert-bucket M p bs*)
using *bp-local-insert-bucket-sizes-ok*[*OF M-pos sizes*, *of p*]
bp-local-insert-bucket-markers-sorted[*OF M-pos sorted lower boundaries*, *of p*]
bp-local-insert-bucket-markers-lower-bound[*OF M-pos lower*, *of p*]
bp-local-insert-bucket-boundaries-ok[*OF M-pos sorted lower boundaries*, *of p*]
by *blast*

The local-insert proof is deliberately split into many list-level lemmas because each invariant component has a different reason for being preserved. Size preservation depends on re-chunking only the affected bucket. Marker ordering depends on inserting into the unique marker interval selected by the value of the new entry. Boundary preservation depends on the fact that every entry of the previous bucket remains below the next bucket marker. The bundled lemma ⟦*0 < ?M*; ∀ *b*∈*set ?bs*. *length* (*bp-bucket-entries b*) ≤ *?M*; *sorted-wrt* (λ*b c*. *bp-marker b* ≤ *bp-marker c*) *?bs*; ∀ *b*∈*set ?bs*. ∀ *q*∈*set* (*bp-bucket-entries b*). *bp-marker b* ≤ *snd q*; *bp-bucket-boundaries-ok ?bs*⟧ ⟹ (∀ *b*∈*set* (*bp-local-insert-bucket ?M ?p ?bs*). *length* (*bp-bucket-entries b*) ≤ *?M*) ∧ *sorted-wrt* (λ*b c*. *bp-marker b* ≤ *bp-marker c*) (*bp-local-insert-bucket ?M ?p ?bs*) ∧ (∀ *b*∈*set* (*bp-local-insert-bucket ?M ?p ?bs*). ∀ *q*∈*set* (*bp-bucket-entries b*). *bp-marker b* ≤ *snd q*) ∧ *bp-bucket-boundaries-ok* (*bp-local-insert-bucket ?M ?p ?bs*) is the high-level summary used by the state-level Insert proof.

lemma *length-bp-bucket-entries-flat-bucketize-entries* [*simp*]:
assumes *0 < M*
shows *length* (*bp-bucket-entries-flat* (*bp-bucketize-entries M xs*)) =
length xs
using *bp-bucket-entries-flat-bucketize-entries*[*OF assms*, *of xs*] **by** *simp*

lemma *length-bp-bucketize-sorted-entries-aux-le-ratio*:
assumes *M-pos*: *0 < M*
and *fuel*: *length xs* ≤ *fuel*
shows *length* (*bp-bucketize-sorted-entries-aux fuel M xs*) ≤
Suc (*length xs div M*)
using *M-pos fuel*
proof (*induction fuel arbitrary*: *xs*)
case *0*
then show *?case*
by *simp*
next
case (*Suc fuel*)
show *?case*
proof (*cases xs* = [])
case *True*
then show *?thesis*
by *simp*
next
case *False*
show *?thesis*

```
    proof (cases length xs < M)
      case True
      then have drop M xs = []
        by simp
      then show ?thesis
        using Suc.prems False True by simp
    next
      case False
      have M-le: M ≤ length xs
        using False by simp
      have len-drop: length (drop M xs) ≤ fuel
        using Suc.prems False by simp
      have tail:
        length (bp-bucketize-sorted-entries-aux fuel M (drop M xs)) ≤
          Suc (length (drop M xs) div M)
        by (rule Suc.IH[OF Suc.prems(1) len-drop])
      have len-xs:
        length xs = M + length (drop M xs)
        using M-le by simp
      have div-eq:
        length xs div M = Suc (length (drop M xs) div M)
      proof −
        have length xs div M = (M + length (drop M xs)) div M
          using len-xs by simp
        also have ... = length (drop M xs) div M + 1
          using Suc.prems(1) by (simp add: div-add-self1)
        finally show ?thesis
          by simp
      qed
      show ?thesis
        using Suc.prems False tail div-eq by simp
    qed
  qed
qed

lemma length-bp-bucketize-sorted-entries-le-ratio:
  assumes 0 < M
  shows length (bp-bucketize-sorted-entries M xs) ≤
    Suc (length xs div M)
  unfolding bp-bucketize-sorted-entries-def
  by (rule length-bp-bucketize-sorted-entries-aux-le-ratio[OF assms]) simp

lemma length-bp-bucketize-entries-le-ratio:
  assumes 0 < M
  shows length (bp-bucketize-entries M xs) ≤
    Suc (length xs div M)
  unfolding bp-bucketize-entries-def
  using length-bp-bucketize-sorted-entries-le-ratio[OF assms, of sort-key snd xs]
  by simp
```

```
lemma length-bp-bucketize-entries-singleton [simp]:
  assumes 0 < M
  shows length (bp-bucketize-entries M [p]) = 1
  using assms
  unfolding bp-bucketize-entries-def bp-bucketize-sorted-entries-def
  by (cases M) simp-all

lemma length-bp-bucketize-sorted-entries-aux-le-length:
  assumes M-pos: 0 < M
    and fuel: length xs ≤ fuel
  shows length (bp-bucketize-sorted-entries-aux fuel M xs) ≤
    length xs
  using M-pos fuel
proof (induction fuel arbitrary: xs)
  case 0
  then show ?case
    by simp
next
  case (Suc fuel)
  show ?case
  proof (cases xs = [])
    case True
    then show ?thesis by simp
  next
    case False
    have len-drop: length (drop M xs) ≤ fuel
      using Suc.prems False by (cases M) auto
    have tail:
      length (bp-bucketize-sorted-entries-aux fuel M (drop M xs)) ≤
        length (drop M xs)
      by (rule Suc.IH[OF Suc.prems(1) len-drop])
    have drop-less: length (drop M xs) < length xs
      using Suc.prems False by simp
    have drop-le: Suc (length (drop M xs)) ≤ length xs
      using drop-less by simp
    have unfold:
      length (bp-bucketize-sorted-entries-aux (Suc fuel) M xs) =
        Suc (length (bp-bucketize-sorted-entries-aux fuel M (drop M xs)))
      using Suc.prems False by simp
    show ?thesis
      using tail drop-le unfolding unfold by simp
  qed
qed

lemma length-bp-bucketize-entries-le-length:
  assumes 0 < M
  shows length (bp-bucketize-entries M xs) ≤ length xs
  unfolding bp-bucketize-entries-def bp-bucketize-sorted-entries-def
```

```
  using length-bp-bucketize-sorted-entries-aux-le-length
    [OF assms, of sort-key snd xs length xs]
  by simp

lemma length-bp-bucketize-entries-le-three:
  assumes M-pos: 0 < M
    and len: length xs ≤ Suc (2 * M)
  shows length (bp-bucketize-entries M xs) ≤ 3
proof (cases M = 1)
  case True
  have length (bp-bucketize-entries M xs) ≤ length xs
    by (rule length-bp-bucketize-entries-le-length[OF M-pos])
  also have ... ≤ 3
    using len True by simp
  finally show ?thesis .
next
  case False
  have M-ge2: 2 ≤ M
    using M-pos False by simp
  have div-bound: length xs div M ≤ 2
  proof −
    have length xs div M ≤ Suc (2 * M) div M
      by (rule div-le-mono[OF len])
    moreover have Suc (2 * M) div M < 3
    proof −
      have Suc (2 * M) < 3 * M
        using M-ge2 by simp
      then show ?thesis
        using M-pos by (simp add: div-less-iff-less-mult)
    qed
    ultimately show ?thesis
      by simp
  qed
  have length (bp-bucketize-entries M xs) ≤ Suc (length xs div M)
    by (rule length-bp-bucketize-entries-le-ratio[OF M-pos])
  also have ... ≤ 3
    using div-bound by simp
  finally show ?thesis .
qed

lemma length-bp-lazy-bucket-insert-entries-le-three:
  assumes M-pos: 0 < M
    and size: length (bp-bucket-entries b) ≤ 2 * M
  shows length (bp-lazy-bucket-insert-entries M p b) ≤ 3
proof (cases length (bp-bucket-entries b) < 2 * M)
  case True
  then show ?thesis
    unfolding bp-lazy-bucket-insert-entries-def by simp
next
```

```
  case False
  have len: length (p # bp-bucket-entries b) ≤ Suc (2 * M)
    using size by simp
  show ?thesis
    using False length-bp-bucketize-entries-le-three[OF M-pos len]
    unfolding bp-lazy-bucket-insert-entries-def by simp
qed

lemma length-bp-lazy-insert-bucket-le:
  assumes M-pos: 0 < M
    and sizes: ∀ b∈set bs. length (bp-bucket-entries b) ≤ 2 * M
  shows length (bp-lazy-insert-bucket M p bs) ≤ length bs + 2
  using sizes
proof (induction bs arbitrary: p)
  case Nil
  then show ?case
    using length-bp-bucketize-entries-singleton[OF M-pos, of p] by simp
next
  case (Cons b bs)
  note IH = Cons.IH
  note sizes = Cons.prems
  have b-size: length (bp-bucket-entries b) ≤ 2 * M
    using sizes by simp
  show ?case
  proof (cases bs)
    case Nil
    show ?thesis
    proof (cases snd p < bp-marker b)
      case True
      then show ?thesis
        using Nil length-bp-bucketize-entries-singleton[OF M-pos, of p]
        by simp
    next
      case False
      have length (bp-lazy-bucket-insert-entries M p b) ≤ 3
        by (rule length-bp-lazy-bucket-insert-entries-le-three
            [OF M-pos b-size])
      then show ?thesis
        using Nil False by simp
    qed
  next
    case (Cons c cs)
    have tail-sizes:
      ∀ b∈set bs. length (bp-bucket-entries b) ≤ 2 * M
      using sizes by simp
    show ?thesis
    proof (cases snd p < bp-marker b)
      case True
      then show ?thesis
```

```
      using Cons length-bp-bucketize-entries-singleton[OF M-pos, of p]
      by simp
    next
      case False
      show ?thesis
      proof (cases snd p < bp-marker c)
        case True
        have length (bp-lazy-bucket-insert-entries M p b) ≤ 3
          by (rule length-bp-lazy-bucket-insert-entries-le-three
              [OF M-pos b-size])
        then show ?thesis
          using Cons False True by simp
      next
        case False
        have tail:
          length (bp-lazy-insert-bucket M p bs) ≤ length bs + 2
          by (rule IH[OF tail-sizes])
        show ?thesis
          using Cons ‹¬ snd p < bp-marker b› False tail by simp
      qed
    qed
  qed
qed

lemma length-bp-bucket-entries-flat-local-insert-bucket [simp]:
  assumes 0 < M
  shows length (bp-bucket-entries-flat (bp-local-insert-bucket M p bs)) =
    Suc (length (bp-bucket-entries-flat bs))
  using assms
proof (induction bs arbitrary: p)
  case Nil
  have len:
    length (bp-bucket-entries-flat (bp-bucketize-entries M [p])) =
      length [p]
    by (rule length-bp-bucket-entries-flat-bucketize-entries[OF Nil.prems])
  show ?case
    using len by (simp add: bp-bucket-entries-flat-def)
next
  case (Cons b bs)
  then show ?case
  proof (cases bs)
    case Nil
    have len-single:
      length (bp-bucket-entries-flat (bp-bucketize-entries M [p])) =
        length [p]
      by (rule length-bp-bucket-entries-flat-bucketize-entries[OF Cons.prems])
    have len-inserted:
      length
        (bp-bucket-entries-flat
```

```
        (bp-bucketize-entries M (p # bp-bucket-entries b))) =
       length (p # bp-bucket-entries b)
      by (rule length-bp-bucket-entries-flat-bucketize-entries[OF Cons.prems])
    then show ?thesis
      using Nil len-single len-inserted
      by (auto simp: bp-bucket-entries-flat-def)
  next
    case (Cons c cs)
    have len-single:
      length (bp-bucket-entries-flat (bp-bucketize-entries M [p])) =
        length [p]
      by (rule length-bp-bucket-entries-flat-bucketize-entries[OF Cons.prems])
    have len-inserted:
      length
        (bp-bucket-entries-flat
          (bp-bucketize-entries M (p # bp-bucket-entries b))) =
       length (p # bp-bucket-entries b)
      by (rule length-bp-bucket-entries-flat-bucketize-entries[OF Cons.prems])
    have tail-len:
      length (bp-bucket-entries-flat
          (bp-local-insert-bucket M p (c # cs))) =
       Suc (length (bp-bucket-entries-flat (c # cs)))
      using Cons.IH[OF Cons.prems, of p] Cons by simp
    show ?thesis
    proof (cases snd p < bp-marker b)
      case True
      then show ?thesis
        using Cons len-single by (simp add: bp-bucket-entries-flat-def)
    next
      case False
      note not-before-b = False
      show ?thesis
      proof (cases snd p < bp-marker c)
        case True
        then show ?thesis
          using Cons not-before-b len-inserted
          by (simp add: bp-bucket-entries-flat-def)
      next
        case False
        then show ?thesis
          using Cons not-before-b tail-len
          by (simp add: bp-bucket-entries-flat-def)
      qed
    qed
  qed
qed

lemma bp-local-insert-bucket-distinct-keys:
  assumes M-pos: 0 < M
```

```
    and distinct: distinct (map fst (bp-bucket-entries-flat bs))
    and fresh: fst p ∉ bp-entry-keys (bp-bucket-entries-flat bs)
  shows distinct
    (map fst (bp-bucket-entries-flat (bp-local-insert-bucket M p bs)))
proof (rule card-distinct)
  let ?old = map fst (bp-bucket-entries-flat bs)
  let ?new =
    map fst (bp-bucket-entries-flat (bp-local-insert-bucket M p bs))
  have set-new: set ?new = insert (fst p) (set ?old)
    using set-bp-bucket-entries-flat-local-insert-bucket[OF M-pos, of p bs]
    by auto
  have fresh-set: fst p ∉ set ?old
    using fresh unfolding bp-entry-keys-def by simp
  have card (set ?new) = Suc (card (set ?old))
    unfolding set-new using fresh-set by simp
  also have ... = Suc (length ?old)
    using distinct-card[OF distinct] by simp
  also have ... = length ?new
    using length-bp-bucket-entries-flat-local-insert-bucket[OF M-pos, of p bs]
    by simp
  finally show card (set ?new) = length ?new .
qed

lemma bp-local-insert-bucket-values-consistent:
  assumes M-pos: 0 < M
    and vals: ∀ q∈set (bp-bucket-entries-flat bs). f (fst q) = snd q
    and fresh: fst p ∉ bp-entry-keys (bp-bucket-entries-flat bs)
  shows ∀ q∈set (bp-bucket-entries-flat (bp-local-insert-bucket M p bs)).
    (f(fst p := snd p)) (fst q) = snd q
proof
  fix q
  assume q: q ∈
    set (bp-bucket-entries-flat (bp-local-insert-bucket M p bs))
  then have q-cases: q = p ∨ q ∈ set (bp-bucket-entries-flat bs)
    using set-bp-bucket-entries-flat-local-insert-bucket[OF M-pos, of p bs]
    by auto
  from q-cases show (f(fst p := snd p)) (fst q) = snd q
  proof
    assume q = p
    then show ?thesis by simp
  next
    assume q-old: q ∈ set (bp-bucket-entries-flat bs)
    have fst q ∈ bp-entry-keys (bp-bucket-entries-flat bs)
      using q-old unfolding bp-entry-keys-def by auto
    then have fst q ≠ fst p
      using fresh by auto
    moreover have f (fst q) = snd q
      using vals q-old by blast
    ultimately show ?thesis
```

```
    by simp
  qed
qed

lemma bp-empty-invariant:
  assumes 0 < M
  shows bp-invariant (bp-empty M B)
  using assms
  unfolding bp-invariant-def bp-empty-def bp-entries-def
    bp-bucket-entries-flat-def bp-bucket-sizes-ok-def
    bp-bucket-markers-sorted-def bp-bucket-markers-lower-bound-def
    bp-values-consistent-def bp-distinct-keys-def
  by simp

lemma bp-empty-boundaries-state-ok [simp]:
  bp-bucket-boundaries-state-ok (bp-empty M B)
  unfolding bp-bucket-boundaries-state-ok-def bp-empty-def by simp

lemma bp-empty-ordered-invariant:
  assumes 0 < M
  shows bp-ordered-invariant (bp-empty M B)
  unfolding bp-ordered-invariant-def
  using bp-empty-invariant[OF assms] by simp

lemma bp-empty-view [simp]:
  bp-view (bp-empty M B) = (|keys-of = {}, value-of = (λ-. B)|)
  unfolding bp-view-def bp-empty-def bp-entries-def
    bp-bucket-entries-flat-def bp-entry-keys-def
  by simp

lemma bp-empty-partition-upper-bound [simp]:
  partition-upper-bound (bp-view (bp-empty M B0)) B
  unfolding partition-upper-bound-def by simp

lemma set-bp-insert-bucket:
  set (bp-insert-bucket p bs) = insert (bp-singleton-bucket p) (set bs)
  by (induction bs) auto

lemma set-bp-bucket-entries-flat-insert-bucket:
  set (bp-bucket-entries-flat (bp-insert-bucket p bs)) =
    insert p (set (bp-bucket-entries-flat bs))
  unfolding bp-bucket-entries-flat-def
  by (induction bs) auto

lemma length-bp-bucket-entries-flat-insert-bucket [simp]:
  length (bp-bucket-entries-flat (bp-insert-bucket p bs)) =
    Suc (length (bp-bucket-entries-flat bs))
  unfolding bp-bucket-entries-flat-def
  by (induction bs) auto
```

```
lemma bp-entry-keys-insert-bucket [simp]:
  bp-entry-keys (bp-bucket-entries-flat (bp-insert-bucket p bs)) =
    insert (fst p) (bp-entry-keys (bp-bucket-entries-flat bs))
  unfolding bp-entry-keys-def
  using set-bp-bucket-entries-flat-insert-bucket[of p bs] by auto

lemma bp-insert-bucket-sizes-ok:
  assumes ∀ b∈set bs. length (bp-bucket-entries b) ≤ M
    and 0 < M
  shows ∀ b∈set (bp-insert-bucket p bs).
    length (bp-bucket-entries b) ≤ M
  using assms unfolding bp-singleton-bucket-def
  by (induction bs) auto

lemma bp-insert-bucket-markers-sorted:
  assumes sorted-wrt (λb c. bp-marker b ≤ bp-marker c) bs
  shows sorted-wrt (λb c. bp-marker b ≤ bp-marker c)
    (bp-insert-bucket p bs)
  using assms
proof (induction bs)
  case Nil
  then show ?case
    unfolding bp-singleton-bucket-def by simp
next
  case (Cons b bs)
  show ?case
  proof (cases snd p ≤ bp-marker b)
    case True
    then show ?thesis
      using Cons.prems by auto
  next
    case False
    have tail-sorted:
      sorted-wrt (λb c. bp-marker b ≤ bp-marker c) bs
      using Cons.prems by simp
    have inserted-sorted:
      sorted-wrt (λb c. bp-marker b ≤ bp-marker c)
        (bp-insert-bucket p bs)
      by (rule Cons.IH[OF tail-sorted])
    have lower:
      ⋀c. c ∈ set (bp-insert-bucket p bs) ⟹
        bp-marker b ≤ bp-marker c
      using Cons.prems False
      by (auto simp: set-bp-insert-bucket bp-singleton-bucket-def)
    show ?thesis
      using False inserted-sorted lower by simp
  qed
qed
```

```
lemma bp-insert-bucket-markers-lower-bound:
  assumes ∀ b∈set bs. ∀ p∈set (bp-bucket-entries b).
      bp-marker b ≤ snd p
  shows ∀ b∈set (bp-insert-bucket p bs). ∀ p∈set (bp-bucket-entries b).
      bp-marker b ≤ snd p
  using assms
  by (induction bs) auto

lemma bp-insert-bucket-values-consistent:
  assumes ∀ q∈set (bp-bucket-entries-flat bs). f (fst q) = snd q
    and fst p ∉ bp-entry-keys (bp-bucket-entries-flat bs)
  shows ∀ q∈set (bp-bucket-entries-flat (bp-insert-bucket p bs)).
    (f(fst p := snd p)) (fst q) = snd q
proof
  fix q
  assume q: q ∈ set (bp-bucket-entries-flat (bp-insert-bucket p bs))
  then have q-cases: q = p ∨ q ∈ set (bp-bucket-entries-flat bs)
    using set-bp-bucket-entries-flat-insert-bucket[of p bs] by auto
  from q-cases show (f(fst p := snd p)) (fst q) = snd q
  proof
    assume q = p
    then show ?thesis by simp
  next
    assume q-old: q ∈ set (bp-bucket-entries-flat bs)
    have fst q ∈ bp-entry-keys (bp-bucket-entries-flat bs)
      using q-old unfolding bp-entry-keys-def by auto
    then have fst q ≠ fst p
      using assms(2) by auto
    moreover have f (fst q) = snd q
      using assms(1) q-old by blast
    ultimately show ?thesis
      by simp
  qed
qed

lemma bp-insert-bucket-distinct-keys:
  assumes distinct: distinct (map fst (bp-bucket-entries-flat bs))
    and fresh: fst p ∉ bp-entry-keys (bp-bucket-entries-flat bs)
  shows distinct (map fst (bp-bucket-entries-flat (bp-insert-bucket p bs)))
proof (rule card-distinct)
  let ?old = map fst (bp-bucket-entries-flat bs)
  let ?new = map fst (bp-bucket-entries-flat (bp-insert-bucket p bs))
  have set-new: set ?new = insert (fst p) (set ?old)
    using set-bp-bucket-entries-flat-insert-bucket[of p bs] by auto
  have fresh-set: fst p ∉ set ?old
    using fresh unfolding bp-entry-keys-def by simp
  have card (set ?new) = Suc (card (set ?old))
    unfolding set-new using fresh-set by simp
```

```
  also have ... = Suc (length ?old)
    using distinct-card[OF distinct] by simp
  also have ... = length ?new
    by simp
  finally show card (set ?new) = length ?new .
qed

lemma distinct-map-fst-filter-neq:
  assumes distinct (map fst xs)
  shows distinct (map fst (filter (λp. fst p ≠ x) xs))
  using assms by (induction xs) auto

lemma distinct-map-fst-filter-notin:
  assumes distinct (map fst xs)
  shows distinct (map fst (filter (λp. fst p ∉ S) xs))
  using assms by (induction xs) auto

lemma bp-bucket-entries-flat-delete-key:
  bp-bucket-entries-flat (map (bp-delete-key-from-bucket x) bs) =
    filter (λp. fst p ≠ x) (bp-bucket-entries-flat bs)
  unfolding bp-bucket-entries-flat-def bp-delete-key-from-bucket-def
  by (induction bs) simp-all

lemma bp-bucket-entries-flat-delete-keys:
  bp-bucket-entries-flat (map (bp-delete-keys-from-bucket S) bs) =
    filter (λp. fst p ∉ S) (bp-bucket-entries-flat bs)
  unfolding bp-bucket-entries-flat-def bp-delete-keys-from-bucket-def
  by (induction bs) simp-all

lemma bp-entries-delete-key:
  bp-entries (bp-delete-key x P) =
    filter (λp. fst p ≠ x) (bp-entries P)
  unfolding bp-entries-def bp-delete-key-def
  by (simp add: bp-bucket-entries-flat-delete-key)

lemma bp-entries-delete-keys:
  bp-entries (bp-delete-keys S P) =
    filter (λp. fst p ∉ S) (bp-entries P)
  unfolding bp-entries-def bp-delete-keys-def
  by (simp add: bp-bucket-entries-flat-delete-keys)

lemma length-filter-fst-neq-ge-pred:
  assumes distinct: distinct (map fst xs)
  shows length xs ≤ Suc (length (filter (λp. fst p ≠ x) xs))
  using distinct
proof (induction xs)
  case Nil
  then show ?case by simp
next
```

```
  case (Cons p xs)
  show ?case
  proof (cases fst p = x)
    case True
    have fst p ∉ fst ‘ set xs
      using Cons.prems by simp
    have ∀ q∈set xs. fst q ≠ x
    proof
      fix q
      assume q ∈ set xs
      then have fst q ∈ fst ‘ set xs
        by blast
      have fst q ≠ fst p
      proof
        assume fst q = fst p
        then have fst p ∈ fst ‘ set xs
          using ‹fst q ∈ fst ‘ set xs› by simp
        then show False
          using ‹fst p ∉ fst ‘ set xs› by simp
      qed
      then show fst q ≠ x
        using True by simp
    qed
    then have filter (λp. fst p ≠ x) xs = xs
      by (simp add: filter-id-conv)
    then show ?thesis
      using True by simp
  next
    case False
    have tail: length xs ≤ Suc (length (filter (λp. fst p ≠ x) xs))
      by (rule Cons.IH) (use Cons.prems in simp)
    show ?thesis
      using False tail by simp
  qed
qed

lemma length-bp-entries-delete-key-ge-pred:
  assumes bp-distinct-keys P
  shows length (bp-entries P) ≤
    Suc (length (bp-entries (bp-delete-key x P)))
  using length-filter-fst-neq-ge-pred
    [of bp-entries P x] assms
  unfolding bp-entries-delete-key bp-distinct-keys-def by simp

lemma bp-entry-keys-delete-key [simp]:
  bp-entry-keys (bp-entries (bp-delete-key x P)) =
    bp-entry-keys (bp-entries P) − {x}
  by (simp add: bp-entries-delete-key)
```

lemma *bp-entry-keys-delete-keys* [*simp*]:
bp-entry-keys (*bp-entries* (*bp-delete-keys S P*)) =
bp-entry-keys (*bp-entries P*) − *S*
by (*simp add*: *bp-entries-delete-keys*)

lemma *bp-values-delete-key* [*simp*]:
bp-values (*bp-delete-key x P*) = *bp-values P*
unfolding *bp-delete-key-def* **by** *simp*

lemma *bp-values-delete-keys* [*simp*]:
bp-values (*bp-delete-keys S P*) = *bp-values P*
unfolding *bp-delete-keys-def* **by** *simp*

lemma *bp-delete-key-markers-sorted-list*:
assumes *sorted-wrt* (λ*b c*. *bp-marker b* ≤ *bp-marker c*) *bs*
shows *sorted-wrt* (λ*b c*. *bp-marker b* ≤ *bp-marker c*)
(*map* (*bp-delete-key-from-bucket x*) *bs*)
using *assms* **unfolding** *bp-delete-key-from-bucket-def*
by (*induction bs*) *auto*

lemma *bp-delete-keys-markers-sorted-list*:
assumes *sorted-wrt* (λ*b c*. *bp-marker b* ≤ *bp-marker c*) *bs*
shows *sorted-wrt* (λ*b c*. *bp-marker b* ≤ *bp-marker c*)
(*map* (*bp-delete-keys-from-bucket S*) *bs*)
using *assms* **unfolding** *bp-delete-keys-from-bucket-def*
by (*induction bs*) *auto*

lemma *bp-delete-key-markers-sorted*:
assumes *bp-bucket-markers-sorted P*
shows *bp-bucket-markers-sorted* (*bp-delete-key x P*)
using *assms*
unfolding *bp-bucket-markers-sorted-def bp-delete-key-def*
bp-delete-key-from-bucket-def
by (*simp add*: *bp-delete-key-markers-sorted-list*
[*unfolded bp-delete-key-from-bucket-def*])

lemma *bp-delete-keys-markers-sorted*:
assumes *bp-bucket-markers-sorted P*
shows *bp-bucket-markers-sorted* (*bp-delete-keys S P*)
using *assms*
unfolding *bp-bucket-markers-sorted-def bp-delete-keys-def*
bp-delete-keys-from-bucket-def
by (*simp add*: *bp-delete-keys-markers-sorted-list*
[*unfolded bp-delete-keys-from-bucket-def*])

lemma *bp-delete-key-bucket-sizes-ok*:
assumes *bp-bucket-sizes-ok P*
shows *bp-bucket-sizes-ok* (*bp-delete-key x P*)
unfolding *bp-bucket-sizes-ok-def*

proof
fix *b*
assume *b*: *b* ∈ *set* (*bp-buckets* (*bp-delete-key x P*))
then obtain *b0* **where** *b0*: *b0* ∈ *set* (*bp-buckets P*)
and *b-def*: *b* = *bp-delete-key-from-bucket x b0*
unfolding *bp-delete-key-def* **by** *auto*
have *length* (*bp-bucket-entries b*) ≤ *length* (*bp-bucket-entries b0*)
unfolding *b-def bp-delete-key-from-bucket-def* **by** *simp*
also have ... ≤ *bp-block-size P*
using *assms b0* **unfolding** *bp-bucket-sizes-ok-def* **by** *blast*
finally show *length* (*bp-bucket-entries b*) ≤
bp-block-size (*bp-delete-key x P*)
unfolding *bp-delete-key-def* **by** *simp*
qed

lemma *bp-delete-keys-bucket-sizes-ok*:
assumes *bp-bucket-sizes-ok P*
shows *bp-bucket-sizes-ok* (*bp-delete-keys S P*)
unfolding *bp-bucket-sizes-ok-def*
proof
fix *b*
assume *b*: *b* ∈ *set* (*bp-buckets* (*bp-delete-keys S P*))
then obtain *b0* **where** *b0*: *b0* ∈ *set* (*bp-buckets P*)
and *b-def*: *b* = *bp-delete-keys-from-bucket S b0*
unfolding *bp-delete-keys-def* **by** *auto*
have *length* (*bp-bucket-entries b*) ≤ *length* (*bp-bucket-entries b0*)
unfolding *b-def bp-delete-keys-from-bucket-def* **by** *simp*
also have ... ≤ *bp-block-size P*
using *assms b0* **unfolding** *bp-bucket-sizes-ok-def* **by** *blast*
finally show *length* (*bp-bucket-entries b*) ≤
bp-block-size (*bp-delete-keys S P*)
unfolding *bp-delete-keys-def* **by** *simp*
qed

lemma *bp-delete-key-lazy-bucket-sizes-ok*:
assumes *bp-lazy-bucket-sizes-ok P*
shows *bp-lazy-bucket-sizes-ok* (*bp-delete-key x P*)
unfolding *bp-lazy-bucket-sizes-ok-def*
proof
fix *b*
assume *b*: *b* ∈ *set* (*bp-buckets* (*bp-delete-key x P*))
then obtain *b0* **where** *b0*: *b0* ∈ *set* (*bp-buckets P*)
and *b-def*: *b* = *bp-delete-key-from-bucket x b0*
unfolding *bp-delete-key-def* **by** *auto*
have *length* (*bp-bucket-entries b*) ≤ *length* (*bp-bucket-entries b0*)
unfolding *b-def bp-delete-key-from-bucket-def* **by** *simp*
also have ... ≤ *2* ∗ *bp-block-size P*
using *assms b0* **unfolding** *bp-lazy-bucket-sizes-ok-def* **by** *blast*
finally show *length* (*bp-bucket-entries b*) ≤

```
    2 * bp-block-size (bp-delete-key x P)
    unfolding bp-delete-key-def by simp
qed

lemma bp-delete-keys-lazy-bucket-sizes-ok:
  assumes bp-lazy-bucket-sizes-ok P
  shows bp-lazy-bucket-sizes-ok (bp-delete-keys S P)
  unfolding bp-lazy-bucket-sizes-ok-def
proof
  fix b
  assume b: b ∈ set (bp-buckets (bp-delete-keys S P))
  then obtain b0 where b0: b0 ∈ set (bp-buckets P)
    and b-def: b = bp-delete-keys-from-bucket S b0
    unfolding bp-delete-keys-def by auto
  have length (bp-bucket-entries b) ≤ length (bp-bucket-entries b0)
    unfolding b-def bp-delete-keys-from-bucket-def by simp
  also have ... ≤ 2 * bp-block-size P
    using assms b0 unfolding bp-lazy-bucket-sizes-ok-def by blast
  finally show length (bp-bucket-entries b) ≤
    2 * bp-block-size (bp-delete-keys S P)
    unfolding bp-delete-keys-def by simp
qed

lemma bp-delete-key-markers-lower-bound:
  assumes bp-bucket-markers-lower-bound P
  shows bp-bucket-markers-lower-bound (bp-delete-key x P)
  using assms
  unfolding bp-bucket-markers-lower-bound-def bp-delete-key-def
    bp-delete-key-from-bucket-def
  by auto

lemma bp-delete-keys-markers-lower-bound:
  assumes bp-bucket-markers-lower-bound P
  shows bp-bucket-markers-lower-bound (bp-delete-keys S P)
  using assms
  unfolding bp-bucket-markers-lower-bound-def bp-delete-keys-def
    bp-delete-keys-from-bucket-def
  by auto

lemma bp-delete-key-values-consistent:
  assumes bp-values-consistent P
  shows bp-values-consistent (bp-delete-key x P)
  using assms
  unfolding bp-values-consistent-def
  by (auto simp: bp-entries-delete-key)

lemma bp-delete-keys-values-consistent:
  assumes bp-values-consistent P
  shows bp-values-consistent (bp-delete-keys S P)
```

```
  using assms
  unfolding bp-values-consistent-def
  by (auto simp: bp-entries-delete-keys)

lemma bp-delete-key-distinct-keys:
  assumes bp-distinct-keys P
  shows bp-distinct-keys (bp-delete-key x P)
  using assms
  unfolding bp-distinct-keys-def
  by (simp add: bp-entries-delete-key distinct-map-fst-filter-neq)

lemma bp-delete-keys-distinct-keys:
  assumes bp-distinct-keys P
  shows bp-distinct-keys (bp-delete-keys S P)
  using assms
  unfolding bp-distinct-keys-def
  by (simp add: bp-entries-delete-keys distinct-map-fst-filter-notin)

lemma bp-bucket-boundaries-ok-delete-key-from-bucket:
  assumes bp-bucket-boundaries-ok bs
  shows bp-bucket-boundaries-ok (map (bp-delete-key-from-bucket x) bs)
  using assms
proof (induction bs)
  case Nil
  then show ?case
    by simp
next
  case (Cons b bs)
  then show ?case
  proof (cases bs)
    case Nil
    then show ?thesis
      by simp
  next
    case (Cons c cs)
    have head:
      ∀ p∈set (bp-bucket-entries b). snd p ≤ bp-marker c
      using Cons.prems Cons by simp
    have tail: bp-bucket-boundaries-ok
      (map (bp-delete-key-from-bucket x) (c # cs))
      using Cons.IH Cons.prems Cons by simp
    show ?thesis
      using Cons head tail
      unfolding bp-delete-key-from-bucket-def by auto
  qed
qed

lemma bp-bucket-boundaries-ok-delete-keys-from-bucket:
  assumes bp-bucket-boundaries-ok bs
```

```
  shows bp-bucket-boundaries-ok (map (bp-delete-keys-from-bucket S) bs)
  using assms
proof (induction bs)
  case Nil
  then show ?case
    by simp
next
  case (Cons b bs)
  then show ?case
  proof (cases bs)
    case Nil
    then show ?thesis
      by simp
  next
    case (Cons c cs)
    have head:
      ∀ p∈set (bp-bucket-entries b). snd p ≤ bp-marker c
      using Cons.prems Cons by simp
    have tail: bp-bucket-boundaries-ok
      (map (bp-delete-keys-from-bucket S) (c # cs))
      using Cons.IH Cons.prems Cons by simp
    show ?thesis
      using Cons head tail
      unfolding bp-delete-keys-from-bucket-def by auto
  qed
qed

lemma bp-delete-key-boundaries-state-ok:
  assumes bp-bucket-boundaries-state-ok P
  shows bp-bucket-boundaries-state-ok (bp-delete-key x P)
  using assms
  unfolding bp-bucket-boundaries-state-ok-def bp-delete-key-def
  by (simp add: bp-bucket-boundaries-ok-delete-key-from-bucket)

lemma bp-delete-keys-boundaries-state-ok:
  assumes bp-bucket-boundaries-state-ok P
  shows bp-bucket-boundaries-state-ok (bp-delete-keys S P)
  using assms
  unfolding bp-bucket-boundaries-state-ok-def bp-delete-keys-def
  by (simp add: bp-bucket-boundaries-ok-delete-keys-from-bucket)

lemma bp-delete-key-invariant:
  assumes bp-invariant P
  shows bp-invariant (bp-delete-key x P)
proof −
  have block: 0 < bp-block-size (bp-delete-key x P)
    using assms unfolding bp-invariant-def bp-delete-key-def by simp
  have distinct: bp-distinct-keys (bp-delete-key x P)
    using assms unfolding bp-invariant-def
```

```
    by (auto intro: bp-delete-key-distinct-keys)
  have sizes: bp-bucket-sizes-ok (bp-delete-key x P)
    using assms unfolding bp-invariant-def
    by (auto intro: bp-delete-key-bucket-sizes-ok)
  have sorted: bp-bucket-markers-sorted (bp-delete-key x P)
    using assms unfolding bp-invariant-def
    by (auto intro: bp-delete-key-markers-sorted)
  have markers: bp-bucket-markers-lower-bound (bp-delete-key x P)
    using assms unfolding bp-invariant-def
    by (auto intro: bp-delete-key-markers-lower-bound)
  have vals: bp-values-consistent (bp-delete-key x P)
    using assms unfolding bp-invariant-def
    by (auto intro: bp-delete-key-values-consistent)
  show ?thesis
    unfolding bp-invariant-def using block distinct sizes sorted markers vals by
blast
qed

lemma bp-delete-key-ordered-invariant:
  assumes bp-ordered-invariant P
  shows bp-ordered-invariant (bp-delete-key x P)
  unfolding bp-ordered-invariant-def
  using bp-delete-key-invariant[OF bp-ordered-invariant-invariant[OF assms]]
    bp-delete-key-boundaries-state-ok
      [OF bp-ordered-invariant-boundaries-state-ok[OF assms]]
  by blast

lemma bp-delete-key-lazy-invariant:
  assumes bp-lazy-invariant P
  shows bp-lazy-invariant (bp-delete-key x P)
proof −
  have block: 0 < bp-block-size (bp-delete-key x P)
    using assms unfolding bp-lazy-invariant-def bp-delete-key-def by simp
  have distinct: bp-distinct-keys (bp-delete-key x P)
    using assms unfolding bp-lazy-invariant-def
    by (auto intro: bp-delete-key-distinct-keys)
  have sizes: bp-lazy-bucket-sizes-ok (bp-delete-key x P)
    using assms unfolding bp-lazy-invariant-def
    by (auto intro: bp-delete-key-lazy-bucket-sizes-ok)
  have sorted: bp-bucket-markers-sorted (bp-delete-key x P)
    using assms unfolding bp-lazy-invariant-def
    by (auto intro: bp-delete-key-markers-sorted)
  have markers: bp-bucket-markers-lower-bound (bp-delete-key x P)
    using assms unfolding bp-lazy-invariant-def
    by (auto intro: bp-delete-key-markers-lower-bound)
  have vals: bp-values-consistent (bp-delete-key x P)
    using assms unfolding bp-lazy-invariant-def
    by (auto intro: bp-delete-key-values-consistent)
  show ?thesis
```

```
    unfolding bp-lazy-invariant-def
    using block distinct sizes sorted markers vals by blast
qed

lemma bp-delete-key-lazy-ordered-invariant:
  assumes bp-lazy-ordered-invariant P
  shows bp-lazy-ordered-invariant (bp-delete-key x P)
  unfolding bp-lazy-ordered-invariant-def
  using bp-delete-key-lazy-invariant
      [OF bp-lazy-ordered-invariant-lazy-invariant[OF assms]]
    bp-delete-key-boundaries-state-ok
      [OF bp-lazy-ordered-invariant-boundaries-state-ok[OF assms]]
  by blast

lemma bp-delete-keys-invariant:
  assumes bp-invariant P
  shows bp-invariant (bp-delete-keys S P)
proof −
  have block: 0 < bp-block-size (bp-delete-keys S P)
    using assms unfolding bp-invariant-def bp-delete-keys-def by simp
  have distinct: bp-distinct-keys (bp-delete-keys S P)
    using assms unfolding bp-invariant-def
    by (auto intro: bp-delete-keys-distinct-keys)
  have sizes: bp-bucket-sizes-ok (bp-delete-keys S P)
    using assms unfolding bp-invariant-def
    by (auto intro: bp-delete-keys-bucket-sizes-ok)
  have sorted: bp-bucket-markers-sorted (bp-delete-keys S P)
    using assms unfolding bp-invariant-def
    by (auto intro: bp-delete-keys-markers-sorted)
  have markers: bp-bucket-markers-lower-bound (bp-delete-keys S P)
    using assms unfolding bp-invariant-def
    by (auto intro: bp-delete-keys-markers-lower-bound)
  have vals: bp-values-consistent (bp-delete-keys S P)
    using assms unfolding bp-invariant-def
    by (auto intro: bp-delete-keys-values-consistent)
  show ?thesis
    unfolding bp-invariant-def using block distinct sizes sorted markers vals by
blast
qed

lemma bp-delete-keys-ordered-invariant:
  assumes bp-ordered-invariant P
  shows bp-ordered-invariant (bp-delete-keys S P)
  unfolding bp-ordered-invariant-def
  using bp-delete-keys-invariant[OF bp-ordered-invariant-invariant[OF assms]]
    bp-delete-keys-boundaries-state-ok
      [OF bp-ordered-invariant-boundaries-state-ok[OF assms]]
  by blast
```

```
lemma bp-delete-keys-lazy-invariant:
  assumes bp-lazy-invariant P
  shows bp-lazy-invariant (bp-delete-keys S P)
proof −
  have block: 0 < bp-block-size (bp-delete-keys S P)
    using assms unfolding bp-lazy-invariant-def bp-delete-keys-def by simp
  have distinct: bp-distinct-keys (bp-delete-keys S P)
    using assms unfolding bp-lazy-invariant-def
    by (auto intro: bp-delete-keys-distinct-keys)
  have sizes: bp-lazy-bucket-sizes-ok (bp-delete-keys S P)
    using assms unfolding bp-lazy-invariant-def
    by (auto intro: bp-delete-keys-lazy-bucket-sizes-ok)
  have sorted: bp-bucket-markers-sorted (bp-delete-keys S P)
    using assms unfolding bp-lazy-invariant-def
    by (auto intro: bp-delete-keys-markers-sorted)
  have markers: bp-bucket-markers-lower-bound (bp-delete-keys S P)
    using assms unfolding bp-lazy-invariant-def
    by (auto intro: bp-delete-keys-markers-lower-bound)
  have vals: bp-values-consistent (bp-delete-keys S P)
    using assms unfolding bp-lazy-invariant-def
    by (auto intro: bp-delete-keys-values-consistent)
  show ?thesis
    unfolding bp-lazy-invariant-def
    using block distinct sizes sorted markers vals by blast
qed

lemma bp-delete-keys-view:
  bp-view (bp-delete-keys S P) =
    (| keys-of = keys-of (bp-view P) − S,
      value-of = value-of (bp-view P) |)
  unfolding bp-view-def by (simp add: bp-entries-delete-keys)

lemma bp-entry-keys-insert-bucket-state [simp]:
  bp-entry-keys (bp-entries
      (P(| bp-buckets := bp-insert-bucket p (bp-buckets P)|))) =
    insert (fst p) (bp-entry-keys (bp-entries P))
  unfolding bp-entries-def by simp

lemma bp-entry-keys-insert-bucket-state-values [simp]:
  bp-entry-keys (bp-entries
      (P(| bp-buckets := bp-insert-bucket p (bp-buckets P),
        bp-values := f|))) =
    insert (fst p) (bp-entry-keys (bp-entries P))
  unfolding bp-entries-def by simp

lemma bp-insert-invariant:
  assumes inv: bp-invariant P
  shows bp-invariant (bp-insert x b P)
proof −
```

let *?old-keys* = *bp-entry-keys* (*bp-entries P*)
let *?b* = *if x* ∈ *?old-keys then min* (*bp-values P x*) *b else b*
let *?P0* = *bp-delete-key x P*
have *inv0*: *bp-invariant ?P0*
by (*rule bp-delete-key-invariant*[*OF inv*])
have *block-pos*: *0* < *bp-block-size ?P0*
using *inv0* **unfolding** *bp-invariant-def* **by** *blast*
have *distinct0*:
distinct (*map fst* (*bp-entries ?P0*))
using *inv0* **unfolding** *bp-invariant-def bp-distinct-keys-def* **by** *blast*
have *sizes0*:
∀ *c*∈*set* (*bp-buckets ?P0*).
length (*bp-bucket-entries c*) ≤ *bp-block-size ?P0*
using *inv0* **unfolding** *bp-invariant-def bp-bucket-sizes-ok-def* **by** *blast*
have *sorted0*:
sorted-wrt (λ*c d*. *bp-marker c* ≤ *bp-marker d*) (*bp-buckets ?P0*)
using *inv0* **unfolding** *bp-invariant-def bp-bucket-markers-sorted-def* **by** *blast*
have *markers0*:
∀ *c*∈*set* (*bp-buckets ?P0*). ∀ *p*∈*set* (*bp-bucket-entries c*).
bp-marker c ≤ *snd p*
using *inv0* **unfolding** *bp-invariant-def bp-bucket-markers-lower-bound-def* **by** *blast*
have *values0*:
∀ *p*∈*set* (*bp-entries ?P0*). *bp-values ?P0* (*fst p*) = *snd p*
using *inv0* **by** (*auto simp*: *bp-invariant-def bp-values-consistent-def*)
have *fresh*: *x* ∉ *bp-entry-keys* (*bp-entries ?P0*)
unfolding *bp-entries-delete-key* **by** *simp*
have *distinct-insert*:
distinct (*map fst*
(*bp-bucket-entries-flat* (*bp-insert-bucket* (*x*, *?b*) (*bp-buckets ?P0*))))
by (*rule bp-insert-bucket-distinct-keys*)
(*use distinct0 fresh* **in** ‹*simp-all add*: *bp-entries-def*›)
have *values0-P*:
∀ *p*∈*set* (*bp-entries ?P0*). *bp-values P* (*fst p*) = *snd p*
using *values0* **unfolding** *bp-delete-key-def* **by** *simp*
have *values-insert*:
∀ *p*∈*set*
(*bp-bucket-entries-flat* (*bp-insert-bucket* (*x*, *?b*) (*bp-buckets ?P0*))).
((*bp-values P*)(*x* := *?b*)) (*fst p*) = *snd p*
proof
fix *p*
assume *p*:
p ∈ *set* (*bp-bucket-entries-flat*
(*bp-insert-bucket* (*x*, *?b*) (*bp-buckets ?P0*)))
then have *p-cases*: *p* = (*x*, *?b*) ∨
p ∈ *set* (*bp-bucket-entries-flat* (*bp-buckets ?P0*))
using *set-bp-bucket-entries-flat-insert-bucket*[*of* (*x*, *?b*) *bp-buckets ?P0*]
by *auto*
from *p-cases* **show** ((*bp-values P*)(*x* := *?b*)) (*fst p*) = *snd p*

```
  proof
    assume p = (x, ?b)
    then show ?thesis by simp
  next
    assume p-flat: p ∈ set (bp-bucket-entries-flat (bp-buckets ?P0))
    then have p-old: p ∈ set (bp-entries ?P0)
      unfolding bp-entries-def .
    have fst p ≠ x
    proof
      assume fst p = x
      then have x ∈ bp-entry-keys (bp-entries ?P0)
        using p-old unfolding bp-entry-keys-def by auto
      then show False
        using fresh by simp
    qed
    moreover have bp-values P (fst p) = snd p
      using values0-P p-old by blast
    ultimately show ?thesis
      by simp
  qed
qed
show ?thesis
  unfolding bp-insert-def Let-def bp-invariant-def
proof (intro conjI)
  show 0 < bp-block-size
    (?P0(|bp-buckets := bp-insert-bucket (x, ?b) (bp-buckets ?P0),
      bp-values := (bp-values P)(x := ?b)|))
    using block-pos by simp
  show bp-distinct-keys
    (?P0(|bp-buckets := bp-insert-bucket (x, ?b) (bp-buckets ?P0),
      bp-values := (bp-values P)(x := ?b)|))
    using distinct-insert
    unfolding bp-distinct-keys-def bp-entries-def by simp
  show bp-bucket-sizes-ok
    (?P0(|bp-buckets := bp-insert-bucket (x, ?b) (bp-buckets ?P0),
      bp-values := (bp-values P)(x := ?b)|))
    using bp-insert-bucket-sizes-ok[OF sizes0 block-pos]
    unfolding bp-bucket-sizes-ok-def by simp
  show bp-bucket-markers-sorted
    (?P0(|bp-buckets := bp-insert-bucket (x, ?b) (bp-buckets ?P0),
      bp-values := (bp-values P)(x := ?b)|))
    using bp-insert-bucket-markers-sorted[OF sorted0]
    unfolding bp-bucket-markers-sorted-def by simp
  show bp-bucket-markers-lower-bound
    (?P0(|bp-buckets := bp-insert-bucket (x, ?b) (bp-buckets ?P0),
      bp-values := (bp-values P)(x := ?b)|))
    using bp-insert-bucket-markers-lower-bound[OF markers0]
    unfolding bp-bucket-markers-lower-bound-def by simp
  show bp-values-consistent
```

```
      (?P0(|bp-buckets := bp-insert-bucket (x, ?b) (bp-buckets ?P0),
        bp-values := (bp-values P)(x := ?b)|))
      using values-insert
      unfolding bp-values-consistent-def bp-entries-def by simp
  qed
qed

lemma bp-insert-keys [simp]:
  bp-entry-keys (bp-entries (bp-insert x b P)) =
    insert x (bp-entry-keys (bp-entries P))
proof −
  let ?b = if x ∈ bp-entry-keys (bp-entries P)
    then min (bp-values P x) b else b
  let ?P0 = bp-delete-key x P
  have bp-entry-keys
      (bp-entries
        (?P0(|bp-buckets := bp-insert-bucket (x, ?b) (bp-buckets ?P0),
          bp-values := (bp-values P)(x := ?b)|))) =
    insert x (bp-entry-keys (bp-entries ?P0))
    by simp
  also have ... = insert x (bp-entry-keys (bp-entries P))
    by simp
  finally show ?thesis
    unfolding bp-insert-def Let-def .
qed

lemma bp-insert-values [simp]:
  bp-values (bp-insert x b P) =
    (bp-values P)
      (x := if x ∈ bp-entry-keys (bp-entries P)
            then min (bp-values P x) b else b)
  unfolding bp-insert-def Let-def by simp

theorem bp-insert-refines-min-update:
  bp-view (bp-insert x b P) = min-update (bp-view P) x b
  unfolding bp-view-def min-update-def by simp

theorem bp-insert-refines-insert-spec:
  insert-spec (bp-view P) x b (bp-view (bp-insert x b P))
  unfolding bp-insert-refines-min-update
  by (rule min-update-insert-spec)

theorem bp-insert-preserves-upper-bound:
  assumes upper: partition-upper-bound (bp-view P) B
    and b-lt: b < B
  shows partition-upper-bound (bp-view (bp-insert x b P)) B
  unfolding bp-insert-refines-min-update
  by (rule min-update-preserves-upper-bound[OF upper b-lt])
```

```
lemma bp-local-insert-state-keys:
  assumes M-pos: 0 < bp-block-size P
  shows bp-entry-keys (bp-entries (bp-local-insert-state x b P)) =
    insert x (bp-entry-keys (bp-entries P))
proof −
  let ?b = if x ∈ bp-entry-keys (bp-entries P)
    then min (bp-values P x) b else b
  let ?P0 = bp-delete-key x P
  have M0: 0 < bp-block-size ?P0
    using M-pos unfolding bp-delete-key-def by simp
  have keys-step:
    bp-entry-keys
      (bp-entries
        (?P0(|bp-buckets :=
            bp-local-insert-bucket (bp-block-size ?P0) (x, ?b)
              (bp-buckets ?P0),
          bp-values := (bp-values P)(x := ?b)|))) =
     insert x (bp-entry-keys (bp-entries ?P0))
    using set-bp-bucket-entries-flat-local-insert-bucket
      [OF M0, of (x, ?b) bp-buckets ?P0]
    unfolding bp-entries-def bp-entry-keys-def by auto
  show ?thesis
    unfolding bp-local-insert-state-def Let-def
    using keys-step by simp
qed

lemma bp-local-insert-state-values [simp]:
  bp-values (bp-local-insert-state x b P) =
    (bp-values P)
      (x := if x ∈ bp-entry-keys (bp-entries P)
          then min (bp-values P x) b else b)
  unfolding bp-local-insert-state-def Let-def by simp

theorem bp-local-insert-state-refines-min-update:
  assumes 0 < bp-block-size P
  shows bp-view (bp-local-insert-state x b P) = min-update (bp-view P) x b
  using bp-local-insert-state-keys[OF assms, of x b]
  unfolding bp-view-def min-update-def by simp

theorem bp-local-insert-state-refines-insert-spec:
  assumes 0 < bp-block-size P
  shows insert-spec (bp-view P) x b
    (bp-view (bp-local-insert-state x b P))
  unfolding bp-local-insert-state-refines-min-update[OF assms]
  by (rule min-update-insert-spec)

theorem bp-local-insert-state-preserves-upper-bound:
  assumes M-pos: 0 < bp-block-size P
    and upper: partition-upper-bound (bp-view P) B
```

```
    and b-lt: b < B
  shows partition-upper-bound (bp-view (bp-local-insert-state x b P)) B
  unfolding bp-local-insert-state-refines-min-update[OF M-pos]
  by (rule min-update-preserves-upper-bound[OF upper b-lt])

lemma bp-lazy-insert-state-keys:
  assumes M-pos: 0 < bp-block-size P
  shows bp-entry-keys (bp-entries (bp-lazy-insert-state x b P)) =
    insert x (bp-entry-keys (bp-entries P))
proof −
  let ?b = if x ∈ bp-entry-keys (bp-entries P)
    then min (bp-values P x) b else b
  let ?P0 = bp-delete-key x P
  have M0: 0 < bp-block-size ?P0
    using M-pos unfolding bp-delete-key-def by simp
  have keys-step:
    bp-entry-keys
      (bp-entries
        (?P0(| bp-buckets :=
             bp-lazy-insert-bucket (bp-block-size ?P0) (x, ?b)
               (bp-buckets ?P0),
           bp-values := (bp-values P)(x := ?b) |))) =
     insert x (bp-entry-keys (bp-entries ?P0))
    using set-bp-bucket-entries-flat-lazy-insert-bucket
      [OF M0, of (x, ?b) bp-buckets ?P0]
    unfolding bp-entries-def bp-entry-keys-def by auto
  show ?thesis
    unfolding bp-lazy-insert-state-def Let-def
    using keys-step by simp
qed

lemma bp-lazy-insert-state-values [simp]:
  bp-values (bp-lazy-insert-state x b P) =
    (bp-values P)
      (x := if x ∈ bp-entry-keys (bp-entries P)
            then min (bp-values P x) b else b)
  unfolding bp-lazy-insert-state-def Let-def by simp

theorem bp-lazy-insert-state-refines-min-update:
  assumes 0 < bp-block-size P
  shows bp-view (bp-lazy-insert-state x b P) =
    min-update (bp-view P) x b
  using bp-lazy-insert-state-keys[OF assms, of x b]
  unfolding bp-view-def min-update-def by simp

theorem bp-lazy-insert-state-refines-insert-spec:
  assumes 0 < bp-block-size P
  shows insert-spec (bp-view P) x b
    (bp-view (bp-lazy-insert-state x b P))
```

unfolding *bp-lazy-insert-state-refines-min-update*[*OF assms*]
by (*rule min-update-insert-spec*)

theorem *bp-lazy-insert-state-preserves-upper-bound*:
assumes *M-pos*: *0* < *bp-block-size P*
and *upper*: *partition-upper-bound* (*bp-view P*) *B*
and *b-lt*: *b* < *B*
shows *partition-upper-bound* (*bp-view* (*bp-lazy-insert-state x b P*)) *B*
unfolding *bp-lazy-insert-state-refines-min-update*[*OF M-pos*]
by (*rule min-update-preserves-upper-bound*[*OF upper b-lt*])

lemma *bp-lazy-insert-state-lazy-bucket-sizes-ok*:
assumes *lazy*: *bp-lazy-ordered-invariant P*
shows *bp-lazy-bucket-sizes-ok* (*bp-lazy-insert-state x b P*)
proof −
let *?old-keys* = *bp-entry-keys* (*bp-entries P*)
let *?b* = *if x* ∈ *?old-keys then min* (*bp-values P x*) *b else b*
let *?P0* = *bp-delete-key x P*
have *inv*: *bp-lazy-invariant P*
by (*rule bp-lazy-ordered-invariant-lazy-invariant*[*OF lazy*])
have *inv0*: *bp-lazy-invariant ?P0*
by (*rule bp-delete-key-lazy-invariant*[*OF inv*])
have *M-pos*: *0* < *bp-block-size ?P0*
using *inv0* **unfolding** *bp-lazy-invariant-def* **by** *blast*
have *sizes0*:
∀ *c*∈*set* (*bp-buckets ?P0*).
length (*bp-bucket-entries c*) ≤ *2* ∗ *bp-block-size ?P0*
using *inv0* **unfolding** *bp-lazy-invariant-def*
bp-lazy-bucket-sizes-ok-def **by** *blast*
have *sizes*:
∀ *c*∈*set*
(*bp-lazy-insert-bucket* (*bp-block-size ?P0*) (*x*, *?b*)
(*bp-buckets ?P0*)).
length (*bp-bucket-entries c*) ≤ *2* ∗ *bp-block-size ?P0*
by (*rule bp-lazy-insert-bucket-sizes-ok*[*OF M-pos sizes0*])
show *?thesis*
unfolding *bp-lazy-insert-state-def Let-def bp-lazy-bucket-sizes-ok-def*
using *sizes* **by** *simp*
qed

lemma *bp-lazy-insert-state-markers-lower-bound*:
assumes *lazy*: *bp-lazy-ordered-invariant P*
shows *bp-bucket-markers-lower-bound* (*bp-lazy-insert-state x b P*)
proof −
let *?old-keys* = *bp-entry-keys* (*bp-entries P*)
let *?b* = *if x* ∈ *?old-keys then min* (*bp-values P x*) *b else b*
let *?P0* = *bp-delete-key x P*
have *inv*: *bp-lazy-invariant P*
by (*rule bp-lazy-ordered-invariant-lazy-invariant*[*OF lazy*])

```
  have inv0: bp-lazy-invariant ?P0
    by (rule bp-delete-key-lazy-invariant[OF inv])
  have M-pos: 0 < bp-block-size ?P0
    using inv0 unfolding bp-lazy-invariant-def by blast
  have lower0:
    ∀ c∈set (bp-buckets ?P0). ∀ p∈set (bp-bucket-entries c).
      bp-marker c ≤ snd p
    using inv0 unfolding bp-lazy-invariant-def
      bp-bucket-markers-lower-bound-def by blast
  have lower:
    ∀ c∈set
        (bp-lazy-insert-bucket (bp-block-size ?P0) (x, ?b)
          (bp-buckets ?P0)).
      ∀ q∈set (bp-bucket-entries c). bp-marker c ≤ snd q
    by (rule bp-lazy-insert-bucket-markers-lower-bound[OF M-pos lower0])
  show ?thesis
    unfolding bp-lazy-insert-state-def Let-def
      bp-bucket-markers-lower-bound-def
    using lower by simp
qed

lemma bp-lazy-insert-state-values-consistent:
  assumes lazy: bp-lazy-ordered-invariant P
  shows bp-values-consistent (bp-lazy-insert-state x b P)
proof −
  let ?old-keys = bp-entry-keys (bp-entries P)
  let ?b = if x ∈ ?old-keys then min (bp-values P x) b else b
  let ?P0 = bp-delete-key x P
  let ?M = bp-block-size ?P0
  let ?bs = bp-buckets ?P0
  let ?new-bs = bp-lazy-insert-bucket ?M (x, ?b) ?bs
  let ?P' = ?P0(|bp-buckets := ?new-bs,
      bp-values := (bp-values P)(x := ?b)|)
  have inv: bp-lazy-invariant P
    by (rule bp-lazy-ordered-invariant-lazy-invariant[OF lazy])
  have inv0: bp-lazy-invariant ?P0
    by (rule bp-delete-key-lazy-invariant[OF inv])
  have M-pos: 0 < ?M
    using inv0 unfolding bp-lazy-invariant-def by blast
  have values0:
    ∀ p∈set (bp-entries ?P0). bp-values ?P0 (fst p) = snd p
    using inv0 unfolding bp-lazy-invariant-def bp-values-consistent-def
    by blast
  have values0-P:
    ∀ p∈set (bp-entries ?P0). bp-values P (fst p) = snd p
    using values0 unfolding bp-delete-key-def by simp
  have fresh: x ∉ bp-entry-keys (bp-entries ?P0)
    by simp
  have values-insert:
```

```
    ∀ p∈set (bp-bucket-entries-flat ?new-bs).
      ((bp-values P)(x := ?b)) (fst p) = snd p
  proof
    fix p
    assume p: p ∈ set (bp-bucket-entries-flat ?new-bs)
    then have p-cases:
      p = (x, ?b) ∨ p ∈ set (bp-bucket-entries-flat ?bs)
      using set-bp-bucket-entries-flat-lazy-insert-bucket
        [OF M-pos, of (x, ?b) ?bs]
      by auto
    from p-cases show ((bp-values P)(x := ?b)) (fst p) = snd p
    proof
      assume p = (x, ?b)
      then show ?thesis by simp
    next
      assume p-flat: p ∈ set (bp-bucket-entries-flat ?bs)
      then have p-old: p ∈ set (bp-entries ?P0)
        unfolding bp-entries-def .
      have fst p ≠ x
      proof
        assume fst p = x
        then have x ∈ bp-entry-keys (bp-entries ?P0)
          using p-old unfolding bp-entry-keys-def by auto
        then show False
          using fresh by simp
      qed
      moreover have bp-values P (fst p) = snd p
        using values0-P p-old by blast
      ultimately show ?thesis
        by simp
    qed
  qed
  have state: bp-lazy-insert-state x b P = ?P'
    unfolding bp-lazy-insert-state-def Let-def by simp
  have bp-values-consistent ?P'
    using values-insert
    unfolding bp-values-consistent-def bp-entries-def by simp
  show ?thesis
    unfolding state by (rule ‹bp-values-consistent ?P'›)
qed

lemma bp-lazy-insert-state-distinct-keys:
  assumes lazy: bp-lazy-ordered-invariant P
  shows bp-distinct-keys (bp-lazy-insert-state x b P)
proof −
  let ?old-keys = bp-entry-keys (bp-entries P)
  let ?b = if x ∈ ?old-keys then min (bp-values P x) b else b
  let ?P0 = bp-delete-key x P
  let ?M = bp-block-size ?P0
```

```
  let ?bs = bp-buckets ?P0
  let ?new-bs = bp-lazy-insert-bucket ?M (x, ?b) ?bs
  let ?P' = ?P0(|bp-buckets := ?new-bs,
      bp-values := (bp-values P)(x := ?b)|)
  have inv: bp-lazy-invariant P
    by (rule bp-lazy-ordered-invariant-lazy-invariant[OF lazy])
  have inv0: bp-lazy-invariant ?P0
    by (rule bp-delete-key-lazy-invariant[OF inv])
  have M-pos: 0 < ?M
    using inv0 unfolding bp-lazy-invariant-def by blast
  have distinct0:
    distinct (map fst (bp-entries ?P0))
    using inv0 unfolding bp-lazy-invariant-def bp-distinct-keys-def
    by blast
  have fresh: x ∉ bp-entry-keys (bp-entries ?P0)
    by simp
  have distinct-lazy:
    distinct (map fst (bp-bucket-entries-flat ?new-bs))
    by (rule bp-lazy-insert-bucket-distinct-keys[OF M-pos])
      (use distinct0 fresh in ‹simp-all add: bp-entries-def›)
  have state: bp-lazy-insert-state x b P = ?P'
    unfolding bp-lazy-insert-state-def Let-def by simp
  have bp-distinct-keys ?P'
    using distinct-lazy unfolding bp-distinct-keys-def bp-entries-def
    by simp
  show ?thesis
    unfolding state by (rule ‹bp-distinct-keys ?P'›)
qed

lemma bp-lazy-insert-state-markers-sorted:
  assumes lazy: bp-lazy-ordered-invariant P
  shows bp-bucket-markers-sorted (bp-lazy-insert-state x b P)
proof −
  let ?old-keys = bp-entry-keys (bp-entries P)
  let ?b = if x ∈ ?old-keys then min (bp-values P x) b else b
  let ?P0 = bp-delete-key x P
  let ?M = bp-block-size ?P0
  let ?bs = bp-buckets ?P0
  let ?new-bs = bp-lazy-insert-bucket ?M (x, ?b) ?bs
  have inv: bp-lazy-invariant P
    by (rule bp-lazy-ordered-invariant-lazy-invariant[OF lazy])
  have inv0: bp-lazy-invariant ?P0
    by (rule bp-delete-key-lazy-invariant[OF inv])
  have boundaries0-state: bp-bucket-boundaries-state-ok ?P0
    by (rule bp-delete-key-boundaries-state-ok
        [OF bp-lazy-ordered-invariant-boundaries-state-ok[OF lazy]])
  have M-pos: 0 < ?M
    using inv0 unfolding bp-lazy-invariant-def by blast
  have sorted0:
```

```
    sorted-wrt (λc d. bp-marker c ≤ bp-marker d) ?bs
    using inv0 unfolding bp-lazy-invariant-def
      bp-bucket-markers-sorted-def by blast
  have lower0:
    ∀ c∈set ?bs. ∀ p∈set (bp-bucket-entries c).
      bp-marker c ≤ snd p
    using inv0 unfolding bp-lazy-invariant-def
      bp-bucket-markers-lower-bound-def by blast
  have boundaries0: bp-bucket-boundaries-ok ?bs
    using boundaries0-state unfolding bp-bucket-boundaries-state-ok-def .
  have sorted-new:
    sorted-wrt (λc d. bp-marker c ≤ bp-marker d) ?new-bs
    by (rule bp-lazy-insert-bucket-markers-sorted
        [OF M-pos sorted0 lower0 boundaries0])
  show ?thesis
    unfolding bp-lazy-insert-state-def Let-def
      bp-bucket-markers-sorted-def
    using sorted-new by simp
qed

lemma bp-lazy-insert-state-boundaries-state-ok:
  assumes lazy: bp-lazy-ordered-invariant P
  shows bp-bucket-boundaries-state-ok (bp-lazy-insert-state x b P)
proof −
  let ?old-keys = bp-entry-keys (bp-entries P)
  let ?b = if x ∈ ?old-keys then min (bp-values P x) b else b
  let ?P0 = bp-delete-key x P
  let ?M = bp-block-size ?P0
  let ?bs = bp-buckets ?P0
  let ?new-bs = bp-lazy-insert-bucket ?M (x, ?b) ?bs
  have inv: bp-lazy-invariant P
    by (rule bp-lazy-ordered-invariant-lazy-invariant[OF lazy])
  have inv0: bp-lazy-invariant ?P0
    by (rule bp-delete-key-lazy-invariant[OF inv])
  have boundaries0-state: bp-bucket-boundaries-state-ok ?P0
    by (rule bp-delete-key-boundaries-state-ok
        [OF bp-lazy-ordered-invariant-boundaries-state-ok[OF lazy]])
  have M-pos: 0 < ?M
    using inv0 unfolding bp-lazy-invariant-def by blast
  have sorted0:
    sorted-wrt (λc d. bp-marker c ≤ bp-marker d) ?bs
    using inv0 unfolding bp-lazy-invariant-def
      bp-bucket-markers-sorted-def by blast
  have lower0:
    ∀ c∈set ?bs. ∀ p∈set (bp-bucket-entries c).
      bp-marker c ≤ snd p
    using inv0 unfolding bp-lazy-invariant-def
      bp-bucket-markers-lower-bound-def by blast
  have boundaries0: bp-bucket-boundaries-ok ?bs
```

```
    using boundaries0-state unfolding bp-bucket-boundaries-state-ok-def .
  have boundaries-new: bp-bucket-boundaries-ok ?new-bs
    by (rule bp-lazy-insert-bucket-boundaries-ok
        [OF M-pos sorted0 lower0 boundaries0])
  show ?thesis
    unfolding bp-lazy-insert-state-def Let-def
      bp-bucket-boundaries-state-ok-def
    using boundaries-new by simp
qed

lemma bp-lazy-insert-state-lazy-invariant:
  assumes lazy: bp-lazy-ordered-invariant P
  shows bp-lazy-invariant (bp-lazy-insert-state x b P)
proof −
  have inv: bp-lazy-invariant P
    by (rule bp-lazy-ordered-invariant-lazy-invariant[OF lazy])
  have M-pos: 0 < bp-block-size P
    using inv unfolding bp-lazy-invariant-def by blast
  have block:
    bp-block-size (bp-lazy-insert-state x b P) = bp-block-size P
    unfolding bp-lazy-insert-state-def bp-delete-key-def
    by (simp add: Let-def split: if-splits)
  show ?thesis
    unfolding bp-lazy-invariant-def
  proof (intro conjI)
    show 0 < bp-block-size (bp-lazy-insert-state x b P)
      using M-pos block by simp
    show bp-distinct-keys (bp-lazy-insert-state x b P)
      by (rule bp-lazy-insert-state-distinct-keys[OF lazy])
    show bp-lazy-bucket-sizes-ok (bp-lazy-insert-state x b P)
      by (rule bp-lazy-insert-state-lazy-bucket-sizes-ok[OF lazy])
    show bp-bucket-markers-sorted (bp-lazy-insert-state x b P)
      by (rule bp-lazy-insert-state-markers-sorted[OF lazy])
    show bp-bucket-markers-lower-bound (bp-lazy-insert-state x b P)
      by (rule bp-lazy-insert-state-markers-lower-bound[OF lazy])
    show bp-values-consistent (bp-lazy-insert-state x b P)
      by (rule bp-lazy-insert-state-values-consistent[OF lazy])
  qed
qed

lemma bp-lazy-insert-state-lazy-ordered-invariant:
  assumes lazy: bp-lazy-ordered-invariant P
  shows bp-lazy-ordered-invariant (bp-lazy-insert-state x b P)
  unfolding bp-lazy-ordered-invariant-def
  using bp-lazy-insert-state-lazy-invariant[OF lazy]
    bp-lazy-insert-state-boundaries-state-ok[OF lazy]
  by blast

lemma length-bp-lazy-insert-state-entries:
```

```
  assumes M-pos: 0 < bp-block-size P
  shows length (bp-entries (bp-lazy-insert-state x b P)) =
    Suc (length (bp-entries (bp-delete-key x P)))
proof −
  let ?old-keys = bp-entry-keys (bp-entries P)
  let ?b = if x ∈ ?old-keys then min (bp-values P x) b else b
  let ?P0 = bp-delete-key x P
  let ?M = bp-block-size ?P0
  let ?new-bs = bp-lazy-insert-bucket ?M (x, ?b) (bp-buckets ?P0)
  let ?P′ = ?P0(|bp-buckets := ?new-bs,
      bp-values := (bp-values P)(x := ?b)|)
  have M0: 0 < ?M
    using M-pos unfolding bp-delete-key-def by simp
  have len:
    length (bp-bucket-entries-flat ?new-bs) =
      Suc (length (bp-bucket-entries-flat (bp-buckets ?P0)))
    by (rule length-bp-bucket-entries-flat-lazy-insert-bucket[OF M0])
  have state: bp-lazy-insert-state x b P = ?P′
    unfolding bp-lazy-insert-state-def Let-def by simp
  have length (bp-entries ?P′) =
    Suc (length (bp-entries ?P0))
    using len unfolding bp-entries-def by simp
  show ?thesis
    unfolding state by (rule ‹length (bp-entries ?P′) =
      Suc (length (bp-entries ?P0))›)
qed

lemma length-bp-lazy-insert-state-entries-ge:
  assumes lazy: bp-lazy-ordered-invariant P
  shows length (bp-entries P) ≤
    length (bp-entries (bp-lazy-insert-state x b P))
proof −
  have inv: bp-lazy-invariant P
    by (rule bp-lazy-ordered-invariant-lazy-invariant[OF lazy])
  have M-pos: 0 < bp-block-size P
    using inv unfolding bp-lazy-invariant-def by blast
  have distinct: bp-distinct-keys P
    using inv unfolding bp-lazy-invariant-def by blast
  have delete-ge:
    length (bp-entries P) ≤
      Suc (length (bp-entries (bp-delete-key x P)))
    by (rule length-bp-entries-delete-key-ge-pred[OF distinct])
  have insert-len:
    length (bp-entries (bp-lazy-insert-state x b P)) =
      Suc (length (bp-entries (bp-delete-key x P)))
    by (rule length-bp-lazy-insert-state-entries[OF M-pos])
  show ?thesis
    using delete-ge unfolding insert-len .
qed
```

lemma *length-bp-lazy-insert-state-buckets-le*:
assumes *lazy*: *bp-lazy-ordered-invariant P*
shows *length* (*bp-buckets* (*bp-lazy-insert-state x b P*)) ≤
length (*bp-buckets P*) + *2*
proof −
let *?old-keys* = *bp-entry-keys* (*bp-entries P*)
let *?b* = *if x* ∈ *?old-keys then min* (*bp-values P x*) *b else b*
let *?P0* = *bp-delete-key x P*
let *?M* = *bp-block-size ?P0*
let *?bs* = *bp-buckets ?P0*
let *?new-bs* = *bp-lazy-insert-bucket ?M* (*x*, *?b*) *?bs*
let *?P′* = *?P0*(|*bp-buckets* := *?new-bs*,
bp-values := (*bp-values P*)(*x* := *?b*)|)
have *inv*: *bp-lazy-invariant P*
by (*rule bp-lazy-ordered-invariant-lazy-invariant*[*OF lazy*])
have *inv0*: *bp-lazy-invariant ?P0*
by (*rule bp-delete-key-lazy-invariant*[*OF inv*])
have *M-pos*: *0* < *?M*
using *inv0* **unfolding** *bp-lazy-invariant-def* **by** *blast*
have *sizes0*:
∀ *b*∈*set ?bs*. *length* (*bp-bucket-entries b*) ≤ *2* ∗ *?M*
using *inv0* **unfolding** *bp-lazy-invariant-def*
bp-lazy-bucket-sizes-ok-def **by** *blast*
have *len-new*:
length ?new-bs ≤ *length ?bs* + *2*
by (*rule length-bp-lazy-insert-bucket-le*[*OF M-pos sizes0*])
have *state*: *bp-lazy-insert-state x b P* = *?P′*
unfolding *bp-lazy-insert-state-def Let-def* **by** *simp*
have *length* (*bp-buckets ?P′*) ≤ *length* (*bp-buckets P*) + *2*
using *len-new* **unfolding** *bp-delete-key-def* **by** *simp*
show *?thesis*
unfolding *state* **by** (*rule* ‹*length* (*bp-buckets ?P′*) ≤
length (*bp-buckets P*) + *2*›)
qed

lemma *bp-local-insert-state-invariant*:
assumes *ord*: *bp-ordered-invariant P*
shows *bp-invariant* (*bp-local-insert-state x b P*)
proof −
let *?old-keys* = *bp-entry-keys* (*bp-entries P*)
let *?b* = *if x* ∈ *?old-keys then min* (*bp-values P x*) *b else b*
let *?P0* = *bp-delete-key x P*
let *?M* = *bp-block-size ?P0*
let *?bs* = *bp-buckets ?P0*
let *?new-bs* = *bp-local-insert-bucket ?M* (*x*, *?b*) *?bs*
let *?P′* = *?P0*(|*bp-buckets* := *?new-bs*,
bp-values := (*bp-values P*)(*x* := *?b*)|)
have *inv*: *bp-invariant P*

```
  by (rule bp-ordered-invariant-invariant[OF ord])
have inv0: bp-invariant ?P0
  by (rule bp-delete-key-invariant[OF inv])
have boundaries0-state: bp-bucket-boundaries-state-ok ?P0
  by (rule bp-delete-key-boundaries-state-ok
      [OF bp-ordered-invariant-boundaries-state-ok[OF ord]])
have M-pos: 0 < ?M
  using inv0 unfolding bp-invariant-def by blast
have distinct0: distinct (map fst (bp-entries ?P0))
  using inv0 unfolding bp-invariant-def bp-distinct-keys-def by blast
have sizes0:
  ∀ c∈set ?bs. length (bp-bucket-entries c) ≤ ?M
  using inv0 unfolding bp-invariant-def bp-bucket-sizes-ok-def by blast
have sorted0:
  sorted-wrt (λc d. bp-marker c ≤ bp-marker d) ?bs
  using inv0 unfolding bp-invariant-def bp-bucket-markers-sorted-def by blast
have markers0:
  ∀ c∈set ?bs. ∀ p∈set (bp-bucket-entries c).
    bp-marker c ≤ snd p
  using inv0 unfolding bp-invariant-def bp-bucket-markers-lower-bound-def
  by blast
have boundaries0: bp-bucket-boundaries-ok ?bs
  using boundaries0-state unfolding bp-bucket-boundaries-state-ok-def .
have shape:
  (∀ c∈set ?new-bs. length (bp-bucket-entries c) ≤ ?M) ∧
   sorted-wrt (λc d. bp-marker c ≤ bp-marker d) ?new-bs ∧
   (∀ c∈set ?new-bs. ∀ q∈set (bp-bucket-entries c).
      bp-marker c ≤ snd q) ∧
   bp-bucket-boundaries-ok ?new-bs
  by (rule bp-local-insert-bucket-preserves-bucket-shape
      [OF M-pos sizes0 sorted0 markers0 boundaries0])
have fresh: x ∉ bp-entry-keys (bp-entries ?P0)
  by simp
have distinct-local:
  distinct (map fst (bp-bucket-entries-flat ?new-bs))
  by (rule bp-local-insert-bucket-distinct-keys[OF M-pos])
    (use distinct0 fresh in ‹simp-all add: bp-entries-def›)
have values0:
  ∀ p∈set (bp-entries ?P0). bp-values ?P0 (fst p) = snd p
  using inv0 unfolding bp-invariant-def bp-values-consistent-def by blast
have values0-P:
  ∀ p∈set (bp-entries ?P0). bp-values P (fst p) = snd p
  using values0 unfolding bp-delete-key-def by simp
have values-insert:
  ∀ p∈set (bp-bucket-entries-flat ?new-bs).
    ((bp-values P)(x := ?b)) (fst p) = snd p
proof
  fix p
  assume p: p ∈ set (bp-bucket-entries-flat ?new-bs)
```

```
    then have p-cases:
      p = (x, ?b) ∨ p ∈ set (bp-bucket-entries-flat ?bs)
      using set-bp-bucket-entries-flat-local-insert-bucket
        [OF M-pos, of (x, ?b) ?bs]
      by auto
    from p-cases show ((bp-values P)(x := ?b)) (fst p) = snd p
    proof
      assume p = (x, ?b)
      then show ?thesis by simp
    next
      assume p-old: p ∈ set (bp-bucket-entries-flat ?bs)
      then have p-old-entry: p ∈ set (bp-entries ?P0)
        unfolding bp-entries-def .
      have fst p ≠ x
      proof
        assume fst p = x
        then have x ∈ bp-entry-keys (bp-entries ?P0)
          using p-old-entry unfolding bp-entry-keys-def by auto
        then show False
          using fresh by simp
      qed
      moreover have bp-values P (fst p) = snd p
        using values0-P p-old-entry by blast
      ultimately show ?thesis
        by simp
    qed
  qed
  show ?thesis
    unfolding bp-local-insert-state-def Let-def bp-invariant-def
  proof (intro conjI)
    show 0 < bp-block-size ?P'
      using M-pos by simp
    show bp-distinct-keys ?P'
      using distinct-local unfolding bp-distinct-keys-def bp-entries-def
      by simp
    show bp-bucket-sizes-ok ?P'
      using shape unfolding bp-bucket-sizes-ok-def by simp
    show bp-bucket-markers-sorted ?P'
      using shape unfolding bp-bucket-markers-sorted-def by simp
    show bp-bucket-markers-lower-bound ?P'
      using shape unfolding bp-bucket-markers-lower-bound-def by simp
    show bp-values-consistent ?P'
      using values-insert unfolding bp-values-consistent-def bp-entries-def
      by simp
  qed
qed

lemma bp-local-insert-state-boundaries-state-ok:
  assumes ord: bp-ordered-invariant P
```

shows *bp-bucket-boundaries-state-ok* (*bp-local-insert-state x b P*)
proof −
let *?old-keys* = *bp-entry-keys* (*bp-entries P*)
let *?b* = *if x* ∈ *?old-keys then min* (*bp-values P x*) *b else b*
let *?P0* = *bp-delete-key x P*
let *?M* = *bp-block-size ?P0*
let *?bs* = *bp-buckets ?P0*
let *?new-bs* = *bp-local-insert-bucket ?M* (*x*, *?b*) *?bs*
have *inv*: *bp-invariant P*
by (*rule bp-ordered-invariant-invariant*[*OF ord*])
have *inv0*: *bp-invariant ?P0*
by (*rule bp-delete-key-invariant*[*OF inv*])
have *boundaries0-state*: *bp-bucket-boundaries-state-ok ?P0*
by (*rule bp-delete-key-boundaries-state-ok*
[*OF bp-ordered-invariant-boundaries-state-ok*[*OF ord*]])
have *M-pos*: *0* < *?M*
using *inv0* **unfolding** *bp-invariant-def* **by** *blast*
have *sizes0*:
∀ *c*∈*set ?bs*. *length* (*bp-bucket-entries c*) ≤ *?M*
using *inv0* **unfolding** *bp-invariant-def bp-bucket-sizes-ok-def* **by** *blast*
have *sorted0*:
sorted-wrt (λ*c d*. *bp-marker c* ≤ *bp-marker d*) *?bs*
using *inv0* **unfolding** *bp-invariant-def bp-bucket-markers-sorted-def* **by** *blast*
have *markers0*:
∀ *c*∈*set ?bs*. ∀ *p*∈*set* (*bp-bucket-entries c*).
bp-marker c ≤ *snd p*
using *inv0* **unfolding** *bp-invariant-def bp-bucket-markers-lower-bound-def*
by *blast*
have *boundaries0*: *bp-bucket-boundaries-ok ?bs*
using *boundaries0-state* **unfolding** *bp-bucket-boundaries-state-ok-def* **.**
have *shape*:
(∀ *c*∈*set ?new-bs*. *length* (*bp-bucket-entries c*) ≤ *?M*) ∧
sorted-wrt (λ*c d*. *bp-marker c* ≤ *bp-marker d*) *?new-bs* ∧
(∀ *c*∈*set ?new-bs*. ∀ *q*∈*set* (*bp-bucket-entries c*).
bp-marker c ≤ *snd q*) ∧
bp-bucket-boundaries-ok ?new-bs
by (*rule bp-local-insert-bucket-preserves-bucket-shape*
[*OF M-pos sizes0 sorted0 markers0 boundaries0*])
show *?thesis*
unfolding *bp-local-insert-state-def Let-def*
bp-bucket-boundaries-state-ok-def
using *shape* **by** *simp*
qed

lemma *bp-local-insert-state-ordered-invariant*:
assumes *ord*: *bp-ordered-invariant P*
shows *bp-ordered-invariant* (*bp-local-insert-state x b P*)
unfolding *bp-ordered-invariant-def*
using *bp-local-insert-state-invariant*[*OF ord*]

```
    bp-local-insert-state-boundaries-state-ok[OF ord]
  by blast

lemma bp-batch-prepend-invariant:
  assumes bp-invariant P
  shows bp-invariant (bp-batch-prepend xs P)
  using assms
proof (induction xs arbitrary: P)
  case Nil
  then show ?case
    unfolding bp-batch-prepend-def by simp
next
  case (Cons xb xs)
  obtain x b where xb: xb = (x, b)
    by force
  have step: bp-invariant (bp-insert x b P)
    by (rule bp-insert-invariant[OF Cons.prems])
  have unfold-step:
    bp-batch-prepend (xb # xs) P =
      bp-batch-prepend xs (bp-insert x b P)
    unfolding bp-batch-prepend-def xb by simp
  show ?case
    unfolding unfold-step by (rule Cons.IH[OF step])
qed

theorem bp-batch-prepend-refines-batch-min-update:
  assumes bp-invariant P
  shows bp-view (bp-batch-prepend xs P) =
    batch-min-update (bp-view P) xs
  using assms
proof (induction xs arbitrary: P)
  case Nil
  then show ?case
    unfolding bp-batch-prepend-def batch-min-update-def by simp
next
  case (Cons xb xs)
  obtain x b where xb: xb = (x, b)
    by force
  have step-inv: bp-invariant (bp-insert x b P)
    by (rule bp-insert-invariant[OF Cons.prems])
  have step-view: bp-view (bp-insert x b P) = min-update (bp-view P) x b
    by (rule bp-insert-refines-min-update)
  have unfold-state:
    bp-batch-prepend (xb # xs) P =
      bp-batch-prepend xs (bp-insert x b P)
    unfolding bp-batch-prepend-def xb by simp
  have unfold-view:
    batch-min-update (bp-view P) (xb # xs) =
      batch-min-update (min-update (bp-view P) x b) xs
```

```
    unfolding batch-min-update-def xb by simp
  show ?case
    unfolding unfold-state unfold-view
    using Cons.IH[OF step-inv] step-view by simp
qed

theorem bp-batch-prepend-preserves-upper-bound:
  assumes inv: bp-invariant P
    and upper: partition-upper-bound (bp-view P) B
    and values-lt: ⋀x b. (x, b) ∈ set xs ⟹ b < B
  shows partition-upper-bound (bp-view (bp-batch-prepend xs P)) B
  unfolding bp-batch-prepend-refines-batch-min-update[OF inv]
  by (rule batch-min-update-preserves-upper-bound[OF upper values-lt])

definition bp-rebucketed-insert ::
  'k ⇒ real ⇒ 'k bucketed-partition ⇒ 'k bucketed-partition where
  bp-rebucketed-insert x b P = bp-rebucket (bp-insert x b P)

definition bp-rebucketed-batch-prepend ::
  ('k × real) list ⇒ 'k bucketed-partition ⇒
    'k bucketed-partition where
  bp-rebucketed-batch-prepend xs P = bp-rebucket (bp-batch-prepend xs P)

lemma bp-rebucketed-insert-invariant:
  assumes inv: bp-invariant P
  shows bp-invariant (bp-rebucketed-insert x b P)
  unfolding bp-rebucketed-insert-def
  by (rule bp-rebucket-invariant[OF bp-insert-invariant[OF inv]])

lemma bp-rebucketed-insert-ordered-invariant:
  assumes inv: bp-invariant P
  shows bp-ordered-invariant (bp-rebucketed-insert x b P)
  unfolding bp-rebucketed-insert-def
  by (rule bp-rebucket-ordered-invariant[OF bp-insert-invariant[OF inv]])

theorem bp-rebucketed-insert-refines-min-update:
  assumes inv: bp-invariant P
  shows bp-view (bp-rebucketed-insert x b P) = min-update (bp-view P) x b
proof −
  have step-inv: bp-invariant (bp-insert x b P)
    by (rule bp-insert-invariant[OF inv])
  then have block: 0 < bp-block-size (bp-insert x b P)
    unfolding bp-invariant-def by blast
  have rebucket-view:
    bp-view (bp-rebucket (bp-insert x b P)) = bp-view (bp-insert x b P)
    by (rule bp-rebucket-view[OF block])
  show ?thesis
    unfolding bp-rebucketed-insert-def
    using rebucket-view bp-insert-refines-min-update by simp
```

qed

theorem *bp-rebucketed-insert-refines-insert-spec*:
assumes *inv*: *bp-invariant P*
shows *insert-spec* (*bp-view P*) *x b* (*bp-view* (*bp-rebucketed-insert x b P*))
unfolding *bp-rebucketed-insert-refines-min-update*[*OF inv*]
by (*rule min-update-insert-spec*)

theorem *bp-rebucketed-insert-preserves-upper-bound*:
assumes *inv*: *bp-invariant P*
and *upper*: *partition-upper-bound* (*bp-view P*) *B*
and *b-lt*: *b* < *B*
shows *partition-upper-bound* (*bp-view* (*bp-rebucketed-insert x b P*)) *B*
unfolding *bp-rebucketed-insert-refines-min-update*[*OF inv*]
by (*rule min-update-preserves-upper-bound*[*OF upper b-lt*])

lemma *bp-rebucketed-batch-prepend-invariant*:
assumes *inv*: *bp-invariant P*
shows *bp-invariant* (*bp-rebucketed-batch-prepend xs P*)
unfolding *bp-rebucketed-batch-prepend-def*
by (*rule bp-rebucket-invariant*[*OF bp-batch-prepend-invariant*[*OF inv*]])

lemma *bp-rebucketed-batch-prepend-ordered-invariant*:
assumes *inv*: *bp-invariant P*
shows *bp-ordered-invariant* (*bp-rebucketed-batch-prepend xs P*)
unfolding *bp-rebucketed-batch-prepend-def*
by (*rule bp-rebucket-ordered-invariant*[*OF bp-batch-prepend-invariant*[*OF inv*]])

theorem *bp-rebucketed-batch-prepend-refines-batch-min-update*:
assumes *inv*: *bp-invariant P*
shows *bp-view* (*bp-rebucketed-batch-prepend xs P*) =
batch-min-update (*bp-view P*) *xs*
proof −
have *step-inv*: *bp-invariant* (*bp-batch-prepend xs P*)
by (*rule bp-batch-prepend-invariant*[*OF inv*])
then have *block*: *0* < *bp-block-size* (*bp-batch-prepend xs P*)
unfolding *bp-invariant-def* **by** *blast*
have *rebucket-view*:
bp-view (*bp-rebucket* (*bp-batch-prepend xs P*)) =
bp-view (*bp-batch-prepend xs P*)
by (*rule bp-rebucket-view*[*OF block*])
show *?thesis*
unfolding *bp-rebucketed-batch-prepend-def*
using *rebucket-view bp-batch-prepend-refines-batch-min-update*[*OF inv*]
by *simp*
qed

theorem *bp-rebucketed-batch-prepend-preserves-upper-bound*:
assumes *inv*: *bp-invariant P*

```
    and upper: partition-upper-bound (bp-view P) B
    and values-lt: ⋀x b. (x, b) ∈ set xs ⟹ b < B
  shows partition-upper-bound
    (bp-view (bp-rebucketed-batch-prepend xs P)) B
  unfolding bp-rebucketed-batch-prepend-refines-batch-min-update[OF inv]
  by (rule batch-min-update-preserves-upper-bound[OF upper values-lt])

lemma bp-min-value-le-entry:
  assumes (x, b) ∈ set xs
  shows bp-min-value B xs ≤ b
  using assms by (induction xs) auto

lemma bp-entries-Cons:
  assumes bp-buckets P = b # bs
  shows bp-entries P = bp-bucket-entries b @ bp-bucket-entries-flat bs
  using assms unfolding bp-entries-def bp-bucket-entries-flat-def by simp

lemma bp-tail-entry-ge-head-marker:
  assumes sorted: sorted-wrt (λb c. bp-marker b ≤ bp-marker c) (c # bs)
    and lower: ⋀d p. ⟦d ∈ set (c # bs); p ∈ set (bp-bucket-entries d)⟧
      ⟹ bp-marker d ≤ snd p
    and p: p ∈ set (bp-bucket-entries-flat (c # bs))
  shows bp-marker c ≤ snd p
proof −
  obtain d where d: d ∈ set (c # bs)
    and p-d: p ∈ set (bp-bucket-entries d)
    using p unfolding bp-bucket-entries-flat-def by auto
  have c-le-d: bp-marker c ≤ bp-marker d
    using sorted d by (cases d = c) auto
  have d-le-p: bp-marker d ≤ snd p
    by (rule lower[OF d p-d])
  show ?thesis
    using c-le-d d-le-p by linarith
qed

lemma bp-first-bucket-pull-invariant:
  assumes inv: bp-invariant P
    and pull: bp-first-bucket-pull M B P = (S, beta, P')
  shows bp-invariant P'
proof (cases bp-buckets P)
  case Nil
  then have P' = P
    using pull unfolding bp-first-bucket-pull-def by simp
  then show ?thesis
    using inv by simp
next
  case (Cons b rest)
  then have buckets-outer: bp-buckets P = b # rest .
  then show ?thesis
```

```
  proof (cases rest)
    case Nil
    then have P' = P
      using buckets-outer pull unfolding bp-first-bucket-pull-def by simp
    then show ?thesis
      using inv by simp
  next
    case (Cons c bs)
    let ?S = bp-bucket-keys b
    have pull-eval:
      bp-first-bucket-pull M B P = (?S, bp-marker c, bp-delete-keys ?S P)
      using buckets-outer Cons
      unfolding bp-first-bucket-pull-def by (simp add: Let-def)
    have tuple-eq: (?S, bp-marker c, bp-delete-keys ?S P) = (S, beta, P')
      using pull-eval pull by simp
    have S-eq: S = ?S
      using tuple-eq by simp
    have P'-eq-raw: bp-delete-keys ?S P = P'
      using tuple-eq by (metis prod.inject)
    then have P' = bp-delete-keys ?S P
      by simp
    then have P' = bp-delete-keys S P
      using S-eq by simp
    then show ?thesis
      by (simp add: bp-delete-keys-invariant[OF inv])
  qed
qed

theorem bp-first-bucket-pull-refines-pull-separates:
  assumes inv: bp-invariant P
    and buckets: bp-buckets P = b # c # bs
    and len-b: length (bp-bucket-entries b) ≤ M
    and below: bp-bucket-below-bound b (bp-marker c)
    and tail-nonempty: bp-bucket-entries-flat (c # bs) ≠ []
    and pull: bp-first-bucket-pull M B P = (S, beta, P')
  shows pull-separates (bp-view P) M B S beta (bp-view P')
proof −
  let ?tail = bp-bucket-entries-flat (c # bs)
  have S-def: S = bp-bucket-keys b
    and beta-def: beta = bp-marker c
    and P'-def: P' = bp-delete-keys S P
    using pull buckets unfolding bp-first-bucket-pull-def by (auto simp: Let-def)
  have entries-P: bp-entries P = bp-bucket-entries b @ ?tail
    using buckets by (simp add: bp-entries-Cons)
  have distinct-P: distinct (map fst (bp-entries P))
    using inv unfolding bp-invariant-def bp-distinct-keys-def by blast
  have values-P:
    ⋀p. p ∈ set (bp-entries P) ⟹ bp-values P (fst p) = snd p
    using inv unfolding bp-invariant-def bp-values-consistent-def by blast
```

```
  have sorted-tail:
    sorted-wrt (λx y. bp-marker x ≤ bp-marker y) (c # bs)
    using inv buckets unfolding bp-invariant-def bp-bucket-markers-sorted-def by
simp
  have lower-tail:
    ⋀d p. ⟦d ∈ set (c # bs); p ∈ set (bp-bucket-entries d)⟧
      ⟹ bp-marker d ≤ snd p
    using inv buckets unfolding bp-invariant-def
      bp-bucket-markers-lower-bound-def by auto
  have S-subset: S ⊆ keys-of (bp-view P)
    unfolding S-def bp-view-def bp-bucket-keys-def bp-entry-keys-def entries-P
    by auto
  have card-S: card S ≤ M
  proof −
    have card S ≤ length (bp-bucket-entries b)
    proof −
      have card S = card (fst ‘ set (bp-bucket-entries b))
        unfolding S-def bp-bucket-keys-def bp-entry-keys-def by simp
      also have ... ≤ card (set (bp-bucket-entries b))
        by (rule card-image-le) simp
      also have ... ≤ length (bp-bucket-entries b)
        by (rule card-length)
      finally show ?thesis .
    qed
    then show ?thesis
      using len-b by linarith
  qed
  have P′-keys: keys-of (bp-view P′) = keys-of (bp-view P) − S
    unfolding P′-def bp-delete-keys-view by simp
  have P′-values: value-of (bp-view P′) = value-of (bp-view P)
    unfolding P′-def bp-delete-keys-view by simp
  have pulled-lt-beta:
    ⋀u. u ∈ S ⟹ value-of (bp-view P) u < beta
  proof −
    fix u
    assume u: u ∈ S
    then obtain p where p-b: p ∈ set (bp-bucket-entries b)
      and fst-p: fst p = u
      unfolding S-def bp-bucket-keys-def bp-entry-keys-def by auto
    have snd p < beta
      using below p-b unfolding bp-bucket-below-bound-def beta-def by blast
    moreover have bp-values P u = snd p
      using values-P[of p] p-b fst-p entries-P by simp
    ultimately show value-of (bp-view P) u < beta
      unfolding bp-view-def by simp
  qed
  have tail-key-notin-S:
    ⋀p. p ∈ set ?tail ⟹ fst p ∉ S
  proof −
```

```
  fix p
  assume p-tail: p ∈ set ?tail
  show fst p ∉ S
  proof
    assume fst p ∈ S
    then obtain q where q-b: q ∈ set (bp-bucket-entries b)
      and fst-q: fst q = fst p
      unfolding S-def bp-bucket-keys-def bp-entry-keys-def by auto
    have distinct-append: distinct (map fst (bp-bucket-entries b @ ?tail))
      using distinct-P unfolding entries-P .
    have disjoint:
      set (map fst (bp-bucket-entries b)) ∩ set (map fst ?tail) = {}
      using distinct-append by simp
    have fst p ∈ set (map fst ?tail)
      using p-tail by force
    then have fst p ∉ set (map fst (bp-bucket-entries b))
      using disjoint by blast
    moreover have fst p ∈ set (map fst (bp-bucket-entries b))
    proof −
      have fst q ∈ set (map fst (bp-bucket-entries b))
        using q-b by force
      then show ?thesis
        using fst-q by simp
    qed
    ultimately show False
      by blast
  qed
qed
have tail-key-remaining:
  ⋀p. p ∈ set ?tail ⟹ fst p ∈ keys-of (bp-view P′)
proof −
  fix p
  assume p-tail: p ∈ set ?tail
  have old: fst p ∈ keys-of (bp-view P)
    unfolding bp-view-def bp-entry-keys-def entries-P using p-tail by auto
  have not-S: fst p ∉ S
    by (rule tail-key-notin-S[OF p-tail])
  show fst p ∈ keys-of (bp-view P′)
    using old not-S unfolding P′-keys by blast
qed
have remaining-nonempty: keys-of (bp-view P′) ≠ {}
proof −
  obtain p where p-tail: p ∈ set ?tail
    using tail-nonempty by (cases ?tail) auto
  then have fst p ∈ keys-of (bp-view P′)
    by (rule tail-key-remaining)
  then show ?thesis by blast
qed
have lower-remaining:
```

```
  ⋀v. v ∈ keys-of (bp-view P′) ⟹ beta ≤ value-of (bp-view P′) v
proof −
  fix v
  assume v: v ∈ keys-of (bp-view P′)
  then have v-old: v ∈ keys-of (bp-view P)
    using P′-keys by blast
  then obtain p where p-entries: p ∈ set (bp-entries P)
    and fst-p: fst p = v
    unfolding bp-view-def bp-entry-keys-def by auto
  have v-not-S: v ∉ S
    using v P′-keys by blast
  have p-tail: p ∈ set ?tail
  proof −
    have p ∉ set (bp-bucket-entries b)
      using v-not-S fst-p unfolding S-def bp-bucket-keys-def bp-entry-keys-def
      by auto
    then show ?thesis
      using p-entries unfolding entries-P by auto
  qed
  have beta ≤ snd p
    unfolding beta-def
    by (rule bp-tail-entry-ge-head-marker[OF sorted-tail lower-tail p-tail])
  moreover have value-of (bp-view P′) v = snd p
  proof −
    have value-of (bp-view P) v = snd p
      using values-P[OF p-entries] fst-p unfolding bp-view-def by simp
    moreover have value-of (bp-view P′) v = value-of (bp-view P) v
      using P′-values by simp
    ultimately show ?thesis by simp
  qed
  ultimately show beta ≤ value-of (bp-view P′) v
    by simp
qed
have pulled-before-remaining:
  ⋀u v. ⟦u ∈ S; v ∈ keys-of (bp-view P′)⟧
    ⟹ value-of (bp-view P) u ≤ value-of (bp-view P′) v
proof −
  fix u v
  assume u: u ∈ S and v: v ∈ keys-of (bp-view P′)
  have value-of (bp-view P) u < beta
    by (rule pulled-lt-beta[OF u])
  moreover have beta ≤ value-of (bp-view P′) v
    by (rule lower-remaining[OF v])
  ultimately show value-of (bp-view P) u ≤ value-of (bp-view P′) v
    by simp
qed
show ?thesis
  unfolding pull-separates-def
  using S-subset card-S P′-keys P′-values remaining-nonempty
```

```
    pulled-before-remaining pulled-lt-beta lower-remaining
  by auto
qed

lemma bp-conservative-pull-invariant:
 assumes inv: bp-invariant P
  and pull: bp-conservative-pull M B P = (S, beta, P')
 shows bp-invariant P'
proof −
 have P' = bp-delete-keys S P
  using pull unfolding bp-conservative-pull-def by (auto simp: Let-def)
 then show ?thesis
  by (simp add: bp-delete-keys-invariant[OF inv])
qed

theorem bp-conservative-pull-refines-pull-separates:
 assumes inv: bp-invariant P
  and upper: ⋀u. u ∈ keys-of (bp-view P) ⟹
    value-of (bp-view P) u < B
  and pull: bp-conservative-pull M B P = (S, beta, P')
 shows pull-separates (bp-view P) M B S beta (bp-view P')
proof −
 have S-def: S = bp-pull-set M P
  and beta-def: beta = bp-pull-bound M B P
  and P'-def: P' = bp-delete-keys S P
  using pull unfolding bp-conservative-pull-def by (auto simp: Let-def)
 show ?thesis
 proof (cases length (bp-entries P) ≤ M)
  case True
  have S-keys: S = keys-of (bp-view P)
   using True unfolding S-def bp-pull-set-def bp-view-def by simp
  have beta-B: beta = B
   using True unfolding beta-def bp-pull-bound-def by simp
  have keys-empty: keys-of (bp-view P') = {}
   unfolding P'-def bp-delete-keys-view S-keys by simp
  have card-S: card S ≤ M
  proof −
   have card S = card (fst ‘ set (bp-entries P))
    unfolding S-keys bp-view-def bp-entry-keys-def by simp
   also have ... ≤ card (set (bp-entries P))
    by (rule card-image-le) simp
   also have ... ≤ length (bp-entries P)
    by (rule card-length)
   also have ... ≤ M
    by (rule True)
   finally show ?thesis .
  qed
  show ?thesis
   unfolding pull-separates-def P'-def bp-delete-keys-view
```

```
    using S-keys beta-B keys-empty card-S by auto
  next
    case False
    have S-empty: S = {}
      using False unfolding S-def bp-pull-set-def by simp
    have beta-min: beta = bp-min-value B (bp-entries P)
      using False unfolding beta-def bp-pull-bound-def by simp
    have P'-same: P' = P
      unfolding P'-def S-empty bp-delete-keys-def bp-delete-keys-from-bucket-def
      by simp
    have keys-nonempty: keys-of (bp-view P') ≠ {}
    proof
      assume empty: keys-of (bp-view P') = {}
      then have set (bp-entries P) = {}
        unfolding P'-same bp-view-def bp-entry-keys-def by auto
      then have length (bp-entries P) = 0
        by simp
      then show False
        using False by simp
    qed
    have lower:
      ⋀v. v ∈ keys-of (bp-view P') ⟹ beta ≤ value-of (bp-view P') v
    proof −
      fix v
      assume v: v ∈ keys-of (bp-view P')
      then obtain b where vb: (v, b) ∈ set (bp-entries P)
        unfolding P'-same bp-view-def bp-entry-keys-def by auto
      have bp-min-value B (bp-entries P) ≤ b
        by (rule bp-min-value-le-entry[OF vb])
      moreover have bp-values P v = b
        using inv vb by (auto simp: bp-invariant-def bp-values-consistent-def)
      ultimately show beta ≤ value-of (bp-view P') v
        unfolding beta-min P'-same bp-view-def by simp
    qed
    have keys-nonempty-P: keys-of (bp-view P) ≠ {}
      using keys-nonempty unfolding P'-same .
    have lower-P:
      ⋀v. v ∈ keys-of (bp-view P) ⟹ beta ≤ value-of (bp-view P) v
      using lower unfolding P'-same .
    show ?thesis
      unfolding pull-separates-def P'-same S-empty
      using lower-P keys-nonempty-P by auto
  qed
qed

lemma bp-pull-invariant:
  assumes inv: bp-invariant P
    and pull: bp-pull M B P = (S, beta, P')
  shows bp-invariant P'
```

```
proof (cases bp-can-first-bucket-pull M P)
  case True
  have first-pull: bp-first-bucket-pull M B P = (S, beta, P')
    using pull True unfolding bp-pull-def by simp
  show ?thesis
    by (rule bp-first-bucket-pull-invariant[OF inv first-pull])
next
  case False
  have conservative-pull: bp-conservative-pull M B P = (S, beta, P')
    using pull False unfolding bp-pull-def by simp
  show ?thesis
    by (rule bp-conservative-pull-invariant[OF inv conservative-pull])
qed

lemma bp-first-bucket-pull-ordered-invariant:
  assumes ord: bp-ordered-invariant P
    and pull: bp-first-bucket-pull M B P = (S, beta, P')
  shows bp-ordered-invariant P'
proof −
  have cases: P' = P ∨ (∃ T. P' = bp-delete-keys T P)
    using pull unfolding bp-first-bucket-pull-def
    by (cases bp-buckets P; auto split: list.splits)
  then show ?thesis
  proof
    assume P' = P
    then show ?thesis
      using ord by simp
  next
    assume ∃ T. P' = bp-delete-keys T P
    then obtain T where P' = bp-delete-keys T P
      by blast
    then show ?thesis
      using bp-delete-keys-ordered-invariant[OF ord] by simp
  qed
qed

lemma bp-conservative-pull-ordered-invariant:
  assumes ord: bp-ordered-invariant P
    and pull: bp-conservative-pull M B P = (S, beta, P')
  shows bp-ordered-invariant P'
proof −
  have P' = bp-delete-keys S P
    using pull unfolding bp-conservative-pull-def by (auto simp: Let-def)
  then show ?thesis
    using bp-delete-keys-ordered-invariant[OF ord] by simp
qed

lemma bp-pull-ordered-invariant:
  assumes ord: bp-ordered-invariant P
```

```
    and pull: bp-pull M B P = (S, beta, P')
  shows bp-ordered-invariant P'
proof (cases bp-can-first-bucket-pull M P)
  case True
  have first-pull: bp-first-bucket-pull M B P = (S, beta, P')
    using pull True unfolding bp-pull-def by simp
  show ?thesis
    by (rule bp-first-bucket-pull-ordered-invariant[OF ord first-pull])
next
  case False
  have conservative-pull: bp-conservative-pull M B P = (S, beta, P')
    using pull False unfolding bp-pull-def by simp
  show ?thesis
    by (rule bp-conservative-pull-ordered-invariant
        [OF ord conservative-pull])
qed

theorem bp-pull-refines-pull-separates:
  assumes inv: bp-invariant P
    and upper: ⋀u. u ∈ keys-of (bp-view P) ⟹
      value-of (bp-view P) u < B
    and pull: bp-pull M B P = (S, beta, P')
  shows pull-separates (bp-view P) M B S beta (bp-view P')
proof (cases bp-can-first-bucket-pull M P)
  case True
  obtain b c bs where buckets: bp-buckets P = b # c # bs
    and len-b: length (bp-bucket-entries b) ≤ M
    and below: bp-bucket-below-bound b (bp-marker c)
    and tail-nonempty: bp-bucket-entries-flat (c # bs) ≠ []
    by (rule bp-can-first-bucket-pullE[OF True])
  have first-pull: bp-first-bucket-pull M B P = (S, beta, P')
    using pull True unfolding bp-pull-def by simp
  show ?thesis
    by (rule bp-first-bucket-pull-refines-pull-separates
        [OF inv buckets len-b below tail-nonempty first-pull])
next
  case False
  have conservative-pull: bp-conservative-pull M B P = (S, beta, P')
    using pull False unfolding bp-pull-def by simp
  show ?thesis
    by (rule bp-conservative-pull-refines-pull-separates
        [OF inv upper conservative-pull])
qed

theorem bp-pull-preserves-upper-bound:
  assumes inv: bp-invariant P
    and pull-upper: partition-upper-bound (bp-view P) Bpull
    and upper: partition-upper-bound (bp-view P) B
    and pull: bp-pull M Bpull P = (S, beta, P')
```

```
  shows partition-upper-bound (bp-view P') B
proof −
  have sep: pull-separates (bp-view P) M Bpull S beta (bp-view P')
  proof (rule bp-pull-refines-pull-separates[OF inv - pull])
    fix u
    assume u ∈ keys-of (bp-view P)
    then show value-of (bp-view P) u < Bpull
      using pull-upper unfolding partition-upper-bound-def by blast
  qed
  show ?thesis
    by (rule pull-separates-preserves-upper-bound[OF sep upper])
qed
```

The preceding block completes the functional refinement layer. Insert and BatchPrepend update the view by minimum-value update, while Pull satisfies *pull-separates*: it returns at most the requested block, removes those keys, and establishes the returned boundary between pulled and remaining entries. The implementation attempts a cheap first-bucket pull when the ordered boundary proves it is safe; otherwise it uses the conservative pull specification. Both branches preserve *bp-invariant* and the abstract upper-bound condition.

14 Cost Budgets for the Bucketed Structure

The cost layer counts primitive functional steps by pairing each operation result with a natural number. The budget definitions isolate the paper-critical logarithmic search term as a function of N / M, where N is represented by an entry-list length and M by the block size. The potential terms record the credits that pay for lazy bucket splitting and for restoring the bucket-count ratio after a rebuild.

```
definition bp-steps-of :: 'a × nat ⇒ nat where
  bp-steps-of r = snd r

definition bp-result-of :: 'a × nat ⇒ 'a where
  bp-result-of r = fst r

lemma bp-steps-of-pair [simp]:
  bp-steps-of (x, c) = c
  unfolding bp-steps-of-def by simp

lemma bp-result-of-pair [simp]:
  bp-result-of (x, c) = x
  unfolding bp-result-of-def by simp

lemma bp-result-of-add-cost-case [simp]:
  bp-result-of (case r of (x, c') ⇒ (x, c + c')) = bp-result-of r
  by (cases r) simp
```

lemma *bp-steps-of-add-cost-case* [*simp*]:
bp-steps-of (*case r of* (*x*, *c'*) ⇒ (*x*, *c* + *c'*)) =
c + *bp-steps-of r*
by (*cases r*) *simp*

definition *bp-ratio-log-budget* :: *nat* ⇒ *nat* ⇒ *nat* **where**
bp-ratio-log-budget N M = *Suc* (*floor-log* (*Suc* (*N div M*)))

definition *bp-insert-search-budget* :: *nat* ⇒ *nat* ⇒ *nat* **where**
bp-insert-search-budget N M = *Suc* (*bp-ratio-log-budget N M*)

definition *bp-lazy-insert-amortized-budget* ::
'k bucketed-partition ⇒ *nat* **where**
bp-lazy-insert-amortized-budget P =
bp-insert-search-budget (*length* (*bp-entries P*)) (*bp-block-size P*) + 7

definition *bp-batch-prepend-log-budget* :: (*'k* × *real*) *list* ⇒ *nat* ⇒ *nat* **where**
bp-batch-prepend-log-budget xs M =
length xs ∗ *bp-ratio-log-budget* (*length xs*) *M*

definition *bp-batch-prepend-per-item-budget* ::
(*'k* × *real*) *list* ⇒ *nat* ⇒ *nat* **where**
bp-batch-prepend-per-item-budget xs M =
Suc (*bp-ratio-log-budget* (*length xs*) *M*)

definition *bp-batch-prepend-amortized-budget* ::
(*'k* × *real*) *list* ⇒ *nat* ⇒ *nat* **where**
bp-batch-prepend-amortized-budget xs M =
length xs + *bp-batch-prepend-log-budget xs M*

definition *bp-pull-amortized-budget* :: *nat* ⇒ *nat* **where**
bp-pull-amortized-budget M = 2 ∗ *M*

definition *bp-local-split-budget* :: *nat* ⇒ *nat* **where**
bp-local-split-budget M = *Suc* (2 ∗ *Suc M*)

definition *bp-lazy-split-budget* :: *nat* ⇒ *nat* **where**
bp-lazy-split-budget M = 5 + 4 ∗ *M*

definition *bp-bucket-overflow* :: *nat* ⇒ *'k bp-bucket* ⇒ *nat* **where**
bp-bucket-overflow M b = *length* (*bp-bucket-entries b*) − *M*

definition *bp-bucket-overflow-sum* ::
nat ⇒ *'k bp-bucket list* ⇒ *nat* **where**
bp-bucket-overflow-sum M bs =
sum-list (*map* (*bp-bucket-overflow M*) *bs*)

definition *bp-overflow-potential* :: *'k bucketed-partition* ⇒ *nat* **where**
bp-overflow-potential P =

```
    sum-list (map (bp-bucket-overflow (bp-block-size P)) (bp-buckets P))

definition bp-bucket-count-slack :: 'k bucketed-partition ⇒ nat where
  bp-bucket-count-slack P =
    length (bp-buckets P) −
      Suc (length (bp-entries P) div bp-block-size P)

definition bp-split-potential :: 'k bucketed-partition ⇒ nat where
  bp-split-potential P =
    4 ∗ bp-overflow-potential P + bp-bucket-count-slack P

definition bp-amortized-step-bound ::
  nat ⇒ 'k bucketed-partition ⇒ 'k bucketed-partition ⇒ nat ⇒ bool where
  bp-amortized-step-bound c P P' t ⟷
    c + bp-split-potential P' ≤ t + bp-split-potential P

definition bp-pull-state-of ::
  'k set × real × 'k bucketed-partition ⇒ 'k bucketed-partition where
  bp-pull-state-of R = (case R of (-, -, P') ⇒ P')

definition bp-pull-amortized-step-bound ::
  nat ⇒ 'k bucketed-partition ⇒
    ('k set × real × 'k bucketed-partition) ⇒ nat ⇒ bool where
  bp-pull-amortized-step-bound c P R t ⟷
    c + bp-split-potential (bp-pull-state-of R) ≤
      t + bp-split-potential P

definition bp-bucket-count-ratio-ok :: 'k bucketed-partition ⇒ bool where
  bp-bucket-count-ratio-ok P ⟷
    0 < bp-block-size P ∧
    length (bp-buckets P) ≤
      Suc (length (bp-entries P) div bp-block-size P)

definition bp-regular-invariant :: 'k bucketed-partition ⇒ bool where
  bp-regular-invariant P ⟷
    bp-ordered-invariant P ∧ bp-bucket-count-ratio-ok P

definition bp-bucket-count-loaded :: 'k bucketed-partition ⇒ bool where
  bp-bucket-count-loaded P ⟷
    bp-block-size P ∗ (length (bp-buckets P) − 1) ≤
      length (bp-entries P)

definition bp-bucket-search-steps :: 'k bucketed-partition ⇒ nat where
  bp-bucket-search-steps P = Suc (floor-log (length (bp-buckets P)))

fun bp-halving-search-steps :: nat ⇒ nat where
  [simp del]: bp-halving-search-steps n =
    (if n < 2 then 1 else Suc (bp-halving-search-steps (n div 2)))
```

```
definition bp-local-insert-search-charge :: 'k bucketed-partition ⇒ nat where
  bp-local-insert-search-charge P = Suc (bp-bucket-search-steps P)

definition c-bp-bucket-directory-search ::
  'k bp-bucket list ⇒ real ⇒ unit × nat where
  c-bp-bucket-directory-search bs b =
    ((), bp-halving-search-steps (length bs))

fun c-bp-bucketize-sorted-entries-aux ::
  nat ⇒ nat ⇒ ('k × real) list ⇒ 'k bp-bucket list × nat where
  c-bp-bucketize-sorted-entries-aux 0 M xs = ([], 0)
| c-bp-bucketize-sorted-entries-aux (Suc fuel) M xs =
    (if M = 0 ∨ xs = []
     then ([], 0)
     else (let (bs, c) =
                 c-bp-bucketize-sorted-entries-aux fuel M (drop M xs)
           in (bp-make-bucket (take M xs) # bs,
               Suc (length (take M xs) + c))))

definition c-bp-bucketize-sorted-entries ::
  nat ⇒ ('k × real) list ⇒ 'k bp-bucket list × nat where
  c-bp-bucketize-sorted-entries M xs =
    c-bp-bucketize-sorted-entries-aux (length xs) M xs

definition c-bp-bucketize-entries ::
  nat ⇒ ('k × real) list ⇒ 'k bp-bucket list × nat where
  c-bp-bucketize-entries M xs =
    c-bp-bucketize-sorted-entries M (sort-key snd xs)

definition c-bp-rebucket-build ::
  'k bucketed-partition ⇒ 'k bucketed-partition × nat where
  c-bp-rebucket-build P =
    (case c-bp-bucketize-entries (bp-block-size P) (bp-entries P) of
      (bs, c) ⇒ (P(|bp-buckets := bs|), c))
```

The definitions above introduce the amortized accounting used throughout the cost proof. *bp-ratio-log-budget* is the formal ratio-log term: it applies *floor-log* to *Suc* (*N div M*), avoiding division by zero corner cases at the statement level. The actual bucket-directory search is modeled by *bp-halving-search-steps*, and *bp-bucket-search-steps* records the resulting cost for a state.

The potential *bp-split-potential* has two components. The weighted overflow component *bp-overflow-potential* counts entries above the strict block size, multiplied by four in the combined potential. The *bp-bucket-count-slack* component measures how far the bucket count is above the target ratio. A regular state, captured by *bp-regular-invariant*, has no such debt: it is ordered and satisfies *bp-bucket-count-ratio-ok*. The amortized predicates *bp-amortized-step-bound* and *bp-pull-amortized-step-bound* express the usual inequality *actual* + *Phi*(*after*) <= *budget* + *Phi*(*before*).

definition *c-bp-local-insert-search* ::
  'k ⇒ real ⇒ 'k bucketed-partition ⇒
    'k bucketed-partition × nat **where**
  c-bp-local-insert-search x b P =
    (bp-local-insert-state x b P, bp-local-insert-search-charge P)

definition *c-bp-local-insert-split* ::
  'k ⇒ real ⇒ 'k bucketed-partition ⇒
    'k bucketed-partition × nat **where**
  c-bp-local-insert-split x b P =
    (bp-local-insert-state x b P,
     bp-local-insert-search-charge P +
       bp-local-split-budget (bp-block-size P))

definition *c-bp-lazy-insert* ::
  'k ⇒ real ⇒ 'k bucketed-partition ⇒
    'k bucketed-partition × nat **where**
  c-bp-lazy-insert x b P =
    (bp-lazy-insert-state x b P,
     bp-local-insert-search-charge P +
       bp-lazy-split-budget (bp-block-size P))

definition *bp-lazy-bucket-insert-charge* ::
  nat ⇒ 'k bp-bucket ⇒ nat **where**
  bp-lazy-bucket-insert-charge M b =
    *(if length (bp-bucket-entries b) < 2 * M*
     then 1 else bp-lazy-split-budget M)

fun *bp-lazy-insert-bucket-charge* ::
  nat ⇒ 'k × real ⇒ 'k bp-bucket list ⇒ nat **where**
  bp-lazy-insert-bucket-charge M p [] = 1
| *bp-lazy-insert-bucket-charge M p [b] =*
    (if snd p < bp-marker b then 1
     else bp-lazy-bucket-insert-charge M b)
| *bp-lazy-insert-bucket-charge M p (b # c # bs) =*
    (if snd p < bp-marker b then 1
     else if snd p < bp-marker c
       then bp-lazy-bucket-insert-charge M b
       else bp-lazy-insert-bucket-charge M p (c # bs))

definition *bp-lazy-insert-charge* ::
  'k ⇒ real ⇒ 'k bucketed-partition ⇒ nat **where**
  bp-lazy-insert-charge x b P =
    (let b' = (if x ∈ bp-entry-keys (bp-entries P)
         then min (bp-values P x) b else b);
       P0 = bp-delete-key x P
     in bp-lazy-insert-bucket-charge (bp-block-size P0) (x, b')
         (bp-buckets P0))

```
definition c-bp-lazy-insert-precise ::
  'k ⇒ real ⇒ 'k bucketed-partition ⇒
    'k bucketed-partition × nat where
  c-bp-lazy-insert-precise x b P =
     (bp-lazy-insert-state x b P,
      bp-local-insert-search-charge P + bp-lazy-insert-charge x b P)

fun bp-lazy-batch-prepend-state ::
  ('k × real) list ⇒ 'k bucketed-partition ⇒
    'k bucketed-partition where
  bp-lazy-batch-prepend-state [] P = P
| bp-lazy-batch-prepend-state ((x, b) # xs) P =
     bp-lazy-batch-prepend-state xs (bp-lazy-insert-state x b P)

fun c-bp-lazy-batch-prepend-precise ::
  ('k × real) list ⇒ 'k bucketed-partition ⇒
    'k bucketed-partition × nat where
  c-bp-lazy-batch-prepend-precise [] P = (P, 0)
| c-bp-lazy-batch-prepend-precise ((x, b) # xs) P =
     (case c-bp-lazy-insert-precise x b P of
        (P1, c1) ⇒
          (case c-bp-lazy-batch-prepend-precise xs P1 of
            (P2, c2) ⇒ (P2, c1 + c2)))

fun bp-lazy-batch-ratio-ok ::
  ('k × real) list ⇒ 'k bucketed-partition ⇒ bool where
  bp-lazy-batch-ratio-ok [] P ⟷ True
| bp-lazy-batch-ratio-ok ((x, b) # xs) P ⟷
     bp-bucket-count-ratio-ok P ∧
     bp-lazy-batch-ratio-ok xs
       (bp-result-of (c-bp-lazy-insert-precise x b P))

fun bp-lazy-batch-insert-budget-sum ::
  ('k × real) list ⇒ 'k bucketed-partition ⇒ nat where
  bp-lazy-batch-insert-budget-sum [] P = 0
| bp-lazy-batch-insert-budget-sum ((x, b) # xs) P =
     bp-lazy-insert-amortized-budget P +
     bp-lazy-batch-insert-budget-sum xs
       (bp-result-of (c-bp-lazy-insert-precise x b P))

lemma c-bp-lazy-insert-precise-eq [simp]:
  c-bp-lazy-insert-precise x b P =
    (bp-lazy-insert-state x b P,
     bp-local-insert-search-charge P + bp-lazy-insert-charge x b P)
  unfolding c-bp-lazy-insert-precise-def by simp

definition bp-regularized-local-insert ::
  'k ⇒ real ⇒ 'k bucketed-partition ⇒ 'k bucketed-partition where
  bp-regularized-local-insert x b P =
```

```
    bp-rebucket (bp-local-insert-state x b P)

definition c-bp-regularized-local-insert ::
  'k ⇒ real ⇒ 'k bucketed-partition ⇒
    'k bucketed-partition × nat where
  c-bp-regularized-local-insert x b P =
    (bp-regularized-local-insert x b P, bp-local-insert-search-charge P)
```

This wrapper counts the concrete rebuilding pass explicitly. It is an accounting bridge for the later lazy-split analysis, not the final insert operation bound: rebuilding every bucket has a linear term.

```
definition c-bp-regularized-local-insert-build ::
  'k ⇒ real ⇒ 'k bucketed-partition ⇒
    'k bucketed-partition × nat where
  c-bp-regularized-local-insert-build x b P =
    (case c-bp-rebucket-build (bp-local-insert-state x b P) of
      (P', c) ⇒ (P', bp-local-insert-search-charge P + c))

definition c-bp-rebucketed-batch-prepend-bulk ::
  ('k × real) list ⇒ 'k bucketed-partition ⇒
    'k bucketed-partition × nat where
  c-bp-rebucketed-batch-prepend-bulk xs P =
    (bp-rebucketed-batch-prepend xs P,
     bp-batch-prepend-log-budget xs (bp-block-size P))

definition c-bp-bucketed-batch-prepend-direct ::
  ('k × real) list ⇒ 'k bucketed-partition ⇒
    'k bucketed-partition × nat where
  c-bp-bucketed-batch-prepend-direct xs P =
    (bp-bucketed-batch-prepend-state xs P,
     bp-batch-prepend-amortized-budget xs (bp-block-size P))

definition c-bp-bucketed-batch-prepend-direct-actual ::
  ('k × real) list ⇒ 'k bucketed-partition ⇒
    'k bucketed-partition × nat where
  c-bp-bucketed-batch-prepend-direct-actual xs P =
    (bp-bucketed-batch-prepend-state xs P,
     bp-batch-prepend-log-budget xs (bp-block-size P))

definition c-bp-first-bucket-pull ::
  nat ⇒ real ⇒ 'k bucketed-partition ⇒
    ('k set × real × 'k bucketed-partition) × nat where
  c-bp-first-bucket-pull M B P = (bp-first-bucket-pull M B P, M)

definition c-bp-first-bucket-pull-scan ::
  nat ⇒ real ⇒ 'k bucketed-partition ⇒
    ('k set × real × 'k bucketed-partition) × nat where
  c-bp-first-bucket-pull-scan M B P =
    (case bp-first-bucket-pull M B P of
```

$(S,\ beta,\ P') \Rightarrow ((S,\ beta,\ P'),\ card\ S))$

lemma *bp-ratio-log-budget-pos* [*simp*]:
0 < bp-ratio-log-budget N M
unfolding *bp-ratio-log-budget-def* **by** *simp*

lemma *bp-insert-search-budget-pos* [*simp*]:
0 < bp-insert-search-budget N M
unfolding *bp-insert-search-budget-def* **by** *simp*

lemma *bp-lazy-insert-amortized-budget-ratio*:
bp-lazy-insert-amortized-budget P =
bp-ratio-log-budget (length (bp-entries P)) (bp-block-size P) + 8
unfolding *bp-lazy-insert-amortized-budget-def bp-insert-search-budget-def*
by *simp*

lemma *bp-batch-prepend-amortized-budget-alt*:
bp-batch-prepend-amortized-budget xs M =
*bp-batch-prepend-per-item-budget xs M * length xs*
unfolding *bp-batch-prepend-amortized-budget-def*
bp-batch-prepend-log-budget-def
bp-batch-prepend-per-item-budget-def
by (*simp add*: *algebra-simps*)

lemma *bp-batch-prepend-log-budget-le-amortized-budget*:
bp-batch-prepend-log-budget xs M ≤
bp-batch-prepend-amortized-budget xs M
unfolding *bp-batch-prepend-amortized-budget-def* **by** *simp*

lemma *bp-halving-search-steps-eq-floor-log*:
bp-halving-search-steps n = Suc (floor-log n)
by (*induction n rule*: *bp-halving-search-steps.induct*)
(*simp add*: *bp-halving-search-steps.simps floor-log.simps*)

lemma *c-bp-bucket-directory-search-result* [*simp*]:
bp-result-of (c-bp-bucket-directory-search bs b) = ()
unfolding *bp-result-of-def c-bp-bucket-directory-search-def* **by** *simp*

lemma *c-bp-bucket-directory-search-steps* [*simp*]:
bp-steps-of (c-bp-bucket-directory-search bs b) =
bp-halving-search-steps (length bs)
unfolding *bp-steps-of-def c-bp-bucket-directory-search-def* **by** *simp*

lemma *c-bp-bucket-directory-search-steps-floor-log*:
bp-steps-of (c-bp-bucket-directory-search bs b) =
Suc (floor-log (length bs))
by (*simp add*: *bp-halving-search-steps-eq-floor-log*)

lemma *c-bp-bucket-directory-search-state-steps*:

```
  bp-steps-of (c-bp-bucket-directory-search (bp-buckets P) b) =
    bp-bucket-search-steps P
proof −
  have bp-steps-of (c-bp-bucket-directory-search (bp-buckets P) b) =
      Suc (floor-log (length (bp-buckets P)))
    by (simp add: bp-halving-search-steps-eq-floor-log)
  also have ... = bp-bucket-search-steps P
    unfolding bp-bucket-search-steps-def by simp
  finally show ?thesis .
qed

lemma c-bp-bucket-directory-search-ratio-bound:
  assumes count-le: length bs ≤ Suc (N div M)
  shows bp-steps-of (c-bp-bucket-directory-search bs b) ≤
    bp-ratio-log-budget N M
proof −
  have floor-log (length bs) ≤ floor-log (Suc (N div M))
    by (rule floor-log-le-iff[OF count-le])
  then show ?thesis
    unfolding bp-ratio-log-budget-def
    by (simp add: bp-halving-search-steps-eq-floor-log)
qed

lemma bp-local-insert-search-charge-search-steps:
  bp-local-insert-search-charge P =
    Suc (bp-steps-of
      (c-bp-bucket-directory-search (bp-buckets P) b))
proof −
  have search:
    bp-steps-of (c-bp-bucket-directory-search (bp-buckets P) b) =
      bp-bucket-search-steps P
    by (rule c-bp-bucket-directory-search-state-steps)
  show ?thesis
    unfolding bp-local-insert-search-charge-def using search by simp
qed

lemma c-bp-bucketize-sorted-entries-aux-empty [simp]:
  c-bp-bucketize-sorted-entries-aux fuel M [] = ([], 0)
  by (cases fuel) simp-all

lemma c-bp-bucketize-sorted-entries-aux-result [simp]:
  bp-result-of (c-bp-bucketize-sorted-entries-aux fuel M xs) =
    bp-bucketize-sorted-entries-aux fuel M xs
proof (induction fuel arbitrary: xs)
  case 0
  then show ?case
    by (simp add: bp-result-of-def)
next
  case (Suc fuel)
```

```
  show ?case
  proof (cases M = 0 ∨ xs = [])
    case True
    then show ?thesis
      by (simp add: bp-result-of-def)
  next
    case False
    obtain bs c where rec:
      c-bp-bucketize-sorted-entries-aux fuel M (drop M xs) = (bs, c)
      by (cases c-bp-bucketize-sorted-entries-aux fuel M (drop M xs))
        simp
    have bs:
      bs = bp-bucketize-sorted-entries-aux fuel M (drop M xs)
      using Suc.IH[of drop M xs] rec
      by (simp add: bp-result-of-def)
    show ?thesis
      using False rec bs by (simp add: bp-result-of-def Let-def)
  qed
qed

lemma c-bp-bucketize-sorted-entries-result [simp]:
  bp-result-of (c-bp-bucketize-sorted-entries M xs) =
    bp-bucketize-sorted-entries M xs
  unfolding c-bp-bucketize-sorted-entries-def
    bp-bucketize-sorted-entries-def by simp

lemma c-bp-bucketize-entries-result [simp]:
  bp-result-of (c-bp-bucketize-entries M xs) = bp-bucketize-entries M xs
  unfolding c-bp-bucketize-entries-def bp-bucketize-entries-def by simp

lemma c-bp-bucketize-sorted-entries-aux-steps-le:
  assumes M-pos: 0 < M
    and fuel: length xs ≤ fuel
  shows bp-steps-of (c-bp-bucketize-sorted-entries-aux fuel M xs) ≤
    length xs + Suc (length xs div M)
  using M-pos fuel
proof (induction fuel arbitrary: xs)
  case 0
  then show ?case
    by (simp add: bp-steps-of-def)
next
  case (Suc fuel)
  show ?case
  proof (cases xs = [])
    case True
    then show ?thesis
      by (simp add: bp-steps-of-def)
  next
    case False
```

note *xs-nonempty* = *False*
show *?thesis*
proof (*cases length xs* < *M*)
case *True*
then have *drop M xs* = []
by *simp*
then show *?thesis*
using *Suc.prems xs-nonempty True* **by** (*simp add*: *bp-steps-of-def*)
next
case *False*
have *M-le*: *M* ≤ *length xs*
using *False* **by** *simp*
have *drop-fuel*: *length* (*drop M xs*) ≤ *fuel*
using *Suc.prems False* **by** *simp*
have *tail*:
bp-steps-of
(*c-bp-bucketize-sorted-entries-aux fuel M* (*drop M xs*)) ≤
length (*drop M xs*) + *Suc* (*length* (*drop M xs*) *div M*)
by (*rule Suc.IH*[*OF Suc.prems*(*1*) *drop-fuel*])
have *len-xs*: *length xs* = *M* + *length* (*drop M xs*)
using *M-le* **by** *simp*
have *take-len*: *length* (*take M xs*) = *M*
using *M-le* **by** *simp*
have *div-eq*:
length xs div M = *Suc* (*length* (*drop M xs*) *div M*)
proof −
have *length xs div M* = (*M* + *length* (*drop M xs*)) *div M*
using *len-xs* **by** *simp*
also have ... = *length* (*drop M xs*) *div M* + *1*
using *Suc.prems*(*1*) **by** (*simp add*: *div-add-self1*)
finally show *?thesis* **by** *simp*
qed
obtain *bs c* **where** *rec*:
c-bp-bucketize-sorted-entries-aux fuel M (*drop M xs*) = (*bs*, *c*)
by (*cases c-bp-bucketize-sorted-entries-aux fuel M* (*drop M xs*))
simp
have *c-le*:
c ≤ *length* (*drop M xs*) + *Suc* (*length* (*drop M xs*) *div M*)
using *tail rec* **unfolding** *bp-steps-of-def* **by** *simp*
have *step-eq*:
bp-steps-of
(*c-bp-bucketize-sorted-entries-aux* (*Suc fuel*) *M xs*) =
Suc (*M* + *c*)
using *Suc.prems xs-nonempty False rec take-len*
by (*simp add*: *bp-steps-of-def Let-def*)
have *Suc* (*M* + *c*) ≤
M + *length* (*drop M xs*) +
Suc (*Suc* (*length* (*drop M xs*) *div M*))
using *c-le* **by** *linarith*

```
      show ?thesis
        using step-eq len-xs div-eq
          ‹Suc (M + c) ≤
            M + length (drop M xs) +
              Suc (Suc (length (drop M xs) div M))›
        by simp
    qed
  qed
qed

lemma c-bp-bucketize-sorted-entries-steps-le:
  assumes 0 < M
  shows bp-steps-of (c-bp-bucketize-sorted-entries M xs) ≤
    length xs + Suc (length xs div M)
  unfolding c-bp-bucketize-sorted-entries-def
  by (rule c-bp-bucketize-sorted-entries-aux-steps-le[OF assms]) simp

lemma c-bp-bucketize-entries-steps-le:
  assumes 0 < M
  shows bp-steps-of (c-bp-bucketize-entries M xs) ≤
    length xs + Suc (length xs div M)
  unfolding c-bp-bucketize-entries-def
  using c-bp-bucketize-sorted-entries-steps-le[OF assms,
      of sort-key snd xs]
  by simp

lemma c-bp-bucketize-entries-steps-le-linear-length:
  assumes 0 < M
  shows bp-steps-of (c-bp-bucketize-entries M xs) ≤
    Suc (2 * length xs)
proof −
  have length xs div M ≤ length xs
    by simp
  then have length xs + Suc (length xs div M) ≤
      Suc (2 * length xs)
    by linarith
  moreover have bp-steps-of (c-bp-bucketize-entries M xs) ≤
      length xs + Suc (length xs div M)
    by (rule c-bp-bucketize-entries-steps-le[OF assms])
  ultimately show ?thesis
    by linarith
qed

lemma c-bp-bucketize-entries-steps-le-local-split-budget:
  assumes M-pos: 0 < M
    and len: length xs ≤ Suc M
  shows bp-steps-of (c-bp-bucketize-entries M xs) ≤
    bp-local-split-budget M
proof −
```

```
  have steps:
    bp-steps-of (c-bp-bucketize-entries M xs) ≤
      Suc (2 * length xs)
    by (rule c-bp-bucketize-entries-steps-le-linear-length[OF M-pos])
  have Suc (2 * length xs) ≤ bp-local-split-budget M
    using len unfolding bp-local-split-budget-def by simp
  then show ?thesis
    using steps by linarith
qed

lemma bp-local-split-budget-le-lazy-split-budget:
  bp-local-split-budget M ≤ bp-lazy-split-budget M
  unfolding bp-local-split-budget-def bp-lazy-split-budget-def by simp

lemma c-bp-bucketize-entries-steps-le-lazy-split-budget:
  assumes M-pos: 0 < M
    and len: length xs ≤ Suc (2 * M)
  shows bp-steps-of (c-bp-bucketize-entries M xs) ≤
    bp-lazy-split-budget M
proof −
  have steps:
    bp-steps-of (c-bp-bucketize-entries M xs) ≤
      Suc (2 * length xs)
    by (rule c-bp-bucketize-entries-steps-le-linear-length[OF M-pos])
  have Suc (2 * length xs) ≤ bp-lazy-split-budget M
    using len unfolding bp-lazy-split-budget-def by simp
  then show ?thesis
    using steps by linarith
qed

lemma bp-lazy-split-budget-paid-by-bucket-overflow:
  assumes large: 2 * M ≤ length (bp-bucket-entries b)
  shows bp-lazy-split-budget M ≤
    5 + 4 * bp-bucket-overflow M b
proof −
  have M ≤ bp-bucket-overflow M b
    using large unfolding bp-bucket-overflow-def by linarith
  then show ?thesis
    unfolding bp-lazy-split-budget-def by simp
qed

lemma bp-bucket-overflow-sum-simps [simp]:
  bp-bucket-overflow-sum M [] = 0
  bp-bucket-overflow-sum M (b # bs) =
    bp-bucket-overflow M b + bp-bucket-overflow-sum M bs
  bp-bucket-overflow-sum M (bs @ cs) =
    bp-bucket-overflow-sum M bs + bp-bucket-overflow-sum M cs
  unfolding bp-bucket-overflow-sum-def by simp-all
```

```
lemma bp-overflow-potential-alt:
  bp-overflow-potential P =
    bp-bucket-overflow-sum (bp-block-size P) (bp-buckets P)
  unfolding bp-overflow-potential-def bp-bucket-overflow-sum-def by simp

lemma bp-bucket-overflow-sum-zero-if-sizes-ok:
  assumes sizes: ∀ b∈set bs. length (bp-bucket-entries b) ≤ M
  shows bp-bucket-overflow-sum M bs = 0
  using sizes unfolding bp-bucket-overflow-sum-def bp-bucket-overflow-def
  by (induction bs) auto

lemma bp-bucket-overflow-cons-credit:
  fixes b :: 'k bp-bucket
  shows 1 + 4 * bp-bucket-overflow M
      (b(|bp-bucket-entries := p # bp-bucket-entries b|)) ≤
    5 + 4 * bp-bucket-overflow M b
proof −
  let ?l = length (bp-bucket-entries b)
  show ?thesis
  proof (cases ?l < M)
    case True
    then have Suc ?l ≤ M
      by simp
    then have after: bp-bucket-overflow M
        (b(|bp-bucket-entries := p # bp-bucket-entries b|)) = 0
      unfolding bp-bucket-overflow-def by simp
    have before: bp-bucket-overflow M b = 0
      using True unfolding bp-bucket-overflow-def by simp
    show ?thesis
      unfolding after before by simp
  next
    case False
    then have M-le: M ≤ ?l
      by simp
    have after: bp-bucket-overflow M
        (b(|bp-bucket-entries := p # bp-bucket-entries b|)) =
      Suc (bp-bucket-overflow M b)
      using M-le unfolding bp-bucket-overflow-def by simp
    show ?thesis
      unfolding after by simp
  qed
qed

lemma bp-lazy-bucket-insert-entries-weighted-overflow-credit:
  assumes M-pos: 0 < M
    and size: length (bp-bucket-entries b) ≤ 2 * M
  shows bp-lazy-bucket-insert-charge M b +
      4 * bp-bucket-overflow-sum M
        (bp-lazy-bucket-insert-entries M p b) ≤
```

```
    5 + 4 * bp-bucket-overflow M b
proof (cases length (bp-bucket-entries b) < 2 * M)
  case True
  have charge: bp-lazy-bucket-insert-charge M b = 1
    using True unfolding bp-lazy-bucket-insert-charge-def by simp
  have result:
    bp-lazy-bucket-insert-entries M p b =
      [b(|bp-bucket-entries := p # bp-bucket-entries b|)]
    using True unfolding bp-lazy-bucket-insert-entries-def by simp
  have credit:
    1 + 4 * bp-bucket-overflow M
        (b(|bp-bucket-entries := p # bp-bucket-entries b|)) ≤
      5 + 4 * bp-bucket-overflow M b
    by (rule bp-bucket-overflow-cons-credit)
  show ?thesis
    using credit unfolding charge result by simp
next
  case False
  have len-eq: length (bp-bucket-entries b) = 2 * M
    using False size by simp
  have charge: bp-lazy-bucket-insert-charge M b = bp-lazy-split-budget M
    using False unfolding bp-lazy-bucket-insert-charge-def by simp
  have result:
    bp-lazy-bucket-insert-entries M p b =
      bp-bucketize-entries M (p # bp-bucket-entries b)
    using False unfolding bp-lazy-bucket-insert-entries-def by simp
  have sizes:
    ∀ c∈set (bp-bucketize-entries M (p # bp-bucket-entries b)).
      length (bp-bucket-entries c) ≤ M
    by (rule bp-bucketize-entries-sizes-ok[OF M-pos])
  have overflow-zero:
    bp-bucket-overflow-sum M
      (bp-bucketize-entries M (p # bp-bucket-entries b)) = 0
    by (rule bp-bucket-overflow-sum-zero-if-sizes-ok[OF sizes])
  have before: bp-bucket-overflow M b = M
    using len-eq unfolding bp-bucket-overflow-def by simp
  show ?thesis
    unfolding charge result overflow-zero before bp-lazy-split-budget-def
    by simp
qed

lemma bp-lazy-insert-bucket-weighted-overflow-credit:
  assumes M-pos: 0 < M
    and sizes: ∀ b∈set bs. length (bp-bucket-entries b) ≤ 2 * M
  shows bp-lazy-insert-bucket-charge M p bs +
      4 * bp-bucket-overflow-sum M (bp-lazy-insert-bucket M p bs) ≤
    5 + 4 * bp-bucket-overflow-sum M bs
  using sizes
proof (induction bs arbitrary: p)
```

```
  case Nil
  have result: bp-lazy-insert-bucket M p [] = bp-bucketize-entries M [p]
    by simp
  have sizes-result:
    ∀ c∈set (bp-bucketize-entries M [p]).
      length (bp-bucket-entries c) ≤ M
    by (rule bp-bucketize-entries-sizes-ok[OF M-pos])
  have overflow-zero:
    bp-bucket-overflow-sum M (bp-bucketize-entries M [p]) = 0
    by (rule bp-bucket-overflow-sum-zero-if-sizes-ok[OF sizes-result])
  show ?case
    unfolding result overflow-zero by simp
next
  case (Cons b bs)
  note IH = Cons.IH
  note sizes = Cons.prems
  have b-size: length (bp-bucket-entries b) ≤ 2 * M
    using sizes by simp
  show ?case
  proof (cases bs)
    case Nil
    show ?thesis
    proof (cases snd p < bp-marker b)
      case True
      have result:
        bp-lazy-insert-bucket M p (b # bs) =
          bp-bucketize-entries M [p] @ [b]
        using Nil True by simp
      have charge: bp-lazy-insert-bucket-charge M p (b # bs) = 1
        using Nil True by simp
      have sizes-single:
        ∀ c∈set (bp-bucketize-entries M [p]).
          length (bp-bucket-entries c) ≤ M
        by (rule bp-bucketize-entries-sizes-ok[OF M-pos])
      have overflow-single:
        bp-bucket-overflow-sum M (bp-bucketize-entries M [p]) = 0
        by (rule bp-bucket-overflow-sum-zero-if-sizes-ok[OF sizes-single])
      show ?thesis
        using Nil True overflow-single unfolding result charge by simp
    next
      case False
      have result:
        bp-lazy-insert-bucket M p (b # bs) =
          bp-lazy-bucket-insert-entries M p b
        using Nil False by simp
      have charge:
        bp-lazy-insert-bucket-charge M p (b # bs) =
          bp-lazy-bucket-insert-charge M b
        using Nil False by simp
```

```
    show ?thesis
      unfolding result charge using Nil
      by (simp add: bp-lazy-bucket-insert-entries-weighted-overflow-credit
          [OF M-pos b-size, of p])
  qed
next
  case (Cons c cs)
  have tail-sizes:
    ∀ b∈set bs. length (bp-bucket-entries b) ≤ 2 * M
    using sizes by simp
  show ?thesis
  proof (cases snd p < bp-marker b)
    case True
    have result:
      bp-lazy-insert-bucket M p (b # bs) =
        bp-bucketize-entries M [p] @ b # bs
      using Cons True by simp
    have charge: bp-lazy-insert-bucket-charge M p (b # bs) = 1
      using Cons True by simp
    have sizes-single:
      ∀ c∈set (bp-bucketize-entries M [p]).
        length (bp-bucket-entries c) ≤ M
      by (rule bp-bucketize-entries-sizes-ok[OF M-pos])
    have overflow-single:
      bp-bucket-overflow-sum M (bp-bucketize-entries M [p]) = 0
      by (rule bp-bucket-overflow-sum-zero-if-sizes-ok[OF sizes-single])
    show ?thesis
      using Cons True overflow-single unfolding result charge by simp
  next
    case False
    show ?thesis
    proof (cases snd p < bp-marker c)
      case True
      have result:
        bp-lazy-insert-bucket M p (b # bs) =
          bp-lazy-bucket-insert-entries M p b @ bs
        using Cons False True by simp
      have charge:
        bp-lazy-insert-bucket-charge M p (b # bs) =
          bp-lazy-bucket-insert-charge M b
        using Cons False True by simp
      have bucket-credit:
        bp-lazy-bucket-insert-charge M b +
            4 * bp-bucket-overflow-sum M
              (bp-lazy-bucket-insert-entries M p b) ≤
          5 + 4 * bp-bucket-overflow M b
        by (rule bp-lazy-bucket-insert-entries-weighted-overflow-credit
            [OF M-pos b-size])
      show ?thesis
```

```
      using bucket-credit unfolding result charge
      by simp
    next
     case False
     have result:
      bp-lazy-insert-bucket M p (b # bs) =
       b # bp-lazy-insert-bucket M p bs
      using Cons ‹¬ snd p < bp-marker b› False by simp
     have charge:
      bp-lazy-insert-bucket-charge M p (b # bs) =
       bp-lazy-insert-bucket-charge M p bs
      using Cons ‹¬ snd p < bp-marker b› False by simp
     have tail-credit:
      bp-lazy-insert-bucket-charge M p bs +
          4 * bp-bucket-overflow-sum M
           (bp-lazy-insert-bucket M p bs) ≤
       5 + 4 * bp-bucket-overflow-sum M bs
      by (rule IH[OF tail-sizes])
     show ?thesis
      using tail-credit unfolding result charge by simp
    qed
   qed
  qed
qed

lemma bp-delete-key-from-bucket-overflow-le:
  bp-bucket-overflow M (bp-delete-key-from-bucket x b) ≤
   bp-bucket-overflow M b
proof −
  have len-le:
   length (bp-bucket-entries (bp-delete-key-from-bucket x b)) ≤
    length (bp-bucket-entries b)
   unfolding bp-delete-key-from-bucket-def by simp
  show ?thesis
   using len-le unfolding bp-bucket-overflow-def by simp
qed

lemma bp-bucket-overflow-sum-map-delete-key-le:
  bp-bucket-overflow-sum M (map (bp-delete-key-from-bucket x) bs) ≤
   bp-bucket-overflow-sum M bs
proof (induction bs)
  case Nil
  then show ?case by simp
next
  case (Cons b bs)
  have head:
   bp-bucket-overflow M (bp-delete-key-from-bucket x b) ≤
    bp-bucket-overflow M b
   by (rule bp-delete-key-from-bucket-overflow-le)
```

```
  show ?case
    using head Cons.IH by simp
qed

lemma bp-delete-key-overflow-potential-le:
  bp-overflow-potential (bp-delete-key x P) ≤ bp-overflow-potential P
proof −
  have bp-bucket-overflow-sum (bp-block-size P)
      (map (bp-delete-key-from-bucket x) (bp-buckets P)) ≤
    bp-bucket-overflow-sum (bp-block-size P) (bp-buckets P)
    by (rule bp-bucket-overflow-sum-map-delete-key-le)
  then show ?thesis
    unfolding bp-delete-key-def bp-overflow-potential-alt by simp
qed

lemma bp-delete-keys-from-bucket-overflow-le:
  bp-bucket-overflow M (bp-delete-keys-from-bucket S b) ≤
    bp-bucket-overflow M b
proof −
  have len-le:
    length (bp-bucket-entries (bp-delete-keys-from-bucket S b)) ≤
      length (bp-bucket-entries b)
    unfolding bp-delete-keys-from-bucket-def by simp
  show ?thesis
    using len-le unfolding bp-bucket-overflow-def by simp
qed

lemma bp-bucket-overflow-sum-map-delete-keys-le:
  bp-bucket-overflow-sum M (map (bp-delete-keys-from-bucket S) bs) ≤
    bp-bucket-overflow-sum M bs
proof (induction bs)
  case Nil
  then show ?case by simp
next
  case (Cons b bs)
  have head:
    bp-bucket-overflow M (bp-delete-keys-from-bucket S b) ≤
      bp-bucket-overflow M b
    by (rule bp-delete-keys-from-bucket-overflow-le)
  show ?case
    using head Cons.IH by simp
qed

lemma bp-delete-keys-overflow-potential-le:
  bp-overflow-potential (bp-delete-keys S P) ≤ bp-overflow-potential P
proof −
  have bp-bucket-overflow-sum (bp-block-size P)
      (map (bp-delete-keys-from-bucket S) (bp-buckets P)) ≤
    bp-bucket-overflow-sum (bp-block-size P) (bp-buckets P)
```

```
    by (rule bp-bucket-overflow-sum-map-delete-keys-le)
  then show ?thesis
    unfolding bp-delete-keys-def bp-overflow-potential-alt by simp
qed

lemma length-filter-fst-notin-le-card:
  assumes distinct: distinct (map fst xs)
    and finite: finite S
  shows length xs ≤ length (filter (λp. fst p ∉ S) xs) + card S
proof −
  let ?removed = filter (λp. fst p ∈ S) xs
  have len-part:
    length xs = length (filter (λp. fst p ∉ S) xs) + length ?removed
    by (induction xs) auto
  have distinct-removed: distinct (map fst ?removed)
    using distinct by (induction xs) auto
  have length ?removed = card (set (map fst ?removed))
    using distinct-card[OF distinct-removed] by simp
  also have ... ≤ card S
    using finite by (intro card-mono) auto
  finally have length ?removed ≤ card S .
  then show ?thesis
    using len-part by linarith
qed

lemma div-add-right-le-div-plus:
  fixes n k M :: nat
  assumes M-pos: 0 < M
  shows (n + k) div M ≤ n div M + k
proof −
  have n-less: n < Suc (n div M) * M
  proof −
    have n div M < Suc (n div M)
      by simp
    moreover have
      n div M < Suc (n div M) ⟷ n < Suc (n div M) * M
      by (rule div-less-iff-less-mult[OF M-pos])
    then show ?thesis
      using calculation by simp
  qed
  have k-le: k ≤ k * M
    using M-pos by (cases M) auto
  have n + k < Suc (n div M) * M + k * M
    using n-less k-le by linarith
  also have ... = (Suc (n div M) + k) * M
    by (simp add: algebra-simps)
  finally have (n + k) div M < Suc (n div M) + k
    using M-pos by (simp add: div-less-iff-less-mult)
  then show ?thesis by simp
```

qed

lemma *div-bound-after-delete*:
assumes *M-pos*: $0 < M$
and *len*: $n \leq n' + k$
shows $Suc\ (n\ div\ M) \leq Suc\ (n'\ div\ M) + k$
proof −
have $n\ div\ M \leq (n' + k)\ div\ M$
by (*rule div-le-mono*[*OF len*])
also have $\ldots \leq n'\ div\ M + k$
by (*rule div-add-right-le-div-plus*[*OF M-pos*])
finally show *?thesis* **by** *simp*
qed

lemma *nat-diff-le-diff-add-right*:
fixes $a\ c\ d\ k :: nat$
assumes $c \leq d + k$
shows $a - d \leq (a - c) + k$
proof −
have *base*: $a - d \leq (a - c) + (c - d)$
by *simp*
have $c - d \leq k$
using *assms* **by** *simp*
then show *?thesis*
using *base* **by** *linarith*
qed

lemma *nat-diff-add-right-le*:
fixes $a\ b\ c\ d\ k :: nat$
assumes *a-le*: $a \leq k + b$
and *d-le-c*: $d \leq c$
shows $a - c \leq k + (b - d)$
proof −
have $a - c \leq (k + b) - c$
using *a-le* **by** *simp*
also have $\ldots \leq k + (b - c)$
by *arith*
also have $\ldots \leq k + (b - d)$
using *d-le-c* **by** *simp*
finally show *?thesis* **.**
qed

lemma *bp-delete-keys-entries-length-le*:
assumes *distinct*: *bp-distinct-keys P*
and *finite*: *finite S*
shows *length* (*bp-entries P*) ≤
length (*bp-entries* (*bp-delete-keys S P*)) + *card S*
using *length-filter-fst-notin-le-card*[*OF - finite*, *of bp-entries P*]
distinct

unfolding *bp-entries-delete-keys bp-distinct-keys-def* **by** *simp*

lemma *bp-delete-keys-bucket-count-slack-le*:
assumes *M-pos*: *0 < bp-block-size P*
and *distinct*: *bp-distinct-keys P*
and *finite*: *finite S*
shows *bp-bucket-count-slack* (*bp-delete-keys S P*) ≤
bp-bucket-count-slack P + *card S*
proof −
let *?M* = *bp-block-size P*
let *?N* = *length* (*bp-entries P*)
let *?N′* = *length* (*bp-entries* (*bp-delete-keys S P*))
let *?K* = *length* (*bp-buckets P*)
have *len*: *?N* ≤ *?N′* + *card S*
by (*rule bp-delete-keys-entries-length-le*[*OF distinct finite*])
have *divs*: *Suc* (*?N div ?M*) ≤ *Suc* (*?N′ div ?M*) + *card S*
by (*rule div-bound-after-delete*[*OF M-pos len*])
have *slack*:
?K − *Suc* (*?N′ div ?M*) ≤ (*?K* − *Suc* (*?N div ?M*)) + *card S*
by (*rule nat-diff-le-diff-add-right*[*OF divs*])
show *?thesis*
using *slack* **unfolding** *bp-bucket-count-slack-def bp-delete-keys-def* **by** *simp*
qed

lemma *bp-delete-keys-split-potential-le*:
assumes *M-pos*: *0 < bp-block-size P*
and *distinct*: *bp-distinct-keys P*
and *finite*: *finite S*
shows *bp-split-potential* (*bp-delete-keys S P*) ≤
bp-split-potential P + *card S*
proof −
have *overflow*:
bp-overflow-potential (*bp-delete-keys S P*) ≤ *bp-overflow-potential P*
by (*rule bp-delete-keys-overflow-potential-le*)
have *slack*:
bp-bucket-count-slack (*bp-delete-keys S P*) ≤
bp-bucket-count-slack P + *card S*
by (*rule bp-delete-keys-bucket-count-slack-le*[*OF M-pos distinct finite*])
show *?thesis*
using *overflow slack* **unfolding** *bp-split-potential-def* **by** *linarith*
qed

lemma *bp-lazy-insert-state-weighted-overflow-credit-after-delete*:
assumes *lazy*: *bp-lazy-ordered-invariant P*
shows *bp-lazy-insert-charge x b P* +
4 ∗ *bp-overflow-potential* (*bp-lazy-insert-state x b P*) ≤
5 + *4* ∗ *bp-overflow-potential* (*bp-delete-key x P*)
proof −
let *?old-keys* = *bp-entry-keys* (*bp-entries P*)

```
  let ?b = if x ∈ ?old-keys then min (bp-values P x) b else b
  let ?P0 = bp-delete-key x P
  let ?M = bp-block-size ?P0
  let ?bs = bp-buckets ?P0
  let ?new-bs = bp-lazy-insert-bucket ?M (x, ?b) ?bs
  let ?P' = ?P0(|bp-buckets := ?new-bs,
      bp-values := (bp-values P)(x := ?b)|)
  have inv: bp-lazy-invariant P
    by (rule bp-lazy-ordered-invariant-lazy-invariant[OF lazy])
  have inv0: bp-lazy-invariant ?P0
    by (rule bp-delete-key-lazy-invariant[OF inv])
  have M-pos: 0 < ?M
    using inv0 unfolding bp-lazy-invariant-def by blast
  have sizes0:
    ∀ c∈set ?bs. length (bp-bucket-entries c) ≤ 2 * ?M
    using inv0 unfolding bp-lazy-invariant-def
      bp-lazy-bucket-sizes-ok-def by blast
  have list-credit:
    bp-lazy-insert-bucket-charge ?M (x, ?b) ?bs +
        4 * bp-bucket-overflow-sum ?M ?new-bs ≤
      5 + 4 * bp-bucket-overflow-sum ?M ?bs
    by (rule bp-lazy-insert-bucket-weighted-overflow-credit
        [OF M-pos sizes0])
  have state: bp-lazy-insert-state x b P = ?P'
    unfolding bp-lazy-insert-state-def Let-def by simp
  have charge:
    bp-lazy-insert-charge x b P =
      bp-lazy-insert-bucket-charge ?M (x, ?b) ?bs
    unfolding bp-lazy-insert-charge-def Let-def by simp
  show ?thesis
    using list-credit
    unfolding state charge bp-overflow-potential-alt by simp
qed

lemma bp-lazy-insert-state-weighted-overflow-credit:
  assumes lazy: bp-lazy-ordered-invariant P
  shows bp-lazy-insert-charge x b P +
      4 * bp-overflow-potential (bp-lazy-insert-state x b P) ≤
    5 + 4 * bp-overflow-potential P
proof −
  have after-delete:
    bp-lazy-insert-charge x b P +
        4 * bp-overflow-potential (bp-lazy-insert-state x b P) ≤
      5 + 4 * bp-overflow-potential (bp-delete-key x P)
    by (rule bp-lazy-insert-state-weighted-overflow-credit-after-delete
        [OF lazy])
  have delete-le:
    bp-overflow-potential (bp-delete-key x P) ≤ bp-overflow-potential P
    by (rule bp-delete-key-overflow-potential-le)
```

```
  show ?thesis
    using after-delete delete-le by linarith
qed

lemma nat-diff-le-diff-add:
  fixes a b c d k :: nat
  assumes a-le: a ≤ b + k
    and c-le: c ≤ d
  shows a − d ≤ (b − c) + k
proof −
  have a − d ≤ a − c
    using c-le by simp
  also have ... ≤ (b + k) − c
    using a-le by simp
  also have ... ≤ (b − c) + k
    by simp
  finally show ?thesis .
qed

lemma bp-lazy-insert-state-bucket-count-slack-le:
  assumes lazy: bp-lazy-ordered-invariant P
  shows bp-bucket-count-slack (bp-lazy-insert-state x b P) ≤
    bp-bucket-count-slack P + 2
proof −
  let ?P′ = bp-lazy-insert-state x b P
  let ?M = bp-block-size P
  have inv: bp-lazy-invariant P
    by (rule bp-lazy-ordered-invariant-lazy-invariant[OF lazy])
  have M-pos: 0 < ?M
    using inv unfolding bp-lazy-invariant-def by blast
  have block: bp-block-size ?P′ = ?M
    unfolding bp-lazy-insert-state-def bp-delete-key-def
    by (simp add: Let-def split: if-splits)
  have buckets:
    length (bp-buckets ?P′) ≤ length (bp-buckets P) + 2
    by (rule length-bp-lazy-insert-state-buckets-le[OF lazy])
  have entries:
    length (bp-entries P) ≤ length (bp-entries ?P′)
    by (rule length-bp-lazy-insert-state-entries-ge[OF lazy])
  have divs:
    Suc (length (bp-entries P) div ?M) ≤
      Suc (length (bp-entries ?P′) div ?M)
    using entries by (simp add: div-le-mono)
  have slack:
    length (bp-buckets ?P′) −
        Suc (length (bp-entries ?P′) div ?M) ≤
      (length (bp-buckets P) −
        Suc (length (bp-entries P) div ?M)) + 2
    by (rule nat-diff-le-diff-add[OF buckets divs])
```

```
  show ?thesis
    using slack unfolding bp-bucket-count-slack-def block by simp
qed

lemma bp-lazy-insert-state-split-potential-credit:
  assumes lazy: bp-lazy-ordered-invariant P
  shows bp-lazy-insert-charge x b P +
      bp-split-potential (bp-lazy-insert-state x b P) ≤
    7 + bp-split-potential P
proof −
  have overflow:
    bp-lazy-insert-charge x b P +
        4 * bp-overflow-potential (bp-lazy-insert-state x b P) ≤
      5 + 4 * bp-overflow-potential P
    by (rule bp-lazy-insert-state-weighted-overflow-credit[OF lazy])
  have slack:
    bp-bucket-count-slack (bp-lazy-insert-state x b P) ≤
      bp-bucket-count-slack P + 2
    by (rule bp-lazy-insert-state-bucket-count-slack-le[OF lazy])
  show ?thesis
    using overflow slack unfolding bp-split-potential-def by linarith
qed
```

The lazy Insert credit proof is the core amortized argument. A local insert either adds an entry to a bucket below the lazy threshold or splits the bucket into strict chunks. The weighted overflow lemma pays for the expensive case: any actual lazy-insert charge plus four times the post-state overflow is at most a constant plus four times the pre-state overflow. The bucket-count slack can increase by only a small constant. Combining those facts yields *bp-lazy-ordered-invariant ?P* $\Longrightarrow$ *bp-lazy-insert-charge ?x ?b ?P + bp-split-potential (bp-lazy-insert-state ?x ?b ?P)* $\leq$ *7 + bp-split-potential ?P*, which is the precise credit inequality used by the costed Insert operation.

```
lemma c-bp-rebucket-build-result [simp]:
  bp-result-of (c-bp-rebucket-build P) = bp-rebucket P
proof −
  obtain bs c where build:
    c-bp-bucketize-entries (bp-block-size P) (bp-entries P) = (bs, c)
    by (cases c-bp-bucketize-entries (bp-block-size P) (bp-entries P))
      simp
  have bs:
    bs = bp-bucketize-entries (bp-block-size P) (bp-entries P)
    using c-bp-bucketize-entries-result[of bp-block-size P bp-entries P]
      build
    by (simp add: bp-result-of-def)
  show ?thesis
    using build bs unfolding bp-result-of-def c-bp-rebucket-build-def
      bp-rebucket-def
    by simp
```

qed

lemma *c-bp-rebucket-build-steps-le*:
assumes *inv*: *bp-invariant P*
shows *bp-steps-of* (*c-bp-rebucket-build P*) ≤
length (*bp-entries P*) +
Suc (*length* (*bp-entries P*) *div bp-block-size P*)
proof −
have *M-pos*: *0* < *bp-block-size P*
using *inv* **unfolding** *bp-invariant-def* **by** *blast*
have *steps*:
bp-steps-of
(*c-bp-bucketize-entries* (*bp-block-size P*) (*bp-entries P*)) ≤
length (*bp-entries P*) +
Suc (*length* (*bp-entries P*) *div bp-block-size P*)
by (*rule c-bp-bucketize-entries-steps-le*[*OF M-pos*])
show *?thesis*
using *steps* **unfolding** *bp-steps-of-def c-bp-rebucket-build-def*
by (*auto split*: *prod.splits*)
qed

lemma *bp-bucket-overflow-zero*:
assumes *length* (*bp-bucket-entries b*) ≤ *M*
shows *bp-bucket-overflow M b* = *0*
using *assms* **unfolding** *bp-bucket-overflow-def* **by** *simp*

lemma *bp-overflow-potential-zero-if-sizes-ok*:
assumes *bp-bucket-sizes-ok P*
shows *bp-overflow-potential P* = *0*
proof −
have ∀ *b*∈*set* (*bp-buckets P*). *bp-bucket-overflow* (*bp-block-size P*) *b* = *0*
using *assms*
unfolding *bp-bucket-sizes-ok-def*
by (*simp add*: *bp-bucket-overflow-zero*)
then show *?thesis*
unfolding *bp-overflow-potential-def* **by** (*induction bp-buckets P*) *auto*
qed

lemma *bp-invariant-overflow-potential-zero*:
assumes *bp-invariant P*
shows *bp-overflow-potential P* = *0*
by (*rule bp-overflow-potential-zero-if-sizes-ok*)
(*use assms* **in** ‹*simp add*: *bp-invariant-def*›)

lemma *bp-bucket-count-slack-zero-if-ratio-ok*:
assumes *bp-bucket-count-ratio-ok P*
shows *bp-bucket-count-slack P* = *0*
using *assms* **unfolding** *bp-bucket-count-ratio-ok-def*
bp-bucket-count-slack-def **by** *simp*

```
lemma bp-split-potential-zero-if-regular:
  assumes inv: bp-invariant P
    and ratio: bp-bucket-count-ratio-ok P
  shows bp-split-potential P = 0
  using bp-invariant-overflow-potential-zero[OF inv]
    bp-bucket-count-slack-zero-if-ratio-ok[OF ratio]
  unfolding bp-split-potential-def by simp

lemma bp-bucket-search-steps-ratio-bound:
  assumes ratio: bp-bucket-count-ratio-ok P
  shows bp-bucket-search-steps P ≤
    bp-ratio-log-budget (length (bp-entries P)) (bp-block-size P)
proof −
  have count-le:
    length (bp-buckets P) ≤
      Suc (length (bp-entries P) div bp-block-size P)
    using ratio unfolding bp-bucket-count-ratio-ok-def by blast
  have floor-log (length (bp-buckets P)) ≤
      floor-log (Suc (length (bp-entries P) div bp-block-size P))
    by (rule floor-log-le-iff[OF count-le])
  then show ?thesis
    unfolding bp-bucket-search-steps-def bp-ratio-log-budget-def by simp
qed

lemma bp-bucket-count-ratio-okI-loaded:
  assumes M-pos: 0 < bp-block-size P
    and loaded: bp-bucket-count-loaded P
  shows bp-bucket-count-ratio-ok P
proof −
  let ?M = bp-block-size P
  let ?k = length (bp-buckets P)
  let ?N = length (bp-entries P)
  have count-le: ?k ≤ Suc (?N div ?M)
  proof (cases ?k)
    case 0
    then show ?thesis by simp
  next
    case (Suc j)
    have ?M * j ≤ ?N
      using loaded Suc unfolding bp-bucket-count-loaded-def by simp
    then have j ≤ ?N div ?M
    proof −
      have j = (?M * j) div ?M
        using M-pos by simp
      also have ... ≤ ?N div ?M
        by (rule div-le-mono) (rule ‹?M * j ≤ ?N›)
      finally show ?thesis .
    qed
```

```
    then show ?thesis
      using Suc by simp
  qed
  show ?thesis
    unfolding bp-bucket-count-ratio-ok-def using M-pos count-le by blast
qed

lemma bp-empty-bucket-count-ratio-ok:
  assumes 0 < M
  shows bp-bucket-count-ratio-ok (bp-empty M B)
  using assms unfolding bp-bucket-count-ratio-ok-def bp-empty-def
    bp-entries-def bp-bucket-entries-flat-def by simp

lemma bp-regular-invariant-ordered-invariant:
  assumes bp-regular-invariant P
  shows bp-ordered-invariant P
  using assms unfolding bp-regular-invariant-def by blast

lemma bp-regular-invariant-ratio-ok:
  assumes bp-regular-invariant P
  shows bp-bucket-count-ratio-ok P
  using assms unfolding bp-regular-invariant-def by blast

lemma bp-empty-regular-invariant:
  assumes 0 < M
  shows bp-regular-invariant (bp-empty M B)
  unfolding bp-regular-invariant-def
  using bp-empty-ordered-invariant[OF assms]
    bp-empty-bucket-count-ratio-ok[OF assms]
  by blast

lemma bp-rebucket-bucket-count-ratio-ok:
  assumes inv: bp-invariant P
  shows bp-bucket-count-ratio-ok (bp-rebucket P)
proof −
  let ?M = bp-block-size P
  let ?xs = bp-entries P
  have M-pos: 0 < ?M
    using inv unfolding bp-invariant-def by blast
  have count:
    length (bp-bucketize-entries ?M ?xs) ≤
      Suc (length ?xs div ?M)
    by (rule length-bp-bucketize-entries-le-ratio[OF M-pos])
  have entries-len:
    length (bp-entries (bp-rebucket P)) = length ?xs
    using bp-bucket-entries-flat-bucketize-entries[OF M-pos, of ?xs]
    unfolding bp-rebucket-def bp-entries-def by simp
  have block-eq [simp]: bp-block-size (bp-rebucket P) = ?M
    unfolding bp-rebucket-def by simp
```

```
  have buckets-eq [simp]:
    bp-buckets (bp-rebucket P) = bp-bucketize-entries ?M ?xs
    unfolding bp-rebucket-def by simp
  show ?thesis
    unfolding bp-bucket-count-ratio-ok-def
    using M-pos count entries-len by simp
qed

lemma bp-rebucket-bucket-count-ratio-ok-from-lazy:
  assumes inv: bp-lazy-invariant P
  shows bp-bucket-count-ratio-ok (bp-rebucket P)
proof −
  let ?M = bp-block-size P
  let ?xs = bp-entries P
  have M-pos: 0 < ?M
    using inv unfolding bp-lazy-invariant-def by blast
  have count:
    length (bp-bucketize-entries ?M ?xs) ≤
      Suc (length ?xs div ?M)
    by (rule length-bp-bucketize-entries-le-ratio[OF M-pos])
  have entries-len:
    length (bp-entries (bp-rebucket P)) = length ?xs
    using bp-bucket-entries-flat-bucketize-entries[OF M-pos, of ?xs]
    unfolding bp-rebucket-def bp-entries-def by simp
  have block-eq [simp]: bp-block-size (bp-rebucket P) = ?M
    unfolding bp-rebucket-def by simp
  have buckets-eq [simp]:
    bp-buckets (bp-rebucket P) = bp-bucketize-entries ?M ?xs
    unfolding bp-rebucket-def by simp
  show ?thesis
    unfolding bp-bucket-count-ratio-ok-def
    using M-pos count entries-len by simp
qed

lemma bp-rebucket-regular-split-potential-zero:
  assumes inv: bp-invariant P
  shows bp-split-potential (bp-rebucket P) = 0
proof (rule bp-split-potential-zero-if-regular)
  show bp-invariant (bp-rebucket P)
    by (rule bp-rebucket-invariant[OF inv])
  show bp-bucket-count-ratio-ok (bp-rebucket P)
    by (rule bp-rebucket-bucket-count-ratio-ok[OF inv])
qed

lemma bp-rebucket-regular-invariant:
  assumes inv: bp-invariant P
  shows bp-regular-invariant (bp-rebucket P)
  unfolding bp-regular-invariant-def
  using bp-rebucket-ordered-invariant[OF inv]
```

bp-rebucket-bucket-count-ratio-ok[*OF inv*]
by *blast*

lemma *bp-rebucket-regular-split-potential-zero-from-lazy*:
assumes *inv*: *bp-lazy-invariant P*
shows *bp-split-potential* (*bp-rebucket P*) = *0*
proof (*rule bp-split-potential-zero-if-regular*)
show *bp-invariant* (*bp-rebucket P*)
by (*rule bp-rebucket-invariant-from-lazy*[*OF inv*])
show *bp-bucket-count-ratio-ok* (*bp-rebucket P*)
by (*rule bp-rebucket-bucket-count-ratio-ok-from-lazy*[*OF inv*])
qed

lemma *bp-rebucket-regular-invariant-from-lazy*:
assumes *inv*: *bp-lazy-invariant P*
shows *bp-regular-invariant* (*bp-rebucket P*)
unfolding *bp-regular-invariant-def*
using *bp-rebucket-ordered-invariant-from-lazy*[*OF inv*]
bp-rebucket-bucket-count-ratio-ok-from-lazy[*OF inv*]
by *blast*

lemma *c-bp-rebucket-build-steps-le-from-lazy*:
assumes *inv*: *bp-lazy-invariant P*
shows *bp-steps-of* (*c-bp-rebucket-build P*) ≤
length (*bp-entries P*) +
Suc (*length* (*bp-entries P*) *div bp-block-size P*)
proof −
have *M-pos*: *0* < *bp-block-size P*
using *inv* **unfolding** *bp-lazy-invariant-def* **by** *blast*
have *steps*:
bp-steps-of
(*c-bp-bucketize-entries* (*bp-block-size P*) (*bp-entries P*)) ≤
length (*bp-entries P*) +
Suc (*length* (*bp-entries P*) *div bp-block-size P*)
by (*rule c-bp-bucketize-entries-steps-le*[*OF M-pos*])
show *?thesis*
using *steps* **unfolding** *bp-steps-of-def c-bp-rebucket-build-def*
by (*auto split*: *prod.splits*)
qed

lemma *c-bp-rebucket-build-regular-invariant-from-lazy*:
assumes *inv*: *bp-lazy-invariant P*
shows *bp-regular-invariant* (*bp-result-of* (*c-bp-rebucket-build P*))
by (*simp add*: *bp-rebucket-regular-invariant-from-lazy*[*OF inv*])

lemma *c-bp-rebucket-build-split-potential-zero-from-lazy*:
assumes *inv*: *bp-lazy-invariant P*
shows *bp-split-potential* (*bp-result-of* (*c-bp-rebucket-build P*)) = *0*
by (*simp add*: *bp-rebucket-regular-split-potential-zero-from-lazy*[*OF inv*])

lemma *c-bp-rebucket-build-regular-invariant*:
assumes *inv*: *bp-invariant P*
shows *bp-regular-invariant* (*bp-result-of* (*c-bp-rebucket-build P*))
by (*simp add*: *bp-rebucket-regular-invariant*[*OF inv*])

lemma *c-bp-rebucket-build-split-potential-zero*:
assumes *inv*: *bp-invariant P*
shows *bp-split-potential* (*bp-result-of* (*c-bp-rebucket-build P*)) = *0*
by (*simp add*: *bp-rebucket-regular-split-potential-zero*[*OF inv*])

lemma *bp-rebucketed-insert-regular-invariant*:
assumes *inv*: *bp-invariant P*
shows *bp-regular-invariant* (*bp-rebucketed-insert x b P*)
unfolding *bp-rebucketed-insert-def*
by (*rule bp-rebucket-regular-invariant*[*OF bp-insert-invariant*[*OF inv*]])

lemma *bp-rebucketed-batch-prepend-regular-invariant*:
assumes *inv*: *bp-invariant P*
shows *bp-regular-invariant* (*bp-rebucketed-batch-prepend xs P*)
unfolding *bp-rebucketed-batch-prepend-def*
by (*rule bp-rebucket-regular-invariant*[*OF bp-batch-prepend-invariant*[*OF inv*]])

lemma *bp-rebucketed-insert-split-potential-zero*:
assumes *inv*: *bp-invariant P*
shows *bp-split-potential* (*bp-rebucketed-insert x b P*) = *0*
unfolding *bp-rebucketed-insert-def*
by (*rule bp-rebucket-regular-split-potential-zero*
[*OF bp-insert-invariant*[*OF inv*]])

lemma *bp-rebucketed-batch-prepend-split-potential-zero*:
assumes *inv*: *bp-invariant P*
shows *bp-split-potential* (*bp-rebucketed-batch-prepend xs P*) = *0*
unfolding *bp-rebucketed-batch-prepend-def*
by (*rule bp-rebucket-regular-split-potential-zero*
[*OF bp-batch-prepend-invariant*[*OF inv*]])

lemma *bp-regularized-local-insert-invariant*:
assumes *ord*: *bp-ordered-invariant P*
shows *bp-invariant* (*bp-regularized-local-insert x b P*)
unfolding *bp-regularized-local-insert-def*
by (*rule bp-rebucket-invariant*[*OF bp-local-insert-state-invariant*[*OF ord*]])

lemma *bp-regularized-local-insert-ordered-invariant*:
assumes *ord*: *bp-ordered-invariant P*
shows *bp-ordered-invariant* (*bp-regularized-local-insert x b P*)
unfolding *bp-regularized-local-insert-def*
by (*rule bp-rebucket-ordered-invariant*
[*OF bp-local-insert-state-invariant*[*OF ord*]])

lemma *bp-regularized-local-insert-regular-invariant*:
assumes *ord*: *bp-ordered-invariant P*
shows *bp-regular-invariant* (*bp-regularized-local-insert x b P*)
unfolding *bp-regularized-local-insert-def*
by (*rule bp-rebucket-regular-invariant*
[*OF bp-local-insert-state-invariant*[*OF ord*]])

lemma *bp-regularized-local-insert-refines-min-update*:
assumes *ord*: *bp-ordered-invariant P*
shows *bp-view* (*bp-regularized-local-insert x b P*) =
min-update (*bp-view P*) *x b*
proof −
have *inv*: *bp-invariant P*
by (*rule bp-ordered-invariant-invariant*[*OF ord*])
then have *M-pos*: *0* < *bp-block-size P*
unfolding *bp-invariant-def* **by** *blast*
have *step-inv*: *bp-invariant* (*bp-local-insert-state x b P*)
by (*rule bp-local-insert-state-invariant*[*OF ord*])
then have *step-M-pos*:
0 < *bp-block-size* (*bp-local-insert-state x b P*)
unfolding *bp-invariant-def* **by** *blast*
have *view-rebucket*:
bp-view (*bp-rebucket* (*bp-local-insert-state x b P*)) =
bp-view (*bp-local-insert-state x b P*)
by (*rule bp-rebucket-view*[*OF step-M-pos*])
show *?thesis*
unfolding *bp-regularized-local-insert-def*
using *view-rebucket bp-local-insert-state-refines-min-update*[*OF M-pos*, *of x b*]
by *simp*
qed

lemma *bp-regularized-local-insert-refines-insert-spec*:
assumes *ord*: *bp-ordered-invariant P*
shows *insert-spec* (*bp-view P*) *x b*
(*bp-view* (*bp-regularized-local-insert x b P*))
unfolding *bp-regularized-local-insert-refines-min-update*[*OF ord*]
by (*rule min-update-insert-spec*)

lemma *bp-regularized-local-insert-split-potential-zero*:
assumes *ord*: *bp-ordered-invariant P*
shows *bp-split-potential* (*bp-regularized-local-insert x b P*) = *0*
unfolding *bp-regularized-local-insert-def*
by (*rule bp-rebucket-regular-split-potential-zero*
[*OF bp-local-insert-state-invariant*[*OF ord*]])

lemma *bp-local-insert-search-charge-ratio-bound*:
assumes *ratio*: *bp-bucket-count-ratio-ok P*
shows *bp-local-insert-search-charge P* ≤

bp-insert-search-budget (*length* (*bp-entries P*)) (*bp-block-size P*)
using *bp-bucket-search-steps-ratio-bound*[*OF ratio*]
unfolding *bp-local-insert-search-charge-def bp-insert-search-budget-def*
by *simp*

lemma *c-bp-local-insert-search-result* [*simp*]:
bp-result-of (*c-bp-local-insert-search x b P*) =
bp-local-insert-state x b P
unfolding *bp-result-of-def c-bp-local-insert-search-def* **by** *simp*

lemma *c-bp-local-insert-search-steps* [*simp*]:
bp-steps-of (*c-bp-local-insert-search x b P*) =
bp-local-insert-search-charge P
unfolding *bp-steps-of-def c-bp-local-insert-search-def* **by** *simp*

lemma *c-bp-local-insert-search-refines-min-update*:
assumes *0* < *bp-block-size P*
shows *bp-view* (*bp-result-of* (*c-bp-local-insert-search x b P*)) =
min-update (*bp-view P*) *x b*
unfolding *c-bp-local-insert-search-result*
by (*rule bp-local-insert-state-refines-min-update*[*OF assms*])

lemma *c-bp-local-insert-search-invariant*:
assumes *bp-ordered-invariant P*
shows *bp-invariant* (*bp-result-of* (*c-bp-local-insert-search x b P*))
unfolding *c-bp-local-insert-search-result*
by (*rule bp-local-insert-state-invariant*[*OF assms*])

lemma *c-bp-local-insert-search-ordered-invariant*:
assumes *bp-ordered-invariant P*
shows *bp-ordered-invariant*
(*bp-result-of* (*c-bp-local-insert-search x b P*))
unfolding *c-bp-local-insert-search-result*
by (*rule bp-local-insert-state-ordered-invariant*[*OF assms*])

lemma *c-bp-local-insert-search-refines-insert-spec*:
assumes *0* < *bp-block-size P*
shows *insert-spec* (*bp-view P*) *x b*
(*bp-view* (*bp-result-of* (*c-bp-local-insert-search x b P*)))
unfolding *c-bp-local-insert-search-refines-min-update*[*OF assms*]
by (*rule min-update-insert-spec*)

lemma *c-bp-local-insert-search-ratio-bound*:
assumes *ratio*: *bp-bucket-count-ratio-ok P*
shows *bp-steps-of* (*c-bp-local-insert-search x b P*) ≤
bp-insert-search-budget (*length* (*bp-entries P*)) (*bp-block-size P*)
by (*simp add*: *bp-local-insert-search-charge-ratio-bound*[*OF ratio*])

lemma *c-bp-local-insert-search-partition-insert-cost-bound*:

assumes *ratio*: *bp-bucket-count-ratio-ok P*
shows *partition-insert-cost-bound*
(*bp-steps-of* (*c-bp-local-insert-search x b P*))
(*bp-insert-search-budget* (*length* (*bp-entries P*)) (*bp-block-size P*))
using *c-bp-local-insert-search-ratio-bound*[*OF ratio*]
unfolding *partition-insert-cost-bound-def* **.**

lemma *c-bp-local-insert-split-result* [*simp*]:
bp-result-of (*c-bp-local-insert-split x b P*) =
bp-local-insert-state x b P
unfolding *bp-result-of-def c-bp-local-insert-split-def* **by** *simp*

lemma *c-bp-local-insert-split-steps* [*simp*]:
bp-steps-of (*c-bp-local-insert-split x b P*) =
bp-local-insert-search-charge P +
bp-local-split-budget (*bp-block-size P*)
unfolding *bp-steps-of-def c-bp-local-insert-split-def* **by** *simp*

lemma *c-bp-local-insert-split-refines-min-update*:
assumes *0* < *bp-block-size P*
shows *bp-view* (*bp-result-of* (*c-bp-local-insert-split x b P*)) =
min-update (*bp-view P*) *x b*
unfolding *c-bp-local-insert-split-result*
by (*rule bp-local-insert-state-refines-min-update*[*OF assms*])

lemma *c-bp-local-insert-split-invariant*:
assumes *bp-ordered-invariant P*
shows *bp-invariant* (*bp-result-of* (*c-bp-local-insert-split x b P*))
unfolding *c-bp-local-insert-split-result*
by (*rule bp-local-insert-state-invariant*[*OF assms*])

lemma *c-bp-local-insert-split-ordered-invariant*:
assumes *bp-ordered-invariant P*
shows *bp-ordered-invariant*
(*bp-result-of* (*c-bp-local-insert-split x b P*))
unfolding *c-bp-local-insert-split-result*
by (*rule bp-local-insert-state-ordered-invariant*[*OF assms*])

lemma *c-bp-local-insert-split-refines-insert-spec*:
assumes *0* < *bp-block-size P*
shows *insert-spec* (*bp-view P*) *x b*
(*bp-view* (*bp-result-of* (*c-bp-local-insert-split x b P*)))
unfolding *c-bp-local-insert-split-refines-min-update*[*OF assms*]
by (*rule min-update-insert-spec*)

lemma *c-bp-local-insert-split-steps-le*:
assumes *ratio*: *bp-bucket-count-ratio-ok P*
shows *bp-steps-of* (*c-bp-local-insert-split x b P*) ≤
bp-insert-search-budget (*length* (*bp-entries P*)) (*bp-block-size P*) +

```
    bp-local-split-budget (bp-block-size P)
  using bp-local-insert-search-charge-ratio-bound[OF ratio]
  by simp

lemma c-bp-local-insert-split-steps-le-lazy-budget:
  assumes ratio: bp-bucket-count-ratio-ok P
  shows bp-steps-of (c-bp-local-insert-split x b P) ≤
    bp-insert-search-budget (length (bp-entries P)) (bp-block-size P) +
      bp-lazy-split-budget (bp-block-size P)
proof −
  have local:
    bp-steps-of (c-bp-local-insert-split x b P) ≤
      bp-insert-search-budget (length (bp-entries P)) (bp-block-size P) +
        bp-local-split-budget (bp-block-size P)
    by (rule c-bp-local-insert-split-steps-le[OF ratio])
  have bp-local-split-budget (bp-block-size P) ≤
      bp-lazy-split-budget (bp-block-size P)
    by (rule bp-local-split-budget-le-lazy-split-budget)
  then show ?thesis
    using local by linarith
qed

lemma c-bp-local-insert-split-amortized-if-split-credit:
  assumes ratio: bp-bucket-count-ratio-ok P
    and paid:
      bp-local-split-budget (bp-block-size P) +
        bp-split-potential
          (bp-result-of (c-bp-local-insert-split x b P)) ≤
        bp-split-potential P
  shows bp-amortized-step-bound
    (bp-steps-of (c-bp-local-insert-split x b P)) P
    (bp-result-of (c-bp-local-insert-split x b P))
    (bp-insert-search-budget (length (bp-entries P)) (bp-block-size P))
proof −
  have search:
    bp-local-insert-search-charge P ≤
      bp-insert-search-budget (length (bp-entries P)) (bp-block-size P)
    by (rule bp-local-insert-search-charge-ratio-bound[OF ratio])
  have steps:
    bp-steps-of (c-bp-local-insert-split x b P) =
      bp-local-insert-search-charge P +
        bp-local-split-budget (bp-block-size P)
    by simp
  show ?thesis
    using search paid steps
    unfolding bp-amortized-step-bound-def
    by linarith
qed
```

lemma *c-bp-lazy-insert-result* [*simp*]:
bp-result-of (*c-bp-lazy-insert x b P*) =
bp-lazy-insert-state x b P
unfolding *bp-result-of-def c-bp-lazy-insert-def* **by** *simp*

lemma *c-bp-lazy-insert-steps* [*simp*]:
bp-steps-of (*c-bp-lazy-insert x b P*) =
bp-local-insert-search-charge P +
bp-lazy-split-budget (*bp-block-size P*)
unfolding *bp-steps-of-def c-bp-lazy-insert-def* **by** *simp*

lemma *c-bp-lazy-insert-refines-min-update*:
assumes *0* < *bp-block-size P*
shows *bp-view* (*bp-result-of* (*c-bp-lazy-insert x b P*)) =
min-update (*bp-view P*) *x b*
unfolding *c-bp-lazy-insert-result*
by (*rule bp-lazy-insert-state-refines-min-update*[*OF assms*])

lemma *c-bp-lazy-insert-lazy-ordered-invariant*:
assumes *bp-lazy-ordered-invariant P*
shows *bp-lazy-ordered-invariant*
(*bp-result-of* (*c-bp-lazy-insert x b P*))
unfolding *c-bp-lazy-insert-result*
by (*rule bp-lazy-insert-state-lazy-ordered-invariant*[*OF assms*])

lemma *c-bp-lazy-insert-refines-insert-spec*:
assumes *0* < *bp-block-size P*
shows *insert-spec* (*bp-view P*) *x b*
(*bp-view* (*bp-result-of* (*c-bp-lazy-insert x b P*)))
unfolding *c-bp-lazy-insert-refines-min-update*[*OF assms*]
by (*rule min-update-insert-spec*)

lemma *c-bp-lazy-insert-steps-le-lazy-budget*:
assumes *ratio*: *bp-bucket-count-ratio-ok P*
shows *bp-steps-of* (*c-bp-lazy-insert x b P*) ≤
bp-insert-search-budget (*length* (*bp-entries P*)) (*bp-block-size P*) +
bp-lazy-split-budget (*bp-block-size P*)
using *bp-local-insert-search-charge-ratio-bound*[*OF ratio*]
by *simp*

lemma *c-bp-lazy-insert-amortized-if-split-credit*:
assumes *ratio*: *bp-bucket-count-ratio-ok P*
and *paid*:
bp-lazy-split-budget (*bp-block-size P*) +
bp-split-potential (*bp-result-of* (*c-bp-lazy-insert x b P*)) ≤
bp-split-potential P
shows *bp-amortized-step-bound*
(*bp-steps-of* (*c-bp-lazy-insert x b P*)) *P*
(*bp-result-of* (*c-bp-lazy-insert x b P*))

```
    (bp-insert-search-budget (length (bp-entries P)) (bp-block-size P))
proof −
  have search:
    bp-local-insert-search-charge P ≤
      bp-insert-search-budget (length (bp-entries P)) (bp-block-size P)
    by (rule bp-local-insert-search-charge-ratio-bound[OF ratio])
  have steps:
    bp-steps-of (c-bp-lazy-insert x b P) =
      bp-local-insert-search-charge P +
        bp-lazy-split-budget (bp-block-size P)
    by simp
  show ?thesis
    using search paid steps
    unfolding bp-amortized-step-bound-def
    by linarith
qed

lemma bp-lazy-bucket-insert-charge-le:
  bp-lazy-bucket-insert-charge M b ≤ bp-lazy-split-budget M
  unfolding bp-lazy-bucket-insert-charge-def bp-lazy-split-budget-def
  by simp

lemma bp-lazy-insert-bucket-charge-le:
  bp-lazy-insert-bucket-charge M p bs ≤ bp-lazy-split-budget M
proof (induction bs arbitrary: p)
  case Nil
  then show ?case
    unfolding bp-lazy-split-budget-def by simp
next
  case (Cons b bs)
  note IH = Cons.IH
  show ?case
  proof (cases bs)
    case Nil
    then show ?thesis
      using bp-lazy-bucket-insert-charge-le[of M b]
      unfolding bp-lazy-split-budget-def by auto
  next
    case (Cons c cs)
    have one-le: 1 ≤ bp-lazy-split-budget M
      unfolding bp-lazy-split-budget-def by simp
    then show ?thesis
      using Cons IH[of p] bp-lazy-bucket-insert-charge-le[of M b] one-le
      by auto
  qed
qed

lemma bp-lazy-insert-charge-le:
  bp-lazy-insert-charge x b P ≤ bp-lazy-split-budget (bp-block-size P)
```

proof −
let *?old-keys* = *bp-entry-keys* (*bp-entries P*)
let *?b* = *if x* ∈ *?old-keys then min* (*bp-values P x*) *b else b*
let *?P0* = *bp-delete-key x P*
have *charge*:
bp-lazy-insert-bucket-charge (*bp-block-size ?P0*) (*x*, *?b*)
(*bp-buckets ?P0*) ≤ *bp-lazy-split-budget* (*bp-block-size ?P0*)
by (*rule bp-lazy-insert-bucket-charge-le*)
have *block*: *bp-block-size ?P0* = *bp-block-size P*
unfolding *bp-delete-key-def* **by** *simp*
show *?thesis*
using *charge* **unfolding** *bp-lazy-insert-charge-def Let-def block* **by** *simp*
qed

lemma *c-bp-lazy-insert-precise-result* [*simp*]:
bp-result-of (*c-bp-lazy-insert-precise x b P*) =
bp-lazy-insert-state x b P
unfolding *bp-result-of-def c-bp-lazy-insert-precise-def* **by** *simp*

lemma *c-bp-lazy-insert-precise-steps* [*simp*]:
bp-steps-of (*c-bp-lazy-insert-precise x b P*) =
bp-local-insert-search-charge P + *bp-lazy-insert-charge x b P*
unfolding *bp-steps-of-def c-bp-lazy-insert-precise-def* **by** *simp*

lemma *c-bp-lazy-insert-precise-refines-min-update*:
assumes *0* < *bp-block-size P*
shows *bp-view* (*bp-result-of* (*c-bp-lazy-insert-precise x b P*)) =
min-update (*bp-view P*) *x b*
unfolding *c-bp-lazy-insert-precise-result*
by (*rule bp-lazy-insert-state-refines-min-update*[*OF assms*])

lemma *c-bp-lazy-insert-precise-lazy-ordered-invariant*:
assumes *bp-lazy-ordered-invariant P*
shows *bp-lazy-ordered-invariant*
(*bp-result-of* (*c-bp-lazy-insert-precise x b P*))
unfolding *c-bp-lazy-insert-precise-result*
by (*rule bp-lazy-insert-state-lazy-ordered-invariant*[*OF assms*])

lemma *c-bp-lazy-insert-precise-refines-insert-spec*:
assumes *0* < *bp-block-size P*
shows *insert-spec* (*bp-view P*) *x b*
(*bp-view* (*bp-result-of* (*c-bp-lazy-insert-precise x b P*)))
unfolding *c-bp-lazy-insert-precise-refines-min-update*[*OF assms*]
by (*rule min-update-insert-spec*)

lemma *c-bp-lazy-insert-precise-steps-le-lazy-budget*:
assumes *ratio*: *bp-bucket-count-ratio-ok P*
shows *bp-steps-of* (*c-bp-lazy-insert-precise x b P*) ≤
bp-insert-search-budget (*length* (*bp-entries P*)) (*bp-block-size P*) +

```
    bp-lazy-split-budget (bp-block-size P)
proof −
  have search:
    bp-local-insert-search-charge P ≤
      bp-insert-search-budget (length (bp-entries P)) (bp-block-size P)
    by (rule bp-local-insert-search-charge-ratio-bound[OF ratio])
  have charge:
    bp-lazy-insert-charge x b P ≤
      bp-lazy-split-budget (bp-block-size P)
    by (rule bp-lazy-insert-charge-le)
  show ?thesis
    using search charge by simp
qed

lemma c-bp-lazy-insert-precise-steps-le-coarse:
  bp-steps-of (c-bp-lazy-insert-precise x b P) ≤
    bp-steps-of (c-bp-lazy-insert x b P)
  using bp-lazy-insert-charge-le[of x b P] by simp

lemma c-bp-lazy-insert-precise-amortized-if-credit:
  assumes ratio: bp-bucket-count-ratio-ok P
    and paid:
      bp-lazy-insert-charge x b P +
        bp-split-potential
          (bp-result-of (c-bp-lazy-insert-precise x b P)) ≤
        extra + bp-split-potential P
  shows bp-amortized-step-bound
    (bp-steps-of (c-bp-lazy-insert-precise x b P)) P
    (bp-result-of (c-bp-lazy-insert-precise x b P))
    (bp-insert-search-budget (length (bp-entries P)) (bp-block-size P) +
      extra)
proof −
  have search:
    bp-local-insert-search-charge P ≤
      bp-insert-search-budget (length (bp-entries P)) (bp-block-size P)
    by (rule bp-local-insert-search-charge-ratio-bound[OF ratio])
  show ?thesis
    using search paid
    unfolding bp-amortized-step-bound-def
    by simp
qed

lemma c-bp-lazy-insert-precise-amortized-bound:
  assumes lazy: bp-lazy-ordered-invariant P
    and ratio: bp-bucket-count-ratio-ok P
  shows bp-amortized-step-bound
    (bp-steps-of (c-bp-lazy-insert-precise x b P)) P
    (bp-result-of (c-bp-lazy-insert-precise x b P))
    (bp-insert-search-budget (length (bp-entries P)) (bp-block-size P) + 7)
```

proof (*rule c-bp-lazy-insert-precise-amortized-if-credit*[*OF ratio*])
  show *bp-lazy-insert-charge x b P* +
    bp-split-potential (*bp-result-of* (*c-bp-lazy-insert-precise x b P*)) ≤
    7 + *bp-split-potential P*
    using *bp-lazy-insert-state-split-potential-credit*[*OF lazy*, *of x b*]
    by *simp*
qed

lemma *c-bp-lazy-insert-precise-amortized-ratio-budget*:
  assumes *lazy*: *bp-lazy-ordered-invariant P*
    and *ratio*: *bp-bucket-count-ratio-ok P*
  shows *bp-amortized-step-bound*
    (*bp-steps-of* (*c-bp-lazy-insert-precise x b P*)) *P*
    (*bp-result-of* (*c-bp-lazy-insert-precise x b P*))
    (*bp-lazy-insert-amortized-budget P*)
  unfolding *bp-lazy-insert-amortized-budget-def*
  by (*rule c-bp-lazy-insert-precise-amortized-bound*[*OF lazy ratio*])

lemma *c-bp-lazy-insert-precise-regular-ratio-bound*:
  assumes *reg*: *bp-regular-invariant P*
  shows *bp-steps-of* (*c-bp-lazy-insert-precise x b P*) ≤
    bp-lazy-insert-amortized-budget P
proof −
  have *ord*: *bp-ordered-invariant P*
    by (*rule bp-regular-invariant-ordered-invariant*[*OF reg*])
  have *lazy*: *bp-lazy-ordered-invariant P*
    by (*rule bp-ordered-invariant-lazy-ordered-invariant*[*OF ord*])
  have *ratio*: *bp-bucket-count-ratio-ok P*
    by (*rule bp-regular-invariant-ratio-ok*[*OF reg*])
  have *am*: *bp-amortized-step-bound*
      (*bp-steps-of* (*c-bp-lazy-insert-precise x b P*)) *P*
      (*bp-result-of* (*c-bp-lazy-insert-precise x b P*))
      (*bp-lazy-insert-amortized-budget P*)
    by (*rule c-bp-lazy-insert-precise-amortized-ratio-budget*[*OF lazy ratio*])
  have *pot*: *bp-split-potential P* = *0*
    by (*rule bp-split-potential-zero-if-regular*
        [*OF bp-ordered-invariant-invariant*[*OF ord*] *ratio*])
  show *?thesis*
    using *am pot* **unfolding** *bp-amortized-step-bound-def* **by** *linarith*
qed

lemma *c-bp-lazy-insert-precise-regular-partition-insert-cost-bound*:
  assumes *reg*: *bp-regular-invariant P*
  shows *partition-insert-cost-bound*
    (*bp-steps-of* (*c-bp-lazy-insert-precise x b P*))
    (*bp-lazy-insert-amortized-budget P*)
  using *c-bp-lazy-insert-precise-regular-ratio-bound*[*OF reg*]
  unfolding *partition-insert-cost-bound-def* **.**

lemma *c-bp-lazy-batch-prepend-precise-result* [*simp*]:
  bp-result-of (*c-bp-lazy-batch-prepend-precise xs P*) =
    bp-lazy-batch-prepend-state xs P
proof (*induction xs arbitrary*: *P*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons xb xs*)
  obtain *x b* **where** *xb*: *xb* = (*x*, *b*)
    by *force*
  show *?case*
    unfolding *xb* **by** (*simp add*: *Cons.IH*)
qed

lemma *c-bp-lazy-batch-prepend-precise-steps-Nil* [*simp*]:
  bp-steps-of (*c-bp-lazy-batch-prepend-precise* [] *P*) = *0*
  unfolding *bp-steps-of-def* **by** *simp*

lemma *c-bp-lazy-batch-prepend-precise-steps-Cons* [*simp*]:
  bp-steps-of (*c-bp-lazy-batch-prepend-precise* ((*x*, *b*) # *xs*) *P*) =
    bp-steps-of (*c-bp-lazy-insert-precise x b P*) +
    bp-steps-of
      (*c-bp-lazy-batch-prepend-precise xs*
        (*bp-result-of* (*c-bp-lazy-insert-precise x b P*)))
  unfolding *bp-steps-of-def bp-result-of-def*
  by (*simp split*: *prod.splits*)

lemma *c-bp-lazy-batch-prepend-precise-lazy-ordered-invariant*:
  assumes *lazy*: *bp-lazy-ordered-invariant P*
  shows *bp-lazy-ordered-invariant*
    (*bp-result-of* (*c-bp-lazy-batch-prepend-precise xs P*))
  using *lazy*
proof (*induction xs arbitrary*: *P*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons xb xs*)
  obtain *x b* **where** *xb*: *xb* = (*x*, *b*)
    by *force*
  have *step-lazy*: *bp-lazy-ordered-invariant*
      (*bp-result-of* (*c-bp-lazy-insert-precise x b P*))
    by (*rule c-bp-lazy-insert-precise-lazy-ordered-invariant*[*OF Cons.prems*])
  show *?case*
    unfolding *xb* **using** *Cons.IH*[*OF step-lazy*] **by** *simp*
qed

lemma *c-bp-lazy-batch-prepend-precise-refines-batch-min-update*:
  assumes *lazy*: *bp-lazy-ordered-invariant P*
  shows *bp-view* (*bp-result-of* (*c-bp-lazy-batch-prepend-precise xs P*)) =

```
    batch-min-update (bp-view P) xs
  using lazy
proof (induction xs arbitrary: P)
  case Nil
  then show ?case
    unfolding batch-min-update-def by simp
next
  case (Cons xb xs)
  obtain x b where xb: xb = (x, b)
    by force
  have M-pos: 0 < bp-block-size P
    using Cons.prems unfolding bp-lazy-ordered-invariant-def
      bp-lazy-invariant-def by blast
  have step-view:
    bp-view (bp-result-of (c-bp-lazy-insert-precise x b P)) =
      min-update (bp-view P) x b
    by (rule c-bp-lazy-insert-precise-refines-min-update[OF M-pos])
  have step-lazy: bp-lazy-ordered-invariant
      (bp-result-of (c-bp-lazy-insert-precise x b P))
    by (rule c-bp-lazy-insert-precise-lazy-ordered-invariant[OF Cons.prems])
  show ?case
    unfolding xb
    using Cons.IH[OF step-lazy] step-view
    by (simp add: batch-min-update-def)
qed

lemma c-bp-lazy-batch-prepend-precise-amortized-bound:
  assumes lazy: bp-lazy-ordered-invariant P
    and ratios: bp-lazy-batch-ratio-ok xs P
  shows bp-amortized-step-bound
    (bp-steps-of (c-bp-lazy-batch-prepend-precise xs P)) P
    (bp-result-of (c-bp-lazy-batch-prepend-precise xs P))
    (bp-lazy-batch-insert-budget-sum xs P)
  using lazy ratios
proof (induction xs arbitrary: P)
  case Nil
  then show ?case
    unfolding bp-amortized-step-bound-def by simp
next
  case (Cons xb xs)
  obtain x b where xb: xb = (x, b)
    by force
  let ?I = c-bp-lazy-insert-precise x b P
  let ?P1 = bp-result-of ?I
  have ratio: bp-bucket-count-ratio-ok P
    using Cons.prems(2) unfolding xb by simp
  have tail-ratios: bp-lazy-batch-ratio-ok xs ?P1
    using Cons.prems(2) unfolding xb by simp
  have step-lazy: bp-lazy-ordered-invariant ?P1
```

```
    by (rule c-bp-lazy-insert-precise-lazy-ordered-invariant[OF Cons.prems(1)])
  have insert-bound:
    bp-steps-of ?I + bp-split-potential ?P1 ≤
      bp-lazy-insert-amortized-budget P + bp-split-potential P
    using c-bp-lazy-insert-precise-amortized-ratio-budget
      [OF Cons.prems(1) ratio, of x b]
    unfolding bp-amortized-step-bound-def .
  have tail-bound:
    bp-steps-of (c-bp-lazy-batch-prepend-precise xs ?P1) +
        bp-split-potential
          (bp-result-of (c-bp-lazy-batch-prepend-precise xs ?P1)) ≤
      bp-lazy-batch-insert-budget-sum xs ?P1 + bp-split-potential ?P1
    using Cons.IH[OF step-lazy tail-ratios]
    unfolding bp-amortized-step-bound-def .
  show ?case
    unfolding xb bp-amortized-step-bound-def
    using insert-bound tail-bound by simp
qed

lemma c-bp-lazy-batch-prepend-precise-regular-steps-le-budget-sum:
  assumes reg: bp-regular-invariant P
    and ratios: bp-lazy-batch-ratio-ok xs P
  shows bp-steps-of (c-bp-lazy-batch-prepend-precise xs P) ≤
    bp-lazy-batch-insert-budget-sum xs P
proof −
  have ord: bp-ordered-invariant P
    by (rule bp-regular-invariant-ordered-invariant[OF reg])
  have lazy: bp-lazy-ordered-invariant P
    by (rule bp-ordered-invariant-lazy-ordered-invariant[OF ord])
  have am: bp-amortized-step-bound
      (bp-steps-of (c-bp-lazy-batch-prepend-precise xs P)) P
      (bp-result-of (c-bp-lazy-batch-prepend-precise xs P))
      (bp-lazy-batch-insert-budget-sum xs P)
    by (rule c-bp-lazy-batch-prepend-precise-amortized-bound[OF lazy ratios])
  have pot: bp-split-potential P = 0
    by (rule bp-split-potential-zero-if-regular
        [OF bp-ordered-invariant-invariant[OF ord]
          bp-regular-invariant-ratio-ok[OF reg]])
  show ?thesis
    using am pot unfolding bp-amortized-step-bound-def by linarith
qed

lemma c-bp-regularized-local-insert-result [simp]:
  bp-result-of (c-bp-regularized-local-insert x b P) =
    bp-regularized-local-insert x b P
  unfolding bp-result-of-def c-bp-regularized-local-insert-def by simp

lemma c-bp-regularized-local-insert-steps [simp]:
  bp-steps-of (c-bp-regularized-local-insert x b P) =
```

bp-local-insert-search-charge P
unfolding *bp-steps-of-def c-bp-regularized-local-insert-def* **by** *simp*

lemma *c-bp-regularized-local-insert-regular-invariant*:
assumes *reg*: *bp-regular-invariant P*
shows *bp-regular-invariant*
(*bp-result-of* (*c-bp-regularized-local-insert x b P*))
unfolding *c-bp-regularized-local-insert-result*
by (*rule bp-regularized-local-insert-regular-invariant*
[*OF bp-regular-invariant-ordered-invariant*[*OF reg*]])

lemma *c-bp-regularized-local-insert-refines-min-update*:
assumes *reg*: *bp-regular-invariant P*
shows *bp-view* (*bp-result-of* (*c-bp-regularized-local-insert x b P*)) =
min-update (*bp-view P*) *x b*
unfolding *c-bp-regularized-local-insert-result*
by (*rule bp-regularized-local-insert-refines-min-update*
[*OF bp-regular-invariant-ordered-invariant*[*OF reg*]])

lemma *c-bp-regularized-local-insert-refines-insert-spec*:
assumes *reg*: *bp-regular-invariant P*
shows *insert-spec* (*bp-view P*) *x b*
(*bp-view* (*bp-result-of* (*c-bp-regularized-local-insert x b P*)))
unfolding *c-bp-regularized-local-insert-refines-min-update*[*OF reg*]
by (*rule min-update-insert-spec*)

lemma *c-bp-regularized-local-insert-ratio-bound*:
assumes *reg*: *bp-regular-invariant P*
shows *bp-steps-of* (*c-bp-regularized-local-insert x b P*) ≤
bp-insert-search-budget (*length* (*bp-entries P*)) (*bp-block-size P*)
by (*simp add*: *bp-local-insert-search-charge-ratio-bound*
[*OF bp-regular-invariant-ratio-ok*[*OF reg*]])

lemma *c-bp-regularized-local-insert-split-potential-zero*:
assumes *reg*: *bp-regular-invariant P*
shows *bp-split-potential*
(*bp-result-of* (*c-bp-regularized-local-insert x b P*)) = *0*
unfolding *c-bp-regularized-local-insert-result*
by (*rule bp-regularized-local-insert-split-potential-zero*
[*OF bp-regular-invariant-ordered-invariant*[*OF reg*]])

lemma *bp-regular-invariant-split-potential-zero*:
assumes *reg*: *bp-regular-invariant P*
shows *bp-split-potential P* = *0*
proof (*rule bp-split-potential-zero-if-regular*)
show *bp-invariant P*
by (*rule bp-ordered-invariant-invariant*
[*OF bp-regular-invariant-ordered-invariant*[*OF reg*]])
show *bp-bucket-count-ratio-ok P*

```
    by (rule bp-regular-invariant-ratio-ok[OF reg])
qed

lemma c-bp-regularized-local-insert-amortized-ratio-bound:
  assumes reg: bp-regular-invariant P
  shows bp-amortized-step-bound
    (bp-steps-of (c-bp-regularized-local-insert x b P)) P
    (bp-result-of (c-bp-regularized-local-insert x b P))
    (bp-insert-search-budget (length (bp-entries P)) (bp-block-size P))
  using c-bp-regularized-local-insert-ratio-bound[OF reg]
    c-bp-regularized-local-insert-split-potential-zero[OF reg, of x b]
    bp-regular-invariant-split-potential-zero[OF reg]
  unfolding bp-amortized-step-bound-def by simp

lemma c-bp-regularized-local-insert-partition-insert-cost-bound:
  assumes reg: bp-regular-invariant P
  shows partition-insert-cost-bound
    (bp-steps-of (c-bp-regularized-local-insert x b P))
    (bp-insert-search-budget (length (bp-entries P)) (bp-block-size P))
  using c-bp-regularized-local-insert-ratio-bound[OF reg]
  unfolding partition-insert-cost-bound-def .

lemma c-bp-regularized-local-insert-build-result [simp]:
  bp-result-of (c-bp-regularized-local-insert-build x b P) =
    bp-regularized-local-insert x b P
proof −
  let ?Q = bp-local-insert-state x b P
  obtain P' c where build: c-bp-rebucket-build ?Q = (P', c)
    by (cases c-bp-rebucket-build ?Q)
  have P': P' = bp-rebucket ?Q
    using c-bp-rebucket-build-result[of ?Q] build
    unfolding bp-result-of-def by simp
  show ?thesis
    using build P'
    unfolding c-bp-regularized-local-insert-build-def
      bp-regularized-local-insert-def bp-result-of-def
    by simp
qed

lemma c-bp-regularized-local-insert-build-steps:
  bp-steps-of (c-bp-regularized-local-insert-build x b P) =
    bp-local-insert-search-charge P +
      bp-steps-of (c-bp-rebucket-build (bp-local-insert-state x b P))
  unfolding c-bp-regularized-local-insert-build-def
  by (auto simp: bp-steps-of-def split: prod.splits)

lemma c-bp-regularized-local-insert-build-regular-invariant:
  assumes reg: bp-regular-invariant P
  shows bp-regular-invariant
```

```
    (bp-result-of (c-bp-regularized-local-insert-build x b P))
  unfolding c-bp-regularized-local-insert-build-result
  by (rule bp-regularized-local-insert-regular-invariant
      [OF bp-regular-invariant-ordered-invariant[OF reg]])

lemma c-bp-regularized-local-insert-build-refines-min-update:
  assumes reg: bp-regular-invariant P
  shows bp-view (bp-result-of
      (c-bp-regularized-local-insert-build x b P)) =
    min-update (bp-view P) x b
  unfolding c-bp-regularized-local-insert-build-result
  by (rule bp-regularized-local-insert-refines-min-update
      [OF bp-regular-invariant-ordered-invariant[OF reg]])

lemma c-bp-regularized-local-insert-build-refines-insert-spec:
  assumes reg: bp-regular-invariant P
  shows insert-spec (bp-view P) x b
    (bp-view (bp-result-of
      (c-bp-regularized-local-insert-build x b P)))
  unfolding c-bp-regularized-local-insert-build-refines-min-update[OF reg]
  by (rule min-update-insert-spec)

lemma c-bp-regularized-local-insert-build-split-potential-zero:
  assumes reg: bp-regular-invariant P
  shows bp-split-potential
    (bp-result-of (c-bp-regularized-local-insert-build x b P)) = 0
  unfolding c-bp-regularized-local-insert-build-result
  by (rule bp-regularized-local-insert-split-potential-zero
      [OF bp-regular-invariant-ordered-invariant[OF reg]])

lemma c-bp-regularized-local-insert-build-rebuild-steps-le:
  assumes ord: bp-ordered-invariant P
  shows bp-steps-of (c-bp-regularized-local-insert-build x b P) ≤
    bp-local-insert-search-charge P +
      length (bp-entries (bp-local-insert-state x b P)) +
      Suc (length (bp-entries (bp-local-insert-state x b P)) div
        bp-block-size P)
proof −
  have step-inv: bp-invariant (bp-local-insert-state x b P)
    by (rule bp-local-insert-state-invariant[OF ord])
  have build:
    bp-steps-of (c-bp-rebucket-build (bp-local-insert-state x b P)) ≤
      length (bp-entries (bp-local-insert-state x b P)) +
      Suc (length (bp-entries (bp-local-insert-state x b P)) div
        bp-block-size (bp-local-insert-state x b P))
    by (rule c-bp-rebucket-build-steps-le[OF step-inv])
  have block:
    bp-block-size (bp-local-insert-state x b P) = bp-block-size P
    unfolding bp-local-insert-state-def bp-delete-key-def
```

```
    by (simp add: Let-def split: if-splits)
  show ?thesis
    using build
    unfolding c-bp-regularized-local-insert-build-steps block
    by simp
qed

lemma c-bp-regularized-local-insert-build-steps-le:
  assumes reg: bp-regular-invariant P
  shows bp-steps-of (c-bp-regularized-local-insert-build x b P) ≤
    bp-insert-search-budget (length (bp-entries P)) (bp-block-size P) +
      length (bp-entries (bp-local-insert-state x b P)) +
      Suc (length (bp-entries (bp-local-insert-state x b P)) div
        bp-block-size P)
proof −
  have ord: bp-ordered-invariant P
    by (rule bp-regular-invariant-ordered-invariant[OF reg])
  have search:
    bp-local-insert-search-charge P ≤
      bp-insert-search-budget (length (bp-entries P)) (bp-block-size P)
    by (rule bp-local-insert-search-charge-ratio-bound
        [OF bp-regular-invariant-ratio-ok[OF reg]])
  have build:
    bp-steps-of (c-bp-regularized-local-insert-build x b P) ≤
      bp-local-insert-search-charge P +
        length (bp-entries (bp-local-insert-state x b P)) +
        Suc (length (bp-entries (bp-local-insert-state x b P)) div
          bp-block-size P)
    by (rule c-bp-regularized-local-insert-build-rebuild-steps-le[OF ord])
  show ?thesis
    using search build by linarith
qed

lemma c-bp-regularized-local-insert-build-amortized-rebuild-bound:
  assumes reg: bp-regular-invariant P
  shows bp-amortized-step-bound
    (bp-steps-of (c-bp-regularized-local-insert-build x b P)) P
    (bp-result-of (c-bp-regularized-local-insert-build x b P))
    (bp-insert-search-budget (length (bp-entries P)) (bp-block-size P) +
      length (bp-entries (bp-local-insert-state x b P)) +
      Suc (length (bp-entries (bp-local-insert-state x b P)) div
        bp-block-size P))
  using c-bp-regularized-local-insert-build-steps-le[OF reg, of x b]
    c-bp-regularized-local-insert-build-split-potential-zero[OF reg, of x b]
    bp-regular-invariant-split-potential-zero[OF reg]
  unfolding bp-amortized-step-bound-def by simp

lemma c-bp-rebucketed-batch-prepend-bulk-result [simp]:
  bp-result-of (c-bp-rebucketed-batch-prepend-bulk xs P) =
```

bp-rebucketed-batch-prepend xs P
unfolding *bp-result-of-def c-bp-rebucketed-batch-prepend-bulk-def* **by** *simp*

lemma *c-bp-rebucketed-batch-prepend-bulk-steps* [*simp*]:
bp-steps-of (*c-bp-rebucketed-batch-prepend-bulk xs P*) =
bp-batch-prepend-log-budget xs (*bp-block-size P*)
unfolding *bp-steps-of-def c-bp-rebucketed-batch-prepend-bulk-def* **by** *simp*

lemma *c-bp-bucketed-batch-prepend-direct-result* [*simp*]:
bp-result-of (*c-bp-bucketed-batch-prepend-direct xs P*) =
bp-bucketed-batch-prepend-state xs P
unfolding *c-bp-bucketed-batch-prepend-direct-def* **by** *simp*

lemma *c-bp-bucketed-batch-prepend-direct-steps* [*simp*]:
bp-steps-of (*c-bp-bucketed-batch-prepend-direct xs P*) =
bp-batch-prepend-amortized-budget xs (*bp-block-size P*)
unfolding *c-bp-bucketed-batch-prepend-direct-def* **by** *simp*

lemma *c-bp-bucketed-batch-prepend-direct-actual-result* [*simp*]:
bp-result-of (*c-bp-bucketed-batch-prepend-direct-actual xs P*) =
bp-bucketed-batch-prepend-state xs P
unfolding *c-bp-bucketed-batch-prepend-direct-actual-def* **by** *simp*

lemma *c-bp-bucketed-batch-prepend-direct-actual-steps* [*simp*]:
bp-steps-of (*c-bp-bucketed-batch-prepend-direct-actual xs P*) =
bp-batch-prepend-log-budget xs (*bp-block-size P*)
unfolding *c-bp-bucketed-batch-prepend-direct-actual-def* **by** *simp*

lemma *c-bp-bucketed-batch-prepend-direct-paper-batch-cost-bound*:
partition-batch-cost-bound
(*bp-steps-of* (*c-bp-bucketed-batch-prepend-direct xs P*))
(*bp-batch-prepend-per-item-budget xs* (*bp-block-size P*)) *xs*
unfolding *partition-batch-cost-bound-def*
by (*simp add*: *bp-batch-prepend-amortized-budget-alt*)

lemma *bp-bucketed-batch-prepend-state-empty* [*simp*]:
bp-bucketed-batch-prepend-state [] *P* = *P*
unfolding *bp-bucketed-batch-prepend-state-def* **by** *simp*

lemma *bp-bucketed-batch-prepend-state-entries-nonempty*:
assumes *M-pos*: *0* < *bp-block-size P*
and *xs*: *xs* ≠ []
shows *bp-entries* (*bp-bucketed-batch-prepend-state xs P*) =
sort-key snd xs @ *bp-entries P*
proof −
obtain *p ps* **where** *xs-def*: *xs* = *p* # *ps*
using *xs* **by** (*cases xs*) *auto*
have *flat*:
bp-bucket-entries-flat

```
    (bp-bucketize-entries (bp-block-size P) xs) = sort-key snd xs
   by (rule bp-bucket-entries-flat-bucketize-entries[OF M-pos])
  have flat':
   bp-bucket-entries-flat
    (bp-bucketize-entries (bp-block-size P) (p # ps)) =
    sort-key snd (p # ps)
   using flat unfolding xs-def .
  show ?thesis
   unfolding bp-bucketed-batch-prepend-state-def xs-def bp-entries-def
   using flat' by simp
qed

lemma bp-bucketed-batch-prepend-state-keys-nonempty:
  assumes M-pos: 0 < bp-block-size P
   and xs: xs ≠ []
  shows keys-of (bp-view (bp-bucketed-batch-prepend-state xs P)) =
   bp-entry-keys xs ∪ keys-of (bp-view P)
  unfolding bp-view-def
   bp-bucketed-batch-prepend-state-entries-nonempty[OF M-pos xs]
   bp-entry-keys-def
  by auto

lemma bp-bucketed-batch-prepend-state-values:
  assumes distinct: distinct (map fst xs)
   and disjoint: bp-entry-keys xs ∩ keys-of (bp-view P) = {}
  shows value-of (bp-view (bp-bucketed-batch-prepend-state xs P)) =
   value-of (batch-min-update (bp-view P) xs)
proof −
  have disjoint': fst ‘ set xs ∩ keys-of (bp-view P) = {}
   using disjoint unfolding bp-entry-keys-def by simp
  have vals-abs: value-of (batch-min-update (bp-view P) xs) =
     bp-batch-value-update xs (value-of (bp-view P))
   by (rule batch-min-update-value-distinct-disjoint
      [OF distinct disjoint'])
  show ?thesis
   using vals-abs
   unfolding bp-view-def bp-bucketed-batch-prepend-state-def
   by (cases xs) simp-all
qed

theorem bp-bucketed-batch-prepend-state-refines-batch-min-update:
  assumes inv: bp-invariant P
   and distinct: distinct (map fst xs)
   and disjoint: bp-entry-keys xs ∩ keys-of (bp-view P) = {}
  shows bp-view (bp-bucketed-batch-prepend-state xs P) =
   batch-min-update (bp-view P) xs
proof (cases xs = [])
  case True
  then show ?thesis
```

```
    unfolding batch-min-update-def by simp
next
  case False
  have M-pos: 0 < bp-block-size P
    using inv unfolding bp-invariant-def by blast
  have keys: keys-of (bp-view (bp-bucketed-batch-prepend-state xs P)) =
      keys-of (batch-min-update (bp-view P) xs)
  proof −
    have state: keys-of (bp-view (bp-bucketed-batch-prepend-state xs P)) =
        bp-entry-keys xs ∪ keys-of (bp-view P)
      by (rule bp-bucketed-batch-prepend-state-keys-nonempty[OF M-pos False])
    have abstract: keys-of (batch-min-update (bp-view P) xs) =
        keys-of (bp-view P) ∪ bp-entry-keys xs
      by (simp add: batch-min-update-keys bp-entry-keys-def)
    show ?thesis
      using state abstract by auto
  qed
  have vals-state: value-of (bp-view (bp-bucketed-batch-prepend-state xs P)) =
      value-of (batch-min-update (bp-view P) xs)
    by (rule bp-bucketed-batch-prepend-state-values
        [OF distinct disjoint])
  show ?thesis
    using keys vals-state
    by (cases bp-view (bp-bucketed-batch-prepend-state xs P);
        cases batch-min-update (bp-view P) xs; simp)
qed

lemma bp-rebase-first-bucket-marker-markers-lower-bound:
  assumes lower:
    ∀ b∈set bs. ∀ p∈set (bp-bucket-entries b). bp-marker b ≤ snd p
    and beta-le: ∀ p∈set (bp-bucket-entries-flat bs). beta ≤ snd p
  shows ∀ b∈set (bp-rebase-first-bucket-marker beta bs).
    ∀ p∈set (bp-bucket-entries b). bp-marker b ≤ snd p
  using assms
  by (cases bs) (auto simp: bp-bucket-entries-flat-def)

lemma bp-rebase-first-bucket-marker-markers-sorted:
  assumes sorted: sorted-wrt (λb c. bp-marker b ≤ bp-marker c) bs
    and boundaries: bp-bucket-boundaries-ok bs
    and head-nonempty:
      ⋀b bs'. bp-drop-empty-prefix bs = b # bs' ⟹
        bp-bucket-entries b ≠ []
    and beta-le:
      ∀ p∈set (bp-bucket-entries-flat (bp-drop-empty-prefix bs)).
        beta ≤ snd p
  shows sorted-wrt (λb c. bp-marker b ≤ bp-marker c)
    (bp-rebase-first-bucket-marker beta (bp-drop-empty-prefix bs))
proof (cases bp-drop-empty-prefix bs)
  case Nil
```

```
  then show ?thesis by simp
next
  case (Cons b rest)
  note old0 = Cons
  have old0-sorted: sorted-wrt (λb c. bp-marker b ≤ bp-marker c)
      (bp-drop-empty-prefix bs)
    by (rule bp-bucket-markers-sorted-drop-empty-prefix[OF sorted])
  have old0-boundaries:
    bp-bucket-boundaries-ok (bp-drop-empty-prefix bs)
    by (rule bp-bucket-boundaries-ok-drop-empty-prefix[OF boundaries])
  have b-nonempty: bp-bucket-entries b ≠ []
    using head-nonempty Cons by blast
  show ?thesis
  proof (cases rest)
    case Nil
    then show ?thesis
      using old0 by simp
  next
    case (Cons c cs)
    obtain q where q: q ∈ set (bp-bucket-entries b)
      using b-nonempty by (cases bp-bucket-entries b) auto
    have beta-le-q: beta ≤ snd q
      using beta-le old0 q unfolding bp-bucket-entries-flat-def by auto
    have q-le-c: snd q ≤ bp-marker c
      using old0-boundaries old0 Cons q by simp
    have beta-le-c: beta ≤ bp-marker c
      using beta-le-q q-le-c by linarith
    have tail-sorted:
      sorted-wrt (λb c. bp-marker b ≤ bp-marker c) (c # cs)
      using old0-sorted old0 Cons by simp
    have beta-le-tail:
      ∀ d∈set (c # cs). beta ≤ bp-marker d
    proof
      fix d
      assume d: d ∈ set (c # cs)
      have bp-marker c ≤ bp-marker d
        using tail-sorted d by (cases d = c) auto
      then show beta ≤ bp-marker d
        using beta-le-c by linarith
    qed
    show ?thesis
      using old0 Cons tail-sorted beta-le-tail by simp
  qed
qed

lemma bp-batch-max-value-le-old-entry:
  assumes xs: xs = p # ps
    and admissible: batch-prepend-admissible (bp-view P) xs
    and inv: bp-invariant P
```

```
    and r: r ∈ set (bp-entries P)
  shows bp-batch-max-value (snd p) ps ≤ snd r
proof −
  have r-key: fst r ∈ keys-of (bp-view P)
    using r unfolding bp-view-def bp-entry-keys-def by auto
  have r-value: value-of (bp-view P) (fst r) = snd r
    using inv r unfolding bp-invariant-def bp-values-consistent-def
      bp-view-def by simp
  have all-le: ⋀q. q ∈ set xs ⟹ snd q ≤ snd r
  proof −
    fix q
    assume q: q ∈ set xs
    then have snd q ≤ value-of (bp-view P) (fst r)
      using admissible r-key
      unfolding batch-prepend-admissible-def by force
    then show snd q ≤ snd r
      using r-value by simp
  qed
  have ps-le: ⋀q. q ∈ set ps ⟹ snd q ≤ snd r
    using all-le xs by auto
  have p-le: snd p ≤ snd r
    using all-le xs by simp
  show ?thesis
    by (rule bp-batch-max-value-upper[OF ps-le p-le])
qed

lemma bp-bucketed-batch-prepend-state-invariant:
  assumes ord: bp-ordered-invariant P
    and distinct: distinct (map fst xs)
    and disjoint: bp-entry-keys xs ∩ keys-of (bp-view P) = {}
    and admissible: batch-prepend-admissible (bp-view P) xs
  shows bp-invariant (bp-bucketed-batch-prepend-state xs P)
proof (cases xs)
  case Nil
  then show ?thesis
    using bp-ordered-invariant-invariant[OF ord] by simp
next
  case (Cons p ps)
  let ?M = bp-block-size P
  let ?beta = bp-batch-max-value (snd p) ps
  let ?new = bp-bucketize-entries ?M xs
  let ?trimmed = bp-drop-empty-prefix (bp-buckets P)
  let ?old = bp-rebase-first-bucket-marker ?beta ?trimmed
  let ?P' = bp-bucketed-batch-prepend-state xs P
  have inv: bp-invariant P
    by (rule bp-ordered-invariant-invariant[OF ord])
  have boundaries: bp-bucket-boundaries-ok (bp-buckets P)
    using bp-ordered-invariant-boundaries-state-ok[OF ord]
    unfolding bp-bucket-boundaries-state-ok-def .
```

have *M-pos*: *0 < ?M*
using *inv* **unfolding** *bp-invariant-def* **by** *blast*
have *entries-state*: *bp-entries ?P' = sort-key snd xs @ bp-entries P*
by (*rule bp-bucketed-batch-prepend-state-entries-nonempty*[*OF M-pos*])
(*simp add*: *Cons*)
have *old-distinct*: *distinct* (*map fst* (*bp-entries P*))
using *inv* **unfolding** *bp-invariant-def bp-distinct-keys-def* **by** *blast*
have *new-distinct*: *distinct* (*map fst* (*sort-key snd xs*))
by (*rule distinct-map-fst-sort-key*[*OF distinct*])
have *disj-keys*:
set (*map fst* (*sort-key snd xs*)) ∩ *set* (*map fst* (*bp-entries P*)) = {}
using *disjoint* **unfolding** *bp-entry-keys-def bp-view-def* **by** *auto*
have *distinct-entries*:
distinct (*map fst* (*sort-key snd xs* @ *bp-entries P*))
using *new-distinct old-distinct disj-keys* **by** *simp*
have *beta-le-old*:
∀ *r*∈*set* (*bp-entries P*). *?beta* ≤ *snd r*
proof
fix *r*
assume *r*: *r* ∈ *set* (*bp-entries P*)
show *?beta* ≤ *snd r*
by (*rule bp-batch-max-value-le-old-entry*
[*OF Cons admissible inv r*])
qed
have *beta-le-trimmed*:
∀ *r*∈*set* (*bp-bucket-entries-flat ?trimmed*). *?beta* ≤ *snd r*
using *beta-le-old* **unfolding** *bp-entries-def* **by** *simp*
have *new-sizes*:
∀ *b*∈*set ?new*. *length* (*bp-bucket-entries b*) ≤ *?M*
by (*rule bp-bucketize-entries-sizes-ok*[*OF M-pos*])
have *trimmed-subset*: *set ?trimmed* ⊆ *set* (*bp-buckets P*)
by (*rule bp-drop-empty-prefix-set-subset*)
have *trimmed-sizes*:
∀ *b*∈*set ?trimmed*. *length* (*bp-bucket-entries b*) ≤ *?M*
using *inv trimmed-subset* **unfolding** *bp-invariant-def bp-bucket-sizes-ok-def*
by *blast*
have *old-sizes*:
∀ *b*∈*set ?old*. *length* (*bp-bucket-entries b*) ≤ *?M*
proof (*cases ?trimmed*)
case *Nil*
then show *?thesis* **by** *simp*
next
case (*Cons b bs*)
then show *?thesis*
using *trimmed-sizes* **by** *simp*
qed
have *new-sorted*:
sorted-wrt (λ*b c*. *bp-marker b* ≤ *bp-marker c*) *?new*
by (*rule bp-bucketize-entries-markers-sorted*[*OF M-pos*])

have *old-sorted*:
sorted-wrt (λ*b c*. *bp-marker b* ≤ *bp-marker c*) *?old*
proof (*rule bp-rebase-first-bucket-marker-markers-sorted*)
show *sorted-wrt* (λ*b c*. *bp-marker b* ≤ *bp-marker c*) (*bp-buckets P*)
using *inv* **unfolding** *bp-invariant-def bp-bucket-markers-sorted-def*
by *blast*
show *bp-bucket-boundaries-ok* (*bp-buckets P*)
by (*rule boundaries*)
show ⋀*b bs′*. *bp-drop-empty-prefix* (*bp-buckets P*) = *b* # *bs′* ⟹
bp-bucket-entries b ≠ []
by (*rule bp-drop-empty-prefix-head-nonempty*)
show ∀ *r*∈*set* (*bp-bucket-entries-flat*
(*bp-drop-empty-prefix* (*bp-buckets P*))). *?beta* ≤ *snd r*
by (*rule beta-le-trimmed*)
qed
have *new-marker-le-beta*:
∀ *b*∈*set ?new*. *bp-marker b* ≤ *?beta*
proof (*rule bp-bucketize-entries-markers-le*)
show ∀ *q*∈*set xs*. *snd q* ≤ *?beta*
using *Cons*
by (*auto simp*: *bp-batch-max-value-ge-member-Cons*)
qed
have *beta-le-old-markers*:
∀ *b*∈*set ?old*. *?beta* ≤ *bp-marker b*
proof (*cases ?old*)
case *Nil*
then show *?thesis* **by** *simp*
next
case (*Cons b bs*)
have *head*: *bp-marker b* = *?beta*
using *Cons* **by** (*cases ?trimmed*) *auto*
have *tail-ge*: ∀ *c*∈*set bs*. *bp-marker b* ≤ *bp-marker c*
using *old-sorted Cons* **by** *simp*
show *?thesis*
using *Cons head tail-ge* **by** *auto*
qed
have *sorted-all*:
sorted-wrt (λ*b c*. *bp-marker b* ≤ *bp-marker c*) (*?new* @ *?old*)
unfolding *sorted-wrt-append*
proof (*intro conjI ballI*)
show *sorted-wrt* (λ*b c*. *bp-marker b* ≤ *bp-marker c*) *?new*
by (*rule new-sorted*)
show *sorted-wrt* (λ*b c*. *bp-marker b* ≤ *bp-marker c*) *?old*
by (*rule old-sorted*)
fix *b c*
assume *b*: *b* ∈ *set ?new* **and** *c*: *c* ∈ *set ?old*
have *bp-marker b* ≤ *?beta*
using *new-marker-le-beta b* **by** *blast*
moreover have *?beta* ≤ *bp-marker c*

```
    using beta-le-old-markers c by blast
  ultimately show bp-marker b ≤ bp-marker c
    by linarith
qed
have new-lower:
  ∀ b∈set ?new. ∀ q∈set (bp-bucket-entries b).
    bp-marker b ≤ snd q
  by (rule bp-bucketize-entries-markers-lower-bound[OF M-pos])
have trimmed-lower:
  ∀ b∈set ?trimmed. ∀ q∈set (bp-bucket-entries b).
    bp-marker b ≤ snd q
  using inv trimmed-subset
  unfolding bp-invariant-def bp-bucket-markers-lower-bound-def by blast
have old-lower:
  ∀ b∈set ?old. ∀ q∈set (bp-bucket-entries b).
    bp-marker b ≤ snd q
  by (rule bp-rebase-first-bucket-marker-markers-lower-bound
      [OF trimmed-lower beta-le-trimmed])
have lower-all:
  ∀ b∈set (?new @ ?old). ∀ q∈set (bp-bucket-entries b).
    bp-marker b ≤ snd q
  using new-lower old-lower by auto
have values-entries:
  ∀ r∈set (sort-key snd xs @ bp-entries P).
    bp-batch-value-update xs (bp-values P) (fst r) = snd r
proof
  fix r
  assume r: r ∈ set (sort-key snd xs @ bp-entries P)
  show bp-batch-value-update xs (bp-values P) (fst r) = snd r
  proof (cases r ∈ set (sort-key snd xs))
    case True
    then have r ∈ set xs
      by simp
    obtain x b where r-def: r = (x, b)
      by force
    then have (x, b) ∈ set xs
      using ‹r ∈ set xs› by simp
    then have bp-batch-value-update xs (bp-values P) x = b
      by (rule bp-batch-value-update-in-set-distinct[OF distinct])
    then show ?thesis
      unfolding r-def by simp
  next
    case False
    then have r-old: r ∈ set (bp-entries P)
      using r by simp
    have fst r ∉ fst ‘ set xs
      using disjoint r-old unfolding bp-entry-keys-def bp-view-def
      by auto
    then have unchanged:
```

```
      bp-batch-value-update xs (bp-values P) (fst r) =
        bp-values P (fst r)
      by (rule bp-batch-value-update-notin)
    have bp-values P (fst r) = snd r
      using inv r-old unfolding bp-invariant-def bp-values-consistent-def
      by blast
    then show ?thesis
      using unchanged by simp
  qed
 qed
 have flat-new: bp-bucket-entries-flat ?new = sort-key snd xs
  by (rule bp-bucket-entries-flat-bucketize-entries[OF M-pos])
 have flat-old: bp-bucket-entries-flat ?old = bp-entries P
  unfolding bp-entries-def by simp
 show ?thesis
  using M-pos distinct-entries new-sizes old-sizes sorted-all lower-all
    values-entries flat-new flat-old
  unfolding bp-bucketed-batch-prepend-state-def Cons Let-def
    bp-invariant-def bp-distinct-keys-def bp-bucket-sizes-ok-def
    bp-bucket-markers-sorted-def bp-bucket-markers-lower-bound-def
    bp-values-consistent-def bp-entries-def
  by auto
qed

lemma bp-bucketed-batch-prepend-state-boundaries-state-ok:
 assumes ord: bp-ordered-invariant P
  and admissible: batch-prepend-admissible (bp-view P) xs
 shows bp-bucket-boundaries-state-ok
  (bp-bucketed-batch-prepend-state xs P)
proof (cases xs)
 case Nil
 then show ?thesis
  using bp-ordered-invariant-boundaries-state-ok[OF ord] by simp
next
 case (Cons p ps)
 let ?M = bp-block-size P
 let ?beta = bp-batch-max-value (snd p) ps
 let ?new = bp-bucketize-entries ?M xs
 let ?trimmed = bp-drop-empty-prefix (bp-buckets P)
 let ?old = bp-rebase-first-bucket-marker ?beta ?trimmed
 have inv: bp-invariant P
  by (rule bp-ordered-invariant-invariant[OF ord])
 have M-pos: 0 < ?M
  using inv unfolding bp-invariant-def by blast
 have trimmed-boundaries:
  bp-bucket-boundaries-ok ?trimmed
  by (rule bp-bucket-boundaries-ok-drop-empty-prefix)
   (use bp-ordered-invariant-boundaries-state-ok[OF ord] in
     ‹simp add: bp-bucket-boundaries-state-ok-def›)
```

have *old-boundaries*: *bp-bucket-boundaries-ok ?old*
proof (*cases ?trimmed*)
  case *Nil*
  then show *?thesis* **by** *simp*
next
  case (*Cons b bs*)
  then show *?thesis*
  proof (*cases bs*)
    case *Nil*
    then show *?thesis*
      using *Cons* **by** *simp*
  next
    case (*Cons c cs*)
    then show *?thesis*
      using *Cons trimmed-boundaries* **by** *simp*
  qed
qed
have *new-boundaries*: *bp-bucket-boundaries-ok ?new*
  by (*rule bp-bucketize-entries-boundaries-ok*[*OF M-pos*])
have *new-entries-le-beta*:
  ∀ *b*∈*set ?new*. ∀ *q*∈*set* (*bp-bucket-entries b*).
    snd q ≤ *?beta*
proof (*intro ballI*)
  fix *b q*
  assume *b*: *b* ∈ *set ?new*
    and *q*: *q* ∈ *set* (*bp-bucket-entries b*)
  have *q* ∈ *set xs*
    by (*rule bp-bucketize-entries-entry-in-source*[*OF M-pos b q*])
  then show *snd q* ≤ *?beta*
    using *Cons* **by** (*auto simp*: *bp-batch-max-value-ge-member-Cons*)
qed
have *cross*:
  ?old ≠ [] ⟹
    ∀ *b*∈*set ?new*. ∀ *q*∈*set* (*bp-bucket-entries b*).
      snd q ≤ *bp-marker* (*hd ?old*)
proof −
  assume *old-nonempty*: *?old* ≠ []
  have *bp-marker* (*hd ?old*) = *?beta*
    using *old-nonempty* **by** (*cases ?trimmed*) *auto*
  then show ∀ *b*∈*set ?new*. ∀ *q*∈*set* (*bp-bucket-entries b*).
      snd q ≤ *bp-marker* (*hd ?old*)
    using *new-entries-le-beta* **by** *simp*
qed
have *appended*: *bp-bucket-boundaries-ok* (*?new* @ *?old*)
  by (*rule bp-bucket-boundaries-ok-append*
      [*OF new-boundaries old-boundaries cross*])
show *?thesis*
  using *appended Cons*
  unfolding *bp-bucketed-batch-prepend-state-def Let-def*

bp-bucket-boundaries-state-ok-def
by *simp*
qed

lemma *bp-bucketed-batch-prepend-state-ordered-invariant*:
assumes *ord*: *bp-ordered-invariant P*
and *distinct*: *distinct* (*map fst xs*)
and *disjoint*: *bp-entry-keys xs* ∩ *keys-of* (*bp-view P*) = {}
and *admissible*: *batch-prepend-admissible* (*bp-view P*) *xs*
shows *bp-ordered-invariant* (*bp-bucketed-batch-prepend-state xs P*)
unfolding *bp-ordered-invariant-def*
using *bp-bucketed-batch-prepend-state-invariant*
[*OF ord distinct disjoint admissible*]
bp-bucketed-batch-prepend-state-boundaries-state-ok
[*OF ord admissible*]
by *blast*

Bucketed BatchPrepend is not implemented as repeated Insert. The incoming batch is first bucketized on its own, then placed before the existing buckets. To preserve the boundary condition, the old prefix of empty buckets is dropped and the first surviving old bucket is rebased to the maximum value of the batch. The admissibility assumption is the abstract promise that every old entry is above the batch range. Together with distinctness and disjointness of keys, this yields both the view refinement ⟦*bp-invariant ?P*; *distinct* (*map fst ?xs*); *bp-entry-keys ?xs* ∩ *keys-of* (*bp-view ?P*) = {}⟧ ⟹ *bp-view* (*bp-bucketed-batch-prepend-state ?xs ?P*) = *batch-min-update* (*bp-view ?P*) *?xs* and the ordered invariant theorem above.

lemma *c-bp-bucketed-batch-prepend-direct-invariant*:
assumes *ord*: *bp-ordered-invariant P*
and *distinct*: *distinct* (*map fst xs*)
and *disjoint*: *bp-entry-keys xs* ∩ *keys-of* (*bp-view P*) = {}
and *admissible*: *batch-prepend-admissible* (*bp-view P*) *xs*
shows *bp-invariant*
(*bp-result-of* (*c-bp-bucketed-batch-prepend-direct xs P*))
unfolding *c-bp-bucketed-batch-prepend-direct-result*
by (*rule bp-bucketed-batch-prepend-state-invariant*
[*OF ord distinct disjoint admissible*])

lemma *c-bp-bucketed-batch-prepend-direct-ordered-invariant*:
assumes *ord*: *bp-ordered-invariant P*
and *distinct*: *distinct* (*map fst xs*)
and *disjoint*: *bp-entry-keys xs* ∩ *keys-of* (*bp-view P*) = {}
and *admissible*: *batch-prepend-admissible* (*bp-view P*) *xs*
shows *bp-ordered-invariant*
(*bp-result-of* (*c-bp-bucketed-batch-prepend-direct xs P*))
unfolding *c-bp-bucketed-batch-prepend-direct-result*
by (*rule bp-bucketed-batch-prepend-state-ordered-invariant*
[*OF ord distinct disjoint admissible*])

lemma *c-bp-bucketed-batch-prepend-direct-refines-batch-min-update*:
assumes *ord*: *bp-ordered-invariant P*
and *distinct*: *distinct* (*map fst xs*)
and *disjoint*: *bp-entry-keys xs* ∩ *keys-of* (*bp-view P*) = {}
shows *bp-view* (*bp-result-of* (*c-bp-bucketed-batch-prepend-direct xs P*)) =
batch-min-update (*bp-view P*) *xs*
unfolding *c-bp-bucketed-batch-prepend-direct-result*
by (*rule bp-bucketed-batch-prepend-state-refines-batch-min-update*
[*OF bp-ordered-invariant-invariant*[*OF ord*] *distinct disjoint*])

lemma *c-bp-bucketed-batch-prepend-direct-preserves-upper-bound*:
assumes *ord*: *bp-ordered-invariant P*
and *distinct*: *distinct* (*map fst xs*)
and *disjoint*: *bp-entry-keys xs* ∩ *keys-of* (*bp-view P*) = {}
and *upper*: *partition-upper-bound* (*bp-view P*) *B*
and *values-lt*: ⋀*x b*. (*x*, *b*) ∈ *set xs* ⟹ *b* < *B*
shows *partition-upper-bound*
(*bp-view* (*bp-result-of* (*c-bp-bucketed-batch-prepend-direct xs P*))) *B*
unfolding *c-bp-bucketed-batch-prepend-direct-refines-batch-min-update*
[*OF ord distinct disjoint*]
by (*rule batch-min-update-preserves-upper-bound*[*OF upper values-lt*])

lemma *c-bp-bucketed-batch-prepend-direct-batch-cost-bound*:
partition-batch-cost-bound
(*bp-steps-of* (*c-bp-bucketed-batch-prepend-direct xs P*))
(*bp-batch-prepend-per-item-budget xs* (*bp-block-size P*)) *xs*
by (*rule c-bp-bucketed-batch-prepend-direct-paper-batch-cost-bound*)

lemma *bp-bucketed-batch-prepend-state-buckets-length-nonempty*:
assumes *M-pos*: *0* < *bp-block-size P*
and *xs*: *xs* ≠ []
shows *length* (*bp-buckets* (*bp-bucketed-batch-prepend-state xs P*)) ≤
length xs + *length* (*bp-buckets P*)
proof −
obtain *p ps* **where** *xs-def*: *xs* = *p* # *ps*
using *xs* **by** (*cases xs*) *auto*
have *new*:
length (*bp-bucketize-entries* (*bp-block-size P*) *xs*) ≤ *length xs*
by (*rule length-bp-bucketize-entries-le-length*[*OF M-pos*])
have *new′*:
length (*bp-bucketize-entries* (*bp-block-size P*) (*p* # *ps*)) ≤
Suc (*length ps*)
using *new* **unfolding** *xs-def* **by** *simp*
have *old*:
length (*bp-rebase-first-bucket-marker*
(*bp-batch-max-value* (*snd p*) *ps*)
(*bp-drop-empty-prefix* (*bp-buckets P*))) ≤
length (*bp-buckets P*)

```
    by simp
  have old':
    length (bp-drop-empty-prefix (bp-buckets P)) ≤
      length (bp-buckets P)
    by simp
  have total:
    length (bp-bucketize-entries (bp-block-size P) (p # ps)) +
      length (bp-drop-empty-prefix (bp-buckets P)) ≤
      Suc (length ps) + length (bp-buckets P)
    using new' old' by linarith
  show ?thesis
    unfolding bp-bucketed-batch-prepend-state-def xs-def Let-def
    using total by simp
qed

lemma bp-bucketed-batch-prepend-state-bucket-count-slack-le:
  assumes inv: bp-invariant P
  shows bp-bucket-count-slack (bp-bucketed-batch-prepend-state xs P) ≤
    length xs + bp-bucket-count-slack P
proof (cases xs = [])
  case True
  then show ?thesis by simp
next
  case False
  let ?P' = bp-bucketed-batch-prepend-state xs P
  let ?M = bp-block-size P
  let ?N = length (bp-entries P)
  let ?N' = length (bp-entries ?P')
  let ?K = length (bp-buckets P)
  let ?K' = length (bp-buckets ?P')
  let ?L = length xs
  have M-pos: 0 < ?M
    using inv unfolding bp-invariant-def by blast
  have entries-len: ?N' = ?L + ?N
    using bp-bucketed-batch-prepend-state-entries-nonempty[OF M-pos False]
    by simp
  have buckets-len: ?K' ≤ ?L + ?K
    by (rule bp-bucketed-batch-prepend-state-buckets-length-nonempty
        [OF M-pos False])
  have div-le: Suc (?N div ?M) ≤ Suc (?N' div ?M)
    unfolding entries-len by (simp add: div-le-mono)
  have block: bp-block-size ?P' = ?M
    unfolding bp-bucketed-batch-prepend-state-def
    by (cases xs) (simp-all add: Let-def)
  have diff:
    ?K' − Suc (?N' div ?M) ≤
      ?L + (?K − Suc (?N div ?M))
    by (rule nat-diff-add-right-le[OF buckets-len div-le])
  show ?thesis
```

```
    using diff
    unfolding bp-bucket-count-slack-def entries-len block
    by simp
qed

lemma bp-bucketed-batch-prepend-state-split-potential-le:
  assumes ord: bp-ordered-invariant P
    and distinct: distinct (map fst xs)
    and disjoint: bp-entry-keys xs ∩ keys-of (bp-view P) = {}
    and admissible: batch-prepend-admissible (bp-view P) xs
  shows bp-split-potential (bp-bucketed-batch-prepend-state xs P) ≤
    length xs + bp-split-potential P
proof −
  have inv: bp-invariant P
    by (rule bp-ordered-invariant-invariant[OF ord])
  have inv':
    bp-invariant (bp-bucketed-batch-prepend-state xs P)
    by (rule bp-bucketed-batch-prepend-state-invariant
        [OF ord distinct disjoint admissible])
  have overflow':
    bp-overflow-potential (bp-bucketed-batch-prepend-state xs P) = 0
    by (rule bp-invariant-overflow-potential-zero[OF inv'])
  have slack:
    bp-bucket-count-slack (bp-bucketed-batch-prepend-state xs P) ≤
      length xs + bp-bucket-count-slack P
    by (rule bp-bucketed-batch-prepend-state-bucket-count-slack-le[OF inv])
  have bp-split-potential (bp-bucketed-batch-prepend-state xs P) =
      bp-bucket-count-slack (bp-bucketed-batch-prepend-state xs P)
    unfolding bp-split-potential-def using overflow' by simp
  also have ... ≤ length xs + bp-bucket-count-slack P
    by (rule slack)
  also have ... ≤ length xs + bp-split-potential P
    unfolding bp-split-potential-def by simp
  finally show ?thesis .
qed

lemma c-bp-bucketed-batch-prepend-direct-actual-refines-batch-min-update:
  assumes ord: bp-ordered-invariant P
    and distinct: distinct (map fst xs)
    and disjoint: bp-entry-keys xs ∩ keys-of (bp-view P) = {}
  shows bp-view
    (bp-result-of (c-bp-bucketed-batch-prepend-direct-actual xs P)) =
    batch-min-update (bp-view P) xs
  unfolding c-bp-bucketed-batch-prepend-direct-actual-result
  by (rule bp-bucketed-batch-prepend-state-refines-batch-min-update
      [OF bp-ordered-invariant-invariant[OF ord] distinct disjoint])

lemma c-bp-bucketed-batch-prepend-direct-actual-invariant:
  assumes ord: bp-ordered-invariant P
```

```
    and distinct: distinct (map fst xs)
    and disjoint: bp-entry-keys xs ∩ keys-of (bp-view P) = {}
    and admissible: batch-prepend-admissible (bp-view P) xs
  shows bp-invariant
    (bp-result-of (c-bp-bucketed-batch-prepend-direct-actual xs P))
  unfolding c-bp-bucketed-batch-prepend-direct-actual-result
  by (rule bp-bucketed-batch-prepend-state-invariant
      [OF ord distinct disjoint admissible])

lemma c-bp-bucketed-batch-prepend-direct-actual-ordered-invariant:
  assumes ord: bp-ordered-invariant P
    and distinct: distinct (map fst xs)
    and disjoint: bp-entry-keys xs ∩ keys-of (bp-view P) = {}
    and admissible: batch-prepend-admissible (bp-view P) xs
  shows bp-ordered-invariant
    (bp-result-of (c-bp-bucketed-batch-prepend-direct-actual xs P))
  unfolding c-bp-bucketed-batch-prepend-direct-actual-result
  by (rule bp-bucketed-batch-prepend-state-ordered-invariant
      [OF ord distinct disjoint admissible])

lemma c-bp-bucketed-batch-prepend-direct-actual-preserves-upper-bound:
  assumes ord: bp-ordered-invariant P
    and distinct: distinct (map fst xs)
    and disjoint: bp-entry-keys xs ∩ keys-of (bp-view P) = {}
    and upper: partition-upper-bound (bp-view P) B
    and values-lt: ⋀x b. (x, b) ∈ set xs ⟹ b < B
  shows partition-upper-bound
    (bp-view
      (bp-result-of (c-bp-bucketed-batch-prepend-direct-actual xs P))) B
  unfolding c-bp-bucketed-batch-prepend-direct-actual-refines-batch-min-update
    [OF ord distinct disjoint]
  by (rule batch-min-update-preserves-upper-bound[OF upper values-lt])

lemma c-bp-bucketed-batch-prepend-direct-actual-amortized-paper-bound:
  assumes ord: bp-ordered-invariant P
    and distinct: distinct (map fst xs)
    and disjoint: bp-entry-keys xs ∩ keys-of (bp-view P) = {}
    and admissible: batch-prepend-admissible (bp-view P) xs
  shows bp-amortized-step-bound
    (bp-steps-of (c-bp-bucketed-batch-prepend-direct-actual xs P)) P
    (bp-result-of (c-bp-bucketed-batch-prepend-direct-actual xs P))
    (bp-batch-prepend-amortized-budget xs (bp-block-size P))
proof −
  have pot:
    bp-split-potential
        (bp-bucketed-batch-prepend-state xs P) ≤
      length xs + bp-split-potential P
    by (rule bp-bucketed-batch-prepend-state-split-potential-le
        [OF ord distinct disjoint admissible])
```

```
  show ?thesis
    using pot
    unfolding bp-amortized-step-bound-def
      bp-batch-prepend-amortized-budget-def
    by simp
qed

lemma c-bp-bucketed-batch-prepend-direct-actual-batch-cost-bound:
  partition-batch-cost-bound
    (bp-steps-of (c-bp-bucketed-batch-prepend-direct-actual xs P))
    (bp-ratio-log-budget (length xs) (bp-block-size P)) xs
  unfolding partition-batch-cost-bound-def
    c-bp-bucketed-batch-prepend-direct-actual-steps
    bp-batch-prepend-log-budget-def
  by (simp add: mult.commute)

lemma c-bp-rebucketed-batch-prepend-bulk-regular-invariant:
  assumes reg: bp-regular-invariant P
  shows bp-regular-invariant
    (bp-result-of (c-bp-rebucketed-batch-prepend-bulk xs P))
  unfolding c-bp-rebucketed-batch-prepend-bulk-result
  by (rule bp-rebucketed-batch-prepend-regular-invariant
      [OF bp-ordered-invariant-invariant
        [OF bp-regular-invariant-ordered-invariant[OF reg]]])

lemma c-bp-rebucketed-batch-prepend-bulk-refines-batch-min-update:
  assumes reg: bp-regular-invariant P
  shows bp-view (bp-result-of (c-bp-rebucketed-batch-prepend-bulk xs P)) =
    batch-min-update (bp-view P) xs
  unfolding c-bp-rebucketed-batch-prepend-bulk-result
  by (rule bp-rebucketed-batch-prepend-refines-batch-min-update
      [OF bp-ordered-invariant-invariant
        [OF bp-regular-invariant-ordered-invariant[OF reg]]])

lemma c-bp-rebucketed-batch-prepend-bulk-preserves-upper-bound:
  assumes reg: bp-regular-invariant P
    and upper: partition-upper-bound (bp-view P) B
    and values-lt: ⋀x b. (x, b) ∈ set xs ⟹ b < B
  shows partition-upper-bound
    (bp-view (bp-result-of (c-bp-rebucketed-batch-prepend-bulk xs P))) B
  unfolding c-bp-rebucketed-batch-prepend-bulk-result
  by (rule bp-rebucketed-batch-prepend-preserves-upper-bound
      [OF bp-ordered-invariant-invariant
        [OF bp-regular-invariant-ordered-invariant[OF reg]]
        upper values-lt])

lemma c-bp-rebucketed-batch-prepend-bulk-split-potential-zero:
  assumes reg: bp-regular-invariant P
  shows bp-split-potential
```

```
    (bp-result-of (c-bp-rebucketed-batch-prepend-bulk xs P)) = 0
  unfolding c-bp-rebucketed-batch-prepend-bulk-result
  by (rule bp-rebucketed-batch-prepend-split-potential-zero
      [OF bp-ordered-invariant-invariant
        [OF bp-regular-invariant-ordered-invariant[OF reg]]])

lemma c-bp-rebucketed-batch-prepend-bulk-amortized-bound:
  assumes reg: bp-regular-invariant P
  shows bp-amortized-step-bound
    (bp-steps-of (c-bp-rebucketed-batch-prepend-bulk xs P)) P
    (bp-result-of (c-bp-rebucketed-batch-prepend-bulk xs P))
    (bp-batch-prepend-log-budget xs (bp-block-size P))
  using c-bp-rebucketed-batch-prepend-bulk-split-potential-zero[OF reg, of xs]
    bp-regular-invariant-split-potential-zero[OF reg]
  unfolding bp-amortized-step-bound-def by simp

lemma c-bp-rebucketed-batch-prepend-bulk-steps-le-amortized-budget:
  bp-steps-of (c-bp-rebucketed-batch-prepend-bulk xs P) ≤
    bp-batch-prepend-amortized-budget xs (bp-block-size P)
  using bp-batch-prepend-log-budget-le-amortized-budget[of xs bp-block-size P]
  by simp

lemma c-bp-rebucketed-batch-prepend-bulk-amortized-paper-bound:
  assumes reg: bp-regular-invariant P
  shows bp-amortized-step-bound
    (bp-steps-of (c-bp-rebucketed-batch-prepend-bulk xs P)) P
    (bp-result-of (c-bp-rebucketed-batch-prepend-bulk xs P))
    (bp-batch-prepend-amortized-budget xs (bp-block-size P))
proof −
  have base: bp-amortized-step-bound
      (bp-steps-of (c-bp-rebucketed-batch-prepend-bulk xs P)) P
      (bp-result-of (c-bp-rebucketed-batch-prepend-bulk xs P))
      (bp-batch-prepend-log-budget xs (bp-block-size P))
    by (rule c-bp-rebucketed-batch-prepend-bulk-amortized-bound[OF reg])
  have budget:
    bp-batch-prepend-log-budget xs (bp-block-size P) ≤
      bp-batch-prepend-amortized-budget xs (bp-block-size P)
    by (rule bp-batch-prepend-log-budget-le-amortized-budget)
  show ?thesis
    using base budget unfolding bp-amortized-step-bound-def by linarith
qed

lemma c-bp-rebucketed-batch-prepend-bulk-batch-cost-bound:
  partition-batch-cost-bound
    (bp-steps-of (c-bp-rebucketed-batch-prepend-bulk xs P))
    (bp-ratio-log-budget (length xs) (bp-block-size P)) xs
  unfolding partition-batch-cost-bound-def
  by (simp add: c-bp-rebucketed-batch-prepend-bulk-steps
      bp-batch-prepend-log-budget-def mult.commute)
```

lemma *c-bp-rebucketed-batch-prepend-bulk-paper-batch-cost-bound*:
partition-batch-cost-bound
(*bp-steps-of* (*c-bp-rebucketed-batch-prepend-bulk xs P*))
(*bp-batch-prepend-per-item-budget xs* (*bp-block-size P*)) *xs*
proof −
have *steps*:
bp-steps-of (*c-bp-rebucketed-batch-prepend-bulk xs P*) ≤
bp-batch-prepend-amortized-budget xs (*bp-block-size P*)
by (*rule c-bp-rebucketed-batch-prepend-bulk-steps-le-amortized-budget*)
show *?thesis*
using *steps*
unfolding *partition-batch-cost-bound-def*
bp-batch-prepend-amortized-budget-alt
by *simp*
qed

lemma *c-bp-first-bucket-pull-result* [*simp*]:
bp-result-of (*c-bp-first-bucket-pull M B P*) =
bp-first-bucket-pull M B P
unfolding *bp-result-of-def c-bp-first-bucket-pull-def* **by** *simp*

lemma *c-bp-first-bucket-pull-steps* [*simp*]:
bp-steps-of (*c-bp-first-bucket-pull M B P*) = *M*
unfolding *bp-steps-of-def c-bp-first-bucket-pull-def* **by** *simp*

lemma *c-bp-first-bucket-pull-M-bound*:
bp-steps-of (*c-bp-first-bucket-pull M B P*) ≤ *M*
by *simp*

lemma *c-bp-first-bucket-pull-scan-result* [*simp*]:
bp-result-of (*c-bp-first-bucket-pull-scan M B P*) =
bp-first-bucket-pull M B P
unfolding *bp-result-of-def c-bp-first-bucket-pull-scan-def*
by (*auto split*: *prod.splits*)

lemma *c-bp-first-bucket-pull-scan-steps*:
assumes *bp-first-bucket-pull M B P* = (*S*, *beta*, *P'*)
shows *bp-steps-of* (*c-bp-first-bucket-pull-scan M B P*) = *card S*
using *assms* **unfolding** *bp-steps-of-def c-bp-first-bucket-pull-scan-def*
by *simp*

lemma *bp-bucket-keys-card-le-length*:
card (*bp-bucket-keys b*) ≤ *length* (*bp-bucket-entries b*)
unfolding *bp-bucket-keys-def bp-entry-keys-def*
by (*metis card-length length-map set-map*)

lemma *c-bp-first-bucket-pull-scan-M-bound*:
assumes *can*: *bp-can-first-bucket-pull M P*

shows *bp-steps-of* (*c-bp-first-bucket-pull-scan M B P*) ≤ *M*
proof −
obtain *b c bs* **where** *buckets*: *bp-buckets P* = *b* # *c* # *bs*
and *len-b*: *length* (*bp-bucket-entries b*) ≤ *M*
by (*rule bp-can-first-bucket-pullE*[*OF can*])
have *pull*:
bp-first-bucket-pull M B P =
(*bp-bucket-keys b*, *bp-marker c*, *bp-delete-keys* (*bp-bucket-keys b*) *P*)
unfolding *bp-first-bucket-pull-def buckets* **by** *simp*
have *bp-steps-of* (*c-bp-first-bucket-pull-scan M B P*) =
card (*bp-bucket-keys b*)
by (*rule c-bp-first-bucket-pull-scan-steps*[*OF pull*])
also have ... ≤ *length* (*bp-bucket-entries b*)
by (*rule bp-bucket-keys-card-le-length*)
also have ... ≤ *M*
by (*rule len-b*)
finally show *?thesis* **.**
qed

lemma *c-bp-first-bucket-pull-scan-partition-pull-cost-bound*:
assumes *pull*: *bp-first-bucket-pull M B P* = (*S*, *beta*, *P*′)
shows *partition-pull-cost-bound*
(*bp-steps-of* (*c-bp-first-bucket-pull-scan M B P*)) *S*
using *c-bp-first-bucket-pull-scan-steps*[*OF pull*]
unfolding *partition-pull-cost-bound-def* **by** *simp*

lemma *c-bp-first-bucket-pull-scan-partition-pull-cost-bound-from-result*:
assumes *pull*:
bp-result-of (*c-bp-first-bucket-pull-scan M B P*) = (*S*, *beta*, *P*′)
shows *partition-pull-cost-bound*
(*bp-steps-of* (*c-bp-first-bucket-pull-scan M B P*)) *S*
proof −
have *pull*′: *bp-first-bucket-pull M B P* = (*S*, *beta*, *P*′)
using *pull* **by** *simp*
show *?thesis*
by (*rule c-bp-first-bucket-pull-scan-partition-pull-cost-bound*[*OF pull*′])
qed

lemma *c-bp-first-bucket-pull-scan-invariant*:
assumes *inv*: *bp-invariant P*
shows *bp-invariant*
(*bp-pull-state-of* (*bp-result-of* (*c-bp-first-bucket-pull-scan M B P*)))
proof −
obtain *S beta P*′ **where** *pull*: *bp-first-bucket-pull M B P* = (*S*, *beta*, *P*′)
by (*cases bp-first-bucket-pull M B P*)
have *bp-invariant P*′
by (*rule bp-first-bucket-pull-invariant*[*OF inv pull*])
then show *?thesis*
using *pull* **unfolding** *bp-pull-state-of-def* **by** *simp*

qed

lemma *c-bp-first-bucket-pull-scan-amortized-M-bound*:
assumes *inv*: *bp-invariant P*
and *can*: *bp-can-first-bucket-pull M P*
shows *bp-pull-amortized-step-bound*
(*bp-steps-of* (*c-bp-first-bucket-pull-scan M B P*)) *P*
(*bp-result-of* (*c-bp-first-bucket-pull-scan M B P*)) (*2* ∗ *M*)
proof −
obtain *b c bs* **where** *buckets*: *bp-buckets P* = *b* # *c* # *bs*
and *len-b*: *length* (*bp-bucket-entries b*) ≤ *M*
by (*rule bp-can-first-bucket-pullE*[*OF can*])
let *?S* = *bp-bucket-keys b*
have *pull*:
bp-first-bucket-pull M B P =
(*?S*, *bp-marker c*, *bp-delete-keys ?S P*)
unfolding *bp-first-bucket-pull-def buckets* **by** *simp*
have *state*:
bp-pull-state-of
(*bp-result-of* (*c-bp-first-bucket-pull-scan M B P*)) =
bp-delete-keys ?S P
unfolding *bp-pull-state-of-def* **using** *pull* **by** *simp*
have *steps*: *bp-steps-of* (*c-bp-first-bucket-pull-scan M B P*) ≤ *M*
by (*rule c-bp-first-bucket-pull-scan-M-bound*[*OF can*])
have *card-S*: *card ?S* ≤ *M*
proof −
have *card ?S* ≤ *length* (*bp-bucket-entries b*)
by (*rule bp-bucket-keys-card-le-length*)
also have . . . ≤ *M*
by (*rule len-b*)
finally show *?thesis* **.**
qed
have *finite-S*: *finite ?S*
unfolding *bp-bucket-keys-def bp-entry-keys-def* **by** *simp*
have *potential*:
bp-split-potential (*bp-delete-keys ?S P*) ≤
bp-split-potential P + *card ?S*
proof (*rule bp-delete-keys-split-potential-le*)
show *0* < *bp-block-size P*
using *inv* **unfolding** *bp-invariant-def* **by** *blast*
show *bp-distinct-keys P*
using *inv* **unfolding** *bp-invariant-def* **by** *blast*
show *finite ?S*
by (*rule finite-S*)
qed
show *?thesis*
using *steps card-S potential* **unfolding** *bp-pull-amortized-step-bound-def*
state
by *linarith*

```
qed

lemma c-bp-first-bucket-pull-scan-amortized-paper-bound:
  assumes inv: bp-invariant P
    and can: bp-can-first-bucket-pull M P
  shows bp-pull-amortized-step-bound
    (bp-steps-of (c-bp-first-bucket-pull-scan M B P)) P
    (bp-result-of (c-bp-first-bucket-pull-scan M B P))
    (bp-pull-amortized-budget M)
  unfolding bp-pull-amortized-budget-def
  by (rule c-bp-first-bucket-pull-scan-amortized-M-bound[OF inv can])

lemma c-bp-first-bucket-pull-scan-refines-pull-separates:
  assumes inv: bp-invariant P
    and can: bp-can-first-bucket-pull M P
    and pull:
      bp-result-of (c-bp-first-bucket-pull-scan M B P) = (S, beta, P')
  shows pull-separates (bp-view P) M B S beta (bp-view P')
proof −
  obtain b c bs where buckets: bp-buckets P = b # c # bs
    and len-b: length (bp-bucket-entries b) ≤ M
    and below: bp-bucket-below-bound b (bp-marker c)
    and tail-nonempty: bp-bucket-entries-flat (c # bs) ≠ []
    by (rule bp-can-first-bucket-pullE[OF can])
  have pull': bp-first-bucket-pull M B P = (S, beta, P')
    using pull by simp
  show ?thesis
    by (rule bp-first-bucket-pull-refines-pull-separates
        [OF inv buckets len-b below tail-nonempty pull'])
qed
```

Pull is charged by the size of the set it returns. The first-bucket policy is sound when *bp-can-first-bucket-pull* exposes two adjacent nonempty ranges and the first bucket lies entirely below the next marker. In that case the returned boundary is the next marker, the deleted set is the key set of the first bucket, and *pull-separates* follows from the bucket boundary invariant. The scan-cost lemmas count exactly the returned keys, so the primitive cost is at most M; the amortized Pull bound additionally allows potential to change after deleting those keys.

14.1 Paper-Facing Costed Operations

The following aliases are the operations exported to the rest of the BMSSP development. They choose the lazy precise Insert, the direct bucketed BatchPrepend, and the first-bucket Pull scan. The aliases keep the public layer small: clients do not need to know which internal costed variant proved the bound, only that the selected operation refines the abstract interface and has the paper budget.

definition *c-bp-paper-insert* ::
'k ⇒ real ⇒ 'k bucketed-partition ⇒
'k bucketed-partition × nat **where**
c-bp-paper-insert x b P = c-bp-lazy-insert-precise x b P

definition *c-bp-paper-batch-prepend* ::
('k × real) list ⇒ 'k bucketed-partition ⇒
'k bucketed-partition × nat **where**
c-bp-paper-batch-prepend xs P =
c-bp-bucketed-batch-prepend-direct-actual xs P

definition *c-bp-paper-pull* ::
nat ⇒ real ⇒ 'k bucketed-partition ⇒
('k set × real × 'k bucketed-partition) × nat **where**
c-bp-paper-pull M B P = c-bp-first-bucket-pull-scan M B P

definition *bp-insert-paper-budget* :: *nat ⇒ nat ⇒ nat* **where**
bp-insert-paper-budget N M =
9 + floor-log (Suc (N div M))

definition *bp-batch-prepend-paper-budget* :: *nat ⇒ nat ⇒ nat* **where**
bp-batch-prepend-paper-budget N M =
2 + floor-log (Suc (N div M))

definition *bp-pull-paper-budget* :: *nat ⇒ nat* **where**
bp-pull-paper-budget M = 2 ∗ M

lemma *bp-lazy-insert-amortized-budget-paper-form*:
bp-lazy-insert-amortized-budget P =
bp-insert-paper-budget (length (bp-entries P)) (bp-block-size P)
unfolding *bp-lazy-insert-amortized-budget-def*
bp-insert-search-budget-def bp-ratio-log-budget-def
bp-insert-paper-budget-def
by *simp*

lemma *bp-batch-prepend-amortized-budget-paper-form*:
bp-batch-prepend-amortized-budget xs M =
length xs ∗ bp-batch-prepend-paper-budget (length xs) M
unfolding *bp-batch-prepend-amortized-budget-def*
bp-batch-prepend-log-budget-def bp-ratio-log-budget-def
bp-batch-prepend-paper-budget-def
by (*simp add*: *algebra-simps*)

lemma *bp-ratio-log-budget-le-insert-paper-budget*:
bp-ratio-log-budget N M ≤ bp-insert-paper-budget N M
unfolding *bp-ratio-log-budget-def bp-insert-paper-budget-def* **by** *simp*

lemma *bp-ratio-log-budget-le-batch-prepend-paper-budget*:
bp-ratio-log-budget N M ≤ bp-batch-prepend-paper-budget N M

unfolding *bp-ratio-log-budget-def bp-batch-prepend-paper-budget-def* **by** *simp*

lemma *bp-insert-paper-budget-bigo-ratio-log*:
(λ*N*. *real* (*bp-insert-paper-budget N M*)) ∈
O(λ*N*. *real* (*bp-ratio-log-budget N M*))
proof −
have *bound*:
⋀*N*. *norm* (*real* (*bp-insert-paper-budget N M*)) ≤
9 ∗ *norm* (*real* (*bp-ratio-log-budget N M*))
proof −
fix *N*
have *bp-insert-paper-budget N M* ≤
9 ∗ *bp-ratio-log-budget N M*
unfolding *bp-insert-paper-budget-def bp-ratio-log-budget-def* **by** *simp*
then show *norm* (*real* (*bp-insert-paper-budget N M*)) ≤
9 ∗ *norm* (*real* (*bp-ratio-log-budget N M*))
by *simp*
qed
show *?thesis*
unfolding *bigo-def*
by (*intro CollectI exI*[*of* - *9*] *eventuallyI*)
(*simp add*: *bp-insert-paper-budget-def bp-ratio-log-budget-def*)
qed

lemma *bp-batch-prepend-paper-budget-bigo-ratio-log*:
(λ*N*. *real* (*bp-batch-prepend-paper-budget N M*)) ∈
O(λ*N*. *real* (*bp-ratio-log-budget N M*))
proof −
have *bound*:
⋀*N*. *norm* (*real* (*bp-batch-prepend-paper-budget N M*)) ≤
2 ∗ *norm* (*real* (*bp-ratio-log-budget N M*))
proof −
fix *N*
have *bp-batch-prepend-paper-budget N M* ≤
2 ∗ *bp-ratio-log-budget N M*
unfolding *bp-batch-prepend-paper-budget-def bp-ratio-log-budget-def*
by *simp*
then show *norm* (*real* (*bp-batch-prepend-paper-budget N M*)) ≤
2 ∗ *norm* (*real* (*bp-ratio-log-budget N M*))
by *simp*
qed
show *?thesis*
unfolding *bigo-def*
by (*intro CollectI exI*[*of* - *2*] *eventuallyI*)
(*simp add*: *bp-batch-prepend-paper-budget-def bp-ratio-log-budget-def*)
qed

lemma *bp-pull-paper-budget-bigo-linear*:
(λ*M*. *real* (*bp-pull-paper-budget M*)) ∈ *O*(λ*M*. *real M*)

unfolding *bp-pull-paper-budget-def bigo-def*
by (*intro CollectI exI*[*of - 2*] *eventuallyI*) *simp*

The named paper budgets are intentionally simple. The Insert budget *bp-insert-paper-budget* is a constant plus the ratio-log term over stored entries and block size. The BatchPrepend budget *bp-batch-prepend-paper-budget* is the corresponding per-item budget; the abstract batch predicate then multiplies it by the batch length. Pull uses *bp-pull-paper-budget*, linear in the requested block size. The Big-O lemmas above record exactly the envelopes consumed by the headline bridge.

lemma *c-bp-paper-insert-result* [*simp*]:
bp-result-of (*c-bp-paper-insert x b P*) =
bp-lazy-insert-state x b P
unfolding *c-bp-paper-insert-def* **by** *simp*

lemma *c-bp-paper-insert-steps* [*simp*]:
bp-steps-of (*c-bp-paper-insert x b P*) =
bp-local-insert-search-charge P + *bp-lazy-insert-charge x b P*
unfolding *c-bp-paper-insert-def* **by** *simp*

lemma *c-bp-paper-insert-refines-min-update*:
assumes *lazy*: *bp-lazy-ordered-invariant P*
shows *bp-view* (*bp-result-of* (*c-bp-paper-insert x b P*)) =
min-update (*bp-view P*) *x b*
proof −
have *M-pos*: *0* < *bp-block-size P*
using *lazy* **unfolding** *bp-lazy-ordered-invariant-def*
bp-lazy-invariant-def **by** *blast*
show *?thesis*
unfolding *c-bp-paper-insert-def*
by (*rule c-bp-lazy-insert-precise-refines-min-update*[*OF M-pos*])
qed

lemma *c-bp-paper-insert-refines-insert-spec*:
assumes *lazy*: *bp-lazy-ordered-invariant P*
shows *insert-spec* (*bp-view P*) *x b*
(*bp-view* (*bp-result-of* (*c-bp-paper-insert x b P*)))
unfolding *c-bp-paper-insert-refines-min-update*[*OF lazy*]
by (*rule min-update-insert-spec*)

lemma *c-bp-paper-insert-lazy-ordered-invariant*:
assumes *lazy*: *bp-lazy-ordered-invariant P*
shows *bp-lazy-ordered-invariant*
(*bp-result-of* (*c-bp-paper-insert x b P*))
unfolding *c-bp-paper-insert-def*
by (*rule c-bp-lazy-insert-precise-lazy-ordered-invariant*[*OF lazy*])

lemma *c-bp-paper-insert-amortized-paper-bound*:
assumes *lazy*: *bp-lazy-ordered-invariant P*

and *ratio*: *bp-bucket-count-ratio-ok P*
shows *bp-amortized-step-bound*
(*bp-steps-of* (*c-bp-paper-insert x b P*)) *P*
(*bp-result-of* (*c-bp-paper-insert x b P*))
(*bp-lazy-insert-amortized-budget P*)
unfolding *c-bp-paper-insert-def*
by (*rule c-bp-lazy-insert-precise-amortized-ratio-budget*
[*OF lazy ratio*])

lemma *c-bp-paper-insert-regular-partition-insert-cost-bound*:
assumes *reg*: *bp-regular-invariant P*
shows *partition-insert-cost-bound*
(*bp-steps-of* (*c-bp-paper-insert x b P*))
(*bp-lazy-insert-amortized-budget P*)
unfolding *c-bp-paper-insert-def*
by (*rule c-bp-lazy-insert-precise-regular-partition-insert-cost-bound*
[*OF reg*])

lemma *c-bp-paper-batch-prepend-result* [*simp*]:
bp-result-of (*c-bp-paper-batch-prepend xs P*) =
bp-bucketed-batch-prepend-state xs P
unfolding *c-bp-paper-batch-prepend-def* **by** *simp*

lemma *c-bp-paper-batch-prepend-steps* [*simp*]:
bp-steps-of (*c-bp-paper-batch-prepend xs P*) =
bp-batch-prepend-log-budget xs (*bp-block-size P*)
unfolding *c-bp-paper-batch-prepend-def* **by** *simp*

lemma *c-bp-paper-batch-prepend-refines-batch-min-update*:
assumes *ord*: *bp-ordered-invariant P*
and *distinct*: *distinct* (*map fst xs*)
and *disjoint*: *bp-entry-keys xs* ∩ *keys-of* (*bp-view P*) = {}
shows *bp-view* (*bp-result-of* (*c-bp-paper-batch-prepend xs P*)) =
batch-min-update (*bp-view P*) *xs*
unfolding *c-bp-paper-batch-prepend-def*
by (*rule c-bp-bucketed-batch-prepend-direct-actual-refines-batch-min-update*
[*OF ord distinct disjoint*])

lemma *c-bp-paper-batch-prepend-ordered-invariant*:
assumes *ord*: *bp-ordered-invariant P*
and *distinct*: *distinct* (*map fst xs*)
and *disjoint*: *bp-entry-keys xs* ∩ *keys-of* (*bp-view P*) = {}
and *admissible*: *batch-prepend-admissible* (*bp-view P*) *xs*
shows *bp-ordered-invariant*
(*bp-result-of* (*c-bp-paper-batch-prepend xs P*))
unfolding *c-bp-paper-batch-prepend-def*
by (*rule c-bp-bucketed-batch-prepend-direct-actual-ordered-invariant*
[*OF ord distinct disjoint admissible*])

lemma *c-bp-paper-batch-prepend-preserves-upper-bound*:
assumes *ord*: *bp-ordered-invariant P*
and *distinct*: *distinct* (*map fst xs*)
and *disjoint*: *bp-entry-keys xs* ∩ *keys-of* (*bp-view P*) = {}
and *upper*: *partition-upper-bound* (*bp-view P*) *B*
and *values-lt*: ⋀*x b*. (*x*, *b*) ∈ *set xs* ⟹ *b* < *B*
shows *partition-upper-bound*
(*bp-view* (*bp-result-of* (*c-bp-paper-batch-prepend xs P*))) *B*
unfolding *c-bp-paper-batch-prepend-def*
by (*rule c-bp-bucketed-batch-prepend-direct-actual-preserves-upper-bound*
[*OF ord distinct disjoint upper values-lt*])

lemma *c-bp-paper-batch-prepend-amortized-paper-bound*:
assumes *ord*: *bp-ordered-invariant P*
and *distinct*: *distinct* (*map fst xs*)
and *disjoint*: *bp-entry-keys xs* ∩ *keys-of* (*bp-view P*) = {}
and *admissible*: *batch-prepend-admissible* (*bp-view P*) *xs*
shows *bp-amortized-step-bound*
(*bp-steps-of* (*c-bp-paper-batch-prepend xs P*)) *P*
(*bp-result-of* (*c-bp-paper-batch-prepend xs P*))
(*bp-batch-prepend-amortized-budget xs* (*bp-block-size P*))
unfolding *c-bp-paper-batch-prepend-def*
by (*rule c-bp-bucketed-batch-prepend-direct-actual-amortized-paper-bound*
[*OF ord distinct disjoint admissible*])

lemma *c-bp-paper-batch-prepend-batch-cost-bound*:
partition-batch-cost-bound
(*bp-steps-of* (*c-bp-paper-batch-prepend xs P*))
(*bp-ratio-log-budget* (*length xs*) (*bp-block-size P*)) *xs*
unfolding *c-bp-paper-batch-prepend-def*
by (*rule c-bp-bucketed-batch-prepend-direct-actual-batch-cost-bound*)

lemma *c-bp-paper-pull-result* [*simp*]:
bp-result-of (*c-bp-paper-pull M B P*) =
bp-first-bucket-pull M B P
unfolding *c-bp-paper-pull-def* **by** *simp*

lemma *c-bp-paper-pull-M-bound*:
assumes *can*: *bp-can-first-bucket-pull M P*
shows *bp-steps-of* (*c-bp-paper-pull M B P*) ≤ *M*
unfolding *c-bp-paper-pull-def*
by (*rule c-bp-first-bucket-pull-scan-M-bound*[*OF can*])

lemma *c-bp-paper-pull-partition-pull-cost-bound*:
assumes *pull*:
bp-result-of (*c-bp-paper-pull M B P*) = (*S*, *beta*, *P'*)
shows *partition-pull-cost-bound*
(*bp-steps-of* (*c-bp-paper-pull M B P*)) *S*
unfolding *c-bp-paper-pull-def*

```
  by (rule c-bp-first-bucket-pull-scan-partition-pull-cost-bound-from-result
      [OF pull[unfolded c-bp-paper-pull-def]])

lemma c-bp-paper-pull-invariant:
  assumes inv: bp-invariant P
  shows bp-invariant
    (bp-pull-state-of (bp-result-of (c-bp-paper-pull M B P)))
  unfolding c-bp-paper-pull-def
  by (rule c-bp-first-bucket-pull-scan-invariant[OF inv])

lemma c-bp-paper-pull-amortized-paper-bound:
  assumes inv: bp-invariant P
    and can: bp-can-first-bucket-pull M P
  shows bp-pull-amortized-step-bound
    (bp-steps-of (c-bp-paper-pull M B P)) P
    (bp-result-of (c-bp-paper-pull M B P))
    (bp-pull-amortized-budget M)
  unfolding c-bp-paper-pull-def
  by (rule c-bp-first-bucket-pull-scan-amortized-paper-bound[OF inv can])

lemma c-bp-paper-pull-refines-pull-separates:
  assumes inv: bp-invariant P
    and can: bp-can-first-bucket-pull M P
    and pull:
      bp-result-of (c-bp-paper-pull M B P) = (S, beta, P')
  shows pull-separates (bp-view P) M B S beta (bp-view P')
  unfolding c-bp-paper-pull-def
  by (rule c-bp-first-bucket-pull-scan-refines-pull-separates
      [OF inv can pull[unfolded c-bp-paper-pull-def]])

corollary bp-realises-partition-insert-cost-bound:
  assumes reg: bp-regular-invariant P
  shows ∃ t. partition-insert-cost-bound
      (bp-steps-of (c-bp-paper-insert x b P)) t ∧
    t = bp-insert-paper-budget (length (bp-entries P)) (bp-block-size P)
proof (intro exI conjI)
  show partition-insert-cost-bound
      (bp-steps-of (c-bp-paper-insert x b P))
      (bp-insert-paper-budget (length (bp-entries P)) (bp-block-size P))
    using c-bp-paper-insert-regular-partition-insert-cost-bound[OF reg, of x b]
    by (simp add: bp-lazy-insert-amortized-budget-paper-form)
qed simp

corollary bp-realises-partition-batch-cost-bound:
  shows ∃ t. partition-batch-cost-bound
      (bp-steps-of (c-bp-paper-batch-prepend xs P)) t xs ∧
    t = bp-batch-prepend-paper-budget (length xs) (bp-block-size P)
proof (intro exI conjI)
  have cost:
```

```
    partition-batch-cost-bound
      (bp-steps-of (c-bp-paper-batch-prepend xs P))
      (bp-ratio-log-budget (length xs) (bp-block-size P)) xs
    by (rule c-bp-paper-batch-prepend-batch-cost-bound)
  have le:
    bp-ratio-log-budget (length xs) (bp-block-size P) ≤
      bp-batch-prepend-paper-budget (length xs) (bp-block-size P)
    by (rule bp-ratio-log-budget-le-batch-prepend-paper-budget)
  show partition-batch-cost-bound
      (bp-steps-of (c-bp-paper-batch-prepend xs P))
      (bp-batch-prepend-paper-budget (length xs) (bp-block-size P)) xs
  proof −
    have steps-le:
      bp-steps-of (c-bp-paper-batch-prepend xs P) ≤
        bp-ratio-log-budget (length xs) (bp-block-size P) * length xs
      using cost unfolding partition-batch-cost-bound-def .
    have budget-le:
      bp-ratio-log-budget (length xs) (bp-block-size P) * length xs ≤
        bp-batch-prepend-paper-budget (length xs) (bp-block-size P) *
          length xs
      using le by simp
    show ?thesis
      using steps-le budget-le unfolding partition-batch-cost-bound-def
      by linarith
  qed
qed simp

corollary bp-realises-partition-pull-cost-bound:
  assumes pull:
    bp-result-of (c-bp-paper-pull M B P) = (S, beta, P′)
  shows partition-pull-cost-bound
    (bp-steps-of (c-bp-paper-pull M B P)) S
  by (rule c-bp-paper-pull-partition-pull-cost-bound[OF pull])
```

The realization corollaries are the interface-facing summary. They convert the concrete step counters of *c-bp-paper-insert*, *c-bp-paper-batch-prepend*, and *c-bp-paper-pull* into the abstract predicates *partition-insert-cost-bound*, *partition-batch-cost-bound*, and *partition-pull-cost-bound*. These are the facts imported by the headline bridge; the operation-specific theorems below keep the sharper amortized statements visible for readers who want to inspect the data-structure proof itself.

```
theorem bp-insert-cost-bound:
  assumes lazy: bp-lazy-ordered-invariant P
    and ratio: bp-bucket-count-ratio-ok P
  shows bp-amortized-step-bound
    (bp-steps-of (c-bp-paper-insert x b P)) P
    (bp-result-of (c-bp-paper-insert x b P))
    (9 + floor-log
      (Suc (length (bp-entries P) div bp-block-size P)))
```

using *c-bp-paper-insert-amortized-paper-bound*[*OF lazy ratio*, *of x b*]
by (*simp add*: *bp-lazy-insert-amortized-budget-paper-form*
bp-insert-paper-budget-def)

theorem *bp-batch-prepend-cost-bound*:
assumes *ord*: *bp-ordered-invariant P*
and *distinct*: *distinct* (*map fst xs*)
and *disjoint*: *bp-entry-keys xs* ∩ *keys-of* (*bp-view P*) = {}
and *admissible*: *batch-prepend-admissible* (*bp-view P*) *xs*
shows *bp-amortized-step-bound*
(*bp-steps-of* (*c-bp-paper-batch-prepend xs P*)) *P*
(*bp-result-of* (*c-bp-paper-batch-prepend xs P*))
(*length xs* ∗
(*2* + *floor-log* (*Suc* (*length xs div bp-block-size P*))))
using *c-bp-paper-batch-prepend-amortized-paper-bound*
[*OF ord distinct disjoint admissible*]
by (*simp add*: *bp-batch-prepend-amortized-budget-paper-form*
bp-batch-prepend-paper-budget-def)

theorem *bp-pull-cost-bound*:
assumes *inv*: *bp-invariant P*
and *can*: *bp-can-first-bucket-pull M P*
shows *bp-pull-amortized-step-bound*
(*bp-steps-of* (*c-bp-paper-pull M B P*)) *P*
(*bp-result-of* (*c-bp-paper-pull M B P*))
(*2* ∗ *M*)
using *c-bp-paper-pull-amortized-paper-bound*[*OF inv can*, *of B*]
unfolding *bp-pull-amortized-budget-def* **.**

The three theorems above are the exported paper-tight cost bounds for the bucketed structure. ⟦*bp-lazy-ordered-invariant ?P*; *bp-bucket-count-ratio-ok ?P*⟧ ⟹ *bp-amortized-step-bound* (*bp-steps-of* (*c-bp-paper-insert ?x ?b ?P*)) *?P* (*bp-result-of* (*c-bp-paper-insert ?x ?b ?P*)) (*9* + *floor-log* (*Suc* (*length* (*bp-entries ?P*) *div bp-block-size ?P*))) combines a ratio-log bucket search with the lazy split potential, giving an amortized Insert budget of a constant plus *floor-log* of *Suc* (*N div M*). This is the formal $\log(N/M)$ term required by the BMSSP parameter schedule. The batch theorem ⟦*bp-ordered-invariant ?P*; *distinct* (*map fst ?xs*); *bp-entry-keys ?xs* ∩ *keys-of* (*bp-view ?P*) = {}; *batch-prepend-admissible* (*bp-view ?P*) *?xs*⟧ ⟹ *bp-amortized-step-bound* (*bp-steps-of* (*c-bp-paper-batch-prepend ?xs ?P*)) *?P* (*bp-result-of* (*c-bp-paper-batch-prepend ?xs ?P*)) (*length ?xs* ∗ (*2* + *floor-log* (*Suc* (*length ?xs div bp-block-size ?P*)))) charges the same ratio-log scale per incoming item, matching the paper's batched prepend cost. Finally, ⟦*bp-invariant ?P*; *bp-can-first-bucket-pull ?M ?P*⟧ ⟹ *bp-pull-amortized-step-bound* (*bp-steps-of* (*c-bp-paper-pull ?M ?B ?P*)) *?P* (*bp-result-of* (*c-bp-paper-pull ?M ?B ?P*)) (*2* ∗ *?M*) gives the amortized linear Pull budget *2* ∗ *M*. These statements are intentionally stronger and more concrete than the abstract cost-interface

corollaries: they expose the primitive-step and potential accounting that justifies plugging the structure into the top-level runtime proof.

end
theory *BMSSP-Bucketed-Partition*
imports *BMSSP-Bucketed-Partition-Internal*
begin

15 Bucketed Partition

This theory is the public import point for the bucketed partition implementation. The definitions and proof-engineering lemmas live in *BMSSP-Bucketed-Partition-Internal*; importing this wrapper exports the same constants and facts to the rest of the BMSSP development while keeping the documented surface short.

end
theory *BMSSP-Code-Export*
imports *BMSSP-Bucketed-Partition*
begin

16 Executable Bucketed BMSSP Smoke Test

This theory gives the development an executable surface. All preceding imported theory proves the paper-faithful bucketed operation costs and the concrete partition API. Those facts are essential, but they do not by themselves show a human reader that the formalized program can be run on an ordinary graph. The purpose of this file is to close that gap with a small concrete instantiation and a proof by evaluation.

The executable graph representation is deliberately plain: vertices are natural numbers, edges are triples of natural numbers, and distances are association lists sorted by vertex at the end of the run. The work-list is the paper-style bucketed partition structure imported from the bucketed theory, not the baseline sorted-list partition model. The loop below uses Pull to obtain the next bucket of unsettled vertices, relaxes outgoing edges from that batch, and feeds discovered or improved labels through BatchPrepend for an empty partition and Insert otherwise. This shape is chosen to exercise all three bucketed operations in the worked example.

The fuel argument is an executable guard, not part of the mathematical correctness theorem. For the small smoke graph it is chosen large enough to allow all relaxations needed by the hand calculation. The checked statement at the bottom is the important artifact: *example-bmssp-correct* is proved by Isabelle's evaluator, so the bundled Poly/ML runtime executes the exported equations during the build and compares the result with the literal expected distance map.

type-synonym *nat-edge* = *nat* × *nat* × *nat*
type-synonym *nat-graph* = *nat-edge list*
type-synonym *nat-dist* = (*nat* × *nat*) *list*

definition *bmssp-block-size* :: *nat* **where**
bmssp-block-size = *1*

definition *bmssp-infinity* :: *nat* **where**
bmssp-infinity = *1000000*

definition *bmssp-bound* :: *real* **where**
bmssp-bound = *real bmssp-infinity*

fun *bmssp-edge-vertices* :: *nat-edge* ⇒ *nat list* **where**
bmssp-edge-vertices (*u*, *v*, *w*) = [*u*, *v*]

definition *bmssp-vertices* :: *nat-graph* ⇒ *nat* ⇒ *nat list* **where**
bmssp-vertices G s = *sort* (*remdups* (*s* # *concat* (*map bmssp-edge-vertices G*)))

fun *bmssp-lookup-dist* :: *nat-dist* ⇒ *nat* ⇒ *nat option* **where**
bmssp-lookup-dist [] *x* = *None*
| *bmssp-lookup-dist* ((*y*, *d*) # *ds*) *x* =
(*if x* = *y then Some d else bmssp-lookup-dist ds x*)

fun *bmssp-set-dist* :: *nat* ⇒ *nat* ⇒ *nat-dist* ⇒ *nat-dist* **where**
bmssp-set-dist x d [] = [(*x*, *d*)]
| *bmssp-set-dist x d* ((*y*, *e*) # *ds*) =
(*if x* = *y then* (*x*, *d*) # *ds else* (*y*, *e*) # *bmssp-set-dist x d ds*)

definition *bmssp-normalize-dist* :: *nat-dist* ⇒ *nat-dist* **where**
bmssp-normalize-dist ds = *sort-key fst ds*

definition *bmssp-partition-key* :: *nat* ⇒ *nat* ⇒ *real* **where**
bmssp-partition-key x d = *real d* + *real x* / *real* (*Suc x*)

lemma *bmssp-partition-key-fraction-nonneg*:
0 ≤ *real x* / *real* (*Suc x*)
by *simp*

lemma *bmssp-partition-key-fraction-lt-one*:
real x / *real* (*Suc x*) < *1*
proof −
have *real x* < *real* (*Suc x*)
by *simp*
then show *?thesis*
by (*simp add*: *divide-less-eq*)
qed

lemma *bmssp-partition-key-ge-distance*:

```
  real d ≤ bmssp-partition-key x d
  unfolding bmssp-partition-key-def by simp

lemma bmssp-partition-key-lt-suc-distance:
  bmssp-partition-key x d < real (Suc d)
proof −
  have frac-lt: real x / real (Suc x) < 1
    by (rule bmssp-partition-key-fraction-lt-one)
  have bmssp-partition-key x d < real d + 1
    unfolding bmssp-partition-key-def using frac-lt by linarith
  then show ?thesis
    by simp
qed

lemma bmssp-partition-key-floor:
  nat (floor (bmssp-partition-key x d)) = d
proof (rule floor-eq4)
  show real d ≤ bmssp-partition-key x d
    by (rule bmssp-partition-key-ge-distance)
  show bmssp-partition-key x d < real (Suc d)
    by (rule bmssp-partition-key-lt-suc-distance)
qed

lemma bmssp-partition-key-strict-mono-distance:
  assumes d < e
  shows bmssp-partition-key x d < bmssp-partition-key y e
proof −
  have bmssp-partition-key x d < real (Suc d)
    by (rule bmssp-partition-key-lt-suc-distance)
  also have ... ≤ real e
    using assms by simp
  also have ... ≤ bmssp-partition-key y e
    by (rule bmssp-partition-key-ge-distance)
  finally show ?thesis .
qed

fun bmssp-insert-updates ::
  (nat × real) list ⇒ nat bucketed-partition ⇒
    nat bucketed-partition where
  bmssp-insert-updates [] P = P
| bmssp-insert-updates ((x, b) # xs) P =
    bmssp-insert-updates xs
      (bp-result-of (c-bp-regularized-local-insert x b P))

lemma bmssp-insert-updates-regular-invariant:
  assumes bp-regular-invariant P
  shows bp-regular-invariant (bmssp-insert-updates xs P)
  using assms
proof (induction xs arbitrary: P)
```

```
  case Nil
  then show ?case by simp
next
  case (Cons p xs)
  then obtain x b where p: p = (x, b)
    by force
  have step:
    bp-regular-invariant
      (bp-result-of (c-bp-regularized-local-insert x b P))
    by (rule c-bp-regularized-local-insert-regular-invariant[OF Cons.prems])
  show ?case
    unfolding p bmssp-insert-updates.simps
    by (rule Cons.IH[OF step])
qed

lemma bmssp-insert-updates-ordered-invariant:
  assumes bp-regular-invariant P
  shows bp-ordered-invariant (bmssp-insert-updates xs P)
  using bp-regular-invariant-ordered-invariant
    [OF bmssp-insert-updates-regular-invariant[OF assms]]
  .

lemma bmssp-insert-updates-refines-batch-min-update:
  assumes reg: bp-regular-invariant P
  shows bp-view (bmssp-insert-updates xs P) =
    batch-min-update (bp-view P) xs
  using reg
proof (induction xs arbitrary: P)
  case Nil
  then show ?case
    unfolding batch-min-update-def by simp
next
  case (Cons p xs)
  then obtain x b where p: p = (x, b)
    by force
  let ?P1 = bp-result-of (c-bp-regularized-local-insert x b P)
  have step-reg: bp-regular-invariant ?P1
    by (rule c-bp-regularized-local-insert-regular-invariant[OF Cons.prems])
  have step-view: bp-view ?P1 = min-update (bp-view P) x b
    by (rule c-bp-regularized-local-insert-refines-min-update[OF Cons.prems])
  have tail:
    bp-view (bmssp-insert-updates xs ?P1) =
      batch-min-update (bp-view ?P1) xs
    by (rule Cons.IH[OF step-reg])
  have head:
    batch-min-update (bp-view P) ((x, b) # xs) =
      batch-min-update (min-update (bp-view P) x b) xs
    unfolding batch-min-update-def by simp
  show ?case
```

unfolding *p bmssp-insert-updates.simps*
using *tail step-view head* **by** *simp*
qed

lemma *bmssp-insert-updates-ordered-invariant-from-ordered*:
assumes *ord*: *bp-ordered-invariant P*
shows *bp-ordered-invariant* (*bmssp-insert-updates xs P*)
proof (*cases xs*)
case *Nil*
then show *?thesis*
using *ord* **by** *simp*
next
case (*Cons p ys*)
then obtain *x b* **where** *p*: *p* = (*x*, *b*)
by *force*
let *?P1* = *bp-result-of* (*c-bp-regularized-local-insert x b P*)
have *step-reg*: *bp-regular-invariant ?P1*
unfolding *c-bp-regularized-local-insert-result*
by (*rule bp-regularized-local-insert-regular-invariant*[*OF ord*])
have *tail-ord*: *bp-ordered-invariant* (*bmssp-insert-updates ys ?P1*)
by (*rule bmssp-insert-updates-ordered-invariant*[*OF step-reg*])
show *?thesis*
unfolding *Cons p bmssp-insert-updates.simps*
by (*rule tail-ord*)
qed

lemma *bmssp-insert-updates-refines-batch-min-update-from-ordered*:
assumes *ord*: *bp-ordered-invariant P*
shows *bp-view* (*bmssp-insert-updates xs P*) =
batch-min-update (*bp-view P*) *xs*
proof (*cases xs*)
case *Nil*
then show *?thesis*
unfolding *batch-min-update-def* **by** *simp*
next
case (*Cons p ys*)
then obtain *x b* **where** *p*: *p* = (*x*, *b*)
by *force*
let *?P1* = *bp-result-of* (*c-bp-regularized-local-insert x b P*)
have *step-reg*: *bp-regular-invariant ?P1*
unfolding *c-bp-regularized-local-insert-result*
by (*rule bp-regularized-local-insert-regular-invariant*[*OF ord*])
have *step-view*: *bp-view ?P1* = *min-update* (*bp-view P*) *x b*
unfolding *c-bp-regularized-local-insert-result*
by (*rule bp-regularized-local-insert-refines-min-update*[*OF ord*])
have *tail*:
bp-view (*bmssp-insert-updates ys ?P1*) =
batch-min-update (*bp-view ?P1*) *ys*
by (*rule bmssp-insert-updates-refines-batch-min-update*[*OF step-reg*])

```
  have head:
    batch-min-update (bp-view P) ((x, b) # ys) =
      batch-min-update (min-update (bp-view P) x b) ys
    unfolding batch-min-update-def by simp
  show ?thesis
    unfolding Cons p bmssp-insert-updates.simps
    using tail step-view head by simp
qed

definition bmssp-trim-empty-prefix ::
  nat bucketed-partition ⇒ nat bucketed-partition where
  bmssp-trim-empty-prefix P =
    P(|bp-buckets := bp-drop-empty-prefix (bp-buckets P)|)

lemma bp-bucket-entries-flat-drop-empty-prefix [simp]:
  bp-bucket-entries-flat (bp-drop-empty-prefix bs) =
    bp-bucket-entries-flat bs
  by (induction bs) (simp-all add: bp-bucket-entries-flat-def)

lemma bp-drop-empty-prefix-set-subset:
  set (bp-drop-empty-prefix bs) ⊆ set bs
  by (induction bs) auto

lemma bp-drop-empty-prefix-length-le [simp]:
  length (bp-drop-empty-prefix bs) ≤ length bs
proof (induction bs)
  case Nil
  then show ?case by simp
next
  case (Cons b bs)
  show ?case
  proof (cases bp-bucket-entries b = [])
    case True
    have length (bp-drop-empty-prefix bs) ≤ Suc (length bs)
      using Cons.IH by linarith
    then show ?thesis
      using True by simp
  next
    case False
    then show ?thesis by simp
  qed
qed

lemma bp-drop-empty-prefix-sorted-wrt:
  assumes sorted-wrt R bs
  shows sorted-wrt R (bp-drop-empty-prefix bs)
  using assms
  by (induction bs) auto
```

lemma *bp-bucket-boundaries-ok-drop-empty-prefix*:
assumes *bp-bucket-boundaries-ok bs*
shows *bp-bucket-boundaries-ok* (*bp-drop-empty-prefix bs*)
using *assms*
proof (*induction bs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons b bs*)
show *?case*
proof (*cases bp-bucket-entries b* = [])
case *True*
have *bp-bucket-boundaries-ok bs*
using *Cons.prems* **by** (*cases bs*) *auto*
then show *?thesis*
using *True Cons.IH* **by** *simp*
next
case *False*
then show *?thesis*
using *Cons.prems* **by** *simp*
qed
qed

lemma *bmssp-trim-empty-prefix-entries* [*simp*]:
bp-entries (*bmssp-trim-empty-prefix P*) = *bp-entries P*
unfolding *bmssp-trim-empty-prefix-def bp-entries-def* **by** *simp*

lemma *bmssp-trim-empty-prefix-view* [*simp*]:
bp-view (*bmssp-trim-empty-prefix P*) = *bp-view P*
unfolding *bmssp-trim-empty-prefix-def bp-view-def bp-entries-def* **by** *simp*

lemma *bmssp-trim-empty-prefix-ordered-invariant*:
assumes *bp-ordered-invariant P*
shows *bp-ordered-invariant* (*bmssp-trim-empty-prefix P*)
proof −
have *inv*: *bp-invariant P*
using *assms* **unfolding** *bp-ordered-invariant-def* **by** *blast*
have *boundaries*: *bp-bucket-boundaries-state-ok P*
using *assms* **unfolding** *bp-ordered-invariant-def* **by** *blast*
have *set-subset*:
set (*bp-buckets* (*bmssp-trim-empty-prefix P*)) ⊆ *set* (*bp-buckets P*)
unfolding *bmssp-trim-empty-prefix-def*
by (*simp add*: *bp-drop-empty-prefix-set-subset*)
have *bp-invariant* (*bmssp-trim-empty-prefix P*)
proof −
have *0* < *bp-block-size* (*bmssp-trim-empty-prefix P*)
using *inv* **unfolding** *bmssp-trim-empty-prefix-def bp-invariant-def* **by** *simp*
moreover have *bp-distinct-keys* (*bmssp-trim-empty-prefix P*)
using *inv* **unfolding** *bp-invariant-def bp-distinct-keys-def* **by** *simp*

```
    moreover have bp-bucket-sizes-ok (bmssp-trim-empty-prefix P)
    proof (unfold bp-bucket-sizes-ok-def, intro ballI)
      fix b
      assume b: b ∈ set (bp-buckets (bmssp-trim-empty-prefix P))
      then have b ∈ set (bp-buckets P)
        using set-subset by blast
      then have length (bp-bucket-entries b) ≤ bp-block-size P
        using inv unfolding bp-invariant-def bp-bucket-sizes-ok-def by blast
      then show length (bp-bucket-entries b) ≤
        bp-block-size (bmssp-trim-empty-prefix P)
        unfolding bmssp-trim-empty-prefix-def by simp
    qed
    moreover have bp-bucket-markers-sorted (bmssp-trim-empty-prefix P)
      using inv
      unfolding bmssp-trim-empty-prefix-def bp-invariant-def
        bp-bucket-markers-sorted-def
      by (simp add: bp-drop-empty-prefix-sorted-wrt)
    moreover have bp-bucket-markers-lower-bound (bmssp-trim-empty-prefix P)
      using inv set-subset
      unfolding bp-invariant-def bp-bucket-markers-lower-bound-def by auto
    moreover have bp-values-consistent (bmssp-trim-empty-prefix P)
      using inv
      unfolding bmssp-trim-empty-prefix-def bp-invariant-def
        bp-values-consistent-def bp-entries-def
      by simp
    ultimately show ?thesis
      unfolding bp-invariant-def by blast
  qed
  moreover have bp-bucket-boundaries-state-ok (bmssp-trim-empty-prefix P)
    using boundaries
    unfolding bmssp-trim-empty-prefix-def bp-bucket-boundaries-state-ok-def
    by (simp add: bp-bucket-boundaries-ok-drop-empty-prefix)
  ultimately show ?thesis
    unfolding bp-ordered-invariant-def by blast
qed

lemma bmssp-trim-empty-prefix-regular-invariant:
  assumes bp-regular-invariant P
  shows bp-regular-invariant (bmssp-trim-empty-prefix P)
proof −
  have ordered: bp-ordered-invariant P
    by (rule bp-regular-invariant-ordered-invariant[OF assms])
  have ratio: bp-bucket-count-ratio-ok P
    by (rule bp-regular-invariant-ratio-ok[OF assms])
  have bp-bucket-count-ratio-ok (bmssp-trim-empty-prefix P)
  proof −
    have len-le:
      length (bp-drop-empty-prefix (bp-buckets P)) ≤ length (bp-buckets P)
      by (rule bp-drop-empty-prefix-length-le)
```

```
    have block-pos: 0 < bp-block-size P
      using ratio unfolding bp-bucket-count-ratio-ok-def by blast
    have orig-len:
      length (bp-buckets P) ≤
        Suc (length (bp-entries P) div bp-block-size P)
      using ratio unfolding bp-bucket-count-ratio-ok-def by blast
    have drop-len:
      length (bp-drop-empty-prefix (bp-buckets P)) ≤
        Suc (length (bp-entries P) div bp-block-size P)
      using len-le orig-len by (rule order-trans)
    show ?thesis
      using block-pos drop-len
      unfolding bmssp-trim-empty-prefix-def bp-bucket-count-ratio-ok-def
        bp-entries-def
      by simp
  qed
  then show ?thesis
    using bmssp-trim-empty-prefix-ordered-invariant[OF ordered]
    unfolding bp-regular-invariant-def by blast
qed

definition bmssp-apply-updates ::
  (nat × real) list ⇒ nat bucketed-partition ⇒
    nat bucketed-partition where
  bmssp-apply-updates xs P =
    (let P0 = bmssp-trim-empty-prefix P in
      if bp-entries P0 = []
      then bp-result-of (c-bp-paper-batch-prepend xs P0)
      else bmssp-insert-updates xs P0)

lemma bmssp-apply-updates-refines-batch-min-update:
  assumes reg: bp-regular-invariant P
    and distinct: distinct (map fst xs)
  shows bp-view (bmssp-apply-updates xs P) =
    batch-min-update (bp-view P) xs
proof −
  let ?P0 = bmssp-trim-empty-prefix P
  have reg0: bp-regular-invariant ?P0
    by (rule bmssp-trim-empty-prefix-regular-invariant[OF reg])
  have ord0: bp-ordered-invariant ?P0
    by (rule bp-regular-invariant-ordered-invariant[OF reg0])
  show ?thesis
  proof (cases bp-entries ?P0 = [])
    case True
    have disjoint: bp-entry-keys xs ∩ keys-of (bp-view ?P0) = {}
      using True unfolding bp-view-def bp-entry-keys-def by simp
    have bp-view (bp-result-of (c-bp-paper-batch-prepend xs ?P0)) =
      batch-min-update (bp-view ?P0) xs
      by (rule c-bp-paper-batch-prepend-refines-batch-min-update
```

```
      [OF ord0 distinct disjoint])
    then show ?thesis
      using True unfolding bmssp-apply-updates-def by simp
  next
    case False
    have bp-view (bmssp-insert-updates xs ?P0) =
      batch-min-update (bp-view ?P0) xs
      by (rule bmssp-insert-updates-refines-batch-min-update[OF reg0])
    then show ?thesis
      using False unfolding bmssp-apply-updates-def by simp
  qed
qed

lemma bmssp-apply-updates-ordered-invariant:
  assumes reg: bp-regular-invariant P
    and distinct: distinct (map fst xs)
    and admissible:
      batch-prepend-admissible (bp-view (bmssp-trim-empty-prefix P)) xs
  shows bp-ordered-invariant (bmssp-apply-updates xs P)
proof −
  let ?P0 = bmssp-trim-empty-prefix P
  have reg0: bp-regular-invariant ?P0
    by (rule bmssp-trim-empty-prefix-regular-invariant[OF reg])
  have ord0: bp-ordered-invariant ?P0
    by (rule bp-regular-invariant-ordered-invariant[OF reg0])
  show ?thesis
  proof (cases bp-entries ?P0 = [])
    case True
    have disjoint: bp-entry-keys xs ∩ keys-of (bp-view ?P0) = {}
      using True unfolding bp-view-def bp-entry-keys-def by simp
    have bp-ordered-invariant
      (bp-result-of (c-bp-paper-batch-prepend xs ?P0))
      by (rule c-bp-paper-batch-prepend-ordered-invariant
          [OF ord0 distinct disjoint admissible])
    then show ?thesis
      using True unfolding bmssp-apply-updates-def by simp
  next
    case False
    have bp-ordered-invariant (bmssp-insert-updates xs ?P0)
      by (rule bmssp-insert-updates-ordered-invariant[OF reg0])
    then show ?thesis
      using False unfolding bmssp-apply-updates-def by simp
  qed
qed

lemma bmssp-apply-updates-refines-batch-min-update-from-ordered:
  assumes ord: bp-ordered-invariant P
    and distinct: distinct (map fst xs)
  shows bp-view (bmssp-apply-updates xs P) =
```

```
    batch-min-update (bp-view P) xs
proof −
  let ?P0 = bmssp-trim-empty-prefix P
  have ord0: bp-ordered-invariant ?P0
    by (rule bmssp-trim-empty-prefix-ordered-invariant[OF ord])
  show ?thesis
  proof (cases bp-entries ?P0 = [])
    case True
    have disjoint: bp-entry-keys xs ∩ keys-of (bp-view ?P0) = {}
      using True unfolding bp-view-def bp-entry-keys-def by simp
    have bp-view (bp-result-of (c-bp-paper-batch-prepend xs ?P0)) =
      batch-min-update (bp-view ?P0) xs
      by (rule c-bp-paper-batch-prepend-refines-batch-min-update
          [OF ord0 distinct disjoint])
    then show ?thesis
      using True unfolding bmssp-apply-updates-def by simp
  next
    case False
    have bp-view (bmssp-insert-updates xs ?P0) =
      batch-min-update (bp-view ?P0) xs
      by (rule bmssp-insert-updates-refines-batch-min-update-from-ordered
          [OF ord0])
    then show ?thesis
      using False unfolding bmssp-apply-updates-def by simp
  qed
qed

lemma bmssp-apply-updates-ordered-invariant-from-ordered:
  assumes ord: bp-ordered-invariant P
    and distinct: distinct (map fst xs)
  shows bp-ordered-invariant (bmssp-apply-updates xs P)
proof −
  let ?P0 = bmssp-trim-empty-prefix P
  have ord0: bp-ordered-invariant ?P0
    by (rule bmssp-trim-empty-prefix-ordered-invariant[OF ord])
  show ?thesis
  proof (cases bp-entries ?P0 = [])
    case True
    have disjoint: bp-entry-keys xs ∩ keys-of (bp-view ?P0) = {}
      using True unfolding bp-view-def bp-entry-keys-def by simp
    have admissible: batch-prepend-admissible (bp-view ?P0) xs
      using True
      unfolding batch-prepend-admissible-def bp-view-def bp-entry-keys-def
      by simp
    have bp-ordered-invariant
      (bp-result-of (c-bp-paper-batch-prepend xs ?P0))
      by (rule c-bp-paper-batch-prepend-ordered-invariant
          [OF ord0 distinct disjoint admissible])
    then show ?thesis
```

```
    using True unfolding bmssp-apply-updates-def by simp
  next
    case False
    have bp-ordered-invariant (bmssp-insert-updates xs ?P0)
      by (rule bmssp-insert-updates-ordered-invariant-from-ordered[OF ord0])
    then show ?thesis
      using False unfolding bmssp-apply-updates-def by simp
  qed
qed

lemma bmssp-apply-updates-preserves-upper-bound:
  assumes reg: bp-regular-invariant P
    and upper: partition-upper-bound (bp-view P) B
    and distinct: distinct (map fst xs)
    and values-lt: ⋀x b. (x, b) ∈ set xs ⟹ b < B
  shows partition-upper-bound (bp-view (bmssp-apply-updates xs P)) B
  unfolding bmssp-apply-updates-refines-batch-min-update[OF reg distinct]
  by (rule batch-min-update-preserves-upper-bound[OF upper values-lt])

lemma bmssp-apply-updates-preserves-upper-bound-from-ordered:
  assumes ord: bp-ordered-invariant P
    and upper: partition-upper-bound (bp-view P) B
    and distinct: distinct (map fst xs)
    and values-lt: ⋀x b. (x, b) ∈ set xs ⟹ b < B
  shows partition-upper-bound (bp-view (bmssp-apply-updates xs P)) B
  unfolding bmssp-apply-updates-refines-batch-min-update-from-ordered
    [OF ord distinct]
  by (rule batch-min-update-preserves-upper-bound[OF upper values-lt])

lemma bmssp-pull-refines-pull-separates:
  assumes ord: bp-ordered-invariant P
    and upper: partition-upper-bound (bp-view P) B
    and pull: bp-pull M B P = (S, beta, P')
  shows pull-separates (bp-view P) M B S beta (bp-view P')
proof (rule bp-pull-refines-pull-separates[OF - - pull])
  show bp-invariant P
    by (rule bp-ordered-invariant-invariant[OF ord])
next
  fix u
  assume u ∈ keys-of (bp-view P)
  then show value-of (bp-view P) u < B
    using upper unfolding partition-upper-bound-def by blast
qed

lemma bmssp-pull-ordered-invariant:
  assumes ord: bp-ordered-invariant P
    and pull: bp-pull M B P = (S, beta, P')
  shows bp-ordered-invariant P'
  by (rule bp-pull-ordered-invariant[OF ord pull])
```

lemma *bmssp-pull-preserves-upper-bound*:
  assumes *ord*: *bp-ordered-invariant P*
    and *upper*: *partition-upper-bound* (*bp-view P*) *B*
    and *pull*: *bp-pull M B P* = (*S*, *beta*, *P'*)
  shows *partition-upper-bound* (*bp-view P'*) *B*
proof −
  have *inv*: *bp-invariant P*
    by (*rule bp-ordered-invariant-invariant*[*OF ord*])
  show *?thesis*
    by (*rule bp-pull-preserves-upper-bound*[*OF inv upper upper pull*])
qed

lemma *bmssp-pull-step-bridge*:
  assumes *ord*: *bp-ordered-invariant P*
    and *upper*: *partition-upper-bound* (*bp-view P*) *B*
    and *pull*: *bp-pull M B P* = (*S*, *beta*, *P'*)
  shows *pull-separates* (*bp-view P*) *M B S beta* (*bp-view P'*)
    and *bp-ordered-invariant P'*
    and *partition-upper-bound* (*bp-view P'*) *B*
  using *bmssp-pull-refines-pull-separates*[*OF assms*]
    bmssp-pull-ordered-invariant[*OF ord pull*]
    bmssp-pull-preserves-upper-bound[*OF assms*]
  by *blast+*

lemma *bmssp-vertices-distinct* [*simp*]:
  distinct (*bmssp-vertices G s*)
  unfolding *bmssp-vertices-def* **by** *simp*

lemma *finite-keys-of-bp-view* [*simp*]:
  finite (*keys-of* (*bp-view P*))
  unfolding *bp-view-def bp-entry-keys-def* **by** *simp*

lemma *bmssp-pulled-length-le-block-size*:
  assumes *distinct-vertices*: *distinct vertices*
    and *pull*: *pull-separates* (*bp-view P*) *bmssp-block-size B S beta D'*
  shows *length* (*filter* (λ*x*. *x* ∈ *S* ∧ *x* ∉ *set settled*) *vertices*)
    ≤ *bmssp-block-size*
proof −
  let *?pulled* = *filter* (λ*x*. *x* ∈ *S* ∧ *x* ∉ *set settled*) *vertices*
  have *S-subset*: *S* ⊆ *keys-of* (*bp-view P*)
    by (*rule pull-separates-subset*[*OF pull*])
  then have *finite-S*: *finite S*
    by (*rule finite-subset*) *simp*
  have *set-pulled*: *set ?pulled* ⊆ *S*
    by *auto*
  have *distinct-pulled*: *distinct ?pulled*
    using *distinct-vertices* **by** *simp*
  have *length ?pulled* = *card* (*set ?pulled*)

```
    using distinct-card[OF distinct-pulled] by simp
  also have ... ≤ card S
    by (rule card-mono[OF finite-S set-pulled])
  also have ... ≤ bmssp-block-size
    by (rule pull-separates-card[OF pull])
  finally show ?thesis .
qed

lemma bmssp-pulled-length-le-one:
  assumes distinct-vertices: distinct vertices
    and pull: pull-separates (bp-view P) bmssp-block-size B S beta D'
  shows length (filter (λx. x ∈ S ∧ x ∉ set settled) vertices)
    ≤ 1
  using bmssp-pulled-length-le-block-size[OF distinct-vertices pull]
  unfolding bmssp-block-size-def .

fun bmssp-relax-edges ::
  nat-graph ⇒ nat list ⇒ nat ⇒ nat ⇒ nat-dist ⇒
    (nat × real) list × nat-dist where
  bmssp-relax-edges [] settled u du ds = ([], ds)
| bmssp-relax-edges ((a, b, w) # es) settled u du ds =
    (case bmssp-relax-edges es settled u du ds of
      (updates, ds1) ⇒
        (if a = u ∧ b ∉ set settled
         then
           (let nd = du + w in
            case bmssp-lookup-dist ds1 b of
              None ⇒
                ((b, bmssp-partition-key b nd) # updates,
                 bmssp-set-dist b nd ds1)
            | Some old ⇒
                (if nd < old
                 then ((b, bmssp-partition-key b nd) # updates,
                       bmssp-set-dist b nd ds1)
                 else (updates, ds1)))
         else (updates, ds1)))

fun bmssp-relax-vertices ::
  nat-graph ⇒ nat list ⇒ nat list ⇒ nat-dist ⇒
    (nat × real) list × nat-dist where
  bmssp-relax-vertices G settled [] ds = ([], ds)
| bmssp-relax-vertices G settled (u # us) ds =
    (case bmssp-lookup-dist ds u of
       None ⇒ bmssp-relax-vertices G settled us ds
     | Some du ⇒
         (case bmssp-relax-edges G settled u du ds of
           (updates-u, ds1) ⇒
             (case bmssp-relax-vertices G settled us ds1 of
               (updates-us, ds2) ⇒ (updates-u @ updates-us, ds2))))
```

```
fun bmssp-loop ::
  nat ⇒ nat-graph ⇒ nat list ⇒ nat list ⇒
    nat-dist ⇒ nat bucketed-partition ⇒ nat-dist where
  bmssp-loop 0 G vertices settled ds P = bmssp-normalize-dist ds
| bmssp-loop (Suc fuel) G vertices settled ds P =
    (case bp-pull bmssp-block-size bmssp-bound P of
      (S, beta, P1) ⇒
        (let pulled = filter (λx. x ∈ S ∧ x ∉ set settled) vertices in
         if pulled = []
         then bmssp-normalize-dist ds
         else
           (let settled' = pulled @ settled;
                relaxed = bmssp-relax-vertices G settled' pulled ds;
                updates = fst relaxed;
                ds' = snd relaxed;
                P2 = bmssp-apply-updates updates P1
            in bmssp-loop fuel G vertices settled' ds' P2)))

definition bmssp-distances :: nat-graph ⇒ nat ⇒ nat-dist where
  bmssp-distances G s =
    (let vertices = bmssp-vertices G s;
         P0 = bp-empty bmssp-block-size bmssp-bound;
         P1 = bp-result-of
           (c-bp-regularized-local-insert s (bmssp-partition-key s 0) P0);
         fuel = Suc (length vertices * Suc (length G))
     in bmssp-loop fuel G vertices [] [(s, 0)] P1)
```

The entry point *bmssp-distances* initializes the bucketed partition with the source at distance zero and then iterates *bmssp-loop*. The helper *bmssp-apply-updates* makes the example exercise both bulk and singleton update paths: an empty work-list receives a batch prepend, while a non-empty work-list receives local inserts. The bucketed Pull operation is invoked in every loop iteration through *bp-pull*.

```
definition example-graph :: nat-graph where
  example-graph =
    [(0, 1, 3), (0, 4, 6), (0, 2, 10),
     (1, 2, 2), (1, 4, 4), (1, 3, 9),
     (2, 3, 3), (2, 4, 1), (4, 3, 5)]
```

From source 0: d(0) = 0 d(1) = 3 via 0 -> 1 d(2) = 5 via 0 -> 1 -> 2 d(3) = 8 via 0 -> 1 -> 2 -> 3 d(4) = 6 via 0 -> 4, tied by 0 -> 1 -> 2 -> 4

```
definition example-expected-dist :: nat-dist where
  example-expected-dist = [(0, 0), (1, 3), (2, 5), (3, 8), (4, 6)]
```

The graph is small enough to audit by hand but nontrivial enough to check the executable plumbing. Vertex 0 starts the run. Vertex 1 is reached directly with cost 3; vertex 2 improves from the direct edge of cost 10 to cost 5 via 1; vertex 3 is then reached at cost 8 via 2; and vertex 4 has final

cost 6, tied between the direct edge and the path through 1 and 2. The constant *example-expected-dist* records that hand calculation as a literal sorted association list.

lemma *example-bmssp-correct*:
  bmssp-distances example-graph 0 = example-expected-dist
  by *eval*

lemma *bmssp-counterexample-fixed*:
  bmssp-distances [(*0*::*nat*, *1*::*nat*, *1*::*nat*), (*0*, *2*, *1*), (*1*, *3*, *2*)] *0*
    = [(*0*, *0*), (*1*, *1*), (*2*, *1*), (*3*, *3*)]
  by *eval*

lemma *bmssp-line-graph-fixed*:
  bmssp-distances [(*0*::*nat*, *1*::*nat*, *1*::*nat*), (*1*, *2*, *1*), (*2*, *3*, *1*)] *0*
    = [(*0*, *0*), (*1*, *1*), (*2*, *2*), (*3*, *3*)]
  by *eval*

lemma *bmssp-empty-prefix-progress-fixed*:
  bmssp-distances
    [(*0*::*nat*, *1*::*nat*, *1*::*nat*), (*0*, *2*, *2*), (*0*, *3*, *3*), (*2*, *4*, *1*)] *0*
    = [(*0*, *0*), (*1*, *1*), (*2*, *2*), (*3*, *3*), (*4*, *3*)]
  by *eval*

The theorem *bmssp-distances example-graph 0 = example-expected-dist* is the end-to-end smoke test. The regression lemmas above cover equal-distance frontier updates, a multi-hop line graph, and progress after a pull leaves an empty bucket in front of pending vertices. The proof method *eval* runs the generated equations inside Isabelle and proves that the result equals the expected literal distance map. The following **value** command is intentionally kept in the theory so the build log prints the computed list for human inspection, and the final **export-code** declaration emits the same executable entry point as SML in the generated directory.

value *bmssp-distances example-graph 0*

export-code *bmssp-distances example-graph example-expected-dist*
  in *SML* **module-name** *BMSSP* **file-prefix** *generated/BMSSP*

end
theory *BMSSP-Executable-Base-Case*
  imports *BMSSP-Base-Case BMSSP-Code-Export HOL−Library.Multiset*
begin

17 Executable Base-Case Ordering

The semantic base-case theory orders a finite set by *finite-weighted-digraph.dist*. The functions below provide an executable finite counterpart: enumerate simple walks over explicit vertex and edge lists, compute their

weights by a list minimum, and sort the target vertices by the resulting finite distance scan.

```
fun exec-walk :: 'v list ⇒ ('v × 'v) list ⇒ 'v list ⇒ bool where
  exec-walk vs es [] ⟷ False
| exec-walk vs es [x] ⟷ x ∈ set vs
| exec-walk vs es (x # y # xs) ⟷
    x ∈ set vs ∧ (x, y) ∈ set es ∧ exec-walk vs es (y # xs)

fun exec-walk-weight :: ('v ⇒ 'v ⇒ real) ⇒ 'v list ⇒ real where
  exec-walk-weight W [] = 0
| exec-walk-weight W [x] = 0
| exec-walk-weight W (x # y # xs) = W x y + exec-walk-weight W (y # xs)

definition exec-simple-walks-betw ::
  'v list ⇒ ('v × 'v) list ⇒ 'v ⇒ 'v ⇒ 'v list list where
  exec-simple-walks-betw vs es a b =
    filter (λp. p ≠ [] ∧ hd p = a ∧ last p = b ∧
      exec-walk vs es p ∧ distinct p)
      (concat (map (λn. List.n-lists n vs) [1..<Suc (length vs)]))

definition exec-walk-weights ::
  'v list ⇒ ('v × 'v) list ⇒ ('v ⇒ 'v ⇒ real) ⇒ 'v ⇒ 'v ⇒ real list where
  exec-walk-weights vs es W a b =
    map (exec-walk-weight W) (exec-simple-walks-betw vs es a b)

definition exec-dist ::
  'v list ⇒ ('v × 'v) list ⇒ ('v ⇒ 'v ⇒ real) ⇒ 'v ⇒ 'v ⇒ real where
  exec-dist vs es W a b =
    (case exec-walk-weights vs es W a b of [] ⇒ 0 | ws ⇒ min-list ws)

definition exec-closest-vertices ::
  'v list ⇒ ('v × 'v) list ⇒ ('v ⇒ 'v ⇒ real) ⇒ 'v ⇒ 'v list ⇒ 'v list where
  exec-closest-vertices vs es W src xs =
    sort-key (exec-dist vs es W src) (remdups xs)

definition exec-reachable ::
  'v list ⇒ ('v × 'v) list ⇒ 'v ⇒ 'v ⇒ bool where
  exec-reachable vs es a b ⟷ exec-simple-walks-betw vs es a b ≠ []

definition exec-shortest-through ::
  'v list ⇒ ('v × 'v) list ⇒ ('v ⇒ 'v ⇒ real) ⇒ 'v ⇒ 'v ⇒ 'v ⇒ bool where
  exec-shortest-through vs es W src x v ⟷
    (∃ p∈set (exec-simple-walks-betw vs es src v).
       exec-walk-weight W p = exec-dist vs es W src v ∧ x ∈ set p)

definition exec-bound-tree-vertices ::
  'v list ⇒ ('v × 'v) list ⇒ ('v ⇒ 'v ⇒ real) ⇒ 'v ⇒ 'v ⇒ bound ⇒ 'v list where
  exec-bound-tree-vertices vs es W src x B =
    filter (λv. exec-reachable vs es src v ∧
```

```
    exec-shortest-through vs es W src x v ∧
    below-bound (exec-dist vs es W src v) B)
    (remdups vs)

definition exec-base-case-order ::
  ('v ⇒ 'v ⇒ real) ⇒ 'v ⇒ 'v list ⇒ ('v × 'v) list ⇒
    'v ⇒ bound ⇒ 'v list where
  exec-base-case-order W src vs es x B =
     exec-closest-vertices vs es W src (exec-bound-tree-vertices vs es W src x B)

definition exec-base-case-vertices ::
  ('v ⇒ 'v ⇒ real) ⇒ 'v ⇒ 'v list ⇒ ('v × 'v) list ⇒
    nat ⇒ 'v ⇒ bound ⇒ 'v set where
  exec-base-case-vertices W src vs es k x B =
     (let xs = exec-base-case-order W src vs es x B in
       if length xs ≤ k then set xs
       else set (filter (λv. exec-dist vs es W src v < exec-dist vs es W src (xs ! k))
             (take (Suc k) xs)))

definition exec-base-case-bound ::
  ('v ⇒ 'v ⇒ real) ⇒ 'v ⇒ 'v list ⇒ ('v × 'v) list ⇒
    nat ⇒ 'v ⇒ bound ⇒ bound where
  exec-base-case-bound W src vs es k x B =
     (let xs = exec-base-case-order W src vs es x B in
       if length xs ≤ k then B else Fin (exec-dist vs es W src (xs ! k)))

definition exec-base-case-result ::
  ('v ⇒ 'v ⇒ real) ⇒ 'v ⇒ 'v list ⇒ ('v × 'v) list ⇒
    nat ⇒ 'v ⇒ bound ⇒ bound × 'v set where
  exec-base-case-result W src vs es k x B =
     (exec-base-case-bound W src vs es k x B,
      exec-base-case-vertices W src vs es k x B)

context finite-weighted-digraph
begin

lemma exec-walk-iff-walk:
  assumes set vs = V set es = E
  shows exec-walk vs es p ⟷ walk p
  using assms by (induction p rule: exec-walk.induct) simp-all

lemma exec-walk-weight-eq-walk-weight [simp]:
  exec-walk-weight w p = walk-weight p
  by (induction p rule: walk-weight.induct) simp-all

lemma set-exec-simple-walks-betw:
  assumes vs: set vs = V
    and es: set es = E
  shows set (exec-simple-walks-betw vs es a b) = {p. simple-walk-betw a p b}
```

```
proof
  show set (exec-simple-walks-betw vs es a b) ⊆ {p. simple-walk-betw a p b}
    unfolding exec-simple-walks-betw-def simple-walk-betw-def walk-betw-def
    using exec-walk-iff-walk[OF vs es] by auto
next
  show {p. simple-walk-betw a p b} ⊆ set (exec-simple-walks-betw vs es a b)
  proof
    fix p
    assume p: p ∈ {p. simple-walk-betw a p b}
    then have simple: simple-walk-betw a p b
      by simp
    then have walk-p: walk p and distinct-p: distinct p
      and p-ne: p ≠ [] and hd-p: hd p = a and last-p: last p = b
      unfolding simple-walk-betw-def walk-betw-def by auto
    have len-le-card: length p ≤ card V
      by (rule simple-walk-length-le-card[OF simple])
    have card-le-len: card V ≤ length vs
      using card-length[of vs] vs by simp
    have len-le: length p ≤ length vs
      using len-le-card card-le-len by linarith
    have len-pos: 0 < length p
      using p-ne by (cases p) auto
    have p-in-lists: p ∈ set (List.n-lists (length p) vs)
      using walk-set-subset[OF walk-p] vs
      by (auto simp: set-n-lists)
    have len-mem: length p ∈ set [1..<Suc (length vs)]
    proof −
      have 1 ≤ length p
        using len-pos by linarith
      moreover have length p < Suc (length vs)
        using len-le by simp
      ultimately show ?thesis
        by auto
    qed
    have enum: p ∈ set (concat (map (λn. List.n-lists n vs) [1..<Suc (length vs)]))
      using p-in-lists len-mem by auto
    have exec: exec-walk vs es p
      using exec-walk-iff-walk[OF vs es] walk-p by simp
    show p ∈ set (exec-simple-walks-betw vs es a b)
      unfolding exec-simple-walks-betw-def
      using enum exec distinct-p hd-p last-p p-ne by auto
  qed
qed

lemma set-exec-walk-weights:
  assumes vs: set vs = V
    and es: set es = E
  shows set (exec-walk-weights vs es w a b) = simple-walk-weights a b
  unfolding exec-walk-weights-def simple-walk-weights-def
```

using *set-exec-simple-walks-betw*[*OF vs es*, *of a b*] **by** *auto*

lemma *exec-walk-weights-nonempty*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
and *reach*: *reachable a b*
shows *exec-walk-weights vs es w a b* ≠ []
proof
assume *empty*: *exec-walk-weights vs es w a b* = []
then have *simple-walk-weights a b* = {}
using *set-exec-walk-weights*[*OF vs es*, *of a b*] **by** *simp*
then show *False*
using *simple-walk-weights-nonempty*[*OF reach*] **by** *simp*
qed

lemma *exec-dist-eq-dist*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
and *reach*: *reachable a b*
shows *exec-dist vs es w a b* = *dist a b*
proof −
have *nonempty*: *exec-walk-weights vs es w a b* ≠ []
using *exec-walk-weights-nonempty*[*OF vs es reach*] **.**
have *exec-dist vs es w a b* = *min-list* (*exec-walk-weights vs es w a b*)
using *nonempty* **unfolding** *exec-dist-def* **by** (*cases exec-walk-weights vs es w a b*) *simp-all*
also have ... = *Min* (*set* (*exec-walk-weights vs es w a b*))
using *min-list-Min*[*OF nonempty*] **.**
also have ... = *Min* (*simple-walk-weights a b*)
using *set-exec-walk-weights*[*OF vs es*, *of a b*] **by** *simp*
also have ... = *dist a b*
unfolding *dist-def* **by** *simp*
finally show *?thesis* **.**
qed

lemma *exec-closest-vertices-properties*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
and *xs*: *set xs* = *A*
and *reach-A*: *A* ⊆ {*v*. *reachable s v*}
shows *set* (*exec-closest-vertices vs es w s xs*) = *A*
and *distinct* (*exec-closest-vertices vs es w s xs*)
and *sorted-wrt* (λ*u v*. *dist s u* ≤ *dist s v*)
(*exec-closest-vertices vs es w s xs*)
proof −
let *?ys* = *exec-closest-vertices vs es w s xs*
have *set-ys*: *set ?ys* = *A*
unfolding *exec-closest-vertices-def* **using** *xs* **by** *simp*
then show *set ?ys* = *A* **.**

show *distinct ?ys*
unfolding *exec-closest-vertices-def* **by** *simp*
have *eq-on*: $\bigwedge v.\ v \in set\ ?ys \Longrightarrow exec\text{-}dist\ vs\ es\ w\ s\ v = dist\ s\ v$
using *exec-dist-eq-dist*[*OF vs es*] *reach-A set-ys* **by** *blast*
have *sorted* (*map* (*exec-dist vs es w s*) *?ys*)
unfolding *exec-closest-vertices-def* **by** (*rule sorted-sort-key*)
then have *sorted* (*map* (*dist s*) *?ys*)
by (*simp add*: *map-cong*[*OF refl eq-on*])
then show *sorted-wrt* ($\lambda u\ v.\ dist\ s\ u \leq dist\ s\ v$) *?ys*
by (*simp add*: *sorted-map*)
qed

lemma *closest-vertices-executable*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
and *xs*: *set xs* = *A*
and *reach-A*: $A \subseteq \{v.\ reachable\ s\ v\}$
shows *set* (*exec-closest-vertices vs es w s xs*) = *A*
and *distinct* (*exec-closest-vertices vs es w s xs*)
and *sorted-wrt* ($\lambda u\ v.\ dist\ s\ u \leq dist\ s\ v$)
(*exec-closest-vertices vs es w s xs*)
using *exec-closest-vertices-properties*[*OF assms*] **by** *blast+*

lemma *closest-vertices-eq-exec-if-inj*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
and *xs*: *set xs* = *A*
and *reach-A*: $A \subseteq \{v.\ reachable\ s\ v\}$
and *inj*: *inj-on* (*dist s*) *A*
shows *closest-vertices A* = *exec-closest-vertices vs es w s xs*
proof −
have *finite-A*: *finite A*
using *xs* **by** *auto*
have *cv-set*: *set* (*closest-vertices A*) = *A*
by (*rule closest-vertices-properties*(*1*)[*OF finite-A*])
have *cv-distinct*: *distinct* (*closest-vertices A*)
by (*rule closest-vertices-properties*(*2*)[*OF finite-A*])
have *cv-sorted*: *sorted-wrt* ($\lambda u\ v.\ dist\ s\ u \leq dist\ s\ v$) (*closest-vertices A*)
by (*rule closest-vertices-properties*(*3*)[*OF finite-A*])
have *ex-set*: *set* (*exec-closest-vertices vs es w s xs*) = *A*
by (*rule exec-closest-vertices-properties*(*1*)[*OF vs es xs reach-A*])
have *ex-distinct*: *distinct* (*exec-closest-vertices vs es w s xs*)
by (*rule exec-closest-vertices-properties*(*2*)[*OF vs es xs reach-A*])
have *ex-sorted*:
sorted-wrt ($\lambda u\ v.\ dist\ s\ u \leq dist\ s\ v$)
(*exec-closest-vertices vs es w s xs*)
by (*rule exec-closest-vertices-properties*(*3*)[*OF vs es xs reach-A*])
have *inj-union*:
inj-on (*dist s*)

```
    (set (closest-vertices A) ∪ set (exec-closest-vertices vs es w s xs))
   using inj cv-set ex-set by simp
  have cv-sorted-map: sorted (map (dist s) (closest-vertices A))
   using cv-sorted by (simp add: sorted-map)
  have ex-sorted-map: sorted (map (dist s) (exec-closest-vertices vs es w s xs))
   using ex-sorted by (simp add: sorted-map)
  have cv-distinct-map: distinct (map (dist s) (closest-vertices A))
   using cv-distinct inj cv-set by (simp add: distinct-map)
  have ex-distinct-map: distinct (map (dist s) (exec-closest-vertices vs es w s xs))
   using ex-distinct inj ex-set by (simp add: distinct-map)
  show ?thesis
   by (rule map-sorted-distinct-set-unique
    [OF inj-union cv-sorted-map cv-distinct-map ex-sorted-map ex-distinct-map])
    (simp add: cv-set ex-set)
qed

lemma exec-reachable-iff-reachable:
  assumes vs: set vs = V
   and es: set es = E
  shows exec-reachable vs es a b ⟷ reachable a b
proof −
  have exec-reachable vs es a b ⟷
    set (exec-simple-walks-betw vs es a b) ≠ {}
   unfolding exec-reachable-def by auto
  also have ... ⟷ {p. simple-walk-betw a p b} ≠ {}
   using set-exec-simple-walks-betw[OF vs es, of a b] by simp
  also have ... ⟷ reachable a b
   unfolding reachable-def by auto
  finally show ?thesis .
qed

lemma exec-shortest-through-iff-through-single:
  assumes vs: set vs = V
   and es: set es = E
  shows exec-shortest-through vs es w s x v ⟷ through {x} v
proof
  assume exec: exec-shortest-through vs es w s x v
  then obtain p where p-exec:
    p ∈ set (exec-simple-walks-betw vs es s v)
    exec-walk-weight w p = exec-dist vs es w s v
    x ∈ set p
   unfolding exec-shortest-through-def by blast
  have p-simple: simple-walk-betw s p v
   using p-exec(1) set-exec-simple-walks-betw[OF vs es, of s v] by blast
  then have reach-v: reachable s v
   unfolding reachable-def by blast
  have walk-weight p = dist s v
   using p-exec(2) exec-dist-eq-dist[OF vs es reach-v] by simp
  then have shortest-walk s p v
```

```
    using p-simple unfolding shortest-walk-def by blast
  then show through {x} v
    using p-exec(3) unfolding through-def by blast
next
  assume through: through {x} v
  then obtain p where p: shortest-walk s p v x ∈ set p
    unfolding through-def by blast
  have p-simple: simple-walk-betw s p v
    using p(1) unfolding shortest-walk-def by blast
  have reach-v: reachable s v
    using p-simple unfolding reachable-def by blast
  have p ∈ set (exec-simple-walks-betw vs es s v)
    using p-simple set-exec-simple-walks-betw[OF vs es, of s v] by blast
  moreover have exec-walk-weight w p = exec-dist vs es w s v
    using p(1) exec-dist-eq-dist[OF vs es reach-v]
    unfolding shortest-walk-def by simp
  ultimately show exec-shortest-through vs es w s x v
    using p(2) unfolding exec-shortest-through-def by blast
qed

lemma set-exec-bound-tree-vertices:
  assumes vs: set vs = V
    and es: set es = E
  shows set (exec-bound-tree-vertices vs es w s x B) = bound-tree {x} B
proof
  show set (exec-bound-tree-vertices vs es w s x B) ⊆ bound-tree {x} B
  proof
    fix v
    assume v: v ∈ set (exec-bound-tree-vertices vs es w s x B)
    then have vV: v ∈ V
      using vs unfolding exec-bound-tree-vertices-def by auto
    have reach-v: reachable s v
      using v exec-reachable-iff-reachable[OF vs es, of s v]
      unfolding exec-bound-tree-vertices-def by auto
    have through-v: through {x} v
      using v exec-shortest-through-iff-through-single[OF vs es, of x v]
      unfolding exec-bound-tree-vertices-def by auto
    have below-v: below-bound (dist s v) B
      using v exec-dist-eq-dist[OF vs es reach-v]
      unfolding exec-bound-tree-vertices-def by auto
    show v ∈ bound-tree {x} B
      using vV reach-v through-v below-v unfolding bound-tree-def by blast
  qed
next
  show bound-tree {x} B ⊆ set (exec-bound-tree-vertices vs es w s x B)
  proof
    fix v
    assume v: v ∈ bound-tree {x} B
    then have vV: v ∈ V and reach-v: reachable s v
```

```
    and through-v: through {x} v and below-v: below-bound (dist s v) B
    unfolding bound-tree-def by auto
  have exec-reachable vs es s v
    using exec-reachable-iff-reachable[OF vs es, of s v] reach-v by simp
  moreover have exec-shortest-through vs es w s x v
    using exec-shortest-through-iff-through-single[OF vs es, of x v] through-v by
simp
  moreover have below-bound (exec-dist vs es w s v) B
    using below-v exec-dist-eq-dist[OF vs es reach-v] by simp
  ultimately show v ∈ set (exec-bound-tree-vertices vs es w s x B)
    using vV vs unfolding exec-bound-tree-vertices-def by auto
 qed
qed

definition base-case-order-impl ::
 'v list ⇒ ('v × 'v) list ⇒ 'v ⇒ bound ⇒ 'v list where
 base-case-order-impl vs es x B =
   exec-closest-vertices vs es w s (exec-bound-tree-vertices vs es w s x B)

definition base-case-vertices-impl ::
 'v list ⇒ ('v × 'v) list ⇒ nat ⇒ 'v ⇒ bound ⇒ 'v set where
 base-case-vertices-impl vs es k x B =
   (let xs = base-case-order-impl vs es x B in
     if length xs ≤ k then set xs
     else set (filter (λv. exec-dist vs es w s v < exec-dist vs es w s (xs ! k))
           (take (Suc k) xs)))

definition base-case-bound-impl ::
 'v list ⇒ ('v × 'v) list ⇒ nat ⇒ 'v ⇒ bound ⇒ bound where
 base-case-bound-impl vs es k x B =
   (let xs = base-case-order-impl vs es x B in
     if length xs ≤ k then B else Fin (exec-dist vs es w s (xs ! k)))

definition base-case-result-impl ::
 'v list ⇒ ('v × 'v) list ⇒ nat ⇒ 'v ⇒ bound ⇒ bound × 'v set where
 base-case-result-impl vs es k x B =
   (base-case-bound-impl vs es k x B, base-case-vertices-impl vs es k x B)

declare base-case-order-impl-def [code]
declare base-case-vertices-impl-def [code]
declare base-case-bound-impl-def [code]
declare base-case-result-impl-def [code]

lemma base-case-order-impl-set:
 assumes vs: set vs = V
   and es: set es = E
 shows set (base-case-order-impl vs es x B) = bound-tree {x} B
 unfolding base-case-order-impl-def
 by (rule exec-closest-vertices-properties(1)
```

```
    [OF vs es set-exec-bound-tree-vertices[OF vs es]])
  (auto simp: bound-tree-def)

lemma base-case-order-impl-eq-if-inj:
  assumes vs: set vs = V
    and es: set es = E
    and inj: inj-on (dist s) (bound-tree {x} B)
  shows base-case-order x B = base-case-order-impl vs es x B
  unfolding base-case-order-def base-case-order-impl-def
  by (rule closest-vertices-eq-exec-if-inj
      [OF vs es set-exec-bound-tree-vertices[OF vs es]])
    (use inj in ‹auto simp: bound-tree-def›)

lemma base-case-order-impl-distinct:
  distinct (base-case-order-impl vs es x B)
  unfolding base-case-order-impl-def exec-closest-vertices-def by simp

lemma base-case-order-impl-sorted:
  assumes vs: set vs = V
    and es: set es = E
  shows sorted-wrt (λu v. dist s u ≤ dist s v) (base-case-order-impl vs es x B)
  unfolding base-case-order-impl-def
  by (rule exec-closest-vertices-properties(3)
      [OF vs es set-exec-bound-tree-vertices[OF vs es]])
    (auto simp: bound-tree-def)

lemma base-case-order-impl-length-eq:
  assumes vs: set vs = V
    and es: set es = E
  shows length (base-case-order-impl vs es x B) = length (base-case-order x B)
proof −
  have length (base-case-order-impl vs es x B) =
      card (set (base-case-order-impl vs es x B))
    using base-case-order-impl-distinct[of vs es x B] by (simp add: distinct-card)
  also have ... = card (bound-tree {x} B)
    using base-case-order-impl-set[OF vs es, of x B] by simp
  also have ... = card (set (base-case-order x B))
    using base-case-order-set[of x B] by simp
  also have ... = length (base-case-order x B)
    using base-case-order-distinct[of x B] by (simp add: distinct-card)
  finally show ?thesis .
qed

lemma dist-map-eq-if-same-sorted-set:
  assumes sorted-x: sorted-wrt (λu v. dist s u ≤ dist s v) xs
    and sorted-y: sorted-wrt (λu v. dist s u ≤ dist s v) ys
    and distinct-x: distinct xs
    and distinct-y: distinct ys
    and set-eq: set xs = set ys
```

```
  shows map (dist s) xs = map (dist s) ys
proof −
  have sorted-x': sorted (map (dist s) xs)
    using sorted-x by (simp add: sorted-map)
  have sorted-y': sorted (map (dist s) ys)
    using sorted-y by (simp add: sorted-map)
  have mset xs = mset ys
    using distinct-x distinct-y set-eq
    by (simp add: set-eq-iff-mset-eq-distinct)
  then have mset-dist: mset (map (dist s) xs) = mset (map (dist s) ys)
    by simp
  have sort (map (dist s) ys) = map (dist s) xs
    by (rule properties-for-sort[OF mset-dist sorted-x'])
  moreover have sort (map (dist s) ys) = map (dist s) ys
    using sorted-y' by (simp add: sort-key-id-if-sorted)
  ultimately show ?thesis
    by simp
qed

lemma base-case-order-impl-kth-dist-eq:
  assumes vs: set vs = V
    and es: set es = E
    and k: k < length (base-case-order x B)
  shows dist s ((base-case-order x B) ! k) =
    dist s ((base-case-order-impl vs es x B) ! k)
proof −
  let ?xs = base-case-order x B
  let ?ys = base-case-order-impl vs es x B
  have len-y: k < length ?ys
    using k base-case-order-impl-length-eq[OF vs es, of x B] by simp
  have map-eq: map (dist s) ?xs = map (dist s) ?ys
    by (rule dist-map-eq-if-same-sorted-set)
      (use base-case-order-sorted base-case-order-impl-sorted[OF vs es]
        base-case-order-distinct base-case-order-impl-distinct
        base-case-order-set base-case-order-impl-set[OF vs es] in auto)
  have map (dist s) ?xs ! k = map (dist s) ?ys ! k
    using map-eq by simp
  then show ?thesis
    using k len-y by simp
qed

lemma exec-dist-eq-dist-on-base-case-order-impl:
  assumes vs: set vs = V
    and es: set es = E
    and v: v ∈ set (base-case-order-impl vs es x B)
  shows exec-dist vs es w s v = dist s v
proof −
  have v ∈ bound-tree {x} B
    using v base-case-order-impl-set[OF vs es, of x B] by simp
```

```
  then have reachable s v
    unfolding bound-tree-def by blast
  then show ?thesis
    by (rule exec-dist-eq-dist[OF vs es])
qed

lemma base-case-bound-impl-eq:
  assumes vs: set vs = V
    and es: set es = E
  shows base-case-bound-impl vs es k x B = base-case-bound k x B
proof (cases length (base-case-order x B) ≤ k)
  case True
  then have length (base-case-order-impl vs es x B) ≤ k
    using base-case-order-impl-length-eq[OF vs es, of x B] by simp
  then show ?thesis
    using True unfolding base-case-bound-impl-def base-case-bound-def by (simp
add: Let-def)
next
  case False
  then have k: k < length (base-case-order x B)
    by simp
  let ?ys = base-case-order-impl vs es x B
  have len-y: k < length ?ys
    using k base-case-order-impl-length-eq[OF vs es, of x B] by simp
  have y-mem: ?ys ! k ∈ set ?ys
    using len-y by simp
  have exec-eq: exec-dist vs es w s (?ys ! k) = dist s (?ys ! k)
    using exec-dist-eq-dist-on-base-case-order-impl[OF vs es y-mem] .
  have dist-eq: dist s ((base-case-order x B) ! k) = dist s (?ys ! k)
    by (rule base-case-order-impl-kth-dist-eq[OF vs es k])
  show ?thesis
    using False len-y exec-eq dist-eq
    unfolding base-case-bound-impl-def base-case-bound-def
    by (simp add: Let-def)
qed

lemma base-case-vertices-impl-success:
  assumes vs: set vs = V
    and es: set es = E
    and len: length (base-case-order-impl vs es x B) ≤ k
  shows base-case-vertices-impl vs es k x B = bound-tree {x} B
  using len base-case-order-impl-set[OF vs es, of x B]
  unfolding base-case-vertices-impl-def by (simp add: Let-def)

lemma base-case-vertices-impl-partial:
  assumes vs: set vs = V
    and es: set es = E
    and len: k < length (base-case-order-impl vs es x B)
  shows base-case-vertices-impl vs es k x B =
```

```
    bound-tree {x} (Fin (exec-dist vs es w s ((base-case-order-impl vs es x B) ! k)))
proof −
  let ?xs = base-case-order-impl vs es x B
  let ?b = dist s (?xs ! k)
  have set-xs: set ?xs = bound-tree {x} B
    using base-case-order-impl-set[OF vs es, of x B] .
  have sorted: sorted-wrt (λu v. dist s u ≤ dist s v) ?xs
    using base-case-order-impl-sorted[OF vs es, of x B] .
  have kth-in: ?xs ! k ∈ bound-tree {x} B
    using len set-xs nth-mem by metis
  have kth-exec: exec-dist vs es w s (?xs ! k) = ?b
    using exec-dist-eq-dist-on-base-case-order-impl[OF vs es] len nth-mem by blast
  have exec-on-xs: ⋀v. v ∈ set ?xs ⟹ exec-dist vs es w s v = dist s v
    by (rule exec-dist-eq-dist-on-base-case-order-impl[OF vs es])
  have exec-on-take: ⋀v. v ∈ set (take (Suc k) ?xs) ⟹
      exec-dist vs es w s v = dist s v
  proof −
    fix v
    assume v ∈ set (take (Suc k) ?xs)
    then have v ∈ set ?xs
      by (rule in-set-takeD)
    then show exec-dist vs es w s v = dist s v
      by (rule exec-on-xs)
  qed
  have vertices-eq:
    base-case-vertices-impl vs es k x B =
      {v ∈ set (take (Suc k) ?xs). dist s v < ?b}
    using len kth-exec exec-on-take
    unfolding base-case-vertices-impl-def by (auto simp: Let-def)
  have base-case-vertices-impl vs es k x B = bound-tree {x} (Fin ?b)
  proof
    show base-case-vertices-impl vs es k x B ⊆ bound-tree {x} (Fin ?b)
    proof
      fix v
      assume v ∈ base-case-vertices-impl vs es k x B
      then have v: v ∈ set (take (Suc k) ?xs) dist s v < ?b
        using vertices-eq by auto
      have v ∈ set ?xs
        using v(1) by (meson in-set-takeD)
      then have v ∈ bound-tree {x} B
        using set-xs by simp
      then show v ∈ bound-tree {x} (Fin ?b)
        using v(2) unfolding bound-tree-def by auto
    qed
  next
    show bound-tree {x} (Fin ?b) ⊆ base-case-vertices-impl vs es k x B
    proof
      fix v
      assume v: v ∈ bound-tree {x} (Fin ?b)
```

```
    then have lt: dist s v < ?b
      unfolding bound-tree-def by auto
    have kth-below: below-bound (dist s (?xs ! k)) B
      using kth-in unfolding bound-tree-def by auto
    have below-bound (dist s v) B
      using below-bound-less-trans[OF lt kth-below] .
    then have v ∈ bound-tree {x} B
      using v unfolding bound-tree-def by auto
    then have v ∈ set ?xs
      using set-xs by simp
    then have v ∈ set (take k ?xs)
      using in-set-take-dist-lt-nth[OF sorted - lt len] by blast
    then have v ∈ set (take (Suc k) ?xs)
      using set-take-subset-set-take[of k Suc k ?xs] by auto
    then show v ∈ base-case-vertices-impl vs es k x B
      using vertices-eq lt by simp
  qed
 qed
 then show ?thesis
  using kth-exec by simp
qed

lemma base-case-vertices-impl-eq:
 assumes vs: set vs = V
  and es: set es = E
 shows base-case-vertices-impl vs es k x B = base-case-vertices k x B
proof (cases length (base-case-order x B) ≤ k)
 case True
 then have len-impl: length (base-case-order-impl vs es x B) ≤ k
  using base-case-order-impl-length-eq[OF vs es, of x B] by simp
 show ?thesis
  using base-case-success[OF True] base-case-vertices-impl-success[OF vs es len-
impl] by simp
next
 case False
 then have k: k < length (base-case-order x B)
  by simp
 have len-impl: k < length (base-case-order-impl vs es x B)
  using k base-case-order-impl-length-eq[OF vs es, of x B] by simp
 let ?ys = base-case-order-impl vs es x B
 have y-mem: ?ys ! k ∈ set ?ys
  using len-impl by simp
 have exec-eq: exec-dist vs es w s (?ys ! k) = dist s (?ys ! k)
  using exec-dist-eq-dist-on-base-case-order-impl[OF vs es y-mem] .
 have dist-eq: dist s ((base-case-order x B) ! k) = dist s (?ys ! k)
  by (rule base-case-order-impl-kth-dist-eq[OF vs es k])
 have impl: base-case-vertices-impl vs es k x B =
    bound-tree {x} (Fin (exec-dist vs es w s (?ys ! k)))
  by (rule base-case-vertices-impl-partial[OF vs es len-impl])
```

have *sem*: *base-case-vertices k x B* =
bound-tree {*x*} (*Fin* (*dist s* ((*base-case-order x B*) ! *k*)))
by (*rule base-case-partial*[*OF k*])
show *?thesis*
using *impl sem exec-eq dist-eq* **by** *simp*
qed

lemma *base-case-result-impl-eq*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
shows *base-case-result-impl vs es k x B* = *base-case-result k x B*
using *base-case-bound-impl-eq*[*OF vs es*, *of k x B*]
base-case-vertices-impl-eq[*OF vs es*, *of k x B*]
unfolding *base-case-result-impl-def base-case-result-def* **by** *simp*

lemma *base-case-result-impl-bmssp-post*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
and *S*: *S* = {*x*}
shows *case base-case-result-impl vs es k x B of* (*B′*, *U*) ⇒
bmssp-post d S B (λ*v*. *if v* ∈ *U then dist s v else d v*) *B′ U*
using *base-case-result-bmssp-post*[*OF S*, **where** *k*=*k* **and** *B*=*B* **and** *d*=*d*]
base-case-result-impl-eq[*OF vs es*, *of k x B*]
by *simp*

lemma *base-case-order-impl-eq-exec-base-case-order*:
base-case-order-impl vs es x B = *exec-base-case-order w s vs es x B*
unfolding *base-case-order-impl-def exec-base-case-order-def* **by** *simp*

lemma *base-case-vertices-impl-eq-exec-base-case-vertices*:
base-case-vertices-impl vs es k x B = *exec-base-case-vertices w s vs es k x B*
unfolding *base-case-vertices-impl-def exec-base-case-vertices-def*
base-case-order-impl-eq-exec-base-case-order **by** *simp*

lemma *base-case-bound-impl-eq-exec-base-case-bound*:
base-case-bound-impl vs es k x B = *exec-base-case-bound w s vs es k x B*
unfolding *base-case-bound-impl-def exec-base-case-bound-def*
base-case-order-impl-eq-exec-base-case-order **by** *simp*

lemma *base-case-result-impl-eq-exec-base-case-result*:
base-case-result-impl vs es k x B = *exec-base-case-result w s vs es k x B*
unfolding *base-case-result-impl-def exec-base-case-result-def*
base-case-bound-impl-eq-exec-base-case-bound
base-case-vertices-impl-eq-exec-base-case-vertices **by** *simp*

lemma *exec-base-case-bound-eq*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
shows *exec-base-case-bound w s vs es k x B* = *base-case-bound k x B*

using *base-case-bound-impl-eq*[*OF vs es*, *of k x B*]
unfolding *base-case-bound-impl-eq-exec-base-case-bound* .

lemma *exec-base-case-vertices-eq*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
shows *exec-base-case-vertices w s vs es k x B* = *base-case-vertices k x B*
using *base-case-vertices-impl-eq*[*OF vs es*, *of k x B*]
unfolding *base-case-vertices-impl-eq-exec-base-case-vertices* .

lemma *exec-base-case-result-eq*:
assumes *vs*: *set vs* = *V*
and *es*: *set es* = *E*
shows *exec-base-case-result w s vs es k x B* = *base-case-result k x B*
using *base-case-result-impl-eq*[*OF vs es*, *of k x B*]
unfolding *base-case-result-impl-eq-exec-base-case-result* .

lemmas *base-case-bound-exec-code* [*code*] = *exec-base-case-bound-eq*[*symmetric*]
lemmas *base-case-vertices-exec-code* [*code*] = *exec-base-case-vertices-eq*[*symmetric*]
lemmas *base-case-result-exec-code* [*code*] = *exec-base-case-result-eq*[*symmetric*]

end

declare *finite-weighted-digraph.base-case-bound-exec-code* [*code*]
declare *finite-weighted-digraph.base-case-vertices-exec-code* [*code*]
declare *finite-weighted-digraph.base-case-result-exec-code* [*code*]

definition *example-edges* :: (*nat* × *nat*) *list* **where**
example-edges = *map* (λ(*u*, *v*, *c*). (*u*, *v*)) *example-graph*

definition *example-vertices* :: *nat list* **where**
example-vertices = *bmssp-vertices example-graph 0*

definition *example-V* :: *nat set* **where**
example-V = *set example-vertices*

definition *example-E* :: (*nat* × *nat*) *set* **where**
example-E = *set example-edges*

definition *example-weight* :: *nat* ⇒ *nat* ⇒ *real* **where**
example-weight u v =
(*case map-of* (*map* (λ(*a*, *b*, *c*). ((*a*, *b*), *real c*)) *example-graph*) (*u*, *v*) *of*
None ⇒ *0*
| *Some c* ⇒ *c*)

lemma *example-vertices-set* [*simp*]: *set example-vertices* = *example-V*
unfolding *example-vertices-def example-V-def* **by** *simp*

lemma *example-edges-set* [*simp*]: *set example-edges* = *example-E*

unfolding *example-edges-def example-E-def* **by** *simp*

lemma *example-finite-weighted-digraph*:
finite-weighted-digraph example-V example-E example-weight 0
unfolding *finite-weighted-digraph-def*
example-V-def example-E-def example-vertices-def example-edges-def
bmssp-vertices-def example-graph-def example-weight-def
by (*auto split*: *prod.splits option.splits*)

interpretation *example-fw*: *finite-weighted-digraph example-V example-E example-weight 0*
by (*rule example-finite-weighted-digraph*)

lemma *example-exec-simple-walk-0*:
[*0*] ∈ *set* (*exec-simple-walks-betw example-vertices example-edges 0 0*)
by *eval*

lemma *example-exec-simple-walk-1*:
[*0*, *1*] ∈ *set* (*exec-simple-walks-betw example-vertices example-edges 0 1*)
by *eval*

lemma *example-exec-simple-walk-2*:
[*0*, *1*, *2*] ∈ *set* (*exec-simple-walks-betw example-vertices example-edges 0 2*)
by *eval*

lemma *example-exec-simple-walk-3*:
[*0*, *1*, *2*, *3*] ∈ *set* (*exec-simple-walks-betw example-vertices example-edges 0 3*)
by *eval*

lemma *example-exec-simple-walk-4*:
[*0*, *4*] ∈ *set* (*exec-simple-walks-betw example-vertices example-edges 0 4*)
by *eval*

lemma *example-exec-reachable-if-in-V*:
assumes *v* ∈ *example-V*
shows *exec-reachable example-vertices example-edges 0 v*
proof −
have *v* = *0* ∨ *v* = *1* ∨ *v* = *2* ∨ *v* = *3* ∨ *v* = *4*
using *assms*
unfolding *example-V-def example-vertices-def bmssp-vertices-def example-graph-def*
by *auto*
then show *?thesis*
using *example-exec-simple-walk-0 example-exec-simple-walk-1*
example-exec-simple-walk-2 example-exec-simple-walk-3
example-exec-simple-walk-4
unfolding *exec-reachable-def* **by** *auto*
qed

lemma *example-dist-inj-on-V*:
inj-on (*finite-weighted-digraph.dist example-V example-E example-weight 0*) *example-V*
proof (*rule inj-onI*)
fix *u v*
assume *uV*: *u* ∈ *example-V* **and** *vV*: *v* ∈ *example-V*
and *dist-eq*:
finite-weighted-digraph.dist example-V example-E example-weight 0 u =
finite-weighted-digraph.dist example-V example-E example-weight 0 v
have *reach-u*: *finite-weighted-digraph.reachable example-V example-E 0 u*
using *example-exec-reachable-if-in-V*[*OF uV*]
example-fw.exec-reachable-iff-reachable[*of example-vertices example-edges 0 u*]
by *simp*
have *reach-v*: *finite-weighted-digraph.reachable example-V example-E 0 v*
using *example-exec-reachable-if-in-V*[*OF vV*]
example-fw.exec-reachable-iff-reachable[*of example-vertices example-edges 0 v*]
by *simp*
have *u-eq*:
exec-dist example-vertices example-edges example-weight 0 u =
finite-weighted-digraph.dist example-V example-E example-weight 0 u
using *example-fw.exec-dist-eq-dist*[*OF example-vertices-set example-edges-set reach-u*] **.**
have *v-eq*:
exec-dist example-vertices example-edges example-weight 0 v =
finite-weighted-digraph.dist example-V example-E example-weight 0 v
using *example-fw.exec-dist-eq-dist*[*OF example-vertices-set example-edges-set reach-v*] **.**
have *exec-inj*:
inj-on (*exec-dist example-vertices example-edges example-weight 0*) *example-V*
by *eval*
show *u* = *v*
using *inj-onD*[*OF exec-inj - -*] *uV vV dist-eq u-eq v-eq* **by** *metis*
qed

lemma *example-base-case-order-code* [*code-unfold*]:
finite-weighted-digraph.base-case-order example-V example-E example-weight 0 x B =
exec-base-case-order example-weight 0 example-vertices example-edges x B
proof −
have *tree-subset*:
finite-weighted-digraph.bound-tree example-V example-E example-weight 0 {*x*} *B* ⊆
example-V
unfolding *example-fw.bound-tree-def* **by** *auto*
have *inj*:
inj-on (*finite-weighted-digraph.dist example-V example-E example-weight 0*)
(*finite-weighted-digraph.bound-tree example-V example-E example-weight 0* {*x*} *B*)
by (*rule inj-on-subset*[*OF example-dist-inj-on-V tree-subset*])

```
  have finite-weighted-digraph.base-case-order example-V example-E example-weight
0 x B =
      finite-weighted-digraph.base-case-order-impl example-weight 0
        example-vertices example-edges x B
    using example-fw.base-case-order-impl-eq-if-inj
      [OF example-vertices-set example-edges-set inj] .
  then show ?thesis
    using example-fw.base-case-order-impl-eq-exec-base-case-order
      [of example-vertices example-edges x B]
    by simp
qed

lemma example-base-case-vertices-code [code-unfold]:
  finite-weighted-digraph.base-case-vertices example-V example-E example-weight 0
k x B =
    exec-base-case-vertices example-weight 0 example-vertices example-edges k x B
  using example-fw.exec-base-case-vertices-eq[of example-vertices example-edges k x
B]
  by simp

lemma example-base-case-result-code [code-unfold]:
  finite-weighted-digraph.base-case-result example-V example-E example-weight 0 k x
B =
    exec-base-case-result example-weight 0 example-vertices example-edges k x B
  using example-fw.exec-base-case-result-eq[of example-vertices example-edges k x
B]
  by simp

value exec-closest-vertices
  (bmssp-vertices example-graph 0) example-edges example-weight 0
  (bmssp-vertices example-graph 0)

value exec-base-case-order
  example-weight 0 (bmssp-vertices example-graph 0) example-edges 0 (Fin 100)

value exec-base-case-result
  example-weight 0 (bmssp-vertices example-graph 0) example-edges 3 0 (Fin 100)
::
  bound × nat set

value finite-weighted-digraph.base-case-vertices
  example-V example-E example-weight 0 3 0 (Fin 100) :: nat set

value finite-weighted-digraph.base-case-order
  example-V example-E example-weight 0 0 (Fin 100)

value finite-weighted-digraph.base-case-result
  example-V example-E example-weight 0 3 0 (Fin 100) :: bound × nat set
```

end
theory *BMSSP-Unique-Shortest-Tree*
imports *BMSSP-Shortest-Path-Lemmas*
begin

18 Unique Shortest-Path Tree

After tie-breaking, shortest paths form a tree rooted at the source. This locale makes that assumption explicit and derives the tree notions used by later BMSSP arguments from the graph model.

locale *unique-shortest-digraph* = *finite-weighted-digraph* +
assumes *unique-shortest-walk*:
⟦*shortest-walk s p v*; *shortest-walk s q v*⟧ ⟹ *p* = *q*
begin

definition *shortest-path-to* **where**
shortest-path-to v = (*THE p*. *shortest-walk s p v*)

definition *tree-path* **where**
tree-path u v ⟷ *reachable s v* ∧ *u* ∈ *set* (*shortest-path-to v*)

definition *tree-of* **where**
tree-of u = {*v* ∈ *V*. *tree-path u v*}

definition *tree-set* **where**
tree-set S = {*v* ∈ *V*. ∃ *u*∈*S*. *tree-path u v*}

definition *tree-antichain* **where**
tree-antichain S ⟷ (∀ *u*∈*S*. ∀ *v*∈*S*. *tree-path u v* ⟶ *u* = *v*)

lemma *tree-antichain-subset*:
assumes *anti*: *tree-antichain T*
and *subset*: *S* ⊆ *T*
shows *tree-antichain S*
using *anti subset* **unfolding** *tree-antichain-def* **by** *blast*

lemma *tree-antichain-singleton* [*simp*]:
tree-antichain {*x*}
unfolding *tree-antichain-def* **by** *blast*

lemma *tree-pathD*:
assumes *tree-path u v*
shows *reachable s v u* ∈ *set* (*shortest-path-to v*)
using *assms* **unfolding** *tree-path-def* **by** *auto*

lemma *tree-pathI*:
assumes *reachable s v u* ∈ *set* (*shortest-path-to v*)
shows *tree-path u v*

using *assms* **unfolding** *tree-path-def* **by** *auto*

lemma *shortest-path-to-ex1*:
assumes *reachable s v*
shows $\exists!p.$ *shortest-walk s p v*
proof −
obtain *p* **where** *p*: *shortest-walk s p v*
using *assms* **by** (*rule shortest-walk-exists*)
have $\bigwedge q.$ *shortest-walk s q v* $\Longrightarrow$ $q = p$
using *p unique-shortest-walk* **by** *blast*
then show *?thesis*
using *p* **by** *blast*
qed

lemma *shortest-path-to-shortest*:
assumes *reachable s v*
shows *shortest-walk s* (*shortest-path-to v*) *v*
unfolding *shortest-path-to-def*
using *shortest-path-to-ex1*[*OF assms*] **by** (*rule theI'*)

lemma *shortest-path-to-successively-tight*:
assumes *reachable s v*
shows *successively tight-edge-step* (*shortest-path-to v*)
using *shortest-walk-successively-tight*[*OF shortest-path-to-shortest*[*OF assms*]] **.**

lemma *shortest-path-prefix-eq*:
assumes *reach-v*: *reachable s v*
and *i*: $i <$ *length* (*shortest-path-to v*)
shows *shortest-path-to* (*shortest-path-to v* ! *i*) =
take (*Suc i*) (*shortest-path-to v*)
proof −
let *?p* = *shortest-path-to v*
have *sp*: *shortest-walk s ?p v*
using *shortest-path-to-shortest*[*OF reach-v*] **.**
have *pref-short*: *shortest-walk s* (*take* (*Suc i*) *?p*) (*?p* ! *i*)
using *shortest-walk-prefix-shortest*[*OF sp i*] **.**
have *reach-i*: *reachable s* (*?p* ! *i*)
using *pref-short* **unfolding** *shortest-walk-def reachable-def* **by** *blast*
have *canon-short*: *shortest-walk s* (*shortest-path-to* (*?p* ! *i*)) (*?p* ! *i*)
using *shortest-path-to-shortest*[*OF reach-i*] **.**
show *?thesis*
using *unique-shortest-walk*[*OF canon-short pref-short*] **by** *simp*
qed

lemma *tree-path-between-shortest-path-indices*:
assumes *reach-v*: *reachable s v*
and *ij*: $i \leq j$
and *j*: $j <$ *length* (*shortest-path-to v*)
shows *tree-path* (*shortest-path-to v* ! *i*) (*shortest-path-to v* ! *j*)

```
proof −
  let ?p = shortest-path-to v
  have sp: shortest-walk s ?p v
    using shortest-path-to-shortest[OF reach-v] .
  have pref-j: shortest-walk s (take (Suc j) ?p) (?p ! j)
    using shortest-walk-prefix-shortest[OF sp j] .
  have reach-j: reachable s (?p ! j)
    using pref-j unfolding shortest-walk-def reachable-def by blast
  have i-take: i < length (take (Suc j) ?p)
    using ij j by simp
  have take-i: take (Suc j) ?p ! i = ?p ! i
    using ij by simp
  have ?p ! i ∈ set (take (Suc j) ?p)
    using nth-mem[OF i-take] take-i by metis
  then have in-canon: ?p ! i ∈ set (shortest-path-to (?p ! j))
    using shortest-path-prefix-eq[OF reach-v j] by simp
  show ?thesis
    using reach-j in-canon by (rule tree-pathI)
qed

lemma tree-prefix-of-shortest-path-suffix:
  assumes reach-v: reachable s v
    and i: i < length (shortest-path-to v)
  shows set (take k (drop i (shortest-path-to v))) ⊆
    tree-of (shortest-path-to v ! i)
proof
  let ?p = shortest-path-to v
  fix x
  assume x-take: x ∈ set (take k (drop i ?p))
  obtain r where r: r < length (take k (drop i ?p))
    take k (drop i ?p) ! r = x
    using x-take by (auto simp: in-set-conv-nth)
  have r-drop: r < length (drop i ?p)
    using r(1) by simp
  have j-len: i + r < length ?p
    using r-drop i by simp
  have x-drop: x = drop i ?p ! r
    using r by simp
  have x-eq: x = ?p ! (i + r)
    using x-drop r-drop by simp
  have tree: tree-path (?p ! i) x
    using tree-path-between-shortest-path-indices[OF reach-v - j-len, of i]
      x-eq by simp
  have xV: x ∈ V
  proof −
    have shortest-walk s ?p v
      using shortest-path-to-shortest[OF reach-v] .
    then have walk ?p
      unfolding shortest-walk-def simple-walk-betw-def walk-betw-def by blast
```

```
    moreover have x ∈ set ?p
      using x-eq nth-mem[OF j-len] by simp
    ultimately show ?thesis
      using walk-set-subset by blast
  qed
  show x ∈ tree-of (?p ! i)
    using xV tree unfolding tree-of-def by blast
qed

lemma tree-path-dist-le:
  assumes tree-path u v
  shows dist s u ≤ dist s v
proof −
  have reach: reachable s v and u: u ∈ set (shortest-path-to v)
    using assms by (auto dest: tree-pathD)
  have shortest-walk s (shortest-path-to v) v
    using shortest-path-to-shortest[OF reach] .
  then show ?thesis
    using shortest-walk-prefix-dist-le u by blast
qed

lemma tree-path-root-reachable:
  assumes tree-path u v
  shows reachable s u
proof −
  have reach-v: reachable s v and u-path: u ∈ set (shortest-path-to v)
    using assms by (auto dest: tree-pathD)
  obtain i where i: i < length (shortest-path-to v) shortest-path-to v ! i = u
    using u-path by (auto simp: in-set-conv-nth)
  have sp: shortest-walk s (shortest-path-to v) v
    using shortest-path-to-shortest[OF reach-v] .
  have shortest-walk s (take (Suc i) (shortest-path-to v)) u
    using shortest-walk-prefix-shortest[OF sp i(1)] i(2) by simp
  then show ?thesis
    unfolding shortest-walk-def reachable-def by blast
qed

lemma tree-path-root-in-V:
  assumes tree-path u v
  shows u ∈ V
proof −
  have reachable s u
    using tree-path-root-reachable[OF assms] .
  then obtain p where p: simple-walk-betw s p u
    unfolding reachable-def by blast
  then have walk p and u ∈ set p
    unfolding simple-walk-betw-def walk-betw-def
    by (auto intro: last-in-set)
  then show ?thesis
```

```
    using walk-set-subset by blast
qed

lemma tree-path-trans:
  assumes uv: tree-path u v
    and xu: tree-path x u
  shows tree-path x v
proof −
  have reach-v: reachable s v and u-path: u ∈ set (shortest-path-to v)
    using uv by (auto dest: tree-pathD)
  have x-path-u: x ∈ set (shortest-path-to u)
    using xu by (auto dest: tree-pathD)
  obtain i where i: i < length (shortest-path-to v) shortest-path-to v ! i = u
    using u-path by (auto simp: in-set-conv-nth)
  have shortest-path-to u = take (Suc i) (shortest-path-to v)
    using shortest-path-prefix-eq[OF reach-v i(1)] i(2) by simp
  then have x ∈ set (take (Suc i) (shortest-path-to v))
    using x-path-u by simp
  then have x ∈ set (shortest-path-to v)
    by (meson in-set-takeD)
  then show ?thesis
    by (rule tree-pathI[OF reach-v])
qed

lemma tree-path-comparable:
  assumes ux: tree-path u x
    and vx: tree-path v x
  shows tree-path u v ∨ tree-path v u
proof −
  have reach-x: reachable s x
    using ux by (auto dest: tree-pathD)
  have u-path: u ∈ set (shortest-path-to x)
    and v-path: v ∈ set (shortest-path-to x)
    using ux vx by (auto dest: tree-pathD)
  obtain i where i: i < length (shortest-path-to x) shortest-path-to x ! i = u
    using u-path by (auto simp: in-set-conv-nth)
  obtain j where j: j < length (shortest-path-to x) shortest-path-to x ! j = v
    using v-path by (auto simp: in-set-conv-nth)
  show ?thesis
  proof (cases i ≤ j)
    case True
    have tree-path u v
      using tree-path-between-shortest-path-indices[OF reach-x True j(1)] i(2) j(2)
      by simp
    then show ?thesis
      by blast
  next
    case False
    then have j ≤ i
```

```
    by simp
  have tree-path v u
    using tree-path-between-shortest-path-indices[OF reach-x ‹j ≤ i› i(1)] i(2)
j(2)
    by simp
  then show ?thesis
    by blast
 qed
qed

lemma tree-antichainD:
 assumes tree-antichain S
  and u ∈ S v ∈ S tree-path u v
 shows u = v
 using assms unfolding tree-antichain-def by blast

lemma tree-of-disjoint-if-antichain:
 assumes anti: tree-antichain S
  and uS: u ∈ S
  and vS: v ∈ S
  and neq: u ≠ v
 shows tree-of u ∩ tree-of v = {}
proof
 show tree-of u ∩ tree-of v ⊆ {}
 proof
  fix x
  assume x: x ∈ tree-of u ∩ tree-of v
  then have ux: tree-path u x and vx: tree-path v x
    unfolding tree-of-def by auto
  have tree-path u v ∨ tree-path v u
    using tree-path-comparable[OF ux vx] .
  then have u = v
  proof
    assume uv′: tree-path u v
    then show ?thesis
      using tree-antichainD[OF anti uS vS uv′] by simp
  next
    assume vu′: tree-path v u
    then show ?thesis
      using tree-antichainD[OF anti vS uS vu′] by simp
  qed
  with neq show x ∈ {}
    by blast
 qed
qed simp

lemma through-iff-tree-path:
 assumes reachable s v
 shows through S v ⟷ (∃ u∈S. tree-path u v)
```

proof
assume *through S v*
then obtain *u p* **where** *u*: *u* ∈ *S shortest-walk s p v u* ∈ *set p*
unfolding *through-def* **by** *blast*
have *p* = *shortest-path-to v*
using *shortest-path-to-shortest*[*OF assms*] *u*(*2*) *unique-shortest-walk* **by** *blast*
then have *tree-path u v*
using *assms u* **by** (*auto intro*: *tree-pathI*)
then show ∃ *u*∈*S*. *tree-path u v*
using *u*(*1*) **by** *blast*
next
assume ∃ *u*∈*S*. *tree-path u v*
then obtain *u* **where** *u*: *u* ∈ *S tree-path u v*
by *blast*
then have *u* ∈ *set* (*shortest-path-to v*)
by (*auto dest*: *tree-pathD*)
moreover have *shortest-walk s* (*shortest-path-to v*) *v*
using *shortest-path-to-shortest*[*OF assms*] **.**
ultimately show *through S v*
using *u*(*1*) **unfolding** *through-def* **by** *blast*
qed

lemma *bound-tree-eq-tree-set*:
bound-tree S B = {*v* ∈ *tree-set S*. *below-bound* (*dist s v*) *B*}
proof
show *bound-tree S B* ⊆ {*v* ∈ *tree-set S*. *below-bound* (*dist s v*) *B*}
proof
fix *v*
assume *v*: *v* ∈ *bound-tree S B*
then have *reach*: *reachable s v* **and** *through S v* **and** *below*: *below-bound* (*dist s v*) *B* **and** *v* ∈ *V*
unfolding *bound-tree-def* **by** *auto*
then obtain *u* **where** *u* ∈ *S tree-path u v*
using *through-iff-tree-path*[*OF reach*, *of S*] **by** *blast*
then have *v* ∈ *tree-set S*
using ‹*v* ∈ *V*› **unfolding** *tree-set-def* **by** *blast*
then show *v* ∈ {*v* ∈ *tree-set S*. *below-bound* (*dist s v*) *B*}
using *below* **by** *simp*
qed
next
show {*v* ∈ *tree-set S*. *below-bound* (*dist s v*) *B*} ⊆ *bound-tree S B*
proof
fix *v*
assume *v*: *v* ∈ {*v* ∈ *tree-set S*. *below-bound* (*dist s v*) *B*}
then obtain *u* **where** *vV*: *v* ∈ *V* **and** *uS*: *u* ∈ *S* **and** *tp*: *tree-path u v*
and *below*: *below-bound* (*dist s v*) *B*
unfolding *tree-set-def* **by** *blast*
then have *reach*: *reachable s v*
by (*auto dest*: *tree-pathD*)

```
    have through S v
      using through-iff-tree-path[OF reach, of S] uS tp by blast
    then show v ∈ bound-tree S B
      using vV reach below unfolding bound-tree-def by blast
  qed
qed

end

end
theory BMSSP-Find-Pivots-Core
  imports BMSSP-Unique-Shortest-Tree
begin
```

19 FindPivots Core Relaxation Lemmas

The concrete FindPivots procedure performs bounded Bellman-Ford-style relaxations. This theory packages the generic relaxation facts in the form needed for that proof: vertices reached by a bounded tight path from a complete source are complete after the corresponding path relaxations, and remain complete through additional valid relaxations.

```
context unique-shortest-digraph
begin

lemma finite-E [simp]: finite E
proof −
  have E ⊆ V × V
    using edge-in-V by auto
  moreover have finite (V × V)
    using finite-V by simp
  ultimately show ?thesis
    by (rule finite-subset)
qed

definition outgoing-edges where
  outgoing-edges F = {(u, v) ∈ E. u ∈ F}

lemma finite-outgoing-edges [simp]:
  finite (outgoing-edges F)
proof −
  have outgoing-edges F ⊆ E
    unfolding outgoing-edges-def by blast
  then show ?thesis
    by (rule finite-subset) simp
qed

definition edge-list-of where
  edge-list-of A = (SOME xs. set xs = A ∧ distinct xs)
```

lemma *edge-list-of-properties*:
  assumes *finite A*
  shows *set* (*edge-list-of A*) = *A*
    and *distinct* (*edge-list-of A*)
proof −
  have ∃ *xs*. *set xs* = *A* ∧ *distinct xs*
    using *finite-distinct-list*[*OF assms*] **.**
  then have *props*: *set* (*edge-list-of A*) = *A* ∧ *distinct* (*edge-list-of A*)
    unfolding *edge-list-of-def* **by** (*rule someI-ex*)
  then show *set* (*edge-list-of A*) = *A*
    by *blast*
  from *props* **show** *distinct* (*edge-list-of A*)
    by *blast*
qed

definition *relax-frontier* **where**
  relax-frontier d F = *relax-edges d* (*edge-list-of* (*outgoing-edges F*))

lemma *relax-frontier-le*:
  relax-frontier d F x ≤ *d x*
  unfolding *relax-frontier-def* **by** (*rule relax-edges-le*)

lemma *relax-frontier-sound*:
  assumes *sound*: *sound-label d*
    and *frontier-reaches*: ⋀*u*. *u* ∈ *F* ⟹ *reachable s u*
  shows *sound-label* (*relax-frontier d F*)
proof −
  let *?es* = *edge-list-of* (*outgoing-edges F*)
  have *set-es*: *set ?es* = *outgoing-edges F*
    using *edge-list-of-properties*(*1*)[*OF finite-outgoing-edges*] **.**
  have *edge*: ⋀*u v*. (*u*, *v*) ∈ *set ?es* ⟹ (*u*, *v*) ∈ *E*
    using *set-es* **unfolding** *outgoing-edges-def* **by** *blast*
  have *reach*: ⋀*u v*. (*u*, *v*) ∈ *set ?es* ⟹ *reachable s u*
    using *set-es frontier-reaches* **unfolding** *outgoing-edges-def* **by** *blast*
  show *?thesis*
    unfolding *relax-frontier-def* **using** *relax-edges-sound*[*OF sound edge reach*] **.**
qed

fun *relax-rounds* **where**
  relax-rounds d [] = *d*
| *relax-rounds d* (*r* # *rs*) = *relax-rounds* (*relax-edges d r*) *rs*

definition *within-k-edges* **where**
  within-k-edges p k ⟷ *p* ≠ [] ∧ *length* (*path-edges p*) ≤ *k*

lemma *relax-rounds-concat*:
  relax-rounds d rounds = *relax-edges d* (*concat rounds*)
  by (*induction rounds arbitrary*: *d*) (*simp-all add*: *relax-edges-append*)

lemma *relax-rounds-sound*:
assumes *sound*: *sound-label d*
and *edges*: $\bigwedge u\ v.\ (u, v) \in set\ (concat\ rounds) \Longrightarrow (u, v) \in E$
and *reaches*: $\bigwedge u\ v.\ (u, v) \in set\ (concat\ rounds) \Longrightarrow reachable\ s\ u$
shows *sound-label* (*relax-rounds d rounds*)
proof −
have *sound-label* (*relax-edges d* (*concat rounds*))
by (*rule relax-edges-sound*[*OF sound edges reaches*])
then show *?thesis*
by (*simp add*: *relax-rounds-concat*)
qed

lemma *relax-rounds-preserves-complete-sound*:
assumes *sound*: *sound-label d*
and *complete-x*: *d x* = *dist s x*
and *edges*: $\bigwedge u\ v.\ (u, v) \in set\ (concat\ rounds) \Longrightarrow (u, v) \in E$
and *reaches*: $\bigwedge u\ v.\ (u, v) \in set\ (concat\ rounds) \Longrightarrow reachable\ s\ u$
shows *relax-rounds d rounds x* = *dist s x*
proof −
have *relax-edges d* (*concat rounds*) *x* = *dist s x*
by (*rule relax-edges-preserves-complete-sound*[*OF sound complete-x edges reaches*])
then show *?thesis*
by (*simp add*: *relax-rounds-concat*)
qed

lemma *path-edges-length*:
length (*path-edges p*) = *length p* − *1*
by (*induction p rule*: *path-edges.induct*) *auto*

lemma *within-k-edgesD*:
assumes *within-k-edges p k*
shows *p* ≠ [] *length* (*path-edges p*) ≤ *k*
using *assms* **unfolding** *within-k-edges-def* **by** *auto*

lemma *find-pivots-tight-path-prefix-completion*:
assumes *bounded*: *within-k-edges p k*
and *sound*: *sound-label d*
and *complete-source*: *d* (*hd p*) = *dist s* (*hd p*)
and *tight*: *successively tight-edge-step p*
and *es*: *es* = *path-edges p* @ *extra*
and *extra-edges*: $\bigwedge u\ v.\ (u, v) \in set\ extra \Longrightarrow (u, v) \in E$
and *extra-reaches*: $\bigwedge u\ v.\ (u, v) \in set\ extra \Longrightarrow reachable\ s\ u$
shows *sound-label* (*relax-edges d es*)
and *relax-edges d es* (*last p*) = *dist s* (*last p*)
proof −
have *nonempty*: *p* ≠ []
using *within-k-edgesD*(*1*)[*OF bounded*] **.**

show *sound-label* (*relax-edges d es*)
using *relax-edges-with-tight-path-prefix-complete*(*1*)
[*OF nonempty sound complete-source tight es extra-edges extra-reaches*] **.**
show *relax-edges d es* (*last p*) = *dist s* (*last p*)
using *relax-edges-with-tight-path-prefix-complete*(*2*)
[*OF nonempty sound complete-source tight es extra-edges extra-reaches*] **.**
qed

lemma *find-pivots-tight-path-block-completion*:
assumes *bounded*: *within-k-edges p k*
and *sound*: *sound-label d*
and *complete-source*: *d* (*hd p*) = *dist s* (*hd p*)
and *tight*: *successively tight-edge-step p*
and *es*: *es* = *pre* @ *path-edges p* @ *extra*
and *pre-edges*: ⋀*u v*. (*u*, *v*) ∈ *set pre* ⟹ (*u*, *v*) ∈ *E*
and *pre-reaches*: ⋀*u v*. (*u*, *v*) ∈ *set pre* ⟹ *reachable s u*
and *extra-edges*: ⋀*u v*. (*u*, *v*) ∈ *set extra* ⟹ (*u*, *v*) ∈ *E*
and *extra-reaches*: ⋀*u v*. (*u*, *v*) ∈ *set extra* ⟹ *reachable s u*
shows *sound-label* (*relax-edges d es*)
and *relax-edges d es* (*last p*) = *dist s* (*last p*)
proof −
let *?d0* = *relax-edges d pre*
have *nonempty*: *p* ≠ []
using *within-k-edgesD*(*1*)[*OF bounded*] **.**
have *sound0*: *sound-label ?d0*
using *relax-edges-sound*[*OF sound pre-edges pre-reaches*] **.**
have *complete0*: *?d0* (*hd p*) = *dist s* (*hd p*)
using *relax-edges-preserves-complete-sound*[*OF sound complete-source pre-edges pre-reaches*] **.**
have *tail-sound*: *sound-label* (*relax-edges ?d0* (*path-edges p* @ *extra*))
using *relax-edges-with-tight-path-prefix-complete*(*1*)
[*OF nonempty sound0 complete0 tight refl extra-edges extra-reaches*] **.**
have *tail-complete*: *relax-edges ?d0* (*path-edges p* @ *extra*) (*last p*) = *dist s* (*last p*)
using *relax-edges-with-tight-path-prefix-complete*(*2*)
[*OF nonempty sound0 complete0 tight refl extra-edges extra-reaches*] **.**
have *unfold-es*: *relax-edges d es* = *relax-edges ?d0* (*path-edges p* @ *extra*)
using *es relax-edges-append*[*of d pre path-edges p* @ *extra*] **by** *simp*
show *sound-label* (*relax-edges d es*)
using *tail-sound unfold-es* **by** *simp*
show *relax-edges d es* (*last p*) = *dist s* (*last p*)
using *tail-complete unfold-es* **by** *simp*
qed

lemma *find-pivots-rounds-block-completion*:
assumes *bounded*: *within-k-edges p k*
and *sound*: *sound-label d*
and *complete-source*: *d* (*hd p*) = *dist s* (*hd p*)
and *tight*: *successively tight-edge-step p*

and *rounds*: *concat rounds* = *pre* @ *path-edges p* @ *extra*
and *pre-edges*: $\bigwedge u\ v.\ (u,\ v) \in set\ pre \Longrightarrow (u,\ v) \in E$
and *pre-reaches*: $\bigwedge u\ v.\ (u,\ v) \in set\ pre \Longrightarrow reachable\ s\ u$
and *extra-edges*: $\bigwedge u\ v.\ (u,\ v) \in set\ extra \Longrightarrow (u,\ v) \in E$
and *extra-reaches*: $\bigwedge u\ v.\ (u,\ v) \in set\ extra \Longrightarrow reachable\ s\ u$
shows *sound-label* (*relax-rounds d rounds*)
and *relax-rounds d rounds* (*last p*) = *dist s* (*last p*)
proof −
have *edge-sound*: *sound-label* (*relax-edges d* (*concat rounds*))
using *find-pivots-tight-path-block-completion*(*1*)
[*OF bounded sound complete-source tight rounds pre-edges pre-reaches extra-edges extra-reaches*] .
have *endpoint*: *relax-edges d* (*concat rounds*) (*last p*) = *dist s* (*last p*)
using *find-pivots-tight-path-block-completion*(*2*)
[*OF bounded sound complete-source tight rounds pre-edges pre-reaches extra-edges extra-reaches*] .
show *sound-label* (*relax-rounds d rounds*)
using *edge-sound* **by** (*simp add*: *relax-rounds-concat*)
show *relax-rounds d rounds* (*last p*) = *dist s* (*last p*)
using *endpoint* **by** (*simp add*: *relax-rounds-concat*)
qed

lemma *relax-frontier-tight-successor-complete*:
assumes *sound*: *sound-label d*
and *uF*: $u \in F$
and *frontier-reaches*: $\bigwedge x.\ x \in F \Longrightarrow reachable\ s\ x$
and *complete-u*: *d u* = *dist s u*
and *tight*: *tight-edge-step u v*
shows *sound-label* (*relax-frontier d F*)
and *relax-frontier d F v* = *dist s v*
proof −
let *?es* = *edge-list-of* (*outgoing-edges F*)
have *set-es*: *set ?es* = *outgoing-edges F*
using *edge-list-of-properties*(*1*)[*OF finite-outgoing-edges*] .
have *edge-uv*: $(u,\ v) \in E$
using *tight* **unfolding** *tight-edge-step-def* **by** *blast*
have *mem-uv*: $(u,\ v) \in set\ ?es$
using *set-es edge-uv uF* **unfolding** *outgoing-edges-def* **by** *blast*
obtain *pre extra* **where** *es-split*: *?es* = *pre* @ $(u,\ v)$ # *extra*
using *split-list*[*OF mem-uv*] **by** *blast*
have *bounded*: *within-k-edges* $[u,\ v]$ (*Suc 0*)
unfolding *within-k-edges-def* **by** *simp*
have *tight-path*: *successively tight-edge-step* $[u,\ v]$
using *tight* **by** *simp*
have *es-path*: *?es* = *pre* @ *path-edges* $[u,\ v]$ @ *extra*
using *es-split* **by** *simp*
have *pre-edges*: $\bigwedge a\ b.\ (a,\ b) \in set\ pre \Longrightarrow (a,\ b) \in E$
using *set-es es-split* **unfolding** *outgoing-edges-def* **by** *auto*
have *pre-reaches*: $\bigwedge a\ b.\ (a,\ b) \in set\ pre \Longrightarrow reachable\ s\ a$

using *set-es es-split frontier-reaches* **unfolding** *outgoing-edges-def* **by** *auto*
have *extra-edges*: $\bigwedge a\ b.\ (a,\ b) \in set\ extra \Longrightarrow (a,\ b) \in E$
using *set-es es-split* **unfolding** *outgoing-edges-def* **by** *auto*
have *extra-reaches*: $\bigwedge a\ b.\ (a,\ b) \in set\ extra \Longrightarrow reachable\ s\ a$
using *set-es es-split frontier-reaches* **unfolding** *outgoing-edges-def* **by** *auto*
have *core1-path*: *sound-label* (*relax-edges d* (*pre* @ *path-edges* [*u*, *v*] @ *extra*))
proof (*rule find-pivots-tight-path-block-completion*(*1*)
[**where** $p = [u,\ v]$ **and** $k = Suc\ 0$ **and** *pre* = *pre* **and** *extra* = *extra*])
show *within-k-edges* [*u*, *v*] (*Suc 0*)
using *bounded* **.**
show *sound-label d*
using *sound* **.**
show *d* (*hd* [*u*, *v*]) = *dist s* (*hd* [*u*, *v*])
using *complete-u* **by** *simp*
show *successively tight-edge-step* [*u*, *v*]
using *tight-path* **.**
show *pre* @ *path-edges* [*u*, *v*] @ *extra* = *pre* @ *path-edges* [*u*, *v*] @ *extra*
by *simp*
show $\bigwedge a\ b.\ (a,\ b) \in set\ pre \Longrightarrow (a,\ b) \in E$
using *pre-edges* **.**
show $\bigwedge a\ b.\ (a,\ b) \in set\ pre \Longrightarrow reachable\ s\ a$
using *pre-reaches* **.**
show $\bigwedge a\ b.\ (a,\ b) \in set\ extra \Longrightarrow (a,\ b) \in E$
using *extra-edges* **.**
show $\bigwedge a\ b.\ (a,\ b) \in set\ extra \Longrightarrow reachable\ s\ a$
using *extra-reaches* **.**
qed
have *core2-path*: *relax-edges d* (*pre* @ *path-edges* [*u*, *v*] @ *extra*) *v* = *dist s v*
proof −
have *relax-edges d* (*pre* @ *path-edges* [*u*, *v*] @ *extra*) (*last* [*u*, *v*]) =
dist s (*last* [*u*, *v*])
proof (*rule find-pivots-tight-path-block-completion*(*2*)
[**where** $p = [u,\ v]$ **and** $k = Suc\ 0$ **and** *pre* = *pre* **and** *extra* = *extra*])
show *within-k-edges* [*u*, *v*] (*Suc 0*)
using *bounded* **.**
show *sound-label d*
using *sound* **.**
show *d* (*hd* [*u*, *v*]) = *dist s* (*hd* [*u*, *v*])
using *complete-u* **by** *simp*
show *successively tight-edge-step* [*u*, *v*]
using *tight-path* **.**
show *pre* @ *path-edges* [*u*, *v*] @ *extra* = *pre* @ *path-edges* [*u*, *v*] @ *extra*
by *simp*
show $\bigwedge a\ b.\ (a,\ b) \in set\ pre \Longrightarrow (a,\ b) \in E$
using *pre-edges* **.**
show $\bigwedge a\ b.\ (a,\ b) \in set\ pre \Longrightarrow reachable\ s\ a$
using *pre-reaches* **.**
show $\bigwedge a\ b.\ (a,\ b) \in set\ extra \Longrightarrow (a,\ b) \in E$
using *extra-edges* **.**

```
    show ⋀a b. (a, b) ∈ set extra ⟹ reachable s a
      using extra-reaches .
  qed
  then show ?thesis
    by simp
 qed
 have core1: sound-label (relax-edges d ?es)
  using core1-path es-path by simp
 have core2: relax-edges d ?es v = dist s v
  using core2-path es-path by simp
 show sound-label (relax-frontier d F)
  using core1 unfolding relax-frontier-def .
 show relax-frontier d F v = dist s v
  using core2 unfolding relax-frontier-def .
qed

end

end
theory BMSSP-Find-Pivots
 imports BMSSP-Find-Pivots-Core BMSSP-Algorithm-Correctness
begin
```

20 Concrete FindPivots Rounds

FindPivots is the first nontrivial subroutine in a BMSSP recursive step. Conceptually, it starts from a current source frontier, performs a bounded number of relaxation rounds, records the vertices reached by those rounds, and chooses a smaller set of pivots whose shortest-path trees are large enough to justify the next recursive partitioning phase.

This theory gives the concrete round-by-round model used by the later correctness and cost proofs. The round state consists of a label function, a frontier, and a seen set. One round relaxes edges out of the frontier, selects the next frontier under the current upper bound, and extends the seen set. The capped variant stops early once the seen set exceeds a cardinality cap. That cap is the formal hook for the paper's parameter schedule: if the local search grows too quickly, the original source set is already large enough to serve as the next pivot set.

The proof obligations in this file fall into three groups. First, the frontier and seen sets stay inside the finite vertex set and have the expected cardinality bounds under an outdegree assumption. Second, relaxation preserves the BMSSP source precondition and label soundness. Third, if the capped search does not overflow, every short tight path starting from a complete source is completed, and long tight trees expose a pivot. These facts are assembled into the concrete FindPivots postcondition consumed by the abstract algorithm-correctness theory.

context *unique-shortest-digraph*
begin

definition *fp-next* **where**
  fp-next d F B =
    $\{v \in V.\ \exists u{\in}F.\ (u, v) \in E \wedge d\ u + w\ u\ v \leq d\ v \wedge$
      below-bound $(d\ u + w\ u\ v)\ B\}$

fun *fp-iter* **where**
  fp-iter 0 d F W B $= (d, F, W)$
| *fp-iter* $(Suc\ n)$ *d F W B =*
    (*let* $d' =$ *relax-frontier d F*;
        $F' =$ *fp-next d F B*;
        $W' = W \cup F'$
     in fp-iter n $d'\ F'\ W'\ B$)

fun *fp-iter-capped* **where**
  fp-iter-capped 0 cap d F W B $= (d, F, W)$
| *fp-iter-capped* $(Suc\ n)$ *cap d F W B =*
    (*let* $d' =$ *relax-frontier d F*;
        $F' =$ *fp-next d F B*;
        $W' = W \cup F'$
     in if card $W' >$ *cap then* (d', F', W')
       else fp-iter-capped n cap $d'\ F'\ W'\ B$)

fun *fp-iter-capped-scan-cost* **where**
  fp-iter-capped-scan-cost 0 cap d F W B $= 0$
| *fp-iter-capped-scan-cost* $(Suc\ n)$ *cap d F W B =*
    (*let* $d' =$ *relax-frontier d F*;
        $F' =$ *fp-next d F B*;
        $W' = W \cup F'$;
        round $=$ *card* (*outgoing-edges F*)
     in round $+$ (*if card* $W' >$ *cap then 0*
       else fp-iter-capped-scan-cost n cap $d'\ F'\ W'\ B$))

definition *fp-label* **where**
  fp-label st $=$ *fst st*

definition *fp-frontier* **where**
  fp-frontier st $=$ *fst* (*snd st*)

definition *fp-seen* **where**
  fp-seen st $=$ *snd* (*snd st*)

definition *find-pivots-label* **where**
  find-pivots-label k d S B $=$ *fp-label* (*fp-iter k d S S B*)

definition *find-pivots-seen* **where**
  find-pivots-seen k d S B $=$ *fp-seen* (*fp-iter k d S S B*)

definition *find-pivots-seen-capped* **where**
find-pivots-seen-capped k cap d S B = fp-seen (fp-iter-capped k cap d S S B)

definition *find-pivots-label-capped* **where**
find-pivots-label-capped k cap d S B = fp-label (fp-iter-capped k cap d S S B)

definition *find-pivots-pivots-capped* **where**
find-pivots-pivots-capped k cap d S B =
(if card (find-pivots-seen-capped k cap d S B) > cap then S
else {u ∈ S. k ≤ card (tree-of u ∩ find-pivots-seen-capped k cap d S B)})

definition *find-pivots-pivots* **where**
find-pivots-pivots k d S B =
{u ∈ S. k ≤ card (tree-of u ∩ find-pivots-seen k d S B)}

definition *short-tight-witness* **where**
short-tight-witness k d S B x ⟷
(∃ p. p ≠ [] ∧ hd p ∈ S ∧ last p = x ∧
d (hd p) = dist s (hd p) ∧ successively tight-edge-step p ∧
(∀ y∈set p. below-bound (dist s y) B) ∧
length (path-edges p) ≤ k)

definition *find-pivots-short-vertices* **where**
find-pivots-short-vertices k d S B =
{x ∈ V. short-tight-witness k d S B x}

definition *out-neighbors* **where**
out-neighbors F = {v ∈ V. ∃ u∈F. (u, v) ∈ E}

definition *outdegree-le* **where**
outdegree-le Δ ⟷ (∀ u∈V. card (out-neighbors {u}) ≤ Δ)

definition *edge-outdegree-le* **where**
edge-outdegree-le Δ ⟷ (∀ u∈V. card (outgoing-edges {u}) ≤ Δ)

The definitions above mirror the paper's informal description. The set *fp-next* is the next bounded frontier: it contains vertices reached by a currently tight relaxation whose value remains below the call bound. The iterator *fp-iter* performs exactly k such rounds, while *fp-iter-capped* returns early when the accumulated seen set exceeds the cap. The projections *find-pivots-label-capped*, *find-pivots-seen-capped*, and *find-pivots-pivots-capped* are the concrete objects used by later BMSSP steps.

Two degree predicates are kept separate. *outdegree-le* bounds the number of distinct out-neighbours and is convenient for bounding the size of the seen set. *edge-outdegree-le* bounds outgoing edge records and is used for scan-cost accounting. The distinction matters in multigraph-style encodings where several edges may point to the same neighbour.

lemma *finite-tree-of* [*simp*]:
finite (*tree-of u*)
unfolding *tree-of-def* **using** *finite-V* **by** *auto*

lemma *fp-next-reaches*:
assumes *reaches*: $\bigwedge u.\ u \in F \Longrightarrow$ *reachable s u*
and *v*: *v* ∈ *fp-next d F B*
shows *reachable s v*
proof −
obtain *u* **where** $u \in F$ $(u, v) \in E$
using *v* **unfolding** *fp-next-def* **by** *blast*
then show *?thesis*
using *reaches dist-triangle-edge* **by** *blast*
qed

lemma *fp-next-subset-V*:
fp-next d F B ⊆ *V*
unfolding *fp-next-def* **by** *blast*

lemma *fp-next-subset-out-neighbors*:
fp-next d F B ⊆ *out-neighbors F*
unfolding *fp-next-def out-neighbors-def* **by** *blast*

lemma *out-neighbors-subset-V*:
out-neighbors F ⊆ *V*
unfolding *out-neighbors-def* **by** *blast*

lemma *finite-out-neighbors* [*simp*]:
finite (*out-neighbors F*)
using *finite-subset*[*OF out-neighbors-subset-V finite-V*] **.**

lemma *out-neighbors-UN-singleton*:
out-neighbors F = $(\bigcup u{\in}F.$ *out-neighbors* $\{u\})$
unfolding *out-neighbors-def* **by** *auto*

lemma *outgoing-edges-UN-singleton*:
outgoing-edges F = $(\bigcup u{\in}F.$ *outgoing-edges* $\{u\})$
unfolding *outgoing-edges-def* **by** *auto*

lemma *card-out-neighbors-le*:
assumes *F-subset*: *F* ⊆ *V*
and *degree*: *outdegree-le* Δ
shows *card* (*out-neighbors F*) ≤ Δ ∗ *card F*
proof −
have *finite-F*: *finite F*
using *finite-subset*[*OF F-subset finite-V*] **.**
have *card-union*: *card* (*out-neighbors F*) ≤ $(\sum u{\in}F.$ *card* (*out-neighbors* $\{u\}))$
proof −
have *card* (*out-neighbors F*) = *card* $(\bigcup u{\in}F.$ *out-neighbors* $\{u\})$

```
    using out-neighbors-UN-singleton[of F] by simp
  also have ... ≤ (∑ u∈F. card (out-neighbors {u}))
    using card-UN-le[OF finite-F, of λu. out-neighbors {u}] .
  finally show ?thesis .
 qed
 have sum-bound: (∑ u∈F. card (out-neighbors {u})) ≤ (∑ u∈F. Δ)
 proof (rule sum-mono)
  fix u
  assume u ∈ F
  then have u ∈ V
    using F-subset by blast
  then show card (out-neighbors {u}) ≤ Δ
    using degree unfolding outdegree-le-def by blast
 qed
 also have ... = Δ * card F
  by (simp add: mult.commute)
 finally have (∑ u∈F. card (out-neighbors {u})) ≤ Δ * card F .
 then show ?thesis
  using card-union by linarith
qed

lemma card-fp-next-le:
 assumes F-subset: F ⊆ V
  and degree: outdegree-le Δ
 shows card (fp-next d F B) ≤ Δ * card F
proof −
 have card (fp-next d F B) ≤ card (out-neighbors F)
  by (rule card-mono) (simp-all add: fp-next-subset-out-neighbors)
 also have ... ≤ Δ * card F
  using card-out-neighbors-le[OF F-subset degree] .
 finally show ?thesis .
qed

lemma card-outgoing-edges-le:
 assumes F-subset: F ⊆ V
  and degree: edge-outdegree-le Δ
 shows card (outgoing-edges F) ≤ Δ * card F
proof −
 have finite-F: finite F
  using finite-subset[OF F-subset finite-V] .
 have card-union: card (outgoing-edges F) ≤
    (∑ u∈F. card (outgoing-edges {u}))
 proof −
  have card (outgoing-edges F) = card (⋃ u∈F. outgoing-edges {u})
    using outgoing-edges-UN-singleton[of F] by simp
  also have ... ≤ (∑ u∈F. card (outgoing-edges {u}))
    using card-UN-le[OF finite-F, of λu. outgoing-edges {u}] .
  finally show ?thesis .
 qed
```

```
  have sum-bound: (∑ u∈F. card (outgoing-edges {u})) ≤ (∑ u∈F. Δ)
  proof (rule sum-mono)
    fix u
    assume u ∈ F
    then have u ∈ V
      using F-subset by blast
    then show card (outgoing-edges {u}) ≤ Δ
      using degree unfolding edge-outdegree-le-def by blast
  qed
  also have ... = Δ * card F
    by (simp add: mult.commute)
  finally have (∑ u∈F. card (outgoing-edges {u})) ≤ Δ * card F .
  then show ?thesis
    using card-union by linarith
qed

lemma card-seen-after-fp-step-le:
  assumes W-subset: W ⊆ V
    and F-subset: F ⊆ V
    and degree: outdegree-le Δ
  shows card (W ∪ fp-next d F B) ≤ card W + Δ * card F
proof −
  have card (W ∪ fp-next d F B) ≤ card W + card (fp-next d F B)
    by (rule card-Un-le)
  also have ... ≤ card W + Δ * card F
    using card-fp-next-le[OF F-subset degree] by simp
  finally show ?thesis .
qed

lemma card-seen-after-fp-step-cap-le:
  assumes F-subset-W: F ⊆ W
    and W-subset: W ⊆ V
    and degree: outdegree-le Δ
    and W-cap: card W ≤ cap
  shows card (W ∪ fp-next d F B) ≤ Suc Δ * cap
proof −
  have F-subset: F ⊆ V
    using F-subset-W W-subset by blast
  have finite-W: finite W
    using finite-subset[OF W-subset finite-V] .
  have card-F-le-W: card F ≤ card W
    using card-mono[OF finite-W F-subset-W] .
  have card-F-le-cap: card F ≤ cap
    using card-F-le-W W-cap by linarith
  have card (W ∪ fp-next d F B) ≤ card W + Δ * card F
    using card-seen-after-fp-step-le[OF W-subset F-subset degree] .
  also have ... ≤ cap + Δ * cap
    using W-cap card-F-le-cap by (intro add-mono mult-left-mono) simp-all
  also have ... = Suc Δ * cap
```

```
    by simp
  finally show ?thesis .
qed

lemma fp-iter-capped-seen-subset-V:
  assumes F-subset: F ⊆ V
    and W-subset: W ⊆ V
  shows fp-seen (fp-iter-capped n cap d F W B) ⊆ V
  using assms
proof (induction n arbitrary: d F W)
  case 0
  then show ?case
    unfolding fp-seen-def by simp
next
  case (Suc n)
  let ?F' = fp-next d F B
  let ?W' = W ∪ ?F'
  have F'-subset: ?F' ⊆ V
    using fp-next-subset-V .
  have W'-subset: ?W' ⊆ V
    using Suc.prems F'-subset by blast
  show ?case
  proof (cases card ?W' > cap)
    case True
    then show ?thesis
      using W'-subset by (simp add: fp-seen-def Let-def)
  next
    case False
    have fp-seen (fp-iter-capped n cap (relax-frontier d F) ?F' ?W' B) ⊆ V
      using Suc.IH[OF F'-subset W'-subset] .
    then show ?thesis
      using False by (simp add: fp-seen-def Let-def)
  qed
qed

lemma fp-iter-capped-frontier-subset-seen:
  assumes F ⊆ W
  shows fp-frontier (fp-iter-capped n cap d F W B) ⊆
    fp-seen (fp-iter-capped n cap d F W B)
  using assms
proof (induction n arbitrary: d F W)
  case 0
  then show ?case
    unfolding fp-frontier-def fp-seen-def by simp
next
  case (Suc n)
  let ?F' = fp-next d F B
  let ?W' = W ∪ ?F'
  have F'-W': ?F' ⊆ ?W'
```

```
    by blast
  show ?case
  proof (cases card ?W' > cap)
    case True
    then show ?thesis
      using F'-W' by (simp add: fp-frontier-def fp-seen-def Let-def)
  next
    case False
    show ?thesis
      using Suc.IH[OF F'-W'] False by (simp add: fp-frontier-def fp-seen-def
Let-def)
  qed
qed

lemma fp-iter-capped-seen-initial-subset:
  W ⊆ fp-seen (fp-iter-capped n cap d F W B)
proof (induction n arbitrary: d F W)
  case 0
  then show ?case
    unfolding fp-seen-def by simp
next
  case (Suc n)
  let ?d' = relax-frontier d F
  let ?F' = fp-next d F B
  let ?W' = W ∪ ?F'
  show ?case
  proof (cases card ?W' > cap)
    case True
    then show ?thesis
      unfolding fp-seen-def by (simp add: Let-def)
  next
    case False
    have W ⊆ ?W'
      by blast
    also have ... ⊆ fp-seen (fp-iter-capped n cap ?d' ?F' ?W' B)
      by (rule Suc.IH)
    finally show ?thesis
      using False unfolding fp-seen-def by (simp add: Let-def)
  qed
qed

lemma fp-iter-capped-seen-reaches:
  assumes F-reaches: ⋀u. u ∈ F ⟹ reachable s u
    and W-reaches: ⋀u. u ∈ W ⟹ reachable s u
  shows ∀ u∈fp-seen (fp-iter-capped n cap d F W B). reachable s u
  using assms
proof (induction n arbitrary: d F W)
  case 0
  then show ?case
```

```
    unfolding fp-seen-def by simp
next
  case (Suc n)
  let ?F' = fp-next d F B
  let ?W' = W ∪ ?F'
  have F'-reaches: ⋀u. u ∈ ?F' ⟹ reachable s u
  proof −
    fix u
    assume uF': u ∈ ?F'
    then obtain x where xF: x ∈ F and edge: (x, u) ∈ E
      unfolding fp-next-def by blast
    have reachable s x
      by (rule Suc.prems(1)[OF xF])
    then show reachable s u
      using dist-triangle-edge[OF edge] by blast
  qed
  have W'-reaches: ⋀u. u ∈ ?W' ⟹ reachable s u
    using Suc.prems(2) F'-reaches by blast
  show ?case
  proof (cases card ?W' > cap)
    case True
    then show ?thesis
      using W'-reaches by (simp add: fp-seen-def Let-def)
  next
    case False
    have tail:
      ∀u∈fp-seen (fp-iter-capped n cap (relax-frontier d F) ?F' ?W' B).
        reachable s u
      by (rule Suc.IH
        [where d = relax-frontier d F and F = ?F' and W = ?W',
          OF F'-reaches W'-reaches])
    show ?thesis
      using tail False by (simp add: fp-seen-def Let-def)
  qed
qed

lemma find-pivots-seen-capped-reaches:
  assumes S-reaches: ⋀u. u ∈ S ⟹ reachable s u
    and u: u ∈ find-pivots-seen-capped k cap d S B
  shows reachable s u
  using fp-iter-capped-seen-reaches
    [where F = S and W = S and n = k and cap = cap and d = d and B = B,
      OF S-reaches S-reaches]
    u
  unfolding find-pivots-seen-capped-def by blast

theorem fp-iter-capped-seen-card-le:
  assumes F-subset-W: F ⊆ W
    and W-subset: W ⊆ V
```

```
    and degree: outdegree-le Δ
    and W-cap: card W ≤ cap
  shows card (fp-seen (fp-iter-capped n cap d F W B)) ≤ Suc Δ * cap
  using assms
proof (induction n arbitrary: d F W)
  case 0
  have card W ≤ Suc Δ * cap
    using 0.prems(4) by simp
  then show ?case
    unfolding fp-seen-def by simp
next
  case (Suc n)
  let ?d' = relax-frontier d F
  let ?F' = fp-next d F B
  let ?W' = W ∪ ?F'
  have F-subset: F ⊆ V
    using Suc.prems(1,2) by blast
  have F'-subset: ?F' ⊆ V
    using fp-next-subset-V .
  have W'-subset: ?W' ⊆ V
    using Suc.prems(2) F'-subset by blast
  have F'-W': ?F' ⊆ ?W'
    by blast
  have step-cap: card ?W' ≤ Suc Δ * cap
    using card-seen-after-fp-step-cap-le
      [OF Suc.prems(1,2,3,4), of d B] .
  show ?case
  proof (cases card ?W' > cap)
    case True
    then show ?thesis
      using step-cap by (simp add: fp-seen-def Let-def)
  next
    case False
    have W'-cap: card ?W' ≤ cap
      using False by simp
    show ?thesis
      using Suc.IH[OF F'-W' W'-subset Suc.prems(3) W'-cap] False
      by (simp add: fp-seen-def Let-def)
  qed
qed

theorem find-pivots-seen-capped-card-le:
  assumes S-subset: S ⊆ V
    and degree: outdegree-le Δ
    and S-cap: card S ≤ cap
  shows card (find-pivots-seen-capped k cap d S B) ≤ Suc Δ * cap
  unfolding find-pivots-seen-capped-def
  by (rule fp-iter-capped-seen-card-le[OF subset-refl S-subset degree S-cap])
```

```
theorem fp-iter-capped-scan-cost-le:
  assumes F-subset-W: F ⊆ W
    and W-subset: W ⊆ V
    and degree: edge-outdegree-le Δ
    and W-cap: card W ≤ cap
  shows fp-iter-capped-scan-cost n cap d F W B ≤ n * Δ * cap
  using assms
proof (induction n arbitrary: d F W)
  case 0
  then show ?case
    by simp
next
  case (Suc n)
  let ?d' = relax-frontier d F
  let ?F' = fp-next d F B
  let ?W' = W ∪ ?F'
  have F-subset: F ⊆ V
    using Suc.prems(1,2) by blast
  have finite-W: finite W
    using finite-subset[OF Suc.prems(2) finite-V] .
  have card-F-le-W: card F ≤ card W
    using card-mono[OF finite-W Suc.prems(1)] .
  have card-F-le-cap: card F ≤ cap
    using card-F-le-W Suc.prems(4) by linarith
  have round-le: card (outgoing-edges F) ≤ Δ * cap
  proof −
    have card (outgoing-edges F) ≤ Δ * card F
      using card-outgoing-edges-le[OF F-subset Suc.prems(3)] .
    also have ... ≤ Δ * cap
      using card-F-le-cap by simp
    finally show ?thesis .
  qed
  show ?case
  proof (cases card ?W' > cap)
    case True
    have card (outgoing-edges F) ≤ Suc n * Δ * cap
    proof −
      have Δ * cap ≤ Suc n * (Δ * cap)
        by simp
      also have ... = Suc n * Δ * cap
        by (simp add: algebra-simps)
      finally show ?thesis
        using round-le by linarith
    qed
    then show ?thesis
      using True by (simp add: Let-def)
  next
    case False
    have F'-W': ?F' ⊆ ?W'
```

```
    by blast
  have F'-subset: ?F' ⊆ V
    using fp-next-subset-V .
  have W'-subset: ?W' ⊆ V
    using Suc.prems(2) F'-subset by blast
  have W'-cap: card ?W' ≤ cap
    using False by simp
  have tail-le:
    fp-iter-capped-scan-cost n cap ?d' ?F' ?W' B ≤ n * Δ * cap
    using Suc.IH[OF F'-W' W'-subset Suc.prems(3) W'-cap] .
  have card (outgoing-edges F) +
      fp-iter-capped-scan-cost n cap ?d' ?F' ?W' B
      ≤ Δ * cap + n * Δ * cap
    using round-le tail-le by simp
  also have ... = Suc n * Δ * cap
    by (simp add: algebra-simps)
  finally show ?thesis
    using False by (simp add: Let-def)
 qed
qed

theorem find-pivots-capped-scan-cost-le:
 assumes S-subset: S ⊆ V
   and degree: edge-outdegree-le Δ
   and S-cap: card S ≤ cap
 shows fp-iter-capped-scan-cost k cap d S S B ≤ k * Δ * cap
 by (rule fp-iter-capped-scan-cost-le[OF subset-refl S-subset degree S-cap])

theorem fp-iter-capped-scan-cost-le-seen:
 assumes F-subset-W: F ⊆ W
   and W-subset: W ⊆ V
   and degree: edge-outdegree-le Δ
 shows fp-iter-capped-scan-cost n cap d F W B ≤
   n * Δ * card (fp-seen (fp-iter-capped n cap d F W B))
 using F-subset-W W-subset
proof (induction n arbitrary: d F W)
 case 0
 then show ?case
   unfolding fp-iter-capped-scan-cost.simps by simp
next
 case (Suc n)
 let ?d' = relax-frontier d F
 let ?F' = fp-next d F B
 let ?W' = W ∪ ?F'
 let ?final = fp-seen (fp-iter-capped (Suc n) cap d F W B)
 have F-subset-V: F ⊆ V
   using Suc.prems by blast
 have finite-final: finite ?final
   using fp-iter-capped-seen-subset-V[OF F-subset-V Suc.prems(2)] finite-V
```

```
  by (rule finite-subset)
have W-final: W ⊆ ?final
  by (rule fp-iter-capped-seen-initial-subset)
have F-final: F ⊆ ?final
  using Suc.prems(1) W-final by blast
have card-F-le-final: card F ≤ card ?final
  by (rule card-mono[OF finite-final F-final])
have round-le:
  card (outgoing-edges F) ≤ Δ * card ?final
proof −
  have card (outgoing-edges F) ≤ Δ * card F
    by (rule card-outgoing-edges-le[OF F-subset-V degree])
  also have ... ≤ Δ * card ?final
    using card-F-le-final by simp
  finally show ?thesis .
qed
show ?case
proof (cases card ?W' > cap)
  case True
  have card (outgoing-edges F) ≤ Suc n * Δ * card ?final
  proof −
    have Δ * card ?final ≤ Suc n * (Δ * card ?final)
      by simp
    then show ?thesis
      using round-le by linarith
  qed
  then show ?thesis
    using True by (simp add: Let-def)
next
  case False
  have F'-subset-W': ?F' ⊆ ?W'
    by blast
  have W'-subset: ?W' ⊆ V
    using Suc.prems fp-next-subset-V by blast
  have tail:
    fp-iter-capped-scan-cost n cap ?d' ?F' ?W' B ≤
      n * Δ * card (fp-seen (fp-iter-capped n cap ?d' ?F' ?W' B))
    by (rule Suc.IH[OF F'-subset-W' W'-subset])
  have final-eq:
    ?final = fp-seen (fp-iter-capped n cap ?d' ?F' ?W' B)
    using False by (simp add: fp-seen-def Let-def)
  have card (outgoing-edges F) +
      fp-iter-capped-scan-cost n cap ?d' ?F' ?W' B ≤
      Δ * card ?final + n * Δ * card ?final
    using round-le tail final-eq by simp
  also have ... = Suc n * Δ * card ?final
    by (simp add: algebra-simps)
  finally show ?thesis
    using False by (simp add: Let-def)
```

```
  qed
qed

theorem find-pivots-capped-scan-cost-le-seen:
  assumes S-subset: S ⊆ V
    and degree: edge-outdegree-le Δ
  shows fp-iter-capped-scan-cost k cap d S S B ≤
    k * Δ * card (find-pivots-seen-capped k cap d S B)
  unfolding find-pivots-seen-capped-def
  by (rule fp-iter-capped-scan-cost-le-seen[OF subset-refl S-subset degree])

theorem fp-iter-capped-scan-cost-le-outgoing-seen:
  assumes F-subset-W: F ⊆ W
  shows fp-iter-capped-scan-cost n cap d F W B ≤
    n * card (outgoing-edges (fp-seen (fp-iter-capped n cap d F W B)))
  using F-subset-W
proof (induction n arbitrary: d F W)
  case 0
  then show ?case
    by simp
next
  case (Suc n)
  let ?d' = relax-frontier d F
  let ?F' = fp-next d F B
  let ?W' = W ∪ ?F'
  let ?final = fp-seen (fp-iter-capped (Suc n) cap d F W B)
  have W-final: W ⊆ ?final
    by (rule fp-iter-capped-seen-initial-subset)
  have F-final: F ⊆ ?final
    using Suc.prems W-final by blast
  have outgoing-subset: outgoing-edges F ⊆ outgoing-edges ?final
    using F-final unfolding outgoing-edges-def by blast
  have round-le:
    card (outgoing-edges F) ≤ card (outgoing-edges ?final)
    by (rule card-mono[OF finite-outgoing-edges outgoing-subset])
  show ?case
  proof (cases card ?W' > cap)
    case True
    have card (outgoing-edges F) ≤
        Suc n * card (outgoing-edges ?final)
    proof −
      have card (outgoing-edges ?final) ≤
          Suc n * card (outgoing-edges ?final)
        by simp
      then show ?thesis
        using round-le by linarith
    qed
    then show ?thesis
      using True by (simp add: Let-def)
```

```
  next
    case False
    have tail:
      fp-iter-capped-scan-cost n cap ?d' ?F' ?W' B ≤
        n * card (outgoing-edges (fp-seen (fp-iter-capped n cap ?d' ?F' ?W' B)))
      by (rule Suc.IH) blast
    have final-eq:
      ?final = fp-seen (fp-iter-capped n cap ?d' ?F' ?W' B)
      using False by (simp add: fp-seen-def Let-def)
    have card (outgoing-edges F) +
        fp-iter-capped-scan-cost n cap ?d' ?F' ?W' B ≤
        card (outgoing-edges ?final) +
        n * card (outgoing-edges ?final)
      using round-le tail final-eq by simp
    also have ... = Suc n * card (outgoing-edges ?final)
      by (simp add: algebra-simps)
    finally show ?thesis
      using False by (simp add: Let-def)
  qed
qed

theorem find-pivots-capped-scan-cost-le-outgoing-seen:
  shows fp-iter-capped-scan-cost k cap d S S B ≤
    k * card (outgoing-edges (find-pivots-seen-capped k cap d S B))
  unfolding find-pivots-seen-capped-def
  by (rule fp-iter-capped-scan-cost-le-outgoing-seen[OF subset-refl])
```

The capped search scans only edges leaving frontiers that have already been absorbed into the final seen set. The theorem *fp-iter-capped-scan-cost ?k ?cap ?d ?S ?S ?B ≤ ?k * card (outgoing-edges (find-pivots-seen-capped ?k ?cap ?d ?S ?B))* is the compact cost statement used later: *fp-iter-capped-scan-cost* is bounded by the number of rounds times the outgoing-edge volume of *find-pivots-seen-capped*. This is deliberately stated in terms of the final seen set rather than a sum over intermediate frontiers, because the top-level analysis controls the size of that set through the cap.

```
lemma fp-iter-capped-eq-fp-iter-if-final-within-cap:
  assumes final-cap: card (fp-seen (fp-iter-capped n cap d F W B)) ≤ cap
  shows fp-iter-capped n cap d F W B = fp-iter n d F W B
  using final-cap
proof (induction n arbitrary: d F W)
  case 0
  then show ?case
    by simp
next
  case (Suc n)
  let ?d' = relax-frontier d F
  let ?F' = fp-next d F B
  let ?W' = W ∪ ?F'
  show ?case
```

```
  proof (cases card ?W' > cap)
    case True
    then have fp-seen (fp-iter-capped (Suc n) cap d F W B) = ?W'
      by (simp add: fp-seen-def Let-def)
    then have False
      using Suc.prems True by simp
    then show ?thesis
      by blast
  next
    case False
    have tail-cap:
      card (fp-seen (fp-iter-capped n cap ?d' ?F' ?W' B)) ≤ cap
      using Suc.prems False by (simp add: fp-seen-def Let-def)
    have fp-iter-capped n cap ?d' ?F' ?W' B = fp-iter n ?d' ?F' ?W' B
      using Suc.IH[OF tail-cap] .
    then show ?thesis
      using False by (simp add: Let-def)
  qed
qed

lemma fp-iter-seen-subset-V:
  assumes F-subset: F ⊆ V
    and W-subset: W ⊆ V
  shows fp-seen (fp-iter n d F W B) ⊆ V
  using assms
proof (induction n arbitrary: d F W)
  case 0
  then show ?case
    unfolding fp-seen-def by simp
next
  case (Suc n)
  let ?F' = fp-next d F B
  let ?W' = W ∪ ?F'
  have ?F' ⊆ V
    using fp-next-subset-V .
  then have ?W' ⊆ V
    using Suc.prems by blast
  then have fp-seen (fp-iter n (relax-frontier d F) ?F' ?W' B) ⊆ V
    using Suc.IH[OF ‹?F' ⊆ V›] by blast
  then show ?case
    by (simp add: fp-seen-def Let-def)
qed

lemma find-pivots-seen-subset-V:
  assumes S ⊆ V
  shows find-pivots-seen k d S B ⊆ V
  unfolding find-pivots-seen-def
  using fp-iter-seen-subset-V[OF assms assms] .
```

lemma *finite-find-pivots-seen* [*simp*]:
  assumes $S \subseteq V$
  shows *finite* (*find-pivots-seen k d S B*)
  using *finite-subset*[*OF find-pivots-seen-subset-V*[*OF assms*] *finite-V*] **.**

lemma *fp-next-tight-successor*:
  assumes *sound*: *sound-label d*
    and *uF*: $u \in F$
    and *complete-u*: $d\ u = dist\ s\ u$
    and *tight*: *tight-edge-step u v*
    and *below-v*: *below-bound* (*dist s v*) *B*
  shows $v \in$ *fp-next d F B*
proof −
  have *edge*: $(u, v) \in E$ **and** *reach-u*: *reachable s u*
    and *dist-v*: $dist\ s\ v = dist\ s\ u + w\ u\ v$
    using *tight* **unfolding** *tight-edge-step-def* **by** *auto*
  have *vV*: $v \in V$
    using *edge edge-in-V* **by** *blast*
  have *reach-v*: *reachable s v*
    using *dist-triangle-edge*[*OF edge reach-u*] **by** *blast*
  have $dist\ s\ v \leq d\ v$
    using *sound vV reach-v* **unfolding** *sound-label-def* **by** *blast*
  then have $d\ u + w\ u\ v \leq d\ v$
    using *complete-u dist-v* **by** *simp*
  moreover have *below-bound* $(d\ u + w\ u\ v)\ B$
    using *complete-u dist-v below-v* **by** *simp*
  ultimately show *?thesis*
    using *uF edge vV* **unfolding** *fp-next-def* **by** *blast*
qed

lemma *fp-step-tight-successor-complete*:
  assumes *sound*: *sound-label d*
    and *uF*: $u \in F$
    and *frontier-reaches*: $\bigwedge x.\ x \in F \Longrightarrow$ *reachable s x*
    and *complete-u*: $d\ u = dist\ s\ u$
    and *tight*: *tight-edge-step u v*
    and *below-v*: *below-bound* (*dist s v*) *B*
  shows *sound-label* (*relax-frontier d F*)
    and *relax-frontier d F v* = *dist s v*
    and $v \in$ *fp-next d F B*
proof −
  show *sound-label* (*relax-frontier d F*)
    using *relax-frontier-tight-successor-complete*(*1*)
      [*OF sound uF frontier-reaches complete-u tight*] **.**
  show *relax-frontier d F v* = *dist s v*
    using *relax-frontier-tight-successor-complete*(*2*)
      [*OF sound uF frontier-reaches complete-u tight*] **.**
  show $v \in$ *fp-next d F B*
    using *fp-next-tight-successor*[*OF sound uF complete-u tight below-v*] **.**

qed

lemma *fp-iter-seen-mono*:
assumes $x \in W$
shows $x \in$ *fp-seen* (*fp-iter n d F W B*)
using *assms*
proof (*induction n arbitrary*: *d F W*)
case *0*
then show *?case*
unfolding *fp-seen-def* **by** *simp*
next
case (*Suc n*)
let *?d'* = *relax-frontier d F*
let *?F'* = *fp-next d F B*
let *?W'* = $W \cup$ *?F'*
have $x \in$ *?W'*
using *Suc.prems* **by** *blast*
then have $x \in$ *fp-seen* (*fp-iter n ?d' ?F' ?W' B*)
using *Suc.IH* **by** *blast*
then show *?case*
by (*simp add*: *fp-seen-def Let-def*)
qed

lemma *relax-frontier-preserves-complete-sound*:
assumes *sound*: *sound-label d*
and *complete-x*: *d x* = *dist s x*
and *reaches*: $\bigwedge u.\ u \in F \Longrightarrow$ *reachable s u*
shows *relax-frontier d F x* = *dist s x*
proof −
let *?es* = *edge-list-of* (*outgoing-edges F*)
have *set-es*: *set ?es* = *outgoing-edges F*
using *edge-list-of-properties*(*1*)[*OF finite-outgoing-edges*] **.**
have *edges*: $\bigwedge u\ v.\ (u, v) \in$ *set ?es* $\Longrightarrow (u, v) \in E$
using *set-es* **unfolding** *outgoing-edges-def* **by** *blast*
have *reaches-es*: $\bigwedge u\ v.\ (u, v) \in$ *set ?es* $\Longrightarrow$ *reachable s u*
using *set-es reaches* **unfolding** *outgoing-edges-def* **by** *blast*
show *?thesis*
unfolding *relax-frontier-def*
using *relax-edges-preserves-complete-sound*[*OF sound complete-x edges reaches-es*]
.
qed

lemma *relax-frontier-preserves-bmssp-pre-full*:
assumes *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *reaches*: $\bigwedge u.\ u \in F \Longrightarrow$ *reachable s u*
shows *bmssp-pre-full* (*relax-frontier d F*) *S B*
proof −
have *S-subset*: $S \subseteq V$

```
    using pre unfolding bmssp-pre-full-def by blast
  have cover:
    ⋀v. ⟦v ∈ V; reachable s v; below-bound (dist s v) B;
       relax-frontier d F v ≠ dist s v⟧
      ⟹ through-complete (relax-frontier d F) S v
  proof −
    fix v
    assume vV: v ∈ V and reach-v: reachable s v
      and below-v: below-bound (dist s v) B
      and incomplete-v: relax-frontier d F v ≠ dist s v
    have d-incomplete: d v ≠ dist s v
    proof
      assume complete-v: d v = dist s v
      have relax-frontier d F v = dist s v
        by (rule relax-frontier-preserves-complete-sound
          [OF sound complete-v reaches])
      then show False
        using incomplete-v by simp
    qed
    have through-d: through-complete d S v
      using pre vV reach-v below-v d-incomplete
      unfolding bmssp-pre-full-def by blast
    then obtain u p where uS: u ∈ S
      and u-complete: u ∈ complete-vertices d
      and sp: shortest-walk s p v
      and up: u ∈ set p
      unfolding through-complete-def through-def by blast
    have uV: u ∈ V and reach-u: reachable s u
      and du: d u = dist s u
      using u-complete unfolding complete-vertices-def by auto
    have relax-u: relax-frontier d F u = dist s u
      by (rule relax-frontier-preserves-complete-sound[OF sound du reaches])
    have u ∈ complete-vertices (relax-frontier d F)
      using uV reach-u relax-u unfolding complete-vertices-def by blast
    then have through (S ∩ complete-vertices (relax-frontier d F)) v
      using uS sp up unfolding through-def by blast
    then show through-complete (relax-frontier d F) S v
      unfolding through-complete-def .
  qed
  show ?thesis
    using S-subset cover unfolding bmssp-pre-full-def by blast
qed

lemma fp-iter-capped-preserves-bmssp-pre-full:
  assumes sound: sound-label d
    and pre: bmssp-pre-full d S B0
    and reaches: ⋀u. u ∈ F ⟹ reachable s u
  shows bmssp-pre-full (fp-label (fp-iter-capped n cap d F W B)) S B0
  using assms
```

```
proof (induction n arbitrary: d F W)
  case 0
  then show ?case
    unfolding fp-label-def by simp
next
  case (Suc n)
  let ?d' = relax-frontier d F
  let ?F' = fp-next d F B
  let ?W' = W ∪ ?F'
  have sound': sound-label ?d'
    using relax-frontier-sound[OF Suc.prems(1,3)] .
  have pre': bmssp-pre-full ?d' S B0
    by (rule relax-frontier-preserves-bmssp-pre-full
      [OF Suc.prems(1,2,3)])
  have reaches': ⋀u. u ∈ ?F' ⟹ reachable s u
    using fp-next-reaches[OF Suc.prems(3)] by blast
  show ?case
  proof (cases card ?W' > cap)
    case True
    then show ?thesis
      using pre' by (simp add: fp-label-def Let-def)
  next
    case False
    show ?thesis
      using Suc.IH[OF sound' pre' reaches'] False
      by (simp add: fp-label-def Let-def)
  qed
qed

lemma find-pivots-label-capped-preserves-source-pre:
  assumes sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀u. u ∈ S ⟹ reachable s u
  shows bmssp-pre-full (find-pivots-label-capped k cap d S B) S B
  unfolding find-pivots-label-capped-def
  by (rule fp-iter-capped-preserves-bmssp-pre-full
    [OF sound pre S-reaches])

lemma fp-iter-capped-label-le:
  fp-label (fp-iter-capped n cap d F W B) x ≤ d x
proof (induction n arbitrary: d F W)
  case 0
  then show ?case
    unfolding fp-label-def by simp
next
  case (Suc n)
  let ?d' = relax-frontier d F
  let ?F' = fp-next d F B
  let ?W' = W ∪ ?F'
```

```
  show ?case
  proof (cases card ?W' > cap)
    case True
    then show ?thesis
      using relax-frontier-le[of d F x]
      by (simp add: fp-label-def Let-def)
  next
    case False
    have tail: fp-label (fp-iter-capped n cap ?d' ?F' ?W' B) x ≤ ?d' x
      by (rule Suc.IH)
    also have ... ≤ d x
      by (rule relax-frontier-le)
    finally show ?thesis
      using False by (simp add: fp-label-def Let-def)
  qed
qed

lemma find-pivots-label-capped-le:
  find-pivots-label-capped k cap d S B x ≤ d x
  unfolding find-pivots-label-capped-def
  by (rule fp-iter-capped-label-le)

lemma fp-iter-capped-label-sound:
  assumes sound: sound-label d
    and reaches: ⋀u. u ∈ F ⟹ reachable s u
  shows sound-label (fp-label (fp-iter-capped n cap d F W B))
  using assms
proof (induction n arbitrary: d F W)
  case 0
  then show ?case
    unfolding fp-label-def by simp
next
  case (Suc n)
  let ?d' = relax-frontier d F
  let ?F' = fp-next d F B
  let ?W' = W ∪ ?F'
  have sound': sound-label ?d'
    using relax-frontier-sound[OF Suc.prems] .
  have reaches': ⋀u. u ∈ ?F' ⟹ reachable s u
    using fp-next-reaches[OF Suc.prems(2)] by blast
  show ?case
  proof (cases card ?W' > cap)
    case True
    then show ?thesis
      using sound' by (simp add: fp-label-def Let-def)
  next
    case False
    show ?thesis
      using Suc.IH[OF sound' reaches'] False by (simp add: fp-label-def Let-def)
```

```
  qed
qed

lemma fp-iter-capped-preserves-complete-sound:
  assumes sound: sound-label d
    and complete-x: d x = dist s x
    and reaches: ⋀u. u ∈ F ⟹ reachable s u
  shows fp-label (fp-iter-capped n cap d F W B) x = dist s x
  using assms
proof (induction n arbitrary: d F W)
  case 0
  then show ?case
    unfolding fp-label-def by simp
next
  case (Suc n)
  let ?d' = relax-frontier d F
  let ?F' = fp-next d F B
  let ?W' = W ∪ ?F'
  have sound': sound-label ?d'
    using relax-frontier-sound[OF Suc.prems(1,3)] .
  have complete': ?d' x = dist s x
    using relax-frontier-preserves-complete-sound[OF Suc.prems] .
  have reaches': ⋀u. u ∈ ?F' ⟹ reachable s u
    using fp-next-reaches[OF Suc.prems(3)] by blast
  show ?case
  proof (cases card ?W' > cap)
    case True
    then show ?thesis
      using complete' by (simp add: fp-label-def Let-def)
  next
    case False
    show ?thesis
      using Suc.IH[OF sound' complete' reaches'] False
      by (simp add: fp-label-def Let-def)
  qed
qed

theorem find-pivots-capped-overflow-establishes-pre:
  assumes sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and overflow: card (find-pivots-seen-capped k cap d S B) > cap
  shows bmssp-pre-full
    (find-pivots-label-capped k cap d S B)
    (find-pivots-pivots-capped k cap d S B) B
proof −
  let ?d' = find-pivots-label-capped k cap d S B
  have pivots-eq: find-pivots-pivots-capped k cap d S B = S
    using overflow unfolding find-pivots-pivots-capped-def by simp
```

have *S-subset*: $S \subseteq V$
using *pre* **unfolding** *bmssp-pre-full-def* **by** *blast*
have *preserve-complete*:
$\bigwedge x.\ d\ x = dist\ s\ x \Longrightarrow ?d'\ x = dist\ s\ x$
unfolding *find-pivots-label-capped-def*
by (*rule fp-iter-capped-preserves-complete-sound*[*OF sound - S-reaches*])
have *cover*:
$\bigwedge v.\ [\![v \in V;\ reachable\ s\ v;\ below\text{-}bound\ (dist\ s\ v)\ B;$
$?d'\ v \neq dist\ s\ v]\!] \Longrightarrow through\text{-}complete\ ?d'\ S\ v$
proof −
fix *v*
assume *vV*: $v \in V$
and *reach-v*: *reachable s v*
and *below-v*: *below-bound* (*dist s v*) *B*
and *d'-incomplete*: $?d'\ v \neq dist\ s\ v$
have *d-incomplete*: $d\ v \neq dist\ s\ v$
proof
assume $d\ v = dist\ s\ v$
then have $?d'\ v = dist\ s\ v$
by (*rule preserve-complete*)
with *d'-incomplete* **show** *False*
by *simp*
qed
have *through-complete d S v*
using *pre vV reach-v below-v d-incomplete* **unfolding** *bmssp-pre-full-def* **by** *blast*
then obtain *u p* **where** *uS*: $u \in S$ **and** *ucomp*: $u \in complete\text{-}vertices\ d$
and *sp*: *shortest-walk s p v* **and** *u-path*: $u \in set\ p$
unfolding *through-complete-def through-def* **by** *blast*
have *uV*: $u \in V$ **and** *reach-u*: *reachable s u* **and** *complete-u*: $d\ u = dist\ s\ u$
using *ucomp* **unfolding** *complete-vertices-def* **by** *auto*
have *complete-u'*: $?d'\ u = dist\ s\ u$
using *preserve-complete*[*OF complete-u*] **.**
have $u \in complete\text{-}vertices\ ?d'$
using *uV reach-u complete-u'* **unfolding** *complete-vertices-def* **by** *blast*
then have *through* $(S \cap complete\text{-}vertices\ ?d')\ v$
using *uS sp u-path* **unfolding** *through-def* **by** *blast*
then show *through-complete ?d' S v*
unfolding *through-complete-def* **.**
qed
show *?thesis*
unfolding *bmssp-pre-full-def*
proof
show *find-pivots-pivots-capped k cap d S B* $\subseteq V$
using *pivots-eq S-subset* **by** *simp*
next
show $\forall\, v{\in}V.\ reachable\ s\ v \longrightarrow below\text{-}bound\ (dist\ s\ v)\ B \longrightarrow$
$?d'\ v \neq dist\ s\ v \longrightarrow$
$through\text{-}complete\ ?d'\ (find\text{-}pivots\text{-}pivots\text{-}capped\ k\ cap\ d\ S\ B)\ v$

```
    proof (intro ballI impI)
      fix v
      assume vV: v ∈ V
        and reach-v: reachable s v
        and below-v: below-bound (dist s v) B
        and incomplete-v: ?d' v ≠ dist s v
      have through-complete ?d' S v
        using cover[OF vV reach-v below-v incomplete-v] .
      then show through-complete ?d' (find-pivots-pivots-capped k cap d S B) v
        using pivots-eq by simp
    qed
  qed
qed

lemma fp-iter-preserves-complete-sound:
  assumes sound: sound-label d
    and complete-x: d x = dist s x
    and reaches: ⋀u. u ∈ F ⟹ reachable s u
  shows fp-label (fp-iter n d F W B) x = dist s x
  using sound complete-x reaches
proof (induction n arbitrary: d F W)
  case 0
  then show ?case
    unfolding fp-label-def by simp
next
  case (Suc n)
  let ?d' = relax-frontier d F
  let ?F' = fp-next d F B
  let ?W' = W ∪ ?F'
  have sound': sound-label ?d'
    using relax-frontier-sound[OF Suc.prems(1,3)] .
  have complete': ?d' x = dist s x
    using relax-frontier-preserves-complete-sound[OF Suc.prems] .
  have reaches': ⋀u. u ∈ ?F' ⟹ reachable s u
    using fp-next-reaches[OF Suc.prems(3)] by blast
  show ?case
    using Suc.IH[OF sound' complete' reaches'] by (simp add: fp-label-def Let-def)
qed

lemma fp-iter-one-tight-step-completion:
  assumes sound: sound-label d
    and uF: u ∈ F
    and frontier-reaches: ⋀x. x ∈ F ⟹ reachable s x
    and complete-u: d u = dist s u
    and tight: tight-edge-step u v
    and below-v: below-bound (dist s v) B
  shows fp-label (fp-iter (Suc n) d F W B) v = dist s v
    and v ∈ fp-seen (fp-iter (Suc n) d F W B)
proof −
```

```
  let ?d' = relax-frontier d F
  let ?F' = fp-next d F B
  let ?W' = W ∪ ?F'
  have sound': sound-label ?d'
    using fp-step-tight-successor-complete(1)
      [OF sound uF frontier-reaches complete-u tight below-v] .
  have complete-v: ?d' v = dist s v
    using fp-step-tight-successor-complete(2)
      [OF sound uF frontier-reaches complete-u tight below-v] .
  have vF': v ∈ ?F'
    using fp-step-tight-successor-complete(3)
      [OF sound uF frontier-reaches complete-u tight below-v] .
  have reaches': ⋀x. x ∈ ?F' ⟹ reachable s x
    using fp-next-reaches[OF frontier-reaches] by blast
  have tail-complete:
    fp-label (fp-iter n ?d' ?F' ?W' B) v = dist s v
    by (rule fp-iter-preserves-complete-sound[OF sound' complete-v reaches'])
  show fp-label (fp-iter (Suc n) d F W B) v = dist s v
    using tail-complete by (simp add: fp-label-def Let-def)
  have v ∈ ?W'
    using vF' by blast
  then have tail-seen: v ∈ fp-seen (fp-iter n ?d' ?F' ?W' B)
    using fp-iter-seen-mono[of v ?W' n ?d' ?F' B] by blast
  then show v ∈ fp-seen (fp-iter (Suc n) d F W B)
    using tail-seen by (simp add: fp-seen-def Let-def)
qed

lemma fp-iter-tight-path-completion:
  assumes sound: sound-label d
    and frontier-reaches: ⋀x. x ∈ F ⟹ reachable s x
    and F-seen: F ⊆ W
    and nonempty: p ≠ []
    and start: hd p ∈ F
    and complete-start: d (hd p) = dist s (hd p)
    and tight: successively tight-edge-step p
    and below: ⋀x. x ∈ set p ⟹ below-bound (dist s x) B
    and len: length (path-edges p) ≤ n
  shows fp-label (fp-iter n d F W B) (last p) = dist s (last p) ∧
    last p ∈ fp-seen (fp-iter n d F W B)
  using assms
proof (induction p arbitrary: n d F W rule: path-edges.induct)
  case 1
  then show ?case
    by simp
next
  case (2 x)
  have sound0: sound-label d
    using 2.prems by blast
  have reaches0: ⋀z. z ∈ F ⟹ reachable s z
```

```
    using 2.prems by blast
  have complete0: d x = dist s x
    using 2.prems by simp
  have complete-x: fp-label (fp-iter n d F W B) x = dist s x
    using fp-iter-preserves-complete-sound[OF sound0 complete0 reaches0] .
  have seen-x: x ∈ fp-seen (fp-iter n d F W B)
  proof −
    have x ∈ W
      using 2.prems by auto
    then show ?thesis
      using fp-iter-seen-mono by blast
  qed
  show ?case
    using complete-x seen-x by simp
next
  case (3 x y xs)
  show ?case
  proof (cases n)
    case 0
    then show ?thesis
      using 3.prems(9) by simp
  next
    case (Suc m)
    let ?d' = relax-frontier d F
    let ?F' = fp-next d F B
    let ?W' = W ∪ ?F'
    have sound0: sound-label d
      using 3.prems by blast
    have reaches0: ⋀z. z ∈ F ⟹ reachable s z
      using 3.prems by blast
    have xF: x ∈ F
      using 3.prems by simp
    have complete-x: d x = dist s x
      using 3.prems by simp
    have step: tight-edge-step x y
      using 3.prems by (simp add: successively-Cons)
    have below-y: below-bound (dist s y) B
      using 3.prems by simp
    have sound': sound-label ?d'
      using fp-step-tight-successor-complete(1)
        [OF sound0 xF reaches0 complete-x step below-y] .
    have complete-y: ?d' y = dist s y
      using fp-step-tight-successor-complete(2)
        [OF sound0 xF reaches0 complete-x step below-y] .
    have yF': y ∈ ?F'
      using fp-step-tight-successor-complete(3)
        [OF sound0 xF reaches0 complete-x step below-y] .
    have reaches': ⋀z. z ∈ ?F' ⟹ reachable s z
      using fp-next-reaches[OF reaches0] by blast
```

```
    have F-seen': ?F' ⊆ ?W'
      by blast
    have tail-nonempty: y # xs ≠ []
      by simp
    have tail-start: hd (y # xs) ∈ ?F'
      using yF' by simp
    have tail-complete-start: ?d' (hd (y # xs)) = dist s (hd (y # xs))
      using complete-y by simp
    have tail-tight: successively tight-edge-step (y # xs)
      using 3.prems by (simp add: successively-Cons)
    have tail-below: ⋀z. z ∈ set (y # xs) ⟹ below-bound (dist s z) B
      using 3.prems by simp
    have tail-len: length (path-edges (y # xs)) ≤ m
      using 3.prems Suc by simp
    have tail:
      fp-label (fp-iter m ?d' ?F' ?W' B) (last (y # xs)) =
        dist s (last (y # xs)) ∧
       last (y # xs) ∈ fp-seen (fp-iter m ?d' ?F' ?W' B)
      using 3.IH[of ?d' ?F' ?W' m,
          OF sound' reaches' F-seen' tail-nonempty tail-start tail-complete-start
          tail-tight tail-below tail-len]
      by simp
    then show ?thesis
      using Suc by (simp add: fp-label-def fp-seen-def Let-def)
  qed
qed

corollary find-pivots-short-tight-path-complete:
  assumes sound: sound-label d
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and nonempty: p ≠ []
    and start: hd p ∈ S
    and complete-start: d (hd p) = dist s (hd p)
    and tight: successively tight-edge-step p
    and below: ⋀x. x ∈ set p ⟹ below-bound (dist s x) B
    and len: length (path-edges p) ≤ k
  shows find-pivots-label k d S B (last p) = dist s (last p)
    and last p ∈ find-pivots-seen k d S B
proof −
  have combined:
    fp-label (fp-iter k d S S B) (last p) = dist s (last p) ∧
      last p ∈ fp-seen (fp-iter k d S S B)
    using fp-iter-tight-path-completion
      [OF sound S-reaches subset-refl nonempty start complete-start tight below len] .
  show find-pivots-label k d S B (last p) = dist s (last p)
    using combined unfolding find-pivots-label-def by blast
  show last p ∈ find-pivots-seen k d S B
    using combined unfolding find-pivots-seen-def by blast
qed
```

lemma *find-pivots-short-paths-complete-on*:
assumes *sound*: *sound-label d*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *witnesses*:
$\bigwedge x.\ x \in X \Longrightarrow \exists\, p.\ p \neq [] \wedge$ *hd p* $\in S \wedge$ *last p* $= x \wedge$
d (*hd p*) = *dist s* (*hd p*) $\wedge$ *successively tight-edge-step p* $\wedge$
$(\forall\, y \in$ *set p. below-bound* (*dist s y*) *B*) $\wedge$ *length* (*path-edges p*) $\leq k$
shows *complete-on* (*find-pivots-label k d S B*) *X* $\wedge$
X $\subseteq$ *find-pivots-seen k d S B*
proof −
have *complete*: *complete-on* (*find-pivots-label k d S B*) *X*
proof (*unfold complete-on-def*, *intro ballI impI*)
fix *x*
assume *xX*: $x \in X$
obtain *p* **where** *p*:
$p \neq []$ *hd p* $\in S$ *last p* $= x$
d (*hd p*) = *dist s* (*hd p*)
successively tight-edge-step p
$\forall\, y \in$ *set p. below-bound* (*dist s y*) *B*
length (*path-edges p*) $\leq k$
using *witnesses*[*OF xX*] **by** *blast*
have *find-pivots-label k d S B* (*last p*) = *dist s* (*last p*)
using *find-pivots-short-tight-path-complete*(*1*)
[*OF sound S-reaches p*(*1*,*2*,*4*,*5*) - *p*(*7*)]
p(*6*) **by** *blast*
then show *reachable s x* $\Longrightarrow$ *find-pivots-label k d S B x* = *dist s x*
using *p*(*3*) **by** *simp*
qed
have *seen*: *X* $\subseteq$ *find-pivots-seen k d S B*
proof
fix *x*
assume *xX*: $x \in X$
obtain *p* **where** *p*:
$p \neq []$ *hd p* $\in S$ *last p* $= x$
d (*hd p*) = *dist s* (*hd p*)
successively tight-edge-step p
$\forall\, y \in$ *set p. below-bound* (*dist s y*) *B*
length (*path-edges p*) $\leq k$
using *witnesses*[*OF xX*] **by** *blast*
have *last p* $\in$ *find-pivots-seen k d S B*
using *find-pivots-short-tight-path-complete*(*2*)
[*OF sound S-reaches p*(*1*,*2*,*4*,*5*) - *p*(*7*)]
p(*6*) **by** *blast*
then show *x* $\in$ *find-pivots-seen k d S B*
using *p*(*3*) **by** *simp*
qed
show *?thesis*
using *complete seen* **by** *blast*

qed

The path-completion lemmas are the semantic core of FindPivots. A short tight witness is a shortest-path prefix, already correct at its start, whose edges remain below the current bound. The round induction proves that such a witness is followed by the relaxation process: ⟦*sound-label ?d*; ⋀*x*. *x* ∈ *?S* ⟹ *reachable s x*; *?p* ≠ []; *hd ?p* ∈ *?S*; *?d* (*hd ?p*) = *local.dist s* (*hd ?p*); *successively tight-edge-step ?p*; ⋀*x*. *x* ∈ *set ?p* ⟹ *below-bound* (*local.dist s x*) *?B*; *length* (*path-edges ?p*) ≤ *?k*⟧ ⟹ *find-pivots-label ?k ?d ?S ?B* (*last ?p*) = *local.dist s* (*last ?p*)

⟦*sound-label ?d*; ⋀*x*. *x* ∈ *?S* ⟹ *reachable s x*; *?p* ≠ []; *hd ?p* ∈ *?S*; *?d* (*hd ?p*) = *local.dist s* (*hd ?p*); *successively tight-edge-step ?p*; ⋀*x*. *x* ∈ *set ?p* ⟹ *below-bound* (*local.dist s x*) *?B*; *length* (*path-edges ?p*) ≤ *?k*⟧ ⟹ *last ?p* ∈ *find-pivots-seen ?k ?d ?S ?B* records both the resulting label equality and membership in the seen set. The set-level lemma above packages the same fact for any family of witnesses. The next theorem instantiates that package with *find-pivots-short-vertices*, the set defined exactly by *short-tight-witness*, yielding completeness via *complete-on*.

theorem *find-pivots-completes-short-vertices*:
  assumes *sound*: *sound-label d*
    and *S-reaches*: ⋀*x*. *x* ∈ *S* ⟹ *reachable s x*
  shows *complete-on* (*find-pivots-label k d S B*)
      (*find-pivots-short-vertices k d S B*) ∧
    find-pivots-short-vertices k d S B ⊆ *find-pivots-seen k d S B*
proof (*rule find-pivots-short-paths-complete-on*[*OF sound S-reaches*])
  fix *x*
  assume *x* ∈ *find-pivots-short-vertices k d S B*
  then show ∃ *p*. *p* ≠ [] ∧ *hd p* ∈ *S* ∧ *last p* = *x* ∧
      d (*hd p*) = *dist s* (*hd p*) ∧ *successively tight-edge-step p* ∧
      (∀ *y*∈*set p*. *below-bound* (*dist s y*) *B*) ∧ *length* (*path-edges p*) ≤ *k*
    unfolding *find-pivots-short-vertices-def short-tight-witness-def* **by** *blast*
qed

lemma *card-set-take-distinct*:
  assumes *distinct xs k* ≤ *length xs*
  shows *card* (*set* (*take k xs*)) = *k*
  using *assms* **by** (*simp add*: *distinct-card*)

lemma *successively-take*:
  assumes *successively R xs*
  shows *successively R* (*take n xs*)
  unfolding *successively-conv-nth*
proof *clarify*
  fix *i*
  assume *i-take*: *Suc i* < *length* (*take n xs*)
  have *i-xs*: *Suc i* < *length xs*
    using *i-take* **by** *simp*

```
  have si-n: Suc i < n
    using i-take by simp
  have step: R (xs ! i) (xs ! Suc i)
    using assms i-xs unfolding successively-conv-nth by blast
  have take n xs ! i = xs ! i
    using si-n by simp
  moreover have take n xs ! Suc i = xs ! Suc i
    using si-n by simp
  ultimately show R (take n xs ! i) (take n xs ! Suc i)
    using step by simp
qed

lemma successively-drop:
  assumes successively R xs
  shows successively R (drop n xs)
  unfolding successively-conv-nth
proof clarify
  fix i
  assume i-drop: Suc i < length (drop n xs)
  have i-xs: Suc (n + i) < length xs
    using i-drop by simp
  have step: R (xs ! (n + i)) (xs ! Suc (n + i))
    using assms i-xs unfolding successively-conv-nth by blast
  have drop n xs ! i = xs ! (n + i)
    using i-drop by simp
  moreover have drop n xs ! Suc i = xs ! Suc (n + i)
    using i-drop by simp
  ultimately show R (drop n xs ! i) (drop n xs ! Suc i)
    using step by simp
qed

lemma below-bound-le-trans:
  assumes a ≤ b below-bound b B
  shows below-bound a B
  using assms by (cases B) auto

lemma find-pivots-pivots-capped-label-below:
  assumes below: ⋀x. x ∈ S ⟹ below-bound (d x) B
    and xP: x ∈ find-pivots-pivots-capped k cap d S B
  shows below-bound (find-pivots-label-capped k cap d S B x) B
proof −
  have xS: x ∈ S
    using xP unfolding find-pivots-pivots-capped-def
    by (auto split: if-splits)
  have le: find-pivots-label-capped k cap d S B x ≤ d x
    by (rule find-pivots-label-capped-le)
  show ?thesis
    by (rule below-bound-le-trans[OF le below[OF xS]])
qed
```

lemma *find-pivots-completes-short-tree-suffix*:
assumes *sound*: *sound-label d*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *reach-v*: *reachable s v*
and *below-v*: *below-bound* (*dist s v*) *B*
and *i*: $i <$ *length* (*shortest-path-to v*)
and *source-in-S*: *shortest-path-to v* ! *i* $\in S$
and *source-complete*: *d* (*shortest-path-to v* ! *i*) = *dist s* (*shortest-path-to v* ! *i*)
and *suffix-len*: *length* (*path-edges* (*drop i* (*shortest-path-to v*))) $\leq k$
shows *find-pivots-label k d S B v* = *dist s v*
and *v* $\in$ *find-pivots-seen k d S B*
proof −
let *?p* = *shortest-path-to v*
let *?q* = *drop i ?p*
have *sp*: *shortest-walk s ?p v*
using *shortest-path-to-shortest*[*OF reach-v*] **.**
have *tight-p*: *successively tight-edge-step ?p*
using *shortest-path-to-successively-tight*[*OF reach-v*] **.**
have *q-nonempty*: *?q* $\neq$ []
using *i* **by** *auto*
have *hd-q*: *hd ?q* = *?p* ! *i*
using *i* **by** (*simp add*: *hd-drop-conv-nth*)
have *last-q*: *last ?q* = *v*
using *sp i* **unfolding** *shortest-walk-def simple-walk-betw-def walk-betw-def*
by (*simp add*: *last-drop*)
have *tight-q*: *successively tight-edge-step ?q*
using *successively-drop*[*OF tight-p*, *of i*] **.**
have *below-q*: $\bigwedge x.\ x \in$ *set ?q* $\Longrightarrow$ *below-bound* (*dist s x*) *B*
proof −
fix *x*
assume *xq*: *x* $\in$ *set ?q*
have *x* $\in$ *set ?p*
using *in-set-dropD*[*OF xq*] **.**
then have *tree-path x v*
using *reach-v* **unfolding** *tree-path-def* **by** *blast*
then have *dist s x* $\leq$ *dist s v*
by (*rule tree-path-dist-le*)
then show *below-bound* (*dist s x*) *B*
using *below-bound-le-trans*[*OF - below-v*] **by** *blast*
qed
have *hd-q-S*: *hd ?q* $\in S$
using *hd-q source-in-S* **by** *simp*
have *hd-q-complete*: *d* (*hd ?q*) = *dist s* (*hd ?q*)
using *hd-q source-complete* **by** *simp*
have *label-last*: *find-pivots-label k d S B* (*last ?q*) = *dist s* (*last ?q*)
using *find-pivots-short-tight-path-complete*(*1*)
[*OF sound S-reaches q-nonempty hd-q-S hd-q-complete tight-q below-q suffix-len*]
.

```
  have seen-last: last ?q ∈ find-pivots-seen k d S B
    using find-pivots-short-tight-path-complete(2)
      [OF sound S-reaches q-nonempty hd-q-S hd-q-complete tight-q below-q suffix-len]
.
  show find-pivots-label k d S B v = dist s v
    using label-last last-q by simp
  show v ∈ find-pivots-seen k d S B
    using seen-last last-q by simp
qed

lemma find-pivots-pivot-if-k-seen-tree-vertices:
  assumes uS: u ∈ S
    and distinct: distinct xs
    and len: k ≤ length xs
    and in-tree: set (take k xs) ⊆ tree-of u
    and seen: set (take k xs) ⊆ find-pivots-seen k d S B
  shows u ∈ find-pivots-pivots k d S B
proof −
  let ?A = set (take k xs)
  have card-A: card ?A = k
    using card-set-take-distinct[OF distinct len] .
  have A-subset: ?A ⊆ tree-of u ∩ find-pivots-seen k d S B
    using in-tree seen by blast
  have card ?A ≤ card (tree-of u ∩ find-pivots-seen k d S B)
    using A-subset by (intro card-mono) simp-all
  then show ?thesis
    using uS card-A unfolding find-pivots-pivots-def by simp
qed

lemma finite-find-pivots-pivots [simp]:
  assumes S ⊆ V
  shows finite (find-pivots-pivots k d S B)
proof −
  have find-pivots-pivots k d S B ⊆ V
    using assms unfolding find-pivots-pivots-def by blast
  then show ?thesis
    using finite-subset[OF - finite-V] by blast
qed

lemma find-pivots-pivots-subset:
  find-pivots-pivots k d S B ⊆ S
  unfolding find-pivots-pivots-def by blast

theorem find-pivots-pivots-card-times-le-seen:
  assumes S-subset: S ⊆ V
    and anti: tree-antichain S
  shows k * card (find-pivots-pivots k d S B) ≤
    card (find-pivots-seen k d S B)
proof −
```

let *?P = find-pivots-pivots k d S B*
let *?W = find-pivots-seen k d S B*
let *?A = λu. tree-of u ∩ ?W*
have *finite-P*: *finite ?P*
  using *finite-find-pivots-pivots*[*OF S-subset*] **.**
have *finite-W*: *finite ?W*
  using *finite-find-pivots-seen*[*OF S-subset*] **.**
have *finite-A*: ⋀*u. u ∈ ?P ⟹ finite* (*?A u*)
  using *finite-W* **by** *blast*
have *disjoint-A*:
  ⋀*u v.* ⟦*u ∈ ?P*; *v ∈ ?P*; *u ≠ v*⟧ *⟹ ?A u ∩ ?A v* = {}
proof −
  fix *u v*
  assume *uP*: *u ∈ ?P* **and** *vP*: *v ∈ ?P* **and** *neq*: *u ≠ v*
  have *uS*: *u ∈ S* **and** *vS*: *v ∈ S*
    using *uP vP* **unfolding** *find-pivots-pivots-def* **by** *auto*
  have *tree-of u ∩ tree-of v* = {}
    using *tree-of-disjoint-if-antichain*[*OF anti uS vS neq*] **.**
  then show *?A u ∩ ?A v* = {}
    by *blast*
qed
have *finite-A-all*: ∀ *u*∈*?P. finite* (*?A u*)
  using *finite-A* **by** *blast*
have *disjoint-A-all*: ∀ *u*∈*?P.* ∀ *v*∈*?P. u ≠ v ⟶ ?A u ∩ ?A v* = {}
  using *disjoint-A* **by** *blast*
have *card-union-image*:
  card (⋃(*?A* ‘ *?P*)) = (∑ *u*∈*?P. card* (*?A u*))
  by (*rule card-UN-disjoint*[*OF finite-P finite-A-all disjoint-A-all*])
have *union-eq*: (⋃ *u*∈*?P. ?A u*) = ⋃(*?A* ‘ *?P*)
  by *blast*
have *card-union*:
  card (⋃ *u*∈*?P. ?A u*) = (∑ *u*∈*?P. card* (*?A u*))
  using *card-union-image union-eq* **by** *simp*
have *lower*: (∑ *u*∈*?P. k*) ≤ (∑ *u*∈*?P. card* (*?A u*))
proof (*rule sum-mono*)
  fix *u*
  assume *u ∈ ?P*
  then show *k* ≤ *card* (*?A u*)
    unfolding *find-pivots-pivots-def* **by** *simp*
qed
have *const-sum*: (∑ *u*∈*?P. k*) = *k* ∗ *card ?P*
  by (*simp add*: *mult.commute*)
have *k* ∗ *card ?P* ≤ *card* (⋃ *u*∈*?P. ?A u*)
proof −
  have *k* ∗ *card ?P* ≤ (∑ *u*∈*?P. card* (*?A u*))
    using *lower const-sum* **by** (*simp add*: *mult.commute*)
  also have ... = *card* (⋃ *u*∈*?P. ?A u*)
    using *card-union* **by** *simp*
  finally show *?thesis* **.**

```
  qed
  also have ... ≤ card ?W
  proof (rule card-mono[OF finite-W])
    show (⋃ u∈?P. ?A u) ⊆ ?W
      by blast
  qed
  finally show ?thesis .
qed

theorem find-pivots-pivots-capped-card-times-le-seen-or-overflow:
  assumes S-subset: S ⊆ V
    and anti: tree-antichain S
  shows k * card (find-pivots-pivots-capped k cap d S B) ≤
      card (find-pivots-seen-capped k cap d S B) ∨
    (find-pivots-pivots-capped k cap d S B = S ∧
      card (find-pivots-seen-capped k cap d S B) > cap)
proof (cases card (find-pivots-seen-capped k cap d S B) > cap)
  case True
  then show ?thesis
    unfolding find-pivots-pivots-capped-def by simp
next
  case False
  have cap-le: card (find-pivots-seen-capped k cap d S B) ≤ cap
    using False by simp
  have iter-eq: fp-iter-capped k cap d S S B = fp-iter k d S S B
    using fp-iter-capped-eq-fp-iter-if-final-within-cap[of k cap d S S B]
      cap-le unfolding find-pivots-seen-capped-def by simp
  have seen-eq: find-pivots-seen-capped k cap d S B = find-pivots-seen k d S B
    using iter-eq unfolding find-pivots-seen-capped-def find-pivots-seen-def by simp
  have pivots-eq:
    find-pivots-pivots-capped k cap d S B = find-pivots-pivots k d S B
    using False seen-eq
    unfolding find-pivots-pivots-capped-def find-pivots-pivots-def by simp
  have k * card (find-pivots-pivots k d S B) ≤
      card (find-pivots-seen k d S B)
    by (rule find-pivots-pivots-card-times-le-seen[OF S-subset anti])
  then show ?thesis
    unfolding seen-eq pivots-eq by blast
qed

lemma find-pivots-long-tight-path-root-pivot:
  assumes sound: sound-label d
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and uS: u ∈ S
    and complete-u: d u = dist s u
    and p-nonempty: p ≠ []
    and hd-p: hd p = u
    and distinct-p: distinct p
    and len: k ≤ length p
```

```
    and tight: successively tight-edge-step p
    and below-prefix:
      ⋀z. z ∈ set (take k p) ⟹ below-bound (dist s z) B
    and tree-prefix: set (take k p) ⊆ tree-of u
  shows u ∈ find-pivots-pivots k d S B
proof (rule find-pivots-pivot-if-k-seen-tree-vertices[OF uS distinct-p len tree-prefix])
  show set (take k p) ⊆ find-pivots-seen k d S B
  proof
    fix z
    assume z-take: z ∈ set (take k p)
    obtain i where i: i < length (take k p) take k p ! i = z
      using z-take by (auto simp: in-set-conv-nth)
    have len-take: length (take k p) = k
      using len by simp
    have i-k: i < k
      using i(1) len-take by simp
    have i-p: i < length p
      using i-k len by linarith
    have p-i: p ! i = z
      using i i-k by simp
    let ?q = take (Suc i) p
    have q-nonempty: ?q ≠ []
      using i-p by (cases p) auto
    have hd-q: hd ?q = u
      using hd-p p-nonempty by (cases p) auto
    have hd-q-S: hd ?q ∈ S
      using hd-q uS by simp
    have complete-hd-q: d (hd ?q) = dist s (hd ?q)
      using hd-q complete-u by simp
    have last-q: last ?q = z
    proof −
      have len-q: length ?q = Suc i
        using i-p by simp
      have last ?q = ?q ! i
        using len-q last-conv-nth[OF q-nonempty] by simp
      also have ... = z
        using p-i i-p by simp
      finally show ?thesis .
    qed
    have tight-q: successively tight-edge-step ?q
      using successively-take[OF tight, of Suc i] .
    have below-q: ⋀y. y ∈ set ?q ⟹ below-bound (dist s y) B
    proof −
      fix y
      assume yq: y ∈ set ?q
      have set ?q ⊆ set (take k p)
        using i-k set-take-subset-set-take[of Suc i k p] by auto
      then show below-bound (dist s y) B
        using below-prefix yq by blast
```

```
    qed
    have len-q: length (path-edges ?q) ≤ k
    proof −
      have length (path-edges ?q) = length ?q − 1
        by (rule path-edges-length)
      also have ... = i
        using i-p by simp
      also have ... ≤ k
        using i-k by simp
      finally show ?thesis .
    qed
    have last ?q ∈ find-pivots-seen k d S B
      using find-pivots-short-tight-path-complete(2)
        [OF sound S-reaches q-nonempty hd-q-S complete-hd-q tight-q below-q len-q] .
    then show z ∈ find-pivots-seen k d S B
      using last-q by simp
  qed
qed

lemma find-pivots-long-tree-suffix-root-pivot:
  assumes sound: sound-label d
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and reach-v: reachable s v
    and below-v: below-bound (dist s v) B
    and i: i < length (shortest-path-to v)
    and source-in-S: shortest-path-to v ! i ∈ S
    and source-complete: d (shortest-path-to v ! i) = dist s (shortest-path-to v ! i)
    and long: k ≤ length (drop i (shortest-path-to v))
    and tree-prefix:
      set (take k (drop i (shortest-path-to v))) ⊆
        tree-of (shortest-path-to v ! i)
  shows shortest-path-to v ! i ∈ find-pivots-pivots k d S B
proof −
  let ?p = shortest-path-to v
  let ?q = drop i ?p
  let ?u = ?p ! i
  have sp: shortest-walk s ?p v
    using shortest-path-to-shortest[OF reach-v] .
  have simple-p: simple-walk-betw s ?p v
    using sp unfolding shortest-walk-def by blast
  have distinct-p: distinct ?p
    using simple-p unfolding simple-walk-betw-def by blast
  have distinct-q: distinct ?q
    using distinct-p by simp
  have q-nonempty: ?q ≠ []
    using i by auto
  have hd-q: hd ?q = ?u
    using i by (simp add: hd-drop-conv-nth)
  have tight-q: successively tight-edge-step ?q
```

```
    using successively-drop[OF shortest-path-to-successively-tight[OF reach-v], of i] .
  have below-prefix: ⋀x. x ∈ set (take k ?q) ⟹ below-bound (dist s x) B
  proof −
    fix x
    assume x-take: x ∈ set (take k ?q)
    have xq: x ∈ set ?q
      using x-take by (meson in-set-takeD)
    have x ∈ set ?p
      using in-set-dropD[OF xq] .
    then have tree-path x v
      using reach-v unfolding tree-path-def by blast
    then have dist s x ≤ dist s v
      by (rule tree-path-dist-le)
    then show below-bound (dist s x) B
      using below-bound-le-trans[OF - below-v] by blast
  qed
  show ?thesis
    using find-pivots-long-tight-path-root-pivot
      [OF sound S-reaches source-in-S source-complete q-nonempty hd-q
        distinct-q long tight-q below-prefix tree-prefix] .
qed

corollary find-pivots-one-tight-step-complete:
  assumes sound: sound-label d
    and uS: u ∈ S
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and complete-u: d u = dist s u
    and tight: tight-edge-step u v
    and below-v: below-bound (dist s v) B
  shows find-pivots-label (Suc n) d S B v = dist s v
    and v ∈ find-pivots-seen (Suc n) d S B
  using fp-iter-one-tight-step-completion
    [OF sound uS S-reaches complete-u tight below-v, of n S]
  unfolding find-pivots-label-def find-pivots-seen-def by auto

theorem find-pivots-covers-bound-tree-complete-sources:
  assumes sound: sound-label d
    and S-complete: complete-on d S
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and v-bound: v ∈ bound-tree S B
  shows
    (find-pivots-label k d S B v = dist s v ∧
      v ∈ find-pivots-seen k d S B) ∨
     through-complete (find-pivots-label k d S B)
       (find-pivots-pivots k d S B) v
proof −
  let ?p = shortest-path-to v
  let ?d' = find-pivots-label k d S B
  let ?P = find-pivots-pivots k d S B
```

```
  have reach-v: reachable s v
    and through-S: through S v
    and below-v: below-bound (dist s v) B
    using v-bound unfolding bound-tree-def by auto
  obtain u where uS: u ∈ S and tree-uv: tree-path u v
    using through-iff-tree-path[OF reach-v, of S] through-S by blast
  then have u-in-path: u ∈ set ?p
    by (auto dest: tree-pathD)
  have sp-v: shortest-walk s ?p v
    using shortest-path-to-shortest[OF reach-v] .
  have uV: u ∈ V
  proof −
    have walk ?p
      using sp-v unfolding shortest-walk-def simple-walk-betw-def walk-betw-def by
blast
    then show ?thesis
      using u-in-path walk-set-subset by blast
  qed
  obtain i where i: i < length ?p ?p ! i = u
    using u-in-path by (auto simp: in-set-conv-nth)
  have complete-u: d u = dist s u
    using S-complete S-reaches[OF uS] uS unfolding complete-on-def by blast
  have source-in: ?p ! i ∈ S
    using i(2) uS by simp
  have source-complete: d (?p ! i) = dist s (?p ! i)
    using i(2) complete-u by simp

  show ?thesis
  proof (cases length (path-edges (drop i ?p)) ≤ k)
    case True
    have complete-v: ?d' v = dist s v
      by (rule find-pivots-completes-short-tree-suffix(1)
          [OF sound S-reaches reach-v below-v i(1) source-in source-complete True])
    have seen-v: v ∈ find-pivots-seen k d S B
      by (rule find-pivots-completes-short-tree-suffix(2)
          [OF sound S-reaches reach-v below-v i(1) source-in source-complete True])
    then show ?thesis
      using complete-v by blast
  next
    case False
    let ?q = drop i ?p
    have q-nonempty: ?q ≠ []
      using i(1) by auto
    have long: k ≤ length ?q
    proof −
      have k < length (path-edges ?q)
        using False by simp
      also have ... = length ?q − 1
        by (rule path-edges-length)
```

```
    finally show ?thesis
      by linarith
  qed
  have source-tree-prefix:
    set (take k ?q) ⊆ tree-of (?p ! i)
    using tree-prefix-of-shortest-path-suffix[OF reach-v i(1), of k] .
  have source-pivot: ?p ! i ∈ ?P
    by (rule find-pivots-long-tree-suffix-root-pivot
        [OF sound S-reaches reach-v below-v i(1) source-in source-complete long
          source-tree-prefix])
  have u-pivot: u ∈ ?P
    using source-pivot i(2) by simp
  have complete-u-after: ?d' u = dist s u
    unfolding find-pivots-label-def
    by (rule fp-iter-preserves-complete-sound[OF sound complete-u S-reaches])
  have u-complete-vertex: u ∈ complete-vertices ?d'
    using uV S-reaches[OF uS] complete-u-after
    unfolding complete-vertices-def by blast
  have through (?P ∩ complete-vertices ?d') v
    using u-pivot u-complete-vertex sp-v u-in-path unfolding through-def by blast
  then show ?thesis
    unfolding through-complete-def by blast
 qed
qed

theorem find-pivots-post-complete-sources:
 assumes sound: sound-label d
   and S-subset: S ⊆ V
   and S-complete: complete-on d S
   and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
 shows find-pivots-post d S B
   (find-pivots-label k d S B)
   (find-pivots-pivots k d S B)
   {v ∈ bound-tree S B. find-pivots-label k d S B v = dist s v}
proof −
 let ?d' = find-pivots-label k d S B
 let ?P = find-pivots-pivots k d S B
 let ?W = {v ∈ bound-tree S B. ?d' v = dist s v}
 have P-subset: ?P ⊆ S
   unfolding find-pivots-pivots-def by blast
 have W-subset: ?W ⊆ bound-tree S B
   by blast
 have complete-W: complete-on ?d' ?W
   unfolding complete-on-def by blast
 have preserve:
   ∀ v∈V. reachable s v ⟶ below-bound (dist s v) B ⟶
     d v = dist s v ⟶ ?d' v = dist s v
 proof clarify
   fix v
```

```
    assume v-complete: d v = dist s v
    show ?d' v = dist s v
      unfolding find-pivots-label-def
      by (rule fp-iter-preserves-complete-sound[OF sound v-complete S-reaches])
  qed
  have cover: ∀ v∈bound-tree S B. v ∈ ?W ∨ through-complete ?d' ?P v
  proof
    fix v
    assume v: v ∈ bound-tree S B
    have cov:
      (?d' v = dist s v ∧ v ∈ find-pivots-seen k d S B) ∨
       through-complete ?d' ?P v
      using find-pivots-covers-bound-tree-complete-sources
        [OF sound S-complete S-reaches v]
      .
    then show v ∈ ?W ∨ through-complete ?d' ?P v
      using v by blast
  qed
  show ?thesis
    using S-subset P-subset W-subset complete-W preserve cover
    unfolding find-pivots-post-def by blast
qed

theorem find-pivots-establishes-pivot-pre-concrete:
  assumes sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
  shows bmssp-pre-full
    (find-pivots-label k d S B)
    (find-pivots-pivots k d S B) B
proof −
  let ?d' = find-pivots-label k d S B
  let ?P = find-pivots-pivots k d S B
  have P-subset-S: ?P ⊆ S
    unfolding find-pivots-pivots-def by blast
  have P-subset-V: ?P ⊆ V
    using P-subset-S pre unfolding bmssp-pre-full-def by blast
  have preserve-complete: ⋀x. d x = dist s x ⟹ ?d' x = dist s x
    unfolding find-pivots-label-def
    by (rule fp-iter-preserves-complete-sound[OF sound - S-reaches])
  have cover:
    ⋀v. ⟦v ∈ V; reachable s v; below-bound (dist s v) B;
       ?d' v ≠ dist s v⟧ ⟹ through-complete ?d' ?P v
  proof −
    fix v
    assume vV: v ∈ V
      and reach-v: reachable s v
      and below-v: below-bound (dist s v) B
      and d'-incomplete: ?d' v ≠ dist s v
```

```
have d-incomplete: d v ≠ dist s v
proof
  assume d v = dist s v
  then have ?d' v = dist s v
    by (rule preserve-complete)
  with d'-incomplete show False
    by simp
qed
have through-comp: through-complete d S v
  using pre vV reach-v below-v d-incomplete
  unfolding bmssp-pre-full-def by blast
obtain u p where uS: u ∈ S and ucomp: u ∈ complete-vertices d
  and sp: shortest-walk s p v and u-in-p: u ∈ set p
  using through-comp unfolding through-complete-def through-def by blast
let ?p = shortest-path-to v
have sp-canon: shortest-walk s ?p v
  using shortest-path-to-shortest[OF reach-v] .
have p-eq: p = ?p
  using unique-shortest-walk[OF sp sp-canon] .
have u-in-path: u ∈ set ?p
  using u-in-p p-eq by simp
obtain i where i: i < length ?p ?p ! i = u
  using u-in-path by (auto simp: in-set-conv-nth)
have complete-u: d u = dist s u
  using ucomp unfolding complete-vertices-def by blast
have source-in: ?p ! i ∈ S
  using i(2) uS by simp
have source-complete: d (?p ! i) = dist s (?p ! i)
  using i(2) complete-u by simp

show through-complete ?d' ?P v
proof (cases length (path-edges (drop i ?p)) ≤ k)
  case True
  have ?d' v = dist s v
    by (rule find-pivots-completes-short-tree-suffix(1)
        [OF sound S-reaches reach-v below-v i(1) source-in source-complete True])
  with d'-incomplete have False
    by simp
  then show ?thesis
    by blast
next
  case False
  let ?q = drop i ?p
  have long: k ≤ length ?q
  proof −
    have k < length (path-edges ?q)
      using False by simp
    also have ... = length ?q − 1
      by (rule path-edges-length)
```

```
      finally show ?thesis
        by linarith
    qed
    have source-tree-prefix:
      set (take k ?q) ⊆ tree-of (?p ! i)
      using tree-prefix-of-shortest-path-suffix[OF reach-v i(1), of k] .
    have source-pivot: ?p ! i ∈ ?P
      by (rule find-pivots-long-tree-suffix-root-pivot
          [OF sound S-reaches reach-v below-v i(1) source-in source-complete long
            source-tree-prefix])
    have u-pivot: u ∈ ?P
      using source-pivot i(2) by simp
    have uV: u ∈ V and reach-u: reachable s u
      using ucomp unfolding complete-vertices-def by auto
    have complete-u-after: ?d' u = dist s u
      by (rule preserve-complete[OF complete-u])
    have u-complete-after: u ∈ complete-vertices ?d'
      using uV reach-u complete-u-after unfolding complete-vertices-def by blast
    have through (?P ∩ complete-vertices ?d') v
      using u-pivot u-complete-after sp-canon u-in-path unfolding through-def by
blast
    then show ?thesis
      unfolding through-complete-def .
  qed
 qed
 show ?thesis
  using P-subset-V cover unfolding bmssp-pre-full-def by blast
qed

theorem find-pivots-post-from-pre-concrete:
 assumes sound: sound-label d
  and pre: bmssp-pre-full d S B
  and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
 shows find-pivots-post d S B
  (find-pivots-label k d S B)
  (find-pivots-pivots k d S B)
  {v ∈ bound-tree S B. find-pivots-label k d S B v = dist s v}
proof −
 let ?d' = find-pivots-label k d S B
 let ?P = find-pivots-pivots k d S B
 let ?W = {v ∈ bound-tree S B. ?d' v = dist s v}
 have S-subset: S ⊆ V
  using pre unfolding bmssp-pre-full-def by blast
 have P-subset: ?P ⊆ S
  unfolding find-pivots-pivots-def by blast
 have W-subset: ?W ⊆ bound-tree S B
  by blast
 have complete-W: complete-on ?d' ?W
  unfolding complete-on-def by blast
```

```
  have preserve:
    ∀ v∈V. reachable s v ⟶ below-bound (dist s v) B ⟶
      d v = dist s v ⟶ ?d' v = dist s v
  proof clarify
    fix v
    assume complete-v: d v = dist s v
    show ?d' v = dist s v
      unfolding find-pivots-label-def
      by (rule fp-iter-preserves-complete-sound[OF sound complete-v S-reaches])
  qed
  have pivot-pre: bmssp-pre-full ?d' ?P B
    by (rule find-pivots-establishes-pivot-pre-concrete
      [OF sound pre S-reaches])
  have cover: ∀ v∈bound-tree S B. v ∈ ?W ∨ through-complete ?d' ?P v
  proof
    fix v
    assume v-bound: v ∈ bound-tree S B
    show v ∈ ?W ∨ through-complete ?d' ?P v
    proof (cases ?d' v = dist s v)
      case True
      then show ?thesis
        using v-bound by blast
    next
      case False
      have vV: v ∈ V and reach-v: reachable s v
        and below-v: below-bound (dist s v) B
        using v-bound unfolding bound-tree-def by auto
      have through-complete ?d' ?P v
        using pivot-pre vV reach-v below-v False
        unfolding bmssp-pre-full-def by blast
      then show ?thesis
        by blast
    qed
  qed
  show ?thesis
    using S-subset P-subset W-subset complete-W preserve cover
    unfolding find-pivots-post-def by blast
qed

theorem find-pivots-capped-no-overflow-establishes-pre:
  assumes sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and no-overflow: ¬ card (find-pivots-seen-capped k cap d S B) > cap
  shows bmssp-pre-full
    (find-pivots-label-capped k cap d S B)
    (find-pivots-pivots-capped k cap d S B) B
proof −
  have cap-le: card (find-pivots-seen-capped k cap d S B) ≤ cap
```

```
    using no-overflow by simp
  have iter-eq: fp-iter-capped k cap d S S B = fp-iter k d S S B
    using fp-iter-capped-eq-fp-iter-if-final-within-cap[of k cap d S S B]
      cap-le unfolding find-pivots-seen-capped-def by simp
  have label-eq: find-pivots-label-capped k cap d S B = find-pivots-label k d S B
   using iter-eq unfolding find-pivots-label-capped-def find-pivots-label-def by simp
  have seen-eq: find-pivots-seen-capped k cap d S B = find-pivots-seen k d S B
   using iter-eq unfolding find-pivots-seen-capped-def find-pivots-seen-def by simp
  have pivots-eq:
    find-pivots-pivots-capped k cap d S B = find-pivots-pivots k d S B
    using no-overflow seen-eq
    unfolding find-pivots-pivots-capped-def find-pivots-pivots-def by simp
  show ?thesis
    using find-pivots-establishes-pivot-pre-concrete[OF sound pre S-reaches]
    unfolding label-eq pivots-eq .
qed

theorem find-pivots-capped-establishes-pivot-pre-concrete:
  assumes sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
  shows bmssp-pre-full
    (find-pivots-label-capped k cap d S B)
    (find-pivots-pivots-capped k cap d S B) B
proof (cases card (find-pivots-seen-capped k cap d S B) > cap)
  case True
  show ?thesis
     by (rule find-pivots-capped-overflow-establishes-pre[OF sound pre S-reaches
True])
next
  case False
  show ?thesis
    by (rule find-pivots-capped-no-overflow-establishes-pre[OF sound pre S-reaches
False])
qed
```

The capped variant has two proof branches but one external contract. If the cap overflows, the paper keeps the original source set as the pivot set; the existing BMSSP precondition is therefore preserved. If it does not overflow, *card (fp-seen (fp-iter-capped ?n ?cap ?d ?F ?W ?B)) ≤ ?cap ⟹ fp-iter-capped ?n ?cap ?d ?F ?W ?B = fp-iter ?n ?d ?F ?W ?B* identifies the capped execution with the uncapped one, so the pivot-precondition proof can reuse the short-path completeness argument. The theorem above hides that case split and exposes a single *bmssp-pre-full* fact for the capped labels and *find-pivots-pivots-capped*.

```
theorem find-pivots-capped-post-from-pre-concrete:
  assumes sound: sound-label d
    and pre: bmssp-pre-full d S B
```

```
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
  shows find-pivots-post d S B
    (find-pivots-label-capped k cap d S B)
    (find-pivots-pivots-capped k cap d S B)
    {v ∈ bound-tree S B.
      find-pivots-label-capped k cap d S B v = dist s v}
proof −
  let ?d′ = find-pivots-label-capped k cap d S B
  let ?P = find-pivots-pivots-capped k cap d S B
  let ?W = {v ∈ bound-tree S B. ?d′ v = dist s v}
  have S-subset: S ⊆ V
    using pre unfolding bmssp-pre-full-def by blast
  have P-subset: ?P ⊆ S
    unfolding find-pivots-pivots-capped-def by auto
  have W-subset: ?W ⊆ bound-tree S B
    by blast
  have complete-W: complete-on ?d′ ?W
    unfolding complete-on-def by blast
  have preserve:
    ∀ v∈V. reachable s v ⟶ below-bound (dist s v) B ⟶
      d v = dist s v ⟶ ?d′ v = dist s v
  proof clarify
    fix v
    assume complete-v: d v = dist s v
    show ?d′ v = dist s v
      unfolding find-pivots-label-capped-def
      by (rule fp-iter-capped-preserves-complete-sound
        [OF sound complete-v S-reaches])
  qed
  have pivot-pre: bmssp-pre-full ?d′ ?P B
    by (rule find-pivots-capped-establishes-pivot-pre-concrete
      [OF sound pre S-reaches])
  have cover: ∀ v∈bound-tree S B. v ∈ ?W ∨ through-complete ?d′ ?P v
  proof
    fix v
    assume v-bound: v ∈ bound-tree S B
    show v ∈ ?W ∨ through-complete ?d′ ?P v
    proof (cases ?d′ v = dist s v)
      case True
      then show ?thesis
        using v-bound by blast
    next
      case False
      have vV: v ∈ V and reach-v: reachable s v
        and below-v: below-bound (dist s v) B
        using v-bound unfolding bound-tree-def by auto
      have through-complete ?d′ ?P v
        using pivot-pre vV reach-v below-v False
        unfolding bmssp-pre-full-def by blast
```

```
      then show ?thesis
        by blast
    qed
  qed
  show ?thesis
    using S-subset P-subset W-subset complete-W preserve cover
    unfolding find-pivots-post-def by blast
qed

end

end
theory BMSSP-Pull-Minimum
  imports BMSSP-Find-Pivots
begin
```

21 Pull Minimum

This theory starts replacing the abstract partition-loop contract by concrete range-splitting lemmas. In the BMSSP loop, a partition Pull returns a boundary and a lower batch of candidate sources. The recursive child call is made below that boundary. To justify that call, we need to know that every incomplete vertex below the boundary is still covered by a complete source in the pulled lower part.

The rest of the theory is range algebra. It defines the half-open range tree between two distance boundaries, proves that a monotone boundary list partitions a bounded tree into disjoint child ranges, and assembles completeness of the lower prefix and every child range into the BMSSP postcondition. These lemmas are the semantic counterpart of the operational partition loop and are later used by the concrete one-step assembly theory.

```
context unique-shortest-digraph
begin

definition split-below where
  split-below d S beta = {x ∈ S. d x < beta}

lemma split-below-label-below:
  assumes x ∈ split-below d S beta
  shows below-bound (d x) (Fin beta)
  using assms unfolding split-below-def by simp

lemma split-below-tree-antichain:
  assumes anti: tree-antichain S
  shows tree-antichain (split-below d S beta)
  by (rule tree-antichain-subset[OF anti]) (simp add: split-below-def)

lemma split-below-scaled-card-le:
```

```
  assumes S-subset: S ⊆ V
    and S-k-cap: k * card S ≤ cap
  shows k * card (split-below d S beta) ≤ cap
proof −
  have finite-S: finite S
    using finite-subset[OF S-subset finite-V] .
  have card (split-below d S beta) ≤ card S
    by (rule card-mono[OF finite-S]) (simp add: split-below-def)
  then have k * card (split-below d S beta) ≤ k * card S
    by simp
  then show ?thesis
    using S-k-cap by linarith
qed
```

The set *split-below* is the lower part produced by a Pull boundary: it keeps exactly those sources whose current labels are below *beta*. The first lemmas show that this split respects the finite bound, preserves the tree-antichain property, and cannot increase the scaled cardinality used by the FindPivots cap. These are small facts, but they are the side conditions needed before the lower recursive call can be formed.

```
definition complete-tree-cover where
  complete-tree-cover d S B ⟷ (∀ v∈bound-tree S B. through-complete d S v)

definition range-tree where
  range-tree S a B =
    {v ∈ tree-set S. a ≤ dist s v ∧ below-bound (dist s v) B}

fun nondecreasing-from where
  nondecreasing-from a [] ⟷ True
| nondecreasing-from a (b # bs) ⟷ a ≤ b ∧ nondecreasing-from b bs

fun bounds-le where
  bounds-le B [] ⟷ True
| bounds-le B (b # bs) ⟷ bound-le (Fin b) B ∧ bounds-le B bs

fun range-tree-chain where
  range-tree-chain S a [] B = range-tree S a B
| range-tree-chain S a (b # bs) B =
    range-tree S a (Fin b) ∪ range-tree-chain S b bs B

fun range-tree-chain-list where
  range-tree-chain-list S a [] B = [range-tree S a B]
| range-tree-chain-list S a (b # bs) B =
    range-tree S a (Fin b) # range-tree-chain-list S b bs B
```

The predicates *complete-tree-cover* and *range-tree* describe the two main geometric facts used by the loop. A complete tree cover says every vertex in a bounded tree is certified by a complete source. A range tree is the part of a source tree whose true distances lie between a lower boundary and an upper

bound. The functions *range-tree-chain* and *range-tree-chain-list* extend this to a monotone list of child boundaries; the list form is convenient for relating one child postcondition to one child range.

lemma *below-bound-strict-trans*:
assumes $a < b$ *below-bound* b B
shows *below-bound* a B
using *assms* **by** (*cases* B) *auto*

lemma *complete-verticesD*:
assumes $u \in$ *complete-vertices* d
shows $u \in V$ *reachable* s u d $u =$ *dist* s u
using *assms* **unfolding** *complete-vertices-def* **by** *auto*

lemma *bound-tree-complete-vertices-eq*:
assumes *cover*:
$\bigwedge u.$ $[\![u \in S$; *reachable* s u; *below-bound* (*dist* s u) B;
d $u \neq$ *dist* s $u]\!] \Longrightarrow$ *through-complete* d S u
shows *bound-tree* S $B =$ *bound-tree* ($S \cap$ *complete-vertices* d) B
proof
show *bound-tree* ($S \cap$ *complete-vertices* d) $B \subseteq$ *bound-tree* S B
using *bound-tree-mono*[*of* $S \cap$ *complete-vertices* d S B] **by** *blast*
next
show *bound-tree* S $B \subseteq$ *bound-tree* ($S \cap$ *complete-vertices* d) B
proof
fix v
assume *v-bound*: $v \in$ *bound-tree* S B
then have *vV*: $v \in V$ **and** *reach-v*: *reachable* s v
and *through-S*: *through* S v **and** *below-v*: *below-bound* (*dist* s v) B
unfolding *bound-tree-def* **by** *auto*
obtain u **where** *uS*: $u \in S$ **and** *uv*: *tree-path* u v
using *through-iff-tree-path*[*OF reach-v*, *of S*] *through-S* **by** *blast*
have *reach-u*: *reachable* s u
using *tree-path-root-reachable*[*OF uv*] **.**
have *u-below*: *below-bound* (*dist* s u) B
using *tree-path-dist-le*[*OF uv*] *below-v*
by (*cases* B) *auto*
have *through* ($S \cap$ *complete-vertices* d) v
proof (*cases* d $u =$ *dist* s u)
case *True*
have *u-complete*: $u \in$ *complete-vertices* d
using *tree-path-root-in-V*[*OF uv*] *reach-u True*
unfolding *complete-vertices-def* **by** *blast*
have *u-path*: $u \in$ *set* (*shortest-path-to* v)
using *uv* **by** (*auto dest*: *tree-pathD*)
have *sp-v*: *shortest-walk* s (*shortest-path-to* v) v
using *shortest-path-to-shortest*[*OF reach-v*] **.**
show *?thesis*
using *uS u-complete sp-v u-path* **unfolding** *through-def* **by** *blast*
next

```
    case False
    have through-complete d S u
      using cover[OF uS reach-u u-below False] .
    then have through (S ∩ complete-vertices d) u
      unfolding through-complete-def .
    then obtain q where qS: q ∈ S ∩ complete-vertices d
      and qu: tree-path q u
      using through-iff-tree-path[OF reach-u, of S ∩ complete-vertices d]
      by blast
    have qv: tree-path q v
      using tree-path-trans[OF uv qu] .
    have q-path: q ∈ set (shortest-path-to v)
      using qv by (auto dest: tree-pathD)
    have sp-v: shortest-walk s (shortest-path-to v) v
      using shortest-path-to-shortest[OF reach-v] .
    show ?thesis
      using qS sp-v q-path unfolding through-def by blast
  qed
  then show v ∈ bound-tree (S ∩ complete-vertices d) B
    using vV reach-v below-v unfolding bound-tree-def by blast
 qed
qed

lemma bmssp-pre-full-bound-tree-complete-vertices-eq:
 assumes pre: bmssp-pre-full d S B
 shows bound-tree S B = bound-tree (S ∩ complete-vertices d) B
proof (rule bound-tree-complete-vertices-eq)
 fix u
 assume uS: u ∈ S and reach-u: reachable s u
  and below-u: below-bound (dist s u) B
  and incomplete-u: d u ≠ dist s u
 have u ∈ V
  using uS pre unfolding bmssp-pre-full-def by blast
 then show through-complete d S u
  using uS reach-u below-u incomplete-u pre unfolding bmssp-pre-full-def by
blast
qed

lemma bound-tree-complete-vertices-eq-imp-cover:
 assumes eq: bound-tree S B = bound-tree (S ∩ complete-vertices d) B
 shows complete-tree-cover d S B
proof (unfold complete-tree-cover-def, intro ballI)
 fix v
 assume v ∈ bound-tree S B
 then have v ∈ bound-tree (S ∩ complete-vertices d) B
  using eq by simp
 then have through (S ∩ complete-vertices d) v
  unfolding bound-tree-def by blast
 then show through-complete d S v
```

```
    unfolding through-complete-def by simp
qed

lemma bmssp-pre-full-complete-tree-cover:
  assumes pre: bmssp-pre-full d S B
  shows complete-tree-cover d S B
  using bound-tree-complete-vertices-eq-imp-cover
    [OF bmssp-pre-full-bound-tree-complete-vertices-eq[OF pre]] .

lemma bmssp-pre-full-bound-mono:
  assumes pre: bmssp-pre-full d S B
    and le: bound-le B' B
  shows bmssp-pre-full d S B'
proof −
  have S-subset: S ⊆ V
    using pre unfolding bmssp-pre-full-def by blast
  have cover:
    ⋀v. ⟦v ∈ V; reachable s v; below-bound (dist s v) B';
       d v ≠ dist s v⟧ ⟹ through-complete d S v
  proof −
    fix v
    assume v: v ∈ V reachable s v below-bound (dist s v) B'
      d v ≠ dist s v
    have below-bound (dist s v) B
      using le v(3) by (cases B'; cases B) auto
    then show through-complete d S v
      using pre v unfolding bmssp-pre-full-def by blast
  qed
  show ?thesis
    using S-subset cover unfolding bmssp-pre-full-def by blast
qed

theorem concrete-find-pivots-complete-pivot-tree:
  assumes sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
  shows bound-tree (find-pivots-pivots k d S B) B =
    bound-tree
      (find-pivots-pivots k d S B ∩
        complete-vertices (find-pivots-label k d S B)) B
  using bmssp-pre-full-bound-tree-complete-vertices-eq
    [OF find-pivots-establishes-pivot-pre-concrete[OF sound pre S-reaches]] .

theorem concrete-find-pivots-pivot-tree-cover:
  assumes sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
  shows complete-tree-cover
    (find-pivots-label k d S B)
```

```
    (find-pivots-pivots k d S B) B
  using bmssp-pre-full-complete-tree-cover
    [OF find-pivots-establishes-pivot-pre-concrete[OF sound pre S-reaches]] .

theorem concrete-capped-find-pivots-complete-pivot-tree:
  assumes sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
  shows bound-tree (find-pivots-pivots-capped k cap d S B) B =
    bound-tree
      (find-pivots-pivots-capped k cap d S B ∩
        complete-vertices (find-pivots-label-capped k cap d S B)) B
  using bmssp-pre-full-bound-tree-complete-vertices-eq
    [OF find-pivots-capped-establishes-pivot-pre-concrete[OF sound pre S-reaches]] .

theorem concrete-capped-find-pivots-pivot-tree-cover:
  assumes sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
  shows complete-tree-cover
    (find-pivots-label-capped k cap d S B)
    (find-pivots-pivots-capped k cap d S B) B
  using bmssp-pre-full-complete-tree-cover
    [OF find-pivots-capped-establishes-pivot-pre-concrete[OF sound pre S-reaches]] .

lemma through-complete-obtain:
  assumes through-complete d S v
  obtains u p where
    u ∈ S
    u ∈ complete-vertices d
    shortest-walk s p v
    u ∈ set p
    d u = dist s u
    dist s u ≤ dist s v
proof −
  obtain u p where u: u ∈ S u ∈ complete-vertices d shortest-walk s p v u ∈ set p
    using assms unfolding through-complete-def through-def by blast
  have d u = dist s u
    using u(2) unfolding complete-vertices-def by blast
  moreover have dist s u ≤ dist s v
    using shortest-walk-prefix-dist-le u by blast
  ultimately show thesis
    using that u by blast
qed

lemma pull-minimum-incomplete-covered:
  assumes pre: bmssp-pre-full d S B
    and beta: below-bound beta B
    and vV: v ∈ V
```

```
    and reach: reachable s v
    and lt: dist s v < beta
    and incomplete: d v ≠ dist s v
  shows through-complete d (split-below d S beta) v
proof −
  have below-v-B: below-bound (dist s v) B
    using below-bound-strict-trans[OF lt beta] .
  have cover: through-complete d S v
    using pre vV reach below-v-B incomplete unfolding bmssp-pre-full-def by blast
  then obtain u p where uS: u ∈ S and ucomp: u ∈ complete-vertices d
    and p: shortest-walk s p v and up: u ∈ set p
    and du: d u = dist s u and dist-le: dist s u ≤ dist s v
    by (rule through-complete-obtain)
  have d u < beta
    using du dist-le lt by linarith
  then have u ∈ split-below d S beta
    using uS unfolding split-below-def by blast
  then have through (split-below d S beta ∩ complete-vertices d) v
    using ucomp p up unfolding through-def by blast
  then show ?thesis
    unfolding through-complete-def .
qed

lemma pull-minimum-pre-for-lower-split:
  assumes pre: bmssp-pre-full d S B
    and beta: below-bound beta B
  shows bmssp-pre-full d (split-below d S beta) (Fin beta)
proof −
  have S-subset: split-below d S beta ⊆ V
    using pre unfolding bmssp-pre-full-def split-below-def by blast
  have cover:
    ⋀v. ⟦v ∈ V; reachable s v; below-bound (dist s v) (Fin beta); d v ≠ dist s v⟧
      ⟹ through-complete d (split-below d S beta) v
    using pull-minimum-incomplete-covered[OF pre beta] by auto
  show ?thesis
    using S-subset cover unfolding bmssp-pre-full-def by blast
qed

lemma split-below-subset:
  split-below d S beta ⊆ S
  unfolding split-below-def by blast

lemma range-tree-eq-bound-diff:
  range-tree S a B = bound-tree S B − bound-tree S (Fin a)
  unfolding range-tree-def bound-tree-eq-tree-set by auto

lemma range-tree-subset-bound-tree:
  range-tree S a B ⊆ bound-tree S B
  unfolding range-tree-eq-bound-diff by blast
```

lemma *finite-bound-tree* [*simp*]:
finite (*bound-tree S B*)
proof −
have *bound-tree S B* ⊆ *V*
unfolding *bound-tree-def* **by** *blast*
then show *?thesis*
using *finite-subset*[*OF - finite-V*] **by** *blast*
qed

lemma *finite-range-tree* [*simp*]:
finite (*range-tree S a B*)
proof −
have *range-tree S a B* ⊆ *V*
unfolding *range-tree-def tree-set-def* **by** *blast*
then show *?thesis*
using *finite-subset*[*OF - finite-V*] **by** *blast*
qed

lemma *finite-range-tree-chain* [*simp*]:
finite (*range-tree-chain S a bs B*)
by (*induction bs arbitrary*: *a*) *simp-all*

lemma *finite-range-tree-chain-list-sets* [*simp*]:
assumes *X* ∈ *set* (*range-tree-chain-list S a bs B*)
shows *finite X*
using *assms* **by** (*induction bs arbitrary*: *a*) *auto*

lemma *Union-range-tree-chain-list*:
⋃(*set* (*range-tree-chain-list S a bs B*)) = *range-tree-chain S a bs B*
by (*induction bs arbitrary*: *a*) *auto*

lemma *range-tree-chain-lower-bound*:
assumes *mono*: *nondecreasing-from a bs*
and *x*: *x* ∈ *range-tree-chain S a bs B*
shows *a* ≤ *dist s x*
using *assms*
proof (*induction bs arbitrary*: *a*)
case *Nil*
then show *?case*
by (*simp add*: *range-tree-def*)
next
case (*Cons b bs*)
then consider *x* ∈ *range-tree S a* (*Fin b*) |
x ∈ *range-tree-chain S b bs B*
by *auto*
then show *?case*
proof *cases*
case *1*

```
    then show ?thesis
      unfolding range-tree-def by blast
  next
    case 2
    have b ≤ dist s x
      using Cons.IH[of b] Cons.prems 2 by simp
    moreover have a ≤ b
      using Cons.prems by simp
    ultimately show ?thesis
      by linarith
  qed
qed

lemma range-tree-disjoint-lower:
  bound-tree S (Fin a) ∩ range-tree S a B = {}
  unfolding range-tree-eq-bound-diff by blast

lemma bound-tree-disjoint-range-tree-chain:
  assumes mono: nondecreasing-from a bs
  shows bound-tree S (Fin a) ∩ range-tree-chain S a bs B = {}
proof
  show bound-tree S (Fin a) ∩ range-tree-chain S a bs B ⊆ {}
  proof
    fix x
    assume x: x ∈ bound-tree S (Fin a) ∩ range-tree-chain S a bs B
    then have dist s x < a
      unfolding bound-tree-def by auto
    moreover have a ≤ dist s x
      using range-tree-chain-lower-bound[OF mono] x by blast
    ultimately show x ∈ {}
      by simp
  qed
qed simp

lemma range-tree-disjoint-range-tree-chain-tail:
  assumes a-le-b: a ≤ b
    and mono-tail: nondecreasing-from b bs
  shows range-tree S a (Fin b) ∩ range-tree-chain S b bs B = {}
proof
  show range-tree S a (Fin b) ∩ range-tree-chain S b bs B ⊆ {}
  proof
    fix x
    assume x: x ∈ range-tree S a (Fin b) ∩ range-tree-chain S b bs B
    then have dist s x < b
      unfolding range-tree-def by auto
    moreover have b ≤ dist s x
      using range-tree-chain-lower-bound[OF mono-tail] x by blast
    ultimately show x ∈ {}
      by simp
```

```
  qed
qed simp

lemma card-range-tree-chain-eq-sum-list:
  assumes mono: nondecreasing-from a bs
  shows card (range-tree-chain S a bs B) =
    sum-list (map card (range-tree-chain-list S a bs B))
  using mono
proof (induction bs arbitrary: a)
  case Nil
  then show ?case
    by simp
next
  case (Cons b bs)
  have a-le-b: a ≤ b
    using Cons.prems by simp
  have mono-tail: nondecreasing-from b bs
    using Cons.prems by simp
  have disjoint:
    range-tree S a (Fin b) ∩ range-tree-chain S b bs B = {}
    using range-tree-disjoint-range-tree-chain-tail[OF a-le-b mono-tail] .
  have card-union:
    card (range-tree S a (Fin b) ∪ range-tree-chain S b bs B) =
      card (range-tree S a (Fin b)) + card (range-tree-chain S b bs B)
    by (rule card-Un-disjoint) (simp-all add: disjoint)
  show ?case
    using card-union Cons.IH[OF mono-tail] by simp
qed

lemma sum-list-map-mult-left:
  fixes c :: nat
    and f :: 'a ⇒ nat
  shows sum-list (map (λx. c * f x) xs) = c * sum-list (map f xs)
  by (induction xs) (simp-all add: algebra-simps)

lemma sum-list-map-card-mult-left:
  fixes C :: nat
    and Xs :: 'v set list
  shows sum-list (map (λX. C * card X) Xs) = C * sum-list (map card Xs)
  by (induction Xs) (simp-all add: algebra-simps)

lemma sum-list-le-if-list-all2:
  assumes list-all2 (λc X. c ≤ C * card X) costs Xs
  shows sum-list costs ≤ sum-list (map (λX. C * card X) Xs)
  using assms
proof (induction rule: list-all2-induct)
  case Nil
  then show ?case by simp
next
```

```
  case (Cons c X costs Xs)
  then show ?case by simp
qed

lemma child-costs-le-range-tree-chain-card:
  assumes mono: nondecreasing-from a bs
    and costs: list-all2 (λc X. c ≤ C * card X) costs
      (range-tree-chain-list S a bs B)
  shows sum-list costs ≤ C * card (range-tree-chain S a bs B)
proof −
  have sum-list costs ≤
    sum-list (map (λX. C * card X) (range-tree-chain-list S a bs B))
    using sum-list-le-if-list-all2[OF costs] .
  also have ... =
    C * sum-list (map card (range-tree-chain-list S a bs B))
    by (rule sum-list-map-card-mult-left)
  also have ... = C * card (range-tree-chain S a bs B)
    using card-range-tree-chain-eq-sum-list[OF mono, of S B] by simp
  finally show ?thesis .
qed

lemma range-tree-adjacent-disjoint:
  range-tree S a (Fin b) ∩ range-tree S b B = {}
  unfolding range-tree-def by auto

lemma range-tree-adjacent-union:
  assumes a ≤ b
    and bound-le (Fin b) B
  shows range-tree S a (Fin b) ∪ range-tree S b B = range-tree S a B
  using assms unfolding range-tree-def by (cases B) auto

lemma bound-tree-split-at:
  assumes bound-le (Fin a) B
  shows bound-tree S B = bound-tree S (Fin a) ∪ range-tree S a B
  using assms unfolding range-tree-def bound-tree-eq-tree-set
  by (cases B) auto

lemma bound-tree-adjacent-union:
  assumes a ≤ b
    and bound-le (Fin b) B
  shows bound-tree S (Fin a) ∪ range-tree S a (Fin b) ∪ range-tree S b B =
    bound-tree S B
proof −
  have union: range-tree S a (Fin b) ∪ range-tree S b B = range-tree S a B
    using range-tree-adjacent-union[OF assms] .
  have le-a-B: bound-le (Fin a) B
    using assms by (cases B) auto
  show ?thesis
    using union bound-tree-split-at[OF le-a-B, of S]
```

by (*cases B*) *auto*
qed

lemma *complete-on-Un*:
assumes *complete-on d A complete-on d B*
shows *complete-on d* (*A* ∪ *B*)
using *assms* **unfolding** *complete-on-def* **by** *blast*

lemma *complete-on-Union-list*:
assumes ⋀*X*. *X* ∈ *set Xs* ⟹ *complete-on d X*
shows *complete-on d* (⋃(*set Xs*))
using *assms* **unfolding** *complete-on-def* **by** *blast*

lemma *complete-on-bound-tree-split*:
assumes *bound-le* (*Fin a*) *B*
and *lower*: *complete-on d* (*bound-tree S* (*Fin a*))
and *upper*: *complete-on d* (*range-tree S a B*)
shows *complete-on d* (*bound-tree S B*)
proof −
have *complete-on d* (*bound-tree S* (*Fin a*) ∪ *range-tree S a B*)
using *complete-on-Un*[*OF lower upper*] **.**
then show *?thesis*
using *bound-tree-split-at*[*OF assms*(*1*), *of S*] **by** *simp*
qed

lemma *bound-tree-range-chain-union*:
assumes *mono*: *nondecreasing-from a bs*
and *bounds*: *bounds-le B* (*a* # *bs*)
shows *bound-tree S* (*Fin a*) ∪ *range-tree-chain S a bs B* = *bound-tree S B*
using *assms*
proof (*induction bs arbitrary*: *a*)
case *Nil*
then show *?case*
using *bound-tree-split-at*[*of a B S*] **by** *simp*
next
case (*Cons b bs*)
have *a-le-b*: *a* ≤ *b*
using *Cons.prems*(*1*) **by** *simp*
have *mono-tail*: *nondecreasing-from b bs*
using *Cons.prems*(*1*) **by** *simp*
have *b-le-B*: *bound-le* (*Fin b*) *B*
using *Cons.prems*(*2*) **by** *simp*
have *bounds-tail*: *bounds-le B* (*b* # *bs*)
using *Cons.prems*(*2*) **by** *simp*
have *split-ab*:
bound-tree S (*Fin b*) = *bound-tree S* (*Fin a*) ∪ *range-tree S a* (*Fin b*)
using *bound-tree-split-at*[*of a Fin b S*] *a-le-b* **by** *simp*
have *tail*:
bound-tree S (*Fin b*) ∪ *range-tree-chain S b bs B* = *bound-tree S B*

```
    using Cons.IH[OF mono-tail bounds-tail] .
  show ?case
    using split-ab tail by auto
qed

lemma card-bound-tree-range-chain-union:
  assumes mono: nondecreasing-from a bs
    and bounds: bounds-le B (a # bs)
  shows card (bound-tree S B) =
    card (bound-tree S (Fin a)) + card (range-tree-chain S a bs B)
proof −
  have union: bound-tree S (Fin a) ∪ range-tree-chain S a bs B = bound-tree S B
    using bound-tree-range-chain-union[OF mono bounds, of S] .
  have disjoint: bound-tree S (Fin a) ∩ range-tree-chain S a bs B = {}
    using bound-tree-disjoint-range-tree-chain[OF mono, of S] .
  have card (bound-tree S (Fin a) ∪ range-tree-chain S a bs B) =
    card (bound-tree S (Fin a)) + card (range-tree-chain S a bs B)
    by (rule card-Un-disjoint) (simp-all add: disjoint)
  then show ?thesis
    using union by simp
qed

lemma complete-on-range-chain-union:
  assumes mono: nondecreasing-from a bs
    and bounds: bounds-le B (a # bs)
    and lower: complete-on d (bound-tree S (Fin a))
    and chain: complete-on d (range-tree-chain S a bs B)
  shows complete-on d (bound-tree S B)
proof −
  have complete-on d (bound-tree S (Fin a) ∪ range-tree-chain S a bs B)
    using complete-on-Un[OF lower chain] .
  then show ?thesis
    using bound-tree-range-chain-union[OF mono bounds, of S] by simp
qed

lemma complete-on-range-tree-chain-list:
  assumes ⋀X. X ∈ set (range-tree-chain-list S a bs B) ⟹ complete-on d X
  shows complete-on d (range-tree-chain S a bs B)
proof −
  have complete-on d (⋃(set (range-tree-chain-list S a bs B)))
    using complete-on-Union-list[OF assms] .
  then show ?thesis
    using Union-range-tree-chain-list[of S a bs B] by simp
qed

theorem partition-loop-trace-assembly-post:
  assumes le: bound-le B' B
    and mono: nondecreasing-from a bs
    and bounds: bounds-le B' (a # bs)
```

```
    and lower: complete-on d' (bound-tree S (Fin a))
    and children: list-all2 (λU X. U = X ∧ complete-on d' U) Us
      (range-tree-chain-list S a bs B')
    and U-def: U = bound-tree S (Fin a) ∪ ⋃(set Us)
  shows bmssp-post-full d S B d' B' U
proof −
  have Union-eq:
    ⋃(set Us) = ⋃(set (range-tree-chain-list S a bs B'))
    using children by (induction rule: list-all2-induct) auto
  have children-complete: ⋀X. X ∈ set Us ⟹ complete-on d' X
    using children by (induction rule: list-all2-induct) auto
  have chain-complete: complete-on d' (range-tree-chain S a bs B')
  proof −
    have complete-on d' (⋃(set Us))
      by (rule complete-on-Union-list[OF children-complete])
    then show ?thesis
      using Union-eq Union-range-tree-chain-list[of S a bs B'] by simp
  qed
  have U-tree: U = bound-tree S B'
    using U-def Union-eq Union-range-tree-chain-list[of S a bs B']
      bound-tree-range-chain-union[OF mono bounds, of S]
    by simp
  have complete-on d' U
    unfolding U-def
    using complete-on-Un[OF lower]
      complete-on-Union-list[OF children-complete] by blast
  then show ?thesis
    using le U-tree unfolding bmssp-post-full-def by blast
qed
```

The range-chain lemmas culminate in ⟦*bound-le ?B' ?B*; *nondecreasing-from ?a ?bs*; *bounds-le ?B' (?a # ?bs)*; *complete-on ?d' (bound-tree ?S (Fin ?a))*; *list-all2 (λU X. U = X ∧ complete-on ?d' U) ?Us (range-tree-chain-list ?S ?a ?bs ?B')*; *?U = bound-tree ?S (Fin ?a) ∪ ⋃ (set ?Us)*⟧ ⟹ *bmssp-post-full ?d ?S ?B ?d' ?B' ?U*. The hypotheses are exactly the certificate carried by the concrete partition trace: a returned bound below the input bound, nondecreasing child boundaries, child bounds below the returned bound, completeness of the lower prefix, and one completed set for each range in *range-tree-chain-list*. The theorem proves that their union is precisely *bound-tree* for the returned bound and that the final labels are complete there.

```
lemma pull-minimum-bound-tree-lower-split:
  assumes cover: complete-tree-cover d S (Fin beta)
    and beta'-le: beta' ≤ beta
  shows bound-tree S (Fin beta') = bound-tree (split-below d S beta) (Fin beta')
proof
  show bound-tree S (Fin beta') ⊆ bound-tree (split-below d S beta) (Fin beta')
  proof
```

```
    fix v
    assume v: v ∈ bound-tree S (Fin beta')
    then have vV: v ∈ V and reach-v: reachable s v
      and through-S: through S v and lt-beta': dist s v < beta'
      unfolding bound-tree-def by auto
    have v-beta: v ∈ bound-tree S (Fin beta)
      using v beta'-le unfolding bound-tree-def by auto
    then have through-complete d S v
      using cover unfolding complete-tree-cover-def by blast
    then obtain u p where uS: u ∈ S and ucomp: u ∈ complete-vertices d
      and p: shortest-walk s p v and up: u ∈ set p
      and du: d u = dist s u and dist-le: dist s u ≤ dist s v
      by (rule through-complete-obtain)
    have d u < beta
      using du dist-le lt-beta' beta'-le by linarith
    then have u ∈ split-below d S beta
      using uS unfolding split-below-def by blast
    then have through (split-below d S beta) v
      using p up unfolding through-def by blast
    then show v ∈ bound-tree (split-below d S beta) (Fin beta')
      using vV reach-v lt-beta' unfolding bound-tree-def by auto
  qed
next
  show bound-tree (split-below d S beta) (Fin beta') ⊆ bound-tree S (Fin beta')
    using bound-tree-mono[OF split-below-subset[of d S beta], of Fin beta'] by blast
qed

lemma pull-recursive-post-lifts-bound-tree:
  assumes cover: complete-tree-cover d S (Fin beta)
    and post: bmssp-post-full d (split-below d S beta) (Fin beta) d' B' U
  shows U = bound-tree S B'
proof −
  have le: bound-le B' (Fin beta)
    and U: U = bound-tree (split-below d S beta) B'
    using post unfolding bmssp-post-full-def by auto
  show ?thesis
  proof (cases B')
    case Infinity
    then have False
      using le by simp
    then show ?thesis
      by blast
  next
    case (Fin beta')
    then have beta'-le: beta' ≤ beta
      using le by simp
    have bound-tree S (Fin beta') =
      bound-tree (split-below d S beta) (Fin beta')
      using pull-minimum-bound-tree-lower-split[OF cover beta'-le] .
```

```
    then show ?thesis
      using U Fin by simp
  qed
qed

theorem concrete-find-pivots-pull-pre:
  assumes sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and beta: below-bound beta B
  shows bmssp-pre-full
    (find-pivots-label k d S B)
    (split-below (find-pivots-label k d S B)
      (find-pivots-pivots k d S B) beta)
    (Fin beta)
proof −
  have pivot-pre:
    bmssp-pre-full
      (find-pivots-label k d S B)
      (find-pivots-pivots k d S B) B
    using find-pivots-establishes-pivot-pre-concrete[OF sound pre S-reaches] .
  show ?thesis
    using pull-minimum-pre-for-lower-split[OF pivot-pre beta] .
qed

theorem concrete-capped-find-pivots-pull-pre:
  assumes sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and beta: below-bound beta B
  shows bmssp-pre-full
    (find-pivots-label-capped k cap d S B)
    (split-below (find-pivots-label-capped k cap d S B)
      (find-pivots-pivots-capped k cap d S B) beta)
    (Fin beta)
proof −
  have pivot-pre:
    bmssp-pre-full
      (find-pivots-label-capped k cap d S B)
      (find-pivots-pivots-capped k cap d S B) B
    using find-pivots-capped-establishes-pivot-pre-concrete[OF sound pre S-reaches] .
  show ?thesis
    using pull-minimum-pre-for-lower-split[OF pivot-pre beta] .
qed

theorem concrete-find-pivots-pull-bound-tree-lower:
  assumes sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
```

```
    and beta: below-bound beta B
    and beta'-le: beta' ≤ beta
  shows bound-tree (find-pivots-pivots k d S B) (Fin beta') =
    bound-tree
      (split-below (find-pivots-label k d S B)
        (find-pivots-pivots k d S B) beta)
      (Fin beta')
proof −
  let ?d' = find-pivots-label k d S B
  let ?P = find-pivots-pivots k d S B
  have pivot-pre-B: bmssp-pre-full ?d' ?P B
    using find-pivots-establishes-pivot-pre-concrete[OF sound pre S-reaches] .
  have pivot-pre-beta: bmssp-pre-full ?d' ?P (Fin beta)
    using bmssp-pre-full-bound-mono[OF pivot-pre-B] beta
    by (cases B) auto
  have cover-beta: complete-tree-cover ?d' ?P (Fin beta)
    using bmssp-pre-full-complete-tree-cover[OF pivot-pre-beta] .
  show ?thesis
    using pull-minimum-bound-tree-lower-split[OF cover-beta beta'-le] .
qed

theorem concrete-capped-find-pivots-pull-bound-tree-lower:
  assumes sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and beta: below-bound beta B
    and beta'-le: beta' ≤ beta
  shows bound-tree (find-pivots-pivots-capped k cap d S B) (Fin beta') =
    bound-tree
      (split-below (find-pivots-label-capped k cap d S B)
        (find-pivots-pivots-capped k cap d S B) beta)
      (Fin beta')
proof −
  let ?d' = find-pivots-label-capped k cap d S B
  let ?P = find-pivots-pivots-capped k cap d S B
  have pivot-pre-B: bmssp-pre-full ?d' ?P B
    using find-pivots-capped-establishes-pivot-pre-concrete[OF sound pre S-reaches] .
  have pivot-pre-beta: bmssp-pre-full ?d' ?P (Fin beta)
    using bmssp-pre-full-bound-mono[OF pivot-pre-B] beta
    by (cases B) auto
  have cover-beta: complete-tree-cover ?d' ?P (Fin beta)
    using bmssp-pre-full-complete-tree-cover[OF pivot-pre-beta] .
  show ?thesis
    using pull-minimum-bound-tree-lower-split[OF cover-beta beta'-le] .
qed

theorem concrete-find-pivots-pull-recursive-post-lifts:
  assumes sound: sound-label d
    and pre: bmssp-pre-full d S B
```

```
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and beta: below-bound beta B
    and post: bmssp-post-full
      (find-pivots-label k d S B)
      (split-below (find-pivots-label k d S B)
        (find-pivots-pivots k d S B) beta)
      (Fin beta) d'' B' U
  shows U = bound-tree (find-pivots-pivots k d S B) B'
proof −
  let ?d' = find-pivots-label k d S B
  let ?P = find-pivots-pivots k d S B
  have pivot-pre-B: bmssp-pre-full ?d' ?P B
    using find-pivots-establishes-pivot-pre-concrete[OF sound pre S-reaches] .
  have pivot-pre-beta: bmssp-pre-full ?d' ?P (Fin beta)
    using bmssp-pre-full-bound-mono[OF pivot-pre-B] beta
    by (cases B) auto
  have cover-beta: complete-tree-cover ?d' ?P (Fin beta)
    using bmssp-pre-full-complete-tree-cover[OF pivot-pre-beta] .
  show ?thesis
    using pull-recursive-post-lifts-bound-tree[OF cover-beta post] .
qed

theorem concrete-capped-find-pivots-pull-recursive-post-lifts:
  assumes sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and beta: below-bound beta B
    and post: bmssp-post-full
      (find-pivots-label-capped k cap d S B)
      (split-below (find-pivots-label-capped k cap d S B)
        (find-pivots-pivots-capped k cap d S B) beta)
      (Fin beta) d'' B' U
  shows U = bound-tree (find-pivots-pivots-capped k cap d S B) B'
proof −
  let ?d' = find-pivots-label-capped k cap d S B
  let ?P = find-pivots-pivots-capped k cap d S B
  have pivot-pre-B: bmssp-pre-full ?d' ?P B
    using find-pivots-capped-establishes-pivot-pre-concrete[OF sound pre S-reaches] .
  have pivot-pre-beta: bmssp-pre-full ?d' ?P (Fin beta)
    using bmssp-pre-full-bound-mono[OF pivot-pre-B] beta
    by (cases B) auto
  have cover-beta: complete-tree-cover ?d' ?P (Fin beta)
    using bmssp-pre-full-complete-tree-cover[OF pivot-pre-beta] .
  show ?thesis
    using pull-recursive-post-lifts-bound-tree[OF cover-beta post] .
qed
```

The Pull-minimum lemmas connect the range algebra to FindPivots. Once FindPivots establishes a pivot precondition, *split-below* can be used as the source set for a recursive call below a Pull boundary. If that recursive call

returns a BMSSP postcondition for the split set, the lifting lemmas identify its output tree with the corresponding pivot tree. Both capped and uncapped FindPivots variants have parallel theorems because later operational relations use the capped executable search.

```
theorem concrete-find-pivots-final-tree-assembly:
  assumes sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and loop: bmssp-post-full
      (find-pivots-label k d S B)
      (find-pivots-pivots k d S B) B d-loop B′ U-loop
  shows U-loop ∪
      {v ∈ bound-tree S B′. find-pivots-label k d S B v = dist s v} =
    bound-tree S B′
proof −
  let ?d′ = find-pivots-label k d S B
  let ?P = find-pivots-pivots k d S B
  let ?W = {v ∈ bound-tree S B′. ?d′ v = dist s v}
  have loop-le: bound-le B′ B and U-loop: U-loop = bound-tree ?P B′
    using loop unfolding bmssp-post-full-def by auto
  have P-subset-S: ?P ⊆ S
    unfolding find-pivots-pivots-def by blast
  have pivot-pre: bmssp-pre-full ?d′ ?P B
    using find-pivots-establishes-pivot-pre-concrete[OF sound pre S-reaches] .
  show ?thesis
  proof
    show U-loop ∪ ?W ⊆ bound-tree S B′
    proof
      fix v
      assume v ∈ U-loop ∪ ?W
      then show v ∈ bound-tree S B′
      proof
        assume v ∈ U-loop
        then have v ∈ bound-tree ?P B′
          using U-loop by simp
        then show ?thesis
          using bound-tree-mono[OF P-subset-S, of B′] by blast
      next
        assume v ∈ ?W
        then show ?thesis
          by blast
      qed
    qed
  next
    show bound-tree S B′ ⊆ U-loop ∪ ?W
    proof
      fix v
      assume v-bound: v ∈ bound-tree S B′
      then have vV: v ∈ V and reach-v: reachable s v
```

```
      and below-B': below-bound (dist s v) B'
      unfolding bound-tree-def by auto
    show v ∈ U-loop ∪ ?W
    proof (cases ?d' v = dist s v)
      case True
      then show ?thesis
        using v-bound by blast
    next
      case False
      have below-B: below-bound (dist s v) B
        using loop-le below-B' by (cases B'; cases B) auto
      have through-complete ?d' ?P v
        using pivot-pre vV reach-v below-B False
        unfolding bmssp-pre-full-def by blast
      then have through ?P v
        by (rule through-complete-imp-through)
      then have v ∈ bound-tree ?P B'
        using vV reach-v below-B' unfolding bound-tree-def by blast
      then show ?thesis
        using U-loop by blast
    qed
  qed
 qed
qed

theorem concrete-capped-find-pivots-final-tree-assembly:
  assumes sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and loop: bmssp-post-full
      (find-pivots-label-capped k cap d S B)
      (find-pivots-pivots-capped k cap d S B) B d-loop B' U-loop
  shows U-loop ∪
      {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v} =
    bound-tree S B'
proof −
  let ?d' = find-pivots-label-capped k cap d S B
  let ?P = find-pivots-pivots-capped k cap d S B
  let ?W = {v ∈ bound-tree S B'. ?d' v = dist s v}
  have loop-le: bound-le B' B and U-loop: U-loop = bound-tree ?P B'
    using loop unfolding bmssp-post-full-def by auto
  have pivot-pre: bmssp-pre-full ?d' ?P B
    using find-pivots-capped-establishes-pivot-pre-concrete[OF sound pre S-reaches] .
  have P-subset-V: ?P ⊆ V
    using pivot-pre unfolding bmssp-pre-full-def by blast
  have P-through-S:
    ⋀v. v ∈ bound-tree ?P B' ⟹ v ∈ bound-tree S B'
  proof −
    fix v
```

```
  assume vP: v ∈ bound-tree ?P B'
  then have vV: v ∈ V and reach-v: reachable s v
    and through-P: through ?P v and below-v: below-bound (dist s v) B'
    unfolding bound-tree-def by auto
  have through S v
  proof −
    obtain u p where uP: u ∈ ?P and sp: shortest-walk s p v and up: u ∈ set p
      using through-P unfolding through-def by blast
    have u ∈ S
    proof (cases card (find-pivots-seen-capped k cap d S B) > cap)
      case True
      then show ?thesis
        using uP unfolding find-pivots-pivots-capped-def by simp
    next
      case False
      then show ?thesis
        using uP unfolding find-pivots-pivots-capped-def by auto
    qed
    then show ?thesis
      using sp up unfolding through-def by blast
  qed
  then show v ∈ bound-tree S B'
    using vV reach-v below-v unfolding bound-tree-def by blast
qed
show ?thesis
proof
  show U-loop ∪ ?W ⊆ bound-tree S B'
    using U-loop P-through-S by blast
next
  show bound-tree S B' ⊆ U-loop ∪ ?W
  proof
    fix v
    assume v-bound: v ∈ bound-tree S B'
    then have vV: v ∈ V and reach-v: reachable s v
      and below-B': below-bound (dist s v) B'
      unfolding bound-tree-def by auto
    show v ∈ U-loop ∪ ?W
    proof (cases ?d' v = dist s v)
      case True
      then show ?thesis
        using v-bound by blast
    next
      case False
      have below-B: below-bound (dist s v) B
        using loop-le below-B' by (cases B'; cases B) auto
      have through-complete ?d' ?P v
        using pivot-pre vV reach-v below-B False
        unfolding bmssp-pre-full-def by blast
      then have through ?P v
```

```
      by (rule through-complete-imp-through)
    then have v ∈ bound-tree ?P B'
      using vV reach-v below-B' unfolding bound-tree-def by blast
    then show ?thesis
      using U-loop by blast
   qed
  qed
 qed
qed

theorem concrete-find-pivots-final-assembly-post:
 assumes sound: sound-label d
  and pre: bmssp-pre-full d S B
  and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
  and loop: bmssp-post-full
    (find-pivots-label k d S B)
    (find-pivots-pivots k d S B) B d-final B' U-loop
  and compW: complete-on d-final
    {v ∈ bound-tree S B'. find-pivots-label k d S B v = dist s v}
 defines U ≡ U-loop ∪
    {v ∈ bound-tree S B'. find-pivots-label k d S B v = dist s v}
 shows bmssp-post-full d S B d-final B' U
proof −
 let ?W = {v ∈ bound-tree S B'. find-pivots-label k d S B v = dist s v}
 have le: bound-le B' B
  using loop unfolding bmssp-post-full-def by blast
 have U-eq: U = bound-tree S B'
  using concrete-find-pivots-final-tree-assembly[OF sound pre S-reaches loop]
  unfolding U-def by simp
 have loop-complete: complete-on d-final U-loop
  using loop unfolding bmssp-post-full-def by blast
 have complete-on d-final U
  unfolding U-def using complete-on-Un[OF loop-complete compW] .
 then show ?thesis
  using le U-eq unfolding bmssp-post-full-def by blast
qed

theorem concrete-capped-find-pivots-final-assembly-post:
 assumes sound: sound-label d
  and pre: bmssp-pre-full d S B
  and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
  and loop: bmssp-post-full
    (find-pivots-label-capped k cap d S B)
    (find-pivots-pivots-capped k cap d S B) B d-final B' U-loop
  and compW: complete-on d-final
    {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v}
 defines U ≡ U-loop ∪
    {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v}
 shows bmssp-post-full d S B d-final B' U
```

proof −
let *?W* = {*v* ∈ *bound-tree S B′*. *find-pivots-label-capped k cap d S B v* = *dist s v*}
have *le*: *bound-le B′ B*
using *loop* **unfolding** *bmssp-post-full-def* **by** *blast*
have *U-eq*: *U* = *bound-tree S B′*
using *concrete-capped-find-pivots-final-tree-assembly*[*OF sound pre S-reaches loop*]
unfolding *U-def* **by** *simp*
have *loop-complete*: *complete-on d-final U-loop*
using *loop* **unfolding** *bmssp-post-full-def* **by** *blast*
have *complete-on d-final U*
unfolding *U-def* **using** *complete-on-Un*[*OF loop-complete compW*] **.**
then show *?thesis*
using *le U-eq* **unfolding** *bmssp-post-full-def* **by** *blast*
qed

The final assembly theorems are the facts used by *concrete-capped-bmssp-step*. The loop solves the pivot tree, while FindPivots may have completed additional vertices in the caller's original tree. The assembly theorem proves that the union of those two regions is the caller's returned bounded tree. The postcondition theorems then add completeness of the explicitly completed FindPivots region and conclude *bmssp-post-full* for the whole non-base step.

end

end
theory *BMSSP-Partition-Pull-Bridge*
imports *BMSSP-Pull-Minimum BMSSP-Partition-Interface*
begin

22 Partition Pull to BMSSP Pull Minimum

The partition interface is graph-independent: it knows about keys and labels, but not about shortest-path trees. BMSSP, on the other hand, phrases a child recursive call as a lower split of the current source set by a tentative distance bound. This small bridge theory identifies those two views.

The conversion is simple but important. A label function d and a source set S induce a partition view whose keys are exactly S and whose values are the labels d. A pull from that view is therefore a pull by tentative distance. The theorems below prove that the pulled key set is exactly *split-below* at the returned bound, and that the BMSSP precondition descends from the parent problem to the child problem selected by the pull.

This file is used twice in the development. The sorted reference pull uses it to justify the executable specification path. The operational and bucketed theories use the abstract *pull-separates* statement, so they can reason about BMSSP recursion without mentioning the representation of the partition data structure.

context *unique-shortest-digraph*
begin

definition *label-partition-view* **where**
label-partition-view d S = (⦇*keys-of* = *S*, *value-of* = *d*⦈)

lemma *label-partition-view-keys* [*simp*]:
keys-of (*label-partition-view d S*) = *S*
unfolding *label-partition-view-def* **by** *simp*

lemma *label-partition-view-value* [*simp*]:
value-of (*label-partition-view d S*) = *d*
unfolding *label-partition-view-def* **by** *simp*

The first two theorems are the semantic translation. For the sorted reference operation, *sorted-pull-set-eq-split-below* unfolds the sorted pull and the label partition view to obtain the mathematical lower split. For any implementation satisfying the abstract separator predicate, *pull-separates-label-set-eq-split-below* gives the same conclusion from *pull-separates*. The latter is the theorem used by the bucketed implementation after it proves its pull operation realises the separator contract.

theorem *sorted-pull-set-eq-split-below*:
fixes *M* :: *nat*
and *Bmax* :: *real*
assumes *finite-S*: *finite S*
and *upper*: ⋀*u*. *u* ∈ *S* ⟹ *d u* < *Bmax*
and *D-def*: *D* = *label-partition-view d S*
and *beta-def*: *beta* = *sorted-pull-bound M Bmax D*
shows *sorted-pull-set M D* = *split-below d S beta*
proof −
have *sorted-pull-set M D* =
{*u* ∈ *keys-of D*. *value-of D u* < *sorted-pull-bound M Bmax D*}
using *sorted-pull-exact-split*[*of D Bmax M*] *finite-S upper*
unfolding *D-def* **by** *simp*
then show *?thesis*
unfolding *D-def beta-def split-below-def* **by** *simp*
qed

theorem *pull-separates-label-set-eq-split-below*:
fixes *Bmax* :: *real*
assumes *pull*: *pull-separates* (*label-partition-view d S*) *M Bmax S′ beta D′*
and *upper*: ⋀*u*. *u* ∈ *S* ⟹ *d u* < *Bmax*
shows *S′* = *split-below d S beta*
proof −
have *S′* =
{*u* ∈ *keys-of* (*label-partition-view d S*).
value-of (*label-partition-view d S*) *u* < *beta*}
by (*rule pull-separates-exact-split*[*OF pull*]) (*simp add*: *upper*)
then show *?thesis*

```
    unfolding split-below-def by simp
qed

theorem sorted-pull-establishes-lower-pre:
  fixes M :: nat
    and Bmax :: real
  assumes pre: bmssp-pre-full d S (Fin Bmax)
    and upper: ⋀u. u ∈ S ⟹ d u < Bmax
    and D-def: D = label-partition-view d S
    and S'-def: S' = sorted-pull-set M D
    and beta-def: beta = sorted-pull-bound M Bmax D
  shows bmssp-pre-full d S' (Fin beta)
proof −
  have S-subset: S ⊆ V
    using pre unfolding bmssp-pre-full-def by blast
  have finite-S: finite S
    using finite-subset[OF S-subset finite-V] .
  let ?xs = partition-key-order D
  have set-xs: set ?xs = S
    using partition-key-order-properties(1)[of D] finite-S
    unfolding D-def by simp
  show ?thesis
  proof (cases length ?xs ≤ M)
    case True
    have S'-eq: S' = S
      using True set-xs unfolding S'-def sorted-pull-set-def by (simp add: Let-def)
    have beta-eq: beta = Bmax
      using True unfolding beta-def sorted-pull-bound-def by (simp add: Let-def)
    show ?thesis
      using pre unfolding S'-eq beta-eq .
  next
    case False
    then have M-lt: M < length ?xs
      by simp
    have beta-eq: beta = d (?xs ! M)
      using False unfolding beta-def sorted-pull-bound-def D-def by (simp add:
Let-def)
    have nth-in: ?xs ! M ∈ S
      using set-xs M-lt nth-mem by metis
    have beta-lt: beta < Bmax
      using upper[OF nth-in] beta-eq by simp
    have S'-split: S' = split-below d S beta
      using sorted-pull-set-eq-split-below[OF finite-S upper D-def beta-def]
      unfolding S'-def .
    show ?thesis
      unfolding S'-split
      by (rule pull-minimum-pre-for-lower-split[OF pre]) (simp add: beta-lt)
  qed
qed
```

The final theorem packages the precondition transfer needed by a recursive BMSSP child call. If the parent problem is valid up to *Bmax* and all current source labels lie below *Bmax*, then the set returned by a sorted pull is valid up to the returned child bound. The all-keys case is just the parent precondition again; the proper-prefix case uses the pull-minimum lemma for lower splits.

Later operational proofs use this theorem as a template for arbitrary partition implementations: once a pull is known to select the same lower split, the recursive child precondition follows from the same argument.

end

end
theory *BMSSP-Initialization*
imports *BMSSP-Shortest-Path-Lemmas*
begin

23 Initial Labels

BMSSP starts from the usual shortest-path initialization: the source is complete and all other labels are conservative upper bounds. In an imperative implementation one would normally write this as distance zero for the source and infinity for every other vertex. The abstract correctness layer in this entry uses real-valued labels rather than an extended-real datatype, so this theory gives a finite replacement for infinity.

The replacement is the maximum true source distance over the finite vertex set. This is not intended as an executable way to discover shortest paths; it is a semantic initial label used in the proof stack. Because the graph locale is finite, the maximum exists. Because every reachable vertex has distance at most that maximum, the label function is sound. The source remains exact because the distance from any vertex to itself is zero under non-negative edge weights.

The lemmas below isolate precisely the facts needed by later top-level theorems: the source label is complete, the whole label function is sound, and the root call with source set $\{s\}$ and bound *Infinity* satisfies the BMSSP precondition.

context *finite-weighted-digraph*
begin

definition *finite-initial-label* **where**
finite-initial-label v = (*if v* = *s then 0 else Max* (*dist s* ‘ *V*))

The definition *finite-initial-label* is intentionally semantic. Its non-source value mentions *local.dist*, so it is not the data structure used by the exported example. Its role is to let the correctness proof start from a finite real-valued label function while preserving the same logical shape as Dijkstra's standard initialization.

```
lemma dist-refl-zero:
  assumes v ∈ V
  shows dist v v = 0
proof −
  have simple: simple-walk-betw v [v] v
    using assms unfolding simple-walk-betw-def walk-betw-def by simp
  have dist v v ≤ 0
    using dist-le-simple-walk-weight[OF simple] by simp
  moreover have 0 ≤ dist v v
  proof −
    have reachable v v
      using reachable-refl[OF assms] .
    then have dist v v ∈ simple-walk-weights v v
      using dist-is-simple-walk-weight by blast
    then obtain p where p: simple-walk-betw v p v walk-weight p = dist v v
      unfolding simple-walk-weights-def by auto
    then have walk p
      unfolding simple-walk-betw-def walk-betw-def by auto
    then have 0 ≤ walk-weight p
      by (rule walk-weight-nonneg)
    then show ?thesis
      using p by simp
  qed
  ultimately show ?thesis
    by simp
qed

lemma finite-initial-label-source-complete:
  finite-initial-label s = dist s s
  using dist-refl-zero[OF source-in-V] unfolding finite-initial-label-def by simp
```

The proof of $?v \in V \Longrightarrow local.dist\ ?v\ ?v = 0$ uses the one-vertex simple walk for the upper bound and non-negativity of walk weights for the lower bound. From it, $finite\text{-}initial\text{-}label\ s = local.dist\ s\ s$ gives the exact source label required by root BMSSP initialization. The two soundness lemmas below differ only in whether reachability of every vertex is supplied explicitly or left as the local reachable premise of *sound-label*.

```
lemma finite-initial-label-sound:
  assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
  shows sound-label finite-initial-label
proof −
  have nonempty: dist s ‘ V ≠ {}
    using source-in-V by blast
  show ?thesis
    unfolding sound-label-def finite-initial-label-def
  proof clarify
    fix v
    assume vV: v ∈ V and reach-v: reachable s v
    show dist s v ≤ (if v = s then 0 else Max (dist s ‘ V))
```

```
    proof (cases v = s)
      case True
      then show ?thesis
        using dist-refl-zero[OF source-in-V] by simp
    next
      case False
      have dist s v ∈ dist s ‘ V
        using vV by blast
      then have dist s v ≤ Max (dist s ‘ V)
        using finite-V by simp
      then show ?thesis
        using False by simp
    qed
  qed
qed

lemma finite-initial-label-sound-reachable:
  shows sound-label finite-initial-label
proof −
  have nonempty: dist s ‘ V ≠ {}
    using source-in-V by blast
  show ?thesis
    unfolding sound-label-def finite-initial-label-def
  proof clarify
    fix v
    assume vV: v ∈ V and reach-v: reachable s v
    show dist s v ≤ (if v = s then 0 else Max (dist s ‘ V))
    proof (cases v = s)
      case True
      then show ?thesis
        using dist-refl-zero[OF source-in-V] by simp
    next
      case False
      have dist s v ∈ dist s ‘ V
        using vV by blast
      then have dist s v ≤ Max (dist s ‘ V)
        using finite-V nonempty by simp
      then show ?thesis
        using False by simp
    qed
  qed
qed

lemma initialized-root-pre-full:
  assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
  shows bmssp-pre finite-initial-label {s} Infinity
  using top-bmssp-pre[OF all-reachable, of finite-initial-label] .
```

Finally, $(\bigwedge v.\ v \in V \Longrightarrow \mathit{reachable}\ s\ v) \Longrightarrow \mathit{bmssp\text{-}pre}\ \mathit{finite\text{-}initial\text{-}label}\ \{s\}\ \mathit{Infinity}$ packages the initialized labels for the root abstract BMSSP call.

The source set is the singleton $\{s\}$, the bound is *Infinity*, and every reachable vertex lies on a shortest path through the source. Later theories strengthen this precondition to the complete-source form and then discharge the full algorithmic step relation.

end

end
theory *BMSSP-Concrete-Step*
  imports *BMSSP-Pull-Minimum*
begin

24 Concrete One-Step BMSSP Assembly

This theory packages the concrete part of a non-base BMSSP call. The abstract correctness skeleton states its recursive middle phase with the predicate *partition-loop-post*: after FindPivots has chosen a pivot set, the loop must solve the bounded tree induced by those pivots. That statement is intentionally compact, but it hides the range structure that the implementation actually follows.

The concrete step exposed here replaces that abstract loop contract by an explicit monotone trace of child ranges. A trace contains the first lower boundary, the list of subsequent boundaries, the label function after the loop, and the collection of vertex sets solved by the child calls. The trace proof checks that the boundaries are nondecreasing, remain below the returned BMSSP bound, and cover exactly the chain of bounded shortest-path trees that the partition loop is supposed to assemble.

The final step definitions then restore FindPivots. The uncapped version uses the conceptual FindPivots operation, while the capped version uses the executable bounded search from the paper. Both definitions have the same shape: perform FindPivots, solve the pivot loop, keep the vertices completed directly by FindPivots, and return their union. The two correctness theorems below show that this concrete assembly is still a valid *bmssp-post-full* result.

context *unique-shortest-digraph*
begin

definition *concrete-partition-loop-trace* **where**
  concrete-partition-loop-trace P B a bs d′ B′ Us U $\longleftrightarrow$
    bound-le B′ B $\wedge$
    nondecreasing-from a bs $\wedge$
    bounds-le B′ (*a* # *bs*) $\wedge$
    complete-on d′ (*bound-tree P* (*Fin a*)) $\wedge$
    list-all2 ($\lambda U\ X.\ U = X \wedge$ *complete-on d′ U*) *Us*
      (*range-tree-chain-list P a bs B′*) $\wedge$
    U = *bound-tree P* (*Fin a*) $\cup \bigcup$(*set Us*)

theorem *concrete-partition-loop-trace-post*:
  assumes *trace*: *concrete-partition-loop-trace P B a bs d′ B′ Us U*
  shows *bmssp-post-full d P B d′ B′ U*
proof −
  have *le*: *bound-le B′ B*
    and *mono*: *nondecreasing-from a bs*
    and *bounds*: *bounds-le B′ (a # bs)*
    and *lower*: *complete-on d′ (bound-tree P (Fin a))*
    and *children*: *list-all2 (λU X. U = X ∧ complete-on d′ U) Us*
      (range-tree-chain-list P a bs B′)
    and *U*: *U = bound-tree P (Fin a) ∪ ⋃(set Us)*
    using *trace* **unfolding** *concrete-partition-loop-trace-def* **by** *blast+*
  show *?thesis*
    by (*rule partition-loop-trace-assembly-post*
      [*OF le mono bounds lower children U*])
qed

definition *concrete-bmssp-step* **where**
  concrete-bmssp-step k d S B a bs d′ B′ Us U ⟷
    (*let d-fp = find-pivots-label k d S B*;
        P = find-pivots-pivots k d S B;
        W = {v ∈ bound-tree S B′. d-fp v = dist s v}
     in ∃ U-loop.
        concrete-partition-loop-trace P B a bs d′ B′ Us U-loop ∧
        complete-on d′ W ∧
        U = U-loop ∪ W)

definition *concrete-capped-bmssp-step* **where**
  concrete-capped-bmssp-step k cap d S B a bs d′ B′ Us U ⟷
    (*let d-fp = find-pivots-label-capped k cap d S B*;
        P = find-pivots-pivots-capped k cap d S B;
        W = {v ∈ bound-tree S B′. d-fp v = dist s v}
     in ∃ U-loop.
        concrete-partition-loop-trace P B a bs d′ B′ Us U-loop ∧
        complete-on d′ W ∧
        U = U-loop ∪ W)

The definition *concrete-partition-loop-trace* is the local certificate for the recursive partition loop. It does not say how pulls are generated; it says what must be true of the resulting range sequence. The theorem *concrete-partition-loop-trace ?P ?B ?a ?bs ?d′ ?B′ ?Us ?U ⟹ bmssp-post-full ?d ?P ?B ?d′ ?B′ ?U* then converts such a trace into *bmssp-post-full* for the pivot set. The step predicates *concrete-bmssp-step* and *concrete-capped-bmssp-step* wrap the trace with FindPivots and the final union of loop-completed and FindPivots-completed vertices.

theorem *concrete-bmssp-step-correct*:
  assumes *sound*: *sound-label d*
    and *pre*: *bmssp-pre-full d S B*
    and *S-reaches*: *⋀x. x ∈ S ⟹ reachable s x*

and *step*: *concrete-bmssp-step k d S B a bs d′ B′ Us U*
shows *bmssp-post-full d S B d′ B′ U*
proof −
let *?d-fp* = *find-pivots-label k d S B*
let *?P* = *find-pivots-pivots k d S B*
let *?W* = {*v* ∈ *bound-tree S B′*. *?d-fp v* = *dist s v*}
obtain *U-loop* **where**
trace: *concrete-partition-loop-trace ?P B a bs d′ B′ Us U-loop*
and *compW*: *complete-on d′ ?W*
and *U*: *U* = *U-loop* ∪ *?W*
using *step* **unfolding** *concrete-bmssp-step-def* **by** (*auto simp*: *Let-def*)
have *loop*: *bmssp-post-full ?d-fp ?P B d′ B′ U-loop*
using *concrete-partition-loop-trace-post*[*OF trace*] **.**
have *le*: *bound-le B′ B*
using *loop* **unfolding** *bmssp-post-full-def* **by** *blast*
have *U-tree*: *U-loop* ∪ *?W* = *bound-tree S B′*
using *concrete-find-pivots-final-tree-assembly*[*OF sound pre S-reaches loop*] **.**
have *loop-complete*: *complete-on d′ U-loop*
using *loop* **unfolding** *bmssp-post-full-def* **by** *blast*
have *complete*: *complete-on d′* (*U-loop* ∪ *?W*)
using *complete-on-Un*[*OF loop-complete compW*] **.**
have *bmssp-post-full d S B d′ B′* (*U-loop* ∪ *?W*)
using *le U-tree complete* **unfolding** *bmssp-post-full-def* **by** *blast*
then show *?thesis*
using *U* **by** *simp*
qed

theorem *concrete-capped-bmssp-step-correct*:
assumes *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: ⋀*x*. *x* ∈ *S* ⟹ *reachable s x*
and *step*: *concrete-capped-bmssp-step k cap d S B a bs d′ B′ Us U*
shows *bmssp-post-full d S B d′ B′ U*
proof −
let *?d-fp* = *find-pivots-label-capped k cap d S B*
let *?P* = *find-pivots-pivots-capped k cap d S B*
let *?W* = {*v* ∈ *bound-tree S B′*. *?d-fp v* = *dist s v*}
obtain *U-loop* **where**
trace: *concrete-partition-loop-trace ?P B a bs d′ B′ Us U-loop*
and *compW*: *complete-on d′ ?W*
and *U*: *U* = *U-loop* ∪ *?W*
using *step* **unfolding** *concrete-capped-bmssp-step-def* **by** (*auto simp*: *Let-def*)
have *loop*: *bmssp-post-full ?d-fp ?P B d′ B′ U-loop*
using *concrete-partition-loop-trace-post*[*OF trace*] **.**
have *le*: *bound-le B′ B*
using *loop* **unfolding** *bmssp-post-full-def* **by** *blast*
have *U-tree*: *U-loop* ∪ *?W* = *bound-tree S B′*
using *concrete-capped-find-pivots-final-tree-assembly*[*OF sound pre S-reaches loop*] **.**

have *loop-complete*: *complete-on* d' *U-loop*
using *loop* **unfolding** *bmssp-post-full-def* **by** *blast*
have *complete*: *complete-on* d' (*U-loop* $\cup$ *?W*)
using *complete-on-Un*[*OF loop-complete compW*] **.**
have *bmssp-post-full d S B* d' B' (*U-loop* $\cup$ *?W*)
using *le U-tree complete* **unfolding** *bmssp-post-full-def* **by** *blast*
then show *?thesis*
using *U* **by** *simp*
qed

end

end
theory *BMSSP-Concrete-Top-Level*
imports *BMSSP-Initialization BMSSP-Concrete-Step*
begin

25 Concrete Top-Level Correctness

These theorems connect the finite initial labels to the concrete one-step BMSSP assembly relation. They avoid the abstract *abstract-bmssp-step* contract used by the abstract interface.

The top-level SSSP call is a BMSSP call with singleton source set $\{s\}$ and infinite bound. The finite initial label gives distance *0* at the source and an effectively infinite value elsewhere. Once a concrete BMSSP step proves the full BMSSP postcondition for this root problem, the infinite-bound tree is the whole vertex set, so completeness of the returned label function is exactly SSSP correctness.

Two entry points are provided because the development has both an uncapped concrete step and the capped step used by the paper's parameter schedule. They share the same proof: establish the root precondition, invoke the corresponding concrete-step correctness theorem, then collapse the root BMSSP postcondition to *sssp-correct*.

context *unique-shortest-digraph*
begin

theorem *finite-initial-label-concrete-top-level-correct*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *step*: *concrete-bmssp-step k finite-initial-label* $\{s\}$ *Infinity a bs* d' *Infinity Us U*
shows *sssp-correct* d'
proof −
have *pre*: *bmssp-pre-full finite-initial-label* $\{s\}$ *Infinity*
using *all-reachable finite-initial-label-source-complete*
by (*rule top-bmssp-pre-full*)
have *sound*: *sound-label finite-initial-label*
using *finite-initial-label-sound*[*OF all-reachable*] **.**

have *S-reaches*: $\bigwedge x.\ x \in \{s\} \Longrightarrow$ *reachable s x*
using *all-reachable source-in-V* **by** *blast*
have *post-full*:
bmssp-post-full finite-initial-label $\{s\}$ *Infinity d' Infinity U*
by (*rule concrete-bmssp-step-correct*[*OF sound pre S-reaches step*])
have *bmssp-post finite-initial-label* $\{s\}$ *Infinity d' Infinity U*
using *bmssp-post-full-imp-post*[*OF post-full*] **.**
then have *U-V*: $U = V$
using *bound-tree-source-infinity*[*OF all-reachable*]
unfolding *bmssp-post-def* **by** *auto*
have *complete*: *complete-on d' U*
using ‹*bmssp-post finite-initial-label* $\{s\}$ *Infinity d' Infinity U*›
unfolding *bmssp-post-def* **by** *auto*
then show *?thesis*
using *U-V* **unfolding** *complete-on-def sssp-correct-def* **by** *auto*
qed

theorem *finite-initial-label-concrete-capped-top-level-correct*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *step*: *concrete-capped-bmssp-step k cap finite-initial-label* $\{s\}$ *Infinity a bs d' Infinity Us U*
shows *sssp-correct d'*
proof −
have *pre*: *bmssp-pre-full finite-initial-label* $\{s\}$ *Infinity*
using *all-reachable finite-initial-label-source-complete*
by (*rule top-bmssp-pre-full*)
have *sound*: *sound-label finite-initial-label*
using *finite-initial-label-sound*[*OF all-reachable*] **.**
have *S-reaches*: $\bigwedge x.\ x \in \{s\} \Longrightarrow$ *reachable s x*
using *all-reachable source-in-V* **by** *blast*
have *post-full*:
bmssp-post-full finite-initial-label $\{s\}$ *Infinity d' Infinity U*
by (*rule concrete-capped-bmssp-step-correct*[*OF sound pre S-reaches step*])
have *bmssp-post finite-initial-label* $\{s\}$ *Infinity d' Infinity U*
using *bmssp-post-full-imp-post*[*OF post-full*] **.**
then have *U-V*: $U = V$
using *bound-tree-source-infinity*[*OF all-reachable*]
unfolding *bmssp-post-def* **by** *auto*
have *complete*: *complete-on d' U*
using ‹*bmssp-post finite-initial-label* $\{s\}$ *Infinity d' Infinity U*›
unfolding *bmssp-post-def* **by** *auto*
then show *?thesis*
using *U-V* **unfolding** *complete-on-def sssp-correct-def* **by** *auto*
qed

end

end
theory *BMSSP-Recursive*

imports *BMSSP-Base-Case BMSSP-Concrete-Top-Level*
begin

26 Level-Indexed BMSSP Relation

This theory introduces the first level-indexed BMSSP relation. Earlier theories prove a base case and a concrete non-base step in isolation. The inductive relation below is the small bridge that turns those one-step facts into a recursive BMSSP run indexed by a natural recursion level.

The relation is deliberately simple. At level zero it uses the checked base case for a singleton source set. At a successor level it accepts any concrete capped BMSSP step assembled from FindPivots and a monotone partition trace. This file does not yet model the operational generation of every pull and child call; that more detailed relation is proved separately in *BMSSP-Operational-Pull*. Here the purpose is to show that the base/step decomposition is already enough for semantic correctness.

The main theorem *concrete-bmssp-correct* below is an induction over the level-indexed relation. The base rule delegates to the base-case postcondition, and the step rule delegates to the concrete capped step theorem. The final theorem specializes the relation to the root source and infinite bound, obtaining ordinary single-source shortest-path correctness for the finite initial label.

context *unique-shortest-digraph*
begin

inductive *concrete-bmssp* **where**
Base:
$S = \{x\} \Longrightarrow$
concrete-bmssp k cap 0 d S B
(λv. *if* $v \in$ *base-case-vertices k x B then dist s v else d v*)
(*base-case-bound k x B*)
(*base-case-vertices k x B*)
| *Step*:
concrete-capped-bmssp-step k cap d S B a bs d′ B′ Us U $\Longrightarrow$
concrete-bmssp k cap (*Suc l*) *d S B d′ B′ U*

The two rules of *concrete-bmssp* mirror the recursive structure of the paper. The base rule solves the bounded region reachable from one source by the base-case routine. The successor rule abstracts over the child runs already summarized by *concrete-capped-bmssp-step*. This keeps the induction proof short and makes clear which theorem is responsible for each layer of the algorithm.

lemma *bmssp-post-imp-post-full*:
assumes *bmssp-post d S B d′ B′ U*
shows *bmssp-post-full d S B d′ B′ U*

using *assms* **unfolding** *bmssp-post-def bmssp-post-full-def* **by** *blast*

theorem *concrete-bmssp-correct*:
assumes *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *run*: *concrete-bmssp k cap l d S B d′ B′ U*
shows *bmssp-post-full d S B d′ B′ U*
using *run sound pre S-reaches*
proof (*induction arbitrary*: *rule*: *concrete-bmssp.induct*)
case (*Base S x k cap d B*)
have *post*:
bmssp-post d S B
(*λv. if v ∈ base-case-vertices k x B then dist s v else d v*)
(*base-case-bound k x B*)
(*base-case-vertices k x B*)
using *base-case-result-bmssp-post*[*OF Base.hyps*, **where** $k = k$ **and** $B = B$ **and** $d = d$]
unfolding *base-case-result-def* **by** *simp*
then show *?case*
by (*rule bmssp-post-imp-post-full*)
next
case (*Step k cap d S B a bs d′ B′ Us U l*)
show *?case*
by (*rule concrete-capped-bmssp-step-correct*
[*OF Step.prems*(*1*) *Step.prems*(*2*) *Step.prems*(*3*) *Step.hyps*])
qed

The proof of ⟦*sound-label ?d*; *bmssp-pre-full ?d ?S ?B*; $\bigwedge x.\ x \in$ *?S* $\Longrightarrow$ *reachable s x*; *concrete-bmssp ?k ?cap ?l ?d ?S ?B ?d′ ?B′ ?U*⟧ $\Longrightarrow$ *bmssp-post-full ?d ?S ?B ?d′ ?B′ ?U* is intentionally direct. All shortest-path reasoning has already been packaged in the base-case and concrete-step theories, so the induction only selects the appropriate correctness theorem for each rule. The remaining top-level theorem then supplies the standard root precondition, soundness of the initial label, and reachability of the singleton source set.

theorem *finite-initial-label-recursive-top-level-correct*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *run*: *concrete-bmssp k cap l finite-initial-label* {*s*} *Infinity d′ Infinity U*
shows *sssp-correct d′*
proof −
have *pre*: *bmssp-pre-full finite-initial-label* {*s*} *Infinity*
using *all-reachable finite-initial-label-source-complete*
by (*rule top-bmssp-pre-full*)
have *sound*: *sound-label finite-initial-label*
using *finite-initial-label-sound*[*OF all-reachable*] .
have *S-reaches*: $\bigwedge x.\ x \in \{s\} \Longrightarrow$ *reachable s x*
using *all-reachable source-in-V* **by** *blast*
have *post-full*:

bmssp-post-full finite-initial-label {*s*} *Infinity d′ Infinity U*
by (*rule concrete-bmssp-correct*[*OF sound pre S-reaches run*])
have *post*: *bmssp-post finite-initial-label* {*s*} *Infinity d′ Infinity U*
using *bmssp-post-full-imp-post*[*OF post-full*] **.**
then have *U-V*: *U* = *V*
using *bound-tree-source-infinity*[*OF all-reachable*]
unfolding *bmssp-post-def* **by** *auto*
have *complete*: *complete-on d′ U*
using *post* **unfolding** *bmssp-post-def* **by** *auto*
then show *?thesis*
using *U-V* **unfolding** *complete-on-def sssp-correct-def* **by** *auto*
qed

end

end
theory *BMSSP-Operational-Pull*
imports *BMSSP-Partition-Pull-Bridge BMSSP-Recursive*
begin

27 Operational Pull Step

This theory is the bridge from the abstract recursive BMSSP specification to an operational loop that explicitly pulls a bounded prefix out of a partition, recurses on that prefix, and then pushes newly discovered labels into the remaining partition. Earlier theories prove the abstract correctness theorem for *concrete-bmssp*; here we expose the loop shape that the later cost relations charge.

The paper's partition loop is not merely a list traversal. Each iteration selects a child source set below a temporary bound, runs BMSSP recursively on that child problem, and then refreshes the parent partition with two kinds of labels: edge relaxations out of the completed child tree and old source labels that now fall into the open range just processed. The predicate *pull-separates* is the abstract statement that the pull operation has found exactly such a prefix and retained an ordered residual partition.

The main proof obligation is a lifting argument. If a child recursive call satisfies *bmssp-post-full* for the split prefix, and if the parent already satisfied *bmssp-pre-full* up to the larger bound, then the child result can be viewed as progress for the parent. The connective tissue is the lower split *split-below*: it identifies the prefix chosen by the data structure with the mathematical set of vertices whose current labels are below the recursive child bound.

Later in the file the same idea is rephrased as inductive operational traces. The traces remember the sequence of child output bounds, the range trees *range-tree* that those bounds cut out, and the final tree assembled from all slices. This is the form needed by the range-synchronised cost theories, where every loop iteration must be charged to the slice it actually completed.

context *unique-shortest-digraph*
begin

theorem *sorted-pull-recursive-child-lifts*:
assumes *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S* (*Fin Bmax*)
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *upper*: $\bigwedge u.\ u \in S \Longrightarrow d\ u < Bmax$
and *S'-def*: $S' =$ *sorted-pull-set M* (*label-partition-view d S*)
and *beta-def*: *beta* = *sorted-pull-bound M Bmax* (*label-partition-view d S*)
and *run*: *concrete-bmssp k cap l d S'* (*Fin beta*) *d' B' U*
shows *bmssp-post-full d S* (*Fin Bmax*) *d' B' U*
proof −
have *S-subset*: $S \subseteq V$
using *pre* **unfolding** *bmssp-pre-full-def* **by** *blast*
have *finite-S*: *finite S*
using *finite-subset*[*OF S-subset finite-V*] **.**
let *?D* = *label-partition-view d S*
let *?xs* = *partition-key-order ?D*
have *set-xs*: *set ?xs* = *S*
using *partition-key-order-properties*(*1*)[*of ?D*] *finite-S* **by** *simp*
have *child-pre*: *bmssp-pre-full d S'* (*Fin beta*)
by (*rule sorted-pull-establishes-lower-pre*
[*OF pre upper refl S'-def beta-def*])
have *S'-split*: $S' =$ *split-below d S beta*
using *sorted-pull-set-eq-split-below*[*OF finite-S upper refl beta-def*]
unfolding *S'-def* **.**
have *S'-reaches*: $\bigwedge x.\ x \in S' \Longrightarrow$ *reachable s x*
using *S-reaches S'-split* **unfolding** *split-below-def* **by** *blast*
have *child-post*: *bmssp-post-full d S'* (*Fin beta*) *d' B' U*
by (*rule concrete-bmssp-correct*[*OF sound child-pre S'-reaches run*])
have *beta-le*: *beta* $\leq$ *Bmax*
proof (*cases length ?xs* $\leq$ *M*)
case *True*
then show *?thesis*
unfolding *beta-def sorted-pull-bound-def* **by** (*simp add*: *Let-def*)
next
case *False*
then have *M-lt*: $M <$ *length ?xs*
by *simp*
have *beta-eq*: *beta* = *d* (*?xs* ! *M*)
using *False* **unfolding** *beta-def sorted-pull-bound-def* **by** (*simp add*: *Let-def*)
have *?xs* ! *M* $\in S$
using *set-xs M-lt nth-mem* **by** *metis*
then have *beta* $<$ *Bmax*
using *upper beta-eq* **by** *simp*
then show *?thesis*
by *simp*
qed

```
  have pre-beta: bmssp-pre-full d S (Fin beta)
    using bmssp-pre-full-bound-mono[OF pre, of Fin beta] beta-le by simp
  have cover-beta: complete-tree-cover d S (Fin beta)
    using bmssp-pre-full-complete-tree-cover[OF pre-beta] .
  have child-post-split:
    bmssp-post-full d (split-below d S beta) (Fin beta) d' B' U
    using child-post unfolding S'-split .
  have U-tree: U = bound-tree S B'
    using pull-recursive-post-lifts-bound-tree[OF cover-beta child-post-split] .
  have le-child: bound-le B' (Fin beta)
    using child-post unfolding bmssp-post-full-def by blast
  have le-parent: bound-le B' (Fin Bmax)
    using le-child beta-le by (cases B') auto
  have complete: complete-on d' U
    using child-post unfolding bmssp-post-full-def by blast
  show ?thesis
    using le-parent U-tree complete unfolding bmssp-post-full-def by blast
qed

theorem pull-separates-recursive-child-lifts:
  fixes Bmax :: real
  assumes sound: sound-label d
    and pre: bmssp-pre-full d S (Fin Bmax)
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and upper: ⋀u. u ∈ S ⟹ d u < Bmax
    and pull: pull-separates (label-partition-view d S) M Bmax S' beta D'
    and run: concrete-bmssp k cap l d S' (Fin beta) d' B' U
  shows bmssp-post-full d S (Fin Bmax) d' B' U
proof −
  have S'-split: S' = split-below d S beta
    using pull-separates-label-set-eq-split-below[OF pull upper] .
  have S'-subset: S' ⊆ S
    using pull-separates-subset[OF pull] by simp
  have beta-le: beta ≤ Bmax
  proof (cases keys-of D' = {})
    case True
    then show ?thesis
      using pull-separates-empty-bound[OF pull True] by simp
  next
    case False
    then obtain v where vD': v ∈ keys-of D'
      by blast
    have D'-keys: keys-of D' = S − S'
      using pull unfolding pull-separates-def by simp
    then have vS: v ∈ S
      using vD' by blast
    have beta ≤ value-of D' v
      by (rule pull-separates-nonempty-bound[OF pull False vD'])
    also have ... = d v
```

```
    using pull unfolding pull-separates-def by simp
  also have ... < Bmax
    using upper[OF vS] .
  finally show ?thesis
    by simp
qed
have child-pre: bmssp-pre-full d S' (Fin beta)
proof (cases beta = Bmax)
  case True
  have split-all: split-below d S beta = S
    unfolding split-below-def True using upper by blast
  show ?thesis
    using pre S'-split split-all True by simp
next
  case False
  then have beta-lt: beta < Bmax
    using beta-le by simp
  have bmssp-pre-full d (split-below d S beta) (Fin beta)
    by (rule pull-minimum-pre-for-lower-split[OF pre]) (simp add: beta-lt)
  then show ?thesis
    using S'-split by simp
qed
have S'-reaches: ⋀x. x ∈ S' ⟹ reachable s x
  using S'-subset S-reaches by blast
have child-post: bmssp-post-full d S' (Fin beta) d' B' U
  by (rule concrete-bmssp-correct[OF sound child-pre S'-reaches run])
have pre-beta: bmssp-pre-full d S (Fin beta)
proof −
  have bound-le (Fin beta) (Fin Bmax)
    using beta-le by simp
  then show ?thesis
    by (rule bmssp-pre-full-bound-mono[OF pre])
qed
have cover-beta: complete-tree-cover d S (Fin beta)
  using bmssp-pre-full-complete-tree-cover[OF pre-beta] .
have child-post-split:
  bmssp-post-full d (split-below d S beta) (Fin beta) d' B' U
  using child-post S'-split by simp
have U-tree: U = bound-tree S B'
  using pull-recursive-post-lifts-bound-tree[OF cover-beta child-post-split] .
have le-child: bound-le B' (Fin beta)
  using child-post unfolding bmssp-post-full-def by blast
have le-parent: bound-le B' (Fin Bmax)
  using le-child beta-le by (cases B') auto
have complete: complete-on d' U
  using child-post unfolding bmssp-post-full-def by blast
show ?thesis
  using le-parent U-tree complete unfolding bmssp-post-full-def by blast
qed
```

The first two lifting lemmas isolate the correctness content of a pull. The theorem $[\![$*sound-label ?d*; *bmssp-pre-full ?d ?S* (*Fin ?Bmax*); $\bigwedge$*x*. *x* $\in$ *?S* $\Longrightarrow$ *reachable s x*; $\bigwedge$*u*. *u* $\in$ *?S* $\Longrightarrow$ *?d u* $<$ *?Bmax*; *?S′* = *sorted-pull-set ?M* (*label-partition-view ?d ?S*); *?beta* = *sorted-pull-bound ?M ?Bmax* (*label-partition-view ?d ?S*); *concrete-bmssp ?k ?cap ?l ?d ?S′* (*Fin ?beta*) *?d′ ?B′ ?U*$]\!]$ $\Longrightarrow$ *bmssp-post-full ?d ?S* (*Fin ?Bmax*) *?d′ ?B′ ?U* is phrased for the executable sorted-list partition view: the child set and child bound are computed by sorting labels and taking the first M positions. The theorem $[\![$*sound-label ?d*; *bmssp-pre-full ?d ?S* (*Fin ?Bmax*); $\bigwedge$*x*. *x* $\in$ *?S* $\Longrightarrow$ *reachable s x*; $\bigwedge$*u*. *u* $\in$ *?S* $\Longrightarrow$ *?d u* $<$ *?Bmax*; *pull-separates* (*label-partition-view ?d ?S*) *?M ?Bmax ?S′ ?beta ?D′*; *concrete-bmssp ?k ?cap ?l ?d ?S′* (*Fin ?beta*) *?d′ ?B′ ?U*$]\!]$ $\Longrightarrow$ *bmssp-post-full ?d ?S* (*Fin ?Bmax*) *?d′ ?B′ ?U* is the abstract version used by the bucketed partition and by the costed loop relations.

Both statements follow the same pattern. The pulled set is shown to equal the mathematical split below the child bound, the child precondition is inherited from the parent precondition, and the child postcondition is lifted through the parent's complete tree cover. The conclusion deliberately reuses *bmssp-post-full*; nothing about the later operational or costed presentation changes the semantic contract of BMSSP.

definition *complete-preserved* **where**
  complete-preserved d d′ U $\longleftrightarrow$ (*complete-on d U* $\longrightarrow$ *complete-on d′ U*)

lemma *relax-edges-complete-preserved*:
  assumes *sound*: *sound-label d*
    and *edges*: $\bigwedge$*u v*. (*u*, *v*) $\in$ *set es* $\Longrightarrow$ (*u*, *v*) $\in$ *E*
    and *reaches*: $\bigwedge$*u v*. (*u*, *v*) $\in$ *set es* $\Longrightarrow$ *reachable s u*
  shows *complete-preserved d* (*relax-edges d es*) *U*
proof −
  have *preserve*: $\bigwedge$*x*. *d x* = *dist s x* $\Longrightarrow$ *relax-edges d es x* = *dist s x*
  proof −
    fix *x*
    assume *complete-x*: *d x* = *dist s x*
    show *relax-edges d es x* = *dist s x*
      by (*rule relax-edges-preserves-complete-sound*
        [*OF sound complete-x edges reaches*])
  qed
  show *?thesis*
    unfolding *complete-preserved-def complete-on-def* **using** *preserve* **by** *blast*
qed

lemma *relax-frontier-complete-preserved*:
  assumes *sound*: *sound-label d*
    and *reaches*: $\bigwedge$*u*. *u* $\in$ *F* $\Longrightarrow$ *reachable s u*
  shows *complete-preserved d* (*relax-frontier d F*) *U*
proof −

```
  have preserve: ⋀x. d x = dist s x ⟹ relax-frontier d F x = dist s x
  proof −
    fix x
    assume complete-x: d x = dist s x
    show relax-frontier d F x = dist s x
      by (rule relax-frontier-preserves-complete-sound
        [OF sound complete-x reaches])
  qed
  show ?thesis
    unfolding complete-preserved-def complete-on-def using preserve by blast
qed
```

After a child call returns, the parent loop relaxes edges out of the completed child tree and may also reinsert source labels from the pulled set. The small predicate *complete-preserved* packages the only semantic requirement imposed on that refresh: labels already complete on the child slice stay complete after the refresh.

The lemmas ⟦*sound-label ?d*; ⋀*u v*. (*u*, *v*) ∈ *set ?es* ⟹ (*u*, *v*) ∈ *E*; ⋀*u v*. (*u*, *v*) ∈ *set ?es* ⟹ *reachable s u*⟧ ⟹ *complete-preserved ?d* (*relax-edges ?d ?es*) *?U* and ⟦*sound-label ?d*; ⋀*u*. *u* ∈ *?F* ⟹ *reachable s u*⟧ ⟹ *complete-preserved ?d* (*relax-frontier ?d ?F*) *?U* are deliberately local and modest. They do not prove a new shortest-path theorem; they just record that ordinary sound relaxation cannot destroy a vertex whose label has already reached its true distance. This lets the loop proofs compose recursive completion with parent-side batch updates without reopening the relaxation algebra every time.

```
definition edge-relaxation-pairs-between where
  edge-relaxation-pairs-between d U L H =
    map (λ(u, v). (v, d u + w u v))
      (filter (λ(u, v). L ≤ d u + w u v ∧ d u + w u v < H)
        (edge-list-of (outgoing-edges U)))

definition label-pairs-between where
  label-pairs-between d S L H =
    map (λx. (x, d x))
      (filter (λx. L ≤ d x ∧ d x < H)
        (partition-key-order (label-partition-view d S)))

lemma edge-relaxation-pairs-between-value-le-high:
  assumes (x, b) ∈ set (edge-relaxation-pairs-between d U L H)
  shows b ≤ H
  using assms unfolding edge-relaxation-pairs-between-def
  by (auto split: prod.splits)

lemma edge-relaxation-pairs-between-value-ge-low:
  assumes (x, b) ∈ set (edge-relaxation-pairs-between d U L H)
  shows L ≤ b
  using assms unfolding edge-relaxation-pairs-between-def
```

by (*auto split*: *prod.splits*)

lemma *label-pairs-between-value-le-high*:
assumes $(x, b) \in$ *set* (*label-pairs-between d S L H*)
shows $b \leq H$
using *assms* **unfolding** *label-pairs-between-def* **by** *auto*

lemma *label-pairs-between-value-ge-low*:
assumes $(x, b) \in$ *set* (*label-pairs-between d S L H*)
shows $L \leq b$
using *assms* **unfolding** *label-pairs-between-def* **by** *auto*

lemma *edge-relaxation-pairs-between-length-le-outgoing*:
length (*edge-relaxation-pairs-between d U L H*) ≤
card (*outgoing-edges U*)
proof −
let *?es* = *edge-list-of* (*outgoing-edges U*)
have *distinct-es*: *distinct ?es*
using *edge-list-of-properties*(*2*)[*OF finite-outgoing-edges*] **.**
have *length-es*: *length ?es* = *card* (*outgoing-edges U*)
using *edge-list-of-properties*(*1*)[*OF finite-outgoing-edges*] *distinct-es*
by (*metis distinct-card*)
have *length* (*edge-relaxation-pairs-between d U L H*) ≤ *length ?es*
unfolding *edge-relaxation-pairs-between-def* **by** *simp*
then show *?thesis*
using *length-es* **by** *simp*
qed

lemma *label-pairs-between-length-le-card*:
assumes *finite-S*: *finite S*
shows *length* (*label-pairs-between d S L H*) ≤ *card S*
proof −
let *?xs* = *partition-key-order* (*label-partition-view d S*)
have *set-xs*: *set ?xs* = *S*
using *partition-key-order-properties*(*1*)[*of label-partition-view d S*]
finite-S **by** *simp*
have *distinct-xs*: *distinct ?xs*
using *partition-key-order-properties*(*2*)[*of label-partition-view d S*]
finite-S **by** *simp*
have *length-xs*: *length ?xs* = *card S*
using *set-xs distinct-xs* **by** (*metis distinct-card*)
have *length* (*label-pairs-between d S L H*) ≤ *length ?xs*
unfolding *label-pairs-between-def* **by** *simp*
then show *?thesis*
using *length-xs* **by** *simp*
qed

theorem *pull-separates-batch-prepend-for-relaxation-pairs*:
assumes *pull*: *pull-separates D M B S-pull beta D′*

shows *batch-prepend-admissible D′*
(*edge-relaxation-pairs-between d U L beta* @ *label-pairs-between d S L beta*)
proof (*rule pull-separates-batch-prepend-admissible*[*OF pull*])
fix *x b*
assume (*x*, *b*) ∈ *set*
(*edge-relaxation-pairs-between d U L beta* @ *label-pairs-between d S L beta*)
then show *b* ≤ *beta*
using *edge-relaxation-pairs-between-value-le-high*
label-pairs-between-value-le-high **by** *auto*
qed

The two batch constructors separate the sources of new partition entries. *edge-relaxation-pairs-between* enumerates outgoing child-tree edges whose relaxed values lie in the current open range, while *label-pairs-between* keeps old source labels that also lie in that range. The length lemmas above connect these lists to the quantities charged later: outgoing edges for the first list and source cardinality for the second.

The admissibility theorem *pull-separates ?D ?M ?B ?S-pull ?beta ?D′* ⟹ *batch-prepend-admissible ?D′* (*edge-relaxation-pairs-between ?d ?U ?L ?beta* @ *label-pairs-between ?d ?S ?L ?beta*) is the key local invariant for pushing the batch into the residual partition. It says that every generated value is still below the bound exposed by the pull, so the residual ordered partition can accept the batch without invalidating the next pull.

lemma *range-tree-same-empty* [*simp*]:
range-tree S a (*Fin a*) = {}
unfolding *range-tree-def* **by** *auto*

The relation below is the first operational presentation of the parent loop. It abstracts away the concrete partition representation, but it records the same sequence of events as the algorithm: stop when the lower bound has already produced enough completed vertices, otherwise recurse on the split prefix, preserve completion through the parent refresh, and continue from the child output bound.

The lists *betas*, *bs*, and *Us* are not auxiliary clutter. They are the trace that later cost proofs consume. The child output bounds *bs* determine adjacent range slices, and each stored set in *Us* is identified with the corresponding *range-tree*. Proving this trace property once keeps the later runtime arguments independent of the details of the recursive correctness proof.

inductive *operational-partition-loop* **where**
Done:
bound-le (*Fin a*) *B* ⟹
complete-on d′ (*bound-tree P* (*Fin a*)) ⟹
operational-partition-loop k cap l d P B d′ a [] [] (*Fin a*)
[*range-tree P a* (*Fin a*)]
(*bound-tree P* (*Fin a*) ∪ ⋃(*set* [*range-tree P a* (*Fin a*)]))
| *Step*:
below-bound beta B ⟹

$a \leq b \Longrightarrow$
complete-on d' (*bound-tree* P (*Fin* a)) $\Longrightarrow$
concrete-bmssp k *cap* l d (*split-below* d P *beta*) (*Fin beta*)
d-child (*Fin* b) *U-child* $\Longrightarrow$
complete-preserved d-child d' *U-child* $\Longrightarrow$
operational-partition-loop k *cap* l d P B d' b *betas bs* B' *Us-tail U-tail* $\Longrightarrow$
operational-partition-loop k *cap* l d P B d' a (*beta* # *betas*) (b # *bs*) B'
(*range-tree* P a (*Fin* b) # *Us-tail*)
(*bound-tree* P (*Fin* a) $\cup \bigcup$ (*set* (*range-tree* P a (*Fin* b) # *Us-tail*)))

theorem *operational-partition-loop-trace*:
assumes *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*
and *loop*: *operational-partition-loop k cap l d P B d' a betas bs B' Us U*
shows *concrete-partition-loop-trace P B a bs d' B' Us U*
using *loop sound pre P-reaches*
proof (*induction*)
case (*Done a B d' P k cap l d*)
then show *?case*
unfolding *concrete-partition-loop-trace-def complete-on-def* **by** *simp*
next
case (*Step beta B a b d' P k cap l d d-child U-child betas bs B' Us-tail U-tail*)
have *beta-le*: *bound-le* (*Fin beta*) *B*
using *Step.hyps*(*1*) **by** (*cases B*) *auto*
have *pre-beta*: *bmssp-pre-full d P* (*Fin beta*)
using *bmssp-pre-full-bound-mono*[*OF Step.prems*(*2*) *beta-le*] **.**
have *child-pre*: *bmssp-pre-full d* (*split-below d P beta*) (*Fin beta*)
using *pull-minimum-pre-for-lower-split*[*OF Step.prems*(*2*) *Step.hyps*(*1*)] **.**
have *child-reaches*:
$\bigwedge x.\ x \in$ *split-below d P beta* $\Longrightarrow$ *reachable s x*
using *Step.prems*(*3*) **unfolding** *split-below-def* **by** *blast*
have *child-post*:
bmssp-post-full d (*split-below d P beta*) (*Fin beta*)
d-child (*Fin* b) *U-child*
by (*rule concrete-bmssp-correct*[*OF Step.prems*(*1*) *child-pre child-reaches Step.hyps*(*4*)])
have *cover-beta*: *complete-tree-cover d P* (*Fin beta*)
using *bmssp-pre-full-complete-tree-cover*[*OF pre-beta*] **.**
have *U-child-tree*: *U-child* = *bound-tree P* (*Fin* b)
using *pull-recursive-post-lifts-bound-tree*[*OF cover-beta child-post*] **.**
have *child-complete*: *complete-on d-child U-child*
using *child-post* **unfolding** *bmssp-post-full-def* **by** *blast*
have *child-complete-final*: *complete-on d' U-child*
using *Step.hyps*(*5*) *child-complete* **unfolding** *complete-preserved-def* **by** *blast*
have *head-complete*: *complete-on d'* (*range-tree P a* (*Fin* b))
proof −
have *range-tree P a* (*Fin* b) $\subseteq$ *U-child*
using *U-child-tree range-tree-subset-bound-tree*[*of P a Fin b*] **by** *simp*

```
    then show ?thesis
      using complete-on-subset[OF child-complete-final] by blast
  qed
  have tail-trace:
    concrete-partition-loop-trace P B b bs d' B' Us-tail U-tail
    using Step.IH[OF Step.prems(1) Step.prems(2) Step.prems(3)] .
  have tail-le: bound-le B' B
    and tail-mono: nondecreasing-from b bs
    and tail-bounds: bounds-le B' (b # bs)
    and tail-children:
      list-all2 (λU X. U = X ∧ complete-on d' U) Us-tail
        (range-tree-chain-list P b bs B')
    using tail-trace unfolding concrete-partition-loop-trace-def by blast+
  have a-bound: bound-le (Fin a) B'
  proof −
    have bound-le (Fin b) B'
      using tail-bounds by simp
    then show ?thesis
      using Step.hyps(2) by (cases B') auto
  qed
  show ?case
    unfolding concrete-partition-loop-trace-def
    using tail-le Step.hyps(2) tail-mono tail-bounds a-bound
      Step.hyps(3) head-complete tail-children
    by auto
qed

definition operational-capped-bmssp-step where
  operational-capped-bmssp-step k cap l d S B a betas bs d' B' Us U ⟷
    (let d-fp = find-pivots-label-capped k cap d S B;
         P = find-pivots-pivots-capped k cap d S B;
         W = {v ∈ bound-tree S B'. d-fp v = dist s v}
     in ∃ U-loop.
         operational-partition-loop k cap l d-fp P B d' a betas bs B' Us U-loop ∧
         complete-on d' W ∧
         U = U-loop ∪ W)

theorem operational-capped-bmssp-step-correct:
  assumes sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and step: operational-capped-bmssp-step k cap l d S B a betas bs d' B' Us U
  shows bmssp-post-full d S B d' B' U
proof −
  let ?d-fp = find-pivots-label-capped k cap d S B
  let ?P = find-pivots-pivots-capped k cap d S B
  let ?W = {v ∈ bound-tree S B'. ?d-fp v = dist s v}
  obtain U-loop where
    loop: operational-partition-loop k cap l ?d-fp ?P B d' a betas bs B' Us U-loop
```

```
    and compW: complete-on d' ?W
    and U: U = U-loop ∪ ?W
   using step unfolding operational-capped-bmssp-step-def by (auto simp: Let-def)
  have sound-fp: sound-label ?d-fp
   unfolding find-pivots-label-capped-def
   by (rule fp-iter-capped-label-sound[OF sound S-reaches])
  have pivot-pre: bmssp-pre-full ?d-fp ?P B
   using find-pivots-capped-establishes-pivot-pre-concrete[OF sound pre S-reaches] .
  have P-subset-S: ?P ⊆ S
   unfolding find-pivots-pivots-capped-def by auto
  have P-reaches: ⋀x. x ∈ ?P ⟹ reachable s x
   using P-subset-S S-reaches by blast
  have trace: concrete-partition-loop-trace ?P B a bs d' B' Us U-loop
   by (rule operational-partition-loop-trace[OF sound-fp pivot-pre P-reaches loop])
  have concrete-step:
   concrete-capped-bmssp-step k cap d S B a bs d' B' Us U
   unfolding concrete-capped-bmssp-step-def
   using trace compW U by (auto simp: Let-def)
  show ?thesis
   by (rule concrete-capped-bmssp-step-correct
     [OF sound pre S-reaches concrete-step])
qed

inductive operational-bmssp where
  Base:
   S = {x} ⟹
    operational-bmssp k cap 0 d S B
     (λv. if v ∈ base-case-vertices k x B then dist s v else d v)
     (base-case-bound k x B)
     (base-case-vertices k x B)
| Step:
   operational-capped-bmssp-step k cap l d S B a betas bs d' B' Us U ⟹
    operational-bmssp k cap (Suc l) d S B d' B' U

theorem operational-bmssp-correct:
  assumes sound: sound-label d
   and pre: bmssp-pre-full d S B
   and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
   and run: operational-bmssp k cap l d S B d' B' U
  shows bmssp-post-full d S B d' B' U
  using run sound pre S-reaches
proof (induction arbitrary: rule: operational-bmssp.induct)
  case (Base S x k cap d B)
  have post:
   bmssp-post d S B
     (λv. if v ∈ base-case-vertices k x B then dist s v else d v)
     (base-case-bound k x B)
     (base-case-vertices k x B)
   using base-case-result-bmssp-post[OF Base.hyps, where k = k and B = B and
```

```
d = d]
    unfolding base-case-result-def by simp
  then show ?case
    by (rule bmssp-post-imp-post-full)
next
  case (Step k cap l d S B a betas bs d' B' Us U)
  show ?case
    by (rule operational-capped-bmssp-step-correct
      [OF Step.prems(1) Step.prems(2) Step.prems(3) Step.hyps])
qed

theorem finite-initial-label-operational-top-level-correct:
  assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
    and run: operational-bmssp k cap l finite-initial-label {s} Infinity d' Infinity U
  shows sssp-correct d'
proof −
  have pre: bmssp-pre-full finite-initial-label {s} Infinity
    using all-reachable finite-initial-label-source-complete
    by (rule top-bmssp-pre-full)
  have sound: sound-label finite-initial-label
    using finite-initial-label-sound[OF all-reachable] .
  have S-reaches: ⋀x. x ∈ {s} ⟹ reachable s x
    using all-reachable source-in-V by blast
  have post-full:
    bmssp-post-full finite-initial-label {s} Infinity d' Infinity U
    by (rule operational-bmssp-correct[OF sound pre S-reaches run])
  have post: bmssp-post finite-initial-label {s} Infinity d' Infinity U
    using bmssp-post-full-imp-post[OF post-full] .
  then have U-V: U = V
    using bound-tree-source-infinity[OF all-reachable]
    unfolding bmssp-post-def by auto
  have complete: complete-on d' U
    using post unfolding bmssp-post-def by auto
  then show ?thesis
    using U-V unfolding complete-on-def sssp-correct-def by auto
qed
```

The plain operational relation still calls *concrete-bmssp* in each loop step. The full relation below unfolds that recursive call as another inductive branch. This mutual presentation is more verbose, but it gives the cost layer a single derivation tree whose nodes are either BMSSP calls or partition-loop iterations.

The theorem ⟦⋀v. v ∈ V ⟹ *reachable s v*; *operational-bmssp ?k ?cap ?l finite-initial-label* {*s*} *Infinity ?d' Infinity ?U*⟧ ⟹ *sssp-correct ?d'* remains the semantic endpoint for this layer: starting from the finite initial label at the source and an infinite top-level bound, any operational run computes a full single-source shortest-path labelling. The later relations refine this one by adding costs and range synchronisation; they do not change this

correctness statement.

inductive *full-operational-partition-loop* **and** *full-operational-bmssp* **where**
Full-Loop-Done:
bound-le (*Fin a*) *B* ⟹
complete-on d′ (*bound-tree P* (*Fin a*)) ⟹
full-operational-partition-loop k cap l d P B d′ a [] [] (*Fin a*)
[*range-tree P a* (*Fin a*)]
(*bound-tree P* (*Fin a*) ∪ ⋃(*set* [*range-tree P a* (*Fin a*)]))
| *Full-Loop-Step*:
below-bound beta B ⟹
a ≤ *b* ⟹
complete-on d′ (*bound-tree P* (*Fin a*)) ⟹
full-operational-bmssp k cap l d (*split-below d P beta*) (*Fin beta*)
d-child (*Fin b*) *U-child* ⟹
complete-preserved d-child d′ U-child ⟹
full-operational-partition-loop k cap l d P B d′ b betas bs B′ Us-tail U-tail ⟹
full-operational-partition-loop k cap l d P B d′ a (*beta* # *betas*) (*b* # *bs*) *B′*
(*range-tree P a* (*Fin b*) # *Us-tail*)
(*bound-tree P* (*Fin a*) ∪ ⋃(*set* (*range-tree P a* (*Fin b*) # *Us-tail*)))
| *Full-Loop-Step-Pre*:
bound-le (*Fin beta*) *B* ⟹
bmssp-pre-full d (*split-below d P beta*) (*Fin beta*) ⟹
a ≤ *b* ⟹
complete-on d′ (*bound-tree P* (*Fin a*)) ⟹
full-operational-bmssp k cap l d (*split-below d P beta*) (*Fin beta*)
d-child (*Fin b*) *U-child* ⟹
complete-preserved d-child d′ U-child ⟹
full-operational-partition-loop k cap l d P B d′ b betas bs B′ Us-tail U-tail ⟹
full-operational-partition-loop k cap l d P B d′ a (*beta* # *betas*) (*b* # *bs*) *B′*
(*range-tree P a* (*Fin b*) # *Us-tail*)
(*bound-tree P* (*Fin a*) ∪ ⋃(*set* (*range-tree P a* (*Fin b*) # *Us-tail*)))
| *Full-Base*:
S = {*x*} ⟹
full-operational-bmssp k cap 0 d S B
(λ*v*. *if v* ∈ *base-case-vertices k x B then dist s v else d v*)
(*base-case-bound k x B*)
(*base-case-vertices k x B*)
| *Full-Step*:
full-operational-partition-loop k cap l
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d′ a betas bs B′ Us U-loop* ⟹
complete-on d′
{*v* ∈ *bound-tree S B′*. *find-pivots-label-capped k cap d S B v* = *dist s v*} ⟹
U = *U-loop* ∪
{*v* ∈ *bound-tree S B′*. *find-pivots-label-capped k cap d S B v* = *dist s v*} ⟹
full-operational-bmssp k cap (*Suc l*) *d S B d′ B′ U*

theorem *full-operational-partition-loop-trace*:
full-operational-partition-loop k cap l d P B d′ a betas bs B′ Us U ⟹

sound-label d ⟹
bmssp-pre-full d P B ⟹
(⋀*x*. *x* ∈ *P* ⟹ *reachable s x*) ⟹
concrete-partition-loop-trace P B a bs d′ B′ Us U
and *full-operational-bmssp-correct*:
full-operational-bmssp k cap l d S B d′ B′ U ⟹
sound-label d ⟹
bmssp-pre-full d S B ⟹
(⋀*x*. *x* ∈ *S* ⟹ *reachable s x*) ⟹
bmssp-post-full d S B d′ B′ U
proof (*induction rule*: *full-operational-partition-loop-full-operational-bmssp.inducts*)
case (*Full-Loop-Done a B d′ P k cap l d*)
then show *?case*
unfolding *concrete-partition-loop-trace-def complete-on-def* **by** *simp*
next
case (*Full-Loop-Step beta B a b d′ P k cap l d d-child U-child betas bs B′ Us-tail U-tail*)
have *beta-le*: *bound-le* (*Fin beta*) *B*
using *Full-Loop-Step*(*1*) **by** (*cases B*) *auto*
have *pre-beta*: *bmssp-pre-full d P* (*Fin beta*)
using *bmssp-pre-full-bound-mono*[*OF Full-Loop-Step.prems*(*2*) *beta-le*] **.**
have *child-pre*: *bmssp-pre-full d* (*split-below d P beta*) (*Fin beta*)
using *pull-minimum-pre-for-lower-split*
[*OF Full-Loop-Step.prems*(*2*) *Full-Loop-Step*(*1*)] **.**
have *child-reaches*:
⋀*x*. *x* ∈ *split-below d P beta* ⟹ *reachable s x*
using *Full-Loop-Step.prems*(*3*) **unfolding** *split-below-def* **by** *blast*
have *child-post*:
bmssp-post-full d (*split-below d P beta*) (*Fin beta*)
d-child (*Fin b*) *U-child*
using *Full-Loop-Step.IH Full-Loop-Step.prems*(*1*) *child-pre child-reaches*
by *blast*
have *cover-beta*: *complete-tree-cover d P* (*Fin beta*)
using *bmssp-pre-full-complete-tree-cover*[*OF pre-beta*] **.**
have *U-child-tree*: *U-child* = *bound-tree P* (*Fin b*)
using *pull-recursive-post-lifts-bound-tree*[*OF cover-beta child-post*] **.**
have *child-complete*: *complete-on d-child U-child*
using *child-post* **unfolding** *bmssp-post-full-def* **by** *blast*
have *child-complete-final*: *complete-on d′ U-child*
using *Full-Loop-Step child-complete*
unfolding *complete-preserved-def* **by** *blast*
have *head-complete*: *complete-on d′* (*range-tree P a* (*Fin b*))
proof −
have *range-tree P a* (*Fin b*) ⊆ *U-child*
using *U-child-tree range-tree-subset-bound-tree*[*of P a Fin b*] **by** *simp*
then show *?thesis*
using *complete-on-subset*[*OF child-complete-final*] **by** *blast*
qed
have *tail-trace*:

concrete-partition-loop-trace P B b bs d′ B′ Us-tail U-tail
using *Full-Loop-Step.IH Full-Loop-Step.prems* **by** *blast*
have *tail-le*: *bound-le B′ B*
and *tail-mono*: *nondecreasing-from b bs*
and *tail-bounds*: *bounds-le B′ (b # bs)*
and *tail-children*:
list-all2 (λU X. U = X ∧ complete-on d′ U) Us-tail
(range-tree-chain-list P b bs B′)
using *tail-trace* **unfolding** *concrete-partition-loop-trace-def* **by** *blast+*
have *a-bound*: *bound-le (Fin a) B′*
proof −
have *bound-le (Fin b) B′*
using *tail-bounds* **by** *simp*
then show *?thesis*
using *Full-Loop-Step(2)* **by** *(cases B′) auto*
qed
show *?case*
unfolding *concrete-partition-loop-trace-def*
using *tail-le Full-Loop-Step(2) tail-mono tail-bounds a-bound*
Full-Loop-Step(3) head-complete tail-children
by *auto*
next
case *(Full-Loop-Step-Pre beta B d P a b d′ k cap l d-child U-child betas bs B′ Us-tail U-tail)*
have *beta-le*: *bound-le (Fin beta) B*
using *Full-Loop-Step-Pre* **by** *blast*
have *a-le-b*: *a ≤ b*
using *Full-Loop-Step-Pre* **by** *blast*
have *lower-complete*: *complete-on d′ (bound-tree P (Fin a))*
using *Full-Loop-Step-Pre* **by** *blast*
have *pre-beta*: *bmssp-pre-full d P (Fin beta)*
using *bmssp-pre-full-bound-mono*
[OF Full-Loop-Step-Pre.prems(2) beta-le] **.**
have *child-pre*: *bmssp-pre-full d (split-below d P beta) (Fin beta)*
using *Full-Loop-Step-Pre* **by** *blast*
have *child-reaches*:
⋀x. x ∈ split-below d P beta ⟹ reachable s x
using *Full-Loop-Step-Pre.prems(3)* **unfolding** *split-below-def* **by** *blast*
have *child-post*:
bmssp-post-full d (split-below d P beta) (Fin beta)
d-child (Fin b) U-child
using *Full-Loop-Step-Pre.IH Full-Loop-Step-Pre.prems(1) child-pre child-reaches*
by *blast*
have *cover-beta*: *complete-tree-cover d P (Fin beta)*
using *bmssp-pre-full-complete-tree-cover[OF pre-beta]* **.**
have *U-child-tree*: *U-child = bound-tree P (Fin b)*
using *pull-recursive-post-lifts-bound-tree[OF cover-beta child-post]* **.**
have *child-complete*: *complete-on d-child U-child*
using *child-post* **unfolding** *bmssp-post-full-def* **by** *blast*

```
  have child-complete-final: complete-on d' U-child
    using Full-Loop-Step-Pre child-complete
    unfolding complete-preserved-def by blast
  have head-complete: complete-on d' (range-tree P a (Fin b))
  proof −
    have range-tree P a (Fin b) ⊆ U-child
      using U-child-tree range-tree-subset-bound-tree[of P a Fin b] by simp
    then show ?thesis
      using complete-on-subset[OF child-complete-final] by blast
  qed
  have tail-trace:
    concrete-partition-loop-trace P B b bs d' B' Us-tail U-tail
    using Full-Loop-Step-Pre.IH Full-Loop-Step-Pre.prems by blast
  have tail-le: bound-le B' B
    and tail-mono: nondecreasing-from b bs
    and tail-bounds: bounds-le B' (b # bs)
    and tail-children:
      list-all2 (λU X. U = X ∧ complete-on d' U) Us-tail
        (range-tree-chain-list P b bs B')
    using tail-trace unfolding concrete-partition-loop-trace-def by blast+
  have a-bound: bound-le (Fin a) B'
  proof −
    have bound-le (Fin b) B'
      using tail-bounds by simp
    then show ?thesis
      using a-le-b by (cases B') auto
  qed
  show ?case
    unfolding concrete-partition-loop-trace-def
    using tail-le a-le-b tail-mono tail-bounds a-bound
      lower-complete head-complete tail-children
    by auto
next
  case (Full-Base S x k cap d B)
  have post:
    bmssp-post d S B
      (λv. if v ∈ base-case-vertices k x B then dist s v else d v)
      (base-case-bound k x B)
      (base-case-vertices k x B)
    using base-case-result-bmssp-post[OF Full-Base.hyps, where k = k and B = B
and d = d]
    unfolding base-case-result-def by simp
  then show ?case
    by (rule bmssp-post-imp-post-full)
next
  case (Full-Step k cap l d S B d' a betas bs B' Us U-loop U)
  let ?d-fp = find-pivots-label-capped k cap d S B
  let ?P = find-pivots-pivots-capped k cap d S B
  let ?W = {v ∈ bound-tree S B'. ?d-fp v = dist s v}
```

have *sound-fp*: *sound-label ?d-fp*
unfolding *find-pivots-label-capped-def*
by (*rule fp-iter-capped-label-sound*[*OF Full-Step.prems*(*1*) *Full-Step.prems*(*3*)])
have *pivot-pre*: *bmssp-pre-full ?d-fp ?P B*
using *find-pivots-capped-establishes-pivot-pre-concrete*
[*OF Full-Step.prems*(*1*) *Full-Step.prems*(*2*) *Full-Step.prems*(*3*)] **.**
have *P-subset-S*: *?P* $\subseteq$ *S*
unfolding *find-pivots-pivots-capped-def* **by** *auto*
have *P-reaches*: $\bigwedge x.\ x \in ?P \Longrightarrow$ *reachable s x*
using *P-subset-S Full-Step.prems*(*3*) **by** *blast*
have *trace*: *concrete-partition-loop-trace ?P B a bs d' B' Us U-loop*
using *Full-Step.IH sound-fp pivot-pre P-reaches* **by** *blast*
have *concrete-step*:
concrete-capped-bmssp-step k cap d S B a bs d' B' Us U
unfolding *concrete-capped-bmssp-step-def*
using *trace Full-Step* **by** (*auto simp*: *Let-def*)
show *?case*
by (*rule concrete-capped-bmssp-step-correct*
[*OF Full-Step.prems*(*1*) *Full-Step.prems*(*2*) *Full-Step.prems*(*3*) *concrete-step*])
qed

theorem *finite-initial-label-full-operational-top-level-correct*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *run*: *full-operational-bmssp k cap l finite-initial-label* {*s*} *Infinity d' Infinity U*
shows *sssp-correct d'*
proof −
have *pre*: *bmssp-pre-full finite-initial-label* {*s*} *Infinity*
using *all-reachable finite-initial-label-source-complete*
by (*rule top-bmssp-pre-full*)
have *sound*: *sound-label finite-initial-label*
using *finite-initial-label-sound*[*OF all-reachable*] **.**
have *S-reaches*: $\bigwedge x.\ x \in \{s\} \Longrightarrow$ *reachable s x*
using *all-reachable source-in-V* **by** *blast*
have *post-full*:
bmssp-post-full finite-initial-label {*s*} *Infinity d' Infinity U*
by (*rule full-operational-bmssp-correct*[*OF run sound pre S-reaches*])
have *post*: *bmssp-post finite-initial-label* {*s*} *Infinity d' Infinity U*
using *bmssp-post-full-imp-post*[*OF post-full*] **.**
then have *U-V*: *U* = *V*
using *bound-tree-source-infinity*[*OF all-reachable*]
unfolding *bmssp-post-def* **by** *auto*
have *complete*: *complete-on d' U*
using *post* **unfolding** *bmssp-post-def* **by** *auto*
then show *?thesis*
using *U-V* **unfolding** *complete-on-def sssp-correct-def* **by** *auto*
qed

inductive *full-operational-partition-loop-state* **where**

```
  State-Done:
    keys-of D = {} ⟹
     bound-le (Fin a) B ⟹
     complete-on d' (bound-tree P (Fin a)) ⟹
     full-operational-partition-loop-state M t k cap l d P B d' D a [] [] (Fin a)
       [range-tree P a (Fin a)]
       (bound-tree P (Fin a) ∪ ⋃(set [range-tree P a (Fin a)])) 0
| State-Step:
    pull-separates D M Bmax S-pull beta D-pull ⟹
     bound-le (Fin beta) B ⟹
     bmssp-pre-full d (split-below d P beta) (Fin beta) ⟹
     S-pull = split-below d P beta ⟹
     a ≤ b ⟹
     complete-on d' (bound-tree P (Fin a)) ⟹
     full-operational-bmssp k cap l d S-pull (Fin beta) d-child (Fin b) U-child ⟹
     complete-preserved d-child d' U-child ⟹
     batch =
       edge-relaxation-pairs-between d-child U-child b beta @
       label-pairs-between d S-pull b beta ⟹
     D-next = batch-min-update D-pull batch ⟹
     partition-pull-cost-bound c-pull S-pull ⟹
     partition-batch-cost-bound c-batch t batch ⟹
     full-operational-partition-loop-state M t k cap l d P B d' D-next b betas bs B'
       Us-tail U-tail c-tail ⟹
     c = c-pull + c-batch + c-child + c-tail ⟹
     full-operational-partition-loop-state M t k cap l d P B d' D a (beta # betas) (b
# bs) B'
       (range-tree P a (Fin b) # Us-tail)
       (bound-tree P (Fin a) ∪ ⋃(set (range-tree P a (Fin b) # Us-tail))) c

lemma full-operational-partition-loop-state-lengths:
  assumes run:
    full-operational-partition-loop-state M t k cap l d P B d' D a
      betas bs B' Us U c
  shows length betas = length bs
    and length Us = Suc (length bs)
proof −
  have both: length betas = length bs ∧ length Us = Suc (length bs)
    using run
  proof (induction)
    case State-Done
    then show ?case by simp
  next
    case (State-Step D M Bmax S-pull beta D-pull B d P a b d' k cap l
        d-child U-child batch D-next c-pull c-batch t betas bs B'
        Us-tail U-tail c-tail c-child c)
    then show ?case by simp
  qed
  show length betas = length bs
```

using *both* **by** *blast*
show *length Us = Suc* (*length bs*)
using *both* **by** *blast*
qed

theorem *full-operational-partition-loop-state-refines*:
assumes *run*:
full-operational-partition-loop-state M t k cap l d P B d' D a betas bs B' Us U c
shows *full-operational-partition-loop k cap l d P B d' a betas bs B' Us U*
using *run*
proof (*induction*)
case (*State-Done D a B d' P M t k cap l d*)
show *?case*
by (*rule Full-Loop-Done*[*OF State-Done.hyps*(*2*,*3*)])
next
case (*State-Step D M Bmax S-pull beta D-pull B d P a b d' k cap l d-child U-child batch D-next c-pull c-batch t betas bs B' Us-tail U-tail c-tail c-step c-total*)
have *child-run*:
full-operational-bmssp k cap l d (*split-below d P beta*) (*Fin beta*)
d-child (*Fin b*) *U-child*
using *State-Step* **by** *blast*
show *?case*
by (*rule Full-Loop-Step-Pre*)
(*use State-Step child-run* **in** *auto*)
qed

theorem *full-operational-partition-loop-state-cost-bound*:
assumes *run*:
full-operational-partition-loop-state M t k cap l d P B d' D a betas bs B' Us U c
shows ∃ *child-costs pulls batches*.
c ≤ *sum-list child-costs* + *sum-list* (*map* (λ*S*. *card* (*S* :: *'a set*)) *pulls*) +
t ∗ *sum-list* (*map* (λ*batch*. *length* (*batch* :: (*'a* × *real*) *list*)) *batches*)
using *run*
proof (*induction*)
case (*State-Done D a B d' P M t k cap l d*)
show *?case*
by (*intro exI*[*of* - []] *exI*[*of* - []] *exI*[*of* - []]) *simp*
next
case (*State-Step D M Bmax S-pull beta D-pull B d P a b d' k cap l d-child U-child batch D-next c-pull c-batch t betas bs B' Us-tail U-tail c-tail c-step c-total*)
have *tail-ex*: ∃ *child-costs pulls batches*.
c-tail ≤ *sum-list child-costs* + *sum-list* (*map* (λ*S*. *card* (*S* :: *'a set*)) *pulls*) +
t ∗ *sum-list* (*map* (λ*batch*. *length* (*batch* :: (*'a* × *real*) *list*)) *batches*)
using *State-Step.IH* **by** *assumption*
obtain *child-costs* :: *nat list* **and** *pulls* :: *'a set list*
and *batches* :: (*'a* × *real*) *list list* **where** *tail*:
c-tail ≤ *sum-list child-costs* + *sum-list* (*map card pulls*) +
t ∗ *sum-list* (*map length batches*)
using *tail-ex* **by** *blast*

```
  have pull-bound: c-pull ≤ card S-pull
    using State-Step unfolding partition-pull-cost-bound-def by blast
  have batch-bound: c-batch ≤ t * length batch
    using State-Step unfolding partition-batch-cost-bound-def by blast
  have c-eq: c-step = c-pull + c-batch + c-total + c-tail
    using State-Step by blast
  let ?child-costs = c-total # child-costs
  let ?pulls = S-pull # pulls
  let ?batches = batch # batches
  have c-step ≤ sum-list ?child-costs + sum-list (map card ?pulls) +
      t * sum-list (map length ?batches)
  proof −
    have c-pull + c-batch + c-total + c-tail ≤
        card S-pull + t * length batch + c-total +
        (sum-list child-costs + sum-list (map card pulls) +
          t * sum-list (map length batches))
      using pull-bound batch-bound tail by simp
    also have ... =
        sum-list ?child-costs + sum-list (map card ?pulls) +
        t * sum-list (map length ?batches)
      by (simp add: algebra-simps)
    finally show ?thesis
      using c-eq by simp
  qed
  then show ?case
    by blast
qed

theorem full-operational-partition-loop-state-trace-and-cost:
  assumes run:
    full-operational-partition-loop-state M t k cap l d P B d' D a betas bs B' Us U c
    and sound: sound-label d
    and pre: bmssp-pre-full d P B
    and P-reaches: ⋀x. x ∈ P ⟹ reachable s x
  obtains child-costs pulls batches where
    concrete-partition-loop-trace P B a bs d' B' Us U
    c ≤ sum-list child-costs + sum-list (map (λS. card (S :: 'a set)) pulls) +
      t * sum-list (map (λbatch. length (batch :: ('a × real) list)) batches)
proof −
  have full-loop:
    full-operational-partition-loop k cap l d P B d' a betas bs B' Us U
    by (rule full-operational-partition-loop-state-refines[OF run])
  have trace: concrete-partition-loop-trace P B a bs d' B' Us U
    by (rule full-operational-partition-loop-trace
      [OF full-loop sound pre P-reaches])
  obtain child-costs pulls batches where cost:
    c ≤ sum-list child-costs + sum-list (map (λS. card (S :: 'a set)) pulls) +
      t * sum-list (map (λbatch. length (batch :: ('a × real) list)) batches)
    using full-operational-partition-loop-state-cost-bound[OF run] by blast
```

```
  then show thesis
    using that trace by blast
qed

theorem full-operational-partition-loop-state-cost-bound-by-child-edges:
  assumes run:
    full-operational-partition-loop-state M t k cap l d P B d' D a betas bs B' Us U c
    and P-subset: P ⊆ V
  shows ∃ child-costs child-sets.
    length child-costs = length child-sets ∧
    c ≤ sum-list child-costs + M * length child-costs +
      t * (sum-list (map (λX. card (outgoing-edges X)) child-sets) +
        M * length child-costs)
  using run P-subset
proof (induction)
  case (State-Done D a B d' P M t k cap l d)
  show ?case
    by (intro exI[of - []] exI[of - []]) simp
next
  case (State-Step D M Bmax S-pull beta D-pull B d P a b d' k cap l
      d-child U-child batch D-next c-pull c-batch t betas bs B'
      Us-tail U-tail c-tail c-step c-total)
  have tail-ex: ∃ child-costs child-sets.
      length child-costs = length child-sets ∧
      c-tail ≤ sum-list child-costs + M * length child-costs +
        t * (sum-list (map (λX. card (outgoing-edges X)) child-sets) +
          M * length child-costs)
    using State-Step.IH State-Step.prems by blast
  obtain child-costs child-sets where len-tail:
      length child-costs = length child-sets
    and tail:
      c-tail ≤ sum-list child-costs + M * length child-costs +
        t * (sum-list (map (λX. card (outgoing-edges X)) child-sets) +
          M * length child-costs)
    using tail-ex by blast
  have finite-P: finite P
    using finite-subset[OF State-Step.prems(1) finite-V] .
  have S-pull-subset: S-pull ⊆ P
    using State-Step.hyps(4) unfolding split-below-def by blast
  have finite-S-pull: finite S-pull
    using finite-subset[OF S-pull-subset finite-P] .
  have card-pull: card S-pull ≤ M
    using pull-separates-card[OF State-Step.hyps(1)] .
  have pull-cost: c-pull ≤ M
    using State-Step.hyps(11) card-pull
    unfolding partition-pull-cost-bound-def by linarith
  have edge-len:
    length (edge-relaxation-pairs-between d-child U-child b beta) ≤
      card (outgoing-edges U-child)
```

```
    by (rule edge-relaxation-pairs-between-length-le-outgoing)
  have label-len:
    length (label-pairs-between d S-pull b beta) ≤ card S-pull
    by (rule label-pairs-between-length-le-card[OF finite-S-pull])
  have batch-len:
    length batch ≤ card (outgoing-edges U-child) + M
    using State-Step.hyps(9) edge-len label-len card-pull by simp
  have batch-cost:
    c-batch ≤ t * (card (outgoing-edges U-child) + M)
  proof −
    have c-batch ≤ t * length batch
      using State-Step.hyps(12)
      unfolding partition-batch-cost-bound-def by simp
    also have ... ≤ t * (card (outgoing-edges U-child) + M)
      using batch-len by simp
    finally show ?thesis .
  qed
  let ?child-costs = c-total # child-costs
  let ?child-sets = U-child # child-sets
  have len: length ?child-costs = length ?child-sets
    using len-tail by simp
  have cost:
    c-step ≤ sum-list ?child-costs + M * length ?child-costs +
      t * (sum-list (map (λX. card (outgoing-edges X)) ?child-sets) +
        M * length ?child-costs)
  proof −
    have c-eq: c-step = c-pull + c-batch + c-total + c-tail
      using State-Step.hyps(14) .
    have c-pull + c-batch + c-total + c-tail ≤
        M + t * (card (outgoing-edges U-child) + M) + c-total +
        (sum-list child-costs + M * length child-costs +
          t * (sum-list (map (λX. card (outgoing-edges X)) child-sets) +
            M * length child-costs))
      using pull-cost batch-cost tail by simp
    also have ... =
        sum-list ?child-costs + M * length ?child-costs +
        t * (sum-list (map (λX. card (outgoing-edges X)) ?child-sets) +
          M * length ?child-costs)
      by (simp add: algebra-simps)
    finally show ?thesis
      using c-eq by simp
  qed
  show ?case
    using len cost by blast
qed

theorem full-operational-partition-loop-state-trace-and-edge-cost:
  assumes run:
    full-operational-partition-loop-state M t k cap l d P B d' D a betas bs B' Us U c
```

```
    and sound: sound-label d
    and pre: bmssp-pre-full d P B
    and P-reaches: ⋀x. x ∈ P ⟹ reachable s x
  obtains child-costs child-sets where
    concrete-partition-loop-trace P B a bs d' B' Us U
    length child-costs = length child-sets
    c ≤ sum-list child-costs + M * length child-costs +
      t * (sum-list (map (λX. card (outgoing-edges X)) child-sets) +
        M * length child-costs)
proof −
  have P-subset: P ⊆ V
    using pre unfolding bmssp-pre-full-def by blast
  have full-loop:
    full-operational-partition-loop k cap l d P B d' a betas bs B' Us U
    by (rule full-operational-partition-loop-state-refines[OF run])
  have trace: concrete-partition-loop-trace P B a bs d' B' Us U
    by (rule full-operational-partition-loop-trace
      [OF full-loop sound pre P-reaches])
  obtain child-costs child-sets where len:
      length child-costs = length child-sets
    and cost:
      c ≤ sum-list child-costs + M * length child-costs +
        t * (sum-list (map (λX. card (outgoing-edges X)) child-sets) +
          M * length child-costs)
    using full-operational-partition-loop-state-cost-bound-by-child-edges
      [OF run P-subset] by blast
  then show thesis
    using that trace by blast
qed

theorem full-operational-partition-loop-state-cost-bound-by-child-edges-complete:
  assumes run:
    full-operational-partition-loop-state M t k cap l d P B d' D a betas bs B' Us U c
    and sound: sound-label d
    and pre: bmssp-pre-full d P B
    and P-reaches: ⋀x. x ∈ P ⟹ reachable s x
  shows ∃ child-costs child-sets.
    length child-costs = length child-sets ∧
    (∀ X∈set child-sets. X ⊆ V) ∧
    c ≤ sum-list child-costs + M * length child-costs +
      t * (sum-list (map (λX. card (outgoing-edges X)) child-sets) +
        M * length child-costs)
  using run sound pre P-reaches
proof (induction)
  case (State-Done D a B d' P M t k cap l d)
  show ?case
    by (intro exI[of - []] exI[of - []]) simp
next
  case (State-Step D M Bmax S-pull beta D-pull B d P a b d' k cap l
```

d-child U-child batch D-next c-pull c-batch t betas bs B′
Us-tail U-tail c-tail c-step c-total)
have *tail-ex*: ∃ *child-costs child-sets*.
length child-costs = length child-sets ∧
(∀ *X*∈*set child-sets*. *X* ⊆ *V*) ∧
c-tail ≤ *sum-list child-costs* + *M* ∗ *length child-costs* +
t ∗ (*sum-list* (*map* (λ*X*. *card* (*outgoing-edges X*)) *child-sets*) +
M ∗ *length child-costs*)
using *State-Step.IH State-Step.prems* **by** *blast*
obtain *child-costs child-sets* **where** *len-tail*:
length child-costs = length child-sets
and *sets-tail*: ∀ *X*∈*set child-sets*. *X* ⊆ *V*
and *tail*:
c-tail ≤ *sum-list child-costs* + *M* ∗ *length child-costs* +
t ∗ (*sum-list* (*map* (λ*X*. *card* (*outgoing-edges X*)) *child-sets*) +
M ∗ *length child-costs*)
using *tail-ex* **by** *blast*
have *child-reaches*:
⋀*x*. *x* ∈ *S-pull* ⟹ *reachable s x*
using *State-Step.prems*(*3*) *State-Step.hyps*(*4*)
unfolding *split-below-def* **by** *blast*
have *child-pre*: *bmssp-pre-full d S-pull* (*Fin beta*)
using *State-Step.hyps*(*3*,*4*) **by** *simp*
have *child-post*:
bmssp-post-full d S-pull (*Fin beta*) *d-child* (*Fin b*) *U-child*
using *full-operational-bmssp-correct*
[*OF State-Step.hyps*(*7*) *State-Step.prems*(*1*) *child-pre child-reaches*] **.**
have *child-set*: *U-child* ⊆ *V*
using *child-post* **unfolding** *bmssp-post-full-def bound-tree-def* **by** *blast*
have *P-subset*: *P* ⊆ *V*
using *State-Step.prems*(*2*) **unfolding** *bmssp-pre-full-def* **by** *blast*
have *finite-P*: *finite P*
using *finite-subset*[*OF P-subset finite-V*] **.**
have *S-pull-subset*: *S-pull* ⊆ *P*
using *State-Step.hyps*(*4*) **unfolding** *split-below-def* **by** *blast*
have *finite-S-pull*: *finite S-pull*
using *finite-subset*[*OF S-pull-subset finite-P*] **.**
have *card-pull*: *card S-pull* ≤ *M*
using *pull-separates-card*[*OF State-Step.hyps*(*1*)] **.**
have *pull-cost*: *c-pull* ≤ *M*
using *State-Step.hyps*(*11*) *card-pull*
unfolding *partition-pull-cost-bound-def* **by** *linarith*
have *edge-len*:
length (*edge-relaxation-pairs-between d-child U-child b beta*) ≤
card (*outgoing-edges U-child*)
by (*rule edge-relaxation-pairs-between-length-le-outgoing*)
have *label-len*:
length (*label-pairs-between d S-pull b beta*) ≤ *card S-pull*
by (*rule label-pairs-between-length-le-card*[*OF finite-S-pull*])

```
  have batch-len:
    length batch ≤ card (outgoing-edges U-child) + M
    using State-Step.hyps(9) edge-len label-len card-pull by simp
  have batch-cost:
    c-batch ≤ t * (card (outgoing-edges U-child) + M)
  proof −
    have c-batch ≤ t * length batch
      using State-Step.hyps(12)
      unfolding partition-batch-cost-bound-def by simp
    also have ... ≤ t * (card (outgoing-edges U-child) + M)
      using batch-len by simp
    finally show ?thesis .
  qed
  let ?child-costs = c-total # child-costs
  let ?child-sets = U-child # child-sets
  have len: length ?child-costs = length ?child-sets
    using len-tail by simp
  have sets: ∀ X∈set ?child-sets. X ⊆ V
    using child-set sets-tail by simp
  have cost:
    c-step ≤ sum-list ?child-costs + M * length ?child-costs +
      t * (sum-list (map (λX. card (outgoing-edges X)) ?child-sets) +
        M * length ?child-costs)
  proof −
    have c-eq: c-step = c-pull + c-batch + c-total + c-tail
      using State-Step.hyps(14) .
    have c-pull + c-batch + c-total + c-tail ≤
        M + t * (card (outgoing-edges U-child) + M) + c-total +
        (sum-list child-costs + M * length child-costs +
          t * (sum-list (map (λX. card (outgoing-edges X)) child-sets) +
            M * length child-costs))
      using pull-cost batch-cost tail by simp
    also have ... =
        sum-list ?child-costs + M * length ?child-costs +
        t * (sum-list (map (λX. card (outgoing-edges X)) ?child-sets) +
          M * length ?child-costs)
      by (simp add: algebra-simps)
    finally show ?thesis
      using c-eq by simp
  qed
  show ?case
    using len sets cost by blast
qed

theorem full-operational-partition-loop-state-trace-and-complete-edge-cost:
  assumes run:
    full-operational-partition-loop-state M t k cap l d P B d' D a betas bs B' Us U c
    and sound: sound-label d
    and pre: bmssp-pre-full d P B
```

```
    and P-reaches: ⋀x. x ∈ P ⟹ reachable s x
  obtains child-costs child-sets where
    concrete-partition-loop-trace P B a bs d' B' Us U
    length child-costs = length child-sets
    ⋀X. X ∈ set child-sets ⟹ X ⊆ V
    c ≤ sum-list child-costs + M * length child-costs +
      t * (sum-list (map (λX. card (outgoing-edges X)) child-sets) +
        M * length child-costs)
proof −
  have full-loop:
    full-operational-partition-loop k cap l d P B d' a betas bs B' Us U
    by (rule full-operational-partition-loop-state-refines[OF run])
  have trace: concrete-partition-loop-trace P B a bs d' B' Us U
    by (rule full-operational-partition-loop-trace
      [OF full-loop sound pre P-reaches])
  obtain child-costs child-sets where len:
      length child-costs = length child-sets
    and sets: ∀ X∈set child-sets. X ⊆ V
    and cost:
      c ≤ sum-list child-costs + M * length child-costs +
        t * (sum-list (map (λX. card (outgoing-edges X)) child-sets) +
          M * length child-costs)
    using full-operational-partition-loop-state-cost-bound-by-child-edges-complete
      [OF run sound pre P-reaches] by blast
  then show thesis
    using that trace by blast
qed

end

end
theory BMSSP-Complexity
  imports HOL−Library.Landau-Symbols BMSSP-Operational-Pull
begin
```

28 Core Cost Accounting

This theory starts the checked complexity side of the development. Earlier theories prove that the recursive BMSSP algorithm returns the right vertices and labels. Here the same traces are interpreted as cost recurrences. The first part is deliberately arithmetic: logarithmic target functions, level widths, level capacities, and the Big-O envelopes that eventually become the SSSP headline.

The file then moves inside the graph locale and introduces the graph-size quantities used by the cost bounds. Vertex counts, edge counts, outgoing ranges, and range-tree chains are all finite combinatorial objects. The proofs establish that the lists produced by a partition loop are disjoint slices, so their

vertex and outgoing-edge costs can be summed without double counting.

The final part defines a costed operational relation for the abstract partition interface. It is intentionally not the bucketed implementation; it is the reusable recurrence layer that later theories refine. The bucketed proof plugs concrete local costs into these abstract cost expressions, while the top-level bounds instantiate the logarithmic parameter schedule.

definition *sssp-log-factor* **where**
sssp-log-factor n = (*ln* (*real n* + *2*)) *powr* (*2* / *3*)

definition *sssp-log-factor-one-third* **where**
sssp-log-factor-one-third n = (*ln* (*real n* + *2*)) *powr* (*1* / *3*)

definition *sssp-time-target* **where**
sssp-time-target m n = *real* (*m n*) ∗ *sssp-log-factor n*

definition *bmssp-level-width* :: *nat* ⇒ *nat* ⇒ *nat* **where**
bmssp-level-width t l = *2* ^ (*l* ∗ *t*)

definition *bmssp-level-cap* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *nat* **where**
bmssp-level-cap k t l = *k* ∗ *bmssp-level-width t l*

lemma *bmssp-level-width-pos* [*simp*]:
0 < *bmssp-level-width t l*
unfolding *bmssp-level-width-def* **by** *simp*

lemma *bmssp-level-cap-eq*:
bmssp-level-cap k t l = *k* ∗ *2* ^ (*l* ∗ *t*)
unfolding *bmssp-level-cap-def bmssp-level-width-def* **by** *simp*

lemma *bmssp-level-width-mono*:
assumes *l* ≤ *l'*
shows *bmssp-level-width t l* ≤ *bmssp-level-width t l'*
proof −
have *l* ∗ *t* ≤ *l'* ∗ *t*
using *assms* **by** *simp*
then show *?thesis*
unfolding *bmssp-level-width-def* **by** (*simp add*: *power-increasing*)
qed

lemma *bmssp-level-width-Suc*:
bmssp-level-width t (*Suc l*) = *bmssp-level-width t l* ∗ *2* ^ *t*
unfolding *bmssp-level-width-def* **by** (*simp add*: *power-add algebra-simps*)

lemma *bmssp-level-width-prev-times-two-le*:
assumes *0* < *t 0* < *l*
shows *2* ∗ *bmssp-level-width t* (*l* − *1*) ≤ *bmssp-level-width t l*
proof (*cases l*)
case *0*

```
  then show ?thesis
    using assms by simp
next
  case (Suc l')
  have two-le: (2::nat) ≤ 2 ^ t
  proof −
    have 2 ^ 1 ≤ (2::nat) ^ t
      by (rule power-increasing) (use assms(1) in simp-all)
    then show ?thesis
      by simp
  qed
  have 2 * bmssp-level-width t (l − 1) ≤
      (2 ^ t) * bmssp-level-width t (l − 1)
    by (rule mult-right-mono[OF two-le]) simp
  also have ... = bmssp-level-width t l
    unfolding Suc bmssp-level-width-Suc by (simp add: algebra-simps)
  finally show ?thesis .
qed

lemma bmssp-level-cap-mono:
  assumes l ≤ l'
  shows bmssp-level-cap k t l ≤ bmssp-level-cap k t l'
  unfolding bmssp-level-cap-def
  using bmssp-level-width-mono[OF assms, of t] by simp

lemma bmssp-level-cap-ge-level:
  k ≤ bmssp-level-cap k t l
  unfolding bmssp-level-cap-def
  using bmssp-level-width-pos[of t l]
  by (cases bmssp-level-width t l) simp-all

lemma bmssp-level-size-constraint-arithmetic:
  assumes t-pos: 0 < t
    and l-pos: 0 < l
    and before: acc < k * bmssp-level-width t l
    and child: child ≤ 4 * k * bmssp-level-width t (l − 1)
    and extra: extra ≤ k * bmssp-level-width t l
    and U-size: U-size ≤ acc + child + extra
  shows U-size ≤ 4 * k * bmssp-level-width t l
proof −
  have acc-le: acc ≤ k * bmssp-level-width t l
    using before by simp
  have child-le: child ≤ 2 * k * bmssp-level-width t l
  proof −
    have 4 * k * bmssp-level-width t (l − 1) ≤
        2 * k * bmssp-level-width t l
    proof −
      have 2 * bmssp-level-width t (l − 1) ≤ bmssp-level-width t l
        by (rule bmssp-level-width-prev-times-two-le[OF t-pos l-pos])
```

```
    then have 2 * k * (2 * bmssp-level-width t (l - 1)) ≤
        2 * k * bmssp-level-width t l
      by simp
    then show ?thesis
      by (simp add: algebra-simps)
  qed
  then show ?thesis
    using child by linarith
 qed
 have U-size ≤
    k * bmssp-level-width t l +
    2 * k * bmssp-level-width t l +
    k * bmssp-level-width t l
  using U-size acc-le child-le extra by linarith
 also have ... = 4 * k * bmssp-level-width t l
  by (simp add: algebra-simps)
 finally show ?thesis .
qed

lemma sssp-log-factor-pos [simp]:
 0 < sssp-log-factor n
proof -
 have 1 < real n + 2
  by simp
 then have 0 < ln (real n + 2)
  by (rule ln-gt-zero)
 then show ?thesis
  unfolding sssp-log-factor-def by simp
qed

lemma sssp-log-factor-one-third-pos [simp]:
 0 < sssp-log-factor-one-third n
proof -
 have 1 < real n + 2
  by simp
 then have 0 < ln (real n + 2)
  by (rule ln-gt-zero)
 then show ?thesis
  unfolding sssp-log-factor-one-third-def by simp
qed

lemma sssp-log-factor-mono:
 assumes n ≤ m
 shows sssp-log-factor n ≤ sssp-log-factor m
proof -
 have ln-le: ln (real n + 2) ≤ ln (real m + 2)
  by (rule ln-mono) (use assms in simp-all)
 have ln-nonneg: 0 ≤ ln (real n + 2)
  by simp
```

```
  show ?thesis
    unfolding sssp-log-factor-def
    by (rule powr-mono2) (use ln-nonneg ln-le in simp-all)
qed

lemma sssp-log-factor-one-third-mono:
  assumes n ≤ m
  shows sssp-log-factor-one-third n ≤ sssp-log-factor-one-third m
proof −
  have ln-le: ln (real n + 2) ≤ ln (real m + 2)
    by (rule ln-mono) (use assms in simp-all)
  have ln-nonneg: 0 ≤ ln (real n + 2)
    by simp
  show ?thesis
    unfolding sssp-log-factor-one-third-def
    by (rule powr-mono2) (use ln-nonneg ln-le in simp-all)
qed

lemma sssp-log-factor-square-arg-le:
  sssp-log-factor (n * n) ≤ 2 * sssp-log-factor n
proof −
  let ?a = ln (real (n * n) + 2)
  let ?b = ln (real n + 2)
  have arg-le: real (n * n) + 2 ≤ (real n + 2) ^ 2
    by (simp add: power2-eq-square algebra-simps)
  have arg-pos: 0 < real (n * n) + 2
  proof −
    have 0 ≤ real (n * n)
      by simp
    then show ?thesis
      by linarith
  qed
  have ln-le: ?a ≤ 2 * ?b
  proof −
    have ?a ≤ ln ((real n + 2) ^ 2)
      by (rule ln-mono) (use arg-le in simp, rule arg-pos)
    also have ... = 2 * ?b
      by (simp add: ln-realpow)
    finally show ?thesis .
  qed
  have a-nonneg: 0 ≤ ?a
    by simp
  have b-nonneg: 0 ≤ ?b
    by simp
  have powr-le: ?a powr (2 / 3) ≤ (2 * ?b) powr (2 / 3)
    by (rule powr-mono2) (use a-nonneg ln-le in simp-all)
  have two-powr-le: (2::real) powr (2 / 3) ≤ 2
  proof −
    have (2::real) powr (2 / 3) ≤ 2 powr 1
```

```
    by (rule powr-mono) simp-all
  then show ?thesis
    by simp
 qed
 have (2 * ?b) powr (2 / 3) =
    (2::real) powr (2 / 3) * (?b powr (2 / 3))
  by (simp add: powr-mult)
 also have ... ≤ 2 * (?b powr (2 / 3))
  by (rule mult-right-mono[OF two-powr-le]) (use b-nonneg in simp)
 finally have (2 * ?b) powr (2 / 3) ≤ 2 * (?b powr (2 / 3)) .
 then show ?thesis
  unfolding sssp-log-factor-def using powr-le by linarith
qed

lemma sssp-log-factor-one-third-square-arg-le:
 sssp-log-factor-one-third (n * n) ≤
  2 * sssp-log-factor-one-third n
proof −
 let ?a = ln (real (n * n) + 2)
 let ?b = ln (real n + 2)
 have arg-le: real (n * n) + 2 ≤ (real n + 2) ^ 2
  by (simp add: power2-eq-square algebra-simps)
 have arg-pos: 0 < real (n * n) + 2
 proof −
  have 0 ≤ real (n * n)
   by simp
  then show ?thesis
   by linarith
 qed
 have ln-le: ?a ≤ 2 * ?b
 proof −
  have ?a ≤ ln ((real n + 2) ^ 2)
   by (rule ln-mono) (use arg-le in simp, rule arg-pos)
  also have ... = 2 * ?b
   by (simp add: ln-realpow)
  finally show ?thesis .
 qed
 have a-nonneg: 0 ≤ ?a
  by simp
 have b-nonneg: 0 ≤ ?b
  by simp
 have powr-le: ?a powr (1 / 3) ≤ (2 * ?b) powr (1 / 3)
  by (rule powr-mono2) (use a-nonneg ln-le in simp-all)
 have two-powr-le: (2::real) powr (1 / 3) ≤ 2
 proof −
  have (2::real) powr (1 / 3) ≤ 2 powr 1
   by (rule powr-mono) simp-all
  then show ?thesis
   by simp
```

```
  qed
  have (2 * ?b) powr (1 / 3) =
      (2::real) powr (1 / 3) * (?b powr (1 / 3))
    by (simp add: powr-mult)
  also have ... ≤ 2 * (?b powr (1 / 3))
    by (rule mult-right-mono[OF two-powr-le]) (use b-nonneg in simp)
  finally have (2 * ?b) powr (1 / 3) ≤ 2 * (?b powr (1 / 3)) .
  then show ?thesis
    unfolding sssp-log-factor-one-third-def using powr-le by linarith
qed

lemma sssp-log-factor-one-third-square:
  sssp-log-factor-one-third n * sssp-log-factor-one-third n =
    sssp-log-factor n
proof −
  have 0 < ln (real n + 2)
  proof −
    have 1 < real n + 2
      by simp
    then show ?thesis
      by (rule ln-gt-zero)
  qed
  then show ?thesis
    unfolding sssp-log-factor-one-third-def sssp-log-factor-def
    by (simp add: powr-add [symmetric])
qed

lemma eventually-one-le-sssp-log-factor-one-third:
  eventually (λn. 1 ≤ sssp-log-factor-one-third n) at-top
proof (rule eventually-at-top-linorderI[of 1])
  fix n :: nat
  assume n-ge: 1 ≤ n
  have exp 1 ≤ (3::real)
    by (rule exp-le)
  also have ... ≤ real n + 2
    using n-ge by simp
  finally have exp-le-arg: exp 1 ≤ real n + 2 .
  have ln-ge: 1 ≤ ln (real n + 2)
    using exp-le-arg by (simp add: ln-ge-iff)
  show 1 ≤ sssp-log-factor-one-third n
    unfolding sssp-log-factor-one-third-def
    by (rule ge-one-powr-ge-zero[OF ln-ge]) simp
qed

lemma sssp-time-target-nonneg [simp]:
  0 ≤ sssp-time-target m n
proof −
  have 0 ≤ real (m n)
    by simp
```

```
  moreover have 0 ≤ sssp-log-factor n
    using sssp-log-factor-pos[of n] by linarith
  ultimately show ?thesis
    unfolding sssp-time-target-def by (rule mult-nonneg-nonneg)
qed

theorem sssp-time-target-bigoI-norm:
  assumes C: 0 < C
    and eventually-bound:
      eventually
        (λn. norm (real (T n)) ≤ C * norm (sssp-time-target m n)) at-top
  shows (λn. real (T n)) ∈ O(λn. sssp-time-target m n)
  using C eventually-bound unfolding bigo-def by blast

theorem sssp-time-target-bigoI:
  assumes C: 0 < C
    and eventually-bound:
      eventually (λn. real (T n) ≤ C * sssp-time-target m n) at-top
  shows (λn. real (T n)) ∈ O(λn. sssp-time-target m n)
proof (rule sssp-time-target-bigoI-norm[OF C])
  show eventually
      (λn. norm (real (T n)) ≤ C * norm (sssp-time-target m n)) at-top
    using eventually-bound
  proof eventually-elim
    case (elim n)
    then show ?case
      by simp
  qed
qed

definition bmssp-graph-time-bound where
  bmssp-graph-time-bound A R l t m n =
    2 * A n * (2 * l n + 1) * n + (R n + l n * t n) * m n

theorem bmssp-graph-time-bound-bigo-sssp-time-target:
  assumes Cn-pos: 0 < Cn
    and Cv-pos: 0 < Cv
    and Ce-pos: 0 < Ce
    and vertex-count-dominated:
      eventually (λn. real n ≤ Cn * real (m n)) at-top
    and vertex-factor:
      eventually
        (λn. real (2 * A n * (2 * l n + 1)) ≤
          Cv * sssp-log-factor n) at-top
    and edge-factor:
      eventually
        (λn. real (R n + l n * t n) ≤
          Ce * sssp-log-factor n) at-top
  shows (λn. real (bmssp-graph-time-bound A R l t m n))
```

```
    ∈ O(λn. sssp-time-target m n)
proof (rule sssp-time-target-bigoI[where C = Cv * Cn + Ce])
  show 0 < Cv * Cn + Ce
  proof −
    have 0 < Cv * Cn
      using Cv-pos Cn-pos by (rule mult-pos-pos)
    then show ?thesis
      using Ce-pos by linarith
  qed
next
  show eventually
      (λn. real (bmssp-graph-time-bound A R l t m n) ≤
        (Cv * Cn + Ce) * sssp-time-target m n) at-top
    using vertex-count-dominated vertex-factor edge-factor
  proof eventually-elim
    case (elim n)
    let ?vf = real (2 * A n * (2 * l n + 1))
    let ?ef = real (R n + l n * t n)
    let ?log = sssp-log-factor n
    have vertex:
      ?vf * real n ≤ (Cv * ?log) * (Cn * real (m n))
    proof (rule mult-mono)
      show ?vf ≤ Cv * ?log
        using elim(2) .
      show real n ≤ Cn * real (m n)
        using elim(1) .
      show 0 ≤ real n
        by simp
      show 0 ≤ Cv * ?log
      proof −
        have 0 ≤ Cv
          using Cv-pos by linarith
        moreover have 0 ≤ ?log
          using sssp-log-factor-pos[of n] by linarith
        ultimately show ?thesis
          by (rule mult-nonneg-nonneg)
      qed
    qed
    also have ... = (Cv * Cn) * (real (m n) * ?log)
      by (simp add: algebra-simps)
    finally have vertex':
      ?vf * real n ≤ (Cv * Cn) * (real (m n) * ?log) .
    have edge:
      ?ef * real (m n) ≤ (Ce * ?log) * real (m n)
      by (rule mult-right-mono[OF elim(3)]) simp
    also have ... = Ce * (real (m n) * ?log)
      by (simp add: algebra-simps)
    finally have edge':
      ?ef * real (m n) ≤ Ce * (real (m n) * ?log) .
```

```
    have real (bmssp-graph-time-bound A R l t m n) =
        ?vf * real n + ?ef * real (m n)
      unfolding bmssp-graph-time-bound-def by simp
    also have ... ≤
        (Cv * Cn) * (real (m n) * ?log) +
        Ce * (real (m n) * ?log)
      using vertex' edge' by linarith
    also have ... = (Cv * Cn + Ce) * sssp-time-target m n
      unfolding sssp-time-target-def by (simp add: algebra-simps)
    finally show ?case .
  qed
qed

theorem bmssp-cost-bound-bigo-sssp-time-target:
  assumes Cn-pos: 0 < Cn
    and Cv-pos: 0 < Cv
    and Ce-pos: 0 < Ce
    and vertex-count-dominated:
      eventually (λn. real n ≤ Cn * real (m n)) at-top
    and vertex-factor:
      eventually
        (λn. real (2 * A n * (2 * l n + 1)) ≤
          Cv * sssp-log-factor n) at-top
    and edge-factor:
      eventually
        (λn. real (R n + l n * t n) ≤
          Ce * sssp-log-factor n) at-top
    and cost-bound:
      eventually (λn. T n ≤ bmssp-graph-time-bound A R l t m n) at-top
  shows (λn. real (T n)) ∈ O(λn. sssp-time-target m n)
proof −
  have graph:
    (λn. real (bmssp-graph-time-bound A R l t m n))
      ∈ O(λn. sssp-time-target m n)
    by (rule bmssp-graph-time-bound-bigo-sssp-time-target
      [OF Cn-pos Cv-pos Ce-pos vertex-count-dominated vertex-factor
        edge-factor])
  then obtain C where C-pos: 0 < C
    and eventually-graph:
      eventually
        (λn. norm (real (bmssp-graph-time-bound A R l t m n)) ≤
          C * norm (sssp-time-target m n)) at-top
    unfolding bigo-def by blast
  show ?thesis
  proof (rule sssp-time-target-bigoI-norm[OF C-pos])
    show eventually
        (λn. norm (real (T n)) ≤ C * norm (sssp-time-target m n)) at-top
      using cost-bound eventually-graph
    proof eventually-elim
```

```
    case (elim n)
    have norm (real (T n)) ≤
        norm (real (bmssp-graph-time-bound A R l t m n))
      using elim(1) by simp
    also have ... ≤ C * norm (sssp-time-target m n)
      using elim(2) .
    finally show ?case .
  qed
 qed
qed

definition bmssp-refined-graph-time-bound where
 bmssp-refined-graph-time-bound A R H l t m n =
  2 * A n * (2 * l n + 1) * n +
  (R n + t n + l n * H n) * m n

theorem bmssp-refined-graph-time-bound-bigo-sssp-time-target:
 assumes Cn-pos: 0 < Cn
  and Cv-pos: 0 < Cv
  and Ce-pos: 0 < Ce
  and vertex-count-dominated:
    eventually (λn. real n ≤ Cn * real (m n)) at-top
  and vertex-factor:
    eventually
      (λn. real (2 * A n * (2 * l n + 1)) ≤
        Cv * sssp-log-factor n) at-top
  and edge-factor:
    eventually
      (λn. real (R n + t n + l n * H n) ≤
        Ce * sssp-log-factor n) at-top
 shows (λn. real (bmssp-refined-graph-time-bound A R H l t m n))
  ∈ O(λn. sssp-time-target m n)
proof (rule sssp-time-target-bigoI[where C = Cv * Cn + Ce])
 show 0 < Cv * Cn + Ce
 proof −
  have 0 < Cv * Cn
    using Cv-pos Cn-pos by (rule mult-pos-pos)
  then show ?thesis
    using Ce-pos by linarith
 qed
next
 show eventually
    (λn. real (bmssp-refined-graph-time-bound A R H l t m n) ≤
      (Cv * Cn + Ce) * sssp-time-target m n) at-top
  using vertex-count-dominated vertex-factor edge-factor
 proof eventually-elim
  case (elim n)
  let ?vf = real (2 * A n * (2 * l n + 1))
  let ?ef = real (R n + t n + l n * H n)
```

```
    let ?log = sssp-log-factor n
    have vertex:
      ?vf * real n ≤ (Cv * ?log) * (Cn * real (m n))
    proof (rule mult-mono)
      show ?vf ≤ Cv * ?log
        using elim(2) .
      show real n ≤ Cn * real (m n)
        using elim(1) .
      show 0 ≤ real n
        by simp
      show 0 ≤ Cv * ?log
      proof −
        have 0 ≤ Cv
          using Cv-pos by linarith
        moreover have 0 ≤ ?log
          using sssp-log-factor-pos[of n] by linarith
        ultimately show ?thesis
          by (rule mult-nonneg-nonneg)
      qed
    qed
    also have ... = (Cv * Cn) * (real (m n) * ?log)
      by (simp add: algebra-simps)
    finally have vertex':
      ?vf * real n ≤ (Cv * Cn) * (real (m n) * ?log) .
    have edge:
      ?ef * real (m n) ≤ (Ce * ?log) * real (m n)
      by (rule mult-right-mono[OF elim(3)]) simp
    also have ... = Ce * (real (m n) * ?log)
      by (simp add: algebra-simps)
    finally have edge':
      ?ef * real (m n) ≤ Ce * (real (m n) * ?log) .
    have real (bmssp-refined-graph-time-bound A R H l t m n) =
        ?vf * real n + ?ef * real (m n)
      unfolding bmssp-refined-graph-time-bound-def by simp
    also have ... ≤
        (Cv * Cn) * (real (m n) * ?log) +
        Ce * (real (m n) * ?log)
      using vertex' edge' by linarith
    also have ... = (Cv * Cn + Ce) * sssp-time-target m n
      unfolding sssp-time-target-def by (simp add: algebra-simps)
    finally show ?case .
  qed
qed

theorem bmssp-refined-cost-bound-bigo-sssp-time-target:
  assumes Cn-pos: 0 < Cn
    and Cv-pos: 0 < Cv
    and Ce-pos: 0 < Ce
    and vertex-count-dominated:
```

```
    eventually (λn. real n ≤ Cn * real (m n)) at-top
  and vertex-factor:
    eventually
      (λn. real (2 * A n * (2 * l n + 1)) ≤
        Cv * sssp-log-factor n) at-top
  and edge-factor:
    eventually
      (λn. real (R n + t n + l n * H n) ≤
        Ce * sssp-log-factor n) at-top
  and cost-bound:
    eventually
      (λn. T n ≤ bmssp-refined-graph-time-bound A R H l t m n) at-top
 shows (λn. real (T n)) ∈ O(λn. sssp-time-target m n)
proof −
 have graph:
  (λn. real (bmssp-refined-graph-time-bound A R H l t m n))
    ∈ O(λn. sssp-time-target m n)
  by (rule bmssp-refined-graph-time-bound-bigo-sssp-time-target
    [OF Cn-pos Cv-pos Ce-pos vertex-count-dominated vertex-factor
      edge-factor])
 then obtain C where C-pos: 0 < C
  and eventually-graph:
    eventually
      (λn. norm (real (bmssp-refined-graph-time-bound A R H l t m n)) ≤
        C * norm (sssp-time-target m n)) at-top
  unfolding bigo-def by blast
 show ?thesis
 proof (rule sssp-time-target-bigoI-norm[OF C-pos])
  show eventually
      (λn. norm (real (T n)) ≤ C * norm (sssp-time-target m n)) at-top
    using cost-bound eventually-graph
  proof eventually-elim
    case (elim n)
    have norm (real (T n)) ≤
        norm (real (bmssp-refined-graph-time-bound A R H l t m n))
      using elim(1) by simp
    also have ... ≤ C * norm (sssp-time-target m n)
      using elim(2) .
    finally show ?case .
  qed
 qed
qed

theorem bmssp-refined-cost-bound-bigo-sssp-time-target-from-component-bounds:
 assumes Cn-pos: 0 < Cn
  and Ca-pos: 0 < Ca
  and Cl-nonneg: 0 ≤ Cl
  and Cr-pos: 0 < Cr
  and Ct-pos: 0 < Ct
```

```
    and Ch-pos: 0 < Ch
    and vertex-count-dominated:
      eventually (λn. real n ≤ Cn * real (m n)) at-top
    and A-bound:
      eventually
        (λn. real (A n) ≤ Ca * sssp-log-factor-one-third n) at-top
    and l-bound:
      eventually
        (λn. real (l n) ≤ Cl * sssp-log-factor-one-third n) at-top
    and R-bound:
      eventually
        (λn. real (R n) ≤ Cr * sssp-log-factor n) at-top
    and t-bound:
      eventually
        (λn. real (t n) ≤ Ct * sssp-log-factor n) at-top
    and H-bound:
      eventually
        (λn. real (H n) ≤ Ch * sssp-log-factor-one-third n) at-top
    and cost-bound:
      eventually
        (λn. T n ≤ bmssp-refined-graph-time-bound A R H l t m n) at-top
  shows (λn. real (T n)) ∈ O(λn. sssp-time-target m n)
proof (rule bmssp-refined-cost-bound-bigo-sssp-time-target
    [where Cn = Cn and Cv = 2 * Ca * (2 * Cl + 1)
      and Ce = Cr + Ct + Cl * Ch])
  show 0 < Cn
    using Cn-pos .
  show 0 < 2 * Ca * (2 * Cl + 1)
    using Ca-pos Cl-nonneg by (intro mult-pos-pos) simp-all
  show 0 < Cr + Ct + Cl * Ch
  proof −
    have 0 ≤ Cl * Ch
      using Cl-nonneg Ch-pos by simp
    then show ?thesis
      using Cr-pos Ct-pos by linarith
  qed
  show eventually (λn. real n ≤ Cn * real (m n)) at-top
    using vertex-count-dominated .
  show eventually
      (λn. real (2 * A n * (2 * l n + 1)) ≤
        2 * Ca * (2 * Cl + 1) * sssp-log-factor n) at-top
    using A-bound l-bound eventually-one-le-sssp-log-factor-one-third
  proof eventually-elim
    case (elim n)
    let ?L = sssp-log-factor-one-third n
    have L-nonneg: 0 ≤ ?L
      using sssp-log-factor-one-third-pos[of n] by linarith
    have A-le: real (A n) ≤ Ca * ?L
      using elim(1) .
```

```
    have l-le: real (2 * l n + 1) ≤ (2 * Cl + 1) * ?L
    proof −
      have real (2 * l n + 1) = 2 * real (l n) + 1
        by simp
      also have ... ≤ 2 * (Cl * ?L) + 1
        using elim(2) by linarith
      also have ... ≤ 2 * (Cl * ?L) + ?L
        using elim(3) by linarith
      also have ... = (2 * Cl + 1) * ?L
        by (simp add: algebra-simps)
      finally show ?thesis .
    qed
    have l-rhs-nonneg: 0 ≤ (2 * Cl + 1) * ?L
      using Cl-nonneg L-nonneg by (intro mult-nonneg-nonneg) simp-all
    have prod:
      real (A n) * real (2 * l n + 1) ≤
        (Ca * ?L) * ((2 * Cl + 1) * ?L)
       by (rule mult-mono[OF A-le l-le]) (use L-nonneg Ca-pos l-rhs-nonneg in
simp-all)
    have real (2 * A n * (2 * l n + 1)) =
        2 * (real (A n) * real (2 * l n + 1))
      by (simp add: algebra-simps)
    also have ... ≤ 2 * ((Ca * ?L) * ((2 * Cl + 1) * ?L))
      using prod by simp
    also have ... = 2 * Ca * (2 * Cl + 1) * sssp-log-factor n
      using sssp-log-factor-one-third-square[of n]
      by (simp add: algebra-simps)
    finally show ?case .
  qed
  show eventually
      (λn. real (R n + t n + l n * H n) ≤
        (Cr + Ct + Cl * Ch) * sssp-log-factor n) at-top
    using R-bound t-bound l-bound H-bound
  proof eventually-elim
    case (elim n)
    let ?L = sssp-log-factor-one-third n
    have L-nonneg: 0 ≤ ?L
      using sssp-log-factor-one-third-pos[of n] by linarith
    have lH:
      real (l n * H n) ≤ (Cl * Ch) * sssp-log-factor n
    proof −
      have prod:
        real (l n) * real (H n) ≤ (Cl * ?L) * (Ch * ?L)
        by (rule mult-mono[OF elim(3) elim(4)])
          (use Cl-nonneg Ch-pos L-nonneg in simp-all)
      have real (l n * H n) = real (l n) * real (H n)
        by simp
      also have ... ≤ (Cl * ?L) * (Ch * ?L)
        using prod .
```

```
    also have ... = (Cl * Ch) * sssp-log-factor n
      using sssp-log-factor-one-third-square[of n]
      by (simp add: algebra-simps)
    finally show ?thesis .
  qed
  have real (R n + t n + l n * H n) =
      real (R n) + real (t n) + real (l n * H n)
    by simp
  also have ... ≤
      Cr * sssp-log-factor n + Ct * sssp-log-factor n +
      (Cl * Ch) * sssp-log-factor n
    using elim(1) elim(2) lH by linarith
  also have ... = (Cr + Ct + Cl * Ch) * sssp-log-factor n
    by (simp add: algebra-simps)
  finally show ?case .
 qed
 show eventually
    (λn. T n ≤ bmssp-refined-graph-time-bound A R H l t m n) at-top
  using cost-bound .
qed

theorem bmssp-refined-graph-time-bound-bigo-sssp-time-target-slack:
 assumes Cn-pos: 0 < Cn
  and Cm-pos: 0 < Cm
  and Cv-pos: 0 < Cv
  and Ce-pos: 0 < Ce
  and vertex-count-dominated:
    eventually (λn. real (v n) ≤ Cn * real (m n)) at-top
  and edge-count-dominated:
    eventually (λn. real (m' n) ≤ Cm * real (m n)) at-top
  and vertex-factor:
    eventually
      (λn. real (2 * A n * (2 * l n + 1)) ≤
        Cv * sssp-log-factor n) at-top
  and edge-factor:
    eventually
      (λn. real (R n + t n + l n * H n) ≤
        Ce * sssp-log-factor n) at-top
 shows (λn. real (bmssp-refined-graph-time-bound
    (λ-. A n) (λ-. R n) (λ-. H n) (λ-. l n) (λ-. t n)
    (λ-. m' n) (v n)))
  ∈ O(λn. sssp-time-target m n)
proof (rule sssp-time-target-bigoI[where C = Cv * Cn + Ce * Cm])
 show 0 < Cv * Cn + Ce * Cm
 proof −
  have 0 < Cv * Cn
    using Cv-pos Cn-pos by (rule mult-pos-pos)
  moreover have 0 < Ce * Cm
    using Ce-pos Cm-pos by (rule mult-pos-pos)
```

```
    ultimately show ?thesis
      by linarith
  qed
next
  show eventually
      (λn. real (bmssp-refined-graph-time-bound
          (λ-. A n) (λ-. R n) (λ-. H n) (λ-. l n) (λ-. t n)
          (λ-. m' n) (v n)) ≤
        (Cv * Cn + Ce * Cm) * sssp-time-target m n) at-top
    using vertex-count-dominated edge-count-dominated vertex-factor edge-factor
  proof eventually-elim
    case (elim n)
    let ?vf = real (2 * A n * (2 * l n + 1))
    let ?ef = real (R n + t n + l n * H n)
    let ?log = sssp-log-factor n
    have vertex:
      ?vf * real (v n) ≤ (Cv * ?log) * (Cn * real (m n))
    proof (rule mult-mono)
      show ?vf ≤ Cv * ?log
        using elim(3) .
      show real (v n) ≤ Cn * real (m n)
        using elim(1) .
      show 0 ≤ real (v n)
        by simp
      show 0 ≤ Cv * ?log
      proof −
        have 0 ≤ Cv
          using Cv-pos by linarith
        moreover have 0 ≤ ?log
          using sssp-log-factor-pos[of n] by linarith
        ultimately show ?thesis
          by (rule mult-nonneg-nonneg)
      qed
    qed
    also have ... = (Cv * Cn) * (real (m n) * ?log)
      by (simp add: algebra-simps)
    finally have vertex':
      ?vf * real (v n) ≤ (Cv * Cn) * (real (m n) * ?log) .
    have edge:
      ?ef * real (m' n) ≤ (Ce * ?log) * (Cm * real (m n))
    proof (rule mult-mono)
      show ?ef ≤ Ce * ?log
        using elim(4) .
      show real (m' n) ≤ Cm * real (m n)
        using elim(2) .
      show 0 ≤ real (m' n)
        by simp
      show 0 ≤ Ce * ?log
      proof −
```

```
      have 0 ≤ Ce
        using Ce-pos by linarith
      moreover have 0 ≤ ?log
        using sssp-log-factor-pos[of n] by linarith
      ultimately show ?thesis
        by (rule mult-nonneg-nonneg)
    qed
  qed
  also have ... = (Ce * Cm) * (real (m n) * ?log)
    by (simp add: algebra-simps)
  finally have edge':
    ?ef * real (m' n) ≤ (Ce * Cm) * (real (m n) * ?log) .
  have real (bmssp-refined-graph-time-bound
      (λ-. A n) (λ-. R n) (λ-. H n) (λ-. l n) (λ-. t n)
      (λ-. m' n) (v n)) =
    ?vf * real (v n) + ?ef * real (m' n)
    unfolding bmssp-refined-graph-time-bound-def by simp
  also have ... ≤
    (Cv * Cn) * (real (m n) * ?log) +
    (Ce * Cm) * (real (m n) * ?log)
    using vertex' edge' by linarith
  also have ... = (Cv * Cn + Ce * Cm) * sssp-time-target m n
    unfolding sssp-time-target-def by (simp add: algebra-simps)
  finally show ?case .
 qed
qed

theorem bmssp-refined-cost-bound-bigo-sssp-time-target-from-component-bounds-slack:
 assumes Cn-pos: 0 < Cn
   and Cm-pos: 0 < Cm
   and Ca-pos: 0 < Ca
   and Cl-nonneg: 0 ≤ Cl
   and Cr-pos: 0 < Cr
   and Ct-pos: 0 < Ct
   and Ch-pos: 0 < Ch
   and vertex-count-dominated:
     eventually (λn. real (v n) ≤ Cn * real (m n)) at-top
   and edge-count-dominated:
     eventually (λn. real (m' n) ≤ Cm * real (m n)) at-top
   and A-bound:
     eventually
       (λn. real (A n) ≤ Ca * sssp-log-factor-one-third n) at-top
   and l-bound:
     eventually
       (λn. real (l n) ≤ Cl * sssp-log-factor-one-third n) at-top
   and R-bound:
     eventually
       (λn. real (R n) ≤ Cr * sssp-log-factor n) at-top
```

```
    and t-bound:
      eventually
        (λn. real (t n) ≤ Ct * sssp-log-factor n) at-top
    and H-bound:
      eventually
        (λn. real (H n) ≤ Ch * sssp-log-factor-one-third n) at-top
    and cost-bound:
      eventually
        (λn. T n ≤ bmssp-refined-graph-time-bound
          (λ-. A n) (λ-. R n) (λ-. H n) (λ-. l n) (λ-. t n)
          (λ-. m' n) (v n)) at-top
  shows (λn. real (T n)) ∈ O(λn. sssp-time-target m n)
proof −
  have Cv-pos: 0 < 2 * Ca * (2 * Cl + 1)
    using Ca-pos Cl-nonneg by (intro mult-pos-pos) simp-all
  have Ce-pos: 0 < Cr + Ct + Cl * Ch
  proof −
    have 0 ≤ Cl * Ch
      using Cl-nonneg Ch-pos by simp
    then show ?thesis
      using Cr-pos Ct-pos by linarith
  qed
  have vertex-factor:
    eventually
      (λn. real (2 * A n * (2 * l n + 1)) ≤
        2 * Ca * (2 * Cl + 1) * sssp-log-factor n) at-top
    using A-bound l-bound eventually-one-le-sssp-log-factor-one-third
  proof eventually-elim
    case (elim n)
    let ?L = sssp-log-factor-one-third n
    have L-nonneg: 0 ≤ ?L
      using sssp-log-factor-one-third-pos[of n] by linarith
    have A-le: real (A n) ≤ Ca * ?L
      using elim(1) .
    have l-le: real (2 * l n + 1) ≤ (2 * Cl + 1) * ?L
    proof −
      have real (2 * l n + 1) = 2 * real (l n) + 1
        by simp
      also have ... ≤ 2 * (Cl * ?L) + 1
        using elim(2) by linarith
      also have ... ≤ 2 * (Cl * ?L) + ?L
        using elim(3) by linarith
      also have ... = (2 * Cl + 1) * ?L
        by (simp add: algebra-simps)
      finally show ?thesis .
    qed
    have l-rhs-nonneg: 0 ≤ (2 * Cl + 1) * ?L
      using Cl-nonneg L-nonneg by (intro mult-nonneg-nonneg) simp-all
    have prod:
```

```
    real (A n) * real (2 * l n + 1) ≤
      (Ca * ?L) * ((2 * Cl + 1) * ?L)
    by (rule mult-mono[OF A-le l-le])
      (use L-nonneg Ca-pos l-rhs-nonneg in simp-all)
  have real (2 * A n * (2 * l n + 1)) =
      2 * (real (A n) * real (2 * l n + 1))
    by (simp add: algebra-simps)
  also have ... ≤ 2 * ((Ca * ?L) * ((2 * Cl + 1) * ?L))
    using prod by simp
  also have ... = 2 * Ca * (2 * Cl + 1) * sssp-log-factor n
    using sssp-log-factor-one-third-square[of n]
    by (simp add: algebra-simps)
  finally show ?case .
qed
have edge-factor:
  eventually
    (λn. real (R n + t n + l n * H n) ≤
      (Cr + Ct + Cl * Ch) * sssp-log-factor n) at-top
  using R-bound t-bound l-bound H-bound
proof eventually-elim
  case (elim n)
  let ?L = sssp-log-factor-one-third n
  have L-nonneg: 0 ≤ ?L
    using sssp-log-factor-one-third-pos[of n] by linarith
  have lH:
    real (l n * H n) ≤ (Cl * Ch) * sssp-log-factor n
  proof −
    have prod:
      real (l n) * real (H n) ≤ (Cl * ?L) * (Ch * ?L)
      by (rule mult-mono[OF elim(3) elim(4)])
        (use Cl-nonneg Ch-pos L-nonneg in simp-all)
    have real (l n * H n) = real (l n) * real (H n)
      by simp
    also have ... ≤ (Cl * ?L) * (Ch * ?L)
      using prod .
    also have ... = (Cl * Ch) * sssp-log-factor n
      using sssp-log-factor-one-third-square[of n]
      by (simp add: algebra-simps)
    finally show ?thesis .
  qed
  have real (R n + t n + l n * H n) =
      real (R n) + real (t n) + real (l n * H n)
    by simp
  also have ... ≤
      Cr * sssp-log-factor n + Ct * sssp-log-factor n +
      (Cl * Ch) * sssp-log-factor n
    using elim(1) elim(2) lH by linarith
  also have ... = (Cr + Ct + Cl * Ch) * sssp-log-factor n
    by (simp add: algebra-simps)
```

```
    finally show ?case .
  qed
  have graph:
    (λn. real (bmssp-refined-graph-time-bound
        (λ-. A n) (λ-. R n) (λ-. H n) (λ-. l n) (λ-. t n)
        (λ-. m' n) (v n)))
      ∈ O(λn. sssp-time-target m n)
    by (rule bmssp-refined-graph-time-bound-bigo-sssp-time-target-slack
      [OF Cn-pos Cm-pos Cv-pos Ce-pos vertex-count-dominated
        edge-count-dominated vertex-factor edge-factor])
  then obtain C where C-pos: 0 < C
    and eventually-graph:
      eventually
        (λn. norm (real (bmssp-refined-graph-time-bound
          (λ-. A n) (λ-. R n) (λ-. H n) (λ-. l n) (λ-. t n)
          (λ-. m' n) (v n))) ≤
          C * norm (sssp-time-target m n)) at-top
    unfolding bigo-def by blast
  show ?thesis
  proof (rule sssp-time-target-bigoI-norm[OF C-pos])
    show eventually
        (λn. norm (real (T n)) ≤ C * norm (sssp-time-target m n)) at-top
      using cost-bound eventually-graph
    proof eventually-elim
      case (elim n)
      have norm (real (T n)) ≤
          norm (real (bmssp-refined-graph-time-bound
            (λ-. A n) (λ-. R n) (λ-. H n) (λ-. l n) (λ-. t n)
            (λ-. m' n) (v n)))
        using elim(1) by simp
      also have ... ≤ C * norm (sssp-time-target m n)
        using elim(2) .
      finally show ?case .
    qed
  qed
qed
```

29 Logarithmic Parameter Schedule

The paper's schedule balances two logarithmic scales. The parameter usually called k is chosen on the order of $log\hat{}(1/3)\ n$, while the recursion depth and partition width use the corresponding $log\hat{}(2/3)\ n$ scale. The two definitions below use ceilings of the real-valued factors so that the executable and inductive parameters remain natural numbers.

The lemmas in this section are mostly analytic bookkeeping. They show that the ceiling parameters are eventually positive, monotone enough for the recurrence, and asymptotically within constant factors of the real logarithmic targets. Those facts are what let the refined graph-time expressions be

rewritten as $O(m * log\hat{}(2/3)\ n)$.

definition *sssp-log-one-third-param* **where**
sssp-log-one-third-param n = *nat* ⌈*sssp-log-factor-one-third n*⌉

definition *sssp-log-two-thirds-param* **where**
sssp-log-two-thirds-param n = *nat* ⌈*sssp-log-factor n*⌉

lemma *sssp-log-one-third-param-pos* [*simp*]:
0 < *sssp-log-one-third-param n*
unfolding *sssp-log-one-third-param-def*
using *sssp-log-factor-one-third-pos*[*of n*] **by** *simp*

lemma *sssp-log-two-thirds-param-pos* [*simp*]:
0 < *sssp-log-two-thirds-param n*
unfolding *sssp-log-two-thirds-param-def*
using *sssp-log-factor-pos*[*of n*] **by** *simp*

lemma *real-cuberoot-at-top*:
filterlim (λ*x*::*real*. *root 3 x*) *at-top at-top*
by (*rule filterlim-at-top-at-top*
[**where** *Q* = λ*x*. *True* **and** *P* = λ*x*. *True* **and** *g* = λ*x*. *x* ^ *3*])
(*auto intro*: *real-root-le-mono simp*: *odd-real-root-power-cancel*)

lemma *sssp-log-factor-one-third-at-top*:
filterlim sssp-log-factor-one-third at-top at-top
proof −
have *ln-at-top'*:
filterlim (λ*n*::*nat*. *ln* (*real* (*n* + *2*))) *at-top at-top*
by (*intro filterlim-compose*[*OF ln-at-top*]
filterlim-compose[*OF filterlim-real-sequentially*]
filterlim-add-const-nat-at-top)
have *root-ln-at-top*:
filterlim (λ*n*::*nat*. *root 3* (*ln* (*real* (*n* + *2*)))) *at-top at-top*
by (*rule filterlim-compose*[*OF real-cuberoot-at-top ln-at-top'*])
have *eq*:
(λ*n*::*nat*. *root 3* (*ln* (*real* (*n* + *2*)))) = *sssp-log-factor-one-third*
proof
fix *n* :: *nat*
have *ln-nonneg*: *0* ≤ *ln* (*real* (*n* + *2*))
by *simp*
have *root 3* (*ln* (*real* (*n* + *2*))) =
ln (*real* (*n* + *2*)) *powr* (*1* / *3*)
using *root-powr-inverse*[*of 3 ln* (*real* (*n* + *2*))] *ln-nonneg*
by *simp*
then show *root 3* (*ln* (*real* (*n* + *2*))) = *sssp-log-factor-one-third n*
unfolding *sssp-log-factor-one-third-def*
by (*simp add*: *add.commute*)
qed
show *?thesis*

using *root-ln-at-top* **unfolding** *eq* **.**
qed

lemma *nat-ceiling-at-top*:
filterlim (λx::*real*. *nat* $\lceil x \rceil$) *at-top at-top*
proof (*rule filterlim-at-top-at-top*
[**where** $Q = \lambda x.\ 0 \leq x$ **and** $P = \lambda$-::*nat*. *True* **and** g = *real*])
fix $x\ y$ :: *real*
assume *x-nonneg*: $0 \leq x$ **and** *y-nonneg*: $0 \leq y$ **and** *xy*: $x \leq y$
have $\lceil x \rceil \leq \lceil y \rceil$
by (*rule ceiling-mono*[*OF xy*])
then show *nat* $\lceil x \rceil \leq$ *nat* $\lceil y \rceil$
by (*rule nat-mono*)
next
fix n :: *nat*
show *nat* $\lceil$*real* $n\rceil = n$
by *simp*
next
fix n :: *nat*
show $0 \leq$ *real* n
by *simp*
next
show *eventually* (λx::*real*. $0 \leq x$) *at-top*
by (*rule eventually-ge-at-top*)
next
show *eventually* (λ-::*nat*. *True*) *at-top*
by *simp*
qed

lemma *sssp-log-one-third-param-at-top*:
filterlim sssp-log-one-third-param at-top at-top
unfolding *sssp-log-one-third-param-def*
by (*rule filterlim-compose*
[*OF nat-ceiling-at-top sssp-log-factor-one-third-at-top*])

lemma *eventually-less-sssp-log-one-third-param*:
eventually ($\lambda n.\ K <$ *sssp-log-one-third-param* n) *at-top*
proof −
have *eventually* ($\lambda n.$ *Suc* $K \leq$ *sssp-log-one-third-param* n) *at-top*
using *sssp-log-one-third-param-at-top*
unfolding *filterlim-at-top* **by** *blast*
then show *?thesis*
by *eventually-elim simp*
qed

lemma *sssp-log-two-thirds-param-le-one-third-square*:
sssp-log-two-thirds-param $n \leq$
sssp-log-one-third-param $n *$ *sssp-log-one-third-param* n
proof −

```
  let ?L = sssp-log-factor-one-third n
  let ?p = sssp-log-one-third-param n
  have L-nonneg: 0 ≤ ?L
    using sssp-log-factor-one-third-pos[of n] by linarith
  have L-le-p: ?L ≤ real ?p
    unfolding sssp-log-one-third-param-def
    using L-nonneg by simp
  have sssp-log-factor n ≤ real (?p * ?p)
  proof −
    have ?L * ?L ≤ real ?p * real ?p
      by (rule mult-mono[OF L-le-p L-le-p])
        (use L-nonneg in simp-all)
    then show ?thesis
      using sssp-log-factor-one-third-square[of n] by simp
  qed
  then show ?thesis
    unfolding sssp-log-two-thirds-param-def by simp
qed

lemma real-nat-ceiling-le-twice:
  fixes x :: real
  assumes 1 ≤ x
  shows real (nat ⌈x⌉) ≤ 2 * x
proof −
  have x-nonneg: 0 ≤ x
    using assms by linarith
  have ceil-nat: real (nat ⌈x⌉) = of-int ⌈x⌉
    using of-nat-int-ceiling[OF x-nonneg] .
  have of-int ⌈x⌉ − 1 < x
    using ceiling-correct[of x] by blast
  then have of-int ⌈x⌉ ≤ x + 1
    by linarith
  also have ... ≤ 2 * x
    using assms by linarith
  finally show ?thesis
    unfolding ceil-nat .
qed

lemma eventually-one-le-sssp-log-factor:
  eventually (λn. 1 ≤ sssp-log-factor n) at-top
  using eventually-one-le-sssp-log-factor-one-third
proof eventually-elim
  case (elim n)
  let ?L = sssp-log-factor-one-third n
  have 1 ≤ ?L * ?L
  proof −
    have L-nonneg: 0 ≤ ?L
      using elim by linarith
    have (1::real) * 1 ≤ ?L * ?L
```

```
    by (rule mult-mono[OF elim elim]) (simp-all add: L-nonneg)
  then show ?thesis
    by simp
 qed
 then show ?case
  using sssp-log-factor-one-third-square[of n] by simp
qed

lemma sssp-log-one-third-param-component-bound:
 eventually
  (λn. real (sssp-log-one-third-param n) ≤
    2 * sssp-log-factor-one-third n) at-top
 using eventually-one-le-sssp-log-factor-one-third
proof eventually-elim
 case (elim n)
 show ?case
  unfolding sssp-log-one-third-param-def
  by (rule real-nat-ceiling-le-twice[OF elim])
qed

lemma sssp-log-two-thirds-param-component-bound:
 eventually
  (λn. real (sssp-log-two-thirds-param n) ≤
    2 * sssp-log-factor n) at-top
 using eventually-one-le-sssp-log-factor
proof eventually-elim
 case (elim n)
 show ?case
  unfolding sssp-log-two-thirds-param-def
  by (rule real-nat-ceiling-le-twice[OF elim])
qed

lemma sssp-log-one-third-param-square-arg-component-bound:
 eventually
  (λn. real (sssp-log-one-third-param (n * n)) ≤
    4 * sssp-log-factor-one-third n) at-top
proof (rule eventually-at-top-linorderI[of 1])
 fix n :: nat
 assume n-ge: 1 ≤ n
 have one-le: 1 ≤ sssp-log-factor-one-third (n * n)
 proof −
  have 1 ≤ sssp-log-factor-one-third n
  proof −
   have exp 1 ≤ (3::real)
    by (rule exp-le)
   also have ... ≤ real n + 2
    using n-ge by simp
   finally have exp-le-arg: exp 1 ≤ real n + 2 .
   have ln-ge: 1 ≤ ln (real n + 2)
```

```
      using exp-le-arg by (simp add: ln-ge-iff)
    show ?thesis
      unfolding sssp-log-factor-one-third-def
      by (rule ge-one-powr-ge-zero[OF ln-ge]) simp
  qed
  also have ... ≤ sssp-log-factor-one-third (n * n)
    by (rule sssp-log-factor-one-third-mono) (use n-ge in simp)
  finally show ?thesis .
 qed
 have real (sssp-log-one-third-param (n * n)) ≤
    2 * sssp-log-factor-one-third (n * n)
  unfolding sssp-log-one-third-param-def
  by (rule real-nat-ceiling-le-twice[OF one-le])
 also have ... ≤ 2 * (2 * sssp-log-factor-one-third n)
  using sssp-log-factor-one-third-square-arg-le[of n] by simp
 also have ... = 4 * sssp-log-factor-one-third n
  by simp
 finally show real (sssp-log-one-third-param (n * n)) ≤
    4 * sssp-log-factor-one-third n .
qed

lemma sssp-log-two-thirds-param-square-arg-component-bound:
 eventually
  (λn. real (sssp-log-two-thirds-param (n * n)) ≤
    4 * sssp-log-factor n) at-top
proof (rule eventually-at-top-linorderI[of 1])
 fix n :: nat
 assume n-ge: 1 ≤ n
 have one-le: 1 ≤ sssp-log-factor (n * n)
 proof −
  have 1 ≤ sssp-log-factor n
  proof −
   have one-third: 1 ≤ sssp-log-factor-one-third n
   proof −
    have exp 1 ≤ (3::real)
     by (rule exp-le)
    also have ... ≤ real n + 2
     using n-ge by simp
    finally have exp-le-arg: exp 1 ≤ real n + 2 .
    have ln-ge: 1 ≤ ln (real n + 2)
     using exp-le-arg by (simp add: ln-ge-iff)
    show ?thesis
     unfolding sssp-log-factor-one-third-def
     by (rule ge-one-powr-ge-zero[OF ln-ge]) simp
   qed
   let ?L = sssp-log-factor-one-third n
   have 1 ≤ ?L * ?L
   proof −
    have L-nonneg: 0 ≤ ?L
```

```
      using one-third by linarith
    have (1::real) * 1 ≤ ?L * ?L
      by (rule mult-mono[OF one-third one-third])
        (simp-all add: L-nonneg)
    then show ?thesis
      by simp
  qed
  then show ?thesis
    using sssp-log-factor-one-third-square[of n] by simp
 qed
 also have ... ≤ sssp-log-factor (n * n)
  by (rule sssp-log-factor-mono) (use n-ge in simp)
 finally show ?thesis .
qed
have real (sssp-log-two-thirds-param (n * n)) ≤
  2 * sssp-log-factor (n * n)
 unfolding sssp-log-two-thirds-param-def
 by (rule real-nat-ceiling-le-twice[OF one-le])
also have ... ≤ 2 * (2 * sssp-log-factor n)
 using sssp-log-factor-square-arg-le[of n] by simp
also have ... = 4 * sssp-log-factor n
 by simp
finally show real (sssp-log-two-thirds-param (n * n)) ≤
  4 * sssp-log-factor n .
qed

theorem bmssp-refined-cost-bound-bigo-sssp-time-target-log-params:
 assumes Cn-pos: 0 < Cn
  and vertex-count-dominated:
   eventually (λn. real n ≤ Cn * real (m n)) at-top
  and cost-bound:
   eventually
    (λn. T n ≤
      bmssp-refined-graph-time-bound sssp-log-one-third-param
       sssp-log-two-thirds-param sssp-log-one-third-param
       sssp-log-one-third-param sssp-log-two-thirds-param m n) at-top
 shows (λn. real (T n)) ∈ O(λn. sssp-time-target m n)
proof (rule bmssp-refined-cost-bound-bigo-sssp-time-target-from-component-bounds
  [where Cn = Cn and Ca = 2 and Cl = 2 and Cr = 2 and Ct = 2 and Ch
= 2])
 show 0 < Cn
  using Cn-pos .
 show 0 < (2::real)
  by simp
 show 0 ≤ (2::real)
  by simp
 show 0 < (2::real)
  by simp
 show 0 < (2::real)
```

```
    by simp
  show 0 < (2::real)
    by simp
  show eventually (λn. real n ≤ Cn * real (m n)) at-top
    using vertex-count-dominated .
  show eventually
      (λn. real (sssp-log-one-third-param n) ≤
        2 * sssp-log-factor-one-third n) at-top
    by (rule sssp-log-one-third-param-component-bound)
  show eventually
      (λn. real (sssp-log-one-third-param n) ≤
        2 * sssp-log-factor-one-third n) at-top
    by (rule sssp-log-one-third-param-component-bound)
  show eventually
      (λn. real (sssp-log-two-thirds-param n) ≤
        2 * sssp-log-factor n) at-top
    by (rule sssp-log-two-thirds-param-component-bound)
  show eventually
      (λn. real (sssp-log-two-thirds-param n) ≤
        2 * sssp-log-factor n) at-top
    by (rule sssp-log-two-thirds-param-component-bound)
  show eventually
      (λn. real (sssp-log-one-third-param n) ≤
        2 * sssp-log-factor-one-third n) at-top
    by (rule sssp-log-one-third-param-component-bound)
  show eventually
      (λn. T n ≤
        bmssp-refined-graph-time-bound sssp-log-one-third-param
          sssp-log-two-thirds-param sssp-log-one-third-param
          sssp-log-one-third-param sssp-log-two-thirds-param m n) at-top
    using cost-bound .
qed

theorem bmssp-refined-cost-bound-bigo-sssp-time-target-log-params-bounded-degree:
  fixes D :: nat
  assumes Cn-pos: 0 < Cn
    and vertex-count-dominated:
      eventually (λn. real n ≤ Cn * real (m n)) at-top
    and cost-bound:
      eventually
        (λn. T n ≤
          bmssp-refined-graph-time-bound
            (λn. Suc D * sssp-log-one-third-param n)
            sssp-log-two-thirds-param sssp-log-one-third-param
            sssp-log-one-third-param sssp-log-two-thirds-param m n) at-top
  shows (λn. real (T n)) ∈ O(λn. sssp-time-target m n)
proof (rule bmssp-refined-cost-bound-bigo-sssp-time-target-from-component-bounds
    [where Cn = Cn and Ca = 2 * real (Suc D)
      and Cl = 2 and Cr = 2 and Ct = 2 and Ch = 2])
```

show *0 < Cn*
  using *Cn-pos* **.**
show *0 < 2 * real (Suc D)*
  by *simp*
show *0 ≤ (2::real)*
  by *simp*
show *0 < (2::real)*
  by *simp*
show *0 < (2::real)*
  by *simp*
show *0 < (2::real)*
  by *simp*
show *eventually (λn. real n ≤ Cn * real (m n)) at-top*
  using *vertex-count-dominated* **.**
show *eventually*
    *(λn. real (Suc D * sssp-log-one-third-param n) ≤*
      *2 * real (Suc D) * sssp-log-factor-one-third n) at-top*
  using *sssp-log-one-third-param-component-bound*
proof *eventually-elim*
  case *(elim n)*
  have *real (Suc D * sssp-log-one-third-param n) =*
      *real (Suc D) * real (sssp-log-one-third-param n)*
    by *(simp add: algebra-simps)*
  also have *... ≤ real (Suc D) **
      *(2 * sssp-log-factor-one-third n)*
    using *elim* **by** *simp*
  also have *... = 2 * real (Suc D) * sssp-log-factor-one-third n*
    by *(simp add: algebra-simps)*
  finally show *?case* **.**
qed
show *eventually*
    (λn. real (sssp-log-one-third-param n) ≤
      *2 * sssp-log-factor-one-third n) at-top*
  by *(rule sssp-log-one-third-param-component-bound)*
show *eventually*
    (λn. real (sssp-log-two-thirds-param n) ≤
      *2 * sssp-log-factor n) at-top*
  by *(rule sssp-log-two-thirds-param-component-bound)*
show *eventually*
    (λn. real (sssp-log-two-thirds-param n) ≤
      *2 * sssp-log-factor n) at-top*
  by *(rule sssp-log-two-thirds-param-component-bound)*
show *eventually*
    (λn. real (sssp-log-one-third-param n) ≤
      *2 * sssp-log-factor-one-third n) at-top*
  by *(rule sssp-log-one-third-param-component-bound)*
show *eventually*
    (λn. T n ≤
      bmssp-refined-graph-time-bound

```
      (λn. Suc D * sssp-log-one-third-param n)
      sssp-log-two-thirds-param sssp-log-one-third-param
      sssp-log-one-third-param sssp-log-two-thirds-param m n) at-top
  using cost-bound .
qed

theorem bmssp-refined-cost-bound-bigo-sssp-time-target-log-params-bounded-degree-slack:
  fixes D :: nat
  assumes Cn-pos: 0 < Cn
    and Cm-pos: 0 < Cm
    and vertex-count-dominated:
      eventually (λn. real (v n) ≤ Cn * real (m n)) at-top
    and edge-count-dominated:
      eventually (λn. real (m' n) ≤ Cm * real (m n)) at-top
    and cost-bound:
      eventually
        (λn. T n ≤
          bmssp-refined-graph-time-bound
            (λ-. Suc D * sssp-log-one-third-param n)
            (λ-. sssp-log-two-thirds-param n)
            (λ-. sssp-log-one-third-param n)
            (λ-. sssp-log-one-third-param n)
            (λ-. sssp-log-two-thirds-param n)
            (λ-. m' n) (v n)) at-top
  shows (λn. real (T n)) ∈ O(λn. sssp-time-target m n)
proof (rule bmssp-refined-cost-bound-bigo-sssp-time-target-from-component-bounds-slack
    [where Cn = Cn and Cm = Cm and Ca = 2 * real (Suc D)
      and Cl = 2 and Cr = 2 and Ct = 2 and Ch = 2])
  show 0 < Cn
    using Cn-pos .
  show 0 < Cm
    using Cm-pos .
  show 0 < 2 * real (Suc D)
    by simp
  show 0 ≤ (2::real)
    by simp
  show 0 < (2::real)
    by simp
  show 0 < (2::real)
    by simp
  show 0 < (2::real)
    by simp
  show eventually (λn. real (v n) ≤ Cn * real (m n)) at-top
    using vertex-count-dominated .
  show eventually (λn. real (m' n) ≤ Cm * real (m n)) at-top
    using edge-count-dominated .
  show eventually
```

```
      (λn. real (Suc D * sssp-log-one-third-param n) ≤
        2 * real (Suc D) * sssp-log-factor-one-third n) at-top
    using sssp-log-one-third-param-component-bound
  proof eventually-elim
    case (elim n)
    have real (Suc D * sssp-log-one-third-param n) =
        real (Suc D) * real (sssp-log-one-third-param n)
      by (simp add: algebra-simps)
    also have ... ≤ real (Suc D) *
        (2 * sssp-log-factor-one-third n)
      using elim by simp
    also have ... = 2 * real (Suc D) * sssp-log-factor-one-third n
      by (simp add: algebra-simps)
    finally show ?case .
  qed
  show eventually
      (λn. real (sssp-log-one-third-param n) ≤
        2 * sssp-log-factor-one-third n) at-top
    by (rule sssp-log-one-third-param-component-bound)
  show eventually
      (λn. real (sssp-log-two-thirds-param n) ≤
        2 * sssp-log-factor n) at-top
    by (rule sssp-log-two-thirds-param-component-bound)
  show eventually
      (λn. real (sssp-log-two-thirds-param n) ≤
        2 * sssp-log-factor n) at-top
    by (rule sssp-log-two-thirds-param-component-bound)
  show eventually
      (λn. real (sssp-log-one-third-param n) ≤
        2 * sssp-log-factor-one-third n) at-top
    by (rule sssp-log-one-third-param-component-bound)
  show eventually
      (λn. T n ≤
        bmssp-refined-graph-time-bound
          (λ-. Suc D * sssp-log-one-third-param n)
          (λ-. sssp-log-two-thirds-param n)
          (λ-. sssp-log-one-third-param n)
          (λ-. sssp-log-one-third-param n)
          (λ-. sssp-log-two-thirds-param n)
          (λ-. m' n) (v n)) at-top
    using cost-bound .
qed

theorem bmssp-refined-cost-bound-bigo-sssp-time-target-log-params-bounded-degree-
square-arg:
  fixes D :: nat
  assumes Cn-pos: 0 < Cn
    and vertex-count-dominated:
      eventually (λn. real n ≤ Cn * real (m n)) at-top
```

```
    and cost-bound:
      eventually
       (λn. T n ≤
         bmssp-refined-graph-time-bound
          (λn. Suc D * sssp-log-one-third-param (n * n))
          (λn. sssp-log-two-thirds-param (n * n))
          (λn. sssp-log-one-third-param (n * n))
          (λn. sssp-log-one-third-param (n * n))
          (λn. sssp-log-two-thirds-param (n * n)) m n) at-top
  shows (λn. real (T n)) ∈ O(λn. sssp-time-target m n)
proof (rule bmssp-refined-cost-bound-bigo-sssp-time-target-from-component-bounds
    [where Cn = Cn and Ca = 4 * real (Suc D)
      and Cl = 4 and Cr = 4 and Ct = 4 and Ch = 4])
  show 0 < Cn
    using Cn-pos .
  show 0 < 4 * real (Suc D)
    by simp
  show 0 ≤ (4::real)
    by simp
  show 0 < (4::real)
    by simp
  show 0 < (4::real)
    by simp
  show 0 < (4::real)
    by simp
  show eventually (λn. real n ≤ Cn * real (m n)) at-top
    using vertex-count-dominated .
  show eventually
      (λn. real (Suc D * sssp-log-one-third-param (n * n)) ≤
        4 * real (Suc D) * sssp-log-factor-one-third n) at-top
    using sssp-log-one-third-param-square-arg-component-bound
  proof eventually-elim
    case (elim n)
    have real (Suc D * sssp-log-one-third-param (n * n)) =
        real (Suc D) * real (sssp-log-one-third-param (n * n))
      by (simp add: algebra-simps)
    also have ... ≤ real (Suc D) *
        (4 * sssp-log-factor-one-third n)
      using elim by simp
    also have ... = 4 * real (Suc D) * sssp-log-factor-one-third n
      by (simp add: algebra-simps)
    finally show ?case .
  qed
  show eventually
      (λn. real (sssp-log-one-third-param (n * n)) ≤
        4 * sssp-log-factor-one-third n) at-top
    by (rule sssp-log-one-third-param-square-arg-component-bound)
  show eventually
      (λn. real (sssp-log-two-thirds-param (n * n)) ≤
```

```
      4 * sssp-log-factor n) at-top
    by (rule sssp-log-two-thirds-param-square-arg-component-bound)
  show eventually
     (λn. real (sssp-log-two-thirds-param (n * n)) ≤
      4 * sssp-log-factor n) at-top
    by (rule sssp-log-two-thirds-param-square-arg-component-bound)
  show eventually
     (λn. real (sssp-log-one-third-param (n * n)) ≤
      4 * sssp-log-factor-one-third n) at-top
    by (rule sssp-log-one-third-param-square-arg-component-bound)
  show eventually
     (λn. T n ≤
      bmssp-refined-graph-time-bound
       (λn. Suc D * sssp-log-one-third-param (n * n))
       (λn. sssp-log-two-thirds-param (n * n))
       (λn. sssp-log-one-third-param (n * n))
       (λn. sssp-log-one-third-param (n * n))
       (λn. sssp-log-two-thirds-param (n * n)) m n) at-top
    using cost-bound .
qed

theorem bmssp-refined-cost-bound-bigo-sssp-time-target-log-params-bounded-degree-
square-arg-slack:
  fixes D :: nat
  assumes Cn-pos: 0 < Cn
    and Cm-pos: 0 < Cm
    and vertex-count-dominated:
      eventually (λn. real (v n) ≤ Cn * real (m n)) at-top
    and edge-count-dominated:
      eventually (λn. real (m' n) ≤ Cm * real (m n)) at-top
    and cost-bound:
      eventually
       (λn. T n ≤
        bmssp-refined-graph-time-bound
         (λ-. Suc D * sssp-log-one-third-param (n * n))
         (λ-. sssp-log-two-thirds-param (n * n))
         (λ-. sssp-log-one-third-param (n * n))
         (λ-. sssp-log-one-third-param (n * n))
         (λ-. sssp-log-two-thirds-param (n * n))
         (λ-. m' n) (v n)) at-top
  shows (λn. real (T n)) ∈ O(λn. sssp-time-target m n)
proof (rule bmssp-refined-cost-bound-bigo-sssp-time-target-from-component-bounds-
slack
    [where Cn = Cn and Cm = Cm and Ca = 4 * real (Suc D)
      and Cl = 4 and Cr = 4 and Ct = 4 and Ch = 4])
  show 0 < Cn
    using Cn-pos .
  show 0 < Cm
    using Cm-pos .
```

show $0 < 4 * real\ (Suc\ D)$
by *simp*
show $0 \leq (4::real)$
by *simp*
show $0 < (4::real)$
by *simp*
show $0 < (4::real)$
by *simp*
show $0 < (4::real)$
by *simp*
show *eventually* $(\lambda n.\ real\ (v\ n) \leq Cn * real\ (m\ n))$ *at-top*
using *vertex-count-dominated* **.**
show *eventually* $(\lambda n.\ real\ (m'\ n) \leq Cm * real\ (m\ n))$ *at-top*
using *edge-count-dominated* **.**
show *eventually*
$(\lambda n.\ real\ (Suc\ D * sssp\text{-}log\text{-}one\text{-}third\text{-}param\ (n * n)) \leq$
$4 * real\ (Suc\ D) * sssp\text{-}log\text{-}factor\text{-}one\text{-}third\ n)$ *at-top*
using *sssp-log-one-third-param-square-arg-component-bound*
proof *eventually-elim*
case (*elim n*)
have $real\ (Suc\ D * sssp\text{-}log\text{-}one\text{-}third\text{-}param\ (n * n)) =$
$real\ (Suc\ D) * real\ (sssp\text{-}log\text{-}one\text{-}third\text{-}param\ (n * n))$
by (*simp add*: *algebra-simps*)
also have $\ldots \leq real\ (Suc\ D) *$
$(4 * sssp\text{-}log\text{-}factor\text{-}one\text{-}third\ n)$
using *elim* **by** *simp*
also have $\ldots = 4 * real\ (Suc\ D) * sssp\text{-}log\text{-}factor\text{-}one\text{-}third\ n$
by (*simp add*: *algebra-simps*)
finally show *?case* **.**
qed
show *eventually*
$(\lambda n.\ real\ (sssp\text{-}log\text{-}one\text{-}third\text{-}param\ (n * n)) \leq$
$4 * sssp\text{-}log\text{-}factor\text{-}one\text{-}third\ n)$ *at-top*
by (*rule sssp-log-one-third-param-square-arg-component-bound*)
show *eventually*
$(\lambda n.\ real\ (sssp\text{-}log\text{-}two\text{-}thirds\text{-}param\ (n * n)) \leq$
$4 * sssp\text{-}log\text{-}factor\ n)$ *at-top*
by (*rule sssp-log-two-thirds-param-square-arg-component-bound*)
show *eventually*
$(\lambda n.\ real\ (sssp\text{-}log\text{-}two\text{-}thirds\text{-}param\ (n * n)) \leq$
$4 * sssp\text{-}log\text{-}factor\ n)$ *at-top*
by (*rule sssp-log-two-thirds-param-square-arg-component-bound*)
show *eventually*
$(\lambda n.\ real\ (sssp\text{-}log\text{-}one\text{-}third\text{-}param\ (n * n)) \leq$
$4 * sssp\text{-}log\text{-}factor\text{-}one\text{-}third\ n)$ *at-top*
by (*rule sssp-log-one-third-param-square-arg-component-bound*)
show *eventually*
$(\lambda n.\ T\ n \leq$
bmssp-refined-graph-time-bound

```
      (λ-. Suc D * sssp-log-one-third-param (n * n))
      (λ-. sssp-log-two-thirds-param (n * n))
      (λ-. sssp-log-one-third-param (n * n))
      (λ-. sssp-log-one-third-param (n * n))
      (λ-. sssp-log-two-thirds-param (n * n))
      (λ-. m' n) (v n)) at-top
  using cost-bound .
qed

theorem bmssp-refined-cost-bound-bigo-sssp-time-target-log-params-bounded-degree-square-arg-real-slack:
  fixes D :: nat
  assumes Cn-pos: 0 < Cn
    and Ccost-pos: 0 < Ccost
    and vertex-count-dominated:
      eventually (λn. real n ≤ Cn * real (m n)) at-top
    and cost-bound:
      eventually
        (λn. real (T n) ≤ Ccost *
          real (bmssp-refined-graph-time-bound
            (λn. Suc D * sssp-log-one-third-param (n * n))
            (λn. sssp-log-two-thirds-param (n * n))
            (λn. sssp-log-one-third-param (n * n))
            (λn. sssp-log-one-third-param (n * n))
            (λn. sssp-log-two-thirds-param (n * n)) m n)) at-top
  shows (λn. real (T n)) ∈ O(λn. sssp-time-target m n)
proof −
  let ?G = λn. bmssp-refined-graph-time-bound
    (λn. Suc D * sssp-log-one-third-param (n * n))
    (λn. sssp-log-two-thirds-param (n * n))
    (λn. sssp-log-one-third-param (n * n))
    (λn. sssp-log-one-third-param (n * n))
    (λn. sssp-log-two-thirds-param (n * n)) m n
  have G-bigo: (λn. real (?G n)) ∈ O(λn. sssp-time-target m n)
    by (rule bmssp-refined-cost-bound-bigo-sssp-time-target-log-params-bounded-degree-square-arg
      [where D = D and T = ?G, OF Cn-pos vertex-count-dominated]) simp
  then obtain C where C-pos: 0 < C
    and G-bound:
      eventually
        (λn. norm (real (?G n)) ≤
          C * norm (sssp-time-target m n)) at-top
    unfolding bigo-def by blast
  have Cprod-pos: 0 < Ccost * C
    using Ccost-pos C-pos by simp
  show ?thesis
  proof (rule sssp-time-target-bigoI-norm[OF Cprod-pos])
    show eventually
      (λn. norm (real (T n)) ≤
```

```
      Ccost * C * norm (sssp-time-target m n)) at-top
    using cost-bound G-bound
  proof eventually-elim
    case (elim n)
    have norm (real (T n)) = real (T n)
      by simp
    also have ... ≤ Ccost * real (?G n)
      using elim(1) .
    also have ... = Ccost * norm (real (?G n))
      by simp
    also have ... ≤ Ccost * (C * norm (sssp-time-target m n))
      using elim(2) Ccost-pos by simp
    also have ... = Ccost * C * norm (sssp-time-target m n)
      by (simp add: algebra-simps)
    finally show ?case .
  qed
 qed
qed

theorem bmssp-refined-cost-bound-bigo-sssp-time-target-log-params-bounded-degree-
real-slack:
 fixes D :: nat
 assumes Cn-pos: 0 < Cn
  and Ccost-pos: 0 < Ccost
  and vertex-count-dominated:
    eventually (λn. real n ≤ Cn * real (m n)) at-top
  and cost-bound:
    eventually
      (λn. real (T n) ≤ Ccost *
        real (bmssp-refined-graph-time-bound
          (λn. Suc D * sssp-log-one-third-param n)
          sssp-log-two-thirds-param sssp-log-one-third-param
          sssp-log-one-third-param sssp-log-two-thirds-param m n)) at-top
 shows (λn. real (T n)) ∈ O(λn. sssp-time-target m n)
proof −
 let ?G = λn. bmssp-refined-graph-time-bound
   (λn. Suc D * sssp-log-one-third-param n)
   sssp-log-two-thirds-param sssp-log-one-third-param
   sssp-log-one-third-param sssp-log-two-thirds-param m n
 have G-bigo: (λn. real (?G n)) ∈ O(λn. sssp-time-target m n)
  by (rule bmssp-refined-cost-bound-bigo-sssp-time-target-log-params-bounded-degree
    [where D = D and T = ?G, OF Cn-pos vertex-count-dominated]) simp
 then obtain C where C-pos: 0 < C
  and G-bound:
    eventually
      (λn. norm (real (?G n)) ≤
        C * norm (sssp-time-target m n)) at-top
  unfolding bigo-def by blast
 have Cprod-pos: 0 < Ccost * C
```

```
    using Ccost-pos C-pos by simp
  show ?thesis
  proof (rule sssp-time-target-bigoI-norm[OF Cprod-pos])
    show eventually
      (λn. norm (real (T n)) ≤
        Ccost * C * norm (sssp-time-target m n)) at-top
      using cost-bound G-bound
    proof eventually-elim
      case (elim n)
      have norm (real (T n)) = real (T n)
        by simp
      also have ... ≤ Ccost * real (?G n)
        using elim(1) .
      also have ... = Ccost * norm (real (?G n))
        by simp
      also have ... ≤ Ccost * (C * norm (sssp-time-target m n))
        using elim(2) Ccost-pos by simp
      also have ... = Ccost * C * norm (sssp-time-target m n)
        by (simp add: algebra-simps)
      finally show ?case .
    qed
  qed
qed

context unique-shortest-digraph
begin
```

From this point on the arithmetic bounds are tied to a concrete finite graph. The definitions *vertex-count* and *edge-count* are simple wrappers around the locale's finite vertex and edge sets, but naming them keeps the final theorems independent of internal set notation. The parent-edge construction is used to relate reachable vertices to edges, which is the standard sparse graph move behind replacing a vertex term by an edge term when all vertices are reachable.

```
definition vertex-count where
  vertex-count = card V

definition edge-count where
  edge-count = card E

definition parent-edge-to where
  parent-edge-to v = (last (butlast (shortest-path-to v)), v)

lemma vertex-count-pos [simp]:
  0 < vertex-count
proof −
  have V ≠ {}
    using source-in-V by blast
  then show ?thesis
```

unfolding *vertex-count-def* **using** *finite-V* **by** (*simp add*: *card-gt-0-iff*)
qed

lemma *edge-count-le-vertex-count-square*:
edge-count $\leq$ *vertex-count* $*$ *vertex-count*
proof −
have *E-subset*: $E \subseteq V \times V$
using *edge-in-V* **by** *auto*
have *edge-count* = *card E*
unfolding *edge-count-def* **by** *simp*
also have ... $\leq$ *card* $(V \times V)$
by (*rule card-mono*[*OF - E-subset*]) (*use finite-V* **in** *simp*)
also have ... = *vertex-count* $*$ *vertex-count*
unfolding *vertex-count-def* **by** (*simp add*: *card-cartesian-product*)
finally show *?thesis* **.**
qed

lemma *parent-edge-to-in-E*:
assumes *reach-v*: *reachable s v*
and *v-ne*: $v \neq s$
shows *parent-edge-to v* $\in$ *E*
proof −
let *?p* = *shortest-path-to v*
have *sp*: *shortest-walk s ?p v*
by (*rule shortest-path-to-shortest*[*OF reach-v*])
have *walk-p*: *walk ?p*
using *sp* **unfolding** *shortest-walk-def simple-walk-betw-def walk-betw-def* **by** *blast*
have *p-ne*: $?p \neq []$
using *sp* **unfolding** *shortest-walk-def simple-walk-betw-def walk-betw-def* **by** *blast*
have *hd-p*: *hd ?p* = *s*
using *sp* **unfolding** *shortest-walk-def simple-walk-betw-def walk-betw-def* **by** *blast*
have *last-p*: *last ?p* = *v*
using *sp* **unfolding** *shortest-walk-def simple-walk-betw-def walk-betw-def* **by** *blast*
have *len-ge2*: *1* < *length ?p*
proof (*rule ccontr*)
assume ¬ *1* < *length ?p*
then have *length ?p* = *1*
using *p-ne* **by** (*cases ?p*) *auto*
then have *hd ?p* = *last ?p*
by (*cases ?p*) *auto*
then show *False*
using *hd-p last-p v-ne* **by** *simp*
qed
have *but-ne*: *butlast ?p* $\neq$ []
using *len-ge2* **by** (*cases ?p*) *auto*

```
  obtain q u where but: butlast ?p = q @ [u]
    using but-ne by (cases butlast ?p rule: rev-cases) auto
  have p-decomp: ?p = q @ [u, v]
  proof −
    have ?p = butlast ?p @ [last ?p]
      using p-ne by simp
    then show ?thesis
      using but last-p by simp
  qed
  have i-lt: Suc (length q) < length ?p
    using p-decomp by simp
  have edge: (?p ! length q, ?p ! Suc (length q)) ∈ E
    by (rule walk-nth-edge[OF walk-p i-lt])
  have ?p ! length q = u
    using p-decomp by (simp add: nth-append)
  moreover have ?p ! Suc (length q) = v
    using p-decomp by (simp add: nth-append)
  ultimately have (u, v) ∈ E
    using edge by simp
  moreover have last (butlast ?p) = u
    using but by simp
  ultimately show ?thesis
    unfolding parent-edge-to-def by simp
qed

theorem vertex-count-le-Suc-edge-count-if-all-reachable:
  assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
  shows vertex-count ≤ Suc edge-count
proof −
  have image-subset: parent-edge-to ‘ (V − {s}) ⊆ E
  proof
    fix e
    assume e ∈ parent-edge-to ‘ (V − {s})
    then obtain v where v: v ∈ V v ≠ s e = parent-edge-to v
      by blast
    have reachable s v
      by (rule all-reachable[OF v(1)])
    then show e ∈ E
      using parent-edge-to-in-E[OF - v(2)] v(3) by simp
  qed
  have inj: inj-on parent-edge-to (V − {s})
    unfolding parent-edge-to-def inj-on-def by auto
  have card-image:
    card (V − {s}) = card (parent-edge-to ‘ (V − {s}))
    using inj finite-V by (simp add: card-image)
  also have ... ≤ edge-count
    unfolding edge-count-def
    by (rule card-mono[OF finite-E image-subset])
  finally have without-s: card (V − {s}) ≤ edge-count .
```

```
  have V-decomp: V = insert s (V − {s})
    using source-in-V by blast
  have card V = card (insert s (V − {s}))
    using V-decomp by simp
  also have ... = Suc (card (V − {s}))
    by (rule card-insert-disjoint) (use finite-V in auto)
  finally have card V = Suc (card (V − {s})) .
  then show ?thesis
    using without-s unfolding vertex-count-def by simp
qed

corollary vertex-count-le-twice-edge-count-if-all-reachable:
  assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
    and nontrivial: 0 < edge-count
  shows vertex-count ≤ 2 * edge-count
proof −
  have vertex-count ≤ Suc edge-count
    by (rule vertex-count-le-Suc-edge-count-if-all-reachable[OF all-reachable])
  also have ... ≤ 2 * edge-count
    using nontrivial by simp
  finally show ?thesis .
qed

lemma card-subset-vertex-count:
  assumes S ⊆ V
  shows card S ≤ vertex-count
  unfolding vertex-count-def
  by (rule card-mono[OF finite-V assms])

lemma eventually-top-level-threshold-exceeds-vertex-count:
  eventually
    (λN. vertex-count <
      sssp-log-one-third-param N *
      bmssp-level-cap (sssp-log-one-third-param N)
        (sssp-log-two-thirds-param N) (sssp-log-one-third-param N))
    at-top
  using eventually-less-sssp-log-one-third-param[of vertex-count]
proof eventually-elim
  case (elim N)
  let ?p = sssp-log-one-third-param N
  let ?q = sssp-log-two-thirds-param N
  let ?cap = bmssp-level-cap ?p ?q ?p
  have cap-pos: 0 < ?cap
    unfolding bmssp-level-cap-def by simp
  have ?p ≤ ?p * ?cap
    using cap-pos by (cases ?cap) simp-all
  then show ?case
    using elim by linarith
qed
```

lemma *find-pivots-pivots-capped-singleton-empty-if-params-exceed-vertex-count*:
  assumes *k-gt*: *vertex-count* < *k*
    and *cap-ge*: *vertex-count* ≤ *cap*
  shows *find-pivots-pivots-capped k cap d* {*s*} *B* = {}
proof −
  let *?seen* = *find-pivots-seen-capped k cap d* {*s*} *B*
  have *source-subset*: {*s*} ⊆ *V*
    using *source-in-V* **by** *simp*
  have *seen-subset*: *?seen* ⊆ *V*
    unfolding *find-pivots-seen-capped-def*
    by (*rule fp-iter-capped-seen-subset-V*[*OF source-subset source-subset*])
  have *card-seen-le*: *card ?seen* ≤ *vertex-count*
    by (*rule card-subset-vertex-count*[*OF seen-subset*])
  have *no-overflow*: ¬ *card ?seen* > *cap*
    using *card-seen-le cap-ge* **by** *linarith*
  have *no-large-tree*:
    {*u* ∈ {*s*}. *k* ≤ *card* (*tree-of u* ∩ *?seen*)} = {}
  proof
    show {*u* ∈ {*s*}. *k* ≤ *card* (*tree-of u* ∩ *?seen*)} ⊆ {}
    proof
      fix *u*
      assume *u-mem*: *u* ∈ {*u* ∈ {*s*}. *k* ≤ *card* (*tree-of u* ∩ *?seen*)}
      have *tree-seen-subset*: *tree-of u* ∩ *?seen* ⊆ *V*
        unfolding *tree-of-def* **by** *blast*
      have *card* (*tree-of u* ∩ *?seen*) ≤ *vertex-count*
        by (*rule card-subset-vertex-count*[*OF tree-seen-subset*])
      moreover have *k* ≤ *card* (*tree-of u* ∩ *?seen*)
        using *u-mem* **by** *blast*
      ultimately show *u* ∈ {}
        using *k-gt* **by** *linarith*
    qed
  qed *simp*
  show *?thesis*
    unfolding *find-pivots-pivots-capped-def*
    using *no-overflow no-large-tree* **by** *simp*
qed

lemma *eventually-top-level-pivots-empty*:
  eventually
    (λ*N*. *find-pivots-pivots-capped* (*sssp-log-one-third-param N*)
      (*bmssp-level-cap* (*sssp-log-one-third-param N*)
        (*sssp-log-two-thirds-param N*) (*sssp-log-one-third-param N*))
      d {*s*} *B* = {})
    at-top
  using *eventually-less-sssp-log-one-third-param*[*of vertex-count*]
proof *eventually-elim*
  case (*elim N*)
  let *?p* = *sssp-log-one-third-param N*

```
  let ?q = sssp-log-two-thirds-param N
  let ?cap = bmssp-level-cap ?p ?q ?p
  have cap-ge-p: ?p ≤ ?cap
    by (rule bmssp-level-cap-ge-level)
  have cap-ge-vertex: vertex-count ≤ ?cap
    using elim cap-ge-p by linarith
  show ?case
    by (rule find-pivots-pivots-capped-singleton-empty-if-params-exceed-vertex-count
      [OF elim cap-ge-vertex])
qed

lemma edge-count-outgoing-bound:
  card (outgoing-edges X) ≤ edge-count
proof −
  have outgoing-edges X ⊆ E
    unfolding outgoing-edges-def by blast
  then show ?thesis
    unfolding edge-count-def by (rule card-mono[OF finite-E])
qed

lemma edge-count-outgoing-bound-tree:
  card (outgoing-edges (bound-tree S B)) ≤ edge-count
  by (rule edge-count-outgoing-bound)
```

The range-based edge definitions are the combinatorial support for the direct-insert and lower-batch split. An edge is charged to a range according to the relaxed value at its source, not merely according to the source vertex. This matches the algorithm: a completed child range may generate relaxations that fall below the pull bound, inside the parent output bound, or outside the current problem.

The length lemmas connect executable lists of relaxation pairs to the mathematical edge ranges. The disjointness and split lemmas that follow make it possible to sum per-child edge charges and bound them by the outgoing edges of the assembled parent range.

```
definition outgoing-edges-range where
  outgoing-edges-range U a B =
    {(u, v) ∈ E. u ∈ U ∧ a ≤ dist s u + w u v ∧
      below-bound (dist s u + w u v) B}

lemma outgoing-edges-range-subset-E:
  outgoing-edges-range U a B ⊆ E
  unfolding outgoing-edges-range-def by blast

lemma outgoing-edges-range-subset-outgoing-edges:
  outgoing-edges-range U a B ⊆ outgoing-edges U
  unfolding outgoing-edges-range-def outgoing-edges-def by blast

lemma finite-outgoing-edges-range [simp]:
```

```
  finite (outgoing-edges-range U a B)
  by (rule finite-subset[OF outgoing-edges-range-subset-E finite-E])

lemma card-outgoing-edges-range-le-edge-count:
  card (outgoing-edges-range U a B) ≤ edge-count
  unfolding edge-count-def
  by (rule card-mono[OF finite-E outgoing-edges-range-subset-E])

definition edge-relaxation-pairs-in-bound where
  edge-relaxation-pairs-in-bound d U L B =
    map (λ(u, v). (v, d u + w u v))
      (filter (λ(u, v). L ≤ d u + w u v ∧
        below-bound (d u + w u v) B)
       (edge-list-of (outgoing-edges U)))

lemma edge-relaxation-pairs-in-bound-value-ge-low:
  assumes (x, b) ∈ set (edge-relaxation-pairs-in-bound d U L B)
  shows L ≤ b
  using assms unfolding edge-relaxation-pairs-in-bound-def by auto

lemma edge-relaxation-pairs-in-bound-value-below-bound:
  assumes (x, b) ∈ set (edge-relaxation-pairs-in-bound d U L B)
  shows below-bound b B
  using assms unfolding edge-relaxation-pairs-in-bound-def by auto

lemma edge-relaxation-pairs-in-bound-length-le-outgoing-edges-range:
  assumes complete: complete-on d U
    and reaches: ⋀u. u ∈ U ⟹ reachable s u
  shows length (edge-relaxation-pairs-in-bound d U L B) ≤
    card (outgoing-edges-range U L B)
proof −
  let ?es = edge-list-of (outgoing-edges U)
  let ?fs =
    filter (λ(u, v). L ≤ d u + w u v ∧
      below-bound (d u + w u v) B) ?es
  have set-es: set ?es = outgoing-edges U
    using edge-list-of-properties(1)[OF finite-outgoing-edges] .
  have distinct-fs: distinct ?fs
    using edge-list-of-properties(2)[OF finite-outgoing-edges] by simp
  have subset:
    set ?fs ⊆ outgoing-edges-range U L B
  proof
    fix e
    assume e: e ∈ set ?fs
    then obtain u v where ev: e = (u, v)
      by (metis surj-pair)
    have edge: (u, v) ∈ E
      and uU: u ∈ U
      and low: L ≤ d u + w u v
```

```
    and high: below-bound (d u + w u v) B
    using e set-es unfolding ev outgoing-edges-def by auto
  have du: d u = dist s u
    using complete reaches[OF uU] uU unfolding complete-on-def by blast
  show e ∈ outgoing-edges-range U L B
    using edge uU low high unfolding ev du outgoing-edges-range-def by simp
 qed
 have length (edge-relaxation-pairs-in-bound d U L B) = length ?fs
  unfolding edge-relaxation-pairs-in-bound-def by simp
 also have ... = card (set ?fs)
  using distinct-card[OF distinct-fs] by simp
 also have ... ≤ card (outgoing-edges-range U L B)
  by (rule card-mono[OF finite-outgoing-edges-range subset])
 finally show ?thesis .
qed

lemma partition-batch-cost-bound-edge-relaxation-pairs-in-bound:
 assumes cost:
    partition-batch-cost-bound c t
     (edge-relaxation-pairs-in-bound d U L B)
  and complete: complete-on d U
  and reaches: ⋀u. u ∈ U ⟹ reachable s u
 shows c ≤ t * card (outgoing-edges-range U L B)
proof −
 have len-le: length (edge-relaxation-pairs-in-bound d U L B) ≤
    card (outgoing-edges-range U L B)
  by (rule edge-relaxation-pairs-in-bound-length-le-outgoing-edges-range
    [OF complete reaches])
 have c ≤ t * length (edge-relaxation-pairs-in-bound d U L B)
  using cost unfolding partition-batch-cost-bound-def by simp
 also have ... ≤ t * card (outgoing-edges-range U L B)
  using len-le by simp
 finally show ?thesis .
qed

lemma edge-relaxation-pairs-between-length-le-outgoing-edges-range:
 assumes complete: complete-on d U
  and reaches: ⋀u. u ∈ U ⟹ reachable s u
 shows length (edge-relaxation-pairs-between d U L H) ≤
  card (outgoing-edges-range U L (Fin H))
proof −
 let ?es = edge-list-of (outgoing-edges U)
 let ?fs =
  filter (λ(u, v). L ≤ d u + w u v ∧ d u + w u v < H) ?es
 have set-es: set ?es = outgoing-edges U
  using edge-list-of-properties(1)[OF finite-outgoing-edges] .
 have distinct-fs: distinct ?fs
  using edge-list-of-properties(2)[OF finite-outgoing-edges] by simp
 have subset:
```

```
    set ?fs ⊆ outgoing-edges-range U L (Fin H)
  proof
    fix e
    assume e: e ∈ set ?fs
    then obtain u v where ev: e = (u, v)
      by (metis surj-pair)
    have edge: (u, v) ∈ E
      and uU: u ∈ U
      and low: L ≤ d u + w u v
      and high: d u + w u v < H
      using e set-es unfolding ev outgoing-edges-def by auto
    have du: d u = dist s u
      using complete reaches[OF uU] uU unfolding complete-on-def by blast
    show e ∈ outgoing-edges-range U L (Fin H)
      using edge uU low high unfolding ev du outgoing-edges-range-def by simp
  qed
  have length (edge-relaxation-pairs-between d U L H) = length ?fs
    unfolding edge-relaxation-pairs-between-def by simp
  also have ... = card (set ?fs)
    using distinct-card[OF distinct-fs] by simp
  also have ... ≤ card (outgoing-edges-range U L (Fin H))
    by (rule card-mono[OF finite-outgoing-edges-range subset])
  finally show ?thesis .
qed

lemma edge-relaxation-pairs-between-length-le-range-tree-outgoing-edges-range:
  assumes complete: complete-on d (range-tree S a B)
  shows length (edge-relaxation-pairs-between d (range-tree S a B) L H) ≤
    card (outgoing-edges-range (range-tree S a B) L (Fin H))
proof (rule edge-relaxation-pairs-between-length-le-outgoing-edges-range[OF com-
plete])
  fix u
  assume u ∈ range-tree S a B
  then show reachable s u
    unfolding range-tree-def tree-set-def tree-path-def by blast
qed

lemma outgoing-edges-range-mono-sources:
  assumes U ⊆ W
  shows outgoing-edges-range U a B ⊆ outgoing-edges-range W a B
  using assms unfolding outgoing-edges-range-def by blast

lemma outgoing-edges-range-split:
  assumes a-le-b: a ≤ b
    and b-le-B: bound-le (Fin b) B
  shows outgoing-edges-range U a B =
    outgoing-edges-range U a (Fin b) ∪ outgoing-edges-range U b B
  using assms unfolding outgoing-edges-range-def
  by (cases B) auto
```

lemma *outgoing-edges-range-split-disjoint*:
outgoing-edges-range U a (Fin b) ∩ outgoing-edges-range U b B = {}
unfolding *outgoing-edges-range-def* **by** *auto*

lemma *card-outgoing-edges-range-split*:
assumes *a-le-b*: *a ≤ b*
and *b-le-B*: *bound-le (Fin b) B*
shows *card (outgoing-edges-range U a B) =*
card (outgoing-edges-range U a (Fin b)) +
card (outgoing-edges-range U b B)
proof −
have *split*:
outgoing-edges-range U a B =
outgoing-edges-range U a (Fin b) ∪ outgoing-edges-range U b B
by (*rule outgoing-edges-range-split*[*OF a-le-b b-le-B*])
have *disj*:
outgoing-edges-range U a (Fin b) ∩ outgoing-edges-range U b B = {}
by (*rule outgoing-edges-range-split-disjoint*)
show *?thesis*
unfolding *split*
by (*rule card-Un-disjoint*) (*use disj* **in** *auto*)
qed

lemma *card-outgoing-edges-range-lower-direct-split*:
assumes *b-le-beta*: *b ≤ beta*
and *beta-le-B*: *bound-le (Fin beta) B*
shows *card (outgoing-edges-range U b (Fin beta)) +*
card (outgoing-edges-range U beta B) =
card (outgoing-edges-range U b B)
using *card-outgoing-edges-range-split*[*OF b-le-beta beta-le-B*, *of U*]
by *simp*

lemma *card-outgoing-edges-range-zero-lower-direct-split-le*:
assumes *beta-le-B*: *bound-le (Fin beta) B*
and *nonneg*:
⋀u v. ⟦(u, v) ∈ E; u ∈ U⟧ ⟹ 0 ≤ dist s u + w u v
shows *card (outgoing-edges-range U 0 (Fin beta)) +*
card (outgoing-edges-range U beta B) ≤
card (outgoing-edges-range U 0 B)
proof −
let *?lower = outgoing-edges-range U 0 (Fin beta)*
let *?direct = outgoing-edges-range U beta B*
let *?full = outgoing-edges-range U 0 B*
have *disj*: *?lower ∩ ?direct = {}*
by (*rule outgoing-edges-range-split-disjoint*)
have *union-subset*: *?lower ∪ ?direct ⊆ ?full*
proof
fix *e*

```
    assume e: e ∈ ?lower ∪ ?direct
    show e ∈ ?full
    proof (cases e ∈ ?lower)
      case True
      then show ?thesis
        using beta-le-B unfolding outgoing-edges-range-def
        by (cases B) auto
    next
      case False
      then have direct: e ∈ ?direct
        using e by blast
      then obtain u v where e-uv: e = (u, v)
        by (metis surj-pair)
      have edge: (u, v) ∈ E and uU: u ∈ U
        using direct unfolding e-uv outgoing-edges-range-def by auto
      have 0 ≤ dist s u + w u v
        by (rule nonneg[OF edge uU])
      then show ?thesis
        using direct unfolding e-uv outgoing-edges-range-def by auto
    qed
  qed
  have card-union:
    card (?lower ∪ ?direct) = card ?lower + card ?direct
    using disj by (simp add: card-Un-disjoint)
  have card (?lower ∪ ?direct) ≤ card ?full
    by (rule card-mono[OF finite-outgoing-edges-range union-subset])
  then show ?thesis
    using card-union by linarith
qed

lemma outgoing-edges-range-disjoint-sources:
  assumes U ∩ W = {}
  shows outgoing-edges-range U a B ∩ outgoing-edges-range W c D = {}
  using assms unfolding outgoing-edges-range-def by blast

fun pairwise-disjoint-list where
  pairwise-disjoint-list [] ⟷ True
| pairwise-disjoint-list (X # Xs) ⟷
    (∀ Y∈set Xs. X ∩ Y = {}) ∧ pairwise-disjoint-list Xs

lemma pairwise-disjoint-list-outgoing-edges-range:
  assumes pairwise-disjoint-list Us
  shows pairwise-disjoint-list (map (λU. outgoing-edges-range U a B) Us)
  using assms
proof (induction Us)
  case Nil
  then show ?case by simp
next
  case (Cons U Us)
```

```
  have disj-head:
    ∀ Y∈set (map (λU. outgoing-edges-range U a B) Us).
      outgoing-edges-range U a B ∩ Y = {}
  proof
    fix Y
    assume Y ∈ set (map (λU. outgoing-edges-range U a B) Us)
    then obtain W where W: W ∈ set Us Y = outgoing-edges-range W a B
      by auto
    have U ∩ W = {}
      using Cons.prems W(1) by simp
    then show outgoing-edges-range U a B ∩ Y = {}
      unfolding W(2) by (rule outgoing-edges-range-disjoint-sources)
  qed
  have tail: pairwise-disjoint-list (map (λU. outgoing-edges-range U a B) Us)
    using Cons.IH Cons.prems by simp
  show ?case
    using disj-head tail by simp
qed

lemma edge-outdegree-le-edge-count:
  edge-outdegree-le edge-count
  unfolding edge-outdegree-le-def
  using edge-count-outgoing-bound by blast

lemma card-Union-pairwise-disjoint-list:
  assumes disj: pairwise-disjoint-list Xs
    and finite-sets: ⋀X. X ∈ set Xs ⟹ finite X
  shows card (⋃(set Xs)) = sum-list (map card Xs)
  using assms
proof (induction Xs)
  case Nil
  then show ?case by simp
next
  case (Cons X Xs)
  have finite-X: finite X
    using Cons.prems by simp
  have finite-tail: finite (⋃(set Xs))
    using Cons.prems by auto
  have disjoint-tail: X ∩ ⋃(set Xs) = {}
    using Cons.prems by auto
  have tail:
    card (⋃(set Xs)) = sum-list (map card Xs)
    using Cons.IH Cons.prems by auto
  have card (⋃(set (X # Xs))) =
      card X + card (⋃(set Xs))
    by (simp add: card-Un-disjoint finite-X finite-tail disjoint-tail)
  then show ?case
    using tail by simp
qed
```

lemma *sum-card-outgoing-edges-range-disjoint-sources-le-edge-count*:
assumes *disj*: *pairwise-disjoint-list Us*
shows *sum-list* (*map* (λ*U*. *card* (*outgoing-edges-range U a B*)) *Us*) ≤ *edge-count*
proof −
have *disj-ranges*:
pairwise-disjoint-list (*map* (λ*U*. *outgoing-edges-range U a B*) *Us*)
by (*rule pairwise-disjoint-list-outgoing-edges-range*[*OF disj*])
have *finite-ranges*:
⋀*X*. *X* ∈ *set* (*map* (λ*U*. *outgoing-edges-range U a B*) *Us*) ⟹ *finite X*
by *auto*
have *card-union*:
card (⋃(*set* (*map* (λ*U*. *outgoing-edges-range U a B*) *Us*))) =
sum-list (*map card* (*map* (λ*U*. *outgoing-edges-range U a B*) *Us*))
by (*rule card-Union-pairwise-disjoint-list*[*OF disj-ranges finite-ranges*])
have *union-subset*:
⋃(*set* (*map* (λ*U*. *outgoing-edges-range U a B*) *Us*)) ⊆ *E*
proof
fix *e*
assume *e* ∈ ⋃(*set* (*map* (λ*U*. *outgoing-edges-range U a B*) *Us*))
then obtain *U* **where** *U* ∈ *set Us* **and** *e* ∈ *outgoing-edges-range U a B*
by *auto*
then show *e* ∈ *E*
unfolding *outgoing-edges-range-def* **by** *auto*
qed
have *sum-maps*:
sum-list (*map* (λ*U*. *card* (*outgoing-edges-range U a B*)) *Us*) =
sum-list (*map card* (*map* (λ*U*. *outgoing-edges-range U a B*) *Us*))
by (*induction Us*) *simp-all*
have *sum-list* (*map* (λ*U*. *card* (*outgoing-edges-range U a B*)) *Us*) =
card (⋃(*set* (*map* (λ*U*. *outgoing-edges-range U a B*) *Us*)))
using *sum-maps card-union* **by** *simp*
also have ... ≤ *edge-count*
unfolding *edge-count-def* **by** (*rule card-mono*[*OF finite-E union-subset*])
finally show *?thesis* **.**
qed

lemma *outgoing-edges-disjoint-if-sources-disjoint*:
assumes *A* ∩ *B* = {}
shows *outgoing-edges A* ∩ *outgoing-edges B* = {}
using *assms* **unfolding** *outgoing-edges-def* **by** *auto*

lemma *pairwise-disjoint-list-outgoing-edges*:
assumes *pairwise-disjoint-list Xs*
shows *pairwise-disjoint-list* (*map outgoing-edges Xs*)
using *assms*
proof (*induction Xs*)
case *Nil*
then show *?case* **by** *simp*

```
next
  case (Cons X Xs)
  have ⋀Y. Y ∈ set Xs ⟹ outgoing-edges X ∩ outgoing-edges Y = {}
    using Cons.prems outgoing-edges-disjoint-if-sources-disjoint by auto
  then show ?case
    using Cons by auto
qed

lemma outgoing-edges-Union-list:
  outgoing-edges (⋃(set Xs)) = ⋃(set (map outgoing-edges Xs))
  unfolding outgoing-edges-def by auto

lemma outgoing-edges-mono:
  assumes A ⊆ B
  shows outgoing-edges A ⊆ outgoing-edges B
  using assms unfolding outgoing-edges-def by blast

fun range-tree-child-list where
  range-tree-child-list S a [] = []
| range-tree-child-list S a (b # bs) =
    range-tree S a (Fin b) # range-tree-child-list S b bs

lemma length-range-tree-child-list [simp]:
  length (range-tree-child-list S a bs) = length bs
  by (induction bs arbitrary: a) simp-all

lemma range-tree-child-list-set-subset-chain-list:
  set (range-tree-child-list S a bs) ⊆ set (range-tree-chain-list S a bs B)
proof (induction bs arbitrary: a)
  case Nil
  then show ?case by simp
next
  case (Cons b bs)
  have set (range-tree-child-list S b bs) ⊆
      set (range-tree-chain-list S b bs B)
    using Cons.IH .
  then show ?case by auto
qed

lemma Union-range-tree-child-list-subset-chain:
  ⋃(set (range-tree-child-list S a bs)) ⊆ range-tree-chain S a bs B
proof (induction bs arbitrary: a)
  case Nil
  then show ?case by simp
next
  case (Cons b bs)
  have ⋃(set (range-tree-child-list S b bs)) ⊆
      range-tree-chain S b bs B
    using Cons.IH .
```

```
  then show ?case by auto
qed

fun range-tree-child-edge-range-list where
  range-tree-child-edge-range-list S a [] [] = []
| range-tree-child-edge-range-list S a [] (b # bs) = []
| range-tree-child-edge-range-list S a (beta # betas) [] = []
| range-tree-child-edge-range-list S a (beta # betas) (b # bs) =
    outgoing-edges-range (range-tree S a (Fin b)) b (Fin beta) #
    range-tree-child-edge-range-list S b betas bs

lemma range-tree-child-edge-range-list-subset-outgoing-sources:
  assumes Y ∈ set (range-tree-child-edge-range-list S a betas bs)
  shows ∃ X∈set (range-tree-child-list S a bs). Y ⊆ outgoing-edges X
  using assms
proof (induction betas arbitrary: a bs)
  case Nil
  then show ?case
    by (cases bs) simp-all
next
  case (Cons beta betas)
  show ?case
  proof (cases bs)
    case Nil
    then show ?thesis
      using Cons.prems by simp
  next
    case (Cons b bs')
    let ?head = outgoing-edges-range (range-tree S a (Fin b)) b (Fin beta)
    let ?tail = range-tree-child-edge-range-list S b betas bs'
    have Y-cases: Y = ?head ∨ Y ∈ set ?tail
      using Cons.prems Cons by simp
    then show ?thesis
    proof
      assume Y-head: Y = ?head
      have ?head ⊆ outgoing-edges (range-tree S a (Fin b))
        by (rule outgoing-edges-range-subset-outgoing-edges)
      then show ?thesis
        unfolding Cons Y-head by auto
    next
      assume Y-tail: Y ∈ set ?tail
      obtain X where X: X ∈ set (range-tree-child-list S b bs')
        and YX: Y ⊆ outgoing-edges X
        using Cons.IH[OF Y-tail] by blast
      show ?thesis
        using X YX unfolding Cons by auto
    qed
  qed
qed
```

lemma *range-tree-child-edge-range-list-member-subset-E*:
assumes $Y \in$ *set* (*range-tree-child-edge-range-list S a betas bs*)
shows $Y \subseteq E$
proof
fix *e*
assume *eY*: $e \in Y$
obtain *X* **where** $X \in$ *set* (*range-tree-child-list S a bs*)
and *Y-subset*: $Y \subseteq$ *outgoing-edges X*
using *range-tree-child-edge-range-list-subset-outgoing-sources*[*OF assms*]
by *blast*
then have $e \in$ *outgoing-edges X*
using *eY* **by** *blast*
then show $e \in E$
unfolding *outgoing-edges-def* **by** *blast*
qed

lemma *pairwise-disjoint-list-range-tree-child-edge-range-list*:
assumes *mono*: *nondecreasing-from a bs*
shows *pairwise-disjoint-list* (*range-tree-child-edge-range-list S a betas bs*)
using *mono*
proof (*induction betas arbitrary*: *a bs*)
case *Nil*
then show *?case*
by (*cases bs*) *simp-all*
next
case (*Cons beta betas*)
show *?case*
proof (*cases bs*)
case *Nil*
then show *?thesis* **by** *simp*
next
case (*Cons b bs'*)
let *?head* = *outgoing-edges-range* (*range-tree S a* (*Fin b*)) *b* (*Fin beta*)
let *?tail* = *range-tree-child-edge-range-list S b betas bs'*
have *a-le-b*: $a \leq b$
using *Cons.prems Cons* **by** *simp*
have *mono-tail*: *nondecreasing-from b bs'*
using *Cons.prems Cons* **by** *simp*
have *head-tail-disj*: $\forall\, Y \in$*set ?tail*. *?head* $\cap\ Y = \{\}$
proof
fix *Y*
assume *Y*: $Y \in$ *set ?tail*
obtain *X* **where** *X*: $X \in$ *set* (*range-tree-child-list S b bs'*)
and *YX*: $Y \subseteq$ *outgoing-edges X*
using *range-tree-child-edge-range-list-subset-outgoing-sources*[*OF Y*]
by *blast*
have *X-subset*: $X \subseteq$ *range-tree-chain S b bs' Infinity*
using *X Union-range-tree-child-list-subset-chain*[*of S b bs' Infinity*]

```
      by blast
    have source-disj: range-tree S a (Fin b) ∩ X = {}
      using range-tree-disjoint-range-tree-chain-tail
        [OF a-le-b mono-tail, of S Infinity] X-subset by blast
    have out-disj:
      outgoing-edges (range-tree S a (Fin b)) ∩ outgoing-edges X = {}
      by (rule outgoing-edges-disjoint-if-sources-disjoint[OF source-disj])
    have head-subset: ?head ⊆ outgoing-edges (range-tree S a (Fin b))
      by (rule outgoing-edges-range-subset-outgoing-edges)
    show ?head ∩ Y = {}
      using head-subset YX out-disj by blast
  qed
  have tail-disj: pairwise-disjoint-list ?tail
    by (rule Cons.IH[OF mono-tail])
  show ?thesis
    unfolding Cons by (simp add: head-tail-disj tail-disj)
 qed
qed

lemma sum-card-range-tree-child-edge-range-list-le-edge-count:
 assumes mono: nondecreasing-from a bs
 shows sum-list (map card (range-tree-child-edge-range-list S a betas bs))
   ≤ edge-count
proof −
 let ?Xs = range-tree-child-edge-range-list S a betas bs
 have disj: pairwise-disjoint-list ?Xs
   by (rule pairwise-disjoint-list-range-tree-child-edge-range-list[OF mono])
 have finite-sets: ⋀X. X ∈ set ?Xs ⟹ finite X
   using range-tree-child-edge-range-list-member-subset-E
   by (meson finite-E finite-subset)
 have card-union:
   card (⋃(set ?Xs)) = sum-list (map card ?Xs)
   by (rule card-Union-pairwise-disjoint-list[OF disj finite-sets])
 have union-subset: ⋃(set ?Xs) ⊆ E
 proof
   fix e
   assume e ∈ ⋃(set ?Xs)
   then obtain X where X: X ∈ set ?Xs and eX: e ∈ X
     by blast
   show e ∈ E
     using range-tree-child-edge-range-list-member-subset-E[OF X] eX by blast
 qed
 have sum-list (map card ?Xs) = card (⋃(set ?Xs))
   using card-union by simp
 also have ... ≤ edge-count
   unfolding edge-count-def by (rule card-mono[OF finite-E union-subset])
 finally show ?thesis .
qed
```

lemma *sum-card-range-tree-child-edge-range-list-le-outgoing-edges-chain*:
  assumes *mono*: *nondecreasing-from a bs*
  shows *sum-list* (*map card* (*range-tree-child-edge-range-list S a betas bs*))
    ≤ *card* (*outgoing-edges* (*range-tree-chain S a bs B*))
proof −
  let *?Rs* = *range-tree-child-edge-range-list S a betas bs*
  let *?Xs* = *range-tree-child-list S a bs*
  have *disj*: *pairwise-disjoint-list ?Rs*
    by (*rule pairwise-disjoint-list-range-tree-child-edge-range-list*[*OF mono*])
  have *finite-sets*: ⋀*R*. *R* ∈ *set ?Rs* ⟹ *finite R*
    using *range-tree-child-edge-range-list-member-subset-E*
    by (*meson finite-E finite-subset*)
  have *card-union*:
    card (⋃(*set ?Rs*)) = *sum-list* (*map card ?Rs*)
    by (*rule card-Union-pairwise-disjoint-list*[*OF disj finite-sets*])
  have *union-subset-children*: ⋃(*set ?Rs*) ⊆ *outgoing-edges* (⋃(*set ?Xs*))
  proof
    fix *e*
    assume *e* ∈ ⋃(*set ?Rs*)
    then obtain *R* **where** *R*: *R* ∈ *set ?Rs* **and** *eR*: *e* ∈ *R*
      by *blast*
    obtain *X* **where** *X*: *X* ∈ *set ?Xs* **and** *R-subset*: *R* ⊆ *outgoing-edges X*
      using *range-tree-child-edge-range-list-subset-outgoing-sources*[*OF R*]
      by *blast*
    show *e* ∈ *outgoing-edges* (⋃(*set ?Xs*))
      using *eR R-subset X* **unfolding** *outgoing-edges-def* **by** *blast*
  qed
  have *children-subset*: ⋃(*set ?Xs*) ⊆ *range-tree-chain S a bs B*
    by (*rule Union-range-tree-child-list-subset-chain*)
  have *union-subset*:
    ⋃(*set ?Rs*) ⊆ *outgoing-edges* (*range-tree-chain S a bs B*)
    using *union-subset-children outgoing-edges-mono*[*OF children-subset*] **by** *blast*
  have *sum-list* (*map card ?Rs*) = *card* (⋃(*set ?Rs*))
    using *card-union* **by** *simp*
  also have ... ≤ *card* (*outgoing-edges* (*range-tree-chain S a bs B*))
    by (*rule card-mono*) (*simp-all add*: *union-subset*)
  finally show *?thesis* **.**
qed

fun *range-tree-child-direct-edge-range-list* **where**
  range-tree-child-direct-edge-range-list S B a [] [] = []
| *range-tree-child-direct-edge-range-list S B a* [] (*b* # *bs*) = []
| *range-tree-child-direct-edge-range-list S B a* (*beta* # *betas*) [] = []
| *range-tree-child-direct-edge-range-list S B a* (*beta* # *betas*) (*b* # *bs*) =
    outgoing-edges-range (*range-tree S a* (*Fin b*)) *beta B* #
    range-tree-child-direct-edge-range-list S B b betas bs

fun *range-tree-child-zero-edge-range-list* **where**
  range-tree-child-zero-edge-range-list S a [] [] = []

```
| range-tree-child-zero-edge-range-list S a [] (b # bs) = []
| range-tree-child-zero-edge-range-list S a (beta # betas) [] = []
| range-tree-child-zero-edge-range-list S a (beta # betas) (b # bs) =
    outgoing-edges-range (range-tree S a (Fin b)) 0 (Fin beta) #
    range-tree-child-zero-edge-range-list S b betas bs

fun range-tree-child-full-edge-range-list where
  range-tree-child-full-edge-range-list S B a [] = []
| range-tree-child-full-edge-range-list S B a (b # bs) =
    outgoing-edges-range (range-tree S a (Fin b)) 0 B #
    range-tree-child-full-edge-range-list S B b bs

fun range-tree-child-bound-pair-list where
  range-tree-child-bound-pair-list S a [] [] = []
| range-tree-child-bound-pair-list S a [] (b # bs) = []
| range-tree-child-bound-pair-list S a (beta # betas) [] = []
| range-tree-child-bound-pair-list S a (beta # betas) (b # bs) =
    (range-tree S a (Fin b), Fin beta) #
    range-tree-child-bound-pair-list S b betas bs

lemma range-tree-child-direct-edge-range-list-subset-outgoing-sources:
  assumes Y ∈ set (range-tree-child-direct-edge-range-list S B a betas bs)
  shows ∃ X∈set (range-tree-child-list S a bs). Y ⊆ outgoing-edges X
  using assms
proof (induction betas arbitrary: a bs)
  case Nil
  then show ?case
    by (cases bs) simp-all
next
  case (Cons beta betas)
  show ?case
  proof (cases bs)
    case Nil
    then show ?thesis
      using Cons.prems by simp
  next
    case (Cons b bs')
    let ?head = outgoing-edges-range (range-tree S a (Fin b)) beta B
    let ?tail = range-tree-child-direct-edge-range-list S B b betas bs'
    have Y-cases: Y = ?head ∨ Y ∈ set ?tail
      using Cons.prems Cons by simp
    then show ?thesis
    proof
      assume Y-head: Y = ?head
      have ?head ⊆ outgoing-edges (range-tree S a (Fin b))
        by (rule outgoing-edges-range-subset-outgoing-edges)
      then show ?thesis
        unfolding Cons Y-head by auto
    next
```

```
      assume Y-tail: Y ∈ set ?tail
      obtain X where X: X ∈ set (range-tree-child-list S b bs')
        and YX: Y ⊆ outgoing-edges X
        using Cons.IH[OF Y-tail] by blast
      show ?thesis
        using X YX unfolding Cons by auto
    qed
  qed
qed

lemma range-tree-child-direct-edge-range-list-member-subset-E:
  assumes Y ∈ set (range-tree-child-direct-edge-range-list S B a betas bs)
  shows Y ⊆ E
proof
  fix e
  assume eY: e ∈ Y
  obtain X where X ∈ set (range-tree-child-list S a bs)
    and Y-subset: Y ⊆ outgoing-edges X
    using range-tree-child-direct-edge-range-list-subset-outgoing-sources[OF assms]
    by blast
  then have e ∈ outgoing-edges X
    using eY by blast
  then show e ∈ E
    unfolding outgoing-edges-def by blast
qed

lemma pairwise-disjoint-list-range-tree-child-direct-edge-range-list:
  assumes mono: nondecreasing-from a bs
  shows pairwise-disjoint-list
    (range-tree-child-direct-edge-range-list S B a betas bs)
  using mono
proof (induction betas arbitrary: a bs)
  case Nil
  then show ?case
    by (cases bs) simp-all
next
  case (Cons beta betas)
  show ?case
  proof (cases bs)
    case Nil
    then show ?thesis by simp
  next
    case (Cons b bs')
    let ?head = outgoing-edges-range (range-tree S a (Fin b)) beta B
    let ?tail = range-tree-child-direct-edge-range-list S B b betas bs'
    have a-le-b: a ≤ b
      using Cons.prems Cons by simp
    have mono-tail: nondecreasing-from b bs'
      using Cons.prems Cons by simp
```

```
    have head-tail-disj: ∀ Y∈set ?tail. ?head ∩ Y = {}
    proof
      fix Y
      assume Y: Y ∈ set ?tail
      obtain X where X: X ∈ set (range-tree-child-list S b bs')
        and YX: Y ⊆ outgoing-edges X
        using range-tree-child-direct-edge-range-list-subset-outgoing-sources[OF Y]
        by blast
      have X-subset: X ⊆ range-tree-chain S b bs' Infinity
        using X Union-range-tree-child-list-subset-chain[of S b bs' Infinity]
        by blast
      have source-disj: range-tree S a (Fin b) ∩ X = {}
        using range-tree-disjoint-range-tree-chain-tail
          [OF a-le-b mono-tail, of S Infinity] X-subset by blast
      have out-disj:
        outgoing-edges (range-tree S a (Fin b)) ∩ outgoing-edges X = {}
        by (rule outgoing-edges-disjoint-if-sources-disjoint[OF source-disj])
      have head-subset: ?head ⊆ outgoing-edges (range-tree S a (Fin b))
        by (rule outgoing-edges-range-subset-outgoing-edges)
      show ?head ∩ Y = {}
        using head-subset YX out-disj by blast
    qed
    have tail-disj: pairwise-disjoint-list ?tail
      by (rule Cons.IH[OF mono-tail])
    show ?thesis
      unfolding Cons by (simp add: head-tail-disj tail-disj)
  qed
qed

lemma sum-card-range-tree-child-direct-edge-range-list-le-edge-count:
  assumes mono: nondecreasing-from a bs
  shows sum-list
    (map card (range-tree-child-direct-edge-range-list S B a betas bs))
    ≤ edge-count
proof −
  let ?Xs = range-tree-child-direct-edge-range-list S B a betas bs
  have disj: pairwise-disjoint-list ?Xs
    by (rule pairwise-disjoint-list-range-tree-child-direct-edge-range-list
      [OF mono])
  have finite-sets: ⋀X. X ∈ set ?Xs ⟹ finite X
    using range-tree-child-direct-edge-range-list-member-subset-E
    by (meson finite-E finite-subset)
  have card-union:
    card (⋃(set ?Xs)) = sum-list (map card ?Xs)
    by (rule card-Union-pairwise-disjoint-list[OF disj finite-sets])
  have union-subset: ⋃(set ?Xs) ⊆ E
  proof
    fix e
    assume e ∈ ⋃(set ?Xs)
```

```
    then obtain X where X: X ∈ set ?Xs and eX: e ∈ X
      by blast
    show e ∈ E
      using range-tree-child-direct-edge-range-list-member-subset-E[OF X] eX
      by blast
  qed
  have sum-list (map card ?Xs) = card (⋃(set ?Xs))
    using card-union by simp
  also have ... ≤ edge-count
    unfolding edge-count-def by (rule card-mono[OF finite-E union-subset])
  finally show ?thesis .
qed

lemma sum-card-range-tree-child-direct-edge-range-list-le-outgoing-edges-chain:
  assumes mono: nondecreasing-from a bs
  shows sum-list
    (map card (range-tree-child-direct-edge-range-list S B' a betas bs))
    ≤ card (outgoing-edges (range-tree-chain S a bs B))
proof −
  let ?Rs = range-tree-child-direct-edge-range-list S B' a betas bs
  let ?Xs = range-tree-child-list S a bs
  have disj: pairwise-disjoint-list ?Rs
    by (rule pairwise-disjoint-list-range-tree-child-direct-edge-range-list
      [OF mono])
  have finite-sets: ⋀R. R ∈ set ?Rs ⟹ finite R
    using range-tree-child-direct-edge-range-list-member-subset-E
    by (meson finite-E finite-subset)
  have card-union:
    card (⋃(set ?Rs)) = sum-list (map card ?Rs)
    by (rule card-Union-pairwise-disjoint-list[OF disj finite-sets])
  have union-subset-children: ⋃(set ?Rs) ⊆ outgoing-edges (⋃(set ?Xs))
  proof
    fix e
    assume e ∈ ⋃(set ?Rs)
    then obtain R where R: R ∈ set ?Rs and eR: e ∈ R
      by blast
    obtain X where X: X ∈ set ?Xs and R-subset: R ⊆ outgoing-edges X
      using range-tree-child-direct-edge-range-list-subset-outgoing-sources[OF R]
      by blast
    show e ∈ outgoing-edges (⋃(set ?Xs))
      using eR R-subset X unfolding outgoing-edges-def by blast
  qed
  have children-subset: ⋃(set ?Xs) ⊆ range-tree-chain S a bs B
    by (rule Union-range-tree-child-list-subset-chain)
  have union-subset:
    ⋃(set ?Rs) ⊆ outgoing-edges (range-tree-chain S a bs B)
    using union-subset-children outgoing-edges-mono[OF children-subset] by blast
  have sum-list (map card ?Rs) = card (⋃(set ?Rs))
    using card-union by simp
```

```
  also have ... ≤ card (outgoing-edges (range-tree-chain S a bs B))
    by (rule card-mono) (simp-all add: union-subset)
  finally show ?thesis .
qed

lemma range-tree-nonneg-edge-value:
  assumes edge: (u, v) ∈ E
    and u-range: u ∈ range-tree S a B
  shows 0 ≤ dist s u + w u v
proof −
  have reach-u: reachable s u
    using u-range unfolding range-tree-def tree-set-def tree-path-def by blast
  have 0 ≤ dist s u
    by (rule dist-nonneg[OF reach-u])
  moreover have 0 ≤ w u v
    by (rule nonneg-weight[OF edge])
  ultimately show ?thesis
    by linarith
qed

lemma sum-card-range-tree-child-zero-direct-edge-ranges-le-full:
  assumes bounds: bounds-le B betas
  shows
    sum-list (map card (range-tree-child-zero-edge-range-list S a betas bs)) +
     sum-list
       (map card (range-tree-child-direct-edge-range-list S B a betas bs))
     ≤ sum-list (map card (range-tree-child-full-edge-range-list S B a bs))
  using bounds
proof (induction betas arbitrary: a bs)
  case Nil
  then show ?case
    by (cases bs) simp-all
next
  case (Cons beta betas)
  show ?case
  proof (cases bs)
    case Nil
    then show ?thesis
      by simp
  next
    case (Cons b bs')
    have beta-le-B: bound-le (Fin beta) B
      using Cons.prems by simp
    have bounds-tail: bounds-le B betas
      using Cons.prems by simp
    have head:
      card (outgoing-edges-range (range-tree S a (Fin b)) 0 (Fin beta)) +
       card (outgoing-edges-range (range-tree S a (Fin b)) beta B)
       ≤ card (outgoing-edges-range (range-tree S a (Fin b)) 0 B)
```

```
    by (rule card-outgoing-edges-range-zero-lower-direct-split-le
      [OF beta-le-B range-tree-nonneg-edge-value])
  have tail:
    sum-list (map card (range-tree-child-zero-edge-range-list S b betas bs')) +
     sum-list
      (map card (range-tree-child-direct-edge-range-list S B b betas bs'))
    ≤ sum-list (map card
      (range-tree-child-full-edge-range-list S B b bs'))
    by (rule Cons.IH[OF bounds-tail])
  show ?thesis
    unfolding Cons by simp (use head tail in linarith)
 qed
qed

lemma range-tree-child-full-edge-range-list-eq-map:
 range-tree-child-full-edge-range-list S B a bs =
  map (λU. outgoing-edges-range U 0 B) (range-tree-child-list S a bs)
 by (induction bs arbitrary: a) simp-all

lemma sum-card-range-tree-child-full-edge-range-list-le-outgoing-edges-range-chain:
 assumes mono: nondecreasing-from a bs
 shows sum-list (map card (range-tree-child-full-edge-range-list S B a bs))
  ≤ card (outgoing-edges-range (range-tree-chain S a bs C) 0 B)
proof −
 let ?Xs = range-tree-child-list S a bs
 let ?Rs = map (λU. outgoing-edges-range U 0 B) ?Xs
 have disj-children: pairwise-disjoint-list ?Xs
  using mono
 proof (induction bs arbitrary: a)
  case Nil
  then show ?case by simp
 next
  case (Cons b bs)
  have a-le-b: a ≤ b
   using Cons.prems by simp
  have mono-tail: nondecreasing-from b bs
   using Cons.prems by simp
  have head-tail-disj:
   ⋀Y. Y ∈ set (range-tree-child-list S b bs) ⟹
    range-tree S a (Fin b) ∩ Y = {}
  proof −
   fix Y
   assume Y: Y ∈ set (range-tree-child-list S b bs)
   have Y ⊆ range-tree-chain S b bs Infinity
    using Y Union-range-tree-child-list-subset-chain[of S b bs Infinity]
    by blast
   moreover have
    range-tree S a (Fin b) ∩ range-tree-chain S b bs Infinity = {}
    using range-tree-disjoint-range-tree-chain-tail[OF a-le-b mono-tail] .
```

```
    ultimately show range-tree S a (Fin b) ∩ Y = {}
      by blast
  qed
  show ?case
    using head-tail-disj Cons.IH[OF mono-tail] by simp
qed
have disj: pairwise-disjoint-list ?Rs
  by (rule pairwise-disjoint-list-outgoing-edges-range[OF disj-children])
have finite-sets: ⋀R. R ∈ set ?Rs ⟹ finite R
  by auto
have card-union:
  card (⋃(set ?Rs)) = sum-list (map card ?Rs)
  by (rule card-Union-pairwise-disjoint-list[OF disj finite-sets])
have children-subset: ⋃(set ?Xs) ⊆ range-tree-chain S a bs C
  by (rule Union-range-tree-child-list-subset-chain)
have union-subset:
  ⋃(set ?Rs) ⊆ outgoing-edges-range (range-tree-chain S a bs C) 0 B
proof
  fix e
  assume e ∈ ⋃(set ?Rs)
  then obtain X where X: X ∈ set ?Xs
    and eX: e ∈ outgoing-edges-range X 0 B
    by auto
  have X-subset: X ⊆ range-tree-chain S a bs C
    using X children-subset by blast
  show e ∈ outgoing-edges-range (range-tree-chain S a bs C) 0 B
    using outgoing-edges-range-mono-sources[OF X-subset] eX by blast
qed
have sum-list (map card (range-tree-child-full-edge-range-list S B a bs)) =
    sum-list (map card ?Rs)
  by (simp add: range-tree-child-full-edge-range-list-eq-map)
also have ... = card (⋃(set ?Rs))
  using card-union by simp
also have ... ≤
    card (outgoing-edges-range (range-tree-chain S a bs C) 0 B)
  by (rule card-mono[OF finite-outgoing-edges-range union-subset])
finally show ?thesis .
qed

lemma sum-card-range-tree-child-zero-direct-edge-ranges-le-outgoing-edges-range-
chain:
  assumes mono: nondecreasing-from a bs
    and bounds: bounds-le B betas
  shows
    sum-list (map card (range-tree-child-zero-edge-range-list S a betas bs)) +
     sum-list
       (map card (range-tree-child-direct-edge-range-list S B a betas bs))
     ≤ card (outgoing-edges-range (range-tree-chain S a bs C) 0 B)
proof −
```

```
  have split:
    sum-list (map card (range-tree-child-zero-edge-range-list S a betas bs)) +
     sum-list
       (map card (range-tree-child-direct-edge-range-list S B a betas bs))
     ≤ sum-list (map card (range-tree-child-full-edge-range-list S B a bs))
    by (rule sum-card-range-tree-child-zero-direct-edge-ranges-le-full
      [OF bounds])
  have full:
    sum-list (map card (range-tree-child-full-edge-range-list S B a bs))
      ≤ card (outgoing-edges-range (range-tree-chain S a bs C) 0 B)
    by (rule sum-card-range-tree-child-full-edge-range-list-le-outgoing-edges-range-
chain
      [OF mono])
  show ?thesis
    using split full by linarith
qed

lemma weighted-sum-child-lower-direct-edge-ranges-le:
  assumes mono: nondecreasing-from a bs
  shows
    h * sum-list (map card (range-tree-child-edge-range-list S a betas bs)) +
     t * sum-list
       (map card (range-tree-child-direct-edge-range-list S B a betas bs))
     ≤ (h + t) * edge-count
proof −
  have lower:
    sum-list (map card (range-tree-child-edge-range-list S a betas bs))
      ≤ edge-count
    by (rule sum-card-range-tree-child-edge-range-list-le-edge-count[OF mono])
  have direct:
    sum-list
      (map card (range-tree-child-direct-edge-range-list S B a betas bs))
      ≤ edge-count
    by (rule sum-card-range-tree-child-direct-edge-range-list-le-edge-count
      [OF mono])
  have lower-w:
    h * sum-list (map card (range-tree-child-edge-range-list S a betas bs))
      ≤ h * edge-count
    using lower by simp
  have direct-w:
    t * sum-list
       (map card (range-tree-child-direct-edge-range-list S B a betas bs))
      ≤ t * edge-count
    using direct by simp
  have h * sum-list (map card (range-tree-child-edge-range-list S a betas bs)) +
     t * sum-list
       (map card (range-tree-child-direct-edge-range-list S B a betas bs))
      ≤ h * edge-count + t * edge-count
    using lower-w direct-w by linarith
```

also have ... = $(h + t) * \mathit{edge\text{-}count}$
by (*simp add*: *algebra-simps*)
finally show *?thesis* **.**
qed

lemma *weighted-sum-child-lower-direct-edge-ranges-le-outgoing-edges-chain*:
assumes *mono*: *nondecreasing-from a bs*
shows
$h * \mathit{sum\text{-}list}\ (\mathit{map}\ \mathit{card}\ (\mathit{range\text{-}tree\text{-}child\text{-}edge\text{-}range\text{-}list}\ S\ a\ \mathit{betas}\ \mathit{bs})) +$
$t * \mathit{sum\text{-}list}$
$(\mathit{map}\ \mathit{card}\ (\mathit{range\text{-}tree\text{-}child\text{-}direct\text{-}edge\text{-}range\text{-}list}\ S\ B'\ a\ \mathit{betas}\ \mathit{bs}))$
$\leq (h + t) * \mathit{card}\ (\mathit{outgoing\text{-}edges}\ (\mathit{range\text{-}tree\text{-}chain}\ S\ a\ \mathit{bs}\ B))$
proof −
have *lower*:
$\mathit{sum\text{-}list}\ (\mathit{map}\ \mathit{card}\ (\mathit{range\text{-}tree\text{-}child\text{-}edge\text{-}range\text{-}list}\ S\ a\ \mathit{betas}\ \mathit{bs}))$
$\leq \mathit{card}\ (\mathit{outgoing\text{-}edges}\ (\mathit{range\text{-}tree\text{-}chain}\ S\ a\ \mathit{bs}\ B))$
by (*rule sum-card-range-tree-child-edge-range-list-le-outgoing-edges-chain*
[*OF mono*])
have *direct*:
$\mathit{sum\text{-}list}$
$(\mathit{map}\ \mathit{card}\ (\mathit{range\text{-}tree\text{-}child\text{-}direct\text{-}edge\text{-}range\text{-}list}\ S\ B'\ a\ \mathit{betas}\ \mathit{bs}))$
$\leq \mathit{card}\ (\mathit{outgoing\text{-}edges}\ (\mathit{range\text{-}tree\text{-}chain}\ S\ a\ \mathit{bs}\ B))$
by (*rule sum-card-range-tree-child-direct-edge-range-list-le-outgoing-edges-chain*
[*OF mono*])
have *lower-w*:
$h * \mathit{sum\text{-}list}\ (\mathit{map}\ \mathit{card}\ (\mathit{range\text{-}tree\text{-}child\text{-}edge\text{-}range\text{-}list}\ S\ a\ \mathit{betas}\ \mathit{bs}))$
$\leq h * \mathit{card}\ (\mathit{outgoing\text{-}edges}\ (\mathit{range\text{-}tree\text{-}chain}\ S\ a\ \mathit{bs}\ B))$
using *lower* **by** *simp*
have *direct-w*:
$t * \mathit{sum\text{-}list}$
$(\mathit{map}\ \mathit{card}\ (\mathit{range\text{-}tree\text{-}child\text{-}direct\text{-}edge\text{-}range\text{-}list}\ S\ B'\ a\ \mathit{betas}\ \mathit{bs}))$
$\leq t * \mathit{card}\ (\mathit{outgoing\text{-}edges}\ (\mathit{range\text{-}tree\text{-}chain}\ S\ a\ \mathit{bs}\ B))$
using *direct* **by** *simp*
have $h * \mathit{sum\text{-}list}\ (\mathit{map}\ \mathit{card}\ (\mathit{range\text{-}tree\text{-}child\text{-}edge\text{-}range\text{-}list}\ S\ a\ \mathit{betas}\ \mathit{bs})) +$
$t * \mathit{sum\text{-}list}$
$(\mathit{map}\ \mathit{card}\ (\mathit{range\text{-}tree\text{-}child\text{-}direct\text{-}edge\text{-}range\text{-}list}\ S\ B'\ a\ \mathit{betas}\ \mathit{bs}))$
$\leq h * \mathit{card}\ (\mathit{outgoing\text{-}edges}\ (\mathit{range\text{-}tree\text{-}chain}\ S\ a\ \mathit{bs}\ B)) +$
$t * \mathit{card}\ (\mathit{outgoing\text{-}edges}\ (\mathit{range\text{-}tree\text{-}chain}\ S\ a\ \mathit{bs}\ B))$
using *lower-w direct-w* **by** *linarith*
also have ... = $(h + t) * \mathit{card}\ (\mathit{outgoing\text{-}edges}\ (\mathit{range\text{-}tree\text{-}chain}\ S\ a\ \mathit{bs}\ B))$
by (*simp add*: *algebra-simps*)
finally show *?thesis* **.**
qed

lemma *sum-card-outgoing-edges-le-degree*:
assumes *degree*: *edge-outdegree-le* Δ
and *sets*: $\bigwedge X.\ X \in \mathit{set}\ \mathit{Xs} \Longrightarrow X \subseteq V$
shows $\mathit{sum\text{-}list}\ (\mathit{map}\ (\lambda X.\ \mathit{card}\ (\mathit{outgoing\text{-}edges}\ X))\ \mathit{Xs}) \leq$
$\Delta * \mathit{sum\text{-}list}\ (\mathit{map}\ \mathit{card}\ \mathit{Xs})$

```
  using sets
proof (induction Xs)
  case Nil
  then show ?case by simp
next
  case (Cons X Xs)
  have X-subset: X ⊆ V
    using Cons.prems by simp
  have X-bound: card (outgoing-edges X) ≤ Δ * card X
    by (rule card-outgoing-edges-le[OF X-subset degree])
  have tail-bound:
    sum-list (map (λX. card (outgoing-edges X)) Xs) ≤
      Δ * sum-list (map card Xs)
    using Cons.IH Cons.prems by simp
  have card (outgoing-edges X) +
      sum-list (map (λX. card (outgoing-edges X)) Xs) ≤
      Δ * card X + Δ * sum-list (map card Xs)
    using X-bound tail-bound by simp
  also have ... = Δ * sum-list (map card (X # Xs))
    by (simp add: algebra-simps)
  finally show ?case
    by simp
qed

lemma card-outgoing-edges-Union-list-eq-sum:
  assumes disj: pairwise-disjoint-list Xs
  shows card (outgoing-edges (⋃(set Xs))) =
    sum-list (map (λX. card (outgoing-edges X)) Xs)
proof −
  have disj-edges: pairwise-disjoint-list (map outgoing-edges Xs)
    using pairwise-disjoint-list-outgoing-edges[OF disj] .
  have finite-edges: ⋀X. X ∈ set (map outgoing-edges Xs) ⟹ finite X
    by auto
  have card (outgoing-edges (⋃(set Xs))) =
      card (⋃(set (map outgoing-edges Xs)))
    using outgoing-edges-Union-list[of Xs] by simp
  also have ... = sum-list (map (λY. card Y) (map outgoing-edges Xs))
    by (rule card-Union-pairwise-disjoint-list[OF disj-edges finite-edges])
  also have ... = sum-list (map (λX. card (outgoing-edges X)) Xs)
    by (induction Xs) simp-all
  finally show ?thesis .
qed

lemma range-tree-chain-list-pairwise-disjoint:
  assumes mono: nondecreasing-from a bs
  shows pairwise-disjoint-list (range-tree-chain-list S a bs B)
  using assms
proof (induction bs arbitrary: a)
  case Nil
```

```
  then show ?case by simp
next
  case (Cons b bs)
  have a-le-b: a ≤ b
    using Cons.prems by simp
  have mono-tail: nondecreasing-from b bs
    using Cons.prems by simp
  have head-tail-disj:
    ⋀Y. Y ∈ set (range-tree-chain-list S b bs B) ⟹
      range-tree S a (Fin b) ∩ Y = {}
  proof −
    fix Y
    assume Y: Y ∈ set (range-tree-chain-list S b bs B)
    have Y ⊆ range-tree-chain S b bs B
      using Y Union-range-tree-chain-list[of S b bs B] by blast
    moreover have
      range-tree S a (Fin b) ∩ range-tree-chain S b bs B = {}
      using range-tree-disjoint-range-tree-chain-tail[OF a-le-b mono-tail] .
    ultimately show range-tree S a (Fin b) ∩ Y = {}
      by blast
  qed
  show ?case
    using head-tail-disj Cons.IH[OF mono-tail] by simp
qed

lemma range-tree-child-list-pairwise-disjoint:
  assumes mono: nondecreasing-from a bs
  shows pairwise-disjoint-list (range-tree-child-list S a bs)
  using assms
proof (induction bs arbitrary: a)
  case Nil
  then show ?case by simp
next
  case (Cons b bs)
  have a-le-b: a ≤ b
    using Cons.prems by simp
  have mono-tail: nondecreasing-from b bs
    using Cons.prems by simp
  have head-tail-disj:
    ⋀Y. Y ∈ set (range-tree-child-list S b bs) ⟹
      range-tree S a (Fin b) ∩ Y = {}
  proof −
    fix Y
    assume Y: Y ∈ set (range-tree-child-list S b bs)
    have Y ⊆ range-tree-chain S b bs Infinity
      using Y Union-range-tree-child-list-subset-chain[of S b bs Infinity] by blast
    moreover have
      range-tree S a (Fin b) ∩ range-tree-chain S b bs Infinity = {}
      using range-tree-disjoint-range-tree-chain-tail[OF a-le-b mono-tail] .
```

```
    ultimately show range-tree S a (Fin b) ∩ Y = {}
      by blast
  qed
  show ?case
    using head-tail-disj Cons.IH[OF mono-tail] by simp
qed

theorem card-range-tree-child-list-le-chain:
  assumes mono: nondecreasing-from a bs
  shows sum-list (map card (range-tree-child-list S a bs)) ≤
    card (range-tree-chain S a bs B)
proof −
  let ?Xs = range-tree-child-list S a bs
  have disj: pairwise-disjoint-list ?Xs
    by (rule range-tree-child-list-pairwise-disjoint[OF mono])
  have finite-sets: ⋀X. X ∈ set ?Xs ⟹ finite X
    using range-tree-child-list-set-subset-chain-list
      finite-range-tree-chain-list-sets by blast
  have card-union:
    card (⋃(set ?Xs)) = sum-list (map card ?Xs)
    by (rule card-Union-pairwise-disjoint-list[OF disj finite-sets])
  have subset: ⋃(set ?Xs) ⊆ range-tree-chain S a bs B
    by (rule Union-range-tree-child-list-subset-chain)
  have sum-list (map card ?Xs) = card (⋃(set ?Xs))
    using card-union by simp
  also have ... ≤ card (range-tree-chain S a bs B)
    by (rule card-mono) (simp-all add: subset)
  finally show ?thesis .
qed

lemma sum-card-list-all2-le:
  assumes list-all2 (λX Y. card X ≤ card Y) Xs Ys
  shows sum-list (map card Xs) ≤ sum-list (map card Ys)
  using assms
proof (induction rule: list-all2-induct)
  case Nil
  then show ?case by simp
next
  case (Cons X Y Xs Ys)
  then show ?case by simp
qed

lemma sum-card-dominated-by-disjoint-union:
  assumes disj: pairwise-disjoint-list Ys
    and finite-sets: ⋀Y. Y ∈ set Ys ⟹ finite Y
    and dominated: list-all2 (λX Y. card X ≤ card Y) Xs Ys
  shows sum-list (map card Xs) ≤ card (⋃(set Ys))
proof −
  have sum-list (map card Xs) ≤ sum-list (map card Ys)
```

```
    by (rule sum-card-list-all2-le[OF dominated])
  also have ... = card (⋃(set Ys))
    using card-Union-pairwise-disjoint-list[OF disj finite-sets] by simp
  finally show ?thesis .
qed

theorem sum-card-dominated-by-range-tree-child-list-le-chain:
  assumes mono: nondecreasing-from a bs
    and dominated:
      list-all2 (λX R. card X ≤ card R) Xs
        (range-tree-child-list S a bs)
  shows sum-list (map card Xs) ≤ card (range-tree-chain S a bs B)
proof −
  have sum-list (map card Xs) ≤
      sum-list (map card (range-tree-child-list S a bs))
    by (rule sum-card-list-all2-le[OF dominated])
  also have ... ≤ card (range-tree-chain S a bs B)
    by (rule card-range-tree-child-list-le-chain[OF mono])
  finally show ?thesis .
qed

theorem card-outgoing-edges-range-tree-chain-eq-sum-list:
  assumes mono: nondecreasing-from a bs
  shows card (outgoing-edges (range-tree-chain S a bs B)) =
    sum-list
      (map (λX. card (outgoing-edges X)) (range-tree-chain-list S a bs B))
proof −
  have disj: pairwise-disjoint-list (range-tree-chain-list S a bs B)
    using range-tree-chain-list-pairwise-disjoint[OF mono] .
  have card (outgoing-edges (range-tree-chain S a bs B)) =
      card (outgoing-edges (⋃(set (range-tree-chain-list S a bs B))))
    using Union-range-tree-chain-list[of S a bs B] by simp
  also have ... =
      sum-list
        (map (λX. card (outgoing-edges X)) (range-tree-chain-list S a bs B))
    by (rule card-outgoing-edges-Union-list-eq-sum[OF disj])
  finally show ?thesis .
qed

theorem card-outgoing-edges-range-tree-child-list-le-chain:
  assumes mono: nondecreasing-from a bs
  shows sum-list
      (map (λX. card (outgoing-edges X)) (range-tree-child-list S a bs)) ≤
    card (outgoing-edges (range-tree-chain S a bs B))
proof −
  let ?Xs = range-tree-child-list S a bs
  have disj: pairwise-disjoint-list ?Xs
    by (rule range-tree-child-list-pairwise-disjoint[OF mono])
  have card-union:
```

```
    card (outgoing-edges (⋃(set ?Xs))) =
      sum-list (map (λX. card (outgoing-edges X)) ?Xs)
    by (rule card-outgoing-edges-Union-list-eq-sum[OF disj])
  have subset: ⋃(set ?Xs) ⊆ range-tree-chain S a bs B
    by (rule Union-range-tree-child-list-subset-chain)
  have outgoing-subset:
    outgoing-edges (⋃(set ?Xs)) ⊆
      outgoing-edges (range-tree-chain S a bs B)
    by (rule outgoing-edges-mono[OF subset])
  have sum-list (map (λX. card (outgoing-edges X)) ?Xs) =
      card (outgoing-edges (⋃(set ?Xs)))
    using card-union by simp
  also have ... ≤ card (outgoing-edges (range-tree-chain S a bs B))
    by (rule card-mono) (simp-all add: outgoing-subset)
  finally show ?thesis .
qed

lemma sum-outgoing-card-list-all2-le:
  assumes list-all2
    (λX Y. card (outgoing-edges X) ≤ card (outgoing-edges Y)) Xs Ys
  shows sum-list (map (λX. card (outgoing-edges X)) Xs) ≤
    sum-list (map (λY. card (outgoing-edges Y)) Ys)
  using assms
proof (induction rule: list-all2-induct)
  case Nil
  then show ?case by simp
next
  case (Cons X Y Xs Ys)
  then show ?case by simp
qed

theorem sum-outgoing-dominated-by-range-tree-child-list-le-chain:
  assumes mono: nondecreasing-from a bs
    and dominated:
      list-all2
        (λX R. card (outgoing-edges X) ≤ card (outgoing-edges R)) Xs
        (range-tree-child-list S a bs)
  shows sum-list (map (λX. card (outgoing-edges X)) Xs) ≤
    card (outgoing-edges (range-tree-chain S a bs B))
proof −
  have sum-list (map (λX. card (outgoing-edges X)) Xs) ≤
      sum-list
        (map (λR. card (outgoing-edges R)) (range-tree-child-list S a bs))
    by (rule sum-outgoing-card-list-all2-le[OF dominated])
  also have ... ≤ card (outgoing-edges (range-tree-chain S a bs B))
    by (rule card-outgoing-edges-range-tree-child-list-le-chain[OF mono])
  finally show ?thesis .
qed
```

```
lemma edge-costs-le-range-tree-chain-outgoing:
  assumes mono: nondecreasing-from a bs
    and costs: list-all2 (λc X. c ≤ C * card (outgoing-edges X)) costs
      (range-tree-chain-list S a bs B)
  shows sum-list costs ≤ C * card (outgoing-edges (range-tree-chain S a bs B))
proof −
  have sum-list costs ≤
      sum-list
        (map (λX. C * card (outgoing-edges X))
          (range-tree-chain-list S a bs B))
    using costs
  proof (induction rule: list-all2-induct)
    case Nil
    then show ?case by simp
  next
    case (Cons c X costs Xs)
    then show ?case by simp
  qed
  also have ... =
      C * sum-list
        (map (λX. card (outgoing-edges X)) (range-tree-chain-list S a bs B))
  proof −
    define Xs where Xs = range-tree-chain-list S a bs B
    have sum-list (map (λX. C * card (outgoing-edges X)) Xs) =
        C * sum-list (map (λX. card (outgoing-edges X)) Xs)
      by (induction Xs) (simp-all add: algebra-simps)
    then show ?thesis
      unfolding Xs-def .
  qed
  also have ... = C * card (outgoing-edges (range-tree-chain S a bs B))
    using card-outgoing-edges-range-tree-chain-eq-sum-list[OF mono, of S B] by
simp
  finally show ?thesis .
qed
```

These two cost shapes are the local language of the runtime proof. The level bound *level-range-cost-bound* is the simple recurrence form: a vertex term scaled by the recursion level and an outgoing-edge term scaled by the range coefficient. The amortised bound *bmssp-amortized-cost-bound* adds the extra direct-edge range term used by the bucketed partition, where direct inserts have their own cost story.

The summation lemmas below say that if every recursive child call satisfies one of these bounds on its own range slice, then the sum of child costs can be rewritten as sums over the corresponding child range lists. Later theories collapse those sums using the disjointness lemmas above.

```
definition level-range-cost-bound where
  level-range-cost-bound A R l X =
    A * l * card X + R * card (outgoing-edges X)
```

```
definition bmssp-amortized-cost-bound where
  bmssp-amortized-cost-bound A R h t q L B X =
    A * L * card X + (R + q * h) * card (outgoing-edges X) +
    t * card (outgoing-edges-range X 0 B)

lemma sum-bmssp-amortized-child-bounds-le:
  assumes bounds:
    list-all2 (λc-child UB. case UB of (U-child, B-child) ⇒
      c-child ≤
        bmssp-amortized-cost-bound A R h t q L B-child U-child)
      child-costs (range-tree-child-bound-pair-list S a betas bs)
  shows sum-list child-costs ≤
    A * L * sum-list (map card (range-tree-child-list S a bs)) +
    (R + q * h) *
      sum-list (map (λU. card (outgoing-edges U))
        (range-tree-child-list S a bs)) +
    t * sum-list
      (map card (range-tree-child-zero-edge-range-list S a betas bs))
  using bounds
proof (induction betas arbitrary: a bs child-costs)
  case Nil
  then have costs-nil: child-costs = []
    by (cases bs; cases child-costs) auto
  show ?case
    using costs-nil by simp
next
  case (Cons beta betas)
  note outer-prems = Cons.prems
  note outer-IH = Cons.IH
  show ?case
  proof (cases bs)
    case Nil
    then have costs-nil: child-costs = []
      using outer-prems by (cases child-costs) auto
    show ?thesis
      using Nil costs-nil by simp
  next
    case bs-Cons: (Cons b bs')
    show ?thesis
    proof (cases child-costs)
      case Nil
      then show ?thesis
        by simp
    next
      case costs-Cons: (Cons c-child child-costs-tail)
      have head:
        c-child ≤
          bmssp-amortized-cost-bound A R h t q L (Fin beta)
```

```
      (range-tree S a (Fin b))
    using outer-prems unfolding bs-Cons costs-Cons by simp
  have tail-bounds:
    list-all2 (λc-child UB. case UB of (U-child, B-child) ⇒
      c-child ≤
        bmssp-amortized-cost-bound A R h t q L B-child U-child)
      child-costs-tail (range-tree-child-bound-pair-list S b betas bs')
    using outer-prems unfolding bs-Cons costs-Cons by simp
  have tail:
    sum-list child-costs-tail ≤
      A * L * sum-list (map card (range-tree-child-list S b bs')) +
      (R + q * h) *
        sum-list (map (λU. card (outgoing-edges U))
          (range-tree-child-list S b bs')) +
      t * sum-list
        (map card (range-tree-child-zero-edge-range-list S b betas bs'))
    by (rule outer-IH[OF tail-bounds])
  have head-unfolded:
    c-child ≤
      A * L * card (range-tree S a (Fin b)) +
      (R + q * h) * card (outgoing-edges (range-tree S a (Fin b))) +
      t * card (outgoing-edges-range (range-tree S a (Fin b)) 0
        (Fin beta))
    using head unfolding bmssp-amortized-cost-bound-def .
  show ?thesis
    unfolding bs-Cons costs-Cons
    using head-unfolded tail by (simp add: algebra-simps)
 qed
qed

lemma child-costs-le-level-range-chain-bound:
 assumes mono: nondecreasing-from a bs
   and costs: list-all2 (λc X. c ≤ level-range-cost-bound A R l X) costs
     (range-tree-chain-list S a bs B)
 shows sum-list costs ≤
   A * l * card (range-tree-chain S a bs B) +
   R * card (outgoing-edges (range-tree-chain S a bs B))
proof −
 define Xs where Xs = range-tree-chain-list S a bs B
 have cost-sum:
   sum-list costs ≤ sum-list (map (level-range-cost-bound A R l) Xs)
   using costs unfolding Xs-def
 proof (induction rule: list-all2-induct)
   case Nil
   then show ?case by simp
 next
   case (Cons c X costs Xs)
   then show ?case
```

```
    unfolding level-range-cost-bound-def by simp
  qed
  have split-sum:
    sum-list (map (level-range-cost-bound A R l) Xs) =
      A * l * sum-list (map (λX. card X) Xs) +
      R * sum-list (map (λX. card (outgoing-edges X)) Xs)
  proof (induction Xs)
    case Nil
    then show ?case
      unfolding level-range-cost-bound-def by simp
  next
    case (Cons X Xs)
    then show ?case
      unfolding level-range-cost-bound-def by (simp add: algebra-simps)
  qed
  have card-sum:
    sum-list (map (λX. card X) Xs) = card (range-tree-chain S a bs B)
    using card-range-tree-chain-eq-sum-list[OF mono, of S B]
    unfolding Xs-def by simp
  have edge-sum:
    sum-list (map (λX. card (outgoing-edges X)) Xs) =
      card (outgoing-edges (range-tree-chain S a bs B))
    using card-outgoing-edges-range-tree-chain-eq-sum-list[OF mono, of S B]
    unfolding Xs-def by simp
  show ?thesis
    using cost-sum split-sum card-sum edge-sum by simp
qed

theorem concrete-partition-loop-trace-card-decomposition:
  assumes trace: concrete-partition-loop-trace P B a bs d' B' Us U
  shows card U =
    card (bound-tree P (Fin a)) + card (range-tree-chain P a bs B')
proof −
  have mono: nondecreasing-from a bs
    and bounds: bounds-le B' (a # bs)
    using trace unfolding concrete-partition-loop-trace-def by blast+
  have post: bmssp-post-full d P B d' B' U
    using concrete-partition-loop-trace-post[OF trace] .
  then have U-eq: U = bound-tree P B'
    unfolding bmssp-post-full-def by blast
  show ?thesis
    using card-bound-tree-range-chain-union[OF mono bounds, of P] U-eq by simp
qed

theorem concrete-partition-loop-trace-range-card-le:
  assumes trace: concrete-partition-loop-trace P B a bs d' B' Us U
  shows card (range-tree-chain P a bs B') ≤ card U
  using concrete-partition-loop-trace-card-decomposition[OF trace]
  by simp
```

```
definition concrete-capped-step-core-cost where
  concrete-capped-step-core-cost k cap d S B child-costs =
    fp-iter-capped-scan-cost k cap d S S B + sum-list child-costs

definition base-case-scan-cost where
  base-case-scan-cost Δ k x B = card (outgoing-edges (base-case-vertices k x B))

theorem base-case-scan-cost-le:
  assumes degree: edge-outdegree-le Δ
  shows base-case-scan-cost Δ k x B ≤ Δ * k
proof −
  have order-subset: set (base-case-order x B) ⊆ V
    using base-case-order-set[of x B]
    unfolding bound-tree-def by auto
  have vertices-subset: base-case-vertices k x B ⊆ V
  proof (cases length (base-case-order x B) ≤ k)
    case True
    then show ?thesis
      using order-subset unfolding base-case-vertices-def by (simp add: Let-def)
  next
    case False
    then show ?thesis
      using order-subset unfolding base-case-vertices-def
      by (auto simp: Let-def dest: in-set-takeD)
  qed
  have base-case-scan-cost Δ k x B ≤
      Δ * card (base-case-vertices k x B)
    unfolding base-case-scan-cost-def
    by (rule card-outgoing-edges-le[OF vertices-subset degree])
  also have ... ≤ Δ * k
    using card-base-case-vertices-le[of k x B] by simp
  finally show ?thesis .
qed

theorem base-case-scan-cost-le-outgoing-edges:
  base-case-scan-cost Δ k x B ≤
    card (outgoing-edges (base-case-vertices k x B))
  unfolding base-case-scan-cost-def
  by simp

theorem base-case-scan-cost-le-edge-count:
  base-case-scan-cost edge-count k x B ≤ edge-count * k
  by (rule base-case-scan-cost-le[OF edge-outdegree-le-edge-count])

definition concrete-capped-step-accounted-cost where
  concrete-capped-step-accounted-cost k cap d S B child-costs edge-costs =
    fp-iter-capped-scan-cost k cap d S S B +
    sum-list child-costs + sum-list edge-costs
```

theorem *concrete-capped-step-core-cost-le-range*:
assumes *S-subset*: $S \subseteq V$
and *degree*: *edge-outdegree-le* Δ
and *S-cap*: *card* $S \leq$ *cap*
and *trace*: *concrete-partition-loop-trace*
(*find-pivots-pivots-capped k cap d S B*) *B a bs d′ B′ Us U-loop*
and *child-costs*: *list-all2* ($\lambda c\ X.\ c \leq C * card\ X$) *costs*
(*range-tree-chain-list* (*find-pivots-pivots-capped k cap d S B*) *a bs B′*)
shows *concrete-capped-step-core-cost k cap d S B costs* $\leq$
$k * \Delta * cap +$
$C * card$ (*range-tree-chain*
(*find-pivots-pivots-capped k cap d S B*) *a bs B′*)
proof −
let *?P* = *find-pivots-pivots-capped k cap d S B*
have *mono*: *nondecreasing-from a bs*
using *trace* **unfolding** *concrete-partition-loop-trace-def* **by** *blast*
have *scan*:
fp-iter-capped-scan-cost k cap d S S B $\leq k * \Delta * cap$
using *find-pivots-capped-scan-cost-le*[*OF S-subset degree S-cap*] **.**
have *child*:
sum-list costs $\leq C *$ *card* (*range-tree-chain ?P a bs B′*)
using *child-costs-le-range-tree-chain-card*[*OF mono child-costs*] **.**
show *?thesis*
unfolding *concrete-capped-step-core-cost-def* **using** *scan child* **by** *linarith*
qed

theorem *concrete-capped-step-core-cost-le-level-chain*:
assumes *S-subset*: $S \subseteq V$
and *degree*: *edge-outdegree-le* Δ
and *S-cap*: *card* $S \leq$ *cap*
and *trace*: *concrete-partition-loop-trace*
(*find-pivots-pivots-capped k cap d S B*) *B a bs d′ B′ Us U-loop*
and *child-costs*: *list-all2* ($\lambda c\ X.\ c \leq$ *level-range-cost-bound A R l X*) *costs*
(*range-tree-chain-list* (*find-pivots-pivots-capped k cap d S B*) *a bs B′*)
shows *concrete-capped-step-core-cost k cap d S B costs* $\leq$
$k * \Delta * cap +$
$A * l *$
card (*range-tree-chain* (*find-pivots-pivots-capped k cap d S B*) *a bs B′*) +
$R *$ *card* (*outgoing-edges*
(*range-tree-chain* (*find-pivots-pivots-capped k cap d S B*) *a bs B′*))
proof −
let *?P* = *find-pivots-pivots-capped k cap d S B*
have *mono*: *nondecreasing-from a bs*
using *trace* **unfolding** *concrete-partition-loop-trace-def* **by** *blast*
have *scan*:
fp-iter-capped-scan-cost k cap d S S B $\leq k * \Delta * cap$
using *find-pivots-capped-scan-cost-le*[*OF S-subset degree S-cap*] **.**
have *child*:

```
    sum-list costs ≤
      A * l * card (range-tree-chain ?P a bs B') +
      R * card (outgoing-edges (range-tree-chain ?P a bs B'))
    using child-costs-le-level-range-chain-bound[OF mono child-costs] .
  show ?thesis
    unfolding concrete-capped-step-core-cost-def using scan child by linarith
qed

theorem concrete-capped-step-core-cost-closes-level-bound:
  assumes S-subset: S ⊆ V
    and degree: edge-outdegree-le Δ
    and S-cap: card S ≤ cap
    and trace: concrete-partition-loop-trace
      (find-pivots-pivots-capped k cap d S B) B a bs d' B' Us U-loop
    and child-costs: list-all2 (λc X. c ≤ level-range-cost-bound A R l X) costs
      (range-tree-chain-list (find-pivots-pivots-capped k cap d S B) a bs B')
    and scan-budget: k * Δ * cap ≤ A * card U-loop
  shows concrete-capped-step-core-cost k cap d S B costs ≤
    level-range-cost-bound A R (Suc l) U-loop
proof −
  let ?P = find-pivots-pivots-capped k cap d S B
  have children:
    list-all2 (λU X. U = X ∧ complete-on d' U) Us
      (range-tree-chain-list ?P a bs B')
    using trace unfolding concrete-partition-loop-trace-def by blast
  have U-def: U-loop = bound-tree ?P (Fin a) ∪ ⋃(set Us)
    using trace unfolding concrete-partition-loop-trace-def by blast
  have Union-eq:
    ⋃(set Us) = ⋃(set (range-tree-chain-list ?P a bs B'))
    using children by (induction rule: list-all2-induct) auto
  have U-eq-chain:
    U-loop = bound-tree ?P (Fin a) ∪ range-tree-chain ?P a bs B'
    using U-def Union-eq Union-range-tree-chain-list[of ?P a bs B'] by simp
  have finite-U: finite U-loop
    using U-eq-chain by simp
  have chain-subset: range-tree-chain ?P a bs B' ⊆ U-loop
    using U-eq-chain by blast
  have card-chain-le: card (range-tree-chain ?P a bs B') ≤ card U-loop
    by (rule card-mono[OF finite-U chain-subset])
  have outgoing-subset:
    outgoing-edges (range-tree-chain ?P a bs B') ⊆ outgoing-edges U-loop
    using outgoing-edges-mono[OF chain-subset] .
  have finite-out-U: finite (outgoing-edges U-loop)
    by simp
  have card-out-le:
    card (outgoing-edges (range-tree-chain ?P a bs B')) ≤
      card (outgoing-edges U-loop)
    by (rule card-mono[OF finite-out-U outgoing-subset])
  have step:
```

```
    concrete-capped-step-core-cost k cap d S B costs ≤
      k * Δ * cap +
      A * l * card (range-tree-chain ?P a bs B') +
      R * card (outgoing-edges (range-tree-chain ?P a bs B'))
    by (rule concrete-capped-step-core-cost-le-level-chain
      [OF S-subset degree S-cap trace child-costs])
  also have ... ≤
      A * card U-loop + A * l * card U-loop +
      R * card (outgoing-edges U-loop)
    using scan-budget card-chain-le card-out-le
    by (intro add-mono mult-left-mono) simp-all
  also have ... = level-range-cost-bound A R (Suc l) U-loop
    unfolding level-range-cost-bound-def by (simp add: algebra-simps)
  finally show ?thesis .
qed

lemma scan-budget-from-loop-card:
  assumes degree-factor: Δ ≤ A
    and progress: k * cap ≤ card U-loop
  shows k * Δ * cap ≤ A * card U-loop
proof −
  have k * Δ * cap ≤ k * A * cap
    using degree-factor by (intro mult-left-mono mult-right-mono) simp-all
  also have ... = A * (k * cap)
    by (simp add: algebra-simps)
  also have ... ≤ A * card U-loop
    using progress by simp
  finally show ?thesis .
qed

lemma scan-insert-budget-from-loop-card:
  assumes degree-factor: Δ ≤ A
    and insert-factor: t ≤ A * k
    and progress: k * cap ≤ card U-loop
  shows k * Δ * cap + t * cap ≤ 2 * A * card U-loop
proof −
  have scan: k * Δ * cap ≤ A * card U-loop
    by (rule scan-budget-from-loop-card[OF degree-factor progress])
  have insert: t * cap ≤ A * card U-loop
  proof −
    have t * cap ≤ (A * k) * cap
      using insert-factor by simp
    also have ... = A * (k * cap)
      by (simp add: algebra-simps)
    also have ... ≤ A * card U-loop
      using progress by simp
    finally show ?thesis .
  qed
  have k * Δ * cap + t * cap ≤ A * card U-loop + A * card U-loop
```

```
    using scan insert by simp
  also have ... = 2 * A * card U-loop
    by simp
  finally show ?thesis .
qed

lemma loop-overhead-budget-from-progress:
  assumes factor: Suc t ≤ A
    and progress: M * q ≤ card U
  shows (Suc t) * M * q ≤ A * card U
proof −
  have (Suc t) * M * q ≤ A * (M * q)
  proof −
    have (Suc t) * (M * q) ≤ A * (M * q)
      by (rule mult-right-mono[OF factor]) simp
    then show ?thesis
      by (simp add: algebra-simps)
  qed
  also have ... ≤ A * card U
    using progress by simp
  finally show ?thesis .
qed

theorem concrete-capped-step-core-cost-closes-level-bound-from-progress:
  assumes S-subset: S ⊆ V
    and degree: edge-outdegree-le Δ
    and S-cap: card S ≤ cap
    and trace: concrete-partition-loop-trace
      (find-pivots-pivots-capped k cap d S B) B a bs d' B' Us U-loop
    and child-costs: list-all2 (λc X. c ≤ level-range-cost-bound A R l X) costs
      (range-tree-chain-list (find-pivots-pivots-capped k cap d S B) a bs B')
    and degree-factor: Δ ≤ A
    and progress: k * cap ≤ card U-loop
  shows concrete-capped-step-core-cost k cap d S B costs ≤
    level-range-cost-bound A R (Suc l) U-loop
proof −
  have scan-budget: k * Δ * cap ≤ A * card U-loop
    by (rule scan-budget-from-loop-card[OF degree-factor progress])
  show ?thesis
    by (rule concrete-capped-step-core-cost-closes-level-bound
      [OF S-subset degree S-cap trace child-costs scan-budget])
qed

theorem concrete-capped-step-core-cost-le-loop-vertices:
  assumes S-subset: S ⊆ V
    and degree: edge-outdegree-le Δ
    and S-cap: card S ≤ cap
    and trace: concrete-partition-loop-trace
      (find-pivots-pivots-capped k cap d S B) B a bs d' B' Us U-loop
```

```
    and child-costs: list-all2 (λc X. c ≤ C * card X) costs
      (range-tree-chain-list (find-pivots-pivots-capped k cap d S B) a bs B')
  shows concrete-capped-step-core-cost k cap d S B costs ≤
    k * Δ * cap + C * card U-loop
proof −
  let ?P = find-pivots-pivots-capped k cap d S B
  have children:
    list-all2 (λU X. U = X ∧ complete-on d' U) Us
      (range-tree-chain-list ?P a bs B')
    using trace unfolding concrete-partition-loop-trace-def by blast
  have U-def: U-loop = bound-tree ?P (Fin a) ∪ ⋃(set Us)
    using trace unfolding concrete-partition-loop-trace-def by blast
  have Union-eq:
    ⋃(set Us) = ⋃(set (range-tree-chain-list ?P a bs B'))
    using children by (induction rule: list-all2-induct) auto
  have U-eq-chain:
    U-loop = bound-tree ?P (Fin a) ∪ range-tree-chain ?P a bs B'
    using U-def Union-eq Union-range-tree-chain-list[of ?P a bs B'] by simp
  have finite-U: finite U-loop
    using U-eq-chain by simp
  have chain-subset: range-tree-chain ?P a bs B' ⊆ U-loop
    using U-eq-chain by blast
  have card-chain-le: card (range-tree-chain ?P a bs B') ≤ card U-loop
    by (rule card-mono[OF finite-U chain-subset])
  have concrete-capped-step-core-cost k cap d S B costs ≤
      k * Δ * cap + C * card (range-tree-chain ?P a bs B')
    by (rule concrete-capped-step-core-cost-le-range
      [OF S-subset degree S-cap trace child-costs])
  also have ... ≤ k * Δ * cap + C * card U-loop
    using card-chain-le by simp
  finally show ?thesis .
qed

theorem concrete-capped-step-accounted-cost-le:
  assumes S-subset: S ⊆ V
    and degree: edge-outdegree-le Δ
    and S-cap: card S ≤ cap
    and trace: concrete-partition-loop-trace
      (find-pivots-pivots-capped k cap d S B) B a bs d' B' Us U-loop
    and child-costs: list-all2 (λc X. c ≤ C * card X) child-costs
      (range-tree-chain-list (find-pivots-pivots-capped k cap d S B) a bs B')
    and edge-costs: list-all2 (λc X. c ≤ R * card (outgoing-edges X)) edge-costs
      (range-tree-chain-list (find-pivots-pivots-capped k cap d S B) a bs B')
  shows concrete-capped-step-accounted-cost k cap d S B child-costs edge-costs ≤
    k * Δ * cap + C * card U-loop +
    R * card (outgoing-edges
      (range-tree-chain (find-pivots-pivots-capped k cap d S B) a bs B'))
proof −
  let ?P = find-pivots-pivots-capped k cap d S B
```

```
  have mono: nondecreasing-from a bs
    using trace unfolding concrete-partition-loop-trace-def by blast
  have core:
    concrete-capped-step-core-cost k cap d S B child-costs ≤
      k * Δ * cap + C * card U-loop
    by (rule concrete-capped-step-core-cost-le-loop-vertices
      [OF S-subset degree S-cap trace child-costs])
  have core-unfolded:
    fp-iter-capped-scan-cost k cap d S S B + sum-list child-costs ≤
      k * Δ * cap + C * card U-loop
    using core unfolding concrete-capped-step-core-cost-def .
  have edges:
    sum-list edge-costs ≤ R * card (outgoing-edges (range-tree-chain ?P a bs B'))
    using edge-costs-le-range-tree-chain-outgoing[OF mono edge-costs] .
  show ?thesis
    unfolding concrete-capped-step-accounted-cost-def
    using core-unfolded edges by linarith
qed

theorem full-operational-partition-loop-state-trace-and-degree-cost:
  assumes run:
    full-operational-partition-loop-state M t k cap l d P B d' D a betas bs B' Us U c
    and sound: sound-label d
    and pre: bmssp-pre-full d P B
    and P-reaches: ⋀x. x ∈ P ⟹ reachable s x
    and degree: edge-outdegree-le Δ
  obtains child-costs child-sets where
    concrete-partition-loop-trace P B a bs d' B' Us U
    length child-costs = length child-sets
    ⋀X. X ∈ set child-sets ⟹ X ⊆ V
    c ≤ sum-list child-costs + M * length child-costs +
      t * (Δ * sum-list (map card child-sets) + M * length child-costs)
proof −
  obtain child-costs child-sets where trace:
      concrete-partition-loop-trace P B a bs d' B' Us U
    and len: length child-costs = length child-sets
    and sets: ⋀X. X ∈ set child-sets ⟹ X ⊆ V
    and cost:
      c ≤ sum-list child-costs + M * length child-costs +
        t * (sum-list (map (λX. card (outgoing-edges X)) child-sets) +
          M * length child-costs)
    using full-operational-partition-loop-state-trace-and-complete-edge-cost
      [OF run sound pre P-reaches] by blast
  have edge-sum:
    sum-list (map (λX. card (outgoing-edges X)) child-sets) ≤
      Δ * sum-list (map card child-sets)
    by (rule sum-card-outgoing-edges-le-degree[OF degree sets])
  have cost-degree:
    c ≤ sum-list child-costs + M * length child-costs +
```

```
    t * (Δ * sum-list (map card child-sets) + M * length child-costs)
  proof −
    have sum-list child-costs + M * length child-costs +
      t * (sum-list (map (λX. card (outgoing-edges X)) child-sets) +
        M * length child-costs) ≤
      sum-list child-costs + M * length child-costs +
      t * (Δ * sum-list (map card child-sets) + M * length child-costs)
      using edge-sum by simp
    then show ?thesis
      using cost by linarith
  qed
  show thesis
    using that trace len sets cost-degree by blast
qed

definition partition-initial-insert-cost-bound where
  partition-initial-insert-cost-bound c t P ⟷ c ≤ t * card P

lemma partition-initial-insert-cost-boundD:
  assumes partition-initial-insert-cost-bound c t P
  shows c ≤ t * card P
  using assms unfolding partition-initial-insert-cost-bound-def .

lemma partition-initial-insert-cost-bound-le:
  assumes cost: partition-initial-insert-cost-bound c t P
    and card-le: card P ≤ N
  shows c ≤ t * N
proof −
  have c ≤ t * card P
    using cost by (rule partition-initial-insert-cost-boundD)
  also have ... ≤ t * N
    using card-le by simp
  finally show ?thesis .
qed

lemma find-pivots-pivots-capped-subset:
  find-pivots-pivots-capped k cap d S B ⊆ S
  unfolding find-pivots-pivots-capped-def by auto

lemma find-pivots-pivots-capped-card-le:
  assumes S-subset: S ⊆ V
  shows card (find-pivots-pivots-capped k cap d S B) ≤ card S
proof −
  have finite-S: finite S
    using finite-subset[OF S-subset finite-V] .
  show ?thesis
    by (rule card-mono[OF finite-S find-pivots-pivots-capped-subset])
qed
```

lemma *find-pivots-pivots-capped-scaled-card-le*:
  assumes *S-subset*: $S \subseteq V$
    and *S-k-cap*: $k * card\ S \leq cap$
  shows $k * card\ (find\text{-}pivots\text{-}pivots\text{-}capped\ k\ cap\ d\ S\ B) \leq cap$
proof −
  have $card\ (find\text{-}pivots\text{-}pivots\text{-}capped\ k\ cap\ d\ S\ B) \leq card\ S$
    by (*rule find-pivots-pivots-capped-card-le*[*OF S-subset*])
  then have $k * card\ (find\text{-}pivots\text{-}pivots\text{-}capped\ k\ cap\ d\ S\ B) \leq$
      $k * card\ S$
    by *simp*
  then show *?thesis*
    using *S-k-cap* **by** *linarith*
qed

lemma *find-pivots-pivots-capped-tree-antichain*:
  assumes *anti*: *tree-antichain S*
  shows *tree-antichain* (*find-pivots-pivots-capped k cap d S B*)
  by (*rule tree-antichain-subset*[*OF anti find-pivots-pivots-capped-subset*])

lemma *partition-initial-insert-cost-bound-capped-pivots-le*:
  assumes *cost*: *partition-initial-insert-cost-bound c t*
      (*find-pivots-pivots-capped k cap d S B*)
    and *S-subset*: $S \subseteq V$
    and *S-cap*: $card\ S \leq cap$
  shows $c \leq t * cap$
proof −
  have *pivots-S*:
    $card\ (find\text{-}pivots\text{-}pivots\text{-}capped\ k\ cap\ d\ S\ B) \leq card\ S$
    by (*rule find-pivots-pivots-capped-card-le*[*OF S-subset*])
  have *pivots-le*: $card\ (find\text{-}pivots\text{-}pivots\text{-}capped\ k\ cap\ d\ S\ B) \leq cap$
    by (*rule order-trans*[*OF pivots-S S-cap*])
  show *?thesis*
    by (*rule partition-initial-insert-cost-bound-le*[*OF cost pivots-le*])
qed

lemma *find-pivots-scan-and-initial-insert-cost-le*:
  assumes *S-subset*: $S \subseteq V$
    and *degree*: *edge-outdegree-le* Δ
    and *S-cap*: $card\ S \leq cap$
    and *insert*: *partition-initial-insert-cost-bound c-insert t*
      (*find-pivots-pivots-capped k cap d S B*)
  shows *fp-iter-capped-scan-cost k cap d S S B* + *c-insert* ≤
    $k * \Delta * cap + t * cap$
proof −
  have *scan*: *fp-iter-capped-scan-cost k cap d S S B* ≤ $k * \Delta * cap$
    by (*rule find-pivots-capped-scan-cost-le*[*OF S-subset degree S-cap*])
  have *insert-le*: *c-insert* ≤ $t * cap$
    by (*rule partition-initial-insert-cost-bound-capped-pivots-le*
      [*OF insert S-subset S-cap*])

```
  show ?thesis
    using scan insert-le by simp
qed

lemma find-pivots-scan-and-initial-insert-cost-le-edge-count:
  assumes S-subset: S ⊆ V
    and S-cap: card S ≤ cap
    and insert: partition-initial-insert-cost-bound c-insert t
      (find-pivots-pivots-capped k cap d S B)
  shows fp-iter-capped-scan-cost k cap d S S B + c-insert ≤
    k * edge-count * cap + t * cap
  by (rule find-pivots-scan-and-initial-insert-cost-le
    [OF S-subset edge-outdegree-le-edge-count S-cap insert])

lemma find-pivots-scan-and-initial-insert-budget-from-progress:
  assumes S-subset: S ⊆ V
    and degree: edge-outdegree-le Δ
    and S-cap: card S ≤ cap
    and insert: partition-initial-insert-cost-bound c-insert t
      (find-pivots-pivots-capped k cap d S B)
    and degree-factor: Δ ≤ A
    and insert-factor: t ≤ A * k
    and progress: k * cap ≤ card U
  shows fp-iter-capped-scan-cost k cap d S S B + c-insert ≤
    2 * A * card U
proof −
  have local:
    fp-iter-capped-scan-cost k cap d S S B + c-insert ≤
      k * Δ * cap + t * cap
    by (rule find-pivots-scan-and-initial-insert-cost-le
      [OF S-subset degree S-cap insert])
  have budget: k * Δ * cap + t * cap ≤ 2 * A * card U
    by (rule scan-insert-budget-from-loop-card
      [OF degree-factor insert-factor progress])
  show ?thesis
    using local budget by linarith
qed

lemma find-pivots-pivots-capped-card-le-seen-if-no-overflow:
  assumes S-subset: S ⊆ V
    and anti: tree-antichain S
    and k-pos: 0 < k
    and no-overflow:
      ¬ card (find-pivots-seen-capped k cap d S B) > cap
  shows card (find-pivots-pivots-capped k cap d S B) ≤
    card (find-pivots-seen-capped k cap d S B)
proof −
  let ?P = find-pivots-pivots-capped k cap d S B
  let ?W = find-pivots-seen-capped k cap d S B
```

```
  have cases:
      k * card ?P ≤ card ?W ∨ (?P = S ∧ card ?W > cap)
    by (rule find-pivots-pivots-capped-card-times-le-seen-or-overflow
      [OF S-subset anti])
  then have mult-le: k * card ?P ≤ card ?W
    using no-overflow by blast
  have card ?P ≤ k * card ?P
    using k-pos by (cases k) simp-all
  then show ?thesis
    using mult-le by linarith
qed

lemma partition-initial-insert-cost-bound-capped-pivots-le-seen-scaled-from-mult:
  assumes insert: partition-initial-insert-cost-bound c t P
    and pivot-mult: k * card P ≤ card W
    and insert-factor: t ≤ A * k
  shows c ≤ A * card W
proof −
  have c ≤ t * card P
    by (rule partition-initial-insert-cost-boundD[OF insert])
  also have ... ≤ (A * k) * card P
    using insert-factor by simp
  also have ... = A * (k * card P)
    by (simp add: algebra-simps)
  also have ... ≤ A * card W
    using pivot-mult by simp
  finally show ?thesis .
qed

lemma find-pivots-pivots-capped-card-times-le-seen:
  assumes S-subset: S ⊆ V
    and anti: tree-antichain S
    and S-k-cap: k * card S ≤ cap
  shows k * card (find-pivots-pivots-capped k cap d S B) ≤
    card (find-pivots-seen-capped k cap d S B)
proof −
  let ?P = find-pivots-pivots-capped k cap d S B
  let ?W = find-pivots-seen-capped k cap d S B
  have cases: k * card ?P ≤ card ?W ∨ (?P = S ∧ card ?W > cap)
    by (rule find-pivots-pivots-capped-card-times-le-seen-or-overflow
      [OF S-subset anti])
  then show ?thesis
  proof
    assume k * card ?P ≤ card ?W
    then show ?thesis .
  next
    assume overflow: ?P = S ∧ card ?W > cap
    then have k * card ?P ≤ cap
      using S-k-cap by simp
```

```
    also have ... ≤ card ?W
      using overflow by simp
    finally show ?thesis .
  qed
qed

lemma partition-initial-insert-cost-bound-capped-pivots-le-seen-scaled:
  assumes insert: partition-initial-insert-cost-bound c t
      (find-pivots-pivots-capped k cap d S B)
    and S-subset: S ⊆ V
    and anti: tree-antichain S
    and S-k-cap: k * card S ≤ cap
    and insert-factor: t ≤ A * k
  shows c ≤ A * card (find-pivots-seen-capped k cap d S B)
proof −
  have pivot-mult:
    k * card (find-pivots-pivots-capped k cap d S B) ≤
      card (find-pivots-seen-capped k cap d S B)
    by (rule find-pivots-pivots-capped-card-times-le-seen
      [OF S-subset anti S-k-cap])
  show ?thesis
    by (rule partition-initial-insert-cost-bound-capped-pivots-le-seen-scaled-from-mult
      [OF insert pivot-mult insert-factor])
qed

lemma partition-initial-insert-cost-bound-capped-pivots-le-seen-if-no-overflow:
  assumes insert: partition-initial-insert-cost-bound c t
      (find-pivots-pivots-capped k cap d S B)
    and S-subset: S ⊆ V
    and anti: tree-antichain S
    and k-pos: 0 < k
    and no-overflow:
      ¬ card (find-pivots-seen-capped k cap d S B) > cap
  shows c ≤ t * card (find-pivots-seen-capped k cap d S B)
proof −
  have pivots-le:
    card (find-pivots-pivots-capped k cap d S B) ≤
      card (find-pivots-seen-capped k cap d S B)
    by (rule find-pivots-pivots-capped-card-le-seen-if-no-overflow
      [OF S-subset anti k-pos no-overflow])
  show ?thesis
    by (rule partition-initial-insert-cost-bound-le[OF insert pivots-le])
qed

lemma partition-initial-insert-cost-bound-capped-pivots-le-seen:
  assumes insert: partition-initial-insert-cost-bound c t
      (find-pivots-pivots-capped k cap d S B)
    and S-subset: S ⊆ V
    and S-cap: card S ≤ cap
```

```
    and anti: tree-antichain S
    and k-pos: 0 < k
  shows c ≤ t * card (find-pivots-seen-capped k cap d S B)
proof (cases card (find-pivots-seen-capped k cap d S B) > cap)
  case True
  have c-le-cap: c ≤ t * cap
    by (rule partition-initial-insert-cost-bound-capped-pivots-le
      [OF insert S-subset S-cap])
  have t * cap ≤ t * card (find-pivots-seen-capped k cap d S B)
    using True by simp
  then show ?thesis
    using c-le-cap by linarith
next
  case False
  show ?thesis
    by (rule partition-initial-insert-cost-bound-capped-pivots-le-seen-if-no-overflow
      [OF insert S-subset anti k-pos False])
qed

lemma find-pivots-scan-and-initial-insert-cost-le-seen-if-no-overflow:
  assumes S-subset: S ⊆ V
    and degree: edge-outdegree-le Δ
    and anti: tree-antichain S
    and k-pos: 0 < k
    and no-overflow:
      ¬ card (find-pivots-seen-capped k cap d S B) > cap
    and insert: partition-initial-insert-cost-bound c-insert t
      (find-pivots-pivots-capped k cap d S B)
  shows fp-iter-capped-scan-cost k cap d S S B + c-insert ≤
    (k * Δ + t) * card (find-pivots-seen-capped k cap d S B)
proof −
  let ?W = find-pivots-seen-capped k cap d S B
  have scan: fp-iter-capped-scan-cost k cap d S S B ≤
      k * Δ * card ?W
    by (rule find-pivots-capped-scan-cost-le-seen[OF S-subset degree])
  have insert-le: c-insert ≤ t * card ?W
    by (rule partition-initial-insert-cost-bound-capped-pivots-le-seen-if-no-overflow
      [OF insert S-subset anti k-pos no-overflow])
  have fp-iter-capped-scan-cost k cap d S S B + c-insert ≤
      k * Δ * card ?W + t * card ?W
    using scan insert-le by simp
  also have ... = (k * Δ + t) * card ?W
    by (simp add: algebra-simps)
  finally show ?thesis .
qed

lemma find-pivots-scan-and-initial-insert-cost-le-seen:
  assumes S-subset: S ⊆ V
    and degree: edge-outdegree-le Δ
```

```
    and S-cap: card S ≤ cap
    and anti: tree-antichain S
    and k-pos: 0 < k
    and insert: partition-initial-insert-cost-bound c-insert t
      (find-pivots-pivots-capped k cap d S B)
  shows fp-iter-capped-scan-cost k cap d S S B + c-insert ≤
    (k * Δ + t) * card (find-pivots-seen-capped k cap d S B)
proof −
  let ?W = find-pivots-seen-capped k cap d S B
  have scan: fp-iter-capped-scan-cost k cap d S S B ≤
      k * Δ * card ?W
    by (rule find-pivots-capped-scan-cost-le-seen[OF S-subset degree])
  have insert-le: c-insert ≤ t * card ?W
    by (rule partition-initial-insert-cost-bound-capped-pivots-le-seen
      [OF insert S-subset S-cap anti k-pos])
  have fp-iter-capped-scan-cost k cap d S S B + c-insert ≤
      k * Δ * card ?W + t * card ?W
    using scan insert-le by simp
  also have ... = (k * Δ + t) * card ?W
    by (simp add: algebra-simps)
  finally show ?thesis .
qed

lemma find-pivots-scan-and-initial-insert-cost-le-seen-scaled:
  assumes S-subset: S ⊆ V
    and degree: edge-outdegree-le Δ
    and anti: tree-antichain S
    and S-k-cap: k * card S ≤ cap
    and insert: partition-initial-insert-cost-bound c-insert t
      (find-pivots-pivots-capped k cap d S B)
    and insert-factor: t ≤ A-insert * k
  shows fp-iter-capped-scan-cost k cap d S S B + c-insert ≤
    (k * Δ + A-insert) * card (find-pivots-seen-capped k cap d S B)
proof −
  let ?W = find-pivots-seen-capped k cap d S B
  have scan: fp-iter-capped-scan-cost k cap d S S B ≤
      k * Δ * card ?W
    by (rule find-pivots-capped-scan-cost-le-seen[OF S-subset degree])
  have insert-le: c-insert ≤ A-insert * card ?W
    by (rule partition-initial-insert-cost-bound-capped-pivots-le-seen-scaled
      [OF insert S-subset anti S-k-cap insert-factor])
  have fp-iter-capped-scan-cost k cap d S S B + c-insert ≤
      k * Δ * card ?W + A-insert * card ?W
    using scan insert-le by simp
  also have ... = (k * Δ + A-insert) * card ?W
    by (simp add: algebra-simps)
  finally show ?thesis .
qed
```

lemma *find-pivots-scan-and-initial-insert-budget-from-seen-scaled*:
  assumes *S-subset*: $S \subseteq V$
    and *degree*: *edge-outdegree-le* Δ
    and *anti*: *tree-antichain S*
    and *S-k-cap*: $k * card\ S \leq cap$
    and *insert*: *partition-initial-insert-cost-bound c-insert t*
      (*find-pivots-pivots-capped k cap d S B*)
    and *insert-factor*: $t \leq A\text{-}insert * k$
    and *factor*: $k * \Delta + A\text{-}insert \leq A$
    and *seen-progress*:
      card (*find-pivots-seen-capped k cap d S B*) $\leq$ *card U*
  shows *fp-iter-capped-scan-cost k cap d S S B* + *c-insert* $\leq$
    $A * card\ U$
proof −
  let *?W* = *find-pivots-seen-capped k cap d S B*
  have *local*:
    fp-iter-capped-scan-cost k cap d S S B + *c-insert* $\leq$
      $(k * \Delta + A\text{-}insert) * card\ ?W$
    by (*rule find-pivots-scan-and-initial-insert-cost-le-seen-scaled*
      [*OF S-subset degree anti S-k-cap insert insert-factor*])
  have $(k * \Delta + A\text{-}insert) * card\ ?W \leq A * card\ U$
    by (*rule mult-mono*[*OF factor seen-progress*]) *simp-all*
  then show *?thesis*
    using *local* **by** *linarith*
qed

lemma *find-pivots-scan-and-initial-insert-budget-from-seen*:
  assumes *S-subset*: $S \subseteq V$
    and *degree*: *edge-outdegree-le* Δ
    and *anti*: *tree-antichain S*
    and *k-pos*: $0 < k$
    and *no-overflow*:
      ¬ *card* (*find-pivots-seen-capped k cap d S B*) > *cap*
    and *insert*: *partition-initial-insert-cost-bound c-insert t*
      (*find-pivots-pivots-capped k cap d S B*)
    and *factor*: $k * \Delta + t \leq A$
    and *seen-progress*:
      card (*find-pivots-seen-capped k cap d S B*) $\leq$ *card U*
  shows *fp-iter-capped-scan-cost k cap d S S B* + *c-insert* $\leq$
    $A * card\ U$
proof −
  let *?W* = *find-pivots-seen-capped k cap d S B*
  have *local*:
    fp-iter-capped-scan-cost k cap d S S B + *c-insert* $\leq$
      $(k * \Delta + t) * card\ ?W$
    by (*rule find-pivots-scan-and-initial-insert-cost-le-seen-if-no-overflow*
      [*OF S-subset degree anti k-pos no-overflow insert*])
  have $(k * \Delta + t) * card\ ?W \leq A * card\ U$
    by (*rule mult-mono*[*OF factor seen-progress*]) *simp-all*

```
  then show ?thesis
    using local by linarith
qed

lemma find-pivots-scan-and-initial-insert-budget-from-seen-total:
  assumes S-subset: S ⊆ V
    and degree: edge-outdegree-le Δ
    and S-cap: card S ≤ cap
    and anti: tree-antichain S
    and k-pos: 0 < k
    and insert: partition-initial-insert-cost-bound c-insert t
      (find-pivots-pivots-capped k cap d S B)
    and factor: k * Δ + t ≤ A
    and seen-progress:
      card (find-pivots-seen-capped k cap d S B) ≤ card U
  shows fp-iter-capped-scan-cost k cap d S S B + c-insert ≤
    A * card U
proof −
  let ?W = find-pivots-seen-capped k cap d S B
  have local:
    fp-iter-capped-scan-cost k cap d S S B + c-insert ≤
      (k * Δ + t) * card ?W
    by (rule find-pivots-scan-and-initial-insert-cost-le-seen
      [OF S-subset degree S-cap anti k-pos insert])
  have (k * Δ + t) * card ?W ≤ A * card U
    by (rule mult-mono[OF factor seen-progress]) simp-all
  then show ?thesis
    using local by linarith
qed
```

The relation below is the first costed operational run in this file. It refines the full operational relation by adding a natural-number cost to each BMSSP call and each partition loop. The constructors charge base-case scans, FindPivots scans and initial inserts, pulls, batch prepends, recursive child calls, and the remaining tail of the loop.

This relation is intentionally generic in the partition-cost predicates. It does not know whether a sorted list, bucketed partition, or another data structure implements the primitive operations. Its job is to expose the exact cost skeleton that the later refinement layers and the final headline theorem instantiate.

```
inductive costed-full-operational-partition-loop-state
  and costed-full-operational-bmssp where
  Cost-State-Done:
    keys-of D = {} ⟹
     bound-le (Fin a) B ⟹
     complete-on d' (bound-tree P (Fin a)) ⟹
     costed-full-operational-partition-loop-state Δ M-of t k cap l d P B d' D a
       [] [] (Fin a) [range-tree P a (Fin a)]
```

```
      (bound-tree P (Fin a) ∪ ⋃(set [range-tree P a (Fin a)])) 0
| Cost-State-Step:
    pull-separates D (M-of l) Bmax S-pull beta D-pull ⟹
     bound-le (Fin beta) B ⟹
     bmssp-pre-full d (split-below d P beta) (Fin beta) ⟹
     S-pull = split-below d P beta ⟹
     a ≤ b ⟹
     complete-on d' (bound-tree P (Fin a)) ⟹
     costed-full-operational-bmssp Δ M-of t k cap l d S-pull (Fin beta)
       d-child (Fin b) U-child c-child ⟹
     complete-preserved d-child d' U-child ⟹
     batch =
       edge-relaxation-pairs-between d-child U-child b beta @
       label-pairs-between d S-pull b beta ⟹
     D-next = batch-min-update D-pull batch ⟹
     partition-pull-cost-bound c-pull S-pull ⟹
     partition-batch-cost-bound c-batch t batch ⟹
     costed-full-operational-partition-loop-state Δ M-of t k cap l d P B d'
       D-next b betas bs B' Us-tail U-tail c-tail ⟹
     c = c-pull + c-batch + c-child + c-tail ⟹
     costed-full-operational-partition-loop-state Δ M-of t k cap l d P B d' D a
       (beta # betas) (b # bs) B'
       (range-tree P a (Fin b) # Us-tail)
       (bound-tree P (Fin a) ∪
        ⋃(set (range-tree P a (Fin b) # Us-tail))) c
| Cost-Base:
    S = {x} ⟹
     costed-full-operational-bmssp Δ M-of t k cap 0 d S B
       (λv. if v ∈ base-case-vertices k x B then dist s v else d v)
       (base-case-bound k x B)
       (base-case-vertices k x B)
       (base-case-scan-cost Δ k x B)
| Cost-Step:
    D = label-partition-view
       (find-pivots-label-capped k cap d S B)
       (find-pivots-pivots-capped k cap d S B) ⟹
     partition-initial-insert-cost-bound c-insert t
       (find-pivots-pivots-capped k cap d S B) ⟹
     costed-full-operational-partition-loop-state Δ M-of t k cap l
       (find-pivots-label-capped k cap d S B)
       (find-pivots-pivots-capped k cap d S B) B d' D a betas bs B'
       Us U-loop c-loop ⟹
     complete-on d'
       {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v} ⟹
     U = U-loop ∪
       {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v} ⟹
     c = fp-iter-capped-scan-cost k cap d S S B + c-insert + c-loop ⟹
     costed-full-operational-bmssp Δ M-of t k cap (Suc l) d S B d' B' U c
```

theorem *costed-full-operational-partition-loop-state-refines*:
costed-full-operational-partition-loop-state Δ *M-of t k cap l d P B d' D a*
betas bs B' Us U c $\Longrightarrow$
full-operational-partition-loop-state (*M-of l*) *t k cap l d P B d' D a*
betas bs B' Us U c
and *costed-full-operational-bmssp-refines*:
costed-full-operational-bmssp Δ *M-of t k cap l d S B d' B' U c* $\Longrightarrow$
full-operational-bmssp k cap l d S B d' B' U
proof (*induction rule*:
costed-full-operational-partition-loop-state-costed-full-operational-bmssp.inducts)
case (*Cost-State-Done D a B d' P* Δ *M-of t k cap l d*)
show *?case*
by (*rule full-operational-partition-loop-state.State-Done*
[*OF Cost-State-Done*(*1*) *Cost-State-Done*(*2*) *Cost-State-Done*(*3*)])
next
case (*Cost-State-Step D M-of l Bmax S-pull beta D-pull B d P a b d'*
Δ *t k cap d-child U-child c-child batch D-next c-pull c-batch*
betas bs B' Us-tail U-tail c-tail c)
have *child-run*:
full-operational-bmssp k cap l d S-pull (*Fin beta*)
d-child (*Fin b*) *U-child*
using *Cost-State-Step* **by** *blast*
have *tail-run*:
full-operational-partition-loop-state (*M-of l*) *t k cap l d P B d'*
D-next b betas bs B' Us-tail U-tail c-tail
using *Cost-State-Step* **by** *blast*
show *?case*
proof (*rule full-operational-partition-loop-state.State-Step*)
show *pull-separates D* (*M-of l*) *Bmax S-pull beta D-pull*
using *Cost-State-Step* **by** *blast*
show *bound-le* (*Fin beta*) *B*
using *Cost-State-Step* **by** *blast*
show *bmssp-pre-full d* (*split-below d P beta*) (*Fin beta*)
using *Cost-State-Step* **by** *blast*
show *S-pull* = *split-below d P beta*
using *Cost-State-Step* **by** *blast*
show $a \leq b$
using *Cost-State-Step* **by** *blast*
show *complete-on d'* (*bound-tree P* (*Fin a*))
using *Cost-State-Step* **by** *blast*
show *full-operational-bmssp k cap l d S-pull* (*Fin beta*)
d-child (*Fin b*) *U-child*
using *child-run* **.**
show *complete-preserved d-child d' U-child*
using *Cost-State-Step* **by** *blast*
show *batch* =
edge-relaxation-pairs-between d-child U-child b beta @
label-pairs-between d S-pull b beta
using *Cost-State-Step* **by** *blast*

```
    show D-next = batch-min-update D-pull batch
      using Cost-State-Step by blast
    show partition-pull-cost-bound c-pull S-pull
      using Cost-State-Step by blast
    show partition-batch-cost-bound c-batch t batch
      using Cost-State-Step by blast
    show full-operational-partition-loop-state (M-of l) t k cap l d P B d'
        D-next b betas bs B' Us-tail U-tail c-tail
      using tail-run .
    show c = c-pull + c-batch + c-child + c-tail
      using Cost-State-Step by blast
  qed
next
  case (Cost-Base S x Δ M-of t k cap d B)
  show ?case
    by (rule Full-Base[OF Cost-Base.hyps])
next
  case (Cost-Step D k cap d S B c-insert t Δ M-of l d' a betas bs B'
      Us U-loop c-loop U c)
  have state:
    full-operational-partition-loop-state (M-of l) t k cap l
      (find-pivots-label-capped k cap d S B)
      (find-pivots-pivots-capped k cap d S B) B d' D a betas bs B'
      Us U-loop c-loop
    using Cost-Step by blast
  have loop:
    full-operational-partition-loop k cap l
      (find-pivots-label-capped k cap d S B)
      (find-pivots-pivots-capped k cap d S B) B d' a betas bs B'
      Us U-loop
    by (rule full-operational-partition-loop-state-refines[OF state])
  show ?case
  proof (rule Full-Step)
    show full-operational-partition-loop k cap l
        (find-pivots-label-capped k cap d S B)
        (find-pivots-pivots-capped k cap d S B) B d' a betas bs B'
        Us U-loop
      using loop .
    show complete-on d'
        {v ∈ bound-tree S B'.
         find-pivots-label-capped k cap d S B v = dist s v}
      using Cost-Step by blast
    show U = U-loop ∪
        {v ∈ bound-tree S B'.
         find-pivots-label-capped k cap d S B v = dist s v}
      using Cost-Step by blast
  qed
qed
```

theorem *costed-full-operational-bmssp-correct*:
assumes *run*:
costed-full-operational-bmssp Δ *M-of t k cap l d S B d' B' U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
shows *bmssp-post-full d S B d' B' U*
proof −
have *run'*: *full-operational-bmssp k cap l d S B d' B' U*
by (*rule costed-full-operational-bmssp-refines*[*OF run*])
show *?thesis*
by (*rule full-operational-bmssp-correct*[*OF run' sound pre S-reaches*])
qed

lemma *costed-full-operational-partition-loop-state-lengths*:
assumes *run*:
costed-full-operational-partition-loop-state Δ *M-of t k cap l d P B d' D a*
betas bs B' Us U c
shows *length betas* = *length bs*
and *length Us* = *Suc* (*length bs*)
proof −
have *state*:
full-operational-partition-loop-state (*M-of l*) *t k cap l d P B d' D a*
betas bs B' Us U c
by (*rule costed-full-operational-partition-loop-state-refines*[*OF run*])
show *length betas* = *length bs*
by (*rule full-operational-partition-loop-state-lengths*(*1*)[*OF state*])
show *length Us* = *Suc* (*length bs*)
by (*rule full-operational-partition-loop-state-lengths*(*2*)[*OF state*])
qed

theorem *finite-initial-label-costed-full-operational-top-level-correct*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *run*:
costed-full-operational-bmssp Δ *M-of t k cap l finite-initial-label* {*s*}
Infinity d' Infinity U c
shows *sssp-correct d'*
proof −
have *run'*: *full-operational-bmssp k cap l finite-initial-label* {*s*}
Infinity d' Infinity U
by (*rule costed-full-operational-bmssp-refines*[*OF run*])
show *?thesis*
by (*rule finite-initial-label-full-operational-top-level-correct*
[*OF all-reachable run'*])
qed

theorem *costed-full-operational-partition-loop-state-trace-and-degree-cost*:
assumes *run*:
costed-full-operational-partition-loop-state Δ *M-of t k cap l d P B d' D a*

```
    betas bs B' Us U c
   and sound: sound-label d
   and pre: bmssp-pre-full d P B
   and P-reaches: ⋀x. x ∈ P ⟹ reachable s x
   and degree: edge-outdegree-le Δ
  obtains child-costs child-sets where
   concrete-partition-loop-trace P B a bs d' B' Us U
   length child-costs = length child-sets
   ⋀X. X ∈ set child-sets ⟹ X ⊆ V
   c ≤ sum-list child-costs + (M-of l) * length child-costs +
    t * (Δ * sum-list (map card child-sets) + (M-of l) * length child-costs)
proof −
  have state:
   full-operational-partition-loop-state (M-of l) t k cap l d P B d' D a
    betas bs B' Us U c
   by (rule costed-full-operational-partition-loop-state-refines[OF run])
  show thesis
   by (rule full-operational-partition-loop-state-trace-and-degree-cost
    [OF state sound pre P-reaches degree that])
qed

theorem costed-full-operational-partition-loop-state-child-call-data:
  costed-full-operational-partition-loop-state Δ M-of t k cap l d P B d' D a
     betas bs B' Us U c ⟹
   sound-label d ⟹
   bmssp-pre-full d P B ⟹
   (⋀x. x ∈ P ⟹ reachable s x) ⟹
   ∃ child-costs child-sets.
     length child-costs = length child-sets ∧
     length child-costs = length bs ∧
     list-all2
      (λc-child U-child. ∃ S-child B-child d-child B-child'.
        costed-full-operational-bmssp Δ M-of t k cap l d S-child B-child
         d-child B-child' U-child c-child ∧
        bmssp-pre-full d S-child B-child ∧
        (∀ x∈S-child. reachable s x) ∧
        card S-child ≤ M-of l)
      child-costs child-sets ∧
     (∀ X∈set child-sets. X ⊆ V) ∧
     c ≤ sum-list child-costs + (M-of l) * length child-costs +
      t * (sum-list (map (λX. card (outgoing-edges X)) child-sets) +
       (M-of l) * length child-costs)
and costed-full-operational-bmssp-child-call-data-trivial:
  costed-full-operational-bmssp Δ M-of t k cap l d S B d' B' U c ⟹ True
proof (induction rule:
    costed-full-operational-partition-loop-state-costed-full-operational-bmssp.inducts)
  case (Cost-State-Done D a B d' P Δ M-of t k cap l d)
  show ?case
   by (intro exI[of - []] exI[of - []]) simp
```

next
case (*Cost-State-Step D M-of l Bmax S-pull beta D-pull B d P a b d′*
Δ t k cap d-child U-child c-child batch D-next c-pull c-batch
betas bs B′ Us-tail U-tail c-tail c)
obtain *child-costs child-sets* **where** *len-tail*:
length child-costs = length child-sets
and *child-len-tail*: *length child-costs = length bs*
and *calls-tail*:
list-all2
(*λc-child U-child. ∃ S-child B-child d-child B-child′.*
costed-full-operational-bmssp Δ M-of t k cap l d S-child B-child
d-child B-child′ U-child c-child ∧
bmssp-pre-full d S-child B-child ∧
(∀ x∈S-child. reachable s x) ∧
card S-child ≤ M-of l)
child-costs child-sets
and *sets-tail*: *∀ X∈set child-sets. X ⊆ V*
and *tail*:
*c-tail ≤ sum-list child-costs + M-of l * length child-costs +*
*t * (sum-list (map (λX. card (outgoing-edges X)) child-sets) +*
*M-of l * length child-costs*)
using *Cost-State-Step.IH*(*8*)
[*OF Cost-State-Step.prems*(*1*) *Cost-State-Step.prems*(*2*) *Cost-State-Step.prems*(*3*)]
by *blast*
have *child-reaches*:
⋀x. x ∈ S-pull ⟹ reachable s x
using *Cost-State-Step*
unfolding *split-below-def* **by** *blast*
have *child-pre*: *bmssp-pre-full d S-pull* (*Fin beta*)
using *Cost-State-Step* **by** *blast*
have *child-run*:
costed-full-operational-bmssp Δ M-of t k cap l d S-pull (*Fin beta*)
d-child (*Fin b*) *U-child c-child*
using *Cost-State-Step* **by** *blast*
have *child-post*:
bmssp-post-full d S-pull (*Fin beta*) *d-child* (*Fin b*) *U-child*
by (*rule costed-full-operational-bmssp-correct*
[*OF child-run Cost-State-Step.prems*(*1*) *child-pre child-reaches*])
have *child-set*: *U-child ⊆ V*
using *child-post* **unfolding** *bmssp-post-full-def bound-tree-def* **by** *blast*
have *P-subset*: *P ⊆ V*
using *Cost-State-Step* **unfolding** *bmssp-pre-full-def* **by** *blast*
have *finite-P*: *finite P*
using *finite-subset*[*OF P-subset finite-V*] **.**
have *S-pull-subset*: *S-pull ⊆ P*
using *Cost-State-Step* **unfolding** *split-below-def* **by** *blast*
have *finite-S-pull*: *finite S-pull*
using *finite-subset*[*OF S-pull-subset finite-P*] **.**

have *card-pull*: *card S-pull* $\leq$ *M-of l*
using *Cost-State-Step* **unfolding** *pull-separates-def* **by** *blast*
have *pull-cost*: *c-pull* $\leq$ *M-of l*
using *Cost-State-Step card-pull*
unfolding *partition-pull-cost-bound-def* **by** *linarith*
have *edge-len*:
length (*edge-relaxation-pairs-between d-child U-child b beta*) $\leq$
card (*outgoing-edges U-child*)
by (*rule edge-relaxation-pairs-between-length-le-outgoing*)
have *label-len*:
length (*label-pairs-between d S-pull b beta*) $\leq$ *card S-pull*
by (*rule label-pairs-between-length-le-card*[*OF finite-S-pull*])
have *batch-len*:
length batch $\leq$ *card* (*outgoing-edges U-child*) + *M-of l*
using *Cost-State-Step edge-len label-len card-pull* **by** *simp*
have *batch-cost*:
c-batch $\leq$ *t* $*$ (*card* (*outgoing-edges U-child*) + *M-of l*)
proof −
have *c-batch* $\leq$ *t* $*$ *length batch*
using *Cost-State-Step*
unfolding *partition-batch-cost-bound-def* **by** *simp*
also have ... $\leq$ *t* $*$ (*card* (*outgoing-edges U-child*) + *M-of l*)
using *batch-len* **by** *simp*
finally show *?thesis* **.**
qed
let *?child-costs* = *c-child* # *child-costs*
let *?child-sets* = *U-child* # *child-sets*
have *len*: *length ?child-costs* = *length ?child-sets*
using *len-tail* **by** *simp*
have *child-len*: *length ?child-costs* = *length* (*b* # *bs*)
using *child-len-tail* **by** *simp*
have *call-head*:
$\exists$ *S-child B-child d-child′ B-child′*.
costed-full-operational-bmssp Δ *M-of t k cap l d S-child B-child*
d-child′ B-child′ U-child c-child $\wedge$
bmssp-pre-full d S-child B-child $\wedge$
($\forall$ *x*$\in$*S-child*. *reachable s x*) $\wedge$
card S-child $\leq$ *M-of l*
using *child-run child-pre child-reaches card-pull* **by** *blast*
have *calls*:
list-all2
(λ*c-child U-child*. $\exists$ *S-child B-child d-child B-child′*.
costed-full-operational-bmssp Δ *M-of t k cap l d S-child B-child*
d-child B-child′ U-child c-child $\wedge$
bmssp-pre-full d S-child B-child $\wedge$
($\forall$ *x*$\in$*S-child*. *reachable s x*) $\wedge$
card S-child $\leq$ *M-of l*)
?child-costs ?child-sets
using *call-head calls-tail* **by** *simp*

```
  have sets: ∀ X∈set ?child-sets. X ⊆ V
    using child-set sets-tail by simp
  have cost:
    c ≤ sum-list ?child-costs + M-of l * length ?child-costs +
      t * (sum-list (map (λX. card (outgoing-edges X)) ?child-sets) +
        M-of l * length ?child-costs)
  proof −
    have c-eq: c = c-pull + c-batch + c-child + c-tail
      using Cost-State-Step by blast
    have c-pull + c-batch + c-child + c-tail ≤
        M-of l + t * (card (outgoing-edges U-child) + M-of l) +
        c-child +
        (sum-list child-costs + M-of l * length child-costs +
          t * (sum-list (map (λX. card (outgoing-edges X)) child-sets) +
            M-of l * length child-costs))
      using pull-cost batch-cost tail by simp
    also have ... =
        sum-list ?child-costs + M-of l * length ?child-costs +
        t * (sum-list (map (λX. card (outgoing-edges X)) ?child-sets) +
          M-of l * length ?child-costs)
      by (simp add: algebra-simps)
    finally show ?thesis
      using c-eq by simp
  qed
  show ?case
  proof (intro exI[of - ?child-costs] exI[of - ?child-sets] conjI)
    show length ?child-costs = length ?child-sets
      using len .
    show length ?child-costs = length (b # bs)
      using child-len .
    show list-all2
      (λc-child U-child. ∃ S-child B-child d-child B-child'.
        costed-full-operational-bmssp Δ M-of t k cap l d S-child B-child
          d-child B-child' U-child c-child ∧
        bmssp-pre-full d S-child B-child ∧
        (∀ x∈S-child. reachable s x) ∧
        card S-child ≤ M-of l)
      ?child-costs ?child-sets
      using calls .
    show ∀ X∈set ?child-sets. X ⊆ V
      using sets .
    show c ≤ sum-list ?child-costs + M-of l * length ?child-costs +
      t * (sum-list (map (λX. card (outgoing-edges X)) ?child-sets) +
        M-of l * length ?child-costs)
      using cost .
  qed
next
  case (Cost-Base S x Δ M-of t k cap d B)
  show ?case
```

by *simp*
next
case (*Cost-Step D k cap d S B c-insert t Δ M-of l d′ a betas bs B′ Us U-loop c-loop U c*)
show *?case*
by *simp*
qed

theorem *costed-full-operational-partition-loop-state-child-calls-and-cost*:
assumes *run*:
costed-full-operational-partition-loop-state Δ M-of t k cap l d P B d′ D a betas bs B′ Us U c
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*
obtains *child-costs child-sets* **where**
concrete-partition-loop-trace P B a bs d′ B′ Us U
length child-costs = length child-sets
length child-costs = length bs
list-all2
(λ*c-child U-child*. ∃ *S-child B-child d-child B-child′*.
costed-full-operational-bmssp Δ M-of t k cap l d S-child B-child d-child B-child′ U-child c-child ∧
bmssp-pre-full d S-child B-child ∧
(∀ *x*∈*S-child*. *reachable s x*) ∧
card S-child ≤ *M-of l*)
child-costs child-sets
$\bigwedge X.\ X \in$ *set child-sets* $\Longrightarrow X \subseteq V$
c ≤ *sum-list child-costs* + (*M-of l*) ∗ *length child-costs* +
t ∗ (*sum-list* (*map* (λ*X*. *card* (*outgoing-edges X*)) *child-sets*) +
(*M-of l*) ∗ *length child-costs*)
proof −
have *state*:
full-operational-partition-loop-state (*M-of l*) *t k cap l d P B d′ D a betas bs B′ Us U c*
by (*rule costed-full-operational-partition-loop-state-refines*[*OF run*])
have *loop*:
full-operational-partition-loop k cap l d P B d′ a betas bs B′ Us U
by (*rule full-operational-partition-loop-state-refines*[*OF state*])
have *trace*: *concrete-partition-loop-trace P B a bs d′ B′ Us U*
by (*rule full-operational-partition-loop-trace*[*OF loop sound pre P-reaches*])
have *data*:
∃ *child-costs child-sets*.
length child-costs = length child-sets ∧
length child-costs = length bs ∧
list-all2
(λ*c-child U-child*. ∃ *S-child B-child d-child B-child′*.
costed-full-operational-bmssp Δ M-of t k cap l d S-child B-child d-child B-child′ U-child c-child ∧

bmssp-pre-full d S-child B-child ∧
(∀ *x*∈*S-child*. *reachable s x*) ∧
card S-child ≤ *M-of l*)
child-costs child-sets ∧
(∀ *X*∈*set child-sets*. *X* ⊆ *V*) ∧
c ≤ *sum-list child-costs* + (*M-of l*) ∗ *length child-costs* +
t ∗ (*sum-list* (*map* (λ*X*. *card* (*outgoing-edges X*)) *child-sets*) +
(*M-of l*) ∗ *length child-costs*)
by (*rule costed-full-operational-partition-loop-state-child-call-data*
[*OF run sound pre P-reaches*])
obtain *child-costs child-sets* **where** *len*:
length child-costs = *length child-sets*
and *child-len*: *length child-costs* = *length bs*
and *calls*:
list-all2
(λ*c-child U-child*. ∃ *S-child B-child d-child B-child*′.
costed-full-operational-bmssp Δ *M-of t k cap l d S-child B-child*
d-child B-child′ *U-child c-child* ∧
bmssp-pre-full d S-child B-child ∧
(∀ *x*∈*S-child*. *reachable s x*) ∧
card S-child ≤ *M-of l*)
child-costs child-sets
and *sets*: ∀ *X*∈*set child-sets*. *X* ⊆ *V*
and *cost*:
c ≤ *sum-list child-costs* + (*M-of l*) ∗ *length child-costs* +
t ∗ (*sum-list* (*map* (λ*X*. *card* (*outgoing-edges X*)) *child-sets*) +
(*M-of l*) ∗ *length child-costs*)
using *data* **by** *blast*
show *thesis*
proof (*rule that*[*OF trace len child-len calls - cost*])
fix *X*
assume *X* ∈ *set child-sets*
then show *X* ⊆ *V*
using *sets* **by** *blast*
qed
qed

theorem *costed-full-operational-partition-loop-state-cost-from-child-bound*:
assumes *run*:
costed-full-operational-partition-loop-state Δ *M-of t k cap l d P B d*′ *D a*
betas bs B′ *Us U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: ⋀*x*. *x* ∈ *P* ⟹ *reachable s x*
and *degree*: *edge-outdegree-le* Δ
and *child-bound*:
⋀*c-child U-child S-child B-child d-child B-child*′.
⟦*costed-full-operational-bmssp* Δ *M-of t k cap l d S-child B-child*
d-child B-child′ *U-child c-child*;

```
        bmssp-pre-full d S-child B-child;
        ⋀x. x ∈ S-child ⟹ reachable s x;
        card S-child ≤ M-of l⟧
      ⟹ c-child ≤ C * card U-child
  obtains child-sets where
    concrete-partition-loop-trace P B a bs d' B' Us U
    ⋀X. X ∈ set child-sets ⟹ X ⊆ V
    length child-sets = length bs
    c ≤ (C + t * Δ) * sum-list (map card child-sets) +
      (Suc t) * (M-of l) * length bs
proof −
  have state:
    full-operational-partition-loop-state (M-of l) t k cap l d P B d' D a
      betas bs B' Us U c
    by (rule costed-full-operational-partition-loop-state-refines[OF run])
  have loop:
    full-operational-partition-loop k cap l d P B d' a betas bs B' Us U
    by (rule full-operational-partition-loop-state-refines[OF state])
  have trace:
      concrete-partition-loop-trace P B a bs d' B' Us U
    by (rule full-operational-partition-loop-trace[OF loop sound pre P-reaches])
  have data:
    ∃ child-costs child-sets.
      length child-costs = length child-sets ∧
      length child-costs = length bs ∧
      list-all2
        (λc-child U-child. ∃ S-child B-child d-child B-child'.
          costed-full-operational-bmssp Δ M-of t k cap l d S-child B-child
            d-child B-child' U-child c-child ∧
          bmssp-pre-full d S-child B-child ∧
          (∀ x∈S-child. reachable s x) ∧
          card S-child ≤ M-of l)
        child-costs child-sets ∧
      (∀ X∈set child-sets. X ⊆ V) ∧
      c ≤ sum-list child-costs + (M-of l) * length child-costs +
        t * (sum-list (map (λX. card (outgoing-edges X)) child-sets) +
          (M-of l) * length child-costs)
    by (rule costed-full-operational-partition-loop-state-child-call-data
      [OF run sound pre P-reaches])
  obtain child-costs child-sets where len: length child-costs = length child-sets
    and child-len: length child-costs = length bs
    and calls:
      list-all2
        (λc-child U-child. ∃ S-child B-child d-child B-child'.
          costed-full-operational-bmssp Δ M-of t k cap l d S-child B-child
            d-child B-child' U-child c-child ∧
          bmssp-pre-full d S-child B-child ∧
          (∀ x∈S-child. reachable s x) ∧
          card S-child ≤ M-of l)
```

```
    child-costs child-sets
  and sets: ⋀X. X ∈ set child-sets ⟹ X ⊆ V
  and cost:
    c ≤ sum-list child-costs + (M-of l) * length child-costs +
      t * (sum-list (map (λX. card (outgoing-edges X)) child-sets) +
        (M-of l) * length child-costs)
  using data by blast
have child-sets-len: length child-sets = length bs
  using len child-len by simp
have child-cost-bounds:
  list-all2 (λc-child U-child. c-child ≤ C * card U-child)
    child-costs child-sets
  using calls
proof (induction rule: list-all2-induct)
  case Nil
  then show ?case by simp
next
  case (Cons c-child child-costs U-child child-sets)
  obtain S-child B-child d-child B-child' where child:
      costed-full-operational-bmssp Δ M-of t k cap l d S-child B-child
        d-child B-child' U-child c-child
    and child-pre: bmssp-pre-full d S-child B-child
    and child-reaches: ∀ x∈S-child. reachable s x
    and child-card: card S-child ≤ M-of l
    using Cons.hyps by blast
  have head: c-child ≤ C * card U-child
    by (rule child-bound[OF child child-pre - child-card])
      (use child-reaches in blast)
  show ?case
    using head Cons.IH Cons.hyps by simp
qed
have child-sum:
  sum-list child-costs ≤ C * sum-list (map card child-sets)
proof −
  have sum-list child-costs ≤
      sum-list (map (λX. C * card X) child-sets)
    using sum-list-le-if-list-all2[OF child-cost-bounds] .
  also have ... = C * sum-list (map card child-sets)
    by (induction child-sets) (simp-all add: algebra-simps)
  finally show ?thesis .
qed
have edge-sum:
  sum-list (map (λX. card (outgoing-edges X)) child-sets) ≤
    Δ * sum-list (map card child-sets)
  by (rule sum-card-outgoing-edges-le-degree[OF degree sets])
have c ≤
    C * sum-list (map card child-sets) + (M-of l) * length child-sets +
    t * (Δ * sum-list (map card child-sets) +
      (M-of l) * length child-sets)
```

```
  proof −
    have middle:
      (M-of l) * length child-costs ≤ (M-of l) * length child-sets
      using len by simp
    have inside:
      sum-list (map (λX. card (outgoing-edges X)) child-sets) +
        (M-of l) * length child-costs ≤
       Δ * sum-list (map card child-sets) +
        (M-of l) * length child-sets
      using edge-sum middle by linarith
    have tail-term:
      t * (sum-list (map (λX. card (outgoing-edges X)) child-sets) +
          (M-of l) * length child-costs) ≤
       t * (Δ * sum-list (map card child-sets) +
          (M-of l) * length child-sets)
      using inside by simp
    have sum-list child-costs + (M-of l) * length child-costs +
        t * (sum-list (map (λX. card (outgoing-edges X)) child-sets) +
          (M-of l) * length child-costs) ≤
        C * sum-list (map card child-sets) + (M-of l) * length child-sets +
        t * (Δ * sum-list (map card child-sets) +
          (M-of l) * length child-sets)
      using child-sum middle tail-term by linarith
    then show ?thesis
      using cost by linarith
  qed
  also have ... =
      (C + t * Δ) * sum-list (map card child-sets) +
      (Suc t) * (M-of l) * length child-sets
    by (simp add: algebra-simps)
  finally have final:
    c ≤ (C + t * Δ) * sum-list (map card child-sets) +
      (Suc t) * (M-of l) * length bs
    using child-sets-len by simp
  show thesis
  proof (rule that[OF trace - child-sets-len final])
    fix X
    assume X ∈ set child-sets
    then show X ⊆ V
      using sets by blast
  qed
qed

end

end
theory BMSSP-Range-Costed
  imports BMSSP-Complexity
begin
```

30 Range-Synchronised Costed Runs

The algorithm's time proof charges the recursive calls in one BMSSP loop to the disjoint distance ranges cut out by the successive returned bounds. The earlier costed relation keeps enough information for correctness and local data-structure costs, but its child-call cost is attached to the raw child output. This theory strengthens the operational run with the range invariant: every child output is exactly the next range slice.

That strengthening is the point at which the informal amortised analysis in the paper becomes a reusable Isabelle recurrence. A child call is not charged against the whole parent tree. It is charged against a slice *range-tree* between the previous loop bound and the child output bound. The list version *range-tree-child-list* records all such slices for one parent loop. Since those slices are disjoint and lie inside the final loop output, their vertex and outgoing-edge sums can be bounded by the final output itself.

The exported cost shape is *level-range-cost-bound*. Its first term counts completed vertices multiplied by a level factor; its second term counts outgoing edges with the range cost coefficient. The main technical work below is to show that the local costs of pulling, batching, and recursive child calls close under this shape when moving from level l to *Suc l*.

There are two related relations. The first one uses a single batch cost parameter for the concatenation of edge relaxations and reinserted source labels. The split relation refines it by charging edge batches and source batches separately. That split form is useful for later progress arguments: edge batches are paid for by outgoing edges of the completed ranges, while source batches are paid for by the number of child sources exposed by the partition pulls.

context *unique-shortest-digraph*
begin

lemma *child-costs-le-level-range-child-list-bound*:
  assumes *mono*: *nondecreasing-from a bs*
    and *costs*: *list-all2* ($\lambda c\ X.\ c \leq$ *level-range-cost-bound A R l X*) *costs*
      (*range-tree-child-list S a bs*)
  shows *sum-list costs* $\leq$
    $A * l *$ *card* (*range-tree-chain S a bs B*) $+$
    $R *$ *card* (*outgoing-edges* (*range-tree-chain S a bs B*))
proof $-$
  define *Xs* **where** *Xs* $=$ *range-tree-child-list S a bs*
  have *cost-sum*:
    sum-list costs $\leq$ *sum-list* (*map* (*level-range-cost-bound A R l*) *Xs*)
    using *costs* **unfolding** *Xs-def*
  proof (*induction rule*: *list-all2-induct*)
    case *Nil*
    then show *?case* **by** *simp*
  next

```
    case (Cons c X costs Xs)
    then show ?case
      unfolding level-range-cost-bound-def by simp
  qed
  have split-sum:
    sum-list (map (level-range-cost-bound A R l) Xs) =
      A * l * sum-list (map card Xs) +
      R * sum-list (map (λX. card (outgoing-edges X)) Xs)
  proof (induction Xs)
    case Nil
    then show ?case
      unfolding level-range-cost-bound-def by simp
  next
    case (Cons X Xs)
    then show ?case
      unfolding level-range-cost-bound-def by (simp add: algebra-simps)
  qed
  have card-sum:
    sum-list (map card Xs) ≤ card (range-tree-chain S a bs B)
    unfolding Xs-def
    by (rule card-range-tree-child-list-le-chain[OF mono])
  have edge-sum:
    sum-list (map (λX. card (outgoing-edges X)) Xs) ≤
      card (outgoing-edges (range-tree-chain S a bs B))
    unfolding Xs-def
    by (rule card-outgoing-edges-range-tree-child-list-le-chain[OF mono])
  have sum-list (map (level-range-cost-bound A R l) Xs) ≤
      A * l * card (range-tree-chain S a bs B) +
      R * card (outgoing-edges (range-tree-chain S a bs B))
  proof −
    have left:
      A * l * sum-list (map card Xs) ≤
        A * l * card (range-tree-chain S a bs B)
      using card-sum by simp
    have right:
      R * sum-list (map (λX. card (outgoing-edges X)) Xs) ≤
        R * card (outgoing-edges (range-tree-chain S a bs B))
      using edge-sum by simp
    show ?thesis
      using left right split-sum by linarith
  qed
  then show ?thesis
    using cost-sum by linarith
qed

inductive range-costed-partition-loop-state
  and range-costed-bmssp where
  Range-State-Done:
    keys-of D = {} ⟹
```

bound-le (*Fin a*) *B* ⟹
complete-on d′ (*bound-tree P B*) ⟹
range-costed-partition-loop-state Δ *M-of t k cap l d P B d′ D a*
[] [] *B* [*range-tree P a B*]
(*bound-tree P* (*Fin a*) ∪ ⋃(*set* [*range-tree P a B*])) *0* []
| *Range-State-Stop*:
bound-le (*Fin a*) *B* ⟹
complete-on d′ (*bound-tree P* (*Fin a*)) ⟹
k ∗ *cap* ≤ *card* (*bound-tree P* (*Fin a*)) ⟹
range-costed-partition-loop-state Δ *M-of t k cap l d P B d′ D a*
[] [] (*Fin a*) [*range-tree P a* (*Fin a*)]
(*bound-tree P* (*Fin a*) ∪ ⋃(*set* [*range-tree P a* (*Fin a*)])) *0* []
| *Range-State-Step*:
pull-separates D (*M-of l*) *Bmax S-pull beta D-pull* ⟹
bound-le (*Fin beta*) *B* ⟹
bmssp-pre-full d S-pull (*Fin beta*) ⟹
(∀ *x*∈*S-pull*. *reachable s x*) ⟹
a ≤ *b* ⟹
bound-le (*Fin a*) *B′* ⟹
complete-on d′ (*bound-tree P* (*Fin a*)) ⟹
card (*bound-tree P* (*Fin a*)) < *k* ∗ *cap* ⟹
range-costed-bmssp Δ *M-of t k cap l d S-pull* (*Fin beta*)
d-child (*Fin b*) *U-child c-child* ⟹
U-child = *range-tree P a* (*Fin b*) ⟹
complete-preserved d-child d′ U-child ⟹
batch =
edge-relaxation-pairs-between d-child U-child b beta @
label-pairs-between d S-pull b beta ⟹
D-next = *batch-min-update D-pull batch* ⟹
partition-pull-cost-bound c-pull S-pull ⟹
partition-batch-cost-bound c-batch t batch ⟹
range-costed-partition-loop-state Δ *M-of t k cap l d P B d′*
D-next b betas bs B′ Us-tail U-tail c-tail child-costs-tail ⟹
c = *c-pull* + *c-batch* + *c-child* + *c-tail* ⟹
range-costed-partition-loop-state Δ *M-of t k cap l d P B d′ D a*
(*beta* # *betas*) (*b* # *bs*) *B′*
(*range-tree P a* (*Fin b*) # *Us-tail*)
(*bound-tree P* (*Fin a*) ∪
⋃(*set* (*range-tree P a* (*Fin b*) # *Us-tail*))) *c*
(*c-child* # *child-costs-tail*)
| *Range-Cost-Base*:
S = {*x*} ⟹
range-costed-bmssp Δ *M-of t k cap 0 d S B*
(λ*v*. *if v* ∈ *base-case-vertices k x B then dist s v else d v*)
(*base-case-bound k x B*)
(*base-case-vertices k x B*)
(*base-case-scan-cost* Δ *k x B*)
| *Range-Cost-Step*:
D = *label-partition-view*

```
    (find-pivots-label-capped k cap d S B)
    (find-pivots-pivots-capped k cap d S B) ⟹
  partition-initial-insert-cost-bound c-insert t
   (find-pivots-pivots-capped k cap d S B) ⟹
  range-costed-partition-loop-state Δ M-of t k cap l
    (find-pivots-label-capped k cap d S B)
    (find-pivots-pivots-capped k cap d S B) B d′ D a betas bs B′
    Us U-loop c-loop child-costs-loop ⟹
  complete-on d′
    {v ∈ bound-tree S B′. find-pivots-label-capped k cap d S B v = dist s v} ⟹
  U = U-loop ∪
    {v ∈ bound-tree S B′. find-pivots-label-capped k cap d S B v = dist s v} ⟹
  c = fp-iter-capped-scan-cost k cap d S S B + c-insert + c-loop ⟹
  range-costed-bmssp Δ M-of t k cap (Suc l) d S B d′ B′ U c
```

The mutually inductive relation above adds the accounting data that the operational theory intentionally omitted. A loop derivation carries the pull bounds *betas*, the child output bounds *bs*, the concrete range slices *Us*, a total cost *c*, and the list of recursive child costs. The step constructor synchronises these fields by requiring the returned child set to be exactly *range-tree* for the current interval.

The BMSSP branch keeps the usual split between the base case and the recursive step. In the recursive step, FindPivots pays the scan and initial insertion cost, then the range-synchronised loop pays all pulls, batches, and child calls. The correctness argument later forgets the costs and reuses the operational trace theorem; the runtime argument keeps the cost and child-cost lists.

```
inductive-cases range-costed-bmssp-zeroE:
  range-costed-bmssp Δ M-of t k cap 0 d S B d′ B′ U c

inductive-cases range-costed-bmssp-SucE:
  range-costed-bmssp Δ M-of t k cap (Suc l) d S B d′ B′ U c

lemma range-costed-partition-loop-state-mono:
  assumes run:
    range-costed-partition-loop-state Δ M-of t k cap l d P B d′ D a
      betas bs B′ Us U c child-costs
  shows nondecreasing-from a bs
  using run by (induction) auto

theorem range-costed-partition-loop-state-child-call-ranges:
  assumes run:
    range-costed-partition-loop-state Δ M-of t k cap l d P B d′ D a
      betas bs B′ Us U c child-costs
  shows length child-costs = length bs
    and list-all2
      (λc-child U-child. ∃ S-child B-child d-child B-child′.
        range-costed-bmssp Δ M-of t k cap l d S-child B-child
```

```
        d-child B-child' U-child c-child ∧
        bmssp-pre-full d S-child B-child ∧
        (∀ x∈S-child. reachable s x) ∧
        card S-child ≤ M-of l)
      child-costs (range-tree-child-list P a bs)
  using run
  by (induction) (auto simp: pull-separates-def)

lemma range-costed-pull-cost-le-M:
  assumes pull: pull-separates D M Bmax S beta D-pull
    and cost: partition-pull-cost-bound c S
  shows c ≤ M
proof −
  have card S ≤ M
    using pull unfolding pull-separates-def by blast
  then show ?thesis
    using cost unfolding partition-pull-cost-bound-def by linarith
qed

lemma range-costed-batch-cost-le:
  assumes pre: bmssp-pre-full d S (Fin beta)
    and batch:
      batch =
        edge-relaxation-pairs-between d-child U b beta @
        label-pairs-between d S b beta
    and cost: partition-batch-cost-bound c t batch
    and card-S: card S ≤ M
  shows c ≤ t * (card (outgoing-edges U) + M)
proof −
  have S-subset: S ⊆ V
    using pre unfolding bmssp-pre-full-def by blast
  have finite-S: finite S
    using finite-subset[OF S-subset finite-V] .
  have edge-len:
    length (edge-relaxation-pairs-between d-child U b beta) ≤
      card (outgoing-edges U)
    by (rule edge-relaxation-pairs-between-length-le-outgoing)
  have label-len:
    length (label-pairs-between d S b beta) ≤ card S
    by (rule label-pairs-between-length-le-card[OF finite-S])
  have batch-len: length batch ≤ card (outgoing-edges U) + M
    using batch edge-len label-len card-S by simp
  have c ≤ t * length batch
    using cost unfolding partition-batch-cost-bound-def by simp
  also have ... ≤ t * (card (outgoing-edges U) + M)
    using batch-len by simp
  finally show ?thesis .
qed
```

lemma *range-costed-split-batch-cost-le*:
assumes *pre*: *bmssp-pre-full d S* (*Fin beta*)
and *edge-cost*:
partition-batch-cost-bound c-edges t
(*edge-relaxation-pairs-between d-child U b beta*)
and *source-cost*:
partition-batch-cost-bound c-sources h
(*label-pairs-between d S b beta*)
and *c-def*: $c = c\text{-}edges + c\text{-}sources$
and *card-S*: $card\ S \leq M$
shows $c \leq t * card\ (outgoing\text{-}edges\ U) + h * M$
proof −
have *S-subset*: $S \subseteq V$
using *pre* **unfolding** *bmssp-pre-full-def* **by** *blast*
have *finite-S*: *finite S*
using *finite-subset*[*OF S-subset finite-V*] **.**
have *edge-len*:
$length\ (edge\text{-}relaxation\text{-}pairs\text{-}between\ d\text{-}child\ U\ b\ beta) \leq$
$card\ (outgoing\text{-}edges\ U)$
by (*rule edge-relaxation-pairs-between-length-le-outgoing*)
have *label-len*:
$length\ (label\text{-}pairs\text{-}between\ d\ S\ b\ beta) \leq card\ S$
by (*rule label-pairs-between-length-le-card*[*OF finite-S*])
have *edge-part*: $c\text{-}edges \leq t * card\ (outgoing\text{-}edges\ U)$
proof −
have $c\text{-}edges \leq t * length\ (edge\text{-}relaxation\text{-}pairs\text{-}between\ d\text{-}child\ U\ b\ beta)$
using *edge-cost* **unfolding** *partition-batch-cost-bound-def* **by** *simp*
also have $\ldots \leq t * card\ (outgoing\text{-}edges\ U)$
using *edge-len* **by** *simp*
finally show *?thesis* **.**
qed
have *source-part*: $c\text{-}sources \leq h * M$
proof −
have $c\text{-}sources \leq h * length\ (label\text{-}pairs\text{-}between\ d\ S\ b\ beta)$
using *source-cost* **unfolding** *partition-batch-cost-bound-def* **by** *simp*
also have $\ldots \leq h * card\ S$
using *label-len* **by** *simp*
also have $\ldots \leq h * M$
using *card-S* **by** *simp*
finally show *?thesis* **.**
qed
show *?thesis*
using *edge-part source-part c-def* **by** *linarith*
qed

lemma *range-costed-split-batch-cost-le-card*:
assumes *pre*: *bmssp-pre-full d S* (*Fin beta*)
and *edge-cost*:
partition-batch-cost-bound c-edges t

```
      (edge-relaxation-pairs-between d-child U b beta)
    and source-cost:
      partition-batch-cost-bound c-sources h
        (label-pairs-between d S b beta)
    and c-def: c = c-edges + c-sources
  shows c ≤ t * card (outgoing-edges U) + h * card S
proof −
  have S-subset: S ⊆ V
    using pre unfolding bmssp-pre-full-def by blast
  have finite-S: finite S
    using finite-subset[OF S-subset finite-V] .
  have edge-len:
    length (edge-relaxation-pairs-between d-child U b beta) ≤
      card (outgoing-edges U)
    by (rule edge-relaxation-pairs-between-length-le-outgoing)
  have label-len:
    length (label-pairs-between d S b beta) ≤ card S
    by (rule label-pairs-between-length-le-card[OF finite-S])
  have edge-part: c-edges ≤ t * card (outgoing-edges U)
  proof −
    have c-edges ≤ t * length (edge-relaxation-pairs-between d-child U b beta)
      using edge-cost unfolding partition-batch-cost-bound-def by simp
    also have ... ≤ t * card (outgoing-edges U)
      using edge-len by simp
    finally show ?thesis .
  qed
  have source-part: c-sources ≤ h * card S
  proof −
    have c-sources ≤ h * length (label-pairs-between d S b beta)
      using source-cost unfolding partition-batch-cost-bound-def by simp
    also have ... ≤ h * card S
      using label-len by simp
    finally show ?thesis .
  qed
  show ?thesis
    using edge-part source-part c-def by linarith
qed

lemma range-costed-split-batch-cost-le-range-card:
  assumes pre: bmssp-pre-full d S (Fin beta)
    and edge-cost:
      partition-batch-cost-bound c-edges t
        (edge-relaxation-pairs-between d-child U b beta)
    and source-cost:
      partition-batch-cost-bound c-sources h
        (label-pairs-between d S b beta)
    and complete: complete-on d-child U
    and reaches: ⋀u. u ∈ U ⟹ reachable s u
    and c-def: c = c-edges + c-sources
```

```
  shows c ≤ t * card (outgoing-edges-range U b (Fin beta)) + h * card S
proof −
  have S-subset: S ⊆ V
    using pre unfolding bmssp-pre-full-def by blast
  have finite-S: finite S
    using finite-subset[OF S-subset finite-V] .
  have edge-len:
    length (edge-relaxation-pairs-between d-child U b beta) ≤
      card (outgoing-edges-range U b (Fin beta))
    by (rule edge-relaxation-pairs-between-length-le-outgoing-edges-range
      [OF complete reaches])
  have label-len:
    length (label-pairs-between d S b beta) ≤ card S
    by (rule label-pairs-between-length-le-card[OF finite-S])
  have edge-part:
    c-edges ≤ t * card (outgoing-edges-range U b (Fin beta))
  proof −
    have c-edges ≤ t * length (edge-relaxation-pairs-between d-child U b beta)
      using edge-cost unfolding partition-batch-cost-bound-def by simp
    also have ... ≤ t * card (outgoing-edges-range U b (Fin beta))
      using edge-len by simp
    finally show ?thesis .
  qed
  have source-part: c-sources ≤ h * card S
  proof −
    have c-sources ≤ h * length (label-pairs-between d S b beta)
      using source-cost unfolding partition-batch-cost-bound-def by simp
    also have ... ≤ h * card S
      using label-len by simp
    finally show ?thesis .
  qed
  show ?thesis
    using edge-part source-part c-def by linarith
qed
```

The single-batch relation is convenient, but the sharper progress lemmas need to distinguish where a batch entry came from. Edge entries are generated by *edge-relaxation-pairs-between* and are charged to outgoing edges of the just-completed range. Source entries are generated by *label-pairs-between* and are charged to the number of pulled child sources. The split relation below records both costs explicitly.

This is a refinement, not a different algorithm. Its proof to the single-batch relation uses the append rule for *partition-batch-cost-bound* and the elementary fact that a source-batch coefficient bounded by the edge coefficient can be absorbed into the combined batch budget.

```
inductive split-range-costed-partition-loop-state
  and split-range-costed-bmssp where
  Split-Range-State-Done:
```

keys-of D = {} ⟹
bound-le (Fin a) B ⟹
complete-on d′ (bound-tree P B) ⟹
split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d′ D a
[] [] B [range-tree P a B]
(bound-tree P (Fin a) ∪ ⋃(set [range-tree P a B])) 0 []
| *Split-Range-State-Stop*:
bound-le (Fin a) B ⟹
complete-on d′ (bound-tree P (Fin a)) ⟹
k ∗ cap ≤ card (bound-tree P (Fin a)) ⟹
split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d′ D a
[] [] (Fin a) [range-tree P a (Fin a)]
(bound-tree P (Fin a) ∪ ⋃(set [range-tree P a (Fin a)])) 0 []
| *Split-Range-State-Step*:
pull-separates D (M-of l) Bmax S-pull beta D-pull ⟹
bound-le (Fin beta) B ⟹
bmssp-pre-full d S-pull (Fin beta) ⟹
(∀ x∈S-pull. reachable s x) ⟹
a ≤ b ⟹
bound-le (Fin a) B′ ⟹
complete-on d′ (bound-tree P (Fin a)) ⟹
card (bound-tree P (Fin a)) < k ∗ cap ⟹
split-range-costed-bmssp Δ M-of t h k cap l d S-pull (Fin beta)
d-child (Fin b) U-child c-child ⟹
U-child = range-tree P a (Fin b) ⟹
complete-preserved d-child d′ U-child ⟹
edge-batch = edge-relaxation-pairs-between d-child U-child b beta ⟹
source-batch = label-pairs-between d S-pull b beta ⟹
batch = edge-batch @ source-batch ⟹
D-next = batch-min-update D-pull batch ⟹
partition-pull-cost-bound c-pull S-pull ⟹
partition-batch-cost-bound c-edges t edge-batch ⟹
partition-batch-cost-bound c-sources h source-batch ⟹
split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d′
D-next b betas bs B′ Us-tail U-tail c-tail child-costs-tail ⟹
c = c-pull + c-edges + c-sources + c-child + c-tail ⟹
split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d′ D a
(beta # betas) (b # bs) B′
(range-tree P a (Fin b) # Us-tail)
(bound-tree P (Fin a) ∪
⋃(set (range-tree P a (Fin b) # Us-tail))) c
(c-child # child-costs-tail)
| *Split-Range-Cost-Base*:
S = {x} ⟹
split-range-costed-bmssp Δ M-of t h k cap 0 d S B
(λv. if v ∈ base-case-vertices k x B then dist s v else d v)
(base-case-bound k x B)
(base-case-vertices k x B)
(base-case-scan-cost Δ k x B)

| *Split-Range-Cost-Step*:
D = label-partition-view
(find-pivots-label-capped k cap d S B)
(find-pivots-pivots-capped k cap d S B) ⟹
partition-initial-insert-cost-bound c-insert t
(find-pivots-pivots-capped k cap d S B) ⟹
split-range-costed-partition-loop-state Δ M-of t h k cap l
(find-pivots-label-capped k cap d S B)
(find-pivots-pivots-capped k cap d S B) B d′ D a betas bs B′
Us U-loop c-loop child-costs-loop ⟹
complete-on d′
{v ∈ bound-tree S B′. find-pivots-label-capped k cap d S B v = dist s v} ⟹
U = U-loop ∪
{v ∈ bound-tree S B′. find-pivots-label-capped k cap d S B v = dist s v} ⟹
c = fp-iter-capped-scan-cost k cap d S S B + c-insert + c-loop ⟹
split-range-costed-bmssp Δ M-of t h k cap (Suc l) d S B d′ B′ U c

inductive-cases *split-range-costed-bmssp-zeroE*:
split-range-costed-bmssp Δ M-of t h k cap 0 d S B d′ B′ U c

inductive-cases *split-range-costed-bmssp-SucE*:
split-range-costed-bmssp Δ M-of t h k cap (Suc l) d S B d′ B′ U c

theorem *split-range-costed-partition-loop-state-refines*:
shows
split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d′ D a
betas bs B′ Us U c child-costs ⟹
h ≤ t ⟹
range-costed-partition-loop-state Δ M-of t k cap l d P B d′ D a
betas bs B′ Us U c child-costs
and
split-range-costed-bmssp Δ M-of t h k cap l d S B d′ B′ U c ⟹
h ≤ t ⟹
range-costed-bmssp Δ M-of t k cap l d S B d′ B′ U c
proof (*induction rule*:
split-range-costed-partition-loop-state-split-range-costed-bmssp.inducts)
case (*Split-Range-State-Done D a B d′ P Δ M-of t h k cap l d*)
show *?case*
using *Split-Range-State-Done* **by** (*meson Range-State-Done*)
next
case (*Split-Range-State-Stop a B d′ P k cap Δ M-of t h l d D*)
show *?case*
using *Split-Range-State-Stop* **by** (*meson Range-State-Stop*)
next
case (*Split-Range-State-Step D M-of l Bmax S-pull beta D-pull B d a b B′*
d′ P k cap Δ t h d-child U-child c-child edge-batch source-batch
batch D-next c-pull c-edges c-sources betas bs Us-tail U-tail c-tail
child-costs-tail c)
have *h-le-t*: *h ≤ t*

```
    using Split-Range-State-Step.prems by simp
  have edge-cost':
    partition-batch-cost-bound c-edges t
      (edge-relaxation-pairs-between d-child U-child b beta)
    using Split-Range-State-Step by simp
  have source-cost':
    partition-batch-cost-bound c-sources h
      (label-pairs-between d S-pull b beta)
    using Split-Range-State-Step by simp
  have batch-cost:
    partition-batch-cost-bound
      (c-edges + c-sources) t
      (edge-relaxation-pairs-between d-child U-child b beta @
       label-pairs-between d S-pull b beta)
    by (rule partition-batch-cost-bound-append[OF edge-cost' source-cost' h-le-t])
  have c-eq:
    c = c-pull + (c-edges + c-sources) + c-child + c-tail
    using Split-Range-State-Step by simp
  show ?case
    by (rule Range-State-Step[
        where batch =
          edge-relaxation-pairs-between d-child U-child b beta @
           label-pairs-between d S-pull b beta
        and c-batch = c-edges + c-sources])
      (use Split-Range-State-Step h-le-t batch-cost c-eq in auto)
next
  case (Split-Range-Cost-Base S x Δ M-of t h k cap d B)
  show ?case
    by (rule Range-Cost-Base[OF Split-Range-Cost-Base.hyps])
next
  case Split-Range-Cost-Step
  show ?case
    by (rule Range-Cost-Step)
      (use Split-Range-Cost-Step in auto)
qed
```

The refinement theorem ⟦*split-range-costed-partition-loop-state ?Δ ?M-of ?t ?h ?k ?cap ?l ?d ?P ?B ?d′ ?D ?a ?betas ?bs ?B′ ?Us ?U ?c ?child-costs*; *?h* ≤ *?t*⟧ ⟹ *range-costed-partition-loop-state ?Δ ?M-of ?t ?k ?cap ?l ?d ?P ?B ?d′ ?D ?a ?betas ?bs ?B′ ?Us ?U ?c ?child-costs*

⟦*split-range-costed-bmssp ?Δ ?M-of ?t ?h ?k ?cap ?l ?d ?S ?B ?d′ ?B′ ?U ?c*; *?h* ≤ *?t*⟧ ⟹ *range-costed-bmssp ?Δ ?M-of ?t ?k ?cap ?l ?d ?S ?B ?d′ ?B′ ?U ?c* is the formal connection between the two presentations. Once it is proved, any semantic theorem established for the single-batch relation can be reused for the split relation. The remaining split-specific lemmas therefore focus on costs: extracting child sources, bounding edge and source batches separately, and then recombining those bounds into the level recurrence.

lemma *split-range-costed-partition-loop-state-mono*:

assumes *run*:
split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs
shows *nondecreasing-from a bs*
using *run* **by** (*induction*) *auto*

theorem *split-range-costed-partition-loop-state-child-call-ranges*:
assumes *run*:
split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs
shows *length child-costs* = *length bs*
and *list-all2*
(λ*c-child U-child*. ∃ *S-child B-child d-child B-child′*.
split-range-costed-bmssp Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child ∧
bmssp-pre-full d S-child B-child ∧
(∀ *x*∈*S-child*. *reachable s x*) ∧
card S-child ≤ *M-of l*)
child-costs (*range-tree-child-list P a bs*)
using *run*
by (*induction*) (*auto simp*: *pull-separates-def*)

theorem *split-range-costed-partition-loop-state-cost-bound-by-child-sources*:
split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs ⟹
∃ *child-sources*.
length child-sources = *length bs* ∧
list-all2
(λ*S-child U-child*. ∃ *c-child B-child d-child B-child′*.
split-range-costed-bmssp Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child ∧
bmssp-pre-full d S-child B-child ∧
(∀ *x*∈*S-child*. *reachable s x*) ∧
card S-child ≤ *M-of l*)
child-sources (*range-tree-child-list P a bs*) ∧
c ≤ *sum-list child-costs* + *sum-list* (*map card child-sources*) +
t ∗ *sum-list*
(*map* (λ*X*. *card* (*outgoing-edges X*)) (*range-tree-child-list P a bs*)) +
h ∗ *sum-list* (*map card child-sources*)
and *split-range-costed-bmssp-child-sources-trivial*:
split-range-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c* ⟹ *True*
proof (*induction rule*:
split-range-costed-partition-loop-state-split-range-costed-bmssp.inducts)
case (*Split-Range-State-Done D a B d′ P* Δ *M-of t h k cap l d*)
show *?case*
by (*intro exI*[*of* - []]) *simp*
next
case (*Split-Range-State-Stop a B d′ P k cap* Δ *M-of t h l d D*)
show *?case*

```
    by (intro exI[of - []]) simp
next
  case (Split-Range-State-Step D M-of l Bmax S-pull beta D-pull B d a b B'
      d' P k cap Δ t h d-child U-child c-child edge-batch source-batch
      batch D-next c-pull c-edges c-sources betas bs Us-tail U-tail c-tail
      child-costs-tail c)
  obtain child-sources-tail where len-tail:
      length child-sources-tail = length bs
    and sources-tail:
      list-all2
        (λS-child U-child. ∃ c-child B-child d-child B-child'.
          split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
            d-child B-child' U-child c-child ∧
          bmssp-pre-full d S-child B-child ∧
          (∀ x∈S-child. reachable s x) ∧
          card S-child ≤ M-of l)
        child-sources-tail (range-tree-child-list P b bs)
    and tail-cost:
      c-tail ≤
        sum-list child-costs-tail + sum-list (map card child-sources-tail) +
        t * sum-list
          (map (λX. card (outgoing-edges X)) (range-tree-child-list P b bs)) +
        h * sum-list (map card child-sources-tail)
    using Split-Range-State-Step.IH by blast
  have card-pull: card S-pull ≤ M-of l
    using Split-Range-State-Step unfolding pull-separates-def by blast
  have pull-cost: c-pull ≤ card S-pull
    using Split-Range-State-Step
    unfolding partition-pull-cost-bound-def by blast
  have batch-cost:
    c-edges + c-sources ≤
      t * card (outgoing-edges U-child) + h * card S-pull
    by (rule range-costed-split-batch-cost-le-card[
        where d = d and S = S-pull and d-child = d-child and b = b
          and beta = beta])
      (use Split-Range-State-Step in auto)
  let ?child-sources = S-pull # child-sources-tail
  have len: length ?child-sources = length (b # bs)
    using len-tail by simp
  have head-source:
    ∃ c-child' B-child d-child' B-child'.
      split-range-costed-bmssp Δ M-of t h k cap l d S-pull B-child
        d-child' B-child' U-child c-child' ∧
      bmssp-pre-full d S-pull B-child ∧
      (∀ x∈S-pull. reachable s x) ∧
      card S-pull ≤ M-of l
    using Split-Range-State-Step card-pull by blast
  have sources:
    list-all2
```

```
    (λS-child U-child. ∃ c-child B-child d-child B-child'.
      split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
        d-child B-child' U-child c-child ∧
      bmssp-pre-full d S-child B-child ∧
      (∀ x∈S-child. reachable s x) ∧
      card S-child ≤ M-of l)
    ?child-sources (range-tree-child-list P a (b # bs))
  using head-source sources-tail Split-Range-State-Step by simp
have cost:
  c ≤ sum-list (c-child # child-costs-tail) +
    sum-list (map card ?child-sources) +
    t * sum-list
      (map (λX. card (outgoing-edges X))
        (range-tree-child-list P a (b # bs))) +
    h * sum-list (map card ?child-sources)
proof −
  have c-eq: c = c-pull + c-edges + c-sources + c-child + c-tail
    using Split-Range-State-Step by blast
  have c-pull + c-edges + c-sources + c-child + c-tail ≤
      card S-pull +
      (t * card (outgoing-edges U-child) + h * card S-pull) +
      c-child +
      (sum-list child-costs-tail + sum-list (map card child-sources-tail) +
        t * sum-list
          (map (λX. card (outgoing-edges X)) (range-tree-child-list P b bs)) +
        h * sum-list (map card child-sources-tail))
    using pull-cost batch-cost tail-cost by linarith
  also have ... =
      sum-list (c-child # child-costs-tail) +
      sum-list (map card ?child-sources) +
      t * (card (outgoing-edges U-child) +
        sum-list
          (map (λX. card (outgoing-edges X)) (range-tree-child-list P b bs))) +
      h * sum-list (map card ?child-sources)
    by (simp add: algebra-simps)
  also have ... =
      sum-list (c-child # child-costs-tail) +
      sum-list (map card ?child-sources) +
      t * sum-list
        (map (λX. card (outgoing-edges X))
          (range-tree-child-list P a (b # bs))) +
      h * sum-list (map card ?child-sources)
    using Split-Range-State-Step by simp
  finally show ?thesis
    using c-eq by simp
qed
show ?case
  by (intro exI[of - ?child-sources] conjI)
    (use len sources cost in simp-all)
```

next
case (*Split-Range-Cost-Base S x Δ M-of t h k cap d B*)
show *?case*
by *simp*
next
case (*Split-Range-Cost-Step D k cap d S B c-insert t Δ M-of h l d′ a*
betas bs B′ Us U-loop c-loop child-costs-loop U c)
show *?case*
by *simp*
qed

theorem *split-range-costed-partition-loop-state-cost-from-child-and-source-bounds*:
assumes *run*:
split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d′ D a
betas bs B′ Us U c child-costs
and *child-cost-bounds*:
list-all2 (λ*c-child U-child*.
c-child ≤ *level-range-cost-bound A R L U-child*)
child-costs (*range-tree-child-list P a bs*)
and *source-progress*:
⋀*S-child B-child d-child B-child′ U-child c-child*.
⟦*split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child;
bmssp-pre-full d S-child B-child;
⋀*x*. *x* ∈ *S-child* ⟹ *reachable s x*;
card S-child ≤ *M-of l*⟧
⟹ *card S-child* ≤ *card U-child*
shows *c* ≤
A ∗ *L* ∗ *card* (*range-tree-chain P a bs B′*) +
(*R* + *t*) ∗ *card* (*outgoing-edges* (*range-tree-chain P a bs B′*)) +
Suc h ∗ *card* (*range-tree-chain P a bs B′*)
proof −
obtain *child-sources* **where** *len-sources*:
length child-sources = *length bs*
and *sources*:
list-all2
(λ*S-child U-child*. ∃ *c-child B-child d-child B-child′*.
split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
d-child B-child′ U-child c-child ∧
bmssp-pre-full d S-child B-child ∧
(∀ *x*∈*S-child*. *reachable s x*) ∧
card S-child ≤ *M-of l*)
child-sources (*range-tree-child-list P a bs*)
and *cost*:
c ≤ *sum-list child-costs* + *sum-list* (*map card child-sources*) +
t ∗ *sum-list*
(*map* (λ*X*. *card* (*outgoing-edges X*)) (*range-tree-child-list P a bs*)) +
h ∗ *sum-list* (*map card child-sources*)
using *split-range-costed-partition-loop-state-cost-bound-by-child-sources*[*OF run*]

by *blast*
have *mono*: *nondecreasing-from a bs*
by (*rule split-range-costed-partition-loop-state-mono*[*OF run*])
have *source-dom*:
list-all2 (λ*S-child U-child*. *card S-child* ≤ *card U-child*)
child-sources (*range-tree-child-list P a bs*)
using *sources*
proof (*induction rule*: *list-all2-induct*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons S-child child-sources U-child child-sets*)
obtain *c-child B-child d-child B-child′* **where** *child*:
split-range-costed-bmssp Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child
and *child-pre*: *bmssp-pre-full d S-child B-child*
and *child-reaches*: ∀ *x*∈*S-child*. *reachable s x*
and *child-card*: *card S-child* ≤ *M-of l*
using *Cons.hyps* **by** *blast*
have *head*: *card S-child* ≤ *card U-child*
by (*rule source-progress*[*OF child child-pre - child-card*])
(*use child-reaches* **in** *blast*)
show *?case*
using *head Cons.IH Cons.hyps* **by** *simp*
qed
have *source-sum*:
sum-list (*map card child-sources*) ≤
card (*range-tree-chain P a bs B′*)
by (*rule sum-card-dominated-by-range-tree-child-list-le-chain*
[*OF mono source-dom*])
have *child-sum*:
sum-list child-costs ≤
A ∗ *L* ∗ *card* (*range-tree-chain P a bs B′*) +
R ∗ *card* (*outgoing-edges* (*range-tree-chain P a bs B′*))
by (*rule child-costs-le-level-range-child-list-bound*
[*OF mono child-cost-bounds*])
have *edge-sum*:
sum-list
(*map* (λ*X*. *card* (*outgoing-edges X*)) (*range-tree-child-list P a bs*)) ≤
card (*outgoing-edges* (*range-tree-chain P a bs B′*))
by (*rule card-outgoing-edges-range-tree-child-list-le-chain*[*OF mono*])
have *c* ≤
A ∗ *L* ∗ *card* (*range-tree-chain P a bs B′*) +
R ∗ *card* (*outgoing-edges* (*range-tree-chain P a bs B′*)) +
sum-list (*map card child-sources*) +
t ∗ *card* (*outgoing-edges* (*range-tree-chain P a bs B′*)) +
h ∗ *sum-list* (*map card child-sources*)
proof −
have *edge-part*:

```
      t * sum-list
          (map (λX. card (outgoing-edges X)) (range-tree-child-list P a bs)) ≤
       t * card (outgoing-edges (range-tree-chain P a bs B'))
      using edge-sum by simp
    have sum-list child-costs + sum-list (map card child-sources) +
        t * sum-list
          (map (λX. card (outgoing-edges X)) (range-tree-child-list P a bs)) +
        h * sum-list (map card child-sources) ≤
        A * L * card (range-tree-chain P a bs B') +
        R * card (outgoing-edges (range-tree-chain P a bs B')) +
        sum-list (map card child-sources) +
        t * card (outgoing-edges (range-tree-chain P a bs B')) +
        h * sum-list (map card child-sources)
      using child-sum edge-part by linarith
    then show ?thesis
      using cost by linarith
  qed
  also have ... ≤
      A * L * card (range-tree-chain P a bs B') +
      (R + t) * card (outgoing-edges (range-tree-chain P a bs B')) +
      Suc h * card (range-tree-chain P a bs B')
  proof −
    have source-part:
      sum-list (map card child-sources) +
        h * sum-list (map card child-sources) ≤
       Suc h * card (range-tree-chain P a bs B')
    proof −
      have h-part:
        h * sum-list (map card child-sources) ≤
          h * card (range-tree-chain P a bs B')
        using source-sum by simp
      have sum-list (map card child-sources) +
          h * sum-list (map card child-sources) ≤
        card (range-tree-chain P a bs B') +
          h * card (range-tree-chain P a bs B')
        using source-sum h-part by linarith
      then show ?thesis
        by (simp add: algebra-simps)
    qed
    show ?thesis
      using source-part by (simp add: algebra-simps)
  qed
  finally show ?thesis .
qed

theorem split-range-costed-partition-loop-state-child-cost-bounds:
  assumes run:
    split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
      betas bs B' Us U c child-costs
```

```
    and child-bound:
      ⋀c-child U-child S-child B-child d-child B-child'.
        ⟦split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
            d-child B-child' U-child c-child;
          bmssp-pre-full d S-child B-child;
          ⋀x. x ∈ S-child ⟹ reachable s x;
          card S-child ≤ M-of l⟧
        ⟹ c-child ≤ level-range-cost-bound A R L U-child
  shows list-all2 (λc-child U-child.
      c-child ≤ level-range-cost-bound A R L U-child)
    child-costs (range-tree-child-list P a bs)
proof −
  have calls:
    list-all2
      (λc-child U-child. ∃ S-child B-child d-child B-child'.
        split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
          d-child B-child' U-child c-child ∧
        bmssp-pre-full d S-child B-child ∧
        (∀ x∈S-child. reachable s x) ∧
        card S-child ≤ M-of l)
      child-costs (range-tree-child-list P a bs)
    by (rule split-range-costed-partition-loop-state-child-call-ranges(2)[OF run])
  show ?thesis
    using calls
  proof (induction rule: list-all2-induct)
    case Nil
    then show ?case by simp
  next
    case (Cons c-child child-costs U-child child-sets)
    obtain S-child B-child d-child B-child' where child:
        split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
          d-child B-child' U-child c-child
      and child-pre: bmssp-pre-full d S-child B-child
      and child-reaches: ∀ x∈S-child. reachable s x
      and child-card: card S-child ≤ M-of l
      using Cons.hyps by blast
    have head:
      c-child ≤ level-range-cost-bound A R L U-child
      by (rule child-bound[OF child child-pre - child-card])
        (use child-reaches in blast)
    show ?case
      using head Cons.IH Cons.hyps by simp
  qed
qed

theorem split-range-costed-partition-loop-state-cost-from-child-bound:
  assumes run:
    split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
      betas bs B' Us U c child-costs
```

```
    and child-bound:
      ⋀c-child U-child S-child B-child d-child B-child'.
        ⟦split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
            d-child B-child' U-child c-child;
          bmssp-pre-full d S-child B-child;
          ⋀x. x ∈ S-child ⟹ reachable s x;
          card S-child ≤ M-of l⟧
        ⟹ c-child ≤ level-range-cost-bound A R L U-child
    and source-progress:
      ⋀S-child B-child d-child B-child' U-child c-child.
        ⟦split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
            d-child B-child' U-child c-child;
          bmssp-pre-full d S-child B-child;
          ⋀x. x ∈ S-child ⟹ reachable s x;
          card S-child ≤ M-of l⟧
        ⟹ card S-child ≤ card U-child
  shows c ≤
    A * L * card (range-tree-chain P a bs B') +
    (R + t) * card (outgoing-edges (range-tree-chain P a bs B')) +
    Suc h * card (range-tree-chain P a bs B')
proof −
  have child-cost-bounds:
    list-all2 (λc-child U-child.
      c-child ≤ level-range-cost-bound A R L U-child)
      child-costs (range-tree-child-list P a bs)
    by (rule split-range-costed-partition-loop-state-child-cost-bounds
      [OF run child-bound])
  show ?thesis
    by (rule split-range-costed-partition-loop-state-cost-from-child-and-source-bounds
      [OF run child-cost-bounds source-progress])
qed

theorem split-range-costed-partition-loop-state-trace:
  split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
    betas bs B' Us U c child-costs ⟹
   sound-label d ⟹
   bmssp-pre-full d P B ⟹
   (⋀x. x ∈ P ⟹ reachable s x) ⟹
   concrete-partition-loop-trace P B a bs d' B' Us U
and split-range-costed-bmssp-correct:
  split-range-costed-bmssp Δ M-of t h k cap l d S B d' B' U c ⟹
   sound-label d ⟹
   bmssp-pre-full d S B ⟹
   (⋀x. x ∈ S ⟹ reachable s x) ⟹
   bmssp-post-full d S B d' B' U
proof (induction rule:
    split-range-costed-partition-loop-state-split-range-costed-bmssp.inducts)
  case (Split-Range-State-Done D a B d' P Δ M-of t h k cap l d)
  have a-le: bound-le (Fin a) B
```

```
    using Split-Range-State-Done by blast
  have complete-B: complete-on d' (bound-tree P B)
    using Split-Range-State-Done by blast
  have lower: complete-on d' (bound-tree P (Fin a))
    by (rule complete-on-subset[OF complete-B])
      (rule bound-tree-bound-mono[OF a-le])
  have range-complete: complete-on d' (range-tree P a B)
    by (rule complete-on-subset[OF complete-B])
      (rule range-tree-subset-bound-tree)
  have B-le: bound-le B B
    by (cases B) simp-all
  show ?case
    unfolding concrete-partition-loop-trace-def
    using B-le a-le lower range-complete by simp
next
  case (Split-Range-State-Stop a B d' P k cap Δ M-of t h l d D)
  then show ?case
    unfolding concrete-partition-loop-trace-def complete-on-def by simp
next
  case Split-Range-State-Step
  then show ?case
    unfolding concrete-partition-loop-trace-def bmssp-post-full-def
      complete-preserved-def
    by (auto split: bound.splits)
next
  case (Split-Range-Cost-Base S x Δ M-of t h k cap d B)
  have post:
    bmssp-post d S B
      (λv. if v ∈ base-case-vertices k x B then dist s v else d v)
      (base-case-bound k x B)
      (base-case-vertices k x B)
    using base-case-result-bmssp-post[OF Split-Range-Cost-Base.hyps,
        where k = k and B = B and d = d]
    unfolding base-case-result-def by simp
  then show ?case
    by (rule bmssp-post-imp-post-full)
next
  case (Split-Range-Cost-Step D k cap d S B c-insert t Δ M-of h l d' a
      betas bs B' Us U-loop c-loop child-costs-loop U c)
  let ?d-fp = find-pivots-label-capped k cap d S B
  let ?P = find-pivots-pivots-capped k cap d S B
  let ?W = {v ∈ bound-tree S B'. ?d-fp v = dist s v}
  have sound-fp: sound-label ?d-fp
    unfolding find-pivots-label-capped-def
    by (rule fp-iter-capped-label-sound[OF Split-Range-Cost-Step.prems(1)
      Split-Range-Cost-Step.prems(3)])
  have pivot-pre: bmssp-pre-full ?d-fp ?P B
    using find-pivots-capped-establishes-pivot-pre-concrete
      [OF Split-Range-Cost-Step.prems(1) Split-Range-Cost-Step.prems(2)
```

```
      Split-Range-Cost-Step.prems(3)] .
  have P-subset-S: ?P ⊆ S
    unfolding find-pivots-pivots-capped-def by auto
  have P-reaches: ⋀x. x ∈ ?P ⟹ reachable s x
    using P-subset-S Split-Range-Cost-Step.prems(3) by blast
  have trace:
    concrete-partition-loop-trace ?P B a bs d' B' Us U-loop
    using Split-Range-Cost-Step.IH sound-fp pivot-pre P-reaches by blast
  have loop-post: bmssp-post-full ?d-fp ?P B d' B' U-loop
    by (rule concrete-partition-loop-trace-post[OF trace])
  have le: bound-le B' B
    using loop-post unfolding bmssp-post-full-def by blast
  have U-tree: U = bound-tree S B'
  proof −
    have U-loop ∪ ?W = bound-tree S B'
      by (rule concrete-capped-find-pivots-final-tree-assembly
        [OF Split-Range-Cost-Step.prems(1) Split-Range-Cost-Step.prems(2)
          Split-Range-Cost-Step.prems(3) loop-post])
    then show ?thesis
      using Split-Range-Cost-Step(6) by simp
  qed
  have loop-complete: complete-on d' U-loop
    using loop-post unfolding bmssp-post-full-def by blast
  have U-complete: complete-on d' U
    using complete-on-Un[OF loop-complete Split-Range-Cost-Step(5)]
      Split-Range-Cost-Step(6) by simp
  show ?case
    using le U-tree U-complete unfolding bmssp-post-full-def by blast
qed
```

The next block closes the level recurrence for the split relation. The input hypotheses have the same shape as the informal induction in the paper: each child call already satisfies a level-L bound on its own output slice, and every pulled child source set makes enough progress to be charged to that slice. The trace theorem supplies the geometric fact needed to sum these charges: the range slices form a chain inside the parent output.

Once the slice sums are bounded by the parent output and its outgoing edges, the arithmetic is straightforward. The child level factor contributes $A * L$, the source-progress term contributes one more A, and the edge batch cost contributes the extra t in the range coefficient. This is why the target has level $Suc\ L$ and coefficient $R + t$.

```
theorem split-range-costed-partition-loop-state-closes-level-bound-from-child-cost-bounds-general:
  assumes run:
    split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
      betas bs B' Us U c child-costs
    and sound: sound-label d
    and pre: bmssp-pre-full d P B
```

```
    and P-reaches: ⋀x. x ∈ P ⟹ reachable s x
    and child-cost-bounds:
      list-all2 (λc-child U-child.
        c-child ≤ level-range-cost-bound A R L U-child)
        child-costs (range-tree-child-list P a bs)
    and source-progress:
      ⋀S-child B-child d-child B-child' U-child c-child.
        ⟦split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
            d-child B-child' U-child c-child;
          bmssp-pre-full d S-child B-child;
          ⋀x. x ∈ S-child ⟹ reachable s x;
          card S-child ≤ M-of l⟧
        ⟹ card S-child ≤ card U-child
    and source-factor: Suc h ≤ A
  shows c ≤ A * Suc L * card U + (R + t) * card (outgoing-edges U)
proof −
  have trace: concrete-partition-loop-trace P B a bs d' B' Us U
    by (rule split-range-costed-partition-loop-state-trace
      [OF run sound pre P-reaches])
  have children:
    list-all2 (λU X. U = X ∧ complete-on d' U) Us
      (range-tree-chain-list P a bs B')
    using trace unfolding concrete-partition-loop-trace-def by blast
  have U-def: U = bound-tree P (Fin a) ∪ ⋃(set Us)
    using trace unfolding concrete-partition-loop-trace-def by blast
  have Union-eq:
    ⋃(set Us) = ⋃(set (range-tree-chain-list P a bs B'))
    using children by (induction rule: list-all2-induct) auto
  have U-eq-chain:
    U = bound-tree P (Fin a) ∪ range-tree-chain P a bs B'
    using U-def Union-eq Union-range-tree-chain-list[of P a bs B'] by simp
  have finite-U: finite U
    using U-eq-chain by simp
  have chain-subset: range-tree-chain P a bs B' ⊆ U
    using U-eq-chain by blast
  have card-chain-le: card (range-tree-chain P a bs B') ≤ card U
    by (rule card-mono[OF finite-U chain-subset])
  have outgoing-subset:
    outgoing-edges (range-tree-chain P a bs B') ⊆ outgoing-edges U
    by (rule outgoing-edges-mono[OF chain-subset])
  have finite-out-U: finite (outgoing-edges U)
    by simp
  have card-out-le:
    card (outgoing-edges (range-tree-chain P a bs B')) ≤
      card (outgoing-edges U)
    by (rule card-mono[OF finite-out-U outgoing-subset])
  have loop:
    c ≤
      A * L * card (range-tree-chain P a bs B') +
```

```
    (R + t) * card (outgoing-edges (range-tree-chain P a bs B')) +
    Suc h * card (range-tree-chain P a bs B')
   by (rule split-range-costed-partition-loop-state-cost-from-child-and-source-bounds
    [OF run child-cost-bounds source-progress])
  have source-term:
   Suc h * card (range-tree-chain P a bs B') ≤ A * card U
  proof −
   have Suc h * card (range-tree-chain P a bs B') ≤
     A * card (range-tree-chain P a bs B')
    using source-factor by (rule mult-right-mono) simp
   also have ... ≤ A * card U
    using card-chain-le by simp
   finally show ?thesis .
  qed
  have c ≤
    A * L * card U + (R + t) * card (outgoing-edges U) + A * card U
  proof −
   have vterm:
    A * L * card (range-tree-chain P a bs B') ≤ A * L * card U
    using card-chain-le by simp
   have eterm:
    (R + t) * card (outgoing-edges (range-tree-chain P a bs B')) ≤
     (R + t) * card (outgoing-edges U)
    using card-out-le by simp
   have A * L * card (range-tree-chain P a bs B') +
     (R + t) * card (outgoing-edges (range-tree-chain P a bs B')) +
     Suc h * card (range-tree-chain P a bs B') ≤
     A * L * card U + (R + t) * card (outgoing-edges U) + A * card U
    using vterm eterm source-term by linarith
   then show ?thesis
    using loop by linarith
  qed
  also have ... =
    A * Suc L * card U + (R + t) * card (outgoing-edges U)
   by (simp add: algebra-simps)
  finally show ?thesis .
qed

theorem split-range-costed-partition-loop-state-closes-level-bound-from-source-
progress-general:
  assumes run:
   split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
    betas bs B' Us U c child-costs
   and sound: sound-label d
   and pre: bmssp-pre-full d P B
   and P-reaches: ⋀x. x ∈ P ⟹ reachable s x
   and child-bound:
    ⋀c-child U-child S-child B-child d-child B-child'.
     ⟦split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
```

```
      d-child B-child' U-child c-child;
     bmssp-pre-full d S-child B-child;
     ⋀x. x ∈ S-child ⟹ reachable s x;
     card S-child ≤ M-of l⟧
    ⟹ c-child ≤ level-range-cost-bound A R L U-child
 and source-progress:
  ⋀S-child B-child d-child B-child' U-child c-child.
   ⟦split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
      d-child B-child' U-child c-child;
     bmssp-pre-full d S-child B-child;
     ⋀x. x ∈ S-child ⟹ reachable s x;
     card S-child ≤ M-of l⟧
    ⟹ card S-child ≤ card U-child
 and source-factor: Suc h ≤ A
shows c ≤ A * Suc L * card U + (R + t) * card (outgoing-edges U)
proof −
 have child-cost-bounds:
  list-all2 (λc-child U-child.
   c-child ≤ level-range-cost-bound A R L U-child)
   child-costs (range-tree-child-list P a bs)
  by (rule split-range-costed-partition-loop-state-child-cost-bounds
   [OF run child-bound])
 show ?thesis
  by (rule
   split-range-costed-partition-loop-state-closes-level-bound-from-child-cost-bounds-
general
   [OF run sound pre P-reaches child-cost-bounds source-progress
    source-factor])
qed

theorem split-range-costed-nonbase-step-closes-level-bound-from-source-progress-
general:
 assumes loop:
  split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
   betas bs B' Us U-loop c-loop child-costs
 and sound: sound-label d
 and pre: bmssp-pre-full d P B
 and P-reaches: ⋀x. x ∈ P ⟹ reachable s x
 and child-bound:
  ⋀c-child U-child S-child B-child d-child B-child'.
   ⟦split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
      d-child B-child' U-child c-child;
     bmssp-pre-full d S-child B-child;
     ⋀x. x ∈ S-child ⟹ reachable s x;
     card S-child ≤ M-of l⟧
    ⟹ c-child ≤ level-range-cost-bound A R L U-child
 and source-progress:
  ⋀S-child B-child d-child B-child' U-child c-child.
   ⟦split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
```

```
        d-child B-child' U-child c-child;
       bmssp-pre-full d S-child B-child;
       ⋀x. x ∈ S-child ⟹ reachable s x;
       card S-child ≤ M-of l⟧
      ⟹ card S-child ≤ card U-child
  and source-factor: Suc h ≤ A
  and U-def: U = U-loop ∪ W
  and finite-U: finite U
  and scan-insert: c-scan-insert ≤ A * card U
  and c-def: c = c-scan-insert + c-loop
 shows c ≤ A * Suc (Suc L) * card U +
  (R + t) * card (outgoing-edges U)
proof −
 have loop-bound:
  c-loop ≤ A * Suc L * card U-loop +
   (R + t) * card (outgoing-edges U-loop)
  by (rule split-range-costed-partition-loop-state-closes-level-bound-from-source-
progress-general
   [OF loop sound pre P-reaches child-bound source-progress source-factor])
 have U-loop-subset: U-loop ⊆ U
  using U-def by blast
 have card-loop-le: card U-loop ≤ card U
  by (rule card-mono[OF finite-U U-loop-subset])
 have outgoing-subset: outgoing-edges U-loop ⊆ outgoing-edges U
  by (rule outgoing-edges-mono[OF U-loop-subset])
 have finite-out-U: finite (outgoing-edges U)
  by simp
 have card-out-le: card (outgoing-edges U-loop) ≤ card (outgoing-edges U)
  by (rule card-mono[OF finite-out-U outgoing-subset])
 have c ≤
   A * card U +
   (A * Suc L * card U + (R + t) * card (outgoing-edges U))
 proof −
  have loop-to-U:
   c-loop ≤ A * Suc L * card U +
    (R + t) * card (outgoing-edges U)
  proof −
   have A * Suc L * card U-loop ≤ A * Suc L * card U
    using card-loop-le by simp
   moreover have (R + t) * card (outgoing-edges U-loop) ≤
     (R + t) * card (outgoing-edges U)
    using card-out-le by simp
   ultimately show ?thesis
    using loop-bound by linarith
  qed
  show ?thesis
   using scan-insert loop-to-U c-def by linarith
 qed
 also have ... =
```

```
    A * Suc (Suc L) * card U + (R + t) * card (outgoing-edges U)
   by (simp add: algebra-simps)
  finally show ?thesis .
qed

lemma split-range-costed-base-level-bound:
  assumes R-pos: 0 < R
  shows base-case-scan-cost Δ k x B ≤
    level-range-cost-bound A R (Suc 0) (base-case-vertices k x B)
proof −
  have base-case-scan-cost Δ k x B ≤
      R * card (outgoing-edges (base-case-vertices k x B))
    using R-pos unfolding base-case-scan-cost-def by simp
  then show ?thesis
    unfolding level-range-cost-bound-def by linarith
qed

theorem split-range-costed-bmssp-level-bound-from-source-progress-and-local-budgets:
  assumes base-budget: Δ ≤ A
    and R-pos: 0 < R
    and source-factor: Suc h ≤ A
    and source-progress:
      ⋀l d S B d' B' U c.
        split-range-costed-bmssp Δ M-of t h k cap l d S B d' B' U c ⟹
        bmssp-pre-full d S B ⟹
        (⋀x. x ∈ S ⟹ reachable s x) ⟹
        card S ≤ M-of l ⟹
        card S ≤ card U
    and step-budget:
      ⋀l d S B D c-insert d' a betas bs B' Us U-loop c-loop
        child-costs U.
        D = label-partition-view
          (find-pivots-label-capped k cap d S B)
          (find-pivots-pivots-capped k cap d S B) ⟹
        partition-initial-insert-cost-bound c-insert t
          (find-pivots-pivots-capped k cap d S B) ⟹
        split-range-costed-partition-loop-state Δ M-of t h k cap l
          (find-pivots-label-capped k cap d S B)
          (find-pivots-pivots-capped k cap d S B) B d' D a betas bs B'
          Us U-loop c-loop child-costs ⟹
        complete-on d'
          {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v} ⟹
        U = U-loop ∪
          {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v} ⟹
        sound-label d ⟹
        bmssp-pre-full d S B ⟹
        (⋀x. x ∈ S ⟹ reachable s x) ⟹
        fp-iter-capped-scan-cost k cap d S S B + c-insert ≤ A * card U
```

```
    and run:
      split-range-costed-bmssp Δ M-of t h k cap l d S B d′ B′ U c
    and sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
  shows c ≤ level-range-cost-bound A (R + l * t) (2 * l + 1) U
  using run sound pre S-reaches
proof (induction l arbitrary: d S B d′ B′ U c rule: less-induct)
  case (less l)
  show ?case
  proof (cases l)
    case 0
    have run0:
      split-range-costed-bmssp Δ M-of t h k cap 0 d S B d′ B′ U c
      using less.prems(1) 0 by simp
    show ?thesis
      using run0 R-pos 0
      by (cases rule: split-range-costed-bmssp.cases)
        (simp-all add: split-range-costed-base-level-bound)
  next
    case (Suc l0)
    have run-suc:
      split-range-costed-bmssp Δ M-of t h k cap (Suc l0) d S B d′ B′ U c
      using less.prems(1) Suc by simp
    show ?thesis
    proof (cases rule: split-range-costed-bmssp-SucE[OF run-suc])
      case (1 c-insert a betas bs Us U-loop c-loop child-costs-loop)
      let ?d-fp = find-pivots-label-capped k cap d S B
      let ?P = find-pivots-pivots-capped k cap d S B
      let ?W = {v ∈ bound-tree S B′. ?d-fp v = dist s v}
      have sound-fp: sound-label ?d-fp
        unfolding find-pivots-label-capped-def
        by (rule fp-iter-capped-label-sound[OF less.prems(2) less.prems(4)])
      have pivot-pre: bmssp-pre-full ?d-fp ?P B
        using find-pivots-capped-establishes-pivot-pre-concrete
          [OF less.prems(2) less.prems(3) less.prems(4)] .
      have P-subset-S: ?P ⊆ S
        unfolding find-pivots-pivots-capped-def by auto
      have P-reaches: ⋀x. x ∈ ?P ⟹ reachable s x
        using P-subset-S less.prems(4) by blast
      have child-bound:
        ⋀c-child U-child S-child B-child d-child B-child′.
          ⟦split-range-costed-bmssp Δ M-of t h k cap l0 ?d-fp S-child B-child
              d-child B-child′ U-child c-child;
            bmssp-pre-full ?d-fp S-child B-child;
            ⋀x. x ∈ S-child ⟹ reachable s x;
            card S-child ≤ M-of l0⟧
          ⟹ c-child ≤ level-range-cost-bound A (R + l0 * t)
            (2 * l0 + 1) U-child
```

proof −
fix *c-child U-child S-child B-child d-child B-child'*
assume *child*:
split-range-costed-bmssp Δ *M-of t h k cap l0 ?d-fp S-child B-child d-child B-child' U-child c-child*
and *child-pre*: *bmssp-pre-full ?d-fp S-child B-child*
and *child-reaches*: ⋀*x*. *x* ∈ *S-child* ⟹ *reachable s x*
show *c-child* ≤ *level-range-cost-bound A* (*R* + *l0* ∗ *t*) (*2* ∗ *l0* + *1*) *U-child*
by (*rule less.IH*[*of l0 ?d-fp S-child B-child d-child B-child' U-child c-child*, *OF - child sound-fp child-pre child-reaches*]) (*simp add*: *Suc*)
qed
have *source-progress-loop*:
⋀*S-child B-child d-child B-child' U-child c-child*.
⟦*split-range-costed-bmssp* Δ *M-of t h k cap l0 ?d-fp S-child B-child d-child B-child' U-child c-child*;
bmssp-pre-full ?d-fp S-child B-child;
⋀*x*. *x* ∈ *S-child* ⟹ *reachable s x*;
card S-child ≤ *M-of l0*⟧
⟹ *card S-child* ≤ *card U-child*
by (*rule source-progress*)
have *scan-insert*:
fp-iter-capped-scan-cost k cap d S S B + *c-insert* ≤ *A* ∗ *card U*
by (*rule step-budget*[*OF refl 1*(*3*) *1*(*4*) *1*(*5*) *1*(*1*) *less.prems*(*2*) *less.prems*(*3*) *less.prems*(*4*)])
have *trace*:
concrete-partition-loop-trace ?P B a bs d' B' Us U-loop
by (*rule split-range-costed-partition-loop-state-trace* [*OF 1*(*4*) *sound-fp pivot-pre P-reaches*])
have *loop-post*: *bmssp-post-full ?d-fp ?P B d' B' U-loop*
by (*rule concrete-partition-loop-trace-post*[*OF trace*])
have *finite-U-loop*: *finite U-loop*
using *loop-post* **unfolding** *bmssp-post-full-def* **by** *simp*
have *finite-W*: *finite ?W*
by *simp*
have *finite-U*: *finite U*
using *1*(*1*) *finite-U-loop finite-W* **by** *simp*
have *step*:
c ≤ *A* ∗ *Suc* (*Suc* (*2* ∗ *l0* + *1*)) ∗ *card U* +
(*R* + *l0* ∗ *t* + *t*) ∗ *card* (*outgoing-edges U*)
by (*rule split-range-costed-nonbase-step-closes-level-bound-from-source-progress-general*
[*OF 1*(*4*) *sound-fp pivot-pre P-reaches child-bound source-progress-loop source-factor 1*(*1*) *finite-U scan-insert 1*(*2*)])
show *?thesis*
using *step Suc* **unfolding** *level-range-cost-bound-def*
by (*simp add*: *algebra-simps*)

qed
qed
qed

theorem *split-range-costed-bmssp-level-bound-from-source-progress-budgets*:
assumes *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *source-factor*: *Suc* $h \leq 2 * A$
and *source-progress*:
$\bigwedge l\ d\ S\ B\ d'\ B'\ U\ c.$
split-range-costed-bmssp Δ *M-of t h k cap l d S B d' B' U c* $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
card S $\leq$ *M-of l* $\Longrightarrow$
card S $\leq$ *card U*
and *step-progress*:
$\bigwedge l\ d\ S\ B\ D$ *c-insert d' a betas bs B' Us U-loop c-loop*
child-costs U.
D = *label-partition-view*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
split-range-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B'*
Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d'
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* = *dist s v*$\}$ $\Longrightarrow$
U = *U-loop* $\cup$
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* = *dist s v*$\}$ $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
card S $\leq$ *cap* $\wedge$ $k * cap \leq$ *card U*
and *run*:
split-range-costed-bmssp Δ *M-of t h k cap l d S B d' B' U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
shows *c* $\leq$ *level-range-cost-bound* $(2 * A)$ $(R + l * t)$ $(2 * l + 1)$ *U*
proof (*rule split-range-costed-bmssp-level-bound-from-source-progress-and-local-budgets*
[**where** $A = 2 * A$ **and** $R = R$,
OF - R-pos source-factor source-progress - run sound pre S-reaches])
show $\Delta \leq 2 * A$
using *degree-factor* **by** *simp*

next
fix *l d S B D c-insert d′ a betas bs B′ Us U-loop c-loop child-costs U*
assume *D-def*:
D = label-partition-view
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*)
and *insert*:
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*)
and *loop*:
split-range-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d′ D a betas bs B′*
Us U-loop c-loop child-costs
and *complete*:
complete-on d′
{*v* ∈ *bound-tree S B′. find-pivots-label-capped k cap d S B v = dist s v*}
and *U-def*:
U = U-loop ∪
{*v* ∈ *bound-tree S B′. find-pivots-label-capped k cap d S B v = dist s v*}
and *sound-s*: *sound-label d*
and *pre-s*: *bmssp-pre-full d S B*
and *reaches-s*: ⋀*x. x* ∈ *S* ⟹ *reachable s x*
have *S-subset*: *S* ⊆ *V*
using *pre-s* **unfolding** *bmssp-pre-full-def* **by** *blast*
have *progress*:
card S ≤ *cap* ∧ *k* ∗ *cap* ≤ *card U*
by (*rule step-progress*[*OF D-def insert loop complete U-def sound-s pre-s reaches-s*])
then have *S-cap*: *card S* ≤ *cap*
by *blast*
from *progress* **have** *U-progress*: *k* ∗ *cap* ≤ *card U*
by *blast*
show *fp-iter-capped-scan-cost k cap d S S B* + *c-insert* ≤
2 ∗ *A* ∗ *card U*
by (*rule find-pivots-scan-and-initial-insert-budget-from-progress*
[*OF S-subset degree S-cap insert degree-factor insert-factor U-progress*])
qed

theorem *finite-initial-label-split-range-costed-top-level-correct*:
assumes *all-reachable*: ⋀*v. v* ∈ *V* ⟹ *reachable s v*
and *run*:
split-range-costed-bmssp Δ *M-of t h k cap l finite-initial-label* {*s*}
Infinity d′ Infinity U c
shows *sssp-correct d′*
proof −
have *pre*: *bmssp-pre-full finite-initial-label* {*s*} *Infinity*
using *all-reachable finite-initial-label-source-complete*
by (*rule top-bmssp-pre-full*)

have *sound*: *sound-label finite-initial-label*
using *finite-initial-label-sound*[*OF all-reachable*] .
have *S-reaches*: $\bigwedge x.\ x \in \{s\} \Longrightarrow$ *reachable s x*
using *all-reachable source-in-V* **by** *blast*
have *post-full*:
bmssp-post-full finite-initial-label $\{s\}$ *Infinity d' Infinity U*
by (*rule split-range-costed-bmssp-correct*[*OF run sound pre S-reaches*])
have *post*: *bmssp-post finite-initial-label* $\{s\}$ *Infinity d' Infinity U*
using *bmssp-post-full-imp-post*[*OF post-full*] .
then have *U-V*: $U = V$
using *bound-tree-source-infinity*[*OF all-reachable*]
unfolding *bmssp-post-def* **by** *auto*
have *complete*: *complete-on d' U*
using *post* **unfolding** *bmssp-post-def* **by** *auto*
then show *?thesis*
using *U-V* **unfolding** *complete-on-def sssp-correct-def* **by** *auto*
qed

theorem *finite-initial-label-split-range-costed-top-level-correct-and-local-budget-bound*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *base-budget*: $\Delta \le A$
and *R-pos*: $0 < R$
and *source-factor*: *Suc h* $\le A$
and *source-progress*:
$\bigwedge l\ d\ S\ B\ d'\ B'\ U\ c.$
split-range-costed-bmssp Δ *M-of t h k cap l d S B d' B' U c* $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
card S $\le$ *M-of l* $\Longrightarrow$
card S $\le$ *card U*
and *step-budget*:
$\bigwedge l\ d\ S\ B\ D$ *c-insert d' a betas bs B' Us U-loop c-loop*
child-costs U.
$D =$ *label-partition-view*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
split-range-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B'*
Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d'
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* $=$ *dist s v*$\} \Longrightarrow$
$U = U\text{-}loop \cup$
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* $=$ *dist s v*$\} \Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$

```
      (⋀x. x ∈ S ⟹ reachable s x) ⟹
      fp-iter-capped-scan-cost k cap d S S B + c-insert ≤ A * card U
    and run:
     split-range-costed-bmssp Δ M-of t h k cap l finite-initial-label {s}
      Infinity d' Infinity U c
  shows sssp-correct d' ∧
    c ≤ level-range-cost-bound A (R + l * t) (2 * l + 1) U
proof
  show sssp-correct d'
    by (rule finite-initial-label-split-range-costed-top-level-correct
      [OF all-reachable run])
next
  have pre: bmssp-pre-full finite-initial-label {s} Infinity
    using all-reachable finite-initial-label-source-complete
    by (rule top-bmssp-pre-full)
  have sound: sound-label finite-initial-label
    using finite-initial-label-sound[OF all-reachable] .
  have S-reaches: ⋀x. x ∈ {s} ⟹ reachable s x
    using all-reachable source-in-V by blast
  show c ≤ level-range-cost-bound A (R + l * t) (2 * l + 1) U
    by (rule split-range-costed-bmssp-level-bound-from-source-progress-and-local-
budgets
      [OF base-budget R-pos source-factor source-progress step-budget run sound pre
        S-reaches])
qed

theorem finite-initial-label-split-range-costed-top-level-correct-and-local-budget-
graph-bound:
  assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
    and base-budget: Δ ≤ A
    and R-pos: 0 < R
    and source-factor: Suc h ≤ A
    and source-progress:
      ⋀l d S B d' B' U c.
        split-range-costed-bmssp Δ M-of t h k cap l d S B d' B' U c ⟹
        bmssp-pre-full d S B ⟹
        (⋀x. x ∈ S ⟹ reachable s x) ⟹
        card S ≤ M-of l ⟹
        card S ≤ card U
    and step-budget:
      ⋀l d S B D c-insert d' a betas bs B' Us U-loop c-loop
        child-costs U.
        D = label-partition-view
          (find-pivots-label-capped k cap d S B)
          (find-pivots-pivots-capped k cap d S B) ⟹
        partition-initial-insert-cost-bound c-insert t
          (find-pivots-pivots-capped k cap d S B) ⟹
        split-range-costed-partition-loop-state Δ M-of t h k cap l
          (find-pivots-label-capped k cap d S B)
```

```
      (find-pivots-pivots-capped k cap d S B) B d' D a betas bs B'
       Us U-loop c-loop child-costs ⟹
      complete-on d'
       {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v} ⟹
      U = U-loop ∪
       {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v} ⟹
      sound-label d ⟹
      bmssp-pre-full d S B ⟹
      (⋀x. x ∈ S ⟹ reachable s x) ⟹
      fp-iter-capped-scan-cost k cap d S S B + c-insert ≤ A * card U
  and run:
    split-range-costed-bmssp Δ M-of t h k cap l finite-initial-label {s}
     Infinity d' Infinity U c
 shows sssp-correct d' ∧
  c ≤ A * (2 * l + 1) * vertex-count + (R + l * t) * edge-count
proof −
 have combined:
  sssp-correct d' ∧
   c ≤ level-range-cost-bound A (R + l * t) (2 * l + 1) U
  by (rule finite-initial-label-split-range-costed-top-level-correct-and-local-budget-bound
    [OF all-reachable base-budget R-pos source-factor source-progress step-budget run])
 then have correct: sssp-correct d'
  by blast
 have pre: bmssp-pre-full finite-initial-label {s} Infinity
  using all-reachable finite-initial-label-source-complete
  by (rule top-bmssp-pre-full)
 have sound: sound-label finite-initial-label
  using finite-initial-label-sound[OF all-reachable] .
 have S-reaches: ⋀x. x ∈ {s} ⟹ reachable s x
  using all-reachable source-in-V by blast
 have post-full:
  bmssp-post-full finite-initial-label {s} Infinity d' Infinity U
  by (rule split-range-costed-bmssp-correct[OF run sound pre S-reaches])
 have post: bmssp-post finite-initial-label {s} Infinity d' Infinity U
  using bmssp-post-full-imp-post[OF post-full] .
 have U-V: U = V
  using post bound-tree-source-infinity[OF all-reachable]
  unfolding bmssp-post-def by auto
 have cost-U:
    c ≤ level-range-cost-bound A (R + l * t) (2 * l + 1) U
  using combined by blast
 have graph-bound:
  level-range-cost-bound A (R + l * t) (2 * l + 1) U
   ≤ A * (2 * l + 1) * vertex-count + (R + l * t) * edge-count
 proof −
  have U-card: card U = vertex-count
   unfolding U-V vertex-count-def by simp
```

have *out-le*: *card* (*outgoing-edges U*) ≤ *edge-count*
by (*rule edge-count-outgoing-bound*)
have *out-term*:
(*R* + *l* ∗ *t*) ∗ *card* (*outgoing-edges U*) ≤ (*R* + *l* ∗ *t*) ∗ *edge-count*
using *out-le* **by** *simp*
show *?thesis*
unfolding *level-range-cost-bound-def*
using *U-card out-term* **by** (*simp add*: *algebra-simps*)
qed
show *?thesis*
using *correct cost-U graph-bound* **by** *linarith*
qed

theorem *finite-initial-label-split-range-costed-top-level-correct-and-source-progress-bound*:
assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: Δ ≤ *A*
and *R-pos*: *0* < *R*
and *insert-factor*: *t* ≤ *A* ∗ *k*
and *source-factor*: *Suc h* ≤ *2* ∗ *A*
and *source-progress*:
⋀*l d S B d*′ *B*′ *U c*.
split-range-costed-bmssp Δ *M-of t h k cap l d S B d*′ *B*′ *U c* ⟹
bmssp-pre-full d S B ⟹
(⋀*x*. *x* ∈ *S* ⟹ *reachable s x*) ⟹
card S ≤ *M-of l* ⟹
card S ≤ *card U*
and *step-progress*:
⋀*l d S B D c-insert d*′ *a betas bs B*′ *Us U-loop c-loop*
child-costs U.
D = *label-partition-view*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) ⟹
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*) ⟹
split-range-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d*′ *D a betas bs B*′
Us U-loop c-loop child-costs ⟹
complete-on d′
{*v* ∈ *bound-tree S B*′. *find-pivots-label-capped k cap d S B v* = *dist s v*} ⟹
U = *U-loop* ∪
{*v* ∈ *bound-tree S B*′. *find-pivots-label-capped k cap d S B v* = *dist s v*} ⟹
sound-label d ⟹
bmssp-pre-full d S B ⟹
(⋀*x*. *x* ∈ *S* ⟹ *reachable s x*) ⟹
card S ≤ *cap* ∧ *k* ∗ *cap* ≤ *card U*
and *run*:

```
    split-range-costed-bmssp Δ M-of t h k cap l finite-initial-label {s}
      Infinity d' Infinity U c
  shows sssp-correct d' ∧
    c ≤ level-range-cost-bound (2 * A) (R + l * t) (2 * l + 1) U
proof
  show sssp-correct d'
    by (rule finite-initial-label-split-range-costed-top-level-correct
      [OF all-reachable run])
next
  have pre: bmssp-pre-full finite-initial-label {s} Infinity
    using all-reachable finite-initial-label-source-complete
    by (rule top-bmssp-pre-full)
  have sound: sound-label finite-initial-label
    using finite-initial-label-sound[OF all-reachable] .
  have S-reaches: ⋀x. x ∈ {s} ⟹ reachable s x
    using all-reachable source-in-V by blast
  show c ≤ level-range-cost-bound (2 * A) (R + l * t) (2 * l + 1) U
    by (rule split-range-costed-bmssp-level-bound-from-source-progress-budgets
      [OF degree degree-factor R-pos insert-factor source-factor source-progress
        step-progress run sound pre S-reaches])
qed

theorem finite-initial-label-split-range-costed-top-level-correct-and-graph-bound:
  assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
    and degree: edge-outdegree-le Δ
    and degree-factor: Δ ≤ A
    and R-pos: 0 < R
    and insert-factor: t ≤ A * k
    and source-factor: Suc h ≤ 2 * A
    and source-progress:
      ⋀l d S B d' B' U c.
        split-range-costed-bmssp Δ M-of t h k cap l d S B d' B' U c ⟹
        bmssp-pre-full d S B ⟹
        (⋀x. x ∈ S ⟹ reachable s x) ⟹
        card S ≤ M-of l ⟹
        card S ≤ card U
    and step-progress:
      ⋀l d S B D c-insert d' a betas bs B' Us U-loop c-loop
        child-costs U.
        D = label-partition-view
          (find-pivots-label-capped k cap d S B)
          (find-pivots-pivots-capped k cap d S B) ⟹
        partition-initial-insert-cost-bound c-insert t
          (find-pivots-pivots-capped k cap d S B) ⟹
        split-range-costed-partition-loop-state Δ M-of t h k cap l
          (find-pivots-label-capped k cap d S B)
          (find-pivots-pivots-capped k cap d S B) B d' D a betas bs B'
          Us U-loop c-loop child-costs ⟹
        complete-on d'
```

$\{v \in$ *bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v*$\} \Longrightarrow$
U = U-loop $\cup$
$\{v \in$ *bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v*$\} \Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
card S $\leq$ *cap* $\wedge$ *k* $*$ *cap* $\leq$ *card U*
and *run*:
split-range-costed-bmssp Δ *M-of t h k cap l finite-initial-label* $\{s\}$
Infinity d' Infinity U c
shows *sssp-correct d'* $\wedge$
c $\leq$ *2* $*$ *A* $*$ *(2* $*$ *l* $+$ *1)* $*$ *vertex-count* $+$ *(R* $+$ *l* $*$ *t)* $*$ *edge-count*
proof $-$
have *combined*:
sssp-correct d' $\wedge$
c $\leq$ *level-range-cost-bound (2* $*$ *A) (R* $+$ *l* $*$ *t) (2* $*$ *l* $+$ *1) U*
by (*rule finite-initial-label-split-range-costed-top-level-correct-and-source-progress-bound*
[*OF all-reachable degree degree-factor R-pos insert-factor source-factor source-progress step-progress run*])
then have *correct*: *sssp-correct d'*
by *blast*
have *pre*: *bmssp-pre-full finite-initial-label* $\{s\}$ *Infinity*
using *all-reachable finite-initial-label-source-complete*
by (*rule top-bmssp-pre-full*)
have *sound*: *sound-label finite-initial-label*
using *finite-initial-label-sound*[*OF all-reachable*] **.**
have *S-reaches*: $\bigwedge x.\ x \in \{s\} \Longrightarrow$ *reachable s x*
using *all-reachable source-in-V* **by** *blast*
have *post-full*:
bmssp-post-full finite-initial-label $\{s\}$ *Infinity d' Infinity U*
by (*rule split-range-costed-bmssp-correct*[*OF run sound pre S-reaches*])
have *post*: *bmssp-post finite-initial-label* $\{s\}$ *Infinity d' Infinity U*
using *bmssp-post-full-imp-post*[*OF post-full*] **.**
have *U-V*: *U* $=$ *V*
using *post bound-tree-source-infinity*[*OF all-reachable*]
unfolding *bmssp-post-def* **by** *auto*
have *cost-U*:
c $\leq$ *level-range-cost-bound (2* $*$ *A) (R* $+$ *l* $*$ *t) (2* $*$ *l* $+$ *1) U*
using *combined* **by** *blast*
have *graph-bound*:
level-range-cost-bound (2 $*$ *A) (R* $+$ *l* $*$ *t) (2* $*$ *l* $+$ *1) U*
$\leq$ *2* $*$ *A* $*$ *(2* $*$ *l* $+$ *1)* $*$ *vertex-count* $+$ *(R* $+$ *l* $*$ *t)* $*$ *edge-count*
proof $-$
have *U-card*: *card U* $=$ *vertex-count*
unfolding *U-V vertex-count-def* **by** *simp*
have *out-le*: *card (outgoing-edges U)* $\leq$ *edge-count*
by (*rule edge-count-outgoing-bound*)
have *out-term*:

```
    (R + l * t) * card (outgoing-edges U) ≤ (R + l * t) * edge-count
    using out-le by simp
  show ?thesis
    unfolding level-range-cost-bound-def
    using U-card out-term by (simp add: algebra-simps)
 qed
 show ?thesis
  using correct cost-U graph-bound by linarith
qed

theorem split-range-costed-bmssp-source-card-le-if-sound-label-below-output:
 assumes run:
    split-range-costed-bmssp Δ M-of t h k cap l d S B d' B' U c
  and sound: sound-label d
  and pre: bmssp-pre-full d S B
  and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
  and below: ⋀x. x ∈ S ⟹ below-bound (d x) B'
 shows card S ≤ card U
proof −
 have post: bmssp-post-full d S B d' B' U
  by (rule split-range-costed-bmssp-correct[OF run sound pre S-reaches])
 have U-eq: U = bound-tree S B'
  using post unfolding bmssp-post-full-def by blast
 have S-subset-U: S ⊆ U
 proof
  fix x
  assume xS: x ∈ S
  have S-subset: S ⊆ V
   using pre unfolding bmssp-pre-full-def by blast
  have xV: x ∈ V
   using S-subset xS by blast
  have reach-x: reachable s x
   by (rule S-reaches[OF xS])
  have dist-le: dist s x ≤ d x
   using sound xV reach-x unfolding sound-label-def by blast
  have below-dist: below-bound (dist s x) B'
   using below-bound-le-trans[OF dist-le below[OF xS]] .
  show x ∈ U
   unfolding U-eq
   by (rule source-in-own-bound-tree[OF xS reach-x below-dist])
 qed
 have finite-U: finite U
  unfolding U-eq bound-tree-def using finite-V by auto
 show ?thesis
  by (rule card-mono[OF finite-U S-subset-U])
qed

theorem split-range-costed-partition-loop-state-success-or-threshold:
 split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
```

betas bs B′ Us U c child-costs ⟹
sound-label d ⟹
bmssp-pre-full d P B ⟹
(⋀*x*. *x* ∈ *P* ⟹ *reachable s x*) ⟹
B′ = *B* ∨ *k* ∗ *cap* ≤ *card U*
and *split-range-costed-bmssp-success-or-threshold-trivial*:
split-range-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c* ⟹ *True*
proof (*induction rule*: *split-range-costed-partition-loop-state-split-range-costed-bmssp.inducts*)
case (*Split-Range-State-Done D a B d′ P* Δ *M-of t h k cap l d*)
then show *?case*
by *simp*
next
case (*Split-Range-State-Stop a B d′ P k cap* Δ *M-of t h l d D*)
then show *?case*
by *simp*
next
case (*Split-Range-State-Step D M-of l Bmax S-pull beta D-pull B d a b B-out d-out P k cap Delta t h d-child U-child c-child edge-batch source-batch batch D-next c-pull c-edges c-sources betas bs Us-tail U-tail c-tail child-costs-tail c*)
have *tail*:
B-out = *B* ∨ *k* ∗ *cap* ≤ *card U-tail*
using *Split-Range-State-Step.IH Split-Range-State-Step.prems* **by** *blast*
then show *?case*
proof
assume *B-out* = *B*
then show *?thesis*
by *blast*
next
assume *threshold-tail*: *k* ∗ *cap* ≤ *card U-tail*
have *tail-trace*:
concrete-partition-loop-trace P B b bs d-out B-out Us-tail U-tail
by (*rule split-range-costed-partition-loop-state-trace*)
(*use Split-Range-State-Step* **in** *blast*)+
have *U-tail-def*:
U-tail = *bound-tree P* (*Fin b*) ∪ ⋃(*set Us-tail*)
using *tail-trace* **unfolding** *concrete-partition-loop-trace-def* **by** *blast*
have *split-b*:
bound-tree P (*Fin b*) =
bound-tree P (*Fin a*) ∪ *range-tree P a* (*Fin b*)
using *bound-tree-split-at*[*of a Fin b P*] *Split-Range-State-Step*
by *simp*
have *U-eq-tail*:
bound-tree P (*Fin a*) ∪
⋃(*set* (*range-tree P a* (*Fin b*) # *Us-tail*)) =
U-tail
using *U-tail-def split-b* **by** *auto*
show *?thesis*

```
    using threshold-tail U-eq-tail by simp
  qed
next
  case (Split-Range-Cost-Base S x Δ M-of t h k cap d B)
  then show ?case
    by simp
next
  case (Split-Range-Cost-Step D k cap d S B c-insert t Δ M-of h l d' a
      betas bs B' Us U-loop c-loop child-costs-loop U c)
  then show ?case
    by simp
qed

theorem split-range-costed-bmssp-Suc-success-or-threshold:
  assumes run:
      split-range-costed-bmssp Δ M-of t h k cap (Suc l) d S B d' B' U c
    and sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
  shows B' = B ∨ k * cap ≤ card U
  using run
proof cases
  case (Split-Range-Cost-Step D c-insert a betas bs Us U-loop c-loop child-costs-loop)
  let ?d-fp = find-pivots-label-capped k cap d S B
  let ?P = find-pivots-pivots-capped k cap d S B
  let ?W = {v ∈ bound-tree S B'. ?d-fp v = dist s v}
  have sound-fp: sound-label ?d-fp
    unfolding find-pivots-label-capped-def
    by (rule fp-iter-capped-label-sound[OF sound S-reaches])
  have pivot-pre: bmssp-pre-full ?d-fp ?P B
    using find-pivots-capped-establishes-pivot-pre-concrete
      [OF sound pre S-reaches] .
  have P-subset-S: ?P ⊆ S
    unfolding find-pivots-pivots-capped-def by auto
  have P-reaches: ⋀x. x ∈ ?P ⟹ reachable s x
    using P-subset-S S-reaches by blast
  have loop-class:
    B' = B ∨ k * cap ≤ card U-loop
    by (rule split-range-costed-partition-loop-state-success-or-threshold
      [OF Split-Range-Cost-Step(3) sound-fp pivot-pre P-reaches])
  have loop-trace:
    concrete-partition-loop-trace ?P B a bs d' B' Us U-loop
    by (rule split-range-costed-partition-loop-state-trace
      [OF Split-Range-Cost-Step(3) sound-fp pivot-pre P-reaches])
  have loop-post: bmssp-post-full ?d-fp ?P B d' B' U-loop
    by (rule concrete-partition-loop-trace-post[OF loop-trace])
  have finite-U-loop: finite U-loop
    using loop-post unfolding bmssp-post-full-def by simp
  show ?thesis
```

```
    using loop-class
  proof
    assume B' = B
    then show ?thesis
      by blast
  next
    assume threshold-loop: k * cap ≤ card U-loop
    have U-eq: U = U-loop ∪ ?W
      using Split-Range-Cost-Step(5) by simp
    have finite-U: finite U
      unfolding U-eq using finite-U-loop by simp
    have U-loop-subset: U-loop ⊆ U
      unfolding U-eq by blast
    have card U-loop ≤ card U
      by (rule card-mono[OF finite-U U-loop-subset])
    then show ?thesis
      using threshold-loop by linarith
  qed
qed

theorem split-range-costed-bmssp-Suc-source-card-le-from-label-below:
  assumes run:
      split-range-costed-bmssp Δ M-of t h k cap (Suc l) d S B d' B' U c
    and sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and below: ⋀x. x ∈ S ⟹ below-bound (d x) B
    and S-cap: card S ≤ cap
    and k-pos: 0 < k
  shows card S ≤ card U
proof −
  have call-class: B' = B ∨ k * cap ≤ card U
    by (rule split-range-costed-bmssp-Suc-success-or-threshold
      [OF run sound pre S-reaches])
  then show ?thesis
  proof
    assume B'-eq: B' = B
    show ?thesis
    proof (rule split-range-costed-bmssp-source-card-le-if-sound-label-below-output
        [OF run sound pre S-reaches])
      fix x
      assume x ∈ S
      then show below-bound (d x) B'
        using below B'-eq by simp
    qed
  next
    assume threshold: k * cap ≤ card U
    have cap ≤ k * cap
      using k-pos by (cases k) simp-all
```

```
    then show ?thesis
      using S-cap threshold by linarith
  qed
qed

theorem split-range-costed-capped-step-threshold-if-not-success:
  assumes loop:
      split-range-costed-partition-loop-state Δ M-of t h k cap l
        (find-pivots-label-capped k cap d S B)
        (find-pivots-pivots-capped k cap d S B) B d' D a betas bs B'
        Us U-loop c-loop child-costs
    and U-def:
      U = U-loop ∪
        {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v}
    and sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and not-success: B' ≠ B
  shows k * cap ≤ card U
proof −
  let ?d-fp = find-pivots-label-capped k cap d S B
  let ?P = find-pivots-pivots-capped k cap d S B
  have sound-fp: sound-label ?d-fp
    unfolding find-pivots-label-capped-def
    by (rule fp-iter-capped-label-sound[OF sound S-reaches])
  have pivot-pre: bmssp-pre-full ?d-fp ?P B
    using find-pivots-capped-establishes-pivot-pre-concrete
      [OF sound pre S-reaches] .
  have P-subset-S: ?P ⊆ S
    unfolding find-pivots-pivots-capped-def by auto
  have P-reaches: ⋀x. x ∈ ?P ⟹ reachable s x
    using P-subset-S S-reaches by blast
  have loop-class:
    B' = B ∨ k * cap ≤ card U-loop
    by (rule split-range-costed-partition-loop-state-success-or-threshold
      [OF loop sound-fp pivot-pre P-reaches])
  have threshold-loop: k * cap ≤ card U-loop
    using loop-class not-success by blast
  have trace: concrete-partition-loop-trace ?P B a bs d' B' Us U-loop
    by (rule split-range-costed-partition-loop-state-trace
      [OF loop sound-fp pivot-pre P-reaches])
  have loop-post: bmssp-post-full ?d-fp ?P B d' B' U-loop
    by (rule concrete-partition-loop-trace-post[OF trace])
  have finite-U-loop: finite U-loop
    using loop-post unfolding bmssp-post-full-def by simp
  have finite-U: finite U
    unfolding U-def using finite-U-loop by simp
  have U-loop-subset: U-loop ⊆ U
    unfolding U-def by blast
```

```
  have card U-loop ≤ card U
    by (rule card-mono[OF finite-U U-loop-subset])
  then show ?thesis
    using threshold-loop by linarith
qed

theorem split-range-costed-capped-step-scan-insert-budget-if-not-success:
  assumes degree: edge-outdegree-le Δ
    and degree-factor: Δ ≤ A
    and insert-factor: t ≤ A * k
    and insert:
      partition-initial-insert-cost-bound c-insert t
        (find-pivots-pivots-capped k cap d S B)
    and loop:
      split-range-costed-partition-loop-state Δ M-of t h k cap l
        (find-pivots-label-capped k cap d S B)
        (find-pivots-pivots-capped k cap d S B) B d' D a betas bs B'
        Us U-loop c-loop child-costs
    and U-def:
      U = U-loop ∪
        {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v}
    and sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and S-cap: card S ≤ cap
    and not-success: B' ≠ B
  shows fp-iter-capped-scan-cost k cap d S S B + c-insert ≤
    2 * A * card U
proof −
  have S-subset: S ⊆ V
    using pre unfolding bmssp-pre-full-def by blast
  have progress: k * cap ≤ card U
    by (rule split-range-costed-capped-step-threshold-if-not-success
      [OF loop U-def sound pre S-reaches not-success])
  show ?thesis
    by (rule find-pivots-scan-and-initial-insert-budget-from-progress
      [OF S-subset degree S-cap insert degree-factor insert-factor progress])
qed

theorem split-range-costed-capped-step-scan-insert-budget-from-seen-or-threshold:
  assumes degree: edge-outdegree-le Δ
    and degree-factor: Δ ≤ A
    and insert-factor: t ≤ A * k
    and seen-factor: k * Δ + t ≤ 2 * A
    and insert:
      partition-initial-insert-cost-bound c-insert t
        (find-pivots-pivots-capped k cap d S B)
    and loop:
      split-range-costed-partition-loop-state Δ M-of t h k cap l
```

```
      (find-pivots-label-capped k cap d S B)
      (find-pivots-pivots-capped k cap d S B) B d' D a betas bs B'
      Us U-loop c-loop child-costs
   and U-def:
    U = U-loop ∪
      {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v}
   and sound: sound-label d
   and pre: bmssp-pre-full d S B
   and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
   and S-cap: card S ≤ cap
   and anti: tree-antichain S
   and k-pos: 0 < k
   and seen-success:
    B' = B ⟹
     card (find-pivots-seen-capped k cap d S B) ≤ card U
 shows fp-iter-capped-scan-cost k cap d S S B + c-insert ≤
  2 * A * card U
proof (cases B' = B)
 case True
 have S-subset: S ⊆ V
  using pre unfolding bmssp-pre-full-def by blast
 show ?thesis
  by (rule find-pivots-scan-and-initial-insert-budget-from-seen-total
   [OF S-subset degree S-cap anti k-pos insert seen-factor
    seen-success[OF True]])
next
 case False
 show ?thesis
  by (rule split-range-costed-capped-step-scan-insert-budget-if-not-success
   [OF degree degree-factor insert-factor insert loop U-def sound pre
    S-reaches S-cap False])
qed

theorem split-range-costed-capped-step-scan-insert-budget-from-scaled-seen-or-
threshold:
 assumes degree: edge-outdegree-le Δ
   and degree-factor: Δ ≤ A
   and insert-factor: t ≤ A * k
   and insert-scaled-factor: t ≤ A-insert * k
   and seen-scaled-factor: k * Δ + A-insert ≤ 2 * A
   and insert:
    partition-initial-insert-cost-bound c-insert t
      (find-pivots-pivots-capped k cap d S B)
   and loop:
    split-range-costed-partition-loop-state Δ M-of t h k cap l
      (find-pivots-label-capped k cap d S B)
      (find-pivots-pivots-capped k cap d S B) B d' D a betas bs B'
      Us U-loop c-loop child-costs
   and U-def:
```

```
      U = U-loop ∪
        {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v}
    and sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and S-k-cap: k * card S ≤ cap
    and anti: tree-antichain S
    and k-pos: 0 < k
    and seen-success:
      B' = B ⟹
       card (find-pivots-seen-capped k cap d S B) ≤ card U
  shows fp-iter-capped-scan-cost k cap d S S B + c-insert ≤
    2 * A * card U
proof (cases B' = B)
  case True
  have S-subset: S ⊆ V
    using pre unfolding bmssp-pre-full-def by blast
  show ?thesis
    by (rule find-pivots-scan-and-initial-insert-budget-from-seen-scaled
      [OF S-subset degree anti S-k-cap insert insert-scaled-factor
        seen-scaled-factor seen-success[OF True]])
next
  case False
  have S-cap: card S ≤ cap
  proof −
    have card S ≤ k * card S
      using k-pos by (cases k) simp-all
    then show ?thesis
      using S-k-cap by linarith
  qed
  show ?thesis
    by (rule split-range-costed-capped-step-scan-insert-budget-if-not-success
      [OF degree degree-factor insert-factor insert loop U-def sound pre
        S-reaches S-cap False])
qed

theorem split-range-costed-bmssp-level-bound-from-seen-progress-budgets:
  assumes degree: edge-outdegree-le Δ
    and degree-factor: Δ ≤ A
    and R-pos: 0 < R
    and insert-factor: t ≤ A * k
    and seen-factor: k * Δ + t ≤ 2 * A
    and source-factor: Suc h ≤ 2 * A
    and k-pos: 0 < k
    and source-progress:
      ⋀l d S B d' B' U c.
        split-range-costed-bmssp Δ M-of t h k cap l d S B d' B' U c ⟹
        bmssp-pre-full d S B ⟹
        (⋀x. x ∈ S ⟹ reachable s x) ⟹
```

card S ≤ M-of l ⟹
card S ≤ card U
and *step-seen-progress*:
⋀*l d S B D c-insert d′ a betas bs B′ Us U-loop c-loop*
child-costs U.
D = label-partition-view
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) ⟹
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*) ⟹
split-range-costed-partition-loop-state Δ M-of t h k cap l
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d′ D a betas bs B′*
Us U-loop c-loop child-costs ⟹
complete-on d′
{*v* ∈ *bound-tree S B′*. *find-pivots-label-capped k cap d S B v = dist s v*} ⟹
U = U-loop ∪
{*v* ∈ *bound-tree S B′*. *find-pivots-label-capped k cap d S B v = dist s v*} ⟹
sound-label d ⟹
bmssp-pre-full d S B ⟹
(⋀*x*. *x* ∈ *S* ⟹ *reachable s x*) ⟹
card S ≤ cap ∧ *tree-antichain S* ∧
(*B′ = B* ⟶
card (*find-pivots-seen-capped k cap d S B*) ≤ *card U*)
and *run*:
split-range-costed-bmssp Δ M-of t h k cap l d S B d′ B′ U c
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: ⋀*x*. *x* ∈ *S* ⟹ *reachable s x*
shows *c ≤ level-range-cost-bound* (*2 * A*) (*R + l * t*) (*2 * l + 1*) *U*
proof (*rule split-range-costed-bmssp-level-bound-from-source-progress-and-local-budgets*
[**where** *A = 2 * A* **and** *R = R*,
OF - R-pos source-factor source-progress - run sound pre S-reaches])
show *Δ ≤ 2 * A*
using *degree-factor* **by** *simp*
next
fix *l d S B D c-insert d′ a betas bs B′ Us U-loop c-loop child-costs U*
assume *D-def*:
D = label-partition-view
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*)
and *insert*:
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*)
and *loop*:
split-range-costed-partition-loop-state Δ M-of t h k cap l
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d′ D a betas bs B′*

```
      Us U-loop c-loop child-costs
    and complete:
      complete-on d'
        {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v}
    and U-def:
      U = U-loop ∪
        {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v}
    and sound-s: sound-label d
    and pre-s: bmssp-pre-full d S B
    and reaches-s: ⋀x. x ∈ S ⟹ reachable s x
  have progress:
      card S ≤ cap ∧ tree-antichain S ∧
        (B' = B ⟶
          card (find-pivots-seen-capped k cap d S B) ≤ card U)
    by (rule step-seen-progress
      [OF D-def insert loop complete U-def sound-s pre-s reaches-s])
  then have S-cap: card S ≤ cap
    by blast
  from progress have anti: tree-antichain S
    by blast
  from progress have seen-success:
      B' = B ⟹
        card (find-pivots-seen-capped k cap d S B) ≤ card U
    by blast
  show fp-iter-capped-scan-cost k cap d S S B + c-insert ≤
      2 * A * card U
    by (rule split-range-costed-capped-step-scan-insert-budget-from-seen-or-threshold
      [OF degree degree-factor insert-factor seen-factor insert loop U-def sound-s
        pre-s reaches-s S-cap anti k-pos seen-success])
qed

theorem split-range-costed-bmssp-level-bound-from-scaled-seen-progress-budgets:
  assumes degree: edge-outdegree-le Δ
    and degree-factor: Δ ≤ A
    and R-pos: 0 < R
    and insert-factor: t ≤ A * k
    and insert-scaled-factor: t ≤ A-insert * k
    and seen-scaled-factor: k * Δ + A-insert ≤ 2 * A
    and source-factor: Suc h ≤ 2 * A
    and k-pos: 0 < k
    and source-progress:
      ⋀l d S B d' B' U c.
        split-range-costed-bmssp Δ M-of t h k cap l d S B d' B' U c ⟹
        bmssp-pre-full d S B ⟹
        (⋀x. x ∈ S ⟹ reachable s x) ⟹
        card S ≤ M-of l ⟹
        card S ≤ card U
    and step-scaled-seen-progress:
      ⋀l d S B D c-insert d' a betas bs B' Us U-loop c-loop
```

```
      child-costs U.
    D = label-partition-view
      (find-pivots-label-capped k cap d S B)
      (find-pivots-pivots-capped k cap d S B) ⟹
    partition-initial-insert-cost-bound c-insert t
      (find-pivots-pivots-capped k cap d S B) ⟹
    split-range-costed-partition-loop-state Δ M-of t h k cap l
      (find-pivots-label-capped k cap d S B)
      (find-pivots-pivots-capped k cap d S B) B d' D a betas bs B'
      Us U-loop c-loop child-costs ⟹
    complete-on d'
      {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v} ⟹
    U = U-loop ∪
      {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v} ⟹
    sound-label d ⟹
    bmssp-pre-full d S B ⟹
    (⋀x. x ∈ S ⟹ reachable s x) ⟹
    k * card S ≤ cap ∧ tree-antichain S ∧
    (B' = B ⟶
      card (find-pivots-seen-capped k cap d S B) ≤ card U)
  and run:
    split-range-costed-bmssp Δ M-of t h k cap l d S B d' B' U c
  and sound: sound-label d
  and pre: bmssp-pre-full d S B
  and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
 shows c ≤ level-range-cost-bound (2 * A) (R + l * t) (2 * l + 1) U
proof (rule split-range-costed-bmssp-level-bound-from-source-progress-and-local-
budgets
  [where A = 2 * A and R = R,
    OF - R-pos source-factor source-progress - run sound pre S-reaches])
 show Δ ≤ 2 * A
  using degree-factor by simp
next
 fix l d S B D c-insert d' a betas bs B' Us U-loop c-loop child-costs U
 assume D-def:
    D = label-partition-view
      (find-pivots-label-capped k cap d S B)
      (find-pivots-pivots-capped k cap d S B)
  and insert:
    partition-initial-insert-cost-bound c-insert t
      (find-pivots-pivots-capped k cap d S B)
  and loop:
    split-range-costed-partition-loop-state Δ M-of t h k cap l
      (find-pivots-label-capped k cap d S B)
      (find-pivots-pivots-capped k cap d S B) B d' D a betas bs B'
      Us U-loop c-loop child-costs
  and complete:
    complete-on d'
      {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v}
```

and *U-def*:
U = *U-loop* ∪
{*v* ∈ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* = *dist s v*}
and *sound-s*: *sound-label d*
and *pre-s*: *bmssp-pre-full d S B*
and *reaches-s*: ⋀*x*. *x* ∈ *S* ⟹ *reachable s x*
have *progress*:
k ∗ *card S* ≤ *cap* ∧ *tree-antichain S* ∧
(*B'* = *B* ⟶
card (*find-pivots-seen-capped k cap d S B*) ≤ *card U*)
by (*rule step-scaled-seen-progress*
[*OF D-def insert loop complete U-def sound-s pre-s reaches-s*])
then have *S-k-cap*: *k* ∗ *card S* ≤ *cap*
by *blast*
from *progress* **have** *anti*: *tree-antichain S*
by *blast*
from *progress* **have** *seen-success*:
B' = *B* ⟹
card (*find-pivots-seen-capped k cap d S B*) ≤ *card U*
by *blast*
show *fp-iter-capped-scan-cost k cap d S S B* + *c-insert* ≤
2 ∗ *A* ∗ *card U*
by (*rule split-range-costed-capped-step-scan-insert-budget-from-scaled-seen-or-threshold*
[*OF degree degree-factor insert-factor insert-scaled-factor*
seen-scaled-factor insert loop U-def sound-s pre-s reaches-s
S-k-cap anti k-pos seen-success])
qed

theorem *finite-initial-label-split-range-costed-top-level-correct-and-seen-progress-bound*:
assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: Δ ≤ *A*
and *R-pos*: *0* < *R*
and *insert-factor*: *t* ≤ *A* ∗ *k*
and *seen-factor*: *k* ∗ Δ + *t* ≤ *2* ∗ *A*
and *source-factor*: *Suc h* ≤ *2* ∗ *A*
and *k-pos*: *0* < *k*
and *source-progress*:
⋀*l d S B d' B' U c*.
split-range-costed-bmssp Δ *M-of t h k cap l d S B d' B' U c* ⟹
bmssp-pre-full d S B ⟹
(⋀*x*. *x* ∈ *S* ⟹ *reachable s x*) ⟹
card S ≤ *M-of l* ⟹
card S ≤ *card U*
and *step-seen-progress*:
⋀*l d S B D c-insert d' a betas bs B' Us U-loop c-loop*
child-costs U.

$D = \textit{label-partition-view}$
($\textit{find-pivots-label-capped}\ k\ \textit{cap}\ d\ S\ B$)
($\textit{find-pivots-pivots-capped}\ k\ \textit{cap}\ d\ S\ B$) $\Longrightarrow$
$\textit{partition-initial-insert-cost-bound}\ \textit{c-insert}\ t$
($\textit{find-pivots-pivots-capped}\ k\ \textit{cap}\ d\ S\ B$) $\Longrightarrow$
$\textit{split-range-costed-partition-loop-state}\ \Delta\ \textit{M-of}\ t\ h\ k\ \textit{cap}\ l$
($\textit{find-pivots-label-capped}\ k\ \textit{cap}\ d\ S\ B$)
($\textit{find-pivots-pivots-capped}\ k\ \textit{cap}\ d\ S\ B$) $B\ d'\ D\ a\ \textit{betas}\ bs\ B'$
$\textit{Us}\ \textit{U-loop}\ \textit{c-loop}\ \textit{child-costs} \Longrightarrow$
$\textit{complete-on}\ d'$
$\{v \in \textit{bound-tree}\ S\ B'.\ \textit{find-pivots-label-capped}\ k\ \textit{cap}\ d\ S\ B\ v = \textit{dist}\ s\ v\} \Longrightarrow$
$U = \textit{U-loop} \cup$
$\{v \in \textit{bound-tree}\ S\ B'.\ \textit{find-pivots-label-capped}\ k\ \textit{cap}\ d\ S\ B\ v = \textit{dist}\ s\ v\} \Longrightarrow$
$\textit{sound-label}\ d \Longrightarrow$
$\textit{bmssp-pre-full}\ d\ S\ B \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow \textit{reachable}\ s\ x) \Longrightarrow$
$\textit{card}\ S \leq \textit{cap} \wedge \textit{tree-antichain}\ S \wedge$
$(B' = B \longrightarrow$
$\textit{card}\ (\textit{find-pivots-seen-capped}\ k\ \textit{cap}\ d\ S\ B) \leq \textit{card}\ U)$
and *run*:
$\textit{split-range-costed-bmssp}\ \Delta\ \textit{M-of}\ t\ h\ k\ \textit{cap}\ l\ \textit{finite-initial-label}\ \{s\}$
$\textit{Infinity}\ d'\ \textit{Infinity}\ U\ c$
shows $\textit{sssp-correct}\ d' \wedge$
$c \leq \textit{level-range-cost-bound}\ (2 * A)\ (R + l * t)\ (2 * l + 1)\ U$
proof
show $\textit{sssp-correct}\ d'$
by (*rule finite-initial-label-split-range-costed-top-level-correct*
[*OF all-reachable run*])
next
have *pre*: $\textit{bmssp-pre-full}\ \textit{finite-initial-label}\ \{s\}\ \textit{Infinity}$
using *all-reachable finite-initial-label-source-complete*
by (*rule top-bmssp-pre-full*)
have *sound*: $\textit{sound-label}\ \textit{finite-initial-label}$
using *finite-initial-label-sound*[*OF all-reachable*] **.**
have *S-reaches*: $\bigwedge x.\ x \in \{s\} \Longrightarrow \textit{reachable}\ s\ x$
using *all-reachable source-in-V* **by** *blast*
show $c \leq \textit{level-range-cost-bound}\ (2 * A)\ (R + l * t)\ (2 * l + 1)\ U$
by (*rule split-range-costed-bmssp-level-bound-from-seen-progress-budgets*
[*OF degree degree-factor R-pos insert-factor seen-factor source-factor k-pos*
source-progress step-seen-progress run sound pre S-reaches])
qed

theorem *finite-initial-label-split-range-costed-top-level-correct-and-scaled-seen-progress-bound*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow \textit{reachable}\ s\ v$
and *degree*: $\textit{edge-outdegree-le}\ \Delta$
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$

```
    and insert-scaled-factor: t ≤ A-insert * k
    and seen-scaled-factor: k * Δ + A-insert ≤ 2 * A
    and source-factor: Suc h ≤ 2 * A
    and k-pos: 0 < k
    and source-progress:
      ⋀l d S B d' B' U c.
        split-range-costed-bmssp Δ M-of t h k cap l d S B d' B' U c ⟹
        bmssp-pre-full d S B ⟹
        (⋀x. x ∈ S ⟹ reachable s x) ⟹
        card S ≤ M-of l ⟹
        card S ≤ card U
    and step-scaled-seen-progress:
      ⋀l d S B D c-insert d' a betas bs B' Us U-loop c-loop
          child-costs U.
        D = label-partition-view
          (find-pivots-label-capped k cap d S B)
          (find-pivots-pivots-capped k cap d S B) ⟹
        partition-initial-insert-cost-bound c-insert t
          (find-pivots-pivots-capped k cap d S B) ⟹
        split-range-costed-partition-loop-state Δ M-of t h k cap l
          (find-pivots-label-capped k cap d S B)
          (find-pivots-pivots-capped k cap d S B) B d' D a betas bs B'
          Us U-loop c-loop child-costs ⟹
        complete-on d'
          {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v} ⟹
        U = U-loop ∪
          {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v} ⟹
        sound-label d ⟹
        bmssp-pre-full d S B ⟹
        (⋀x. x ∈ S ⟹ reachable s x) ⟹
        k * card S ≤ cap ∧ tree-antichain S ∧
        (B' = B ⟶
          card (find-pivots-seen-capped k cap d S B) ≤ card U)
    and run:
      split-range-costed-bmssp Δ M-of t h k cap l finite-initial-label {s}
        Infinity d' Infinity U c
  shows sssp-correct d' ∧
    c ≤ level-range-cost-bound (2 * A) (R + l * t) (2 * l + 1) U
proof
  show sssp-correct d'
    by (rule finite-initial-label-split-range-costed-top-level-correct
      [OF all-reachable run])
next
  have pre: bmssp-pre-full finite-initial-label {s} Infinity
    using all-reachable finite-initial-label-source-complete
    by (rule top-bmssp-pre-full)
  have sound: sound-label finite-initial-label
    using finite-initial-label-sound[OF all-reachable] .
  have S-reaches: ⋀x. x ∈ {s} ⟹ reachable s x
```

using *all-reachable source-in-V* **by** *blast*
show $c \leq$ *level-range-cost-bound* $(2 * A)$ $(R + l * t)$ $(2 * l + 1)$ U
by (*rule split-range-costed-bmssp-level-bound-from-scaled-seen-progress-budgets*
[*OF degree degree-factor R-pos insert-factor insert-scaled-factor*
seen-scaled-factor source-factor k-pos source-progress
step-scaled-seen-progress run sound pre S-reaches])
qed

theorem *finite-initial-label-split-range-costed-top-level-correct-and-seen-progress-graph-bound*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *seen-factor*: $k * \Delta + t \leq 2 * A$
and *source-factor*: *Suc* $h \leq 2 * A$
and *k-pos*: $0 < k$
and *source-progress*:
$\bigwedge l\ d\ S\ B\ d'\ B'\ U\ c.$
split-range-costed-bmssp Δ *M-of t h k cap l d S B d' B' U c* $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
card S $\leq$ *M-of l* $\Longrightarrow$
card S $\leq$ *card U*
and *step-seen-progress*:
$\bigwedge l\ d\ S\ B\ D$ *c-insert d' a betas bs B' Us U-loop c-loop*
child-costs U.
$D =$ *label-partition-view*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
split-range-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B'*
Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d'
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* $=$ *dist s v*$\}$ $\Longrightarrow$
$U =$ *U-loop* $\cup$
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* $=$ *dist s v*$\}$ $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
card S $\leq$ *cap* $\wedge$ *tree-antichain S* $\wedge$
$(B' = B \longrightarrow$
card (*find-pivots-seen-capped k cap d S B*) $\leq$ *card U*$)$
and *run*:
split-range-costed-bmssp Δ *M-of t h k cap l finite-initial-label* $\{s\}$

```
      Infinity d' Infinity U c
  shows sssp-correct d' ∧
    c ≤ 2 * A * (2 * l + 1) * vertex-count + (R + l * t) * edge-count
proof −
  have combined:
    sssp-correct d' ∧
      c ≤ level-range-cost-bound (2 * A) (R + l * t) (2 * l + 1) U
    by (rule finite-initial-label-split-range-costed-top-level-correct-and-seen-progress-
bound
      [OF all-reachable degree degree-factor R-pos insert-factor seen-factor
        source-factor k-pos source-progress step-seen-progress run])
  then have correct: sssp-correct d'
    by blast
  have pre: bmssp-pre-full finite-initial-label {s} Infinity
    using all-reachable finite-initial-label-source-complete
    by (rule top-bmssp-pre-full)
  have sound: sound-label finite-initial-label
    using finite-initial-label-sound[OF all-reachable] .
  have S-reaches: ⋀x. x ∈ {s} ⟹ reachable s x
    using all-reachable source-in-V by blast
  have post-full:
    bmssp-post-full finite-initial-label {s} Infinity d' Infinity U
    by (rule split-range-costed-bmssp-correct[OF run sound pre S-reaches])
  have post: bmssp-post finite-initial-label {s} Infinity d' Infinity U
    using bmssp-post-full-imp-post[OF post-full] .
  have U-V: U = V
    using post bound-tree-source-infinity[OF all-reachable]
    unfolding bmssp-post-def by auto
  have cost-U:
      c ≤ level-range-cost-bound (2 * A) (R + l * t) (2 * l + 1) U
    using combined by blast
  have graph-bound:
    level-range-cost-bound (2 * A) (R + l * t) (2 * l + 1) U
      ≤ 2 * A * (2 * l + 1) * vertex-count + (R + l * t) * edge-count
  proof −
    have U-card: card U = vertex-count
      unfolding U-V vertex-count-def by simp
    have out-le: card (outgoing-edges U) ≤ edge-count
      by (rule edge-count-outgoing-bound)
    have out-term:
      (R + l * t) * card (outgoing-edges U) ≤ (R + l * t) * edge-count
      using out-le by simp
    show ?thesis
      unfolding level-range-cost-bound-def
      using U-card out-term by (simp add: algebra-simps)
  qed
  show ?thesis
    using correct cost-U graph-bound by linarith
qed
```

theorem *finite-initial-label-split-range-costed-top-level-correct-and-scaled-seen-progress-graph-bound*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *insert-scaled-factor*: $t \leq$ *A-insert* $* k$
and *seen-scaled-factor*: $k * \Delta +$ *A-insert* $\leq 2 * A$
and *source-factor*: *Suc h* $\leq 2 * A$
and *k-pos*: $0 < k$
and *source-progress*:
$\bigwedge l\ d\ S\ B\ d'\ B'\ U\ c.$
split-range-costed-bmssp Δ *M-of t h k cap l d S B d' B' U c* $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
card S $\leq$ *M-of l* $\Longrightarrow$
card S $\leq$ *card U*
and *step-scaled-seen-progress*:
$\bigwedge l\ d\ S\ B\ D$ *c-insert d' a betas bs B' Us U-loop c-loop*
child-costs U.
$D =$ *label-partition-view*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
split-range-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B'*
Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d'
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* $=$ *dist s v*$\} \Longrightarrow$
$U =$ *U-loop* $\cup$
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* $=$ *dist s v*$\} \Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
$k *$ *card S* $\leq$ *cap* $\wedge$ *tree-antichain S* $\wedge$
$(B' = B \longrightarrow$
card (*find-pivots-seen-capped k cap d S B*) $\leq$ *card U*$)$
and *run*:
split-range-costed-bmssp Δ *M-of t h k cap l finite-initial-label* $\{s\}$
Infinity d' Infinity U c
shows *sssp-correct d'* $\wedge$
$c \leq 2 * A * (2 * l + 1) *$ *vertex-count* $+ (R + l * t) *$ *edge-count*
proof $-$
have *combined*:
sssp-correct d' $\wedge$

```
    c ≤ level-range-cost-bound (2 * A) (R + l * t) (2 * l + 1) U
  by (rule finite-initial-label-split-range-costed-top-level-correct-and-scaled-seen-progress-bound
      [OF all-reachable degree degree-factor R-pos insert-factor insert-scaled-factor
        seen-scaled-factor source-factor k-pos source-progress
        step-scaled-seen-progress run])
 then have correct: sssp-correct d'
  by blast
 have pre: bmssp-pre-full finite-initial-label {s} Infinity
  using all-reachable finite-initial-label-source-complete
  by (rule top-bmssp-pre-full)
 have sound: sound-label finite-initial-label
  using finite-initial-label-sound[OF all-reachable] .
 have S-reaches: ⋀x. x ∈ {s} ⟹ reachable s x
  using all-reachable source-in-V by blast
 have post-full:
  bmssp-post-full finite-initial-label {s} Infinity d' Infinity U
  by (rule split-range-costed-bmssp-correct[OF run sound pre S-reaches])
 have post: bmssp-post finite-initial-label {s} Infinity d' Infinity U
  using bmssp-post-full-imp-post[OF post-full] .
 have U-V: U = V
  using post bound-tree-source-infinity[OF all-reachable]
  unfolding bmssp-post-def by auto
 have cost-U:
    c ≤ level-range-cost-bound (2 * A) (R + l * t) (2 * l + 1) U
  using combined by blast
 have graph-bound:
  level-range-cost-bound (2 * A) (R + l * t) (2 * l + 1) U
    ≤ 2 * A * (2 * l + 1) * vertex-count + (R + l * t) * edge-count
 proof −
  have U-card: card U = vertex-count
   unfolding U-V vertex-count-def by simp
  have out-le: card (outgoing-edges U) ≤ edge-count
   by (rule edge-count-outgoing-bound)
  have out-term:
   (R + l * t) * card (outgoing-edges U) ≤ (R + l * t) * edge-count
   using out-le by simp
  show ?thesis
   unfolding level-range-cost-bound-def
   using U-card out-term by (simp add: algebra-simps)
 qed
 show ?thesis
  using correct cost-U graph-bound by linarith
qed

theorem range-costed-partition-loop-state-cost-bound:
 assumes run:
  range-costed-partition-loop-state Δ M-of t k cap l d P B d' D a
   betas bs B' Us U c child-costs
```

shows $c \leq$ *sum-list child-costs* + (*M-of l*) $*$ *length child-costs* +
t $*$ (*sum-list*
(*map* (λ*X*. *card* (*outgoing-edges X*)) (*range-tree-child-list P a bs*)) +
(*M-of l*) $*$ *length child-costs*)
using *run*
by (*induction*)
(*auto simp*: *pull-separates-def partition-pull-cost-bound-def*
partition-batch-cost-bound-def algebra-simps
dest: *range-costed-pull-cost-le-M range-costed-batch-cost-le*)

theorem *range-costed-partition-loop-state-cost-bound-by-child-sources*:
range-costed-partition-loop-state Δ *M-of t k cap l d P B d′ D a*
betas bs B′ Us U c child-costs ⟹
∃ *child-sources*.
length child-sources = *length bs* ∧
list-all2
(λ*S-child U-child*. ∃ *c-child B-child d-child B-child′*.
range-costed-bmssp Δ *M-of t k cap l d S-child B-child*
d-child B-child′ U-child c-child ∧
bmssp-pre-full d S-child B-child ∧
(∀ *x*∈*S-child*. *reachable s x*) ∧
card S-child ≤ *M-of l*)
child-sources (*range-tree-child-list P a bs*) ∧
c ≤ *sum-list child-costs* + *sum-list* (*map card child-sources*) +
t $*$ (*sum-list*
(*map* (λ*X*. *card* (*outgoing-edges X*)) (*range-tree-child-list P a bs*)) +
sum-list (*map card child-sources*))
and *range-costed-bmssp-child-sources-trivial*:
range-costed-bmssp Δ *M-of t k cap l d S B d′ B′ U c* ⟹ *True*
proof (*induction rule*:
range-costed-partition-loop-state-range-costed-bmssp.inducts)
case (*Range-State-Done D a B d′ P* Δ *M-of t k cap l d*)
show *?case*
by (*intro exI*[*of* - []]) *simp*
next
case (*Range-State-Stop a B d′ P k cap* Δ *M-of t l d D*)
show *?case*
by (*intro exI*[*of* - []]) *simp*
next
case (*Range-State-Step D M-of l Bmax S-pull beta D-pull B d a b B′*
d′ P k cap Δ *t d-child U-child c-child batch D-next c-pull*
c-batch betas bs Us-tail U-tail c-tail child-costs-tail c)
obtain *child-sources-tail* **where** *len-tail*:
length child-sources-tail = *length bs*
and *sources-tail*:
list-all2
(λ*S-child U-child*. ∃ *c-child B-child d-child B-child′*.
range-costed-bmssp Δ *M-of t k cap l d S-child B-child*
d-child B-child′ U-child c-child ∧

```
      bmssp-pre-full d S-child B-child ∧
      (∀ x∈S-child. reachable s x) ∧
      card S-child ≤ M-of l)
    child-sources-tail (range-tree-child-list P b bs)
 and tail-cost:
  c-tail ≤
    sum-list child-costs-tail + sum-list (map card child-sources-tail) +
    t * (sum-list
      (map (λX. card (outgoing-edges X)) (range-tree-child-list P b bs)) +
      sum-list (map card child-sources-tail))
 using Range-State-Step.IH(11) by blast
have card-pull: card S-pull ≤ M-of l
 using Range-State-Step unfolding pull-separates-def by blast
have pull-cost: c-pull ≤ card S-pull
 using Range-State-Step
 unfolding partition-pull-cost-bound-def by blast
have batch-cost:
 c-batch ≤ t * (card (outgoing-edges U-child) + card S-pull)
 by (rule range-costed-batch-cost-le
  [where d = d and S = S-pull and d-child = d-child and b = b
   and beta = beta and batch = batch and M = card S-pull])
  (use Range-State-Step in auto)
let ?child-sources = S-pull # child-sources-tail
have len: length ?child-sources = length (b # bs)
 using len-tail by simp
have head-source:
 ∃ c-child' B-child d-child' B-child'.
  range-costed-bmssp Δ M-of t k cap l d S-pull B-child
   d-child' B-child' U-child c-child' ∧
  bmssp-pre-full d S-pull B-child ∧
  (∀ x∈S-pull. reachable s x) ∧
  card S-pull ≤ M-of l
 using Range-State-Step card-pull by blast
have sources:
 list-all2
  (λS-child U-child. ∃ c-child B-child d-child B-child'.
   range-costed-bmssp Δ M-of t k cap l d S-child B-child
    d-child B-child' U-child c-child ∧
   bmssp-pre-full d S-child B-child ∧
   (∀ x∈S-child. reachable s x) ∧
   card S-child ≤ M-of l)
  ?child-sources (range-tree-child-list P a (b # bs))
 using head-source sources-tail Range-State-Step by simp
have cost:
 c ≤ sum-list (c-child # child-costs-tail) +
  sum-list (map card ?child-sources) +
  t * (sum-list
   (map (λX. card (outgoing-edges X))
    (range-tree-child-list P a (b # bs))) +
```

```
      sum-list (map card ?child-sources))
  proof −
    have c-eq: c = c-pull + c-batch + c-child + c-tail
      using Range-State-Step by blast
    have c-pull + c-batch + c-child + c-tail ≤
        card S-pull + t * (card (outgoing-edges U-child) + card S-pull) +
        c-child +
        (sum-list child-costs-tail + sum-list (map card child-sources-tail) +
          t * (sum-list
            (map (λX. card (outgoing-edges X)) (range-tree-child-list P b bs)) +
            sum-list (map card child-sources-tail)))
      using pull-cost batch-cost tail-cost by simp
    also have ... =
        sum-list (c-child # child-costs-tail) +
        sum-list (map card ?child-sources) +
        t * (card (outgoing-edges U-child) +
          sum-list
            (map (λX. card (outgoing-edges X)) (range-tree-child-list P b bs)) +
          sum-list (map card ?child-sources))
      by (simp add: algebra-simps)
    also have ... =
        sum-list (c-child # child-costs-tail) +
        sum-list (map card ?child-sources) +
        t * (sum-list
          (map (λX. card (outgoing-edges X))
            (range-tree-child-list P a (b # bs))) +
          sum-list (map card ?child-sources))
      using Range-State-Step by simp
    finally show ?thesis
      using c-eq by simp
  qed
  show ?case
    by (intro exI[of - ?child-sources] conjI)
      (use len sources cost in simp-all)
next
  case (Range-Cost-Base S x Δ M-of t k cap d B)
  show ?case
    by simp
next
  case (Range-Cost-Step D k cap d S B c-insert t Δ M-of l d′ a betas bs B′
      Us U-loop c-loop child-costs-loop U c)
  show ?case
    by simp
qed

theorem range-costed-partition-loop-state-cost-from-child-and-source-bounds:
  assumes run:
    range-costed-partition-loop-state Δ M-of t k cap l d P B d′ D a
      betas bs B′ Us U c child-costs
```

```
    and child-cost-bounds:
      list-all2 (λc-child U-child.
        c-child ≤ level-range-cost-bound A R L U-child)
        child-costs (range-tree-child-list P a bs)
    and source-progress:
      ⋀S-child B-child d-child B-child' U-child c-child.
        ⟦range-costed-bmssp Δ M-of t k cap l d S-child B-child
            d-child B-child' U-child c-child;
          bmssp-pre-full d S-child B-child;
          ⋀x. x ∈ S-child ⟹ reachable s x;
          card S-child ≤ M-of l⟧
        ⟹ card S-child ≤ card U-child
  shows c ≤
    A * L * card (range-tree-chain P a bs B') +
    (R + t) * card (outgoing-edges (range-tree-chain P a bs B')) +
    Suc t * card (range-tree-chain P a bs B')
proof −
  obtain child-sources where len-sources:
      length child-sources = length bs
    and sources:
      list-all2
        (λS-child U-child. ∃ c-child B-child d-child B-child'.
          range-costed-bmssp Δ M-of t k cap l d S-child B-child
            d-child B-child' U-child c-child ∧
          bmssp-pre-full d S-child B-child ∧
          (∀ x∈S-child. reachable s x) ∧
          card S-child ≤ M-of l)
        child-sources (range-tree-child-list P a bs)
    and cost:
      c ≤ sum-list child-costs + sum-list (map card child-sources) +
        t * (sum-list
          (map (λX. card (outgoing-edges X)) (range-tree-child-list P a bs)) +
          sum-list (map card child-sources))
    using range-costed-partition-loop-state-cost-bound-by-child-sources[OF run]
    by blast
  have mono: nondecreasing-from a bs
    by (rule range-costed-partition-loop-state-mono[OF run])
  have source-dom:
    list-all2 (λS-child U-child. card S-child ≤ card U-child)
      child-sources (range-tree-child-list P a bs)
    using sources
  proof (induction rule: list-all2-induct)
    case Nil
    then show ?case by simp
  next
    case (Cons S-child child-sources U-child child-sets)
    obtain c-child B-child d-child B-child' where child:
        range-costed-bmssp Δ M-of t k cap l d S-child B-child
          d-child B-child' U-child c-child
```

```
    and child-pre: bmssp-pre-full d S-child B-child
    and child-reaches: ∀ x∈S-child. reachable s x
    and child-card: card S-child ≤ M-of l
    using Cons.hyps by blast
  have head: card S-child ≤ card U-child
    by (rule source-progress[OF child child-pre - child-card])
      (use child-reaches in blast)
  show ?case
    using head Cons.IH Cons.hyps by simp
qed
have source-sum:
  sum-list (map card child-sources) ≤
    card (range-tree-chain P a bs B')
  by (rule sum-card-dominated-by-range-tree-child-list-le-chain
    [OF mono source-dom])
have child-sum:
  sum-list child-costs ≤
    A * L * card (range-tree-chain P a bs B') +
    R * card (outgoing-edges (range-tree-chain P a bs B'))
  by (rule child-costs-le-level-range-child-list-bound
    [OF mono child-cost-bounds])
have edge-sum:
  sum-list
    (map (λX. card (outgoing-edges X)) (range-tree-child-list P a bs)) ≤
   card (outgoing-edges (range-tree-chain P a bs B'))
  by (rule card-outgoing-edges-range-tree-child-list-le-chain[OF mono])
have c ≤
    A * L * card (range-tree-chain P a bs B') +
    R * card (outgoing-edges (range-tree-chain P a bs B')) +
    sum-list (map card child-sources) +
    t * (card (outgoing-edges (range-tree-chain P a bs B')) +
      sum-list (map card child-sources))
proof −
  have tail:
    t * (sum-list
        (map (λX. card (outgoing-edges X)) (range-tree-child-list P a bs)) +
        sum-list (map card child-sources)) ≤
     t * (card (outgoing-edges (range-tree-chain P a bs B')) +
        sum-list (map card child-sources))
    using edge-sum by simp
  have sum-list child-costs + sum-list (map card child-sources) +
      t * (sum-list
        (map (λX. card (outgoing-edges X)) (range-tree-child-list P a bs)) +
        sum-list (map card child-sources)) ≤
      A * L * card (range-tree-chain P a bs B') +
      R * card (outgoing-edges (range-tree-chain P a bs B')) +
      sum-list (map card child-sources) +
      t * (card (outgoing-edges (range-tree-chain P a bs B')) +
        sum-list (map card child-sources))
```

```
    using child-sum tail by linarith
  then show ?thesis
    using cost by linarith
qed
also have ... ≤
    A * L * card (range-tree-chain P a bs B') +
    (R + t) * card (outgoing-edges (range-tree-chain P a bs B')) +
    Suc t * card (range-tree-chain P a bs B')
proof −
  have source-part:
    sum-list (map card child-sources) +
      t * sum-list (map card child-sources) ≤
     Suc t * card (range-tree-chain P a bs B')
  proof −
    have t-part:
      t * sum-list (map card child-sources) ≤
        t * card (range-tree-chain P a bs B')
      using source-sum by simp
    have sum-list (map card child-sources) +
        t * sum-list (map card child-sources) ≤
      card (range-tree-chain P a bs B') +
        t * card (range-tree-chain P a bs B')
      using source-sum t-part by linarith
    then show ?thesis
      by (simp add: algebra-simps)
  qed
  show ?thesis
    using source-part by (simp add: algebra-simps)
qed
finally show ?thesis .
qed

theorem range-costed-partition-loop-state-cost-from-child-cost-bounds:
  assumes run:
    range-costed-partition-loop-state Δ M-of t k cap l d P B d' D a
      betas bs B' Us U c child-costs
    and child-cost-bounds:
      list-all2 (λc-child U-child.
        c-child ≤ level-range-cost-bound A R L U-child)
        child-costs (range-tree-child-list P a bs)
  shows c ≤
    A * L * card (range-tree-chain P a bs B') +
    (R + t) * card (outgoing-edges (range-tree-chain P a bs B')) +
    (Suc t) * (M-of l) * length bs
proof −
  have child-len: length child-costs = length bs
    by (rule range-costed-partition-loop-state-child-call-ranges(1)[OF run])
  have cost:
    c ≤ sum-list child-costs + (M-of l) * length child-costs +
```

```
    t * (sum-list
      (map (λX. card (outgoing-edges X)) (range-tree-child-list P a bs)) +
      (M-of l) * length child-costs)
  by (rule range-costed-partition-loop-state-cost-bound[OF run])
have mono: nondecreasing-from a bs
  by (rule range-costed-partition-loop-state-mono[OF run])
have child-sum:
  sum-list child-costs ≤
    A * L * card (range-tree-chain P a bs B') +
    R * card (outgoing-edges (range-tree-chain P a bs B'))
  by (rule child-costs-le-level-range-child-list-bound
    [OF mono child-cost-bounds])
have edge-sum:
  sum-list
    (map (λX. card (outgoing-edges X)) (range-tree-child-list P a bs)) ≤
   card (outgoing-edges (range-tree-chain P a bs B'))
  by (rule card-outgoing-edges-range-tree-child-list-le-chain[OF mono])
have c ≤
    A * L * card (range-tree-chain P a bs B') +
    R * card (outgoing-edges (range-tree-chain P a bs B')) +
    (M-of l) * length bs +
    t * (card (outgoing-edges (range-tree-chain P a bs B')) +
      (M-of l) * length bs)
proof −
  have middle:
    (M-of l) * length child-costs = (M-of l) * length bs
    using child-len by simp
  have tail:
    t * (sum-list
        (map (λX. card (outgoing-edges X)) (range-tree-child-list P a bs)) +
        (M-of l) * length child-costs) ≤
     t * (card (outgoing-edges (range-tree-chain P a bs B')) +
        (M-of l) * length bs)
  proof −
    have sum-list
        (map (λX. card (outgoing-edges X)) (range-tree-child-list P a bs)) +
        (M-of l) * length child-costs ≤
      card (outgoing-edges (range-tree-chain P a bs B')) +
        (M-of l) * length bs
      using edge-sum middle by linarith
    then show ?thesis
      by simp
  qed
  have sum-list child-costs + (M-of l) * length child-costs +
      t * (sum-list
        (map (λX. card (outgoing-edges X)) (range-tree-child-list P a bs)) +
        (M-of l) * length child-costs) ≤
      A * L * card (range-tree-chain P a bs B') +
      R * card (outgoing-edges (range-tree-chain P a bs B')) +
```

```
      (M-of l) * length bs +
      t * (card (outgoing-edges (range-tree-chain P a bs B')) +
        (M-of l) * length bs)
    using child-sum middle tail by linarith
  then show ?thesis
    using cost by linarith
qed
also have ... =
    A * L * card (range-tree-chain P a bs B') +
    (R + t) * card (outgoing-edges (range-tree-chain P a bs B')) +
    (Suc t) * (M-of l) * length bs
  by (simp add: algebra-simps)
finally show ?thesis .
qed

theorem range-costed-partition-loop-state-child-cost-bounds:
  assumes run:
    range-costed-partition-loop-state Δ M-of t k cap l d P B d' D a
      betas bs B' Us U c child-costs
    and child-bound:
      ⋀c-child U-child S-child B-child d-child B-child'.
        ⟦range-costed-bmssp Δ M-of t k cap l d S-child B-child
            d-child B-child' U-child c-child;
          bmssp-pre-full d S-child B-child;
          ⋀x. x ∈ S-child ⟹ reachable s x;
          card S-child ≤ M-of l⟧
        ⟹ c-child ≤ level-range-cost-bound A R L U-child
  shows list-all2 (λc-child U-child.
      c-child ≤ level-range-cost-bound A R L U-child)
    child-costs (range-tree-child-list P a bs)
proof −
  have calls:
    list-all2
      (λc-child U-child. ∃ S-child B-child d-child B-child'.
        range-costed-bmssp Δ M-of t k cap l d S-child B-child
          d-child B-child' U-child c-child ∧
        bmssp-pre-full d S-child B-child ∧
        (∀ x∈S-child. reachable s x) ∧
        card S-child ≤ M-of l)
      child-costs (range-tree-child-list P a bs)
    by (rule range-costed-partition-loop-state-child-call-ranges(2)[OF run])
  show ?thesis
    using calls
  proof (induction rule: list-all2-induct)
    case Nil
    then show ?case by simp
  next
    case (Cons c-child child-costs U-child child-sets)
    obtain S-child B-child d-child B-child' where child:
```

```
      range-costed-bmssp Δ M-of t k cap l d S-child B-child
        d-child B-child' U-child c-child
    and child-pre: bmssp-pre-full d S-child B-child
    and child-reaches: ∀ x∈S-child. reachable s x
    and child-card: card S-child ≤ M-of l
    using Cons.hyps by blast
  have head:
    c-child ≤ level-range-cost-bound A R L U-child
    by (rule child-bound[OF child child-pre - child-card])
      (use child-reaches in blast)
  show ?case
    using head Cons.IH Cons.hyps by simp
 qed
qed

theorem range-costed-partition-loop-state-cost-from-child-bound:
 assumes run:
  range-costed-partition-loop-state Δ M-of t k cap l d P B d' D a
    betas bs B' Us U c child-costs
  and child-bound:
    ⋀c-child U-child S-child B-child d-child B-child'.
      ⟦range-costed-bmssp Δ M-of t k cap l d S-child B-child
          d-child B-child' U-child c-child;
        bmssp-pre-full d S-child B-child;
        ⋀x. x ∈ S-child ⟹ reachable s x;
        card S-child ≤ M-of l⟧
      ⟹ c-child ≤ level-range-cost-bound A R L U-child
 shows c ≤
  A * L * card (range-tree-chain P a bs B') +
  (R + t) * card (outgoing-edges (range-tree-chain P a bs B')) +
  (Suc t) * (M-of l) * length bs
proof −
 have child-cost-bounds:
  list-all2 (λc-child U-child.
    c-child ≤ level-range-cost-bound A R L U-child)
    child-costs (range-tree-child-list P a bs)
  by (rule range-costed-partition-loop-state-child-cost-bounds
    [OF run child-bound])
 show ?thesis
  by (rule range-costed-partition-loop-state-cost-from-child-cost-bounds
    [OF run child-cost-bounds])
qed

theorem range-costed-partition-loop-state-trace:
 range-costed-partition-loop-state Δ M-of t k cap l d P B d' D a
   betas bs B' Us U c child-costs ⟹
  sound-label d ⟹
  bmssp-pre-full d P B ⟹
  (⋀x. x ∈ P ⟹ reachable s x) ⟹
```

```
   concrete-partition-loop-trace P B a bs d' B' Us U
and range-costed-bmssp-correct:
  range-costed-bmssp Δ M-of t k cap l d S B d' B' U c ⟹
   sound-label d ⟹
   bmssp-pre-full d S B ⟹
   (⋀x. x ∈ S ⟹ reachable s x) ⟹
   bmssp-post-full d S B d' B' U
proof (induction rule:
    range-costed-partition-loop-state-range-costed-bmssp.inducts)
  case (Range-State-Done D a B d' P Δ M-of t k cap l d)
  have a-le: bound-le (Fin a) B
    using Range-State-Done by blast
  have complete-B: complete-on d' (bound-tree P B)
    using Range-State-Done by blast
  have lower: complete-on d' (bound-tree P (Fin a))
    by (rule complete-on-subset[OF complete-B])
      (rule bound-tree-bound-mono[OF a-le])
  have range-complete: complete-on d' (range-tree P a B)
    by (rule complete-on-subset[OF complete-B])
      (rule range-tree-subset-bound-tree)
  have B-le: bound-le B B
    by (cases B) simp-all
  show ?case
    unfolding concrete-partition-loop-trace-def
    using B-le a-le lower range-complete by simp
next
  case (Range-State-Stop a B d' P k cap Δ M-of t l d D)
  then show ?case
    unfolding concrete-partition-loop-trace-def complete-on-def by simp
next
  case Range-State-Step
  then show ?case
    unfolding concrete-partition-loop-trace-def bmssp-post-full-def
      complete-preserved-def
    by (auto split: bound.splits)
next
  case (Range-Cost-Base S x Δ M-of t k cap d B)
  have post:
    bmssp-post d S B
      (λv. if v ∈ base-case-vertices k x B then dist s v else d v)
      (base-case-bound k x B)
      (base-case-vertices k x B)
    using base-case-result-bmssp-post[OF Range-Cost-Base.hyps,
        where k = k and B = B and d = d]
    unfolding base-case-result-def by simp
  then show ?case
    by (rule bmssp-post-imp-post-full)
next
  case (Range-Cost-Step D k cap d S B c-insert t Δ M-of l d' a
```

```
    betas bs B' Us U-loop c-loop child-costs-loop U c)
  let ?d-fp = find-pivots-label-capped k cap d S B
  let ?P = find-pivots-pivots-capped k cap d S B
  let ?W = {v ∈ bound-tree S B'. ?d-fp v = dist s v}
  have sound-fp: sound-label ?d-fp
    unfolding find-pivots-label-capped-def
    by (rule fp-iter-capped-label-sound[OF Range-Cost-Step.prems(1)
      Range-Cost-Step.prems(3)])
  have pivot-pre: bmssp-pre-full ?d-fp ?P B
    using find-pivots-capped-establishes-pivot-pre-concrete
      [OF Range-Cost-Step.prems(1) Range-Cost-Step.prems(2)
        Range-Cost-Step.prems(3)] .
  have P-subset-S: ?P ⊆ S
    unfolding find-pivots-pivots-capped-def by auto
  have P-reaches: ⋀x. x ∈ ?P ⟹ reachable s x
    using P-subset-S Range-Cost-Step.prems(3) by blast
  have trace:
    concrete-partition-loop-trace ?P B a bs d' B' Us U-loop
    using Range-Cost-Step.IH sound-fp pivot-pre P-reaches by blast
  have loop-post: bmssp-post-full ?d-fp ?P B d' B' U-loop
    by (rule concrete-partition-loop-trace-post[OF trace])
  have le: bound-le B' B
    using loop-post unfolding bmssp-post-full-def by blast
  have U-tree: U = bound-tree S B'
  proof −
    have U-loop ∪ ?W = bound-tree S B'
      by (rule concrete-capped-find-pivots-final-tree-assembly
        [OF Range-Cost-Step.prems(1) Range-Cost-Step.prems(2)
          Range-Cost-Step.prems(3) loop-post])
    then show ?thesis
      using Range-Cost-Step(6) by simp
  qed
  have loop-complete: complete-on d' U-loop
    using loop-post unfolding bmssp-post-full-def by blast
  have U-complete: complete-on d' U
    using complete-on-Un[OF loop-complete Range-Cost-Step(5)]
      Range-Cost-Step(6) by simp
  show ?case
    using le U-tree U-complete unfolding bmssp-post-full-def by blast
qed

theorem range-costed-bmssp-source-card-le-if-label-below-output:
  assumes run:
      range-costed-bmssp Δ M-of t k cap l d S B d' B' U c
    and sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and complete: complete-on d S
    and below: ⋀x. x ∈ S ⟹ below-bound (d x) B'
```

shows *card S* ≤ *card U*
proof −
have *post*: *bmssp-post-full d S B d′ B′ U*
by (*rule range-costed-bmssp-correct*[*OF run sound pre S-reaches*])
show *?thesis*
by (*rule bmssp-post-full-source-card-le-if-label-below-output*
[*OF post complete S-reaches below*])
qed

theorem *range-costed-bmssp-source-card-le-if-sound-label-below-output*:
assumes *run*:
range-costed-bmssp Δ *M-of t k cap l d S B d′ B′ U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: ⋀*x*. *x* ∈ *S* ⟹ *reachable s x*
and *below*: ⋀*x*. *x* ∈ *S* ⟹ *below-bound* (*d x*) *B′*
shows *card S* ≤ *card U*
proof −
have *post*: *bmssp-post-full d S B d′ B′ U*
by (*rule range-costed-bmssp-correct*[*OF run sound pre S-reaches*])
have *U-eq*: *U* = *bound-tree S B′*
using *post* **unfolding** *bmssp-post-full-def* **by** *blast*
have *S-subset-U*: *S* ⊆ *U*
proof
fix *x*
assume *xS*: *x* ∈ *S*
have *S-subset*: *S* ⊆ *V*
using *pre* **unfolding** *bmssp-pre-full-def* **by** *blast*
have *xV*: *x* ∈ *V*
using *S-subset xS* **by** *blast*
have *reach-x*: *reachable s x*
by (*rule S-reaches*[*OF xS*])
have *dist-le*: *dist s x* ≤ *d x*
using *sound xV reach-x* **unfolding** *sound-label-def* **by** *blast*
have *below-dist*: *below-bound* (*dist s x*) *B′*
using *below-bound-le-trans*[*OF dist-le below*[*OF xS*]] **.**
show *x* ∈ *U*
unfolding *U-eq*
by (*rule source-in-own-bound-tree*[*OF xS reach-x below-dist*])
qed
have *finite-U*: *finite U*
unfolding *U-eq bound-tree-def* **using** *finite-V* **by** *auto*
show *?thesis*
by (*rule card-mono*[*OF finite-U S-subset-U*])
qed

theorem *range-costed-partition-loop-state-success-or-threshold*:
range-costed-partition-loop-state Δ *M-of t k cap l d P B d′ D a*
betas bs B′ Us U c child-costs ⟹

sound-label d $\Longrightarrow$
bmssp-pre-full d P B $\Longrightarrow$
$(\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*$) \Longrightarrow$
$B' = B \vee k * cap \leq$ *card U*
and *range-costed-bmssp-success-or-threshold-trivial*:
range-costed-bmssp Δ *M-of t k cap l d S B d' B' U c* $\Longrightarrow$ *True*
proof (*induction rule*: *range-costed-partition-loop-state-range-costed-bmssp.inducts*)
case (*Range-State-Done D a B d' P* Δ *M-of t k cap l d*)
then show *?case*
by *simp*
next
case (*Range-State-Stop a B d' P k cap* Δ *M-of t l d D*)
then show *?case*
by *simp*
next
case (*Range-State-Step D M-of l Bmax S-pull beta D-pull B d a b B-out*
d-out P k cap Delta t d-child U-child c-child batch D-next c-pull
c-batch betas bs Us-tail U-tail c-tail child-costs-tail c)
have *tail*:
B-out $= B \vee k * cap \leq$ *card U-tail*
using *Range-State-Step.IH Range-State-Step.prems* **by** *blast*
then show *?case*
proof
assume *B-out* $= B$
then show *?thesis*
by *blast*
next
assume *threshold-tail*: $k * cap \leq$ *card U-tail*
have *tail-trace*:
concrete-partition-loop-trace P B b bs d-out B-out Us-tail U-tail
by (*rule range-costed-partition-loop-state-trace*)
(*use Range-State-Step* **in** *blast*)+
have *U-tail-def*:
U-tail $=$ *bound-tree P* (*Fin b*) $\cup \bigcup$(*set Us-tail*)
using *tail-trace* **unfolding** *concrete-partition-loop-trace-def* **by** *blast*
have *split-b*:
bound-tree P (*Fin b*) $=$
bound-tree P (*Fin a*) $\cup$ *range-tree P a* (*Fin b*)
using *bound-tree-split-at*[*of a Fin b P*] *Range-State-Step*
by *simp*
have *U-eq-tail*:
bound-tree P (*Fin a*) $\cup$
$\bigcup$(*set* (*range-tree P a* (*Fin b*) # *Us-tail*)) $=$
U-tail
using *U-tail-def split-b* **by** *auto*
show *?thesis*
using *threshold-tail U-eq-tail* **by** *simp*
qed
next

case (*Range-Cost-Base S x Δ M-of t k cap d B*)
then show *?case*
by *simp*
next
case (*Range-Cost-Step D k cap d S B c-insert t Δ M-of l d′ a betas bs B′ Us U-loop c-loop child-costs-loop U c*)
then show *?case*
by *simp*
qed

theorem *range-costed-bmssp-Suc-success-or-threshold*:
assumes *run*:
range-costed-bmssp Δ M-of t k cap (*Suc l*) *d S B d′ B′ U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
shows $B' = B \vee k * cap \leq$ *card U*
using *run*
proof *cases*
case (*Range-Cost-Step D c-insert a betas bs Us U-loop c-loop child-costs-loop*)
let *?d-fp* = *find-pivots-label-capped k cap d S B*
let *?P* = *find-pivots-pivots-capped k cap d S B*
let *?W* = $\{v \in$ *bound-tree S B′. ?d-fp v* = *dist s v*$\}$
have *sound-fp*: *sound-label ?d-fp*
unfolding *find-pivots-label-capped-def*
by (*rule fp-iter-capped-label-sound*[*OF sound S-reaches*])
have *pivot-pre*: *bmssp-pre-full ?d-fp ?P B*
using *find-pivots-capped-establishes-pivot-pre-concrete*
[*OF sound pre S-reaches*] **.**
have *P-subset-S*: *?P* $\subseteq S$
unfolding *find-pivots-pivots-capped-def* **by** *auto*
have *P-reaches*: $\bigwedge x.\ x \in$ *?P* $\Longrightarrow$ *reachable s x*
using *P-subset-S S-reaches* **by** *blast*
have *loop-class*:
$B' = B \vee k * cap \leq$ *card U-loop*
by (*rule range-costed-partition-loop-state-success-or-threshold*
[*OF Range-Cost-Step*(*3*) *sound-fp pivot-pre P-reaches*])
have *loop-trace*:
concrete-partition-loop-trace ?P B a bs d′ B′ Us U-loop
by (*rule range-costed-partition-loop-state-trace*
[*OF Range-Cost-Step*(*3*) *sound-fp pivot-pre P-reaches*])
have *loop-post*: *bmssp-post-full ?d-fp ?P B d′ B′ U-loop*
by (*rule concrete-partition-loop-trace-post*[*OF loop-trace*])
have *finite-U-loop*: *finite U-loop*
using *loop-post* **unfolding** *bmssp-post-full-def* **by** *simp*
show *?thesis*
using *loop-class*
proof
assume $B' = B$

```
    then show ?thesis
      by blast
  next
    assume threshold-loop: k * cap ≤ card U-loop
    have U-eq: U = U-loop ∪ ?W
      using Range-Cost-Step(5) by simp
    have finite-U: finite U
      unfolding U-eq using finite-U-loop by simp
    have U-loop-subset: U-loop ⊆ U
      unfolding U-eq by blast
    have card U-loop ≤ card U
      by (rule card-mono[OF finite-U U-loop-subset])
    then show ?thesis
      using threshold-loop by linarith
  qed
qed

theorem range-costed-bmssp-Suc-source-card-le-from-complete-sources:
  assumes run:
      range-costed-bmssp Δ M-of t k cap (Suc l) d S B d' B' U c
    and sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and complete: complete-on d S
    and below: ⋀x. x ∈ S ⟹ below-bound (d x) B
    and S-cap: card S ≤ cap
    and k-pos: 0 < k
  shows card S ≤ card U
proof −
  have call-class: B' = B ∨ k * cap ≤ card U
    by (rule range-costed-bmssp-Suc-success-or-threshold
      [OF run sound pre S-reaches])
  then show ?thesis
  proof
    assume B'-eq: B' = B
    show ?thesis
    proof (rule range-costed-bmssp-source-card-le-if-label-below-output
        [OF run sound pre S-reaches complete])
      fix x
      assume x ∈ S
      then show below-bound (d x) B'
        using below B'-eq by simp
    qed
  next
    assume threshold: k * cap ≤ card U
    have cap ≤ k * cap
      using k-pos by (cases k) simp-all
    then show ?thesis
      using S-cap threshold by linarith
```

```
  qed
qed

theorem range-costed-bmssp-Suc-source-card-le-from-label-below:
  assumes run:
      range-costed-bmssp Δ M-of t k cap (Suc l) d S B d' B' U c
    and sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and below: ⋀x. x ∈ S ⟹ below-bound (d x) B
    and S-cap: card S ≤ cap
    and k-pos: 0 < k
  shows card S ≤ card U
proof −
  have call-class: B' = B ∨ k * cap ≤ card U
    by (rule range-costed-bmssp-Suc-success-or-threshold
      [OF run sound pre S-reaches])
  then show ?thesis
  proof
    assume B'-eq: B' = B
    show ?thesis
    proof (rule range-costed-bmssp-source-card-le-if-sound-label-below-output
        [OF run sound pre S-reaches])
      fix x
      assume x ∈ S
      then show below-bound (d x) B'
        using below B'-eq by simp
    qed
  next
    assume threshold: k * cap ≤ card U
    have cap ≤ k * cap
      using k-pos by (cases k) simp-all
    then show ?thesis
      using S-cap threshold by linarith
  qed
qed

theorem range-costed-capped-step-threshold-if-not-success:
  assumes loop:
      range-costed-partition-loop-state Δ M-of t k cap l
        (find-pivots-label-capped k cap d S B)
        (find-pivots-pivots-capped k cap d S B) B d' D a betas bs B'
        Us U-loop c-loop child-costs
    and U-def:
      U = U-loop ∪
        {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v}
    and sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
```

```
    and not-success: B' ≠ B
  shows k * cap ≤ card U
proof −
  let ?d-fp = find-pivots-label-capped k cap d S B
  let ?P = find-pivots-pivots-capped k cap d S B
  let ?W = {v ∈ bound-tree S B'. ?d-fp v = dist s v}
  have sound-fp: sound-label ?d-fp
    unfolding find-pivots-label-capped-def
    by (rule fp-iter-capped-label-sound[OF sound S-reaches])
  have pivot-pre: bmssp-pre-full ?d-fp ?P B
    using find-pivots-capped-establishes-pivot-pre-concrete
      [OF sound pre S-reaches] .
  have P-subset-S: ?P ⊆ S
    unfolding find-pivots-pivots-capped-def by auto
  have P-reaches: ⋀x. x ∈ ?P ⟹ reachable s x
    using P-subset-S S-reaches by blast
  have loop-class:
    B' = B ∨ k * cap ≤ card U-loop
    by (rule range-costed-partition-loop-state-success-or-threshold
      [OF loop sound-fp pivot-pre P-reaches])
  have threshold-loop: k * cap ≤ card U-loop
    using loop-class not-success by blast
  have trace: concrete-partition-loop-trace ?P B a bs d' B' Us U-loop
    by (rule range-costed-partition-loop-state-trace
      [OF loop sound-fp pivot-pre P-reaches])
  have loop-post: bmssp-post-full ?d-fp ?P B d' B' U-loop
    by (rule concrete-partition-loop-trace-post[OF trace])
  have finite-U-loop: finite U-loop
    using loop-post unfolding bmssp-post-full-def by simp
  have finite-U: finite U
    unfolding U-def using finite-U-loop by simp
  have U-loop-subset: U-loop ⊆ U
    unfolding U-def by blast
  have card U-loop ≤ card U
    by (rule card-mono[OF finite-U U-loop-subset])
  then show ?thesis
    using threshold-loop by linarith
qed

theorem range-costed-capped-step-scan-insert-budget-if-not-success:
  assumes degree: edge-outdegree-le Δ
    and degree-factor: Δ ≤ A
    and insert-factor: t ≤ A * k
    and insert:
      partition-initial-insert-cost-bound c-insert t
        (find-pivots-pivots-capped k cap d S B)
    and loop:
      range-costed-partition-loop-state Δ M-of t k cap l
        (find-pivots-label-capped k cap d S B)
```

```
      (find-pivots-pivots-capped k cap d S B) B d' D a betas bs B'
      Us U-loop c-loop child-costs
   and U-def:
    U = U-loop ∪
      {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v}
   and sound: sound-label d
   and pre: bmssp-pre-full d S B
   and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
   and S-cap: card S ≤ cap
   and not-success: B' ≠ B
  shows fp-iter-capped-scan-cost k cap d S S B + c-insert ≤
   2 * A * card U
proof −
  have S-subset: S ⊆ V
   using pre unfolding bmssp-pre-full-def by blast
  have progress: k * cap ≤ card U
   by (rule range-costed-capped-step-threshold-if-not-success
     [OF loop U-def sound pre S-reaches not-success])
  show ?thesis
   by (rule find-pivots-scan-and-initial-insert-budget-from-progress
     [OF S-subset degree S-cap insert degree-factor insert-factor progress])
qed

theorem range-costed-partition-loop-state-closes-level-bound-general:
  assumes run:
   range-costed-partition-loop-state Δ M-of t k cap l d P B d' D a
     betas bs B' Us U c child-costs
   and sound: sound-label d
   and pre: bmssp-pre-full d P B
   and P-reaches: ⋀x. x ∈ P ⟹ reachable s x
   and child-bound:
    ⋀c-child U-child S-child B-child d-child B-child'.
     ⟦range-costed-bmssp Δ M-of t k cap l d S-child B-child
        d-child B-child' U-child c-child;
      bmssp-pre-full d S-child B-child;
      ⋀x. x ∈ S-child ⟹ reachable s x;
      card S-child ≤ M-of l⟧
     ⟹ c-child ≤ level-range-cost-bound A R L U-child
   and overhead: (Suc t) * (M-of l) * length bs ≤ A * card U
  shows c ≤ A * Suc L * card U + (R + t) * card (outgoing-edges U)
proof −
  have trace: concrete-partition-loop-trace P B a bs d' B' Us U
   by (rule range-costed-partition-loop-state-trace
     [OF run sound pre P-reaches])
  have children:
   list-all2 (λU X. U = X ∧ complete-on d' U) Us
     (range-tree-chain-list P a bs B')
   using trace unfolding concrete-partition-loop-trace-def by blast
  have U-def: U = bound-tree P (Fin a) ∪ ⋃(set Us)
```

using *trace* **unfolding** *concrete-partition-loop-trace-def* **by** *blast*
have *Union-eq*:
$\bigcup(set\ Us) = \bigcup(set\ (range\text{-}tree\text{-}chain\text{-}list\ P\ a\ bs\ B'))$
using *children* **by** (*induction rule*: *list-all2-induct*) *auto*
have *U-eq-chain*:
$U = bound\text{-}tree\ P\ (Fin\ a) \cup range\text{-}tree\text{-}chain\ P\ a\ bs\ B'$
using *U-def Union-eq Union-range-tree-chain-list*[*of P a bs B'*] **by** *simp*
have *finite-U*: *finite U*
using *U-eq-chain* **by** *simp*
have *chain-subset*: $range\text{-}tree\text{-}chain\ P\ a\ bs\ B' \subseteq U$
using *U-eq-chain* **by** *blast*
have *card-chain-le*: $card\ (range\text{-}tree\text{-}chain\ P\ a\ bs\ B') \leq card\ U$
by (*rule card-mono*[*OF finite-U chain-subset*])
have *outgoing-subset*:
$outgoing\text{-}edges\ (range\text{-}tree\text{-}chain\ P\ a\ bs\ B') \subseteq outgoing\text{-}edges\ U$
by (*rule outgoing-edges-mono*[*OF chain-subset*])
have *finite-out-U*: *finite* (*outgoing-edges U*)
by *simp*
have *card-out-le*:
$card\ (outgoing\text{-}edges\ (range\text{-}tree\text{-}chain\ P\ a\ bs\ B')) \leq$
$card\ (outgoing\text{-}edges\ U)$
by (*rule card-mono*[*OF finite-out-U outgoing-subset*])
have *loop*:
$c \leq$
$A * L * card\ (range\text{-}tree\text{-}chain\ P\ a\ bs\ B') +$
$(R + t) * card\ (outgoing\text{-}edges\ (range\text{-}tree\text{-}chain\ P\ a\ bs\ B')) +$
$(Suc\ t) * (M\text{-}of\ l) * length\ bs$
by (*rule range-costed-partition-loop-state-cost-from-child-bound*
[*OF run child-bound*])
have $c \leq$
$A * L * card\ U + (R + t) * card\ (outgoing\text{-}edges\ U) + A * card\ U$
proof −
have *vterm*:
$A * L * card\ (range\text{-}tree\text{-}chain\ P\ a\ bs\ B') \leq A * L * card\ U$
using *card-chain-le* **by** *simp*
have *eterm*:
$(R + t) * card\ (outgoing\text{-}edges\ (range\text{-}tree\text{-}chain\ P\ a\ bs\ B')) \leq$
$(R + t) * card\ (outgoing\text{-}edges\ U)$
using *card-out-le* **by** *simp*
have $A * L * card\ (range\text{-}tree\text{-}chain\ P\ a\ bs\ B') +$
$(R + t) * card\ (outgoing\text{-}edges\ (range\text{-}tree\text{-}chain\ P\ a\ bs\ B')) +$
$(Suc\ t) * (M\text{-}of\ l) * length\ bs \leq$
$A * L * card\ U + (R + t) * card\ (outgoing\text{-}edges\ U) + A * card\ U$
using *vterm eterm overhead* **by** *linarith*
then show *?thesis*
using *loop* **by** *linarith*
qed
also have ... =
$A * Suc\ L * card\ U + (R + t) * card\ (outgoing\text{-}edges\ U)$

```
    by (simp add: algebra-simps)
  finally show ?thesis .
qed

theorem range-costed-partition-loop-state-closes-level-bound-from-source-progress-general:
  assumes run:
    range-costed-partition-loop-state Δ M-of t k cap l d P B d' D a
      betas bs B' Us U c child-costs
    and sound: sound-label d
    and pre: bmssp-pre-full d P B
    and P-reaches: ⋀x. x ∈ P ⟹ reachable s x
    and child-bound:
      ⋀c-child U-child S-child B-child d-child B-child'.
        ⟦range-costed-bmssp Δ M-of t k cap l d S-child B-child
            d-child B-child' U-child c-child;
          bmssp-pre-full d S-child B-child;
          ⋀x. x ∈ S-child ⟹ reachable s x;
          card S-child ≤ M-of l⟧
        ⟹ c-child ≤ level-range-cost-bound A R L U-child
    and source-progress:
      ⋀S-child B-child d-child B-child' U-child c-child.
        ⟦range-costed-bmssp Δ M-of t k cap l d S-child B-child
            d-child B-child' U-child c-child;
          bmssp-pre-full d S-child B-child;
          ⋀x. x ∈ S-child ⟹ reachable s x;
          card S-child ≤ M-of l⟧
        ⟹ card S-child ≤ card U-child
    and source-factor: Suc t ≤ A
  shows c ≤ A * Suc L * card U + (R + t) * card (outgoing-edges U)
proof −
  have trace: concrete-partition-loop-trace P B a bs d' B' Us U
    by (rule range-costed-partition-loop-state-trace
      [OF run sound pre P-reaches])
  have children:
    list-all2 (λU X. U = X ∧ complete-on d' U) Us
      (range-tree-chain-list P a bs B')
    using trace unfolding concrete-partition-loop-trace-def by blast
  have U-def: U = bound-tree P (Fin a) ∪ ⋃(set Us)
    using trace unfolding concrete-partition-loop-trace-def by blast
  have Union-eq:
    ⋃(set Us) = ⋃(set (range-tree-chain-list P a bs B'))
    using children by (induction rule: list-all2-induct) auto
  have U-eq-chain:
    U = bound-tree P (Fin a) ∪ range-tree-chain P a bs B'
    using U-def Union-eq Union-range-tree-chain-list[of P a bs B'] by simp
  have finite-U: finite U
    using U-eq-chain by simp
  have chain-subset: range-tree-chain P a bs B' ⊆ U
```

```
  using U-eq-chain by blast
have card-chain-le: card (range-tree-chain P a bs B') ≤ card U
  by (rule card-mono[OF finite-U chain-subset])
have outgoing-subset:
  outgoing-edges (range-tree-chain P a bs B') ⊆ outgoing-edges U
  by (rule outgoing-edges-mono[OF chain-subset])
have finite-out-U: finite (outgoing-edges U)
  by simp
have card-out-le:
  card (outgoing-edges (range-tree-chain P a bs B')) ≤
    card (outgoing-edges U)
  by (rule card-mono[OF finite-out-U outgoing-subset])
have child-cost-bounds:
  list-all2 (λc-child U-child.
    c-child ≤ level-range-cost-bound A R L U-child)
    child-costs (range-tree-child-list P a bs)
  by (rule range-costed-partition-loop-state-child-cost-bounds
    [OF run child-bound])
have loop:
  c ≤
    A * L * card (range-tree-chain P a bs B') +
    (R + t) * card (outgoing-edges (range-tree-chain P a bs B')) +
    Suc t * card (range-tree-chain P a bs B')
  by (rule range-costed-partition-loop-state-cost-from-child-and-source-bounds
    [OF run child-cost-bounds source-progress])
have c ≤
    A * L * card U + (R + t) * card (outgoing-edges U) + A * card U
proof −
  have vterm:
    A * L * card (range-tree-chain P a bs B') ≤ A * L * card U
    using card-chain-le by simp
  have eterm:
    (R + t) * card (outgoing-edges (range-tree-chain P a bs B')) ≤
     (R + t) * card (outgoing-edges U)
    using card-out-le by simp
  have source-term:
    Suc t * card (range-tree-chain P a bs B') ≤ A * card U
  proof −
    have Suc t * card (range-tree-chain P a bs B') ≤
        A * card (range-tree-chain P a bs B')
      by (rule mult-right-mono[OF source-factor]) simp
    also have ... ≤ A * card U
      using card-chain-le by simp
    finally show ?thesis .
  qed
  have A * L * card (range-tree-chain P a bs B') +
     (R + t) * card (outgoing-edges (range-tree-chain P a bs B')) +
     Suc t * card (range-tree-chain P a bs B') ≤
     A * L * card U + (R + t) * card (outgoing-edges U) + A * card U
```

```
    using vterm eterm source-term by linarith
  then show ?thesis
    using loop by linarith
 qed
 also have ... =
    A * Suc L * card U + (R + t) * card (outgoing-edges U)
  by (simp add: algebra-simps)
 finally show ?thesis .
qed

theorem range-costed-partition-loop-state-closes-level-bound-from-child-cost-bounds-general:
 assumes run:
   range-costed-partition-loop-state Δ M-of t k cap l d P B d' D a
     betas bs B' Us U c child-costs
   and sound: sound-label d
   and pre: bmssp-pre-full d P B
   and P-reaches: ⋀x. x ∈ P ⟹ reachable s x
   and child-cost-bounds:
     list-all2 (λc-child U-child.
       c-child ≤ level-range-cost-bound A R L U-child)
       child-costs (range-tree-child-list P a bs)
   and overhead: (Suc t) * (M-of l) * length bs ≤ A * card U
 shows c ≤ A * Suc L * card U + (R + t) * card (outgoing-edges U)
proof −
 have trace: concrete-partition-loop-trace P B a bs d' B' Us U
   by (rule range-costed-partition-loop-state-trace
     [OF run sound pre P-reaches])
 have children:
   list-all2 (λU X. U = X ∧ complete-on d' U) Us
     (range-tree-chain-list P a bs B')
   using trace unfolding concrete-partition-loop-trace-def by blast
 have U-def: U = bound-tree P (Fin a) ∪ ⋃(set Us)
   using trace unfolding concrete-partition-loop-trace-def by blast
 have Union-eq:
  ⋃(set Us) = ⋃(set (range-tree-chain-list P a bs B'))
   using children by (induction rule: list-all2-induct) auto
 have U-eq-chain:
   U = bound-tree P (Fin a) ∪ range-tree-chain P a bs B'
   using U-def Union-eq Union-range-tree-chain-list[of P a bs B'] by simp
 have finite-U: finite U
   using U-eq-chain by simp
 have chain-subset: range-tree-chain P a bs B' ⊆ U
   using U-eq-chain by blast
 have card-chain-le: card (range-tree-chain P a bs B') ≤ card U
   by (rule card-mono[OF finite-U chain-subset])
 have outgoing-subset:
   outgoing-edges (range-tree-chain P a bs B') ⊆ outgoing-edges U
   by (rule outgoing-edges-mono[OF chain-subset])
```

```
  have finite-out-U: finite (outgoing-edges U)
    by simp
  have card-out-le:
    card (outgoing-edges (range-tree-chain P a bs B')) ≤
      card (outgoing-edges U)
    by (rule card-mono[OF finite-out-U outgoing-subset])
  have loop:
    c ≤
      A * L * card (range-tree-chain P a bs B') +
      (R + t) * card (outgoing-edges (range-tree-chain P a bs B')) +
      (Suc t) * (M-of l) * length bs
    by (rule range-costed-partition-loop-state-cost-from-child-cost-bounds
      [OF run child-cost-bounds])
  have c ≤
      A * L * card U + (R + t) * card (outgoing-edges U) + A * card U
  proof −
    have vterm:
      A * L * card (range-tree-chain P a bs B') ≤ A * L * card U
      using card-chain-le by simp
    have eterm:
      (R + t) * card (outgoing-edges (range-tree-chain P a bs B')) ≤
       (R + t) * card (outgoing-edges U)
      using card-out-le by simp
    have A * L * card (range-tree-chain P a bs B') +
        (R + t) * card (outgoing-edges (range-tree-chain P a bs B')) +
        (Suc t) * (M-of l) * length bs ≤
        A * L * card U + (R + t) * card (outgoing-edges U) + A * card U
      using vterm eterm overhead by linarith
    then show ?thesis
      using loop by linarith
  qed
  also have ... =
      A * Suc L * card U + (R + t) * card (outgoing-edges U)
    by (simp add: algebra-simps)
  finally show ?thesis .
qed

theorem range-costed-partition-loop-state-closes-level-bound:
  assumes run:
    range-costed-partition-loop-state Δ M-of t k cap l d P B d' D a
      betas bs B' Us U c child-costs
    and sound: sound-label d
    and pre: bmssp-pre-full d P B
    and P-reaches: ⋀x. x ∈ P ⟹ reachable s x
    and child-bound:
      ⋀c-child U-child S-child B-child d-child B-child'.
        ⟦range-costed-bmssp Δ M-of t k cap l d S-child B-child
            d-child B-child' U-child c-child;
          bmssp-pre-full d S-child B-child;
```

```
        ⋀x. x ∈ S-child ⟹ reachable s x;
        card S-child ≤ M-of l⟧
       ⟹ c-child ≤ level-range-cost-bound A R l U-child
    and overhead: (Suc t) * (M-of l) * length bs ≤ A * card U
  shows c ≤ A * Suc l * card U + (R + t) * card (outgoing-edges U)
proof −
  have trace: concrete-partition-loop-trace P B a bs d' B' Us U
    by (rule range-costed-partition-loop-state-trace
      [OF run sound pre P-reaches])
  have children:
    list-all2 (λU X. U = X ∧ complete-on d' U) Us
      (range-tree-chain-list P a bs B')
    using trace unfolding concrete-partition-loop-trace-def by blast
  have U-def: U = bound-tree P (Fin a) ∪ ⋃(set Us)
    using trace unfolding concrete-partition-loop-trace-def by blast
  have Union-eq:
    ⋃(set Us) = ⋃(set (range-tree-chain-list P a bs B'))
    using children by (induction rule: list-all2-induct) auto
  have U-eq-chain:
    U = bound-tree P (Fin a) ∪ range-tree-chain P a bs B'
    using U-def Union-eq Union-range-tree-chain-list[of P a bs B'] by simp
  have finite-U: finite U
    using U-eq-chain by simp
  have chain-subset: range-tree-chain P a bs B' ⊆ U
    using U-eq-chain by blast
  have card-chain-le: card (range-tree-chain P a bs B') ≤ card U
    by (rule card-mono[OF finite-U chain-subset])
  have outgoing-subset:
    outgoing-edges (range-tree-chain P a bs B') ⊆ outgoing-edges U
    by (rule outgoing-edges-mono[OF chain-subset])
  have finite-out-U: finite (outgoing-edges U)
    by simp
  have card-out-le:
    card (outgoing-edges (range-tree-chain P a bs B')) ≤
      card (outgoing-edges U)
    by (rule card-mono[OF finite-out-U outgoing-subset])
  have loop:
    c ≤
      A * l * card (range-tree-chain P a bs B') +
      (R + t) * card (outgoing-edges (range-tree-chain P a bs B')) +
      (Suc t) * (M-of l) * length bs
    by (rule range-costed-partition-loop-state-cost-from-child-bound
      [OF run child-bound])
  have c ≤
      A * l * card U + (R + t) * card (outgoing-edges U) + A * card U
  proof −
    have vterm:
      A * l * card (range-tree-chain P a bs B') ≤ A * l * card U
      using card-chain-le by simp
```

```
    have eterm:
      (R + t) * card (outgoing-edges (range-tree-chain P a bs B')) ≤
       (R + t) * card (outgoing-edges U)
      using card-out-le by simp
    have A * l * card (range-tree-chain P a bs B') +
       (R + t) * card (outgoing-edges (range-tree-chain P a bs B')) +
       (Suc t) * (M-of l) * length bs ≤
       A * l * card U + (R + t) * card (outgoing-edges U) + A * card U
      using vterm eterm overhead by linarith
    then show ?thesis
      using loop by linarith
  qed
  also have ... =
     A * Suc l * card U + (R + t) * card (outgoing-edges U)
    by (simp add: algebra-simps)
  finally show ?thesis .
qed

theorem range-costed-nonbase-step-closes-level-bound-general:
  assumes loop:
    range-costed-partition-loop-state Δ M-of t k cap l d P B d' D a
      betas bs B' Us U-loop c-loop child-costs
    and sound: sound-label d
    and pre: bmssp-pre-full d P B
    and P-reaches: ⋀x. x ∈ P ⟹ reachable s x
    and child-bound:
      ⋀c-child U-child S-child B-child d-child B-child'.
        ⟦range-costed-bmssp Δ M-of t k cap l d S-child B-child
            d-child B-child' U-child c-child;
          bmssp-pre-full d S-child B-child;
          ⋀x. x ∈ S-child ⟹ reachable s x;
          card S-child ≤ M-of l⟧
        ⟹ c-child ≤ level-range-cost-bound A R L U-child
    and overhead: (Suc t) * (M-of l) * length bs ≤ A * card U-loop
    and U-def: U = U-loop ∪ W
    and finite-U: finite U
    and scan-insert: c-scan-insert ≤ A * card U
    and c-def: c = c-scan-insert + c-loop
  shows c ≤ A * Suc (Suc L) * card U +
    (R + t) * card (outgoing-edges U)
proof −
  have loop-bound:
    c-loop ≤ A * Suc L * card U-loop +
      (R + t) * card (outgoing-edges U-loop)
    by (rule range-costed-partition-loop-state-closes-level-bound-general
      [OF loop sound pre P-reaches child-bound overhead])
  have U-loop-subset: U-loop ⊆ U
    using U-def by blast
  have card-loop-le: card U-loop ≤ card U
```

```
    by (rule card-mono[OF finite-U U-loop-subset])
  have outgoing-subset: outgoing-edges U-loop ⊆ outgoing-edges U
    by (rule outgoing-edges-mono[OF U-loop-subset])
  have finite-out-U: finite (outgoing-edges U)
    by simp
  have card-out-le: card (outgoing-edges U-loop) ≤ card (outgoing-edges U)
    by (rule card-mono[OF finite-out-U outgoing-subset])
  have c ≤
      A * card U +
      (A * Suc L * card U + (R + t) * card (outgoing-edges U))
  proof −
    have loop-to-U:
      c-loop ≤ A * Suc L * card U +
        (R + t) * card (outgoing-edges U)
    proof −
      have A * Suc L * card U-loop ≤ A * Suc L * card U
        using card-loop-le by simp
      moreover have (R + t) * card (outgoing-edges U-loop) ≤
          (R + t) * card (outgoing-edges U)
        using card-out-le by simp
      ultimately show ?thesis
        using loop-bound by linarith
    qed
    show ?thesis
      using scan-insert loop-to-U c-def by linarith
  qed
  also have ... =
      A * Suc (Suc L) * card U + (R + t) * card (outgoing-edges U)
    by (simp add: algebra-simps)
  finally show ?thesis .
qed

theorem range-costed-nonbase-step-closes-level-bound-from-source-progress-general:
  assumes loop:
    range-costed-partition-loop-state Δ M-of t k cap l d P B d′ D a
      betas bs B′ Us U-loop c-loop child-costs
    and sound: sound-label d
    and pre: bmssp-pre-full d P B
    and P-reaches: ⋀x. x ∈ P ⟹ reachable s x
    and child-bound:
      ⋀c-child U-child S-child B-child d-child B-child′.
        ⟦range-costed-bmssp Δ M-of t k cap l d S-child B-child
            d-child B-child′ U-child c-child;
          bmssp-pre-full d S-child B-child;
          ⋀x. x ∈ S-child ⟹ reachable s x;
          card S-child ≤ M-of l⟧
        ⟹ c-child ≤ level-range-cost-bound A R L U-child
    and source-progress:
      ⋀S-child B-child d-child B-child′ U-child c-child.
```

```
      ⟦range-costed-bmssp Δ M-of t k cap l d S-child B-child
          d-child B-child′ U-child c-child;
        bmssp-pre-full d S-child B-child;
        ⋀x. x ∈ S-child ⟹ reachable s x;
        card S-child ≤ M-of l⟧
       ⟹ card S-child ≤ card U-child
   and source-factor: Suc t ≤ A
   and U-def: U = U-loop ∪ W
   and finite-U: finite U
   and scan-insert: c-scan-insert ≤ A * card U
   and c-def: c = c-scan-insert + c-loop
 shows c ≤ A * Suc (Suc L) * card U +
   (R + t) * card (outgoing-edges U)
proof −
 have loop-bound:
   c-loop ≤ A * Suc L * card U-loop +
     (R + t) * card (outgoing-edges U-loop)
   by (rule range-costed-partition-loop-state-closes-level-bound-from-source-progress-
general
     [OF loop sound pre P-reaches child-bound source-progress source-factor])
 have U-loop-subset: U-loop ⊆ U
   using U-def by blast
 have card-loop-le: card U-loop ≤ card U
   by (rule card-mono[OF finite-U U-loop-subset])
 have outgoing-subset: outgoing-edges U-loop ⊆ outgoing-edges U
   by (rule outgoing-edges-mono[OF U-loop-subset])
 have finite-out-U: finite (outgoing-edges U)
   by simp
 have card-out-le: card (outgoing-edges U-loop) ≤ card (outgoing-edges U)
   by (rule card-mono[OF finite-out-U outgoing-subset])
 have c ≤
    A * card U +
    (A * Suc L * card U + (R + t) * card (outgoing-edges U))
 proof −
   have loop-to-U:
     c-loop ≤ A * Suc L * card U +
       (R + t) * card (outgoing-edges U)
   proof −
     have A * Suc L * card U-loop ≤ A * Suc L * card U
       using card-loop-le by simp
     moreover have (R + t) * card (outgoing-edges U-loop) ≤
         (R + t) * card (outgoing-edges U)
       using card-out-le by simp
     ultimately show ?thesis
       using loop-bound by linarith
   qed
   show ?thesis
     using scan-insert loop-to-U c-def by linarith
 qed
```

also have ... =
A ∗ *Suc* (*Suc L*) ∗ *card U* + (*R* + *t*) ∗ *card* (*outgoing-edges U*)
by (*simp add*: *algebra-simps*)
finally show *?thesis* **.**
qed

theorem *range-costed-nonbase-step-closes-level-bound-from-child-cost-bounds-general*:
assumes *loop*:
range-costed-partition-loop-state Δ *M-of t k cap l d P B d′ D a*
betas bs B′ Us U-loop c-loop child-costs
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: ⋀*x*. *x* ∈ *P* ⟹ *reachable s x*
and *child-cost-bounds*:
list-all2 (λ*c-child U-child*.
c-child ≤ *level-range-cost-bound A R L U-child*)
child-costs (*range-tree-child-list P a bs*)
and *overhead*: (*Suc t*) ∗ (*M-of l*) ∗ *length bs* ≤ *A* ∗ *card U-loop*
and *U-def*: *U* = *U-loop* ∪ *W*
and *finite-U*: *finite U*
and *scan-insert*: *c-scan-insert* ≤ *A* ∗ *card U*
and *c-def*: *c* = *c-scan-insert* + *c-loop*
shows *c* ≤ *A* ∗ *Suc* (*Suc L*) ∗ *card U* +
(*R* + *t*) ∗ *card* (*outgoing-edges U*)
proof −
have *loop-bound*:
c-loop ≤ *A* ∗ *Suc L* ∗ *card U-loop* +
(*R* + *t*) ∗ *card* (*outgoing-edges U-loop*)
by (*rule*
range-costed-partition-loop-state-closes-level-bound-from-child-cost-bounds-general
[*OF loop sound pre P-reaches child-cost-bounds overhead*])
have *U-loop-subset*: *U-loop* ⊆ *U*
using *U-def* **by** *blast*
have *card-loop-le*: *card U-loop* ≤ *card U*
by (*rule card-mono*[*OF finite-U U-loop-subset*])
have *outgoing-subset*: *outgoing-edges U-loop* ⊆ *outgoing-edges U*
by (*rule outgoing-edges-mono*[*OF U-loop-subset*])
have *finite-out-U*: *finite* (*outgoing-edges U*)
by *simp*
have *card-out-le*: *card* (*outgoing-edges U-loop*) ≤ *card* (*outgoing-edges U*)
by (*rule card-mono*[*OF finite-out-U outgoing-subset*])
have *c* ≤
A ∗ *card U* +
(*A* ∗ *Suc L* ∗ *card U* + (*R* + *t*) ∗ *card* (*outgoing-edges U*))
proof −
have *loop-to-U*:
c-loop ≤ *A* ∗ *Suc L* ∗ *card U* +

```
      (R + t) * card (outgoing-edges U)
    proof −
      have A * Suc L * card U-loop ≤ A * Suc L * card U
        using card-loop-le by simp
      moreover have (R + t) * card (outgoing-edges U-loop) ≤
          (R + t) * card (outgoing-edges U)
        using card-out-le by simp
      ultimately show ?thesis
        using loop-bound by linarith
    qed
    show ?thesis
      using scan-insert loop-to-U c-def by linarith
  qed
  also have ... =
      A * Suc (Suc L) * card U + (R + t) * card (outgoing-edges U)
    by (simp add: algebra-simps)
  finally show ?thesis .
qed

theorem range-costed-nonbase-step-closes-level-bound:
  assumes loop:
    range-costed-partition-loop-state Δ M-of t k cap l d P B d′ D a
      betas bs B′ Us U-loop c-loop child-costs
    and sound: sound-label d
    and pre: bmssp-pre-full d P B
    and P-reaches: ⋀x. x ∈ P ⟹ reachable s x
    and child-bound:
      ⋀c-child U-child S-child B-child d-child B-child′.
        ⟦range-costed-bmssp Δ M-of t k cap l d S-child B-child
            d-child B-child′ U-child c-child;
          bmssp-pre-full d S-child B-child;
          ⋀x. x ∈ S-child ⟹ reachable s x;
          card S-child ≤ M-of l⟧
        ⟹ c-child ≤ level-range-cost-bound A R l U-child
    and overhead: (Suc t) * (M-of l) * length bs ≤ A * card U-loop
    and U-def: U = U-loop ∪ W
    and finite-U: finite U
    and scan-insert: c-scan-insert ≤ A * card U
    and c-def: c = c-scan-insert + c-loop
  shows c ≤ A * Suc (Suc l) * card U +
    (R + t) * card (outgoing-edges U)
proof −
  have loop-bound:
    c-loop ≤ A * Suc l * card U-loop +
      (R + t) * card (outgoing-edges U-loop)
    by (rule range-costed-partition-loop-state-closes-level-bound
      [OF loop sound pre P-reaches child-bound overhead])
  have U-loop-subset: U-loop ⊆ U
    using U-def by blast
```

```
  have card-loop-le: card U-loop ≤ card U
    by (rule card-mono[OF finite-U U-loop-subset])
  have outgoing-subset: outgoing-edges U-loop ⊆ outgoing-edges U
    by (rule outgoing-edges-mono[OF U-loop-subset])
  have finite-out-U: finite (outgoing-edges U)
    by simp
  have card-out-le: card (outgoing-edges U-loop) ≤ card (outgoing-edges U)
    by (rule card-mono[OF finite-out-U outgoing-subset])
  have c ≤
      A * card U +
      (A * Suc l * card U + (R + t) * card (outgoing-edges U))
  proof −
    have loop-to-U:
      c-loop ≤ A * Suc l * card U +
        (R + t) * card (outgoing-edges U)
    proof −
      have A * Suc l * card U-loop ≤ A * Suc l * card U
        using card-loop-le by simp
      moreover have (R + t) * card (outgoing-edges U-loop) ≤
          (R + t) * card (outgoing-edges U)
        using card-out-le by simp
      ultimately show ?thesis
        using loop-bound by linarith
    qed
    show ?thesis
      using scan-insert loop-to-U c-def by linarith
  qed
  also have ... =
      A * Suc (Suc l) * card U + (R + t) * card (outgoing-edges U)
    by (simp add: algebra-simps)
  finally show ?thesis .
qed

theorem range-costed-base-level-bound:
  assumes R-pos: 0 < R
  shows base-case-scan-cost Δ k x B ≤
    level-range-cost-bound A R (Suc 0) (base-case-vertices k x B)
proof −
  have base-case-scan-cost Δ k x B ≤
      R * card (outgoing-edges (base-case-vertices k x B))
    using R-pos unfolding base-case-scan-cost-def by simp
  then show ?thesis
    unfolding level-range-cost-bound-def by linarith
qed
```

The unsplit relation repeats the same recurrence in a form that is closer to the abstract cost interface. The base theorem *0 < ?R ⟹ base-case-scan-cost ?Δ ?k ?x ?B ≤ level-range-cost-bound ?A ?R (Suc 0) (base-case-vertices ?k ?x ?B)* pays for the bounded base scan with the outgoing-edge component

of *level-range-cost-bound*. The main theorem below then performs induction on the recursion level: the induction hypothesis bounds each child call, while the supplied local-budget hypotheses pay for loop overhead and FindPivots overhead at the current level.

This theorem is intentionally parameterised by the constants A, R, t, and *M-of*. Later theories instantiate those parameters with the BMSSP schedule and with the bucketed partition costs, but the recurrence itself does not depend on how the partition primitive is implemented.

theorem *range-costed-bmssp-level-bound-from-local-budgets*:
  assumes *base-budget*: $\Delta \leq A$
    and *R-pos*: $0 < R$
    and *loop-budget*:
      $\bigwedge l\ d\ P\ B\ d'\ D\ a\ betas\ bs\ B'\ Us\ U\ c\ child\text{-}costs.$
        range-costed-partition-loop-state Δ *M-of t k cap l d P B d' D a*
          betas bs B' Us U c child-costs $\Longrightarrow$
        sound-label d $\Longrightarrow$
        bmssp-pre-full d P B $\Longrightarrow$
        $(\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*$) \Longrightarrow$
        (Suc t) $*$ *(M-of l)* $*$ *length bs* $\leq A *$ *card U*
    and *step-budget*:
      $\bigwedge l\ d\ S\ B\ D\ c\text{-}insert\ d'\ a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop$
          child-costs U.
        D $=$ *label-partition-view*
          (*find-pivots-label-capped k cap d S B*)
          (*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
        partition-initial-insert-cost-bound c-insert t
          (*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
        range-costed-partition-loop-state Δ *M-of t k cap l*
          (*find-pivots-label-capped k cap d S B*)
          (*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B'*
          Us U-loop c-loop child-costs $\Longrightarrow$
        complete-on d'
          $\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* $=$ *dist s v*$\} \Longrightarrow$
        U $=$ *U-loop* $\cup$
          $\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* $=$ *dist s v*$\} \Longrightarrow$
        sound-label d $\Longrightarrow$
        bmssp-pre-full d S B $\Longrightarrow$
        $(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
        fp-iter-capped-scan-cost k cap d S S B $+$ *c-insert* $\leq A *$ *card U*
    and *run*:
      range-costed-bmssp Δ *M-of t k cap l d S B d' B' U c*
    and *sound*: *sound-label d*
    and *pre*: *bmssp-pre-full d S B*
    and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
  shows *c* $\leq$ *level-range-cost-bound A (R* $+$ *l* $*$ *t) (2* $*$ *l* $+$ *1) U*
  using *run sound pre S-reaches*
proof (*induction l arbitrary*: *d S B d' B' U c rule*: *less-induct*)
  case (*less l*)

```
show ?case
proof (cases l)
  case 0
  have run0:
    range-costed-bmssp Δ M-of t k cap 0 d S B d' B' U c
    using less.prems(1) 0 by simp
  show ?thesis
    using run0 R-pos 0
    by (cases rule: range-costed-bmssp.cases)
      (simp-all add: range-costed-base-level-bound)
next
  case (Suc l0)
  have run-suc:
    range-costed-bmssp Δ M-of t k cap (Suc l0) d S B d' B' U c
    using less.prems(1) Suc by simp
  show ?thesis
  proof (cases rule: range-costed-bmssp-SucE[OF run-suc])
    case (1 c-insert a betas bs Us U-loop c-loop child-costs-loop)
    let ?d-fp = find-pivots-label-capped k cap d S B
    let ?P = find-pivots-pivots-capped k cap d S B
    let ?W = {v ∈ bound-tree S B'. ?d-fp v = dist s v}
    have sound-fp: sound-label ?d-fp
      unfolding find-pivots-label-capped-def
      by (rule fp-iter-capped-label-sound[OF less.prems(2) less.prems(4)])
    have pivot-pre: bmssp-pre-full ?d-fp ?P B
      using find-pivots-capped-establishes-pivot-pre-concrete
        [OF less.prems(2) less.prems(3) less.prems(4)] .
    have P-subset-S: ?P ⊆ S
      unfolding find-pivots-pivots-capped-def by auto
    have P-reaches: ⋀x. x ∈ ?P ⟹ reachable s x
      using P-subset-S less.prems(4) by blast
    have child-bound:
      ⋀c-child U-child S-child B-child d-child B-child'.
        ⟦range-costed-bmssp Δ M-of t k cap l0 ?d-fp S-child B-child
            d-child B-child' U-child c-child;
          bmssp-pre-full ?d-fp S-child B-child;
          ⋀x. x ∈ S-child ⟹ reachable s x;
          card S-child ≤ M-of l0⟧
        ⟹ c-child ≤ level-range-cost-bound A (R + l0 * t)
          (2 * l0 + 1) U-child
    proof −
      fix c-child U-child S-child B-child d-child B-child'
      assume child:
          range-costed-bmssp Δ M-of t k cap l0 ?d-fp S-child B-child
            d-child B-child' U-child c-child
        and child-pre: bmssp-pre-full ?d-fp S-child B-child
        and child-reaches: ⋀x. x ∈ S-child ⟹ reachable s x
      show c-child ≤ level-range-cost-bound A (R + l0 * t)
          (2 * l0 + 1) U-child
```

```
      by (rule less.IH[of l0 ?d-fp S-child B-child d-child B-child'
            U-child c-child, OF - child sound-fp child-pre child-reaches])
        (simp add: Suc)
    qed
    have overhead:
      (Suc t) * (M-of l0) * length bs ≤ A * card U-loop
      by (rule loop-budget[OF 1(4) sound-fp pivot-pre P-reaches])
    have scan-insert:
      fp-iter-capped-scan-cost k cap d S S B + c-insert ≤
        A * card U
      by (rule step-budget[OF refl 1(3) 1(4) 1(5) 1(1)
          less.prems(2) less.prems(3) less.prems(4)])
    have trace:
      concrete-partition-loop-trace ?P B a bs d' B' Us U-loop
      by (rule range-costed-partition-loop-state-trace
        [OF 1(4) sound-fp pivot-pre P-reaches])
    have loop-post: bmssp-post-full ?d-fp ?P B d' B' U-loop
      by (rule concrete-partition-loop-trace-post[OF trace])
    have finite-U-loop: finite U-loop
      using loop-post unfolding bmssp-post-full-def by simp
    have finite-W: finite ?W
      by simp
    have finite-U: finite U
      using 1(1) finite-U-loop finite-W by simp
    have step:
      c ≤ A * Suc (Suc (2 * l0 + 1)) * card U +
        (R + l0 * t + t) * card (outgoing-edges U)
      by (rule range-costed-nonbase-step-closes-level-bound-general
        [OF 1(4) sound-fp pivot-pre P-reaches child-bound overhead
          1(1) finite-U scan-insert 1(2)])
    show ?thesis
      using step Suc unfolding level-range-cost-bound-def
      by (simp add: algebra-simps)
  qed
 qed
qed

theorem range-costed-bmssp-level-bound-from-source-progress-and-local-budgets:
 assumes base-budget: Δ ≤ A
   and R-pos: 0 < R
   and source-factor: Suc t ≤ A
   and source-progress:
     ⋀l d S B d' B' U c.
       range-costed-bmssp Δ M-of t k cap l d S B d' B' U c ⟹
       bmssp-pre-full d S B ⟹
       (⋀x. x ∈ S ⟹ reachable s x) ⟹
       card S ≤ M-of l ⟹
       card S ≤ card U
   and step-budget:
```

```
    ⋀l d S B D c-insert d′ a betas bs B′ Us U-loop c-loop
       child-costs U.
     D = label-partition-view
      (find-pivots-label-capped k cap d S B)
      (find-pivots-pivots-capped k cap d S B) ⟹
     partition-initial-insert-cost-bound c-insert t
      (find-pivots-pivots-capped k cap d S B) ⟹
     range-costed-partition-loop-state Δ M-of t k cap l
      (find-pivots-label-capped k cap d S B)
      (find-pivots-pivots-capped k cap d S B) B d′ D a betas bs B′
      Us U-loop c-loop child-costs ⟹
     complete-on d′
      {v ∈ bound-tree S B′. find-pivots-label-capped k cap d S B v = dist s v} ⟹
     U = U-loop ∪
      {v ∈ bound-tree S B′. find-pivots-label-capped k cap d S B v = dist s v} ⟹
     sound-label d ⟹
     bmssp-pre-full d S B ⟹
     (⋀x. x ∈ S ⟹ reachable s x) ⟹
     fp-iter-capped-scan-cost k cap d S S B + c-insert ≤ A * card U
  and run:
    range-costed-bmssp Δ M-of t k cap l d S B d′ B′ U c
  and sound: sound-label d
  and pre: bmssp-pre-full d S B
  and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
 shows c ≤ level-range-cost-bound A (R + l * t) (2 * l + 1) U
 using run sound pre S-reaches
proof (induction l arbitrary: d S B d′ B′ U c rule: less-induct)
 case (less l)
 show ?case
 proof (cases l)
  case 0
  have run0:
    range-costed-bmssp Δ M-of t k cap 0 d S B d′ B′ U c
    using less.prems(1) 0 by simp
  show ?thesis
    using run0 R-pos 0
    by (cases rule: range-costed-bmssp.cases)
      (simp-all add: range-costed-base-level-bound)
 next
  case (Suc l0)
  have run-suc:
    range-costed-bmssp Δ M-of t k cap (Suc l0) d S B d′ B′ U c
    using less.prems(1) Suc by simp
  show ?thesis
  proof (cases rule: range-costed-bmssp-SucE[OF run-suc])
    case (1 c-insert a betas bs Us U-loop c-loop child-costs-loop)
    let ?d-fp = find-pivots-label-capped k cap d S B
    let ?P = find-pivots-pivots-capped k cap d S B
    let ?W = {v ∈ bound-tree S B′. ?d-fp v = dist s v}
```

have *sound-fp*: *sound-label ?d-fp*
unfolding *find-pivots-label-capped-def*
by (*rule fp-iter-capped-label-sound*[*OF less.prems*(*2*) *less.prems*(*4*)])
have *pivot-pre*: *bmssp-pre-full ?d-fp ?P B*
using *find-pivots-capped-establishes-pivot-pre-concrete*
[*OF less.prems*(*2*) *less.prems*(*3*) *less.prems*(*4*)] **.**
have *P-subset-S*: *?P* $\subseteq$ *S*
unfolding *find-pivots-pivots-capped-def* **by** *auto*
have *P-reaches*: $\bigwedge x.\ x \in ?P \Longrightarrow$ *reachable s x*
using *P-subset-S less.prems*(*4*) **by** *blast*
have *child-bound*:
$\bigwedge$*c-child U-child S-child B-child d-child B-child'*.
⟦*range-costed-bmssp* Δ *M-of t k cap l0 ?d-fp S-child B-child*
d-child B-child' U-child c-child;
bmssp-pre-full ?d-fp S-child B-child;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l0*⟧
$\Longrightarrow$ *c-child* $\leq$ *level-range-cost-bound A* (*R* + *l0* $*$ *t*)
(*2* $*$ *l0* + *1*) *U-child*
proof −
fix *c-child U-child S-child B-child d-child B-child'*
assume *child*:
range-costed-bmssp Δ *M-of t k cap l0 ?d-fp S-child B-child*
d-child B-child' U-child c-child
and *child-pre*: *bmssp-pre-full ?d-fp S-child B-child*
and *child-reaches*: $\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*
show *c-child* $\leq$ *level-range-cost-bound A* (*R* + *l0* $*$ *t*)
(*2* $*$ *l0* + *1*) *U-child*
by (*rule less.IH*[*of l0 ?d-fp S-child B-child d-child B-child'*
U-child c-child, *OF - child sound-fp child-pre child-reaches*])
(*simp add*: *Suc*)
qed
have *source-progress-loop*:
$\bigwedge$*S-child B-child d-child B-child' U-child c-child*.
⟦*range-costed-bmssp* Δ *M-of t k cap l0 ?d-fp S-child B-child*
d-child B-child' U-child c-child;
bmssp-pre-full ?d-fp S-child B-child;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l0*⟧
$\Longrightarrow$ *card S-child* $\leq$ *card U-child*
by (*rule source-progress*)
have *scan-insert*:
fp-iter-capped-scan-cost k cap d S S B + *c-insert* $\leq$
A $*$ *card U*
by (*rule step-budget*[*OF refl 1*(*3*) *1*(*4*) *1*(*5*) *1*(*1*)
less.prems(*2*) *less.prems*(*3*) *less.prems*(*4*)])
have *trace*:
concrete-partition-loop-trace ?P B a bs d' B' Us U-loop
by (*rule range-costed-partition-loop-state-trace*

```
        [OF 1(4) sound-fp pivot-pre P-reaches])
    have loop-post: bmssp-post-full ?d-fp ?P B d' B' U-loop
      by (rule concrete-partition-loop-trace-post[OF trace])
    have finite-U-loop: finite U-loop
      using loop-post unfolding bmssp-post-full-def by simp
    have finite-W: finite ?W
      by simp
    have finite-U: finite U
      using 1(1) finite-U-loop finite-W by simp
    have step:
      c ≤ A * Suc (Suc (2 * l0 + 1)) * card U +
        (R + l0 * t + t) * card (outgoing-edges U)
      by (rule range-costed-nonbase-step-closes-level-bound-from-source-progress-
general
        [OF 1(4) sound-fp pivot-pre P-reaches child-bound
          source-progress-loop source-factor 1(1) finite-U scan-insert 1(2)])
    show ?thesis
      using step Suc unfolding level-range-cost-bound-def
      by (simp add: algebra-simps)
  qed
 qed
qed

theorem range-costed-bmssp-level-bound-from-source-progress-budgets:
 assumes degree: edge-outdegree-le Δ
   and degree-factor: Δ ≤ A
   and R-pos: 0 < R
   and insert-factor: t ≤ A * k
   and source-factor: Suc t ≤ 2 * A
   and source-progress:
     ⋀l d S B d' B' U c.
       range-costed-bmssp Δ M-of t k cap l d S B d' B' U c ⟹
       bmssp-pre-full d S B ⟹
       (⋀x. x ∈ S ⟹ reachable s x) ⟹
       card S ≤ M-of l ⟹
       card S ≤ card U
   and step-progress:
     ⋀l d S B D c-insert d' a betas bs B' Us U-loop c-loop
       child-costs U.
       D = label-partition-view
         (find-pivots-label-capped k cap d S B)
         (find-pivots-pivots-capped k cap d S B) ⟹
       partition-initial-insert-cost-bound c-insert t
         (find-pivots-pivots-capped k cap d S B) ⟹
       range-costed-partition-loop-state Δ M-of t k cap l
         (find-pivots-label-capped k cap d S B)
         (find-pivots-pivots-capped k cap d S B) B d' D a betas bs B'
         Us U-loop c-loop child-costs ⟹
       complete-on d'
```

$\{v \in$ *bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v*$\} \Longrightarrow$
U = U-loop $\cup$
$\{v \in$ *bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v*$\} \Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
card S $\leq$ *cap* $\wedge$ *k* $*$ *cap* $\leq$ *card U*
and *run*:
range-costed-bmssp Δ *M-of t k cap l d S B d' B' U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
shows *c* $\leq$ *level-range-cost-bound* (*2* $*$ *A*) (*R* + *l* $*$ *t*) (*2* $*$ *l* + *1*) *U*
proof (*rule range-costed-bmssp-level-bound-from-source-progress-and-local-budgets*
[**where** *A* = *2* $*$ *A* **and** *R* = *R*,
OF - R-pos source-factor - - run sound pre S-reaches])
show $\Delta \leq$ *2* $*$ *A*
using *degree-factor* **by** *simp*
next
fix *l d S B d' B' U c*
assume *child*:
range-costed-bmssp Δ *M-of t k cap l d S B d' B' U c*
and *child-pre*: *bmssp-pre-full d S B*
and *child-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *child-card*: *card S* $\leq$ *M-of l*
show *card S* $\leq$ *card U*
by (*rule source-progress*[*OF child child-pre child-reaches child-card*])
next
fix *l d S B D c-insert d' a betas bs B' Us U-loop c-loop child-costs U*
assume *D-def*:
D = *label-partition-view*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*)
and *insert*:
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*)
and *loop*:
range-costed-partition-loop-state Δ *M-of t k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B'*
Us U-loop c-loop child-costs
and *complete*:
complete-on d'
$\{v \in$ *bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v*$\}$
and *U-def*:
U = U-loop $\cup$
$\{v \in$ *bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v*$\}$
and *sound-s*: *sound-label d*
and *pre-s*: *bmssp-pre-full d S B*

and *reaches-s*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
have *S-subset*: $S \subseteq V$
using *pre-s* **unfolding** *bmssp-pre-full-def* **by** *blast*
have *progress*:
card S $\leq$ *cap* $\wedge$ *k* $*$ *cap* $\leq$ *card U*
by (*rule step-progress*[*OF D-def insert loop complete U-def sound-s pre-s reaches-s*])
then have *S-cap*: *card S* $\leq$ *cap*
by *blast*
from *progress* **have** *U-progress*: *k* $*$ *cap* $\leq$ *card U*
by *blast*
show *fp-iter-capped-scan-cost k cap d S S B* + *c-insert* $\leq$
2 $*$ *A* $*$ *card U*
by (*rule find-pivots-scan-and-initial-insert-budget-from-progress*
[*OF S-subset degree S-cap insert degree-factor insert-factor U-progress*])
qed

theorem *range-costed-bmssp-level-bound-from-progress-budgets*:
assumes *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: *0* $<$ *R*
and *insert-factor*: *t* $\leq$ *A* $*$ *k*
and *loop-factor*: *Suc t* $\leq$ *2* $*$ *A*
and *loop-progress*:
$\bigwedge l\ d\ P\ B\ d'\ D\ a\ betas\ bs\ B'\ Us\ U\ c\ child\text{-}costs.$
range-costed-partition-loop-state Δ *M-of t k cap l d P B d' D a*
betas bs B' Us U c child-costs $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d P B $\Longrightarrow$
$(\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*$) \Longrightarrow$
(*M-of l*) $*$ *length bs* $\leq$ *card U*
and *step-progress*:
$\bigwedge l\ d\ S\ B\ D\ c\text{-}insert\ d'\ a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop$
child-costs U.
D = *label-partition-view*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
range-costed-partition-loop-state Δ *M-of t k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B'*
Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d'
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* = *dist s v*$\}$ $\Longrightarrow$
U = *U-loop* $\cup$
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* = *dist s v*$\}$ $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$

```
      (⋀x. x ∈ S ⟹ reachable s x) ⟹
      card S ≤ cap ∧ k * cap ≤ card U
    and run:
      range-costed-bmssp Δ M-of t k cap l d S B d' B' U c
    and sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
  shows c ≤ level-range-cost-bound (2 * A) (R + l * t) (2 * l + 1) U
proof (rule range-costed-bmssp-level-bound-from-local-budgets
    [where A = 2 * A and R = R,
      OF - R-pos - - run sound pre S-reaches])
  show Δ ≤ 2 * A
    using degree-factor by simp
next
  fix l d P B d' D a betas bs B' Us U c child-costs
  assume loop:
      range-costed-partition-loop-state Δ M-of t k cap l d P B d' D a
        betas bs B' Us U c child-costs
    and sound-l: sound-label d
    and pre-l: bmssp-pre-full d P B
    and reaches-l: ⋀x. x ∈ P ⟹ reachable s x
  have progress: (M-of l) * length bs ≤ card U
    by (rule loop-progress[OF loop sound-l pre-l reaches-l])
  show (Suc t) * (M-of l) * length bs ≤ 2 * A * card U
    by (rule loop-overhead-budget-from-progress[OF loop-factor progress])
next
  fix l d S B D c-insert d' a betas bs B' Us U-loop c-loop child-costs U
  assume D-def:
      D = label-partition-view
        (find-pivots-label-capped k cap d S B)
        (find-pivots-pivots-capped k cap d S B)
    and insert:
      partition-initial-insert-cost-bound c-insert t
        (find-pivots-pivots-capped k cap d S B)
    and loop:
      range-costed-partition-loop-state Δ M-of t k cap l
        (find-pivots-label-capped k cap d S B)
        (find-pivots-pivots-capped k cap d S B) B d' D a betas bs B'
        Us U-loop c-loop child-costs
    and complete:
      complete-on d'
        {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v}
    and U-def:
      U = U-loop ∪
        {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v}
    and sound-s: sound-label d
    and pre-s: bmssp-pre-full d S B
    and reaches-s: ⋀x. x ∈ S ⟹ reachable s x
  have S-subset: S ⊆ V
```

using *pre-s* **unfolding** *bmssp-pre-full-def* **by** *blast*
have *progress*:
card S $\leq$ *cap* $\wedge$ *k* $*$ *cap* $\leq$ *card U*
by (*rule step-progress*[*OF D-def insert loop complete U-def sound-s pre-s reaches-s*])
then have *S-cap*: *card S* $\leq$ *cap*
by *blast*
from *progress* **have** *U-progress*: *k* $*$ *cap* $\leq$ *card U*
by *blast*
show *fp-iter-capped-scan-cost k cap d S S B* + *c-insert* $\leq$
2 $*$ *A* $*$ *card U*
by (*rule find-pivots-scan-and-initial-insert-budget-from-progress*
[*OF S-subset degree S-cap insert degree-factor insert-factor U-progress*])
qed

theorem *finite-initial-label-range-costed-top-level-correct*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *run*:
range-costed-bmssp Δ *M-of t k cap l finite-initial-label* $\{s\}$
Infinity d′ Infinity U c
shows *sssp-correct d′*
proof −
have *pre*: *bmssp-pre-full finite-initial-label* $\{s\}$ *Infinity*
using *all-reachable finite-initial-label-source-complete*
by (*rule top-bmssp-pre-full*)
have *sound*: *sound-label finite-initial-label*
using *finite-initial-label-sound*[*OF all-reachable*] **.**
have *S-reaches*: $\bigwedge x.\ x \in \{s\} \Longrightarrow$ *reachable s x*
using *all-reachable source-in-V* **by** *blast*
have *post-full*:
bmssp-post-full finite-initial-label $\{s\}$ *Infinity d′ Infinity U*
by (*rule range-costed-bmssp-correct*[*OF run sound pre S-reaches*])
have *post*: *bmssp-post finite-initial-label* $\{s\}$ *Infinity d′ Infinity U*
using *bmssp-post-full-imp-post*[*OF post-full*] **.**
then have *U-V*: *U* = *V*
using *bound-tree-source-infinity*[*OF all-reachable*]
unfolding *bmssp-post-def* **by** *auto*
have *complete*: *complete-on d′ U*
using *post* **unfolding** *bmssp-post-def* **by** *auto*
then show *?thesis*
using *U-V* **unfolding** *complete-on-def sssp-correct-def* **by** *auto*
qed

theorem *finite-initial-label-range-costed-top-level-correct-and-bound*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: *0* $<$ *R*
and *insert-factor*: *t* $\leq$ *A* $*$ *k*

```
    and loop-factor: Suc t ≤ 2 * A
    and loop-progress:
      ⋀l d P B d' D a betas bs B' Us U c child-costs.
        range-costed-partition-loop-state Δ M-of t k cap l d P B d' D a
          betas bs B' Us U c child-costs ⟹
        sound-label d ⟹
        bmssp-pre-full d P B ⟹
        (⋀x. x ∈ P ⟹ reachable s x) ⟹
        (M-of l) * length bs ≤ card U
    and step-progress:
      ⋀l d S B D c-insert d' a betas bs B' Us U-loop c-loop
          child-costs U.
        D = label-partition-view
          (find-pivots-label-capped k cap d S B)
          (find-pivots-pivots-capped k cap d S B) ⟹
        partition-initial-insert-cost-bound c-insert t
          (find-pivots-pivots-capped k cap d S B) ⟹
        range-costed-partition-loop-state Δ M-of t k cap l
          (find-pivots-label-capped k cap d S B)
          (find-pivots-pivots-capped k cap d S B) B d' D a betas bs B'
          Us U-loop c-loop child-costs ⟹
        complete-on d'
          {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v} ⟹
        U = U-loop ∪
          {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v} ⟹
        sound-label d ⟹
        bmssp-pre-full d S B ⟹
        (⋀x. x ∈ S ⟹ reachable s x) ⟹
        card S ≤ cap ∧ k * cap ≤ card U
    and run:
      range-costed-bmssp Δ M-of t k cap l finite-initial-label {s}
        Infinity d' Infinity U c
  shows sssp-correct d' ∧
    c ≤ level-range-cost-bound (2 * A) (R + l * t) (2 * l + 1) U
proof
  show sssp-correct d'
    by (rule finite-initial-label-range-costed-top-level-correct
      [OF all-reachable run])
next
  have pre: bmssp-pre-full finite-initial-label {s} Infinity
    using all-reachable finite-initial-label-source-complete
    by (rule top-bmssp-pre-full)
  have sound: sound-label finite-initial-label
    using finite-initial-label-sound[OF all-reachable] .
  have S-reaches: ⋀x. x ∈ {s} ⟹ reachable s x
    using all-reachable source-in-V by blast
  show c ≤ level-range-cost-bound (2 * A) (R + l * t) (2 * l + 1) U
    by (rule range-costed-bmssp-level-bound-from-progress-budgets
      [OF degree degree-factor R-pos insert-factor loop-factor loop-progress
```

step-progress run sound pre S-reaches])
qed

theorem *finite-initial-label-range-costed-top-level-correct-and-source-progress-bound*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *source-factor*: *Suc* $t \leq 2 * A$
and *source-progress*:
$\bigwedge l\ d\ S\ B\ d'\ B'\ U\ c.$
range-costed-bmssp Δ *M-of t k cap l d S B d' B' U c* $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
card S $\leq$ *M-of l* $\Longrightarrow$
card S $\leq$ *card U*
and *step-progress*:
$\bigwedge l\ d\ S\ B\ D$ *c-insert d' a betas bs B' Us U-loop c-loop*
child-costs U.
$D =$ *label-partition-view*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
range-costed-partition-loop-state Δ *M-of t k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B'*
Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d'
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* $=$ *dist s v*$\} \Longrightarrow$
$U =$ *U-loop* $\cup$
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* $=$ *dist s v*$\} \Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
card S $\leq$ *cap* $\wedge$ $k * cap \leq$ *card U*
and *run*:
range-costed-bmssp Δ *M-of t k cap l finite-initial-label* $\{s\}$
Infinity d' Infinity U c
shows *sssp-correct d'* $\wedge$
$c \leq$ *level-range-cost-bound* $(2 * A)$ $(R + l * t)$ $(2 * l + 1)$ *U*
proof
show *sssp-correct d'*
by (*rule finite-initial-label-range-costed-top-level-correct*
[*OF all-reachable run*])
next
have *pre*: *bmssp-pre-full finite-initial-label* $\{s\}$ *Infinity*

using *all-reachable finite-initial-label-source-complete*
by (*rule top-bmssp-pre-full*)
have *sound*: *sound-label finite-initial-label*
using *finite-initial-label-sound*[*OF all-reachable*] **.**
have *S-reaches*: $\bigwedge x.\ x \in \{s\} \Longrightarrow$ *reachable s x*
using *all-reachable source-in-V* **by** *blast*
show $c \leq$ *level-range-cost-bound* $(2 * A)\ (R + l * t)\ (2 * l + 1)\ U$
by (*rule range-costed-bmssp-level-bound-from-source-progress-budgets*
[*OF degree degree-factor R-pos insert-factor source-factor source-progress*
step-progress run sound pre S-reaches])
qed

theorem *finite-initial-label-range-costed-top-level-correct-and-graph-bound*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *source-factor*: *Suc* $t \leq 2 * A$
and *source-progress*:
$\bigwedge l\ d\ S\ B\ d'\ B'\ U\ c.$
range-costed-bmssp Δ *M-of t k cap l d S B d' B' U c* $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$)$ $\Longrightarrow$
card S $\leq$ *M-of l* $\Longrightarrow$
card S $\leq$ *card U*
and *step-progress*:
$\bigwedge l\ d\ S\ B\ D$ *c-insert d' a betas bs B' Us U-loop c-loop*
child-costs U.
$D =$ *label-partition-view*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
range-costed-partition-loop-state Δ *M-of t k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B'*
Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d'
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* $=$ *dist s v*$\}$ $\Longrightarrow$
$U = U\text{-}loop \cup$
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* $=$ *dist s v*$\}$ $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$)$ $\Longrightarrow$
card S $\leq$ *cap* $\wedge$ $k * cap \leq$ *card U*
and *run*:
range-costed-bmssp Δ *M-of t k cap l finite-initial-label* $\{s\}$
Infinity d' Infinity U c

```
  shows sssp-correct d' ∧
    c ≤ 2 * A * (2 * l + 1) * vertex-count + (R + l * t) * edge-count
proof −
  have combined:
    sssp-correct d' ∧
      c ≤ level-range-cost-bound (2 * A) (R + l * t) (2 * l + 1) U
     by (rule finite-initial-label-range-costed-top-level-correct-and-source-progress-
bound
      [OF all-reachable degree degree-factor R-pos insert-factor source-factor
        source-progress step-progress run])
  then have correct: sssp-correct d'
    by blast
  have pre: bmssp-pre-full finite-initial-label {s} Infinity
    using all-reachable finite-initial-label-source-complete
    by (rule top-bmssp-pre-full)
  have sound: sound-label finite-initial-label
    using finite-initial-label-sound[OF all-reachable] .
  have S-reaches: ⋀x. x ∈ {s} ⟹ reachable s x
    using all-reachable source-in-V by blast
  have post-full:
    bmssp-post-full finite-initial-label {s} Infinity d' Infinity U
    by (rule range-costed-bmssp-correct[OF run sound pre S-reaches])
  have post: bmssp-post finite-initial-label {s} Infinity d' Infinity U
    using bmssp-post-full-imp-post[OF post-full] .
  have U-V: U = V
    using post bound-tree-source-infinity[OF all-reachable]
    unfolding bmssp-post-def by auto
  have cost-U:
      c ≤ level-range-cost-bound (2 * A) (R + l * t) (2 * l + 1) U
    using combined by blast
  have graph-bound:
    level-range-cost-bound (2 * A) (R + l * t) (2 * l + 1) U
      ≤ 2 * A * (2 * l + 1) * vertex-count + (R + l * t) * edge-count
  proof −
    have U-card: card U = vertex-count
      unfolding U-V vertex-count-def by simp
    have out-le: card (outgoing-edges U) ≤ edge-count
      by (rule edge-count-outgoing-bound)
    have out-term:
      (R + l * t) * card (outgoing-edges U) ≤ (R + l * t) * edge-count
      using out-le by simp
    show ?thesis
      unfolding level-range-cost-bound-def
      using U-card out-term by (simp add: algebra-simps)
  qed
  show ?thesis
    using correct cost-U graph-bound by linarith
qed
```

end

end
theory *BMSSP-Exact-Range-Costed*
imports *BMSSP-Range-Costed*
begin

31 Exact Split Range-Costed Runs

The range-costed relation records the algorithm's range accounting, but leaves the pulled child source abstract. The exact variant below keeps the same cost and range information while additionally recording that each pull source is exactly the lower label split used by the operational algorithm. It refines the existing range-costed relation, so the established correctness and cost lemmas remain reusable.

This extra equality is small syntactically and important analytically. A pull operation returns a child source set and a child bound. For correctness it is enough to know that the child call satisfies the BMSSP precondition. For progress and runtime, however, the source set must be identified with the mathematical split of the parent sources below the child bound. Only then can the proof inherit the antichain and scaled-cardinality facts that were proved for lower splits.

The theory therefore sits between range synchronisation and direct insertion. It keeps the range slices, child-cost list, and edge/source batch split from the preceding layer, while adding enough exactness to discharge the source progress side conditions needed by the recurrence. Later theories refine this one by splitting edge batches further; they rely on the exact source information introduced here rather than reproving it.

No new algorithm is introduced. Each theorem either forgets the exactness to recover the split-range relation, or uses exactness to derive stronger cost premises for the same loop trace.

context *unique-shortest-digraph*
begin

inductive *exact-split-range-costed-partition-loop-state*
and *exact-split-range-costed-bmssp* **where**
Exact-Split-Range-State-Done:
keys-of D = {} ⟹
bound-le (*Fin a*) *B* ⟹
complete-on d′ (*bound-tree P B*) ⟹
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
[] [] *B* [*range-tree P a B*]
(*bound-tree P* (*Fin a*) ∪ ⋃(*set* [*range-tree P a B*])) *0* []
| *Exact-Split-Range-State-Stop*:
bound-le (*Fin a*) *B* ⟹

complete-on d' (*bound-tree* P (*Fin* a)) $\Longrightarrow$
$k * cap \leq$ *card* (*bound-tree* P (*Fin* a)) $\Longrightarrow$
exact-split-range-costed-partition-loop-state Δ *M-of* t h k *cap* l d P B d' D a
[] [] (*Fin* a) [*range-tree* P a (*Fin* a)]
(*bound-tree* P (*Fin* a) $\cup \bigcup$(*set* [*range-tree* P a (*Fin* a)])) 0 []
| *Exact-Split-Range-State-Step*:
pull-separates D (*M-of* l) *Bmax S-pull beta D-pull* $\Longrightarrow$
bound-le (*Fin beta*) B $\Longrightarrow$
bmssp-pre-full d *S-pull* (*Fin beta*) $\Longrightarrow$
S-pull = *split-below* d P *beta* $\Longrightarrow$
$(\forall x \in$ *S-pull*. *reachable* s $x) \Longrightarrow$
$a \leq b \Longrightarrow$
bound-le (*Fin* a) $B' \Longrightarrow$
complete-on d' (*bound-tree* P (*Fin* a)) $\Longrightarrow$
card (*bound-tree* P (*Fin* a)) $< k * cap \Longrightarrow$
exact-split-range-costed-bmssp Δ *M-of* t h k *cap* l d *S-pull* (*Fin beta*)
d-child (*Fin* b) *U-child c-child* $\Longrightarrow$
U-child = *range-tree* P a (*Fin* b) $\Longrightarrow$
complete-preserved d-child d' *U-child* $\Longrightarrow$
edge-batch = *edge-relaxation-pairs-between d-child U-child b beta* $\Longrightarrow$
source-batch = *label-pairs-between* d *S-pull* b *beta* $\Longrightarrow$
batch = *edge-batch* @ *source-batch* $\Longrightarrow$
D-next = *batch-min-update D-pull batch* $\Longrightarrow$
partition-pull-cost-bound c-pull S-pull $\Longrightarrow$
partition-batch-cost-bound c-edges t edge-batch $\Longrightarrow$
partition-batch-cost-bound c-sources h source-batch $\Longrightarrow$
exact-split-range-costed-partition-loop-state Δ *M-of* t h k *cap* l d P B d'
D-next b betas bs B' *Us-tail U-tail c-tail child-costs-tail* $\Longrightarrow$
$c =$ *c-pull* + *c-edges* + *c-sources* + *c-child* + *c-tail* $\Longrightarrow$
exact-split-range-costed-partition-loop-state Δ *M-of* t h k *cap* l d P B d' D a
(*beta* # *betas*) (b # *bs*) B'
(*range-tree* P a (*Fin* b) # *Us-tail*)
(*bound-tree* P (*Fin* a) $\cup$
$\bigcup$(*set* (*range-tree* P a (*Fin* b) # *Us-tail*))) c
(*c-child* # *child-costs-tail*)
| *Exact-Split-Range-Cost-Base*:
$S = \{x\} \Longrightarrow$
exact-split-range-costed-bmssp Δ *M-of* t h k *cap* 0 d S B
(λv. *if* $v \in$ *base-case-vertices* k x B *then dist* s v *else* d v)
(*base-case-bound* k x B)
(*base-case-vertices* k x B)
(*base-case-scan-cost* Δ k x B)
| *Exact-Split-Range-Cost-Step*:
$D =$ *label-partition-view*
(*find-pivots-label-capped* k *cap* d S B)
(*find-pivots-pivots-capped* k *cap* d S B) $\Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped* k *cap* d S B) $\Longrightarrow$
exact-split-range-costed-partition-loop-state Δ *M-of* t h k *cap* l

```
    (find-pivots-label-capped k cap d S B)
    (find-pivots-pivots-capped k cap d S B) B d' D a betas bs B'
    Us U-loop c-loop child-costs-loop ⟹
   complete-on d'
    {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v} ⟹
   U = U-loop ∪
   {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v} ⟹
   c = fp-iter-capped-scan-cost k cap d S S B + c-insert + c-loop ⟹
   exact-split-range-costed-bmssp Δ M-of t h k cap (Suc l) d S B d' B' U c
```

The defining difference from the split-range relation is the premise *S-pull = split-below d P beta* in the loop step. This says that the implementation-level pull has not merely produced an admissible child set; it has produced exactly the semantic lower split of the parent problem. The rest of the constructor mirrors the earlier relation: edge and source batches are charged separately, the child output is the next *range-tree*, and the total cost is the sum of local and recursive costs.

The equality to *split-below* is deliberately stored at the point where the pull happens. Downstream proofs then extract it from the derivation tree rather than reconstructing it from the partition invariant. This keeps the cost proofs focused on summing costs and ranges.

```
inductive-cases exact-split-range-costed-bmssp-zeroE:
  exact-split-range-costed-bmssp Δ M-of t h k cap 0 d S B d' B' U c

inductive-cases exact-split-range-costed-bmssp-SucE:
  exact-split-range-costed-bmssp Δ M-of t h k cap (Suc l) d S B d' B' U c

theorem exact-split-range-costed-refines-split-range-costed:
  shows
    exact-split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
      betas bs B' Us U c child-costs ⟹
     split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
      betas bs B' Us U c child-costs
  and
    exact-split-range-costed-bmssp Δ M-of t h k cap l d S B d' B' U c ⟹
     split-range-costed-bmssp Δ M-of t h k cap l d S B d' B' U c
proof (induction rule:
            exact-split-range-costed-partition-loop-state-exact-split-range-costed-
bmssp.inducts)
  case (Exact-Split-Range-State-Done D a B d' P Δ M-of t h k cap l d)
  show ?case
    by (rule Split-Range-State-Done)
      (use Exact-Split-Range-State-Done in auto)
next
  case (Exact-Split-Range-State-Stop a B d' P k cap Δ M-of t h l d D)
  show ?case
    by (rule Split-Range-State-Stop)
      (use Exact-Split-Range-State-Stop in auto)
```

```
next
  case (Exact-Split-Range-State-Step D M-of l Bmax S-pull beta D-pull B d
      P a b B' d' k cap Δ t h d-child U-child c-child edge-batch
      source-batch batch D-next c-pull c-edges c-sources betas bs Us-tail
      U-tail c-tail child-costs-tail c)
  show ?case
    by (rule Split-Range-State-Step)
      (use Exact-Split-Range-State-Step in auto)
next
  case (Exact-Split-Range-Cost-Base S x Δ M-of t h k cap d B)
  show ?case
    by (rule Split-Range-Cost-Base)
      (use Exact-Split-Range-Cost-Base in auto)
next
  case (Exact-Split-Range-Cost-Step D k cap d S B c-insert t Δ M-of h l
      d' a betas bs B' Us U-loop c-loop child-costs-loop U c)
  show ?case
    by (rule Split-Range-Cost-Step)
      (use Exact-Split-Range-Cost-Step in auto)
qed

theorem exact-split-range-costed-bmssp-correct:
  assumes run:
    exact-split-range-costed-bmssp Δ M-of t h k cap l d S B d' B' U c
    and sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
  shows bmssp-post-full d S B d' B' U
  by (rule split-range-costed-bmssp-correct
    [OF exact-split-range-costed-refines-split-range-costed(2)[OF run]
      sound pre S-reaches])

theorem finite-initial-label-exact-split-range-costed-top-level-correct:
  assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
    and run:
      exact-split-range-costed-bmssp Δ M-of t h k cap l
        finite-initial-label {s} Infinity d' Infinity U c
  shows sssp-correct d'
  by (rule finite-initial-label-split-range-costed-top-level-correct
    [OF all-reachable exact-split-range-costed-refines-split-range-costed(2)
      [OF run]])
```

The refinement theorem *exact-split-range-costed-partition-loop-state ?Δ ?M-of ?t ?h ?k ?cap ?l ?d ?P ?B ?d′ ?D ?a ?betas ?bs ?B′ ?Us ?U ?c ?child-costs ⟹ split-range-costed-partition-loop-state ?Δ ?M-of ?t ?h ?k ?cap ?l ?d ?P ?B ?d′ ?D ?a ?betas ?bs ?B′ ?Us ?U ?c ?child-costs* is the compatibility layer: forgetting the exact source equality gives a valid split-range run. Correctness and top-level SSSP correctness are then inherited immediately from the previous theory.

The remaining results use the exact source equality in the other direction. They expose the child calls together with the lower-split facts needed for the induction hypothesis: bounded cardinality, reachability, and labels below the child bound. Those facts are the source-progress side of the runtime proof.

theorem *exact-split-range-costed-partition-loop-state-child-cost-calls-below*:
  exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
    betas bs B′ Us U c child-costs $\Longrightarrow$
   length child-costs = *length bs* $\wedge$
   list-all2
    (*λc-child U-child*. $\exists$ *S-child B-child d-child B-child′*.
      exact-split-range-costed-bmssp Δ *M-of t h k cap l d S-child B-child*
       d-child B-child′ U-child c-child $\wedge$
      bmssp-pre-full d S-child B-child $\wedge$
      ($\forall$ *x*$\in$*S-child*. *reachable s x*) $\wedge$
      card S-child $\leq$ *M-of l* $\wedge$
      ($\forall$ *x*$\in$*S-child*. *below-bound* (*d x*) *B-child*))
    child-costs (*range-tree-child-list P a bs*)
and *exact-split-range-costed-bmssp-child-cost-calls-below-trivial*:
  exact-split-range-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c* $\Longrightarrow$ *True*
proof (*induction rule*:
        exact-split-range-costed-partition-loop-state-exact-split-range-costed-bmssp.inducts)
  case (*Exact-Split-Range-State-Done D a B d′ P* Δ *M-of t h k cap l d*)
  show *?case*
    by *simp*
next
  case (*Exact-Split-Range-State-Stop a B d′ P k cap* Δ *M-of t h l d D*)
  show *?case*
    by *simp*
next
  case (*Exact-Split-Range-State-Step D M-of l Bmax S-pull beta D-pull B d*
     P a b B′ d′ k cap Δ *t h d-child U-child c-child edge-batch*
     source-batch batch D-next c-pull c-edges c-sources betas bs Us-tail
     U-tail c-tail child-costs-tail c)
  have *tail*:
    length child-costs-tail = *length bs* $\wedge$
     list-all2
      (*λc-child U-child*. $\exists$ *S-child B-child d-child B-child′*.
        exact-split-range-costed-bmssp Δ *M-of t h k cap l d S-child B-child*
         d-child B-child′ U-child c-child $\wedge$
        bmssp-pre-full d S-child B-child $\wedge$
        ($\forall$ *x*$\in$*S-child*. *reachable s x*) $\wedge$
        card S-child $\leq$ *M-of l* $\wedge$
        ($\forall$ *x*$\in$*S-child*. *below-bound* (*d x*) *B-child*))
      child-costs-tail (*range-tree-child-list P b bs*)
    using *Exact-Split-Range-State-Step.IH* **by** *blast*
  have *card-pull*: *card S-pull* $\leq$ *M-of l*
    using *Exact-Split-Range-State-Step*
    unfolding *pull-separates-def* **by** *blast*

```
  have below-pull: ∀ x∈S-pull. below-bound (d x) (Fin beta)
  proof
    fix x
    assume xS: x ∈ S-pull
    have S-pull = split-below d P beta
      using Exact-Split-Range-State-Step by blast
    then show below-bound (d x) (Fin beta)
      using xS unfolding split-below-def by auto
  qed
  have head:
    ∃ S-child B-child d-child' B-child'.
      exact-split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
        d-child' B-child' U-child c-child ∧
      bmssp-pre-full d S-child B-child ∧
      (∀ x∈S-child. reachable s x) ∧
      card S-child ≤ M-of l ∧
      (∀ x∈S-child. below-bound (d x) B-child)
    using Exact-Split-Range-State-Step card-pull below-pull by blast
  show ?case
    using tail head Exact-Split-Range-State-Step by simp
next
  case (Exact-Split-Range-Cost-Base S x Δ M-of t h k cap d B)
  show ?case
    by simp
next
  case (Exact-Split-Range-Cost-Step D k cap d S B c-insert t Δ M-of h l
      d' a betas bs B' Us U-loop c-loop child-costs-loop U c)
  show ?case
    by simp
qed

theorem exact-split-range-costed-partition-loop-state-child-cost-bounds:
  assumes run:
    exact-split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
      betas bs B' Us U c child-costs
    and child-bound:
      ⋀c-child U-child S-child B-child d-child B-child'.
        ⟦exact-split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
            d-child B-child' U-child c-child;
          bmssp-pre-full d S-child B-child;
          ⋀x. x ∈ S-child ⟹ reachable s x;
          card S-child ≤ M-of l;
          ⋀x. x ∈ S-child ⟹ below-bound (d x) B-child⟧
        ⟹ c-child ≤ level-range-cost-bound A R L U-child
  shows list-all2 (λc-child U-child.
      c-child ≤ level-range-cost-bound A R L U-child)
    child-costs (range-tree-child-list P a bs)
proof −
  have calls:
```

list-all2
(λ*c-child U-child*. ∃ *S-child B-child d-child B-child′*.
exact-split-range-costed-bmssp Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child ∧
bmssp-pre-full d S-child B-child ∧
(∀ *x*∈*S-child*. *reachable s x*) ∧
card S-child ≤ *M-of l* ∧
(∀ *x*∈*S-child*. *below-bound* (*d x*) *B-child*))
child-costs (*range-tree-child-list P a bs*)
using *exact-split-range-costed-partition-loop-state-child-cost-calls-below*
[*OF run*] **by** *blast*
then show *?thesis*
proof (*induction rule*: *list-all2-induct*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons c-child child-costs U-child child-ranges*)
obtain *S-child B-child d-child B-child′* **where** *child*:
exact-split-range-costed-bmssp Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child
and *child-pre*: *bmssp-pre-full d S-child B-child*
and *child-reaches*: ∀ *x*∈*S-child*. *reachable s x*
and *child-card*: *card S-child* ≤ *M-of l*
and *child-below*: ∀ *x*∈*S-child*. *below-bound* (*d x*) *B-child*
using *Cons.hyps* **by** *blast*
have *head*:
c-child ≤ *level-range-cost-bound A R L U-child*
by (*rule child-bound*[*OF child child-pre - child-card*])
(*use child-reaches child-below* **in** *blast*)+
show *?case*
using *head Cons.IH Cons.hyps* **by** *simp*
qed
qed

theorem *exact-split-range-costed-partition-loop-state-child-cost-bounds-with-invariants*:
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs ⟹
P ⊆ *V* ⟹
k ∗ *card P* ≤ *cap* ⟹
tree-antichain P ⟹
(⋀*c-child U-child S-child B-child d-child B-child′*.
⟦*exact-split-range-costed-bmssp* Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child;
bmssp-pre-full d S-child B-child;
⋀*x*. *x* ∈ *S-child* ⟹ *reachable s x*;
card S-child ≤ *M-of l*;
⋀*x*. *x* ∈ *S-child* ⟹ *below-bound* (*d x*) *B-child*;
k ∗ *card S-child* ≤ *cap*;

```
        tree-antichain S-child⟧
       ⟹ c-child ≤ level-range-cost-bound A R L U-child) ⟹
    list-all2 (λc-child U-child.
      c-child ≤ level-range-cost-bound A R L U-child)
    child-costs (range-tree-child-list P a bs)
and exact-split-range-costed-bmssp-child-cost-bounds-with-invariants-trivial:
  exact-split-range-costed-bmssp Δ M-of t h k cap l d S B d' B' U c ⟹ True
proof (induction rule:
            exact-split-range-costed-partition-loop-state-exact-split-range-costed-
bmssp.inducts)
  case (Exact-Split-Range-State-Done D a B d' P Δ M-of t h k cap l d)
  then show ?case
    by simp
next
  case (Exact-Split-Range-State-Stop a B d' P k cap Δ M-of t h l d D)
  then show ?case
    by simp
next
  case (Exact-Split-Range-State-Step D M-of l Bmax S-pull beta D-pull B d
      P a b B' d' k cap Δ t h d-child U-child c-child edge-batch
      source-batch batch D-next c-pull c-edges c-sources betas bs Us-tail
      U-tail c-tail child-costs-tail c)
  have tail:
    list-all2 (λc-child U-child.
      c-child ≤ level-range-cost-bound A R L U-child)
      child-costs-tail (range-tree-child-list P b bs)
    using Exact-Split-Range-State-Step.IH Exact-Split-Range-State-Step.prems
    by blast
  have card-pull: card S-pull ≤ M-of l
    using Exact-Split-Range-State-Step
    unfolding pull-separates-def by blast
  have below-pull: ⋀x. x ∈ S-pull ⟹ below-bound (d x) (Fin beta)
  proof −
    fix x
    assume xS: x ∈ S-pull
    have S-pull = split-below d P beta
      using Exact-Split-Range-State-Step by blast
    then show below-bound (d x) (Fin beta)
      using xS unfolding split-below-def by auto
  qed
  have pull-k-cap: k * card S-pull ≤ cap
  proof −
    have S-pull = split-below d P beta
      using Exact-Split-Range-State-Step by blast
    then show ?thesis
      using split-below-scaled-card-le
        [OF Exact-Split-Range-State-Step.prems(1)
          Exact-Split-Range-State-Step.prems(2), of d beta]
      by simp
```

```
  qed
  have pull-anti: tree-antichain S-pull
  proof −
    have S-pull = split-below d P beta
      using Exact-Split-Range-State-Step by blast
    then show ?thesis
      using split-below-tree-antichain
        [OF Exact-Split-Range-State-Step.prems(3), of d beta]
      by simp
  qed
  have child-run:
    exact-split-range-costed-bmssp Δ M-of t h k cap l d S-pull (Fin beta)
      d-child (Fin b) U-child c-child
    using Exact-Split-Range-State-Step by blast
  have pre-pull: bmssp-pre-full d S-pull (Fin beta)
    using Exact-Split-Range-State-Step by blast
  have reaches-pull: ⋀x. x ∈ S-pull ⟹ reachable s x
    using Exact-Split-Range-State-Step by blast
  have head:
    c-child ≤ level-range-cost-bound A R L U-child
    by (rule Exact-Split-Range-State-Step.prems(4)
      [OF child-run pre-pull reaches-pull card-pull below-pull
        pull-k-cap pull-anti])
  show ?case
    using head tail Exact-Split-Range-State-Step by simp
next
  case (Exact-Split-Range-Cost-Base S x Δ M-of t h k cap d B)
  then show ?case
    by simp
next
  case (Exact-Split-Range-Cost-Step D k cap d S B c-insert t Δ M-of h l
      d′ a betas bs B′ Us U-loop c-loop child-costs-loop U c)
  then show ?case
    by simp
qed
```

The child-source extraction theorem packages one loop trace as a list of source sets aligned with *range-tree-child-list*. Alignment matters: each source set is charged against the same range slice as its recursive child call. Once this list exists, subsequent cost theorems can use ordinary list-wise reasoning instead of induction over the full operational relation.

```
theorem exact-split-range-costed-partition-loop-state-child-sources-below:
  exact-split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d′ D a
      betas bs B′ Us U c child-costs ⟹
   ∃ child-sources.
    length child-sources = length bs ∧
    list-all2
      (λS-child U-child. ∃ c-child B-child d-child B-child′.
        exact-split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
```

```
        d-child B-child' U-child c-child ∧
       bmssp-pre-full d S-child B-child ∧
       (∀ x∈S-child. reachable s x) ∧
       card S-child ≤ M-of l ∧
       (∀ x∈S-child. below-bound (d x) B-child))
     child-sources (range-tree-child-list P a bs)
and exact-split-range-costed-bmssp-child-sources-below-trivial:
  exact-split-range-costed-bmssp Δ M-of t h k cap l d S B d' B' U c ⟹ True
proof (induction rule:
            exact-split-range-costed-partition-loop-state-exact-split-range-costed-
bmssp.inducts)
  case (Exact-Split-Range-State-Done D a B d' P Δ M-of t h k cap l d)
  show ?case
    by (intro exI[of - []]) simp
next
  case (Exact-Split-Range-State-Stop a B d' P k cap Δ M-of t h l d D)
  show ?case
    by (intro exI[of - []]) simp
next
  case (Exact-Split-Range-State-Step D M-of l Bmax S-pull beta D-pull B d
      P a b B' d' k cap Δ t h d-child U-child c-child edge-batch
      source-batch batch D-next c-pull c-edges c-sources betas bs Us-tail
      U-tail c-tail child-costs-tail c)
  obtain child-sources-tail where len-tail:
      length child-sources-tail = length bs
    and tail:
      list-all2
       (λS-child U-child. ∃ c-child B-child d-child B-child'.
         exact-split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
           d-child B-child' U-child c-child ∧
         bmssp-pre-full d S-child B-child ∧
         (∀ x∈S-child. reachable s x) ∧
         card S-child ≤ M-of l ∧
         (∀ x∈S-child. below-bound (d x) B-child))
       child-sources-tail (range-tree-child-list P b bs)
    using Exact-Split-Range-State-Step.IH by blast
  have card-pull: card S-pull ≤ M-of l
    using Exact-Split-Range-State-Step
    unfolding pull-separates-def by blast
  have below-pull: ∀ x∈S-pull. below-bound (d x) (Fin beta)
  proof
    fix x
    assume xS: x ∈ S-pull
    have S-pull = split-below d P beta
      using Exact-Split-Range-State-Step by blast
    then show below-bound (d x) (Fin beta)
      using xS unfolding split-below-def by auto
  qed
  let ?child-sources = S-pull # child-sources-tail
```

```
  have head:
    ∃ c-child' B-child d-child' B-child'.
      exact-split-range-costed-bmssp Δ M-of t h k cap l d S-pull B-child
        d-child' B-child' U-child c-child' ∧
      bmssp-pre-full d S-pull B-child ∧
      (∀ x∈S-pull. reachable s x) ∧
      card S-pull ≤ M-of l ∧
      (∀ x∈S-pull. below-bound (d x) B-child)
    using Exact-Split-Range-State-Step card-pull below-pull
    by blast
  show ?case
    by (intro exI[of - ?child-sources] conjI)
      (use len-tail head tail Exact-Split-Range-State-Step in simp-all)
next
  case (Exact-Split-Range-Cost-Base S x Δ M-of t h k cap d B)
  show ?case
    by simp
next
  case (Exact-Split-Range-Cost-Step D k cap d S B c-insert t Δ M-of h l
      d' a betas bs B' Us U-loop c-loop child-costs-loop U c)
  show ?case
    by simp
qed
```

The next cost bounds combine the extracted child sources with the explicit edge and source batch costs stored in the relation. First the local loop cost is bounded by the sum of child costs, source cardinalities, and outgoing-edge charges over the child ranges. Then the range-chain lemmas replace those list sums by the corresponding counts for the parent loop output.

```
theorem exact-split-range-costed-partition-loop-state-cost-bound-by-child-sources-
below:
  exact-split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
      betas bs B' Us U c child-costs ⟹
   ∃ child-sources.
    length child-sources = length bs ∧
    list-all2
      (λS-child U-child. ∃ c-child B-child d-child B-child'.
        exact-split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
          d-child B-child' U-child c-child ∧
        bmssp-pre-full d S-child B-child ∧
        (∀ x∈S-child. reachable s x) ∧
        card S-child ≤ M-of l ∧
        (∀ x∈S-child. below-bound (d x) B-child))
      child-sources (range-tree-child-list P a bs) ∧
    c ≤ sum-list child-costs + sum-list (map card child-sources) +
      t * sum-list
        (map (λX. card (outgoing-edges X)) (range-tree-child-list P a bs)) +
      h * sum-list (map card child-sources)
and exact-split-range-costed-bmssp-cost-child-sources-below-trivial:
```

exact-split-range-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c* ⟹ *True*
proof (*induction rule*:
exact-split-range-costed-partition-loop-state-exact-split-range-costed-bmssp.inducts)
case (*Exact-Split-Range-State-Done D a B d′ P* Δ *M-of t h k cap l d*)
show *?case*
by (*intro exI*[*of* - []]) *simp*
next
case (*Exact-Split-Range-State-Stop a B d′ P k cap* Δ *M-of t h l d D*)
show *?case*
by (*intro exI*[*of* - []]) *simp*
next
case (*Exact-Split-Range-State-Step D M-of l Bmax S-pull beta D-pull B d P a b B′ d′ k cap* Δ *t h d-child U-child c-child edge-batch source-batch batch D-next c-pull c-edges c-sources betas bs Us-tail U-tail c-tail child-costs-tail c*)
obtain *child-sources-tail* **where** *len-tail*:
length child-sources-tail = *length bs*
and *sources-tail*:
list-all2
(λ*S-child U-child*. ∃ *c-child B-child d-child B-child′*.
exact-split-range-costed-bmssp Δ *M-of t h k cap l d S-child B-child d-child B-child′ U-child c-child* ∧
bmssp-pre-full d S-child B-child ∧
(∀ *x*∈*S-child*. *reachable s x*) ∧
card S-child ≤ *M-of l* ∧
(∀ *x*∈*S-child*. *below-bound* (*d x*) *B-child*))
child-sources-tail (*range-tree-child-list P b bs*)
and *tail-cost*:
c-tail ≤
sum-list child-costs-tail + *sum-list* (*map card child-sources-tail*) +
t ∗ *sum-list*
(*map* (λ*X*. *card* (*outgoing-edges X*)) (*range-tree-child-list P b bs*)) +
h ∗ *sum-list* (*map card child-sources-tail*)
using *Exact-Split-Range-State-Step.IH* **by** *blast*
have *card-pull*: *card S-pull* ≤ *M-of l*
using *Exact-Split-Range-State-Step*
unfolding *pull-separates-def* **by** *blast*
have *below-pull*: ∀ *x*∈*S-pull*. *below-bound* (*d x*) (*Fin beta*)
proof
fix *x*
assume *xS*: *x* ∈ *S-pull*
have *S-pull* = *split-below d P beta*
using *Exact-Split-Range-State-Step* **by** *blast*
then show *below-bound* (*d x*) (*Fin beta*)
using *xS* **unfolding** *split-below-def* **by** *auto*
qed
have *pull-cost*: *c-pull* ≤ *card S-pull*
using *Exact-Split-Range-State-Step*

unfolding *partition-pull-cost-bound-def* **by** *blast*
have *batch-cost*:
c-edges + *c-sources* ≤
t ∗ *card* (*outgoing-edges U-child*) + *h* ∗ *card S-pull*
by (*rule range-costed-split-batch-cost-le-card*[
where *d* = *d* **and** *S* = *S-pull* **and** *d-child* = *d-child* **and** *b* = *b*
and *beta* = *beta*])
(*use Exact-Split-Range-State-Step* **in** *auto*)
let *?child-sources* = *S-pull* # *child-sources-tail*
have *len*: *length ?child-sources* = *length* (*b* # *bs*)
using *len-tail* **by** *simp*
have *head-source*:
∃ *c-child′ B-child d-child′ B-child′*.
exact-split-range-costed-bmssp Δ *M-of t h k cap l d S-pull B-child*
d-child′ B-child′ U-child c-child′ ∧
bmssp-pre-full d S-pull B-child ∧
(∀ *x*∈*S-pull*. *reachable s x*) ∧
card S-pull ≤ *M-of l* ∧
(∀ *x*∈*S-pull*. *below-bound* (*d x*) *B-child*)
using *Exact-Split-Range-State-Step card-pull below-pull* **by** *blast*
have *sources*:
list-all2
(λ*S-child U-child*. ∃ *c-child B-child d-child B-child′*.
exact-split-range-costed-bmssp Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child ∧
bmssp-pre-full d S-child B-child ∧
(∀ *x*∈*S-child*. *reachable s x*) ∧
card S-child ≤ *M-of l* ∧
(∀ *x*∈*S-child*. *below-bound* (*d x*) *B-child*))
?child-sources (*range-tree-child-list P a* (*b* # *bs*))
using *head-source sources-tail Exact-Split-Range-State-Step* **by** *simp*
have *cost*:
c ≤ *sum-list* (*c-child* # *child-costs-tail*) +
sum-list (*map card ?child-sources*) +
t ∗ *sum-list*
(*map* (λ*X*. *card* (*outgoing-edges X*))
(*range-tree-child-list P a* (*b* # *bs*))) +
h ∗ *sum-list* (*map card ?child-sources*)
proof −
have *c-eq*: *c* = *c-pull* + *c-edges* + *c-sources* + *c-child* + *c-tail*
using *Exact-Split-Range-State-Step* **by** *blast*
have *c-pull* + *c-edges* + *c-sources* + *c-child* + *c-tail* ≤
card S-pull +
(*t* ∗ *card* (*outgoing-edges U-child*) + *h* ∗ *card S-pull*) +
c-child +
(*sum-list child-costs-tail* + *sum-list* (*map card child-sources-tail*) +
t ∗ *sum-list*
(*map* (λ*X*. *card* (*outgoing-edges X*)) (*range-tree-child-list P b bs*)) +
h ∗ *sum-list* (*map card child-sources-tail*))

```
    using pull-cost batch-cost tail-cost by linarith
  also have ... =
      sum-list (c-child # child-costs-tail) +
      sum-list (map card ?child-sources) +
      t * (card (outgoing-edges U-child) +
        sum-list
          (map (λX. card (outgoing-edges X)) (range-tree-child-list P b bs))) +
      h * sum-list (map card ?child-sources)
    by (simp add: algebra-simps)
  also have ... =
      sum-list (c-child # child-costs-tail) +
      sum-list (map card ?child-sources) +
      t * sum-list
        (map (λX. card (outgoing-edges X))
          (range-tree-child-list P a (b # bs))) +
      h * sum-list (map card ?child-sources)
    using Exact-Split-Range-State-Step by simp
  finally show ?thesis
    using c-eq by simp
 qed
 show ?case
  by (intro exI[of - ?child-sources] conjI)
    (use len sources cost in simp-all)
next
 case (Exact-Split-Range-Cost-Base S x Δ M-of t h k cap d B)
 show ?case
  by simp
next
 case (Exact-Split-Range-Cost-Step D k cap d S B c-insert t Δ M-of h l
    d' a betas bs B' Us U-loop c-loop child-costs-loop U c)
 show ?case
  by simp
qed

theorem exact-split-range-costed-partition-loop-state-cost-bound-by-child-sources-
and-edge-ranges:
 exact-split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
    betas bs B' Us U c child-costs ⟹
  sound-label d ⟹
  ∃ child-sources.
   length child-sources = length bs ∧
   list-all2
    (λS-child U-child. ∃ c-child B-child d-child B-child'.
      exact-split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
        d-child B-child' U-child c-child ∧
      bmssp-pre-full d S-child B-child ∧
      (∀ x∈S-child. reachable s x) ∧
      card S-child ≤ M-of l ∧
      (∀ x∈S-child. below-bound (d x) B-child))
```

```
      child-sources (range-tree-child-list P a bs) ∧
    c ≤ sum-list child-costs + sum-list (map card child-sources) +
      t * sum-list
        (map card (range-tree-child-edge-range-list P a betas bs)) +
      h * sum-list (map card child-sources)
and exact-split-range-costed-bmssp-cost-child-sources-and-edge-ranges-trivial:
  exact-split-range-costed-bmssp Δ M-of t h k cap l d S B d' B' U c ⟹ True
proof (induction rule:
            exact-split-range-costed-partition-loop-state-exact-split-range-costed-
bmssp.inducts)
  case (Exact-Split-Range-State-Done D a B d' P Δ M-of t h k cap l d)
  show ?case
    by (intro exI[of - []]) simp
next
  case (Exact-Split-Range-State-Stop a B d' P k cap Δ M-of t h l d D)
  show ?case
    by (intro exI[of - []]) simp
next
  case (Exact-Split-Range-State-Step D M-of l Bmax S-pull beta D-pull B d
      P a b B' d' k cap Δ t h d-child U-child c-child edge-batch
      source-batch batch D-next c-pull c-edges c-sources betas bs Us-tail
      U-tail c-tail child-costs-tail c)
  obtain child-sources-tail where len-tail:
      length child-sources-tail = length bs
    and sources-tail:
      list-all2
        (λS-child U-child. ∃ c-child B-child d-child B-child'.
          exact-split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
            d-child B-child' U-child c-child ∧
          bmssp-pre-full d S-child B-child ∧
          (∀ x∈S-child. reachable s x) ∧
          card S-child ≤ M-of l ∧
          (∀ x∈S-child. below-bound (d x) B-child))
        child-sources-tail (range-tree-child-list P b bs)
    and tail-cost:
      c-tail ≤
        sum-list child-costs-tail + sum-list (map card child-sources-tail) +
        t * sum-list
          (map card (range-tree-child-edge-range-list P b betas bs)) +
        h * sum-list (map card child-sources-tail)
    using Exact-Split-Range-State-Step.IH
      Exact-Split-Range-State-Step.prems by blast
  have card-pull: card S-pull ≤ M-of l
    using Exact-Split-Range-State-Step
    unfolding pull-separates-def by blast
  have below-pull: ∀ x∈S-pull. below-bound (d x) (Fin beta)
  proof
    fix x
    assume xS: x ∈ S-pull
```

have *S-pull* = *split-below d P beta*
using *Exact-Split-Range-State-Step* **by** *blast*
then show *below-bound* (*d x*) (*Fin beta*)
using *xS* **unfolding** *split-below-def* **by** *auto*
qed
have *pull-cost*: *c-pull* ≤ *card S-pull*
using *Exact-Split-Range-State-Step*
unfolding *partition-pull-cost-bound-def* **by** *blast*
have *child-run*:
exact-split-range-costed-bmssp Δ *M-of t h k cap l d S-pull* (*Fin beta*)
d-child (*Fin b*) *U-child c-child*
using *Exact-Split-Range-State-Step* **by** *blast*
have *child-pre*: *bmssp-pre-full d S-pull* (*Fin beta*)
using *Exact-Split-Range-State-Step* **by** *blast*
have *child-reaches*: ⋀*x*. *x* ∈ *S-pull* ⟹ *reachable s x*
using *Exact-Split-Range-State-Step* **by** *blast*
have *child-post*:
bmssp-post-full d S-pull (*Fin beta*) *d-child* (*Fin b*) *U-child*
by (*rule exact-split-range-costed-bmssp-correct*
[*OF child-run Exact-Split-Range-State-Step.prems child-pre child-reaches*])
have *child-complete*: *complete-on d-child U-child*
using *child-post* **unfolding** *bmssp-post-full-def* **by** *blast*
have *U-child-eq*: *U-child* = *range-tree P a* (*Fin b*)
using *Exact-Split-Range-State-Step* **by** *blast*
have *U-child-reaches*: ⋀*u*. *u* ∈ *U-child* ⟹ *reachable s u*
unfolding *U-child-eq range-tree-def tree-set-def tree-path-def* **by** *blast*
have *batch-cost*:
c-edges + *c-sources* ≤
t ∗ *card* (*outgoing-edges-range U-child b* (*Fin beta*)) +
h ∗ *card S-pull*
by (*rule range-costed-split-batch-cost-le-range-card*[
where *d* = *d* **and** *S* = *S-pull* **and** *d-child* = *d-child* **and** *b* = *b*
and *beta* = *beta*])
(*use Exact-Split-Range-State-Step child-complete U-child-reaches* **in** *auto*)
let *?child-sources* = *S-pull* # *child-sources-tail*
have *len*: *length ?child-sources* = *length* (*b* # *bs*)
using *len-tail* **by** *simp*
have *head-source*:
∃ *c-child′ B-child d-child′ B-child′*.
exact-split-range-costed-bmssp Δ *M-of t h k cap l d S-pull B-child*
d-child′ B-child′ U-child c-child′ ∧
bmssp-pre-full d S-pull B-child ∧
(∀ *x*∈*S-pull*. *reachable s x*) ∧
card S-pull ≤ *M-of l* ∧
(∀ *x*∈*S-pull*. *below-bound* (*d x*) *B-child*)
using *Exact-Split-Range-State-Step card-pull below-pull* **by** *blast*
have *sources*:
list-all2
(λ*S-child U-child*. ∃ *c-child B-child d-child B-child′*.

```
      exact-split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
        d-child B-child' U-child c-child ∧
      bmssp-pre-full d S-child B-child ∧
      (∀ x∈S-child. reachable s x) ∧
      card S-child ≤ M-of l ∧
      (∀ x∈S-child. below-bound (d x) B-child))
    ?child-sources (range-tree-child-list P a (b # bs))
  using head-source sources-tail Exact-Split-Range-State-Step by simp
 have cost:
  c ≤ sum-list (c-child # child-costs-tail) +
    sum-list (map card ?child-sources) +
    t * sum-list
      (map card (range-tree-child-edge-range-list P a (beta # betas) (b # bs))) +
    h * sum-list (map card ?child-sources)
 proof −
  have c-eq: c = c-pull + c-edges + c-sources + c-child + c-tail
    using Exact-Split-Range-State-Step by blast
  have c-pull + c-edges + c-sources + c-child + c-tail ≤
      card S-pull +
      (t * card (outgoing-edges-range U-child b (Fin beta)) +
        h * card S-pull) +
      c-child +
      (sum-list child-costs-tail + sum-list (map card child-sources-tail) +
        t * sum-list
          (map card (range-tree-child-edge-range-list P b betas bs)) +
        h * sum-list (map card child-sources-tail))
    using pull-cost batch-cost tail-cost by linarith
  also have ... =
      sum-list (c-child # child-costs-tail) +
      sum-list (map card ?child-sources) +
      t * (card (outgoing-edges-range U-child b (Fin beta)) +
        sum-list
          (map card (range-tree-child-edge-range-list P b betas bs))) +
      h * sum-list (map card ?child-sources)
    by (simp add: algebra-simps)
  also have ... =
      sum-list (c-child # child-costs-tail) +
      sum-list (map card ?child-sources) +
      t * sum-list
        (map card (range-tree-child-edge-range-list P a (beta # betas) (b # bs))) +
      h * sum-list (map card ?child-sources)
    using U-child-eq by simp
  finally show ?thesis
    using c-eq by simp
 qed
 show ?case
  by (intro exI[of - ?child-sources] conjI)
    (use len sources cost in simp-all)
next
```

case (*Exact-Split-Range-Cost-Base S x Δ M-of t h k cap d B*)
show *?case*
by *simp*
next
case (*Exact-Split-Range-Cost-Step D k cap d S B c-insert t Δ M-of h l d′ a betas bs B′ Us U-loop c-loop child-costs-loop U c*)
show *?case*
by *simp*
qed

theorem *exact-split-range-costed-partition-loop-state-cost-from-child-source-and-edge-range-bounds*:
assumes *run*:
exact-split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d′ D a betas bs B′ Us U c child-costs
and *sound*: *sound-label d*
and *child-cost-bounds*:
list-all2 (λ*c-child U-child*.
c-child ≤ *level-range-cost-bound A R L U-child*)
child-costs (*range-tree-child-list P a bs*)
and *source-progress*:
⋀*S-child B-child d-child B-child′ U-child c-child*.
⟦*exact-split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child d-child B-child′ U-child c-child*;
bmssp-pre-full d S-child B-child;
⋀*x*. *x* ∈ *S-child* ⟹ *reachable s x*;
card S-child ≤ *M-of l*;
⋀*x*. *x* ∈ *S-child* ⟹ *below-bound* (*d x*) *B-child*⟧
⟹ *card S-child* ≤ *card U-child*
shows *c* ≤
A ∗ *L* ∗ *card* (*range-tree-chain P a bs B′*) +
R ∗ *card* (*outgoing-edges* (*range-tree-chain P a bs B′*)) +
t ∗ *sum-list*
(*map card* (*range-tree-child-edge-range-list P a betas bs*)) +
(*Suc h*) ∗ *card* (*range-tree-chain P a bs B′*)
proof −
obtain *child-sources* **where** *len-sources*:
length child-sources = *length bs*
and *sources*:
list-all2
(λ*S-child U-child*. ∃ *c-child B-child d-child B-child′*.
exact-split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child d-child B-child′ U-child c-child ∧
bmssp-pre-full d S-child B-child ∧
(∀ *x*∈*S-child*. *reachable s x*) ∧
card S-child ≤ *M-of l* ∧
(∀ *x*∈*S-child*. *below-bound* (*d x*) *B-child*))
child-sources (*range-tree-child-list P a bs*)
and *cost*:

```
      c ≤ sum-list child-costs + sum-list (map card child-sources) +
        t * sum-list
          (map card (range-tree-child-edge-range-list P a betas bs)) +
        h * sum-list (map card child-sources)
    using exact-split-range-costed-partition-loop-state-cost-bound-by-child-sources-
and-edge-ranges
      [OF run sound] by blast
  have mono: nondecreasing-from a bs
  proof −
    have split-run:
      split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d′ D a
        betas bs B′ Us U c child-costs
      using exact-split-range-costed-refines-split-range-costed(1)[OF run] .
    then show ?thesis
      by (rule split-range-costed-partition-loop-state-mono)
  qed
  have source-dom:
    list-all2 (λS-child U-child. card S-child ≤ card U-child)
      child-sources (range-tree-child-list P a bs)
    using sources
  proof (induction rule: list-all2-induct)
    case Nil
    then show ?case by simp
  next
    case (Cons S-child child-sources U-child child-sets)
    obtain c-child B-child d-child B-child′ where child:
        exact-split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
          d-child B-child′ U-child c-child
      and child-pre: bmssp-pre-full d S-child B-child
      and child-reaches: ∀ x∈S-child. reachable s x
      and child-card: card S-child ≤ M-of l
      and child-below: ∀ x∈S-child. below-bound (d x) B-child
      using Cons.hyps by blast
    have head: card S-child ≤ card U-child
      by (rule source-progress[OF child child-pre - child-card])
        (use child-reaches child-below in blast)+
    show ?case
      using head Cons.IH Cons.hyps by simp
  qed
  have source-sum:
    sum-list (map card child-sources) ≤
      card (range-tree-chain P a bs B′)
    by (rule sum-card-dominated-by-range-tree-child-list-le-chain
      [OF mono source-dom])
  have child-sum:
    sum-list child-costs ≤
      A * L * card (range-tree-chain P a bs B′) +
      R * card (outgoing-edges (range-tree-chain P a bs B′))
    by (rule child-costs-le-level-range-child-list-bound
```

```
    [OF mono child-cost-bounds])
  have c ≤
      A * L * card (range-tree-chain P a bs B') +
      R * card (outgoing-edges (range-tree-chain P a bs B')) +
      t * sum-list
        (map card (range-tree-child-edge-range-list P a betas bs)) +
      sum-list (map card child-sources) +
      h * sum-list (map card child-sources)
    using cost child-sum by linarith
  also have ... ≤
      A * L * card (range-tree-chain P a bs B') +
      R * card (outgoing-edges (range-tree-chain P a bs B')) +
      t * sum-list
        (map card (range-tree-child-edge-range-list P a betas bs)) +
      (Suc h) * card (range-tree-chain P a bs B')
  proof −
    have source-part:
      sum-list (map card child-sources) +
        h * sum-list (map card child-sources) ≤
       Suc h * card (range-tree-chain P a bs B')
    proof −
      have h-part:
        h * sum-list (map card child-sources) ≤
          h * card (range-tree-chain P a bs B')
        using source-sum by simp
      have sum-list (map card child-sources) +
          h * sum-list (map card child-sources) ≤
        card (range-tree-chain P a bs B') +
          h * card (range-tree-chain P a bs B')
        using source-sum h-part by linarith
      then show ?thesis
        by (simp add: algebra-simps)
    qed
    show ?thesis
      using source-part by linarith
  qed
  finally show ?thesis .
qed

corollary exact-split-range-costed-partition-loop-state-cost-from-child-source-and-
edge-range-graph-bound:
  assumes run:
    exact-split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
      betas bs B' Us U c child-costs
    and sound: sound-label d
    and child-cost-bounds:
      list-all2 (λc-child U-child.
        c-child ≤ level-range-cost-bound A R L U-child)
        child-costs (range-tree-child-list P a bs)
```

```
    and source-progress:
      ⋀S-child B-child d-child B-child' U-child c-child.
        ⟦exact-split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
            d-child B-child' U-child c-child;
          bmssp-pre-full d S-child B-child;
          ⋀x. x ∈ S-child ⟹ reachable s x;
          card S-child ≤ M-of l;
          ⋀x. x ∈ S-child ⟹ below-bound (d x) B-child⟧
        ⟹ card S-child ≤ card U-child
  shows c ≤
    A * L * card (range-tree-chain P a bs B') +
    R * card (outgoing-edges (range-tree-chain P a bs B')) +
    t * edge-count +
    (Suc h) * card (range-tree-chain P a bs B')
proof −
  have split-run:
    split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
      betas bs B' Us U c child-costs
    using exact-split-range-costed-refines-split-range-costed(1)[OF run] .
  have mono: nondecreasing-from a bs
    by (rule split-range-costed-partition-loop-state-mono[OF split-run])
  have direct-edges:
    sum-list (map card (range-tree-child-edge-range-list P a betas bs))
      ≤ edge-count
    by (rule sum-card-range-tree-child-edge-range-list-le-edge-count[OF mono])
  have separated:
    c ≤
      A * L * card (range-tree-chain P a bs B') +
      R * card (outgoing-edges (range-tree-chain P a bs B')) +
      t * sum-list
        (map card (range-tree-child-edge-range-list P a betas bs)) +
      (Suc h) * card (range-tree-chain P a bs B')
    by (rule exact-split-range-costed-partition-loop-state-cost-from-child-source-and-
edge-range-bounds
      [OF run sound child-cost-bounds source-progress])
  have t * sum-list (map card (range-tree-child-edge-range-list P a betas bs))
      ≤ t * edge-count
    using direct-edges by simp
  then show ?thesis
    using separated by linarith
qed
```

The final theorem in this theory is the reusable recurrence step for exact split ranges. Given child costs already bounded by *level-range-cost-bound* and a source-progress hypothesis, it proves the parent-loop bound with one additional level of vertex cost and one additional batch coefficient on outgoing edges. This is the shape consumed by the direct-insert and bucketed refinements.

theorem *exact-split-range-costed-partition-loop-state-cost-from-child-and-source-*

bounds:
assumes *run*:
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs
and *child-cost-bounds*:
list-all2 (λ*c-child U-child*.
c-child ≤ *level-range-cost-bound A R L U-child*)
child-costs (*range-tree-child-list P a bs*)
and *source-progress*:
⋀*S-child B-child d-child B-child′ U-child c-child*.
⟦*exact-split-range-costed-bmssp* Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child;
bmssp-pre-full d S-child B-child;
⋀*x*. *x* ∈ *S-child* ⟹ *reachable s x*;
card S-child ≤ *M-of l*;
⋀*x*. *x* ∈ *S-child* ⟹ *below-bound* (*d x*) *B-child*⟧
⟹ *card S-child* ≤ *card U-child*
shows *c* ≤
A ∗ *L* ∗ *card* (*range-tree-chain P a bs B′*) +
(*R* + *t*) ∗ *card* (*outgoing-edges* (*range-tree-chain P a bs B′*)) +
(*Suc h*) ∗ *card* (*range-tree-chain P a bs B′*)
proof −
obtain *child-sources* **where** *len-sources*:
length child-sources = *length bs*
and *sources*:
list-all2
(λ*S-child U-child*. ∃ *c-child B-child d-child B-child′*.
exact-split-range-costed-bmssp Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child ∧
bmssp-pre-full d S-child B-child ∧
(∀ *x*∈*S-child*. *reachable s x*) ∧
card S-child ≤ *M-of l* ∧
(∀ *x*∈*S-child*. *below-bound* (*d x*) *B-child*))
child-sources (*range-tree-child-list P a bs*)
and *cost*:
c ≤ *sum-list child-costs* + *sum-list* (*map card child-sources*) +
t ∗ *sum-list*
(*map* (λ*X*. *card* (*outgoing-edges X*)) (*range-tree-child-list P a bs*)) +
h ∗ *sum-list* (*map card child-sources*)
using *exact-split-range-costed-partition-loop-state-cost-bound-by-child-sources-below*
[*OF run*] **by** *blast*
have *mono*: *nondecreasing-from a bs*
proof −
have *split-run*:
split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs
using *exact-split-range-costed-refines-split-range-costed*(*1*)[*OF run*] **.**
then show *?thesis*

```
    by (rule split-range-costed-partition-loop-state-mono)
qed
have source-dom:
  list-all2 (λS-child U-child. card S-child ≤ card U-child)
    child-sources (range-tree-child-list P a bs)
  using sources
proof (induction rule: list-all2-induct)
  case Nil
  then show ?case by simp
next
  case (Cons S-child child-sources U-child child-sets)
  obtain c-child B-child d-child B-child' where child:
      exact-split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
        d-child B-child' U-child c-child
    and child-pre: bmssp-pre-full d S-child B-child
    and child-reaches: ∀ x∈S-child. reachable s x
    and child-card: card S-child ≤ M-of l
    and child-below: ∀ x∈S-child. below-bound (d x) B-child
    using Cons.hyps by blast
  have head: card S-child ≤ card U-child
    by (rule source-progress[OF child child-pre - child-card])
      (use child-reaches child-below in blast)+
  show ?case
    using head Cons.IH Cons.hyps by simp
qed
have source-sum:
  sum-list (map card child-sources) ≤
    card (range-tree-chain P a bs B')
  by (rule sum-card-dominated-by-range-tree-child-list-le-chain
    [OF mono source-dom])
have child-sum:
  sum-list child-costs ≤
    A * L * card (range-tree-chain P a bs B') +
    R * card (outgoing-edges (range-tree-chain P a bs B'))
  by (rule child-costs-le-level-range-child-list-bound
    [OF mono child-cost-bounds])
have edge-sum:
  sum-list
    (map (λX. card (outgoing-edges X)) (range-tree-child-list P a bs)) ≤
   card (outgoing-edges (range-tree-chain P a bs B'))
  by (rule card-outgoing-edges-range-tree-child-list-le-chain[OF mono])
have c ≤
    A * L * card (range-tree-chain P a bs B') +
    R * card (outgoing-edges (range-tree-chain P a bs B')) +
    sum-list (map card child-sources) +
    t * card (outgoing-edges (range-tree-chain P a bs B')) +
    h * sum-list (map card child-sources)
proof −
  have edge-part:
```

```
    t * sum-list
        (map (λX. card (outgoing-edges X)) (range-tree-child-list P a bs))
     ≤ t * card (outgoing-edges (range-tree-chain P a bs B'))
    using edge-sum by simp
  have sum-list child-costs + sum-list (map card child-sources) +
      t * sum-list
        (map (λX. card (outgoing-edges X)) (range-tree-child-list P a bs)) +
      h * sum-list (map card child-sources) ≤
      A * L * card (range-tree-chain P a bs B') +
      R * card (outgoing-edges (range-tree-chain P a bs B')) +
      sum-list (map card child-sources) +
      t * card (outgoing-edges (range-tree-chain P a bs B')) +
      h * sum-list (map card child-sources)
    using child-sum edge-part by linarith
  then show ?thesis
    using cost by linarith
qed
also have ... ≤
    A * L * card (range-tree-chain P a bs B') +
    R * card (outgoing-edges (range-tree-chain P a bs B')) +
    t * card (outgoing-edges (range-tree-chain P a bs B')) +
    (Suc h) * card (range-tree-chain P a bs B')
proof −
  have source-part:
    sum-list (map card child-sources) +
      h * sum-list (map card child-sources) ≤
     Suc h * card (range-tree-chain P a bs B')
  proof −
    have h-part:
      h * sum-list (map card child-sources) ≤
        h * card (range-tree-chain P a bs B')
      using source-sum by simp
    have sum-list (map card child-sources) +
        h * sum-list (map card child-sources) ≤
      card (range-tree-chain P a bs B') +
        h * card (range-tree-chain P a bs B')
      using source-sum h-part by linarith
    then show ?thesis
      by (simp add: algebra-simps)
  qed
  show ?thesis
    using source-part by linarith
qed
also have ... =
    A * L * card (range-tree-chain P a bs B') +
    (R + t) * card (outgoing-edges (range-tree-chain P a bs B')) +
    (Suc h) * card (range-tree-chain P a bs B')
  by (simp add: algebra-simps)
finally show ?thesis .
```

qed

end

end
theory *BMSSP-Direct-Insert-Costed*
imports *BMSSP-Exact-Range-Costed*
begin

32 Direct-Insert Costed Runs

This theory refines the range-costed BMSSP run with the final accounting distinction needed by the bucketed partition implementation. The earlier split-range relation separates edge-generated batch entries from source-label batch entries. Here the edge-generated entries are split again. Some relaxed edge values belong directly in the current parent range and are paid for by the ordinary insert cost parameter t. Other relaxed values fall below the next child bound and are prepended into the lower block; those are paid for by the batch parameter h.

The distinction mirrors the bucketed data structure. A direct insert is the operation that searches for a bucket position and may trigger the logarithmic bucket-splitting work. A lower-range prepend is cheaper in the abstract accounting because it is grouped with other items destined for the next pull. Recording the two lists separately lets the later amortised theorem keep the direct-insert term visible instead of hiding it inside one combined batch budget.

The semantic content is intentionally unchanged. The inductive run still produces the same completed range slices as the exact range-costed relation, and the same BMSSP postcondition is recovered by forgetting the extra accounting fields. What changes is the cost expression: each loop step now exposes a direct-edge batch, a lower-edge batch, a source batch, the child recursive cost, and the tail-loop cost.

This is the last local-cost layer before the global recurrence is instantiated. The closing theorems near the end of the file package the separated charges into two forms: an amortised bound that keeps direct range inserts explicit, and a level bound that can feed the headline top-level analysis.

context *unique-shortest-digraph*
begin

inductive *direct-insert-costed-partition-loop-state*
and *direct-insert-costed-bmssp* **where**
Direct-Insert-Costed-State-Done:
keys-of $D = \{\} \Longrightarrow$
bound-le (*Fin* a) $B \Longrightarrow$
complete-on d' (*bound-tree* P B) $\Longrightarrow$

direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d' D a*
[] [] *B* [*range-tree P a B*]
(*bound-tree P* (*Fin a*) $\cup \bigcup$ (*set* [*range-tree P a B*])) *0* []
| *Direct-Insert-Costed-State-Stop*:
bound-le (*Fin a*) *B* $\Longrightarrow$
complete-on d' (*bound-tree P* (*Fin a*)) $\Longrightarrow$
k $*$ *cap* $\leq$ *card* (*bound-tree P* (*Fin a*)) $\Longrightarrow$
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d' D a*
[] [] (*Fin a*) [*range-tree P a* (*Fin a*)]
(*bound-tree P* (*Fin a*) $\cup \bigcup$ (*set* [*range-tree P a* (*Fin a*)])) *0* []
| *Direct-Insert-Costed-State-Step*:
pull-separates D (*M-of l*) *Bmax S-pull beta D-pull* $\Longrightarrow$
bound-le (*Fin beta*) *B* $\Longrightarrow$
bmssp-pre-full d S-pull (*Fin beta*) $\Longrightarrow$
S-pull = *split-below d P beta* $\Longrightarrow$
($\forall$ *x*$\in$*S-pull. reachable s x*) $\Longrightarrow$
a $\leq$ *b* $\Longrightarrow$
bound-le (*Fin a*) *B'* $\Longrightarrow$
complete-on d' (*bound-tree P* (*Fin a*)) $\Longrightarrow$
card (*bound-tree P* (*Fin a*)) $<$ *k* $*$ *cap* $\Longrightarrow$
direct-insert-costed-bmssp Δ *M-of t h k cap l d S-pull* (*Fin beta*)
d-child (*Fin b*) *U-child c-child* $\Longrightarrow$
U-child = *range-tree P a* (*Fin b*) $\Longrightarrow$
complete-preserved d-child d' U-child $\Longrightarrow$
direct-edge-batch = *edge-relaxation-pairs-in-bound d-child U-child beta B* $\Longrightarrow$
lower-edge-batch = *edge-relaxation-pairs-between d-child U-child b beta* $\Longrightarrow$
source-batch = *label-pairs-between d S-pull b beta* $\Longrightarrow$
batch = *direct-edge-batch* @ *lower-edge-batch* @ *source-batch* $\Longrightarrow$
D-next = *batch-min-update D-pull batch* $\Longrightarrow$
partition-pull-cost-bound c-pull S-pull $\Longrightarrow$
partition-batch-cost-bound c-direct t direct-edge-batch $\Longrightarrow$
partition-batch-cost-bound c-lower h lower-edge-batch $\Longrightarrow$
partition-batch-cost-bound c-sources h source-batch $\Longrightarrow$
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d'*
D-next b betas bs B' Us-tail U-tail c-tail child-costs-tail $\Longrightarrow$
c = *c-pull* + *c-direct* + *c-lower* + *c-sources* + *c-child* + *c-tail* $\Longrightarrow$
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d' D a*
(*beta* # *betas*) (*b* # *bs*) *B'*
(*range-tree P a* (*Fin b*) # *Us-tail*)
(*bound-tree P* (*Fin a*) $\cup$
$\bigcup$(*set* (*range-tree P a* (*Fin b*) # *Us-tail*))) *c*
(*c-child* # *child-costs-tail*)
| *Direct-Insert-Costed-Base*:
S = {*x*} $\Longrightarrow$
direct-insert-costed-bmssp Δ *M-of t h k cap 0 d S B*
(λ*v. if v* $\in$ *base-case-vertices k x B then dist s v else d v*)
(*base-case-bound k x B*)
(*base-case-vertices k x B*)
(*base-case-scan-cost* Δ *k x B*)

```
| Direct-Insert-Costed-Step:
    D = label-partition-view
      (find-pivots-label-capped k cap d S B)
      (find-pivots-pivots-capped k cap d S B) ⟹
    partition-initial-insert-cost-bound c-insert t
      (find-pivots-pivots-capped k cap d S B) ⟹
    direct-insert-costed-partition-loop-state Δ M-of t h k cap l
      (find-pivots-label-capped k cap d S B)
      (find-pivots-pivots-capped k cap d S B) B d' D a betas bs B'
      Us U-loop c-loop child-costs-loop ⟹
    complete-on d'
      {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v} ⟹
    U = U-loop ∪
      {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v} ⟹
      c = fp-iter-capped-scan-cost k cap d S S B + c-insert + c-loop ⟹
      direct-insert-costed-bmssp Δ M-of t h k cap (Suc l) d S B d' B' U c
```

The constructors are the same operational skeleton as before, but the step constructor now names three batches. *edge-relaxation-pairs-in-bound* extracts direct edge relaxations in the parent range, *edge-relaxation-pairs-between* extracts lower edge relaxations between the child output bound and the pull bound, and *label-pairs-between* extracts old source labels in the same lower interval.

Keeping these lists in the derivation tree is useful because each has a different combinatorial owner. Direct edge entries are counted by the direct range lists; lower edge entries are counted by child edge-range lists; source entries are counted through child source sets and then by the progress lemma. The proof scripts that follow mostly say that this extra bookkeeping is conservative: it refines the earlier correctness trace without changing which vertices are completed.

```
inductive-cases direct-insert-costed-bmssp-zeroE:
  direct-insert-costed-bmssp Δ M-of t h k cap 0 d S B d' B' U c

inductive-cases direct-insert-costed-bmssp-SucE:
  direct-insert-costed-bmssp Δ M-of t h k cap (Suc l) d S B d' B' U c

lemma direct-insert-costed-partition-loop-state-mono:
  assumes run:
    direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
      betas bs B' Us U c child-costs
  shows nondecreasing-from a bs
  using run by (induction) auto

lemma direct-insert-costed-partition-loop-state-beta-bounds:
  assumes run:
    direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
      betas bs B' Us U c child-costs
  shows bounds-le B betas
```

using *run* **by** (*induction*) *auto*

theorem *direct-insert-costed-partition-loop-state-trace*:
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d' D a*
betas bs B' Us U c child-costs $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d P B $\Longrightarrow$
$(\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*$) \Longrightarrow$
concrete-partition-loop-trace P B a bs d' B' Us U
and *direct-insert-costed-bmssp-correct*:
direct-insert-costed-bmssp Δ *M-of t h k cap l d S B d' B' U c* $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
bmssp-post-full d S B d' B' U
proof (*induction rule*:
direct-insert-costed-partition-loop-state-direct-insert-costed-bmssp.inducts)
case (*Direct-Insert-Costed-State-Done D a B d' P* Δ *M-of t h k cap l d*)
have *a-le*: *bound-le* (*Fin a*) *B*
using *Direct-Insert-Costed-State-Done* **by** *blast*
have *complete-B*: *complete-on d'* (*bound-tree P B*)
using *Direct-Insert-Costed-State-Done* **by** *blast*
have *lower*: *complete-on d'* (*bound-tree P* (*Fin a*))
by (*rule complete-on-subset*[*OF complete-B*])
(*rule bound-tree-bound-mono*[*OF a-le*])
have *range-complete*: *complete-on d'* (*range-tree P a B*)
by (*rule complete-on-subset*[*OF complete-B*])
(*rule range-tree-subset-bound-tree*)
have *B-le*: *bound-le B B*
by (*cases B*) *simp-all*
show *?case*
unfolding *concrete-partition-loop-trace-def*
using *B-le a-le lower range-complete* **by** *simp*
next
case (*Direct-Insert-Costed-State-Stop a B d' P k cap* Δ *M-of t h l d D*)
then show *?case*
unfolding *concrete-partition-loop-trace-def complete-on-def* **by** *simp*
next
case *Direct-Insert-Costed-State-Step*
then show *?case*
unfolding *concrete-partition-loop-trace-def bmssp-post-full-def*
complete-preserved-def
by (*auto split*: *bound.splits*)
next
case (*Direct-Insert-Costed-Base S x* Δ *M-of t h k cap d B*)
have *post*:
bmssp-post d S B
(λv. *if v* $\in$ *base-case-vertices k x B then dist s v else d v*)
(*base-case-bound k x B*)

```
    (base-case-vertices k x B)
   using base-case-result-bmssp-post[OF Direct-Insert-Costed-Base.hyps,
     where k = k and B = B and d = d]
   unfolding base-case-result-def by simp
 then show ?case
   by (rule bmssp-post-imp-post-full)
next
 case (Direct-Insert-Costed-Step D k cap d S B c-insert t Δ M-of h l d' a
    betas bs B' Us U-loop c-loop child-costs-loop U c)
 let ?d-fp = find-pivots-label-capped k cap d S B
 let ?P = find-pivots-pivots-capped k cap d S B
 let ?W = {v ∈ bound-tree S B'. ?d-fp v = dist s v}
 have sound-fp: sound-label ?d-fp
   unfolding find-pivots-label-capped-def
   by (rule fp-iter-capped-label-sound[OF Direct-Insert-Costed-Step.prems(1)
     Direct-Insert-Costed-Step.prems(3)])
 have pivot-pre: bmssp-pre-full ?d-fp ?P B
   using find-pivots-capped-establishes-pivot-pre-concrete
     [OF Direct-Insert-Costed-Step.prems(1) Direct-Insert-Costed-Step.prems(2)
       Direct-Insert-Costed-Step.prems(3)] .
 have P-subset-S: ?P ⊆ S
   unfolding find-pivots-pivots-capped-def by auto
 have P-reaches: ⋀x. x ∈ ?P ⟹ reachable s x
   using P-subset-S Direct-Insert-Costed-Step.prems(3) by blast
 have trace:
   concrete-partition-loop-trace ?P B a bs d' B' Us U-loop
   using Direct-Insert-Costed-Step.IH sound-fp pivot-pre P-reaches by blast
 have loop-post: bmssp-post-full ?d-fp ?P B d' B' U-loop
   by (rule concrete-partition-loop-trace-post[OF trace])
 have le: bound-le B' B
   using loop-post unfolding bmssp-post-full-def by blast
 have U-tree: U = bound-tree S B'
 proof −
   have U-loop ∪ ?W = bound-tree S B'
     by (rule concrete-capped-find-pivots-final-tree-assembly
       [OF Direct-Insert-Costed-Step.prems(1) Direct-Insert-Costed-Step.prems(2)
         Direct-Insert-Costed-Step.prems(3) loop-post])
   then show ?thesis
     using Direct-Insert-Costed-Step(6) by simp
 qed
 have loop-complete: complete-on d' U-loop
   using loop-post unfolding bmssp-post-full-def by blast
 have U-complete: complete-on d' U
   using complete-on-Un[OF loop-complete Direct-Insert-Costed-Step(5)]
     Direct-Insert-Costed-Step(6) by simp
 show ?case
   using le U-tree U-complete unfolding bmssp-post-full-def by blast
qed
```

theorem *finite-initial-label-direct-insert-costed-top-level-correct*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *run*:
direct-insert-costed-bmssp Δ *M-of t h k cap l*
finite-initial-label $\{s\}$ *Infinity d' Infinity U c*
shows *sssp-correct d'*
proof −
have *pre*: *bmssp-pre-full finite-initial-label* $\{s\}$ *Infinity*
using *all-reachable finite-initial-label-source-complete*
by (*rule top-bmssp-pre-full*)
have *sound*: *sound-label finite-initial-label*
using *finite-initial-label-sound*[*OF all-reachable*] **.**
have *S-reaches*: $\bigwedge x.\ x \in \{s\} \Longrightarrow$ *reachable s x*
using *all-reachable source-in-V* **by** *blast*
have *post-full*:
bmssp-post-full finite-initial-label $\{s\}$ *Infinity d' Infinity U*
by (*rule direct-insert-costed-bmssp-correct*[*OF run sound pre S-reaches*])
have *post*: *bmssp-post finite-initial-label* $\{s\}$ *Infinity d' Infinity U*
using *bmssp-post-full-imp-post*[*OF post-full*] **.**
then have *U-V*: $U = V$
using *bound-tree-source-infinity*[*OF all-reachable*]
unfolding *bmssp-post-def* **by** *auto*
have *complete*: *complete-on d' U*
using *post* **unfolding** *bmssp-post-def* **by** *auto*
then show *?thesis*
using *U-V* **unfolding** *complete-on-def sssp-correct-def* **by** *auto*
qed

theorem *finite-initial-label-direct-insert-costed-top-level-reachable-correct*:
assumes *run*:
direct-insert-costed-bmssp Δ *M-of t h k cap l*
finite-initial-label $\{s\}$ *Infinity d' Infinity U c*
shows *sssp-correct d'*
proof −
have *pre*: *bmssp-pre-full finite-initial-label* $\{s\}$ *Infinity*
using *finite-initial-label-source-complete*
by (*rule top-bmssp-pre-full-reachable*)
have *sound*: *sound-label finite-initial-label*
by (*rule finite-initial-label-sound-reachable*)
have *S-reaches*: $\bigwedge x.\ x \in \{s\} \Longrightarrow$ *reachable s x*
using *reachable-refl*[*OF source-in-V*] **by** *blast*
have *post-full*:
bmssp-post-full finite-initial-label $\{s\}$ *Infinity d' Infinity U*
by (*rule direct-insert-costed-bmssp-correct*[*OF run sound pre S-reaches*])
have *post*: *bmssp-post finite-initial-label* $\{s\}$ *Infinity d' Infinity U*
using *bmssp-post-full-imp-post*[*OF post-full*] **.**
then show *?thesis*
by (*rule successful-root-bmssp-sssp-correct*)
qed

The correctness theorem ⟦*direct-insert-costed-bmssp ?Δ ?M-of ?t ?h ?k ?cap ?l ?d ?S ?B ?d′ ?B′ ?U ?c*; *sound-label ?d*; *bmssp-pre-full ?d ?S ?B*; ⋀*x*. *x* ∈ *?S* ⟹ *reachable s x*⟧ ⟹ *bmssp-post-full ?d ?S ?B ?d′ ?B′ ?U* is the semantic projection of the direct-insert relation. Once projected, the top-level corollaries are the familiar root BMSSP statements: from the finite initial label at the source and the infinite output bound, the returned label function is a complete SSSP solution.

The next group of lemmas supplies the progress side conditions used by the cost recurrence. They show that successful BMSSP calls either return the original bound or have accumulated enough completed vertices to meet the threshold, and that a source set whose labels are below the output bound is dominated by the output tree. These facts turn source-batch costs into vertex-count terms.

theorem *direct-insert-costed-bmssp-source-card-le-if-label-below-output*:
  assumes *run*:
    direct-insert-costed-bmssp Δ M-of t h k cap l d S B d′ B′ U c
   and *sound*: *sound-label d*
   and *pre*: *bmssp-pre-full d S B*
   and *S-reaches*: ⋀*x*. *x* ∈ *S* ⟹ *reachable s x*
   and *complete*: *complete-on d S*
   and *below*: ⋀*x*. *x* ∈ *S* ⟹ *below-bound* (*d x*) *B′*
  shows *card S* ≤ *card U*
proof −
  have *post*: *bmssp-post-full d S B d′ B′ U*
    by (*rule direct-insert-costed-bmssp-correct*[*OF run sound pre S-reaches*])
  show *?thesis*
    by (*rule bmssp-post-full-source-card-le-if-label-below-output*
      [*OF post complete S-reaches below*])
qed

theorem *direct-insert-costed-bmssp-source-card-le-if-sound-label-below-output*:
  assumes *run*:
    direct-insert-costed-bmssp Δ M-of t h k cap l d S B d′ B′ U c
   and *sound*: *sound-label d*
   and *pre*: *bmssp-pre-full d S B*
   and *S-reaches*: ⋀*x*. *x* ∈ *S* ⟹ *reachable s x*
   and *below*: ⋀*x*. *x* ∈ *S* ⟹ *below-bound* (*d x*) *B′*
  shows *card S* ≤ *card U*
proof −
  have *post*: *bmssp-post-full d S B d′ B′ U*
    by (*rule direct-insert-costed-bmssp-correct*[*OF run sound pre S-reaches*])
  have *U-eq*: *U* = *bound-tree S B′*
    using *post* **unfolding** *bmssp-post-full-def* **by** *blast*
  have *S-subset-U*: *S* ⊆ *U*
  proof
    fix *x*
    assume *xS*: *x* ∈ *S*

```
    have S-subset: S ⊆ V
      using pre unfolding bmssp-pre-full-def by blast
    have xV: x ∈ V
      using S-subset xS by blast
    have reach-x: reachable s x
      by (rule S-reaches[OF xS])
    have dist-le: dist s x ≤ d x
      using sound xV reach-x unfolding sound-label-def by blast
    have below-dist: below-bound (dist s x) B'
      using below-bound-le-trans[OF dist-le below[OF xS]] .
    show x ∈ U
      unfolding U-eq
      by (rule source-in-own-bound-tree[OF xS reach-x below-dist])
  qed
  have finite-U: finite U
    unfolding U-eq bound-tree-def using finite-V by auto
  show ?thesis
    by (rule card-mono[OF finite-U S-subset-U])
qed

theorem direct-insert-costed-partition-loop-state-success-or-threshold:
  direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
      betas bs B' Us U c child-costs ⟹
   sound-label d ⟹
   bmssp-pre-full d P B ⟹
   (⋀x. x ∈ P ⟹ reachable s x) ⟹
   B' = B ∨ k * cap ≤ card U
and direct-insert-costed-bmssp-success-or-threshold-trivial:
  direct-insert-costed-bmssp Δ M-of t h k cap l d S B d' B' U c ⟹ True
proof (induction rule:
    direct-insert-costed-partition-loop-state-direct-insert-costed-bmssp.inducts)
  case (Direct-Insert-Costed-State-Done D a B d' P Δ M-of t h k cap l d)
  then show ?case
    by simp
next
  case (Direct-Insert-Costed-State-Stop a B d' P k cap Δ M-of t h l d D)
  then show ?case
    by simp
next
  case (Direct-Insert-Costed-State-Step D M-of l Bmax S-pull beta D-pull B d
      P a b B-out d-out k cap Δ t h d-child U-child c-child
      direct-edge-batch lower-edge-batch source-batch batch D-next c-pull
      c-direct c-lower c-sources betas bs Us-tail U-tail c-tail
      child-costs-tail c)
  have tail:
    B-out = B ∨ k * cap ≤ card U-tail
    using Direct-Insert-Costed-State-Step.IH Direct-Insert-Costed-State-Step.prems
by blast
  then show ?case
```

proof
assume *B-out* = *B*
then show *?thesis*
by *blast*
next
assume *threshold-tail*: *k* ∗ *cap* ≤ *card U-tail*
have *tail-trace*:
concrete-partition-loop-trace P B b bs d-out B-out Us-tail U-tail
by (*rule direct-insert-costed-partition-loop-state-trace*)
(*use Direct-Insert-Costed-State-Step* **in** *blast*)+
have *U-tail-def*:
U-tail = *bound-tree P* (*Fin b*) ∪ ⋃(*set Us-tail*)
using *tail-trace* **unfolding** *concrete-partition-loop-trace-def* **by** *blast*
have *split-b*:
bound-tree P (*Fin b*) =
bound-tree P (*Fin a*) ∪ *range-tree P a* (*Fin b*)
using *bound-tree-split-at*[*of a Fin b P*] *Direct-Insert-Costed-State-Step*
by *simp*
have *U-eq-tail*:
bound-tree P (*Fin a*) ∪
⋃(*set* (*range-tree P a* (*Fin b*) # *Us-tail*)) =
U-tail
using *U-tail-def split-b* **by** *auto*
show *?thesis*
using *threshold-tail U-eq-tail* **by** *simp*
qed
next
case (*Direct-Insert-Costed-Base S x* Δ *M-of t h k cap d B*)
then show *?case*
by *simp*
next
case (*Direct-Insert-Costed-Step D k cap d S B c-insert t* Δ *M-of h l d′ a*
betas bs B′ Us U-loop c-loop child-costs-loop U c)
then show *?case*
by *simp*
qed

theorem *direct-insert-costed-bmssp-Suc-success-or-threshold*:
assumes *run*:
direct-insert-costed-bmssp Δ *M-of t h k cap* (*Suc l*) *d S B d′ B′ U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: ⋀*x*. *x* ∈ *S* ⟹ *reachable s x*
shows *B′* = *B* ∨ *k* ∗ *cap* ≤ *card U*
using *run*
proof *cases*
case (*Direct-Insert-Costed-Step D c-insert a betas bs Us U-loop c-loop child-costs-loop*)
let *?d-fp* = *find-pivots-label-capped k cap d S B*

let *?P = find-pivots-pivots-capped k cap d S B*
let *?W = {v ∈ bound-tree S B′. ?d-fp v = dist s v}*
have *sound-fp*: *sound-label ?d-fp*
unfolding *find-pivots-label-capped-def*
by (*rule fp-iter-capped-label-sound*[*OF sound S-reaches*])
have *pivot-pre*: *bmssp-pre-full ?d-fp ?P B*
using *find-pivots-capped-establishes-pivot-pre-concrete*
[*OF sound pre S-reaches*] **.**
have *P-subset-S*: *?P ⊆ S*
unfolding *find-pivots-pivots-capped-def* **by** *auto*
have *P-reaches*: *⋀x. x ∈ ?P ⟹ reachable s x*
using *P-subset-S S-reaches* **by** *blast*
have *loop-class*:
B′ = B ∨ k ∗ cap ≤ card U-loop
by (*rule direct-insert-costed-partition-loop-state-success-or-threshold*
[*OF Direct-Insert-Costed-Step*(*3*) *sound-fp pivot-pre P-reaches*])
have *loop-trace*:
concrete-partition-loop-trace ?P B a bs d′ B′ Us U-loop
by (*rule direct-insert-costed-partition-loop-state-trace*
[*OF Direct-Insert-Costed-Step*(*3*) *sound-fp pivot-pre P-reaches*])
have *loop-post*: *bmssp-post-full ?d-fp ?P B d′ B′ U-loop*
by (*rule concrete-partition-loop-trace-post*[*OF loop-trace*])
have *finite-U-loop*: *finite U-loop*
using *loop-post* **unfolding** *bmssp-post-full-def* **by** *simp*
show *?thesis*
using *loop-class*
proof
assume *B′ = B*
then show *?thesis*
by *blast*
next
assume *threshold-loop*: *k ∗ cap ≤ card U-loop*
have *U-eq*: *U = U-loop ∪ ?W*
using *Direct-Insert-Costed-Step*(*5*) **by** *simp*
have *finite-U*: *finite U*
unfolding *U-eq* **using** *finite-U-loop* **by** *simp*
have *U-loop-subset*: *U-loop ⊆ U*
unfolding *U-eq* **by** *blast*
have *card U-loop ≤ card U*
by (*rule card-mono*[*OF finite-U U-loop-subset*])
then show *?thesis*
using *threshold-loop* **by** *linarith*
qed
qed

theorem *direct-insert-costed-bmssp-Suc-source-card-le-from-label-below*:
assumes *run*:
direct-insert-costed-bmssp Δ M-of t h k cap (*Suc l*) *d S B d′ B′ U c*
and *sound*: *sound-label d*

```
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and below: ⋀x. x ∈ S ⟹ below-bound (d x) B
    and S-cap: card S ≤ cap
    and k-pos: 0 < k
  shows card S ≤ card U
proof −
  have call-class: B' = B ∨ k * cap ≤ card U
    by (rule direct-insert-costed-bmssp-Suc-success-or-threshold
      [OF run sound pre S-reaches])
  then show ?thesis
  proof
    assume B'-eq: B' = B
    show ?thesis
    proof (rule direct-insert-costed-bmssp-source-card-le-if-sound-label-below-output
        [OF run sound pre S-reaches])
      fix x
      assume x ∈ S
      then show below-bound (d x) B'
        using below B'-eq by simp
    qed
  next
    assume threshold: k * cap ≤ card U
    have cap ≤ k * cap
      using k-pos by (cases k) simp-all
    then show ?thesis
      using S-cap threshold by linarith
  qed
qed
```

The child-cost lemmas extract the recursive calls hidden inside a partition loop. Each child call is paired with the range slice it completed, together with the exact source set pulled for that child and the proof obligations needed to apply the induction hypothesis. The additional *below-bound* $(d\ x)$ *B-child* invariant is what makes source progress available without re-deriving the pull/split equality at every later use.

```
theorem direct-insert-costed-partition-loop-state-child-cost-calls-below:
  direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
      betas bs B' Us U c child-costs ⟹
   length child-costs = length bs ∧
   list-all2
     (λc-child U-child. ∃S-child B-child d-child B-child'.
       direct-insert-costed-bmssp Δ M-of t h k cap l d S-child B-child
         d-child B-child' U-child c-child ∧
       bmssp-pre-full d S-child B-child ∧
       (∀x∈S-child. reachable s x) ∧
       card S-child ≤ M-of l ∧
       (∀x∈S-child. below-bound (d x) B-child))
     child-costs (range-tree-child-list P a bs)
```

and *direct-insert-costed-bmssp-child-cost-calls-below-trivial*:
direct-insert-costed-bmssp Δ *M-of t h k cap l d S B d' B' U c* $\Longrightarrow$ *True*
proof (*induction rule*:
direct-insert-costed-partition-loop-state-direct-insert-costed-bmssp.inducts)
case (*Direct-Insert-Costed-State-Done D a B d' P* Δ *M-of t h k cap l d*)
show *?case*
by *simp*
next
case (*Direct-Insert-Costed-State-Stop a B d' P k cap* Δ *M-of t h l d D*)
show *?case*
by *simp*
next
case (*Direct-Insert-Costed-State-Step D M-of l Bmax S-pull beta D-pull B d*
P a b B' d' k cap Δ *t h d-child U-child c-child*
direct-edge-batch lower-edge-batch source-batch batch D-next c-pull
c-direct c-lower c-sources betas bs Us-tail U-tail c-tail
child-costs-tail c)
have *tail*:
length child-costs-tail = length bs $\wedge$
list-all2
(λ*c-child U-child*. $\exists$ *S-child B-child d-child B-child'*.
direct-insert-costed-bmssp Δ *M-of t h k cap l d S-child B-child*
d-child B-child' U-child c-child $\wedge$
bmssp-pre-full d S-child B-child $\wedge$
($\forall$ *x*$\in$*S-child*. *reachable s x*) $\wedge$
card S-child $\leq$ *M-of l* $\wedge$
($\forall$ *x*$\in$*S-child*. *below-bound* (*d x*) *B-child*))
child-costs-tail (*range-tree-child-list P b bs*)
using *Direct-Insert-Costed-State-Step.IH* **by** *blast*
have *card-pull*: *card S-pull* $\leq$ *M-of l*
using *Direct-Insert-Costed-State-Step*
unfolding *pull-separates-def* **by** *blast*
have *below-pull*: $\forall$ *x*$\in$*S-pull*. *below-bound* (*d x*) (*Fin beta*)
proof
fix *x*
assume *xS*: *x* $\in$ *S-pull*
have *S-pull = split-below d P beta*
using *Direct-Insert-Costed-State-Step* **by** *blast*
then show *below-bound* (*d x*) (*Fin beta*)
using *xS* **unfolding** *split-below-def* **by** *auto*
qed
have *head*:
$\exists$ *S-child B-child d-child' B-child'*.
direct-insert-costed-bmssp Δ *M-of t h k cap l d S-child B-child*
d-child' B-child' U-child c-child $\wedge$
bmssp-pre-full d S-child B-child $\wedge$
($\forall$ *x*$\in$*S-child*. *reachable s x*) $\wedge$
card S-child $\leq$ *M-of l* $\wedge$
($\forall$ *x*$\in$*S-child*. *below-bound* (*d x*) *B-child*)

using *Direct-Insert-Costed-State-Step card-pull below-pull* **by** *blast*
show *?case*
using *tail head Direct-Insert-Costed-State-Step* **by** *simp*
next
case (*Direct-Insert-Costed-Base S x Δ M-of t h k cap d B*)
show *?case*
by *simp*
next
case (*Direct-Insert-Costed-Step D k cap d S B c-insert t Δ M-of h l*
d′ a betas bs B′ Us U-loop c-loop child-costs-loop U c)
show *?case*
by *simp*
qed

theorem *direct-insert-costed-partition-loop-state-child-cost-calls-below-with-bounds*:
direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B d′ D a
betas bs B′ Us U c child-costs ⟹
list-all2
(λ*c-child UB*. *case UB of* (*U-child*, *B-child*) ⇒
∃ *S-child d-child B-child′*.
direct-insert-costed-bmssp Δ M-of t h k cap l d S-child B-child
d-child B-child′ U-child c-child ∧
bmssp-pre-full d S-child B-child ∧
(∀ *x*∈*S-child*. *reachable s x*) ∧
card S-child ≤ *M-of l* ∧
(∀ *x*∈*S-child*. *below-bound* (*d x*) *B-child*))
child-costs (*range-tree-child-bound-pair-list P a betas bs*)
and *direct-insert-costed-bmssp-child-cost-calls-below-with-bounds-trivial*:
direct-insert-costed-bmssp Δ M-of t h k cap l d S B d′ B′ U c ⟹ *True*
proof (*induction rule*:
direct-insert-costed-partition-loop-state-direct-insert-costed-bmssp.inducts)
case (*Direct-Insert-Costed-State-Done D a B d′ P Δ M-of t h k cap l d*)
show *?case*
by *simp*
next
case (*Direct-Insert-Costed-State-Stop a B d′ P k cap Δ M-of t h l d D*)
show *?case*
by *simp*
next
case (*Direct-Insert-Costed-State-Step D M-of l Bmax S-pull beta D-pull B d*
P a b B′ d′ k cap Δ t h d-child U-child c-child
direct-edge-batch lower-edge-batch source-batch batch D-next c-pull
c-direct c-lower c-sources betas bs Us-tail U-tail c-tail
child-costs-tail c)
have *tail*:
list-all2
(λ*c-child UB*. *case UB of* (*U-child*, *B-child*) ⇒
∃ *S-child d-child B-child′*.
direct-insert-costed-bmssp Δ M-of t h k cap l d S-child B-child

```
        d-child B-child' U-child c-child ∧
        bmssp-pre-full d S-child B-child ∧
        (∀ x∈S-child. reachable s x) ∧
        card S-child ≤ M-of l ∧
        (∀ x∈S-child. below-bound (d x) B-child))
    child-costs-tail (range-tree-child-bound-pair-list P b betas bs)
   using Direct-Insert-Costed-State-Step.IH by blast
 have card-pull: card S-pull ≤ M-of l
   using Direct-Insert-Costed-State-Step
   unfolding pull-separates-def by blast
 have below-pull: ∀ x∈S-pull. below-bound (d x) (Fin beta)
 proof
   fix x
   assume xS: x ∈ S-pull
   have S-pull = split-below d P beta
     using Direct-Insert-Costed-State-Step by blast
   then show below-bound (d x) (Fin beta)
     using xS unfolding split-below-def by auto
 qed
 have head:
   ∃ S-child d-child' B-child'.
     direct-insert-costed-bmssp Δ M-of t h k cap l d S-child (Fin beta)
       d-child' B-child' U-child c-child ∧
     bmssp-pre-full d S-child (Fin beta) ∧
     (∀ x∈S-child. reachable s x) ∧
     card S-child ≤ M-of l ∧
     (∀ x∈S-child. below-bound (d x) (Fin beta))
   using Direct-Insert-Costed-State-Step card-pull below-pull by blast
 show ?case
   using tail head Direct-Insert-Costed-State-Step by simp
next
 case (Direct-Insert-Costed-Base S x Δ M-of t h k cap d B)
 show ?case
   by simp
next
 case (Direct-Insert-Costed-Step D k cap d S B c-insert t Δ M-of h l
     d' a betas bs B' Us U-loop c-loop child-costs-loop U c)
 show ?case
   by simp
qed

theorem direct-insert-costed-partition-loop-state-child-cost-bounds:
 assumes run:
   direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
     betas bs B' Us U c child-costs
   and child-bound:
     ⋀c-child U-child S-child B-child d-child B-child'.
       ⟦direct-insert-costed-bmssp Δ M-of t h k cap l d S-child B-child
           d-child B-child' U-child c-child;
```

```
        bmssp-pre-full d S-child B-child;
        ⋀x. x ∈ S-child ⟹ reachable s x;
        card S-child ≤ M-of l;
        ⋀x. x ∈ S-child ⟹ below-bound (d x) B-child⟧
      ⟹ c-child ≤ level-range-cost-bound A R L U-child
  shows list-all2 (λc-child U-child.
      c-child ≤ level-range-cost-bound A R L U-child)
    child-costs (range-tree-child-list P a bs)
proof −
  have calls:
    list-all2
      (λc-child U-child. ∃ S-child B-child d-child B-child'.
        direct-insert-costed-bmssp Δ M-of t h k cap l d S-child B-child
          d-child B-child' U-child c-child ∧
        bmssp-pre-full d S-child B-child ∧
        (∀ x∈S-child. reachable s x) ∧
        card S-child ≤ M-of l ∧
        (∀ x∈S-child. below-bound (d x) B-child))
      child-costs (range-tree-child-list P a bs)
    using direct-insert-costed-partition-loop-state-child-cost-calls-below
      [OF run] by blast
  then show ?thesis
  proof (induction rule: list-all2-induct)
    case Nil
    then show ?case by simp
  next
    case (Cons c-child child-costs U-child child-ranges)
    obtain S-child B-child d-child B-child' where child:
        direct-insert-costed-bmssp Δ M-of t h k cap l d S-child B-child
          d-child B-child' U-child c-child
      and child-pre: bmssp-pre-full d S-child B-child
      and child-reaches: ∀ x∈S-child. reachable s x
      and child-card: card S-child ≤ M-of l
      and child-below: ∀ x∈S-child. below-bound (d x) B-child
      using Cons.hyps by blast
    have head:
      c-child ≤ level-range-cost-bound A R L U-child
      by (rule child-bound[OF child child-pre - child-card])
        (use child-reaches child-below in blast)+
    show ?case
      using head Cons.IH Cons.hyps by simp
  qed
qed

theorem direct-insert-costed-partition-loop-state-child-cost-bounds-with-invariants:
  direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
      betas bs B' Us U c child-costs ⟹
    P ⊆ V ⟹
    k * card P ≤ cap ⟹
```

```
    tree-antichain P ⟹
    (⋀c-child U-child S-child B-child d-child B-child'.
       ⟦direct-insert-costed-bmssp Δ M-of t h k cap l d S-child B-child
           d-child B-child' U-child c-child;
         bmssp-pre-full d S-child B-child;
         ⋀x. x ∈ S-child ⟹ reachable s x;
         card S-child ≤ M-of l;
         ⋀x. x ∈ S-child ⟹ below-bound (d x) B-child;
         k * card S-child ≤ cap;
         tree-antichain S-child⟧
       ⟹ c-child ≤ level-range-cost-bound A R L U-child) ⟹
    list-all2 (λc-child U-child.
      c-child ≤ level-range-cost-bound A R L U-child)
    child-costs (range-tree-child-list P a bs)
and direct-insert-costed-bmssp-child-cost-bounds-with-invariants-trivial:
  direct-insert-costed-bmssp Δ M-of t h k cap l d S B d' B' U c ⟹ True
proof (induction rule:
    direct-insert-costed-partition-loop-state-direct-insert-costed-bmssp.inducts)
  case (Direct-Insert-Costed-State-Done D a B d' P Δ M-of t h k cap l d)
  then show ?case
    by simp
next
  case (Direct-Insert-Costed-State-Stop a B d' P k cap Δ M-of t h l d D)
  then show ?case
    by simp
next
  case (Direct-Insert-Costed-State-Step D M-of l Bmax S-pull beta D-pull B d
      P a b B' d' k cap Δ t h d-child U-child c-child
      direct-edge-batch lower-edge-batch source-batch batch D-next c-pull
      c-direct c-lower c-sources betas bs Us-tail U-tail c-tail
      child-costs-tail c)
  have tail:
    list-all2 (λc-child U-child.
      c-child ≤ level-range-cost-bound A R L U-child)
      child-costs-tail (range-tree-child-list P b bs)
    using Direct-Insert-Costed-State-Step.IH Direct-Insert-Costed-State-Step.prems
    by blast
  have card-pull: card S-pull ≤ M-of l
    using Direct-Insert-Costed-State-Step
    unfolding pull-separates-def by blast
  have below-pull: ⋀x. x ∈ S-pull ⟹ below-bound (d x) (Fin beta)
  proof −
    fix x
    assume xS: x ∈ S-pull
    have S-pull = split-below d P beta
      using Direct-Insert-Costed-State-Step by blast
    then show below-bound (d x) (Fin beta)
      using xS unfolding split-below-def by auto
  qed
```

```
  have pull-k-cap: k * card S-pull ≤ cap
  proof −
    have S-pull = split-below d P beta
      using Direct-Insert-Costed-State-Step by blast
    then show ?thesis
      using split-below-scaled-card-le
        [OF Direct-Insert-Costed-State-Step.prems(1)
          Direct-Insert-Costed-State-Step.prems(2), of d beta]
      by simp
  qed
  have pull-anti: tree-antichain S-pull
  proof −
    have S-pull = split-below d P beta
      using Direct-Insert-Costed-State-Step by blast
    then show ?thesis
      using split-below-tree-antichain
        [OF Direct-Insert-Costed-State-Step.prems(3), of d beta]
      by simp
  qed
  have child-run:
    direct-insert-costed-bmssp Δ M-of t h k cap l d S-pull (Fin beta)
      d-child (Fin b) U-child c-child
    using Direct-Insert-Costed-State-Step by blast
  have pre-pull: bmssp-pre-full d S-pull (Fin beta)
    using Direct-Insert-Costed-State-Step by blast
  have reaches-pull: ⋀x. x ∈ S-pull ⟹ reachable s x
    using Direct-Insert-Costed-State-Step by blast
  have head:
    c-child ≤ level-range-cost-bound A R L U-child
    by (rule Direct-Insert-Costed-State-Step.prems(4)
      [OF child-run pre-pull reaches-pull card-pull below-pull
        pull-k-cap pull-anti])
  show ?case
    using head tail Direct-Insert-Costed-State-Step by simp
next
  case (Direct-Insert-Costed-Base S x Δ M-of t h k cap d B)
  then show ?case
    by simp
next
  case (Direct-Insert-Costed-Step D k cap d S B c-insert t Δ M-of h l
      d′ a betas bs B′ Us U-loop c-loop child-costs-loop U c)
  then show ?case
    by simp
qed

theorem direct-insert-costed-partition-loop-state-child-amortized-cost-bounds-with-
invariants:
  direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B d′ D a
      betas bs B′ Us U c child-costs ⟹
```

```
    P ⊆ V ⟹
    k * card P ≤ cap ⟹
    tree-antichain P ⟹
    (⋀c-child U-child S-child B-child d-child B-child'.
       ⟦direct-insert-costed-bmssp Δ M-of t h k cap l d S-child B-child
          d-child B-child' U-child c-child;
        bmssp-pre-full d S-child B-child;
        ⋀x. x ∈ S-child ⟹ reachable s x;
        card S-child ≤ M-of l;
        ⋀x. x ∈ S-child ⟹ below-bound (d x) B-child;
        k * card S-child ≤ cap;
        tree-antichain S-child⟧
       ⟹ c-child ≤
        bmssp-amortized-cost-bound A R h t q L B-child U-child) ⟹
    list-all2 (λc-child UB. case UB of (U-child, B-child) ⇒
      c-child ≤ bmssp-amortized-cost-bound A R h t q L B-child U-child)
    child-costs (range-tree-child-bound-pair-list P a betas bs)
and direct-insert-costed-bmssp-child-amortized-cost-bounds-with-invariants-trivial:
  direct-insert-costed-bmssp Δ M-of t h k cap l d S B d' B' U c ⟹ True
proof (induction rule:
    direct-insert-costed-partition-loop-state-direct-insert-costed-bmssp.inducts)
  case (Direct-Insert-Costed-State-Done D a B d' P Δ M-of t h k cap l d)
  then show ?case
    by simp
next
  case (Direct-Insert-Costed-State-Stop a B d' P k cap Δ M-of t h l d D)
  then show ?case
    by simp
next
  case (Direct-Insert-Costed-State-Step D M-of l Bmax S-pull beta D-pull B d
      P a b B' d' k cap Δ t h d-child U-child c-child
      direct-edge-batch lower-edge-batch source-batch batch D-next c-pull
      c-direct c-lower c-sources betas bs Us-tail U-tail c-tail
      child-costs-tail c)
  have tail:
    list-all2 (λc-child UB. case UB of (U-child, B-child) ⇒
      c-child ≤ bmssp-amortized-cost-bound A R h t q L B-child U-child)
      child-costs-tail (range-tree-child-bound-pair-list P b betas bs)
    using Direct-Insert-Costed-State-Step.IH Direct-Insert-Costed-State-Step.prems
    by blast
  have card-pull: card S-pull ≤ M-of l
    using Direct-Insert-Costed-State-Step
    unfolding pull-separates-def by blast
  have below-pull: ⋀x. x ∈ S-pull ⟹ below-bound (d x) (Fin beta)
  proof −
    fix x
    assume xS: x ∈ S-pull
    have S-pull = split-below d P beta
      using Direct-Insert-Costed-State-Step by blast
```

```
    then show below-bound (d x) (Fin beta)
      using xS unfolding split-below-def by auto
  qed
  have pull-k-cap: k * card S-pull ≤ cap
  proof −
    have S-pull = split-below d P beta
      using Direct-Insert-Costed-State-Step by blast
    then show ?thesis
      using split-below-scaled-card-le
        [OF Direct-Insert-Costed-State-Step.prems(1)
          Direct-Insert-Costed-State-Step.prems(2), of d beta]
      by simp
  qed
  have pull-anti: tree-antichain S-pull
  proof −
    have S-pull = split-below d P beta
      using Direct-Insert-Costed-State-Step by blast
    then show ?thesis
      using split-below-tree-antichain
        [OF Direct-Insert-Costed-State-Step.prems(3), of d beta]
      by simp
  qed
  have child-run:
    direct-insert-costed-bmssp Δ M-of t h k cap l d S-pull (Fin beta)
      d-child (Fin b) U-child c-child
    using Direct-Insert-Costed-State-Step by blast
  have pre-pull: bmssp-pre-full d S-pull (Fin beta)
    using Direct-Insert-Costed-State-Step by blast
  have reaches-pull: ⋀x. x ∈ S-pull ⟹ reachable s x
    using Direct-Insert-Costed-State-Step by blast
  have head:
    c-child ≤
      bmssp-amortized-cost-bound A R h t q L (Fin beta) U-child
    by (rule Direct-Insert-Costed-State-Step.prems(4)
      [OF child-run pre-pull reaches-pull card-pull below-pull
        pull-k-cap pull-anti])
  show ?case
    using head tail Direct-Insert-Costed-State-Step by simp
next
  case (Direct-Insert-Costed-Base S x Δ M-of t h k cap d B)
  then show ?case
    by simp
next
  case (Direct-Insert-Costed-Step D k cap d S B c-insert t Δ M-of h l
      d' a betas bs B' Us U-loop c-loop child-costs-loop U c)
  then show ?case
    by simp
qed
```

The central loop cost bound separates the four local charges that survive

after recursive child costs have been bounded: pulled child sources, direct edge-range inserts, lower edge-range batches, and source batches. The list functions over *range-tree-child-list* and its edge-range variants are the combinatorial map from one loop trace to disjoint graph regions. Their disjointness lemmas are what prevent the loop from paying for the same edge range repeatedly.

theorem *direct-insert-costed-partition-loop-state-cost-bound-by-child-sources-and-edge-ranges*:
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs $\Longrightarrow$
sound-label d $\Longrightarrow$
$\exists$ *child-sources*.
length child-sources = *length bs* $\wedge$
list-all2
(λ*S-child U-child*. $\exists$ *c-child B-child d-child B-child′*.
direct-insert-costed-bmssp Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child $\wedge$
bmssp-pre-full d S-child B-child $\wedge$
($\forall$ *x*$\in$*S-child*. *reachable s x*) $\wedge$
card S-child $\leq$ *M-of l* $\wedge$
($\forall$ *x*$\in$*S-child*. *below-bound* (*d x*) *B-child*))
child-sources (*range-tree-child-list P a bs*) $\wedge$
c $\leq$ *sum-list child-costs* + *sum-list* (*map card child-sources*) +
t $*$ *sum-list*
(*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*)) +
h $*$ *sum-list*
(*map card* (*range-tree-child-edge-range-list P a betas bs*)) +
h $*$ *sum-list* (*map card child-sources*)
and *direct-insert-costed-bmssp-cost-child-sources-and-edge-ranges-trivial*:
direct-insert-costed-bmssp Δ *M-of t h k cap l d S B d′ B′ U c* $\Longrightarrow$ *True*
proof (*induction rule*:
direct-insert-costed-partition-loop-state-direct-insert-costed-bmssp.inducts)
case (*Direct-Insert-Costed-State-Done D a B d′ P* Δ *M-of t h k cap l d*)
show *?case*
by (*intro exI*[*of* - []]) *simp*
next
case (*Direct-Insert-Costed-State-Stop a B d′ P k cap* Δ *M-of t h l d D*)
show *?case*
by (*intro exI*[*of* - []]) *simp*
next
case (*Direct-Insert-Costed-State-Step D M-of l Bmax S-pull beta D-pull B d*
P a b B′ d′ k cap Δ *t h d-child U-child c-child*
direct-edge-batch lower-edge-batch source-batch batch D-next c-pull
c-direct c-lower c-sources betas bs Us-tail U-tail c-tail
child-costs-tail c)
obtain *child-sources-tail* **where** *len-tail*:
length child-sources-tail = *length bs*
and *sources-tail*:
list-all2

```
      (λS-child U-child. ∃ c-child B-child d-child B-child'.
        direct-insert-costed-bmssp Δ M-of t h k cap l d S-child B-child
          d-child B-child' U-child c-child ∧
        bmssp-pre-full d S-child B-child ∧
        (∀ x∈S-child. reachable s x) ∧
        card S-child ≤ M-of l ∧
        (∀ x∈S-child. below-bound (d x) B-child))
       child-sources-tail (range-tree-child-list P b bs)
  and tail-cost:
    c-tail ≤
      sum-list child-costs-tail + sum-list (map card child-sources-tail) +
      t * sum-list
        (map card (range-tree-child-direct-edge-range-list P B b betas bs)) +
      h * sum-list
        (map card (range-tree-child-edge-range-list P b betas bs)) +
      h * sum-list (map card child-sources-tail)
  using Direct-Insert-Costed-State-Step.IH Direct-Insert-Costed-State-Step.prems
by blast
 have card-pull: card S-pull ≤ M-of l
  using Direct-Insert-Costed-State-Step
  unfolding pull-separates-def by blast
 have below-pull: ∀ x∈S-pull. below-bound (d x) (Fin beta)
 proof
  fix x
  assume xS: x ∈ S-pull
  have S-pull = split-below d P beta
    using Direct-Insert-Costed-State-Step by blast
  then show below-bound (d x) (Fin beta)
    using xS unfolding split-below-def by auto
 qed
 have pull-cost: c-pull ≤ card S-pull
  using Direct-Insert-Costed-State-Step
  unfolding partition-pull-cost-bound-def by blast
 have child-run:
  direct-insert-costed-bmssp Δ M-of t h k cap l d S-pull (Fin beta)
    d-child (Fin b) U-child c-child
  using Direct-Insert-Costed-State-Step by blast
 have child-pre: bmssp-pre-full d S-pull (Fin beta)
  using Direct-Insert-Costed-State-Step by blast
 have child-reaches: ⋀x. x ∈ S-pull ⟹ reachable s x
  using Direct-Insert-Costed-State-Step by blast
 have child-post:
  bmssp-post-full d S-pull (Fin beta) d-child (Fin b) U-child
  by (rule direct-insert-costed-bmssp-correct
    [OF child-run Direct-Insert-Costed-State-Step.prems child-pre child-reaches])
 have child-complete: complete-on d-child U-child
  using child-post unfolding bmssp-post-full-def by blast
 have U-child-eq: U-child = range-tree P a (Fin b)
  using Direct-Insert-Costed-State-Step by blast
```

have *U-child-reaches*: $\bigwedge u.\ u \in$ *U-child* $\Longrightarrow$ *reachable s u*
unfolding *U-child-eq range-tree-def tree-set-def tree-path-def* **by** *blast*
have *direct-cost*:
c-direct $\leq$ *t* $*$ *card* (*outgoing-edges-range U-child beta B*)
by (*rule partition-batch-cost-bound-edge-relaxation-pairs-in-bound*
[**where** *d* = *d-child*])
(*use Direct-Insert-Costed-State-Step child-complete U-child-reaches* **in** *auto*)
have *lower-source-cost*:
c-lower + *c-sources* $\leq$
h $*$ *card* (*outgoing-edges-range U-child b* (*Fin beta*)) +
h $*$ *card S-pull*
by (*rule range-costed-split-batch-cost-le-range-card*[
where *d* = *d* **and** *S* = *S-pull* **and** *d-child* = *d-child* **and** *b* = *b*
and *beta* = *beta*])
(*use Direct-Insert-Costed-State-Step child-complete U-child-reaches* **in** *auto*)
let *?child-sources* = *S-pull* # *child-sources-tail*
have *len*: *length ?child-sources* = *length* (*b* # *bs*)
using *len-tail* **by** *simp*
have *head-source*:
$\exists$ *c-child′ B-child d-child′ B-child′*.
direct-insert-costed-bmssp Δ *M-of t h k cap l d S-pull B-child*
d-child′ B-child′ U-child c-child′ $\wedge$
bmssp-pre-full d S-pull B-child $\wedge$
($\forall$ *x*$\in$*S-pull*. *reachable s x*) $\wedge$
card S-pull $\leq$ *M-of l* $\wedge$
($\forall$ *x*$\in$*S-pull*. *below-bound* (*d x*) *B-child*)
using *Direct-Insert-Costed-State-Step card-pull below-pull* **by** *blast*
have *sources*:
list-all2
(λ*S-child U-child*. $\exists$ *c-child B-child d-child B-child′*.
direct-insert-costed-bmssp Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child $\wedge$
bmssp-pre-full d S-child B-child $\wedge$
($\forall$ *x*$\in$*S-child*. *reachable s x*) $\wedge$
card S-child $\leq$ *M-of l* $\wedge$
($\forall$ *x*$\in$*S-child*. *below-bound* (*d x*) *B-child*))
?child-sources (*range-tree-child-list P a* (*b* # *bs*))
using *head-source sources-tail Direct-Insert-Costed-State-Step* **by** *simp*
have *cost*:
c $\leq$ *sum-list* (*c-child* # *child-costs-tail*) +
sum-list (*map card ?child-sources*) +
t $*$ *sum-list*
(*map card*
(*range-tree-child-direct-edge-range-list P B a* (*beta* # *betas*) (*b* # *bs*))) +
h $*$ *sum-list*
(*map card*
(*range-tree-child-edge-range-list P a* (*beta* # *betas*) (*b* # *bs*))) +
h $*$ *sum-list* (*map card ?child-sources*)
proof −

```
    have c-eq:
      c = c-pull + c-direct + c-lower + c-sources + c-child + c-tail
      using Direct-Insert-Costed-State-Step by blast
    have c-pull + c-direct + c-lower + c-sources + c-child + c-tail ≤
        card S-pull +
        t * card (outgoing-edges-range U-child beta B) +
        (h * card (outgoing-edges-range U-child b (Fin beta)) +
          h * card S-pull) +
        c-child +
        (sum-list child-costs-tail + sum-list (map card child-sources-tail) +
          t * sum-list
            (map card (range-tree-child-direct-edge-range-list P B b betas bs)) +
          h * sum-list
            (map card (range-tree-child-edge-range-list P b betas bs)) +
          h * sum-list (map card child-sources-tail))
      using pull-cost direct-cost lower-source-cost tail-cost by linarith
    also have ... =
        sum-list (c-child # child-costs-tail) +
        sum-list (map card ?child-sources) +
        t * (card (outgoing-edges-range U-child beta B) +
          sum-list
            (map card (range-tree-child-direct-edge-range-list P B b betas bs))) +
        h * (card (outgoing-edges-range U-child b (Fin beta)) +
          sum-list
            (map card (range-tree-child-edge-range-list P b betas bs))) +
        h * sum-list (map card ?child-sources)
      by (simp add: algebra-simps)
    also have ... =
        sum-list (c-child # child-costs-tail) +
        sum-list (map card ?child-sources) +
        t * sum-list
          (map card
            (range-tree-child-direct-edge-range-list P B a (beta # betas) (b # bs))) +
        h * sum-list
          (map card
            (range-tree-child-edge-range-list P a (beta # betas) (b # bs))) +
        h * sum-list (map card ?child-sources)
      using U-child-eq by simp
    finally show ?thesis
      using c-eq by simp
  qed
  show ?case
    by (intro exI[of - ?child-sources] conjI)
      (use len sources cost in simp-all)
next
  case (Direct-Insert-Costed-Base S x Δ M-of t h k cap d B)
  show ?case
    by simp
next
```

case (*Direct-Insert-Costed-Step D k cap d S B c-insert t Δ M-of h l*
d′ a betas bs B′ Us U-loop c-loop child-costs-loop U c)
show *?case*
by *simp*
qed

theorem *direct-insert-costed-partition-loop-state-cost-from-child-source-and-edge-range-bounds*:
assumes *run*:
direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B d′ D a
betas bs B′ Us U c child-costs
and *sound*: *sound-label d*
and *child-cost-bounds*:
list-all2 (λ*c-child U-child*.
c-child ≤ *level-range-cost-bound A R L U-child*)
child-costs (*range-tree-child-list P a bs*)
and *source-progress*:
⋀*S-child B-child d-child B-child′ U-child c-child*.
⟦*direct-insert-costed-bmssp Δ M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child;
bmssp-pre-full d S-child B-child;
⋀*x*. *x* ∈ *S-child* ⟹ *reachable s x*;
card S-child ≤ *M-of l*;
⋀*x*. *x* ∈ *S-child* ⟹ *below-bound* (*d x*) *B-child*⟧
⟹ *card S-child* ≤ *card U-child*
shows *c* ≤
A ∗ *L* ∗ *card* (*range-tree-chain P a bs B′*) +
R ∗ *card* (*outgoing-edges* (*range-tree-chain P a bs B′*)) +
t ∗ *sum-list*
(*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*)) +
h ∗ *sum-list*
(*map card* (*range-tree-child-edge-range-list P a betas bs*)) +
(*Suc h*) ∗ *card* (*range-tree-chain P a bs B′*)
proof −
obtain *child-sources* **where** *sources*:
list-all2
(λ*S-child U-child*. ∃ *c-child B-child d-child B-child′*.
direct-insert-costed-bmssp Δ M-of t h k cap l d S-child B-child
d-child B-child′ U-child c-child ∧
bmssp-pre-full d S-child B-child ∧
(∀ *x*∈*S-child*. *reachable s x*) ∧
card S-child ≤ *M-of l* ∧
(∀ *x*∈*S-child*. *below-bound* (*d x*) *B-child*))
child-sources (*range-tree-child-list P a bs*)
and *cost*:
c ≤ *sum-list child-costs* + *sum-list* (*map card child-sources*) +
t ∗ *sum-list*
(*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*)) +
h ∗ *sum-list*

```
        (map card (range-tree-child-edge-range-list P a betas bs)) +
      h * sum-list (map card child-sources)
    using direct-insert-costed-partition-loop-state-cost-bound-by-child-sources-and-
edge-ranges
      [OF run sound] by blast
  have mono: nondecreasing-from a bs
    by (rule direct-insert-costed-partition-loop-state-mono[OF run])
  have source-dom:
    list-all2 (λS-child U-child. card S-child ≤ card U-child)
      child-sources (range-tree-child-list P a bs)
    using sources
  proof (induction rule: list-all2-induct)
    case Nil
    then show ?case by simp
  next
    case (Cons S-child child-sources U-child child-sets)
    obtain c-child B-child d-child B-child' where child:
        direct-insert-costed-bmssp Δ M-of t h k cap l d S-child B-child
          d-child B-child' U-child c-child
      and child-pre: bmssp-pre-full d S-child B-child
      and child-reaches: ∀ x∈S-child. reachable s x
      and child-card: card S-child ≤ M-of l
      and child-below: ∀ x∈S-child. below-bound (d x) B-child
      using Cons.hyps by blast
    have head: card S-child ≤ card U-child
      by (rule source-progress[OF child child-pre - child-card])
        (use child-reaches child-below in blast)+
    show ?case
      using head Cons.IH Cons.hyps by simp
  qed
  have source-sum:
    sum-list (map card child-sources) ≤
      card (range-tree-chain P a bs B')
    by (rule sum-card-dominated-by-range-tree-child-list-le-chain
      [OF mono source-dom])
  have child-sum:
    sum-list child-costs ≤
      A * L * card (range-tree-chain P a bs B') +
      R * card (outgoing-edges (range-tree-chain P a bs B'))
    by (rule child-costs-le-level-range-child-list-bound
      [OF mono child-cost-bounds])
  have c ≤
      A * L * card (range-tree-chain P a bs B') +
      R * card (outgoing-edges (range-tree-chain P a bs B')) +
      t * sum-list
        (map card (range-tree-child-direct-edge-range-list P B a betas bs)) +
      h * sum-list
        (map card (range-tree-child-edge-range-list P a betas bs)) +
      sum-list (map card child-sources) +
```

```
    h * sum-list (map card child-sources)
   using cost child-sum by linarith
 also have ... ≤
    A * L * card (range-tree-chain P a bs B') +
    R * card (outgoing-edges (range-tree-chain P a bs B')) +
    t * sum-list
     (map card (range-tree-child-direct-edge-range-list P B a betas bs)) +
    h * sum-list
     (map card (range-tree-child-edge-range-list P a betas bs)) +
    (Suc h) * card (range-tree-chain P a bs B')
 proof −
   have source-part:
    sum-list (map card child-sources) +
      h * sum-list (map card child-sources) ≤
     Suc h * card (range-tree-chain P a bs B')
   proof −
     have h-part:
      h * sum-list (map card child-sources) ≤
        h * card (range-tree-chain P a bs B')
       using source-sum by simp
     have sum-list (map card child-sources) +
         h * sum-list (map card child-sources) ≤
       card (range-tree-chain P a bs B') +
         h * card (range-tree-chain P a bs B')
       using source-sum h-part by linarith
     then show ?thesis
       by (simp add: algebra-simps)
   qed
   show ?thesis
     using source-part by linarith
 qed
 finally show ?thesis .
qed

theorem direct-insert-costed-partition-loop-state-cost-from-child-source-and-
amortized-bounds:
 assumes run:
   direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
    betas bs B' Us U c child-costs
   and sound: sound-label d
   and child-cost-bounds:
    list-all2 (λc-child UB. case UB of (U-child, B-child) ⇒
      c-child ≤ bmssp-amortized-cost-bound A R h t q L B-child U-child)
      child-costs (range-tree-child-bound-pair-list P a betas bs)
   and source-progress:
    ⋀S-child B-child d-child B-child' U-child c-child.
     ⟦direct-insert-costed-bmssp Δ M-of t h k cap l d S-child B-child
        d-child B-child' U-child c-child;
      bmssp-pre-full d S-child B-child;
```

```
        ⋀x. x ∈ S-child ⟹ reachable s x;
        card S-child ≤ M-of l;
        ⋀x. x ∈ S-child ⟹ below-bound (d x) B-child⟧
       ⟹ card S-child ≤ card U-child
  shows c ≤
    A * L * card (range-tree-chain P a bs B') +
    (R + q * h) * card (outgoing-edges (range-tree-chain P a bs B')) +
    t * sum-list
      (map card (range-tree-child-zero-edge-range-list P a betas bs)) +
    t * sum-list
      (map card (range-tree-child-direct-edge-range-list P B a betas bs)) +
    h * sum-list
      (map card (range-tree-child-edge-range-list P a betas bs)) +
    (Suc h) * card (range-tree-chain P a bs B')
proof −
  obtain child-sources where sources:
      list-all2
       (λS-child U-child. ∃ c-child B-child d-child B-child'.
         direct-insert-costed-bmssp Δ M-of t h k cap l d S-child B-child
           d-child B-child' U-child c-child ∧
         bmssp-pre-full d S-child B-child ∧
         (∀ x∈S-child. reachable s x) ∧
         card S-child ≤ M-of l ∧
         (∀ x∈S-child. below-bound (d x) B-child))
       child-sources (range-tree-child-list P a bs)
    and cost:
      c ≤ sum-list child-costs + sum-list (map card child-sources) +
        t * sum-list
          (map card (range-tree-child-direct-edge-range-list P B a betas bs)) +
        h * sum-list
          (map card (range-tree-child-edge-range-list P a betas bs)) +
        h * sum-list (map card child-sources)
    using direct-insert-costed-partition-loop-state-cost-bound-by-child-sources-and-
edge-ranges
      [OF run sound] by blast
  have mono: nondecreasing-from a bs
    by (rule direct-insert-costed-partition-loop-state-mono[OF run])
  have source-dom:
    list-all2 (λS-child U-child. card S-child ≤ card U-child)
      child-sources (range-tree-child-list P a bs)
    using sources
  proof (induction rule: list-all2-induct)
    case Nil
    then show ?case by simp
  next
    case (Cons S-child child-sources U-child child-sets)
    obtain c-child B-child d-child B-child' where child:
        direct-insert-costed-bmssp Δ M-of t h k cap l d S-child B-child
          d-child B-child' U-child c-child
```

```
    and child-pre: bmssp-pre-full d S-child B-child
    and child-reaches: ∀ x∈S-child. reachable s x
    and child-card: card S-child ≤ M-of l
    and child-below: ∀ x∈S-child. below-bound (d x) B-child
    using Cons.hyps by blast
  have head: card S-child ≤ card U-child
    by (rule source-progress[OF child child-pre - child-card])
      (use child-reaches child-below in blast)+
  show ?case
    using head Cons.IH Cons.hyps by simp
qed
have source-sum:
  sum-list (map card child-sources) ≤
    card (range-tree-chain P a bs B')
  by (rule sum-card-dominated-by-range-tree-child-list-le-chain
    [OF mono source-dom])
have child-sum-raw:
  sum-list child-costs ≤
    A * L * sum-list (map card (range-tree-child-list P a bs)) +
    (R + q * h) *
      sum-list (map (λU. card (outgoing-edges U))
        (range-tree-child-list P a bs)) +
    t * sum-list
      (map card (range-tree-child-zero-edge-range-list P a betas bs))
  by (rule sum-bmssp-amortized-child-bounds-le[OF child-cost-bounds])
have child-card-sum:
  sum-list (map card (range-tree-child-list P a bs)) ≤
    card (range-tree-chain P a bs B')
  by (rule card-range-tree-child-list-le-chain[OF mono])
have child-out-sum:
  sum-list (map (λU. card (outgoing-edges U))
      (range-tree-child-list P a bs)) ≤
    card (outgoing-edges (range-tree-chain P a bs B'))
  by (rule card-outgoing-edges-range-tree-child-list-le-chain[OF mono])
have child-sum:
  sum-list child-costs ≤
    A * L * card (range-tree-chain P a bs B') +
    (R + q * h) * card (outgoing-edges (range-tree-chain P a bs B')) +
    t * sum-list
      (map card (range-tree-child-zero-edge-range-list P a betas bs))
proof −
  have vterm:
    A * L * sum-list (map card (range-tree-child-list P a bs)) ≤
     A * L * card (range-tree-chain P a bs B')
    using child-card-sum by simp
  have eterm:
    (R + q * h) *
      sum-list (map (λU. card (outgoing-edges U))
        (range-tree-child-list P a bs)) ≤
```

```
    (R + q * h) * card (outgoing-edges (range-tree-chain P a bs B'))
    using child-out-sum by simp
  show ?thesis
    using child-sum-raw vterm eterm by linarith
qed
have c ≤
    A * L * card (range-tree-chain P a bs B') +
    (R + q * h) * card (outgoing-edges (range-tree-chain P a bs B')) +
    t * sum-list
      (map card (range-tree-child-zero-edge-range-list P a betas bs)) +
    t * sum-list
      (map card (range-tree-child-direct-edge-range-list P B a betas bs)) +
    h * sum-list
      (map card (range-tree-child-edge-range-list P a betas bs)) +
    sum-list (map card child-sources) +
    h * sum-list (map card child-sources)
  using cost child-sum by linarith
also have ... ≤
    A * L * card (range-tree-chain P a bs B') +
    (R + q * h) * card (outgoing-edges (range-tree-chain P a bs B')) +
    t * sum-list
      (map card (range-tree-child-zero-edge-range-list P a betas bs)) +
    t * sum-list
      (map card (range-tree-child-direct-edge-range-list P B a betas bs)) +
    h * sum-list
      (map card (range-tree-child-edge-range-list P a betas bs)) +
    (Suc h) * card (range-tree-chain P a bs B')
proof −
  have source-part:
    sum-list (map card child-sources) +
      h * sum-list (map card child-sources) ≤
     Suc h * card (range-tree-chain P a bs B')
  proof −
    have h-part:
      h * sum-list (map card child-sources) ≤
        h * card (range-tree-chain P a bs B')
      using source-sum by simp
    have sum-list (map card child-sources) +
        h * sum-list (map card child-sources) ≤
      card (range-tree-chain P a bs B') +
        h * card (range-tree-chain P a bs B')
      using source-sum h-part by linarith
    then show ?thesis
      by (simp add: algebra-simps)
  qed
  show ?thesis
    using source-part by linarith
qed
finally show ?thesis .
```

qed

The amortised closing theorem below keeps the direct-insert range term visible because the bucketed implementation has a sharper accounting story for it. The child calls contribute a lower-level amortised bound, the source-progress hypothesis absorbs the pulled source sets into completed child ranges, and the edge-range disjointness lemmas collapse the direct and lower edge lists into the outgoing-edge terms of the parent output.

theorem *direct-insert-costed-partition-loop-state-closes-amortized-bound-from-child-costs*:
assumes *run*:
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*
and *child-cost-bounds*:
list-all2 (λ*c-child UB. case UB of* (*U-child*, *B-child*) $\Rightarrow$
c-child $\leq$ *bmssp-amortized-cost-bound A R h t q L B-child U-child*)
child-costs (*range-tree-child-bound-pair-list P a betas bs*)
and *source-factor*: *Suc h* $\leq$ *A*
and *source-progress*:
$\bigwedge$*S-child B-child d-child B-child′ U-child c-child.*
⟦*direct-insert-costed-bmssp* Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child;
bmssp-pre-full d S-child B-child;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *below-bound* (*d x*) *B-child*⟧
$\Longrightarrow$ *card S-child* $\leq$ *card U-child*
shows *c* $\leq$ *bmssp-amortized-cost-bound A R h t* (*Suc q*) (*Suc L*) *B U*
proof −
have *trace*: *concrete-partition-loop-trace P B a bs d′ B′ Us U*
by (*rule direct-insert-costed-partition-loop-state-trace*
[*OF run sound pre P-reaches*])
have *children*:
list-all2 (λ*U X. U* = *X* $\wedge$ *complete-on d′ U*) *Us*
(*range-tree-chain-list P a bs B′*)
using *trace* **unfolding** *concrete-partition-loop-trace-def* **by** *blast*
have *U-def*: *U* = *bound-tree P* (*Fin a*) $\cup \bigcup$(*set Us*)
using *trace* **unfolding** *concrete-partition-loop-trace-def* **by** *blast*
have *Union-eq*:
$\bigcup$(*set Us*) = $\bigcup$(*set* (*range-tree-chain-list P a bs B′*))
using *children* **by** (*induction rule*: *list-all2-induct*) *auto*
have *U-eq-chain*:
U = *bound-tree P* (*Fin a*) $\cup$ *range-tree-chain P a bs B′*
using *U-def Union-eq Union-range-tree-chain-list*[*of P a bs B′*] **by** *simp*
have *finite-U*: *finite U*
using *U-eq-chain* **by** *simp*

have *chain-subset*: *range-tree-chain P a bs B′ ⊆ U*
using *U-eq-chain* **by** *blast*
have *card-chain-le*: *card* (*range-tree-chain P a bs B′*) ≤ *card U*
by (*rule card-mono*[*OF finite-U chain-subset*])
have *outgoing-subset*:
outgoing-edges (*range-tree-chain P a bs B′*) ⊆ *outgoing-edges U*
by (*rule outgoing-edges-mono*[*OF chain-subset*])
have *finite-out-U*: *finite* (*outgoing-edges U*)
by *simp*
have *card-out-le*:
card (*outgoing-edges* (*range-tree-chain P a bs B′*)) ≤
card (*outgoing-edges U*)
by (*rule card-mono*[*OF finite-out-U outgoing-subset*])
have *range-subset*:
outgoing-edges-range (*range-tree-chain P a bs B′*) *0 B* ⊆
outgoing-edges-range U 0 B
by (*rule outgoing-edges-range-mono-sources*[*OF chain-subset*])
have *card-range-le*:
card (*outgoing-edges-range* (*range-tree-chain P a bs B′*) *0 B*) ≤
card (*outgoing-edges-range U 0 B*)
by (*rule card-mono*[*OF finite-outgoing-edges-range range-subset*])
have *mono*: *nondecreasing-from a bs*
by (*rule direct-insert-costed-partition-loop-state-mono*[*OF run*])
have *beta-bounds*: *bounds-le B betas*
by (*rule direct-insert-costed-partition-loop-state-beta-bounds*[*OF run*])
have *cost*:
c ≤
A ∗ *L* ∗ *card* (*range-tree-chain P a bs B′*) +
(*R* + *q* ∗ *h*) ∗ *card* (*outgoing-edges* (*range-tree-chain P a bs B′*)) +
t ∗ *sum-list*
(*map card* (*range-tree-child-zero-edge-range-list P a betas bs*)) +
t ∗ *sum-list*
(*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*)) +
h ∗ *sum-list*
(*map card* (*range-tree-child-edge-range-list P a betas bs*)) +
(*Suc h*) ∗ *card* (*range-tree-chain P a bs B′*)
by (*rule direct-insert-costed-partition-loop-state-cost-from-child-source-and-amortized-bounds*
[*OF run sound child-cost-bounds source-progress*])
have *lower-sum*:
sum-list (*map card* (*range-tree-child-edge-range-list P a betas bs*))
≤ *card* (*outgoing-edges* (*range-tree-chain P a bs B′*))
by (*rule sum-card-range-tree-child-edge-range-list-le-outgoing-edges-chain*
[*OF mono*])
have *direct-sum*:
sum-list
(*map card* (*range-tree-child-zero-edge-range-list P a betas bs*)) +
sum-list
(*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*))

```
    ≤ card (outgoing-edges-range (range-tree-chain P a bs B') 0 B)
     by (rule sum-card-range-tree-child-zero-direct-edge-ranges-le-outgoing-edges-
range-chain
      [OF mono beta-bounds])
  have vertex-part:
    A * L * card (range-tree-chain P a bs B') +
     Suc h * card (range-tree-chain P a bs B') ≤
    A * Suc L * card U
  proof −
    have source-term:
      Suc h * card (range-tree-chain P a bs B') ≤
       A * card U
    proof −
      have Suc h * card (range-tree-chain P a bs B') ≤
          A * card (range-tree-chain P a bs B')
        by (rule mult-right-mono[OF source-factor]) simp
      also have ... ≤ A * card U
        using card-chain-le by simp
      finally show ?thesis .
    qed
    have child-term:
      A * L * card (range-tree-chain P a bs B') ≤
       A * L * card U
      using card-chain-le by simp
    have A * L * card U + A * card U = A * Suc L * card U
      by (simp add: algebra-simps)
    then show ?thesis
      using child-term source-term by linarith
  qed
  have edge-part:
    (R + q * h) * card (outgoing-edges (range-tree-chain P a bs B')) +
      h * sum-list
        (map card (range-tree-child-edge-range-list P a betas bs))
     ≤ (R + Suc q * h) * card (outgoing-edges U)
  proof −
    have lower-term:
      h * sum-list
        (map card (range-tree-child-edge-range-list P a betas bs))
       ≤ h * card (outgoing-edges (range-tree-chain P a bs B'))
      using lower-sum by simp
    have chain-term:
      (R + q * h) * card (outgoing-edges (range-tree-chain P a bs B')) +
        h * card (outgoing-edges (range-tree-chain P a bs B')) =
       (R + Suc q * h) *
        card (outgoing-edges (range-tree-chain P a bs B'))
      by (simp add: algebra-simps)
    have to-chain:
      (R + q * h) * card (outgoing-edges (range-tree-chain P a bs B')) +
        h * sum-list
```

```
      (map card (range-tree-child-edge-range-list P a betas bs))
    ≤ (R + Suc q * h) *
     card (outgoing-edges (range-tree-chain P a bs B'))
   using lower-term chain-term by linarith
  have to-U:
   (R + Suc q * h) *
     card (outgoing-edges (range-tree-chain P a bs B')) ≤
    (R + Suc q * h) * card (outgoing-edges U)
   using card-out-le by simp
  show ?thesis
   using to-chain to-U by linarith
 qed
 have direct-part:
  t * sum-list
      (map card (range-tree-child-zero-edge-range-list P a betas bs)) +
    t * sum-list
      (map card (range-tree-child-direct-edge-range-list P B a betas bs))
   ≤ t * card (outgoing-edges-range U 0 B)
 proof −
  have t * (sum-list
      (map card (range-tree-child-zero-edge-range-list P a betas bs)) +
    sum-list
      (map card (range-tree-child-direct-edge-range-list P B a betas bs)))
    ≤ t * card (outgoing-edges-range (range-tree-chain P a bs B') 0 B)
    using direct-sum by simp
  also have ... ≤ t * card (outgoing-edges-range U 0 B)
    using card-range-le by simp
  finally show ?thesis
    by (simp add: algebra-simps)
 qed
 have c ≤
    A * Suc L * card U +
    (R + Suc q * h) * card (outgoing-edges U) +
    t * card (outgoing-edges-range U 0 B)
  using cost vertex-part edge-part direct-part by linarith
 then show ?thesis
  unfolding bmssp-amortized-cost-bound-def .
qed
```

The final level-bound theorem is the non-amortised companion used by the abstract headline recurrence. It closes the same loop skeleton, but leaves the direct and lower edge-range sums explicit rather than folding them into *bmssp-amortized-cost-bound*. This gives later theories the freedom to plug in either an abstract partition budget or the bucketed amortised budget without changing the semantic development.

```
theorem direct-insert-costed-partition-loop-state-closes-level-bound-from-child-costs-and-edge-ranges:
 assumes run:
  direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
```

```
    betas bs B' Us U c child-costs
  and sound: sound-label d
  and pre: bmssp-pre-full d P B
  and P-reaches: ⋀x. x ∈ P ⟹ reachable s x
  and child-cost-bounds:
    list-all2 (λc-child U-child.
      c-child ≤ level-range-cost-bound A R L U-child)
      child-costs (range-tree-child-list P a bs)
  and source-factor: Suc h ≤ A
  and source-progress:
    ⋀S-child B-child d-child B-child' U-child c-child.
      ⟦direct-insert-costed-bmssp Δ M-of t h k cap l d S-child B-child
          d-child B-child' U-child c-child;
        bmssp-pre-full d S-child B-child;
        ⋀x. x ∈ S-child ⟹ reachable s x;
        card S-child ≤ M-of l;
        ⋀x. x ∈ S-child ⟹ below-bound (d x) B-child⟧
      ⟹ card S-child ≤ card U-child
shows c ≤
  A * Suc L * card U + R * card (outgoing-edges U) +
  t * sum-list
    (map card (range-tree-child-direct-edge-range-list P B a betas bs)) +
  h * sum-list
    (map card (range-tree-child-edge-range-list P a betas bs))
proof −
  have trace: concrete-partition-loop-trace P B a bs d' B' Us U
    by (rule direct-insert-costed-partition-loop-state-trace
      [OF run sound pre P-reaches])
  have children:
    list-all2 (λU X. U = X ∧ complete-on d' U) Us
      (range-tree-chain-list P a bs B')
    using trace unfolding concrete-partition-loop-trace-def by blast
  have U-def: U = bound-tree P (Fin a) ∪ ⋃(set Us)
    using trace unfolding concrete-partition-loop-trace-def by blast
  have Union-eq:
    ⋃(set Us) = ⋃(set (range-tree-chain-list P a bs B'))
    using children by (induction rule: list-all2-induct) auto
  have U-eq-chain:
    U = bound-tree P (Fin a) ∪ range-tree-chain P a bs B'
    using U-def Union-eq Union-range-tree-chain-list[of P a bs B'] by simp
  have finite-U: finite U
    using U-eq-chain by simp
  have chain-subset: range-tree-chain P a bs B' ⊆ U
    using U-eq-chain by blast
  have card-chain-le: card (range-tree-chain P a bs B') ≤ card U
    by (rule card-mono[OF finite-U chain-subset])
  have outgoing-subset:
    outgoing-edges (range-tree-chain P a bs B') ⊆ outgoing-edges U
    by (rule outgoing-edges-mono[OF chain-subset])
```

have *finite-out-U*: *finite* (*outgoing-edges U*)
by *simp*
have *card-out-le*:
card (*outgoing-edges* (*range-tree-chain P a bs B′*)) ≤
card (*outgoing-edges U*)
by (*rule card-mono*[*OF finite-out-U outgoing-subset*])
have *loop*:
c ≤
A ∗ *L* ∗ *card* (*range-tree-chain P a bs B′*) +
R ∗ *card* (*outgoing-edges* (*range-tree-chain P a bs B′*)) +
t ∗ *sum-list*
(*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*)) +
h ∗ *sum-list*
(*map card* (*range-tree-child-edge-range-list P a betas bs*)) +
Suc h ∗ *card* (*range-tree-chain P a bs B′*)
by (*rule direct-insert-costed-partition-loop-state-cost-from-child-source-and-edge-range-bounds*
[*OF run sound child-cost-bounds source-progress*])
have *c* ≤
A ∗ *L* ∗ *card U* + *R* ∗ *card* (*outgoing-edges U*) +
t ∗ *sum-list*
(*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*)) +
h ∗ *sum-list*
(*map card* (*range-tree-child-edge-range-list P a betas bs*)) +
A ∗ *card U*
proof −
have *vterm*:
A ∗ *L* ∗ *card* (*range-tree-chain P a bs B′*) ≤ *A* ∗ *L* ∗ *card U*
using *card-chain-le* **by** *simp*
have *eterm*:
R ∗ *card* (*outgoing-edges* (*range-tree-chain P a bs B′*)) ≤
R ∗ *card* (*outgoing-edges U*)
using *card-out-le* **by** *simp*
have *source-term*:
Suc h ∗ *card* (*range-tree-chain P a bs B′*) ≤ *A* ∗ *card U*
proof −
have *Suc h* ∗ *card* (*range-tree-chain P a bs B′*) ≤
A ∗ *card* (*range-tree-chain P a bs B′*)
by (*rule mult-right-mono*[*OF source-factor*]) *simp*
also have . . . ≤ *A* ∗ *card U*
using *card-chain-le* **by** *simp*
finally show *?thesis* **.**
qed
show *?thesis*
using *loop vterm eterm source-term* **by** *linarith*
qed
also have . . . =
A ∗ *Suc L* ∗ *card U* + *R* ∗ *card* (*outgoing-edges U*) +
t ∗ *sum-list*

```
    (map card (range-tree-child-direct-edge-range-list P B a betas bs)) +
   h * sum-list
    (map card (range-tree-child-edge-range-list P a betas bs))
  by (simp add: algebra-simps)
 finally show ?thesis .
qed

end

end
theory BMSSP-Strict-Tie-Breaking
 imports BMSSP-Direct-Insert-Costed
begin
```

33 Strict Tie-Breaking Consequences

Lexicographic tie-breaking removes distance ties by comparing extended path values. The existing development abstracts uniqueness of shortest paths, but source progress for the base case also needs strict growth along the selected shortest-path tree. This file records exactly that consequence, so the remaining complexity proof can be connected either to a formal extended-distance construction or to any graph model that already provides the same strict tree order.

The cost recurrence uses progress statements of the form "a source set is no larger than the output range that completed it". For non-base recursive calls this comes from the BMSSP threshold: either the call succeeds with the old bound, or it completes enough vertices. The base case is subtler. It takes the first $k + 1$ vertices in distance order from a singleton source. To know that the singleton source itself is included, the proof needs the source to be the strict first element of that order.

The strict tie-breaking locale supplies this missing monotonicity. Along the chosen shortest-path tree, descendants have strictly larger distance than ancestors. Consequently, in the singleton bound tree rooted at x, the root x is the unique minimum-distance vertex. This is the formal counterpart of the paper's assumption that ties can be perturbed away without changing the combinatorial structure of the algorithm.

The later half of the theory threads this progress fact through the exact range-costed and direct-insert costed relations. It closes the source- progress side condition required by the local-budget theorems, then packages the resulting level and amortised graph bounds for the top-level run.

```
context finite-weighted-digraph
begin

lemma walk-weight-pos-if-positive-edges:
 assumes positive: ⋀u v. (u, v) ∈ E ⟹ 0 < w u v
```

```
    and walk-p: walk p
    and len: 1 < length p
  shows 0 < walk-weight p
proof (cases p)
  case Nil
  then show ?thesis
    using len by simp
next
  case (Cons x xs)
  then show ?thesis
  proof (cases xs)
    case Nil
    then show ?thesis
      using Cons len by simp
  next
    case (Cons y ys)
    have p-def: p = x # y # ys
      using ‹p = x # xs› Cons by simp
    have edge: (x, y) ∈ E
      using walk-p p-def by simp
    have tail-walk: walk (y # ys)
      using walk-p p-def by simp
    have 0 < w x y
      by (rule positive[OF edge])
    moreover have 0 ≤ walk-weight (y # ys)
      by (rule walk-weight-nonneg[OF tail-walk])
    ultimately show ?thesis
      using p-def by simp
  qed
qed

end

context unique-shortest-digraph
begin

lemma tree-path-strict-dist-if-positive-edges:
  assumes positive: ⋀u v. (u, v) ∈ E ⟹ 0 < w u v
    and tree: tree-path u v
    and neq: u ≠ v
  shows dist s u < dist s v
proof −
  let ?p = shortest-path-to v
  have reach-v: reachable s v and u-path: u ∈ set ?p
    using tree by (auto dest: tree-pathD)
  obtain i where i: i < length ?p ?p ! i = u
    using u-path by (auto simp: in-set-conv-nth)
  have sp: shortest-walk s ?p v
    by (rule shortest-path-to-shortest[OF reach-v])
```

```
  have simple-p: simple-walk-betw s ?p v
    using sp unfolding shortest-walk-def by blast
  have weight-p: walk-weight ?p = dist s v
    using sp unfolding shortest-walk-def by blast
  have p-ne: ?p ≠ [] and last-p: last ?p = v
    using sp unfolding shortest-walk-def simple-walk-betw-def walk-betw-def
    by blast+
  have suc-i: Suc i < length ?p
  proof (rule ccontr)
    assume ¬ Suc i < length ?p
    then have last-i: i = length ?p − 1
      using i(1) by simp
    have ?p ! i = last ?p
      using p-ne last-i by (simp add: last-conv-nth)
    then show False
      using i(2) last-p neq by simp
  qed
  let ?pref = take (Suc i) ?p
  let ?suf = drop i ?p
  have pref-short: shortest-walk s ?pref u
    using shortest-walk-prefix-shortest[OF sp i(1)] i(2) by simp
  have pref-weight: walk-weight ?pref = dist s u
    using pref-short unfolding shortest-walk-def by blast
  have pref-betw: walk-betw s ?pref u
    using pref-short unfolding shortest-walk-def simple-walk-betw-def by blast
  have suf-betw0: walk-betw (?p ! i) ?suf v
    by (rule walk-suffix-from-index[OF simple-p i(1)])
  have suf-betw: walk-betw u ?suf v
    using suf-betw0 i(2) by simp
  have suf-walk: walk ?suf
    using suf-betw unfolding walk-betw-def by blast
  have suf-len: 1 < length ?suf
    using suc-i by simp
  have suf-pos: 0 < walk-weight ?suf
    by (rule walk-weight-pos-if-positive-edges[OF positive suf-walk suf-len])
  have split-p: ?pref @ tl ?suf = ?p
    using take-Suc-append-tl-drop[OF i(1)] .
  have walk-weight ?p = walk-weight ?pref + walk-weight ?suf
    using walk-weight-append-tl-betw[OF pref-betw suf-betw] split-p by simp
  then show ?thesis
    using weight-p pref-weight suf-pos by linarith
qed

end

locale strict-tie-breaking-digraph = unique-shortest-digraph +
  assumes tree-path-strict-dist:
    ⟦tree-path u v; u ≠ v⟧ ⟹ dist s u < dist s v
```

locale *positive-unique-shortest-digraph* = *unique-shortest-digraph* +
  assumes *positive-weight*: $(u, v) \in E \Longrightarrow 0 < w\ u\ v$
begin

sublocale *strict-tie-breaking-digraph*
proof
  show $\bigwedge u\ v.\ [\![\textit{tree-path}\ u\ v;\ u \neq v]\!] \Longrightarrow \textit{dist}\ s\ u < \textit{dist}\ s\ v$
    by (*rule tree-path-strict-dist-if-positive-edges*[*OF positive-weight*])
qed

end

The first locale bridge shows one concrete way to obtain strict tree growth: positive edge weights imply every non-empty suffix of a shortest path has positive weight, and therefore every proper descendant in the selected shortest-path tree has strictly larger distance. The abstract *strict-tie-breaking-digraph* locale records only the consequence needed by the runtime proof, avoiding a commitment to any particular perturbation or tie-breaking construction.

context *strict-tie-breaking-digraph*
begin

lemma *bound-tree-singleton-tree-path*:
  assumes $y \in \textit{bound-tree}\ \{x\}\ B$
  shows *tree-path x y*
proof −
  have *reach-y*: *reachable s y* **and** *through-y*: *through* $\{x\}$ *y*
    using *assms* **unfolding** *bound-tree-def* **by** *auto*
  then show *?thesis*
    using *through-iff-tree-path*[*OF reach-y*, *of* $\{x\}$] **by** *auto*
qed

lemma *bound-tree-singleton-root-strict-min*:
  assumes *x-bound*: $x \in \textit{bound-tree}\ \{x\}\ B$
    and *y-bound*: $y \in \textit{bound-tree}\ \{x\}\ B$
    and *neq*: $y \neq x$
  shows $\textit{dist}\ s\ x < \textit{dist}\ s\ y$
proof −
  have *tree-path x y*
    by (*rule bound-tree-singleton-tree-path*[*OF y-bound*])
  then show *?thesis*
    using *tree-path-strict-dist neq* **by** *blast*
qed

lemma *base-case-order-hd-root*:
  assumes *x-bound*: $x \in \textit{bound-tree}\ \{x\}\ B$
  shows $\textit{base-case-order}\ x\ B \neq [] \wedge \textit{hd}\ (\textit{base-case-order}\ x\ B) = x$
proof −
  let *?xs* = *base-case-order x B*
  have *set-xs*: *set ?xs* = *bound-tree* $\{x\}$ *B*

```
    by (rule base-case-order-set)
  have x-set: x ∈ set ?xs
    using x-bound set-xs by simp
  then have xs-ne: ?xs ≠ []
    by auto
  have hd-in: hd ?xs ∈ bound-tree {x} B
    using xs-ne set-xs by (metis hd-in-set)
  have sorted: sorted-wrt (λu v. dist s u ≤ dist s v) ?xs
    by (rule base-case-order-sorted)
  have hd-eq: hd ?xs = x
  proof (rule ccontr)
    assume hd-neq: hd ?xs ≠ x
    obtain i where i: i < length ?xs ?xs ! i = x
      using x-set by (auto simp: in-set-conv-nth)
    have i-pos: 0 < i
    proof (rule ccontr)
      assume ¬ 0 < i
      then have i = 0
        by simp
      then show False
        using i xs-ne hd-neq by (simp add: hd-conv-nth)
    qed
    have hd-nth: ?xs ! 0 = hd ?xs
      using xs-ne by (simp add: hd-conv-nth)
    have dist s (hd ?xs) ≤ dist s x
      using sorted-wrt-nth-less[OF sorted i-pos i(1)] i(2) hd-nth by simp
    moreover have dist s x < dist s (hd ?xs)
      by (rule bound-tree-singleton-root-strict-min[OF x-bound hd-in hd-neq])
    ultimately show False
      by simp
  qed
  show ?thesis
    using xs-ne hd-eq by blast
qed

lemma source-in-base-case-vertices:
  assumes x-bound: x ∈ bound-tree {x} B
    and k-pos: 0 < k
  shows x ∈ base-case-vertices k x B
proof (cases length (base-case-order x B) ≤ k)
  case True
  then show ?thesis
    using x-bound base-case-success[of x B k] by simp
next
  case False
  let ?xs = base-case-order x B
  have len: k < length ?xs
    using False by simp
  have hd-root: hd ?xs = x and xs-ne: ?xs ≠ []
```

```
    using base-case-order-hd-root[OF x-bound] by blast+
  have x-take: x ∈ set (take (Suc k) ?xs)
  proof −
    have ?xs ! 0 = x
      using hd-root xs-ne by (simp add: hd-conv-nth)
    moreover have 0 < length (take (Suc k) ?xs)
      using len by simp
    ultimately show ?thesis
      using nth-mem[of 0 take (Suc k) ?xs] by simp
  qed
  have kth-bound: ?xs ! k ∈ bound-tree {x} B
    using len base-case-order-set[of x B] nth-mem by metis
  have kth-neq: ?xs ! k ≠ x
  proof
    assume kth-eq: ?xs ! k = x
    have distinct: distinct ?xs
      by (rule base-case-order-distinct)
    have ?xs ! 0 = ?xs ! k
      using kth-eq hd-root xs-ne by (simp add: hd-conv-nth)
    moreover have 0 < length ?xs k < length ?xs 0 ≠ k
      using xs-ne len k-pos by auto
    ultimately show False
      using distinct by (meson nth-eq-iff-index-eq)
  qed
  have dist-lt: dist s x < dist s (?xs ! k)
    by (rule bound-tree-singleton-root-strict-min[OF x-bound kth-bound kth-neq])
  have base-case-vertices k x B =
      {v ∈ set (take (Suc k) ?xs). dist s v < dist s (?xs ! k)}
    using len unfolding base-case-vertices-def by (simp add: Let-def)
  then show ?thesis
    using x-take dist-lt by simp
qed
```

The base-case lemmas turn strict tree growth into the concrete progress fact used by the recurrence. In a singleton problem, every vertex in *bound-tree* $\{x\}$ B lies below x in the selected shortest-path tree. Strictness makes x the first element of *base-case-order* x B. Hence the bounded base scan always includes its source whenever $k > 0$.

This small fact is what lets the base case participate in the same source accounting as the recursive cases: the source set has cardinality one and is contained in the returned base-case vertex set.

```
lemma tree-path-tight-successor:
  assumes tight: tight-edge-step u v
  shows tree-path u v
proof (cases u = v)
  case True
  have reach-u: reachable s u
    using tight unfolding tight-edge-step-def by blast
  have sp-u: shortest-walk s (shortest-path-to u) u
```

```
    by (rule shortest-path-to-shortest[OF reach-u])
  have u ∈ set (shortest-path-to u)
  proof −
    have shortest-path-to u ≠ [] last (shortest-path-to u) = u
      using shortest-walk-hd[OF sp-u] by blast+
    then show ?thesis
      by (metis last-in-set)
  qed
  then show ?thesis
    using True reach-u unfolding tree-path-def by blast
next
  case False
  have edge: (u, v) ∈ E
    using tight unfolding tight-edge-step-def by blast
  have reach-u: reachable s u
    using tight unfolding tight-edge-step-def by blast
  have dist-v: dist s v = dist s u + w u v
    using tight unfolding tight-edge-step-def by blast
  let ?p = shortest-path-to u
  have sp-u: shortest-walk s ?p u
    by (rule shortest-path-to-shortest[OF reach-u])
  have simple-p: simple-walk-betw s ?p u
    using sp-u unfolding shortest-walk-def by blast
  have weight-p: walk-weight ?p = dist s u
    using sp-u unfolding shortest-walk-def by blast
  have p-ne: ?p ≠ [] and last-p: last ?p = u
    using shortest-walk-hd[OF sp-u] by blast+
  have v-notin: v ∉ set ?p
  proof
    assume v-in: v ∈ set ?p
    have tree-vu: tree-path v u
      by (rule tree-pathI[OF reach-u v-in])
    have vu-neq: v ≠ u
      using False by simp
    have dist s v < dist s u
      by (rule tree-path-strict-dist[OF tree-vu vu-neq])
    moreover have dist s u ≤ dist s v
      using dist-v nonneg-weight[OF edge] by linarith
    ultimately show False
      by simp
  qed
  have simple-q: simple-walk-betw s (?p @ [v]) v
    by (rule simple-walk-snoc[OF simple-p edge v-notin])
  have weight-q: walk-weight (?p @ [v]) = dist s v
    using weight-p p-ne last-p dist-v
    by (simp add: walk-weight-snoc)
  have sp-q: shortest-walk s (?p @ [v]) v
    using simple-q weight-q unfolding shortest-walk-def by blast
  have reach-v: reachable s v
```

```
    by (rule dist-triangle-edge(1)[OF edge reach-u])
  have sp-v: shortest-walk s (shortest-path-to v) v
    by (rule shortest-path-to-shortest[OF reach-v])
  have path-eq: shortest-path-to v = ?p @ [v]
    using unique-shortest-walk[OF sp-v sp-q] .
  have u ∈ set ?p
    using p-ne last-p by (metis last-in-set)
  then have u ∈ set (?p @ [v])
    by simp
  then have u ∈ set (shortest-path-to v)
    using path-eq by simp
  then show ?thesis
    by (rule tree-pathI[OF reach-v])
qed

lemma fp-next-bound-tree:
  assumes sound: sound-label d
    and pre: bmssp-pre-full d S B
    and F-bound: F ⊆ bound-tree S B
    and v-next: v ∈ fp-next d F B
  shows v ∈ bound-tree S B
proof −
  obtain u where uF: u ∈ F
    and edge: (u, v) ∈ E
    and relax: d u + w u v ≤ d v
    and below-cand: below-bound (d u + w u v) B
    using v-next unfolding fp-next-def by blast
  have u-bound: u ∈ bound-tree S B
    using F-bound uF by blast
  have reach-u: reachable s u
    using u-bound unfolding bound-tree-def by blast
  have reach-v: reachable s v
    by (rule dist-triangle-edge(1)[OF edge reach-u])
  have dist-triangle: dist s v ≤ dist s u + w u v
    by (rule dist-triangle-edge(2)[OF edge reach-u])
  have vV: v ∈ V
    using edge-in-V[OF edge] by simp
  have uV: u ∈ V
    using u-bound unfolding bound-tree-def by blast
  have dist-u-le: dist s u ≤ d u
    using sound uV reach-u unfolding sound-label-def by blast
  have dist-v-le-cand: dist s v ≤ d u + w u v
    using dist-triangle dist-u-le by linarith
  have below-v: below-bound (dist s v) B
    by (rule below-bound-le-trans[OF dist-v-le-cand below-cand])
  have through-v: through S v
  proof (cases d v = dist s v)
    case True
    have cand-le-dist: d u + w u v ≤ dist s v
```

```
    using relax True by simp
  have dist-v-eq: dist s v = dist s u + w u v
    using dist-triangle dist-u-le cand-le-dist by linarith
  have tight-uv: tight-edge-step u v
    using edge reach-u dist-v-eq unfolding tight-edge-step-def by blast
  have tree-uv: tree-path u v
    by (rule tree-path-tight-successor[OF tight-uv])
  have through-u: through S u
    using u-bound unfolding bound-tree-def by blast
  obtain x where xS: x ∈ S and tree-xu: tree-path x u
    using through-iff-tree-path[OF reach-u, of S] through-u by blast
  have tree-xv: tree-path x v
    by (rule tree-path-trans[OF tree-uv tree-xu])
  show ?thesis
    using through-iff-tree-path[OF reach-v, of S] xS tree-xv by blast
 next
  case False
  have through-complete d S v
    using pre vV reach-v below-v False unfolding bmssp-pre-full-def by blast
  then show ?thesis
    unfolding through-complete-def through-def by blast
 qed
 show ?thesis
  using vV reach-v through-v below-v unfolding bound-tree-def by blast
qed

lemma fp-iter-capped-seen-subset-bound-tree:
 assumes sound: sound-label d
  and pre: bmssp-pre-full d S B
  and F-bound: F ⊆ bound-tree S B
  and W-bound: W ⊆ bound-tree S B
 shows fp-seen (fp-iter-capped n cap d F W B) ⊆ bound-tree S B
 using assms
proof (induction n arbitrary: d F W)
 case 0
 then show ?case
  unfolding fp-seen-def by simp
next
 case (Suc n)
 let ?d' = relax-frontier d F
 let ?F' = fp-next d F B
 let ?W' = W ∪ ?F'
 have F-reaches: ⋀u. u ∈ F ⟹ reachable s u
  using Suc.prems(3) unfolding bound-tree-def by blast
 have sound': sound-label ?d'
  by (rule relax-frontier-sound[OF Suc.prems(1) F-reaches])
 have pre': bmssp-pre-full ?d' S B
  by (rule relax-frontier-preserves-bmssp-pre-full
    [OF Suc.prems(1,2) F-reaches])
```

```
  have F'-bound: ?F' ⊆ bound-tree S B
  proof
    fix v
    assume v ∈ ?F'
    then show v ∈ bound-tree S B
      by (rule fp-next-bound-tree[OF Suc.prems(1,2,3)])
  qed
  have W'-bound: ?W' ⊆ bound-tree S B
    using Suc.prems(4) F'-bound by blast
  show ?case
  proof (cases card ?W' > cap)
    case True
    then show ?thesis
      using W'-bound by (simp add: fp-seen-def Let-def)
  next
    case False
    show ?thesis
      using Suc.IH[OF sound' pre' F'-bound W'-bound] False
      by (simp add: fp-seen-def Let-def)
  qed
qed

lemma find-pivots-seen-capped-subset-bound-tree:
  assumes sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and below: ⋀x. x ∈ S ⟹ below-bound (d x) B
  shows find-pivots-seen-capped k cap d S B ⊆ bound-tree S B
proof −
  have S-subset: S ⊆ V
    using pre unfolding bmssp-pre-full-def by blast
  have below-dist: ⋀x. x ∈ S ⟹ below-bound (dist s x) B
  proof −
    fix x
    assume xS: x ∈ S
    have dist s x ≤ d x
      using sound S-subset xS S-reaches[OF xS]
      unfolding sound-label-def by blast
    then show below-bound (dist s x) B
      by (rule below-bound-le-trans[OF - below[OF xS]])
  qed
  have S-bound: S ⊆ bound-tree S B
    by (rule sources-subset-bound-tree[OF S-reaches below-dist])
  show ?thesis
    unfolding find-pivots-seen-capped-def
    by (rule fp-iter-capped-seen-subset-bound-tree
      [OF sound pre S-bound S-bound])
qed
```

The next block connects FindPivots' "seen" set to completed output

ranges. When a non-base call succeeds with the original bound, the final assembly theorem says that the loop output together with already-complete labelled vertices is exactly the parent bound tree. Since the seen set is contained in that tree, successful calls can charge the number of seen vertices to the returned output.

The same argument is stated for both exact split-range runs and direct-insert runs. The direct-insert version first forgets its extra accounting structure down to the exact range trace, then applies the same assembly reasoning. These lemmas are later paired with threshold lemmas for the unsuccessful branch.

```
theorem exact-split-range-costed-step-seen-success:
  assumes loop:
      exact-split-range-costed-partition-loop-state Δ M-of t h k cap l
        (find-pivots-label-capped k cap d S B)
        (find-pivots-pivots-capped k cap d S B) B d' D a betas bs B'
        Us U-loop c-loop child-costs
    and U-def:
      U = U-loop ∪
        {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v =
          dist s v}
    and sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and below: ⋀x. x ∈ S ⟹ below-bound (d x) B
    and success: B' = B
  shows card (find-pivots-seen-capped k cap d S B) ≤ card U
proof −
  let ?d-fp = find-pivots-label-capped k cap d S B
  let ?P = find-pivots-pivots-capped k cap d S B
  have sound-fp: sound-label ?d-fp
    unfolding find-pivots-label-capped-def
    by (rule fp-iter-capped-label-sound[OF sound S-reaches])
  have pivot-pre: bmssp-pre-full ?d-fp ?P B
    using find-pivots-capped-establishes-pivot-pre-concrete
      [OF sound pre S-reaches] .
  have P-subset-S: ?P ⊆ S
    unfolding find-pivots-pivots-capped-def by (auto split: if-splits)
  have P-reaches: ⋀x. x ∈ ?P ⟹ reachable s x
    using P-subset-S S-reaches by blast
  have split-loop:
    split-range-costed-partition-loop-state Δ M-of t h k cap l
      ?d-fp ?P B d' D a betas bs B' Us U-loop c-loop child-costs
    by (rule exact-split-range-costed-refines-split-range-costed(1)[OF loop])
  have trace:
    concrete-partition-loop-trace ?P B a bs d' B' Us U-loop
    by (rule split-range-costed-partition-loop-state-trace
      [OF split-loop sound-fp pivot-pre P-reaches])
  have loop-post: bmssp-post-full ?d-fp ?P B d' B' U-loop
```

by (*rule concrete-partition-loop-trace-post*[*OF trace*])
have *assembly*:
U-loop ∪
{*v* ∈ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* =
dist s v} =
bound-tree S B'
by (*rule concrete-capped-find-pivots-final-tree-assembly*
[*OF sound pre S-reaches loop-post*])
have *U-tree*: *U* = *bound-tree S B*
using *U-def assembly success* **by** *simp*
have *seen-subset*:
find-pivots-seen-capped k cap d S B ⊆ *U*
using *find-pivots-seen-capped-subset-bound-tree*
[*OF sound pre S-reaches below*] *U-tree* **by** *simp*
have *finite-U*: *finite U*
unfolding *U-tree bound-tree-def* **using** *finite-V* **by** *auto*
show *?thesis*
by (*rule card-mono*[*OF finite-U seen-subset*])
qed

theorem *direct-insert-costed-step-seen-success*:
assumes *loop*:
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B'*
Us U-loop c-loop child-costs
and *U-def*:
U = *U-loop* ∪
{*v* ∈ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* =
dist s v}
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: ⋀*x*. *x* ∈ *S* ⟹ *reachable s x*
and *below*: ⋀*x*. *x* ∈ *S* ⟹ *below-bound* (*d x*) *B*
and *success*: *B'* = *B*
shows *card* (*find-pivots-seen-capped k cap d S B*) ≤ *card U*
proof −
let *?d-fp* = *find-pivots-label-capped k cap d S B*
let *?P* = *find-pivots-pivots-capped k cap d S B*
have *sound-fp*: *sound-label ?d-fp*
unfolding *find-pivots-label-capped-def*
by (*rule fp-iter-capped-label-sound*[*OF sound S-reaches*])
have *pivot-pre*: *bmssp-pre-full ?d-fp ?P B*
using *find-pivots-capped-establishes-pivot-pre-concrete*
[*OF sound pre S-reaches*] **.**
have *P-subset-S*: *?P* ⊆ *S*
unfolding *find-pivots-pivots-capped-def* **by** (*auto split*: *if-splits*)
have *P-reaches*: ⋀*x*. *x* ∈ *?P* ⟹ *reachable s x*
using *P-subset-S S-reaches* **by** *blast*

```
  have trace:
    concrete-partition-loop-trace ?P B a bs d' B' Us U-loop
    by (rule direct-insert-costed-partition-loop-state-trace
      [OF loop sound-fp pivot-pre P-reaches])
  have loop-post: bmssp-post-full ?d-fp ?P B d' B' U-loop
    by (rule concrete-partition-loop-trace-post[OF trace])
  have assembly:
    U-loop ∪
      {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v =
        dist s v} =
      bound-tree S B'
    by (rule concrete-capped-find-pivots-final-tree-assembly
      [OF sound pre S-reaches loop-post])
  have U-tree: U = bound-tree S B
    using U-def assembly success by simp
  have seen-subset:
    find-pivots-seen-capped k cap d S B ⊆ U
    using find-pivots-seen-capped-subset-bound-tree
      [OF sound pre S-reaches below] U-tree by simp
  have finite-U: finite U
    unfolding U-tree bound-tree-def using finite-V by auto
  show ?thesis
    by (rule card-mono[OF finite-U seen-subset])
qed

theorem direct-insert-costed-step-seen-success-subset:
  assumes loop:
      direct-insert-costed-partition-loop-state Δ M-of t h k cap l
        (find-pivots-label-capped k cap d S B)
        (find-pivots-pivots-capped k cap d S B) B d' D a betas bs B'
        Us U-loop c-loop child-costs
    and U-def:
      U = U-loop ∪
        {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v =
          dist s v}
    and sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and below: ⋀x. x ∈ S ⟹ below-bound (d x) B
    and success: B' = B
  shows find-pivots-seen-capped k cap d S B ⊆ U
proof −
  let ?d-fp = find-pivots-label-capped k cap d S B
  let ?P = find-pivots-pivots-capped k cap d S B
  have sound-fp: sound-label ?d-fp
    unfolding find-pivots-label-capped-def
    by (rule fp-iter-capped-label-sound[OF sound S-reaches])
  have pivot-pre: bmssp-pre-full ?d-fp ?P B
    using find-pivots-capped-establishes-pivot-pre-concrete
```

```
    [OF sound pre S-reaches] .
  have P-subset-S: ?P ⊆ S
    unfolding find-pivots-pivots-capped-def by (auto split: if-splits)
  have P-reaches: ⋀x. x ∈ ?P ⟹ reachable s x
    using P-subset-S S-reaches by blast
  have trace:
    concrete-partition-loop-trace ?P B a bs d' B' Us U-loop
    by (rule direct-insert-costed-partition-loop-state-trace
      [OF loop sound-fp pivot-pre P-reaches])
  have loop-post: bmssp-post-full ?d-fp ?P B d' B' U-loop
    by (rule concrete-partition-loop-trace-post[OF trace])
  have assembly:
    U-loop ∪
      {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v =
        dist s v} =
      bound-tree S B'
    by (rule concrete-capped-find-pivots-final-tree-assembly
      [OF sound pre S-reaches loop-post])
  have U-tree: U = bound-tree S B
    using U-def assembly success by simp
  show ?thesis
    using find-pivots-seen-capped-subset-bound-tree
      [OF sound pre S-reaches below] U-tree by simp
qed

theorem direct-insert-costed-capped-step-threshold-if-not-success:
  assumes loop:
      direct-insert-costed-partition-loop-state Δ M-of t h k cap l
        (find-pivots-label-capped k cap d S B)
        (find-pivots-pivots-capped k cap d S B) B d' D a betas bs B'
        Us U-loop c-loop child-costs
    and U-def:
      U = U-loop ∪
        {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v =
          dist s v}
    and sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and not-success: B' ≠ B
  shows k * cap ≤ card U
proof −
  let ?d-fp = find-pivots-label-capped k cap d S B
  let ?P = find-pivots-pivots-capped k cap d S B
  have sound-fp: sound-label ?d-fp
    unfolding find-pivots-label-capped-def
    by (rule fp-iter-capped-label-sound[OF sound S-reaches])
  have pivot-pre: bmssp-pre-full ?d-fp ?P B
    using find-pivots-capped-establishes-pivot-pre-concrete
      [OF sound pre S-reaches] .
```

have *P-subset-S*: *?P* $\subseteq$ *S*
unfolding *find-pivots-pivots-capped-def* **by** *auto*
have *P-reaches*: $\bigwedge x.\ x \in ?P \Longrightarrow$ *reachable s x*
using *P-subset-S S-reaches* **by** *blast*
have *loop-class*:
$B' = B \vee k * cap \leq$ *card U-loop*
by (*rule direct-insert-costed-partition-loop-state-success-or-threshold*
[*OF loop sound-fp pivot-pre P-reaches*])
have *threshold-loop*: $k * cap \leq$ *card U-loop*
using *loop-class not-success* **by** *blast*
have *trace*: *concrete-partition-loop-trace ?P B a bs d' B' Us U-loop*
by (*rule direct-insert-costed-partition-loop-state-trace*
[*OF loop sound-fp pivot-pre P-reaches*])
have *loop-post*: *bmssp-post-full ?d-fp ?P B d' B' U-loop*
by (*rule concrete-partition-loop-trace-post*[*OF trace*])
have *finite-U-loop*: *finite U-loop*
using *loop-post* **unfolding** *bmssp-post-full-def* **by** *simp*
have *finite-U*: *finite U*
unfolding *U-def* **using** *finite-U-loop* **by** *simp*
have *U-loop-subset*: *U-loop* $\subseteq$ *U*
unfolding *U-def* **by** *blast*
have *card U-loop* $\leq$ *card U*
by (*rule card-mono*[*OF finite-U U-loop-subset*])
then show *?thesis*
using *threshold-loop* **by** *linarith*
qed

theorem *direct-insert-costed-capped-step-scan-insert-budget-from-scaled-seen-or-threshold*:
assumes *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *insert-factor*: $t \leq A * k$
and *insert-scaled-factor*: $t \leq A\text{-}insert * k$
and *seen-scaled-factor*: $k * \Delta + A\text{-}insert \leq 2 * A$
and *insert*:
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*)
and *loop*:
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B'*
Us U-loop c-loop child-costs
and *U-def*:
U = *U-loop* $\cup$
{*v* $\in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* =
dist s v}
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*

```
    and S-k-cap: k * card S ≤ cap
    and anti: tree-antichain S
    and k-pos: 0 < k
    and seen-success:
      B' = B ⟹
       card (find-pivots-seen-capped k cap d S B) ≤ card U
  shows fp-iter-capped-scan-cost k cap d S S B + c-insert ≤
    2 * A * card U
proof (cases B' = B)
  case True
  have S-subset: S ⊆ V
    using pre unfolding bmssp-pre-full-def by blast
  show ?thesis
    by (rule find-pivots-scan-and-initial-insert-budget-from-seen-scaled
      [OF S-subset degree anti S-k-cap insert insert-scaled-factor
        seen-scaled-factor seen-success[OF True]])
next
  case False
  have S-subset: S ⊆ V
    using pre unfolding bmssp-pre-full-def by blast
  have S-cap: card S ≤ cap
  proof −
    have card S ≤ k * card S
      using k-pos by (cases k) simp-all
    then show ?thesis
      using S-k-cap by linarith
  qed
  have progress: k * cap ≤ card U
    by (rule direct-insert-costed-capped-step-threshold-if-not-success
      [OF loop U-def sound pre S-reaches False])
  show ?thesis
    by (rule find-pivots-scan-and-initial-insert-budget-from-progress
      [OF S-subset degree S-cap insert degree-factor insert-factor progress])
qed

theorem split-range-costed-bmssp-zero-source-card-le-from-label-below:
  assumes run:
      split-range-costed-bmssp Δ M-of t h k cap 0 d S B d' B' U c
    and sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and below: ⋀x. x ∈ S ⟹ below-bound (d x) B
    and k-pos: 0 < k
  shows card S ≤ card U
  using run
proof cases
  case (Split-Range-Cost-Base x)
  have S-eq: S = {x}
    using Split-Range-Cost-Base by simp
```

```
  have xS: x ∈ S
    using S-eq by simp
  have xV: x ∈ V
    using pre xS unfolding bmssp-pre-full-def by blast
  have reach-x: reachable s x
    by (rule S-reaches[OF xS])
  have dist-le: dist s x ≤ d x
    using sound xV reach-x unfolding sound-label-def by blast
  have below-dist: below-bound (dist s x) B
    using below-bound-le-trans[OF dist-le below[OF xS]] .
  have x-bound: x ∈ bound-tree {x} B
    by (rule source-in-own-bound-tree[OF - reach-x below-dist]) simp
  have xU: x ∈ U
    using source-in-base-case-vertices[OF x-bound k-pos]
    using Split-Range-Cost-Base by simp
  have singleton-subset: {x} ⊆ U
    using xU by blast
  have finite-U: finite U
    using Split-Range-Cost-Base by simp
  have card {x} ≤ card U
    by (rule card-mono[OF finite-U singleton-subset])
  then show ?thesis
    using S-eq by simp
qed

theorem split-range-costed-bmssp-source-card-le-from-label-below-all-levels:
  assumes run:
      split-range-costed-bmssp Δ M-of t h k cap l d S B d' B' U c
    and sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and below: ⋀x. x ∈ S ⟹ below-bound (d x) B
    and S-cap: card S ≤ cap
    and k-pos: 0 < k
  shows card S ≤ card U
proof (cases l)
  case 0
  have run0:
    split-range-costed-bmssp Δ M-of t h k cap 0 d S B d' B' U c
    using run 0 by simp
  show ?thesis
    by (rule split-range-costed-bmssp-zero-source-card-le-from-label-below
      [OF run0 sound pre S-reaches below k-pos])
next
  case (Suc l')
  have run-suc:
    split-range-costed-bmssp Δ M-of t h k cap (Suc l') d S B d' B' U c
    using run Suc by simp
  show ?thesis
```

by (*rule split-range-costed-bmssp-Suc-source-card-le-from-label-below*
[*OF run-suc sound pre S-reaches below S-cap k-pos*])
qed

theorem *exact-split-range-costed-bmssp-source-card-le-from-label-below-all-levels*:
assumes *run*:
exact-split-range-costed-bmssp Δ *M-of t h k cap l d S B d' B' U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *below*: $\bigwedge x.\ x \in S \Longrightarrow$ *below-bound* (*d x*) *B*
and *S-cap*: *card S* $\leq$ *cap*
and *k-pos*: *0* $<$ *k*
shows *card S* $\leq$ *card U*
by (*rule split-range-costed-bmssp-source-card-le-from-label-below-all-levels*
[*OF exact-split-range-costed-refines-split-range-costed*(*2*)[*OF run*]
sound pre S-reaches below S-cap k-pos])

theorem *direct-insert-costed-bmssp-zero-source-card-le-from-label-below*:
assumes *run*:
direct-insert-costed-bmssp Δ *M-of t h k cap 0 d S B d' B' U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *below*: $\bigwedge x.\ x \in S \Longrightarrow$ *below-bound* (*d x*) *B*
and *k-pos*: *0* $<$ *k*
shows *card S* $\leq$ *card U*
using *run*
proof *cases*
case (*Direct-Insert-Costed-Base x*)
have *S-eq*: $S = \{x\}$
using *Direct-Insert-Costed-Base* **by** *simp*
have *xS*: $x \in S$
using *S-eq* **by** *simp*
have *xV*: $x \in V$
using *pre xS* **unfolding** *bmssp-pre-full-def* **by** *blast*
have *reach-x*: *reachable s x*
by (*rule S-reaches*[*OF xS*])
have *dist-le*: *dist s x* $\leq$ *d x*
using *sound xV reach-x* **unfolding** *sound-label-def* **by** *blast*
have *below-dist*: *below-bound* (*dist s x*) *B*
using *below-bound-le-trans*[*OF dist-le below*[*OF xS*]] **.**
have *x-bound*: $x \in$ *bound-tree* $\{x\}$ *B*
by (*rule source-in-own-bound-tree*[*OF - reach-x below-dist*]) *simp*
have *xU*: $x \in U$
using *source-in-base-case-vertices*[*OF x-bound k-pos*]
using *Direct-Insert-Costed-Base* **by** *simp*
have *singleton-subset*: $\{x\} \subseteq U$
using *xU* **by** *blast*

```
  have finite-U: finite U
    using Direct-Insert-Costed-Base by simp
  have card {x} ≤ card U
    by (rule card-mono[OF finite-U singleton-subset])
  then show ?thesis
    using S-eq by simp
qed

theorem direct-insert-costed-bmssp-source-card-le-from-label-below-all-levels:
  assumes run:
      direct-insert-costed-bmssp Δ M-of t h k cap l d S B d' B' U c
    and sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and below: ⋀x. x ∈ S ⟹ below-bound (d x) B
    and S-cap: card S ≤ cap
    and k-pos: 0 < k
  shows card S ≤ card U
proof (cases l)
  case 0
  have run0:
    direct-insert-costed-bmssp Δ M-of t h k cap 0 d S B d' B' U c
    using run 0 by simp
  show ?thesis
    by (rule direct-insert-costed-bmssp-zero-source-card-le-from-label-below
      [OF run0 sound pre S-reaches below k-pos])
next
  case (Suc l')
  have run-suc:
    direct-insert-costed-bmssp Δ M-of t h k cap (Suc l') d S B d' B' U c
    using run Suc by simp
  show ?thesis
    by (rule direct-insert-costed-bmssp-Suc-source-card-le-from-label-below
      [OF run-suc sound pre S-reaches below S-cap k-pos])
qed
```

From here the theory closes the recurrence under the strict progress assumptions. The source-progress hypothesis in the direct-insert cost layer is discharged by a case split on the recursion level. At level zero, the strict base-case order gives the source in the returned base set. At positive levels, the success-or-threshold lemmas and the seen-set bound show that a child call's sources are dominated by its output.

The resulting loop-closing theorems are wrappers around the direct-insert cost lemmas from the previous file. Their purpose is to remove the abstract source-progress assumption and replace it by explicit invariants: bounded child source size, tree antichains, and the scaled cardinality condition that the BMSSP schedule maintains.

```
theorem direct-insert-costed-partition-loop-state-closes-level-bound-from-child-
```

```
bound-with-invariants-and-edge-ranges:
  assumes run:
    direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
      betas bs B' Us U c child-costs
    and sound: sound-label d
    and pre: bmssp-pre-full d P B
    and P-reaches: ⋀x. x ∈ P ⟹ reachable s x
    and P-k-cap: k * card P ≤ cap
    and P-anti: tree-antichain P
    and child-bound:
      ⋀c-child U-child S-child B-child d-child B-child'.
        ⟦direct-insert-costed-bmssp Δ M-of t h k cap l d S-child B-child
            d-child B-child' U-child c-child;
          bmssp-pre-full d S-child B-child;
          ⋀x. x ∈ S-child ⟹ reachable s x;
          card S-child ≤ M-of l;
          ⋀x. x ∈ S-child ⟹ below-bound (d x) B-child;
          k * card S-child ≤ cap;
          tree-antichain S-child⟧
        ⟹ c-child ≤ level-range-cost-bound A R L U-child
    and M-cap: M-of l ≤ cap
    and source-factor: Suc h ≤ A
    and k-pos: 0 < k
  shows c ≤
    A * Suc L * card U + R * card (outgoing-edges U) +
    t * sum-list
      (map card (range-tree-child-direct-edge-range-list P B a betas bs)) +
    h * sum-list
      (map card (range-tree-child-edge-range-list P a betas bs))
proof −
  have P-subset: P ⊆ V
    using pre unfolding bmssp-pre-full-def by blast
  have child-cost-bounds:
    list-all2 (λc-child U-child.
      c-child ≤ level-range-cost-bound A R L U-child)
      child-costs (range-tree-child-list P a bs)
    by (rule direct-insert-costed-partition-loop-state-child-cost-bounds-with-invariants
      [OF run P-subset P-k-cap P-anti child-bound])
  have source-progress:
    ⋀S-child B-child d-child B-child' U-child c-child.
      ⟦direct-insert-costed-bmssp Δ M-of t h k cap l d S-child B-child
          d-child B-child' U-child c-child;
        bmssp-pre-full d S-child B-child;
        ⋀x. x ∈ S-child ⟹ reachable s x;
        card S-child ≤ M-of l;
        ⋀x. x ∈ S-child ⟹ below-bound (d x) B-child⟧
      ⟹ card S-child ≤ card U-child
  proof −
    fix S-child B-child d-child B-child' U-child c-child
```

assume *child*:
direct-insert-costed-bmssp Δ *M-of t h k cap l d S-child B-child d-child B-child′ U-child c-child*
and *child-pre*: *bmssp-pre-full d S-child B-child*
and *child-reaches*: $\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*
and *child-card*: *card S-child* $\leq$ *M-of l*
and *child-below*: $\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *below-bound* (*d x*) *B-child*
have *child-cap*: *card S-child* $\leq$ *cap*
using *child-card M-cap* **by** *linarith*
show *card S-child* $\leq$ *card U-child*
by (*rule direct-insert-costed-bmssp-source-card-le-from-label-below-all-levels* [*OF child sound child-pre child-reaches child-below child-cap k-pos*])
qed
show *?thesis*
by (*rule direct-insert-costed-partition-loop-state-closes-level-bound-from-child-costs-and-edge-ranges* [*OF run sound pre P-reaches child-cost-bounds source-factor source-progress*])
qed

theorem *direct-insert-costed-partition-loop-state-closes-amortized-bound-from-child-bound-with-invariants*:
assumes *run*:
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a betas bs B′ Us U c child-costs*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*
and *P-k-cap*: $k * \mathit{card}\ P \leq \mathit{cap}$
and *P-anti*: *tree-antichain P*
and *child-bound*:
$\bigwedge$*c-child U-child S-child B-child d-child B-child′*.
⟦*direct-insert-costed-bmssp* Δ *M-of t h k cap l d S-child B-child d-child B-child′ U-child c-child*;
bmssp-pre-full d S-child B-child;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *below-bound* (*d x*) *B-child*;
$k * \mathit{card}\ \mathit{S\text{-}child} \leq \mathit{cap}$;
tree-antichain S-child⟧
$\Longrightarrow$ *c-child* $\leq$
bmssp-amortized-cost-bound A R h t q L B-child U-child
and *M-cap*: *M-of l* $\leq$ *cap*
and *source-factor*: *Suc h* $\leq$ *A*
and *k-pos*: $0 < k$
shows *c* $\leq$ *bmssp-amortized-cost-bound A R h t* (*Suc q*) (*Suc L*) *B U*
proof −
have *P-subset*: $P \subseteq V$
using *pre* **unfolding** *bmssp-pre-full-def* **by** *blast*

have *child-cost-bounds*:
list-all2 (λ*c-child UB*. *case UB of* (*U-child*, *B-child*) ⇒
c-child ≤ *bmssp-amortized-cost-bound A R h t q L B-child U-child*)
child-costs (*range-tree-child-bound-pair-list P a betas bs*)
by (*rule direct-insert-costed-partition-loop-state-child-amortized-cost-bounds-with-invariants*
[*OF run P-subset P-k-cap P-anti child-bound*])
have *source-progress*:
⋀*S-child B-child d-child B-child′ U-child c-child*.
⟦*direct-insert-costed-bmssp* Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child;
bmssp-pre-full d S-child B-child;
⋀*x*. *x* ∈ *S-child* ⟹ *reachable s x*;
card S-child ≤ *M-of l*;
⋀*x*. *x* ∈ *S-child* ⟹ *below-bound* (*d x*) *B-child*⟧
⟹ *card S-child* ≤ *card U-child*
proof −
fix *S-child B-child d-child B-child′ U-child c-child*
assume *child*:
direct-insert-costed-bmssp Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child
and *child-pre*: *bmssp-pre-full d S-child B-child*
and *child-reaches*: ⋀*x*. *x* ∈ *S-child* ⟹ *reachable s x*
and *child-card*: *card S-child* ≤ *M-of l*
and *child-below*: ⋀*x*. *x* ∈ *S-child* ⟹ *below-bound* (*d x*) *B-child*
have *child-cap*: *card S-child* ≤ *cap*
using *child-card M-cap* **by** *linarith*
show *card S-child* ≤ *card U-child*
by (*rule direct-insert-costed-bmssp-source-card-le-from-label-below-all-levels*
[*OF child sound child-pre child-reaches child-below child-cap k-pos*])
qed
show *?thesis*
by (*rule direct-insert-costed-partition-loop-state-closes-amortized-bound-from-child-costs*
[*OF run sound pre P-reaches child-cost-bounds source-factor*
source-progress])
qed

theorem *direct-insert-costed-nonbase-step-closes-amortized-bound-from-child-bound-with-invariants*:
assumes *loop*:
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U-loop c-loop child-costs
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: ⋀*x*. *x* ∈ *P* ⟹ *reachable s x*
and *P-k-cap*: *k* ∗ *card P* ≤ *cap*
and *P-anti*: *tree-antichain P*
and *child-bound*:

```
    ⋀c-child U-child S-child B-child d-child B-child'.
     ⟦direct-insert-costed-bmssp Δ M-of t h k cap l d S-child B-child
        d-child B-child' U-child c-child;
       bmssp-pre-full d S-child B-child;
       ⋀x. x ∈ S-child ⟹ reachable s x;
       card S-child ≤ M-of l;
       ⋀x. x ∈ S-child ⟹ below-bound (d x) B-child;
       k * card S-child ≤ cap;
       tree-antichain S-child⟧
      ⟹ c-child ≤
       bmssp-amortized-cost-bound A R h t q L B-child U-child
   and M-cap: M-of l ≤ cap
   and source-factor: Suc h ≤ A
   and k-pos: 0 < k
   and U-def: U = U-loop ∪ W
   and finite-U: finite U
   and scan-insert: c-scan-insert ≤ A * card U
   and c-def: c = c-scan-insert + c-loop
 shows c ≤
   bmssp-amortized-cost-bound A R h t (Suc q) (Suc (Suc L)) B U
proof −
 have loop-bound:
   c-loop ≤
     bmssp-amortized-cost-bound A R h t (Suc q) (Suc L) B U-loop
   by (rule direct-insert-costed-partition-loop-state-closes-amortized-bound-from-
child-bound-with-invariants
     [OF loop sound pre P-reaches P-k-cap P-anti child-bound M-cap
       source-factor k-pos])
 have U-loop-subset: U-loop ⊆ U
   using U-def by blast
 have card-loop-le: card U-loop ≤ card U
   by (rule card-mono[OF finite-U U-loop-subset])
 have outgoing-subset: outgoing-edges U-loop ⊆ outgoing-edges U
   by (rule outgoing-edges-mono[OF U-loop-subset])
 have finite-out-U: finite (outgoing-edges U)
   by simp
 have card-out-le: card (outgoing-edges U-loop) ≤ card (outgoing-edges U)
   by (rule card-mono[OF finite-out-U outgoing-subset])
 have range-subset:
   outgoing-edges-range U-loop 0 B ⊆ outgoing-edges-range U 0 B
   by (rule outgoing-edges-range-mono-sources[OF U-loop-subset])
 have card-range-le:
   card (outgoing-edges-range U-loop 0 B) ≤
     card (outgoing-edges-range U 0 B)
   by (rule card-mono[OF finite-outgoing-edges-range range-subset])
 have loop-to-U:
   c-loop ≤
     A * Suc L * card U +
     (R + Suc q * h) * card (outgoing-edges U) +
```

```
    t * card (outgoing-edges-range U 0 B)
  proof −
    have vterm:
      A * Suc L * card U-loop ≤ A * Suc L * card U
      using card-loop-le by simp
    have eterm:
      (R + Suc q * h) * card (outgoing-edges U-loop) ≤
       (R + Suc q * h) * card (outgoing-edges U)
      using card-out-le by simp
    have rterm:
      t * card (outgoing-edges-range U-loop 0 B) ≤
       t * card (outgoing-edges-range U 0 B)
      using card-range-le by simp
    show ?thesis
      using loop-bound vterm eterm rterm
      unfolding bmssp-amortized-cost-bound-def by linarith
  qed
  have c ≤
      A * card U +
      (A * Suc L * card U +
       (R + Suc q * h) * card (outgoing-edges U) +
       t * card (outgoing-edges-range U 0 B))
    using scan-insert loop-to-U c-def by linarith
  also have ... =
      A * Suc (Suc L) * card U +
      (R + Suc q * h) * card (outgoing-edges U) +
      t * card (outgoing-edges-range U 0 B)
    by (simp add: algebra-simps)
  finally show ?thesis
    unfolding bmssp-amortized-cost-bound-def .
qed

theorem direct-insert-costed-bmssp-amortized-bound-from-local-budgets-with-
invariants:
  assumes base-budget: Δ ≤ A
    and R-pos: 0 < R
    and source-factor: Suc h ≤ A
    and k-pos: 0 < k
    and M-cap: ⋀i. i ≤ l ⟹ M-of i ≤ cap
    and step-budget:
      ⋀l d S B D c-insert d' a betas bs B' Us U-loop c-loop
         child-costs U.
        D = label-partition-view
          (find-pivots-label-capped k cap d S B)
          (find-pivots-pivots-capped k cap d S B) ⟹
        partition-initial-insert-cost-bound c-insert t
          (find-pivots-pivots-capped k cap d S B) ⟹
        direct-insert-costed-partition-loop-state Δ M-of t h k cap l
          (find-pivots-label-capped k cap d S B)
```

```
      (find-pivots-pivots-capped k cap d S B) B d' D a betas bs B'
      Us U-loop c-loop child-costs ⟹
    complete-on d'
     {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v} ⟹
    U = U-loop ∪
     {v ∈ bound-tree S B'.
       find-pivots-label-capped k cap d S B v = dist s v} ⟹
    sound-label d ⟹
    bmssp-pre-full d S B ⟹
    (⋀x. x ∈ S ⟹ reachable s x) ⟹
    (⋀x. x ∈ S ⟹ below-bound (d x) B) ⟹
    k * card S ≤ cap ⟹
    tree-antichain S ⟹
    fp-iter-capped-scan-cost k cap d S S B + c-insert ≤ A * card U
  and run:
    direct-insert-costed-bmssp Δ M-of t h k cap l d S B d' B' U c
  and sound: sound-label d
  and pre: bmssp-pre-full d S B
  and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
  and below: ⋀x. x ∈ S ⟹ below-bound (d x) B
  and S-k-cap: k * card S ≤ cap
  and S-anti: tree-antichain S
 shows c ≤ bmssp-amortized-cost-bound A R h t l (2 * l + 1) B U
 using run sound pre S-reaches below S-k-cap S-anti M-cap
proof (induction l arbitrary: d S B d' B' U c rule: less-induct)
 case (less l)
 show ?case
 proof (cases l)
  case 0
  have run0:
    direct-insert-costed-bmssp Δ M-of t h k cap 0 d S B d' B' U c
    using less.prems(1) 0 by simp
  show ?thesis
    using run0
  proof cases
    case (Direct-Insert-Costed-Base x)
    have scan:
      base-case-scan-cost Δ k x B ≤
        R * card (outgoing-edges (base-case-vertices k x B))
      using R-pos unfolding base-case-scan-cost-def by simp
    show ?thesis
      using Direct-Insert-Costed-Base 0 scan
      by (simp add: bmssp-amortized-cost-bound-def; linarith)
  qed
 next
  case (Suc l0)
  have run-suc:
    direct-insert-costed-bmssp Δ M-of t h k cap (Suc l0) d S B d' B' U c
    using less.prems(1) Suc by simp
```

show *?thesis*
proof (*cases rule*: *direct-insert-costed-bmssp-SucE*[*OF run-suc*])
case (*1 c-insert a betas bs Us U-loop c-loop child-costs-loop*)
let *?d-fp* = *find-pivots-label-capped k cap d S B*
let *?P* = *find-pivots-pivots-capped k cap d S B*
let *?W* = {*v* ∈ *bound-tree S B′*. *?d-fp v* = *dist s v*}
have *sound-fp*: *sound-label ?d-fp*
unfolding *find-pivots-label-capped-def*
by (*rule fp-iter-capped-label-sound*[*OF less.prems*(*2*) *less.prems*(*4*)])
have *pivot-pre*: *bmssp-pre-full ?d-fp ?P B*
using *find-pivots-capped-establishes-pivot-pre-concrete*
[*OF less.prems*(*2*) *less.prems*(*3*) *less.prems*(*4*)] **.**
have *S-subset*: *S* ⊆ *V*
using *less.prems*(*3*) **unfolding** *bmssp-pre-full-def* **by** *blast*
have *P-subset-S*: *?P* ⊆ *S*
unfolding *find-pivots-pivots-capped-def* **by** (*auto split*: *if-splits*)
have *P-reaches*: ⋀*x*. *x* ∈ *?P* ⟹ *reachable s x*
using *P-subset-S less.prems*(*4*) **by** *blast*
have *P-k-cap*: *k* ∗ *card ?P* ≤ *cap*
by (*rule find-pivots-pivots-capped-scaled-card-le*
[*OF S-subset less.prems*(*6*)])
have *P-anti*: *tree-antichain ?P*
by (*rule find-pivots-pivots-capped-tree-antichain*[*OF less.prems*(*7*)])
have *child-bound*:
⋀*c-child U-child S-child B-child d-child B-child′*.
⟦*direct-insert-costed-bmssp* Δ *M-of t h k cap l0 ?d-fp S-child B-child*
d-child B-child′ U-child c-child;
bmssp-pre-full ?d-fp S-child B-child;
⋀*x*. *x* ∈ *S-child* ⟹ *reachable s x*;
card S-child ≤ *M-of l0*;
⋀*x*. *x* ∈ *S-child* ⟹ *below-bound* (*?d-fp x*) *B-child*;
k ∗ *card S-child* ≤ *cap*;
tree-antichain S-child⟧
⟹ *c-child* ≤
bmssp-amortized-cost-bound A R h t l0 (*2* ∗ *l0* + *1*)
B-child U-child
proof −
fix *c-child U-child S-child B-child d-child B-child′*
assume *child*:
direct-insert-costed-bmssp Δ *M-of t h k cap l0 ?d-fp S-child B-child*
d-child B-child′ U-child c-child
and *child-pre*: *bmssp-pre-full ?d-fp S-child B-child*
and *child-reaches*: ⋀*x*. *x* ∈ *S-child* ⟹ *reachable s x*
and *child-below*:
⋀*x*. *x* ∈ *S-child* ⟹ *below-bound* (*?d-fp x*) *B-child*
and *child-k-cap*: *k* ∗ *card S-child* ≤ *cap*
and *child-anti*: *tree-antichain S-child*
have *M-cap-child*: ⋀*i*. *i* ≤ *l0* ⟹ *M-of i* ≤ *cap*
using *less.prems*(*8*) *Suc* **by** *simp*

```
      show c-child ≤
          bmssp-amortized-cost-bound A R h t l0 (2 * l0 + 1)
            B-child U-child
        by (rule less.IH[of l0 ?d-fp S-child B-child d-child B-child'
              U-child c-child, OF - child sound-fp child-pre child-reaches
              child-below child-k-cap child-anti M-cap-child])
          (simp add: Suc)
    qed
    have scan-insert:
      fp-iter-capped-scan-cost k cap d S S B + c-insert ≤
        A * card U
      by (rule step-budget[OF refl 1(3) 1(4) 1(5) 1(1)
          less.prems(2) less.prems(3) less.prems(4) less.prems(5)
          less.prems(6) less.prems(7)])
    have trace:
      concrete-partition-loop-trace ?P B a bs d' B' Us U-loop
      by (rule direct-insert-costed-partition-loop-state-trace
        [OF 1(4) sound-fp pivot-pre P-reaches])
    have loop-post: bmssp-post-full ?d-fp ?P B d' B' U-loop
      by (rule concrete-partition-loop-trace-post[OF trace])
    have finite-U-loop: finite U-loop
      using loop-post unfolding bmssp-post-full-def by simp
    have finite-W: finite ?W
      by simp
    have finite-U: finite U
      using 1(1) finite-U-loop finite-W by simp
    have M-cap-l0: M-of l0 ≤ cap
      using less.prems(8) Suc by simp
    have step:
      c ≤
        bmssp-amortized-cost-bound A R h t (Suc l0)
          (Suc (Suc (2 * l0 + 1))) B U
      by (rule direct-insert-costed-nonbase-step-closes-amortized-bound-from-child-
bound-with-invariants
          [OF 1(4) sound-fp pivot-pre P-reaches P-k-cap P-anti child-bound
            M-cap-l0 source-factor k-pos 1(1) finite-U scan-insert 1(2)])
    show ?thesis
      using step Suc by simp
  qed
 qed
qed

theorem direct-insert-costed-partition-loop-state-closes-level-bound-from-child-
bound-with-invariants-and-edge-budget:
 assumes run:
   direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
     betas bs B' Us U c child-costs
   and sound: sound-label d
   and pre: bmssp-pre-full d P B
```

and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*
and *P-k-cap*: $k * card\ P \leq cap$
and *P-anti*: *tree-antichain P*
and *child-bound*:
$\bigwedge$*c-child U-child S-child B-child d-child B-child'*.
⟦*direct-insert-costed-bmssp* Δ *M-of t h k cap l d S-child B-child d-child B-child' U-child c-child*;
bmssp-pre-full d S-child B-child;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *below-bound* (*d x*) *B-child*;
$k * card$ *S-child* $\leq$ *cap*;
tree-antichain S-child⟧
$\Longrightarrow$ *c-child* $\leq$ *level-range-cost-bound A R L U-child*
and *M-cap*: *M-of l* $\leq$ *cap*
and *source-factor*: *Suc h* $\leq$ *A*
and *k-pos*: $0 < k$
and *edge-budget*:
$t * sum\text{-}list$ (*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*)) +
$h * sum\text{-}list$ (*map card* (*range-tree-child-edge-range-list P a betas bs*))
$\leq H * card$ (*outgoing-edges U*)
shows $c \leq A * Suc\ L * card\ U + (R + H) * card$ (*outgoing-edges U*)
proof −
have *loop*:
$c \leq$
$A * Suc\ L * card\ U + R * card$ (*outgoing-edges U*) +
$t * sum\text{-}list$ (*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*)) +
$h * sum\text{-}list$ (*map card* (*range-tree-child-edge-range-list P a betas bs*))
by (*rule direct-insert-costed-partition-loop-state-closes-level-bound-from-child-bound-with-invariants-and-edge-ranges*
[*OF run sound pre P-reaches P-k-cap P-anti child-bound M-cap source-factor k-pos*])
have $c \leq A * Suc\ L * card\ U + R * card$ (*outgoing-edges U*) +
$H * card$ (*outgoing-edges U*)
using *loop edge-budget* **by** *linarith*
also have $\ldots = A * Suc\ L * card\ U + (R + H) * card$ (*outgoing-edges U*)
by (*simp add*: *algebra-simps*)
finally show *?thesis* **.**
qed

theorem *direct-insert-costed-nonbase-step-closes-level-bound-from-child-bound-with-invariants-and-edge-budget*:
assumes *loop*:
direct-insert-costed-partition-loop-state Δ *M-of t h k cap l d P B d' D a betas bs B' Us U-loop c-loop child-costs*

and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*
and *P-k-cap*: $k * card\ P \leq cap$
and *P-anti*: *tree-antichain P*
and *child-bound*:
$\bigwedge$*c-child U-child S-child B-child d-child B-child'*.
⟦*direct-insert-costed-bmssp* Δ *M-of t h k cap l d S-child B-child d-child B-child' U-child c-child*;
bmssp-pre-full d S-child B-child;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *below-bound* (*d x*) *B-child*;
$k * card$ *S-child* $\leq$ *cap*;
tree-antichain S-child⟧
$\Longrightarrow$ *c-child* $\leq$ *level-range-cost-bound A R L U-child*
and *M-cap*: *M-of l* $\leq$ *cap*
and *source-factor*: *Suc h* $\leq$ *A*
and *k-pos*: $0 < k$
and *edge-budget*:
$t * sum\text{-}list$ (*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*)) +
$h * sum\text{-}list$ (*map card* (*range-tree-child-edge-range-list P a betas bs*))
$\leq H * card$ (*outgoing-edges U-loop*)
and *U-def*: $U = U\text{-}loop \cup W$
and *finite-U*: *finite U*
and *scan-insert*: *c-scan-insert* $\leq A * card\ U$
and *c-def*: *c* = *c-scan-insert* + *c-loop*
shows $c \leq A * Suc\ (Suc\ L) * card\ U +$
$(R + H) * card$ (*outgoing-edges U*)
proof −
have *loop-bound*:
c-loop $\leq A * Suc\ L * card\ U\text{-}loop +$
$(R + H) * card$ (*outgoing-edges U-loop*)
by (*rule direct-insert-costed-partition-loop-state-closes-level-bound-from-child-bound-with-invariants-and-edge-budget*
[*OF loop sound pre P-reaches P-k-cap P-anti child-bound M-cap source-factor k-pos edge-budget*])
have *U-loop-subset*: *U-loop* $\subseteq$ *U*
using *U-def* **by** *blast*
have *card-loop-le*: *card U-loop* $\leq$ *card U*
by (*rule card-mono*[*OF finite-U U-loop-subset*])
have *outgoing-subset*: *outgoing-edges U-loop* $\subseteq$ *outgoing-edges U*
by (*rule outgoing-edges-mono*[*OF U-loop-subset*])
have *finite-out-U*: *finite* (*outgoing-edges U*)
by *simp*
have *card-out-le*:
card (*outgoing-edges U-loop*) $\leq$ *card* (*outgoing-edges U*)

```
    by (rule card-mono[OF finite-out-U outgoing-subset])
  have c ≤
      A * card U +
      (A * Suc L * card U + (R + H) * card (outgoing-edges U))
  proof −
    have loop-to-U:
      c-loop ≤ A * Suc L * card U +
        (R + H) * card (outgoing-edges U)
    proof −
      have A * Suc L * card U-loop ≤ A * Suc L * card U
        using card-loop-le by simp
      moreover have (R + H) * card (outgoing-edges U-loop) ≤
          (R + H) * card (outgoing-edges U)
        using card-out-le by simp
      ultimately show ?thesis
        using loop-bound by linarith
    qed
    show ?thesis
      using scan-insert loop-to-U c-def by linarith
  qed
  also have ... =
      A * Suc (Suc L) * card U + (R + H) * card (outgoing-edges U)
    by (simp add: algebra-simps)
  finally show ?thesis .
qed

theorem direct-insert-costed-bmssp-level-bound-from-local-budgets-with-invariants-
and-edge-budget:
  assumes base-budget: Δ ≤ A
    and R-pos: 0 < R
    and source-factor: Suc h ≤ A
    and k-pos: 0 < k
    and M-cap: ⋀i. i ≤ l ⟹ M-of i ≤ cap
    and edge-budget:
      ⋀l d P B d' D a betas bs B' Us U c child-costs.
        direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
          betas bs B' Us U c child-costs ⟹
        sound-label d ⟹
        bmssp-pre-full d P B ⟹
        (⋀x. x ∈ P ⟹ reachable s x) ⟹
        t * sum-list
          (map card (range-tree-child-direct-edge-range-list P B a betas bs)) +
        h * sum-list
          (map card (range-tree-child-edge-range-list P a betas bs))
          ≤ H * card (outgoing-edges U)
    and step-budget:
      ⋀l d S B D c-insert d' a betas bs B' Us U-loop c-loop
        child-costs U.
      D = label-partition-view
```

```
    (find-pivots-label-capped k cap d S B)
    (find-pivots-pivots-capped k cap d S B) ⟹
   partition-initial-insert-cost-bound c-insert t
    (find-pivots-pivots-capped k cap d S B) ⟹
   direct-insert-costed-partition-loop-state Δ M-of t h k cap l
    (find-pivots-label-capped k cap d S B)
    (find-pivots-pivots-capped k cap d S B) B d' D a betas bs B'
    Us U-loop c-loop child-costs ⟹
   complete-on d'
    {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v} ⟹
   U = U-loop ∪
    {v ∈ bound-tree S B'.
      find-pivots-label-capped k cap d S B v = dist s v} ⟹
   sound-label d ⟹
   bmssp-pre-full d S B ⟹
   (⋀x. x ∈ S ⟹ reachable s x) ⟹
   (⋀x. x ∈ S ⟹ below-bound (d x) B) ⟹
   k * card S ≤ cap ⟹
   tree-antichain S ⟹
   fp-iter-capped-scan-cost k cap d S S B + c-insert ≤ A * card U
 and run:
  direct-insert-costed-bmssp Δ M-of t h k cap l d S B d' B' U c
 and sound: sound-label d
 and pre: bmssp-pre-full d S B
 and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
 and below: ⋀x. x ∈ S ⟹ below-bound (d x) B
 and S-k-cap: k * card S ≤ cap
 and S-anti: tree-antichain S
shows c ≤ level-range-cost-bound A (R + l * H) (2 * l + 1) U
using run sound pre S-reaches below S-k-cap S-anti M-cap
proof (induction l arbitrary: d S B d' B' U c rule: less-induct)
 case (less l)
 show ?case
 proof (cases l)
  case 0
  have run0:
   direct-insert-costed-bmssp Δ M-of t h k cap 0 d S B d' B' U c
   using less.prems(1) 0 by simp
  show ?thesis
   using run0 R-pos 0
   by (cases rule: direct-insert-costed-bmssp.cases)
    (simp-all add: split-range-costed-base-level-bound)
 next
  case (Suc l0)
  have run-suc:
   direct-insert-costed-bmssp Δ M-of t h k cap (Suc l0) d S B d' B' U c
   using less.prems(1) Suc by simp
  show ?thesis
  proof (cases rule: direct-insert-costed-bmssp-SucE[OF run-suc])
```

case (*1 c-insert a betas bs Us U-loop c-loop child-costs-loop*)
let *?d-fp* = *find-pivots-label-capped k cap d S B*
let *?P* = *find-pivots-pivots-capped k cap d S B*
let *?W* = {*v* ∈ *bound-tree S B'*. *?d-fp v* = *dist s v*}
have *sound-fp*: *sound-label ?d-fp*
  unfolding *find-pivots-label-capped-def*
  by (*rule fp-iter-capped-label-sound*[*OF less.prems*(*2*) *less.prems*(*4*)])
have *pivot-pre*: *bmssp-pre-full ?d-fp ?P B*
  using *find-pivots-capped-establishes-pivot-pre-concrete*
    [*OF less.prems*(*2*) *less.prems*(*3*) *less.prems*(*4*)] **.**
have *S-subset*: *S* ⊆ *V*
  using *less.prems*(*3*) **unfolding** *bmssp-pre-full-def* **by** *blast*
have *P-subset-S*: *?P* ⊆ *S*
  unfolding *find-pivots-pivots-capped-def* **by** (*auto split*: *if-splits*)
have *P-reaches*: ⋀*x*. *x* ∈ *?P* ⟹ *reachable s x*
  using *P-subset-S less.prems*(*4*) **by** *blast*
have *P-k-cap*: *k* ∗ *card ?P* ≤ *cap*
  by (*rule find-pivots-pivots-capped-scaled-card-le*
    [*OF S-subset less.prems*(*6*)])
have *P-anti*: *tree-antichain ?P*
  by (*rule find-pivots-pivots-capped-tree-antichain*[*OF less.prems*(*7*)])
have *child-bound*:
  ⋀*c-child U-child S-child B-child d-child B-child'*.
    ⟦*direct-insert-costed-bmssp* Δ *M-of t h k cap l0 ?d-fp S-child B-child*
        d-child B-child' U-child c-child;
      bmssp-pre-full ?d-fp S-child B-child;
      ⋀*x*. *x* ∈ *S-child* ⟹ *reachable s x*;
      card S-child ≤ *M-of l0*;
      ⋀*x*. *x* ∈ *S-child* ⟹ *below-bound* (*?d-fp x*) *B-child*;
      k ∗ *card S-child* ≤ *cap*;
      tree-antichain S-child⟧
    ⟹ *c-child* ≤ *level-range-cost-bound A* (*R* + *l0* ∗ *H*)
      (*2* ∗ *l0* + *1*) *U-child*
proof −
  fix *c-child U-child S-child B-child d-child B-child'*
  assume *child*:
      direct-insert-costed-bmssp Δ *M-of t h k cap l0 ?d-fp S-child B-child*
        d-child B-child' U-child c-child
    and *child-pre*: *bmssp-pre-full ?d-fp S-child B-child*
    and *child-reaches*: ⋀*x*. *x* ∈ *S-child* ⟹ *reachable s x*
    and *child-below*:
      ⋀*x*. *x* ∈ *S-child* ⟹ *below-bound* (*?d-fp x*) *B-child*
    and *child-k-cap*: *k* ∗ *card S-child* ≤ *cap*
    and *child-anti*: *tree-antichain S-child*
  have *M-cap-child*: ⋀*i*. *i* ≤ *l0* ⟹ *M-of i* ≤ *cap*
    using *less.prems*(*8*) *Suc* **by** *simp*
  show *c-child* ≤ *level-range-cost-bound A* (*R* + *l0* ∗ *H*)
      (*2* ∗ *l0* + *1*) *U-child*
    by (*rule less.IH*[*of l0 ?d-fp S-child B-child d-child B-child'*

```
        U-child c-child, OF - child sound-fp child-pre child-reaches
        child-below child-k-cap child-anti M-cap-child])
    (simp add: Suc)
 qed
 have loop-edge-budget:
  t * sum-list
   (map card (range-tree-child-direct-edge-range-list ?P B a betas bs)) +
  h * sum-list
   (map card (range-tree-child-edge-range-list ?P a betas bs))
   ≤ H * card (outgoing-edges U-loop)
  by (rule edge-budget[OF 1(4) sound-fp pivot-pre P-reaches])
 have scan-insert:
  fp-iter-capped-scan-cost k cap d S S B + c-insert ≤
   A * card U
  by (rule step-budget[OF refl 1(3) 1(4) 1(5) 1(1)
     less.prems(2) less.prems(3) less.prems(4) less.prems(5)
     less.prems(6) less.prems(7)])
 have trace:
  concrete-partition-loop-trace ?P B a bs d' B' Us U-loop
  by (rule direct-insert-costed-partition-loop-state-trace
   [OF 1(4) sound-fp pivot-pre P-reaches])
 have loop-post: bmssp-post-full ?d-fp ?P B d' B' U-loop
  by (rule concrete-partition-loop-trace-post[OF trace])
 have finite-U-loop: finite U-loop
  using loop-post unfolding bmssp-post-full-def by simp
 have finite-W: finite ?W
  by simp
 have finite-U: finite U
  using 1(1) finite-U-loop finite-W by simp
 have M-cap-l0: M-of l0 ≤ cap
  using less.prems(8) Suc by simp
 have step:
  c ≤ A * Suc (Suc (2 * l0 + 1)) * card U +
   (R + l0 * H + H) * card (outgoing-edges U)
  by (rule direct-insert-costed-nonbase-step-closes-level-bound-from-child-bound-
with-invariants-and-edge-budget
    [OF 1(4) sound-fp pivot-pre P-reaches P-k-cap P-anti child-bound
     M-cap-l0 source-factor k-pos loop-edge-budget 1(1) finite-U
     scan-insert 1(2)])
 show ?thesis
  using step Suc unfolding level-range-cost-bound-def
  by (simp add: algebra-simps)
 qed
 qed
qed

theorem finite-initial-label-direct-insert-costed-top-level-correct-and-closed-refined-
graph-bound-level-cap-from-local-budgets:
 assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
```

and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *source-factor*: $Suc\ h \leq 2 * A$
and *k-pos*: $0 < k$
and *edge-budget*:
$\bigwedge l'\ d\ P\ B\ d'\ D\ a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop\ child\text{-}costs.$
direct-insert-costed-partition-loop-state Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) l' *d P B d' D a betas bs B'*
Us U-loop c-loop child-costs $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d P B $\Longrightarrow$
$(\bigwedge x.\ x \in P \Longrightarrow reachable\ s\ x) \Longrightarrow$
$t * sum\text{-}list$
(*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*)) +
$h * sum\text{-}list$
(*map card* (*range-tree-child-edge-range-list P a betas bs*))
$\leq H * card\ (outgoing\text{-}edges\ U\text{-}loop)$
and *step-budget*:
$\bigwedge l'\ d\ S\ B\ D\ c\text{-}insert\ d'\ a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop$
child-costs U.
$D = label\text{-}partition\text{-}view$
(*find-pivots-label-capped k* (*bmssp-level-cap k t l*) *d S B*)
(*find-pivots-pivots-capped k* (*bmssp-level-cap k t l*) *d S B*) $\Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k* (*bmssp-level-cap k t l*) *d S B*) $\Longrightarrow$
direct-insert-costed-partition-loop-state Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) l'
(*find-pivots-label-capped k* (*bmssp-level-cap k t l*) *d S B*)
(*find-pivots-pivots-capped k* (*bmssp-level-cap k t l*) *d S B*) *B d' D*
a betas bs B' Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d'
$\{v \in bound\text{-}tree\ S\ B'.$
find-pivots-label-capped k (*bmssp-level-cap k t l*) *d S B v* =
dist s v$\} \Longrightarrow$
$U = U\text{-}loop \cup$
$\{v \in bound\text{-}tree\ S\ B'.$
find-pivots-label-capped k (*bmssp-level-cap k t l*) *d S B v* =
dist s v$\} \Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow reachable\ s\ x) \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow below\text{-}bound\ (d\ x)\ B) \Longrightarrow$
$k * card\ S \leq bmssp\text{-}level\text{-}cap\ k\ t\ l \Longrightarrow$
tree-antichain S $\Longrightarrow$
fp-iter-capped-scan-cost k (*bmssp-level-cap k t l*) *d S S B* +
c-insert $\leq 2 * A * card\ U$
and *run*:
direct-insert-costed-bmssp Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) *l*

```
      finite-initial-label {s} Infinity d' Infinity U c
  shows sssp-correct d' ∧
    c ≤ 2 * A * (2 * l + 1) * vertex-count + (R + l * H) * edge-count
proof
  show sssp-correct d'
    by (rule finite-initial-label-direct-insert-costed-top-level-correct
      [OF all-reachable run])
next
  have pre: bmssp-pre-full finite-initial-label {s} Infinity
    using all-reachable finite-initial-label-source-complete
    by (rule top-bmssp-pre-full)
  have sound: sound-label finite-initial-label
    using finite-initial-label-sound[OF all-reachable] .
  have S-reaches: ⋀x. x ∈ {s} ⟹ reachable s x
    using all-reachable source-in-V by blast
  have below: ⋀x. x ∈ {s} ⟹
      below-bound (finite-initial-label x) Infinity
    by simp
  have top-cap: k * card {s} ≤ bmssp-level-cap k t l
  proof −
    have one-le: 1 ≤ bmssp-level-width t l
      unfolding bmssp-level-width-def by simp
    have k * 1 ≤ k * bmssp-level-width t l
      by (rule mult-left-mono[OF one-le]) simp
    then show ?thesis
      unfolding bmssp-level-cap-def by simp
  qed
  have top-anti: tree-antichain {s}
    by simp
  have range-bound:
    c ≤ level-range-cost-bound (2 * A) (R + l * H) (2 * l + 1) U
     proof (rule direct-insert-costed-bmssp-level-bound-from-local-budgets-with-
invariants-and-edge-budget
      [where A = 2 * A and R = R and H = H,
        OF - R-pos source-factor k-pos - - - run sound pre S-reaches below
          top-cap top-anti])
    show Δ ≤ 2 * A
      using degree-factor by simp
  next
    fix i
    assume i ≤ l
    then show bmssp-level-cap k t i ≤ bmssp-level-cap k t l
      by (rule bmssp-level-cap-mono)
  next
    fix l' d P B d' D a betas bs B' Us U-loop c-loop child-costs
    assume loop:
        direct-insert-costed-partition-loop-state Δ
          (bmssp-level-cap k t) t h k (bmssp-level-cap k t l) l'
          d P B d' D a betas bs B' Us U-loop c-loop child-costs
```

and *sound-l*: *sound-label d*
and *pre-l*: *bmssp-pre-full d P B*
and *reaches-l*: $\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*
show $t *$ *sum-list*
(*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*)) +
$h *$ *sum-list* (*map card* (*range-tree-child-edge-range-list P a betas bs*))
$\leq H *$ *card* (*outgoing-edges U-loop*)
by (*rule edge-budget*[*OF loop sound-l pre-l reaches-l*])
next
fix l' *d S B D c-insert d' a betas bs B' Us U-loop c-loop child-costs U*
assume *D-def*:
D = *label-partition-view*
(*find-pivots-label-capped k* (*bmssp-level-cap k t l*) *d S B*)
(*find-pivots-pivots-capped k* (*bmssp-level-cap k t l*) *d S B*)
and *insert*:
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k* (*bmssp-level-cap k t l*) *d S B*)
and *loop*:
direct-insert-costed-partition-loop-state Δ
(*bmssp-level-cap k t*) *t h k* (*bmssp-level-cap k t l*) l'
(*find-pivots-label-capped k* (*bmssp-level-cap k t l*) *d S B*)
(*find-pivots-pivots-capped k* (*bmssp-level-cap k t l*) *d S B*) *B d' D*
a betas bs B' Us U-loop c-loop child-costs
and *complete*:
complete-on d'
{*v* ∈ *bound-tree S B'*.
find-pivots-label-capped k (*bmssp-level-cap k t l*) *d S B v* =
dist s v}
and *U-def*:
U = *U-loop* ∪
{*v* ∈ *bound-tree S B'*.
find-pivots-label-capped k (*bmssp-level-cap k t l*) *d S B v* =
dist s v}
and *sound-s*: *sound-label d*
and *pre-s*: *bmssp-pre-full d S B*
and *reaches-s*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *below-s*: $\bigwedge x.\ x \in S \Longrightarrow$ *below-bound* (*d x*) *B*
and *S-k-cap*: $k * card\ S \leq$ *bmssp-level-cap k t l*
and *anti*: *tree-antichain S*
show *fp-iter-capped-scan-cost k* (*bmssp-level-cap k t l*) *d S S B* +
c-insert ≤ 2 $*$ *A* $*$ *card U*
by (*rule step-budget*[*OF D-def insert loop complete U-def sound-s pre-s reaches-s below-s S-k-cap anti*])
qed
have *post-full*:
bmssp-post-full finite-initial-label {*s*} *Infinity d' Infinity U*
by (*rule direct-insert-costed-bmssp-correct*[*OF run sound pre S-reaches*])
have *post*: *bmssp-post finite-initial-label* {*s*} *Infinity d' Infinity U*
using *bmssp-post-full-imp-post*[*OF post-full*] **.**

```
  have U-V: U = V
    using post bound-tree-source-infinity[OF all-reachable]
    unfolding bmssp-post-def by auto
  have graph-bound:
    level-range-cost-bound (2 * A) (R + l * H) (2 * l + 1) U
      ≤ 2 * A * (2 * l + 1) * vertex-count + (R + l * H) * edge-count
  proof −
    have U-card: card U = vertex-count
      unfolding U-V vertex-count-def by simp
    have out-le: card (outgoing-edges U) ≤ edge-count
      by (rule edge-count-outgoing-bound)
    have out-term:
      (R + l * H) * card (outgoing-edges U) ≤ (R + l * H) * edge-count
      using out-le by simp
    show ?thesis
      unfolding level-range-cost-bound-def
      using U-card out-term by (simp add: algebra-simps)
  qed
  show c ≤ 2 * A * (2 * l + 1) * vertex-count + (R + l * H) * edge-count
    using range-bound graph-bound by linarith
qed

corollary finite-initial-label-direct-insert-costed-top-level-correct-and-closed-bmssp-
refined-graph-time-bound-level-cap-from-local-budgets:
  assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
    and degree-factor: Δ ≤ A
    and R-pos: 0 < R
    and source-factor: Suc h ≤ 2 * A
    and k-pos: 0 < k
    and edge-budget:
      ⋀l' d P B d' D a betas bs B' Us U-loop c-loop child-costs.
        direct-insert-costed-partition-loop-state Δ (bmssp-level-cap k t) t h k
          (bmssp-level-cap k t l) l' d P B d' D a betas bs B'
          Us U-loop c-loop child-costs ⟹
        sound-label d ⟹
        bmssp-pre-full d P B ⟹
        (⋀x. x ∈ P ⟹ reachable s x) ⟹
        t * sum-list
          (map card (range-tree-child-direct-edge-range-list P B a betas bs)) +
        h * sum-list
          (map card (range-tree-child-edge-range-list P a betas bs))
          ≤ H * card (outgoing-edges U-loop)
    and step-budget:
      ⋀l' d S B D c-insert d' a betas bs B' Us U-loop c-loop
        child-costs U.
        D = label-partition-view
          (find-pivots-label-capped k (bmssp-level-cap k t l) d S B)
          (find-pivots-pivots-capped k (bmssp-level-cap k t l) d S B) ⟹
        partition-initial-insert-cost-bound c-insert t
```

(*find-pivots-pivots-capped k* (*bmssp-level-cap k t l*) *d S B*) ⟹
direct-insert-costed-partition-loop-state Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) *l'*
(*find-pivots-label-capped k* (*bmssp-level-cap k t l*) *d S B*)
(*find-pivots-pivots-capped k* (*bmssp-level-cap k t l*) *d S B*) *B d' D*
a betas bs B' Us U-loop c-loop child-costs ⟹
complete-on d'
{*v* ∈ *bound-tree S B'*.
find-pivots-label-capped k (*bmssp-level-cap k t l*) *d S B v* =
dist s v} ⟹
U = *U-loop* ∪
{*v* ∈ *bound-tree S B'*.
find-pivots-label-capped k (*bmssp-level-cap k t l*) *d S B v* =
dist s v} ⟹
sound-label d ⟹
bmssp-pre-full d S B ⟹
(⋀*x*. *x* ∈ *S* ⟹ *reachable s x*) ⟹
(⋀*x*. *x* ∈ *S* ⟹ *below-bound* (*d x*) *B*) ⟹
k ∗ *card S* ≤ *bmssp-level-cap k t l* ⟹
tree-antichain S ⟹
fp-iter-capped-scan-cost k (*bmssp-level-cap k t l*) *d S S B* +
c-insert ≤ 2 ∗ *A* ∗ *card U*
and *run*:
direct-insert-costed-bmssp Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) *l*
finite-initial-label {*s*} *Infinity d' Infinity U c*
shows *sssp-correct d'* ∧
c ≤ *bmssp-refined-graph-time-bound* (λ-. *A*) (λ-. *R*) (λ-. *H*)
(λ-. *l*) (λ-. *t*) (λ-. *edge-count*) *vertex-count*
proof −
have *graph*:
sssp-correct d' ∧
c ≤ 2 ∗ *A* ∗ (2 ∗ *l* + 1) ∗ *vertex-count* + (*R* + *l* ∗ *H*) ∗ *edge-count*
by (*rule finite-initial-label-direct-insert-costed-top-level-correct-and-closed-refined-graph-bound-level-cap-from-local-budgets*
[*OF all-reachable degree-factor R-pos source-factor k-pos edge-budget step-budget run*])
have *bound*:
2 ∗ *A* ∗ (2 ∗ *l* + 1) ∗ *vertex-count* + (*R* + *l* ∗ *H*) ∗ *edge-count*
≤ *bmssp-refined-graph-time-bound* (λ-. *A*) (λ-. *R*) (λ-. *H*)
(λ-. *l*) (λ-. *t*) (λ-. *edge-count*) *vertex-count*
unfolding *bmssp-refined-graph-time-bound-def* **by** *simp*
show *?thesis*
using *graph bound* **by** *linarith*
qed

theorem *finite-initial-label-direct-insert-costed-top-level-correct-and-closed-refined-graph-bound-level-cap*:
assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*

and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *insert-scaled-factor*: $t \leq A\text{-}insert * k$
and *seen-scaled-factor*: $k * \Delta + A\text{-}insert \leq 2 * A$
and *source-factor*: $Suc\ h \leq 2 * A$
and *k-pos*: $0 < k$
and *edge-budget*:
$\bigwedge l'\ d\ P\ B\ d'\ D\ a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop\ child\text{-}costs.$
direct-insert-costed-partition-loop-state Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) l' *d P B* d' *D a betas bs* B'
Us U-loop c-loop child-costs $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d P B $\Longrightarrow$
$(\bigwedge x.\ x \in P \Longrightarrow reachable\ s\ x) \Longrightarrow$
$t * sum\text{-}list$
(*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*)) +
$h * sum\text{-}list$
(*map card* (*range-tree-child-edge-range-list P a betas bs*))
$\leq H * card$ (*outgoing-edges U-loop*)
and *run*:
direct-insert-costed-bmssp Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) *l*
finite-initial-label {*s*} *Infinity* d' *Infinity U c*
shows *sssp-correct* $d' \wedge$
$c \leq 2 * A * (2 * l + 1) * vertex\text{-}count + (R + l * H) * edge\text{-}count$
proof (*rule finite-initial-label-direct-insert-costed-top-level-correct-and-closed-refined-graph-bound-level-cap-from-local-budgets*
[*OF all-reachable degree-factor R-pos source-factor k-pos edge-budget - run*])
fix l' *d S B D c-insert* d' *a betas bs* B' *Us U-loop c-loop*
child-costs U
assume *D-def*:
$D = label\text{-}partition\text{-}view$
(*find-pivots-label-capped k* (*bmssp-level-cap k t l*) *d S B*)
(*find-pivots-pivots-capped k* (*bmssp-level-cap k t l*) *d S B*)
and *insert*:
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k* (*bmssp-level-cap k t l*) *d S B*)
and *loop*:
direct-insert-costed-partition-loop-state Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) l'
(*find-pivots-label-capped k* (*bmssp-level-cap k t l*) *d S B*)
(*find-pivots-pivots-capped k* (*bmssp-level-cap k t l*) *d S B*) *B* d' *D*
a betas bs B' *Us U-loop c-loop child-costs*
and *complete*:
complete-on d'
{$v \in$ *bound-tree S* B'.
find-pivots-label-capped k (*bmssp-level-cap k t l*) *d S B v* =

```
        dist s v}
    and U-def:
      U = U-loop ∪
        {v ∈ bound-tree S B'.
          find-pivots-label-capped k (bmssp-level-cap k t l) d S B v =
            dist s v}
    and sound-s: sound-label d
    and pre-s: bmssp-pre-full d S B
    and reaches-s: ⋀x. x ∈ S ⟹ reachable s x
    and below-s: ⋀x. x ∈ S ⟹ below-bound (d x) B
    and S-k-cap: k * card S ≤ bmssp-level-cap k t l
    and anti: tree-antichain S
  have seen-success:
    B' = B ⟹
      card (find-pivots-seen-capped k (bmssp-level-cap k t l) d S B)
      ≤ card U
    by (rule direct-insert-costed-step-seen-success
      [OF loop U-def sound-s pre-s reaches-s below-s])
  show fp-iter-capped-scan-cost k (bmssp-level-cap k t l) d S S B +
      c-insert ≤ 2 * A * card U
    by (rule direct-insert-costed-capped-step-scan-insert-budget-from-scaled-seen-or-
threshold
      [OF degree degree-factor insert-factor insert-scaled-factor
        seen-scaled-factor insert loop U-def sound-s pre-s reaches-s
        S-k-cap anti k-pos seen-success])
qed

corollary finite-initial-label-direct-insert-costed-top-level-correct-and-closed-bmssp-
refined-graph-time-bound-level-cap:
  assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
    and degree: edge-outdegree-le Δ
    and degree-factor: Δ ≤ A
    and R-pos: 0 < R
    and insert-factor: t ≤ A * k
    and insert-scaled-factor: t ≤ A-insert * k
    and seen-scaled-factor: k * Δ + A-insert ≤ 2 * A
    and source-factor: Suc h ≤ 2 * A
    and k-pos: 0 < k
    and edge-budget:
      ⋀l' d P B d' D a betas bs B' Us U-loop c-loop child-costs.
        direct-insert-costed-partition-loop-state Δ (bmssp-level-cap k t) t h k
          (bmssp-level-cap k t l) l' d P B d' D a betas bs B'
          Us U-loop c-loop child-costs ⟹
        sound-label d ⟹
        bmssp-pre-full d P B ⟹
        (⋀x. x ∈ P ⟹ reachable s x) ⟹
        t * sum-list
          (map card (range-tree-child-direct-edge-range-list P B a betas bs)) +
        h * sum-list
```

```
      (map card (range-tree-child-edge-range-list P a betas bs))
      ≤ H * card (outgoing-edges U-loop)
  and run:
    direct-insert-costed-bmssp Δ (bmssp-level-cap k t) t h k
      (bmssp-level-cap k t l) l
      finite-initial-label {s} Infinity d' Infinity U c
 shows sssp-correct d' ∧
  c ≤ bmssp-refined-graph-time-bound (λ-. A) (λ-. R) (λ-. H)
    (λ-. l) (λ-. t) (λ-. edge-count) vertex-count
proof −
 have graph:
  sssp-correct d' ∧
    c ≤ 2 * A * (2 * l + 1) * vertex-count + (R + l * H) * edge-count
  by (rule finite-initial-label-direct-insert-costed-top-level-correct-and-closed-refined-
graph-bound-level-cap
    [OF all-reachable degree degree-factor R-pos insert-factor insert-scaled-factor
      seen-scaled-factor source-factor k-pos edge-budget run])
 have bound:
  2 * A * (2 * l + 1) * vertex-count + (R + l * H) * edge-count
    ≤ bmssp-refined-graph-time-bound (λ-. A) (λ-. R) (λ-. H)
      (λ-. l) (λ-. t) (λ-. edge-count) vertex-count
  unfolding bmssp-refined-graph-time-bound-def by simp
 show ?thesis
  using graph bound by linarith
qed

lemma direct-insert-costed-partition-loop-state-trivial-edge-budget:
 assumes run:
  direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
    betas bs B' Us U c child-costs
  and sound: sound-label d
  and pre: bmssp-pre-full d P B
  and P-reaches: ⋀x. x ∈ P ⟹ reachable s x
 shows t * sum-list
    (map card (range-tree-child-direct-edge-range-list P B a betas bs)) +
  h * sum-list (map card (range-tree-child-edge-range-list P a betas bs))
  ≤ (t + h) * card (outgoing-edges U)
proof −
 have mono: nondecreasing-from a bs
  by (rule direct-insert-costed-partition-loop-state-mono[OF run])
 have trace: concrete-partition-loop-trace P B a bs d' B' Us U
  by (rule direct-insert-costed-partition-loop-state-trace
    [OF run sound pre P-reaches])
 have children:
  list-all2 (λU X. U = X ∧ complete-on d' U) Us
    (range-tree-chain-list P a bs B')
  using trace unfolding concrete-partition-loop-trace-def by blast
 have U-def: U = bound-tree P (Fin a) ∪ ⋃(set Us)
  using trace unfolding concrete-partition-loop-trace-def by blast
```

have *Union-eq*:
$\bigcup$(*set Us*) = $\bigcup$(*set* (*range-tree-chain-list P a bs B'*))
using *children* **by** (*induction rule*: *list-all2-induct*) *auto*
have *U-eq-chain*:
U = *bound-tree P* (*Fin a*) ∪ *range-tree-chain P a bs B'*
using *U-def Union-eq Union-range-tree-chain-list*[*of P a bs B'*] **by** *simp*
have *chain-subset*: *range-tree-chain P a bs B'* ⊆ *U*
using *U-eq-chain* **by** *blast*
have *outgoing-subset*:
outgoing-edges (*range-tree-chain P a bs B'*) ⊆ *outgoing-edges U*
by (*rule outgoing-edges-mono*[*OF chain-subset*])
have *card-out-le*:
card (*outgoing-edges* (*range-tree-chain P a bs B'*)) ≤
card (*outgoing-edges U*)
by (*rule card-mono*) (*simp-all add*: *outgoing-subset*)
have *sum-le-chain*:
h ∗ *sum-list* (*map card* (*range-tree-child-edge-range-list P a betas bs*)) +
t ∗ *sum-list*
(*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*))
≤ (*h* + *t*) ∗ *card* (*outgoing-edges* (*range-tree-chain P a bs B'*))
by (*rule weighted-sum-child-lower-direct-edge-ranges-le-outgoing-edges-chain*
[*OF mono*])
have *chain-to-U*:
(*h* + *t*) ∗ *card* (*outgoing-edges* (*range-tree-chain P a bs B'*)) ≤
(*h* + *t*) ∗ *card* (*outgoing-edges U*)
using *card-out-le* **by** *simp*
show *?thesis*
using *sum-le-chain chain-to-U* **by** (*simp add*: *algebra-simps*)
qed

corollary *finite-initial-label-direct-insert-costed-top-level-correct-and-closed-bmssp-refined-graph-time-bound-level-cap-trivial-edge-budget*:
assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: Δ ≤ *A*
and *R-pos*: *0* < *R*
and *insert-factor*: *t* ≤ *A* ∗ *k*
and *insert-scaled-factor*: *t* ≤ *A-insert* ∗ *k*
and *seen-scaled-factor*: *k* ∗ Δ + *A-insert* ≤ *2* ∗ *A*
and *source-factor*: *Suc h* ≤ *2* ∗ *A*
and *k-pos*: *0* < *k*
and *run*:
direct-insert-costed-bmssp Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) *l*
finite-initial-label {*s*} *Infinity d' Infinity U c*
shows *sssp-correct d'* ∧
c ≤ *bmssp-refined-graph-time-bound* (λ-. *A*) (λ-. *R*) (λ-. *t* + *h*)
(λ-. *l*) (λ-. *t*) (λ-. *edge-count*) *vertex-count*
proof (*rule finite-initial-label-direct-insert-costed-top-level-correct-and-closed-bmssp-*

refined-graph-time-bound-level-cap
[*OF all-reachable degree degree-factor R-pos insert-factor insert-scaled-factor seen-scaled-factor source-factor k-pos - run*])
fix *l′ d P B d′ D a betas bs B′ Us U-loop c-loop child-costs*
assume *loop*:
direct-insert-costed-partition-loop-state Δ (*bmssp-level-cap k t*) *t h k* (*bmssp-level-cap k t l*) *l′ d P B d′ D a betas bs B′ Us U-loop c-loop child-costs*
and *sound-l*: *sound-label d*
and *pre-l*: *bmssp-pre-full d P B*
and *reaches-l*: ⋀*x*. *x* ∈ *P* ⟹ *reachable s x*
show *t* ∗ *sum-list* (*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*)) + *h* ∗ *sum-list* (*map card* (*range-tree-child-edge-range-list P a betas bs*)) ≤ (*t* + *h*) ∗ *card* (*outgoing-edges U-loop*)
by (*rule direct-insert-costed-partition-loop-state-trivial-edge-budget* [*OF loop sound-l pre-l reaches-l*])
qed

theorem *finite-initial-label-direct-insert-costed-top-level-correct-and-closed-bmssp-refined-graph-time-bound-level-cap-amortized*:
assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: Δ ≤ *A*
and *R-pos*: *0* < *R*
and *insert-factor*: *t* ≤ *A* ∗ *k*
and *insert-scaled-factor*: *t* ≤ *A-insert* ∗ *k*
and *seen-scaled-factor*: *k* ∗ Δ + *A-insert* ≤ *2* ∗ *A*
and *source-factor*: *Suc h* ≤ *2* ∗ *A*
and *k-pos*: *0* < *k*
and *run*:
direct-insert-costed-bmssp Δ (*bmssp-level-cap k t*) *t h k* (*bmssp-level-cap k t l*) *l finite-initial-label* {*s*} *Infinity d′ Infinity U c*
shows *sssp-correct d′* ∧
c ≤ *bmssp-refined-graph-time-bound* (λ-. *A*) (λ-. *R*) (λ-. *h*) (λ-. *l*) (λ-. *t*) (λ-. *edge-count*) *vertex-count*
proof
show *sssp-correct d′*
by (*rule finite-initial-label-direct-insert-costed-top-level-correct* [*OF all-reachable run*])
next
have *pre*: *bmssp-pre-full finite-initial-label* {*s*} *Infinity*
using *all-reachable finite-initial-label-source-complete*
by (*rule top-bmssp-pre-full*)
have *sound*: *sound-label finite-initial-label*
using *finite-initial-label-sound*[*OF all-reachable*] **.**
have *S-reaches*: ⋀*x*. *x* ∈ {*s*} ⟹ *reachable s x*
using *all-reachable source-in-V* **by** *blast*

```
  have below: ⋀x. x ∈ {s} ⟹
      below-bound (finite-initial-label x) Infinity
    by simp
  have top-cap: k * card {s} ≤ bmssp-level-cap k t l
  proof −
    have one-le: 1 ≤ bmssp-level-width t l
      unfolding bmssp-level-width-def by simp
    have k * 1 ≤ k * bmssp-level-width t l
      by (rule mult-left-mono[OF one-le]) simp
    then show ?thesis
      unfolding bmssp-level-cap-def by simp
  qed
  have top-anti: tree-antichain {s}
    by simp
  have amortized:
    c ≤ bmssp-amortized-cost-bound (2 * A) R h t l (2 * l + 1)
      Infinity U
  proof (rule direct-insert-costed-bmssp-amortized-bound-from-local-budgets-with-
invariants
      [where A = 2 * A and R = R,
        OF - R-pos source-factor k-pos - - run sound pre S-reaches below
          top-cap top-anti])
    show Δ ≤ 2 * A
      using degree-factor by simp
  next
    fix i
    assume i ≤ l
    then show bmssp-level-cap k t i ≤ bmssp-level-cap k t l
      by (rule bmssp-level-cap-mono)
  next
    fix l' d S B D c-insert d' a betas bs B' Us U-loop c-loop
      child-costs U
    assume D-def:
        D = label-partition-view
          (find-pivots-label-capped k (bmssp-level-cap k t l) d S B)
          (find-pivots-pivots-capped k (bmssp-level-cap k t l) d S B)
      and insert:
        partition-initial-insert-cost-bound c-insert t
          (find-pivots-pivots-capped k (bmssp-level-cap k t l) d S B)
      and loop:
        direct-insert-costed-partition-loop-state Δ (bmssp-level-cap k t) t h k
          (bmssp-level-cap k t l) l'
          (find-pivots-label-capped k (bmssp-level-cap k t l) d S B)
          (find-pivots-pivots-capped k (bmssp-level-cap k t l) d S B) B d' D
          a betas bs B' Us U-loop c-loop child-costs
      and complete:
        complete-on d'
          {v ∈ bound-tree S B'.
            find-pivots-label-capped k (bmssp-level-cap k t l) d S B v =
```

```
        dist s v}
    and U-def:
      U = U-loop ∪
        {v ∈ bound-tree S B'.
          find-pivots-label-capped k (bmssp-level-cap k t l) d S B v =
            dist s v}
    and sound-s: sound-label d
    and pre-s: bmssp-pre-full d S B
    and reaches-s: ⋀x. x ∈ S ⟹ reachable s x
    and below-s: ⋀x. x ∈ S ⟹ below-bound (d x) B
    and S-k-cap: k * card S ≤ bmssp-level-cap k t l
    and anti: tree-antichain S
  have seen-success:
    B' = B ⟹
      card (find-pivots-seen-capped k (bmssp-level-cap k t l) d S B)
      ≤ card U
    by (rule direct-insert-costed-step-seen-success
      [OF loop U-def sound-s pre-s reaches-s below-s])
  show fp-iter-capped-scan-cost k (bmssp-level-cap k t l) d S S B +
      c-insert ≤ 2 * A * card U
    by (rule direct-insert-costed-capped-step-scan-insert-budget-from-scaled-seen-or-
threshold
      [OF degree degree-factor insert-factor insert-scaled-factor
        seen-scaled-factor insert loop U-def sound-s pre-s reaches-s
        S-k-cap anti k-pos seen-success])
qed
have post-full:
  bmssp-post-full finite-initial-label {s} Infinity d' Infinity U
  by (rule direct-insert-costed-bmssp-correct[OF run sound pre S-reaches])
have post: bmssp-post finite-initial-label {s} Infinity d' Infinity U
  using bmssp-post-full-imp-post[OF post-full] .
have U-V: U = V
  using post bound-tree-source-infinity[OF all-reachable]
  unfolding bmssp-post-def by auto
have graph-bound:
  bmssp-amortized-cost-bound (2 * A) R h t l (2 * l + 1) Infinity U
    ≤ bmssp-refined-graph-time-bound (λ-. A) (λ-. R) (λ-. h)
      (λ-. l) (λ-. t) (λ-. edge-count) vertex-count
proof −
  have U-card: card U = vertex-count
    unfolding U-V vertex-count-def by simp
  have out-le: card (outgoing-edges U) ≤ edge-count
    by (rule edge-count-outgoing-bound)
  have range-le: card (outgoing-edges-range U 0 Infinity) ≤ edge-count
    by (rule card-outgoing-edges-range-le-edge-count)
  have out-term:
    (R + l * h) * card (outgoing-edges U) ≤
     (R + l * h) * edge-count
    using out-le by simp
```

```
    have range-term:
      t * card (outgoing-edges-range U 0 Infinity) ≤ t * edge-count
      using range-le by simp
    show ?thesis
      unfolding bmssp-amortized-cost-bound-def bmssp-refined-graph-time-bound-def
      using U-card out-term range-term by (simp add: algebra-simps; linarith)
  qed
  show c ≤ bmssp-refined-graph-time-bound (λ-. A) (λ-. R) (λ-. h)
      (λ-. l) (λ-. t) (λ-. edge-count) vertex-count
    using amortized graph-bound by linarith
qed
```

The remaining exact split-range results repeat the same source-progress closure without the direct-insert edge split. They are kept because some clients of the development may want the simpler range-costed interface, while the bucketed headline uses the more refined direct-insert and amortised path. The proof shape is the same: extract child calls, use strict progress to dominate child sources by child outputs, and close the level recurrence.

Having both interfaces documented makes the proof stack easier to audit. The exact split-range family is the clean mathematical recurrence, and the direct-insert family is the implementation-aware recurrence that exposes the bucketed partition's local costs.

```
theorem exact-split-range-costed-partition-loop-state-closes-level-bound-from-child-costs:
  assumes run:
    exact-split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
      betas bs B' Us U c child-costs
    and sound: sound-label d
    and pre: bmssp-pre-full d P B
    and P-reaches: ⋀x. x ∈ P ⟹ reachable s x
    and child-cost-bounds:
      list-all2 (λc-child U-child.
        c-child ≤ level-range-cost-bound A R L U-child)
        child-costs (range-tree-child-list P a bs)
    and M-cap: M-of l ≤ cap
    and source-factor: Suc h ≤ A
    and k-pos: 0 < k
  shows c ≤ A * Suc L * card U + (R + t) * card (outgoing-edges U)
proof −
  have split-run:
    split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
      betas bs B' Us U c child-costs
    using exact-split-range-costed-refines-split-range-costed(1)[OF run] .
  have trace: concrete-partition-loop-trace P B a bs d' B' Us U
    by (rule split-range-costed-partition-loop-state-trace
      [OF split-run sound pre P-reaches])
  have children:
    list-all2 (λU X. U = X ∧ complete-on d' U) Us
```

(*range-tree-chain-list P a bs B'*)
using *trace* **unfolding** *concrete-partition-loop-trace-def* **by** *blast*
have *U-def*: $U = \textit{bound-tree}\ P\ (\textit{Fin}\ a) \cup \bigcup(\textit{set}\ \textit{Us})$
using *trace* **unfolding** *concrete-partition-loop-trace-def* **by** *blast*
have *Union-eq*:
$\bigcup(\textit{set}\ \textit{Us}) = \bigcup(\textit{set}\ (\textit{range-tree-chain-list}\ P\ a\ bs\ B'))$
using *children* **by** (*induction rule*: *list-all2-induct*) *auto*
have *U-eq-chain*:
$U = \textit{bound-tree}\ P\ (\textit{Fin}\ a) \cup \textit{range-tree-chain}\ P\ a\ bs\ B'$
using *U-def Union-eq Union-range-tree-chain-list*[*of P a bs B'*] **by** *simp*
have *finite-U*: *finite U*
using *U-eq-chain* **by** *simp*
have *chain-subset*: $\textit{range-tree-chain}\ P\ a\ bs\ B' \subseteq U$
using *U-eq-chain* **by** *blast*
have *card-chain-le*: $\textit{card}\ (\textit{range-tree-chain}\ P\ a\ bs\ B') \leq \textit{card}\ U$
by (*rule card-mono*[*OF finite-U chain-subset*])
have *outgoing-subset*:
$\textit{outgoing-edges}\ (\textit{range-tree-chain}\ P\ a\ bs\ B') \subseteq \textit{outgoing-edges}\ U$
by (*rule outgoing-edges-mono*[*OF chain-subset*])
have *finite-out-U*: $\textit{finite}\ (\textit{outgoing-edges}\ U)$
by *simp*
have *card-out-le*:
$\textit{card}\ (\textit{outgoing-edges}\ (\textit{range-tree-chain}\ P\ a\ bs\ B')) \leq$
$\textit{card}\ (\textit{outgoing-edges}\ U)$
by (*rule card-mono*[*OF finite-out-U outgoing-subset*])
have *loop*:
$c \leq$
$A * L * \textit{card}\ (\textit{range-tree-chain}\ P\ a\ bs\ B') +$
$(R + t) * \textit{card}\ (\textit{outgoing-edges}\ (\textit{range-tree-chain}\ P\ a\ bs\ B')) +$
$\textit{Suc}\ h * \textit{card}\ (\textit{range-tree-chain}\ P\ a\ bs\ B')$
proof (*rule exact-split-range-costed-partition-loop-state-cost-from-child-and-source-bounds*
[*OF run child-cost-bounds*])
fix *S-child B-child d-child B-child' U-child c-child*
assume *child*:
exact-split-range-costed-bmssp Δ *M-of t h k cap l d S-child B-child*
d-child B-child' U-child c-child
and *child-pre*: *bmssp-pre-full d S-child B-child*
and *child-reaches*: $\bigwedge x.\ x \in \textit{S-child} \Longrightarrow \textit{reachable}\ s\ x$
and *child-card*: $\textit{card}\ \textit{S-child} \leq \textit{M-of}\ l$
and *child-below*: $\bigwedge x.\ x \in \textit{S-child} \Longrightarrow \textit{below-bound}\ (d\ x)\ \textit{B-child}$
have *child-cap*: $\textit{card}\ \textit{S-child} \leq \textit{cap}$
using *child-card M-cap* **by** *linarith*
show $\textit{card}\ \textit{S-child} \leq \textit{card}\ \textit{U-child}$
by (*rule exact-split-range-costed-bmssp-source-card-le-from-label-below-all-levels*
[*OF child sound child-pre child-reaches child-below child-cap k-pos*])
qed
have $c \leq$
$A * L * \textit{card}\ U + (R + t) * \textit{card}\ (\textit{outgoing-edges}\ U) + A * \textit{card}\ U$

```
  proof −
    have vterm:
      A * L * card (range-tree-chain P a bs B') ≤ A * L * card U
      using card-chain-le by simp
    have eterm:
      (R + t) * card (outgoing-edges (range-tree-chain P a bs B')) ≤
       (R + t) * card (outgoing-edges U)
      using card-out-le by simp
    have source-term:
      Suc h * card (range-tree-chain P a bs B') ≤ A * card U
    proof −
      have Suc h * card (range-tree-chain P a bs B') ≤
          A * card (range-tree-chain P a bs B')
        by (rule mult-right-mono[OF source-factor]) simp
      also have ... ≤ A * card U
        using card-chain-le by simp
      finally show ?thesis .
    qed
    show ?thesis
      using loop vterm eterm source-term by linarith
  qed
  also have ... =
      A * Suc L * card U + (R + t) * card (outgoing-edges U)
    by (simp add: algebra-simps)
  finally show ?thesis .
qed

theorem exact-split-range-costed-partition-loop-state-closes-level-bound-from-child-costs-and-edge-ranges:
  assumes run:
    exact-split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
      betas bs B' Us U c child-costs
    and sound: sound-label d
    and pre: bmssp-pre-full d P B
    and P-reaches: ⋀x. x ∈ P ⟹ reachable s x
    and child-cost-bounds:
      list-all2 (λc-child U-child.
        c-child ≤ level-range-cost-bound A R L U-child)
        child-costs (range-tree-child-list P a bs)
    and M-cap: M-of l ≤ cap
    and source-factor: Suc h ≤ A
    and k-pos: 0 < k
  shows c ≤
    A * Suc L * card U + R * card (outgoing-edges U) +
    t * sum-list (map card (range-tree-child-edge-range-list P a betas bs))
proof −
  have split-run:
    split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
      betas bs B' Us U c child-costs
```

```
    using exact-split-range-costed-refines-split-range-costed(1)[OF run] .
  have trace: concrete-partition-loop-trace P B a bs d' B' Us U
    by (rule split-range-costed-partition-loop-state-trace
      [OF split-run sound pre P-reaches])
  have children:
    list-all2 (λU X. U = X ∧ complete-on d' U) Us
      (range-tree-chain-list P a bs B')
    using trace unfolding concrete-partition-loop-trace-def by blast
  have U-def: U = bound-tree P (Fin a) ∪ ⋃(set Us)
    using trace unfolding concrete-partition-loop-trace-def by blast
  have Union-eq:
    ⋃(set Us) = ⋃(set (range-tree-chain-list P a bs B'))
    using children by (induction rule: list-all2-induct) auto
  have U-eq-chain:
    U = bound-tree P (Fin a) ∪ range-tree-chain P a bs B'
    using U-def Union-eq Union-range-tree-chain-list[of P a bs B'] by simp
  have finite-U: finite U
    using U-eq-chain by simp
  have chain-subset: range-tree-chain P a bs B' ⊆ U
    using U-eq-chain by blast
  have card-chain-le: card (range-tree-chain P a bs B') ≤ card U
    by (rule card-mono[OF finite-U chain-subset])
  have outgoing-subset:
    outgoing-edges (range-tree-chain P a bs B') ⊆ outgoing-edges U
    by (rule outgoing-edges-mono[OF chain-subset])
  have finite-out-U: finite (outgoing-edges U)
    by simp
  have card-out-le:
    card (outgoing-edges (range-tree-chain P a bs B')) ≤
      card (outgoing-edges U)
    by (rule card-mono[OF finite-out-U outgoing-subset])
  have loop:
    c ≤
      A * L * card (range-tree-chain P a bs B') +
      R * card (outgoing-edges (range-tree-chain P a bs B')) +
      t * sum-list (map card (range-tree-child-edge-range-list P a betas bs)) +
      Suc h * card (range-tree-chain P a bs B')
   proof (rule exact-split-range-costed-partition-loop-state-cost-from-child-source-
and-edge-range-bounds
      [OF run sound child-cost-bounds])
    fix S-child B-child d-child B-child' U-child c-child
    assume child:
        exact-split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
          d-child B-child' U-child c-child
      and child-pre: bmssp-pre-full d S-child B-child
      and child-reaches: ⋀x. x ∈ S-child ⟹ reachable s x
      and child-card: card S-child ≤ M-of l
      and child-below: ⋀x. x ∈ S-child ⟹ below-bound (d x) B-child
    have child-cap: card S-child ≤ cap
```

```
    using child-card M-cap by linarith
  show card S-child ≤ card U-child
    by (rule exact-split-range-costed-bmssp-source-card-le-from-label-below-all-levels
      [OF child sound child-pre child-reaches child-below child-cap k-pos])
qed
have c ≤
    A * L * card U + R * card (outgoing-edges U) +
    t * sum-list (map card (range-tree-child-edge-range-list P a betas bs)) +
    A * card U
proof −
  have vterm:
    A * L * card (range-tree-chain P a bs B′) ≤ A * L * card U
    using card-chain-le by simp
  have eterm:
    R * card (outgoing-edges (range-tree-chain P a bs B′)) ≤
     R * card (outgoing-edges U)
    using card-out-le by simp
  have source-term:
    Suc h * card (range-tree-chain P a bs B′) ≤ A * card U
  proof −
    have Suc h * card (range-tree-chain P a bs B′) ≤
        A * card (range-tree-chain P a bs B′)
      by (rule mult-right-mono[OF source-factor]) simp
    also have ... ≤ A * card U
      using card-chain-le by simp
    finally show ?thesis .
  qed
  show ?thesis
    using loop vterm eterm source-term by linarith
qed
also have ... =
    A * Suc L * card U + R * card (outgoing-edges U) +
    t * sum-list (map card (range-tree-child-edge-range-list P a betas bs))
  by (simp add: algebra-simps)
finally show ?thesis .
qed

theorem exact-split-range-costed-partition-loop-state-closes-level-bound-from-child-bound:
  assumes run:
    exact-split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d′ D a
      betas bs B′ Us U c child-costs
    and sound: sound-label d
    and pre: bmssp-pre-full d P B
    and P-reaches: ⋀x. x ∈ P ⟹ reachable s x
    and child-bound:
      ⋀c-child U-child S-child B-child d-child B-child′.
        ⟦exact-split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
            d-child B-child′ U-child c-child;
```

```
        bmssp-pre-full d S-child B-child;
        ⋀x. x ∈ S-child ⟹ reachable s x;
        card S-child ≤ M-of l;
        ⋀x. x ∈ S-child ⟹ below-bound (d x) B-child⟧
       ⟹ c-child ≤ level-range-cost-bound A R L U-child
   and M-cap: M-of l ≤ cap
   and source-factor: Suc h ≤ A
   and k-pos: 0 < k
 shows c ≤ A * Suc L * card U + (R + t) * card (outgoing-edges U)
proof −
 have child-cost-bounds:
   list-all2 (λc-child U-child.
     c-child ≤ level-range-cost-bound A R L U-child)
     child-costs (range-tree-child-list P a bs)
   by (rule exact-split-range-costed-partition-loop-state-child-cost-bounds
     [OF run child-bound])
 show ?thesis
  by (rule exact-split-range-costed-partition-loop-state-closes-level-bound-from-child-
costs
     [OF run sound pre P-reaches child-cost-bounds M-cap source-factor k-pos])
qed

theorem exact-split-range-costed-partition-loop-state-closes-level-bound-from-child-
bound-and-edge-ranges:
 assumes run:
   exact-split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d′ D a
     betas bs B′ Us U c child-costs
   and sound: sound-label d
   and pre: bmssp-pre-full d P B
   and P-reaches: ⋀x. x ∈ P ⟹ reachable s x
   and child-bound:
     ⋀c-child U-child S-child B-child d-child B-child′.
       ⟦exact-split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
           d-child B-child′ U-child c-child;
         bmssp-pre-full d S-child B-child;
         ⋀x. x ∈ S-child ⟹ reachable s x;
         card S-child ≤ M-of l;
         ⋀x. x ∈ S-child ⟹ below-bound (d x) B-child⟧
       ⟹ c-child ≤ level-range-cost-bound A R L U-child
   and M-cap: M-of l ≤ cap
   and source-factor: Suc h ≤ A
   and k-pos: 0 < k
 shows c ≤
   A * Suc L * card U + R * card (outgoing-edges U) +
   t * sum-list (map card (range-tree-child-edge-range-list P a betas bs))
proof −
 have child-cost-bounds:
   list-all2 (λc-child U-child.
     c-child ≤ level-range-cost-bound A R L U-child)
```

child-costs (*range-tree-child-list P a bs*)
by (*rule exact-split-range-costed-partition-loop-state-child-cost-bounds*
[*OF run child-bound*])
show *?thesis*
by (*rule exact-split-range-costed-partition-loop-state-closes-level-bound-from-child-costs-and-edge-ranges*
[*OF run sound pre P-reaches child-cost-bounds M-cap source-factor k-pos*])
qed

theorem *exact-split-range-costed-partition-loop-state-closes-level-bound-from-child-bound-with-invariants*:
assumes *run*:
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: ⋀*x*. *x* ∈ *P* ⟹ *reachable s x*
and *P-k-cap*: *k* ∗ *card P* ≤ *cap*
and *P-anti*: *tree-antichain P*
and *child-bound*:
⋀*c-child U-child S-child B-child d-child B-child′*.
⟦*exact-split-range-costed-bmssp* Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child;
bmssp-pre-full d S-child B-child;
⋀*x*. *x* ∈ *S-child* ⟹ *reachable s x*;
card S-child ≤ *M-of l*;
⋀*x*. *x* ∈ *S-child* ⟹ *below-bound* (*d x*) *B-child*;
k ∗ *card S-child* ≤ *cap*;
tree-antichain S-child⟧
⟹ *c-child* ≤ *level-range-cost-bound A R L U-child*
and *M-cap*: *M-of l* ≤ *cap*
and *source-factor*: *Suc h* ≤ *A*
and *k-pos*: *0* < *k*
shows *c* ≤ *A* ∗ *Suc L* ∗ *card U* + (*R* + *t*) ∗ *card* (*outgoing-edges U*)
proof −
have *P-subset*: *P* ⊆ *V*
using *pre* **unfolding** *bmssp-pre-full-def* **by** *blast*
have *child-cost-bounds*:
list-all2 (λ*c-child U-child*.
c-child ≤ *level-range-cost-bound A R L U-child*)
child-costs (*range-tree-child-list P a bs*)
by (*rule exact-split-range-costed-partition-loop-state-child-cost-bounds-with-invariants*
[*OF run P-subset P-k-cap P-anti child-bound*])
show *?thesis*
by (*rule exact-split-range-costed-partition-loop-state-closes-level-bound-from-child-costs*
[*OF run sound pre P-reaches child-cost-bounds M-cap source-factor k-pos*])
qed

theorem *exact-split-range-costed-nonbase-step-closes-level-bound-from-child-bound*:
assumes *loop*:
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U-loop c-loop child-costs
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: ⋀*x*. *x* ∈ *P* ⟹ *reachable s x*
and *child-bound*:
⋀*c-child U-child S-child B-child d-child B-child′*.
⟦*exact-split-range-costed-bmssp* Δ *M-of t h k cap l d S-child B-child*
d-child B-child′ U-child c-child;
bmssp-pre-full d S-child B-child;
⋀*x*. *x* ∈ *S-child* ⟹ *reachable s x*;
card S-child ≤ *M-of l*;
⋀*x*. *x* ∈ *S-child* ⟹ *below-bound* (*d x*) *B-child*⟧
⟹ *c-child* ≤ *level-range-cost-bound A R L U-child*
and *M-cap*: *M-of l* ≤ *cap*
and *source-factor*: *Suc h* ≤ *A*
and *k-pos*: *0* < *k*
and *U-def*: *U* = *U-loop* ∪ *W*
and *finite-U*: *finite U*
and *scan-insert*: *c-scan-insert* ≤ *A* ∗ *card U*
and *c-def*: *c* = *c-scan-insert* + *c-loop*
shows *c* ≤ *A* ∗ *Suc* (*Suc L*) ∗ *card U* +
(*R* + *t*) ∗ *card* (*outgoing-edges U*)
proof −
have *loop-bound*:
c-loop ≤ *A* ∗ *Suc L* ∗ *card U-loop* +
(*R* + *t*) ∗ *card* (*outgoing-edges U-loop*)
by (*rule exact-split-range-costed-partition-loop-state-closes-level-bound-from-child-bound*
[*OF loop sound pre P-reaches child-bound M-cap source-factor k-pos*])
have *U-loop-subset*: *U-loop* ⊆ *U*
using *U-def* **by** *blast*
have *card-loop-le*: *card U-loop* ≤ *card U*
by (*rule card-mono*[*OF finite-U U-loop-subset*])
have *outgoing-subset*: *outgoing-edges U-loop* ⊆ *outgoing-edges U*
by (*rule outgoing-edges-mono*[*OF U-loop-subset*])
have *finite-out-U*: *finite* (*outgoing-edges U*)
by *simp*
have *card-out-le*: *card* (*outgoing-edges U-loop*) ≤ *card* (*outgoing-edges U*)
by (*rule card-mono*[*OF finite-out-U outgoing-subset*])
have *c* ≤
A ∗ *card U* +
(*A* ∗ *Suc L* ∗ *card U* + (*R* + *t*) ∗ *card* (*outgoing-edges U*))
proof −
have *loop-to-U*:
c-loop ≤ *A* ∗ *Suc L* ∗ *card U* +

```
      (R + t) * card (outgoing-edges U)
    proof −
      have A * Suc L * card U-loop ≤ A * Suc L * card U
        using card-loop-le by simp
      moreover have (R + t) * card (outgoing-edges U-loop) ≤
          (R + t) * card (outgoing-edges U)
        using card-out-le by simp
      ultimately show ?thesis
        using loop-bound by linarith
    qed
    show ?thesis
      using scan-insert loop-to-U c-def by linarith
  qed
  also have ... =
      A * Suc (Suc L) * card U + (R + t) * card (outgoing-edges U)
    by (simp add: algebra-simps)
  finally show ?thesis .
qed

theorem exact-split-range-costed-nonbase-step-closes-level-bound-from-child-bound-
and-edge-ranges:
  assumes loop:
    exact-split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d′ D a
      betas bs B′ Us U-loop c-loop child-costs
    and sound: sound-label d
    and pre: bmssp-pre-full d P B
    and P-reaches: ⋀x. x ∈ P ⟹ reachable s x
    and child-bound:
      ⋀c-child U-child S-child B-child d-child B-child′.
        ⟦exact-split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
            d-child B-child′ U-child c-child;
          bmssp-pre-full d S-child B-child;
          ⋀x. x ∈ S-child ⟹ reachable s x;
          card S-child ≤ M-of l;
          ⋀x. x ∈ S-child ⟹ below-bound (d x) B-child⟧
        ⟹ c-child ≤ level-range-cost-bound A R L U-child
    and M-cap: M-of l ≤ cap
    and source-factor: Suc h ≤ A
    and k-pos: 0 < k
    and U-def: U = U-loop ∪ W
    and finite-U: finite U
    and scan-insert: c-scan-insert ≤ A * card U
    and c-def: c = c-scan-insert + c-loop
  shows c ≤ A * Suc (Suc L) * card U +
    R * card (outgoing-edges U) +
    t * sum-list (map card (range-tree-child-edge-range-list P a betas bs))
proof −
  have loop-bound:
    c-loop ≤
```

A ∗ Suc L ∗ card U-loop + R ∗ card (outgoing-edges U-loop) +
t ∗ sum-list (map card (range-tree-child-edge-range-list P a betas bs))
by (*rule exact-split-range-costed-partition-loop-state-closes-level-bound-from-child-bound-and-edge-ranges*
[OF loop sound pre P-reaches child-bound M-cap source-factor k-pos])
have *U-loop-subset*: *U-loop ⊆ U*
using *U-def* **by** *blast*
have *card-loop-le*: *card U-loop ≤ card U*
by (*rule card-mono[OF finite-U U-loop-subset]*)
have *outgoing-subset*: *outgoing-edges U-loop ⊆ outgoing-edges U*
by (*rule outgoing-edges-mono[OF U-loop-subset]*)
have *finite-out-U*: *finite (outgoing-edges U)*
by *simp*
have *card-out-le*: *card (outgoing-edges U-loop) ≤ card (outgoing-edges U)*
by (*rule card-mono[OF finite-out-U outgoing-subset]*)
have *c ≤*
A ∗ card U +
(A ∗ Suc L ∗ card U + R ∗ card (outgoing-edges U) +
t ∗ sum-list (map card (range-tree-child-edge-range-list P a betas bs)))
proof −
have *loop-to-U*:
c-loop ≤
A ∗ Suc L ∗ card U + R ∗ card (outgoing-edges U) +
t ∗ sum-list (map card (range-tree-child-edge-range-list P a betas bs))
proof −
have *A ∗ Suc L ∗ card U-loop ≤ A ∗ Suc L ∗ card U*
using *card-loop-le* **by** *simp*
moreover have *R ∗ card (outgoing-edges U-loop) ≤*
R ∗ card (outgoing-edges U)
using *card-out-le* **by** *simp*
ultimately show *?thesis*
using *loop-bound* **by** *linarith*
qed
show *?thesis*
using *scan-insert loop-to-U c-def* **by** *linarith*
qed
also have *. . . =*
A ∗ Suc (Suc L) ∗ card U + R ∗ card (outgoing-edges U) +
t ∗ sum-list (map card (range-tree-child-edge-range-list P a betas bs))
by (*simp add*: *algebra-simps*)
finally show *?thesis* **.**
qed

theorem *exact-split-range-costed-nonbase-step-closes-level-bound-from-child-bound-with-invariants*:
assumes *loop*:
exact-split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d′ D a
betas bs B′ Us U-loop c-loop child-costs
and *sound*: *sound-label d*

```
    and pre: bmssp-pre-full d P B
    and P-reaches: ⋀x. x ∈ P ⟹ reachable s x
    and P-k-cap: k * card P ≤ cap
    and P-anti: tree-antichain P
    and child-bound:
      ⋀c-child U-child S-child B-child d-child B-child'.
        ⟦exact-split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
            d-child B-child' U-child c-child;
          bmssp-pre-full d S-child B-child;
          ⋀x. x ∈ S-child ⟹ reachable s x;
          card S-child ≤ M-of l;
          ⋀x. x ∈ S-child ⟹ below-bound (d x) B-child;
          k * card S-child ≤ cap;
          tree-antichain S-child⟧
        ⟹ c-child ≤ level-range-cost-bound A R L U-child
    and M-cap: M-of l ≤ cap
    and source-factor: Suc h ≤ A
    and k-pos: 0 < k
    and U-def: U = U-loop ∪ W
    and finite-U: finite U
    and scan-insert: c-scan-insert ≤ A * card U
    and c-def: c = c-scan-insert + c-loop
  shows c ≤ A * Suc (Suc L) * card U +
    (R + t) * card (outgoing-edges U)
proof −
  have loop-bound:
    c-loop ≤ A * Suc L * card U-loop +
      (R + t) * card (outgoing-edges U-loop)
    by (rule exact-split-range-costed-partition-loop-state-closes-level-bound-from-child-
bound-with-invariants
      [OF loop sound pre P-reaches P-k-cap P-anti child-bound M-cap
        source-factor k-pos])
  have U-loop-subset: U-loop ⊆ U
    using U-def by blast
  have card-loop-le: card U-loop ≤ card U
    by (rule card-mono[OF finite-U U-loop-subset])
  have outgoing-subset: outgoing-edges U-loop ⊆ outgoing-edges U
    by (rule outgoing-edges-mono[OF U-loop-subset])
  have finite-out-U: finite (outgoing-edges U)
    by simp
  have card-out-le: card (outgoing-edges U-loop) ≤ card (outgoing-edges U)
    by (rule card-mono[OF finite-out-U outgoing-subset])
  have c ≤
      A * card U +
      (A * Suc L * card U + (R + t) * card (outgoing-edges U))
  proof −
    have loop-to-U:
      c-loop ≤ A * Suc L * card U +
        (R + t) * card (outgoing-edges U)
```

```
    proof −
      have A * Suc L * card U-loop ≤ A * Suc L * card U
        using card-loop-le by simp
      moreover have (R + t) * card (outgoing-edges U-loop) ≤
          (R + t) * card (outgoing-edges U)
        using card-out-le by simp
      ultimately show ?thesis
        using loop-bound by linarith
    qed
    show ?thesis
      using scan-insert loop-to-U c-def by linarith
  qed
  also have ... =
      A * Suc (Suc L) * card U + (R + t) * card (outgoing-edges U)
    by (simp add: algebra-simps)
  finally show ?thesis .
qed

theorem exact-split-range-costed-bmssp-level-bound-from-local-budgets:
  assumes base-budget: Δ ≤ A
    and R-pos: 0 < R
    and source-factor: Suc h ≤ A
    and k-pos: 0 < k
    and M-cap: ⋀i. i ≤ l ⟹ M-of i ≤ cap
    and step-budget:
      ⋀l d S B D c-insert d' a betas bs B' Us U-loop c-loop
          child-costs U.
        D = label-partition-view
          (find-pivots-label-capped k cap d S B)
          (find-pivots-pivots-capped k cap d S B) ⟹
        partition-initial-insert-cost-bound c-insert t
          (find-pivots-pivots-capped k cap d S B) ⟹
        exact-split-range-costed-partition-loop-state Δ M-of t h k cap l
          (find-pivots-label-capped k cap d S B)
          (find-pivots-pivots-capped k cap d S B) B d' D a betas bs B'
          Us U-loop c-loop child-costs ⟹
        complete-on d'
          {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v} ⟹
        U = U-loop ∪
          {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v} ⟹
        sound-label d ⟹
        bmssp-pre-full d S B ⟹
        (⋀x. x ∈ S ⟹ reachable s x) ⟹
        (⋀x. x ∈ S ⟹ below-bound (d x) B) ⟹
        fp-iter-capped-scan-cost k cap d S S B + c-insert ≤ A * card U
    and run:
      exact-split-range-costed-bmssp Δ M-of t h k cap l d S B d' B' U c
    and sound: sound-label d
    and pre: bmssp-pre-full d S B
```

```
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
    and below: ⋀x. x ∈ S ⟹ below-bound (d x) B
  shows c ≤ level-range-cost-bound A (R + l * t) (2 * l + 1) U
  using run sound pre S-reaches below M-cap
proof (induction l arbitrary: d S B d' B' U c rule: less-induct)
  case (less l)
  show ?case
  proof (cases l)
    case 0
    have run0:
      exact-split-range-costed-bmssp Δ M-of t h k cap 0 d S B d' B' U c
      using less.prems(1) 0 by simp
    have split-run0:
      split-range-costed-bmssp Δ M-of t h k cap 0 d S B d' B' U c
      using exact-split-range-costed-refines-split-range-costed(2)[OF run0] .
    show ?thesis
      using split-run0 R-pos 0
      by (cases rule: split-range-costed-bmssp.cases)
        (simp-all add: split-range-costed-base-level-bound)
  next
    case (Suc l0)
    have run-suc:
      exact-split-range-costed-bmssp Δ M-of t h k cap (Suc l0) d S B d' B' U c
      using less.prems(1) Suc by simp
    show ?thesis
    proof (cases rule: exact-split-range-costed-bmssp-SucE[OF run-suc])
      case (1 c-insert a betas bs Us U-loop c-loop child-costs-loop)
      let ?d-fp = find-pivots-label-capped k cap d S B
      let ?P = find-pivots-pivots-capped k cap d S B
      let ?W = {v ∈ bound-tree S B'. ?d-fp v = dist s v}
      have sound-fp: sound-label ?d-fp
        unfolding find-pivots-label-capped-def
        by (rule fp-iter-capped-label-sound[OF less.prems(2) less.prems(4)])
      have pivot-pre: bmssp-pre-full ?d-fp ?P B
        using find-pivots-capped-establishes-pivot-pre-concrete
          [OF less.prems(2) less.prems(3) less.prems(4)] .
      have P-subset-S: ?P ⊆ S
        unfolding find-pivots-pivots-capped-def by (auto split: if-splits)
      have P-reaches: ⋀x. x ∈ ?P ⟹ reachable s x
        using P-subset-S less.prems(4) by blast
      have child-bound:
        ⋀c-child U-child S-child B-child d-child B-child'.
          ⟦exact-split-range-costed-bmssp Δ M-of t h k cap l0 ?d-fp S-child B-child
              d-child B-child' U-child c-child;
            bmssp-pre-full ?d-fp S-child B-child;
            ⋀x. x ∈ S-child ⟹ reachable s x;
            card S-child ≤ M-of l0;
            ⋀x. x ∈ S-child ⟹ below-bound (?d-fp x) B-child⟧
          ⟹ c-child ≤ level-range-cost-bound A (R + l0 * t)
```

```
    (2 * l0 + 1) U-child
proof −
  fix c-child U-child S-child B-child d-child B-child'
  assume child:
      exact-split-range-costed-bmssp Δ M-of t h k cap l0 ?d-fp S-child B-child
        d-child B-child' U-child c-child
    and child-pre: bmssp-pre-full ?d-fp S-child B-child
    and child-reaches: ⋀x. x ∈ S-child ⟹ reachable s x
    and child-below: ⋀x. x ∈ S-child ⟹ below-bound (?d-fp x) B-child
  have M-cap-child: ⋀i. i ≤ l0 ⟹ M-of i ≤ cap
    using less.prems(6) Suc by simp
  show c-child ≤ level-range-cost-bound A (R + l0 * t)
      (2 * l0 + 1) U-child
    by (rule less.IH[of l0 ?d-fp S-child B-child d-child B-child'
          U-child c-child, OF - child sound-fp child-pre child-reaches
          child-below M-cap-child])
      (simp add: Suc)
qed
have scan-insert:
  fp-iter-capped-scan-cost k cap d S S B + c-insert ≤
    A * card U
  by (rule step-budget[OF refl 1(3) 1(4) 1(5) 1(1)
      less.prems(2) less.prems(3) less.prems(4) less.prems(5)])
have trace:
  concrete-partition-loop-trace ?P B a bs d' B' Us U-loop
proof −
  have split-loop:
    split-range-costed-partition-loop-state Δ M-of t h k cap l0
      ?d-fp ?P B d'
      (label-partition-view ?d-fp ?P) a betas bs B' Us U-loop c-loop
      child-costs-loop
    using exact-split-range-costed-refines-split-range-costed(1)[OF 1(4)]
      by simp
  show ?thesis
    by (rule split-range-costed-partition-loop-state-trace
      [OF split-loop sound-fp pivot-pre P-reaches])
qed
have loop-post: bmssp-post-full ?d-fp ?P B d' B' U-loop
  by (rule concrete-partition-loop-trace-post[OF trace])
have finite-U-loop: finite U-loop
  using loop-post unfolding bmssp-post-full-def by simp
have finite-W: finite ?W
  by simp
have finite-U: finite U
  using 1(1) finite-U-loop finite-W by simp
have M-cap-l0: M-of l0 ≤ cap
  using less.prems(6) Suc by simp
have step:
  c ≤ A * Suc (Suc (2 * l0 + 1)) * card U +
```

$(R + l0 * t + t) * card\ (outgoing\text{-}edges\ U)$
by (*rule exact-split-range-costed-nonbase-step-closes-level-bound-from-child-bound*
[*OF 1*(*4*) *sound-fp pivot-pre P-reaches child-bound M-cap-l0*
source-factor k-pos 1(*1*) *finite-U scan-insert 1*(*2*)])
show *?thesis*
using *step Suc* **unfolding** *level-range-cost-bound-def*
by (*simp add*: *algebra-simps*)
qed
qed
qed

theorem *exact-split-range-costed-bmssp-level-bound-from-local-budgets-with-invariants*:
assumes *base-budget*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *source-factor*: $Suc\ h \leq A$
and *k-pos*: $0 < k$
and *M-cap*: $\bigwedge i.\ i \leq l \Longrightarrow M\text{-}of\ i \leq cap$
and *step-budget*:
$\bigwedge l\ d\ S\ B\ D\ c\text{-}insert\ d'\ a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop$
child-costs U.
$D = label\text{-}partition\text{-}view$
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B'*
Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d'
$\{v \in bound\text{-}tree\ S\ B'.\ find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B\ v = dist\ s\ v\} \Longrightarrow$
$U = U\text{-}loop \cup$
$\{v \in bound\text{-}tree\ S\ B'.\ find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B\ v = dist\ s\ v\} \Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow reachable\ s\ x) \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow below\text{-}bound\ (d\ x)\ B) \Longrightarrow$
$k * card\ S \leq cap \Longrightarrow$
tree-antichain S $\Longrightarrow$
fp-iter-capped-scan-cost k cap d S S B + *c-insert* $\leq A * card\ U$
and *run*:
exact-split-range-costed-bmssp Δ *M-of t h k cap l d S B d' B' U c*
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S B*
and *S-reaches*: $\bigwedge x.\ x \in S \Longrightarrow reachable\ s\ x$
and *below*: $\bigwedge x.\ x \in S \Longrightarrow below\text{-}bound\ (d\ x)\ B$
and *S-k-cap*: $k * card\ S \leq cap$

```
    and S-anti: tree-antichain S
  shows c ≤ level-range-cost-bound A (R + l * t) (2 * l + 1) U
  using run sound pre S-reaches below S-k-cap S-anti M-cap
proof (induction l arbitrary: d S B d' B' U c rule: less-induct)
  case (less l)
  show ?case
  proof (cases l)
    case 0
    have run0:
      exact-split-range-costed-bmssp Δ M-of t h k cap 0 d S B d' B' U c
      using less.prems(1) 0 by simp
    have split-run0:
      split-range-costed-bmssp Δ M-of t h k cap 0 d S B d' B' U c
      using exact-split-range-costed-refines-split-range-costed(2)[OF run0] .
    show ?thesis
      using split-run0 R-pos 0
      by (cases rule: split-range-costed-bmssp.cases)
        (simp-all add: split-range-costed-base-level-bound)
  next
    case (Suc l0)
    have run-suc:
      exact-split-range-costed-bmssp Δ M-of t h k cap (Suc l0) d S B d' B' U c
      using less.prems(1) Suc by simp
    show ?thesis
    proof (cases rule: exact-split-range-costed-bmssp-SucE[OF run-suc])
      case (1 c-insert a betas bs Us U-loop c-loop child-costs-loop)
      let ?d-fp = find-pivots-label-capped k cap d S B
      let ?P = find-pivots-pivots-capped k cap d S B
      let ?W = {v ∈ bound-tree S B'. ?d-fp v = dist s v}
      have sound-fp: sound-label ?d-fp
        unfolding find-pivots-label-capped-def
        by (rule fp-iter-capped-label-sound[OF less.prems(2) less.prems(4)])
      have pivot-pre: bmssp-pre-full ?d-fp ?P B
        using find-pivots-capped-establishes-pivot-pre-concrete
          [OF less.prems(2) less.prems(3) less.prems(4)] .
      have S-subset: S ⊆ V
        using less.prems(3) unfolding bmssp-pre-full-def by blast
      have P-subset-S: ?P ⊆ S
        unfolding find-pivots-pivots-capped-def by (auto split: if-splits)
      have P-reaches: ⋀x. x ∈ ?P ⟹ reachable s x
        using P-subset-S less.prems(4) by blast
      have P-k-cap: k * card ?P ≤ cap
        by (rule find-pivots-pivots-capped-scaled-card-le
          [OF S-subset less.prems(6)])
      have P-anti: tree-antichain ?P
        by (rule find-pivots-pivots-capped-tree-antichain[OF less.prems(7)])
      have child-bound:
        ⋀c-child U-child S-child B-child d-child B-child'.
          ⟦exact-split-range-costed-bmssp Δ M-of t h k cap l0 ?d-fp S-child B-child
```

```
      d-child B-child' U-child c-child;
     bmssp-pre-full ?d-fp S-child B-child;
     ⋀x. x ∈ S-child ⟹ reachable s x;
     card S-child ≤ M-of l0;
     ⋀x. x ∈ S-child ⟹ below-bound (?d-fp x) B-child;
     k * card S-child ≤ cap;
     tree-antichain S-child⟧
   ⟹ c-child ≤ level-range-cost-bound A (R + l0 * t)
     (2 * l0 + 1) U-child
proof −
  fix c-child U-child S-child B-child d-child B-child'
  assume child:
      exact-split-range-costed-bmssp Δ M-of t h k cap l0 ?d-fp S-child B-child
        d-child B-child' U-child c-child
    and child-pre: bmssp-pre-full ?d-fp S-child B-child
    and child-reaches: ⋀x. x ∈ S-child ⟹ reachable s x
    and child-below:
      ⋀x. x ∈ S-child ⟹ below-bound (?d-fp x) B-child
    and child-k-cap: k * card S-child ≤ cap
    and child-anti: tree-antichain S-child
  have M-cap-child: ⋀i. i ≤ l0 ⟹ M-of i ≤ cap
    using less.prems(8) Suc by simp
  show c-child ≤ level-range-cost-bound A (R + l0 * t)
      (2 * l0 + 1) U-child
    by (rule less.IH[of l0 ?d-fp S-child B-child d-child B-child'
          U-child c-child, OF - child sound-fp child-pre child-reaches
          child-below child-k-cap child-anti M-cap-child])
      (simp add: Suc)
qed
have scan-insert:
  fp-iter-capped-scan-cost k cap d S S B + c-insert ≤
    A * card U
  by (rule step-budget[OF refl 1(3) 1(4) 1(5) 1(1)
      less.prems(2) less.prems(3) less.prems(4) less.prems(5)
      less.prems(6) less.prems(7)])
have trace:
  concrete-partition-loop-trace ?P B a bs d' B' Us U-loop
proof −
  have split-loop:
    split-range-costed-partition-loop-state Δ M-of t h k cap l0
      ?d-fp ?P B d'
      (label-partition-view ?d-fp ?P) a betas bs B' Us U-loop c-loop
      child-costs-loop
    using exact-split-range-costed-refines-split-range-costed(1)[OF 1(4)]
      by simp
  show ?thesis
    by (rule split-range-costed-partition-loop-state-trace
      [OF split-loop sound-fp pivot-pre P-reaches])
qed
```

have *loop-post*: *bmssp-post-full ?d-fp ?P B d′ B′ U-loop*
by (*rule concrete-partition-loop-trace-post*[*OF trace*])
have *finite-U-loop*: *finite U-loop*
using *loop-post* **unfolding** *bmssp-post-full-def* **by** *simp*
have *finite-W*: *finite ?W*
by *simp*
have *finite-U*: *finite U*
using *1*(*1*) *finite-U-loop finite-W* **by** *simp*
have *M-cap-l0*: *M-of l0* ≤ *cap*
using *less.prems*(*8*) *Suc* **by** *simp*
have *step*:
c ≤ *A* ∗ *Suc* (*Suc* (*2* ∗ *l0* + *1*)) ∗ *card U* +
(*R* + *l0* ∗ *t* + *t*) ∗ *card* (*outgoing-edges U*)
by (*rule exact-split-range-costed-nonbase-step-closes-level-bound-from-child-bound-with-invariants*
[*OF 1*(*4*) *sound-fp pivot-pre P-reaches P-k-cap P-anti child-bound*
M-cap-l0 source-factor k-pos 1(*1*) *finite-U scan-insert 1*(*2*)])
show *?thesis*
using *step Suc* **unfolding** *level-range-cost-bound-def*
by (*simp add*: *algebra-simps*)
qed
qed
qed

theorem *finite-initial-label-exact-split-range-costed-top-level-correct-and-local-budget-bound*:
assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
and *base-budget*: Δ ≤ *A*
and *R-pos*: *0* < *R*
and *source-factor*: *Suc h* ≤ *A*
and *k-pos*: *0* < *k*
and *M-cap*: ⋀*i*. *i* ≤ *l* ⟹ *M-of i* ≤ *cap*
and *step-budget*:
⋀*l d S B D c-insert d′ a betas bs B′ Us U-loop c-loop child-costs U*.
D = *label-partition-view*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) ⟹
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*) ⟹
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d′ D a betas bs B′*
Us U-loop c-loop child-costs ⟹
complete-on d′
{*v* ∈ *bound-tree S B′*. *find-pivots-label-capped k cap d S B v* = *dist s v*} ⟹
U = *U-loop* ∪
{*v* ∈ *bound-tree S B′*. *find-pivots-label-capped k cap d S B v* = *dist s v*} ⟹
sound-label d ⟹

bmssp-pre-full d S B $\Longrightarrow$
($\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*) $\Longrightarrow$
($\bigwedge x.\ x \in S \Longrightarrow$ *below-bound* (*d x*) *B*) $\Longrightarrow$
fp-iter-capped-scan-cost k cap d S S B + *c-insert* $\leq A * card\ U$
and *run*:
exact-split-range-costed-bmssp Δ *M-of t h k cap l*
finite-initial-label $\{s\}$ *Infinity d′ Infinity U c*
shows *sssp-correct d′* $\wedge$
$c \leq$ *level-range-cost-bound A* ($R + l * t$) ($2 * l + 1$) *U*
proof
show *sssp-correct d′*
by (*rule finite-initial-label-exact-split-range-costed-top-level-correct*
[*OF all-reachable run*])
next
have *pre*: *bmssp-pre-full finite-initial-label* $\{s\}$ *Infinity*
using *all-reachable finite-initial-label-source-complete*
by (*rule top-bmssp-pre-full*)
have *sound*: *sound-label finite-initial-label*
using *finite-initial-label-sound*[*OF all-reachable*] **.**
have *S-reaches*: $\bigwedge x.\ x \in \{s\} \Longrightarrow$ *reachable s x*
using *all-reachable source-in-V* **by** *blast*
have *below*: $\bigwedge x.\ x \in \{s\} \Longrightarrow$ *below-bound* (*finite-initial-label x*) *Infinity*
by *simp*
show $c \leq$ *level-range-cost-bound A* ($R + l * t$) ($2 * l + 1$) *U*
by (*rule exact-split-range-costed-bmssp-level-bound-from-local-budgets*
[*OF base-budget R-pos source-factor k-pos M-cap step-budget run sound pre*
S-reaches below])
qed

theorem *finite-initial-label-exact-split-range-costed-top-level-correct-and-local-budget-graph-bound*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *base-budget*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *source-factor*: *Suc h* $\leq A$
and *k-pos*: $0 < k$
and *M-cap*: $\bigwedge i.\ i \leq l \Longrightarrow$ *M-of i* $\leq$ *cap*
and *step-budget*:
$\bigwedge l\ d\ S\ B\ D$ *c-insert d′ a betas bs B′ Us U-loop c-loop*
child-costs U.
D = *label-partition-view*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d′ D a betas bs B′*
Us U-loop c-loop child-costs $\Longrightarrow$

complete-on d'
$\{v \in$ *bound-tree* $S\ B'$. *find-pivots-label-capped* k *cap* $d\ S\ B\ v =$ *dist* $s\ v\} \Longrightarrow$
$U =$ *U-loop* $\cup$
$\{v \in$ *bound-tree* $S\ B'$. *find-pivots-label-capped* k *cap* $d\ S\ B\ v =$ *dist* $s\ v\} \Longrightarrow$
sound-label $d \Longrightarrow$
bmssp-pre-full $d\ S\ B \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable* $s\ x) \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *below-bound* $(d\ x)\ B) \Longrightarrow$
fp-iter-capped-scan-cost k *cap* $d\ S\ S\ B$ + *c-insert* $\leq A * card\ U$
and *run*:
exact-split-range-costed-bmssp Δ *M-of* $t\ h\ k$ *cap* l
finite-initial-label $\{s\}$ *Infinity* d' *Infinity* $U\ c$
shows *sssp-correct* $d' \wedge$
$c \leq A * (2 * l + 1) *$ *vertex-count* $+ (R + l * t) *$ *edge-count*
proof −
have *combined*:
sssp-correct $d' \wedge$
$c \leq$ *level-range-cost-bound* $A\ (R + l * t)\ (2 * l + 1)\ U$
by (*rule finite-initial-label-exact-split-range-costed-top-level-correct-and-local-budget-bound*
[*OF all-reachable base-budget R-pos source-factor k-pos M-cap step-budget run*])
then have *correct*: *sssp-correct* d'
by *blast*
have *pre*: *bmssp-pre-full finite-initial-label* $\{s\}$ *Infinity*
using *all-reachable finite-initial-label-source-complete*
by (*rule top-bmssp-pre-full*)
have *sound*: *sound-label finite-initial-label*
using *finite-initial-label-sound*[*OF all-reachable*] **.**
have *S-reaches*: $\bigwedge x.\ x \in \{s\} \Longrightarrow$ *reachable* $s\ x$
using *all-reachable source-in-V* **by** *blast*
have *post-full*:
bmssp-post-full finite-initial-label $\{s\}$ *Infinity* d' *Infinity* U
by (*rule exact-split-range-costed-bmssp-correct*[*OF run sound pre S-reaches*])
have *post*: *bmssp-post finite-initial-label* $\{s\}$ *Infinity* d' *Infinity* U
using *bmssp-post-full-imp-post*[*OF post-full*] **.**
have *U-V*: $U = V$
using *post bound-tree-source-infinity*[*OF all-reachable*]
unfolding *bmssp-post-def* **by** *auto*
have *cost-U*:
$c \leq$ *level-range-cost-bound* $A\ (R + l * t)\ (2 * l + 1)\ U$
using *combined* **by** *blast*
have *graph-bound*:
level-range-cost-bound $A\ (R + l * t)\ (2 * l + 1)\ U$
$\leq A * (2 * l + 1) *$ *vertex-count* $+ (R + l * t) *$ *edge-count*
proof −
have *U-card*: *card* $U =$ *vertex-count*
unfolding *U-V vertex-count-def* **by** *simp*
have *out-le*: *card* (*outgoing-edges* U) $\leq$ *edge-count*
by (*rule edge-count-outgoing-bound*)

have *out-term*:
*(R + l * t) * card (outgoing-edges U) ≤ (R + l * t) * edge-count*
using *out-le* **by** *simp*
show *?thesis*
unfolding *level-range-cost-bound-def*
using *U-card out-term* **by** (*simp add*: *algebra-simps*)
qed
show *?thesis*
using *correct cost-U graph-bound* **by** *linarith*
qed

theorem *finite-initial-label-exact-split-range-costed-top-level-correct-and-scaled-seen-progress-graph-bound*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: *0 < R*
and *insert-factor*: $t \leq A * k$
and *insert-scaled-factor*: $t \leq$ *A-insert* $* k$
and *seen-scaled-factor*: $k * \Delta +$ *A-insert* $\leq 2 * A$
and *source-factor*: *Suc h* $\leq 2 * A$
and *k-pos*: *0 < k*
and *M-cap*: $\bigwedge i.\ i \leq l \Longrightarrow$ *M-of i ≤ cap*
and *step-scaled-seen-progress*:
$\bigwedge l\ d\ S\ B\ D$ *c-insert* d' *a betas bs* B' *Us U-loop c-loop child-costs U*.
D = label-partition-view
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B* d' *D a betas bs* B'
Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d'
$\{v \in$ *bound-tree S* B'. *find-pivots-label-capped k cap d S B v = dist s v*$\}$ $\Longrightarrow$
U = U-loop $\cup$
$\{v \in$ *bound-tree S* B'. *find-pivots-label-capped k cap d S B v = dist s v*$\}$ $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *below-bound* $(d\ x)\ B) \Longrightarrow$
$k *$ *card S* $\leq$ *cap* $\wedge$ *tree-antichain S* $\wedge$
$(B' = B \longrightarrow$
card (*find-pivots-seen-capped k cap d S B*) $\leq$ *card U*)
and *run*:
exact-split-range-costed-bmssp Δ *M-of t h k cap l*
finite-initial-label $\{s\}$ *Infinity* d' *Infinity U c*

shows *sssp-correct* $d' \wedge$
$c \leq 2 * A * (2 * l + 1) *$ *vertex-count* $+ (R + l * t) *$ *edge-count*
proof (*rule finite-initial-label-exact-split-range-costed-top-level-correct-and-local-budget-graph-bound*
[**where** $A = 2 * A$ **and** $R = R$,
OF all-reachable - R-pos source-factor k-pos M-cap - run])
show $\Delta \leq 2 * A$
using *degree-factor* **by** *simp*
next
fix *l d S B D c-insert d' a betas bs B' Us U-loop c-loop child-costs U*
assume *D-def*:
D = label-partition-view
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*)
and *insert*:
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*)
and *loop*:
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B'*
Us U-loop c-loop child-costs
and *complete*:
complete-on d'
{*v* ∈ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v = dist s v*}
and *U-def*:
U = U-loop ∪
{*v* ∈ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v = dist s v*}
and *sound-s*: *sound-label d*
and *pre-s*: *bmssp-pre-full d S B*
and *reaches-s*: $\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*
and *below-s*: $\bigwedge x.\ x \in S \Longrightarrow$ *below-bound* (*d x*) *B*
have *progress*:
$k *$ *card S* $\leq$ *cap* $\wedge$ *tree-antichain S* $\wedge$
($B' = B \longrightarrow$
card (*find-pivots-seen-capped k cap d S B*) $\leq$ *card U*)
by (*rule step-scaled-seen-progress*
[*OF D-def insert loop complete U-def sound-s pre-s reaches-s below-s*])
then have *S-k-cap*: $k *$ *card S* $\leq$ *cap*
by *blast*
from *progress* **have** *anti*: *tree-antichain S*
by *blast*
from *progress* **have** *seen-success*:
$B' = B \Longrightarrow$
card (*find-pivots-seen-capped k cap d S B*) $\leq$ *card U*
by *blast*
have *split-loop*:
split-range-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)

```
    (find-pivots-pivots-capped k cap d S B) B d' D a betas bs B'
    Us U-loop c-loop child-costs
  by (rule exact-split-range-costed-refines-split-range-costed(1)[OF loop])
 show fp-iter-capped-scan-cost k cap d S S B + c-insert ≤
    2 * A * card U
  by (rule split-range-costed-capped-step-scan-insert-budget-from-scaled-seen-or-
threshold
    [OF degree degree-factor insert-factor insert-scaled-factor
     seen-scaled-factor insert split-loop U-def sound-s pre-s reaches-s
     S-k-cap anti k-pos seen-success])
qed

theorem finite-initial-label-exact-split-range-costed-top-level-correct-and-scaled-seen-
progress-graph-bound-level-cap:
 assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
  and degree: edge-outdegree-le Δ
  and degree-factor: Δ ≤ A
  and R-pos: 0 < R
  and insert-factor: t ≤ A * k
  and insert-scaled-factor: t ≤ A-insert * k
  and seen-scaled-factor: k * Δ + A-insert ≤ 2 * A
  and source-factor: Suc h ≤ 2 * A
  and k-pos: 0 < k
  and step-scaled-seen-progress:
   ⋀l' d S B D c-insert d' a betas bs B' Us U-loop c-loop
     child-costs U.
    D = label-partition-view
     (find-pivots-label-capped k (bmssp-level-cap k t l) d S B)
     (find-pivots-pivots-capped k (bmssp-level-cap k t l) d S B) ⟹
    partition-initial-insert-cost-bound c-insert t
     (find-pivots-pivots-capped k (bmssp-level-cap k t l) d S B) ⟹
    exact-split-range-costed-partition-loop-state Δ (bmssp-level-cap k t) t h k
     (bmssp-level-cap k t l) l'
     (find-pivots-label-capped k (bmssp-level-cap k t l) d S B)
     (find-pivots-pivots-capped k (bmssp-level-cap k t l) d S B) B d' D
     a betas bs B' Us U-loop c-loop child-costs ⟹
    complete-on d'
     {v ∈ bound-tree S B'.
      find-pivots-label-capped k (bmssp-level-cap k t l) d S B v =
        dist s v} ⟹
    U = U-loop ∪
     {v ∈ bound-tree S B'.
      find-pivots-label-capped k (bmssp-level-cap k t l) d S B v =
        dist s v} ⟹
    sound-label d ⟹
    bmssp-pre-full d S B ⟹
    (⋀x. x ∈ S ⟹ reachable s x) ⟹
    (⋀x. x ∈ S ⟹ below-bound (d x) B) ⟹
    k * card S ≤ bmssp-level-cap k t l ∧ tree-antichain S ∧
```

```
      (B' = B ⟶
        card (find-pivots-seen-capped k (bmssp-level-cap k t l) d S B) ≤
          card U)
    and run:
      exact-split-range-costed-bmssp Δ (bmssp-level-cap k t) t h k
        (bmssp-level-cap k t l) l
        finite-initial-label {s} Infinity d' Infinity U c
  shows sssp-correct d' ∧
    c ≤ 2 * A * (2 * l + 1) * vertex-count + (R + l * t) * edge-count
proof (rule finite-initial-label-exact-split-range-costed-top-level-correct-and-scaled-
seen-progress-graph-bound
    [OF all-reachable degree degree-factor R-pos insert-factor insert-scaled-factor
      seen-scaled-factor source-factor k-pos - - run])
  fix i
  assume i ≤ l
  then show bmssp-level-cap k t i ≤ bmssp-level-cap k t l
    by (rule bmssp-level-cap-mono)
next
  fix l' d S B D c-insert d' a betas bs B' Us U-loop c-loop child-costs U
  assume D-def:
      D = label-partition-view
        (find-pivots-label-capped k (bmssp-level-cap k t l) d S B)
        (find-pivots-pivots-capped k (bmssp-level-cap k t l) d S B)
    and insert:
      partition-initial-insert-cost-bound c-insert t
        (find-pivots-pivots-capped k (bmssp-level-cap k t l) d S B)
    and loop:
      exact-split-range-costed-partition-loop-state Δ (bmssp-level-cap k t) t h k
        (bmssp-level-cap k t l) l'
        (find-pivots-label-capped k (bmssp-level-cap k t l) d S B)
        (find-pivots-pivots-capped k (bmssp-level-cap k t l) d S B) B d' D
        a betas bs B' Us U-loop c-loop child-costs
    and complete:
      complete-on d'
        {v ∈ bound-tree S B'.
          find-pivots-label-capped k (bmssp-level-cap k t l) d S B v =
            dist s v}
    and U-def:
      U = U-loop ∪
        {v ∈ bound-tree S B'.
          find-pivots-label-capped k (bmssp-level-cap k t l) d S B v =
            dist s v}
    and sound-s: sound-label d
    and pre-s: bmssp-pre-full d S B
    and reaches-s: ⋀x. x ∈ S ⟹ reachable s x
    and below-s: ⋀x. x ∈ S ⟹ below-bound (d x) B
  show k * card S ≤ bmssp-level-cap k t l ∧ tree-antichain S ∧
      (B' = B ⟶
        card (find-pivots-seen-capped k (bmssp-level-cap k t l) d S B)
```

$\leq$ *card U*)
by (*rule step-scaled-seen-progress*
[*OF D-def insert loop complete U-def sound-s pre-s reaches-s below-s*])
qed

theorem *finite-initial-label-exact-split-range-costed-top-level-correct-and-scaled-seen-success-graph-bound*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *insert-scaled-factor*: $t \leq$ *A-insert* $* k$
and *seen-scaled-factor*: $k * \Delta +$ *A-insert* $\leq 2 * A$
and *source-factor*: *Suc h* $\leq 2 * A$
and *k-pos*: $0 < k$
and *top-cap*: $k \leq$ *cap*
and *M-cap*: $\bigwedge i.\ i \leq l \Longrightarrow$ *M-of i* $\leq$ *cap*
and *seen-success*:
$\bigwedge l\ d\ S\ B\ D$ *c-insert d' a betas bs B' Us U-loop c-loop*
child-costs U.
D = *label-partition-view*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B'*
Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d'
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* = *dist s v*$\}$ $\Longrightarrow$
U = *U-loop* $\cup$
$\{v \in$ *bound-tree S B'*. *find-pivots-label-capped k cap d S B v* = *dist s v*$\}$ $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d S B $\Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable s x*$) \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *below-bound* (*d x*) *B*$) \Longrightarrow$
$k *$ *card S* $\leq$ *cap* $\Longrightarrow$
tree-antichain S $\Longrightarrow$
B' = *B* $\Longrightarrow$
card (*find-pivots-seen-capped k cap d S B*) $\leq$ *card U*
and *run*:
exact-split-range-costed-bmssp Δ *M-of t h k cap l*
finite-initial-label $\{s\}$ *Infinity d' Infinity U c*
shows *sssp-correct d'* $\wedge$
$c \leq 2 * A * (2 * l + 1) *$ *vertex-count* $+ (R + l * t) *$ *edge-count*
proof $-$
have *correct*: *sssp-correct d'*

by (*rule finite-initial-label-exact-split-range-costed-top-level-correct*
[*OF all-reachable run*])
have *pre*: *bmssp-pre-full finite-initial-label* {*s*} *Infinity*
using *all-reachable finite-initial-label-source-complete*
by (*rule top-bmssp-pre-full*)
have *sound*: *sound-label finite-initial-label*
using *finite-initial-label-sound*[*OF all-reachable*] **.**
have *S-reaches*: $\bigwedge x.\ x \in \{s\} \Longrightarrow$ *reachable s x*
using *all-reachable source-in-V* **by** *blast*
have *below*: $\bigwedge x.\ x \in \{s\} \Longrightarrow$
below-bound (*finite-initial-label x*) *Infinity*
by *simp*
have *S-k-cap*: $k * card\ \{s\} \leq cap$
using *top-cap* **by** *simp*
have *S-anti*: *tree-antichain* {*s*}
by *simp*
have *range-bound*:
$c \leq$ *level-range-cost-bound* $(2 * A)\ (R + l * t)\ (2 * l + 1)\ U$
proof (*rule exact-split-range-costed-bmssp-level-bound-from-local-budgets-with-invariants*
[**where** $A = 2 * A$ **and** $R = R$,
OF - R-pos source-factor k-pos M-cap - run sound pre S-reaches below S-k-cap S-anti])
show $\Delta \leq 2 * A$
using *degree-factor* **by** *simp*
next
fix *l d S B D c-insert d′ a betas bs B′ Us U-loop c-loop child-costs U*
assume *D-def*:
$D =$ *label-partition-view*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*)
and *insert*:
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*)
and *loop*:
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d′ D a betas bs B′*
Us U-loop c-loop child-costs
and *complete*:
complete-on d′
{$v \in$ *bound-tree S B′. find-pivots-label-capped k cap d S B v* $=$
dist s v}
and *U-def*:
$U = U\text{-}loop \cup$
{$v \in$ *bound-tree S B′. find-pivots-label-capped k cap d S B v* $=$
dist s v}
and *sound-s*: *sound-label d*
and *pre-s*: *bmssp-pre-full d S B*

```
    and reaches-s: ⋀x. x ∈ S ⟹ reachable s x
    and below-s: ⋀x. x ∈ S ⟹ below-bound (d x) B
    and S-k-cap: k * card S ≤ cap
    and anti: tree-antichain S
  have split-loop:
    split-range-costed-partition-loop-state Δ M-of t h k cap l
      (find-pivots-label-capped k cap d S B)
      (find-pivots-pivots-capped k cap d S B) B d' D a betas bs B'
      Us U-loop c-loop child-costs
    by (rule exact-split-range-costed-refines-split-range-costed(1)[OF loop])
  have seen-success-step:
    B' = B ⟹
      card (find-pivots-seen-capped k cap d S B) ≤ card U
    by (rule seen-success
      [OF D-def insert loop complete U-def sound-s pre-s reaches-s below-s
        S-k-cap anti])
  show fp-iter-capped-scan-cost k cap d S S B + c-insert ≤
      2 * A * card U
    by (rule split-range-costed-capped-step-scan-insert-budget-from-scaled-seen-or-
threshold
      [OF degree degree-factor insert-factor insert-scaled-factor
        seen-scaled-factor insert split-loop U-def sound-s pre-s reaches-s
        S-k-cap anti k-pos seen-success-step])
qed
have post-full:
  bmssp-post-full finite-initial-label {s} Infinity d' Infinity U
  by (rule exact-split-range-costed-bmssp-correct[OF run sound pre S-reaches])
have post: bmssp-post finite-initial-label {s} Infinity d' Infinity U
  using bmssp-post-full-imp-post[OF post-full] .
have U-V: U = V
  using post bound-tree-source-infinity[OF all-reachable]
  unfolding bmssp-post-def by auto
have graph-bound:
  level-range-cost-bound (2 * A) (R + l * t) (2 * l + 1) U
    ≤ 2 * A * (2 * l + 1) * vertex-count + (R + l * t) * edge-count
proof −
  have U-card: card U = vertex-count
    unfolding U-V vertex-count-def by simp
  have out-le: card (outgoing-edges U) ≤ edge-count
    by (rule edge-count-outgoing-bound)
  have out-term:
    (R + l * t) * card (outgoing-edges U) ≤ (R + l * t) * edge-count
    using out-le by simp
  show ?thesis
    unfolding level-range-cost-bound-def
    using U-card out-term by (simp add: algebra-simps)
qed
show ?thesis
  using correct range-bound graph-bound by linarith
```

qed

theorem *finite-initial-label-exact-split-range-costed-top-level-correct-and-scaled-seen-success-graph-bound-level-cap*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable* $s\ v$
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *insert-scaled-factor*: $t \leq A\text{-}insert * k$
and *seen-scaled-factor*: $k * \Delta + A\text{-}insert \leq 2 * A$
and *source-factor*: $Suc\ h \leq 2 * A$
and *k-pos*: $0 < k$
and *seen-success*:
$\bigwedge l'\ d\ S\ B\ D\ c\text{-}insert\ d'\ a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop$
child-costs U.
$D =$ *label-partition-view*
(*find-pivots-label-capped* k (*bmssp-level-cap* $k\ t\ l$) $d\ S\ B$)
(*find-pivots-pivots-capped* k (*bmssp-level-cap* $k\ t\ l$) $d\ S\ B$) $\Longrightarrow$
partition-initial-insert-cost-bound $c\text{-}insert\ t$
(*find-pivots-pivots-capped* k (*bmssp-level-cap* $k\ t\ l$) $d\ S\ B$) $\Longrightarrow$
exact-split-range-costed-partition-loop-state Δ (*bmssp-level-cap* $k\ t$) $t\ h\ k$
(*bmssp-level-cap* $k\ t\ l$) l'
(*find-pivots-label-capped* k (*bmssp-level-cap* $k\ t\ l$) $d\ S\ B$)
(*find-pivots-pivots-capped* k (*bmssp-level-cap* $k\ t\ l$) $d\ S\ B$) $B\ d'\ D$
$a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop\ child\text{-}costs \Longrightarrow$
complete-on d'
$\{v \in$ *bound-tree* $S\ B'$.
find-pivots-label-capped k (*bmssp-level-cap* $k\ t\ l$) $d\ S\ B\ v =$
dist $s\ v\} \Longrightarrow$
$U = U\text{-}loop \cup$
$\{v \in$ *bound-tree* $S\ B'$.
find-pivots-label-capped k (*bmssp-level-cap* $k\ t\ l$) $d\ S\ B\ v =$
dist $s\ v\} \Longrightarrow$
sound-label $d \Longrightarrow$
bmssp-pre-full $d\ S\ B \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *reachable* $s\ x) \Longrightarrow$
$(\bigwedge x.\ x \in S \Longrightarrow$ *below-bound* $(d\ x)\ B) \Longrightarrow$
$k * card\ S \leq$ *bmssp-level-cap* $k\ t\ l \Longrightarrow$
tree-antichain $S \Longrightarrow$
$B' = B \Longrightarrow$
card (*find-pivots-seen-capped* k (*bmssp-level-cap* $k\ t\ l$) $d\ S\ B$) $\leq$
card U
and *run*:
exact-split-range-costed-bmssp Δ (*bmssp-level-cap* $k\ t$) $t\ h\ k$
(*bmssp-level-cap* $k\ t\ l$) l
finite-initial-label $\{s\}$ *Infinity* d' *Infinity* $U\ c$
shows *sssp-correct* $d' \wedge$
$c \leq 2 * A * (2 * l + 1) *$ *vertex-count* $+ (R + l * t) *$ *edge-count*

proof (*rule finite-initial-label-exact-split-range-costed-top-level-correct-and-scaled-seen-success-graph-bound*
[*OF all-reachable degree degree-factor R-pos insert-factor insert-scaled-factor seen-scaled-factor source-factor k-pos - - - run*])
show $k \leq$ *bmssp-level-cap k t l*
proof −
have *one-le*: $1 \leq$ *bmssp-level-width t l*
unfolding *bmssp-level-width-def* **by** *simp*
have $k * 1 \leq k *$ *bmssp-level-width t l*
by (*rule mult-left-mono*[*OF one-le*]) *simp*
then show *?thesis*
unfolding *bmssp-level-cap-def* **by** *simp*
qed
next
fix *i*
assume $i \leq l$
then show *bmssp-level-cap k t i* $\leq$ *bmssp-level-cap k t l*
by (*rule bmssp-level-cap-mono*)
next
fix *l′ d S B D c-insert d′ a betas bs B′ Us U-loop c-loop child-costs U*
assume *D-def*:
D = *label-partition-view*
(*find-pivots-label-capped k* (*bmssp-level-cap k t l*) *d S B*)
(*find-pivots-pivots-capped k* (*bmssp-level-cap k t l*) *d S B*)
and *insert*:
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k* (*bmssp-level-cap k t l*) *d S B*)
and *loop*:
exact-split-range-costed-partition-loop-state Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) *l′*
(*find-pivots-label-capped k* (*bmssp-level-cap k t l*) *d S B*)
(*find-pivots-pivots-capped k* (*bmssp-level-cap k t l*) *d S B*) *B d′ D*
a betas bs B′ Us U-loop c-loop child-costs
and *complete*:
complete-on d′
{*v* ∈ *bound-tree S B′*.
find-pivots-label-capped k (*bmssp-level-cap k t l*) *d S B v* =
dist s v}
and *U-def*:
U = *U-loop* ∪
{*v* ∈ *bound-tree S B′*.
find-pivots-label-capped k (*bmssp-level-cap k t l*) *d S B v* =
dist s v}
and *sound-s*: *sound-label d*
and *pre-s*: *bmssp-pre-full d S B*
and *reaches-s*: ⋀*x*. *x* ∈ *S* ⟹ *reachable s x*
and *below-s*: ⋀*x*. *x* ∈ *S* ⟹ *below-bound* (*d x*) *B*
and *S-k-cap*: *k* ∗ *card S* ≤ *bmssp-level-cap k t l*
and *anti*: *tree-antichain S*

show $B' = B \Longrightarrow$
$card\ (find\text{-}pivots\text{-}seen\text{-}capped\ k\ (bmssp\text{-}level\text{-}cap\ k\ t\ l)\ d\ S\ B)$
$\leq card\ U$
by (*rule seen-success*
[*OF D-def insert loop complete U-def sound-s pre-s reaches-s below-s S-k-cap anti*])
qed

theorem *finite-initial-label-exact-split-range-costed-top-level-correct-and-closed-scaled-graph-bound-level-cap*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow reachable\ s\ v$
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *insert-scaled-factor*: $t \leq A\text{-}insert * k$
and *seen-scaled-factor*: $k * \Delta + A\text{-}insert \leq 2 * A$
and *source-factor*: $Suc\ h \leq 2 * A$
and *k-pos*: $0 < k$
and *run*:
exact-split-range-costed-bmssp Δ $(bmssp\text{-}level\text{-}cap\ k\ t)\ t\ h\ k$
$(bmssp\text{-}level\text{-}cap\ k\ t\ l)\ l$
finite-initial-label $\{s\}$ *Infinity* d' *Infinity* $U\ c$
shows *sssp-correct* $d' \wedge$
$c \leq 2 * A * (2 * l + 1) * vertex\text{-}count + (R + l * t) * edge\text{-}count$
proof (*rule finite-initial-label-exact-split-range-costed-top-level-correct-and-scaled-seen-success-graph-bound-level-cap*
[*OF all-reachable degree degree-factor R-pos insert-factor insert-scaled-factor seen-scaled-factor source-factor k-pos - run*])
fix $l'\ d\ S\ B\ D\ c\text{-}insert\ d'\ a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop\ child\text{-}costs\ U$
assume *loop*:
exact-split-range-costed-partition-loop-state Δ $(bmssp\text{-}level\text{-}cap\ k\ t)\ t\ h\ k$
$(bmssp\text{-}level\text{-}cap\ k\ t\ l)\ l'$
$(find\text{-}pivots\text{-}label\text{-}capped\ k\ (bmssp\text{-}level\text{-}cap\ k\ t\ l)\ d\ S\ B)$
$(find\text{-}pivots\text{-}pivots\text{-}capped\ k\ (bmssp\text{-}level\text{-}cap\ k\ t\ l)\ d\ S\ B)\ B\ d'\ D$
$a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop\ child\text{-}costs$
and *U-def*:
$U = U\text{-}loop \cup$
$\{v \in bound\text{-}tree\ S\ B'.$
$find\text{-}pivots\text{-}label\text{-}capped\ k\ (bmssp\text{-}level\text{-}cap\ k\ t\ l)\ d\ S\ B\ v =$
$dist\ s\ v\}$
and *sound-s*: *sound-label d*
and *pre-s*: *bmssp-pre-full d S B*
and *reaches-s*: $\bigwedge x.\ x \in S \Longrightarrow reachable\ s\ x$
and *below-s*: $\bigwedge x.\ x \in S \Longrightarrow below\text{-}bound\ (d\ x)\ B$
and *success*: $B' = B$
show *card* (*find-pivots-seen-capped k* (*bmssp-level-cap k t l*) *d S B*)
$\leq card\ U$
by (*rule exact-split-range-costed-step-seen-success*

```
    [OF loop U-def sound-s pre-s reaches-s below-s success])
qed

corollary finite-initial-label-exact-split-range-costed-top-level-correct-and-closed-
bmssp-graph-time-bound-level-cap:
  assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
    and degree: edge-outdegree-le Δ
    and degree-factor: Δ ≤ A
    and R-pos: 0 < R
    and insert-factor: t ≤ A * k
    and insert-scaled-factor: t ≤ A-insert * k
    and seen-scaled-factor: k * Δ + A-insert ≤ 2 * A
    and source-factor: Suc h ≤ 2 * A
    and k-pos: 0 < k
    and run:
      exact-split-range-costed-bmssp Δ (bmssp-level-cap k t) t h k
        (bmssp-level-cap k t l) l
        finite-initial-label {s} Infinity d' Infinity U c
  shows sssp-correct d' ∧
    c ≤ bmssp-graph-time-bound (λ-. A) (λ-. R) (λ-. l) (λ-. t)
      (λ-. edge-count) vertex-count
proof −
  have graph:
    sssp-correct d' ∧
      c ≤ 2 * A * (2 * l + 1) * vertex-count +
        (R + l * t) * edge-count
    by (rule finite-initial-label-exact-split-range-costed-top-level-correct-and-closed-
scaled-graph-bound-level-cap
      [OF all-reachable degree degree-factor R-pos insert-factor insert-scaled-factor
        seen-scaled-factor source-factor k-pos run])
  then show ?thesis
    unfolding bmssp-graph-time-bound-def by simp
qed

lemma exact-split-range-costed-partition-loop-state-trivial-edge-budget:
  assumes run:
    exact-split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
      betas bs B' Us U c child-costs
    and sound: sound-label d
    and pre: bmssp-pre-full d P B
    and P-reaches: ⋀x. x ∈ P ⟹ reachable s x
  shows t * sum-list (map card (range-tree-child-edge-range-list P a betas bs))
    ≤ t * card (outgoing-edges U)
proof −
  have split-run:
    split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
      betas bs B' Us U c child-costs
    using exact-split-range-costed-refines-split-range-costed(1)[OF run] .
  have mono: nondecreasing-from a bs
```

by (*rule split-range-costed-partition-loop-state-mono*[*OF split-run*])
have *trace*: *concrete-partition-loop-trace P B a bs d′ B′ Us U*
by (*rule split-range-costed-partition-loop-state-trace*
[*OF split-run sound pre P-reaches*])
have *children*:
list-all2 (λ*U X*. *U* = *X* ∧ *complete-on d′ U*) *Us*
(*range-tree-chain-list P a bs B′*)
using *trace* **unfolding** *concrete-partition-loop-trace-def* **by** *blast*
have *U-def*: *U* = *bound-tree P* (*Fin a*) ∪ ⋃(*set Us*)
using *trace* **unfolding** *concrete-partition-loop-trace-def* **by** *blast*
have *Union-eq*:
⋃(*set Us*) = ⋃(*set* (*range-tree-chain-list P a bs B′*))
using *children* **by** (*induction rule*: *list-all2-induct*) *auto*
have *U-eq-chain*:
U = *bound-tree P* (*Fin a*) ∪ *range-tree-chain P a bs B′*
using *U-def Union-eq Union-range-tree-chain-list*[*of P a bs B′*] **by** *simp*
have *chain-subset*: *range-tree-chain P a bs B′* ⊆ *U*
using *U-eq-chain* **by** *blast*
have *outgoing-subset*:
outgoing-edges (*range-tree-chain P a bs B′*) ⊆ *outgoing-edges U*
by (*rule outgoing-edges-mono*[*OF chain-subset*])
have *card-out-le*:
card (*outgoing-edges* (*range-tree-chain P a bs B′*)) ≤
card (*outgoing-edges U*)
by (*rule card-mono*) (*simp-all add*: *outgoing-subset*)
have *sum-le-chain*:
sum-list (*map card* (*range-tree-child-edge-range-list P a betas bs*))
≤ *card* (*outgoing-edges* (*range-tree-chain P a bs B′*))
by (*rule sum-card-range-tree-child-edge-range-list-le-outgoing-edges-chain*
[*OF mono*])
have *sum-le-U*:
sum-list (*map card* (*range-tree-child-edge-range-list P a betas bs*))
≤ *card* (*outgoing-edges U*)
using *sum-le-chain card-out-le* **by** *linarith*
then show *?thesis*
by *simp*
qed

theorem *exact-split-range-costed-partition-loop-state-closes-level-bound-from-child-bound-with-invariants-and-edge-budget*:
assumes *run*:
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
betas bs B′ Us U c child-costs
and *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d P B*
and *P-reaches*: ⋀*x*. *x* ∈ *P* ⟹ *reachable s x*
and *P-k-cap*: *k* ∗ *card P* ≤ *cap*
and *P-anti*: *tree-antichain P*
and *child-bound*:

```
    ⋀c-child U-child S-child B-child d-child B-child'.
      ⟦exact-split-range-costed-bmssp Δ M-of t h k cap l d S-child B-child
          d-child B-child' U-child c-child;
        bmssp-pre-full d S-child B-child;
        ⋀x. x ∈ S-child ⟹ reachable s x;
        card S-child ≤ M-of l;
        ⋀x. x ∈ S-child ⟹ below-bound (d x) B-child;
        k * card S-child ≤ cap;
        tree-antichain S-child⟧
      ⟹ c-child ≤ level-range-cost-bound A R L U-child
  and M-cap: M-of l ≤ cap
  and source-factor: Suc h ≤ A
  and k-pos: 0 < k
  and edge-budget:
    t * sum-list (map card (range-tree-child-edge-range-list P a betas bs)) ≤
      H * card (outgoing-edges U)
 shows c ≤ A * Suc L * card U + (R + H) * card (outgoing-edges U)
proof −
 have P-subset: P ⊆ V
   using pre unfolding bmssp-pre-full-def by blast
 have child-cost-bounds:
   list-all2 (λc-child U-child.
     c-child ≤ level-range-cost-bound A R L U-child)
     child-costs (range-tree-child-list P a bs)
     by (rule exact-split-range-costed-partition-loop-state-child-cost-bounds-with-
invariants
     [OF run P-subset P-k-cap P-anti child-bound])
 have loop:
   c ≤
     A * Suc L * card U + R * card (outgoing-edges U) +
     t * sum-list (map card (range-tree-child-edge-range-list P a betas bs))
  by (rule exact-split-range-costed-partition-loop-state-closes-level-bound-from-child-
costs-and-edge-ranges
     [OF run sound pre P-reaches child-cost-bounds M-cap source-factor k-pos])
 have c ≤ A * Suc L * card U + R * card (outgoing-edges U) +
     H * card (outgoing-edges U)
   using loop edge-budget by linarith
 also have ... = A * Suc L * card U + (R + H) * card (outgoing-edges U)
   by (simp add: algebra-simps)
 finally show ?thesis .
qed

theorem exact-split-range-costed-nonbase-step-closes-level-bound-from-child-bound-
with-invariants-and-edge-budget:
 assumes loop:
   exact-split-range-costed-partition-loop-state Δ M-of t h k cap l d P B d' D a
     betas bs B' Us U-loop c-loop child-costs
   and sound: sound-label d
   and pre: bmssp-pre-full d P B
```

and *P-reaches*: $\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*
and *P-k-cap*: $k * card\ P \leq cap$
and *P-anti*: *tree-antichain P*
and *child-bound*:
$\bigwedge$*c-child U-child S-child B-child d-child B-child'*.
⟦*exact-split-range-costed-bmssp* Δ *M-of t h k cap l d S-child B-child d-child B-child' U-child c-child*;
bmssp-pre-full d S-child B-child;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l*;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *below-bound* (*d x*) *B-child*;
$k * card$ *S-child* $\leq$ *cap*;
tree-antichain S-child⟧
$\Longrightarrow$ *c-child* $\leq$ *level-range-cost-bound A R L U-child*
and *M-cap*: *M-of l* $\leq$ *cap*
and *source-factor*: *Suc h* $\leq$ *A*
and *k-pos*: $0 < k$
and *edge-budget*:
$t * sum\text{-}list$ (*map card* (*range-tree-child-edge-range-list P a betas bs*)) $\leq$
$H * card$ (*outgoing-edges U-loop*)
and *U-def*: $U = U\text{-}loop \cup W$
and *finite-U*: *finite U*
and *scan-insert*: *c-scan-insert* $\leq A * card\ U$
and *c-def*: *c* = *c-scan-insert* + *c-loop*
shows $c \leq A * Suc\ (Suc\ L) * card\ U + (R + H) * card\ (outgoing\text{-}edges\ U)$
proof −
have *loop-bound*:
c-loop $\leq A * Suc\ L * card$ *U-loop* +
$(R + H) * card$ (*outgoing-edges U-loop*)
by (*rule exact-split-range-costed-partition-loop-state-closes-level-bound-from-child-bound-with-invariants-and-edge-budget*
[*OF loop sound pre P-reaches P-k-cap P-anti child-bound M-cap source-factor k-pos edge-budget*])
have *U-loop-subset*: *U-loop* $\subseteq U$
using *U-def* **by** *blast*
have *card-loop-le*: *card U-loop* $\leq$ *card U*
by (*rule card-mono*[*OF finite-U U-loop-subset*])
have *outgoing-subset*: *outgoing-edges U-loop* $\subseteq$ *outgoing-edges U*
by (*rule outgoing-edges-mono*[*OF U-loop-subset*])
have *finite-out-U*: *finite* (*outgoing-edges U*)
by *simp*
have *card-out-le*:
card (*outgoing-edges U-loop*) $\leq$ *card* (*outgoing-edges U*)
by (*rule card-mono*[*OF finite-out-U outgoing-subset*])
have $c \leq$
$A * card\ U +$
$(A * Suc\ L * card\ U + (R + H) * card\ (outgoing\text{-}edges\ U))$
proof −
have *loop-to-U*:

c-loop $\leq A * Suc\ L * card\ U +$
$(R + H) * card\ (outgoing\text{-}edges\ U)$
proof −
have $A * Suc\ L * card\ U\text{-}loop \leq A * Suc\ L * card\ U$
using *card-loop-le* **by** *simp*
moreover have $(R + H) * card\ (outgoing\text{-}edges\ U\text{-}loop) \leq$
$(R + H) * card\ (outgoing\text{-}edges\ U)$
using *card-out-le* **by** *simp*
ultimately show *?thesis*
using *loop-bound* **by** *linarith*
qed
show *?thesis*
using *scan-insert loop-to-U c-def* **by** *linarith*
qed
also have ... =
$A * Suc\ (Suc\ L) * card\ U + (R + H) * card\ (outgoing\text{-}edges\ U)$
by (*simp add*: *algebra-simps*)
finally show *?thesis* **.**
qed

theorem *exact-split-range-costed-bmssp-level-bound-from-local-budgets-with-invariants-and-edge-budget*:
assumes *base-budget*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *source-factor*: $Suc\ h \leq A$
and *k-pos*: $0 < k$
and *M-cap*: $\bigwedge i.\ i \leq l \Longrightarrow M\text{-}of\ i \leq cap$
and *edge-budget*:
$\bigwedge l\ d\ P\ B\ d'\ D\ a\ betas\ bs\ B'\ Us\ U\ c\ child\text{-}costs.$
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l d P B d' D a betas bs B' Us U c child-costs* $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d P B $\Longrightarrow$
$(\bigwedge x.\ x \in P \Longrightarrow reachable\ s\ x) \Longrightarrow$
$t * sum\text{-}list\ (map\ card\ (range\text{-}tree\text{-}child\text{-}edge\text{-}range\text{-}list\ P\ a\ betas\ bs))$
$\leq H * card\ (outgoing\text{-}edges\ U)$
and *step-budget*:
$\bigwedge l\ d\ S\ B\ D\ c\text{-}insert\ d'\ a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop$
child-costs U.
$D = label\text{-}partition\text{-}view$
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k cap d S B*) $\Longrightarrow$
exact-split-range-costed-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d' D a betas bs B'*
Us U-loop c-loop child-costs $\Longrightarrow$
complete-on d'

```
      {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v} ⟹
    U = U-loop ∪
      {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v = dist s v} ⟹
    sound-label d ⟹
    bmssp-pre-full d S B ⟹
    (⋀x. x ∈ S ⟹ reachable s x) ⟹
    (⋀x. x ∈ S ⟹ below-bound (d x) B) ⟹
    k * card S ≤ cap ⟹
    tree-antichain S ⟹
    fp-iter-capped-scan-cost k cap d S S B + c-insert ≤ A * card U
  and run:
    exact-split-range-costed-bmssp Δ M-of t h k cap l d S B d' B' U c
  and sound: sound-label d
  and pre: bmssp-pre-full d S B
  and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
  and below: ⋀x. x ∈ S ⟹ below-bound (d x) B
  and S-k-cap: k * card S ≤ cap
  and S-anti: tree-antichain S
 shows c ≤ level-range-cost-bound A (R + l * H) (2 * l + 1) U
 using run sound pre S-reaches below S-k-cap S-anti M-cap
proof (induction l arbitrary: d S B d' B' U c rule: less-induct)
 case (less l)
 show ?case
 proof (cases l)
  case 0
  have run0:
    exact-split-range-costed-bmssp Δ M-of t h k cap 0 d S B d' B' U c
    using less.prems(1) 0 by simp
  have split-run0:
    split-range-costed-bmssp Δ M-of t h k cap 0 d S B d' B' U c
    using exact-split-range-costed-refines-split-range-costed(2)[OF run0] .
  show ?thesis
    using split-run0 R-pos 0
    by (cases rule: split-range-costed-bmssp.cases)
      (simp-all add: split-range-costed-base-level-bound)
 next
  case (Suc l0)
  have run-suc:
    exact-split-range-costed-bmssp Δ M-of t h k cap (Suc l0) d S B d' B' U c
    using less.prems(1) Suc by simp
  show ?thesis
  proof (cases rule: exact-split-range-costed-bmssp-SucE[OF run-suc])
   case (1 c-insert a betas bs Us U-loop c-loop child-costs-loop)
   let ?d-fp = find-pivots-label-capped k cap d S B
   let ?P = find-pivots-pivots-capped k cap d S B
   let ?W = {v ∈ bound-tree S B'. ?d-fp v = dist s v}
   have sound-fp: sound-label ?d-fp
     unfolding find-pivots-label-capped-def
     by (rule fp-iter-capped-label-sound[OF less.prems(2) less.prems(4)])
```

have *pivot-pre*: *bmssp-pre-full ?d-fp ?P B*
using *find-pivots-capped-establishes-pivot-pre-concrete*
[*OF less.prems*(*2*) *less.prems*(*3*) *less.prems*(*4*)] **.**
have *S-subset*: $S \subseteq V$
using *less.prems*(*3*) **unfolding** *bmssp-pre-full-def* **by** *blast*
have *P-subset-S*: $?P \subseteq S$
unfolding *find-pivots-pivots-capped-def* **by** (*auto split*: *if-splits*)
have *P-reaches*: $\bigwedge x.\ x \in ?P \Longrightarrow$ *reachable s x*
using *P-subset-S less.prems*(*4*) **by** *blast*
have *P-k-cap*: $k * card\ ?P \leq cap$
by (*rule find-pivots-pivots-capped-scaled-card-le*
[*OF S-subset less.prems*(*6*)])
have *P-anti*: *tree-antichain ?P*
by (*rule find-pivots-pivots-capped-tree-antichain*[*OF less.prems*(*7*)])
have *child-bound*:
$\bigwedge$*c-child U-child S-child B-child d-child B-child'*.
⟦*exact-split-range-costed-bmssp* Δ *M-of t h k cap l0 ?d-fp S-child B-child*
d-child B-child' U-child c-child;
bmssp-pre-full ?d-fp S-child B-child;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*;
card S-child $\leq$ *M-of l0*;
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *below-bound* (*?d-fp x*) *B-child*;
$k * card$ *S-child* $\leq$ *cap*;
tree-antichain S-child⟧
$\Longrightarrow$ *c-child* $\leq$ *level-range-cost-bound A* $(R + l0 * H)$
(*2* $*$ *l0* $+$ *1*) *U-child*
proof $-$
fix *c-child U-child S-child B-child d-child B-child'*
assume *child*:
exact-split-range-costed-bmssp Δ *M-of t h k cap l0 ?d-fp S-child B-child*
d-child B-child' U-child c-child
and *child-pre*: *bmssp-pre-full ?d-fp S-child B-child*
and *child-reaches*: $\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *reachable s x*
and *child-below*:
$\bigwedge x.\ x \in$ *S-child* $\Longrightarrow$ *below-bound* (*?d-fp x*) *B-child*
and *child-k-cap*: $k * card$ *S-child* $\leq$ *cap*
and *child-anti*: *tree-antichain S-child*
have *M-cap-child*: $\bigwedge i.\ i \leq l0 \Longrightarrow$ *M-of* $i \leq$ *cap*
using *less.prems*(*8*) *Suc* **by** *simp*
show *c-child* $\leq$ *level-range-cost-bound A* $(R + l0 * H)$
(*2* $*$ *l0* $+$ *1*) *U-child*
by (*rule less.IH*[*of l0 ?d-fp S-child B-child d-child B-child'*
U-child c-child, *OF - child sound-fp child-pre child-reaches*
child-below child-k-cap child-anti M-cap-child])
(*simp add*: *Suc*)
qed
have *loop-edge-budget*:
$t * sum\text{-}list\ (map\ card\ (range\text{-}tree\text{-}child\text{-}edge\text{-}range\text{-}list\ ?P\ a\ betas\ bs))$
$\leq H * card\ (outgoing\text{-}edges\ U\text{-}loop)$

```
      by (rule edge-budget[OF 1(4) sound-fp pivot-pre P-reaches])
    have scan-insert:
      fp-iter-capped-scan-cost k cap d S S B + c-insert ≤
        A * card U
      by (rule step-budget[OF refl 1(3) 1(4) 1(5) 1(1)
          less.prems(2) less.prems(3) less.prems(4) less.prems(5)
          less.prems(6) less.prems(7)])
    have trace:
      concrete-partition-loop-trace ?P B a bs d' B' Us U-loop
    proof −
      have split-loop:
        split-range-costed-partition-loop-state Δ M-of t h k cap l0
          ?d-fp ?P B d'
          (label-partition-view ?d-fp ?P) a betas bs B' Us U-loop c-loop
          child-costs-loop
        using exact-split-range-costed-refines-split-range-costed(1)[OF 1(4)]
          by simp
      show ?thesis
        by (rule split-range-costed-partition-loop-state-trace
          [OF split-loop sound-fp pivot-pre P-reaches])
    qed
    have loop-post: bmssp-post-full ?d-fp ?P B d' B' U-loop
      by (rule concrete-partition-loop-trace-post[OF trace])
    have finite-U-loop: finite U-loop
      using loop-post unfolding bmssp-post-full-def by simp
    have finite-W: finite ?W
      by simp
    have finite-U: finite U
      using 1(1) finite-U-loop finite-W by simp
    have M-cap-l0: M-of l0 ≤ cap
      using less.prems(8) Suc by simp
    have step:
      c ≤ A * Suc (Suc (2 * l0 + 1)) * card U +
        (R + l0 * H + H) * card (outgoing-edges U)
      by (rule exact-split-range-costed-nonbase-step-closes-level-bound-from-child-
bound-with-invariants-and-edge-budget
        [OF 1(4) sound-fp pivot-pre P-reaches P-k-cap P-anti child-bound
          M-cap-l0 source-factor k-pos loop-edge-budget 1(1) finite-U
          scan-insert 1(2)])
    show ?thesis
      using step Suc unfolding level-range-cost-bound-def
      by (simp add: algebra-simps)
  qed
 qed
qed

theorem finite-initial-label-exact-split-range-costed-top-level-correct-and-closed-
refined-graph-bound-level-cap:
 assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
```

```
    and degree: edge-outdegree-le Δ
    and degree-factor: Δ ≤ A
    and R-pos: 0 < R
    and insert-factor: t ≤ A * k
    and insert-scaled-factor: t ≤ A-insert * k
    and seen-scaled-factor: k * Δ + A-insert ≤ 2 * A
    and source-factor: Suc h ≤ 2 * A
    and k-pos: 0 < k
    and edge-budget:
      ⋀l' d P B d' D a betas bs B' Us U-loop c-loop child-costs.
        exact-split-range-costed-partition-loop-state Δ (bmssp-level-cap k t) t h k
          (bmssp-level-cap k t l) l' d P B d' D a betas bs B'
          Us U-loop c-loop child-costs ⟹
        sound-label d ⟹
        bmssp-pre-full d P B ⟹
        (⋀x. x ∈ P ⟹ reachable s x) ⟹
        t * sum-list (map card (range-tree-child-edge-range-list P a betas bs))
          ≤ H * card (outgoing-edges U-loop)
    and run:
      exact-split-range-costed-bmssp Δ (bmssp-level-cap k t) t h k
        (bmssp-level-cap k t l) l
        finite-initial-label {s} Infinity d' Infinity U c
  shows sssp-correct d' ∧
    c ≤ 2 * A * (2 * l + 1) * vertex-count + (R + l * H) * edge-count
proof
  show sssp-correct d'
    by (rule finite-initial-label-exact-split-range-costed-top-level-correct
      [OF all-reachable run])
next
  have pre: bmssp-pre-full finite-initial-label {s} Infinity
    using all-reachable finite-initial-label-source-complete
    by (rule top-bmssp-pre-full)
  have sound: sound-label finite-initial-label
    using finite-initial-label-sound[OF all-reachable] .
  have S-reaches: ⋀x. x ∈ {s} ⟹ reachable s x
    using all-reachable source-in-V by blast
  have below: ⋀x. x ∈ {s} ⟹
      below-bound (finite-initial-label x) Infinity
    by simp
  have top-cap: k * card {s} ≤ bmssp-level-cap k t l
  proof −
    have one-le: 1 ≤ bmssp-level-width t l
      unfolding bmssp-level-width-def by simp
    have k * 1 ≤ k * bmssp-level-width t l
      by (rule mult-left-mono[OF one-le]) simp
    then show ?thesis
      unfolding bmssp-level-cap-def by simp
  qed
  have top-anti: tree-antichain {s}
```

by *simp*
have *range-bound*:
c ≤ *level-range-cost-bound* (*2* ∗ *A*) (*R* + *l* ∗ *H*) (*2* ∗ *l* + *1*) *U*
proof (*rule exact-split-range-costed-bmssp-level-bound-from-local-budgets-with-invariants-and-edge-budget*
[**where** *A* = *2* ∗ *A* **and** *R* = *R* **and** *H* = *H*,
OF - R-pos source-factor k-pos - - - run sound pre S-reaches below top-cap top-anti])
show Δ ≤ *2* ∗ *A*
using *degree-factor* **by** *simp*
next
fix *i*
assume *i* ≤ *l*
then show *bmssp-level-cap k t i* ≤ *bmssp-level-cap k t l*
by (*rule bmssp-level-cap-mono*)
next
fix *l′ d P B d′ D a betas bs B′ Us U-loop c-loop child-costs*
assume *loop*:
exact-split-range-costed-partition-loop-state Δ
(*bmssp-level-cap k t*) *t h k* (*bmssp-level-cap k t l*) *l′*
d P B d′ D a betas bs B′ Us U-loop c-loop child-costs
and *sound-l*: *sound-label d*
and *pre-l*: *bmssp-pre-full d P B*
and *reaches-l*: ⋀*x*. *x* ∈ *P* ⟹ *reachable s x*
show *t* ∗ *sum-list* (*map card* (*range-tree-child-edge-range-list P a betas bs*))
≤ *H* ∗ *card* (*outgoing-edges U-loop*)
by (*rule edge-budget*[*OF loop sound-l pre-l reaches-l*])
next
fix *l′ d S B D c-insert d′ a betas bs B′ Us U-loop c-loop child-costs U*
assume *D-def*:
D = *label-partition-view*
(*find-pivots-label-capped k* (*bmssp-level-cap k t l*) *d S B*)
(*find-pivots-pivots-capped k* (*bmssp-level-cap k t l*) *d S B*)
and *insert*:
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k* (*bmssp-level-cap k t l*) *d S B*)
and *loop*:
exact-split-range-costed-partition-loop-state Δ
(*bmssp-level-cap k t*) *t h k* (*bmssp-level-cap k t l*) *l′*
(*find-pivots-label-capped k* (*bmssp-level-cap k t l*) *d S B*)
(*find-pivots-pivots-capped k* (*bmssp-level-cap k t l*) *d S B*) *B d′ D*
a betas bs B′ Us U-loop c-loop child-costs
and *complete*:
complete-on d′
{*v* ∈ *bound-tree S B′*.
find-pivots-label-capped k (*bmssp-level-cap k t l*) *d S B v* =
dist s v}
and *U-def*:
U = *U-loop* ∪

```
      {v ∈ bound-tree S B'.
        find-pivots-label-capped k (bmssp-level-cap k t l) d S B v =
          dist s v}
    and sound-s: sound-label d
    and pre-s: bmssp-pre-full d S B
    and reaches-s: ⋀x. x ∈ S ⟹ reachable s x
    and below-s: ⋀x. x ∈ S ⟹ below-bound (d x) B
    and S-k-cap: k * card S ≤ bmssp-level-cap k t l
    and anti: tree-antichain S
  have split-loop:
    split-range-costed-partition-loop-state Δ (bmssp-level-cap k t) t h k
      (bmssp-level-cap k t l) l'
      (find-pivots-label-capped k (bmssp-level-cap k t l) d S B)
      (find-pivots-pivots-capped k (bmssp-level-cap k t l) d S B) B d' D
      a betas bs B' Us U-loop c-loop child-costs
    by (rule exact-split-range-costed-refines-split-range-costed(1)[OF loop])
  have seen-success:
    B' = B ⟹
      card (find-pivots-seen-capped k (bmssp-level-cap k t l) d S B)
      ≤ card U
    by (rule exact-split-range-costed-step-seen-success
      [OF loop U-def sound-s pre-s reaches-s below-s])
  show fp-iter-capped-scan-cost k (bmssp-level-cap k t l) d S S B +
      c-insert ≤ 2 * A * card U
    by (rule split-range-costed-capped-step-scan-insert-budget-from-scaled-seen-or-
threshold
      [OF degree degree-factor insert-factor insert-scaled-factor
        seen-scaled-factor insert split-loop U-def sound-s pre-s reaches-s
        S-k-cap anti k-pos seen-success])
qed
have post-full:
  bmssp-post-full finite-initial-label {s} Infinity d' Infinity U
  by (rule exact-split-range-costed-bmssp-correct[OF run sound pre S-reaches])
have post: bmssp-post finite-initial-label {s} Infinity d' Infinity U
  using bmssp-post-full-imp-post[OF post-full] .
have U-V: U = V
  using post bound-tree-source-infinity[OF all-reachable]
  unfolding bmssp-post-def by auto
have graph-bound:
  level-range-cost-bound (2 * A) (R + l * H) (2 * l + 1) U
    ≤ 2 * A * (2 * l + 1) * vertex-count + (R + l * H) * edge-count
proof −
  have U-card: card U = vertex-count
    unfolding U-V vertex-count-def by simp
  have out-le: card (outgoing-edges U) ≤ edge-count
    by (rule edge-count-outgoing-bound)
  have out-term:
    (R + l * H) * card (outgoing-edges U) ≤ (R + l * H) * edge-count
    using out-le by simp
```

show *?thesis*
unfolding *level-range-cost-bound-def*
using *U-card out-term* **by** (*simp add*: *algebra-simps*)
qed
show $c \leq 2 * A * (2 * l + 1) * \textit{vertex-count} + (R + l * H) * \textit{edge-count}$
using *range-bound graph-bound* **by** *linarith*
qed

corollary *finite-initial-label-exact-split-range-costed-top-level-correct-and-closed-bmssp-refined-graph-time-bound-level-cap*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow \textit{reachable}\ s\ v$
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *insert-scaled-factor*: $t \leq A\text{-}\textit{insert} * k$
and *seen-scaled-factor*: $k * \Delta + A\text{-}\textit{insert} \leq 2 * A$
and *source-factor*: $\textit{Suc}\ h \leq 2 * A$
and *k-pos*: $0 < k$
and *edge-budget*:
$\bigwedge l'\ d\ P\ B\ d'\ D\ a\ \textit{betas}\ \textit{bs}\ B'\ \textit{Us}\ \textit{U-loop}\ \textit{c-loop}\ \textit{child-costs}.$
exact-split-range-costed-partition-loop-state Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) l' *d P B* d' *D a betas bs* B'
Us U-loop c-loop child-costs $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d P B $\Longrightarrow$
$(\bigwedge x.\ x \in P \Longrightarrow \textit{reachable}\ s\ x) \Longrightarrow$
$t * \textit{sum-list}\ (\textit{map}\ \textit{card}\ (\textit{range-tree-child-edge-range-list}\ P\ a\ \textit{betas}\ \textit{bs}))$
$\leq H * \textit{card}\ (\textit{outgoing-edges}\ \textit{U-loop})$
and *run*:
exact-split-range-costed-bmssp Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) *l*
finite-initial-label {*s*} *Infinity* d' *Infinity U c*
shows *sssp-correct* $d' \wedge$
$c \leq$ *bmssp-refined-graph-time-bound* (λ-. *A*) (λ-. *R*) (λ-. *H*)
(λ-. *l*) (λ-. *t*) (λ-. *edge-count*) *vertex-count*
proof −
have *graph*:
sssp-correct $d' \wedge$
$c \leq 2 * A * (2 * l + 1) * \textit{vertex-count} + (R + l * H) * \textit{edge-count}$
by (*rule finite-initial-label-exact-split-range-costed-top-level-correct-and-closed-refined-graph-bound-level-cap*
[*OF all-reachable degree degree-factor R-pos insert-factor insert-scaled-factor seen-scaled-factor source-factor k-pos edge-budget run*])
have *bound*:
$2 * A * (2 * l + 1) * \textit{vertex-count} + (R + l * H) * \textit{edge-count}$
$\leq$ *bmssp-refined-graph-time-bound* (λ-. *A*) (λ-. *R*) (λ-. *H*)
(λ-. *l*) (λ-. *t*) (λ-. *edge-count*) *vertex-count*
unfolding *bmssp-refined-graph-time-bound-def* **by** *simp*

show *?thesis*
using *graph bound* **by** *linarith*
qed

corollary *finite-initial-label-exact-split-range-costed-top-level-correct-and-closed-bmssp-refined-graph-time-bound-level-cap-trivial-edge-budget*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *insert-scaled-factor*: $t \leq A\text{-}insert * k$
and *seen-scaled-factor*: $k * \Delta + A\text{-}insert \leq 2 * A$
and *source-factor*: $Suc\ h \leq 2 * A$
and *k-pos*: $0 < k$
and *run*:
exact-split-range-costed-bmssp Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) *l*
finite-initial-label $\{s\}$ *Infinity d' Infinity U c*
shows *sssp-correct d'* $\wedge$
$c \leq$ *bmssp-refined-graph-time-bound* (λ-. *A*) (λ-. *R*) (λ-. *t*)
(λ-. *l*) (λ-. *t*) (λ-. *edge-count*) *vertex-count*
proof (*rule finite-initial-label-exact-split-range-costed-top-level-correct-and-closed-bmssp-refined-graph-time-bound-level-cap*
[*OF all-reachable degree degree-factor R-pos insert-factor insert-scaled-factor seen-scaled-factor source-factor k-pos - run*])
fix *l' d P B d' D a betas bs B' Us U-loop c-loop child-costs*
assume *loop*:
exact-split-range-costed-partition-loop-state Δ
(*bmssp-level-cap k t*) *t h k* (*bmssp-level-cap k t l*) *l'*
d P B d' D a betas bs B' Us U-loop c-loop child-costs
and *sound-l*: *sound-label d*
and *pre-l*: *bmssp-pre-full d P B*
and *reaches-l*: $\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*
show *t* $*$ *sum-list* (*map card* (*range-tree-child-edge-range-list P a betas bs*))
$\leq$ *t* $*$ *card* (*outgoing-edges U-loop*)
by (*rule exact-split-range-costed-partition-loop-state-trivial-edge-budget*
[*OF loop sound-l pre-l reaches-l*])
qed

end

end
theory *BMSSP-Partition-Data-Structure*
imports *BMSSP-Partition-Pull-Bridge*
begin

34 Concrete Partition Data Structure

The abstract partition interface uses a set of active keys together with a value function. A concrete implementation must be careful around *Pull*: the abstract contract removes keys but leaves the value function unchanged. The state below therefore stores a sorted active list and a value memory. Removing a key only removes it from the active list; the remembered value is retained for the abstraction relation.

```
record 'k partition-state =
  entries-of :: ('k × real) list
  values-of :: 'k ⇒ real

definition entry-keys :: ('k × real) list ⇒ 'k set where
  entry-keys xs = fst ‘ set xs

definition partition-state-invar :: 'k partition-state ⇒ bool where
  partition-state-invar P ⟷
    distinct (map fst (entries-of P)) ∧
    sorted-wrt (λp q. snd p ≤ snd q) (entries-of P) ∧
    (∀(x, b)∈set (entries-of P). values-of P x = b)

definition partition-state-view :: 'k partition-state ⇒ 'k partition-view where
  partition-state-view P =
    (| keys-of = entry-keys (entries-of P), value-of = values-of P |)

definition partition-state-refines ::
  'k partition-state ⇒ 'k partition-view ⇒ bool where
  partition-state-refines P D ⟷
    partition-state-invar P ∧ partition-state-view P = D

definition empty-partition-state :: ('k ⇒ real) ⇒ 'k partition-state where
  empty-partition-state v = (| entries-of = [], values-of = v |)

definition partition-state-from-keys ::
  ('k ⇒ real) ⇒ 'k list ⇒ 'k partition-state where
  partition-state-from-keys d xs =
    (| entries-of = sort-key snd (map (λx. (x, d x)) xs),
      values-of = d |)

definition partition-state-insert ::
  'k ⇒ real ⇒ 'k partition-state ⇒ 'k partition-state where
  partition-state-insert x b P =
    (let b' = (if x ∈ entry-keys (entries-of P)
        then min (values-of P x) b else b);
      rest = filter (λp. fst p ≠ x) (entries-of P)
     in (| entries-of = sort-key snd ((x, b') # rest),
        values-of = (values-of P)(x := b') |))
```

definition *partition-state-batch-prepend* ::
  ('k × real) list ⇒ 'k partition-state ⇒ 'k partition-state **where**
  partition-state-batch-prepend xs P =
    fold (λ(*x*, *b*) *P'*. *partition-state-insert x b P'*) *xs P*

definition *partition-state-pull-prefix* ::
  nat ⇒ 'k partition-state ⇒ ('k × real) list **where**
  partition-state-pull-prefix M P =
    (*let xs* = *entries-of P in*
     if length xs ≤ *M then xs*
     else takeWhile (λ*p*. *snd p* < *snd* (*xs* ! *M*)) *xs*)

definition *partition-state-pull-set* ::
  nat ⇒ 'k partition-state ⇒ 'k set **where**
  partition-state-pull-set M P =
    entry-keys (*partition-state-pull-prefix M P*)

definition *partition-state-pull-bound* ::
  nat ⇒ *real* ⇒ 'k partition-state ⇒ *real* **where**
  partition-state-pull-bound M B P =
    (*let xs* = *entries-of P in if length xs* ≤ *M then B else snd* (*xs* ! *M*))

definition *partition-state-delete-keys* ::
  'k set ⇒ 'k partition-state ⇒ 'k partition-state **where**
  partition-state-delete-keys S P =
    (| *entries-of* = *filter* (λ*p*. *fst p* ∉ *S*) (*entries-of P*),
      values-of = *values-of P* |)

definition *partition-state-pull* ::
  nat ⇒ *real* ⇒ 'k partition-state ⇒
   'k set × real × 'k partition-state **where**
  partition-state-pull M B P =
    (*let S* = *partition-state-pull-set M P*;
       beta = *partition-state-pull-bound M B P*
     in (*S*, *beta*, *partition-state-delete-keys S P*))

definition *costed-partition-state-insert* ::
  nat ⇒ 'k partition-state ⇒ 'k ⇒ *real* ⇒ *nat* ⇒
   'k partition-state ⇒ *bool* **where**
  costed-partition-state-insert t P x b c P' ⟷
    P' = *partition-state-insert x b P* ∧ *c* = *t*

definition *costed-partition-state-batch-prepend* ::
  nat ⇒ 'k partition-state ⇒ ('k × real) list ⇒ *nat* ⇒
   'k partition-state ⇒ *bool* **where**
  costed-partition-state-batch-prepend t P xs c P' ⟷
    P' = *partition-state-batch-prepend xs P* ∧ *c* = *t* ∗ *length xs*

definition *costed-partition-state-pull* ::

nat ⇒ *real* ⇒ *'k partition-state* ⇒ *'k set* ⇒ *real* ⇒
'k partition-state ⇒ *nat* ⇒ *bool* **where**
costed-partition-state-pull M B P S beta P' c ⟷
partition-state-pull M B P = (*S*, *beta*, *P'*) ∧ *c* = *card S*

lemma *costed-partition-state-insert-exists*:
∃ *c P'*. *costed-partition-state-insert t P x b c P'*
unfolding *costed-partition-state-insert-def* **by** *blast*

lemma *costed-partition-state-insert-deterministic*:
assumes *costed-partition-state-insert t P x b c P'*
and *costed-partition-state-insert t P x b c' P''*
shows *c* = *c'* ∧ *P'* = *P''*
using *assms* **unfolding** *costed-partition-state-insert-def* **by** *blast*

lemma *costed-partition-state-batch-prepend-exists*:
∃ *c P'*. *costed-partition-state-batch-prepend t P xs c P'*
unfolding *costed-partition-state-batch-prepend-def* **by** *blast*

lemma *costed-partition-state-batch-prepend-deterministic*:
assumes *costed-partition-state-batch-prepend t P xs c P'*
and *costed-partition-state-batch-prepend t P xs c' P''*
shows *c* = *c'* ∧ *P'* = *P''*
using *assms* **unfolding** *costed-partition-state-batch-prepend-def* **by** *blast*

lemma *costed-partition-state-pull-exists*:
∃ *S beta P' c*. *costed-partition-state-pull M B P S beta P' c*
unfolding *costed-partition-state-pull-def*
by (*metis prod-cases3*)

lemma *costed-partition-state-pull-deterministic*:
assumes *costed-partition-state-pull M B P S beta P' c*
and *costed-partition-state-pull M B P S' beta' P'' c'*
shows *S* = *S'* ∧ *beta* = *beta'* ∧ *P'* = *P''* ∧ *c* = *c'*
using *assms* **unfolding** *costed-partition-state-pull-def* **by** *auto*

lemma *entry-keys-simps* [*simp*]:
entry-keys [] = {}
entry-keys (*x* # *xs*) = *insert* (*fst x*) (*entry-keys xs*)
unfolding *entry-keys-def* **by** *auto*

lemma *entry-keys-filter-notin*:
entry-keys (*filter* (λ*p*. *fst p* ∉ *S*) *xs*) = *entry-keys xs* − *S*
unfolding *entry-keys-def* **by** *auto*

lemma *entry-keys-filter-neq*:
entry-keys (*filter* (λ*p*. *fst p* ≠ *x*) *xs*) = *entry-keys xs* − {*x*}
unfolding *entry-keys-def* **by** *auto*

lemma *entry-keys-sort-key* [*simp*]:
entry-keys (*sort-key f xs*) = *entry-keys xs*
unfolding *entry-keys-def* **by** *simp*

lemma *entry-keys-insort-key* [*simp*]:
entry-keys (*insort-key f x xs*) = *insert* (*fst x*) (*entry-keys xs*)
unfolding *entry-keys-def* **by** (*simp add*: *set-insort-key*)

lemma *sorted-wrt-snd-sort-key* [*simp*]:
sorted-wrt (λ*p q*. *snd p* ≤ *snd q*) (*sort-key snd xs*)
proof −
have *sorted* (*map snd* (*sort-key snd xs*))
by (*rule sorted-sort-key*)
then show *?thesis*
by (*simp add*: *sorted-map*)
qed

lemma *distinct-map-fst-sort-key*:
assumes *distinct* (*map fst xs*)
shows *distinct* (*map fst* (*sort-key f xs*))
proof −
have *distinct-sort*: *distinct* (*sort-key f xs*)
using *assms* **unfolding** *distinct-map* **by** *simp*
have *inj*: *inj-on fst* (*set* (*sort-key f xs*))
using *assms* **unfolding** *distinct-map* **by** *simp*
show *?thesis*
unfolding *distinct-map* **using** *distinct-sort inj* **by** *simp*
qed

lemma *empty-partition-state-invar* [*simp*]:
partition-state-invar (*empty-partition-state v*)
unfolding *empty-partition-state-def partition-state-invar-def* **by** *simp*

lemma *empty-partition-state-view* [*simp*]:
partition-state-view (*empty-partition-state v*) =
(| *keys-of* = {}, *value-of* = *v* |)
unfolding *empty-partition-state-def partition-state-view-def* **by** *simp*

lemma *partition-state-from-keys-invar*:
assumes *distinct xs*
shows *partition-state-invar* (*partition-state-from-keys d xs*)
proof −
let *?ps* = *map* (λ*x*. (*x*, *d x*)) *xs*
have *base-distinct*: *distinct* (*map fst ?ps*)
using *assms* **by** (*simp add*: *o-def*)
have *distinct-ps*: *distinct* (*map fst* (*sort-key snd ?ps*))
by (*rule distinct-map-fst-sort-key*[*OF base-distinct*])
have *values-consistent*:
⋀*x b*. (*x*, *b*) ∈ *set* (*sort-key snd ?ps*) ⟹ *d x* = *b*

```
    by auto
  have sorted-ps:
    sorted-wrt (λp q. snd p ≤ snd q) (sort-key snd ?ps)
    by simp
  have consistent-ps:
    ∀(x, b)∈set (sort-key snd ?ps). d x = b
    using values-consistent by blast
  show ?thesis
    unfolding partition-state-from-keys-def partition-state-invar-def
    using distinct-ps sorted-ps consistent-ps by simp
qed

lemma partition-state-from-keys-view:
  partition-state-view (partition-state-from-keys d xs) =
    (| keys-of = set xs, value-of = d |)
  unfolding partition-state-from-keys-def partition-state-view-def entry-keys-def
  by (simp add: image-image)

lemma partition-state-from-keys-refines:
  assumes distinct xs
  shows partition-state-refines (partition-state-from-keys d xs)
    (| keys-of = set xs, value-of = d |)
  using partition-state-from-keys-invar[OF assms] partition-state-from-keys-view
  unfolding partition-state-refines-def by blast

lemma partition-state-insert-invar:
  assumes inv: partition-state-invar P
  shows partition-state-invar (partition-state-insert x b P)
proof −
  let ?b = if x ∈ entry-keys (entries-of P) then min (values-of P x) b else b
  let ?rest = filter (λp. fst p ≠ x) (entries-of P)
  have distinct-rest: distinct (map fst ?rest)
    using inv unfolding partition-state-invar-def by (simp add: distinct-map-filter)
  have x-not-rest: x ∉ fst ‘ set ?rest
    by auto
  have distinct-new: distinct (map fst ((x, ?b) # ?rest))
    using distinct-rest x-not-rest by simp
  have values-consistent:
    ⋀y c. (y, c) ∈ set (sort-key snd ((x, ?b) # ?rest)) ⟹
      ((values-of P)(x := ?b)) y = c
  proof −
    fix y c
    assume yc: (y, c) ∈ set (sort-key snd ((x, ?b) # ?rest))
    then have yc-set: (y, c) ∈ set ((x, ?b) # ?rest)
      by (metis set-sort)
    show ((values-of P)(x := ?b)) y = c
    proof (cases y = x)
      case True
      then have c = ?b
```

```
      using yc-set x-not-rest by auto
    then show ?thesis
      using True by simp
  next
    case False
    then have (y, c) ∈ set (entries-of P)
      using yc-set by auto
    then have values-of P y = c
      using inv unfolding partition-state-invar-def by auto
    then show ?thesis
      using False by simp
  qed
 qed
 have distinct-sorted:
  distinct (map fst (sort-key snd ((x, ?b) # ?rest)))
  by (rule distinct-map-fst-sort-key[OF distinct-new])
 have sorted-new:
  sorted-wrt (λp q. snd p ≤ snd q) (sort-key snd ((x, ?b) # ?rest))
 proof −
  have sorted (map snd (sort-key snd ((x, ?b) # ?rest)))
    by (rule sorted-sort-key)
  then show ?thesis
    by (simp add: sorted-map)
 qed
 have consistent-new:
  ∀(y, c)∈set (sort-key snd ((x, ?b) # ?rest)).
    ((values-of P)(x := ?b)) y = c
  using values-consistent by blast
 show ?thesis
  unfolding partition-state-insert-def partition-state-invar-def Let-def
  using distinct-sorted sorted-new consistent-new by simp
qed

theorem partition-state-insert-refines-min-update:
 assumes partition-state-invar P
 shows partition-state-view (partition-state-insert x b P) =
  min-update (partition-state-view P) x b
 unfolding partition-state-insert-def partition-state-view-def min-update-def Let-def
 by (simp add: entry-keys-filter-neq)

theorem partition-state-insert-refines-insert-spec:
 assumes inv: partition-state-invar P
 shows insert-spec (partition-state-view P) x b
  (partition-state-view (partition-state-insert x b P))
 unfolding partition-state-insert-refines-min-update[OF inv]
 by (rule min-update-insert-spec)

theorem partition-state-insert-refines-view:
 assumes ref: partition-state-refines P D
```

```
  shows partition-state-refines (partition-state-insert x b P)
    (min-update D x b)
proof −
  have inv: partition-state-invar P
    and view: partition-state-view P = D
    using ref unfolding partition-state-refines-def by blast+
  have inv': partition-state-invar (partition-state-insert x b P)
    by (rule partition-state-insert-invar[OF inv])
  have view': partition-state-view (partition-state-insert x b P) =
      min-update D x b
    using partition-state-insert-refines-min-update[OF inv, of x b]
    unfolding view .
  show ?thesis
    using inv' view' unfolding partition-state-refines-def by blast
qed

lemma partition-state-batch-prepend-invar:
  assumes partition-state-invar P
  shows partition-state-invar (partition-state-batch-prepend xs P)
  using assms
proof (induction xs arbitrary: P)
  case Nil
  then show ?case
    unfolding partition-state-batch-prepend-def by simp
next
  case (Cons xb xs)
  obtain x b where xb: xb = (x, b)
    by force
  have step: partition-state-invar (partition-state-insert x b P)
    by (rule partition-state-insert-invar[OF Cons.prems])
  have unfold-step:
    partition-state-batch-prepend (xb # xs) P =
      partition-state-batch-prepend xs (partition-state-insert x b P)
    unfolding partition-state-batch-prepend-def xb by simp
  show ?case
    unfolding unfold-step
    by (rule Cons.IH[OF step])
qed

theorem partition-state-batch-prepend-refines-batch-min-update:
  assumes partition-state-invar P
  shows partition-state-view (partition-state-batch-prepend xs P) =
    batch-min-update (partition-state-view P) xs
  using assms
proof (induction xs arbitrary: P)
  case Nil
  then show ?case
    unfolding partition-state-batch-prepend-def batch-min-update-def by simp
next
```

```
  case (Cons xb xs)
  obtain x b where xb: xb = (x, b)
    by force
  have step-inv: partition-state-invar (partition-state-insert x b P)
    by (rule partition-state-insert-invar[OF Cons.prems])
  have step-view:
    partition-state-view (partition-state-insert x b P) =
      min-update (partition-state-view P) x b
    by (rule partition-state-insert-refines-min-update[OF Cons.prems])
  have unfold-state:
    partition-state-batch-prepend (xb # xs) P =
      partition-state-batch-prepend xs (partition-state-insert x b P)
    unfolding partition-state-batch-prepend-def xb by simp
  have unfold-view:
    batch-min-update (partition-state-view P) (xb # xs) =
      batch-min-update (min-update (partition-state-view P) x b) xs
    unfolding batch-min-update-def xb by simp
  show ?case
    unfolding unfold-state unfold-view
    using Cons.IH[OF step-inv] step-view by simp
qed

theorem partition-state-batch-prepend-refines-view:
  assumes ref: partition-state-refines P D
  shows partition-state-refines (partition-state-batch-prepend xs P)
    (batch-min-update D xs)
proof −
  have inv: partition-state-invar P
    and view: partition-state-view P = D
    using ref unfolding partition-state-refines-def by blast+
  have inv': partition-state-invar (partition-state-batch-prepend xs P)
    by (rule partition-state-batch-prepend-invar[OF inv])
  have view': partition-state-view (partition-state-batch-prepend xs P) =
      batch-min-update D xs
    using partition-state-batch-prepend-refines-batch-min-update[OF inv, of xs]
    unfolding view .
  show ?thesis
    using inv' view' unfolding partition-state-refines-def by blast
qed

lemma partition-state-delete-keys-invar:
  assumes partition-state-invar P
  shows partition-state-invar (partition-state-delete-keys S P)
  using assms
  unfolding partition-state-delete-keys-def partition-state-invar-def
  by (auto simp: distinct-map-filter sorted-wrt-filter)

lemma partition-state-delete-keys-view:
  partition-state-view (partition-state-delete-keys S P) =
```

```
    (| keys-of = keys-of (partition-state-view P) − S,
      value-of = value-of (partition-state-view P) |)
  unfolding partition-state-delete-keys-def partition-state-view-def
  by (auto simp: entry-keys-def)

lemma partition-state-pull-prefix-subset:
  set (partition-state-pull-prefix M P) ⊆ set (entries-of P)
   unfolding partition-state-pull-prefix-def by (auto simp: Let-def dest: set-takeWhileD)

lemma partition-state-pull-set-subset:
  partition-state-pull-set M P ⊆ keys-of (partition-state-view P)
  using partition-state-pull-prefix-subset[of M P]
  unfolding partition-state-pull-set-def partition-state-view-def entry-keys-def by
auto

lemma partition-state-pull-prefix-card-le:
  assumes inv: partition-state-invar P
  shows card (partition-state-pull-set M P) ≤ M
proof (cases length (entries-of P) ≤ M)
  case True
  have card (partition-state-pull-set M P) =
      card (set (map fst (entries-of P)))
    unfolding partition-state-pull-set-def partition-state-pull-prefix-def
      entry-keys-def using True by (simp add: Let-def)
  also have ... = length (entries-of P)
  proof −
    have distinct (map fst (entries-of P))
      using inv unfolding partition-state-invar-def by blast
    then have card (set (map fst (entries-of P))) =
        length (map fst (entries-of P))
      by (rule distinct-card)
    then show ?thesis
      by simp
  qed
  finally show ?thesis
    using True by simp
next
  case False
  let ?xs = entries-of P
  let ?beta = snd (?xs ! M)
  let ?pref = takeWhile (λp. snd p < ?beta) ?xs
  have M-lt: M < length ?xs
    using False by simp
  have len-pref: length ?pref ≤ M
  proof (rule ccontr)
    assume ¬ length ?pref ≤ M
    then have M-pref: M < length ?pref
      by simp
```

```
    have ?pref ! M = ?xs ! M
      by (rule takeWhile-nth[OF M-pref])
    moreover have ?pref ! M ∈ set ?pref
      using M-pref by simp
    ultimately show False
      using set-takeWhileD[of ?pref ! M λp. snd p < ?beta ?xs]
      by simp
  qed
  have card (partition-state-pull-set M P) ≤ length ?pref
  proof −
    have card (fst ‘ set ?pref) ≤ card (set ?pref)
      by (rule card-image-le) simp
    also have ... ≤ length ?pref
      by (rule card-length)
    finally show ?thesis
      unfolding partition-state-pull-set-def partition-state-pull-prefix-def
        entry-keys-def using False by (simp add: Let-def)
  qed
  then show ?thesis
    using len-pref by linarith
qed

lemma sorted-wrt-takeWhile-less-eq-filter:
  fixes f :: 'a ⇒ 'b::linorder
  assumes sorted: sorted-wrt (λx y. f x ≤ f y) xs
  shows takeWhile (λx. f x < t) xs = filter (λx. f x < t) xs
proof (rule takeWhile-eq-filter)
  fix x
  assume x-drop: x ∈ set (dropWhile (λx. f x < t) xs)
  let ?pref = takeWhile (λx. f x < t) xs
  let ?drop = dropWhile (λx. f x < t) xs
  obtain i where i: i < length ?drop x = ?drop ! i
    using x-drop by (auto simp: in-set-conv-nth)
  have xs-split: ?pref @ ?drop = xs
    by simp
  have len-xs: length xs = length ?pref + length ?drop
    by (metis length-append xs-split)
  then have idx: length ?pref + i < length xs
    using i len-xs by simp
  have pref-len: length ?pref < length xs
    using idx by simp
  have not-first: ¬ f (xs ! length ?pref) < t
    using nth-length-takeWhile[of λx. f x < t xs] pref-len by simp
  have f (xs ! length ?pref) ≤ f (xs ! (length ?pref + i))
  proof (cases i)
    case 0
    then show ?thesis by simp
  next
    case (Suc j)
```

```
    have length ?pref < length ?pref + i
      using Suc by simp
    then show ?thesis
      by (rule sorted-wrt-nth-less[OF sorted]) (use pref-len idx in simp-all)
  qed
  moreover have x = xs ! (length ?pref + i)
  proof −
    have ?drop ! i = (?pref @ ?drop) ! (length ?pref + i)
      by (simp only: nth-append-length-plus)
    also have ... = xs ! (length ?pref + i)
      using xs-split by simp
    finally show ?thesis
      using i by simp
  qed
  ultimately show ¬ f x < t
    using not-first by auto
qed

lemma partition-state-pull-set-exact:
  assumes inv: partition-state-invar P
    and upper: ⋀u. u ∈ keys-of (partition-state-view P) ⟹
      value-of (partition-state-view P) u < B
  shows partition-state-pull-set M P =
    {u ∈ keys-of (partition-state-view P).
      value-of (partition-state-view P) u < partition-state-pull-bound M B P}
proof (cases length (entries-of P) ≤ M)
  case True
  then have pull-all:
    partition-state-pull-set M P = keys-of (partition-state-view P)
    unfolding partition-state-pull-set-def partition-state-pull-prefix-def
      partition-state-view-def by (simp add: Let-def)
  moreover have partition-state-pull-bound M B P = B
    using True unfolding partition-state-pull-bound-def by (simp add: Let-def)
  ultimately show ?thesis
    using upper by auto
next
  case False
  let ?xs = entries-of P
  let ?beta = snd (?xs ! M)
  have beta: partition-state-pull-bound M B P = ?beta
    using False unfolding partition-state-pull-bound-def by (simp add: Let-def)
  have sorted: sorted-wrt (λp q. snd p ≤ snd q) ?xs
    using inv unfolding partition-state-invar-def by blast
  have filter-eq:
    takeWhile (λp. snd p < ?beta) ?xs =
      filter (λp. snd p < ?beta) ?xs
    by (rule sorted-wrt-takeWhile-less-eq-filter[OF sorted])
  have consistent:
    ⋀x b. (x, b) ∈ set ?xs ⟹ values-of P x = b
```

```
    using inv unfolding partition-state-invar-def by auto
  show ?thesis
    unfolding partition-state-pull-set-def partition-state-pull-prefix-def
      partition-state-view-def entry-keys-def beta using False filter-eq consistent
    by (auto simp: Let-def)
qed

theorem partition-state-pull-refines-pull-separates:
  assumes inv: partition-state-invar P
    and upper: ⋀u. u ∈ keys-of (partition-state-view P) ⟹
      value-of (partition-state-view P) u < B
    and pull: partition-state-pull M B P = (S, beta, P')
  shows pull-separates (partition-state-view P) M B S beta
    (partition-state-view P')
proof −
  have S-def: S = partition-state-pull-set M P
    and beta-def: beta = partition-state-pull-bound M B P
    and P'-def: P' = partition-state-delete-keys S P
    using pull unfolding partition-state-pull-def by (auto simp: Let-def)
  have S-subset:
    S ⊆ keys-of (partition-state-view P)
    unfolding S-def by (rule partition-state-pull-set-subset)
  have card-S: card S ≤ M
    unfolding S-def by (rule partition-state-pull-prefix-card-le[OF inv])
  have P'-keys:
    keys-of (partition-state-view P') = keys-of (partition-state-view P) − S
    unfolding P'-def partition-state-delete-keys-view by simp
  have P'-values:
    value-of (partition-state-view P') = value-of (partition-state-view P)
    unfolding P'-def partition-state-delete-keys-view by simp
  have exact:
    S = {u ∈ keys-of (partition-state-view P).
      value-of (partition-state-view P) u < beta}
    unfolding S-def beta-def
    by (rule partition-state-pull-set-exact[OF inv upper])
  have pulled-before-remaining:
    ⋀u v. ⟦u ∈ S; v ∈ keys-of (partition-state-view P')⟧ ⟹
      value-of (partition-state-view P) u ≤
      value-of (partition-state-view P') v
  proof −
    fix u v
    assume u: u ∈ S and v: v ∈ keys-of (partition-state-view P')
    have value-of (partition-state-view P) u < beta
      using u exact by blast
    moreover have ¬ value-of (partition-state-view P) v < beta
      using v P'-keys exact by blast
    ultimately show
      value-of (partition-state-view P) u ≤
        value-of (partition-state-view P') v
```

```
    using P'-values by simp
  qed
  show ?thesis
  proof (cases keys-of (partition-state-view P') = {})
    case True
    have beta-B: beta = B
    proof (cases length (entries-of P) ≤ M)
      case True
      then show ?thesis
        using beta-def unfolding partition-state-pull-bound-def by (simp add:
Let-def)
    next
      case False
      let ?xs = entries-of P
      have M-lt: M < length ?xs
        using False by simp
      have nth-key: fst (?xs ! M) ∈ keys-of (partition-state-view P)
        unfolding partition-state-view-def entry-keys-def using nth-mem[OF M-lt]
by auto
      have nth-not-S: fst (?xs ! M) ∉ S
      proof −
        have beta-eq: beta = snd (?xs ! M)
          using beta-def False unfolding partition-state-pull-bound-def by (simp
add: Let-def)
        have value-eq:
          value-of (partition-state-view P) (fst (?xs ! M)) = snd (?xs ! M)
          using inv nth-mem[OF M-lt]
          unfolding partition-state-invar-def partition-state-view-def by auto
        show ?thesis
          using exact beta-eq value-eq by auto
      qed
      then have fst (?xs ! M) ∈ keys-of (partition-state-view P')
        using nth-key P'-keys by blast
      then show ?thesis
        using True by blast
    qed
    show ?thesis
      unfolding pull-separates-def
      using S-subset card-S P'-keys P'-values True beta-B
        pulled-before-remaining by auto
  next
    case False
    have upper-pulled:
      ⋀u. u ∈ S ⟹ value-of (partition-state-view P) u < beta
      using exact by blast
    have lower-remaining:
      ⋀v. v ∈ keys-of (partition-state-view P') ⟹
        beta ≤ value-of (partition-state-view P') v
      using exact P'-keys P'-values by fastforce
```

```
    show ?thesis
      unfolding pull-separates-def
      using S-subset card-S P'-keys P'-values False pulled-before-remaining
        upper-pulled lower-remaining by auto
  qed
qed

theorem partition-state-pull-refines-view:
  assumes ref: partition-state-refines P D
    and upper: ⋀u. u ∈ keys-of D ⟹ value-of D u < B
    and pull: partition-state-pull M B P = (S, beta, P')
  defines D' ≡ (| keys-of = keys-of D − S, value-of = value-of D |)
  shows partition-state-refines P' D' ∧ pull-separates D M B S beta D'
proof −
  have inv: partition-state-invar P
    and view: partition-state-view P = D
    using ref unfolding partition-state-refines-def by blast+
  have S-def: S = partition-state-pull-set M P
    and P'-def: P' = partition-state-delete-keys S P
    using pull unfolding partition-state-pull-def by (auto simp: Let-def)
  have inv': partition-state-invar P'
    unfolding P'-def by (rule partition-state-delete-keys-invar[OF inv])
  have sep:
    pull-separates (partition-state-view P) M B S beta
      (partition-state-view P')
  proof (rule partition-state-pull-refines-pull-separates[OF inv - pull])
    fix u
    assume u ∈ keys-of (partition-state-view P)
    then show value-of (partition-state-view P) u < B
      using upper unfolding view by simp
  qed
  have view': partition-state-view P' = D'
  proof −
    show ?thesis
      unfolding P'-def D'-def partition-state-delete-keys-view view by simp
  qed
  show ?thesis
    using inv' view' sep unfolding partition-state-refines-def view D'-def by simp
qed

theorem partition-state-pull-invar:
  assumes inv: partition-state-invar P
    and pull: partition-state-pull M B P = (S, beta, P')
  shows partition-state-invar P'
  using pull partition-state-delete-keys-invar[OF inv]
  unfolding partition-state-pull-def by (auto simp: Let-def)

theorem costed-partition-state-insert-refines:
  assumes inv: partition-state-invar P
```

```
    and op: costed-partition-state-insert t P x b c P'
  shows partition-state-invar P' ∧
    insert-spec (partition-state-view P) x b (partition-state-view P') ∧
    partition-insert-cost-bound c t
proof −
  have P'-def: P' = partition-state-insert x b P
    and c-def: c = t
    using op unfolding costed-partition-state-insert-def by blast+
  have inv': partition-state-invar P'
    unfolding P'-def by (rule partition-state-insert-invar[OF inv])
  have spec:
    insert-spec (partition-state-view P) x b (partition-state-view P')
    unfolding P'-def by (rule partition-state-insert-refines-insert-spec[OF inv])
  have cost: partition-insert-cost-bound c t
    unfolding c-def partition-insert-cost-bound-def by (rule order-refl)
  show ?thesis
    using inv' spec cost by blast
qed

theorem costed-partition-state-batch-prepend-refines:
  assumes inv: partition-state-invar P
    and op: costed-partition-state-batch-prepend t P xs c P'
  shows partition-state-invar P' ∧
    partition-state-view P' = batch-min-update (partition-state-view P) xs ∧
    partition-batch-cost-bound c t xs
proof −
  have P'-def: P' = partition-state-batch-prepend xs P
    and c-def: c = t * length xs
    using op unfolding costed-partition-state-batch-prepend-def by blast+
  have inv': partition-state-invar P'
    unfolding P'-def by (rule partition-state-batch-prepend-invar[OF inv])
  have view:
    partition-state-view P' = batch-min-update (partition-state-view P) xs
    unfolding P'-def
    by (rule partition-state-batch-prepend-refines-batch-min-update[OF inv])
  have cost: partition-batch-cost-bound c t xs
    unfolding c-def partition-batch-cost-bound-def by (rule order-refl)
  show ?thesis
    using inv' view cost by blast
qed

theorem costed-partition-state-pull-refines:
  assumes inv: partition-state-invar P
    and upper: ⋀u. u ∈ keys-of (partition-state-view P) ⟹
      value-of (partition-state-view P) u < B
    and op: costed-partition-state-pull M B P S beta P' c
  shows partition-state-invar P' ∧
    pull-separates (partition-state-view P) M B S beta
      (partition-state-view P') ∧
```

```
    partition-pull-cost-bound c S
proof −
  have pull: partition-state-pull M B P = (S, beta, P')
    and c-def: c = card S
    using op unfolding costed-partition-state-pull-def by blast+
  have inv': partition-state-invar P'
    by (rule partition-state-pull-invar[OF inv pull])
  have sep:
    pull-separates (partition-state-view P) M B S beta
      (partition-state-view P')
    by (rule partition-state-pull-refines-pull-separates[OF inv upper pull])
  have cost: partition-pull-cost-bound c S
    unfolding c-def partition-pull-cost-bound-def by (rule order-refl)
  show ?thesis
    using inv' sep cost by blast
qed

context unique-shortest-digraph
begin

lemma partition-state-from-keys-label-view:
  partition-state-view (partition-state-from-keys d xs) =
    label-partition-view d (set xs)
  unfolding partition-state-from-keys-view label-partition-view-def by simp

theorem partition-state-pull-label-set-eq-split-below:
  assumes inv: partition-state-invar P_D
    and view: partition-state-view P_D = label-partition-view d S
    and upper: ⋀u. u ∈ S ⟹ d u < B
    and pull: partition-state-pull M B P_D = (S-pull, beta, P_D')
  shows S-pull = split-below d S beta
proof −
  have pull-sep:
    pull-separates (partition-state-view P_D) M B S-pull beta
      (partition-state-view P_D')
  proof (rule partition-state-pull-refines-pull-separates[OF inv - pull])
    fix u
    assume u ∈ keys-of (partition-state-view P_D)
    then show value-of (partition-state-view P_D) u < B
      using upper unfolding view by simp
  qed
  show ?thesis
    using pull-separates-label-set-eq-split-below[of d S M B S-pull beta
        partition-state-view P_D'] pull-sep upper
    unfolding view .
qed

theorem partition-state-pull-establishes-lower-pre:
  assumes inv: partition-state-invar P_D
```

```
    and view: partition-state-view P_D = label-partition-view d S
    and pre: bmssp-pre-full d S (Fin B)
    and upper: ⋀u. u ∈ S ⟹ d u < B
    and pull: partition-state-pull M B P_D = (S-pull, beta, P_D')
  shows bmssp-pre-full d S-pull (Fin beta)
proof −
  have pull-eq: S-pull = split-below d S beta
    by (rule partition-state-pull-label-set-eq-split-below[OF inv view upper pull])
  have pull-sep:
    pull-separates (partition-state-view P_D) M B S-pull beta
      (partition-state-view P_D')
  proof (rule partition-state-pull-refines-pull-separates[OF inv - pull])
    fix u
    assume u ∈ keys-of (partition-state-view P_D)
    then show value-of (partition-state-view P_D) u < B
      using upper unfolding view by simp
  qed
  show ?thesis
    unfolding pull-eq
  proof (cases keys-of (partition-state-view P_D') = {})
    case True
    have beta = B
      using pull-separates-empty-bound[OF pull-sep True] .
    moreover have split-below d S B = S
      using upper unfolding split-below-def by auto
    then show bmssp-pre-full d (split-below d S beta) (Fin beta)
      using pre calculation by simp
  next
    case False
    obtain v where v: v ∈ keys-of (partition-state-view P_D')
      using False by blast
    have beta-le-B: beta < B
    proof −
      have beta-le: beta ≤ value-of (partition-state-view P_D') v
        by (rule pull-separates-nonempty-bound[OF pull-sep False v])
      have v-old: v ∈ S
        using v pull-sep unfolding pull-separates-def view by auto
      have val-eq: value-of (partition-state-view P_D') v = d v
        using pull-sep unfolding pull-separates-def view by auto
      show ?thesis
        using beta-le upper[OF v-old] val-eq by linarith
    qed
    show bmssp-pre-full d (split-below d S beta) (Fin beta)
      by (rule pull-minimum-pre-for-lower-split[OF pre]) (simp add: beta-le-B)
  qed
qed

end
```

end
theory *BMSSP-Exact-Concrete-Cost*
imports *BMSSP-Direct-Insert-Costed BMSSP-Partition-Data-Structure*
begin

35 Exact Concrete Cost Runs

The existing direct-insert costed run relation records data-structure costs by upper-bound predicates. The relation below is a concrete-cost refinement layer: pulls and batch-prepends are performed by the concrete partition state operations, whose costs are exact. Its main theorem shows that every exact concrete run is accepted by the existing direct-insert costed relation, so all already-proved graph-time bounds apply to such runs.

Initial insertion is kept as a separate exact initial-state predicate. It records an exact insertion trace and the resulting abstract view; future totality work can instantiate it with a concrete enumeration of the pivot set.

This theory is not the final executable bucketed structure. It is the bridge from abstract cost predicates to concrete partition-state operations. The direct-insert relation says "there exists a pull cost satisfying the pull predicate" and similarly for batch costs. Here those costs are the values returned by *costed-partition-state-pull* and *costed-partition-state-batch-prepend*. The refinement theorem then proves that these exact operations satisfy the abstract cost relation.

The file also contains witness lemmas. They show that an initial partition can be built by inserting the pivot set, that terminal loop cases can be produced directly, and that nonterminal loop cases can be assembled from one concrete pull, one recursive child call, and three concrete batch prepends. These witnesses make the concrete-cost layer a useful target for later totality or executable-search work, even though the main Track 2 deliverable uses the bucketed amortised bounds.

inductive *exact-partition-state-insert-list* ::
  nat ⇒ *'k partition-state* ⇒ (*'k* × *real*) *list* ⇒ *nat* ⇒
    'k partition-state ⇒ *bool* **where**
  Exact-Insert-List-Nil:
    exact-partition-state-insert-list t P [] *0 P*
| *Exact-Insert-List-Cons*:
    costed-partition-state-insert t P x b c-x P_1 ⟹
     exact-partition-state-insert-list t P_1 *xs c-tail* P' ⟹
     c = *c-x* + *c-tail* ⟹
     exact-partition-state-insert-list t P ((*x*, *b*) # *xs*) *c* P'

Exact initial insertion is represented as a list trace. The trace is small enough to reason about directly: each element costs exactly t, preserves the partition invariant, and refines to a minimum-update on the abstract partition view. Building an initial partition from a distinct enumeration of

the pivot set therefore gives the abstract label partition view expected by the BMSSP loop.

lemma *batch-min-update-append* [*simp*]:
  batch-min-update (*batch-min-update D xs*) *ys* = *batch-min-update D* (*xs* @ *ys*)
  unfolding *batch-min-update-def* **by** *simp*

lemma *exact-partition-state-insert-list-cost*:
  assumes *exact-partition-state-insert-list t P xs c P′*
  shows *c* = *t* ∗ *length xs*
  using *assms*
proof (*induction*)
  case *Exact-Insert-List-Nil*
  then show *?case*
    by *simp*
next
  case (*Exact-Insert-List-Cons t P x b c-x* P_1 *xs c-tail P′ c*)
  then show *?case*
    unfolding *costed-partition-state-insert-def* **by** *simp*
qed

lemma *exact-partition-state-insert-list-invar*:
  assumes *inv*: *partition-state-invar P*
    and *inserts*: *exact-partition-state-insert-list t P xs c P′*
  shows *partition-state-invar P′*
  using *inserts inv*
proof (*induction*)
  case (*Exact-Insert-List-Nil t P*)
  then show *?case*
    by *simp*
next
  case (*Exact-Insert-List-Cons t P x b c-x* P_1 *xs c-tail P′ c*)
  have *inv1*: *partition-state-invar* P_1
    using *costed-partition-state-insert-refines*
      [*OF Exact-Insert-List-Cons.prems Exact-Insert-List-Cons.hyps*(*1*)]
    by *blast*
  show *?case*
    by (*rule Exact-Insert-List-Cons.IH*[*OF inv1*])
qed

lemma *exact-partition-state-insert-list-batch-prepend*:
  exact-partition-state-insert-list t P xs (*t* ∗ *length xs*)
    (*partition-state-batch-prepend xs P*)
proof (*induction xs arbitrary*: *P*)
  case *Nil*
  then show *?case*
   unfolding *partition-state-batch-prepend-def* **by** (*simp add*: *Exact-Insert-List-Nil*)
next
  case (*Cons xb xs*)
  obtain *x b* **where** *xb*: *xb* = (*x*, *b*)

```
    by force
  have step:
    costed-partition-state-insert t P x b t (partition-state-insert x b P)
    unfolding costed-partition-state-insert-def by simp
  have tail:
    exact-partition-state-insert-list t (partition-state-insert x b P) xs
      (t * length xs)
      (partition-state-batch-prepend xs (partition-state-insert x b P))
    by (rule Cons.IH)
  have run:
    exact-partition-state-insert-list t P (xb # xs)
      (t + t * length xs)
      (partition-state-batch-prepend xs (partition-state-insert x b P))
    unfolding xb by (rule Exact-Insert-List-Cons[OF step tail refl])
  then show ?case
    unfolding xb partition-state-batch-prepend-def by simp
qed

context unique-shortest-digraph
begin

lemma batch-min-update-label-pairs-value-disjoint:
  assumes distinct: distinct xs
    and disjoint: set xs ∩ keys-of D = {}
  shows value-of (batch-min-update D (map (λx. (x, d x)) xs)) y =
    (if y ∈ set xs then d y else value-of D y)
  using distinct disjoint
proof (induction xs arbitrary: D y)
  case Nil
  then show ?case
    unfolding batch-min-update-def by simp
next
  case (Cons x xs)
  have x-not-D: x ∉ keys-of D
    using Cons.prems by auto
  have xs-disjoint:
    set xs ∩ keys-of (min-update D x (d x)) = {}
    using Cons.prems unfolding min-update-def by auto
  have tail:
    value-of
      (batch-min-update (min-update D x (d x))
        (map (λx. (x, d x)) xs)) y =
      (if y ∈ set xs then d y else value-of (min-update D x (d x)) y)
    by (rule Cons.IH) (use Cons.prems xs-disjoint in auto)
  have unfold:
    batch-min-update D (map (λx. (x, d x)) (x # xs)) =
      batch-min-update (min-update D x (d x)) (map (λx. (x, d x)) xs)
    unfolding batch-min-update-def by simp
  show ?case
```

```
  proof (cases y = x)
    case True
    then have y ∉ set xs
      using Cons.prems(1) by simp
    then show ?thesis
      using tail x-not-D True unfolding unfold by (simp add: min-update-def)
  next
    case False
    then show ?thesis
      using tail unfolding unfold by (simp add: min-update-def)
  qed
qed

lemma min-update-label-partition-view-new:
  assumes x ∉ S
  shows min-update (label-partition-view d S) x (d x) =
    label-partition-view d (insert x S)
  using assms unfolding min-update-def label-partition-view-def by simp

lemma batch-min-update-label-partition-view-disjoint:
  assumes distinct: distinct xs
    and disjoint: set xs ∩ S = {}
  shows batch-min-update (label-partition-view d S)
      (map (λx. (x, d x)) xs) =
    label-partition-view d (S ∪ set xs)
  using distinct disjoint
proof (induction xs arbitrary: S)
  case Nil
  then show ?case
    unfolding batch-min-update-def label-partition-view-def by simp
next
  case (Cons x xs)
  have x-not-S: x ∉ S
    using Cons.prems by auto
  have xs-disjoint: set xs ∩ insert x S = {}
    using Cons.prems by auto
  have step:
    min-update (label-partition-view d S) x (d x) =
      label-partition-view d (insert x S)
    by (rule min-update-label-partition-view-new[OF x-not-S])
  have tail:
    batch-min-update (label-partition-view d (insert x S))
      (map (λx. (x, d x)) xs) =
      label-partition-view d (insert x S ∪ set xs)
    by (rule Cons.IH) (use Cons.prems xs-disjoint in auto)
  show ?case
    unfolding batch-min-update-def
    using step tail[unfolded batch-min-update-def] by auto
qed
```

```
lemma batch-min-update-empty-label-view:
  assumes distinct: distinct xs
  shows batch-min-update (partition-state-view (empty-partition-state d))
      (map (λx. (x, d x)) xs) =
    label-partition-view d (set xs)
proof −
  have empty-view:
    partition-state-view (empty-partition-state d) = label-partition-view d {}
    unfolding label-partition-view-def by simp
  have batch-min-update (partition-state-view (empty-partition-state d))
      (map (λx. (x, d x)) xs) =
    batch-min-update (label-partition-view d {}) (map (λx. (x, d x)) xs)
    unfolding empty-view by simp
  also have ... = label-partition-view d ({} ∪ set xs)
    by (rule batch-min-update-label-partition-view-disjoint[OF distinct]) simp
  finally show ?thesis
    by simp
qed

definition exact-partition-initial-state where
  exact-partition-initial-state t d P P_D c ⟷
    partition-state-invar P_D ∧
    partition-state-view P_D = label-partition-view d P ∧
    c = t * card P ∧
    (∃ xs. distinct xs ∧ set xs = P ∧
      exact-partition-state-insert-list t (empty-partition-state d)
        (map (λx. (x, d x)) xs) c P_D)
```

The predicate *exact-partition-initial-state* packages the initial build phase for a non-base BMSSP call. It records three facts at once: the concrete partition state is well formed, its abstract view is exactly the label view of the pivot set, and the build cost is exactly $t * \mathit{card}\ P$. The existence lemmas below use finite pivot sets to construct such a state by a batch of concrete inserts.

```
lemma exact-partition-initial-state-invar:
  assumes exact-partition-initial-state t d P P_D c
  shows partition-state-invar P_D
  using assms unfolding exact-partition-initial-state-def by blast

lemma exact-partition-initial-state-view:
  assumes exact-partition-initial-state t d P P_D c
  shows partition-state-view P_D = label-partition-view d P
  using assms unfolding exact-partition-initial-state-def by blast

lemma exact-partition-initial-state-cost-bound:
  assumes exact-partition-initial-state t d P P_D c
  shows partition-initial-insert-cost-bound c t P
  using assms unfolding exact-partition-initial-state-def
```

partition-initial-insert-cost-bound-def **by** *simp*

lemma *exact-partition-initial-state-cost*:
assumes *exact-partition-initial-state t d P* P_D *c*
shows $c = t * card\ P$
using *assms* **unfolding** *exact-partition-initial-state-def* **by** *blast*

lemma *exact-partition-initial-state-exists*:
assumes *finite-P*: *finite P*
shows $\exists P_D\ c.$ *exact-partition-initial-state t d P* P_D *c*
proof −
obtain *xs* **where** *xs*: *set xs* = *P distinct xs*
using *finite-distinct-list*[*OF finite-P*] **by** *blast*
let *?pairs* = *map* (λ*x*. (*x*, *d x*)) *xs*
let $?P_D$ = *partition-state-batch-prepend ?pairs* (*empty-partition-state d*)
let *?c* = *t* ∗ *card P*
have *run*:
exact-partition-state-insert-list t (*empty-partition-state d*) *?pairs*
(*t* ∗ *length ?pairs*) $?P_D$
by (*rule exact-partition-state-insert-list-batch-prepend*)
have *len*: *length xs* = *card P*
using *xs finite-P distinct-card* **by** *fastforce*
have *run-c*:
exact-partition-state-insert-list t (*empty-partition-state d*) *?pairs*
?c $?P_D$
using *run len* **by** *simp*
have *inv*: *partition-state-invar* $?P_D$
by (*rule exact-partition-state-insert-list-invar*[*OF empty-partition-state-invar run*])
have *view*: *partition-state-view* $?P_D$ = *label-partition-view d P*
proof −
have *partition-state-view* $?P_D$ =
batch-min-update (*partition-state-view* (*empty-partition-state d*)) *?pairs*
by (*rule partition-state-batch-prepend-refines-batch-min-update*)
(*rule empty-partition-state-invar*)
also have ... = *label-partition-view d* (*set xs*)
by (*rule batch-min-update-empty-label-view*[*OF xs*(*2*)])
also have ... = *label-partition-view d P*
using *xs*(*1*) **by** *simp*
finally show *?thesis* **.**
qed
have *exact-partition-initial-state t d P* $?P_D$ *?c*
unfolding *exact-partition-initial-state-def*
using *inv view run-c xs* **by** *blast*
then show *?thesis*
by *blast*
qed

lemma *exact-partition-initial-state-exists-for-capped-pivots*:

assumes *S-subset*: $S \subseteq V$
shows $\exists P_D\ c.$ *exact-partition-initial-state* t
(*find-pivots-label-capped* k *cap* d S B)
(*find-pivots-pivots-capped* k *cap* d S B) P_D c
proof −
have *pivots-subset-V*:
find-pivots-pivots-capped k *cap* d S $B \subseteq V$
using *find-pivots-pivots-capped-subset S-subset* **by** *blast*
have *finite-pivots*: *finite* (*find-pivots-pivots-capped* k *cap* d S B)
by (*rule finite-subset*[*OF pivots-subset-V finite-V*])
show *?thesis*
by (*rule exact-partition-initial-state-exists*[*OF finite-pivots*])
qed

lemma *exact-partition-initial-state-exists-for-capped-pivots-with-cost*:
assumes *S-subset*: $S \subseteq V$
shows $\exists P_D.$ *exact-partition-initial-state* t
(*find-pivots-label-capped* k *cap* d S B)
(*find-pivots-pivots-capped* k *cap* d S B) P_D
($t *$ *card* (*find-pivots-pivots-capped* k *cap* d S B))
proof −
have *pivots-subset-V*:
find-pivots-pivots-capped k *cap* d S $B \subseteq V$
using *find-pivots-pivots-capped-subset S-subset* **by** *blast*
have *finite-pivots*: *finite* (*find-pivots-pivots-capped* k *cap* d S B)
by (*rule finite-subset*[*OF pivots-subset-V finite-V*])
obtain P_D c **where** *init*: *exact-partition-initial-state* t
(*find-pivots-label-capped* k *cap* d S B)
(*find-pivots-pivots-capped* k *cap* d S B) P_D c
using *exact-partition-initial-state-exists*[*OF finite-pivots*] **by** *blast*
have $c = t *$ *card* (*find-pivots-pivots-capped* k *cap* d S B)
by (*rule exact-partition-initial-state-cost*[*OF init*])
then show *?thesis*
using *init* **by** *blast*
qed

lemma *find-pivots-pivots-capped-reaches*:
assumes *reaches*: $\forall x{\in}S.$ *reachable* s x
shows $\forall x{\in}$*find-pivots-pivots-capped* k *cap* d S $B.$ *reachable* s x
using *reaches find-pivots-pivots-capped-subset* **by** *blast*

lemma *find-pivots-pivots-capped-label-less-finite*:
assumes *below*: $\bigwedge y.\ y \in S \Longrightarrow d\ y < Bmax$
and *xP*: $x \in$ *find-pivots-pivots-capped* k *cap* d S (*Fin Bmax*)
shows *find-pivots-label-capped* k *cap* d S (*Fin Bmax*) $x < Bmax$
proof −
have *below-bound*: $\bigwedge y.\ y \in S \Longrightarrow$ *below-bound* (d y) (*Fin Bmax*)
using *below* **by** *simp*
have *below-bound* (*find-pivots-label-capped* k *cap* d S (*Fin Bmax*) x)

```
    (Fin Bmax)
   by (rule find-pivots-pivots-capped-label-below[OF below-bound xP])
 then show ?thesis
   by simp
qed
```

The main exact concrete run relation follows the same tree as the direct-insert costed relation, but it keeps concrete partition states in the loop. A step performs a concrete pull, recursively solves the pulled lower split, then performs three concrete batch prepends: direct edge relaxations, lower edge relaxations, and source labels. The accumulated cost is not merely bounded by predicates; it is the exact sum of the concrete operation costs, the child cost, and the tail-loop cost.

This exactness is useful for refinement. The direct-insert relation is the mature cost interface used by the graph-time theorems. By proving that every exact concrete run refines to that relation, all existing correctness and runtime results become available for the concrete partition-state execution.

```
inductive exact-concrete-partition-loop-state
 and exact-concrete-bmssp where
 Exact-Concrete-State-Done:
   keys-of (partition-state-view D) = {} ⟹
    bound-le (Fin a) B ⟹
    complete-on d' (bound-tree P B) ⟹
    exact-concrete-partition-loop-state Δ M-of t h k cap l d P B d' D a
      [] [] B [range-tree P a B]
      (bound-tree P (Fin a) ∪ ⋃(set [range-tree P a B])) 0 []
| Exact-Concrete-State-Stop:
   bound-le (Fin a) B ⟹
    complete-on d' (bound-tree P (Fin a)) ⟹
    k * cap ≤ card (bound-tree P (Fin a)) ⟹
    exact-concrete-partition-loop-state Δ M-of t h k cap l d P B d' D a
      [] [] (Fin a) [range-tree P a (Fin a)]
      (bound-tree P (Fin a) ∪ ⋃(set [range-tree P a (Fin a)])) 0 []
| Exact-Concrete-State-Step:
   partition-state-invar D ⟹
    partition-upper-bound (partition-state-view D) Bmax ⟹
    costed-partition-state-pull (M-of l) Bmax D S-pull beta D-pull c-pull ⟹
    bound-le (Fin beta) B ⟹
    bmssp-pre-full d S-pull (Fin beta) ⟹
    S-pull = split-below d P beta ⟹
    (∀ x∈S-pull. reachable s x) ⟹
    a ≤ b ⟹
    bound-le (Fin a) B' ⟹
    complete-on d' (bound-tree P (Fin a)) ⟹
    card (bound-tree P (Fin a)) < k * cap ⟹
    exact-concrete-bmssp Δ M-of t h k cap l d S-pull (Fin beta)
      d-child (Fin b) U-child c-child ⟹
    U-child = range-tree P a (Fin b) ⟹
```

complete-preserved d-child d′ U-child $\Longrightarrow$
direct-edge-batch $=$ *edge-relaxation-pairs-in-bound d-child U-child beta B* $\Longrightarrow$
lower-edge-batch $=$ *edge-relaxation-pairs-between d-child U-child b beta* $\Longrightarrow$
source-batch $=$ *label-pairs-between d S-pull b beta* $\Longrightarrow$
costed-partition-state-batch-prepend t D-pull direct-edge-batch c-direct D-direct
$\Longrightarrow$
costed-partition-state-batch-prepend h D-direct lower-edge-batch c-lower D-lower
$\Longrightarrow$
costed-partition-state-batch-prepend h D-lower source-batch c-sources D-next
$\Longrightarrow$
exact-concrete-partition-loop-state Δ *M-of t h k cap l d P B d′*
D-next b betas bs B′ Us-tail U-tail c-tail child-costs-tail $\Longrightarrow$
$c =$ *c-pull* $+$ *c-direct* $+$ *c-lower* $+$ *c-sources* $+$ *c-child* $+$ *c-tail* $\Longrightarrow$
exact-concrete-partition-loop-state Δ *M-of t h k cap l d P B d′ D a*
(*beta* $\#$ *betas*) (*b* $\#$ *bs*) *B′*
(*range-tree P a* (*Fin b*) $\#$ *Us-tail*)
(*bound-tree P* (*Fin a*) $\cup$
$\bigcup$(*set* (*range-tree P a* (*Fin b*) $\#$ *Us-tail*))) *c*
(*c-child* $\#$ *child-costs-tail*)
| *Exact-Concrete-Base*:
$S = \{x\} \Longrightarrow$
exact-concrete-bmssp Δ *M-of t h k cap 0 d S B*
(λv. *if* $v \in$ *base-case-vertices k x B then dist s v else d v*)
(*base-case-bound k x B*)
(*base-case-vertices k x B*)
(*base-case-scan-cost* Δ *k x B*)
| *Exact-Concrete-Step*:
exact-partition-initial-state t
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *D c-insert* $\Longrightarrow$
exact-concrete-partition-loop-state Δ *M-of t h k cap l*
(*find-pivots-label-capped k cap d S B*)
(*find-pivots-pivots-capped k cap d S B*) *B d′ D a*
betas bs B′ Us U-loop c-loop child-costs-loop $\Longrightarrow$
complete-on d′
$\{v \in$ *bound-tree S B′. find-pivots-label-capped k cap d S B v* $=$ *dist s v*$\}$ $\Longrightarrow$
$U =$ *U-loop* $\cup$
$\{v \in$ *bound-tree S B′. find-pivots-label-capped k cap d S B v* $=$ *dist s v*$\}$ $\Longrightarrow$
$c =$ *fp-iter-capped-scan-cost k cap d S S B* $+$ *c-insert* $+$ *c-loop* $\Longrightarrow$
exact-concrete-bmssp Δ *M-of t h k cap* (*Suc l*) *d S B d′ B′ U c*

inductive-cases *exact-concrete-bmssp-zeroE*:
exact-concrete-bmssp Δ *M-of t h k cap 0 d S B d′ B′ U c*

inductive-cases *exact-concrete-bmssp-SucE*:
exact-concrete-bmssp Δ *M-of t h k cap* (*Suc l*) *d S B d′ B′ U c*

lemma *exact-partition-initial-state-cost-unique*:
assumes *exact-partition-initial-state t d P* P_D *c*

```
    and exact-partition-initial-state t d P P_D' c'
  shows c = c'
  using exact-partition-initial-state-cost[OF assms(1)]
    exact-partition-initial-state-cost[OF assms(2)]
  by simp

lemma exact-partition-initial-state-upper-bound:
  assumes init: exact-partition-initial-state t d P P_D c
    and upper: ⋀u. u ∈ P ⟹ d u < B
  shows partition-upper-bound (partition-state-view P_D) B
  using exact-partition-initial-state-view[OF init] upper
  unfolding partition-upper-bound-def by simp

lemma exact-partition-initial-state-pull-split:
  assumes init: exact-partition-initial-state t d P P_D c
    and upper: ⋀u. u ∈ P ⟹ d u < B
    and pull: partition-state-pull M B P_D = (S-pull, beta, P_D')
  shows S-pull = split-below d P beta
proof −
  have inv: partition-state-invar P_D
    by (rule exact-partition-initial-state-invar[OF init])
  have view: partition-state-view P_D = label-partition-view d P
    by (rule exact-partition-initial-state-view[OF init])
  show ?thesis
    by (rule partition-state-pull-label-set-eq-split-below
      [OF inv view upper pull])
qed

lemma exact-partition-initial-state-pull-exact-data:
  assumes init: exact-partition-initial-state t d P P_D c
    and pre: bmssp-pre-full d P (Fin B)
    and upper: ⋀u. u ∈ P ⟹ d u < B
    and pull: partition-state-pull M B P_D = (S-pull, beta, P_D')
  shows costed-partition-state-pull M B P_D S-pull beta P_D' (card S-pull) ∧
    S-pull = split-below d P beta ∧
    bmssp-pre-full d S-pull (Fin beta) ∧
    partition-state-invar P_D'
proof −
  have inv: partition-state-invar P_D
    by (rule exact-partition-initial-state-invar[OF init])
  have view: partition-state-view P_D = label-partition-view d P
    by (rule exact-partition-initial-state-view[OF init])
  have costed:
    costed-partition-state-pull M B P_D S-pull beta P_D' (card S-pull)
    using pull unfolding costed-partition-state-pull-def by simp
  have split: S-pull = split-below d P beta
    by (rule partition-state-pull-label-set-eq-split-below
      [OF inv view upper pull])
  have lower-pre: bmssp-pre-full d S-pull (Fin beta)
```

```
    by (rule partition-state-pull-establishes-lower-pre
      [OF inv view pre upper pull])
  have inv': partition-state-invar P_D'
    by (rule partition-state-pull-invar[OF inv pull])
  show ?thesis
    using costed split lower-pre inv' by blast
qed

lemma exact-partition-initial-state-pull-reaches:
  assumes init: exact-partition-initial-state t d P P_D c
    and upper: ⋀u. u ∈ P ⟹ d u < B
    and pull: partition-state-pull M B P_D = (S-pull, beta, P_D')
    and P-reaches: ∀x∈P. reachable s x
  shows ∀x∈S-pull. reachable s x
proof −
  have S-pull = split-below d P beta
    by (rule exact-partition-initial-state-pull-split[OF init upper pull])
  then show ?thesis
    using P-reaches unfolding split-below-def by blast
qed

lemma exact-partition-initial-state-pull-beta-bound:
  assumes init: exact-partition-initial-state t d P P_D c
    and upper: ⋀u. u ∈ P ⟹ d u < B
    and pull: partition-state-pull M B P_D = (S-pull, beta, P_D')
  shows bound-le (Fin beta) (Fin B)
proof −
  have inv: partition-state-invar P_D
    by (rule exact-partition-initial-state-invar[OF init])
  have view: partition-state-view P_D = label-partition-view d P
    by (rule exact-partition-initial-state-view[OF init])
  have upper-D: partition-upper-bound (partition-state-view P_D) B
    by (rule exact-partition-initial-state-upper-bound[OF init upper])
  have pull-sep:
    pull-separates (partition-state-view P_D) M B S-pull beta
      (partition-state-view P_D')
  proof (rule partition-state-pull-refines-pull-separates[OF inv - pull])
    fix u
    assume u ∈ keys-of (partition-state-view P_D)
    then show value-of (partition-state-view P_D) u < B
      using upper unfolding view by simp
  qed
  have beta ≤ B
    by (rule pull-separates-bound-le[OF pull-sep upper-D])
  then show ?thesis
    by simp
qed
```

The pull lemmas are the local bridge from a concrete partition state to the BMSSP child problem. Starting from an exact initial state, a concrete pull

refines to *pull-separates* on the abstract label partition view. Therefore the pulled source set is exactly *split-below d P beta*, it inherits the child BMSSP precondition, and its vertices remain reachable. These are the premises required by the recursive exact concrete step.

lemma *exact-partition-initial-state-exists-with-cost*:
assumes *finite-P*: *finite P*
shows $\exists P_D$. *exact-partition-initial-state t d P* P_D *(t * card P)*
proof −
obtain P_D *c* **where** *init*: *exact-partition-initial-state t d P* P_D *c*
using *exact-partition-initial-state-exists*[*OF finite-P*] **by** *blast*
have *c = t * card P*
by (*rule exact-partition-initial-state-cost*[*OF init*])
then show *?thesis*
using *init* **by** *blast*
qed

lemma *exact-concrete-bmssp-base-exists*:
assumes *S = {x}*
shows *exact-concrete-bmssp Δ M-of t h k cap 0 d S B*
(λv. if v ∈ base-case-vertices k x B then dist s v else d v)
(base-case-bound k x B)
(base-case-vertices k x B)
(base-case-scan-cost Δ k x B)
by (*rule Exact-Concrete-Base*[*OF assms*])

lemma *exact-concrete-bmssp-zero-singletonD*:
assumes *run*: *exact-concrete-bmssp Δ M-of t h k cap 0 d {x} B d′ B′ U c*
shows *d′ =*
(λv. if v ∈ base-case-vertices k x B then dist s v else d v) ∧
B′ = base-case-bound k x B ∧
U = base-case-vertices k x B ∧
c = base-case-scan-cost Δ k x B
using *run* **by** *cases auto*

lemma *exact-concrete-bmssp-zero-singleton-cost-unique*:
assumes *run*: *exact-concrete-bmssp Δ M-of t h k cap 0 d {x} B d′ B′ U c*
and *run′*: *exact-concrete-bmssp Δ M-of t h k cap 0 d {x} B d″ B″ U′ c′*
shows *c = c′*
using *exact-concrete-bmssp-zero-singletonD*[*OF run*]
exact-concrete-bmssp-zero-singletonD[*OF run′*]
by *simp*

lemma *exact-concrete-bmssp-zeroD*:
assumes *run*: *exact-concrete-bmssp Δ M-of t h k cap 0 d S B d′ B′ U c*
shows $\exists x$. *S = {x} ∧*
d′ = (λv. if v ∈ base-case-vertices k x B then dist s v else d v) ∧
B′ = base-case-bound k x B ∧
U = base-case-vertices k x B ∧
c = base-case-scan-cost Δ k x B

using *run*
proof *cases*
case (*Exact-Concrete-Base x*)
then show *?thesis*
by *blast*
qed

lemma *exact-concrete-bmssp-zero-deterministic*:
assumes *run*: *exact-concrete-bmssp* Δ *M-of t h k cap 0 d S B d' B' U c*
and *run'*: *exact-concrete-bmssp* Δ *M-of t h k cap 0 d S B d'' B'' U' c'*
shows $d' = d'' \wedge B' = B'' \wedge U = U' \wedge c = c'$
proof −
obtain *x* **where** *x*:
$S = \{x\}$
$d' = (\lambda v.$ *if* $v \in$ *base-case-vertices k x B then dist s v else d v*$)$
$B' =$ *base-case-bound k x B*
$U =$ *base-case-vertices k x B*
$c =$ *base-case-scan-cost* Δ *k x B*
using *exact-concrete-bmssp-zeroD*[*OF run*] **by** *blast*
obtain *y* **where** *y*:
$S = \{y\}$
$d'' = (\lambda v.$ *if* $v \in$ *base-case-vertices k y B then dist s v else d v*$)$
$B'' =$ *base-case-bound k y B*
$U' =$ *base-case-vertices k y B*
$c' =$ *base-case-scan-cost* Δ *k y B*
using *exact-concrete-bmssp-zeroD*[*OF run'*] **by** *blast*
have $x = y$
using *x*(*1*) *y*(*1*) **by** *simp*
then show *?thesis*
using *x y* **by** *simp*
qed

lemma *exact-concrete-bmssp-zero-exists-iff-singleton*:
$(\exists\, d'\ B'\ U\ c.$ *exact-concrete-bmssp* Δ *M-of t h k cap 0 d S B d' B' U c*$)$
$\longleftrightarrow (\exists\, x.\ S = \{x\})$
proof
assume $\exists\, d'\ B'\ U\ c.$
exact-concrete-bmssp Δ *M-of t h k cap 0 d S B d' B' U c*
then obtain *d' B' U c* **where**
run: *exact-concrete-bmssp* Δ *M-of t h k cap 0 d S B d' B' U c*
by *blast*
obtain *x* **where** $S = \{x\}$
using *exact-concrete-bmssp-zeroD*[*OF run*] **by** *blast*
then show $\exists\, x.\ S = \{x\}$
by *blast*
next
assume $\exists\, x.\ S = \{x\}$
then obtain *x* **where** $S = \{x\}$
by *blast*

then show $\exists\, d'\ B'\ U\ c.$
exact-concrete-bmssp Δ *M-of t h k cap 0 d S B d' B' U c*
by (*intro exI*) (*rule exact-concrete-bmssp-base-exists*)
qed

lemma *complete-on-dist*:
complete-on (*dist s*) *U*
unfolding *complete-on-def* **by** *simp*

lemma *exact-concrete-partition-loop-state-done-witness*:
assumes *empty*: *keys-of* (*partition-state-view D*) = {}
and *a-bound*: *bound-le* (*Fin a*) *B*
and *complete*: *complete-on d'* (*bound-tree P B*)
shows *exact-concrete-partition-loop-state* Δ *M-of t h k cap l d P B d'*
D a [] [] *B* [*range-tree P a B*]
(*bound-tree P* (*Fin a*) $\cup \bigcup$(*set* [*range-tree P a B*])) *0* []
by (*rule Exact-Concrete-State-Done*[*OF empty a-bound complete*])

lemma *exact-concrete-partition-loop-state-stop-witness*:
assumes *a-bound*: *bound-le* (*Fin a*) *B*
and *complete*: *complete-on d'* (*bound-tree P* (*Fin a*))
and *threshold*: $k * cap \leq$ *card* (*bound-tree P* (*Fin a*))
shows *exact-concrete-partition-loop-state* Δ *M-of t h k cap l d P B d'*
D a [] [] (*Fin a*) [*range-tree P a* (*Fin a*)]
(*bound-tree P* (*Fin a*) $\cup \bigcup$(*set* [*range-tree P a* (*Fin a*)])) *0* []
by (*rule Exact-Concrete-State-Stop*[*OF a-bound complete threshold*])

lemma *exact-concrete-partition-loop-state-done-exists*:
assumes *empty*: *keys-of* (*partition-state-view D*) = {}
and *a-bound*: *bound-le* (*Fin a*) *B*
and *complete*: *complete-on d'* (*bound-tree P B*)
shows $\exists$ *betas bs B' Us U c child-costs.*
exact-concrete-partition-loop-state Δ *M-of t h k cap l d P B d'*
D a betas bs B' Us U c child-costs
proof −
have *run*:
exact-concrete-partition-loop-state Δ *M-of t h k cap l d P B d'*
D a [] [] *B* [*range-tree P a B*]
(*bound-tree P* (*Fin a*) $\cup \bigcup$(*set* [*range-tree P a B*])) *0* []
by (*rule exact-concrete-partition-loop-state-done-witness*
[*OF empty a-bound complete*])
then show *?thesis*
by *blast*
qed

lemma *exact-concrete-partition-loop-state-stop-exists*:
assumes *a-bound*: *bound-le* (*Fin a*) *B*
and *complete*: *complete-on d'* (*bound-tree P* (*Fin a*))
and *threshold*: $k * cap \leq$ *card* (*bound-tree P* (*Fin a*))

shows $\exists$ *betas bs B′ Us U c child-costs*.
exact-concrete-partition-loop-state Δ *M-of t h k cap l d P B d′*
D a betas bs B′ Us U c child-costs
proof −
have *run*:
exact-concrete-partition-loop-state Δ *M-of t h k cap l d P B d′*
D a [] [] (*Fin a*) [*range-tree P a* (*Fin a*)]
(*bound-tree P* (*Fin a*) ∪ $\bigcup$(*set* [*range-tree P a* (*Fin a*)])) *0* []
by (*rule exact-concrete-partition-loop-state-stop-witness*
[*OF a-bound complete threshold*])
then show *?thesis*
by *blast*
qed

lemma *exact-concrete-partition-loop-state-step-from-concrete-ops*:
assumes *inv-D*: *partition-state-invar D*
and *upper-D*: *partition-upper-bound* (*partition-state-view D*) *Bmax*
and *pull*:
partition-state-pull (*M-of l*) *Bmax D* = (*S-pull*, *beta*, *D-pull*)
and *beta-bound*: *bound-le* (*Fin beta*) *B*
and *child-pre*: *bmssp-pre-full d S-pull* (*Fin beta*)
and *S-pull-def*: *S-pull* = *split-below d P beta*
and *S-pull-reaches*: ∀ *x*∈*S-pull*. *reachable s x*
and *a-le-b*: *a* ≤ *b*
and *a-bound*: *bound-le* (*Fin a*) *B′*
and *complete-prefix*: *complete-on d′* (*bound-tree P* (*Fin a*))
and *below-threshold*: *card* (*bound-tree P* (*Fin a*)) < *k* ∗ *cap*
and *child*:
exact-concrete-bmssp Δ *M-of t h k cap l d S-pull* (*Fin beta*)
d-child (*Fin b*) *U-child c-child*
and *U-child*: *U-child* = *range-tree P a* (*Fin b*)
and *preserved*: *complete-preserved d-child d′ U-child*
and *tail*:
exact-concrete-partition-loop-state Δ *M-of t h k cap l d P B d′*
(*partition-state-batch-prepend* (*label-pairs-between d S-pull b beta*)
(*partition-state-batch-prepend*
(*edge-relaxation-pairs-between d-child U-child b beta*)
(*partition-state-batch-prepend*
(*edge-relaxation-pairs-in-bound d-child U-child beta B*)
D-pull)))
b betas bs B′ Us-tail U-tail c-tail child-costs-tail
shows *exact-concrete-partition-loop-state* Δ *M-of t h k cap l d P B d′ D a*
(*beta* # *betas*) (*b* # *bs*) *B′*
(*range-tree P a* (*Fin b*) # *Us-tail*)
(*bound-tree P* (*Fin a*) ∪
$\bigcup$(*set* (*range-tree P a* (*Fin b*) # *Us-tail*)))
(*card S-pull* +
t ∗ *length* (*edge-relaxation-pairs-in-bound d-child U-child beta B*) +
h ∗ *length* (*edge-relaxation-pairs-between d-child U-child b beta*) +

```
    h * length (label-pairs-between d S-pull b beta) +
    c-child + c-tail)
  (c-child # child-costs-tail)
proof −
 let ?direct = edge-relaxation-pairs-in-bound d-child U-child beta B
 let ?lower = edge-relaxation-pairs-between d-child U-child b beta
 let ?sources = label-pairs-between d S-pull b beta
 let ?D-direct = partition-state-batch-prepend ?direct D-pull
 let ?D-lower = partition-state-batch-prepend ?lower ?D-direct
 let ?D-next = partition-state-batch-prepend ?sources ?D-lower
 have pull-op:
  costed-partition-state-pull (M-of l) Bmax D S-pull beta D-pull
   (card S-pull)
  using pull unfolding costed-partition-state-pull-def by simp
 have direct-op:
  costed-partition-state-batch-prepend t D-pull ?direct
   (t * length ?direct) ?D-direct
  unfolding costed-partition-state-batch-prepend-def by simp
 have lower-op:
  costed-partition-state-batch-prepend h ?D-direct ?lower
   (h * length ?lower) ?D-lower
  unfolding costed-partition-state-batch-prepend-def by simp
 have source-op:
  costed-partition-state-batch-prepend h ?D-lower ?sources
   (h * length ?sources) ?D-next
  unfolding costed-partition-state-batch-prepend-def by simp
 show ?thesis
  by (rule Exact-Concrete-State-Step
   [OF inv-D upper-D pull-op beta-bound child-pre S-pull-def
    S-pull-reaches a-le-b a-bound complete-prefix below-threshold child
    U-child preserved refl refl refl direct-op lower-op source-op tail])
   simp
qed
```

The witness lemmas turn concrete operations into one exact loop step. The most explicit version assumes the concrete pull and the concrete tail state are already known, then constructs the exact step and computes its cost. The initialized version derives the pull split and child precondition from *exact-partition-initial-state*. Later existence lemmas use these builders to cover the terminal, threshold, and recursive cases of a non-base call.

```
lemma exact-concrete-initialized-partition-loop-state-step:
 assumes init: exact-partition-initial-state t d P D c-insert
  and pre: bmssp-pre-full d P (Fin Bmax)
  and upper: ⋀u. u ∈ P ⟹ d u < Bmax
  and pull: partition-state-pull (M-of l) Bmax D = (S-pull, beta, D-pull)
  and beta-bound: bound-le (Fin beta) B
  and S-pull-reaches: ∀x∈S-pull. reachable s x
  and a-le-b: a ≤ b
  and a-bound: bound-le (Fin a) B′
```

```
    and complete-prefix: complete-on d' (bound-tree P (Fin a))
    and below-threshold: card (bound-tree P (Fin a)) < k * cap
    and child:
      exact-concrete-bmssp Δ M-of t h k cap l d S-pull (Fin beta)
        d-child (Fin b) U-child c-child
    and U-child: U-child = range-tree P a (Fin b)
    and preserved: complete-preserved d-child d' U-child
    and tail:
      exact-concrete-partition-loop-state Δ M-of t h k cap l d P B d'
        (partition-state-batch-prepend (label-pairs-between d S-pull b beta)
          (partition-state-batch-prepend
            (edge-relaxation-pairs-between d-child U-child b beta)
            (partition-state-batch-prepend
              (edge-relaxation-pairs-in-bound d-child U-child beta B)
              D-pull)))
        b betas bs B' Us-tail U-tail c-tail child-costs-tail
  shows exact-concrete-partition-loop-state Δ M-of t h k cap l d P B d' D a
    (beta # betas) (b # bs) B'
    (range-tree P a (Fin b) # Us-tail)
    (bound-tree P (Fin a) ∪
      ⋃(set (range-tree P a (Fin b) # Us-tail)))
    (card S-pull +
      t * length (edge-relaxation-pairs-in-bound d-child U-child beta B) +
      h * length (edge-relaxation-pairs-between d-child U-child b beta) +
      h * length (label-pairs-between d S-pull b beta) +
      c-child + c-tail)
    (c-child # child-costs-tail)
proof −
  have inv-D: partition-state-invar D
    by (rule exact-partition-initial-state-invar[OF init])
  have upper-D: partition-upper-bound (partition-state-view D) Bmax
    by (rule exact-partition-initial-state-upper-bound[OF init upper])
  have pull-data:
    costed-partition-state-pull (M-of l) Bmax D S-pull beta D-pull
        (card S-pull) ∧
     S-pull = split-below d P beta ∧
     bmssp-pre-full d S-pull (Fin beta) ∧
     partition-state-invar D-pull
    by (rule exact-partition-initial-state-pull-exact-data
      [OF init pre upper pull])
  have child-pre: bmssp-pre-full d S-pull (Fin beta)
    using pull-data by blast
  have S-pull-def: S-pull = split-below d P beta
    using pull-data by blast
  show ?thesis
    by (rule exact-concrete-partition-loop-state-step-from-concrete-ops
      [OF inv-D upper-D pull beta-bound child-pre S-pull-def
        S-pull-reaches a-le-b a-bound complete-prefix below-threshold child
        U-child preserved tail])
```

qed

lemma *partition-state-pull-three-batch-prepend-invar*:
assumes *inv*: *partition-state-invar D*
and *pull*: *partition-state-pull M B D* = (*S-pull*, *beta*, *D-pull*)
shows *partition-state-invar*
(*partition-state-batch-prepend zs*
(*partition-state-batch-prepend ys*
(*partition-state-batch-prepend xs D-pull*)))
proof −
have *inv-pull*: *partition-state-invar D-pull*
by (*rule partition-state-pull-invar*[*OF inv pull*])
have *inv-xs*: *partition-state-invar* (*partition-state-batch-prepend xs D-pull*)
by (*rule partition-state-batch-prepend-invar*[*OF inv-pull*])
have *inv-ys*:
partition-state-invar
(*partition-state-batch-prepend ys*
(*partition-state-batch-prepend xs D-pull*))
by (*rule partition-state-batch-prepend-invar*[*OF inv-xs*])
show *?thesis*
by (*rule partition-state-batch-prepend-invar*[*OF inv-ys*])
qed

lemma *exact-concrete-initialized-partition-loop-state-step-exists*:
assumes *init*: *exact-partition-initial-state t d P D c-insert*
and *pre*: *bmssp-pre-full d P* (*Fin Bmax*)
and *upper*: ⋀*u*. *u* ∈ *P* ⟹ *d u* < *Bmax*
and *pull*: *partition-state-pull* (*M-of l*) *Bmax D* = (*S-pull*, *beta*, *D-pull*)
and *beta-bound*: *bound-le* (*Fin beta*) *B*
and *S-pull-reaches*: ∀ *x*∈*S-pull*. *reachable s x*
and *a-le-b*: *a* ≤ *b*
and *a-bound*: *bound-le* (*Fin a*) *B′*
and *complete-prefix*: *complete-on d′* (*bound-tree P* (*Fin a*))
and *below-threshold*: *card* (*bound-tree P* (*Fin a*)) < *k* ∗ *cap*
and *child*:
exact-concrete-bmssp Δ *M-of t h k cap l d S-pull* (*Fin beta*)
d-child (*Fin b*) *U-child c-child*
and *U-child*: *U-child* = *range-tree P a* (*Fin b*)
and *preserved*: *complete-preserved d-child d′ U-child*
and *tail*:
exact-concrete-partition-loop-state Δ *M-of t h k cap l d P B d′*
(*partition-state-batch-prepend* (*label-pairs-between d S-pull b beta*)
(*partition-state-batch-prepend*
(*edge-relaxation-pairs-between d-child U-child b beta*)
(*partition-state-batch-prepend*
(*edge-relaxation-pairs-in-bound d-child U-child beta B*)
D-pull)))
b betas bs B′ Us-tail U-tail c-tail child-costs-tail
shows ∃ *c child-costs*.

```
    exact-concrete-partition-loop-state Δ M-of t h k cap l d P B d' D a
      (beta # betas) (b # bs) B'
      (range-tree P a (Fin b) # Us-tail)
      (bound-tree P (Fin a) ∪
        ⋃(set (range-tree P a (Fin b) # Us-tail))) c child-costs
proof −
  have run:
    exact-concrete-partition-loop-state Δ M-of t h k cap l d P B d' D a
      (beta # betas) (b # bs) B'
      (range-tree P a (Fin b) # Us-tail)
      (bound-tree P (Fin a) ∪
        ⋃(set (range-tree P a (Fin b) # Us-tail)))
      (card S-pull +
        t * length (edge-relaxation-pairs-in-bound d-child U-child beta B) +
        h * length (edge-relaxation-pairs-between d-child U-child b beta) +
        h * length (label-pairs-between d S-pull b beta) +
        c-child + c-tail)
      (c-child # child-costs-tail)
    by (rule exact-concrete-initialized-partition-loop-state-step
      [OF init pre upper pull beta-bound S-pull-reaches a-le-b a-bound
        complete-prefix below-threshold child U-child preserved tail])
  show ?thesis
    using run by blast
qed

lemma exact-concrete-initialized-partition-loop-state-step-exists-finite-bound:
  assumes init: exact-partition-initial-state t d P D c-insert
    and pre: bmssp-pre-full d P (Fin Bmax)
    and upper: ⋀u. u ∈ P ⟹ d u < Bmax
    and pull: partition-state-pull (M-of l) Bmax D = (S-pull, beta, D-pull)
    and P-reaches: ∀x∈P. reachable s x
    and a-le-b: a ≤ b
    and a-bound: bound-le (Fin a) B'
    and complete-prefix: complete-on d' (bound-tree P (Fin a))
    and below-threshold: card (bound-tree P (Fin a)) < k * cap
    and child:
      exact-concrete-bmssp Δ M-of t h k cap l d S-pull (Fin beta)
        d-child (Fin b) (range-tree P a (Fin b)) c-child
    and preserved:
      complete-preserved d-child d' (range-tree P a (Fin b))
    and tail:
      exact-concrete-partition-loop-state Δ M-of t h k cap l d P
        (Fin Bmax) d'
        (partition-state-batch-prepend (label-pairs-between d S-pull b beta)
          (partition-state-batch-prepend
            (edge-relaxation-pairs-between d-child
              (range-tree P a (Fin b)) b beta)
            (partition-state-batch-prepend
              (edge-relaxation-pairs-in-bound d-child
```

```
        (range-tree P a (Fin b)) beta (Fin Bmax))
       D-pull)))
    b betas bs B' Us-tail U-tail c-tail child-costs-tail
 shows ∃ c child-costs.
  exact-concrete-partition-loop-state Δ M-of t h k cap l d P
   (Fin Bmax) d' D a (beta # betas) (b # bs) B'
   (range-tree P a (Fin b) # Us-tail)
   (bound-tree P (Fin a) ∪
    ⋃(set (range-tree P a (Fin b) # Us-tail))) c child-costs
proof −
 have beta-bound: bound-le (Fin beta) (Fin Bmax)
  by (rule exact-partition-initial-state-pull-beta-bound[OF init upper pull])
 have S-pull-reaches: ∀ x∈S-pull. reachable s x
  by (rule exact-partition-initial-state-pull-reaches
   [OF init upper pull P-reaches])
 show ?thesis
  by (rule exact-concrete-initialized-partition-loop-state-step-exists
   [OF init pre upper pull beta-bound S-pull-reaches a-le-b a-bound
    complete-prefix below-threshold child refl preserved tail])
qed

lemma exact-concrete-initialized-partition-loop-state-nonterminal-exists:
 assumes init: exact-partition-initial-state t d P D c-insert
  and pre: bmssp-pre-full d P (Fin Bmax)
  and upper: ⋀u. u ∈ P ⟹ d u < Bmax
  and pull: partition-state-pull (M-of l) Bmax D = (S-pull, beta, D-pull)
  and beta-bound: bound-le (Fin beta) B
  and P-reaches: ∀ x∈P. reachable s x
  and complete-prefix: complete-on d' (bound-tree P (Fin a))
  and below-threshold: card (bound-tree P (Fin a)) < k * cap
  and step:
   ∃ d-child b c-child betas bs B' Us-tail U-tail c-tail
     child-costs-tail.
    a ≤ b ∧
    bound-le (Fin a) B' ∧
    exact-concrete-bmssp Δ M-of t h k cap l d S-pull (Fin beta)
     d-child (Fin b) (range-tree P a (Fin b)) c-child ∧
    complete-preserved d-child d' (range-tree P a (Fin b)) ∧
    exact-concrete-partition-loop-state Δ M-of t h k cap l d P B d'
     (partition-state-batch-prepend (label-pairs-between d S-pull b beta)
      (partition-state-batch-prepend
       (edge-relaxation-pairs-between d-child
        (range-tree P a (Fin b)) b beta)
       (partition-state-batch-prepend
        (edge-relaxation-pairs-in-bound d-child
         (range-tree P a (Fin b)) beta B)
        D-pull)))
    b betas bs B' Us-tail U-tail c-tail child-costs-tail
 shows ∃ betas bs B' Us U c child-costs.
```

exact-concrete-partition-loop-state Δ *M-of t h k cap l d P B d′*
D a betas bs B′ Us U c child-costs
proof −
have *S-pull-reaches*: ∀ *x*∈*S-pull*. *reachable s x*
by (*rule exact-partition-initial-state-pull-reaches*
[*OF init upper pull P-reaches*])
obtain *d-child b c-child betas bs B′ Us-tail U-tail c-tail*
child-costs-tail **where**
a-le-b: *a* ≤ *b*
and *a-bound*: *bound-le* (*Fin a*) *B′*
and *child*:
exact-concrete-bmssp Δ *M-of t h k cap l d S-pull* (*Fin beta*)
d-child (*Fin b*) (*range-tree P a* (*Fin b*)) *c-child*
and *preserved*:
complete-preserved d-child d′ (*range-tree P a* (*Fin b*))
and *tail*:
exact-concrete-partition-loop-state Δ *M-of t h k cap l d P B d′*
(*partition-state-batch-prepend* (*label-pairs-between d S-pull b beta*)
(*partition-state-batch-prepend*
(*edge-relaxation-pairs-between d-child*
(*range-tree P a* (*Fin b*)) *b beta*)
(*partition-state-batch-prepend*
(*edge-relaxation-pairs-in-bound d-child*
(*range-tree P a* (*Fin b*)) *beta B*)
D-pull)))
b betas bs B′ Us-tail U-tail c-tail child-costs-tail
using *step* **by** *blast*
have *run*:
∃ *c child-costs*.
exact-concrete-partition-loop-state Δ *M-of t h k cap l d P B d′*
D a (*beta* # *betas*) (*b* # *bs*) *B′*
(*range-tree P a* (*Fin b*) # *Us-tail*)
(*bound-tree P* (*Fin a*) ∪
⋃(*set* (*range-tree P a* (*Fin b*) # *Us-tail*))) *c child-costs*
by (*rule exact-concrete-initialized-partition-loop-state-step-exists*
[*OF init pre upper pull beta-bound S-pull-reaches a-le-b a-bound*
complete-prefix below-threshold child refl preserved tail])
then show *?thesis*
by *blast*
qed

lemma *exact-concrete-initialized-partition-loop-state-nonterminal-exists-finite-bound*:
assumes *init*: *exact-partition-initial-state t d P D c-insert*
and *pre*: *bmssp-pre-full d P* (*Fin Bmax*)
and *upper*: ⋀*u*. *u* ∈ *P* ⟹ *d u* < *Bmax*
and *pull*: *partition-state-pull* (*M-of l*) *Bmax D* = (*S-pull*, *beta*, *D-pull*)
and *P-reaches*: ∀ *x*∈*P*. *reachable s x*
and *complete-prefix*: *complete-on d′* (*bound-tree P* (*Fin a*))

and *below-threshold*: *card* (*bound-tree P* (*Fin a*)) < *k* ∗ *cap*
and *step*:
∃ *d-child b c-child betas bs B′ Us-tail U-tail c-tail*
child-costs-tail.
a ≤ *b* ∧
bound-le (*Fin a*) *B′* ∧
exact-concrete-bmssp Δ *M-of t h k cap l d S-pull* (*Fin beta*)
d-child (*Fin b*) (*range-tree P a* (*Fin b*)) *c-child* ∧
complete-preserved d-child d′ (*range-tree P a* (*Fin b*)) ∧
exact-concrete-partition-loop-state Δ *M-of t h k cap l d P*
(*Fin Bmax*) *d′*
(*partition-state-batch-prepend* (*label-pairs-between d S-pull b beta*)
(*partition-state-batch-prepend*
(*edge-relaxation-pairs-between d-child*
(*range-tree P a* (*Fin b*)) *b beta*)
(*partition-state-batch-prepend*
(*edge-relaxation-pairs-in-bound d-child*
(*range-tree P a* (*Fin b*)) *beta* (*Fin Bmax*))
D-pull)))
b betas bs B′ Us-tail U-tail c-tail child-costs-tail
shows ∃ *betas bs B′ Us U c child-costs*.
exact-concrete-partition-loop-state Δ *M-of t h k cap l d P*
(*Fin Bmax*) *d′ D a betas bs B′ Us U c child-costs*
proof (*rule exact-concrete-initialized-partition-loop-state-nonterminal-exists*
[*OF init pre upper pull - P-reaches complete-prefix below-threshold step*])
show *bound-le* (*Fin beta*) (*Fin Bmax*)
by (*rule exact-partition-initial-state-pull-beta-bound*[*OF init upper pull*])
qed

lemma *exact-concrete-initialized-partition-loop-state-cases-exists-finite-bound*:
assumes *init*: *exact-partition-initial-state t d P D c-insert*
and *pre*: *bmssp-pre-full d P* (*Fin Bmax*)
and *upper*: ⋀*u*. *u* ∈ *P* ⟹ *d u* < *Bmax*
and *P-reaches*: ∀ *x*∈*P*. *reachable s x*
and *cases*:
(*keys-of* (*partition-state-view D*) = {} ∧
bound-le (*Fin a*) (*Fin Bmax*) ∧
complete-on d′ (*bound-tree P* (*Fin Bmax*))) ∨
(*bound-le* (*Fin a*) (*Fin Bmax*) ∧
complete-on d′ (*bound-tree P* (*Fin a*)) ∧
k ∗ *cap* ≤ *card* (*bound-tree P* (*Fin a*))) ∨
(∃ *S-pull beta D-pull*.
partition-state-pull (*M-of l*) *Bmax D* = (*S-pull*, *beta*, *D-pull*) ∧
complete-on d′ (*bound-tree P* (*Fin a*)) ∧
card (*bound-tree P* (*Fin a*)) < *k* ∗ *cap* ∧
(∃ *d-child b c-child betas bs B′ Us-tail U-tail c-tail*
child-costs-tail.
a ≤ *b* ∧
bound-le (*Fin a*) *B′* ∧

```
      exact-concrete-bmssp Δ M-of t h k cap l d S-pull (Fin beta)
        d-child (Fin b) (range-tree P a (Fin b)) c-child ∧
      complete-preserved d-child d' (range-tree P a (Fin b)) ∧
      exact-concrete-partition-loop-state Δ M-of t h k cap l d P
        (Fin Bmax) d'
        (partition-state-batch-prepend
          (label-pairs-between d S-pull b beta)
          (partition-state-batch-prepend
            (edge-relaxation-pairs-between d-child
              (range-tree P a (Fin b)) b beta)
            (partition-state-batch-prepend
              (edge-relaxation-pairs-in-bound d-child
                (range-tree P a (Fin b)) beta (Fin Bmax))
              D-pull)))
          b betas bs B' Us-tail U-tail c-tail child-costs-tail))
  shows ∃ betas bs B' Us U c child-costs.
  exact-concrete-partition-loop-state Δ M-of t h k cap l d P
    (Fin Bmax) d' D a betas bs B' Us U c child-costs
proof −
  consider
    (Done-Case) keys-of (partition-state-view D) = {}
      bound-le (Fin a) (Fin Bmax)
      complete-on d' (bound-tree P (Fin Bmax))
  | (Stop-Case) bound-le (Fin a) (Fin Bmax)
      complete-on d' (bound-tree P (Fin a))
      k * cap ≤ card (bound-tree P (Fin a))
  | (Step-Case) S-pull beta D-pull where
      partition-state-pull (M-of l) Bmax D = (S-pull, beta, D-pull)
      complete-on d' (bound-tree P (Fin a))
      card (bound-tree P (Fin a)) < k * cap
      ∃ d-child b c-child betas bs B' Us-tail U-tail c-tail
          child-costs-tail.
        a ≤ b ∧
        bound-le (Fin a) B' ∧
        exact-concrete-bmssp Δ M-of t h k cap l d S-pull (Fin beta)
          d-child (Fin b) (range-tree P a (Fin b)) c-child ∧
        complete-preserved d-child d' (range-tree P a (Fin b)) ∧
        exact-concrete-partition-loop-state Δ M-of t h k cap l d P
          (Fin Bmax) d'
          (partition-state-batch-prepend (label-pairs-between d S-pull b beta)
            (partition-state-batch-prepend
              (edge-relaxation-pairs-between d-child
                (range-tree P a (Fin b)) b beta)
              (partition-state-batch-prepend
                (edge-relaxation-pairs-in-bound d-child
                  (range-tree P a (Fin b)) beta (Fin Bmax))
                D-pull)))
          b betas bs B' Us-tail U-tail c-tail child-costs-tail
    using cases by blast
```

then show *?thesis*
proof *cases*
case *Done-Case*
show *?thesis*
by (*rule exact-concrete-partition-loop-state-done-exists*
[*OF Done-Case*])
next
case *Stop-Case*
show *?thesis*
by (*rule exact-concrete-partition-loop-state-stop-exists*
[*OF Stop-Case*])
next
case (*Step-Case S-pull beta D-pull*)
show *?thesis*
by (*rule exact-concrete-initialized-partition-loop-state-nonterminal-exists-finite-bound*
[*OF init pre upper Step-Case*(*1*) *P-reaches Step-Case*(*2*)
Step-Case(*3*) *Step-Case*(*4*)])
qed
qed

lemma *exact-concrete-initialized-partition-loop-state-cases-exists-finite-bound-with-complete*:
fixes *W* :: *bound* ⇒ *'a set*
assumes *init*: *exact-partition-initial-state t d P D c-insert*
and *pre*: *bmssp-pre-full d P* (*Fin Bmax*)
and *upper*: ⋀*u*. *u* ∈ *P* ⟹ *d u* < *Bmax*
and *P-reaches*: ∀ *x*∈*P*. *reachable s x*
and *cases*:
(*keys-of* (*partition-state-view D*) = {} ∧
bound-le (*Fin a*) (*Fin Bmax*) ∧
complete-on d' (*bound-tree P* (*Fin Bmax*)) ∧
complete-on d' (*W* (*Fin Bmax*))) ∨
(*bound-le* (*Fin a*) (*Fin Bmax*) ∧
complete-on d' (*bound-tree P* (*Fin a*)) ∧
k ∗ *cap* ≤ *card* (*bound-tree P* (*Fin a*)) ∧
complete-on d' (*W* (*Fin a*))) ∨
(∃ *S-pull beta D-pull B'*.
partition-state-pull (*M-of l*) *Bmax D* = (*S-pull*, *beta*, *D-pull*) ∧
complete-on d' (*bound-tree P* (*Fin a*)) ∧
card (*bound-tree P* (*Fin a*)) < *k* ∗ *cap* ∧
complete-on d' (*W B'*) ∧
(∃ *d-child b c-child betas bs Us-tail U-tail c-tail*
child-costs-tail.
a ≤ *b* ∧
bound-le (*Fin a*) *B'* ∧
exact-concrete-bmssp Δ *M-of t h k cap l d S-pull* (*Fin beta*)
d-child (*Fin b*) (*range-tree P a* (*Fin b*)) *c-child* ∧
complete-preserved d-child d' (*range-tree P a* (*Fin b*)) ∧

```
    exact-concrete-partition-loop-state Δ M-of t h k cap l d P
     (Fin Bmax) d'
     (partition-state-batch-prepend
      (label-pairs-between d S-pull b beta)
      (partition-state-batch-prepend
       (edge-relaxation-pairs-between d-child
        (range-tree P a (Fin b)) b beta)
       (partition-state-batch-prepend
        (edge-relaxation-pairs-in-bound d-child
         (range-tree P a (Fin b)) beta (Fin Bmax))
        D-pull)))
     b betas bs B' Us-tail U-tail c-tail child-costs-tail))
 shows ∃ betas bs B' Us U c child-costs.
   exact-concrete-partition-loop-state Δ M-of t h k cap l d P
    (Fin Bmax) d' D a betas bs B' Us U c child-costs ∧
   complete-on d' (W B')
proof −
 consider
   (Done-Case) keys-of (partition-state-view D) = {}
     bound-le (Fin a) (Fin Bmax)
     complete-on d' (bound-tree P (Fin Bmax))
     complete-on d' (W (Fin Bmax))
 | (Stop-Case) bound-le (Fin a) (Fin Bmax)
     complete-on d' (bound-tree P (Fin a))
     k * cap ≤ card (bound-tree P (Fin a))
     complete-on d' (W (Fin a))
 | (Step-Case) S-pull beta D-pull B' where
     partition-state-pull (M-of l) Bmax D = (S-pull, beta, D-pull)
     complete-on d' (bound-tree P (Fin a))
     card (bound-tree P (Fin a)) < k * cap
     complete-on d' (W B')
     ∃ d-child b c-child betas bs Us-tail U-tail c-tail
        child-costs-tail.
       a ≤ b ∧
       bound-le (Fin a) B' ∧
       exact-concrete-bmssp Δ M-of t h k cap l d S-pull (Fin beta)
        d-child (Fin b) (range-tree P a (Fin b)) c-child ∧
       complete-preserved d-child d' (range-tree P a (Fin b)) ∧
       exact-concrete-partition-loop-state Δ M-of t h k cap l d P
        (Fin Bmax) d'
        (partition-state-batch-prepend (label-pairs-between d S-pull b beta)
         (partition-state-batch-prepend
          (edge-relaxation-pairs-between d-child
           (range-tree P a (Fin b)) b beta)
          (partition-state-batch-prepend
           (edge-relaxation-pairs-in-bound d-child
            (range-tree P a (Fin b)) beta (Fin Bmax))
           D-pull)))
       b betas bs B' Us-tail U-tail c-tail child-costs-tail
```

```
  using cases by blast
then show ?thesis
proof cases
  case Done-Case
  have run:
    exact-concrete-partition-loop-state Δ M-of t h k cap l d P
      (Fin Bmax) d' D a [] [] (Fin Bmax)
      [range-tree P a (Fin Bmax)]
      (bound-tree P (Fin a) ∪
        ⋃(set [range-tree P a (Fin Bmax)])) 0 []
    by (rule exact-concrete-partition-loop-state-done-witness
      [OF Done-Case(1) Done-Case(2) Done-Case(3)])
  then show ?thesis
    using Done-Case(4) by blast
next
  case Stop-Case
  have run:
    exact-concrete-partition-loop-state Δ M-of t h k cap l d P
      (Fin Bmax) d' D a [] [] (Fin a)
      [range-tree P a (Fin a)]
      (bound-tree P (Fin a) ∪
        ⋃(set [range-tree P a (Fin a)])) 0 []
    by (rule exact-concrete-partition-loop-state-stop-witness
      [OF Stop-Case(1) Stop-Case(2) Stop-Case(3)])
  then show ?thesis
    using Stop-Case(4) by blast
next
  case (Step-Case S-pull beta D-pull B')
  then obtain d-child b c-child betas bs Us-tail U-tail c-tail
      child-costs-tail where
    a-le-b: a ≤ b
    and a-bound: bound-le (Fin a) B'
    and child:
      exact-concrete-bmssp Δ M-of t h k cap l d S-pull (Fin beta)
        d-child (Fin b) (range-tree P a (Fin b)) c-child
    and preserved:
      complete-preserved d-child d' (range-tree P a (Fin b))
    and tail:
      exact-concrete-partition-loop-state Δ M-of t h k cap l d P
        (Fin Bmax) d'
        (partition-state-batch-prepend (label-pairs-between d S-pull b beta)
          (partition-state-batch-prepend
            (edge-relaxation-pairs-between d-child
              (range-tree P a (Fin b)) b beta)
            (partition-state-batch-prepend
              (edge-relaxation-pairs-in-bound d-child
                (range-tree P a (Fin b)) beta (Fin Bmax))
              D-pull)))
        b betas bs B' Us-tail U-tail c-tail child-costs-tail
```

```
    by blast
  have loop:
    ∃ c child-costs.
      exact-concrete-partition-loop-state Δ M-of t h k cap l d P
        (Fin Bmax) d' D a (beta # betas) (b # bs) B'
        (range-tree P a (Fin b) # Us-tail)
        (bound-tree P (Fin a) ∪
          ⋃(set (range-tree P a (Fin b) # Us-tail))) c child-costs
    by (rule exact-concrete-initialized-partition-loop-state-step-exists-finite-bound
      [OF init pre upper Step-Case(1) P-reaches a-le-b a-bound
        Step-Case(2) Step-Case(3) child preserved tail])
  then show ?thesis
    using Step-Case(4) by blast
 qed
qed

lemma exact-concrete-initialized-partition-loop-state-cases-exists-with-complete:
 fixes W :: bound ⇒ 'a set
 assumes init: exact-partition-initial-state t d P D c-insert
   and pre: bmssp-pre-full d P (Fin Bmax)
   and upper: ⋀u. u ∈ P ⟹ d u < Bmax
   and P-reaches: ∀ x∈P. reachable s x
   and cases:
    (keys-of (partition-state-view D) = {} ∧
      bound-le (Fin a) B ∧
      complete-on d' (bound-tree P B) ∧
      complete-on d' (W B)) ∨
     (bound-le (Fin a) B ∧
      complete-on d' (bound-tree P (Fin a)) ∧
      k * cap ≤ card (bound-tree P (Fin a)) ∧
      complete-on d' (W (Fin a))) ∨
     (∃ S-pull beta D-pull B'.
      partition-state-pull (M-of l) Bmax D = (S-pull, beta, D-pull) ∧
      bound-le (Fin beta) B ∧
      complete-on d' (bound-tree P (Fin a)) ∧
      card (bound-tree P (Fin a)) < k * cap ∧
      complete-on d' (W B') ∧
      (∃ d-child b c-child betas bs Us-tail U-tail c-tail
          child-costs-tail.
        a ≤ b ∧
        bound-le (Fin a) B' ∧
        exact-concrete-bmssp Δ M-of t h k cap l d S-pull (Fin beta)
          d-child (Fin b) (range-tree P a (Fin b)) c-child ∧
        complete-preserved d-child d' (range-tree P a (Fin b)) ∧
        exact-concrete-partition-loop-state Δ M-of t h k cap l d P
          B d'
          (partition-state-batch-prepend
            (label-pairs-between d S-pull b beta)
            (partition-state-batch-prepend
```

```
            (edge-relaxation-pairs-between d-child
              (range-tree P a (Fin b)) b beta)
            (partition-state-batch-prepend
              (edge-relaxation-pairs-in-bound d-child
                (range-tree P a (Fin b)) beta B)
              D-pull)))
        b betas bs B' Us-tail U-tail c-tail child-costs-tail))
  shows ∃ betas bs B' Us U c child-costs.
    exact-concrete-partition-loop-state Δ M-of t h k cap l d P
      B d' D a betas bs B' Us U c child-costs ∧
    complete-on d' (W B')
proof −
  consider
    (Done-Case) keys-of (partition-state-view D) = {}
      bound-le (Fin a) B
      complete-on d' (bound-tree P B)
      complete-on d' (W B)
  | (Stop-Case) bound-le (Fin a) B
      complete-on d' (bound-tree P (Fin a))
      k * cap ≤ card (bound-tree P (Fin a))
      complete-on d' (W (Fin a))
  | (Step-Case) S-pull beta D-pull B' where
      partition-state-pull (M-of l) Bmax D = (S-pull, beta, D-pull)
      bound-le (Fin beta) B
      complete-on d' (bound-tree P (Fin a))
      card (bound-tree P (Fin a)) < k * cap
      complete-on d' (W B')
      ∃ d-child b c-child betas bs Us-tail U-tail c-tail
          child-costs-tail.
        a ≤ b ∧
        bound-le (Fin a) B' ∧
        exact-concrete-bmssp Δ M-of t h k cap l d S-pull (Fin beta)
          d-child (Fin b) (range-tree P a (Fin b)) c-child ∧
        complete-preserved d-child d' (range-tree P a (Fin b)) ∧
        exact-concrete-partition-loop-state Δ M-of t h k cap l d P
          B d'
          (partition-state-batch-prepend (label-pairs-between d S-pull b beta)
            (partition-state-batch-prepend
              (edge-relaxation-pairs-between d-child
                (range-tree P a (Fin b)) b beta)
              (partition-state-batch-prepend
                (edge-relaxation-pairs-in-bound d-child
                  (range-tree P a (Fin b)) beta B)
                D-pull)))
          b betas bs B' Us-tail U-tail c-tail child-costs-tail
    using cases by blast
  then show ?thesis
  proof cases
    case Done-Case
```

```
  have run:
    exact-concrete-partition-loop-state Δ M-of t h k cap l d P
      B d' D a [] [] B [range-tree P a B]
      (bound-tree P (Fin a) ∪
        ⋃(set [range-tree P a B])) 0 []
    by (rule exact-concrete-partition-loop-state-done-witness
      [OF Done-Case(1) Done-Case(2) Done-Case(3)])
  then show ?thesis
    using Done-Case(4) by blast
next
  case Stop-Case
  have run:
    exact-concrete-partition-loop-state Δ M-of t h k cap l d P
      B d' D a [] [] (Fin a)
      [range-tree P a (Fin a)]
      (bound-tree P (Fin a) ∪
        ⋃(set [range-tree P a (Fin a)])) 0 []
    by (rule exact-concrete-partition-loop-state-stop-witness
      [OF Stop-Case(1) Stop-Case(2) Stop-Case(3)])
  then show ?thesis
    using Stop-Case(4) by blast
next
  case (Step-Case S-pull beta D-pull B')
  then obtain d-child b c-child betas bs Us-tail U-tail c-tail
      child-costs-tail where
    a-le-b: a ≤ b
    and a-bound: bound-le (Fin a) B'
    and child:
      exact-concrete-bmssp Δ M-of t h k cap l d S-pull (Fin beta)
        d-child (Fin b) (range-tree P a (Fin b)) c-child
    and preserved:
      complete-preserved d-child d' (range-tree P a (Fin b))
    and tail:
      exact-concrete-partition-loop-state Δ M-of t h k cap l d P
        B d'
        (partition-state-batch-prepend (label-pairs-between d S-pull b beta)
          (partition-state-batch-prepend
            (edge-relaxation-pairs-between d-child
              (range-tree P a (Fin b)) b beta)
            (partition-state-batch-prepend
              (edge-relaxation-pairs-in-bound d-child
                (range-tree P a (Fin b)) beta B)
              D-pull)))
        b betas bs B' Us-tail U-tail c-tail child-costs-tail
    by blast
  have S-pull-reaches: ∀ x∈S-pull. reachable s x
    by (rule exact-partition-initial-state-pull-reaches
      [OF init upper Step-Case(1) P-reaches])
  have loop:
```

```
    ∃ c child-costs.
      exact-concrete-partition-loop-state Δ M-of t h k cap l d P
        B d′ D a (beta # betas) (b # bs) B′
        (range-tree P a (Fin b) # Us-tail)
        (bound-tree P (Fin a) ∪
          ⋃(set (range-tree P a (Fin b) # Us-tail))) c child-costs
    by (rule exact-concrete-initialized-partition-loop-state-step-exists
      [OF init pre upper Step-Case(1) Step-Case(2) S-pull-reaches
        a-le-b a-bound Step-Case(3) Step-Case(4) child refl
        preserved tail])
  then show ?thesis
    using Step-Case(5) by blast
 qed
qed

lemma exact-concrete-partition-loop-state-step-exists-from-concrete-ops:
 assumes inv-D: partition-state-invar D
   and upper-D: partition-upper-bound (partition-state-view D) Bmax
   and pull:
     partition-state-pull (M-of l) Bmax D = (S-pull, beta, D-pull)
   and beta-bound: bound-le (Fin beta) B
   and child-pre: bmssp-pre-full d S-pull (Fin beta)
   and S-pull-def: S-pull = split-below d P beta
   and S-pull-reaches: ∀ x∈S-pull. reachable s x
   and a-le-b: a ≤ b
   and a-bound: bound-le (Fin a) B′
   and complete-prefix: complete-on d′ (bound-tree P (Fin a))
   and below-threshold: card (bound-tree P (Fin a)) < k * cap
   and child:
     exact-concrete-bmssp Δ M-of t h k cap l d S-pull (Fin beta)
       d-child (Fin b) U-child c-child
   and U-child: U-child = range-tree P a (Fin b)
   and preserved: complete-preserved d-child d′ U-child
   and tail:
     exact-concrete-partition-loop-state Δ M-of t h k cap l d P B d′
       (partition-state-batch-prepend (label-pairs-between d S-pull b beta)
         (partition-state-batch-prepend
           (edge-relaxation-pairs-between d-child U-child b beta)
           (partition-state-batch-prepend
             (edge-relaxation-pairs-in-bound d-child U-child beta B)
             D-pull)))
       b betas bs B′ Us-tail U-tail c-tail child-costs-tail
 shows ∃ c child-costs.
   exact-concrete-partition-loop-state Δ M-of t h k cap l d P B d′ D a
     (beta # betas) (b # bs) B′
     (range-tree P a (Fin b) # Us-tail)
     (bound-tree P (Fin a) ∪
       ⋃(set (range-tree P a (Fin b) # Us-tail))) c child-costs
proof −
```

```
  have run:
    exact-concrete-partition-loop-state Δ M-of t h k cap l d P B d' D a
      (beta # betas) (b # bs) B'
      (range-tree P a (Fin b) # Us-tail)
      (bound-tree P (Fin a) ∪
        ⋃(set (range-tree P a (Fin b) # Us-tail)))
      (card S-pull +
        t * length (edge-relaxation-pairs-in-bound d-child U-child beta B) +
        h * length (edge-relaxation-pairs-between d-child U-child b beta) +
        h * length (label-pairs-between d S-pull b beta) +
        c-child + c-tail)
      (c-child # child-costs-tail)
    by (rule exact-concrete-partition-loop-state-step-from-concrete-ops
      [OF inv-D upper-D pull beta-bound child-pre S-pull-def
        S-pull-reaches a-le-b a-bound complete-prefix below-threshold child
        U-child preserved tail])
  show ?thesis
    using run by blast
qed

lemma exact-concrete-bmssp-step-with-exact-insert-cost:
  assumes init: exact-partition-initial-state t
      (find-pivots-label-capped k cap d S B)
      (find-pivots-pivots-capped k cap d S B) D c-insert
    and loop:
      exact-concrete-partition-loop-state Δ M-of t h k cap l
        (find-pivots-label-capped k cap d S B)
        (find-pivots-pivots-capped k cap d S B) B d' D a
        betas bs B' Us U-loop c-loop child-costs-loop
    and complete:
      complete-on d'
        {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v =
          dist s v}
    and U:
      U = U-loop ∪
        {v ∈ bound-tree S B'. find-pivots-label-capped k cap d S B v =
          dist s v}
  shows exact-concrete-bmssp Δ M-of t h k cap (Suc l) d S B d' B' U
    (fp-iter-capped-scan-cost k cap d S S B +
      t * card (find-pivots-pivots-capped k cap d S B) + c-loop)
proof −
  have c-insert:
    c-insert = t * card (find-pivots-pivots-capped k cap d S B)
    by (rule exact-partition-initial-state-cost[OF init])
  show ?thesis
    by (rule Exact-Concrete-Step[OF init loop complete U])
      (simp add: c-insert)
qed
```

```
lemma exact-concrete-bmssp-Suc-exists-from-initial-loop:
  assumes S-subset: S ⊆ V
    and loop:
      ⋀D. exact-partition-initial-state t
        (find-pivots-label-capped k cap d S B)
        (find-pivots-pivots-capped k cap d S B) D
        (t * card (find-pivots-pivots-capped k cap d S B)) ⟹
        ∃ d' a betas bs B' Us U-loop c-loop child-costs-loop.
          exact-concrete-partition-loop-state Δ M-of t h k cap l
            (find-pivots-label-capped k cap d S B)
            (find-pivots-pivots-capped k cap d S B) B d' D a
            betas bs B' Us U-loop c-loop child-costs-loop ∧
          complete-on d'
            {v ∈ bound-tree S B'.
              find-pivots-label-capped k cap d S B v = dist s v}
  shows ∃ d' B' U c.
    exact-concrete-bmssp Δ M-of t h k cap (Suc l) d S B d' B' U c
proof −
  let ?d-fp = find-pivots-label-capped k cap d S B
  let ?P = find-pivots-pivots-capped k cap d S B
  obtain D where init:
    exact-partition-initial-state t ?d-fp ?P D (t * card ?P)
    using exact-partition-initial-state-exists-for-capped-pivots-with-cost
      [OF S-subset] by blast
  obtain d' a betas bs B' Us U-loop c-loop child-costs-loop where
    loop-run:
      exact-concrete-partition-loop-state Δ M-of t h k cap l
        ?d-fp ?P B d' D a betas bs B' Us U-loop c-loop child-costs-loop
    and complete:
      complete-on d'
        {v ∈ bound-tree S B'. ?d-fp v = dist s v}
    using loop[OF init] by blast
  let ?U = U-loop ∪ {v ∈ bound-tree S B'. ?d-fp v = dist s v}
  have run:
    exact-concrete-bmssp Δ M-of t h k cap (Suc l) d S B d' B' ?U
      (fp-iter-capped-scan-cost k cap d S S B + t * card ?P + c-loop)
    by (rule exact-concrete-bmssp-step-with-exact-insert-cost
      [OF init loop-run complete refl])
  show ?thesis
    using run by blast
qed
```

The non-base existence lemmas are phrased as case combinators. They do not decide which branch an implementation should take; instead they say that if a client can provide one of the expected witnesses (empty partition, threshold, or a recursive child plus tail loop), then a full exact concrete BMSSP step exists. This keeps totality concerns separate from the correctness and cost refinement already proved for any exact run.

```
lemma exact-concrete-bmssp-Suc-exists-from-initial-loop-cases:
```

fixes $d :: 'a \Rightarrow real$
and $S :: 'a\ set$
and $B :: bound$
and $Bmax :: real$
and $k\ cap :: nat$
and $d\text{-}fp :: 'a \Rightarrow real$
and $P :: 'a\ set$
defines *d-fp-def*:
$d\text{-}fp \equiv find\text{-}pivots\text{-}label\text{-}capped\ k\ cap\ d\ S\ B$
and *P-def*:
$P \equiv find\text{-}pivots\text{-}pivots\text{-}capped\ k\ cap\ d\ S\ B$
assumes *S-subset*: $S \subseteq V$
and *pre-fp*: $bmssp\text{-}pre\text{-}full\ d\text{-}fp\ P\ (Fin\ Bmax)$
and *upper*: $\bigwedge x.\ x \in P \Longrightarrow d\text{-}fp\ x < Bmax$
and *P-reaches*: $\forall x{\in}P.\ reachable\ s\ x$
and *cases*:
$\bigwedge D.\ exact\text{-}partition\text{-}initial\text{-}state\ t\ d\text{-}fp\ P\ D\ (t * card\ P) \Longrightarrow$
$\exists d'\ a.$
$(keys\text{-}of\ (partition\text{-}state\text{-}view\ D) = \{\} \wedge$
$bound\text{-}le\ (Fin\ a)\ B \wedge$
$complete\text{-}on\ d'\ (bound\text{-}tree\ P\ B) \wedge$
$complete\text{-}on\ d'$
$\{v \in bound\text{-}tree\ S\ B.\ d\text{-}fp\ v = dist\ s\ v\}) \vee$
$(bound\text{-}le\ (Fin\ a)\ B \wedge$
$complete\text{-}on\ d'\ (bound\text{-}tree\ P\ (Fin\ a)) \wedge$
$k * cap \leq card\ (bound\text{-}tree\ P\ (Fin\ a)) \wedge$
$complete\text{-}on\ d'$
$\{v \in bound\text{-}tree\ S\ (Fin\ a).\ d\text{-}fp\ v = dist\ s\ v\}) \vee$
$(\exists S\text{-}pull\ beta\ D\text{-}pull\ B'.$
$partition\text{-}state\text{-}pull\ (M\text{-}of\ l)\ Bmax\ D = (S\text{-}pull,\ beta,\ D\text{-}pull) \wedge$
$bound\text{-}le\ (Fin\ beta)\ B \wedge$
$complete\text{-}on\ d'\ (bound\text{-}tree\ P\ (Fin\ a)) \wedge$
$card\ (bound\text{-}tree\ P\ (Fin\ a)) < k * cap \wedge$
$complete\text{-}on\ d'$
$\{v \in bound\text{-}tree\ S\ B'.\ d\text{-}fp\ v = dist\ s\ v\} \wedge$
$(\exists d\text{-}child\ b\ c\text{-}child\ betas\ bs\ Us\text{-}tail\ U\text{-}tail\ c\text{-}tail$
$child\text{-}costs\text{-}tail.$
$a \leq b \wedge$
$bound\text{-}le\ (Fin\ a)\ B' \wedge$
$exact\text{-}concrete\text{-}bmssp\ \Delta\ M\text{-}of\ t\ h\ k\ cap\ l\ d\text{-}fp\ S\text{-}pull$
$(Fin\ beta)\ d\text{-}child\ (Fin\ b)\ (range\text{-}tree\ P\ a\ (Fin\ b))$
$c\text{-}child \wedge$
$complete\text{-}preserved\ d\text{-}child\ d'\ (range\text{-}tree\ P\ a\ (Fin\ b)) \wedge$
$exact\text{-}concrete\text{-}partition\text{-}loop\text{-}state\ \Delta\ M\text{-}of\ t\ h\ k\ cap\ l$
$d\text{-}fp\ P\ B\ d'$
$(partition\text{-}state\text{-}batch\text{-}prepend$
$(label\text{-}pairs\text{-}between\ d\text{-}fp\ S\text{-}pull\ b\ beta)$
$(partition\text{-}state\text{-}batch\text{-}prepend$
$(edge\text{-}relaxation\text{-}pairs\text{-}between\ d\text{-}child$

```
                (range-tree P a (Fin b)) b beta)
              (partition-state-batch-prepend
                (edge-relaxation-pairs-in-bound d-child
                  (range-tree P a (Fin b)) beta B)
                D-pull)))
          b betas bs B' Us-tail U-tail c-tail child-costs-tail))
  shows ∃ d' B' U c.
    exact-concrete-bmssp Δ M-of t h k cap (Suc l) d S B d' B' U c
proof −
  obtain D where init:
    exact-partition-initial-state t d-fp P D (t * card P)
    using exact-partition-initial-state-exists-for-capped-pivots-with-cost
      [OF S-subset]
    unfolding d-fp-def P-def by blast
  obtain d' a where case-data:
    (keys-of (partition-state-view D) = {} ∧
      bound-le (Fin a) B ∧
      complete-on d' (bound-tree P B) ∧
      complete-on d'
        {v ∈ bound-tree S B. d-fp v = dist s v}) ∨
     (bound-le (Fin a) B ∧
      complete-on d' (bound-tree P (Fin a)) ∧
      k * cap ≤ card (bound-tree P (Fin a)) ∧
      complete-on d'
        {v ∈ bound-tree S (Fin a). d-fp v = dist s v}) ∨
     (∃ S-pull beta D-pull B'.
      partition-state-pull (M-of l) Bmax D = (S-pull, beta, D-pull) ∧
      bound-le (Fin beta) B ∧
      complete-on d' (bound-tree P (Fin a)) ∧
      card (bound-tree P (Fin a)) < k * cap ∧
      complete-on d'
        {v ∈ bound-tree S B'. d-fp v = dist s v} ∧
      (∃ d-child b c-child betas bs Us-tail U-tail c-tail
          child-costs-tail.
        a ≤ b ∧
        bound-le (Fin a) B' ∧
        exact-concrete-bmssp Δ M-of t h k cap l d-fp S-pull
          (Fin beta) d-child (Fin b) (range-tree P a (Fin b))
          c-child ∧
        complete-preserved d-child d' (range-tree P a (Fin b)) ∧
        exact-concrete-partition-loop-state Δ M-of t h k cap l
          d-fp P B d'
          (partition-state-batch-prepend
            (label-pairs-between d-fp S-pull b beta)
            (partition-state-batch-prepend
              (edge-relaxation-pairs-between d-child
                (range-tree P a (Fin b)) b beta)
              (partition-state-batch-prepend
                (edge-relaxation-pairs-in-bound d-child
```

```
            (range-tree P a (Fin b)) beta B)
          D-pull)))
      b betas bs B' Us-tail U-tail c-tail child-costs-tail))
  using cases[OF init] by blast
obtain betas bs B' Us U-loop c-loop child-costs-loop where
  loop:
    exact-concrete-partition-loop-state Δ M-of t h k cap l d-fp P
      B d' D a betas bs B' Us U-loop c-loop child-costs-loop
  and complete:
    complete-on d'
      {v ∈ bound-tree S B'. d-fp v = dist s v}
  using exact-concrete-initialized-partition-loop-state-cases-exists-with-complete
    [where W = λB'. {v ∈ bound-tree S B'. d-fp v = dist s v},
      OF init pre-fp upper P-reaches case-data]
  by blast
have init-actual:
  exact-partition-initial-state t
    (find-pivots-label-capped k cap d S B)
    (find-pivots-pivots-capped k cap d S B) D
    (t * card (find-pivots-pivots-capped k cap d S B))
  using init unfolding d-fp-def P-def .
have loop-actual:
  exact-concrete-partition-loop-state Δ M-of t h k cap l
    (find-pivots-label-capped k cap d S B)
    (find-pivots-pivots-capped k cap d S B)
    B d' D a betas bs B' Us U-loop c-loop child-costs-loop
  using loop unfolding d-fp-def P-def .
have complete-actual:
  complete-on d'
    {v ∈ bound-tree S B'.
      find-pivots-label-capped k cap d S B v = dist s v}
  using complete unfolding d-fp-def .
let ?U =
  U-loop ∪
    {v ∈ bound-tree S B'.
      find-pivots-label-capped k cap d S B v = dist s v}
have run:
  exact-concrete-bmssp Δ M-of t h k cap (Suc l) d S B d' B' ?U
    (fp-iter-capped-scan-cost k cap d S S B +
      t * card (find-pivots-pivots-capped k cap d S B) + c-loop)
  by (rule exact-concrete-bmssp-step-with-exact-insert-cost
    [OF init-actual loop-actual complete-actual refl])
then show ?thesis
  by blast
qed

lemma exact-concrete-bmssp-Suc-exists-from-finite-initial-loop-cases:
fixes d :: 'a ⇒ real
  and S :: 'a set
```

and *Bmax* :: *real*
and *k cap* :: *nat*
and *d-fp* :: *'a* ⇒ *real*
and *P* :: *'a set*
defines *d-fp-def*:
d-fp ≡ *find-pivots-label-capped k cap d S* (*Fin Bmax*)
and *P-def*:
P ≡ *find-pivots-pivots-capped k cap d S* (*Fin Bmax*)
assumes *sound*: *sound-label d*
and *pre*: *bmssp-pre-full d S* (*Fin Bmax*)
and *S-reaches*: ∀ *x*∈*S*. *reachable s x*
and *below*: ⋀*x*. *x* ∈ *S* ⟹ *d x* < *Bmax*
and *cases*:
⋀*D*. *exact-partition-initial-state t d-fp P D* (*t* ∗ *card P*) ⟹
∃ *d'* *a*.
(*keys-of* (*partition-state-view D*) = {} ∧
bound-le (*Fin a*) (*Fin Bmax*) ∧
complete-on d' (*bound-tree P* (*Fin Bmax*)) ∧
complete-on d'
{*v* ∈ *bound-tree S* (*Fin Bmax*). *d-fp v* = *dist s v*}) ∨
(*bound-le* (*Fin a*) (*Fin Bmax*) ∧
complete-on d' (*bound-tree P* (*Fin a*)) ∧
k ∗ *cap* ≤ *card* (*bound-tree P* (*Fin a*)) ∧
complete-on d'
{*v* ∈ *bound-tree S* (*Fin a*). *d-fp v* = *dist s v*}) ∨
(∃ *S-pull beta D-pull B'*.
partition-state-pull (*M-of l*) *Bmax D* = (*S-pull*, *beta*, *D-pull*) ∧
complete-on d' (*bound-tree P* (*Fin a*)) ∧
card (*bound-tree P* (*Fin a*)) < *k* ∗ *cap* ∧
complete-on d'
{*v* ∈ *bound-tree S B'*. *d-fp v* = *dist s v*} ∧
(∃ *d-child b c-child betas bs Us-tail U-tail c-tail*
child-costs-tail.
a ≤ *b* ∧
bound-le (*Fin a*) *B'* ∧
exact-concrete-bmssp Δ *M-of t h k cap l d-fp S-pull*
(*Fin beta*) *d-child* (*Fin b*) (*range-tree P a* (*Fin b*))
c-child ∧
complete-preserved d-child d' (*range-tree P a* (*Fin b*)) ∧
exact-concrete-partition-loop-state Δ *M-of t h k cap l*
d-fp P (*Fin Bmax*) *d'*
(*partition-state-batch-prepend*
(*label-pairs-between d-fp S-pull b beta*)
(*partition-state-batch-prepend*
(*edge-relaxation-pairs-between d-child*
(*range-tree P a* (*Fin b*)) *b beta*)
(*partition-state-batch-prepend*
(*edge-relaxation-pairs-in-bound d-child*
(*range-tree P a* (*Fin b*)) *beta* (*Fin Bmax*))

```
                D-pull)))
            b betas bs B' Us-tail U-tail c-tail child-costs-tail))
  shows ∃ d' B' U c.
    exact-concrete-bmssp Δ M-of t h k cap (Suc l) d S (Fin Bmax)
      d' B' U c
proof −
  have S-subset: S ⊆ V
    using pre unfolding bmssp-pre-full-def by blast
  obtain D where init:
    exact-partition-initial-state t d-fp P D (t ∗ card P)
    using exact-partition-initial-state-exists-for-capped-pivots-with-cost
      [OF S-subset]
    unfolding d-fp-def P-def by blast
  have pre-fp: bmssp-pre-full d-fp P (Fin Bmax)
    unfolding d-fp-def P-def
    by (rule find-pivots-capped-establishes-pivot-pre-concrete
      [OF sound pre]) (use S-reaches in blast)
  have upper: ⋀u. u ∈ P ⟹ d-fp u < Bmax
    unfolding d-fp-def P-def
    by (rule find-pivots-pivots-capped-label-less-finite[OF below])
  have P-reaches: ∀ x∈P. reachable s x
    unfolding P-def
    by (rule find-pivots-pivots-capped-reaches[OF S-reaches])
  obtain d' a where case-data:
    (keys-of (partition-state-view D) = {} ∧
      bound-le (Fin a) (Fin Bmax) ∧
      complete-on d' (bound-tree P (Fin Bmax)) ∧
      complete-on d'
        {v ∈ bound-tree S (Fin Bmax). d-fp v = dist s v}) ∨
     (bound-le (Fin a) (Fin Bmax) ∧
      complete-on d' (bound-tree P (Fin a)) ∧
      k ∗ cap ≤ card (bound-tree P (Fin a)) ∧
      complete-on d'
        {v ∈ bound-tree S (Fin a). d-fp v = dist s v}) ∨
     (∃ S-pull beta D-pull B'.
      partition-state-pull (M-of l) Bmax D = (S-pull, beta, D-pull) ∧
      complete-on d' (bound-tree P (Fin a)) ∧
      card (bound-tree P (Fin a)) < k ∗ cap ∧
      complete-on d'
        {v ∈ bound-tree S B'. d-fp v = dist s v} ∧
      (∃ d-child b c-child betas bs Us-tail U-tail c-tail
          child-costs-tail.
        a ≤ b ∧
        bound-le (Fin a) B' ∧
        exact-concrete-bmssp Δ M-of t h k cap l d-fp S-pull
          (Fin beta) d-child (Fin b) (range-tree P a (Fin b))
          c-child ∧
        complete-preserved d-child d' (range-tree P a (Fin b)) ∧
        exact-concrete-partition-loop-state Δ M-of t h k cap l
```

```
      d-fp P (Fin Bmax) d'
      (partition-state-batch-prepend
        (label-pairs-between d-fp S-pull b beta)
        (partition-state-batch-prepend
          (edge-relaxation-pairs-between d-child
            (range-tree P a (Fin b)) b beta)
          (partition-state-batch-prepend
            (edge-relaxation-pairs-in-bound d-child
              (range-tree P a (Fin b)) beta (Fin Bmax))
            D-pull)))
      b betas bs B' Us-tail U-tail c-tail child-costs-tail))
  using cases[OF init] by blast
obtain betas bs B' Us U-loop c-loop child-costs-loop where
  loop:
    exact-concrete-partition-loop-state Δ M-of t h k cap l d-fp P
      (Fin Bmax) d' D a betas bs B' Us U-loop c-loop
      child-costs-loop
  and complete:
    complete-on d'
      {v ∈ bound-tree S B'. d-fp v = dist s v}
  using exact-concrete-initialized-partition-loop-state-cases-exists-finite-bound-with-
complete
    [where W = λB'. {v ∈ bound-tree S B'. d-fp v = dist s v},
      OF init pre-fp upper P-reaches case-data]
  by blast
have init-actual:
  exact-partition-initial-state t
    (find-pivots-label-capped k cap d S (Fin Bmax))
    (find-pivots-pivots-capped k cap d S (Fin Bmax)) D
    (t * card (find-pivots-pivots-capped k cap d S (Fin Bmax)))
  using init unfolding d-fp-def P-def .
have loop-actual:
  exact-concrete-partition-loop-state Δ M-of t h k cap l
    (find-pivots-label-capped k cap d S (Fin Bmax))
    (find-pivots-pivots-capped k cap d S (Fin Bmax))
    (Fin Bmax) d' D a betas bs B' Us U-loop c-loop
    child-costs-loop
  using loop unfolding d-fp-def P-def .
have complete-actual:
  complete-on d'
    {v ∈ bound-tree S B'.
      find-pivots-label-capped k cap d S (Fin Bmax) v = dist s v}
  using complete unfolding d-fp-def .
let ?U =
  U-loop ∪
    {v ∈ bound-tree S B'.
      find-pivots-label-capped k cap d S (Fin Bmax) v = dist s v}
have run:
  exact-concrete-bmssp Δ M-of t h k cap (Suc l) d S (Fin Bmax)
```

```
    d' B' ?U
    (fp-iter-capped-scan-cost k cap d S S (Fin Bmax) +
      t * card (find-pivots-pivots-capped k cap d S (Fin Bmax)) +
      c-loop)
  by (rule exact-concrete-bmssp-step-with-exact-insert-cost
    [OF init-actual loop-actual complete-actual refl])
 then show ?thesis
  by blast
qed

lemma exact-concrete-bmssp-Suc-exists-if-pivots-empty:
 assumes S-subset: S ⊆ V
  and a-bound: bound-le (Fin a) B
  and pivots-empty:
    find-pivots-pivots-capped k cap d S B = {}
 shows ∃ d' B' U c.
  exact-concrete-bmssp Δ M-of t h k cap (Suc l) d S B d' B' U c
proof (rule exact-concrete-bmssp-Suc-exists-from-initial-loop[OF S-subset])
 let ?d-fp = find-pivots-label-capped k cap d S B
 let ?P = find-pivots-pivots-capped k cap d S B
 fix D
 assume init: exact-partition-initial-state t ?d-fp ?P D (t * card ?P)
 have view: partition-state-view D = label-partition-view ?d-fp ?P
  by (rule exact-partition-initial-state-view[OF init])
 have empty: keys-of (partition-state-view D) = {}
  using view pivots-empty unfolding label-partition-view-def by simp
 have loop:
  exact-concrete-partition-loop-state Δ M-of t h k cap l
    ?d-fp ?P B (dist s) D a [] [] B [range-tree ?P a B]
    (bound-tree ?P (Fin a) ∪ ⋃(set [range-tree ?P a B])) 0 []
  by (rule exact-concrete-partition-loop-state-done-witness
    [OF empty a-bound complete-on-dist])
 have complete:
  complete-on (dist s)
    {v ∈ bound-tree S B. ?d-fp v = dist s v}
  by (rule complete-on-dist)
 show ∃ d' a betas bs B' Us U-loop c-loop child-costs-loop.
    exact-concrete-partition-loop-state Δ M-of t h k cap l
      ?d-fp ?P B d' D a betas bs B' Us U-loop c-loop
      child-costs-loop ∧
    complete-on d'
      {v ∈ bound-tree S B'. ?d-fp v = dist s v}
  using loop complete by blast
qed

lemma exact-concrete-bmssp-Suc-exists-if-initial-threshold:
 assumes S-subset: S ⊆ V
  and a-bound: bound-le (Fin a) B
  and threshold:
```

```
    k * cap ≤
      card (bound-tree (find-pivots-pivots-capped k cap d S B) (Fin a))
  shows ∃ d' B' U c.
    exact-concrete-bmssp Δ M-of t h k cap (Suc l) d S B d' B' U c
proof (rule exact-concrete-bmssp-Suc-exists-from-initial-loop[OF S-subset])
  let ?d-fp = find-pivots-label-capped k cap d S B
  let ?P = find-pivots-pivots-capped k cap d S B
  fix D
  assume init: exact-partition-initial-state t ?d-fp ?P D (t * card ?P)
  have loop:
    exact-concrete-partition-loop-state Δ M-of t h k cap l
      ?d-fp ?P B (dist s) D a [] [] (Fin a)
      [range-tree ?P a (Fin a)]
      (bound-tree ?P (Fin a) ∪ ⋃(set [range-tree ?P a (Fin a)]))
      0 []
    by (rule exact-concrete-partition-loop-state-stop-witness
      [OF a-bound complete-on-dist threshold])
  have complete:
    complete-on (dist s)
      {v ∈ bound-tree S (Fin a). ?d-fp v = dist s v}
    by (rule complete-on-dist)
  show ∃ d' a betas bs B' Us U-loop c-loop child-costs-loop.
      exact-concrete-partition-loop-state Δ M-of t h k cap l
        ?d-fp ?P B d' D a betas bs B' Us U-loop c-loop
        child-costs-loop ∧
      complete-on d'
        {v ∈ bound-tree S B'. ?d-fp v = dist s v}
    using loop complete by blast
qed

lemma exact-concrete-bmssp-Suc-exists-if-pivots-empty-same-bound:
  assumes S-subset: S ⊆ V
    and a-bound: bound-le (Fin a) B
    and pivots-empty:
      find-pivots-pivots-capped k cap d S B = {}
  shows ∃ d' U c.
    exact-concrete-bmssp Δ M-of t h k cap (Suc l) d S B d' B U c
proof −
  let ?d-fp = find-pivots-label-capped k cap d S B
  let ?P = find-pivots-pivots-capped k cap d S B
  let ?U-loop =
    bound-tree ?P (Fin a) ∪ ⋃(set [range-tree ?P a B])
  obtain D where init:
    exact-partition-initial-state t ?d-fp ?P D (t * card ?P)
    using exact-partition-initial-state-exists-for-capped-pivots-with-cost
      [OF S-subset] by blast
  have view: partition-state-view D = label-partition-view ?d-fp ?P
    by (rule exact-partition-initial-state-view[OF init])
  have empty: keys-of (partition-state-view D) = {}
```

```
    using view pivots-empty unfolding label-partition-view-def by simp
  have loop:
    exact-concrete-partition-loop-state Δ M-of t h k cap l
      ?d-fp ?P B (dist s) D a [] [] B [range-tree ?P a B] ?U-loop
      0 []
    by (rule exact-concrete-partition-loop-state-done-witness
      [OF empty a-bound complete-on-dist])
  have complete:
    complete-on (dist s)
      {v ∈ bound-tree S B. ?d-fp v = dist s v}
    by (rule complete-on-dist)
  let ?U = ?U-loop ∪ {v ∈ bound-tree S B. ?d-fp v = dist s v}
  have run:
    exact-concrete-bmssp Δ M-of t h k cap (Suc l) d S B
      (dist s) B ?U
      (fp-iter-capped-scan-cost k cap d S S B + t * card ?P + 0)
    by (rule exact-concrete-bmssp-step-with-exact-insert-cost
      [OF init loop complete refl])
  show ?thesis
    using run by blast
qed

lemma exact-concrete-bmssp-Suc-exists-if-initial-threshold-bound:
  assumes S-subset: S ⊆ V
    and a-bound: bound-le (Fin a) B
    and threshold:
      k * cap ≤
        card (bound-tree (find-pivots-pivots-capped k cap d S B) (Fin a))
  shows ∃ d' U c.
    exact-concrete-bmssp Δ M-of t h k cap (Suc l) d S B
      d' (Fin a) U c
proof −
  let ?d-fp = find-pivots-label-capped k cap d S B
  let ?P = find-pivots-pivots-capped k cap d S B
  let ?U-loop =
    bound-tree ?P (Fin a) ∪ ⋃(set [range-tree ?P a (Fin a)])
  obtain D where init:
    exact-partition-initial-state t ?d-fp ?P D (t * card ?P)
    using exact-partition-initial-state-exists-for-capped-pivots-with-cost
      [OF S-subset] by blast
  have loop:
    exact-concrete-partition-loop-state Δ M-of t h k cap l
      ?d-fp ?P B (dist s) D a [] [] (Fin a)
      [range-tree ?P a (Fin a)] ?U-loop 0 []
    by (rule exact-concrete-partition-loop-state-stop-witness
      [OF a-bound complete-on-dist threshold])
  have complete:
    complete-on (dist s)
      {v ∈ bound-tree S (Fin a). ?d-fp v = dist s v}
```

```
    by (rule complete-on-dist)
  let ?U =
    ?U-loop ∪ {v ∈ bound-tree S (Fin a). ?d-fp v = dist s v}
  have run:
    exact-concrete-bmssp Δ M-of t h k cap (Suc l) d S B
      (dist s) (Fin a) ?U
      (fp-iter-capped-scan-cost k cap d S S B + t * card ?P + 0)
    by (rule exact-concrete-bmssp-step-with-exact-insert-cost
      [OF init loop complete refl])
  show ?thesis
    using run by blast
qed
```

The central refinement theorem forgets the concrete partition state and keeps only its abstract view. Concrete pulls refine to *pull-separates* and concrete batch prepends refine to *batch-min-update* with the expected cost predicates. After this projection, the exact run is a direct-insert costed run with the same labels, bounds, output sets, and total cost.

The final two theorems are immediate consequences of that projection: exact concrete BMSSP runs inherit the success-or-threshold property and the full BMSSP correctness theorem from the direct-insert layer.

```
theorem exact-concrete-refines-direct-insert:
  exact-concrete-partition-loop-state Δ M-of t h k cap l d P B d' D a
    betas bs B' Us U c child-costs ⟹
   direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B d'
    (partition-state-view D) a betas bs B' Us U c child-costs
and exact-concrete-bmssp-refines-direct-insert:
  exact-concrete-bmssp Δ M-of t h k cap l d S B d' B' U c ⟹
   direct-insert-costed-bmssp Δ M-of t h k cap l d S B d' B' U c
proof (induction rule: exact-concrete-partition-loop-state-exact-concrete-
bmssp.inducts)
  case (Exact-Concrete-State-Done D a B d' P Δ M-of t h k cap l d)
  then show ?case
    by (rule Direct-Insert-Costed-State-Done)
next
  case (Exact-Concrete-State-Stop D a B d' P k cap Δ M-of t h l d)
  then show ?case
    by (rule Direct-Insert-Costed-State-Stop)
next
  case (Exact-Concrete-State-Step D Bmax M-of l S-pull beta D-pull c-pull B
      d P a b B' d' k cap Δ t h d-child U-child c-child
      direct-edge-batch lower-edge-batch source-batch c-direct D-direct c-lower
      D-lower c-sources D-next betas bs Us-tail U-tail c-tail
      child-costs-tail c)
  have inv-D: partition-state-invar D
    using Exact-Concrete-State-Step by blast
  have upper-D: partition-upper-bound (partition-state-view D) Bmax
    using Exact-Concrete-State-Step by blast
  have pull-op:
```

costed-partition-state-pull (*M-of l*) *Bmax D S-pull beta D-pull c-pull*
using *Exact-Concrete-State-Step* **by** *blast*
have *pull*:
pull-separates (*partition-state-view D*) (*M-of l*) *Bmax S-pull beta*
(*partition-state-view D-pull*)
and *pull-cost*: *partition-pull-cost-bound c-pull S-pull*
and *inv-pull*: *partition-state-invar D-pull*
using *costed-partition-state-pull-refines*
[*OF inv-D - pull-op*] *upper-D*
unfolding *partition-upper-bound-def* **by** *blast+*
have *direct-op*:
costed-partition-state-batch-prepend t D-pull direct-edge-batch c-direct D-direct
using *Exact-Concrete-State-Step* **by** *blast*
have *direct-ref*:
partition-state-invar D-direct ∧
partition-state-view D-direct =
batch-min-update (*partition-state-view D-pull*) *direct-edge-batch* ∧
partition-batch-cost-bound c-direct t direct-edge-batch
by (*rule costed-partition-state-batch-prepend-refines*
[*OF inv-pull direct-op*])
have *inv-direct*: *partition-state-invar D-direct*
using *direct-ref* **by** *blast*
have *direct-view*:
partition-state-view D-direct =
batch-min-update (*partition-state-view D-pull*) *direct-edge-batch*
using *direct-ref* **by** *blast*
have *direct-cost*:
partition-batch-cost-bound c-direct t direct-edge-batch
using *direct-ref* **by** *blast*
have *lower-op*:
costed-partition-state-batch-prepend h D-direct lower-edge-batch c-lower D-lower
using *Exact-Concrete-State-Step* **by** *blast*
have *lower-ref*:
partition-state-invar D-lower ∧
partition-state-view D-lower =
batch-min-update (*partition-state-view D-direct*) *lower-edge-batch* ∧
partition-batch-cost-bound c-lower h lower-edge-batch
by (*rule costed-partition-state-batch-prepend-refines*
[*OF inv-direct lower-op*])
have *inv-lower*: *partition-state-invar D-lower*
using *lower-ref* **by** *blast*
have *lower-view*:
partition-state-view D-lower =
batch-min-update (*partition-state-view D-direct*) *lower-edge-batch*
using *lower-ref* **by** *blast*
have *lower-cost*:
partition-batch-cost-bound c-lower h lower-edge-batch
using *lower-ref* **by** *blast*
have *source-op*:

```
    costed-partition-state-batch-prepend h D-lower source-batch c-sources D-next
    using Exact-Concrete-State-Step by blast
  have source-ref:
    partition-state-invar D-next ∧
     partition-state-view D-next =
       batch-min-update (partition-state-view D-lower) source-batch ∧
     partition-batch-cost-bound c-sources h source-batch
    by (rule costed-partition-state-batch-prepend-refines
      [OF inv-lower source-op])
  have source-view:
    partition-state-view D-next =
      batch-min-update (partition-state-view D-lower) source-batch
    using source-ref by blast
  have source-cost:
    partition-batch-cost-bound c-sources h source-batch
    using source-ref by blast
  have D-next-view:
    partition-state-view D-next =
      batch-min-update (partition-state-view D-pull)
        (direct-edge-batch @ lower-edge-batch @ source-batch)
    using direct-view lower-view source-view by simp
  have child-ref:
    direct-insert-costed-bmssp Δ M-of t h k cap l d S-pull (Fin beta)
      d-child (Fin b) U-child c-child
    using Exact-Concrete-State-Step by blast
  have tail-ref:
    direct-insert-costed-partition-loop-state Δ M-of t h k cap l d P B d'
      (partition-state-view D-next) b betas bs B' Us-tail U-tail c-tail
      child-costs-tail
    using Exact-Concrete-State-Step by blast
  show ?case
    by (rule Direct-Insert-Costed-State-Step
      [where batch = direct-edge-batch @ lower-edge-batch @ source-batch
        and D-next = partition-state-view D-next])
      (use Exact-Concrete-State-Step pull child-ref D-next-view pull-cost
        direct-cost lower-cost source-cost tail-ref in auto)
next
  case (Exact-Concrete-Base S x Δ M-of t h k cap d B)
  then show ?case
    by (rule Direct-Insert-Costed-Base)
next
  case (Exact-Concrete-Step t k cap d S B D c-insert Δ M-of h l d' a
      betas bs B' Us U-loop c-loop child-costs-loop U c)
  let ?d-fp = find-pivots-label-capped k cap d S B
  let ?P = find-pivots-pivots-capped k cap d S B
  have init: exact-partition-initial-state t ?d-fp ?P D c-insert
    and loop:
      exact-concrete-partition-loop-state Δ M-of t h k cap l ?d-fp ?P B d' D a
        betas bs B' Us U-loop c-loop child-costs-loop
```

```
    and complete:
      complete-on d' {v ∈ bound-tree S B'. ?d-fp v = dist s v}
    and U-def:
      U = U-loop ∪ {v ∈ bound-tree S B'. ?d-fp v = dist s v}
    and c-def:
      c = fp-iter-capped-scan-cost k cap d S S B + c-insert + c-loop
    using Exact-Concrete-Step by blast+
  have view: partition-state-view D = label-partition-view ?d-fp ?P
    using exact-partition-initial-state-view[OF init] .
  have insert-cost: partition-initial-insert-cost-bound c-insert t ?P
    using exact-partition-initial-state-cost-bound[OF init] .
  have loop-ref:
    direct-insert-costed-partition-loop-state Δ M-of t h k cap l ?d-fp
      ?P B d' (partition-state-view D) a betas bs B' Us U-loop c-loop
      child-costs-loop
    using Exact-Concrete-Step by blast
  show ?case
    by (rule Direct-Insert-Costed-Step
      [OF view insert-cost loop-ref complete U-def c-def])
qed

theorem exact-concrete-bmssp-Suc-success-or-threshold:
  assumes run:
      exact-concrete-bmssp Δ M-of t h k cap (Suc l) d S B d' B' U c
    and sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
  shows B' = B ∨ k * cap ≤ card U
  by (rule direct-insert-costed-bmssp-Suc-success-or-threshold
    [OF exact-concrete-bmssp-refines-direct-insert[OF run] sound pre
      S-reaches])

theorem exact-concrete-bmssp-correct:
  assumes run:
    exact-concrete-bmssp Δ M-of t h k cap l d S B d' B' U c
    and sound: sound-label d
    and pre: bmssp-pre-full d S B
    and S-reaches: ⋀x. x ∈ S ⟹ reachable s x
  shows bmssp-post-full d S B d' B' U
  by (rule direct-insert-costed-bmssp-correct
    [OF exact-concrete-bmssp-refines-direct-insert[OF run] sound pre S-reaches])

end

end
theory BMSSP-Top-Level-Bounds
  imports BMSSP-Strict-Tie-Breaking BMSSP-Exact-Concrete-Cost
begin
```

36 Top-Level BMSSP Theorems

This theory is the entry point for the checked correctness result for the top-level BMSSP algorithm modelled in this development. Everything before this point proves local correctness and local cost statements: FindPivots preserves the complete-source invariant, the partition loop realizes the recursive BMSSP step, and the concrete partition operations account for Insert, BatchPrepend, and Pull. This file is where those ingredients are closed over a root call and converted into the single-source shortest-path headline.

The algorithmic run relations used here are the strongest ones in the project. They record the capped FindPivots step, the exact pulled child source, the range-synchronised recursive calls, and the separated edge and source batch costs. The early theorems therefore look a little verbose: their purpose is to expose a precise bridge from the executable recurrence to the abstract correctness theorem while retaining enough cost structure for the asymptotic proof.

The locale assumption is the paper's tie-breaking consequence: shortest paths form a strict tree order. Under that assumption, the proof instantiates the BMSSP parameters with *sssp-log-one-third-param* and *sssp-log-two-thirds-param*. These correspond to the paper's $log^{\frown}\{1/3\}\ n$ and $log^{\frown}\{2/3\}\ n$ scales. The level capacity is computed by *bmssp-level-cap*; the point of the schedule is that the recursive tree depth, pivot growth, and bucketed partition cost telescope into the target $O(m * log^{\frown}\{2/3\}\ n)$ envelope.

The file proceeds in four stages. First, it proves root-level correctness for operational and costed runs. Second, it specializes the refined graph bound to the logarithmic parameter schedule. Third, it packages the least top-level execution cost as a function and proves Big-O bounds for it. Finally, it exposes locale-level headline theorems that downstream theories can cite without reopening the cost recurrence.

context *strict-tie-breaking-digraph*
begin

theorem *operational-range-split-algorithm-correct*:
  assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
    and *run*:
      full-operational-bmssp k cap l
        finite-initial-label $\{s\}$ *Infinity d′ Infinity U*
  shows $U = V \wedge$ *sssp-correct d′*
proof
  have *pre*: *bmssp-pre-full finite-initial-label* $\{s\}$ *Infinity*
    using *all-reachable finite-initial-label-source-complete*
    by (*rule top-bmssp-pre-full*)
  have *sound*: *sound-label finite-initial-label*
    using *finite-initial-label-sound*[*OF all-reachable*] **.**
  have *S-reaches*: $\bigwedge x.\ x \in \{s\} \Longrightarrow$ *reachable s x*

```
    using all-reachable source-in-V by blast
  have post-full:
    bmssp-post-full finite-initial-label {s} Infinity d' Infinity U
    by (rule full-operational-bmssp-correct[OF run sound pre S-reaches])
  have post: bmssp-post finite-initial-label {s} Infinity d' Infinity U
    using bmssp-post-full-imp-post[OF post-full] .
  show U = V
    using post bound-tree-source-infinity[OF all-reachable]
    unfolding bmssp-post-def by auto
  show sssp-correct d'
    using finite-initial-label-full-operational-top-level-correct
      [OF all-reachable run] .
qed

theorem operational-range-split-algorithm-reachable-correct:
  assumes run:
      full-operational-bmssp k cap l
        finite-initial-label {s} Infinity d' Infinity U
  shows sssp-correct d'
proof −
  have pre: bmssp-pre-full finite-initial-label {s} Infinity
    using finite-initial-label-source-complete
    by (rule top-bmssp-pre-full-reachable)
  have sound: sound-label finite-initial-label
    by (rule finite-initial-label-sound-reachable)
  have S-reaches: ⋀x. x ∈ {s} ⟹ reachable s x
    using reachable-refl[OF source-in-V] by blast
  have post-full:
    bmssp-post-full finite-initial-label {s} Infinity d' Infinity U
    by (rule full-operational-bmssp-correct[OF run sound pre S-reaches])
  have post: bmssp-post finite-initial-label {s} Infinity d' Infinity U
    using bmssp-post-full-imp-post[OF post-full] .
  then show ?thesis
    by (rule successful-root-bmssp-sssp-correct)
qed

theorem range-split-algorithm-correct:
  assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
    and run:
      exact-split-range-costed-bmssp Δ (bmssp-level-cap k t) t h k
        (bmssp-level-cap k t l) l
        finite-initial-label {s} Infinity d' Infinity U c
  shows U = V ∧ sssp-correct d'
proof
  have pre: bmssp-pre-full finite-initial-label {s} Infinity
    using all-reachable finite-initial-label-source-complete
    by (rule top-bmssp-pre-full)
  have sound: sound-label finite-initial-label
    using finite-initial-label-sound[OF all-reachable] .
```

```
  have S-reaches: ⋀x. x ∈ {s} ⟹ reachable s x
    using all-reachable source-in-V by blast
  have post-full:
    bmssp-post-full finite-initial-label {s} Infinity d' Infinity U
    by (rule exact-split-range-costed-bmssp-correct
      [OF run sound pre S-reaches])
  have post: bmssp-post finite-initial-label {s} Infinity d' Infinity U
    using bmssp-post-full-imp-post[OF post-full] .
  show U = V
    using post bound-tree-source-infinity[OF all-reachable]
    unfolding bmssp-post-def by auto
  show sssp-correct d'
    by (rule finite-initial-label-exact-split-range-costed-top-level-correct
      [OF all-reachable run])
qed

theorem range-split-algorithm-reachable-correct:
  assumes run:
      exact-split-range-costed-bmssp Δ (bmssp-level-cap k t) t h k
        (bmssp-level-cap k t l) l
        finite-initial-label {s} Infinity d' Infinity U c
  shows sssp-correct d'
proof −
  have pre: bmssp-pre-full finite-initial-label {s} Infinity
    using finite-initial-label-source-complete
    by (rule top-bmssp-pre-full-reachable)
  have sound: sound-label finite-initial-label
    by (rule finite-initial-label-sound-reachable)
  have S-reaches: ⋀x. x ∈ {s} ⟹ reachable s x
    using reachable-refl[OF source-in-V] by blast
  have post-full:
    bmssp-post-full finite-initial-label {s} Infinity d' Infinity U
    by (rule exact-split-range-costed-bmssp-correct
      [OF run sound pre S-reaches])
  have post: bmssp-post finite-initial-label {s} Infinity d' Infinity U
    using bmssp-post-full-imp-post[OF post-full] .
  then show ?thesis
    by (rule successful-root-bmssp-sssp-correct)
qed

theorem direct-insert-algorithm-correct:
  assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
    and run:
      direct-insert-costed-bmssp Δ (bmssp-level-cap k t) t h k
        (bmssp-level-cap k t l) l
        finite-initial-label {s} Infinity d' Infinity U c
  shows U = V ∧ sssp-correct d'
proof
  have pre: bmssp-pre-full finite-initial-label {s} Infinity
```

using *all-reachable finite-initial-label-source-complete*
by (*rule top-bmssp-pre-full*)
have *sound*: *sound-label finite-initial-label*
using *finite-initial-label-sound*[*OF all-reachable*] **.**
have *S-reaches*: $\bigwedge x.\ x \in \{s\} \Longrightarrow$ *reachable s x*
using *all-reachable source-in-V* **by** *blast*
have *post-full*:
bmssp-post-full finite-initial-label $\{s\}$ *Infinity* d' *Infinity U*
by (*rule direct-insert-costed-bmssp-correct*[*OF run sound pre S-reaches*])
have *post*: *bmssp-post finite-initial-label* $\{s\}$ *Infinity* d' *Infinity U*
using *bmssp-post-full-imp-post*[*OF post-full*] **.**
show $U = V$
using *post bound-tree-source-infinity*[*OF all-reachable*]
unfolding *bmssp-post-def* **by** *auto*
show *sssp-correct* d'
by (*rule finite-initial-label-direct-insert-costed-top-level-correct*
[*OF all-reachable run*])
qed

theorem *direct-insert-algorithm-reachable-correct*:
assumes *run*:
direct-insert-costed-bmssp Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) *l*
finite-initial-label $\{s\}$ *Infinity* d' *Infinity U c*
shows *sssp-correct* d'
by (*rule finite-initial-label-direct-insert-costed-top-level-reachable-correct*[*OF run*])

theorem *direct-insert-algorithm-correct-and-refined-graph-bound-under-edge-budget*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \le A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \le A * k$
and *insert-scaled-factor*: $t \le A\text{-}insert * k$
and *seen-scaled-factor*: $k * \Delta + A\text{-}insert \le 2 * A$
and *source-factor*: $Suc\ h \le 2 * A$
and *k-pos*: $0 < k$
and *edge-budget*:
$\bigwedge l'\ d\ P\ B\ d'\ D\ a\ betas\ bs\ B'\ Us\ U\text{-}loop\ c\text{-}loop\ child\text{-}costs.$
direct-insert-costed-partition-loop-state Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) l' d P B d' D a *betas bs* B'
Us U-loop c-loop child-costs $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d P B $\Longrightarrow$
$(\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*$) \Longrightarrow$
t $*$ *sum-list*
(*map card* (*range-tree-child-direct-edge-range-list P B a betas bs*)) $+$
h $*$ *sum-list*
(*map card* (*range-tree-child-edge-range-list P a betas bs*))

```
      ≤ H * card (outgoing-edges U-loop)
    and run:
      direct-insert-costed-bmssp Δ (bmssp-level-cap k t) t h k
        (bmssp-level-cap k t l) l
        finite-initial-label {s} Infinity d' Infinity U c
  shows U = V ∧ sssp-correct d' ∧
    c ≤ bmssp-refined-graph-time-bound (λ-. A) (λ-. R) (λ-. H)
      (λ-. l) (λ-. t) (λ-. edge-count) vertex-count
proof −
  have UV-correct: U = V ∧ sssp-correct d'
    by (rule direct-insert-algorithm-correct[OF all-reachable run])
  have cost:
    sssp-correct d' ∧
      c ≤ bmssp-refined-graph-time-bound (λ-. A) (λ-. R) (λ-. H)
        (λ-. l) (λ-. t) (λ-. edge-count) vertex-count
    by (rule finite-initial-label-direct-insert-costed-top-level-correct-and-closed-bmssp-
refined-graph-time-bound-level-cap
      [OF all-reachable degree degree-factor R-pos insert-factor insert-scaled-factor
        seen-scaled-factor source-factor k-pos edge-budget run])
  show ?thesis
    using UV-correct cost by blast
qed

theorem direct-insert-algorithm-correct-and-refined-graph-bound-trivial-edge-
budget:
  assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
    and degree: edge-outdegree-le Δ
    and degree-factor: Δ ≤ A
    and R-pos: 0 < R
    and insert-factor: t ≤ A * k
    and insert-scaled-factor: t ≤ A-insert * k
    and seen-scaled-factor: k * Δ + A-insert ≤ 2 * A
    and source-factor: Suc h ≤ 2 * A
    and k-pos: 0 < k
    and run:
      direct-insert-costed-bmssp Δ (bmssp-level-cap k t) t h k
        (bmssp-level-cap k t l) l
        finite-initial-label {s} Infinity d' Infinity U c
  shows U = V ∧ sssp-correct d' ∧
    c ≤ bmssp-refined-graph-time-bound (λ-. A) (λ-. R) (λ-. t + h)
      (λ-. l) (λ-. t) (λ-. edge-count) vertex-count
proof −
  have UV-correct: U = V ∧ sssp-correct d'
    by (rule direct-insert-algorithm-correct[OF all-reachable run])
  have cost:
    sssp-correct d' ∧
      c ≤ bmssp-refined-graph-time-bound (λ-. A) (λ-. R) (λ-. t + h)
        (λ-. l) (λ-. t) (λ-. edge-count) vertex-count
    by (rule finite-initial-label-direct-insert-costed-top-level-correct-and-closed-bmssp-
```

```
refined-graph-time-bound-level-cap-trivial-edge-budget
      [OF all-reachable degree degree-factor R-pos insert-factor insert-scaled-factor
        seen-scaled-factor source-factor k-pos run])
  show ?thesis
    using UV-correct cost by blast
qed

theorem direct-insert-algorithm-correct-and-refined-graph-bound-amortized:
  assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
    and degree: edge-outdegree-le Δ
    and degree-factor: Δ ≤ A
    and R-pos: 0 < R
    and insert-factor: t ≤ A * k
    and insert-scaled-factor: t ≤ A-insert * k
    and seen-scaled-factor: k * Δ + A-insert ≤ 2 * A
    and source-factor: Suc h ≤ 2 * A
    and k-pos: 0 < k
    and run:
      direct-insert-costed-bmssp Δ (bmssp-level-cap k t) t h k
        (bmssp-level-cap k t l) l
        finite-initial-label {s} Infinity d' Infinity U c
  shows U = V ∧ sssp-correct d' ∧
    c ≤ bmssp-refined-graph-time-bound (λ-. A) (λ-. R) (λ-. h)
      (λ-. l) (λ-. t) (λ-. edge-count) vertex-count
proof −
  have UV-correct: U = V ∧ sssp-correct d'
    by (rule direct-insert-algorithm-correct[OF all-reachable run])
  have cost:
    sssp-correct d' ∧
      c ≤ bmssp-refined-graph-time-bound (λ-. A) (λ-. R) (λ-. h)
        (λ-. l) (λ-. t) (λ-. edge-count) vertex-count
    by (rule finite-initial-label-direct-insert-costed-top-level-correct-and-closed-bmssp-
refined-graph-time-bound-level-cap-amortized
      [OF all-reachable degree degree-factor R-pos insert-factor insert-scaled-factor
        seen-scaled-factor source-factor k-pos run])
  show ?thesis
    using UV-correct cost by blast
qed
```

The preceding theorems keep the parameters abstract. They are useful because they show exactly which inequalities are required of a costed BMSSP run: outdegree must be bounded, Insert must be charged at the chosen parameter scale, and the source and seen-set terms must fit into the refined graph bound *bmssp-refined-graph-time-bound*. The amortized variant is the one that matches the bucketed partition analysis: the split work is not charged to each operation eagerly, but through the aggregate accounting used by the partition interface.

The next group substitutes the logarithmic schedule. The one-third parameter controls the number of levels and source growth, while the two-

thirds parameter controls the Insert scale supplied to the partition layer. The simple arithmetic lemmas in *sssp-log-one-third-param* and *sssp-log-two-thirds-param* prove that these choices satisfy the abstract side conditions.

theorem *direct-insert-algorithm-correct-and-refined-graph-bound-log-params-bounded-degree*:
  fixes N p q :: *nat*
  defines *p-def*: $p \equiv$ *sssp-log-one-third-param* N
    and *q-def*: $q \equiv$ *sssp-log-two-thirds-param* N
  assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable* s v
    and *degree*: *edge-outdegree-le* Δ
    and *run*:
      direct-insert-costed-bmssp Δ (*bmssp-level-cap* p q) q p p
        (*bmssp-level-cap* p q p) p
        finite-initial-label $\{s\}$ *Infinity* d' *Infinity* U c
  shows $U = V \wedge$ *sssp-correct* $d' \wedge$
    $c \leq$ *bmssp-refined-graph-time-bound*
      (λ-. *Suc* $\Delta * p$) (λ-. q) (λ-. p)
      (λ-. p) (λ-. q) (λ-. *edge-count*) *vertex-count*
proof −
  have *p-pos*: $0 < p$
    unfolding *p-def* **by** *simp*
  have *q-pos*: $0 < q$
    unfolding *q-def* **by** *simp*
  have *q-le-square*: $q \leq p * p$
    unfolding *p-def q-def*
    by (*rule sssp-log-two-thirds-param-le-one-third-square*)
  have *degree-factor*: $\Delta \leq$ *Suc* $\Delta * p$
    using *p-pos* **by** (*cases p*) *simp-all*
  have *insert-factor*: $q \leq$ (*Suc* $\Delta * p$) $* p$
  proof −
    have $p * p \leq$ (*Suc* $\Delta * p$) $* p$
      by *simp*
    then show *?thesis*
      using *q-le-square* **by** *linarith*
  qed
  have *insert-scaled-factor*: $q \leq p * p$
    using *q-le-square* **.**
  have *seen-scaled-factor*: $p * \Delta + p \leq 2 *$ (*Suc* $\Delta * p$)
  proof −
    have $p * \Delta + p =$ *Suc* $\Delta * p$
      by (*simp add*: *algebra-simps*)
    also have $\ldots \leq 2 *$ (*Suc* $\Delta * p$)
      by *simp*
    finally show *?thesis* **.**
  qed
  have *source-factor*: *Suc* $p \leq 2 *$ (*Suc* $\Delta * p$)
  proof −
    have *Suc* $p \leq 2 * p$
      using *p-pos* **by** *simp*

```
    also have ... ≤ 2 * (Suc Δ * p)
      by simp
    finally show ?thesis .
  qed
  show ?thesis
    by (rule direct-insert-algorithm-correct-and-refined-graph-bound-amortized
      [OF all-reachable degree degree-factor q-pos insert-factor
        insert-scaled-factor seen-scaled-factor source-factor p-pos run])
qed

theorem direct-insert-algorithm-correct-and-refined-graph-bound-log-params-fixed-
degree:
  fixes N D p q :: nat
  defines p-def: p ≡ sssp-log-one-third-param N
    and q-def: q ≡ sssp-log-two-thirds-param N
  assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
    and degree: edge-outdegree-le D
    and run:
      direct-insert-costed-bmssp D (bmssp-level-cap p q) q p p
        (bmssp-level-cap p q p) p
        finite-initial-label {s} Infinity d' Infinity U c
  shows U = V ∧ sssp-correct d' ∧
    c ≤ bmssp-refined-graph-time-bound
      (λ-. Suc D * p) (λ-. q) (λ-. p)
      (λ-. p) (λ-. q) (λ-. edge-count) vertex-count
proof −
  have run':
    direct-insert-costed-bmssp D
      (bmssp-level-cap (sssp-log-one-third-param N) (sssp-log-two-thirds-param N))
      (sssp-log-two-thirds-param N) (sssp-log-one-third-param N)
      (sssp-log-one-third-param N)
      (bmssp-level-cap (sssp-log-one-third-param N) (sssp-log-two-thirds-param N)
        (sssp-log-one-third-param N))
      (sssp-log-one-third-param N)
      finite-initial-label {s} Infinity d' Infinity U c
    using run unfolding p-def q-def by simp
  show ?thesis
    unfolding p-def q-def
    by (rule direct-insert-algorithm-correct-and-refined-graph-bound-log-params-
bounded-degree
      [OF all-reachable degree run'])
qed

theorem exact-concrete-algorithm-correct-and-refined-graph-bound-log-params-fixed-
degree:
  fixes N D p q :: nat
  defines p-def: p ≡ sssp-log-one-third-param N
    and q-def: q ≡ sssp-log-two-thirds-param N
  assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
```

```
    and degree: edge-outdegree-le D
    and run:
      exact-concrete-bmssp D (bmssp-level-cap p q) q p p
        (bmssp-level-cap p q p) p
        finite-initial-label {s} Infinity d' Infinity U c
  shows U = V ∧ sssp-correct d' ∧
    c ≤ bmssp-refined-graph-time-bound
      (λ-. Suc D * p) (λ-. q) (λ-. p)
      (λ-. p) (λ-. q) (λ-. edge-count) vertex-count
proof −
  have direct-run:
    direct-insert-costed-bmssp D (bmssp-level-cap p q) q p p
      (bmssp-level-cap p q p) p
      finite-initial-label {s} Infinity d' Infinity U c
    by (rule exact-concrete-bmssp-refines-direct-insert[OF run])
  have direct-run':
    direct-insert-costed-bmssp D
      (bmssp-level-cap (sssp-log-one-third-param N) (sssp-log-two-thirds-param N))
      (sssp-log-two-thirds-param N) (sssp-log-one-third-param N)
      (sssp-log-one-third-param N)
      (bmssp-level-cap (sssp-log-one-third-param N) (sssp-log-two-thirds-param N)
        (sssp-log-one-third-param N))
      (sssp-log-one-third-param N)
      finite-initial-label {s} Infinity d' Infinity U c
    using direct-run unfolding p-def q-def by simp
  show ?thesis
    unfolding p-def q-def
    by (rule direct-insert-algorithm-correct-and-refined-graph-bound-log-params-fixed-
degree
      [OF all-reachable degree direct-run'])
qed
```

The definitions below package the root execution as an ordinary total cost predicate. *exact-concrete-top-level-run* fixes the root source set to $\{s\}$ and the input bound to *Infinity*; the auxiliary *exact-concrete-root-run* keeps the returned bound visible for existence proofs that reason about threshold cases. The least-cost wrapper *exact-concrete-top-level-time* is not an algorithm in its own right; it is a mathematical way to speak about the cost of any concrete run satisfying the checked run relation.

```
definition exact-concrete-top-level-run ::
  nat ⇒ nat ⇒ ('a ⇒ real) ⇒ 'a set ⇒ nat ⇒ bool where
  exact-concrete-top-level-run D N d' U c ⟷
    (let p = sssp-log-one-third-param N;
         q = sssp-log-two-thirds-param N
     in exact-concrete-bmssp D (bmssp-level-cap p q) q p p
       (bmssp-level-cap p q p) p
       finite-initial-label {s} Infinity d' Infinity U c)

definition exact-concrete-root-run ::
```

```
  nat ⇒ nat ⇒ ('a ⇒ real) ⇒ bound ⇒ 'a set ⇒ nat ⇒ bool where
  exact-concrete-root-run D N d' B' U c ⟷
    (let p = sssp-log-one-third-param N;
         q = sssp-log-two-thirds-param N
     in exact-concrete-bmssp D (bmssp-level-cap p q) q p p
       (bmssp-level-cap p q p) p
       finite-initial-label {s} Infinity d' B' U c)

lemma exact-concrete-top-level-run-root-run-iff:
  exact-concrete-top-level-run D N d' U c ⟷
    exact-concrete-root-run D N d' Infinity U c
  unfolding exact-concrete-top-level-run-def exact-concrete-root-run-def
  by (simp add: Let-def)

definition exact-concrete-top-level-cost :: nat ⇒ nat ⇒ nat ⇒ bool where
  exact-concrete-top-level-cost D N c ⟷
    (∃ d' U. exact-concrete-top-level-run D N d' U c)

definition exact-concrete-top-level-time :: nat ⇒ nat ⇒ nat where
  exact-concrete-top-level-time D N =
    (LEAST c. exact-concrete-top-level-cost D N c)

lemma exact-concrete-top-level-cost-exists-if-top-pivots-empty:
  assumes pivots-empty:
    find-pivots-pivots-capped (sssp-log-one-third-param N)
      (bmssp-level-cap (sssp-log-one-third-param N)
        (sssp-log-two-thirds-param N) (sssp-log-one-third-param N))
      finite-initial-label {s} Infinity = {}
  shows ∃ c. exact-concrete-top-level-cost D N c
proof −
  let ?p = sssp-log-one-third-param N
  let ?q = sssp-log-two-thirds-param N
  have p-eq: ?p = Suc (?p − 1)
    using sssp-log-one-third-param-pos[of N] by simp
  have source-subset: {s} ⊆ V
    using source-in-V by simp
  have run:
    ∃ d' U c.
      exact-concrete-bmssp D (bmssp-level-cap ?p ?q) ?q ?p ?p
        (bmssp-level-cap ?p ?q ?p) (Suc (?p − 1))
        finite-initial-label {s} Infinity d' Infinity U c
    by (rule exact-concrete-bmssp-Suc-exists-if-pivots-empty-same-bound
      [OF source-subset - pivots-empty]) simp
  then obtain d' U c where
    exact-concrete-bmssp D (bmssp-level-cap ?p ?q) ?q ?p ?p
      (bmssp-level-cap ?p ?q ?p) (Suc (?p − 1))
      finite-initial-label {s} Infinity d' Infinity U c
    by blast
  then have root-run:
```

```
    exact-concrete-bmssp D (bmssp-level-cap ?p ?q) ?q ?p ?p
      (bmssp-level-cap ?p ?q ?p) ?p
      finite-initial-label {s} Infinity d' Infinity U c
    using p-eq by simp
  have exact-concrete-top-level-run D N d' U c
    unfolding exact-concrete-top-level-run-def
    using root-run by (simp add: Let-def)
  then have exact-concrete-top-level-cost D N c
    unfolding exact-concrete-top-level-cost-def by blast
  then show ?thesis
    by blast
qed

lemma exact-concrete-top-level-cost-exists-from-initial-loop:
  fixes D N p q cap l :: nat
    and d-fp :: 'a ⇒ real
    and P :: 'a set
  defines p-def: p ≡ sssp-log-one-third-param N
    and q-def: q ≡ sssp-log-two-thirds-param N
    and cap-def: cap ≡ bmssp-level-cap p q p
    and l-def: l ≡ p − 1
    and d-fp-def:
      d-fp ≡ find-pivots-label-capped p cap finite-initial-label {s}
        Infinity
    and P-def:
      P ≡ find-pivots-pivots-capped p cap finite-initial-label {s}
        Infinity
  assumes loop:
      ⋀D-part. exact-partition-initial-state q d-fp P D-part
        (q * card P) ⟹
        ∃ d' a betas bs Us U-loop c-loop child-costs-loop.
          exact-concrete-partition-loop-state D (bmssp-level-cap p q)
            q p p cap l d-fp P Infinity d' D-part a betas bs Infinity
            Us U-loop c-loop child-costs-loop ∧
          complete-on d'
            {v ∈ bound-tree {s} Infinity. d-fp v = dist s v}
  shows ∃ c. exact-concrete-top-level-cost D N c
proof −
  have p-eq: p = Suc l
    using sssp-log-one-third-param-pos[of N]
    unfolding p-def l-def by simp
  have source-subset: {s} ⊆ V
    using source-in-V by simp
  obtain D-part where init:
    exact-partition-initial-state q d-fp P D-part (q * card P)
    using exact-partition-initial-state-exists-for-capped-pivots-with-cost
      [OF source-subset]
    unfolding d-fp-def P-def by blast
  obtain d' a betas bs Us U-loop c-loop child-costs-loop where
```

```
  loop-run:
    exact-concrete-partition-loop-state D (bmssp-level-cap p q)
      q p p cap l d-fp P Infinity d' D-part a betas bs Infinity
      Us U-loop c-loop child-costs-loop
  and complete:
    complete-on d'
      {v ∈ bound-tree {s} Infinity. d-fp v = dist s v}
  using loop[OF init] by blast
have init-actual:
  exact-partition-initial-state q
    (find-pivots-label-capped p cap finite-initial-label {s} Infinity)
    (find-pivots-pivots-capped p cap finite-initial-label {s} Infinity)
    D-part
    (q * card
      (find-pivots-pivots-capped p cap finite-initial-label {s}
        Infinity))
  using init unfolding d-fp-def P-def .
have loop-actual:
  exact-concrete-partition-loop-state D (bmssp-level-cap p q)
    q p p cap l
    (find-pivots-label-capped p cap finite-initial-label {s} Infinity)
    (find-pivots-pivots-capped p cap finite-initial-label {s} Infinity)
    Infinity d' D-part a betas bs Infinity Us U-loop c-loop
    child-costs-loop
  using loop-run unfolding d-fp-def P-def .
have complete-actual:
  complete-on d'
    {v ∈ bound-tree {s} Infinity.
      find-pivots-label-capped p cap finite-initial-label {s}
        Infinity v = dist s v}
  using complete unfolding d-fp-def .
let ?U =
  U-loop ∪
    {v ∈ bound-tree {s} Infinity.
      find-pivots-label-capped p cap finite-initial-label {s}
        Infinity v = dist s v}
have run-root-suc:
  exact-concrete-bmssp D (bmssp-level-cap p q) q p p cap (Suc l)
    finite-initial-label {s} Infinity d' Infinity ?U
    (fp-iter-capped-scan-cost p cap finite-initial-label {s} {s} Infinity +
      q * card
        (find-pivots-pivots-capped p cap finite-initial-label {s}
          Infinity) + c-loop)
  by (rule exact-concrete-bmssp-step-with-exact-insert-cost
    [OF init-actual loop-actual complete-actual refl])
let ?c =
  fp-iter-capped-scan-cost p cap finite-initial-label {s} {s} Infinity +
    q * card
      (find-pivots-pivots-capped p cap finite-initial-label {s}
```

```
      Infinity) + c-loop
  have run-root:
    exact-concrete-bmssp D (bmssp-level-cap p q) q p p cap p
     finite-initial-label {s} Infinity d' Infinity ?U ?c
    using run-root-suc p-eq by simp
  have exact-concrete-top-level-run D N d' ?U ?c
    unfolding exact-concrete-top-level-run-def
    using run-root
    by (simp add: Let-def p-def q-def cap-def)
  then have exact-concrete-top-level-cost D N ?c
    unfolding exact-concrete-top-level-cost-def by blast
  then show ?thesis
    by blast
qed

lemma exact-concrete-top-level-cost-exists-from-root-run-below-threshold:
  assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
    and below-threshold:
      vertex-count <
        sssp-log-one-third-param N *
        bmssp-level-cap (sssp-log-one-third-param N)
          (sssp-log-two-thirds-param N) (sssp-log-one-third-param N)
    and run: exact-concrete-root-run D N d' B' U c
  shows ∃c. exact-concrete-top-level-cost D N c
proof −
  let ?p = sssp-log-one-third-param N
  let ?q = sssp-log-two-thirds-param N
  let ?cap = bmssp-level-cap ?p ?q ?p
  have p-pos: 0 < ?p
    by simp
  have run-bmssp:
    exact-concrete-bmssp D (bmssp-level-cap ?p ?q) ?q ?p ?p ?cap ?p
     finite-initial-label {s} Infinity d' B' U c
    using run unfolding exact-concrete-root-run-def by (simp add: Let-def)
  have run-suc:
    exact-concrete-bmssp D (bmssp-level-cap ?p ?q) ?q ?p ?p ?cap
     (Suc (?p − 1)) finite-initial-label {s} Infinity d' B' U c
    using run-bmssp p-pos by simp
  have pre: bmssp-pre-full finite-initial-label {s} Infinity
    using all-reachable finite-initial-label-source-complete
    by (rule top-bmssp-pre-full)
  have sound: sound-label finite-initial-label
    using finite-initial-label-sound[OF all-reachable] .
  have source-reaches:
    ⋀x. x ∈ {s} ⟹ reachable s x
    using all-reachable source-in-V by blast
  have success-or-threshold: B' = Infinity ∨ ?p * ?cap ≤ card U
    by (rule exact-concrete-bmssp-Suc-success-or-threshold
     [OF run-suc sound pre source-reaches])
```

```
  have post: bmssp-post-full finite-initial-label {s} Infinity d' B' U
    by (rule exact-concrete-bmssp-correct
      [OF run-bmssp sound pre source-reaches])
  have U-subset: U ⊆ V
    using post unfolding bmssp-post-full-def bound-tree-def by blast
  have card-U-le: card U ≤ vertex-count
    by (rule card-subset-vertex-count[OF U-subset])
  have not-threshold: ¬ ?p * ?cap ≤ card U
    using below-threshold card-U-le by linarith
  have B'-eq: B' = Infinity
    using success-or-threshold not-threshold by blast
  have top-run: exact-concrete-top-level-run D N d' U c
    using run B'-eq
    unfolding exact-concrete-top-level-run-root-run-iff
      exact-concrete-root-run-def
    by simp
  then have exact-concrete-top-level-cost D N c
    unfolding exact-concrete-top-level-cost-def by blast
  then show ?thesis
    by blast
qed

lemma eventually-exact-concrete-top-level-cost-if-root-runs-exist-below-threshold:
  assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
    and below-threshold:
      eventually
        (λN. vertex-count <
          sssp-log-one-third-param N *
          bmssp-level-cap (sssp-log-one-third-param N)
            (sssp-log-two-thirds-param N) (sssp-log-one-third-param N))
        at-top
    and root-runs:
      eventually
        (λN. ∃ d' B' U c. exact-concrete-root-run D N d' B' U c)
        at-top
  shows eventually (λN. ∃ c. exact-concrete-top-level-cost D N c) at-top
  using below-threshold root-runs
proof eventually-elim
  case (elim N)
  then obtain d' B' U c where run:
    exact-concrete-root-run D N d' B' U c
    by blast
  show ?case
    by (rule exact-concrete-top-level-cost-exists-from-root-run-below-threshold
      [OF all-reachable elim(1) run])
qed

lemma eventually-exact-concrete-top-level-cost-from-empty-top-pivots:
  eventually (λN. ∃ c. exact-concrete-top-level-cost D N c) at-top
```

using *eventually-top-level-pivots-empty*
[*of finite-initial-label Infinity*]
proof *eventually-elim*
case (*elim N*)
show *?case*
by (*rule exact-concrete-top-level-cost-exists-if-top-pivots-empty*
[*OF elim*])
qed

lemma *exact-concrete-top-level-time-witness*:
assumes *exact-concrete-top-level-cost D N c*
shows *exact-concrete-top-level-cost D N*
(*exact-concrete-top-level-time D N*)
unfolding *exact-concrete-top-level-time-def*
by (*rule LeastI-ex*) (*use assms* **in** *blast*)

lemma *exact-concrete-top-level-time-le*:
assumes *exact-concrete-top-level-cost D N c*
shows *exact-concrete-top-level-time D N* $\leq$ *c*
unfolding *exact-concrete-top-level-time-def*
by (*rule Least-le*) (*rule assms*)

lemma *eventually-exact-concrete-top-level-cost-if-all*:
assumes $\bigwedge N.\ \exists\, c.$ *exact-concrete-top-level-cost D N c*
shows *eventually* ($\lambda N.\ \exists\, c.$ *exact-concrete-top-level-cost D N c*) *at-top*
by (*rule eventuallyI*) (*rule assms*)

lemma *eventually-exact-concrete-top-level-cost-square-if-all*:
assumes $\bigwedge N.\ \exists\, c.$ *exact-concrete-top-level-cost D N c*
shows *eventually*
($\lambda N.\ \exists\, c.$ *exact-concrete-top-level-cost D* (*N* $*$ *N*) *c*) *at-top*
by (*rule eventuallyI*) (*rule assms*)

theorem *exact-concrete-top-level-run-correct-and-refined-bound-log-params-fixed-degree*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le D*
and *run*: *exact-concrete-top-level-run D N d' U c*
shows *U* = *V* $\wedge$ *sssp-correct d'* $\wedge$
c $\leq$ *bmssp-refined-graph-time-bound*
(λ-. *Suc D* $*$ *sssp-log-one-third-param N*)
(λ-. *sssp-log-two-thirds-param N*)
(λ-. *sssp-log-one-third-param N*)
(λ-. *sssp-log-one-third-param N*)
(λ-. *sssp-log-two-thirds-param N*)
(λ-. *edge-count*) *vertex-count*
proof −
let *?p* = *sssp-log-one-third-param N*
let *?q* = *sssp-log-two-thirds-param N*

have *run'*:
exact-concrete-bmssp D (*bmssp-level-cap ?p ?q*) *?q ?p ?p*
(*bmssp-level-cap ?p ?q ?p*) *?p*
finite-initial-label {*s*} *Infinity d' Infinity U c*
using *run* **unfolding** *exact-concrete-top-level-run-def* **by** (*simp add*: *Let-def*)
show *?thesis*
by (*rule exact-concrete-algorithm-correct-and-refined-graph-bound-log-params-fixed-degree*
[*OF all-reachable degree run'*])
qed

theorem *exact-concrete-top-level-cost-refined-bound-log-params-fixed-degree*:
assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
and *degree*: *edge-outdegree-le D*
and *cost*: *exact-concrete-top-level-cost D N c*
shows *c* ≤ *bmssp-refined-graph-time-bound*
(λ-. *Suc D* ∗ *sssp-log-one-third-param N*)
(λ-. *sssp-log-two-thirds-param N*)
(λ-. *sssp-log-one-third-param N*)
(λ-. *sssp-log-one-third-param N*)
(λ-. *sssp-log-two-thirds-param N*)
(λ-. *edge-count*) *vertex-count*
proof −
obtain *d' U* **where** *run*: *exact-concrete-top-level-run D N d' U c*
using *cost* **unfolding** *exact-concrete-top-level-cost-def* **by** *blast*
show *?thesis*
using *exact-concrete-top-level-run-correct-and-refined-bound-log-params-fixed-degree*
[*OF all-reachable degree run*]
by *blast*
qed

theorem *exact-concrete-top-level-time-refined-bound-log-params-fixed-degree*:
assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
and *degree*: *edge-outdegree-le D*
and *run-exists*: *exact-concrete-top-level-cost D N c*
shows *exact-concrete-top-level-time D N* ≤
bmssp-refined-graph-time-bound
(λ-. *Suc D* ∗ *sssp-log-one-third-param N*)
(λ-. *sssp-log-two-thirds-param N*)
(λ-. *sssp-log-one-third-param N*)
(λ-. *sssp-log-one-third-param N*)
(λ-. *sssp-log-two-thirds-param N*)
(λ-. *edge-count*) *vertex-count*
proof −
have *cost*:
exact-concrete-top-level-cost D N (*exact-concrete-top-level-time D N*)
by (*rule exact-concrete-top-level-time-witness*[*OF run-exists*])
show *?thesis*

by (*rule exact-concrete-top-level-cost-refined-bound-log-params-fixed-degree*
[*OF all-reachable degree cost*])
qed

From this point onward, the file is purely asymptotic. The function *T-bmssp* abbreviates the least top-level concrete cost, and the following theorems compare it with increasingly convenient targets. The central comparison target is *sssp-time-target*, which expands to an edge-count term multiplied by the two-thirds logarithmic factor. The proofs reuse the complexity lemmas for *bmssp-refined-graph-time-bound*; no algorithmic invariant is reproved in this section.

definition *T-bmssp* :: *nat* ⇒ *nat* ⇒ *nat* **where**
T-bmssp D N = *exact-concrete-top-level-time D N*

theorem *T-bmssp-refined-bound-log-params-fixed-degree*:
assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
and *degree*: *edge-outdegree-le D*
and *run-exists*: *exact-concrete-top-level-cost D N c*
shows *T-bmssp D N* ≤
bmssp-refined-graph-time-bound
(λ-. *Suc D* ∗ *sssp-log-one-third-param N*)
(λ-. *sssp-log-two-thirds-param N*)
(λ-. *sssp-log-one-third-param N*)
(λ-. *sssp-log-one-third-param N*)
(λ-. *sssp-log-two-thirds-param N*)
(λ-. *edge-count*) *vertex-count*
unfolding *T-bmssp-def*
by (*rule exact-concrete-top-level-time-refined-bound-log-params-fixed-degree*
[*OF all-reachable degree run-exists*])

theorem *T-bmssp-bigo-sssp-time-target-log-params-fixed-degree-if-exact-runs-exist*:
fixes *m* :: *nat* ⇒ *nat*
and *Cn Cm* :: *real*
assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
and *degree*: *edge-outdegree-le D*
and *Cn-pos*: *0* < *Cn*
and *Cm-pos*: *0* < *Cm*
and *vertex-count-dominated*:
eventually (λ*n*. *real vertex-count* ≤ *Cn* ∗ *real* (*m n*)) *at-top*
and *edge-count-dominated*:
eventually (λ*n*. *real edge-count* ≤ *Cm* ∗ *real* (*m n*)) *at-top*
and *exact-runs*:
eventually (λ*n*. ∃ *c*. *exact-concrete-top-level-cost D n c*) *at-top*
shows (λ*n*. *real* (*T-bmssp D n*)) ∈ *O*(λ*n*. *sssp-time-target m n*)
proof (*rule bmssp-refined-cost-bound-bigo-sssp-time-target-log-params-bounded-degree-slack*
[**where** *D* = *D* **and** *Cn* = *Cn* **and** *Cm* = *Cm*
and *v* = λ-. *vertex-count* **and** *m′* = λ-. *edge-count*])
show *0* < *Cn*

```
    by (rule Cn-pos)
  show 0 < Cm
    by (rule Cm-pos)
  show eventually (λn. real ((λ-. vertex-count) n) ≤
      Cn * real (m n)) at-top
    using vertex-count-dominated by simp
  show eventually (λn. real ((λ-. edge-count) n) ≤
      Cm * real (m n)) at-top
    using edge-count-dominated by simp
  show eventually
      (λn. T-bmssp D n ≤
        bmssp-refined-graph-time-bound
          (λ-. Suc D * sssp-log-one-third-param n)
          (λ-. sssp-log-two-thirds-param n)
          (λ-. sssp-log-one-third-param n)
          (λ-. sssp-log-one-third-param n)
          (λ-. sssp-log-two-thirds-param n)
          (λ-. (λ-. edge-count) n) ((λ-. vertex-count) n)) at-top
    using exact-runs
  proof eventually-elim
    case (elim n)
    then obtain c where cost: exact-concrete-top-level-cost D n c
      by blast
    show ?case
      by (rule T-bmssp-refined-bound-log-params-fixed-degree
        [OF all-reachable degree cost])
  qed
qed

theorem T-bmssp-bigo-sssp-time-target-square-size-params-fixed-degree-if-exact-runs-exist:
  fixes m :: nat ⇒ nat
    and Cn Cm :: real
  assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
    and degree: edge-outdegree-le D
    and Cn-pos: 0 < Cn
    and Cm-pos: 0 < Cm
    and vertex-count-dominated:
      eventually (λn. real vertex-count ≤ Cn * real (m n)) at-top
    and edge-count-dominated:
      eventually (λn. real edge-count ≤ Cm * real (m n)) at-top
    and exact-runs:
      eventually (λn. ∃ c. exact-concrete-top-level-cost D (n * n) c) at-top
  shows (λn. real (T-bmssp D (n * n))) ∈
    O(λn. sssp-time-target m n)
proof (rule bmssp-refined-cost-bound-bigo-sssp-time-target-log-params-bounded-degree-square-arg-slack
    [where D = D and Cn = Cn and Cm = Cm
      and v = λ-. vertex-count and m' = λ-. edge-count])
```

```
  show 0 < Cn
    by (rule Cn-pos)
  show 0 < Cm
    by (rule Cm-pos)
  show eventually (λn. real ((λ-. vertex-count) n) ≤
      Cn * real (m n)) at-top
    using vertex-count-dominated by simp
  show eventually (λn. real ((λ-. edge-count) n) ≤
      Cm * real (m n)) at-top
    using edge-count-dominated by simp
  show eventually
      (λn. T-bmssp D (n * n) ≤
        bmssp-refined-graph-time-bound
          (λ-. Suc D * sssp-log-one-third-param (n * n))
          (λ-. sssp-log-two-thirds-param (n * n))
          (λ-. sssp-log-one-third-param (n * n))
          (λ-. sssp-log-one-third-param (n * n))
          (λ-. sssp-log-two-thirds-param (n * n))
          (λ-. (λ-. edge-count) n) ((λ-. vertex-count) n)) at-top
    using exact-runs
  proof eventually-elim
    case (elim n)
    then obtain c where cost: exact-concrete-top-level-cost D (n * n) c
      by blast
    show ?case
      by (rule T-bmssp-refined-bound-log-params-fixed-degree
        [OF all-reachable degree cost])
  qed
qed

theorem T-bmssp-bigo-edge-count-target-log-params-fixed-degree-if-exact-runs-exist:
  assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
    and degree: edge-outdegree-le D
    and edge-count-pos: 0 < edge-count
    and exact-runs:
      eventually (λn. ∃ c. exact-concrete-top-level-cost D n c) at-top
  shows (λn. real (T-bmssp D n)) ∈
    O(λn. sssp-time-target (λ-. edge-count) n)
proof (rule T-bmssp-bigo-sssp-time-target-log-params-fixed-degree-if-exact-runs-
exist
    [OF all-reachable degree, where Cn = 2 and Cm = 1])
  show 0 < (2::real)
    by simp
  show 0 < (1::real)
    by simp
  have vertex-le: vertex-count ≤ 2 * edge-count
    by (rule vertex-count-le-twice-edge-count-if-all-reachable
      [OF all-reachable edge-count-pos])
  show eventually
```

```
    (λn. real vertex-count ≤ 2 * real ((λ-. edge-count) n)) at-top
   using vertex-le by simp
 show eventually
    (λn. real edge-count ≤ 1 * real ((λ-. edge-count) n)) at-top
   by simp
 show eventually (λn. ∃ c. exact-concrete-top-level-cost D n c) at-top
   by (rule exact-runs)
qed

theorem T-bmssp-bigo-edge-count-target-log-params-fixed-degree:
 assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
   and degree: edge-outdegree-le D
   and edge-count-pos: 0 < edge-count
 shows (λn. real (T-bmssp D n)) ∈
   O(λn. sssp-time-target (λ-. edge-count) n)
 by (rule T-bmssp-bigo-edge-count-target-log-params-fixed-degree-if-exact-runs-exist
   [OF all-reachable degree edge-count-pos
     eventually-exact-concrete-top-level-cost-from-empty-top-pivots])

theorem T-bmssp-bigo-size-target-log-params-fixed-degree:
 assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
   and degree: edge-outdegree-le D
 shows (λn. real (T-bmssp D n)) ∈
   O(λn. sssp-time-target (λ-. Suc edge-count) n)
proof (rule T-bmssp-bigo-sssp-time-target-log-params-fixed-degree-if-exact-runs-
exist
   [OF all-reachable degree, where Cn = 1 and Cm = 1])
 show 0 < (1::real)
   by simp
 show 0 < (1::real)
   by simp
 have vertex-le: vertex-count ≤ Suc edge-count
   by (rule vertex-count-le-Suc-edge-count-if-all-reachable[OF all-reachable])
 show eventually
    (λn. real vertex-count ≤ 1 * real ((λ-. Suc edge-count) n)) at-top
   using vertex-le by simp
 show eventually
    (λn. real edge-count ≤ 1 * real ((λ-. Suc edge-count) n)) at-top
   by simp
 show eventually (λn. ∃ c. exact-concrete-top-level-cost D n c) at-top
   by (rule eventually-exact-concrete-top-level-cost-from-empty-top-pivots)
qed

theorem T-bmssp-bigo-target-log-params-fixed-degree-if-exact-runs-exist:
 assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
   and degree: edge-outdegree-le D
   and edge-count-pos: 0 < edge-count
   and exact-runs:
     eventually (λn. ∃ c. exact-concrete-top-level-cost D n c) at-top
```

shows ($\lambda n.$ *real* (*T-bmssp D n*)) $\in$
$O(\lambda n.$ *real edge-count* $*$ (*ln* (*real n* + *2*)) *powr* (*2* / *3*))
using *T-bmssp-bigo-edge-count-target-log-params-fixed-degree-if-exact-runs-exist*
[*OF all-reachable degree edge-count-pos exact-runs*]
unfolding *sssp-time-target-def sssp-log-factor-def* **by** *simp*

theorem *T-bmssp-bigo-target-log-params-fixed-degree-if-root-runs-exist-below-threshold*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le D*
and *edge-count-pos*: *0* $<$ *edge-count*
and *below-threshold*:
eventually
($\lambda N.$ *vertex-count* $<$
sssp-log-one-third-param N $*$
bmssp-level-cap (*sssp-log-one-third-param N*)
(*sssp-log-two-thirds-param N*) (*sssp-log-one-third-param N*))
at-top
and *root-runs*:
eventually
($\lambda N.\ \exists\, d'\ B'\ U\ c.$ *exact-concrete-root-run D N d' B' U c*)
at-top
shows ($\lambda n.$ *real* (*T-bmssp D n*)) $\in$
$O(\lambda n.$ *real edge-count* $*$ (*ln* (*real n* + *2*)) *powr* (*2* / *3*))
proof (*rule T-bmssp-bigo-target-log-params-fixed-degree-if-exact-runs-exist*
[*OF all-reachable degree edge-count-pos*])
show *eventually* ($\lambda N.\ \exists\, c.$ *exact-concrete-top-level-cost D N c*) *at-top*
by (*rule eventually-exact-concrete-top-level-cost-if-root-runs-exist-below-threshold*
[*OF all-reachable below-threshold root-runs*])
qed

theorem *T-bmssp-bigo-target-log-params-fixed-degree-if-root-runs-exist*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le D*
and *edge-count-pos*: *0* $<$ *edge-count*
and *root-runs*:
eventually
($\lambda N.\ \exists\, d'\ B'\ U\ c.$ *exact-concrete-root-run D N d' B' U c*)
at-top
shows ($\lambda n.$ *real* (*T-bmssp D n*)) $\in$
$O(\lambda n.$ *real edge-count* $*$ (*ln* (*real n* + *2*)) *powr* (*2* / *3*))
proof (*rule T-bmssp-bigo-target-log-params-fixed-degree-if-root-runs-exist-below-threshold*
[*OF all-reachable degree edge-count-pos - root-runs*])
show *eventually*
($\lambda N.$ *vertex-count* $<$
sssp-log-one-third-param N $*$
bmssp-level-cap (*sssp-log-one-third-param N*)
(*sssp-log-two-thirds-param N*) (*sssp-log-one-third-param N*))

at-top
by (*rule eventually-top-level-threshold-exceeds-vertex-count*)
qed

theorem *T-bmssp-bigo-target-log-params-fixed-degree*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le D*
and *edge-count-pos*: *0* $<$ *edge-count*
shows $(\lambda n.$ *real* (*T-bmssp D n*)) $\in$
$O(\lambda n.$ *real edge-count* $*$ (*ln* (*real n* + *2*)) *powr* (*2* / *3*))
using *T-bmssp-bigo-edge-count-target-log-params-fixed-degree*
[*OF all-reachable degree edge-count-pos*]
unfolding *sssp-time-target-def sssp-log-factor-def* **by** *simp*

theorem *bmssp-runtime-bigo-target*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le D*
and *edge-count-pos*: *0* $<$ *edge-count*
shows $(\lambda n.$ *real* (*T-bmssp D n*)) $\in$
$O(\lambda n.$ *real edge-count* $*$ (*ln* (*real n* + *2*)) *powr* (*2* / *3*))
by (*rule T-bmssp-bigo-target-log-params-fixed-degree*
[*OF all-reachable degree edge-count-pos*])

theorem *T-bmssp-bigo-target-log-params-fixed-degree-if-total*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le D*
and *edge-count-pos*: *0* $<$ *edge-count*
and *total*: $\bigwedge N.\ \exists\, c.$ *exact-concrete-top-level-cost D N c*
shows $(\lambda n.$ *real* (*T-bmssp D n*)) $\in$
$O(\lambda n.$ *real edge-count* $*$ (*ln* (*real n* + *2*)) *powr* (*2* / *3*))
by (*rule T-bmssp-bigo-target-log-params-fixed-degree-if-exact-runs-exist*
[*OF all-reachable degree edge-count-pos*
eventually-exact-concrete-top-level-cost-if-all[*OF total*]])

theorem *T-bmssp-bigo-edge-count-target-square-size-params-fixed-degree-if-exact-runs-exist*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le D*
and *edge-count-pos*: *0* $<$ *edge-count*
and *exact-runs*:
eventually $(\lambda n.\ \exists\, c.$ *exact-concrete-top-level-cost D* $(n * n)$ *c*) *at-top*
shows $(\lambda n.$ *real* (*T-bmssp D* $(n * n)$)) $\in$
$O(\lambda n.$ *sssp-time-target* $(\lambda\text{-}.$ *edge-count*) *n*)
proof (*rule T-bmssp-bigo-sssp-time-target-square-size-params-fixed-degree-if-exact-runs-exist*
[*OF all-reachable degree*, **where** *Cn* = *2* **and** *Cm* = *1*])
show *0* $<$ (*2*::*real*)
by *simp*
show *0* $<$ (*1*::*real*)

by *simp*
have *vertex-le*: *vertex-count* $\leq$ *2* $*$ *edge-count*
by (*rule vertex-count-le-twice-edge-count-if-all-reachable*
[*OF all-reachable edge-count-pos*])
show *eventually*
(λn. *real vertex-count* $\leq$ *2* $*$ *real* ((*λ-. edge-count*) *n*)) *at-top*
using *vertex-le* **by** *simp*
show *eventually*
(λn. *real edge-count* $\leq$ *1* $*$ *real* ((*λ-. edge-count*) *n*)) *at-top*
by *simp*
show *eventually*
(λn. $\exists$ *c. exact-concrete-top-level-cost D* (*n* $*$ *n*) *c*) *at-top*
by (*rule exact-runs*)
qed

theorem *T-bmssp-bigo-target-square-size-params-fixed-degree-if-exact-runs-exist*:
assumes *all-reachable*: $\bigwedge v$. *v* $\in$ *V* $\Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le D*
and *edge-count-pos*: *0* $<$ *edge-count*
and *exact-runs*:
eventually (λn. $\exists$ *c. exact-concrete-top-level-cost D* (*n* $*$ *n*) *c*) *at-top*
shows (λn. *real* (*T-bmssp D* (*n* $*$ *n*))) $\in$
O(λn. *real edge-count* $*$ (*ln* (*real n* + *2*)) *powr* (*2* / *3*))
using *T-bmssp-bigo-edge-count-target-square-size-params-fixed-degree-if-exact-runs-exist*
[*OF all-reachable degree edge-count-pos exact-runs*]
unfolding *sssp-time-target-def sssp-log-factor-def* **by** *simp*

theorem *T-bmssp-bigo-target-square-size-params-fixed-degree-if-total*:
assumes *all-reachable*: $\bigwedge v$. *v* $\in$ *V* $\Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le D*
and *edge-count-pos*: *0* $<$ *edge-count*
and *total*: $\bigwedge N$. $\exists$ *c. exact-concrete-top-level-cost D N c*
shows (λn. *real* (*T-bmssp D* (*n* $*$ *n*))) $\in$
O(λn. *real edge-count* $*$ (*ln* (*real n* + *2*)) *powr* (*2* / *3*))
by (*rule T-bmssp-bigo-target-square-size-params-fixed-degree-if-exact-runs-exist*
[*OF all-reachable degree edge-count-pos*
eventually-exact-concrete-top-level-cost-square-if-all[*OF total*]])

theorem *range-split-algorithm-correct-and-closed-graph-bound*:
assumes *all-reachable*: $\bigwedge v$. *v* $\in$ *V* $\Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: *0* $<$ *R*
and *insert-factor*: *t* $\leq$ *A* $*$ *k*
and *insert-scaled-factor*: *t* $\leq$ *A-insert* $*$ *k*
and *seen-scaled-factor*: *k* $*$ Δ + *A-insert* $\leq$ *2* $*$ *A*
and *source-factor*: *Suc h* $\leq$ *2* $*$ *A*
and *k-pos*: *0* $<$ *k*

and *run*:
exact-split-range-costed-bmssp Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) *l*
finite-initial-label {*s*} *Infinity d' Infinity U c*
shows $U = V \wedge$ *sssp-correct d'* $\wedge$
$c \leq$ *bmssp-graph-time-bound* (λ-. *A*) (λ-. *R*) (λ-. *l*) (λ-. *t*)
(λ-. *edge-count*) *vertex-count*
proof −
have *UV-correct*: $U = V \wedge$ *sssp-correct d'*
by (*rule range-split-algorithm-correct*[*OF all-reachable run*])
have *cost*:
sssp-correct d' $\wedge$
$c \leq$ *bmssp-graph-time-bound* (λ-. *A*) (λ-. *R*) (λ-. *l*) (λ-. *t*)
(λ-. *edge-count*) *vertex-count*
by (*rule finite-initial-label-exact-split-range-costed-top-level-correct-and-closed-bmssp-graph-time-bound-level-cap*
[*OF all-reachable degree degree-factor R-pos insert-factor insert-scaled-factor seen-scaled-factor source-factor k-pos run*])
show *?thesis*
using *UV-correct cost* **by** *blast*
qed

theorem *range-split-algorithm-correct-and-refined-graph-bound-under-edge-budget*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *insert-scaled-factor*: $t \leq$ *A-insert* $* k$
and *seen-scaled-factor*: $k * \Delta +$ *A-insert* $\leq 2 * A$
and *source-factor*: *Suc h* $\leq 2 * A$
and *k-pos*: $0 < k$
and *edge-budget*:
$\bigwedge l'$ *d P B d' D a betas bs B' Us U-loop c-loop child-costs.*
exact-split-range-costed-partition-loop-state Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) *l' d P B d' D a betas bs B'*
Us U-loop c-loop child-costs $\Longrightarrow$
sound-label d $\Longrightarrow$
bmssp-pre-full d P B $\Longrightarrow$
($\bigwedge x.\ x \in P \Longrightarrow$ *reachable s x*) $\Longrightarrow$
t $*$ *sum-list* (*map card* (*range-tree-child-edge-range-list P a betas bs*))
$\leq H *$ *card* (*outgoing-edges U-loop*)
and *run*:
exact-split-range-costed-bmssp Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) *l*
finite-initial-label {*s*} *Infinity d' Infinity U c*
shows $U = V \wedge$ *sssp-correct d'* $\wedge$
$c \leq$ *bmssp-refined-graph-time-bound* (λ-. *A*) (λ-. *R*) (λ-. *H*)
(λ-. *l*) (λ-. *t*) (λ-. *edge-count*) *vertex-count*

proof −
have *UV-correct*: $U = V \wedge$ *sssp-correct* d'
by (*rule range-split-algorithm-correct*[*OF all-reachable run*])
have *cost*:
sssp-correct $d' \wedge$
$c \leq$ *bmssp-refined-graph-time-bound* (λ-. *A*) (λ-. *R*) (λ-. *H*)
(λ-. *l*) (λ-. *t*) (λ-. *edge-count*) *vertex-count*
by (*rule finite-initial-label-exact-split-range-costed-top-level-correct-and-closed-bmssp-refined-graph-time-bound-level-cap*
[*OF all-reachable degree degree-factor R-pos insert-factor insert-scaled-factor seen-scaled-factor source-factor k-pos edge-budget run*])
show *?thesis*
using *UV-correct cost* **by** *blast*
qed

theorem *range-split-algorithm-correct-and-refined-graph-bound-trivial-edge-budget*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le* Δ
and *degree-factor*: $\Delta \leq A$
and *R-pos*: $0 < R$
and *insert-factor*: $t \leq A * k$
and *insert-scaled-factor*: $t \leq A\text{-}insert * k$
and *seen-scaled-factor*: $k * \Delta + A\text{-}insert \leq 2 * A$
and *source-factor*: $Suc\ h \leq 2 * A$
and *k-pos*: $0 < k$
and *run*:
exact-split-range-costed-bmssp Δ (*bmssp-level-cap k t*) *t h k*
(*bmssp-level-cap k t l*) *l*
finite-initial-label {*s*} *Infinity d′ Infinity U c*
shows $U = V \wedge$ *sssp-correct* $d' \wedge$
$c \leq$ *bmssp-refined-graph-time-bound* (λ-. *A*) (λ-. *R*) (λ-. *t*)
(λ-. *l*) (λ-. *t*) (λ-. *edge-count*) *vertex-count*
proof −
have *UV-correct*: $U = V \wedge$ *sssp-correct* d'
by (*rule range-split-algorithm-correct*[*OF all-reachable run*])
have *cost*:
sssp-correct $d' \wedge$
$c \leq$ *bmssp-refined-graph-time-bound* (λ-. *A*) (λ-. *R*) (λ-. *t*)
(λ-. *l*) (λ-. *t*) (λ-. *edge-count*) *vertex-count*
by (*rule finite-initial-label-exact-split-range-costed-top-level-correct-and-closed-bmssp-refined-graph-time-bound-level-cap-trivial-edge-budget*
[*OF all-reachable degree degree-factor R-pos insert-factor insert-scaled-factor seen-scaled-factor source-factor k-pos run*])
show *?thesis*
using *UV-correct cost* **by** *blast*
qed

end

context *positive-unique-shortest-digraph*
begin

theorem *positive-weight-direct-insert-algorithm-correct-and-refined-graph-bound-log-params-fixed-degree*:
fixes *N D p q* :: *nat*
defines *p-def*: $p \equiv$ *sssp-log-one-third-param N*
and *q-def*: $q \equiv$ *sssp-log-two-thirds-param N*
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le D*
and *run*:
direct-insert-costed-bmssp D (*bmssp-level-cap p q*) *q p p*
(*bmssp-level-cap p q p*) *p*
finite-initial-label {*s*} *Infinity d′ Infinity U c*
shows $U = V \wedge$ *sssp-correct d′* $\wedge$
$c \leq$ *bmssp-refined-graph-time-bound*
(*λ-. Suc D* $*$ *p*) (*λ-. q*) (*λ-. p*)
(*λ-. p*) (*λ-. q*) (*λ-. edge-count*) *vertex-count*
proof −
have *run′*:
direct-insert-costed-bmssp D
(*bmssp-level-cap* (*sssp-log-one-third-param N*) (*sssp-log-two-thirds-param N*))
(*sssp-log-two-thirds-param N*) (*sssp-log-one-third-param N*)
(*sssp-log-one-third-param N*)
(*bmssp-level-cap* (*sssp-log-one-third-param N*) (*sssp-log-two-thirds-param N*)
(*sssp-log-one-third-param N*))
(*sssp-log-one-third-param N*)
finite-initial-label {*s*} *Infinity d′ Infinity U c*
using *run* **unfolding** *p-def q-def* **by** *simp*
show *?thesis*
unfolding *p-def q-def*
by (*rule direct-insert-algorithm-correct-and-refined-graph-bound-log-params-fixed-degree*
[*OF all-reachable degree run′*])
qed

theorem *positive-weight-exact-concrete-algorithm-correct-and-refined-graph-bound-log-params-fixed-degree*:
fixes *N D p q* :: *nat*
defines *p-def*: $p \equiv$ *sssp-log-one-third-param N*
and *q-def*: $q \equiv$ *sssp-log-two-thirds-param N*
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le D*
and *run*:
exact-concrete-bmssp D (*bmssp-level-cap p q*) *q p p*
(*bmssp-level-cap p q p*) *p*
finite-initial-label {*s*} *Infinity d′ Infinity U c*
shows $U = V \wedge$ *sssp-correct d′* $\wedge$
$c \leq$ *bmssp-refined-graph-time-bound*

```
    (λ-. Suc D * p) (λ-. q) (λ-. p)
    (λ-. p) (λ-. q) (λ-. edge-count) vertex-count
proof −
  have direct-run:
    direct-insert-costed-bmssp D (bmssp-level-cap p q) q p p
      (bmssp-level-cap p q p) p
      finite-initial-label {s} Infinity d' Infinity U c
    by (rule exact-concrete-bmssp-refines-direct-insert[OF run])
  have direct-run':
    direct-insert-costed-bmssp D
      (bmssp-level-cap (sssp-log-one-third-param N) (sssp-log-two-thirds-param N))
      (sssp-log-two-thirds-param N) (sssp-log-one-third-param N)
      (sssp-log-one-third-param N)
      (bmssp-level-cap (sssp-log-one-third-param N) (sssp-log-two-thirds-param N)
        (sssp-log-one-third-param N))
      (sssp-log-one-third-param N)
      finite-initial-label {s} Infinity d' Infinity U c
    using direct-run unfolding p-def q-def by simp
  show ?thesis
    unfolding p-def q-def
    by (rule positive-weight-direct-insert-algorithm-correct-and-refined-graph-bound-
log-params-fixed-degree
      [OF all-reachable degree direct-run'])
qed

end

locale bounded-reduced-positive-instance = positive-unique-shortest-digraph +
  fixes D :: nat
  assumes all-vertices-reachable: ⋀v. v ∈ V ⟹ reachable s v
    and bounded-edge-outdegree: edge-outdegree-le D
begin

theorem bounded-reduced-direct-insert-algorithm-correct-and-refined-graph-bound-
log-params:
  fixes N p q :: nat
  defines p-def: p ≡ sssp-log-one-third-param N
    and q-def: q ≡ sssp-log-two-thirds-param N
  assumes run:
      direct-insert-costed-bmssp D (bmssp-level-cap p q) q p p
        (bmssp-level-cap p q p) p
        finite-initial-label {s} Infinity d' Infinity U c
  shows U = V ∧ sssp-correct d' ∧
    c ≤ bmssp-refined-graph-time-bound
      (λ-. Suc D * p) (λ-. q) (λ-. p)
      (λ-. p) (λ-. q) (λ-. edge-count) vertex-count
proof −
  have run':
    direct-insert-costed-bmssp D
```

```
    (bmssp-level-cap (sssp-log-one-third-param N) (sssp-log-two-thirds-param N))
    (sssp-log-two-thirds-param N) (sssp-log-one-third-param N)
    (sssp-log-one-third-param N)
    (bmssp-level-cap (sssp-log-one-third-param N) (sssp-log-two-thirds-param N)
      (sssp-log-one-third-param N))
    (sssp-log-one-third-param N)
    finite-initial-label {s} Infinity d' Infinity U c
  using run unfolding p-def q-def by simp
 show ?thesis
  unfolding p-def q-def
  by (rule positive-weight-direct-insert-algorithm-correct-and-refined-graph-bound-
log-params-fixed-degree
    [OF all-vertices-reachable bounded-edge-outdegree run'])
qed

theorem bounded-reduced-exact-concrete-algorithm-correct-and-refined-graph-bound-
log-params:
 fixes N p q :: nat
 defines p-def: p ≡ sssp-log-one-third-param N
  and q-def: q ≡ sssp-log-two-thirds-param N
 assumes run:
    exact-concrete-bmssp D (bmssp-level-cap p q) q p p
      (bmssp-level-cap p q p) p
      finite-initial-label {s} Infinity d' Infinity U c
 shows U = V ∧ sssp-correct d' ∧
  c ≤ bmssp-refined-graph-time-bound
    (λ-. Suc D * p) (λ-. q) (λ-. p)
    (λ-. p) (λ-. q) (λ-. edge-count) vertex-count
proof −
 have direct-run:
  direct-insert-costed-bmssp D (bmssp-level-cap p q) q p p
    (bmssp-level-cap p q p) p
    finite-initial-label {s} Infinity d' Infinity U c
  by (rule exact-concrete-bmssp-refines-direct-insert[OF run])
 have direct-run':
  direct-insert-costed-bmssp D
    (bmssp-level-cap (sssp-log-one-third-param N) (sssp-log-two-thirds-param N))
    (sssp-log-two-thirds-param N) (sssp-log-one-third-param N)
    (sssp-log-one-third-param N)
    (bmssp-level-cap (sssp-log-one-third-param N) (sssp-log-two-thirds-param N)
      (sssp-log-one-third-param N))
    (sssp-log-one-third-param N)
    finite-initial-label {s} Infinity d' Infinity U c
  using direct-run unfolding p-def q-def by simp
 show ?thesis
  unfolding p-def q-def
  by (rule bounded-reduced-direct-insert-algorithm-correct-and-refined-graph-bound-
log-params
    [OF direct-run'])
```

qed

theorem *bounded-reduced-bmssp-runtime-bigo-target*:
assumes *edge-count-pos*: *0 < edge-count*
shows (*λn. real* (*T-bmssp D n*)) ∈
O(*λn. real edge-count* ∗ (*ln* (*real n* + *2*)) *powr* (*2* / *3*))
by (*rule T-bmssp-bigo-target-log-params-fixed-degree*
[*OF all-vertices-reachable bounded-edge-outdegree edge-count-pos*])

end

locale *bmssp-runtime-headline-instance* =
bounded: *bounded-reduced-positive-instance V E w s D*
for *V* :: *'v set*
and *E* :: (*'v* × *'v*) *set*
and *w* :: *'v* ⇒ *'v* ⇒ *real*
and *s* :: *'v*
and *D* :: *nat*
begin

This locale is the public asymptotic headline for the top-level theory. It hides the reduced positive instance, the fixed outdegree bound, and the least-cost definition behind two simple functions, *T-bmssp* for the checked BMSSP running time and *m* for the edge-count scale. The theorem below is deliberately assumption-free inside the locale: all graph hypotheses are already part of the locale interpretation.

definition *T-bmssp* :: *nat* ⇒ *nat* **where**
T-bmssp N = *bounded.T-bmssp D N*

definition *m* :: *nat* ⇒ *nat* **where**
m N = *Suc bounded.edge-count*

theorem *bmssp-runtime-bigo-target*:
shows (*λn. real* (*T-bmssp n*)) ∈
O(*λn. real* (*m n*) ∗ (*ln* (*real n* + *2*)) *powr* (*2* / *3*))
using *bounded.T-bmssp-bigo-size-target-log-params-fixed-degree*
[*OF bounded.all-vertices-reachable bounded.bounded-edge-outdegree*]
unfolding *T-bmssp-def m-def sssp-time-target-def sssp-log-factor-def* **by** *simp*

The theorem (*λn. real* (*T-bmssp n*)) ∈ *O*(*λn. real* (*m n*) ∗ *ln* (*real n* + *2*) *powr* (*2* / *3*)) is the headline statement used by the rest of the AFP entry. Its right-hand side is the deterministic BMSSP target: graph-size work times *log*^{*2*/*3*}. The proof is short here because the recurrence has already been solved in the preceding fixed-degree theorem and the locale definitions only rename the time and size functions.

end

sublocale *bounded-reduced-positive-instance* <

runtime-headline: *bmssp-runtime-headline-instance V E w s D*
by *unfold-locales*

context *bounded-reduced-positive-instance*
begin

theorem *bounded-reduced-runtime-headline-bmssp-runtime-bigo-target*:
shows $(\lambda n.\ real\ (runtime\text{-}headline.T\text{-}bmssp\ n)) \in$
$O(\lambda n.\ real\ (runtime\text{-}headline.m\ n) * (ln\ (real\ n + 2))\ powr\ (2\ /\ 3))$
by (*rule runtime-headline.bmssp-runtime-bigo-target*)

end

locale *reduction-correctness-contract* =
original: *finite-weighted-digraph V E w s* +
reduced: *bounded-reduced-positive-instance V' E' w' s' D*
for $V :: 'v\ set$
and $E :: ('v \times 'v)\ set$
and $w :: 'v \Rightarrow 'v \Rightarrow real$
and $s :: 'v$
and $V' :: 'r\ set$
and $E' :: ('r \times 'r)\ set$
and $w' :: 'r \Rightarrow 'r \Rightarrow real$
and $s' :: 'r$
and $D :: nat$ +
fixes *decode-label* $:: ('r \Rightarrow real) \Rightarrow 'v \Rightarrow real$
assumes *transfer-sssp-correct*:
reduced.sssp-correct d' $\Longrightarrow$ *original.sssp-correct* (*decode-label d'*)
begin

theorem *reduced-direct-insert-run-transfers-correctness-and-refined-bound*:
fixes *N p q* :: *nat*
defines *p-def*: $p \equiv$ *sssp-log-one-third-param N*
and *q-def*: $q \equiv$ *sssp-log-two-thirds-param N*
assumes *run*:
reduced.direct-insert-costed-bmssp D (*bmssp-level-cap p q*) *q p p*
(*bmssp-level-cap p q p*) *p*
reduced.finite-initial-label $\{s'\}$ *Infinity d' Infinity U c*
shows *original.sssp-correct* (*decode-label d'*) $\wedge$
$c \leq$ *bmssp-refined-graph-time-bound*
$(\lambda\text{-}.\ Suc\ D * p)$ $(\lambda\text{-}.\ q)$ $(\lambda\text{-}.\ p)$
$(\lambda\text{-}.\ p)$ $(\lambda\text{-}.\ q)$ $(\lambda\text{-}.\ reduced.edge\text{-}count)$
reduced.vertex-count
proof −
have *run'*:
reduced.direct-insert-costed-bmssp D
(*bmssp-level-cap* (*sssp-log-one-third-param N*) (*sssp-log-two-thirds-param N*))
(*sssp-log-two-thirds-param N*) (*sssp-log-one-third-param N*)
(*sssp-log-one-third-param N*)

```
    (bmssp-level-cap (sssp-log-one-third-param N) (sssp-log-two-thirds-param N)
      (sssp-log-one-third-param N))
    (sssp-log-one-third-param N)
    reduced.finite-initial-label {s'} Infinity d' Infinity U c
  using run unfolding p-def q-def by simp
 have reduced-result:
  U = V' ∧ reduced.sssp-correct d' ∧
    c ≤ bmssp-refined-graph-time-bound
      (λ-. Suc D * sssp-log-one-third-param N)
      (λ-. sssp-log-two-thirds-param N)
      (λ-. sssp-log-one-third-param N)
      (λ-. sssp-log-one-third-param N)
      (λ-. sssp-log-two-thirds-param N)
      (λ-. reduced.edge-count) reduced.vertex-count
   by (rule reduced.bounded-reduced-direct-insert-algorithm-correct-and-refined-
graph-bound-log-params
    [OF run'])
 then show ?thesis
  unfolding p-def q-def
  using transfer-sssp-correct by blast
qed

theorem reduced-exact-concrete-run-transfers-correctness-and-refined-bound:
 fixes N p q :: nat
 defines p-def: p ≡ sssp-log-one-third-param N
  and q-def: q ≡ sssp-log-two-thirds-param N
 assumes run:
    reduced.exact-concrete-bmssp D (bmssp-level-cap p q) q p p
      (bmssp-level-cap p q p) p
      reduced.finite-initial-label {s'} Infinity d' Infinity U c
 shows original.sssp-correct (decode-label d') ∧
  c ≤ bmssp-refined-graph-time-bound
    (λ-. Suc D * p) (λ-. q) (λ-. p)
    (λ-. p) (λ-. q) (λ-. reduced.edge-count)
    reduced.vertex-count
proof −
 have direct-run:
  reduced.direct-insert-costed-bmssp D (bmssp-level-cap p q) q p p
    (bmssp-level-cap p q p) p
    reduced.finite-initial-label {s'} Infinity d' Infinity U c
  by (rule reduced.exact-concrete-bmssp-refines-direct-insert[OF run])
 have direct-run':
  reduced.direct-insert-costed-bmssp D
    (bmssp-level-cap (sssp-log-one-third-param N) (sssp-log-two-thirds-param N))
    (sssp-log-two-thirds-param N) (sssp-log-one-third-param N)
    (sssp-log-one-third-param N)
    (bmssp-level-cap (sssp-log-one-third-param N) (sssp-log-two-thirds-param N)
      (sssp-log-one-third-param N))
    (sssp-log-one-third-param N)
```

```
    reduced.finite-initial-label {s'} Infinity d' Infinity U c
   using direct-run unfolding p-def q-def by simp
  show ?thesis
   unfolding p-def q-def
   by (rule reduced-direct-insert-run-transfers-correctness-and-refined-bound
    [OF direct-run'])
qed

theorem reduced-exact-concrete-top-level-time-transfers-correctness-and-refined-bound:
  assumes run-exists: reduced.exact-concrete-top-level-cost D N c
  shows ∃ d' U. reduced.exact-concrete-top-level-run D N d' U
     (reduced.exact-concrete-top-level-time D N) ∧
    original.sssp-correct (decode-label d') ∧
    reduced.exact-concrete-top-level-time D N ≤
     bmssp-refined-graph-time-bound
      (λ-. Suc D * sssp-log-one-third-param N)
      (λ-. sssp-log-two-thirds-param N)
      (λ-. sssp-log-one-third-param N)
      (λ-. sssp-log-one-third-param N)
      (λ-. sssp-log-two-thirds-param N)
      (λ-. reduced.edge-count) reduced.vertex-count
proof −
  have cost:
   reduced.exact-concrete-top-level-cost D N
    (reduced.exact-concrete-top-level-time D N)
   by (rule reduced.exact-concrete-top-level-time-witness[OF run-exists])
  obtain d' U where run:
   reduced.exact-concrete-top-level-run D N d' U
    (reduced.exact-concrete-top-level-time D N)
   using cost unfolding reduced.exact-concrete-top-level-cost-def by blast
  have reduced-result:
   U = V' ∧ reduced.sssp-correct d' ∧
    reduced.exact-concrete-top-level-time D N ≤
     bmssp-refined-graph-time-bound
      (λ-. Suc D * sssp-log-one-third-param N)
      (λ-. sssp-log-two-thirds-param N)
      (λ-. sssp-log-one-third-param N)
      (λ-. sssp-log-one-third-param N)
      (λ-. sssp-log-two-thirds-param N)
      (λ-. reduced.edge-count) reduced.vertex-count
    by (rule reduced.exact-concrete-top-level-run-correct-and-refined-bound-log-params-fixed-degree
     [OF reduced.all-vertices-reachable reduced.bounded-edge-outdegree run])
  have original-correct: original.sssp-correct (decode-label d')
   by (rule transfer-sssp-correct) (use reduced-result in blast)
  show ?thesis
   using run reduced-result original-correct by blast
qed
```

theorem *reduced-runtime-headline-bmssp-runtime-bigo-target*:
shows $(\lambda n.\ real\ (reduced.runtime\text{-}headline.T\text{-}bmssp\ n)) \in$
$O(\lambda n.\ real\ (reduced.runtime\text{-}headline.m\ n) *$
$(ln\ (real\ n + 2))\ powr\ (2 / 3))$
by (*rule reduced.runtime-headline.bmssp-runtime-bigo-target*)

end

37 Reduction Contract for the Asymptotic Bound

The theorem below is the checked asymptotic contract needed from the graph reduction layer. Once a family of reduced instances supplies a fixed outdegree bound D, a linear edge-count domination for the vertex term, and the per-instance cost bound produced by the locale theorem above, the final SSSP target bound follows.

theorem *reduction-contract-bigo-sssp-time-target*:
fixes $D :: nat$
assumes *Cn-pos*: $0 < Cn$
and *vertex-count-dominated*:
eventually $(\lambda n.\ real\ n \le Cn * real\ (m\ n))$ *at-top*
and *reduced-instance-cost-bound*:
eventually
$(\lambda n.\ T\ n \le$
bmssp-refined-graph-time-bound
$(\lambda n.\ Suc\ D * sssp\text{-}log\text{-}one\text{-}third\text{-}param\ n)$
sssp-log-two-thirds-param sssp-log-one-third-param
sssp-log-one-third-param sssp-log-two-thirds-param m n) *at-top*
shows $(\lambda n.\ real\ (T\ n)) \in O(\lambda n.\ sssp\text{-}time\text{-}target\ m\ n)$
by (*rule bmssp-refined-cost-bound-bigo-sssp-time-target-log-params-bounded-degree*
[*OF Cn-pos vertex-count-dominated reduced-instance-cost-bound*])

theorem *reduction-contract-bigo-sssp-time-target-with-constant-slack*:
fixes $D :: nat$
assumes *Cn-pos*: $0 < Cn$
and *Ccost-pos*: $0 < Ccost$
and *vertex-count-dominated*:
eventually $(\lambda n.\ real\ n \le Cn * real\ (m\ n))$ *at-top*
and *reduced-instance-cost-bound*:
eventually
$(\lambda n.\ real\ (T\ n) \le Ccost *$
real (*bmssp-refined-graph-time-bound*
$(\lambda n.\ Suc\ D * sssp\text{-}log\text{-}one\text{-}third\text{-}param\ n)$
sssp-log-two-thirds-param sssp-log-one-third-param
sssp-log-one-third-param sssp-log-two-thirds-param m n)) *at-top*
shows $(\lambda n.\ real\ (T\ n)) \in O(\lambda n.\ sssp\text{-}time\text{-}target\ m\ n)$
by (*rule bmssp-refined-cost-bound-bigo-sssp-time-target-log-params-bounded-degree-real-slack*

[*OF Cn-pos Ccost-pos vertex-count-dominated reduced-instance-cost-bound*])

theorem *reduction-contract-bigo-sssp-time-target-square-size-params*:
fixes *D* :: *nat*
assumes *Cn-pos*: *0* < *Cn*
and *vertex-count-dominated*:
eventually (λ*n*. *real n* ≤ *Cn* ∗ *real* (*m n*)) *at-top*
and *reduced-instance-cost-bound*:
eventually
(λ*n*. *T n* ≤
bmssp-refined-graph-time-bound
(λ*n*. *Suc D* ∗ *sssp-log-one-third-param* (*n* ∗ *n*))
(λ*n*. *sssp-log-two-thirds-param* (*n* ∗ *n*))
(λ*n*. *sssp-log-one-third-param* (*n* ∗ *n*))
(λ*n*. *sssp-log-one-third-param* (*n* ∗ *n*))
(λ*n*. *sssp-log-two-thirds-param* (*n* ∗ *n*)) *m n*) *at-top*
shows (λ*n*. *real* (*T n*)) ∈ *O*(λ*n*. *sssp-time-target m n*)
by (*rule bmssp-refined-cost-bound-bigo-sssp-time-target-log-params-bounded-degree-square-arg*
[*OF Cn-pos vertex-count-dominated reduced-instance-cost-bound*])

theorem *reduction-contract-bigo-sssp-time-target-square-size-params-with-constant-slack*:
fixes *D* :: *nat*
assumes *Cn-pos*: *0* < *Cn*
and *Ccost-pos*: *0* < *Ccost*
and *vertex-count-dominated*:
eventually (λ*n*. *real n* ≤ *Cn* ∗ *real* (*m n*)) *at-top*
and *reduced-instance-cost-bound*:
eventually
(λ*n*. *real* (*T n*) ≤ *Ccost* ∗
real (*bmssp-refined-graph-time-bound*
(λ*n*. *Suc D* ∗ *sssp-log-one-third-param* (*n* ∗ *n*))
(λ*n*. *sssp-log-two-thirds-param* (*n* ∗ *n*))
(λ*n*. *sssp-log-one-third-param* (*n* ∗ *n*))
(λ*n*. *sssp-log-one-third-param* (*n* ∗ *n*))
(λ*n*. *sssp-log-two-thirds-param* (*n* ∗ *n*)) *m n*)) *at-top*
shows (λ*n*. *real* (*T n*)) ∈ *O*(λ*n*. *sssp-time-target m n*)
by (*rule bmssp-refined-cost-bound-bigo-sssp-time-target-log-params-bounded-degree-square-arg-real-slack*
[*OF Cn-pos Ccost-pos vertex-count-dominated reduced-instance-cost-bound*])

end
theory *BMSSP-Executable-Refinement-Internal*
imports *BMSSP-Executable-Base-Case BMSSP-Top-Level-Bounds*
begin

38 Executable Graph Refinement Bridge

This theory starts the refinement bridge between the concrete executable entry point *bmssp-distances* and the locale-based BMSSP correctness stack. The executable graph is a list of natural-number edge triples; the abstract semantics expects a finite weighted digraph locale. The definitions below name the carrier, edge set, and weight function induced by the list representation, then package the assumptions needed before the executable loop can be related to the direct-insert BMSSP relation.

fun *nat-edge-source* :: *nat-edge* ⇒ *nat* **where**
  nat-edge-source (*u*, *v*, *w*) = *u*

fun *nat-edge-target* :: *nat-edge* ⇒ *nat* **where**
  nat-edge-target (*u*, *v*, *w*) = *v*

fun *nat-edge-weight* :: *nat-edge* ⇒ *nat* **where**
  nat-edge-weight (*u*, *v*, *w*) = *w*

definition *nat-graph-edge-list* :: *nat-graph* ⇒ (*nat* × *nat*) *list* **where**
  nat-graph-edge-list G = *map* (λ*e*. (*nat-edge-source e*, *nat-edge-target e*)) *G*

definition *nat-graph-edges* :: *nat-graph* ⇒ (*nat* × *nat*) *set* **where**
  nat-graph-edges G = *set* (*nat-graph-edge-list G*)

definition *nat-graph-vertices* :: *nat-graph* ⇒ *nat set* **where**
  nat-graph-vertices G = *set* (*map nat-edge-source G*) ∪ *set* (*map nat-edge-target G*)

definition *nat-graph-vertex-list* :: *nat-graph* ⇒ *nat list* **where**
  nat-graph-vertex-list G =
    sort (*remdups* (*map nat-edge-source G* @ *map nat-edge-target G*))

definition *nat-graph-weight* :: *nat-graph* ⇒ *nat* ⇒ *nat* ⇒ *real* **where**
  nat-graph-weight G u v =
    (*case map-of*
      (*map* (λ*e*. ((*nat-edge-source e*, *nat-edge-target e*),
          real (*nat-edge-weight e*))) *G*) (*u*, *v*) *of*
      None ⇒ *0*
    | *Some c* ⇒ *c*)

definition *nat-graph-total-weight* :: *nat-graph* ⇒ *nat* **where**
  nat-graph-total-weight G = *sum-list* (*map nat-edge-weight G*)

definition *nat-graph-well-formed* :: *nat-graph* ⇒ *bool* **where**
  nat-graph-well-formed G ⟷
    distinct (*nat-graph-edge-list G*) ∧
    nat-graph-total-weight G < *bmssp-infinity*

lemma *nat-graph-well-formed-distinct-edge-list*:
assumes *nat-graph-well-formed G*
shows *distinct* (*nat-graph-edge-list G*)
using *assms* **unfolding** *nat-graph-well-formed-def* **by** *blast*

definition *nat-graph-reachable* :: *nat-graph* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
nat-graph-reachable G a b ⟷
finite-weighted-digraph.reachable
(*nat-graph-vertices G*) (*nat-graph-edges G*) *a b*

definition *nat-graph-dist* :: *nat-graph* ⇒ *nat* ⇒ *nat* ⇒ *real* **where**
nat-graph-dist G a b =
finite-weighted-digraph.dist
(*nat-graph-vertices G*) (*nat-graph-edges G*) (*nat-graph-weight G*) *a b*

definition *executable-label-of* :: *nat-dist* ⇒ *nat* ⇒ *real* **where**
executable-label-of ds v =
(*case bmssp-lookup-dist ds v of*
None ⇒ *bmssp-bound*
| *Some d* ⇒ *real d*)

lemma *bmssp-lookup-dist-mem-key*:
assumes *bmssp-lookup-dist ds v* = *Some d*
shows *v* ∈ *set* (*map fst ds*)
using *assms*
proof (*induction ds*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons p ds*)
obtain *x e* **where** *p*: *p* = (*x*, *e*)
by (*cases p*)
show *?case*
proof (*cases v* = *x*)
case *True*
then show *?thesis*
using *p* **by** *simp*
next
case *False*
then have *bmssp-lookup-dist ds v* = *Some d*
using *Cons.prems p* **by** *simp*
then have *v* ∈ *set* (*map fst ds*)
by (*rule Cons.IH*)
then show *?thesis*
using *p* **by** *simp*
qed
qed

```
lemma bmssp-lookup-dist-Some-pair-mem:
  assumes bmssp-lookup-dist ds v = Some d
  shows (v, d) ∈ set ds
  using assms
proof (induction ds)
  case Nil
  then show ?case
    by simp
next
  case (Cons p ds)
  obtain x e where p: p = (x, e)
    by (cases p)
  show ?case
  proof (cases v = x)
    case True
    then show ?thesis
      using Cons.prems unfolding p by simp
  next
    case False
    then have bmssp-lookup-dist ds v = Some d
      using Cons.prems unfolding p by simp
    then have (v, d) ∈ set ds
      by (rule Cons.IH)
    then show ?thesis
      unfolding p by simp
  qed
qed

lemma bmssp-lookup-dist-Some-if-distinct-mem:
  assumes distinct: distinct (map fst ds)
    and mem: (v, d) ∈ set ds
  shows bmssp-lookup-dist ds v = Some d
  using distinct mem
proof (induction ds)
  case Nil
  then show ?case
    by simp
next
  case (Cons p ds)
  obtain x e where p: p = (x, e)
    by (cases p)
  show ?case
  proof (cases v = x)
    case True
    have d-eq: d = e
    proof −
      have mem-cases: (v, d) = p ∨ (v, d) ∈ set ds
        using Cons.prems p by simp
      then show ?thesis
```

```
      proof
        assume (v, d) = p
        then show ?thesis
          using True p by simp
      next
        assume (v, d) ∈ set ds
        then have x ∈ set (map fst ds)
          using True by force
        then show ?thesis
          using Cons.prems p by simp
      qed
    qed
    then show ?thesis
      using True p by simp
  next
    case False
    then have mem-tail: (v, d) ∈ set ds
      using Cons.prems p by auto
    have distinct-tail: distinct (map fst ds)
      using Cons.prems p by simp
    show ?thesis
      using False p Cons.IH[OF distinct-tail mem-tail] by simp
  qed
qed

lemma bmssp-lookup-dist-None-if-distinct-not-mem:
  assumes distinct: distinct (map fst ds)
    and not-mem: v ∉ set (map fst ds)
  shows bmssp-lookup-dist ds v = None
  using distinct not-mem by (induction ds) auto

lemma bmssp-set-dist-keys:
  set (map fst (bmssp-set-dist v d ds)) = insert v (set (map fst ds))
proof (induction ds)
  case Nil
  then show ?case
    by simp
next
  case (Cons p ds)
  obtain x e where p: p = (x, e)
    by (cases p)
  show ?case
  proof (cases v = x)
    case True
    then show ?thesis
      unfolding p by simp
  next
    case False
    then show ?thesis
```

```
    using Cons.IH unfolding p by auto
  qed
qed

lemma bmssp-set-dist-fst-image [simp]:
  fst ‘ set (bmssp-set-dist v d ds) = insert v (fst ‘ set ds)
  using bmssp-set-dist-keys[of v d ds] by simp

lemma bmssp-set-dist-preserves-distinct:
  assumes distinct (map fst ds)
  shows distinct (map fst (bmssp-set-dist v d ds))
  using assms
proof (induction ds)
  case Nil
  then show ?case
    by simp
next
  case (Cons p ds)
  obtain x e where p: p = (x, e)
    by (cases p)
  have distinct-tail: distinct (map fst ds)
    using Cons.prems p by simp
  show ?case
  proof (cases v = x)
    case True
    then show ?thesis
      using Cons.prems p by simp
  next
    case False
    have x-not-tail: x ∉ set (map fst ds)
      using Cons.prems p by simp
    have x-not-updated: x ∉ set (map fst (bmssp-set-dist v d ds))
      using x-not-tail False unfolding bmssp-set-dist-keys by simp
    have updated-distinct: distinct (map fst (bmssp-set-dist v d ds))
      by (rule Cons.IH[OF distinct-tail])
    show ?thesis
      using False p x-not-updated updated-distinct by simp
  qed
qed

lemma bmssp-lookup-dist-set-dist-other:
  assumes v ≠ x
  shows bmssp-lookup-dist (bmssp-set-dist x dx ds) v =
    bmssp-lookup-dist ds v
  using assms
proof (induction ds)
  case Nil
  then show ?case
    by simp
```

```
next
  case (Cons p ds)
  obtain y e where p: p = (y, e)
    by (cases p)
  then show ?case
    using Cons by auto
qed

lemma bmssp-lookup-dist-set-dist-same:
  bmssp-lookup-dist (bmssp-set-dist x dx ds) x = Some dx
  by (induction ds) auto

lemma bmssp-normalize-dist-key-set [simp]:
  set (map fst (bmssp-normalize-dist ds)) = set (map fst ds)
  unfolding bmssp-normalize-dist-def by (metis set-map set-sort)

lemma bmssp-normalize-dist-fst-image [simp]:
  fst ‘ set (bmssp-normalize-dist ds) = fst ‘ set ds
  unfolding bmssp-normalize-dist-def by simp

lemma bmssp-lookup-dist-None-iff-not-key:
  assumes distinct: distinct (map fst ds)
  shows bmssp-lookup-dist ds v = None ⟷ v ∉ set (map fst ds)
proof
  assume none: bmssp-lookup-dist ds v = None
  show v ∉ set (map fst ds)
  proof
    assume v ∈ set (map fst ds)
    then obtain d where (v, d) ∈ set ds
      by force
    then have bmssp-lookup-dist ds v = Some d
      by (rule bmssp-lookup-dist-Some-if-distinct-mem[OF distinct])
    then show False
      using none by simp
  qed
next
  assume v ∉ set (map fst ds)
  then show bmssp-lookup-dist ds v = None
    by (rule bmssp-lookup-dist-None-if-distinct-not-mem[OF distinct])
qed

lemma bmssp-lookup-dist-normalize-dist:
  assumes distinct: distinct (map fst ds)
  shows bmssp-lookup-dist (bmssp-normalize-dist ds) v =
    bmssp-lookup-dist ds v
proof (cases bmssp-lookup-dist ds v)
  case None
  have distinct-norm:
    distinct (map fst (bmssp-normalize-dist ds))
```

```
    using distinct unfolding bmssp-normalize-dist-def
    by (rule distinct-map-fst-sort-key)
  have v ∉ set (map fst (bmssp-normalize-dist ds))
    using None distinct
    unfolding bmssp-lookup-dist-None-iff-not-key[OF distinct] by simp
  then show ?thesis
    using None bmssp-lookup-dist-None-if-distinct-not-mem[OF distinct-norm]
    by simp
next
  case (Some d)
  have mem: (v, d) ∈ set (bmssp-normalize-dist ds)
    using bmssp-lookup-dist-Some-pair-mem[OF Some]
    unfolding bmssp-normalize-dist-def by simp
  have distinct-norm:
    distinct (map fst (bmssp-normalize-dist ds))
    using distinct unfolding bmssp-normalize-dist-def
    by (rule distinct-map-fst-sort-key)
  have bmssp-lookup-dist (bmssp-normalize-dist ds) v = Some d
    by (rule bmssp-lookup-dist-Some-if-distinct-mem[OF distinct-norm mem])
  then show ?thesis
    using Some by simp
qed

lemma bmssp-relax-edges-preserves-distinct-dist:
  assumes distinct: distinct (map fst ds)
  shows distinct (map fst (snd (bmssp-relax-edges G settled u du ds)))
  using distinct
proof (induction G arbitrary: ds)
  case Nil
  then show ?case
    by simp
next
  case (Cons e es)
  obtain a b w where e: e = (a, b, w)
    by (cases e)
  obtain updates ds1 where rec:
    bmssp-relax-edges es settled u du ds = (updates, ds1)
    by force
  have distinct-ds1: distinct (map fst ds1)
    using Cons.IH[OF Cons.prems] rec by simp
  show ?case
  proof (cases a = u ∧ b ∉ set settled)
    case False
    then show ?thesis
      using distinct-ds1 rec unfolding e
      by (auto simp: Let-def split: option.splits if-splits)
  next
    case True
    note active = True
```

```
    show ?thesis
    proof (cases bmssp-lookup-dist ds1 b)
      case None
      then show ?thesis
        using active rec bmssp-set-dist-preserves-distinct[OF distinct-ds1]
        unfolding e by (simp add: Let-def)
    next
      case (Some old)
      show ?thesis
      proof (cases du + w < old)
        case True
        then show ?thesis
          using active rec Some
            bmssp-set-dist-preserves-distinct[OF distinct-ds1]
          unfolding e by (simp add: Let-def)
      next
        case False
        then show ?thesis
          using active rec Some distinct-ds1 unfolding e by (simp add: Let-def)
      qed
    qed
  qed
qed

lemma bmssp-relax-vertices-preserves-distinct-dist:
  assumes distinct: distinct (map fst ds)
  shows distinct (map fst (snd (bmssp-relax-vertices G settled xs ds)))
  using distinct
proof (induction xs arbitrary: ds)
  case Nil
  then show ?case
    by simp
next
  case (Cons u us)
  show ?case
  proof (cases bmssp-lookup-dist ds u)
    case None
    then show ?thesis
      using Cons.IH[OF Cons.prems] by simp
  next
    case (Some du)
    obtain updates ds1 where edge-rec:
      bmssp-relax-edges G settled u du ds = (updates, ds1)
      by force
    have distinct-ds1: distinct (map fst ds1)
      using bmssp-relax-edges-preserves-distinct-dist[OF Cons.prems,
          of G settled u du] edge-rec by simp
    obtain updates-us ds2 where vertices-rec:
      bmssp-relax-vertices G settled us ds1 = (updates-us, ds2)
```

```
    by force
  have distinct-ds2: distinct (map fst ds2)
    using Cons.IH[OF distinct-ds1] vertices-rec by simp
  show ?thesis
    using Some edge-rec vertices-rec distinct-ds2 by simp
 qed
qed

lemma bmssp-relax-edges-dist-keys-subset:
 assumes keys: set (map fst ds) ⊆ set vertices
  and uV: u ∈ set vertices
  and edge-targets:
   ⋀a b w. (a, b, w) ∈ set G ⟹ b ∈ set vertices
 shows set (map fst (snd (bmssp-relax-edges G settled u du ds)))
  ⊆ set vertices
 using keys edge-targets
proof (induction G arbitrary: ds)
 case Nil
 then show ?case
  by simp
next
 case (Cons e es)
 obtain a b w where e: e = (a, b, w)
  by (cases e)
 obtain updates ds1 where rec:
  bmssp-relax-edges es settled u du ds = (updates, ds1)
  by force
 have tail-subset: set (map fst ds1) ⊆ set vertices
 proof −
  have targets-tail:
   ⋀x y z. (x, y, z) ∈ set es ⟹ y ∈ set vertices
  proof −
   fix x y z
   assume xyz: (x, y, z) ∈ set es
   have (x, y, z) ∈ set (e # es)
    using xyz by simp
   then show y ∈ set vertices
    by (rule Cons.prems(2))
  qed
  show ?thesis
   using Cons.IH[OF Cons.prems(1) targets-tail] rec by simp
 qed
 have bV: b ∈ set vertices
  using Cons.prems(2)[of a b w] unfolding e by simp
 show ?case
 proof (cases a = u ∧ b ∉ set settled)
  case False
  then show ?thesis
   using tail-subset rec unfolding e
```

```
    by (auto simp: Let-def split: option.splits if-splits)
  next
    case True
    note active = True
    let ?nd = du + w
    show ?thesis
    proof (cases bmssp-lookup-dist ds1 b)
      case None
      then show ?thesis
        using active rec tail-subset bV
        unfolding e by (simp add: Let-def bmssp-set-dist-keys)
    next
      case (Some old)
      show ?thesis
      proof (cases ?nd < old)
        case True
        then show ?thesis
          using active rec Some tail-subset bV
          unfolding e by (simp add: Let-def bmssp-set-dist-keys)
      next
        case False
        then show ?thesis
          using active rec Some tail-subset unfolding e by (simp add: Let-def)
      qed
    qed
  qed
qed

lemma bmssp-relax-vertices-dist-keys-subset:
  assumes keys: set (map fst ds) ⊆ set vertices
    and xs-subset: set xs ⊆ set vertices
    and edge-targets:
      ⋀a b w. (a, b, w) ∈ set G ⟹ b ∈ set vertices
  shows set (map fst (snd (bmssp-relax-vertices G settled xs ds)))
    ⊆ set vertices
  using keys xs-subset
proof (induction xs arbitrary: ds)
  case Nil
  then show ?case
    by simp
next
  case (Cons u us)
  have uV: u ∈ set vertices
    using Cons.prems by simp
  have us-subset: set us ⊆ set vertices
    using Cons.prems by simp
  show ?case
  proof (cases bmssp-lookup-dist ds u)
    case None
```

```
    then show ?thesis
      using Cons.IH[OF Cons.prems(1) us-subset] by simp
  next
    case (Some du)
    obtain updates ds1 where edge-rec:
      bmssp-relax-edges G settled u du ds = (updates, ds1)
      by force
    have keys1: set (map fst ds1) ⊆ set vertices
    proof −
      have set (map fst (snd (bmssp-relax-edges G settled u du ds)))
          ⊆ set vertices
        by (rule bmssp-relax-edges-dist-keys-subset
            [OF Cons.prems(1) uV edge-targets])
      then show ?thesis
        using edge-rec by simp
    qed
    obtain updates-us ds2 where vertices-rec:
      bmssp-relax-vertices G settled us ds1 = (updates-us, ds2)
      by force
    have keys2: set (map fst ds2) ⊆ set vertices
      using Cons.IH[OF keys1 us-subset] vertices-rec by simp
    show ?thesis
      using Some edge-rec vertices-rec keys2 by simp
  qed
qed

definition real-label-integral-on :: nat set ⇒ (nat ⇒ real) ⇒ bool where
  real-label-integral-on U d ⟷
    (∀ v∈U. real (nat (floor (d v))) = d v)

definition encode-dist-assoc-list :: nat list ⇒ (nat ⇒ real) ⇒ nat-dist where
  encode-dist-assoc-list xs d =
    map (λv. (v, nat (floor (d v)))) (sort (remdups xs))

lemma encode-dist-assoc-list-memI:
  assumes v ∈ set xs
  shows ∃ d. (v, d) ∈ set (encode-dist-assoc-list xs f)
  using assms unfolding encode-dist-assoc-list-def by auto

lemma encode-dist-assoc-list-memD:
  assumes (v, d) ∈ set (encode-dist-assoc-list xs f)
  shows v ∈ set xs
  using assms unfolding encode-dist-assoc-list-def by auto

lemma sort-remdups-eq-if-set-eq:
  assumes set xs = set ys
  shows sort (remdups xs) = sort (remdups ys)
proof −
  have set-eq: set (sort (remdups xs)) = set (sort (remdups ys))
```

```
    using assms by simp
  have distinct-x: distinct (sort (remdups xs))
    by simp
  have distinct-y: distinct (sort (remdups ys))
    by simp
  have sorted-x: sorted (sort (remdups xs))
    by simp
  have sorted-y: sorted (sort (remdups ys))
    by simp
  have mset-eq:
    mset (sort (remdups xs)) = mset (sort (remdups ys))
    using distinct-x distinct-y set-eq
    by (metis set-eq-iff-mset-eq-distinct)
  have sort (sort (remdups ys)) = sort (remdups xs)
  proof (rule properties-for-sort)
    show mset (sort (remdups xs)) = mset (sort (remdups ys))
      by (rule mset-eq)
    show sorted (sort (remdups xs))
      by (rule sorted-x)
  qed
  moreover have sort (sort (remdups ys)) = sort (remdups ys)
  proof (rule sort-key-id-if-sorted)
    show sorted (map (λx. x) (sort (remdups ys)))
      using sorted-y by simp
  qed
  ultimately show ?thesis
    by simp
qed

lemma encode-dist-assoc-list-cong-set:
  assumes set xs = set ys
  shows encode-dist-assoc-list xs f = encode-dist-assoc-list ys f
  unfolding encode-dist-assoc-list-def
  using sort-remdups-eq-if-set-eq[OF assms] by simp

lemma encode-dist-assoc-list-cong-set-floor:
  assumes set-eq: set xs = set ys
    and floors:
      ⋀v. v ∈ set xs ⟹
        nat (floor (f v)) = nat (floor (g v))
  shows encode-dist-assoc-list xs f = encode-dist-assoc-list ys g
proof −
  have order-eq: sort (remdups xs) = sort (remdups ys)
    by (rule sort-remdups-eq-if-set-eq[OF set-eq])
  have map-eq:
    map (λv. (v, nat (floor (f v)))) (sort (remdups xs)) =
      map (λv. (v, nat (floor (g v)))) (sort (remdups xs))
    by (rule map-cong[OF refl]) (use floors in auto)
  show ?thesis
```

```
    unfolding encode-dist-assoc-list-def using order-eq map-eq by simp
qed

lemma encode-dist-assoc-list-partition-key:
  encode-dist-assoc-list xs (λv. bmssp-partition-key v (f v)) =
    map (λv. (v, f v)) (sort (remdups xs))
  unfolding encode-dist-assoc-list-def
  by (simp add: bmssp-partition-key-floor)

lemma map-fst-pair-map [simp]:
  map fst (map (λx. (x, f x)) xs) = xs
  by (induction xs) simp-all

lemma bmssp-normalize-dist-encode:
  assumes distinct: distinct (map fst ds)
  shows bmssp-normalize-dist ds =
    encode-dist-assoc-list (map fst ds) (executable-label-of ds)
proof −
  let ?lhs = bmssp-normalize-dist ds
  let ?rhs = encode-dist-assoc-list (map fst ds) (executable-label-of ds)
  have lhs-sorted: sorted (map fst ?lhs)
    unfolding bmssp-normalize-dist-def by simp
  have rhs-sorted: sorted (map fst ?rhs)
  proof −
    have map fst ?rhs = sort (remdups (map fst ds))
      unfolding encode-dist-assoc-list-def by (simp add: o-def)
    then show ?thesis
      by simp
  qed
  have lhs-distinct: distinct (map fst ?lhs)
    unfolding bmssp-normalize-dist-def
    by (rule distinct-map-fst-sort-key[OF distinct])
  have rhs-distinct: distinct (map fst ?rhs)
  proof −
    have map fst ?rhs = sort (remdups (map fst ds))
      unfolding encode-dist-assoc-list-def by (simp add: o-def)
    then show ?thesis
      by simp
  qed
  have lhs-set: set ?lhs = set ds
    unfolding bmssp-normalize-dist-def by simp
  have rhs-set: set ?rhs = set ds
  proof
    show set ?rhs ⊆ set ds
    proof
      fix p
      assume p: p ∈ set ?rhs
      then obtain v where v:
          v ∈ set (map fst ds)
```

```
        p = (v, nat (floor (executable-label-of ds v)))
      unfolding encode-dist-assoc-list-def by auto
    then obtain d where mem: (v, d) ∈ set ds
      by force
    have lookup: bmssp-lookup-dist ds v = Some d
      by (rule bmssp-lookup-dist-Some-if-distinct-mem[OF distinct mem])
    then have nat (floor (executable-label-of ds v)) = d
      unfolding executable-label-of-def by simp
    then show p ∈ set ds
      using v mem by simp
  qed
 next
  show set ds ⊆ set ?rhs
  proof
    fix p
    assume p: p ∈ set ds
    obtain v d where p-eq: p = (v, d)
      by (cases p)
    then have mem: (v, d) ∈ set ds
      using p by simp
    have lookup: bmssp-lookup-dist ds v = Some d
      by (rule bmssp-lookup-dist-Some-if-distinct-mem[OF distinct mem])
    then have encoded-d: nat (floor (executable-label-of ds v)) = d
      unfolding executable-label-of-def by simp
    have v ∈ set (map fst ds)
      using mem by force
    then have (v, nat (floor (executable-label-of ds v))) ∈ set ?rhs
      unfolding encode-dist-assoc-list-def by simp
    then show p ∈ set ?rhs
      using p-eq encoded-d by simp
  qed
 qed
 have inj: inj-on fst (set ?lhs ∪ set ?rhs)
 proof −
  have inj-on fst (set ds)
    using distinct by (simp add: distinct-map inj-on-def)
  then show ?thesis
    using lhs-set rhs-set by simp
 qed
 show ?thesis
  by (rule map-sorted-distinct-set-unique
    [OF inj lhs-sorted lhs-distinct rhs-sorted rhs-distinct])
    (simp add: lhs-set rhs-set)
qed

lemma bmssp-loop-zero-encode:
 assumes distinct (map fst ds)
 shows bmssp-loop 0 G vertices settled ds P =
   encode-dist-assoc-list (map fst ds) (executable-label-of ds)
```

using *bmssp-normalize-dist-encode*[*OF assms*] **by** *simp*

lemma *bmssp-loop-empty-pull-encode*:
assumes *pull*: *bp-pull bmssp-block-size bmssp-bound P* = (*S*, *beta*, *P1*)
and *empty*: *filter* (λ*x*. *x* ∈ *S* ∧ *x* ∉ *set settled*) *vertices* = []
and *distinct*: *distinct* (*map fst ds*)
shows *bmssp-loop* (*Suc fuel*) *G vertices settled ds P* =
encode-dist-assoc-list (*map fst ds*) (*executable-label-of ds*)
using *pull empty bmssp-normalize-dist-encode*[*OF distinct*] **by** *simp*

lemma *bmssp-loop-preserves-distinct-output*:
assumes *distinct*: *distinct* (*map fst ds*)
shows *distinct* (*map fst* (*bmssp-loop fuel G vertices settled ds P*))
using *distinct*
by (*induction fuel arbitrary*: *settled ds P*)
(*auto simp*: *Let-def bmssp-normalize-dist-def*
split: *prod.splits*
intro: *distinct-map-fst-sort-key*
dest: *bmssp-relax-vertices-preserves-distinct-dist*)

lemma *bmssp-loop-output-sorted-keys*:
sorted (*map fst* (*bmssp-loop fuel G vertices settled ds P*))
by (*induction fuel arbitrary*: *settled ds P*)
(*auto simp*: *Let-def bmssp-normalize-dist-def split*: *prod.splits*)

lemma *distinct-sorted-dist-encode*:
assumes *distinct*: *distinct* (*map fst ds*)
and *sorted*: *sorted* (*map fst ds*)
shows *ds* = *encode-dist-assoc-list* (*map fst ds*) (*executable-label-of ds*)
proof −
have *bmssp-normalize-dist ds* = *ds*
using *sorted* **unfolding** *bmssp-normalize-dist-def*
by (*simp add*: *sort-key-id-if-sorted*)
then show *?thesis*
using *bmssp-normalize-dist-encode*[*OF distinct*] **by** *simp*
qed

lemma *bmssp-loop-output-encode*:
assumes *distinct*: *distinct* (*map fst ds*)
shows *bmssp-loop fuel G vertices settled ds P* =
encode-dist-assoc-list
(*map fst* (*bmssp-loop fuel G vertices settled ds P*))
(*executable-label-of* (*bmssp-loop fuel G vertices settled ds P*))
proof −
have *distinct-out*:
distinct (*map fst* (*bmssp-loop fuel G vertices settled ds P*))
by (*rule bmssp-loop-preserves-distinct-output*[*OF distinct*])
have *sorted-out*:
sorted (*map fst* (*bmssp-loop fuel G vertices settled ds P*))

```
    by (rule bmssp-loop-output-sorted-keys)
  show ?thesis
    by (rule distinct-sorted-dist-encode[OF distinct-out sorted-out])
qed

lemma bmssp-loop-output-keys-subset-vertices:
  assumes keys: set (map fst ds) ⊆ set vertices
    and edge-targets:
      ⋀a b w. (a, b, w) ∈ set G ⟹ b ∈ set vertices
  shows set (map fst (bmssp-loop fuel G vertices settled ds P))
    ⊆ set vertices
  using keys
proof (induction fuel arbitrary: settled ds P)
  case 0
  then show ?case
    unfolding bmssp-normalize-dist-def by simp
next
  case (Suc fuel)
  obtain S beta P1 where pull:
    bp-pull bmssp-block-size bmssp-bound P = (S, beta, P1)
    by (cases bp-pull bmssp-block-size bmssp-bound P) auto
  let ?pulled = filter (λx. x ∈ S ∧ x ∉ set settled) vertices
  show ?case
  proof (cases ?pulled = [])
    case True
    then show ?thesis
      using pull Suc.prems unfolding bmssp-normalize-dist-def by simp
  next
    case False
    let ?settled' = ?pulled @ settled
    obtain updates ds' where relaxed:
      bmssp-relax-vertices G ?settled' ?pulled ds = (updates, ds')
      by force
    have pulled-subset: set ?pulled ⊆ set vertices
      by simp
    have keys-ds': set (map fst ds') ⊆ set vertices
    proof −
      have set (map fst
          (snd (bmssp-relax-vertices G ?settled' ?pulled ds)))
          ⊆ set vertices
        by (rule bmssp-relax-vertices-dist-keys-subset
            [OF Suc.prems pulled-subset edge-targets])
      then show ?thesis
        using relaxed by simp
    qed
    have rec:
      set (map fst
        (bmssp-loop fuel G vertices ?settled' ds'
          (bmssp-apply-updates updates P1))) ⊆ set vertices
```

```
    by (rule Suc.IH[OF keys-ds'])
   show ?thesis
    using pull False relaxed rec by (simp add: Let-def)
  qed
qed

lemma executable-label-integral-on-keys:
  assumes distinct: distinct (map fst ds)
  shows real-label-integral-on (set (map fst ds)) (executable-label-of ds)
  unfolding real-label-integral-on-def
proof
  fix v
  assume v ∈ set (map fst ds)
  then obtain d where mem: (v, d) ∈ set ds
    by force
  have lookup: bmssp-lookup-dist ds v = Some d
    by (rule bmssp-lookup-dist-Some-if-distinct-mem[OF distinct mem])
  then show real (nat (floor (executable-label-of ds v))) =
    executable-label-of ds v
    unfolding executable-label-of-def by simp
qed

lemma nat-graph-vertex-list-set [simp]:
  set (nat-graph-vertex-list G) = nat-graph-vertices G
  unfolding nat-graph-vertex-list-def nat-graph-vertices-def by simp

lemma nat-graph-edge-list-set [simp]:
  set (nat-graph-edge-list G) = nat-graph-edges G
  unfolding nat-graph-edge-list-def nat-graph-edges-def by simp

lemma bmssp-relax-edges-update-edge:
  assumes (v, b) ∈ set (fst (bmssp-relax-edges G settled u du ds))
  shows (u, v) ∈ set (nat-graph-edge-list G)
  using assms
proof (induction G arbitrary: ds v b)
  case Nil
  then show ?case
    by simp
next
  case (Cons e es)
  obtain a c w where e: e = (a, c, w)
    by (cases e)
  obtain updates ds1 where rec:
    bmssp-relax-edges es settled u du ds = (updates, ds1)
    by force
  have tail:
    ⋀v b. (v, b) ∈ set updates ⟹
      (u, v) ∈ set (nat-graph-edge-list es)
  proof −
```

```
  fix v b
  assume update: (v, b) ∈ set updates
  have (v, b) ∈ set (fst (bmssp-relax-edges es settled u du ds))
    using update rec by simp
  then show (u, v) ∈ set (nat-graph-edge-list es)
    by (rule Cons.IH)
qed
show ?case
proof (cases a = u ∧ c ∉ set settled)
  case False
  then have (v, b) ∈ set updates
    using Cons.prems by (auto simp: e rec split: option.splits if-splits)
  then have (u, v) ∈ set (nat-graph-edge-list es)
    by (rule tail)
  then show ?thesis
    unfolding e nat-graph-edge-list-def by simp
next
  case True
  note active = True
  let ?nd = du + w
  show ?thesis
  proof (cases bmssp-lookup-dist ds1 c)
    case None
    then have update-or-tail:
      (v, b) = (c, bmssp-partition-key c ?nd) ∨
        (v, b) ∈ set updates
      using Cons.prems active unfolding e by (simp add: rec Let-def)
    then show ?thesis
    proof
      assume (v, b) = (c, bmssp-partition-key c ?nd)
      then show ?thesis
        using True unfolding e nat-graph-edge-list-def by simp
    next
      assume (v, b) ∈ set updates
      then have (u, v) ∈ set (nat-graph-edge-list es)
        by (rule tail)
      then show ?thesis
        unfolding e nat-graph-edge-list-def by simp
    qed
  next
    case (Some old)
    show ?thesis
    proof (cases ?nd < old)
      case False
      then have (v, b) ∈ set updates
        using Cons.prems active Some unfolding e by (simp add: rec Let-def)
      then have (u, v) ∈ set (nat-graph-edge-list es)
        by (rule tail)
      then show ?thesis
```

```
        unfolding e nat-graph-edge-list-def by simp
      next
        case True
        then have update-or-tail:
          (v, b) = (c, bmssp-partition-key c ?nd) ∨
            (v, b) ∈ set updates
          using Cons.prems active Some unfolding e by (simp add: rec Let-def)
        then show ?thesis
        proof
          assume (v, b) = (c, bmssp-partition-key c ?nd)
          then show ?thesis
            using active unfolding e nat-graph-edge-list-def by simp
        next
          assume (v, b) ∈ set updates
          then have (u, v) ∈ set (nat-graph-edge-list es)
            by (rule tail)
          then show ?thesis
            unfolding e nat-graph-edge-list-def by simp
        qed
      qed
    qed
  qed
qed

lemma bmssp-relax-edges-update-partition-key:
  assumes (v, b) ∈ set (fst (bmssp-relax-edges G settled u du ds))
  shows ∃ d. b = bmssp-partition-key v d
  using assms
proof (induction G arbitrary: ds v b)
  case Nil
  then show ?case
    by simp
next
  case (Cons e es)
  obtain a c w where e: e = (a, c, w)
    by (cases e)
  obtain updates ds1 where rec:
    bmssp-relax-edges es settled u du ds = (updates, ds1)
    by force
  have tail:
    ⋀v b. (v, b) ∈ set updates ⟹
      ∃ d. b = bmssp-partition-key v d
  proof −
    fix v b
    assume update: (v, b) ∈ set updates
    have (v, b) ∈ set (fst (bmssp-relax-edges es settled u du ds))
      using update rec by simp
    then show ∃ d. b = bmssp-partition-key v d
      by (rule Cons.IH)
```

```
qed
show ?case
proof (cases a = u ∧ c ∉ set settled)
  case False
  then have (v, b) ∈ set updates
    using Cons.prems by (auto simp: e rec split: option.splits if-splits)
  then show ?thesis
    by (rule tail)
next
  case True
  note active = True
  let ?nd = du + w
  show ?thesis
  proof (cases bmssp-lookup-dist ds1 c)
    case None
    then have update-or-tail:
      (v, b) = (c, bmssp-partition-key c ?nd) ∨
        (v, b) ∈ set updates
      using Cons.prems active unfolding e by (simp add: rec Let-def)
    then show ?thesis
    proof
      assume (v, b) = (c, bmssp-partition-key c ?nd)
      then show ?thesis
        by auto
    next
      assume (v, b) ∈ set updates
      then show ?thesis
        by (rule tail)
    qed
  next
    case (Some old)
    show ?thesis
    proof (cases ?nd < old)
      case False
      then have (v, b) ∈ set updates
        using Cons.prems active Some unfolding e by (simp add: rec Let-def)
      then show ?thesis
        by (rule tail)
    next
      case True
      then have update-or-tail:
        (v, b) = (c, bmssp-partition-key c ?nd) ∨
          (v, b) ∈ set updates
        using Cons.prems active Some unfolding e by (simp add: rec Let-def)
      then show ?thesis
      proof
        assume (v, b) = (c, bmssp-partition-key c ?nd)
        then show ?thesis
          by auto
```

```
      next
        assume (v, b) ∈ set updates
        then show ?thesis
          by (rule tail)
      qed
    qed
  qed
qed

lemma bmssp-relax-vertices-update-partition-key:
  assumes (v, b) ∈ set (fst (bmssp-relax-vertices G settled xs ds))
  shows ∃ d. b = bmssp-partition-key v d
  using assms
proof (induction xs arbitrary: ds)
  case Nil
  then show ?case
    by simp
next
  case (Cons u us)
  show ?case
  proof (cases bmssp-lookup-dist ds u)
    case None
    then have (v, b) ∈ set (fst (bmssp-relax-vertices G settled us ds))
      using Cons.prems by simp
    then show ?thesis
      by (rule Cons.IH)
  next
    case (Some du)
    obtain updates-u ds1 where edge-rec:
      bmssp-relax-edges G settled u du ds = (updates-u, ds1)
      by force
    obtain updates-us ds2 where vertices-rec:
      bmssp-relax-vertices G settled us ds1 = (updates-us, ds2)
      by force
    have update-cases:
      (v, b) ∈ set updates-u ∨ (v, b) ∈ set updates-us
      using Cons.prems Some edge-rec vertices-rec by auto
    then show ?thesis
    proof
      assume (v, b) ∈ set updates-u
      then have (v, b) ∈ set (fst (bmssp-relax-edges G settled u du ds))
        using edge-rec by simp
      then show ?thesis
        by (rule bmssp-relax-edges-update-partition-key)
    next
      assume (v, b) ∈ set updates-us
      then have (v, b) ∈ set (fst (bmssp-relax-vertices G settled us ds1))
        using vertices-rec by simp
```

```
    then show ?thesis
      by (rule Cons.IH)
  qed
 qed
qed

lemma bmssp-relax-vertices-update-floor:
 assumes (v, b) ∈ set (fst (bmssp-relax-vertices G settled xs ds))
 shows ∃ d. b = bmssp-partition-key v d ∧ nat (floor b) = d
proof −
 obtain d where d: b = bmssp-partition-key v d
   using bmssp-relax-vertices-update-partition-key[OF assms] by blast
 then have nat (floor b) = d
   by (simp add: bmssp-partition-key-floor)
 then show ?thesis
   using d by blast
qed

lemma bmssp-relax-edges-update-floor:
 assumes (v, b) ∈ set (fst (bmssp-relax-edges G settled u du ds))
 shows ∃ d. b = bmssp-partition-key v d ∧ nat (floor b) = d
proof −
 obtain d where d: b = bmssp-partition-key v d
   using bmssp-relax-edges-update-partition-key[OF assms] by blast
 then have nat (floor b) = d
   by (simp add: bmssp-partition-key-floor)
 then show ?thesis
   using d by blast
qed

lemma bmssp-relax-edges-update-not-settled:
 assumes (v, b) ∈ set (fst (bmssp-relax-edges G settled u du ds))
 shows v ∉ set settled
 using assms
proof (induction G arbitrary: ds v b)
 case Nil
 then show ?case
   by simp
next
 case (Cons e es)
 obtain a c w where e: e = (a, c, w)
   by (cases e)
 obtain updates ds1 where rec:
   bmssp-relax-edges es settled u du ds = (updates, ds1)
   by force
 have tail:
   ⋀v b. (v, b) ∈ set updates ⟹ v ∉ set settled
 proof −
   fix v b
```

```
  assume update: (v, b) ∈ set updates
  have (v, b) ∈ set (fst (bmssp-relax-edges es settled u du ds))
    using update rec by simp
  then show v ∉ set settled
    by (rule Cons.IH)
qed
show ?case
proof (cases a = u ∧ c ∉ set settled)
  case False
  then have (v, b) ∈ set updates
    using Cons.prems rec unfolding e
    by (auto simp: Let-def split: option.splits if-splits)
  then show ?thesis
    by (rule tail)
next
  case True
  note active = True
  let ?nd = du + w
  show ?thesis
  proof (cases bmssp-lookup-dist ds1 c)
    case None
    then have update-or-tail:
      (v, b) = (c, bmssp-partition-key c ?nd) ∨
        (v, b) ∈ set updates
      using Cons.prems active rec unfolding e by (simp add: Let-def)
    then show ?thesis
    proof
      assume (v, b) = (c, bmssp-partition-key c ?nd)
      then show ?thesis
        using active by simp
    next
      assume (v, b) ∈ set updates
      then show ?thesis
        by (rule tail)
    qed
  next
    case (Some old)
    show ?thesis
    proof (cases ?nd < old)
      case False
      then have (v, b) ∈ set updates
        using Cons.prems active Some rec unfolding e by (simp add: Let-def)
      then show ?thesis
        by (rule tail)
    next
      case True
      then have update-or-tail:
        (v, b) = (c, bmssp-partition-key c ?nd) ∨
          (v, b) ∈ set updates
```

```
        using Cons.prems active Some rec unfolding e by (simp add: Let-def)
      then show ?thesis
      proof
        assume (v, b) = (c, bmssp-partition-key c ?nd)
        then show ?thesis
          using active by simp
      next
        assume (v, b) ∈ set updates
        then show ?thesis
          by (rule tail)
      qed
    qed
  qed
qed

lemma bmssp-relax-vertices-update-not-settled:
  assumes (v, b) ∈ set (fst (bmssp-relax-vertices G settled xs ds))
  shows v ∉ set settled
  using assms
proof (induction xs arbitrary: ds)
  case Nil
  then show ?case
    by simp
next
  case (Cons u us)
  show ?case
  proof (cases bmssp-lookup-dist ds u)
    case None
    then have (v, b) ∈ set (fst (bmssp-relax-vertices G settled us ds))
      using Cons.prems by simp
    then show ?thesis
      by (rule Cons.IH)
  next
    case (Some du)
    obtain updates-u ds1 where edge-rec:
      bmssp-relax-edges G settled u du ds = (updates-u, ds1)
      by force
    obtain updates-us ds2 where vertices-rec:
      bmssp-relax-vertices G settled us ds1 = (updates-us, ds2)
      by force
    have update-cases:
      (v, b) ∈ set updates-u ∨ (v, b) ∈ set updates-us
      using Cons.prems Some edge-rec vertices-rec by auto
    then show ?thesis
    proof
      assume (v, b) ∈ set updates-u
      then have (v, b) ∈ set (fst (bmssp-relax-edges G settled u du ds))
        using edge-rec by simp
```

```
      then show ?thesis
        by (rule bmssp-relax-edges-update-not-settled)
    next
      assume (v, b) ∈ set updates-us
      then have (v, b) ∈ set (fst (bmssp-relax-vertices G settled us ds1))
        using vertices-rec by simp
      then show ?thesis
        by (rule Cons.IH)
    qed
  qed
qed

lemma bmssp-relax-edges-dist-keys:
  set (map fst (snd (bmssp-relax-edges G settled u du ds))) =
    set (map fst ds) ∪ fst ‘ set (fst (bmssp-relax-edges G settled u du ds))
proof (induction G arbitrary: ds)
  case Nil
  then show ?case
    by simp
next
  case (Cons e es)
  obtain a c w where e: e = (a, c, w)
    by (cases e)
  obtain updates ds1 where rec:
    bmssp-relax-edges es settled u du ds = (updates, ds1)
    by force
  have tail:
    set (map fst ds1) = set (map fst ds) ∪ fst ‘ set updates
    using Cons.IH[of ds] rec by simp
  show ?case
  proof (cases a = u ∧ c ∉ set settled)
    case False
    then show ?thesis
      using rec tail unfolding e
      by (auto simp: Let-def split: option.splits if-splits)
  next
    case True
    note active = True
    let ?nd = du + w
    show ?thesis
    proof (cases bmssp-lookup-dist ds1 c)
      case None
      then show ?thesis
        using active rec tail unfolding e
        by (auto simp: Let-def bmssp-set-dist-keys)
    next
      case (Some old)
      show ?thesis
      proof (cases ?nd < old)
```

```
      case False
      then show ?thesis
        using active rec Some tail unfolding e by (auto simp: Let-def)
    next
      case True
      then show ?thesis
        using active rec Some tail unfolding e
        by (auto simp: Let-def bmssp-set-dist-keys)
    qed
  qed
qed

lemma bmssp-relax-vertices-dist-keys:
  set (map fst (snd (bmssp-relax-vertices G settled xs ds))) =
    set (map fst ds) ∪ fst ‘ set (fst (bmssp-relax-vertices G settled xs ds))
proof (induction xs arbitrary: ds)
  case Nil
  then show ?case
    by simp
next
  case (Cons u us)
  show ?case
  proof (cases bmssp-lookup-dist ds u)
    case None
    then show ?thesis
      using Cons.IH[of ds] by simp
  next
    case (Some du)
    obtain updates-u ds1 where edge-rec:
      bmssp-relax-edges G settled u du ds = (updates-u, ds1)
      by force
    obtain updates-us ds2 where vertices-rec:
      bmssp-relax-vertices G settled us ds1 = (updates-us, ds2)
      by force
    have edge-keys:
      set (map fst ds1) = set (map fst ds) ∪ fst ‘ set updates-u
      using bmssp-relax-edges-dist-keys[of G settled u du ds] edge-rec
      by simp
    have tail-keys:
      set (map fst ds2) = set (map fst ds1) ∪ fst ‘ set updates-us
      using Cons.IH[of ds1] vertices-rec by simp
    show ?thesis
      using Some edge-rec vertices-rec edge-keys tail-keys by auto
  qed
qed

lemma bmssp-relax-edges-lookup-not-updated:
  assumes v ∉ fst ‘ set (fst (bmssp-relax-edges G settled u du ds))
```

```
  shows bmssp-lookup-dist (snd (bmssp-relax-edges G settled u du ds)) v =
    bmssp-lookup-dist ds v
  using assms
proof (induction G arbitrary: ds)
  case Nil
  then show ?case
    by simp
next
  case (Cons e es)
  obtain a c w where e: e = (a, c, w)
    by (cases e)
  obtain updates ds1 where rec:
    bmssp-relax-edges es settled u du ds = (updates, ds1)
    by force
  have v-not-tail: v ∉ fst ‘ set updates
    using Cons.prems rec unfolding e
    by (auto simp: Let-def split: option.splits if-splits)
  have tail-lookup: bmssp-lookup-dist ds1 v = bmssp-lookup-dist ds v
    using Cons.IH[of ds] v-not-tail rec by simp
  show ?case
  proof (cases a = u ∧ c ∉ set settled)
    case False
    then show ?thesis
      using rec tail-lookup unfolding e
      by (auto simp: Let-def split: option.splits if-splits)
  next
    case True
    note active = True
    let ?nd = du + w
    show ?thesis
    proof (cases bmssp-lookup-dist ds1 c)
      case None
      then have v-ne-c: v ≠ c
        using Cons.prems active rec unfolding e by (auto simp: Let-def)
      show ?thesis
        using active rec None tail-lookup v-ne-c unfolding e
        by (simp add: Let-def bmssp-lookup-dist-set-dist-other)
    next
      case (Some old)
      show ?thesis
      proof (cases ?nd < old)
        case False
        then show ?thesis
          using active rec Some tail-lookup unfolding e by (simp add: Let-def)
      next
        case True
        then have v-ne-c: v ≠ c
          using Cons.prems active rec Some unfolding e by (auto simp: Let-def)
        show ?thesis
```

```
        using active rec Some True tail-lookup v-ne-c unfolding e
        by (simp add: Let-def bmssp-lookup-dist-set-dist-other)
      qed
    qed
  qed
qed

lemma bmssp-relax-vertices-lookup-not-updated:
  assumes v ∉ fst ‘ set (fst (bmssp-relax-vertices G settled xs ds))
  shows bmssp-lookup-dist (snd (bmssp-relax-vertices G settled xs ds)) v =
    bmssp-lookup-dist ds v
  using assms
proof (induction xs arbitrary: ds)
  case Nil
  then show ?case
    by simp
next
  case (Cons u us)
  show ?case
  proof (cases bmssp-lookup-dist ds u)
    case None
    then have v-not-tail:
      v ∉ fst ‘ set (fst (bmssp-relax-vertices G settled us ds))
      using Cons.prems by simp
    show ?thesis
      using None Cons.IH[OF v-not-tail] by simp
  next
    case (Some du)
    obtain updates-u ds1 where edge-rec:
      bmssp-relax-edges G settled u du ds = (updates-u, ds1)
      by force
    obtain updates-us ds2 where vertices-rec:
      bmssp-relax-vertices G settled us ds1 = (updates-us, ds2)
      by force
    have v-not-u: v ∉ fst ‘ set updates-u
      using Cons.prems Some edge-rec vertices-rec by auto
    have v-not-us: v ∉ fst ‘ set updates-us
      using Cons.prems Some edge-rec vertices-rec by auto
    have lookup-u: bmssp-lookup-dist ds1 v = bmssp-lookup-dist ds v
      using bmssp-relax-edges-lookup-not-updated[of v G settled u du ds]
        v-not-u edge-rec by simp
    have lookup-us: bmssp-lookup-dist ds2 v = bmssp-lookup-dist ds1 v
      using Cons.IH[of ds1] v-not-us vertices-rec by simp
    show ?thesis
      using Some edge-rec vertices-rec lookup-u lookup-us by simp
  qed
qed

lemma bmssp-relax-edges-update-lookup-floor:
```

```
  assumes distinct-updates:
      distinct (map fst (fst (bmssp-relax-edges G settled u du ds)))
    and update: (v, b) ∈ set (fst (bmssp-relax-edges G settled u du ds))
  shows bmssp-lookup-dist (snd (bmssp-relax-edges G settled u du ds)) v =
    Some (nat (floor b))
  using distinct-updates update
proof (induction G arbitrary: ds)
  case Nil
  then show ?case
    by simp
next
  case (Cons e es)
  obtain a c w where e: e = (a, c, w)
    by (cases e)
  obtain updates ds1 where rec:
    bmssp-relax-edges es settled u du ds = (updates, ds1)
    by force
  show ?case
  proof (cases a = u ∧ c ∉ set settled)
    case False
    have distinct-tail: distinct (map fst updates)
      using Cons.prems(1) False rec unfolding e
      by (auto simp: Let-def split: option.splits if-splits)
    have update-tail: (v, b) ∈ set updates
      using Cons.prems(2) False rec unfolding e
      by (auto simp: Let-def split: option.splits if-splits)
    have tail-lookup:
      bmssp-lookup-dist ds1 v = Some (nat (floor b))
      using Cons.IH[of ds] distinct-tail update-tail rec by simp
    then show ?thesis
      using False rec unfolding e
      by (auto simp: Let-def split: option.splits if-splits)
  next
    case True
    note active = True
    let ?nd = du + w
    show ?thesis
    proof (cases bmssp-lookup-dist ds1 c)
      case None
      let ?new = (c, bmssp-partition-key c ?nd)
      have all-updates:
        fst (bmssp-relax-edges (e # es) settled u du ds) = ?new # updates
        using active None rec unfolding e by (simp add: Let-def)
      have all-ds:
        snd (bmssp-relax-edges (e # es) settled u du ds) =
          bmssp-set-dist c ?nd ds1
        using active None rec unfolding e by (simp add: Let-def)
      have distinct-tail: distinct (map fst updates)
        using Cons.prems(1) all-updates by simp
```

```
  have c-not-tail: c ∉ fst ‘ set updates
    using Cons.prems(1) all-updates by simp
  show ?thesis
  proof (cases (v, b) = ?new)
    case True
    then show ?thesis
      using all-ds by (simp add: bmssp-partition-key-floor
          bmssp-lookup-dist-set-dist-same)
  next
    case False
    then have update-tail: (v, b) ∈ set updates
      using Cons.prems(2) all-updates by auto
    have v-ne-c: v ≠ c
      using update-tail c-not-tail by force
    have tail-lookup:
      bmssp-lookup-dist ds1 v = Some (nat (floor b))
      using Cons.IH[of ds] distinct-tail update-tail rec by simp
    show ?thesis
      using all-ds v-ne-c tail-lookup
      by (simp add: bmssp-lookup-dist-set-dist-other)
  qed
next
  case (Some old)
  show ?thesis
  proof (cases ?nd < old)
    case False
    have distinct-tail: distinct (map fst updates)
      using Cons.prems(1) active Some False rec unfolding e
      by (simp add: Let-def)
    have update-tail: (v, b) ∈ set updates
      using Cons.prems(2) active Some False rec unfolding e
      by (simp add: Let-def)
    have tail-lookup:
      bmssp-lookup-dist ds1 v = Some (nat (floor b))
      using Cons.IH[of ds] distinct-tail update-tail rec by simp
    then show ?thesis
      using active Some False rec unfolding e by (simp add: Let-def)
  next
    case True
    let ?new = (c, bmssp-partition-key c ?nd)
    have all-updates:
      fst (bmssp-relax-edges (e # es) settled u du ds) = ?new # updates
      using active Some True rec unfolding e by (simp add: Let-def)
    have all-ds:
      snd (bmssp-relax-edges (e # es) settled u du ds) =
        bmssp-set-dist c ?nd ds1
      using active Some True rec unfolding e by (simp add: Let-def)
    have distinct-tail: distinct (map fst updates)
      using Cons.prems(1) all-updates by simp
```

```
      have c-not-tail: c ∉ fst ‘ set updates
        using Cons.prems(1) all-updates by simp
      show ?thesis
      proof (cases (v, b) = ?new)
        case True
        then show ?thesis
          using all-ds by (simp add: bmssp-partition-key-floor
              bmssp-lookup-dist-set-dist-same)
      next
        case False
        then have update-tail: (v, b) ∈ set updates
          using Cons.prems(2) all-updates by auto
        have v-ne-c: v ≠ c
          using update-tail c-not-tail by force
        have tail-lookup:
          bmssp-lookup-dist ds1 v = Some (nat (floor b))
          using Cons.IH[of ds] distinct-tail update-tail rec by simp
        show ?thesis
          using all-ds v-ne-c tail-lookup
          by (simp add: bmssp-lookup-dist-set-dist-other)
      qed
    qed
  qed
qed

lemma bmssp-relax-vertices-update-lookup-floor:
  assumes distinct-updates:
      distinct (map fst (fst (bmssp-relax-vertices G settled xs ds)))
    and update: (v, b) ∈ set (fst (bmssp-relax-vertices G settled xs ds))
  shows bmssp-lookup-dist (snd (bmssp-relax-vertices G settled xs ds)) v =
    Some (nat (floor b))
  using distinct-updates update
proof (induction xs arbitrary: ds)
  case Nil
  then show ?case
    by simp
next
  case (Cons u us)
  show ?case
  proof (cases bmssp-lookup-dist ds u)
    case None
    have distinct-tail:
      distinct (map fst (fst (bmssp-relax-vertices G settled us ds)))
      using Cons.prems(1) None by simp
    have update-tail:
      (v, b) ∈ set (fst (bmssp-relax-vertices G settled us ds))
      using Cons.prems(2) None by simp
    show ?thesis
```

```
    using None Cons.IH[OF distinct-tail update-tail] by simp
  next
    case (Some du)
    obtain updates-u ds1 where edge-rec:
      bmssp-relax-edges G settled u du ds = (updates-u, ds1)
      by force
    obtain updates-us ds2 where vertices-rec:
      bmssp-relax-vertices G settled us ds1 = (updates-us, ds2)
      by force
    have updates-eq:
      fst (bmssp-relax-vertices G settled (u # us) ds) =
        updates-u @ updates-us
      using Some edge-rec vertices-rec by simp
    have ds-eq:
      snd (bmssp-relax-vertices G settled (u # us) ds) = ds2
      using Some edge-rec vertices-rec by simp
    have distinct-append: distinct (map fst (updates-u @ updates-us))
      using Cons.prems(1) updates-eq by simp
    show ?thesis
    proof (cases (v, b) ∈ set updates-u)
      case True
      have distinct-u: distinct (map fst updates-u)
        using distinct-append by simp
      have lookup-u: bmssp-lookup-dist ds1 v = Some (nat (floor b))
        using bmssp-relax-edges-update-lookup-floor[of G settled u du ds v b]
          distinct-u True edge-rec by simp
      have v-not-us: v ∉ fst ‘ set updates-us
        using distinct-append True by force
      have lookup-us: bmssp-lookup-dist ds2 v = bmssp-lookup-dist ds1 v
        using bmssp-relax-vertices-lookup-not-updated[of v G settled us ds1]
          v-not-us vertices-rec by simp
      show ?thesis
        using ds-eq lookup-u lookup-us by simp
    next
      case False
      have update-us: (v, b) ∈ set updates-us
        using Cons.prems(2) updates-eq False by auto
      have distinct-us: distinct (map fst updates-us)
        using distinct-append by simp
      have lookup-us: bmssp-lookup-dist ds2 v = Some (nat (floor b))
        using Cons.IH[of ds1] distinct-us update-us vertices-rec by simp
      show ?thesis
        using ds-eq lookup-us by simp
    qed
  qed
qed

lemma bmssp-relax-edges-lookup-Some-preserved:
  assumes distinct: distinct (map fst ds)
```

```
    and lookup: bmssp-lookup-dist ds v = Some d
  shows ∃ d'. bmssp-lookup-dist
    (snd (bmssp-relax-edges G settled u du ds)) v = Some d'
proof −
  have v-key: v ∈ set (map fst ds)
    using lookup by (rule bmssp-lookup-dist-mem-key)
  have keys:
    set (map fst (snd (bmssp-relax-edges G settled u du ds))) =
      set (map fst ds) ∪ fst ‘ set (fst (bmssp-relax-edges G settled u du ds))
    by (rule bmssp-relax-edges-dist-keys)
  have distinct':
    distinct (map fst (snd (bmssp-relax-edges G settled u du ds)))
    by (rule bmssp-relax-edges-preserves-distinct-dist[OF distinct])
  have v ∈ set (map fst (snd (bmssp-relax-edges G settled u du ds)))
    using v-key keys by blast
  then obtain d' where (v, d') ∈
      set (snd (bmssp-relax-edges G settled u du ds))
    by force
  then have bmssp-lookup-dist
      (snd (bmssp-relax-edges G settled u du ds)) v = Some d'
    by (rule bmssp-lookup-dist-Some-if-distinct-mem[OF distinct'])
  then show ?thesis
    by blast
qed

lemma bmssp-relax-vertices-lookup-Some-preserved:
  assumes distinct: distinct (map fst ds)
    and lookup: bmssp-lookup-dist ds v = Some d
  shows ∃ d'. bmssp-lookup-dist
    (snd (bmssp-relax-vertices G settled xs ds)) v = Some d'
proof −
  have v-key: v ∈ set (map fst ds)
    using lookup by (rule bmssp-lookup-dist-mem-key)
  have keys:
    set (map fst (snd (bmssp-relax-vertices G settled xs ds))) =
      set (map fst ds) ∪
        fst ‘ set (fst (bmssp-relax-vertices G settled xs ds))
    by (rule bmssp-relax-vertices-dist-keys)
  have distinct':
    distinct (map fst (snd (bmssp-relax-vertices G settled xs ds)))
    by (rule bmssp-relax-vertices-preserves-distinct-dist[OF distinct])
  have v ∈ set (map fst (snd (bmssp-relax-vertices G settled xs ds)))
    using v-key keys by blast
  then obtain d' where (v, d') ∈
      set (snd (bmssp-relax-vertices G settled xs ds))
    by force
  then have bmssp-lookup-dist
      (snd (bmssp-relax-vertices G settled xs ds)) v = Some d'
    by (rule bmssp-lookup-dist-Some-if-distinct-mem[OF distinct'])
```

```
  then show ?thesis
    by blast
qed

lemma bmssp-relax-edges-lookup-le:
  assumes distinct: distinct (map fst ds)
    and old: bmssp-lookup-dist ds v = Some old
    and new:
      bmssp-lookup-dist (snd (bmssp-relax-edges G settled u du ds)) v =
        Some new
  shows new ≤ old
  using distinct old new
proof (induction G arbitrary: ds v old new)
  case Nil
  then show ?case
    by simp
next
  case (Cons e es)
  obtain a c w where e: e = (a, c, w)
    by (cases e)
  obtain updates ds1 where rec:
    bmssp-relax-edges es settled u du ds = (updates, ds1)
    by force
  have tail-distinct: distinct (map fst ds1)
    using bmssp-relax-edges-preserves-distinct-dist[OF Cons.prems(1),
        of es settled u du] rec by simp
  show ?case
  proof (cases a = u ∧ c ∉ set settled)
    case False
    have tail-new:
      bmssp-lookup-dist (snd (bmssp-relax-edges es settled u du ds)) v =
        Some new
      using Cons.prems(3) False rec unfolding e
      by (auto simp: Let-def split: option.splits if-splits)
    show ?thesis
      by (rule Cons.IH[OF Cons.prems(1) Cons.prems(2) tail-new])
  next
    case True
    note active = True
    let ?nd = du + w
    show ?thesis
    proof (cases bmssp-lookup-dist ds1 c)
      case None
      show ?thesis
      proof (cases v = c)
        case True
        have old-c: bmssp-lookup-dist ds c = Some old
          using Cons.prems(2) True by simp
        have ∃ d'. bmssp-lookup-dist
```

```
      (snd (bmssp-relax-edges es settled u du ds)) c = Some d'
    by (rule bmssp-relax-edges-lookup-Some-preserved
      [OF Cons.prems(1) old-c])
  then show ?thesis
    using None rec by simp
 next
  case False
  have tail-new:
    bmssp-lookup-dist
      (snd (bmssp-relax-edges es settled u du ds)) v = Some new
    using Cons.prems(3) active None False rec unfolding e
    by (simp add: Let-def bmssp-lookup-dist-set-dist-other)
  show ?thesis
    by (rule Cons.IH[OF Cons.prems(1) Cons.prems(2) tail-new])
 qed
next
 case (Some cur)
 show ?thesis
 proof (cases ?nd < cur)
  case False
  have tail-new:
    bmssp-lookup-dist
      (snd (bmssp-relax-edges es settled u du ds)) v = Some new
    using Cons.prems(3) active Some False rec unfolding e
    by (simp add: Let-def)
  show ?thesis
    by (rule Cons.IH[OF Cons.prems(1) Cons.prems(2) tail-new])
 next
  case True
  note improves = True
  show ?thesis
  proof (cases v = c)
   case True
   have old-c: bmssp-lookup-dist ds c = Some old
    using Cons.prems(2) True by simp
   have tail-cur:
    bmssp-lookup-dist
      (snd (bmssp-relax-edges es settled u du ds)) c = Some cur
    using Some rec by simp
   have cur-le-old: cur ≤ old
    by (rule Cons.IH[OF Cons.prems(1) old-c tail-cur])
   have new-eq: new = ?nd
    using Cons.prems(3) active Some improves ‹v = c› rec unfolding e
    by (simp add: Let-def bmssp-lookup-dist-set-dist-same)
   then show ?thesis
    using improves cur-le-old by linarith
  next
   case False
   have tail-new:
```

```
          bmssp-lookup-dist
            (snd (bmssp-relax-edges es settled u du ds)) v = Some new
          using Cons.prems(3) active Some True False rec unfolding e
          by (simp add: Let-def bmssp-lookup-dist-set-dist-other)
        show ?thesis
          by (rule Cons.IH[OF Cons.prems(1) Cons.prems(2) tail-new])
      qed
    qed
  qed
qed

lemma bmssp-relax-vertices-lookup-le:
  assumes distinct: distinct (map fst ds)
    and old: bmssp-lookup-dist ds v = Some old
    and new:
      bmssp-lookup-dist (snd (bmssp-relax-vertices G settled xs ds)) v =
        Some new
  shows new ≤ old
  using distinct old new
proof (induction xs arbitrary: ds v old new)
  case Nil
  then show ?case
    by simp
next
  case (Cons u us)
  show ?case
  proof (cases bmssp-lookup-dist ds u)
    case None
    have tail-new:
      bmssp-lookup-dist (snd (bmssp-relax-vertices G settled us ds)) v =
        Some new
      using Cons.prems(3) None by simp
    show ?thesis
      by (rule Cons.IH[OF Cons.prems(1) Cons.prems(2) tail-new])
  next
    case (Some du)
    obtain updates-u ds1 where edge-rec:
      bmssp-relax-edges G settled u du ds = (updates-u, ds1)
      by force
    obtain updates-us ds2 where vertices-rec:
      bmssp-relax-vertices G settled us ds1 = (updates-us, ds2)
      by force
    have distinct-ds1: distinct (map fst ds1)
      using bmssp-relax-edges-preserves-distinct-dist[OF Cons.prems(1),
          of G settled u du] edge-rec by simp
    obtain mid where mid: bmssp-lookup-dist ds1 v = Some mid
    proof −
      have ∃ d'. bmssp-lookup-dist
```

```
        (snd (bmssp-relax-edges G settled u du ds)) v = Some d'
      by (rule bmssp-relax-edges-lookup-Some-preserved
        [OF Cons.prems(1) Cons.prems(2)])
    then obtain d' where bmssp-lookup-dist
        (snd (bmssp-relax-edges G settled u du ds)) v = Some d'
      by blast
    then have bmssp-lookup-dist ds1 v = Some d'
      using edge-rec by simp
    then show ?thesis
      by (rule that)
  qed
  have mid-le-old: mid ≤ old
  proof −
    have bmssp-lookup-dist
        (snd (bmssp-relax-edges G settled u du ds)) v = Some mid
      using mid edge-rec by simp
    then show ?thesis
      by (rule bmssp-relax-edges-lookup-le[OF Cons.prems(1) Cons.prems(2)])
  qed
  have final-new:
    bmssp-lookup-dist (snd (bmssp-relax-vertices G settled us ds1)) v =
      Some new
    using Cons.prems(3) Some edge-rec vertices-rec by simp
  have new-le-mid: new ≤ mid
    by (rule Cons.IH[OF distinct-ds1 mid final-new])
  then show ?thesis
    using mid-le-old by linarith
 qed
qed

lemma bmssp-relax-edges-edge-lookup-le-candidate:
 assumes edge: (u, v, w) ∈ set G
   and v-unsettled: v ∉ set settled
 shows ∃ dv.
   bmssp-lookup-dist (snd (bmssp-relax-edges G settled u du ds)) v =
     Some dv ∧
   dv ≤ du + w
 using edge
proof (induction G arbitrary: ds)
 case Nil
 then show ?case
   by simp
next
 case (Cons e es)
 obtain a b c where e: e = (a, b, c)
   by (cases e)
 obtain updates ds1 where rec:
   bmssp-relax-edges es settled u du ds = (updates, ds1)
   by force
```

```
show ?case
proof (cases (u, v, w) ∈ set es)
  case True
  then obtain dv where tail-lookup:
      bmssp-lookup-dist
        (snd (bmssp-relax-edges es settled u du ds)) v = Some dv
    and tail-le: dv ≤ du + w
    using Cons.IH by blast
  have ds1-lookup: bmssp-lookup-dist ds1 v = Some dv
    using tail-lookup rec by simp
  show ?thesis
  proof (cases a = u ∧ b ∉ set settled)
    case False
    then have final-lookup:
      bmssp-lookup-dist
        (snd (bmssp-relax-edges (e # es) settled u du ds)) v =
        Some dv
      using rec ds1-lookup unfolding e
      by (auto simp: Let-def split: option.splits if-splits)
    show ?thesis
      using final-lookup tail-le by blast
  next
    case True
    note active = True
    let ?nd = du + c
    show ?thesis
    proof (cases bmssp-lookup-dist ds1 b)
      case None
      show ?thesis
      proof (cases v = b)
        case True
        note v-eq-b = True
        then have final-lookup:
          bmssp-lookup-dist
            (snd (bmssp-relax-edges (e # es) settled u du ds)) v =
            Some ?nd
          using active v-eq-b None rec unfolding e
          by (simp add: Let-def bmssp-lookup-dist-set-dist-same)
        have ?nd ≤ du + w
          using v-eq-b None ds1-lookup by simp
        then show ?thesis
          using final-lookup by blast
      next
        case False
        note v-ne-b = False
        then have final-lookup:
          bmssp-lookup-dist
            (snd (bmssp-relax-edges (e # es) settled u du ds)) v =
            Some dv
```

```
      using active v-ne-b None rec ds1-lookup unfolding e
      by (simp add: Let-def bmssp-lookup-dist-set-dist-other)
    show ?thesis
      using final-lookup tail-le by blast
  qed
next
  case (Some old)
  show ?thesis
  proof (cases ?nd < old)
    case False
    then have final-lookup:
      bmssp-lookup-dist
        (snd (bmssp-relax-edges (e # es) settled u du ds)) v =
        Some dv
      using active Some rec ds1-lookup unfolding e by (simp add: Let-def)
    show ?thesis
      using final-lookup tail-le by blast
  next
    case True
    note improves = True
    show ?thesis
    proof (cases v = b)
      case True
      note v-eq-b = True
      then have final-lookup:
        bmssp-lookup-dist
          (snd (bmssp-relax-edges (e # es) settled u du ds)) v =
          Some ?nd
        using active Some improves rec v-eq-b
        unfolding e
        by (simp add: Let-def bmssp-lookup-dist-set-dist-same)
      have ?nd ≤ du + w
      proof −
        have old-eq: old = dv
          using v-eq-b Some ds1-lookup by simp
        show ?thesis
          using improves old-eq tail-le by linarith
      qed
      then show ?thesis
        using final-lookup by blast
    next
      case False
      note v-ne-b = False
      then have final-lookup:
        bmssp-lookup-dist
          (snd (bmssp-relax-edges (e # es) settled u du ds)) v =
          Some dv
        using active Some improves rec ds1-lookup v-ne-b
        unfolding e
```

```
          by (simp add: Let-def bmssp-lookup-dist-set-dist-other)
        show ?thesis
          using final-lookup tail-le by blast
      qed
    qed
  qed
next
  case False
  then have head: e = (u, v, w)
    using Cons.prems by simp
  have active: a = u ∧ b ∉ set settled
    using head v-unsettled unfolding e by simp
  let ?nd = du + w
  show ?thesis
  proof (cases bmssp-lookup-dist ds1 v)
    case None
    then have final-lookup:
      bmssp-lookup-dist
        (snd (bmssp-relax-edges (e # es) settled u du ds)) v =
        Some ?nd
      using active rec head unfolding e
      by (simp add: Let-def bmssp-lookup-dist-set-dist-same)
    show ?thesis
      using final-lookup by blast
  next
    case (Some old)
    show ?thesis
    proof (cases ?nd < old)
      case True
      then have final-lookup:
        bmssp-lookup-dist
          (snd (bmssp-relax-edges (e # es) settled u du ds)) v =
          Some ?nd
        using active Some rec head unfolding e
        by (simp add: Let-def bmssp-lookup-dist-set-dist-same)
      show ?thesis
        using final-lookup by blast
    next
      case False
      then have final-lookup:
        bmssp-lookup-dist
          (snd (bmssp-relax-edges (e # es) settled u du ds)) v =
          Some old
        using active Some rec head unfolding e by (simp add: Let-def)
      have old ≤ ?nd
        using False by simp
      then show ?thesis
        using final-lookup by blast
```

```
    qed
   qed
  qed
qed

lemma bmssp-relax-edges-update-improves-lookup:
  assumes distinct: distinct (map fst ds)
    and update: (v, b) ∈ set (fst (bmssp-relax-edges G settled u du ds))
    and old: bmssp-lookup-dist ds v = Some old
  shows nat (floor b) < old
  using distinct update old
proof (induction G arbitrary: ds v b old)
  case Nil
  then show ?case
    by simp
next
  case (Cons e es)
  obtain a c w where e: e = (a, c, w)
    by (cases e)
  obtain updates ds1 where rec:
    bmssp-relax-edges es settled u du ds = (updates, ds1)
    by force
  show ?case
  proof (cases a = u ∧ c ∉ set settled)
    case False
    have update-tail:
      (v, b) ∈ set (fst (bmssp-relax-edges es settled u du ds))
      using Cons.prems(2) False rec unfolding e
      by (auto simp: Let-def split: option.splits if-splits)
    show ?thesis
      by (rule Cons.IH[OF Cons.prems(1) update-tail Cons.prems(3)])
  next
    case True
    note active = True
    let ?nd = du + w
    show ?thesis
    proof (cases bmssp-lookup-dist ds1 c)
      case None
      have update-or-tail:
        (v, b) = (c, bmssp-partition-key c ?nd) ∨
          (v, b) ∈ set updates
        using Cons.prems(2) active None rec unfolding e by (simp add: Let-def)
      then show ?thesis
      proof
        assume current: (v, b) = (c, bmssp-partition-key c ?nd)
        have old-c: bmssp-lookup-dist ds c = Some old
          using Cons.prems(3) current by simp
        have ∃ d'. bmssp-lookup-dist
            (snd (bmssp-relax-edges es settled u du ds)) c = Some d'
```

```
      by (rule bmssp-relax-edges-lookup-Some-preserved
          [OF Cons.prems(1) old-c])
    then show ?thesis
      using None rec by simp
  next
    assume (v, b) ∈ set updates
    then have update-tail:
      (v, b) ∈ set (fst (bmssp-relax-edges es settled u du ds))
      using rec by simp
    show ?thesis
      by (rule Cons.IH[OF Cons.prems(1) update-tail Cons.prems(3)])
  qed
next
  case (Some cur)
  show ?thesis
  proof (cases ?nd < cur)
    case False
    have update-tail:
      (v, b) ∈ set (fst (bmssp-relax-edges es settled u du ds))
      using Cons.prems(2) active Some False rec unfolding e
      by (simp add: Let-def)
    show ?thesis
      by (rule Cons.IH[OF Cons.prems(1) update-tail Cons.prems(3)])
  next
    case True
    have update-or-tail:
      (v, b) = (c, bmssp-partition-key c ?nd) ∨
        (v, b) ∈ set updates
      using Cons.prems(2) active Some True rec unfolding e
      by (simp add: Let-def)
    then show ?thesis
    proof
      assume current: (v, b) = (c, bmssp-partition-key c ?nd)
      have old-c: bmssp-lookup-dist ds c = Some old
        using Cons.prems(3) current by simp
      have tail-cur:
        bmssp-lookup-dist
          (snd (bmssp-relax-edges es settled u du ds)) c = Some cur
        using Some rec by simp
      have cur-le-old: cur ≤ old
        by (rule bmssp-relax-edges-lookup-le
            [OF Cons.prems(1) old-c tail-cur])
      have nat (floor b) = ?nd
        using current by (simp add: bmssp-partition-key-floor)
      then show ?thesis
        using True cur-le-old by linarith
    next
      assume (v, b) ∈ set updates
      then have update-tail:
```

```
          (v, b) ∈ set (fst (bmssp-relax-edges es settled u du ds))
          using rec by simp
        show ?thesis
          by (rule Cons.IH[OF Cons.prems(1) update-tail Cons.prems(3)])
      qed
    qed
  qed
qed

lemma bmssp-relax-vertices-update-improves-lookup:
  assumes distinct: distinct (map fst ds)
    and update: (v, b) ∈ set (fst (bmssp-relax-vertices G settled xs ds))
    and old: bmssp-lookup-dist ds v = Some old
  shows nat (floor b) < old
  using distinct update old
proof (induction xs arbitrary: ds v b old)
  case Nil
  then show ?case
    by simp
next
  case (Cons u us)
  show ?case
  proof (cases bmssp-lookup-dist ds u)
    case None
    have update-tail:
      (v, b) ∈ set (fst (bmssp-relax-vertices G settled us ds))
      using Cons.prems(2) None by simp
    show ?thesis
      by (rule Cons.IH[OF Cons.prems(1) update-tail Cons.prems(3)])
  next
    case (Some du)
    obtain updates-u ds1 where edge-rec:
      bmssp-relax-edges G settled u du ds = (updates-u, ds1)
      by force
    obtain updates-us ds2 where vertices-rec:
      bmssp-relax-vertices G settled us ds1 = (updates-us, ds2)
      by force
    have update-cases: (v, b) ∈ set updates-u ∨ (v, b) ∈ set updates-us
      using Cons.prems(2) Some edge-rec vertices-rec by auto
    then show ?thesis
    proof
      assume (v, b) ∈ set updates-u
      then have edge-update:
        (v, b) ∈ set (fst (bmssp-relax-edges G settled u du ds))
        using edge-rec by simp
      show ?thesis
        by (rule bmssp-relax-edges-update-improves-lookup
            [OF Cons.prems(1) edge-update Cons.prems(3)])
```

```
    next
      assume update-us: (v, b) ∈ set updates-us
      have distinct-ds1: distinct (map fst ds1)
        using bmssp-relax-edges-preserves-distinct-dist[OF Cons.prems(1),
            of G settled u du] edge-rec by simp
      obtain mid where mid: bmssp-lookup-dist ds1 v = Some mid
      proof −
        have ∃ d'. bmssp-lookup-dist
            (snd (bmssp-relax-edges G settled u du ds)) v = Some d'
          by (rule bmssp-relax-edges-lookup-Some-preserved
              [OF Cons.prems(1) Cons.prems(3)])
        then obtain d' where bmssp-lookup-dist
            (snd (bmssp-relax-edges G settled u du ds)) v = Some d'
          by blast
        then have bmssp-lookup-dist ds1 v = Some d'
          using edge-rec by simp
        then show ?thesis
          by (rule that)
      qed
      have mid-le-old: mid ≤ old
      proof −
        have bmssp-lookup-dist
            (snd (bmssp-relax-edges G settled u du ds)) v = Some mid
          using mid edge-rec by simp
        then show ?thesis
          by (rule bmssp-relax-edges-lookup-le
              [OF Cons.prems(1) Cons.prems(3)])
      qed
      have update-tail:
        (v, b) ∈ set (fst (bmssp-relax-vertices G settled us ds1))
        using update-us vertices-rec by simp
      have nat (floor b) < mid
        by (rule Cons.IH[OF distinct-ds1 update-tail mid])
      then show ?thesis
        using mid-le-old by linarith
    qed
  qed
qed

lemma bmssp-relax-edges-updates-distinct:
  assumes distinct-edges: distinct (nat-graph-edge-list G)
  shows distinct (map fst (fst (bmssp-relax-edges G settled u du ds)))
  using distinct-edges
proof (induction G arbitrary: ds)
  case Nil
  then show ?case
    by simp
next
  case (Cons e es)
```

obtain *a c w* **where** *e*: $e = (a, c, w)$
by (*cases e*)
obtain *updates ds1* **where** *rec*:
bmssp-relax-edges es settled u du ds = (*updates*, *ds1*)
by *force*
have *tail-edges*: *distinct* (*nat-graph-edge-list es*)
using *Cons.prems* **unfolding** *e nat-graph-edge-list-def* **by** *simp*
have *tail-distinct*: *distinct* (*map fst updates*)
using *Cons.IH*[*OF tail-edges*, *of ds*] *rec* **by** *simp*
have *c-notin-tail*:
$a = u \Longrightarrow c \notin$ *fst ‘ set updates*
proof
assume *a-u*: $a = u$ **and** *c-tail*: $c \in$ *fst ‘ set updates*
then obtain *b* **where** *update*: $(c, b) \in$ *set updates*
by *force*
have *update-rec*:
$(c, b) \in$ *set* (*fst* (*bmssp-relax-edges es settled u du ds*))
using *update rec* **by** *simp*
have $(u, c) \in$ *set* (*nat-graph-edge-list es*)
by (*rule bmssp-relax-edges-update-edge*[*OF update-rec*])
then show *False*
using *Cons.prems a-u* **unfolding** *e nat-graph-edge-list-def* **by** *simp*
qed
show *?case*
proof (*cases* $a = u \wedge c \notin$ *set settled*)
case *False*
then show *?thesis*
using *tail-distinct* **by** (*auto simp*: *e rec split*: *option.splits if-splits*)
next
case *True*
note *active* = *True*
let *?nd* = *du* + *w*
have *c-notin*: $c \notin$ *fst ‘ set updates*
using *c-notin-tail active* **by** *blast*
show *?thesis*
proof (*cases bmssp-lookup-dist ds1 c*)
case *None*
then show *?thesis*
using *tail-distinct c-notin active* **unfolding** *e*
by (*simp add*: *rec Let-def*)
next
case (*Some old*)
show *?thesis*
proof (*cases ?nd* < *old*)
case *False*
then show *?thesis*
using *tail-distinct active Some* **unfolding** *e* **by** (*simp add*: *rec Let-def*)
next
case *True*

```
      then show ?thesis
        using tail-distinct c-notin active Some unfolding e
        by (simp add: rec Let-def)
    qed
  qed
 qed
qed

lemma bmssp-relax-vertices-singleton-updates-distinct:
 assumes distinct-edges: distinct (nat-graph-edge-list G)
 shows distinct
  (map fst (fst (bmssp-relax-vertices G settled [u] ds)))
 using bmssp-relax-edges-updates-distinct[OF distinct-edges]
 by (cases bmssp-lookup-dist ds u) (auto split: prod.splits)

lemma bmssp-relax-vertices-updates-distinct-if-length-le-one:
 assumes distinct-edges: distinct (nat-graph-edge-list G)
  and len: length xs ≤ 1
 shows distinct (map fst (fst (bmssp-relax-vertices G settled xs ds)))
proof (cases xs)
 case Nil
 then show ?thesis
  by simp
next
 case (Cons u us)
 then have us = []
  using len by (cases us) simp-all
 then show ?thesis
  using Cons bmssp-relax-vertices-singleton-updates-distinct[OF distinct-edges]
  by simp
qed

lemma bmssp-relax-vertices-pulled-updates-distinct:
 assumes wf: nat-graph-well-formed G
  and distinct-vertices: distinct vertices
  and pull: pull-separates (bp-view P) bmssp-block-size B S beta D'
 shows distinct
  (map fst
   (fst (bmssp-relax-vertices G settled
    (filter (λx. x ∈ S ∧ x ∉ set old-settled) vertices) ds)))
proof −
 have len:
  length (filter (λx. x ∈ S ∧ x ∉ set old-settled) vertices)
   ≤ 1
  by (rule bmssp-pulled-length-le-one[OF distinct-vertices pull])
 have distinct-edges: distinct (nat-graph-edge-list G)
  by (rule nat-graph-well-formed-distinct-edge-list[OF wf])
 show ?thesis
  by (rule bmssp-relax-vertices-updates-distinct-if-length-le-one
```

```
    [OF distinct-edges len])
qed

lemma bmssp-apply-updates-ordered-after-pull:
  fixes old-settled settled :: nat list
  assumes wf: nat-graph-well-formed G
    and distinct-vertices: distinct vertices
    and ord: bp-ordered-invariant P
    and upper: partition-upper-bound (bp-view P) B
    and pull: bp-pull bmssp-block-size B P = (S, beta, P1)
  shows bp-ordered-invariant
    (bmssp-apply-updates
      (fst (bmssp-relax-vertices G settled
        (filter (λx. x ∈ S ∧ x ∉ set old-settled) vertices) ds))
      P1)
proof −
  let ?pulled =
    filter (λx. x ∈ S ∧ x ∉ set old-settled) vertices
  let ?updates = fst (bmssp-relax-vertices G settled ?pulled ds)
  have sep: pull-separates (bp-view P) bmssp-block-size B S beta (bp-view P1)
    by (rule bmssp-pull-refines-pull-separates[OF ord upper pull])
  have P1-ord: bp-ordered-invariant P1
    by (rule bmssp-pull-ordered-invariant[OF ord pull])
  have distinct-updates: distinct (map fst ?updates)
    by (rule bmssp-relax-vertices-pulled-updates-distinct
        [OF wf distinct-vertices sep])
  show ?thesis
    by (rule bmssp-apply-updates-ordered-invariant-from-ordered
        [OF P1-ord distinct-updates])
qed

lemma bmssp-partition-key-zero-lt-bound [simp]:
  bmssp-partition-key s 0 < bmssp-bound
proof −
  have bmssp-partition-key s 0 < real (Suc 0)
    by (rule bmssp-partition-key-lt-suc-distance)
  then show ?thesis
    unfolding bmssp-bound-def bmssp-infinity-def by simp
qed

lemma bmssp-partition-key-lt-bound-if-distance-lt:
  assumes d < bmssp-infinity
  shows bmssp-partition-key v d < bmssp-bound
proof −
  have bmssp-partition-key v d < real (Suc d)
    by (rule bmssp-partition-key-lt-suc-distance)
  also have ... ≤ real bmssp-infinity
    using assms by simp
  finally show ?thesis
```

```
    unfolding bmssp-bound-def .
qed

lemma bmssp-initial-partition-bridge:
  shows bp-ordered-invariant
      (bp-result-of
        (c-bp-regularized-local-insert s (bmssp-partition-key s 0)
          (bp-empty bmssp-block-size bmssp-bound)))
    and partition-upper-bound
      (bp-view
        (bp-result-of
          (c-bp-regularized-local-insert s (bmssp-partition-key s 0)
            (bp-empty bmssp-block-size bmssp-bound))))
      bmssp-bound
    and bp-view
        (bp-result-of
          (c-bp-regularized-local-insert s (bmssp-partition-key s 0)
            (bp-empty bmssp-block-size bmssp-bound))) =
      min-update (bp-view (bp-empty bmssp-block-size bmssp-bound))
        s (bmssp-partition-key s 0)
proof −
  let ?P0 = bp-empty bmssp-block-size bmssp-bound
  let ?P1 =
    bp-result-of
      (c-bp-regularized-local-insert s (bmssp-partition-key s 0) ?P0)
  have block-pos: 0 < bmssp-block-size
    unfolding bmssp-block-size-def by simp
  have reg0: bp-regular-invariant ?P0
    by (rule bp-empty-regular-invariant[OF block-pos])
  have reg1: bp-regular-invariant ?P1
    by (rule c-bp-regularized-local-insert-regular-invariant[OF reg0])
  show bp-ordered-invariant ?P1
    by (rule bp-regular-invariant-ordered-invariant[OF reg1])
  have view:
    bp-view ?P1 = min-update (bp-view ?P0) s (bmssp-partition-key s 0)
    by (rule c-bp-regularized-local-insert-refines-min-update[OF reg0])
  show bp-view ?P1 =
      min-update (bp-view ?P0) s (bmssp-partition-key s 0)
    by (rule view)
  have upper0: partition-upper-bound (bp-view ?P0) bmssp-bound
    unfolding partition-upper-bound-def by simp
  have partition-upper-bound
      (min-update (bp-view ?P0) s (bmssp-partition-key s 0)) bmssp-bound
    by (rule min-update-preserves-upper-bound[OF upper0])
      (rule bmssp-partition-key-zero-lt-bound)
  then show partition-upper-bound (bp-view ?P1) bmssp-bound
    unfolding view .
qed
```

lemma *bmssp-loop-partition-step-bridge*:
fixes *old-settled settled vertices* :: *nat list*
and *pulled* :: *nat list*
and *updates* :: (*nat* × *real*) *list*
assumes *pulled-def*:
pulled =
filter (λ*x*. *x* ∈ *S* ∧ *x* ∉ *set old-settled*) *vertices*
and *updates-def*: *updates* = *fst* (*bmssp-relax-vertices G settled pulled ds*)
and *P2-def*: *P2* = *bmssp-apply-updates updates P1*
and *wf*: *nat-graph-well-formed G*
and *distinct-vertices*: *distinct vertices*
and *ord*: *bp-ordered-invariant P*
and *upper*: *partition-upper-bound* (*bp-view P*) *B*
and *pull*: *bp-pull bmssp-block-size B P* = (*S*, *beta*, *P1*)
shows *pull-separates* (*bp-view P*) *bmssp-block-size B S beta* (*bp-view P1*)
and *bp-ordered-invariant P1*
and *partition-upper-bound* (*bp-view P1*) *B*
and *length pulled* ≤ *1*
and *distinct* (*map fst updates*)
and (⋀*x b*. (*x*, *b*) ∈ *set updates* ⟹
∃ *d*. *b* = *bmssp-partition-key x d* ∧ *nat* (*floor b*) = *d*)
and *bp-view P2* = *batch-min-update* (*bp-view P1*) *updates*
and *bp-ordered-invariant P2*
and (⋀*x b*. (*x*, *b*) ∈ *set updates* ⟹ *b* < *B*) ⟹
partition-upper-bound (*bp-view P2*) *B*
proof −
have *sep*:
pull-separates (*bp-view P*) *bmssp-block-size B S beta* (*bp-view P1*)
by (*rule bmssp-pull-refines-pull-separates*[*OF ord upper pull*])
have *P1-ord*: *bp-ordered-invariant P1*
by (*rule bmssp-pull-ordered-invariant*[*OF ord pull*])
have *P1-upper*: *partition-upper-bound* (*bp-view P1*) *B*
by (*rule bmssp-pull-preserves-upper-bound*[*OF ord upper pull*])
have *pulled-len*: *length pulled* ≤ *1*
unfolding *pulled-def*
by (*rule bmssp-pulled-length-le-one*[*OF distinct-vertices sep*])
have *updates-distinct*: *distinct* (*map fst updates*)
unfolding *updates-def pulled-def*
by (*rule bmssp-relax-vertices-pulled-updates-distinct*
[*OF wf distinct-vertices sep*])
have *updates-floor*:
⋀*x b*. (*x*, *b*) ∈ *set updates* ⟹
∃ *d*. *b* = *bmssp-partition-key x d* ∧ *nat* (*floor b*) = *d*
proof −
fix *x b*
assume (*x*, *b*) ∈ *set updates*
then have (*x*, *b*) ∈
set (*fst* (*bmssp-relax-vertices G settled pulled ds*))
unfolding *updates-def* **.**

```
    then show ∃ d. b = bmssp-partition-key x d ∧ nat (floor b) = d
      by (rule bmssp-relax-vertices-update-floor)
  qed
  have view:
    bp-view P2 = batch-min-update (bp-view P1) updates
    unfolding P2-def
    by (rule bmssp-apply-updates-refines-batch-min-update-from-ordered
        [OF P1-ord updates-distinct])
  have P2-ord: bp-ordered-invariant P2
    unfolding P2-def
    by (rule bmssp-apply-updates-ordered-invariant-from-ordered
        [OF P1-ord updates-distinct])
  show pull-separates (bp-view P) bmssp-block-size B S beta (bp-view P1)
    by (rule sep)
  show bp-ordered-invariant P1
    by (rule P1-ord)
  show partition-upper-bound (bp-view P1) B
    by (rule P1-upper)
  show length pulled ≤ 1
    by (rule pulled-len)
  show distinct (map fst updates)
    by (rule updates-distinct)
  show ⋀x b. (x, b) ∈ set updates ⟹
      ∃ d. b = bmssp-partition-key x d ∧ nat (floor b) = d
    by (rule updates-floor)
  show bp-view P2 = batch-min-update (bp-view P1) updates
    by (rule view)
  show bp-ordered-invariant P2
    by (rule P2-ord)
  show (⋀x b. (x, b) ∈ set updates ⟹ b < B) ⟹
      partition-upper-bound (bp-view P2) B
    unfolding P2-def
    by (rule bmssp-apply-updates-preserves-upper-bound-from-ordered
        [OF P1-ord P1-upper updates-distinct])
qed

definition bmssp-partition-keys-match ::
  nat list ⇒ nat-dist ⇒ nat bucketed-partition ⇒ bool where
  bmssp-partition-keys-match settled ds P ⟷
    keys-of (bp-view P) = set (map fst ds) − set settled

lemma bmssp-partition-keys-match-initial:
  bmssp-partition-keys-match [] [(src, 0)]
    (bp-result-of
      (c-bp-regularized-local-insert src (bmssp-partition-key src 0)
        (bp-empty bmssp-block-size bmssp-bound)))
proof −
  have view:
    bp-view
```

```
    (bp-result-of
      (c-bp-regularized-local-insert src (bmssp-partition-key src 0)
        (bp-empty bmssp-block-size bmssp-bound))) =
    min-update (bp-view (bp-empty bmssp-block-size bmssp-bound))
      src (bmssp-partition-key src 0)
  by (rule bmssp-initial-partition-bridge(3))
 show ?thesis
  unfolding bmssp-partition-keys-match-def view by simp
qed

lemma bmssp-filter-pulled-set-eq:
 assumes S-vertices: S ⊆ set vertices
  and S-unsettled: S ∩ set settled = {}
 shows set (filter (λx. x ∈ S ∧ x ∉ set settled) vertices) = S
proof
 show set (filter (λx. x ∈ S ∧ x ∉ set settled) vertices)
    ⊆ S
  by auto
next
 show S ⊆
    set (filter (λx. x ∈ S ∧ x ∉ set settled) vertices)
 proof
  fix x
  assume xS: x ∈ S
  then have x ∈ set vertices
   using S-vertices by blast
  moreover have x ∉ set settled
   using S-unsettled xS by blast
  ultimately show x ∈
      set (filter (λx. x ∈ S ∧ x ∉ set settled) vertices)
   using xS by simp
 qed
qed

lemma bmssp-partition-keys-match-step:
 fixes old-settled settled vertices :: nat list
  and pulled :: nat list
  and updates :: (nat × real) list
 assumes match: bmssp-partition-keys-match old-settled ds P
  and keys-vertices: keys-of (bp-view P) ⊆ set vertices
  and pulled-def:
    pulled =
      filter (λx. x ∈ S ∧ x ∉ set old-settled) vertices
  and settled-def: settled = pulled @ old-settled
  and relaxed: bmssp-relax-vertices G settled pulled ds = (updates, ds')
  and pull: pull-separates (bp-view P) bmssp-block-size B S beta
    (bp-view P1)
  and view: bp-view P2 = batch-min-update (bp-view P1) updates
 shows bmssp-partition-keys-match settled ds' P2
```

```
proof −
  have S-subset-keys: S ⊆ keys-of (bp-view P)
    by (rule pull-separates-subset[OF pull])
  have S-vertices: S ⊆ set vertices
    using S-subset-keys keys-vertices by blast
  have S-unsettled: S ∩ set old-settled = {}
    using S-subset-keys match unfolding bmssp-partition-keys-match-def by auto
  have pulled-set: set pulled = S
    unfolding pulled-def
    by (rule bmssp-filter-pulled-set-eq[OF S-vertices S-unsettled])
  have settled-set: set settled = S ∪ set old-settled
    unfolding settled-def using pulled-set by auto
  have P1-keys: keys-of (bp-view P1) = keys-of (bp-view P) − S
    by (rule pull-separates-remaining-keys[OF pull])
  have P2-keys:
    keys-of (bp-view P2) =
      (set (map fst ds) − set old-settled − S) ∪ fst ‘ set updates
    using match view P1-keys
    unfolding bmssp-partition-keys-match-def
    by (auto simp: batch-min-update-keys)
  have ds'-keys:
    set (map fst ds') = set (map fst ds) ∪ fst ‘ set updates
    using bmssp-relax-vertices-dist-keys[of G settled pulled ds] relaxed
    by simp
  have updates-unsettled:
    fst ‘ set updates ∩ set settled = {}
  proof
    show fst ‘ set updates ∩ set settled ⊆ {}
    proof
      fix x
      assume x: x ∈ fst ‘ set updates ∩ set settled
      then obtain b where xb: (x, b) ∈ set updates
        by auto
      have (x, b) ∈ set (fst (bmssp-relax-vertices G settled pulled ds))
        using xb relaxed by simp
      then have x ∉ set settled
        by (rule bmssp-relax-vertices-update-not-settled)
      then show x ∈ {}
        using x by simp
    qed
    show {} ⊆ fst ‘ set updates ∩ set settled
      by simp
  qed
  have set (map fst ds') − set settled =
      (set (map fst ds) − set old-settled − S) ∪ fst ‘ set updates
    using ds'-keys settled-set updates-unsettled by auto
  then show ?thesis
    using P2-keys unfolding bmssp-partition-keys-match-def by simp
qed
```

lemma *bmssp-partition-keys-match-loop-step-bridge*:
fixes *old-settled settled vertices* :: *nat list*
and *pulled* :: *nat list*
and *updates* :: (*nat* × *real*) *list*
assumes *match*: *bmssp-partition-keys-match old-settled ds P*
and *keys-vertices*: *keys-of* (*bp-view P*) ⊆ *set vertices*
and *pulled-def*:
pulled =
filter (λ*x*. *x* ∈ *S* ∧ *x* ∉ *set old-settled*) *vertices*
and *settled-def*: *settled* = *pulled* @ *old-settled*
and *relaxed*: *bmssp-relax-vertices G settled pulled ds* = (*updates*, *ds'*)
and *P2-def*: *P2* = *bmssp-apply-updates updates P1*
and *wf*: *nat-graph-well-formed G*
and *distinct-vertices*: *distinct vertices*
and *ord*: *bp-ordered-invariant P*
and *upper*: *partition-upper-bound* (*bp-view P*) *B*
and *pull*: *bp-pull bmssp-block-size B P* = (*S*, *beta*, *P1*)
shows *bmssp-partition-keys-match settled ds' P2*
proof −
have *updates-def*: *updates* = *fst* (*bmssp-relax-vertices G settled pulled ds*)
using *relaxed* **by** *simp*
have *sep*:
pull-separates (*bp-view P*) *bmssp-block-size B S beta* (*bp-view P1*)
by (*rule bmssp-loop-partition-step-bridge*(*1*)
[*OF pulled-def updates-def P2-def wf distinct-vertices ord upper pull*])
have *view*: *bp-view P2* = *batch-min-update* (*bp-view P1*) *updates*
by (*rule bmssp-loop-partition-step-bridge*(*7*)
[*OF pulled-def updates-def P2-def wf distinct-vertices ord upper pull*])
show *?thesis*
by (*rule bmssp-partition-keys-match-step*
[*OF match keys-vertices pulled-def settled-def relaxed sep view*])
qed

lemma *batch-min-update-value-not-updated*:
assumes *x* ∉ *fst* ‘ *set xs*
shows *value-of* (*batch-min-update D xs*) *x* = *value-of D x*
using *assms*
proof (*induction xs arbitrary*: *D*)
case *Nil*
then show *?case*
unfolding *batch-min-update-def* **by** *simp*
next
case (*Cons xb xs*)
obtain *y b* **where** *xb*: *xb* = (*y*, *b*)
by *force*
have *x-ne-y*: *x* ≠ *y*
using *Cons.prems* **unfolding** *xb* **by** *simp*
have *x-not-tail*: *x* ∉ *fst* ‘ *set xs*

```
    using Cons.prems unfolding xb by simp
  have tail:
    value-of (batch-min-update (min-update D y b) xs) x =
      value-of (min-update D y b) x
    by (rule Cons.IH[OF x-not-tail])
  have step: value-of (min-update D y b) x = value-of D x
    using min-update-value-same[OF x-ne-y, of D b] .
  show ?case
  proof −
    have value-of (batch-min-update D (xb # xs)) x =
        value-of (batch-min-update (min-update D y b) xs) x
      unfolding batch-min-update-def xb by simp
    also have ... = value-of (min-update D y b) x
      by (rule tail)
    also have ... = value-of D x
      by (rule step)
    finally show ?case .
  qed
qed

lemma batch-min-update-value-pair-less-old:
  assumes distinct: distinct (map fst xs)
    and pair: (x, b) ∈ set xs
    and better: x ∈ keys-of D ⟹ b < value-of D x
  shows value-of (batch-min-update D xs) x = b
  using distinct pair better
proof (induction xs arbitrary: D)
  case Nil
  then show ?case
    by simp
next
  case (Cons xb xs)
  obtain y c where xb: xb = (y, c)
    by force
  have distinct-tail: distinct (map fst xs)
    using Cons.prems(1) unfolding xb by simp
  show ?case
  proof (cases x = y)
    case True
    have x-not-tail: x ∉ fst ‘ set xs
      using Cons.prems(1) True unfolding xb by simp
    have not-pair-tail: (x, b) ∉ set xs
      using x-not-tail by force
    have c-eq: c = b
    proof −
      have (x, b) = (y, c)
        using Cons.prems(2) not-pair-tail unfolding xb by auto
      then show ?thesis
        using True by simp
```

```
  qed
  have step: value-of (min-update D y c) x = b
  proof (cases x ∈ keys-of D)
    case True
    then have c < value-of D x
      using Cons.prems(3) c-eq by simp
    then show ?thesis
      using True ‹x = y› c-eq min-update-value-key-old[of x D c]
      by simp
  next
    case False
    then show ?thesis
      using ‹x = y› c-eq min-update-value-key-new[of y D c]
      by simp
  qed
  have tail:
    value-of (batch-min-update (min-update D y c) xs) x =
      value-of (min-update D y c) x
    by (rule batch-min-update-value-not-updated[OF x-not-tail])
  show ?thesis
  proof −
    have value-of (batch-min-update D (xb # xs)) x =
        value-of (batch-min-update (min-update D y c) xs) x
      unfolding batch-min-update-def xb by simp
    also have ... = value-of (min-update D y c) x
      by (rule tail)
    also have ... = b
      by (rule step)
    finally show ?thesis .
  qed
next
  case False
  have pair-tail: (x, b) ∈ set xs
    using Cons.prems(2) False unfolding xb by auto
  have better-tail:
    x ∈ keys-of (min-update D y c) ⟹
      b < value-of (min-update D y c) x
  proof −
    assume x-in: x ∈ keys-of (min-update D y c)
    then have x ∈ keys-of D
      using False by simp
    then have b < value-of D x
      by (rule Cons.prems(3))
    moreover have value-of (min-update D y c) x = value-of D x
      using min-update-value-same[OF False, of D c] .
    ultimately show b < value-of (min-update D y c) x
      by simp
  qed
  have tail:
```

```
      value-of (batch-min-update (min-update D y c) xs) x = b
      by (rule Cons.IH[OF distinct-tail pair-tail better-tail])
    show ?thesis
    proof −
      have value-of (batch-min-update D (xb # xs)) x =
          value-of (batch-min-update (min-update D y c) xs) x
        unfolding batch-min-update-def xb by simp
      also have ... = b
        by (rule tail)
      finally show ?thesis .
    qed
  qed
qed

definition bmssp-partition-values-match ::
  nat list ⇒ nat-dist ⇒ nat partition-view ⇒ bool where
  bmssp-partition-values-match settled ds D ⟷
    (∀ v d. bmssp-lookup-dist ds v = Some d ⟶
      v ∉ set settled ⟶ value-of D v = bmssp-partition-key v d)

lemma bmssp-partition-values-match-initial:
  bmssp-partition-values-match [] [(src, 0)]
    (bp-view
      (bp-result-of
        (c-bp-regularized-local-insert src (bmssp-partition-key src 0)
          (bp-empty bmssp-block-size bmssp-bound))))
proof −
  have view:
    bp-view
      (bp-result-of
        (c-bp-regularized-local-insert src (bmssp-partition-key src 0)
          (bp-empty bmssp-block-size bmssp-bound))) =
      min-update (bp-view (bp-empty bmssp-block-size bmssp-bound))
        src (bmssp-partition-key src 0)
    by (rule bmssp-initial-partition-bridge(3))
  show ?thesis
    unfolding bmssp-partition-values-match-def view
    by (auto simp: min-update-def)
qed

lemma bmssp-partition-values-match-pull:
  bmssp-partition-values-match settled ds D ⟹
    value-of Dp = value-of D ⟹
    bmssp-partition-values-match settled ds Dp
  unfolding bmssp-partition-values-match-def by simp

lemma bmssp-partition-values-match-step:
  fixes old-settled settled vertices :: nat list
    and pulled :: nat list
```

```
    and updates :: (nat × real) list
  assumes vals: bmssp-partition-values-match old-settled ds (bp-view P)
    and keys: bmssp-partition-keys-match old-settled ds P
    and distinct-ds: distinct (map fst ds)
    and pulled-def:
      pulled =
        filter (λx. x ∈ S ∧ x ∉ set old-settled) vertices
    and settled-def: settled = pulled @ old-settled
    and relaxed: bmssp-relax-vertices G settled pulled ds = (updates, ds')
    and distinct-updates: distinct (map fst updates)
    and pull: pull-separates (bp-view P) bmssp-block-size B S beta
      (bp-view P1)
    and view: bp-view P2 = batch-min-update (bp-view P1) updates
  shows bmssp-partition-values-match settled ds' (bp-view P2)
proof −
  have pull-values: value-of (bp-view P1) = value-of (bp-view P)
    using pull unfolding pull-separates-def by simp
  show ?thesis
    unfolding bmssp-partition-values-match-def
  proof (intro allI impI)
    fix v d
    assume lookup': bmssp-lookup-dist ds' v = Some d
    assume v-unsettled: v ∉ set settled
    show value-of (bp-view P2) v = bmssp-partition-key v d
    proof (cases v ∈ fst ‘ set updates)
      case True
      then obtain b where update: (v, b) ∈ set updates
        by auto
      have update-rel:
        (v, b) ∈ set (fst (bmssp-relax-vertices G settled pulled ds))
        using update relaxed by simp
      have distinct-rel:
        distinct (map fst (fst (bmssp-relax-vertices G settled pulled ds)))
        using distinct-updates relaxed by simp
      have lookup-floor:
        bmssp-lookup-dist ds' v = Some (nat (floor b))
        using bmssp-relax-vertices-update-lookup-floor
            [OF distinct-rel update-rel] relaxed by simp
      have d-floor: d = nat (floor b)
        using lookup' lookup-floor by simp
      obtain du where b-key: b = bmssp-partition-key v du
        and du-floor: nat (floor b) = du
        using bmssp-relax-vertices-update-floor[OF update-rel] by blast
      have b-d: b = bmssp-partition-key v d
        using b-key du-floor d-floor by simp
      have better:
        v ∈ keys-of (bp-view P1) ⟹ b < value-of (bp-view P1) v
      proof −
        assume vP1: v ∈ keys-of (bp-view P1)
```

```
    have P1-keys: keys-of (bp-view P1) = keys-of (bp-view P) − S
      by (rule pull-separates-remaining-keys[OF pull])
    have vP: v ∈ keys-of (bp-view P)
      using vP1 P1-keys by blast
    have v-key-ds: v ∈ set (map fst ds)
      using keys vP unfolding bmssp-partition-keys-match-def by blast
    have v-not-old: v ∉ set old-settled
      using keys vP unfolding bmssp-partition-keys-match-def by blast
    obtain old where old-pair: (v, old) ∈ set ds
      using v-key-ds by force
    have old-lookup: bmssp-lookup-dist ds v = Some old
      by (rule bmssp-lookup-dist-Some-if-distinct-mem
          [OF distinct-ds old-pair])
    have old-value:
      value-of (bp-view P) v = bmssp-partition-key v old
      using vals old-lookup v-not-old
      unfolding bmssp-partition-values-match-def by blast
    have du-lt-old: du < old
      using bmssp-relax-vertices-update-improves-lookup
          [OF distinct-ds update-rel old-lookup] du-floor by simp
    have bmssp-partition-key v du < bmssp-partition-key v old
      by (rule bmssp-partition-key-strict-mono-distance[OF du-lt-old])
    then have b < value-of (bp-view P) v
      using b-key old-value by simp
    then show b < value-of (bp-view P1) v
      using pull-values by simp
  qed
  have value-of (bp-view P2) v = b
    unfolding view
    by (rule batch-min-update-value-pair-less-old
        [OF distinct-updates update better])
  then show ?thesis
    using b-d by simp
next
  case False
  have lookup-same: bmssp-lookup-dist ds' v = bmssp-lookup-dist ds v
    using bmssp-relax-vertices-lookup-not-updated[of v G settled pulled ds]
      False relaxed by simp
  have lookup-old: bmssp-lookup-dist ds v = Some d
    using lookup' lookup-same by simp
  have v-not-old: v ∉ set old-settled
    using v-unsettled settled-def by auto
  have old-value:
    value-of (bp-view P) v = bmssp-partition-key v d
    using vals lookup-old v-not-old
    unfolding bmssp-partition-values-match-def by blast
  have p2-p1: value-of (bp-view P2) v = value-of (bp-view P1) v
    unfolding view
    by (rule batch-min-update-value-not-updated[OF False])
```

```
    show ?thesis
      using p2-p1 pull-values old-value by simp
  qed
 qed
qed

lemma bmssp-partition-values-match-loop-step-bridge:
 fixes old-settled settled vertices :: nat list
   and pulled :: nat list
   and updates :: (nat × real) list
 assumes vals: bmssp-partition-values-match old-settled ds (bp-view P)
   and keys: bmssp-partition-keys-match old-settled ds P
   and distinct-ds: distinct (map fst ds)
   and pulled-def:
     pulled =
       filter (λx. x ∈ S ∧ x ∉ set old-settled) vertices
   and settled-def: settled = pulled @ old-settled
   and relaxed: bmssp-relax-vertices G settled pulled ds = (updates, ds')
   and P2-def: P2 = bmssp-apply-updates updates P1
   and wf: nat-graph-well-formed G
   and distinct-vertices: distinct vertices
   and ord: bp-ordered-invariant P
   and upper: partition-upper-bound (bp-view P) B
   and pull: bp-pull bmssp-block-size B P = (S, beta, P1)
 shows bmssp-partition-values-match settled ds' (bp-view P2)
proof −
 have updates-def: updates = fst (bmssp-relax-vertices G settled pulled ds)
   using relaxed by simp
 have sep:
   pull-separates (bp-view P) bmssp-block-size B S beta (bp-view P1)
   by (rule bmssp-loop-partition-step-bridge(1)
       [OF pulled-def updates-def P2-def wf distinct-vertices ord upper pull])
 have distinct-updates: distinct (map fst updates)
   by (rule bmssp-loop-partition-step-bridge(5)
       [OF pulled-def updates-def P2-def wf distinct-vertices ord upper pull])
 have view: bp-view P2 = batch-min-update (bp-view P1) updates
   by (rule bmssp-loop-partition-step-bridge(7)
       [OF pulled-def updates-def P2-def wf distinct-vertices ord upper pull])
 show ?thesis
   by (rule bmssp-partition-values-match-step
       [OF vals keys distinct-ds pulled-def settled-def relaxed
        distinct-updates sep view])
qed

lemma bmssp-partition-key-le-imp-distance-le:
 assumes bmssp-partition-key u du ≤ bmssp-partition-key v dv
 shows du ≤ dv
proof (rule ccontr)
 assume ¬ du ≤ dv
```

```
  then have dv < du
    by simp
  then have bmssp-partition-key v dv < bmssp-partition-key u du
    by (rule bmssp-partition-key-strict-mono-distance)
  then show False
    using assms by linarith
qed

lemma bmssp-pull-residual-label-le:
  assumes vals: bmssp-partition-values-match old-settled ds (bp-view P)
    and keys: bmssp-partition-keys-match old-settled ds P
    and pull: pull-separates (bp-view P) bmssp-block-size B S beta
      (bp-view P1)
    and uS: u ∈ S
    and u-lookup: bmssp-lookup-dist ds u = Some du
    and v-lookup: bmssp-lookup-dist ds v = Some dv
    and v-unsettled: v ∉ set old-settled
    and v-not-S: v ∉ S
  shows du ≤ dv
proof −
  have S-keys: S ⊆ keys-of (bp-view P)
    by (rule pull-separates-subset[OF pull])
  have u-unsettled: u ∉ set old-settled
    using keys S-keys uS unfolding bmssp-partition-keys-match-def by blast
  have u-value: value-of (bp-view P) u = bmssp-partition-key u du
    using vals u-lookup u-unsettled
    unfolding bmssp-partition-values-match-def by blast
  have P1-keys: keys-of (bp-view P1) = keys-of (bp-view P) − S
    by (rule pull-separates-remaining-keys[OF pull])
  have v-key: v ∈ set (map fst ds)
    using v-lookup by (rule bmssp-lookup-dist-mem-key)
  have vP: v ∈ keys-of (bp-view P)
    using keys v-key v-unsettled unfolding bmssp-partition-keys-match-def
    by blast
  have vP1: v ∈ keys-of (bp-view P1)
    using P1-keys vP v-not-S by blast
  have v-value: value-of (bp-view P) v = bmssp-partition-key v dv
    using vals v-lookup v-unsettled
    unfolding bmssp-partition-values-match-def by blast
  have pull-values: value-of (bp-view P1) = value-of (bp-view P)
    using pull unfolding pull-separates-def by simp
  have value-of (bp-view P) u ≤ value-of (bp-view P1) v
    by (rule pull-separates-pulled-smallest[OF pull uS vP1])
  then have bmssp-partition-key u du ≤ bmssp-partition-key v dv
    using u-value v-value pull-values by simp
  then show ?thesis
    by (rule bmssp-partition-key-le-imp-distance-le)
qed
```

definition *bmssp-partition-state-match* ::
  nat list ⇒ *nat list* ⇒ *nat-dist* ⇒ *nat bucketed-partition* ⇒ *bool* **where**
  bmssp-partition-state-match vertices settled ds P ⟷
    distinct (*map fst ds*) ∧
    set (*map fst ds*) ⊆ *set vertices* ∧
    bmssp-partition-keys-match settled ds P ∧
    bmssp-partition-values-match settled ds (*bp-view P*)

lemma *bmssp-partition-state-match-initial*:
  assumes *src-vertices*: *src* ∈ *set vertices*
  shows *bmssp-partition-state-match vertices* [] [(*src*, *0*)]
    (*bp-result-of*
      (*c-bp-regularized-local-insert src* (*bmssp-partition-key src 0*)
        (*bp-empty bmssp-block-size bmssp-bound*)))
proof −
  have *distinct*: *distinct* (*map fst* [(*src*, *0*)])
    by *simp*
  have *keys-subset*: *set* (*map fst* [(*src*, *0*)]) ⊆ *set vertices*
    using *src-vertices* **by** *simp*
  have *keys*:
    bmssp-partition-keys-match [] [(*src*, *0*)]
      (*bp-result-of*
        (*c-bp-regularized-local-insert src* (*bmssp-partition-key src 0*)
          (*bp-empty bmssp-block-size bmssp-bound*)))
    by (*rule bmssp-partition-keys-match-initial*)
  have *vals*:
    bmssp-partition-values-match [] [(*src*, *0*)]
      (*bp-view*
        (*bp-result-of*
          (*c-bp-regularized-local-insert src* (*bmssp-partition-key src 0*)
            (*bp-empty bmssp-block-size bmssp-bound*))))
    by (*rule bmssp-partition-values-match-initial*)
  show *?thesis*
    unfolding *bmssp-partition-state-match-def*
    using *distinct keys-subset keys vals* **by** *simp*
qed

lemma *bmssp-partition-state-match-keys-subset*:
  assumes *bmssp-partition-state-match vertices settled ds P*
  shows *keys-of* (*bp-view P*) ⊆ *set vertices*
proof −
  have *keys*: *bmssp-partition-keys-match settled ds P*
    and *ds-keys*: *set* (*map fst ds*) ⊆ *set vertices*
    using *assms* **unfolding** *bmssp-partition-state-match-def* **by** *auto*
  show *?thesis*
    using *keys ds-keys* **unfolding** *bmssp-partition-keys-match-def* **by** *blast*
qed

lemma *bmssp-partition-state-match-step-bridge*:

```
  fixes old-settled settled vertices :: nat list
    and pulled :: nat list
    and updates :: (nat × real) list
  assumes match: bmssp-partition-state-match vertices old-settled ds P
    and edge-targets:
      ⋀a b w. (a, b, w) ∈ set G ⟹ b ∈ set vertices
    and pulled-def:
      pulled =
        filter (λx. x ∈ S ∧ x ∉ set old-settled) vertices
    and settled-def: settled = pulled @ old-settled
    and relaxed: bmssp-relax-vertices G settled pulled ds = (updates, ds')
    and P2-def: P2 = bmssp-apply-updates updates P1
    and wf: nat-graph-well-formed G
    and distinct-vertices: distinct vertices
    and ord: bp-ordered-invariant P
    and upper: partition-upper-bound (bp-view P) B
    and pull: bp-pull bmssp-block-size B P = (S, beta, P1)
  shows bmssp-partition-state-match vertices settled ds' P2
proof −
  have distinct-ds: distinct (map fst ds)
    and ds-keys: set (map fst ds) ⊆ set vertices
    and keys: bmssp-partition-keys-match old-settled ds P
    and vals: bmssp-partition-values-match old-settled ds (bp-view P)
    using match unfolding bmssp-partition-state-match-def by auto
  have pulled-subset: set pulled ⊆ set vertices
    unfolding pulled-def by auto
  have distinct-ds': distinct (map fst ds')
    using bmssp-relax-vertices-preserves-distinct-dist
      [OF distinct-ds, of G settled pulled] relaxed by simp
  have ds'-keys: set (map fst ds') ⊆ set vertices
  proof −
    have set (map fst (snd (bmssp-relax-vertices G settled pulled ds)))
      ⊆ set vertices
      by (rule bmssp-relax-vertices-dist-keys-subset
        [OF ds-keys pulled-subset edge-targets])
    then show ?thesis
      using relaxed by simp
  qed
  have keys-vertices: keys-of (bp-view P) ⊆ set vertices
    by (rule bmssp-partition-state-match-keys-subset[OF match])
  have keys':
    bmssp-partition-keys-match settled ds' P2
    by (rule bmssp-partition-keys-match-loop-step-bridge
      [OF keys keys-vertices pulled-def settled-def relaxed P2-def
        wf distinct-vertices ord upper pull])
  have vals':
    bmssp-partition-values-match settled ds' (bp-view P2)
    by (rule bmssp-partition-values-match-loop-step-bridge
      [OF vals keys distinct-ds pulled-def settled-def relaxed P2-def
```

```
      wf distinct-vertices ord upper pull])
  show ?thesis
    unfolding bmssp-partition-state-match-def
    using distinct-ds' ds'-keys keys' vals' by simp
qed

lemma bmssp-partition-state-pull-residual-label-le:
  assumes match: bmssp-partition-state-match vertices old-settled ds P
    and pull: pull-separates (bp-view P) bmssp-block-size B S beta
      (bp-view P1)
    and uS: u ∈ S
    and u-lookup: bmssp-lookup-dist ds u = Some du
    and v-lookup: bmssp-lookup-dist ds v = Some dv
    and v-unsettled: v ∉ set old-settled
    and v-not-S: v ∉ S
  shows du ≤ dv
proof −
  have vals: bmssp-partition-values-match old-settled ds (bp-view P)
    and keys: bmssp-partition-keys-match old-settled ds P
    using match unfolding bmssp-partition-state-match-def by auto
  show ?thesis
    by (rule bmssp-pull-residual-label-le
        [OF vals keys pull uS u-lookup v-lookup v-unsettled v-not-S])
qed

lemma bmssp-partition-state-pulled-not-settled:
  assumes match: bmssp-partition-state-match vertices settled ds P
    and pull: pull-separates (bp-view P) bmssp-block-size B S beta D'
    and uS: u ∈ S
  shows u ∉ set settled
proof −
  have keys: bmssp-partition-keys-match settled ds P
    using match unfolding bmssp-partition-state-match-def by blast
  have S-keys: S ⊆ keys-of (bp-view P)
    by (rule pull-separates-subset[OF pull])
  show ?thesis
    using keys S-keys uS unfolding bmssp-partition-keys-match-def by blast
qed

lemma bmssp-partition-state-pulled-lookup:
  assumes match: bmssp-partition-state-match vertices settled ds P
    and pull: pull-separates (bp-view P) bmssp-block-size B S beta D'
    and uS: u ∈ S
  obtains d where bmssp-lookup-dist ds u = Some d
proof −
  have distinct-ds: distinct (map fst ds)
    and keys: bmssp-partition-keys-match settled ds P
    using match unfolding bmssp-partition-state-match-def by auto
  have S-keys: S ⊆ keys-of (bp-view P)
```

```
    by (rule pull-separates-subset[OF pull])
  have u-key: u ∈ set (map fst ds)
    using keys S-keys uS unfolding bmssp-partition-keys-match-def by blast
  then obtain d where mem: (u, d) ∈ set ds
    by force
  have bmssp-lookup-dist ds u = Some d
    by (rule bmssp-lookup-dist-Some-if-distinct-mem[OF distinct-ds mem])
  then show ?thesis
    by (rule that)
qed

lemma bmssp-partition-state-pulled-list-lookup:
  assumes match: bmssp-partition-state-match vertices old-settled ds P
    and pull: pull-separates (bp-view P) bmssp-block-size B S beta D'
    and pulled-def:
      pulled =
        filter (λx. x ∈ S ∧ x ∉ set old-settled) vertices
    and u-pulled: u ∈ set pulled
  obtains d where bmssp-lookup-dist ds u = Some d
proof −
  have uS: u ∈ S
    using u-pulled unfolding pulled-def by auto
  show ?thesis
    by (rule bmssp-partition-state-pulled-lookup
        [OF match pull uS that])
qed

lemma nat-graph-edge-in-vertices:
  assumes (u, v) ∈ nat-graph-edges G
  shows u ∈ nat-graph-vertices G v ∈ nat-graph-vertices G
  using assms
  unfolding nat-graph-edges-def nat-graph-edge-list-def nat-graph-vertices-def
  by auto

lemma nat-graph-weight-nonneg [simp]:
  0 ≤ nat-graph-weight G u v
proof (cases map-of
    (map (λe. ((nat-edge-source e, nat-edge-target e),
          real (nat-edge-weight e))) G) (u, v))
  case None
  then show ?thesis
    unfolding nat-graph-weight-def by simp
next
  case (Some c)
  then have ((u, v), c) ∈ set
      (map (λe. ((nat-edge-source e, nat-edge-target e),
          real (nat-edge-weight e))) G)
    by (rule map-of-SomeD)
  then show ?thesis
```

using *Some* **unfolding** *nat-graph-weight-def* **by** *auto*
qed

lemma *bmssp-edge-vertices-set*:
set (*concat* (*map bmssp-edge-vertices G*)) = *nat-graph-vertices G*
unfolding *nat-graph-vertices-def*
by (*induction G*) (*auto split*: *prod.splits*)

lemma *bmssp-vertices-set*:
set (*bmssp-vertices G src*) = *insert src* (*nat-graph-vertices G*)
unfolding *bmssp-vertices-def* **using** *bmssp-edge-vertices-set*[*of G*] **by** *auto*

lemma *bmssp-vertices-set-if-source*:
assumes *src* ∈ *nat-graph-vertices G*
shows *set* (*bmssp-vertices G src*) = *nat-graph-vertices G*
using *bmssp-vertices-set*[*of G src*] *assms* **by** *auto*

lemma *bmssp-edge-target-in-vertices*:
assumes (*a*, *b*, *w*) ∈ *set G*
shows *b* ∈ *set* (*bmssp-vertices G src*)
proof −
have *b* ∈ *nat-graph-vertices G*
using *assms* **unfolding** *nat-graph-vertices-def* **by** *force*
then show *?thesis*
using *bmssp-vertices-set*[*of G src*] **by** *auto*
qed

lemma *bmssp-lookup-dist-Some-mem*:
assumes *bmssp-lookup-dist ds v* = *Some d*
shows (*v*, *d*) ∈ *set ds*
using *assms*
proof (*induction ds*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons p ds*)
obtain *x e* **where** *p*: *p* = (*x*, *e*)
by (*cases p*)
show *?case*
proof (*cases v* = *x*)
case *True*
then show *?thesis*
using *Cons.prems* **unfolding** *p* **by** *simp*
next
case *False*
then have *bmssp-lookup-dist ds v* = *Some d*
using *Cons.prems* **unfolding** *p* **by** *simp*
then have (*v*, *d*) ∈ *set ds*

```
    by (rule Cons.IH)
  then show ?thesis
    unfolding p by simp
 qed
qed

lemma bmssp-set-dist-mem-cases:
 assumes (v, d) ∈ set (bmssp-set-dist x dx ds)
 shows (v = x ∧ d = dx) ∨ (v, d) ∈ set ds
 using assms
proof (induction ds)
 case Nil
 then show ?case
  by simp
next
 case (Cons p ds)
 obtain y e where p: p = (y, e)
  by (cases p)
 show ?case
 proof (cases x = y)
  case True
  then show ?thesis
   using Cons.prems unfolding p by auto
 next
  case False
  then have mem:
   (v, d) = (y, e) ∨ (v, d) ∈ set (bmssp-set-dist x dx ds)
   using Cons.prems unfolding p by auto
  then show ?thesis
  proof
   assume (v, d) = (y, e)
   then show ?thesis
    using False unfolding p by auto
  next
   assume (v, d) ∈ set (bmssp-set-dist x dx ds)
   then show ?thesis
    using Cons.IH by auto
  qed
 qed
qed

lemma nat-graph-weight-of-edge:
 assumes wf: nat-graph-well-formed G
  and edge: (u, v, w) ∈ set G
 shows nat-graph-weight G u v = real w
proof −
 let ?xs =
  map (λe. ((nat-edge-source e, nat-edge-target e),
    real (nat-edge-weight e))) G
```

```
  have distinct-edges: distinct (nat-graph-edge-list G)
    by (rule nat-graph-well-formed-distinct-edge-list[OF wf])
  have fst-xs: map fst ?xs = nat-graph-edge-list G
    unfolding nat-graph-edge-list-def by (induction G) auto
  have distinct-xs: distinct (map fst ?xs)
    using distinct-edges unfolding fst-xs .
  have mem: ((u, v), real w) ∈ set ?xs
    using edge by force
  have map-of ?xs (u, v) = Some (real w)
    using map-of-eq-Some-iff[OF distinct-xs] mem by blast
  then show ?thesis
    unfolding nat-graph-weight-def by simp
qed

lemma exec-walk-append-edge:
  assumes walk: exec-walk vs es p
    and p-ne: p ≠ []
    and last-p: last p = u
    and vV: v ∈ set vs
    and edge: (u, v) ∈ set es
  shows exec-walk vs es (p @ [v])
proof −
  obtain x xs where p-def: p = x # xs
    using p-ne by (cases p) auto
  have aux:
    ⋀x u. ⟦exec-walk vs es (x # xs); last (x # xs) = u;
      (u, v) ∈ set es⟧ ⟹
      exec-walk vs es ((x # xs) @ [v])
  proof (induction xs)
    case Nil
    then show ?case
      using vV by simp
  next
    case (Cons y ys)
    have tail-walk: exec-walk vs es (y # ys)
      using Cons.prems by simp
    have tail-last: last (y # ys) = u
      using Cons.prems by simp
    have tail-app: exec-walk vs es ((y # ys) @ [v])
      by (rule Cons.IH[OF tail-walk tail-last Cons.prems(3)])
    show ?case
      using Cons.prems tail-app by simp
  qed
  show ?thesis
    unfolding p-def
    by (rule aux[OF - - edge]) (use walk last-p p-def in simp-all)
qed

lemma exec-walk-weight-append-edge:
```

```
  assumes p ≠ []
  shows exec-walk-weight W (p @ [v]) =
    exec-walk-weight W p + W (last p) v
  using assms
proof (induction W p rule: exec-walk-weight.induct)
  case (1 W)
  then show ?case
    by simp
next
  case (2 W x)
  then show ?case
    by simp
next
  case (3 W x y xs)
  have tail: exec-walk-weight W ((y # xs) @ [v]) =
      exec-walk-weight W (y # xs) + W (last (y # xs)) v
    by (rule 3.IH) simp
  then show ?case
    by simp
qed

fun exec-path-edges where
  exec-path-edges [] = []
| exec-path-edges [x] = []
| exec-path-edges (x # y # xs) = (x, y) # exec-path-edges (y # xs)

lemma exec-walk-path-edges-subset:
  assumes exec-walk vs es p
  shows set (exec-path-edges p) ⊆ set es
  using assms by (induction p rule: exec-path-edges.induct) auto

lemma exec-path-edges-vertices:
  assumes e ∈ set (exec-path-edges p)
  shows fst e ∈ set p snd e ∈ set p
  using assms by (induction p rule: exec-path-edges.induct) auto

lemma exec-path-edges-distinct-if-distinct:
  assumes distinct p
  shows distinct (exec-path-edges p)
  using assms
proof (induction p rule: exec-path-edges.induct)
  case 1
  then show ?case
    by simp
next
  case (2 x)
  then show ?case
    by simp
next
```

```
  case (3 x y xs)
  have (x, y) ∉ set (exec-path-edges (y # xs))
  proof
    assume edge: (x, y) ∈ set (exec-path-edges (y # xs))
    then have x ∈ set (y # xs)
      using exec-path-edges-vertices(1)[OF edge] by simp
    then show False
      using 3.prems by simp
  qed
  moreover have distinct (exec-path-edges (y # xs))
    using 3.IH 3.prems by simp
  ultimately show ?case
    by simp
qed

lemma exec-walk-weight-sum-path-edges:
  exec-walk-weight W p =
    sum-list (map (λe. W (fst e) (snd e)) (exec-path-edges p))
  by (induction p rule: exec-path-edges.induct) simp-all

lemma nat-graph-weight-edge-list-sum:
  assumes wf: nat-graph-well-formed G
  shows sum-list
      (map (λe. nat-graph-weight G (fst e) (snd e))
        (nat-graph-edge-list G)) =
    real (nat-graph-total-weight G)
proof −
  have weights:
    ⋀e. e ∈ set G ⟹
      nat-graph-weight G (nat-edge-source e) (nat-edge-target e) =
        real (nat-edge-weight e)
  proof −
    fix e
    assume e: e ∈ set G
    obtain u v w where e-def: e = (u, v, w)
      by (cases e)
    show nat-graph-weight G (nat-edge-source e) (nat-edge-target e) =
        real (nat-edge-weight e)
      unfolding e-def
      by (rule nat-graph-weight-of-edge[OF wf]) (use e e-def in simp)
  qed
  have sum-list
      (map (λe. nat-graph-weight G (fst e) (snd e))
        (nat-graph-edge-list G)) =
    sum-list
      (map (λe. nat-graph-weight G (nat-edge-source e)
        (nat-edge-target e)) G)
    unfolding nat-graph-edge-list-def by (simp add: o-def)
  also have ... = sum-list (map (λe. real (nat-edge-weight e)) G)
```

```
  proof −
    have aux: ⋀xs. set xs ⊆ set G ⟹
      sum-list (map (λe. nat-graph-weight G (nat-edge-source e)
          (nat-edge-target e)) xs) =
      sum-list (map (λe. real (nat-edge-weight e)) xs)
    proof −
      fix xs
      assume set xs ⊆ set G
      then show sum-list (map (λe. nat-graph-weight G
            (nat-edge-source e) (nat-edge-target e)) xs) =
          sum-list (map (λe. real (nat-edge-weight e)) xs)
        by (induction xs) (auto simp: weights)
    qed
    show ?thesis
      by (rule aux) simp
  qed
  also have ... = real (sum-list (map nat-edge-weight G))
    by (induction G) auto
  finally show ?thesis
    unfolding nat-graph-total-weight-def .
qed

lemma nat-graph-weight-nonnegative:
  0 ≤ nat-graph-weight G u v
proof (cases map-of
    (map (λe. ((nat-edge-source e, nat-edge-target e),
          real (nat-edge-weight e))) G) (u, v))
  case None
  then show ?thesis
    unfolding nat-graph-weight-def by simp
next
  case (Some c)
  then have ((u, v), c) ∈ set
      (map (λe. ((nat-edge-source e, nat-edge-target e),
          real (nat-edge-weight e))) G)
    by (rule map-of-SomeD)
  then obtain e where e ∈ set G
    and c: c = real (nat-edge-weight e)
    by auto
  then show ?thesis
    unfolding nat-graph-weight-def using Some by simp
qed

lemma exec-walk-weight-nat-graph-le-total:
  assumes wf: nat-graph-well-formed G
    and walk: exec-walk vs (nat-graph-edge-list G) p
    and distinct: distinct p
  shows exec-walk-weight (nat-graph-weight G) p ≤
    real (nat-graph-total-weight G)
```

```
proof −
  let ?w = λe. nat-graph-weight G (fst e) (snd e)
  have path-subset: set (exec-path-edges p) ⊆ set (nat-graph-edge-list G)
    by (rule exec-walk-path-edges-subset[OF walk])
  have path-distinct: distinct (exec-path-edges p)
    by (rule exec-path-edges-distinct-if-distinct[OF distinct])
  have graph-distinct: distinct (nat-graph-edge-list G)
    by (rule nat-graph-well-formed-distinct-edge-list[OF wf])
  have path-sum:
    sum-list (map ?w (exec-path-edges p)) =
      sum ?w (set (exec-path-edges p))
    by (rule sum-list-distinct-conv-sum-set[OF path-distinct])
  have graph-sum:
    sum-list (map ?w (nat-graph-edge-list G)) =
      sum ?w (set (nat-graph-edge-list G))
    by (rule sum-list-distinct-conv-sum-set[OF graph-distinct])
  have finite-graph: finite (set (nat-graph-edge-list G))
    by (rule finite-set)
  have sum ?w (set (exec-path-edges p)) ≤
      sum ?w (set (nat-graph-edge-list G))
    by (rule sum-mono2[OF finite-graph path-subset])
      (simp add: nat-graph-weight-nonnegative)
  also have ... = real (nat-graph-total-weight G)
    using graph-sum nat-graph-weight-edge-list-sum[OF wf] by simp
  finally have sum-list (map ?w (exec-path-edges p)) ≤
      real (nat-graph-total-weight G)
    using path-sum by simp
  then show ?thesis
    unfolding exec-walk-weight-sum-path-edges .
qed

definition bmssp-label-witnesses ::
  nat-graph ⇒ nat ⇒ nat list ⇒ nat-dist ⇒ bool where
  bmssp-label-witnesses G src settled ds ⟷
    (∀ v d. (v, d) ∈ set ds ⟶
      (∃ p. p ≠ [] ∧ hd p = src ∧ last p = v ∧ distinct p ∧
        exec-walk (bmssp-vertices G src) (nat-graph-edge-list G) p ∧
        exec-walk-weight (nat-graph-weight G) p = real d ∧
        set (butlast p) ⊆ set settled))

lemma bmssp-label-witnesses-mono-settled:
  assumes witnesses: bmssp-label-witnesses G src settled ds
    and subset: set settled ⊆ set settled'
  shows bmssp-label-witnesses G src settled' ds
  unfolding bmssp-label-witnesses-def
proof (intro allI impI)
  fix v d
  assume (v, d) ∈ set ds
  then obtain p where p:
```

```
    p ≠ [] hd p = src last p = v distinct p
    exec-walk (bmssp-vertices G src) (nat-graph-edge-list G) p
    exec-walk-weight (nat-graph-weight G) p = real d
    set (butlast p) ⊆ set settled
    using witnesses unfolding bmssp-label-witnesses-def by blast
  have set (butlast p) ⊆ set settled'
    using p(7) subset by blast
  then show ∃p. p ≠ [] ∧ hd p = src ∧ last p = v ∧ distinct p ∧
      exec-walk (bmssp-vertices G src) (nat-graph-edge-list G) p ∧
      exec-walk-weight (nat-graph-weight G) p = real d ∧
      set (butlast p) ⊆ set settled'
    using p(1–6) by blast
qed

lemma bmssp-label-witnesses-initial:
  bmssp-label-witnesses G src [] [(src, 0)]
  unfolding bmssp-label-witnesses-def
  by (intro allI impI, auto simp: bmssp-vertices-set intro!: exI[of - [src]])

lemma bmssp-label-witnesses-set-dist:
  assumes witnesses: bmssp-label-witnesses G src settled ds
    and p: p ≠ [] hd p = src last p = x distinct p
      exec-walk (bmssp-vertices G src) (nat-graph-edge-list G) p
      exec-walk-weight (nat-graph-weight G) p = real dx
      set (butlast p) ⊆ set settled
  shows bmssp-label-witnesses G src settled (bmssp-set-dist x dx ds)
  unfolding bmssp-label-witnesses-def
proof (intro allI impI)
  fix v d
  assume mem: (v, d) ∈ set (bmssp-set-dist x dx ds)
  consider (new) v = x d = dx | (old) (v, d) ∈ set ds
    using bmssp-set-dist-mem-cases[OF mem] by blast
  then show ∃p. p ≠ [] ∧ hd p = src ∧ last p = v ∧ distinct p ∧
      exec-walk (bmssp-vertices G src) (nat-graph-edge-list G) p ∧
      exec-walk-weight (nat-graph-weight G) p = real d ∧
      set (butlast p) ⊆ set settled
  proof cases
    case new
    then show ?thesis
      using p by blast
  next
    case old
    then show ?thesis
      using witnesses unfolding bmssp-label-witnesses-def by blast
  qed
qed

lemma bmssp-relax-edges-lookup-settled:
  assumes v ∈ set settled
```

```
  shows bmssp-lookup-dist (snd (bmssp-relax-edges G settled u du ds)) v =
    bmssp-lookup-dist ds v
  using assms
proof (induction G arbitrary: ds)
  case Nil
  then show ?case
    by simp
next
  case (Cons e es)
  obtain a b w where e: e = (a, b, w)
    by (cases e)
  obtain updates ds1 where rec:
    bmssp-relax-edges es settled u du ds = (updates, ds1)
    by force
  have lookup-tail: bmssp-lookup-dist ds1 v = bmssp-lookup-dist ds v
    using Cons.IH[OF Cons.prems, of ds] rec by simp
  show ?case
  proof (cases a = u ∧ b ∉ set settled)
    case False
    then show ?thesis
      using rec lookup-tail unfolding e
      by (auto simp: Let-def split: option.splits if-splits)
  next
    case True
    then have b-ne-v: b ≠ v
      using Cons.prems by auto
    show ?thesis
    proof (cases bmssp-lookup-dist ds1 b)
      case None
      then show ?thesis
        using True rec lookup-tail b-ne-v
        unfolding e by (simp add: Let-def bmssp-lookup-dist-set-dist-other)
    next
      case (Some old)
      show ?thesis
      proof (cases du + w < old)
        case True
        then show ?thesis
          using ‹a = u ∧ b ∉ set settled› rec Some lookup-tail b-ne-v
          unfolding e by (simp add: Let-def bmssp-lookup-dist-set-dist-other)
      next
        case False
        then show ?thesis
          using ‹a = u ∧ b ∉ set settled› rec Some lookup-tail
          unfolding e by (simp add: Let-def)
      qed
    qed
  qed
qed
```

lemma *bmssp-relax-vertices-lookup-settled*:
  assumes $v \in$ *set settled*
  shows *bmssp-lookup-dist* (*snd* (*bmssp-relax-vertices G settled xs ds*)) $v =$
    bmssp-lookup-dist ds v
proof (*induction xs arbitrary*: *ds*)
  case *Nil*
  then show *?case*
    by *simp*
next
  case (*Cons u us*)
  show *?case*
  proof (*cases bmssp-lookup-dist ds u*)
    case *None*
    then show *?thesis*
      using *Cons.IH* **by** *simp*
  next
    case (*Some du*)
    obtain *updates-u ds1* **where** *edge-rec*:
      bmssp-relax-edges G settled u du ds = (*updates-u*, *ds1*)
      by *force*
    have *lookup-ds1*:
      bmssp-lookup-dist ds1 v = *bmssp-lookup-dist ds v*
      using *bmssp-relax-edges-lookup-settled*[*OF assms*, *of G u du ds*]
        edge-rec **by** *simp*
    obtain *updates-us ds2* **where** *vertices-rec*:
      bmssp-relax-vertices G settled us ds1 = (*updates-us*, *ds2*)
      by *force*
    have *lookup-ds2*:
      bmssp-lookup-dist ds2 v = *bmssp-lookup-dist ds1 v*
      using *Cons.IH*[*of ds1*] *vertices-rec* **by** *simp*
    show *?thesis*
      using *Some edge-rec vertices-rec lookup-ds1 lookup-ds2* **by** *simp*
  qed
qed

lemma *bmssp-relax-vertices-edge-lookup-le-candidate*:
  assumes *distinct*: *distinct* (*map fst ds*)
    and *xs-subset*: *set xs* $\subseteq$ *set settled*
    and *u-xs*: $u \in$ *set xs*
    and *lookup-u*: *bmssp-lookup-dist ds u* = *Some du*
    and *edge*: $(u, v, w) \in$ *set G*
    and *v-unsettled*: $v \notin$ *set settled*
  shows $\exists\, dv.$
    bmssp-lookup-dist (*snd* (*bmssp-relax-vertices G settled xs ds*)) $v =$
      Some dv $\wedge$
    $dv \leq du + w$
  using *distinct xs-subset u-xs lookup-u*
proof (*induction xs arbitrary*: *ds du*)

```
  case Nil
  then show ?case
    by simp
next
  case (Cons x xs)
  show ?case
  proof (cases bmssp-lookup-dist ds x)
    case None
    have u-ne-x: u ≠ x
      using None Cons.prems(4) by auto
    then have u-tail: u ∈ set xs
      using Cons.prems(3) by simp
    obtain dv where tail-lookup:
        bmssp-lookup-dist
          (snd (bmssp-relax-vertices G settled xs ds)) v = Some dv
      and tail-le: dv ≤ du + w
      using Cons.IH[OF Cons.prems(1) - u-tail Cons.prems(4)]
      using Cons.prems(2) by auto
    have final-lookup:
      bmssp-lookup-dist
        (snd (bmssp-relax-vertices G settled (x # xs) ds)) v = Some dv
      using None tail-lookup by simp
    show ?thesis
      using final-lookup tail-le by blast
  next
    case (Some dx)
    obtain updates-x ds1 where edge-rec:
      bmssp-relax-edges G settled x dx ds = (updates-x, ds1)
      by force
    have distinct-ds1: distinct (map fst ds1)
      using bmssp-relax-edges-preserves-distinct-dist[OF Cons.prems(1),
          of G settled x dx] edge-rec by simp
    have xs-subset: set xs ⊆ set settled
      using Cons.prems(2) by simp
    obtain updates-xs ds2 where vertices-rec:
      bmssp-relax-vertices G settled xs ds1 = (updates-xs, ds2)
      by force
    show ?thesis
    proof (cases u = x)
      case True
      then have du-eq: du = dx
        using Some Cons.prems(4) by simp
      have edge-x: (x, v, w) ∈ set G
        using edge True by simp
      obtain mid where mid-lookup:
          bmssp-lookup-dist (snd (bmssp-relax-edges G settled x dx ds)) v =
            Some mid
        and mid-le: mid ≤ dx + w
        using bmssp-relax-edges-edge-lookup-le-candidate
```

```
        [OF edge-x v-unsettled, of dx ds]
      by blast
    have mid-ds1: bmssp-lookup-dist ds1 v = Some mid
      using mid-lookup edge-rec by simp
    obtain dv where final-lookup:
      bmssp-lookup-dist
        (snd (bmssp-relax-vertices G settled xs ds1)) v = Some dv
      using bmssp-relax-vertices-lookup-Some-preserved
        [OF distinct-ds1 mid-ds1]
      by blast
    have dv-le-mid:
      dv ≤ mid
      by (rule bmssp-relax-vertices-lookup-le
        [OF distinct-ds1 mid-ds1 final-lookup])
    have dv ≤ du + w
      using dv-le-mid mid-le du-eq by linarith
    then show ?thesis
      using Some edge-rec vertices-rec final-lookup by auto
  next
    case False
    have u-tail: u ∈ set xs
      using Cons.prems(3) False by simp
    have u-settled: u ∈ set settled
      using xs-subset u-tail by blast
    have lookup-u-ds1: bmssp-lookup-dist ds1 u = Some du
      using bmssp-relax-edges-lookup-settled[OF u-settled, of G x dx ds]
        Cons.prems(4) edge-rec by simp
    obtain dv where final-lookup:
      bmssp-lookup-dist
        (snd (bmssp-relax-vertices G settled xs ds1)) v = Some dv
      and final-le: dv ≤ du + w
      using Cons.IH[OF distinct-ds1 xs-subset u-tail lookup-u-ds1]
      by blast
    show ?thesis
      using Some edge-rec vertices-rec final-lookup final-le by auto
  qed
 qed
qed

definition bmssp-settled-exact ::
  nat-graph ⇒ nat ⇒ nat list ⇒ nat-dist ⇒ bool where
  bmssp-settled-exact G src settled ds ⟷
    (∀ v d. v ∈ set settled ⟶
      bmssp-lookup-dist ds v = Some d ⟶
      real d = nat-graph-dist G src v)

lemma bmssp-settled-exact-initial:
  bmssp-settled-exact G src [] ds
  unfolding bmssp-settled-exact-def by simp
```

lemma *bmssp-settled-exact-step*:
  assumes *exact-old*: *bmssp-settled-exact G src old-settled ds*
    and *pulled-exact*:
      $\bigwedge v\ d.\ [\![v \in set\ pulled;\ bmssp\text{-}lookup\text{-}dist\ ds\ v = Some\ d]\!]$
        $\Longrightarrow real\ d = nat\text{-}graph\text{-}dist\ G\ src\ v$
    and *settled-def*: *settled* = *pulled* @ *old-settled*
    and *relaxed*: *bmssp-relax-vertices G settled pulled ds* = (*updates*, *ds'*)
  shows *bmssp-settled-exact G src settled ds'*
  unfolding *bmssp-settled-exact-def*
proof (*intro allI impI*)
  fix *v d*
  assume *v-settled*: *v* ∈ *set settled*
    and *lookup'*: *bmssp-lookup-dist ds' v* = *Some d*
  have *lookup-old*: *bmssp-lookup-dist ds v* = *Some d*
    using *bmssp-relax-vertices-lookup-settled*[*OF v-settled*, *of G pulled ds*]
      relaxed lookup' **by** *simp*
  have *v* ∈ *set pulled* ∨ *v* ∈ *set old-settled*
    using *v-settled settled-def* **by** *auto*
  then show *real d* = *nat-graph-dist G src v*
  proof
    assume *v* ∈ *set pulled*
    then show *?thesis*
      by (*rule pulled-exact*[*OF - lookup-old*])
  next
    assume *v* ∈ *set old-settled*
    then show *?thesis*
      using *exact-old lookup-old* **unfolding** *bmssp-settled-exact-def* **by** *blast*
  qed
qed

definition *bmssp-frontier-relaxed* ::
  nat-graph ⇒ *nat* ⇒ *nat list* ⇒ *nat-dist* ⇒ *bool* **where**
  bmssp-frontier-relaxed G src settled ds ⟷
    (∀ *u v w du*. (*u*, *v*, *w*) ∈ *set G* ⟶
      u ∈ *set settled* ⟶
      v ∉ *set settled* ⟶
      bmssp-lookup-dist ds u = *Some du* ⟶
      real du = *nat-graph-dist G src u* ⟶
      (∃ *dv*. *bmssp-lookup-dist ds v* = *Some dv* ∧
        real dv ≤ *real du* + *real w*))

lemma *bmssp-frontier-relaxed-initial*:
  bmssp-frontier-relaxed G src [] *ds*
  unfolding *bmssp-frontier-relaxed-def* **by** *simp*

lemma *bmssp-frontier-relaxed-step*:
  assumes *frontier-old*: *bmssp-frontier-relaxed G src old-settled ds*
    and *distinct*: *distinct* (*map fst ds*)

```
    and settled-def: settled = pulled @ old-settled
    and relaxed: bmssp-relax-vertices G settled pulled ds = (updates, ds')
  shows bmssp-frontier-relaxed G src settled ds'
  unfolding bmssp-frontier-relaxed-def
proof (intro allI impI)
  fix u v w du
  assume edge: (u, v, w) ∈ set G
    and u-settled: u ∈ set settled
    and v-unsettled: v ∉ set settled
    and lookup-u': bmssp-lookup-dist ds' u = Some du
    and exact-u: real du = nat-graph-dist G src u
  have lookup-u: bmssp-lookup-dist ds u = Some du
    using bmssp-relax-vertices-lookup-settled[OF u-settled, of G pulled ds]
      relaxed lookup-u' by simp
  have u-cases: u ∈ set pulled ∨ u ∈ set old-settled
    using u-settled settled-def by auto
  show ∃ dv. bmssp-lookup-dist ds' v = Some dv ∧
      real dv ≤ real du + real w
  proof (cases u ∈ set old-settled)
    case True
    have v-old-unsettled: v ∉ set old-settled
      using v-unsettled settled-def by auto
    obtain old-dv where old-lookup:
        bmssp-lookup-dist ds v = Some old-dv
      and old-le: real old-dv ≤ real du + real w
      using frontier-old edge True v-old-unsettled lookup-u exact-u
      unfolding bmssp-frontier-relaxed-def by blast
    obtain dv where final-lookup:
        bmssp-lookup-dist
          (snd (bmssp-relax-vertices G settled pulled ds)) v = Some dv
      using bmssp-relax-vertices-lookup-Some-preserved
        [OF distinct old-lookup]
      by blast
    have dv-le-old: dv ≤ old-dv
      by (rule bmssp-relax-vertices-lookup-le
        [OF distinct old-lookup final-lookup])
    have real dv ≤ real du + real w
      using dv-le-old old-le by linarith
    then show ?thesis
      using final-lookup relaxed by auto
  next
    case False
    then have u-pulled: u ∈ set pulled
      using u-cases by blast
    have pulled-subset: set pulled ⊆ set settled
      using settled-def by auto
    obtain dv where lookup:
        bmssp-lookup-dist
          (snd (bmssp-relax-vertices G settled pulled ds)) v = Some dv
```

```
    and dv-le: dv ≤ du + w
    using bmssp-relax-vertices-edge-lookup-le-candidate
      [OF distinct pulled-subset u-pulled lookup-u edge v-unsettled]
    by blast
  have real dv ≤ real du + real w
    using dv-le by simp
  then show ?thesis
    using lookup relaxed by auto
 qed
qed

definition bmssp-source-zero :: nat ⇒ nat-dist ⇒ bool where
 bmssp-source-zero src ds ⟷ bmssp-lookup-dist ds src = Some 0

lemma bmssp-source-zero-initial:
 bmssp-source-zero src [(src, 0)]
 unfolding bmssp-source-zero-def by simp

lemma bmssp-source-zero-step:
 assumes zero: bmssp-source-zero src ds
   and distinct: distinct (map fst ds)
   and relaxed: bmssp-relax-vertices G settled pulled ds = (updates, ds')
 shows bmssp-source-zero src ds'
proof −
 have lookup0: bmssp-lookup-dist ds src = Some 0
   using zero unfolding bmssp-source-zero-def .
 obtain d' where lookup':
   bmssp-lookup-dist
     (snd (bmssp-relax-vertices G settled pulled ds)) src = Some d'
   using bmssp-relax-vertices-lookup-Some-preserved[OF distinct lookup0]
   by blast
 have d' ≤ 0
   by (rule bmssp-relax-vertices-lookup-le[OF distinct lookup0 lookup'])
 then have d' = 0
   by simp
 then show ?thesis
   using lookup' relaxed unfolding bmssp-source-zero-def by simp
qed

definition bmssp-settled-have-lookups ::
 nat list ⇒ nat-dist ⇒ bool where
 bmssp-settled-have-lookups settled ds ⟷
   (∀ v∈set settled. ∃ d. bmssp-lookup-dist ds v = Some d)

lemma bmssp-settled-have-lookups-initial:
 bmssp-settled-have-lookups [] ds
 unfolding bmssp-settled-have-lookups-def by simp

lemma bmssp-settled-have-lookups-step:
```

```
  assumes have-old: bmssp-settled-have-lookups old-settled ds
    and match: bmssp-partition-state-match vertices old-settled ds P
    and pull: pull-separates (bp-view P) bmssp-block-size B S beta D'
    and pulled-def:
      pulled = filter (λx. x ∈ S ∧ x ∉ set old-settled) vertices
    and settled-def: settled = pulled @ old-settled
    and relaxed: bmssp-relax-vertices G settled pulled ds = (updates, ds')
  shows bmssp-settled-have-lookups settled ds'
  unfolding bmssp-settled-have-lookups-def
proof
  fix v
  assume v-settled: v ∈ set settled
  have v-cases: v ∈ set pulled ∨ v ∈ set old-settled
    using v-settled settled-def by auto
  then obtain d where lookup: bmssp-lookup-dist ds v = Some d
  proof
    assume v-pulled: v ∈ set pulled
    obtain d where bmssp-lookup-dist ds v = Some d
      by (rule bmssp-partition-state-pulled-list-lookup
          [OF match pull pulled-def v-pulled])
    then show ?thesis ..
  next
    assume v ∈ set old-settled
    then obtain d where bmssp-lookup-dist ds v = Some d
      using have-old unfolding bmssp-settled-have-lookups-def by blast
    then show ?thesis ..
  qed
  have lookup':
    bmssp-lookup-dist
      (snd (bmssp-relax-vertices G settled pulled ds)) v =
      bmssp-lookup-dist ds v
    by (rule bmssp-relax-vertices-lookup-settled[OF v-settled])
  show ∃ d. bmssp-lookup-dist ds' v = Some d
    using lookup lookup' relaxed by simp
qed

lemma finite-card-le-one-member-eq:
  assumes finite: finite A
    and card: card A ≤ 1
    and x: x ∈ A
    and y: y ∈ A
  shows x = y
proof (rule ccontr)
  assume x ≠ y
  then have card {x, y} = 2
    by simp
  moreover have {x, y} ⊆ A
    using x y by blast
  ultimately have 2 ≤ card A
```

```
    using card-mono[OF finite, of {x, y}] by simp
  then show False
    using card by simp
qed

definition bmssp-dijkstra-state ::
  nat-graph ⇒ nat ⇒ nat list ⇒ nat list ⇒
    nat-dist ⇒ nat bucketed-partition ⇒ bool where
  bmssp-dijkstra-state G src vertices settled ds P ⟷
      bmssp-partition-state-match vertices settled ds P ∧
      bmssp-source-zero src ds ∧
      bmssp-settled-have-lookups settled ds ∧
      bmssp-label-witnesses G src settled ds ∧
      bmssp-settled-exact G src settled ds ∧
      bmssp-frontier-relaxed G src settled ds

lemma bmssp-dijkstra-state-initial:
  assumes src-vertices: src ∈ set vertices
  shows bmssp-dijkstra-state G src vertices [] [(src, 0)]
    (bp-result-of
      (c-bp-regularized-local-insert src (bmssp-partition-key src 0)
        (bp-empty bmssp-block-size bmssp-bound)))
proof −
  have match:
    bmssp-partition-state-match vertices [] [(src, 0)]
      (bp-result-of
        (c-bp-regularized-local-insert src (bmssp-partition-key src 0)
          (bp-empty bmssp-block-size bmssp-bound)))
    by (rule bmssp-partition-state-match-initial[OF src-vertices])
  show ?thesis
    unfolding bmssp-dijkstra-state-def
    using match bmssp-source-zero-initial
      bmssp-settled-have-lookups-initial bmssp-label-witnesses-initial
      bmssp-settled-exact-initial bmssp-frontier-relaxed-initial
    by simp
qed

lemma bmssp-relax-edges-preserves-label-witnesses-aux:
  assumes wf: nat-graph-well-formed Gfull
    and edge-subset: set Erel ⊆ set Gfull
    and witnesses: bmssp-label-witnesses Gfull src settled ds
    and u-settled: u ∈ set settled
    and lookup-u: bmssp-lookup-dist ds u = Some du
  shows bmssp-label-witnesses Gfull src settled
    (snd (bmssp-relax-edges Erel settled u du ds))
  using edge-subset witnesses lookup-u
proof (induction Erel arbitrary: ds)
  case Nil
  then show ?case
```

```
    by simp
next
  case (Cons e es)
  obtain a b w where e-def: e = (a, b, w)
    by (cases e)
  obtain updates ds1 where rec:
    bmssp-relax-edges es settled u du ds = (updates, ds1)
    by force
  have es-subset: set es ⊆ set Gfull
    using Cons.prems(1) by simp
  have tail-witnesses: bmssp-label-witnesses Gfull src settled ds1
    using Cons.IH[OF es-subset Cons.prems(2) Cons.prems(3)] rec by simp
  have lookup-u-ds1: bmssp-lookup-dist ds1 u = Some du
    using bmssp-relax-edges-lookup-settled[OF u-settled, of es u du ds]
      Cons.prems(3) rec by simp
  show ?case
  proof (cases a = u ∧ b ∉ set settled)
    case False
    then show ?thesis
      using rec tail-witnesses unfolding e-def
      by (auto simp: Let-def split: option.splits if-splits)
  next
    case True
    note active = True
    let ?nd = du + w
    have edge-rel: (u, b, w) ∈ set (e # es)
      using active unfolding e-def by simp
    have edge-mem: (u, b, w) ∈ set Gfull
      using Cons.prems(1) edge-rel by blast
    have edge-pair: (u, b) ∈ set (nat-graph-edge-list Gfull)
      unfolding nat-graph-edge-list-def using edge-mem by force
    have b-vertex: b ∈ set (bmssp-vertices Gfull src)
      by (rule bmssp-edge-target-in-vertices[OF edge-mem])
    have weight-b: nat-graph-weight Gfull u b = real w
      by (rule nat-graph-weight-of-edge[OF wf edge-mem])
    have u-mem: (u, du) ∈ set ds1
      by (rule bmssp-lookup-dist-Some-mem[OF lookup-u-ds1])
    obtain p where p:
      p ≠ [] hd p = src last p = u distinct p
      exec-walk (bmssp-vertices Gfull src)
        (nat-graph-edge-list Gfull) p
      exec-walk-weight (nat-graph-weight Gfull) p = real du
      set (butlast p) ⊆ set settled
      using tail-witnesses u-mem unfolding bmssp-label-witnesses-def by blast
    have set-p: set p ⊆ set settled
    proof
      fix x
      assume x: x ∈ set p
      have p = butlast p @ [last p]
```

```
    using p(1) by simp
  then have x ∈ set (butlast p @ [last p])
    using x by simp
  then have x ∈ set (butlast p) ∪ {last p}
    by (simp only: set-append set-simps)
  then have x ∈ set (butlast p) ∨ x = last p
    by blast
  then show x ∈ set settled
  proof
    assume x ∈ set (butlast p)
    then show ?thesis
      using p(7) by blast
  next
    assume x = last p
    then show ?thesis
      using p(3) u-settled by simp
  qed
qed
have b-not-p: b ∉ set p
  using active set-p by auto
have walk-pb:
  exec-walk (bmssp-vertices Gfull src)
    (nat-graph-edge-list Gfull) (p @ [b])
  by (rule exec-walk-append-edge[OF p(5) p(1) p(3) b-vertex edge-pair])
have distinct-pb: distinct (p @ [b])
  using p(4) b-not-p by simp
have weight-pb:
  exec-walk-weight (nat-graph-weight Gfull) (p @ [b]) = real ?nd
  using exec-walk-weight-append-edge[OF p(1), of nat-graph-weight Gfull b]
    p(3,6) weight-b by simp
have butlast-pb: set (butlast (p @ [b])) ⊆ set settled
  using p(1) set-p by simp
have new-witnesses:
  bmssp-label-witnesses Gfull src settled
    (bmssp-set-dist b ?nd ds1)
  by (rule bmssp-label-witnesses-set-dist
      [OF tail-witnesses, of p @ [b] b ?nd])
    (use p(1,2) active walk-pb distinct-pb weight-pb butlast-pb in auto)
show ?thesis
proof (cases bmssp-lookup-dist ds1 b)
  case None
  then show ?thesis
    using active rec new-witnesses unfolding e-def by (simp add: Let-def)
next
  case (Some old)
  show ?thesis
  proof (cases ?nd < old)
    case True
    then show ?thesis
```

```
        using active rec Some new-witnesses unfolding e-def
        by (simp add: Let-def)
      next
        case False
        then show ?thesis
        using active rec Some tail-witnesses unfolding e-def
        by (simp add: Let-def)
      qed
    qed
  qed
qed

lemma bmssp-relax-edges-preserves-label-witnesses:
  assumes wf: nat-graph-well-formed G
    and witnesses: bmssp-label-witnesses G src settled ds
    and u-settled: u ∈ set settled
    and lookup-u: bmssp-lookup-dist ds u = Some du
  shows bmssp-label-witnesses G src settled
    (snd (bmssp-relax-edges G settled u du ds))
  by (rule bmssp-relax-edges-preserves-label-witnesses-aux
      [OF wf - witnesses u-settled lookup-u]) simp

lemma bmssp-relax-vertices-preserves-label-witnesses:
  assumes wf: nat-graph-well-formed G
    and witnesses: bmssp-label-witnesses G src settled ds
    and xs-subset: set xs ⊆ set settled
  shows bmssp-label-witnesses G src settled
    (snd (bmssp-relax-vertices G settled xs ds))
  using witnesses xs-subset
proof (induction xs arbitrary: ds)
  case Nil
  then show ?case
    by simp
next
  case (Cons u us)
  show ?case
  proof (cases bmssp-lookup-dist ds u)
    case None
    have us-subset: set us ⊆ set settled
      using Cons.prems(2) by simp
    show ?thesis
      using None Cons.IH[OF Cons.prems(1) us-subset] by simp
  next
    case (Some du)
    have u-settled: u ∈ set settled
      using Cons.prems(2) by simp
    obtain updates-u ds1 where edge-rec:
      bmssp-relax-edges G settled u du ds = (updates-u, ds1)
      by force
```

```
    have witnesses1: bmssp-label-witnesses G src settled ds1
      using bmssp-relax-edges-preserves-label-witnesses
        [OF wf Cons.prems(1) u-settled Some] edge-rec by simp
    obtain updates-us ds2 where vertices-rec:
      bmssp-relax-vertices G settled us ds1 = (updates-us, ds2)
      by force
    have us-subset: set us ⊆ set settled
      using Cons.prems(2) by simp
    have witnesses2: bmssp-label-witnesses G src settled ds2
      using Cons.IH[OF witnesses1 us-subset] vertices-rec by simp
    show ?thesis
      using Some edge-rec vertices-rec witnesses2 by simp
  qed
qed

lemma bmssp-label-witnesses-lookup-lt-infinity:
  assumes wf: nat-graph-well-formed G
    and witnesses: bmssp-label-witnesses G src settled ds
    and lookup: bmssp-lookup-dist ds v = Some d
  shows d < bmssp-infinity
proof −
  have mem: (v, d) ∈ set ds
    by (rule bmssp-lookup-dist-Some-mem[OF lookup])
  then obtain p where p:
    distinct p
    exec-walk (bmssp-vertices G src) (nat-graph-edge-list G) p
    exec-walk-weight (nat-graph-weight G) p = real d
    using witnesses unfolding bmssp-label-witnesses-def by blast
  have d-le-total:
    real d ≤ real (nat-graph-total-weight G)
    using exec-walk-weight-nat-graph-le-total[OF wf p(2) p(1)] p(3)
    by simp
  have total-lt: nat-graph-total-weight G < bmssp-infinity
    using wf unfolding nat-graph-well-formed-def by simp
  show ?thesis
    using d-le-total total-lt by linarith
qed

lemma bmssp-relax-vertices-update-lt-bound:
  assumes wf: nat-graph-well-formed G
    and witnesses: bmssp-label-witnesses G src settled ds
    and xs-subset: set xs ⊆ set settled
    and distinct-updates:
      distinct (map fst (fst (bmssp-relax-vertices G settled xs ds)))
    and update: (v, b) ∈ set (fst (bmssp-relax-vertices G settled xs ds))
  shows b < bmssp-bound
proof −
  have witnesses':
    bmssp-label-witnesses G src settled
```

```
    (snd (bmssp-relax-vertices G settled xs ds))
   by (rule bmssp-relax-vertices-preserves-label-witnesses
     [OF wf witnesses xs-subset])
 have lookup:
   bmssp-lookup-dist (snd (bmssp-relax-vertices G settled xs ds)) v =
    Some (nat (floor b))
   by (rule bmssp-relax-vertices-update-lookup-floor
     [OF distinct-updates update])
 have d-lt: nat (floor b) < bmssp-infinity
   by (rule bmssp-label-witnesses-lookup-lt-infinity
     [OF wf witnesses' lookup])
 obtain d where b-key: b = bmssp-partition-key v d
   and d-floor: nat (floor b) = d
   using bmssp-relax-vertices-update-floor[OF update] by blast
 have key-lt: bmssp-partition-key v d < bmssp-bound
   by (rule bmssp-partition-key-lt-bound-if-distance-lt)
    (use d-lt d-floor in simp)
 show ?thesis
   using b-key key-lt by simp
qed

lemma bmssp-loop-lookup-settled:
 assumes distinct: distinct (map fst ds)
   and v-settled: v ∈ set settled
 shows bmssp-lookup-dist (bmssp-loop fuel G vertices settled ds P) v =
   bmssp-lookup-dist ds v
 using distinct v-settled
proof (induction fuel arbitrary: settled ds P)
 case 0
 then show ?case
   by (simp add: bmssp-lookup-dist-normalize-dist)
next
 case (Suc fuel)
 obtain S beta P1 where pull:
   bp-pull bmssp-block-size bmssp-bound P = (S, beta, P1)
   by (cases bp-pull bmssp-block-size bmssp-bound P) auto
 let ?pulled = filter (λx. x ∈ S ∧ x ∉ set settled) vertices
 show ?case
 proof (cases ?pulled = [])
   case True
   then show ?thesis
     using pull Suc.prems by (simp add: bmssp-lookup-dist-normalize-dist)
 next
   case False
   let ?settled' = ?pulled @ settled
   have v-settled': v ∈ set ?settled'
     using Suc.prems by simp
   obtain updates ds' where relaxed:
     bmssp-relax-vertices G ?settled' ?pulled ds = (updates, ds')
```

```
    by force
  have lookup-ds':
    bmssp-lookup-dist ds' v = bmssp-lookup-dist ds v
    using bmssp-relax-vertices-lookup-settled[OF v-settled', of G ?pulled ds]
      relaxed by simp
  have distinct-ds':
    distinct (map fst ds')
    using bmssp-relax-vertices-preserves-distinct-dist[OF Suc.prems(1),
        of G ?settled' ?pulled] relaxed by simp
  have rec:
    bmssp-lookup-dist
      (bmssp-loop fuel G vertices ?settled' ds'
        (bmssp-apply-updates updates P1)) v =
      bmssp-lookup-dist ds' v
    by (rule Suc.IH[OF distinct-ds' v-settled'])
  have loop-eq:
    bmssp-loop (Suc fuel) G vertices settled ds P =
      bmssp-loop fuel G vertices ?settled' ds'
        (bmssp-apply-updates updates P1)
    using pull False relaxed by (simp add: Let-def)
  show ?thesis
    using loop-eq rec lookup-ds' by simp
 qed
qed

lemma bmssp-loop-preserves-label-witnesses:
 assumes wf: nat-graph-well-formed G
   and witnesses: bmssp-label-witnesses G src settled ds
 shows ∃settled'. set settled ⊆ set settled' ∧
   bmssp-label-witnesses G src settled'
     (bmssp-loop fuel G vertices settled ds P)
 using witnesses
proof (induction fuel arbitrary: settled ds P)
 case 0
 have set-norm: set (bmssp-normalize-dist ds) = set ds
   unfolding bmssp-normalize-dist-def by simp
 have witnesses-norm:
   bmssp-label-witnesses G src settled (bmssp-normalize-dist ds)
   using 0.prems set-norm unfolding bmssp-label-witnesses-def by auto
 show ?case
 proof (intro exI conjI)
   show set settled ⊆ set settled
     by simp
   show bmssp-label-witnesses G src settled
       (bmssp-loop 0 G vertices settled ds P)
     using witnesses-norm by simp
 qed
next
 case (Suc fuel)
```

obtain *S beta P1* **where** *pull*:
bp-pull bmssp-block-size bmssp-bound P = (*S*, *beta*, *P1*)
by (*cases bp-pull bmssp-block-size bmssp-bound P*) *auto*
let *?pulled* = *filter* (λ*x*. *x* ∈ *S* ∧ *x* ∉ *set settled*) *vertices*
show *?case*
proof (*cases ?pulled* = [])
case *True*
have *set-norm*: *set* (*bmssp-normalize-dist ds*) = *set ds*
unfolding *bmssp-normalize-dist-def* **by** *simp*
have *witnesses-norm*:
bmssp-label-witnesses G src settled (*bmssp-normalize-dist ds*)
using *Suc.prems set-norm* **unfolding** *bmssp-label-witnesses-def* **by** *auto*
have *loop-eq*:
bmssp-loop (*Suc fuel*) *G vertices settled ds P* =
bmssp-normalize-dist ds
using *pull True* **by** *simp*
show *?thesis*
proof (*intro exI conjI*)
show *set settled* ⊆ *set settled*
by *simp*
show *bmssp-label-witnesses G src settled*
(*bmssp-loop* (*Suc fuel*) *G vertices settled ds P*)
using *witnesses-norm loop-eq* **by** *simp*
qed
next
case *False*
let *?settled′* = *?pulled* @ *settled*
have *settled-subset*: *set settled* ⊆ *set ?settled′*
by *auto*
have *witnesses-settled′*:
bmssp-label-witnesses G src ?settled′ ds
by (*rule bmssp-label-witnesses-mono-settled*
[*OF Suc.prems settled-subset*])
have *pulled-subset*: *set ?pulled* ⊆ *set ?settled′*
by *simp*
obtain *updates ds′* **where** *relaxed*:
bmssp-relax-vertices G ?settled′ ?pulled ds = (*updates*, *ds′*)
by *force*
have *witnesses-ds′*:
bmssp-label-witnesses G src ?settled′ ds′
using *bmssp-relax-vertices-preserves-label-witnesses*
[*OF wf witnesses-settled′ pulled-subset*] *relaxed* **by** *simp*
obtain *settled″* **where** *settled″*:
set ?settled′ ⊆ *set settled″*
bmssp-label-witnesses G src settled″
(*bmssp-loop fuel G vertices ?settled′ ds′*
(*bmssp-apply-updates updates P1*))
using *Suc.IH*[*OF witnesses-ds′*] **by** *blast*
have *loop-eq*:

```
      bmssp-loop (Suc fuel) G vertices settled ds P =
        bmssp-loop fuel G vertices ?settled' ds'
          (bmssp-apply-updates updates P1)
      using pull False relaxed by (simp add: Let-def)
    have settled-to-final: set settled ⊆ set settled''
      using settled-subset settled''(1) by blast
    show ?thesis
    proof (intro exI conjI)
      show set settled ⊆ set settled''
        by (rule settled-to-final)
      show bmssp-label-witnesses G src settled''
          (bmssp-loop (Suc fuel) G vertices settled ds P)
        using settled''(2) loop-eq by simp
    qed
  qed
qed

lemma bmssp-distances-label-witnesses:
  assumes wf: nat-graph-well-formed G
  shows ∃settled. bmssp-label-witnesses G src settled
    (bmssp-distances G src)
proof −
  let ?vertices = bmssp-vertices G src
  let ?P0 = bp-empty bmssp-block-size bmssp-bound
  let ?P1 =
    bp-regularized-local-insert src (bmssp-partition-key src 0) ?P0
  let ?fuel = Suc (length ?vertices * Suc (length G))
  have initial: bmssp-label-witnesses G src [] [(src, 0)]
    by (rule bmssp-label-witnesses-initial)
  obtain settled where loop:
    bmssp-label-witnesses G src settled
      (bmssp-loop ?fuel G ?vertices [] [(src, 0)] ?P1)
    using bmssp-loop-preserves-label-witnesses[OF wf initial] by blast
  have unfold:
    bmssp-distances G src =
      bmssp-loop ?fuel G ?vertices [] [(src, 0)] ?P1
    unfolding bmssp-distances-def by (simp add: Let-def)
  show ?thesis
  proof
    show bmssp-label-witnesses G src settled (bmssp-distances G src)
      using loop unfold by simp
  qed
qed

lemma bmssp-distances-output-simple-walk:
  assumes wf: nat-graph-well-formed G
    and mem: (v, d) ∈ set (bmssp-distances G src)
  obtains p where
    p ≠ []
```

```
    hd p = src
    last p = v
    distinct p
    exec-walk (bmssp-vertices G src) (nat-graph-edge-list G) p
    exec-walk-weight (nat-graph-weight G) p = real d
proof −
  obtain settled where witnesses:
    bmssp-label-witnesses G src settled (bmssp-distances G src)
    using bmssp-distances-label-witnesses[OF wf] by blast
  then obtain p where
    p ≠ []
    hd p = src
    last p = v
    distinct p
    exec-walk (bmssp-vertices G src) (nat-graph-edge-list G) p
    exec-walk-weight (nat-graph-weight G) p = real d
    using mem unfolding bmssp-label-witnesses-def by blast
  then show ?thesis
    using that by blast
qed

lemma bmssp-distances-distinct-keys:
  distinct (map fst (bmssp-distances G src))
proof −
  let ?vertices = bmssp-vertices G src
  let ?P0 = bp-empty bmssp-block-size bmssp-bound
  let ?P1 =
    bp-result-of
      (c-bp-regularized-local-insert src (bmssp-partition-key src 0) ?P0)
  let ?fuel = Suc (length ?vertices * Suc (length G))
  have unfold:
    bmssp-distances G src =
      bmssp-loop ?fuel G ?vertices [] [(src, 0)] ?P1
    unfolding bmssp-distances-def by (simp add: Let-def)
  have distinct-loop:
    distinct
      (map fst (bmssp-loop ?fuel G ?vertices [] [(src, 0)] ?P1))
    by (rule bmssp-loop-preserves-distinct-output) simp
  show ?thesis
    using unfold distinct-loop by simp
qed

lemma bmssp-distances-keys-subset-bmssp-vertices:
  set (map fst (bmssp-distances G src)) ⊆ set (bmssp-vertices G src)
proof −
  let ?vertices = bmssp-vertices G src
  let ?P0 = bp-empty bmssp-block-size bmssp-bound
  let ?P1 =
    bp-result-of
```

```
    (c-bp-regularized-local-insert src (bmssp-partition-key src 0) ?P0)
  let ?fuel = Suc (length ?vertices * Suc (length G))
  have initial: set (map fst [(src, 0)]) ⊆ set ?vertices
    unfolding bmssp-vertices-def by simp
  have targets:
    ⋀a b w. (a, b, w) ∈ set G ⟹ b ∈ set ?vertices
    by (rule bmssp-edge-target-in-vertices)
  have set (map fst
      (bmssp-loop ?fuel G ?vertices [] [(src, 0)] ?P1))
      ⊆ set ?vertices
    by (rule bmssp-loop-output-keys-subset-vertices[OF initial targets])
  then show ?thesis
    unfolding bmssp-distances-def by (simp add: Let-def)
qed

lemma bmssp-distances-encode-own-keys:
  bmssp-distances G src =
    encode-dist-assoc-list (map fst (bmssp-distances G src))
      (executable-label-of (bmssp-distances G src))
proof −
  let ?vertices = bmssp-vertices G src
  let ?P0 = bp-empty bmssp-block-size bmssp-bound
  let ?P1 =
    bp-result-of
      (c-bp-regularized-local-insert src (bmssp-partition-key src 0) ?P0)
  let ?fuel = Suc (length ?vertices * Suc (length G))
  have loop:
    bmssp-loop ?fuel G ?vertices [] [(src, 0)] ?P1 =
      encode-dist-assoc-list
        (map fst (bmssp-loop ?fuel G ?vertices [] [(src, 0)] ?P1))
        (executable-label-of
          (bmssp-loop ?fuel G ?vertices [] [(src, 0)] ?P1))
    by (rule bmssp-loop-output-encode) simp
  show ?thesis
    using loop unfolding bmssp-distances-def by (simp add: Let-def)
qed

lemma nat-graph-finite-weighted-digraph:
  assumes wf: nat-graph-well-formed G
    and src: src ∈ nat-graph-vertices G
  shows finite-weighted-digraph
    (nat-graph-vertices G) (nat-graph-edges G) (nat-graph-weight G) src
proof
  show finite (nat-graph-vertices G)
    unfolding nat-graph-vertices-def by simp
  show src ∈ nat-graph-vertices G
    by (rule src)
  show ⋀u v. (u, v) ∈ nat-graph-edges G ⟹
    u ∈ nat-graph-vertices G ∧ v ∈ nat-graph-vertices G
```

```
    using nat-graph-edge-in-vertices by blast
  show ⋀u v. (u, v) ∈ nat-graph-edges G ⟹
    0 ≤ nat-graph-weight G u v
    by simp
qed

lemma nat-graph-reachable-iff-exec-reachable:
  assumes wf: nat-graph-well-formed G
    and src: src ∈ nat-graph-vertices G
  shows exec-reachable (nat-graph-vertex-list G) (nat-graph-edge-list G) a b
    ⟷ nat-graph-reachable G a b
proof −
  interpret concrete: finite-weighted-digraph
    nat-graph-vertices G nat-graph-edges G nat-graph-weight G src
    by (rule nat-graph-finite-weighted-digraph[OF wf src])
  have exec:
      exec-reachable (nat-graph-vertex-list G) (nat-graph-edge-list G) a b
        ⟷ concrete.reachable a b
    by (rule concrete.exec-reachable-iff-reachable
        [OF nat-graph-vertex-list-set nat-graph-edge-list-set])
  show ?thesis
    using exec unfolding nat-graph-reachable-def .
qed

lemma nat-graph-exec-dist-eq-dist:
  assumes wf: nat-graph-well-formed G
    and src: src ∈ nat-graph-vertices G
    and reach: nat-graph-reachable G a b
  shows exec-dist (nat-graph-vertex-list G) (nat-graph-edge-list G)
      (nat-graph-weight G) a b =
    nat-graph-dist G a b
proof −
  interpret concrete: finite-weighted-digraph
    nat-graph-vertices G nat-graph-edges G nat-graph-weight G src
    by (rule nat-graph-finite-weighted-digraph[OF wf src])
  have reach': concrete.reachable a b
    using reach unfolding nat-graph-reachable-def .
  show ?thesis
    unfolding nat-graph-dist-def
    by (rule concrete.exec-dist-eq-dist[OF nat-graph-vertex-list-set
        nat-graph-edge-list-set reach'])
qed

lemma nat-graph-weight-integral:
  ∃n. nat-graph-weight G u v = real n
proof (cases map-of
    (map (λe. ((nat-edge-source e, nat-edge-target e),
          real (nat-edge-weight e))) G) (u, v))
  case None
```

```
  then show ?thesis
    unfolding nat-graph-weight-def by simp
next
  case (Some c)
  then have ((u, v), c) ∈ set
      (map (λe. ((nat-edge-source e, nat-edge-target e),
            real (nat-edge-weight e))) G)
    by (rule map-of-SomeD)
  then obtain e where e ∈ set G
    and c: c = real (nat-edge-weight e)
    by auto
  then show ?thesis
    unfolding nat-graph-weight-def using Some by auto
qed

lemma exec-walk-weight-nat-graph-integral:
  ∃ n. exec-walk-weight (nat-graph-weight G) p = real n
proof (induction p)
  case Nil
  then show ?case
    by simp
next
  case (Cons x xs)
  show ?case
  proof (cases xs)
    case Nil
    then show ?thesis
      by simp
  next
    case (Cons y ys)
    obtain n where w: nat-graph-weight G x y = real n
      using nat-graph-weight-integral[of G x y] by blast
    obtain m where tail:
        exec-walk-weight (nat-graph-weight G) (y # ys) = real m
      using Cons.IH Cons by blast
    have exec-walk-weight (nat-graph-weight G) (x # y # ys) =
        real (n + m)
      using w tail by simp
    then show ?thesis
      using Cons by blast
  qed
qed

lemma min-list-integral-nonempty:
  assumes xs ≠ []
    and ⋀x. x ∈ set xs ⟹ ∃ n. x = real n
  shows ∃ n. min-list xs = real n
  using assms
proof (induction xs)
```

```
  case Nil
  then show ?case
    by simp
next
  case (Cons x xs)
  obtain n where x: x = real n
    using Cons.prems(2) by auto
  show ?case
  proof (cases xs)
    case Nil
    then show ?thesis
      using x by simp
  next
    case (Cons y ys)
    have tail-nonempty: xs ≠ []
      using Cons by simp
    have tail-integral: ⋀z. z ∈ set xs ⟹ ∃ m. z = real m
      using Cons.prems by simp
    obtain m where tail: min-list xs = real m
      using Cons.IH[OF tail-nonempty tail-integral] by blast
    have min-list (x # xs) = min x (min-list xs)
      using Cons by simp
    also have ... = real (min n m)
      using x tail by simp
    finally have min-list (x # xs) = real (min n m) .
    then show ?thesis
      by blast
  qed
qed

lemma exec-dist-nat-graph-integral:
  ∃ n. exec-dist vs es (nat-graph-weight G) a b = real n
proof (cases exec-walk-weights vs es (nat-graph-weight G) a b)
  case Nil
  then show ?thesis
    unfolding exec-dist-def by simp
next
  case (Cons w ws)
  have integral:
    ⋀x. x ∈ set (w # ws) ⟹ ∃ n. x = real n
  proof −
    fix x
    assume x ∈ set (w # ws)
    then obtain p where
      p ∈ set (exec-simple-walks-betw vs es a b)
      and x: x = exec-walk-weight (nat-graph-weight G) p
      unfolding Cons[symmetric] exec-walk-weights-def by auto
    then show ∃ n. x = real n
      using exec-walk-weight-nat-graph-integral[of G p] by blast
```

```
  qed
  have nonempty: w # ws ≠ []
    by simp
  obtain n where min-list (w # ws) = real n
    using min-list-integral-nonempty[OF nonempty integral] by blast
  then show ?thesis
    unfolding exec-dist-def Cons by simp
qed

lemma nat-graph-dist-integral:
  assumes wf: nat-graph-well-formed G
    and src: src ∈ nat-graph-vertices G
    and reach: nat-graph-reachable G a b
  shows ∃ n. nat-graph-dist G a b = real n
proof −
  have exec-eq:
    exec-dist (nat-graph-vertex-list G) (nat-graph-edge-list G)
       (nat-graph-weight G) a b =
      nat-graph-dist G a b
    by (rule nat-graph-exec-dist-eq-dist[OF wf src reach])
  obtain n where exec-dist (nat-graph-vertex-list G) (nat-graph-edge-list G)
      (nat-graph-weight G) a b = real n
    using exec-dist-nat-graph-integral by blast
  then have nat-graph-dist G a b = real n
    using exec-eq by simp
  then show ?thesis
    by blast
qed

lemma real-nat-floor-integral:
  assumes ∃ n::nat. x = real n
  shows real (nat (floor x)) = x
  using assms by auto

locale nat-graph-instance =
  fixes G :: nat-graph
    and src :: nat
  assumes wf: nat-graph-well-formed G
    and src-in: src ∈ nat-graph-vertices G
begin

interpretation concrete: finite-weighted-digraph
  nat-graph-vertices G nat-graph-edges G nat-graph-weight G src
  by (rule nat-graph-finite-weighted-digraph[OF wf src-in])

lemma exec-reachable-iff-reachable:
  exec-reachable (nat-graph-vertex-list G) (nat-graph-edge-list G) a b
    ⟷ concrete.reachable a b
  by (rule concrete.exec-reachable-iff-reachable
```

[*OF nat-graph-vertex-list-set nat-graph-edge-list-set*])

lemma *exec-dist-eq-dist*:
assumes *concrete.reachable a b*
shows *exec-dist* (*nat-graph-vertex-list G*) (*nat-graph-edge-list G*)
(*nat-graph-weight G*) *a b* =
concrete.dist a b
by (*rule concrete.exec-dist-eq-dist*[*OF nat-graph-vertex-list-set*
nat-graph-edge-list-set assms])

lemma *bmssp-vertices-carrier*:
set (*bmssp-vertices G src*) = *nat-graph-vertices G*
using *bmssp-vertices-set-if-source*[*OF src-in*] **.**

lemma *bmssp-distances-keys-subset-carrier*:
set (*map fst* (*bmssp-distances G src*)) ⊆ *nat-graph-vertices G*
using *bmssp-distances-keys-subset-bmssp-vertices*[*of G src*]
bmssp-vertices-carrier **by** *simp*

lemma *bmssp-distances-encode-filtered-vertex-list*:
bmssp-distances G src =
encode-dist-assoc-list
(*filter* (λ*v*. *v* ∈ *set* (*map fst* (*bmssp-distances G src*)))
(*nat-graph-vertex-list G*))
(*executable-label-of* (*bmssp-distances G src*))
proof −
let *?U* = *set* (*map fst* (*bmssp-distances G src*))
have *filter-set*:
set (*filter* (λ*v*. *v* ∈ *?U*) (*nat-graph-vertex-list G*)) =
set (*map fst* (*bmssp-distances G src*))
using *bmssp-distances-keys-subset-carrier* **by** *auto*
have *own*:
bmssp-distances G src =
encode-dist-assoc-list (*map fst* (*bmssp-distances G src*))
(*executable-label-of* (*bmssp-distances G src*))
by (*rule bmssp-distances-encode-own-keys*)
have *encode-eq*:
encode-dist-assoc-list
(*filter* (λ*v*. *v* ∈ *?U*) (*nat-graph-vertex-list G*))
(*executable-label-of* (*bmssp-distances G src*)) =
encode-dist-assoc-list (*map fst* (*bmssp-distances G src*))
(*executable-label-of* (*bmssp-distances G src*))
by (*rule encode-dist-assoc-list-cong-set*[*OF filter-set*])
show *?thesis*
using *own encode-eq* **by** *simp*
qed

lemma *bmssp-distances-output-abstract-simple-walk*:
assumes *mem*: (*v*, *d*) ∈ *set* (*bmssp-distances G src*)

```
  obtains p where
    concrete.simple-walk-betw src p v
    concrete.walk-weight p = real d
proof −
  obtain p where p:
    p ≠ []
    hd p = src
    last p = v
    distinct p
    exec-walk (bmssp-vertices G src) (nat-graph-edge-list G) p
    exec-walk-weight (nat-graph-weight G) p = real d
    by (rule bmssp-distances-output-simple-walk[OF wf mem])
  have walk-p: concrete.walk p
    using p(5)
    by (simp add: concrete.exec-walk-iff-walk
        [OF bmssp-vertices-carrier nat-graph-edge-list-set])
  have simple: concrete.simple-walk-betw src p v
    using p(1−4) walk-p
    unfolding concrete.simple-walk-betw-def concrete.walk-betw-def by blast
  have weight: concrete.walk-weight p = real d
    using p(6) by simp
  show ?thesis
    by (rule that[OF simple weight])
qed

lemma bmssp-distances-output-reachable:
  assumes (v, d) ∈ set (bmssp-distances G src)
  shows nat-graph-reachable G src v
proof −
  obtain p where concrete.simple-walk-betw src p v
    by (rule bmssp-distances-output-abstract-simple-walk[OF assms])
  then have concrete.reachable src v
    unfolding concrete.reachable-def by blast
  then show ?thesis
    unfolding nat-graph-reachable-def .
qed

lemma bmssp-distances-output-dist-le:
  assumes mem: (v, d) ∈ set (bmssp-distances G src)
  shows nat-graph-dist G src v ≤ real d
proof −
  obtain p where simple: concrete.simple-walk-betw src p v
    and weight: concrete.walk-weight p = real d
    by (rule bmssp-distances-output-abstract-simple-walk[OF mem])
  have weight-mem:
    real d ∈ concrete.simple-walk-weights src v
  proof −
    have concrete.walk-weight p ∈ concrete.walk-weight ‘
        {p. concrete.simple-walk-betw src p v}
```

```
    using simple by blast
  then show ?thesis
    using weight unfolding concrete.simple-walk-weights-def by simp
 qed
 have concrete.dist src v ≤ real d
  unfolding concrete.dist-def
  by (rule Min-le[OF concrete.finite-simple-walk-weights weight-mem])
 then show ?thesis
  unfolding nat-graph-dist-def by simp
qed

lemma bmssp-label-witnesses-lookup-abstract-simple-walk:
 assumes witnesses: bmssp-label-witnesses G src settled ds
  and lookup: bmssp-lookup-dist ds v = Some d
 obtains p where
  concrete.simple-walk-betw src p v
  concrete.walk-weight p = real d
proof −
 have mem: (v, d) ∈ set ds
  by (rule bmssp-lookup-dist-Some-mem[OF lookup])
 then obtain p where p:
  p ≠ []
  hd p = src
  last p = v
  distinct p
  exec-walk (bmssp-vertices G src) (nat-graph-edge-list G) p
  exec-walk-weight (nat-graph-weight G) p = real d
  using witnesses unfolding bmssp-label-witnesses-def by blast
 have walk-p: concrete.walk p
  using p(5)
  by (simp add: concrete.exec-walk-iff-walk
      [OF bmssp-vertices-carrier nat-graph-edge-list-set])
 have simple: concrete.simple-walk-betw src p v
  using p(1−4) walk-p
  unfolding concrete.simple-walk-betw-def concrete.walk-betw-def by blast
 have weight: concrete.walk-weight p = real d
  using p(6) by simp
 show ?thesis
  by (rule that[OF simple weight])
qed

lemma bmssp-label-witnesses-lookup-reachable:
 assumes witnesses: bmssp-label-witnesses G src settled ds
  and lookup: bmssp-lookup-dist ds v = Some d
 shows nat-graph-reachable G src v
proof −
 obtain p where concrete.simple-walk-betw src p v
  by (rule bmssp-label-witnesses-lookup-abstract-simple-walk
      [OF witnesses lookup])
```

```
  then have concrete.reachable src v
    unfolding concrete.reachable-def by blast
  then show ?thesis
    unfolding nat-graph-reachable-def .
qed

lemma bmssp-label-witnesses-lookup-dist-le:
  assumes witnesses: bmssp-label-witnesses G src settled ds
    and lookup: bmssp-lookup-dist ds v = Some d
  shows nat-graph-dist G src v ≤ real d
proof −
  obtain p where simple: concrete.simple-walk-betw src p v
    and weight: concrete.walk-weight p = real d
    by (rule bmssp-label-witnesses-lookup-abstract-simple-walk
        [OF witnesses lookup])
  have weight-mem:
    real d ∈ concrete.simple-walk-weights src v
  proof −
    have concrete.walk-weight p ∈ concrete.walk-weight ‘
        {p. concrete.simple-walk-betw src p v}
      using simple by blast
    then show ?thesis
      using weight unfolding concrete.simple-walk-weights-def by simp
  qed
  have concrete.dist src v ≤ real d
    unfolding concrete.dist-def
    by (rule Min-le[OF concrete.finite-simple-walk-weights weight-mem])
  then show ?thesis
    unfolding nat-graph-dist-def by simp
qed

lemma bmssp-shortest-walk-first-unsettled-lookup-le-dist:
  assumes sp: concrete.shortest-walk src p u
    and u-unsettled: u ∉ set settled
    and source-zero: bmssp-source-zero src ds
    and settled-lookup: bmssp-settled-have-lookups settled ds
    and exact: bmssp-settled-exact G src settled ds
    and frontier: bmssp-frontier-relaxed G src settled ds
  obtains y dy where
    y ∈ set p
    y ∉ set settled
    bmssp-lookup-dist ds y = Some dy
    real dy ≤ nat-graph-dist G src y
    nat-graph-dist G src y ≤ nat-graph-dist G src u
proof −
  have p-ne: p ≠ [] and hd-p: hd p = src and last-p: last p = u
    using concrete.shortest-walk-hd[OF sp] by blast+
  have last-idx: length p − 1 < length p
    using p-ne by simp
```

have *last-nth*: $p\ !\ (length\ p - 1) = u$
  using *last-conv-nth*[*OF p-ne*] *last-p* **by** *simp*
let *?P* = $\lambda i.\ i < length\ p \wedge p\ !\ i \notin set\ settled$
have *ex-unsettled*: $\exists\, i.\ ?P\ i$
  using *last-idx last-nth u-unsettled* **by** *blast*
define *i* **where** $i = (LEAST\ i.\ ?P\ i)$
have *i-props*: $i < length\ p\ p\ !\ i \notin set\ settled$
  using *LeastI-ex*[*OF ex-unsettled*] **unfolding** *i-def* **by** *blast+*
have *before-settled*:
  $\bigwedge j.\ j < i \Longrightarrow p\ !\ j \in set\ settled$
proof −
  fix *j*
  assume *j-lt*: $j < i$
  then have *j-len*: $j < length\ p$
    using *i-props* **by** *simp*
  show $p\ !\ j \in set\ settled$
  proof (*rule ccontr*)
    assume $p\ !\ j \notin set\ settled$
    then have *?P j*
      using *j-len* **by** *blast*
    then have $i \leq j$
      unfolding *i-def* **by** (*rule Least-le*)
    then show *False*
      using *j-lt* **by** *simp*
  qed
qed
let *?y* = $p\ !\ i$
have *y-mem*: $?y \in set\ p$
  using *nth-mem*[*OF i-props*(*1*)] **.**
have *y-dist-u*: $nat\text{-}graph\text{-}dist\ G\ src\ ?y \leq nat\text{-}graph\text{-}dist\ G\ src\ u$
proof −
  have $concrete.dist\ src\ ?y \leq concrete.dist\ src\ u$
    using *concrete.shortest-walk-prefix-dist-le*[*OF sp y-mem*] **.**
  then show *?thesis*
    unfolding *nat-graph-dist-def* **by** *simp*
qed
have *y-lookup-le*: $\exists\, dy.\ bmssp\text{-}lookup\text{-}dist\ ds\ ?y = Some\ dy \wedge$
    $real\ dy \leq nat\text{-}graph\text{-}dist\ G\ src\ ?y$
proof (*cases i*)
  case *0*
  then have *y-src*: $?y = src$
    using *hd-p p-ne* **by** (*cases p*) *auto*
  have *lookup*: $bmssp\text{-}lookup\text{-}dist\ ds\ ?y = Some\ 0$
    using *source-zero y-src* **unfolding** *bmssp-source-zero-def* **by** *simp*
  have $real\ 0 \leq nat\text{-}graph\text{-}dist\ G\ src\ ?y$
    using *concrete.dist-refl-zero*[*OF src-in*] *y-src*
    unfolding *nat-graph-dist-def* **by** *simp*
  then show *?thesis*
    using *lookup* **by** *blast*

```
  next
    case (Suc j)
    let ?x = p ! j
    have j-lt-i: j < i
      using Suc by simp
    have j-len: j < length p
      using j-lt-i i-props by simp
    have x-settled: ?x ∈ set settled
      by (rule before-settled[OF j-lt-i])
    obtain dx where lookup-x: bmssp-lookup-dist ds ?x = Some dx
      using settled-lookup x-settled
      unfolding bmssp-settled-have-lookups-def by blast
    have exact-x: real dx = nat-graph-dist G src ?x
      using exact x-settled lookup-x
      unfolding bmssp-settled-exact-def by blast
    have suc-i: Suc j = i
      using Suc by simp
    have edge-idx: Suc j < length p
      using i-props suc-i by simp
    have tight: concrete.tight-edge-step ?x ?y
      using concrete.shortest-walk-successively-tight[OF sp]
      unfolding successively-conv-nth
      using edge-idx suc-i by blast
    have edge: (?x, ?y) ∈ nat-graph-edges G
      using tight unfolding concrete.tight-edge-step-def by blast
    obtain w where edge-w: (?x, ?y, w) ∈ set G
      using edge unfolding nat-graph-edges-def nat-graph-edge-list-def by auto
    obtain dy where lookup-y: bmssp-lookup-dist ds ?y = Some dy
      and dy-le: real dy ≤ real dx + real w
      using frontier edge-w x-settled i-props(2) lookup-x exact-x
      unfolding bmssp-frontier-relaxed-def by blast
    have weight-eq: nat-graph-weight G ?x ?y = real w
      by (rule nat-graph-weight-of-edge[OF wf edge-w])
    have dist-y-eq:
      nat-graph-dist G src ?y =
        nat-graph-dist G src ?x + real w
      using tight weight-eq
      unfolding concrete.tight-edge-step-def nat-graph-dist-def by simp
    have real dy ≤ nat-graph-dist G src ?y
      using dy-le exact-x dist-y-eq by simp
    then show ?thesis
      using lookup-y by blast
  qed
  obtain dy where lookup-y: bmssp-lookup-dist ds ?y = Some dy
    and dy-le: real dy ≤ nat-graph-dist G src ?y
    using y-lookup-le by blast
  show thesis
    by (rule that[OF y-mem i-props(2) lookup-y dy-le y-dist-u])
qed
```

```
lemma bmssp-partition-state-pulled-label-le-dist:
  assumes match: bmssp-partition-state-match vertices settled ds P
    and source-zero: bmssp-source-zero src ds
    and settled-lookup: bmssp-settled-have-lookups settled ds
    and witnesses: bmssp-label-witnesses G src settled ds
    and exact: bmssp-settled-exact G src settled ds
    and frontier: bmssp-frontier-relaxed G src settled ds
    and pull: pull-separates (bp-view P) bmssp-block-size B S beta
      (bp-view P1)
    and uS: u ∈ S
    and lookup-u: bmssp-lookup-dist ds u = Some du
  shows real du ≤ nat-graph-dist G src u
proof −
  have u-unsettled: u ∉ set settled
    by (rule bmssp-partition-state-pulled-not-settled[OF match pull uS])
  have reach-u: concrete.reachable src u
    using bmssp-label-witnesses-lookup-reachable[OF witnesses lookup-u]
    unfolding nat-graph-reachable-def .
  obtain p where sp: concrete.shortest-walk src p u
    using concrete.shortest-walk-exists[OF reach-u] by blast
  obtain y dy where y-mem: y ∈ set p
    and y-unsettled: y ∉ set settled
    and lookup-y: bmssp-lookup-dist ds y = Some dy
    and dy-le-dist: real dy ≤ nat-graph-dist G src y
    and dist-y-le-u: nat-graph-dist G src y ≤ nat-graph-dist G src u
    by (rule bmssp-shortest-walk-first-unsettled-lookup-le-dist
        [OF sp u-unsettled source-zero settled-lookup exact frontier])
  have du-le-dy: du ≤ dy
  proof (cases y ∈ S)
    case True
    have S-subset: S ⊆ keys-of (bp-view P)
      by (rule pull-separates-subset[OF pull])
    have finite-S: finite S
      by (rule finite-subset[OF S-subset]) simp
    have card-S: card S ≤ 1
      using pull-separates-card[OF pull] unfolding bmssp-block-size-def .
    have u = y
      by (rule finite-card-le-one-member-eq[OF finite-S card-S uS True])
    then show ?thesis
      using lookup-u lookup-y by simp
  next
    case False
    show ?thesis
      by (rule bmssp-partition-state-pull-residual-label-le
          [OF match pull uS lookup-u lookup-y y-unsettled False])
  qed
  have real du ≤ real dy
    using du-le-dy by simp
```

```
  also have ... ≤ nat-graph-dist G src y
    by (rule dy-le-dist)
  also have ... ≤ nat-graph-dist G src u
    by (rule dist-y-le-u)
  finally show ?thesis .
qed

lemma bmssp-partition-state-pulled-label-exact:
  assumes match: bmssp-partition-state-match vertices settled ds P
    and source-zero: bmssp-source-zero src ds
    and settled-lookup: bmssp-settled-have-lookups settled ds
    and witnesses: bmssp-label-witnesses G src settled ds
    and exact: bmssp-settled-exact G src settled ds
    and frontier: bmssp-frontier-relaxed G src settled ds
    and pull: pull-separates (bp-view P) bmssp-block-size B S beta
      (bp-view P1)
    and uS: u ∈ S
    and lookup-u: bmssp-lookup-dist ds u = Some du
  shows real du = nat-graph-dist G src u
proof −
  have lower: nat-graph-dist G src u ≤ real du
    by (rule bmssp-label-witnesses-lookup-dist-le
        [OF witnesses lookup-u])
  have upper: real du ≤ nat-graph-dist G src u
    by (rule bmssp-partition-state-pulled-label-le-dist
        [OF match source-zero settled-lookup witnesses exact frontier
          pull uS lookup-u])
  show ?thesis
    using lower upper by simp
qed

lemma bmssp-settled-exact-partition-step:
  assumes match: bmssp-partition-state-match vertices old-settled ds P
    and source-zero: bmssp-source-zero src ds
    and settled-lookup: bmssp-settled-have-lookups old-settled ds
    and witnesses: bmssp-label-witnesses G src old-settled ds
    and exact: bmssp-settled-exact G src old-settled ds
    and frontier: bmssp-frontier-relaxed G src old-settled ds
    and pull: pull-separates (bp-view P) bmssp-block-size B S beta
      (bp-view P1)
    and pulled-def:
      pulled = filter (λx. x ∈ S ∧ x ∉ set old-settled)
        vertices
    and settled-def: settled = pulled @ old-settled
    and relaxed: bmssp-relax-vertices G settled pulled ds = (updates, ds')
  shows bmssp-settled-exact G src settled ds'
proof −
  have pulled-exact:
    ⋀v d. ⟦v ∈ set pulled; bmssp-lookup-dist ds v = Some d⟧
```

```
    ⟹ real d = nat-graph-dist G src v
  proof −
    fix v d
    assume v-pulled: v ∈ set pulled
      and lookup-v: bmssp-lookup-dist ds v = Some d
    have vS: v ∈ S
      using v-pulled unfolding pulled-def by simp
    show real d = nat-graph-dist G src v
      by (rule bmssp-partition-state-pulled-label-exact
          [OF match source-zero settled-lookup witnesses exact frontier
            pull vS lookup-v])
  qed
  show ?thesis
    by (rule bmssp-settled-exact-step
        [OF exact pulled-exact settled-def relaxed])
qed

lemma bmssp-dijkstra-state-step-bridge:
  fixes old-settled settled vertices :: nat list
    and pulled :: nat list
    and updates :: (nat × real) list
  assumes state: bmssp-dijkstra-state G src vertices old-settled ds P
    and edge-targets:
      ⋀a b w. (a, b, w) ∈ set G ⟹ b ∈ set vertices
    and pulled-def:
      pulled = filter (λx. x ∈ S ∧ x ∉ set old-settled)
        vertices
    and settled-def: settled = pulled @ old-settled
    and relaxed: bmssp-relax-vertices G settled pulled ds = (updates, ds′)
    and P2-def: P2 = bmssp-apply-updates updates P1
    and distinct-vertices: distinct vertices
    and ord: bp-ordered-invariant P
    and upper: partition-upper-bound (bp-view P) B
    and pull: bp-pull bmssp-block-size B P = (S, beta, P1)
  shows bmssp-dijkstra-state G src vertices settled ds′ P2
proof −
  have match: bmssp-partition-state-match vertices old-settled ds P
    and zero: bmssp-source-zero src ds
    and settled-lookup: bmssp-settled-have-lookups old-settled ds
    and witnesses: bmssp-label-witnesses G src old-settled ds
    and exact: bmssp-settled-exact G src old-settled ds
    and frontier: bmssp-frontier-relaxed G src old-settled ds
    using state unfolding bmssp-dijkstra-state-def by auto
  have distinct-ds: distinct (map fst ds)
    using match unfolding bmssp-partition-state-match-def by simp
  have updates-def: updates = fst (bmssp-relax-vertices G settled pulled ds)
    using relaxed by simp
  have sep:
    pull-separates (bp-view P) bmssp-block-size B S beta (bp-view P1)
```

```
    by (rule bmssp-loop-partition-step-bridge(1)
        [OF pulled-def updates-def P2-def wf distinct-vertices ord upper
          pull])
  have match':
    bmssp-partition-state-match vertices settled ds' P2
    by (rule bmssp-partition-state-match-step-bridge
        [OF match edge-targets pulled-def settled-def relaxed P2-def wf
          distinct-vertices ord upper pull])
  have zero': bmssp-source-zero src ds'
    by (rule bmssp-source-zero-step[OF zero distinct-ds relaxed])
  have settled-lookup':
    bmssp-settled-have-lookups settled ds'
    by (rule bmssp-settled-have-lookups-step
        [OF settled-lookup match sep pulled-def settled-def relaxed])
  have settled-subset: set old-settled ⊆ set settled
    using settled-def by auto
  have witnesses-settled:
    bmssp-label-witnesses G src settled ds
    by (rule bmssp-label-witnesses-mono-settled
        [OF witnesses settled-subset])
  have pulled-subset: set pulled ⊆ set settled
    using settled-def by simp
  have witnesses':
    bmssp-label-witnesses G src settled ds'
  proof −
    have bmssp-label-witnesses G src settled
        (snd (bmssp-relax-vertices G settled pulled ds))
      by (rule bmssp-relax-vertices-preserves-label-witnesses
          [OF wf witnesses-settled pulled-subset])
    then show ?thesis
      using relaxed by simp
  qed
  have exact': bmssp-settled-exact G src settled ds'
    by (rule bmssp-settled-exact-partition-step
        [OF match zero settled-lookup witnesses exact frontier sep
          pulled-def settled-def relaxed])
  have frontier': bmssp-frontier-relaxed G src settled ds'
    by (rule bmssp-frontier-relaxed-step
        [OF frontier distinct-ds settled-def relaxed])
  show ?thesis
    unfolding bmssp-dijkstra-state-def
    using match' zero' settled-lookup' witnesses' exact' frontier' by simp
qed

lemma bmssp-dijkstra-state-step-upper-bound:
  fixes old-settled settled vertices :: nat list
    and pulled :: nat list
    and updates :: (nat × real) list
  assumes state: bmssp-dijkstra-state G src vertices old-settled ds P
```

```
    and wf: nat-graph-well-formed G
    and pulled-def:
      pulled =
        filter (λx. x ∈ S ∧ x ∉ set old-settled)
          vertices
    and settled-def: settled = pulled @ old-settled
    and relaxed: bmssp-relax-vertices G settled pulled ds = (updates, ds')
    and P2-def: P2 = bmssp-apply-updates updates P1
    and distinct-vertices: distinct vertices
    and ord: bp-ordered-invariant P
    and upper: partition-upper-bound (bp-view P) bmssp-bound
    and pull: bp-pull bmssp-block-size bmssp-bound P = (S, beta, P1)
  shows partition-upper-bound (bp-view P2) bmssp-bound
proof −
  have witnesses-old: bmssp-label-witnesses G src old-settled ds
    using state unfolding bmssp-dijkstra-state-def by simp
  have updates-def: updates = fst (bmssp-relax-vertices G settled pulled ds)
    using relaxed by simp
  have distinct-updates: distinct (map fst updates)
    by (rule bmssp-loop-partition-step-bridge(5)
        [OF pulled-def updates-def P2-def wf distinct-vertices ord upper pull])
  have settled-subset: set old-settled ⊆ set settled
    using settled-def by auto
  have witnesses:
    bmssp-label-witnesses G src settled ds
    by (rule bmssp-label-witnesses-mono-settled
        [OF witnesses-old settled-subset])
  have pulled-subset: set pulled ⊆ set settled
    using settled-def by simp
  have update-lt:
    ⋀x b. (x, b) ∈ set updates ⟹ b < bmssp-bound
  proof −
    fix x b
    assume (x, b) ∈ set updates
    then have update-rel:
      (x, b) ∈ set
        (fst (bmssp-relax-vertices G settled pulled ds))
      using relaxed by simp
    have distinct-rel:
      distinct
        (map fst (fst (bmssp-relax-vertices G settled pulled ds)))
      using distinct-updates relaxed by simp
    show b < bmssp-bound
      by (rule bmssp-relax-vertices-update-lt-bound
          [OF wf witnesses pulled-subset distinct-rel update-rel])
  qed
  show ?thesis
    by (rule bmssp-loop-partition-step-bridge(9)
        [OF pulled-def updates-def P2-def wf distinct-vertices ord upper
```

```
        pull update-lt])
qed

definition bmssp-singleton-bucket-shape ::
  nat bucketed-partition ⇒ bool where
  bmssp-singleton-bucket-shape P ⟷
    bp-block-size P = bmssp-block-size ∧
    (∀ b∈set (bp-buckets P).
      ∃ p. bp-bucket-entries b = [p] ∧ bp-marker b = snd p)

lemma bmssp-bucketize-sorted-entries-aux-singleton-shape:
  ∀ b∈set (bp-bucketize-sorted-entries-aux fuel 1 xs).
    ∃ p. bp-bucket-entries b = [p] ∧ bp-marker b = snd p
proof (induction fuel arbitrary: xs)
  case 0
  then show ?case
    by simp
next
  case (Suc fuel)
  show ?case
  proof (cases xs)
    case Nil
    then show ?thesis
      by simp
  next
    case (Cons p ps)
    have tail:
      ∀ b∈set (bp-bucketize-sorted-entries-aux fuel 1 (drop 1 xs)).
        ∃ p. bp-bucket-entries b = [p] ∧ bp-marker b = snd p
      by (rule Suc.IH)
    show ?thesis
    proof
      fix b
      assume b:
        b ∈ set (bp-bucketize-sorted-entries-aux (Suc fuel) 1 xs)
      have b-cases:
        b = bp-make-bucket [p] ∨
         b ∈ set (bp-bucketize-sorted-entries-aux fuel 1 ps)
        using b Cons by auto
      then show ∃ p. bp-bucket-entries b = [p] ∧ bp-marker b = snd p
      proof
        assume b-def: b = bp-make-bucket [p]
        show ?thesis
          unfolding b-def bp-make-bucket-def
          by (rule exI[of - p]) simp
      next
        assume b ∈ set (bp-bucketize-sorted-entries-aux fuel 1 ps)
        then show ?thesis
          using tail Cons by simp
```

```
    qed
  qed
 qed
qed

lemma bmssp-bucketize-entries-singleton-shape:
 ∀ b∈set (bp-bucketize-entries bmssp-block-size xs).
  ∃ p. bp-bucket-entries b = [p] ∧ bp-marker b = snd p
 unfolding bp-bucketize-entries-def bp-bucketize-sorted-entries-def
  bmssp-block-size-def
 by (rule bmssp-bucketize-sorted-entries-aux-singleton-shape)

lemma bmssp-rebucket-singleton-shape:
 assumes block: bp-block-size P = bmssp-block-size
 shows bmssp-singleton-bucket-shape (bp-rebucket P)
 unfolding bmssp-singleton-bucket-shape-def bp-rebucket-def
 using block bmssp-bucketize-entries-singleton-shape[of bp-entries P]
 by simp

lemma bmssp-singleton-bucket-shape-drop-empty-prefix-id:
 assumes shape:
  ∀ b∈set bs. ∃ p. bp-bucket-entries b = [p] ∧ bp-marker b = snd p
 shows bp-drop-empty-prefix bs = bs
 using shape
 by (induction bs) auto

lemma bmssp-singleton-bucket-shape-flat-length:
 assumes shape:
  ∀ b∈set bs. ∃ p. bp-bucket-entries b = [p] ∧ bp-marker b = snd p
 shows length (bp-bucket-entries-flat bs) = length bs
 using shape
proof (induction bs)
 case Nil
 then show ?case
  unfolding bp-bucket-entries-flat-def by simp
next
 case (Cons b bs)
 obtain p where b-entries: bp-bucket-entries b = [p]
  and b-marker: bp-marker b = snd p
 proof −
  have ∃ p. bp-bucket-entries b = [p] ∧ bp-marker b = snd p
   using Cons.prems by simp
  then show ?thesis
   by (elim exE conjE) (rule that)
 qed
 have tail:
  ∀ b∈set bs. ∃ p. bp-bucket-entries b = [p] ∧
   bp-marker b = snd p
  using Cons.prems by simp
```

```
  have length (bp-bucket-entries-flat bs) = length bs
    by (rule Cons.IH[OF tail])
  then show ?case
    using b-entries unfolding bp-bucket-entries-flat-def by simp
qed

lemma bmssp-singleton-bucket-shape-entries-empty-buckets:
  assumes shape: bmssp-singleton-bucket-shape P
    and empty: bp-entries P = []
  shows bp-buckets P = []
proof (cases bp-buckets P)
  case Nil
  then show ?thesis .
next
  case (Cons b bs)
  obtain p where b-entries: bp-bucket-entries b = [p]
    using shape Cons unfolding bmssp-singleton-bucket-shape-def by auto
  then have bp-entries P ≠ []
    using Cons unfolding bp-entries-def bp-bucket-entries-flat-def by simp
  then show ?thesis
    using empty by simp
qed

lemma bmssp-singleton-bucket-shape-trim:
  assumes shape: bmssp-singleton-bucket-shape P
  shows bmssp-trim-empty-prefix P = P
proof −
  have buckets:
    ∀ b∈set (bp-buckets P).
      ∃ p. bp-bucket-entries b = [p] ∧ bp-marker b = snd p
    using shape unfolding bmssp-singleton-bucket-shape-def by simp
  have bp-drop-empty-prefix (bp-buckets P) = bp-buckets P
    by (rule bmssp-singleton-bucket-shape-drop-empty-prefix-id[OF buckets])
  then show ?thesis
    unfolding bmssp-trim-empty-prefix-def by simp
qed

lemma bmssp-singleton-bucket-shape-delete-first-bucket:
  assumes buckets: bp-buckets P = b # bs
    and shape: bmssp-singleton-bucket-shape P
    and distinct: distinct (map fst (bp-entries P))
  shows bmssp-singleton-bucket-shape
    (bmssp-trim-empty-prefix (bp-delete-keys (bp-bucket-keys b) P))
proof −
  obtain p where b-entries: bp-bucket-entries b = [p]
    and b-marker: bp-marker b = snd p
    using buckets shape unfolding bmssp-singleton-bucket-shape-def by auto
  have tail-shape:
    ∀ c∈set bs. ∃ q. bp-bucket-entries c = [q] ∧ bp-marker c = snd q
```

```
  using buckets shape unfolding bmssp-singleton-bucket-shape-def by auto
have tail-unchanged:
  map (bp-delete-keys-from-bucket (bp-bucket-keys b)) bs = bs
proof (rule map-idI)
  fix c
  assume c: c ∈ set bs
  obtain q where c-entries: bp-bucket-entries c = [q]
    and c-marker: bp-marker c = snd q
  proof −
    have ∃ q. bp-bucket-entries c = [q] ∧ bp-marker c = snd q
      using tail-shape c by simp
    then show ?thesis
      by (elim exE conjE) (rule that)
  qed
  have fst-ne: fst q ≠ fst p
  proof
    assume eq: fst q = fst p
    have entries:
      bp-entries P = p # bp-bucket-entries-flat bs
      using buckets b-entries unfolding bp-entries-def
        bp-bucket-entries-flat-def by simp
    have q-flat: q ∈ set (bp-bucket-entries-flat bs)
      using c c-entries
      unfolding bp-bucket-entries-flat-def
      by (induction bs) auto
    then have fst q ∈ set (map fst (bp-bucket-entries-flat bs))
      by simp
    then show False
      using distinct eq entries by simp
  qed
  have fst q ∉ bp-bucket-keys b
    using fst-ne b-entries unfolding bp-bucket-keys-def bp-entry-keys-def
    by simp
  then have keep-c:
    filter (λr. fst r ∉ bp-bucket-keys b)
      (bp-bucket-entries c) = bp-bucket-entries c
    using c-entries by simp
  then show bp-delete-keys-from-bucket (bp-bucket-keys b) c = c
    unfolding bp-delete-keys-from-bucket-def by simp
qed
have deleted-head:
  bp-delete-keys-from-bucket (bp-bucket-keys b) b =
    b(|bp-bucket-entries := []|)
  unfolding bp-delete-keys-from-bucket-def b-entries
    bp-bucket-keys-def bp-entry-keys-def by simp
have buckets-deleted:
  bp-buckets (bp-delete-keys (bp-bucket-keys b) P) =
    b(|bp-bucket-entries := []|) # bs
  using buckets tail-unchanged deleted-head
```

```
    unfolding bp-delete-keys-def by simp
  have trimmed-buckets:
    bp-buckets
      (bmssp-trim-empty-prefix (bp-delete-keys (bp-bucket-keys b) P)) =
      bs
  proof −
    have tail-drop: bp-drop-empty-prefix bs = bs
      by (rule bmssp-singleton-bucket-shape-drop-empty-prefix-id[OF tail-shape])
    show ?thesis
      using buckets-deleted tail-drop
      unfolding bmssp-trim-empty-prefix-def by simp
  qed
  have block:
    bp-block-size
      (bmssp-trim-empty-prefix (bp-delete-keys (bp-bucket-keys b) P)) =
      bmssp-block-size
    using shape unfolding bmssp-singleton-bucket-shape-def
      bmssp-trim-empty-prefix-def bp-delete-keys-def by simp
  show ?thesis
    unfolding bmssp-singleton-bucket-shape-def
    using block trimmed-buckets tail-shape by simp
qed

lemma bmssp-singleton-bucket-shape-delete-all-small:
  assumes shape: bmssp-singleton-bucket-shape P
    and small: length (bp-entries P) ≤ bmssp-block-size
  shows bmssp-singleton-bucket-shape
    (bmssp-trim-empty-prefix
      (bp-delete-keys (bp-entry-keys (bp-entries P)) P))
proof −
  have block: bp-block-size P = bmssp-block-size
    using shape unfolding bmssp-singleton-bucket-shape-def by simp
  have singleton:
    ∀ b∈set (bp-buckets P).
      ∃ p. bp-bucket-entries b = [p] ∧ bp-marker b = snd p
    using shape unfolding bmssp-singleton-bucket-shape-def by simp
  have entries-len:
    length (bp-entries P) = length (bp-buckets P)
  proof −
    have length (bp-bucket-entries-flat (bp-buckets P)) =
        length (bp-buckets P)
      by (rule bmssp-singleton-bucket-shape-flat-length[OF singleton])
    then show ?thesis
      unfolding bp-entries-def .
  qed
  have buckets-len: length (bp-buckets P) ≤ 1
    using small entries-len unfolding bmssp-block-size-def by simp
  have trimmed-empty:
    bp-drop-empty-prefix
```

```
    (bp-buckets (bp-delete-keys (bp-entry-keys (bp-entries P)) P)) = []
  proof (cases bp-buckets P)
    case Nil
    then show ?thesis
      unfolding bp-delete-keys-def by simp
  next
    case (Cons b bs)
    then have bs-empty: bs = []
      using buckets-len by (cases bs) simp-all
    obtain p where b-entries: bp-bucket-entries b = [p]
    proof −
      have ∃ p. bp-bucket-entries b = [p] ∧ bp-marker b = snd p
        using singleton Cons by simp
      then show ?thesis
        by (elim exE conjE) (rule that)
    qed
    have key-in:
      fst p ∈ bp-entry-keys (bp-entries P)
      using Cons b-entries
      unfolding bp-entries-def bp-bucket-entries-flat-def bp-entry-keys-def
      by simp
    have deleted:
      bp-delete-keys-from-bucket (bp-entry-keys (bp-entries P)) b =
        b(|bp-bucket-entries := []|)
      unfolding bp-delete-keys-from-bucket-def b-entries
      using key-in by simp
    show ?thesis
      using Cons bs-empty deleted unfolding bp-delete-keys-def by simp
  qed
  have block-deleted:
    bp-block-size (bp-delete-keys (bp-entry-keys (bp-entries P)) P) =
      bmssp-block-size
    using block unfolding bp-delete-keys-def by simp
  show ?thesis
    unfolding bmssp-singleton-bucket-shape-def bmssp-trim-empty-prefix-def
    using block-deleted trimmed-empty by simp
qed

lemma bmssp-singleton-bucket-shape-pull-trim:
  assumes shape: bmssp-singleton-bucket-shape P
    and ord: bp-ordered-invariant P
    and pull: bp-pull bmssp-block-size B P = (S, beta, P1)
  shows bmssp-singleton-bucket-shape (bmssp-trim-empty-prefix P1)
proof (cases bp-can-first-bucket-pull bmssp-block-size P)
  case True
  then obtain b c bs where buckets: bp-buckets P = b # c # bs
    by (rule bp-can-first-bucket-pullE) blast
  have first:
    bp-first-bucket-pull bmssp-block-size B P = (S, beta, P1)
```

```
    using True pull unfolding bp-pull-def by simp
  have P1-def: P1 = bp-delete-keys (bp-bucket-keys b) P
    using first buckets unfolding bp-first-bucket-pull-def
    by (auto simp: Let-def)
  have distinct: distinct (map fst (bp-entries P))
    using ord unfolding bp-ordered-invariant-def bp-invariant-def
      bp-distinct-keys-def by blast
  show ?thesis
    unfolding P1-def
    by (rule bmssp-singleton-bucket-shape-delete-first-bucket
        [OF - shape distinct])
      (use buckets in simp)
next
  case False
  have conservative:
    bp-conservative-pull bmssp-block-size B P = (S, beta, P1)
    using False pull unfolding bp-pull-def by simp
  have S-def: S = bp-pull-set bmssp-block-size P
    and P1-def: P1 = bp-delete-keys S P
    using conservative unfolding bp-conservative-pull-def
    by (auto simp: Let-def)
  show ?thesis
  proof (cases length (bp-entries P) ≤ bmssp-block-size)
    case True
    have S-all: S = bp-entry-keys (bp-entries P)
      using S-def True unfolding bp-pull-set-def by simp
    show ?thesis
      unfolding P1-def S-all
      by (rule bmssp-singleton-bucket-shape-delete-all-small
          [OF shape True])
  next
    case False
    have S-empty: S = {}
      using S-def False unfolding bp-pull-set-def by simp
    have P1-eq: P1 = P
      unfolding P1-def S-empty bp-delete-keys-def
        bp-delete-keys-from-bucket-def by simp
    show ?thesis
      using shape
      unfolding P1-eq
      by (simp add: bmssp-singleton-bucket-shape-trim[OF shape])
  qed
qed

lemma bmssp-singleton-bucket-shape-regularized-insert:
  assumes shape: bmssp-singleton-bucket-shape P
  shows bmssp-singleton-bucket-shape
    (bp-result-of (c-bp-regularized-local-insert x b P))
proof −
```

```
  have block: bp-block-size P = bmssp-block-size
    using shape unfolding bmssp-singleton-bucket-shape-def by simp
  have local-block:
    bp-block-size (bp-local-insert-state x b P) = bmssp-block-size
    using block
    by (simp add: bp-local-insert-state-def bp-delete-key-def Let-def)
  show ?thesis
    unfolding c-bp-regularized-local-insert-result
      bp-regularized-local-insert-def
    by (rule bmssp-rebucket-singleton-shape[OF local-block])
qed

lemma bmssp-insert-updates-singleton-shape:
  assumes shape: bmssp-singleton-bucket-shape P
  shows bmssp-singleton-bucket-shape (bmssp-insert-updates xs P)
  using shape
proof (induction xs arbitrary: P)
  case Nil
  then show ?case
    by simp
next
  case (Cons xb xs)
  obtain x b where xb: xb = (x, b)
    by force
  have step-shape:
    bmssp-singleton-bucket-shape
      (bp-result-of (c-bp-regularized-local-insert x b P))
    by (rule bmssp-singleton-bucket-shape-regularized-insert[OF Cons.prems])
  show ?case
    unfolding xb bmssp-insert-updates.simps
    by (rule Cons.IH[OF step-shape])
qed

lemma bmssp-bucketed-batch-prepend-empty-singleton-shape:
  assumes shape: bmssp-singleton-bucket-shape P
    and empty: bp-entries P = []
  shows bmssp-singleton-bucket-shape
    (bp-result-of (c-bp-paper-batch-prepend xs P))
proof (cases xs)
  case Nil
  then show ?thesis
    using shape empty
    unfolding c-bp-paper-batch-prepend-result
      bp-bucketed-batch-prepend-state-def
    by simp
next
  case (Cons p ps)
  have buckets-empty: bp-buckets P = []
    by (rule bmssp-singleton-bucket-shape-entries-empty-buckets
```

```
      [OF shape empty])
  have block: bp-block-size P = bmssp-block-size
    using shape unfolding bmssp-singleton-bucket-shape-def by simp
  have bucket-shape:
    ∀ b∈set (bp-bucketize-entries (bp-block-size P) xs).
      ∃ p. bp-bucket-entries b = [p] ∧ bp-marker b = snd p
    using block bmssp-bucketize-entries-singleton-shape[of xs] by simp
  show ?thesis
    using block buckets-empty bucket-shape Cons
    unfolding c-bp-paper-batch-prepend-result
      bp-bucketed-batch-prepend-state-def
      bmssp-singleton-bucket-shape-def
    by (simp add: Let-def)
qed

lemma bmssp-apply-updates-singleton-shape:
  assumes shape: bmssp-singleton-bucket-shape (bmssp-trim-empty-prefix P)
  shows bmssp-singleton-bucket-shape (bmssp-apply-updates xs P)
proof −
  let ?P0 = bmssp-trim-empty-prefix P
  show ?thesis
  proof (cases bp-entries ?P0 = [])
    case True
    have result:
      bmssp-apply-updates xs P =
        bp-result-of (c-bp-paper-batch-prepend xs ?P0)
      using True unfolding bmssp-apply-updates-def by simp
    have shape′:
      bmssp-singleton-bucket-shape
        (bp-result-of (c-bp-paper-batch-prepend xs ?P0))
      by (rule bmssp-bucketed-batch-prepend-empty-singleton-shape
          [OF shape True])
    show ?thesis
      using result shape′ by simp
  next
    case False
    have result:
      bmssp-apply-updates xs P = bmssp-insert-updates xs ?P0
      using False unfolding bmssp-apply-updates-def by simp
    have shape′:
      bmssp-singleton-bucket-shape (bmssp-insert-updates xs ?P0)
      by (rule bmssp-insert-updates-singleton-shape[OF shape])
    show ?thesis
      using result shape′ by simp
  qed
qed

lemma bmssp-partition-key-inject:
  assumes eq: bmssp-partition-key v d = bmssp-partition-key w e
```

```
  shows v = w ∧ d = e
proof −
  have d-eq: d = e
    using eq bmssp-partition-key-floor[of v d]
      bmssp-partition-key-floor[of w e]
    by simp
  have frac-eq:
    real v / real (Suc v) = real w / real (Suc w)
    using eq d-eq unfolding bmssp-partition-key-def by simp
  have frac-v:
    real v / real (Suc v) = 1 − inverse (real (Suc v))
    by (simp add: field-simps)
  have frac-w:
    real w / real (Suc w) = 1 − inverse (real (Suc w))
    by (simp add: field-simps)
  have inv-eq:
    inverse (real (Suc v)) = inverse (real (Suc w))
    using frac-eq frac-v frac-w by linarith
  then have real (Suc v) = real (Suc w)
    by simp
  then have v = w
    by simp
  then show ?thesis
    using d-eq by simp
qed

lemma distinct-map-fst-mem-eq:
  assumes distinct: distinct (map fst xs)
    and p: p ∈ set xs
    and q: q ∈ set xs
    and fst-eq: fst p = fst q
  shows p = q
  using assms
proof (induction xs)
  case Nil
  then show ?case
    by simp
next
  case (Cons r xs)
  show ?case
  proof (cases p = r)
    case True
    have q = r
    proof (rule ccontr)
      assume q ≠ r
      then have q ∈ set xs
        using Cons.prems(3) by simp
      then have fst q ∈ set (map fst xs)
        by force
```

```
      moreover have fst r ∉ set (map fst xs)
        using Cons.prems(1) by simp
      ultimately show False
        using Cons.prems(4) True by simp
    qed
    then show ?thesis
      using True by simp
  next
    case False
    then have p-tail: p ∈ set xs
      using Cons.prems(2) by simp
    show ?thesis
    proof (cases q = r)
      case True
      have fst p ∈ set (map fst xs)
        using p-tail by force
      moreover have fst r ∉ set (map fst xs)
        using Cons.prems(1) by simp
      ultimately show ?thesis
        using Cons.prems(4) True by simp
    next
      case False
      have q-tail: q ∈ set xs
        using False Cons.prems(3) by simp
      have distinct-tail: distinct (map fst xs)
        using Cons.prems(1) by simp
      show ?thesis
        by (rule Cons.IH[OF distinct-tail p-tail q-tail Cons.prems(4)])
    qed
  qed
qed

lemma bmssp-partition-entries-value-inj:
  assumes match: bmssp-partition-state-match vertices settled ds P
    and ord: bp-ordered-invariant P
  shows inj-on snd (set (bp-entries P))
proof (rule inj-onI)
  fix p q
  assume p: p ∈ set (bp-entries P)
    and q: q ∈ set (bp-entries P)
    and snd-eq: snd p = snd q
  have distinct-ds: distinct (map fst ds)
    and keys: bmssp-partition-keys-match settled ds P
    and vals: bmssp-partition-values-match settled ds (bp-view P)
    using match unfolding bmssp-partition-state-match-def by auto
  have values-consistent:
    ⋀r. r ∈ set (bp-entries P) ⟹
      value-of (bp-view P) (fst r) = snd r
    using ord
```

unfolding *bp-ordered-invariant-def bp-invariant-def*
bp-values-consistent-def bp-view-def
by *simp*
have *p-key*: *fst p* ∈ *keys-of* (*bp-view P*)
using *p* **unfolding** *bp-view-def bp-entry-keys-def* **by** *auto*
have *q-key*: *fst q* ∈ *keys-of* (*bp-view P*)
using *q* **unfolding** *bp-view-def bp-entry-keys-def* **by** *auto*
obtain *dp* **where** *p-lookup*: *bmssp-lookup-dist ds* (*fst p*) = *Some dp*
and *p-unsettled*: *fst p* ∉ *set settled*
proof −
have *key*: *fst p* ∈ *set* (*map fst ds*) − *set settled*
using *keys p-key* **unfolding** *bmssp-partition-keys-match-def* **by** *simp*
then obtain *dp* **where** *mem*: (*fst p*, *dp*) ∈ *set ds*
by *force*
have *lookup*: *bmssp-lookup-dist ds* (*fst p*) = *Some dp*
by (*rule bmssp-lookup-dist-Some-if-distinct-mem*
[*OF distinct-ds mem*])
show *?thesis*
using *that lookup key* **by** *blast*
qed
obtain *dq* **where** *q-lookup*: *bmssp-lookup-dist ds* (*fst q*) = *Some dq*
and *q-unsettled*: *fst q* ∉ *set settled*
proof −
have *key*: *fst q* ∈ *set* (*map fst ds*) − *set settled*
using *keys q-key* **unfolding** *bmssp-partition-keys-match-def* **by** *simp*
then obtain *dq* **where** *mem*: (*fst q*, *dq*) ∈ *set ds*
by *force*
have *lookup*: *bmssp-lookup-dist ds* (*fst q*) = *Some dq*
by (*rule bmssp-lookup-dist-Some-if-distinct-mem*
[*OF distinct-ds mem*])
show *?thesis*
using *that lookup key* **by** *blast*
qed
have *p-value*:
snd p = *bmssp-partition-key* (*fst p*) *dp*
using *vals p-lookup p-unsettled values-consistent*[*OF p*]
unfolding *bmssp-partition-values-match-def* **by** *simp*
have *q-value*:
snd q = *bmssp-partition-key* (*fst q*) *dq*
using *vals q-lookup q-unsettled values-consistent*[*OF q*]
unfolding *bmssp-partition-values-match-def* **by** *simp*
have *fst-eq*: *fst p* = *fst q*
using *bmssp-partition-key-inject*[*of fst p dp fst q dq*]
snd-eq p-value q-value **by** *auto*
have *distinct-entries*: *distinct* (*map fst* (*bp-entries P*))
using *ord* **unfolding** *bp-ordered-invariant-def bp-invariant-def*
bp-distinct-keys-def **by** *blast*
show *p* = *q*
by (*rule distinct-map-fst-mem-eq*

```
    [OF distinct-entries p q fst-eq])
qed

lemma bmssp-can-first-bucket-pull-if-singleton-shape:
  assumes shape: bmssp-singleton-bucket-shape P
    and ord: bp-ordered-invariant P
    and values-inj: inj-on snd (set (bp-entries P))
    and many: length (bp-entries P) > bmssp-block-size
  shows bp-can-first-bucket-pull bmssp-block-size P
proof −
  have singleton:
    ∀ b∈set (bp-buckets P).
      ∃ p. bp-bucket-entries b = [p] ∧ bp-marker b = snd p
    using shape unfolding bmssp-singleton-bucket-shape-def by simp
  have distinct-entries: distinct (map fst (bp-entries P))
    using ord unfolding bp-ordered-invariant-def bp-invariant-def
      bp-distinct-keys-def by blast
  obtain b rest where buckets-outer: bp-buckets P = b # rest
  proof (cases bp-buckets P)
    case Nil
    then have bp-entries P = []
      unfolding bp-entries-def bp-bucket-entries-flat-def by simp
    then show ?thesis
      using many unfolding bmssp-block-size-def by simp
  next
    case (Cons b rest)
    then show ?thesis
      by (rule that)
  qed
  obtain p where b-entries: bp-bucket-entries b = [p]
    and b-marker: bp-marker b = snd p
    using singleton buckets-outer by auto
  obtain c bs where rest: rest = c # bs
  proof (cases rest)
    case Nil
    then have bp-entries P = [p]
      using buckets-outer b-entries
      unfolding bp-entries-def bp-bucket-entries-flat-def by simp
    then show ?thesis
      using many unfolding bmssp-block-size-def by simp
  next
    case (Cons c bs)
    then show ?thesis
      by (rule that)
  qed
  have buckets: bp-buckets P = b # c # bs
    using buckets-outer rest by simp
  obtain q where c-entries: bp-bucket-entries c = [q]
    and c-marker: bp-marker c = snd q
```

```
    using singleton buckets by auto
  have p-entry: p ∈ set (bp-entries P)
    using buckets b-entries
    unfolding bp-entries-def bp-bucket-entries-flat-def by simp
  have q-entry: q ∈ set (bp-entries P)
    using buckets c-entries
    unfolding bp-entries-def bp-bucket-entries-flat-def by simp
  have p-neq-q: p ≠ q
  proof
    assume p = q
    then have fst p = fst q
      by simp
    moreover have map fst (bp-entries P) =
      fst p # fst q # map fst (bp-bucket-entries-flat bs)
      using buckets b-entries c-entries
      unfolding bp-entries-def bp-bucket-entries-flat-def by simp
    ultimately show False
      using distinct-entries by simp
  qed
  have snd-ne: snd p ≠ snd q
    using values-inj p-entry q-entry p-neq-q unfolding inj-on-def by blast
  have boundary:
    ∀ r∈set (bp-bucket-entries b). snd r ≤ bp-marker c
    using ord buckets
    unfolding bp-ordered-invariant-def
      bp-bucket-boundaries-state-ok-def
    by simp
  have below: bp-bucket-below-bound b (bp-marker c)
  proof −
    have snd p < snd q
      using boundary snd-ne unfolding b-entries c-marker by auto
    then show ?thesis
      unfolding bp-bucket-below-bound-def b-entries c-marker by simp
  qed
  have len-b: length (bp-bucket-entries b) ≤ bmssp-block-size
    unfolding b-entries bmssp-block-size-def by simp
  have tail-nonempty: bp-bucket-entries-flat (c # bs) ≠ []
    using c-entries unfolding bp-bucket-entries-flat-def by simp
  show ?thesis
    unfolding bp-can-first-bucket-pull-def buckets
    using many len-b below tail-nonempty by simp
qed

lemma bmssp-pull-nonempty-if-keys-nonempty:
  assumes shape: bmssp-singleton-bucket-shape P
    and ord: bp-ordered-invariant P
    and values-inj: inj-on snd (set (bp-entries P))
    and pull: bp-pull bmssp-block-size B P = (S, beta, P1)
    and keys-nonempty: keys-of (bp-view P) ≠ {}
```

```
  shows S ≠ {}
proof (cases bp-can-first-bucket-pull bmssp-block-size P)
  case True
  obtain b c bs where buckets: bp-buckets P = b # c # bs
    by (rule bp-can-first-bucket-pullE[OF True])
  obtain p where b-entries: bp-bucket-entries b = [p]
    using shape buckets unfolding bmssp-singleton-bucket-shape-def by auto
  have first:
    bp-first-bucket-pull bmssp-block-size B P = (S, beta, P1)
    using True pull unfolding bp-pull-def by simp
  have S = bp-bucket-keys b
    using first buckets unfolding bp-first-bucket-pull-def
    by (simp add: Let-def)
  then show ?thesis
    using b-entries unfolding bp-bucket-keys-def bp-entry-keys-def by simp
next
  case False
  note not-can = False
  have conservative:
    bp-conservative-pull bmssp-block-size B P = (S, beta, P1)
    using False pull unfolding bp-pull-def by simp
  have S-def: S = bp-pull-set bmssp-block-size P
    using conservative unfolding bp-conservative-pull-def
    by (simp add: Let-def)
  show ?thesis
  proof (cases length (bp-entries P) ≤ bmssp-block-size)
    case True
    have bp-entry-keys (bp-entries P) ≠ {}
      using keys-nonempty unfolding bp-view-def by simp
    then show ?thesis
      using S-def True unfolding bp-pull-set-def by simp
  next
    case False
    have can: bp-can-first-bucket-pull bmssp-block-size P
      by (rule bmssp-can-first-bucket-pull-if-singleton-shape
          [OF shape ord values-inj])
        (use False in simp)
    then show ?thesis
      using not-can by simp
  qed
qed

definition bmssp-dijkstra-loop-state ::
  nat list ⇒ nat list ⇒
    nat-dist ⇒ nat bucketed-partition ⇒ bool where
  bmssp-dijkstra-loop-state vertices settled ds P ⟷
     bmssp-dijkstra-state G src vertices settled ds P ∧
     bp-ordered-invariant P ∧
     partition-upper-bound (bp-view P) bmssp-bound
```

lemma *bmssp-dijkstra-loop-state-initial*:
assumes *src-vertices*: *src* ∈ *set vertices*
shows *bmssp-dijkstra-loop-state vertices* [] [(*src*, *0*)]
(*bp-result-of*
(*c-bp-regularized-local-insert src* (*bmssp-partition-key src 0*)
(*bp-empty bmssp-block-size bmssp-bound*)))
proof −
have *dijkstra*:
bmssp-dijkstra-state G src vertices [] [(*src*, *0*)]
(*bp-result-of*
(*c-bp-regularized-local-insert src* (*bmssp-partition-key src 0*)
(*bp-empty bmssp-block-size bmssp-bound*)))
by (*rule bmssp-dijkstra-state-initial*[*OF src-vertices*])
have *ord*:
bp-ordered-invariant
(*bp-result-of*
(*c-bp-regularized-local-insert src* (*bmssp-partition-key src 0*)
(*bp-empty bmssp-block-size bmssp-bound*)))
by (*rule bmssp-initial-partition-bridge*(*1*))
have *upper*:
partition-upper-bound
(*bp-view*
(*bp-result-of*
(*c-bp-regularized-local-insert src* (*bmssp-partition-key src 0*)
(*bp-empty bmssp-block-size bmssp-bound*))))
bmssp-bound
by (*rule bmssp-initial-partition-bridge*(*2*))
show *?thesis*
unfolding *bmssp-dijkstra-loop-state-def*
using *dijkstra ord upper* **by** *simp*
qed

lemma *bmssp-dijkstra-loop-state-step-bridge*:
fixes *old-settled settled vertices* :: *nat list*
and *pulled* :: *nat list*
and *updates* :: (*nat* × *real*) *list*
assumes *loop-state*:
bmssp-dijkstra-loop-state vertices old-settled ds P
and *edge-targets*:
⋀*a b w*. (*a*, *b*, *w*) ∈ *set G* ⟹ *b* ∈ *set vertices*
and *pulled-def*:
pulled =
filter (λ*x*. *x* ∈ *S* ∧ *x* ∉ *set old-settled*)
vertices
and *settled-def*: *settled* = *pulled* @ *old-settled*
and *relaxed*: *bmssp-relax-vertices G settled pulled ds* = (*updates*, *ds*′)
and *P2-def*: *P2* = *bmssp-apply-updates updates P1*

```
    and wf: nat-graph-well-formed G
    and distinct-vertices: distinct vertices
    and pull: bp-pull bmssp-block-size bmssp-bound P = (S, beta, P1)
  shows bmssp-dijkstra-loop-state vertices settled ds' P2
proof −
  have state: bmssp-dijkstra-state G src vertices old-settled ds P
    and ord: bp-ordered-invariant P
    and upper: partition-upper-bound (bp-view P) bmssp-bound
    using loop-state unfolding bmssp-dijkstra-loop-state-def by auto
  have state':
    bmssp-dijkstra-state G src vertices settled ds' P2
    by (rule bmssp-dijkstra-state-step-bridge
        [OF state edge-targets pulled-def settled-def relaxed P2-def
          distinct-vertices ord upper pull])
  have updates-def: updates = fst (bmssp-relax-vertices G settled pulled ds)
    using relaxed by simp
  have ord':
    bp-ordered-invariant P2
    by (rule bmssp-loop-partition-step-bridge(8)
        [OF pulled-def updates-def P2-def wf distinct-vertices ord upper
          pull])
  have upper':
    partition-upper-bound (bp-view P2) bmssp-bound
    by (rule bmssp-dijkstra-state-step-upper-bound
        [OF state wf pulled-def settled-def relaxed P2-def distinct-vertices
          ord upper pull])
  show ?thesis
    unfolding bmssp-dijkstra-loop-state-def
    using state' ord' upper' by simp
qed

lemma distinct-subset-length-le:
  assumes xs: distinct xs
    and ys: distinct ys
    and subset: set xs ⊆ set ys
  shows length xs ≤ length ys
proof −
  have length xs = card (set xs)
    using xs by (simp add: distinct-card)
  also have ... ≤ card (set ys)
    by (rule card-mono) (use subset in simp-all)
  also have ... = length ys
    using ys by (simp add: distinct-card)
  finally show ?thesis .
qed

lemma bmssp-dijkstra-loop-state-settled-subset-vertices:
  assumes loop-state: bmssp-dijkstra-loop-state vertices settled ds P
  shows set settled ⊆ set vertices
```

```
proof
  fix v
  assume v-settled: v ∈ set settled
  have state: bmssp-dijkstra-state G src vertices settled ds P
    using loop-state unfolding bmssp-dijkstra-loop-state-def by simp
  have match: bmssp-partition-state-match vertices settled ds P
    and have-lookups: bmssp-settled-have-lookups settled ds
    using state unfolding bmssp-dijkstra-state-def by auto
  obtain d where lookup: bmssp-lookup-dist ds v = Some d
    using have-lookups v-settled
    unfolding bmssp-settled-have-lookups-def by blast
  have v ∈ set (map fst ds)
    by (rule bmssp-lookup-dist-mem-key[OF lookup])
  then show v ∈ set vertices
    using match unfolding bmssp-partition-state-match-def by blast
qed

lemma bmssp-loop-quiescent-state:
  assumes loop-state: bmssp-dijkstra-loop-state vertices settled ds P
    and shape: bmssp-singleton-bucket-shape P
    and edge-targets:
      ⋀a b w. (a, b, w) ∈ set G ⟹ b ∈ set vertices
    and distinct-vertices: distinct vertices
    and distinct-settled: distinct settled
    and fuel: fuel > length vertices − length settled
  shows ∃ settled' ds' P' S beta P1.
    bmssp-loop fuel G vertices settled ds P = bmssp-normalize-dist ds' ∧
    bmssp-dijkstra-loop-state vertices settled' ds' P' ∧
    bmssp-singleton-bucket-shape P' ∧
    distinct settled' ∧
    set settled ⊆ set settled' ∧
    bp-pull bmssp-block-size bmssp-bound P' = (S, beta, P1) ∧
    filter (λx. x ∈ S ∧ x ∉ set settled') vertices = []
  using loop-state shape distinct-settled fuel
proof (induction fuel arbitrary: settled ds P)
  case 0
  then show ?case
    by simp
next
  case (Suc fuel)
  obtain S beta P1 where pull:
    bp-pull bmssp-block-size bmssp-bound P = (S, beta, P1)
    by (cases bp-pull bmssp-block-size bmssp-bound P) auto
  let ?pulled =
    filter (λx. x ∈ S ∧ x ∉ set settled) vertices
  show ?case
  proof (cases ?pulled = [])
    case True
    have loop-eq:
```

```
    bmssp-loop (Suc fuel) G vertices settled ds P =
      bmssp-normalize-dist ds
    using pull True by simp
  show ?thesis
  proof (intro exI conjI)
    show bmssp-loop (Suc fuel) G vertices settled ds P =
        bmssp-normalize-dist ds
      by (rule loop-eq)
    show bmssp-dijkstra-loop-state vertices settled ds P
      by (rule Suc.prems(1))
    show bmssp-singleton-bucket-shape P
      by (rule Suc.prems(2))
    show distinct settled
      by (rule Suc.prems(3))
    show set settled ⊆ set settled
      by simp
    show bp-pull bmssp-block-size bmssp-bound P = (S, beta, P1)
      by (rule pull)
    show filter (λx. x ∈ S ∧ x ∉ set settled) vertices = []
      by (rule True)
  qed
next
  case False
  let ?settled = ?pulled @ settled
  obtain updates ds' where relaxed:
    bmssp-relax-vertices G ?settled ?pulled ds = (updates, ds')
    by force
  let ?P2 = bmssp-apply-updates updates P1
  have pulled-def:
    ?pulled =
      filter (λx. x ∈ S ∧ x ∉ set settled) vertices
    by simp
  have settled-def: ?settled = ?pulled @ settled
    by simp
  have P2-def: ?P2 = bmssp-apply-updates updates P1
    by simp
  have loop-state':
    bmssp-dijkstra-loop-state vertices ?settled ds' ?P2
    by (rule bmssp-dijkstra-loop-state-step-bridge
        [OF Suc.prems(1) edge-targets pulled-def settled-def relaxed
          P2-def wf distinct-vertices pull])
  have state: bmssp-dijkstra-state G src vertices settled ds P
    and ord: bp-ordered-invariant P
    and upper: partition-upper-bound (bp-view P) bmssp-bound
    using Suc.prems(1) unfolding bmssp-dijkstra-loop-state-def by auto
  have trim-shape:
    bmssp-singleton-bucket-shape (bmssp-trim-empty-prefix P1)
    by (rule bmssp-singleton-bucket-shape-pull-trim
        [OF Suc.prems(2) ord pull])
```

have *shape'*:
bmssp-singleton-bucket-shape ?P2
by (*rule bmssp-apply-updates-singleton-shape*[*OF trim-shape*])
have *updates-def*:
updates = *fst* (*bmssp-relax-vertices G ?settled ?pulled ds*)
using *relaxed* **by** *simp*
have *pulled-len*: *length ?pulled* ≤ *1*
by (*rule bmssp-loop-partition-step-bridge*(*4*)
[*OF pulled-def updates-def P2-def wf distinct-vertices ord upper pull*])
have *pulled-one*: *length ?pulled* = *1*
using *False pulled-len* **by** (*cases ?pulled*) *auto*
have *pulled-distinct*: *distinct ?pulled*
using *pulled-len* **by** (*cases ?pulled*) *auto*
have *pulled-disjoint*: *set ?pulled* ∩ *set settled* = {}
by *auto*
have *distinct-settled'*: *distinct ?settled*
using *pulled-distinct pulled-disjoint Suc.prems*(*3*) **by** *simp*
have *settled-subset-vertices*:
set settled ⊆ *set vertices*
by (*rule bmssp-dijkstra-loop-state-settled-subset-vertices*
[*OF Suc.prems*(*1*)])
have *old-lt-vertices*: *length settled* < *length vertices*
proof −
obtain *x* **where** *x-pulled*: *x* ∈ *set ?pulled*
using *False* **by** (*cases ?pulled*) *auto*
have *x-vertices*: *x* ∈ *set vertices*
using *x-pulled* **by** *simp*
have *x-not-settled*: *x* ∉ *set settled*
using *x-pulled* **by** *simp*
have *proper*: *set settled* < *set vertices*
using *settled-subset-vertices x-vertices x-not-settled*
unfolding *psubset-eq* **by** *blast*
have *card* (*set settled*) < *card* (*set vertices*)
by (*rule psubset-card-mono*) (*use proper* **in** *simp-all*)
then show *?thesis*
using *Suc.prems*(*3*) *distinct-vertices*
by (*simp add*: *distinct-card*)
qed
have *diff-step*:
Suc (*length vertices* − *Suc* (*length settled*)) =
length vertices − *length settled*
by (*rule Suc-diff-Suc*[*OF old-lt-vertices*])
have *fuel'*:
fuel > *length vertices* − *length ?settled*
proof −
have *fuel* > *length vertices* − *Suc* (*length settled*)
using *Suc.prems*(*4*) *diff-step* **by** *simp*
then show *?thesis*

```
    using pulled-one by simp
  qed
  obtain settled'' ds'' P'' S' beta' P1' where final:
    bmssp-loop fuel G vertices ?settled ds' ?P2 =
      bmssp-normalize-dist ds''
    bmssp-dijkstra-loop-state vertices settled'' ds'' P''
    bmssp-singleton-bucket-shape P''
    distinct settled''
    set ?settled ⊆ set settled''
    bp-pull bmssp-block-size bmssp-bound P'' = (S', beta', P1')
    filter (λx. x ∈ S' ∧ x ∉ set settled'') vertices = []
  proof −
    have ∃ settled-out ds-out P-out S-out beta-out P1-out.
      bmssp-loop fuel G vertices ?settled ds' ?P2 =
        bmssp-normalize-dist ds-out ∧
      bmssp-dijkstra-loop-state vertices settled-out ds-out P-out ∧
      bmssp-singleton-bucket-shape P-out ∧
      distinct settled-out ∧
      set ?settled ⊆ set settled-out ∧
      bp-pull bmssp-block-size bmssp-bound P-out =
        (S-out, beta-out, P1-out) ∧
      filter (λx. x ∈ S-out ∧ x ∉ set settled-out) vertices = []
      by (rule Suc.IH[OF loop-state' shape' distinct-settled' fuel'])
    then show ?thesis
      by (elim exE conjE) (rule that)
  qed
  have loop-eq:
    bmssp-loop (Suc fuel) G vertices settled ds P =
      bmssp-loop fuel G vertices ?settled ds' ?P2
    using pull False relaxed by (simp add: Let-def)
  show ?thesis
  proof (intro exI conjI)
    show bmssp-loop (Suc fuel) G vertices settled ds P =
        bmssp-normalize-dist ds''
      using loop-eq final(1) by simp
    show bmssp-dijkstra-loop-state vertices settled'' ds'' P''
      by (rule final(2))
    show bmssp-singleton-bucket-shape P''
      by (rule final(3))
    show distinct settled''
      by (rule final(4))
    show set settled ⊆ set settled''
      using final(5) by auto
    show bp-pull bmssp-block-size bmssp-bound P'' = (S', beta', P1')
      by (rule final(6))
    show filter (λx. x ∈ S' ∧ x ∉ set settled'') vertices = []
      by (rule final(7))
  qed
qed
```

qed

lemma *bmssp-dijkstra-loop-state-lookup-exact-if-queue-empty*:
assumes *loop-state*: *bmssp-dijkstra-loop-state vertices settled ds P*
and *empty*: *keys-of* (*bp-view P*) = {}
and *lookup*: *bmssp-lookup-dist ds v* = *Some d*
shows *real d* = *nat-graph-dist G src v*
proof −
have *state*: *bmssp-dijkstra-state G src vertices settled ds P*
using *loop-state* **unfolding** *bmssp-dijkstra-loop-state-def* **by** *simp*
have *match*: *bmssp-partition-state-match vertices settled ds P*
and *exact*: *bmssp-settled-exact G src settled ds*
using *state* **unfolding** *bmssp-dijkstra-state-def* **by** *auto*
have *keys*: *bmssp-partition-keys-match settled ds P*
using *match* **unfolding** *bmssp-partition-state-match-def* **by** *blast*
have *v-key*: *v* ∈ *set* (*map fst ds*)
by (*rule bmssp-lookup-dist-mem-key*[*OF lookup*])
have *v-settled*: *v* ∈ *set settled*
using *keys empty v-key* **unfolding** *bmssp-partition-keys-match-def* **by** *blast*
show *?thesis*
using *exact v-settled lookup* **unfolding** *bmssp-settled-exact-def* **by** *blast*
qed

lemma *bmssp-dijkstra-loop-state-normalize-exact-if-queue-empty*:
assumes *loop-state*: *bmssp-dijkstra-loop-state vertices settled ds P*
and *empty*: *keys-of* (*bp-view P*) = {}
and *mem*: (*v*, *d*) ∈ *set* (*bmssp-normalize-dist ds*)
shows *real d* = *nat-graph-dist G src v*
proof −
have *state*: *bmssp-dijkstra-state G src vertices settled ds P*
using *loop-state* **unfolding** *bmssp-dijkstra-loop-state-def* **by** *simp*
have *match*: *bmssp-partition-state-match vertices settled ds P*
using *state* **unfolding** *bmssp-dijkstra-state-def* **by** *auto*
have *distinct-ds*: *distinct* (*map fst ds*)
using *match* **unfolding** *bmssp-partition-state-match-def* **by** *simp*
have *mem-ds*: (*v*, *d*) ∈ *set ds*
using *mem* **unfolding** *bmssp-normalize-dist-def* **by** *simp*
have *lookup*: *bmssp-lookup-dist ds v* = *Some d*
by (*rule bmssp-lookup-dist-Some-if-distinct-mem*[*OF distinct-ds mem-ds*])
show *?thesis*
by (*rule bmssp-dijkstra-loop-state-lookup-exact-if-queue-empty*
[*OF loop-state empty lookup*])
qed

lemma *bmssp-dijkstra-loop-state-normalize-reachable*:
assumes *loop-state*: *bmssp-dijkstra-loop-state vertices settled ds P*
and *mem*: (*v*, *d*) ∈ *set* (*bmssp-normalize-dist ds*)
shows *nat-graph-reachable G src v*
proof −

have *state*: *bmssp-dijkstra-state G src vertices settled ds P*
using *loop-state* **unfolding** *bmssp-dijkstra-loop-state-def* **by** *simp*
have *match*: *bmssp-partition-state-match vertices settled ds P*
and *witnesses*: *bmssp-label-witnesses G src settled ds*
using *state* **unfolding** *bmssp-dijkstra-state-def* **by** *auto*
have *distinct-ds*: *distinct* (*map fst ds*)
using *match* **unfolding** *bmssp-partition-state-match-def* **by** *simp*
have *mem-ds*: (*v*, *d*) ∈ *set ds*
using *mem* **unfolding** *bmssp-normalize-dist-def* **by** *simp*
have *lookup*: *bmssp-lookup-dist ds v* = *Some d*
by (*rule bmssp-lookup-dist-Some-if-distinct-mem*[*OF distinct-ds mem-ds*])
show *?thesis*
by (*rule bmssp-label-witnesses-lookup-reachable*[*OF witnesses lookup*])
qed

lemma *bmssp-dijkstra-loop-state-reachable-lookup-if-queue-empty*:
assumes *loop-state*: *bmssp-dijkstra-loop-state vertices settled ds P*
and *empty*: *keys-of* (*bp-view P*) = {}
and *reach*: *nat-graph-reachable G src v*
shows ∃ *d*. *bmssp-lookup-dist ds v* = *Some d*
proof (*rule ccontr*)
assume *no-lookup*: ¬ (∃ *d*. *bmssp-lookup-dist ds v* = *Some d*)
have *v-unsettled*: *v* ∉ *set settled*
proof
assume *v-settled*: *v* ∈ *set settled*
have *state*: *bmssp-dijkstra-state G src vertices settled ds P*
using *loop-state* **unfolding** *bmssp-dijkstra-loop-state-def* **by** *simp*
have *settled-lookup*: *bmssp-settled-have-lookups settled ds*
using *state* **unfolding** *bmssp-dijkstra-state-def* **by** *simp*
then obtain *d* **where** *bmssp-lookup-dist ds v* = *Some d*
using *v-settled* **unfolding** *bmssp-settled-have-lookups-def* **by** *blast*
then show *False*
using *no-lookup* **by** *blast*
qed
have *state*: *bmssp-dijkstra-state G src vertices settled ds P*
using *loop-state* **unfolding** *bmssp-dijkstra-loop-state-def* **by** *simp*
have *match*: *bmssp-partition-state-match vertices settled ds P*
and *source-zero*: *bmssp-source-zero src ds*
and *settled-lookup*: *bmssp-settled-have-lookups settled ds*
and *exact*: *bmssp-settled-exact G src settled ds*
and *frontier*: *bmssp-frontier-relaxed G src settled ds*
using *state* **unfolding** *bmssp-dijkstra-state-def* **by** *auto*
have *reach′*: *concrete.reachable src v*
using *reach* **unfolding** *nat-graph-reachable-def* **.**
obtain *p* **where** *sp*: *concrete.shortest-walk src p v*
using *concrete.shortest-walk-exists*[*OF reach′*] **by** *blast*
obtain *y dy* **where** *y-unsettled*: *y* ∉ *set settled*
and *lookup-y*: *bmssp-lookup-dist ds y* = *Some dy*
by (*rule bmssp-shortest-walk-first-unsettled-lookup-le-dist*

```
      [OF sp v-unsettled source-zero settled-lookup exact frontier])
    blast
  have keys: bmssp-partition-keys-match settled ds P
    using match unfolding bmssp-partition-state-match-def by blast
  have y-key: y ∈ set (map fst ds)
    by (rule bmssp-lookup-dist-mem-key[OF lookup-y])
  have y ∈ keys-of (bp-view P)
    using keys y-key y-unsettled
    unfolding bmssp-partition-keys-match-def by blast
  then show False
    using empty by simp
qed

lemma bmssp-dijkstra-loop-state-normalize-complete-if-queue-empty:
  assumes loop-state: bmssp-dijkstra-loop-state vertices settled ds P
    and empty: keys-of (bp-view P) = {}
    and reach: nat-graph-reachable G src v
  shows ∃ d. (v, d) ∈ set (bmssp-normalize-dist ds)
proof −
  obtain d where lookup: bmssp-lookup-dist ds v = Some d
    using bmssp-dijkstra-loop-state-reachable-lookup-if-queue-empty
      [OF loop-state empty reach]
    by blast
  have mem-ds: (v, d) ∈ set ds
    by (rule bmssp-lookup-dist-Some-mem[OF lookup])
  have (v, d) ∈ set (bmssp-normalize-dist ds)
    using mem-ds unfolding bmssp-normalize-dist-def by simp
  then show ?thesis
    by blast
qed

lemma bmssp-dijkstra-loop-state-quiescent-pull-empty:
  assumes loop-state: bmssp-dijkstra-loop-state vertices settled ds P
    and pull: bp-pull bmssp-block-size bmssp-bound P = (S, beta, P1)
    and stopped:
      filter (λx. x ∈ S ∧ x ∉ set settled) vertices = []
  shows S = {}
proof
  show S ⊆ {}
  proof
    fix x
    assume xS: x ∈ S
    have state: bmssp-dijkstra-state G src vertices settled ds P
      and ord: bp-ordered-invariant P
      and upper: partition-upper-bound (bp-view P) bmssp-bound
      using loop-state unfolding bmssp-dijkstra-loop-state-def by auto
    have match: bmssp-partition-state-match vertices settled ds P
      using state unfolding bmssp-dijkstra-state-def by simp
    have sep:
```

```
      pull-separates (bp-view P) bmssp-block-size bmssp-bound S beta
        (bp-view P1)
      by (rule bmssp-pull-refines-pull-separates[OF ord upper pull])
    have S-keys: S ⊆ keys-of (bp-view P)
      by (rule pull-separates-subset[OF sep])
    have keys-vertices: keys-of (bp-view P) ⊆ set vertices
      by (rule bmssp-partition-state-match-keys-subset[OF match])
    have keys: bmssp-partition-keys-match settled ds P
      using match unfolding bmssp-partition-state-match-def by blast
    have x-vertices: x ∈ set vertices
      using S-keys keys-vertices xS by blast
    have x-unsettled: x ∉ set settled
      using keys S-keys xS unfolding bmssp-partition-keys-match-def by blast
    have x ∈ set
        (filter (λx. x ∈ S ∧ x ∉ set settled) vertices)
      using xS x-vertices x-unsettled by simp
    then show x ∈ {}
      using stopped by simp
  qed
next
  show {} ⊆ S
    by simp
qed

lemma bmssp-dijkstra-loop-state-keys-empty-if-pull-empty:
  assumes loop-state: bmssp-dijkstra-loop-state vertices settled ds P
    and shape: bmssp-singleton-bucket-shape P
    and pull: bp-pull bmssp-block-size bmssp-bound P = (S, beta, P1)
    and S-empty: S = {}
  shows keys-of (bp-view P) = {}
proof (rule ccontr)
  assume nonempty: keys-of (bp-view P) ≠ {}
  have state: bmssp-dijkstra-state G src vertices settled ds P
    and ord: bp-ordered-invariant P
    using loop-state unfolding bmssp-dijkstra-loop-state-def by auto
  have match: bmssp-partition-state-match vertices settled ds P
    using state unfolding bmssp-dijkstra-state-def by simp
  have values-inj: inj-on snd (set (bp-entries P))
    by (rule bmssp-partition-entries-value-inj[OF match ord])
  have S ≠ {}
    by (rule bmssp-pull-nonempty-if-keys-nonempty
        [OF shape ord values-inj pull nonempty])
  then show False
    using S-empty by simp
qed

lemma bmssp-dijkstra-loop-state-quiescent-queue-empty:
  assumes loop-state: bmssp-dijkstra-loop-state vertices settled ds P
    and shape: bmssp-singleton-bucket-shape P
```

```
    and pull: bp-pull bmssp-block-size bmssp-bound P = (S, beta, P1)
    and stopped:
      filter (λx. x ∈ S ∧ x ∉ set settled) vertices = []
  shows keys-of (bp-view P) = {}
proof −
  have S-empty: S = {}
    by (rule bmssp-dijkstra-loop-state-quiescent-pull-empty
        [OF loop-state pull stopped])
  show ?thesis
    by (rule bmssp-dijkstra-loop-state-keys-empty-if-pull-empty
        [OF loop-state shape pull S-empty])
qed

definition bmssp-executable-distances-fuel :: nat where
  bmssp-executable-distances-fuel =
    Suc (length (bmssp-vertices G src) * Suc (length G))

definition bmssp-executable-initial-partition :: nat bucketed-partition where
  bmssp-executable-initial-partition =
    bp-result-of
      (c-bp-regularized-local-insert src (bmssp-partition-key src 0)
        (bp-empty bmssp-block-size bmssp-bound))

lemma bmssp-distances-unfold-executable:
  bmssp-distances G src =
    bmssp-loop bmssp-executable-distances-fuel G (bmssp-vertices G src) []
      [(src, 0)] bmssp-executable-initial-partition
  unfolding bmssp-distances-def bmssp-executable-distances-fuel-def
    bmssp-executable-initial-partition-def
  by (simp add: Let-def)

lemma bmssp-executable-initial-partition-singleton-shape:
  bmssp-singleton-bucket-shape bmssp-executable-initial-partition
proof −
  have empty-shape:
    bmssp-singleton-bucket-shape (bp-empty bmssp-block-size bmssp-bound)
    unfolding bmssp-singleton-bucket-shape-def bp-empty-def by simp
  show ?thesis
    unfolding bmssp-executable-initial-partition-def
    by (rule bmssp-singleton-bucket-shape-regularized-insert
        [OF empty-shape])
qed

lemma bmssp-dijkstra-loop-state-initial-executable:
  bmssp-dijkstra-loop-state (bmssp-vertices G src) [] [(src, 0)]
    bmssp-executable-initial-partition
proof −
  have src-vertices: src ∈ set (bmssp-vertices G src)
    using bmssp-vertices-set[of G src] by simp
```

```
  show ?thesis
    unfolding bmssp-executable-initial-partition-def
    by (rule bmssp-dijkstra-loop-state-initial[OF src-vertices])
qed

lemma bmssp-distances-quiescent-state-executable:
  obtains settled ds P S beta P1 where
    bmssp-distances G src = bmssp-normalize-dist ds
    bmssp-dijkstra-loop-state (bmssp-vertices G src) settled ds P
    bmssp-singleton-bucket-shape P
    bp-pull bmssp-block-size bmssp-bound P = (S, beta, P1)
    filter (λx. x ∈ S ∧ x ∉ set settled)
      (bmssp-vertices G src) = []
proof −
  let ?vertices = bmssp-vertices G src
  have fuel-gt:
    bmssp-executable-distances-fuel > length ?vertices − length ([] :: nat list)
  proof (cases ?vertices)
    case Nil
    then show ?thesis
      unfolding bmssp-executable-distances-fuel-def by simp
  next
    case (Cons v vs)
    then show ?thesis
      unfolding bmssp-executable-distances-fuel-def by simp
  qed
  have distinct-empty: distinct ([] :: nat list)
    by simp
  have quiescent:
    ∃ settled ds P S beta P1.
      bmssp-loop bmssp-executable-distances-fuel G ?vertices []
        [(src, 0)] bmssp-executable-initial-partition =
          bmssp-normalize-dist ds ∧
      bmssp-dijkstra-loop-state ?vertices settled ds P ∧
      bmssp-singleton-bucket-shape P ∧
      distinct settled ∧
      set [] ⊆ set settled ∧
      bp-pull bmssp-block-size bmssp-bound P = (S, beta, P1) ∧
      filter (λx. x ∈ S ∧ x ∉ set settled) ?vertices = []
    by (rule bmssp-loop-quiescent-state
        [OF bmssp-dijkstra-loop-state-initial-executable
          bmssp-executable-initial-partition-singleton-shape
          bmssp-edge-target-in-vertices bmssp-vertices-distinct
          distinct-empty fuel-gt])
  obtain settled ds P S beta P1 where final:
    bmssp-loop bmssp-executable-distances-fuel G ?vertices []
      [(src, 0)] bmssp-executable-initial-partition =
        bmssp-normalize-dist ds
    bmssp-dijkstra-loop-state ?vertices settled ds P
```

```
    bmssp-singleton-bucket-shape P
    bp-pull bmssp-block-size bmssp-bound P = (S, beta, P1)
    filter (λx. x ∈ S ∧ x ∉ set settled) ?vertices = []
    using quiescent by blast
  have distances-eq: bmssp-distances G src = bmssp-normalize-dist ds
    using final(1) unfolding bmssp-distances-unfold-executable .
  show ?thesis
    by (rule that[OF distances-eq final(2) final(3) final(4) final(5)])
qed

theorem bmssp-correct-instance:
  shows
    ∀(v, d)∈set (bmssp-distances G src).
       real d = nat-graph-dist G src v
    ∀ v∈nat-graph-vertices G. nat-graph-reachable G src v ⟶
       (∃ d. (v, d) ∈ set (bmssp-distances G src))
proof −
  obtain settled ds P S beta P1 where final:
    bmssp-distances G src = bmssp-normalize-dist ds
    bmssp-dijkstra-loop-state (bmssp-vertices G src) settled ds P
    bmssp-singleton-bucket-shape P
    bp-pull bmssp-block-size bmssp-bound P = (S, beta, P1)
    filter (λx. x ∈ S ∧ x ∉ set settled)
      (bmssp-vertices G src) = []
    by (rule bmssp-distances-quiescent-state-executable)
  have empty: keys-of (bp-view P) = {}
    by (rule bmssp-dijkstra-loop-state-quiescent-queue-empty
        [OF final(2) final(3) final(4) final(5)])
  show ∀(v, d)∈set (bmssp-distances G src).
       real d = nat-graph-dist G src v
  proof
    fix p
    assume p-mem: p ∈ set (bmssp-distances G src)
    obtain v d where p-eq: p = (v, d)
      by force
    have norm-mem: (v, d) ∈ set (bmssp-normalize-dist ds)
      using p-mem p-eq final(1) by simp
    have exact: real d = nat-graph-dist G src v
      by (rule bmssp-dijkstra-loop-state-normalize-exact-if-queue-empty
          [OF final(2) empty norm-mem])
    show case p of (v, d) ⇒ real d = nat-graph-dist G src v
      using p-eq exact by simp
  qed
  show ∀ v∈nat-graph-vertices G. nat-graph-reachable G src v ⟶
       (∃ d. (v, d) ∈ set (bmssp-distances G src))
  proof
    fix v
    assume vV: v ∈ nat-graph-vertices G
    show nat-graph-reachable G src v ⟶
```

```
      (∃ d. (v, d) ∈ set (bmssp-distances G src))
    proof
      assume reach: nat-graph-reachable G src v
      obtain d where norm-mem: (v, d) ∈ set (bmssp-normalize-dist ds)
        using bmssp-dijkstra-loop-state-normalize-complete-if-queue-empty
          [OF final(2) empty reach] by blast
      have (v, d) ∈ set (bmssp-distances G src)
        using norm-mem final(1) by simp
      then show ∃ d. (v, d) ∈ set (bmssp-distances G src)
        by blast
    qed
  qed
qed

lemma bmssp-distances-executable-integral-on-keys:
  real-label-integral-on (set (map fst (bmssp-distances G src)))
    (executable-label-of (bmssp-distances G src))
  by (rule executable-label-integral-on-keys)
    (rule bmssp-distances-distinct-keys)

end

theorem bmssp-correct-executable:
  assumes nat-graph-well-formed G
    and src ∈ nat-graph-vertices G
  shows
    ∀ (v, d) ∈ set (bmssp-distances G src).
      real d = nat-graph-dist G src v
    ∀ v ∈ nat-graph-vertices G. nat-graph-reachable G src v ⟶
      (∃ d. (v, d) ∈ set (bmssp-distances G src))
proof −
  interpret inst: nat-graph-instance G src
    by standard (rule assms(1), rule assms(2))
  have sound:
    ∀ (v, d) ∈ set (bmssp-distances G src).
      real d = nat-graph-dist G src v
    by (rule inst.bmssp-correct-instance(1))
  have complete:
    ∀ v ∈ nat-graph-vertices G. nat-graph-reachable G src v ⟶
      (∃ d. (v, d) ∈ set (bmssp-distances G src))
    by (rule inst.bmssp-correct-instance(2))
  show ∀ (v, d) ∈ set (bmssp-distances G src).
      real d = nat-graph-dist G src v
    by (rule sound)
  show ∀ v ∈ nat-graph-vertices G. nat-graph-reachable G src v ⟶
      (∃ d. (v, d) ∈ set (bmssp-distances G src))
    by (rule complete)
qed
```

end
theory *BMSSP-Executable-Refinement*
imports *BMSSP-Executable-Refinement-Internal*
begin

39 Executable Refinement

This theory is the public import point for the executable refinement chain. The detailed refinement proof lives in *BMSSP-Executable-Refinement-Internal*. The aliases below keep the outward-facing facts under the public theory name.

lemmas *bmssp-distances-distinct-keys* =
BMSSP-Executable-Refinement-Internal.bmssp-distances-distinct-keys

lemmas *bmssp-correct-executable* =
BMSSP-Executable-Refinement-Internal.bmssp-correct-executable

end
theory *BMSSP-Executable-Headline*
imports *BMSSP-Executable-Refinement*
begin

40 Executable Headline

This theory states the headline correctness theorem of the entire development. It is the entry point for a reader who wants one statement that summarises what is proved, and it is the theorem that should be consulted before any other. The body of the proof is a short forward reference to the verified executable contract.

The headline applies to the executable entry point *bmssp-distances*, which takes a finite directed graph of natural vertices with natural-valued edge weights and a source vertex, and returns a finite association list mapping each reachable vertex to its shortest-path distance from the source. The graph representation *nat-graph*, the well-formedness predicate *nat-graph-well-formed*, the vertex set *nat-graph-vertices*, the reachability relation *nat-graph-reachable*, and the shortest distance function *nat-graph-dist* are all defined earlier in the session and used unchanged here.

The theorem below is intentionally a single complete statement. It exposes soundness, completeness, and uniqueness of output keys together, so readers do not have to reconstruct the exact map property from several smaller auxiliary facts.

theorem *bmssp-correct-strong*:
assumes *well-formed*: *nat-graph-well-formed G*
and *src-in*: *src* ∈ *nat-graph-vertices G*
shows

```
    distinct (map fst (bmssp-distances G src)) ∧
     set (map fst (bmssp-distances G src)) =
       {v ∈ nat-graph-vertices G. nat-graph-reachable G src v} ∧
     (∀(v, d) ∈ set (bmssp-distances G src).
        real d = nat-graph-dist G src v)
proof −
  interpret inst: nat-graph-instance G src
    by standard (rule well-formed, rule src-in)
  have distinct-keys:
    distinct (map fst (bmssp-distances G src))
    by (rule BMSSP-Executable-Refinement.bmssp-distances-distinct-keys)
  have output-subset:
    set (map fst (bmssp-distances G src)) ⊆
      {v ∈ nat-graph-vertices G. nat-graph-reachable G src v}
  proof
    fix v
    assume v-key: v ∈ set (map fst (bmssp-distances G src))
    then obtain d where mem: (v, d) ∈ set (bmssp-distances G src)
      by force
    have v-vertices: v ∈ nat-graph-vertices G
      using inst.bmssp-distances-keys-subset-carrier v-key by blast
    have reachable: nat-graph-reachable G src v
      by (rule inst.bmssp-distances-output-reachable[OF mem])
    show v ∈ {v ∈ nat-graph-vertices G. nat-graph-reachable G src v}
      using v-vertices reachable by simp
  qed
  have reachable-subset:
    {v ∈ nat-graph-vertices G. nat-graph-reachable G src v} ⊆
      set (map fst (bmssp-distances G src))
  proof
    fix v
    assume v ∈ {v ∈ nat-graph-vertices G. nat-graph-reachable G src v}
    then have v-vertices: v ∈ nat-graph-vertices G
      and reachable: nat-graph-reachable G src v
      by simp-all
    obtain d where (v, d) ∈ set (bmssp-distances G src)
      using BMSSP-Executable-Refinement.bmssp-correct-executable(2)
        [OF well-formed src-in] v-vertices reachable by blast
    then show v ∈ set (map fst (bmssp-distances G src))
      by force
  qed
  have exact-distances:
    ∀(v, d) ∈ set (bmssp-distances G src).
       real d = nat-graph-dist G src v
    by (rule BMSSP-Executable-Refinement.bmssp-correct-executable(1)
        [OF well-formed src-in])
  have key-set:
    set (map fst (bmssp-distances G src)) =
      {v ∈ nat-graph-vertices G. nat-graph-reachable G src v}
```

using *output-subset reachable-subset* **by** *blast*
show *?thesis*
using *distinct-keys key-set exact-distances* **by** *blast*
qed

The proof combines the underlying executable theorem ⟦*nat-graph-well-formed ?G*; *?src* ∈ *nat-graph-vertices ?G*⟧ ⟹ ∀ (*v*, *d*)∈*set* (*bmssp-distances ?G ?src*). *real d* = *nat-graph-dist ?G ?src v*

⟦*nat-graph-well-formed ?G*; *?src* ∈ *nat-graph-vertices ?G*⟧ ⟹ ∀ *v*∈*nat-graph-vertices ?G*. *nat-graph-reachable ?G ?src v* ⟶ (∃ *d*. (*v*, *d*) ∈ *set* (*bmssp-distances ?G ?src*)) with the distinct-key and output-reachability facts from the executable refinement layer. The two clauses of ⟦*nat-graph-well-formed ?G*; *?src* ∈ *nat-graph-vertices ?G*⟧ ⟹ ∀ (*v*, *d*)∈*set* (*bmssp-distances ?G ?src*). *real d* = *nat-graph-dist ?G ?src v*

⟦*nat-graph-well-formed ?G*; *?src* ∈ *nat-graph-vertices ?G*⟧ ⟹ ∀ *v*∈*nat-graph-vertices ?G*. *nat-graph-reachable ?G ?src v* ⟶ (∃ *d*. (*v*, *d*) ∈ *set* (*bmssp-distances ?G ?src*)) give exact distances for every returned pair and completeness for every reachable vertex; the local key-set argument above adds the converse inclusion and the distinctness statement.

The proof is deliberately structured rather than written as a one-line automation script. The intention is to make the proof obligation visible and to expose how the headline is derived from the verified executable contract. The automated steps are confined to small set and tuple manipulations after the named refinement facts have been supplied. Tracing the proof reveals exactly which named theorem proves what.

In short, the entire development reduces to:

- *bmssp-distances* computes a finite list of pairs.
- For every well-formed graph and every source vertex of that graph, every returned pair gives the exact shortest-path distance from the source.
- For every well-formed graph and every source vertex of that graph, every reachable vertex appears in the output, and no unreachable vertex appears.

The remaining theories prove these facts. This file states them.

end
theory *BMSSP-Runtime-Instance*
imports *BMSSP-Top-Level-Bounds BMSSP-Executable-Base-Case*
begin

41 A Concrete Non-Vacuous Instance of the Runtime Locale

The asymptotic running-time headline lives in the *bounded-reduced-positive-instance* locale, whose interpretation yields the closed theorem *bmssp-runtime-headline-instance.bmssp-runtime-bigo-target*. A locale theorem only carries content once the locale is shown to be inhabited. This theory exhibits a concrete, non-trivial graph—the unit-weight directed path $0 \to 1 \to 2 \to 3$—and discharges every assumption of that locale, obtaining the $O(m * (ln\ n)\ powr\ (2\ /\ 3))$ bound for it as a closed, assumption-free statement. This certifies that the runtime headline is not vacuously true.

Uniqueness of shortest walks—the only non-routine locale assumption—is obtained from the verified simple-walk enumerator *exec-simple-walks-betw*: on this graph the enumerator returns exactly one simple walk to each vertex, so any two shortest walks to a vertex coincide.

definition *path-vs* :: *nat list* **where**
path-vs = [*0*, *1*, *2*, *3*]

definition *path-es* :: (*nat* × *nat*) *list* **where**
path-es = [(*0*, *1*), (*1*, *2*), (*2*, *3*)]

definition *path-V* :: *nat set* **where**
path-V = *set path-vs*

definition *path-E* :: (*nat* × *nat*) *set* **where**
path-E = *set path-es*

definition *path-weight* :: *nat* ⇒ *nat* ⇒ *real* **where**
path-weight u v = (*if* (*u*, *v*) ∈ *path-E then 1 else 0*)

lemma *path-vs-V*: *set path-vs* = *path-V*
unfolding *path-V-def* **by** (*rule refl*)

lemma *path-es-E*: *set path-es* = *path-E*
unfolding *path-E-def* **by** (*rule refl*)

lemma *path-positive-weight*: (*u*, *v*) ∈ *path-E* ⟹ *0* < *path-weight u v*
unfolding *path-weight-def* **by** *simp*

lemma *path-finite-weighted-digraph*:
finite-weighted-digraph path-V path-E path-weight 0
unfolding *finite-weighted-digraph-def*
path-V-def path-vs-def path-E-def path-es-def path-weight-def
by *auto*

interpretation *pfw*: *finite-weighted-digraph path-V path-E path-weight 0*
by (*rule path-finite-weighted-digraph*)

The enumerator returns exactly one simple walk from *0* to each vertex, checked by evaluation of the generated code.

lemma *path-walk-lists*:
  exec-simple-walks-betw path-vs path-es 0 0 = [[*0*]]
  exec-simple-walks-betw path-vs path-es 0 (*Suc 0*) = [[*0*, *1*]]
  exec-simple-walks-betw path-vs path-es 0 2 = [[*0*, *1*, *2*]]
  exec-simple-walks-betw path-vs path-es 0 3 = [[*0*, *1*, *2*, *3*]]
  by *eval+*

lemma *path-simple-walk-sets*:
  {*p*. *pfw.simple-walk-betw 0 p 0*} = {[*0*]}
  {*p*. *pfw.simple-walk-betw 0 p* (*Suc 0*)} = {[*0*, *1*]}
  {*p*. *pfw.simple-walk-betw 0 p 2*} = {[*0*, *1*, *2*]}
  {*p*. *pfw.simple-walk-betw 0 p 3*} = {[*0*, *1*, *2*, *3*]}
  using *pfw.set-exec-simple-walks-betw*[*OF path-vs-V path-es-E*, *symmetric*]
  by (*simp-all add*: *path-walk-lists*)

lemma *path-walk-vertex*:
  assumes *pfw.simple-walk-betw 0 p v*
  shows *v* ∈ *path-V*
proof −
  have *walk-p*: *pfw.walk p* **and** *p-ne*: *p* ≠ [] **and** *last-p*: *last p* = *v*
    using *assms* **unfolding** *pfw.simple-walk-betw-def pfw.walk-betw-def* **by** *auto*
  have *v* ∈ *set p*
    using *p-ne last-p* **by** (*metis last-in-set*)
  moreover have *set p* ⊆ *path-V*
    using *walk-p* **by** (*rule pfw.walk-set-subset*)
  ultimately show *?thesis* **by** *blast*
qed

lemma *path-unique-shortest-walk*:
  assumes *pfw.shortest-walk 0 p v* **and** *pfw.shortest-walk 0 q v*
  shows *p* = *q*
proof −
  have *p*: *pfw.simple-walk-betw 0 p v* **and** *q*: *pfw.simple-walk-betw 0 q v*
    using *assms* **unfolding** *pfw.shortest-walk-def* **by** *blast+*
  have *vV*: *v* ∈ *path-V*
    using *p* **by** (*rule path-walk-vertex*)
  have *pm*: *p* ∈ {*r*. *pfw.simple-walk-betw 0 r v*}
    using *p* **by** *simp*
  have *qm*: *q* ∈ {*r*. *pfw.simple-walk-betw 0 r v*}
    using *q* **by** *simp*
  from *vV* **consider** *v* = *0* | *v* = *Suc 0* | *v* = *2* | *v* = *3*
    unfolding *path-V-def path-vs-def* **by** *auto*
  then show *p* = *q*
  proof *cases*
    case *1* **with** *pm qm* **show** *?thesis* **by** (*simp add*: *path-simple-walk-sets*)
  next
    case *2* **with** *pm qm* **show** *?thesis* **by** (*simp add*: *path-simple-walk-sets*)

```
  next
    case 3 with pm qm show ?thesis by (simp add: path-simple-walk-sets)
  next
    case 4 with pm qm show ?thesis by (simp add: path-simple-walk-sets)
  qed
qed

lemma path-all-reachable:
  assumes v ∈ path-V
  shows pfw.reachable 0 v
proof −
  from assms consider v = 0 | v = Suc 0 | v = 2 | v = 3
    unfolding path-V-def path-vs-def by auto
  then show ?thesis
  proof cases
    case 1
    have [0] ∈ {r. pfw.simple-walk-betw 0 r 0}
      by (simp add: path-simple-walk-sets)
    with 1 show ?thesis unfolding pfw.reachable-def by auto
  next
    case 2
    have [0, 1] ∈ {r. pfw.simple-walk-betw 0 r (Suc 0)}
      by (simp add: path-simple-walk-sets)
    with 2 show ?thesis unfolding pfw.reachable-def by auto
  next
    case 3
    have [0, 1, 2] ∈ {r. pfw.simple-walk-betw 0 r 2}
      by (simp add: path-simple-walk-sets)
    with 3 show ?thesis unfolding pfw.reachable-def by auto
  next
    case 4
    have [0, 1, 2, 3] ∈ {r. pfw.simple-walk-betw 0 r 3}
      by (simp add: path-simple-walk-sets)
    with 4 show ?thesis unfolding pfw.reachable-def by auto
  qed
qed

interpretation pusd: unique-shortest-digraph path-V path-E path-weight 0
  by unfold-locales (rule path-unique-shortest-walk)

lemma path-edge-outdegree: pusd.edge-outdegree-le 1
  unfolding pusd.edge-outdegree-le-def
proof
  fix u :: nat
  assume u ∈ path-V
  then consider u = 0 | u = Suc 0 | u = 2 | u = 3
    unfolding path-V-def path-vs-def by auto
  then show card (pusd.outgoing-edges {u}) ≤ 1
  proof cases
```

```
    case 1
    then have pusd.outgoing-edges {u} = {(0, 1)}
      unfolding pusd.outgoing-edges-def by (auto simp: path-E-def path-es-def)
    then show ?thesis by simp
  next
    case 2
    then have pusd.outgoing-edges {u} = {(1, 2)}
      unfolding pusd.outgoing-edges-def by (auto simp: path-E-def path-es-def)
    then show ?thesis by simp
  next
    case 3
    then have pusd.outgoing-edges {u} = {(2, 3)}
      unfolding pusd.outgoing-edges-def by (auto simp: path-E-def path-es-def)
    then show ?thesis by simp
  next
    case 4
    then have pusd.outgoing-edges {u} = {}
      unfolding pusd.outgoing-edges-def by (auto simp: path-E-def path-es-def)
    then show ?thesis by simp
  qed
qed

interpretation pbr: bounded-reduced-positive-instance path-V path-E path-weight 0
1
  by unfold-locales
     (use path-positive-weight path-all-reachable path-edge-outdegree in auto)
```

The closed running-time bound for the concrete path instance. Because it is obtained by interpreting *bounded-reduced-positive-instance* on an explicit graph, it has no remaining locale hypotheses: it is an assumption-free witness that the BMSSP runtime headline is satisfiable by a genuine non-trivial digraph.

```
lemmas path-runtime-bigo-target =
  pbr.runtime-headline.bmssp-runtime-bigo-target

end
theory BMSSP-Bucketed-Cost-Bridge
  imports BMSSP-Bucketed-Partition BMSSP-Top-Level-Bounds
begin
```

42 Bridging the Bucketed Operation Costs to the Schedule Parameters

This theory closes the gap between the two cost layers of the development.

The costed BMSSP recurrence (the relations *direct-insert-costed-bmssp* and *exact-concrete-bmssp*) charges every direct insert at the abstract parameter t and every batch-prepended item at the abstract parameter h,

through the predicates *partition-insert-cost-bound* and *partition-batch-cost-bound*. The top-level analysis then sets t to the two-thirds logarithmic schedule parameter and h to the one-third parameter.

Independently, the bucketed partition theory proves that its concrete operations realise the abstract predicates with the paper-tight budgets *bp-insert-paper-budget* and *bp-batch-prepend-paper-budget*, each of the shape *const* + *floor-log* (*Suc* (*N div M*)).

What was missing is the arithmetic bridge: the realised bucketed budget *const* + *floor-log* (*Suc* (*N div M*)) is no larger than the schedule parameter whenever the ratio *N div M* is below the corresponding power of two. The bucket directory is searched in *floor-log* (*Suc* (*N div M*)) steps, so the only fact needed is that *floor-log* of a number strictly below $2 \mathbin{\hat{}} E$ does not exceed E. This theory proves that arithmetic, packages it into the two interface predicates, and exposes the resulting connected statements: with the bucketed costs in force, the per-level ratio bound is exactly what makes the abstract Insert and BatchPrepend cost parameters realisable at the logarithmic schedule scales used by the headline recurrence.

42.1 Logarithmic Arithmetic

The single arithmetic fact underlying the whole bridge: if a positive number is strictly below $2 \mathbin{\hat{}} E$, its binary logarithm is at most E. Everything about the bucketed budgets reduces to this through monotonicity of *floor-log*.

```
lemma floor-log-less-exp2-le:
  assumes n < 2 ^ E
  shows floor-log n ≤ E
proof −
  have n ≤ 2 ^ E
    using assms by simp
  then have floor-log n ≤ floor-log (2 ^ E)
    by (rule floor-log-le-iff)
  also have ... = E
    by (rule floor-log-power)
  finally show ?thesis .
qed

lemma floor-log-Suc-div-le:
  assumes Suc (N div M) ≤ 2 ^ E
  shows floor-log (Suc (N div M)) ≤ E
proof −
  have floor-log (Suc (N div M)) ≤ floor-log (2 ^ E)
    using assms by (rule floor-log-le-iff)
  also have ... = E
    by (rule floor-log-power)
  finally show ?thesis .
qed
```

The ratio condition we use throughout: at most $2 \mathbin{\hat{}} E$ blocks worth of entries are live, i.e. $N \mathbin{div} M < 2 \mathbin{\hat{}} E$. Equivalently $Suc\ (N \mathbin{div} M) \leq 2 \mathbin{\hat{}} E$, the form consumed by $Suc\ (?N \mathbin{div} ?M) \leq 2^{?E} \Longrightarrow floor\text{-}log\ (Suc\ (?N \mathbin{div} ?M)) \leq ?E$.

definition *ratio-below-pow2* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *bool* **where**
  ratio-below-pow2 N M E ⟷ *N div M < 2 ^ E*

lemma *ratio-below-pow2-floor-log*:
  assumes *ratio-below-pow2 N M E*
  shows *floor-log* (*Suc* (*N div M*)) ≤ *E*
proof −
  have *Suc* (*N div M*) ≤ *2 ^ E*
    using *assms* **unfolding** *ratio-below-pow2-def* **by** *simp*
  then show *?thesis*
    by (*rule floor-log-Suc-div-le*)
qed

42.2 Bucketed Budgets at the Ratio Bound

Under the ratio bound the two paper budgets collapse to a constant plus the exponent E. These are the budgets that the realisation corollaries $bp\text{-}regular\text{-}invariant\ ?P \Longrightarrow \exists t.\ partition\text{-}insert\text{-}cost\text{-}bound\ (bp\text{-}steps\text{-}of\ (c\text{-}bp\text{-}paper\text{-}insert\ ?x\ ?b\ ?P))\ t \wedge t = bp\text{-}insert\text{-}paper\text{-}budget\ (length\ (bp\text{-}entries\ ?P))\ (bp\text{-}block\text{-}size\ ?P)$ and $\exists t.\ partition\text{-}batch\text{-}cost\text{-}bound\ (bp\text{-}steps\text{-}of\ (c\text{-}bp\text{-}paper\text{-}batch\text{-}prepend\ ?xs\ ?P))\ t\ ?xs \wedge t = bp\text{-}batch\text{-}prepend\text{-}paper\text{-}budget\ (length\ ?xs)\ (bp\text{-}block\text{-}size\ ?P)$ attach to the concrete bucketed operations.

lemma *bp-insert-paper-budget-le-const-plus-exp*:
  assumes *ratio-below-pow2 N M E*
  shows *bp-insert-paper-budget N M* ≤ *9 + E*
  unfolding *bp-insert-paper-budget-def*
  using *ratio-below-pow2-floor-log*[*OF assms*] **by** *simp*

lemma *bp-batch-prepend-paper-budget-le-const-plus-exp*:
  assumes *ratio-below-pow2 N M E*
  shows *bp-batch-prepend-paper-budget N M* ≤ *2 + E*
  unfolding *bp-batch-prepend-paper-budget-def*
  using *ratio-below-pow2-floor-log*[*OF assms*] **by** *simp*

42.3 Monotonicity of the Abstract Cost Predicates

The abstract cost predicates are $c \leq t$ and $c \leq t * length\ xs$, so they are monotone in the budget parameter. Weakening a realised budget to a larger one therefore preserves the predicate; this is what lets us replace the precise bucketed budget by the constant-plus-exponent bound and then by the schedule parameter.

lemma *partition-insert-cost-bound-mono*:

fixes $c\ t\ t' :: nat$
assumes *partition-insert-cost-bound* $c\ t$
and $t \leq t'$
shows *partition-insert-cost-bound* $c\ t'$
proof −
have $c \leq t$
using *assms*(*1*) **unfolding** *partition-insert-cost-bound-def* **.**
then have $c \leq t'$
using *assms*(*2*) **by** (*rule le-trans*)
then show *?thesis*
unfolding *partition-insert-cost-bound-def* **.**
qed

lemma *partition-batch-cost-bound-mono*:
fixes $c\ t\ t' :: nat$
assumes *partition-batch-cost-bound* $c\ t\ xs$
and $t \leq t'$
shows *partition-batch-cost-bound* $c\ t'\ xs$
proof −
have $c \leq t * length\ xs$
using *assms*(*1*) **unfolding** *partition-batch-cost-bound-def* **.**
also have $\ldots \leq t' * length\ xs$
using *assms*(*2*) **by** (*rule mult-right-mono*) *simp*
finally show *?thesis*
unfolding *partition-batch-cost-bound-def* **.**
qed

42.4 Bucketed Operations Realise the Schedule Cost Parameters

These are the central connecting facts. Each says: the *actual* step count of a concrete bucketed operation satisfies the abstract cost predicate with the budget set to *const* + *E*, provided the operation runs on a state whose live ratio is below $2\ \widehat{}\ E$. No free abstract parameter remains: the budget is the measured cost of the bucketed implementation, bounded through the logarithmic arithmetic above.

These bridge the realisation corollaries *bp-regular-invariant ?P* $\Longrightarrow \exists\, t.$ *partition-insert-cost-bound* (*bp-steps-of* (*c-bp-paper-insert ?x ?b ?P*)) $t \wedge t$ = *bp-insert-paper-budget* (*length* (*bp-entries ?P*)) (*bp-block-size ?P*) and $\exists\, t.$ *partition-batch-cost-bound* (*bp-steps-of* (*c-bp-paper-batch-prepend ?xs ?P*)) *t ?xs* $\wedge$ *t* = *bp-batch-prepend-paper-budget* (*length ?xs*) (*bp-block-size ?P*) (which exhibit the budget as *some* value equal to the paper budget) to a concrete, schedule-facing inequality.

theorem *c-bp-paper-insert-realises-const-plus-exp*:
assumes *reg*: *bp-regular-invariant P*
and *ratio*: *ratio-below-pow2* (*length* (*bp-entries P*)) (*bp-block-size P*) *E*
shows *partition-insert-cost-bound*

```
    (bp-steps-of (c-bp-paper-insert x b P)) (9 + E)
proof −
  have base: partition-insert-cost-bound
      (bp-steps-of (c-bp-paper-insert x b P))
      (bp-lazy-insert-amortized-budget P)
    by (rule c-bp-paper-insert-regular-partition-insert-cost-bound[OF reg])
  have budget-eq: bp-lazy-insert-amortized-budget P =
      bp-insert-paper-budget (length (bp-entries P)) (bp-block-size P)
    by (rule bp-lazy-insert-amortized-budget-paper-form)
  have budget-le: bp-insert-paper-budget
      (length (bp-entries P)) (bp-block-size P) ≤ 9 + E
    by (rule bp-insert-paper-budget-le-const-plus-exp[OF ratio])
  show ?thesis
    by (rule partition-insert-cost-bound-mono[OF base])
       (use budget-eq budget-le in linarith)
qed

theorem c-bp-paper-batch-prepend-realises-const-plus-exp:
  assumes ratio: ratio-below-pow2 (length xs) (bp-block-size P) E
  shows partition-batch-cost-bound
    (bp-steps-of (c-bp-paper-batch-prepend xs P)) (2 + E) xs
proof −
  have base: partition-batch-cost-bound
      (bp-steps-of (c-bp-paper-batch-prepend xs P))
      (bp-ratio-log-budget (length xs) (bp-block-size P)) xs
    by (rule c-bp-paper-batch-prepend-batch-cost-bound)
  have budget-le: bp-ratio-log-budget (length xs) (bp-block-size P) ≤ 2 + E
  proof −
    have bp-ratio-log-budget (length xs) (bp-block-size P) ≤
        bp-batch-prepend-paper-budget (length xs) (bp-block-size P)
      by (rule bp-ratio-log-budget-le-batch-prepend-paper-budget)
    also have ... ≤ 2 + E
      by (rule bp-batch-prepend-paper-budget-le-const-plus-exp[OF ratio])
    finally show ?thesis .
  qed
  show ?thesis
    by (rule partition-batch-cost-bound-mono[OF base budget-le])
qed
```

The Pull cost needs no ratio bound: the bucketed Pull already realises the abstract pull predicate exactly (cost = *card S*), independent of the live ratio. We re-expose it here so the three operation costs sit together as the connected interface used by the recurrence.

```
theorem c-bp-paper-pull-realises-partition-pull-cost-bound:
  assumes pull: bp-result-of (c-bp-paper-pull M B P) = (S, beta, P')
  shows partition-pull-cost-bound (bp-steps-of (c-bp-paper-pull M B P)) S
  by (rule bp-realises-partition-pull-cost-bound[OF pull])
```

42.5 The Bucketed Cost Parameters Sit at the Schedule Scales

The previous theorems show, per operation, that the measured bucketed step count satisfies the abstract cost predicate with budget *const* + E, where E is the live-ratio exponent. The headline recurrence is solved by $[\![0 < ?Cn;\ 0 < ?Ca;\ 0 \leq ?Cl;\ 0 < ?Cr;\ 0 < ?Ct;\ 0 < ?Ch;\ \forall_F\ n\ in\ sequentially.\ real\ n \leq ?Cn * real\ (?m\ n);\ \forall_F\ n\ in\ sequentially.\ real\ (?A\ n) \leq ?Ca * sssp\text{-}log\text{-}factor\text{-}one\text{-}third\ n;\ \forall_F\ n\ in\ sequentially.\ real\ (?l\ n) \leq ?Cl * sssp\text{-}log\text{-}factor\text{-}one\text{-}third\ n;\ \forall_F\ n\ in\ sequentially.\ real\ (?R\ n) \leq ?Cr * sssp\text{-}log\text{-}factor\ n;\ \forall_F\ n\ in\ sequentially.\ real\ (?t\ n) \leq ?Ct * sssp\text{-}log\text{-}factor\ n;\ \forall_F\ n\ in\ sequentially.\ real\ (?H\ n) \leq ?Ch * sssp\text{-}log\text{-}factor\text{-}one\text{-}third\ n;\ \forall_F\ n\ in\ sequentially.\ ?T\ n \leq bmssp\text{-}refined\text{-}graph\text{-}time\text{-}bound\ ?A\ ?R\ ?H\ ?l\ ?t\ ?m\ n]\!] \Longrightarrow (\lambda n.\ real\ (?T\ n)) \in O(sssp\text{-}time\text{-}target\ ?m)$, which needs the Insert cost parameter t to grow like $log^{2/3}$ and the BatchPrepend cost parameter H to grow like $log^{1/3}$.

We now discharge exactly those two envelope facts for the bucketed budgets. For Insert, the live ratio is allowed to reach the two-thirds schedule parameter, so the budget is *9* + *sssp-log-two-thirds-param n*; a constant added to $log^{2/3}$ is still $O(log^{2/3})$. For BatchPrepend, the per-pull batch has length at most a constant multiple of the block size (bounded out-degree), so its ratio exponent is a constant c; the budget is then *2* + c, a constant, which is $O(log^{1/3})$ because the one-third factor tends to infinity. These are the genuine reasons the bucketed costs fit the deterministic schedule.

lemma *sssp-log-factor-at-top*:
filterlim sssp-log-factor at-top at-top
proof −
have *filterlim* (λ*n*. *sssp-log-factor-one-third n* ∗ *sssp-log-factor-one-third n*)
at-top at-top
by (*rule filterlim-at-top-mult-at-top*
[*OF sssp-log-factor-one-third-at-top sssp-log-factor-one-third-at-top*])
moreover have
(λ*n*. *sssp-log-factor-one-third n* ∗ *sssp-log-factor-one-third n*) =
sssp-log-factor
by (*rule ext*) (*rule sssp-log-factor-one-third-square*)
ultimately show *?thesis* **by** *simp*
qed

Insert envelope: a constant plus the two-thirds schedule parameter is eventually dominated by twice the two-thirds log factor. This is the *t-bound* hypothesis with the bucketed Insert budget in the parameter slot.

lemma *const-plus-two-thirds-param-le-two-thirds-factor*:
eventually
(λ*n*. *real* (*c* + *sssp-log-two-thirds-param n*) ≤
real (*c* + *2*) ∗ *sssp-log-factor n*) *at-top*
using *eventually-one-le-sssp-log-factor*

```
proof eventually-elim
  case (elim n)
  have one-le: 1 ≤ sssp-log-factor n
    using elim .
  have param-le: real (sssp-log-two-thirds-param n) ≤ sssp-log-factor n + 1
  proof −
    have nonneg: 0 ≤ sssp-log-factor n
      using one-le by linarith
    have real (nat ⌈sssp-log-factor n⌉) = of-int ⌈sssp-log-factor n⌉
      using of-nat-int-ceiling[OF nonneg] .
    moreover have of-int ⌈sssp-log-factor n⌉ ≤ sssp-log-factor n + 1
      using ceiling-correct[of sssp-log-factor n] by linarith
    ultimately show ?thesis
      unfolding sssp-log-two-thirds-param-def by simp
  qed
  have c-le: real c ≤ real c * sssp-log-factor n
  proof −
    have real c * 1 ≤ real c * sssp-log-factor n
      by (rule mult-left-mono[OF one-le]) simp
    then show ?thesis by simp
  qed
  have real (c + sssp-log-two-thirds-param n) =
      real c + real (sssp-log-two-thirds-param n)
    by simp
  also have ... ≤ real c + (sssp-log-factor n + 1)
    using param-le by linarith
  also have ... ≤ real c * sssp-log-factor n + 2 * sssp-log-factor n
    using c-le one-le by linarith
  also have ... = real (c + 2) * sssp-log-factor n
    by (simp add: algebra-simps)
  finally show ?case .
qed
```

BatchPrepend envelope: a constant budget is eventually dominated by any positive multiple of the one-third log factor, because that factor tends to infinity. This is the *H-bound* hypothesis with the bucketed BatchPrepend budget (a constant under bounded out-degree) in the parameter slot.

```
lemma const-le-one-third-factor:
  eventually
    (λn. real (c::nat) ≤ sssp-log-factor-one-third n) at-top
proof −
  have filterlim sssp-log-factor-one-third at-top at-top
    by (rule sssp-log-factor-one-third-at-top)
  then have eventually (λn. real c ≤ sssp-log-factor-one-third n) at-top
    unfolding filterlim-at-top by blast
  then show ?thesis .
qed
```

The two envelopes packaged as the schedule-facing component bounds.

The first states that the bucketed Insert cost parameter $\lambda n.\ 9 + sssp\text{-}log\text{-}two\text{-}thirds\text{-}param\ n$ is $O(log^{2/3})$; the second that the bucketed BatchPrepend cost parameter $\lambda n.\ 2 + c$ is $O(log^{1/3})$. These are exactly the *t-bound* and *H-bound* premises consumed by the recurrence solver, now justified by the realised bucketed budgets instead of left as free parameters.

theorem *bucketed-insert-param-two-thirds-envelope*:
eventually
(λn. *real* (*9* + *sssp-log-two-thirds-param n*) ≤
11 ∗ *sssp-log-factor n*) *at-top*
using *const-plus-two-thirds-param-le-two-thirds-factor*[*of 9*]
by *simp*

theorem *bucketed-batch-param-one-third-envelope*:
eventually
(λn. *real* (*2* + *c*) ≤ *real* (*Suc* (*2* + *c*)) ∗ *sssp-log-factor-one-third n*)
at-top
using *const-le-one-third-factor*[*of 2* + *c*]
proof *eventually-elim*
case (*elim n*)
have *nonneg*: *0* ≤ *sssp-log-factor-one-third n*
using *sssp-log-factor-one-third-pos*[*of n*] **by** *linarith*
have *real* (*2* + *c*) ≤ *sssp-log-factor-one-third n*
using *elim* **.**
also have ... ≤ *real* (*Suc* (*2* + *c*)) ∗ *sssp-log-factor-one-third n*
proof −
have *sssp-log-factor-one-third n* ∗ *1* ≤
sssp-log-factor-one-third n ∗ *real* (*Suc* (*2* + *c*))
by (*rule mult-left-mono*) (*use nonneg* **in** *simp-all*)
then show *?thesis*
by (*simp add*: *mult.commute*)
qed
finally show *?case* **.**
qed

42.6 Connected Headline: Bucketed-Costed BMSSP is $O(m\ log^{2/3}\ n)$

The capstone. It plugs the bucketed cost parameters directly into the recurrence solver ⟦*0* < *?Cn*; *0* < *?Ca*; *0* ≤ *?Cl*; *0* < *?Cr*; *0* < *?Ct*; *0* < *?Ch*; ∀$_F$ *n in sequentially. real n* ≤ *?Cn* ∗ *real* (*?m n*); ∀$_F$ *n in sequentially. real* (*?A n*) ≤ *?Ca* ∗ *sssp-log-factor-one-third n*; ∀$_F$ *n in sequentially. real* (*?l n*) ≤ *?Cl* ∗ *sssp-log-factor-one-third n*; ∀$_F$ *n in sequentially. real* (*?R n*) ≤ *?Cr* ∗ *sssp-log-factor n*; ∀$_F$ *n in sequentially. real* (*?t n*) ≤ *?Ct* ∗ *sssp-log-factor n*; ∀$_F$ *n in sequentially. real* (*?H n*) ≤ *?Ch* ∗ *sssp-log-factor-one-third n*; ∀$_F$ *n in sequentially. ?T n* ≤ *bmssp-refined-graph-time-bound ?A ?R ?H ?l ?t ?m n*⟧ ⟹ (λ*n. real* (*?T n*)) ∈ *O*(*sssp-time-target ?m*). The Insert cost slot *t* carries the bucketed Insert budget $\lambda n.\ 9 + sssp\text{-}log\text{-}two\text{-}thirds\text{-}param$

n, and the BatchPrepend cost slot H carries the bucketed BatchPrepend budget λ-. *2 + cbatch* (a constant under bounded out-degree, when the per-pull batch length stays within a constant multiple of the block size). Crucially, no cost parameter is left free: both are the measured bucketed budgets established above. The conclusion is the deterministic SSSP target $O(m * log^{2/3}\ n)$.

Recall the argument order of *bmssp-refined-graph-time-bound* is *A R H l t m n*, with body *2 A (2 l + 1) n + (R + t + l H) m*; here the fifth argument t is the Insert cost and the third argument H is the BatchPrepend per-item cost.

This theorem is what unifies the two previously separate claims of the entry: the asymptotic recurrence (claim 2) and the bucketed operation costs (claim 3). In the original development the recurrence's Insert cost was a free parameter that the headline simply set to the schedule scale; here that parameter is the actual bucketed budget, and the bound still holds.

theorem *bucketed-refined-cost-bigo-sssp-time-target*:
  fixes *m A l R T* :: *nat* ⇒ *nat*
    and *cbatch* :: *nat*
    and *Cn Ca Cl Cr* :: *real*
  assumes *Cn-pos*: *0 < Cn*
    and *Ca-pos*: *0 < Ca*
    and *Cl-nonneg*: *0 ≤ Cl*
    and *Cr-pos*: *0 < Cr*
    and *vertex-count-dominated*:
      eventually (λ*n*. *real n ≤ Cn * real* (*m n*)) *at-top*
    and *A-bound*:
      eventually (λ*n*. *real* (*A n*) ≤ *Ca * sssp-log-factor-one-third n*) *at-top*
    and *l-bound*:
      eventually (λ*n*. *real* (*l n*) ≤ *Cl * sssp-log-factor-one-third n*) *at-top*
    and *R-bound*:
      eventually (λ*n*. *real* (*R n*) ≤ *Cr * sssp-log-factor n*) *at-top*
    and *cost-bound*:
      eventually
        (λ*n*. *T n* ≤ *bmssp-refined-graph-time-bound*
          A R (λ-. *2 + cbatch*) *l*
          (λ*n*. *9 + sssp-log-two-thirds-param n*) *m n*)
        at-top
  shows (λ*n*. *real* (*T n*)) ∈ *O*(λ*n*. *sssp-time-target m n*)
proof (*rule bmssp-refined-cost-bound-bigo-sssp-time-target-from-component-bounds*
    [**where** *Cn* = *Cn* **and** *Ca* = *Ca* **and** *Cl* = *Cl* **and** *Cr* = *Cr*
      and *Ct* = *11* **and** *Ch* = *real* (*Suc* (*2 + cbatch*))
      and *A* = *A* **and** *l* = *l* **and** *R* = *R*
      and *t* = λ*n*. *9 + sssp-log-two-thirds-param n*
      and *H* = λ-. *2 + cbatch* **and** *m* = *m* **and** *T* = *T*])
  show *0 < Cn* **by** (*rule Cn-pos*)
  show *0 < Ca* **by** (*rule Ca-pos*)
  show *0 ≤ Cl* **by** (*rule Cl-nonneg*)

```
  show 0 < Cr by (rule Cr-pos)
  show 0 < (11::real) by simp
  show 0 < real (Suc (2 + cbatch)) by simp
  show eventually (λn. real n ≤ Cn * real (m n)) at-top
    by (rule vertex-count-dominated)
  show eventually (λn. real (A n) ≤ Ca * sssp-log-factor-one-third n) at-top
    by (rule A-bound)
  show eventually (λn. real (l n) ≤ Cl * sssp-log-factor-one-third n) at-top
    by (rule l-bound)
  show eventually (λn. real (R n) ≤ Cr * sssp-log-factor n) at-top
    by (rule R-bound)
  show eventually
      (λn. real (9 + sssp-log-two-thirds-param n) ≤ 11 * sssp-log-factor n)
      at-top
    by (rule bucketed-insert-param-two-thirds-envelope)
  show eventually
      (λn. real ((λ-. 2 + cbatch) n) ≤
        real (Suc (2 + cbatch)) * sssp-log-factor-one-third n) at-top
    by (rule bucketed-batch-param-one-third-envelope)
  show eventually
      (λn. T n ≤ bmssp-refined-graph-time-bound
          A R (λ-. 2 + cbatch) l
          (λn. 9 + sssp-log-two-thirds-param n) m n) at-top
    by (rule cost-bound)
qed
```

Specialised to the sparse-graph regime where the vertex term is dominated by the edge term, this is the familiar deterministic SSSP envelope $O(m * (ln\ n)\ powr\ (2/3))$, now carrying the bucketed Insert and BatchPrepend costs in the recurrence rather than free abstract parameters.

```
corollary bucketed-refined-cost-bigo-log-target:
  fixes m A l R T :: nat ⇒ nat
    and cbatch :: nat
    and Cn Ca Cl Cr :: real
  assumes Cn-pos: 0 < Cn
    and Ca-pos: 0 < Ca
    and Cl-nonneg: 0 ≤ Cl
    and Cr-pos: 0 < Cr
    and vertex-count-dominated:
      eventually (λn. real n ≤ Cn * real (m n)) at-top
    and A-bound:
      eventually (λn. real (A n) ≤ Ca * sssp-log-factor-one-third n) at-top
    and l-bound:
      eventually (λn. real (l n) ≤ Cl * sssp-log-factor-one-third n) at-top
    and R-bound:
      eventually (λn. real (R n) ≤ Cr * sssp-log-factor n) at-top
    and cost-bound:
      eventually
        (λn. T n ≤ bmssp-refined-graph-time-bound
```

```
      A R (λ-. 2 + cbatch) l
      (λn. 9 + sssp-log-two-thirds-param n) m n)
    at-top
 shows (λn. real (T n)) ∈
   O(λn. real (m n) * (ln (real n + 2)) powr (2 / 3))
proof −
 have (λn. real (T n)) ∈ O(λn. sssp-time-target m n)
   by (rule bucketed-refined-cost-bigo-sssp-time-target
     [OF Cn-pos Ca-pos Cl-nonneg Cr-pos vertex-count-dominated
       A-bound l-bound R-bound cost-bound])
 then show ?thesis
   unfolding sssp-time-target-def sssp-log-factor-def .
qed

end
theory BMSSP-Path-Family
 imports BMSSP-Top-Level-Bounds BMSSP-Executable-Base-Case
begin
```

43 A Size-Indexed Family of Runtime Instances

The runtime headline *bmssp-runtime-headline-instance ?V ?E ?w ?s ?D* $\Longrightarrow$ *(λn. real (bmssp-runtime-headline-instance.T-bmssp ?V ?E ?w ?s ?D n)) ∈ O(λn. real (bmssp-runtime-headline-instance.m ?E n) * ln (real n + 2) powr (2 / 3))* is a theorem of the *bounded-reduced-positive-instance* locale. The theory `BMSSP_Runtime_Instance.thy` interprets that locale on the single fixed graph $0 \to 1 \to 2 \to 3$, which certifies non-vacuity at one size only, and whose edge count is therefore a constant.

This theory removes that limitation. For every n it builds the unit-weight directed path $0 \to 1 \to \ldots \to n$, proves that it satisfies all assumptions of *bounded-reduced-positive-instance uniformly in n*, and obtains the deterministic running-time bound for a family whose vertex count $n + 1$ and edge count n grow without bound. The only non-routine assumption, uniqueness of shortest walks, is discharged by the structural fact that in such a path the only walk starting at 0 is an initial segment $[0, 1, \ldots, k]$. Unlike the fixed instance, no step uses code evaluation; the argument is a single induction valid for all n.

43.1 The Path Graph of Size n

```
definition pf-V :: nat ⇒ nat set where
  pf-V n = {0..n}

definition pf-E :: nat ⇒ (nat × nat) set where
  pf-E n = {(i, Suc i) | i. i < n}
```

definition *pf-w* :: *nat* ⇒ *nat* ⇒ *real* **where**
pf-w u v = *1*

lemma *pf-E-iff*: (*a*, *b*) ∈ *pf-E n* ⟷ *a* < *n* ∧ *b* = *Suc a*
unfolding *pf-E-def* **by** *auto*

lemma *pf-finite-weighted-digraph*:
finite-weighted-digraph (*pf-V n*) (*pf-E n*) *pf-w 0*
proof
show *finite* (*pf-V n*)
unfolding *pf-V-def* **by** *simp*
show *0* ∈ *pf-V n*
unfolding *pf-V-def* **by** *simp*
show ⋀*u v*. (*u*, *v*) ∈ *pf-E n* ⟹ *u* ∈ *pf-V n* ∧ *v* ∈ *pf-V n*
unfolding *pf-V-def* **by** (*auto simp*: *pf-E-iff*)
show ⋀*u v*. (*u*, *v*) ∈ *pf-E n* ⟹ *0* ≤ *pf-w u v*
unfolding *pf-w-def* **by** *simp*
qed

43.2 The Path Family Locale

Fixing the size n in a locale lets us interpret the graph locales for *pf-V n*, *pf-E n*, while keeping n schematic across the whole development.

locale *path-family* =
fixes *n* :: *nat*
begin

interpretation *pf*: *finite-weighted-digraph pf-V n pf-E n pf-w 0*
by (*rule pf-finite-weighted-digraph*)

43.2.1 The Only Walk From the Source Is an Initial Segment

Vertex i has the single out-edge (*i*, *Suc i*), so any walk starting at *0* is forced to visit *0*, *1*, *2*, ... in order: its i-th element is i.

lemma *pf-walk-nth*:
assumes *walk*: *pf.walk p*
and *hd0*: *hd p* = *0*
and *i*: *i* < *length p*
shows *p* ! *i* = *i*
using *i*
proof (*induction i*)
case *0*
have *p* ≠ [] **using** *assms*(*3*) **by** *auto*
then show *?case* **using** *hd0* **by** (*simp add*: *hd-conv-nth*)
next
case (*Suc i*)
have *i-lt*: *i* < *length p* **using** *Suc.prems* **by** *simp*
have *ih*: *p* ! *i* = *i* **using** *Suc.IH*[*OF i-lt*] **.**

```
  have edge: (p ! i, p ! Suc i) ∈ pf-E n
    by (rule pf.walk-nth-edge[OF walk Suc.prems])
  have p ! Suc i = Suc (p ! i)
    using edge by (simp add: pf-E-iff)
  then show ?case using ih by simp
qed

lemma pf-walk-from-0-eq-upt:
  assumes walk: pf.walk p
    and hd0: hd p = 0
  shows p = [0..<length p]
proof (rule nth-equalityI)
  show length p = length [0..<length p] by simp
  fix i assume i < length p
  then show p ! i = [0..<length p] ! i
    using pf-walk-nth[OF walk hd0] by simp
qed

lemma pf-simple-walk-betw-eq:
  assumes pf.simple-walk-betw 0 p v
  shows p = [0..<Suc v]
proof −
  have walk: pf.walk p and ne: p ≠ []
    and hd0: hd p = 0 and last-v: last p = v
    using assms unfolding pf.simple-walk-betw-def pf.walk-betw-def by auto
  have p-eq: p = [0..<length p]
    by (rule pf-walk-from-0-eq-upt[OF walk hd0])
  have len-pos: 0 < length p using ne by auto
  have p ! (length p − 1) = length p − 1
    using pf-walk-nth[OF walk hd0] len-pos by simp
  moreover have last p = p ! (length p − 1)
    using ne by (simp add: last-conv-nth)
  ultimately have v = length p − 1 using last-v by simp
  then have length p = Suc v using len-pos by simp
  then show ?thesis using p-eq by simp
qed
```

43.2.2 Reachability and Uniqueness

```
lemma pf-upt-simple-walk:
  assumes v ≤ n
  shows pf.simple-walk-betw 0 [0..<Suc v] v
proof −
  have walk: pf.walk [0..<Suc v]
    using assms
  proof (induction v)
    case 0
    have [0..<Suc 0] = [0] by simp
    moreover have pf.walk [0]
```

```
    using pf.walk.simps(2)[of 0] unfolding pf-V-def by simp
   ultimately show ?case by simp
  next
   case (Suc v)
   have v-le: v ≤ n using Suc.prems by simp
   have wv: pf.walk [0..<Suc v] by (rule Suc.IH[OF v-le])
   have upt-eq: [0..<Suc (Suc v)] = [0..<Suc v] @ [Suc v]
    by (simp add: upt-Suc-append)
   have edge: (v, Suc v) ∈ pf-E n
    using Suc.prems by (simp add: pf-E-iff)
   have last-wv: last [0..<Suc v] = v by simp
   have ne: [0..<Suc v] ≠ [] by simp
   have hd-wv: hd [0..<Suc v] = 0
    using upt-conv-Cons[of 0 Suc v] by simp
   have sb: pf.simple-walk-betw 0 [0..<Suc v] v
    unfolding pf.simple-walk-betw-def pf.walk-betw-def
    using wv last-wv ne hd-wv by simp
   have fresh: Suc v ∉ set [0..<Suc v] by simp
   have sw: pf.simple-walk-betw 0 ([0..<Suc v] @ [Suc v]) (Suc v)
    by (rule pf.simple-walk-snoc[OF sb edge fresh])
   then have pf.walk ([0..<Suc v] @ [Suc v])
    unfolding pf.simple-walk-betw-def pf.walk-betw-def by simp
   then show ?case
    unfolding upt-eq .
  qed
  have hd0: hd [0..<Suc v] = 0
   using upt-conv-Cons[of 0 Suc v] by simp
  have last-v: last [0..<Suc v] = v by simp
  show ?thesis
   unfolding pf.simple-walk-betw-def pf.walk-betw-def
   using walk hd0 last-v by simp
qed

lemma pf-all-reachable:
  assumes v ∈ pf-V n
  shows pf.reachable 0 v
proof −
  have v ≤ n using assms unfolding pf-V-def by simp
  then have pf.simple-walk-betw 0 [0..<Suc v] v
   by (rule pf-upt-simple-walk)
  then show ?thesis unfolding pf.reachable-def by blast
qed

lemma pf-unique-shortest-walk:
  assumes pf.shortest-walk 0 p v and pf.shortest-walk 0 q v
  shows p = q
proof −
  have sp: pf.simple-walk-betw 0 p v and sq: pf.simple-walk-betw 0 q v
   using assms unfolding pf.shortest-walk-def by blast+
```

have $p = [0..<Suc\ v]$ **by** (*rule pf-simple-walk-betw-eq*[*OF sp*])
moreover have $q = [0..<Suc\ v]$ **by** (*rule pf-simple-walk-betw-eq*[*OF sq*])
ultimately show *?thesis* **by** *simp*
qed

With uniqueness in hand we may interpret the *unique-shortest-digraph* layer, which is where *unique-shortest-digraph.edge-outdegree-le* and *unique-shortest-digraph.outgoing-edges* live.

interpretation *pf-usd*: *unique-shortest-digraph pf-V n pf-E n pf-w 0*
by *unfold-locales* (*rule pf-unique-shortest-walk*)

43.2.3 Bounded Out-Degree and Positive Weights

lemma *pf-positive-weight*: $(u, v) \in pf\text{-}E\ n \Longrightarrow 0 < pf\text{-}w\ u\ v$
unfolding *pf-w-def* **by** *simp*

lemma *pf-edge-outdegree*: *pf-usd.edge-outdegree-le 1*
unfolding *pf-usd.edge-outdegree-le-def*
proof
fix $u :: nat$
assume $u \in pf\text{-}V\ n$
have *sub*: *pf-usd.outgoing-edges* $\{u\} \subseteq \{(u, Suc\ u)\}$
unfolding *pf-usd.outgoing-edges-def* **by** (*auto simp*: *pf-E-iff*)
have *card* (*pf-usd.outgoing-edges* $\{u\}$) $\leq$ *card* $\{(u, Suc\ u)\}$
by (*rule card-mono*[*OF - sub*]) *simp*
then show *card* (*pf-usd.outgoing-edges* $\{u\}$) ≤ 1 **by** *simp*
qed

43.2.4 The Reduced Positive Instance and Its Running-Time Bound

sublocale *pf-bri*: *bounded-reduced-positive-instance pf-V n pf-E n pf-w 0 1*
proof *unfold-locales*
show $\bigwedge u\ v.\ (u, v) \in pf\text{-}E\ n \Longrightarrow 0 < pf\text{-}w\ u\ v$
by (*rule pf-positive-weight*)
show $\bigwedge v.\ v \in pf\text{-}V\ n \Longrightarrow pf.reachable\ 0\ v$
by (*rule pf-all-reachable*)
show *pf-usd.edge-outdegree-le 1*
by (*rule pf-edge-outdegree*)
qed

The vertex count is $n + 1$ and the edge count is exactly n: both grow with the size parameter, unlike the constant edge count of the fixed instance.

lemma *pf-vertex-count*: *pf-usd.vertex-count* $= Suc\ n$
proof −
have *pf-usd.vertex-count* = *card* (*pf-V n*)
by (*rule pf-usd.vertex-count-def*)
also have $\ldots$ = *card* $\{0..n\}$
unfolding *pf-V-def* **by** (*rule refl*)

```
  also have ... = Suc n
    by simp
  finally show ?thesis .
qed

lemma pf-edge-count: pf-usd.edge-count = n
proof −
  have inj: inj-on (λi. (i, Suc i)) {i. i < n}
    by (rule inj-onI) auto
  have pf-E n = (λi. (i, Suc i)) ‘ {i. i < n}
    unfolding pf-E-def by auto
  then have pf-usd.edge-count = card ((λi. (i, Suc i)) ‘ {i. i < n})
    unfolding pf-usd.edge-count-def by simp
  also have ... = card {i. i < n}
    using inj by (simp add: card-image)
  also have ... = n by simp
  finally show ?thesis .
qed
```

The closed running-time bound for the path of size n, with all locale hypotheses discharged. This is the deterministic $O(m * (ln\ N)\ powr\ (2/3))$ envelope of the headline, now holding for a member of the family of *every* size. The time and size functions are named locally so the re-exported statement below is free of deep qualified names.

```
definition pf-T :: nat ⇒ nat where
  pf-T = pf-bri.runtime-headline.T-bmssp

definition pf-m :: nat ⇒ nat where
  pf-m = pf-bri.runtime-headline.m

lemma pf-runtime-bigo-target:
  (λN. real (pf-T N)) ∈
    O(λN. real (pf-m N) * (ln (real N + 2)) powr (2 / 3))
  unfolding pf-T-def pf-m-def
  by (rule pf-bri.runtime-headline.bmssp-runtime-bigo-target)
```

Locale-level names for the graph-size measures, so the external re-exports avoid deep interpretation-qualified constants.

```
definition pf-verts :: nat where
  pf-verts = pf-usd.vertex-count

definition pf-edges :: nat where
  pf-edges = pf-usd.edge-count

lemma pf-verts-eq: pf-verts = Suc n
  unfolding pf-verts-def by (rule pf-vertex-count)

lemma pf-edges-eq: pf-edges = n
  unfolding pf-edges-def by (rule pf-edge-count)
```

end

43.3 The Bound Holds at Every Size

Re-exported outside the locale: for every size parameter n, the path graph on $n + 1$ vertices and n edges satisfies the deterministic BMSSP running-time bound, and its edge count really is n. Because n is universally quantified, this is a single statement witnessing non-vacuity of the runtime locale across all sizes, not merely at one fixed graph.

theorem *path-family-runtime-bigo-target*:
fixes *n* :: *nat*
shows (λ*N*. *real* (*path-family.pf-T n N*)) ∈
O(λ*N*. *real* (*path-family.pf-m n N*) ∗ (*ln* (*real N* + *2*)) *powr* (*2* / *3*))
proof −
interpret *path-family n* **.**
show *?thesis*
by (*rule pf-runtime-bigo-target*)
qed

theorem *path-family-edge-count*:
fixes *n* :: *nat*
shows *path-family.pf-edges n* = *n*
proof −
interpret *path-family n* **.**
show *?thesis*
by (*rule pf-edges-eq*)
qed

theorem *path-family-vertex-count*:
fixes *n* :: *nat*
shows *path-family.pf-verts n* = *Suc n*
proof −
interpret *path-family n* **.**
show *?thesis*
by (*rule pf-verts-eq*)
qed

end
theory *BMSSP-Bucketed-Runtime*
imports *BMSSP-Top-Level-Bounds BMSSP-Bucketed-Cost-Bridge*
begin

44 Non-Conditional Bucketed Running-Time Bound

The capstone of `BMSSP_Bucketed_Cost_Bridge.thy` is conditional: it assumes a cost function already bounded by the refined graph-time expression with the bucketed costs in the Insert and BatchPrepend slots. This theory discharges that assumption against an actual costed BMSSP run, producing a hypothesis-free $O(m * log^{2/3}\ n)$ bound that carries the bucketed operation costs.

The key structural fact making this possible is that the local-budget amortised theorem *direct-insert-costed-bmssp-amortized-bound-from-local-budgets-with-invariants* is parametric in the level geometry *M-of/cap* and in the per-operation costs *t/h independently*. We therefore keep the level geometry at the paper's schedule exponents $p = log^{1/3}$, $q = log^{2/3}$ (so the recursion tree, pivot growth, and pull blocks are unchanged), while charging Insert at the bucketed budget *9 + q* and BatchPrepend at *2 + p*. Because the FindPivots pivot set *find-pivots-pivots-capped* depends only on the geometry, not on the costs, the same emptiness argument that makes the existing headline total applies here verbatim, so the bound is total in *N*.

Throughout, *D* is a fixed outdegree bound and the running graph is the locale graph of *strict-tie-breaking-digraph*.

44.1 A One-Third-Scale Envelope for the BatchPrepend Cost

The BatchPrepend cost *2 + p* here grows like $p = log^{1/3}$ (the schedule parameter), not a constant. This envelope, the one-third analogue of the two-thirds envelope in the cost bridge, shows it stays within a constant multiple of $log^{1/3}$ — exactly the *H-bound* the recurrence solver needs.

```
lemma const-plus-one-third-param-le-one-third-factor:
  eventually
    (λn. real (c + sssp-log-one-third-param n) ≤
      real (c + 2) * sssp-log-factor-one-third n) at-top
  using eventually-one-le-sssp-log-factor-one-third
proof eventually-elim
  case (elim n)
  have one-le: 1 ≤ sssp-log-factor-one-third n
    using elim .
  have param-le:
    real (sssp-log-one-third-param n) ≤ sssp-log-factor-one-third n + 1
  proof −
    have nonneg: 0 ≤ sssp-log-factor-one-third n
      using one-le by linarith
    have real (nat ⌈sssp-log-factor-one-third n⌉) =
        of-int ⌈sssp-log-factor-one-third n⌉
      using of-nat-int-ceiling[OF nonneg] .
```

```
    moreover have of-int ⌈sssp-log-factor-one-third n⌉ ≤
        sssp-log-factor-one-third n + 1
      using ceiling-correct[of sssp-log-factor-one-third n] by linarith
    ultimately show ?thesis
      unfolding sssp-log-one-third-param-def by simp
  qed
  have c-le: real c ≤ real c * sssp-log-factor-one-third n
  proof −
    have real c * 1 ≤ real c * sssp-log-factor-one-third n
      by (rule mult-left-mono[OF one-le]) simp
    then show ?thesis by simp
  qed
  have real (c + sssp-log-one-third-param n) =
      real c + real (sssp-log-one-third-param n)
    by simp
  also have ... ≤ real c + (sssp-log-factor-one-third n + 1)
    using param-le by linarith
  also have ... ≤ real c * sssp-log-factor-one-third n +
      2 * sssp-log-factor-one-third n
    using c-le one-le by linarith
  also have ... = real (c + 2) * sssp-log-factor-one-third n
    by (simp add: algebra-simps)
  finally show ?case .
qed
```

An affine version used for the vertex factor $A = a * p + b$: it is dominated by $(2\ a + b)$ times the one-third log factor, hence $O(log^{1/3})$.

```
lemma affine-one-third-param-le-one-third-factor:
  eventually
    (λn. real (a * sssp-log-one-third-param n + b) ≤
      real (2 * a + b) * sssp-log-factor-one-third n) at-top
  using eventually-one-le-sssp-log-factor-one-third
proof eventually-elim
  case (elim n)
  have one-le: 1 ≤ sssp-log-factor-one-third n
    using elim .
  have nonneg: 0 ≤ sssp-log-factor-one-third n
    using one-le by linarith
  let ?L = sssp-log-factor-one-third n
  have param-le:
    real (sssp-log-one-third-param n) ≤ ?L + 1
  proof −
    have real (nat ⌈?L⌉) = of-int ⌈?L⌉
      using of-nat-int-ceiling[OF nonneg] .
    moreover have of-int ⌈?L⌉ ≤ ?L + 1
      using ceiling-correct[of ?L] by linarith
    ultimately show ?thesis
      unfolding sssp-log-one-third-param-def by simp
  qed
```

```
  have real (a * sssp-log-one-third-param n + b) =
      real a * real (sssp-log-one-third-param n) + real b
    by simp
  also have ... ≤ real a * (?L + 1) + real b
    using param-le by (intro add-mono mult-left-mono) simp-all
  also have ... = real a * ?L + (real a + real b)
    by (simp add: algebra-simps)
  also have ... ≤ real a * ?L + (real a + real b) * ?L
  proof −
    have (real a + real b) * 1 ≤ (real a + real b) * ?L
      by (rule mult-left-mono[OF one-le]) simp
    then show ?thesis by simp
  qed
  also have ... = real (2 * a + b) * ?L
    by (simp add: algebra-simps)
  finally show ?case .
qed

context strict-tie-breaking-digraph
begin
```

44.2 A Decoupled Amortised Bound: Geometry Exponent vs. Operation Cost

This is the existing level-cap amortised theorem with one generalisation: the level geometry uses an exponent *tg* that is decoupled from the per-Insert cost *t*. In the original theorem the two coincide (*M-of* = *bmssp-level-cap k t*); here *M-of* = *bmssp-level-cap k tg*, while *t* remains the cost charged at each direct insert. The proof is the original one: the local-budget amortised lemma is parametric in *M-of*, *cap*, *t*, *h*, *A*, *R*, *k*, so the only coupling needed is *M-of i* ≤ *cap* for $i \le l$, which still holds by monotonicity of *bmssp-level-cap* in the level.

```
theorem finite-initial-label-decoupled-amortized:
  assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
    and degree: edge-outdegree-le Δ
    and degree-factor: Δ ≤ A
    and R-pos: 0 < R
    and insert-factor: t ≤ A * k
    and insert-scaled-factor: t ≤ A-insert * k
    and seen-scaled-factor: k * Δ + A-insert ≤ 2 * A
    and source-factor: Suc h ≤ 2 * A
    and k-pos: 0 < k
    and run:
      direct-insert-costed-bmssp Δ (bmssp-level-cap k tg) t h k
        (bmssp-level-cap k tg l) l
        finite-initial-label {s} Infinity d' Infinity U c
  shows sssp-correct d' ∧
    c ≤ bmssp-refined-graph-time-bound (λ-. A) (λ-. R) (λ-. h)
```

```
    (λ-. l) (λ-. t) (λ-. edge-count) vertex-count
proof
  show sssp-correct d'
    by (rule finite-initial-label-direct-insert-costed-top-level-correct
      [OF all-reachable run])
next
  have pre: bmssp-pre-full finite-initial-label {s} Infinity
    using all-reachable finite-initial-label-source-complete
    by (rule top-bmssp-pre-full)
  have sound: sound-label finite-initial-label
    using finite-initial-label-sound[OF all-reachable] .
  have S-reaches: ⋀x. x ∈ {s} ⟹ reachable s x
    using all-reachable source-in-V by blast
  have below: ⋀x. x ∈ {s} ⟹
      below-bound (finite-initial-label x) Infinity
    by simp
  have top-cap: k * card {s} ≤ bmssp-level-cap k tg l
  proof −
    have one-le: 1 ≤ bmssp-level-width tg l
      unfolding bmssp-level-width-def by simp
    have k * 1 ≤ k * bmssp-level-width tg l
      by (rule mult-left-mono[OF one-le]) simp
    then show ?thesis
      unfolding bmssp-level-cap-def by simp
  qed
  have top-anti: tree-antichain {s}
    by simp
  have amortized:
    c ≤ bmssp-amortized-cost-bound (2 * A) R h t l (2 * l + 1)
      Infinity U
  proof (rule direct-insert-costed-bmssp-amortized-bound-from-local-budgets-with-
invariants
      [where A = 2 * A and R = R,
        OF - R-pos source-factor k-pos - - run sound pre S-reaches below
          top-cap top-anti])
    show Δ ≤ 2 * A
      using degree-factor by simp
  next
    fix i
    assume i ≤ l
    then show bmssp-level-cap k tg i ≤ bmssp-level-cap k tg l
      by (rule bmssp-level-cap-mono)
  next
    fix l' d S B D c-insert d' a betas bs B' Us U-loop c-loop
      child-costs U
    assume D-def:
        D = label-partition-view
          (find-pivots-label-capped k (bmssp-level-cap k tg l) d S B)
          (find-pivots-pivots-capped k (bmssp-level-cap k tg l) d S B)
```

and *insert*:
partition-initial-insert-cost-bound c-insert t
(*find-pivots-pivots-capped k* (*bmssp-level-cap k tg l*) *d S B*)
and *loop*:
direct-insert-costed-partition-loop-state Δ (*bmssp-level-cap k tg*) *t h k*
(*bmssp-level-cap k tg l*) *l'*
(*find-pivots-label-capped k* (*bmssp-level-cap k tg l*) *d S B*)
(*find-pivots-pivots-capped k* (*bmssp-level-cap k tg l*) *d S B*) *B d' D*
a betas bs B' Us U-loop c-loop child-costs
and *complete*:
complete-on d'
{*v* ∈ *bound-tree S B'*.
find-pivots-label-capped k (*bmssp-level-cap k tg l*) *d S B v* =
dist s v}
and *U-def*:
U = *U-loop* ∪
{*v* ∈ *bound-tree S B'*.
find-pivots-label-capped k (*bmssp-level-cap k tg l*) *d S B v* =
dist s v}
and *sound-s*: *sound-label d*
and *pre-s*: *bmssp-pre-full d S B*
and *reaches-s*: ⋀*x*. *x* ∈ *S* ⟹ *reachable s x*
and *below-s*: ⋀*x*. *x* ∈ *S* ⟹ *below-bound* (*d x*) *B*
and *S-k-cap*: *k* ∗ *card S* ≤ *bmssp-level-cap k tg l*
and *anti*: *tree-antichain S*
have *seen-success*:
B' = *B* ⟹
card (*find-pivots-seen-capped k* (*bmssp-level-cap k tg l*) *d S B*)
≤ *card U*
by (*rule direct-insert-costed-step-seen-success*
[*OF loop U-def sound-s pre-s reaches-s below-s*])
show *fp-iter-capped-scan-cost k* (*bmssp-level-cap k tg l*) *d S S B* +
c-insert ≤ *2* ∗ *A* ∗ *card U*
by (*rule direct-insert-costed-capped-step-scan-insert-budget-from-scaled-seen-or-threshold*
[*OF degree degree-factor insert-factor insert-scaled-factor*
seen-scaled-factor insert loop U-def sound-s pre-s reaches-s
S-k-cap anti k-pos seen-success])
qed
have *post-full*:
bmssp-post-full finite-initial-label {*s*} *Infinity d' Infinity U*
by (*rule direct-insert-costed-bmssp-correct*[*OF run sound pre S-reaches*])
have *post*: *bmssp-post finite-initial-label* {*s*} *Infinity d' Infinity U*
using *bmssp-post-full-imp-post*[*OF post-full*] .
have *U-V*: *U* = *V*
using *post bound-tree-source-infinity*[*OF all-reachable*]
unfolding *bmssp-post-def* **by** *auto*
have *graph-bound*:
bmssp-amortized-cost-bound (*2* ∗ *A*) *R h t l* (*2* ∗ *l* + *1*) *Infinity U*

```
    ≤ bmssp-refined-graph-time-bound (λ-. A) (λ-. R) (λ-. h)
      (λ-. l) (λ-. t) (λ-. edge-count) vertex-count
  proof −
    have U-card: card U = vertex-count
      unfolding U-V vertex-count-def by simp
    have out-le: card (outgoing-edges U) ≤ edge-count
      by (rule edge-count-outgoing-bound)
    have range-le: card (outgoing-edges-range U 0 Infinity) ≤ edge-count
      by (rule card-outgoing-edges-range-le-edge-count)
    have out-term:
      (R + l * h) * card (outgoing-edges U) ≤
       (R + l * h) * edge-count
      using out-le by simp
    have range-term:
      t * card (outgoing-edges-range U 0 Infinity) ≤ t * edge-count
      using range-le by simp
    show ?thesis
      unfolding bmssp-amortized-cost-bound-def bmssp-refined-graph-time-bound-def
      using U-card out-term range-term by (simp add: algebra-simps; linarith)
  qed
  show c ≤ bmssp-refined-graph-time-bound (λ-. A) (λ-. R) (λ-. h)
      (λ-. l) (λ-. t) (λ-. edge-count) vertex-count
    using amortized graph-bound by linarith
qed
```

44.3 A Costed Run With Decoupled Geometry and Bucketed Costs

bmssp-bucketed-run D N is a top-level costed run whose level geometry uses the schedule exponents p, q but whose Insert cost is the bucketed budget $9 + q$ and whose BatchPrepend cost is $2 + p$.

```
definition bmssp-bucketed-run ::
  nat ⇒ nat ⇒ ('a ⇒ real) ⇒ 'a set ⇒ nat ⇒ bool where
  bmssp-bucketed-run D N d' U c ⟷
    (let p = sssp-log-one-third-param N;
         q = sssp-log-two-thirds-param N
     in exact-concrete-bmssp D (bmssp-level-cap p q) (9 + q) (2 + p) p
       (bmssp-level-cap p q p) p
       finite-initial-label {s} Infinity d' Infinity U c)

definition bmssp-bucketed-cost :: nat ⇒ nat ⇒ nat ⇒ bool where
  bmssp-bucketed-cost D N c ⟷ (∃ d' U. bmssp-bucketed-run D N d' U c)

definition bmssp-bucketed-time :: nat ⇒ nat ⇒ nat where
  bmssp-bucketed-time D N = (LEAST c. bmssp-bucketed-cost D N c)
```

44.4 Totality of the Bucketed Cost

The same FindPivots-emptiness argument as the existing headline, with the bucketed costs in the cost slots. Since *exact-concrete-bmssp-Suc-exists-if-pivots-empty-same-bound* is parametric in the costs, the substitution is immediate.

lemma *bmssp-bucketed-cost-exists-if-top-pivots-empty*:
  assumes *pivots-empty*:
    find-pivots-pivots-capped (*sssp-log-one-third-param N*)
      (*bmssp-level-cap* (*sssp-log-one-third-param N*)
        (*sssp-log-two-thirds-param N*) (*sssp-log-one-third-param N*))
      finite-initial-label {*s*} *Infinity* = {}
  shows ∃ *c*. *bmssp-bucketed-cost D N c*
proof −
  let *?p* = *sssp-log-one-third-param N*
  let *?q* = *sssp-log-two-thirds-param N*
  have *p-eq*: *?p* = *Suc* (*?p* − *1*)
    using *sssp-log-one-third-param-pos*[*of N*] **by** *simp*
  have *source-subset*: {*s*} ⊆ *V*
    using *source-in-V* **by** *simp*
  have *run*:
    ∃ *d′ U c*.
      exact-concrete-bmssp D (*bmssp-level-cap ?p ?q*) (*9* + *?q*) (*2* + *?p*) *?p*
        (*bmssp-level-cap ?p ?q ?p*) (*Suc* (*?p* − *1*))
        finite-initial-label {*s*} *Infinity d′ Infinity U c*
    by (*rule exact-concrete-bmssp-Suc-exists-if-pivots-empty-same-bound*
      [*OF source-subset - pivots-empty*]) *simp*
  then obtain *d′ U c* **where**
    exact-concrete-bmssp D (*bmssp-level-cap ?p ?q*) (*9* + *?q*) (*2* + *?p*) *?p*
      (*bmssp-level-cap ?p ?q ?p*) (*Suc* (*?p* − *1*))
      finite-initial-label {*s*} *Infinity d′ Infinity U c*
    by *blast*
  then have *root-run*:
    exact-concrete-bmssp D (*bmssp-level-cap ?p ?q*) (*9* + *?q*) (*2* + *?p*) *?p*
      (*bmssp-level-cap ?p ?q ?p*) *?p*
      finite-initial-label {*s*} *Infinity d′ Infinity U c*
    using *p-eq* **by** *simp*
  have *bmssp-bucketed-run D N d′ U c*
    unfolding *bmssp-bucketed-run-def*
    using *root-run* **by** (*simp add*: *Let-def*)
  then have *bmssp-bucketed-cost D N c*
    unfolding *bmssp-bucketed-cost-def* **by** *blast*
  then show *?thesis*
    by *blast*
qed

lemma *eventually-bmssp-bucketed-cost*:
  eventually (λ*N*. ∃ *c*. *bmssp-bucketed-cost D N c*) *at-top*
  using *eventually-top-level-pivots-empty*[*of finite-initial-label Infinity*]

proof *eventually-elim*
case (*elim N*)
show *?case*
by (*rule bmssp-bucketed-cost-exists-if-top-pivots-empty*[*OF elim*])
qed

lemma *bmssp-bucketed-time-witness*:
assumes *bmssp-bucketed-cost D N c*
shows *bmssp-bucketed-cost D N* (*bmssp-bucketed-time D N*)
unfolding *bmssp-bucketed-time-def*
by (*rule LeastI-ex*) (*use assms* **in** *blast*)

44.5 Cost Bound for a Bucketed Run

A bucketed run refines a direct-insert run with the same (decoupled) geometry and costs, to which the decoupled amortised theorem applies. Choosing $A = Suc\ D * p + 9$ makes the local-budget side conditions hold for *all* N (the binding constraint $9 + q \le A * p$ follows from $q \le p^2$ and $9 \le 9 * p$), and A still grows only like $log^{1/3}$.

lemma *bmssp-bucketed-run-refined-bound*:
assumes *all-reachable*: $\bigwedge v.\ v \in V \Longrightarrow$ *reachable s v*
and *degree*: *edge-outdegree-le D*
and *run*: *bmssp-bucketed-run D N d′ U c*
shows *c* ≤ *bmssp-refined-graph-time-bound*
(λ-. *Suc D* ∗ *sssp-log-one-third-param N* + *9*)
(λ-. *sssp-log-two-thirds-param N*)
(λ-. *2* + *sssp-log-one-third-param N*)
(λ-. *sssp-log-one-third-param N*)
(λ-. *9* + *sssp-log-two-thirds-param N*)
(λ-. *edge-count*) *vertex-count*
proof −
let *?p* = *sssp-log-one-third-param N*
let *?q* = *sssp-log-two-thirds-param N*
let *?A* = *Suc D* ∗ *?p* + *9*
have *direct*:
direct-insert-costed-bmssp D (*bmssp-level-cap ?p ?q*) (*9* + *?q*) (*2* + *?p*) *?p*
(*bmssp-level-cap ?p ?q ?p*) *?p*
finite-initial-label {*s*} *Infinity d′ Infinity U c*
by (*rule exact-concrete-bmssp-refines-direct-insert*
[*OF run*[*unfolded bmssp-bucketed-run-def Let-def*]])
have *p-pos*: *0* < *?p* **by** *simp*
have *p-ge1*: *1* ≤ *?p* **using** *p-pos* **by** *linarith*
have *q-le-sq*: *?q* ≤ *?p* ∗ *?p*
by (*rule sssp-log-two-thirds-param-le-one-third-square*)
have *degree-factor*: *D* ≤ *?A*
proof −
have *D* ≤ *Suc D* ∗ *?p*
using *p-pos* **by** (*cases ?p*) *simp-all*

```
  then show ?thesis by simp
 qed
 have insert-factor: 9 + ?q ≤ ?A * ?p
 proof −
  have 9 + ?q ≤ 9 * ?p + ?p * ?p
   using q-le-sq p-ge1 by (simp add: algebra-simps; linarith)
  also have ... ≤ 9 * ?p + Suc D * ?p * ?p
   by simp
  also have ... = ?A * ?p
   by (simp add: algebra-simps)
  finally show ?thesis .
 qed
 have insert-scaled-factor: 9 + ?q ≤ ?A * ?p
  by (rule insert-factor)
 have seen-scaled-factor: ?p * D + ?A ≤ 2 * ?A
 proof −
  have ?p * D ≤ Suc D * ?p
   by (simp add: algebra-simps)
  also have ... ≤ ?A
   by simp
  finally have ?p * D ≤ ?A .
  then show ?thesis by simp
 qed
 have source-factor: Suc (2 + ?p) ≤ 2 * ?A
 proof −
  have 2 * ?A = 2 * Suc D * ?p + 18
   by (simp add: algebra-simps)
  moreover have 2 * ?p ≤ 2 * Suc D * ?p
   by (simp add: algebra-simps)
  ultimately have 2 * ?p + 18 ≤ 2 * ?A
   by linarith
  moreover have Suc (2 + ?p) ≤ 2 * ?p + 18
   by simp
  ultimately show ?thesis by linarith
 qed
 have R-pos: 0 < ?q
  by (rule sssp-log-two-thirds-param-pos)
 have sssp-correct d' ∧
  c ≤ bmssp-refined-graph-time-bound
   (λ-. ?A) (λ-. ?q) (λ-. 2 + ?p)
   (λ-. ?p) (λ-. 9 + ?q) (λ-. edge-count) vertex-count
  by (rule finite-initial-label-decoupled-amortized
   [OF all-reachable degree degree-factor R-pos insert-factor
    insert-scaled-factor seen-scaled-factor source-factor p-pos direct])
 then show ?thesis
  by blast
qed
```

44.6 The Non-Conditional Bucketed Running-Time Bound

Assembling the pieces. Totality of the bucketed cost (via the FindPivots-emptiness argument) discharges the existence premise; the cost bound discharges the recurrence premise; and the schedule envelopes for the five cost parameters discharge the component bounds. Nothing is left conditional: the bucketed BMSSP running time is $O(m * log^{2/3}\ n)$.

theorem *bmssp-bucketed-time-bigo-sssp-time-target*:
  fixes *m* :: *nat* ⇒ *nat*
    and *Cn Cm* :: *real*
  assumes *all-reachable*: ⋀*v*. *v* ∈ *V* ⟹ *reachable s v*
    and *degree*: *edge-outdegree-le D*
    and *Cn-pos*: *0* < *Cn*
    and *Cm-pos*: *0* < *Cm*
    and *vc-dom*: *eventually* (λ*n*. *real vertex-count* ≤ *Cn* ∗ *real* (*m n*)) *at-top*
    and *ec-dom*: *eventually* (λ*n*. *real edge-count* ≤ *Cm* ∗ *real* (*m n*)) *at-top*
  shows (λ*n*. *real* (*bmssp-bucketed-time D n*)) ∈ *O*(λ*n*. *sssp-time-target m n*)
proof (*rule bmssp-refined-cost-bound-bigo-sssp-time-target-from-component-bounds-slack*)
  show *0* < *Cn* **by** (*rule Cn-pos*)
  show *0* < *Cm* **by** (*rule Cm-pos*)
  show *eventually* (λ*n*. *real* ((λ-. *vertex-count*) *n*) ≤ *Cn* ∗ *real* (*m n*)) *at-top*
    using *vc-dom* **by** *simp*
  show *eventually* (λ*n*. *real* ((λ-. *edge-count*) *n*) ≤ *Cm* ∗ *real* (*m n*)) *at-top*
    using *ec-dom* **by** *simp*
  show *eventually*
      (λ*n*. *real* (*Suc D* ∗ *sssp-log-one-third-param n* + *9*) ≤
        real (*2* ∗ *Suc D* + *9*) ∗ *sssp-log-factor-one-third n*) *at-top*
    using *affine-one-third-param-le-one-third-factor*[*of Suc D 9*] **by** *simp*
  show *eventually*
      (λ*n*. *real* (*sssp-log-one-third-param n*) ≤ *2* ∗ *sssp-log-factor-one-third n*)
      at-top
    using *const-plus-one-third-param-le-one-third-factor*[*of 0*] **by** *simp*
  show *eventually*
      (λ*n*. *real* (*sssp-log-two-thirds-param n*) ≤ *2* ∗ *sssp-log-factor n*) *at-top*
    using *const-plus-two-thirds-param-le-two-thirds-factor*[*of 0*] **by** *simp*
  show *eventually*
      (λ*n*. *real* (*9* + *sssp-log-two-thirds-param n*) ≤ *11* ∗ *sssp-log-factor n*)
      at-top
    using *const-plus-two-thirds-param-le-two-thirds-factor*[*of 9*] **by** *simp*
  show *eventually*
      (λ*n*. *real* (*2* + *sssp-log-one-third-param n*) ≤
        4 ∗ *sssp-log-factor-one-third n*) *at-top*
    using *const-plus-one-third-param-le-one-third-factor*[*of 2*] **by** *simp*
  have *cost-ev*: *eventually*
      (λ*n*. *bmssp-bucketed-time D n* ≤
        bmssp-refined-graph-time-bound
          (λ-. *Suc D* ∗ *sssp-log-one-third-param n* + *9*)
          (λ-. *sssp-log-two-thirds-param n*)

```
      (λ-. 2 + sssp-log-one-third-param n)
      (λ-. sssp-log-one-third-param n)
      (λ-. 9 + sssp-log-two-thirds-param n)
      (λ-. edge-count) vertex-count) at-top
  proof (rule eventually-mono[OF eventually-bmssp-bucketed-cost])
    fix n :: nat
    assume ∃ c. bmssp-bucketed-cost D n c
    then obtain c where c: bmssp-bucketed-cost D n c
      by blast
    have cost: bmssp-bucketed-cost D n (bmssp-bucketed-time D n)
      by (rule bmssp-bucketed-time-witness[OF c])
    then obtain d' U where run:
      bmssp-bucketed-run D n d' U (bmssp-bucketed-time D n)
      unfolding bmssp-bucketed-cost-def by blast
    show bmssp-bucketed-time D n ≤
        bmssp-refined-graph-time-bound
          (λ-. Suc D * sssp-log-one-third-param n + 9)
          (λ-. sssp-log-two-thirds-param n)
          (λ-. 2 + sssp-log-one-third-param n)
          (λ-. sssp-log-one-third-param n)
          (λ-. 9 + sssp-log-two-thirds-param n)
          (λ-. edge-count) vertex-count
      by (rule bmssp-bucketed-run-refined-bound[OF all-reachable degree run])
  qed
  show eventually
      (λn. bmssp-bucketed-time D n ≤
        bmssp-refined-graph-time-bound
          (λ-. Suc D * sssp-log-one-third-param n + 9)
          (λ-. sssp-log-two-thirds-param n)
          (λ-. 2 + sssp-log-one-third-param n)
          (λ-. sssp-log-one-third-param n)
          (λ-. 9 + sssp-log-two-thirds-param n)
          (λ-. (λ-. edge-count) n) ((λ-. vertex-count) n)) at-top
    using cost-ev by simp
qed auto
```

The sparse-graph specialisation, fully closed: for the bounded-outdegree locale graph with at least one edge, the bucketed BMSSP running time is $O(m * (ln\ n)\ powr\ (2/3))$, where the Insert and BatchPrepend costs charged inside the recurrence are the actual bucketed budgets, not free parameters.

```
theorem bmssp-bucketed-runtime-bigo-target:
  assumes all-reachable: ⋀v. v ∈ V ⟹ reachable s v
    and degree: edge-outdegree-le D
    and edge-count-pos: 0 < edge-count
  shows (λn. real (bmssp-bucketed-time D n)) ∈
    O(λn. real edge-count * (ln (real n + 2)) powr (2 / 3))
proof −
  have vertex-le: vertex-count ≤ 2 * edge-count
    by (rule vertex-count-le-twice-edge-count-if-all-reachable
```

```
    [OF all-reachable edge-count-pos])
  have (λn. real (bmssp-bucketed-time D n)) ∈
    O(λn. sssp-time-target (λ-. edge-count) n)
  proof (rule bmssp-bucketed-time-bigo-sssp-time-target [OF all-reachable degree])
    show 0 < (2::real) by simp
    show 0 < (1::real) by simp
    show eventually (λn. real vertex-count ≤ 2 * real edge-count) at-top
      using vertex-le by simp
    show eventually (λn. real edge-count ≤ 1 * real edge-count) at-top
      by simp
  qed
  then show ?thesis
    unfolding sssp-time-target-def sssp-log-factor-def by simp
qed

end

end
```

References

[1] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. DOI: `https://doi.org/10.1007/BF01386390`.

[2] R. Duan, J. Mao, X. Mao, X. Shu, and L. Yin. Breaking the sorting barrier for directed single-source shortest paths. In *Proceedings of the 57th Annual ACM Symposium on Theory of Computing*, 2025. DOI: `https://doi.org/10.1145/3717823.3718179`.

[3] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987. DOI: `https://doi.org/10.1145/28869.28874`.