

# Purely Functional, Simple, and Efficient Implementation of Prim and Dijkstra

Peter Lammich      Tobias Nipkow

June 17, 2024

## Abstract

We verify purely functional, simple and efficient implementations of Prim's and Dijkstra's algorithms. This constitutes the first verification of an executable and even efficient version of Prim's algorithm. This entry formalizes the second part of our ITP-2019 proof pearl *Purely Functional, Simple and Efficient Priority Search Trees and Applications to Prim and Dijkstra* [3].

# Contents

<b>1</b>	<b>Prim's Minimum Spanning Tree Algorithm</b>	<b>4</b>
1.1	Undirected Graphs . . . . .	4
1.1.1	Nodes and Edges . . . . .	5
1.1.2	Connectedness Relation . . . . .	6
1.1.3	Constructing Graphs . . . . .	7
1.1.4	Paths . . . . .	8
1.1.5	Cycles . . . . .	13
1.1.6	Connected Graphs . . . . .	17
1.1.7	Component Containing Node . . . . .	17
1.1.8	Trees . . . . .	19
1.1.9	Spanning Trees . . . . .	25
1.2	Weighted Undirected Graphs . . . . .	26
1.2.1	Minimum Spanning Trees . . . . .	28
1.3	Abstract Graph Datatype . . . . .	28
1.3.1	Abstract Weighted Graph . . . . .	28
1.3.2	Generic From-List Algorithm . . . . .	29
1.4	Abstract Prim Algorithm . . . . .	32
1.4.1	Generic Algorithm: Light Edges . . . . .	32
1.4.2	Abstract Prim: Growing a Tree . . . . .	34
1.4.3	Prim: Using a Priority Queue . . . . .	39
1.4.4	Refinement of Inner Foreach Loop . . . . .	48
1.5	Implementation of Weighted Undirected Graph by Map . . . . .	50
1.5.1	Doubleton Set to Pair . . . . .	50
1.5.2	Generic Implementation . . . . .	50
1.6	Implementation of Prim's Algorithm . . . . .	54
1.6.1	Implementation using ADT Interfaces . . . . .	54
1.6.2	Refinement of State . . . . .	56
1.6.3	Refinement of Algorithm . . . . .	58
1.6.4	Instantiation with Actual Data Structures . . . . .	60
1.6.5	Main Correctness Theorem . . . . .	62
1.6.6	Code Generation and Test . . . . .	62

<b>2</b>	<b>Dijkstra's Shortest Path Algorithm</b>	<b>64</b>
2.1	Weighted Directed Graphs . . . . .	64
2.1.1	Paths . . . . .	65
2.1.2	Distance . . . . .	66
2.2	Abstract Datatype for Weighted Directed Graphs . . . . .	67
2.2.1	Constructing Weighted Graphs from Lists . . . . .	68
2.3	Abstract Dijkstra Algorithm . . . . .	70
2.3.1	Abstract Algorithm . . . . .	70
2.3.2	Refinement by Priority Map and Map . . . . .	74
2.4	Weighted Digraph Implementation by Adjacency Map . . . . .	78
2.5	Implementation of Dijkstra's Algorithm . . . . .	79
2.5.1	Implementation using ADT Interfaces . . . . .	80
2.5.2	Instantiation of ADTs and Code Generation . . . . .	83
2.5.3	Combination with Graph Parser . . . . .	85

# Chapter 1

## Prim's Minimum Spanning Tree Algorithm

Prim's algorithm [4] is a classical algorithm to find a minimum spanning tree of an undirected graph. In this section we describe our formalization of Prim's algorithm, roughly following the presentation of Cormen et al. [1].

Our approach features stepwise refinement. We start by a generic MST algorithm (Section 1.4.1) that covers both Prim's and Kruskal's algorithms. It maintains a subgraph  $A$  of an MST. Initially,  $A$  contains no edges and only the root node. In each iteration, the algorithm adds a new edge to  $A$ , maintaining the property that  $A$  is a subgraph of an MST. In a next refinement step, we only add edges that are adjacent to the current  $A$ , thus maintaining the invariant that  $A$  is always a tree (Section 1.4.2). Next, we show how to use a priority queue to efficiently determine a next edge to be added (Section 1.4.3), and implement the necessary update of the priority queue using a foreach-loop (Section 1.4.4). Finally we parameterize our algorithm over ADTs for graphs, maps, and priority queues (Section 1.6.1), instantiate these with actual data structures (Section 1.6.4), and extract executable ML code (Section 1.6.6).

The advantage of this stepwise refinement approach is that the proof obligations of each step are mostly independent from the other steps. This modularization greatly helps to keep the proof manageable. Moreover, the steps also correspond to a natural split of the ideas behind Prim's algorithm: The same structuring is also done in the presentation of Cormen et al. [1], though not as detailed as ours.

### 1.1 Undirected Graphs

```
theory Undirected-Graph
imports
  Common
```

begin

### 1.1.1 Nodes and Edges

**typedef** *'v ugraph*  
= { ( $V::'v \text{ set}$ ,  $E$ ).  $E \subseteq V \times V \wedge \text{finite } V \wedge \text{sym } E \wedge \text{irrefl } E$  }  
**unfolding** *sym-def irrefl-def* **by** *blast*

**setup-lifting** *type-definition-ugraph*

**lift-definition** *nodes-internal* :: *'v ugraph*  $\Rightarrow$  *'v set* **is** *fst* .

**lift-definition** *edges-internal* :: *'v ugraph*  $\Rightarrow$  (*'v*  $\times$  *'v*) *set* **is** *snd* .

**lift-definition** *graph-internal* :: *'v set*  $\Rightarrow$  (*'v*  $\times$  *'v*) *set*  $\Rightarrow$  *'v ugraph*

**is**  $\lambda V E$ . *if* *finite*  $V \wedge \text{finite } E$  *then* ( $V \cup \text{fst}'E \cup \text{snd}'E$ ,  $(E \cup E^{-1}) - \text{Id}$ ) *else* ( $\{\}, \{\}$ )  
**by** (*auto simp: sym-def irrefl-def; force*)

**definition** *nodes* :: *'v ugraph*  $\Rightarrow$  *'v set*

**where** *nodes* = *nodes-internal*

**definition** *edges* :: *'v ugraph*  $\Rightarrow$  (*'v*  $\times$  *'v*) *set*

**where** *edges* = *edges-internal*

**definition** *graph* :: *'v set*  $\Rightarrow$  (*'v*  $\times$  *'v*) *set*  $\Rightarrow$  *'v ugraph*

**where** *graph* = *graph-internal*

**lemma** *edges-subset*: *edges*  $g \subseteq \text{nodes } g \times \text{nodes } g$

**unfolding** *edges-def nodes-def* **by** *transfer auto*

**lemma** *nodes-finite*[*simp, intro!*]: *finite* (*nodes*  $g$ )

**unfolding** *edges-def nodes-def* **by** *transfer auto*

**lemma** *edges-sym*: *sym* (*edges*  $g$ )

**unfolding** *edges-def nodes-def* **by** *transfer auto*

**lemma** *edges-irrefl*: *irrefl* (*edges*  $g$ )

**unfolding** *edges-def nodes-def* **by** *transfer auto*

**lemma** *nodes-graph*:  $\llbracket \text{finite } V; \text{finite } E \rrbracket \Longrightarrow \text{nodes } (\text{graph } V E) = V \cup \text{fst}'E \cup \text{snd}'E$

**unfolding** *edges-def nodes-def graph-def* **by** *transfer auto*

**lemma** *edges-graph*:  $\llbracket \text{finite } V; \text{finite } E \rrbracket \Longrightarrow \text{edges } (\text{graph } V E) = (E \cup E^{-1}) - \text{Id}$

**unfolding** *edges-def nodes-def graph-def* **by** *transfer auto*

**lemmas** *graph-accs* = *nodes-graph edges-graph*

**lemma** *nodes-edges-graph-presentation*:  $\llbracket \text{finite } V; \text{finite } E \rrbracket$

$\Longrightarrow \text{nodes } (\text{graph } V E) = V \cup \text{fst}'E \cup \text{snd}'E \wedge \text{edges } (\text{graph } V E) = E \cup E^{-1}$   
 $- \text{Id}$

**by** (*simp add: graph-accs*)

**lemma** *graph-eq[simp]*:  $\text{graph } (\text{nodes } g) (\text{edges } g) = g$   
**unfolding** *edges-def nodes-def graph-def*  
**apply** *transfer*  
**unfolding** *sym-def irrefl-def*  
**apply** (*clarsimp split: prod.splits*)  
**by** (*fastforce simp: finite-subset*)

**lemma** *edges-finite[simp, intro!]*:  $\text{finite } (\text{edges } g)$   
**using** *edges-subset finite-subset by fastforce*

**lemma** *graph-cases[cases type]*: **obtains**  $V E$   
**where**  $g = \text{graph } V E \text{ finite } V \text{ finite } E E \subseteq V \times V \text{ sym } E \text{ irrefl } E$   
**proof** –  
**show** *?thesis*  
**apply** (*rule that[of nodes g edges g]*)  
**using** *edges-subset edges-sym edges-irrefl[of g]*  
**by** *auto*  
**qed**

**lemma** *graph-eq-iff*:  $g = g' \iff \text{nodes } g = \text{nodes } g' \wedge \text{edges } g = \text{edges } g'$   
**unfolding** *edges-def nodes-def graph-def by transfer auto*

**lemma** *edges-sym'*:  $(u, v) \in \text{edges } g \implies (v, u) \in \text{edges } g$  **using** *edges-sym*  
**by** (*blast intro: symD*)

**lemma** *edges-irrefl'[simp, intro!]*:  $(u, u) \notin \text{edges } g$   
**by** (*meson edges-irrefl irrefl-def*)

**lemma** *edges-irreflI[simp, intro]*:  $(u, v) \in \text{edges } g \implies u \neq v$  **by** *auto*

**lemma** *edgesT-diff-sng-inv-eq[simp]*:  
 $(\text{edges } T - \{(x, y), (y, x)\})^{-1} = \text{edges } T - \{(x, y), (y, x)\}$   
**using** *edges-sym' by fast*

**lemma** *nodesI[simp, intro]*: **assumes**  $(u, v) \in \text{edges } g$  **shows**  $u \in \text{nodes } g \ v \in \text{nodes } g$   
**using** *assms edges-subset by auto*

**lemma** *split-edges-sym*:  $\exists E. E \cap E^{-1} = \{\}$   $\wedge \text{edges } g = E \cup E^{-1}$   
**using** *split-sym-rel[OF edges-sym edges-irrefl, of g] by metis*

### 1.1.2 Connectedness Relation

**lemma** *rtrancl-edges-sym'*:  $(u, v) \in (\text{edges } g)^* \implies (v, u) \in (\text{edges } g)^*$   
**by** (*simp add: edges-sym symD sym-rtrancl*)

**lemma** *trancl-edges-subset*:  $(\text{edges } g)^+ \subseteq \text{nodes } g \times \text{nodes } g$   
**by** (*simp add: edges-subset trancl-subset-Sigma*)

**lemma** *find-crossing-edge*:  
**assumes**  $(u,v) \in E^*$   $u \in V$   $v \notin V$   
**obtains**  $u' v'$  **where**  $(u',v') \in E \cap V \times -V$   
**using** *assms* **apply** (*induction rule: converse-rtrancl-induct*)  
**by** *auto*

### 1.1.3 Constructing Graphs

**definition** *graph-empty*  $\equiv$  *graph*  $\{\}$   $\{\}$   
**definition** *ins-node*  $v$   $g \equiv$  *graph* (*insert*  $v$  (*nodes*  $g$ )) (*edges*  $g$ )  
**definition** *ins-edge*  $e$   $g \equiv$  *graph* (*nodes*  $g$ ) (*insert*  $e$  (*edges*  $g$ ))  
**definition** *graph-join*  $g_1$   $g_2 \equiv$  *graph* (*nodes*  $g_1 \cup$  *nodes*  $g_2$ ) (*edges*  $g_1 \cup$  *edges*  $g_2$ )  
**definition** *restrict-nodes*  $g$   $V \equiv$  *graph* (*nodes*  $g \cap V$ ) (*edges*  $g \cap V \times V$ )  
**definition** *restrict-edges*  $g$   $E \equiv$  *graph* (*nodes*  $g$ ) (*edges*  $g \cap (E \cup E^{-1})$ )

**definition** *nodes-edges-consistent*  $V$   $E \equiv$  *finite*  $V \wedge$  *irrefl*  $E \wedge$  *sym*  $E \wedge E \subseteq V \times V$

**lemma** [*simp*]:  
**assumes** *nodes-edges-consistent*  $V$   $E$   
**shows** *nodes-graph'*: *nodes* (*graph*  $V$   $E$ ) =  $V$  (**is** ? $G1$ )  
**and** *edges-graph'*: *edges* (*graph*  $V$   $E$ ) =  $E$  (**is** ? $G2$ )  
**proof** –  
**from** *assms* **have** [*simp*]: *finite*  $E$  **unfolding** *nodes-edges-consistent-def*  
**by** (*meson finite-SigmaI rev-finite-subset*)  
  
**show** ? $G1$  ? $G2$  **using** *assms*  
**by** (*auto simp: nodes-edges-consistent-def nodes-graph edges-graph irrefl-def*)

qed

**lemma** *nec-empty*[*simp*]: *nodes-edges-consistent*  $\{\}$   $\{\}$   
**by** (*auto simp: nodes-edges-consistent-def irrefl-def sym-def*)

**lemma** *graph-empty-accs*[*simp*]:  
*nodes* *graph-empty* =  $\{\}$   
*edges* *graph-empty* =  $\{\}$   
**unfolding** *graph-empty-def* **by** (*auto*)

**lemma** *graph-empty*[*simp*]: *graph*  $\{\}$   $\{\}$  = *graph-empty*  
**by** (*simp add: graph-empty-def*)

**lemma** *nodes-empty-iff-empty*[*simp*]:  
*nodes*  $G = \{\} \longleftrightarrow G =$  *graph*  $\{\}$   $\{\}$   
 $\{\} =$  *nodes*  $G \longleftrightarrow G =$  *graph-empty*  
**using** *edges-subset*  
**by** (*auto simp: graph-eq-iff*)

**lemma** *nodes-ins-nodes*[simp]:  $\text{nodes } (\text{ins-node } v \ g) = \text{insert } v \ (\text{nodes } g)$   
**and** *edges-ins-nodes*[simp]:  $\text{edges } (\text{ins-node } v \ g) = \text{edges } g$   
**unfolding** *ins-node-def* **by** (*auto simp: graph-accs edges-sym'*)

**lemma** *nodes-ins-edge*[simp]:  $\text{nodes } (\text{ins-edge } e \ g) = \{\text{fst } e, \text{snd } e\} \cup \text{nodes } g$   
**and** *edges-ins-edge*:  
 $\text{edges } (\text{ins-edge } e \ g)$   
 $= (\text{if } \text{fst } e = \text{snd } e \ \text{then } \text{edges } g \ \text{else } \{e, \text{prod.swap } e\} \cup (\text{edges } g))$   
**unfolding** *ins-edge-def*  
**apply** (*all <cases e>*)  
**by** (*auto simp: graph-accs dest: edges-sym'*)

**lemma** *edges-ins-edge'*[simp]:  
 $u \neq v \implies \text{edges } (\text{ins-edge } (u, v) \ g) = \{(u, v), (v, u)\} \cup \text{edges } g$   
**by** (*auto simp: edges-ins-edge*)

**lemma** *edges-ins-edge-ss*:  $\text{edges } g \subseteq \text{edges } (\text{ins-edge } e \ g)$   
**by** (*auto simp: edges-ins-edge*)

**lemma** *nodes-join*[simp]:  $\text{nodes } (\text{graph-join } g_1 \ g_2) = \text{nodes } g_1 \cup \text{nodes } g_2$   
**and** *edges-join*[simp]:  $\text{edges } (\text{graph-join } g_1 \ g_2) = \text{edges } g_1 \cup \text{edges } g_2$   
**unfolding** *graph-join-def*  
**by** (*auto simp: graph-accs dest: edges-sym'*)

**lemma** *nodes-restrict-nodes*[simp]:  $\text{nodes } (\text{restrict-nodes } g \ V) = \text{nodes } g \cap V$   
**and** *edges-restrict-nodes*[simp]:  $\text{edges } (\text{restrict-nodes } g \ V) = \text{edges } g \cap V \times V$   
**unfolding** *restrict-nodes-def*  
**by** (*auto simp: graph-accs dest: edges-sym'*)

**lemma** *nodes-restrict-edges*[simp]:  $\text{nodes } (\text{restrict-edges } g \ E) = \text{nodes } g$   
**and** *edges-restrict-edges*[simp]:  $\text{edges } (\text{restrict-edges } g \ E) = \text{edges } g \cap (E \cup E^{-1})$   
**unfolding** *restrict-edges-def*  
**by** (*auto simp: graph-accs dest: edges-sym'*)

**lemma** *unrestrict-edges*:  $\text{edges } (\text{restrict-edges } g \ E) \subseteq \text{edges } g$  **by** *auto*  
**lemma** *unrestrictn-edges*:  $\text{edges } (\text{restrict-nodes } g \ V) \subseteq \text{edges } g$  **by** *auto*

**lemma** *unrestrict-nodes*:  $\text{nodes } (\text{restrict-edges } g \ E) \subseteq \text{nodes } g$  **by** *auto*

#### 1.1.4 Paths

**fun** *path* **where**

$\text{path } g \ u \ [] \ v \longleftrightarrow u = v$   
 $| \text{path } g \ u \ (e \# ps) \ w \longleftrightarrow (\exists v. e = (u, v) \wedge e \in \text{edges } g \wedge \text{path } g \ v \ ps \ w)$

**lemma** *path-emptyI*[intro!]:  $\text{path } g \ u \ [] \ u$  **by** *auto*



**lemma** *path-append*[simp]:  
 $path\ g\ u\ (p1 @ p2)\ w \longleftrightarrow (\exists v. path\ g\ u\ p1\ v \wedge path\ g\ v\ p2\ w)$   
**by** (*induction p1 arbitrary: u*) *auto*

**lemma** *path-transs1*[trans]:  
 $path\ g\ u\ p\ v \implies (v,w) \in edges\ g \implies path\ g\ u\ (p @ [(v,w)])\ w$   
 $(u,v) \in edges\ g \implies path\ g\ v\ p\ w \implies path\ g\ u\ ((u,v) \# p)\ w$   
 $path\ g\ u\ p1\ v \implies path\ g\ v\ p2\ w \implies path\ g\ u\ (p1 @ p2)\ w$   
**by** *auto*

**lemma** *path-graph-empty*[simp]:  $path\ graph\ empty\ u\ p\ v \longleftrightarrow v = u \wedge p = []$   
**by** (*cases p*) *auto*

**abbreviation** *revp*  $p \equiv rev\ (map\ prod.swap\ p)$   
**lemma** *revp-alt*:  $revp\ p = rev\ (map\ (\lambda(u,v). (v,u))\ p)$  **by** *auto*

**lemma** *path-rev*[simp]:  $path\ g\ u\ (revp\ p)\ v \longleftrightarrow path\ g\ v\ p\ u$   
**by** (*induction p arbitrary: v*) (*auto dest: edges-sym'*)

**lemma** *path-rev-sym*[sym]:  $path\ g\ v\ p\ u \implies path\ g\ u\ (revp\ p)\ v$  **by** *simp*

**lemma** *path-transs2*[trans]:  
 $path\ g\ u\ p\ v \implies (w,v) \in edges\ g \implies path\ g\ u\ (p @ [(v,w)])\ w$   
 $(v,u) \in edges\ g \implies path\ g\ v\ p\ w \implies path\ g\ u\ ((u,v) \# p)\ w$   
 $path\ g\ u\ p1\ v \implies path\ g\ w\ p2\ v \implies path\ g\ u\ (p1 @ revp\ p2)\ w$   
**by** (*auto dest: edges-sym'*)

**lemma** *path-edges*:  $path\ g\ u\ p\ v \implies set\ p \subseteq edges\ g$   
**by** (*induction p arbitrary: u*) *auto*

**lemma** *path-graph-cong*:  
 $[[path\ g_1\ u\ p\ v; set\ p \subseteq edges\ g_1 \implies set\ p \subseteq edges\ g_2]] \implies path\ g_2\ u\ p\ v$   
**apply** (*frule path-edges; simp*)  
**apply** (*induction p arbitrary: u*)  
**by** *auto*

**lemma** *path-endpoints*:  
**assumes**  $path\ g\ u\ p\ v\ p \neq []$  **shows**  $u \in nodes\ g\ v \in nodes\ g$   
**subgoal using** *assms* **by** (*cases p*) (*auto intro: nodesI*)  
**subgoal using** *assms* **by** (*cases p rule: rev-cases*) (*auto intro: nodesI*)  
**done**

**lemma** *path-mono*:  $edges\ g \subseteq edges\ g' \implies path\ g\ u\ p\ v \implies path\ g'\ u\ p\ v$   
**by** (*meson path-edges path-graph-cong subset-trans*)

**lemmas** *unrestrict-path* = *path-mono*[*OF unrestrict-edges*]  
**lemmas** *unrestrictn-path* = *path-mono*[*OF unrestrictn-edges*]

**lemma** *unrestrict-path-edges*: *path (restrict-edges g E) u p v*  $\implies$  *path g u p v*  
**by** (*induction p arbitrary: u*) *auto*

**lemma** *unrestrict-path-nodes*: *path (restrict-nodes g E) u p v*  $\implies$  *path g u p v*  
**by** (*induction p arbitrary: u*) *auto*

## Paths and Connectedness

**lemma** *rtrancl-edges-iff-path*:  $(u,v) \in (\text{edges } g)^* \iff (\exists p. \text{path } g \ u \ p \ v)$   
**apply** *rule*  
**subgoal**  
**apply** (*induction rule: converse-rtrancl-induct*)  
**by** (*auto dest: path-transs1*)  
**apply** *clarify*  
**subgoal for** *p* **by** (*induction p arbitrary: u; force*)  
**done**

**lemma** *rtrancl-edges-pathE*:  
**assumes**  $(u,v) \in (\text{edges } g)^*$  **obtains** *p* **where** *path g u p v*  
**using** *assms* **by** (*auto simp: rtrancl-edges-iff-path*)

**lemma** *path-rtrancl-edgesD*: *path g u p v*  $\implies (u,v) \in (\text{edges } g)^*$   
**by** (*auto simp: rtrancl-edges-iff-path*)

## Simple Paths

**definition** *uedge*  $\equiv \lambda(a,b). \{a,b\}$

**definition** *simple p*  $\equiv \text{distinct } (\text{map } \text{uedge } p)$

**lemma** *in-uedge-conv[simp]*:  $x \in \text{uedge } (u,v) \iff x=u \vee x=v$   
**by** (*auto simp: uedge-def*)

**lemma** *uedge-eq-iff*:  $\text{uedge } (a,b) = \text{uedge } (c,d) \iff a=c \wedge b=d \vee a=d \wedge b=c$   
**by** (*auto simp: uedge-def doubleton-eq-iff*)

**lemma** *uedge-degen[simp]*:  $\text{uedge } (a,a) = \{a\}$   
**by** (*auto simp: uedge-def*)

**lemma** *uedge-in-set-eq*:  $\text{uedge } (u, v) \in \text{uedge } 'S \iff (u,v) \in S \vee (v,u) \in S$   
**by** (*auto simp: uedge-def doubleton-eq-iff*)

**lemma** *uedge-commute*:  $\text{uedge } (a,b) = \text{uedge } (b,a)$  **by** *auto*

**lemma** *simple-empty[simp]*: *simple*  $\square$   
**by** (*auto simp: simple-def*)

**lemma** *simple-cons*[simp]:  $simple (e\#p) \longleftrightarrow uedge\ e \notin uedge\ 'set\ p \wedge simple\ p$   
**by** (*auto simp: simple-def*)

**lemma** *simple-append*[simp]:  $simple (p_1 @ p_2)$   
 $\longleftrightarrow simple\ p_1 \wedge simple\ p_2 \wedge uedge\ 'set\ p_1 \cap uedge\ 'set\ p_2 = \{\}$   
**by** (*auto simp: simple-def*)

**lemma** *simplify-pathD*:  
 $path\ g\ u\ p\ v \implies \exists p'. path\ g\ u\ p'\ v \wedge simple\ p' \wedge set\ p' \subseteq set\ p$   
**proof** (*induction p arbitrary: u v rule: length-induct*)  
**case** *A*: ( $1\ p$ )  
**then show** *?case* **proof** (*cases simple p*)  
**assume** *simple p with A.prem*s **show** *?case* **by** *blast*  
**next**  
**assume**  $\neg simple\ p$   
**then consider**  $p_1\ a\ b\ p_2\ p_3$  **where**  $p = p_1 @ [(a,b)] @ p_2 @ [(a,b)] @ p_3$   
 $| p_1\ a\ b\ p_2\ p_3$  **where**  $p = p_1 @ [(a,b)] @ p_2 @ [(b,a)] @ p_3$   
**by** (*auto*  
*simp: simple-def map-eq-append-conv uedge-eq-iff*  
*dest!: not-distinct-decomp*)  
**then obtain**  $p'$  **where**  $path\ g\ u\ p'\ v$   $length\ p' < length\ p$   $set\ p' \subseteq set\ p$   
**proof** *cases*  
**case** [simp]:  $1$   
**from** *A.prem*s **have**  $path\ g\ u\ (p_1 @ [(a,b)] @ p_3)\ v$  **by** *auto*  
**from** *that*[*OF this*] **show** *?thesis* **by** *auto*  
**next**  
**case** [simp]:  $2$   
**from** *A.prem*s **have**  $path\ g\ u\ (p_1 @ p_3)\ v$  **by** *auto*  
**from** *that*[*OF this*] **show** *?thesis* **by** *auto*  
**qed**  
**with** *A.IH* **show** *?thesis* **by** *blast*  
**qed**  
**qed**

**lemma** *simplify-pathE*:  
**assumes**  $path\ g\ u\ p\ v$   
**obtains**  $p'$  **where**  $path\ g\ u\ p'\ v$   $simple\ p'$   $set\ p' \subseteq set\ p$   
**using** *assms* **by** (*auto dest: simplify-pathD*)

## Splitting Paths

**lemma** *find-crossing-edge-on-path*:  
**assumes**  $path\ g\ u\ p\ v$   $\neg P\ u\ P\ v$   
**obtains**  $u'\ v'$  **where**  $(u',v') \in set\ p$   $\neg P\ u'\ P\ v'$   
**using** *assms* **by** (*induction p arbitrary: u*) *auto*

**lemma** *find-crossing-edges-on-path*:

**assumes**  $P$ : *path*  $g$   $u$   $p$   $v$  **and**  $P$   $u$   $P$   $v$   
**obtains**  $\forall (u,v) \in \text{set } p. P$   $u$   $\wedge$   $P$   $v$   
    |  $u_1$   $v_1$   $v_2$   $u_2$   $p_1$   $p_2$   $p_3$   
    **where**  $p = p_1 @ [(u_1, v_1)] @ p_2 @ [(u_2, v_2)] @ p_3$   $P$   $u_1$   $\neg P$   $v_1$   $\neg P$   $u_2$   $P$   $v_2$   
**proof** (*cases*  $\forall (u,v) \in \text{set } p. P$   $u$   $\wedge$   $P$   $v$ )  
    **case** *True* **with that show** *?thesis* **by** *blast*  
**next**  
    **case** *False*  
    **with**  $P$   $\langle P$   $u \rangle$  **have**  $\exists (u_1, v_1) \in \text{set } p. P$   $u_1$   $\wedge$   $\neg P$   $v_1$   
    **apply** *clarsimp* **apply** (*induction*  $p$  *arbitrary*:  $u$ ) **by** *auto*  
    **then obtain**  $u_1$   $v_1$  **where**  $(u_1, v_1) \in \text{set } p$  **and**  $PRED1$ :  $P$   $u_1$   $\neg P$   $v_1$  **by** *blast*  
    **then obtain**  $p_1$   $p_{23}$  **where** [*simp*]:  $p = p_1 @ [(u_1, v_1)] @ p_{23}$   
    **by** (*auto simp: in-set-conv-decomp*)  
    **with**  $P$  **have** *path*  $g$   $v_1$   $p_{23}$   $v$  **by** *auto*  
    **from** *find-crossing-edge-on-path* [**where**  $P = P$ , *OF* *this*  $\langle \neg P$   $v_1 \rangle$   $\langle P$   $v \rangle$ ] **obtain**  $u_2$   
     $v_2$   
    **where**  $(u_2, v_2) \in \text{set } p_{23}$   $\neg P$   $u_2$   $P$   $v_2$  .  
    **then show** *thesis* **using**  $PRED1$   
    **by** (*auto simp: in-set-conv-decomp intro: that*)  
**qed**

**lemma** *find-crossing-edge-rtrancl*:  
**assumes**  $(u,v) \in (\text{edges } g)^*$   $\neg P$   $u$   $P$   $v$   
**obtains**  $u'$   $v'$  **where**  $(u', v') \in \text{edges } g$   $\neg P$   $u'$   $P$   $v'$   
**using** *assms*  
**by** (*metis converse-rtrancl-induct*)

**lemma** *path-change*:  
**assumes**  $u \in S$   $v \notin S$  *path*  $g$   $u$   $p$   $v$  *simple*  $p$   
**obtains**  $x$   $y$   $p1$   $p2$  **where**  
     $(x,y) \in \text{set } p$   $x \in S$   $y \notin S$   
    *path* (*restrict-edges*  $g$  ( $-\{(x,y), (y,x)\}$ ))  $u$   $p1$   $x$   
    *path* (*restrict-edges*  $g$  ( $-\{(x,y), (y,x)\}$ ))  $y$   $p2$   $v$   
**proof** –  
**from** *find-crossing-edge-on-path* [**where**  $P = \lambda x. x \notin S$ ] *assms* **obtain**  $x$   $y$  **where**  
    1:  $(x,y) \in \text{set } p$   $x \in S$   $y \notin S$  **by** *blast*  
**then obtain**  $p1$   $p2$  **where** [*simp*]:  $p = p1 @ [(x,y)] @ p2$   
    **by** (*auto simp: in-set-conv-decomp*)

**let**  $?g' = \text{restrict-edges } g$  ( $-\{(x,y), (y,x)\}$ )

**from**  $\langle \text{path } g$   $u$   $p$   $v \rangle$  **have**  $P1$ : *path*  $g$   $u$   $p1$   $x$  **and**  $P2$ : *path*  $g$   $y$   $p2$   $v$  **by** *auto*  
**from**  $\langle \text{simple } p \rangle$   
    **have** *uedge*  $(x,y) \notin \text{set } (\text{map } \text{uedge } p1)$  *uedge*  $(x,y) \notin \text{set } (\text{map } \text{uedge } p2)$   
    **by** *auto*  
**then have** *path*  $?g'$   $u$   $p1$   $x$  *path*  $?g'$   $y$   $p2$   $v$   
    **using** *path-graph-cong* [*OF*  $P1$ , *of*  $?g'$ ] *path-graph-cong* [*OF*  $P2$ , *of*  $?g'$ ]  
    **by** (*auto simp: uedge-in-set-eq*)

with 1 show ?thesis by (blast intro: that)  
qed

### 1.1.5 Cycles

**definition** *cycle-free*  $g \equiv \nexists p u. p \neq [] \wedge \text{simple } p \wedge \text{path } g u p u$

**lemma** *cycle-free-alt-in-nodes*:

*cycle-free*  $g \equiv \nexists p u. p \neq [] \wedge u \in \text{nodes } g \wedge \text{simple } p \wedge \text{path } g u p u$   
by (smt *cycle-free-def path-endpoints(2)*)

**lemma** *cycle-freeI*:

assumes  $\bigwedge p u. \llbracket \text{path } g u p u; p \neq []; \text{simple } p \rrbracket \implies \text{False}$   
shows *cycle-free*  $g$   
using *assms* **unfolding** *cycle-free-def* **by** *auto*

**lemma** *cycle-freeD*:

assumes *cycle-free*  $g$   $\text{path } g u p u p \neq []$  *simple*  $p$   
shows *False*  
using *assms* **unfolding** *cycle-free-def* **by** *auto*

**lemma** *cycle-free-antimono*:  $\text{edges } g \subseteq \text{edges } g' \implies \text{cycle-free } g' \implies \text{cycle-free } g$

**unfolding** *cycle-free-def*  
**by** (*auto dest: path-mono*)

**lemma** *cycle-free-empty[simp]*: *cycle-free* *graph-empty*

**unfolding** *cycle-free-def* **by** *auto*

**lemma** *cycle-free-no-edges*:  $\text{edges } g = \{\} \implies \text{cycle-free } g$

**by** (*rule cycle-freeI*) (*auto simp: neq-Nil-conv*)

**lemma** *simple-path-cycle-free-unique*:

assumes *CF*: *cycle-free*  $g$   
assumes *P*:  $\text{path } g u p v \text{ path } g u p' v \text{ simple } p \text{ simple } p'$   
shows  $p = p'$   
using *P*

**proof** (*induction*  $p$  *arbitrary*:  $u p'$ )

case *Nil*

then show ?case **using** *cycle-freeD[OF CF]* **by** *auto*

**next**

case (*Cons*  $e p$ )

**note**  $CF = \text{cycle-freeD}[OF CF]$

**from** *Cons.prem*s **obtain**  $u'$  **where**

$[simp]: e = (u, u')$

**and** *P'*:  $(u, u') \notin \text{set } p \ (u', u) \notin \text{set } p \ (u, u') \in \text{edges } g$

**by** (*auto simp: uedge-in-set-eq*)

**with** *Cons.prem*s **obtain**  $sp_1$  **where**

*SP1*:  $path\ g\ u\ ((u,u')\#sp_1)\ v\ simple\ ((u,u')\#sp_1)$

**by** *blast*

**from** *Cons.prem*s **obtain**  $u''\ p''$  **where**

[*simp*]:  $p' = (u,u'')\#p''$

**and**  $P''$ :  $(u,u'')\notin set\ p''\ (u'',u)\notin set\ p''\ (u,u'')\in edges\ g$

**apply** (*cases*  $p'$ )

**subgoal** **by** *auto* (*metis* *Cons.prem*s(1) *Cons.prem*s(3) *CF list.distinct*(1))

**by** (*auto simp: uedge-in-set-eq*)

**with** *Cons.prem*s **obtain**  $sp_2$  **where**

*SP2*:  $path\ g\ u\ ((u,u'')\#sp_2)\ v\ simple\ ((u,u'')\#sp_2)$

**by** *blast*

**have**  $u''=u'$  **proof** (*rule ccontr*)

**assume** [*simp, symmetric, simp*]:  $u''\neq u'$

**have** *AUX1*:  $(u,x)\notin set\ sp_1$  **for**  $x$

**proof**

**assume**  $(u, x) \in set\ sp_1$

**with** *SP1* **obtain**  $sp'$  **where**  $path\ g\ u\ ((u,u')\#sp')\ u$  **and**  $simple\ ((u,u')\#sp')$

**by** (*clarsimp simp: in-set-conv-decomp; blast*)

**with** *CF* **show** *False* **by** *blast*

**qed**

**have** *AUX2*:  $(x,u)\notin set\ sp_1$  **for**  $x$

**proof**

**assume**  $(x, u) \in set\ sp_1$

**with** *SP1* **obtain**  $sp'$  **where**  $path\ g\ u\ ((u,u')\#sp')\ u$  **and**  $simple\ ((u,u')\#sp')$

**apply** (*clarsimp simp: in-set-conv-decomp*)

**by** (*metis* *Cons.prem*s(1) *Cons.prem*s(3) *Un-iff*)

*AUX1*  $\langle e = (u, u') \rangle insert-iff\ list.simps(15)$

*path.elims*(2) *path.simps*(2) *prod.sel*(2) *set-append simple-cons*)

**with** *CF* **show** *False* **by** *blast*

**qed**

**have** *AUX3*:  $(u,x)\notin set\ sp_2$  **for**  $x$

**proof**

**assume**  $(u, x) \in set\ sp_2$

**then** **obtain**  $sp'\ sp''$  **where** [*simp*]:  $sp_2 = sp'\@[u,x]\@sp''$

**by** (*auto simp: in-set-conv-decomp*)

**from** *SP2* **have**  $path\ g\ u\ ((u,u'')\#sp')\ u$   $simple\ ((u,u'')\#sp')$  **by** *auto*

**with** *CF* **show** *False* **by** *blast*

**qed**

**have** *AUX4*:  $(x,u)\notin set\ sp_2$  **for**  $x$

**proof**

**assume**  $(x, u) \in \text{set } sp_2$   
**then obtain**  $sp' sp''$  **where**  $[simp]: sp_2 = sp' @ [(x, u)] @ sp''$   
**by**  $(\text{auto } simp: \text{in-set-conv-decomp})$   
**from**  $SP2$   
**have**  $\text{path } g \ u \ ((u, u'') \# sp' @ [(x, u)]) \ u \ \text{simple } ((u, u'') \# sp' @ [(x, u)])$   
**by**  $\text{auto}$   
**with**  $CF$  **show**  $False$  **by**  $\text{blast}$   
**qed**

**have**  $[simp]: \text{set } (\text{revp } p) = (\text{set } p)^{-1}$  **by**  $\text{auto}$

**from**  $SP1 \ SP2$  **have**  $\text{path } g \ u' \ (sp_1 @ \text{revp } sp_2) \ u''$  **by**  $\text{auto}$   
**then obtain**  $sp$  **where**  
 $SP: \text{path } g \ u' \ sp \ u'' \ \text{simple } sp \ \text{set } sp \subseteq \text{set } sp_1 \cup \text{set } (\text{revp } sp_2)$   
**by**  $(\text{erule-tac } \text{simplify-pathE}) \ \text{auto}$   
**with**  $\langle (u, u') \in \text{edges } g \rangle \langle (u, u'') \in \text{edges } g \rangle$   
**have**  $\text{path } g \ u \ ((u, u') \# sp @ [(u'', u)]) \ u$   
**by**  $(\text{auto } \text{dest: } \text{edges-sym}' \ \text{simp: } \text{uedge-eq-iff})$   
**moreover**  
**from**  $SP \ SP1 \ SP2 \ AUX1 \ AUX2 \ AUX3 \ AUX4$  **have**  $\text{simple } (((u, u') \# sp @ [(u'', u)]))$   
**by**  $(\text{auto } 0 \ 3 \ \text{simp: } \text{uedge-eq-iff})$   
**ultimately show**  $False$  **using**  $CF$  **by**  $\text{blast}$   
**qed**

**with**  $\text{Cons.IH}[of \ u' \ p'] \ \text{Cons.premis}$  **show**  $?case$  **by**  $\text{simp}$   
**qed**

## Characterization by Removing Edge

**lemma**  $\text{cycle-free-alt: cycle-free } g$   
 $\longleftrightarrow (\forall e \in \text{edges } g. e \notin (\text{edges } (\text{restrict-edges } g \ (-\{e, \text{prod.swap } e\})))^*)$   
**apply**  $(\text{rule})$   
**apply**  $(\text{clarsimp } \text{simp } \text{del: } \text{edges-restrict-edges})$   
**subgoal premises**  $\text{prems}$  **for**  $u \ v$  **proof**  $-$   
**note**  $\text{edges-restrict-edges}[simp \ \text{del}]$   
**let**  $?rg = (\text{restrict-edges } g \ (-\{(u, v), (v, u)\}))$   
**from**  $\langle (u, v) \in (\text{edges } ?rg)^* \rangle$   
**obtain**  $p$  **where**  $P: \text{path } ?rg \ u \ p \ v$  **and**  $\text{simple } p$   
**by**  $(\text{auto } \text{simp: } \text{rtrancd-edges-iff-path } \text{elim: } \text{simplify-pathE})$   
**from**  $P$  **have**  $\text{path } g \ u \ p \ v$  **by**  $(\text{rule } \text{unrestrictd-path})$   
**also note**  $\langle (u, v) \in \text{edges } g \rangle$  **finally have**  $\text{path } g \ u \ (p @ [(v, u)]) \ u$  .  
**moreover from**  $\text{path-edges}[OF \ P]$  **have**  $\text{uedge } (u, v) \notin \text{set } (\text{map } \text{uedge } p)$   
**by**  $(\text{auto } \text{simp: } \text{uedge-eq-iff } \text{edges-restrict-edges})$   
**with**  $\langle \text{simple } p \rangle$  **have**  $\text{simple } (p @ [(v, u)])$   
**by**  $(\text{auto } \text{simp: } \text{uedge-eq-iff } \text{uedge-in-set-eq})$   
**ultimately show**  $?thesis$  **using**  $\langle \text{cycle-free } g \rangle$   
**unfolding**  $\text{cycle-free-def}$  **by**  $\text{blast}$   
**qed**  
**apply**  $(\text{clarsimp } \text{simp: } \text{cycle-free-def})$

**subgoal premises  $prems$  for  $p$   $u$  proof** –  
**from**  $\langle p \neq [] \rangle \langle path\ g\ u\ p\ u \rangle$  **obtain**  $v\ p'$  **where**  
 $[simp]: p = (u,v) \# p'$  **and**  $(u,v) \in edges\ g\ path\ g\ v\ p'\ u$   
**by**  $(cases\ p)\ auto$   
**from**  $\langle simple\ p \rangle$  **have**  $simple\ p'\ uedge\ (u,v) \notin set\ (map\ uedge\ p')$  **by**  $auto$   
**hence**  $(u,v) \notin set\ p'\ (v,u) \notin set\ p'$  **by**  $(auto\ simp:\ uedge\ in\ set\ eq)$   
**with**  $\langle path\ g\ v\ p'\ u \rangle$   
**have**  $path\ (restrict\ edges\ g\ (-\{(u,v),(v,u)\}))\ v\ p'\ u$  **(is path ?rg - -)**  
**by**  $(erule\ tac\ path\ graph\ cong)\ auto$   
  
**hence**  $(u,v) \in (edges\ ?rg)^*$   
**by**  $(meson\ path\ rev\ rtrancl\ edges\ iff\ path)$   
**with**  $prems(1)\ \langle (u,v) \in edges\ g \rangle$  **show**  $False$  **by**  $auto$   
**qed**  
**done**

**lemma**  $cycle\ free\ altI$ :  
**assumes**  $\bigwedge u\ v.\ \llbracket (u,v) \in edges\ g;\ (u,v) \in (edges\ g - \{(u,v),(v,u)\})^* \rrbracket \implies False$   
**shows**  $cycle\ free\ g$   
**unfolding**  $cycle\ free\ alt$  **using**  $assms$  **by**  $(force)$

**lemma**  $cycle\ free\ altD$ :  
**assumes**  $cycle\ free\ g$   
**assumes**  $(u,v) \in edges\ g$   
**shows**  $(u,v) \notin (edges\ g - \{(u,v),(v,u)\})^*$   
**using**  $assms$  **unfolding**  $cycle\ free\ alt$  **by**  $(auto)$

**lemma**  $remove\ redundant\ edge$ :  
**assumes**  $(u,v) \in (edges\ g - \{(u,v),(v,u)\})^*$   
**shows**  $(edges\ g - \{(u,v),(v,u)\})^* = (edges\ g)^*$  **(is ?E'\* = -)**  
**proof**  
**show**  $?E'^* \subseteq (edges\ g)^*$   
**by**  $(simp\ add:\ Diff\ subset\ rtrancl\ mono)$   
**next**  
**show**  $(edges\ g)^* \subseteq ?E'^*$   
**proof**  $clarify$   
**fix**  $a\ b$  **assume**  $(a,b) \in (edges\ g)^*$  **then**  
**show**  $(a,b) \in ?E'^*$   
**proof**  $induction$   
**case**  $base$   
**then** **show**  $?case$  **by**  $simp$   
**next**  
**case**  $(step\ b\ c)$   
**then** **show**  $?case$   
**proof**  $(cases\ (b,c) \in \{(u,v),(v,u)\})$   
**case**  $True$



```

have SYME: sym (?E*)
  apply (rule sym-rtrancl)
  using edges-sym[of g]
  by (auto simp: sym-def)
with step.IH assms have
  IH': (b,a) ∈ ?E*
  by (auto intro: symD)

from True show ?thesis apply safe
  subgoal using assms step.IH by simp
  subgoal using assms IH' apply (rule-tac symD[OF SYME]) by simp
  done

next
  case False
  then show ?thesis
    by (meson DiffI rtrancl.rtrancl-into-rtrancl step.IH step.hyps(2))
qed

  qed
qed
qed

```

### 1.1.6 Connected Graphs

**definition** *connected*

**where** *connected g* ≡ *nodes g* × *nodes g* ⊆ (*edges g*)<sup>\*</sup>

**lemma** *connectedI*[*intro?*]:

**assumes**  $\bigwedge u v. \llbracket u \in \text{nodes } g; v \in \text{nodes } g \rrbracket \implies (u,v) \in (\text{edges } g)^*$

**shows** *connected g*

**using** *assms* **unfolding** *connected-def* **by** *auto*

**lemma** *connectedD*[*intro?*]:

**assumes** *connected g u* ∈ *nodes g v* ∈ *nodes g*

**shows** (u,v) ∈ (*edges g*)<sup>\*</sup>

**using** *assms* **unfolding** *connected-def* **by** *auto*

**lemma** *connected-empty*[*simp*]: *connected graph-empty*

**unfolding** *connected-def* **by** *auto*

### 1.1.7 Component Containing Node

**definition** *reachable-nodes g r* ≡ (*edges g*)<sup>\*</sup> “{r}

**definition** *component-of g r*

≡ *ins-node r* (*restrict-nodes g* (*reachable-nodes g r*))

**lemma** *reachable-nodes-refl*[*simp, intro!*]: *r* ∈ *reachable-nodes g r*

**by** (*auto simp: reachable-nodes-def*)

**lemma** *reachable-nodes-step*:  
*edges g* “ *reachable-nodes g r*  $\subseteq$  *reachable-nodes g r*  
**by** (*auto simp: reachable-nodes-def*)

**lemma** *reachable-nodes-steps*:  
(*edges g*)<sup>\*</sup> “ *reachable-nodes g r*  $\subseteq$  *reachable-nodes g r*  
**by** (*auto simp: reachable-nodes-def*)

**lemma** *reachable-nodes-step'*:  
**assumes**  $u \in \text{reachable-nodes } g \ r$   $(u, v) \in \text{edges } g$   
**shows**  $v \in \text{reachable-nodes } g \ r$   $(u, v) \in \text{edges } (\text{component-of } g \ r)$   
**proof** –  
**show**  $v \in \text{reachable-nodes } g \ r$   
**by** (*meson ImageI assms(1) assms(2) reachable-nodes-step rev-subsetD*)  
**then show**  $(u, v) \in \text{edges } (\text{component-of } g \ r)$   
**by** (*simp add: assms(1) assms(2) component-of-def*)  
**qed**

**lemma** *reachable-nodes-steps'*:  
**assumes**  $u \in \text{reachable-nodes } g \ r$   $(u, v) \in (\text{edges } g)^*$   
**shows**  $v \in \text{reachable-nodes } g \ r$   $(u, v) \in (\text{edges } (\text{component-of } g \ r))^*$   
**proof** –  
**show**  $v \in \text{reachable-nodes } g \ r$  **using** *reachable-nodes-steps assms* **by fast**  
**show**  $(u, v) \in (\text{edges } (\text{component-of } g \ r))^*$   
**using** *assms(2,1)*  
**apply** (*induction rule: converse-rtrancl-induct*)  
**subgoal by auto**  
**subgoal by** (*smt converse-rtrancl-into-rtrancl reachable-nodes-step'*)  
**done**  
**qed**

**lemma** *reachable-not-node*:  $r \notin \text{nodes } g \implies \text{reachable-nodes } g \ r = \{r\}$   
**by** (*force elim: converse-rtranclE simp: reachable-nodes-def intro: nodesI*)

**lemma** *nodes-of-component[simp]*:  $\text{nodes } (\text{component-of } g \ r) = \text{reachable-nodes } g \ r$   
**apply** (*rule equalityI*)  
**unfolding** *component-of-def reachable-nodes-def*  
**subgoal by auto**  
**subgoal by** *clarsimp (metis nodesI(2) rtranclE)*  
**done**

**lemma** *component-connected[simp, intro!]*: *connected* (*component-of g r*)  
**proof** (*rule connectedI; simp*)  
**fix**  $u \ v$   
**assume**  $A: u \in \text{reachable-nodes } g \ r \ v \in \text{reachable-nodes } g \ r$   
**hence**  $(u, r) \in (\text{edges } g)^* \ (r, v) \in (\text{edges } g)^*$   
**by** (*auto simp: reachable-nodes-def dest: rtrancl-edges-sym'*)

**hence**  $(u,v) \in (\text{edges } g)^*$  **by** (rule rtrancl-trans)  
**with**  $A$  **show**  $(u, v) \in (\text{edges } (\text{component-of } g \ r))^*$   
**by** (rule-tac reachable-nodes-steps'(2))  
**qed**

**lemma** *component-edges-subset*:  $\text{edges } (\text{component-of } g \ r) \subseteq \text{edges } g$   
**by** (auto simp: component-of-def)

**lemma** *component-path*:  $u \in \text{nodes } (\text{component-of } g \ r) \implies$   
 $\text{path } (\text{component-of } g \ r) \ u \ p \ v \longleftrightarrow \text{path } g \ u \ p \ v$   
**apply** rule  
**subgoal** **by** (erule path-mono[OF component-edges-subset])  
**subgoal** **by** (induction p arbitrary: u) (auto simp: reachable-nodes-step')  
**done**

**lemma** *component-cycle-free*:  $\text{cycle-free } g \implies \text{cycle-free } (\text{component-of } g \ r)$   
**by** (meson component-edges-subset cycle-free-antimono)

**lemma** *component-of-connected-graph*:  
 $\llbracket \text{connected } g; r \in \text{nodes } g \rrbracket \implies \text{component-of } g \ r = g$   
**unfolding** graph-eq-iff  
**apply** safe  
**subgoal** **by** simp (metis Image-singleton-iff nodesI(2) reachable-nodes-def rtranclE)  
**subgoal** **by** (simp add: connectedD reachable-nodes-def)  
**subgoal** **by** (simp add: component-of-def)  
**subgoal** **by** (simp add: connectedD reachable-nodes-def reachable-nodes-step'(2))  
**done**

**lemma** *component-of-not-node*:  $r \notin \text{nodes } g \implies \text{component-of } g \ r = \text{graph } \{r\} \ \{\}$   
**by** (clarsimp simp: graph-eq-iff component-of-def reachable-not-node graph-accs)

### 1.1.8 Trees

**definition** *tree*  $g \equiv \text{connected } g \wedge \text{cycle-free } g$

**lemma** *tree-empty[simp]*: *tree* *graph-empty* **by** (simp add: tree-def)

**lemma** *component-of-tree*: *tree*  $T \implies \text{tree } (\text{component-of } T \ r)$   
**unfolding** tree-def **using** component-connected component-cycle-free **by** auto

### Joining and Splitting Trees on Single Edge

**lemma** *join-connected*:  
**assumes** *CONN*: *connected*  $g_1$  *connected*  $g_2$   
**assumes** *IN-NODES*:  $u \in \text{nodes } g_1 \ v \in \text{nodes } g_2$   
**shows** *connected* (*ins-edge*  $(u,v)$  (*graph-join*  $g_1 \ g_2$ )) (**is** *connected*  $?g'$ )  
**unfolding** connected-def  
**proof** *clarify*  
**fix**  $a \ b$

```

assume  $A: a \in \text{nodes } ?g' \ b \in \text{nodes } ?g'$ 

have  $ESS: (\text{edges } g_1)^* \subseteq (\text{edges } ?g')^* (\text{edges } g_2)^* \subseteq (\text{edges } ?g')^*$ 
  using  $\text{edges-ins-edge-ss}$ 
  by  $(\text{force intro!}: \text{rtrancl-mono})+$ 

have  $UV: (u,v) \in (\text{edges } ?g')^*$ 
  by  $(\text{simp add}: \text{edges-ins-edge r-into-rtrancl})$ 

show  $(a,b) \in (\text{edges } ?g')^*$ 
proof -
  {
    assume  $a \in \text{nodes } g_1 \ b \in \text{nodes } g_1$ 
    hence  $?thesis$  using  $\langle \text{connected } g_1 \rangle ESS(1)$  unfolding  $\text{connected-def}$  by  $\text{blast}$ 
  } moreover {
    assume  $a \in \text{nodes } g_2 \ b \in \text{nodes } g_2$ 
    hence  $?thesis$  using  $\langle \text{connected } g_2 \rangle ESS(2)$  unfolding  $\text{connected-def}$  by  $\text{blast}$ 
  } moreover {
    assume  $a \in \text{nodes } g_1 \ b \in \text{nodes } g_2$ 
    with  $\text{connectedD}[OF \text{CONN}(1)] \text{connectedD}[OF \text{CONN}(2)] ESS$ 
    have  $?thesis$  by  $(\text{meson } UV \text{IN-NODES } \text{contra-subsetD } \text{rtrancl-trans})$ 
  } moreover {
    assume  $a \in \text{nodes } g_2 \ b \in \text{nodes } g_1$ 
    with  $\text{connectedD}[OF \text{CONN}(1)] \text{connectedD}[OF \text{CONN}(2)] ESS$ 
    have  $?thesis$ 
    by  $(\text{meson } UV \text{IN-NODES } \text{contra-subsetD } \text{rtrancl-edges-sym}' \text{rtrancl-trans})$ 
  }
  }
  ultimately show  $?thesis$  using  $A \text{IN-NODES}$  by  $\text{auto}$ 
qed
qed

```

**lemma**  $\text{join-cycle-free}$ :

```

assumes  $CYCF: \text{cycle-free } g_1 \ \text{cycle-free } g_2$ 
assumes  $DJ: \text{nodes } g_1 \cap \text{nodes } g_2 = \{\}$ 
assumes  $\text{IN-NODES}: u \in \text{nodes } g_1 \ v \in \text{nodes } g_2$ 
shows  $\text{cycle-free } (\text{ins-edge } (u,v) (\text{graph-join } g_1 \ g_2))$  (is cycle-free } ?g')
proof  $(\text{rule } \text{cycle-freeI})$ 
  fix  $p \ a$ 
  assume  $P: \text{path } ?g' \ a \ p \ a \ p \neq []$   $\text{simple } p$ 
  from  $\text{path-endpoints}[OF \text{this}(1,2)] \text{IN-NODES}$ 
    have  $A\text{-NODE}: a \in \text{nodes } g_1 \cup \text{nodes } g_2$ 
    by  $\text{auto}$ 
  thus  $\text{False}$  proof
    assume  $N1: a \in \text{nodes } g_1$ 
    have  $\text{set } p \subseteq \text{nodes } g_1 \times \text{nodes } g_1$ 
    proof  $(\text{cases})$ 
      rule:  $\text{find-crossing-edges-on-path}[\text{where } P = \lambda x. x \in \text{nodes } g_1, OF \text{P}(1) \text{N1 } \text{N1}]$ 
      case  $1$ 

```

```

    then show ?thesis by auto
  next
    case (2 u1 v1 v2 u2 p1 p2 p3)
    then show ?thesis using ‹simple p› P
      apply clarsimp
      apply (drule path-edges)+
      apply (cases u=v; clarsimp simp: edges-ins-edge uedge-in-set-eq)
      apply (metis DJ IntI IN-NODES empty-iff)
      by (metis DJ IntI empty-iff nodesI uedge-eq-iff)

  qed
  hence set p ⊆ edges g1 using DJ edges-subset path-edges[OF P(1)] IN-NODES
    by (auto simp: edges-ins-edge split: if-splits; blast)
  hence path g1 a p a by (meson P(1) path-graph-cong)
  thus False using cycle-freeD[OF CYCF(1)] P(2,3) by blast
next
  assume N2: a ∈ nodes g2
  have set p ⊆ nodes g2 × nodes g2
  proof (cases
    rule: find-crossing-edges-on-path[where P=λx. x ∈ nodes g2, OF P(1) N2 N2])
  case 1
    then show ?thesis by auto
  next
    case (2 u1 v1 v2 u2 p1 p2 p3)
    then show ?thesis using ‹simple p› P
      apply clarsimp
      apply (drule path-edges)+
      apply (cases u=v; clarsimp simp: edges-ins-edge uedge-in-set-eq)
      apply (metis DJ IntI IN-NODES empty-iff)
      by (metis DJ IntI empty-iff nodesI uedge-eq-iff)

  qed
  hence set p ⊆ edges g2 using DJ edges-subset path-edges[OF P(1)] IN-NODES
    by (auto simp: edges-ins-edge split: if-splits; blast)
  hence path g2 a p a by (meson P(1) path-graph-cong)
  thus False using cycle-freeD[OF CYCF(2)] P(2,3) by blast
qed
qed

lemma join-trees:
  assumes TREE: tree g1 tree g2
  assumes DJ: nodes g1 ∩ nodes g2 = {}
  assumes IN-NODES: u ∈ nodes g1 v ∈ nodes g2
  shows tree (ins-edge (u,v) (graph-join g1 g2))
  using assms join-cycle-free join-connected unfolding tree-def by metis

lemma split-tree:
  assumes tree T (x,y) ∈ edges T

```

**defines**  $E' \equiv (\text{edges } T - \{(x,y),(y,x)\})$   
**obtains**  $T1\ T2$  **where**  
*tree*  $T1$  *tree*  $T2$   
 $\text{nodes } T1 \cap \text{nodes } T2 = \{\}$   $\text{nodes } T = \text{nodes } T1 \cup \text{nodes } T2$   
 $\text{edges } T1 \cup \text{edges } T2 = E'$   
 $\text{nodes } T1 = \{ u. (x,u) \in E'^* \}$   $\text{nodes } T2 = \{ u. (y,u) \in E'^* \}$   
 $x \in \text{nodes } T1\ y \in \text{nodes } T2$

**proof** –

**define**  $N1$  **where**  $N1 = \{ u. (x,u) \in E'^* \}$   
**define**  $N2$  **where**  $N2 = \{ u. (y,u) \in E'^* \}$

**define**  $T1$  **where**  $T1 = \text{restrict-nodes } T\ N1$   
**define**  $T2$  **where**  $T2 = \text{restrict-nodes } T\ N2$

**have**  $SYME$ : *sym*  $(E'^*)$   
**apply** (*rule sym-rtrancl*)  
**using** *edges-sym*[*of*  $T$ ] **by** (*auto simp: sym-def E'-def*)

**from** *assms* **have** *connected*  $T$  *cycle-free*  $T$  **unfolding** *tree-def* **by** *auto*  
**from**  $\langle \text{cycle-free } T \rangle$  **have** *cycle-free*  $T1$  *cycle-free*  $T2$   
**unfolding** *T1-def* *T2-def*  
**using** *cycle-free-antimono unrestrictn-edges* **by** *blast+*

**from**  $\langle (x,y) \in \text{edges } T \rangle$  **have**  $XYN$ :  $x \in \text{nodes } T\ y \in \text{nodes } T$   
**using** *edges-subset* **by** *auto*  
**from**  $XYN$  **have** [*simp*]:  $\text{nodes } T1 = N1\ \text{nodes } T2 = N2$   
**unfolding** *T1-def* *T2-def* *N1-def* *N2-def* **unfolding** *E'-def*  
**apply** (*safe*)  
**apply** (*all*  $\langle \text{clarsimp} \rangle$ )  
**by** (*metis DiffD1 nodesI(2) rtrancl.simps*)+

**have**  $x \in N1\ y \in N2$  **by** (*auto simp: N1-def N2-def*)

**have**  $N1 \cap N2 = \{\}$

**proof** (*safe; simp*)

**fix**  $u$

**assume**  $u \in N1\ u \in N2$

**hence**  $(x,u) \in E'^*\ (u,y) \in E'^*$  **by** (*auto simp: N1-def N2-def symD[OF SYME]*)

**with** *cycle-free-altD*[*OF*  $\langle \text{cycle-free } T \rangle\ \langle (x,y) \in \text{edges } T \rangle$ ] **show** *False*

**unfolding** *E'-def* **by** (*meson rtrancl-trans*)

**qed**

**have**  $N1C$ :  $E' \text{“} N1 \subseteq N1$

**unfolding** *N1-def*

**apply** *clarsimp*

**by** (*simp add: rtrancl.rtrancl-into-rtrancl*)

```

have N2C:  $E' \setminus N2 \subseteq N2$ 
  unfolding N2-def
  apply clarsimp
  by (simp add: rtrancl.rtrancl-into-rtrancl)

have XE1:  $(x,u) \in (\text{edges } T1)^*$  if  $u \in N1$  for  $u$ 
proof -
  from that have  $(x,u) \in E'^*$  by (auto simp: N1-def)
  then show ?thesis using  $\langle x \in N1 \rangle$ 
    unfolding T1-def
  proof (induction rule: converse-rtrancl-induct)
    case (step  $y z$ )
    with N1C have  $z \in N1$  by auto
    with step.hyps(1) step.prem1 have  $(y,z) \in \text{Restr } (\text{edges } T) N1$ 
      unfolding E'-def by auto
    with step.IH[OF  $\langle z \in N1 \rangle$ ] show ?case
      by (metis converse-rtrancl-into-rtrancl edges-restrict-nodes)
  qed auto
qed

have XE2:  $(y,u) \in (\text{edges } T2)^*$  if  $u \in N2$  for  $u$ 
proof -
  from that have  $(y,u) \in E'^*$  by (auto simp: N2-def)
  then show ?thesis using  $\langle y \in N2 \rangle$ 
    unfolding T2-def
  proof (induction rule: converse-rtrancl-induct)
    case (step  $y z$ )
    with N2C have  $z \in N2$  by auto
    with step.hyps(1) step.prem1 have  $(y,z) \in \text{Restr } (\text{edges } T) N2$ 
      unfolding E'-def by auto
    with step.IH[OF  $\langle z \in N2 \rangle$ ] show ?case
      by (metis converse-rtrancl-into-rtrancl edges-restrict-nodes)
  qed auto
qed

have connected T1
  apply rule
  apply simp
  apply (drule XE1)+
  by (meson rtrancl-edges-sym' rtrancl-trans)

have connected T2
  apply rule
  apply simp
  apply (drule XE2)+
  by (meson rtrancl-edges-sym' rtrancl-trans)

```

```

have  $u \in N1 \cup N2$  if  $u \in \text{nodes } T$  for  $u$ 
proof -
  from connectedD[OF  $\langle \text{connected } T \rangle \langle x \in \text{nodes } T \rangle$  that ]
  obtain  $p$  where  $P$ : path  $T$   $x$   $p$   $u$  simple  $p$ 
  by (auto simp: rtrancl-edges-iff-path elim: simplify-pathE)
  show ?thesis proof cases
  assume  $(x,y) \notin \text{set } p \wedge (y,x) \notin \text{set } p$ 
  with  $P(1)$  have path (restrict-edges  $T$   $E'$ )  $x$   $p$   $u$ 
  unfolding  $E'$ -def by (erule-tac path-graph-cong) auto
  from path-rtrancl-edgesD[OF this]
  show ?thesis unfolding  $N1$ -def  $E'$ -def by auto
next
  assume  $\neg((x,y) \notin \text{set } p \wedge (y,x) \notin \text{set } p)$ 
  with  $P$  obtain  $p'$  where
    uedge  $(x,y) \notin \text{set } (\text{map } \text{uedge } p')$  path  $T$   $y$   $p'$   $u \vee \text{path } T$   $x$   $p'$   $u$ 
    by (auto simp: in-set-conv-decomp uedge-commute)
  hence path (restrict-edges  $T$   $E'$ )  $y$   $p'$   $u \vee \text{path}$  (restrict-edges  $T$   $E'$ )  $x$   $p'$   $u$ 
  apply (clarsimp simp: uedge-in-set-eq E'-def)
  by (smt ComplD DiffI Int-iff UnCI edges-restrict-edges insertE
    path-graph-cong subset-Compl-singleton subset-iff)
  then show ?thesis unfolding  $N1$ -def  $N2$ -def  $E'$ -def
  by (auto dest: path-rtrancl-edgesD)
qed
qed
then have nodes  $T = N1 \cup N2$ 
  unfolding  $N1$ -def  $N2$ -def using XYN
  unfolding  $E'$ -def
  apply (safe)
  subgoal by auto []
  subgoal by (metis DiffD1 nodesI(2) rtrancl.cases)
  subgoal by (metis DiffD1 nodesI(2) rtrancl.cases)
  done

have edges  $T1 \cup \text{edges } T2 \subseteq E'$ 
  unfolding  $T1$ -def  $T2$ -def  $E'$ -def using  $\langle N1 \cap N2 = \{\} \rangle \langle x \in N1 \rangle \langle y \in N2 \rangle$ 
  by auto
also have edges  $T1 \cup \text{edges } T2 \supseteq E'$ 
proof -
  note  $ED1 = \text{nodesI}[\text{where } g=T, \text{unfolded } \langle \text{nodes } T = N1 \cup N2 \rangle]$ 
  have  $E' \subseteq \text{edges } T$  by (auto simp: E'-def)
  thus edges  $T1 \cup \text{edges } T2 \supseteq E'$ 
  unfolding  $T1$ -def  $T2$ -def
  using  $ED1$   $N1C$   $N2C$  by (auto; blast)
qed
finally have edges  $T1 \cup \text{edges } T2 = E'$  .

show ?thesis
  apply (rule that[of  $T1$   $T2$ , unfolded tree-def]; (intro conjI)?; fact?)
  apply simp-all

```



apply fact+  
 done  
 qed

### 1.1.9 Spanning Trees

**definition** *is-spanning-tree*  $G T$   
 $\equiv \text{tree } T \wedge \text{nodes } T = \text{nodes } G \wedge \text{edges } T \subseteq \text{edges } G$

**lemma** *connected-singleton[simp]*: *connected* (*ins-node*  $u$  *graph-empty*)  
**unfolding** *connected-def* **by** *auto*

**lemma** *path-singleton[simp]*: *path* (*ins-node*  $u$  *graph-empty*)  $v p w \longleftrightarrow v=w \wedge p=[]$

**by** (*cases*  $p$ ) *auto*

**lemma** *tree-singleton[simp]*: *tree* (*ins-node*  $u$  *graph-empty*)  
**by** (*simp* *add: cycle-free-no-edges tree-def*)

**lemma** *tree-add-edge-in-out*:

**assumes** *tree*  $T$

**assumes**  $u \in \text{nodes } T \ v \notin \text{nodes } T$

**shows** *tree* (*ins-edge*  $(u,v)$   $T$ )

**proof** –

**from** *assms* **have** [*simp*]:  $u \neq v$  **by** *auto*

**have** *ins-edge*  $(u,v)$   $T = \text{ins-edge } (u,v)$  (*graph-join*  $T$  (*ins-node*  $v$  *graph-empty*))

**by** (*auto simp: graph-eq-iff*)

**also have** *tree* ...

**apply** (*rule join-trees*)

**using** *assms*

**by** *auto*

**finally show** *?thesis* .

qed

Remove edges on cycles until the graph is cycle free

**lemma** *ex-spanning-tree*:

*connected*  $g \implies \exists t. \text{is-spanning-tree } g t$

**using** *edges-finite*[*of*  $g$ ]

**proof** (*induction edges*  $g$  *arbitrary: g rule: finite-psubset-induct*)

**case** *psubset*

**show** *?case* **proof** (*cases cycle-free*  $g$ )

**case** *True*

**with**  $\langle \text{connected } g \rangle$  **show** *?thesis* **by** (*auto simp: is-spanning-tree-def tree-def*)

**next**

**case** *False*

**then obtain**  $u v$  **where**

$EDGE: (u,v) \in \text{edges } g$

**and**  $RED: (u,v) \in (edges\ g - \{(u,v), (v,u)\})^*$   
**using** *cycle-free-altI* **by** *metis*  
**from**  $\langle connected\ g \rangle$   
**have**  $connected\ (restrict\ edges\ g\ (-\ \{(u,v), (v,u)\}))$  **(is**  $connected\ ?g'$   
**unfolding** *connected-def*  
**by**  $(auto\ simp: remove\ redundant\ edge[OF\ RED])$   
**moreover** **have**  $edges\ ?g' \subset edges\ g$  **using** *EDGE* **by** *auto*  
**ultimately** **obtain**  $t$  **where**  $is\ spanning\ tree\ ?g'\ t$   
**using**  $psubset.hyps(2)[of\ ?g']$  **by** *blast*  
**hence**  $is\ spanning\ tree\ g\ t$  **by**  $(auto\ simp: is\ spanning\ tree\ def)$   
**thus**  $?thesis\ ..$   
**qed**  
**qed**

## 1.2 Weighted Undirected Graphs

**definition**  $weight :: ('v\ set \Rightarrow nat) \Rightarrow 'v\ ugraph \Rightarrow nat$   
**where**  $weight\ w\ g \equiv (\sum\ e \in edges\ g.\ w\ (uedge\ e))\ div\ 2$

**lemma** *weight-alt*:  $weight\ w\ g = (\sum\ e \in uedge\ 'edges\ g.\ w\ e)$

**proof** –

**from** *split-edges-sym*[*of*  $g$ ] **obtain**  $E$  **where**  
 $edges\ g = E \cup E^{-1}$  **and**  $E \cap E^{-1} = \{\}$  **by** *auto*  
**hence** [*simp, intro!*]:  $finite\ E$  **by**  $(metis\ edges\ finite\ finite\ Un)$   
**hence** [*simp, intro!*]:  $finite\ (E^{-1})$  **by** *blast*

**have** [*simp*]:  $(\sum\ e \in E^{-1}.\ w\ (uedge\ e)) = (\sum\ e \in E.\ w\ (uedge\ e))$   
**apply**  $(rule\ sum.reindex\ cong[where\ l=prod.swap\ and\ A=E^{-1}\ and\ B=E])$   
**by**  $(auto\ simp: uedge\ def\ insert\ commute)$

**have** [*simp*]:  $inj\ on\ uedge\ E$  **using**  $\langle E \cap E^{-1} = \{\} \rangle$   
**by**  $(auto\ simp: uedge\ def\ inj\ on\ def\ doubleton\ eq\ iff)$

**have**  $weight\ w\ g = (\sum\ e \in E.\ w\ (uedge\ e))$   
**unfolding** *weight-def*  $\langle edges\ g = \rightarrow \rangle$  **using**  $\langle E \cap E^{-1} = \{\} \rangle$   
**by**  $(auto\ simp: sum.union\ disjoint)$   
**also** **have**  $\dots = (\sum\ e \in uedge\ 'E.\ w\ e)$   
**using** *sum.reindex*[*of*  $uedge\ E\ w$ ]  
**by** *auto*  
**also** **have**  $uedge\ 'E = uedge\ (edges\ g)$   
**unfolding**  $\langle edges\ g = \rightarrow \rangle$  *uedge-def* **using**  $\langle E \cap E^{-1} = \{\} \rangle$   
**by** *auto*  
**finally** **show**  $?thesis\ .$

**qed**

**lemma** *weight-empty*[*simp*]:  $weight\ w\ graph\ empty = 0$  **unfolding** *weight-def* **by** *auto*

**lemma** *weight-ins-edge*[simp]:  $\llbracket u \neq v; (u,v) \notin \text{edges } g \rrbracket$   
 $\implies \text{weight } w (\text{ins-edge } (u,v) g) = w \{u,v\} + \text{weight } w g$   
**unfolding** *weight-def*  
**apply** *clarsimp*  
**apply** (*subst sum.insert*)  
**by** (*auto dest: edges-sym' simp: uedge-def insert-commute*)

**lemma** *uedge-img-disj-iff*[simp]:  
 $\text{uedge } \langle \text{edges } g_1 \cap \text{edges } g_2 = \{\} \rangle \longleftrightarrow \text{edges } g_1 \cap \text{edges } g_2 = \{\}$   
**by** (*auto simp: uedge-eq-iff dest: edges-sym'*)<sup>+</sup>

**lemma** *weight-join*[simp]:  $\text{edges } g_1 \cap \text{edges } g_2 = \{\}$   
 $\implies \text{weight } w (\text{graph-join } g_1 g_2) = \text{weight } w g_1 + \text{weight } w g_2$   
**unfolding** *weight-alt* **by** (*auto simp: sum.union-disjoint image-Un*)

**lemma** *weight-cong*:  $\text{edges } g_1 = \text{edges } g_2 \implies \text{weight } w g_1 = \text{weight } w g_2$   
**by** (*auto simp: weight-def*)

**lemma** *weight-mono*:  $\text{edges } g \subseteq \text{edges } g' \implies \text{weight } w g \leq \text{weight } w g'$   
**unfolding** *weight-alt* **by** (*rule sum-mono2*) *auto*

**lemma** *weight-ge-edge*:  
**assumes**  $(x,y) \in \text{edges } T$   
**shows**  $\text{weight } w T \geq w \{x,y\}$   
**using** *assms* **unfolding** *weight-alt*  
**by** (*auto simp: uedge-def intro: member-le-sum*)

**lemma** *weight-del-edge*[simp]:  
**assumes**  $(x,y) \in \text{edges } T$   
**shows**  $\text{weight } w (\text{restrict-edges } T (- \{(x,y), (y,x)\})) = \text{weight } w T - w \{x,y\}$   
**proof** –  
**define** *E* **where**  $E = \text{uedge } \langle \text{edges } T - \{(x,y)\} \rangle$   
**have** [simp]:  $(\text{uedge } \langle \text{edges } T - \{(x,y), (y,x)\} \rangle) = E$   
**by** (*safe; simp add: E-def uedge-def doubleton-eq-iff; blast*)

**from** *assms* **have** [simp]:  $\text{uedge } \langle \text{edges } T = \text{insert } \{x,y\} E \rangle$   
**unfolding** *E-def* **by** *force*

**have** [simp]:  $\{x,y\} \notin E$  **unfolding** *E-def* **by** *blast*

**then show** *?thesis*  
**unfolding** *weight-alt*  
**apply** *simp*  
**by** (*metis E-def ‹uedge ‹edges T = insert {x,y} E› insertI1 sum-diff1-nat*)

**qed**

### 1.2.1 Minimum Spanning Trees

**definition** *is-MST*  $w\ g\ t \equiv is\text{-spanning-tree}\ g\ t$   
 $\wedge (\forall t'. is\text{-spanning-tree}\ g\ t' \longrightarrow weight\ w\ t \leq weight\ w\ t')$

**lemma** *exists-MST*: *connected*  $g \implies \exists t. is\text{-MST}\ w\ g\ t$   
**using** *ex-has-least-nat*[*of is-spanning-tree*  $g$ ] *ex-spanning-tree*  
**unfolding** *is-MST-def*  
**by** *blast*

**end**

## 1.3 Abstract Graph Datatype

**theory** *Undirected-Graph-Specs*  
**imports** *Undirected-Graph*  
**begin**

### 1.3.1 Abstract Weighted Graph

**locale** *adt-wgraph* =  
**fixes**  $\alpha w :: 'g \Rightarrow 'v\ set \Rightarrow nat$  **and**  $\alpha g :: 'g \Rightarrow 'v\ ugraph$   
**and**  $invar :: 'g \Rightarrow bool$   
**and**  $adj :: 'g \Rightarrow 'v \Rightarrow ('v \times nat)\ list$   
**and**  $empty :: 'g$   
**and**  $add\text{-edge} :: 'v \times 'v \Rightarrow nat \Rightarrow 'g \Rightarrow 'g$   
**assumes** *adj-correct*: *invar*  $g$   
 $\implies set\ (adj\ g\ u) = \{(v,d). (u,v) \in edges\ (\alpha g\ g) \wedge \alpha w\ g\ \{u,v\} = d\}$   
**assumes** *empty-correct*:  
*invar*  $empty$   
 $\alpha g\ empty = graph\ empty$   
 $\alpha w\ empty = (\lambda\_. 0)$   
**assumes** *add-edge-correct*:  
 $\llbracket invar\ g; (u,v) \notin edges\ (\alpha g\ g); u \neq v \rrbracket \implies invar\ (add\text{-edge}\ (u,v)\ d\ g)$   
 $\llbracket invar\ g; (u,v) \notin edges\ (\alpha g\ g); u \neq v \rrbracket$   
 $\implies \alpha g\ (add\text{-edge}\ (u,v)\ d\ g) = ins\text{-edge}\ (u,v)\ (\alpha g\ g)$   
 $\llbracket invar\ g; (u,v) \notin edges\ (\alpha g\ g); u \neq v \rrbracket$   
 $\implies \alpha w\ (add\text{-edge}\ (u,v)\ d\ g) = (\alpha w\ g)(\{u,v\} := d)$

**begin**

**lemmas** *wgraph-specs* = *adj-correct* *empty-correct* *add-edge-correct*

**lemma** *empty-spec-presentation*:  
 $invar\ empty \wedge \alpha g\ empty = graph\ \{\}\ \{\} \wedge \alpha w\ empty = (\lambda\_. 0)$   
**by** (*auto simp: wgraph-specs*)

**lemma** *add-edge-spec-presentation*:  
 $\llbracket invar\ g; (u,v) \notin edges\ (\alpha g\ g); u \neq v \rrbracket \implies$

*invar* (*add-edge* (*u,v*) *d g*)  
 $\wedge \alpha g$  (*add-edge* (*u,v*) *d g*) = *ins-edge* (*u,v*) ( $\alpha g$  *g*)  
 $\wedge \alpha w$  (*add-edge* (*u,v*) *d g*) = ( $\alpha w$  *g*)( $\{u,v\}:=d$ )  
**by** (*auto simp: wgraph-specs*)

**end**

### 1.3.2 Generic From-List Algorithm

**definition** *valid-graph-repr* :: ( $'v \times 'v$ ) *list*  $\Rightarrow$  *bool*  
**where** *valid-graph-repr* *l*  $\longleftrightarrow$  ( $\forall (u,v) \in \text{set } l. u \neq v$ )

**definition** *graph-from-list* :: ( $'v \times 'v$ ) *list*  $\Rightarrow$   $'v$  *ugraph*  
**where** *graph-from-list* *l* = *foldr ins-edge l graph-empty*

**lemma** *graph-from-list-foldl*: *graph-from-list* *l* = *fold ins-edge l graph-empty*  
**unfolding** *graph-from-list-def*  
**apply** (*rule foldr-fold[THEN fun-cong]*)  
**by** (*auto simp: fun-eq-iff graph-eq-iff edges-ins-edge*)

**lemma** *nodes-of-graph-from-list*: *nodes* (*graph-from-list* *l*) = *fst*'*set* *l*  $\cup$  *snd*'*set* *l*  
**apply** (*induction l*)  
**unfolding** *graph-from-list-def*  
**by** *auto*

**lemma** *edges-of-graph-from-list*:  
**assumes** *valid*: *valid-graph-repr* *l*  
**shows** *edges* (*graph-from-list* *l*) = *set* *l*  $\cup$  (*set* *l*)<sup>-1</sup>  
**using** *valid* **apply** (*induction l*)  
**unfolding** *graph-from-list-def valid-graph-repr-def*  
**by** *auto*

**definition** *valid-weight-repr* *l*  $\equiv$  *distinct* (*map* (*uedge o fst*) *l*)

**definition** *weight-from-list* :: ( $(('v \times 'v) \times \text{nat})$ ) *list*  $\Rightarrow$   $'v$  *set*  $\Rightarrow$  *nat* **where**  
*weight-from-list* *l*  $\equiv$  *foldr* ( $\lambda((u,v),d) w. w(\{u,v\}:=d)$ ) *l* ( $\lambda-. 0$ )

**lemma** *graph-from-list-simps*:  
*graph-from-list* [] = *graph-empty*  
*graph-from-list* ((*u,v*)#*l*) = *ins-edge* (*u,v*) (*graph-from-list* *l*)  
**by** (*auto simp: graph-from-list-def*)

**lemma** *weight-from-list-simps*:  
*weight-from-list* [] = ( $\lambda-. 0$ )  
*weight-from-list* (((*u,v*),*d*)#*xs*) = (*weight-from-list* *xs*)( $\{u,v\}:=d$ )  
**by** (*auto simp: weight-from-list-def*)

**lemma** *valid-graph-repr-simps*:  
*valid-graph-repr* []  
*valid-graph-repr* ((*u,v*)#*xs*)  $\longleftrightarrow$  *u*≠*v* ∧ *valid-graph-repr xs*  
**unfolding** *valid-graph-repr-def* **by** *auto*

**lemma** *valid-weight-repr-simps*:  
*valid-weight-repr* []  
*valid-weight-repr* (((*u,v*),*w*)#*xs*)  
 $\longleftrightarrow$  *uedge* (*u,v*)∉*uedge'fst'set xs* ∧ *valid-weight-repr xs*  
**unfolding** *valid-weight-repr-def*  
**by** (*force simp: uedge-def doubleton-eq-iff*)**+**

**lemma** *weight-from-list-correct*:  
**assumes** *valid-weight-repr l*  
**assumes** ((*u,v*),*d*)∈*set l*  
**shows** *weight-from-list l* {*u,v*} = *d*  
**proof** –  
**from** *assms* **show** ?*thesis*  
**apply** (*induction l*)  
**unfolding** *valid-weight-repr-def weight-from-list-def*  
**subgoal by** *simp*  
**by** (*force simp: doubleton-eq-iff*)

qed

**context** *adt-wgraph*  
**begin**

**definition** *valid-wgraph-repr l*  
 $\longleftrightarrow$  *valid-graph-repr* (*map fst l*) ∧ *valid-weight-repr l*

**definition** *from-list l* = *foldr* ( $\lambda$ (*e,d*). *add-edge e d*) *l empty*

**lemma** *from-list-refine: valid-wgraph-repr l*  $\implies$   
*invar* (*from-list l*)  
 $\wedge$   $\alpha_g$  (*from-list l*) = *graph-from-list* (*map fst l*)  
 $\wedge$   $\alpha_w$  (*from-list l*) = *weight-from-list l*  
**unfolding** *from-list-def valid-wgraph-repr-def*  
**supply** [*simp*] = *wgraph-specs graph-from-list-simps weight-from-list-simps*  
**apply** (*induction l*)  
**subgoal by** *auto*  
**subgoal by** (  
*intro conjI*;  
*clarsimp*

```

    simp: uedge-def valid-graph-repr-simps valid-weight-repr-simps
    split: prod.splits;
    subst wgraph-specs;
    auto simp: edges-of-graph-from-list
  )
done

```

**lemma** *from-list-correct*:

```

assumes valid-wgraph-repr l
shows
  invar (from-list l)
  nodes ( $\alpha g$  (from-list l)) = fst'fst'set l  $\cup$  snd'fst'set l
  edges ( $\alpha g$  (from-list l)) = (fst'set l)  $\cup$  (fst'set l)-1
  ((u,v),d)  $\in$  set l  $\implies$   $\alpha w$  (from-list l) {u,v} = d
apply (simp-all add: from-list-refine[OF assms])
using assms unfolding valid-wgraph-repr-def
apply (simp-all add:
  edges-of-graph-from-list nodes-of-graph-from-list weight-from-list-correct)
done

```

**lemma** *valid-wgraph-repr-presentation*: valid-wgraph-repr l  $\longleftrightarrow$

( $\forall ((u,v),d) \in \text{set } l. u \neq v \wedge \text{distinct } [\{u,v\}. ((u,v),d) \leftarrow l]$ )

**proof** –

```

have [simp]: uedge  $\circ$  fst = ( $\lambda((u, v), w). \{u, v\}$ )
unfolding uedge-def by auto
show ?thesis
unfolding valid-wgraph-repr-def valid-graph-repr-def valid-weight-repr-def
by (auto split: prod.splits)

```

**qed**

**lemma** *from-list-correct-presentation*:

```

assumes valid-wgraph-repr l
shows let gi=from-list l; g= $\alpha g$  gi; w= $\alpha w$  gi in
  invar gi
   $\wedge$  nodes g =  $\bigcup \{\{u,v\} \mid u \ v. \exists d. ((u,v),d) \in \text{set } l\}$ 
   $\wedge$  edges g =  $\bigcup \{\{(u,v),(v,u)\} \mid u \ v. \exists d. ((u,v),d) \in \text{set } l\}$ 
   $\wedge$  ( $\forall ((u,v),d) \in \text{set } l. w \{u,v\} = d$ )

```

```

unfolding Let-def from-list-correct(2-3)[OF assms]
apply (intro conjI)
subgoal by (simp add: from-list-correct(1)[OF assms])
subgoal by (auto 0 0 simp: in-set-conv-decomp; blast)
subgoal by (auto 0 0 simp: in-set-conv-decomp; blast)
subgoal using from-list-correct(4)[OF assms] by auto
done

```

**end**

**end**

## 1.4 Abstract Prim Algorithm

```

theory Prim-Abstract
imports
  Main
  Common
  Undirected-Graph
  HOL-Eisbach.Eisbach
begin

```

### 1.4.1 Generic Algorithm: Light Edges

**definition** *is-subset-MST*  $w\ g\ A \equiv \exists t. \text{is-MST } w\ g\ t \wedge A \subseteq \text{edges } t$

**lemma** *is-subset-MST-empty*[simp]:  $\text{connected } g \implies \text{is-subset-MST } w\ g\ \{\}$   
**using** *exists-MST* **unfolding** *is-subset-MST-def* **by** *blast*

We fix a start node and a weighted graph

```

locale Prim =
  fixes  $w :: 'v \text{ set} \Rightarrow \text{nat}$  and  $g :: 'v \text{ ugraph}$  and  $r :: 'v$ 
begin

```

Reachable part of the graph

**definition** *rg*  $\equiv \text{component-of } g\ r$

**lemma** *reachable-connected*[simp, intro!]:  $\text{connected } rg$   
**unfolding** *rg-def* **by** *auto*

**lemma** *reachable-edges-subset*:  $\text{edges } rg \subseteq \text{edges } g$   
**unfolding** *rg-def* **by** (*rule component-edges-subset*)

**definition** *light-edge*  $C\ u\ v$   
 $\equiv u \in C \wedge v \notin C \wedge (u, v) \in \text{edges } rg$   
 $\wedge (\forall (u', v') \in \text{edges } rg \cap C \times -C. w\ \{u, v\} \leq w\ \{u', v'\})$

**definition** *respects-cut*  $A\ C \equiv A \subseteq C \times C \cup (-C) \times (-C)$

**lemma** *light-edge-is-safe*:  
**fixes**  $A :: ('v \times 'v) \text{ set}$  **and**  $C :: 'v \text{ set}$   
**assumes** *subset-MST*:  $\text{is-subset-MST } w\ rg\ A$   
**assumes** *respects-cut*:  $\text{respects-cut } A\ C$   
**assumes** *light-edge*:  $\text{light-edge } C\ u\ v$   
**shows**  $\text{is-subset-MST } w\ rg\ (\{(v, u)\} \cup A)$

**proof** –

```

have crossing-edge:  $u \in C\ v \notin C\ (u, v) \in \text{edges } rg$ 
and min-edge:  $\forall (u', v') \in \text{edges } rg \cap C \times -C. w\ \{u, v\} \leq w\ \{u', v'\}$ 
using light-edge unfolding light-edge-def by auto

```

**from** *subset-MST* **obtain**  $T$  **where**  $T: \text{is-MST } w\ rg\ T\ A \subseteq \text{edges } T$



**unfolding** *is-subset-MST-def* **by** *auto*  
**hence** *tree T edges T ⊆ edges rg nodes T = nodes rg*  
**by** (*simp-all add: is-MST-def is-spanning-tree-def*)  
**hence** *connected T* **by**(*simp-all add: tree-def*)  
**show** *?thesis*  
**proof** *cases*  
**assume**  $(u,v) \in \text{edges } T$   
**thus** *?thesis unfolding is-subset-MST-def using T by (auto simp: edges-sym')*  
**next**  
**assume**  $(u,v) \notin \text{edges } T$  **hence**  $(v,u) \notin \text{edges } T$  **by** (*auto simp: edges-sym'*)  
**from**  $\langle (u,v) \in \text{edges } rg \rangle$  **obtain**  $p$  **where**  $p: \text{path } T \ u \ p \ v$  *simple p*  
**by** (*metis connectedD ‹connected T› ‹nodes T = nodes rg› nodesI*  
*rtrancl-edges-iff-path simplify-pathE*)  
  
**have** [*simp*]:  $u \neq v$  **using** *crossing-edge* **by** *blast*  
  
**from** *find-crossing-edge-on-path[OF p(1), where P = λx. x ∉ C]*  
*crossing-edge(1,2)*  
**obtain**  $x \ y \ p1 \ p2$  **where**  $xy: (x,y) \in \text{set } p \ x \in C \ y \notin C$   
**and**  $ux: \text{path } (\text{restrict-edges } T \ (-\{(x,y),(y,x)\})) \ u \ p1 \ x$   
**and**  $yv: \text{path } (\text{restrict-edges } T \ (-\{(x,y),(y,x)\})) \ y \ p2 \ v$   
**using** *path-change[OF crossing-edge(1,2) p]* **by** *blast*  
**have**  $(x,y) \in \text{edges } T$   
**by** (*meson contra-subsetD p(1) path-edges xy(1)*)  
  
**let**  $?E' = \text{edges } T - \{(x,y),(y,x)\}$   
  
**from** *split-tree[OF ‹tree T› ‹(x,y) ∈ edges T›]*  
**obtain**  $T1 \ T2$  **where**  $T12$ :  
*tree T1 tree T2*  
**and**  $\text{nodes } T1 \cap \text{nodes } T2 = \{\}$   
**and**  $\text{nodes } T = \text{nodes } T1 \cup \text{nodes } T2$   
**and**  $\text{edges } T1 \cup \text{edges } T2 = ?E'$   
**and**  $\text{nodes } T1 = \{ u . (x,u) \in ?E'^* \}$   
**and**  $\text{nodes } T2 = \{ u . (y,u) \in ?E'^* \}$   
**and**  $x \in \text{nodes } T1 \ y \in \text{nodes } T2$  .  
  
**let**  $?T' = \text{ins-edge } (u,v) \ (\text{graph-join } T1 \ T2)$   
  
**have** *is-spanning-tree rg ?T'* **proof** –  
  
**have**  $E'\text{-sym}: \text{sym } (?E'^*)$   
**by** (*meson edgesT-diff-sng-inv-eq sym-conv-converse-eq sym-rtrancl*)  
  
**have**  $u \in \text{nodes } T1$   
**unfolding**  $\langle \text{nodes } T1 = - \rangle$   
**using** *path-rtrancl-edgesD[OF ux]* **by** (*auto dest: symD[OF E'-sym]*)  
  
**have**  $v \in \text{nodes } T2$

```

unfolding ⟨nodes T2 = -⟩
using path-rtrancl-edgesD[OF yv] by auto

have tree ?T' by (rule join-trees) fact+

show is-spanning-tree rg ?T'
  unfolding is-spanning-tree-def
  using ⟨nodes T = nodes rg⟩ ⟨nodes T = nodes T1 ∪ nodes T2⟩[symmetric]
  using ⟨tree ?T'⟩ ⟨u≠v⟩
  using ⟨edges T ⊆ edges rg⟩ ⟨edges T1 ∪ edges T2 = ?E'⟩
  apply simp
  by (metis Diff-subset crossing-edge(3) edges-sym' insert-absorb
        nodesI(2) subset-trans)
qed
moreover

have weight w ?T' ≤ weight w T' if is-spanning-tree rg T' for T'
proof -
  have ww: w {u,v} ≤ w{x,y}
    using min-edge ⟨(x,y)∈edges T⟩ ⟨edges T ⊆ edges rg⟩ ⟨x∈C⟩ ⟨y∉C⟩
    by blast

  have weight w ?T' = weight w T - w {x,y} + w{u,v}
    using ⟨(u, v) ∉ edges T⟩ ⟨(x, y) ∈ edges T⟩
    using ⟨edges T1 ∪ edges T2 = edges T - {(x, y), (y, x)}⟩ ⟨u ≠ v⟩
    by (smt Diff-eq Diff-subset add commute contra-subsetD edges-join
          edges-restrict-edges minus-inv-sym-aux sup.idem weight-cong
          weight-del-edge weight-ins-edge)

  also have ... ≤ weight w T
    using weight-ge-edge[OF ⟨(x,y)∈edges T⟩, of w] ww by auto
  also have weight w T ≤ weight w T' using T(1) ⟨is-spanning-tree rg T'⟩
    unfolding is-MST-def by simp
  finally show ?thesis .
qed
ultimately have is-MST w rg ?T' using is-MST-def by blast
have {(u,v),(v,u)} ∪ A ⊆ edges ?T'
  using T(2) respects-cut xy(2,3) ⟨edges T1 ∪ edges T2 = ?E'⟩
  unfolding respects-cut-def
  by auto

  with ⟨is-MST w rg ?T'⟩ show ?thesis unfolding is-subset-MST-def by force
qed
qed

end

```

## 1.4.2 Abstract Prim: Growing a Tree

**context** Prim **begin**

The current nodes

**definition**  $S A \equiv \{r\} \cup fst'A \cup snd'A$

**lemma** *respects-cut'*:  $A \subseteq S A \times S A$

**unfolding** *S-def* **by** *force*

**corollary** *respects-cut*: *respects-cut*  $A (S A)$

**unfolding** *respects-cut-def* **using** *respects-cut'* **by** *auto*

Refined invariant: Adds connectedness of  $A$

**definition** *prim-invar1*  $A \equiv is-subset-MST w rg A \wedge (\forall (u,v) \in A. (v,r) \in A^*)$

Measure: Number of nodes not in tree

**definition** *T-measure1*  $A = card (nodes rg - S A)$

**end**

We use a locale that fixes a state and assumes the invariant

**locale** *Prim-Invar1-loc* =

*Prim w g r* **for** *w g* **and** *r* :: ' $v$  +

**fixes**  $A$  :: (' $v \times v$ ) *set*

**assumes** *invar1*: *prim-invar1*  $A$

**begin**

**lemma** *subset-MST*: *is-subset-MST*  $w rg A$

**using** *invar1* **unfolding** *prim-invar1-def* **by** *auto*

**lemma** *A-connected*:  $(u,v) \in A \implies (v,r) \in A^*$

**using** *invar1* **unfolding** *prim-invar1-def* **by** *auto*

**lemma** *S-alt-def*:  $S A = \{r\} \cup fst'A$

**unfolding** *S-def*

**apply** (*safe;simp*)

**by** (*metis A-connected Domain-fst Not-Domain-rtrancl*)

**lemma** *finite-rem-nodes*[*simp,intro!*]: *finite*  $(nodes rg - S A)$  **by** *auto*

**lemma** *A-edges*:  $A \subseteq edges g$

**using** *subset-MST*

**by** (*meson is-MST-def is-spanning-tree-def is-subset-MST-def reachable-edges-subset subset-eq*)

**lemma** *S-reachable*:  $S A \subseteq nodes rg$

**unfolding** *S-alt-def*

**by** (*smt DomainE Un-insert-left fst-eq-Domain insert-subset is-MST-def is-spanning-tree-def is-subset-MST-def nodesI(1) nodes-of-component reachable-nodes-refl rg-def subset-MST subset-iff sup-bot.left-neutral*)

**lemma** *S-edge-reachable*:  $\llbracket u \in S A; (u,v) \in edges g \rrbracket \implies (u,v) \in edges rg$

```

using S-reachable unfolding rg-def
using reachable-nodes-step'(2) by fastforce

lemma edges-S-rg-edges:  $edges\ g \cap S\ A \times -S\ A = edges\ rg \cap S\ A \times -S\ A$ 
using S-edge-reachable reachable-edges-subset by auto

lemma T-measure1-less:  $T-measure1\ A < card\ (nodes\ rg)$ 
unfolding T-measure1-def S-def
by (metis Diff-subset S-def S-reachable Un-insert-left le-supE nodes-finite
      psubsetI psubset-card-mono singletonI subset-Diff-insert)

lemma finite-A[simp, intro!]: finite A
using A-edges finite-subset by auto

lemma finite-S[simp, intro!]: finite (S A)
using S-reachable rev-finite-subset by blast

lemma S-A-consistent[simp, intro!]: nodes-edges-consistent (S A) (AUA-1)
unfolding nodes-edges-consistent-def
apply (intro conjI)
subgoal by simp
subgoal using A-edges irreft-def by fastforce
subgoal by (simp add: sym-Un-converse)
using respects-cut' by auto

end

context Prim begin

lemma invar1-initial: prim-invar1 {}
by (auto simp: is-subset-MST-def prim-invar1-def exists-MST)

lemma maintain-invar1:
assumes invar: prim-invar1 A
assumes light-edge: light-edge (S A) u v
shows prim-invar1 ({(v,u)} $\cup$ A)
       $\wedge T-measure1\ ({(v,u)}\cup A) < T-measure1\ A$  (is ?G1  $\wedge$  ?G2)
proof

from invar interpret Prim-Invar1-loc w g r A by unfold-locales

from light-edge have  $u \in S\ A\ v \notin S\ A$  by (simp-all add: light-edge-def)

show ?G1
unfolding prim-invar1-def
proof (intro conjI)

```

**show** *is-subset-MST* *w rg* ( $\{(v, u)\} \cup A$ )  
**by** (*rule light-edge-is-safe*[*OF subset-MST respects-cut light-edge*])

**next**

**show**  $\forall (ua, va) \in \{(v, u)\} \cup A. (va, r) \in (\{(v, u)\} \cup A)^*$   
**apply** *safe*  
**subgoal**  
**using** *A-connected*  
**by** (*simp add: rtrancl-insert*)  
*(metis DomainE S-alt-def converse-rtrancl-into-rtrancl ‹u∈S A›*  
*fst-eq-Domain insertE insert-is-Un rtrancl-eq-or-trancl)*  
**subgoal using** *A-connected* **by** (*simp add: rtrancl-insert*)  
**done**

**qed**

**then interpret** *N: Prim-Invar1-loc w g r*  $\{(v, u)\} \cup A$  **by** *unfold-locales*

**have**  $S A \subset S (\{(v, u)\} \cup A)$  **using**  $\langle v \notin S A \rangle$   
**unfolding** *S-def* **by** *auto*  
**then show** *?G2* **unfolding** *T-measure1-def*  
**using** *S-reachable N.S-reachable*  
**by** (*auto intro!: psubset-card-mono*)

**qed**

**lemma** *invar1-finish:*

**assumes** *INV: prim-invar1 A*  
**assumes** *FIN: edges g ∩ S A × -S A = {}*  
**shows** *is-MST w rg (graph {r} A)*

**proof** –

**from** *INV* **interpret** *Prim-Invar1-loc w g r A* **by** *unfold-locales*

**from** *subset-MST* **obtain** *t* **where** *MST: is-MST w rg t* **and**  $A \subseteq \text{edges } t$   
**unfolding** *is-subset-MST-def* **by** *auto*

**have**  $S A = \text{nodes } t$

**proof** *safe*

**fix** *u*  
**show**  $u \in S A \implies u \in \text{nodes } t$  **using** *MST*  
**unfolding** *is-MST-def is-spanning-tree-def*  
**using** *S-reachable* **by** *auto*

**next**

**fix** *u*  
**assume**  $u \in \text{nodes } t$   
**hence**  $u \in \text{nodes } rg$   
**using** *MST is-MST-def is-spanning-tree-def* **by** *force*  
**hence**  $1: (u, r) \in (\text{edges } rg)^*$  **by** (*simp add: connectedD rg-def*)  
**have**  $r \in S A$  **by** (*simp add: S-def*)  
**show**  $u \in S A$  **proof** (*rule ccontr*)  
**assume**  $u \notin S A$

```

from find-crossing-edge-rtrancl[where  $P = \lambda u. u \in S \ A, \text{ OF } 1 \ \langle u \notin S \ A \rangle \ \langle r \in S$ 
A>]
  FIN reachable-edges-subset
show False
  by (smt ComplI IntI contra-subsetD edges-sym' emptyE mem-Sigma-iff)

qed
qed
also have nodes  $t = \text{nodes } rg$ 
  using MST unfolding is-MST-def is-spanning-tree-def
  by auto
finally have S-eq:  $S \ A = \text{nodes } rg$  .

define  $t'$  where  $t' = \text{graph } \{r\} \ A$ 

have [simp]: nodes  $t' = S \ A$  and  $Et'$ : edges  $t' = (A \cup A^{-1})$  unfolding  $t'$ -def
  using A-edges
  by (auto simp: graph-accs S-def)

hence edges  $t' \subseteq \text{edges } t$ 
  by (smt UnE  $\langle A \subseteq \text{edges } t \rangle$  converseD edges-sym' subrelI subset-eq)

have is-spanning-tree  $rg \ t'$ 
proof -
  have connected  $t'$ 
    apply rule
    apply (simp add:  $Et' \ S$ -def)
    apply safe
    apply ((simp add: A-connected converse-rtrancl-into-rtrancl
      in-rtrancl-UnI rtrancl-converse
    )+
  ) [4]
    apply simp-all [4]
    apply ((meson A-connected in-rtrancl-UnI r-into-rtrancl
      rtrancl-converseI rtrancl-trans
    )+
  ) [4]
  done

moreover have cycle-free  $t'$ 
  by (meson MST  $\langle \text{edges } t' \subseteq \text{edges } t \rangle$  cycle-free-antimono is-MST-def
    is-spanning-tree-def tree-def)
moreover have edges  $t' \subseteq \text{edges } rg$ 
  by (meson MST  $\langle \text{edges } t' \subseteq \text{edges } t \rangle$  dual-order.trans is-MST-def
    is-spanning-tree-def)
ultimately show ?thesis
  unfolding is-spanning-tree-def tree-def
  by (auto simp: S-eq)
qed

```

**then show** *?thesis*  
**using** *MST weight-mono*[*OF*  $\langle \text{edges } t' \subseteq \text{edges } t \rangle$ ]  
**unfolding** *t'-def is-MST-def*  
**using** *dual-order.trans* **by** *blast*  
**qed**  
**end**

### 1.4.3 Prim: Using a Priority Queue

We define a new locale. Note that we could also reuse *Prim*, however, this would complicate referencing the constants later in the theories from which we generate the paper.

**locale** *Prim2* = *Prim w g r* **for**  $w :: 'v \text{ set} \Rightarrow \text{nat}$  **and**  $g :: 'v \text{ ugraph}$  **and**  $r :: 'v$   
**begin**

Abstraction to edge set

**definition**  $A \ Q \ \pi \equiv \{(u,v). \ \pi \ u = \text{Some } v \wedge Q \ u = \infty\}$

Initialization

**definition**  $\text{init}Q :: 'v \Rightarrow \text{enat}$  **where**  $\text{init}Q \equiv (\lambda-. \ \infty)(r := 0)$

**definition**  $\text{init}\pi :: 'v \Rightarrow 'v \text{ option}$  **where**  $\text{init}\pi \equiv \text{Map.empty}$

Step

**definition**  $\text{upd-cond } Q \ \pi \ u \ v' \equiv$   
 $(v',u) \in \text{edges } g$   
 $\wedge v' \neq r \wedge (Q \ v' = \infty \longrightarrow \pi \ v' = \text{None})$   
 $\wedge \text{enat } (w \ \{v',u\}) < Q \ v'$

State after inner loop

**definition**  $Q_{\text{inter}} \ Q \ \pi \ u \ v'$   
 $= (\text{if } \text{upd-cond } Q \ \pi \ u \ v' \text{ then } \text{enat } (w \ \{v',u\}) \text{ else } Q \ v')$

State after one step

**definition**  $Q' \ Q \ \pi \ u \equiv (Q_{\text{inter}} \ Q \ \pi \ u)(u := \infty)$

**definition**  $\pi' \ Q \ \pi \ u \ v' = (\text{if } \text{upd-cond } Q \ \pi \ u \ v' \text{ then } \text{Some } u \text{ else } \pi \ v')$

**definition**  $\text{prim-invar2-init } Q \ \pi \equiv Q = \text{init}Q \wedge \pi = \text{init}\pi$

**definition**  $\text{prim-invar2-ctd } Q \ \pi \equiv \text{let } A = A \ Q \ \pi; S = S \ A \ \text{in}$   
 $\text{prim-invar1 } A$

$\wedge \pi \ r = \text{None} \wedge Q \ r = \infty$   
 $\wedge (\forall (u,v) \in \text{edges } rg \cap (-S) \times S. \ Q \ u \neq \infty)$   
 $\wedge (\forall u. \ Q \ u \neq \infty \longrightarrow \pi \ u \neq \text{None})$   
 $\wedge (\forall u \ v. \ \pi \ u = \text{Some } v \longrightarrow v \in S \wedge (u,v) \in \text{edges } rg)$   
 $\wedge (\forall u \ v \ d. \ Q \ u = \text{enat } d \wedge \pi \ u = \text{Some } v$   
 $\longrightarrow d = w \ \{u,v\} \wedge (\forall v' \in S. \ (u,v') \in \text{edges } rg \longrightarrow d \leq w \ \{u,v'\}))$

**lemma** *prim-invar2-ctd-alt-aux1*:  
**assumes** *prim-invar1* (*A Q π*)  
**assumes**  $Q\ u \neq \infty\ u \neq r$   
**shows**  $u \notin S\ (A\ Q\ \pi)$   
**proof** –  
**interpret** *Prim-Invar1-loc*  $w\ g\ r\ A\ Q\ \pi$  **by** *unfold-locales fact*  
**show** *?thesis*  
**unfolding** *S-alt-def* **unfolding** *A-def* **using** *assms*  
**by** *auto*  
**qed**

**lemma** *prim-invar2-ctd-alt*:  $prim-invar2-ctd\ Q\ \pi \longleftrightarrow$  (  
*let*  $A = A\ Q\ \pi$ ;  $S = S\ A$ ;  $cE = edges\ rg \cap (-S) \times S$  *in*  
*prim-invar1*  $A$   
 $\wedge\ \pi\ r = None \wedge Q\ r = \infty$   
 $\wedge\ (\forall (u,v) \in cE. Q\ u \neq \infty)$   
 $\wedge\ (\forall u\ v. \pi\ u = Some\ v \longrightarrow v \in S \wedge (u,v) \in edges\ rg)$   
 $\wedge\ (\forall u\ d. Q\ u = enat\ d$   
 $\longrightarrow (\exists v. \pi\ u = Some\ v \wedge d = w\ \{u,v\} \wedge (\forall v'. (u,v') \in cE \longrightarrow d \leq w\ \{u,v'\})))$   
 $)$   
**unfolding** *prim-invar2-ctd-def* *Let-def*  
**using** *prim-invar2-ctd-alt-aux1* [*of*  $Q\ \pi$ ]  
**apply** *safe*  
**subgoal** **by** *auto*  
**subgoal** **by** (*auto* 0 3)  
**subgoal** **by** (*auto* 0 3)  
**subgoal** **by** *clarsimp* (*metis* (*no-types,lifting*) *option.simps*(3))  
**done**

**definition** *prim-invar2*  $Q\ \pi \equiv prim-invar2-init\ Q\ \pi \vee prim-invar2-ctd\ Q\ \pi$

**definition** *T-measure2*  $Q\ \pi$   
 $\equiv$  *if*  $Q\ r = \infty$  *then* *T-measure1* ( $A\ Q\ \pi$ ) *else* *card* (*nodes*  $rg$ )

**lemma** *Q'-init-eq*:  
 $Q'\ initQ\ init\pi\ r = (\lambda u. \text{if } (u,r) \in edges\ rg \text{ then } enat\ (w\ \{u,r\}) \text{ else } \infty)$   
**apply** (*rule ext*)  
**using** *reachable-edges-subset*  
**apply** (*simp add: Q'-def Qinter-def upd-cond-def initQ-def initπ-def*)  
**by** (*auto simp: Prim.rg-def edges-sym' reachable-nodes-step'(2)*)

**lemma** *π'-init-eq*:  
 $\pi'\ initQ\ init\pi\ r = (\lambda u. \text{if } (u,r) \in edges\ rg \text{ then } Some\ r \text{ else } None)$   
**apply** (*rule ext*)  
**using** *reachable-edges-subset*  
**apply** (*simp add: π'-def upd-cond-def initQ-def initπ-def*)



by (auto simp: Prim.rg-def edges-sym' reachable-nodes-step'(2))

**lemma** *A-init-eq*:  $A \text{ init}Q \text{ init}\pi = \{\}$   
unfolding *A-def init $\pi$ -def*  
by *auto*

**lemma** *S-empty*:  $S \{\} = \{r\}$  unfolding *S-def* by (auto simp: *A-init-eq*)

**lemma** *maintain-invar2-first-step*:  
assumes *INV*: *prim-invar2-init*  $Q \pi$   
assumes *UNS*:  $Q u = \text{enat } d$   
shows *prim-invar2-ctd* ( $Q' Q \pi u$ ) ( $\pi' Q \pi u$ ) (is ?G1)  
and *T-measure2* ( $Q' Q \pi u$ ) ( $\pi' Q \pi u$ )  $<$  *T-measure2*  $Q \pi$  (is ?G2)  
**proof** –  
from *INV* have [simp]:  $Q = \text{init}Q \ \pi = \text{init}\pi$   
unfolding *prim-invar2-init-def* by *auto*  
from *UNS* have [simp]:  $u = r$  by (auto simp: *initQ-def split: if-splits*)

**note** *Q'-init-eq  $\pi'$ -init-eq A-init-eq*

**have** [simp]:  $(A (Q' \text{init}Q \text{init}\pi r) (\pi' \text{init}Q \text{init}\pi r)) = \{\}$   
apply (simp add: *Q'-init-eq  $\pi'$ -init-eq*)  
by (auto simp: *A-def split: if-splits*)

**show** ?G1  
apply (simp add: *prim-invar2-ctd-def Let-def invar1-initial*)  
by (auto simp: *Q'-init-eq  $\pi'$ -init-eq S-empty split: if-splits*)

**have** [simp]:  $Q' \text{init}Q \text{init}\pi r r = \infty$   
by (auto simp: *Q'-init-eq*)

**have** [simp]:  $\text{init}Q r = 0$  by (simp add: *initQ-def*)

**show** ?G2  
unfolding *T-measure2-def*  
apply *simp*  
apply (simp add: *T-measure1-def S-empty*)  
by (*metis card-Diff1-less nodes-finite nodes-of-component*  
*reachable-nodes-refl rg-def*)

**qed**

**lemma** *maintain-invar2-first-step-presentation*:  
assumes *INV*: *prim-invar2-init*  $Q \pi$   
assumes *UNS*:  $Q u = \text{enat } d$   
shows *prim-invar2-ctd* ( $Q' Q \pi u$ ) ( $\pi' Q \pi u$ )  
 $\wedge$  *T-measure2* ( $Q' Q \pi u$ ) ( $\pi' Q \pi u$ )  $<$  *T-measure2*  $Q \pi$   
using *maintain-invar2-first-step assms* by *blast*

**end**

Again, we define a locale to fix a state and assume the invariant

```
locale Prim-Invar2-ctd-loc =  
  Prim2 w g r for w g and r :: 'v +  
  fixes Q  $\pi$   
  assumes invar2: prim-invar2-ctd Q  $\pi$   
begin
```

```
sublocale Prim-Invar1-loc w g r A Q  $\pi$   
  using invar2 unfolding prim-invar2-ctd-def  
  apply unfold-locales by (auto simp: Let-def)
```

```
lemma upd-cond-alt: upd-cond Q  $\pi$  u v'  $\longleftrightarrow$   
   $(v',u) \in \text{edges } g \wedge v' \notin S (A Q \pi) \wedge \text{enat } (w \{v',u\}) < Q v'$   
  unfolding upd-cond-def S-alt-def unfolding A-def  
  by (auto simp: fst-eq-Domain)
```

```
lemma  $\pi$ -root:  $\pi r = \text{None}$   
  and Q-root: Q r =  $\infty$   
  and Q-defined:  $\llbracket (u,v) \in \text{edges } rg; u \notin S (A Q \pi); v \in S (A Q \pi) \rrbracket \implies Q u \neq \infty$   
  and  $\pi$ -defined:  $\llbracket Q u \neq \infty \rrbracket \implies \pi u \neq \text{None}$   
  and frontier:  $\pi u = \text{Some } v \implies v \in S (A Q \pi)$   
  and edges:  $\pi u = \text{Some } v \implies (u,v) \in \text{edges } rg$   
  and Q- $\pi$ -consistent:  $\llbracket Q u = \text{enat } d; \pi u = \text{Some } v \rrbracket \implies d = w \{u,v\}$   
  and Q-min: Q u = enat d  
     $\implies (\forall v' \in S (A Q \pi). (u,v') \in \text{edges } rg \longrightarrow d \leq w \{u,v'\})$   
  using invar2 unfolding prim-invar2-ctd-def Let-def by auto
```

```
lemma  $\pi$ -def-on-S:  $\llbracket u \in S (A Q \pi); u \neq r \rrbracket \implies \pi u \neq \text{None}$   
  unfolding S-alt-def  
  unfolding A-def  
  by auto
```

```
lemma  $\pi$ -def-on-edges-to-S:  $\llbracket v \in S (A Q \pi); u \neq r; (u,v) \in \text{edges } rg \rrbracket \implies \pi u \neq \text{None}$   
  apply (cases  $u \in S (A Q \pi)$ )  
  subgoal using  $\pi$ -def-on-S by auto  
  subgoal by (simp add: Q-defined  $\pi$ -defined)  
  done
```

```
lemma Q-min-is-light:  
  assumes UNS: Q u = enat d  
  assumes MIN:  $\forall v. \text{enat } d \leq Q v$   
  obtains v where  $\pi u = \text{Some } v$  light-edge (S (A Q  $\pi$ )) v u  
proof –  
  let ?A = A Q  $\pi$   
  let ?S = S ?A
```

**from** *UNS* **obtain**  $v$  **where**  
 $S1[simp]: \pi u = \text{Some } v \ d = w \ \{u, v\}$   
**using**  $\pi$ -defined  $Q$ - $\pi$ -consistent  
**by** *blast*

**have**  $v \in ?S$  **using** *frontier*[of  $u \ v$ ] **by** *auto*

**have**  $[simp]: u \neq r$  **using**  $\pi$ -root **using**  $S1$  **by** (*auto simp del: S1*)

**have**  $u \notin ?S$  **unfolding**  $S$ -alt-def **unfolding**  $A$ -def **using** *UNS* **by** *auto*

**have**  $(v, u) \in \text{edges } rg$  **using**  $\text{edges}[OF \ S1(1)]$   
**by** (*meson edges-sym' rev-subsetD*)

**have**  $M: \forall (u', v') \in \text{edges } rg \cap ?S \times - ?S. w \ \{v, u\} \leq w \ \{u', v'\}$   
**proof** *safe*  
**fix**  $a \ b$   
**assume**  $(a, b) \in \text{edges } rg \ a \in ?S \ b \notin ?S$   
**hence**  $(b, a) \in \text{edges } rg$  **by** (*simp add: edges-sym'*)

**from**  $Q$ -defined[ $OF \ \langle (b, a) \in \text{edges } rg \rangle \ \langle b \notin ?S \rangle \ \langle a \in ?S \rangle$ ]  
**obtain**  $d'$  **where**  $1: Q \ b = \text{enat } d'$  **by** *blast*  
**with**  $\pi$ -defined **obtain**  $a'$  **where**  $\pi \ b = \text{Some } a'$  **by** *auto*  
**from**  $MIN \ 1$  **have**  $d \leq d'$  **by** (*metis enat-ord-simps(1)*)  
**also from**  $Q$ -min[ $OF \ 1$ ]  $\langle (b, a) \in \text{edges } rg \rangle \ \langle a \in ?S \rangle$  **have**  $d' \leq w \ \{b, a\}$  **by** *blast*  
**finally show**  $w \ \{v, u\} \leq w \ \{a, b\}$  **by** (*simp add: insert-commute*)  
**qed**

**have**  $LE: \text{light-edge } ?S \ v \ u$  **using** *invar1*  $\langle v \in ?S \rangle \ \langle u \notin ?S \rangle \ \langle (v, u) \in \text{edges } rg \rangle \ M$   
**unfolding** *light-edge-def* **by** *blast*

**thus** *thesis* **using** *that* **by** *auto*  
**qed**

**lemma** *maintain-invar-ctd*:  
**assumes**  $UNS: Q \ u = \text{enat } d$   
**assumes**  $MIN: \forall v. \text{enat } d \leq Q \ v$   
**shows** *prim-invar2-ctd*  $(Q' \ Q \ \pi \ u) \ (\pi' \ Q \ \pi \ u)$  **(is**  $?G1$ **)**  
**and**  $T$ -measure2  $(Q' \ Q \ \pi \ u) \ (\pi' \ Q \ \pi \ u) < T$ -measure2  $Q \ \pi$  **(is**  $?G2$ **)**  
**proof** –  
**let**  $?A = A \ Q \ \pi$   
**let**  $?S = S \ ?A$

**from**  $Q$ -min-is-light[ $OF \ UNS \ MIN$ ] **obtain**  $v$  **where**  
 $[simp]: \pi \ u = \text{Some } v$  **and**  $LE: \text{light-edge } ?S \ v \ u$  .

**let**  $?Q' = Q' \ Q \ \pi \ u$   
**let**  $?\pi' = \pi' \ Q \ \pi \ u$

```

let ?A' = A ?Q' ?π'
let ?S' = S ?A'

have NA: ?A' = {(u,v)} ∪ ?A
  unfolding A-def
  unfolding Q'-def π'-def upd-cond-def Qinter-def
  by (auto split: if-splits)

from maintain-invar1[OF invar1 LE]
have prim-invar1 ?A' and M1: T-measure1 ?A' < T-measure1 ?A
  by (auto simp: NA)
then interpret N: Prim-Invar1-loc w g r ?A' by unfold-locales

have [simp]: ?S' = insert u ?S
  unfolding S-alt-def N.S-alt-def
  unfolding Q'-def Qinter-def π'-def upd-cond-def
  unfolding A-def
  by (auto split: if-splits simp: image-iff)

show ?G1
  unfolding prim-invar2-ctd-def Let-def
  apply safe
  subgoal by fact
  subgoal
    unfolding π'-def upd-cond-def
    by (auto simp: π-root)
  subgoal
    by (simp add: Prim2.Q'-def Prim2.Qinter-def Prim2.upd-cond-def Q-root)
  subgoal for a b
    apply simp
    apply safe
    subgoal
      unfolding Q'-def Qinter-def upd-cond-def
      apply (simp add: S-alt-def A-def)
      apply safe
      subgoal using reachable-edges-subset by blast
      subgoal by (simp add: Prim.S-def)
      subgoal by (metis (no-types) A-def Q-defined edges frontier)
      subgoal using not-infinity-eq by fastforce
      done
    subgoal
      unfolding S-alt-def N.S-alt-def
      unfolding A-def Q'-def Qinter-def upd-cond-def
      apply (simp; safe; (auto;fail)?)
      subgoal
        proof –
          assume a1: (a, r) ∈ edges rg
          assume a ∉ fst ' {(u, v). π u = Some v ∧ Q u = ∞}
          then have a ∉ fst ' A Q π

```

```

    by (simp add: A-def)
  then show ?thesis
    using a1
    by (metis (no-types) S-alt-def Q-defined Un-insert-left
        edges-irrefl' insert-iff not-infinity-eq sup-bot.left-neutral)
qed
subgoal by (simp add: fst-eq-Domain)
subgoal
  apply clarsimp
  by (smt Domain.intros Q-defined  $\pi$ -def-on-edges-to-S case-prod-conv
      edges-enat.exhaust frontier fst-eq-Domain mem-Collect-eq
      option.exhaust)
subgoal by (simp add: fst-eq-Domain)
done
done
subgoal
  by (metis Q'-def Qinter-def  $\pi'$ -def  $\pi$ -defined enat.distinct(2)
      fun-upd-apply not-None-eq)

subgoal
  by (metis  $\langle S (A (Q' Q \pi u) (\pi' Q \pi u)) = insert u (S (A Q \pi)) \rangle$   $\pi'$ -def
      frontier insertCI option.inject)
subgoal
  by (metis N.S-edge-reachable upd-cond-def
       $\langle S (A (Q' Q \pi u) (\pi' Q \pi u)) = insert u (S (A Q \pi)) \rangle$   $\pi'$ -def edges
      edges-sym' insertI1 option.inject)
subgoal
  by (smt Q'-def  $\pi'$ -def Q- $\pi$ -consistent Qinter-def fun-upd-apply
      insert-absorb not-enat-eq option.inject the-enat.simps)
subgoal for v' d'
  apply clarsimp
  unfolding Q'-def Qinter-def upd-cond-def
  using Q-min
  apply (clarsimp split: if-splits; safe)
  apply (all  $\langle$ (auto;fail) $\rangle$ ?)
  subgoal by (simp add: le-less less-le-trans)
  subgoal using  $\pi$ -def-on-edges-to-S by auto
  subgoal using reachable-edges-subset by auto
  subgoal by (simp add: Q-root)
done
done
then interpret N: Prim-Invar2-ctd-loc w g r ?Q' ? $\pi'$  by unfold-locales

show ?G2
  unfolding T-measure2-def
  by (auto simp: Q-root N.Q-root M1)

qed

```

**end**

**context** *Prim2* **begin**

**lemma** *maintain-invar2-ctd*:

**assumes** *INV*: *prim-invar2-ctd*  $Q \pi$

**assumes** *UNS*:  $Q u = \text{enat } d$

**assumes** *MIN*:  $\forall v. \text{enat } d \leq Q v$

**shows** *prim-invar2-ctd* ( $Q' Q \pi u$ ) ( $\pi' Q \pi u$ ) (**is** ?*G1*)

**and** *T-measure2* ( $Q' Q \pi u$ ) ( $\pi' Q \pi u$ )  $<$  *T-measure2*  $Q \pi$  (**is** ?*G2*)

**proof** –

**interpret** *Prim-Invar2-ctd-loc*  $w g r Q \pi$  **using** *INV* **by** *unfold-locales*

**from** *maintain-invar-ctd*[*OF UNS MIN*] **show** ?*G1* ?*G2* **by** *auto*

**qed**

**lemma** *Q-min-is-light-presentation*:

**assumes** *INV*: *prim-invar2-ctd*  $Q \pi$

**assumes** *UNS*:  $Q u = \text{enat } d$

**assumes** *MIN*:  $\forall v. \text{enat } d \leq Q v$

**obtains**  $v$  **where**  $\pi u = \text{Some } v \text{ light-edge } (S (A Q \pi)) v u$

**proof** –

**interpret** *Prim-Invar2-ctd-loc*  $w g r Q \pi$  **using** *INV* **by** *unfold-locales*

**from** *Q-min-is-light*[*OF UNS MIN*] **show** ?*thesis* **using** *that* .

**qed**

**lemma** *maintain-invar2-ctd-presentation*:

**assumes** *INV*: *prim-invar2-ctd*  $Q \pi$

**assumes** *UNS*:  $Q u = \text{enat } d$

**assumes** *MIN*:  $\forall v. \text{enat } d \leq Q v$

**shows** *prim-invar2-ctd* ( $Q' Q \pi u$ ) ( $\pi' Q \pi u$ )

$\wedge$  *T-measure2* ( $Q' Q \pi u$ ) ( $\pi' Q \pi u$ )  $<$  *T-measure2*  $Q \pi$

**using** *maintain-invar2-ctd* *assms* **by** *blast*

**lemma** *not-invar2-ctd-init*:

*prim-invar2-init*  $Q \pi \implies \neg \text{prim-invar2-ctd } Q \pi$

**unfolding** *prim-invar2-init-def* *prim-invar2-ctd-def* *initQ-def* *Let-def*

**by** (*auto*)

**lemma** *invar2-init-init*: *prim-invar2-init* *initQ* *init* $\pi$

**unfolding** *prim-invar2-init-def* **by** *auto*

**lemma** *invar2-init*: *prim-invar2* *initQ* *init* $\pi$

**unfolding** *prim-invar2-def* **using** *invar2-init-init* **by** *auto*

**lemma** *maintain-invar2*:

**assumes** *A*: *prim-invar2*  $Q \pi$

**assumes** *UNS*:  $Q u = \text{enat } d$

**assumes** *MIN*:  $\forall v. \text{enat } d \leq Q v$

```

shows prim-invar2 (Q' Q π u) (π' Q π u) (is ?G1)
  and T-measure2 (Q' Q π u) (π' Q π u) < T-measure2 Q π (is ?G2)
using A unfolding prim-invar2-def
using maintain-invar2-first-step[of Q, OF - UNS]
using maintain-invar2-ctd[OF - UNS MIN]
using not-invar2-ctd-init
apply blast+
done

```

**lemma** *invar2-ctd-finish*:

```

assumes INV: prim-invar2-ctd Q π
assumes FIN: Q = (λ-. ∞)
shows is-MST w rg (graph {r}) {(u, v). π u = Some v}
proof -
  from INV interpret Prim-Invar2-ctd-loc w g r Q π by unfold-locales

```

```

let ?A = A Q π let ?S=S ?A

```

```

have FC: edges g ∩ ?S × - ?S = {}

```

```

proof (safe; simp)

```

```

  fix a b

```

```

  assume (a,b)∈edges g a∈?S b∉?S

```

```

  with Q-defined[OF edges-sym↑] S-edge-reachable have Q b ≠ ∞

```

```

  by blast

```

```

  with FIN show False by auto

```

```

qed

```

```

have Aeq: ?A = {(u, v). π u = Some v}

```

```

  unfolding A-def using FIN by auto

```

```

from invar1-finish[OF invar1 FC, unfolded Aeq] show ?thesis .
qed

```

**lemma** *invar2-finish*:

```

assumes INV: prim-invar2 Q π

```

```

assumes FIN: Q = (λ-. ∞)

```

```

shows is-MST w rg (graph {r}) {(u, v). π u = Some v}

```

```

proof -

```

```

  from INV have prim-invar2-ctd Q π

```

```

  unfolding prim-invar2-def prim-invar2-init-def initQ-def

```

```

  by (auto simp: fun-eq-iff FIN split: if-splits)

```

```

  with FIN invar2-ctd-finish show ?thesis by blast

```

```

qed

```

```

end

```

## 1.4.4 Refinement of Inner Foreach Loop

context *Prim2* begin

**definition** *foreach-body*  $u \equiv \lambda(v,d) (Q,\pi)$ .

if  $v=r$  then  $(Q,\pi)$

else

case  $(Q v, \pi v)$  of

$(\infty, \text{None}) \Rightarrow (Q(v:=\text{enat } d), \pi(v \mapsto u))$

|  $(\text{enat } d', -) \Rightarrow$  if  $d < d'$  then  $(Q(v:=\text{enat } d), \pi(v \mapsto u))$  else  $(Q,\pi)$

|  $(\infty, \text{Some } -) \Rightarrow (Q,\pi)$

**lemma** *foreach-body-alt*: *foreach-body*  $u = (\lambda(v,d) (Q,\pi))$ .

if  $v \neq r \wedge (\pi v = \text{None} \vee Q v \neq \infty) \wedge \text{enat } d < Q v$  then

$(Q(v:=\text{enat } d), \pi(v \mapsto u))$

else

$(Q,\pi)$

)

**unfolding** *foreach-body-def* *S-def*

**by** (*auto split: enat.splits option.splits simp: fst-eq-Domain fun-eq-iff*)

**definition** *foreach where*

*foreach*  $u$  *adjs*  $Q\pi = \text{foldr } (\text{foreach-body } u) \text{ adjs } Q\pi$

**definition**  $\bigwedge Q V$ .

$Q\text{igen } Q \pi u \text{ adjs } v = (\text{if } v \notin \text{fst'set adjs then } Q v \text{ else } Q\text{inter } Q \pi u v)$

**definition**  $\bigwedge Q V \pi$ .

$\pi'\text{gen } Q \pi u \text{ adjs } v = (\text{if } v \notin \text{fst'set adjs then } \pi v \text{ else } \pi' Q \pi u v)$

context begin

**private lemma** *Qc*:

$Q\text{igen } Q \pi u ((v, w \{u, v\}) \# \text{adjs}) x$

$= (\text{if } x=v \text{ then } Q\text{inter } Q \pi u v \text{ else } Q\text{igen } Q \pi u \text{ adjs } x)$  **for**  $x$

**unfolding** *Qigen-def* **by** *auto*

**private lemma** *πc*:

$\pi'\text{gen } Q \pi u ((v, w \{u, v\}) \# \text{adjs}) x$

$= (\text{if } x=v \text{ then } \pi' Q \pi u v \text{ else } \pi'\text{gen } Q \pi u \text{ adjs } x)$  **for**  $x$

**unfolding** *π'gen-def* **by** *auto*

**lemma** *foreach-refine-gen*:

**assumes**  $\text{set adjs} \subseteq \{(v,d). (u,v) \in \text{edges } g \wedge w \{u,v\} = d\}$

**shows**  $\text{foreach } u \text{ adjs } (Q,\pi) = (Q\text{igen } Q \pi u \text{ adjs}, \pi'\text{gen } Q \pi u \text{ adjs})$

**using** *assms*

**unfolding** *foreach-def*

**proof** (*induction adjs arbitrary: Q π*)

**case** *Nil*

**have** *INVAR-INIT*:  $Q\text{igen } Q \pi u [] = Q \pi'\text{gen } Q \pi u [] = \pi$  **for**  $Q \pi$



```

  unfolding assms Qigen-def π'gen-def
  by (auto simp: fun-eq-iff image-def Q'-def π'-def edges-def)
with Nil show ?case by (simp add: INVAR-INIT)
next
case (Cons a adjs)
obtain v d where [simp]: a=(v,d) by (cases a)

have [simp]: u≠v v≠u using Cons.prems by auto

have QinfD: Qigen Q π u adjs v = ∞ ⇒ Q v = ∞
  unfolding Qigen-def Q'-def Qinter-def by (auto split: if-splits)

show ?case using Cons.prems
  apply (cases a)
  apply (clarsimp simp: Cons.IH)
  unfolding foreach-body-def
  apply (clarsimp; safe)
  subgoal by (auto simp: Qigen-def Qinter-def upd-cond-def)
  subgoal by (auto simp: π'gen-def π'-def upd-cond-def)
  subgoal
    apply (clarsimp split: enat.split option.split simp: πc Qc fun-eq-iff)
    unfolding Qinter-def Qigen-def π'-def π'gen-def upd-cond-def
    apply (safe; simp split: if-splits add: insert-commute)
    by (auto dest: edges-sym')
  done

```

qed

lemma *foreach-refine*:

```

  assumes set adjs = {(v,d). (u,v)∈edges g ∧ w {u,v} = d}
  shows foreach u adjs (Q,π) = (Qinter Q π u,π' Q π u)
proof –
  have INVAR-INIT: Qigen Q π u [] = Q π'gen Q π u [] = π for Q π
  unfolding assms Qigen-def π'gen-def
  by (auto simp: fun-eq-iff image-def Q'-def π'-def edges-def)
  from assms have 1: set adjs ⊆ {(v,d). (u,v)∈edges g ∧ w {u,v} = d}
  by simp
  have [simp]:
    v ∈ fst ‘{(v, d). (u, v) ∈ edges g ∧ w {u, v} = d}
     $\longleftrightarrow (u,v) \in \text{edges } g$ 
  for v
  by force

```

show *?thesis*

```

  unfolding foreach-refine-gen[OF 1]
  unfolding Qigen-def π'gen-def assms upd-cond-def Qinter-def π'-def
  by (auto simp: fun-eq-iff image-def dest: edges-sym')

```

qed

```

end
end

end

```

## 1.5 Implementation of Weighted Undirected Graph by Map

```

theory Undirected-Graph-Impl
imports
  HOL-Data-Structures.Map-Specs
  Common
  Undirected-Graph-Specs
begin

```

### 1.5.1 Doubleton Set to Pair

**definition** *epair*  $e = (if\ card\ e = 2\ then\ Some\ (SOME\ (u,v).\ e=\{u,v\})\ else\ None)$

```

lemma epair-eqD:  $epair\ e = Some\ (x,y) \implies (x \neq y \wedge e = \{x,y\})$ 
apply (cases card e = 2)
unfolding epair-def
apply simp-all
apply (clarsimp simp: card-Suc-eq eval-nat-numeral doubleton-eq-iff)
by (smt case-prodD case-prodI someI)

```

```

lemma epair-not-sng[simp]:  $epair\ e \neq Some\ (x,x)$ 
by (auto dest: epair-eqD)

```

```

lemma epair-None[simp]:  $epair\ \{a,b\} = None \longleftrightarrow a=b$ 
unfolding epair-def by (auto simp: card2-eq)

```

### 1.5.2 Generic Implementation

When instantiated with a map ADT, this locale provides a weighted graph ADT.

```

locale wgraph-by-map =
  M: Map M-empty M-update M-delete M-lookup M-invar

  for M-empty M-update M-delete
  and M-lookup :: 'm  $\Rightarrow$  'v  $\Rightarrow$  (('v  $\times$  nat) list) option
  and M-invar
begin

```

```

definition  $\alpha nodes\text{-aux}\ g \equiv dom\ (M\text{-lookup}\ g)$ 

```

```

definition  $\alpha edges\text{-aux}\ g$ 

```

$\equiv (\{(u,v). \exists xs d. M\text{-lookup } g \ u = \text{Some } xs \wedge (v,d) \in \text{set } xs \})$

**definition**  $\alpha g \ g \equiv \text{graph } (\alpha \text{nodes-aux } g) (\alpha \text{edges-aux } g)$

**definition**  $\alpha w \ g \ e \equiv \text{case epair } e \text{ of}$

$\text{Some } (u,v) \Rightarrow (\text{case } M\text{-lookup } g \ u \text{ of}$   
 $\text{None} \Rightarrow 0$   
 $| \text{Some } xs \Rightarrow \text{the-default } 0 \ (\text{map-of } xs \ v)$   
 $)$   
 $| \text{None} \Rightarrow 0$

**definition**  $\text{invar} :: 'm \Rightarrow \text{bool}$  **where**

$\text{invar } g \equiv$   
 $M\text{-invar } g \wedge \text{finite } (\text{dom } (M\text{-lookup } g))$   
 $\wedge (\forall u \ xs. M\text{-lookup } g \ u = \text{Some } xs \longrightarrow$   
 $\text{distinct } (\text{map fst } xs)$   
 $\wedge u \notin \text{set } (\text{map fst } xs)$   
 $\wedge (\forall (v,d) \in \text{set } xs. (u,d) \in \text{set } (\text{the-default } [] (M\text{-lookup } g \ v)))$   
 $)$

**lemma**  $\text{in-the-default-empty-conv}[simp]:$

$x \in \text{set } (\text{the-default } [] \ m) \longleftrightarrow (\exists xs. m = \text{Some } xs \wedge x \in \text{set } xs)$   
**by**  $(\text{cases } m) \text{ auto}$

**lemma**  $\alpha \text{edges-irrefl}: \text{invar } g \Longrightarrow \text{irrefl } (\alpha \text{edges-aux } g)$

**unfolding**  $\text{invar-def irrefl-def } \alpha \text{edges-aux-def}$   
**by**  $(\text{force})$

**lemma**  $\alpha \text{edges-sym}: \text{invar } g \Longrightarrow \text{sym } (\alpha \text{edges-aux } g)$

**unfolding**  $\text{invar-def sym-def } \alpha \text{edges-aux-def}$   
**by**  $\text{force}$

**lemma**  $\alpha \text{edges-subset}: \text{invar } g \Longrightarrow \alpha \text{edges-aux } g \subseteq \alpha \text{nodes-aux } g \times \alpha \text{nodes-aux } g$

**unfolding**  $\text{invar-def } \alpha \text{nodes-aux-def } \alpha \text{edges-aux-def}$   
**by**  $\text{force}$

**lemma**  $\alpha \text{nodes-finite}[simp, \text{intro!}]: \text{invar } g \Longrightarrow \text{finite } (\alpha \text{nodes-aux } g)$

**unfolding**  $\text{invar-def } \alpha \text{nodes-aux-def}$  **by**  $\text{simp}$

**lemma**  $\alpha \text{edges-finite}[simp, \text{intro!}]: \text{invar } g \Longrightarrow \text{finite } (\alpha \text{edges-aux } g)$

**using**  $\text{finite-subset}[OF \ \alpha \text{edges-subset}]$  **by**  $\text{blast}$

**definition**  $\text{adj} :: 'm \Rightarrow 'v \Rightarrow ('v \times \text{nat}) \text{ list}$  **where**

$\text{adj } g \ v = \text{the-default } [] (M\text{-lookup } g \ v)$

**definition**  $\text{empty} :: 'm$  **where**  $\text{empty} = M\text{-empty}$

**definition** *add-edge1* :: 'v × 'v ⇒ nat ⇒ 'm ⇒ 'm **where**  
*add-edge1* ≡ λ(u,v) d g. M-update u ((v,d) # the-default [] (M-lookup g u)) g

**definition** *add-edge* :: 'v × 'v ⇒ nat ⇒ 'm ⇒ 'm **where**  
*add-edge* ≡ λ(u,v) d g. add-edge1 (v,u) d (add-edge1 (u,v) d g)

**lemma** *edges-αg-aux*: *invar g* ⇒ *edges (αg g) = αedges-aux g*  
**unfolding** *αg-def* **using** *αedges-sym αedges-irrefl*  
**by** (*auto simp: irrefl-def graph-accs*)

**lemma** *nodes-αg-aux*: *invar g* ⇒ *nodes (αg g) = αnodes-aux g*  
**unfolding** *αg-def* **using** *αedges-subset*  
**by** (*force simp: graph-accs*)

**lemma** *card-doubleton-eq2[simp]*: *card {a,b} = 2* ↔ *a ≠ b* **by** *auto*

**lemma** *the-dflt-Z-eq*: *the-default 0 m = d* ↔ (*m = None* ∧ *d = 0* ∨ *m = Some d*)  
**by** (*cases m*) *auto*

**lemma** *adj-correct-aux*:  
*invar g* ⇒ *set (adj g u) = {(v, d). (u, v) ∈ edges (αg g) ∧ αw g {u, v} = d}*  
**apply** (*simp add: edges-αg-aux*)  
**apply** *safe*  
**subgoal unfolding** *adj-def αedges-aux-def* **by** *auto*  
**subgoal for** *a d*  
**unfolding** *adj-def αw-def*  
**apply** (*clarsimp split: prod.splits option.splits simp: the-dflt-Z-eq*)  
**unfolding** *invar-def*  
**by** (*force dest!: epair-eqD simp: doubleton-eq-iff*)+  
**subgoal for** *a*  
**unfolding** *adj-def αw-def*  
**using** *αedges-irrefl[of g]*  
**apply** (*clarsimp split: prod.splits option.splits*)  
**apply** *safe*  
**subgoal by** (*auto simp: irrefl-def*)  
**subgoal**  
**apply** (*clarsimp dest!: epair-eqD simp: doubleton-eq-iff*)  
**unfolding** *invar-def αedges-aux-def*  
**by** *force*  
**subgoal**  
**apply** (*clarsimp dest!: epair-eqD simp: doubleton-eq-iff*)  
**unfolding** *invar-def αedges-aux-def*  
**apply** *clarsimp*  
**by** (*smt case-prod-conv map-of-is-SomeI the-default.simps(2)*)  
**done**  
**done**

```

lemma invar-empty-aux: invar empty
  by (simp add: invar-def empty-def M.map-specs)

lemma dist-fst-the-dflt-aux: distinct (map fst (the-default [] m))
   $\longleftrightarrow (\forall xs. m = \text{Some } xs \longrightarrow \text{distinct } (\text{map fst } xs))$ 
  by (cases m; auto)

lemma invar-add-edge-aux:
   $\llbracket \text{invar } g; (u, v) \notin \text{edges } (\alpha g g); u \neq v \rrbracket \implies \text{invar } (\text{add-edge } (u, v) d g)$ 
  apply (simp add: edges- $\alpha g$ -aux)
  unfolding add-edge-def add-edge1-def invar-def  $\alpha$ edges-aux-def
  by (auto simp: M.map-specs dist-fst-the-dflt-aux; force)

sublocale adt-wgraph  $\alpha w$   $\alpha g$  invar adj empty add-edge
  apply unfold-locales
  subgoal by (simp add: adj-correct-aux)
  subgoal by (simp add: invar-empty-aux)
  subgoal
    apply (simp
      add: graph-eq-iff nodes- $\alpha g$ -aux invar-empty-aux edges- $\alpha g$ -aux
      add:  $\alpha$ nodes-aux-def  $\alpha$ edges-aux-def
    )
    apply (simp add: empty-def M.map-specs)
  done
  subgoal
    unfolding  $\alpha w$ -def
    by (auto simp: empty-def M.map-specs fun-eq-iff split: option.splits)
  subgoal by (simp add: invar-add-edge-aux)
  subgoal for g u v d
    apply (simp add: edges- $\alpha g$ -aux nodes- $\alpha g$ -aux graph-eq-iff invar-add-edge-aux)
    apply (rule conjI)
    subgoal
      unfolding add-edge-def add-edge1-def invar-def  $\alpha$ nodes-aux-def
      by (auto simp: M.map-specs)
    subgoal
      unfolding add-edge-def add-edge1-def invar-def  $\alpha$ edges-aux-def
      by (fastforce simp: M.map-specs split!: if-splits)
    done
  subgoal for g u v d
    apply (simp add: edges- $\alpha g$ -aux invar-add-edge-aux)
    unfolding invar-def  $\alpha w$ -def add-edge-def add-edge1-def
    by (auto
      dest: epair-eqD
      simp: fun-eq-iff M.map-specs
      split!: prod.splits option.splits if-splits)
  done

```

end

end

## 1.6 Implementation of Prim's Algorithm

**theory** *Prim-Impl*

**imports**

*Prim-Abstract*

*Undirected-Graph-Impl*

*HOL-Library.While-Combinator*

*Priority-Search-Trees.PST-RBT*

*HOL-Data-Structures.RBT-Map*

**begin**

### 1.6.1 Implementation using ADT Interfaces

**locale** *Prim-Impl-Adts* =

*G*: *adt-wgraph* *G- $\alpha$ w* *G- $\alpha$ g* *G-invar* *G-adj* *G-empty* *G-add-edge*

+ *M*: *Map* *M-empty* *M-update* *M-delete* *M-lookup* *M-invar*

+ *Q*: *PrioMap* *Q-empty* *Q-update* *Q-delete* *Q-invar* *Q-lookup* *Q-is-empty* *Q-getmin*

**for** *typG* :: '*g* *itself* **and** *typM* :: '*m* *itself* **and** *typQ* :: '*q* *itself*

**and** *G- $\alpha$ w* **and** *G- $\alpha$ g* :: '*g*  $\Rightarrow$  ('*v*) *ugraph* **and** *G-invar* *G-adj* *G-empty* *G-add-edge*

**and** *M-empty* *M-update* *M-delete* **and** *M-lookup* :: '*m*  $\Rightarrow$  '*v*  $\Rightarrow$  '*v* *option* **and**  
*M-invar*

**and** *Q-empty* *Q-update* *Q-delete* *Q-invar* **and** *Q-lookup* :: '*q*  $\Rightarrow$  '*v*  $\Rightarrow$  *nat option*

**and** *Q-is-empty* *Q-getmin*

**begin**

Simplifier setup

**lemmas** [*simp*] = *G.wgraph-specs*

**lemmas** [*simp*] = *M.map-specs*

**lemmas** [*simp*] = *Q.prio-map-specs*

**end**

**locale** *Prim-Impl-Defs* = *Prim-Impl-Adts*

**where** *typG* = *typG* **and** *typM* = *typM* **and** *typQ* = *typQ* **and** *G- $\alpha$ w* = *G- $\alpha$ w*

**and** *G- $\alpha$ g* = *G- $\alpha$ g*

**for** *typG* :: '*g* *itself* **and** *typM* :: '*m* *itself* **and** *typQ* :: '*q* *itself*

**and** *G- $\alpha$ w* **and** *G- $\alpha$ g* :: '*g*  $\Rightarrow$  ('*v*::*linorder*) *ugraph* **and** *g* :: '*g* **and** *r* :: '*v*

**begin**

## Concrete Algorithm

**term** *M-lookup*

**definition** *foreach-impl-body*  $u \equiv (\lambda(v,d) (Qi,\pi i).$

*if*  $v=r$  *then*  $(Qi,\pi i)$

*else*

*case*  $(Q\text{-lookup } Qi \ v, M\text{-lookup } \pi i \ v)$  *of*

$(None, None) \Rightarrow (Q\text{-update } v \ d \ Qi, M\text{-update } v \ u \ \pi i)$

|  $(Some \ d', -) \Rightarrow (if \ d < d' \ then \ (Q\text{-update } v \ d \ Qi, M\text{-update } v \ u \ \pi i) \ else \ (Qi, \pi i))$

|  $(None, Some \ -) \Rightarrow (Qi, \pi i)$

)

**definition** *foreach-impl*  $:: 'q \Rightarrow 'm \Rightarrow 'v \Rightarrow ('v \times nat) \ list \Rightarrow 'q \times 'm$  **where**

*foreach-impl*  $Qi \ \pi i \ u \ adjs = foldr \ (foreach\text{-impl}\text{-body } u) \ adjs \ (Qi, \pi i)$

**definition** *outer-loop-impl*  $Qi \ \pi i \equiv while \ (\lambda(Qi, \pi i). \neg Q\text{-is-empty } Qi) \ (\lambda(Qi, \pi i).$

*let*

$(u, -) = Q\text{-getmin } Qi;$

$adjs = G\text{-adj } g \ u;$

$(Qi, \pi i) = foreach\text{-impl } Qi \ \pi i \ u \ adjs;$

$Qi = Q\text{-delete } u \ Qi$

*in*  $(Qi, \pi i) \ (Qi, \pi i)$

**definition** *prim-impl*  $= (let$

$Qi = Q\text{-update } r \ 0 \ Q\text{-empty};$

$\pi i = M\text{-empty};$

$(Qi, \pi i) = outer\text{-loop}\text{-impl } Qi \ \pi i$

*in*  $\pi i)$

The whole algorithm as one function

**lemma** *prim-impl-alt*: *prim-impl*  $= (let$

— Initialization

$(Q, \pi) = (Q\text{-update } r \ 0 \ Q\text{-empty}, M\text{-empty});$

— Main loop: Iterate until PQ is empty

$(Q, \pi) =$

*while*  $(\lambda(Q, \pi). \neg Q\text{-is-empty } Q) \ (\lambda(Q, \pi). let$

$(u, -) = Q\text{-getmin } Q;$

— Inner loop: Update for adjacent nodes

$(Q, \pi) =$

*foldr*  $((\lambda(v, d) (Q, \pi). let$

$qv = Q\text{-lookup } Q \ v;$

$\pi v = M\text{-lookup } \pi \ v$

*in*

*if*  $v \neq r \wedge (qv \neq None \vee \pi v = None) \wedge enat \ d < enat\text{-of-option } qv$

*then*  $(Q\text{-update } v \ d \ Q, M\text{-update } v \ u \ \pi)$

*else*  $(Q, \pi)$

)  $(G\text{-adj } g \ u) \ (Q, \pi);$

$Q = Q\text{-delete } u \ Q$

```

    in (Q, π) (Q, π)
  in π
)
proof –

have 1: foreach-impl-body u = (λ(v,d) (Qi,πi). let
  qiv = (Q-lookup Qi v);
  πiv = M-lookup πi v
  in
  if v≠r ∧ (qiv≠None ∨ πiv=None) ∧ enat d < enat-of-option qiv
  then (Q-update v d Qi, M-update v u πi)
  else (Qi, πi)) for u
unfolding foreach-impl-body-def
apply (intro ext)
by (auto split: option.split)

show ?thesis
unfolding prim-impl-def outer-loop-impl-def foreach-impl-def 1
by (simp)
qed

```

## Abstraction of Result

Invariant for the result, and its interpretation as (minimum spanning) tree:

- The map  $\pi i$  and set  $V_i$  satisfy their implementation invariants
- The  $\pi i$  encodes irreflexive edges consistent with the nodes determined by  $V_i$ . Note that the edges in  $\pi i$  will not be symmetric, thus we take their symmetric closure  $E \cup E^{-1}$ .

**definition** *invar-MST*  $\pi i \equiv M\text{-invar } \pi i$

**definition**  $\alpha\text{-MST } \pi i \equiv \text{graph } \{r\} \{(u,v) \mid u v. M\text{-lookup } \pi i u = \text{Some } v\}$

**end**

### 1.6.2 Refinement of State

**locale** *Prim-Impl* = *Prim-Impl-Defs*

**where**  $\text{typ}G = \text{typ}G$  **and**  $\text{typ}M = \text{typ}M$  **and**  $\text{typ}Q = \text{typ}Q$  **and**  $G\text{-}\alpha w = G\text{-}\alpha w$   
**and**  $G\text{-}\alpha g = G\text{-}\alpha g$

**for**  $\text{typ}G :: 'g \text{ itself}$  **and**  $\text{typ}M :: 'm \text{ itself}$  **and**  $\text{typ}Q :: 'q \text{ itself}$

**and**  $G\text{-}\alpha w$  **and**  $G\text{-}\alpha g :: 'g \Rightarrow ('v::\text{linorder}) \text{ ugraph}$

+

**assumes**  $G\text{-invar}[\text{simp}]: G\text{-invar } g$

**begin**

**sublocale** *Prim2*  $G\text{-}\alpha w g G\text{-}\alpha g g r$  .



## Abstraction of $Q$

The priority map implements a function of type  $'v \Rightarrow \text{enat}$ , mapping  $\text{None}$  to  $\infty$ .

**definition**  $Q\text{-}\alpha$   $Qi \equiv \text{enat-of-option } o \text{ } Q\text{-lookup } Qi :: 'v \Rightarrow \text{enat}$

**lemma**  $Q\text{-}\alpha\text{-empty}$ :  $Q\text{-}\alpha \text{ } Q\text{-empty} = (\lambda\cdot. \infty)$   
**unfolding**  $Q\text{-}\alpha\text{-def}$  **by**  $(\text{auto})$

**lemma**  $Q\text{-}\alpha\text{-update}$ :  $Q\text{-invar } Q \Longrightarrow Q\text{-}\alpha (Q\text{-update } u \ d \ Q) = (Q\text{-}\alpha \ Q)(u := \text{enat } d)$   
**unfolding**  $Q\text{-}\alpha\text{-def}$  **by**  $(\text{auto})$

**lemma**  $Q\text{-}\alpha\text{-is-empty}$ :  $Q\text{-invar } Q \Longrightarrow Q\text{-lookup } Q = \text{Map.empty} \longleftrightarrow Q\text{-}\alpha \ Q = (\lambda\cdot. \infty)$   
**unfolding**  $Q\text{-}\alpha\text{-def}$  **by**  $(\text{auto simp: fun-eq-iff})$

**lemma**  $Q\text{-}\alpha\text{-delete}$ :  $Q\text{-invar } Q \Longrightarrow Q\text{-}\alpha (Q\text{-delete } u \ Q) = (Q\text{-}\alpha \ Q)(u := \infty)$   
**unfolding**  $Q\text{-}\alpha\text{-def}$  **by**  $(\text{auto simp: fun-eq-iff})$

**lemma**  $Q\text{-}\alpha\text{-min}$ :  
**assumes**  $MIN$ :  $Q\text{-getmin } Qi = (u, d)$   
**assumes**  $I$ :  $Q\text{-invar } Qi$   
**assumes**  $NE$ :  $\neg Q\text{-is-empty } Qi$   
**shows**  $Q\text{-}\alpha \ Qi \ u = \text{enat } d$  (**is**  $?G1$ ) **and**  
 $\forall v. \text{enat } d \leq Q\text{-}\alpha \ Qi \ v$  (**is**  $?G2$ )

**proof** –

**from**  $Q.\text{map-getmin}[OF \ MIN]$   
**have**  $Q\text{-lookup } Qi \ u = \text{Some } d$  ( $\forall x \in \text{ran } (Q\text{-lookup } Qi). \ d \leq x$ )  
**using**  $NE \ I$  **by**  $\text{auto}$   
**thus**  $?G1 \ ?G2$   
**unfolding**  $Q\text{-}\alpha\text{-def}$  **apply**  $\text{simp-all}$   
**by**  $(\text{metis enat-of-option.elims enat-ord-simps(1) enat-ord-simps(3) ranI})$   
**qed**

**lemmas**  $Q\text{-}\alpha\text{-specs} = Q\text{-}\alpha\text{-empty } Q\text{-}\alpha\text{-update } Q\text{-}\alpha\text{-is-empty } Q\text{-}\alpha\text{-delete}$

## Concrete Invariant

The implementation invariants of the concrete state's components, and the abstract invariant of the state's abstraction

**definition**  $\text{prim-invar-impl } Qi \ \pi i \equiv$   
 $Q\text{-invar } Qi \ \wedge \ M\text{-invar } \pi i \ \wedge \ \text{prim-invar2 } (Q\text{-}\alpha \ Qi) \ (M\text{-lookup } \pi i)$

**end**

### 1.6.3 Refinement of Algorithm

context *Prim-Impl*

begin

lemma *foreach-impl-correct*:

fixes  $Qi\ Vi\ \pi i$  defines  $Q \equiv Q-\alpha\ Qi$  and  $\pi \equiv M\text{-lookup}\ \pi i$

assumes  $A$ : *foreach-impl*  $Qi\ \pi i\ u\ (G\text{-adj}\ g\ u) = (Qi', \pi i')$

assumes  $I$ : *prim-invar-impl*  $Qi\ \pi i$

shows *Q-invar*  $Qi'$  and *M-invar*  $\pi i'$

and  $Q-\alpha\ Qi' = Q\text{-inter}\ Q\ \pi\ u$  and  $M\text{-lookup}\ \pi i' = \pi' Q\ \pi\ u$

proof –

from  $I$  have [*simp*]: *Q-invar*  $Qi\ M\text{-invar}\ \pi i$

unfolding *prim-invar-impl-def* *Q-def*  $\pi\text{-def}$  by *auto*

{  
 fix  $Qi\ \pi i\ d\ v$  and  $adjs :: ('v \times nat)$  list  
 assume *Q-invar*  $Qi\ M\text{-invar}\ \pi i\ (v, d) \in \text{set}\ adjs$   
 then have  
 (case *foreach-impl-body*  $u\ (v, d)\ (Qi, \pi i)$  of  
 ( $Qi, \pi i$ )  $\Rightarrow$  *Q-invar*  $Qi \wedge M\text{-invar}\ \pi i$   
 $\wedge$  *map-prod*  $Q-\alpha\ M\text{-lookup}\ (\text{foreach-impl-body}\ u\ (v, d)\ (Qi, \pi i))$   
 $=$  *foreach-body*  $u\ (v, d)\ (Q-\alpha\ Qi, M\text{-lookup}\ \pi i)$   
 unfolding *foreach-impl-body-def* *foreach-body-def*  
 unfolding *Q-alpha-def*  
 by (*auto simp: fun-eq-iff split: option.split*)

} note *aux=this*

from *foldr-refine*[

where  $I = \lambda(Qi, \pi i). \text{Q-invar}\ Qi \wedge M\text{-invar}\ \pi i$  and  $\alpha = \text{map-prod}\ Q-\alpha\ M\text{-lookup},$   
 of  $(Qi, \pi i)\ (G\text{-adj}\ g\ u)\ \text{foreach-impl-body}\ u\ \text{foreach-body}\ u$

]

and  $A\ \text{aux}[\text{where}\ ?adjs3 = (G\text{-adj}\ g\ u)]$

have *Q-invar*  $Qi'\ M\text{-invar}\ \pi i'$

and  $1$ : *foreach*  $u\ (G\text{-adj}\ g\ u)\ (Q-\alpha\ Qi, M\text{-lookup}\ \pi i)$   
 $= (Q-\alpha\ Qi', M\text{-lookup}\ \pi i')$

unfolding *foreach-impl-def* *foreach-def*

unfolding *Q-def*  $\pi\text{-def}$

by (*auto split: prod.splits*)

then show *Q-invar*  $Qi'\ M\text{-invar}\ \pi i'$  by *auto*

from  $1$  *foreach-refine*[where  $adjs = G\text{-adj}\ g\ u$  and  $u = u$ ] show

$Q-\alpha\ Qi' = Q\text{-inter}\ Q\ \pi\ u$  and  $M\text{-lookup}\ \pi i' = \pi' Q\ \pi\ u$

by (*auto simp: Q-def*  $\pi\text{-def}$ )

qed

definition *T-measure-impl*  $\equiv \lambda(Qi, \pi i). T\text{-measure2}\ (Q-\alpha\ Qi)\ (M\text{-lookup}\ \pi i)$

**lemma** *prim-invar-impl-init*: *prim-invar-impl* (*Q-update* *r* 0 *Q-empty*) *M-empty*  
**using** *invar2-init*  
**by** (*auto simp*: *prim-invar-impl-def Q- $\alpha$ -specs initQ-def init $\pi$ -def zero-enat-def*)

**lemma** *maintain-prim-invar-impl*:

**assumes**

*I*: *prim-invar-impl* *Qi*  $\pi$  *i* **and**

*NE*:  $\neg$  *Q-is-empty* *Qi* **and**

*MIN*: *Q-getmin* *Qi* = (*u*, *d*) **and**

*FOREACH*: *foreach-impl* *Qi*  $\pi$  *i* *u* (*G-adj* *g* *u*) = (*Qi'*,  $\pi$  *i'*)

**shows** *prim-invar-impl* (*Q-delete* *u* *Qi'*)  $\pi$  *i'* (**is** ?*G1*)

**and** *T-measure-impl* (*Q-delete* *u* *Qi'*,  $\pi$  *i'*) < *T-measure-impl* (*Qi*,  $\pi$  *i*) (**is** ?*G2*)

**proof** –

**note** *II*[*simp*] = *I*[*unfolded prim-invar-impl-def*]

**note** *FI*[*simp*] = *foreach-impl-correct*[*OF FOREACH I*]

**note** *MIN'* = *Q- $\alpha$ -min*[*OF MIN - NE, simplified*]

**show** ?*G1*

**unfolding** *prim-invar-impl-def*

**using** *Q- $\alpha$ -delete maintain-invar2*[*OF - MIN*]

**by** (*simp add*: *Q'-def*)

**show** ?*G2*

**unfolding** *prim-invar-impl-def T-measure-impl-def*

**using** *Q- $\alpha$ -delete maintain-invar2*[*OF - MIN*]

**apply** (*simp add*: *Q'-def Q- $\alpha$ -def*)

**by** (*metis FI*(3) *II Q'-def Q- $\alpha$ -def*

$\langle \wedge \pi. \text{prim-invar2 } (Q-\alpha \text{ } Qi) \pi$

$\implies T\text{-measure2 } (Q' (Q-\alpha \text{ } Qi) \pi \text{ } u) (\pi' (Q-\alpha \text{ } Qi) \pi \text{ } u)$

$< T\text{-measure2 } (Q-\alpha \text{ } Qi) \pi \rangle$ )

**qed**

**lemma** *maintain-prim-invar-impl-presentation*:

**assumes**

*I*: *prim-invar-impl* *Qi*  $\pi$  *i* **and**

*NE*:  $\neg$  *Q-is-empty* *Qi* **and**

*MIN*: *Q-getmin* *Qi* = (*u*, *d*) **and**

*FOREACH*: *foreach-impl* *Qi*  $\pi$  *i* *u* (*G-adj* *g* *u*) = (*Qi'*,  $\pi$  *i'*)

**shows** *prim-invar-impl* (*Q-delete* *u* *Qi'*)  $\pi$  *i'*

$\wedge$  *T-measure-impl* (*Q-delete* *u* *Qi'*,  $\pi$  *i'*) < *T-measure-impl* (*Qi*,  $\pi$  *i*)

**using** *maintain-prim-invar-impl* *assms* **by** *blast*

**lemma** *prim-invar-impl-finish*:

$\llbracket$  *Q-is-empty* *Q*; *prim-invar-impl* *Q*  $\pi$   $\rrbracket$

$\implies$  *invar-MST*  $\pi \wedge$  *is-MST* (*G- $\alpha$ w* *g*) *rg* ( $\alpha$ -*MST*  $\pi$ )

**using** *invar2-finish*

**by** (*auto simp*: *Q- $\alpha$ -specs prim-invar-impl-def invar-MST-def  $\alpha$ -MST-def Let-def*)

```

lemma prim-impl-correct:
  assumes prim-impl =  $\pi i$ 
  shows
    invar-MST  $\pi i$  (is ?G1)
    is-MST ( $G\text{-}\alpha w$   $g$ ) (component-of ( $G\text{-}\alpha g$   $g$ )  $r$ ) ( $\alpha\text{-MST}$   $\pi i$ ) (is ?G2)
proof -
  have let ( $Qi, \pi i$ ) = outer-loop-impl (Q-update  $r$  0 Q-empty) M-empty in
    invar-MST  $\pi i$   $\wedge$  is-MST ( $G\text{-}\alpha w$   $g$ )  $rg$  ( $\alpha\text{-MST}$   $\pi i$ )
  unfolding outer-loop-impl-def
  apply (rule while-rule[where
     $P = \lambda(Qi, \pi i). \text{prim-invar-impl } Qi \ \pi i$  and  $r = \text{measure } T\text{-measure-impl}$ ])
  apply (all  $\langle \text{clarsimp split: prod.splits simp: } Q\text{-}\alpha\text{-specs} \rangle$ )
  apply (simp-all  $\text{add: prim-invar-impl-init maintain-prim-invar-impl}$ 
     $\text{prim-invar-impl-finish}$ )

  done
  with assms show ?G1 ?G2
  unfolding rg-def prim-impl-def by (simp-all split: prod.splits)
qed

end

```

#### 1.6.4 Instantiation with Actual Data Structures

##### global-interpretation

```

 $G$ : wgraph-by-map RBT-Set.empty RBT-Map.update RBT-Map.delete
  Lookup2.lookup RBT-Map.M.invar
defines  $G\text{-empty}$  =  $G\text{-empty}$ 
  and  $G\text{-add-edge}$  =  $G\text{-add-edge}$ 
  and  $G\text{-add-edge1}$  =  $G\text{-add-edge1}$ 
  and  $G\text{-adj}$  =  $G\text{-adj}$ 
  and  $G\text{-from-list}$  =  $G\text{-from-list}$ 
  and  $G\text{-valid-wgraph-repr}$  =  $G\text{-valid-wgraph-repr}$ 
by unfold-locales

```

```

lemma G-from-list-unfold:  $G\text{-from-list}$  =  $G\text{-from-list}$ 
  by (simp add: G-add-edge-def G-empty-def G-from-list-def)

```

```

lemma [code]:  $G\text{-from-list } l$  = foldr ( $\lambda(e, d). G\text{-add-edge } e \ d$ )  $l$   $G\text{-empty}$ 
  by (simp add: G-from-list-def G-from-list-unfold)

```

##### global-interpretation *Prim-Impl-Adts* - - -

```

 $G.\alpha w$   $G.\alpha g$   $G.invar$   $G.adj$   $G.empty$   $G.add-edge$ 

```

```

RBT-Set.empty RBT-Map.update RBT-Map.delete Lookup2.lookup RBT-Map.M.invar

```

```

PST-RBT.empty PST-RBT.update PST-RBT.delete PST-RBT.PM.invar

```

```

Lookup2.lookup PST-RBT.rbt-is-empty pst-getmin
..

global-interpretation P: Prim-Impl-Defs G.invar G.adj G.empty G.add-edge
RBT-Set.empty RBT-Map.update RBT-Map.delete Lookup2.lookup RBT-Map.M.invar

PST-RBT.empty PST-RBT.update PST-RBT.delete PST-RBT.PM.invar
Lookup2.lookup PST-RBT.rbt-is-empty pst-getmin

- - - G.αw G.αg g r
for g and r::'a::linorder
defines prim-impl = P.prim-impl
        and outer-loop-impl = P.outer-loop-impl
        and foreach-impl = P.foreach-impl
        and foreach-impl-body = P.foreach-impl-body
by unfold-locales

lemmas [code] = P.prim-impl-alt

context
  fixes g
  assumes [simp]: G.invar g
begin

interpretation AUX: Prim-Impl
  G.invar G.adj G.empty G.add-edge

  RBT-Set.empty RBT-Map.update RBT-Map.delete Lookup2.lookup RBT-Map.M.invar

  PST-RBT.empty PST-RBT.update PST-RBT.delete PST-RBT.PM.invar
  Lookup2.lookup PST-RBT.rbt-is-empty pst-getmin

  g r - - - G.αw G.αg for r::'a::linorder
  by unfold-locales simp-all

lemmas prim-impl-correct = AUX.prim-impl-correct[folded prim-impl-def]

end

```

## Adding a Graph-From-List Parser

```

definition prim-list-impl l r
  ≡ if G-valid-wgraph-repr l then Some (prim-impl (G-from-list l) r) else None

```

## 1.6.5 Main Correctness Theorem

The *prim-list-impl* algorithm returns *None*, if the input was invalid. Otherwise it returns *Some*  $(\pi i, Vi)$ , which satisfy the map/set invariants and encode a minimum spanning tree of the component of the graph that contains *r*.

Notes:

- If *r* is not a node of the graph, *component-of* will return the graph with the only node *r*. (*component-of-not-node*)

**theorem** *prim-list-impl-correct*:

**shows** *case prim-list-impl l r of*

*None*  $\Rightarrow \neg G.\text{valid-wgraph-repr } l$  — Invalid input

| *Some*  $\pi i \Rightarrow$

$G.\text{valid-wgraph-repr } l \wedge (\text{let } Gi = G.\text{from-list } l \text{ in } G.\text{invar } Gi$  — Valid input

$\wedge P.\text{invar-MST } \pi i$  — Output satisfies invariants

$\wedge \text{is-MST } (G.\alpha w \ Gi) (\text{component-of } (G.\alpha g \ Gi) \ r) (P.\alpha\text{-MST } r \ \pi i)$  — and

represents MST

**unfolding** *prim-list-impl-def G-from-list-unfold*

**using** *prim-impl-correct[of G.from-list l r]* *G.from-list-correct[of l]*

**by** (*auto simp: Let-def*)

**theorem** *prim-list-impl-correct-presentation*:

**shows** *case prim-list-impl l r of*

*None*  $\Rightarrow \neg G.\text{valid-wgraph-repr } l$  — Invalid input

| *Some*  $\pi i \Rightarrow \text{let}$

$g = G.\alpha g \ (G.\text{from-list } l);$

$w = G.\alpha w \ (G.\text{from-list } l);$

$rg = \text{component-of } g \ r;$

$t = P.\alpha\text{-MST } r \ \pi i$

*in*

$G.\text{valid-wgraph-repr } l$  — Valid input

$\wedge P.\text{invar-MST } \pi i$  — Output satisfies invariants

$\wedge \text{is-MST } w \ rg \ t$  — and represents MST

**using** *prim-list-impl-correct[of l r]* **unfolding** *Let-def*

**by** (*auto split: option.splits*)

## 1.6.6 Code Generation and Test

**definition** *prim-list-impl-int* ::  $- \Rightarrow \text{int} \Rightarrow -$

**where** *prim-list-impl-int*  $\equiv$  *prim-list-impl*

**export-code** *prim-list-impl prim-list-impl-int checking SML*

**experiment begin**

abbreviation  $a \equiv 1$   
abbreviation  $b \equiv 2$   
abbreviation  $c \equiv 3$   
abbreviation  $d \equiv 4$   
abbreviation  $e \equiv 5$   
abbreviation  $f \equiv 6$   
abbreviation  $g \equiv 7$   
abbreviation  $h \equiv 8$   
abbreviation  $i \equiv 9$

value (*prim-list-impl-int* [  
  ( $a,b,4$ ),  
  ( $a,h,8$ ),  
  ( $b,h,11$ ),  
  ( $b,c,8$ ),  
  ( $h,i,7$ ),  
  ( $h,g,1$ ),  
  ( $c,i,2$ ),  
  ( $g,i,6$ ),  
  ( $c,d,7$ ),  
  ( $c,f,4$ ),  
  ( $g,f,2$ ),  
  ( $d,f,14$ ),  
  ( $d,e,9$ ),  
  ( $e,f,10$ )  
] 1)

end

end

## Chapter 2

# Dijkstra's Shortest Path Algorithm

Dijkstra's algorithm [2] is a classical algorithm to determine the shortest paths from a root node to all other nodes in a weighted directed graph. Although it solves a different problem, and works on a different type of graphs, its structure is very similar to Prim's algorithm. In particular, like Prim's algorithm, it has a simple loop structure and can be efficiently implemented by a priority queue.

Again, our formalization of Dijkstra's algorithm follows the presentation of Cormen et al. [1]. However, for the sake of simplicity, our algorithm does not compute actual shortest paths, but only their weights.

### 2.1 Weighted Directed Graphs

```
theory Directed-Graph  
imports Common  
begin
```

A weighted graph is represented by a function from edges to weights. For simplicity, we use *enat* as weights,  $\infty$  meaning that there is no edge.

```
type-synonym ('v) wgraph = ('v  $\times$  'v)  $\Rightarrow$  enat
```

We encapsulate weighted graphs into a locale that fixes a graph

```
locale WGraph = fixes w :: 'v wgraph  
begin
```

Set of edges with finite weight

```
definition edges  $\equiv$   $\{(u,v) . w (u,v) \neq \infty\}$ 
```



### 2.1.1 Paths

A path between nodes  $u$  and  $v$  is a list of edge weights of a sequence of edges from  $u$  to  $v$ .

Note that a path may also contain edges with weight  $\infty$ .

```
fun path :: 'v  $\Rightarrow$  enat list  $\Rightarrow$  'v  $\Rightarrow$  bool where
  path u [] v  $\longleftrightarrow$  u=v
| path u (l#ls) v  $\longleftrightarrow$  ( $\exists$  uh. l = w (u,uh)  $\wedge$  path uh ls v)
```

```
lemma path-append[simp]:
  path u (ls1@ls2) v  $\longleftrightarrow$  ( $\exists$  w. path u ls1 w  $\wedge$  path w ls2 v)
by (induction ls1 arbitrary: u) auto
```

There is a singleton path between every two nodes (it's weight might be  $\infty$ ).

```
lemma triv-path: path u [w (u,v)] v by auto
```

Shortcut for the set of all paths between two nodes

```
definition paths u v  $\equiv$  {p . path u p v}
```

```
lemma paths-ne: paths u v  $\neq$  {} using triv-path unfolding paths-def by blast
```

If there is a path from a node inside a set  $S$ , to a node outside a set  $S$ , this path must contain an edge from inside  $S$  to outside  $S$ .

```
lemma find-leave-edgeE:
  assumes path u p v
  assumes u $\in$ S v $\notin$ S
  obtains p1 x y p2
    where p = p1@w (x,y)#p2 x $\in$ S y $\notin$ S path u p1 x path y p2 v
proof -
  have  $\exists$  p1 x y p2. p = p1@w (x,y)#p2  $\wedge$  x $\in$ S  $\wedge$  y $\notin$ S  $\wedge$  path u p1 x  $\wedge$  path y p2 v
  using assms
proof (induction p arbitrary: u)
  case Nil
  then show ?case by auto
next
  case (Cons a p)
  from Cons.prem1 obtain x where [simp]: a=w (u,x) and PX: path x p v
  by auto

  show ?case proof (cases x $\in$ S)
  case False with PX  $\langle$ u $\in$ S $\rangle$  show ?thesis by fastforce
  next
  case True from Cons.IH[OF PX True  $\langle$ v $\notin$ S $\rangle$ ] show ?thesis
    by clarsimp (metis WGraph.path.simps(2) append-Cons)
  qed
qed
thus ?thesis by (fast intro: that)
```

qed

### 2.1.2 Distance

The (minimum) distance between two nodes  $u$  and  $v$  is called  $\delta u v$ .

**definition**  $\delta u v \equiv \text{LEAST } w::\text{enat. } w \in \text{sum-list 'paths } u v$

**lemma** *obtain-shortest-path*:

**obtains**  $p$  **where**  $\text{path } s p u \delta s u = \text{sum-list } p$

**unfolding**  $\delta\text{-def}$  *paths-ne*

**by** (*smt Collect-empty-eq LeastI-ex WGraph.paths-def imageI image-iff mem-Collect-eq paths-def*)

**lemma** *shortest-path-least*:

$\text{path } s p u \implies \delta s u \leq \text{sum-list } p$

**unfolding**  $\delta\text{-def}$  *paths-def*

**by** (*simp add: Least-le*)

**lemma** *distance-refl[simp]*:  $\delta s s = 0$

**using** *shortest-path-least[of s [] s]* **by** *auto*

**lemma** *distance-direct*:  $\delta s u \leq w (s, u)$

**using** *shortest-path-least[of s [w (s,u)] u]* **by** *auto*

Triangle inequality: The distance from  $s$  to  $v$  is shorter than the distance from  $s$  to  $u$  and the edge weight from  $u$  to  $v$ .

**lemma** *triangle*:  $\delta s v \leq \delta s u + w (u, v)$

**proof** –

**have**  $\text{path } s (p@[w (u,v)]) v$  **if**  $\text{path } s p u$  **for**  $p$  **using** *that* **by** *auto*

**then have**  $(+) (w (u,v)) \text{ 'sum-list 'paths } s u \subseteq \text{sum-list 'paths } s v$

**by** (*fastforce simp: paths-def image-iff simp del: path.simps path-append*)

**from** *least-antimono[OF - this] paths-ne* **have**

$(\text{LEAST } y::\text{enat. } y \in \text{sum-list 'paths } s v)$

$\leq (\text{LEAST } x::\text{enat. } x \in (+) (w (u,v)) \text{ 'sum-list 'paths } s u)$

**by** (*auto simp: paths-def*)

**also have**  $\dots = (\text{LEAST } x. x \in \text{sum-list 'paths } s u) + w (u,v)$

**apply** (*subst Least-mono[of (+) (w (u,v)) sum-list 'paths s u]*)

**subgoal by** (*auto simp: mono-def*)

**subgoal by** *simp (metis paths-def mem-Collect-eq*

*obtain-shortest-path shortest-path-least*)

**subgoal by** *auto*

**done**

**finally show** *?thesis* **unfolding**  $\delta\text{-def}$  .

qed

Any prefix of a shortest path is a shortest path itself. Note: The  $< \infty$  conditions are required to avoid saturation in adding to  $\infty$ !

**lemma** *shortest-path-prefix*:

```

    assumes path s p1 x path x p2 u
    and DSU:  $\delta s u = \text{sum-list } p_1 + \text{sum-list } p_2 \delta s u < \infty$ 
    shows  $\delta s x = \text{sum-list } p_1 \delta s x < \infty$ 
  proof -
    have  $\delta s x \leq \text{sum-list } p_1$  using assms shortest-path-least by blast
    moreover have  $\neg \delta s x < \text{sum-list } p_1$  proof
      assume  $\delta s x < \text{sum-list } p_1$ 
      then obtain p1' where path s p1' x sum-list p1' < sum-list p1
        by (auto intro: obtain-shortest-path[of s x])
      with ⟨path x p2 u⟩ shortest-path-least[of s p1'@p2 u] DSU show False
        by fastforce
    qed
    ultimately show  $\delta s x = \text{sum-list } p_1$  by auto
    with DSU show  $\delta s x < \infty$  using le-iff-add by fastforce
  qed

end

end

```

## 2.2 Abstract Datatype for Weighted Directed Graphs

```

theory Directed-Graph-Specs
imports Directed-Graph
begin

locale adt-wgraph =
  fixes  $\alpha :: 'g \Rightarrow ('v) \text{wgraph}$ 
  and invar ::  $'g \Rightarrow \text{bool}$ 
  and succ ::  $'g \Rightarrow 'v \Rightarrow (\text{nat} \times 'v) \text{list}$ 
  and empty-graph ::  $'g$ 
  and add-edge ::  $'v \times 'v \Rightarrow \text{nat} \Rightarrow 'g \Rightarrow 'g$ 
  assumes succ-correct:  $\text{invar } g \Longrightarrow \text{set } (\text{succ } g u) = \{(d,v). \alpha g (u,v) = \text{enat } d\}$ 
  assumes empty-graph-correct:
    invar empty-graph
     $\alpha \text{ empty-graph} = (\lambda-. \infty)$ 
  assumes add-edge-correct:
    invar  $g \Longrightarrow \alpha g e = \infty \Longrightarrow \text{invar } (\text{add-edge } e d g)$ 
    invar  $g \Longrightarrow \alpha g e = \infty \Longrightarrow \alpha (\text{add-edge } e d g) = (\alpha g)(e:=\text{enat } d)$ 
begin

lemmas wgraph-specs = succ-correct empty-graph-correct add-edge-correct

end

locale adt-finite-wgraph = adt-wgraph where  $\alpha = \alpha$  for  $\alpha :: 'g \Rightarrow ('v) \text{wgraph} +$ 
  assumes finite:  $\text{invar } g \Longrightarrow \text{finite } (\text{WGraph.edges } (\alpha g))$ 

```

## 2.2.1 Constructing Weighted Graphs from Lists

**lemma** *edges-empty[simp]*:  $WGraph.edges (\lambda \cdot. \infty) = \{\}$   
**by** (*auto simp: WGraph.edges-def*)

**lemma** *edges-insert[simp]*:  
 $WGraph.edges (g(e:=enat d)) = Set.insert e (WGraph.edges g)$   
**by** (*auto simp: WGraph.edges-def*)

A list represents a graph if there are no multi-edges or duplicate edges

**definition** *valid-graph-rep*  $l \equiv$   
 $(\forall u d d' v. (u,v,d) \in set l \wedge (u,v,d') \in set l \longrightarrow d=d')$   
 $\wedge distinct l$

Alternative characterization: all node pairs must be distinct

**lemma** *valid-graph-rep-code[code]*:  
 $valid-graph-rep l \longleftrightarrow distinct (map (\lambda(u,v,-). (u,v)) l)$   
**by** (*auto simp: valid-graph-rep-def distinct-map inj-on-def*)

**lemma** *valid-graph-rep-simps[simp]*:  
 $valid-graph-rep []$   
 $valid-graph-rep ((u,v,d) \# l) \longleftrightarrow valid-graph-rep l \wedge (\forall d'. (u,v,d') \notin set l)$   
**by** (*auto simp: valid-graph-rep-def*)

For a valid graph representation, there is exactly one graph that corresponds to it

**lemma** *valid-graph-rep-ex1*:  
 $valid-graph-rep l \implies \exists! w. \forall u v d. w (u,v) = enat d \longleftrightarrow (u,v,d) \in set l$   
**unfolding** *valid-graph-rep-code*  
**apply** *safe*  
**subgoal**  
**apply** (*rule exI[where*  $x = \lambda(u,v).$   
 $if \exists d. (u,v,d) \in set l then enat (SOME d. (u,v,d) \in set l) else \infty]$ )  
**by** (*auto intro: someI simp: distinct-map inj-on-def split: prod.splits;*  
*blast*)  
**subgoal for**  $w w'$   
**apply** (*simp add: fun-eq-iff*)  
**by** (*metis (mono-tags, opaque-lifting) not-enat-eq*)  
**done**

We define this graph using determinate choice

**definition** *wgraph-of-list*  $l \equiv THE w. \forall u v d. w (u,v) = enat d \longleftrightarrow (u,v,d) \in set l$

**locale** *wgraph-from-list-algo* = *adt-wgraph*  
**begin**

**definition** *from-list*  $l \equiv fold (\lambda(u,v,d). add-edge (u,v) d) l empty-graph$

**definition**  $edges-undef\ l\ w \equiv \forall u\ v\ d. (u,v,d) \in set\ l \longrightarrow w\ (u,v) = \infty$

**lemma**  $edges-undef-simps[simp]$ :

$edges-undef\ []\ w$   
 $edges-undef\ l\ (\lambda-. \infty)$   
 $edges-undef\ ((u,v,d)\#l)\ w \longleftrightarrow edges-undef\ l\ w \wedge w\ (u,v) = \infty$   
 $edges-undef\ l\ (w((u,v) := enat\ d)) \longleftrightarrow edges-undef\ l\ w \wedge (\forall d'. (u,v,d') \notin set\ l)$   
**by** ( $auto\ simp: edges-undef-def$ )

**lemma**  $from-list-correct-aux$ :

**assumes**  $valid-graph-rep\ l$   
**assumes**  $edges-undef\ l\ (\alpha\ g)$   
**assumes**  $invar\ g$   
**defines**  $g' \equiv fold\ (\lambda(u,v,d). add-edge\ (u,v)\ d)\ l\ g$   
**shows**  $invar\ g'$   
**and**  $(\forall u\ v\ d. \alpha\ g'\ (u,v) = enat\ d \longleftrightarrow \alpha\ g\ (u,v) = enat\ d \vee (u,v,d) \in set\ l)$   
**using**  $assms(1-3)$  **unfolding**  $g'-def$   
**apply** ( $induction\ l\ arbitrary: g$ )  
**by** ( $auto\ simp: wgraph-specs\ split: if-splits$ )

**lemma**  $from-list-correct'$ :

**assumes**  $valid-graph-rep\ l$   
**shows**  $invar\ (from-list\ l)$   
**and**  $(u,v,d) \in set\ l \longleftrightarrow \alpha\ (from-list\ l)\ (u,v) = enat\ d$   
**unfolding**  $from-list-def$   
**using**  $from-list-correct-aux[OF\ assms, where\ g=empty-graph]$   
**by** ( $auto\ simp: wgraph-specs$ )

**lemma**  $from-list-correct$ :

**assumes**  $valid-graph-rep\ l$   
**shows**  $invar\ (from-list\ l)\ \alpha\ (from-list\ l) = wgraph-of-list\ l$

**proof** –

**from**  $theI'[OF\ valid-graph-rep-ex1[OF\ assms], folded\ wgraph-of-list-def]$   
**have**  $(wgraph-of-list\ l\ (u, v) = enat\ d) = ((u, v, d) \in set\ l)$  **for**  $u\ v\ d$   
**by**  $blast$

**then show**  $\alpha\ (from-list\ l) = wgraph-of-list\ l$

**using**  $from-list-correct-aux[OF\ assms, where\ g=empty-graph]$   
**apply** ( $clarsimp\ simp: fun-eq-iff\ wgraph-specs\ from-list-def$ )  
**apply** ( $metis\ (no-types)\ enat.exhaust$ )  
**done**

**show**  $invar\ (from-list\ l)$

**by** ( $simp\ add: assms\ from-list-correct'$ )

**qed**

**end**

end

## 2.3 Abstract Dijkstra Algorithm

**theory** *Dijkstra-Abstract*  
**imports** *Directed-Graph*  
**begin**

### 2.3.1 Abstract Algorithm

**type-synonym** *'v estimate = 'v  $\Rightarrow$  enat*

We fix a start node and a weighted graph

**locale** *Dijkstra = WGraph w for w :: ('v) wgraph +*  
**fixes** *s :: 'v*  
**begin**

Relax all outgoing edges of node *u*

**definition** *relax-outgoing :: 'v  $\Rightarrow$  'v estimate  $\Rightarrow$  'v estimate*  
**where** *relax-outgoing u D  $\equiv$   $\lambda v. \min (D v) (D u + w (u,v))$*

Initialization

**definition** *initD  $\equiv$  ( $\lambda -. \infty$ )(s:=0)*  
**definition** *initS  $\equiv$  {}*

Relaxing will never increase estimates

**lemma** *relax-mono: relax-outgoing u D v  $\leq$  D v*  
**by** (*auto simp: relax-outgoing-def*)

**definition** *all-dnodes  $\equiv$  Set.insert s { v .  $\exists u. w (u,v) \neq \infty$  }*

**definition** *unfinished-dnodes S  $\equiv$  all-dnodes - S*

**lemma** *unfinished-nodes-subset: unfinished-dnodes S  $\subseteq$  all-dnodes*  
**by** (*auto simp: unfinished-dnodes-def*)

end

### Invariant

The invariant is defined as locale

**locale** *Dijkstra-Invar = Dijkstra w s for w and s :: 'v +*  
**fixes** *D :: 'v estimate and S :: 'v set*  
**assumes** *upper-bound:  $\langle \delta s u \leq D u \rangle$  — D is a valid estimate*

**assumes** *s-in-S*:  $\langle s \in S \vee (D = (\lambda \cdot \infty)(s := 0) \wedge S = \{\}) \rangle$  — The start node is finished, or we are in initial state

**assumes** *S-precise*:  $u \in S \implies D u = \delta s u$  — Finished nodes have precise estimate

**assumes** *S-relaxed*:  $\langle v \in S \implies D u \leq \delta s v + w(v, u) \rangle$  — Outgoing edges of finished nodes have been relaxed, using precise distance

**begin**

**abbreviation** (in *Dijkstra*)  $D\text{-invar} \equiv \text{Dijkstra-Invar } w s$

The invariant holds for the initial state

**theorem** (in *Dijkstra*) *invar-init*:  $D\text{-invar } \text{init}D \text{ init}S$

**apply** *unfold-locales*

**unfolding** *initD-def initS-def*

**by** (*auto simp: relax-outgoing-def distance-direct*)

Relaxing some edges maintains the upper bound property

**lemma** *maintain-upper-bound*:  $\delta s u \leq (\text{relax-outgoing } v D) u$

**apply** (*clarsimp simp: relax-outgoing-def upper-bound split: prod.splits*)

**using** *triangle upper-bound add-right-mono dual-order.trans* **by** *blast*

Relaxing edges will not affect nodes with already precise estimates

**lemma** *relax-precise-id*:  $D v = \delta s v \implies \text{relax-outgoing } u D v = \delta s v$

**using** *maintain-upper-bound upper-bound relax-mono*

**by** (*metis antisym*)

In particular, relaxing edges will not affect finished nodes

**lemma** *relax-finished-id*:  $v \in S \implies \text{relax-outgoing } u D v = D v$

**by** (*simp add: S-precise relax-precise-id*)

The least (finite) estimate among all nodes  $u$  not in  $S$  is already precise. This will allow us to add the node  $u$  to  $S$ .

**lemma** *maintain-S-precise-and-connected*:

**assumes** *UNS*:  $u \notin S$

**assumes** *MIN*:  $\forall v. v \in S \implies D u \leq D v$

**shows**  $D u = \delta s u$

We start with a case distinction whether we are in the first step of the loop, where we process the start node, or in subsequent steps, where the start node has already been finished.

**proof** (*cases u=s*)

**assume** [*simp*]:  $u = s$  — First step of loop

**then show** *?thesis* **using**  $\langle u \notin S \rangle$  *s-in-S* **by** *simp*

**next**

**assume**  $\langle u \neq s \rangle$  — Later step of loop

The start node has already been finished

**with** *s-in-S MIN* **have**  $\langle s \in S \rangle$  **apply** *clarsimp* **using** *infinity-ne-i0* **by** *metis*

**show** *?thesis*

Next, we handle the case that  $u$  is unreachable.

**proof** (*cases*  $\langle \delta s u < \infty \rangle$ )  
**assume**  $\neg(\delta s u < \infty)$  — Node is unreachable (infinite distance)

By the upper-bound property, we get  $D u = \delta s u = \infty$

**then show** *?thesis using upper-bound[of u] by auto*  
**next**  
**assume**  $\delta s u < \infty$  — Main case: Node has finite distance

Consider a shortest path from  $s$  to  $u$

**obtain**  $p$  **where** *path s p u and DSU:  $\delta s u = \text{sum-list } p$*   
**by** (*rule obtain-shortest-path*)

It goes from inside  $S$  to outside  $S$ , so there must be an edge at the border.  
Let  $(x,y)$  be such an edge, with  $x \in S$  and  $y \notin S$ .

**from** *find-leave-edgeE[OF  $\langle \text{path } s p u \rangle \langle s \in S \rangle \langle u \notin S \rangle$ ]* **obtain**  $p1 x y p2$  **where**  
*[simp]:  $p = p1 @ w(x,y) \# p2$*   
**and** *DECOMP:  $x \in S y \notin S \text{ path } s p1 x \text{ path } y p2 u$* .

As prefixes of shortest paths are again shortest paths, the shortest path to  $y$  ends with edge  $(x,y)$

**have** *DSX:  $\delta s x = \text{sum-list } p1$  and DSY:  $\delta s y = \delta s x + w(x,y)$*   
**using** *shortest-path-prefix[of s p1 x w(x,y) # p2 u]*  
**and** *shortest-path-prefix[of s p1 @ [w(x,y)] y p2 u]*  
**and**  $\langle \delta s u < \infty \rangle$  *DECOMP*  
**by** (*force simp: DSU*)**+**

Upon adding  $x$  to  $S$ , this edge has been relaxed with the precise estimate for  $x$ . At this point the estimate for  $y$  has become precise, too

**with**  $\langle x \in S \rangle$  **have**  $D y = \delta s y$   
**by** (*metis S-relaxed antisym-conv upper-bound*)  
**moreover**

The shortest path to  $y$  is a prefix of that to  $u$ , thus it shorter or equal

**have**  $\dots \leq \delta s u$  **using** *DSU by (simp add: DSX DSY)*  
**moreover**

The estimate for  $u$  is an upper bound

**have**  $\dots \leq D u$  **using** *upper-bound by (auto)*  
**moreover**

$u$  was a node with smallest estimate

**have**  $\dots \leq D y$  **using**  $\langle u \notin S \rangle \langle y \notin S \rangle$  *MIN by auto*



**ultimately**

This closed a cycle in the inequation chain. Thus, by antisymmetry, all items are equal. In particular,  $D u = \delta s u$ , qed.

**show**  $D u = \delta s u$  **by** *simp*  
**qed**  
**qed**

A step of Dijkstra's algorithm maintains the invariant. More precisely, in a step of Dijkstra's algorithm, we pick a node  $u \notin S$  with least finite estimate, relax the outgoing edges of  $u$ , and add  $u$  to  $S$ .

**theorem** *maintain-D-invar*:

**assumes** *UNS*:  $u \notin S$   
**assumes** *UNI*:  $D u < \infty$   
**assumes** *MIN*:  $\forall v. v \notin S \longrightarrow D u \leq D v$   
**shows** *D-invar* (*relax-outgoing u D*) (*Set.insert u S*)  
**apply** (*cases*  $\langle s \in S \rangle$ )  
**subgoal**  
**apply** (*unfold-locales*)  
**subgoal by** (*simp add: maintain-upper-bound*)  
**subgoal by** *simp*  
**subgoal**  
**using** *maintain-S-precise-and-connected*[*OF UNS MIN*] *S-precise*  
**by** (*auto simp: relax-precise-id*)  
**subgoal**  
**using** *maintain-S-precise-and-connected*[*OF UNS MIN*]  
**by** (*auto simp: relax-outgoing-def S-relaxed min.coboundedI1*)  
**done**  
**subgoal**  
**apply** *unfold-locales*  
**using** *s-in-S UNI distance-direct*  
**by** (*auto simp: relax-outgoing-def split: if-splits*)  
**done**

When the algorithm is finished, i.e., when there are no unfinished nodes with finite estimates left, then all estimates are accurate.

**lemma** *invar-finish-imp-correct*:

**assumes** *F*:  $\forall u. u \notin S \longrightarrow D u = \infty$   
**shows**  $D u = \delta s u$   
**proof** (*cases*  $u \in S$ )  
**assume**  $u \in S$

The estimates of finished nodes are accurate

**then show** *?thesis* **using** *S-precise* **by** *simp*  
**next**  
**assume**  $\langle u \notin S \rangle$

$D u$  is minimal, and minimal estimates are precise

**then show** *?thesis*  
**using** *F maintain-S-precise-and-connected[of u]* **by** *auto*

**qed**

A step decreases the set of unfinished nodes.

**lemma** *unfinished-nodes-decr*:  
**assumes** *UNS: u ∉ S*  
**assumes** *UNI: D u < ∞*  
**shows** *unfinished-dnodes (Set.insert u S) ⊂ unfinished-dnodes S*  
**proof** –

There is a path to *u*

**from** *UNI* **have**  $\delta s u < \infty$  **using** *upper-bound[of u] leD* **by** *fastforce*

Thus, *u* is among *all-dnodes*

**have** *u ∈ all-dnodes*  
**proof** –  
**obtain** *p* **where** *path s p u sum-list p < ∞*  
**apply** (*rule obtain-shortest-path[of s u]*)  
**using**  $\langle \delta s u < \infty \rangle$  **by** *auto*  
**with**  $\langle u \notin S \rangle$  **show** *?thesis*  
**apply** (*cases p rule: rev-cases*)  
**by** (*auto simp: Dijkstra.all-dnodes-def*)  
**qed**

Which implies the proposition

**with**  $\langle u \notin S \rangle$  **show** *?thesis* **by** (*auto simp: unfinished-dnodes-def*)  
**qed**

**end**

### 2.3.2 Refinement by Priority Map and Map

In a second step, we implement *D* and *S* by a priority map *Q* and a map *V*. Both map nodes to finite weights, where *Q* maps unfinished nodes, and *V* maps finished nodes.

Note that this implementation is slightly non-standard: In the standard implementation, *Q* contains also unfinished nodes with infinite weight.

We chose this implementation because it avoids enumerating all nodes of the graph upon initialization of *Q*. However, on relaxing an edge to a node not in *Q*, we require an extra lookup to check whether the node is finished.

## Implementing *enat* by *Option*

Our maps are functions to *nat option*, which are interpreted as *enat*, *None* being  $\infty$

**fun** *enat-of-option* :: *nat option*  $\Rightarrow$  *enat* **where**  
*enat-of-option* *None* =  $\infty$   
| *enat-of-option* (*Some* *n*) = *enat* *n*

**lemma** *enat-of-option-inj*[*simp*]: *enat-of-option* *x* = *enat-of-option* *y*  $\longleftrightarrow$  *x*=*y*  
**by** (*cases* *x*; *cases* *y*; *simp*)

**lemma** *enat-of-option-simps*[*simp*]:  
*enat-of-option* *x* = *enat* *n*  $\longleftrightarrow$  *x* = *Some* *n*  
*enat-of-option* *x* =  $\infty$   $\longleftrightarrow$  *x* = *None*  
*enat* *n* = *enat-of-option* *x*  $\longleftrightarrow$  *x* = *Some* *n*  
 $\infty$  = *enat-of-option* *x*  $\longleftrightarrow$  *x* = *None*  
**by** (*cases* *x*; *auto*; *fail*)<sup>+</sup>

**lemma** *enat-of-option-le-conv*:  
*enat-of-option* *m*  $\leq$  *enat-of-option* *n*  $\longleftrightarrow$  (*case* (*m*,*n*) *of*  
  (*-,None*)  $\Rightarrow$  *True*  
  | (*Some* *a*, *Some* *b*)  $\Rightarrow$  *a* $\leq$ *b*  
  | (*-, -*)  $\Rightarrow$  *False*  
) )  
**by** (*auto split: option.split*)

## Implementing *D,S* by Priority Map and Map

**context** *Dijkstra* **begin**

We define a coupling relation, that connects the concrete with the abstract data.

**definition** *coupling* *Q V D S*  $\equiv$   
  *D* = *enat-of-option* *o* (*V* ++ *Q*)  
 $\wedge$  *S* = *dom* *V*  
 $\wedge$  *dom* *V*  $\cap$  *dom* *Q* = {}

Note that our coupling relation is functional.

**lemma** *coupling-fun*: *coupling* *Q V D S*  $\Longrightarrow$  *coupling* *Q V D' S'*  $\Longrightarrow$  *D'*=*D*  $\wedge$  *S'*=*S*  
**by** (*auto simp: coupling-def*)

The concrete version of the invariant.

**definition** *D-invar'* *Q V*  $\equiv$   
 $\exists$  *D S. coupling* *Q V D S*  $\wedge$  *D-invar* *D S*

Refinement of *relax-outgoing*

**definition** *relax-outgoing'* *u du V Q v*  $\equiv$   
  *case* *w* (*u*,*v*) *of*

```

 $\infty \Rightarrow Q v$ 
| enat d  $\Rightarrow$  (case Q v of
  None  $\Rightarrow$  if  $v \in \text{dom } V$  then None else Some (du+d)
  | Some d'  $\Rightarrow$  Some (min d' (du+d)))

```

A step preserves the coupling relation.

**lemma** (in *Dijkstra-Invar*) *coupling-step*:

**assumes** *C*: *coupling* *Q V D S*

**assumes** *UNS*:  $u \notin S$

**assumes** *UNI*:  $D u = \text{enat } du$

**shows** *coupling*

$((\text{relax-outgoing}' u du V Q)(u := \text{None})) (V(u \mapsto du))$

$(\text{relax-outgoing } u D) (\text{Set.insert } u S)$

**using** *C* **unfolding** *coupling-def*

**proof** (*intro ext conjI*; *elim conjE*)

**assume**  $\alpha$ :  $D = \text{enat-of-option} \circ V ++ Q S = \text{dom } V$

**and** *DD*:  $\text{dom } V \cap \text{dom } Q = \{\}$

**show**  $\text{Set.insert } u S = \text{dom } (V(u \mapsto du))$

**by** (*auto simp*:  $\alpha$ )

**have** [*simp*]:  $Q u = \text{Some } du \vee u = \text{None}$

**using** *DD UNI UNS* **by** (*auto simp*:  $\alpha$ )

**from** *DD*

**show**  $\text{dom } (V(u \mapsto du)) \cap \text{dom } ((\text{relax-outgoing}' u du V Q)(u := \text{None})) = \{\}$

**by** (*auto 0 3*

*simp*: *relax-outgoing'-def dom-def*

*split*: *if-splits enat.splits option.splits*)

**fix** *v*

**show** *relax-outgoing* *u D v*

$= (\text{enat-of-option} \circ V(u \mapsto du)) ++ ((\text{relax-outgoing}' u du V Q)(u := \text{None})) v$

**proof** (*cases*  $v \in S$ )

**case** *True*

**then show** *?thesis* **using** *DD*

**apply** (*simp add*: *relax-finished-id*)

**by** (*auto*

*simp*: *relax-outgoing'-def map-add-apply*  $\alpha$  *min-def*

*split*: *option.splits enat.splits*)

**next**

**case** *False*

**then show** *?thesis*

**by** (*auto*

*simp*: *relax-outgoing-def relax-outgoing'-def map-add-apply*  $\alpha$  *min-def*

*split*: *option.splits enat.splits*)

qed  
qed

Refinement of initial state

**definition**  $initQ \equiv Map.empty(s \mapsto 0)$

**definition**  $initV \equiv Map.empty$

**lemma** *coupling-init*:

*coupling initQ initV initD initS*

**unfolding** *coupling-def initD-def initQ-def initS-def initV-def*

**by** (*auto*

*simp: coupling-def relax-outgoing-def map-add-apply enat-0*

*split: option.split enat.split*

*del: ext intro!: ext*)

**lemma** *coupling-cond*:

**assumes** *coupling Q V D S*

**shows**  $(Q = Map.empty) \longleftrightarrow (\forall u. u \notin S \longrightarrow D u = \infty)$

**using** *assms*

**by** (*fastforce simp add: coupling-def*)

Termination argument: Refinement of unfinished nodes.

**definition**  $unfinished-dnodes' V \equiv unfinished-dnodes (dom V)$

**lemma** *coupling-unfinished*:

*coupling Q V D S  $\implies$  unfinished-dnodes' V = unfinished-dnodes S*

**by** (*auto simp: coupling-def unfinished-dnodes'-def unfinished-dnodes-def*)

## Implementing graph by successor list

**definition**  $relax-outgoing'' l du V Q = fold (\lambda(d,v) Q.$

*case Q v of None  $\Rightarrow$  if  $v \in dom V$  then  $Q$  else  $Q(v \mapsto du + d)$*

*| Some  $d' \Rightarrow Q(v \mapsto \min (du + d) d')$ ) l Q*

**lemma** *relax-outgoing''-refine*:

**assumes** *set l = {(d,v). w (u,v) = enat d}*

**shows**  $relax-outgoing'' l du V Q = relax-outgoing' u du V Q$

**proof**

**fix**  $v$

**have** *aux1*:

*relax-outgoing'' l du V Q v*

*= (if  $v \in snd\ set\ l$  then  $relax-outgoing' u du V Q v$  else  $Q v$ )*

**if** *set l  $\subseteq$  {(d,v). w (u,v) = enat d}*

**using** *that*

**apply** (*induction l arbitrary: Q v*)

**by** (*auto*

*simp: relax-outgoing''-def relax-outgoing'-def image-iff*)

```

    split!: if-splits option.splits)

have aux2:
  relax-outgoing' u du V Q v = Q v if w (u,v) = ∞
  using that by (auto simp: relax-outgoing'-def)

show relax-outgoing'' l du V Q v = relax-outgoing' u du V Q v
  using aux1
  apply (cases w (u,v))
  by (all ⟨force simp: aux2 assms⟩)
qed

end

end

```

## 2.4 Weighted Digraph Implementation by Adjacency Map

```

theory Directed-Graph-Impl
imports
  Directed-Graph-Specs
  HOL-Data-Structures.Map-Specs
begin

locale wgraph-by-map =
  M: Map M-empty M-update M-delete M-lookup M-invar

  for M-empty M-update M-delete
  and M-lookup :: 'm ⇒ 'v ⇒ ((nat × 'v) list) option and M-invar
begin

definition α :: 'm ⇒ ('v) wgraph where
  α g ≡ λ(u,v). case M-lookup g u of
    None ⇒ ∞
  | Some l ⇒ if ∃ d. (d,v) ∈ set l then enat (SOME d. (d,v) ∈ set l) else ∞

definition invar :: 'm ⇒ bool where invar g ≡
  M-invar g
  ∧ (∀ l ∈ ran (M-lookup g). distinct (map snd l))
  ∧ finite (WGraph.edges (α g))

definition succ :: 'm ⇒ 'v ⇒ (nat × 'v) list where
  succ g v = the-default [] (M-lookup g v)

definition empty-graph :: 'm where empty-graph = M-empty

definition add-edge :: 'v × 'v ⇒ nat ⇒ 'm ⇒ 'm where

```

```

add-edge ≡ λ(u,v) d g. M-update u ((d,v) # the-default [] (M-lookup g u)) g

sublocale adt-finite-wgraph invar succ empty-graph add-edge α
apply unfold-locales
subgoal for g u
  by (cases M-lookup g u)
    (auto
      simp: invar-def α-def succ-def ran-def
      intro: distinct-map-snd-inj someI
      split: option.splits
    )
  subgoal by (auto
    simp: invar-def α-def empty-graph-def add-edge-def M.map-specs
    split: option.split)
  subgoal by (auto
    simp: invar-def α-def empty-graph-def add-edge-def M.map-specs
    split: option.split)
proof –
Explicit proof to nicely handle finiteness constraint, using already proved
shape of abstract result
  fix g e d
  assume A: invar g α g e = ∞
  then show AAE: α (add-edge e d g) = (α g)(e := enat d)
    by (auto
      simp: invar-def α-def add-edge-def M.map-specs
      split: option.splits if-splits prod.splits
    )

  from A show invar (add-edge e d g)
  apply (simp add: invar-def AAE)
  by (force
    simp: invar-def α-def empty-graph-def add-edge-def M.map-specs ran-def
    split: option.splits if-splits prod.splits)
qed (simp add: invar-def)

end

end

```

## 2.5 Implementation of Dijkstra’s Algorithm

```

theory Dijkstra-Impl
imports
  Dijkstra-Abstract
  Directed-Graph-Impl
  HOL-Library.While-Combinator
  Priority-Search-Trees.PST-RBT
  HOL-Data-Structures.RBT-Map

```

**begin**

### 2.5.1 Implementation using ADT Interfaces

**locale** *Dijkstra-Impl-Adts* =

*G*: *adt-finite-wgraph* *G-invar* *G-succ* *G-empty* *G-add* *G-α*

+ *M*: *Map* *M-empty* *M-update* *M-delete* *M-lookup* *M-invar*

+ *Q*: *PrioMap* *Q-empty* *Q-update* *Q-delete* *Q-invar* *Q-lookup* *Q-is-empty* *Q-getmin*

**for** *G-α* :: '*g* ⇒ ('*v*) *wgraph* **and** *G-invar* *G-succ* *G-empty* *G-add*

**and** *M-empty* *M-update* *M-delete* **and** *M-lookup* :: '*m* ⇒ '*v* ⇒ *nat option*

**and** *M-invar*

**and** *Q-empty* *Q-update* *Q-delete* *Q-invar* **and** *Q-lookup* :: '*q* ⇒ '*v* ⇒ *nat option*

**and** *Q-is-empty* *Q-getmin*

**begin**

Simplifier setup

**lemmas** [*simp*] = *G.wgraph-specs*

**lemmas** [*simp*] = *M.map-specs*

**lemmas** [*simp*] = *Q.prio-map-specs*

**end**

**context** *PrioMap* **begin**

**lemma** *map-getminE*:

**assumes** *getmin* *m* = (*k*,*p*) *invar* *m* *lookup* *m* ≠ *Map.empty*

**obtains** *lookup* *m* *k* = *Some p* ∀ *k'* *p'*. *lookup* *m* *k'* = *Some p'* → *p* ≤ *p'*

**using** *map-getmin*[*OF* *assms*]

**by** (*auto simp: ran-def*)

**end**

**locale** *Dijkstra-Impl-Defs* = *Dijkstra-Impl-Adts* **where** *G-α* = *G-α*

+ *Dijkstra* ⟨*G-α* *g*⟩ *s*

**for** *G-α* :: '*g* ⇒ ('*v*::*linorder*) *wgraph* **and** *g* *s*

**locale** *Dijkstra-Impl* = *Dijkstra-Impl-Defs* **where** *G-α* = *G-α*

**for** *G-α* :: '*g* ⇒ ('*v*::*linorder*) *wgraph*

+

**assumes** *G-invar*[*simp*]: *G-invar* *g*

**begin**

**lemma** *finite-all-dnodes*[*simp, intro!*]: *finite all-dnodes*

**proof** –



**have**  $all\_dnodes \subseteq Set.insert\ s\ (snd\ 'edges)$   
**by** (*fastforce simp: all-dnodes-def edges-def image-iff*)  
**also have**  $finite\ \dots$  **by** (*auto simp: G.finite*)  
**finally** (*finite-subset*) **show** *?thesis* .  
**qed**

**lemma**  $finite\_unfinished\_dnodes[simp, intro!]: finite\ (unfinished\_dnodes\ S)$   
**using**  $finite\_subset[OF\ unfinished\_nodes\_subset]$  **by** *auto*

**lemma** (**in**  $-$ ) *fold-refine*:  
**assumes**  $I\ s$   
**assumes**  $\bigwedge s\ x. I\ s \implies x \in set\ l \implies I\ (f\ x\ s) \wedge \alpha\ (f\ x\ s) = f'\ x\ (\alpha\ s)$   
**shows**  $I\ (fold\ f\ l\ s) \wedge \alpha\ (fold\ f\ l\ s) = fold\ f'\ l\ (\alpha\ s)$   
**using** *assms*  
**by** (*induction l arbitrary: s*) *auto*

**definition** (**in** *Dijkstra-Impl-Defs*)  $Q\_relax\_outgoing\ u\ du\ V\ Q = fold\ (\lambda(d,v)\ Q.$   
*case Q-lookup Q v of*  
 $None \implies if\ M\_lookup\ V\ v \neq None\ then\ Q\ else\ Q\_update\ v\ (du+d)\ Q$   
 $| Some\ d' \implies Q\_update\ v\ (min\ (du+d)\ d')\ Q)\ ((G\_succ\ g\ u))\ Q$

**lemma**  $Q\_relax\_outgoing[simp]$ :  
**assumes**  $[simp]: Q\_invar\ Q$   
**shows**  $Q\_invar\ (Q\_relax\_outgoing\ u\ du\ V\ Q)$   
 $\wedge Q\_lookup\ (Q\_relax\_outgoing\ u\ du\ V\ Q)$   
 $= relax\_outgoing'\ u\ du\ (M\_lookup\ V)\ (Q\_lookup\ Q)$   
**apply** (*subst relax-outgoing''-refine[symmetric, where l=G-succ g u]*)  
**apply** *simp*  
**unfolding**  $Q\_relax\_outgoing\_def\ relax\_outgoing''\_def$   
**apply** (*rule fold-refine[where I=Q-invar and  $\alpha=Q\_lookup$ ]*)  
**by** (*auto split: option.split*)

**definition** (**in** *Dijkstra-Impl-Defs*)  $D\_invar\_impl\ Q\ V \equiv$   
 $Q\_invar\ Q \wedge M\_invar\ V \wedge D\_invar'\ (Q\_lookup\ Q)\ (M\_lookup\ V)$

**definition** (**in** *Dijkstra-Impl-Defs*)  
 $Q\_initQ \equiv Q\_update\ s\ 0\ Q\_empty$

**lemma**  $Q\_initQ[simp]$ :  
**shows**  $Q\_invar\ (Q\_initQ)\ Q\_lookup\ (Q\_initQ) = initQ$   
**by** (*auto simp: Q-initQ-def initQ-def*)

**definition** (**in** *Dijkstra-Impl-Defs*)  
 $M\_initV \equiv M\_empty$

**lemma**  $M\_initS[simp]: M\_invar\ M\_initV\ M\_lookup\ M\_initV = initV$   
**unfolding**  $M\_initV\_def\ initV\_def$  **by** *auto*

**term**  $Q\text{-getmin}$

**definition** (in *Dijkstra-Impl-Defs*)

```
dijkstra-loop  $\equiv$  while ( $\lambda(Q,V). \neg Q\text{-is-empty } Q$ ) ( $\lambda(Q,V).$   
  let  
    ( $u, du$ ) =  $Q\text{-getmin } Q$ ;  
     $Q = Q\text{-relax-outgoing } u \ du \ V \ Q$ ;  
     $Q = Q\text{-delete } u \ Q$ ;  
     $V = M\text{-update } u \ du \ V$   
  in  
    ( $Q, V$ )  
) ( $Q\text{-init}Q, M\text{-init}V$ )
```

**definition** (in *Dijkstra-Impl-Defs*)  $dijkstra \equiv \text{snd } dijkstra\text{-loop}$

**lemma** *transfer-preconditions*:

```
assumes coupling  $Q \ V \ D \ S$   
shows  $Q \ u = \text{Some } du \longleftrightarrow D \ u = \text{enat } du \wedge u \notin S$   
using assms  
by (auto simp: coupling-def)
```

**lemma** *dijkstra-loop-invar-and-empty*:

```
shows case  $dijkstra\text{-loop}$  of ( $Q, V$ )  $\Rightarrow D\text{-invar-impl } Q \ V \wedge Q\text{-is-empty } Q$   
unfolding dijkstra-loop-def  
apply (rule while-rule[where  
   $P = \text{case-prod } D\text{-invar-impl}$   
  and  $r = \text{inv-image finite-psubset (unfinished-dnodes' o } M\text{-lookup o snd)}$ ])  
apply (all  $\langle(\text{clarsimp split: prod.splits})?\rangle$ )  
subgoal  
  apply (simp add: D-invar-impl-def)  
  apply (simp add: D-invar'-def)  
  apply (intro exI conjI)  
  apply (rule coupling-init)  
  using initD-def initS-def invar-init by auto
```

**proof** –

```
fix  $Q \ V \ u \ du$   
assume  $\neg Q\text{-is-empty } Q \ D\text{-invar-impl } Q \ V \ Q\text{-getmin } Q = (u, du)$   
hence  $Q\text{-lookup } Q \neq \text{Map.empty } D\text{-invar}' (Q\text{-lookup } Q) (M\text{-lookup } V)$   
  and [simp]:  $Q\text{-invar } Q \ M\text{-invar } V$   
  and  $Q\text{-lookup } Q \ u = \text{Some } du \ \forall k' \ p'. \ Q\text{-lookup } Q \ k' = \text{Some } p' \longrightarrow du \leq p'$   
  by (auto simp: D-invar-impl-def elim: Q.map-getminE)
```

**then obtain**  $D \ S$  **where**

```
 $D\text{-invar } D \ S$   
and COUPLING:  $\text{coupling } (Q\text{-lookup } Q) (M\text{-lookup } V) \ D \ S$   
and ABS-PRE:  $D \ u = \text{enat } du \ u \notin S \ \forall v. \ v \notin S \longrightarrow D \ u \leq D \ v$   
by (auto  
  simp: D-invar'-def transfer-preconditions less-eq-enat-def)
```

```

    split: enat.splits)

then interpret Dijkstra-Invar  $G-\alpha$   $g$   $s$   $D$   $S$  by simp

have COUPLING': coupling
  ((relax-outgoing'  $u$   $du$  (M-lookup  $V$ ) (Q-lookup  $Q$ ))( $u := None$ ))
  ((M-lookup  $V$ )( $u \mapsto du$ ))
  (relax-outgoing  $u$   $D$ )
  (Set.insert  $u$   $S$ )
  using coupling-step[OF COUPLING  $\langle u \notin S \rangle$   $\langle D$   $u = enat$   $du \rangle$ ] by auto

show D-invar-impl (Q-delete  $u$  (Q-relax-outgoing  $u$   $du$   $V$   $Q$ )) (M-update  $u$   $du$   $V$ )
  using maintain-D-invar[OF  $\langle u \notin S \rangle$ ] ABS-PRE
  using COUPLING'
  by (auto simp: D-invar-impl-def D-invar'-def)

show unfinished-dnodes' (M-lookup (M-update  $u$   $du$   $V$ ))
   $\subset$  unfinished-dnodes' (M-lookup  $V$ )
   $\wedge$  finite (unfinished-dnodes' (M-lookup  $V$ ))
  using coupling-unfinished[OF COUPLING] coupling-unfinished[OF COUPLING']
  using unfinished-nodes-decr[OF  $\langle u \notin S \rangle$ ] ABS-PRE
  by simp
qed

lemma dijkstra-correct:
  M-invar dijkstra
  M-lookup dijkstra  $u = Some$   $d \iff \delta$   $s$   $u = enat$   $d$ 
  using dijkstra-loop-invar-and-empty
  unfolding dijkstra-def
  apply  $-$ 
  apply (all  $\langle$  clarsimp simp: D-invar-impl-def  $\rangle$ )
  apply (clarsimp simp: D-invar'-def)
  subgoal for  $Q$   $V$   $D$   $S$ 
    using Dijkstra-Invar.invar-finish-imp-correct[of  $G-\alpha$   $g$   $s$   $D$   $S$   $u$ ]
    apply (clarsimp simp: coupling-def)
    by (auto simp: domIff)
  done

end

```

## 2.5.2 Instantiation of ADTs and Code Generation

### global-interpretation

```

G: wgraph-by-map RBT-Set.empty RBT-Map.update
      RBT-Map.delete Lookup2.lookup RBT-Map.M.invar
defines G-empty = G.empty-graph
  and G-add-edge = G.add-edge
  and G-succ = G.succ

```

```

by unfold-locales

global-interpretation Dijkstra-Impl-Adts
  G.α G.invar G.succ G.empty-graph G.add-edge

  RBT-Set.empty RBT-Map.update RBT-Map.delete Lookup2.lookup RBT-Map.M.invar

  PST-RBT.empty PST-RBT.update PST-RBT.delete PST-RBT.PM.invar
  Lookup2.lookup PST-RBT.rbt-is-empty pst-getmin
  ..

global-interpretation D: Dijkstra-Impl-Defs
  G.invar G.succ G.empty-graph G.add-edge

  RBT-Set.empty RBT-Map.update RBT-Map.delete Lookup2.lookup RBT-Map.M.invar

  PST-RBT.empty PST-RBT.update PST-RBT.delete PST-RBT.PM.invar
  Lookup2.lookup PST-RBT.rbt-is-empty pst-getmin

  G.α g s for g and s::'v::linorder
  defines dijkstra = D.dijkstra
    and dijkstra-loop = D.dijkstra-loop
    and Q-relax-outgoing = D.Q-relax-outgoing
    and M-initV = D.M-initV
    and Q-initQ = D.Q-initQ
  ..

lemmas [code] =
  D.dijkstra-def D.dijkstra-loop-def

context
  fixes g
  assumes [simp]: G.invar g
begin

interpretation AUX: Dijkstra-Impl
  G.invar G.succ G.empty-graph G.add-edge

  RBT-Set.empty RBT-Map.update RBT-Map.delete Lookup2.lookup RBT-Map.M.invar

  PST-RBT.empty PST-RBT.update PST-RBT.delete PST-RBT.PM.invar
  Lookup2.lookup PST-RBT.rbt-is-empty pst-getmin

  g s G.α for s
  by unfold-locales simp-all

lemmas dijkstra-correct = AUX.dijkstra-correct[folded dijkstra-def]

```

**end**

### 2.5.3 Combination with Graph Parser

We combine the algorithm with a parser from lists to graphs

**global-interpretation**

*G*: *wgraph-from-list-algo* *G.α* *G.invar* *G.succ* *G.empty-graph* *G.add-edge*

**defines** *from-list* = *G.from-list*

..

**definition** *dijkstra-list* *l s* ≡

*if valid-graph-rep l then Some (dijkstra (from-list l) s) else None*

**theorem** *dijkstra-list-correct*:

*case dijkstra-list l s of*

*None* ⇒ ¬*valid-graph-rep l*

| *Some D* ⇒

*valid-graph-rep l*

∧ *M.invar D*

∧ (∀ *u d*. *lookup D u* = *Some d* ↔ *WGraph.δ (wgraph-of-list l) s u* = *enat*

*d*)

**unfolding** *dijkstra-list-def*

**by** (*auto simp: dijkstra-correct G.from-list-correct*)

**export-code** *dijkstra-list checking SML OCaml? Scala Haskell?*

**value** *dijkstra-list* [(1::nat,2,7),(1,3,1),(3,2,2)] 1

**value** *dijkstra-list* [(1::nat,2,7),(1,3,1),(3,2,2)] 3

**end**

# Bibliography

- [1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2009.
- [2] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, Dec 1959.
- [3] P. Lammich and T. Nipkow. Proof pearl: Purely functional, simple and efficient Priority Search Trees and applications to Prim and Dijkstra. In *Proc. of ITP*, 2019. to appear.
- [4] R. C. Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, Nov 1957.