

# 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 . . . . .	6
1.1.4	Paths . . . . .	8
1.1.5	Cycles . . . . .	11
1.1.6	Connected Graphs . . . . .	12
1.1.7	Component Containing Node . . . . .	12
1.1.8	Trees . . . . .	13
1.1.9	Spanning Trees . . . . .	14
1.2	Weighted Undirected Graphs . . . . .	15
1.2.1	Minimum Spanning Trees . . . . .	16
1.3	Abstract Graph Datatype . . . . .	16
1.3.1	Abstract Weighted Graph . . . . .	16
1.3.2	Generic From-List Algorithm . . . . .	17
1.4	Abstract Prim Algorithm . . . . .	19
1.4.1	Generic Algorithm: Light Edges . . . . .	19
1.4.2	Abstract Prim: Growing a Tree . . . . .	20
1.4.3	Prim: Using a Priority Queue . . . . .	22
1.4.4	Refinement of Inner Foreach Loop . . . . .	26
1.5	Implementation of Weighted Undirected Graph by Map . . . . .	27
1.5.1	Doubleton Set to Pair . . . . .	27
1.5.2	Generic Implementation . . . . .	27
1.6	Implementation of Prim's Algorithm . . . . .	29
1.6.1	Implementation using ADT Interfaces . . . . .	30
1.6.2	Refinement of State . . . . .	32
1.6.3	Refinement of Algorithm . . . . .	33
1.6.4	Instantiation with Actual Data Structures . . . . .	34
1.6.5	Main Correctness Theorem . . . . .	35
1.6.6	Code Generation and Test . . . . .	36

<b>2</b>	<b>Dijkstra's Shortest Path Algorithm</b>	<b>38</b>
2.1	Weighted Directed Graphs . . . . .	38
2.1.1	Paths . . . . .	39
2.1.2	Distance . . . . .	39
2.2	Abstract Datatype for Weighted Directed Graphs . . . . .	40
2.2.1	Constructing Weighted Graphs from Lists . . . . .	41
2.3	Abstract Dijkstra Algorithm . . . . .	42
2.3.1	Abstract Algorithm . . . . .	42
2.3.2	Refinement by Priority Map and Map . . . . .	44
2.4	Weighted Digraph Implementation by Adjacency Map . . . . .	47
2.5	Implementation of Dijkstra's Algorithm . . . . .	47
2.5.1	Implementation using ADT Interfaces . . . . .	48
2.5.2	Instantiation of ADTs and Code Generation . . . . .	50
2.5.3	Combination with Graph Parser . . . . .	51

# 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 set* , *E*).  $E \subseteq V \times V \wedge \text{finite } V \wedge \text{sym } E \wedge \text{irrefl } E$  }  
*<proof>*

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

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

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

**lift-definition** *graph-internal* :: *'v set*  $\Rightarrow$  (*'v* $\times$ *'v*) *set*  $\Rightarrow$  *'v ugraph*  
**is**  $\lambda V E$ . if *finite* *V*  $\wedge$  *finite* *E* then  $(V \cup \text{fst}'E \cup \text{snd}'E, (E \cup E^{-1}) - \text{Id})$  else  $(\{\}, \{\})$   
*<proof>*

**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*  $\subseteq$  *nodes* *g*  $\times$  *nodes* *g*  
*<proof>*

**lemma** *nodes-finite*[*simp*, *intro!*]: *finite* (*nodes* *g*)  
*<proof>*

**lemma** *edges-sym*: *sym* (*edges* *g*)  
*<proof>*

**lemma** *edges-irrefl*: *irrefl* (*edges* *g*)  
*<proof>*

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

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

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

**lemma** *nodes-edges-graph-presentation*:  $\llbracket \text{finite } V; \text{finite } E \rrbracket$   
 $\Longrightarrow \text{nodes } (\text{graph } V E) = V \cup \text{fst}'E \cup \text{snd}'E \wedge \text{edges } (\text{graph } V E) = E \cup E^{-1}$   
 $- \text{Id}$   
*<proof>*

**lemma** *graph-eq[simp]*:  $\text{graph } (\text{nodes } g) (\text{edges } g) = g$   
 ⟨proof⟩

**lemma** *edges-finite[simp, intro!]*:  $\text{finite } (\text{edges } g)$   
 ⟨proof⟩

**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⟩

**lemma** *graph-eq-iff*:  $g = g' \iff \text{nodes } g = \text{nodes } g' \wedge \text{edges } g = \text{edges } g'$   
 ⟨proof⟩

**lemma** *edges-sym'*:  $(u, v) \in \text{edges } g \implies (v, u) \in \text{edges } g$  ⟨proof⟩

**lemma** *edges-irrefl'[simp, intro!]*:  $(u, u) \notin \text{edges } g$   
 ⟨proof⟩

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

**lemma** *edgesT-diff-sng-inv-eq[simp]*:  
 $(\text{edges } T - \{(x, y), (y, x)\})^{-1} = \text{edges } T - \{(x, y), (y, x)\}$   
 ⟨proof⟩

**lemma** *nodesI[simp, intro]*: **assumes**  $(u, v) \in \text{edges } g$  **shows**  $u \in \text{nodes } g \ v \in \text{nodes } g$   
 ⟨proof⟩

**lemma** *split-edges-sym*:  $\exists E. E \cap E^{-1} = \{\} \wedge \text{edges } g = E \cup E^{-1}$   
 ⟨proof⟩

### 1.1.2 Connectedness Relation

**lemma** *rtrancl-edges-sym'*:  $(u, v) \in (\text{edges } g)^* \implies (v, u) \in (\text{edges } g)^*$   
 ⟨proof⟩

**lemma** *trancl-edges-subset*:  $(\text{edges } g)^+ \subseteq \text{nodes } g \times \text{nodes } g$   
 ⟨proof⟩

**lemma** *find-crossing-edge*:  
**assumes**  $(u, v) \in E^* \ u \in V \ v \notin V$   
**obtains**  $u' \ v'$  **where**  $(u', v') \in E \cap V \times -V$   
 ⟨proof⟩

### 1.1.3 Constructing Graphs

**definition** *graph-empty*  $\equiv \text{graph } \{\} \{\}$

**definition** *ins-node*  $v \ g \equiv \text{graph } (\text{insert } v (\text{nodes } g)) (\text{edges } g)$

**definition** *ins-edge*  $e \ g \equiv \text{graph } (\text{nodes } g) (\text{insert } e (\text{edges } g))$

**definition** *graph-join*  $g_1 g_2 \equiv \text{graph } (\text{nodes } g_1 \cup \text{nodes } g_2) (\text{edges } g_1 \cup \text{edges } g_2)$

**definition** *restrict-nodes*  $g V \equiv \text{graph } (\text{nodes } g \cap V) (\text{edges } g \cap V \times V)$

**definition** *restrict-edges*  $g E \equiv \text{graph } (\text{nodes } g) (\text{edges } g \cap (E \cup E^{-1}))$

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

**lemma** [*simp*]:

**assumes** *nodes-edges-consistent*  $V E$

**shows** *nodes-graph'*:  $\text{nodes } (\text{graph } V E) = V$  (**is** ?G1)

**and** *edges-graph'*:  $\text{edges } (\text{graph } V E) = E$  (**is** ?G2)

*<proof>*

**lemma** *nec-empty*[*simp*]: *nodes-edges-consistent*  $\{\} \{\}$

*<proof>*

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

*nodes graph-empty* =  $\{\}$

*edges graph-empty* =  $\{\}$

*<proof>*

**lemma** *graph-empty*[*simp*]:  $\text{graph } \{\} \{\} = \text{graph-empty}$

*<proof>*

**lemma** *nodes-empty-iff-empty*[*simp*]:

$\text{nodes } G = \{\} \longleftrightarrow G = \text{graph } \{\} \{\}$

$\{\} = \text{nodes } G \longleftrightarrow G = \text{graph-empty}$

*<proof>*

**lemma** *nodes-ins-nodes*[*simp*]:  $\text{nodes } (\text{ins-node } v g) = \text{insert } v (\text{nodes } g)$

**and** *edges-ins-nodes*[*simp*]:  $\text{edges } (\text{ins-node } v g) = \text{edges } g$

*<proof>*

**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*:

$\text{edges } (\text{ins-edge } e g)$

= (if  $\text{fst } e = \text{snd } e$  then  $\text{edges } g$  else  $\{e, \text{prod.swap } e\} \cup (\text{edges } g)$ )

*<proof>*

**lemma** *edges-ins-edge'*[*simp*]:

$u \neq v \implies \text{edges } (\text{ins-edge } (u,v) g) = \{(u,v), (v,u)\} \cup \text{edges } g$

*<proof>*

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

*<proof>*

**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]:  $edges (graph-join\ g_1\ g_2) = edges\ g_1 \cup edges\ g_2$   
 ⟨proof⟩

**lemma** *nodes-restrict-nodes*[simp]:  $nodes (restrict-nodes\ g\ V) = nodes\ g \cap V$   
**and** *edges-restrict-nodes*[simp]:  $edges (restrict-nodes\ g\ V) = edges\ g \cap V \times V$   
 ⟨proof⟩

**lemma** *nodes-restrict-edges*[simp]:  $nodes (restrict-edges\ g\ E) = nodes\ g$   
**and** *edges-restrict-edges*[simp]:  $edges (restrict-edges\ g\ E) = edges\ g \cap (E \cup E^{-1})$   
 ⟨proof⟩

**lemma** *unrestrict-edges*:  $edges (restrict-edges\ g\ E) \subseteq edges\ g$  ⟨proof⟩

**lemma** *unrestrictn-edges*:  $edges (restrict-nodes\ g\ V) \subseteq edges\ g$  ⟨proof⟩

**lemma** *unrestrict-nodes*:  $nodes (restrict-edges\ g\ E) \subseteq nodes\ g$  ⟨proof⟩

#### 1.1.4 Paths

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

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

**lemma** *path-emptyI*[intro!]:  $path\ g\ u\ []\ u$  ⟨proof⟩

**lemma** *path-append*[simp]:  
 $path\ g\ u\ (p1\@\!p2)\ w \longleftrightarrow (\exists v. path\ g\ u\ p1\ v \wedge path\ g\ v\ p2\ w)$   
 ⟨proof⟩

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

**lemma** *path-graph-empty*[simp]:  $path\ graph-empty\ u\ p\ v \longleftrightarrow v=u \wedge p=[]$   
 ⟨proof⟩

**abbreviation** *revp*  $p \equiv rev (map\ prod.swap\ p)$

**lemma** *revp-alt*:  $revp\ p = rev (map (\lambda(u,v). (v,u))\ p)$  ⟨proof⟩

**lemma** *path-rev*[simp]:  $path\ g\ u\ (revp\ p)\ v \longleftrightarrow path\ g\ v\ p\ u$   
 ⟨proof⟩

**lemma** *path-rev-sym*[sym]:  $path\ g\ v\ p\ u \implies path\ g\ u\ (revp\ p)\ v$  ⟨proof⟩

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



*<proof>*

**lemma** *path-edges*:  $path\ g\ u\ p\ v \implies set\ p \subseteq edges\ g$   
*<proof>*

**lemma** *path-graph-cong*:  
 $\llbracket path\ g_1\ u\ p\ v; set\ p \subseteq edges\ g_1 \implies set\ p \subseteq edges\ g_2 \rrbracket \implies path\ g_2\ u\ p\ v$   
*<proof>*

**lemma** *path-endpoints*:  
**assumes**  $path\ g\ u\ p\ v\ p \neq []$  **shows**  $u \in nodes\ g\ v \in nodes\ g$   
*<proof>*

**lemma** *path-mono*:  $edges\ g \subseteq edges\ g' \implies path\ g\ u\ p\ v \implies path\ g'\ u\ p\ v$   
*<proof>*

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

**lemma** *unrestrict-path-edges*:  $path\ (restrict-edges\ g\ E)\ u\ p\ v \implies path\ g\ u\ p\ v$   
*<proof>*

**lemma** *unrestrict-path-nodes*:  $path\ (restrict-nodes\ g\ E)\ u\ p\ v \implies path\ g\ u\ p\ v$   
*<proof>*

## Paths and Connectedness

**lemma** *rtrancl-edges-iff-path*:  $(u,v) \in (edges\ g)^* \iff (\exists p. path\ g\ u\ p\ v)$   
*<proof>*

**lemma** *rtrancl-edges-pathE*:  
**assumes**  $(u,v) \in (edges\ g)^*$  **obtains**  $p$  **where**  $path\ g\ u\ p\ v$   
*<proof>*

**lemma** *path-rtrancl-edgesD*:  $path\ g\ u\ p\ v \implies (u,v) \in (edges\ g)^*$   
*<proof>*

## Simple Paths

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

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

**lemma** *in-uedge-conv[simp]*:  $x \in uedge\ (u,v) \iff x=u \vee x=v$   
*<proof>*

**lemma** *uedge-eq-iff*:  $uedge (a,b) = uedge (c,d) \longleftrightarrow a=c \wedge b=d \vee a=d \wedge b=c$   
 ⟨proof⟩

**lemma** *uedge-degen[simp]*:  $uedge (a,a) = \{a\}$   
 ⟨proof⟩

**lemma** *uedge-in-set-eq*:  $uedge (u, v) \in uedge ' S \longleftrightarrow (u,v) \in S \vee (v,u) \in S$   
 ⟨proof⟩

**lemma** *uedge-commute*:  $uedge (a,b) = uedge (b,a)$  ⟨proof⟩

**lemma** *simple-empty[simp]*:  $simple []$   
 ⟨proof⟩

**lemma** *simple-cons[simp]*:  $simple (e\#p) \longleftrightarrow uedge e \notin uedge ' set p \wedge simple p$   
 ⟨proof⟩

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

**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⟩

**lemma** *simplify-pathE*:  
**assumes**  $path g u p v$   
**obtains**  $p'$  **where**  $path g u p' v \wedge simple p' \wedge set p' \subseteq set p$   
 ⟨proof⟩

## Splitting Paths

**lemma** *find-crossing-edge-on-path*:  
**assumes**  $path g u p v \wedge \neg P u P v$   
**obtains**  $u' v'$  **where**  $(u',v') \in set p \wedge \neg P u' P v'$   
 ⟨proof⟩

**lemma** *find-crossing-edges-on-path*:  
**assumes**  $P$ :  $path g u p v$  **and**  $P u P v$   
**obtains**  $\forall (u,v) \in set p. P u \wedge P v$   
 $| u_1 v_1 v_2 u_2 p_1 p_2 p_3$   
**where**  $p = p_1 @ [(u_1, v_1)] @ p_2 @ [(u_2, v_2)] @ p_3 \wedge P u_1 \wedge \neg P v_1 \wedge \neg P u_2 \wedge P v_2$   
 ⟨proof⟩

**lemma** *find-crossing-edge-rtrancl*:  
**assumes**  $(u,v) \in (edges g)^* \wedge \neg P u P v$   
**obtains**  $u' v'$  **where**  $(u',v') \in edges g \wedge \neg P u' P v'$

*<proof>*

**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>*

### 1.1.5 Cycles

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

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

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

*<proof>*

**lemma** *cycle-freeI*:

**assumes**  $\bigwedge p \ u. \ [\text{path } g \ u \ p \ u; \ p \neq []; \text{simple } p] \implies \text{False}$

**shows** *cycle-free*  $g$

*<proof>*

**lemma** *cycle-freeD*:

**assumes** *cycle-free*  $g$  *path*  $g \ u \ p \ u \ p \neq []$  *simple*  $p$

**shows** *False*

*<proof>*

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

*<proof>*

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

*<proof>*

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

*<proof>*

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

**assumes** *CF*: *cycle-free*  $g$

**assumes** *P*: *path*  $g \ u \ p \ v$  *path*  $g \ u \ p' \ v$  *simple*  $p$  *simple*  $p'$

**shows**  $p = p'$

*<proof>*

### Characterization by Removing Edge

**lemma** *cycle-free-alt*: *cycle-free*  $g$

$\iff (\forall e \in \text{edges } g. \ e \notin (\text{edges } (\text{restrict-edges } g \ (-\{e, \text{prod.swap } e\}))))^*$

*<proof>*

**lemma** *cycle-free-altI*:  
**assumes**  $\bigwedge u v. \llbracket (u,v) \in \text{edges } g; (u,v) \in (\text{edges } g - \{(u,v), (v,u)\})^* \rrbracket \implies \text{False}$   
**shows** *cycle-free*  $g$   
 $\langle \text{proof} \rangle$

**lemma** *cycle-free-altD*:  
**assumes** *cycle-free*  $g$   
**assumes**  $(u,v) \in \text{edges } g$   
**shows**  $(u,v) \notin (\text{edges } g - \{(u,v), (v,u)\})^*$   
 $\langle \text{proof} \rangle$

**lemma** *remove-redundant-edge*:  
**assumes**  $(u, v) \in (\text{edges } g - \{(u, v), (v, u)\})^*$   
**shows**  $(\text{edges } g - \{(u, v), (v, u)\})^* = (\text{edges } g)^*$  (**is**  $?E'^* = -$ )  
 $\langle \text{proof} \rangle$

### 1.1.6 Connected Graphs

**definition** *connected*  
**where** *connected*  $g \equiv \text{nodes } g \times \text{nodes } g \subseteq (\text{edges } g)^*$

**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*  $g$   
 $\langle \text{proof} \rangle$

**lemma** *connectedD*[*intro?*]:  
**assumes** *connected*  $g$   $u \in \text{nodes } g$   $v \in \text{nodes } g$   
**shows**  $(u,v) \in (\text{edges } g)^*$   
 $\langle \text{proof} \rangle$

**lemma** *connected-empty*[*simp*]: *connected graph-empty*  
 $\langle \text{proof} \rangle$

### 1.1.7 Component Containing Node

**definition** *reachable-nodes*  $g r \equiv (\text{edges } g)^* \text{ `` } \{r\}$

**definition** *component-of*  $g r$   
 $\equiv \text{ins-node } r (\text{restrict-nodes } g (\text{reachable-nodes } g r))$

**lemma** *reachable-nodes-refl*[*simp*, *intro!*]:  $r \in \text{reachable-nodes } g r$   
 $\langle \text{proof} \rangle$

**lemma** *reachable-nodes-step*:  
 $\text{edges } g \text{ `` } \text{reachable-nodes } g r \subseteq \text{reachable-nodes } g r$   
 $\langle \text{proof} \rangle$

**lemma** *reachable-nodes-steps*:

$(edges\ g)^* \text{ “ } reachable\text{-nodes}\ g\ r \subseteq reachable\text{-nodes}\ g\ r$

$\langle proof \rangle$

**lemma** *reachable-nodes-step'*:

**assumes**  $u \in reachable\text{-nodes}\ g\ r\ (u, v) \in edges\ g$

**shows**  $v \in reachable\text{-nodes}\ g\ r\ (u, v) \in edges\ (component\text{-of}\ g\ r)$

$\langle proof \rangle$

**lemma** *reachable-nodes-steps'*:

**assumes**  $u \in reachable\text{-nodes}\ g\ r\ (u, v) \in (edges\ g)^*$

**shows**  $v \in reachable\text{-nodes}\ g\ r\ (u, v) \in (edges\ (component\text{-of}\ g\ r))^*$

$\langle proof \rangle$

**lemma** *reachable-not-node*:  $r \notin nodes\ g \implies reachable\text{-nodes}\ g\ r = \{r\}$

$\langle proof \rangle$

**lemma** *nodes-of-component[simp]*:  $nodes\ (component\text{-of}\ g\ r) = reachable\text{-nodes}\ g\ r$

$\langle proof \rangle$

**lemma** *component-connected[simp, intro!]*:  $connected\ (component\text{-of}\ g\ r)$

$\langle proof \rangle$

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

$\langle proof \rangle$

**lemma** *component-path*:  $u \in nodes\ (component\text{-of}\ g\ r) \implies$

$path\ (component\text{-of}\ g\ r)\ u\ p\ v \longleftrightarrow path\ g\ u\ p\ v$

$\langle proof \rangle$

**lemma** *component-cycle-free*:  $cycle\text{-free}\ g \implies cycle\text{-free}\ (component\text{-of}\ g\ r)$

$\langle proof \rangle$

**lemma** *component-of-connected-graph*:

$\llbracket connected\ g; r \in nodes\ g \rrbracket \implies component\text{-of}\ g\ r = g$

$\langle proof \rangle$

**lemma** *component-of-not-node*:  $r \notin nodes\ g \implies component\text{-of}\ g\ r = graph\ \{r\}\ \{\}$

$\langle proof \rangle$

### 1.1.8 Trees

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

**lemma** *tree-empty[simp]*:  $tree\ graph\text{-empty}\ \langle proof \rangle$

**lemma** *component-of-tree*:  $tree\ T \implies tree\ (component\text{-of}\ T\ r)$

*<proof>*

## 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* ? $g'$ )

*<proof>*

**lemma** *join-cycle-free*:

**assumes** *CYCF*: *cycle-free*  $g_1$  *cycle-free*  $g_2$

**assumes** *DJ*:  $\text{nodes } g_1 \cap \text{nodes } g_2 = \{\}$

**assumes** *IN-NODES*:  $u \in \text{nodes } g_1$   $v \in \text{nodes } g_2$

**shows** *cycle-free* (*ins-edge* ( $u,v$ ) (*graph-join*  $g_1$   $g_2$ )) (**is** *cycle-free* ? $g'$ )

*<proof>*

**lemma** *join-trees*:

**assumes** *TREE*: *tree*  $g_1$  *tree*  $g_2$

**assumes** *DJ*:  $\text{nodes } g_1 \cap \text{nodes } g_2 = \{\}$

**assumes** *IN-NODES*:  $u \in \text{nodes } g_1$   $v \in \text{nodes } g_2$

**shows** *tree* (*ins-edge* ( $u,v$ ) (*graph-join*  $g_1$   $g_2$ ))

*<proof>*

**lemma** *split-tree*:

**assumes** *tree*  $T$  ( $x,y$ )  $\in$  *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>*

### 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*)

*<proof>*

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

*<proof>*

**lemma** *tree-singleton*[simp]: *tree (ins-node u graph-empty)*  
 ⟨proof⟩

**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⟩

Remove edges on cycles until the graph is cycle free

**lemma** *ex-spanning-tree*:  
 $\text{connected } g \implies \exists t. \text{is-spanning-tree } g \ t$   
 ⟨proof⟩

## 1.2 Weighted Undirected Graphs

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

**lemma** *weight-alt*:  $\text{weight } w \ g = (\sum e \in \text{uedge}'\text{edges } g. w \ e)$   
 ⟨proof⟩

**lemma** *weight-empty*[simp]:  $\text{weight } w \ \text{graph-empty} = 0$  ⟨proof⟩

**lemma** *weight-ins-edge*[simp]:  $\llbracket u \neq v; (u,v) \notin \text{edges } g \rrbracket$   
 $\implies \text{weight } w \ (\text{ins-edge } (u,v) \ g) = w \ \{u,v\} + \text{weight } w \ g$   
 ⟨proof⟩

**lemma** *uedge-img-disj-iff*[simp]:  
 $\text{uedge}'\text{edges } g_1 \cap \text{uedge}'\text{edges } g_2 = \{\} \longleftrightarrow \text{edges } g_1 \cap \text{edges } g_2 = \{\}$   
 ⟨proof⟩

**lemma** *weight-join*[simp]:  $\text{edges } g_1 \cap \text{edges } g_2 = \{\}$   
 $\implies \text{weight } w \ (\text{graph-join } g_1 \ g_2) = \text{weight } w \ g_1 + \text{weight } w \ g_2$   
 ⟨proof⟩

**lemma** *weight-cong*:  $\text{edges } g_1 = \text{edges } g_2 \implies \text{weight } w \ g_1 = \text{weight } w \ g_2$   
 ⟨proof⟩

**lemma** *weight-mono*:  $\text{edges } g \subseteq \text{edges } g' \implies \text{weight } w \ g \leq \text{weight } w \ g'$   
 ⟨proof⟩

**lemma** *weight-ge-edge*:  
**assumes**  $(x,y) \in \text{edges } T$   
**shows**  $\text{weight } w \ T \geq w \ \{x,y\}$   
 ⟨proof⟩

**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>*

### 1.2.1 Minimum Spanning Trees

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

**lemma** *exists-MST*:  $\text{connected } g \Longrightarrow \exists t. \text{is-MST } w g t$   
*<proof>*

**end**

## 1.3 Abstract Graph Datatype

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

### 1.3.1 Abstract Weighted Graph

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

**begin**

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

**lemma** *empty-spec-presentation*:



$invar\ empty \wedge \alpha g\ empty = graph\ \{\}\ \{\} \wedge \alpha w\ empty = (\lambda-. 0)$   
 ⟨proof⟩

**lemma** *add-edge-spec-presentation*:

[[ $invar\ g; (u,v) \notin edges\ (\alpha g\ g); u \neq v$ ]]  $\implies$   
 $invar\ (add-edge\ (u,v)\ d\ g)$   
 $\wedge \alpha g\ (add-edge\ (u,v)\ d\ g) = ins-edge\ (u,v)\ (\alpha g\ g)$   
 $\wedge \alpha w\ (add-edge\ (u,v)\ d\ g) = (\alpha w\ g)(\{u,v\}:=d)$   
 ⟨proof⟩

**end**

### 1.3.2 Generic From-List Algorithm

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

**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*:  $graph-from-list\ l = fold\ ins-edge\ l\ graph-empty$   
 ⟨proof⟩

**lemma** *nodes-of-graph-from-list*:  $nodes\ (graph-from-list\ l) = fst'set\ l \cup snd'set\ l$   
 ⟨proof⟩

**lemma** *edges-of-graph-from-list*:  
**assumes** *valid*:  $valid-graph-repr\ l$   
**shows**  $edges\ (graph-from-list\ l) = set\ l \cup (set\ l)^{-1}$   
 ⟨proof⟩

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

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

**lemma** *graph-from-list-simps*:  
 $graph-from-list\ [] = graph-empty$   
 $graph-from-list\ ((u,v)\#l) = ins-edge\ (u,v)\ (graph-from-list\ l)$   
 ⟨proof⟩

**lemma** *weight-from-list-simps*:  
 $weight-from-list\ [] = (\lambda-. 0)$   
 $weight-from-list\ (((u,v),d)\#xs) = (weight-from-list\ xs)(\{u,v\}:=d)$   
 ⟨proof⟩

**lemma** *valid-graph-repr-simps*:  
*valid-graph-repr* []  
*valid-graph-repr* ((*u,v*)#*xs*)  $\longleftrightarrow$  *u*≠*v* ∧ *valid-graph-repr xs*  
 ⟨*proof*⟩

**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*  
 ⟨*proof*⟩

**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*⟩

**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*)  
 ∧  $\alpha_g$  (*from-list l*) = *graph-from-list* (*map fst l*)  
 ∧  $\alpha_w$  (*from-list l*) = *weight-from-list l*  
 ⟨*proof*⟩

**lemma** *from-list-correct*:  
**assumes** *valid-wgraph-repr l*  
**shows**  
*invar* (*from-list l*)  
*nodes* ( $\alpha_g$  (*from-list l*)) = *fst'fst'set l* ∪ *snd'fst'set l*  
*edges* ( $\alpha_g$  (*from-list l*)) = (*fst'set l*) ∪ (*fst'set l*)<sup>-1</sup>  
 ((*u,v*),*d*)∈*set l*  $\implies$   $\alpha_w$  (*from-list l*) {*u,v*} = *d*  
 ⟨*proof*⟩

**lemma** *valid-wgraph-repr-presentation*: *valid-wgraph-repr l*  $\longleftrightarrow$   
 ( $\forall$ ((*u,v*),*d*)∈*set l*. *u*≠*v*) ∧ *distinct* [ {*u,v*}. ((*u,v*),*d*)←*l* ]  
 ⟨*proof*⟩

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

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

*<proof>*

**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]: connected g  $\implies$  is-subset-MST w g {}*  
*<proof>*

We fix a start node and a weighted graph

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

Reachable part of the graph

**definition** *rg  $\equiv$  component-of g r*

**lemma** *reachable-connected[simp, intro!]: connected rg*  
*<proof>*

**lemma** *reachable-edges-subset: edges rg  $\subseteq$  edges g*  
*<proof>*

**definition** *light-edge C u v*  
 $\equiv$  *u  $\in$  C  $\wedge$  v  $\notin$  C  $\wedge$  (u,v)  $\in$  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**  $A :: ('v \times 'v)$  set **and**  $C :: 'v$  set  
**assumes** *subset-MST*: *is-subset-MST*  $w\ rg\ A$   
**assumes** *respects-cut*: *respects-cut*  $A\ C$   
**assumes** *light-edge*: *light-edge*  $C\ u\ v$   
**shows** *is-subset-MST*  $w\ rg\ (\{(v,u)\} \cup A)$   
*<proof>*

**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'*:  $A \subseteq S\ A \times S\ A$   
*<proof>*

**corollary** *respects-cut*: *respects-cut*  $A\ (S\ A)$   
*<proof>*

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*  $w\ rg\ A$   
*<proof>*

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

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

**lemma** *finite-rem-nodes*[*simp,intro!*]: *finite*  $(nodes\ rg - S\ A)$  *<proof>*

**lemma** *A-edges*:  $A \subseteq \text{edges } g$

*<proof>*

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

*<proof>*

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

*<proof>*

**lemma** *edges-S-rg-edges*:  $\text{edges } g \cap S A \times -S A = \text{edges } rg \cap S A \times -S A$

*<proof>*

**lemma** *T-measure1-less*:  $T\text{-measure1 } A < \text{card } (\text{nodes } rg)$

*<proof>*

**lemma** *finite-A[simp, intro!]*: *finite*  $A$

*<proof>*

**lemma** *finite-S[simp, intro!]*: *finite*  $(S A)$

*<proof>*

**lemma** *S-A-consistent[simp, intro!]*: *nodes-edges-consistent*  $(S A) (A \cup A^{-1})$

*<proof>*

**end**

**context** *Prim* **begin**

**lemma** *invar1-initial*: *prim-invar1*  $\{\}$

*<proof>*

**lemma** *maintain-invar1*:

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

**assumes** *light-edge*: *light-edge*  $(S A) u v$

**shows** *prim-invar1*  $(\{(v,u)\} \cup A)$

$\wedge T\text{-measure1 } (\{(v,u)\} \cup A) < T\text{-measure1 } A$  (**is**  $?G1 \wedge ?G2$ )

*<proof>*

**lemma** *invar1-finish*:

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

**assumes** *FIN*:  $\text{edges } g \cap S A \times -S A = \{\}$

**shows** *is-MST*  $w rg (\text{graph } \{r\} A)$

*<proof>*

**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 set  $\Rightarrow$  nat **and** *g* :: 'v 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** *initQ* :: 'v  $\Rightarrow$  enat **where** *initQ*  $\equiv (\lambda-. \infty)(r := 0)$

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

Step

**definition** *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** *Qinter 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*  $\equiv (Qinter Q \pi u)(u:=\infty)$

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

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

**assumes** *Q u  $\neq$   $\infty$  u  $\neq$  r*

**shows** *u  $\notin$  S (A Q  $\pi$ )*

*<proof>*

**lemma** *prim-invar2-ctd-alt*:  $\text{prim-invar2-ctd } Q \pi \longleftrightarrow ($   
*let*  $A = A \ Q \ \pi; S = S \ A; cE = \text{edges } rg \cap (-S) \times S \ \text{in}$   
*prim-invar1*  $A$   
 $\wedge \pi \ r = \text{None} \wedge Q \ r = \infty$   
 $\wedge (\forall (u,v) \in cE. Q \ u \neq \infty)$   
 $\wedge (\forall u \ v. \pi \ u = \text{Some } v \longrightarrow v \in S \wedge (u,v) \in \text{edges } rg)$   
 $\wedge (\forall u \ d. Q \ u = \text{enat } d$   
 $\longrightarrow (\exists v. \pi \ u = \text{Some } v \wedge d = w \ \{u,v\} \wedge (\forall v'. (u,v') \in cE \longrightarrow d \leq w \ \{u,v'\})))$   
 $)$   
*<proof>*

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

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

**lemma** *Q'-init-eq*:  
 $Q' \ \text{init} \ Q \ \text{init} \ \pi \ r = (\lambda u. \ \text{if } (u,r) \in \text{edges } rg \ \text{then } \text{enat } (w \ \{u,r\}) \ \text{else } \infty)$   
*<proof>*

**lemma** *π'-init-eq*:  
 $\pi' \ \text{init} \ Q \ \text{init} \ \pi \ r = (\lambda u. \ \text{if } (u,r) \in \text{edges } rg \ \text{then } \text{Some } r \ \text{else } \text{None})$   
*<proof>*

**lemma** *A-init-eq*:  $A \ \text{init} \ Q \ \text{init} \ \pi = \{\}$   
*<proof>*

**lemma** *S-empty*:  $S \ \{\} = \{r\}$  *<proof>*

**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*  $(Q' \ Q \ \pi \ u) (\pi' \ Q \ \pi \ u) < T\text{-measure2 } Q \ \pi$  (**is** ?G2)  
*<proof>*

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

**end**  
*<proof>**<proof>*

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 π*  
**assumes** *invar2*: *prim-invar2-ctd* *Q π*  
**begin**

**sublocale** *Prim-Invar1-loc* *w g r A Q π*  
 ⟨*proof*⟩

**lemma** *upd-cond-alt*: *upd-cond* *Q π u v'*  $\longleftrightarrow$   
 $(v',u) \in \text{edges } g \wedge v' \notin S(A Q \pi) \wedge \text{enat } (w \{v',u\}) < Q v'$   
 ⟨*proof*⟩

**lemma** *π-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** *π-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-π-consistent*:  $\llbracket Q u = \text{enat } d; \pi u = \text{Some } v \rrbracket \implies d = w \{u,v\}$   
**and** *Q-min*:  $Q u = \text{enat } d$   
 $\implies (\forall v' \in S(A Q \pi). (u,v') \in \text{edges } rg \longrightarrow d \leq w \{u,v'\})$   
 ⟨*proof*⟩

**lemma** *π-def-on-S*:  $\llbracket u \in S(A Q \pi); u \neq r \rrbracket \implies \pi u \neq \text{None}$   
 ⟨*proof*⟩

**lemma** *π-def-on-edges-to-S*:  $\llbracket v \in S(A Q \pi); u \neq r; (u,v) \in \text{edges } rg \rrbracket \implies \pi u \neq \text{None}$   
 ⟨*proof*⟩

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

**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*  $(Q' Q \pi u) (\pi' Q \pi u) < T\text{-measure2 } Q \pi$  **(is ?G2)**  
 ⟨*proof*⟩

**end**

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

**lemma** *maintain-invar2-ctd*:  
**assumes** *INV*: *prim-invar2-ctd* *Q π*



**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>

**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**  $\pi\ u = \text{Some } v$  *light-edge* ( $S\ (A\ Q\ \pi)$ )  $v\ u$   
 <proof>

**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*  $Q\ \pi$   
 <proof>

**lemma** *not-invar2-ctd-init*:  
*prim-invar2-init*  $Q\ \pi \implies \neg \text{prim-invar2-ctd } Q\ \pi$   
 <proof>

**lemma** *invar2-init-init*: *prim-invar2-init*  $\text{init } Q\ \text{init } \pi$   
 <proof>

**lemma** *invar2-init*: *prim-invar2*  $\text{init } Q\ \text{init } \pi$   
 <proof>

**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* ( $Q'\ Q\ \pi\ u$ ) ( $\pi'\ Q\ \pi\ u$ ) (**is** ?G1)  
**and** *T-measure2* ( $Q'\ Q\ \pi\ u$ ) ( $\pi'\ Q\ \pi\ u$ )  $<$  *T-measure2*  $Q\ \pi$  (**is** ?G2)  
 <proof>

**lemma** *invar2-ctd-finish*:  
**assumes** *INV*: *prim-invar2-ctd*  $Q\ \pi$   
**assumes** *FIN*:  $Q = (\lambda-. \infty)$   
**shows** *is-MST*  $w\ \text{rg } (\text{graph } \{r\}) \{(u, v). \pi\ u = \text{Some } v\}$   
 <proof>

**lemma** *invar2-finish*:  
**assumes** *INV*: *prim-invar2*  $Q\ \pi$   
**assumes** *FIN*:  $Q = (\lambda-. \infty)$

**shows** *is-MST w rg* (graph {r} {(u, v).  $\pi u = \text{Some } v$ })  
 ⟨proof⟩

**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  
    $(Q, \pi)$   
 )  
 ⟨proof⟩

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

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

**definition**  $\bigwedge Q V \pi$ .  
 $\pi'\text{igen } Q \pi u \text{ adjs } v = (\text{if } v \notin \text{fst'set adjs} \text{ then } \pi v \text{ else } \pi' Q \pi u v)$

**context** **begin**

**private lemma** *Qc*:  
 $Q\text{igen } Q \pi u ((v, w \{u, v\}) \# \text{adjs}) x$   
 = (if  $x=v$  then  $Q\text{inter } Q \pi u v$  else  $Q\text{igen } Q \pi u \text{ adjs } x$ ) **for**  $x$   
 ⟨proof⟩ **lemma**  *$\pi c$* :  
 $\pi'\text{igen } Q \pi u ((v, w \{u, v\}) \# \text{adjs}) x$   
 = (if  $x=v$  then  $\pi' Q \pi u v$  else  $\pi'\text{igen } Q \pi u \text{ adjs } x$ ) **for**  $x$   
 ⟨proof⟩

**lemma** *foreach-refine-gen*:  
**assumes**  $\text{set adjs} \subseteq \{(v, d). (u, v) \in \text{edges } g \wedge w \{u, v\} = d\}$   
**shows** *foreach*  $u$  *adjs*  $(Q, \pi) = (Q\text{igen } Q \pi u \text{ adjs}, \pi'\text{igen } Q \pi u \text{ adjs})$   
 ⟨proof⟩

```

lemma foreach-refine:
  assumes set adjs =  $\{(v,d). (u,v) \in \text{edges } g \wedge w \{u,v\} = d\}$ 
  shows foreach u adjs  $(Q,\pi) = (Q \text{inter } Q \pi \ u, \pi' \ Q \ \pi \ u)$ 
  <proof>

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

```

```

lemma epair-eqD:  $\text{epair } e = \text{Some } (x,y) \implies (x \neq y \wedge e = \{x,y\})$ 
  <proof>

```

```

lemma epair-not-sng[simp]:  $\text{epair } e \neq \text{Some } (x,x)$ 
  <proof>

```

```

lemma epair-None[simp]:  $\text{epair } \{a,b\} = \text{None} \longleftrightarrow a = b$ 
  <proof>

```

### 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 ::  $'m \Rightarrow 'v \Rightarrow (('v \times \text{nat}) \text{ list}) \text{ option}$ 
  and M-invar
begin

```

```

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

```

```

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

```

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

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

$x \in \text{set } (\text{the-default } [] \ m) \longleftrightarrow (\exists xs. m = \text{Some } xs \wedge x \in \text{set } xs)$   
 $\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

**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**  $\text{add-edge1} :: 'v \times 'v \Rightarrow \text{nat} \Rightarrow 'm \Rightarrow 'm$  **where**

$\text{add-edge1} \equiv \lambda(u,v) \ d \ g. M\text{-update } u \ ((v,d) \# \text{the-default } [] (M\text{-lookup } g \ u)) \ g$

**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) = α*edges-aux* g  
 ⟨*proof*⟩

**lemma** *nodes-αg-aux*: *invar g* ⇒ *nodes* (αg g) = α*nodes-aux* g  
 ⟨*proof*⟩

**lemma** *card-doubleton-eq2[simp]*: *card* {a,b} = 2 ↔ a ≠ b ⟨*proof*⟩

**lemma** *the-dflt-Z-eq*: *the-default* 0 m = d ↔ (m = None ∧ d = 0 ∨ m = Some d)  
 ⟨*proof*⟩

**lemma** *adj-correct-aux*:  
*invar g* ⇒ *set* (*adj* g u) = {(v, d). (u, v) ∈ *edges* (αg g) ∧ αw g {u, v} = d}  
 ⟨*proof*⟩

**lemma** *invar-empty-aux*: *invar empty*  
 ⟨*proof*⟩

**lemma** *dist-fst-the-dflt-aux*: *distinct* (map *fst* (*the-default* [] m))  
 ↔ (∀ xs. m = Some xs → *distinct* (map *fst* xs))  
 ⟨*proof*⟩

**lemma** *invar-add-edge-aux*:  
 [[*invar* g; (u, v) ∉ *edges* (αg g); u ≠ v]] ⇒ *invar* (*add-edge* (u, v) d g)  
 ⟨*proof*⟩

**sublocale** *adt-wgraph* αw αg *invar adj empty add-edge*  
 ⟨*proof*⟩

**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-lookup* *Qi v*, *M-lookup*  $\pi i$  *v*) *of*

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

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

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

  )

**definition** *foreach-impl* :: 'q ⇒ 'm ⇒ 'v ⇒ ('v × nat) list ⇒ 'q × 'm **where**  
*foreach-impl* Qi πi u adjs = foldr (*foreach-impl-body* u) adjs (Qi, πi)

**definition** *outer-loop-impl* Qi πi ≡ while (λ(Qi, πi). ¬Q-is-empty Qi) (λ(Qi, πi).  
*let*  
 (u, -) = Q-getmin Qi;  
 adjs = G-adj g u;  
 (Qi, πi) = *foreach-impl* Qi πi u adjs;  
 Qi = Q-delete u Qi  
*in* (Qi, πi)) (Qi, πi)

**definition** *prim-impl* = (*let*  
 Qi = Q-update r 0 Q-empty;  
 πi = M-empty;  
 (Qi, πi) = *outer-loop-impl* Qi πi  
*in* πi)

The whole algorithm as one function

**lemma** *prim-impl-alt*: *prim-impl* = (*let*  
 — Initialization  
 (Q, π) = (Q-update r 0 Q-empty, M-empty);  
 — Main loop: Iterate until PQ is empty  
 (Q, π) =  
 while (λ(Q, π). ¬ Q-is-empty Q) (λ(Q, π). *let*  
 (u, -) = Q-getmin Q;  
 — Inner loop: Update for adjacent nodes  
 (Q, π) =  
 foldr ((λ(v, d) (Q, π). *let*  
 qv = Q-lookup Q v;  
 πv = M-lookup π v  
*in*  
 if v ≠ r ∧ (qv ≠ None ∨ πv = None) ∧ enat d < enat-of-option qv  
 then (Q-update v d Q, M-update v u π)  
 else (Q, π))  
 ) (G-adj g u) (Q, π);  
 Q = Q-delete u Q  
*in* (Q, π)) (Q, π)  
*in* π  
 )  
 ⟨*proof*⟩

## Abstraction of Result

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

- The map πi and set Vi satisfy their implementation invariants

- The  $\pi i$  encodes irreflexive edges consistent with the nodes determined by  $Vi$ . 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**  $G\text{-invar}[simp]: G\text{-invar } g$

**begin**

**sublocale** *Prim2*  $G\text{-}\alpha w g G\text{-}\alpha g r \langle \text{proof} \rangle$

### Abstraction of $Q$

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

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

**lemma** *Q- $\alpha$ -empty*:  $Q\text{-}\alpha Q\text{-empty} = (\lambda\cdot. \infty)$   
 $\langle \text{proof} \rangle$

**lemma** *Q- $\alpha$ -update*:  $Q\text{-invar } Q \Longrightarrow Q\text{-}\alpha (Q\text{-update } u d Q) = (Q\text{-}\alpha Q)(u := \text{enat } d)$   
 $\langle \text{proof} \rangle$

**lemma** *Q- $\alpha$ -is-empty*:  $Q\text{-invar } Q \Longrightarrow Q\text{-lookup } Q = \text{Map.empty} \longleftrightarrow Q\text{-}\alpha Q = (\lambda\cdot. \infty)$   
 $\langle \text{proof} \rangle$

**lemma** *Q- $\alpha$ -delete*:  $Q\text{-invar } Q \Longrightarrow Q\text{-}\alpha (Q\text{-delete } u Q) = (Q\text{-}\alpha Q)(u:=\infty)$   
 $\langle \text{proof} \rangle$

**lemma** *Q- $\alpha$ -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**



$\forall v. \text{enat } d \leq Q\text{-}\alpha \text{ } Qi \text{ } v \text{ (is ?G2)}$   
 <proof>

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

**end**

### 1.6.3 Refinement of Algorithm

**context**  $\text{Prim-Impl}$

**begin**

**lemma**  $\text{foreach-impl-correct}$ :

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

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

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

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

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

<proof><proof>

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

**lemma**  $\text{prim-invar-impl-init}$ :  $\text{prim-invar-impl } (Q\text{-update } r \text{ } 0 \text{ } Q\text{-empty}) \text{ } M\text{-empty}$

<proof>

**lemma**  $\text{maintain-prim-invar-impl}$ :

**assumes**

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

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

$MIN$ :  $Q\text{-getmin } Qi = (u, d)$  **and**

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

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

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

<proof>

**lemma**  $\text{maintain-prim-invar-impl-presentation}$ :

**assumes**

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

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

$MIN$ :  $Q\text{-getmin } Qi = (u, d)$  **and**

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

**shows** *prim-invar-impl* (*Q-delete* *u Qi'*)  $\pi i'$   
 $\wedge$  *T-measure-impl* (*Q-delete* *u Qi'*,  $\pi i'$ )  $<$  *T-measure-impl* (*Qi*,  $\pi i$ )  
 $\langle$ *proof* $\rangle$

**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*  $\pi$ )  
 $\langle$ *proof* $\rangle$

**lemma** *prim-impl-correct*:  
**assumes** *prim-impl* =  $\pi i$   
**shows**  
*invar-MST*  $\pi i$  (**is** ?*G1*)  
*is-MST* (*G- $\alpha$ w* *g*) (*component-of* (*G- $\alpha$ g* *g*) *r*) ( $\alpha$ -*MST*  $\pi i$ ) (**is** ?*G2*)  
 $\langle$ *proof* $\rangle$  $\langle$ *proof* $\rangle$   
**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-empty* = *G.empty*  
**and** *G-add-edge* = *G.add-edge*  
**and** *G-add-edge1* = *G.add-edge1*  
**and** *G-adj* = *G.adj*  
**and** *G-from-list* = *G.from-list*  
**and** *G-valid-wgraph-repr* = *G.valid-wgraph-repr*  
 $\langle$ *proof* $\rangle$

**lemma** *G-from-list-unfold*: *G-from-list* = *G.from-list*  
 $\langle$ *proof* $\rangle$

**lemma** [*code*]: *G-from-list* *l* = *foldr* ( $\lambda(e, d)$ . *G-add-edge* *e d*) *l* *G-empty*  
 $\langle$ *proof* $\rangle$

**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*  
 $\langle$ *proof* $\rangle$

**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
⟨proof⟩
```

**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
⟨proof⟩
```

**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

*<proof>*

**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

*<proof>*

## 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 × 'v) ⇒ 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 ≡ {(u,v) . w (u,v) ≠ ∞}
```

### 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**  $path-append[simp]$ :  
 $path\ u\ (ls1@ls2)\ v \longleftrightarrow (\exists\ w.\ path\ u\ ls1\ w \wedge path\ w\ ls2\ v)$   
 $\langle proof \rangle$

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

**lemma**  $triv-path$ :  $path\ u\ [w\ (u,v)]\ v \langle proof \rangle$

Shortcut for the set of all paths between two nodes

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

**lemma**  $paths-ne$ :  $paths\ u\ v \neq \{\}$   $\langle proof \rangle$

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**  $p = p1@w\ (x,y)\#p2\ x \in S\ y \notin S\ path\ u\ p1\ x\ path\ y\ p2\ v$   
 $\langle proof \rangle$

### 2.1.2 Distance

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

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

**lemma**  $obtain-shortest-path$ :  
**obtains**  $p$  **where**  $path\ s\ p\ u\ \delta\ s\ u = sum-list\ p$   
 $\langle proof \rangle$

**lemma**  $shortest-path-least$ :  
 $path\ s\ p\ u \implies \delta\ s\ u \leq sum-list\ p$   
 $\langle proof \rangle$

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

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

*<proof>*

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>*

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 p<sub>1</sub> x path x p<sub>2</sub> 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>*

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*:

$\text{invar } g \Longrightarrow \alpha g e = \infty \Longrightarrow \text{invar } (\text{add-edge } e d g)$

$\text{invar } g \Longrightarrow \alpha g e = \infty \Longrightarrow \alpha (\text{add-edge } e d g) = (\alpha g)(e:=\text{enat } d)$

**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]*:  $WGraph.edges (\lambda-. \infty) = \{\}$   
 $\langle proof \rangle$

**lemma** *edges-insert[simp]*:  
 $WGraph.edges (g(e:=enat d)) = Set.insert e (WGraph.edges g)$   
 $\langle proof \rangle$

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]*:  
 $valid-graph-rep\ l \longleftrightarrow distinct\ (map\ (\lambda(u,v,-). (u,v))\ l)$   
 $\langle proof \rangle$

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

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

**lemma** *valid-graph-rep-ex1*:  
 $valid-graph-rep\ l \implies \exists! w. \forall u v d. w(u,v) = enat\ d \longleftrightarrow (u,v,d) \in set\ l$   
 $\langle proof \rangle$

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

```

lemma from-list-correct-aux:
  assumes valid-graph-rep l
  assumes edges-undef l (α g)
  assumes invar g
  defines  $g' \equiv \text{fold } (\lambda(u,v,d). \text{add-edge } (u,v) \ d) \ l \ g$ 
  shows invar g'
    and  $(\forall u \ v \ d. \alpha \ g' \ (u,v) = \text{enat } d \longleftrightarrow \alpha \ g \ (u,v) = \text{enat } d \vee (u,v,d) \in \text{set } l)$ 
  <proof>

```

```

lemma from-list-correct':
  assumes valid-graph-rep l
  shows invar (from-list l)
    and  $(u,v,d) \in \text{set } l \longleftrightarrow \alpha \ (\text{from-list } l) \ (u,v) = \text{enat } d$ 
  <proof>

```

```

lemma from-list-correct:
  assumes valid-graph-rep l
  shows invar (from-list l) α (from-list l) = wgraph-of-list l
  <proof>

```

**end**

**end**

## 2.3 Abstract Dijkstra Algorithm

```

theory Dijkstra-Abstract
imports Directed-Graph
begin

```

### 2.3.1 Abstract Algorithm

```

type-synonym  $'v \ \text{estimate} = 'v \Rightarrow \text{enat}$ 

```

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 ::  $'v \Rightarrow 'v \ \text{estimate} \Rightarrow 'v \ \text{estimate}$ 
  where  $\text{relax-outgoing } u \ D \equiv \lambda v. \min (D \ v) (D \ u + w \ (u,v))$ 

```

Initialization

```

definition  $\text{init}D \equiv (\lambda-. \ \infty)(s:=0)$ 
definition  $\text{init}S \equiv \{\}$ 

```

Relaxing will never increase estimates

**lemma** *relax-mono: relax-outgoing*  $u \ D \ v \leq D \ v$   
*<proof>*

**definition** *all-dnodes*  $\equiv \text{Set.insert } s \ \{ v \ . \ \exists u. \ w \ (u,v) \neq \infty \}$

**definition** *unfinished-dnodes*  $S \equiv \text{all-dnodes} - S$

**lemma** *unfinished-nodes-subset: unfinished-dnodes*  $S \subseteq \text{all-dnodes}$   
*<proof>*

**end**

## Invariant

The invariant is defined as locale

**locale** *Dijkstra-Invar* = *Dijkstra*  $w \ s$  **for**  $w$  **and**  $s :: 'v +$   
**fixes**  $D :: 'v \text{ estimate}$  **and**  $S :: 'v \text{ set}$   
**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-invar*  $\equiv \text{Dijkstra-Invar } w \ s$

The invariant holds for the initial state

**theorem** (**in** *Dijkstra*) *invar-init: D-invar*  $\text{init} \ D \ \text{init} \ S$   
*<proof>*

Relaxing some edges maintains the upper bound property

**lemma** *maintain-upper-bound:  $\delta \ s \ u \leq (\text{relax-outgoing } v \ D) \ u$*   
*<proof>*

Relaxing edges will not affect nodes with already precise estimates

**lemma** *relax-precise-id:  $D \ v = \delta \ s \ v \implies \text{relax-outgoing } u \ D \ v = \delta \ s \ v$*   
*<proof>*

In particular, relaxing edges will not affect finished nodes

**lemma** *relax-finished-id:  $v \in S \implies \text{relax-outgoing } u \ D \ v = D \ v$*   
*<proof>*

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**  $MIN: \forall v. v \notin S \longrightarrow D u \leq D v$   
**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>*

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**  $D\text{-invar}$  (*relax-outgoing*  $u$   $D$ ) (*Set.insert*  $u$   $S$ )  
*<proof>*

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>*

A step decreases the set of unfinished nodes.

**lemma** *unfinished-nodes-decr*:

**assumes**  $UNS: u \notin S$   
**assumes**  $UNI: D u < \infty$   
**shows**  $unfinished\text{-}dnodes$  (*Set.insert*  $u$   $S$ )  $\subset$   $unfinished\text{-}dnodes$   $S$   
*<proof>*

**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*  
 $\langle$ *proof* $\rangle$

**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*  
 $\langle$ *proof* $\rangle$

**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*  
) )  
 $\langle$ *proof* $\rangle$

## 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*  
 $\langle$ *proof* $\rangle$

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*

(*relax-outgoing'*  $u du V Q$ )( $u := \text{None}$ ) ( $V(u \mapsto du)$ )

(*relax-outgoing*  $u D$ ) (*Set.insert*  $u S$ )

*<proof>*

Refinement of initial state

**definition** *initQ*  $\equiv \text{Map.empty}(s \mapsto 0)$

**definition** *initV*  $\equiv \text{Map.empty}$

**lemma** *coupling-init*:

*coupling* *initQ* *initV* *initD* *initS*

*<proof>*

**lemma** *coupling-cond*:

**assumes** *coupling*  $Q V D S$

**shows** ( $Q = \text{Map.empty}$ )  $\longleftrightarrow$  ( $\forall u. u \notin S \longrightarrow D u = \infty$ )

*<proof>*

Termination argument: Refinement of unfinished nodes.

**definition** *unfinished-dnodes'*  $V \equiv \text{unfinished-dnodes } (\text{dom } V)$

**lemma** *coupling-unfinished*:

*coupling*  $Q V D S \implies \text{unfinished-dnodes}' V = \text{unfinished-dnodes } S$

*<proof>*

## Implementing graph by successor list

**definition** *relax-outgoing''*  $l du V Q = \text{fold } (\lambda(d,v) Q.$

  case  $Q v$  of None  $\Rightarrow$  if  $v \in \text{dom } V$  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  $l = \{(d,v). w(u,v) = \text{enat } d\}$

**shows** *relax-outgoing''*  $l du V Q = \text{relax-outgoing}' u du V Q$

*<proof>*

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* ::  $'m \Rightarrow 'v \Rightarrow ((\text{nat} \times 'v) \text{ list}) \text{ option}$  **and** *M-invar*

**begin**

**definition**  $\alpha$  ::  $'m \Rightarrow ('v)$  *wgraph* **where**

$\alpha$  *g*  $\equiv \lambda(u,v).$  *case M-lookup g u of*

*None*  $\Rightarrow \infty$

| *Some l*  $\Rightarrow$  *if*  $\exists d. (d,v) \in \text{set } l$  *then* *enat (SOME d. (d,v)  $\in$  set l)* *else*  $\infty$

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

*M-invar g*

$\wedge (\forall l \in \text{ran } (M\text{-lookup } g).$  *distinct (map snd l)*)

$\wedge$  *finite (WGraph.edges ( $\alpha$  g))*

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

*succ g v* = *the-default [] (M-lookup g v)*

**definition** *empty-graph* ::  $'m$  **where** *empty-graph* = *M-empty*

**definition** *add-edge* ::  $'v \times 'v \Rightarrow \text{nat} \Rightarrow 'm \Rightarrow 'm$  **where**

*add-edge*  $\equiv \lambda(u,v) d g.$  *M-update u ((d,v) # the-default [] (M-lookup g u)) g*

**sublocale** *adt-finite-wgraph invar succ empty-graph add-edge*  $\alpha$

*<proof>*

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- $\alpha$ 
+ 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- $\alpha$  :: 'g  $\Rightarrow$  ('v) wgraph and G-invar G-succ G-empty G-add

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

**context** PrioMap **begin**

```

lemma map-getminE:
  assumes getmin m = (k,p) invar m lookup m  $\neq$  Map.empty
  obtains lookup m k = Some p  $\forall$  k' p'. lookup m k' = Some p'  $\longrightarrow$  p  $\leq$  p'
  <proof>

```

**end**

```

locale Dijkstra-Impl-Defs = Dijkstra-Impl-Adts where G- $\alpha$  = G- $\alpha$ 
+ Dijkstra <G- $\alpha$  g> s
for G- $\alpha$  :: 'g  $\Rightarrow$  ('v::linorder) wgraph and g s

```

```

locale Dijkstra-Impl = Dijkstra-Impl-Defs where G- $\alpha$  = G- $\alpha$ 
for G- $\alpha$  :: 'g  $\Rightarrow$  ('v::linorder) wgraph
+

```



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

**lemma**  $finite\text{-all}\text{-dnodes}[simp, intro!]$ :  $finite\ all\text{-dnodes}$   
 $\langle proof \rangle$

**lemma**  $finite\text{-unfinished}\text{-dnodes}[simp, intro!]$ :  $finite\ (unfinished\text{-dnodes } S)$   
 $\langle proof \rangle$

**lemma** (**in**  $-$ )  $fold\text{-refine}$ :

**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**  $I\ (fold\ f\ l\ s) \wedge \alpha\ (fold\ f\ l\ s) = fold\ f'\ l\ (\alpha\ s)$

$\langle proof \rangle$

**definition** (**in**  $Dijkstra\text{-Impl}\text{-Defs}$ )  $Q\text{-relax}\text{-outgoing } u\ du\ V\ Q = fold\ (\lambda(d,v)\ Q.$

$case\ Q\text{-lookup } Q\ v\ of$

$None \implies if\ M\text{-lookup } V\ v \neq None\ then\ Q\ else\ Q\text{-update } v\ (du+d)\ Q$

$| Some\ d' \implies Q\text{-update } v\ (min\ (du+d)\ d')\ Q)\ ((G\text{-succ } g\ u))\ Q$

**lemma**  $Q\text{-relax}\text{-outgoing}[simp]$ :

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

**shows**  $Q\text{-invar } (Q\text{-relax}\text{-outgoing } u\ du\ V\ Q)$

$\wedge Q\text{-lookup } (Q\text{-relax}\text{-outgoing } u\ du\ V\ Q)$

$= relax\text{-outgoing}'\ u\ du\ (M\text{-lookup } V)\ (Q\text{-lookup } Q)$

$\langle proof \rangle$

**definition** (**in**  $Dijkstra\text{-Impl}\text{-Defs}$ )  $D\text{-invar}\text{-impl } Q\ V \equiv$

$Q\text{-invar } Q \wedge M\text{-invar } V \wedge D\text{-invar}'\ (Q\text{-lookup } Q)\ (M\text{-lookup } V)$

**definition** (**in**  $Dijkstra\text{-Impl}\text{-Defs}$ )

$Q\text{-init}Q \equiv Q\text{-update } s\ 0\ Q\text{-empty}$

**lemma**  $Q\text{-init}\text{-}Q[simp]$ :

**shows**  $Q\text{-invar } (Q\text{-init}Q)\ Q\text{-lookup } (Q\text{-init}Q) = initQ$

$\langle proof \rangle$

**definition** (**in**  $Dijkstra\text{-Impl}\text{-Defs}$ )

$M\text{-init}V \equiv M\text{-empty}$

**lemma**  $M\text{-init}S[simp]$ :  $M\text{-invar } M\text{-init}V\ M\text{-lookup } M\text{-init}V = initV$

$\langle proof \rangle$

**term**  $Q\text{-getmin}$

**definition** (**in**  $Dijkstra\text{-Impl}\text{-Defs}$ )

$dijkstra\text{-loop} \equiv while\ (\lambda(Q,V). \neg Q\text{-is}\text{-empty } Q)\ (\lambda(Q,V).$

$let$

```

    (u,du) = Q-getmin Q;
    Q = Q-relax-outgoing u du V Q;
    Q = Q-delete u Q;
    V = M-update u du V
  in
    (Q,V)
) (Q-initQ,M-initV)

```

**definition** (in *Dijkstra-Impl-Defs*) *dijkstra*  $\equiv$  *snd dijkstra-loop*

**lemma** *transfer-preconditions*:

```

  assumes coupling Q V D S
  shows Q u = Some du  $\longleftrightarrow$  D u = enat du  $\wedge$  u  $\notin$  S
  <proof>

```

**lemma** *dijkstra-loop-invar-and-empty*:

```

  shows case dijkstra-loop of (Q,V)  $\Rightarrow$  D-invar-impl Q V  $\wedge$  Q-is-empty Q
  <proof>

```

**lemma** *dijkstra-correct*:

```

  M-invar dijkstra
  M-lookup dijkstra u = Some d  $\longleftrightarrow$   $\delta$  s u = enat d
  <proof>

```

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
  <proof>

```

**global-interpretation** *Dijkstra-Impl-Adts*

```

  G. $\alpha$  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
  <proof>

```

**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*  
*<proof>*

**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*  
*<proof>*

**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*  
    *<proof>*

**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  $\Rightarrow \neg$ valid-graph-rep l
  | Some D  $\Rightarrow$ 
    valid-graph-rep l
     $\wedge$  M.invar D
     $\wedge (\forall u d. \text{lookup } D u = \text{Some } d \iff \text{WGraph}.\delta (\text{wgraph-of-list } l) s u = \text{enat } d)$ 
    <proof>

```

```

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.