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

Peter Lammich Tobias Nipkow

March 17, 2025

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

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 $'v\ ugraph$
  $= \{\ (V{::}'v\ set\ ,\ E).\ E \subseteq V \times V \wedge \mathit{finite}\ V \wedge \mathit{sym}\ E \wedge \mathit{irrefl}\ E\ \}$
  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 $\lambda V\ E.\ \mathit{if\ finite}\ V \wedge \mathit{finite}\ E\ \mathit{then}\ (V \cup \mathit{fst}`E \cup \mathit{snd}`E,\ (E \cup E^{-1}) - \mathit{Id})\ \mathit{else}\ (\{\},\{\})$
  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*: $\mathit{edges}\ g \subseteq \mathit{nodes}\ g \times \mathit{nodes}\ g$
  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*: $[\![\mathit{finite}\ V;\ \mathit{finite}\ E]\!] \Longrightarrow \mathit{nodes}\ (\mathit{graph}\ V\ E) = V \cup \mathit{fst}`E \cup \mathit{snd}`E$

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

lemma *edges-graph*: $[\![\mathit{finite}\ V;\ \mathit{finite}\ E]\!] \Longrightarrow \mathit{edges}\ (\mathit{graph}\ V\ E) = (E \cup E^{-1}) - \mathit{Id}$
  unfolding *edges-def nodes-def graph-def* **by** *transfer auto*

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

lemma *nodes-edges-graph-presentation*: $[\![\mathit{finite}\ V;\ \mathit{finite}\ E]\!]$
    $\Longrightarrow \mathit{nodes}\ (\mathit{graph}\ V\ E) = V \cup \mathit{fst}`E \cup \mathit{snd}`E \wedge \mathit{edges}\ (\mathit{graph}\ V\ E) = E \cup E^{-1}$
$- \mathit{Id}$
  by (*simp add*: *graph-accs*)

lemma *graph-eq*[*simp*]: *graph* (*nodes g*) (*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*!]: *finite* (*edges g*)
  using *edges-subset finite-subset* **by** *fastforce*

lemma *graph-cases*[*cases type*]: **obtains** *V E*
  where $g = graph\ V\ E\ finite\ V\ finite\ E\ E \subseteq V \times V\ sym\ E\ irrefl\ E$
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*: $g=g' \longleftrightarrow nodes\ g = nodes\ g' \wedge edges\ g = edges\ g'$
  unfolding *edges-def nodes-def graph-def* **by** *transfer auto*

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

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

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

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

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

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

1.1.2 Connectedness Relation

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

lemma *trancl-edges-subset*: $(edges\ g)^+ \subseteq nodes\ g \times nodes\ g$
  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*]: *nodes* (*ins-node v g*) = *insert v* (*nodes g*)
  and *edges-ins-nodes*[*simp*]: *edges* (*ins-node v g*) = *edges g*
  unfolding *ins-node-def* **by** (*auto simp*: *graph-accs edges-sym′*)

lemma *nodes-ins-edge*[*simp*]: *nodes* (*ins-edge e g*) = {*fst e*, *snd e*} ∪ *nodes g*
  and *edges-ins-edge*:
    edges (*ins-edge e g*)
      = (*if fst e* = *snd e then edges g else* {*e*,*prod.swap e*}∪(*edges g*))
  unfolding *ins-edge-def*
  apply (*all* ‹*cases e*›)
  by (*auto simp*: *graph-accs dest*: *edges-sym′*)

lemma *edges-ins-edge′*[*simp*]:
  $u \neq v \Longrightarrow$ *edges* (*ins-edge* (*u*,*v*) *g*) = {(*u*,*v*),(*v*,*u*)} ∪ *edges g*
  by (*auto simp*: *edges-ins-edge*)

lemma *edges-ins-edge-ss*: *edges g* ⊆ *edges* (*ins-edge e g*)
  by (*auto simp*: *edges-ins-edge*)

lemma *nodes-join*[*simp*]: *nodes* (*graph-join* g_1 g_2) = *nodes* g_1 ∪ *nodes* g_2
  and *edges-join*[*simp*]: *edges* (*graph-join* g_1 g_2) = *edges* g_1 ∪ *edges* g_2
  unfolding *graph-join-def*
  by (*auto simp*: *graph-accs dest*: *edges-sym′*)

lemma *nodes-restrict-nodes*[*simp*]: *nodes* (*restrict-nodes g V*) = *nodes g* ∩ *V*
  and *edges-restrict-nodes*[*simp*]: *edges* (*restrict-nodes g V*) = *edges g* ∩ $V \times V$
  unfolding *restrict-nodes-def*
  by (*auto simp*: *graph-accs dest*: *edges-sym′*)

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

lemma *unrestricte-edges*: *edges* (*restrict-edges g E*) ⊆ *edges g* **by** *auto*
lemma *unrestrictn-edges*: *edges* (*restrict-nodes g V*) ⊆ *edges g* **by** *auto*

lemma *unrestrict-nodes*: *nodes* (*restrict-edges g E*) ⊆ *nodes g* **by** *auto*

1.1.4 Paths

fun *path* **where**
  path g u [] *v* ⟷ *u*=*v*
| *path g u* (*e*#*ps*) *w* ⟷ (∃ *v*. *e*=(*u*,*v*) ∧ *e*∈*edges g* ∧ *path g v ps w*)

lemma *path-emptyI*[*intro*!]: *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 $\Longrightarrow$ (*v*,*w*)$\in$*edges g* $\Longrightarrow$ *path g u* (*p*@[(*v*,*w*)]) *w*
(*u*,*v*)$\in$*edges g* $\Longrightarrow$ *path g v p w* $\Longrightarrow$ *path g u* ((*u*,*v*)#*p*) *w*
path g u p1 v $\Longrightarrow$ *path g v p2 w* $\Longrightarrow$ *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* (λ(*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* $\Longrightarrow$ *path g u* (*revp p*) *v* **by** *simp*

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

lemma *path-edges*: *path g u p v* $\Longrightarrow$ *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 $\Longrightarrow$ *set p* $\subseteq$ *edges* $g_2]\!]$ $\Longrightarrow$ *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'* $\Longrightarrow$ *path g u p v* $\Longrightarrow$ *path g' u p v*
by (*meson path-edges path-graph-cong subset-trans*)

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

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

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

Paths and Connectedness

lemma *rtrancl-edges-iff-path*: $(u,v) \in (edges\ g)^*$ $\longleftrightarrow$ ($\exists$ *p*. *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 $(u,v) \in (edges\ g)^*$ **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* $\Longrightarrow$ $(u,v) \in (edges\ g)^*$
  by (*auto simp*: *rtrancl-edges-iff-path*)

Simple Paths

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

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

lemma *in-uedge-conv*[*simp*]: $x \in uedge\ (u,v) \longleftrightarrow x=u \lor x=v$
  by (*auto simp*: *uedge-def*)

lemma *uedge-eq-iff*: $uedge\ (a,b) = uedge\ (c,d) \longleftrightarrow a=c \land b=d \lor a=d \land b=c$
  by (*auto simp*: *uedge-def doubleton-eq-iff*)

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

lemma *uedge-in-set-eq*: $uedge\ (u,\ v) \in uedge$ ‘ $S \longleftrightarrow (u,v) \in S \lor (v,u) \in S$
  by (*auto simp*: *uedge-def doubleton-eq-iff*)

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

lemma *simple-empty*[*simp*]: *simple* []
  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 $g\ u\ p\ v \Longrightarrow \exists\, p'.$ *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.prems* **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.prems* **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.prems* **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 $\forall (u,v) \in set\ p.\ P\ u \wedge P\ v$
| 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* $\forall (u,v) \in set\ p.\ P\ u \wedge P\ v$)
case *True* **with** *that* **show** *?thesis* **by** *blast*
next
case *False*
with P ‹$P\ u$› **have** $\exists (u_1,v_1) \in set\ p.\ P\ u_1 \wedge \neg P\ v_1$
apply *clarsimp* **apply** (*induction p arbitrary*: u) **by** *auto*
then obtain u_1 v_1 **where** $(u_1,v_1) \in set\ p$ **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* ‹$\neg P\ v_1$› ‹$P\ v$›] **obtain** u_2 v_2
where $(u_2,v_2) \in set\ p_{23}$ $\neg P\ u_2$ $P\ v_2$ **.**
then show *thesis* **using** *PRED1*
by (*auto simp*: *in-set-conv-decomp intro*: *that*)
qed

lemma *find-crossing-edge-rtrancl*:
assumes $(u,v) \in (edges\ g)^*$ $\neg P\ u$ $P\ v$
obtains u' v' **where** $(u',v') \in edges\ g$ $\neg P\ u'$ $P\ v'$
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**
$(x,y) \in set\ p$ $x \in S$ $y \notin S$
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: $(x,y) \in set\ p$ $x \in S$ $y \notin S$ **by** *blast*
then obtain *p1* *p2* **where** [*simp*]: $p=p1 @ [(x,y)] @ p2$
by (*auto simp*: *in-set-conv-decomp*)

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

from ‹*path* g u p v› **have** *P1*: *path* g u *p1* x **and** *P2*: *path* g y *p2* v **by** *auto*
from ‹*simple* p›
have *uedge* $(x,y) \notin set$ (*map uedge p1*) *uedge* $(x,y) \notin set$ (*map 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* $g \equiv \nexists p\ u.\ p{\neq}[] \wedge simple\ p \wedge path\ g\ u\ p\ u$

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

lemma *cycle-freeI*:
  assumes $\bigwedge p\ u.\ [\![\ path\ g\ u\ p\ u;\ p{\neq}[];\ simple\ p\ ]\!] \Longrightarrow False$
  shows *cycle-free g*
  using *assms* **unfolding** *cycle-free-def* **by** *auto*

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

lemma *cycle-free-antimono*: $edges\ g \subseteq edges\ g' \Longrightarrow cycle\text{-}free\ g' \Longrightarrow cycle\text{-}free\ g$
  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*: $edges\ g = \{\} \Longrightarrow cycle\text{-}free\ g$
  by (*rule cycle-freeI*) (*auto simp*: *neq-Nil-conv*)

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 $p{=}p'$
  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* = *cycle-freeD*[*OF CF*]

  from *Cons.prems* **obtain** u' **where**
    [*simp*]: $e{=}(u{,}u')$
    and P': $(u{,}u'){\notin}set\ p\ (u'{,}u){\notin}set\ p\ (u{,}u'){\in}edges\ g$
    by (*auto simp*: *uedge-in-set-eq*)

```
with Cons.prems obtain sp₁ where
  SP1: path g u ((u,u′)#sp₁) v simple ((u,u′)#sp₁)
  by blast

from Cons.prems obtain u′′ p′′ where
  [simp]: p′ = (u,u′′)#p′′
  and P′′: (u,u′′)∉set p′′ (u′′,u)∉set p′′ (u,u′′)∈edges g
  apply (cases p′)
  subgoal by auto (metis Cons.prems(1) Cons.prems(3) CF list.distinct(1))
  by (auto simp: uedge-in-set-eq)
with Cons.prems obtain sp₂ where
  SP2: path g u ((u,u′′)#sp₂) v simple ((u,u′′)#sp₂)
  by blast

have u′′=u′ proof (rule ccontr)
  assume [simp, symmetric, simp]: u′′≠u′

  have AUX1: (u,x)∉set sp₁ for x
  proof
    assume (u, x) ∈ set sp₁
   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)∉set sp₁ for x
  proof
    assume (x, u) ∈ set sp₁

   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.prems(1) Cons.prems(3) Un-iff
     AUX1 ‹e = (u, u′)› 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)∉set sp₂ for x
  proof
    assume (u, x) ∈ set sp₂
    then obtain sp′ sp′′ where [simp]: sp₂ = 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)∉set sp₂ for x
  proof
```

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

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

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

with *Cons.IH*[*of* u' p''] *Cons.prems* **show** *?case* **by** *simp*
qed

Characterization by Removing Edge

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

```
  subgoal premises prems for p u proof -
    from ‹p≠[]› ‹path g u p u› obtain v p' where
      [simp]: p=(u,v)#p' and (u,v)∈edges g path g v p' u
      by (cases p) auto
    from ‹simple p› have simple p' uedge (u,v) ∉ set (map uedge p') by auto
    hence (u,v)∉set p' (v,u)∉set p' by (auto simp: uedge-in-set-eq)
    with ‹path g v p' u›
      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)∈(edges ?rg)*
      by (meson path-rev rtrancl-edges-iff-path)
    with prems(1) ‹(u,v)∈edges g› show False by auto
  qed
  done

lemma cycle-free-altI:
  assumes ⋀u v. ⟦ (u,v)∈edges g; (u,v)∈(edges g − {(u,v),(v,u)})* ⟧ ⟹ False
  shows cycle-free g
  unfolding cycle-free-alt using assms by (force)

lemma cycle-free-altD:
  assumes cycle-free g
  assumes (u,v)∈edges g
  shows (u,v)∉(edges g − {(u,v),(v,u)})*
  using assms unfolding cycle-free-alt by (auto)



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

lemma *connectedI*[*intro?*]:
  assumes ⋀*u v*. ⟦*u*∈*nodes g*; *v*∈*nodes g*⟧ ⟹ (*u*,*v*)∈(*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*)*
  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*)* ‘‘{*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*)* ‘‘ *reachable-nodes g r* $\subseteq$ *reachable-nodes g r*
by (*auto simp*: *reachable-nodes-def*)

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

lemma *reachable-nodes-steps*′:
assumes *u* $\in$ *reachable-nodes g r* (*u*, *v*) $\in$ (*edges g*)*
shows *v*$\in$*reachable-nodes g r* (*u*, *v*) $\in$ (*edges* (*component-of g r*))*
proof −
show *v*$\in$*reachable-nodes g r* **using** *reachable-nodes-steps assms* **by** *fast*
show (*u*, *v*) $\in$ (*edges* (*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*$\notin$*nodes g* $\Longrightarrow$ *reachable-nodes g r* = $\{r\}$
by (*force elim*: *converse-rtranclE simp*: *reachable-nodes-def intro*: *nodesI*)

lemma *nodes-of-component*[*simp*]: *nodes* (*component-of g r*) = *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* $\in$ *reachable-nodes g r v* $\in$ *reachable-nodes g r*
hence (*u*,*r*)$\in$(*edges g*)* (*r*,*v*)$\in$(*edges g*)*
by (*auto simp*: *reachable-nodes-def dest*: *rtrancl-edges-sym*′)

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

lemma *component-edges-subset*: *edges* (*component-of g r*) ⊆ *edges g*
by (*auto simp*: *component-of-def*)

lemma *component-path*: *u*∈*nodes* (*component-of g r*) ⟹
path (*component-of g r*) *u p v* ⟷ *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*: *cycle-free g* ⟹ *cycle-free* (*component-of g r*)
by (*meson component-edges-subset cycle-free-antimono*)

lemma *component-of-connected-graph*:
⟦*connected g*; *r*∈*nodes g*⟧ ⟹ *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*∉*nodes g* ⟹ *component-of g r* = *graph* {*r*} {}
by (*clarsimp simp*: *graph-eq-iff component-of-def reachable-not-node graph-accs*)

1.1.8 Trees

definition *tree g* ≡ *connected g* ∧ *cycle-free g*

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

lemma *component-of-tree*: *tree T* ⟹ *tree* (*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*∈*nodes* g_1 *v*∈*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∈nodes ?g' b∈nodes ?g'

  have ESS: (edges g₁)* ⊆ (edges ?g')* (edges g₂)* ⊆ (edges ?g')*
    using edges-ins-edge-ss
    by (force intro!: rtrancl-mono)+

  have UV: (u,v)∈(edges ?g')*
    by (simp add: edges-ins-edge r-into-rtrancl)

  show (a,b)∈(edges ?g')*
  proof −
    {
      assume a∈nodes g₁ b∈nodes g₁
     hence ?thesis using ‹connected g₁› ESS(1) unfolding connected-def by blast
    } moreover {
      assume a∈nodes g₂ b∈nodes g₂
     hence ?thesis using ‹connected g₂› ESS(2) unfolding connected-def by blast
    } moreover {
      assume a∈nodes g₁ b∈nodes g₂
      with connectedD[OF CONN(1)] connectedD[OF CONN(2)] ESS
      have ?thesis by (meson UV IN-NODES contra-subsetD rtrancl-trans)
    } moreover {
      assume a∈nodes g₂ b∈nodes g₁
      with connectedD[OF CONN(1)] connectedD[OF CONN(2)] ESS
      have ?thesis
        by (meson UV IN-NODES contra-subsetD rtrancl-edges-sym' rtrancl-trans)
    }
    ultimately show ?thesis using A IN-NODES by auto
  qed
qed


lemma join-cycle-free:
  assumes CYCF: cycle-free g₁ cycle-free g₂
  assumes DJ: nodes g₁ ∩ nodes g₂ = {}
  assumes IN-NODES: u∈nodes g₁ v∈nodes g₂
  shows cycle-free (ins-edge (u,v) (graph-join g₁ g₂)) (is cycle-free ?g')
proof (rule cycle-freeI)
  fix p a
  assume P: path ?g' a p a p≠[] simple p
  from path-endpoints[OF this(1,2)] IN-NODES
    have A-NODE: a∈nodes g₁ ∪ nodes g₂
    by auto
  thus False proof
    assume N1: a∈nodes g₁
    have set p ⊆ nodes g₁ × nodes g₁
    proof (cases
     rule: find-crossing-edges-on-path[where P=λx. x∈nodes g₁, OF P(1) N1 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′ ≡ (edges T − {(x,y),(y,x)})
  obtains T1 T2 where
    tree T1 tree T2
    nodes T1 ∩ nodes T2 = {} nodes T = nodes T1 ∪ nodes T2
    edges T1 ∪ edges T2 = E′
    nodes T1 = { u. (x,u)∈E′*} nodes T2 = { u. (y,u)∈E′*}
    x∈nodes T1 y∈nodes T2
proof −

  define N1 where N1 = { u. (x,u)∈E′* }
  define N2 where N2 = { u. (y,u)∈E′* }

  define T1 where T1 = restrict-nodes T N1
  define T2 where T2 = 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 ‹cycle-free T› have cycle-free T1 cycle-free T2
    unfolding T1-def T2-def
    using cycle-free-antimono unrestrictn-edges by blast+

  from ‹(x,y) ∈ edges T› have XYN: x∈nodes T y∈nodes T
    using edges-subset by auto
  from XYN have [simp]: nodes T1 = N1 nodes T2 = N2
    unfolding T1-def T2-def N1-def N2-def unfolding E′-def
    apply (safe)
    apply (all ‹clarsimp›)
    by (metis DiffD1 nodesI(2) rtrancl.simps)+

  have x∈N1 y∈N2 by (auto simp: N1-def N2-def)

  have N1 ∩ N2 = {}
  proof (safe;simp)
    fix u
    assume u∈N1 u∈N2
    hence (x,u)∈E′* (u,y)∈E′* by (auto simp: N1-def N2-def symD[OF SYME])
    with cycle-free-altD[OF ‹cycle-free T› ‹(x,y)∈edges T›] show False
      unfolding E′-def by (meson rtrancl-trans)
  qed


  have N1C: E′``N1 ⊆ N1
    unfolding N1-def
    apply clarsimp
    by (simp add: rtrancl.rtrancl-into-rtrancl)
```

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

have *XE1*: $(x,u) \in (edges\ T1)^*$ **if** $u{\in}N1$ **for** *u*
proof −
  from *that* **have** $(x,u){\in}E'^*$ **by** (*auto simp*: *N1-def*)
  then show *?thesis* **using** ‹$x{\in}N1$›
    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.prems* **have** $(y,z){\in}Restr\ (edges\ T)\ N1$
      unfolding *E′-def* **by** *auto*
    with *step.IH*[*OF* ‹$z{\in}N1$›] **show** *?case*
      by (*metis converse-rtrancl-into-rtrancl edges-restrict-nodes*)
  qed *auto*
qed

have *XE2*: $(y,u) \in (edges\ T2)^*$ **if** $u{\in}N2$ **for** *u*
proof −
  from *that* **have** $(y,u){\in}E'^*$ **by** (*auto simp*: *N2-def*)
  then show *?thesis* **using** ‹$y{\in}N2$›
    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.prems* **have** $(y,z){\in}Restr\ (edges\ T)\ N2$
      unfolding *E′-def* **by** *auto*
    with *step.IH*[*OF* ‹$z{\in}N2$›] **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∈N1 ∪ N2 if u∈nodes T for u
proof -
  from connectedD[OF ‹connected T› ‹x∈nodes T› 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)∉set p ∧ (y,x)∉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 ¬((x,y)∉set p ∧ (y,x)∉set p)
    with P obtain p′ where
      uedge (x,y)∉set (map uedge p′) path T y p′ u ∨ path T x p′ u
      by (auto simp: in-set-conv-decomp uedge-commute)
    hence path (restrict-edges T E′) y p′ u ∨ 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 ∪ 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 ∪ edges T2 ⊆ E′
  unfolding T1-def T2-def E′-def using ‹N1 ∩ N2 = {}› ‹x ∈ N1› ‹y ∈ N2›
  by auto
also have edges T1 ∪ edges T2 ⊇ E′
proof -
  note ED1 = nodesI[where g=T, unfolded ‹nodes T = N1∪N2›]
  have E′ ⊆ edges T by (auto simp: E′-def)
  thus edges T1 ∪ edges T2 ⊇ E′
    unfolding T1-def T2-def
    using ED1 N1C N2C by (auto; blast)
qed
finally have edges T1 ∪ 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
  ≡ tree T ∧ nodes T = nodes G ∧ edges T ⊆ 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 ⟷ v=w ∧ 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∈nodes T v∉nodes T
  shows tree (ins-edge (u,v) T)
proof –
  from assms have [simp]: u≠v by auto
  have ins-edge (u,v) T = 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 ⟹ ∃ t. 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 ‹connected g› show ?thesis by (auto simp: is-spanning-tree-def tree-def)
  next
    case False
    then obtain u v where
        EDGE: (u,v)∈edges g
```

and *RED*: $(u,v) \in (edges\ g - \{(u,v),(v,u)\})^*$
using *cycle-free-altI* **by** *metis*
from ‹*connected g*›
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′* ⊂ *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* ⇒ *nat*) ⇒ ′*v ugraph* ⇒ *nat*
where *weight w g* ≡ ($\sum$ *e*∈*edges g*. *w* (*uedge e*)) *div 2*

lemma *weight-alt*: *weight w g* = ($\sum$ *e*∈*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*∈*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** ‹$E \cap E^{-1}$=-›
by (*auto simp*: *uedge-def inj-on-def doubleton-eq-iff*)

have *weight w g* = ($\sum$ *e*∈*E*. *w* (*uedge e*))
unfolding *weight-def* ‹*edges g* = -› **using** ‹$E \cap E^{-1} = \{\}$›
by (*auto simp*: *sum.union-disjoint*)
also have ... = ($\sum$ *e*∈*uedge*‘*E*. *w e*)
using *sum.reindex*[*of uedge E w*]
by *auto*
also have *uedge*‘*E* = *uedge*‘(*edges g*)
unfolding ‹*edges g* = -› *uedge-def* **using** ‹$E \cap E^{-1} = \{\}$›
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*]: $[\![u{\neq}v;\ (u,v){\notin}edges\ g]\!]$
$\Longrightarrow$ *weight w* (*ins-edge* (*u*,*v*) *g*) = *w* {*u*,*v*} + *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*]:
uedge‘*edges* g_1 ∩ *uedge*‘*edges* g_2 = {} ⟷ *edges* g_1 ∩ *edges* g_2 = {}
by (*auto simp*: *uedge-eq-iff dest*: *edges-sym′*)+

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

lemma *weight-cong*: *edges* g_1 = *edges* g_2 $\Longrightarrow$ *weight w* g_1 = *weight w* g_2
by (*auto simp*: *weight-def*)

lemma *weight-mono*: *edges g* ⊆ *edges g′* $\Longrightarrow$ *weight w g* ≤ *weight w g′*
unfolding *weight-alt* **by** (*rule sum-mono2*) *auto*

lemma *weight-ge-edge*:
assumes (*x*,*y*)∈*edges T*
shows *weight w T* ≥ *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*)∈*edges T*
shows *weight w* (*restrict-edges T* (− {(*x*, *y*), (*y*, *x*)})) = *weight w T* − *w* {*x*,*y*}
proof −
define *E* **where** *E* = *uedge* ‘ *edges T* − {{*x*,*y*}}
have [*simp*]: (*uedge* ‘ (*edges T* − {(*x*, *y*), (*y*, *x*)})) = *E*
by (*safe*; *simp add*: *E-def uedge-def doubleton-eq-iff*; *blast*)

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

have [*simp*]: {*x*,*y*}∉*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* ≡ *is-spanning-tree g t*
  ∧ (∀ *t'*. *is-spanning-tree g t'* ⟶ *weight w t* ≤ *weight w t'*)

lemma *exists-MST*: *connected g* ⟹ ∃ *t*. *is-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 *αw* :: *'g* ⇒ *'v set* ⇒ *nat* **and** *αg* :: *'g* ⇒ *'v ugraph*
    and *invar* :: *'g* ⇒ *bool*
    and *adj* :: *'g* ⇒ *'v* ⇒ (*'v*×*nat*) *list*
    and *empty* :: *'g*
    and *add-edge* :: *'v*×*'v* ⇒ *nat* ⇒ *'g* ⇒ *'g*
  assumes *adj-correct*: *invar g*
    ⟹ *set* (*adj g u*) = {(*v*,*d*). (*u*,*v*)∈*edges* (*αg g*) ∧ *αw g* {*u*,*v*} = *d*}
  assumes *empty-correct*:
    invar empty
    αg empty = *graph-empty*
    αw empty = (λ-. *0*)
  assumes *add-edge-correct*:
    ⟦*invar g*; (*u*,*v*)∉*edges* (*αg g*); *u*≠*v*⟧ ⟹ *invar* (*add-edge* (*u*,*v*) *d g*)
    ⟦*invar g*; (*u*,*v*)∉*edges* (*αg g*); *u*≠*v*⟧
      ⟹ *αg* (*add-edge* (*u*,*v*) *d g*) = *ins-edge* (*u*,*v*) (*αg g*)
    ⟦*invar g*; (*u*,*v*)∉*edges* (*αg g*); *u*≠*v*⟧
      ⟹ *αw* (*add-edge* (*u*,*v*) *d g*) = (*αw g*)({*u*,*v*}:=*d*)

begin

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

lemma *empty-spec-presentation*:
  invar empty ∧ *αg empty* = *graph* {} {} ∧ *αw empty* = (λ-. *0*)
  by (*auto simp*: *wgraph-specs*)

lemma *add-edge-spec-presentation*:
  ⟦*invar g*; (*u*,*v*)∉*edges* (*αg g*); *u*≠*v*⟧ ⟹

invar (*add-edge* (*u*,*v*) *d* *g*)
∧ *αg* (*add-edge* (*u*,*v*) *d* *g*) = *ins-edge* (*u*,*v*) (*αg* *g*)
∧ *αw* (*add-edge* (*u*,*v*) *d* *g*) = (*αw* *g*)({*u*,*v*}:=*d*)
by (*auto simp*: *wgraph-specs*)

end

1.3.2 Generic From-List Algorithm

definition *valid-graph-repr* :: ('*v*×'*v*) *list* ⇒ *bool*
where *valid-graph-repr l* ⟷ (∀ (*u*,*v*)∈*set l*. *u*≠*v*)

definition *graph-from-list* :: ('*v*×'*v*) *list* ⇒ '*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* ∪ *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* ∪ (*set l*)$^{-1}$
using *valid* **apply** (*induction l*)
unfolding *graph-from-list-def valid-graph-repr-def*
by *auto*

definition *valid-weight-repr l* ≡ *distinct* (*map* (*uedge o fst*) *l*)

definition *weight-from-list* :: (('*v*×'*v*)×*nat*) *list* ⇒ '*v set* ⇒ *nat* **where**
weight-from-list l ≡ *foldr* (λ((*u*,*v*),*d*) *w*. *w*({*u*,*v*}:=*d*)) *l* (λ-. *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 [] = (λ-. *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) ⟷ 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)
    ⟷ 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
  ⟷ valid-graph-repr (map fst l) ∧ valid-weight-repr l

definition from-list l = foldr (λ(e,d). add-edge e d) l empty


lemma from-list-refine: valid-wgraph-repr l ⟹
    invar (from-list l)
  ∧ αg (from-list l) = graph-from-list (map fst l)
  ∧ αw (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 (αg (from-list l)) = fst`fst`set l ∪ snd`fst`set l
    edges (αg (from-list l)) = (fst`set l) ∪ (fst`set l)⁻¹
    ((u,v),d)∈set l ⟹ αw (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 ⟷
  (∀((u,v),d)∈set l. u≠v) ∧ distinct [ {u,v}. ((u,v),d)←l ]
proof –
  have [simp]: uedge ∘ fst = (λ((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=αg gi; w=αw gi in
      invar gi
    ∧ nodes g = ⋃{{u,v} | u v. ∃d. ((u,v),d)∈set l}
    ∧ edges g = ⋃{{(u,v),(v,u)} | u v. ∃d. ((u,v),d)∈set l}
    ∧ (∀((u,v),d)∈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* ≡ ∃ *t*. *is-MST w g t* ∧ *A* ⊆ *edges t*

lemma *is-subset-MST-empty*[*simp*]: *connected g* ⟹ *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 set* ⇒ *nat* **and** *g* :: ′*v ugraph* **and** *r* :: ′*v*
begin

Reachable part of the graph

definition *rg* ≡ *component-of g r*

lemma *reachable-connected*[*simp*, *intro*!]: *connected rg*
  unfolding *rg-def* **by** *auto*

lemma *reachable-edges-subset*: *edges rg* ⊆ *edges g*
  unfolding *rg-def* **by** (*rule component-edges-subset*)

definition *light-edge C u v*
  ≡ *u*∈*C* ∧ *v*∉*C* ∧ (*u*,*v*)∈*edges rg*
    ∧ (∀ (*u*′,*v*′)∈*edges rg* ∩ *C*×−*C*. *w* {*u*,*v*} ≤ *w* {*u*′,*v*′})

definition *respects-cut A C* ≡ *A* ⊆ *C*×*C* ∪ (−*C*)×(−*C*)

lemma *light-edge-is-safe*:
  fixes *A* :: (′*v*×′*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*)} ∪ *A*)
proof −
  have *crossing-edge*: *u*∈*C* *v*∉*C* (*u*,*v*)∈*edges rg*
    and *min-edge*: ∀ (*u*′,*v*′)∈*edges rg* ∩ *C*×−*C*. *w* {*u*,*v*} ≤ *w* {*u*′,*v*′}
    using *light-edge* **unfolding** *light-edge-def* **by** *auto*

  from *subset-MST* **obtain** *T* **where** *T*: *is-MST w rg T A* ⊆ *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*) ∈ *edges T*
thus *?thesis* **unfolding** *is-subset-MST-def* **using** *T* **by** (*auto simp*: *edges-sym′*)
next
assume (*u*,*v*) ∉ *edges T* **hence** (*v*,*u*)∉*edges T* **by** (*auto simp*: *edges-sym′*)
from ‹(*u*,*v*)∈*edges rg*› **obtain** *p* **where** *p*: *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*≠*v* **using** *crossing-edge* **by** *blast*

from *find-crossing-edge-on-path*[*OF p*(*1*), **where** *P*=λ*x*. *x*∉*C*]
crossing-edge(*1*,*2*)
obtain *x y p1 p2* **where** *xy*: (*x*,*y*) ∈ *set p x* ∈ *C y* ∉ *C*
and *ux*: *path* (*restrict-edges T* (−{(*x*,*y*),(*y*,*x*)})) *u p1 x*
and *yv*: *path* (*restrict-edges T* (−{(*x*,*y*),(*y*,*x*)})) *y p2 v*
using *path-change*[*OF crossing-edge*(*1*,*2*) *p*] **by** *blast*
have (*x*,*y*) ∈ *edges T*
by (*meson contra-subsetD p*(*1*) *path-edges xy*(*1*))

let *?E′* = *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 *nodes T1* ∩ *nodes T2* = {}
and *nodes T* = *nodes T1* ∪ *nodes T2*
and *edges T1* ∪ *edges T2* = *?E′*
and *nodes T1* = { *u* . (*x*,*u*)∈*?E′**}
and *nodes T2* = { *u* . (*y*,*u*)∈*?E′**}
and *x*∈*nodes T1 y*∈*nodes T2* **.**

let *?T′* = *ins-edge* (*u*,*v*) (*graph-join T1 T2*)

have *is-spanning-tree rg ?T′* **proof** −

have *E′-sym*: *sym* (*?E′**)
by (*meson edgesT-diff-sng-inv-eq sym-conv-converse-eq sym-rtrancl*)

have *u*∈*nodes T1*
unfolding ‹*nodes T1* = -›
using *path-rtrancl-edgesD*[*OF ux*] **by** (*auto dest*: *symD*[*OF E′-sym*])

have *v*∈*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*: $(u,v) \in A \Longrightarrow (v,r) \in A^*$
  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*: $[\![u \in S\ A;\ (u,v) \in$ *edges* $g\ ]\!] \Longrightarrow (u,v) \in$ *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*: $T\text{-}measure1\ A < card\ (nodes\ rg)$
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*) $(A \cup A^{-1})$
unfolding *nodes-edges-consistent-def*
apply (*intro conjI*)
subgoal by *simp*
subgoal using *A-edges irrefl-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)} ∪ A)
      by (rule light-edge-is-safe[OF subset-MST respects-cut light-edge])

  next
    show ∀ (ua, va)∈{(v, u)} ∪ A. (va, r) ∈ ({(v, u)} ∪ 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)}∪A by unfold-locales

  have S A ⊂ S ({(v,u)}∪A) using ‹v∉S A›
    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 ⊆ edges t
    unfolding is-subset-MST-def by auto

  have S A = nodes t
  proof safe
    fix u
    show u∈S A ⟹ u∈nodes t using MST
      unfolding is-MST-def is-spanning-tree-def
      using S-reachable by auto
  next
    fix u
    assume u∈nodes t
    hence u∈nodes rg
      using MST is-MST-def is-spanning-tree-def by force
    hence 1: (u,r)∈(edges rg)* by (simp add: connectedD rg-def)
    have r∈S A by (simp add: S-def)
    show u∈S A proof (rule ccontr)
      assume u∉S A
```

```
      from find-crossing-edge-rtrancl[where P=λu. u∈S A, OF 1 ‹u∉S A› ‹r∈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 = nodes rg
    using MST unfolding is-MST-def is-spanning-tree-def
    by auto
  finally have S-eq: S A = nodes rg .

  define t' where t' = graph {r} A

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

  hence edges t' ⊆ edges t
    by (smt UnE ‹A ⊆ edges t› 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 ‹edges t' ⊆ edges t› cycle-free-antimono is-MST-def
              is-spanning-tree-def tree-def)
    moreover have edges t' ⊆ edges rg
      by (meson MST ‹edges t' ⊆ edges t› 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* ‹*edges* $t' \subseteq$ *edges* t›]
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 set* ⇒ *nat* **and** *g* :: *'v ugraph* **and** *r* :: *'v*
begin

Abstraction to edge set

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

Initialization

definition $\mathit{initQ} :: {}'v \Rightarrow \mathit{enat}$ **where** $\mathit{initQ} \equiv (\lambda\text{-}.\ \infty)(r := 0)$
definition $\mathit{init}\pi :: {}'v \Rightarrow {}'v\ \mathit{option}$ **where** $\mathit{init}\pi \equiv \mathit{Map.empty}$

Step

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

State after inner loop

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

State after one step

definition $Q'\ Q\ \pi\ u \equiv (\mathit{Qinter}\ Q\ \pi\ u)(u{:=}\infty)$
definition $\pi'\ Q\ \pi\ u\ v' = (\mathit{if}\ \mathit{upd\text{-}cond}\ Q\ \pi\ u\ v'\ \mathit{then}\ \mathit{Some}\ u\ \mathit{else}\ \pi\ v')$

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

definition $\mathit{prim\text{-}invar2\text{-}ctd}\ Q\ \pi \equiv \mathit{let}\ A = A\ Q\ \pi;\ S = S\ A\ \mathit{in}$
$\mathit{prim\text{-}invar1}\ A$
$\wedge\ \pi\ r = \mathit{None} \wedge Q\ r = \infty$
$\wedge\ (\forall (u,v){\in}\mathit{edges}\ rg \cap (-S){\times}S.\ Q\ u \neq \infty)$
$\wedge\ (\forall u.\ Q\ u \neq \infty \longrightarrow \pi\ u \neq \mathit{None})$
$\wedge\ (\forall u\ v.\ \pi\ u = \mathit{Some}\ v \longrightarrow v{\in}S \wedge (u,v){\in}\mathit{edges}\ rg)$
$\wedge\ (\forall u\ v\ d.\ Q\ u = \mathit{enat}\ d \wedge \pi\ u = \mathit{Some}\ v$
$\longrightarrow d{=}w\ \{u,v\} \wedge (\forall v'{\in}S.\ (u,v'){\in}\mathit{edges}\ rg \longrightarrow d \leq w\ \{u,v'\}))$

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 −
  interpret *Prim-Invar1-loc w g r A Q* π **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* π]
  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*π $r = (\lambda u.$ *if* $(u,r){\in}edges\ rg$ *then enat* $(w\ \{u,r\})$ *else* $\infty)$
  apply (*rule ext*)
  using *reachable-edges-subset*
  apply (*simp add*: *Q′-def Qinter-def upd-cond-def initQ-def init*π*-def*)
  by (*auto simp*: *Prim.rg-def edges-sym′ reachable-nodes-step′*(*2*))

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

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

lemma *A-init-eq*: *A initQ initπ* = {}
  unfolding *A-def initπ-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 π*
  assumes *UNS*: *Q u* = *enat d*
  shows *prim-invar2-ctd* (*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*)
proof −
  from *INV* **have** [*simp*]: *Q*=*initQ π*=*initπ*
    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 π′-init-eq A-init-eq*

  have [*simp*]: (*A* (*Q′ initQ initπ r*) (*π′ initQ initπ r*)) = {}
    apply (*simp add*: *Q′-init-eq π′-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 π′-init-eq S-empty split*: *if-splits*)

  have [*simp*]: *Q′ initQ initπ r r* = ∞
    by (*auto simp*: *Q′-init-eq*)

  have [*simp*]: *initQ 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 π*
  assumes *UNS*: *Q u* = *enat d*
  shows *prim-invar2-ctd* (*Q′ Q π u*) (*π′ Q π u*)
      ∧ *T-measure2* (*Q′ Q π u*) (*π′ Q π u*) < *T-measure2 Q π*
  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 π
  assumes invar2: prim-invar2-ctd Q π
begin

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

lemma upd-cond-alt: upd-cond Q π u v' ⟷
  (v',u) ∈ edges g ∧ v'∉S (A Q π) ∧ enat (w {v',u}) < Q v'
  unfolding upd-cond-def S-alt-def unfolding A-def
  by (auto simp: fst-eq-Domain)

lemma π-root: π r = None
  and Q-root: Q r = ∞
  and Q-defined: ⟦ (u,v)∈edges rg; u∉S (A Q π); v∈S (A Q π) ⟧ ⟹ Q u ≠ ∞
  and π-defined: ⟦ Q u ≠ ∞ ⟧ ⟹ π u ≠ None
  and frontier: π u = Some v ⟹ v∈S (A Q π)
  and edges: π u = Some v ⟹ (u,v)∈edges rg
  and Q-π-consistent: ⟦ Q u = enat d; π u = Some v ⟧ ⟹ d = w {u,v}
  and Q-min: Q u = enat d
      ⟹ (∀ v'∈S (A Q π). (u,v')∈edges rg ⟶ d ≤ w {u,v'})
  using invar2 unfolding prim-invar2-ctd-def Let-def by auto

lemma π-def-on-S: ⟦u∈S (A Q π); u≠r⟧ ⟹ π u ≠ None
  unfolding S-alt-def
  unfolding A-def
  by auto

lemma π-def-on-edges-to-S: ⟦v∈S (A Q π); u≠r; (u,v)∈edges rg⟧ ⟹ π u ≠ None
  apply (cases u∈S (A Q π))
  subgoal using π-def-on-S by auto
  subgoal by (simp add: Q-defined π-defined)
  done

lemma Q-min-is-light:
  assumes UNS: Q u = enat d
  assumes MIN: ∀ v. enat d ≤ Q v
  obtains v where π u = Some v light-edge (S (A Q π)) v u
proof –
  let ?A = A Q π
  let ?S = S ?A
```

```
  from UNS obtain v where
    S1[simp]: π u = Some v d = w {u,v}
    using π-defined Q-π-consistent
    by blast

  have v∈?S using frontier[of u v] by auto

  have [simp]: u≠r using π-root using S1 by (auto simp del: S1)

  have u∉?S unfolding S-alt-def unfolding A-def using UNS by auto

  have (v,u)∈edges rg using edges[OF S1(1)]
    by (meson edges-sym' rev-subsetD)

  have M: ∀(u', v')∈edges rg ∩ ?S × − ?S. w {v, u} ≤ w {u', v'}
  proof safe
    fix a b
    assume (a,b)∈edges rg a∈?S b∉?S
    hence (b,a)∈edges rg by (simp add: edges-sym')

    from Q-defined[OF ‹(b,a)∈edges rg› ‹b∉?S› ‹a∈?S›]
      obtain d' where 1: Q b = enat d' by blast
    with π-defined obtain a' where π b = Some a' by auto
    from MIN 1 have d≤d' by (metis enat-ord-simps(1))
    also from Q-min[OF 1] ‹(b,a)∈edges rg› ‹a∈?S› have d'≤w {b,a} by blast
    finally show w {v,u} ≤ w {a,b} by (simp add: insert-commute)
  qed

  have LE: light-edge ?S v u using invar1 ‹v∈?S› ‹u∉?S› ‹(v,u)∈edges rg› M
    unfolding light-edge-def by blast

  thus ?thesis using that by auto
qed

lemma maintain-invar-ctd:
  assumes UNS: Q u = enat d
  assumes MIN: ∀v. enat d ≤ Q v
  shows prim-invar2-ctd (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)
proof −
  let ?A = A Q π
  let ?S = S ?A

  from Q-min-is-light[OF UNS MIN] obtain v where
    [simp]: π u = Some v and LE: light-edge ?S v u .

  let ?Q' = Q' Q π u
  let ?π' = π' Q π 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 π-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 π'-def π-defined enat.distinct(2)
          fun-upd-apply not-None-eq)

  subgoal
    by (metis ‹S (A (Q' Q π u) (π' Q π u)) = insert u (S (A Q π))› π'-def
          frontier insertCI option.inject)
  subgoal
    by (metis N.S-edge-reachable upd-cond-def
        ‹S (A (Q' Q π u) (π' Q π u)) = insert u (S (A Q π))› π'-def edges
        edges-sym' insertI1 option.inject)
  subgoal
    by (smt Q'-def π'-def Q-π-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 ‹(auto;fail)?›)
    subgoal by (simp add: le-less less-le-trans)
    subgoal using π-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' ?π' 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* π
  assumes *UNS*: *Q u* = *enat d*
  assumes *MIN*: $\forall$ *v*. *enat d* $\leq$ *Q v*
  shows *prim-invar2-ctd* (*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*)
proof −
  interpret *Prim-Invar2-ctd-loc w g r Q* π **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* π
  assumes *UNS*: *Q u* = *enat d*
  assumes *MIN*: $\forall$ *v*. *enat d* $\leq$ *Q v*
  obtains *v* **where** π *u* = *Some v light-edge* (*S* (*A Q* π)) *v u*
proof −
  interpret *Prim-Invar2-ctd-loc w g r Q* π **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* π
  assumes *UNS*: *Q u* = *enat d*
  assumes *MIN*: $\forall$ *v*. *enat d* $\leq$ *Q v*
  shows *prim-invar2-ctd* (*Q'* *Q* π *u*) (π' *Q* π *u*)
      $\wedge$ *T-measure2* (*Q'* *Q* π *u*) (π' *Q* π *u*) < *T-measure2 Q* π
  using *maintain-invar2-ctd assms* **by** *blast*

lemma *not-invar2-ctd-init*:
  prim-invar2-init Q π $\Longrightarrow$ $\neg$*prim-invar2-ctd Q* π
  unfolding *prim-invar2-init-def prim-invar2-ctd-def initQ-def Let-def*
  by (*auto*)

lemma *invar2-init-init*: *prim-invar2-init initQ init*π
  unfolding *prim-invar2-init-def* **by** *auto*

lemma *invar2-init*: *prim-invar2 initQ init*π
  unfolding *prim-invar2-def* **using** *invar2-init-init* **by** *auto*

lemma *maintain-invar2*:
  assumes *A*: *prim-invar2 Q* π
  assumes *UNS*: *Q u* = *enat d*
  assumes *MIN*: $\forall$ *v*. *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*: $Q = (\lambda\text{-}.\ \infty)$
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*: $Q = (\lambda\text{-}.\ \infty)$
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,π)
  else
    case $(Q\ v,\ \pi\ v)$ *of*
      $(\infty,None) \Rightarrow (Q(v:=enat\ d),\ \pi(v\mapsto u))$
    $|\ (enat\ d',\text{-}) \Rightarrow$ *if* $d<d'$ *then* $(Q(v:=enat\ d),\ \pi(v\mapsto u))$ *else* (Q,π)
    $|\ (\infty,Some\ \text{-}) \Rightarrow (Q,\pi)$

lemma *foreach-body-alt*: *foreach-body* $u = (\lambda(v,d)\ (Q,\pi).$
  if $v\neq r \wedge (\pi\ v = None \vee Q\ v \neq \infty) \wedge enat\ d < Q\ v$ *then*
    $(Q(v:=enat\ d),\ \pi(v\mapsto u))$
  else
    (Q,π)
)
  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 $Q\pi$ = *foldr* (*foreach-body u*) *adjs* $Q\pi$

definition $\bigwedge Q\ V.$
  Qigen $Q\ \pi\ u$ *adjs* v = (*if* $v \notin$ *fst‘set adjs then* $Q\ v$ *else Qinter* $Q\ \pi\ u\ v$)
definition $\bigwedge Q\ V\ \pi.$
  $\pi'gen\ Q\ \pi\ u$ *adjs* v = (*if* $v \notin$ *fst‘set adjs then* $\pi\ v$ *else* $\pi'\ Q\ \pi\ u\ v$)

context begin

private lemma *Qc*:
  Qigen $Q\ \pi\ u\ ((v,\ w\ \{u,\ v\})\ \#\ adjs)\ x$
  = (*if* $x=v$ *then Qinter* $Q\ \pi\ u\ v$ *else Qigen* $Q\ \pi\ u$ *adjs* x) **for** x
  unfolding *Qigen-def* **by** *auto*

private lemma πc:
  $\pi'gen\ Q\ \pi\ u\ ((v,\ w\ \{u,\ v\})\ \#\ adjs)\ x$
  = (*if* $x=v$ *then* $\pi'\ Q\ \pi\ u\ v$ *else* $\pi'gen\ Q\ \pi\ u$ *adjs* x) **for** x
  unfolding $\pi'gen\text{-}def$ **by** *auto*

lemma *foreach-refine-gen*:
  assumes *set adjs* $\subseteq \{(v,d).\ (u,v)\in edges\ g \wedge w\ \{u,v\} = d\}$
  shows *foreach u adjs* (Q,π) = (*Qigen* $Q\ \pi\ u$ *adjs*,$\pi'gen\ Q\ \pi\ u$ *adjs*)
  using *assms*
  unfolding *foreach-def*
proof (*induction adjs arbitrary*: $Q\ \pi$)
  case *Nil*
  have *INVAR-INIT*: *Qigen* $Q\ \pi\ u\ [] = Q\ \pi'gen\ Q\ \pi\ u\ [] = \pi$ **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}
    ⟷ (u,v)∈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*: $epair\ e = Some\ (x,y) \Longrightarrow (x\neq y \wedge e=\{x,y\})$
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 $M\text{-}lookup :: 'm \Rightarrow 'v \Rightarrow (('v\times nat)\ list)\ option$
and *M-invar*
begin

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

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

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

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

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

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

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

lemma *αedges-irrefl*: $invar\ g \Longrightarrow irrefl\ (\alpha edges\text{-}aux\ g)$
  unfolding *invar-def irrefl-def αedges-aux-def*
  by (*force*)

lemma *αedges-sym*: $invar\ g \Longrightarrow sym\ (\alpha edges\text{-}aux\ g)$
  unfolding *invar-def sym-def αedges-aux-def*
  by *force*

lemma *αedges-subset*: $invar\ g \Longrightarrow \alpha edges\text{-}aux\ g \subseteq \alpha nodes\text{-}aux\ g \times \alpha nodes\text{-}aux\ g$
  unfolding *invar-def αnodes-aux-def αedges-aux-def*
  by *force*

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

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

definition *adj* :: $'m \Rightarrow 'v \Rightarrow ('v\times nat)\ list$ **where**
  $adj\ g\ v = the\text{-}default\ []\ (M\text{-}lookup\ g\ v)$

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

definition *add-edge1* :: $'v \times 'v \Rightarrow nat \Rightarrow 'm \Rightarrow 'm$ **where**
add-edge1 $\equiv \lambda(u,v)\ d\ g.\ M\text{-}update\ u\ ((v,d)\ \#\ the\text{-}default\ []\ (M\text{-}lookup\ g\ u))\ g$

definition *add-edge* :: $'v \times 'v \Rightarrow nat \Rightarrow 'm \Rightarrow 'm$ **where**
add-edge $\equiv \lambda(u,v)\ d\ g.\ add\text{-}edge1\ (v,u)\ d\ (add\text{-}edge1\ (u,v)\ d\ g)$

lemma *edges-αg-aux*: $invar\ g \Longrightarrow edges\ (\alpha g\ g) = \alpha edges\text{-}aux\ g$
unfolding *αg-def* **using** *αedges-sym αedges-irrefl*
by (*auto simp*: *irrefl-def graph-accs*)

lemma *nodes-αg-aux*: $invar\ g \Longrightarrow nodes\ (\alpha g\ g) = \alpha nodes\text{-}aux\ g$
unfolding *αg-def* **using** *αedges-subset*
by (*force simp*: *graph-accs*)

lemma *card-doubleton-eq2*[*simp*]: $card\ \{a,b\} = 2 \longleftrightarrow a \neq b$ **by** *auto*

lemma *the-dflt-Z-eq*: $the\text{-}default\ 0\ m = d \longleftrightarrow (m=None \wedge d=0 \vee m=Some\ d)$
by (*cases m*) *auto*

lemma *adj-correct-aux*:
$invar\ g \Longrightarrow set\ (adj\ g\ u) = \{(v,\ d).\ (u,\ v) \in edges\ (\alpha g\ g) \wedge \alpha w\ g\ \{u,\ v\} = d\}$
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* = *Some xs* $\longrightarrow$ *distinct* (*map fst xs*))
  by (*cases m*; *auto*)

lemma *invar-add-edge-aux*:
  $[\![$*invar g*; (*u*, *v*) $\notin$ *edges* (α*g g*); *u* $\neq$ *v*$]\!]$ $\Longrightarrow$ *invar* (*add-edge* (*u*, *v*) *d g*)
  apply (*simp add*: *edges-*α*g-aux*)
  unfolding *add-edge-def add-edge1-def invar-def* α*edges-aux-def*
  by (*auto simp*: *M.map-specs dist-fst-the-dflt-aux*; *force*)

sublocale *adt-wgraph* α*w* α*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-*α*g-aux invar-empty-aux edges-*α*g-aux*
        add: α*nodes-aux-def* α*edges-aux-def*
)
    apply (*simp add*: *empty-def M.map-specs*)
    done
  subgoal
    unfolding α*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-*α*g-aux nodes-*α*g-aux graph-eq-iff invar-add-edge-aux*)
    apply (*rule conjI*)
    subgoal
      unfolding *add-edge-def add-edge1-def invar-def* α*nodes-aux-def*
      by (*auto simp*: *M.map-specs*)
    subgoal
      unfolding *add-edge-def add-edge1-def invar-def* α*edges-aux-def*
      by (*fastforce simp*: *M.map-specs split*!: *if-splits*)
    done
  subgoal for *g u v d*
    apply (*simp add*: *edges-*α*g-aux invar-add-edge-aux*)
    unfolding *invar-def* α*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-*α*w G-*α*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-*α*w* **and** *G-*α*g* :: *'g* ⇒ (*'v*) *ugraph* **and** *G-invar G-adj G-empty G-add-edge*

and *M-empty M-update M-delete* **and** *M-lookup* :: *'m* ⇒ *'v* ⇒ *'v 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

locale *Prim-Impl-Defs* = *Prim-Impl-Adts*
where *typG* = *typG* **and** *typM* = *typM* **and** *typQ* = *typQ* **and** *G-*α*w* = *G-*α*w* **and** *G-*α*g* = *G-*α*g*
for *typG* :: *'g itself* **and** *typM* :: *'m itself* **and** *typQ* :: *'q itself*
and *G-*α*w* **and** *G-*α*g* :: *'g* ⇒ (*'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* πi *v*) *of*
(*None*,*None*) $\Rightarrow$ (*Q-update v d Qi*, *M-update v u* πi)
| (*Some* d',-) $\Rightarrow$ (*if* $d<d'$ *then* (*Q-update v d Qi*, *M-update v u* πi) *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 πi *u adjs* = *foldr* (*foreach-impl-body u*) *adjs* $(Qi,\pi i)$

definition *outer-loop-impl Qi* πi $\equiv$ *while* $(\lambda(Qi,\pi i).\ \neg$*Q-is-empty Qi*$)$ $(\lambda(Qi,\pi i).$
let
(*u*,-) = *Q-getmin Qi*;
adjs = *G-adj g u*;
$(Qi,\pi i)$ = *foreach-impl Qi* πi *u adjs*;
Qi = *Q-delete u Qi*
in $(Qi,\pi i))$ $(Qi,\pi i)$

definition *prim-impl* = (*let*
Qi = *Q-update r 0 Q-empty*;
πi = *M-empty*;
$(Qi,\pi i)$ = *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 $(\lambda(Q, \pi).\ \neg$ *Q-is-empty Q*$)$ $(\lambda(Q, \pi).$ *let*
(*u*, -) = *Q-getmin Q*;
— Inner loop: Update for adjacent nodes
(Q, π) =
foldr $((\lambda(v,\ d)\ (Q,\ \pi).$ *let*
qv = *Q-lookup Q v*;
πv = *M-lookup* π *v*
in
if $v{\neq}r \wedge (qv{\neq}None \vee \pi v{=}None) \wedge$ *enat d* $<$ *enat-of-option qv*
then (*Q-update v d Q*, *M-update v u* π)
else $(Q, \pi))$
) (*G-adj g u*) (Q, π);
Q = *Q-delete u Q*

in $(Q, \pi))\ (Q, \pi)$
in π
)
proof −

have *1*: *foreach-impl-body u* = $(\lambda(v,d)\ (Qi,\pi i).$ *let*
qiv = (*Q-lookup Qi v*);
πiv = *M-lookup* πi *v*
in
if $v \neq r \wedge (qiv \neq None \vee \pi iv = None) \wedge$ *enat d* < *enat-of-option qiv*
then (*Q-update v d Qi*, *M-update v u* πi)
else $(Qi, \pi i))$ **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 πi and set *Vi* satisfy their implementation invariants
- The πi encodes irreflexive edges consistent with the nodes determined by *Vi*. Note that the edges in πi will not be symmetric, thus we take their symmetric closure $E \cup E^{-1}$.

definition *invar-MST* $\pi i \equiv$ *M-invar* πi

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

end

1.6.2 Refinement of State

locale *Prim-Impl* = *Prim-Impl-Defs*
where *typG* = *typG* **and** *typM* = *typM* **and** *typQ* = *typQ* **and** *G-*α*w* = *G-*α*w*
and *G-*α*g* = *G-*α*g*
for *typG* :: *'g itself* **and** *typM* :: *'m itself* **and** *typQ* :: *'q itself*
and *G-*α*w* **and** *G-*α*g* :: *'g* ⇒ (*'v*::*linorder*) *ugraph*
+
assumes *G-invar*[*simp*]: *G-invar g*
begin

sublocale *Prim2 G-*α*w g G-*α*g g r* .

Abstraction of Q

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

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

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

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

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

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** $(auto\ simp\colon fun\text{-}eq\text{-}iff)$

lemma $Q\text{-}\alpha\text{-}min$:
  assumes MIN: $Q\text{-}getmin\ Qi = (u,\ d)$
  assumes I: $Q\text{-}invar\ Qi$
  assumes NE: $\neg\ Q\text{-}is\text{-}empty\ Qi$
  shows $Q\text{-}\alpha\ Qi\ u = enat\ d$ (**is** $?G1$) **and**
    $\forall\, v.\ enat\ d \le Q\text{-}\alpha\ Qi\ v$ (**is** $?G2$)
proof $-$
  from $Q.map\text{-}getmin[OF\ MIN]$
    have $Q\text{-}lookup\ Qi\ u = Some\ d\ (\forall\, x{\in}ran\ (Q\text{-}lookup\ Qi).\ d \le x)$
    using $NE\ I$ **by** $auto$
  thus $?G1\ ?G2$
    unfolding $Q\text{-}\alpha\text{-}def$ **apply** $simp\text{-}all$
    by $(metis\ enat\text{-}of\text{-}option.elims\ enat\text{-}ord\text{-}simps(1)\ enat\text{-}ord\text{-}simps(3)\ ranI)$
qed

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

Concrete Invariant

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

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

end

1.6.3 Refinement of Algorithm

context *Prim-Impl*
begin

lemma *foreach-impl-correct*:
  fixes *Qi Vi* πi **defines** $Q \equiv$ *Q-*α *Qi* **and** $\pi \equiv$ *M-lookup* πi
  assumes *A*: *foreach-impl Qi* πi *u* (*G-adj g u*) = (*Qi'*,$\pi i'$)
  assumes *I*: *prim-invar-impl Qi* πi
  shows *Q-invar Qi'* **and** *M-invar* $\pi i'$
    and *Q-*α *Qi'* = *Qinter Q* π *u* **and** *M-lookup* $\pi i'$ = π' *Q* π *u*
proof −
  from *I* **have** [*simp*]: *Q-invar Qi M-invar* πi
    unfolding *prim-invar-impl-def Q-def* π*-def* **by** *auto*

  {
    fix *Qi* πi *d v* **and** *adjs* :: ('*v* × *nat*) *list*
    assume *Q-invar Qi M-invar* πi (*v*, *d*) ∈ *set adjs*
    then have
      (*case foreach-impl-body u* (*v*, *d*) (*Qi*, πi) *of*
        (*Qi*, πi) ⇒ *Q-invar Qi* ∧ *M-invar* πi)
          ∧ *map-prod Q-*α *M-lookup* (*foreach-impl-body u* (*v*, *d*) (*Qi*, πi))
            = *foreach-body u* (*v*, *d*) (*Q-*α *Qi*, *M-lookup* πi)
      unfolding *foreach-impl-body-def foreach-body-def*
      unfolding *Q-*α*-def*
      by (*auto simp*: *fun-eq-iff split*: *option.split*)

  } **note** *aux=this*

  from *foldr-refine*[
    where *I*=λ(*Qi*,πi). *Q-invar Qi* ∧ *M-invar* πi **and** α=*map-prod Q-*α *M-lookup*,
    of (*Qi*,πi) (*G-adj g u*) *foreach-impl-body u foreach-body u*
    ]
  and *A aux*[**where** *?adjs3*=(*G-adj g u*)]
    have *Q-invar Qi' M-invar* $\pi i'$
      and *1*: *foreach u* (*G-adj g u*) (*Q-*α *Qi*, *M-lookup* πi)
          = (*Q-*α *Qi'*, *M-lookup* $\pi i'$)
    unfolding *foreach-impl-def foreach-def*
    unfolding *Q-def* π*-def*
    by (*auto split*: *prod.splits*)
  then show *Q-invar Qi' M-invar* $\pi i'$ **by** *auto*

  from *1 foreach-refine*[**where** *adjs*=*G-adj g u* **and** *u*=*u*] **show**
    Q-α *Qi'* = *Qinter Q* π *u* **and** *M-lookup* $\pi i'$ = π' *Q* π *u*
    by (*auto simp*: *Q-def* π*-def*)

qed

definition *T-measure-impl* ≡ λ(*Qi*,πi). *T-measure2* (*Q-*α *Qi*) (*M-lookup* πi)

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-α-specs initQ-def initπ-def zero-enat-def*)

lemma *maintain-prim-invar-impl*:
  assumes
    I: *prim-invar-impl Qi πi* **and**
    NE: ¬ *Q-is-empty Qi* **and**
    MIN: *Q-getmin Qi* = (*u*, *d*) **and**
    FOREACH: *foreach-impl Qi πi u* (*G-adj g u*) = (*Qi′*, *πi′*)
  shows *prim-invar-impl* (*Q-delete u Qi′*) *πi′* (**is** *?G1*)
    and *T-measure-impl* (*Q-delete u Qi′*, *πi′*) < *T-measure-impl* (*Qi*,*πi*) (**is** *?G2*)
proof −
  note *II*[*simp*] = *I*[*unfolded prim-invar-impl-def*]
  note *FI*[*simp*] = *foreach-impl-correct*[*OF FOREACH I*]
  note *MIN′* = *Q-α-min*[*OF MIN - NE*, *simplified*]

  show *?G1*
    unfolding *prim-invar-impl-def*
    using *Q-α-delete maintain-invar2*[*OF - MIN′*]
    by (*simp add*: *Q′-def*)

  show *?G2*
    unfolding *prim-invar-impl-def T-measure-impl-def*
    using *Q-α-delete maintain-invar2*[*OF - MIN′*]
    apply (*simp add*: *Q′-def Q-α-def*)
    by (*metis FI*(*3*) *II Q′-def Q-α-def*
      ‹⋀π. *prim-invar2* (*Q-α Qi*) π
        ⟹ *T-measure2* (*Q′* (*Q-α Qi*) π *u*) (π′ (*Q-α Qi*) π *u*)
          < *T-measure2* (*Q-α Qi*) π›)

qed

lemma *maintain-prim-invar-impl-presentation*:
  assumes
    I: *prim-invar-impl Qi πi* **and**
    NE: ¬ *Q-is-empty Qi* **and**
    MIN: *Q-getmin Qi* = (*u*, *d*) **and**
    FOREACH: *foreach-impl Qi πi u* (*G-adj g u*) = (*Qi′*, *πi′*)
  shows *prim-invar-impl* (*Q-delete u Qi′*) *πi′*
    ∧ *T-measure-impl* (*Q-delete u Qi′*, *πi′*) < *T-measure-impl* (*Qi*,*πi*)
  using *maintain-prim-invar-impl assms* **by** *blast*

lemma *prim-invar-impl-finish*:
  ⟦*Q-is-empty Q*; *prim-invar-impl Q π*⟧
    ⟹ *invar-MST π* ∧ *is-MST* (*G-αw g*) *rg* (*α-MST π*)
  using *invar2-finish*
  by (*auto simp*: *Q-α-specs prim-invar-impl-def invar-MST-def α-MST-def Let-def*)

```
lemma prim-impl-correct:
  assumes prim-impl = πi
  shows
    invar-MST πi (is ?G1)
    is-MST (G-αw g) (component-of (G-αg g) r) (α-MST πi) (is ?G2)
proof −
  have let (Qi, πi) = outer-loop-impl (Q-update r 0 Q-empty) M-empty in
    invar-MST πi ∧ is-MST (G-αw g) rg (α-MST πi)
    unfolding outer-loop-impl-def
    apply (rule while-rule[where
      P=λ(Qi,πi). prim-invar-impl Qi πi and r=measure T-measure-impl])
    apply (all ‹clarsimp split: prod.splits simp: Q-α-specs›)
    apply (simp-all add: prim-invar-impl-init maintain-prim-invar-impl
                      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-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
  by unfold-locales


lemma G-from-list-unfold:  G-from-list = G.from-list
  by (simp add: G-add-edge-def G-empty-def G-from-list-def)


lemma [code]: G-from-list l = foldr (λ(e, d). G-add-edge e d) l G-empty
  by (simp add: G.from-list-def G-from-list-unfold)


global-interpretation Prim-Impl-Adts - - -
  G.αw G.αg 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

- - - $G.\alpha w$ $G.\alpha g$ *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 - - - $G.\alpha w$ $G.\alpha g$ **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* (πi, *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 n ot 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 ⇒ ¬*G.valid-wgraph-repr l* — Invalid input
  | *Some* πi ⇒
      G.valid-wgraph-repr l ∧ (*let Gi* = *G.from-list l in G.invar Gi* — Valid input
      ∧ *P.invar-MST* πi — Output satisfies invariants
      ∧ *is-MST* (*G.αw Gi*) (*component-of* (*G.αg Gi*) *r*) (*P.α-MST r* πi)) — 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 ⇒ ¬*G.valid-wgraph-repr l* — Invalid input
  | *Some* πi ⇒ *let*
      g=*G.αg* (*G.from-list l*);
      w=*G.αw* (*G.from-list l*);
      rg=*component-of g r*;
      t=*P.α-MST r* πi
    in
      G.valid-wgraph-repr l — Valid input
    ∧ *P.invar-MST* πi — Output satisfies invariants
    ∧ *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* :: - ⇒ *int* ⇒ -
  where *prim-list-impl-int* ≡ *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, ∞ 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 ∞.

fun *path* :: *'v* ⇒ *enat list* ⇒ *'v* ⇒ *bool* **where**
  path u [] *v* ⟷ *u*=*v*
| *path u* (*l*#*ls*) *v* ⟷ (∃ *uh*. *l* = *w* (*u*,*uh*) ∧ *path uh ls v*)

lemma *path-append*[*simp*]:
  path u (*ls1*@*ls2*) *v* ⟷ (∃ *w*. *path u ls1 w* ∧ *path w ls2 v*)
  by (*induction ls1 arbitrary*: *u*) *auto*

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

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* ≡ {*p* . *path u p v*}

lemma *paths-ne*: *paths u v* ≠ {} **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*∈*S* *v*∉*S*
  obtains *p1 x y p2*
    where *p* = *p1*@*w* (*x*,*y*)#*p2* *x*∈*S* *y*∉*S* *path u p1 x* *path y p2 v*
proof −
  have ∃ *p1 x y p2*. *p* = *p1*@*w* (*x*,*y*)#*p2* ∧ *x*∈*S* ∧ *y*∉*S* ∧ *path u p1 x* ∧ *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.prems* **obtain** *x* **where** [*simp*]: *a*=*w* (*u*,*x*) **and** *PX*: *path x p v*
      by *auto*

    show *?case* **proof** (*cases x*∈*S*)
      case *False* **with** *PX* ‹*u*∈*S*› **show** *?thesis* **by** *fastforce*
    next
      case *True* **from** *Cons.IH*[*OF PX True* ‹*v*∉*S*›] **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 $\delta\ u\ v \equiv LEAST\ w{::}enat.\ w{\in}sum\text{-}list`paths\ u\ v$

lemma *obtain-shortest-path*:
  obtains p **where** *path s p u* $\delta\ s\ u = sum\text{-}list\ p$
  unfolding *δ-def* **using** *paths-ne*
  by (*smt Collect-empty-eq LeastI-ex WGraph.paths-def imageI image-iff mem-Collect-eq paths-def*)

lemma *shortest-path-least*:
  path s p u $\Longrightarrow \delta\ s\ u \le sum\text{-}list\ p$
  unfolding *δ-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*: $\delta\ s\ u \le w\ (s,\ u)$
  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*: $\delta\ s\ v \le \delta\ s\ u + w\ (u,v)$
proof −
  have *path s* (*p*@[*w* (*u*,*v*)]) *v* **if** *path s p u* **for** *p* **using** *that* **by** *auto*
  then have (+) (*w* (*u*,*v*)) ‘ *sum-list* ‘ *paths s u* ⊆ *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**
    (*LEAST y*::*enat*. *y* ∈ *sum-list* ‘ *paths s v*)
    ≤ (*LEAST x*::*enat*. *x* ∈ (+) (*w* (*u*,*v*)) ‘ *sum-list* ‘ *paths s u*)
    by (*auto simp*: *paths-def*)
  also have ... = (*LEAST x*. *x* ∈ *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** *δ-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 ∞!

lemma *shortest-path-prefix*:

assumes *path s* p_1 *x path x* p_2 *u*
and *DSU*: $\delta\ s\ u = $ *sum-list* p_1 + *sum-list* p_2 $\delta\ s\ u < \infty$
shows $\delta\ s\ x = $ *sum-list* p_1 $\delta\ s\ x < \infty$
proof −
have $\delta\ s\ x \leq$ *sum-list* p_1 **using** *assms shortest-path-least* **by** *blast*
moreover have $\neg\delta\ s\ x <$ *sum-list* p_1 **proof**
assume $\delta\ s\ x <$ *sum-list* p_1
then obtain $p_1{}'$ **where** *path s* $p_1{}'$ *x sum-list* $p_1{}'$ < *sum-list* p_1
by (*auto intro*: *obtain-shortest-path*[*of s x*])
with ‹*path x* p_2 *u*› *shortest-path-least*[*of s* $p_1{}'$@p_2 *u*] *DSU* **show** *False*
by *fastforce*
qed
ultimately show $\delta\ s\ x = $ *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 α :: $'g \Rightarrow ('v)$ *wgraph*
and *invar* :: $'g \Rightarrow$ *bool*
and *succ* :: $'g \Rightarrow 'v \Rightarrow (nat \times 'v)$ *list*
and *empty-graph* :: $'g$
and *add-edge* :: $'v \times 'v \Rightarrow nat \Rightarrow 'g \Rightarrow 'g$
assumes *succ-correct*: *invar g* $\Longrightarrow$ *set* (*succ g u*) = $\{(d,v).\ \alpha\ g\ (u,v) = $ *enat d*$\}$
assumes *empty-graph-correct*:
invar empty-graph
α *empty-graph* = (λ-. ∞)
assumes *add-edge-correct*:
invar g $\Longrightarrow \alpha\ g\ e = \infty \Longrightarrow$ *invar* (*add-edge e d g*)
invar g $\Longrightarrow \alpha\ g\ e = \infty \Longrightarrow \alpha$ (*add-edge e d g*) = (α *g*)(*e*:=*enat d*)
begin

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

end

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

2.2.1 Constructing Weighted Graphs from Lists

lemma *edges-empty*[*simp*]: *WGraph.edges* (λ-. ∞) = {}
  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* (λ(*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 $\Longrightarrow$ $\exists$! *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*=λ(*u*,*v*).
      if $\exists$ *d*. (*u*,*v*,*d*)$\in$*set l then enat* (*SOME d*. (*u*,*v*,*d*)$\in$*set l*) *else* ∞])
    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* (λ(*u*,*v*,*d*). *add-edge* (*u*,*v*) *d*) *l empty-graph*

definition *edges-undef l w* ≡ ∀ *u v d*. (*u*,*v*,*d*)∈*set l* ⟶ *w* (*u*,*v*) = ∞

lemma *edges-undef-simps*[*simp*]:
  edges-undef [] *w*
  edges-undef l (λ-. ∞)
  edges-undef ((*u*,*v*,*d*)#*l*) *w* ⟷ *edges-undef l w* ∧ *w* (*u*,*v*) = ∞
  edges-undef l (*w*((*u*,*v*) := *enat d*)) ⟷ *edges-undef l w* ∧ (∀ *d'*. (*u*,*v*,*d'*)∉*set l*)
  by (*auto simp*: *edges-undef-def*)

lemma *from-list-correct-aux*:
  assumes *valid-graph-rep l*
  assumes *edges-undef l* (α *g*)
  assumes *invar g*
  defines *g'* ≡ *fold* (λ(*u*,*v*,*d*). *add-edge* (*u*,*v*) *d*) *l g*
  shows *invar g'*
    and (∀ *u v d*. α *g'* (*u*,*v*) = *enat d* ⟷ α *g* (*u*,*v*) = *enat d* ∨ (*u*,*v*,*d*)∈*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*)∈*set l* ⟷ α (*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*) α (*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*) ∈ *set l*) **for** *u v d*
    by *blast*

  then show α (*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* $\equiv (\lambda\text{-}.\ \infty)(s{:=}0)$
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*: ‹$s \in S \lor (D=(\lambda\text{-}.\ \infty)(s:=0) \land S=\{\})$› — The start node is finished, or we are in initial state
assumes *S-precise*: $u \in S \Longrightarrow D\ u = \delta\ s\ u$ — Finished nodes have precise estimate
assumes *S-relaxed*: ‹$v \in S \Longrightarrow D\ u \le \delta\ s\ v + w\ (v,u)$› — Outgoing edges of finished nodes have been relaxed, using precise distance
begin

abbreviation (**in** *Dijkstra*) *D-invar* $\equiv$ *Dijkstra-Invar w s*

The invariant holds for the initial state

theorem (**in** *Dijkstra*) *invar-init*: *D-invar initD initS*
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*: $\delta\ s\ u \le$ (*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*: $D\ v = \delta\ s\ v \Longrightarrow$ *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*: $v \in S \Longrightarrow$ *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*: $\forall v.\ v \notin S \longrightarrow D\ u \le D\ v$
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** ‹$u \notin S$› *s-in-S* **by** *simp*
next
assume ‹$u \ne s$› — Later step of loop

The start node has already been finished

with *s-in-S MIN* **have** ‹$s \in S$› **apply** *clarsimp* **using** *infinity-ne-i0* **by** *metis*

show *?thesis*

Next, we handle the case that u is unreachable.

proof (*cases* ‹$\delta\ s\ u < \infty$›)
  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*: $\delta\ s\ u =$ *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* ‹*path s p u*› ‹$s \in S$› ‹$u \notin S$›] **obtain** *p1 x y p2* **where**
    [*simp*]: $p = p1\ @\ w\ (x,\ y)\ \#\ p2$
    and *DECOMP*: $x \in S\ y \notin S$ *path s p1 x 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*: $\delta\ s\ x =$ *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 ‹$\delta\ s\ u < \infty$› *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 ‹$x \in S$› **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 $\ldots \le \delta\ s\ u$ **using** *DSU* **by** (*simp add*: *DSX DSY*)
  moreover

The estimate for u is an upper bound

  have $\ldots \le D\ u$ **using** *upper-bound* **by** (*auto*)
  moreover

u was a node with smallest estimate

  have $\ldots \le D\ y$ **using** ‹$u \notin S$› ‹$y \notin S$› *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* ‹$s \in S$›)
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 ‹$u \notin S$›

$D\ u$ 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*: $u \notin S$
assumes *UNI*: $D\ u < \infty$
shows *unfinished-dnodes* (*Set.insert u S*) $\subset$ *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* $< \infty$
apply (*rule obtain-shortest-path*[*of s u*])
using ‹$\delta\ s\ u < \infty$› **by** *auto*
with ‹$u \notin S$› **show** *?thesis*
apply (*cases p rule*: *rev-cases*)
by (*auto simp*: *Dijkstra.all-dnodes-def*)
qed

Which implies the proposition

with ‹$u \notin S$› **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 ∞

fun *enat-of-option* :: *nat option* $\Rightarrow$ *enat* **where**
  enat-of-option None = ∞
| *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 = ∞ $\longleftrightarrow$ *x* = *None*
  enat n = *enat-of-option x* $\longleftrightarrow$ *x* = *Some n*
  ∞ = *enat-of-option x* $\longleftrightarrow$ *x* = *None*
  by (*cases x*; *auto*; *fail*)+

lemma *enat-of-option-le-conv*:
  enat-of-option m $\le$ *enat-of-option n* $\longleftrightarrow$ (*case* (*m*,*n*) *of*
    (-,*None*) $\Rightarrow$ *True*
   | (*Some a*, *Some b*) $\Rightarrow$ *a*$\le$*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*

```
  ∞ ⇒ Q v
| enat d ⇒ (case Q v of
    None ⇒ if v∈dom V then None else Some (du+d)
  | Some d' ⇒ 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∉S
  assumes UNI: D u = enat du

  shows coupling
    ((relax-outgoing' u du V Q)(u:=None)) (V(u↦du))
    (relax-outgoing u D) (Set.insert u S)
  using C unfolding coupling-def
proof (intro ext conjI; elim conjE)
  assume α: D = enat-of-option ∘ V ++ Q S = dom V
     and DD: dom V ∩ dom Q = {}

  show Set.insert u S = dom (V(u ↦ du))
    by (auto simp: α)

  have [simp]: Q u = Some du V u = None
    using DD UNI UNS by (auto simp: α)

  from DD
  show dom (V(u ↦ du)) ∩ dom ((relax-outgoing' u du V Q)(u := 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
    = (enat-of-option ∘ V(u ↦ du) ++ (relax-outgoing' u du V Q)(u := None)) v
  proof (cases v∈S)
    case True
    then show ?thesis using DD
      apply (simp add: relax-finished-id)
      by (auto
        simp: relax-outgoing'-def map-add-apply α 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 α min-def
        split: option.splits enat.splits)
```

qed
qed

Refinement of initial state

definition *initQ* ≡ *Map.empty*$(s \mapsto 0)$
definition *initV* ≡ *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* ≡ *unfinished-dnodes* (*dom V*)

lemma *coupling-unfinished*:
coupling Q V D S ⟹ *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 ⇒ *if* $v \in$*dom V then Q else Q*$(v \mapsto du+d)$
| *Some d′* ⇒ *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* ⊆ $\{(d,v).\ w\ (u,v) = enat\ d\}$
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 $(u,v) = \infty$
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* . . . **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 $\bigwedge s\ x.\ I\ s \Longrightarrow x \in set\ l \Longrightarrow I\ (f\ x\ s) \wedge \alpha\ (f\ x\ s) = f'\ x\ (\alpha\ s)$
  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* (λ(*d*,*v*) *Q*.
  case Q-lookup Q v of
    None $\Rightarrow$ *if M-lookup V v* $\neq$ *None then Q else Q-update v* (*du*+*d*) *Q*
  | *Some d'* $\Rightarrow$ *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** α=*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-init-Q*[*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-getmin*

definition (**in** *Dijkstra-Impl-Defs*)
  dijkstra-loop $\equiv$ *while* ($\lambda(Q,V)$. $\neg$ *Q-is-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*
  using *assms*
  by (*auto simp*: *coupling-def*)

lemma *dijkstra-loop-invar-and-empty*:
  shows *case dijkstra-loop of* (*Q*,*V*) $\Rightarrow$ *D-invar-impl Q V* $\wedge$ *Q-is-empty Q*
  unfolding *dijkstra-loop-def*
  apply (*rule while-rule*[**where**
      P=*case-prod D-invar-impl*
    and *r*=*inv-image finite-psubset* (*unfinished-dnodes$'$ o M-lookup o snd*)])
  apply (*all* ‹(*clarsimp split*: *prod.splits*)?›)
  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-is-empty Q D-invar-impl Q V Q-getmin Q* = (*u*, *du*)
  hence *Q-lookup Q* $\neq$ *Map.empty D-invar$'$* (*Q-lookup Q*) (*M-lookup V*)
    and [*simp*]: *Q-invar Q M-invar V*
    and *Q-lookup Q u* = *Some du* $\forall k'$ p'. *Q-lookup Q k$'$* = *Some p$'$* $\longrightarrow$ *du* $\leq$ *p$'$*
    by (*auto simp*: *D-invar-impl-def elim*: *Q.map-getminE*)

  then obtain *D S* **where**
    D-invar D S
    and *COUPLING*: *coupling* (*Q-lookup Q*) (*M-lookup V*) *D S*
    and *ABS-PRE*: *D u* = *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-α 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* ↦ *du*))
(*relax-outgoing u D*)
(*Set.insert u S*)
using *coupling-step*[*OF COUPLING* ‹*u*∉*S*› ‹*D u* = *enat du*›] **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* ‹*u*∉*S*›] *ABS-PRE*
using *COUPLING′*
by (*auto simp*: *D-invar-impl-def D-invar′-def*)

show *unfinished-dnodes′* (*M-lookup* (*M-update u du V*))
⊂ *unfinished-dnodes′* (*M-lookup V*)
∧ *finite* (*unfinished-dnodes′* (*M-lookup V*))
using *coupling-unfinished*[*OF COUPLING*] *coupling-unfinished*[*OF COUPLING′*]
using *unfinished-nodes-decr*[*OF* ‹*u*∉*S*›] *ABS-PRE*
by *simp*
qed

lemma *dijkstra-correct*:
M-invar dijkstra
M-lookup dijkstra u = *Some d* ⟷ *δ s u* = *enat d*
using *dijkstra-loop-invar-and-empty*
unfolding *dijkstra-def*
apply −
apply (*all* ‹*clarsimp simp*: *D-invar-impl-def*›)
apply (*clarsimp simp*: *D-invar′-def*)
subgoal for *Q V D S*
using *Dijkstra-Invar.invar-finish-imp-correct*[*of G-α 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* $\equiv$
  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. lookup D u* = *Some d* $\longleftrightarrow$ *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.