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*
$= \{ (V::'v\ set\ ,\ E).\ E \subseteq V \times V \wedge finite\ V \wedge sym\ E \wedge irrefl\ E \}$
⟨*proof*⟩

setup-lifting *type-definition-ugraph*

lift-definition *nodes-internal* :: $'v$ *ugraph* ⇒ $'v$ *set* **is** *fst* ⟨*proof*⟩
lift-definition *edges-internal* :: $'v$ *ugraph* ⇒ $('v\times'v)$ *set* **is** *snd* ⟨*proof*⟩
lift-definition *graph-internal* :: $'v$ *set* ⇒ $('v\times'v)$ *set* ⇒ $'v$ *ugraph*
is $\lambda V\ E.$ *if finite* $V \wedge$ *finite* E *then* $(V \cup fst`E \cup snd`E,\ (E \cup E^{-1}) - Id)$ *else* $(\{\},\{\})$
⟨*proof*⟩

definition *nodes* :: $'v$ *ugraph* ⇒ $'v$ *set*
where *nodes* = *nodes-internal*
definition *edges* :: $'v$ *ugraph* ⇒ $('v\times'v)$ *set*
where *edges* = *edges-internal*
definition *graph* :: $'v$ *set* ⇒ $('v\times'v)$ *set* ⇒ $'v$ *ugraph*
where *graph* = *graph-internal*

lemma *edges-subset*: *edges g* ⊆ *nodes g* × *nodes g*
⟨*proof*⟩

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

lemma *edges-sym*: *sym* (*edges g*)
⟨*proof*⟩

lemma *edges-irrefl*: *irrefl* (*edges g*)
⟨*proof*⟩

lemma *nodes-graph*: ⟦*finite* V; *finite* E⟧ ⟹ *nodes* (*graph* V E) = $V \cup fst`E \cup snd`E$

⟨*proof*⟩

lemma *edges-graph*: ⟦*finite* V; *finite* E⟧ ⟹ *edges* (*graph* V E) = $(E \cup E^{-1}) - Id$
⟨*proof*⟩

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

lemma *nodes-edges-graph-presentation*: ⟦*finite* V; *finite* E⟧
⟹ *nodes* (*graph* V E) = $V \cup fst`E \cup snd`E$ ∧ *edges* (*graph* V E) = $E \cup E^{-1} - Id$
⟨*proof*⟩

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

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

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*⟩

lemma *graph-eq-iff*: $g = g' \longleftrightarrow nodes\ g = nodes\ g' \land edges\ g = edges\ g'$
⟨*proof*⟩

lemma *edges-sym'*: $(u,v) \in edges\ g \Longrightarrow (v,u) \in edges\ g$ ⟨*proof*⟩

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

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

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

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

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

1.1.2 Connectedness Relation

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

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

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

1.1.3 Constructing Graphs

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

definition *graph-join* g_1 g_2 ≡ *graph* (*nodes* g_1 ∪ *nodes* g_2) (*edges* g_1 ∪ *edges* g_2)
definition *restrict-nodes* *g* *V* ≡ *graph* (*nodes* *g* ∩ *V*) (*edges* *g* ∩ *V*×*V*)
definition *restrict-edges* *g* *E* ≡ *graph* (*nodes* *g*) (*edges* *g* ∩ (E∪E^{-1}))

definition *nodes-edges-consistent* *V* *E* ≡ *finite* *V* ∧ *irrefl* *E* ∧ *sym* *E* ∧ *E* ⊆ *V*×*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*⟩

lemma *nec-empty*[*simp*]: *nodes-edges-consistent* {} {}
  ⟨*proof*⟩

lemma *graph-empty-accs*[*simp*]:
  nodes *graph-empty* = {}
  edges *graph-empty* = {}
  ⟨*proof*⟩

lemma *graph-empty*[*simp*]: *graph* {} {} = *graph-empty*
  ⟨*proof*⟩

lemma *nodes-empty-iff-empty*[*simp*]:
  nodes *G* = {} ⟷ *G*=*graph* {} {}
  {} = *nodes* *G* ⟷ *G*=*graph-empty*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*))
  ⟨*proof*⟩

lemma *edges-ins-edge′*[*simp*]:
  u≠*v* ⟹ *edges* (*ins-edge* (*u*,*v*) *g*) = {(*u*,*v*),(*v*,*u*)} ∪ *edges* *g*
  ⟨*proof*⟩

lemma *edges-ins-edge-ss*: *edges* *g* ⊆ *edges* (*ins-edge* *e* *g*)
  ⟨*proof*⟩

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
⟨*proof*⟩

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*×*V*
⟨*proof*⟩

lemma *nodes-restrict-edges*[*simp*]: *nodes* (*restrict-edges g E*) = *nodes g*
and *edges-restrict-edges*[*simp*]: *edges* (*restrict-edges g E*) = *edges g* ∩ (E∪E^{-1})
⟨*proof*⟩

lemma *unrestricte-edges*: *edges* (*restrict-edges g E*) ⊆ *edges g* ⟨*proof*⟩
lemma *unrestrictn-edges*: *edges* (*restrict-nodes g V*) ⊆ *edges g* ⟨*proof*⟩

lemma *unrestrict-nodes*: *nodes* (*restrict-edges g E*) ⊆ *nodes g* ⟨*proof*⟩

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* ⟨*proof*⟩

lemma *path-append*[*simp*]:
path g u (*p1*@*p2*) *w* ⟷ (∃ *v*. *path g u p1 v* ∧ *path g v p2 w*)
⟨*proof*⟩

lemma *path-transs1*[*trans*]:
path g u p v ⟹ (*v*,*w*)∈*edges g* ⟹ *path g u* (*p*@[(*v*,*w*)]) *w*
(*u*,*v*)∈*edges g* ⟹ *path g v p w* ⟹ *path g u* ((*u*,*v*)#*p*) *w*
path g u p1 v ⟹ *path g v p2 w* ⟹ *path g u* (*p1*@*p2*) *w*
⟨*proof*⟩

lemma *path-graph-empty*[*simp*]: *path graph-empty u p v* ⟷ *v*=*u* ∧ *p*=[]
⟨*proof*⟩

abbreviation *revp p* ≡ *rev* (*map prod.swap p*)
lemma *revp-alt*: *revp p* = *rev* (*map* (λ(*u*,*v*). (*v*,*u*)) *p*) ⟨*proof*⟩

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

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

lemma *path-transs2*[*trans*]:
path g u p v ⟹ (*w*,*v*)∈*edges g* ⟹ *path g u* (*p*@[(*v*,*w*)]) *w*
(*v*,*u*)∈*edges g* ⟹ *path g v p w* ⟹ *path g u* ((*u*,*v*)#*p*) *w*
path g u p1 v ⟹ *path g w p2 v* ⟹ *path g u* (*p1*@*revp p2*) *w*

⟨*proof*⟩

lemma *path-edges*: $path\ g\ u\ p\ v \Longrightarrow set\ p \subseteq edges\ g$
⟨*proof*⟩

lemma *path-graph-cong*:
$[\![path\ g_1\ u\ p\ v;\ set\ p \subseteq edges\ g_1 \Longrightarrow set\ p \subseteq edges\ g_2]\!] \Longrightarrow path\ g_2\ u\ p\ v$
⟨*proof*⟩

lemma *path-endpoints*:
assumes $path\ g\ u\ p\ v\ p{\neq}[]$ **shows** $u{\in}nodes\ g\ v{\in}nodes\ g$
⟨*proof*⟩

lemma *path-mono*: $edges\ g \subseteq edges\ g' \Longrightarrow path\ g\ u\ p\ v \Longrightarrow path\ g'\ u\ p\ v$
⟨*proof*⟩

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

lemma *unrestrict-path-edges*: $path\ (restrict\text{-}edges\ g\ E)\ u\ p\ v \Longrightarrow path\ g\ u\ p\ v$
⟨*proof*⟩

lemma *unrestrict-path-nodes*: $path\ (restrict\text{-}nodes\ g\ E)\ u\ p\ v \Longrightarrow path\ g\ u\ p\ v$
⟨*proof*⟩

Paths and Connectedness

lemma *rtrancl-edges-iff-path*: $(u,v){\in}(edges\ g)^* \longleftrightarrow (\exists\, p.\ path\ g\ u\ p\ v)$
⟨*proof*⟩

lemma *rtrancl-edges-pathE*:
assumes $(u,v){\in}(edges\ g)^*$ **obtains** p **where** $path\ g\ u\ p\ v$
⟨*proof*⟩

lemma *path-rtrancl-edgesD*: $path\ g\ u\ p\ v \Longrightarrow (u,v){\in}(edges\ g)^*$
⟨*proof*⟩

Simple Paths

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

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

lemma *in-uedge-conv*[*simp*]: $x{\in}uedge\ (u,v) \longleftrightarrow x{=}u \lor x{=}v$
⟨*proof*⟩

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

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

lemma *uedge-in-set-eq*: $uedge\ (u,\ v) \in uedge\ `\ S \longleftrightarrow (u,v) \in S \vee (v,u) \in S$
$\langle proof \rangle$

lemma *uedge-commute*: $uedge\ (a,b) = uedge\ (b,a)$ $\langle proof \rangle$

lemma *simple-empty*[*simp*]: *simple* []
$\langle proof \rangle$

lemma *simple-cons*[*simp*]: $simple\ (e\#p) \longleftrightarrow uedge\ e \notin uedge\ `\ set\ p \wedge simple\ p$
$\langle proof \rangle$

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

lemma *simplify-pathD*:
$path\ g\ u\ p\ v \Longrightarrow \exists p'.\ path\ g\ u\ p'\ v \wedge simple\ p' \wedge set\ p' \subseteq set\ p$
$\langle proof \rangle$

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

Splitting Paths

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

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 $p = p_1 @ [(u_1,v_1)] @ p_2 @ [(u_2,v_2)] @ p_3$ $P\ u_1$ $\neg P\ v_1$ $\neg P\ u_2$ $P\ v_2$
$\langle proof \rangle$

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

⟨*proof*⟩

lemma *path-change*:
assumes $u \in S$ $v \notin S$ *path g u p v simple p*
obtains *x y p1 p2* **where**
$(x,y) \in$ *set p* $x \in S$ $y \notin S$
path (*restrict-edges g* $(-\{(x,y),(y,x)\})$) *u p1 x*
path (*restrict-edges g* $(-\{(x,y),(y,x)\})$) *y p2 v*
⟨*proof*⟩

1.1.5 Cycles

definition *cycle-free g* $\equiv \nexists$ *p u.* $p \neq [] \land$ *simple p* $\land$ *path g u p u*

lemma *cycle-free-alt-in-nodes*:
cycle-free g $\equiv \nexists$ *p u.* $p \neq [] \land u \in$ *nodes g* $\land$ *simple p* $\land$ *path g u p u*
⟨*proof*⟩

lemma *cycle-freeI*:
assumes $\bigwedge p\ u.$ ⟦ *path g u p u*; $p \neq []$; *simple p* ⟧ $\Longrightarrow$ *False*
shows *cycle-free g*
⟨*proof*⟩

lemma *cycle-freeD*:
assumes *cycle-free g path g u p u* $p \neq []$ *simple p*
shows *False*
⟨*proof*⟩

lemma *cycle-free-antimono*: *edges g* $\subseteq$ *edges g'* $\Longrightarrow$ *cycle-free g'* $\Longrightarrow$ *cycle-free g*
⟨*proof*⟩

lemma *cycle-free-empty*[*simp*]: *cycle-free graph-empty*
⟨*proof*⟩

lemma *cycle-free-no-edges*: *edges g* = {} $\Longrightarrow$ *cycle-free g*
⟨*proof*⟩

lemma *simple-path-cycle-free-unique*:
assumes *CF*: *cycle-free g*
assumes *P*: *path g u p v path g u p' v simple p simple p'*
shows $p = p'$
⟨*proof*⟩

Characterization by Removing Edge

lemma *cycle-free-alt*: *cycle-free g*
$\longleftrightarrow$ ($\forall$ $e \in$ *edges g.* $e \notin$(*edges* (*restrict-edges g* $(-\{e,$*prod.swap e*$\})))^*$)
⟨*proof*⟩

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

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

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

1.1.6 Connected Graphs

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

lemma *connectedI*[*intro?*]:
assumes $\bigwedge u\ v.\ \llbracket u \in nodes\ g;\ v \in nodes\ g \rrbracket \Longrightarrow (u,v) \in (edges\ g)^*$
shows *connected g*
⟨*proof*⟩

lemma *connectedD*[*intro?*]:
assumes $connected\ g\ u \in nodes\ g\ v \in nodes\ g$
shows $(u,v) \in (edges\ g)^*$
⟨*proof*⟩

lemma *connected-empty*[*simp*]: *connected graph-empty*
⟨*proof*⟩

1.1.7 Component Containing Node

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

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

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

lemma *reachable-nodes-steps*:
$(edges\ g)^*$ `` *reachable-nodes g r* ⊆ *reachable-nodes g r*
⟨*proof*⟩

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*⟩

lemma *reachable-nodes-steps'*:
assumes $u \in$ *reachable-nodes g r* $(u, v) \in (edges\ g)^*$
shows $v \in$*reachable-nodes g r* $(u, v) \in (edges\ (component\text{-}of\ g\ r))^*$
⟨*proof*⟩

lemma *reachable-not-node*: $r \notin$*nodes g* ⟹ *reachable-nodes g r* = $\{r\}$
⟨*proof*⟩

lemma *nodes-of-component*[*simp*]: *nodes* (*component-of g r*) = *reachable-nodes g r*
⟨*proof*⟩

lemma *component-connected*[*simp*, *intro*!]: *connected* (*component-of g r*)
⟨*proof*⟩

lemma *component-edges-subset*: *edges* (*component-of g r*) ⊆ *edges g*
⟨*proof*⟩

lemma *component-path*: $u \in$*nodes* (*component-of g r*) ⟹
path (*component-of g r*) *u p v* ⟷ *path g u p v*
⟨*proof*⟩

lemma *component-cycle-free*: *cycle-free g* ⟹ *cycle-free* (*component-of g r*)
⟨*proof*⟩

lemma *component-of-connected-graph*:
⟦*connected g*; $r \in$*nodes g*⟧ ⟹ *component-of g r* = *g*
⟨*proof*⟩

lemma *component-of-not-node*: $r \notin$*nodes g* ⟹ *component-of g r* = *graph* $\{r\}$ $\{\}$
⟨*proof*⟩

1.1.8 Trees

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

lemma *tree-empty*[*simp*]: *tree graph-empty* ⟨*proof*⟩

lemma *component-of-tree*: *tree T* ⟹ *tree* (*component-of T r*)

⟨*proof*⟩

Joining and Splitting Trees on Single Edge

lemma *join-connected*:
  assumes *CONN*: *connected* g_1 *connected* g_2
  assumes *IN-NODES*: *u*∈*nodes* g_1 *v*∈*nodes* g_2
  shows *connected* (*ins-edge* (*u*,*v*) (*graph-join* g_1 g_2)) (**is** *connected ?g′*)
  ⟨*proof*⟩

lemma *join-cycle-free*:
  assumes *CYCF*: *cycle-free* g_1 *cycle-free* g_2
  assumes *DJ*: *nodes* g_1 ∩ *nodes* g_2 = {}
  assumes *IN-NODES*: *u*∈*nodes* g_1 *v*∈*nodes* g_2
  shows *cycle-free* (*ins-edge* (*u*,*v*) (*graph-join* g_1 g_2)) (**is** *cycle-free ?g′*)
⟨*proof*⟩

lemma *join-trees*:
  assumes *TREE*: *tree* g_1 *tree* g_2
  assumes *DJ*: *nodes* g_1 ∩ *nodes* g_2 = {}
  assumes *IN-NODES*: *u*∈*nodes* g_1 *v*∈*nodes* g_2
  shows *tree* (*ins-edge* (*u*,*v*) (*graph-join* g_1 g_2))
  ⟨*proof*⟩

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*⟩

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*)
  ⟨*proof*⟩

lemma *path-singleton*[*simp*]: *path* (*ins-node u graph-empty*) *v p w* ⟷ *v*=*w* ∧ *p*=[]

  ⟨*proof*⟩

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

lemma *tree-add-edge-in-out*:
assumes *tree T*
assumes $u \in$ *nodes T* $v \notin$ *nodes T*
shows *tree* (*ins-edge* (u,v) *T*)
⟨*proof*⟩

Remove edges on cycles until the graph is cycle free

lemma *ex-spanning-tree*:
connected g $\Longrightarrow \exists\, t.$ *is-spanning-tree g t*
⟨*proof*⟩

1.2 Weighted Undirected Graphs

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

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

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

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*
⟨*proof*⟩

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

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

lemma *weight-cong*: $edges\ g_1 = edges\ g_2 \Longrightarrow$ *weight w* $g_1 =$ *weight w* g_2
⟨*proof*⟩

lemma *weight-mono*: $edges\ g \subseteq edges\ g' \Longrightarrow$ *weight w g* $\leq$ *weight w* g'
⟨*proof*⟩

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

lemma *weight-del-edge*[*simp*]:
  assumes $(x,y) \in edges\ T$
  shows $weight\ w\ (restrict\text{-}edges\ T\ (-\ \{(x,\ y),\ (y,\ x)\}))\ =\ weight\ w\ T\ -\ w\ \{x,y\}$
⟨*proof*⟩

1.2.1 Minimum Spanning Trees

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

lemma *exists-MST*: $connected\ g \Longrightarrow \exists\, t.\ is\text{-}MST\ w\ g\ t$
  ⟨*proof*⟩

end

1.3 Abstract Graph Datatype

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

1.3.1 Abstract Weighted Graph

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

begin

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

lemma *empty-spec-presentation*:

invar empty ∧ *αg empty* = *graph* {} {} ∧ *αw empty* = (λ-. *0*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *nodes-of-graph-from-list*: *nodes* (*graph-from-list l*) = *fst*‘*set l* ∪ *snd*‘*set l*
⟨*proof*⟩

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

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*)
⟨*proof*⟩

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

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

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*
⟨*proof*⟩

lemma *weight-from-list-correct*:
assumes *valid-weight-repr l*
assumes ((*u*,*v*),*d*)∈*set l*
shows *weight-from-list l* {*u*,*v*} = *d*
⟨*proof*⟩

context *adt-wgraph*
begin

definition *valid-wgraph-repr l*
⟷ *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*
⟨*proof*⟩

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*
⟨*proof*⟩

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

lemma *from-list-correct-presentation*:

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

⟨*proof*⟩

end

end

1.4 Abstract Prim Algorithm

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

1.4.1 Generic Algorithm: Light Edges

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

lemma *is-subset-MST-empty*[*simp*]: *connected g* $\Longrightarrow$ *is-subset-MST w g* {}
⟨*proof*⟩

We fix a start node and a weighted graph

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

Reachable part of the graph

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

lemma *reachable-connected*[*simp*, *intro*!]: *connected rg*
⟨*proof*⟩

lemma *reachable-edges-subset*: *edges rg* $\subseteq$ *edges g*
⟨*proof*⟩

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

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

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

end

1.4.2 Abstract Prim: Growing a Tree

context *Prim* **begin**

The current nodes

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

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

corollary *respects-cut*: *respects-cut A* $(S\ A)$
⟨*proof*⟩

Refined invariant: Adds connectedness of A

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

Measure: Number of nodes not in tree

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

end

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

locale *Prim-Invar1-loc* =
Prim w g r **for** *w g* **and** r :: $'v$ +
fixes A :: $('v{\times}'v)$ *set*
assumes *invar1*: *prim-invar1 A*
begin
lemma *subset-MST*: *is-subset-MST w rg A*
⟨*proof*⟩

lemma *A-connected*: $(u,v){\in}A \Longrightarrow (v,r){\in}A^*$
⟨*proof*⟩

lemma *S-alt-def*: $S\ A = \{r\} \cup$ *fst*‘A
⟨*proof*⟩

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

lemma *A-edges*: $A \subseteq$ *edges g*
⟨*proof*⟩

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

lemma *S-edge-reachable*: ⟦*u*∈*S A*; (*u*,*v*)∈*edges g* ⟧ ⟹ (*u*,*v*)∈*edges rg*
⟨*proof*⟩

lemma *edges-S-rg-edges*: *edges g* ∩ *S A*×−*S A* = *edges rg* ∩ *S A*×−*S A*
⟨*proof*⟩

lemma *T-measure1-less*: *T-measure1 A* < *card* (*nodes rg*)
⟨*proof*⟩

lemma *finite-A*[*simp*, *intro*!]: *finite A*
⟨*proof*⟩

lemma *finite-S*[*simp*, *intro*!]: *finite* (*S A*)
⟨*proof*⟩

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

end

context *Prim* **begin**

lemma *invar1-initial*: *prim-invar1* {}
⟨*proof*⟩

lemma *maintain-invar1*:
assumes *invar*: *prim-invar1 A*
assumes *light-edge*: *light-edge* (*S A*) *u v*
shows *prim-invar1* ({(*v*,*u*)}∪*A*)
∧ *T-measure1* ({(*v*,*u*)}∪*A*) < *T-measure1 A* (**is** *?G1* ∧ *?G2*)
⟨*proof*⟩

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*⟩

end

1.4.3 Prim: Using a Priority Queue

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

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

Abstraction to edge set

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

Initialization

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

Step

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

State after inner loop

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

State after one step

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

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

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

lemma *prim-invar2-ctd-alt-aux1*:
assumes $prim\text{-}invar1\ (A\ Q\ \pi)$
assumes $Q\ u \neq \infty\ \ u{\neq}r$
shows $u{\notin}S\ (A\ Q\ \pi)$
⟨*proof*⟩

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

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

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

lemma *Q'-init-eq*:
Q' initQ initπ *r* = (λ*u*. *if* (*u*,*r*)$\in$*edges rg then enat* (*w* {*u*,*r*}) *else* ∞)
⟨*proof*⟩

lemma π*'-init-eq*:
π*' initQ init*π *r* = (λ*u*. *if* (*u*,*r*)$\in$*edges rg then Some r else None*)
⟨*proof*⟩

lemma *A-init-eq*: *A initQ init*π = {}
⟨*proof*⟩

lemma *S-empty*: *S* {} = {*r*} ⟨*proof*⟩

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*⟩

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*)
$\wedge$ *T-measure2* (*Q' Q* π *u*) (π*' Q* π *u*) < *T-measure2 Q* π
⟨*proof*⟩

end
⟨*proof*⟩⟨*proof*⟩

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

locale *Prim-Invar2-ctd-loc* =

Prim2 w g r **for** *w g* **and** *r* :: *'v* +
fixes *Q π*
assumes *invar2*: *prim-invar2-ctd Q π*
begin

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

lemma *upd-cond-alt*: *upd-cond Q π u v'* ⟷
(*v'*,*u*) ∈ *edges g* ∧ *v'*∉*S* (*A Q π*) ∧ *enat* (*w* {*v'*,*u*}) < *Q v'*
⟨*proof*⟩

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'*})
⟨*proof*⟩

lemma *π-def-on-S*: ⟦*u*∈*S* (*A Q π*); *u*≠*r*⟧ ⟹ *π u* ≠ *None*
⟨*proof*⟩

lemma *π-def-on-edges-to-S*: ⟦*v*∈*S* (*A Q π*); *u*≠*r*; (*u*,*v*)∈*edges rg*⟧ ⟹ *π u* ≠ *None*
⟨*proof*⟩

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*⟩

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*⟩

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

lemma *Q-min-is-light-presentation*:
assumes *INV*: *prim-invar2-ctd* $Q\ \pi$
assumes *UNS*: $Q\ u = enat\ d$
assumes *MIN*: $\forall\, v.\ enat\ d \leq Q\ v$
obtains v **where** $\pi\ u = Some\ v$ *light-edge* $(S\ (A\ Q\ \pi))\ v\ u$
⟨*proof*⟩

lemma *maintain-invar2-ctd-presentation*:
assumes *INV*: *prim-invar2-ctd* $Q\ \pi$
assumes *UNS*: $Q\ u = enat\ d$
assumes *MIN*: $\forall\, v.\ enat\ d \leq Q\ v$
shows *prim-invar2-ctd* $(Q'\ Q\ \pi\ u)\ (\pi'\ Q\ \pi\ u)$
$\wedge$ *T-measure2* $(Q'\ Q\ \pi\ u)\ (\pi'\ Q\ \pi\ u) <$ *T-measure2* $Q\ \pi$
⟨*proof*⟩

lemma *not-invar2-ctd-init*:
prim-invar2-init $Q\ \pi \Longrightarrow \neg$*prim-invar2-ctd* $Q\ \pi$
⟨*proof*⟩

lemma *invar2-init-init*: *prim-invar2-init initQ init*π
⟨*proof*⟩

lemma *invar2-init*: *prim-invar2 initQ init*π
⟨*proof*⟩

lemma *maintain-invar2*:
assumes *A*: *prim-invar2* $Q\ \pi$
assumes *UNS*: $Q\ u = enat\ d$
assumes *MIN*: $\forall\, v.\ enat\ d \leq Q\ v$
shows *prim-invar2* $(Q'\ Q\ \pi\ u)\ (\pi'\ Q\ \pi\ u)$ (**is** *?G1*)
and *T-measure2* $(Q'\ Q\ \pi\ u)\ (\pi'\ Q\ \pi\ u) <$ *T-measure2* $Q\ \pi$ (**is** *?G2*)
⟨*proof*⟩

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

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

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

end

1.4.4 Refinement of Inner Foreach Loop

context *Prim2* **begin**

definition *foreach-body u ≡ λ(v,d) (Q,π).*
if v=r then (Q,π)
else
case (Q v, π v) of
(∞,None) ⇒ (Q(v:=enat d), π(v↦u))
| (enat d′,-) ⇒ if d<d′ then (Q(v:=enat d), π(v↦u)) else (Q,π)
| (∞,Some -) ⇒ (Q,π)

lemma *foreach-body-alt*: *foreach-body u = (λ(v,d) (Q,π).*
if v≠r ∧ (π v = None ∨ Q v ≠ ∞) ∧ enat d < Q v then
(Q(v:=enat d), π(v↦u))
else
(Q,π)
)
⟨*proof*⟩

definition *foreach* **where**
foreach u adjs Qπ = foldr (foreach-body u) adjs Qπ

definition ⋀*Q V*.
Qigen Q π u adjs v = (if v ∉ fst‘set adjs then Q v else Qinter Q π u v)
definition ⋀*Q V π*.
π′gen Q π u adjs v = (if v ∉ fst‘set adjs then π v else π′ Q π u v)

context begin

private lemma *Qc*:
Qigen Q π u ((v, w {u, v}) # adjs) x
= (if x=v then Qinter Q π u v else Qigen Q π u adjs x) **for** *x*
⟨*proof*⟩ **lemma** *πc*:
π′gen Q π u ((v, w {u, v}) # adjs) x
= (if x=v then π′ Q π u v else π′gen Q π u adjs x) **for** *x*
⟨*proof*⟩

lemma *foreach-refine-gen*:
assumes *set adjs ⊆ {(v,d). (u,v)∈edges g ∧ w {u,v} = d}*
shows *foreach u adjs (Q,π) = (Qigen Q π u adjs,π′gen Q π u adjs)*
⟨*proof*⟩

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*⟩

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*) ⟹ (*x*≠*y* ∧ *e*={*x*,*y*})
  ⟨*proof*⟩

lemma *epair-not-sng*[*simp*]: *epair e* ≠ *Some* (*x*,*x*)
  ⟨*proof*⟩

lemma *epair-None*[*simp*]: *epair* {*a*,*b*} = *None* ⟷ *a*=*b*
  ⟨*proof*⟩

1.5.2 Generic Implementation

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

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

  for *M-empty M-update M-delete*
  and *M-lookup* :: *′m* ⇒ *′v* ⇒ ((*′v*×*nat*) *list*) *option*
  and *M-invar*
begin

definition *αnodes-aux g* ≡ *dom* (*M-lookup g*)

definition *αedges-aux g*

≡ ({(*u*,*v*). ∃ *xs d*. *M-lookup g u* = *Some xs* ∧ (*v*,*d*)∈*set xs* })

definition *αg g* ≡ *graph* (*αnodes-aux g*) (*αedges-aux g*)

definition *αw g e* ≡ *case epair e of*
Some (*u*,*v*) ⇒ (
case M-lookup g u of
None ⇒ *0*
| *Some xs* ⇒ *the-default 0* (*map-of xs v*)
)
| *None* ⇒ *0*

definition *invar* :: *'m* ⇒ *bool* **where**
invar g ≡
M-invar g ∧ *finite* (*dom* (*M-lookup g*))
∧ (∀ *u xs*. *M-lookup g u* = *Some xs* ⟶
distinct (*map fst xs*)
∧ *u*∉*set* (*map fst xs*)
∧ (∀ (*v*,*d*)∈*set xs*. (*u*,*d*)∈*set* (*the-default* [] (*M-lookup g v*)))
)

lemma *in-the-default-empty-conv*[*simp*]:
x∈*set* (*the-default* [] *m*) ⟷ (∃ *xs*. *m*=*Some xs* ∧ *x*∈*set xs*)
⟨*proof*⟩

lemma *αedges-irrefl*: *invar g* ⟹ *irrefl* (*αedges-aux g*)
⟨*proof*⟩

lemma *αedges-sym*: *invar g* ⟹ *sym* (*αedges-aux g*)
⟨*proof*⟩

lemma *αedges-subset*: *invar g* ⟹ *αedges-aux g* ⊆ *αnodes-aux g* × *αnodes-aux g*
⟨*proof*⟩

lemma *αnodes-finite*[*simp*, *intro*!]: *invar g* ⟹ *finite* (*αnodes-aux g*)
⟨*proof*⟩

lemma *αedges-finite*[*simp*, *intro*!]: *invar g* ⟹ *finite* (*αedges-aux g*)
⟨*proof*⟩

definition *adj* :: *'m* ⇒ *'v* ⇒ (*'v*×*nat*) *list* **where**
adj g v = *the-default* [] (*M-lookup g v*)

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

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

definition *add-edge* :: $'v \times 'v \Rightarrow nat \Rightarrow 'm \Rightarrow 'm$ **where**
add-edge ≡ λ(*u*,*v*) *d g*. *add-edge1* (*v*,*u*) *d* (*add-edge1* (*u*,*v*) *d g*)

lemma *edges-αg-aux*: *invar g* ⟹ *edges* (*αg g*) = *αedges-aux g*
⟨*proof*⟩

lemma *nodes-αg-aux*: *invar g* ⟹ *nodes* (*αg g*) = *αnodes-aux g*
⟨*proof*⟩

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

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

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

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

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

lemma *invar-add-edge-aux*:
⟦*invar g*; (*u*, *v*) ∉ *edges* (*αg g*); *u* ≠ *v*⟧ ⟹ *invar* (*add-edge* (*u*, *v*) *d g*)
⟨*proof*⟩

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

end

end

1.6 Implementation of Prim's Algorithm

theory *Prim-Impl*
imports
Prim-Abstract
Undirected-Graph-Impl
HOL−Library.While-Combinator

Priority-Search-Trees.PST-RBT
HOL−Data-Structures.RBT-Map
begin

1.6.1 Implementation using ADT Interfaces

locale *Prim-Impl-Adts* =
G: *adt-wgraph G-αw G-αg 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-αw* **and** *G-αg* :: *'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-αw* = *G-αw* **and** *G-αg* = *G-αg*
for *typG* :: *'g itself* **and** *typM* :: *'m itself* **and** *typQ* :: *'q itself*
and *G-αw* **and** *G-αg* :: *'g* ⇒ (*'v*::*linorder*) *ugraph* **and** *g* :: *'g* **and** *r* :: *'v*
begin

Concrete Algorithm

term *M-lookup*
definition *foreach-impl-body u* ≡ (λ(*v*,*d*) (*Qi*,*πi*).
if v=*r then* (*Qi*,*πi*)
else
case (*Q-lookup Qi v*, *M-lookup πi v*) *of*
(*None*,*None*) ⇒ (*Q-update v d Qi*, *M-update v u πi*)
| (*Some d'*,-) ⇒ (*if d*<*d' then* (*Q-update v d Qi*, *M-update v u πi*) *else* (*Qi*,*πi*))
| (*None*, *Some* -) ⇒ (*Qi*,*πi*)
)

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,πi)

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

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

The whole algorithm as one function

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

Abstraction of Result

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

- The map *πi* and set *Vi* satisfy their implementation invariants

- The 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* πi ≡ *M-invar* πi

definition *α-MST* πi ≡ *graph* {*r*} {(*u*,*v*) | *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-αw* = *G-αw* **and** *G-αg* = *G-αg*
  for *typG* :: *'g itself* **and** *typM* :: *'m itself* **and** *typQ* :: *'q itself*
  and *G-αw* **and** *G-αg* :: *'g* ⇒ (*'v*::*linorder*) *ugraph*
  +
  assumes *G-invar*[*simp*]: *G-invar g*
begin

sublocale *Prim2 G-αw g G-αg g r* ⟨*proof*⟩

Abstraction of *Q*

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

definition *Q-α Qi* ≡ *enat-of-option o Q-lookup Qi* :: *'v* ⇒ *enat*

lemma *Q-α-empty*: *Q-α Q-empty* = (λ-. ∞)
  ⟨*proof*⟩

lemma *Q-α-update*: *Q-invar Q* ⟹ *Q-α* (*Q-update u d Q*) = (*Q-α Q*)(*u* := *enat d*)
  ⟨*proof*⟩

lemma *Q-α-is-empty*: *Q-invar Q* ⟹ *Q-lookup Q* = *Map.empty* ⟷ *Q-α Q* = (λ-. ∞)
  ⟨*proof*⟩

lemma *Q-α-delete*: *Q-invar Q* ⟹ *Q-α* (*Q-delete u Q*) = (*Q-α Q*)(*u*:=∞)
  ⟨*proof*⟩

lemma *Q-α-min*:
  assumes *MIN*: *Q-getmin Qi* = (*u*, *d*)
  assumes *I*: *Q-invar Qi*
  assumes *NE*: ¬ *Q-is-empty Qi*
  shows *Q-α Qi u* = *enat d* (**is** *?G1*) **and**

$\forall v.$ *enat* $d \le$ *Q-α Qi v* (**is** *?G2*)
⟨*proof*⟩

lemmas *Q-α-specs* = *Q-α-empty Q-α-update Q-α-is-empty Q-α-delete*

Concrete Invariant

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

definition *prim-invar-impl Qi πi* ≡
Q-invar Qi ∧ *M-invar πi* ∧ *prim-invar2* (*Q-α Qi*) (*M-lookup πi*)

end

1.6.3 Refinement of Algorithm

context *Prim-Impl*
begin

lemma *foreach-impl-correct*:
fixes *Qi Vi πi* **defines** *Q* ≡ *Q-α Qi* **and** *π* ≡ *M-lookup πi*
assumes *A*: *foreach-impl Qi πi u* (*G-adj g u*) = (*Qi′*,*πi′*)
assumes *I*: *prim-invar-impl Qi πi*
shows *Q-invar Qi′* **and** *M-invar πi′*
and *Q-α Qi′* = *Qinter Q π u* **and** *M-lookup πi′* = *π′ Q π u*
⟨*proof*⟩⟨*proof*⟩
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*
⟨*proof*⟩

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*⟩

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'*) $\pi i'$
∧ *T-measure-impl* (*Q-delete u Qi'*, $\pi i'$) < *T-measure-impl* (*Qi*,πi)
⟨*proof*⟩

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)
⟨*proof*⟩

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*⟩⟨*proof*⟩
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*
⟨*proof*⟩

lemma *G-from-list-unfold*: *G-from-list* = *G.from-list*
⟨*proof*⟩

lemma [*code*]: *G-from-list l* = *foldr* (λ(*e*, *d*). *G-add-edge e d*) *l G-empty*
⟨*proof*⟩

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
⟨*proof*⟩

global-interpretation *P*: *Prim-Impl-Defs G.invar G.adj G.empty G.add-edge*

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

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

- - - G.αw G.αg g r
for *g* **and** *r::'a::linorder*
defines *prim-impl = P.prim-impl*
and *outer-loop-impl = P.outer-loop-impl*
and *foreach-impl = P.foreach-impl*
and *foreach-impl-body = P.foreach-impl-body*
⟨*proof*⟩

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

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

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

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

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

g r - - - G.αw G.αg **for** *r::'a::linorder*
⟨*proof*⟩

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

end

Adding a Graph-From-List Parser

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

1.6.5 Main Correctness Theorem

The *prim-list-impl* algorithm returns *None*, if the input was invalid. Otherwise it returns *Some* (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.*α*w Gi*) (*component-of* (*G.*α*g Gi*) *r*) (*P.*α*-MST r* πi)) — and represents MST
  ⟨*proof*⟩

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.*α*g* (*G.from-list l*);
      w=*G.*α*w* (*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
  ⟨*proof*⟩

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* ≡ *1*
abbreviation *b* ≡ *2*
abbreviation *c* ≡ *3*
abbreviation *d* ≡ *4*
abbreviation *e* ≡ *5*
abbreviation *f* ≡ *6*
abbreviation *g* ≡ *7*
abbreviation *h* ≡ *8*
abbreviation *i* ≡ *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* :: $'v \Rightarrow$ *enat list* $\Rightarrow$ $'v \Rightarrow$ *bool* **where**
  path u [] *v* $\longleftrightarrow$ *u*=*v*
| *path u* (*l*#*ls*) *v* $\longleftrightarrow$ ($\exists$ *uh*. *l* = *w* (*u*,*uh*) $\wedge$ *path uh ls v*)

lemma *path-append*[*simp*]:
  path u (*ls1*@*ls2*) *v* $\longleftrightarrow$ ($\exists$ *w*. *path u ls1 w* $\wedge$ *path w ls2 v*)
  ⟨*proof*⟩

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

lemma *triv-path*: *path u* [*w* (*u*,*v*)] *v* ⟨*proof*⟩

Shortcut for the set of all paths between two nodes

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

lemma *paths-ne*: *paths u v* $\neq$ {} ⟨*proof*⟩

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

lemma *find-leave-edgeE*:
  assumes *path u p v*
  assumes *u*$\in$*S* *v*$\notin$*S*
  obtains *p1 x y p2*
    where *p* = *p1*@*w* (*x*,*y*)#*p2* *x*$\in$*S* *y*$\notin$*S* *path u p1 x* *path y p2 v*
⟨*proof*⟩

2.1.2 Distance

The (minimum) distance between two nodes u and v is called δ u v.

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

lemma *obtain-shortest-path*:
  obtains *p* **where** *path s p u* δ *s u* = *sum-list p*
  ⟨*proof*⟩

lemma *shortest-path-least*:
  path s p u $\Longrightarrow$ δ *s u* $\leq$ *sum-list p*
  ⟨*proof*⟩

lemma *distance-refl*[*simp*]: δ *s s* = *0*
  ⟨*proof*⟩

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

⟨*proof*⟩

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

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

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

lemma *shortest-path-prefix*:
  assumes $path\ s\ p_1\ x\ path\ x\ p_2\ u$
  and DSU: $\delta\ s\ u = sum\text{-}list\ p_1 + sum\text{-}list\ p_2\ \delta\ s\ u < \infty$
  shows $\delta\ s\ x = sum\text{-}list\ p_1\ \delta\ s\ x < \infty$
⟨*proof*⟩

end

end

2.2 Abstract Datatype for Weighted Directed Graphs

theory *Directed-Graph-Specs*
imports *Directed-Graph*
begin

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

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

end

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

2.2.1 Constructing Weighted Graphs from Lists

lemma *edges-empty*[*simp*]: $WGraph.edges\ (\lambda\text{-}.\ \infty) = \{\}$
⟨*proof*⟩

lemma *edges-insert*[*simp*]:
$WGraph.edges\ (g(e{:=}enat\ d)) = Set.insert\ e\ (WGraph.edges\ g)$
⟨*proof*⟩

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*]:
$valid\text{-}graph\text{-}rep\ l \longleftrightarrow distinct\ (map\ (\lambda(u,v,\text{-}).\ (u,v))\ l)$
⟨*proof*⟩

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

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

lemma *valid-graph-rep-ex1*:
$valid\text{-}graph\text{-}rep\ l \Longrightarrow \exists!\ w.\ \forall\, u\ v\ d.\ w\ (u,v) = enat\ d \longleftrightarrow (u,v,d){\in}set\ l$
⟨*proof*⟩

We define this graph using determinate choice

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

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

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

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

lemma *edges-undef-simps*[*simp*]:
$edges\text{-}undef\ []\ w$
$edges\text{-}undef\ l\ (\lambda\text{-}.\ \infty)$
$edges\text{-}undef\ ((u,v,d)\#l)\ w \longleftrightarrow edges\text{-}undef\ l\ w \wedge w\ (u,v) = \infty$
$edges\text{-}undef\ l\ (w((u,v) := enat\ d)) \longleftrightarrow edges\text{-}undef\ l\ w \wedge (\forall\, d'.\ (u,v,d'){\notin}set\ l)$
⟨*proof*⟩

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

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

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

end

end

2.3 Abstract Dijkstra Algorithm

theory *Dijkstra-Abstract*
imports *Directed-Graph*
begin

2.3.1 Abstract Algorithm

type-synonym *'v 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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*: ‹δ *s u* $\leq$ *D u*› — *D* is a valid estimate
assumes *s-in-S*: ‹*s*$\in$*S* $\vee$ (*D*=(λ-. ∞)(*s*:=*0*) $\wedge$ *S*={})› — The start node is finished, or we are in initial state
assumes *S-precise*: *u*$\in$*S* $\Longrightarrow$ *D u* = δ *s u* — Finished nodes have precise estimate
assumes *S-relaxed*: ‹*v*$\in$*S* $\Longrightarrow$ *D u* $\leq$ δ *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*
⟨*proof*⟩

Relaxing some edges maintains the upper bound property

lemma *maintain-upper-bound*: δ *s u* $\leq$ (*relax-outgoing v D*) *u*
⟨*proof*⟩

Relaxing edges will not affect nodes with already precise estimates

lemma *relax-precise-id*: *D v* = δ *s v* $\Longrightarrow$ *relax-outgoing u D v* = δ *s v*
⟨*proof*⟩

In particular, relaxing edges will not affect finished nodes

lemma *relax-finished-id*: *v*$\in$*S* $\Longrightarrow$ *relax-outgoing u D v* = *D v*
⟨*proof*⟩

The least (finite) estimate among all nodes *u* not in *S* is already precise. This will allow us to add the node *u* to *S*.

lemma *maintain-S-precise-and-connected*:

assumes *UNS*: $u \notin S$
assumes *MIN*: $\forall v.\ v \notin S \longrightarrow D\ u \leq D\ v$
shows $D\ u = \delta\ s\ u$

We start with a case distinction whether we are in the first step of the loop, where we process the start node, or in subsequent steps, where the start node has already been finished.

⟨*proof*⟩

A step of Dijkstra's algorithm maintains the invariant. More precisely, in a step of Dijkstra's algorithm, we pick a node $u \notin S$ with least finite estimate, relax the outgoing edges of u, and add u to S.

theorem *maintain-D-invar*:
assumes *UNS*: $u \notin S$
assumes *UNI*: $D\ u < \infty$
assumes *MIN*: $\forall v.\ v \notin S \longrightarrow D\ u \leq D\ v$
shows *D-invar* (*relax-outgoing u D*) (*Set.insert u S*)
⟨*proof*⟩

When the algorithm is finished, i.e., when there are no unfinished nodes with finite estimates left, then all estimates are accurate.

lemma *invar-finish-imp-correct*:
assumes *F*: $\forall u.\ u \notin S \longrightarrow D\ u = \infty$
shows $D\ u = \delta\ s\ u$
⟨*proof*⟩

A step decreases the set of unfinished nodes.

lemma *unfinished-nodes-decr*:
assumes *UNS*: $u \notin S$
assumes *UNI*: $D\ u < \infty$
shows *unfinished-dnodes* (*Set.insert u S*) ⊂ *unfinished-dnodes S*
⟨*proof*⟩

end

2.3.2 Refinement by Priority Map and Map

In a second step, we implement D and S by a priority map Q and a map V. Both map nodes to finite weights, where Q maps unfinished nodes, and V maps finished nodes.

Note that this implementation is slightly non-standard: In the standard implementation, Q contains also unfinished nodes with infinite weight.

We chose this implementation because it avoids enumerating all nodes of the graph upon initialization of Q. However, on relaxing an edge to a node not in Q, we require an extra lookup to check whether the node is finished.

Implementing *enat* by Option

Our maps are functions to *nat option*,which are interpreted as *enat*, *None* being ∞

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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

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*
⟨*proof*⟩

The concrete version of the invariant.

definition *D-invar' Q V* $\equiv$
$\exists$ *D S*. *coupling Q V D S* $\wedge$ *D-invar D S*

Refinement of *relax*−*outgoing*

definition *relax-outgoing' u du V Q v* $\equiv$
case w (*u*,*v*) *of*

$\infty \Rightarrow Q\ v$
$\mid enat\ d \Rightarrow (case\ Q\ v\ of$
$None \Rightarrow if\ v \in dom\ V\ then\ None\ else\ Some\ (du+d)$
$\mid Some\ d' \Rightarrow Some\ (min\ d'\ (du+d)))$

A step preserves the coupling relation.

lemma (**in** *Dijkstra-Invar*) *coupling-step*:
assumes *C*: *coupling Q V D S*
assumes *UNS*: $u \notin S$
assumes *UNI*: $D\ u = enat\ du$

shows *coupling*
$((relax\text{-}outgoing'\ u\ du\ V\ Q)(u{:=}None))\ (V(u \mapsto du))$
$(relax\text{-}outgoing\ u\ D)\ (Set.insert\ u\ S)$
⟨*proof*⟩

Refinement of initial state

definition $initQ \equiv Map.empty(s \mapsto 0)$
definition $initV \equiv Map.empty$

lemma *coupling-init*:
coupling initQ initV initD initS
⟨*proof*⟩

lemma *coupling-cond*:
assumes *coupling Q V D S*
shows $(Q = Map.empty) \longleftrightarrow (\forall u.\ u \notin S \longrightarrow D\ u = \infty)$
⟨*proof*⟩

Termination argument: Refinement of unfinished nodes.

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

lemma *coupling-unfinished*:
$coupling\ Q\ V\ D\ S \Longrightarrow unfinished\text{-}dnodes'\ V = unfinished\text{-}dnodes\ S$
⟨*proof*⟩

Implementing graph by successor list

definition $relax\text{-}outgoing''\ l\ du\ V\ Q = fold\ (\lambda(d,v)\ Q.$
$case\ Q\ v\ of\ None \Rightarrow if\ v \in dom\ V\ then\ Q\ else\ Q(v \mapsto du+d)$
$\mid Some\ d' \Rightarrow Q(v \mapsto min\ (du+d)\ d'))\ l\ Q$

lemma *relax-outgoing''-refine*:
assumes $set\ l = \{(d,v).\ w\ (u,v) = enat\ d\}$
shows $relax\text{-}outgoing''\ l\ du\ V\ Q = relax\text{-}outgoing'\ u\ du\ V\ Q$
⟨*proof*⟩

end

end

2.4 Weighted Digraph Implementation by Adjacency Map

theory *Directed-Graph-Impl*
imports
  Directed-Graph-Specs
  HOL−Data-Structures.Map-Specs
begin

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

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

definition α :: $'m \Rightarrow ('v)\ wgraph$ **where**
  $\alpha\ g \equiv \lambda(u,v).$ *case M-lookup g u of*
    None $\Rightarrow \infty$
  | *Some l* $\Rightarrow$ *if* $\exists\, d.\ (d,v) \in$ *set l then enat* (*SOME d*. $(d,v) \in$ *set l*) *else* ∞

definition *invar* :: $'m \Rightarrow bool$ **where** *invar g* $\equiv$
    M-invar g
  $\wedge$ ($\forall\, l \in$ *ran* (*M-lookup g*). *distinct* (*map snd l*))
  $\wedge$ *finite* (*WGraph.edges* (α *g*))

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

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

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

sublocale *adt-finite-wgraph invar succ empty-graph add-edge* α
  ⟨*proof*⟩

end

end

2.5 Implementation of Dijkstra's Algorithm

theory *Dijkstra-Impl*

imports
Dijkstra-Abstract
Directed-Graph-Impl
HOL−Library.While-Combinator
Priority-Search-Trees.PST-RBT
HOL−Data-Structures.RBT-Map
begin

2.5.1 Implementation using ADT Interfaces

locale *Dijkstra-Impl-Adts* =
G: *adt-finite-wgraph G-invar G-succ G-empty G-add G-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'*
⟨*proof*⟩

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*⟩

lemma *finite-unfinished-dnodes*[*simp*, *intro*!]: *finite* (*unfinished-dnodes S*)
⟨*proof*⟩

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

definition (**in** *Dijkstra-Impl-Defs*) *Q-relax-outgoing u du V Q* = *fold* (λ(*d*,*v*) *Q*.
case Q-lookup Q v of
None ⇒ *if M-lookup V v* ≠ *None then Q else Q-update v* (*du*+*d*) *Q*
| *Some d'* ⇒ *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*)
∧ *Q-lookup* (*Q-relax-outgoing u du V Q*)
= *relax-outgoing' u du* (*M-lookup V*) (*Q-lookup Q*)
⟨*proof*⟩

definition (**in** *Dijkstra-Impl-Defs*) *D-invar-impl Q V* ≡
Q-invar Q ∧ *M-invar V* ∧ *D-invar'* (*Q-lookup Q*) (*M-lookup V*)

definition (**in** *Dijkstra-Impl-Defs*)
Q-initQ ≡ *Q-update s 0 Q-empty*

lemma *Q-init-Q*[*simp*]:
shows *Q-invar* (*Q-initQ*) *Q-lookup* (*Q-initQ*) = *initQ*
⟨*proof*⟩

definition (**in** *Dijkstra-Impl-Defs*)
M-initV ≡ *M-empty*

lemma *M-initS*[*simp*]: *M-invar M-initV M-lookup M-initV* = *initV*
⟨*proof*⟩

term *Q-getmin*

definition (**in** *Dijkstra-Impl-Defs*)
dijkstra-loop ≡ *while* (λ(*Q*,*V*). ¬ *Q-is-empty Q*) (λ(*Q*,*V*).
let

    $(u,du) = Q\text{-}getmin\ Q;$
    $Q = Q\text{-}relax\text{-}outgoing\ u\ du\ V\ Q;$
    $Q = Q\text{-}delete\ u\ Q;$
    $V = M\text{-}update\ u\ du\ V$
  in
    (Q,V)
) $(Q\text{-}initQ, M\text{-}initV)$

definition (**in** *Dijkstra-Impl-Defs*) *dijkstra* ≡ *snd dijkstra-loop*

lemma *transfer-preconditions*:
  assumes *coupling Q V D S*
  shows $Q\ u = Some\ du \longleftrightarrow D\ u = enat\ du \wedge u \notin S$
  ⟨*proof*⟩

lemma *dijkstra-loop-invar-and-empty*:
  shows *case dijkstra-loop of* $(Q,V) \Rightarrow D\text{-}invar\text{-}impl\ Q\ V \wedge Q\text{-}is\text{-}empty\ Q$
  ⟨*proof*⟩

lemma *dijkstra-correct*:
  M-invar dijkstra
  $M\text{-}lookup\ dijkstra\ u = Some\ d \longleftrightarrow \delta\ s\ u = enat\ d$
  ⟨*proof*⟩

end

2.5.2 Instantiation of ADTs and Code Generation

global-interpretation
  G: *wgraph-by-map RBT-Set.empty RBT-Map.update*
      RBT-Map.delete Lookup2.lookup RBT-Map.M.invar
  defines *G-empty* = *G.empty-graph*
    and *G-add-edge* = *G.add-edge*
    and *G-succ* = *G.succ*
  ⟨*proof*⟩

global-interpretation *Dijkstra-Impl-Adts*
  G.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
  ⟨*proof*⟩

global-interpretation *D*: *Dijkstra-Impl-Defs*
  G.invar G.succ G.empty-graph G.add-edge

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

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

G.α g s **for** *g* **and** *s::'v::linorder*
defines *dijkstra = D.dijkstra*
and *dijkstra-loop = D.dijkstra-loop*
and *Q-relax-outgoing = D.Q-relax-outgoing*
and *M-initV = D.M-initV*
and *Q-initQ = D.Q-initQ*
⟨*proof*⟩

lemmas [*code*] =
D.dijkstra-def D.dijkstra-loop-def

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

interpretation *AUX*: *Dijkstra-Impl*
G.invar G.succ G.empty-graph G.add-edge

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

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

g s G.α **for** *s*
⟨*proof*⟩

lemmas *dijkstra-correct = AUX.dijkstra-correct*[*folded dijkstra-def*]

end

2.5.3 Combination with Graph Parser

We combine the algorithm with a parser from lists to graphs

global-interpretation
G: *wgraph-from-list-algo G.α G.invar G.succ G.empty-graph G.add-edge*
defines *from-list = G.from-list*
⟨*proof*⟩

definition *dijkstra-list l s* ≡
if valid-graph-rep l then Some (*dijkstra* (*from-list l*) *s*) *else None*

theorem *dijkstra-list-correct*:
  case dijkstra-list l s of
    None ⇒ ¬*valid-graph-rep l*
  | *Some D* ⇒
        valid-graph-rep l
      ∧ *M.invar D*
      ∧ (∀ *u d*. *lookup D u* = *Some d* ⟷ *WGraph*.δ (*wgraph-of-list l*) *s u* = *enat d*)
  ⟨*proof*⟩

export-code *dijkstra-list* **checking** *SML OCaml? Scala Haskell?*

value *dijkstra-list* [(*1*::*nat*,*2*,*7*),(*1*,*3*,*1*),(*3*,*2*,*2*)] *1*
value *dijkstra-list* [(*1*::*nat*,*2*,*7*),(*1*,*3*,*1*),(*3*,*2*,*2*)] *3*

end

Bibliography

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2009.

[2] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, Dec 1959.

[3] P. Lammich and T. Nipkow. Proof pearl: Purely functional, simple and efficient Priority Search Trees and applications to Prim and Dijkstra. In *Proc. of ITP*, 2019. to appear.

[4] R. C. Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, Nov 1957.