Dijkstra's Algorithm

Benedikt Nordhoff Peter Lammich

March 12, 2013

Abstract

We implement and prove correct Dijkstra's algorithm for the single source shortest path problem, conceived in 1956 by E. Dijkstra. The algorithm is implemented using the data refinement framework for monadic, nondeterministic programs. An efficient implementation is derived using data structures from the Isabelle Collection Framework.

Contents

1 Introduction and Overview

Dijkstra's algorithm [1] is an algorithm used to find shortest paths from one given vertex to all other vertices in a non-negatively weighted graph.

The implementation of the algorithm is meant to be an application of our extensions to the Isabelle Collections Framework (ICF) [4, 6, 7]. Moreover, it serves as a test case for our data refinement framework [5]. We use ICF-Maps to efficiently represent the graph and result and the newly introduced unique priority queues for the work list.

For a documentation of the refinement framework see [5], that also contains a userguide and some simpler examples.

The development utilizes a stepwise refinement approach. Starting from an abstract algorithm that has a nice correctness proof, we stepwise refine the algorithm until we end up with an efficient implementation, for that we generate code using Isabelle/HOL's code generator[2, 3].

Structure of the Submission. The abstract version of the algorithm with the correctness proof, as well as the main refinement steps are contained in the theory `Dijkstra`. The refinement steps involving the ICF and code generation are contained in `Dijkstra-Impl`. The theory `Infty` contains an extension of numbers with an infinity element. The theory `Graph` contains a formalization of graphs, paths, and related concepts. The theories `GraphSpec,GraphGA,GraphByMap,HashGraphImpl` contain an ICF-style specification of graphs. The theory `Test` contains a small performance test on random graphs. It uses the ML-code generated by the code generator.

2 Miscellaneous Lemmas

theory *Dijkstra-Misc*
imports *Main*
begin
  inductive-set *least-map* **for** f S **where**
    $[\![x{\in}S; \forall x'{\in}S.\ f\ x \leq f\ x']\!] \Longrightarrow x \in$ *least-map* $f\ S$

  lemma *least-map-subset*: *least-map* $f\ S \subseteq S$
    by (*auto elim*: *least-map.cases*)

  lemmas *least-map-elemD* = *set-mp*[*OF least-map-subset*]

  lemma *least-map-leD*:
    assumes $x \in$ *least-map* $f\ S$
    assumes $y{\in}S$
    shows $f\ x \leq f\ y$
    using *assms*

by (*auto elim*: *least-map.cases*)

lemma *least-map-empty*[*simp*]: *least-map f* {} = {}
by (*auto elim*: *least-map.cases*)

lemma *least-map-singleton*[*simp*]: *least-map* (*f*::′*a*⇒′*b*::*order*) {*x*} = {*x*}
by (*auto elim*: *least-map.cases intro*!: *least-map.intros simp*: *refl*)

lemma *least-map-insert-min*:
fixes *f*::′*a*⇒′*b*::*order*
assumes ∀ *y*∈*S*. *f x* ≤ *f y*
shows *x* ∈ *least-map f* (*insert x S*)
using *assms* **by** (*auto intro*: *least-map.intros*)

lemma *least-map-insert-nmin*:
⟦ *x*∈*least-map f S*; *f x* ≤ *f a* ⟧ ⟹ *x*∈*least-map f* (*insert a S*)
by (*auto elim*: *least-map.cases intro*: *least-map.intros*)

context *semilattice-inf*
begin
lemmas [*simp*] = *inf-absorb1 inf-absorb2*

lemma *inf-absorb-less*[*simp*]:
a < *b* ⟹ *inf a b* = *a*
a < *b* ⟹ *inf b a* = *a*
apply (*metis le-iff-inf less-imp-le*)
by (*metis inf-commute le-iff-inf less-imp-le*)
end

end

3 Graphs

theory *Graph*
imports *Main*
begin

This theory defines a notion of graphs. A graph is a record that contains a set of nodes V and a set of labeled edges $E \subseteq V \times W \times V$, where W are the edge labels.

3.1 Definitions

A graph is represented by a record.

record $('v,'w)$ *graph* =
  nodes :: $'v$ *set*
  edges :: $('v \times 'w \times 'v)$ *set*

In a valid graph, edges only go from nodes to nodes.

locale *valid-graph* =
  fixes G :: $('v,'w)$ *graph*
  assumes *E-valid*: *fst*‘*edges* $G \subseteq$ *nodes* G
    snd‘*snd*‘*edges* $G \subseteq$ *nodes* G
begin
  abbreviation $V \equiv$ *nodes* G
  abbreviation $E \equiv$ *edges* G

  lemma *E-validD*: **assumes** $(v,e,v') \in E$
    shows $v \in V$ $v' \in V$
    apply −
    apply (*rule set-mp*[*OF E-valid*(*1*)])
    using *assms* **apply** *force*
    apply (*rule set-mp*[*OF E-valid*(*2*)])
    using *assms* **apply** *force*
    done

end

3.2 Basic operations on Graphs

The empty graph.

definition *empty* **where**
  empty ≡ (| *nodes* = {}, *edges* = {} |)

Adds a node to a graph.

definition *add-node* **where**
  add-node v g ≡ (| *nodes* = *insert v* (*nodes g*), *edges*=*edges g*|)

Deletes a node from a graph. Also deletes all adjacent edges.

definition *delete-node* **where** *delete-node v g* ≡ (|
  nodes = *nodes g* − {*v*},
  edges = *edges g* ∩ (−{*v*})×*UNIV*×(−{*v*})
  |)

Adds an edge to a graph.

definition *add-edge* **where** *add-edge v e v' g* ≡ (|
  nodes = {*v*,*v'*} ∪ *nodes g*,
  edges = *insert* (*v*,*e*,*v'*) (*edges g*)

⦈

Deletes an edge from a graph.

definition *delete-edge* **where** *delete-edge v e v′ g* ≡ ⦇
nodes = *nodes g*, *edges* = *edges g* − {(*v*,*e*,*v′*)} ⦈

Successors of a node.

definition *succ* :: (′*v*,′*w*) *graph* ⇒ ′*v* ⇒ (′*w*×′*v*) *set*
where *succ G v* ≡ {(*w*,*v′*). (*v*,*w*,*v′*)∈*edges G*}

Now follow some simplification lemmas.

lemma *empty-valid*[*simp*]: *valid-graph empty*
unfolding *empty-def* **by** *unfold-locales auto*
lemma *add-node-valid*[*simp*]: **assumes** *valid-graph g*
shows *valid-graph* (*add-node v g*)
proof −
interpret *valid-graph g* **by** *fact*
show *?thesis*
unfolding *add-node-def*
by *unfold-locales* (*auto dest*: *E-validD*)
qed
lemma *delete-node-valid*[*simp*]: **assumes** *valid-graph g*
shows *valid-graph* (*delete-node v g*)
proof −
interpret *valid-graph g* **by** *fact*
show *?thesis*
unfolding *delete-node-def*
by *unfold-locales* (*auto dest*: *E-validD*)
qed
lemma *add-edge-valid*[*simp*]: **assumes** *valid-graph g*
shows *valid-graph* (*add-edge v e v′ g*)
proof −
interpret *valid-graph g* **by** *fact*
show *?thesis*
unfolding *add-edge-def*
by *unfold-locales* (*auto dest*: *E-validD*)
qed
lemma *delete-edge-valid*[*simp*]: **assumes** *valid-graph g*
shows *valid-graph* (*delete-edge v e v′ g*)
proof −
interpret *valid-graph g* **by** *fact*
show *?thesis*
unfolding *delete-edge-def*
by *unfold-locales* (*auto dest*: *E-validD*)
qed

lemma *succ-finite*[*simp*, *intro*]: *finite* (*edges G*) ⟹ *finite* (*succ G v*)
unfolding *succ-def*
by (*rule finite-subset*[**where** *B*=*snd*‘*edges G*]) *force*+

lemma *nodes-empty*[*simp*]: *nodes empty* = {} **unfolding** *empty-def* **by** *simp*
lemma *edges-empty*[*simp*]: *edges empty* = {} **unfolding** *empty-def* **by** *simp*
lemma *succ-empty*[*simp*]: *succ empty v* = {} **unfolding** *empty-def succ-def* **by** *auto*

lemma *nodes-add-node*[*simp*]: *nodes* (*add-node v g*) = *insert v* (*nodes g*)
by (*simp add*: *add-node-def*)
lemma *nodes-add-edge*[*simp*]:
nodes (*add-edge v e v′ g*) = *insert v* (*insert v′* (*nodes g*))
by (*simp add*: *add-edge-def*)
lemma *edges-add-edge*[*simp*]:
edges (*add-edge v e v′ g*) = *insert* (*v*,*e*,*v′*) (*edges g*)
by (*simp add*: *add-edge-def*)
lemma *edges-add-node*[*simp*]:
edges (*add-node v g*) = *edges g*
by (*simp add*: *add-node-def*)

lemma (**in** *valid-graph*) *succ-subset*: *succ G v* ⊆ *UNIV*×*V*
unfolding *succ-def* **using** *E-valid*
by (*force*)

3.3 Paths

A path is represented by a list of adjacent edges.

type-synonym (*′v*,*′w*) *path* = (*′v*×*′w*×*′v*) *list*

context *valid-graph*
begin

The following predicate describes a valid path:

fun *is-path* :: *′v* ⇒ (*′v*,*′w*) *path* ⇒ *′v* ⇒ *bool* **where**
is-path v [] *v′* ⟷ *v*=*v′* ∧ *v′*∈*V* |
is-path v ((*v1*,*w*,*v2*)#*p*) *v′* ⟷ *v*=*v1* ∧ (*v1*,*w*,*v2*)∈*E* ∧ *is-path v2 p v′*

lemma *is-path-simps*[*simp*, *intro*!]:
is-path v [] *v* ⟷ *v*∈*V*
is-path v [(*v*,*w*,*v′*)] *v′* ⟷ (*v*,*w*,*v′*)∈*E*
by (*auto dest*: *E-validD*)

lemma *is-path-memb*[*simp*]:
is-path v p v′ ⟹ *v*∈*V* ∧ *v′*∈*V*
apply (*induct p arbitrary*: *v*)
apply (*auto dest*: *E-validD*)
done

lemma *is-path-split*:
is-path v (*p1*@*p2*) *v′* ⟷ (∃ *u*. *is-path v p1 u* ∧ *is-path u p2 v′*)
by (*induct p1 arbitrary*: *v*) *auto*

```
  lemma is-path-split'[simp]:
    is-path v (p1@(u,w,u')#p2) v'
      ⟷ is-path v p1 u ∧ (u,w,u')∈E ∧ is-path u' p2 v'
    by (auto simp add: is-path-split)
end
```

Set of intermediate vertices of a path. These are all vertices but the last one. Note that, if the last vertex also occurs earlier on the path, it is contained in *int-vertices*.

```
  definition int-vertices :: ('v,'w) path ⇒ 'v set where
    int-vertices p ≡ set (map fst p)

  lemma int-vertices-simps[simp]:
    int-vertices [] = {}
    int-vertices (vv#p) = insert (fst vv) (int-vertices p)
    int-vertices (p1@p2) = int-vertices p1 ∪ int-vertices p2
    by (auto simp add: int-vertices-def)

  lemma (in valid-graph) int-vertices-subset:
    is-path v p v' ⟹ int-vertices p ⊆ V
    apply (induct p arbitrary: v)
    apply (simp)
    apply (force dest: E-validD)
    done

  lemma int-vertices-empty[simp]: int-vertices p = {} ⟷ p=[]
    by (cases p) auto
```

3.3.1 Splitting Paths

Split a path at the point where it first leaves the set W:

```
  lemma (in valid-graph) path-split-set:
    assumes is-path v p v' and v∈W and v'∉W
    obtains p1 p2 u w u' where
    p=p1@(u,w,u')#p2 and
    int-vertices p1 ⊆ W and u∈W and u'∉W
    using assms
  proof (induct p arbitrary: v thesis)
    case Nil thus ?case by auto
  next
    case (Cons vv p)
    note [simp, intro!] = ‹v∈W› ‹v'∉W›
    from Cons.prems obtain w u' where
      [simp]: vv=(v,w,u') and
        REST: is-path u' p v'
      by (cases vv) auto
```

Distinguish wether the second node u' of the path is in W. If yes, the proposition

follows by the induction hypothesis, otherwise it is straightforward, as the split takes place at the first edge of the path.

```
  {
    assume A [simp, intro!]: u'∈W
    from Cons.hyps[OF - REST] obtain p1 uu ww uu' p2 where
      p=p1@(uu,ww,uu')#p2 int-vertices p1 ⊆ W uu ∈ W uu' ∉ W
      by blast
    with Cons.prems(1)[of vv#p1 uu ww uu' p2] have thesis by auto
  } moreover {
    assume u'∉W
    with Cons.prems(1)[of [] v w u' p] have thesis by auto
  } ultimately show thesis by blast
qed
```

Split a path at the point where it first enters the set W:

```
lemma (in valid-graph) path-split-set':
  assumes is-path v p v' and v'∈W
  obtains p1 p2 u where
  p=p1@p2 and
  is-path v p1 u and
  is-path u p2 v' and
  int-vertices p1 ⊆ -W and u∈W
  using assms
proof (cases v∈W)
  case True with that[of [] p] assms show ?thesis
    by auto
next
  case False with assms that show ?thesis
  proof (induct p arbitrary: v thesis)
    case Nil thus ?case by auto
  next
    case (Cons vv p)
    note [simp, intro!] = ‹v'∈W› ‹v∉W›
    from Cons.prems obtain w u' where
      [simp]: vv=(v,w,u') and [simp]: (v,w,u')∈E and
        REST: is-path u' p v'
      by (cases vv) auto
```

Distinguish wether the second node u' of the path is in W. If yes, the proposition is straightforward, otherwise, it follows by the induction hypothesis.

```
    {
      assume A [simp, intro!]: u'∈W
      from Cons.prems(3)[of [vv] p u'] REST have ?case by auto
    } moreover {
      assume [simp, intro!]: u'∉W
      from Cons.hyps[OF REST] obtain p1 p2 u'' where
        [simp]: p=p1@p2 and
          is-path u' p1 u'' and
          is-path u'' p2 v' and
```

```
      int-vertices p1 ⊆ − W and
      u''∈W by blast
    with Cons.prems(3)[of vv#p1] have ?case by auto
  } ultimately show ?case by blast
 qed
qed
```

Split a path at the point where a given vertex is first visited:

```
lemma (in valid-graph) path-split-vertex:
 assumes is-path v p v' and u∈int-vertices p
 obtains p1 p2 where
 p=p1@p2 and
 is-path v p1 u and
 u ∉ int-vertices p1
 using assms
proof (induct p arbitrary: v thesis)
 case Nil thus ?case by auto
next
 case (Cons vv p)
 from Cons.prems obtain w u' where
   [simp]: vv=(v,w,u') v∈V (v,w,u')∈E and
     REST: is-path u' p v'
   by (cases vv) auto

 {
   assume u=v
   with Cons.prems(1)[of [] vv#p] have thesis by auto
 } moreover {
   assume [simp]: u≠v
   with Cons.hyps(1)[OF - REST] Cons.prems(3) obtain p1 p2 where
     p=p1@p2 is-path u' p1 u u∉int-vertices p1
     by auto
   with Cons.prems(1)[of vv#p1 p2] have thesis
     by auto
 } ultimately show ?case by blast
qed
```

3.4 Weighted Graphs

```
locale valid-mgraph = valid-graph G for G::('v,'w::monoid-add) graph

definition path-weight :: ('v,'w::monoid-add) path ⇒ 'w
 where path-weight p ≡ listsum (map (fst ∘ snd) p)



lemma path-weight-split[simp]:
 (path-weight (p1@p2)::'w::monoid-add) = path-weight p1 + path-weight p2
 unfolding path-weight-def
```

by (*auto*)

lemma *path-weight-empty*[*simp*]: *path-weight* [] = *0*
unfolding *path-weight-def*
by *auto*

lemma *path-weight-cons*[*simp*]:
(*path-weight* (*e*#*p*)::′*w*::*monoid-add*) = *fst* (*snd e*) + *path-weight p*
unfolding *path-weight-def*
by (*auto*)

end

4 Weights for Dijkstra's Algorithm

theory *Weight*
imports *Complex-Main*
begin

In this theory, we set up a type class for weights, and a typeclass for weights with an infinity element. The latter one is used internally in Dijkstra's algorithm.

Moreover, we provide a datatype that adds an infinity element to a given base type.

4.1 Type Classes Setup

class *weight* = *ordered-ab-semigroup-add* + *comm-monoid-add* + *linorder*
begin

lemma *add-nonneg-nonneg* [*simp*]:
assumes *0* ≤ *a* **and** *0* ≤ *b* **shows** *0* ≤ *a* + *b*
proof −
have *0* + *0* ≤ *a* + *b*
using *assms* **by** (*rule add-mono*)
then show *?thesis* **by** *simp*
qed

lemma *add-nonpos-nonpos*[*simp*]:
assumes *a* ≤ *0* **and** *b* ≤ *0* **shows** *a* + *b* ≤ *0*
proof −
have *a* + *b* ≤ *0* + *0*
using *assms* **by** (*rule add-mono*)
then show *?thesis* **by** *simp*
qed

lemma *add-nonneg-eq-0-iff*:

assumes x: $0 \leq x$ **and** y: $0 \leq y$
shows $x + y = 0 \longleftrightarrow x = 0 \wedge y = 0$
by (*metis add.comm-neutral add.left-neutral add-left-mono antisym x y*)

lemma *add-incr*: $0 \leq b \Longrightarrow a \leq a+b$
by (*metis add.comm-neutral add-left-mono*)

lemma *add-incr-left*[*simp*, *intro*!]: $0 \leq b \Longrightarrow a \leq b + a$
by (*metis add-incr add.commute*)

lemma *sum-not-less*[*simp*, *intro*!]:
$0 \leq b \Longrightarrow \neg\ (a+b < a)$
$0 \leq a \Longrightarrow \neg\ (a+b < b)$
apply (*metis add-incr less-le-not-le*)
apply (*metis add-incr-left less-le-not-le*)
done

end

instance *nat* :: *weight* **..**
instance *int* :: *weight* **..**
instance *rat* :: *weight* **..**
instance *real* :: *weight* **..**

class *top-weight* = *top* + *weight* +
assumes *inf-add-right*[*simp*]: *a* + *top* = *top*
begin

lemma *inf-add-left*[*simp*]: *top* + *a* = *top*
by (*metis add-commute inf-add-right*)

lemmas [*simp*] = *top-unique less-top*[*symmetric*]

lemma *not-less-inf*[*simp*]:
$\neg\ (a < top) \longleftrightarrow a{=}top$
by *simp*

end

4.2 Adding Infinity

We provide a standard way to add an infinity element to any type.

datatype *'a infty* = *Infty* | *Num 'a*
primrec *val* **where** *val* (*Num d*) = *d*

lemma *num-val-iff*[*simp*]: $e{\neq}Infty \Longrightarrow Num\ (val\ e) = e$ **by** (*cases e*) *auto*

type-synonym *NatB* = *nat infty*

```
instantiation infty :: (weight) top-weight
begin
  definition (0::'a infty) == Num 0
  definition top ≡ Infty

  fun less-eq-infty where
    less-eq Infty (Num -) ⟷ False |
    less-eq - Infty ⟷ True |
    less-eq (Num a) (Num b) ⟷ a≤b

  lemma [simp]: Infty≤a ⟷ a=Infty
    by (cases a) auto

  fun less-infty where
    less Infty - ⟷ False |
    less (Num -) Infty ⟷ True |
    less (Num a) (Num b) ⟷ a<b

  lemma [simp]: less a Infty ⟷ a ≠ Infty
    by (cases a) auto

  fun plus-infty where
    plus - Infty = Infty |
    plus Infty - = Infty |
    plus (Num a) (Num b) = Num (a+b)

  lemma [simp]: plus Infty a = Infty by (cases a) simp-all


  instance
    apply (intro-classes)
    apply (case-tac [!] x) [4]
    apply simp-all
    apply (case-tac [!] y) [3]
    apply (simp-all add: less-le-not-le)
    apply (case-tac z)
    apply (simp-all add: top-infty-def zero-infty-def)
    apply (case-tac [!] a) [4]
    apply simp-all
    apply (case-tac [!] b) [3]
    apply (simp-all add: add-ac)
    apply (case-tac [!] c) [2]
    apply (simp-all add: add-ac add-right-mono)
    apply (case-tac (x,y) rule: less-eq-infty.cases)
    apply (simp-all add: linear)
    done
end
```

4.2.1 Unboxing

Conversion between the constants defined by the typeclass, and the concrete functions on the *'a infty* type.

lemma *infty-inf-unbox*:
Num a ≠ *top*
top ≠ *Num a*
Infty = *top*
by (*auto simp add*: *top-infty-def*)

lemma *infty-ord-unbox*:
Num a ≤ *Num b* ⟷ *a* ≤ *b*
Num a < *Num b* ⟷ *a* < *b*
by *auto*

lemma *infty-plus-unbox*:
Num a + *Num b* = *Num* (*a*+*b*)
by (*auto*)

lemma *infty-zero-unbox*:
Num a = *0* ⟷ *a* = *0*
Num 0 = *0*
by (*auto simp*: *zero-infty-def*)

lemmas *infty-unbox* =
infty-inf-unbox infty-zero-unbox infty-ord-unbox infty-plus-unbox

lemma *inf-not-zero*[*simp*]:
top≠(*0*::- *infty*) (*0*::- *infty*)≠*top*
apply (*unfold zero-infty-def top-infty-def*)
apply *auto*
done

lemma *num-val-iff'*[*simp*]: *e*≠*top* ⟹ *Num* (*val e*) = *e*
by (*cases e*) (*auto simp add*: *infty-unbox*)

lemma *infty-neE*:
⟦*a*≠*Infty*; ⋀*d*. *a*=*Num d* ⟹ *P*⟧ ⟹ *P*
⟦*a*≠*top*; ⋀*d*. *a*=*Num d* ⟹ *P*⟧ ⟹ *P*
by (*case-tac* [!] *a*) (*auto simp add*: *infty-unbox*)

end

5 Dijkstra's Algorithm

theory *Dijkstra*
imports *Graph Dijkstra-Misc*

../Refine-Monadic/Refine ../Refine-Monadic/Collection-Bindings Weight
begin

This theory defines Dijkstra's algorithm. First, a correct result of Dijkstra's algorithm w.r.t. a graph and a start vertex is specified. Then, the refinement framework is used to specify Dijkstra's Algorithm, prove it correct, and finally refine it to datatypes that are closer to an implementation than the original specification.

5.1 Graph's for Dijkstra's Algorithm

A graph annotated with weights.

locale *weighted-graph* = *valid-graph G*
for $G :: ('V,'W::weight)$ *graph*

5.2 Specification of Correct Result

context *weighted-graph*
begin

A result of Dijkstra's algorithm is correct, if it is a map from nodes v to the shortest path from the start node *v0* to v. Iff there is no such path, the node is not in the map.

definition *is-shortest-path-map* :: $'V \Rightarrow ('V \rightharpoonup ('V,'W)\ path) \Rightarrow bool$
where
is-shortest-path-map v0 res $\equiv \forall v \in V.$ (*case res v of*
None $\Rightarrow \neg(\exists p.$ *is-path v0 p v*) |
Some p $\Rightarrow$ *is-path v0 p v*
$\wedge\ (\forall p'.$ *is-path v0 p' v* $\longrightarrow$ *path-weight p* $\leq$ *path-weight p'*)
)
end

The following function returns the weight of an optional path, where *None* is interpreted as infinity.

fun *path-weight'* **where**
path-weight' None = *top* |
path-weight' (Some p) = *Num* (*path-weight p*)

5.3 Dijkstra's Algorithm

The state in the main loop of the algorithm consists of a workset *wl* of vertexes that still need to be explored, and a map *res* that contains the current shortest path for each vertex.

type-synonym $('V,'W)$ *state* = $('V\ set) \times ('V \rightharpoonup ('V,'W)\ path)$

The preconditions of Dijkstra's algorithm, i.e., that it operates on a valid and finite graph, and that the start node is a node of the graph, are summarized in a locale.

locale *Dijkstra* = *weighted-graph G*
for *G* :: ('*V*,'*W*::*weight*) *graph*+
fixes *v0* :: '*V*
assumes *finite*[*simp*,*intro*!]: *finite V finite E*
assumes *v0-in-V*[*simp*, *intro*!]: *v0*∈*V*
assumes *nonneg-weights*[*simp*, *intro*]: (*v*,*w*,*v'*)∈*edges G* ⟹ *0*≤*w*
begin

Paths have non-negative weights.

lemma *path-nonneg-weight*: *is-path v p v'* ⟹ *0* ≤ *path-weight p*
by (*induct rule*: *is-path.induct*) *auto*

Invariant of the main loop:

- The workset only contains nodes of the graph.
- If the result set contains a path for a node, it is actually a path, and uses only intermediate vertices outside the workset.
- For all vertices outside the workset, the result map contains the shortest path.
- For all vertices in the workset, the result map contains the shortest path among all paths that only use intermediate vertices outside the workset.

definition *dinvar* σ ≡ *let* (*wl*,*res*)=σ *in*
wl ⊆ *V* ∧
(∀ *v*∈*V*. ∀ *p*. *res v* = *Some p* ⟶ *is-path v0 p v* ∧ *int-vertices p* ⊆ *V*−*wl*) ∧
(∀ *v*∈*V*−*wl*. ∀ *p*. *is-path v0 p v*
⟶ *path-weight'* (*res v*) ≤ *path-weight'* (*Some p*)) ∧
(∀ *v*∈*wl*. ∀ *p*. *is-path v0 p v* ∧ *int-vertices p* ⊆ *V*−*wl*
⟶ *path-weight'* (*res v*) ≤ *path-weight'* (*Some p*)
)

Sanity check: The invariant is strong enough to imply correctness of result.

lemma *invar-imp-correct*: *dinvar* ({},*res*) ⟹ *is-shortest-path-map v0 res*
unfolding *dinvar-def is-shortest-path-map-def*
by (*auto simp*: *infty-unbox split*: *option.split*)

The initial workset contains all vertices. The initial result maps *v0* to the empty path, and all other vertices to *None*.

definition *dinit* :: ('*V*,'*W*) *state nres* **where**

```
dinit ≡ SPEC ( λ(wl,res) .
    wl=V ∧ res v0 = Some [] ∧ (∀ v∈V−{v0}. res v = None))
```

The initial state satisfies the invariant.

```
lemma dinit-invar: dinit ≤ SPEC dinvar
  unfolding dinit-def
  apply (intro refine-vcg)
  apply (force simp: dinvar-def split: option.split)
  done
```

In each iteration, the main loop of the algorithm pops a minimal node from the workset, and then updates the result map accordingly.

Pop a minimal node from the workset. The node is minimal in the sense that the length of the current path for that node is minimal.

```
definition pop-min :: ('V,'W) state ⇒ ('V × ('V,'W) state) nres where
  pop-min σ ≡ do {
    let (wl,res)=σ;
    ASSERT (wl≠{});
    v ← RES (least-map (path-weight' ∘ res) wl);
    RETURN (v,(wl−{v},res))
  }
```

Updating the result according to a node v is done by checking, for each successor node, whether the path over v is shorter than the path currently stored into the result map.

```
inductive update-spec :: 'V ⇒ ('V,'W) state ⇒ ('V,'W) state ⇒ bool
  where
  ⟦ ∀ v'∈V.
    res' v' ∈ least-map path-weight' (
      { res v' } ∪ { Some (p@[(v,w,v')]) | p w. res v = Some p ∧ (v,w,v')∈E }
    )
   ⟧ ⟹ update-spec v (wl,res) (wl,res')
```

In order to ease the refinement proof, we will assert the following precondition for updating.

```
definition update-pre :: 'V ⇒ ('V,'W) state ⇒ bool where
  update-pre v σ ≡ let (wl,res)=σ in v∈V
    ∧ (∀ v'∈V−wl. v'≠v ⟶ (∀ p. is-path v0 p v'
        ⟶ path-weight' (res v') ≤ path-weight' (Some p)))
    ∧ (∀ v'∈V. ∀ p. res v' = Some p ⟶ is-path v0 p v')

definition update :: 'V ⇒ ('V,'W) state ⇒ ('V,'W) state nres where
  update v σ ≡ do {ASSERT (update-pre v σ); SPEC (update-spec v σ)}
```

Finally, we define Dijkstra's algorithm:

```
definition dijkstra where
  dijkstra ≡ do {
```

```
  σ0←dinit;
  (-,res) ← WHILE_T^dinvar (λ(wl,-). wl≠{})
      (λσ.
        do { (v,σ') ← pop-min σ; update v σ' }
      )
      σ0;
  RETURN res }
```

The following theorem states (total) correctness of Dijkstra's algorithm.

```
theorem dijkstra-correct: dijkstra ≤ SPEC (is-shortest-path-map v0)
  unfolding dijkstra-def
  unfolding dinit-def
  unfolding pop-min-def update-def [abs-def]
  apply (intro
    WHILEIT-rule[where R=inv-image {(x,y). x<y} (card ∘ fst)]
    refine-vcg conjI)
  apply (simp-all split: prod.split-asm)
proof −
  fix wl res v
  assume INV: dinvar (wl,res)
  and LM: v∈least-map (path-weight' ∘ res) wl
  hence v∈V unfolding dinvar-def by (auto dest: least-map-elemD)
  moreover
  from INV have ∀ v'∈V − (wl−{v}). v' ≠ v ⟶
    (∀ p. is-path v0 p v' ⟶ path-weight' (res v') ≤ Num (path-weight p))
    by (auto simp: dinvar-def)
  moreover from INV have ∀ v'∈V. ∀ p. res v'=Some p ⟶ is-path v0 p v'
    by (auto simp: dinvar-def)
  ultimately show update-pre v (wl−{v},res) by (auto simp: update-pre-def)
next
  fix res
  assume dinvar ({}, res)
  thus is-shortest-path-map v0 res
    by (rule invar-imp-correct)
next
  show wf (inv-image {(x, y). x < y} (card ∘ fst))
    by (blast intro: wf-less)
next
  fix wl res v σ''
  assume
    LM: v∈least-map (path-weight' ∘ res) wl and
    UD: update-spec v (wl−{v},res) σ'' and
    INV: dinvar (wl,res)

  from LM have v∈wl by (auto dest: least-map-elemD)
  moreover from UD have fst σ'' = wl−{v} by (auto elim: update-spec.cases)
  moreover from INV have finite wl
    unfolding dinvar-def by (auto dest: finite-subset)
```

```
  ultimately show card (fst σ'') < card wl
    apply simp
    by (metis card-gt-0-iff diff-Suc-less empty-iff)
next
  fix a and res :: 'V ⇀ ('V,'W) path
  assume a = V ∧ res v0 = Some [] ∧ (∀ v∈V−{v0}. res v = None)
  thus dinvar (V,res)
    by (force simp: dinvar-def split: option.split)
next
  fix wl res
  assume INV: dinvar (wl,res)
  hence
    WL-SUBSET: wl ⊆ V and
    PATH-VALID: ∀ v∈V. ∀ p. res v = Some p
      ⟶ is-path v0 p v ∧ int-vertices p ⊆ V − wl and
    NWL-MIN: ∀ v∈V − wl. ∀ p. is-path v0 p v
      ⟶ path-weight' (res v) ≤ Num (path-weight p) and
    WL-MIN: ∀ v∈wl. ∀ p. is-path v0 p v ∧ int-vertices p ⊆ V − wl
      ⟶ path-weight' (res v) ≤ Num (path-weight p)
    unfolding dinvar-def by auto

  fix v σ''
  assume V-LEAST: v∈least-map (path-weight' o res) wl
    and update-spec v (wl−{v},res) σ''
  then obtain res' where
    [simp]: σ''=(wl−{v},res')
   and CONSIDERED-NEW-PATHS: ∀ v'∈V. res' v' ∈ least-map path-weight'
      (insert (res v')
            ({ Some (p@[(v,w,v')]) | p w. res v = Some p ∧ (v,w,v')∈E }))
    by (auto elim!: update-spec.cases)

  from V-LEAST have V-MEM: v∈wl by (blast intro: least-map-elemD)

  show dinvar σ''
    apply (unfold dinvar-def, simp)
    apply (intro conjI)
  proof −
    from WL-SUBSET show wl−{v} ⊆ V by auto

    show ∀ va∈V. ∀ p. res' va = Some p
      ⟶ is-path v0 p va ∧ int-vertices p ⊆ V − (wl − {v})
    proof (intro ballI conjI impI allI)
      fix v' p
      assume V'-MEM: v'∈V and [simp]: res' v' = Some p
```

The new paths that we have added are valid and only use intermediate vertices outside the workset.

This proof works as follows: A path *res' v'* is either the old path, or has been assembled as a path over node *v*. In the former case the proposition follows straightforwardly from the invariant for the old state. In the latter case we get, by the

invariant for the old state, that the path over node v is valid. Then, we observe that appending an edge to a valid path yields a valid path again. Also, adding v as intermediate node is legal, as we just removed v from the workset.

```
with CONSIDERED-NEW-PATHS have res' v' ∈ (insert (res v')
  ({ Some (p@[(v,w,v')]) | p w. res v = Some p ∧ (v,w,v')∈E }))
  by (rule-tac least-map-elemD) blast
moreover {
  assume [symmetric,simp]: res' v' = res v'
  from V'-MEM PATH-VALID have
    is-path v0 p v'
    int-vertices p ⊆ V − (wl−{v})
    by force+
} moreover {
  fix pv w
  assume res' v' = Some (pv@[(v,w,v')])
    and [simp]: res v = Some pv
    and EDGE: (v,w,v')∈E
  hence [simp]: p = pv@[(v,w,v')] by simp

  from bspec[OF PATH-VALID set-rev-mp[OF V-MEM WL-SUBSET]] have

    PATHV: is-path v0 pv v and IVV: int-vertices pv ⊆ V − wl by auto
  hence
    is-path v0 p v'
    int-vertices p ⊆ V − (wl−{v})
    by (auto simp: EDGE V'-MEM)
}
ultimately show
  is-path v0 p v'
  int-vertices p ⊆ V − (wl−{v})
  by blast+
qed
```

We show that already the *original* result stores the minimal path for all vertices not in the *new* workset. For vertices also not in the original workset, this follows straightforwardly from the invariant.

For the vertex v, that has been removed from the workset, we split a path p' to v at the point u where it first enters the original workset.

As we chose v to be the vertex in the workset with the minimal weight, its weight is less than the current weight of u. As the vertices of the prefix of p' up to u are not in the workset, the current weight of u is less than the weight of the prefix of p', and thus less than the weight of p'. Together, the current weight of v is less than the weight of p'.

```
have RES-MIN: ∀ v∈V − (wl − {v}). ∀ p. is-path v0 p v
  ⟶ path-weight' (res v) ≤ Num (path-weight p)
proof (intro ballI allI impI)
  fix v' p'
  assume NOT-IN-WL: v' ∈ V − (wl − {v})
    and PATH: is-path v0 p' v'
```

```
  hence [simp, intro!]: v'∈V by auto

  show path-weight' (res v') ≤ Num (path-weight p')
  proof (cases v' = v)
    assume NE[simp]: v'≠v
    from bspec[OF NWL-MIN, of v'] NOT-IN-WL PATH show
      path-weight' (res v') ≤ Num (path-weight p') by auto
  next
    assume EQ[simp]: v'=v

    from path-split-set'[OF PATH, of wl] V-MEM obtain p1 p2 u where
      [simp]: p'=p1@p2
        and P1: is-path v0 p1 u
        and P2: is-path u p2 v'
        and P1V: int-vertices p1 ⊆ -wl
        and [simp]: u∈wl
      by auto

    from least-map-leD[OF V-LEAST]
    have path-weight' (res v') ≤ path-weight' (res u)by auto
    also from bspec[OF WL-MIN, of u] P1 P1V int-vertices-subset[OF P1]
    have path-weight' (res u) ≤ Num (path-weight p1) by auto
    also have ... ≤ Num (path-weight p')
      using path-nonneg-weight[OF P2]
      apply (auto simp: infty-unbox )
      by (metis add-0-right add-left-mono)
    finally show ?thesis .
  qed
qed
```

With the previous statement, we easily show the third part of the invariant, as the new paths are not longer than the old ones.

```
show ∀ v∈V - (wl - {v}). ∀ p. is-path v0 p v
  ⟶ path-weight' (res' v) ≤ Num (path-weight p)
proof (intro allI ballI impI)
  fix v' p
  assume NOT-IN-WL: v' ∈ V - (wl - {v})
    and PATH: is-path v0 p v'
  hence [simp, intro!]: v'∈V by auto
  from bspec[OF CONSIDERED-NEW-PATHS, of v']
  have path-weight' (res' v') ≤ path-weight' (res v')
    by (auto dest: least-map-leD)
  also from bspec[OF RES-MIN NOT-IN-WL] PATH
  have path-weight' (res v') ≤ Num (path-weight p) by blast
  finally show path-weight' (res' v') ≤ Num (path-weight p) .
qed
```

Finally, we have to show that for nodes on the worklist, the stored paths are not longer than any path using only nodes not on the worklist. Compared to the situation before the step, those path may also use the node v.

show $\forall\, va{\in}wl - \{v\}.\ \forall\, p.$
$is\text{-}path\ v0\ p\ va \wedge int\text{-}vertices\ p \subseteq V - (wl - \{v\})$
$\longrightarrow path\text{-}weight'\ (res'\ va) \leq Num\ (path\text{-}weight\ p)$
proof (*intro allI impI ballI*, *elim conjE*)
fix $v'\ p$
assume *IWS*: $v'{\in}wl - \{v\}$
and *PATH*: *is-path v0 p v'*
and *VERTICES*: *int-vertices* $p \subseteq V - (wl - \{v\})$
from *IWS WL-SUBSET* **have** [*simp*, *intro*!]: $v'{\in}V$ **by** *auto*

{

If the path is empty, the proposition follows easily from the invariant for the original states, as no intermediate nodes are used at all.

assume [*simp*]: $p{=}[]$
from *bspec*[*OF CONSIDERED-NEW-PATHS*, *of* v'] **have**
$path\text{-}weight'\ (res'\ v') \leq path\text{-}weight'\ (res\ v')$
using *IWS WL-SUBSET* **by** (*auto dest*: *least-map-leD*)
also have *int-vertices* $p \subseteq V{-}wl$ **by** *auto*
with *WL-MIN IWS PATH*
have $path\text{-}weight'\ (res\ v') \leq Num\ (path\text{-}weight\ p)$
by (*auto simp del*: *path-weight-empty*)
finally have $path\text{-}weight'\ (res'\ v') \leq Num\ (path\text{-}weight\ p)$ **.**
} moreover {
fix *p1 u w*
assume [*simp*]: $p = p1@[(u,w,v')]$

If the path is not empty, we pick the last but one vertex, and call it u.

from *PATH* **have** *PATH1*: *is-path v0 p1 u* **and** *EDGE*: $(u,w,v'){\in}E$ **by** *auto*
from *VERTICES* **have** *NIV*: $u{\in}V - (wl{-}\{v\})$ **by** *simp*
hence *U-MEM*[*simp*]: $u{\in}V$ **by** *auto*

From *RES-MIN*, we know that *res u* holds the shortest path to u. Thus p is longer than the path that is constructed by replacing the prefix of p by term "res u"

from *NIV RES-MIN PATH1*
have *G*: $Num\ (path\text{-}weight\ p1) \geq path\text{-}weight'\ (res\ u)$ **by** *simp*
then obtain *pu* **where** [*simp*]: *res u = Some pu*
by (*cases res u*) (*auto simp*: *infty-unbox*)
from *G* **have** $Num\ (path\text{-}weight\ p) \geq path\text{-}weight'\ (res\ u) + Num\ w$
by (*auto simp*: *infty-unbox add-right-mono*)
also
have $path\text{-}weight'\ (res\ u) + Num\ w \geq path\text{-}weight'\ (res'\ v')$

The remaining argument depends on wether u equals v. In the case $u \neq v$, all vertices of *res u* are outside the original workset. Thus, appending the edge (u, w, v') to *res u* yields a path to v over intermediate nodes only outside the workset. By the invariant for the original state, *res* v' is shorter than this path. As a step does not replace paths by longer ones, also *res'* v' is shorter.

In the case $u = v$, the step has considered the path to v' over v, and thus the result path is not longer.

```
      proof (cases u=v)
        assume u≠v
        with NIV have NIV': u∈V−wl by auto
        from bspec[OF PATH-VALID U-MEM] NIV'
        have is-path v0 pu u and VU: int-vertices (pu@[(u,w,v')]) ⊆ V−wl
          by auto
        with EDGE have PV': is-path v0 (pu@[(u,w,v')]) v' by auto
        with bspec[OF WL-MIN, of v'] IWS VU have
          path-weight' (res v') ≤ Num (path-weight (pu@[(u,w,v')]))
          by blast
        hence path-weight' (res u) + Num w ≥ path-weight' (res v')
          by (auto simp: infty-unbox)
        also from CONSIDERED-NEW-PATHS have
          path-weight' (res v') ≥ path-weight' (res' v')
          by (auto dest: least-map-leD)
        finally (order-trans[rotated]) show ?thesis .
      next
        assume [symmetric,simp]: u=v
        from CONSIDERED-NEW-PATHS EDGE have
          path-weight' (res' v') ≤ path-weight' (Some (pu@[(v,w,v')]))
          by (rule-tac least-map-leD) auto
        thus ?thesis by (auto simp: infty-unbox)
      qed
      finally (order-trans[rotated]) have
        path-weight' (res' v') ≤ Num (path-weight p) .
    } ultimately show path-weight' (res' v') ≤ Num (path-weight p)
      using PATH apply (cases p rule: rev-cases) by auto
  qed
 qed
qed
```

5.4 Structural Refinement of Update

Now that we have proved correct the initial version of the algorithm, we start refinement towards an efficient implementation.

First, the update function is refined to iterate over each successor of the selected node, and update the result on demand.

```
definition uinvar
  :: 'V ⇒ 'V set ⇒ - ⇒ ('W×'V) set ⇒ ('V,'W) state ⇒ bool where
  uinvar v wl res it σ ≡ let (wl',res')=σ in wl'=wl
  ∧ (∀ v'∈V.
    res' v' ∈ least-map path-weight' (
      { res v' } ∪ { Some (p@[(v,w,v')]) | p w. res v = Some p
        ∧ (w,v') ∈ succ G v − it }
    ))
```

```
  ∧ (∀ v'∈V. ∀ p. res' v' = Some p ⟶ is-path v0 p v')
  ∧ res' v = res v


definition update' :: 'V ⇒ ('V,'W) state ⇒ ('V,'W) state nres where
  update' v σ ≡ do {
    ASSERT (update-pre v σ);
    let (wl,res) = σ;
    let wv = path-weight' (res v);
    let pv = res v;
    FOREACH^uinvar v wl res (succ G v) (λ(w',v') (wl,res).
      if (wv + Num w' < path-weight' (res v')) then do {
          ASSERT (v'∈wl ∧ pv≠None);
          RETURN (wl,res(v' ↦ the pv@[(v,w',v')]))
      } else RETURN (wl,res)
    ) (wl,res)}

lemma update'-refines:
  assumes v'=v and σ'=σ
  shows update' v' σ' ≤ ⇓Id (update v σ)
  apply (simp only: assms)
  unfolding update'-def update-def
  apply (refine-rcg)
  apply assumption

  apply (intro refine-vcg conjI)
  apply (simp-all only: singleton-iff)
proof −
  fix wl res
  assume update-pre v (wl,res)
  thus uinvar v wl res (succ G v) (wl,res)
    by (simp add: uinvar-def update-pre-def)
next

  fix wl res it wl' res' v' w'
  assume PRE: update-pre v (wl,res)
  assume INV: uinvar v wl res it (wl',res')
  assume MEM: (w',v')∈it
  assume IT-SS: it⊆ succ G v
  assume LESS: path-weight' (res v) + Num w' < path-weight' (res' v')

  from PRE have [simp, intro!]: v∈V by (simp add: update-pre-def)

  from MEM IT-SS have [simp,intro!]: v'∈V using succ-subset
    by auto

  from LESS obtain pv where [simp]: res v = Some pv
    by (cases res v) auto
```

```
thus res v ≠ None by simp

have [simp]: wl'=wl and [simp]: res' v = res v
  using INV unfolding uinvar-def by auto

from MEM IT-SS have EDGE[simp]: (v,w',v')∈E
  unfolding succ-def by auto
with INV have [simp]: is-path v0 pv v
  unfolding uinvar-def by auto

have 0≤w' by (rule nonneg-weights[OF EDGE])
hence [simp]: v'≠v using LESS
  by auto
hence [simp]: v≠v' by blast

show [simp]: v'∈wl' proof (rule ccontr)
  assume [simp]: v'∉wl'
  hence [simp]: v'∈V−wl and [simp]: v'∉wl by auto
  note LESS
  also
  from INV have path-weight' (res' v') ≤ path-weight' (res v')
    unfolding uinvar-def by (auto dest: least-map-leD)
  also
  from PRE have PW: ⋀p. is-path v0 p v' ⟹
    path-weight' (res v') ≤ path-weight' (Some p)
    unfolding update-pre-def
    by auto
  have P: is-path v0 (pv@[(v,w',v')]) v' by simp
  from PW[OF P] have
    path-weight' (res v') ≤ Num (path-weight (pv@[(v,w',v')]))
    by auto
  finally show False by (simp add: infty-unbox)
qed

show uinvar v wl res (it−{(w',v')}) (wl',res'(v'↦the (res v)@[(v,w',v')]))
proof −
  have (res'(v'↦the (res v)@[(v,w',v')])) v = res' v by simp
  moreover {
    fix v'' assume VMEM: v''∈V
    have (res'(v'↦the (res v)@[(v,w',v')])) v'' ∈ least-map path-weight' (
      { res v'' } ∪ { Some (p@[(v,w,v'')]) | p w. res v = Some p
      ∧ (w,v'') ∈ succ G v − (it − {(w',v')}) }
      ) ∧ (∀ p. (res'(v'↦the (res v)@[(v,w',v')])) v'' = Some p
            ⟶ is-path v0 p v'')
    proof (cases v''=v')
      case False[simp]
      have { Some (p@[(v,w,v'')]) | p w. res v = Some p
      ∧ (w,v'') ∈ succ G v − (it − {(w',v')}) } =
        { Some (p@[(v,w,v'')]) | p w. res v = Some p
```

```
      ∧ (w,v'') ∈ succ G v − it }
        by auto
      with INV VMEM show ?thesis unfolding uinvar-def
        by simp
    next
      case True[simp]
      have EQ: { res v'' } ∪ { Some (p@[(v,w,v'')]) | p w. res v = Some p
      ∧ (w,v'') ∈ succ G v − (it − {(w',v')}) } =
      insert (Some (pv@[(v,w',v')])) (
        { res v'' } ∪ { Some (p@[(v,w,v'')]) | p w. res v = Some p
      ∧ (w,v'') ∈ succ G v − it })
        using MEM IT-SS
        by auto
      show ?thesis
        apply (subst EQ)
        apply simp
        apply (rule least-map-insert-min)
        apply (rule ballI)
      proof −
        fix r'
        assume A:
          r' ∈ insert (res v')
           {Some (pv @ [(v, w, v')]) |w. (w, v') ∈ succ G v ∧ (w, v') ∉ it}

        from LESS have
          path-weight' (Some (pv @ [(v, w', v')])) < path-weight' (res' v')
          by (auto simp: infty-unbox)
        also from INV[unfolded uinvar-def] have
          res' v' ∈ least-map path-weight' (
            insert (res v')
            {Some (pv @ [(v, w, v')]) |w. (w, v') ∈ succ G v ∧ (w, v') ∉ it}
          )
          by auto
        with A have path-weight' (res' v') ≤ path-weight' r'
          by (auto dest: least-map-leD)
        finally show
          path-weight' (Some (pv @ [(v, w', v')])) ≤ path-weight' r'
          by simp
      qed
    qed
  }
  ultimately show ?thesis
    unfolding uinvar-def Let-def
    by auto
qed
next
  fix wl res it w' v' wl' res'
  assume INV: uinvar v wl res it (wl',res')
  and NLESS: ¬ path-weight' (res v) + Num w' < path-weight' (res' v')
```

```
  and IN-IT: (w',v')∈it
  and IT-SS: it ⊆ succ G v

  from IN-IT IT-SS have [simp, intro!]: (w',v')∈succ G v by auto
  hence [simp,intro!]: v'∈V using succ-subset
    by auto

  show uinvar v wl res (it − {(w',v')}) (wl',res')
  proof (cases res v)
    case None [simp]
    from INV show ?thesis
      unfolding uinvar-def by auto
  next
    case (Some p) [simp]
    {
      fix v''
      assume [simp, intro!]: v''∈V
      have res' v'' ∈ least-map path-weight' (
        { res v'' } ∪ { Some (p@[(v,w,v'')]) | p w. res v = Some p
        ∧ (w,v'') ∈ succ G v − (it − {(w',v')}) }
        ) (is - ∈ least-map path-weight' ?S)
      proof (cases v''=v')
        case False with INV show ?thesis
          unfolding uinvar-def by auto
      next
        case True[simp]

        have EQ: ?S = insert (Some (p@[(v,w',v')])) (
          { res v' } ∪ { Some (p@[(v,w,v'')]) | p w. res v = Some p
                       ∧ (w,v'') ∈ succ G v − it }
          )
          by auto
        from NLESS have
          path-weight' (res' v') ≤ path-weight' (Some (p@[(v,w',v')]))
          by (auto simp: infty-unbox)
        thus ?thesis
          apply (subst EQ)
          apply (rule least-map-insert-nmin)
          using INV unfolding uinvar-def apply auto []
          apply simp
          done
      qed
    } with INV
    show ?thesis
      unfolding uinvar-def by auto
  qed
next
  fix wl res σ'
```

```
  assume uinvar v wl res {} σ'
  thus update-spec v (wl,res) σ'
    unfolding uinvar-def
    apply (cases σ')
    apply (auto intro: update-spec.intros simp: succ-def)
    done
next
  show finite (succ G v) by simp
qed
```

We integrate the new update function into the main algorithm:

```
definition dijkstra' where
  dijkstra' ≡ do {
    σ0 ← dinit;
    (-,res) ← WHILE_T^dinvar (λ(wl,-). wl≠{})
        (λσ. do {(v,σ') ← pop-min σ; update' v σ'})
        σ0;
    RETURN res
  }


lemma dijkstra'-refines: dijkstra' ≤ ⇓Id dijkstra
proof −
  note [refine] = update'-refines
  have [refine]: ⋀σ σ'. σ=σ' ⟹ pop-min σ ≤ ⇓Id (pop-min σ') by simp
  show ?thesis
    unfolding dijkstra-def dijkstra'-def
    apply (refine-rcg)
    apply simp-all
    done
qed
end
```

5.5 Refinement to Cached Weights

Next, we refine the data types of the workset and the result map. The workset becomes a map from nodes to their current weights. The result map stores, in addition to the shortest path, also the weight of the shortest path. Moreover, we store the shortest paths in reversed order, which makes appending new edges more effcient.

These refinements allow to implement the workset as a priority queue, and save recomputation of the path weights in the inner loop of the algorithm.

```
type-synonym ('V,'W) mwl = ('V ⇀ 'W infty)
type-synonym ('V,'W) mres = ('V ⇀ (('V,'W) path × 'W))
type-synonym ('V,'W) mstate = ('V,'W) mwl × ('V,'W) mres
```

Map a path with cached weight to one without cached weight.

```
fun mpath' :: (('V,'W) path × 'W) option ⇀ ('V,'W) path where
```

```
  mpath' None = None |
  mpath' (Some (p,w)) = Some p

fun mpath-weight' :: (('V,'W) path × 'W) option ⇒ ('W::weight) infty where
  mpath-weight' None = top |
  mpath-weight' (Some (p,w)) = Num w

context Dijkstra
begin
  definition αw::('V,'W) mwl ⇒ 'V set where αw ≡ dom
  definition αr::('V,'W) mres ⇒ 'V ⇀ ('V,'W) path where
    αr ≡ λres v. case res v of None ⇒ None | Some (p,w) ⇒ Some (rev p)
  definition αs:: ('V,'W) mstate ⇒ ('V,'W) state where
    αs ≡ map-pair αw αr
```

Additional invariants for the new state. They guarantee that the cached weights are consistent.

```
  definition res-invarm :: ('V ⇀ (('V,'W) path×'W)) ⇒ bool where
    res-invarm res ≡ (∀ v. case res v of
        None ⇒ True |
        Some (p,w) ⇒ w = path-weight (rev p))
  definition dinvarm :: ('V,'W) mstate ⇒ bool where
    dinvarm σ ≡ let (wl,res) = σ in
      (∀ v∈dom wl. the (wl v) = mpath-weight' (res v)) ∧ res-invarm res

  lemma mpath-weight'-correct: ⟦dinvarm (wl,res)⟧ ⟹
    mpath-weight' (res v) = path-weight' (αr res v)

    unfolding dinvarm-def res-invarm-def αr-def
    by (auto split: option.split option.split-asm)

  lemma mpath'-correct: ⟦dinvarm (wl,res)⟧ ⟹
    mpath' (res v) = Option.map rev (αr res v)
    unfolding dinvarm-def αr-def
    by (auto split: option.split option.split-asm)

  lemma wl-weight-correct:
    assumes INV: dinvarm (wl,res)
    assumes WLV: wl v = Some w
    shows path-weight' (αr res v) = w
  proof −
    from INV WLV have w = mpath-weight' (res v)
      unfolding dinvarm-def by force
    also from mpath-weight'-correct[OF INV] have
      ... = path-weight' (αr res v) .
    finally show ?thesis by simp
  qed
```

The initial state is constructed using an iterator:

```
definition mdinit :: ('V,'W) mstate nres where
  mdinit ≡ do {
    wl ← FOREACH V (λv wl. RETURN (wl(v↦Infty))) Map.empty;
    RETURN (wl(v0↦Num 0),[v0 ↦ ([],0)])
  }

lemma mdinit-refines: mdinit ≤ ⇓(build-rel αs dinvarm) dinit
  unfolding mdinit-def dinit-def
  apply (rule build-rel-SPEC)
  apply (intro FOREACH-rule[where I=λit wl. (∀ v∈V−it. wl v = Some Infty)
∧
    dom wl = V−it]
      refine-vcg)
  apply (auto
    simp: αs-def αw-def αr-def dinvarm-def res-invarm-def infty-unbox
    split: split-if-asm
  )
  done
```

The new pop function:

```
definition
  mpop-min :: ('V,'W) mstate ⇒ ('V × 'W infty × ('V,'W) mstate) nres
  where
  mpop-min σ ≡ do {
    let (wl,res) = σ;
    (v,w,wl')←prio-pop-min wl;
    RETURN (v,w,(wl',res))
  }

lemma mpop-min-refines:
  ⟦ (σ,σ') ∈ build-rel αs dinvarm ⟧ ⟹
    mpop-min σ ≤
     ⇓(build-rel
        (λ(v,w,σ). (v,αs σ))
        (λ(v,w,σ). dinvarm σ ∧ w = mpath-weight' (snd σ v)))
    (pop-min σ')
   — The two algorithms are structurally different, so we use the nofail/inres
    method to prove refinement.
  unfolding mpop-min-def pop-min-def prio-pop-min-def

  apply (rule pw-ref-svI)
  apply rule
  apply (auto simp add: refine-pw-simps αs-def αw-def
    split: prod.split prod.split-asm)

  apply (auto simp: dinvarm-def) []

  apply (auto simp: mpath-weight'-correct wl-weight-correct) []
```

```
  apply (auto
    simp: wl-weight-correct
    intro!: least-map.intros
  ) []
  done
```

The new update function:

```
  definition uinvarm v wl res it σ ≡
    uinvar v wl res it (αs σ) ∧ dinvarm σ

  definition mupdate :: 'V ⇒ 'W infty ⇒ ('V,'W) mstate ⇒ ('V,'W) mstate
nres
  where
    mupdate v wv σ ≡ do {
      ASSERT (update-pre v (αs σ) ∧ wv=mpath-weight' (snd σ v));
      let (wl,res) = σ;
      let pv = mpath' (res v);
      FOREACH^uinvarm v (αw wl) (αr res) (succ G v) (λ(w',v') (wl,res).
        if (wv + Num w' < mpath-weight' (res v')) then do {
          ASSERT (v'∈dom wl ∧ pv ≠ None);
          RETURN (wl(v'↦wv + Num w'),
                  res(v' ↦ ((v,w',v')#the pv,val wv + w') ))
        } else RETURN (wl,res)
        ) (wl,res)
    }

  lemma mupdate-refines:
    assumes SREF: (σ,σ')∈build-rel αs dinvarm
    assumes WV: wv = mpath-weight' (snd σ v)
    assumes VV': v'=v
    shows mupdate v wv σ ≤ ⇓(build-rel αs dinvarm) (update' v' σ')
  proof (simp only: VV')
    {
```

Show that IF-condition is a refinement:

```
      fix wl res wl' res' it w' v'
      assume uinvarm v (αw wl) (αr res) it (wl',res')
        and dinvarm (wl,res)
      hence mpath-weight' (res v) + Num w' < mpath-weight' (res' v') ⟷
        path-weight' (αr res v) + Num w' < path-weight' (αr res' v')
        unfolding uinvarm-def
        by (auto simp add: mpath-weight'-correct)
    } note COND-refine=this

    {
```

THEN-case:

```
      fix wl res wl' res' it w' v'
      assume UINV: uinvarm v (αw wl) (αr res) it (wl',res')
```

```
    and DINV: dinvarm (wl,res)
    and mpath-weight' (res v) + Num w' < mpath-weight' (res' v')
    and path-weight' (αr res v) + Num w' < path-weight' (αr res' v')
    and V'MEM: v'∈αw wl'
    and NN: αr res v ≠ None

  from NN obtain pv wv where
    ARV: αr res v = Some (rev pv) and
    RV: res v = Some (pv,wv)
    unfolding αr-def by (auto split: option.split-asm)

  with DINV have [simp]: wv = path-weight (rev pv)
    unfolding dinvarm-def res-invarm-def by (auto split: option.split-asm)

  note [simp] = ARV RV

  from V'MEM NN have v'∈dom wl' (is ?G1)
    and mpath' (res v) ≠ None (is ?G2)
    unfolding αw-def αr-def by (auto split: option.split-asm)

  hence ⋀x. αw wl' = αw (wl'(v'↦x)) by (auto simp: αw-def)
  moreover have mpath' (res v) = Option.map rev (αr res v) using DINV
    by (simp add: mpath'-correct)
  ultimately have
    αw wl' = αw (wl'(v' ↦ mpath-weight' (res v) + Num w'))
    ∧ (αr res')(v' ↦ the (αr res v)@[(v, w', v')])
      = αr (res'(v' ↦ ((v, w', v')#the (mpath' (res v)),
            val (mpath-weight' (res v)) + w'))) (is ?G3)
    by (auto simp add: αr-def intro!: ext)
  have
    (dinvarm (wl'(v'↦mpath-weight' (res v) + Num w'),
                  res'(v' ↦ ((v,w',v') # the (mpath' (res v)),
                              val (mpath-weight' (res v)) + w'
                             )))) (is ?G4)
    using UINV unfolding uinvarm-def dinvarm-def res-invarm-def
    by (auto simp: infty-unbox split: option.split option.split-asm)
  note ‹?G1› ‹?G2› ‹?G3› ‹?G4›
} note THEN-refine=this


note [refine2] = inj-on-id

show mupdate v wv σ ≤ ⇓(build-rel αs dinvarm) (update' v σ')
  using SREF WV
  unfolding mupdate-def update'-def
  apply −

  apply (refine-rcg)
  apply simp-all [4]
```

apply (*simp add*: α*s-def uinvarm-def*)
apply (*simp-all add*: α*s-def COND-refine THEN-refine*(*1*−*2*)) [*3*]
using *THEN-refine*(*3*,*4*)
apply (*auto simp*: α*s-def*) []

The ELSE-case is trivial:

apply *simp*
done
qed

Finally, we assemble the refined algorithm:

definition *mdijkstra* **where**
mdijkstra ≡ *do* {
$\sigma 0$ ← *mdinit*;
(-,*res*) ← *WHILE*$_T$^{dinvarm} (λ(*wl*,-). *dom wl*≠{})
(λσ. *do* { (*v*,*wv*,σ') ← *mpop-min* σ; *mupdate v wv* σ' })
$\sigma 0$;
RETURN res
}

lemma *mdijkstra-refines*: *mdijkstra* ≤ ⇓(*build-rel* α*r res-invarm*) *dijkstra'*
proof −
note [*refine*] = *mdinit-refines mpop-min-refines mupdate-refines*
show *?thesis*
unfolding *mdijkstra-def dijkstra'-def*
apply (*refine-rcg*)
apply (*simp-all split*: *prod.split*
add: α*s-def* α*w-def dinvarm-def*)
done
qed

end

end

6 Graph Interface

theory *GraphSpec*
imports *Main Graph ../Refine-Monadic/Refine*

begin

This theory defines an ICF-style interface for graphs.

type-synonym ('*V*,'*W*,'*G*) *graph-*α = '*G* ⇒ ('*V*,'*W*) *graph*

locale *graph* =
fixes α :: '*G* ⇒ ('*V*,'*W*) *graph*

fixes *invar* :: $'G \Rightarrow bool$
assumes *finite*[*simp*, *intro*!]:
$invar\ g \Longrightarrow finite\ (nodes\ (\alpha\ g))$
$invar\ g \Longrightarrow finite\ (edges\ (\alpha\ g))$
assumes *valid*: $invar\ g \Longrightarrow valid\text{-}graph\ (\alpha\ g)$

type-synonym $('V,'W,'G)$ *graph-empty* = *unit* $\Rightarrow$ $'G$
locale *graph-empty* = *graph* +
constrains α :: $'G \Rightarrow ('V,'W)$ *graph*
fixes *empty* :: *unit* $\Rightarrow$ $'G$
assumes *empty-correct*:
$\alpha\ (empty\ ()) = Graph.empty$
$invar\ (empty\ ())$

type-synonym $('V,'W,'G)$ *graph-add-node* = $'V \Rightarrow 'G \Rightarrow 'G$
locale *graph-add-node* = *graph* +
constrains α :: $'G \Rightarrow ('V,'W)$ *graph*
fixes *add-node* :: $'V \Rightarrow 'G \Rightarrow 'G$
assumes *add-node-correct*:
$invar\ g \Longrightarrow invar\ (add\text{-}node\ v\ g)$
$invar\ g \Longrightarrow \alpha\ (add\text{-}node\ v\ g) = Graph.add\text{-}node\ v\ (\alpha\ g)$

type-synonym $('V,'W,'G)$ *graph-delete-node* = $'V \Rightarrow 'G \Rightarrow 'G$
locale *graph-delete-node* = *graph* +
constrains α :: $'G \Rightarrow ('V,'W)$ *graph*
fixes *delete-node* :: $'V \Rightarrow 'G \Rightarrow 'G$
assumes *delete-node-correct*:
$invar\ g \Longrightarrow invar\ (delete\text{-}node\ v\ g)$
$invar\ g \Longrightarrow \alpha\ (delete\text{-}node\ v\ g) = Graph.delete\text{-}node\ v\ (\alpha\ g)$

type-synonym $('V,'W,'G)$ *graph-add-edge* = $'V \Rightarrow 'W \Rightarrow 'V \Rightarrow 'G \Rightarrow 'G$
locale *graph-add-edge* = *graph* +
constrains α :: $'G \Rightarrow ('V,'W)$ *graph*
fixes *add-edge* :: $'V \Rightarrow 'W \Rightarrow 'V \Rightarrow 'G \Rightarrow 'G$
assumes *add-edge-correct*:
$invar\ g \Longrightarrow invar\ (add\text{-}edge\ v\ e\ v'\ g)$
$invar\ g \Longrightarrow \alpha\ (add\text{-}edge\ v\ e\ v'\ g) = Graph.add\text{-}edge\ v\ e\ v'\ (\alpha\ g)$

type-synonym $('V,'W,'G)$ *graph-delete-edge* = $'V \Rightarrow 'W \Rightarrow 'V \Rightarrow 'G \Rightarrow 'G$
locale *graph-delete-edge* = *graph* +
constrains α :: $'G \Rightarrow ('V,'W)$ *graph*
fixes *delete-edge* :: $'V \Rightarrow 'W \Rightarrow 'V \Rightarrow 'G \Rightarrow 'G$
assumes *delete-edge-correct*:
$invar\ g \Longrightarrow invar\ (delete\text{-}edge\ v\ e\ v'\ g)$
$invar\ g \Longrightarrow \alpha\ (delete\text{-}edge\ v\ e\ v'\ g) = Graph.delete\text{-}edge\ v\ e\ v'\ (\alpha\ g)$

type-synonym $('V,'W,'\sigma,'G)$ *graph-nodes-it* = $'G \Rightarrow ('V,'\sigma)$ *set-iterator*
locale *graph-nodes-it* = *graph* +
constrains α :: $'G \Rightarrow ('V,'W)$ *graph*

fixes *nodes-it* :: $'G \Rightarrow ('V,'\sigma)$ *set-iterator*
assumes *nodes-it-correct*:
invar g $\Longrightarrow$ *set-iterator* (*nodes-it g*) (*Graph.nodes* (α *g*))

type-synonym $('V,'W,'\sigma,'G)$ *graph-edges-it*
$= 'G \Rightarrow (('V \times 'W \times 'V),'\sigma)$ *set-iterator*
locale *graph-edges-it* = *graph* +
constrains $\alpha :: 'G \Rightarrow ('V,'W)$ *graph*
fixes *edges-it* :: $('V,'W,'\sigma,'G)$ *graph-edges-it*
assumes *edges-it-correct*:
invar g $\Longrightarrow$ *set-iterator* (*edges-it g*) (*Graph.edges* (α *g*))

type-synonym $('V,'W,'\sigma,'G)$ *graph-succ-it* =
$'G \Rightarrow 'V \Rightarrow ('W \times 'V,'\sigma)$ *set-iterator*
locale *graph-succ-it* = *graph* +
constrains $\alpha :: 'G \Rightarrow ('V,'W)$ *graph*
fixes *succ-it* :: $'G \Rightarrow 'V \Rightarrow ('W \times 'V,'\sigma)$ *set-iterator*
assumes *succ-it-correct*:
invar g $\Longrightarrow$ *set-iterator* (*succ-it g v*) (*Graph.succ* (α *g*) *v*)

6.1 Adjacency Lists

type-synonym $('V,'W)$ *adj-list* = $'V$ *list* $\times$ $('V \times 'W \times 'V)$ *list*

definition *adjl-*α :: $('V,'W)$ *adj-list* $\Rightarrow$ $('V,'W)$ *graph* **where**
adjl-α *l* $\equiv$ *let* (*nl*,*el*) = *l* *in* ⦇
nodes = *set nl* $\cup$ *fst*‘*set el* $\cup$ *snd*‘*snd*‘*set el*,
edges = *set el*
⦈

lemma *adjl-is-graph*: *graph adjl-*α (λ-. *True*)
apply (*unfold-locales*)
unfolding *adjl-*α*-def*
by *force*+

type-synonym $('V,'W,'G)$ *graph-from-list* = $('V,'W)$ *adj-list* $\Rightarrow$ $'G$
locale *graph-from-list* = *graph* +
constrains $\alpha :: 'G \Rightarrow ('V,'W)$ *graph*
fixes *from-list* :: $('V,'W)$ *adj-list* $\Rightarrow$ $'G$
assumes *from-list-correct*:
invar (*from-list l*)
α (*from-list l*) = *adjl-*α *l*

type-synonym $('V,'W,'G)$ *graph-to-list* = $'G \Rightarrow ('V,'W)$ *adj-list*
locale *graph-to-list* = *graph* +
constrains $\alpha :: 'G \Rightarrow ('V,'W)$ *graph*
fixes *to-list* :: $'G \Rightarrow ('V,'W)$ *adj-list*
assumes *to-list-correct*:

invar g $\Longrightarrow$ *adjl-α* (*to-list g*) = *α g*

6.2 Record Based Interface

record (*'V*,*'W*,*'G*) *graph-ops* =
gop-α :: (*'V*,*'W*,*'G*) *graph-α*
gop-invar :: *'G* ⇒ *bool*
gop-empty :: (*'V*,*'W*,*'G*) *graph-empty*
gop-add-node :: (*'V*,*'W*,*'G*) *graph-add-node*
gop-delete-node :: (*'V*,*'W*,*'G*) *graph-delete-node*
gop-add-edge :: (*'V*,*'W*,*'G*) *graph-add-edge*
gop-delete-edge :: (*'V*,*'W*,*'G*) *graph-delete-edge*
gop-from-list :: (*'V*,*'W*,*'G*) *graph-from-list*
gop-to-list :: (*'V*,*'W*,*'G*) *graph-to-list*

locale *StdGraphDefs* =
fixes *ops* :: (*'V*,*'W*,*'G*,*'m*) *graph-ops-scheme*
begin
abbreviation *α* **where** *α* ≡ *gop-α ops*
abbreviation *invar* **where** *invar* ≡ *gop-invar ops*
abbreviation *empty* **where** *empty* ≡ *gop-empty ops*
abbreviation *add-node* **where** *add-node* ≡ *gop-add-node ops*
abbreviation *delete-node* **where** *delete-node* ≡ *gop-delete-node ops*
abbreviation *add-edge* **where** *add-edge* ≡ *gop-add-edge ops*
abbreviation *delete-edge* **where** *delete-edge* ≡ *gop-delete-edge ops*
abbreviation *from-list* **where** *from-list* ≡ *gop-from-list ops*
abbreviation *to-list* **where** *to-list* ≡ *gop-to-list ops*
end

locale *StdGraph* = *StdGraphDefs* +
graph α invar +
graph-empty α invar empty +
graph-add-node α invar add-node +
graph-delete-node α invar delete-node +
graph-add-edge α invar add-edge +
graph-delete-edge α invar delete-edge +
graph-from-list α invar from-list +
graph-to-list α invar to-list
begin
lemmas *correct* = *empty-correct add-node-correct delete-node-correct*
add-edge-correct delete-edge-correct
from-list-correct to-list-correct

end

6.3 Refinement Framework Bindings

lemma (**in** *graph-nodes-it*) *nodes-it-is-iterator*[*refine-transfer*]:
invar g $\Longrightarrow$ *set-iterator* (*nodes-it g*) (*nodes* (*α g*))
by (*rule nodes-it-correct*)

lemma (**in** *graph-edges-it*) *edges-it-is-iterator*[*refine-transfer*]:
invar g ⟹ *set-iterator* (*edges-it g*) (*edges* (*α g*))
by (*rule edges-it-correct*)

lemma (**in** *graph-succ-it*) *succ-it-is-iterator*[*refine-transfer*]:
invar g ⟹ *set-iterator* (*succ-it g v*) (*Graph.succ* (*α g*) *v*)
by (*rule succ-it-correct*)

lemma (**in** *graph*) *drh*[*refine-dref-RELATES*]: *RELATES* (*build-rel α invar*)
by (*simp add*: *RELATES-def*)

end

7 Generic Algorithms for Graphs

theory *GraphGA*
imports *GraphSpec* ../*Collections*/*iterator*/*SetIteratorOperations*
begin

definition *gga-from-list* ::
('*V*,'*W*,'*G*) *graph-empty* ⇒ ('*V*,'*W*,'*G*) *graph-add-node*
⇒ ('*V*,'*W*,'*G*) *graph-add-edge*
⇒ ('*V*,'*W*,'*G*) *graph-from-list*
where
gga-from-list e a u l ≡
let (*nl*,*el*) = *l*;
g1 = *foldl* (λ*g v*. *a v g*) (*e* ()) *nl*
in foldl (λ*g* (*v*,*e*,*v'*). *u v e v' g*) *g1 el*

lemma *gga-from-list-correct*:
fixes *α* :: '*G* ⇒ ('*V*,'*W*) *graph*
assumes *graph-empty α invar e*
assumes *graph-add-node α invar a*
assumes *graph-add-edge α invar u*
shows *graph-from-list α invar* (*gga-from-list e a u*)
proof −
interpret
graph-empty α invar e +
graph-add-node α invar a +
graph-add-edge α invar u
by *fact*+

{
fix *nl el*
def *g1*≡(*foldl* (λ*g v*. *a v g*) (*e* ()) *nl*)

```
    def g2≡(foldl (λg (v,e,v'). u v e v' g) g1 el)
    have invar g1 ∧ α g1 = (| nodes = set nl, edges = {} |)
      unfolding g1-def
      by (induct nl rule: rev-induct)
        (auto simp: empty-correct add-node-correct empty-def add-node-def)
    hence invar g2
      ∧ α g2 = (| nodes = set nl ∪ fst`set el ∪ snd`snd`set el,
                  edges = set el |)
      unfolding g2-def
      by (induct el rule: rev-induct) (auto simp: add-edge-correct add-edge-def)
    hence invar g2 ∧ adjl-α (nl,el) = α g2
      unfolding adjl-α-def by auto
  }
  thus ?thesis
    unfolding gga-from-list-def [abs-def]
    apply unfold-locales
    apply auto
    done
qed

term map-iterator-product

definition gga-edges-it ::
  ('V,'W,'σ,'G) graph-nodes-it ⇒ ('V,'W,'σ,'G) graph-succ-it
  ⇒ ('V,'W,'σ,'G) graph-edges-it
  where gga-edges-it ni si G ≡ set-iterator-product (ni G) (λv. si G v)



lemma gga-edges-it-correct:
  fixes ni::('V,'W,'σ,'G) graph-nodes-it
  fixes α :: 'G ⇒ ('V,'W) graph
  assumes graph-nodes-it α invar ni
  assumes graph-succ-it α invar si
  shows graph-edges-it α invar (gga-edges-it ni si)
proof −
  interpret graph-nodes-it α invar ni +
    graph-succ-it α invar si by fact+

  show ?thesis
  proof
    fix g
    assume INV: invar g

    from set-iterator-product-correct[OF
      nodes-it-correct[OF INV] succ-it-correct[OF INV]]
    have set-iterator (set-iterator-product (ni g) (λv. si g v))
      (SIGMA v:nodes (α g). succ (α g) v)
      .
```

```
    also have (SIGMA v:nodes (α g). succ (α g) v) = edges (α g)
      unfolding succ-def
      by (auto dest: valid-graph.E-validD[OF valid[OF INV]])
    finally show set-iterator (gga-edges-it ni si g) (edges (α g))
      unfolding gga-edges-it-def .
  qed
qed

definition gga-to-list ::
  ('V,'W,-,'G) graph-nodes-it ⇒ ('V,'W,-,'G) graph-edges-it ⇒
   ('V,'W,'G) graph-to-list
  where
  gga-to-list ni ei g ≡
    (ni g (λ-. True) (op #) [], ei g (λ-. True) (op #) [])


lemma gga-to-list-correct:
  fixes α :: 'G ⇒ ('V,'W) graph
  assumes graph-nodes-it α invar ni
  assumes graph-edges-it α invar ei
  shows graph-to-list α invar (gga-to-list ni ei)
proof −
  interpret graph-nodes-it α invar ni +
    graph-edges-it α invar ei
    by fact+

  show ?thesis
  proof
    fix g
    assume [simp, intro!]: invar g
    then interpret valid-graph α g by (rule valid)

    have set (ni g (λ-. True) (op #) []) = V
      apply (rule-tac I=λit σ. set σ = V − it
        in set-iterator-rule-P[OF nodes-it-correct])
      by auto
    moreover have set (ei g (λ-. True) (op #) []) = E
      apply (rule-tac I=λit σ. set σ = E − it
        in set-iterator-rule-P[OF edges-it-correct])
      by auto
    ultimately show adjl-α (gga-to-list ni ei g) = α g
      unfolding adjl-α-def gga-to-list-def
      apply simp
      apply (rule graph.equality)
      apply (auto intro: E-validD)
      done
  qed
qed
```

end

8 Implementing Graphs by Maps

theory *GraphByMap*
imports *../Collections/Collections*
GraphSpec GraphGA
begin

locale *gbm-defs* =
m1!: *StdMap m1-ops* +
m2!: *StdMap m2-ops* +
s3!: *StdSet s3-ops* +
m1!: *map-value-image-filter m1.α m1.invar m1.α m1.invar m1-mvif*
for *m1-ops*::('*V*,'*m2*,'*m1*,-) *map-ops-scheme*
and *m2-ops*::('*V*,'*s3*,'*m2*,-) *map-ops-scheme*
and *s3-ops*::('*W*,'*s3*,-) *set-ops-scheme*
and *m1-mvif* :: ('*V* ⇒ '*m2* ⇀ '*m2*) ⇒ '*m1* ⇒ '*m1*
begin
definition *gbm-α* :: ('*V*,'*W*,'*m1*) *graph-α* **where**
gbm-α m1 ≡
(| *nodes* = *dom* (*m1.α m1*),
edges = {(*v*,*w*,*v'*).
∃ *m2 s3*. *m1.α m1 v* = *Some m2*
∧ *m2.α m2 v'* = *Some s3*
∧ *w*∈*s3.α s3*
}
|)

definition *gbm-invar m1* ≡
m1.invar m1 ∧
(∀ *m2*∈*ran* (*m1.α m1*). *m2.invar m2* ∧
(∀ *s3*∈*ran* (*m2.α m2*). *s3.invar s3*)
) ∧ *valid-graph* (*gbm-α m1*)

lemma *gbm-invar-split*:
assumes *gbm-invar g*
shows
m1.invar g
⋀*v m2*. *m1.α g v* = *Some m2* ⟹ *m2.invar m2*
⋀*v m2 v' s3*. *m1.α g v* = *Some m2* ⟹ *m2.α m2 v'* = *Some s3* ⟹ *s3.invar s3*
valid-graph (*gbm-α g*)
using *assms* **unfolding** *gbm-invar-def*
by (*auto intro*: *ranI*)

end

definition *map-Sigma M1 F2* ≡ {
  (*x*,*y*). ∃ *v*. *M1 x* = *Some v* ∧ *y*∈*F2 v*
}

lemma *map-Sigma-alt*: *map-Sigma M1 F2* = *Sigma* (*dom M1*) (λ*x*.
  F2 (*the* (*M1 x*)))
  unfolding *map-Sigma-def*
  by *auto*

sublocale *gbm-defs* < *graph gbm-α gbm-invar*
proof
  fix *g*
  assume *INV*: *gbm-invar g*
  then interpret *vg*!: *valid-graph* (*gbm-α g*) **by** (*simp add*: *gbm-invar-def*)

  from *vg.E-valid*
  show *fst* ‘ *edges* (*gbm-α g*) ⊆ *nodes* (*gbm-α g*) **and**
    snd ‘ *snd* ‘ *edges* (*gbm-α g*) ⊆ *nodes* (*gbm-α g*) **.**

  from *INV* **show** *finite* (*nodes* (*gbm-α g*))
    unfolding *gbm-invar-def gbm-α-def* **by** *auto*

  note [*simp*] = *gbm-invar-split*[*OF INV*]

  show *finite* (*edges* (*gbm-α g*))
    apply (*rule finite-imageD*[**where** *f*=λ(*v*,*e*,*v*′). (*v*,*v*′,*e*)])
    apply (*rule finite-subset*[**where** *B*=
      map-Sigma (*m1.α g*) (λ*m2*. *map-Sigma* (*m2.α m2*) (*s3.α*))])
    apply (*auto simp add*: *map-Sigma-def gbm-α-def*) []
    apply (*unfold map-Sigma-alt*)
    apply (*auto intro*!: *finite-SigmaI inj-onI*)
    done
qed

lemma *ranE*:
  assumes *v*∈*ran m*
  obtains *k* **where** *m k* = *Some v*
  using *assms*
by (*metis ran-restrictD restrict-map-self*)
lemma *option-bind-alt*:
  Option.bind x f = (*case x of None* ⇒ *None* | *Some v* ⇒ *f v*)
  by (*auto split*: *option.split*)

```
context gbm-defs
begin

  definition gbm-empty :: ('V,'W,'m1) graph-empty where
    gbm-empty ≡ m1.empty

  lemma gbm-empty-correct:
    graph-empty gbm-α gbm-invar gbm-empty
    apply (unfold-locales)
    unfolding gbm-α-def gbm-invar-def gbm-empty-def
    apply (auto simp: m1.correct Graph.empty-def)
    apply (unfold-locales)
    apply auto
    done

  definition gbm-add-node :: ('V,'W,'m1) graph-add-node where
    gbm-add-node v g ≡ case m1.lookup v g of
    None ⇒ m1.update v (m2.empty ()) g |
    Some - ⇒ g

  lemma gbm-add-node-correct:
    graph-add-node gbm-α gbm-invar gbm-add-node
  proof
    fix g v
    assume INV: gbm-invar g
    note [simp]= gbm-invar-split[OF INV]
    show gbm-α (gbm-add-node v g) = add-node v (gbm-α g)
      unfolding gbm-α-def gbm-add-node-def
      by (auto simp: m1.correct m2.correct s3.correct add-node-def
        split: option.split split-if-asm)

    thus gbm-invar (gbm-add-node v g)
      unfolding gbm-invar-def
      apply (simp)
      unfolding gbm-α-def gbm-add-node-def add-node-def
      apply (auto simp: m1.correct m2.correct s3.correct add-node-def
        split: option.split split-if-asm elim!: ranE)
      done
  qed

  definition gbm-delete-node :: ('V,'W,'m1) graph-delete-node where
    gbm-delete-node v g ≡ let g=m1.delete v g in
    m1-mvif (λ- m2. Some (m2.delete v m2)) g

  lemma gbm-delete-node-correct:
    graph-delete-node gbm-α gbm-invar gbm-delete-node
  proof
    fix g v
    assume INV: gbm-invar g
```

```
  note [simp]= gbm-invar-split[OF INV]
  show gbm-α (gbm-delete-node v g) = delete-node v (gbm-α g)
    unfolding gbm-α-def gbm-delete-node-def
    by (auto simp: restrict-map-def option-bind-alt
      m1.correct m2.correct s3.correct m1.map-value-image-filter-correct
      delete-node-def
      split: option.split split-if-asm option.split-asm)

  thus gbm-invar (gbm-delete-node v g)
    unfolding gbm-invar-def
    apply (simp)
    unfolding gbm-α-def gbm-delete-node-def delete-node-def
    apply (auto simp: restrict-map-def option-bind-alt
      m1.correct m2.correct s3.correct m1.map-value-image-filter-correct
      split: option.split split-if-asm option.split-asm elim!: ranE)
    done
qed

definition gbm-add-edge :: ('V,'W,'m1) graph-add-edge where
  gbm-add-edge v e v' g ≡
  let g = (case m1.lookup v' g of
    None ⇒ m1.update v' (m2.empty ()) g | Some - ⇒ g
  ) in
  case m1.lookup v g of
    None ⇒ (m1.update v (m2.sng v' (s3.sng e)) g) |
    Some m2 ⇒ (case m2.lookup v' m2 of
      None ⇒ m1.update v (m2.update v' (s3.sng e) m2) g |
      Some s3 ⇒ m1.update v (m2.update v' (s3.ins e s3) m2) g)


lemma gbm-add-edge-correct:
  graph-add-edge gbm-α gbm-invar gbm-add-edge
proof
  fix g v e v'
  assume INV: gbm-invar g
  note [simp]= gbm-invar-split[OF INV]
  show gbm-α (gbm-add-edge v e v' g) = add-edge v e v' (gbm-α g)
    unfolding gbm-α-def gbm-add-edge-def
    apply (auto simp: m1.correct m2.correct s3.correct
      Let-def
      split: option.split split-if-asm)
    unfolding add-edge-def

    apply (fastforce split: split-if-asm
      simp: m1.correct m2.correct s3.correct
    )+
    done

  thus gbm-invar (gbm-add-edge v e v' g)
```

```
    unfolding gbm-invar-def
    apply (simp)
    unfolding gbm-α-def gbm-add-edge-def
    apply (force simp: m1.correct m2.correct s3.correct
      Let-def
      split: option.split split-if-asm elim!: ranE)
    done
qed


definition gbm-delete-edge :: ('V,'W,'m1) graph-delete-edge where
  gbm-delete-edge v e v' g ≡
  case m1.lookup v g of
    None ⇒ g |
    Some m2 ⇒ (
      case m2.lookup v' m2 of
        None ⇒ g |
        Some s3 ⇒ m1.update v (m2.update v' (s3.delete e s3) m2) g
    )


lemma gbm-delete-edge-correct:
  graph-delete-edge gbm-α gbm-invar gbm-delete-edge
proof
  fix g v e v'
  assume INV: gbm-invar g
  note [simp]= gbm-invar-split[OF INV]
  show gbm-α (gbm-delete-edge v e v' g) = delete-edge v e v' (gbm-α g)
    unfolding gbm-α-def gbm-delete-edge-def delete-edge-def
    apply (auto simp: m1.correct m2.correct s3.correct
      Let-def
      split: option.split split-if-asm)
    done

  thus gbm-invar (gbm-delete-edge v e v' g)
    unfolding gbm-invar-def
    apply (simp)
    unfolding gbm-α-def gbm-delete-edge-def
    apply (auto simp: m1.correct m2.correct s3.correct
      Let-def
      split: option.split split-if-asm elim!: ranE)
    done
qed

definition gbm-nodes-it
  :: ('m1 ⇒ ('V×'m2,'σ) set-iterator) ⇒ ('V,'W,'σ,'m1) graph-nodes-it
  where
  gbm-nodes-it mit g ≡ map-iterator-dom (mit g)
```

lemma *gbm-nodes-it-correct*:
assumes *mit*: *map-iteratei m1.α m1.invar mit*
shows *graph-nodes-it gbm-α gbm-invar* (*gbm-nodes-it mit*)
proof
interpret *m*: *map-iteratei m1.α m1.invar mit* **by** *fact*

fix *g*
assume *gbm-invar g*
hence *MINV*: *map-op-invar m1-ops g* **unfolding** *gbm-invar-def* **by** *auto*
from *map-iterator-dom-correct*[*OF m.iteratei-rule*[*OF MINV*]]
show *set-iterator* (*gbm-nodes-it mit g*) (*nodes* (*gbm-α g*))
unfolding *gbm-nodes-it-def gbm-α-def* **by** *simp*
qed

term *set-iterator-product*

definition *gbm-edges-it*
::
('*m1* ⇒ ('*V*×'*m2*,'*σ*) *set-iterator*) ⇒
('*m2* ⇒ ('*V*×'*s3*,'*σ*) *set-iterator*) ⇒
('*s3* ⇒ ('*W*,'*σ*) *set-iterator*) ⇒
('*V*,'*W*,'*σ*,'*m1*) *graph-edges-it*
where
gbm-edges-it mit1 mit2 sit g ≡ *set-iterator-image*
(λ((*v1*,*m1*),(*v2*,*m2*),*w*). (*v1*,*w*,*v2*))
(*set-iterator-product* (*mit1 g*)
(λ(*v*,*m2*). *set-iterator-product* (*mit2 m2*) (λ(*w*,*s3*). *sit s3*)))

lemma *gbm-edges-it-correct*:
assumes *mit1*: *map-iteratei m1.α m1.invar mit1*
assumes *mit2*: *map-iteratei m2.α m2.invar mit2*
assumes *sit*: *set-iteratei s3.α s3.invar sit*
shows *graph-edges-it gbm-α gbm-invar* (*gbm-edges-it mit1 mit2 sit*)
proof
fix *g*
assume *INV*: *gbm-invar g*

from *INV* **have** *I1*: *m1.invar g* **unfolding** *gbm-invar-def* **by** *auto*
from *INV* **have** *I2*: ⋀*v m2*. (*v*,*m2*)∈*map-to-set* (*m1.α g*) ⟹ *m2.invar m2*
unfolding *gbm-invar-def map-to-set-def*
by (*auto simp*: *ran-def*)
from *INV* **have** *I3*: ⋀*v m2 v*′ *s*. ⟦
(*v*,*m2*)∈*map-to-set* (*m1.α g*);
(*v*′,*s*)∈*map-to-set* (*m2.α m2*)⟧
⟹ *s3.invar s*
unfolding *gbm-invar-def map-to-set-def*
by (*auto simp*: *ran-def*)

```
  show set-iterator (gbm-edges-it mit1 mit2 sit g) (edges (gbm-α g))
    unfolding gbm-edges-it-def
    apply (rule set-iterator-image-correct)
    apply (rule set-iterator-product-correct)
    apply (rule map-iteratei.iteratei-rule[OF mit1])
    apply (rule I1)
    apply (case-tac a)
    apply clarsimp
    apply (rule set-iterator-product-correct)
    apply (drule I2)
    apply (subgoal-tac map-iterator (mit2 ba)
      (map-op-α m2-ops (snd (aa,ba))))
    apply assumption
    apply (simp add: map-iteratei.iteratei-rule[OF mit2])

    apply (case-tac a)
    apply clarsimp
    apply (subgoal-tac set-iterator (sit bb)
      (s3.α (snd (ab,bb))))
    apply assumption
    apply (simp add: set-iteratei.iteratei-rule[OF sit] I3)

    apply (auto simp: inj-on-def map-to-set-def) []

    apply (force simp: gbm-α-def map-to-set-def) []
    done
  qed

definition gbm-succ-it ::
  ('m2 ⇒ ('V×'s3,'σ) set-iterator) ⇒
    ('s3 ⇒ ('W,'σ) set-iterator) ⇒
    ('V,'W,'σ,'m1) graph-succ-it
  where
gbm-succ-it mit2 sit g v ≡ case m1.lookup v g of
  None ⇒ set-iterator-emp |
  Some m2 ⇒
    set-iterator-image (λ((v',m2),w). (w,v'))
      (set-iterator-product (mit2 m2) (λ(v',s). sit s))


lemma gbm-succ-it-correct:
  assumes mit2: map-iteratei m2.α m2.invar mit2
  assumes sit: set-iteratei s3.α s3.invar sit
  shows graph-succ-it gbm-α gbm-invar (gbm-succ-it mit2 sit)
proof
  fix g v
  assume INV: gbm-invar g
  hence I1[simp]: m1.invar g unfolding gbm-invar-def by auto
```

```
    show set-iterator (gbm-succ-it mit2 sit g v) (succ (gbm-α g) v)
    proof (cases m1.lookup v g)
      case None hence (succ (gbm-α g) v) = {}
        unfolding succ-def gbm-α-def by (auto simp: m1.lookup-correct)
      with None show ?thesis unfolding gbm-succ-it-def
        by (auto simp: set-iterator-emp-correct)
    next
      case (Some m2)
      hence [simp]: m2.invar m2 using gbm-invar-split[OF INV]
        by (simp add: m1.lookup-correct)

      from INV Some have
        I2: ⋀v' s. (v', s) ∈ map-to-set (map-op-α m2-ops m2) ⟹ s3.invar s
        unfolding gbm-invar-def
        by (auto simp: map-to-set-def ran-def m1.lookup-correct)

      from Some show ?thesis
        unfolding gbm-succ-it-def apply simp
        apply (rule set-iterator-image-correct)
        apply (rule set-iterator-product-correct)
        apply (rule map-iteratei.iteratei-rule)
        apply (rule mit2)
        apply simp
        apply (case-tac a, simp)
        apply (subgoal-tac set-iterator (sit b) (s3.α (snd (aa, b))))
        apply assumption
        apply simp
        apply (rule set-iteratei.iteratei-rule[OF sit])
        apply (simp add: I2)

        apply (auto simp: inj-on-def map-to-set-def) []

        apply (force simp: succ-def gbm-α-def map-to-set-def m1.lookup-correct)
        done
    qed
  qed


  definition
    gbm-from-list ≡ gga-from-list gbm-empty gbm-add-node gbm-add-edge

  lemmas gbm-from-list-correct = gga-from-list-correct[
    OF gbm-empty-correct gbm-add-node-correct gbm-add-edge-correct,
    folded gbm-from-list-def]

end

end
```

9 Graphs by Hashmaps

theory *HashGraphImpl*
imports *GraphByMap ../Collections/Collections*
begin

Abbreviation: hlg

type-synonym $('V,'E)$ *hlg* =
$('V,('V,'E\ ls)\ HashMap.hashmap)\ HashMap.hashmap$

interpretation *hlg-defs*!: *gbm-defs hm-ops hm-ops ls-ops hh-map-value-image-filter*
using *hh-map-value-image-filter-impl*
apply *intro-locales*
apply (*simp add*: *hm-ops-def*)
done

definition *hlg-α* :: $('V::hashable,'E)$ *hlg* ⇒ $('V,'E)$ *graph*
where *hlg-α* ≡ *hlg-defs.gbm-α*
definition *hlg-invar* ≡ *hlg-defs.gbm-invar*
definition *hlg-empty* ≡ *hlg-defs.gbm-empty*
definition *hlg-add-node* ≡ *hlg-defs.gbm-add-node*
definition *hlg-delete-node* ≡ *hlg-defs.gbm-delete-node*
definition *hlg-add-edge* ≡ *hlg-defs.gbm-add-edge*
definition *hlg-delete-edge* ≡ *hlg-defs.gbm-delete-edge*
definition *hlg-from-list* ≡ *hlg-defs.gbm-from-list*

definition *hlg-nodes-it* ≡ *hlg-defs.gbm-nodes-it hm-iteratei*
definition *hlg-edges-it* ≡ *hlg-defs.gbm-edges-it hm-iteratei hm-iteratei ls-iteratei*
definition *hlg-succ-it* ≡ *hlg-defs.gbm-succ-it hm-iteratei ls-iteratei*

definition *hlg-to-list* ≡ *gga-to-list hlg-nodes-it hlg-edges-it*

lemmas *hlg-gbm-defs* =
hlg-defs.gbm-empty-def [*abs-def*]
hlg-defs.gbm-add-node-def [*abs-def*]
hlg-defs.gbm-delete-node-def [*abs-def*]
hlg-defs.gbm-add-edge-def [*abs-def*]
hlg-defs.gbm-delete-edge-def [*abs-def*]
hlg-defs.gbm-from-list-def [*abs-def*]
hlg-defs.gbm-nodes-it-def [*abs-def*]
hlg-defs.gbm-succ-it-def [*abs-def*]
hlg-defs.gbm-edges-it-def [*abs-def*]

lemmas *hlg-defs* =
hlg-α-def
hlg-invar-def
hlg-empty-def
hlg-add-node-def

hlg-delete-node-def
hlg-add-edge-def
hlg-delete-edge-def
hlg-from-list-def
hlg-to-list-def
hlg-nodes-it-def
hlg-edges-it-def
hlg-succ-it-def

lemmas [*code*] = *hlg-defs*[*unfolded hlg-gbm-defs*]

lemmas *hlg-empty-impl* = *hlg-defs.gbm-empty-correct*[*folded hlg-defs*]
interpretation *hlg*!: *graph-empty hlg-α hlg-invar hlg-empty*
using *hlg-empty-impl* **.**
lemmas *hlg-add-node-impl* = *hlg-defs.gbm-add-node-correct*[*folded hlg-defs*]
interpretation *hlg*!: *graph-add-node hlg-α hlg-invar hlg-add-node*
using *hlg-add-node-impl* **.**
lemmas *hlg-delete-node-impl* = *hlg-defs.gbm-delete-node-correct*[*folded hlg-defs*]
interpretation *hlg*!: *graph-delete-node hlg-α hlg-invar hlg-delete-node*
using *hlg-delete-node-impl* **.**
lemmas *hlg-add-edge-impl* = *hlg-defs.gbm-add-edge-correct*[*folded hlg-defs*]
interpretation *hlg*!: *graph-add-edge hlg-α hlg-invar hlg-add-edge*
using *hlg-add-edge-impl* **.**
lemmas *hlg-delete-edge-impl* = *hlg-defs.gbm-delete-edge-correct*[*folded hlg-defs*]
interpretation *hlg*!: *graph-delete-edge hlg-α hlg-invar hlg-delete-edge*
using *hlg-delete-edge-impl* **.**
lemmas *hlg-from-list-impl* = *hlg-defs.gbm-from-list-correct*[*folded hlg-defs*]
interpretation *hlg*!: *graph-from-list hlg-α hlg-invar hlg-from-list*
using *hlg-from-list-impl* **.**

lemmas *hlg-nodes-it-impl* = *hlg-defs.gbm-nodes-it-correct*[
folded hlg-defs,
unfolded hm-ops-def, *simplified*, *OF hm-iteratei-impl*, *folded hlg-defs*
]
interpretation *hlg*!: *graph-nodes-it hlg-α hlg-invar hlg-nodes-it*
using *hlg-nodes-it-impl* **.**

lemmas *hlg-edges-it-impl* = *hlg-defs.gbm-edges-it-correct*[*folded hlg-defs*,
unfolded hm-ops-def ls-ops-def, *simplified*,
OF hm-iteratei-impl hm-iteratei-impl ls-iteratei-impl, *folded hlg-defs*]
interpretation *hlg*!: *graph-edges-it hlg-α hlg-invar hlg-edges-it*
using *hlg-edges-it-impl* **.**

lemmas *hlg-succ-it-impl* = *hlg-defs.gbm-succ-it-correct*[*of hm-iteratei ls-iteratei*,
folded hlg-defs, *unfolded hm-ops-def ls-ops-def*, *simplified*,
OF hm-iteratei-impl ls-iteratei-impl]
interpretation *hlg*!: *graph-succ-it hlg-α hlg-invar hlg-succ-it*

using *hlg-succ-it-impl* **.**

lemmas *hlg-to-list-impl* = *gga-to-list-correct*[*OF*
hlg-nodes-it-impl hlg-edges-it-impl, *simplified hlg-defs*[*symmetric*]]
interpretation *hlg*!: *graph-to-list hlg-α hlg-invar hlg-to-list*
using *hlg-to-list-impl* **.**

definition *hlg-ops* ≡ (|
gop-α = *hlg-α*,
gop-invar = *hlg-invar*,
gop-empty = *hlg-empty*,
gop-add-node = *hlg-add-node*,
gop-delete-node = *hlg-delete-node*,
gop-add-edge = *hlg-add-edge*,
gop-delete-edge = *hlg-delete-edge*,
gop-from-list = *hlg-from-list*,
gop-to-list = *hlg-to-list*
|)

lemma *hlg-ops-unfold*[*code-unfold*]:
gop-α hlg-ops = *hlg-α*
gop-invar hlg-ops = *hlg-invar*
gop-empty hlg-ops = *hlg-empty*
gop-add-node hlg-ops = *hlg-add-node*
gop-delete-node hlg-ops = *hlg-delete-node*
gop-add-edge hlg-ops = *hlg-add-edge*
gop-delete-edge hlg-ops = *hlg-delete-edge*
gop-from-list hlg-ops = *hlg-from-list*
gop-to-list hlg-ops = *hlg-to-list*
by (*simp-all add*: *hlg-ops-def*)

lemma *hlgr-impl*: *StdGraph hlg-ops*
apply (*rule StdGraph.intro*)
apply (*simp-all add*: *hlg-ops-def*)
apply *unfold-locales*
done

interpretation *hlgr*!: *StdGraph hlg-ops* **using** *hlgr-impl* **.**

end

10 Implementation of Dijkstra's-Algorithm using the ICF

theory *Dijkstra-Impl*
imports *Dijkstra GraphSpec HashGraphImpl* ~~/*src*/*HOL*/*Library*/*Code-Target-Numeral*
begin

In this second refinement step, we use interfaces from the Isabelle Collection Framework (ICF) to implement the priority queue and the result map. Moreover, we use a graph interface (that is not contained in the ICF, but in this development) to represent the graph.

The data types of the first refinement step were designed to fit the abstract data types of the used ICF-interfaces, which makes this refinement quite straightforward.

Finally, we instantiate the ICF-interfaces by concrete implementations, obtaining an executable algorithm, for that we generate code using Isabelle/HOL's code generator.

Due to idiosyncrasies of the code generator, we have to split the locale for the definitions, and the locale that assumes the preconditions.

```
locale dijkstraC-def =
  g!: StdGraphDefs g-ops +
  mr!: StdMapDefs mr-ops +
  qw!: StdUprioDefs qw-ops
  for g-ops :: ('V,'W::weight,'G,'moreg) graph-ops-scheme
  and mr-ops :: ('V, (('V,'W) path × 'W), 'mr, 'more-mr) map-ops-scheme
  and qw-ops :: ('V ,'W infty,'qw, 'more-qw) uprio-ops-scheme
  and nodes-it :: ('V, 'W, 'qw, 'G) graph-nodes-it
  and succ-it :: ('V, 'W, ('qw×'mr), 'G) graph-succ-it
+
  fixes v0 :: 'V
  fixes ga :: ('V,'W) graph
  fixes g :: 'G
begin
  definition αsc == map-pair qw.α mr.α
  definition dinvarC-add == λ(wl,res). qw.invar wl ∧ mr.invar res

  definition cdinit :: ('qw×'mr) nres where
    cdinit ≡ do {
      wl ← FOREACH (nodes ga)
        (λv wl. RETURN (qw.insert wl v Weight.Infty)) (qw.empty ());
      RETURN (qw.insert wl v0 (Num 0),mr.sng v0 ([],0))
    }

  definition cpop-min :: ('qw×'mr) ⇒ ('V×'W infty×('qw×'mr)) nres where
    cpop-min σ ≡ do {
      let (wl,res) = σ;
      let (v,w,wl')=qw.pop wl;
      RETURN (v,w,(wl',res))
    }

  definition cupdate :: 'V ⇒ 'W infty ⇒ ('qw×'mr) ⇒ ('qw×'mr) nres where
    cupdate v wv σ = do {
      ASSERT (dinvarC-add σ);
      let (wl,res)=σ;
```

```
    let pv=mpath' (mr.lookup v res);
    FOREACH (succ ga v) (λ(w',v') (wl,res).
      if (wv + Num w' < mpath-weight' (mr.lookup v' res)) then do {
        RETURN (qw.insert wl v' (wv+Num w'),
              mr.update v' ((v,w',v')#the pv,val wv + w') res)
      } else RETURN (wl,res)
    ) (wl,res)
  }

  definition cdijkstra where
    cdijkstra ≡ do {
      σ0 ← cdinit;
      (-,res) ← WHILE_T (λ(wl,-). ¬ qw.isEmpty wl)
          (λσ. do { (v,wv,σ') ← cpop-min σ; cupdate v wv σ' } )
          σ0;
      RETURN res
    }

end

locale dijkstraC =
  dijkstraC-def g-ops mr-ops qw-ops nodes-it succ-it v0 ga g +
  Dijkstra ga v0 +
  g!: StdGraph g-ops +
  mr!: StdMap mr-ops +
  qw!: StdUprio qw-ops +
  g!: graph-nodes-it g.α g.invar nodes-it +
  g!: graph-succ-it g.α g.invar succ-it
  for g-ops :: ('V,'W::weight,'G,'moreg) graph-ops-scheme
  and mr-ops :: ('V, (('V,'W) path × 'W), 'mr,'more-mr) map-ops-scheme
  and qw-ops :: ('V ,'W infty,'qw,'more-qw) uprio-ops-scheme
  and nodes-it :: ('V, 'W, 'qw, 'G) graph-nodes-it
  and succ-it :: ('V, 'W, ('qw×'mr), 'G) graph-succ-it
  and ga::('V,'W) graph and V :: 'V set and v0::'V and g :: 'G+
  assumes in-abs: g.α g = ga
  assumes g-invar[simp, intro!]: g.invar g
begin

  schematic-lemma cdinit-refines:
    notes [refine] = inj-on-id
    shows cdinit ≤⇓?R mdinit
    unfolding cdinit-def mdinit-def
    apply (refine-rcg)
    apply (refine-dref-type)
    apply (simp-all add: αsc-def dinvarC-add-def
      qw.correct mr.correct)
    done

  schematic-lemma cpop-min-refines:
```

```
  (σ,σ') ∈ build-rel αsc dinvarC-add
    ⟹ cpop-min σ ≤ ⇓?R (mpop-min σ')
  unfolding cpop-min-def mpop-min-def
  apply (refine-rcg)
  apply (refine-dref-type)
  apply (simp add: αsc-def dinvarC-add-def)
  apply (simp add: αsc-def dinvarC-add-def)
  done

schematic-lemma cupdate-refines:
  notes [refine] = inj-on-id
  shows (σ,σ')∈build-rel αsc dinvarC-add ⟹ v=v' ⟹ wv=wv' ⟹
  cupdate v wv σ ≤ ⇓?R (mupdate v' wv' σ')
  unfolding cupdate-def mupdate-def
  apply (refine-rcg)
  apply (refine-dref-type)
  apply (simp-all add: αsc-def dinvarC-add-def
    qw.correct mr.correct)
  done

lemma cdijkstra-refines: cdijkstra ≤ ⇓(build-rel mr.α mr.invar) mdijkstra
proof −
  note [refine] = cdinit-refines cpop-min-refines cupdate-refines
  show ?thesis
    unfolding cdijkstra-def mdijkstra-def
    apply (refine-rcg)

    apply (simp-all
      split: prod.split prod.split-asm
      add: qw.correct mr.correct dinvarC-add-def αsc-def)
    done
qed

schematic-lemma idijkstra-refines-aux:
  notes [refine-transfer] = g-invar
  shows RETURN ?f ≤ cdijkstra
  unfolding cdijkstra-def cdinit-def cpop-min-def cupdate-def
  unfolding in-abs[symmetric]
  apply (refine-transfer)
  done

thm idijkstra-refines-aux[no-vars]

end
```

Copy-Pasted from the above lemma:

```
definition (in dijkstraC-def) idijkstra ≡
(let x = let x = nodes-it g (λ-. True)
                (λx s. let s = s in upr-insert qw-ops s x infty.Infty)
```

```
                (upr-empty qw-ops ())
          in (upr-insert qw-ops x v0 (Num (0::'W)),
              map-op-sng mr-ops v0 ([], 0::'W));
      (a, b) =
        while (λ(wl, -). ¬ upr-isEmpty qw-ops wl)
         (λxa. let (a, b) =
                    let (a, b) = xa; (aa, ba) = upr-pop qw-ops a
                    in case ba of (ab, bb) ⇒ (aa, ab, bb, b)
               in case b of
                  (aa, ba) ⇒
                    let (ab, bb) = ba;
                        xd = mpath' (map-op-lookup mr-ops a bb)
                    in succ-it g a (λ-. True)
                        (λx s. let s = s
                               in case x of
                                  (ac, bc) ⇒
                                    case s of
                                    (ad, bd) ⇒
if aa + Num ac < mpath-weight' (map-op-lookup mr-ops bc bd)
then (upr-insert qw-ops ad bc (aa + Num ac),
      map-op-update mr-ops bc ((a, ac, bc) # the xd, val aa + ac) bd)
else (ad, bd))
                        (ab, bb))
          x
  in b)
```

Example instantiation with HashSet.based graph, red-black-tree based result map, and finger-tree based priority queue.

```
definition dijkstra-impl ::
  ('V::{hashable,linorder},'W::weight) hlg ⇒ 'V
  ⇒ ('V,('V,'W) path × 'W) rm where
  dijkstra-impl g v0 ≡
    dijkstraC-def.idijkstra rm-ops aluprioi-ops hlg-nodes-it hlg-succ-it v0 g

lemmas [code] = dijkstraC-def.idijkstra-def
```

Code can be exported to all available target languages:

```
export-code dijkstra-impl in SML file –
export-code dijkstra-impl in OCaml file –
export-code dijkstra-impl in Haskell file –
export-code dijkstra-impl in Scala file –


context dijkstraC
begin
  lemma idijkstra-refines: RETURN idijkstra ≤ cdijkstra
    by (rule idijkstra-refines-aux[folded idijkstra-def])
```

The following theorem states correctness of the algorithm independent from the refinement framework.

Intuitively, the first goal states that the abstraction of the returned result is correct, the second goal states that the result datastructure satisfies its invariant, and the third goal states that the cached weights in the returned result are correct.

Note that this is the main theorem for a user of Dijkstra's algorithm in some bigger context. It may also be specialized for concrete instances of the implementation, as exemplarily done below.

```
theorem (in dijkstraC) idijkstra-correct:
  shows
  weighted-graph.is-shortest-path-map ga v0 (αr (mr.α idijkstra)) (is ?G1)
  and mr.invar idijkstra (is ?G2)
  and res-invarm (mr.α idijkstra) (is ?G3)
proof −
  note idijkstra-refines
  also note cdijkstra-refines
  also note mdijkstra-refines
  finally have Z: RETURN idijkstra ≤
    ⇓(build-rel (αr ∘ mr.α) (λm. mr.invar m ∧ res-invarm (mr.α m)))
      dijkstra′
    apply (subst (asm) conc-fun-chain)
    apply rule
    apply (simp only: br-chain)
    done
  also note dijkstra′-refines[simplified]
  also note dijkstra-correct
  finally show ?G1 ?G2 ?G3
    by (auto elim: RETURN-ref-SPECD)
qed

end
```

We also specialize the correctness theorem. Note that the data structure invariant for red-black trees is encoded into the type, and hence λ-. *True* is constantly *True*. Hence, it is not stated in this theorem.

```
theorem dijkstra-impl-correct:
  assumes INV: hlg-invar g
  assumes V0: v0 ∈ nodes (hlg-α g)
  assumes nonneg-weights: ⋀v w v′. (v,w,v′)∈edges (hlg-α g) ⟹ 0≤w
  shows
  weighted-graph.is-shortest-path-map (hlg-α g) v0
     (Dijkstra.αr (rm-α (dijkstra-impl g v0))) (is ?G1)
  and Dijkstra.res-invarm (rm-α (dijkstra-impl g v0)) (is ?G2)
proof −
  interpret hlgv!: valid-graph hlg-α g using hlg.valid INV .
  interpret hlgv!:
```

graph-nodes-it gop-α hlg-ops gop-invar hlg-ops hlg-nodes-it
by (*auto simp*: *hlg-nodes-it-impl hlg-ops-def*)
interpret *hlgv*!: *graph-succ-it gop-α hlg-ops gop-invar hlg-ops hlg-succ-it*
by (*auto simp*: *hlg-succ-it-impl hlg-ops-def*)

interpret *dc*: *dijkstraC hlg-ops rm-ops aluprioi-ops hlg-nodes-it hlg-succ-it*
hlg-α g
apply *unfold-locales*
apply (*simp-all add*: *hlg.finite INV V0 hlg-ops-def nonneg-weights*)
done

from *dc.idijkstra-correct* **show** *?G1 ?G2*
by (*auto simp*: *rm-ops-def dijkstra-impl-def*)
qed

end

11 Implementation of Dijkstra's-Algorithm using Automatic Determinization

theory *Dijkstra-Impl-Adet*
imports *Dijkstra GraphSpec HashGraphImpl ~~/src/HOL/Library/Code-Target-Numeral*
../Refine-Monadic/Autoref-Collection-Bindings
begin

locale *dijkstraC-def* =
g!: *StdGraphDefs g-ops* +
mr!: *StdMapDefs mr-ops* +
qw!: *StdUprioDefs qw-ops*
for *g-ops* :: ('*V*,'*W*::*weight*,'*G*,'*moreg*) *graph-ops-scheme*
and *mr-ops* :: ('*V*, (('*V*,'*W*) *path* × '*W*), '*mr*, '*more-mr*) *map-ops-scheme*
and *qw-ops* :: ('*V* ,'*W infty*,'*qw*, '*more-qw*) *uprio-ops-scheme* +
fixes *nodes-it* :: ('*V*, '*W*, '*qw*, '*G*) *graph-nodes-it*
fixes *succ-it* :: ('*V*, '*W*, '*qw*×'*mr*, '*G*) *graph-succ-it*
fixes *v0* :: '*V*
fixes *ga* :: ('*V*,'*W*) *graph*
fixes *g* :: '*G*
begin
end

locale *dijkstraC* =
dijkstraC-def g-ops mr-ops qw-ops nodes-it succ-it v0 ga g +
Dijkstra ga v0 +
g!: *StdGraph g-ops* +
mr!: *StdMap mr-ops* +
qw!: *StdUprio qw-ops* +

```
  g!: graph-nodes-it g.α g.invar nodes-it +
  g!: graph-succ-it g.α g.invar succ-it
  for g-ops :: ('V,'W::weight,'G,'moreg) graph-ops-scheme
  and mr-ops :: ('V, (('V,'W) path × 'W), 'mr,'more-mr) map-ops-scheme
  and qw-ops :: ('V ,'W infty,'qw,'more-qw) uprio-ops-scheme
  and nodes-it :: ('V, 'W, 'qw, 'G) graph-nodes-it
  and succ-it :: ('V, 'W, 'qw×'mr, 'G) graph-succ-it
  and ga::('V,'W) graph and V :: 'V set and v0::'V and g :: 'G+
  assumes in-abs: g.α g = ga
  assumes g-invar[simp, intro!]: g.invar g
begin

  lemma ga-trans: (g,ga)∈br g.α g.invar using in-abs by auto

  lemma ga-nodes-trans: (nodes (g.α g),nodes ga)∈Id using in-abs by auto
  lemma ga-succs-trans: (v,v')∈Id ⟹ (succ (g.α g) v,succ ga v')∈Id
    using in-abs by auto

  schematic-lemma cdijkstra-refines-aux:
    notes [autoref-spec] =
      spec-R[where 'c=bool and 'a=bool]
      spec-R[where 'c='V and 'a='V]
      spec-R[where 'c='W and 'a='W]
      spec-R[where 'c='W infty and 'a='W infty]

      spec-R[where 'c='mr and 'a='V⇀(('V,'W) path × 'W)]
      spec-R[where 'c='qw and 'a='V⇀'W infty]
      spec-R[where 'c=('V,'W) path and 'a=('V,'W) path]
      spec-R[where 'c=(('V,'W) path × 'W) and 'a=(('V,'W) path × 'W)]
      spec-R[where 'c=('V,'W) path option and 'a=('V,'W) path option]
      spec-R[where 'c=(('V,'W) path × 'W) option and
        'a=(('V,'W) path × 'W) option]

    notes [autoref-ex] = mr.map-lookup-t ga-succs-trans


    shows RETURN (?cdijkstra::'mr) ≤⇓?R  mdijkstra
    apply (rule autoref-det-optI)
    unfolding mdijkstra-def mdinit-def mpop-min-def prio-pop-min-def
      mupdate-def
    using ga-trans ga-nodes-trans IdI[of v0]
    apply −
    apply (refine-autoref)
    using g-invar
    apply (refine-transfer) []
    unfolding prod-case-rename
    apply (rule refl)
    done
```

```
  notepad begin
    note [[show-types]]
    thm cdijkstra-refines-aux
  end
end

definition (in dijkstraC-def) cdijkstra ≡
 (let (a::'qw, b::'mr) =
       let x::'qw =
            (nodes-it::'G ⇒ ('qw ⇒ bool) ⇒ ('V ⇒ 'qw ⇒ 'qw) ⇒ 'qw ⇒ 'qw)
             (g::'G) (λ-::'qw. True)
             (λ(x::'V) s::'qw.
                 let s::'qw = s
                 in upr-insert
                     (qw-ops::('V, 'W infty, 'qw,
                               'more-qw) uprio-ops-scheme)
                     s x infty.Infty)
             (upr-empty qw-ops ())
       in (upr-insert qw-ops x (v0::'V) (Num (0::'W)),
           map-op-update
            (mr-ops::('V, ('V × 'W × 'V) list × 'W, 'mr,
                      'more-mr) map-ops-scheme)
            v0 ([], 0::'W) (map-op-empty mr-ops ())))
  in Let (while ((λa::'qw. ¬ upr-isEmpty qw-ops a) ∘ fst)
          (λ(aa::'qw, ba::'mr).
             let (ab::'V, bb::'W infty × 'qw × 'mr) =
                  let (ab::'qw, bb::'mr) = (aa, ba);
                      (ac::'V, bc::'W infty × 'qw) = upr-pop qw-ops ab
                  in case bc of (ad::'W infty, bd::'qw) ⇒ (ac, ad, bd, bb)
             in case bb of
                (ac::'W infty, ad::'qw, bd::'mr) ⇒
                  let (ae::'qw, be::'mr) = (ad, bd);
                      xd::('V × 'W × 'V) list option =
                        mpath' (map-op-lookup mr-ops ab be)
                  in (succ-it::'G ⇒ 'V ⇒ ('qw × 'mr ⇒ bool)
    ⇒ ('W × 'V ⇒ 'qw × 'mr ⇒ 'qw × 'mr) ⇒ 'qw × 'mr ⇒ 'qw × 'mr)
                       g ab (λ-::'qw × 'mr. True)
                       (λ(x::'W × 'V) s::'qw × 'mr.
                           let (af::'qw, bf::'mr) = s
                           in case x of
                              (ag::'W, bg::'V) ⇒
                                if ac + Num ag
                                   < mpath-weight'
  (map-op-lookup mr-ops bg bf)
                                then (upr-insert qw-ops af bg
  (ac + Num ag),
  map-op-update mr-ops bg ((ab, ag, bg) # the xd, val ac + ag) bf)
                                else (af, bf))
                       (ae, be))
```

```
      (a, b))
  ((λba::'mr. ba) ∘ snd))
```

```
context dijkstraC
begin
  lemma cdijkstra-refines:
    RETURN cdijkstra ≤ ⇓(build-rel mr.α mr.invar) mdijkstra
    by (rule cdijkstra-refines-aux[folded cdijkstra-def])
```

The following theorem states correctness of the algorithm independent from the refinement framework.

Intuitively, the first goal states that the abstraction of the returned result is correct, the second goal states that the result datastructure satisfies its invariant, and the third goal states that the cached weights in the returned result are correct.

Note that this is the main theorem for a user of Dijkstra's algorithm in some bigger context. It may also be specialized for concrete instances of the implementation, as exemplarily done below.

```
  theorem cdijkstra-correct:
    shows
    weighted-graph.is-shortest-path-map ga v0 (αr (mr.α cdijkstra)) (is ?G1)
    and mr.invar cdijkstra (is ?G2)
    and res-invarm (mr.α cdijkstra) (is ?G3)
  proof −
    note cdijkstra-refines
    also note mdijkstra-refines
    finally have Z: RETURN cdijkstra ≤
      ⇓(build-rel (αr ∘ mr.α) (λm. mr.invar m ∧ res-invarm (mr.α m)))
        dijkstra'
      apply (subst (asm) conc-fun-chain)
      apply rule
      apply (simp only: br-chain)
      done
    also note dijkstra'-refines[simplified]
    also note dijkstra-correct
    finally show ?G1 ?G2 ?G3 by (auto elim: RETURN-ref-SPECD)
  qed

end
```

Example instantiation with HashSet.based graph, red-black-tree based result map, and finger-tree based priority queue.

```
definition dijkstra-impl ::
  ('V::{hashable,linorder},'W::weight) hlg ⇒ 'V
  ⇒ ('V,('V,'W) path × 'W) rm where
```

```
  dijkstra-impl g v0 ≡
    dijkstraC-def.cdijkstra rm-ops aluprioi-ops hlg-nodes-it hlg-succ-it v0 g
```

We also specialize the correctness theorem. Note that the data structure invariant for red-black trees is encoded into the type, and hence λ-. *True* is constantly *True*. Hence, it is not stated in this theorem.

```
theorem dijkstra-impl-correct:
  assumes INV: hlg-invar g
  assumes V0: v0 ∈ nodes (hlg-α g)
  assumes nonneg-weights: ⋀v w v'. (v,w,v')∈edges (hlg-α g) ⟹ 0≤w
  shows
  weighted-graph.is-shortest-path-map (hlg-α g) v0
      (Dijkstra.αr (rm-α (dijkstra-impl g v0))) (is ?G1)
  and Dijkstra.res-invarm (rm-α (dijkstra-impl g v0)) (is ?G2)
proof −
  interpret hlgv!: valid-graph hlg-α g using hlg.valid INV .
  interpret hlgv!:
    graph-nodes-it gop-α hlg-ops gop-invar hlg-ops hlg-nodes-it
    by (auto simp: hlg-nodes-it-impl hlg-ops-def)
  interpret hlgv!: graph-succ-it gop-α hlg-ops gop-invar hlg-ops hlg-succ-it
    by (auto simp: hlg-succ-it-impl hlg-ops-def)

  interpret dc: dijkstraC hlg-ops rm-ops aluprioi-ops hlg-nodes-it hlg-succ-it
    hlg-α g
    apply unfold-locales
    apply (simp-all add: hlg.finite INV V0 hlg-ops-def nonneg-weights)
    done

  from dc.cdijkstra-correct show ?G1 ?G2
    by (auto simp: rm-ops-def dijkstra-impl-def)
qed

lemmas [code] = dijkstraC-def.cdijkstra-def
```

Code can be exported to all available target languages:

```
export-code dijkstra-impl in SML file −
export-code dijkstra-impl in OCaml file −
export-code dijkstra-impl in Haskell file −
export-code dijkstra-impl in Scala file −

end
```

12 Performance Test

```
theory Test
  imports Dijkstra-Impl
begin
```

In this theory, we test our implementation of Dijkstra's algorithm for larger, randomly generated graphs.

Simple linear congruence generator for (low-quality) random numbers:

definition *lcg-next s* = ((*81*::*nat*)∗*s* + *173*) *mod 268435456*

Generate a complete graph over the given number of vertices, with random weights:

```
definition ran-graph :: nat ⇒ nat ⇒ (nat list×(nat × nat × nat) list) where
  ran-graph vertices seed ==
    ([0::nat..<vertices],fst
     (while (λ (g,v,s). v < vertices)
     (λ (g,v,s).
     let (g'',v'',s'') = (while (λ (g',v',s'). v' < vertices)
      (λ (g',v',s'). ((v,s',v')#g',v'+1,lcg-next s'))
      (g,0,s))
     in (g'',v+1,s''))
     ([],0,lcg-next seed)))
```

To experiment with the exported code, we fix the node type to natural numbers, and add a from-list conversion:

```
type-synonym nat-res = (nat,((nat,nat) path × nat)) rm
type-synonym nat-list-res = (nat × (nat,nat) path × nat) list

definition nat-dijkstra :: (nat,nat) hlg ⇒ nat ⇒ nat-res where
  nat-dijkstra ≡ dijkstra-impl

definition hlg-from-list-nat :: (nat,nat) adj-list ⇒(nat,nat) hlg where
  hlg-from-list-nat ≡ hlg-from-list

definition
  nat-res-to-list :: nat-res ⇒ nat-list-res
  where nat-res-to-list ≡ rm-to-list

value nat-res-to-list (nat-dijkstra (hlg-from-list (ran-graph 4 8912)) 0)

ML-val ⟪
let
  (∗ Configuration of test: ∗)
  val vertices = @{code nat-of-integer} 1000; (∗ Number of vertices ∗)
  val seed = @{code nat-of-integer} 123454; (∗ Seed for random number generator
∗)
  val cfg-print-paths = true; (∗ Whether to output complete paths ∗)
  val cfg-print-res = true; (∗ Whether to output result at all ∗)

  (∗ Internals ∗)
  fun string-of-edge (u,(w,v)) =
    ( ^ string-of-int (@{code integer-of-nat} u)
```

```
    ^ , ^ string-of-int (@{code integer-of-nat} w)
    ^ , ^ string-of-int (@{code integer-of-nat} v) ^ );

  fun print-entry (dest,(path,weight)) =
    writeln (string-of-int (@{code integer-of-nat} dest)
    ^ :  ^ string-of-int (@{code integer-of-nat} weight) ^
    ( if cfg-print-paths then
        via [ ^ commas (map string-of-edge (rev path)) ^ ]
      else
    )
  );

  fun print-res [] = ()
    | print-res (a::l) = let val - = print-entry a in print-res l end;

  val start = Time.now();
  val graph = @{code hlg-from-list-nat} (@{code ran-graph} vertices seed);
  val rt1 = Time.toMilliseconds (Time.now() - start);

  val start = Time.now();
  val res = @{code nat-dijkstra} graph (@{code nat-of-integer} 0);
  val rt2 = Time.toMilliseconds (Time.now() - start);
in
  writeln (string-of-int (@{code integer-of-nat} vertices) ^ vertices:
  ^ string-of-int rt2 ^ ms +
  ^ string-of-int rt1 ^ ms to create graph =
  ^ string-of-int (rt1+rt2) ^ ms);

  if cfg-print-res then
    print-res (@{code nat-res-to-list} res)
  else ()
end;
⟩⟩
```

end

References

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

[2] F. Haftmann. *Code Generation from Specifications in Higher Order Logic*. PhD thesis, Technische Universität München, 2009.

[3] F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In *Functional and Logic Programming (FLOPS 2010)*, LNCS. Springer, 2010.

[4] P. Lammich. Collections framework. In G. Klein, T. Nipkow, and L. Paulson, editors, *Archive of Formal Proofs*. http://afp.sf.net/entries/collections.shtml, Dec. 2009. Formal proof development.

[5] P. Lammich. Refinement for monadic programs. 2011. Submitted to AFP.

[6] P. Lammich and A. Lochbihler. The Isabelle collections framework. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 339–354. Springer, 2010.

[7] B. Nordhoff, S. Körner, and P. Lammich. Finger trees. In G. Klein, T. Nipkow, and L. Paulson, editors, *Archive of Formal Proofs*. http://afp.sf.net/entries/Tree-Automata.shtml, Oct. 2010. Formal proof development.