Dijkstra’s Algorithm

Benedikt Nordhoff Peter Lammich

June 11, 2019

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 3
2 Miscellaneous Lemmas 3
3 Graphs 4
  3.1 Definitions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5
  3.2 Basic operations on Graphs . . . . . . . . . . . . . . . . . . . . . 5
  3.3 Paths . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6
    3.3.1 Splitting Paths . . . . . . . . . . . . . . . . . . . . . . . . . . . 7
  3.4 Weighted Graphs . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
4 Weights for Dijkstra’s Algorithm 9
  4.1 Type Classes Setup . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
  4.2 Adding Infinity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 10
    4.2.1 Unboxing . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
5 Dijkstra’s Algorithm 12
  5.1 Graph’s for Dijkstra’s Algorithm . . . . . . . . . . . . . . . . . . . . . 12
  5.2 Specification of Correct Result . . . . . . . . . . . . . . . . . . . . . 12
  5.3 Dijkstra’s Algorithm . . . . . . . . . . . . . . . . . . . . . . . . . . . 13
  5.4 Structural Refinement of Update . . . . . . . . . . . . . . . . . . . . 15
  5.5 Refinement to Cached Weights . . . . . . . . . . . . . . . . . . . . . 16
6  Graph Interface 19
   6.1  Adjacency Lists 22
   6.2  Record Based Interface 22
   6.3  Refinement Framework Bindings 23

7  Generic Algorithms for Graphs 24

8  Implementing Graphs by Maps 26

9  Graphs by Hashmaps 31

10 Implementation of Dijkstra’s-Algorithm using the ICF 32

11 Implementation of Dijkstra’s-Algorithm using Automatic Determinization 36
   11.1  Setup 36
      11.1.1  Infinity 36
      11.1.2  Graph 38
   11.2  Refinement 39

12 Performance Test 42
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∈S; ∀x′∈S. f x ≤ f x′ ] ⇒ x ∈ least-map f S

lemma least-map-subset: least-map f S ⊆ S
⟨proof⟩

lemmas least-map-elemD = subsetD[OF least-map-subset]

lemma least-map-leD:
  assumes x ∈ least-map f S
  assumes y∈S
  shows f x ≤ f y
⟨proof⟩
lemma least-map-empty [simp]: least-map f {} = {}
⟨proof⟩
lemma least-map-singleton [simp]: least-map (f :: 'a ⇒ 'b::order) {x} = {x}
⟨proof⟩
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)
⟨proof⟩
lemma least-map-insert-nmin:
[ x ∈ least-map f S; f x ≤ f a ] ⇒ x ∈ least-map f (insert a S)
⟨proof⟩

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
⟨proof⟩
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 ⊆ V × W × V, where W are the
dge labels.
3.1 Definitions

A graph is represented by a record.

record ('v,'w) graph =
  nodes :: 'v set
  edges :: ('v × 'w × '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 ⊆ nodes G
                  snd'snd'edges G ⊆ nodes G
begin
  abbreviation V ≡ nodes G
  abbreviation E ≡ edges G

  lemma E-validD: assumes (v,e,v')∈E
                   shows v∈V v'∈V
  ⟨proof⟩
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** \( \text{succ} :: ('v,'w) \text{ graph} \Rightarrow 'v \Rightarrow (\text{'w}'\times'v) \text{ set} \)

where \( \text{succ } G \ v \equiv \{(w,v'),(v,w,v'):\epsilon_{\text{edges } G}\} \)

Now follow some simplification lemmas.

**lemma** empty-valid[simp]: \( \text{valid-graph } \text{empty} \)

**proof**

**lemma** add-node-valid[simp]: \( \text{assumes } \text{valid-graph } g \)

shows \( \text{valid-graph } (\text{add-node } v \ g) \)

**proof**

**lemma** delete-node-valid[simp]: \( \text{assumes } \text{valid-graph } g \)

shows \( \text{valid-graph } (\text{delete-node } v \ g) \)

**proof**

**lemma** add-edge-valid[simp]: \( \text{assumes } \text{valid-graph } g \)

shows \( \text{valid-graph } (\text{add-edge } v \ e \ v' \ g) \)

**proof**

**lemma** delete-edge-valid[simp]: \( \text{assumes } \text{valid-graph } g \)

shows \( \text{valid-graph } (\text{delete-edge } v \ e \ v' \ g) \)

**proof**

**lemma** succ-finite[simp, intro]: \( \text{finite } (\text{edges } G) \Rightarrow \text{finite } (\text{succ } G \ v) \)

**proof**

**lemma** nodes-empty[simp]: \( \text{nodes } \text{empty} = \{\} \)

**proof**

**lemma** edges-empty[simp]: \( \text{edges } \text{empty} = \{\} \)

**proof**

**lemma** succ-empty[simp]: \( \text{succ } \text{empty } v = \{\} \)

**proof**

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

**proof**

**lemma** nodes-add-edge[simp]: \( \text{nodes } (\text{add-edge } v \ e \ v' \ g) = \text{insert } v \ (\text{insert } v' \ (\text{nodes } g)) \)

**proof**

**lemma** edges-add-edge[simp]: \( \text{edges } (\text{add-edge } v \ e \ v' \ g) = \text{insert } (v,e,v') \ (\text{edges } g) \)

**proof**

**lemma** edges-add-node[simp]: \( \text{edges } (\text{add-node } v \ g) = \text{edges } g \)

**proof**

**lemma** (in valid-graph) succ-subset: \( \text{succ } G \ v \subseteq \text{UNIV} \times V \)

**proof**

3.3 Paths

A path is represented by a list of adjacent edges.

**type-synonym** \(('v,'w) \text{ path} = (\text{'w}'\times'v) \text{ list} \)

**context** valid-graph
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

**Lemma is-path-memb[simp]:**

- is-path v p v' ⇒ v∈V ∧ v'∈V

**Lemma is-path-split:***

- is-path v (p1@p2) v' ←→ (∃ u. is-path v p1 u ∧ is-path u p2 v')

**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'

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 (ww#p) = insert (fst ww) (int-vertices p)  
- int-vertices (p1@p2) = int-vertices p1 ∪ int-vertices p2

**Lemma (in valid-graph) int-vertices-subset:**

- is-path v p v' ⇒ int-vertices p ⊆ V

**Lemma int-vertices-empty[simp]:**

- int-vertices p = [] ←→ p=[]

### 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\in W and v'\notin W
obtains p1 p2 u w u' where
p\in(p\circ(u,w,u')\circ p2) and
int-vertices p1 \subseteq W and u\in W and u'\notin W
(proof)

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'\in W
obtains p1 p2 u where
p\in(p1\circ p2) and
is-path v p1 u and
is-path u p2 v' and
int-vertices p1 \subseteq - W and u\in W
(proof)

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\in int-vertices p
obtains p1 p2 where
p\in(p1\circ p2) and
is-path v p1 u and
u \notin int-vertices p1
(proof)

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 \Rightarrow 'w
where path-weight p \equiv \text{sum-list} (\text{map} (\text{fst} \circ \text{snd}) p)

lemma path-weight-split[simp]:
(path-weight (p1\circ p2)::'w::monoid-add) = path-weight p1 + path-weight p2
(proof)

lemma path-weight-empty[simp]: path-weight [] = 0
(proof)

lemma path-weight-cons[simp]:
(path-weight (e#p)::'w::monoid-add) = \text{fst} (\text{snd} e) + path-weight p
(proof)

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⟩

lemma add-nonpos-nonpos [simp]:
  assumes a ≤ 0 and b ≤ 0 shows a + b ≤ 0
⟨proof⟩

lemma add-nonneg-eq-0-iff:
  assumes x: 0 ≤ x and y: 0 ≤ y
  shows x + y = 0 <-> x = 0 ∧ y = 0
⟨proof⟩

lemma add-incr: 0 ≤ b ⇒ a ≤ a + b
⟨proof⟩

lemma add-incr-left [simp, intro!] : 0 ≤ b ⇒ a ≤ b + a
⟨proof⟩

lemma sum-not-less [simp, intro!] :
  0 ≤ b ⇒ ¬ (a + b < a)
  0 ≤ a ⇒ ¬ (a + b < b)
⟨proof⟩

end

instance nat :: weight ⟨proof⟩
instance int :: weight ⟨proof⟩
instance rat :: weight ⟨proof⟩
instance real :: weight ⟨proof⟩

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

lemma inf-add-left[simp]: top + a = top
  ⟨proof⟩

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

lemma not-less-inf[simp]:
  ¬ (a < top) ⟷ a=top
  ⟨proof⟩

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 ≠ Infty ⟷ Num (val e) = e ⟨proof⟩
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
    ⟨proof⟩

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

lemma [simp]: plus Infty a = Infty ⟨proof⟩

instance ⟨proof⟩
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
  ⟨proof⟩

lemma infty-ord-unbox:
  Num a ≤ Num b  ⟷  a ≤ b
  Num a < Num b  ⟷  a < b
  ⟨proof⟩

lemma infty-plus-unbox:
  Num a + Num b = Num (a+b)
  ⟨proof⟩

lemma infty-zero-unbox:
  Num a = 0  ⟷  a = 0
  Num 0 = 0
  ⟨proof⟩

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
  ⟨proof⟩

lemma num-val-iff [simp]: e≠top  ⇒  Num (val e) = e
  ⟨proof⟩

lemma infty-neE:
  [a≠Infty;  d.  a=Num d  ⇒  P]  ⇒  P
  [a≠top;  d.  a=Num d  ⇒  P]  ⇒  P
proof
end

5 Dijkstra’s Algorithm

theory Dijkstra
imports
Graph
Dijkstra-Misc
Collections.Refine-Dflt-ICF
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 ⇒ (V → (V, W) path) ⇒ bool
where
is-shortest-path-map v0 res ≡ ∀ v ∈ V. (case res v of
  None ⇒ ¬ (∃ p. is-path v0 p v) |
  Some p ⇒ is-path v0 p v
            ∧ (∀ p′. is-path v0 p′ v → path-weight p ≤ 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.

\[
\text{type-synonym} \ (\'V,\'W) \ state = (\'V \ set) \times (\'V \rightarrow (\'V,\'W) \ path)
\]

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.

\[
\text{locale Dijkstra} = \begin{aligned}
\text{for} \ G :: (\'V,\'W::weight) \ graph+ \\
\text{fixes} \ v0 :: \'V \\
\text{assumes} \ \text{finite[simp,intro]}: \ \text{finite} \ V \ \text{finite} \ E \\
\text{assumes} \ v0-in-V[simp, intro]: \ v0 \in V \\
\text{assumes} \ \text{nonneg-weights[simp, intro]}: \ (v,w,v') \in \text{edges} \ G \Longrightarrow 0 \leq w
\end{aligned}
\]

Paths have non-negative weights.

\[
\text{lemma path-nonneg-weight: is-path v p v'} \implies 0 \leq \text{path-weight} p
\]

\[
\langle \text{proof} \rangle
\]

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.

\[
\text{definition dinvar} \ \sigma \equiv \text{let} \ (wl,res)=\sigma \ in \\
\forall v \in V \ \land \ \forall v \in V-\text{wl} \ \land \ \forall v \in V-\text{wl} \ \land \ \forall v \in V-\text{wl} \\
\text{is-path} v0 p v \land \text{int-vertices} p \subseteq V-\text{wl} \land \\
\text{is-path} v0 p v \land \text{path-weight} (\text{res} v) \leq \text{path-weight} (\text{Some} p) \\
\text{path-weight} (\text{res} v) \leq \text{path-weight} (\text{Some} p)
\]

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

\[
\text{lemma invar-imp-correct: dinvar} (\{\},res) \implies \text{is-shortest-path-map} v0 res
\]
The initial workset contains all vertices. The initial result maps \( v_0 \) to the empty path, and all other vertices to \( \text{None} \).

**Definition**

\[
d_{\text{init}} :: (V, W) \text{ state } nres \text{ where}
\]

\[
d_{\text{init}} \equiv \text{SPEC } (\lambda (wl, res). \ \\
wl = V \land res v_0 = \text{Some} [] \land (\forall v \in V - \{v_0\}. res v = \text{None}))
\]

The initial state satisfies the invariant.

**Lemma**

\[
d_{\text{init-invar}}: d_{\text{init}} \leq \text{SPEC } d_{\text{invar}}
\]

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

**Definition**

\[
pop-min :: (V, W) \text{ state } \Rightarrow (V \times (V, W) \text{ state}) \text{ state } \Rightarrow \text{bool}
\]

\[
pop-min \sigma \equiv \text{do } \ \\
\text{let } (wl, res) = \sigma; \\
\text{ASSERT } (wl \neq \{\}); \\
v \leftarrow \text{RES (least-map } (\text{path-weight'} \circ res) \text{) } wl); \\
\text{RETURN } (v, (wl - \{v\}, res))
\]

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.

**Definition**

\[
update \equiv \text{do } \ \\
\text{ASSERT } (\text{update-pre } v \sigma); \\
\text{SPEC } (\text{update-spec } v \sigma)
\]

Finally, we define Dijkstra's algorithm:
The following theorem states (total) correctness of Dijkstra’s algorithm.

**Theorem** dijkstra-correct: \( \text{dijkstra} \leq \text{SPEC} (\text{is-shortest-path-map } v0) \)

**Proof**
We integrate the new update function into the main algorithm:

**definition dijkstra′ where**

\[ \text{dijkstra′ } \equiv \text{do } \]

\[
\begin{align*}
&\sigma_0 \leftarrow \text{dinit; } \\
&(-, \text{res}) \leftarrow \text{WHILE}_{\text{dinv}} (\lambda(wl,-). \ wtl\neq\emptyset) \\
&\quad (\lambda\sigma. \ \text{do } \{(v,\sigma') \leftarrow \text{pop-min } \sigma; \text{update' } v \ \sigma'\}) \\
&\quad \sigma_0; \\
&\text{RETURN res}
\end{align*}
\]

**lemma dijkstra′-refines: dijkstra′ ≤⇓ Id dijkstra**

(\text{proof})

\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 efficient.

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**

\[ \text{mpath'} \text{ None } = \text{None } | \]

\[ \text{mpath'} \text{ Some } (p,w) = \text{Some } p \]

**fun mpath-weight′ :: ((′V,′W) path × ′W) option ⇒ (′W::weight) infty where**

\[ \text{mpath-weight'} \text{ None } = \text{top } | \]

\[ \text{mpath-weight'} \text{ Some } (p,w) = \text{Num } w \]

**context Dijkstra**

**begin**

**definition αw::(′V,′W) mwl ⇒ ′V set where** \( \alpha w \equiv \text{dom} \)**

**definition αr::(′V,′W) mres ⇒ ′V → ((′V,′W) path × ′W) where**

\( \alpha r \equiv \lambda v. \ \text{case res } v \ \text{of } \text{None } ⇒ \text{None } | \text{Some } (p,w) ⇒ \text{Some } (\text{rev } p) \)**

**definition αs:: (′V,′W) mstate ⇒ (′V,′W) state where**

\( \alpha s \equiv \text{map-prod } \alpha w \ \alpha r \)
Additional invariants for the new state. They guarantee that the cached weights are consistent.

**definition** res-invarm :: (\'V \to ((\'V,\'W) \times \'W)) \to \text{bool} \ where 

res-invarm \( \sigma \) \equiv (\forall v. \text{case } \text{res } v \text{ of} 

None \Rightarrow True \ |

Some (p,w) \Rightarrow w = \text{path-weight} (\text{rev } p))

**definition** dinvarm :: (\'V,\'W) mstate \to \text{bool} \ where 

\( \text{dinvarm } \sigma \equiv \text{let } (\text{wl, res}) = \sigma \text{ in} \)

(\forall v \in \text{dom } \text{wl}. \text{ the } (\text{wl } v) = \text{mpath-weight}' (\text{res } v)) \land \text{res-invarm res}

**lemma** mpath-weight'-correct: \[ \text{[dinvarm (\text{wl, res})]} \implies \text{mpath-weight}' (\text{res } v) = \text{path-weight}' (\text{\text{ar } res } v) \]

(proof)

**lemma** mpath'-correct: \[ \text{[dinvarm (\text{wl, res})]} \implies \text{mpath}' (\text{res } v) = \text{map-option rev} (\text{\text{ar } res } v) \]

(proof)

**lemma** wl-weight-correct:

assumes INV: \text{dinvarm (\text{wl, res})}

assumes WLV: \text{wl } v = \text{Some } w

shows \text{path-weight}' (\text{\text{ar } res } v) = w

(proof)

The initial state is constructed using an iterator:

**definition** mdinit :: (\'V,\'W) mstate nres \ where 

\( \text{mdinit} \equiv \text{do } \)

\( \text{wl} \leftarrow \text{FOREACH } \text{V} (\lambda v. \text{RETURN } (\text{wl } v \mapsto \text{Infty})) \text{ Map.empty; } \)

\( \text{RETURN } (\text{wl } v_0 \mapsto \text{Num } 0, [v_0 \mapsto ([], 0)]) ) \}

**lemma** mdinit-refines: \text{mdinit } \leq \Downarrow (\text{build-rel } \text{as dinvarm}) \text{ dinit}

(proof)

The new pop function:

**definition** mpop-min :: (\'V,\'W) mstate \to (\'V \times \text{\'W } \text{infty} \times (\'V,\'W) \text{ mstate}) \text{ nres} \ where 

\( \text{mpop-min } \sigma \equiv \text{do } \)

\( \text{let } (\text{wl, res}) = \sigma; \)

\( (v, w, w') \leftarrow \text{prio-pop-min } \text{wl}; \)

\( \text{RETURN } (v, w, (w', \text{res})) ) \}

**lemma** mpop-min-refines: \[ \text{[ (\sigma, \sigma') \in build-rel as dinvarm ] } \implies \text{mpop-min } \sigma \leq \]

17
The two algorithms are structurally different, so we use the nofail/inres method to prove refinement.

Finally, we assemble the refined algorithm:

\[ \downarrow (\text{build-rel}) \]
\[ (\lambda(v,w,\sigma). \ (v, as \ \sigma)) \]
\[ (\lambda(v,w,\sigma). \ \text{dinvarm } \sigma \land w = \text{mpath-weight'} (\text{snd } \sigma \ v)) \]

\[ (\text{pop-min } \sigma') \]

18
end

end

6 Graph Interface

theory GraphSpec
imports Main Graph
    Collections.Collections

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 ⇒ bool
   assumes finite[simp, intro!]:
      invar g ⇒ finite (nodes (α g))
      invar g ⇒ finite (edges (α g))
   assumes valid: invar g ⇒ valid-graph (α g)

locale graph-empty = graph +
   constrains α :: 'G ⇒ ('V,'W) graph
   fixes empty :: unit ⇒ 'G
   assumes empty-correct:
      invar g ⇒ invar (empty ())
      invar g ⇒ α (empty ()) = Graph.empty

locale graph-add-node = graph +
   constrains α :: 'G ⇒ ('V,'W) graph
   fixes add-node :: 'V ⇒ 'G ⇒ 'G
   assumes add-node-correct:
      invar g ⇒ invar (add-node v g)
      invar g ⇒ α (add-node v g) = Graph.add-node v (α g)

locale graph-delete-node = graph +
   constrains α :: 'G ⇒ ('V,'W) graph
   fixes delete-node :: 'V ⇒ 'G ⇒ 'G
   assumes delete-node-correct:
      invar g ⇒ invar (delete-node v g)
      invar g ⇒ α (delete-node v g) = Graph.delete-node v (α g)

locale graph-add-edge = graph +
   constrains α :: 'G ⇒ ('V,'W) graph
   fixes add-edge :: 'V ⇒ 'W ⇒ 'V ⇒ 'G ⇒ 'G
   assumes add-edge-correct:
      invar g ⇒ invar (add-edge v w v g)
      invar g ⇒ α (add-edge v w v g) = Graph.add-edge v w (α g)
locale graph-add-edge = graph +
  constrains \( \alpha : \, 'G \Rightarrow ('V, 'W) \) graph
  fixes add-edge :: 'V \Rightarrow 'W \Rightarrow 'V \Rightarrow 'G \Rightarrow 'G
  assumes add-edge-correct:
  \[
  \begin{align*}
  \text{invar } g \implies \text{invar } (\text{add-edge } v e v' g) \\
  \text{invar } g \implies \alpha (\text{add-edge } v e v' g) = \text{Graph}.\text{add-edge } v e v' (\alpha g)
  \end{align*}
  \]

locale graph-delete-edge = graph +
  constrains \( \alpha : \, 'G \Rightarrow ('V, 'W) \) graph
  fixes delete-edge :: 'V \Rightarrow 'W \Rightarrow 'V \Rightarrow 'G \Rightarrow 'G
  assumes delete-edge-correct:
  \[
  \begin{align*}
  \text{invar } g \implies \text{invar } (\text{delete-edge } v e v' g) \\
  \text{invar } g \implies \alpha (\text{delete-edge } v e v' g) = \text{Graph}.\text{delete-edge } v e v' (\alpha g)
  \end{align*}
  \]

locale graph-nodes-it-defs =
  fixes nodes-list-it :: 'G \Rightarrow ('V, 'V list) set-iterator
  begin
  definition nodes-it g \equiv it-to-it (nodes-list-it g)
  end

locale graph-nodes-it = graph \alpha invar + graph-nodes-it-defs nodes-list-it
  for \( \alpha : \, 'G \Rightarrow ('V, 'W) \) graph and invar and
  nodes-list-it :: 'G \Rightarrow ('V, 'V list) set-iterator
  +
  assumes nodes-list-it-correct:
  \[
  \begin{align*}
  \text{invar } g \implies \text{set-iterator } (\text{nodes-list-it } g) (\text{Graph}.\text{nodes } \alpha g)
  \end{align*}
  \]

locale graph-edges-it-defs =
  fixes edges-list-it :: ('V, 'W, ('V \times 'W \times 'V) list, 'G) graph-edges-it
  begin
  definition edges-it g \equiv it-to-it (edges-list-it g)
  end

locale graph-edges-it = graph \alpha invar + graph-edges-it-defs edges-list-it
  for \( \alpha : \, 'G \Rightarrow ('V, 'W) \) graph and invar and
  edges-list-it :: ('G, ('V, 'W, ('V \times 'W \times 'V) list) set-iterator
  +
  assumes edges-list-it-correct:
  \[
  \begin{align*}
  \text{invar } g \implies \text{set-iterator } (\text{edges-list-it } g) (\text{Graph}.\text{edges } \alpha g)
  \end{align*}
  \]
begin
  definition edges-it g ≡ it-to-it (edges-list-it g)
end

locale graph-edges-it = graph α invar + graph-edges-it-defs edges-list-it
  for α :: 'G ⇒ ('V,'W) graph and invar and
  edges-list-it :: ('V,'W,('V×'W×'V) list,'G) graph-edges-it
  +
  assumes edges-list-it-correct:
    invar g ⇒ set-iterator (edges-list-it g) (Graph.edges (α g))
begin
  lemma edges-it-correct:
    invar g ⇒ set-iterator (edges-it g) (Graph.edges (α g))
  ⟨proof⟩

  lemma pi-edges-it[icf-proper-iteratorI]:
    proper-it (edges-it S) (edges-it S)
  ⟨proof⟩

  lemma edges-it-proper[proper-it]:
    proper-it' edges-it edges-it
  ⟨proof⟩
end

type-synonym ('V,'W,α,'G) graph-succ-it =
  'G ⇒ 'V ⇒ ('W×'V,('W×'V) list) set-iterator

locale graph-succ-it-defs =
  fixes succ-list-it :: 'G ⇒ 'V ⇒ ('W×'V,('W×'V) list) set-iterator
begin
  definition succ-it g v ≡ it-to-it (succ-list-it g v)
end

locale graph-succ-it = graph α invar + graph-succ-it-defs succ-list-it
  for α :: 'G ⇒ ('V,'W) graph and invar and
  succ-list-it :: 'G ⇒ 'V ⇒ ('W×'V,('W×'V) list) set-iterator +
  assumes succ-list-it-correct:
    invar g ⇒ set-iterator (succ-list-it g v) (Graph.succ (α g) v)
begin
  lemma succ-it-correct:
    invar g ⇒ set-iterator (succ-it g v) (Graph.succ (α g) v)
  ⟨proof⟩

  lemma pi-succ-it[icf-proper-iteratorI]:
    proper-it (succ-it S v) (succ-it S v)
  ⟨proof⟩

  lemma succ-it-proper[proper-it]:
  ⟨proof⟩
proper-it' (λS. succ-it S v) (λS. succ-it S v)
(proof)
end

6.1 Adjacency Lists

**type-synonym** ('V,'W) adj-list = 'V list × ('V×'W×'V) list

definition adjl-α :: ('V,'W) adj-list ⇒ ('V,'W) graph where
  adjl-α l ≡ let (nl,el) = l in
  nodes = set nl ∪ fst'set el ∪ snd'snd'set el,
  edges = set el

**lemma** adjl-is-graph: graph adjl-α (λ-. True)
(proof)

**type-synonym** ('V,'W,'G) graph-from-list = ('V,'W) adj-list ⇒ 'G
locale graph-from-list = graph +
  constrains α :: 'G ⇒ ('V,'W) graph
  fixes from-list :: ('V,'W) adj-list ⇒ '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 ⇒ ('V,'W) adj-list
locale graph-to-list = graph +
  constrains α :: 'G ⇒ ('V,'W) graph
  fixes to-list :: 'G ⇒ ('V,'W) adj-list
  assumes to-list-correct:
  invar g ⇒ 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
  gop-nodes-list-it :: 'G ⇒ ('V,'V list) set-iterator
  gop-edges-list-it :: ('V,'W,('V×'W×'V) list,'G) graph-edges-it
  gop-succ-list-it :: 'G ⇒ 'V ⇒ ('W×'V,('W×'V) list) set-iterator
locale StdGraphDefs =
  graph-nodes-it-defs gop-nodes-list-it ops
+ graph-edges-it-defs gop-edges-list-it ops
+ graph-succ-it-defs gop-succ-list-it ops
for 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
  abbreviation nodes-list-it where nodes-list-it ≡ gop-nodes-list-it ops
  abbreviation edges-list-it where edges-list-it ≡ gop-edges-list-it ops
  abbreviation succ-list-it where succ-list-it ≡ gop-succ-list-it 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 +
  graph-nodes-it α invar nodes-list-it +
  graph-edges-it α invar edges-list-it +
  graph-succ-it α invar succ-list-it
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 ⇒ set-iterator (nodes-it g) (nodes α g)
  ⟨proof⟩

lemma (in graph-edges-it) edges-it-is-iterator[refine-transfer]:
  invar g ⇒ set-iterator (edges-it g) (edges α g)
  ⟨proof⟩
lemma (in graph-succ-it) succ-it-is-iterator[refine-transfer]:
  invar g ⇒ set-iterator (succ-it g v) (Graph.succ α g v)
⟨proof⟩

lemma (in graph) drh[refine-dref-RELATES]: RELATES (build-rel α invar)
⟨proof⟩

end

7 Generic Algorithms for Graphs

theory GraphGA
imports
  GraphSpec
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 (λg v. a v g) (e ()) nl
  in foldl (λg (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 a
  shows graph-from-list α invar (gga-from-list e a u)
⟨proof⟩

term map-iterator-product

locale gga-edges-it-defs =
  graph-nodes-it-defs nodes-list-it +
  graph-succ-it-defs succ-list-it
for nodes-list-it :: ('V,'W,'V list,'G) graph-nodes-it
and succ-list-it :: ('V,'W,('W×'V) list,'G) graph-succ-it
begin
  definition gga-edges-list-it ::
    ('V,'W,('V ×'W ×'V) list,'G) graph-edges-it
  where gga-edges-list-it G ≡ set-iterator-product
    (nodes-it G) (succ-it G)
locale gga-edges-it = gga-edges-it-defs nodes-list-it succ-list-it
  + graph α invar
  + graph-nodes-it α invar nodes-list-it
  + graph-succ-it α invar succ-list-it
for α :: 'G ⇒ ('V,'W) graph
  and invar
  and nodes-list-it :: ('V,'W,'V list,'G) graph-nodes-it
  and succ-list-it :: ('V,'W,'(W×'V) list,'G) graph-succ-it
begin
  lemma gga-edges-list-it-impl:
  shows graph-edges-it α invar gga-edges-list-it
⟨proof⟩
end
locale gga-to-list-defs-loc =
  graph-nodes-it-defs nodes-list-it
  + graph-edges-it-defs edges-list-it
for nodes-list-it :: ('V,'W,'V list,'G) graph-nodes-it
  and edges-list-it :: ('V,'W,'(V×'W×'V) list,'G) graph-edges-it
begin
  definition gga-to-list ::
    ('V,'W,'G) graph-to-list
  where
    gga-to-list g ≡
    (nodes-it g (λ-. True) (#) []), edges-it g (λ-. True) (#) [])
end
locale gga-to-list-loc = gga-to-list-defs-loc nodes-list-it edges-list-it +
  graph α invar
  + graph-nodes-it α invar nodes-list-it
  + graph-edges-it α invar edges-list-it
for α :: 'G ⇒ ('V,'W) graph and invar
  and nodes-list-it :: ('V,'W,'V list,'G) graph-nodes-it
  and edges-list-it :: ('V,'W,'(V×'W×'V) list,'G) graph-edges-it
begin
  lemma gga-to-list-correct:
  shows graph-to-list α invar gga-to-list
⟨proof⟩
end
8 Implementing Graphs by Maps

theory GraphByMap
imports
  GraphSpec
  GraphGA
begin

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) (λx. F2 (the (M1 x)))
(proof)

lemma ranE: assumes v∈ran m obtains k where m k = Some v
(proof)

lemma option-bind-alt:
  Option.bind x f = (case x of None ⇒ None | Some v ⇒ f v)
(proof)

locale GraphByMapDefs =
  m1: StdMapDefs m1-ops +
  m2: StdMapDefs m2-ops +
  s3: StdSetDefs s3-ops
  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)

26
definition gbm-empty :: ('V', 'W', 'm1) graph-empty where
  gbm-empty ≡ m1.empty

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

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

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 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.delete e s3) m2) g)
  ) in
  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 |

  )

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 |

  )

definition gbm-nodes-list-it :: ('V', 'W', 'V list list', 'm1) graph-nodes-it where
  gbm-nodes-list-it g ≡ map-iterator-dom (m1.iteratei g)
  (ML)

definition gbm-edges-list-it :: ('V', 'W', ('V'×'W'×'V') list, 'm1) graph-edges-it where
  gbm-edges-list-it g ≡ set-iterator-image
  ((λ((v1,m1),(v2,m2),w). (v1,w,v2)))
  (set-iterator-product (m1.iteratei g))
  (λ(v,m2). set-iterator-product)
\((m2.\text{iterate} m2) \ (\lambda(w,s3). \ s3.\text{iterate} s3))\)

\(\langle \text{ML} \rangle\)

\textbf{definition} \(\text{gbm\dash succ\dash list\dash it} :: (V, W, (W \times V) \text{ list}, m1) \text{ graph\dash succ\dash it}\)

\textbf{where}

\(\text{gbm\dash succ\dash list\dash it} g v \equiv \text{case } m1.\text{lookup } v \ g \text{ of}\)

\(\text{None } \Rightarrow \text{set\dash iterator\dash emp } v \)

\(\text{Some } m2 \Rightarrow \text{set\dash iterator\dash image}\)

\(\lambda((v',m2), w). \ (w,v')\)

\((\text{set\dash iterator\dash product} (m2.\text{iterate} m2) \ (\lambda(v', s3). \ s3.\text{iterate} s3)))\)

\(\langle \text{ML} \rangle\)

\textbf{definition}\n
\(\text{gbm\dash from\dash list } \equiv \text{gga\dash from\dash list gbm\dash empty gbm\dash add\dash node gbm\dash add\dash edge}\)

\textbf{lemma} \(\text{gbm\dash nodes\dash list\dash it\dash unf} : \)

\(\text{it\dash to\dash it } (\text{gbm\dash nodes\dash list\dash it } g) \equiv \text{map\dash iterator\dash dom } \text{it\dash to\dash it } (m1.\text{list\dash it } g)\)

\(\langle \text{proof} \rangle\)

\textbf{lemma} \(\text{gbm\dash edges\dash list\dash it\dash unf} : \)

\(\text{it\dash to\dash it } (\text{gbm\dash edges\dash list\dash it } g) \equiv \text{set\dash iterator\dash image}\)

\(\lambda((v1, m1),(v2, m2), w). \ (v1, w, v2))\)

\((\text{set\dash iterator\dash product } \text{it\dash to\dash it } (m1.\text{list\dash it } g))\)

\((\lambda(v, m2). \text{set\dash iterator\dash product})\)

\((\text{it\dash to\dash it } (m2.\text{list\dash it } m2)) \ (\lambda(w, s3). \text{it\dash to\dash it } (s3.\text{list\dash it } s3))))\)

\(\langle \text{proof} \rangle\)

\textbf{lemma} \(\text{gbm\dash succ\dash list\dash it\dash unf} : \)

\(\text{it\dash to\dash it } (\text{gbm\dash succ\dash list\dash it } g v) \equiv \text{case } m1.\text{lookup } v \ g \text{ of}\)

\(\text{None } \Rightarrow \text{set\dash iterator\dash emp } v \)

\(\text{Some } m2 \Rightarrow \text{set\dash iterator\dash image}\)

\(\lambda((v', m2), w). \ (w, v')\)

\((\text{set\dash iterator\dash product } \text{it\dash to\dash it } (m2.\text{list\dash it } m2))\)

\((\lambda(v', s3). \text{it\dash to\dash it } (s3.\text{list\dash it } s3))))\)

\(\langle \text{proof} \rangle\)

\textbf{end}\n
\textbf{sublocale} GraphByMapDefs < graph\dash nodes\dash it\dash defs gbm\dash nodes\dash list\dash it \langle \text{proof} \rangle


sublocale GraphByMapDefs < graph-edges-it-defs gbm-edges-list-it ⟨proof⟩

sublocale GraphByMapDefs < graph-succ-it-defs gbm-succ-list-it ⟨proof⟩

sublocale GraphByMapDefs < gga-to-list-defs-loc gbm-nodes-list-it gbm-edges-list-it ⟨proof⟩

context GraphByMapDefs begin

definition [icf-rec-def]: gbm-ops ≡ |
  gop-α = gbm-α,
  gop-invar = gbm-invar,
  gop-empty = gbm-empty,
  gop-add-node = gbm-add-node,
  gop-delete-node = gbm-delete-node,
  gop-add-edge = gbm-add-edge,
  gop-delete-edge = gbm-delete-edge,
  gop-from-list = gbm-from-list,
  gop-to-list = gga-to-list,
  gop-nodes-list-it = gbm-nodes-list-it,
  gop-edges-list-it = gbm-edges-list-it,
  gop-succ-list-it = gbm-succ-list-it |
⟩

⟨ML⟩
end

locale GraphByMap = GraphByMapDefs m1-ops m2-ops s3-ops m1-mvif +
  m1: StdMap m1-ops +
  m2: StdMap m2-ops +
  s3: StdSet s3-ops +
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

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)
⟨proof⟩

end

sublocale GraphByMap < graph gbm-α gbm-invar
⟨proof⟩

29
context GraphByMap
begin

lemma gbm-empty-impl:
  graph-empty gbm-α gbm-invar gbm-empty
  ⟨proof⟩

lemma gbm-add-node-impl:
  graph-add-node gbm-α gbm-invar gbm-add-node
  ⟨proof⟩

lemma gbm-delete-node-impl:
  graph-delete-node gbm-α gbm-invar gbm-delete-node
  ⟨proof⟩

lemma gbm-add-edge-impl:
  graph-add-edge gbm-α gbm-invar gbm-add-edge
  ⟨proof⟩

lemma gbm-delete-edge-impl:
  graph-delete-edge gbm-α gbm-invar gbm-delete-edge
  ⟨proof⟩

lemma gbm-nodes-list-it-impl:
  shows graph-nodes-it gbm-α gbm-invar gbm-nodes-list-it
  ⟨proof⟩

lemma gbm-edges-list-it-impl:
  shows graph-edges-it gbm-α gbm-invar gbm-edges-list-it
  ⟨proof⟩

lemma gbm-succ-list-it-impl:
  shows graph-succ-it gbm-α gbm-invar gbm-succ-list-it
  ⟨proof⟩

lemma gbm-from-list-impl:
  shows graph-from-list gbm-α gbm-invar gbm-from-list
  ⟨proof⟩

end

sublocale GraphByMap < graph-nodes-it gbm-α gbm-invar gbm-nodes-list-it
⟨proof⟩

sublocale GraphByMap < graph-edges-it gbm-α gbm-invar gbm-edges-list-it
⟨proof⟩

sublocale GraphByMap < graph-succ-it gbm-α gbm-invar gbm-succ-list-it
⟨proof⟩
sublocale GraphByMap
< gga-to-list-loc gbm-α gbm-invar gbm-nodes-list-it gbm-edges-list-it
⟨proof⟩

category GraphByMap
begin
lemma gbm-to-list-impl: graph-to-list gbm-α gbm-invar gga-to-list
⟨proof⟩

lemma gbm-ops-impl: StdGraph gbm-ops
⟨proof⟩

end
⟨ML⟩
end

9 Graphs by Hashmaps

theory HashGraphImpl
imports
  GraphByMap
begin

Abbreviation: hlg

type-synonym (′V,′E) hlg =
  (′V,(′V,′E ls) HashMap.hashmap) HashMap.hashmap
⟨ML⟩
interpretation hh-mvif: g-value-image-filter-loc hm-ops hm-ops
⟨proof⟩
interpretation hlg-gbm: GraphByMap hm-ops hm-ops ls-ops
 hh-mvif.g-value-image-filter
 ⟨proof⟩
 ⟨ML⟩
definition [ief-rec-def]: hlg-ops ≡ hlg-gbm.gbm-ops
 ⟨ML⟩
interpretation hlg: StdGraph hlg-ops
 ⟨proof⟩
 ⟨ML⟩

thm map-iterator-dom-def set-iterator-image-def
 set-iterator-image-filter-def

definition test-codegen where test-codegen ≡ (
 hlg.empty,
hlg.add-node,
hlg.delete-node,
hlg.add-edge,
hlg.delete-edge,
hlg.from-list,
hlg.to-list,
hlg.nodes-it,
hlg.edges-it,
hlg.succ-it
)

export-code test-codegen in SML
end

10 Implementation of Dijkstra’s-Algorithm using the ICF

theory Dijkstra-Impl
imports
  Dijkstra
  GraphSpec
  HashGraphImpl
  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.

locale dijkstraC =
  g: StdGraph g-ops +
  mr: StdMap mr-ops +
  qw: StdUprio 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
begin
  definition αsc == map-prod qw.α mr.α
  definition dinvarC-add == λ(wl,res). qw.invar wl ∧ mr.invar res

32
definition cdinit :: 'G ⇒ 'V ⇒ ('qw×'mr) nres where
cdinit g v0 ≡ do 
  wl ← FOREACH (nodes (g α g))
    (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 :: 'G ⇒ 'V ⇒ 'W infty ⇒ ('qw×'mr) ⇒ ('qw×'mr) nres where
cupdate g v wv σ= do 
  ASSERT (dinvarC-add σ);
  let (wl,res)=σ;
  let pv=mpath′ (mr.lookup v res);
  FOREACH (succ (g α g) v) (λ(w,v') (wl,res).
    if (wv + Num w' < mpath-weight′ (mr.lookup w' 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 g v0 ≡ do 
  σ0 ← cdinit g v0;
  (,-res) ← WHILET (λ(wl,-). ¬ qw.isEmpty wl)
    (λσ. do { (v,wv,σ') ← cpop-min σ; cupdate g v wv σ' } ) 
  σ0;
  RETURN res
}
end
locale dijkstraC-fixg = dijkstraC g-ops mr-ops qw-ops +
Dijkstra ga v0
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 ga :: ('V,'W) graph
and v0 :: 'V +
fixes g :: 'G
assumes g-rel: (g,ga)∈ br g α g.invar
begin
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-fixg) idijkstra-correct:**

- **Shows**
  - `weighted-graph.is-shortest-path-map ga v0 (αr (mr.α (idijkstra g v0)))`
  - `is ?G1`
  - `and mr.invar (idijkstra g v0) (is ?G2)`
  - `and Dijkstra.res-invarm (mr.α (idijkstra g v0)) (is ?G3)`

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

- **ML**
  - `interpretation hrf: dijkstraC hlg-ops rm-ops aluprioi-ops`
  - **proof**

**ML**

- **Definition** `hrf-dijkstra ≡ hrf.idijkstra`
- **Lemmas** `hrf-dijkstra-correct = hrf.idijkstra-correct[folded hrf-dijkstra-def]`

- **Export-code** `hrf-dijkstra checking SML`

- **Export-code** `hrf-dijkstra in OCaml`
- **Export-code** `hrf-dijkstra in Haskell`
- **Export-code** `hrf-dijkstra checking Scala`

- **Definition** `hrfn-dijkstra :: (nat,nat) hlg ⇒ -`
  - **where** `hrfn-dijkstra ≡ hrf-dijkstra`

- **Export-code** `hrfn-dijkstra in SML`

- **Lemmas** `hrfn-dijkstra-correct = hrf-dijkstra-correct[where ?′a = nat and ?′b = nat, folded hrfn-dijkstra-def]`

- **Term** `hrfn-dijkstra`
- **Term** `hlg.from-list`

**Definition** `test-hrfn-dijkstra`
≡ rm.to-list
  (hrfn-dijkstra (hlg.from-list [(0..<4),(0,3,1),(0,4,2),(2,1,3),(1,4,3)]) 0)

⟨ML⟩
end

11 Implementation of Dijkstra’s-Algorithm using Automatic Determinization

theory Dijkstra-Impl-Adet
imports
  Dijkstra
  GraphSpec
  HashGraphImpl
  Collections.Refine-Dflt-ICF
  HOL−Library.Code-Target-Numeral
begin

11.1 Setup

11.1.1 Infinity

definition infty-rel-internal-def:
infty-rel R ≡ {{Num a, Num a'}| a a'. (a,a')∈R} ∪ {{Infty,Infty}}

lemma infty-rel- deriv[refine-rel-defs]:
(R)infty-rel = {{Num a, Num a'}| a a'. (a,a')∈R} ∪ {{Infty,Infty}}
⟨proof⟩

lemma infty-relI:
(Infty,Infty)∈(R)infty-rel
(a,a')∈R ⇒ (Num a, Num a')∈(R)infty-rel
⟨proof⟩

lemma infty-relE:
assumes (x,x')∈(R)infty-rel
obtains x=Infty and x'=Infty
| a a' where x=Num a and x'=Num a' and (a,a')∈R
⟨proof⟩

lemma infty-rel-simps[simp]:
(Infty,x')∈(R)infty-rel ℓ→ x'=Infty
(x,Infty)∈(R)infty-rel ℓ→ x=Infty
(Num a, Num a')∈(R)infty-rel ℓ→ (a,a')∈R
⟨proof⟩

lemma infty-rel-sv[relator-props]:
single-valued R ⇒ single-valued ((R)infty-rel)
⟨proof⟩

36
lemma infty-rel-id[simp, relator-props]: (Id) infty-rel = Id
  ⟨proof⟩

consts i-infty :: interface ⇒ interface
lemmas [autoref-rel-intf] = REL-INTFI[of infty-rel i-infty]

lemma autoref-infty[param, autoref-rules]:
  (Infty, Infty) ∈ (R) infty-rel
  (Num, Num) ∈ R → (R) infty-rel
  (case-infty, case-infty) ∈ Rr → (R → Rr) → (R) infty-rel → Rr
  (rec-infty, rec-infty) ∈ Rr → (R → Rr) → (R) infty-rel → Rr
  ⟨proof⟩

definition [simp]: is-Infty x ≡ case x of Infty ⇒ True | - ⇒ False

context begin interpretation autoref-syn ⟨proof⟩
lemma pat-is-Infty[autoref-op-pat]:
  x = Infty ≡ (OP is-Infty :: i ⟹ i-i-infty → i-bool)$x
  Infty = x ≡ (OP is-Infty :: i ⟹ i-i-infty → i-bool)$x
  ⟨proof⟩
end

lemma autoref-is-Infty[autoref-rules]:
  (is-Infty, is-Infty) ∈ (R) infty-rel → bool-rel
  ⟨proof⟩

definition infty-eq eq v1 v2 ≡
  case (v1, v2) of
    (Infty, Infty) ⇒ True
    | (Num a1, Num a2) ⇒ eq a1 a2
    | - ⇒ False

lemma infty-eq-autoref[autoref-rules (overloaded)]:
  [[ GEN-OP eq (=) (R → R → bool-rel) ]]
  ⇒ (infty-eq eq, (=)) ∈ (R) infty-rel → (R) infty-rel → bool-rel
  ⟨proof⟩

lemma infty-eq-expand[autoref-struct-expand]: (=) = infty-eq (=)
  ⟨proof⟩

context begin interpretation autoref-syn ⟨proof⟩
lemma infty-val-autoref[autoref-rules]:
  [SIDE-PRECOND (x ≠ Infty); (xi, x) ∈ (R) infty-rel]
  ⇒ (val xi, (OP val :: (R) infty-rel → R) $ x) ∈ R
  ⟨proof⟩
end

definition infty-plus where
\( \infty + pl \ a \ b \equiv \text{case } (a, b) \text{ of } (\text{Num } a, \text{Num } b) \Rightarrow \text{Num } (pl \ a \ b) \mid - \Rightarrow \text{Infty} \)

**Lemma infty-plus-param[\text{param}]:**
\[
(\text{infty-plus, infty-plus}) \in (\text{R} \rightarrow \text{R} \rightarrow \text{R}) \rightarrow \langle \text{R} \rangle \text{infty-rel} \rightarrow \langle \text{R} \rangle \text{infty-rel} \\
\langle \text{proof} \rangle
\]

**Lemma infty-plus-eq-plus:**
\[
\text{infty-plus}(+) = (+) \\
\langle \text{proof} \rangle
\]

**Lemma infty-plus-autoref[\text{autoref-rules}]:**
\[
\text{GEN-OP } pl \ (+) (\text{R} \rightarrow \text{R} \rightarrow \text{R}) \\
\Rightarrow (\text{infty-plus } pl,(+)) \in (\text{R}) \text{infty-rel} \rightarrow (\text{R}) \text{infty-rel} \rightarrow (\text{R}) \text{infty-rel} \\
\langle \text{proof} \rangle
\]

### 11.1.2 Graph

**Consts** \text{i-graph} :: \text{interface} \Rightarrow \text{interface} \Rightarrow \text{interface}

**Definition graph-more-rel-internal-def:**
\[
\text{graph-more-rel } Rm \ Rv \ Rw \equiv \{(g,g') \}. \\
\quad (\text{graph.nodes } g, \text{graph.nodes } g') \in (\text{Rv}) \text{set-rel} \\
\quad \land (\text{graph.edges } g, \text{graph.edges } g') \in (\langle \text{Rv}, (\text{Rw}, \text{Rv}) \text{prod-rel} \rangle \text{prod-rel}) \text{set-rel} \\
\quad \land (\text{graph.more } g, \text{graph.more } g') \in Rm \\
\langle \text{proof} \rangle
\]

**Lemma graph-more-rel-def[\text{refine-rel-defs}]:**
\[
\langle Rm, Rv, Rw \rangle \text{graph-more-rel } \equiv \{(g,g') \}. \\
\quad (\text{graph.nodes } g, \text{graph.nodes } g') \in (\text{Rv}) \text{set-rel} \\
\quad \land (\text{graph.edges } g, \text{graph.edges } g') \in (\langle \text{Rv}, (\text{Rw}, \text{Rv}) \text{prod-rel} \rangle \text{prod-rel}) \text{set-rel} \\
\quad \land (\text{graph.more } g, \text{graph.more } g') \in Rm \\
\langle \text{proof} \rangle
\]

**Abbreviation graph-rel \equiv \langle \text{unit-rel} \rangle \text{graph-more-rel}**

**Lemmas graph-rel-def = graph-more-rel-def[\text{where } Rm=\text{unit-rel, simplified}]**

**Lemma graph-rel-id[simp]:** \(\langle \text{Id}, \text{Id} \rangle \text{graph-rel} = \text{Id}\)
\langle \text{proof} \rangle

**Lemma graph-more-rel-sv[\text{relator-props}]:**
\[
\text{[single-valued } Rm; \text{single-valued } Rv; \text{single-valued } Rw] \\
\Rightarrow \text{single-valued } ((Rm, Rv, Rw) \text{graph-more-rel}) \\
\langle \text{proof} \rangle
\]

**Lemma [\text{autoref-itype}]:**
\[
\text{graph.nodes } :: \text{i} \langle \text{Iv}, \text{Iw} \rangle \text{i-graph} \rightarrow \text{i} \langle \text{Iv} \rangle \text{i-set} \\
\langle \text{proof} \rangle
\]

**Thm** is-map-to-sorted-list-def

38
definition nodes-to-list g ≡ it-to-sorted-list (λ- . True) (graph.nodes g)
lemma nodes-to-list-itype[autoref-itype]: nodes-to-list :: i⟨Iv, Iw⟩i-graph → i⟨⟨Iv⟩i-list⟩i-nres
lemma nodes-to-list-pat[autoref-op-pat]: it-to-sorted-list (λ- . True) (graph.nodes g) ≡ nodes-to-list g

lemma succ-to-list g v ≡ it-to-sorted-list (λ- . True) (Graph.succ g v)
lemma succ-to-list-itype[autoref-itype]: succ-to-list :: i⟨Iv, Iw⟩i-graph → Iv → i⟨⟨Iw⟩i-list⟩i-nres
lemma succ-to-list-pat[autoref-op-pat]: it-to-sorted-list (λ- . True) (Graph.succ g v) ≡ succ-to-list g v

class graph begin
definition rel-def-internal: rel Rv Rw ≡ br α invar O ⟨Rv, Rw⟩graph-rel
lemma rel-def: ⟨Rv, Rw⟩rel ≡ br α invar O ⟨Rv, Rw⟩graph-rel
lemma rel-id[simp]: ⟨Id, Id⟩rel = br α invar
lemma rel-so[relator-props]: [single-valued Rv; single-valued Rw] → single-valued ⟨⟨Rv, Rw⟩⟩rel
lemmas [autoref-rel-intf] = REL-INTFI[of rel i-graph]
end

lemma (in graph-nodes-it) autoref-nodes-it[autoref-rules]:
assumes ID: PREFER-id Rv
shows (As. RETURN (it-to-list nodes-it s).nodes-to-list) ∈ ⟨Rv, Rw⟩rel → ⟨⟨Rv⟩list-rel⟩nres-rel

lemma (in graph-succ-it) autoref-succ-it[autoref-rules]:
assumes ID: PREFER-id Rv PREFER-id Rw
shows (λs v. RETURN (it-to-list (λs. succ-it s v) s).succ-to-list)
 ∈ ⟨Rv, Rw⟩rel → Rw → ⟨⟨Rv⟩prod-rel⟩list-rel⟩nres-rel

11.2 Refinement
locale dijkstraC =
g: StdGraph g-ops +
mr: StdMap mr-ops +
qw: StdUprio 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
locale dijkstraC-fixg = dijkstraC g-ops mr-ops qw-ops +
  Dijkstra ga v0

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 infinity","qw","more-qw") uprio-ops-scheme

assumes ga-trans: (g,ga) ∈ br g.α

begin
abbreviation v-rel ≡ Id :: ("V" × "V") set
abbreviation w-rel ≡ Id :: ("W" × "W") set

definition i-node :: interface where i-node ≡ undefined
definition i-weight :: interface where i-weight ≡ undefined

lemmas [autoref-rel-intf] = REL-INTFI[of v-rel i-node]
lemmas [autoref-rel-intf] = REL-INTFI[of w-rel i-weight]

lemma weight-plus-autoref[autoref-rules]:
  (0,0) ∈ w-rel
  (",",+) ∈ w-rel → w-rel → w-rel
  (",",+) ∈ (w-rel)infty-rel → (w-rel)infty-rel → (w-rel)infty-rel
  (",",-) ∈ (w-rel)infty-rel → (w-rel)infty-rel → bool-rel
 ⟨proof⟩

lemma [autoref-rules]: (g,ga) ∈ (v-rel,w-rel)g.rel ⟨proof⟩

lemma [autoref-rules]: (v0,v0) ∈ v-rel ⟨proof⟩

term mpath-weight'
lemma [autoref-rules]:
  (mpath-weight',mpath-weight')
  ∈ (⟨v-rel × w-rel × v-rel⟩list-rel × w-rel)option-rel → (w-rel)infty-rel
  (mpath', mpath')
  ∈ (⟨v-rel × w-rel × v-rel⟩list-rel × w-rel)option-rel
  → (⟨v-rel × w-rel × v-rel⟩list-rel)option-rel
 ⟨proof⟩

term mdinit

lemmas [autoref-tyrel] =
ty-REL[where R=v-rel]
ty-REL[where R=w-rel]
ty-REL[where R=(w-rel)infty-rel]
ty-REL[where R=⟨v-rel,(w-rel)infty-rel⟩qw.rel]
ty-REL[where R=⟨v-rel,⟨v-rel × w-rel × v-rel⟩list-rel,w-rel⟩mr.rel]
ty-REL[where R=⟨v-rel × w-rel × v-rel⟩list-rel]
lemmas [autoref-op-pat] = uprio-pats

schematic-goal cdijkstra-refines-aux:
  shows (\<\< c, mdijkstra \>>\) \in ?R
(\langle proof \rangle)

end

class context dijkstraC

begin

concrete-definition cdijkstra for g ?v0.0
  uses dijkstraC-fixg,cdijkstra-refines-aux
  [of g-ops mr-ops qw-ops]

  term cdijkstra

end

context dijkstraC-fixg

begin

  term cdijkstra
  term mdijkstra

lemma cdijkstra-refines:
  RETURN (cdijkstra g v0) \leq \langle build-rel mr.\alpha mr.invar \rangle mdijkstra
(\langle proof \rangle)

theorem cdijkstra-correct:
  shows
  weighted-graph.is-shortest-path-map ga v0 (\langle or (mr.\alpha (cdijkstra g v0)))
  (is ?G1)
  and mr.invar (cdijkstra g v0) (is ?G2)
  and res-invarm (mr.\alpha (cdijkstra g v0)) (is ?G3)
(\langle proof \rangle)

end

theorem (in dijkstraC) cdijkstra-correct:
  assumes INV: g.invar g
  assumes V0: v0 \in nodes (g.\alpha g)
  assumes nonneg-weights: \forall v w v'. (v,w,v') \in edges (g.\alpha g) \Longrightarrow 0 \leq w
  shows
  weighted-graph.is-shortest-path-map (g.\alpha g) v0
  (Dijkstra.or (mr.\alpha (cdijkstra g v0))) (is ?G1)
and Dijkstra.res-invarm \((mr.\alpha (cdijkstra g v0))\) (is \(?G2)\)

(proof)

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

⟨ML⟩
interpretation hrf: dijkstraC hlg-ops rm-ops aluprioi-ops
(proof)
⟨ML⟩
definition hrf-dijkstra ≡ hrf.cdijkstra
lemmas hrf-dijkstra-correct = hrf.cdijkstra-correct[folded hrf-dijkstra-def]

export-code hrf-dijkstra checking SML
export-code hrf-dijkstra in OCaml
export-code hrf-dijkstra in Haskell
export-code hrf-dijkstra checking Scala
definition hrfn-dijkstra :: (nat,nat) hlg ⇒ -
where hrfn-dijkstra ≡ hrf-dijkstra
export-code hrfn-dijkstra checking SML

lemmas hrfn-dijkstra-correct =
hrf-dijkstra-correct[where \(?'a = nat and ?'b = nat, folded hrfn-dijkstra-def]
end

12 Performance Test

theory Test
  imports Dijkstra-Impl-Adet
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 leg-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
run-graph vertices seed ==
\([\{v\}::<vertices]\)fst
(while (λ (g,v,s). v < vertices)
λ (g,v,s).
let (g'',v'',s'') = (while (λ (g',v',s'). v' < vertices)
(\lambda (g',v',s'). ((v,s',v')\#g',v'+1,\text{lcg-next } s'))
\in (g',v+1,s')
([],[0,\text{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 \times nat)) \text{ rm}

type-synonym nat-list-res = (nat \times (nat,nat) path \times nat) \text{ list}

definition nat-dijkstra :: (nat,nat) hlg \Rightarrow \text{ nat} \Rightarrow \text{ nat-res}
\text{ where}
\text{ nat-dijkstra} \equiv \text{ hrfn-dijkstra}

definition hlg-from-list-nat :: (nat,nat) \text{ adj-list} \Rightarrow \text{ (nat,nat) hlg}
\text{ where}
\text{ hlg-from-list-nat} \equiv \text{ hlg,from-list}

definition
\text{ nat-res-to-list} :: \text{ nat-res} \Rightarrow \text{ nat-list-res}
\text{ where} \text{ nat-res-to-list} \equiv \text{ rm.to-list}

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

\langle \text{ ML} \rangle

\text{ end}

References


