thys

mitjakrebs

June 26, 2022

Contents

0.0.1 Adaptors 4

1 Map **11**
1.1 Medium level 12
1.1.1 Adjacency structure 12
1.1.2 Directed adjacency structure 19
1.1.3 Undirected adjacency structure 21
1.1.4 Directed graphs 24

2 Queue **24**

3 BFS **25**
3.1 Specification of the algorithm 25
3.2 Verification of the correctness of the algorithm 27
3.2.1 Assumptions on the input 27
3.2.2 Loop invariants 27
3.2.3 Termination 37
3.2.4 Correctness 38

4 Alternating BFS **40**
4.1 Specification of the algorithm 40
4.2 Verification of the correctness of the algorithm 41
4.2.1 Assumptions on the input 41
4.2.2 Loop invariants 42
4.2.3 Termination 46
4.2.4 Correctness 47
4.3 Implementation of the algorithm 48
4.4 Implementation of the algorithm 49
4.5 Low level 50

5 Shortest augmenting path algorithm **55**
5.1 Specification of the algorithm 55
5.2 Verification of the correctness of the algorithm 58
5.2.1 Assumptions on the input 58
5.2.2 Loop invariants 59
5.2.3 Termination 63
5.2.4 Correctness 64
5.3 Implementation of the algorithm 66
5.3.1 Undirected graphs 68

6 Graph **68**
6.1 High level 68

theory *Dgraph*
imports
AGF.DDFS
begin

type-synonym *'a vertex* = *'a*

An edge in a directed graph is a pair of vertices.

type-synonym *'a edge* = (*'a vertex* × *'a vertex*)

type-synonym *'a dgraph* = *'a edge set*

locale *dgraph* =
fixes *G* :: *'a dgraph*

Let us identify a couple of special types of graphs.

locale *finite-dgraph* = *dgraph G* **for** *G* +
assumes *finite-edges*: *finite G*

lemma (**in** *finite-dgraph*) *finite-vertices*:
shows *finite* (*dVs G*)

locale *simple-dgraph* = *dgraph G* **for** *G* +
assumes *no-loop*: (*u*, *v*) ∈ *G* ⟹ *u* ≠ *v*

locale *symmetric-dgraph* = *dgraph G* **for** *G* +
assumes *symmetric*: (*u*, *v*) ∈ *G* ⟷ (*v*, *u*) ∈ *G*

end
theory *Dpath*
imports
Dgraph
Ports.Berge-to-DDFS
Ports.Mitja-to-DDFS
Ports.Noschinski-to-DDFS
begin

A directed path (*dpath* and *dpath-bet*) is a sequence $v_0, \ldots, v_k$ of vertices such that (v_{i-1}, v_i) is an edge for every $i = 1, \ldots, k$.

type-synonym *'a dpath* = *'a list*

lemmas *dpath-induct* = *edges-of-dpath.induct*

lemma *dpath-rev-induct*:
  assumes *P* []
  assumes $\bigwedge v.$ *P* [*v*]
  assumes $\bigwedge v\ v'\ l.$ *P* (*l* @ [*v*]) $\Longrightarrow$ *P* (*l* @ [*v*, *v'*])
  shows *P p*

The length of a *dpath* is the number of its edges.

abbreviation *dpath-length* :: *'a dpath* $\Rightarrow$ *nat* **where**
  dpath-length p $\equiv$ *length* (*edges-of-dpath p*)

A simple directed path is a directed path in which all vertices are distinct. Any directed path can be transformed into a directed simple path via function *dpath-bet-to-distinct*.

lemma *distinct-dpath-length-le-dpath-length*:
  assumes *dpath-bet G p u v*
  shows *dpath-length* (*dpath-bet-to-distinct G p*) $\leq$ *dpath-length p*

A vertex v is reachable from a vertex u if and only if there is a directed path from u to v.

lemma *reachable-iff-dpath-bet*:
  shows *reachable G u v* $\longleftrightarrow$ ($\exists$ *p*. *dpath-bet G p u v*)

lemma *reachable-trans*:
  assumes *reachable G u v*
  assumes *reachable G v w*
  shows *reachable G u w*

end
theory *Graph-Ext*
  imports
    AGF.Berge
begin

type-synonym *'a vertex* = *'a*

An edge in an undirected graph is a set of vertices.

type-synonym *'a edge* = *'a vertex set*

type-synonym *'a graph = 'a edge set*

Since this definition allows for hyperedges, we define a graph, as opposed to a hypergraph, as follows.

locale *graph* =
  fixes *G* :: *'a graph*
  assumes *graph*: ∀ *e*∈*G*. ∃ *u v*. *e* = {*u*, *v*}

lemma (**in** *graph*) *graph-subset*:
  assumes *G'* ⊆ *G*
  shows *graph G'*

lemma *graphs-eqI*:
  assumes *graph G1*
  assumes *graph G2*
  assumes ⋀*u v*. {*u*, *v*} ∈ *G1* ⟷ {*u*, *v*} ∈ *G2*
  shows *G1* = *G2*

locale *finite-graph* = *graph G* **for** *G* +
  assumes *finite-edges*: *finite G*

lemma (**in** *finite-graph*) *finite-vertices*:
  shows *finite* (*Vs G*)

end

0.0.1 Adaptors

theory *Graph-Adaptor*
  imports
    ../Directed-Graph/Dgraph
    ../Undirected-Graph/Graph-Ext
begin

An undirected graph can be viewed as a symmetric directed graph. Session `AGF` shows how to transform a *graph* into a symmetric *dgraph*. We extend, or rather redo, (parts of) their theory. Our issue with their theory is that the lemmas are inside a locale that assumes that the graph does not have loops. Most–if not all–of the lemmas hold even if the graph contains loops, though.

definition (**in** *graph*) *dEs* :: *'a dgraph* **where**
  dEs ≡ {(*u*, *v*). {*u*, *v*} ∈ *G*}

lemma (**in** *graph*) *dEs-symmetric*:
  shows (*u*, *v*) ∈ *dEs* ⟷ (*v*, *u*) ∈ *dEs*

context *finite-graph*
begin

sublocale *F*: *finite-dgraph dEs*
end

end
theory *Path*
imports
Graph-Ext
../../Misc-Ext
begin

A path (*path* and *walk-betw*) is a sequence $v_0, \ldots, v_k$ of vertices such that $\{v_{i-1}, v_i\}$ is an edge for every $i = 1, \ldots, k$.

type-synonym *'a path* = *'a list*

lemma *pathI*:
assumes *set* (*edges-of-path p*) ⊆ *G*
assumes *set p* ⊆ *Vs G*
shows *path G p*

lemma *walk-betw-induct* [*consumes 1*]:
assumes *walk-betw G u p v*
assumes ⋀*v*. *P* [*v*]
assumes ⋀*u v vs*. *P* (*v* # *vs*) ⟹ *P* (*u* # *v* # *vs*)
shows *P p*

lemma *walk-betw-induct-2* [*consumes 1*]:
assumes *walk-betw G u p v*
assumes *P* [*v*]
assumes ⋀*u*. *P* [*u*, *v*]
assumes ⋀*u x xs*. *P* (*x* # *xs* @ [*v*]) ⟹ *P* (*u* # *x* # *xs* @ [*v*])
shows *P p*

We can concatenate paths.

lemma *walk-betw-appendI*:
assumes *walk-betw G u p v*
assumes *walk-betw G v p′ w*
shows *walk-betw G u* ((*butlast p* @ [*v*]) @ *tl p′*) *w*

lemma *edges-of-path-append*:
assumes *walk-betw G u p v*
assumes *walk-betw G v p′ w*
shows *edges-of-path* ((*butlast p* @ [*v*]) @ *tl p′*) = *edges-of-path p* @ *edges-of-path p′*

lemma *walk-betw-Cons-snocI*:
assumes *walk-betw G v p x*
assumes {*u*, *v*} ∈ *G*
assumes {*x*, *y*} ∈ *G*

shows
walk-betw G u (u # p @ [y]) y
{u, v} ∈ set (edges-of-path (u # p @ [y]))
{x, y} ∈ set (edges-of-path (u # p @ [y]))

And we can split paths.

fun *is-path-vertex-decomp* :: *'a graph ⇒ 'a path ⇒ 'a ⇒ 'a path × 'a path ⇒ bool* **where**
is-path-vertex-decomp G p v (q, r) ⟷ p = q @ tl r ∧ (∃ u w. walk-betw G u q v ∧ walk-betw G v r w)

definition *path-vertex-decomp* :: *'a graph ⇒ 'a path ⇒ 'a ⇒ 'a path × 'a path* **where**
path-vertex-decomp G p v ≡ SOME qr. is-path-vertex-decomp G p v qr

abbreviation *closed-path* :: *'a graph ⇒ 'a path ⇒ 'a ⇒ bool* **where**
closed-path G c v ≡ walk-betw G v c v ∧ Suc 0 < length c

fun *is-closed-path-decomp* :: *'a graph ⇒ 'a path ⇒ 'a path × 'a path × 'a path ⇒ bool* **where**
is-closed-path-decomp G p (q, r, s) ⟷
p = q @ tl r @ tl s ∧
(∃ u v w. walk-betw G u q v ∧ closed-path G r v ∧ walk-betw G v s w) ∧
distinct q

definition *closed-path-decomp* :: *'a graph ⇒ 'a path ⇒ 'a path × 'a path × 'a path* **where**
closed-path-decomp G p ≡ SOME qrs. is-closed-path-decomp G p qrs

A simple path is a path in which all vertices are distinct.

definition *distinct-path* :: *'a graph ⇒ 'a path ⇒ 'a ⇒ 'a ⇒ bool* **where**
distinct-path G p u v ≡ walk-betw G u p v ∧ distinct p

A vertex v is reachable from a vertex u if and only if there is a path from u to v.

definition *reachable* :: *'a graph ⇒ 'a ⇒ 'a ⇒ bool* **where**
reachable G u v ≡ ∃ p. walk-betw G u p v

The length of a *path* is the number of its edges.

abbreviation *path-length* :: *'a path ⇒ nat* **where**
path-length p ≡ length (edges-of-path p)

end
theory *Path-Adaptor*
imports
../Directed-Graph/Dpath

Graph-Adaptor
../Undirected-Graph/Path
begin

Since undirected and directed paths are defined in a very similar way, it is no surprise that the transition between them is very smooth.

lemmas *path-induct* = *dpath-induct*
lemmas *path-rev-induct* = *dpath-rev-induct*

lemma (**in** *graph*) *path-length-eq-dpath-length*:
shows *path-length p* = *dpath-length p*

lemma (**in** *graph*) *path-iff-dpath*:
shows *path G p* ⟷ *dpath dEs p*

lemma (**in** *graph*) *walk-betw-iff-dpath-bet*:
shows *walk-betw G u p v* ⟷ *dpath-bet dEs p u v*

lemma (**in** *graph*) *reachable-iff-reachable*:
shows *reachable G u v* ⟷ *Noschinski-to-DDFS.reachable dEs u v*

end
theory *Odd-Cycle*
imports
Path
begin

We redefine odd-length cycles–compared to the definition in session `AGF`–to also include loops for the following reason. We show that to find a shortest alternating path it suffices to consider a finite number of alternating paths. For this, we show that if there are no odd-length cycles, we can transform any alternating path into a simple alternating path by repeatedly removing cycles. If we do not consider loops as odd cycles, however, and hence do not exclude them, removing a single loop may destroy the alternation of the path.

definition *odd-cycle* **where**
odd-cycle p ≡ *odd* (*path-length p*) ∧ *hd p* = *last p*

end
theory *Alternating-Path*
imports
../Adaptors/Path-Adaptor
Odd-Cycle
begin

An alternating path w.r.t. a matching M is a path that alternates between edges in M and edges not in M. We generalize this definition to arbitrary

predicates P, Q: *alt-list ?a1.0 ?a2.0 ?a3.0* = ((∃ *P1 P2*. *?a1.0* = *P1* ∧ *?a2.0* = *P2* ∧ *?a3.0* = []) ∨ (∃ *P1 x P2 l*. *?a1.0* = *P1* ∧ *?a2.0* = *P2* ∧ *?a3.0* = *x* # *l* ∧ *P1 x* ∧ *alt-list P2 P1 l*)). The special case of an alternating path w.r.t. a matching M can then be obtained by instantiating the predicates as follows: *alt-path* ≡ λ*M p*. *alt-list* (λ*e*. *e* ∉ *M*) (λ*e*. *e* ∈ *M*) (*edges-of-path p*).

definition *alt-path* :: ('*a set* ⇒ *bool*) ⇒ ('*a set* ⇒ *bool*) ⇒ '*a graph* ⇒ '*a path* ⇒ '*a* ⇒ '*a* ⇒ *bool* **where**
alt-path P Q G p u v ≡ *alt-list P Q* (*edges-of-path p*) ∧ *walk-betw G u p v*

lemma *two-alt-pathsD*:
assumes *alt-path P Q G p u v*
assumes *alt-path P Q G q u v*
assumes ¬ (∃ *c*. *path G c* ∧ *odd-cycle c*)
shows *odd* (*path-length p*) = *odd* (*path-length q*)

As is the case for paths, we can reverse alternating paths.

lemma *alt-path-rev-oddI*:
assumes *alt-path P Q G p u v*
assumes *odd* (*path-length p*)
shows *alt-path P Q G* (*rev p*) *v u*

lemma *alt-path-rev-evenI*:
assumes *alt-path P Q G p u v*
assumes *even* (*path-length p*)
shows *alt-path Q P G* (*rev p*) *v u*

lemma *alt-path-revI*:
assumes *alt-path P Q G p u v*
shows *alt-path P Q G* (*rev p*) *v u* ∨ *alt-path Q P G* (*rev p*) *v u*

And we can split alternating paths.

lemma *alt-path-pref*:
assumes *alt-path P Q G* (*p* @ *v* # *q*) *u w*
shows *alt-path P Q G* (*p* @ [*v*]) *u v*

lemma *alt-path-pref-2*:
assumes *alt-path P Q G* (*p* @ *q*) *u w*
assumes *p* ≠ []
shows *alt-path P Q G p u* (*last p*)

lemma *alt-path-suf*:
assumes *alt-path P* (*Not* ∘ *P*) *G* (*p* @ [*v*, *v'*] @ *q*) *u w*
assumes *P* {*v*, *v'*}
shows *alt-path P* (*Not* ∘ *P*) *G* ([*v*, *v'*] @ *q*) *v w*

lemma *alt-path-suf-2*:
assumes *alt-path P* (*Not* ∘ *P*) *G* (*p* @ [*v*, *v'*] @ *q*) *u w*
assumes ¬ *P* {*v*, *v'*}
shows *alt-path* (*Not* ∘ *P*) *P G* ([*v*, *v'*] @ *q*) *v w*

lemma *alt-path-subst-pref*:
assumes *alt-path P Q G* (*p* @ *v* # *q*) *u w*
assumes *alt-path P Q G p' u v*
assumes ¬ (∃ *c*. *path G c* ∧ *odd-cycle c*)
shows *alt-path P Q G* (*p'* @ *q*) *u w*

definition *distinct-alt-path* :: ('*a set* ⇒ *bool*) ⇒ ('*a set* ⇒ *bool*) ⇒ '*a graph* ⇒ '*a path* ⇒ '*a* ⇒ '*a* ⇒ *bool* **where**
distinct-alt-path P Q G p u v ≡ *alt-path P Q G p u v* ∧ *distinct p*

A simple alternating path (*distinct-alt-path*) is an alternating path in which all vertices are distinct.

lemma (**in** *finite-graph*) *distinct-alt-paths-finite*:
shows *finite* {*p*. *distinct-alt-path P Q G p u v*}

If there are no odd-length cycles, we can transform any alternating path into a simple alternating path by repeatedly removing cycles. Removing an odd-length cycle, however, may destroy the alternation of the path.

lemma (**in** *graph*) *distinct-alt-path-alt-path-to-distinct*:
assumes *alt-path P Q G p u v*
assumes ¬ (∃ *c*. *path G c* ∧ *odd-cycle c*)
shows *distinct-alt-path P Q G* (*path-to-distinct p*) *u v*

Finally, we define reachability via alternating paths in the natural way.

definition *alt-reachable* :: ('*a set* ⇒ *bool*) ⇒ ('*a set* ⇒ *bool*) ⇒ '*a graph* ⇒ '*a* ⇒ '*a* ⇒ *bool* **where**
alt-reachable P Q G u v ≡ ∃ *p*. *alt-path P Q G p u v*

end
theory *Shortest-Path*
imports
Path
begin

definition *dist* :: '*a graph* ⇒ '*a* ⇒ '*a* ⇒ *enat* **where**
dist G u v ≡ *INF p*∈{*p*. *walk-betw G u p v*}. *enat* (*path-length p*)

abbreviation *is-shortest-path* :: '*a graph* ⇒ '*a path* ⇒ '*a* ⇒ '*a* ⇒ *bool* **where**
is-shortest-path G p u v ≡ *walk-betw G u p v* ∧ *path-length p* = *dist G u v*

end
theory *Shortest-Alternating-Path*
imports
Alternating-Path
Shortest-Path
begin

We generalize the notion of shortest paths to alternating paths in the natural way.

definition *alt-dist* :: ('*a set* ⇒ *bool*) ⇒ ('*a set* ⇒ *bool*) ⇒ '*a graph* ⇒ '*a* ⇒ '*a* ⇒ *enat* **where**
alt-dist P Q G u v ≡ *INF* p∈{p. *alt-path P Q G p u v*}. *enat* (*path-length p*)

definition *is-shortest-alt-path* :: ('*a set* ⇒ *bool*) ⇒ ('*a set* ⇒ *bool*) ⇒ '*a graph* ⇒ '*a path* ⇒ '*a* ⇒ '*a* ⇒ *bool* **where**
is-shortest-alt-path P Q G p u v ≡ *path-length p* = *alt-dist P Q G u v* ∧ *alt-path P Q G p u v*

lemma *alt-dist-le-alt-path-length*:
assumes *alt-path P Q G p u v*
shows *alt-dist P Q G u v* ≤ *path-length p*

lemma *alt-dist-alt-reachable-conv*:
shows *alt-dist P Q G u v* ≠ ∞ = *alt-reachable P Q G u v*

lemma (**in** *graph*) *alt-dist-eq-shortest-distinct-alt-path-length*:
assumes ¬ (∃ *c*. *path G c* ∧ *odd-cycle c*)
shows
alt-dist P Q G u v =
(*INF* p∈{p. *distinct-alt-path P Q G p u v*}. *enat* (*path-length p*))

lemma (**in** *finite-graph*) *is-shortest-alt-pathE*:
assumes *alt-reachable P Q G u v*
assumes ¬ (∃ *c*. *path G c* ∧ *odd-cycle c*)
obtains *p* **where** *is-shortest-alt-path P Q G p u v*

Again, we can reverse shortest alternating paths.

lemma (**in** *finite-graph*) *is-shortest-alt-path-revI*:
assumes *is-shortest-alt-path P Q G p u v*
assumes ¬ (∃ *c*. *path G c* ∧ *odd-cycle c*)
shows *is-shortest-alt-path P Q G* (*rev p*) *v u* ∨ *is-shortest-alt-path Q P G* (*rev p*) *v u*

And we can split shortest alternating paths.

lemma (**in** *finite-graph*) *is-shortest-alt-path-pref*:
assumes *is-shortest-alt-path P Q G* (*p* @ *v* # *q*) *u w*

assumes $\neg$ ($\exists$ *c*. *path G c* $\wedge$ *odd-cycle c*)
shows *is-shortest-alt-path P Q G* (*p* @ [*v*]) *u v*

lemma (**in** *finite-graph*) *is-shortest-alt-path-suf*:
assumes *is-shortest-alt-path P Q G* (*p* @ *v* # *q*) *u w*
assumes $\neg$ ($\exists$ *c*. *path G c* $\wedge$ *odd-cycle c*)
shows *is-shortest-alt-path P Q G* (*v* # *q*) *v w* $\vee$ *is-shortest-alt-path Q P G* (*v* # *q*) *v w*

lemma (**in** *finite-graph*) *is-shortest-alt-path-snoc-snocD*:
assumes *is-shortest-alt-path P Q G* (*p* @ [*v*, *w*]) *u w*
assumes $\neg$ ($\exists$ *c*. *path G c* $\wedge$ *odd-cycle c*)
shows *alt-dist P Q G u w* = *alt-dist P Q G u v* + *1*

end

1 Map

This section considers maps from three levels of abstraction.

theory *Map-Specs-Ext*
imports
../Misc-Ext
HOL−Data-Structures.Map-Specs
begin

On the high level, a map is a function (*map*). On the medium level, a map is specified via the interfaces *Map* and *Map-by-Ordered*. We extend theory *HOL−Data-Structures.Map-Specs*.

lemma *map-of-eq-Some-imp-mem*:
assumes *map-of l a* = *Some b*
shows (*a*, *b*) $\in$ *set l*

lemma *map-of-eq-Some-if-mem*:
assumes *sorted1 l*
assumes (*a*, *b*) $\in$ *set l*
shows *map-of l a* = *Some b*

lemma *map-of-eq-Some-iff-mem*:
assumes *sorted1 l*
shows *map-of l a* = *Some b* $\longleftrightarrow$ (*a*, *b*) $\in$ *set l*

lemma (**in** *Map-by-Ordered*) *mem-inorder-iff-lookup-eq-Some*:
assumes *invar m*
shows *lookup m a* = *Some b* $\longleftrightarrow$ (*a*, *b*) $\in$ *set* (*inorder m*)

lemma (**in** *Map-by-Ordered*) *set-inorder-delete-cong*:
assumes *invar m*

shows *set* (*inorder* (*delete a m*)) = *set* (*inorder m*) − (*case lookup m a of None* ⇒ {} | *Some b* ⇒ {(*a*, *b*)})

lemma (**in** *Map-by-Ordered*) *set-inorder-update-cong*:
assumes *invar m*
shows *set* (*inorder* (*update a b m*)) = *set* (*inorder m*) − (*case lookup m a of None* ⇒ {} | *Some y* ⇒ {(*a*, *y*)}) ∪ {(*a*, *b*)}

We define the domain and range of a map.

definition (**in** *Map*) *dom* :: *'m* ⇒ *'a set* **where**
dom m ≡ {*a*. *lookup m a* ≠ *None*}

lemma (**in** *Map-by-Ordered*) *dom-inorder-cong*:
assumes *invar m*
shows *dom m* = *fst* ‘ *set* (*inorder m*)

lemma (**in** *Map-by-Ordered*) *finite-dom*:
assumes *invar m*
shows *finite* (*dom m*)

definition (**in** *Map*) *ran* :: *'m* ⇒ *'b set* **where**
ran m ≡ {*b*. ∃ *a*. *lookup m a* = *Some b*}

lemma (**in** *Map-by-Ordered*) *finite-ran*:
assumes *invar m*
shows *finite* (*ran m*)

On the low level, the interfaces *Map* and *Map-by-Ordered* are implemented via red-black trees.

end

1.1 Medium level

1.1.1 Adjacency structure

theory *Adjacency*
imports
HOL−Data-Structures.Set-Specs
../../Map/Map-Specs-Ext
../../Orderings-Ext
begin

As mentioned above, a graph on the high level of abstraction is a set of edges. Hence, we would expect a graph to provide basic set operations such as insert, delete, union, intersection, and difference. Moreover, many graph algorithms, including breadth-first and depth-first search, involve iterating, or folding, over all vertices adjacent to a given vertex. Thus, we would have

liked to specify a graph on the medium level of abstraction via the following locales.

locale *Adjacency-Structure* =
fixes *empty* :: *'g*
fixes *insert* :: *'a*::*linorder* ⇒ *'a* ⇒ *'g* ⇒ *'g*
fixes *delete* :: *'a* ⇒ *'a* ⇒ *'g* ⇒ *'g*
fixes *adj* :: *'a* ⇒ *'g* ⇒ *'a list*
fixes *inv* :: *'g* ⇒ *bool*
assumes *adj-empty*: *adj v empty* = []
assumes *adj-insert*:
inv G ∧ *Sorted-Less.sorted* (*adj u G*) ⟹
adj u (*insert v w G*) = (*if u* = *v then ins-list w* (*adj u G*) *else adj u G*)
assumes *adj-delete*:
inv G ∧ *Sorted-Less.sorted* (*adj u G*) ⟹
adj u (*delete v w G*) = (*if u* = *v then List-Ins-Del.del-list w* (*adj u G*) *else adj u G*)
assumes *inv-empty*: *inv empty*
assumes *inv-insert*: *inv G* ∧ *Sorted-Less.sorted* (*adj u G*) ⟹ *inv* (*insert u v G*)
assumes *inv-delete*: *inv G* ∧ *Sorted-Less.sorted* (*adj u G*) ⟹ *inv* (*delete u v G*)

locale *Finite-Adjacency-Structure* = *Adjacency-Structure* **where** *insert* = *insert* **for**
insert :: *'a*::*linorder* ⇒ *'a* ⇒ *'g* ⇒ *'g* +
assumes *finite-domain-tbd*: *inv G* ⟹ *finite* {*v*. *adj v G* ≠ []}

locale *Adjacency-Structure-2* = *Adjacency-Structure* **where** *insert* = *insert* **for**
insert :: *'a*::*linorder* ⇒ *'a* ⇒ *'g* ⇒ *'g* +
fixes *union* :: *'g* ⇒ *'g* ⇒ *'g*
fixes *difference* :: *'g* ⇒ *'g* ⇒ *'g*
assumes *adj-union*:
⟦ *inv G1*; *Sorted-Less.sorted* (*adj v G1*); *inv G2*; *Sorted-Less.sorted* (*adj v G2*) ⟧ ⟹
adj v (*union G1 G2*) = *fold ins-list* (*adj v G2*) (*adj v G1*)
assumes *adj-difference*:
⟦ *inv G1*; *Sorted-Less.sorted* (*adj v G1*); *inv G2*; *Sorted-Less.sorted* (*adj v G2*) ⟧ ⟹
adj v (*difference G1 G2*) = *fold List-Ins-Del.del-list* (*adj v G2*) (*adj v G1*)
assumes *inv-union*: *inv G1* ⟹ *inv G2* ⟹ *inv* (*union G1 G2*)
assumes *inv-difference*: *inv G1* ⟹ *inv G2* ⟹ *inv* (*difference G1 G2*)

locale *Finite-Adjacency-Structure-2* = *Adjacency-Structure-2* **where** *insert* = *insert* **for**
insert :: *'a*::*linorder* ⇒ *'a* ⇒ *'g* ⇒ *'g* +
assumes *finite-domain-tbd*: *inv G* ⟹ *finite* {*v*. *adj v G* ≠ []}

Unfortunately, we were not able to refactor in time the entire formalization such that it uses locale *Finite-Adjacency-Structure-2* instead of the following one.

locale *adjacency* =

M: *Map-by-Ordered* **where**
empty = *Map-empty* **and**
update = *Map-update* **and**
delete = *Map-delete* **and**
lookup = *Map-lookup* **and**
inorder = *Map-inorder* **and**
inv = *Map-inv* +
S: *Set-by-Ordered* **where**
empty = *Set-empty* **and**
insert = *Set-insert* **and**
delete = *Set-delete* **and**
isin = *Set-isin* **and**
inorder = *Set-inorder* **and**
inv = *Set-inv* **for**
Map-empty **and**
Map-update :: *'a*::*linorder* ⇒ *'s* ⇒ *'m* ⇒ *'m* **and**
Map-delete **and**
Map-lookup **and**
Map-inorder **and**
Map-inv **and**
Set-empty **and**
Set-insert :: *'a* ⇒ *'s* ⇒ *'s* **and**
Set-delete **and**
Set-isin **and**
Set-inorder **and**
Set-inv

definition (**in** *adjacency*) *invar* :: *'m* ⇒ *bool* **where**
invar G ≡ *M.invar G* ∧ *Ball* (*M.ran G*) *S.invar*

definition (**in** *adjacency*) *adjacency-list* :: *'m* ⇒ *'a* ⇒ *'a list* **where**
adjacency-list G u ≡ *case Map-lookup G u of None* ⇒ [] | *Some s* ⇒ *Set-inorder s*

lemma (**in** *adjacency*) *finite-adjacency*:
shows *finite* (*set* (*adjacency-list G v*))

lemma (**in** *adjacency*) *distinct-adjacency-list*:
assumes *invar G*
shows *distinct* (*adjacency-list G v*)

This locale specifies a graph as a *Map-by-Ordered* mapping a vertex to its adjacency, which is specified as a *Set-by-Ordered*.

We define graph operations insert, delete, union, as well as difference, and show that they correspond to the respective set operations in terms of *adjacency.adjacency-list*. Let us first look at inserting an edge into a graph.

definition (**in** *adjacency*) *insert* :: *'a* × *'a* ⇒ *'m* ⇒ *'m* **where**

```
  insert p G ≡
   let u = fst p; v = snd p
   in let s = case Map-lookup G u of None ⇒ Set-empty | Some s' ⇒ s'
      in Map-update u (Set-insert v s) G

lemma (in adjacency) invar-insert:
  assumes invar G
  shows invar (insert p G)

lemma (in adjacency) adjacency-list-insert-cong:
  assumes invar G
  shows
    adjacency-list (insert p G) w =
     (if w = fst p then ins-list (snd p) (adjacency-list G w) else adjacency-list G w)

lemma (in adjacency) adjacency-insert-cong:
  assumes invar G
  shows
    set (adjacency-list (insert p G) u) =
     set (adjacency-list G u) ∪ (if u = fst p then {snd p} else {})

lemma (in adjacency) invar-fold-insert:
  assumes invar G
  shows invar (fold insert l G)

lemma (in adjacency) adjacency-fold-insert-cong:
  assumes invar G
  shows
    set (adjacency-list (fold insert l G) v) =
     set (adjacency-list G v) ∪ (⋃p∈set l. if v = fst p then {snd p} else {})

definition (in adjacency) insert' :: 'a ⇒ 'a ⇒ 'm ⇒ 'm where
  insert' ≡ curry insert

lemma (in adjacency) invar-insert':
  assumes invar G
  shows invar (insert' u v G)

lemma (in adjacency) adjacency-list-insert'-cong:
  assumes invar G
  shows
    adjacency-list (insert' u v G) w =
     (if w = u then ins-list v (adjacency-list G w) else adjacency-list G w)

lemma (in adjacency) adjacency-insert'-cong:
  assumes invar G
  shows
    set (adjacency-list (insert' u v G) w) =
     set (adjacency-list G w) ∪ (if w = u then {v} else {})
```

lemma (**in** *adjacency*) *invar-fold-insert'*:
assumes *invar G*
shows *invar* (*fold* (*insert' u*) *l G*)

lemma (**in** *adjacency*) *adjacency-fold-insert'-cong*:
assumes *invar G*
shows
set (*adjacency-list* (*fold* (*insert' u*) *l G*) *v*) =
set (*adjacency-list G v*) ∪ (⋃ *w*∈*set l*. *if v* = *u then* {*w*} *else* {})

Next, let us look at deleting an edge from a graph.

definition (**in** *adjacency*) *delete* :: *'a* × *'a* ⇒ *'m* ⇒ *'m* **where**
delete p G ≡
case Map-lookup G (*fst p*) *of*
None ⇒ *G* |
Some s ⇒ *Map-update* (*fst p*) (*Set-delete* (*snd p*) *s*) *G*

lemma (**in** *adjacency*) *invar-delete*:
assumes *invar G*
shows *invar* (*delete p G*)

lemma (**in** *adjacency*) *adjacency-list-delete-cong*:
assumes *invar G*
shows
adjacency-list (*delete p G*) *w* =
(*if w* = *fst p then List-Ins-Del.del-list* (*snd p*) (*adjacency-list G w*) *else adjacency-list G w*)

definition (**in** *adjacency*) *delete'* :: *'a* ⇒ *'a* ⇒ *'m* ⇒ *'m* **where**
delete' ≡ *curry delete*

lemma (**in** *adjacency*) *invar-delete'*:
assumes *invar G*
shows *invar* (*delete' u v G*)

lemma (**in** *adjacency*) *adjacency-list-delete'-cong*:
assumes *invar G*
shows
adjacency-list (*delete' u v G*) *w* =
(*if w* = *u then List-Ins-Del.del-list v* (*adjacency-list G w*) *else adjacency-list G w*)

Let us now look at computing the union two graphs.

definition (**in** *adjacency*) *insert-2* :: *'a* × *'s* ⇒ *'m* ⇒ *'m* **where**
insert-2 p G ≡
let v = *fst p*; *s* = *snd p*

```
    in let s' = case Map-lookup G v of None ⇒ s | Some s'' ⇒ fold Set-insert
(Set-inorder s) s''
      in Map-update v s' G

lemma (in adjacency) invar-insert-2:
  assumes invar G
  assumes S.invar (snd p)
  shows invar (insert-2 p G)

lemma (in adjacency) adjacency-insert-2-cong:
  assumes invar G
  assumes S.invar (snd p)
  shows
    set (adjacency-list (insert-2 p G) u) =
     set (adjacency-list G u) ∪ (if u = fst p then S.set (snd p) else {})

lemma (in adjacency) invar-fold-insert-2:
  assumes invar G
  assumes Ball (set l) (S.invar ∘ snd)
  shows invar (fold insert-2 l G)

lemma (in adjacency) adjacency-fold-insert-2-cong:
  assumes invar G
  assumes Ball (set l) (S.invar ∘ snd)
  shows
    set (adjacency-list (fold insert-2 l G) v) =
     set (adjacency-list G v) ∪ (⋃p∈set l. if v = fst p then S.set (snd p) else {})

definition (in adjacency) union :: 'm ⇒ 'm ⇒ 'm where
  union G1 G2 ≡ fold insert-2 (Map-inorder G2) G1

lemma (in adjacency) invar-union:
  assumes invar G1
  assumes invar G2
  shows invar (union G1 G2)

lemma (in adjacency) adjacency-union-cong:
  assumes invar G1
  assumes invar G2
  shows
    set (adjacency-list (union G1 G2) v) =
     set (adjacency-list G1 v) ∪ set (adjacency-list G2 v)
```

Finally, let us look at computing the difference of two graphs.

```
definition (in adjacency) delete-2 :: 'a × 's ⇒ 'm ⇒ 'm where
  delete-2 p G ≡
   let v = fst p; s = snd p
   in case Map-lookup G v of
```

None ⇒ *G* |
Some s' ⇒ *Map-update v* (*fold Set-delete* (*Set-inorder s*) *s'*) *G*

lemma (**in** *adjacency*) *invar-delete-2*:
assumes *invar G*
shows *invar* (*delete-2 p G*)

lemma (**in** *adjacency*) *adjacency-delete-2-cong*:
assumes *invar G*
shows
set (*adjacency-list* (*delete-2 p G*) *u*) =
set (*adjacency-list G u*) − (*if u* = *fst p then S.set* (*snd p*) *else* {})

lemma (**in** *adjacency*) *invar-fold-delete-2*:
assumes *invar G*
assumes *Ball* (*set l*) (*S.invar* ∘ *snd*)
shows *invar* (*fold delete-2 l G*)

lemma (**in** *adjacency*) *adjacency-fold-delete-2-cong*:
assumes *invar G*
assumes *Ball* (*set l*) (*S.invar* ∘ *snd*)
shows
set (*adjacency-list* (*fold delete-2 l G*) *v*) =
set (*adjacency-list G v*) − (⋃*p*∈*set l*. *if v* = *fst p then S.set* (*snd p*) *else* {})

definition (**in** *adjacency*) *difference* :: *'m* ⇒ *'m* ⇒ *'m* **where**
difference G1 G2 ≡ *fold delete-2* (*Map-inorder G2*) *G1*

lemma (**in** *adjacency*) *invar-difference*:
assumes *invar G1*
assumes *invar G2*
shows *invar* (*difference G1 G2*)

lemma (**in** *adjacency*) *adjacency-difference-cong*:
assumes *invar G1*
assumes *invar G2*
shows
set (*adjacency-list* (*difference G1 G2*) *v*) =
set (*adjacency-list G1 v*) − *set* (*adjacency-list G2 v*)

We show that our specifications of operations insert and delete satisfy all assumptions of locale *Finite-Adjacency-Structure*.

context *adjacency*
begin
sublocale *G*: *Finite-Adjacency-Structure* **where**
empty = *Map-empty* **and**
insert = *insert'* **and**

delete = *delete'* **and**
adj = (λ*v* *G*. *adjacency-list* *G* *v*) **and**
inv = *invar*
end

end

1.1.2 Directed adjacency structure

theory *Directed-Adjacency*
imports
Adjacency
../Directed-Graph/Dgraph
../Directed-Graph/Dpath
begin

An adjacency structure specified via the locale *adjacency* naturally induces a directed graph, where we have an edge from vertex u to vertex v if and only if v is contained in the adjacency of u.

definition (**in** *adjacency*) *dE* :: *'m* ⇒ (*'a* × *'a*) *set* **where**
dE *G* ≡ {(*u*, *v*). *v* ∈ *set* (*adjacency-list* *G* *u*)}

definition (**in** *adjacency*) *dV* :: *'m* ⇒ *'a set* **where**
dV *G* ≡ *dVs* (*dE* *G*)

lemma (**in** *adjacency*) *mem-adjacency-iff-edge*:
shows *v* ∈ *set* (*adjacency-list* *G* *u*) ⟷ (*u*, *v*) ∈ *dE* *G*

lemma (**in** *adjacency*) *finite-dE*:

assumes *invar G*
shows *finite* (*dE G*)

lemma (**in** *adjacency*) *adjacency-subset-dV*:
shows *set* (*adjacency-list G v*) ⊆ *dV G*

lemma (**in** *adjacency*) *finite-dV*:
assumes *invar G*
shows *finite* (*dV G*)

We show that graph operations union and difference correspond to the respective set operations in terms of *adjacency.dE*.

lemma (**in** *adjacency*) *dE-union-cong*:
assumes *invar G1*
assumes *invar G2*
shows *dE* (*union G1 G2*) = *dE G1* ∪ *dE G2*

lemma (**in** *adjacency*) *dV-union-cong*:
assumes *invar G1*
assumes *invar G2*
shows *dV* (*union G1 G2*) = *dV G1* ∪ *dV G2*

lemma (**in** *adjacency*) *finite-dE-union*:
assumes *invar G1*
assumes *invar G2*
shows *finite* (*dE* (*union G1 G2*))

lemma (**in** *adjacency*) *finite-dV-union*:
assumes *invar G1*
assumes *invar G2*
shows *finite* (*dV* (*union G1 G2*))

lemma (**in** *adjacency*) *dE-difference-cong*:
assumes *invar G1*
assumes *invar G2*
shows *dE* (*difference G1 G2*) = *dE G1* − *dE G2*

lemma (**in** *adjacency*) *finite-dE-difference*:
assumes *invar G1*
assumes *invar G2*
shows *finite* (*dE* (*difference G1 G2*))

lemma (**in** *adjacency*) *finite-dV-difference*:
assumes *invar G1*
assumes *invar G2*
shows *finite* (*dV* (*difference G1 G2*))

end

1.1.3 Undirected adjacency structure

theory *Undirected-Adjacency*
imports
Adjacency
AGF.Berge
../Undirected-Graph/Graph-Ext
begin

If the adjacency structure is symmetric, then it induces an undirected graph.

locale *adjacency'* = *adjacency* **where**
Map-update = *Map-update* **for**
Map-update :: *'a*::*linorder* ⇒ *'t* ⇒ *'m* ⇒ *'m* +
fixes *G* :: *'m*
assumes *invar*: *invar G*

locale *symmetric-adjacency* = *adjacency'* **where**
Map-update = *Map-update* **for**
Map-update :: *'a*::*linorder* ⇒ *'t* ⇒ *'m* ⇒ *'m* +
assumes *symmetric*: *v* ∈ *set* (*adjacency-list G u*) ⟷ *u* ∈ *set* (*adjacency-list G v*)

definition (**in** *adjacency*) *E* :: *'m* ⇒ *'a set set* **where**
E G ≡ {{*u*, *v*} |*u v*. *v* ∈ *set* (*adjacency-list G u*)}

definition (**in** *adjacency*) *V* :: *'m* ⇒ *'a set* **where**
V G ≡ *Vs* (*E G*)

lemma (**in** *adjacency*) *finite-E*:
assumes *invar G*
shows *finite* (*E G*)

lemma (**in** *symmetric-adjacency*) *mem-adjacency-iff-edge*:
shows *v* ∈ *set* (*adjacency-list G u*) ⟷ {*u*, *v*} ∈ *E G*

lemma (**in** *symmetric-adjacency*) *mem-adjacency-iff-edge-2*:
shows *u* ∈ *set* (*adjacency-list G v*) ⟷ {*u*, *v*} ∈ *E G*

lemma (**in** *adjacency*) *finite-V*:
assumes *invar G*
shows *finite* (*V G*)

context *adjacency'*
begin
sublocale *finite-graph E G*
end

We redefine graph operation insert such that it maintains symmetry.

definition (**in** *adjacency*) *insert-edge* :: *'a* ⇒ *'a* ⇒ *'m* ⇒ *'m* **where**
insert-edge u v G ≡ *insert' v u* (*insert' u v G*)

lemma (**in** *adjacency*) *invar-insert-edge*:
assumes *invar G*
shows *invar* (*insert-edge u v G*)

lemma (**in** *adjacency*) *adjacency-insert-edge-cong*:
assumes *invar G*
shows
set (*adjacency-list* (*insert-edge u v G*) *w*) =
set (*adjacency-list G w*) ∪ (*if w* = *u then* {*v*} *else if w* = *v then* {*u*} *else* {})

lemma (**in** *adjacency*) *E-insert-edge-cong*:
assumes *invar G*
shows *E* (*insert-edge u v G*) = *E G* ∪ {{*u*, *v*}}

lemma (**in** *adjacency*) *invar-fold-insert-edge*:
assumes *invar G*
shows *invar* (*fold* (*insert-edge u*) *l G*)

lemma (**in** *adjacency*) *adjacency-fold-insert-edge-cong*:
assumes *invar G*
shows
set (*adjacency-list* (*fold* (*insert-edge u*) *l G*) *v*) =
set (*adjacency-list G v*) ∪
(⋃*w*∈*set l*. *if v* = *u then* {*w*} *else if v* = *w then* {*u*} *else* {})

lemma (**in** *adjacency*) *E-fold-insert-edge-cong*:
assumes *invar G*
shows *E* (*fold* (*insert-edge u*) *l G*) = *E G* ∪ {{*u*, *v*} |*v*. *v* ∈ *set l*}

Next, we show that graph operations union and difference correspond to the respective set operations in terms of *adjacency.E*, and that they maintain symmetry.

lemma (**in** *adjacency*) *E-union-cong*:
assumes *invar G1*
assumes *invar G2*
shows *E* (*union G1 G2*) = *E G1* ∪ *E G2*

lemma (**in** *adjacency*) *V-union-cong*:
assumes *invar G1*
assumes *invar G2*
shows *V* (*union G1 G2*) = *V G1* ∪ *V G2*

lemma (**in** *adjacency*) *finite-V-union*:
assumes *invar G1*
assumes *invar G2*
shows *finite* (*V* (*union G1 G2*))

lemma (**in** *adjacency*) *symmetric-adjacency-union*:
assumes *symmetric-adjacency' G1*
assumes *symmetric-adjacency' G2*
shows *symmetric-adjacency' (union G1 G2)*

lemma (**in** *adjacency*) *symmetric-adjacency-difference*:
assumes *symmetric-adjacency' G1*
assumes *symmetric-adjacency' G2*
shows *symmetric-adjacency' (difference G1 G2)*

lemma (**in** *adjacency*) *E-difference-cong*:
assumes *symmetric-adjacency' G1*
assumes *symmetric-adjacency' G2*
shows *E (difference G1 G2) = E G1 − E G2*

lemma (**in** *adjacency*) *finite-V-difference*:
assumes *invar G1*
assumes *invar G2*
shows *finite (V (difference G1 G2))*

end
theory *Shortest-Dpath*
imports
../../Misc-Ext
Ports.Mitja-to-DDFS
Ports.Noschinski-to-DDFS
Weighted-Dpath
begin

We extend theory *Ports.Mitja-to-DDFS* and formalize shortest directed paths.

definition *δ :: 'a dgraph ⇒ 'a weight-fun ⇒ 'a ⇒ 'a ⇒ enat* **where**
δ G f u v ≡ INF p∈{p. dpath-bet G p u v}. enat (dpath-weight f p)

definition *is-shortest-dpath :: 'a dgraph ⇒ 'a weight-fun ⇒ 'a dpath ⇒ 'a ⇒ 'a ⇒ bool* **where**
is-shortest-dpath G f p u v ≡ dpath-bet G p u v ∧ dpath-weight f p = δ G f u v

definition *dist :: 'a dgraph ⇒ 'a ⇒ 'a ⇒ enat* **where**
dist G u v ≡ INF p∈{p. dpath-bet G p u v}. enat (dpath-length p)

theorem *dist-eq-δ*:
shows *dist G = δ G (λ-. 1)*

lemma (**in** *finite-dgraph*) *dist-le-dpath-length*:
assumes *dpath-bet G p u v*
shows *dist G u v ≤ dpath-length p*

lemma (**in** *finite-dgraph*) *is-shortest-dpath-if-reachable-2*:
assumes *reachable G u v*

obtains p **where**
dpath-bet G p u v
dpath-length p = dist G u v

lemma (**in** *finite-dgraph*) *is-shortest-dpathE-2*:
assumes *dpath-bet G (p @ [v] @ q) u w* ∧ *dpath-length (p @ [v] @ q) = dist G u w*
obtains
dpath-bet G (p @ [v]) u v ∧ *dpath-length (p @ [v]) = dist G u v*
dpath-bet G (v # q) v w ∧ *dpath-length (v # q) = dist G v w*
dist G u w = dist G u v + dist G v w

lemma (**in** *finite-dgraph*) *dist-triangle-inequality-edge*:
assumes $(v, w) \in G$
shows *dist G u w* ≤ *dist G u v + 1*

end

1.1.4 Directed graphs

theory *Directed-Graph*
imports
Shortest-Dpath
begin

end

2 Queue

This section considers first-in first-out queues from three levels of abstraction.

theory *Queue-Specs*
imports *Main*
begin

On the high level, a queue is a list (*list*). On the medium level, a queue is specified via the following interface.

locale *Queue* =
fixes *empty* :: *'q*
fixes *is-empty* :: *'q* ⇒ *bool*
fixes *snoc* :: *'q* ⇒ *'a* ⇒ *'q*
fixes *head* :: *'q* ⇒ *'a*
fixes *tail* :: *'q* ⇒ *'q*
fixes *invar* :: *'q* ⇒ *bool*
fixes *list* :: *'q* ⇒ *'a list*
assumes *list-empty*: *list empty = Nil*
assumes *is-empty*: *invar q* ⟹ *is-empty q = (list q = Nil)*
assumes *list-snoc*: *invar q* ⟹ *list (snoc q x) = list q @ [x]*

assumes *list-head*: ⟦ *invar q*; *list q* ≠ *Nil* ⟧ ⟹ *head q* = *hd* (*list q*)
assumes *list-tail*: ⟦ *invar q*; *list q* ≠ *Nil* ⟧ ⟹ *list* (*tail q*) = *tl* (*list q*)
assumes *invar-empty*: *invar empty*
assumes *invar-snoc*: *invar q* ⟹ *invar* (*snoc q x*)
assumes *invar-tail*: ⟦ *invar q*; *list q* ≠ *Nil* ⟧ ⟹ *invar* (*tail q*)

end

3 BFS

This section specifies and verifies breadth-first search (BFS). More specifically, we verify that given a directed graph G and a source vertex s, the output of the algorithm induces a breadth-first tree T of G w.r.t. s, that is, T consists of the vertices reachable from s in G, and for every vertex v in T, T contains a unique simple path from s to v that is also a shortest path from s to v in G.

theory *BFS*
imports
../Graph/Adjacency/Directed-Adjacency
../Graph/Directed-Graph/Directed-Graph
../Map/Map-Specs-Ext
Parent-Relation
../Queue/Queue-Specs
begin

3.1 Specification of the algorithm

record ('*q*, '*m*) *state* =
queue :: '*q*
parent :: '*m*

locale *bfs* =
G: *adjacency* **where** *Map-update* = *Map-update* +
P: *Map* **where**
empty = *P-empty* **and**
update = *P-update* **and**
delete = *P-delete* **and**
lookup = *P-lookup* **and**
invar = *P-invar* +
Q: *Queue* **where**
empty = *Q-empty* **and**
is-empty = *Q-is-empty* **and**
snoc = *Q-snoc* **and**
head = *Q-head* **and**
tail = *Q-tail* **and**
invar = *Q-invar* **and**
list = *Q-list* **for**
Map-update :: '*a*::*linorder* ⇒ '*s* ⇒ '*n* ⇒ '*n* **and**

```
  P-empty and
  P-update :: 'a ⇒ 'a ⇒ 'm ⇒ 'm and
  P-delete and
  P-lookup and
  P-invar and
  Q-empty and
  Q-is-empty and
  Q-snoc :: 'q ⇒ 'a ⇒ 'q and
  Q-head and
  Q-tail and
  Q-invar and
  Q-list
begin
```

Our specification of BFS maintains a first-in first-out queue, initialized to contain the source vertex *src*, and a parent map, initialized to the empty map. As long as the queue is not empty, the algorithm pops the head u of the queue, and for every adjacent vertex v, discovers v if it hasn't been discovered yet, where discovering v entails adding v to the queue as well as setting v's parent to u.

```
definition init :: 'a ⇒ ('q, 'm) state where
  init src ≡
   (|queue = Q-snoc Q-empty src,
    parent = P-empty|)

definition DONE :: ('q, 'm) state ⇒ bool where
  DONE s ⟷ Q-is-empty (queue s)

definition is-discovered :: 'a ⇒ 'm ⇒ 'a ⇒ bool where
  is-discovered src m v ⟷ v = src ∨ P-lookup m v ≠ None

definition discover :: 'a ⇒ 'a ⇒ ('q, 'm) state ⇒ ('q, 'm) state where
  discover u v s ≡
   (|queue = Q-snoc (queue s) v,
    parent = P-update v u (parent s)|)

definition traverse-edge :: 'a ⇒ 'a ⇒ 'a ⇒ ('q, 'm) state ⇒ ('q, 'm) state where
  traverse-edge src u v s ≡
   if ¬ is-discovered src (parent s) v then discover u v s
   else s

function (domintros) loop :: 'n ⇒ 'a ⇒ ('q, 'm) state ⇒ ('q, 'm) state where
  loop G src s =
   (if ¬ DONE s
    then let
          u = Q-head (queue s);
          q = Q-tail (queue s)
         in loop G src (fold (traverse-edge src u) (G.adjacency-list G u) (s(|queue :=
q|)))
```

else s)

definition *bfs* :: *'n* ⇒ *'a* ⇒ *'m* **where**
bfs G src ≡ *parent* (*loop G src* (*init src*))

abbreviation *fold* :: *'n* ⇒ *'a* ⇒ (*'q*, *'m*) *state* ⇒ (*'q*, *'m*) *state* **where**
fold G src s ≡
List.fold
(*traverse-edge src* (*Q-head* (*queue s*)))
(*G.adjacency-list G* (*Q-head* (*queue s*)))
(*s*(|*queue* := *Q-tail* (*queue s*)|))

abbreviation *T* :: *'m* ⇒ *'a dgraph* **where**
T m ≡ {(*u*, *v*). *P-lookup m v* = *Some u*}

end

3.2 Verification of the correctness of the algorithm

3.2.1 Assumptions on the input

Algorithm *bfs.bfs* expects a directed graph G and a source vertex *src* in G as input, and the correctness theorem will assume such an input. We remark that the assumption that *src* is indeed a vertex in G is for the purpose of convenience. Let us formally specify these assumptions.

locale *bfs-valid-input* = *bfs* **where**
Map-update = *Map-update* **and**
P-update = *P-update* **and**
Q-snoc = *Q-snoc* **for**
Map-update :: *'a*::*linorder* ⇒ *'s* ⇒ *'n* ⇒ *'n* **and**
P-update :: *'a* ⇒ *'a* ⇒ *'m* ⇒ *'m* **and**
Q-snoc :: *'q* ⇒ *'a* ⇒ *'q* +
fixes *G* :: *'n*
fixes *src* :: *'a*
assumes *invar-G*: *G.invar G*
assumes *src-mem-dV*: *src* ∈ *G.dV G*

Graph G is represented as an *adjacency*, that is, as a *Map-by-Ordered* mapping a vertex to its adjacency, which is represented as a *Set-by-Ordered*.

3.2.2 Loop invariants

Unfolding the definition of algorithm *bfs.bfs*, we see that recursive function *bfs.loop* lies at the heart of the algorithm. It expects an input $\langle G, src, s\rangle$, which constitutes the program state, such that

- G, *src* satisfy the assumptions specified above, and

- s comprises a queue and a parent map satisfying the assumptions stated below.

As s is the only state variable that is subject to change from one iteration to the next, the following assumptions constitute the (non-trivial) loop invariants of *bfs.loop*.

abbreviation (**in** *bfs-valid-input*) *white* :: ('*q*, '*m*) *state* ⇒ '*a* ⇒ *bool* **where**
white s v ≡ ¬ *is-discovered src* (*parent s*) *v*

abbreviation (**in** *bfs-valid-input*) *gray* :: ('*q*, '*m*) *state* ⇒ '*a* ⇒ *bool* **where**
gray s v ≡ *is-discovered src* (*parent s*) *v* ∧ *v* ∈ *set* (*Q-list* (*queue s*))

abbreviation (**in** *bfs-valid-input*) *black* :: ('*q*, '*m*) *state* ⇒ '*a* ⇒ *bool* **where**
black s v ≡ *is-discovered src* (*parent s*) *v* ∧ *v* ∉ *set* (*Q-list* (*queue s*))

abbreviation (**in** *bfs*) *rev-follow* :: '*m* ⇒ '*a* ⇒ '*a dpath* **where**
rev-follow m v ≡ *rev* (*parent.follow* (*P-lookup m*) *v*)

abbreviation (**in** *bfs-valid-input*) *d* :: '*m* ⇒ '*a* ⇒ *nat* **where**
d m v ≡ *dpath-length* (*rev-follow m v*)

locale *bfs-invar* =
bfs-valid-input **where** *P-update* = *P-update* **and** *Q-snoc* = *Q-snoc* +
parent P-lookup (*parent s*) **for**
P-update :: '*a*::*linorder* ⇒ '*a* ⇒ '*m* ⇒ '*m* **and**
Q-snoc :: '*q* ⇒ '*a* ⇒ '*q* **and**
s :: ('*q*, '*m*) *state* +
assumes *invar-queue*: *Q-invar* (*queue s*)
assumes *invar-parent*: *P-invar* (*parent s*)
assumes *parent-src*: *P-lookup* (*parent s*) *src* = *None*
assumes *parent-imp-edge*: *P-lookup* (*parent s*) *v* = *Some u* ⟹ (*u*, *v*) ∈ *G.dE G*
assumes *not-white-if-mem-queue*: *v* ∈ *set* (*Q-list* (*queue s*)) ⟹ ¬ *white s v*
assumes *not-white-if-parent*: *P-lookup* (*parent s*) *v* = *Some u* ⟹ ¬ *white s u*
assumes *black-imp-adjacency-not-white*: ⟦ (*u*, *v*) ∈ *G.dE G*; *black s u* ⟧ ⟹ ¬ *white s v*
assumes *queue-sorted-wrt-d*: *sorted-wrt* (λ*u v*. *d* (*parent s*) *u* ≤ *d* (*parent s*) *v*) (*Q-list* (*queue s*))
assumes *d-last-queue-le*:
¬ *Q-is-empty* (*queue s*) ⟹
d (*parent s*) (*last* (*Q-list* (*queue s*))) ≤ *d* (*parent s*) (*Q-head* (*queue s*)) + *1*
assumes *d-triangle-inequality*:
⟦ *dpath-bet* (*G.dE G*) *p u v*; ¬ *white s u*; ¬ *white s v* ⟧ ⟹
d (*parent s*) *v* ≤ *d* (*parent s*) *u* + *dpath-length p*

As mentioned above, state s comprises a queue, represented as a *Queue*, and a map, represented as a *Map*.

Invariant *bfs-invar.black-imp-adjacency-not-white* says that all vertices ad-

jacent to a black vertex have been discovered.

For a vertex v, let $d(v) = d\ (state.parent\ s)\ v$ denote the distance from the source *src* to v induced by the current state s.

Let $\langle v_1, \ldots, v_k \rangle$ be the contents of the queue, where v_1 is the head. Then invariant *bfs-invar.queue-sorted-wrt-d* says that $d(v_i) \leq d(v_{i+1})$ for all $i < k$. And invariant *bfs-invar.d-last-queue-le* says that $d(v_k) \leq d(v_1) + 1$. That is, the current queue holds at most two distinct d values.

Finally, invariant *bfs-invar.d-triangle-inequality* says that d satisfies a variant of the triangle inequality. More specifically, if there is a path in G between two vertices u, v that both have been discovered by the algorithm, then their d values differ by at most the length of that path.

To verify the correctness of loop *bfs.loop*, we need to show that

1. the loop invariants are satisfied prior to the first iteration of the loop, and that
2. the loop invariants are maintained.

Let us start with the former, that is, let us prove that the initial configurations of the queue–containing only the source vertex *src*–and parent map–the empty map–satisfy the loop invariants.

lemma (**in** *bfs-valid-input*) *bfs-invar-init*:
shows *bfs-invar''* (*init src*)

Let us now verify that the loop invariants are maintained, that is, if they hold at the start of an iteration of loop *bfs.loop*, then they will also hold at the end. For this, let us first look at how the different subroutines change the queue and parent map.

lemma (**in** *bfs*) *queue-discover-cong* [*simp*]:
shows *queue* (*discover u v s*) = *Q-snoc* (*queue s*) *v*

lemma (**in** *bfs*) *parent-discover-cong* [*simp*]:
shows *parent* (*discover u v s*) = *P-update v u* (*parent s*)

lemma (**in** *bfs*) *queue-traverse-edge-cong*:
shows *queue* (*traverse-edge src u v s*) = (*if* ¬ *is-discovered src* (*parent s*) *v then Q-snoc* (*queue s*) *v else queue s*)

lemma (**in** *bfs*) *list-queue-traverse-edge-cong*:
assumes *Q-invar* (*queue s*)
shows
Q-list (*queue* (*traverse-edge src u v s*)) =

Q-list (*queue s*) @ (*if* ¬ *is-discovered src* (*parent s*) *v then* [*v*] *else* [])

lemma (**in** *bfs*) *lookup-parent-traverse-edge-cong*:
assumes *P-invar* (*parent s*)
shows
P-lookup (*parent* (*traverse-edge src u v s*)) =
override-on
(*P-lookup* (*parent s*))
(λ-. *Some u*)
(*if* ¬ *is-discovered src* (*parent s*) *v then* {*v*} *else* {})

lemma (**in** *bfs*) *T-traverse-edge-cong*:
assumes *P-invar* (*parent s*)
shows *T* (*parent* (*traverse-edge src u v s*)) = *T* (*parent s*) ∪ (*if* ¬ *is-discovered src* (*parent s*) *v then* {(*u*, *v*)} *else* {})

lemma (**in** *bfs*) *list-queue-fold-cong*:
assumes *Q-invar* (*queue s*)
assumes *P-invar* (*parent s*)
assumes *distinct l*
shows
Q-list (*queue* (*List.fold* (*traverse-edge src u*) *l s*)) =
Q-list (*queue s*) @ *filter* (*Not* ∘ *is-discovered src* (*parent s*)) *l*

lemma (**in** *bfs*) *list-queue-fold-cong-2*:
assumes *G.invar G*
assumes *Q-invar* (*queue s*)
assumes *P-invar* (*parent s*)
assumes ¬ *DONE s*
shows
Q-list (*queue* (*fold G src s*)) =
Q-list (*Q-tail* (*queue s*)) @
filter (*Not* ∘ *is-discovered src* (*parent s*)) (*G.adjacency-list G* (*Q-head* (*queue s*)))

lemma (**in** *bfs*) *lookup-parent-fold-cong*:
assumes *P-invar* (*parent s*)
assumes *distinct l*
shows
P-lookup (*parent* (*List.fold* (*traverse-edge src u*) *l s*)) =
override-on
(*P-lookup* (*parent s*))
(λ-. *Some u*)
(*set* (*filter* (*Not* ∘ *is-discovered src* (*parent s*)) *l*))

lemma (**in** *bfs*) *lookup-parent-fold-cong-2*:
assumes *G.invar G*
assumes *P-invar* (*parent s*)
shows

P-lookup (*parent* (*fold G src s*)) =
override-on
(*P-lookup* (*parent s*))
(*λ-. Some* (*Q-head* (*queue s*)))
(*set* (*filter* (*Not* ∘ *is-discovered src* (*parent s*)) (*G.adjacency-list G* (*Q-head* (*queue s*)))))

lemma (**in** *bfs-invar*) *lookup-parent-fold-cong*:
shows
P-lookup (*parent* (*fold G src s*)) =
override-on
(*P-lookup* (*parent s*))
(*λ-. Some* (*Q-head* (*queue s*)))
(*set* (*filter* (*Not* ∘ *is-discovered src* (*parent s*)) (*G.adjacency-list G* (*Q-head* (*queue s*)))))

lemma (**in** *bfs*) *T-fold-cong*:
assumes *P-invar* (*parent s*)
assumes *distinct l*
shows *T* (*parent* (*List.fold* (*traverse-edge src u*) *l s*)) = *T* (*parent s*) ∪ {(*u*, *v*) |*v*. *v* ∈ *set l* ∧ ¬ *is-discovered src* (*parent s*) *v*}

lemma (**in** *bfs*) *T-fold-cong-2*:
assumes *G.invar G*
assumes *P-invar* (*parent s*)
shows
T (*parent* (*fold G src s*)) =
T (*parent s*) ∪
{(*Q-head* (*queue s*), *v*) |*v*. *v* ∈ *set* (*G.adjacency-list G* (*Q-head* (*queue s*))) ∧ ¬ *is-discovered src* (*parent s*) *v*}

lemma (**in** *bfs-invar*) *T-fold-cong*:
shows
T (*parent* (*fold G src s*)) =
T (*parent s*) ∪
{(*Q-head* (*queue s*), *v*) |*v*. *v* ∈ *set* (*G.adjacency-list G* (*Q-head* (*queue s*))) ∧ ¬ *is-discovered src* (*parent s*) *v*}

Next, we verify the maintenance of the loop invariants one by one.

locale *bfs-invar-not-DONE* = *bfs-invar* **where** *P-update* = *P-update* **and** *Q-snoc* = *Q-snoc* **for**
P-update :: *'a*::*linorder* ⇒ *'a* ⇒ *'m* ⇒ *'m* **and**
Q-snoc :: *'q* ⇒ *'a* ⇒ *'q* +
assumes *not-DONE*: ¬ *DONE s*

lemma (**in** *bfs-invar-not-DONE*) *follow-invar-parent-fold*:
shows *follow-invar* (*P-lookup* (*parent* (*fold G src s*)))

lemma (**in** *bfs-invar-not-DONE*) *invar-queue-fold*:
shows *Q-invar* (*queue* (*fold G src s*))

lemma (**in** *bfs-invar*) *invar-parent-fold*:
shows *P-invar* (*parent* (*fold G src s*))

lemma (**in** *bfs-invar*) *parent-src-fold*:
shows *P-lookup* (*parent* (*fold G src s*)) *src* = *None*

lemma (**in** *bfs-invar-not-DONE*) *parent-imp-edge-fold*:
assumes *P-lookup* (*parent* (*fold G src s*)) *v* = *Some u*
shows (*u*, *v*) ∈ *G.dE G*

lemma (**in** *bfs-invar-not-DONE*) *list-queue-fold-cong*:
shows
Q-list (*queue* (*fold G src s*)) =
Q-list (*Q-tail* (*queue s*)) @
filter (*Not* ∘ *is-discovered src* (*parent s*)) (*G.adjacency-list G* (*Q-head* (*queue s*)))

lemma (**in** *bfs-invar-not-DONE*) *not-white-if-mem-queue-fold*:
assumes *v* ∈ *set* (*Q-list* (*queue* (*fold G src s*)))
shows ¬ *white* (*fold G src s*) *v*

lemma (**in** *bfs-invar-not-DONE*) *not-white-if-parent-fold*:
assumes *P-lookup* (*parent* (*fold G src s*)) *v* = *Some u*
shows ¬ *white* (*fold G src s*) *u*

lemma (**in** *bfs-valid-input*) *vertex-color-induct* [*case-names white gray black*]:
assumes *white s v* ⟹ *P s v*
assumes *gray s v* ⟹ *P s v*
assumes *black s v* ⟹ *P s v*
shows *P s v*

lemma (**in** *bfs-invar-not-DONE*) *black-imp-adjacency-not-white-fold*:
assumes *black* (*fold G src s*) *u*
assumes (*u*, *v*) ∈ *G.dE G*
shows ¬ *white* (*fold G src s*) *v*

lemma (**in** *bfs-invar*) *not-white-imp-lookup-parent-fold-eq-lookup-parent*:
assumes ¬ *white s v*
shows *P-lookup* (*parent* (*fold G src s*)) *v* = *P-lookup* (*parent s*) *v*

lemma (**in** *bfs-invar-not-DONE*) *not-white-imp-rev-follow-fold-eq-rev-follow*:
assumes ¬ *white s v*
shows *rev-follow* (*parent* (*fold G src s*)) *v* = *rev-follow* (*parent s*) *v*

lemma (**in** *bfs-invar-not-DONE*) *queue-sorted-wrt-d-fold*:
shows *sorted-wrt* (λ*u v*. *d* (*parent* (*fold G src s*)) *u* ≤ *d* (*parent* (*fold G src s*)))

v) (*Q-list* (*queue* (*fold G src s*)))

lemma (**in** *bfs-invar-not-DONE*) *d-last-queue-le-fold*:
assumes ¬ *Q-is-empty* (*queue* (*fold G src s*))
shows *d* (*parent* (*fold G src s*)) (*last* (*Q-list* (*queue* (*fold G src s*)))) ≤ *d* (*parent* (*fold G src s*)) (*Q-head* (*queue* (*fold G src s*))) + *1*

The last invariant, *bfs-invar.d-triangle-inequality*, is, at least in our minds, the most interesting one. Our first attempt at this invariant was the following: If a vertex v has been discovered, then the path from *src* to v induced by the parent map (*λva. rev* (*parent.follow* (*state.parent s v*) *va*)) is a shortest path in graph G. This invariant was so strong, however, that it did not require using the induction hypothesis (of the induction rule given by *bfs.loop*) to prove one of the two implications of the correctness theorem. Indeed, the following invariant is sufficient to prove the implication, provided that it can be maintained: If there is an edge (u, v) in graph G such that both u and v have been discovered, then $d(v) \leq d(u) + 1$. However, we were not able to prove that this invariant can be maintained without requiring an additional invariant. Therefore, we generalized the invariant from edges to arbitrary paths in graph G, yielding invariant *bfs-invar.d-triangle-inequality*. We realized only recently that this generalization is strong enough to imply our first attempt, that is, we now have an invariant that is at least as strong as an invariant we deemed too strong.

lemma (**in** *bfs-invar*) *white-imp-gray-ancestor*:
assumes *dpath-bet* (*G.dE G*) *p u w*
assumes ¬ *white s u*
assumes *white s w*
obtains *v* **where**
v ∈ *set p*
gray s v
proof (*induct p arbitrary*: *w rule*: *dpath-rev-induct*)
case *1*
thus *?case*
by *simp*
next
case *2*
thus *?case*
using *hd-of-dpath-bet′ last-of-dpath-bet*
by (*fastforce intro*: *list-length-1*)
next
case (*3 v v′ l*)
show *?case*
proof (*induct s v rule*: *vertex-color-induct*)
case *white*
have *dpath-bet* (*G.dE G*) (*l* @ [*v*]) *u v*
using *3.prems*(*2*)

```
    by (intro dpath-bet-pref) simp
  with 3.prems(1)
  show ?case
    using 3.prems(3) white
    by (force intro: 3.hyps)
 next
  case gray
  thus ?case
    by (auto intro: 3.prems(1))
 next
  case black
  have (v, w) ∈ G.dE G
    using 3.prems(2)
    by (auto simp add: dpath-bet-def intro: dpath-snoc-edge-2)
  thus ?case
    using black black-imp-adjacency-not-white 3.prems(4)
    by blast
 qed
qed

lemma (in bfs-invar-not-DONE) d-triangle-inequality-fold:
 assumes dpath-p: dpath-bet (G.dE G) p u v
 assumes not-white-fold-u: ¬ white (fold G src s) u
 assumes not-white-fold-v: ¬ white (fold G src s) v
 shows d (parent (fold G src s)) v ≤ d (parent (fold G src s)) u + dpath-length p
proof −
 consider
   (white-white) white s u ∧ white s v |
   (white-not-white) white s u ∧ ¬ white s v |
   (gray-white) gray s u ∧ white s v |
   (black-white) black s u ∧ white s v |
   (not-white-not-white) ¬ white s u ∧ ¬ white s v
   by fast
 thus ?thesis
 proof (cases)
   case white-white
   hence d (parent (fold G src s)) v = d (parent (fold G src s)) (Q-head (queue
s)) + 1
     using not-white-fold-v
     by (intro white-not-white-foldD(3)) simp
   also have ... = d (parent (fold G src s)) u
     using white-white not-white-fold-u
     by (intro white-not-white-foldD(3)[symmetric]) simp
   finally show ?thesis
     by simp
 next
   case white-not-white
   hence dpath-Cons: dpath-bet (G.dE G) (Q-head (queue s) # p) (Q-head (queue
s)) v
```

```
    using not-white-fold-u white-not-white-foldD(1) dpath-p
    by (auto simp add: G.mem-adjacency-iff-edge intro: dpath-bet-ConsI)
  have d (parent (fold G src s)) v = d (parent s) v
    using white-not-white
    by (simp add: not-white-imp-rev-follow-fold-eq-rev-follow)
  also have ... ≤ d (parent s) (Q-head (queue s)) + dpath-length (Q-head (queue
s) # p)
    using not-white-head-queue white-not-white dpath-Cons
    by (auto intro: d-triangle-inequality)
  also have ... = d (parent s) (Q-head (queue s)) + 1 + dpath-length p
    using dpath-p
    by (simp add: dpath-length-Cons)
  also have ... = d (parent (fold G src s)) (Q-head (queue s)) + 1 + dpath-length
p
    using not-white-head-queue
    by (simp add: not-white-imp-rev-follow-fold-eq-rev-follow)
  also have ... = d (parent (fold G src s)) u + dpath-length p
    using white-not-white not-white-fold-u
    by (simp add: white-not-white-foldD(3))
  finally show ?thesis
    .
next
  case gray-white
  hence d (parent (fold G src s)) v = d (parent (fold G src s)) (Q-head (queue
s)) + 1
    using not-white-fold-v
    by (intro white-not-white-foldD(3)) simp
  also have ... = d (parent s) (Q-head (queue s)) + 1
    using not-white-head-queue
    by (auto simp add: not-white-imp-rev-follow-fold-eq-rev-follow)
  also have ... ≤ d (parent s) u + 1
    using gray-white
    by (intro mem-queue-imp-d-ge add-right-mono) simp
  also have ... = d (parent (fold G src s)) u + 1
    using gray-white
    by (simp add: not-white-imp-rev-follow-fold-eq-rev-follow)
  also have ... ≤ d (parent (fold G src s)) u + dpath-length p
    using dpath-p gray-white
    by (fastforce intro: dpath-length-geq-1I add-left-mono)
  finally show ?thesis
    .
next
  case black-white
  then obtain w where
    w ∈ set p and
    gray-w: gray s w
    using dpath-p
    by (elim white-imp-gray-ancestor) simp+
  then obtain q r where
```

```
    p = q @ tl r and
    dpath-q: dpath-bet (G.dE G) q u w and
    dpath-r: dpath-bet (G.dE G) r w v
    using dpath-p
    by (auto simp add: in-set-conv-decomp elim: dpath-bet-vertex-decompE)
  hence dpath-length-p: dpath-length p = dpath-length q + dpath-length r
    by (auto dest: dpath-betD(2−4) intro: dpath-length-append-2)

  have d (parent (fold G src s)) v = d (parent (fold G src s)) (Q-head (queue s))
+ 1
    using black-white not-white-fold-v
    by (intro white-not-white-foldD(3)) simp
  also have ... = d (parent s) (Q-head (queue s)) + 1
    using not-white-head-queue
    by (auto simp add: not-white-imp-rev-follow-fold-eq-rev-follow)
  also have ... ≤ d (parent s) w + 1
    using gray-w
    by (intro mem-queue-imp-d-ge add-right-mono) simp
  also have ... ≤ d (parent s) u + dpath-length q + 1
    using dpath-q black-white gray-w
    by (auto intro: d-triangle-inequality add-right-mono)
  also have ... ≤ d (parent s) u + dpath-length p
    using dpath-r gray-w black-white dpath-length-geq-1I
    by (fastforce simp add: dpath-length-p)
  also have ... = d (parent (fold G src s)) u + dpath-length p
    using black-white
    by (simp add: not-white-imp-rev-follow-fold-eq-rev-follow)
  finally show ?thesis
    .
 next
  case not-white-not-white
  hence d (parent (fold G src s)) v = d (parent s) v
    by (simp add: not-white-imp-rev-follow-fold-eq-rev-follow)
  also have ... ≤ d (parent s) u + dpath-length p
    using dpath-p not-white-not-white
    by (intro d-triangle-inequality) simp+
  also have ... = d (parent (fold G src s)) u + dpath-length p
    using not-white-not-white
    by (simp add: not-white-imp-rev-follow-fold-eq-rev-follow)
  finally show ?thesis
    .
 qed
qed

lemma (in bfs-invar-not-DONE) bfs-invar-fold:
 shows bfs-invar'' (fold G src s)
```

3.2.3 Termination

Before we can prove the correctness of loop *bfs.loop*, we need to prove that it terminates on appropriate inputs.

lemma (**in** *bfs*) *loop-dom*:
assumes *G.invar G*
assumes *Q-invar* (*queue s*)
assumes *P-invar* (*parent s*)
assumes *set* (*Q-list* (*queue s*)) ⊆ *G.dV G*
assumes *P.dom* (*parent s*) ⊆ *G.dV G*
shows *loop-dom* (*G*, *src*, *s*)
proof (*induct card* (*G.dV G*) + *length* (*Q-list* (*queue s*)) − *card* (*P.dom* (*parent s*))
arbitrary: *s*
rule: *less-induct*)
case *less*
show *?case*
proof (*cases DONE s*)
case *True*
thus *?thesis*
by (*blast intro*: *loop.domintros*)
next
case *False*
let *?u* = *Q-head* (*queue s*)
let *?q* = *Q-tail* (*queue s*)
let *?S* = *set* (*filter* (*Not* ∘ *is-discovered src* (*parent s*)) (*G.adjacency-list G* (*Q-head* (*queue s*))))

have *length* (*Q-list* (*queue* (*fold G src s*))) = *length* (*Q-list ?q*) + *card ?S*
using *less.prems*(*1*−*3*) *False G.distinct-adjacency-list*
by (*simp add*: *list-queue-fold-cong-2 distinct-card*[*symmetric*])
moreover have *card* (*P.dom* (*parent* (*fold G src s*))) = *card* (*P.dom* (*parent s*)) + *card ?S*
using *less.prems*(*1*, *3*, *5*)
by (*intro loop-dom-aux*)
ultimately have
card (*G.dV G*) + *length* (*Q-list* (*queue* (*fold G src s*))) − *card* (*P.dom* (*parent* (*fold G src s*))) =
card (*G.dV G*) + *length* (*Q-list ?q*) + *card ?S* − (*card* (*P.dom* (*parent s*)) + *card ?S*)
by *presburger*
also have ... = *card* (*G.dV G*) + *length* (*Q-list ?q*) − *card* (*P.dom* (*parent s*))
by *simp*
also have ... < *card* (*G.dV G*) + *length* (*Q-list* (*queue s*)) − *card* (*P.dom* (*parent s*))
using *less.prems False*
by (*intro loop-dom-aux-2*)
finally have
card (*G.dV G*) + *length* (*Q-list* (*queue* (*fold G src s*))) − *card* (*P.dom* (*parent*

(*fold G src s*))) <
card (*G.dV G*) + *length* (*Q-list* (*queue s*)) − *card* (*P.dom* (*parent s*))
.
thus *?thesis*
using *less.prems*
by (*intro invar-queue-fold-2 invar-parent-fold-2 queue-fold-subset-dV dom-parent-fold-subset-dV-2 less.hyps loop.domintros*)
qed
qed

lemma (**in** *bfs-invar*) *not-white-imp-dpath-rev-follow*:
assumes ¬ *white s v*
shows *dpath-bet* (*G.dE G*) (*rev-follow* (*parent s*) *v*) *src v*

3.2.4 Correctness

We are now finally ready to prove the correctness of algorithm *bfs.bfs*.

abbreviation (**in** *bfs*) *dist* :: *'n* ⇒ *'a* ⇒ *'a* ⇒ *enat* **where**
dist G ≡ *Shortest-Dpath.dist* (*G.dE G*)

abbreviation (**in** *bfs*) *is-shortest-dpath* :: *'n* ⇒ *'a list* ⇒ *'a* ⇒ *'a* ⇒ *bool* **where**
is-shortest-dpath G p u v ≡ *dpath-bet* (*G.dE G*) *p u v* ∧ *dpath-length p* = *dist G u v*

locale *bfs-invar-DONE* = *bfs-invar* **where** *P-update* = *P-update* **and** *Q-snoc* = *Q-snoc* **for**
P-update :: *'a*::*linorder* ⇒ *'a* ⇒ *'m* ⇒ *'m* **and**
Q-snoc :: *'q* ⇒ *'a* ⇒ *'q* +
assumes *DONE*: *DONE s*

context *bfs-valid-input*
begin
sublocale *finite-dgraph G.dE G*
end

lemma (**in** *bfs-invar*) *distinct-rev-follow*:
shows *distinct* (*rev-follow* (*parent s*) *v*)

lemma (**in** *bfs-invar-DONE*) *white-imp-not-reachable*:
assumes *white s v*
shows ¬ *reachable* (*G.dE G*) *src v*

lemma (**in** *bfs-valid-input*) *loop-complete*:
assumes *bfs-invar'' s*
assumes ¬ *is-discovered src* (*parent* (*loop G src s*)) *v*
shows ¬ *reachable* (*G.dE G*) *src v*

lemma (**in** *bfs-invar-DONE*) *not-white-imp-d-le-dist*:

assumes $\neg$ *white s v*
shows *d* (*parent s*) *v* $\leq$ *dist G src v*

lemma (**in** *bfs-invar-DONE*) *not-white-imp-is-shortest-dpath*:
assumes $\neg$ *white s v*
shows *is-shortest-dpath G* (*rev-follow* (*parent s*) *v*) *src v*

lemma (**in** *bfs-valid-input*) *loop-sound*:
assumes *bfs-invar''' s*
assumes *is-discovered src* (*parent* (*loop G src s*)) *v*
shows *is-shortest-dpath G* (*rev-follow* (*parent* (*loop G src s*)) *v*) *src v*

abbreviation (**in** *bfs*) *is-shortest-dpath-Map* :: *'n* $\Rightarrow$ *'a* $\Rightarrow$ *'m* $\Rightarrow$ *bool* **where**
is-shortest-dpath-Map G src m $\equiv$
$\forall$ *v*. (*is-discovered src m v* $\longrightarrow$ *is-shortest-dpath G* (*rev-follow m v*) *src v*) $\wedge$
($\neg$ *is-discovered src m v* $\longrightarrow$ $\neg$ *reachable* (*G.dE G*) *src v*)

lemma (**in** *bfs-valid-input*) *loop-correct*:
assumes *bfs-invar''' s*
shows *is-shortest-dpath-Map G src* (*parent* (*loop G src s*))

lemma (**in** *bfs-valid-input*) *bfs-correct*:
shows *is-shortest-dpath-Map G src* (*bfs G src*)

theorem (**in** *bfs*) *bfs-correct*:
assumes *bfs-valid-input' G src*
shows *is-shortest-dpath-Map G src* (*bfs G src*)

end
theory *Shortest-Path-Adaptor*
imports
Path-Adaptor
../Directed-Graph/Shortest-Dpath
../Undirected-Graph/Shortest-Path
begin

abbreviation *is-shortest-dpath* :: *'a dgraph* $\Rightarrow$ *'a dpath* $\Rightarrow$ *'a* $\Rightarrow$ *'a* $\Rightarrow$ *bool* **where**
is-shortest-dpath G p u v $\equiv$ *dpath-bet G p u v* $\wedge$ *dpath-length p* = *Shortest-Dpath.dist G u v*

lemma (**in** *graph*) *dist-eq-dist*:
shows *dist G u v* = *Shortest-Dpath.dist dEs u v*

lemma (**in** *graph*) *is-shortest-path-iff-is-shortest-dpath*:
shows *is-shortest-path G p u v* = *is-shortest-dpath dEs p u v*

end

4 Alternating BFS

This section specifies and verifies a modified BFS that alternates between edges in two given graphs.

theory *Alternating-BFS*
imports
../Graph/Undirected-Graph/Shortest-Alternating-Path
../BFS/Undirected-BFS
begin

4.1 Specification of the algorithm

locale *alt-bfs* = *bfs* **where**
Map-update = *Map-update* **and**
P-update = *P-update* **and**
Q-snoc = *Q-snoc* **for**
Map-update :: *'a::linorder* ⇒ *'s* ⇒ *'n* ⇒ *'n* **and**
P-update :: *'a* ⇒ *'a* ⇒ *'m* ⇒ *'m* **and**
Q-snoc :: *'q* ⇒ *'a* ⇒ *'q*
begin

Apart from enforcing alternation, the algorithm works identically to BFS.

thm *init-def*

thm *DONE-def*

thm *is-discovered-def*

thm *discover-def*

thm *traverse-edge-def*

And we enforce alternation by checking, when determining the vertices adjacent to a vertex *u*, how *u* was reached from its parent. If it was reached via an edge in *G1*, then we consider only vertices adjacent to *u* in *G2* and vice versa.

definition *P* :: *'n* ⇒ *'a* ⇒ *'a* ⇒ *bool* **where**
P G u v ≡ *case Map-lookup G u of None* ⇒ *False* | *Some s* ⇒ *Set-isin s v*

definition *P'* :: *'n* ⇒ *'a option* ⇒ *'a* ⇒ *bool* **where**
P' G uo v ≡ *case uo of None* ⇒ *False* | *Some u* ⇒ *P G u v*

definition *adjacency* :: *'n* ⇒ *'n* ⇒ (*'q*, *'m*) *state* ⇒ *'a* ⇒ *'a list* **where**
adjacency G1 G2 s v ≡
if P' G2 (*P-lookup* (*parent s*) *v*) *v then G.adjacency-list G1 v*
else G.adjacency-list G2 v

function (*domintros*) *alt-loop* :: *'n* ⇒ *'n* ⇒ *'a* ⇒ (*'q*, *'m*) *state* ⇒ (*'q*, *'m*) *state*
where
alt-loop G1 G2 src s =
(*if* ¬ *DONE s*
then let
u = *Q-head* (*queue s*);
q = *Q-tail* (*queue s*)
in alt-loop G1 G2 src (*List.fold* (*traverse-edge src u*) (*adjacency G1 G2 s u*)
(*s*(|*queue* := *q*|)))
else s)

definition *alt-bfs* :: *'n* ⇒ *'n* ⇒ *'a* ⇒ *'m* **where**
alt-bfs G1 G2 src ≡ *parent* (*alt-loop G1 G2 src* (*init src*))

abbreviation *alt-fold* :: *'n* ⇒ *'n* ⇒ *'a* ⇒ (*'q*, *'m*) *state* ⇒ (*'q*, *'m*) *state* **where**
alt-fold G1 G2 src s ≡
List.fold
(*traverse-edge src* (*Q-head* (*queue s*)))
(*adjacency G1 G2 s* (*Q-head* (*queue s*)))
(*s*(|*queue* := *Q-tail* (*queue s*)|))

end

4.2 Verification of the correctness of the algorithm

4.2.1 Assumptions on the input

Algorithm *alt-bfs.alt-bfs* expects two undirected graphs *G1* and *G2* such that *G1*'s and *G2*'s edges are disjoint and the union of *G1* and *G2* does not contain any odd-length cycles, as well a source vertex *src* in *G2* as input, and the correctness theorem will assume such an input. We remark that the assumption that *G1*'s and *G2*'s edges are disjoint is for the purpose of convenience. More specifically, when determining the vertices adjacent to a vertex *u*, with this assumption it is sufficient to check whether the edge from *u*'s parent to *u* is in *G1* or *G2*.

Let us now formally specify our assumptions on the input.

locale *alt-bfs-valid-input* = *alt-bfs* **where**
Map-update = *Map-update* **and**
P-update = *P-update* **and**
Q-snoc = *Q-snoc* **for**
Map-update :: *'a*::*linorder* ⇒ *'s* ⇒ *'n* ⇒ *'n* **and**
P-update :: *'a* ⇒ *'a* ⇒ *'m* ⇒ *'m* **and**
Q-snoc :: *'q* ⇒ *'a* ⇒ *'q* +
fixes *G1 G2* :: *'n*
fixes *src* :: *'a*
assumes *invar-G1*: *G.invar G1*
assumes *invar-G2*: *G.invar G2*
assumes *G1-symmetric*: *v* ∈ *set* (*G.adjacency-list G1 u*) ⟷ *u* ∈ *set* (*G.adjacency-list*

G1 v)
assumes *G2-symmetric*: $v \in set\ (G.adjacency\text{-}list\ G2\ u) \longleftrightarrow u \in set\ (G.adjacency\text{-}list\ G2\ v)$
assumes *E1-E2-disjoint*: $G.E\ G1 \cap G.E\ G2 = \{\}$
assumes *no-odd-cycle*: $\neg\ (\exists\ c.\ path\ (G.E\ (G.union\ G1\ G2))\ c \wedge odd\text{-}cycle\ c)$
assumes *src-mem-V2*: $src \in G.V\ G2$

context *alt-bfs-valid-input*
begin

sublocale *G1*: *symmetric-adjacency* **where** $G = G1$

sublocale *G2*: *symmetric-adjacency* **where** $G = G2$

end

abbreviation (**in** *alt-bfs-valid-input*) $G :: 'n$ **where**
$G \equiv G.union\ G1\ G2$

lemma (**in** *alt-bfs-valid-input*) *invar-G*:
shows $G.invar\ G$

context *alt-bfs-valid-input*
begin
sublocale *G*: *symmetric-adjacency* **where** $G = G$
end

4.2.2 Loop invariants

The loop invariants of *alt-bfs.alt-loop* are very similar to those of *bfs.loop*.

abbreviation (**in** *alt-bfs-valid-input*) $d :: 'm \Rightarrow 'a \Rightarrow nat$ **where**
$d\ m\ v \equiv path\text{-}length\ (rev\text{-}follow\ m\ v)$

abbreviation (**in** *alt-bfs-valid-input*) $P'' :: 'a\ set \Rightarrow bool$ **where**
$P''\ e \equiv e \in G.E\ G2$

abbreviation (**in** *alt-bfs-valid-input*) $alt :: ('q, 'm)\ state \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ **where**
$alt\ s\ u\ v \equiv P'\ G2\ (P\text{-}lookup\ (parent\ s)\ u)\ u \longleftrightarrow \neg\ P\ G2\ u\ v$

abbreviation (**in** *alt-bfs-valid-input*) $Q :: ('q, 'm)\ state \Rightarrow 'a \Rightarrow 'a\ set \Rightarrow bool$ **where**
$Q\ s\ v \equiv if\ P'\ G2\ (P\text{-}lookup\ (parent\ s)\ v)\ v\ then\ (Not \circ P'')\ else\ P''$

abbreviation (**in** *alt-bfs-valid-input*) $white :: ('q, 'm)\ state \Rightarrow 'a \Rightarrow bool$ **where**
$white\ s\ v \equiv \neg\ is\text{-}discovered\ src\ (parent\ s)\ v$

abbreviation (**in** *alt-bfs-valid-input*) $gray :: ('q, 'm)\ state \Rightarrow 'a \Rightarrow bool$ **where**
$gray\ s\ v \equiv is\text{-}discovered\ src\ (parent\ s)\ v \wedge v \in set\ (Q\text{-}list\ (queue\ s))$

abbreviation (**in** *alt-bfs-valid-input*) *black* :: ('*q*, '*m*) *state* ⇒ '*a* ⇒ *bool* **where**
black s v ≡ *is-discovered src* (*parent s*) *v* ∧ *v* ∉ *set* (*Q-list* (*queue s*))

locale *alt-bfs-invar* =
alt-bfs-valid-input **where** *P-update* = *P-update* **and** *Q-snoc* = *Q-snoc* +
parent P-lookup (*parent s*) **for**
P-update :: '*a*::*linorder* ⇒ '*a* ⇒ '*m* ⇒ '*m* **and**
Q-snoc :: '*q* ⇒ '*a* ⇒ '*q* **and**
s :: ('*q*, '*m*) *state* +
assumes *invar-queue*: *Q-invar* (*queue s*)
assumes *invar-parent*: *P-invar* (*parent s*)
assumes *parent-src*: *P-lookup* (*parent s*) *src* = *None*
assumes *parent-imp-alt*: *P-lookup* (*parent s*) *v* = *Some u* ⟹ *alt s u v*
assumes *parent-imp-edge*: *P-lookup* (*parent s*) *v* = *Some u* ⟹ {*u*, *v*} ∈ *G.E G*
assumes *not-white-if-mem-queue*: *v* ∈ *set* (*Q-list* (*queue s*)) ⟹ ¬ *white s v*
assumes *not-white-if-parent*: *P-lookup* (*parent s*) *v* = *Some u* ⟹ ¬ *white s u*
assumes *black-imp-adjacency-not-white*: ⟦ *alt s u v*; {*u*, *v*} ∈ *G.E G*; *black s u* ⟧
⟹ ¬ *white s v*
assumes *queue-sorted-wrt-d*: *sorted-wrt* (λ*u v*. *d* (*parent s*) *u* ≤ *d* (*parent s*) *v*)
(*Q-list* (*queue s*))
assumes *d-last-queue-le*:
¬ *Q-is-empty* (*queue s*) ⟹
d (*parent s*) (*last* (*Q-list* (*queue s*))) ≤ *d* (*parent s*) (*Q-head* (*queue s*)) + *1*
assumes *d-triangle-inequality*:
⟦ *alt-path* (*Q s u*) (*Not* ∘ *Q s u*) (*G.E G*) *p u v*; ¬ *white s u*; ¬ *white s v* ⟧ ⟹
d (*parent s*) *v* ≤ *d* (*parent s*) *u* + *path-length p*

Compared to *bfs.loop*, we need one additional invariant, *alt-bfs-invar.parent-imp-alt*, which captures alternation.

Let us verify that the loop invariants of *alt-bfs.alt-loop* are satisfied prior to the first iteration of the loop.

lemma (**in** *alt-bfs-valid-input*) *alt-bfs-invar-init*:
shows *alt-bfs-invar''* (*init src*)

Let us now verify that the loop invariants are maintained. For this, let us first look at how the different subroutines change the queue and parent map.

lemma (**in** *alt-bfs*) *list-queue-alt-fold-cong*:
assumes *G.invar G1*
assumes *G.invar G2*
assumes *Q-invar* (*queue s*)
assumes *P-invar* (*parent s*)
assumes ¬ *DONE s*
shows
Q-list (*queue* (*alt-fold G1 G2 src s*)) =
Q-list (*Q-tail* (*queue s*)) @

filter (*Not* ∘ *is-discovered src* (*parent s*)) (*adjacency G1 G2 s* (*Q-head* (*queue s*)))

lemma (**in** *alt-bfs*) *lookup-parent-alt-fold-cong*:
assumes *G.invar G1*
assumes *G.invar G2*
assumes *P-invar* (*parent s*)
shows
P-lookup (*parent* (*alt-fold G1 G2 src s*)) =
override-on
(*P-lookup* (*parent s*))
(λ-. *Some* (*Q-head* (*queue s*)))
(*set* (*filter* (*Not* ∘ *is-discovered src* (*parent s*)) (*adjacency G1 G2 s* (*Q-head* (*queue s*)))))

lemma (**in** *alt-bfs-invar*) *lookup-parent-alt-fold-cong*:
shows
P-lookup (*parent* (*alt-fold G1 G2 src s*)) =
override-on
(*P-lookup* (*parent s*))
(λ-. *Some* (*Q-head* (*queue s*)))
(*set* (*filter* (*Not* ∘ *is-discovered src* (*parent s*)) (*adjacency G1 G2 s* (*Q-head* (*queue s*)))))

lemma (**in** *alt-bfs*) *T-alt-fold-cong*:
assumes *G.invar G1*
assumes *G.invar G2*
assumes *P-invar* (*parent s*)
shows
T (*parent* (*alt-fold G1 G2 src s*)) =
T (*parent s*) ∪
{(*Q-head* (*queue s*), *v*) |*v*. *v* ∈ *set* (*adjacency G1 G2 s* (*Q-head* (*queue s*))) ∧ ¬ *is-discovered src* (*parent s*) *v*}

lemma (**in** *alt-bfs-invar*) *T-fold-cong*:
shows
T (*parent* (*alt-fold G1 G2 src s*)) =
T (*parent s*) ∪
{(*Q-head* (*queue s*), *v*) |*v*. *v* ∈ *set* (*adjacency G1 G2 s* (*Q-head* (*queue s*))) ∧ ¬ *is-discovered src* (*parent s*) *v*}

Next, we verify the maintenance of the loop invariants one by one.

locale *alt-bfs-invar-not-DONE* = *alt-bfs-invar* **where** *P-update* = *P-update* **and** *Q-snoc* = *Q-snoc* **for**
P-update :: *'a*::*linorder* ⇒ *'a* ⇒ *'m* ⇒ *'m* **and**
Q-snoc :: *'q* ⇒ *'a* ⇒ *'q* +
assumes *not-DONE*: ¬ *DONE s*

lemma (**in** *alt-bfs-invar-not-DONE*) *follow-invar-parent-alt-fold*:
shows *follow-invar* (*P-lookup* (*parent* (*alt-fold G1 G2 src s*)))

lemma (**in** *alt-bfs-invar-not-DONE*) *invar-queue-alt-fold*:
shows *Q-invar* (*queue* (*alt-fold G1 G2 src s*))

lemma (**in** *alt-bfs-invar*) *invar-parent-alt-fold*:
shows *P-invar* (*parent* (*alt-fold G1 G2 src s*))

lemma (**in** *alt-bfs-invar*) *parent-src-alt-fold*:
shows *P-lookup* (*parent* (*alt-fold G1 G2 src s*)) *src* = *None*

lemma (**in** *alt-bfs-invar-not-DONE*) *parent-imp-alt-alt-fold*:
assumes *P-lookup* (*parent* (*alt-fold G1 G2 src s*)) *v* = *Some u*
shows *alt* (*alt-fold G1 G2 src s*) *u v*

lemma (**in** *alt-bfs-invar-not-DONE*) *parent-imp-edge-alt-fold*:
assumes *P-lookup* (*parent* (*alt-fold G1 G2 src s*)) *v* = *Some u*
shows {*u*, *v*} ∈ *G.E G*

lemma (**in** *alt-bfs-invar-not-DONE*) *list-queue-alt-fold-cong*:
shows
Q-list (*queue* (*alt-fold G1 G2 src s*)) =
Q-list (*Q-tail* (*queue s*)) @
filter (*Not* ∘ *is-discovered src* (*parent s*)) (*adjacency G1 G2 s* (*Q-head* (*queue s*)))

lemma (**in** *alt-bfs-invar-not-DONE*) *black-imp-adjacency-not-white-alt-fold*:
assumes *alt* (*alt-fold G1 G2 src s*) *u v*
assumes {*u*, *v*} ∈ *G.E G*
assumes *black* (*alt-fold G1 G2 src s*) *u*
shows ¬ *white* (*alt-fold G1 G2 src s*) *v*

lemma (**in** *alt-bfs-invar-not-DONE*) *queue-sorted-wrt-d-alt-fold*:
shows *sorted-wrt* (λ*u v*. *d* (*parent* (*alt-fold G1 G2 src s*)) *u* ≤ *d* (*parent* (*alt-fold G1 G2 src s*)) *v*) (*Q-list* (*queue* (*alt-fold G1 G2 src s*)))

lemma (**in** *alt-bfs-invar-not-DONE*) *d-last-queue-le-alt-fold*:
assumes ¬ *Q-is-empty* (*queue* (*alt-fold G1 G2 src s*))
shows
d (*parent* (*alt-fold G1 G2 src s*)) (*last* (*Q-list* (*queue* (*alt-fold G1 G2 src s*)))) ≤
d (*parent* (*alt-fold G1 G2 src s*)) (*Q-head* (*queue* (*alt-fold G1 G2 src s*))) + *1*

lemma (**in** *alt-bfs-invar*) *alt-path-rev-follow-snocI*:
assumes *alt-path P′* (*Not* ∘ *P′*) (*G.E G*) (*rev-follow* (*parent s*) *u*) *src u*
assumes {*u*, *v*} ∈ *G.E G*
assumes *alt s u v*
assumes ¬ *white s u*
shows *alt-path P′* (*Not* ∘ *P′*) (*G.E G*) (*rev-follow* (*parent s*) *u* @ [*v*]) *src v*

lemma (**in** *alt-bfs-invar*) *alt-path-rev-follow-appendI*:
assumes *alt-path*: *alt-path* (*Q s u*) (*Not* ∘ *Q s u*) (*G.E G*) (*p* @ [*v*, *w*]) *u w*
assumes *not-white*: ¬ *white s u*
shows *alt-path P''* (*Not* ∘ *P''*) (*G.E G*) (*butlast* (*rev-follow* (*parent s*) *u*) @ *p* @ [*v*, *w*]) *src w*

lemma (**in** *alt-bfs-invar*) *alt-path-snoc-snocD*:
assumes *alt-path*: *alt-path P''* (*Not* ∘ *P''*) (*G.E G*) (*p* @ [*u*, *v*]) *src v*
assumes *not-white*: ¬ *white s u*
shows
{u, v} ∈ G.E G
alt s u v

lemma (**in** *alt-bfs-invar*) *white-imp-gray-ancestor*:
assumes *alt-path* (*Q s u*) (*Not* ∘ *Q s u*) (*G.E G*) *p u w*
assumes ¬ *white s u*
assumes *white s w*
obtains *v* **where**
v ∈ set p
gray s v

lemma (**in** *alt-bfs-valid-input*) *parent-imp-d*:
assumes *Parent-Relation.parent* (*P-lookup* (*parent s*))
assumes *P-lookup* (*parent s*) *v* = *Some u*
shows *d* (*parent s*) *v* = *d* (*parent s*) *u* + *1*

lemma (**in** *alt-bfs-invar-not-DONE*) *d-triangle-inequality-alt-fold*:
assumes *alt-path-p*: *alt-path* (*Q* (*alt-fold G1 G2 src s*) *u*) (*Not* ∘ *Q* (*alt-fold G1 G2 src s*) *u*) (*G.E G*) *p u v*
assumes *not-white-alt-fold-u*: ¬ *white* (*alt-fold G1 G2 src s*) *u*
assumes *not-white-alt-fold-v*: ¬ *white* (*alt-fold G1 G2 src s*) *v*
shows *d* (*parent* (*alt-fold G1 G2 src s*)) *v* ≤ *d* (*parent* (*alt-fold G1 G2 src s*)) *u* + *path-length p*

lemma (**in** *alt-bfs-invar-not-DONE*) *alt-bfs-invar-alt-fold*:
shows *alt-bfs-invar''* (*alt-fold G1 G2 src s*)

4.2.3 Termination

Before we can prove the correctness of loop *alt-bfs.alt-loop*, we need to prove that it terminates on appropriate inputs.

lemma (**in** *alt-bfs*) *alt-loop-dom*:
assumes *G.invar G1*
assumes *G.invar G2*
assumes *Q-invar* (*queue s*)
assumes *P-invar* (*parent s*)
assumes *set* (*Q-list* (*queue s*)) ⊆ *G.V* (*G.union G1 G2*)

assumes *P.dom* (*parent s*) ⊆ *G.V* (*G.union G1 G2*)
shows *alt-loop-dom* (*G1*, *G2*, *src*, *s*)

4.2.4 Correctness

We are now finally ready to prove the correctness of algorithm *alt-bfs.alt-bfs*.

locale *alt-bfs-invar-DONE* = *alt-bfs-invar* **where** *P-update* = *P-update* **and** *Q-snoc* = *Q-snoc* **for**
P-update :: *'a*::*linorder* ⇒ *'a* ⇒ *'m* ⇒ *'m* **and**
Q-snoc :: *'q* ⇒ *'a* ⇒ *'q* +
assumes *DONE*: *DONE s*

lemma (**in** *alt-bfs-invar-DONE*) *white-imp-not-alt-reachable*:
assumes *white s v*
shows ¬ *alt-reachable P''* (*Not* ∘ *P''*) (*G.E G*) *src v*

lemma (**in** *alt-bfs-invar-DONE*) *not-white-imp-d-le-alt-dist*:
assumes ¬ *white s v*
shows *d* (*parent s*) *v* ≤ *alt-dist P''* (*Not* ∘ *P''*) (*G.E G*) *src v*

lemma (**in** *alt-bfs-invar-DONE*) *not-white-imp-is-shortest-alt-path*:
assumes ¬ *white s v*
shows *is-shortest-alt-path P''* (*Not* ∘ *P''*) (*G.E G*) (*rev-follow* (*parent s*) *v*) *src v*

lemma (**in** *alt-bfs-valid-input*) *alt-loop-sound*:
assumes *alt-bfs-invar'' s*
assumes *is-discovered src* (*parent* (*alt-loop G1 G2 src s*)) *v*
shows *is-shortest-alt-path P''* (*Not* ∘ *P''*) (*G.E G*) (*rev-follow* (*parent* (*alt-loop G1 G2 src s*)) *v*) *src v*

lemma (**in** *alt-bfs-valid-input*) *alt-loop-complete*:
assumes *alt-bfs-invar'' s*
assumes ¬ *is-discovered src* (*parent* (*alt-loop G1 G2 src s*)) *v*
shows ¬ *alt-reachable P''* (*Not* ∘ *P''*) (*G.E G*) *src v*

abbreviation (**in** *alt-bfs*) *is-shortest-alt-path-Map* :: (*'a set* ⇒ *bool*) ⇒ *'n* ⇒ *'a* ⇒ *'m* ⇒ *bool* **where**
is-shortest-alt-path-Map Q G src m ≡
∀ *v*.
is-discovered src m v ⟶ *is-shortest-alt-path Q* (*Not* ∘ *Q*) (*G.E G*) (*rev-follow m v*) *src v* ∧
¬ *is-discovered src m v* ⟶ ¬ *alt-reachable Q* (*Not* ∘ *Q*) (*G.E G*) *src v*

lemma (**in** *alt-bfs-valid-input*) *correctness*:
assumes *alt-bfs-invar'' s*
shows *is-shortest-alt-path-Map P'' G src* (*parent* (*alt-loop G1 G2 src s*))

lemma (**in** *alt-bfs-valid-input*) *alt-bfs-correct*:

shows *is-shortest-alt-path-Map P'' G src* (*alt-bfs G1 G2 src*)

theorem (**in** *alt-bfs*) *alt-bfs-correct*:
assumes *alt-bfs-valid-input' G1 G2 src*
shows *is-shortest-alt-path-Map* (λ*e*. *e* ∈ *G.E G2*) (*G.union G1 G2*) *src* (*alt-bfs G1 G2 src*)

lemma (**in** *alt-bfs-invar*) *hd-rev-follow-eq-src*:
assumes ¬ *white s v*
shows *hd* (*rev-follow* (*parent s*) *v*) = *src*

end

4.3 Implementation of the algorithm

theory *Alternating-BFS-Partial*
imports
Alternating-BFS
begin

As is the case for BFS, we verified only partial termination and correctness of loop *alt-bfs.alt-loop*, since we assumed an appropriate input as specified via locale *alt-bfs-valid-input*. To obtain executable code, we make this explicit and use a partial function.

partial-function (**in** *alt-bfs*) (*tailrec*) *alt-loop-partial* **where**
alt-loop-partial G1 G2 src s =
(*if* ¬ *DONE s*
then let
u = *Q-head* (*queue s*);
q = *Q-tail* (*queue s*)
in alt-loop-partial G1 G2 src (*List.fold* (*traverse-edge src u*) (*adjacency G1 G2 s u*) (*s*(|*queue* := *q*|)))
else s)

definition (**in** *alt-bfs*) *alt-bfs-partial* :: ′*n* ⇒ ′*n* ⇒ ′*a* ⇒ ′*m* **where**
alt-bfs-partial G1 G2 src ≡ *parent* (*alt-loop-partial G1 G2 src* (*init src*))

lemma (**in** *alt-bfs-valid-input*) *alt-loop-partial-eq-alt-loop*:
assumes *alt-bfs-invar'' s*
shows *alt-loop-partial G1 G2 src s* = *alt-loop G1 G2 src s*

lemma (**in** *alt-bfs-valid-input*) *alt-bfs-partial-eq-alt-bfs*:
shows *alt-bfs-partial G1 G2 src* = *alt-bfs G1 G2 src*

theorem (**in** *alt-bfs-valid-input*) *alt-bfs-partial-correct*:
shows *is-shortest-alt-path-Map P'' G src* (*alt-bfs-partial G1 G2 src*)

corollary (**in** *alt-bfs*) *alt-bfs-partial-correct*:
assumes *alt-bfs-valid-input' G1 G2 src*

shows *is-shortest-alt-path-Map* (λ*e*. *e* ∈ *G.E G2*) (*G.union G1 G2*) *src* (*alt-bfs-partial G1 G2 src*)

end

4.4 Implementation of the algorithm

theory *BFS-Partial*
imports
BFS
begin

One point to note is that we verified only partial termination and correctness of loop *bfs.loop*, since we assumed an appropriate input as specified via locale *bfs-valid-input*. To obtain executable code, we make this explicit and use a partial function.

partial-function (**in** *bfs*) (*tailrec*) *loop-partial* **where**
loop-partial G src s =
(*if* ¬ *DONE s*
then let
u = *Q-head* (*queue s*);
q = *Q-tail* (*queue s*)
in loop-partial G src (*List.fold* (*traverse-edge src u*) (*G.adjacency-list G u*) (*s*(|*queue* := *q*|)))
else s)

definition (**in** *bfs*) *bfs-partial* :: ′*n* ⇒ ′*a* ⇒ ′*m* **where**
bfs-partial G src ≡ *parent* (*loop-partial G src* (*init src*))

lemma (**in** *bfs-valid-input*) *loop-partial-eq-loop*:
assumes *bfs-invar*′′ *s*
shows *loop-partial G src s* = *loop G src s*

lemma (**in** *bfs-valid-input*) *bfs-partial-eq-bfs*:
shows *bfs-partial G src* = *bfs G src*

theorem (**in** *bfs-valid-input*) *bfs-partial-correct*:
shows *is-shortest-dpath-Map G src* (*bfs-partial G src*)

corollary (**in** *bfs*) *bfs-partial-correct*:
assumes *bfs-valid-input*′ *G src*
shows *is-shortest-dpath-Map G src* (*bfs-partial G src*)

end
theory *Queue*
imports *Queue-Specs*
begin

On the low level, this interface is implemented using a pair of *list*s. Our

implementation is based on Okasaki, C. (1999). Purely functional data structures. Cambridge University Press.

type-synonym *'a queue* = *'a list* × *'a list*

definition *empty* :: *'a queue* **where**
empty = ([], [])

fun *is-empty* :: *'a queue* ⇒ *bool* **where**
is-empty (*f*, -) ⟷ *f* = []

fun *queue* :: *'a queue* ⇒ *'a queue* **where**
queue ([], *r*) = (*rev r*, []) |
queue (*f*, *r*) = (*f*, *r*)

fun *snoc* :: *'a queue* ⇒ *'a* ⇒ *'a queue* **where**
snoc (*f*, *r*) *x* = *queue* (*f*, *x* # *r*)

fun *head* :: *'a queue* ⇒ *'a* **where**
head (*x* # *f*, -) = *x*

fun *tail* :: *'a queue* ⇒ *'a queue* **where**
tail (*x* # *f*, *r*) = *queue* (*f*, *r*)

fun *invar* :: *'a queue* ⇒ *bool* **where**
invar ([], *r*) ⟷ *r* = [] |
invar (*f*, *r*) = *True*

fun *list* :: *'a queue* ⇒ *'a list* **where**
list (*f*, *r*) = *f* @ (*rev r*)

interpretation *Q*: *Queue* **where**
empty = *empty* **and**
is-empty = *is-empty* **and**
snoc = *snoc* **and**
head = *head* **and**
tail = *tail* **and**
invar = *invar* **and**
list = *list*

end

4.5 Low level

theory *Adjacency-Impl*
imports
Adjacency
Directed-Adjacency
Undirected-Adjacency
HOL−Data-Structures.RBT-Map

HOL−Data-Structures.RBT-Set2
begin

On the medium level of abstraction, we specified a graph via the interface *adjacency*. We now show that, on the low level, this interface can be implemented via red-black trees.

global-interpretation *G*: *adjacency* **where**
Map-empty = *empty* **and**
Map-update = *update* **and**
Map-delete = *RBT-Map.delete* **and**
Map-lookup = *lookup* **and**
Map-inorder = *inorder* **and**
Map-inv = *rbt* **and**
Set-empty = *empty* **and**
Set-insert = *insert* **and**
Set-delete = *delete* **and**
Set-isin = *isin* **and**
Set-inorder = *inorder* **and**
Set-inv = *rbt*
defines *invar* = *G.invar*
and *adjacency-list* = *G.adjacency-list*
and *insert* = *G.insert*
and *insert′* = *G.insert′*
and *insert-2* = *G.insert-2*
and *delete-2* = *G.delete-2*
and *union* = *G.union*
and *difference* = *G.difference*
and *dE* = *G.dE*
and *dV* = *G.dV*
and *E* = *G.E*
and *V* = *G.V*
and *insert-edge* = *G.insert-edge*

end
theory *BFS-Impl*
imports
BFS-Partial
HOL−Data-Structures.RBT-Set2
../Queue/Queue
../Graph/Adjacency/Adjacency-Impl
begin

We now show that our specification of BFS in locale *bfs* can be implemented via red-black trees.

global-interpretation *B*: *bfs* **where**
Map-empty = *empty* **and**
Map-update = *update* **and**
Map-delete = *RBT-Map.delete* **and**
Map-lookup = *lookup* **and**

Map-inorder = *inorder* **and**
Map-inv = *rbt* **and**
Set-empty = *empty* **and**
Set-insert = *RBT-Set.insert* **and**
Set-delete = *delete* **and**
Set-isin = *isin* **and**
Set-inorder = *inorder* **and**
Set-inv = *rbt* **and**
P-empty = *empty* **and**
P-update = *update* **and**
P-delete = *RBT-Map.delete* **and**
P-lookup = *lookup* **and**
P-invar = *M.invar* **and**
Q-empty = *Queue.empty* **and**
Q-is-empty = *is-empty* **and**
Q-snoc = *snoc* **and**
Q-head = *head* **and**
Q-tail = *tail* **and**
Q-invar = *Queue.invar* **and**
Q-list = *list*
defines *init* = *B.init*
and *DONE* = *B.DONE*
and *is-discovered* = *B.is-discovered*
and *discover* = *B.discover*
and *traverse-edge* = *B.traverse-edge*
and *loop-partial* = *B.loop-partial*
and *bfs-partial* = *B.bfs-partial*

declare *B.loop-partial.simps* [*code*]

end
theory *Alternating-BFS-Impl*
imports
Alternating-BFS-Partial
../BFS/BFS-Impl
begin

We now show that our specification of the modified BFS in locale *alt-bfs* can be implemented via red-black trees.

global-interpretation *A*: *alt-bfs* **where**
Map-empty = *empty* **and**
Map-update = *update* **and**
Map-delete = *RBT-Map.delete* **and**
Map-lookup = *lookup* **and**
Map-inorder = *inorder* **and**
Map-inv = *rbt* **and**
Set-empty = *empty* **and**
Set-insert = *RBT-Set.insert* **and**
Set-delete = *delete* **and**

Set-isin = *isin* **and**
Set-inorder = *inorder* **and**
Set-inv = *rbt* **and**
P-empty = *empty* **and**
P-update = *update* **and**
P-delete = *RBT-Map.delete* **and**
P-lookup = *lookup* **and**
P-invar = *M.invar* **and**
Q-empty = *Queue.empty* **and**
Q-is-empty = *is-empty* **and**
Q-snoc = *snoc* **and**
Q-head = *head* **and**
Q-tail = *tail* **and**
Q-invar = *Queue.invar* **and**
Q-list = *list*
defines $P = A.P$
and $P' = A.P'$
and *adjacency* = *A.adjacency*
and *alt-loop-partial* = *A.alt-loop-partial*
and *alt-bfs-partial* = *A.alt-bfs-partial*

declare *A.alt-loop-partial.simps* [*code*]

end
theory *Augmenting-Path*
imports
Alternating-Path
begin

A free vertex w.r.t. a matching M is a vertex not incident to any edge in M, and an augmenting path w.r.t. M is an alternating path w.r.t. M whose endpoints are distinct free vertices. Session `AGF` introduces the following two definitions: *augmenting-path ?M ?p* $\equiv$ *2* $\leq$ *length ?p* $\wedge$ *Berge.alt-path ?M ?p* $\wedge$ *hd ?p* $\notin$ *Vs ?M* $\wedge$ *last ?p* $\notin$ *Vs ?M*, and *augpath* $\equiv \lambda E\ M\ p.$ *path E p* $\wedge$ *distinct p* $\wedge$ *augmenting-path M p*. We extend their formalization and show that augmenting paths can be reversed.

lemma *augmenting-path-revI*:
assumes *augmenting-path M p*
shows *augmenting-path M* (*rev p*)

lemma *augpath-revI*:
assumes *augpath G M p*
shows *augpath G M* (*rev p*)

end
theory *Bipartite-Graph*
imports

Odd-Cycle
../Adaptors/Path-Adaptor
begin

A bipartite graph is an undirected graph G whose set of vertices *Vs* G can be partitioned into two sets L, R such that every edge in G has an endpoint in L and an endpoint in R.

locale *bipartite-graph* = *graph* G **for** G +
  fixes L R :: $'a$ *set*
  assumes *L-union-R-eq-Vs*: $L \cup R =$ *Vs* G
  assumes *L-R-disjoint*: $L \cap R = \{\}$
  assumes *endpoints*: $\{u, v\} \in G \Longrightarrow u \in L \longleftrightarrow v \in R$

Equivalently, a bipartite graph is an undirected graph whose set of vertices can be partitioned into two independent sets. We only show one implication.

lemma (**in** *bipartite-graph*) *L-independent*:
  shows $\forall\, u{\in}L.\ \forall\, v{\in}L.\ \{u, v\} \notin G$

lemma (**in** *bipartite-graph*) *R-independent*:
  shows $\forall\, u{\in}R.\ \forall\, v{\in}R.\ \{u, v\} \notin G$

lemma (**in** *bipartite-graph*) *no-loop*:
  shows $\{v, v\} \notin G$

Equivalently, a bipartite graph is an undirected graph that does not contain any odd-length cycles. Again, we only show one implication.

lemma (**in** *bipartite-graph*) *nth-mem-L-iff-even*:
  assumes *path* G p
  assumes *hd* $p \in L$
  assumes $i <$ *length* p
  shows $p\ !\ i \in L \longleftrightarrow$ *even* i

lemma (**in** *bipartite-graph*) *nth-mem-R-iff-even*:
  assumes *path* G p
  assumes *hd* $p \in R$
  assumes $i <$ *length* p
  shows $p\ !\ i \in R \longleftrightarrow$ *even* i

theorem (**in** *bipartite-graph*) *no-odd-cycle*:
  shows $\neg\ (\exists\, c.$ *path* G $c \wedge$ *odd-cycle* $c)$

end

5 Shortest augmenting path algorithm

This section specifies an algorithm that solves the maximum cardinality matching problem in bipartite graphs, and verifies its correctness.

The algorithm is based on Berge's theorem, which states that a matching M is maximum if and only if there is no augmenting path w.r.t. M. This immediately suggests the following algorithm for finding a maximum matching: repeatedly find an augmenting path and augment the matching until there are no augmenting paths. We claim that the algorithm specified below, in each iteration, finds not just any augmenting path but a shortest one. We do not verify this claim, however, as the distinction is not relevant for the correctness of the algorithm.

Hence, the algorithm takes the same general approach as the Edmonds-Karp algorithm, which solves the maximum flow problem, to which the maximum cardinality matching problem reduces.

theory *Edmonds-Karp*
  imports
    ../Alternating-BFS/Alternating-BFS
    ../Graph/Undirected-Graph/Augmenting-Path
    ../Graph/Undirected-Graph/Bipartite-Graph
begin

5.1 Specification of the algorithm

locale *edmonds-karp* =
  alt-bfs **where**
  Map-update = *Map-update* **and**
  P-update = *P-update* +
  M: *Map-by-Ordered* **where**
  empty = *M-empty* **and**
  update = *M-update* **and**
  delete = *M-delete* **and**
  lookup = *M-lookup* **and**
  inorder = *M-inorder* **and**
  inv = *M-inv* **for**
  Map-update :: *'a*::*linorder* ⇒ *'s* ⇒ *'n* ⇒ *'n* **and**
  P-update :: *'a* ⇒ *'a* ⇒ *'m* ⇒ *'m* **and**
  M-empty **and**
  M-update :: *'a* ⇒ *'a* ⇒ *'m* ⇒ *'m* **and**
  M-delete **and**
  M-lookup **and**
  M-inorder **and**
  M-inv
begin

definition *is-free-vertex* :: *'m* ⇒ *'a* ⇒ *bool* **where**
  is-free-vertex M v ≡ *M-lookup M v* = *None*

definition *free-vertices* :: *'s* ⇒ *'m* ⇒ *'a list* **where**
free-vertices V M ≡ *filter* (*is-free-vertex M*) (*Set-inorder V*)

To find an augmenting path, we use a modified BFS *local.alt-bfs*, which takes two graphs *G1*, *G2* as well as a source vertex *src* as input and outputs a parent relation such that any path from *src* induced by the parent relation is a shortest alternating path, that is, it alternates between edges in *G2* and *G1* and is shortest among all such paths.

Let $(L \cup R, G)$ be a bipartite graph and M be a matching in G. Recall that an augmenting path in G w.r.t. M is a path between two free vertices that alternates between edges not in M and edges in M. Since G is bipartite, any such path is between a free vertex in L and a free vertex in R (every augmenting path in a bipartite graph has odd length, and every path of odd length starting at a vertex in L ends at a vertex in R). This suggests to let *src* be a free vertex v in L, *G1* be the graph comprising all edges contained in M, and *G2* be the graph comprising all other edges.

As there may not be an augmenting path starting at v but one starting at another free vertex in L and *local.alt-bfs* takes only a single source vertex as input, we augment our input for *local.alt-bfs* as follows. Let G' be the graph comprising all edges contained in M and G'' be the graph comprising all other edges. We add a new vertex s to G' and connect it to all free vertices in L. Let p be a path in graph G, that is, not containing s. We then have that p is an augmenting path from a free vertex in L if and only if $s \# p$ is a path alternating between edges in G' and G'', ending at a free vertex in R.

Moreover, we add another new vertex t to graph G' and connect all free vertices in R to it. Again, let p be a path in graph G, that is, containing neither s nor t. We then have that p is an augmenting path from a free vertex in L if and only if $s \# p \mathbin{@} [t]$ is a path alternating between edges in G' and G''.

We now choose the input for *local.alt-bfs* as follows. We set *G1* to be G'', that is, the graph comprising all edges in graph G not in matching M, *G2* to be G', that is, the graph comprising all edges in M as well as two new vertices s, t such that s is connected to all free vertices in L and all free vertices in R are connected to t, and *src* to be s.

definition *G2-1* :: *'m* ⇒ *'n* **where**
G2-1 M ≡ *List.fold G.insert* (*M-inorder M*) *Map-empty*

Graph *G2-1* is the graph induced by the current matching M.

definition *G2-2* :: *'s* ⇒ *'a* ⇒ *'m* ⇒ *'n* **where**
G2-2 L s M ≡ *List.fold* (*G.insert-edge s*) (*free-vertices L M*) (*G2-1 M*)

Graph *G2-2* connects vertex s in graph *G2-1* to every free vertex in L.

definition *G2-3* :: *'s* ⇒ *'s* ⇒ *'a* ⇒ *'a* ⇒ *'m* ⇒ *'n* **where**
G2-3 L R s t M ≡ *List.fold* (*G.insert-edge t*) (*free-vertices R M*) (*G2-2 L s M*)

Graph *G2-3* connects every free vertex in R to vertex t in graph *G2-2*.

definition *G2* :: *'s* ⇒ *'s* ⇒ *'a* ⇒ *'a* ⇒ *'m* ⇒ *'n* **where**
G2 ≡ *G2-3*

definition *G1* :: *'n* ⇒ *'n* ⇒ *'n* **where**
G1 ≡ *G.difference*

As described above, the algorithm repeatedly finds an augmenting path and augments the matching until there are no augmenting paths. And there are no augmenting paths if

1. either side of the bipartite graph contains no free vertex, or
2. *local.alt-bfs* does not find an alternating path between vertices s and t.

definition *done-1* :: *'s* ⇒ *'s* ⇒ *'m* ⇒ *bool* **where**
done-1 L R M ≡ *free-vertices L M* = [] ∨ *free-vertices R M* = []

definition *done-2* :: *'a* ⇒ *'m* ⇒ *bool* **where**
done-2 t m ≡ *P-lookup m t* = *None*

fun *augment* :: *'m* ⇒ *'a path* ⇒ *'m* **where**
augment M [] = *M* |
augment M [*u*, *v*] = (*M-update v u* (*M-update u v M*)) |
augment M (*u* # *v* # *w* # *ws*) = *augment* (*M-update v u* (*M-update u v* (*M-delete w M*))) (*w* # *ws*)

function (*domintros*) *loop'* **where**
loop' G L R s t M =
(*if done-1 L R M then M*
else if done-2 t (*alt-bfs* (*G1 G* (*G2 L R s t M*)) (*G2 L R s t M*) *s*) *then M*
else loop' G L R s t (*augment M* (*butlast* (*tl* (*rev-follow* (*alt-bfs* (*G1 G* (*G2 L R s t M*)) (*G2 L R s t M*) *s*) *t*)))))

definition *edmonds-karp* :: *'n* ⇒ *'s* ⇒ *'s* ⇒ *'a* ⇒ *'a* ⇒ *'m* **where**
edmonds-karp G L R s t ≡ *loop' G L R s t M-empty*

abbreviation *m-tbd* :: *'n* ⇒ *'s* ⇒ *'s* ⇒ *'a* ⇒ *'a* ⇒ *'m* ⇒ *'m* **where**
m-tbd G L R s t M ≡ *let G2* = *G2 L R s t M in alt-bfs* (*G1 G G2*) *G2 s*

abbreviation *p-tbd* :: *'n* ⇒ *'s* ⇒ *'s* ⇒ *'a* ⇒ *'a* ⇒ *'m* ⇒ *'a path* **where**
p-tbd G L R s t M ≡ *butlast* (*tl* (*rev-follow* (*m-tbd G L R s t M*) *t*))

abbreviation *M-tbd* :: *'m* ⇒ *'a graph* **where**
M-tbd M ≡ {{*u*, *v*} |*u v*. *M-lookup M u* = *Some v*}

abbreviation *P-tbd* :: *'a path* ⇒ *'a graph* **where**
P-tbd p ≡ *set* (*edges-of-path p*)

abbreviation *is-symmetric-Map* :: *'m* ⇒ *bool* **where**
is-symmetric-Map M ≡ ∀ *u v*. *M-lookup M u* = *Some v* ⟷ *M-lookup M v* = *Some u*

end

5.2 Verification of the correctness of the algorithm

5.2.1 Assumptions on the input

Algorithm *edmonds-karp.edmonds-karp* expects an input $\langle G, L, R, s, t\rangle$ such that

- $(L \cup R, G)$ is a bipartite graph, and
- s and t are two new vertices, that is, vertices not in G,

and the correctness theorem will assume such an input. Let us formally specify these assumptions.

locale *edmonds-karp-valid-input* = *edmonds-karp* **where**
Map-update = *Map-update* **and**
P-update = *P-update* **and**
M-update = *M-update* **for**
Map-update :: *'a*::*linorder* ⇒ *'s* ⇒ *'n* ⇒ *'n* **and**
P-update :: *'a* ⇒ *'a* ⇒ *'m* ⇒ *'m* **and**
M-update :: *'a* ⇒ *'a* ⇒ *'m* ⇒ *'m* +
fixes *G* :: *'n*
fixes *L R* :: *'s*
fixes *s t* :: *'a*
assumes *symmetric-adjacency-G*: *G.symmetric-adjacency' G*
assumes *bipartite-graph*: *bipartite-graph* (*G.E G*) (*G.S.set L*) (*G.S.set R*)
assumes *s-not-mem-V*: *s* ∉ *G.V G*
assumes *t-not-mem-V*: *t* ∉ *G.V G*
assumes *s-neq-t*: *s* ≠ *t*

As is the case for locale *alt-bfs*, graph G is represented as an *adjacency*, that is, as a *Map-by-Ordered* mapping a vertex to its adjacency, which

is represented as a *Set-by-Ordered*. And sets L and R are represented as *Set-by-Ordered*s.

5.2.2 Loop invariants

Unfolding the definition of algorithm *edmonds-karp.edmonds-karp*, we see that recursive function *edmonds-karp.loop′* lies at the heart of the algorithm. It expects an input $\langle G, L, R, s, t, M\rangle$, which constitutes the program state, such that

- G, L, R, s, t satisfy the assumptions specified above, and
- M is a matching in G.

Let us now formally specify the assumptions on M. As M is the only state variable that is subject to change from one iteration to the next, these assumptions constitute the (non-trivial) loop invariants of *edmonds-karp.loop′*.

locale *edmonds-karp-invar* = *edmonds-karp-valid-input* **where**
  Map-update = *Map-update* **and**
  P-update = *P-update* **and**
  M-update = *M-update* **for**
  Map-update :: *'a*::*linorder* ⇒ *'s* ⇒ *'n* ⇒ *'n* **and**
  P-update :: *'a* ⇒ *'a* ⇒ *'m* ⇒ *'m* **and**
  M-update :: *'a* ⇒ *'a* ⇒ *'m* ⇒ *'m* +
  fixes *M* :: *'m*
  assumes *invar-M*: *M.invar M*
  assumes *is-symmetric-Map-M*: *is-symmetric-Map M*
  assumes *match-imp-edge*: *M-lookup M u* = *Some v* ⟹ {*u*, *v*} ∈ *G.E G*

lemma (**in** *edmonds-karp-invar*) *M-tbd-subset-E*:
  shows *M-tbd M* ⊆ *G.E G*

Matching M is represented as a *Map-by-Ordered* mapping a vertex to another vertex–its match.

lemma (**in** *edmonds-karp-invar*) *matching-M-tbd*:
  shows *matching* (*M-tbd M*)

lemma (**in** *edmonds-karp-invar*) *graph-matching-M-tbd*:
  shows *graph-matching* (*G.E G*) (*M-tbd M*)

To verify the correctness of loop *edmonds-karp.loop′*, we need to show that

1. the loop invariants are satisfied prior to the first iteration of the loop, and that

2. the loop invariants are maintained.

Let us start with the former, that is, let us prove that the empty matching satisfies the loop invariants.

lemma (**in** *edmonds-karp-valid-input*) *edmonds-karp-invar-empty*:
shows *edmonds-karp-invar''* *M-empty*

Let us now verify that the loop invariants are maintained, that is, if they hold at the start of an iteration of loop *edmonds-karp.loop'*, then they will also hold at the end. For this, we verify the correctness of the body of the loop, that is,

1. if there is an augmenting path, then the algorithm will find one, and
2. given an augmenting path, the algorithm correctly augments the current matching.

Let us start with the former.

locale *edmonds-karp-invar-not-done-1* = *edmonds-karp-invar* **where**
Map-update = *Map-update* **and**
P-update = *P-update* **and**
M-update = *M-update* **for**
Map-update :: *'a*::*linorder* ⇒ *'s* ⇒ *'n* ⇒ *'n* **and**
P-update :: *'a* ⇒ *'a* ⇒ *'m* ⇒ *'m* **and**
M-update :: *'a* ⇒ *'a* ⇒ *'m* ⇒ *'m* +
assumes *not-done-1*: ¬ *done-1 L R M*

locale *edmonds-karp-invar-not-done-2* = *edmonds-karp-invar-not-done-1* **where**
Map-update = *Map-update* **and**
P-update = *P-update* **and**
M-update = *M-update* **for**
Map-update :: *'a*::*linorder* ⇒ *'s* ⇒ *'n* ⇒ *'n* **and**
P-update :: *'a* ⇒ *'a* ⇒ *'m* ⇒ *'m* **and**
M-update :: *'a* ⇒ *'a* ⇒ *'m* ⇒ *'m* +
assumes *not-done-2*: ¬ *done-2 t* (*m-tbd G L R s t M*)

Assuming appropriate input for algorithm *alt-bfs.alt-bfs*, the statement follows from the correctness of *alt-bfs.alt-bfs*. Hence, we mainly have to show that our construction of *edmonds-karp.G1*, *edmonds-karp.G2* is correct and that it satisfies the input assumptions of *alt-bfs.alt-bfs*.

We first prove that graph *edmonds-karp.G2* comprises all edges in the current matching M as well as vertices s, t that are connected to all free vertices in L, R, respectively.

lemma (**in** *edmonds-karp*) *E2-1-cong*:

assumes *M.invar M*
shows *G.E (G2-1 M) = M-tbd M*

lemma (**in** *edmonds-karp*) *E2-2-cong*:
shows *G.E (G2-2 L s M) = G.E (G2-1 M) ∪ {{s, v} |v. v ∈ set (free-vertices L M)}*

lemma (**in** *edmonds-karp*) *E2-3-cong*:
shows *G.E (G2-3 L R s t M) = G.E (G2-2 L s M) ∪ {{t, v} |v. v ∈ set (free-vertices R M)}*

lemma (**in** *edmonds-karp*) *E2-cong*:
assumes *M.invar M*
shows
G.E (G2 L R s t M) =
M-tbd M ∪
{{s, v} |v. v ∈ set (free-vertices L M)} ∪
{{t, v} |v. v ∈ set (free-vertices R M)}

Next, we show that graph *edmonds-karp.G1* comprises all edges not in the current matching.

lemma (**in** *edmonds-karp*) *E1-cong*:
assumes *G.symmetric-adjacency′ G*
assumes *G.symmetric-adjacency′ G′*
shows *G.E (G1 G G′) = G.E G − G.E G′*

One point to note is that, given graphs *edmonds-karp.G1*, *edmonds-karp.G2*, algorithm *alt-bfs.alt-bfs* finds alternating paths in the union of *edmonds-karp.G1* and *edmonds-karp.G2*. We, on the other hand, are interested in paths in the input graph *G*, which, due to our augmentation by vertices *s* and *t*, is not equal to the union of *edmonds-karp.G1* and *edmonds-karp.G2*. So let us relate the union to the input graph.

lemma (**in** *edmonds-karp-invar*) *E-union-G1-G2-cong*:
shows
G.E (G.union (G1 G (G2 L R s t M)) (G2 L R s t M)) =
G.E G ∪ {{s, v} |v. v ∈ set (free-vertices L M)} ∪ {{t, v} |v. v ∈ set (free-vertices R M)}

lemma (**in** *edmonds-karp-invar-not-done-1*) *V-union-G1-G2-cong*:
shows *G.V (G.union (G1 G (G2 L R s t M)) (G2 L R s t M)) = G.V G ∪ {s} ∪ {t}*

We are now ready to show that ⟨*edmonds-karp.G1*, *edmonds-karp.G2*, *s*⟩ constitutes a valid input for algorithm *alt-bfs.alt-bfs*.

lemma (**in** *edmonds-karp-invar-not-done-1*) *alt-bfs-valid-input*:

shows *alt-bfs-valid-input'* (*G1 G* (*G2 L R s t M*)) (*G2 L R s t M*) *s*

Hence, by the soundness of algorithm *alt-bfs.alt-bfs*, any path from vertex *s* induced by the parent relation output by *alt-bfs.alt-bfs* is a shortest alternating path in the union of graphs *edmonds-karp.G1* and *edmonds-karp.G2*.

lemma (**in** *edmonds-karp-invar-not-done-1*) *is-shortest-alt-path-rev-follow*:
assumes *P-lookup* (*m-tbd G L R s t M*) *v* ≠ *None*
shows
is-shortest-alt-path
(λ*e*. *e* ∈ *G.E* (*G2 L R s t M*))
(*Not* ∘ (λ*e*. *e* ∈ *G.E* (*G2 L R s t M*)))
(*G.E* (*G.union* (*G1 G* (*G2 L R s t M*)) (*G2 L R s t M*)))
(*rev-follow* (*m-tbd G L R s t M*) *v*) *s v*

By our construction of graphs *edmonds-karp.G1* and *edmonds-karp.G2*, we can use this–as described above–to obtain an augmenting path in graph *G* w.r.t. the current matching *M*.

lemma (**in** *edmonds-karp-invar-not-done-2*) *augmenting-path-p-tbd*:
shows *augmenting-path* (*M-tbd M*) (*p-tbd G L R s t M*)

lemma (**in** *edmonds-karp-invar-not-done-2*) *augpath-p-tbd*:
shows *augpath* (*G.E G*) (*M-tbd M*) (*p-tbd G L R s t M*)

Having found an augmenting path *P* in graph *G* w.r.t. the current matching *M*, we now verify that the algorithm correctly augments *M* by *P*, that is, we show that function *edmonds-karp.augment* implements the symmetric difference *M* ⊕ *P*.

lemma (**in** *edmonds-karp*) *M-tbd-augment-cong*:
assumes *M.invar M*
assumes *is-symmetric-Map M*
assumes *augmenting-path* (*M-tbd M*) *p*
assumes *distinct p*
assumes *even* (*length p*)
shows *M-tbd* (*augment M p*) = *M-tbd M* ⊕ *P-tbd p*

Having verified the correctness of the body of loop *edmonds-karp.loop'*, we are now finally able to show that the loop invariants are maintained.

lemma (**in** *edmonds-karp-invar-not-done-2*) *edmonds-karp-invar-augment*:
shows *edmonds-karp-invar''* (*augment M* (*p-tbd G L R s t M*))

5.2.3 Termination

Before we can prove the correctness of loop *edmonds-karp.loop′*, we need to prove that it terminates on appropriate inputs. For this, we show that the size of matching M increases from one iteration to the next.

```
lemma (in edmonds-karp-valid-input) loop′-dom:
  assumes edmonds-karp-invar′′ M
  shows loop′-dom (G, L, R, s, t, M)
proof (induct card (G.E G) − card (M-tbd M) arbitrary: M rule: less-induct)
  case less
  let ?G2 = G2 L R s t M
  let ?G1 = G1 G ?G2
  let ?m = alt-bfs ?G1 ?G2 s
  have m: ?m = m-tbd G L R s t M
    by metis
  show ?case
  proof (cases done-1 L R M)
    case True
    thus ?thesis
      by (blast intro: loop′.domintros)
  next
    case not-done-1: False
    show ?thesis
    proof (cases done-2 t ?m)
      case True
      thus ?thesis
        by (blast intro: loop′.domintros)
    next
      case False
      let ?p = butlast (tl (rev-follow ?m t))
      have p: ?p = p-tbd G L R s t M
        by metis
      let ?M = augment M ?p
      have edmonds-karp-invar-not-done-2: edmonds-karp-invar-not-done-2′′ M
        using less.prems not-done-1 False
        unfolding m
        by (intro edmonds-karp-invar-not-done-2I-2)
      hence augpath-p: augpath (G.E G) (M-tbd M) ?p
        unfolding m
        by (intro edmonds-karp-invar-not-done-2.augpath-p-tbd)
      show ?thesis
      proof (rule loop′.domintros, rule less.hyps, goal-cases)
        case 1
        have card (M-tbd M) < card (M-tbd ?M)
        moreover have card (M-tbd ?M) ≤ card (G.E G)
        ultimately show ?case
          by linarith
      next
        case 2
```

```
      thus ?case
        unfolding p
        using edmonds-karp-invar-not-done-2
        by (intro edmonds-karp-invar-not-done-2.edmonds-karp-invar-augment)
    qed
  qed
 qed
qed
```

5.2.4 Correctness

We are now finally ready to prove the correctness of algorithm *edmonds-karp.edmonds-karp*. We still need to show that if the algorithm doesn't find an augmenting path, then the current matching M is already maximum.

```
abbreviation is-maximum-matching :: 'a graph ⇒ 'a graph ⇒ bool where
  is-maximum-matching G M ≡ graph-matching G M ∧ (∀ M'. graph-matching G
M' ⟶ card M' ≤ card M)

locale edmonds-karp-invar-done-1 = edmonds-karp-invar where
  Map-update = Map-update and
  P-update = P-update and
  M-update = M-update for
  Map-update :: 'a::linorder ⇒ 's ⇒ 'n ⇒ 'n and
  P-update :: 'a ⇒ 'a ⇒ 'm ⇒ 'm and
  M-update :: 'a ⇒ 'a ⇒ 'm ⇒ 'm +
  assumes done-1: done-1 L R M

lemma (in edmonds-karp-invar-done-1) is-maximum-matching-M-tbd:
  shows is-maximum-matching (G.E G) (M-tbd M)

locale edmonds-karp-invar-done-2 = edmonds-karp-invar-not-done-1 where
  Map-update = Map-update and
  P-update = P-update and
  M-update = M-update for
  Map-update :: 'a::linorder ⇒ 's ⇒ 'n ⇒ 'n and
  P-update :: 'a ⇒ 'a ⇒ 'm ⇒ 'm and
  M-update :: 'a ⇒ 'a ⇒ 'm ⇒ 'm +
  assumes done-2: done-2 t (m-tbd G L R s t M)

lemma (in edmonds-karp-invar-done-2) is-maximum-matching-M-tbd:
  shows is-maximum-matching (G.E G) (M-tbd M)
```

Otherwise, we augment matching M by the augmenting path found as verified above, and it follows by induction (via the induction rule given by function *edmonds-karp.loop'*) that the algorithm outputs a maximum matching.

```
lemma (in edmonds-karp-valid-input) is-maximum-matching-M-tbd-loop':
```

```
  assumes edmonds-karp-invar'' M
  shows is-maximum-matching (G.E G) (M-tbd (loop' G L R s t M))
proof (induct rule: edmonds-karp-induct[OF assms])
  case (1 G L R s t M)
  show ?case
  proof (cases done-1 L R M)
    case True
    with 1.prems
    have edmonds-karp-invar-done-1' G L R s t M
      by (intro edmonds-karp-invar-done-1I)
    thus ?thesis
      by
        (intro edmonds-karp-invar-done-1.is-maximum-matching-M-tbd)
        (simp add: edmonds-karp-invar-done-1.loop'-psimps)
  next
    case not-done-1: False
    show ?thesis
    proof (cases done-2 t (m-tbd G L R s t M))
      case True
      with 1.prems not-done-1
      have edmonds-karp-invar-done-2' G L R s t M
        by (intro edmonds-karp-invar-done-2I-2)
      thus ?thesis
        by
          (intro edmonds-karp-invar-done-2.is-maximum-matching-M-tbd)
          (simp add: edmonds-karp-invar-done-2.loop'-psimps)
    next
      case False
      with 1.prems not-done-1
      have edmonds-karp-invar-not-done-2' G L R s t M
        by (intro edmonds-karp-invar-not-done-2I-2)
      thus ?thesis
        using not-done-1 False
        by
          (auto
            simp add: edmonds-karp-invar-not-done-2.loop'-psimps
            dest: 1.hyps
            intro: edmonds-karp-invar-not-done-2.edmonds-karp-invar-augment)
    qed
  qed
qed
```

We finally have everything to state and prove the correctness theorem for algorithm *edmonds-karp.edmonds-karp*.

```
lemma (in edmonds-karp-valid-input) edmonds-karp-correct:
  shows is-maximum-matching (G.E G) (M-tbd (edmonds-karp G L R s t))

theorem (in edmonds-karp) edmonds-karp-correct:
  assumes edmonds-karp-valid-input' G L R s t
```

shows *is-maximum-matching* (*G.E G*) (*M-tbd* (*edmonds-karp G L R s t*))

end

5.3 Implementation of the algorithm

theory *Edmonds-Karp-Partial*
imports
../Alternating-BFS/Alternating-BFS-Partial
Edmonds-Karp
../BFS/Parent-Relation-Partial
begin

One point to note is that we verified only partial termination and correctness of loop *edmonds-karp.loop′*, since we assumed an appropriate input as specified via locale *edmonds-karp-valid-input*. To obtain executable code, we make this explicit and use a partial function.

partial-function (**in** *edmonds-karp*) (*tailrec*) *loop′-partial* **where**
loop′-partial G L R s t M =
(*if done-1 L R M then M*
else if done-2 t (*alt-bfs-partial* (*G1 G* (*G2 L R s t M*)) (*G2 L R s t M*) *s*) *then*
M
else loop′-partial G L R s t (*augment M* (*butlast* (*tl* (*Parent-Relation-Partial.rev-follow*
(*P-lookup* (*alt-bfs-partial* (*G1 G* (*G2 L R s t M*)) (*G2 L R s t M*) *s*)) *t*)))))

definition (**in** *edmonds-karp*) *edmonds-karp-partial* **where**
edmonds-karp-partial G L R s t ≡ *loop′-partial G L R s t M-empty*

lemma (**in** *edmonds-karp-valid-input*) *loop′-partial-eq-loop′*:
assumes *edmonds-karp-invar′′ M*
shows *loop′-partial G L R s t M* = *loop′ G L R s t M*

lemma (**in** *edmonds-karp-valid-input*) *edmonds-karp-partial-eq-edmonds-karp*:
shows *edmonds-karp-partial G L R s t* = *edmonds-karp G L R s t*

lemma (**in** *edmonds-karp-valid-input*) *edmonds-karp-partial-correct*:
shows *is-maximum-matching* (*G.E G*) (*M-tbd* (*edmonds-karp-partial G L R s t*))

theorem (**in** *edmonds-karp*) *edmonds-karp-partial-correct*:
assumes *edmonds-karp-valid-input′ G L R s t*
shows *is-maximum-matching* (*G.E G*) (*M-tbd* (*edmonds-karp-partial G L R s t*))

end
theory *Edmonds-Karp-Impl*

imports
../Alternating-BFS/Alternating-BFS-Impl
Edmonds-Karp-Partial
begin

We now show that our specification of the Edmonds-Karp algorithm in locale *edmonds-karp* can be implemented via red-black trees.

global-interpretation *E*: *edmonds-karp* **where**
Map-empty = *empty* **and**
Map-update = *update* **and**
Map-delete = *RBT-Map.delete* **and**
Map-lookup = *lookup* **and**
Map-inorder = *inorder* **and**
Map-inv = *rbt* **and**
Set-empty = *empty* **and**
Set-insert = *RBT-Set.insert* **and**
Set-delete = *delete* **and**
Set-isin = *isin* **and**
Set-inorder = *inorder* **and**
Set-inv = *rbt* **and**
P-empty = *empty* **and**
P-update = *update* **and**
P-delete = *RBT-Map.delete* **and**
P-lookup = *lookup* **and**
P-invar = *M.invar* **and**
Q-empty = *Queue.empty* **and**
Q-is-empty = *is-empty* **and**
Q-snoc = *snoc* **and**
Q-head = *head* **and**
Q-tail = *tail* **and**
Q-invar = *Queue.invar* **and**
Q-list = *list* **and**
M-empty = *empty* **and**
M-update = *update* **and**
M-delete = *RBT-Map.delete* **and**
M-lookup = *lookup* **and**
M-inorder = *inorder* **and**
M-inv = *rbt*
defines *is-free-vertex* = *E.is-free-vertex*
and *free-vertices* = *E.free-vertices*
and *G2-1* = *E.G2-1*
and *G2-2* = *E.G2-2*
and *G2-3* = *E.G2-3*
and *G2* = *E.G2*
and *G1* = *E.G1*
and *done-1* = *E.done-1*
and *done-2* = *E.done-2*
and *augment* = *E.augment*
and *loop′-partial* = *E.loop′-partial*

and *edmonds-karp-partial* = *E.edmonds-karp-partial*

declare *rev-follow-partial.simps* [*code*]
declare *E.loop'-partial.simps* [*code*]

end

5.3.1 Undirected graphs

theory *Undirected-Graph*
imports
Augmenting-Path
Bipartite-Graph
Shortest-Alternating-Path
begin

end

6 Graph

theory *Graph*
imports
Adjacency/Adjacency
Adjacency/Adjacency-Impl
Directed-Graph/Directed-Graph
Undirected-Graph/Undirected-Graph
begin

This section considers graphs from three levels of abstraction. On the high level, a graph is a set of edges (*graph* for undirected graphs, and *dgraph* for directed graphs). On the medium level, a graph is specified via the interface *adjacency*. On the low level, this interface is then implemented via red-black trees.

6.1 High level

For the high level of abstraction, we extend the archive of graph formalizations `AGF`, which formalizes both directed (*dgraph*) and undirected (*graph*) graphs as sets of edges. The set of vertices of a graph is then defined as the union of all endpoints of all edges in the graph ($dVs\ ?dG \equiv \bigcup \{\{v1, v2\} | v1\ v2.\ v1 \rightarrow_{?dG} v2\}$ for directed graphs, and $Vs\ ?E \equiv \bigcup\ ?E$ for undirected graphs). Let us first look at directed graphs.

end