thys

mitjakrebs

June 23, 2022

Contents

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 few 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*: $(u, v) \in G \Longrightarrow u \neq v$

locale *symmetric-dgraph* = *dgraph* G **for** G +
assumes *symmetric*: $(u, v) \in G \longleftrightarrow (v, u) \in G$

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* ⇒ *nat* **where**
dpath-length p ≡ *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*) ≤ *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* ⟷ (∃ *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*: $\forall\, e{\in}G.\ \exists\, u\ v.\ e = \{u,\ v\}$

lemma (**in** *graph*) *graph-subset*:
assumes $G' \subseteq G$

shows *graph* G'

lemma *graphs-eqI*:
assumes *graph G1*
assumes *graph G2*
assumes $\bigwedge u\ v.\ \{u,\ v\} \in G1 \longleftrightarrow \{u,\ v\} \in G2$
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**
$dEs \equiv \{(u,\ v).\ \{u,\ v\} \in G\}$

lemma (**in** *graph*) *dEs-symmetric*:
shows $(u,\ v) \in dEs \longleftrightarrow (v,\ u) \in dEs$

context *finite-graph*
begin
sublocale *F*: *finite-dgraph dEs*
end

end
theory *Misc-Ext*
imports
HOL−Library.Extended-Nat
HOL−Data-Structures.List-Ins-Del
HOL−Data-Structures.Set-Specs
begin

1 *enat*

lemma *enat-add-strict-right-mono*:
  fixes $a\ b\ c :: enat$
  assumes $a < b$
  assumes $c \neq \infty$
  shows $a + c < b + c$

lemma *enat-add-strict-left-mono*:
  fixes $a\ b\ c :: enat$
  assumes $b < c$
  assumes $a \neq \infty$
  shows $a + b < a + c$

lemma *INF-in-image*:
  fixes $f :: {}'a \Rightarrow enat$
  assumes *S-finite*: *finite S*
  assumes *S-non-empty*: $S \neq \{\}$
  shows $Inf\ (f\ `\ S) \in f\ `\ S$

2 *list*

2.1 *length*

lemma *length-ge-2D*:
  assumes $2 \leq length\ l$
  shows
    $l \neq []$
    $tl\ l \neq []$
    $butlast\ l \neq []$

lemma *length-ge-2E*:
  assumes $2 \leq length\ l$
  obtains $x\ xs\ y$ **where**
    $l = x\ \#\ xs\ @\ [y]$

lemma *length-butlast-tl*:
  assumes $2 \leq length\ l$
  shows $length\ (butlast\ (tl\ l)) = length\ l - 2$

2.2 *distinct*

lemma *distinct-ins-listD*:
  assumes $distinct\ (ins\text{-}list\ x\ xs)$
  shows $distinct\ xs$

lemma *distinct-ins-listI*:

assumes *Sorted-Less.sorted xs*
assumes *distinct xs*
shows *distinct* (*ins-list x xs*)

lemma *distinct-ins-list-cong*:
assumes *Sorted-Less.sorted xs*
shows *distinct* (*ins-list x xs*) = *distinct xs*

lemma *distinct-imp-hd-not-mem-set-tl*:
assumes $l \neq []$
assumes *distinct l*
shows *hd l* $\notin$ *set* (*tl l*)

lemma *distinct-imp-last-not-mem-set-butlast*:
assumes $l \neq []$
assumes *distinct l*
shows *last l* $\notin$ *set* (*butlast l*)

2.3 *sorted-wrt*

lemma *sorted-wrt-imp-hd*:
assumes *l-sorted-wrt*: *sorted-wrt P l*
assumes *x-mem-l*: $x \in$ *set l*
assumes *x-not-hd*: $x \neq$ *hd l*
shows *P* (*hd l*) *x*

lemma *sorted-wrt-imp-last-aux*:
assumes *x-mem-l*: $x \in$ *set l*
assumes *x-neq-last*: $x \neq$ *last l*
obtains *i* **where**
$i <$ *length l* $-$ *1*
$x = l\ !\ i$

lemma *sorted-wrt-imp-last*:
assumes *l-sorted-wrt*: *sorted-wrt P l*
assumes *x-mem-l*: $x \in$ *set l*
assumes *x-neq-last*: $x \neq$ *last l*
shows *P x* (*last l*)

lemma *sorted-wrt-if*:
assumes $\bigwedge x\ y.\ x \in$ *set l* $\Longrightarrow y \in$ *set l* $\Longrightarrow P\ x\ y$
shows *sorted-wrt P l*

2.4

lemma *list-split-tbd*:
assumes $l \neq []$
assumes *hd l* $\neq$ *last l*

obtains l' **where**
$l = hd\ l\ \#\ l'\ @\ [last\ l]$

lemma *butlast-tl-conv*:
assumes $l1 \neq []$
assumes $l2 \neq []$
assumes $last\ l1 = hd\ l2$
shows $butlast\ l1\ @\ l2 = l1\ @\ tl\ l2$

3 *Set-by-Ordered*

lemma (**in** *Set-by-Ordered*) *inorder-distinct*:
assumes *invar s*
shows *distinct* (*inorder s*)

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 $set\ (edges\text{-}of\text{-}path\ p) \subseteq G$
assumes $set\ p \subseteq Vs\ G$
shows *path G p*

lemma *walk-betw-induct* [*consumes 1*]:
assumes *walk-betw G u p v*
assumes $\bigwedge v.\ P\ [v]$
assumes $\bigwedge u\ v\ vs.\ P\ (v\ \#\ vs) \Longrightarrow P\ (u\ \#\ v\ \#\ vs)$
shows *P p*

lemma *walk-betw-induct-2* [*consumes 1*]:
assumes *walk-betw G u p v*
assumes $P\ [v]$
assumes $\bigwedge u.\ P\ [u,\ v]$
assumes $\bigwedge u\ x\ xs.\ P\ (x\ \#\ xs\ @\ [v]) \Longrightarrow P\ (u\ \#\ x\ \#\ xs\ @\ [v])$
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 $\{u, v\} \in G$
assumes $\{x, y\} \in G$
shows
walk-betw G u (u # p @ [y]) y
$\{u, v\} \in$ *set (edges-of-path (u # p @ [y]))*
$\{x, y\} \in$ *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

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 simple path (*distinct-path*) is a path in which all vertices are distinct.

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* $\Rightarrow$ $'a \Rightarrow$ $'a \Rightarrow$ *bool* **where**
  reachable G u v $\equiv \exists\, p.$ *walk-betw G u p v*

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

abbreviation *path-length* :: $'a$ *path* $\Rightarrow$ *nat* **where**
  path-length p $\equiv$ *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* $\longleftrightarrow$ *dpath dEs p*

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

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

end
theory *Odd-Cycle*
  imports
    Path
begin

We redefine odd 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 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**
  $odd\text{-}cycle\ p \equiv odd\ (path\text{-}length\ p) \wedge hd\ p = last\ p$

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

We generalize this definition to arbitrary predicates P, Q: *alt-list*. 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*.

definition $alt\text{-}path :: ('a\ set \Rightarrow bool) \Rightarrow ('a\ set \Rightarrow bool) \Rightarrow 'a\ graph \Rightarrow 'a\ path \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ **where**
  $alt\text{-}path\ P\ Q\ G\ p\ u\ v \equiv alt\text{-}list\ P\ Q\ (edges\text{-}of\text{-}path\ p) \wedge walk\text{-}betw\ G\ u\ p\ v$

lemma *two-alt-pathsD*:
  assumes $alt\text{-}path\ P\ Q\ G\ p\ u\ v$
  assumes $alt\text{-}path\ P\ Q\ G\ q\ u\ v$
  assumes $\neg\ (\exists\, c.\ path\ G\ c \wedge odd\text{-}cycle\ c)$
  shows $odd\ (path\text{-}length\ p) = odd\ (path\text{-}length\ q)$

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

We can concatenate alternating paths.

lemma *alt-path-ConsI*:
  assumes $alt\text{-}path\ P\ Q\ G\ p\ v\ w$
  assumes $\{u,\ v\} \in G$
  assumes $Q\ \{u,\ v\}$
  shows $alt\text{-}path\ Q\ P\ G\ (u\ \#\ p)\ u\ w$

lemma *alt-path-snocI*:
  assumes *alt-path*: *alt-path* P $(Not \circ P)$ G $(vs$ @ $[v'', v'])$ u v'
  assumes *alt*: P $\{v'', v'\} = (Not \circ P)$ $\{v', v\}$
  assumes *edge*: $\{v', v\} \in G$
  shows *alt-path* P $(Not \circ P)$ G $(vs$ @ $[v'', v', v])$ u v

lemma *alt-path-snoc-oddI*:
  assumes *alt-path* P Q G p u v
  assumes *odd* (*path-length* p)
  assumes $\{v, w\} \in G$
  assumes Q $\{v, w\}$
  shows *alt-path* P Q G $(p$ @ $[w])$ u w

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 $p \neq []$
  shows *alt-path* P Q G p u (*last* p)

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

lemma *alt-path-suf-2*:
  assumes *alt-path* P $(Not \circ P)$ G $(p$ @ $[v, v']$ @ $q)$ u w
  assumes $\neg$ P $\{v, v'\}$
  shows *alt-path* $(Not \circ P)$ 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 $\neg$ $(\exists\, c.$ *path* G c $\wedge$ *odd-cycle* $c)$
  shows *alt-path* P Q G $(p'$ @ $q)$ u w

definition *distinct-alt-path* :: $('a\ set \Rightarrow bool) \Rightarrow ('a\ set \Rightarrow bool) \Rightarrow 'a\ graph \Rightarrow 'a\ path \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ **where**
  distinct-alt-path P Q G p u v $\equiv$ *alt-path* P Q G p u v $\wedge$ *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 ¬ (∃ *c*. *path G c* ∧ *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 ¬ (∃ *c*. *path G c* ∧ *odd-cycle c*)
shows *is-shortest-alt-path P Q G* (*v* # *q*) *v w* ∨ *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 ¬ (∃ *c*. *path G c* ∧ *odd-cycle c*)
shows *alt-dist P Q G u w* = *alt-dist P Q G u v* + *1*

end

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

4 *Map*

definition (**in** *Map*) *dom* :: $'m \Rightarrow 'a$ *set* **where**
$dom\ m \equiv \{a.\ lookup\ m\ a \neq None\}$

lemma (**in** *Map*) *mem-dom-iff*:
shows $a \in dom\ m \longleftrightarrow lookup\ m\ a \neq None$

definition (**in** *Map*) *ran* :: $'m \Rightarrow 'b$ *set* **where**
$ran\ m \equiv \{b.\ \exists\, a.\ lookup\ m\ a = Some\ b\}$

lemma (**in** *Map*) *finite-dom-imp-finite-ran*:
assumes *finite* (*dom m*)
shows *finite* (*ran m*)

5 *Map-by-Ordered*

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

lemma *sorted-imp-distinct*:
assumes *sorted l*
shows *distinct l*

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

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

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*) *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*)

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

lemma (**in** *Map-by-Ordered*) *set-filter-inorder-cong*:
 assumes *invar m*
 shows *set* (*filter* ($\lambda p.$ *fst p* = *a*) (*inorder m*)) = (*case lookup m a of None* ⇒ {} | *Some b* ⇒ {(*a*, *b*)})

lemma *sorted1D*:
 assumes *sorted* (*a* # *map fst ps*)
 shows (*a*, *y*) ∉ *set ps*

lemma *sorted1D-2*:
 assumes *sorted* (*a* # *map fst ps*)
 assumes $x < a$
 shows (*x*, *y*) ∉ *set ps*

lemma *set-del-list-cong*:
 assumes *sorted1 l*
 shows *set* (*del-list x l*) = *set l* − *set* (*filter* ($\lambda p.$ *fst p* = *x*) *l*)

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 *set-upd-list-cong*:
 assumes *sorted1 l*
 shows *set* (*upd-list x y l*) = *set l* − *set* (*filter* ($\lambda p.$ *fst p* = *x*) *l*) ∪ {(*x*, *y*)}

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*)}

end
theory *Orderings-Ext*
 imports *Main*
begin

instantiation *prod* :: (*linorder*, *linorder*) *linorder*
begin

abbreviation *less-prod′* **where**
  less-prod′ p1 p2 ≡
   case p1 of (*a1*::′*a*::*linorder*, *b1*::′*b*::*linorder*) ⇒
     case p2 of (*a2*::′*a*::*linorder*, *b2*::′*b*::*linorder*) ⇒
       if (*a1* < *a2*) ∨ (*a1* = *a2* ∧ *b1* < *b2*) *then True else False*

definition *less-prod* **where**
  less-prod ≡ *less-prod′*

definition *eq-prod* **where**
  eq-prod p1 p2 ≡
   case p1 of (*a1*::′*a*::*linorder*, *b1*::′*b*::*linorder*) ⇒
     case p2 of (*a2*::′*a*::*linorder*, *b2*::′*b*::*linorder*) ⇒
       if (*a1* = *a2*) ∧ (*b1* = *b2*) *then True else False*

definition *less-eq-prod* **where**
  less-eq-prod p1 p2 ≡ *less-prod′ p1 p2* ∨ *eq-prod p1 p2*

instance

end

end

5.1 Medium level

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.

5.1.1 Adjacency structure

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

`Ports`

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**

$delete = Map\text{-}delete$ **and**
$lookup = Map\text{-}lookup$ **and**
$inorder = Map\text{-}inorder$ **and**
$inv = Map\text{-}inv$ +
S: *Set-by-Ordered* **where**
$empty = Set\text{-}empty$ **and**
$insert = Set\text{-}insert$ **and**
$delete = Set\text{-}delete$ **and**
$isin = Set\text{-}isin$ **and**
$inorder = Set\text{-}inorder$ **and**
$inv = Set\text{-}inv$ **for**
Map-empty **and**
$Map\text{-}update :: 'a::linorder \Rightarrow 's \Rightarrow 'm \Rightarrow 'm$ **and**
Map-delete **and**
Map-lookup **and**
Map-inorder **and**
Map-inv **and**
Set-empty **and**
$Set\text{-}insert :: 'a \Rightarrow 's \Rightarrow 's$ **and**
Set-delete **and**
Set-isin **and**
Set-inorder **and**
Set-inv

definition (**in** *adjacency*) $invar :: 'm \Rightarrow bool$ **where**
$invar\ G \equiv M.invar\ G \wedge Ball\ (M.ran\ G)\ S.invar$

definition (**in** *adjacency*) $adjacency\text{-}list :: 'm \Rightarrow 'a \Rightarrow 'a\ list$ **where**
$adjacency\text{-}list\ G\ u \equiv case\ Map\text{-}lookup\ G\ u\ of\ None \Rightarrow []\ |\ Some\ s \Rightarrow Set\text{-}inorder\ s$

lemma (**in** *adjacency*) *finite-adjacency*:
shows $finite\ (set\ (adjacency\text{-}list\ G\ v))$

lemma (**in** *adjacency*) *distinct-adjacency-list*:
assumes $invar\ G$
shows $distinct\ (adjacency\text{-}list\ G\ v)$

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 how to insert an edge.

definition (**in** *adjacency*) $insert :: 'a \times 'a \Rightarrow 'm \Rightarrow 'm$ **where**
$insert\ p\ G \equiv$
$let\ u = fst\ p;\ v = snd\ p$
$in\ let\ s = case\ Map\text{-}lookup\ G\ u\ of\ None \Rightarrow Set\text{-}empty\ |\ Some\ s' \Rightarrow s'$

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*) $\cup$ ($\bigcup$ *w*$\in$*set* *l*. *if* *v* = *u* *then* {*w*} *else* {})

Let us now look at how to delete an edge.

definition (**in** *adjacency*) *delete* :: $'a \times 'a \Rightarrow 'm \Rightarrow 'm$ **where**
delete *p* *G* $\equiv$
case *Map-lookup* *G* (*fst* *p*) *of*
None $\Rightarrow$ *G* |
Some *s* $\Rightarrow$ *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'* :: $'a \Rightarrow 'a \Rightarrow 'm \Rightarrow 'm$ **where**
delete' $\equiv$ *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 how to union two graphs.

definition (**in** *adjacency*) *insert-2* :: $'a \times 's \Rightarrow 'm \Rightarrow 'm$ **where**
insert-2 *p* *G* $\equiv$
let *v* = *fst* *p*; *s* = *snd* *p*
in *let* *s'* = *case* *Map-lookup* *G* *v* *of* *None* $\Rightarrow$ *s* | *Some* *s''* $\Rightarrow$ *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
    $set\ (adjacency\text{-}list\ (insert\text{-}2\ p\ G)\ u) =$
     $set\ (adjacency\text{-}list\ G\ u) \cup (if\ u = fst\ p\ then\ S.set\ (snd\ p)\ else\ \{\})$

lemma (**in** *adjacency*) *invar-fold-insert-2*:
  assumes *invar G*
  assumes *Ball* (*set l*) (*S.invar* $\circ$ *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* $\circ$ *snd*)
  shows
    $set\ (adjacency\text{-}list\ (fold\ insert\text{-}2\ l\ G)\ v) =$
     $set\ (adjacency\text{-}list\ G\ v) \cup (\bigcup p \in set\ l.\ if\ v = fst\ p\ then\ S.set\ (snd\ p)\ else\ \{\})$

definition (**in** *adjacency*) *union* :: $'m \Rightarrow 'm \Rightarrow 'm$ **where**
  $union\ G1\ G2 \equiv fold\ insert\text{-}2\ (Map\text{-}inorder\ G2)\ G1$

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
    $set\ (adjacency\text{-}list\ (union\ G1\ G2)\ v) =$
     $set\ (adjacency\text{-}list\ G1\ v) \cup set\ (adjacency\text{-}list\ G2\ v)$

Finally, let us look at how to compute the difference of two graphs.

definition (**in** *adjacency*) *delete-2* :: $'a \times 's \Rightarrow 'm \Rightarrow 'm$ **where**
  $delete\text{-}2\ p\ G \equiv$
   $let\ v = fst\ p;\ s = snd\ p$
   $in\ case\ Map\text{-}lookup\ G\ v\ of$
     $None \Rightarrow G\ |$
     $Some\ s' \Rightarrow Map\text{-}update\ v\ (fold\ Set\text{-}delete\ (Set\text{-}inorder\ s)\ s')\ G$

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

abbreviation $f :: 'a \Rightarrow 'a \Rightarrow 's \Rightarrow 's$ **where**
$f\ u\ v \equiv E\text{-}insert\ (u,\ v)$

abbreviation $g :: 'a \times 't \Rightarrow 's \Rightarrow 's$ **where**
$g\ p \equiv fold\ (f\ (fst\ p))\ (Set\text{-}inorder\ (snd\ p))$

abbreviation $E :: 'm \Rightarrow 's$ **where**
$E\ G \equiv fold\ g\ (Map\text{-}inorder\ G)\ E\text{-}empty$

lemma *invar-f*:
assumes $E.invar\ s$
shows $E.invar\ (f\ u\ v\ s)$

lemma *set-f-cong*:
assumes $E.invar\ s$
shows $E.set\ (f\ u\ v\ s) = E.set\ s \cup \{(u,\ v)\}$

lemma *invar-fold-f*:
assumes $E.invar\ s$
shows $E.invar\ (fold\ (f\ u)\ l\ s)$

lemma *invar-g*:
assumes $E.invar\ s$
shows $E.invar\ (g\ p\ s)$

lemma *set-fold-f-cong*:
assumes $E.invar\ s$
shows $E.set\ (fold\ (f\ u)\ l\ s) = E.set\ s \cup \{u\} \times set\ l$

lemma *set-g-cong*:
assumes $E.invar\ s$
shows $E.set\ (g\ p\ s) = E.set\ s \cup \{fst\ p\} \times G.S.set\ (snd\ p)$

lemma *invar-fold-g*:
assumes $E.invar\ s$
shows $E.invar\ (fold\ g\ l\ s)$

lemma *invar-E*:
shows $E.invar\ (E\ G)$

lemma *set-fold-g-cong*:
assumes $E.invar\ s$
shows $E.set\ (fold\ g\ l\ s) = E.set\ s \cup (\bigcup p{\in}set\ l.\ \{fst\ p\} \times G.S.set\ (snd\ p))$

lemma *set-E-cong*:
assumes $G.invar\ G$
shows $E.set\ (E\ G) = \{(u,\ v).\ v \in set\ (G.adjacency\text{-}list\ G\ u)\}$

end

5.1.2 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*) $dE :: 'm \Rightarrow ('a \times 'a)$ *set* **where**
$dE\ G \equiv \{(u,\ v).\ v \in set\ (adjacency\text{-}list\ G\ u)\}$

definition (**in** *adjacency*) $dV :: 'm \Rightarrow 'a\ set$ **where**
$dV\ G \equiv dVs\ (dE\ G)$

lemma (**in** *adjacency*) *mem-adjacency-iff-edge*:
shows $v \in set\ (adjacency\text{-}list\ G\ u) \longleftrightarrow (u,\ v) \in dE\ G$

lemma (**in** *adjacency*) *finite-dE*:
assumes *invar G*
shows *finite* $(dE\ G)$

lemma (**in** *adjacency*) *adjacency-subset-dV*:
shows $set\ (adjacency\text{-}list\ G\ v) \subseteq dV\ G$

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 $dE\ (union\ G1\ G2) = dE\ G1 \cup dE\ G2$

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

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

5.1.3 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**

$E\ G \equiv \{\{u,\ v\} \mid u\ v.\ v \in set\ (adjacency\text{-}list\ G\ u)\}$

definition (**in** *adjacency*) $V :: {'m} \Rightarrow {'a}\ set$ **where**
$V\ G \equiv\ Vs\ (E\ G)$

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

lemma (**in** *symmetric-adjacency*) *mem-adjacency-iff-edge*:
shows $v \in set\ (adjacency\text{-}list\ G\ u) \longleftrightarrow \{u,\ v\} \in E\ G$

lemma (**in** *symmetric-adjacency*) *mem-adjacency-iff-edge-2*:
shows $u \in set\ (adjacency\text{-}list\ G\ v) \longleftrightarrow \{u,\ v\} \in E\ G$

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*) $insert\text{-}edge :: {'a} \Rightarrow {'a} \Rightarrow {'m} \Rightarrow {'m}$ **where**
$insert\text{-}edge\ u\ v\ G \equiv insert'\ v\ u\ (insert'\ u\ v\ G)$

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
$set\ (adjacency\text{-}list\ (insert\text{-}edge\ u\ v\ G)\ w) =$
$set\ (adjacency\text{-}list\ G\ w) \cup (if\ w = u\ then\ \{v\}\ else\ if\ w = v\ then\ \{u\}\ else\ \{\})$

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

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*) ∪
($\bigcup$*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*}

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 *Adjacency-Adaptor*

imports
Directed-Adjacency
../Adaptors/Graph-Adaptor
Undirected-Adjacency
begin

5.2 Edges

5.3 Vertices

lemma (**in** *adjacency*) *V-eq-dV*:
shows $V\ G = dV\ G$

lemma (**in** *adjacency*) *adjacency-subset-V*:
shows $set\ (adjacency\text{-}list\ G\ v) \subseteq V\ G$

5.4

lemma (**in** *symmetric-adjacency*) *dE-eq-dEs*:
shows $dE\ G = dEs$

end
theory *Weighted-Dpath*
imports
Dpath
begin

type-synonym $'a\ weight\text{-}fun = 'a \times 'a \Rightarrow nat$

definition $edges\text{-}weight :: 'a\ weight\text{-}fun \Rightarrow ('a \times 'a)\ list \Rightarrow nat$ **where**
$edges\text{-}weight\ f\ l = sum\text{-}list\ (map\ f\ l)$

definition $dpath\text{-}weight :: 'a\ weight\text{-}fun \Rightarrow 'a\ dpath \Rightarrow nat$ **where**
$dpath\text{-}weight\ f\ p = edges\text{-}weight\ f\ (edges\text{-}of\text{-}dpath\ p)$

lemma *edges-weight-Nil* [*simp*]:
shows $edges\text{-}weight\ f\ [] = 0$

lemma *dpath-weight-Nil* [*simp*]:
shows $dpath\text{-}weight\ f\ [] = 0$

lemma *edges-weight-Cons* [*simp*]:
shows $edges\text{-}weight\ f\ (x\ \#\ xs) = f\ x + edges\text{-}weight\ f\ xs$

lemma *edges-weight-append* [*simp*]:
shows $edges\text{-}weight\ f\ (xs\ @\ ys) = edges\text{-}weight\ f\ xs + edges\text{-}weight\ f\ ys$

lemma *dpath-weight-append*:
assumes $p \neq []$

shows *dpath-weight f* (*p* @ *q*) = *dpath-weight f p* + *dpath-weight f* (*last p* # *q*)

lemma *dpath-weight-append-2*:
  assumes *p* ≠ []
  assumes *q* ≠ []
  assumes *last p* = *hd q*
  shows *dpath-weight f* (*p* @ *tl q*) = *dpath-weight f p* + *dpath-weight f q*

lemma *dpath-weight-append-3*:
  assumes *q* ≠ []
  shows *dpath-weight f* (*p* @ *q*) = *dpath-weight f* (*p* @ [*hd q*]) + *dpath-weight f q*

lemma *dpath-weight-append-append*:
  assumes *p* ≠ []
  assumes *Suc 0* < *length q*
  assumes *r* ≠ []
  assumes *last p* = *hd q*
  assumes *last q* = *hd r*
  shows *dpath-weight f* (*p* @ *tl q* @ *tl r*) = *dpath-weight f p* + *dpath-weight f q* + *dpath-weight f r*

lemma *dpath-weight-closed-dpath-bet-decomp*:
  assumes *dpath-bet G p u v*
  assumes ¬ *distinct p*
  assumes *closed-dpath-bet-decomp G p* = (*q*, *r*, *s*)
  shows *dpath-weight f p* = *dpath-weight f q* + *dpath-weight f r* + *dpath-weight f s*

lemma *dpath-weight-ge-dpath-weight-dpath-bet-to-distinct*:
  assumes *dpath-bet G p u v*
  shows *dpath-weight f* (*dpath-bet-to-distinct G p*) ≤ *dpath-weight f p*

lemma *dpath-length-eq-dpath-weight*:
  shows *dpath-length p* = *dpath-weight* (λ-. *1*) *p*

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*) ∈ *G*
shows *dist G u w* ≤ *dist G u v* + *1*

end

5.4.1 Directed graphs

theory *Directed-Graph*
imports
Shortest-Dpath
begin

end
theory *Parent-Relation*
imports
Main
begin

We (redefine and) extend the formalization of a well-formed parent relation.

definition *follow-invar* :: (*'a* ⇀ *'a*) ⇒ *bool* **where**
follow-invar parent ≡ *wf* {(*u*, *v*). *parent v* = *Some u*}

locale *parent* =
  fixes *parent* :: $'a \rightharpoonup 'a$
  assumes *follow-invar*: *follow-invar parent*

function (**in** *parent*) (*domintros*) *follow* :: $'a \Rightarrow 'a$ *list* **where**
  follow v = (*case parent v of None* ⇒ [*v*] | *Some u* ⇒ *v* # *follow u*)

5.5 Termination

lemma (**in** *parent*)
  assumes *parent v* = *None*
  shows *follow-dom v*

lemma (**in** *parent*)
  assumes *parent v* = *Some u*
  assumes *follow-dom u*
  shows *Wellfounded.accp follow-rel v*

lemma (**in** *parent*) *follow-dom-if-wfP-follow-rel*:
  assumes *wfP follow-rel*
  shows *follow-dom v*

lemma (**in** *parent*) *follow-dom-if-wf-follow-rel*:
  assumes *wf* {(*u*, *v*). *follow-rel u v*}
  shows *follow-dom v*

lemma (**in** *parent*) *follow-rel-eq-parent*:
  shows *follow-rel* = (λ*u v*. *parent v* = *Some u*)

lemma (**in** *parent*) *wf-follow-rel*:
  shows *wf* {(*u*, *v*). *follow-rel u v*}

lemma (**in** *parent*) *follow-dom*:
  shows *follow-dom v*

lemma (**in** *parent*) *follow-pinduct*:
  assumes $\bigwedge v.\ (\bigwedge u.\ parent\ v = Some\ u \Longrightarrow P\ u) \Longrightarrow P\ v$
  shows *P v*

lemma (**in** *parent*) *follow-psimps*:
  shows *follow v* = (*case parent v of None* ⇒ [*v*] | *Some u* ⇒ *v* # *follow u*)

5.6

lemma (**in** *parent*) *follow-non-empty*:
  shows *follow v* ≠ []

lemma (**in** *parent*) *follow-ConsD*:

assumes *follow* $u = v \# p$
shows $v = u$

lemma (**in** *parent*) *follow-Cons-ConsD*:
assumes *follow* $v = v \# u \# p$
shows
follow $u = u \# p$
parent $v =$ *Some* u

lemma (**in** *parent*) *follow-Cons-ConsE*:
assumes *follow* $v = v \# p$
assumes $p \neq []$
obtains u **where** *follow* $u = p$

lemma (**in** *parent*) *follow-appendD*:
assumes *follow* $v = p @ u \# p'$
shows *follow* $u = u \# p'$

lemma (**in** *parent*) *parent-last-follow-eq-None*:
shows *parent* (*last* (*follow* v)) = *None*

lemma (**in** *parent*) *parent-eq-SomeE*:
assumes *parent* $v =$ *Some* u
obtains p **where** *follow* $v = v \# u \# p$

lemma (**in** *parent*) *parent-eq-SomeD*:
assumes *parent* $v =$ *Some* u
shows
$u \neq v$
$v \notin$ *set* (*follow* u)

lemma (**in** *parent*) *distinct-follow*:
shows *distinct* (*follow* v)

lemma (**in** *parent*) *tbd*:
assumes *follow* $v1 = p1 @ u \# p1'$
assumes *follow* $v2 = p2 @ u \# p2'$
shows $p1' = p2'$

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

locale *Queue* =
fixes *empty* :: $'q$
fixes *is-empty* :: $'q \Rightarrow bool$
fixes *snoc* :: $'q \Rightarrow 'a \Rightarrow 'q$

fixes *head* :: $'q \Rightarrow 'a$
fixes *tail* :: $'q \Rightarrow 'q$
fixes *invar* :: $'q \Rightarrow bool$
fixes *list* :: $'q \Rightarrow 'a\ list$
assumes *list-empty*: *list empty* = *Nil*
assumes *is-empty*: *invar q* $\Longrightarrow$ *is-empty q* = (*list q* = *Nil*)
assumes *list-snoc*: *invar q* $\Longrightarrow$ *list* (*snoc q x*) = *list q* @ [*x*]
assumes *list-head*: $[\![$ *invar q*; *list q* $\neq$ *Nil* $]\!]$ $\Longrightarrow$ *head q* = *hd* (*list q*)
assumes *list-tail*: $[\![$ *invar q*; *list q* $\neq$ *Nil* $]\!]$ $\Longrightarrow$ *list* (*tail q*) = *tl* (*list q*)
assumes *invar-empty*: *invar empty*
assumes *invar-snoc*: *invar q* $\Longrightarrow$ *invar* (*snoc q x*)
assumes *invar-tail*: $[\![$ *invar q*; *list q* $\neq$ *Nil* $]\!]$ $\Longrightarrow$ *invar* (*tail q*)

lemma (**in** *Queue*) *list-queue*:
assumes *invar q*
assumes *list q* $\neq$ []
shows *list q* = *head q* # *list* (*tail q*)

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

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

6 BFS

6.1 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* $\Rightarrow$ $'s \Rightarrow$ $'n \Rightarrow$ $'n$ **and**
P-empty **and**
P-update :: $'a \Rightarrow 'a \Rightarrow 'm \Rightarrow 'm$ **and**
P-delete **and**
P-lookup **and**
P-invar **and**
Q-empty **and**
Q-is-empty **and**
Q-snoc :: $'q \Rightarrow 'a \Rightarrow 'q$ **and**
Q-head **and**
Q-tail **and**
Q-invar **and**
Q-list
begin

Our implementation of BFS keeps two data structures, 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 enqueuing v as well as setting v's parent to u.

definition *init* :: $'a \Rightarrow ('q, 'm)$ *state* **where**
init src $\equiv$
(|*queue* = *Q-snoc Q-empty src*,
parent = *P-empty*|)

definition *DONE* :: $('q, 'm)$ *state* $\Rightarrow$ *bool* **where**
DONE s $\longleftrightarrow$ *Q-is-empty* (*queue s*)

definition *is-discovered* :: $'a \Rightarrow 'm \Rightarrow 'a \Rightarrow$ *bool* **where**
is-discovered src m v $\longleftrightarrow$ *v* = *src* $\vee$ *P-lookup m v* $\neq$ *None*

definition *discover* :: $'a \Rightarrow 'a \Rightarrow ('q, 'm)$ *state* $\Rightarrow ('q, 'm)$ *state* **where**
discover u v s $\equiv$
(|*queue* = *Q-snoc* (*queue s*) *v*,
parent = *P-update v u* (*parent s*)|)

definition *traverse-edge* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow ('q, 'm)$ *state* $\Rightarrow ('q, 'm)$ *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* :: $'n \Rightarrow 'a \Rightarrow ('q, 'm)$ *state* $\Rightarrow ('q, 'm)$ *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)

abbreviation *bfs* :: $'n \Rightarrow 'a \Rightarrow 'm$ **where**
  bfs G src ≡ *parent* (*loop G src* (*init src*))

abbreviation *fold* :: $'n \Rightarrow 'a \Rightarrow ('q, 'm)$ *state* $\Rightarrow ('q, 'm)$ *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* :: $'m \Rightarrow 'a$ *dgraph* **where**
  T m ≡ {(*u*, *v*). *P-lookup m v* = *Some u*}

end

6.2 Verification of the correctness of the algorithm

6.2.1 Input

Algorithm λ*Map-lookup Set-inorder P-empty P-update P-lookup Q-empty Q-is-empty Q-snoc Q-head Q-tail G src. state.parent* (*bfs.loop Map-lookup Set-inorder P-update P-lookup Q-is-empty Q-snoc Q-head Q-tail G src* (*bfs.init P-empty Q-empty Q-snoc src*)) 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* $\Rightarrow 's \Rightarrow 'n \Rightarrow 'n$ **and**
  P-update :: $'a \Rightarrow 'a \Rightarrow 'm \Rightarrow 'm$ **and**
  Q-snoc :: $'q \Rightarrow 'a \Rightarrow 'q$ +
  fixes *G* :: $'n$

fixes *src* :: *'a*
assumes *invar-G*: *G.invar G*
assumes *src-mem-dV*: *src* ∈ *G.dV G*

abbreviation (**in** *bfs*) *bfs-valid-input'* :: *'n* ⇒ *'a* ⇒ *bool* **where**
bfs-valid-input' G src ≡
bfs-valid-input
Map-empty Map-delete Map-lookup Map-inorder Map-inv
Set-empty Set-insert Set-delete Set-isin Set-inorder Set-inv
P-empty P-delete P-lookup P-invar
Q-empty Q-is-empty Q-head Q-tail Q-invar Q-list
Map-update P-update Q-snoc G src

6.2.2 Loop invariants

Unfolding the definition of λ*Map-lookup Set-inorder P-empty P-update P-lookup Q-empty Q-is-empty Q-snoc Q-head Q-tail G src. state.parent* (*bfs.loop Map-lookup Set-inorder P-update P-lookup Q-is-empty Q-snoc Q-head Q-tail G src* (*bfs.init P-empty Q-empty Q-snoc src*)), we see that function *bfs.loop* lies at the heart of the algorithm. It expects the undirected graph G, the source vertex src in G, as well as the current state s, which comprises the queue and parent map, as input. Let us look at the assumptions on the queue and parent map. As these are the only two data structures that may change from one iteration to the next, these assumptions constitute the loop invariants of *bfs.loop*.

To keep track of progress, the algorithm colors every vertex in G either white, gray, or black. All vertices start out white and may later become gray and then black.

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* $\Longrightarrow$ (*u*, *v*) $\in$ *G.dE G*
assumes *not-white-if-mem-queue*: *v* $\in$ *set* (*Q-list* (*queue s*)) $\Longrightarrow$ $\neg$ *white s v*
assumes *not-white-if-parent*: *P-lookup* (*parent s*) *v* = *Some u* $\Longrightarrow$ $\neg$ *white s u*
assumes *black-imp-adjacency-not-white*: $[\![$ (*u*, *v*) $\in$ *G.dE G*; *black s u* $]\!]$ $\Longrightarrow$ $\neg$ *white s v*
assumes *queue-sorted-wrt-d*: *sorted-wrt* (λu *v*. *d* (*parent s*) *u* $\le$ *d* (*parent s*) *v*) (*Q-list* (*queue s*))
assumes *d-last-queue-le*:
$\neg$ *Q-is-empty* (*queue s*) $\Longrightarrow$
d (*parent s*) (*last* (*Q-list* (*queue s*))) $\le$ *d* (*parent s*) (*Q-head* (*queue s*)) + *1*
assumes *d-triangle-inequality*:
$[\![$ *dpath-bet* (*G.dE G*) *p u v*; $\neg$ *white s u*; $\neg$ *white s v* $]\!]$ $\Longrightarrow$
d (*parent s*) *v* $\le$ *d* (*parent s*) *u* + *dpath-length p*

Invariant $[\![$*bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G ?src ?P-update ?Q-snoc ?s*; *?u* $\rightarrow_{adjacency.dE\ ?Map\text{-}lookup\ ?Set\text{-}inorder\ ?G}$ *?v*; *bfs.is-discovered ?P-lookup ?src* (*state.parent ?s*) *?u* $\wedge$ *?u* $\notin$ *set* (*?Q-list* (*queue ?s*))$]\!]$ $\Longrightarrow$ $\neg$ $\neg$ *bfs.is-discovered ?P-lookup ?src* (*state.parent ?s*) *?v* says that all vertices adjacent to black vertices have been discovered.

For a vertex v in G, let d v denote the distance from the source src to v induced by the current parent map.

Let $v_1, \ldots, v_k$ be the content of the current queue, where v_1 is the head. Then invariant *bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G ?src ?P-update ?Q-snoc ?s* $\Longrightarrow$ *sorted-wrt* (λu *v*. *dpath-length* (*rev* (*parent.follow* (*?P-lookup* (*state.parent ?s*)) *u*)) $\le$ *dpath-length* (*rev* (*parent.follow* (*?P-lookup* (*state.parent ?s*)) *v*))) (*?Q-list* (*queue ?s*)) says that $dv_i \le dv_{i+1}$ for all $i < k$. And invariant $[\![$*bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G ?src ?P-update ?Q-snoc ?s*; $\neg$ *?Q-is-empty* (*queue ?s*)$]\!]$ $\Longrightarrow$ *dpath-length* (*rev* (*parent.follow* (*?P-lookup* (*state.parent ?s*)) (*last* (*?Q-list* (*queue ?s*))))) $\le$ *dpath-length* (*rev* (*parent.follow* (*?P-lookup* (*state.parent ?s*)) (*?Q-head* (*queue ?s*)))) + *1* says that $dv_k \le dv_1 + 1$. That is, the current queue holds at most two distinct d values.

Finally, invariant $[\![$*bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder*

?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G ?src ?P-update ?Q-snoc ?s; *dpath-bet* (*adjacency.dE ?Map-lookup ?Set-inorder ?G*) *?p ?u ?v*; ¬ ¬ *bfs.is-discovered ?P-lookup ?src* (*state.parent ?s*) *?u*; ¬ ¬ *bfs.is-discovered ?P-lookup ?src* (*state.parent ?s*) *?v*⟧ ⟹ *dpath-length* (*rev* (*parent.follow* (*?P-lookup* (*state.parent ?s*)) *?v*)) ≤ *dpath-length* (*rev* (*parent.follow* (*?P-lookup* (*state.parent ?s*)) *?u*)) + *dpath-length ?p* 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 have been discovered by the algorithm, then their d values differ by at most the length of that path.

abbreviation (**in** *bfs*) *bfs-invar′* :: *′n* ⇒ *′a* ⇒ (*′q*, *′m*) *state* ⇒ *bool* **where**
  bfs-invar′ G src s ≡
   bfs-invar
    Map-empty Map-delete Map-lookup Map-inorder Map-inv
    Set-empty Set-insert Set-delete Set-isin Set-inorder Set-inv
    P-empty P-delete P-lookup P-invar
    Q-empty Q-is-empty Q-head Q-tail Q-invar Q-list
    Map-update G src P-update Q-snoc s

abbreviation (**in** *bfs-valid-input*) *bfs-invar″* :: (*′q*, *′m*) *state* ⇒ *bool* **where**
  bfs-invar″ ≡ *bfs-invar′ G src*

Let us quickly show that the initial configuration of the queue–containing only the source vertex src–and parent map–the empty map–satisfies the loop invariants.

lemma (**in** *bfs-valid-input*) *follow-invar-parent-init*:
  shows *follow-invar* (*P-lookup* (*parent* (*init src*)))

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

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

lemma (**in** *bfs-valid-input*) *parent-src-init*:
  shows *P-lookup* (*parent* (*init src*)) *src* = *None*

lemma (**in** *bfs-valid-input*) *parent-imp-edge-init*:
  assumes *P-lookup* (*parent* (*init src*)) *v* = *Some u*
  shows (*u*, *v*) ∈ *G.dE G*

lemma (**in** *bfs-valid-input*) *not-white-if-mem-queue-init*:
  assumes *v* ∈ *set* (*Q-list* (*queue* (*init src*)))
  shows ¬ *white* (*init src*) *v*

lemma (**in** *bfs-valid-input*) *not-white-if-parent-init*:

assumes *P-lookup* (*parent* (*init src*)) *v* = *Some u*
shows ¬ *white* (*init src*) *u*

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

lemma (**in** *bfs-valid-input*) *queue-sorted-wrt-d-init*:
shows *sorted-wrt* (λ*u v*. *d* (*parent* (*init src*)) *u* ≤ *d* (*parent* (*init src*)) *v*) (*Q-list* (*queue* (*init src*)))

lemma (**in** *bfs-valid-input*) *d-last-queue-le-init*:
assumes ¬ *Q-is-empty* (*queue* (*init src*))
shows
d (*parent* (*init src*)) (*last* (*Q-list* (*queue* (*init src*)))) ≤
d (*parent* (*init src*)) (*Q-head* (*queue* (*init src*))) + *1*

lemma (**in** *bfs-valid-input*) *d-triangle-inequality-init*:
assumes *dpath-bet* (*G.dE G*) *p u v*
assumes ¬ *white* (*init src*) *u*
assumes ¬ *white* (*init src*) *v*
shows *d* (*parent* (*init src*)) *v* ≤ *d* (*parent* (*init src*)) *u* + *dpath-length p*

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

Let us now show that the loop invariants are maintained, that is, if they are satisfied at the start of an iteration, then they also will be satisfied at the end.

For this, let us first look at how the different subroutines change the queue and parent map.

How does *bfs.discover* 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*)

How does *bfs.traverse-edge* change the queue and parent map?

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*) *invar-queue-traverse-edge*:

assumes *Q-invar* (*queue s*)
shows *Q-invar* (*queue* (*traverse-edge src u v 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*) *invar-parent-traverse-edge*:
assumes *P-invar* (*parent s*)
shows *P-invar* (*parent* (*traverse-edge src u v s*))

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* {})

How does λ*Map-lookup Set-inorder P-update P-lookup Q-snoc Q-head Q-tail G src s. fold* (*bfs.traverse-edge P-update P-lookup Q-snoc src* (*Q-head* (*queue s*))) (*adjacency.adjacency-list Map-lookup Set-inorder G* (*Q-head* (*queue s*))) (*s*(|*queue* := *Q-tail* (*queue s*)|)) change the queue and parent map?

lemma (**in** *bfs*) *list-queue-fold-cong-aux*:
assumes *P-invar* (*parent s*)
assumes *distinct* (*v* # *vs*)
shows *filter* (*Not* ∘ *is-discovered src* (*parent* (*traverse-edge src u v s*))) *vs* = *filter* (*Not* ∘ *is-discovered src* (*parent s*)) *vs*

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*) *invar-tail*:
assumes *Q-invar* (*queue s*)
assumes ¬ *DONE s*

shows *Q-invar* (*queue* (*s*⦇*queue* := *Q-tail* (*queue s*)⦈))

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-aux*:
assumes *P-invar* (*parent s*)
assumes *distinct* (*v* # *vs*)
shows *w* ∈ *set vs* ∧ ¬ *is-discovered src* (*parent* (*traverse-edge src u v s*)) *w* ⟷ *w* ∈ *set vs* ∧ ¬ *is-discovered src* (*parent s*) *w*

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*) $\cup$ {(*u*, *v*) |*v*. *v* $\in$ *set l* $\wedge$ $\neg$ *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*) $\cup$
{(*Q-head* (*queue s*), *v*) |*v*. *v* $\in$ *set* (*G.adjacency-list G* (*Q-head* (*queue s*))) $\wedge$ $\neg$ *is-discovered src* (*parent s*) *v*}

lemma (**in** *bfs-invar*) *T-fold-cong*:
shows
T (*parent* (*fold G src s*)) =
T (*parent s*) $\cup$
{(*Q-head* (*queue s*), *v*) |*v*. *v* $\in$ *set* (*G.adjacency-list G* (*Q-head* (*queue s*))) $\wedge$ $\neg$ *is-discovered src* (*parent s*) *v*}

We are now ready to prove that the variants are maintained.

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

abbreviation (**in** *bfs*) *bfs-invar-not-DONE'* :: $'n$ $\Rightarrow$ $'a$ $\Rightarrow$ ($'q$, $'m$) *state* $\Rightarrow$ *bool* **where**
bfs-invar-not-DONE' G src s $\equiv$
bfs-invar-not-DONE
Map-empty Map-delete Map-lookup Map-inorder Map-inv
Set-empty Set-insert Set-delete Set-isin Set-inorder Set-inv
P-empty P-delete P-lookup P-invar
Q-empty Q-is-empty Q-head Q-tail Q-invar Q-list
Map-update G src s P-update Q-snoc

abbreviation (**in** *bfs-valid-input*) *bfs-invar-not-DONE''* :: ($'q$, $'m$) *state* $\Rightarrow$ *bool* **where**
bfs-invar-not-DONE'' $\equiv$ *bfs-invar-not-DONE' G src*

We start with the first invariant.

lemma (**in** *bfs*) *list-queue-non-empty*:
assumes *Q-invar* (*queue s*)
assumes $\neg$ *DONE s*
shows *Q-list* (*queue s*) $\neq$ []

lemma (**in** *bfs-invar-not-DONE*) *list-queue-non-empty*:
shows *Q-list* (*queue s*) ≠ []

lemma (**in** *bfs-invar-not-DONE*) *head-queue-mem-queue*:
shows *Q-head* (*queue s*) ∈ *set* (*Q-list* (*queue s*))

lemma (**in** *bfs-invar-not-DONE*) *not-white-head-queue*:
shows ¬ *white s* (*Q-head* (*queue s*))

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

Then the second invariant, *bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G ?src ?P-update ?Q-snoc ?s* ⟹ *?Q-invar* (*queue ?s*).

lemma (**in** *bfs*) *invar-queue-fold*:
assumes *Q-invar* (*queue s*)
assumes *distinct l*
shows *Q-invar* (*queue* (*List.fold* (*traverse-edge src u*) *l s*))

lemma (**in** *bfs*) *invar-queue-fold-2*:
assumes *G.invar G*
assumes *Q-invar* (*queue s*)
assumes ¬ *DONE s*
shows *Q-invar* (*queue* (*fold G src s*))

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

Then the third invariant, *bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G ?src ?P-update ?Q-snoc ?s* ⟹ *?P-invar* (*state.parent ?s*).

lemma (**in** *bfs*) *invar-parent-fold*:
assumes *P-invar* (*parent s*)
assumes *distinct l*
shows *P-invar* (*parent* (*List.fold* (*traverse-edge src u*) *l s*))

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

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

Then the fourth invariant, *bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G ?src ?P-update ?Q-snoc ?s* $\Longrightarrow$ *?P-lookup* (*state.parent ?s*) *?src* = *None*.

lemma (**in** *bfs-valid-input*) *src-not-white*:
  shows ¬ *white s src*

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

Then the fifth invariant, ⟦*bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G ?src ?P-update ?Q-snoc ?s*; *?P-lookup* (*state.parent ?s*) *?v* = *Some ?u*⟧ $\Longrightarrow$ $?u \rightarrow_{adjacency.dE\ ?Map\text{-}lookup\ ?Set\text{-}inorder\ ?G} ?v$.

lemma (**in** *bfs-invar*) *head-queueI*:
  assumes *P-lookup* (*parent s*) *v* ≠ *Some u*
  assumes *P-lookup* (*parent* (*fold G src s*)) *v* = *Some u*
  shows *u* = *Q-head* (*queue s*)

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*

Then the sixth invariant, ⟦*bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G ?src ?P-update ?Q-snoc ?s*; *?v* ∈ *set* (*?Q-list* (*queue ?s*))⟧ $\Longrightarrow$ ¬ ¬ *bfs.is-discovered ?P-lookup ?src* (*state.parent ?s*) *?v*.

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

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 $v \in$ *set* (*Q-list* (*queue* (*fold G src s*)))
  shows ¬ *white* (*fold G src s*) *v*

Then the seventh invariant, ⟦*bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G ?src ?P-update ?Q-snoc ?s*; *?P-lookup* (*state.parent ?s*) *?v* = *Some ?u*⟧ ⟹ ¬ ¬ *bfs.is-discovered ?P-lookup ?src* (*state.parent ?s*) *?u*.

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*

Then the eighth invariant, ⟦*bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G ?src ?P-update ?Q-snoc ?s*; $?u \rightarrow_{adjacency.dE\ ?Map\text{-}lookup\ ?Set\text{-}inorder\ ?G} ?v$; *bfs.is-discovered ?P-lookup ?src* (*state.parent ?s*) *?u* ∧ *?u* ∉ *set* (*?Q-list* (*queue ?s*))⟧ ⟹ ¬ ¬ *bfs.is-discovered ?P-lookup ?src* (*state.parent ?s*) *?v*.

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*) *whiteD*:
  assumes *white s v*
  shows ¬ *black* (*fold G src s*) *v*

lemma (**in** *bfs-invar-not-DONE*) *head-queueI-2*:
  assumes $v \in$ *set* (*Q-list* (*queue s*))
  assumes $v \notin$ *set* (*Q-list* (*queue* (*fold G src s*)))
  shows *v* = *Q-head* (*queue s*)

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*

Then the ninth invariant, *bfs-invar ?Map-empty ?Map-delete ?Map-lookup*

?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G ?src ?P-update ?Q-snoc ?s $\Longrightarrow$ *sorted-wrt* ($\lambda u\ v.$ *dpath-length* (*rev* (*parent.follow* (*?P-lookup* (*state.parent ?s*)) *u*)) $\leq$ *dpath-length* (*rev* (*parent.follow* (*?P-lookup* (*state.parent ?s*)) *v*))) (*?Q-list* (*queue ?s*)).

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

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

lemma (**in** *bfs-invar*) *mem-queue-imp-d-le*:
assumes *v* $\in$ *set* (*Q-list* (*queue s*))
shows *d* (*parent s*) *v* $\leq$ *d* (*parent s*) (*last* (*Q-list* (*queue s*)))

lemma (**in** *bfs-invar-not-DONE*) *mem-filterD*:
assumes *v* $\in$ *set* (*filter* (*Not* $\circ$ *is-discovered src* (*parent s*)) (*G.adjacency-list G* (*Q-head* (*queue s*))))
shows
d (*parent* (*fold G src s*)) *v* = *d* (*parent* (*fold G src s*)) (*Q-head* (*queue s*)) + *1*
d (*parent* (*fold G src s*)) (*last* (*Q-list* (*queue s*))) $\leq$ *d* (*parent* (*fold G src s*)) *v*

lemma (**in** *bfs-invar-not-DONE*) *queue-sorted-wrt-d-fold-aux*:
assumes *u-mem-tail-queue*: *u* $\in$ *set* (*Q-list* (*Q-tail* (*queue s*)))
assumes *v-mem-filter*: *v* $\in$ *set* (*filter* (*Not* $\circ$ *is-discovered src* (*parent s*)) (*G.adjacency-list G* (*Q-head* (*queue s*))))
shows *d* (*parent* (*fold G src s*)) *u* $\leq$ *d* (*parent* (*fold G src s*)) *v*

lemma (**in** *bfs-invar-not-DONE*) *queue-sorted-wrt-d-fold*:
shows *sorted-wrt* ($\lambda u\ v.$ *d* (*parent* (*fold G src s*)) *u* $\leq$ *d* (*parent* (*fold G src s*)) *v*) (*Q-list* (*queue* (*fold G src s*)))

Then the tenth invariant, $[\![$*bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G ?src ?P-update ?Q-snoc ?s*; $\neg$ *?Q-is-empty* (*queue ?s*)$]\!]$ $\Longrightarrow$ *dpath-length* (*rev* (*parent.follow* (*?P-lookup* (*state.parent ?s*)) (*last* (*?Q-list* (*queue ?s*))))) $\leq$ *dpath-length* (*rev* (*parent.follow* (*?P-lookup* (*state.parent ?s*)) (*?Q-head* (*queue ?s*)))) + *1*.

lemma (**in** *bfs-invar-not-DONE*) *d-last-queue-le-fold-aux*:

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 s*)) + *1*

lemma (**in** *bfs-invar*) *mem-queue-imp-d-ge*:
assumes *v* ∈ *set* (*Q-list* (*queue s*))
shows *d* (*parent s*) (*Q-head* (*queue s*)) ≤ *d* (*parent s*) *v*

lemma (**in** *bfs-invar-not-DONE*) *d-last-queue-le-fold-aux-2*:
assumes ¬ *Q-is-empty* (*queue* (*fold G src s*))
shows *d* (*parent* (*fold G src s*)) (*Q-head* (*queue s*)) ≤ *d* (*parent* (*fold G src s*)) (*Q-head* (*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*

Finally, the eleventh invariant, ⟦*bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G ?src ?P-update ?Q-snoc ?s*; *dpath-bet* (*adjacency.dE ?Map-lookup ?Set-inorder ?G*) *?p ?u ?v*; ¬ ¬ *bfs.is-discovered ?P-lookup ?src* (*state.parent ?s*) *?u*; ¬ ¬ *bfs.is-discovered ?P-lookup ?src* (*state.parent ?s*) *?v*⟧ ⟹ *dpath-length* (*rev* (*parent.follow* (*?P-lookup* (*state.parent ?s*)) *?v*)) ≤ *dpath-length* (*rev* (*parent.follow* (*?P-lookup* (*state.parent ?s*)) *?u*)) + *dpath-length ?p*.

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

lemma (**in** *bfs-invar*) *white-not-white-foldD*:
assumes *white s v*
assumes ¬ *white* (*fold G src s*) *v*
shows
v ∈ *set* (*G.adjacency-list G* (*Q-head* (*queue s*)))
P-lookup (*parent* (*fold G src s*)) *v* = *Some* (*Q-head* (*queue s*))

lemma (**in** *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** *bfs-invar-not-DONE*) *white-not-white-foldD-2*:
  assumes *white s v*
  assumes ¬ *white* (*fold G src s*) *v*
  shows *d* (*parent* (*fold G src s*)) *v* = *d* (*parent* (*fold G src s*)) (*Q-head* (*queue s*)) + *1*

lemmas (**in** *bfs-invar-not-DONE*) *white-not-white-foldD* =
  white-not-white-foldD
  white-not-white-foldD-2

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*

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

6.3 *Q-list* ∘ *queue*

6.4 *Q-head* ∘ *queue*

lemma (**in** *bfs*) *head-queue-mem-dV*:
  assumes *Q-invar* (*queue s*)
  assumes *set* (*Q-list* (*queue s*)) ⊆ *G.dV G*
  assumes ¬ *DONE s*
  shows *Q-head* (*queue s*) ∈ *G.dV G*

7 Basic Lemmas

7.1 *discover*

7.1.1 *queue*

7.1.2 *state.parent*

7.2 *traverse-edge*

7.2.1 *queue*

7.2.2 *Q-list* ∘ *queue*

7.2.3 *P-lookup* ∘ *state.parent*

7.2.4 *P-invar* ∘ *state.parent*

7.2.5 *T*

7.3 *fold*

7.3.1 *Q-invar* ∘ *queue*

7.3.2 *Q-list* ∘ *queue*

7.3.3 *set* ∘ *Q-list* ∘ *queue*

lemma (**in** *bfs*) *queue-fold-subset-dV*:
  assumes *G.invar G*
  assumes *Q-invar* (*queue s*)
  assumes *P-invar* (*parent s*)
  assumes *set* (*Q-list* (*queue s*)) ⊆ *G.dV G*
  assumes ¬ *DONE s*
  shows *set* (*Q-list* (*queue* (*fold G src s*))) ⊆ *G.dV G*

7.3.4 *state.parent*

7.3.5 *P-invar* ∘ *state.parent*

lemma (**in** *bfs*) *dom-parent-fold-subset-dV*:
  assumes *P-invar* (*parent s*)
  assumes *distinct l*
  assumes *P.dom* (*parent s*) ⊆ *G.dV G*
  assumes *set l* ⊆ *G.dV G*
  shows *P.dom* (*parent* (*List.fold* (*traverse-edge src u*) *l s*)) ⊆ *G.dV G*

lemma (**in** *bfs*) *dom-parent-fold-subset-dV-2*:
  assumes *G.invar G*
  assumes *P-invar* (*parent s*)
  assumes *P.dom* (*parent s*) ⊆ *G.dV G*
  shows *P.dom* (*parent* (*fold G src s*)) ⊆ *G.dV G*

lemma (**in** *bfs*) *ran-parent-fold-cong*:
  assumes *G.invar G*
  assumes *P-invar* (*parent s*)
  shows
    P.ran (*parent* (*fold G src s*)) =
     P.ran (*parent s*) ∪
     (*if set* (*filter* (*Not* ∘ *is-discovered src* (*parent s*)) (*G.adjacency-list G* (*Q-head* (*queue s*)))) = {}
       then {}
       else {*Q-head* (*queue s*)})

7.3.6 *T*

8 Termination

lemma (**in** *bfs*) *loop-dom-aux*:
  assumes *G.invar G*
  assumes *P-invar* (*parent s*)
  assumes *P.dom* (*parent s*) ⊆ *G.dV G*
  shows
    card (*P.dom* (*parent* (*fold G src s*))) =
     card (*P.dom* (*parent s*)) +
    card (*set* (*filter* (*Not* ∘ *is-discovered src* (*parent s*)) (*G.adjacency-list G* (*Q-head* (*queue s*)))))

lemma (**in** *bfs*) *loop-dom-aux-2*:
  assumes *invar-G*: *G.invar G*
  assumes *invar-queue*: *Q-invar* (*queue s*)
  assumes *not-DONE*: ¬ *DONE s*
  assumes *dom-parent-subset-dV*: *P.dom* (*parent s*) ⊆ *G.dV G*
  shows
    card (*G.dV G*) +
     length (*Q-list* (*Q-tail* (*queue s*))) −
     card (*P.dom* (*parent s*)) <
     card (*G.dV G*) +
     length (*Q-list* (*queue s*)) −
     card (*P.dom* (*parent s*))

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*)

9 Invariants

9.1 Definitions

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*

abbreviation (**in** *bfs*) *bfs-invar-DONE'* :: *'n* ⇒ *'a* ⇒ (*'q*, *'m*) *state* ⇒ *bool* **where**
bfs-invar-DONE' G src s ≡
bfs-invar-DONE
Map-empty Map-delete Map-lookup Map-inorder Map-inv
Set-empty Set-insert Set-delete Set-isin Set-inorder Set-inv
P-empty P-delete P-lookup P-invar
Q-empty Q-is-empty Q-head Q-tail Q-invar Q-list
Map-update G src s P-update Q-snoc

abbreviation (**in** *bfs-valid-input*) *bfs-invar-DONE''* :: (*'q*, *'m*) *state* ⇒ *bool* **where**
bfs-invar-DONE'' ≡ *bfs-invar-DONE' G src*

9.2 Convenience Lemmas

9.2.1 *bfs*

lemma (**in** *bfs*) *bfs-invar-not-DONE'I*:
assumes *bfs-invar' G src s*
assumes ¬ *DONE s*
shows *bfs-invar-not-DONE' G src s*

lemma (**in** *bfs*) *bfs-invar-DONE'I*:
assumes *bfs-invar' G src s*
assumes *DONE s*
shows *bfs-invar-DONE' G src s*

lemma (**in** *bfs*) *rev-follow-non-empty*:
assumes *Parent-Relation.parent* (*P-lookup m*)
shows *rev-follow m v* ≠ []

lemma (**in** *bfs*) *distinct-rev-follow*:
assumes *Parent-Relation.parent* (*P-lookup m*)
shows *distinct* (*rev-follow m v*)

lemma (**in** *bfs*) *last-rev-follow*:
  assumes *Parent-Relation.parent* (*P-lookup m*)
  shows *last* (*rev-follow m v*) = *v*

9.2.2 *bfs-valid-input*

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

9.2.3 *bfs-invar*

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

9.3 Basic Lemmas

9.3.1 *bfs-valid-input*

9.3.2 *bfs-invar*

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*

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

lemma (**in** *bfs-invar*) *d-triangle-inequality-edge*:
  assumes (*u*, *v*) ∈ *G.dE G*
  assumes ¬ *white s u*
  assumes ¬ *white s v*
  shows *d* (*parent s*) *v* ≤ *d* (*parent s*) *u* + *1*

9.4 *bfs.init*

9.4.1

9.4.2

9.5 λ*Map-lookup Set-inorder P-update P-lookup Q-snoc Q-head Q-tail G src s. fold* (*bfs.traverse-edge P-update P-lookup Q-snoc src* (*Q-head* (*queue s*))) (*adjacency.adjacency-list Map-lookup Set-inorder G* (*Q-head* (*queue s*))) (*s*⦇*queue* := *Q-tail* (*queue s*)⦈)

9.5.1 Convenience Lemmas

9.5.2

9.5.3 *bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G ?src ?P-update ?Q-snoc ?s* $\Longrightarrow$ *?Q-invar* (*queue ?s*)

9.5.4 *bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G ?src ?P-update ?Q-snoc ?s* $\Longrightarrow$ *?P-invar* (*state.parent ?s*)

9.5.5 *bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G ?src ?P-update ?Q-snoc ?s* $\Longrightarrow$ *?P-lookup* (*state.parent ?s*) *?src* = *None*

9.5.6 Basic Lemmas

lemmas (**in** *bfs-invar-not-DONE*) *not-whiteD* =
not-white-imp-not-white-fold
not-white-imp-lookup-parent-fold-eq-lookup-parent
not-white-imp-rev-follow-fold-eq-rev-follow

9.5.7 $\llbracket$*bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G ?src ?P-update ?Q-snoc ?s*; *?P-lookup* (*state.parent ?s*) *?v* = *Some ?u*$\rrbracket \Longrightarrow$ *?u* $\rightarrow_{adjacency.dE\ ?Map\text{-}lookup\ ?Set\text{-}inorder\ ?G}$

9.5.8 $\llbracket$*bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G ?src ?P-update ?Q-snoc ?s*; *?v* $\in$ *set* (*?Q-list* (*queue ?s*))$\rrbracket \Longrightarrow \neg\ \neg$ *bfs.is-discovered ?P-lookup ?src* (*state.parent ?s*) *?v*

9.5.9 $\llbracket$*bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G ?src ?P-update ?Q-snoc ?s*; *?P-lookup* (*state.parent ?s*) *?v* = *Some ?u*$\rrbracket \Longrightarrow \neg\ \neg$ *bfs.is-discovered ?P-lookup ?src* (*state.parent ?s*) *?u*

9.5.10 $\llbracket$*bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G ?src ?P-update ?Q-snoc ?s*; *?u* $\rightarrow_{adjacency.dE\ ?Map\text{-}lookup\ ?Set\text{-}inorder\ ?G}$ *?v*; *bfs.is-discovered ?P-lookup ?src* (*state.parent ?s*) *?u* $\wedge$ *?u* $\notin$ *set* (*?Q-list* (*queue ?s*))$\rrbracket$ $\Longrightarrow \neg\ \neg$ *bfs.is-discovered ?P-lookup ?src* (*state.parent ?s*) *?v*

9.5.11 *bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G ?src ?P-update ?Q-snoc ?s* $\Longrightarrow$ *sorted-wrt* ($\lambda u\ v.$ *dpath-length* (*rev* (*parent.follow* (*?P-lookup* (*state.parent ?s*)) *u*)) $\le$ *dpath-length* (*rev* (*parent.follow* (*?P-lookup* (*state.parent ?s*)) *v*))) (*?Q-list* (*queue ?s*))

9.5.12 $\llbracket$*bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G ?src ?P-update ?Q-snoc ?s*; $\neg$ *?Q-is-empty* (*queue ?s*)$\rrbracket \Longrightarrow$ *dpath-length* (*rev* (*parent.follow* (*?P-lookup* (*state.parent ?s*)) (*last* (*?Q-list* (*queue ?s*))))) $\le$ *dpath-length* (*rev* (*parent.follow* (*?P-lookup* (*state.parent ?s*)) (*?Q-head* (*queue ?s*)))) + *1*

9.5.13 $\llbracket$*bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G ?src ?P-update ?Q-snoc ?s*; *dpath-bet* (*adjacency.dE ?Map-lookup ?Set-inorder ?G*) *?p ?u ?v*; $\neg\ \neg$ *bfs.is-discovered ?P-lookup ?src* (*state.parent ?s*) *?u*; $\neg\ \neg$ *bfs.is-discovered ?P-lookup ?src* (*state.parent ?s*) *?v*$\rrbracket \Longrightarrow$ *dpath-length* (*rev* (*parent.follow* (*?P-lookup* (*state.parent ?s*)) *?v*)) $\le$ *dpath-length* (*rev* (*parent.follow* (*?P-lookup* (*state.parent ?s*)) *?u*)) + *dpath-length ?p*

9.5.14

$dist\ G \equiv Shortest\text{-}Dpath.dist\ (G.dE\ G)$

abbreviation (**in** *bfs*) $is\text{-}shortest\text{-}dpath :: {'}n \Rightarrow {'}a\ list \Rightarrow {'}a \Rightarrow {'}a \Rightarrow bool$ **where**
$is\text{-}shortest\text{-}dpath\ G\ p\ u\ v \equiv dpath\text{-}bet\ (G.dE\ G)\ p\ u\ v \wedge dpath\text{-}length\ p = dist\ G\ u\ v$

10.2 Basic Lemmas

lemma (**in** *bfs-invar*) *queue-subset-dV*:
shows $set\ (Q\text{-}list\ (queue\ s)) \subseteq G.dV\ G$

lemma (**in** *bfs-invar*) *dom-parent-subset-dV*:
shows $P.dom\ (parent\ s) \subseteq G.dV\ G$

10.3 Convenience Lemmas

lemma (**in** *bfs-invar*) *loop-dom*:
shows $loop\text{-}dom\ (G,\ src,\ s)$

lemma (**in** *bfs*) *loop-psimps*:
assumes $bfs\text{-}invar'\ G\ src\ s$
shows $loop\ G\ src\ s = (if\ \neg\ DONE\ s\ then\ loop\ G\ src\ (fold\ G\ src\ s)\ else\ s)$

lemma (**in** *bfs-invar-not-DONE*) *loop-psimps*:
shows $loop\ G\ src\ s = loop\ G\ src\ (fold\ G\ src\ s)$

lemma (**in** *bfs-invar-DONE*) *loop-psimps*:
shows $loop\ G\ src\ s = s$

lemma (**in** *bfs*) *bfs-induct*:
assumes $bfs\text{-}invar'\ G\ src\ s$
assumes $\bigwedge G\ src\ s.\ (\neg\ DONE\ s \Longrightarrow P\ G\ src\ (fold\ G\ src\ s)) \Longrightarrow P\ G\ src\ s$
shows $P\ G\ src\ s$

10.4 Completeness

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

lemma (**in** *bfs-valid-input*) *completeness*:
assumes $bfs\text{-}invar''\ s$
assumes $\neg\ is\text{-}discovered\ src\ (parent\ (loop\ G\ src\ s))\ v$
shows $\neg\ reachable\ (G.dE\ G)\ src\ v$

10.5 Soundness

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*) *soundness*:
  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*

10.6 Correctness

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*) *correctness*:
  assumes *bfs-invar*$''$ *s*
  shows *is-shortest-dpath-Map G src* (*parent* (*loop G src s*))

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

corollary (**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
theory *Undirected-BFS*
imports
../Graph/Adjacency/Adjacency-Adaptor
BFS
../Graph/Adaptors/Shortest-Path-Adaptor
begin

11 Invariants

locale *undirected-bfs-valid-input* = *bfs* **where**
Map-update = *Map-update* **and**
P-update = *P-update* **and**
Q-snoc = *Q-snoc* **for**
Map-update :: $'a$::*linorder* $\Rightarrow$ $'s \Rightarrow 'n \Rightarrow 'n$ **and**
P-update :: $'a \Rightarrow 'a \Rightarrow 'm \Rightarrow 'm$ **and**
Q-snoc :: $'q \Rightarrow 'a \Rightarrow 'q$ +
fixes *G* :: $'n$
fixes *src* :: $'a$
assumes *invar-G*: *G.invar G*
assumes *symmetric*: $v \in$ *set* (*G.adjacency-list G u*) $\longleftrightarrow$ $u \in$ *set* (*G.adjacency-list G v*)
assumes *src-mem-V*: *src* $\in$ *G.V G*
begin

sublocale *symmetric-adjacency*

sublocale *bfs-valid-input*

end

abbreviation (**in** *bfs*) *undirected-bfs-valid-input'* :: $'n \Rightarrow 'a \Rightarrow$ *bool* **where**
undirected-bfs-valid-input' G src $\equiv$
undirected-bfs-valid-input
Map-empty Map-delete Map-lookup Map-inorder Map-inv
Set-empty Set-insert Set-delete Set-isin Set-inorder Set-inv
P-empty P-delete P-lookup P-invar
Q-empty Q-is-empty Q-head Q-tail Q-invar Q-list
Map-update P-update Q-snoc G src

12 Correctness

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

($\neg$ *is-discovered src m v* $\longrightarrow$ $\neg$ *reachable* (*G.E G*) *src v*)

lemma (**in** *undirected-bfs-valid-input*) *dist-eq-dist*:
shows *Shortest-Path.dist* (*G.E G*) *u v* = *dist G u v*

lemma (**in** *undirected-bfs-valid-input*) *is-shortest-path-iff-is-shortest-dpath*:
shows *is-shortest-path* (*G.E G*) *p u v* $\longleftrightarrow$ *is-shortest-dpath G p u v*

lemma (**in** *undirected-bfs-valid-input*) *reachable-iff-reachable*:
shows *reachable* (*G.E G*) *u v* $\longleftrightarrow$ *Noschinski-to-DDFS.reachable* (*G.dE G*) *u v*

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

lemma (**in** *bfs*) *undirected-bfs-correct*:
assumes *undirected-bfs-valid-input' G src*
shows *is-shortest-path-Map G src* (*bfs G src*)

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

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

13 Algorithm

thm *init-def*

thm *DONE-def*

thm *is-discovered-def*

thm *discover-def*

thm *traverse-edge-def*

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' :: $'n \Rightarrow 'a$ *option* $\Rightarrow 'a \Rightarrow$ *bool* **where**
  P' *G uo v* $\equiv$ *case uo of None* $\Rightarrow$ *False* | *Some u* $\Rightarrow$ *P G u v*

definition *adjacency* :: $'n \Rightarrow 'n \Rightarrow ('q, 'm)$ *state* $\Rightarrow 'a \Rightarrow 'a$ *list* **where**
  adjacency G1 G2 s v $\equiv$
   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* :: $'n \Rightarrow 'n \Rightarrow 'a \Rightarrow ('q, 'm)$ *state* $\Rightarrow ('q, 'm)$ *state*
where
  alt-loop G1 G2 src s =
   (*if* $\neg$ *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* :: $'n \Rightarrow 'n \Rightarrow 'a \Rightarrow 'm$ **where**
  alt-bfs G1 G2 src $\equiv$ *parent* (*alt-loop G1 G2 src* (*init src*))

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

14 Convenience Lemmas

14.1 *P*

lemma *P-iff-mem-adjacency*:
  assumes *G.invar G*
  shows *P G u v* $\longleftrightarrow$ *v* $\in$ *set* (*G.adjacency-list G u*)

14.2 *local.adjacency*

lemma *distinct-adjacency*:
  assumes *G.invar G1*
  assumes *G.invar G2*
  shows *distinct* (*adjacency G1 G2 s v*)

lemma *adjacency-subset-V-union*:
  assumes *G.invar G1*
  assumes *G.invar G2*
  shows *set* (*adjacency G1 G2 s v*) $\subseteq$ *G.V* (*G.union G1 G2*)

15 Basic Lemmas

15.1 *alt-fold*

15.1.1 *Q-invar* ∘ *queue*

lemma *invar-queue-alt-fold*:
  assumes *G.invar G1*
  assumes *G.invar G2*
  assumes *Q-invar* (*queue s*)
  assumes ¬ *DONE s*
  shows *Q-invar* (*queue* (*alt-fold G1 G2 src s*))

15.1.2 *Q-list* ∘ *queue*

lemma *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*)))

15.1.3 *set* ∘ *Q-list* ∘ *queue*

lemma *queue-alt-fold-subset-V-union*:
  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 ¬ *DONE s*
  shows *set* (*Q-list* (*queue* (*alt-fold G1 G2 src s*))) ⊆ *G.V* (*G.union G1 G2*)

15.1.4 *state.parent*

lemma *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* $\circ$ *is-discovered src* (*parent s*)) (*adjacency G1 G2 s* (*Q-head* (*queue s*)))))

15.1.5 *P-invar* $\circ$ *state.parent*

lemma *invar-parent-alt-fold*:
assumes *G.invar G1*
assumes *G.invar G2*
assumes *P-invar* (*parent s*)
shows *P-invar* (*parent* (*alt-fold G1 G2 src s*))

15.1.6 *P.dom* $\circ$ *state.parent*

lemma *dom-parent-fold-subset-V*:
assumes *P-invar* (*parent s*)
assumes *distinct l*
assumes *P.dom* (*parent s*) $\subseteq$ *G.V G*
assumes *set l* $\subseteq$ *G.V G*
shows *P.dom* (*parent* (*List.fold* (*traverse-edge src u*) *l s*)) $\subseteq$ *G.V G*

lemma *dom-parent-alt-fold-subset-V-union*:
assumes *G.invar G1*
assumes *G.invar G2*
assumes *P-invar* (*parent s*)
assumes *P.dom* (*parent s*) $\subseteq$ *G.V* (*G.union G1 G2*)
shows *P.dom* (*parent* (*alt-fold G1 G2 src s*)) $\subseteq$ *G.V* (*G.union G1 G2*)

15.1.7 *T*

lemma *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*) $\cup$
$\{$(*Q-head* (*queue s*), *v*) $|$*v*. *v* $\in$ *set* (*adjacency G1 G2 s* (*Q-head* (*queue s*))) $\wedge$ $\neg$ *is-discovered src* (*parent s*) *v*$\}$

16 Termination

lemma *alt-loop-dom*:
  assumes $G.invar\ G1$
  assumes $G.invar\ G2$
  assumes $Q\text{-}invar\ (queue\ s)$
  assumes $P\text{-}invar\ (parent\ s)$
  assumes $set\ (Q\text{-}list\ (queue\ s)) \subseteq G.V\ (G.union\ G1\ G2)$
  assumes $P.dom\ (parent\ s) \subseteq G.V\ (G.union\ G1\ G2)$
  shows $alt\text{-}loop\text{-}dom\ (G1,\ G2,\ src,\ s)$

end

17 Invariants

17.1 Definitions

locale *alt-bfs-valid-input* = *alt-bfs* **where**
  $Map\text{-}update = Map\text{-}update$ **and**
  $P\text{-}update = P\text{-}update$ **and**
  $Q\text{-}snoc = Q\text{-}snoc$ **for**
  $Map\text{-}update :: 'a{::}linorder \Rightarrow 's \Rightarrow 'n \Rightarrow 'n$ **and**
  $P\text{-}update :: 'a \Rightarrow 'a \Rightarrow 'm \Rightarrow 'm$ **and**
  $Q\text{-}snoc :: 'q \Rightarrow 'a \Rightarrow 'q$ +
  fixes $G1\ G2 :: 'n$
  fixes $src :: 'a$
  assumes *invar-G1*: $G.invar\ G1$
  assumes *invar-G2*: $G.invar\ G2$
  assumes *G1-symmetric*: $v \in set\ (G.adjacency\text{-}list\ G1\ u) \longleftrightarrow u \in set\ (G.adjacency\text{-}list\ G1\ v)$
  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*: $src \in G.V\ G2$

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*) *G* :: *'n* **where**
G ≡ *G.union G1 G2*

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

abbreviation (**in** *alt-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** *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*

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*

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*

abbreviation (**in** *alt-bfs*) *alt-bfs-valid-input'* :: *'n* ⇒ *'n* ⇒ *'a* ⇒ *bool* **where**
alt-bfs-valid-input' G1 G2 src ≡
alt-bfs-valid-input

Map-empty Map-delete Map-lookup Map-inorder Map-inv
Set-empty Set-insert Set-delete Set-isin Set-inorder Set-inv
P-empty P-delete P-lookup P-invar
Q-empty Q-is-empty Q-head Q-tail Q-invar Q-list
Map-update P-update Q-snoc G1 G2 src

abbreviation (**in** *alt-bfs*) *alt-bfs-invar′* :: *′n* ⇒ *′n* ⇒ *′a* ⇒ (*′q*, *′m*) *state* ⇒ *bool*
where
alt-bfs-invar′ G1 G2 src s ≡
alt-bfs-invar
Map-empty Map-delete Map-lookup Map-inorder Map-inv
Set-empty Set-insert Set-delete Set-isin Set-inorder Set-inv
P-empty P-delete P-lookup P-invar
Q-empty Q-is-empty Q-head Q-tail Q-invar Q-list
Map-update G1 G2 src P-update Q-snoc s

abbreviation (**in** *alt-bfs-valid-input*) *alt-bfs-invar′′* :: (*′q*, *′m*) *state* ⇒ *bool* **where**
alt-bfs-invar′′ ≡ *alt-bfs-invar′ G1 G2 src*

abbreviation (**in** *alt-bfs*) *alt-bfs-invar-not-DONE′* :: *′n* ⇒ *′n* ⇒ *′a* ⇒ (*′q*, *′m*) *state* ⇒ *bool* **where**
alt-bfs-invar-not-DONE′ G1 G2 src s ≡
alt-bfs-invar-not-DONE
Map-empty Map-delete Map-lookup Map-inorder Map-inv
Set-empty Set-insert Set-delete Set-isin Set-inorder Set-inv
P-empty P-delete P-lookup P-invar
Q-empty Q-is-empty Q-head Q-tail Q-invar Q-list
Map-update G1 G2 src s P-update Q-snoc

abbreviation (**in** *alt-bfs-valid-input*) *alt-bfs-invar-not-DONE′′* :: (*′q*, *′m*) *state* ⇒ *bool* **where**
alt-bfs-invar-not-DONE′′ ≡ *alt-bfs-invar-not-DONE′ G1 G2 src*

abbreviation (**in** *alt-bfs*) *alt-bfs-invar-DONE′* :: *′n* ⇒ *′n* ⇒ *′a* ⇒ (*′q*, *′m*) *state* ⇒ *bool* **where**
alt-bfs-invar-DONE′ G1 G2 src s ≡
alt-bfs-invar-DONE
Map-empty Map-delete Map-lookup Map-inorder Map-inv
Set-empty Set-insert Set-delete Set-isin Set-inorder Set-inv
P-empty P-delete P-lookup P-invar
Q-empty Q-is-empty Q-head Q-tail Q-invar Q-list
Map-update G1 G2 src s P-update Q-snoc

abbreviation (**in** *alt-bfs-valid-input*) *alt-bfs-invar-DONE′′* :: (*′q*, *′m*) *state* ⇒ *bool*
where
alt-bfs-invar-DONE′′ ≡ *alt-bfs-invar-DONE′ G1 G2 src*

17.2 Convenience Lemmas

17.2.1 *alt-bfs*

lemma (**in** *alt-bfs*) *alt-bfs-invar-not-DONE'I*:
assumes *alt-bfs-invar' G1 G2 src s*
assumes $\neg$ *DONE s*
shows *alt-bfs-invar-not-DONE' G1 G2 src s*

lemma (**in** *alt-bfs*) *alt-bfs-invar-DONE'I*:
assumes *alt-bfs-invar' G1 G2 src s*
assumes *DONE s*
shows *alt-bfs-invar-DONE' G1 G2 src s*

17.2.2 *alt-bfs-valid-input*

lemma (**in** *alt-bfs-valid-input*) *vertex-color-induct* [*case-names white gray black*]:
assumes *white s v* $\Longrightarrow$ *Q' s v*
assumes *gray s v* $\Longrightarrow$ *Q' s v*
assumes *black s v* $\Longrightarrow$ *Q' s v*
shows *Q' s v*

lemma (**in** *alt-bfs-valid-input*) *Q-P''-cong*:
assumes *P' G2* (*P-lookup* (*parent s*) *v*) *v*
shows
$Q\ s\ v = (Not \circ P'')$
$(Not \circ Q\ s\ v) = P''$

lemma (**in** *alt-bfs-valid-input*) *Q-P''-cong-2*:
assumes $\neg$ *P' G2* (*P-lookup* (*parent s*) *v*) *v*
shows
$Q\ s\ v = P''$
$(Not \circ Q\ s\ v) = (Not \circ P'')$

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

lemma (**in** *alt-bfs-valid-input*) *mem-E-if-mem-E1*:
assumes $e \in$ *G.E G1*
shows $e \in$ *G.E G*

lemma (**in** *alt-bfs-valid-input*) *mem-E-if-mem-E2*:
assumes $e \in$ *G.E G2*
shows $e \in$ *G.E G*

lemma (**in** *alt-bfs-valid-input*) *mem-E1-iff-not-mem-E2*:
assumes $e \in$ *G.E G*
shows $e \notin$ *G.E G1* $=$ *P'' e*

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

context *alt-bfs-valid-input*
begin

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

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

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

end

lemma (**in** *alt-bfs-valid-input*) *P-P''-cong*:
  shows $P\ G2\ u\ v \longleftrightarrow P''\ \{u, v\}$

lemma (**in** *alt-bfs-valid-input*) *mem-adjacency-imp-alt*:
  assumes $v \in set\ (adjacency\ G1\ G2\ s\ u)$
  shows $alt\ s\ u\ v$

lemma (**in** *alt-bfs-valid-input*) *mem-adjacency-imp-edge*:
  assumes $v \in set\ (adjacency\ G1\ G2\ s\ u)$
  shows $\{u, v\} \in G.E\ G$

lemma (**in** *alt-bfs-valid-input*) *mem-adjacency-if-edge*:
  assumes $alt\ s\ u\ v$
  assumes $\{u, v\} \in G.E\ G$
  assumes $\neg\ white\ s\ u$
  shows $v \in set\ (adjacency\ G1\ G2\ s\ u)$

lemma (**in** *alt-bfs-valid-input*) *src-not-white*:
  shows $\neg\ white\ s\ src$

17.3 Basic Lemmas

17.3.1 *alt-bfs-valid-input*

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

lemma (**in** *alt-bfs-valid-input*) *P'E*:
  assumes $P'\ G2\ (P\text{-}lookup\ (parent\ s)\ v)\ v$
  obtains u **where**
    $P\text{-}lookup\ (parent\ s)\ v = Some\ u$
    $P''\ \{u, v\}$

17.3.2 *alt-bfs-invar*

lemma (**in** *alt-bfs-invar*) *rev-follow*:
shows
rev-follow (*parent s*) $v \neq []$
last (*rev-follow* (*parent s*) v) $= v$

lemma (**in** *alt-bfs-invar*) *parent-rev-followE*:
assumes *P-lookup* (*parent s*) $v =$ *Some u*
obtains p **where** *rev-follow* (*parent s*) $v = p$ @ $[u, v]$

lemma (**in** *alt-bfs-invar*) *parent-imp-rev-follow*:
assumes *P-lookup* (*parent s*) $v =$ *Some u*
shows *rev-follow* (*parent s*) $v =$ *rev-follow* (*parent s*) u @ $[v]$

lemma (**in** *alt-bfs-invar*) *not-P'E*:
assumes $\neg$ P' *G2* (*P-lookup* (*parent s*) v) v
assumes $v \neq$ *src*
assumes $\neg$ *white s v*
obtains u **where**
P-lookup (*parent s*) $v =$ *Some u*
$\neg$ P'' $\{u, v\}$

lemma (**in** *alt-bfs-invar*) *not-P'D*:
assumes $\neg$ P' *G2* (*P-lookup* (*parent s*) v) v
assumes $v \neq$ *src*
assumes $\neg$ *white s v*
shows
edges-of-path (*rev-follow* (*parent s*) v) $\neq []$
$\neg$ P'' (*last* (*edges-of-path* (*rev-follow* (*parent s*) v)))

lemma (**in** *alt-bfs-invar*) *P'-P''-cong*:
shows P' *G2* (*P-lookup* (*parent s*) v) $v \longleftrightarrow$ *edges-of-path* (*rev-follow* (*parent s*) v) $\neq []$ $\wedge$ P'' (*last* (*edges-of-path* (*rev-follow* (*parent s*) v)))

lemma (**in** *alt-bfs-invar*) *alt-path-rev-follow-src*:
shows *alt-path* P'' (*Not* $\circ$ P'') (*G.E G*) (*rev-follow* (*parent s*) *src*) *src src*

lemma (**in** *alt-bfs-invar*) *alt-path-rev-follow-snocI*:
assumes *alt-path* P'' (*Not* $\circ$ P'') (*G.E G*) (*rev-follow* (*parent s*) u) *src u*
assumes $\{u, v\} \in$ *G.E G*
assumes *alt s u v*
assumes $\neg$ *white s u*
shows *alt-path* P'' (*Not* $\circ$ P'') (*G.E G*) (*rev-follow* (*parent s*) u @ $[v]$) *src v*

lemma (**in** *alt-bfs-invar*) *not-white-imp-alt-path-rev-follow*:
assumes $\neg$ *white s v*
shows *alt-path* P'' (*Not* $\circ$ P'') (*G.E G*) (*rev-follow* (*parent s*) v) *src v*

lemma (**in** *alt-bfs-invar*) *hd-rev-follow-eq-src*:

assumes $\neg$ *white s v*
shows *hd* (*rev-follow* (*parent s*) *v*) = *src*

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

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

lemma (**in** *alt-bfs-invar*) *mem-queue-imp-d-ge*:
assumes *v* $\in$ *set* (*Q-list* (*queue s*))
shows *d* (*parent s*) (*Q-head* (*queue s*)) $\leq$ *d* (*parent s*) *v*

lemma (**in** *alt-bfs-invar*) *mem-queue-imp-d-le*:
assumes *v* $\in$ *set* (*Q-list* (*queue s*))
shows *d* (*parent s*) *v* $\leq$ *d* (*parent s*) (*last* (*Q-list* (*queue s*)))

lemma (**in** *alt-bfs-invar*) *d-triangle-inequality-edge*:
assumes {*u*, *v*} $\in$ *G.E G*
assumes *alt s u v*
assumes $\neg$ *white s u*
assumes $\neg$ *white s v*
shows *d* (*parent s*) *v* $\leq$ *d* (*parent s*) *u* + *1*

17.4 *bfs.init*

17.4.1

lemma (**in** *alt-bfs-valid-input*) *follow-invar-parent-init*:
shows *follow-invar* (*P-lookup* (*parent* (*init src*)))

17.4.2 *alt-bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G1.0 ?G2.0 ?src ?P-update ?Q-snoc ?s* $\Longrightarrow$ *?Q-invar* (*queue ?s*)

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

17.4.3 *alt-bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G1.0 ?G2.0 ?src ?P-update ?Q-snoc ?s* $\Longrightarrow$ *?P-invar* (*state.parent ?s*)

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

17.4.4 *alt-bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G1.0 ?G2.0 ?src ?P-update ?Q-snoc ?s* $\Longrightarrow$ *?P-lookup* (*state.parent ?s*) *?src* = *None*

lemma (**in** *alt-bfs-valid-input*) *parent-src-init*:
 shows *P-lookup* (*parent* (*init src*)) *src* = *None*

17.4.5 $[\![$*alt-bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G1.0 ?G2.0 ?src ?P-update ?Q-snoc ?s*; *?P-lookup* (*state.parent ?s*) *?v* = *Some ?u*$]\!]$ $\Longrightarrow$ *alt-bfs.P′ ?Map-lookup ?Set-isin ?G2.0* (*?P-lookup* (*state.parent ?s*) *?u*) *?u* = ($\neg$ *alt-bfs.P ?Map-lookup ?Set-isin ?G2.0 ?u ?v*)

lemma (**in** *alt-bfs-valid-input*) *parent-imp-alt-init*:
 assumes *P-lookup* (*parent* (*init src*)) *v* = *Some u*
 shows *alt* (*init src*) *u v*

17.4.6 $[\![$*alt-bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G1.0 ?G2.0 ?src ?P-update ?Q-snoc ?s*; *?P-lookup* (*state.parent ?s*) *?v* = *Some ?u*$]\!]$ $\Longrightarrow$ {*?u*, *?v*} $\in$ *adjacency.E ?Map-lookup ?Set-inorder* (*adjacency.union ?Map-update ?Map-lookup ?Map-inorder ?Set-insert ?Set-inorder ?G1.0 ?G2.0*)

lemma (**in** *alt-bfs-valid-input*) *parent-imp-edge-init*:
 assumes *P-lookup* (*parent* (*init src*)) *v* = *Some u*
 shows {*u*, *v*} $\in$ *G.E G*

17.4.7 $\llbracket$*alt-bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G1.0 ?G2.0 ?src ?P-update ?Q-snoc ?s*; *?v* $\in$ *set* (*?Q-list* (*queue ?s*))$\rrbracket$ $\Longrightarrow$ $\neg$ $\neg$ *bfs.is-discovered ?P-lookup ?src* (*state.parent ?s*) *?v*

lemma (**in** *alt-bfs-valid-input*) *not-white-if-mem-queue-init*:
assumes *v* $\in$ *set* (*Q-list* (*queue* (*init src*)))
shows $\neg$ *white* (*init src*) *v*

17.4.8 $\llbracket$*alt-bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G1.0 ?G2.0 ?src ?P-update ?Q-snoc ?s*; *?P-lookup* (*state.parent ?s*) *?v* = *Some ?u*$\rrbracket$ $\Longrightarrow$ $\neg$ $\neg$ *bfs.is-discovered ?P-lookup ?src* (*state.parent ?s*) *?u*

lemma (**in** *alt-bfs-valid-input*) *not-white-if-parent-init*:
assumes *P-lookup* (*parent* (*init src*)) *v* = *Some u*
shows $\neg$ *white* (*init src*) *u*

17.4.9 $\llbracket$*alt-bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G1.0 ?G2.0 ?src ?P-update ?Q-snoc ?s*; *alt-bfs.P′ ?Map-lookup ?Set-isin ?G2.0* (*?P-lookup* (*state.parent ?s*) *?u*) *?u* = ($\neg$ *alt-bfs.P ?Map-lookup ?Set-isin ?G2.0 ?u ?v*); {*?u*, *?v*} $\in$ *adjacency.E ?Map-lookup ?Set-inorder* (*adjacency.union ?Map-update ?Map-lookup ?Map-inorder ?Set-insert ?Set-inorder ?G1.0 ?G2.0*); *bfs.is-discovered ?P-lookup ?src* (*state.parent ?s*) *?u* $\wedge$ *?u* $\notin$ *set* (*?Q-list* (*queue ?s*))$\rrbracket$ $\Longrightarrow$ $\neg$ $\neg$ *bfs.is-discovered ?P-lookup ?src* (*state.parent ?s*) *?v*

lemma (**in** *alt-bfs-valid-input*) *black-imp-adjacency-not-white-init*:
assumes *alt* (*init src*) *u v*
assumes {*u*, *v*} $\in$ *G.E G*
assumes *black* (*init src*) *u*
shows $\neg$ *white s v*

17.4.10 *alt-bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G1.0 ?G2.0 ?src ?P-update ?Q-snoc ?s* $\Longrightarrow$ *sorted-wrt* ($\lambda u\ v.$ *path-length* (*rev* (*parent.follow* (*?P-lookup* (*state.parent ?s*)) *u*)) $\leq$ *path-length* (*rev* (*parent.follow* (*?P-lookup* (*state.parent ?s*)) *v*))) (*?Q-list* (*queue ?s*))

lemma (**in** *alt-bfs-valid-input*) *queue-sorted-wrt-d-init*:
shows *sorted-wrt* ($\lambda u\ v.$ *d* (*parent* (*init src*)) *u* $\leq$ *d* (*parent* (*init src*)) *v*) (*Q-list* (*queue* (*init src*)))

17.4.11 ⟦*alt-bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G1.0 ?G2.0 ?src ?P-update ?Q-snoc ?s*; $\neg$ *?Q-is-empty* (*queue ?s*)⟧ $\Longrightarrow$ *path-length* (*rev* (*parent.follow* (*?P-lookup* (*state.parent ?s*)) (*last* (*?Q-list* (*queue ?s*))))) $\leq$ *path-length* (*rev* (*parent.follow* (*?P-lookup* (*state.parent ?s*)) (*?Q-head* (*queue ?s*)))) + *1*

lemma (**in** *alt-bfs-valid-input*) *d-last-queue-le-init*:
assumes $\neg$ *Q-is-empty* (*queue* (*init src*))
shows *d* (*parent* (*init src*)) (*last* (*Q-list* (*queue* (*init src*)))) $\leq$ *d* (*parent* (*init src*)) (*Q-head* (*queue* (*init src*))) + *1*

17.4.12 $\llbracket$*alt-bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G1.0 ?G2.0 ?src ?P-update ?Q-snoc ?s*; *Alternating-Path.alt-path* (*if alt-bfs.P′ ?Map-lookup ?Set-isin ?G2.0* (*?P-lookup* (*state.parent ?s*) *?u*) *?u then Not* ∘ (λ*e. e* ∈ *adjacency.E ?Map-lookup ?Set-inorder ?G2.0*) *else* (λ*e. e* ∈ *adjacency.E ?Map-lookup ?Set-inorder ?G2.0*)) (*Not* ∘ (*if alt-bfs.P′ ?Map-lookup ?Set-isin ?G2.0* (*?P-lookup* (*state.parent ?s*) *?u*) *?u then Not* ∘ (λ*e. e* ∈ *adjacency.E ?Map-lookup ?Set-inorder ?G2.0*) *else* (λ*e. e* ∈ *adjacency.E ?Map-lookup ?Set-inorder ?G2.0*))) (*adjacency.E ?Map-lookup ?Set-inorder* (*adjacency.union ?Map-update ?Map-lookup ?Map-inorder ?Set-insert ?Set-inorder ?G1.0 ?G2.0*)) *?p ?u ?v*; ¬ ¬ *bfs.is-discovered ?P-lookup ?src* (*state.parent ?s*) *?u*; ¬ ¬ *bfs.is-discovered ?P-lookup ?src* (*state.parent ?s*) *?v*$\rrbracket \Longrightarrow$ *path-length* (*rev* (*parent.follow* (*?P-lookup* (*state.parent ?s*)) *?v*)) ≤ *path-length* (*rev* (*parent.follow* (*?P-lookup* (*state.parent ?s*)) *?u*)) + *path-length ?p*

lemma (**in** *alt-bfs-valid-input*) *d-triangle-inequality-init*:
 assumes *alt-path* (*Q* (*init src*) *u*) (*Not* ∘ *Q* (*init src*) *u*) (*G.E G*) *p u v*
 assumes ¬ *white* (*init src*) *u*
 assumes ¬ *white* (*init src*) *v*
 shows *d* (*parent* (*init src*)) *v* ≤ *d* (*parent* (*init src*)) *u* + *path-length p*

17.4.13

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

17.5 λ*Map-lookup Set-isin Set-inorder P-lookup Q-head Q-tail P-update Q-snoc G1 G2 src s. fold* (*bfs.traverse-edge P-update P-lookup Q-snoc src* (*Q-head* (*queue s*))) (*alt-bfs.adjacency Map-lookup Set-isin Set-inorder P-lookup G1 G2 s* (*Q-head* (*queue s*))) (*s*⦇*queue* := *Q-tail* (*queue s*)⦈)

17.5.1 Convenience Lemmas

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*) *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-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*}

lemma (**in** *alt-bfs-invar-not-DONE*) *list-queue-non-empty*:
shows *Q-list* (*queue s*) ≠ []

lemma (**in** *alt-bfs-invar-not-DONE*) *head-queue-mem-queue*:
shows *Q-head* (*queue s*) ∈ *set* (*Q-list* (*queue s*))

lemma (**in** *alt-bfs-invar-not-DONE*) *not-white-head-queue*:
shows ¬ *white s* (*Q-head* (*queue s*))

17.5.2

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*) *parent-alt-fold*:
shows *Parent-Relation.parent* (*P-lookup* (*parent* (*alt-fold G1 G2 src s*)))

17.5.3 *alt-bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G1.0 ?G2.0 ?src ?P-update ?Q-snoc ?s* ⟹ *?Q-invar* (*queue ?s*)

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

17.5.4 *alt-bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G1.0 ?G2.0 ?src ?P-update ?Q-snoc ?s* $\Longrightarrow$ *?P-invar* (*state.parent ?s*)

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

17.5.5 *bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G ?src ?P-update ?Q-snoc ?s* $\Longrightarrow$ *?P-lookup* (*state.parent ?s*) *?src* = *None*

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

17.5.6 Basic Lemmas

lemma (**in** *alt-bfs-invar-not-DONE*) *head-queueI*:
  assumes $v \in$ *set* (*Q-list* (*queue s*))
  assumes $v \notin$ *set* (*Q-list* (*queue* (*alt-fold G1 G2 src s*)))
  shows v = *Q-head* (*queue s*)

lemma (**in** *alt-bfs-invar*) *head-queueI-2*:
  assumes *P-lookup* (*parent s*) $v \neq$ *Some u*
  assumes *P-lookup* (*parent* (*alt-fold G1 G2 src s*)) v = *Some u*
  shows u = *Q-head* (*queue s*)

lemma (**in** *alt-bfs-invar-not-DONE*) *whiteD*:
  assumes *white s v*
  shows $\neg$ *black* (*alt-fold G1 G2 src s*) *v*

lemma (**in** *alt-bfs-invar*) *whiteI*:
  assumes *P-lookup* (*parent s*) $v \neq$ *Some u*
  assumes *P-lookup* (*parent* (*alt-fold G1 G2 src s*)) v = *Some u*
  shows *white s v*

lemma (**in** *alt-bfs-invar*) *not-white-imp-not-white-alt-fold*:
  assumes $\neg$ *white s v*
  shows $\neg$ *white* (*alt-fold G1 G2 src s*) *v*

lemma (**in** *alt-bfs-invar*) *not-white-imp-lookup-parent-alt-fold-eq-lookup-parent*:
  assumes $\neg$ *white s v*
  shows *P-lookup* (*parent* (*alt-fold G1 G2 src s*)) v = *P-lookup* (*parent s*) *v*

lemma (**in** *alt-bfs-invar-not-DONE*) *not-white-imp-rev-follow-alt-fold-eq-rev-follow*:
assumes ¬ *white s v*
shows *rev-follow* (*parent* (*alt-fold G1 G2 src s*)) *v* = *rev-follow* (*parent s*) *v*

lemmas (**in** *alt-bfs-invar-not-DONE*) *not-* =
not-white-imp-not-white-alt-fold
not-white-imp-lookup-parent-alt-fold-eq-lookup-parent
not-white-imp-rev-follow-alt-fold-eq-rev-follow

lemma (**in** *alt-bfs-invar-not-DONE*) *mem-filterD*:
assumes *v* ∈ *set* (*filter* (*Not* ∘ *is-discovered src* (*parent s*)) (*adjacency G1 G2 s* (*Q-head* (*queue s*))))
shows
d (*parent* (*alt-fold G1 G2 src s*)) *v* = *d* (*parent* (*alt-fold G1 G2 src s*)) (*Q-head* (*queue s*)) + *1*
d (*parent* (*alt-fold G1 G2 src s*)) (*last* (*Q-list* (*queue s*))) ≤ *d* (*parent* (*alt-fold G1 G2 src s*)) *v*

lemma (**in** *alt-bfs-invar*) *white-not-white-alt-foldD*:
assumes *white s v*
assumes ¬ *white* (*alt-fold G1 G2 src s*) *v*
shows
v ∈ *set* (*adjacency G1 G2 s* (*Q-head* (*queue s*)))
P-lookup (*parent* (*alt-fold G1 G2 src s*)) *v* = *Some* (*Q-head* (*queue s*))

lemma (**in** *alt-bfs-invar-not-DONE*) *white-not-white-alt-foldD-2*:
assumes *white s v*
assumes ¬ *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*)) (*Q-head* (*queue s*)) + *1*

lemmas (**in** *alt-bfs-invar-not-DONE*) *white-not-white-alt-foldD* =
white-not-white-alt-foldD
white-not-white-alt-foldD-2

17.5.7 ⟦*alt-bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G1.0 ?G2.0 ?src ?P-update ?Q-snoc ?s*; *?P-lookup* (*state.parent ?s*) *?v* = *Some ?u*⟧ ⟹ *alt-bfs.P′ ?Map-lookup ?Set-isin ?G2.0* (*?P-lookup* (*state.parent ?s*) *?u*) *?u* = (¬ *alt-bfs.P ?Map-lookup ?Set-isin ?G2.0 ?u ?v*)

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*

17.5.8 $\llbracket$*alt-bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G1.0 ?G2.0 ?src ?P-update ?Q-snoc ?s*; *?P-lookup* (*state.parent ?s*) *?v* = *Some ?u*$\rrbracket$ $\Longrightarrow$ {*?u*, *?v*} $\in$ *adjacency.E ?Map-lookup ?Set-inorder* (*adjacency.union ?Map-update ?Map-lookup ?Map-inorder ?Set-insert ?Set-inorder ?G1.0 ?G2.0*)

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*} $\in$ *G.E G*

17.5.9 $\llbracket$*alt-bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G1.0 ?G2.0 ?src ?P-update ?Q-snoc ?s*; *?v* $\in$ *set* (*?Q-list* (*queue ?s*))$\rrbracket$ $\Longrightarrow$ $\neg$ $\neg$ *bfs.is-discovered ?P-lookup ?src* (*state.parent ?s*) *?v*

lemma (**in** *alt-bfs-invar-not-DONE*) *not-white-if-mem-queue-alt-fold*:
assumes *v* $\in$ *set* (*Q-list* (*queue* (*alt-fold G1 G2 src s*)))
shows $\neg$ *white* (*alt-fold G1 G2 src s*) *v*

17.5.10 $\llbracket$*alt-bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G1.0 ?G2.0 ?src ?P-update ?Q-snoc ?s*; *?P-lookup* (*state.parent ?s*) *?v* = *Some ?u*$\rrbracket$ $\Longrightarrow$ $\neg$ $\neg$ *bfs.is-discovered ?P-lookup ?src* (*state.parent ?s*) *?u*

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

17.5.11 $\llbracket$*alt-bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G1.0 ?G2.0 ?src ?P-update ?Q-snoc ?s*; *alt-bfs.P′ ?Map-lookup ?Set-isin ?G2.0* (*?P-lookup* (*state.parent ?s*) *?u*) *?u* = ($\neg$ *alt-bfs.P ?Map-lookup ?Set-isin ?G2.0 ?u ?v*); {*?u*, *?v*} $\in$ *adjacency.E ?Map-lookup ?Set-inorder* (*adjacency.union ?Map-update ?Map-lookup ?Map-inorder ?Set-insert ?Set-inorder ?G1.0 ?G2.0*); *bfs.is-discovered ?P-lookup ?src* (*state.parent ?s*) *?u* $\wedge$ *?u* $\notin$ *set* (*?Q-list* (*queue ?s*))$\rrbracket$ $\Longrightarrow$ $\neg$ $\neg$ *bfs.is-discovered ?P-lookup ?src* (*state.parent ?s*) *?v*

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*} $\in$ *G.E G*
assumes *black* (*alt-fold G1 G2 src s*) *u*
shows $\neg$ *white* (*alt-fold G1 G2 src s*) *v*

17.5.12 *alt-bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G1.0 ?G2.0 ?src ?P-update ?Q-snoc ?s* $\Longrightarrow$ *sorted-wrt* (λ*u v*. *path-length* (*rev* (*parent.follow* (*?P-lookup* (*state.parent ?s*)) *u*)) $\leq$ *path-length* (*rev* (*parent.follow* (*?P-lookup* (*state.parent ?s*)) *v*))) (*?Q-list* (*queue ?s*))

lemma (**in** *alt-bfs-invar-not-DONE*) *queue-sorted-wrt-d-alt-fold-aux*:
assumes *u-mem-tail-queue*: *u* $\in$ *set* (*Q-list* (*Q-tail* (*queue s*)))
assumes *v-mem-filter*: *v* $\in$ *set* (*filter* (*Not* $\circ$ *is-discovered src* (*parent s*)) (*adjacency G1 G2 s* (*Q-head* (*queue s*))))
shows *d* (*parent* (*alt-fold G1 G2 src s*)) *u* $\leq$ *d* (*parent* (*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* $\leq$ *d* (*parent* (*alt-fold G1 G2 src s*)) *v*) (*Q-list* (*queue* (*alt-fold G1 G2 src s*)))

17.5.13 ⟦*alt-bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G1.0 ?G2.0 ?src ?P-update ?Q-snoc ?s*; ¬ *?Q-is-empty* (*queue ?s*)⟧ ⟹ *path-length* (*rev* (*parent.follow* (*?P-lookup* (*state.parent ?s*)) (*last* (*?Q-list* (*queue ?s*))))) ≤ *path-length* (*rev* (*parent.follow* (*?P-lookup* (*state.parent ?s*)) (*?Q-head* (*queue ?s*)))) + *1*

lemma (**in** *alt-bfs-invar-not-DONE*) *d-last-queue-le-alt-fold-aux*:
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 s*)) + *1*

lemma (**in** *alt-bfs-invar-not-DONE*) *d-last-queue-le-alt-fold-aux-2*:
assumes ¬ *Q-is-empty* (*queue* (*alt-fold G1 G2 src s*))
shows *d* (*parent* (*alt-fold G1 G2 src s*)) (*Q-head* (*queue s*)) ≤ *d* (*parent* (*alt-fold G1 G2 src s*)) (*Q-head* (*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*

17.5.14 ⟦*alt-bfs-invar ?Map-empty ?Map-delete ?Map-lookup ?Map-inorder ?Map-inv ?Set-empty ?Set-insert ?Set-delete ?Set-isin ?Set-inorder ?Set-inv ?P-empty ?P-delete ?P-lookup ?P-invar ?Q-empty ?Q-is-empty ?Q-head ?Q-tail ?Q-invar ?Q-list ?Map-update ?G1.0 ?G2.0 ?src ?P-update ?Q-snoc ?s*; *Alternating-Path.alt-path* (*if alt-bfs.P′ ?Map-lookup ?Set-isin ?G2.0* (*?P-lookup* (*state.parent ?s*) *?u*) *?u then Not* ∘ (*λe. e* ∈ *adjacency.E ?Map-lookup ?Set-inorder ?G2.0*) *else* (*λe. e* ∈ *adjacency.E ?Map-lookup ?Set-inorder ?G2.0*)) (*Not* ∘ (*if alt-bfs.P′ ?Map-lookup ?Set-isin ?G2.0* (*?P-lookup* (*state.parent ?s*) *?u*) *?u then Not* ∘ (*λe. e* ∈ *adjacency.E ?Map-lookup ?Set-inorder ?G2.0*) *else* (*λe. e* ∈ *adjacency.E ?Map-lookup ?Set-inorder ?G2.0*))) (*adjacency.E ?Map-lookup ?Set-inorder* (*adjacency.union ?Map-update ?Map-lookup ?Map-inorder ?Set-insert ?Set-inorder ?G1.0 ?G2.0*)) *?p ?u ?v*; ¬ ¬ *bfs.is-discovered ?P-lookup ?src* (*state.parent ?s*) *?u*; ¬ ¬ *bfs.is-discovered ?P-lookup ?src* (*state.parent ?s*) *?v*⟧ ⟹ *path-length* (*rev* (*parent.follow* (*?P-lookup* (*state.parent ?s*)) *?v*)) ≤ *path-length* (*rev* (*parent.follow* (*?P-lookup* (*state.parent ?s*)) *?u*)) + *path-length ?p*

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-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*

17.5.15

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

18 *alt-bfs.alt-loop*

18.1 Convenience Lemmas

lemma (**in** *alt-bfs-invar*) *queue-subset-V*:
shows *set* (*Q-list* (*queue s*)) ⊆ *G.V G*

lemma (**in** *alt-bfs-invar*) *dom-parent-subset-V*:
shows *P.dom* (*parent s*) ⊆ *G.V G*

lemma (**in** *alt-bfs-invar*) *alt-loop-dom*:
shows *alt-loop-dom* (*G1*, *G2*, *src*, *s*)

lemma (**in** *alt-bfs*) *alt-loop-psimps*:
assumes *alt-bfs-invar' G1 G2 src s*
shows *alt-loop G1 G2 src s* = (*if* ¬ *DONE s then alt-loop G1 G2 src* (*alt-fold G1 G2 src s*) *else s*)

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

lemma (**in** *alt-bfs-invar-DONE*) *alt-loop-psimps*:
shows *alt-loop G1 G2 src s* = *s*

lemma (**in** *alt-bfs*) *alt-bfs-induct*:
assumes *alt-bfs-invar' G1 G2 src s*
assumes ⋀*G1 G2 src s*. (¬ *DONE s* ⟹ *Q G1 G2 src* (*alt-fold G1 G2 src s*)) ⟹ *Q G1 G2 src s*

shows *Q G1 G2 src s*

18.2

lemma (**in** *alt-bfs-invar*) *alt-bfs-invar-alt-loop*:
shows *alt-bfs-invar''* (*alt-loop G1 G2 src s*)

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

lemma (**in** *alt-bfs-valid-input*) *alt-bfs-invar-alt-loop-init*:
shows *alt-bfs-invar''* (*alt-loop G1 G2 src* (*init src*))

lemma (**in** *alt-bfs*) *alt-bfs-invar-alt-loop-init*:
assumes *alt-bfs-valid-input' G1 G2 src*
shows *alt-bfs-invar' G1 G2 src* (*alt-loop G1 G2 src* (*init src*))

19 Correctness

19.1 Completeness

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-valid-input*) *completeness*:
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*

19.2 Soundness

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*) *soundness*:
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*

19.3 Correctness

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*))

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

corollary (**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*)

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

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* ($\lambda e.\ e \in G.E\ G2$) (*G.union G1 G2*) *src* (*alt-bfs-partial G1 G2 src*)

end
theory *BFS-Partial*
  imports
    BFS
begin

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* :: $'n \Rightarrow 'a \Rightarrow 'm$ **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

20

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

type-synonym $'a\ queue = 'a\ list \times 'a\ list$

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

fun *is-empty* :: $'a\ queue \Rightarrow bool$ **where**
  $is\text{-}empty\ (f, \text{-}) \longleftrightarrow f = []$

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

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

fun *head* :: $'a\ queue \Rightarrow 'a$ **where**
  $head\ (x \# f, \text{-}) = x$

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

fun *invar* :: $'a\ queue \Rightarrow bool$ **where**
  $invar\ ([], r) \longleftrightarrow r = []\ |$
  $invar\ (f, r) = True$

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

20.1 Functional correctness

lemma *list-empty*:
  shows $list\ empty = []$

lemma *is-empty*:
  assumes $invar\ q$
  shows $is\text{-}empty\ q \longleftrightarrow list\ q = []$

lemma *list-snoc*:
  assumes $invar\ q$
  shows $list\ (snoc\ q\ x) = list\ q\ @\ [x]$

lemma *list-non-emptyE*:
  assumes $invar\ q$
  assumes $list\ q \neq []$
  obtains $x\ f\ r$ **where**

$q = (x \# f, r)$

lemma *list-head*:
assumes *invar q*
assumes *list q* $\neq$ []
shows *head q* = *hd* (*list q*)

lemma *list-tail*:
assumes *invar q*
assumes *list q* $\neq$ []
shows *list* (*tail q*) = *tl* (*list q*)

lemma *invar-empty*:
shows *invar empty*

lemma *invar-snoc*:
assumes *invar q*
shows *invar* (*snoc q x*)

lemma *invar-if-r-empty*:
assumes *r* = []
shows *invar* (*f*, *r*)

lemma *invar-tail*:
assumes *invar q*
assumes *list q* $\neq$ []
shows *invar* (*tail q*)

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

20.2 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

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*]
thm *B.loop-partial.simps*
value *bfs-partial* (*update* (*1*::*nat*) (*RBT-Set.insert* (*2*::*nat*) *empty*) *empty*) *1*

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

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*]
thm *A.alt-loop-partial.simps*
value *alt-bfs-partial* (*update* (*4*::*nat*) (*RBT-Set.insert* (*3*::*nat*) *empty*) (*update* (*3*::*nat*) (*RBT-Set.insert* (*4*::*nat*) *empty*) *empty*)) (*update* (*2*::*nat*) (*RBT-Set.insert* (*1*::*nat*) *empty*) (*update* (*1*::*nat*) (*RBT-Set.insert* (*2*::*nat*) *empty*) *empty*)) *1*

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

In graph theory, 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* ≡ *2* ≤ *length ?p* ∧ *Berge.alt-path ?M ?p* ∧ *hd ?p* ∉ *Vs ?M* ∧ *last ?p* ∉ *Vs ?M*, and *augpath*. We show that we can reverse augmenting paths.

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 $\neg\ (\exists c.\ path\ G\ c \wedge odd\text{-}cycle\ c)$

end

21 Edmonds-Karp 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.

The algorithm is an adaptation of the Edmonds-Karp algorithm, which solves the maximum flow problem, to the maximum cardinality matching problem in bipartite graphs, which reduces to the maximum flow problem.

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

21.1 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* $\Rightarrow$ $'s \Rightarrow$ $'n \Rightarrow$ $'n$ **and**
  P-update :: $'a \Rightarrow$ $'a \Rightarrow$ $'m \Rightarrow$ $'m$ **and**
  M-empty **and**
  M-update :: $'a \Rightarrow$ $'a \Rightarrow$ $'m \Rightarrow$ $'m$ **and**
  M-delete **and**
  M-lookup **and**
  M-inorder **and**
  M-inv
begin

definition *is-free-vertex* :: $'m \Rightarrow$ $'a \Rightarrow$ *bool* **where**
  is-free-vertex M v $\equiv$ *M-lookup M v* = *None*

definition *free-vertices* :: $'s \Rightarrow 'm \Rightarrow 'a$ *list* **where**
free-vertices V M $\equiv$ *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, 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 @ $[t]$ 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* :: $'m \Rightarrow 'n$ **where**
G2-1 M $\equiv$ *List.fold G.insert* (*M-inorder M*) *Map-empty*

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

definition *G2-2* :: $'s \Rightarrow 'a \Rightarrow 'm \Rightarrow 'n$ **where**
G2-2 L s M $\equiv$ *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* :: $'s \Rightarrow 's \Rightarrow 'a \Rightarrow 'a \Rightarrow 'm \Rightarrow 'n$ **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* :: $'s \Rightarrow 's \Rightarrow 'a \Rightarrow 'a \Rightarrow 'm \Rightarrow 'n$ **where**
G2 ≡ *G2-3*

definition *G1* :: $'n \Rightarrow 'n \Rightarrow 'n$ **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* :: $'s \Rightarrow 's \Rightarrow 'm \Rightarrow$ *bool* **where**
done-1 L R M ≡ *free-vertices L M* = [] ∨ *free-vertices R M* = []

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

fun *augment* :: $'m \Rightarrow 'a$ *path* $\Rightarrow 'm$ **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* :: $'n \Rightarrow 's \Rightarrow 's \Rightarrow 'a \Rightarrow 'a \Rightarrow 'm$ **where**
edmonds-karp G L R s t ≡ *loop′ G L R s t M-empty*

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

abbreviation *M-tbd* :: $'m \Rightarrow 'a$ *graph* **where**
M-tbd M $\equiv \{\{u, v\} \mid u\ v.$ *M-lookup M u* $=$ *Some v*$\}$

abbreviation *P-tbd* :: $'a$ *path* $\Rightarrow 'a$ *graph* **where**
P-tbd p $\equiv$ *set* (*edges-of-path p*)

abbreviation *is-symmetric-Map* :: $'m \Rightarrow$ *bool* **where**
is-symmetric-Map M $\equiv \forall\, u\ v.$ *M-lookup M u* $=$ *Some v* $\longleftrightarrow$ *M-lookup M v* $=$ *Some u*

end

21.2 Verification of the correctness of the algorithm

21.2.1 Assumptions on the input

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

- (L, 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* $\Rightarrow 's \Rightarrow 'n \Rightarrow 'n$ **and**
P-update :: $'a \Rightarrow 'a \Rightarrow 'm \Rightarrow 'm$ **and**
M-update :: $'a \Rightarrow 'a \Rightarrow 'm \Rightarrow 'm$ +
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*: $s \notin G.V\ G$
assumes *t-not-mem-V*: $t \notin G.V\ G$
assumes *s-neq-t*: $s \neq t$

As was 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.

21.2.2 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 G, L, R, s, t, M 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 data structure that is subject to change from one iteration to the next, these assumptions constitute the 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* ⟹ $\{u, v\} \in$ *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 $G.E\ (G2\text{-}2\ L\ s\ M) = G.E\ (G2\text{-}1\ M) \cup \{\{s,\ v\}\ |v.\ v \in set\ (free\text{-}vertices\ L\ M)\}$

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

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

We now show that graph *edmonds-karp.G1* comprises all edges not in the current matching.

lemma (**in** *edmonds-karp*) *E1-cong*:
assumes $G.symmetric\text{-}adjacency'\ G$
assumes $G.symmetric\text{-}adjacency'\ G'$
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
$G.E\ (G.union\ (G1\ G\ (G2\ L\ R\ s\ t\ M))\ (G2\ L\ R\ s\ t\ M)) =$
$G.E\ G \cup \{\{s,\ v\}\ |v.\ v \in set\ (free\text{-}vertices\ L\ M)\} \cup \{\{t,\ v\}\ |v.\ v \in set\ (free\text{-}vertices\ R\ M)\}$

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

We are now able 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 $alt\text{-}bfs\text{-}valid\text{-}input'\ (G1\ G\ (G2\ L\ R\ s\ t\ M))\ (G2\ L\ R\ s\ t\ M)\ s$

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*) $v \neq$ *None*
  shows
    is-shortest-alt-path
      ($\lambda e.\ e \in$ *G.E* (*G2 L R s t M*))
      (*Not* $\circ$ ($\lambda e.\ e \in$ *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 $M \oplus P$.

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* $\oplus$ *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*))

21.2.3 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

21.2.4 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* $\Rightarrow$ $'a$ *graph* $\Rightarrow$ *bool* **where**
is-maximum-matching G M $\equiv$ *graph-matching* G M $\wedge$ $(\forall M'.$ *graph-matching* G $M' \longrightarrow$ *card* $M' \leq$ *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 :: $'a::linorder \Rightarrow 's \Rightarrow 'n \Rightarrow 'n$ **and**
P-update :: $'a \Rightarrow 'a \Rightarrow 'm \Rightarrow 'm$ **and**
M-update :: $'a \Rightarrow 'a \Rightarrow 'm \Rightarrow 'm$ +
assumes *done-1*: *done-1* L R M

lemma (**in** *edmonds-karp-invar-done-1*) *is-maximum-matching-M-tbd*:
shows *is-maximum-matching* $(G.E\ G)$ $(M\text{-}tbd\ M)$

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 :: $'a::linorder \Rightarrow 's \Rightarrow 'n \Rightarrow 'n$ **and**
P-update :: $'a \Rightarrow 'a \Rightarrow 'm \Rightarrow 'm$ **and**
M-update :: $'a \Rightarrow 'a \Rightarrow 'm \Rightarrow 'm$ +
assumes *done-2*: *done-2* t $(m\text{-}tbd\ G\ L\ R\ s\ t\ M)$

lemma (**in** *edmonds-karp-invar-done-2*) *is-maximum-matching-M-tbd*:
shows *is-maximum-matching* $(G.E\ G)$ $(M\text{-}tbd\ M)$

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* $(G.E\ G)$ $(M\text{-}tbd\ (loop'\ G\ L\ R\ s\ t\ M))$
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
theory *Parent-Relation-Partial*

imports
Parent-Relation
begin

partial-function (*tailrec*) *rev-follow-partial* **where**
rev-follow-partial a m v = (*case m v of None* ⇒ *v* # *a* | *Some u* ⇒ *rev-follow-partial* (*v* # *a*) *m u*)

definition *rev-follow* :: (*'a* ⇒ *'a option*) ⇒ *'a* ⇒ *'a list* **where**
rev-follow ≡ *rev-follow-partial* []

lemma *rev-follow-partial-eq-rev-follow*:
assumes *parent m*
shows *rev-follow-partial a m v* = *rev* (*parent.follow m v*) @ *a*

lemma *rev-follow-eq-rev-follow*:
assumes *parent m*
shows *rev-follow m v* = *rev* (*parent.follow m v*)

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

partial-function (**in** *edmonds-karp*) (*tailrec*) *loop'-partial* **where**
loop'-partial G U V s t M =
(*if done-1 U V M then M*
else if done-2 t (*alt-bfs-partial* (*G1 G* (*G2 U V s t M*)) (*G2 U V s t M*) *s*) *then M*
else loop'-partial G U V s t (*augment M* (*butlast* (*tl* (*Parent-Relation-Partial.rev-follow* (*P-lookup* (*alt-bfs-partial* (*G1 G* (*G2 U V s t M*)) (*G2 U V 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*

end
theory *Edmonds-Karp-Impl*
imports
../Alternating-BFS/Alternating-BFS-Impl
Edmonds-Karp-Partial

begin

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*]
thm *E.loop'-partial.simps*

value *alt-bfs-partial* (*update* (*4*::*nat*) (*RBT-Set.insert* (*3*::*nat*) *empty*) (*update* (*3*::*nat*) (*RBT-Set.insert* (*4*::*nat*) *empty*) *empty*)) (*update* (*2*::*nat*) (*RBT-Set.insert* (*1*::*nat*) *empty*) (*update* (*1*::*nat*) (*RBT-Set.insert* (*2*::*nat*) *empty*) *empty*)) *1*
value *loop′-partial* (*update* (*2*::*nat*) (*RBT-Set.insert* (*1*::*nat*) *empty*) (*update* (*1*::*nat*) (*RBT-Set.insert* (*2*::*nat*) *empty*) *empty*)) (*RBT-Set.insert* (*1*::*nat*) *empty*) (*RBT-Set.insert* (*2*::*nat*) *empty*) *1 2 empty*
value *edmonds-karp-partial* (*update* (*2*::*nat*) (*RBT-Set.insert* (*1*::*nat*) *empty*) (*update* (*1*::*nat*) (*RBT-Set.insert* (*2*::*nat*) *empty*) *empty*))

$(nat \times color)\ tree \Rightarrow (nat \times color)\ tree \Rightarrow nat \Rightarrow nat \Rightarrow ((nat \times nat) \times color)\ tree$

end

21.2.5 Undirected graphs

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

end

22 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.

22.1 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