Elementary Graph Traversal Algorithms

Mohammad Abdulaziz
King's College London

March 17, 2025

Abstract

In this entry, we verify the correctness of:

- Depth-first search
- Breadth-first search

The entry's main aim is pedagogical. It has the formal material presented in one chapter of the book Functional Data Structures and Algorithms: A Proof Assistant Approach. The verification is based on a simple set-based representation of directed graphs. The main feature is that a graph's set of vertices is implicitly represented by its edges. The main focus of the development is to develop a representation suitable for mathematical reasoning and for executable algorithms. The entry also shows the verification of executable algorithms that need background mathematical libraries with basic Isabelle tools. Two main features are its automation setup aimed at verifying executable graph algorithms in a human-readable way and to verify the algorithms w.r.t. the same graph representation, making all of them compatible. The setup we use here includes using the function package to model loops and reason about their termination, using records to model program states, and using *locales* to implement parametric as well as step-wise refinement based verification. The material presented here is a part of an ongoing development of formal results on graphs and graph algorithms in Isabelle/HOL[1].

Contents

[1] https://github.com/mabdula/Isabelle-Graph-Library

theory *More-Lists*
imports *Main*
begin

lemma *list-2-preds-aux*:
$[\![x' \in set\ xs;\ P1\ x';\ \bigwedge xs1\ x\ xs2.\ [\![xs = xs1\ @\ [x]\ @\ xs2;\ P1\ x]\!] \Longrightarrow$
$\exists\, ys1\ y\ ys2.\ x\ \#\ xs2 = ys1\ @\ [y]\ @\ ys2 \wedge P2\ y]\!] \Longrightarrow$
$\exists\ xs1\ x\ xs2.\ xs = xs1\ @\ [x]\ @\ xs2 \wedge P2\ x \wedge (\forall\, y{\in}set\ xs2.\ \neg\ P1\ y)$
⟨*proof*⟩

lemma *list-2-preds*:
$[\![\ x{\in}set\ xs;\ P1\ x;\ \bigwedge xs1\ x\ xs2.\ [\![xs = xs1\ @\ [x]\ @\ xs2;\ P1\ x]\!] \Longrightarrow \exists\, ys1\ y\ ys2.$
$x\#xs2 = ys1\ @\ [y]\ @\ ys2 \wedge P2\ y]\!] \Longrightarrow$
$\exists\ xs1\ x\ xs2.\ xs = xs1\ @\ [x]\ @\ xs2 \wedge P2\ x \wedge (\forall\, y{\in}set\ xs2.\ \neg\ P1\ y \wedge \neg\ P2\ y)$

⟨*proof*⟩

lemma *list-inter-mem-iff*: *set xs* ∩ *s* ≠ {} ⟷ (∃ *xs1 x xs2*. *xs* = *xs1* @ [*x*] @ *xs2* ∧ *x* ∈ *s*)
⟨*proof*⟩
end
theory *Pair-Graph*
imports
Main
Graph-Theory.Rtrancl-On
begin

1 A Basic Representation of Diraphs

type-synonym *'v dgraph* = (*'v* × *'v*) *set*

definition *dVs*::(*'v* × *'v*) *set* ⇒ *'v set* **where**
dVs G = ⋃ {{*v1*,*v2*} | *v1 v2*. (*v1*, *v2*) ∈ *G*}

lemma *induct-pcpl*:
⟦*P* []; ⋀*x*. *P* [*x*]; ⋀*x y zs*. *P zs* ⟹ *P* (*x* # *y* # *zs*)⟧ ⟹ *P xs*
⟨*proof*⟩

lemma *induct-pcpl-2*:
⟦*P* []; ⋀*x*. *P* [*x*]; ⋀*x y*. *P* [*x*,*y*]; ⋀*x y z*. *P* [*x*,*y*,*z*]; ⋀*w x y z zs*. *P zs* ⟹ *P* (*w* # *x* # *y* # *z* # *zs*)⟧ ⟹ *P xs*
⟨*proof*⟩

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

lemma *dVs-empty-iff*[*simp*]: *dVs E* = {} ⟷ *E* = {}
⟨*proof*⟩

lemma *dVsI*[*intro*]:
assumes (*a*, *v*) ∈ *dG* **shows** *a* ∈ *dVs dG* **and** *v* ∈ *dVs dG*
⟨*proof*⟩

lemma *dVsI'*:
assumes *e* ∈ *dG* **shows** *fst e* ∈ *dVs dG* **and** *snd e* ∈ *dVs dG*
⟨*proof*⟩

lemma *dVs-union-distr*[*simp*]: *dVs* (*G* ∪ *H*) = *dVs G* ∪ *dVs H*
⟨*proof*⟩

lemma *dVs-union-big-distr*: *dVs* (⋃ *G*) = ⋃(*dVs* ‘ *G*)
⟨*proof*⟩

lemma *dVs-eq*: *dVs E* = *fst* ‘ *E* ∪ *snd* ‘ *E*

⟨*proof*⟩

lemma *finite-vertices-iff*: *finite* (*dVs E*) ⟷ *finite E* (**is** *?L* ⟷ *?R*)
⟨*proof*⟩

abbreviation *reachable1* :: ('*v* × '*v*) *set* ⇒ '*v* ⇒ '*v* ⇒ *bool* (- →⁺₁ - [*100*,*100*] *40*) **where**
reachable1 E u v ≡ (*u*,*v*) ∈ E^+

definition *reachable* :: ('*v* × '*v*) *set* ⇒ '*v* ⇒ '*v* ⇒ *bool* (- →*₁ - [*100*,*100*] *40*) **where**
reachable E u v = ((*u*,*v*) ∈ *rtrancl-on* (*dVs E*) *E*)

lemma *reachableE*[*elim?*]:
assumes (*u*,*v*) ∈ *E*
obtains *e* **where** *e* ∈ *E e* = (*u*, *v*)
⟨*proof*⟩

lemma *reachable-refl*[*intro*!, *Pure.intro*!, *simp*]: *v* ∈ *dVs E* ⟹ *v* →*$_E$ *v*
⟨*proof*⟩

lemma *reachable-trans*[*trans*,*intro*]:
assumes *u* →*$_E$ *v v* →*$_E$ *w* **shows** *u* →*$_E$ *w*
⟨*proof*⟩

lemma *reachable-edge*[*dest*,*intro*]: (*u*,*v*) ∈ *E* ⟹ *u* →*$_E$ *v*
⟨*proof*⟩

lemma *reachable-induct*[*consumes 1*, *case-names base step*]:
assumes *major*: *u* →*$_E$ *v*
and *cases*: ⟦*u* ∈ *dVs E*⟧ ⟹ *P u*
⋀*x y*. ⟦*u* →*$_E$ *x*; (*x*,*y*) ∈ *E*; *P x*⟧ ⟹ *P y*
shows *P v*
⟨*proof*⟩

lemma *converse-reachable-induct*[*consumes 1*, *case-names base step*, *induct pred*: *reachable*]:
assumes *major*: *u* →*$_E$ *v*
and *cases*: *v* ∈ *dVs E* ⟹ *P v*
⋀*x y*. ⟦(*x*,*y*) ∈ *E*; *y* →*$_E$ *v*; *P y*⟧ ⟹ *P x*
shows *P u*
⟨*proof*⟩

lemma *reachable-in-dVs*:
assumes *u* →*$_E$ *v*
shows *u* ∈ *dVs E v* ∈ *dVs E*
⟨*proof*⟩

lemma *reachable1-in-dVs*:

assumes $u \rightarrow^+_E v$
shows $u \in dVs\ E\ v \in dVs\ E$
⟨*proof*⟩

lemma *reachable1-reachable*[*intro*]:
$v \rightarrow^+_E w \Longrightarrow v \rightarrow^*_E w$
⟨*proof*⟩

lemmas *reachable1-reachableE*[*elim*] = *reachable1-reachable*[*elim-format*]

lemma *reachable-neq-reachable1*[*intro*]:
assumes *reach*: $v \rightarrow^*_E w$
and *neq*: $v \neq w$
shows $v \rightarrow^+_E w$
⟨*proof*⟩

lemmas *reachable-neq-reachable1E*[*elim*] = *reachable-neq-reachable1*[*elim-format*]

lemma *arc-implies-dominates*: $e \in E \Longrightarrow (fst\ e,\ snd\ e) \in E$ ⟨*proof*⟩

definition *neighbourhood*::$('v \times 'v)\ set \Rightarrow 'v \Rightarrow 'v\ set$ **where**
neighbourhood $G\ u = \{v.\ (u,v) \in G\}$

lemma
neighbourhoodI[*intro*]: $v \in (neighbourhood\ G\ u) \Longrightarrow (u,v) \in G$ **and**
neighbourhoodD[*dest*]: $(u,v) \in G \Longrightarrow v \in (neighbourhood\ G\ u)$
⟨*proof*⟩

definition *sources* $G = \{u \mid u\ v\ .\ (u,v) \in G\}$

definition *sinks* $G = \{v \mid u\ v\ .\ (u,v) \in G\}$

lemma *dVs-subset*: $G \subseteq G' \Longrightarrow dVs\ G \subseteq dVs\ G'$
⟨*proof*⟩

lemma *dVs-insert*[*elim*]:
$v \in dVs\ (insert\ (x,y)\ G) \Longrightarrow [\![v = x \Longrightarrow P;\ v = y \Longrightarrow P;\ v \in dVs\ G \Longrightarrow P]\!] \Longrightarrow P$
⟨*proof*⟩

lemma *in-neighbourhood-dVs*[*simp*, *intro*]:
$v \in neighbourhood\ G\ u \Longrightarrow v \in dVs\ G$
⟨*proof*⟩

lemma *subset-neighbourhood-dVs*[*simp*, *intro*]:
$neighbourhood\ G\ u \subseteq dVs\ G$
⟨*proof*⟩

lemma *in-dVsE*: $v \in dVs\ G \Longrightarrow [\![(\bigwedge u.\ (u,\ v) \in G \Longrightarrow P);\ (\bigwedge u.\ (v,\ u) \in G \Longrightarrow P)]\!] \Longrightarrow P$
$v \notin dVs\ G \Longrightarrow ([\![(\bigwedge u.\ (u,\ v) \notin G);\ (\bigwedge u.\ (v,\ u) \notin G)]\!] \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *neighoubrhood-union*[*simp*]: *neighbourhood* $(G \cup G')\ u = neighbourhood\ G\ u \cup neighbourhood\ G'\ u$
⟨*proof*⟩

lemma *vs-are-gen*: $dVs\ (set\ E\text{-}impl) = set\ (map\ prod.fst\ E\text{-}impl) \cup set\ (map\ prod.snd\ E\text{-}impl)$
⟨*proof*⟩

lemma *dVs-swap*: $dVs\ (prod.swap\ `\ E) = dVs\ E$
⟨*proof*⟩

end
theory *Vwalk*
imports *Pair-Graph*
begin

2 A Theory on Vertex Walks in a Digraph

context fixes $G :: 'v\ dgraph$ **begin**
inductive *vwalk* **where**
vwalk0: *vwalk* [] |
vwalk1: $v \in dVs\ G \Longrightarrow vwalk\ [v]$ |
vwalk2: $(u,v) \in G \Longrightarrow vwalk\ (v\#vs) \Longrightarrow vwalk\ (u\#v\#vs)$
end
declare *vwalk0*[*simp*]

inductive-simps *vwalk-1*[*simp*]: *vwalk E* [*v*]

inductive-simps *vwalk-2*[*simp*]: *vwalk E* (*u* # *v* # *vs*)

definition *vwalk-bet*::$('v \times 'v)\ set \Rightarrow 'v \Rightarrow 'v\ list \Rightarrow 'v \Rightarrow bool$ **where**
vwalk-bet G u p v = (*vwalk G p* ∧ *p* ≠ [] ∧ *hd p* = *u* ∧ *last p* = *v*)

lemma *vwalk-then-edge*: *vwalk-bet dG u p v* $\Longrightarrow u \neq v \Longrightarrow (\exists v''.\ (u,\ v'') \in dG)$
⟨*proof*⟩

lemma *vwalk-then-in-dVs*: *vwalk dG p* $\Longrightarrow v \in set\ p \Longrightarrow v \in dVs\ dG$
⟨*proof*⟩

lemma *vwalk-cons*: *vwalk dG* (*v1*#*v2*#*p*) ⟹ (*v1*,*v2*) ∈ *dG*
⟨*proof*⟩

lemma *hd-of-vwalk-bet*:
vwalk-bet dG v p v' $\Longrightarrow \exists p'.\ p = v\ \#\ p'$

⟨*proof*⟩

lemma *hd-of-vwalk-bet'*:
vwalk-bet dG v p v' $\Longrightarrow$ *v = hd p*
⟨*proof*⟩

lemma *hd-of-vwalk-bet''*: *vwalk-bet dG u p v* $\Longrightarrow$ *u* $\in$ *set p*
⟨*proof*⟩

lemma *last-of-vwalk-bet*:
vwalk-bet dG v p v' $\Longrightarrow$ *v' = last p*
⟨*proof*⟩

lemma *last-of-vwalk-bet'*:
vwalk-bet dG v p v' $\Longrightarrow$ $\exists p'.$ *p = p' @ [v']*
⟨*proof*⟩

lemma *append-vwalk-pref*: *vwalk dG (p1 @ p2)* $\Longrightarrow$ *vwalk dG p1*
⟨*proof*⟩

lemma *append-vwalk-suff*: *vwalk dG (p1 @ p2)* $\Longrightarrow$ *vwalk dG p2*
⟨*proof*⟩

lemma *append-vwalk*: *vwalk dG p1* $\Longrightarrow$ *vwalk dG p2* $\Longrightarrow$ (*p1* $\neq$ [] $\Longrightarrow$ *p2* $\neq$ [] $\Longrightarrow$ (*last p1*, *hd p2*) $\in$ *dG*) $\Longrightarrow$ *vwalk dG (p1 @ p2)*
⟨*proof*⟩
lemma *split-vwalk*:
vwalk-bet dG v1 (p1 @ v2 # p2) v3 $\Longrightarrow$ *vwalk-bet dG v2 (v2 # p2) v3*
⟨*proof*⟩

lemma *vwalk-bet-cons*:
vwalk-bet dG v (v # p) u $\Longrightarrow$ *p* $\neq$ [] $\Longrightarrow$ *vwalk-bet dG (hd p) p u*
⟨*proof*⟩

lemma *vwalk-bet-cons-2*:
vwalk-bet dG v p v' $\Longrightarrow$ *p* $\neq$ [] $\Longrightarrow$ *vwalk-bet dG v (v # tl p) v'*
⟨*proof*⟩

lemma *vwalk-bet-snoc*:
vwalk-bet dG v' (p @ [v]) v'' $\Longrightarrow$ *v = v''*
⟨*proof*⟩

lemma *vwalk-bet-vwalk*:
p $\neq$ [] $\Longrightarrow$ *vwalk-bet dG (hd p) p (last p)* $\longleftrightarrow$ *vwalk dG p*
⟨*proof*⟩

lemma *vwalk-snoc-edge'*: *vwalk dG (p @ [v])* $\Longrightarrow$ (*v*, *v'*) $\in$ *dG* $\Longrightarrow$ *vwalk dG ((p @ [v]) @ [v'])*
⟨*proof*⟩

lemma *vwalk-snoc-edge*: *vwalk dG* (*p* @ [*v*]) ⟹ (*v*, *v'*) ∈ *dG* ⟹ *vwalk dG* (*p* @ [*v*, *v'*])
⟨*proof*⟩

lemma *vwalk-snoc-edge-2*: *vwalk dG* (*p* @ [*v*, *v'*]) ⟹ (*v*, *v'*) ∈ *dG*
⟨*proof*⟩

lemma *vwalk-append-edge*: *vwalk dG* (*p1* @ *p2*) ⟹ *p1* ≠ [] ⟹ *p2* ≠ [] ⟹ (*last p1*, *hd p2*) ∈ *dG*
⟨*proof*⟩

lemma *vwalk-transitive-rel*:
assumes (⋀*x y z*. *R x y* ⟹ *R y z* ⟹ *R x z*) (⋀*v1 v2*. (*v1*, *v2*) ∈ *dG* ⟹ *R v1 v2*)
shows *vwalk dG* (*v*#*vs*) ⟹ *v'* ∈ *set vs* ⟹ *R v v'*
⟨*proof*⟩

lemma *vwalk-transitive-rel'*:
assumes (⋀*x y z*. *R x y* ⟹ *R y z* ⟹ *R x z*) (⋀*v1 v2*. (*v1*, *v2*) ∈ *dG* ⟹ *R v1 v2*)
shows *vwalk dG* (*vs* @ [*v*]) ⟹ *v'* ∈ *set vs* ⟹ *R v' v*
⟨*proof*⟩

fun *edges-of-vwalk* **where**
edges-of-vwalk [] = [] |
edges-of-vwalk [*v*] = [] |
edges-of-vwalk (*v*#*v'*#*l*) = (*v*, *v'*) # *edges-of-vwalk* (*v'*#*l*)

lemma *vwalk-ball-edges*: *vwalk E p* ⟹ *b* ∈ *set* (*edges-of-vwalk p*) ⟹ *b* ∈ *E*
⟨*proof*⟩

lemma *edges-of-vwalk-index*:
Suc i < *length p* ⟹ *edges-of-vwalk p* ! *i* = (*p* ! *i*, *p* ! *Suc i*)
⟨*proof*⟩

lemma *edges-of-vwalk-length*: *length* (*edges-of-vwalk p*) = *length p* − *1*
⟨*proof*⟩

With the given assumptions we can only obtain an outgoing edge from *v*.

lemma *edges-of-vwalk-for-inner*:
assumes *p* ! *i* = *v Suc i* < *length p*
obtains *w* **where** *edges-of-vwalk p* ! *i* = (*v*, *w*)
⟨*proof*⟩

For an incoming edge we need an additional assumption (*0* < *i*).

lemma *edges-of-vwalk-for-inner'*:

assumes $p\ !\ (Suc\ i) = v\ Suc\ i < length\ p$
obtains u **where** $(u,\ v) = edges\text{-}of\text{-}vwalk\ p\ !\ i$
⟨*proof*⟩

lemma *hd-edges-neq-last*:
$[\![(last\ p,\ hd\ p) \notin set\ (edges\text{-}of\text{-}vwalk\ p);\ hd\ p \neq last\ p;\ p \neq []]\!] \Longrightarrow$
$hd\ (edges\text{-}of\text{-}vwalk\ (last\ p\ \#\ p)) \neq last\ (edges\text{-}of\text{-}vwalk\ (last\ p\ \#\ p))$
⟨*proof*⟩

lemma *v-in-edge-in-vwalk*:
assumes $(u,\ v) \in set\ (edges\text{-}of\text{-}vwalk\ p)$
shows $u \in set\ p\ v \in set\ p$
⟨*proof*⟩

lemma *distinct-edges-of-vwalk*:
$distinct\ p \Longrightarrow distinct\ (edges\text{-}of\text{-}vwalk\ p)$
⟨*proof*⟩

lemma *distinct-edges-of-vwalk-cons*:
$distinct\ (edges\text{-}of\text{-}vwalk\ (v\ \#\ p)) \Longrightarrow distinct\ (edges\text{-}of\text{-}vwalk\ p)$
⟨*proof*⟩

lemma *tl-vwalk-is-vwalk*: $vwalk\ E\ p \Longrightarrow vwalk\ E\ (tl\ p)$
⟨*proof*⟩

lemma *vwalk-concat*:
assumes $vwalk\ E\ p\ vwalk\ E\ q\ q \neq []\ p \neq [] \Longrightarrow last\ p = hd\ q$
shows $vwalk\ E\ (p\ @\ tl\ q)$
⟨*proof*⟩

lemma *edges-of-vwalk-append-2*:
$p' \neq [] \Longrightarrow edges\text{-}of\text{-}vwalk\ (p\ @\ p') = edges\text{-}of\text{-}vwalk\ (p\ @\ [hd\ p'])\ @\ edges\text{-}of\text{-}vwalk\ p'$
⟨*proof*⟩

lemma *edges-of-vwalk-append*: $\exists\, ep.\ edges\text{-}of\text{-}vwalk\ (p\ @\ p') = ep\ @\ edges\text{-}of\text{-}vwalk\ p'$
⟨*proof*⟩

lemma *append-butlast-last-cancel*: $p \neq [] \Longrightarrow butlast\ p\ @\ last\ p\ \#\ p' = p\ @\ p'$
⟨*proof*⟩

lemma *edges-of-vwalk-append-3*:
assumes $p \neq []$
shows $edges\text{-}of\text{-}vwalk\ (p\ @\ p') = edges\text{-}of\text{-}vwalk\ p\ @\ edges\text{-}of\text{-}vwalk\ (last\ p\ \#\ p')$
⟨*proof*⟩

lemma *vwalk-vertex-has-edge*:

assumes *length p* ≥ *2 v* ∈ *set p*
obtains *e u* **where** *e* ∈ *set* (*edges-of-vwalk p*) *e* = (*u*, *v*) ∨ *e* = (*v*, *u*)
⟨*proof*⟩

lemma *v-in-edge-in-vwalk-inj*:
assumes *e* ∈ *set* (*edges-of-vwalk p*)
obtains *u v* **where** *e* = (*u*, *v*)
⟨*proof*⟩

lemma *v-in-edge-in-vwalk-gen*:
assumes *e* ∈ *set* (*edges-of-vwalk p*) *e* = (*u*, *v*)
shows *u* ∈ *set p v* ∈ *set p*
⟨*proof*⟩

lemma *edges-of-vwalk-dVs*: *dVs* (*set* (*edges-of-vwalk p*)) ⊆ *set p*
⟨*proof*⟩

lemma *last-v-snd-last-e*:
assumes *length p* ≥ *2*
shows *last p* = *snd* (*last* (*edges-of-vwalk p*)) — is this the best formulation for this?
⟨*proof*⟩

lemma *hd-v-fst-hd-e*:
assumes *length p* ≥ *2*
shows *hd p* = *fst* (*hd* (*edges-of-vwalk p*))
⟨*proof*⟩

lemma *last-in-edge*:
p ≠ [] ⟹ ∃ *u*. (*u*, *last p*) ∈ *set* (*edges-of-vwalk* (*v* # *p*)) ∧ *u* ∈ *set* (*v* # *p*)
⟨*proof*⟩

lemma *edges-of-vwalk-append-subset*:
shows *set* (*edges-of-vwalk p'*) ⊆ *set* (*edges-of-vwalk* (*p* @ *p'*))
⟨*proof*⟩

lemma *nonempty-vwalk-vwalk-bet*[*intro?*]:
assumes *vwalk E p p* ≠ [] *hd p* = *u last p* = *v*
shows *vwalk-bet E u p v*
⟨*proof*⟩

lemma *vwalk-bet-nonempty*:
assumes *vwalk-bet E u p v*
shows [*simp*]: *p* ≠ []
⟨*proof*⟩

lemma *vwalk-bet-nonempty-vwalk*[*elim*]:
assumes *vwalk-bet E u p v*
shows *vwalk E p p* ≠ [] *hd p* = *u last p* = *v*

⟨*proof*⟩

lemma *vwalk-bet-reflexive*[*intro*]:
assumes $w \in dVs\ E$
shows *vwalk-bet* $E\ w\ [w]\ w$
⟨*proof*⟩

lemma *singleton-hd-last*: $q \neq [] \Longrightarrow tl\ q = [] \Longrightarrow hd\ q = last\ q$
⟨*proof*⟩

lemma *singleton-hd-last'*: $q \neq [] \Longrightarrow butlast\ q = [] \Longrightarrow hd\ q = last\ q$
⟨*proof*⟩

lemma *vwalk-bet-transitive*:
assumes *vwalk-bet* $E\ u\ p\ v$ *vwalk-bet* $E\ v\ q\ w$
shows *vwalk-bet* $E\ u\ (p\ @\ tl\ q)\ w$
⟨*proof*⟩

lemma *vwalk-bet-in-dVs*:
assumes *vwalk-bet* $E\ a\ p\ b$
shows $set\ p \subseteq dVs\ E$
⟨*proof*⟩

lemma *vwalk-bet-endpoints*:
assumes *vwalk-bet* $E\ u\ p\ v$
shows [*simp*]: $u \in dVs\ E$
and [*simp*]: $v \in dVs\ E$
⟨*proof*⟩

lemma *vwalk-bet-pref*:
assumes *vwalk-bet* $E\ u\ (pr\ @\ [x]\ @\ su)\ v$
shows *vwalk-bet* $E\ u\ (pr\ @\ [x])\ x$
⟨*proof*⟩

lemma *vwalk-bet-suff*:
assumes *vwalk-bet* $E\ u\ (pr\ @\ [x]\ @\ su)\ v$
shows *vwalk-bet* $E\ x\ (x\ \#\ su)\ v$
⟨*proof*⟩

lemma *edges-are-vwalk-bet*:
assumes $(v,\ w) \in E$
shows *vwalk-bet* $E\ v\ [v,\ w]\ w$
⟨*proof*⟩

lemma *induct-vwalk-bet*[*case-names path1 path2*, *consumes 1*, *induct set*: *vwalk-bet*]:
assumes *vwalk-bet* $E\ a\ p\ b$
assumes *Path1*: $\bigwedge v.\ v \in dVs\ E \Longrightarrow P\ E\ [v]\ v\ v$
assumes *Path2*: $\bigwedge v\ v'\ vs\ b.\ (v,\ v') \in E \Longrightarrow vwalk\text{-}bet\ E\ v'\ (v'\ \#\ vs)\ b \Longrightarrow P\ E\ (v'\ \#\ vs)\ v'\ b \Longrightarrow P\ E\ (v\ \#\ v'\ \#\ vs)\ v\ b$

shows *P E p a b*
⟨*proof*⟩

lemma *vwalk-append*:
assumes *vwalk E xs vwalk E ys last xs = hd ys*
shows *vwalk E* (*xs* @ *tl ys*)
⟨*proof*⟩

lemma *vwalk-append2*:
assumes *vwalk E* (*xs* @ [*x*]) *vwalk E* (*x* # *ys*)
shows *vwalk E* (*xs* @ *x* # *ys*)
⟨*proof*⟩

lemma *vwalk-appendD-last*:
vwalk E (*xs* @ [*x*, *y*]) ⟹ *vwalk E* (*xs* @ [*x*])
⟨*proof*⟩

lemma *vwalk-ConsD*:
vwalk E (*x* # *xs*) ⟹ *vwalk E xs*
⟨*proof*⟩

lemmas *vwalkD* = *vwalk-ConsD append-vwalk-pref append-vwalk-suff*

lemma *vwalk-alt-induct*[*consumes 1*, *case-names Single Snoc*]:
assumes
vwalk E p P [] (⋀*x*. *P* [*x*])
⋀*y x xs*. (*y*, *x*) ∈ *E* ⟹ *vwalk E* (*xs* @ [*y*]) ⟹ *P* (*xs* @ [*y*]) ⟹ *P* (*xs* @ [*y*, *x*])
shows *P p*
⟨*proof*⟩

lemma *vwalk-append-single*:
assumes *vwalk E p* (*last p*, *x*) ∈ *E*
shows *vwalk E* (*p* @ [*x*])
⟨*proof*⟩

lemmas *vwalk-decomp* = *append-vwalk-pref append-vwalk-suff vwalk-append-edge*

lemma *vwalk-rotate*:
assumes *vwalk E* (*x* # *xs* @ *y* # *ys* @ [*x*])
shows *vwalk E* (*y* # *ys* @ *x* # *xs* @ [*y*])
⟨*proof*⟩

lemma *vwalk-bet-nonempty'*[*simp*]: ¬ *vwalk-bet E u* [] *v*
⟨*proof*⟩

lemma *vwalk-ConsE*:
assumes *vwalk E* (*a* # *p*) *p* ≠ []
obtains *e* **where** *e* ∈ *E e* = (*a*, *hd p*) *vwalk E p*

⟨*proof*⟩

lemma *vwalk-reachable*:
$p \neq [] \Longrightarrow vwalk\ E\ p \Longrightarrow hd\ p \rightarrow^*_E last\ p$
⟨*proof*⟩

lemma *vwalk-reachable'*:
$vwalk\ E\ p \Longrightarrow p \neq [] \Longrightarrow hd\ p = u \Longrightarrow last\ p = v \Longrightarrow u \rightarrow^*_E v$
⟨*proof*⟩

lemma *vwalkI*: $(x,\ hd\ p) \in E \Longrightarrow vwalk\ E\ p \Longrightarrow vwalk\ E\ (x\#p)$
⟨*proof*⟩

lemma *reachable-vwalk*:
assumes $u \rightarrow^*_E v$
shows $\exists\, p.\ hd\ p = u \wedge last\ p = v \wedge vwalk\ E\ p \wedge p \neq []$
⟨*proof*⟩

lemma *reachable-vwalk-iff*:
$u \rightarrow^*_E v \longleftrightarrow (\exists\, p.\ hd\ p = u \wedge last\ p = v \wedge vwalk\ E\ p \wedge p \neq [])$
⟨*proof*⟩

lemma *reachable-vwalk-bet-iff*:
$u \rightarrow^*_E v \longleftrightarrow (\exists\, p.\ vwalk\text{-}bet\ E\ u\ p\ v)$
⟨*proof*⟩

lemma *reachable-vwalk-betD*:
$vwalk\text{-}bet\ E\ u\ p\ v \Longrightarrow u \rightarrow^*_E v$
⟨*proof*⟩

lemma *vwalk-reachable1*:
$vwalk\ E\ (u\ \#\ p\ @\ [v]) \Longrightarrow u \rightarrow^+_E v$
⟨*proof*⟩

lemma *reachable1-vwalk*:
assumes $u \rightarrow^+_E v$
shows $\exists\, p.\ vwalk\ E\ (u\ \#\ p\ @\ [v])$
⟨*proof*⟩

lemma *reachable1-vwalk-iff*:
$u \rightarrow^+_E v \longleftrightarrow (\exists\, p.\ vwalk\ E\ (u\ \#\ p\ @\ [v]))$
⟨*proof*⟩

lemma *reachable-vwalk-iff2*:
$u \rightarrow^*_E v \longleftrightarrow (u = v \wedge u \in dVs\ E \vee (\exists\, p.\ vwalk\ E\ (u\ \#\ p\ @\ [v])))$
⟨*proof*⟩

lemma *vwalk-remove-cycleE*:
assumes $vwalk\ E\ (u\ \#\ p\ @\ [v])$

obtains p' **where** *vwalk* E $(u \# p' @ [v])$
$\quad$ *distinct* p' $u \notin$ *set* p' $v \notin$ *set* p' *set* $p' \subseteq$ *set* p
⟨*proof*⟩

abbreviation *closed-vwalk-bet* :: $('v \times 'v)$ *set* $\Rightarrow$ $'v$ *list* $\Rightarrow$ $'v \Rightarrow$ *bool* **where**
$\quad$ *closed-vwalk-bet* E c $v \equiv$ *vwalk-bet* E v c $v \wedge$ *Suc* $0 <$ *length* c

lemma *edge-iff-vwalk-bet*: $(u, v) \in E =$ *vwalk-bet* E u $[u, v]$ v
⟨*proof*⟩

lemma *vwalk-bet-in-vertices*: *vwalk-bet* E u p $v \Longrightarrow w \in$ *set* $p \Longrightarrow w \in$ *dVs* E
⟨*proof*⟩

lemma *vwalk-bet-hd-neq-last-implies-edges-nonempty*:
$\quad$ **assumes** *vwalk-bet* E u p v
$\quad$ **assumes** $u \neq v$
$\quad$ **shows** $E \neq \{\}$
$\quad$ ⟨*proof*⟩

lemma *vwalk-bet-edges-in-edges*: *vwalk-bet* E u p $v \Longrightarrow$ *set* (*edges-of-vwalk* p) $\subseteq E$
$\quad$ ⟨*proof*⟩

lemma *vwalk-bet-prefix-is-vwalk-bet*:
$\quad$ **assumes** $p \neq []$
$\quad$ **assumes** *vwalk-bet* E u $(p @ q)$ v
$\quad$ **shows** *vwalk-bet* E u p (*last* p)
$\quad$ ⟨*proof*⟩

lemma *vwalk-bet-suffix-is-vwalk-bet*:
$\quad$ **assumes** $q \neq []$
$\quad$ **assumes** *vwalk-bet* E u $(p @ q)$ v
$\quad$ **shows** *vwalk-bet* E (*hd* q) q v
$\quad$ ⟨*proof*⟩

lemma *vwalk-bet-append-append-is-vwalk-bet*:
$\quad$ **assumes** *vwalk-bet* E u p v
$\quad$ **assumes** *vwalk-bet* E v q w
$\quad$ **assumes** *vwalk-bet* E w r x
$\quad$ **shows** *vwalk-bet* E u $(p @ tl\ q @ tl\ r)$ x
$\quad$ ⟨*proof*⟩

lemma
$\quad$ **assumes** $p \neq []$
$\quad$ **shows** *edges-of-vwalk* $(p @ q) =$ *edges-of-vwalk* $p @$ *edges-of-vwalk* $([last\ p] @ q)$
$\quad$ ⟨*proof*⟩

fun *is-vwalk-bet-vertex-decomp* :: $('v \times 'v)$ *set* $\Rightarrow$ $'v$ *list* $\Rightarrow$ $'v \Rightarrow$ $'v$ *list* $\times$ $'v$ *list* $\Rightarrow$ *bool* **where**
$\quad$ *is-vwalk-bet-vertex-decomp* E p v $(q, r) \longleftrightarrow p = q @ tl\ r \wedge (\exists u\ w.$ *vwalk-bet* E

u q v ∧ vwalk-bet E v r w)

definition *vwalk-bet-vertex-decomp* :: (*'v* × *'v*) *set* ⇒ *'v list* ⇒ *'v* ⇒ *'v list* × *'v list* **where**
vwalk-bet-vertex-decomp E p v = (*SOME qr*. *is-vwalk-bet-vertex-decomp E p v qr*)

lemma *vwalk-bet-vertex-decompE*:
assumes *p-vwalk*: *vwalk-bet E u p v*
assumes *p-decomp*: *p* = *xs* @ *y* # *ys*
obtains *q r* **where**
p = *q* @ *tl r*
q = *xs* @ [*y*]
r = *y* # *ys*
vwalk-bet E u q y
vwalk-bet E y r v
⟨*proof*⟩

lemma *vwalk-bet-vertex-decomp-is-vwalk-bet-vertex-decomp*:
assumes *p-vwalk*: *vwalk-bet E u p w*
assumes *v-in-p*: *v* ∈ *set p*
shows *is-vwalk-bet-vertex-decomp E p v* (*vwalk-bet-vertex-decomp E p v*)
⟨*proof*⟩

lemma *vwalk-bet-vertex-decompE-2*:
assumes *p-vwalk*: *vwalk-bet E u p w*
assumes *v-in-p*: *v* ∈ *set p*
assumes *qr-def*: *vwalk-bet-vertex-decomp E p v* = (*q*, *r*)
obtains
p = *q* @ *tl r*
vwalk-bet E u q v
vwalk-bet E v r w
⟨*proof*⟩

definition *vtrail* :: (*'v* × *'v*) *set* ⇒ *'v* ⇒ *'v list* ⇒ *'v* ⇒ *bool* **where**
vtrail E u p v = (*vwalk-bet E u p v* ∧ *distinct* (*edges-of-vwalk p*))

abbreviation *closed-vtrail* :: (*'v* × *'v*) *set* ⇒ *'v list* ⇒ *'v* ⇒ *bool* **where**
closed-vtrail E c v ≡ *vtrail E v c v* ∧ *Suc 0* < *length c*

lemma *closed-vtrail-implies-Cons*:
assumes *closed-vtrail E c v*
shows *c* = *v* # *tl c*
⟨*proof*⟩

lemma *tl-non-empty-conv*:
shows *tl l* ≠ [] ⟷ *Suc 0* < *length l*

⟨*proof*⟩

lemma *closed-vtrail-implies-tl-nonempty*:
assumes *closed-vtrail E c v*
shows *tl c* ≠ []
⟨*proof*⟩

lemma *vtrail-in-vertices*:
vtrail E u p v ⟹ *w* ∈ *set p* ⟹ *w* ∈ *dVs E*
⟨*proof*⟩

lemma
assumes *vtrail E u p v*
shows *vtrail-hd-in-vertices*: *u* ∈ *dVs E*
and *vtrail-last-in-vertices*: *v* ∈ *dVs E*
⟨*proof*⟩

lemma *list-set-tl*: *x* ∈ *set* (*tl xs*) ⟹ *x* ∈ *set xs*
⟨*proof*⟩

lemma *closed-vtrail-hd-tl-in-vertices*:
assumes *closed-vtrail E c v*
shows *hd* (*tl c*) ∈ *dVs E*
⟨*proof*⟩

lemma *vtrail-prefix-is-vtrail*:
notes *vtrail-def* [*simp*]
assumes *p* ≠ []
assumes *vtrail E u* (*p* @ *q*) *v*
shows *vtrail E u p* (*last p*)
⟨*proof*⟩

lemma *vtrail-suffix-is-vtrail*:
notes *vtrail-def* [*simp*]
assumes *q* ≠ []
assumes *vtrail E u* (*p* @ *q*) *v*
shows *vtrail E* (*hd q*) *q v*
⟨*proof*⟩

definition *distinct-vwalk-bet* :: (*'v* × *'v*) *set* ⇒ *'v* ⇒ *'v list* ⇒ *'v* ⇒ *bool* **where**
distinct-vwalk-bet E u p v = (*vwalk-bet E u p v* ∧ *distinct p*)

lemma *distinct-vwalk-bet-length-le-card-vertices*:
assumes *distinct-vwalk-bet E u p v*
assumes *finite E*
shows *length p* ≤ *card* (*dVs E*)
⟨*proof*⟩

lemma *distinct-vwalk-bet-triples-finite*:

assumes *finite E*
shows *finite* {(*p*, *u*, *v*). *distinct-vwalk-bet E u p v*}
⟨*proof*⟩

lemma *distinct-vwalk-bets-finite*:
finite E ⟹ *finite* {*p*. *distinct-vwalk-bet E u p v*}
⟨*proof*⟩

3 Vwalks to paths (as opposed to arc walks (*awalk-to-apath* before)

fun *is-closed-decomp* :: ('*v* × '*v*) *set* ⇒ '*v list* ⇒ '*v list* × '*v list* × '*v list* ⇒ *bool*
where
is-closed-decomp E p (*q*, *r*, *s*) ⟷
p = *q* @ *tl r* @ *tl s* ∧
(∃ *u v w*. *vwalk-bet E u q v* ∧ *closed-vwalk-bet E r v* ∧ *vwalk-bet E v s w*) ∧
distinct q

definition *closed-vwalk-bet-decomp* :: ('*v* × '*v*) *set* ⇒ '*v list* ⇒ '*v list* × '*v list* × '*v list* **where**
closed-vwalk-bet-decomp E p = (*SOME qrs*. *is-closed-decomp E p qrs*)

lemma *closed-vwalk-bet-decompE*:
assumes *p-vwalk*: *vwalk-bet E u p v*
assumes *p-decomp*: *p* = *xs* @ *y* # *ys* @ *y* # *zs*
obtains *q r s* **where**
p = *q* @ *tl r* @ *tl s*
q = *xs* @ [*y*]
r = *y* # *ys* @ [*y*]
s = *y* # *zs*
vwalk-bet E u q y
vwalk-bet E y r y
vwalk-bet E y s v
⟨*proof*⟩

lemma *closed-vwalk-bet-decomp-is-closed-decomp*:
assumes *p-vwalk*: *vwalk-bet E u p v*
assumes *p-not-distinct*: ¬ *distinct p*
shows *is-closed-decomp E p* (*closed-vwalk-bet-decomp E p*)
⟨*proof*⟩

lemma *closed-vwalk-bet-decompE-2*:
assumes *p-vwalk*: *vwalk-bet E u p v*
assumes *p-not-distinct*: ¬ *distinct p*
assumes *qrs-def*: *closed-vwalk-bet-decomp E p* = (*q*, *r*, *s*)
obtains
p = *q* @ *tl r* @ *tl s*
∃ *w*. *vwalk-bet E u q w* ∧ *closed-vwalk-bet E r w* ∧ *vwalk-bet E w s v*

distinct q
⟨*proof*⟩

function *vwalk-bet-to-distinct* :: (*'v* × *'v*) *set* ⇒ *'v list* ⇒ *'v list* **where**
vwalk-bet-to-distinct E p =
(*if* (∃ *u v*. *vwalk-bet E u p v*) ∧ ¬ *distinct p*
then let (*q*, *r*, *s*) = *closed-vwalk-bet-decomp E p in vwalk-bet-to-distinct E* (*q* @ *tl s*)
else p)
⟨*proof*⟩

termination *vwalk-bet-to-distinct*
⟨*proof*⟩

lemma *vwalk-bet-to-distinct-induct* [*consumes 1*, *case-names path decomp*]:
assumes *vwalk-bet E u p v*
assumes *distinct*: ⋀*p*. ⟦ *vwalk-bet E u p v*; *distinct p* ⟧ ⟹ *P p*
assumes
decomp: ⋀*p q r s*. ⟦ *vwalk-bet E u p v*; ¬ *distinct p*;
closed-vwalk-bet-decomp E p = (*q*, *r*, *s*); *P* (*q* @ *tl s*) ⟧ ⟹ *P p*
shows *P p*
⟨*proof*⟩

lemma *vwalk-bet-to-distinct-is-distinct-vwalk-bet*:
assumes *vwalk-bet E u p v*
shows *distinct-vwalk-bet E u* (*vwalk-bet-to-distinct E p*) *v*
⟨*proof*⟩

lemma *vwalk-betE*[*elim?*]:
assumes *vwalk-bet E u p v*
assumes *singleton*: ⟦ *v*∈ *dVs E*; *u* = *v*⟧ ⟹ *P*
assumes
step: ⋀*p' x*. ⟦ *p* = *u*#*x*#*p'*; (*u*,*x*)∈*E*; *vwalk-bet E x* (*x*#*p'*) *v*⟧ ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *vwalk-subset*:
⟦*vwalk G p*; *G* ⊆ *G'*⟧ ⟹ *vwalk G' p*
⟨*proof*⟩

lemma *vwalk-bet-subset*:
⟦*vwalk-bet G u p v*; *G* ⊆ *G'*⟧ ⟹ *vwalk-bet G' u p v*
⟨*proof*⟩

lemma *reachable-subset*[*intro*]:
⟦*u* →** *G v*; *G* ⊆ *G'*⟧ ⟹ *u* →** *G' v*
⟨*proof*⟩

lemma *reachable-Union-1*[*intro*]:

$[\![u \to^*_G v]\!] \Longrightarrow u \to^*_{G \cup G'} v$
$[\![u \to^*_G v]\!] \Longrightarrow u \to^*_{G' \cup G} v$
⟨*proof*⟩

lemma *reachableE*[*elim?*]:
assumes *reachable E u v*
assumes *singleton*: $[\![v \in dVs\ E;\ u = v]\!] \Longrightarrow P$
assumes
step: $\bigwedge x.\ [\![(u,x) \in E;\ reachable\ E\ x\ v]\!] \Longrightarrow P$
shows *P*
⟨*proof*⟩

lemma *vwalk-insertE*[*case-names nil sing1 sing2 in-e in-E*]:
$[\![vwalk\ (insert\ e\ E)\ p;$
$(p = [] \Longrightarrow P);$
$(\bigwedge u\ v.\ p = [v] \Longrightarrow e = (u,v) \Longrightarrow P);$
$(\bigwedge u\ v.\ p = [v] \Longrightarrow e = (v,u) \Longrightarrow P);$
$(\bigwedge v.\ p = [v] \Longrightarrow v \in dVs\ E \Longrightarrow P);$
$(\bigwedge p'\ v1\ v2.\ [\![vwalk\ \{e\}\ [v1, v2];\ vwalk\ (insert\ e\ E)\ (v2\ \#\ p');\ p = v1\ \#\ v2\ \#\ p']\!] \Longrightarrow P);$
$(\bigwedge p'\ v1\ v2.\ [\![vwalk\ E\ [v1,v2];\ vwalk\ (insert\ e\ E)\ (v2\ \#\ p');\ p = v1\ \#\ v2\ \#\ p']\!] \Longrightarrow P\ )]\!]$
$\Longrightarrow P$
⟨*proof*⟩

A lemma which allows for case splitting over paths when doing induction on graph edges.

lemma *vwalk-bet-insertE*[*case-names nil sing1 sing2 in-e in-E*]:
$[\![vwalk\text{-}bet\ (insert\ e\ E)\ v1\ p\ v2;$
$([\![v1 \in dVs\ (insert\ e\ E);\ v1 = v2;\ p = []]\!] \Longrightarrow P);$
$(\bigwedge u\ v.\ p = [u] \Longrightarrow e = (u,v) \Longrightarrow P);$
$(\bigwedge u\ v.\ p = [v] \Longrightarrow e = (u,v) \Longrightarrow P);$
$(\bigwedge v.\ p = [v] \Longrightarrow v = v1 \Longrightarrow v = v2 \Longrightarrow v \in dVs\ E \Longrightarrow P);$
$(\bigwedge p'\ v3.\ [\![vwalk\text{-}bet\ \{e\}\ v1\ [v1,v3]\ v3;\ vwalk\text{-}bet\ (insert\ e\ E)\ v3\ (v3\ \#\ p')\ v2;\ p = v1\ \#\ v3\ \#\ p']\!] \Longrightarrow P);$
$(\bigwedge p'\ v3.\ [\![vwalk\text{-}bet\ E\ v1\ [v1,v3]\ v3;\ vwalk\text{-}bet\ (insert\ e\ E)\ v3\ (v3\ \#\ p')\ v2;\ p = v1\ \#\ v3\ \#\ p']\!] \Longrightarrow P)]\!]$
$\Longrightarrow P$
⟨*proof*⟩

find-theorems *name*: *induct vwalk-bet*

lemma *vwalk-bet2*[*simp*]:
vwalk-bet G u (*u # v # vs*) *b* $\longleftrightarrow$ ((*u*,*v*) $\in$ *G* $\wedge$ *vwalk-bet G v* (*v # vs*) *b*)
⟨*proof*⟩

lemma *butlast-vwalk-is-vwalk*: *vwalk E p* $\Longrightarrow$ *vwalk E* (*butlast p*)
⟨*proof*⟩

lemma *vwalk-concat-2*:
assumes *vwalk E p vwalk E q q* $\neq$ [] *p* $\neq$ [] $\Longrightarrow$ *last p* = *hd q*
shows *vwalk E* (*butlast p* @ *q*)
⟨*proof*⟩

lemma *vwalk-bet-transitive-2*:
assumes *vwalk-bet E u p v vwalk-bet E v q w*
shows *vwalk-bet E u* (*butlast p* @ *q*) *w*
⟨*proof*⟩

lemma *vwalk-not-vwalk*:
⟦*vwalk G p*; ¬*vwalk G′ p*⟧ $\Longrightarrow$
(∃ (*u*,*v*) ∈ *set* (*edges-of-vwalk p*). (*u*,*v*) ∈ (*G* − *G′*)) ∨
(∃ *v*∈*set p*. *v* ∈ (*dVs G* − *dVs G′*))
⟨*proof*⟩

lemma *vwalk-not-vwalk-2*:
⟦*vwalk G p*; ¬*vwalk G′ p*; *length p* ≥ *2*⟧ $\Longrightarrow$
(∃ (*u*,*v*) ∈ *set* (*edges-of-vwalk p*). (*u*,*v*) ∈ (*G* − *G′*))
⟨*proof*⟩

lemma *vwalk-not-vwalk-elim*:
⟦*vwalk G p*; ¬*vwalk G′ p*⟧ $\Longrightarrow$
⟦$\bigwedge$*u v*. ⟦(*u*,*v*) ∈ *set* (*edges-of-vwalk p*); (*u*,*v*) ∈ (*G* − *G′*)⟧ $\Longrightarrow$ *P*;
$\bigwedge$*v*. ⟦*v*∈*set p*; *v* ∈ (*dVs G* − *dVs G′*)⟧ $\Longrightarrow$ *P*⟧ $\Longrightarrow$ *P*
⟨*proof*⟩

lemma *vwalk-not-vwalk-elim-2*:
⟦*vwalk G p*; ¬*vwalk G′ p*; *length p* ≥ *2*⟧ $\Longrightarrow$
⟦$\bigwedge$*u v*. ⟦(*u*,*v*) ∈ *set* (*edges-of-vwalk p*); (*u*,*v*) ∈ (*G* − *G′*)⟧ $\Longrightarrow$ *P*
⟧ $\Longrightarrow$ *P*
⟨*proof*⟩

lemma *vwalk-bet-not-vwalk-bet*:
⟦*vwalk-bet G u p v*; ¬*vwalk-bet G′ u p v*⟧ $\Longrightarrow$
(∃ (*u*,*v*) ∈ *set* (*edges-of-vwalk p*). (*u*,*v*) ∈ (*G* − *G′*)) ∨
(∃ *v*∈*set p*. *v* ∈ (*dVs G* − *dVs G′*))
⟨*proof*⟩

lemma *vwalk-bet-not-vwalk-bet-elim*:
⟦*vwalk-bet G u p v*; ¬*vwalk-bet G′ u p v*⟧ $\Longrightarrow$
⟦$\bigwedge$*u v*. ⟦(*u*,*v*) ∈ *set* (*edges-of-vwalk p*); (*u*,*v*) ∈ (*G* − *G′*)⟧ $\Longrightarrow$ *P*;
$\bigwedge$*v*. ⟦*v*∈*set p*; *v* ∈ (*dVs G* − *dVs G′*)⟧ $\Longrightarrow$ *P*⟧ $\Longrightarrow$ *P*

⟨*proof*⟩

lemma *vwalk-bet-not-vwalk-bet-elim-2*:
⟦*vwalk-bet G u p v*; ¬*vwalk-bet G′ u p v*; *length p* ≥ *2*⟧ ⟹
⟦⋀*u v*. ⟦(*u*,*v*) ∈ *set* (*edges-of-vwalk p*); (*u*,*v*) ∈ (*G* − *G′*)⟧ ⟹ *P*
⟧ ⟹ *P*
⟨*proof*⟩

lemma *vwalk-bet-props*:
vwalk-bet G u p v ⟹ (⟦*vwalk G p*; *p*≠[]; *hd p* = *u*; *last p* = *v*⟧ ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *no-outgoing-last*:
⟦*vwalk G p*; ⋀*v*. (*u*,*v*) ∉ *G*; *u* ∈ *set p*⟧ ⟹ *last p* = *u*
⟨*proof*⟩

lemma *not-vwalk-bet-empty*: ¬ *Vwalk.vwalk-bet* {} *u p v*
⟨*proof*⟩

lemma *edges-in-vwalk-split*:
(*u*, *v*) ∈ *set* (*edges-of-vwalk p*) ⟹ ∃ *p1 p2*. *p* = *p1* @[*u*,*v*]@*p2*
⟨*proof*⟩

end
theory *Enat-Misc*
imports *Main HOL−Library.Extended-Nat*
begin

declare *one-enat-def*

declare *zero-enat-def*

lemma *eval-enat-numeral*:
numeral Num.One = *eSuc 0*
numeral (*Num.Bit0 n*) = *eSuc* (*numeral* (*Num.BitM n*))
numeral (*Num.Bit1 n*) = *eSuc* (*numeral* (*Num.Bit0 n*))
⟨*proof*⟩

declare *eSuc-enat*[*symmetric*, *simp*]

end
theory *Dist*
imports *Enat-Misc Vwalk*
begin

4 Distances

4.1 Distance from a vertex

definition *distance*::$('v \times 'v)$ *set* $\Rightarrow$ $'v \Rightarrow 'v \Rightarrow$ *enat* **where**
distance G u v = (*INF p. if Vwalk.vwalk-bet G u p v then length p − 1 else* ∞)

lemma *vwalk-bet-dist*:
Vwalk.vwalk-bet G u p v $\Longrightarrow$ *distance G u v* $\leq$ *length p − 1*
⟨*proof*⟩

lemma *reachable-dist*:
reachable G u v $\Longrightarrow$ *distance G u v* $< \infty$
⟨*proof*⟩

lemma *unreachable-dist*:
$\neg$*reachable G u v* $\Longrightarrow$ *distance G u v* $= \infty$
⟨*proof*⟩

lemma *dist-reachable*:
distance G u v $< \infty \Longrightarrow$ *reachable G u v*
⟨*proof*⟩

lemma *reachable-dist-2*:
assumes *reachable G u v*
obtains *p* **where** *Vwalk.vwalk-bet G u p v distance G u v = length p − 1*
⟨*proof*⟩

lemma *triangle-ineq-reachable*:
assumes *reachable G u v reachable G v w*
shows *distance G u w* $\leq$ *distance G u v* + *distance G v w*
⟨*proof*⟩

lemma *triangle-ineq*:
distance G u w $\leq$ *distance G u v* + *distance G v w*
⟨*proof*⟩

lemma *distance-split*:
⟦*distance G u v* $\neq \infty$; *distance G u v = distance G u w + distance G w v* ⟧ $\Longrightarrow$
$\exists\, w'$. *reachable G u w'* $\wedge$ *distance G u w' = distance G u w* $\wedge$
reachable G w' v $\wedge$ *distance G w' v = distance G w' v*
⟨*proof*⟩

lemma *dist-inf*: *v* $\notin$ *dVs G* $\Longrightarrow$ *distance G u v* $= \infty$
⟨*proof*⟩

lemma *dist-inf-2*: *v* $\notin$ *dVs G* $\Longrightarrow$ *distance G v u* $= \infty$
⟨*proof*⟩

lemma *dist-eq*: $[\![\bigwedge p.\ Vwalk.vwalk\text{-}bet\ G'\ u\ p\ v \Longrightarrow Vwalk.vwalk\text{-}bet\ G\ u\ (map\ f\ p)\ v]\!] \Longrightarrow$
$distance\ G\ u\ v \leq distance\ G'\ u\ v$
⟨*proof*⟩

lemma *distance-subset*: $G \subseteq G' \Longrightarrow distance\ G'\ u\ v \leq distance\ G\ u\ v$
⟨*proof*⟩

lemma *distance-neighbourhood*:
$[\![v \in neighbourhood\ G\ u]\!] \Longrightarrow distance\ G\ u\ v \leq 1$
⟨*proof*⟩

4.2 Shortest Paths

definition *shortest-path*::$('v \times 'v)\ set \Rightarrow 'v \Rightarrow 'v\ list \Rightarrow 'v \Rightarrow bool$ **where**
$shortest\text{-}path\ G\ u\ p\ v = (distance\ G\ u\ v = length\ p - 1 \wedge vwalk\text{-}bet\ G\ u\ p\ v)$

lemma *shortest-path-props*[*elim*]:
$shortest\text{-}path\ G\ u\ p\ v \Longrightarrow ([\![distance\ G\ u\ v = length\ p - 1;\ vwalk\text{-}bet\ G\ u\ p\ v]\!] \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *shortest-path-intro*:
$[\![distance\ G\ u\ v = length\ p - 1;\ vwalk\text{-}bet\ G\ u\ p\ v]\!] \Longrightarrow shortest\text{-}path\ G\ u\ p\ v$
⟨*proof*⟩

lemma *shortest-path-vwalk*: $shortest\text{-}path\ G\ u\ p\ v \Longrightarrow vwalk\text{-}bet\ G\ u\ p\ v$
⟨*proof*⟩

lemma *shortest-path-dist*: $shortest\text{-}path\ G\ u\ p\ v \Longrightarrow distance\ G\ u\ v = length\ p - 1$
⟨*proof*⟩

lemma *shortest-path-split-1*:
$[\![shortest\text{-}path\ G\ u\ (p1\ @\ x\ \#\ p2)\ v]\!] \Longrightarrow shortest\text{-}path\ G\ x\ (x\ \#\ p2)\ v$
⟨*proof*⟩

lemma *shortest-path-split-2*:
$[\![shortest\text{-}path\ G\ u\ (p1\ @\ x\ \#\ p2)\ v]\!] \Longrightarrow shortest\text{-}path\ G\ u\ (p1\ @\ [x])\ x$
⟨*proof*⟩

lemma *shortest-path-split-distance*:
$[\![shortest\text{-}path\ G\ u\ (p1\ @\ x\ \#\ p2)\ v]\!] \Longrightarrow distance\ G\ u\ x \leq distance\ G\ u\ v$
⟨*proof*⟩

lemma *shortest-path-split-distance'*:
$[\![x \in set\ p;\ shortest\text{-}path\ G\ u\ p\ v]\!] \Longrightarrow distance\ G\ u\ x \leq distance\ G\ u\ v$
⟨*proof*⟩

lemma *shortest-path-exists*:
assumes *reachable G u v*
obtains *p* **where** *shortest-path G u p v*
⟨*proof*⟩

lemma *shortest-path-exists-2*:
assumes *distance G u v* < ∞
obtains *p* **where** *shortest-path G u p v*
⟨*proof*⟩

lemma *not-distinct-props*:
¬*distinct xs* ⟹ (⋀*x1 x2 xs1 xs2 xs3*. ⟦*xs* = *xs1* @ *x1*# *xs2* @ *x2* # *xs3*; *x1* = *x2*⟧ ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *shortest-path-distinct*:
shortest-path G u p v ⟹ *distinct p*
⟨*proof*⟩

lemma *diet-eq'*: ⟦⋀*p*. *shortest-path G' u p v* ⟹ *shortest-path G u* (*map f p*) *v*⟧
⟹
distance G u v ≤ *distance G' u v*
⟨*proof*⟩

lemma *distance-0*:
(*u* = *v* ∧ *v* ∈ *dVs G*) ⟷ *distance G u v* = *0*
⟨*proof*⟩

lemma *distance-neighbourhood'*:
⟦*v* ∈ *neighbourhood G u*⟧ ⟹ *distance G x v* ≤ *distance G x u* + *1*
⟨*proof*⟩

lemma *Suc-le-length-iff-2*:
(*Suc n* ≤ *length xs*) = (∃ *x ys*. *xs* = *ys* @ [*x*] ∧ *n* ≤ *length ys*)
⟨*proof*⟩

lemma *distance-parent*:
⟦*distance G u v* < ∞; *u* ≠ *v*⟧ ⟹
∃ *w*. *distance G u w* + *1* = *distance G u v* ∧ *v* ∈ *neighbourhood G w*
⟨*proof*⟩

4.3 Distance from a set of vertices

definition *distance-set*::('*v* × '*v*) *set* ⇒ '*v set* ⇒ '*v* ⇒ *enat* **where**
distance-set G U v = (*INF u*∈*U*. *distance G u v*)

lemma *dist-set-inf*: $v \notin dVs\ G \Longrightarrow distance\text{-}set\ G\ U\ v = \infty$
⟨*proof*⟩

lemma *dist-set-mem*[*intro*]: $u \in U \Longrightarrow distance\text{-}set\ G\ U\ v \leq distance\ G\ u\ v$
⟨*proof*⟩

lemma *dist-not-inf''*: ⟦*distance-set G U v* ≠ ∞; *u*∈*U*; *distance G u v* = *distance-set G U v*⟧
$\Longrightarrow \exists p.\ vwalk\text{-}bet\ G\ u\ (u\#p)\ v \wedge length\ p = distance\ G\ u\ v \wedge set\ p \cap U = \{\}$
⟨*proof*⟩

lemma *dist-not-inf'''*:
⟦*distance-set G U v* ≠ ∞; *u*∈*U*; *distance G u v* = *distance-set G U v*⟧
$\Longrightarrow \exists p.\ shortest\text{-}path\ G\ u\ (u\#p)\ v \wedge set\ p \cap U = \{\}$
⟨*proof*⟩

lemma *distance-set-union*:
distance-set G $(U \cup V)\ v \leq$ *distance-set G U v*
⟨*proof*⟩

lemma *lt-lt-infnty*: $x < (y::enat) \Longrightarrow x < \infty$
⟨*proof*⟩

lemma *finite-dist-nempty*:
$distance\text{-}set\ G\ V\ v \neq \infty \Longrightarrow V \neq \{\}$
⟨*proof*⟩

lemma *distance-set-wit*:
assumes $v \in V$
obtains v' **where** $v' \in V$ *distance-set G V x* = *distance G v' x*
⟨*proof*⟩

lemma *distance-set-wit'*:
assumes *distance-set G V v* ≠ ∞
obtains v' **where** $v' \in V$ *distance-set G V x* = *distance G v' x*
⟨*proof*⟩

lemma *dist-set-not-inf*: *distance-set G U v* ≠ ∞ ⟹ ∃*u*∈*U*. *distance G u v* = *distance-set G U v*
⟨*proof*⟩

lemma *dist-not-inf'*: *distance-set G U v* ≠ ∞ ⟹
∃*u*∈*U*. *distance G u v* = *distance-set G U v* ∧ *reachable G u v*
⟨*proof*⟩

lemma *distance-on-vwalk*:
⟦*distance-set G U v* = *distance G u v*; *u* ∈ *U*; *shortest-path G u p v*; *w* ∈ *set p*⟧
⟹ *distance-set G U w* = *distance G u w*
⟨*proof*⟩

lemma *diff-le-self-enat*: *m* − *n* ≤ (*m*::*enat*)
⟨*proof*⟩

lemma *shortest-path-dist-set-union*:
⟦*distance-set G U v* = *distance G u v*; *u* ∈ *U*; *shortest-path G u* (*p1* @ *x* # *p2*) *v*;
x ∈ *V*; ⋀*v'*. *v'* ∈ *V* ⟹ *distance-set G U v'* = *distance G u x*⟧
⟹ *distance-set G* (*U* ∪ *V*) *v* = *distance-set G U v* − *distance G u x*
⟨*proof*⟩

lemma *Inf-enat-def1*:
fixes *K*::*enat set*
assumes *K* ≠ {}
shows *Inf K* ∈ *K*
⟨*proof*⟩

lemma *INF-plus-enat*:
V ≠ {} ⟹ (*INF v*∈*V*. (*f*::*'a* ⇒ *enat*) *v*) + (*x*::*enat*) = (*INF v*∈*V*. *f v* + *x*)
⟨*proof*⟩

lemma *distance-set-neighbourhood*:
⟦*v* ∈ *neighbourhood G u*; *Vs* ≠ {}⟧ ⟹ *distance-set G Vs v* ≤ *distance-set G Vs u* + *1*
⟨*proof*⟩

lemma *distance-set-parent*:
⟦*distance-set G Vs v* < ∞; *Vs* ≠ {}; *v* ∉ *Vs*⟧ ⟹
∃ *w*. *distance-set G Vs w* + *1* = *distance-set G Vs v* ∧ *v* ∈ *neighbourhood G w*
⟨*proof*⟩

lemma *distance-set-parent'*:
⟦*0* < *distance-set G Vs v*; *distance-set G Vs v* < ∞; *Vs* ≠ {}⟧ ⟹
∃ *w*. *distance-set G Vs w* + *1* = *distance-set G Vs v* ∧ *v* ∈ *neighbourhood G w*
⟨*proof*⟩

lemma *distance-set-0*[*simp*]: ⟦*v* ∈ *dVs G*⟧ ⟹ *distance-set G Vs v* = *0* ⟷ *v* ∈ *Vs*
⟨*proof*⟩

lemma *dist-set-leq*:
⟦⋀*u*. *u* ∈ *Vs* ⟹ *distance G u v* ≤ *distance G' u v*⟧ ⟹ *distance-set G Vs v* ≤ *distance-set G' Vs v*
⟨*proof*⟩

lemma *dist-set-eq*:
⟦⋀*u*. *u* ∈ *Vs* ⟹ *distance G u v* = *distance G′ u v*⟧ ⟹ *distance-set G Vs v* = *distance-set G′ Vs v*
⟨*proof*⟩

lemma *distance-set-subset*: *G* ⊆ *G′* ⟹ *distance-set G′ Vs v* ≤ *distance-set G Vs v*
⟨*proof*⟩

lemma *vwalk-bet-dist-set*:
⟦*Vwalk.vwalk-bet G u p v*; *u* ∈ *U*⟧ ⟹ *distance-set G U v* ≤ *length p* − *1*
⟨*proof*⟩

end
theory *Set-Addons*
imports *HOL−Data-Structures.Set-Specs*
begin
context *Set*
begin
bundle *automation* = *set-empty*[*simp*] *set-isin*[*simp*] *set-insert*[*simp*] *set-delete*[*simp*]
invar-empty[*simp*] *invar-insert*[*simp*] *invar-delete*[*simp*]

end
end
theory *Map-Addons*
imports *HOL−Data-Structures.Map-Specs*
begin
context *Map*
begin
bundle *automation* = *map-empty*[*simp*] *map-update*[*simp*] *map-delete*[*simp*] *invar-empty*[*simp*]
invar-update[*simp*] *invar-delete*[*simp*]
end
end
theory *Set2-Addons*
imports *HOL−Data-Structures.Set-Specs*
begin

context *Set2*
begin
bundle *automation* =
set-union[*simp*] *set-inter*[*simp*]
set-diff[*simp*] *invar-union*[*simp*]
invar-inter[*simp*] *invar-diff*[*simp*]

notation *inter* (**infixl** ∩$_G$ *100*)

notation *diff* (**infixl** $-_G$ *100*)
notation *union* (**infixl** $\cup_G$ *100*)
end

end
theory *Pair-Graph-Specs*
imports *Vwalk Map-Addons Set-Addons*
begin

5 A Digraph Representation for Efficient Executable Functions

We develop a locale modelling an abstract data type (ADT) which abstractly models a graph as an adjacency map: i.e. every vertex is mapped to a *set* of adjacent vertices, and this latter set is again modelled using the ADT of sets provided in Isabelle/HOL's distribution.

We then show that this ADT can be implemented using existing implementations of the *set* ADT.

locale *Set-Choose* = *set*: *Set*
where *set* = *t-set* **for** *t-set* ($[\text{-}]_s$) +
fixes *sel* :: $'s \Rightarrow 'a$

assumes *choose* [*simp*]: $s \neq empty \Longrightarrow isin\ s\ (sel\ s)$

begin
context
includes *set.automation*
begin

5.1 Abstraction lemmas

These are lemmas for automation. Their purpose is to remove any mention of the locale set ADT constructs and replace it with Isabelle's native sets.

lemma *choose'*[*simp*, *intro*,*dest*]:
$s \neq empty \Longrightarrow invar\ s \Longrightarrow sel\ s \in t\text{-}set\ s$
⟨*proof*⟩

lemma *choose''*[*intro*]:
$invar\ s \Longrightarrow s \neq empty \Longrightarrow t\text{-}set\ s \subseteq s' \Longrightarrow sel\ s \in s'$
⟨*proof*⟩

lemma *emptyD*[*dest*]:
$s = empty \Longrightarrow t\text{-}set\ s = \{\}$
$s \neq empty \Longrightarrow invar\ s \Longrightarrow t\text{-}set\ s \neq \{\}$
$empty = s \Longrightarrow t\text{-}set\ s = \{\}$
$empty \neq s \Longrightarrow invar\ s \Longrightarrow t\text{-}set\ s \neq \{\}$

⟨*proof*⟩
end
end

named-theorems *Graph-Spec-Elims*
named-theorems *Graph-Spec-Intros*
named-theorems *Graph-Spec-Simps*

locale *Pair-Graph-Specs* =
adjmap: *Map*
where *update* = *update* **and** *invar* = *adjmap-inv* +

vset: *Set-Choose*
where *empty* = *vset-empty* **and** *delete* = *vset-delete* **and** *invar* = *vset-inv*

for *update* :: *'v* ⇒ *'vset* ⇒ *'adjmap* ⇒ *'adjmap* **and** *adjmap-inv* :: *'adjmap* ⇒ *bool*
and

vset-empty :: *'vset* **and** *vset-delete* :: *'v* ⇒ *'vset* ⇒ *'vset* **and**
vset-inv

begin

notation *vset-empty* ($\emptyset_V$)
notation *empty* ($\emptyset_G$)

abbreviation *isin'* (**infixl** $\in_G$ *50*) **where** *isin' G v* ≡ *isin v G*
abbreviation *not-isin'* (**infixl** $\notin_G$ *50*) **where** *not-isin' G v* ≡ ¬ *isin' G v*

definition *set-of-map* (*m*::*'adjmap*) = {(*u*,*v*). *case* (*lookup m u*) *of Some vs* ⇒ *v* $\in_G$ *vs*}

definition *graph-inv G* = (*adjmap-inv G* ∧ (∀ *v vset*. *lookup G v* = *Some vset* ⟶ *vset-inv vset*))
definition *finite-graph G* = (*finite* {*v*. (*lookup G v*) ≠ *None*})
definition *finite-vsets* = (∀ *vset*. *finite* (*t-set vset*))

definition *neighb*::*'adjmap* ⇒ *'v* ⇒ *'vset* **where**
(*neighb G v*) = (*case* (*lookup G v*) *of Some vset* ⇒ *vset* | - ⇒ *vset-empty*)

lemmas [*code*] = *neighb-def*

notation *neighb* ($\mathcal{N}_G$ - - *100*)

definition *digraph-abs* ([-]$_G$) **where** *digraph-abs G* = {(*u*,*v*). *v* $\in_G$ ($\mathcal{N}_G$ *G u*)}

definition *add-edge G u v* =
(
  case (*lookup G u*) *of Some vset* ⇒
  let
    vset = *the* (*lookup G u*);
    vset′ = *insert v vset*;
    digraph′ = *update u vset′ G*
  in
    digraph′
  | - ⇒
  let
    vset′ = *insert v vset-empty*;
    digraph′ = *update u vset′ G*
  in
    digraph′
)

definition *delete-edge G u v* =
(
  case (*lookup G u*) *of Some vset* ⇒
  let
    vset = *the* (*lookup G u*);
    vset′ = *vset-delete v vset*;
    digraph′ = *update u vset′ G*
  in
    digraph′
  | - ⇒ *G*
)

context — Locale properties
  includes *vset.set.automation* **and** *adjmap.automation*
  fixes *G*::*′adjmap*
begin

lemma *graph-invE*[*elim*]:
  graph-inv G ⟹ (⟦*adjmap-inv G*; (⋀*v vset*. *lookup G v* = *Some vset* ⟹ *vset-inv vset*)⟧ ⟹ *P*) ⟹ *P*
  ⟨*proof*⟩

lemma *graph-invI*[*intro*]:
  ⟦*adjmap-inv G*; (⋀*v vset*. *lookup G v* = *Some vset* ⟹ *vset-inv vset*)⟧ ⟹ *graph-inv G*
  ⟨*proof*⟩

lemma *finite-graphE*[*elim*]:
  finite-graph G ⟹ (*finite* {*v*. (*lookup G v*) ≠ *None*} ⟹ *P*) ⟹ *P*
  ⟨*proof*⟩

lemma *finite-graphI*[*intro*]:
finite {*v*. (*lookup G v*) ≠ *None*} ⟹ *finite-graph G*
⟨*proof*⟩

lemma *finite-vsetsE*[*elim*]:
finite-vsets ⟹ ((⋀*N*. *finite* (*t-set N*)) ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *finite-vsetsI*[*intro*]:
(⋀*N*. *finite* (*t-set N*)) ⟹ *finite-vsets*
⟨*proof*⟩

lemma *neighbourhood-invars'*[*simp*,*dest*]:
graph-inv G ⟹ *vset-inv* (*𝒩*$_G$ *G v*)
⟨*proof*⟩

lemma *finite-graph*[*intro*!]:
assumes *graph-inv G finite-graph G finite-vsets*
shows *finite* (*digraph-abs G*)
⟨*proof*⟩

corollary *finite-vertices*[*intro*!]:
assumes *graph-inv G finite-graph G finite-vsets*
shows *finite* (*dVs* (*digraph-abs G*))
⟨*proof*⟩

5.2 Abstraction lemmas

These are lemmas for automation. Their purpose is to remove any mention of the neighbourhood concept implemented using the locale constructs and replace it with abstract terms on pair graphs.

lemma *are-connected-abs*[*simp*]:
graph-inv G ⟹ *v* ∈ *t-set* (*𝒩*$_G$ *G u*) ⟷ (*u*,*v*) ∈ *digraph-abs G*
⟨*proof*⟩

lemma *are-connected-absD*[*dest*]:
⟦*v* ∈ *t-set* (*𝒩*$_G$ *G u*); *graph-inv G*⟧ ⟹ (*u*,*v*) ∈ *digraph-abs G*
⟨*proof*⟩

lemma *are-connected-absI*[*intro*]:
⟦(*u*,*v*) ∈ *digraph-abs G*; *graph-inv G*⟧ ⟹ *v* ∈ *t-set* (*𝒩*$_G$ *G u*)
⟨*proof*⟩

lemma *neighbourhood-absD*[*dest*!]:
⟦*t-set* (*𝒩*$_G$ *G x*) ≠ {}; *graph-inv G*⟧ ⟹ *x* ∈ *dVs* (*digraph-abs G*)
⟨*proof*⟩

lemma *neighbourhood-abs*[*simp*]:
graph-inv G $\Longrightarrow$ *t-set* ($\mathcal{N}_G$ *G u*) = *neighbourhood* (*digraph-abs G*) *u*
⟨*proof*⟩

lemma *adjmap-inv-insert*[*intro*]: *graph-inv G* $\Longrightarrow$ *graph-inv* (*add-edge G u v*)
⟨*proof*⟩

lemma *digraph-abs-insert*[*simp*]: *graph-inv G* $\Longrightarrow$ *digraph-abs* (*add-edge G u v*) = *Set.insert* (*u*,*v*) (*digraph-abs G*)
⟨*proof*⟩

lemma *adjmap-inv-delete*[*intro*]: *graph-inv G* $\Longrightarrow$ *graph-inv* (*delete-edge G u v*)
⟨*proof*⟩

lemma *digraph-abs-delete*[*simp*]: *graph-inv G* $\Longrightarrow$ *digraph-abs* (*delete-edge G u v*) = (*digraph-abs G*) − {(*u*,*v*)}
⟨*proof*⟩

end — Properties context

end

Pair-Graph-Specs

end
theory *DFS*
imports *Pair-Graph-Specs Set2-Addons Set-Addons*
begin

6 Depth-Frist Search

6.1 The program state

datatype *return* = *Reachable* | *NotReachable*

record (*'ver*, *'vset*) *DFS-state* = *stack*:: *'ver list seen*:: *'vset return*:: *return*

6.2 Setup for automation

named-theorems *call-cond-elims*
named-theorems *call-cond-intros*
named-theorems *ret-holds-intros*
named-theorems *invar-props-intros*
named-theorems *invar-props-elims*
named-theorems *invar-holds-intros*
named-theorems *state-rel-intros*
named-theorems *state-rel-holds-intros*

6.3 A *locale* for fixing data structures and their implemenations

locale *DFS* =

Graph: *Pair-Graph-Specs*
where *lookup* = *lookup* +
set-ops: *Set2 vset-empty vset-delete - t-set vset-inv insert*

for *lookup* :: *'adjmap⇒ 'v ⇒ 'vset option* +

fixes *G*::*'adjmap* **and** *s*::*'v* **and** *t*::*'v*

begin

definition *DFS-axioms* = (*Graph.graph-inv G* ∧ *Graph.finite-graph G* ∧ *Graph.finite-vsets*
∧ *s* ∈ *dVs* (*Graph.digraph-abs G*))

abbreviation *neighb' v* == *Graph.neighb G v*

notation *neighb'* ($\mathcal{N}_G$ - *100*)

6.4 Defining the Algorithm

function (*domintros*) *DFS*::(*'v*, *'vset*) *DFS-state* ⇒ (*'v*, *'vset*) *DFS-state* **where**
DFS dfs-state =
(*case* (*stack dfs-state*) *of* (*v* # *stack-tl*) ⇒
(*if v* = *t then*
(*dfs-state* (|*return* := *Reachable*|))
else ((*if* ($\mathcal{N}_G$ *v* $-_G$ (*seen dfs-state*)) ≠ $\emptyset_V$ *then*
let u = (*sel* (($\mathcal{N}_G$ *v*) $-_G$ (*seen dfs-state*)));
stack' = *u*# (*stack dfs-state*);
seen' = *insert u* (*seen dfs-state*)
in
(*DFS* (*dfs-state* (|*stack* := *stack'*,
seen := *seen'* |)))
else
let stack' = *stack-tl in*
DFS (*dfs-state* (|*stack* := *stack'*|)))
)
)
| - ⇒ (*dfs-state* (|*return* := *NotReachable*|))
)
⟨*proof*⟩

6.5 Setup for Reasoning About the Algorithm

definition *initial-state*::(*'v*,*'vset*) *DFS-state* **where**
initial-state = (|*stack* = [*s*], *seen* = *insert s* $\emptyset_V$, *return* = *NotReachable*|)

definition *DFS-call-1-conds dfs-state* =
(*case stack dfs-state of* (*v* # *stack-tl*) ⇒
(*if v* = *t then*
False
else ((*if* ((*N*$_G$ *v*) −$_G$ (*seen dfs-state*)) ≠ (∅$_V$) *then*
True
else False)
)
)
| - ⇒ *False*
)

lemma *DFS-call-1-conds*[*call-cond-elims*]:
DFS-call-1-conds dfs-state ⟹
⟦⟦∃ *v stack-tl*. *stack dfs-state* = *v* # *stack-tl*;
hd (*stack dfs-state*) ≠ *t*;
(*N*$_G$ (*hd* (*stack dfs-state*))) −$_G$ (*seen dfs-state*) ≠ (∅$_V$)⟧ ⟹ *P*⟧ ⟹
P
⟨*proof*⟩

definition *DFS-upd1 dfs-state* = (
let
N = (*N*$_G$ (*hd* (*stack dfs-state*)));
u = (*sel* ((*N* −$_G$ (*seen dfs-state*))));
stack′ = *u* # (*stack dfs-state*);
seen′ = *insert u* (*seen dfs-state*)
in
dfs-state (|*stack* := *stack*′, *seen* := *seen*′|))

definition *DFS-call-2-conds*::('*v*, '*vset*) *DFS-state* ⇒ *bool* **where**
DFS-call-2-conds dfs-state =
(*case stack dfs-state of* (*v* # *stack-tl*) ⇒
(*if v* = *t then*
False
else (
(*if* ((*N*$_G$ *v*) −$_G$ (*seen dfs-state*)) ≠ (∅$_V$) *then*
False
else True)
)
)
| - ⇒ *False*
)

lemma *DFS-call-2-condsE*[*call-cond-elims*]:
DFS-call-2-conds dfs-state ⟹
⟦⟦∃ *v stack-tl*. *stack dfs-state* = *v* # *stack-tl*;
hd (*stack dfs-state*) ≠ *t*;

$(\mathcal{N}_G\ (hd\ (stack\ dfs\text{-}state)))\ -_G\ (seen\ dfs\text{-}state) = (\emptyset_V)]\!] \Longrightarrow P]\!] \Longrightarrow$
P
⟨*proof*⟩

definition *DFS-upd2 dfs-state* =
$((dfs\text{-}state\ (\!|stack := tl\ (stack\ dfs\text{-}state)|\!)))$

definition *DFS-ret-1-conds dfs-state* =
$(case\ stack\ dfs\text{-}state\ of\ (v\ \#\ stack\text{-}tl) \Rightarrow$
$(if\ v = t\ then$
$False$
$else\ ($
$(if\ ((\mathcal{N}_G\ v)\ -_G\ (seen\ dfs\text{-}state)) \neq \emptyset_V\ then$
$False$
$else\ False)$
$)$
$)$
$|\ \text{-} \Rightarrow True$
$)$

lemma *DFS-ret-1-conds*[*call-cond-elims*]:
DFS-ret-1-conds dfs-state $\Longrightarrow$
$[\![[\![stack\ dfs\text{-}state = []]\!] \Longrightarrow P]\!] \Longrightarrow$
P
⟨*proof*⟩

lemma *DFS-call-4-condsI*[*call-cond-intros*]:
$[\![stack\ dfs\text{-}state = []]\!] \Longrightarrow$ *DFS-ret-1-conds dfs-state*
⟨*proof*⟩

definition *DFS-ret1 dfs-state* = $(dfs\text{-}state\ (\!|return := NotReachable|\!))$

definition *DFS-ret-2-conds dfs-state* =
$(case\ stack\ dfs\text{-}state\ of\ (v\ \#\ stack\text{-}tl) \Rightarrow$
$(if\ v = t\ then$
$True$
$else\ ($
$(if\ (\mathcal{N}_G\ v\ -_G\ (seen\ dfs\text{-}state)) \neq \emptyset_V\ then$
$False$
$else\ False)$
$)$
$)$
$|\ \text{-} \Rightarrow False$
$)$

lemma *DFS-ret-2-conds*[*call-cond-elims*]:
DFS-ret-2-conds dfs-state $\Longrightarrow$

$[\![\bigwedge v\ stack\text{-}tl.\ [\![stack\ dfs\text{-}state = v\ \#\ stack\text{-}tl;$
$(hd\ (stack\ dfs\text{-}state)) = t]\!] \Longrightarrow P]\!] \Longrightarrow$
P
⟨*proof*⟩

lemma *DFS-ret-2-condsI*[*call-cond-intros*]:
$\bigwedge v\ stack\text{-}tl.\ [\![stack\ dfs\text{-}state = v\ \#\ stack\text{-}tl;\ (hd\ (stack\ dfs\text{-}state)) = t]\!] \Longrightarrow$
DFS-ret-2-conds dfs-state
⟨*proof*⟩

definition *DFS-ret2 dfs-state* = (*dfs-state* (|*return* := *Reachable*|))

lemma *DFS-cases*:
assumes *DFS-call-1-conds dfs-state* $\Longrightarrow P$
DFS-call-2-conds dfs-state $\Longrightarrow P$
DFS-ret-1-conds dfs-state $\Longrightarrow P$
DFS-ret-2-conds dfs-state $\Longrightarrow P$
shows P
⟨*proof*⟩

lemma *DFS-simps*:
assumes *DFS-dom dfs-state*
shows*DFS-call-1-conds dfs-state* $\Longrightarrow$ *DFS dfs-state* = *DFS* (*DFS-upd1 dfs-state*)
DFS-call-2-conds dfs-state $\Longrightarrow$ *DFS dfs-state* = *DFS* (*DFS-upd2 dfs-state*)
DFS-ret-1-conds dfs-state $\Longrightarrow$ *DFS dfs-state* = *DFS-ret1 dfs-state*
DFS-ret-2-conds dfs-state $\Longrightarrow$ *DFS dfs-state* = *DFS-ret2 dfs-state*
⟨*proof*⟩

lemma *DFS-induct*:
assumes *DFS-dom dfs-state*
assumes $\bigwedge$*dfs-state*. [\![*DFS-dom dfs-state*;
DFS-call-1-conds dfs-state $\Longrightarrow P$ (*DFS-upd1 dfs-state*);
DFS-call-2-conds dfs-state $\Longrightarrow P$ (*DFS-upd2 dfs-state*)]\!] $\Longrightarrow P$
dfs-state
shows P *dfs-state*
⟨*proof*⟩

lemma *DFS-domintros*:
assumes *DFS-call-1-conds dfs-state* $\Longrightarrow$ *DFS-dom* (*DFS-upd1 dfs-state*)
assumes *DFS-call-2-conds dfs-state* $\Longrightarrow$ *DFS-dom* (*DFS-upd2 dfs-state*)
shows *DFS-dom dfs-state*
⟨*proof*⟩

6.6 Loop Invariants

definition *invar-well-formed*::('*v*,'*vset*) *DFS-state* ⇒ *bool* **where**
invar-well-formed dfs-state = *vset-inv* (*seen dfs-state*)

definition *invar-stack-walk*::('*v*,'*vset*) *DFS-state* ⇒ *bool* **where**

invar-stack-walk dfs-state = (*Vwalk.vwalk* (*Graph.digraph-abs G*) (*rev* (*stack dfs-state*)))

definition *invar-seen-stack*::('*v*,'*vset*) *DFS-state* ⇒ *bool* **where**
invar-seen-stack dfs-state ⟷
distinct (*stack dfs-state*)
∧ *set* (*stack dfs-state*) ⊆ *t-set* (*seen dfs-state*)
∧ *t-set* (*seen dfs-state*) ⊆ *dVs* (*Graph.digraph-abs G*)

definition *invar-s-in-stack*::('*v*,'*vset*) *DFS-state* ⇒ *bool* **where**
invar-s-in-stack dfs-state ⟷
(*stack* (*dfs-state*) ≠ [] ⟶ *last* (*stack dfs-state*) = *s*)

definition *invar-visited-through-seen*::('*v*,'*vset*) *DFS-state* ⇒ *bool* **where**
invar-visited-through-seen dfs-state =
(∀ *v* ∈ *t-set* (*seen dfs-state*).
(∀ *p*. *Vwalk.vwalk-bet* (*Graph.digraph-abs G*) *v p t* ∧ *distinct p* ⟶ (*set p* ∩ *set* (*stack dfs-state*) ≠ {})))

definition *call-1-measure*::('*v*,'*vset*) *DFS-state* ⇒ *nat* **where**
call-1-measure dfs-state = *card* (*dVs* (*Graph.digraph-abs G*) − *t-set* (*seen dfs-state*))

definition *call-2-measure*::('*v*,'*vset*) *DFS-state* ⇒ *nat* **where**
call-2-measure dfs-state = *card* (*set* (*stack dfs-state*))

definition *DFS-term-rel*::(('*v*,'*vset*) *DFS-state* × ('*v*,'*vset*) *DFS-state*) *set* **where**
DFS-term-rel = (*call-1-measure*) <∗*mlex*∗> (*call-2-measure*) <∗*mlex*∗> {}

end

locale *DFS-thms* = *DFS* +

assumes *DFS-axioms*: *DFS-axioms*

begin

context
includes *set-ops.automation* **and** *Graph.adjmap.automation* **and** *Graph.vset.set.automation*

begin

lemma *graph-inv*[*simp*,*intro*]:
Graph.graph-inv G
Graph.finite-graph G
Graph.finite-vsets
⟨*proof*⟩

lemma *s-in-G*[*simp*,*intro*]: *s* ∈ *dVs* (*Graph.digraph-abs G*)
⟨*proof*⟩

lemma *finite-neighbourhoods*[*simp*]:
finite (*t-set N*)
⟨*proof*⟩

lemmas *simps*[*simp*] = *Graph.neighbourhood-abs*[*OF graph-inv*(*1*)] *Graph.are-connected-abs*[*OF graph-inv*(*1*)]

lemma *invar-well-formed-props*[*invar-props-elims*]:
invar-well-formed dfs-state ⟹
(⟦*vset-inv* (*seen dfs-state*)⟧ ⟹ *P*) ⟹
P
⟨*proof*⟩

lemma *invar-well-formed-intro*[*invar-props-intros*]: ⟦*vset-inv* (*seen dfs-state*)⟧
⟹ *invar-well-formed dfs-state*
⟨*proof*⟩

lemma *invar-well-formed-holds-1*[*invar-holds-intros*]:
⟦*DFS-call-1-conds dfs-state*; *invar-well-formed dfs-state*⟧ ⟹
invar-well-formed (*DFS-upd1 dfs-state*)
⟨*proof*⟩

lemma *invar-well-formed-holds-2*[*invar-holds-intros*]: ⟦*DFS-call-2-conds dfs-state*; *invar-well-formed dfs-state*⟧ ⟹ *invar-well-formed* (*DFS-upd2 dfs-state*)
⟨*proof*⟩

lemma *invar-well-formed-holds-4*[*invar-holds-intros*]: ⟦*DFS-ret-1-conds dfs-state*; *invar-well-formed dfs-state*⟧ ⟹ *invar-well-formed* (*DFS-ret1 dfs-state*)
⟨*proof*⟩

lemma *invar-well-formed-holds-5*[*invar-holds-intros*]: ⟦*DFS-ret-2-conds dfs-state*; *invar-well-formed dfs-state*⟧ ⟹ *invar-well-formed* (*DFS-ret2 dfs-state*)
⟨*proof*⟩

lemma *invar-well-formed-holds*[*invar-holds-intros*]:
assumes *DFS-dom dfs-state invar-well-formed dfs-state*
shows *invar-well-formed* (*DFS dfs-state*)
⟨*proof*⟩

lemma *invar-stack-walk-props*[*invar-props-elims*]:
invar-stack-walk dfs-state ⟹
((*Vwalk.vwalk* (*Graph.digraph-abs G*) (*rev* (*stack dfs-state*))) ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *invar-stack-walk-intro*[*invar-props-intros*]: *Vwalk.vwalk* (*Graph.digraph-abs G*) (*rev* (*stack dfs-state*)) ⟹ *invar-stack-walk dfs-state*
⟨*proof*⟩

lemma *invar-stack-walk-holds-1*[*invar-holds-intros*]:
assumes *DFS-call-1-conds dfs-state invar-well-formed dfs-state invar-stack-walk dfs-state*
shows *invar-stack-walk* (*DFS-upd1 dfs-state*)
⟨*proof*⟩

lemma *invar-stack-walk-holds-2*[*invar-holds-intros*]: ⟦*DFS-call-2-conds dfs-state*; *invar-stack-walk dfs-state*⟧ ⟹ *invar-stack-walk* (*DFS-upd2 dfs-state*)
⟨*proof*⟩

lemma *invar-stack-walk-holds-4*[*invar-holds-intros*]: ⟦*DFS-ret-1-conds dfs-state*; *invar-stack-walk dfs-state*⟧ ⟹ *invar-stack-walk* (*DFS-ret1 dfs-state*)
⟨*proof*⟩

lemma *invar-2-holds-5*[*invar-holds-intros*]: ⟦*DFS-ret-2-conds dfs-state*; *invar-stack-walk dfs-state*⟧ ⟹ *invar-stack-walk* (*DFS-ret2 dfs-state*)
⟨*proof*⟩

lemma *invar-2-holds*[*invar-holds-intros*]:
assumes *DFS-dom dfs-state invar-well-formed dfs-state invar-stack-walk dfs-state*
shows *invar-stack-walk* (*DFS dfs-state*)
⟨*proof*⟩

lemma *invar-seen-stack-props*[*invar-props-elims*]:
invar-seen-stack dfs-state ⟹
(⟦*distinct* (*stack dfs-state*); *set* (*stack dfs-state*) ⊆ *t-set* (*seen dfs-state*);
t-set (*seen dfs-state*) ⊆ *dVs* (*Graph.digraph-abs G*)⟧ ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *invar-seen-stack-intro*[*invar-props-intros*]:
⟦*distinct* (*stack dfs-state*); *set* (*stack dfs-state*) ⊆ *t-set* (*seen dfs-state*); *t-set* (*seen dfs-state*) ⊆ *dVs* (*Graph.digraph-abs G*)⟧ ⟹ *invar-seen-stack dfs-state*
⟨*proof*⟩

lemma *invar-seen-stack-holds-1*[*invar-holds-intros*]:
⟦*DFS-call-1-conds dfs-state*; *invar-well-formed dfs-state*; *invar-seen-stack dfs-state*⟧
⟹ *invar-seen-stack* (*DFS-upd1 dfs-state*)
⟨*proof*⟩

lemma *invar-seen-stack-holds-2*[*invar-holds-intros*]:
⟦*DFS-call-2-conds dfs-state*; *invar-well-formed dfs-state*; *invar-seen-stack dfs-state*⟧
⟹
invar-seen-stack (*DFS-upd2 dfs-state*)
⟨*proof*⟩

lemma *invar-seen-stack-holds-4*[*invar-holds-intros*]:
⟦*DFS-ret-1-conds dfs-state*; *invar-seen-stack dfs-state*⟧ ⟹

invar-seen-stack (*DFS-ret1 dfs-state*)
⟨*proof*⟩

lemma *invar-seen-stack-holds-5*[*invar-holds-intros*]: ⟦*DFS-ret-2-conds dfs-state*; *invar-seen-stack dfs-state*⟧ ⟹ *invar-seen-stack* (*DFS-ret2 dfs-state*)
⟨*proof*⟩

lemma *invar-seen-stack-holds*[*invar-holds-intros*]:
assumes *DFS-dom dfs-state invar-well-formed dfs-state invar-seen-stack dfs-state*
shows *invar-seen-stack* (*DFS dfs-state*)
⟨*proof*⟩

lemma *invar-s-in-stack-props*[*invar-props-elims*]:
invar-s-in-stack dfs-state ⟹
(⟦(*stack* (*dfs-state*) ≠ [] ⟹ *last* (*stack dfs-state*) = *s*)⟧ ⟹ *P*) ⟹ *P*
⟨*proof*⟩

lemma *invar-s-in-stack-intro*[*invar-props-intros*]:
⟦(*stack* (*dfs-state*) ≠ [] ⟹ *last* (*stack dfs-state*) = *s*)⟧ ⟹ *invar-s-in-stack dfs-state*
⟨*proof*⟩

lemma *invar-s-in-stack-holds-1*[*invar-holds-intros*]:
⟦*DFS-call-1-conds dfs-state*; *invar-well-formed dfs-state*; *invar-s-in-stack dfs-state*⟧ ⟹ *invar-s-in-stack* (*DFS-upd1 dfs-state*)
⟨*proof*⟩

lemma *invar-s-in-stack-holds-2*[*invar-holds-intros*]:
⟦*DFS-call-2-conds dfs-state*; *invar-well-formed dfs-state*; *invar-s-in-stack dfs-state*⟧ ⟹
invar-s-in-stack (*DFS-upd2 dfs-state*)
⟨*proof*⟩

lemma *invar-s-in-stack-holds-4*[*invar-holds-intros*]:
⟦*DFS-ret-1-conds dfs-state*; *invar-s-in-stack dfs-state*⟧ ⟹
invar-s-in-stack (*DFS-ret1 dfs-state*)
⟨*proof*⟩

lemma *invar-s-in-stack-holds-5*[*invar-holds-intros*]: ⟦*DFS-ret-2-conds dfs-state*; *invar-s-in-stack dfs-state*⟧ ⟹ *invar-s-in-stack* (*DFS-ret2 dfs-state*)
⟨*proof*⟩

lemma *invar-s-in-stack-holds*[*invar-holds-intros*]:
assumes *DFS-dom dfs-state invar-well-formed dfs-state invar-s-in-stack dfs-state*
shows *invar-s-in-stack* (*DFS dfs-state*)
⟨*proof*⟩

lemma *invar-visited-through-seen-props*[*elim*!]:
invar-visited-through-seen dfs-state $\Longrightarrow$
($[\![\bigwedge v\ p.\ [\![v \in$ *t-set* (*seen dfs-state*);
(*Vwalk.vwalk-bet* (*Graph.digraph-abs G*) *v p t*); *distinct p* $]\!] \Longrightarrow$
set p $\cap$ *set* (*stack dfs-state*) $\neq \{\}]\!] \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *invar-visited-through-seen-intro*[*invar-props-intros*]:
$[\![\bigwedge v\ p.\ [\![v \in$ *t-set* (*seen dfs-state*);
(*Vwalk.vwalk-bet* (*Graph.digraph-abs G*) *v p t*); *distinct p*$]\!] \Longrightarrow$
set p $\cap$ *set* (*stack dfs-state*) $\neq \{\}]\!] \Longrightarrow$ *invar-visited-through-seen dfs-state*
⟨*proof*⟩

6.7 Proofs that the Invariants Hold

lemma *invar-visited-through-seen-holds-1*[*invar-holds-intros*]:
$[\![$*DFS-call-1-conds dfs-state*; *invar-well-formed dfs-state*; *invar-seen-stack dfs-state*;
invar-visited-through-seen dfs-state$]\!]$
$\Longrightarrow$ *invar-visited-through-seen* (*DFS-upd1 dfs-state*)
⟨*proof*⟩

lemma *invar-visited-through-seen-holds-2*[*invar-holds-intros*]:
$[\![$*DFS-call-2-conds dfs-state*; *invar-well-formed dfs-state*; *invar-seen-stack dfs-state*;
invar-visited-through-seen dfs-state$]\!] \Longrightarrow$ *invar-visited-through-seen* (*DFS-upd2 dfs-state*)
⟨*proof*⟩

lemma *invar-visited-through-seen-holds-4*[*invar-holds-intros*]: $[\![$*DFS-ret-1-conds dfs-state*; *invar-visited-through-seen dfs-state*$]\!] \Longrightarrow$ *invar-visited-through-seen* (*DFS-ret1 dfs-state*)
⟨*proof*⟩

lemma *invar-visited-through-seen-holds-5*[*invar-holds-intros*]: $[\![$*DFS-ret-2-conds dfs-state*; *invar-visited-through-seen dfs-state*$]\!] \Longrightarrow$ *invar-visited-through-seen* (*DFS-ret2 dfs-state*)
⟨*proof*⟩

lemma *invar-visited-through-seen-holds*[*invar-holds-intros*]:
assumes *DFS-dom dfs-state invar-well-formed dfs-state invar-seen-stack dfs-state*
invar-visited-through-seen dfs-state
shows *invar-visited-through-seen* (*DFS dfs-state*)
⟨*proof*⟩

definition *state-rel-1 dfs-state-1 dfs-state-2*
= (*t-set* (*seen dfs-state-1*) $\subseteq$ *t-set* (*seen dfs-state-2*))

lemma *state-rel-1-props*[*elim*!]: *state-rel-1 dfs-state-1 dfs-state-2* $\Longrightarrow$

$(t\text{-}set\ (seen\ dfs\text{-}state\text{-}1) \subseteq t\text{-}set\ (seen\ dfs\text{-}state\text{-}2) \Longrightarrow P) \Longrightarrow P$
⟨*proof*⟩

lemma *state-rel-1-intro*[*state-rel-intros*]:
⟦*t-set* (*seen dfs-state-1*) ⊆ *t-set* (*seen dfs-state-2*)⟧ ⟹ *state-rel-1 dfs-state-1 dfs-state-2*
⟨*proof*⟩

lemma *state-rel-1-trans*:
⟦*state-rel-1 dfs-state-1 dfs-state-2*; *state-rel-1 dfs-state-2 dfs-state-3*⟧ ⟹
state-rel-1 dfs-state-1 dfs-state-3
⟨*proof*⟩

lemma *state-rel-1-holds-1*[*state-rel-holds-intros*]:
⟦*DFS-call-1-conds dfs-state*; *invar-well-formed dfs-state*⟧ ⟹ *state-rel-1 dfs-state* (*DFS-upd1 dfs-state*)
⟨*proof*⟩

lemma *state-rel-1-holds-2*[*state-rel-holds-intros*]:
⟦*DFS-call-2-conds dfs-state*; *invar-well-formed dfs-state*⟧ ⟹ *state-rel-1 dfs-state* (*DFS-upd2 dfs-state*)
⟨*proof*⟩

lemma *state-rel-1-holds-4*[*state-rel-holds-intros*]:
⟦*DFS-ret-1-conds dfs-state*⟧ ⟹ *state-rel-1 dfs-state* (*DFS-ret1 dfs-state*)
⟨*proof*⟩

lemma *state-rel-1-holds-5*[*state-rel-holds-intros*]:
⟦*DFS-ret-2-conds dfs-state*⟧ ⟹ *state-rel-1 dfs-state* (*DFS-ret2 dfs-state*)
⟨*proof*⟩

lemma *state-rel-1-holds*[*state-rel-holds-intros*]:
assumes *DFS-dom dfs-state invar-well-formed dfs-state*
shows *state-rel-1 dfs-state* (*DFS dfs-state*)
⟨*proof*⟩

lemma *DFS-ret-1*[*ret-holds-intros*]: *DFS-ret-1-conds* (*dfs-state*) ⟹ *DFS-ret-1-conds* (*DFS-ret1 dfs-state*)
⟨*proof*⟩

lemma *ret1-holds*[*ret-holds-intros*]:
assumes *DFS-dom dfs-state return* (*DFS dfs-state*) = *NotReachable*
shows *DFS-ret-1-conds* (*DFS dfs-state*)
⟨*proof*⟩

lemma *DFS-correct-ret-1*:
⟦*invar-visited-through-seen dfs-state*; *DFS-ret-1-conds dfs-state*; *u* ∈ *t-set* (*seen dfs-state*)⟧

$\Longrightarrow \nexists p.\ distinct\ p \wedge vwalk\text{-}bet\ (Graph.digraph\text{-}abs\ G)\ u\ p\ t$
⟨*proof*⟩

lemma *DFS-ret-2*[*ret-holds-intros*]: *DFS-ret-2-conds* (*dfs-state*) $\Longrightarrow$ *DFS-ret-2-conds* (*DFS-ret2 dfs-state*)
⟨*proof*⟩

lemma *ret2-holds*[*ret-holds-intros*]:
assumes *DFS-dom dfs-state return* (*DFS dfs-state*) = *Reachable*
shows *DFS-ret-2-conds* (*DFS dfs-state*)
⟨*proof*⟩

lemma *DFS-correct-ret-2*:
⟦*invar-stack-walk dfs-state*; *DFS-ret-2-conds dfs-state*⟧
$\Longrightarrow$ *vwalk-bet* (*Graph.digraph-abs G*) (*last* (*stack dfs-state*)) (*rev* (*stack dfs-state*)) *t*
⟨*proof*⟩

6.8 Termination

named-theorems *termination-intros*

declare *termination-intros*

lemma *in-prod-relI*[*intro*!,*termination-intros*]:
⟦*f1 a* = *f1 a'*; (*a*, *a'*) ∈ *f2* <∗*mlex*∗> *r*⟧ $\Longrightarrow$ (*a*,*a'*) ∈ (*f1* <∗*mlex*∗> *f2* <∗*mlex*∗> *r*)
⟨*proof*⟩

definition *less-rel* = {(*x*::*nat*, *y*::*nat*). *x* < *y*}

lemma *wf-less-rel*[*intro*!]: *wf less-rel*
⟨*proof*⟩

lemma *call-1-measure-nonsym*[*simp*]: (*call-1-measure dfs-state*, *call-1-measure dfs-state*) ∉ *less-rel*
⟨*proof*⟩

lemma *call-1-terminates*[*termination-intros*]:
⟦*DFS-call-1-conds dfs-state*; *invar-well-formed dfs-state*; *invar-seen-stack dfs-state*⟧
$\Longrightarrow$
(*DFS-upd1 dfs-state*, *dfs-state*) ∈ *call-1-measure* <∗*mlex*∗> *r*
⟨*proof*⟩

lemma *call-2-measure-nonsym*[*simp*]: (*call-2-measure dfs-state*, *call-2-measure dfs-state*) ∉ *less-rel*
⟨*proof*⟩

lemma *call-2-measure-1*[*termination-intros*]:

⟦*DFS-call-2-conds dfs-state*; *invar-well-formed dfs-state*⟧ ⟹
call-1-measure dfs-state = *call-1-measure* (*DFS-upd2 dfs-state*)
⟨*proof*⟩

lemma *call-2-terminates*[*termination-intros*]:
⟦*DFS-call-2-conds dfs-state*; *invar-well-formed dfs-state*; *invar-seen-stack dfs-state*⟧
⟹
(*DFS-upd2 dfs-state*, *dfs-state*) ∈ *call-2-measure* <∗*mlex*∗> *r*
⟨*proof*⟩

lemma *wf-term-rel*: *wf DFS-term-rel*
⟨*proof*⟩

lemma *in-DFS-term-rel*[*termination-intros*]:
⟦*DFS-call-1-conds dfs-state*; *invar-well-formed dfs-state*; *invar-seen-stack dfs-state*⟧
⟹
(*DFS-upd1 dfs-state*, *dfs-state*) ∈ *DFS-term-rel*
⟦*DFS-call-2-conds dfs-state*; *invar-well-formed dfs-state*; *invar-seen-stack dfs-state*⟧
⟹
(*DFS-upd2 dfs-state*, *dfs-state*) ∈ *DFS-term-rel*
⟨*proof*⟩

lemma *DFS-terminates*[*termination-intros*]:
assumes *invar-well-formed dfs-state invar-seen-stack dfs-state*
shows *DFS-dom dfs-state*
⟨*proof*⟩

6.9 Final Correctness Theorems

lemma *initial-state-props*[*invar-holds-intros*, *termination-intros*]:
invar-well-formed (*initial-state*) *invar-stack-walk* (*initial-state*) *invar-seen-stack* (*initial-state*)
invar-visited-through-seen (*initial-state*) *invar-s-in-stack initial-state*
DFS-dom initial-state
⟨*proof*⟩

lemma *DFS-correct-1*:
assumes *return* (*DFS initial-state*) = *NotReachable*
shows ∄*p*. *distinct p* ∧ *vwalk-bet* (*Graph.digraph-abs G*) *s p t*
⟨*proof*⟩

lemma *DFS-correct-2*:
assumes *return* (*DFS initial-state*) = *Reachable*
shows *vwalk-bet* (*Graph.digraph-abs G*) *s* (*rev* (*stack* (*DFS initial-state*))) *t*
⟨*proof*⟩
end
end

end

theory *BFS-2*
imports *Pair-Graph-Specs Dist Set2-Addons More-Lists*
begin

7 Breadth-First Search

7.1 The Program State

record (*'parents*, *'vset*) *BFS-state* = *parents*:: *'parents current*:: *'vset visited*:: *'vset*

7.2 Setup for Automation

named-theorems *call-cond-elims*
named-theorems *call-cond-intros*
named-theorems *ret-holds-intros*
named-theorems *invar-props-intros*
named-theorems *invar-props-elims*
named-theorems *invar-holds-intros*
named-theorems *state-rel-intros*
named-theorems *state-rel-holds-intros*

7.3 A *locale* for fixing data structures and their implemenations

locale *BFS* =
Graph: *Pair-Graph-Specs*
where *lookup* = *lookup* +
set-ops: *Set2 vset-empty vset-delete - t-set vset-inv insert*

for *lookup* :: *'adjmap* ⇒ *'ver* ⇒ *'vset option* +

fixes
srcs::*'vset* **and**
G::*'adjmap* **and** *expand-tree*::*'adjmap* ⇒ *'vset* ⇒ *'vset* ⇒ *'adjmap* **and**
next-frontier::*'vset* ⇒ *'vset* ⇒ *'vset*

assumes
expand-tree[*simp*]:
⟦*Graph.graph-inv BFS-tree*; *vset-inv frontier*; *vset-inv vis*; *Graph.graph-inv G*⟧
⟹
Graph.graph-inv (*expand-tree BFS-tree frontier vis*)
⟦*Graph.graph-inv BFS-tree*; *vset-inv frontier*; *vset-inv vis*; *Graph.graph-inv G*⟧
⟹
Graph.digraph-abs (*expand-tree BFS-tree frontier vis*) =
(*Graph.digraph-abs BFS-tree*) ∪
{(*u*,*v*) | *u v*. *u* ∈ *t-set* (*frontier*) ∧
v ∈ (*Pair-Graph.neighbourhood* (*Graph.digraph-abs G*) *u* −
t-set vis)} **and**

next-frontier[*simp*]:
⟦*vset-inv frontier*; *vset-inv vis*; *Graph.graph-inv G*⟧ ⟹ *vset-inv* (*next-frontier frontier vis*)
⟦*vset-inv frontier*; *vset-inv vis*; *Graph.graph-inv G*⟧ ⟹
t-set (*next-frontier frontier vis*) =
(⋃ {*Pair-Graph.neighbourhood* (*Graph.digraph-abs G*) *u* | *u* . *u* ∈ *t-set frontier*}) − *t-set vis*

begin

definition *BFS-axiom* ⟷
Graph.graph-inv G ∧ *Graph.finite-graph G* ∧ *Graph.finite-vsets* ∧
t-set srcs ⊆ *dVs* (*Graph.digraph-abs G*) ∧
(∀ *u*. *finite* (*Pair-Graph.neighbourhood* (*Graph.digraph-abs G*) *u*)) ∧
t-set srcs ≠ {} ∧ *vset-inv srcs*

abbreviation *neighb′* ≡ *Graph.neighb G*
notation *neighb′* (*$\mathcal{N}_G$* - *100*)

7.4 The Algorithm Definition

function (*domintros*) *BFS*::('*adjmap*, '*vset*) *BFS-state* ⇒ ('*adjmap*, '*vset*) *BFS-state*
where
BFS BFS-state =
(
if current BFS-state ≠ *$\emptyset_V$ then*
let
visited′ = *visited BFS-state* ∪*$_G$ current BFS-state*;
parents′ = *expand-tree* (*parents BFS-state*) (*current BFS-state*) *visited′*;
current′ = *next-frontier* (*current BFS-state*) *visited′*
in
BFS (*BFS-state* (|*parents*:= *parents′*, *visited* := *visited′*, *current* := *current′*|))
else
BFS-state)
⟨*proof*⟩

7.5 Setup for Reasoning About the Algorithm

definition *BFS-call-1-conds bfs-state* = ((*current bfs-state*) ≠ *$\emptyset_V$*)

definition *BFS-upd1 BFS-state* =
(*let*
visited′ = *visited BFS-state* ∪*$_G$ current BFS-state*;
parents′ = *expand-tree* (*parents BFS-state*) (*current BFS-state*) *visited′*;
current′ = *next-frontier* (*current BFS-state*) *visited′*
in
BFS-state (|*parents*:= *parents′*, *visited* := *visited′*, *current* := *current′*|)

)

definition *BFS-ret-1-conds bfs-state* = ((*current bfs-state*) = $\emptyset_V$)

abbreviation *BFS-ret1 bfs-state* ≡ *bfs-state*

lemma *DFS-call-1-conds*[*call-cond-elims*]:
BFS-call-1-conds bfs-state ⟹
⟦(*current bfs-state*) ≠ $\emptyset_V$ ⟹ *P*⟧
⟹ *P*
⟨*proof*⟩

lemma *BFS-ret-1-conds*[*call-cond-elims*]:
BFS-ret-1-conds bfs-state ⟹
⟦(*current bfs-state*) = $\emptyset_V$ ⟹ *P*⟧
⟹ *P*
⟨*proof*⟩

lemma *BFS-ret-1-condsI*[*call-cond-intros*]:
⟦(*current bfs-state*) = $\emptyset_V$⟧ ⟹ *BFS-ret-1-conds bfs-state*
⟨*proof*⟩

lemma *BFS-cases*:
assumes *BFS-call-1-conds bfs-state* ⟹ *P*
BFS-ret-1-conds bfs-state ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *BFS-simps*:
assumes *BFS-dom BFS-state*
shows*BFS-call-1-conds BFS-state* ⟹ *BFS BFS-state* = *BFS* (*BFS-upd1 BFS-state*)
BFS-ret-1-conds BFS-state ⟹ *BFS BFS-state* = *BFS-ret1 BFS-state*
⟨*proof*⟩

lemma *BFS-induct*:
assumes *BFS-dom bfs-state*
assumes ⋀*bfs-state*. ⟦*BFS-dom bfs-state*;
(*BFS-call-1-conds bfs-state* ⟹ *P* (*BFS-upd1 bfs-state*))⟧
⟹ *P bfs-state*
shows *P bfs-state*
⟨*proof*⟩

lemma *BFS-domintros*:
assumes *BFS-call-1-conds BFS-state* ⟹ *BFS-dom* (*BFS-upd1 BFS-state*)
shows *BFS-dom BFS-state*
⟨*proof*⟩

7.6 The Loop Invariants

definition *invar-well-formed*::('*adjmap*, '*vset*) *BFS-state* ⇒ *bool* **where**
invar-well-formed bfs-state = (
vset-inv (*visited bfs-state*) ∧ *vset-inv* (*current bfs-state*) ∧
Graph.graph-inv (*parents bfs-state*) ∧
finite (*t-set* (*current bfs-state*)) ∧ *finite* (*t-set* (*visited bfs-state*)))

definition *invar-subsets*::('*adjmap*, '*vset*) *BFS-state* ⇒ *bool* **where**
invar-subsets bfs-state = (
Graph.digraph-abs (*parents bfs-state*) ⊆ *Graph.digraph-abs G* ∧
t-set (*visited bfs-state*) ⊆ *dVs* (*Graph.digraph-abs G*) ∧
t-set (*current bfs-state*) ⊆ *dVs* (*Graph.digraph-abs G*) ∧
dVs (*Graph.digraph-abs* (*parents bfs-state*)) ⊆ *t-set* (*visited bfs-state*) ∪ *t-set* (*current bfs-state*) ∧
t-set srcs ⊆ *t-set* (*visited bfs-state*) ∪ *t-set* (*current bfs-state*))

definition *invar-3-1 bfs-state* =
(∀ *v*∈*t-set* (*current bfs-state*). ∀ *u*. *u* ∈ *t-set* (*current bfs-state*) ⟷
distance-set (*Graph.digraph-abs G*) (*t-set srcs*) *v* =
distance-set (*Graph.digraph-abs G*) (*t-set srcs*) *u*)

definition *invar-3-2 bfs-state* =
(∀ *v*∈*t-set* (*current bfs-state*). ∀ *u* ∈ *t-set* (*visited bfs-state*) ∪ *t-set* (*current bfs-state*).
distance-set (*Graph.digraph-abs G*) (*t-set srcs*) *u* ≤
distance-set (*Graph.digraph-abs G*) (*t-set srcs*) *v*)

definition *invar-3-3 bfs-state* =
(∀ *v*∈*t-set* (*visited bfs-state*).
neighbourhood (*Graph.digraph-abs G*) *v* ⊆ *t-set* (*visited bfs-state*) ∪ *t-set* (*current bfs-state*))

definition *invar-dist-bounded*::('*adjmap*, '*vset*) *BFS-state* ⇒ *bool* **where**
invar-dist-bounded bfs-state =
(∀ *v*∈ *t-set* (*visited bfs-state*) ∪ *t-set* (*current bfs-state*).
∀ *u*. *distance-set* (*Graph.digraph-abs G*) (*t-set srcs*) *u* ≤
distance-set (*Graph.digraph-abs G*) (*t-set srcs*) *v*
⟶ *u* ∈ *t-set* (*visited bfs-state*) ∪ *t-set* (*current bfs-state*))

definition *invar-goes-through-current*::('*adjmap*, '*vset*) *BFS-state* ⇒ *bool* **where**
invar-goes-through-current bfs-state =
(∀ *u*∈*t-set* (*visited bfs-state*) ∪ *t-set* (*current bfs-state*).
∀ *v*. *v* ∉ *t-set* (*visited bfs-state*) ∪ *t-set* (*current bfs-state*) ⟶
(∀ *p*. *Vwalk.vwalk-bet* (*Graph.digraph-abs G*) *u p v* ⟶
set p ∩ *t-set* (*current bfs-state*) ≠ {}))

definition *invar-dist*::('*adjmap*, '*vset*) *BFS-state* ⇒ *bool* **where**
invar-dist bfs-state =
(∀ *v* ∈ *dVs* (*Graph.digraph-abs G*) − *t-set srcs*.
(*v* ∈ (*t-set* (*visited bfs-state*) ∪ *t-set* (*current bfs-state*)) ⟶ *distance-set*

(*Graph.digraph-abs G*) (*t-set srcs*) *v* =
distance-set (*Graph.digraph-abs* (*parents bfs-state*)) (*t-set srcs*) *v*))

definition *invar-parents-shortest-paths*::('*adjmap*, '*vset*) *BFS-state* ⇒ *bool* **where**
invar-parents-shortest-paths bfs-state =
(∀ *u*∈*t-set srcs*.
∀ *p v*. *Vwalk.vwalk-bet* (*Graph.digraph-abs* (*parents bfs-state*)) *u p v* ⟶
length p − *1* = *distance-set* (*Graph.digraph-abs G*) (*t-set srcs*) *v*)

7.7 Termination Measures and Relation

definition *call-1-measure-1*::('*adjmap*, '*vset*) *BFS-state* ⇒ *nat* **where**
call-1-measure-1 bfs-state=
card (*dVs* (*Graph.digraph-abs G*) − ((*t-set* (*visited bfs-state*)) ∪ *t-set* (*current bfs-state*)))

definition *call-1-measure-2*::('*adjmap*, '*vset*) *BFS-state* ⇒ *nat* **where**
call-1-measure-2 bfs-state =
card (*t-set* (*current bfs-state*))

definition *BFS-term-rel*::(('*adjmap*, '*vset*) *BFS-state* × ('*adjmap*, '*vset*) *BFS-state*) *set* **where**
BFS-term-rel = *call-1-measure-1* <∗*mlex*∗> *call-1-measure-2* <∗*mlex*∗> {}

definition *initial-state* = (|*parents* = *empty*, *current* = *srcs*, *visited* = $\emptyset_V$|)

lemmas[*code*] = *initial-state-def*

context
includes *Graph.adjmap.automation* **and** *Graph.vset.set.automation* **and** *set-ops.automation*
assumes *BFS-axiom*
begin

lemma *graph-inv*[*simp*]:
Graph.graph-inv G
Graph.finite-graph G
Graph.finite-vsets **and**
srcs-in-G[*simp*,*intro*]:
t-set srcs ⊆ *dVs* (*Graph.digraph-abs G*) **and**
finite-vset:
finite (*Pair-Graph.neighbourhood* (*Graph.digraph-abs G*) *u*) **and**
srcs-invar[*simp*]:
t-set srcs ≠ {}
vset-inv srcs
⟨*proof*⟩

lemma *invar-well-formed-props*[*invar-props-elims*]:
invar-well-formed bfs-state ⟹
([|*vset-inv* (*visited bfs-state*) ; *vset-inv* (*current bfs-state*) ;

Graph.graph-inv (*parents bfs-state*);
finite (*t-set* (*current bfs-state*)); *finite* (*t-set* (*visited bfs-state*))⟧ ⟹ *P*)
⟹ *P*
⟨*proof*⟩

lemma *invar-well-formed-intro*[*invar-props-intros*]:
⟦*vset-inv* (*visited bfs-state*); *vset-inv* (*current bfs-state*);
Graph.graph-inv (*parents bfs-state*);
finite (*t-set* (*current bfs-state*)); *finite* (*t-set* (*visited bfs-state*))⟧
⟹ *invar-well-formed bfs-state*
⟨*proof*⟩

lemma *finite-simp*:
{(*u*,*v*). *u* ∈ *front* ∧ *v* ∈ (*Pair-Graph.neighbourhood* (*Graph.digraph-abs G*) *u*) ∧ *v* ∉ *vis*} =
{(*u*,*v*). *u* ∈ *front*} ∩ {(*u*,*v*). *v* ∈ (*Pair-Graph.neighbourhood* (*Graph.digraph-abs G*) *u*)} − {(*u*,*v*) . *v* ∈ *vis*}
finite {(*u*, *v*)| *v* . *v* ∈ (*s u*)} = *finite* (*s u*)
⟨*proof*⟩

lemma *invar-well-formed-holds-upd1*[*invar-holds-intros*]:
⟦*BFS-call-1-conds bfs-state*; *invar-well-formed bfs-state*⟧ ⟹ *invar-well-formed* (*BFS-upd1 bfs-state*)
⟨*proof*⟩

lemma *invar-well-formed-holds-ret-1*[*invar-holds-intros*]:
⟦*BFS-ret-1-conds bfs-state*; *invar-well-formed bfs-state*⟧ ⟹ *invar-well-formed* (*BFS-ret1 bfs-state*)
⟨*proof*⟩

lemma *invar-well-formed-holds*[*invar-holds-intros*]:
assumes *BFS-dom bfs-state invar-well-formed bfs-state*
shows *invar-well-formed* (*BFS bfs-state*)
⟨*proof*⟩

lemma *invar-subsets-props*[*invar-props-elims*]:
invar-subsets bfs-state ⟹
(⟦*Graph.digraph-abs* (*parents bfs-state*) ⊆ *Graph.digraph-abs G*;
t-set (*visited bfs-state*) ⊆ *dVs* (*Graph.digraph-abs G*);
t-set (*current bfs-state*) ⊆ *dVs* (*Graph.digraph-abs G*);
dVs (*Graph.digraph-abs* (*parents bfs-state*)) ⊆ *t-set* (*visited bfs-state*) ∪ *t-set* (*current bfs-state*);
t-set srcs ⊆ *t-set* (*visited bfs-state*) ∪ *t-set* (*current bfs-state*)⟧ ⟹ *P*)
⟹ *P*
⟨*proof*⟩

lemma *invar-subsets-intro*[*invar-props-intros*]:
⟦*Graph.digraph-abs* (*parents bfs-state*) ⊆ *Graph.digraph-abs G*;
t-set (*visited bfs-state*) ⊆ *dVs* (*Graph.digraph-abs G*);

t-set (*current bfs-state*) ⊆ *dVs* (*Graph.digraph-abs G*);
dVs (*Graph.digraph-abs* (*parents bfs-state*)) ⊆ *t-set* (*visited bfs-state*) ∪ *t-set* (*current bfs-state*);
t-set srcs ⊆ *t-set* (*visited bfs-state*) ∪ *t-set* (*current bfs-state*)⟧
⟹ *invar-subsets bfs-state*
⟨*proof*⟩

lemma *invar-subsets-holds-upd1*[*invar-holds-intros*]:
⟦*BFS-call-1-conds bfs-state*; *invar-well-formed bfs-state*; *invar-subsets bfs-state*⟧ ⟹ *invar-subsets* (*BFS-upd1 bfs-state*)
⟨*proof*⟩

lemma *invar-subsets-holds-ret-1*[*invar-holds-intros*]:
⟦*BFS-ret-1-conds bfs-state*; *invar-subsets bfs-state*⟧ ⟹ *invar-subsets* (*BFS-ret1 bfs-state*)
⟨*proof*⟩

lemma *invar-subsets-holds*[*invar-holds-intros*]:
assumes *BFS-dom bfs-state invar-well-formed bfs-state invar-subsets bfs-state*
shows *invar-subsets* (*BFS bfs-state*)
⟨*proof*⟩

lemma *invar-3-1-props*[*invar-props-elims*]:
invar-3-1 bfs-state ⟹
(⟦⟦*v* ∈ *t-set* (*current bfs-state*); *u* ∈ *t-set* (*current bfs-state*)⟧ ⟹
distance-set (*Graph.digraph-abs G*) (*t-set srcs*) *v* =
distance-set (*Graph.digraph-abs G*) (*t-set srcs*) *u*;
⟦*v* ∈ *t-set* (*current bfs-state*);
distance-set (*Graph.digraph-abs G*) (*t-set srcs*) *v* =
distance-set (*Graph.digraph-abs G*) (*t-set srcs*) *u*⟧ ⟹
u ∈ *t-set* (*current bfs-state*)⟧ ⟹ *P*)
⟹ *P*
⟨*proof*⟩

lemma *invar-3-1-intro*[*invar-props-intros*]:
⟦⋀*u v*. ⟦*v* ∈ *t-set* (*current bfs-state*); *u* ∈ *t-set* (*current bfs-state*)⟧ ⟹
distance-set (*Graph.digraph-abs G*) (*t-set srcs*) *v* =
distance-set (*Graph.digraph-abs G*) (*t-set srcs*) *u*;
⋀*u v*. ⟦*v* ∈ *t-set* (*current bfs-state*); *distance-set* (*Graph.digraph-abs G*) (*t-set srcs*) *v* =
distance-set (*Graph.digraph-abs G*) (*t-set srcs*) *u*⟧ ⟹
u ∈ *t-set* (*current bfs-state*)⟧
⟹ *invar-3-1 bfs-state*
⟨*proof*⟩

lemma *invar-3-2-props*[*elim*]:
invar-3-2 bfs-state ⟹
(⟦⋀*v u*. ⟦*v*∈*t-set* (*current bfs-state*); *u* ∈ *t-set* (*visited bfs-state*) ∪ *t-set* (*current bfs-state*)⟧ ⟹

distance-set (*Graph.digraph-abs G*) (*t-set srcs*) *u* ≤
distance-set (*Graph.digraph-abs G*) (*t-set srcs*) *v*⟧ ⟹ *P*)
⟹ *P*
⟨*proof*⟩

lemma *invar-3-2-intro*[*invar-props-intros*]:
⟦⋀*v u*. ⟦*v*∈*t-set* (*current bfs-state*); *u* ∈ *t-set* (*visited bfs-state*) ∪ *t-set* (*current bfs-state*)⟧ ⟹
distance-set (*Graph.digraph-abs G*) (*t-set srcs*) *u* ≤
distance-set (*Graph.digraph-abs G*) (*t-set srcs*) *v*⟧
⟹ *invar-3-2 bfs-state*
⟨*proof*⟩

lemma *invar-3-3-props*[*invar-props-elims*]:
invar-3-3 bfs-state ⟹
(⟦⋀*v*. ⟦*v* ∈ *t-set* (*visited bfs-state*)⟧ ⟹
neighbourhood (*Graph.digraph-abs G*) *v* ⊆ *t-set* (*visited bfs-state*) ∪ *t-set* (*current bfs-state*)⟧ ⟹ *P*)
⟹ *P*
⟨*proof*⟩

lemma *invar-3-3-intro*[*invar-props-intros*]:
⟦⋀*v*. ⟦*v* ∈ *t-set* (*visited bfs-state*)⟧ ⟹
neighbourhood (*Graph.digraph-abs G*) *v* ⊆ *t-set* (*visited bfs-state*) ∪ *t-set* (*current bfs-state*)⟧
⟹ *invar-3-3 bfs-state*
⟨*proof*⟩

lemma *invar-dist-bounded-props*[*invar-props-elims*]:
invar-dist-bounded bfs-state ⟹
(⟦⋀*u v*. ⟦*v*∈ *t-set* (*visited bfs-state*) ∪ *t-set* (*current bfs-state*);
distance-set (*Graph.digraph-abs G*) (*t-set srcs*) *u* ≤ *distance-set* (*Graph.digraph-abs G*) (*t-set srcs*) *v*⟧ ⟹
u ∈ *t-set* (*visited bfs-state*) ∪ *t-set* (*current bfs-state*)⟧ ⟹ *P*)
⟹ *P*
⟨*proof*⟩

lemma *invar-dist-bounded-intro*[*invar-props-intros*]:
⟦⋀*u v*. ⟦*v*∈ *t-set* (*visited bfs-state*) ∪ *t-set* (*current bfs-state*);
distance-set (*Graph.digraph-abs G*) (*t-set srcs*) *u* ≤ *distance-set* (*Graph.digraph-abs G*) (*t-set srcs*) *v*⟧ ⟹
u ∈ *t-set* (*visited bfs-state*) ∪ *t-set* (*current bfs-state*)⟧
⟹ *invar-dist-bounded bfs-state*
⟨*proof*⟩

definition *invar-current-reachable bfs-state* =
(∀ *v*∈ *t-set* (*visited bfs-state*) ∪ *t-set* (*current bfs-state*).
distance-set (*Graph.digraph-abs G*) (*t-set srcs*) *v* < ∞)

lemma *invar-current-reachable-props*[*invar-props-elims*]:
invar-current-reachable bfs-state $\Longrightarrow$
($[\![\bigwedge v.\ [\![v \in$ *t-set* (*visited bfs-state*) $\cup$ *t-set* (*current bfs-state*)$]\!] \Longrightarrow$
distance-set (*Graph.digraph-abs G*) (*t-set srcs*) $v < \infty]\!] \Longrightarrow P$)
$\Longrightarrow P$
⟨*proof*⟩

lemma *invar-current-reachable-intro*[*invar-props-intros*]:
$[\![\bigwedge v.\ [\![v \in$ *t-set* (*visited bfs-state*) $\cup$ *t-set* (*current bfs-state*)$]\!] \Longrightarrow$
distance-set (*Graph.digraph-abs G*) (*t-set srcs*) $v < \infty]\!]$
$\Longrightarrow$ *invar-current-reachable bfs-state*
⟨*proof*⟩

7.8 Proofs that the Invariants Hold

lemma *invar-current-reachable-holds-upd1*[*invar-holds-intros*]:
$[\![$*BFS-call-1-conds bfs-state*; *invar-well-formed bfs-state*; *invar-subsets bfs-state*; *invar-current-reachable bfs-state*$]\!]$
$\Longrightarrow$ *invar-current-reachable* (*BFS-upd1 bfs-state*)
⟨*proof*⟩

lemma *invar-current-reachable-holds-ret-1*[*invar-holds-intros*]:
$[\![$*BFS-ret-1-conds bfs-state*; *invar-current-reachable bfs-state*$]\!] \Longrightarrow$ *invar-current-reachable* (*BFS-ret1 bfs-state*)
⟨*proof*⟩

lemma *dist-current-plus-1-new*:
assumes
invar-well-formed bfs-state invar-subsets bfs-state invar-dist-bounded bfs-state
$v \in$ *neighbourhood* (*Graph.digraph-abs G*) v'
$v' \in$ *t-set* (*current bfs-state*)
$v \in$ *t-set* (*current* (*BFS-upd1 bfs-state*))
shows *distance-set* (*Graph.digraph-abs G*) (*t-set srcs*) $v =$
distance-set (*Graph.digraph-abs G*) (*t-set srcs*) $v' + 1$ (**is** *?dv* = *?dv'* + *1*)
⟨*proof*⟩

lemma *plus-lt-enat*: $[\![(a::enat) \neq \infty;\ b < c]\!] \Longrightarrow a + b < a + c$
⟨*proof*⟩

lemma *plus-one-side-lt-enat*: $[\![(a::enat) \neq \infty;\ 0 < b]\!] \Longrightarrow a < a + b$
⟨*proof*⟩

lemma *invar-3-1-holds-upd1-new*[*invar-holds-intros*]:
$[\![$*BFS-call-1-conds bfs-state*; *invar-well-formed bfs-state*; *invar-subsets bfs-state* ; *invar-3-1 bfs-state*;
invar-3-2 bfs-state; *invar-dist-bounded bfs-state*; *invar-current-reachable bfs-state*$]\!]$
$\Longrightarrow$ *invar-3-1* (*BFS-upd1 bfs-state*)
⟨*proof*⟩

lemma *invar-3-1-holds-ret-1*[*invar-holds-intros*]:
⟦*BFS-ret-1-conds bfs-state*; *invar-3-3 bfs-state*⟧ ⟹ *invar-3-3* (*BFS-ret1 bfs-state*)
⟨*proof*⟩

lemma *invar-3-2-holds-upd1-new*[*invar-holds-intros*]:
⟦*BFS-call-1-conds bfs-state*; *invar-well-formed bfs-state*; *invar-subsets bfs-state* ; *invar-3-1 bfs-state*;
invar-3-2 bfs-state; *invar-dist-bounded bfs-state*; *invar-current-reachable bfs-state*⟧
⟹ *invar-3-2* (*BFS-upd1 bfs-state*)
⟨*proof*⟩

lemma *invar-3-2-holds-ret-1*[*invar-holds-intros*]:
⟦*BFS-ret-1-conds bfs-state*; *invar-3-3 bfs-state*⟧ ⟹ *invar-3-3* (*BFS-ret1 bfs-state*)
⟨*proof*⟩

lemma *invar-3-3-holds-upd1*[*invar-holds-intros*]:
⟦*BFS-call-1-conds bfs-state*; *invar-well-formed bfs-state*; *invar-subsets bfs-state*; *invar-3-3 bfs-state*⟧ ⟹ *invar-3-3* (*BFS-upd1 bfs-state*)
⟨*proof*⟩

lemma *invar-3-3-holds-ret-1*[*invar-holds-intros*]:
⟦*BFS-ret-1-conds bfs-state*; *invar-3-3 bfs-state*⟧ ⟹ *invar-3-3* (*BFS-ret1 bfs-state*)
⟨*proof*⟩

lemma *invar-dist-bounded-holds-upd1*[*invar-holds-intros*]:
⟦*BFS-call-1-conds bfs-state*; *invar-well-formed bfs-state*; *invar-subsets bfs-state*;
invar-3-1 bfs-state; *invar-3-2 bfs-state*; *invar-3-3 bfs-state*; *invar-dist-bounded bfs-state*;
invar-current-reachable bfs-state⟧ ⟹
invar-dist-bounded (*BFS-upd1 bfs-state*)
⟨*proof*⟩

lemma *invar-dist-bounded-holds-ret-1*[*invar-holds-intros*]:
⟦*BFS-ret-1-conds bfs-state*; *invar-dist-bounded bfs-state*⟧ ⟹ *invar-dist-bounded* (*BFS-ret1 bfs-state*)
⟨*proof*⟩

lemma *invar-dist-props*[*invar-props-elims*]:
invar-dist bfs-state ⟹ *v* ∈ *dVs* (*Graph.digraph-abs G*) − *t-set srcs* ⟹
⟦
⟦*v* ∈ (*t-set* (*visited bfs-state*) ∪ *t-set* (*current bfs-state*)) ⟹ *distance-set* (*Graph.digraph-abs G*) (*t-set srcs*) *v* =
distance-set (*Graph.digraph-abs* (*parents bfs-state*)) (*t-set srcs*) *v*⟧ ⟹ *P*
⟧
⟹ *P*
⟨*proof*⟩

lemma *invar-dist-intro*[*invar-props-intros*]:
⟦⋀*v*. ⟦*v* ∈ *dVs* (*Graph.digraph-abs G*) − *t-set srcs*; *v* ∈ *t-set* (*visited bfs-state*) ∪ *t-set* (*current bfs-state*)⟧ ⟹
(*distance-set* (*Graph.digraph-abs G*) (*t-set srcs*) *v* =
distance-set (*Graph.digraph-abs* (*parents bfs-state*)) (*t-set srcs*) *v*)⟧

⟹ *invar-dist bfs-state*
⟨*proof*⟩

lemma *invar-goes-through-current-props*[*invar-props-elims*]:
invar-goes-through-current bfs-state ⟹
⟦⟦⋀*u v p*. ⟦*u* ∈*t-set* (*visited bfs-state*) ∪ *t-set* (*current bfs-state*);
v ∉ *t-set* (*visited bfs-state*) ∪ *t-set* (*current bfs-state*);
Vwalk.vwalk-bet (*Graph.digraph-abs G*) *u p v*⟧
⟹ *set p* ∩ *t-set* (*current bfs-state*) ≠ {}⟧
⟹ *P*⟧
⟹ *P*
⟨*proof*⟩

lemma *invar-goes-through-current-intro*[*invar-props-intros*]:
⟦⋀*u v p*. ⟦*u* ∈*t-set* (*visited bfs-state*) ∪ *t-set* (*current bfs-state*);
v ∉ *t-set* (*visited bfs-state*) ∪ *t-set* (*current bfs-state*);
Vwalk.vwalk-bet (*Graph.digraph-abs G*) *u p v*⟧
⟹ *set p* ∩ *t-set* (*current bfs-state*) ≠ {}⟧
⟹ *invar-goes-through-current bfs-state*
⟨*proof*⟩

lemma *invar-goes-through-active-holds-upd1*[*invar-holds-intros*]:
⟦*BFS-call-1-conds bfs-state*; *invar-well-formed bfs-state*; *invar-subsets bfs-state*;
invar-goes-through-current bfs-state⟧
⟹ *invar-goes-through-current* (*BFS-upd1 bfs-state*)
⟨*proof*⟩

lemma *invar-goes-through-current-holds-ret-1*[*invar-holds-intros*]:
⟦*BFS-ret-1-conds bfs-state*; *invar-goes-through-current bfs-state*⟧ ⟹ *invar-goes-through-current* (*BFS-ret1 bfs-state*)
⟨*proof*⟩

lemma *invar-goes-through-current-holds*[*invar-holds-intros*]:
assumes *BFS-dom bfs-state invar-well-formed bfs-state invar-subsets bfs-state*
invar-goes-through-current bfs-state
shows *invar-goes-through-current* (*BFS bfs-state*)
⟨*proof*⟩

lemma *invar-dist-holds-upd1-new*[*invar-holds-intros*]:
⟦*BFS-call-1-conds bfs-state*; *invar-well-formed bfs-state*; *invar-subsets bfs-state*;
invar-dist-bounded bfs-state; *invar-dist bfs-state*⟧
⟹ *invar-dist* (*BFS-upd1 bfs-state*)

⟨*proof*⟩

lemma *invar-dist-holds-ret-1*[*invar-holds-intros*]:
⟦*BFS-ret-1-conds bfs-state*; *invar-dist bfs-state*⟧ ⟹ *invar-dist* (*BFS-ret1 bfs-state*)
⟨*proof*⟩

lemma *invar-dist-holds*[*invar-holds-intros*]:
assumes *BFS-dom bfs-state invar-well-formed bfs-state invar-subsets bfs-state invar-3-1 bfs-state invar-3-2 bfs-state invar-3-3 bfs-state invar-dist-bounded bfs-state invar-dist bfs-state invar-current-reachable bfs-state*
shows *invar-dist* (*BFS bfs-state*)
⟨*proof*⟩

definition *invar-current-no-out bfs-state* =
(∀ *u*∈*t-set*(*current bfs-state*).
∀ *v*. (*u*,*v*) ∉ *Graph.digraph-abs* (*parents bfs-state*))

lemma *invar-current-no-out-props*[*invar-props-elims*]:
invar-current-no-out bfs-state ⟹
(⟦⋀*u v*. *u*∈*t-set*(*current bfs-state*) ⟹ (*u*,*v*) ∉ *Graph.digraph-abs* (*parents bfs-state*)⟧ ⟹ *P*)
⟹ *P*
⟨*proof*⟩

lemma *invar-current-no-out-intro*[*invar-props-intros*]:
⟦⋀*u v*. *u*∈*t-set*(*current bfs-state*) ⟹ (*u*,*v*) ∉ *Graph.digraph-abs* (*parents bfs-state*)⟧
⟹ *invar-current-no-out bfs-state*
⟨*proof*⟩

lemma *invar-current-no-out-holds-upd1*[*invar-holds-intros*]:
⟦*BFS-call-1-conds bfs-state*; *invar-well-formed bfs-state*; *invar-subsets bfs-state*; *invar-current-no-out bfs-state*⟧
⟹ *invar-current-no-out* (*BFS-upd1 bfs-state*)
⟨*proof*⟩

lemma *invar-current-no-out-holds-ret-1*[*invar-holds-intros*]:
⟦*BFS-ret-1-conds bfs-state*; *invar-current-no-out bfs-state*⟧ ⟹ *invar-current-no-out* (*BFS-ret1 bfs-state*)
⟨*proof*⟩

lemma *invar-current-no-out-holds*[*invar-holds-intros*]:
assumes *BFS-dom bfs-state invar-well-formed bfs-state invar-subsets bfs-state invar-current-no-out bfs-state*
shows *invar-current-no-out* (*BFS bfs-state*)
⟨*proof*⟩

lemma *invar-parents-shortest-paths-props*[*invar-props-elims*]:
invar-parents-shortest-paths bfs-state ⟹

(⟦⋀u p v. ⟦u ∈ t-set srcs; Vwalk.vwalk-bet (Graph.digraph-abs (parents bfs-state)) u p v⟧ ⟹
length p − 1 = distance-set (Graph.digraph-abs G) (t-set srcs) v⟧ ⟹ P)
⟹ P
⟨proof⟩

lemma *invar-parents-shortest-paths-intro*[*invar-props-intros*]:
⟦⋀u p v. ⟦u ∈ t-set srcs; Vwalk.vwalk-bet (Graph.digraph-abs (parents bfs-state)) u p v⟧ ⟹
length p − 1 = distance-set (Graph.digraph-abs G) (t-set srcs) v⟧
⟹ invar-parents-shortest-paths bfs-state
⟨proof⟩

lemma *invar-parents-shortest-paths-holds-upd1*[*invar-holds-intros*]:
⟦BFS-call-1-conds bfs-state; invar-well-formed bfs-state; invar-subsets bfs-state; invar-current-no-out bfs-state;
invar-dist-bounded bfs-state; invar-parents-shortest-paths bfs-state⟧
⟹ invar-parents-shortest-paths (BFS-upd1 bfs-state)
⟨proof⟩

lemma *invar-parents-shortest-paths-holds-ret-1*[*invar-holds-intros*]:
⟦BFS-ret-1-conds bfs-state; invar-parents-shortest-paths bfs-state⟧ ⟹
invar-parents-shortest-paths (BFS-ret1 bfs-state)
⟨proof⟩

lemma *invar-parents-shortest-paths-holds*[*invar-holds-intros*]:
assumes BFS-dom bfs-state invar-well-formed bfs-state invar-subsets bfs-state
invar-current-no-out bfs-state invar-3-1 bfs-state
invar-3-2 bfs-state invar-3-3 bfs-state
invar-dist-bounded bfs-state invar-current-reachable bfs-state
invar-parents-shortest-paths bfs-state
shows invar-parents-shortest-paths (BFS bfs-state)
⟨proof⟩

lemma *BFS-ret-1*[*ret-holds-intros*]:
BFS-ret-1-conds (bfs-state) ⟹ BFS-ret-1-conds (BFS-ret1 bfs-state)
⟨proof⟩

lemma *ret1-holds*[*ret-holds-intros*]:
assumes BFS-dom bfs-state
shows BFS-ret-1-conds (BFS bfs-state)
⟨proof⟩

lemma *BFS-correct-1-ret-1*:
⟦invar-subsets bfs-state; invar-goes-through-current bfs-state; BFS-ret-1-conds bfs-state;
u ∈ t-set srcs; t ∉ t-set (visited bfs-state)⟧
⟹ ∄p. vwalk-bet (Graph.digraph-abs G) u p t
⟨proof⟩

lemma *BFS-correct-2-ret-1*:
⟦*invar-well-formed bfs-state*; *invar-subsets bfs-state*; *invar-dist bfs-state*; *BFS-ret-1-conds bfs-state*;
t ∈ *t-set* (*visited bfs-state*) − *t-set srcs*⟧
⟹ *distance-set* (*Graph.digraph-abs G*) (*t-set srcs*) *t* =
distance-set (*Graph.digraph-abs* (*parents bfs-state*)) (*t-set srcs*) *t*
⟨*proof*⟩

lemma *BFS-correct-3-ret-1*:
⟦*invar-parents-shortest-paths bfs-state*; *BFS-ret-1-conds bfs-state*;
u ∈ *t-set srcs*; *Vwalk.vwalk-bet* (*Graph.digraph-abs* (*parents bfs-state*)) *u p v*⟧
⟹ *length p* − *1* = *distance-set* (*Graph.digraph-abs G*) (*t-set srcs*) *v*
⟨*proof*⟩

7.9 Termination

named-theorems *termination-intros*

declare *termination-intros*[*intro*!]

lemma *in-prod-relI*[*intro*!,*termination-intros*]:
⟦*f1 a* = *f1 a*′; (*a*, *a*′) ∈ *f2* <∗*mlex*∗> *r*⟧ ⟹ (*a*,*a*′) ∈ (*f1* <∗*mlex*∗> *f2* <∗*mlex*∗> *r*)
⟨*proof*⟩

definition *less-rel* = {(*x*::*nat*, *y*::*nat*). *x* < *y*}

lemma *wf-less-rel*[*intro*!]: *wf less-rel*
⟨*proof*⟩

definition *state-measure-rel call-measure* = *inv-image less-rel call-measure*

lemma *call-1-measure-nonsym*[*simp*]:
(*call-1-measure-1 BFS-state*, *call-1-measure-1 BFS-state*) ∉ *less-rel*
⟨*proof*⟩

lemma *call-1-terminates*[*termination-intros*]:
assumes *BFS-call-1-conds BFS-state invar-well-formed BFS-state invar-subsets BFS-state*
invar-current-no-out BFS-state
shows (*BFS-upd1 BFS-state*, *BFS-state*) ∈
call-1-measure-1 <∗*mlex*∗> *call-1-measure-2* <∗*mlex*∗> *r*
⟨*proof*⟩
including *Graph.adjmap.automation* **and** *Graph.vset.set.automation*
⟨*proof*⟩

lemma *wf-term-rel*: *wf BFS-term-rel*
⟨*proof*⟩

lemma *in-BFS-term-rel*[*termination-intros*]:
⟦*BFS-call-1-conds BFS-state*; *invar-well-formed BFS-state*; *invar-subsets BFS-state*; *invar-current-no-out BFS-state*⟧ ⟹
(*BFS-upd1 BFS-state*, *BFS-state*) ∈ *BFS-term-rel*
⟨*proof*⟩

lemma *BFS-terminates*[*termination-intros*]:
assumes *invar-well-formed BFS-state invar-subsets BFS-state invar-current-no-out BFS-state*
shows *BFS-dom BFS-state*
⟨*proof*⟩

lemma *not-vwalk-bet-empty*[*simp*]: ¬ *Vwalk.vwalk-bet* (*Graph.digraph-abs empty*) *u p v*
⟨*proof*⟩

lemma *not-edge-in-empty*[*simp*]: (*u*,*v*) ∉ (*Graph.digraph-abs empty*)
⟨*proof*⟩

7.10 Final Correctness Theorems

lemma *initial-state-props*[*invar-holds-intros*, *termination-intros*, *simp*]:
invar-well-formed (*initial-state*) (**is** *?g1*)
invar-subsets (*initial-state*) (**is** *?g2*)
invar-current-no-out (*initial-state*) (**is** *?g3*)
BFS-dom initial-state (**is** *?g4*)
invar-dist initial-state (**is** *?g5*)
invar-3-1 initial-state
invar-3-2 initial-state
invar-3-3 initial-state
invar-dist-bounded initial-state
invar-current-reachable initial-state
invar-goes-through-current initial-state
invar-current-no-out initial-state
invar-parents-shortest-paths initial-state
⟨*proof*⟩

lemma *BFS-correct-1*:
⟦*u* ∈ *t-set srcs*; *t* ∉ *t-set* (*visited* (*BFS initial-state*))⟧
⟹ ∄*p*. *vwalk-bet* (*Graph.digraph-abs G*) *u p t*
⟨*proof*⟩

lemma *BFS-correct-2*:
⟦*t* ∈ *t-set* (*visited* (*BFS initial-state*)) − *t-set srcs*⟧
⟹ *distance-set* (*Graph.digraph-abs G*) (*t-set srcs*) *t* =
distance-set (*Graph.digraph-abs* (*parents* (*BFS initial-state*))) (*t-set srcs*) *t*
⟨*proof*⟩

lemma *BFS-correct-3*:

⟦$u \in$ *t-set srcs*; *Vwalk.vwalk-bet* (*Graph.digraph-abs* (*parents* (*BFS initial-state*))) *u p v*⟧
⟹ *length p − 1 = distance-set* (*Graph.digraph-abs G*) (*t-set srcs*) *v*
⟨*proof*⟩

end

context

end

locale *BFS*

end