

Elementary Graph Traversal Algorithms

Mohammad Abdulaziz
King's College London

February 6, 2026

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/HOL¹.

Contents

1	A Basic Representation of Diraphs	5
2	A Theory on Vertex Walks in a Digraph	8
3	Vwalks to paths (as opposed to arc walks (<i>awalk-to-apat</i> before))	24

¹<https://github.com/mabdula/Isabelle-Graph-Library>

4	Distances	32
4.1	Distance from a vertex	32
4.2	Shortest Paths	35
4.3	Distance from a set of vertices	40
5	A Digraph Representation for Efficient Executable Functions	48
5.1	Abstraction lemmas	48
5.2	Abstraction lemmas	52
6	Depth-Frist Search	53
6.1	The program state	53
6.2	Setup for automation	53
6.3	A <i>locale</i> for fixing data structures and their implemenations .	53
6.4	Defining the Algorithm	54
6.5	Setup for Reasoning About the Algorithm	54
6.6	Loop Invariants	58
6.7	Proofs that the Invariants Hold	63
6.8	Termination	67
6.9	Final Correctness Theorems	69
7	Breadth-First Search	70
7.1	The Program State	70
7.2	Setup for Automation	70
7.3	A <i>locale</i> for fixing data structures and their implemenations .	70
7.4	The Algorithm Definition	71
7.5	Setup for Reasoning About the Algorithm	71
7.6	The Loop Invariants	73
7.7	Termination Measures and Relation	74
7.8	Proofs that the Invariants Hold	79
7.9	Termination	98
7.10	Final Correctness Theorems	100

theory *More-Lists*

imports *Main*

begin

lemma *list-2-preds-aux*:

$\llbracket x' \in \text{set } xs; P1\ x'; \bigwedge xs1\ x\ xs2. \llbracket xs = xs1\ @\ [x]\ @\ xs2; P1\ x \rrbracket \implies$
 $\exists\ ys1\ y\ ys2. x\ \# \ xs2 = ys1\ @\ [y]\ @\ ys2 \wedge P2\ y \rrbracket \implies$
 $\exists\ xs1\ x\ xs2. xs = xs1\ @\ [x]\ @\ xs2 \wedge P2\ x \wedge (\forall y \in \text{set } xs2. \neg P1\ y)$

proof(*goal-cases*)

case *assms: 1*

define *property*

where *property* *xs* =

$(\forall xs2\ xs1\ x. (xs = xs1\ @\ [x]\ @\ xs2 \wedge P1\ x) \longrightarrow$

```

      (∃ ys1 y ys2. x#xs2 = ys1 @ [y] @ ys2 ∧ P2 y)
for xs

have propE: [[property xs;
  (∧ xs1 x xs2. [xs = xs1 @ [x] @ xs2; P1 x] ⇒
    ∃ ys1 y ys2. x#xs2 = ys1 @ [y] @ ys2 ∧ P2 y) ⇒ P]]
by(auto simp add: property-def)

have property-append: property (xs @ ys) ⇒ property ys for xs ys
by(auto simp: property-def)

have property xs
using assms(3)
by (force simp: property-def)

thus ?thesis
using assms(1,2)
proof(induction xs arbitrary: x')
  case Nil
  then show ?case
  by auto
next
  case (Cons a xs)
  hence property xs
  by(auto intro: property-append[where xs = [a]])

  show ?case
  proof(cases x' = a)
    case x-eq-a: True

    then obtain ys1 y ys2 where x'#xs = ys1 @ [y] @ ys2 P2 y
    using ⟨property (a # xs)⟩ ⟨P1 x'⟩
    apply(elim propE)
    by (auto 10 10)

  show ?thesis
  proof(cases ys1 = [])
    case ys1-empty: True
    hence xs = ys2
    using ⟨x'#xs = ys1 @ [y] @ ys2⟩
    by auto
    then show ?thesis
    proof(cases ∃ x∈set ys2. P1 x)
      case x-in-ys2: True
      then obtain x where x ∈ set ys2 P1 x
      by auto

```

```

hence  $\exists xs1\ x\ xs2. xs = xs1 @ [x] @ xs2 \wedge P2\ x \wedge (\forall y \in set\ xs2. \neg P1\ y)$ 
  using  $\langle property\ xs \rangle \langle xs = ys2 \rangle \langle P2\ y \rangle$ 
  apply(intro Cons(1))
  by auto
then obtain  $xs1\ x\ xs2$  where  $(a \# xs) = (a \# xs1) @ [x] @ xs2 \wedge P2\ x$ 
 $\wedge (\forall y \in set\ xs2. \neg P1\ y)$ 
  by auto
  then show ?thesis
  by metis
next
case x-notin-ys2: False
hence  $a \# xs = a \# ys2 \wedge P2\ y \wedge (\forall y \in set\ ys2. \neg P1\ y)$ 
  using  $\langle xs = ys2 \rangle \langle P2\ y \rangle$ 
  by auto
then show ?thesis
  using  $\langle x' \# xs = ys1 @ [y] @ ys2 \rangle$  x-eq-a
  by blast
qed
next
case ys2-nempty: False
then obtain  $ys1'$  where  $xs = ys1' @ [y] @ ys2$ 
  using  $\langle x' \# xs = ys1 @ [y] @ ys2 \rangle$ 
  by (auto simp: neq-Nil-conv)
show ?thesis
proof(cases  $\exists x \in set\ ys2. P1\ x$ )
  case x-in-ys2: True
then obtain  $x$  where  $x \in set\ ys2\ P1\ x$ 
  by auto
hence  $\exists xs1\ x\ xs2. xs = xs1 @ [x] @ xs2 \wedge P2\ x \wedge (\forall y \in set\ xs2. \neg P1\ y)$ 
  using  $\langle property\ xs \rangle \langle xs = ys1' @ [y] @ ys2 \rangle \langle P2\ y \rangle$ 
  apply(intro Cons(1))
  by auto
then obtain  $xs1\ x\ xs2$  where  $(a \# xs) = (a \# xs1) @ [x] @ xs2 \wedge P2\ x$ 
 $\wedge (\forall y \in set\ xs2. \neg P1\ y)$ 
  by auto
  then show ?thesis
  by metis
next
case x-notin-ys2: False
hence  $a \# xs = (a \# ys1') @ [y] @ ys2 \wedge P2\ y \wedge (\forall y \in set\ ys2. \neg P1\ y)$ 
  using  $\langle xs = ys1' @ [y] @ ys2 \rangle \langle P2\ y \rangle$ 
  by auto
then show ?thesis
  by metis
qed
qed
next
case x-neq-a: False
hence  $x' \in set\ xs$ 

```

```

    using Cons
    by auto
  then obtain  $xs1\ x\ xs2$  where  $xs = xs1 @ [x] @ xs2$   $P2\ x$  ( $\forall y \in set\ xs2. \neg P1$ 
 $y$ )
    using Cons  $\langle property\ xs \rangle$ 
    by blast
  hence  $a \# xs = (a \# xs1) @ [x] @ xs2 \wedge P2\ x \wedge (\forall y \in set\ xs2. \neg P1\ y)$ 
    by auto
  thus ?thesis
    by metis
qed
qed
qed

```

lemma *list-2-preds*:

```

 $\llbracket x \in set\ xs; P1\ x; \bigwedge xs1\ x\ xs2. \llbracket xs = xs1 @ [x] @ xs2; P1\ x \rrbracket \implies \exists ys1\ y\ ys2.$ 
 $x \# xs2 = ys1 @ [y] @ ys2 \wedge P2\ y \rrbracket \implies$ 
 $\exists xs1\ x\ xs2. xs = xs1 @ [x] @ xs2 \wedge P2\ x \wedge (\forall y \in set\ xs2. \neg P1\ y \wedge \neg P2\ y)$ 
proof(goal-cases)
  case assms: 1
  hence  $\exists xs1\ x\ xs2. xs = xs1 @ [x] @ xs2 \wedge P2\ x \wedge (\forall y \in set\ xs2. \neg P1\ y)$ 
    by (fastforce intro!: list-2-preds-aux)
  then obtain  $xs1\ x\ xs2$  where  $xs = xs1 @ [x] @ xs2$   $P2\ x$  ( $\forall y \in set\ xs2. \neg P1\ y$ )
    by auto
  hence  $\exists ys1\ y\ ys2. x \# xs2 = ys1 @ [y] @ ys2 \wedge P2\ y \wedge (\forall z \in set\ ys2. \neg P2\ z)$ 
    by (auto intro!: split-list-last-prop)
  then obtain  $ys1\ y\ ys2$  where  $x \# xs2 = ys1 @ [y] @ ys2$   $P2\ y$  ( $\forall z \in set\ ys2. \neg$ 
 $P2\ z$ )
    by auto
  hence  $(\forall z \in set\ ys2. \neg P1\ z \wedge \neg P2\ z)$ 
    using  $\langle (\forall y \in set\ xs2. \neg P1\ y) \rangle \langle P2\ x \rangle$ 
    by (metis Un-iff insert-iff list.simps(15) set-append)
  moreover have  $xs = (xs1 @ ys1) @ [y] @ ys2$ 
    using  $\langle xs = xs1 @ [x] @ xs2 \rangle \langle x \# xs2 = ys1 @ [y] @ ys2 \rangle$ 
    by auto
  ultimately show ?case
    using  $\langle P2\ y \rangle$ 
    by fastforce
qed

```

lemma *list-inter-mem-iff*: $set\ xs \cap s \neq \{\}$ $\longleftrightarrow (\exists xs1\ x\ xs2. xs = xs1 @ [x] @ xs2$

$\wedge x \in s)$

by (metis *append.left-neutral append-Cons disjoint-iff in-set-conv-decomp*)

end

theory *Pair-Graph*

imports

Main

Graph-Theory.Rtrancl-On

begin

1 A Basic Representation of Diraphs

type-synonym $'v$ *dgraph* = ($'v \times 'v$) *set*

definition $dVs::('v \times 'v)$ *set* \Rightarrow $'v$ *set* **where**
 dVs $G = \bigcup \{\{v1, v2\} \mid v1 \ v2. (v1, v2) \in G\}$

lemma *induct-pcpl*:

$\llbracket P \ []; \bigwedge x. P \ [x]; \bigwedge x \ y \ zs. P \ zs \Longrightarrow P \ (x \ \# \ y \ \# \ zs) \rrbracket \Longrightarrow P \ xs$
by *induction-schema (pat-completeness, lexicographic-order)*

lemma *induct-pcpl-2*:

$\llbracket P \ []; \bigwedge x. P \ [x]; \bigwedge x \ y. P \ [x, y]; \bigwedge x \ y \ z. P \ [x, y, z]; \bigwedge w \ x \ y \ z \ zs. P \ zs \Longrightarrow P \ (w \ \# \ x \ \# \ y \ \# \ z \ \# \ zs) \rrbracket \Longrightarrow P \ xs$
by *induction-schema (pat-completeness, lexicographic-order)*

lemma *dVs-empty[simp]*: $dVs \ \{\} = \{\}$

by (*simp add: dVs-def*)

lemma *dVs-empty-iff[simp]*: $dVs \ E = \{\} \longleftrightarrow E = \{\}$

unfolding *dVs-def* **by** *auto*

lemma *dVsI[intro]*:

assumes $(a, v) \in dG$ **shows** $a \in dVs \ dG$ **and** $v \in dVs \ dG$
using *assms* **unfolding** *dVs-def* **by** *auto*

lemma *dVsI'*:

assumes $e \in dG$ **shows** $fst \ e \in dVs \ dG$ **and** $snd \ e \in dVs \ dG$
using *assms*
by (*auto intro: dVsI[of fst e snd e]*)

lemma *dVs-union-distr[simp]*: $dVs \ (G \cup H) = dVs \ G \cup dVs \ H$

unfolding *dVs-def* **by** *blast*

lemma *dVs-union-big-distr*: $dVs \ (\bigcup G) = \bigcup (dVs \ ` G)$

apply (*induction G rule: infinite-finite-induct*)

apply *simp-all*

by (*auto simp add: dVs-def*)

lemma *dVs-eq*: $dVs \ E = fst \ ` E \cup snd \ ` E$

by (*induction E rule: infinite-finite-induct*)

(*auto simp: dVs-def intro!: image-eqI, blast+*)

lemma *finite-vertices-iff*: $finite \ (dVs \ E) \longleftrightarrow finite \ E$ (**is** $?L \longleftrightarrow ?R$)

proof

show $?R \Longrightarrow ?L$

by (*induction E rule: finite-induct*)

(*auto simp: dVs-eq*)

show $?L \Longrightarrow ?R$

proof (*rule ccontr*)
show $?L \implies \neg ?R \implies \text{False}$
unfolding *dVs-eq*
using *finite-subset subset-fst-snd* **by** *fastforce*
qed
qed

abbreviation *reachable1* :: $('v \times 'v) \text{ set} \Rightarrow 'v \Rightarrow 'v \Rightarrow \text{bool}$ ($- \rightarrow^+_1 - [100,100]$ 40) **where**
reachable1 $E u v \equiv (u,v) \in E^+$

definition *reachable* :: $('v \times 'v) \text{ set} \Rightarrow 'v \Rightarrow 'v \Rightarrow \text{bool}$ ($- \rightarrow^*_1 - [100,100]$ 40) **where**
reachable $E u v = ((u,v) \in \text{rtrancl-on } (dVs E) E)$

lemma *reachableE[elim?]*:
assumes $(u,v) \in E$
obtains e **where** $e \in E \ e = (u, v)$
using *assms* **by** *auto*

lemma *reachable-refl[intro!, Pure.intro!, simp]*: $v \in dVs E \implies v \rightarrow^*_E v$
unfolding *reachable-def* **by** *auto*

lemma *reachable-trans[trans,intro]*:
assumes $u \rightarrow^*_E v \ v \rightarrow^*_E w$ **shows** $u \rightarrow^*_E w$
using *assms* **unfolding** *reachable-def* **by** (*rule rtrancl-on-trans*)

lemma *reachable-edge[dest,intro]*: $(u,v) \in E \implies u \rightarrow^*_E v$
unfolding *reachable-def*
by (*auto intro!: rtrancl-consistent-rtrancl-on*)

lemma *reachable-induct[consumes 1, case-names base step]*:
assumes *major*: $u \rightarrow^*_E v$
and cases: $\llbracket u \in dVs E \rrbracket \implies P u$
 $\bigwedge x y. \llbracket u \rightarrow^*_E x; (x,y) \in E; P x \rrbracket \implies P y$
shows $P v$
using *assms* **unfolding** *reachable-def* **by** (*rule rtrancl-on-induct*)

lemma *converse-reachable-induct[consumes 1, case-names base step, induct pred: reachable]*:
assumes *major*: $u \rightarrow^*_E v$
and cases: $v \in dVs E \implies P v$
 $\bigwedge x y. \llbracket (x,y) \in E; y \rightarrow^*_E v; P y \rrbracket \implies P x$
shows $P u$
using *assms* **unfolding** *reachable-def* **by** (*rule converse-rtrancl-on-induct*)

lemma *reachable-in-dVs*:
assumes $u \rightarrow^*_E v$
shows $u \in dVs E \ v \in dVs E$

using *assms* by (*induct*) (*simp-all add: dVsI*)

lemma *reachable1-in-dVs*:

assumes $u \rightarrow^+ E v$

shows $u \in dVs E v \in dVs E$

using *assms* by (*induct*) (*simp-all add: dVsI*)

lemma *reachable1-reachable[intro]*:

$v \rightarrow^+ E w \Longrightarrow v \rightarrow^* E w$

unfolding *reachable-def*

by (*auto intro: rtrancl-consistent-rtrancl-on simp: dVsI reachable1-in-dVs*)

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 $v \rightarrow^+ E w$

using *assms*

unfolding *reachable-def*

by (*auto dest: rtrancl-on-rtranclI rtranclD*)

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

lemma *arc-implies-dominates*: $e \in E \Longrightarrow (\text{fst } e, \text{snd } e) \in E$ **by** *auto*

definition *neighbourhood*:: $('v \times 'v) \text{ set} \Rightarrow 'v \Rightarrow 'v \text{ set}$ **where**

neighbourhood $G u = \{v. (u,v) \in G\}$

lemma

neighbourhoodI[intro]: $v \in (\text{neighbourhood } G u) \Longrightarrow (u,v) \in G$ **and**

neighbourhoodD[dest]: $(u,v) \in G \Longrightarrow v \in (\text{neighbourhood } G u)$

by (*auto simp: neighbourhood-def*)

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

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

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

by (*auto simp: dVs-def*)

lemma *dVs-insert[elim]*:

$v \in dVs (\text{insert } (x,y) G) \Longrightarrow \llbracket v = x \Longrightarrow P; v = y \Longrightarrow P; v \in dVs G \Longrightarrow P \rrbracket \Longrightarrow P$

by (*auto simp: dVs-def*)

lemma *in-neighbourhood-dVs[simp, intro]*:

$v \in \text{neighbourhood } G u \Longrightarrow v \in dVs G$

by *auto*

lemma *subset-neighbourhood-dVs*[*simp, intro*]:
neighbourhood G $u \subseteq dVs$ G
by *auto*

lemma *in-dVsE*: $v \in dVs$ $G \implies [\![\!(\bigwedge u. (u, v) \in G \implies P); (\bigwedge u. (v, u) \in G \implies P)\!] \implies P$
 $v \notin dVs$ $G \implies ([\!(\bigwedge u. (u, v) \notin G); (\bigwedge u. (v, u) \notin G)\!] \implies P) \implies P$
by (*auto simp: dVs-def*)

lemma *neighbourhood-union*[*simp*]: *neighbourhood* $(G \cup G')$ $u =$ *neighbourhood* G $u \cup$ *neighbourhood* G' u
by (*auto simp: neighbourhood-def*)

lemma *vs-are-gen*: dVs (*set E-impl*) = *set* (*map prod.fst E-impl*) \cup *set* (*map prod.snd E-impl*)
by(*induction E-impl*) *auto*

lemma *dVs-swap*: dVs (*prod.swap ' E*) = dVs E
by(*auto simp add: dVs-def*)

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: $v \in dVs$ $G \implies$ *vwalk* [v] |
vwalk2: $(u, v) \in G \implies$ *vwalk* ($v \# vs$) \implies *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 $p \wedge p \neq [] \wedge hd$ $p = u \wedge last$ $p = v)$

lemma *vwalk-then-edge*: *vwalk-bet* dG u p $v \implies u \neq v \implies (\exists v''. (u, v'') \in dG)$
by(*cases p; auto split: if-splits simp: neq-Nil-conv vwalk-bet-def*)

lemma *vwalk-then-in-dVs*: *vwalk* dG $p \implies v \in set$ $p \implies v \in dVs$ dG
by(*induction rule: vwalk.induct*) (*auto simp: dVs-def*)

lemma *vwalk-cons*: $vwalk\ dG\ (v1\ \#\ v2\ \#\ p) \implies (v1, v2) \in dG$
by *simp*

lemma *hd-of-vwalk-bet*:
 $vwalk\ bet\ dG\ v\ p\ v' \implies \exists p'.\ p = v\ \#\ p'$
by(*auto simp: neq-Nil-conv vwalk-bet-def*)

lemma *hd-of-vwalk-bet'*:
 $vwalk\ bet\ dG\ v\ p\ v' \implies v = hd\ p$
by(*auto simp: neq-Nil-conv vwalk-bet-def*)

— TODO: intro

lemma *hd-of-vwalk-bet''*: $vwalk\ bet\ dG\ u\ p\ v \implies u \in set\ p$
using *hd-of-vwalk-bet*
by *force*

lemma *last-of-vwalk-bet*:
 $vwalk\ bet\ dG\ v\ p\ v' \implies v' = last\ p$
by(*auto simp: neq-Nil-conv vwalk-bet-def*)

lemma *last-of-vwalk-bet'*:
 $vwalk\ bet\ dG\ v\ p\ v' \implies \exists p'.\ p = p' \ @\ [v']$
by(*auto simp: vwalk-bet-def dest: append-butlast-last-id[symmetric]*)

lemma *append-vwalk-pref*: $vwalk\ dG\ (p1 \ @\ p2) \implies vwalk\ dG\ p1$
by (*induction p1*) (*fastforce intro: vwalk.intros elim: vwalk.cases simp: dVsI*)+

lemma *append-vwalk-suff*: $vwalk\ dG\ (p1 \ @\ p2) \implies vwalk\ dG\ p2$
by (*induction p1*) (*fastforce intro: vwalk.intros elim: vwalk.cases simp:*)+

lemma *append-vwalk*: $vwalk\ dG\ p1 \implies vwalk\ dG\ p2 \implies (p1 \neq [] \implies p2 \neq [] \implies (last\ p1, hd\ p2) \in dG) \implies vwalk\ dG\ (p1 \ @\ p2)$
by (*induction p1*) (*fastforce intro: vwalk.intros elim: vwalk.cases simp: dVsI*)+

— TODO: dest

lemma *split-vwalk*:
 $vwalk\ bet\ dG\ v1\ (p1 \ @\ v2 \ \#\ p2)\ v3 \implies vwalk\ bet\ dG\ v2\ (v2 \ \#\ p2)\ v3$
unfolding *vwalk-bet-def*
proof (*induction p1*)
case (*Cons v p1*)
then show *?case*
by (*auto intro: append-vwalk-suff[where ?p1.0 = v1 \ \#\ p1] simp add: vwalk-then-in-dVs vwalk-bet-def*)
qed *auto*

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

by(*auto simp: neq-Nil-conv vwalk-bet-def*)

lemma *vwalk-bet-cons-2*:
 $vwalk\ bet\ dG\ v\ p\ v' \implies p \neq [] \implies vwalk\ bet\ dG\ v\ (v \# tl\ p)\ v'$
by(*auto simp: neq-Nil-conv vwalk-bet-def*)

lemma *vwalk-bet-snoc*:
 $vwalk\ bet\ dG\ v'\ (p @ [v])\ v'' \implies v = v''$
by(*auto simp: neq-Nil-conv vwalk-bet-def*)

lemma *vwalk-bet-vwalk*:
 $p \neq [] \implies vwalk\ bet\ dG\ (hd\ p)\ p\ (last\ p) \longleftrightarrow vwalk\ dG\ p$
by(*auto simp: neq-Nil-conv vwalk-bet-def*)

lemma *vwalk-snoc-edge'*: $vwalk\ dG\ (p @ [v]) \implies (v, v') \in dG \implies vwalk\ dG\ ((p @ [v]) @ [v'])$
by (*rule append-vwalk*) (*auto simp add: dVs-def*)

lemma *vwalk-snoc-edge*: $vwalk\ dG\ (p @ [v]) \implies (v, v') \in dG \implies vwalk\ dG\ (p @ [v, v'])$
using *vwalk-snoc-edge'*
by *simp*

lemma *vwalk-snoc-edge-2*: $vwalk\ dG\ (p @ [v, v']) \implies (v, v') \in dG$
using *append-vwalk-suff*[**where** $?p2.0 = [v, v']$]
by *auto*

lemma *vwalk-append-edge*: $vwalk\ dG\ (p1 @ p2) \implies p1 \neq [] \implies p2 \neq [] \implies (last\ p1, hd\ p2) \in dG$
by (*induction p1*) (*auto simp: vwalk-then-in-dVs neq-Nil-conv*)

lemma *vwalk-transitive-rel*:
assumes $(\bigwedge x\ y\ z. R\ x\ y \implies R\ y\ z \implies R\ x\ z) (\bigwedge v1\ v2. (v1, v2) \in dG \implies R\ v1\ v2)$
shows $vwalk\ dG\ (v \# vs) \implies v' \in set\ vs \implies R\ v\ v'$
proof(*induction vs arbitrary: v v'*)
case (*Cons v'' vs*)
hence $R\ v\ v''$
using *assms(2)*
by *simp*
thus *?case*
proof(*cases v' = v''*)
case *False*
thus *?thesis*
apply(*intro assms(1)[OF $R\ v\ v''$] Cons*)
using *Cons.prem*
by *auto*
qed *simp*
qed *auto*

```

lemma vwalk-transitive-rel':
  assumes ( $\bigwedge x y z. R x y \implies R y z \implies R x z$ ) ( $\bigwedge v1 v2. (v1, v2) \in dG \implies R v1 v2$ )
  shows  $vwalk\ dG\ (vs\ @\ [v]) \implies v' \in set\ vs \implies R\ v'\ v$ 
proof(induction vs arbitrary: v v' rule: rev-induct)
  case (snoc v'' vs)
  hence  $R\ v''\ v$ 
    by(auto intro: assms vwalk-snoc-edge-2)
  thus ?case
  proof(cases v' = v'')
    case True
    then show ?thesis
      by (simp add: <R v'' v>)
    next
    case False
    thus ?thesis
      apply(intro assms(1)[OF - <R v'' v>] snoc append-vwalk-pref[where ?p1.0 = vs @ [v'']])
      using snoc(2,3)
      by auto
  qed
qed auto

```

```

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:  $vwalk\ E\ p \implies b \in set\ (edges-of-vwalk\ p) \implies b \in E$ 
by (induction p rule: edges-of-vwalk.induct, auto)

```

```

lemma edges-of-vwalk-index:
   $Suc\ i < length\ p \implies edges-of-vwalk\ p\ !\ i = (p\ !\ i, p\ !\ Suc\ i)$ 
proof (induction i arbitrary: p)
  case (Suc i)
  then obtain  $u\ v\ ps$  where  $p = u\#\ v\#\ ps$  by (auto dest!: Suc-leI simp: Suc-le-length-iff)
  hence  $edges-of-vwalk\ (v\#\ ps)\ !\ i = ((v\#\ ps)\ !\ i, (v\#\ ps)\ !\ Suc\ i)$  using Suc.IH
  Suc.prems by simp
  then show ?case using  $\langle p = u\#\ v\#\ ps \rangle$  by simp
qed (auto dest!: Suc-leI simp: Suc-le-length-iff)

```

```

lemma edges-of-vwalk-length:  $length\ (edges-of-vwalk\ p) = length\ p - 1$ 
by (induction p rule: edges-of-vwalk.induct, auto)

```

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

```

lemma edges-of-vwalk-for-inner:

```

assumes $p ! i = v \text{ Suc } i < \text{length } p$
obtains w **where** $\text{edges-of-vwalk } p ! i = (v, w)$
by (*simp add: assms edges-of-vwalk-index*)

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

lemma *edges-of-vwalk-for-inner!*:
assumes $p ! (\text{Suc } i) = v \text{ Suc } i < \text{length } p$
obtains u **where** $(u, v) = \text{edges-of-vwalk } p ! i$
using *assms* **by** (*simp add: edges-of-vwalk-index*)

lemma *hd-edges-neq-last*:
 $\llbracket (\text{last } p, \text{hd } p) \notin \text{set } (\text{edges-of-vwalk } p); \text{hd } p \neq \text{last } p; p \neq [] \rrbracket \implies$
 $\text{hd } (\text{edges-of-vwalk } (\text{last } p \# p)) \neq \text{last } (\text{edges-of-vwalk } (\text{last } p \# p))$
by (*induction p*) (*auto elim: edges-of-vwalk.elims*)

lemma *v-in-edge-in-vwalk*:
assumes $(u, v) \in \text{set } (\text{edges-of-vwalk } p)$
shows $u \in \text{set } p \ v \in \text{set } p$
using *assms*
by (*induction p rule: edges-of-vwalk.induct*) *auto*

lemma *distinct-edges-of-vwalk*:
 $\text{distinct } p \implies \text{distinct } (\text{edges-of-vwalk } p)$
by (*induction p rule: edges-of-vwalk.induct*) (*auto dest: v-in-edge-in-vwalk*)

lemma *distinct-edges-of-vwalk-cons*:
 $\text{distinct } (\text{edges-of-vwalk } (v \# p)) \implies \text{distinct } (\text{edges-of-vwalk } p)$
by (*cases p; simp*)

lemma *tl-vwalk-is-vwalk*: $\text{vwalk } E \ p \implies \text{vwalk } E \ (\text{tl } p)$
by (*induction p rule: vwalk.induct; simp*)

lemma *vwalk-concat*:
assumes $\text{vwalk } E \ p \ \text{vwalk } E \ q \ q \neq [] \ p \neq [] \implies \text{last } p = \text{hd } q$
shows $\text{vwalk } E \ (p @ \text{tl } q)$
using *assms*
by (*induction p*) (*simp-all add: tl-vwalk-is-vwalk*)

lemma *edges-of-vwalk-append-2*:
 $p' \neq [] \implies \text{edges-of-vwalk } (p @ p') = \text{edges-of-vwalk } (p @ [\text{hd } p']) @ \text{edges-of-vwalk } p'$
by (*induction p rule: induct-list012*) (*auto intro: list.exhaust[of p']*)

lemma *edges-of-vwalk-append*: $\exists ep. \text{edges-of-vwalk } (p @ p') = ep @ \text{edges-of-vwalk } p'$
by (*cases p' = []*) (*auto dest: edges-of-vwalk-append-2*)

lemma *append-butlast-last-cancel*: $p \neq [] \implies \text{butlast } p @ \text{last } p \# p' = p @ p'$

by *simp*

lemma *edges-of-vwalk-append-3*:
 assumes $p \neq []$
 shows $\text{edges-of-vwalk } (p @ p') = \text{edges-of-vwalk } p @ \text{edges-of-vwalk } (\text{last } p \# p')$
 using *assms*
 by (*auto simp flip: append-butlast-last-cancel simp: edges-of-vwalk-append-2*)

lemma *vwalk-vertex-has-edge*:
 assumes $\text{length } p \geq 2 \ v \in \text{set } p$
 obtains $e \ u$ where $e \in \text{set } (\text{edges-of-vwalk } p) \ e = (u, v) \vee e = (v, u)$
proof –
 obtain i where *idef*: $p ! i = v \ i < \text{length } p$
 using *assms*(2) by (*auto simp: in-set-conv-nth*)
 have *eodplength'*: $\text{Suc } (\text{length } (\text{edges-of-vwalk } p)) = \text{length } p$
 using *assms*(1) by (*auto simp: edges-of-vwalk-length*)
 hence *eodplength*: $\text{length } (\text{edges-of-vwalk } p) \geq 1$ using *assms*(1) by *simp*

from *idef* consider $(\text{nil}) \ i = 0 \mid (\text{gt}) \ i > 0 \ \text{Suc } i < \text{length } p \mid (\text{last}) \ \text{Suc } i = \text{length } p$
 by *linarith*
 thus ?*thesis*
proof *cases*
 case *nil*
 hence $\text{edges-of-vwalk } p ! 0 = (p ! 0, p ! 1)$ using *edges-of-vwalk-index* *assms*(1)
 by *force*
 then show ?*thesis* using *that nil idef eodplength*
 by (*force simp: in-set-conv-nth*)
 next
 case *gt*
 then obtain w where $w: (v, w) = \text{edges-of-vwalk } p ! i$
 by (*auto elim: edges-of-vwalk-for-inner[OF idef(1)]*)
 have $i < \text{length } (\text{edges-of-vwalk } p)$
 using *eodplength' gt*(2) by *auto*
 then show ?*thesis* using *that w[symmetric] nth-mem* by *blast*
 next
 case *last*
 then obtain w where $w: (w, v) = \text{edges-of-vwalk } p ! (i - 1)$
 using *edges-of-vwalk-for-inner'[of p i - 1] eodplength' eodplength*
 by (*auto simp: idef*)
 have $i - 1 < \text{length } (\text{edges-of-vwalk } p)$
 using *eodplength eodplength' last* by *linarith*
 then show ?*thesis* using *that w[symmetric] nth-mem* by *blast*
 qed
 qed

lemma *v-in-edge-in-vwalk-inj*:
 assumes $e \in \text{set } (\text{edges-of-vwalk } p)$
 obtains $u \ v$ where $e = (u, v)$

by *fastforce*

lemma *v-in-edge-in-vwalk-gen*:

assumes $e \in \text{set } (\text{edges-of-vwalk } p)$ $e = (u, v)$

shows $u \in \text{set } p$ $v \in \text{set } p$

using *assms v-in-edge-in-vwalk* **by** *simp-all*

lemma *edges-of-vwalk-dVs*: $dVs (\text{set } (\text{edges-of-vwalk } p)) \subseteq \text{set } p$

by (*auto intro: v-in-edge-in-vwalk simp: dVs-def*)

lemma *last-v-snd-last-e*:

assumes $\text{length } p \geq 2$

shows $\text{last } p = \text{snd } (\text{last } (\text{edges-of-vwalk } p))$ — is this the best formulation for this?

using *assms*

by (*induction p rule: induct-list012*)

(*auto split: if-splits elim: edges-of-vwalk.elims simp: Suc-leI*)

lemma *hd-v-fst-hd-e*:

assumes $\text{length } p \geq 2$

shows $\text{hd } p = \text{fst } (\text{hd } (\text{edges-of-vwalk } p))$

using *assms*

by (*auto dest: Suc-leI simp: Suc-le-length-iff numeral-2-eq-2*)

lemma *last-in-edge*:

$p \neq [] \implies \exists u. (u, \text{last } p) \in \text{set } (\text{edges-of-vwalk } (v \# p)) \wedge u \in \text{set } (v \# p)$

by (*induction p arbitrary: v*) *auto*

lemma *edges-of-vwalk-append-subset*:

shows $\text{set } (\text{edges-of-vwalk } p') \subseteq \text{set } (\text{edges-of-vwalk } (p @ p'))$

by (*fastforce intro: exE[OF edges-of-vwalk-append, of p p']*)

lemma *nonempty-vwalk-vwalk-bet[intro?]*:

assumes $\text{vwalk } E p p \neq []$ $\text{hd } p = u$ $\text{last } p = v$

shows $\text{vwalk-bet } E u p v$

using *assms*

unfolding *vwalk-bet-def*

by *blast*

lemma *vwalk-bet-nonempty*:

assumes $\text{vwalk-bet } E u p v$

shows [*simp*]: $p \neq []$

using *assms*

unfolding *vwalk-bet-def* **by** *blast*

lemma *vwalk-bet-nonempty-vwalk[elim]*:

assumes $\text{vwalk-bet } E u p v$

shows $\text{vwalk } E p p \neq []$ $\text{hd } p = u$ $\text{last } p = v$

using *assms*

unfolding *vwalk-bet-def* **by** *blast+*

lemma *vwalk-bet-reflexive*[*intro*]:
assumes $w \in dVs\ E$
shows *vwalk-bet* $E\ w\ [w]\ w$
using *assms*
unfolding *vwalk-bet-def* **by** *simp*

lemma *singleton-hd-last*: $q \neq [] \implies tl\ q = [] \implies hd\ q = last\ q$
by (*cases q*) *simp-all*

lemma *singleton-hd-last'*: $q \neq [] \implies butlast\ q = [] \implies hd\ q = last\ q$
by (*cases q*) *auto*

lemma *vwalk-bet-transitive*:
assumes *vwalk-bet* $E\ u\ p\ v\ vwalk-bet\ E\ v\ q\ w$
shows *vwalk-bet* $E\ u\ (p\ @\ tl\ q)\ w$
using *assms*
unfolding *vwalk-bet-def*
by (*auto intro: vwalk-concat simp: last-append singleton-hd-last last-tl*)

lemma *vwalk-bet-in-dVs*:
assumes *vwalk-bet* $E\ a\ p\ b$
shows $set\ p \subseteq dVs\ E$
using *assms vwalk-then-in-dVs*
unfolding *vwalk-bet-def* **by** *fast*

lemma *vwalk-bet-endpoints*:
assumes *vwalk-bet* $E\ u\ p\ v$
shows [*simp*]: $u \in dVs\ E$
and [*simp*]: $v \in dVs\ E$
using *assms vwalk-then-in-dVs*
unfolding *vwalk-bet-def* **by** *fastforce+*

lemma *vwalk-bet-pref*:
assumes *vwalk-bet* $E\ u\ (pr\ @\ [x]\ @\ su)\ v$
shows *vwalk-bet* $E\ u\ (pr\ @\ [x])\ x$
using *assms*
unfolding *vwalk-bet-def*
by (*auto simp: append-vwalk-pref*) (*simp add: hd-append*)

lemma *vwalk-bet-suff*:
assumes *vwalk-bet* $E\ u\ (pr\ @\ [x]\ @\ su)\ v$
shows *vwalk-bet* $E\ x\ (x\ \#\ su)\ v$
using *append-vwalk-suff assms*
unfolding *vwalk-bet-def* **by** *auto*

lemma *edges-are-vwalk-bet*:
assumes $(v, w) \in E$

shows $vwalk\text{-bet } E v [v, w] w$
unfolding $vwalk\text{-bet-def}$
using $assms$
by ($simp$ $add: dVsI$)

lemma $induct\text{-vwalk-bet}[case\text{-names } path1\ path2, consumes\ 1, induct\ set: vwalk\text{-bet}]$:

assumes $vwalk\text{-bet } E a p b$
assumes $Path1: \bigwedge v. v \in dVs\ E \implies P\ E [v] v v$
assumes $Path2: \bigwedge v\ v'\ vs\ b. (v, v') \in E \implies vwalk\text{-bet } E\ v'\ (v' \# vs)\ b \implies P\ E$
 $(v' \# vs)\ v'\ b \implies P\ E (v \# v' \# vs)\ v\ b$
shows $P\ E p a b$

proof –

have $vwalk\ E\ p\ p \neq []\ hd\ p = a\ last\ p = b$ **using** $assms(1)$ **by** $auto$
thus $?thesis$

proof ($induction\ p\ arbitrary: a\ b$)

case $vwalk0$

then show $?case$ **by** $simp$

next

case $vwalk1$

then show $?case$ **using** $Path1$ **by** $fastforce$

next

case ($vwalk2\ v\ v'\ vs\ a\ b$)

then have $vwalk\text{-bet } E\ v'\ (v' \# vs)\ b$

by ($simp$ $add: vwalk2.hyps(1)\ vwalk\text{-bet-def}$)

then show $?case$ **using** $vwalk2\ Path2$

by $auto$

qed

qed

lemma $vwalk\text{-append}$:

assumes $vwalk\ E\ xs\ vwalk\ E\ ys\ last\ xs = hd\ ys$

shows $vwalk\ E\ (xs\ @\ tl\ ys)$

using $assms\ vwalk\text{-concat}$ **by** $fastforce$

lemma $vwalk\text{-append2}$:

assumes $vwalk\ E\ (xs\ @\ [x])\ vwalk\ E\ (x\ \# ys)$

shows $vwalk\ E\ (xs\ @\ x\ \# ys)$

using $assms$ **by** ($auto\ dest: vwalk\text{-append}$)

lemma $vwalk\text{-appendD-last}$:

$vwalk\ E\ (xs\ @\ [x, y]) \implies vwalk\ E\ (xs\ @\ [x])$

by ($simp$ $add: append\text{-vwalk-pref}$)

lemma $vwalk\text{-ConsD}$:

$vwalk\ E\ (x\ \# xs) \implies vwalk\ E\ xs$

by ($auto\ dest: tl\text{-vwalk-is-vwalk}$)

lemmas $vwalkD = vwalk\text{-ConsD}\ append\text{-vwalk-pref}\ append\text{-vwalk-suff}$

lemma *vwalk-alt-induct*[*consumes 1, case-names Single Snoc*]:

assumes
 $vwalk\ E\ p\ P\ []\ (\bigwedge x. P\ [x])$
 $\bigwedge y\ x\ xs. (y, x) \in E \implies vwalk\ E\ (xs\ @\ [y]) \implies P\ (xs\ @\ [y]) \implies P\ (xs\ @\ [y,$
 $x])$
shows $P\ p$
using *assms(1)*
proof (*induction rule: rev-induct*)
case *Nil*
then show *?case* **by** (*simp add: assms*)
next
case (*snoc x xs*)
then show *?case*
by (*cases xs rule: rev-cases*) (*auto intro!: assms simp add: vwalk-snoc-edge-2*
append-vwalk-pref)
qed

lemma *vwalk-append-single*:

assumes $vwalk\ E\ p\ (last\ p, x) \in E$
shows $vwalk\ E\ (p\ @\ [x])$
using *assms*
by (*auto intro!: append-vwalk dest: dVsI*)

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 –
from *vwalk-decomp*[*of E x # xs y # ys @ [x] assms*] **have**
 $vwalk\ E\ (x\ \# \ xs)\ vwalk\ E\ (y\ \# \ ys\ @\ [x])\ (last\ (x\ \# \ xs), y) \in E$
by *auto*
then have $vwalk\ E\ (x\ \# \ xs\ @\ [y])$
by (*auto dest: vwalk-append-single*)
from *vwalk-append*[*OF <vwalk E (y # ys @ [x])> this*] **show** *?thesis* **by** *auto*
qed

lemma *vwalk-bet-nonempty'*[*simp*]: $\neg\ vwalk\ bet\ E\ u\ []\ v$

unfolding *vwalk-bet-def* **by** *simp*

lemma *vwalk-ConsE*:

assumes $vwalk\ E\ (a\ \# \ p)\ p \neq []$
obtains $e \in E\ e = (a, hd\ p)\ vwalk\ E\ p$
using *assms*
by (*metis vwalk-2 hd-Cons-tl*)

lemma *vwalk-reachable*:

$p \neq [] \implies vwalk\ E\ p \implies hd\ p \rightarrow^*_E\ last\ p$
by (*induction p rule: list-nonempty-induct*)

(*auto elim!*: *vwalk-ConsE simp: reachable-def converse-rtrancl-on-into-rtrancl-on dVsI*)

lemma *vwalk-reachable'*:

vwalk E p $\implies p \neq [] \implies \text{hd } p = u \implies \text{last } p = v \implies u \rightarrow^*_E v$
by (*auto dest: vwalk-reachable*)

lemma *vwalkI*: $(x, \text{hd } p) \in E \implies \text{vwalk } E p \implies \text{vwalk } E (x\#p)$
by (*induction p*) (*auto simp: dVsI*)

lemma *reachable-vwalk*:

assumes $u \rightarrow^*_E v$
shows $\exists p. \text{hd } p = u \wedge \text{last } p = v \wedge \text{vwalk } E p \wedge p \neq []$
using *assms*
apply *induction*
apply *force*
apply *clarsimp*
subgoal for $x p$
by (*auto intro!*: *exI[where x=x#p] vwalkI*)
done

lemma *reachable-vwalk-iff*:

$u \rightarrow^*_E v \iff (\exists p. \text{hd } p = u \wedge \text{last } p = v \wedge \text{vwalk } E p \wedge p \neq [])$
by (*auto simp: vwalk-reachable reachable-vwalk*)

lemma *reachable-vwalk-bet-iff*:

$u \rightarrow^*_E v \iff (\exists p. \text{vwalk-bet } E u p v)$
by (*auto simp: reachable-vwalk-iff vwalk-bet-def*)

lemma *reachable-vwalk-betD*:

vwalk-bet E u p v $\implies u \rightarrow^*_E v$
using *iffD2[OF reachable-vwalk-bet-iff]*
by *force*

lemma *vwalk-reachable1*:

vwalk E (u # p @ [v]) $\implies u \rightarrow^+_E v$
by (*induction p arbitrary: u*) (*auto simp add: trancl-into-trancl2*)

lemma *reachable1-vwalk*:

assumes $u \rightarrow^+_E v$
shows $\exists p. \text{vwalk } E (u \# p @ [v])$
proof –
from *assms* **obtain** w **where** $(u, w) \in E w \rightarrow^*_E v$
by (*meson converse-tranclE reachable1-in-dVs(2) reachable1-reachable reachable-refl*)
from *reachable-vwalk[OF this(2)]* **obtain** p **where** $hd p = w \text{ last } p = v \text{ vwalk } E p p \neq []$
by *auto*
then obtain p' **where** $[simp]: p = p' @ [v]$

by (auto intro!: append-butlast-last-id[symmetric])
 with $\langle (u,w) \in E \rangle$ * show ?thesis
 by (auto intro: vwalkI)
 qed

lemma *reachable1-vwalk-iff*:
 $u \rightarrow^+_E v \iff (\exists p. \text{vwalk } E (u \# p @ [v]))$
 by (auto simp: vwalk-reachable1 reachable1-vwalk)

lemma *reachable-vwalk-iff2*:
 $u \rightarrow^*_E v \iff (u = v \wedge u \in dVs E \vee (\exists p. \text{vwalk } E (u \# p @ [v])))$
 by (auto dest: reachable1-vwalk simp: reachable-in-dVs vwalk-reachable1 reachable1-reachable)

lemma *vwalk-remove-cycleE*:
 assumes $\text{vwalk } E (u \# p @ [v])$
 obtains p' where $\text{vwalk } E (u \# p' @ [v])$
 $\text{distinct } p' u \notin \text{set } p' v \notin \text{set } p' \text{set } p' \subseteq \text{set } p$
 using *assms*
proof (induction length p arbitrary: p rule: less-induct)
 case less
 note $\text{prems} = \text{less.prems}(2)$ and $\text{intro} = \text{less.prems}(1)$ and $IH = \text{less.hyps}$
 consider
 $\text{distinct } p u \notin \text{set } p v \notin \text{set } p \mid u \in \text{set } p \mid v \in \text{set } p \mid \neg \text{distinct } p$ by auto
 then consider (goal) ?case
 | (a) $as \ bs$ where $p = as @ u \# bs$ | (b) $as \ bs$ where $p = as @ v \# bs$
 | (between) $x \ as \ bs \ cs$ where $p = as @ x \# bs @ x \# cs$
 using *prems*
 by (cases, auto dest: not-distinct-decomp split-list intro: intro)
 then show ?case
proof cases
 case a
 with *prems* show ?thesis
 by - (rule IH[where $p = bs$], auto 4 3 intro: intro dest: vwalkD)
 next
 case b
 with *prems* have $\text{vwalk } E (u \# as @ v \# [] @ (bs @ [v]))$ by *simp*
 then have $\text{vwalk } E (u \# as @ [v])$ by (auto simp: append-vwalk-pref)
 with b show ?thesis
 by - (rule IH[where $p = as$], auto 4 3 intro: intro)
 next
 case between
 with *prems* have *expanded*: $\text{vwalk } E ((u \# as @ x \# bs) @ x \# cs @ [v])$ by
simp
 then have xv : $\text{vwalk } E (x \# cs @ [v])$ by (rule append-vwalk-suff)
 have ux : $\text{vwalk } E ((u \# as) @ [x])$ using *expanded* by force
 with xv have $\text{vwalk } E ((u \# as) @ x \# cs @ [v])$
 using *vwalk-append2*[OF $ux \ xv$] by auto
 with *between* show ?thesis

by - (rule IH[where p = as @ x # cs], auto 4 3 intro: intro)
 qed
 qed

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

lemma edge-iff-vwalk-bet: (u, v) ∈ E = vwalk-bet E u [u, v] v
 by (auto simp: edges-are-vwalk-bet vwalk-bet-def dVsI)

lemma vwalk-bet-in-vertices: vwalk-bet E u p v ⇒ w ∈ set p ⇒ w ∈ dVs E
 by (auto intro: vwalk-then-in-dVs)

lemma vwalk-bet-hd-neq-last-implies-edges-nonempty:
 assumes vwalk-bet E u p v
 assumes u ≠ v
 shows E ≠ {}
 using assms
 by (induction p) (auto dest: vwalk-then-edge)

lemma vwalk-bet-edges-in-edges: vwalk-bet E u p v ⇒ set (edges-of-vwalk p) ⊆ E
 by (auto simp add: vwalk-bet-def intro: vwalk-ball-edges)

lemma vwalk-bet-prefix-is-vwalk-bet:
 assumes p ≠ []
 assumes vwalk-bet E u (p @ q) v
 shows vwalk-bet E u p (last p)
 using assms
 by (auto simp: vwalk-bet-def dest: append-vwalk-pref)

lemma vwalk-bet-suffix-is-vwalk-bet:
 assumes q ≠ []
 assumes vwalk-bet E u (p @ q) v
 shows vwalk-bet E (hd q) q v
 using assms
 by (auto simp: vwalk-bet-def dest: append-vwalk-suff)

lemma vwalk-bet-append-append-is-vwalk-bet:
 assumes vwalk-bet E u p v
 assumes vwalk-bet E v q w
 assumes vwalk-bet E w r x
 shows vwalk-bet E u (p @ tl q @ tl r) x
 using assms
 by (auto dest: vwalk-bet-transitive)

lemma
 assumes p ≠ []
 shows edges-of-vwalk (p @ q) = edges-of-vwalk p @ edges-of-vwalk ([last p] @ q)
 using assms

by (*simp add: edges-of-vwalk-append-3*)

fun *is-vwalk-bet-vertex-decomp* :: ('v × 'v) set ⇒ 'v list ⇒ 'v ⇒ 'v list × 'v list ⇒ bool **where**
is-vwalk-bet-vertex-decomp E p v (q, r) ⟷ p = q @ tl r ∧ (∃ 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
using *assms*
by (*simp add: vwalk-bet-pref split-vwalk*)

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 –
obtain xs ys **where**
p = xs @ v # ys
using *v-in-p* **by** (*auto simp: in-set-conv-decomp*)
with *p-vwalk*
obtain q r **where**
p = q @ tl r
vwalk-bet E u q v
vwalk-bet E v r w
by (*blast elim: vwalk-bet-vertex-decompE*)
hence *is-vwalk-bet-vertex-decomp* E p v (q, r)
by (*simp add: vwalk-bet-def*)
hence ∃ qr. *is-vwalk-bet-vertex-decomp* E p v qr
by *blast*
thus ?thesis
unfolding *vwalk-bet-vertex-decomp-def*
..
qed

lemma *vwalk-bet-vertex-decompE-2*:

assumes p -vwalk: vwalk-bet E u p w
assumes v -in- p : $v \in \text{set } p$
assumes qr -def: vwalk-bet-vertex-decomp E p $v = (q, r)$
obtains
 $p = q @ \text{tl } r$
vwalk-bet E u q v
vwalk-bet E v r w
proof
have *: is-vwalk-bet-vertex-decomp E p v (q, r)
using *assms* **by** (*auto* *dest*: vwalk-bet-vertex-decomp-is-vwalk-bet-vertex-decomp)
then obtain $u' w'$ **where**
 p -decomp: $p = q @ \text{tl } r$ **and**
 q -vwalk: vwalk-bet E u' q v **and**
 r -vwalk: vwalk-bet E v r w'
by *auto*
hence vwalk-bet E $u' p w'$ **by** (*simp* *add*: vwalk-bet-transitive)
hence $u' = u$ $w' = w$ **using** p -vwalk **by** (*simp*-*all* *add*: vwalk-bet-def)
thus
 $p = q @ \text{tl } r$
vwalk-bet E u q v
vwalk-bet E v r w
using p -decomp q -vwalk r -vwalk **by** *simp*-*all*
qed

definition $v\text{trail} :: ('v \times 'v) \text{set} \Rightarrow 'v \Rightarrow 'v \text{list} \Rightarrow 'v \Rightarrow \text{bool}$ **where**
 $v\text{trail } E$ u p $v = (\text{vwalk-bet } E$ u p $v \wedge \text{distinct } (\text{edges-of-vwalk } p))$

abbreviation $\text{closed-vtrail} :: ('v \times 'v) \text{set} \Rightarrow 'v \text{list} \Rightarrow 'v \Rightarrow \text{bool}$ **where**
 $\text{closed-vtrail } E$ c $v \equiv v\text{trail } E$ v c $v \wedge \text{Suc } 0 < \text{length } c$

lemma $\text{closed-vtrail-implies-Cons}$:
assumes $\text{closed-vtrail } E$ c v
shows $c = v \# \text{tl } c$
using *assms*
by (*auto* *simp* *add*: $v\text{trail-def}$ vwalk-bet-def)

lemma tl-non-empty-conv :
shows $\text{tl } l \neq [] \iff \text{Suc } 0 < \text{length } l$
proof –
have $\text{tl } l \neq [] \iff 0 < \text{length } (\text{tl } l)$
by *blast*
also have $\dots \iff \text{Suc } 0 < \text{length } l$
by *simp*
finally show *?thesis*

qed

lemma $\text{closed-vtrail-implies-tl-nonempty}$:

assumes *closed-vtrail* $E\ c\ v$
shows $tl\ c \neq []$
using *assms*
by (*simp add: tl-non-empty-conv*)

lemma *vtrail-in-vertices*:
 $vtrail\ E\ u\ p\ v \implies w \in set\ p \implies w \in dVs\ E$
by (*auto simp add: vtrail-def intro: vwalk-bet-in-vertices*)

lemma
assumes *vtrail* $E\ u\ p\ v$
shows *vtrail-hd-in-vertices*: $u \in dVs\ E$
and *vtrail-last-in-vertices*: $v \in dVs\ E$
using *assms*
by (*auto intro: vwalk-bet-endpoints simp: vtrail-def*)

lemma *list-set-tl*: $x \in set\ (tl\ xs) \implies x \in set\ xs$
by (*cases xs auto*)

lemma *closed-vtrail-hd-tl-in-vertices*:
assumes *closed-vtrail* $E\ c\ v$
shows $hd\ (tl\ c) \in dVs\ E$
using *assms*
by (*auto intro!: vtrail-in-vertices simp flip: tl-non-empty-conv simp add: list-set-tl*)

lemma *vtrail-prefix-is-vtrail*:
notes *vtrail-def* [*simp*]
assumes $p \neq []$
assumes *vtrail* $E\ u\ (p\ @\ q)\ v$
shows *vtrail* $E\ u\ p\ (last\ p)$
using *assms*
by (*auto simp: vwalk-bet-prefix-is-vwalk-bet edges-of-vwalk-append-3*)

lemma *vtrail-suffix-is-vtrail*:
notes *vtrail-def* [*simp*]
assumes $q \neq []$
assumes *vtrail* $E\ u\ (p\ @\ q)\ v$
shows *vtrail* $E\ (hd\ q)\ q\ v$
using *assms*
by (*auto simp: vwalk-bet-suffix-is-vwalk-bet edges-of-vwalk-append-2[OF <math>q \neq []>]*)

definition *distinct-vwalk-bet* :: $(\ 'v \times\ 'v) set \implies\ 'v \implies\ 'v\ list \implies\ 'v \implies\ bool$ **where**
 $distinct-vwalk-bet\ E\ u\ p\ v = (vwalk-bet\ E\ u\ p\ v \wedge distinct\ p)$

lemma *distinct-vwalk-bet-length-le-card-vertices*:
assumes *distinct-vwalk-bet* $E\ u\ p\ v$
assumes *finite* E
shows $length\ p \leq card\ (dVs\ E)$
using *assms*

unfolding *distinct-vwalk-bet-def*
by (*auto dest!: distinct-card[symmetric] intro!: card-mono simp: vwalk-bet-in-vertices finite-vertices-iff*)

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

assumes *finite E*
shows *finite {(p, u, v). distinct-vwalk-bet E u p v}*
proof (*rule finite-subset*)
have $\bigwedge p u v. vwalk\text{-bet } E u p v \implies distinct\ p \implies length\ p \leq card\ (dVs\ E)$
by (*auto intro!: distinct-vwalk-bet-length-le-card-vertices simp: distinct-vwalk-bet-def assms*)
thus $\{(p, u, v). distinct\text{-vwalk-bet } E u p v\} \subseteq$
 $\{p. set\ p \subseteq dVs\ E \wedge length\ p \leq card\ (dVs\ E)\} \times dVs\ E \times dVs\ E$
by (*auto simp: distinct-vwalk-bet-def vwalk-bet-in-vertices*)
show *finite ...*
by (*auto intro!: finite-cartesian-product finite-lists-length-le simp: assms finite-vertices-iff*)

qed

lemma *distinct-vwalk-bets-finite:*

finite E \implies finite {p. distinct-vwalk-bet E u p v}
by (*rule finite-surj[OF distinct-vwalk-bet-triples-finite] auto*)

3 Wwalks to paths (as opposed to arc walks (*awalk-to-apat* before))

fun *is-closed-decomp* :: $('v \times 'v)\ set \Rightarrow 'v\ list \Rightarrow 'v\ list \times 'v\ list \times 'v\ list \Rightarrow bool$
where

is-closed-decomp E p (q, r, s) \longleftrightarrow
 $p = q @ tl\ r @ tl\ s \wedge$
 $(\exists u v w. vwalk\text{-bet } E u q v \wedge closed\text{-vwalk-bet } E r v \wedge vwalk\text{-bet } E v s w) \wedge$
distinct q

definition *closed-vwalk-bet-decomp* :: $('v \times 'v)\ set \Rightarrow 'v\ list \Rightarrow 'v\ list \times 'v\ list \times 'v\ list$ **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

using *assms*
by *auto (metis Cons-eq-appendI vwalk-bet-vertex-decompE)*

lemma *closed-vwalk-bet-decomp-is-closed-decomp*:
assumes *p-vwalk*: *vwalk-bet E u p v*
assumes *p-not-distinct*: \neg *distinct p*
shows *is-closed-decomp E p (closed-vwalk-bet-decomp E p)*
proof –
obtain *xs y ys zs* **where**
p = xs @ y # ys @ y # zs **and**
xs-distinct: *distinct xs* **and**
y-not-in-xs: $y \notin$ *set xs*
using *p-not-distinct not-distinct-decomp*
by (*smt append.assoc append.simps(2) in-set-conv-decomp-first not-distinct-conv-prefix*)
from *p-vwalk this(1)*
obtain *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
by (*erule closed-vwalk-bet-decompE*)
moreover **hence**
distinct q
Suc 0 < length r
using *xs-distinct y-not-in-xs*
by *simp+*
ultimately **have**
 $\exists q r s.$
 $p = q @ tl r @ tl s \wedge$
 $(\exists u v w. vwalk-bet E u q v \wedge closed-vwalk-bet E r v \wedge vwalk-bet E v s w) \wedge$
distinct q
by *blast*
hence $\exists qrs.$ *is-closed-decomp E p qrs* **by** *simp*
thus *?thesis* **unfolding** *closed-vwalk-bet-decomp-def ..*
qed

lemma *closed-vwalk-bet-decompE-2*:
assumes *p-vwalk*: *vwalk-bet E u p v*
assumes *p-not-distinct*: \neg *distinct p*
assumes *qrs-def*: *closed-vwalk-bet-decomp E p = (q, r, s)*
obtains
p = q @ tl r @ tl s
 $\exists w. vwalk-bet E u q w \wedge closed-vwalk-bet E r w \wedge vwalk-bet E w s v$
distinct q
proof –
have *is-closed-decomp E p (q, r, s)*

```

unfolding qrs-def[symmetric]
using p-vwalk p-not-distinct
by (rule closed-vwalk-bet-decomp-is-closed-decomp)
then obtain u' w' v' where
  p-decomp:  $p = q @ tl r @ tl s$  and
  q-distinct: distinct q and
  vwalks: vwalk-bet E u' q w'
  closed-vwalk-bet E r w'
  vwalk-bet E w' s v'
by auto
hence vwalk-bet E u' p v'
by (auto intro: vwalk-bet-append-append-is-vwalk-bet)
hence  $u' = u \ v' = v$ 
using p-vwalk
by (simp-all add: vwalk-bet-def)
hence  $\exists w. \text{vwalk-bet } E \ u \ q \ w \wedge \text{closed-vwalk-bet } E \ r \ w \wedge \text{vwalk-bet } E \ w \ s \ v$ 
using vwalks by blast
with p-decomp q-distinct that
show ?thesis by blast
qed

function vwalk-bet-to-distinct :: ('v × 'v) set ⇒ 'v list ⇒ 'v list where
  vwalk-bet-to-distinct E p =
    (if ( $\exists u \ v. \text{vwalk-bet } E \ u \ p \ v$ )  $\wedge \neg \text{distinct } p$ 
      then let (q, r, s) = closed-vwalk-bet-decomp E p in vwalk-bet-to-distinct E (q
@ tl s)
      else p)
by auto

termination vwalk-bet-to-distinct
proof (relation measure (length ∘ snd))
fix E p qrs rs q r s
assume
  p-not-vwalk: ( $\exists u \ v. \text{vwalk-bet } E \ u \ p \ v$ )  $\wedge \neg \text{distinct } p$  and
  assms: qrs = closed-vwalk-bet-decomp E p
  (q, rs) = qrs
  (r, s) = rs
then obtain u v where
  p-vwalk: vwalk-bet E u p v
by blast
hence (q, r, s) = closed-vwalk-bet-decomp E p
using assms
by simp
then obtain
   $p = q @ tl r @ tl s$ 
   $Suc \ 0 < \text{length } r$ 
using p-vwalk p-not-vwalk
by (elim closed-vwalk-bet-decompE-2) auto
thus ( $(E, (q @ tl s)), E, p$ )  $\in \text{measure } (\text{length} \circ \text{snd})$ 

```

by *auto*
qed *simp*

lemma *vwalk-bet-to-distinct-induct* [*consumes 1, case-names path decomp*]:

assumes *vwalk-bet* $E\ u\ p\ v$

assumes *distinct*: $\bigwedge p. \llbracket vwalk-bet\ E\ u\ p\ v; distinct\ p \rrbracket \implies P\ p$

assumes

decomp: $\bigwedge p\ q\ r\ s. \llbracket vwalk-bet\ E\ u\ p\ v; \neg\ distinct\ p;$

closed-vwalk-bet-decomp $E\ p = (q, r, s); P\ (q\ @\ tl\ s) \rrbracket \implies P\ p$

shows $P\ p$

using *assms(1)*

proof (*induct length p arbitrary: p rule: less-induct*)

case *less*

show *?case*

proof (*cases distinct p*)

case *True*

with *less.prem*s

show *?thesis*

by (*rule distinct*)

next

case *False*

obtain $q\ r\ s$ **where**

grs-def: *closed-vwalk-bet-decomp* $E\ p = (q, r, s)$

by (*cases closed-vwalk-bet-decomp E p*)

with *less.prem*s *False*

obtain

$p = q\ @\ tl\ r\ @\ tl\ s$

$\exists w. vwalk-bet\ E\ u\ q\ w \wedge closed-vwalk-bet\ E\ r\ w \wedge vwalk-bet\ E\ w\ s\ v$

by (*elim closed-vwalk-bet-decompE-2*)

hence

$length\ (q\ @\ tl\ s) < length\ p$

vwalk-bet $E\ u\ (q\ @\ tl\ s)\ v$

by (*auto simp: tl-non-empty-conv intro: vwalk-bet-transitive*)

hence $P\ (q\ @\ tl\ s)$

by (*rule less.hyps*)

with *less.prem*s *False grs-def*

show *?thesis*

by (*rule decomp*)

qed

qed

lemma *vwalk-bet-to-distinct-is-distinct-vwalk-bet*:

assumes *vwalk-bet* $E\ u\ p\ v$

shows *distinct-vwalk-bet* $E\ u\ (vwalk-bet-to-distinct\ E\ p)\ v$

using *assms*

by (*induction rule: vwalk-bet-to-distinct-induct*) (*auto simp: distinct-vwalk-bet-def*)

lemma *vwalk-betE[elim?]*:

assumes *vwalk-bet* $E\ u\ p\ v$

assumes *singleton*: $\llbracket v \in dVs\ E; u = v \rrbracket \implies P$
assumes
step: $\bigwedge p' x. \llbracket p = u \# x \# p'; (u, x) \in E; vwalk\text{-bet}\ E\ x\ (x \# p')\ v \rrbracket \implies P$
shows P
using *assms*
by (*induction rule: induct-vwalk-bet*) *auto*

lemma *vwalk-subset*:
 $\llbracket vwalk\ G\ p; G \subseteq G' \rrbracket \implies vwalk\ G'\ p$
by (*induction p rule: vwalk.induct*) (*auto simp: dVs-def*)

lemma *vwalk-bet-subset*:
 $\llbracket vwalk\text{-bet}\ G\ u\ p\ v; G \subseteq G' \rrbracket \implies vwalk\text{-bet}\ G'\ u\ p\ v$
using *vwalk-subset*
by (*auto simp: vwalk-bet-def*)

lemma *reachable-subset[intro]*:
 $\llbracket u \rightarrow^*_G\ v; G \subseteq G' \rrbracket \implies u \rightarrow^*_{G'}\ v$
by(*auto simp add: reachable-vwalk-bet-iff dest: vwalk-bet-subset*)

lemma *reachable-Union-1[intro]*:
 $\llbracket u \rightarrow^*_G\ v \rrbracket \implies u \rightarrow^*_{G \cup G'}\ v$
 $\llbracket u \rightarrow^*_{G'}\ v \rrbracket \implies u \rightarrow^*_{G' \cup G}\ v$
by *auto*

lemma *reachableE[elim?]*:
assumes *reachable* $E\ u\ v$
assumes *singleton*: $\llbracket v \in dVs\ E; u = v \rrbracket \implies P$
assumes
step: $\bigwedge x. \llbracket (u, x) \in E; reachable\ E\ x\ v \rrbracket \implies P$
shows P
using *assms*
by (*metis converse-tranclE reachable1-reachable reachable-in-dVs(2) reachable-neq-reachable1 reachable-refl*)

lemma *vwalk-insertE[case-names nil sing1 sing2 in-e in-E]*:
 $\llbracket vwalk\ (insert\ e\ E)\ p; (p = [] \implies P); (\bigwedge u\ v. p = [v] \implies e = (u, v) \implies P); (\bigwedge u\ v. p = [v] \implies e = (v, u) \implies P); (\bigwedge v. p = [v] \implies v \in dVs\ E \implies P); (\bigwedge p'\ v1\ v2. \llbracket vwalk\ \{e\}\ [v1, v2]; vwalk\ (insert\ e\ E)\ (v2 \# p'); p = v1 \# v2 \# p' \rrbracket \implies P); (\bigwedge p'\ v1\ v2. \llbracket vwalk\ E\ [v1, v2]; vwalk\ (insert\ e\ E)\ (v2 \# p'); p = v1 \# v2 \# p' \rrbracket \implies P) \rrbracket \implies P$
proof(*induction rule: vwalk.induct*)
case *vwalk0*

```

then show ?case
  by auto
next
  case (vwalk1 v)
  then show ?case
    by (auto simp: dVs-def)
next
  case (vwalk2 v v' vs)
  then show ?case
    apply (auto simp: dVs-def)
    by (metis insertCI)
qed

```

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-bet (insert e E) v1 p v2;
  ((v1 ∈ dVs (insert e E); v1 = v2; p = []) ⇒ P);
  (∧ u v. p = [u] ⇒ e = (u,v) ⇒ P);
  (∧ u v. p = [v] ⇒ e = (u,v) ⇒ P);
  (∧ v. p = [v] ⇒ v = v1 ⇒ v = v2 ⇒ v ∈ dVs E ⇒ P);
  (∧ p' v3. [[vwalk-bet {e} v1 [v1,v3] v3; vwalk-bet (insert e E) v3 (v3 # p') v2;
    p = v1 # v3 # p'] ⇒ P);
  (∧ p' v3. [[vwalk-bet E v1 [v1,v3] v3; vwalk-bet (insert e E) v3 (v3 # p') v2;
    p = v1 # v3 # p'] ⇒ P)]]
  ⇒ P
unfolding vwalk-bet-def
apply safe
apply(erule vwalk-insertE)
by (simp | force)+

```

find-theorems *name: induct vwalk-bet*

lemma *vwalk-bet2*[*simp*]:

```

vwalk-bet G u (u # v # vs) b ⇔ ((u,v) ∈ G ∧ vwalk-bet G v (v # vs) b)
by(auto simp: vwalk-bet-def)

```

lemma *butlast-vwalk-is-vwalk*: *vwalk E p ⇒ vwalk E (butlast p)*

by (*induction p rule: vwalk.induct, auto*)

lemma *vwalk-concat-2*:

assumes *vwalk E p vwalk E q q ≠ [] p ≠ [] ⇒ last p = hd q*

shows *vwalk E (butlast p @ q)*

using *assms*

by (*induction rule: vwalk.induct*) (*auto simp add: butlast-vwalk-is-vwalk neq-Nil-conv*)

lemma *vwalk-bet-transitive-2*:

assumes *vwalk-bet* $E\ u\ p\ v\ vwalk\text{-bet}\ E\ v\ q\ w$

shows *vwalk-bet* $E\ u\ (butlast\ p\ @\ q)\ w$

using *assms*

unfolding *vwalk-bet-def*

by (*auto intro!*: *vwalk-concat-2 simp: last-append singleton-hd-last' neq-Nil-conv*)

lemma *vwalk-not-vwalk*:

$\llbracket vwalk\ G\ p;\ \neg vwalk\ G'\ p \rrbracket \implies$

$(\exists (u,v) \in set\ (edges\text{-of}\text{-}vwalk\ p). (u,v) \in (G - G')) \vee$

$(\exists v \in set\ p. v \in (dVs\ G - dVs\ G'))$

by (*induction rule: vwalk.induct*) (*auto simp: dVs-def*)

lemma *vwalk-not-vwalk-2*:

$\llbracket vwalk\ G\ p;\ \neg vwalk\ G'\ p;\ length\ p \geq 2 \rrbracket \implies$

$(\exists (u,v) \in set\ (edges\text{-of}\text{-}vwalk\ p). (u,v) \in (G - G'))$

apply (*induction rule: vwalk.induct*)

apply (*auto simp: dVs-def*)

by (*metis Suc-leI dVsI(2) length-greater-0-conv vwalk-1*)

lemma *vwalk-not-vwalk-elim*:

$\llbracket vwalk\ G\ p;\ \neg vwalk\ G'\ p \rrbracket \implies$

$\llbracket \bigwedge u\ v. \llbracket (u,v) \in set\ (edges\text{-of}\text{-}vwalk\ p); (u,v) \in (G - G') \rrbracket \implies P;$

$\bigwedge v. \llbracket v \in set\ p; v \in (dVs\ G - dVs\ G') \rrbracket \implies P \rrbracket \implies P$

using *vwalk-not-vwalk*

by *blast*

lemma *vwalk-not-vwalk-elim-2*:

$\llbracket vwalk\ G\ p;\ \neg vwalk\ G'\ p;\ length\ p \geq 2 \rrbracket \implies$

$\llbracket \bigwedge u\ v. \llbracket (u,v) \in set\ (edges\text{-of}\text{-}vwalk\ p); (u,v) \in (G - G') \rrbracket \implies P$

$\rrbracket \implies P$

using *vwalk-not-vwalk-2*

by *blast*

lemma *vwalk-bet-not-vwalk-bet*:

$\llbracket vwalk\text{-bet}\ G\ u\ p\ v;\ \neg vwalk\text{-bet}\ G'\ u\ p\ v \rrbracket \implies$

$(\exists (u,v) \in set\ (edges\text{-of}\text{-}vwalk\ p). (u,v) \in (G - G')) \vee$

$(\exists v \in set\ p. v \in (dVs\ G - dVs\ G'))$

using *vwalk-not-vwalk*

by (*auto simp: vwalk-bet-def*)

lemma *vwalk-bet-not-vwalk-bet-elim*:

$\llbracket vwalk\text{-bet}\ G\ u\ p\ v;\ \neg vwalk\text{-bet}\ G'\ u\ p\ v \rrbracket \implies$

$\llbracket \bigwedge u\ v. \llbracket (u,v) \in set\ (edges\text{-of}\text{-}vwalk\ p); (u,v) \in (G - G') \rrbracket \implies P;$

$\bigwedge v. \llbracket v \in set\ p; v \in (dVs\ G - dVs\ G') \rrbracket \implies P \rrbracket \implies P$

using *vwalk-not-vwalk-elim*

apply (*auto simp: vwalk-bet-def*)

```

by blast

lemma vwalk-bet-not-vwalk-bet-elim-2:
   $\llbracket \text{vwalk-bet } G \ u \ p \ v; \neg \text{vwalk-bet } G' \ u \ p \ v; \text{length } p \geq 2 \rrbracket \implies$ 
   $\llbracket \bigwedge u \ v. \llbracket (u,v) \in \text{set } (\text{edges-of-vwalk } p); (u,v) \in (G - G') \rrbracket \implies P \rrbracket \implies P$ 
using vwalk-not-vwalk-elim-2
apply (auto simp: vwalk-bet-def)
by blast

lemma vwalk-bet-props:
   $\text{vwalk-bet } G \ u \ p \ v \implies (\llbracket \text{vwalk } G \ p; p \neq []; \text{hd } p = u; \text{last } p = v \rrbracket \implies P) \implies P$ 
by (auto simp: vwalk-bet-def)

lemma no-outgoing-last:
   $\llbracket \text{vwalk } G \ p; \bigwedge v. (u,v) \notin G; u \in \text{set } p \rrbracket \implies \text{last } p = u$ 
by (induction rule: vwalk.induct) (auto simp: dVs-def)

lemma not-vwalk-bet-empty:  $\neg \text{Vwalk.vwalk-bet } \{ \} \ u \ p \ v$ 
by (auto simp: vwalk-bet-def neq-Nil-conv split: if-splits)

lemma edges-in-vwalk-split:
   $(u, v) \in \text{set } (\text{edges-of-vwalk } p) \implies \exists p1 \ p2. p = p1 \ @ [u,v] @ p2$ 
proof (induction p rule: edges-of-vwalk.induct)
case ( $\exists a \ b \ p$ )
show ?case
proof (cases (a, b) = (u, v))
case True
hence  $a \# b \# p = [u, v] \ @ \ p$  by auto
then show ?thesis by auto
next
case False
hence  $(u, v) \in \text{set } (\text{edges-of-vwalk } (b \# p))$ 
using  $\exists(2)$  by auto
then obtain  $p1 \ p2$  where  $b \# p = p1 \ @ [u, v] \ @ p2$ 
using  $\exists(1)$  by auto
hence  $a \# b \# p = (a \# p1) \ @ [u, v] \ @ p2$  by auto
then show ?thesis by fast
qed
qed simp+

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))

by (*simp-all add: BitM-plus-one eSuc-enat numeral-plus-one[symmetric] zero-enat-def one-enat-def*)

declare *eSuc-enat[symmetric, simp]*

end

theory *Dist*

imports *Enat-Misc Vwalk*

begin

4 Distances

4.1 Distance from a vertex

definition *distance::('v × 'v) set ⇒ 'v ⇒ 'v ⇒ 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 ⇒ distance G u v ≤ length p - 1

by (*auto simp: distance-def image-def intro!: complete-lattice-class.Inf-lower enat-ile*)

lemma *reachable-dist*:

reachable G u v ⇒ distance G u v < ∞

apply(*clarsimp simp add: reachable-vwalk-bet-iff*)

by (*auto simp: distance-def image-def intro!: complete-lattice-class.Inf-lower enat-ile*)

lemma *unreachable-dist*:

¬reachable G u v ⇒ distance G u v = ∞

apply(*clarsimp simp add: reachable-vwalk-bet-iff*)

by (*auto simp: distance-def image-def intro!: complete-lattice-class.Inf-lower enat-ile*)

lemma *dist-reachable*:

distance G u v < ∞ ⇒ reachable G u v

using *wellorder-InfI*

by(*fastforce simp: distance-def image-def reachable-vwalk-bet-iff*)

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*

using *assms*

apply(*clarsimp simp add: reachable-vwalk-bet-iff distance-def image-def*)

by (*smt (verit, del-Insts) Collect-disj-eq Inf-lower enat-ile mem-Collect-eq not-infinity-eq wellorder-InfI*)

lemma *triangle-ineq-reachable*:
assumes *reachable G u v reachable G v w*
shows $\text{distance } G \ u \ w \leq \text{distance } G \ u \ v + \text{distance } G \ v \ w$
proof –
obtain $p \ q$
where $p\text{-}q$: *vwalk-bet G u p v distance G u v = length p - 1*
vwalk-bet G v q w distance G v w = length q - 1
using *assms*
by (*auto elim!*: *reachable-dist-2*)
hence *vwalk-bet G u (p @ tl q) w*
by(*auto intro!*: *vwalk-bet-transitive*)
hence $\text{distance } G \ u \ w \leq \text{length } (p \ @ \ tl \ q) - 1$
by (*auto dest!*: *vwalk-bet-dist*)
moreover have $\text{length } (p \ @ \ tl \ q) = \text{length } p + (\text{length } q - 1)$
using $\langle \text{vwalk-bet } G \ v \ q \ w \rangle$
by (*cases q; auto*)
ultimately have $\text{distance } G \ u \ w \leq \text{length } p + (\text{length } q - 1) - 1$
by (*auto simp: eval-nat-numeral*)
thus *?thesis*
using $p\text{-}q(1)$
by(*cases p; auto simp add: p-q(2,4) eval-nat-numeral*)
qed

lemma *triangle-ineq*:
 $\text{distance } G \ u \ w \leq \text{distance } G \ u \ v + \text{distance } G \ v \ w$
proof(*cases reachable G u v*)
case *reach-u-v: True*
then show *?thesis*
proof(*cases reachable G v w*)
case *reach-v-w: True*
then show *?thesis*
using *triangle-ineq-reachable reach-u-v reach-v-w*
by *auto*
next
case *not-reach-v-w: False*
hence $\text{distance } G \ v \ w = \infty$
by (*simp add: unreachable-dist*)
then show *?thesis*
by *simp*
qed
next
case *not-reach-u-v: False*
hence $\text{distance } G \ u \ v = \infty$
by (*simp add: unreachable-dist*)
then show *?thesis*
by *simp*
qed

lemma *distance-split*:

$\llbracket \text{distance } G \ u \ v \neq \infty; \text{distance } G \ u \ v = \text{distance } G \ u \ w + \text{distance } G \ w \ v \rrbracket \implies$
 $\exists w'. \text{reachable } G \ u \ w' \wedge \text{distance } G \ u \ w' = \text{distance } G \ u \ w \wedge$
 $\text{reachable } G \ w' \ v \wedge \text{distance } G \ w' \ v = \text{distance } G \ w' \ v$
by (*metis dist-reachable enat-ord-simps(4) plus-eq-infty-iff-enat*)

lemma *dist-inf*: $v \notin dVs \ G \implies \text{distance } G \ u \ v = \infty$

proof(*rule ccontr, goal-cases*)

case 1

hence *reachable* $G \ u \ v$

by(*auto intro!: dist-reachable*)

hence $v \in dVs \ G$

by (*simp add: reachable-in-dVs(2)*)

then show *?case*

using 1

by *auto*

qed

lemma *dist-inf-2*: $v \notin dVs \ G \implies \text{distance } G \ v \ u = \infty$

proof(*rule ccontr, goal-cases*)

case 1

hence *reachable* $G \ v \ u$

by(*auto intro!: dist-reachable*)

hence $v \in dVs \ G$

by (*simp add: reachable-in-dVs(1)*)

then show *?case*

using 1

by *auto*

qed

lemma *dist-eq*: $\llbracket \bigwedge p. Vwalk.vwalk-bet \ G' \ u \ p \ v \implies Vwalk.vwalk-bet \ G \ u \ (\text{map } f \ p) \ v \rrbracket \implies$

$\text{distance } G \ u \ v \leq \text{distance } G' \ u \ v$

apply(*auto simp: distance-def image-def intro!: Inf-lower*)

apply (*smt (verit, ccfv-threshold) Un-iff length-map mem-Collect-eq wellorder-InfI*)

by (*meson vwalk-bet-nonempty'*)

lemma *distance-subset*: $G \subseteq G' \implies \text{distance } G' \ u \ v \leq \text{distance } G \ u \ v$

by (*metis enat-ord-simps(3) reachable-dist-2 unreachable-dist vwalk-bet-dist vwalk-bet-subset*)

lemma *distance-neighbourhood*:

$\llbracket v \in \text{neighbourhood } G \ u \rrbracket \implies \text{distance } G \ u \ v \leq 1$

proof(*goal-cases*)

case 1

hence *vwalk-bet* $G \ u \ [u, v] \ v$

by *auto*

moreover have *length* $[u, v] = 2$

by *auto*
 ultimately show *?case*
 by(*force dest!*: *vwalk-bet-dist simp: one-enat-def*)
 qed

4.2 Shortest Paths

definition *shortest-path*::('v × 'v) set ⇒ 'v ⇒ 'v list ⇒ 'v ⇒ bool **where**
shortest-path G u p v = (distance G u v = length p - 1 ∧ vwalk-bet G u p v)

lemma *shortest-path-props*[*elim*]:
shortest-path G u p v ⇒ (⟦distance G u v = length p - 1; vwalk-bet G u p v⟧
 ⇒ P) ⇒ P
 by (*auto simp: shortest-path-def*)

lemma *shortest-path-intro*:
 ⟦distance G u v = length p - 1; vwalk-bet G u p v⟧ ⇒ *shortest-path* G u p v
 by (*auto simp: shortest-path-def*)

lemma *shortest-path-vwalk*: *shortest-path* G u p v ⇒ vwalk-bet G u p v
 by(*auto simp: shortest-path-def*)

lemma *shortest-path-dist*: *shortest-path* G u p v ⇒ distance G u v = length p - 1
 by(*auto simp: shortest-path-def*)

lemma *shortest-path-split-1*:
 ⟦*shortest-path* G u (p1 @ x # p2) v⟧ ⇒ *shortest-path* G x (x # p2) v
proof(*goal-cases*)
 case 1
hence vwalk-bet G u (p1 @ [x]) x vwalk-bet G x (x # p2) v
 by (*auto dest!*: *shortest-path-vwalk simp: split-vwalk vwalk-bet-pref*)

hence *reachable* G x v *reachable* G u x
 by (*auto dest: reachable-vwalk-betD*)

have distance G x v ≥ length (x # p2) - 1

proof(*rule ccontr, goal-cases*)

case 1

moreover from ⟨*reachable* G x v⟩

obtain p **where** vwalk-bet G x p v distance G x v = enat (length p - 1)

by (*rule reachable-dist-2*)

ultimately have length p - 1 < length (x # p2) - 1

by *auto*

hence length (p1 @ p) - 1 < length (p1 @ x # p2) - 1

by *auto*

have vwalk-bet G u ((p1 @ [x]) @ (tl p)) v

```

    using ⟨vwalk-bet G u (p1 @ [x]) x⟩ ⟨vwalk-bet G x p v⟩
    by (fastforce intro: vwalk-bet-transitive)
  moreover have p = x # (tl p)
    using ⟨vwalk-bet G x p v⟩
    by (fastforce dest: hd-of-vwalk-bet)
  ultimately have distance G u v ≤ length (p1 @ p) - 1
    using vwalk-bet-dist
    by fastforce
  moreover have distance G u v = length (p1 @ x # p2) - 1
    using ⟨shortest-path G u (p1 @ x # p2) v⟩
    by (rule shortest-path-dist)
  ultimately show ?case
    using ⟨length (p1 @ p) - 1 < length (p1 @ x # p2) - 1⟩
    by auto
qed
moreover have distance G x v ≤ length (x # p2) - 1
  using ⟨vwalk-bet G x (x # p2) v⟩
  by (force intro!: vwalk-bet-dist)

ultimately show ?case
  using ⟨vwalk-bet G x (x # p2) v⟩
  by (auto simp: shortest-path-def)
qed

lemma shortest-path-split-2:
  [[shortest-path G u (p1 @ x # p2) v] ⇒ shortest-path G u (p1 @ [x]) x
proof(goal-cases)
  case 1
  hence vwalk-bet G u (p1 @ [x]) x vwalk-bet G x (x # p2) v
    by (auto dest!: shortest-path-vwalk simp: split-vwalk vwalk-bet-pref)

  hence reachable G x v reachable G u x
    by (auto dest: reachable-vwalk-betD)

  have distance G u x ≥ length (p1 @ [x]) - 1
  proof(rule ccontr, goal-cases)
    case 1
    moreover from ⟨reachable G u x⟩
    obtain p where vwalk-bet G u p x distance G u x = enat (length p - 1)
      by (rule reachable-dist-2)
    ultimately have length p - 1 < length (p1 @ [x]) - 1
      by auto
    hence length (p @ p2) - 1 < length (p1 @ x # p2) - 1
      by auto
    have vwalk-bet G u (butlast p @ x # p2) v
      using ⟨vwalk-bet G u p x⟩ ⟨vwalk-bet G x (x # p2) v⟩
      by (simp add: vwalk-bet-transitive-2)
    moreover have p = (butlast p) @ [x]
      using ⟨vwalk-bet G u p x⟩

```

```

    by (fastforce dest: last-of-vwalk-bet')
  moreover have length p > 0
    using ‹vwalk-bet G u p x›
    by force
  ultimately have distance G u v ≤ length (p @ p2) - 1
    by (auto dest!: vwalk-bet-dist simp: neq-Nil-conv)
  moreover have distance G u v = length (p1 @ x # p2) - 1
    using ‹shortest-path G u (p1 @ x # p2) v›
    by (rule shortest-path-dist)
  ultimately show ?case
    using ‹length (p @ p2) - 1 < length (p1 @ x # p2) - 1›
    by auto
qed
moreover have distance G u x ≤ length (p1 @ [x]) - 1
  using ‹vwalk-bet G u (p1 @ [x]) x›
  by (force intro!: vwalk-bet-dist)

ultimately show ?case
  using ‹vwalk-bet G u (p1 @ [x]) x›
  by (auto simp: shortest-path-def)
qed

lemma shortest-path-split-distance:
  ‹shortest-path G u (p1 @ x # p2) v› ⇒ distance G u x ≤ distance G u v
  using shortest-path-split-2[where G = G and u = u and ?p1.0 = p1 and x =
x] shortest-path-dist
  by force

lemma shortest-path-split-distance':
  ‹x ∈ set p; shortest-path G u p v› ⇒ distance G u x ≤ distance G u v
  apply(subst (asm) in-set-conv-decomp-last)
  using shortest-path-split-distance
  by auto

lemma shortest-path-exists:
  assumes reachable G u v
  obtains p where shortest-path G u p v
  using assms
  by (force elim!: reachable-dist-2 simp: shortest-path-def)

lemma shortest-path-exists-2:
  assumes distance G u v < ∞
  obtains p where shortest-path G u p v
  using assms
  by (force dest!: dist-reachable elim!: shortest-path-exists simp: shortest-path-def)

lemma not-distinct-props:
  ¬distinct xs ⇒ (∧x1 x2 xs1 xs2 xs3. ‹xs = xs1 @ x1 # xs2 @ x2 # xs3; x1 =
x2› ⇒ P) ⇒ P

```

using *not-distinct-decomp*
by *fastforce*

lemma *shortest-path-distinct*:
 $shortest-path\ G\ u\ p\ v \implies distinct\ p$
 apply(*erule shortest-path-props*)
 apply(*rule ccontr*)
 apply(*erule not-distinct-props*)
proof(*goal-cases*)
 case (*1 x1 x2 xs1 xs2 xs3*)
 hence $Vwalk.vwalk-bet\ G\ u\ (xs1\ @\ x2\ \#\ xs3)\ v$
 using *vwalk-bet-transitive-2 closed-vwalk-bet-decompE*
 by (*smt (verit, best) butlast-snoc*)
 hence $distance\ G\ u\ v \leq length\ (xs1\ @\ x2\ \#\ xs3) - 1$
 using *vwalk-bet-dist*
 by *force*
 moreover have $length\ (xs1\ @\ x2\ \#\ xs3) < length\ p$
 by(*auto simp: <p = xs1 @ x1 # xs2 @ x2 # xs3>*)
 ultimately show *?case*
 using $<distance\ G\ u\ v = enat\ (length\ p - 1)>$
 by *auto*
qed

lemma *diet-eq'*: $\llbracket \bigwedge p. shortest-path\ G'\ u\ p\ v \implies shortest-path\ G\ u\ (map\ f\ p)\ v \rrbracket$
 \implies
 $distance\ G\ u\ v \leq distance\ G'\ u\ v$
 apply(*auto simp: shortest-path-def*)
 by (*metis One-nat-def Orderings.order-eq-iff order.strict-implies-order reachable-dist reachable-dist-2 unreachable-dist*)

lemma *distance-0*:
 $(u = v \wedge v \in dVs\ G) \longleftrightarrow distance\ G\ u\ v = 0$
proof(*safe, goal-cases*)
 case *1*
 hence $vwalk-bet\ G\ v\ [v]\ v$
 by *auto*
 moreover have $length\ [v] = 1$
 by *auto*
 ultimately show *?case*
 using *zero-enat-def*
 by(*auto dest!: vwalk-bet-dist simp:*)
next
 case *2*
 hence $distance\ G\ u\ v < \infty$
 by *auto*
 then obtain *p* where $shortest-path\ G\ u\ p\ v$
 by(*erule shortest-path-exists-2*)
 hence $length\ p = 1\ vwalk-bet\ G\ u\ p\ v$
 using $<distance\ G\ u\ v = 0>$

apply (*auto simp: shortest-path-def*)
by (*metis diff-Suc-Suc diff-zero enat-0-iff(2) hd-of-vwalk-bet length-Cons*)
then obtain x **where** $p = [x]$
by (*auto simp: length-Suc-conv*)
then have $x = u$ $x = v$ $x \in dVs\ G$
using $\langle vwalk\text{-bet}\ G\ u\ p\ v \rangle$ *hd-of-vwalk-bet last-of-vwalk-bet*
by (*fastforce simp: vwalk-bet-in-vertices*)
then show $u = v$ $v \in dVs\ G$
by *auto*
qed

lemma *distance-neighbourhood'*:
 $\llbracket v \in neighbourhood\ G\ u \rrbracket \implies distance\ G\ x\ v \leq distance\ G\ x\ u + 1$
using *triangle-ineq distance-neighbourhood*
by (*metis add commute add-mono-thms-linordered-semiring(3) order-trans*)

lemma *Suc-le-length-iff-2*:
 $(Suc\ n \leq length\ xs) = (\exists x\ ys. xs = ys @ [x] \wedge n \leq length\ ys)$
by (*metis Suc-le-D Suc-le-mono length-Suc-conv-rev*)

lemma *distance-parent*:
 $\llbracket distance\ G\ u\ v < \infty; u \neq v \rrbracket \implies$
 $\exists w. distance\ G\ u\ w + 1 = distance\ G\ u\ v \wedge v \in neighbourhood\ G\ w$

proof(*goal-cases*)

case 1

then obtain p **where** *shortest-path* $G\ u\ p\ v$
by(*force dest!: dist-reachable elim!: shortest-path-exists*)
hence $length\ p \geq 2$
using $\langle u \neq v \rangle$

by(*auto dest!: shortest-path-vwalk simp: Suc-le-length-iff eval-nat-numeral elim: vwalk-betE*)

then obtain p' $x\ y$ **where** $p = p' @ [x, y]$
by(*auto simp: Suc-le-length-iff-2 eval-nat-numeral*)

hence *shortest-path* $G\ u\ (p' @ [x])\ x$
using $\langle shortest\text{-path}\ G\ u\ p\ v \rangle$
by (*fastforce dest: shortest-path-split-2*)

hence $distance\ G\ u\ x + 1 = distance\ G\ u\ v$
using $\langle shortest\text{-path}\ G\ u\ p\ v \rangle$ $\langle p = p' @ [x, y] \rangle$
by (*auto dest!: shortest-path-dist simp: eSuc-plus-1*)

moreover have $y = v$
using $\langle shortest\text{-path}\ G\ u\ p\ v \rangle$ $\langle p = p' @ [x, y] \rangle$

by(*force simp: $\langle p = p' @ [x, y] \rangle$ dest!: shortest-path-vwalk intro!: vwalk-bet-snoc***[where**

$p = p' @ [x]$ **])**

moreover have $y \in neighbourhood\ G\ x$
using $\langle shortest\text{-path}\ G\ u\ p\ v \rangle$ $\langle p = p' @ [x, y] \rangle$ *vwalk-bet-suffix-is-vwalk-bet*

by(*fastforce simp*: $\langle p = p' @ [x,y] \rangle$ *dest!*: *shortest-path-vwalk*)
 ultimately show *?thesis*
 by *auto*
 qed

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 ∉ dVs G ⇒ distance-set G U v = ∞
 by(*auto simp*: *distance-set-def INF-eqI dist-inf*)

lemma *dist-set-mem[intro]*: u ∈ U ⇒ distance-set G U v ≤ distance G u v
 by (*auto simp*: *distance-set-def wellorder-Inf-le1*)

lemma *dist-not-inf''*: $\llbracket \text{distance-set } G \ U \ v \neq \infty; u \in U; \text{distance } G \ u \ v = \text{distance-set } G \ U \ v \rrbracket$

$$\implies \exists p. \text{vwalk-bet } G \ u \ (u \# p) \ v \wedge \text{length } p = \text{distance } G \ u \ v \wedge \text{set } p \cap U = \{\}$$

proof(*goal-cases*)

case *main*: 1

then obtain p where *vwalk-bet* G u (u#p) v *length* p = *distance* G u v

by (*metis dist-reachable enat-ord-simps(4) hd-of-vwalk-bet length-tl list.sel(3) reachable-dist-2*)

moreover have *set* p ∩ U = {}

proof(*rule ccontr, goal-cases*)

case 1

then obtain p1 w p2 where p = p1 @ w # p2 w ∈ U

by (*auto dest!*: *split-list*)

hence *length* (w#p2) < *length* (u#p)

by *auto*

moreover have *vwalk-bet* G w (w#p2) v

using $\langle p = p1 @ w \# p2 \rangle$ $\langle \text{vwalk-bet } G \ u \ (u \# p) \ v \rangle$
split-vwalk vwalk-bet-cons

by *fastforce*

hence *distance* G w v ≤ *length* p2

by(*auto dest!*: *vwalk-bet-dist*)

ultimately have *distance-set* G U v ≤ *length* p2

using $\langle w \in U \rangle$

by(*auto dest!*: *dist-set-mem dest: order.trans*)

moreover have *length* p ≤ *distance-set* G U v

by (*simp add*: $\langle \text{enat } (\text{length } p) = \text{distance } G \ u \ v \rangle$ *main(3)*)

moreover have *enat* (*length* p2) < *eSuc* (*enat* (*length* p1 + *length* p2))

by *auto*

ultimately have *False*

using *leD*

```

    apply –
    apply (drule dual-order.trans[where c = enat (length p)])
    by (fastforce simp: ‹p = p1 @ w # p2› dest: )+
  then show ?case
    by auto
qed
ultimately show ?thesis
  by auto
qed

```

```

lemma dist-not-inf''':
  [[distance-set G U v ≠ ∞; u ∈ U; distance G u v = distance-set G U v]]
  ⇒ ∃ p. shortest-path G u (u # p) v ∧ set p ∩ U = {}
  apply (simp add: shortest-path-def)
  by (metis dist-not-inf'' enat.distinct(1))

```

```

lemma distance-set-union:
  distance-set G (U ∪ V) v ≤ distance-set G U v
  by (simp add: distance-set-def INF-superset-mono)

```

```

lemma lt-lt-infnty: x < (y::enat) ⇒ x < ∞
  using enat-ord-code(3) order-less-le-trans
  by blast

```

```

lemma finite-dist-nempty:
  distance-set G V v ≠ ∞ ⇒ V ≠ {}
  by (auto simp: distance-set-def top-enat-def)

```

```

lemma distance-set-wit:
  assumes v ∈ V
  obtains v' where v' ∈ V distance-set G V x = distance G v' x
  using assms wellorder-InfI[of distance G v x (%v. distance G v x) ' V]
  by (auto simp: distance-set-def image-def dest!: )

```

```

lemma distance-set-wit':
  assumes distance-set G V v ≠ ∞
  obtains v' where v' ∈ V distance-set G V x = distance G v' x
  using finite-dist-nempty[OF assms]
  by (auto elim: distance-set-wit)

```

```

lemma dist-set-not-inf: distance-set G U v ≠ ∞ ⇒ ∃ u ∈ U. distance G u v =
  distance-set G U v
  using distance-set-wit'
  by metis

```

```

lemma dist-not-inf': distance-set G U v ≠ ∞ ⇒
  ∃ u ∈ U. distance G u v = distance-set G U v ∧ reachable G u v

```

by (metis dist-reachable dist-set-not-inf enat-ord-simps(4))

lemma distance-on-vwalk:

$\llbracket \text{distance-set } G \ U \ v = \text{distance } G \ u \ v; u \in U; \text{shortest-path } G \ u \ p \ v; w \in \text{set } p \rrbracket$
 $\implies \text{distance-set } G \ U \ w = \text{distance } G \ u \ w$

proof(goal-cases)

case *assms*: 1

hence distance-set $G \ U \ w \leq \text{distance } G \ u \ w$

by (auto intro: dist-set-mem)

moreover have distance $G \ u \ w \leq \text{distance-set } G \ U \ w$

proof(rule ccontr, goal-cases)

case *dist-gt*: 1

hence distance-set $G \ U \ w \neq \infty$

using lt-lt-infnty

by (auto simp: linorder-class.not-le)

then obtain u' where $u' \in U$ distance $G \ u' \ w < \text{distance } G \ u \ w$

using *dist-gt*

by (fastforce dest!: dist-set-not-inf)

moreover then obtain p' where shortest-path $G \ u' \ p' \ w$

by (fastforce dest: lt-lt-infnty elim!: shortest-path-exists-2)

moreover obtain $p1 \ p2$ where $p = p1 \ @ \ [w] \ @ \ p2$

using $\langle w \in \text{set } p \rangle$

by (fastforce dest: iffD1[OF in-set-conv-decomp-first])

ultimately have vwalk-bet $G \ u' \ (p' \ @ \ tl \ ([w] \ @ \ p2)) \ v$

using $\langle \text{shortest-path } G \ u \ p \ v \rangle$

apply –

apply (rule vwalk-bet-transitive)

by (auto dest!: shortest-path-vwalk shortest-path-split-1)+

moreover have shortest-path $G \ u \ (p1 \ @ \ [w]) \ w$

using $\langle \text{shortest-path } G \ u \ p \ v \rangle$

by (auto dest!: shortest-path-split-2 simp: $\langle p = p1 \ @ \ [w] \ @ \ p2 \rangle$)

hence length $(p' \ @ \ tl \ ([w] \ @ \ p2)) - 1 < \text{length } p - 1$

using $\langle \text{shortest-path } G \ u' \ p' \ w \rangle \ \langle \text{distance } G \ u' \ w < \text{distance } G \ u \ w \rangle$

by (auto dest!: shortest-path-dist simp: $\langle p = p1 \ @ \ [w] \ @ \ p2 \rangle$)

hence length $(p' \ @ \ tl \ ([w] \ @ \ p2)) - 1 < \text{distance-set } G \ U \ v$

using *assms*(1,3) shortest-path-dist

by force

ultimately show *False*

using dist-set-mem [OF $\langle u' \in U \rangle$]

apply –

apply (drule vwalk-bet-dist)

by (meson leD order-le-less-trans)

qed

ultimately show *?thesis*

by auto

qed

lemma diff-le-self-enat: $m - n \leq (m::\text{enat})$

by (metis diff-le-self enat.exhaust enat-ord-code(1) idiff-enat-enat idiff-infinity)

idiff-infinity-right order-refl zero-le)

lemma *shortest-path-dist-set-union*:

$\llbracket \text{distance-set } G \ U \ v = \text{distance } G \ u \ v; u \in U; \text{shortest-path } G \ u \ (p1 \ @ \ x \ \# \ p2) \ v;$

$x \in V; \bigwedge v'. v' \in V \implies \text{distance-set } G \ U \ v' = \text{distance } G \ u \ x \rrbracket$

$\implies \text{distance-set } G \ (U \cup V) \ v = \text{distance-set } G \ U \ v - \text{distance } G \ u \ x$

proof(*goal-cases*)

case *assms: 1*

define *k* **where** $k = \text{distance } G \ u \ x$

have $\text{distance-set } G \ (U \cup V) \ v \leq \text{distance-set } G \ U \ v - k$

proof–

have $\text{vwalk-bet } G \ x \ (x \ \# \ p2) \ v$

using $\langle \text{shortest-path } G \ u \ (p1 \ @ \ x \ \# \ p2) \ v \rangle$

by(*auto dest: shortest-path-vwalk split-vwalk*)

moreover have $\langle x \in U \cup V \rangle$

using $\langle x \in V \rangle$

by *auto*

ultimately have $\text{distance-set } G \ (U \cup V) \ v \leq \text{length } (x \ \# \ p2) - 1$

by(*auto simp only: dest!: vwalk-bet-dist dist-set-mem dest: order-trans*)

moreover have $\text{length } p1 = k$

proof–

have $\text{shortest-path } G \ u \ (p1 \ @ \ [x]) \ x$

using $\langle \text{shortest-path } G \ u \ (p1 \ @ \ x \ \# \ p2) \ v \rangle$

by(*auto intro: shortest-path-split-2*)

moreover have $\text{distance-set } G \ U \ x = k$

using *assms*

by (*auto simp: k-def*)

ultimately have $\text{length } (p1 \ @ \ [x]) - 1 = k$

using *assms(1,2,3) distance-on-vwalk shortest-path-dist*

by *fastforce*

thus *?thesis*

by *auto*

qed

hence $(\text{distance-set } G \ U \ v) - k = \text{length } (x \ \# \ p2) - 1$

using $\langle \text{shortest-path } G \ u \ (p1 \ @ \ x \ \# \ p2) \ v \rangle$

by(*auto dest!: shortest-path-dist simp: <distance-set G U v = distance G u v>*)

ultimately show *?thesis*

by *auto*

qed

moreover have $\text{distance-set } G \ (U \cup V) \ v \geq \text{distance-set } G \ U \ v - k$

proof(*rule ccontr, goal-cases*)

case *dist-lt: 1*

hence $\text{distance-set } G \ (U \cup V) \ v \neq \infty$

using *lt-lt-infnty*

by (*auto simp: linorder-class.not-le*)

then obtain *u'* **where** $u' \in U \cup V \ \text{distance } G \ u' \ v < \text{distance-set } G \ U \ v - k$

using *dist-lt*

by (*fastforce dest!: dist-set-not-inf*)

```

then consider  $u' \in U \mid u' \in V$ 
  by auto
then show ?case
proof(cases)
  case 1
  moreover from  $\langle \text{distance } G \ u' \ v < \text{distance-set } G \ U \ v - k \rangle$ 
  have  $\langle \text{distance } G \ u' \ v < \text{distance-set } G \ U \ v \rangle$ 
    using diff-le-self-enat order-less-le-trans
    by blast
  ultimately show ?thesis
    by(auto simp: dist-set-mem leD)
next
  case 2
  moreover from  $\langle \text{distance } G \ u' \ v < \text{distance-set } G \ U \ v - k \rangle$ 
  obtain p2 where shortest-path  $G \ u' \ p2 \ v$ 
    by(auto elim!: shortest-path-exists-2 dest: lt-lt-infnty)
  have  $\text{distance-set } G \ U \ u' = k$ 
    using assms 2
    by (auto simp: k-def)
  moreover have  $\langle k \neq \infty \rangle$ 
    using assms
    by (fastforce simp: k-def dest: shortest-path-split-2 shortest-path-dist)
  ultimately obtain u where  $u \in U \ \text{distance } G \ u \ u' = k$ 
    by(fastforce dest!: dist-set-not-inf)
  moreover have  $\text{distance } G \ u \ v \leq \text{distance } G \ u \ u' + \text{distance } G \ u' \ v$ 
    using  $\langle k \neq \infty \rangle \ \text{lt-lt-infnty}[OF \ \langle \text{distance } G \ u' \ v < \text{distance-set } G \ U \ v - k \rangle]$ 
     $\langle \text{distance } G \ u \ u' = k \rangle$ 
    by(auto intro!: triangle-ineq simp: dist-reachable)
  ultimately have  $\text{distance } G \ u \ v \leq k + \text{distance } G \ u' \ v$ 
    by auto
  hence  $\text{distance } G \ u \ v < k + (\text{distance-set } G \ U \ v - k)$ 
    using  $\langle \text{distance } G \ u' \ v < \text{distance-set } G \ U \ v - k \rangle$ 
    by (meson  $\langle k \neq \infty \rangle \ \text{dual-order.strict-trans2 enat-add-left-cancel-less}$ )
  moreover have  $k \leq \text{distance-set } G \ U \ v$ 
    using assms(1,3) shortest-path-split-distance
    by (fastforce simp: k-def)
  hence  $k + (\text{distance-set } G \ U \ v - k) \leq \text{distance-set } G \ U \ v$ 
    by (simp add:  $\langle k \neq \infty \rangle \ \text{add-diff-assoc-enat}$ )
  ultimately have  $\text{distance } G \ u \ v < \text{distance-set } G \ U \ v$ 
    by auto
  then show ?thesis
    using  $\langle u \in U \rangle$ 
    by (simp add: dist-set-mem leD)
qed
qed
ultimately show ?case
  by (auto simp: k-def)
qed

```

lemma *Inf-enat-def1*:
fixes $K::\text{enat set}$
assumes $K \neq \{\}$
shows $\text{Inf } K \in K$
using *assms*
by (*auto simp add: Min-def Inf-enat-def*) (*meson LeastI*)

lemma *INF-plus-enat*:
 $V \neq \{\} \implies (\text{INF } v \in V. (f::'a \Rightarrow \text{enat}) v) + (x::\text{enat}) = (\text{INF } v \in V. f v + x)$
proof(*goal-cases*)
case *assms: 1*
have $(\text{INF } v \in V. (f::'a \Rightarrow \text{enat}) v) + (x::\text{enat}) \leq f\cdot v$ **if** $f\cdot v \in \{f v + x \mid v. v \in V\}$
for $f\cdot v$
using *that*
apply(*auto simp: image-def*)
by (*metis (mono-tags, lifting) Inf-lower mem-Collect-eq*)
moreover **have** $(\text{INF } v \in V. (f::'a \Rightarrow \text{enat}) v) + (x::\text{enat}) \in \{f v + x \mid v. v \in V\}$
proof–
have $\text{Inf } \{f v \mid v. v \in V\} \in \{f v \mid v. v \in V\}$
apply (*rule Inf-enat-def1*)
using *assms*
by *simp*

then obtain v **where** $v \in V$ $\text{Inf } \{f v \mid v. v \in V\} = f v$
using *assms*
by *auto*
hence $f v + 1 \in \{f v + 1 \mid v. v \in V\}$
by *auto*
hence $\text{Inf } \{f v \mid v. v \in V\} + x \in \{f v + x \mid v. v \in V\}$
apply(*subst <Inf {f v | v. v ∈ V} = f v>*)
by *auto*
thus *?thesis*
by (*simp add: Setcompr-eq-image*)
qed
ultimately show *?case*
by (*simp add: Inf-lower Setcompr-eq-image order-antisym wellorder-InfI*)
qed

lemma *distance-set-neighbourhood*:
 $\llbracket v \in \text{neighbourhood } G u; Vs \neq \{\} \rrbracket \implies \text{distance-set } G Vs v \leq \text{distance-set } G Vs u + 1$
proof(*goal-cases*)
case *assms: 1*
hence $(\text{INF } w \in Vs. \text{distance } G w v) \leq (\text{INF } w \in Vs. \text{distance } G w u + 1)$
by (*auto simp add: distance-neighbourhood' intro!: INF-mono'*)
also **have** $\dots = (\text{INF } w \in Vs. \text{distance } G w u) + (1::\text{enat})$
using $\langle Vs \neq \{\} \rangle$
by (*auto simp: INF-plus-enat*)
finally show *?case*

by(*simp only: distance-set-def*)
qed

lemma *distance-set-parent*:

$\llbracket \text{distance-set } G \text{ } V_s \text{ } v < \infty; V_s \neq \{\}; v \notin V_s \rrbracket \implies$

$\exists w. \text{distance-set } G \text{ } V_s \text{ } w + 1 = \text{distance-set } G \text{ } V_s \text{ } v \wedge v \in \text{neighbourhood } G \text{ } w$

proof(*goal-cases*)

case 1

moreover hence *distance-set* $G \text{ } V_s \text{ } v \neq \infty$

by *auto*

ultimately obtain u where $\langle u \in V_s \rangle \langle \text{distance-set } G \text{ } V_s \text{ } v = \text{distance } G \text{ } u \text{ } v \rangle$
 $\langle u \neq v \rangle$

by(*fastforce elim!: distance-set-wit'*)

then obtain w where $\text{distance } G \text{ } u \text{ } w + 1 = \text{distance } G \text{ } u \text{ } v \wedge v \in \text{neighbourhood } G \text{ } w$

using 1 *distance-parent*

by *fastforce*

thus *?thesis*

by (*metis* 1(2) $\langle \text{distance-set } G \text{ } V_s \text{ } v = \text{distance } G \text{ } u \text{ } v \rangle \langle u \in V_s \rangle$
add-mono-thms-linordered-semiring(3) *dist-set-mem*
distance-set-neighbourhood nle-le)

qed

lemma *distance-set-parent'*:

$\llbracket 0 < \text{distance-set } G \text{ } V_s \text{ } v; \text{distance-set } G \text{ } V_s \text{ } v < \infty; V_s \neq \{\} \rrbracket \implies$

$\exists w. \text{distance-set } G \text{ } V_s \text{ } w + 1 = \text{distance-set } G \text{ } V_s \text{ } v \wedge v \in \text{neighbourhood } G \text{ } w$

using *distance-set-parent*

by (*metis antisym-conv2 dist-set-inf distance-0 distance-set-def less-INF-D order.strict-implies-order*)

lemma *distance-set-0[simp]*: $\llbracket v \in dV_s \text{ } G \rrbracket \implies \text{distance-set } G \text{ } V_s \text{ } v = 0 \longleftrightarrow v \in V_s$

proof(*safe, goal-cases*)

case 2

moreover have $\text{distance } G \text{ } v \text{ } v = 0$

by (*meson calculation*(1) *distance-0*)

ultimately show *?case*

by (*metis dist-set-mem le-zero-eq*)

next

case 1

thus *?case*

by (*metis dist-set-not-inf distance-0 infinity-ne-i0*)

qed

lemma *dist-set-leq*:

$\llbracket \bigwedge u. u \in V_s \implies \text{distance } G \text{ } u \text{ } v \leq \text{distance } G' \text{ } u \text{ } v \rrbracket \implies \text{distance-set } G \text{ } V_s \text{ } v \leq \text{distance-set } G' \text{ } V_s \text{ } v$

by(*auto simp: distance-set-def INF-superset-mono*)

```

lemma dist-set-eq:
   $\llbracket \bigwedge u. u \in Vs \implies \text{distance } G \ u \ v = \text{distance } G' \ u \ v \rrbracket \implies \text{distance-set } G \ Vs \ v =$ 
 $\text{distance-set } G' \ Vs \ v$ 
  using dist-set-leq
  by (metis nle-le)

lemma distance-set-subset:  $G \subseteq G' \implies \text{distance-set } G' \ Vs \ v \leq \text{distance-set } G \ Vs$ 
 $v$ 
  by (simp add: dist-set-leq distance-subset)

lemma vwalk-bet-dist-set:
   $\llbracket V\text{walk.vwalk-bet } G \ u \ p \ v; u \in U \rrbracket \implies \text{distance-set } G \ U \ v \leq \text{length } p - 1$ 
  apply (auto simp: distance-set-def image-def intro!)
  by (metis (mono-tags, lifting) Inf-lower One-nat-def dual-order.trans mem-Collect-eq
vwalk-bet-dist)

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] in-
var-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  $\cap_G$  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 ( $[-]_s$ ) +
fixes sel :: 's  $\Rightarrow$  'a

assumes choose [simp]:  $s \neq \text{empty} \implies \text{isin } s \text{ (sel } s)$ 

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]:
   $s \neq \text{empty} \implies \text{invar } s \implies \text{sel } s \in \text{t-set } s$ 
  by(auto simp flip: set.set-isin)

lemma choose''[intro]:
   $\text{invar } s \implies s \neq \text{empty} \implies \text{t-set } s \subseteq s' \implies \text{sel } s \in s'$ 
  by(auto simp flip: set.set-isin)

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

```

```

      empty = s  $\implies$  t-set s = {}
      empty  $\neq$  s  $\implies$  invar s  $\implies$  t-set s  $\neq$  {}
using set.set-empty
by auto
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  $\Rightarrow$  'vset  $\Rightarrow$  'adjmap  $\Rightarrow$  'adjmap and adjmap-inv :: 'adjmap  $\Rightarrow$  bool
and

```

```

  vset-empty :: 'vset and vset-delete :: 'v  $\Rightarrow$  'vset  $\Rightarrow$  'vset and
  vset-inv

```

```

begin

```

```

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

```

```

abbreviation isin' (infixl  $\in_G$  50) where isin' G v  $\equiv$  isin v G
abbreviation not-isin' (infixl  $\notin_G$  50) where not-isin' G v  $\equiv$   $\neg$  isin' G v

```

```

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

```

```

definition graph-inv G = (adjmap-inv G  $\wedge$  ( $\forall$  v vset. lookup G v = Some vset  $\longrightarrow$ 
vset-inv vset))

```

```

definition finite-graph G = (finite {v. (lookup G v)  $\neq$  None})

```

```

definition finite-vsets = ( $\forall$  vset. finite (t-set vset))

```

```

definition neighb::'adjmap  $\Rightarrow$  'v  $\Rightarrow$  'vset where
  (neighb G v) = (case (lookup G v) of Some vset  $\Rightarrow$  vset | -  $\Rightarrow$  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]*:

$graph-inv G \implies (\llbracket adjmap-inv G; (\bigwedge v vset. lookup G v = Some vset \implies vset-inv vset) \rrbracket \implies P) \implies P$

by (*auto simp: graph-inv-def*)

lemma *graph-invI[intro]*:

$\llbracket adjmap-inv G; (\bigwedge v vset. lookup G v = Some vset \implies vset-inv vset) \rrbracket \implies graph-inv G$

by (*auto simp: graph-inv-def*)

lemma *finite-graphE*[*elim*]:
 $finite\text{-}graph\ G \implies (finite\ \{v.\ (lookup\ G\ v) \neq None\} \implies P) \implies P$
by (*auto simp: finite-graph-def*)

lemma *finite-graphI*[*intro*]:
 $finite\ \{v.\ (lookup\ G\ v) \neq None\} \implies finite\text{-}graph\ G$
by (*auto simp: finite-graph-def*)

lemma *finite-vssetsE*[*elim*]:
 $finite\text{-}vssets \implies ((\bigwedge N.\ finite\ (t\text{-}set\ N)) \implies P) \implies P$
by (*auto simp: finite-vssets-def*)

lemma *finite-vssetsI*[*intro*]:
 $(\bigwedge N.\ finite\ (t\text{-}set\ N)) \implies finite\text{-}vssets$
by (*auto simp: finite-vssets-def*)

lemma *neighbourhood-invars'*[*simp,dest*]:
 $graph\text{-}inv\ G \implies vset\text{-}inv\ (\mathcal{N}_G\ G\ v)$
by (*auto simp add: graph-inv-def neighb-def split: option.splits*)

lemma *finite-graph*[*intro!*]:
assumes $graph\text{-}inv\ G\ finite\text{-}graph\ G\ finite\text{-}vssets$
shows $finite\ (digraph\text{-}abs\ G)$
proof –

have $digraph\text{-}abs\ G \subseteq \{v.\ lookup\ G\ v \neq None\} \times (\bigcup (t\text{-}set\ ' \{N \mid N\ v.\ lookup\ G\ v = Some\ N\}))$
using *assms*
by (*fastforce simp: digraph-abs-def neighb-def graph-inv-def split: option.splits*)

moreover have $finite\ \{v.\ lookup\ G\ v \neq None\}$
using *assms*
by *fastforce*

moreover have $finite\ (\bigcup (t\text{-}set\ ' \{N \mid N\ v.\ lookup\ G\ v = Some\ N\}))$
using *assms*
by (*force elim!: finite-vssetsE finite-graphE*
intro!: finite-imageI
rev-finite-subset

[**where** $B = (the\ o\ lookup\ G)\ ' \{v.\ \exists y.\ lookup\ G\ v = Some\ y\}$])
ultimately show *?thesis*
by(*fastforce intro!: finite-cartesian-product dest: finite-subset*)+

qed

corollary *finite-vertices*[*intro!*]:
assumes $graph\text{-}inv\ G\ finite\text{-}graph\ G\ finite\text{-}vssets$

shows *finite* (*dVs* (*digraph-abs* *G*))
using *finite-graph*[*OF assms*]
by (*simp add: finite-vertices-iff*)

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*]:
 $graph\text{-}inv\ G \implies v \in t\text{-}set\ (\mathcal{N}_G\ G\ u) \longleftrightarrow (u,v) \in digraph\text{-}abs\ G$
by(*auto simp: digraph-abs-def neighbourhood-def option.discI graph-inv-def split: option.split*)

lemma *are-connected-absD*[*dest*]:
 $\llbracket v \in t\text{-}set\ (\mathcal{N}_G\ G\ u); graph\text{-}inv\ G \rrbracket \implies (u,v) \in digraph\text{-}abs\ G$
by (*auto simp: are-connected-abs*)

lemma *are-connected-absI*[*intro*]:
 $\llbracket (u,v) \in digraph\text{-}abs\ G; graph\text{-}inv\ G \rrbracket \implies v \in t\text{-}set\ (\mathcal{N}_G\ G\ u)$
by (*auto simp: are-connected-abs*)

lemma *neighbourhood-absD*[*dest!*]:
 $\llbracket t\text{-}set\ (\mathcal{N}_G\ G\ x) \neq \{\}; graph\text{-}inv\ G \rrbracket \implies x \in dVs\ (digraph\text{-}abs\ G)$
by *blast*

lemma *neighbourhood-abs*[*simp*]:
 $graph\text{-}inv\ G \implies t\text{-}set\ (\mathcal{N}_G\ G\ u) = neighbourhood\ (digraph\text{-}abs\ G)\ u$
by(*auto simp: digraph-abs-def neighb-def Pair-Graph.neighbourhood-def option.discI graph-inv-def split: option.split*)

lemma *adjmap-inv-insert*[*intro*]: $graph\text{-}inv\ G \implies graph\text{-}inv\ (add\text{-}edge\ G\ u\ v)$
by (*auto simp: add-edge-def graph-inv-def split: option.splits*)

lemma *digraph-abs-insert*[*simp*]: $graph\text{-}inv\ G \implies digraph\text{-}abs\ (add\text{-}edge\ G\ u\ v) = Set.insert\ (u,v)\ (digraph\text{-}abs\ G)$
by (*fastforce simp add: digraph-abs-def set-of-map-def neighb-def add-edge-def split: option.splits if-splits*)

lemma *adjmap-inv-delete*[*intro*]: $graph\text{-}inv\ G \implies graph\text{-}inv\ (delete\text{-}edge\ G\ u\ v)$
by (*auto simp: delete-edge-def graph-inv-def split: option.splits*)

lemma *digraph-abs-delete*[*simp*]: $graph\text{-}inv\ G \implies digraph\text{-}abs\ (delete\text{-}edge\ G\ u\ v) = (digraph\text{-}abs\ G) - \{(u,v)\}$
by (*fastforce simp add: digraph-abs-def set-of-map-def neighb-def delete-edge-def split: option.splits if-splits*)

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-elim*s

named-theorems *call-cond-intros*

named-theorems *ret-holds-intros*

named-theorems *invar-props-intros*

named-theorems *invar-props-elim*s

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*) $DFS::('v, 'vset) DFS\text{-}state \Rightarrow ('v, 'vset) DFS\text{-}state$ **where**
 $DFS\ dfs\text{-}state =$
 (case (stack dfs-state) of (v # stack-tl) \Rightarrow
 (if v = t then
 (dfs-state (return := Reachable))
 else ((if ($\mathcal{N}_G\ v -_G\ (seen\ dfs\text{-}state)$) $\neq\ \emptyset_V$ then
 let u = (sel (($\mathcal{N}_G\ v$) $-_G\ (seen\ dfs\text{-}state)$));
 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'))
)
)
 | - \Rightarrow (dfs-state (return := NotReachable))
)
by *pat-completeness auto*

6.5 Setup for Reasoning About the Algorithm

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

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

lemma $DFS\text{-}call\text{-}1\text{-}conds[\text{call-cond-elim}]$:

$DFS\text{-}call\text{-}1\text{-}conds\ dfs\text{-}state \Longrightarrow$
 $\llbracket \exists v\ stack\text{-}tl.\ stack\ dfs\text{-}state = v\ \#\ stack\text{-}tl;$
 $hd\ (stack\ dfs\text{-}state) \neq t;$
 $(\mathcal{N}_G\ (hd\ (stack\ dfs\text{-}state))) -_G\ (seen\ dfs\text{-}state) \neq (\emptyset_V) \rrbracket \Longrightarrow P \Longrightarrow$
 P

by (*auto simp: DFS-call-1-conds-def split: list.splits option.splits if-splits*)

definition *DFS-upd1* *dfs-state* = (
 let
 $N = (\mathcal{N}_G (\text{hd } (\text{stack } \text{dfs-state})));$
 $u = (\text{sel } ((N -_G (\text{seen } \text{dfs-state})));)$
 $\text{stack}' = u \# (\text{stack } \text{dfs-state});$
 $\text{seen}' = \text{insert } u (\text{seen } \text{dfs-state})$
 in
 $\text{dfs-state } (\text{stack} := \text{stack}', \text{seen} := \text{seen}')$)

definition *DFS-call-2-conds*::('v, 'vset) *DFS-state* \Rightarrow bool **where**
DFS-call-2-conds *dfs-state* =
 (case *stack* *dfs-state* of (v # *stack-tl*) \Rightarrow
 (if v = t then
 False
 else (
 (if ((\mathcal{N}_G v) $-_G$ (seen *dfs-state*)) \neq (\emptyset_V) then
 False
 else True)
)
 | - \Rightarrow False
)

lemma *DFS-call-2-condsE*[*call-cond-elim*]:
DFS-call-2-conds *dfs-state* \Longrightarrow
 $\llbracket \exists v \text{ stack-tl. stack } \text{dfs-state} = v \# \text{stack-tl};$
 $\text{hd } (\text{stack } \text{dfs-state}) \neq t;$
 $(\mathcal{N}_G (\text{hd } (\text{stack } \text{dfs-state}))) -_G (\text{seen } \text{dfs-state}) = (\emptyset_V) \rrbracket \Longrightarrow P \Longrightarrow$
P
by(*auto simp: DFS-call-2-conds-def split: list.splits option.splits if-splits*)

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

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

lemma *DFS-ret-1-conds*[*call-cond-elim*s]:
 $DFS-ret-1-conds\ dfs-state \implies$
 $\llbracket stack\ dfs-state = [] \rrbracket \implies P \implies$
 P
by(*auto simp: DFS-ret-1-conds-def split: list.splits option.splits if-splits*)

lemma *DFS-call-4-condsI*[*call-cond-intros*]:
 $\llbracket stack\ dfs-state = [] \rrbracket \implies DFS-ret-1-conds\ dfs-state$
by(*auto simp: DFS-ret-1-conds-def split: list.splits option.splits if-splits*)

definition *DFS-ret1* $dfs-state = (dfs-state (\return := NotReachable))$

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

lemma *DFS-ret-2-conds*[*call-cond-elim*s]:
 $DFS-ret-2-conds\ dfs-state \implies$
 $\llbracket \bigwedge v\ stack-tl.\ \llbracket stack\ dfs-state = v\ \# \ stack-tl;$
 $\quad (hd\ (stack\ dfs-state)) = t \rrbracket \implies P \rrbracket \implies$
 P
by(*auto simp: DFS-ret-2-conds-def split: list.splits option.splits if-splits*)

lemma *DFS-ret-2-condsI*[*call-cond-intros*]:
 $\bigwedge v\ stack-tl.\ \llbracket stack\ dfs-state = v\ \# \ stack-tl;$
 $(hd\ (stack\ dfs-state)) = t \rrbracket \implies$
 $DFS-ret-2-conds\ dfs-state$
by(*auto simp: DFS-ret-2-conds-def split: list.splits option.splits if-splits*)

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

lemma *DFS-cases*:
assumes *DFS-call-1-conds* $dfs-state \implies P$
 $DFS-call-2-conds\ dfs-state \implies P$
 $DFS-ret-1-conds\ dfs-state \implies P$
 $DFS-ret-2-conds\ dfs-state \implies P$

shows P

proof –

have $DFS-call-1-conds\ dfs-state \vee DFS-call-2-conds\ dfs-state \vee$

```

      DFS-ret-1-conds dfs-state  $\vee$  DFS-ret-2-conds dfs-state
    by (auto simp add: DFS-call-1-conds-def DFS-call-2-conds-def
          DFS-ret-1-conds-def DFS-ret-2-conds-def
          split: list.split-asm option.split-asm if-splits)
  then show ?thesis
    using assms
    by auto
qed

lemma DFS-simps:
  assumes DFS-dom dfs-state
  shows DFS-call-1-conds dfs-state  $\implies$  DFS dfs-state = DFS (DFS-upd1 dfs-state)
        DFS-call-2-conds dfs-state  $\implies$  DFS dfs-state = DFS (DFS-upd2 dfs-state)
        DFS-ret-1-conds dfs-state  $\implies$  DFS dfs-state = DFS-ret1 dfs-state
        DFS-ret-2-conds dfs-state  $\implies$  DFS dfs-state = DFS-ret2 dfs-state
  by (auto simp add: DFS.psimps[OF assms] Let-def
        DFS-call-1-conds-def DFS-upd1-def DFS-call-2-conds-def
        DFS-upd2-def
        DFS-ret-1-conds-def DFS-ret1-def
        DFS-ret-2-conds-def DFS-ret2-def

    split: list.splits option.splits if-splits )

```

```

lemma DFS-induct:
  assumes DFS-dom dfs-state
  assumes  $\bigwedge$ dfs-state.  $\llbracket$ DFS-dom dfs-state;
        DFS-call-1-conds dfs-state  $\implies$  P (DFS-upd1 dfs-state);
        DFS-call-2-conds dfs-state  $\implies$  P (DFS-upd2 dfs-state) $\rrbracket \implies$  P
  dfs-state
  shows P dfs-state
  apply(rule DFS.pinduct[OF assms(1)])
  apply(rule assms(2)[simplified DFS-call-1-conds-def DFS-upd1-def DFS-call-2-conds-def
    DFS-upd2-def])
  by (auto simp: Let-def split: list.splits option.splits if-splits)

```

```

lemma DFS-domintros:
  assumes DFS-call-1-conds dfs-state  $\implies$  DFS-dom (DFS-upd1 dfs-state)
  assumes DFS-call-2-conds dfs-state  $\implies$  DFS-dom (DFS-upd2 dfs-state)
  shows DFS-dom dfs-state
proof(rule DFS.domintros, goal-cases)
  case (1 x21 x22)
  then show ?case
    using assms(1)[simplified DFS-call-1-conds-def DFS-upd1-def]
    by (force simp: Let-def split: list.splits option.splits if-splits)
next
  case (2 x21 x22)
  then show ?case
    using assms(2)[simplified DFS-call-2-conds-def DFS-upd2-def]
    by (force split: list.splits option.splits if-splits)

```

qed

6.6 Loop Invariants

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

definition *invar-stack-walk*::('v,'vset) DFS-state \Rightarrow 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 \Rightarrow bool **where**
invar-seen-stack dfs-state \longleftrightarrow
distinct (stack dfs-state)
 \wedge set (stack dfs-state) \subseteq t-set (seen dfs-state)
 \wedge t-set (seen dfs-state) \subseteq dVs (Graph.digraph-abs G)

definition *invar-s-in-stack*::('v,'vset) DFS-state \Rightarrow bool **where**
invar-s-in-stack dfs-state \longleftrightarrow
(stack (dfs-state) \neq [] \longrightarrow last (stack dfs-state) = s)

definition *invar-visited-through-seen*::('v,'vset) DFS-state \Rightarrow bool **where**
invar-visited-through-seen dfs-state =
($\forall v \in$ t-set (seen dfs-state).
($\forall p. Vwalk.vwalk\text{-bet}$ (Graph.digraph-abs G) v p t \wedge distinct p \longrightarrow (set p \cap set (stack dfs-state) \neq {})))

definition *call-1-measure*::('v,'vset) DFS-state \Rightarrow 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 \Rightarrow nat **where**
call-2-measure dfs-state = card (set (stack dfs-state))

definition *DFS-term-rel*::('v,'vset) DFS-state \times ('v,'vset) DFS-state) set **where**
DFS-term-rel = (call-1-measure) $\langle *mlex* \rangle$ (call-2-measure) $\langle *mlex* \rangle$ {}

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
using *DFS-axioms*
by (*auto simp: DFS-axioms-def*)

lemma *s-in-G*[*simp,intro*]: $s \in dVs$ (*Graph.digraph-abs G*)
using *DFS-axioms*
by (*auto simp: DFS-axioms-def*)

lemma *finite-neighbourhoods*[*simp*]:
finite (t-set N)
using *graph-inv(3)*
by *fastforce*

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-elim*s]:
invar-well-formed dfs-state \implies
 $(\llbracket vset-inv \text{ (seen dfs-state)} \rrbracket \implies P) \implies$
 P
by (*auto simp: invar-well-formed-def*)

lemma *invar-well-formed-intro*[*invar-props-intros*]: $\llbracket vset-inv \text{ (seen dfs-state)} \rrbracket$
 \implies *invar-well-formed dfs-state*
by (*auto simp: invar-well-formed-def*)

lemma *invar-well-formed-holds-1*[*invar-holds-intros*]:
 $\llbracket DFS-call-1-conds \text{ dfs-state; invar-well-formed dfs-state} \rrbracket \implies$
invar-well-formed (DFS-upd1 dfs-state)
by (*auto simp: Let-def DFS-upd1-def elim!: invar-props-elim intro!: invar-props-intros*)

lemma *invar-well-formed-holds-2*[*invar-holds-intros*]: $\llbracket DFS-call-2-conds \text{ dfs-state; invar-well-formed dfs-state} \rrbracket \implies$ *invar-well-formed (DFS-upd2 dfs-state)*
by (*auto simp: DFS-upd2-def elim!: invar-props-elim intro: invar-props-intros*)

lemma *invar-well-formed-holds-4*[*invar-holds-intros*]: $\llbracket DFS-ret-1-conds \text{ dfs-state; invar-well-formed dfs-state} \rrbracket \implies$ *invar-well-formed (DFS-ret1 dfs-state)*
by (*auto elim!: invar-props-elim intro: invar-props-intros simp: DFS-ret1-def*)

lemma *invar-well-formed-holds-5*[*invar-holds-intros*]: $\llbracket DFS-ret-2-conds \text{ dfs-state; invar-well-formed dfs-state} \rrbracket \implies$ *invar-well-formed (DFS-ret2 dfs-state)*
by (*auto elim!: invar-props-elim intro: invar-props-intros simp: DFS-ret2-def*)

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)*

```

using assms(2)
proof(induction rule: DFS-induct[OF assms(1)])
  case IH: (1 dfs-state)
  show ?case
    apply(rule DFS-cases[where dfs-state = dfs-state])
    by (auto intro!: IH(2-4) invar-holds-intros simp: DFS-simps[OF IH(1)])
qed

```

```

lemma invar-stack-walk-props[invar-props-elims]:
  invar-stack-walk dfs-state  $\implies$ 
    ((Vwalk.vwalk (Graph.digraph-abs G) (rev (stack dfs-state))))  $\implies P \implies P$ 
  by (auto simp: invar-stack-walk-def)

```

```

lemma invar-stack-walk-intro[invar-props-intros]: Vwalk.vwalk (Graph.digraph-abs G) (rev (stack dfs-state))  $\implies$  invar-stack-walk dfs-state
  by (auto simp: invar-stack-walk-def)

```

```

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)
  using assms graph-inv
  by (force simp: Let-def DFS-upd1-def elim!: call-cond-elim elim!: invar-props-elim intro!: Vwalk.vwalk-append2 neighbourhoodI invar-props-intros)

```

```

lemma invar-stack-walk-holds-2[invar-holds-intros]: [DFS-call-2-conds dfs-state; invar-stack-walk dfs-state]  $\implies$  invar-stack-walk (DFS-upd2 dfs-state)
  by (auto simp: DFS-upd2-def dest!: append-vwalk-pref elim!: invar-props-elim intro!: invar-props-intros elim: call-cond-elims)

```

```

lemma invar-stack-walk-holds-4[invar-holds-intros]: [DFS-ret-1-conds dfs-state; invar-stack-walk dfs-state]  $\implies$  invar-stack-walk (DFS-ret1 dfs-state)
  by (auto elim!: invar-props-elim intro: invar-props-intros simp: DFS-ret1-def)

```

```

lemma invar-2-holds-5[invar-holds-intros]: [DFS-ret-2-conds dfs-state; invar-stack-walk dfs-state]  $\implies$  invar-stack-walk (DFS-ret2 dfs-state)
  by (auto elim!: invar-props-elim intro: invar-props-intros simp: DFS-ret2-def)

```

```

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)
  using assms(2-3)
proof(induction rule: DFS-induct[OF assms(1)])
  case IH: (1 dfs-state)
  show ?case
    apply(rule DFS-cases[where dfs-state = dfs-state])
    by (auto intro!: IH(2-5) invar-holds-intros simp: DFS-simps[OF IH(1)])

```

qed

lemma *invar-seen-stack-props*[*invar-props-elim*s]:

invar-seen-stack dfs-state \implies
($\llbracket \text{distinct } (\text{stack } \text{dfs-state}); \text{set } (\text{stack } \text{dfs-state}) \subseteq \text{t-set } (\text{seen } \text{dfs-state});$
 $\text{t-set } (\text{seen } \text{dfs-state}) \subseteq \text{dVs } (\text{Graph.digraph-abs } G) \rrbracket \implies P) \implies P$
by (*auto simp: invar-seen-stack-def*)

lemma *invar-seen-stack-intro*[*invar-props-intro*s]:

$\llbracket \text{distinct } (\text{stack } \text{dfs-state}); \text{set } (\text{stack } \text{dfs-state}) \subseteq \text{t-set } (\text{seen } \text{dfs-state}); \text{t-set } (\text{seen } \text{dfs-state}) \subseteq \text{dVs } (\text{Graph.digraph-abs } G) \rrbracket \implies \text{invar-seen-stack } \text{dfs-state}$
by (*auto simp: invar-seen-stack-def*)

lemma *invar-seen-stack-holds-1*[*invar-holds-intro*s]:

$\llbracket \text{DFS-call-1-conds } \text{dfs-state}; \text{invar-well-formed } \text{dfs-state}; \text{invar-seen-stack } \text{dfs-state} \rrbracket$
 $\implies \text{invar-seen-stack } (\text{DFS-upd1 } \text{dfs-state})$
by (*force simp: Let-def DFS-upd1-def dest!: append-vwalk-pref elim!: call-cond-elim*
elim!: invar-props-elim intro!: invar-props-intro)

lemma *invar-seen-stack-holds-2*[*invar-holds-intro*s]:

$\llbracket \text{DFS-call-2-conds } \text{dfs-state}; \text{invar-well-formed } \text{dfs-state}; \text{invar-seen-stack } \text{dfs-state} \rrbracket$
 \implies
invar-seen-stack (*DFS-upd2* *dfs-state*)
by (*auto elim!: call-cond-elim simp: DFS-upd2-def elim: vwalk-betE*
elim!: invar-props-elim dest!: Graph.vset.emptyD append-vwalk-pref intro!
invar-props-intro)

lemma *invar-seen-stack-holds-4*[*invar-holds-intro*s]:

$\llbracket \text{DFS-ret-1-conds } \text{dfs-state}; \text{invar-seen-stack } \text{dfs-state} \rrbracket \implies$
invar-seen-stack (*DFS-ret1* *dfs-state*)
by (*auto elim!: invar-props-elim intro!: invar-props-intro simp: DFS-ret1-def*)

lemma *invar-seen-stack-holds-5*[*invar-holds-intro*s]: $\llbracket \text{DFS-ret-2-conds } \text{dfs-state}; \text{invar-seen-stack } \text{dfs-state} \rrbracket \implies \text{invar-seen-stack } (\text{DFS-ret2 } \text{dfs-state})$

by (*auto elim!: invar-props-elim intro!: invar-props-intro simp: DFS-ret2-def*)

lemma *invar-seen-stack-holds*[*invar-holds-intro*s]:

assumes *DFS-dom* *dfs-state* *invar-well-formed* *dfs-state* *invar-seen-stack* *dfs-state*
shows *invar-seen-stack* (*DFS* *dfs-state*)
using *assms*(2-)

proof(*induction rule: DFS-induct[OF assms(1)]*)

case *IH*: (1 *dfs-state*)

show ?*case*

apply(*rule* *DFS-cases*[**where** *dfs-state* = *dfs-state*])

by (*auto intro!: IH(2-5) invar-holds-intro simp: DFS-simps[OF IH(1)]*)

qed

lemma *invar-s-in-stack-props*[*invar-props-elim*s]:

invar-s-in-stack dfs-state \implies
 $\llbracket (\text{stack } (dfs\text{-state}) \neq [] \implies \text{last } (\text{stack } dfs\text{-state}) = s) \rrbracket \implies P \implies P$
by (*auto simp: invar-s-in-stack-def*)

lemma *invar-s-in-stack-intro*[*invar-props-intros*]:
 $\llbracket (\text{stack } (dfs\text{-state}) \neq [] \implies \text{last } (\text{stack } dfs\text{-state}) = s) \rrbracket \implies \text{invar-s-in-stack } dfs\text{-state}$
by (*auto simp: invar-s-in-stack-def*)

lemma *invar-s-in-stack-holds-1*[*invar-holds-intros*]:
 $\llbracket DFS\text{-call-1-conds } dfs\text{-state}; \text{invar-well-formed } dfs\text{-state}; \text{invar-s-in-stack } dfs\text{-state} \rrbracket$
 $\implies \text{invar-s-in-stack } (DFS\text{-upd1 } dfs\text{-state})$
by (*force simp: Let-def DFS-upd1-def dest!: append-vwalk-pref elim!: call-cond-elims*
elim!: invar-props-elims intro!: invar-props-intros)

lemma *invar-s-in-stack-holds-2*[*invar-holds-intros*]:
 $\llbracket DFS\text{-call-2-conds } dfs\text{-state}; \text{invar-well-formed } dfs\text{-state}; \text{invar-s-in-stack } dfs\text{-state} \rrbracket$
 \implies
 $\text{invar-s-in-stack } (DFS\text{-upd2 } dfs\text{-state})$
by (*auto elim!: call-cond-elims simp: DFS-upd2-def elim: vwalk-betE*
elim!: invar-props-elims dest!: Graph.vset.emptyD append-vwalk-pref intro!
invar-props-intros)

lemma *invar-s-in-stack-holds-4*[*invar-holds-intros*]:
 $\llbracket DFS\text{-ret-1-conds } dfs\text{-state}; \text{invar-s-in-stack } dfs\text{-state} \rrbracket \implies$
 $\text{invar-s-in-stack } (DFS\text{-ret1 } dfs\text{-state})$
by (*auto elim!: invar-props-elims intro!: invar-props-intros simp: DFS-ret1-def*)

lemma *invar-s-in-stack-holds-5*[*invar-holds-intros*]: $\llbracket DFS\text{-ret-2-conds } dfs\text{-state}; \text{invar-s-in-stack } dfs\text{-state} \rrbracket \implies \text{invar-s-in-stack } (DFS\text{-ret2 } dfs\text{-state})$
by (*auto elim!: invar-props-elims intro!: invar-props-intros simp: DFS-ret2-def*)

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)*
using *assms(2-)*
proof (*induction rule: DFS-induct[OF assms(1)]*)
case *IH: (1 dfs-state)*
show *?case*
apply (*rule DFS-cases[where dfs-state = dfs-state]*)
by (*auto intro!: IH(2-5) invar-holds-intros simp: DFS-simps[OF IH(1)]*)
qed

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

by (auto simp: invar-visited-through-seen-def)

lemma *invar-visited-through-seen-intro*[*invar-props-intros*]:

[[$\bigwedge v p. \llbracket v \in t\text{-set } (\text{seen } \text{dfs-state});$
 $(V\text{walk.vwalk-bet } (\text{Graph.digraph-abs } G) v p t); \text{distinct } p \rrbracket \implies$
 $\text{set } p \cap \text{set } (\text{stack } \text{dfs-state}) \neq \{\}$]] \implies *invar-visited-through-seen dfs-state*
 by (auto simp: invar-visited-through-seen-def)

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]]
 \implies *invar-visited-through-seen (DFS-upd1 dfs-state)*
 by(*fastforce simp: Let-def DFS-upd1-def dest: append-vwalk-pref hd-of-vwalk-bet''*
elim!: invar-props-elim intro!: invar-props-intros)

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]] \implies *invar-visited-through-seen (DFS-upd2*
dfs-state)
proof(*rule invar-props-intros, elim invar-visited-through-seen-props call-cond-elim*
exE, goal-cases)
 case (1 *v1 p v2 stack-tl*)

We know one thing: a path starting at *v1* and ending at *t* intersects with the old stack *v2 # stack-tl*. We have two cases:

hence $\text{set } p \cap \text{set } (\text{stack } \text{dfs-state}) \neq \{\}$
by (auto simp: *DFS-upd2-def*)
then obtain *u where* $u \in \text{set } p \cap \text{set } (\text{stack } \text{dfs-state})$
by *auto*
show ?*case*
proof(*cases u \in set stack-tl*)
 case *True*

Case 1: If the point of intersection of the path is in *stack-tl*, then we are done.

thus ?*thesis*
using 1 $\langle u \in \text{set } p \cap \text{set } (\text{stack } \text{dfs-state}) \rangle$
by (auto simp: *DFS-upd2-def*)
next
 case *False*
hence $u = v2$
using 1 $\langle u \in \text{set } p \cap \text{set } (\text{stack } \text{dfs-state}) \rangle$
by *auto*

Case 2: If it intersects the old stack at *v2*, which is the more interesting case as *v2* will not be in the new stack.

then obtain $p1\ p2$ **where** $[simp]: p = p1 @ [v2] @ p2$
using $\langle u \in set\ p \cap set\ (stack\ dfs\ state) \rangle$
by $(auto\ simp: in-set-conv-decomp)$

Let $p = p1 @ [v2] @ p2$.

hence $set\ (v2 \# p2) \cap set\ (stack\ dfs\ state) \neq \{\}$
using 1
by $(auto\ simp: DFS-upd2-def)$

Since the invariant holds for the old state, then $[v2] @ p2$ intersects the old stack $v2 \# stack-tl$.

show *?thesis*
proof $(cases\ p2 = [])$
case *True*

There are two cases we need to consider here: Case a: $p2 = []$ This cannot be the case, since it would imply that $v2 = t$, which violates the assumption of us being in this execution branch.

thus *?thesis*
using 1
by $(auto\ simp: vwalk-bet-snoc)$
next
case *False*

Case b: $p2 \neq []$ From the current branch's assumptions, we know that $hd\ p2$, who is a neighbour of $v2$, is in *seen dfs-state*. This means that, from the invariant at *dfs-state*, we can conclude that $v2 \# p2$ intersects with the old stack. However, since it is distinct, that means that $p2$ cannot contain $v2$. This means that it intersects *stack-tl*, which implies that p with *stack-tl*. This finishes our proof.

hence $hd\ p2 \in t\text{-set}\ (\mathcal{N}_G\ v2)$
using $\langle vwalk\ bet\ (Graph.digraph-abs\ G)\ v1\ p\ t \rangle$
by $(auto\ dest!: split-vwalk\ simp: neq-Nil-conv)$
hence $hd\ p2 \in t\text{-set}\ (seen\ dfs\ state)$
using 1
by $(fastforce\ elim!: invar-props-elim\ simp\ del: \langle p = p1 @ [v2] @ p2 \rangle)$
hence $set\ p2 \cap set\ (stack\ dfs\ state) \neq \{\}$
using $1\ False$
by $(fastforce\ simp: DFS-upd2-def\ neq-Nil-conv\ dest!: split-vwalk)$
moreover have $v2 \notin set\ p2$
using $\langle distinct\ p \rangle$
by *auto*
ultimately have $set\ p2 \cap set\ (stack\ (DFS-upd2\ dfs\ state)) \neq \{\}$
using 1
by $(auto\ simp: DFS-upd2-def)$
thus *?thesis*
by *auto*
qed

qed
qed

lemma *invar-visited-through-seen-holds-4* [*invar-holds-intros*]: $\llbracket \text{DFS-ret-1-conds } \text{dfs-state}; \text{invar-visited-through-seen } \text{dfs-state} \rrbracket \implies \text{invar-visited-through-seen } (\text{DFS-ret1 } \text{dfs-state})$
by (*auto simp: intro: invar-props-intros simp: DFS-ret1-def*)

lemma *invar-visited-through-seen-holds-5* [*invar-holds-intros*]: $\llbracket \text{DFS-ret-2-conds } \text{dfs-state}; \text{invar-visited-through-seen } \text{dfs-state} \rrbracket \implies \text{invar-visited-through-seen } (\text{DFS-ret2 } \text{dfs-state})$
by (*auto simp: intro: invar-props-intros simp: DFS-ret2-def*)

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)*
using *assms(2-)*
proof (*induction rule: DFS-induct[OF assms(1)]*)
case *IH: (1 dfs-state)*
show *?case*
apply (*rule DFS-cases[where dfs-state = dfs-state]*)
by (*auto intro!: IH(2-6) invar-holds-intros simp: DFS-simps[OF IH(1)]*)
qed

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

lemma *state-rel-1-props[elim!]*: $\text{state-rel-1 } \text{dfs-state-1 } \text{dfs-state-2} \implies (t\text{-set } (\text{seen } \text{dfs-state-1}) \subseteq t\text{-set } (\text{seen } \text{dfs-state-2})) \implies P) \implies P$
by (*auto simp: state-rel-1-def*)

lemma *state-rel-1-intro* [*state-rel-intros*]:
 $\llbracket t\text{-set } (\text{seen } \text{dfs-state-1}) \subseteq t\text{-set } (\text{seen } \text{dfs-state-2}) \rrbracket \implies \text{state-rel-1 } \text{dfs-state-1 } \text{dfs-state-2}$
by (*auto simp: state-rel-1-def*)

lemma *state-rel-1-trans*:
 $\llbracket \text{state-rel-1 } \text{dfs-state-1 } \text{dfs-state-2}; \text{state-rel-1 } \text{dfs-state-2 } \text{dfs-state-3} \rrbracket \implies \text{state-rel-1 } \text{dfs-state-1 } \text{dfs-state-3}$
by (*auto intro!: state-rel-intros*)

lemma *state-rel-1-holds-1* [*state-rel-holds-intros*]:
 $\llbracket \text{DFS-call-1-conds } \text{dfs-state}; \text{invar-well-formed } \text{dfs-state} \rrbracket \implies \text{state-rel-1 } \text{dfs-state } (\text{DFS-upd1 } \text{dfs-state})$
by (*auto simp: Let-def DFS-upd1-def elim!: invar-props-elim intro!: state-rel-intros*)

lemma *state-rel-1-holds-2* [*state-rel-holds-intros*]:

$\llbracket \text{DFS-call-2-conds } dfs\text{-state}; \text{invar-well-formed } dfs\text{-state} \rrbracket \Longrightarrow \text{state-rel-1 } dfs\text{-state}$
 $(\text{DFS-upd2 } dfs\text{-state})$

by (*auto simp: DFS-upd2-def intro!: state-rel-intros elim: call-cond-elims*)

lemma *state-rel-1-holds-4*[*state-rel-holds-intros*]:

$\llbracket \text{DFS-ret-1-conds } dfs\text{-state} \rrbracket \Longrightarrow \text{state-rel-1 } dfs\text{-state } (\text{DFS-ret1 } dfs\text{-state})$

by (*auto simp: intro!: state-rel-intros simp: DFS-ret1-def*)

lemma *state-rel-1-holds-5*[*state-rel-holds-intros*]:

$\llbracket \text{DFS-ret-2-conds } dfs\text{-state} \rrbracket \Longrightarrow \text{state-rel-1 } dfs\text{-state } (\text{DFS-ret2 } dfs\text{-state})$

by (*auto simp: intro!: state-rel-intros simp: DFS-ret2-def*)

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)*

using *assms(2-)*

proof(*induction rule: DFS-induct[OF assms(1)]*)

case *IH: (1 dfs-state)*

show *?case*

apply(*rule DFS-cases[where dfs-state = dfs-state]*)

by (*auto intro: state-rel-1-trans invar-holds-intros state-rel-holds-intros intro!:*

IH(2-) simp: DFS-simps[OF IH(1)])

qed

lemma *DFS-ret-1[ret-holds-intros]: DFS-ret-1-conds (dfs-state) \Longrightarrow DFS-ret-1-conds (DFS-ret1 dfs-state)*

by (*auto simp: elim!: call-cond-elims intro!: call-cond-intros simp: DFS-ret1-def*)

lemma *ret1-holds[ret-holds-intros]:*

assumes *DFS-dom dfs-state return (DFS dfs-state) = NotReachable*

shows *DFS-ret-1-conds (DFS dfs-state)*

using *assms(2-)*

proof(*induction rule: DFS-induct[OF assms(1)]*)

case *IH: (1 dfs-state)*

show *?case*

apply(*rule DFS-cases[where dfs-state = dfs-state]*)

using *IH(4)*

by (*auto intro: ret-holds-intros intro!: IH(2-) simp: DFS-simps[OF IH(1)]*

DFS-ret2-def)

qed

lemma *DFS-correct-ret-1:*

$\llbracket \text{invar-visited-through-seen } dfs\text{-state}; \text{DFS-ret-1-conds } dfs\text{-state}; u \in t\text{-set (seen } dfs\text{-state)} \rrbracket$

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

by (*auto elim!: call-cond-elims invar-props-elims*)

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

by (auto simp: elim!: call-cond-elims intro!: call-cond-intros simp: DFS-ret2-def)

lemma *ret2-holds*[*ret-holds-intros*]:
 assumes *DFS-dom* *dfs-state* return (*DFS dfs-state*) = *Reachable*
 shows *DFS-ret-2-conds* (*DFS dfs-state*)
 using *assms*(2-)
proof(*induction* rule: *DFS-induct*[*OF assms*(1)])
 case *IH*: (1 *dfs-state*)
 show ?*case*
 apply(*rule DFS-cases*[**where** *dfs-state* = *dfs-state*])
 using *IH*(4)
 by (auto intro: *ret-holds-intros* intro!: *IH*(2-) simp: *DFS-simps*[*OF IH*(1)]
DFS-ret1-def)
qed

lemma *DFS-correct-ret-2*:
 [[*invar-stack-walk* *dfs-state*; *DFS-ret-2-conds* *dfs-state*]]
 \implies *vwalk-bet* (*Graph.digraph-abs* *G*) (*last* (*stack* *dfs-state*)) (*rev* (*stack*
dfs-state)) *t*
 by (auto elim!: *call-cond-elims* *invar-props-elims* simp: *hd-rev* *vwalk-bet-def*)

6.8 Termination

named-theorems *termination-intros*

declare *termination-intros*

lemma *in-prod-rel*[*intro!*,*termination-intros*]:
 [[*f1 a* = *f1 a'*; (*a*, *a'*) \in *f2* <*$mlex$*> *r*]] \implies (*a*, *a'*) \in (*f1* <*$mlex$*> *f2* <*$mlex$*>
r)
 by (*simp* add: *mlex-iff*)+

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

lemma *wf-less-rel*[*intro!*]: *wf less-rel*
 by(*auto* simp: *less-rel-def* *wf-less*)

lemma *call-1-measure-nonsym*[*simp*]: (*call-1-measure* *dfs-state*, *call-1-measure* *dfs-state*)
 \notin *less-rel*
 by (*auto* simp: *less-rel-def*)

lemma *call-1-terminates*[*termination-intros*]:
 [[*DFS-call-1-conds* *dfs-state*; *invar-well-formed* *dfs-state*; *invar-seen-stack* *dfs-state*]]
 \implies
 (*DFS-upd1* *dfs-state*, *dfs-state*) \in *call-1-measure* <*$mlex$*> *r*
 by(*fastforce* elim!: *invar-props-elims* *call-cond-elims*
simp add: *DFS-upd1-def* *call-1-measure-def* *Let-def*
intro!: *mlex-less* *psubset-card-mono*
dest!: *Graph.vset.choose*¹)

lemma *call-2-measure-nonsym*[*simp*]: (*call-2-measure dfs-state, call-2-measure dfs-state*)
 \notin *less-rel*
by (*auto simp: less-rel-def*)

lemma *call-2-measure-1*[*termination-intros*]:
 $\llbracket \text{DFS-call-2-conds } dfs\text{-state}; \text{invar-well-formed } dfs\text{-state} \rrbracket \implies$
 $\text{call-1-measure } dfs\text{-state} = \text{call-1-measure } (\text{DFS-upd2 } dfs\text{-state})$
by(*auto simp add: DFS-upd2-def call-1-measure-def Let-def*
intro!: psubset-card-mono)

lemma *call-2-terminates*[*termination-intros*]:
 $\llbracket \text{DFS-call-2-conds } dfs\text{-state}; \text{invar-well-formed } dfs\text{-state}; \text{invar-seen-stack } dfs\text{-state} \rrbracket$
 \implies
 $(\text{DFS-upd2 } dfs\text{-state}, dfs\text{-state}) \in \text{call-2-measure } <*\text{mlex}*> r$
by(*auto elim!: invar-props-elims elim!: call-cond-elims*
simp add: DFS-upd2-def call-2-measure-def
intro!: mlex-less)

lemma *wf-term-rel*: *wf DFS-term-rel*
by(*auto simp: wf-mlex DFS-term-rel-def*)

lemma *in-DFS-term-rel*[*termination-intros*]:
 $\llbracket \text{DFS-call-1-conds } dfs\text{-state}; \text{invar-well-formed } dfs\text{-state}; \text{invar-seen-stack } dfs\text{-state} \rrbracket$
 \implies
 $(\text{DFS-upd1 } dfs\text{-state}, dfs\text{-state}) \in \text{DFS-term-rel}$
 $\llbracket \text{DFS-call-2-conds } dfs\text{-state}; \text{invar-well-formed } dfs\text{-state}; \text{invar-seen-stack } dfs\text{-state} \rrbracket$
 \implies
 $(\text{DFS-upd2 } dfs\text{-state}, dfs\text{-state}) \in \text{DFS-term-rel}$
by (*simp add: DFS-term-rel-def termination-intros*)+

lemma *DFS-terminates*[*termination-intros*]:
assumes *invar-well-formed dfs-state invar-seen-stack dfs-state*
shows *DFS-dom dfs-state*
using *wf-term-rel assms*
proof(*induction rule: wf-induct-rule*)
case (*less x*)
show *?case*
by (*rule DFS-domintros*) (*auto intro!: invar-holds-intros less in-DFS-term-rel*)
qed

6.9 Final Correctness Theorems

lemma *initial-state-props*[*invar-holds-intros, termination-intros*]:
 $\text{invar-well-formed } (\text{initial-state}) \text{invar-stack-walk } (\text{initial-state}) \text{invar-seen-stack}$
 (initial-state)
 $\text{invar-visited-through-seen } (\text{initial-state}) \text{invar-s-in-stack } \text{initial-state}$
 $\text{DFS-dom } \text{initial-state}$
by (*auto simp: initial-state-def*)

```

      hd-of-vwalk-bet''
    elim: vwalk-betE
    intro!: termination-intros invar-props-intros)

```

```

lemma DFS-correct-1:
  assumes return (DFS initial-state) = NotReachable
  shows  $\nexists p. \text{distinct } p \wedge \text{vwalk-bet } (\text{Graph.digraph-abs } G) \text{ } s \text{ } p \text{ } t$ 
proof -
  have  $s \in t\text{-set } (\text{seen } (\text{DFS initial-state}))$ 
    by(auto intro!: invar-holds-intros ret-holds-intros state-rel-holds-intros
      intro: state-rel-1-props[where ?dfs-state-1.0 = initial-state]
      simp add: initial-state-def)
  thus ?thesis
  using assms
  by(intro DFS-correct-ret-1[where dfs-state = DFS initial-state]
    (auto intro!: invar-holds-intros ret-holds-intros state-rel-holds-intros))
qed

```

```

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 -
  have vwalk-bet
    (Graph.digraph-abs G)
    (last (stack (DFS initial-state)))
    (rev (stack (DFS initial-state))) t
  using assms
  by(auto intro!: invar-holds-intros ret-holds-intros state-rel-holds-intros
    DFS-correct-ret-2[where dfs-state = DFS initial-state])
  moreover hence (last (stack (DFS initial-state))) = s
    by(fastforce intro!: invar-holds-intros
      intro: invar-s-in-stack-props[where dfs-state = DFS initial-state])+
  ultimately show ?thesis
    by auto
qed
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, 'uset) BFS-state = parents:: 'parents current:: 'uset visited:: 'uset

```

7.2 Setup for Automation

named-theorems *call-cond-elim*
named-theorems *call-cond-intros*
named-theorems *ret-holds-intros*
named-theorems *invar-props-intros*
named-theorems *invar-props-elim*
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*]:

$\llbracket \text{Graph.graph-inv BFS-tree; vset-inv frontier; vset-inv vis; Graph.graph-inv } G \rrbracket$

⇒

Graph.graph-inv (*expand-tree BFS-tree frontier vis*)

$\llbracket \text{Graph.graph-inv BFS-tree; vset-inv frontier; vset-inv vis; Graph.graph-inv } G \rrbracket$

⇒

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*]:

$\llbracket \text{vset-inv frontier; vset-inv vis; Graph.graph-inv } G \rrbracket \implies \text{vset-inv (next-frontier frontier vis)}$

$\llbracket \text{vset-inv frontier; vset-inv vis; Graph.graph-inv } G \rrbracket \implies$

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* \longleftrightarrow

$Graph.graph\text{-}inv\ G \wedge Graph.\text{finite}\text{-}graph\ G \wedge Graph.\text{finite}\text{-}vsets \wedge$
 $t\text{-}set\ srcs \subseteq dVs\ (Graph.\text{digraph}\text{-}abs\ G) \wedge$
 $(\forall u.\ \text{finite}\ (Pair\text{-}Graph.\text{neighbourhood}\ (Graph.\text{digraph}\text{-}abs\ G)\ u)) \wedge$
 $t\text{-}set\ srcs \neq \{\}$ $\wedge\ vset\text{-}inv\ srcs$

abbreviation $neighb' \equiv Graph.\text{neighb}\ G$

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

7.4 The Algorithm Definition

function $(domintros)\ BFS::('adjmap, 'vset)\ BFS\text{-}state \Rightarrow ('adjmap, 'vset)\ BFS\text{-}state$
where

$BFS\ BFS\text{-}state =$
 $($
 \quad *if current BFS-state* $\neq \emptyset_V$ *then*
 \quad *let*
 \quad $visited' = visited\ BFS\text{-}state \cup_G\ current\ BFS\text{-}state;$
 \quad $parents' = expand\text{-}tree\ (parents\ BFS\text{-}state)\ (current\ BFS\text{-}state)\ visited';$
 \quad $current' = next\text{-}frontier\ (current\ BFS\text{-}state)\ visited'$
 \quad *in*
 \quad $BFS\ (BFS\text{-}state\ (\{parents:=\ parents',\ visited := visited',\ current :=$
 $current'\}))$
 \quad *else*
 \quad $BFS\text{-}state)$
by *pat-completeness auto*

7.5 Setup for Reasoning About the Algorithm

definition $BFS\text{-}call\text{-}1\text{-}conds\ bfs\text{-}state = ((current\ bfs\text{-}state) \neq \emptyset_V)$

definition $BFS\text{-}upd1\ BFS\text{-}state =$

$($
 \quad *let*
 \quad $visited' = visited\ BFS\text{-}state \cup_G\ current\ BFS\text{-}state;$
 \quad $parents' = expand\text{-}tree\ (parents\ BFS\text{-}state)\ (current\ BFS\text{-}state)\ visited';$
 \quad $current' = next\text{-}frontier\ (current\ BFS\text{-}state)\ visited'$
 \quad *in*
 \quad $BFS\text{-}state\ (\{parents:=\ parents',\ visited := visited',\ current := current'\})$
 $)$

definition $BFS\text{-}ret\text{-}1\text{-}conds\ bfs\text{-}state = ((current\ bfs\text{-}state) = \emptyset_V)$

abbreviation $BFS\text{-}ret1\ bfs\text{-}state \equiv bfs\text{-}state$

lemma $DFS\text{-}call\text{-}1\text{-}conds[call\text{-}cond\text{-}elims]:$

$BFS\text{-}call\text{-}1\text{-}conds\ bfs\text{-}state \Longrightarrow$
 $\llbracket (current\ bfs\text{-}state) \neq \emptyset_V \Longrightarrow P \rrbracket$
 $\Longrightarrow P$

by(auto simp: BFS-call-1-conds-def split: list.splits option.splits if-splits)

lemma *BFS-ret-1-conds*[call-cond-elim]:

BFS-ret-1-conds bfs-state \implies
 $\llbracket (\text{current bfs-state}) = \emptyset_V \implies P \rrbracket$
 $\implies P$

by(auto simp: BFS-ret-1-conds-def split: list.splits option.splits if-splits)

lemma *BFS-ret-1-condsI*[call-cond-intros]:

$\llbracket (\text{current bfs-state}) = \emptyset_V \rrbracket \implies \text{BFS-ret-1-conds bfs-state}$

by(auto simp: BFS-ret-1-conds-def split: list.splits option.splits if-splits)

lemma *BFS-cases*:

assumes *BFS-call-1-conds* bfs-state $\implies P$

BFS-ret-1-conds bfs-state $\implies P$

shows *P*

proof –

have *BFS-call-1-conds* bfs-state \vee

BFS-ret-1-conds bfs-state

by (auto simp add: *BFS-call-1-conds-def*

BFS-ret-1-conds-def

split: list.split-asm option.split-asm if-splits)

then show ?thesis

using *assms*

by auto

qed

lemma *BFS-simps*:

assumes *BFS-dom* *BFS-state*

shows *BFS-call-1-conds* *BFS-state* $\implies \text{BFS } \text{BFS-state} = \text{BFS } (\text{BFS-upd1 } \text{BFS-state})$

BFS-ret-1-conds *BFS-state* $\implies \text{BFS } \text{BFS-state} = \text{BFS-ret1 } \text{BFS-state}$

by (auto simp add: *BFS.psimps*[OF *assms*]

BFS-call-1-conds-def *BFS-upd1-def*

BFS-ret-1-conds-def *Let-def*

split: list.splits option.splits if-splits)

lemma *BFS-induct*:

assumes *BFS-dom* bfs-state

assumes $\wedge \text{bfs-state. } \llbracket \text{BFS-dom } \text{bfs-state};$

$(\text{BFS-call-1-conds } \text{bfs-state} \implies P (\text{BFS-upd1 } \text{bfs-state})) \rrbracket$

$\implies P \text{ bfs-state}$

shows *P* bfs-state

apply(rule *BFS.pinduct*)

subgoal using *assms*(1) .

apply(rule *assms*(2))

by (auto simp add: *BFS-call-1-conds-def* *BFS-upd1-def* *Let-def*

split: list.splits option.splits if-splits)

lemma *BFS-dominros*:
assumes *BFS-call-1-conds* *BFS-state* \implies *BFS-dom* (*BFS-upd1* *BFS-state*)
shows *BFS-dom* *BFS-state*
proof(*rule* *BFS.dominros*, *goal-cases*)
case (1)
then show ?*case*
using *assms*(1)[*simplified BFS-call-1-conds-def* *BFS-upd1-def*]
by (*force simp: Let-def split: list.splits option.splits if-splits*)
qed

7.6 The Loop Invariants

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

definition *invar-subsets*::('adjmap, 'vset) *BFS-state* \Rightarrow *bool* **where**
invar-subsets *bfs-state* = (
Graph.digraph-abs (*parents* *bfs-state*) \subseteq *Graph.digraph-abs* *G* \wedge
t-set (*visited* *bfs-state*) \subseteq *dVs* (*Graph.digraph-abs* *G*) \wedge
t-set (*current* *bfs-state*) \subseteq *dVs* (*Graph.digraph-abs* *G*) \wedge
dVs (*Graph.digraph-abs* (*parents* *bfs-state*)) \subseteq *t-set* (*visited* *bfs-state*) \cup *t-set*
(*current* *bfs-state*) \wedge
t-set *srcs* \subseteq *t-set* (*visited* *bfs-state*) \cup *t-set* (*current* *bfs-state*)

definition *invar-3-1* *bfs-state* =
 $(\forall v \in t\text{-set } (current\ bfs\ state). \forall u. u \in t\text{-set } (current\ bfs\ state) \longleftrightarrow$
distance-set (*Graph.digraph-abs* *G*) (*t-set* *srcs*) *v* =

$$distance\text{-set } (Graph.digraph\text{-abs } G) (t\text{-set } srcs) u)$$

definition *invar-3-2* *bfs-state* =
 $(\forall v \in t\text{-set } (current\ bfs\ state). \forall u \in t\text{-set } (visited\ bfs\ state) \cup t\text{-set } (current\ bfs\ state).$
distance-set (*Graph.digraph-abs* *G*) (*t-set* *srcs*) *u* \leq
distance-set (*Graph.digraph-abs* *G*) (*t-set* *srcs*) *v*)

definition *invar-3-3* *bfs-state* =
 $(\forall v \in t\text{-set } (visited\ bfs\ state).$
neighbourhood (*Graph.digraph-abs* *G*) *v* \subseteq *t-set* (*visited* *bfs-state*) \cup *t-set*
(*current* *bfs-state*))

definition *invar-dist-bounded*::('adjmap, 'vset) *BFS-state* \Rightarrow *bool* **where**
invar-dist-bounded *bfs-state* =
 $(\forall v \in t\text{-set } (visited\ bfs\ state) \cup t\text{-set } (current\ bfs\ state).$
 $\forall u. distance\text{-set } (Graph.digraph\text{-abs } G) (t\text{-set } srcs) u \leq$
distance-set (*Graph.digraph-abs* *G*) (*t-set* *srcs*) *v*
 $\longrightarrow u \in t\text{-set } (visited\ bfs\ state) \cup t\text{-set } (current\ bfs\ state))$

definition *invar-goes-through-current*::('adjmap, 'vset) BFS-state \Rightarrow bool **where**
invar-goes-through-current bfs-state =
 $(\forall u \in t\text{-set } (visited\ bfs\text{-state}) \cup t\text{-set } (current\ bfs\text{-state}).$
 $\forall v. v \notin t\text{-set } (visited\ bfs\text{-state}) \cup t\text{-set } (current\ bfs\text{-state}) \longrightarrow$
 $(\forall p. Vwalk.vwalk\text{-bet } (Graph.digraph\text{-abs } G) u\ p\ v \longrightarrow$
 $set\ p \cap t\text{-set } (current\ bfs\text{-state}) \neq \{\})$)

definition *invar-dist*::('adjmap, 'vset) BFS-state \Rightarrow bool **where**
invar-dist bfs-state =
 $(\forall v \in dVs (Graph.digraph\text{-abs } G) - t\text{-set } srcs.$
 $(v \in (t\text{-set } (visited\ bfs\text{-state}) \cup t\text{-set } (current\ bfs\text{-state})) \longrightarrow distance\text{-set}$
 $(Graph.digraph\text{-abs } G) (t\text{-set } srcs) v =$
 $distance\text{-set } (Graph.digraph\text{-abs } (parents\ bfs\text{-state})) (t\text{-set } srcs) v))$

definition *invar-parents-shortest-paths*::('adjmap, 'vset) BFS-state \Rightarrow bool **where**
invar-parents-shortest-paths bfs-state =
 $(\forall u \in t\text{-set } srcs.$
 $\forall p\ v. Vwalk.vwalk\text{-bet } (Graph.digraph\text{-abs } (parents\ bfs\text{-state})) u\ p\ v \longrightarrow$
 $length\ p - 1 = distance\text{-set } (Graph.digraph\text{-abs } G) (t\text{-set } srcs) v)$

7.7 Termination Measures and Relation

definition *call-1-measure-1*::('adjmap, 'vset) BFS-state \Rightarrow nat **where**
call-1-measure-1 bfs-state =
 $card (dVs (Graph.digraph\text{-abs } G) - ((t\text{-set } (visited\ bfs\text{-state})) \cup t\text{-set } (current\ bfs\text{-state})))$

definition *call-1-measure-2*::('adjmap, 'vset) BFS-state \Rightarrow nat **where**
call-1-measure-2 bfs-state =
 $card (t\text{-set } (current\ bfs\text{-state}))$

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

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

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]:
 $Graph.graph\text{-inv } G$
 $Graph.finite\text{-graph } G$
 $Graph.finite\text{-vsets}$ **and**
 $srcs\text{-in-}G$ [simp,intro]:

$t\text{-set srcs} \subseteq dVs$ (*Graph.digraph-abs* G) **and**
finite-vset:
 $finite$ (*Pair-Graph.neighbourhood* (*Graph.digraph-abs* G) u) **and**
srcs-invar[simp]:
 $t\text{-set srcs} \neq \{\}$
 $vset\text{-inv srcs}$
using $\langle \text{BFS-axiom} \rangle$
by (*auto simp: BFS-axiom-def*)

lemma *invar-well-formed-props[invar-props-elim]*:
 $invar\text{-well-formed bfs-state} \implies$
 $([vset\text{-inv} (visited\ bfs\text{-state}) ; vset\text{-inv} (current\ bfs\text{-state}) ;$
 $Graph.graph\text{-inv} (parents\ bfs\text{-state}) ;$
 $finite (t\text{-set} (current\ bfs\text{-state})) ; finite (t\text{-set} (visited\ bfs\text{-state}))]) \implies P$
 $\implies P$
by (*auto simp: invar-well-formed-def*)

lemma *invar-well-formed-intro[invar-props-intros]*:
 $[vset\text{-inv} (visited\ bfs\text{-state}) ; vset\text{-inv} (current\ bfs\text{-state}) ;$
 $Graph.graph\text{-inv} (parents\ bfs\text{-state}) ;$
 $finite (t\text{-set} (current\ bfs\text{-state})) ; finite (t\text{-set} (visited\ bfs\text{-state}))]$
 $\implies invar\text{-well-formed bfs-state}$
by (*auto simp: invar-well-formed-def*)

lemma *finite-simp*:
 $\{(u,v). u \in front \wedge v \in (Pair\text{-Graph.neighbourhood} (Graph.digraph\text{-abs } G) u) \wedge$
 $v \notin vis\} =$
 $\{(u,v). u \in front\} \cap \{(u,v). v \in (Pair\text{-Graph.neighbourhood} (Graph.digraph\text{-abs}$
 $G) u)\} - \{(u,v) . v \in vis\}$
 $finite \{(u, v) \mid v . v \in (s\ u)\} = finite (s\ u)$
using *finite-image-iff* [**where** $f = snd$ **and** $A = \{(u, v) \mid v . v \in s\ u\}$]
by (*auto simp: inj-on-def image-def*)

lemma *invar-well-formed-holds-upd1[invar-holds-intros]*:
 $[BFS\text{-call-1-conds bfs-state} ; invar\text{-well-formed bfs-state}] \implies invar\text{-well-formed}$
 $(BFS\text{-upd1 bfs-state})$
using *finite-vset*
by (*auto elim!: invar-well-formed-props call-cond-elim simp: Let-def BFS-upd1-def*
BFS-call-1-conds-def intro!: invar-props-intros) $+$

lemma *invar-well-formed-holds-ret-1[invar-holds-intros]*:
 $[BFS\text{-ret-1-conds bfs-state} ; invar\text{-well-formed bfs-state}] \implies invar\text{-well-formed}$
 $(BFS\text{-ret1 bfs-state})$
by (*auto simp: intro: invar-props-intros*)

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)*
using *assms(2-)*

proof(*induction rule: BFS-induct*[*OF assms*(1)])
case *IH*: (1 *bfs-state*)
show ?*case*
apply(*rule BFS-cases*[**where** *bfs-state = bfs-state*])
by (*auto intro!*: *IH*(2-) *intro: invar-holds-intros simp: BFS-simps*[*OF IH*(1)])
qed

lemma *invar-subsets-props*[*invar-props-elim*s]:
invar-subsets bfs-state \implies
 $\llbracket \text{Graph.digraph-abs (parents bfs-state)} \subseteq \text{Graph.digraph-abs } G;$
 $t\text{-set (visited bfs-state)} \subseteq dVs (\text{Graph.digraph-abs } G);$
 $t\text{-set (current bfs-state)} \subseteq dVs (\text{Graph.digraph-abs } G);$
 $dVs (\text{Graph.digraph-abs (parents bfs-state)}) \subseteq t\text{-set (visited bfs-state)} \cup t\text{-set (current bfs-state)};$
 $t\text{-set srcs} \subseteq t\text{-set (visited bfs-state)} \cup t\text{-set (current bfs-state)} \rrbracket \implies P$
 $\implies P$
by (*auto simp: invar-subsets-def*)

lemma *invar-subsets-intro*[*invar-props-intro*s]:
 $\llbracket \text{Graph.digraph-abs (parents bfs-state)} \subseteq \text{Graph.digraph-abs } G;$
 $t\text{-set (visited bfs-state)} \subseteq dVs (\text{Graph.digraph-abs } G);$
 $t\text{-set (current bfs-state)} \subseteq dVs (\text{Graph.digraph-abs } G);$
 $dVs (\text{Graph.digraph-abs (parents bfs-state)}) \subseteq t\text{-set (visited bfs-state)} \cup t\text{-set (current bfs-state)};$
 $t\text{-set srcs} \subseteq t\text{-set (visited bfs-state)} \cup t\text{-set (current bfs-state)} \rrbracket$
 $\implies \text{invar-subsets bfs-state}$
by (*auto simp: invar-subsets-def*)

lemma *invar-subsets-holds-upd1*[*invar-holds-intro*s]:
 $\llbracket \text{BFS-call-1-conds bfs-state}; \text{invar-well-formed bfs-state}; \text{invar-subsets bfs-state} \rrbracket$
 $\implies \text{invar-subsets (BFS-upd1 bfs-state)}$
apply(*auto elim!*: *call-cond-elim*s *invar-well-formed-props invar-subsets-props intro!*: *invar-props-intro simp: BFS-upd1-def Let-def*)
apply(*auto simp: dVs-def*)
apply (*metis Un-iff dVsI*(1) *dVs-def in-mono*)
by (*metis Un-iff dVsI*(2) *dVs-def in-mono*)

lemma *invar-subsets-holds-ret-1*[*invar-holds-intro*s]:
 $\llbracket \text{BFS-ret-1-conds bfs-state}; \text{invar-subsets bfs-state} \rrbracket \implies \text{invar-subsets (BFS-ret1 bfs-state)}$
by (*auto simp: intro: invar-props-intros*)

lemma *invar-subsets-holds*[*invar-holds-intro*s]:
assumes *BFS-dom bfs-state invar-well-formed bfs-state invar-subsets bfs-state*
shows *invar-subsets (BFS bfs-state)*
using *assms*(2-)
proof(*induction rule: BFS-induct*[*OF assms*(1)])
case *IH*: (1 *bfs-state*)
show ?*case*

apply(*rule BFS-cases*[**where** *bfs-state = bfs-state*])
by (*auto intro!*: *IH(2-)* *intro: invar-holds-intros simp: BFS-simps[OF IH(1)]*)
qed

— This is *invar_100* on the board

lemma *invar-3-1-props*[*invar-props-elim*]:
invar-3-1 bfs-state \implies
 $(\llbracket v \in t\text{-set } (\text{current } \text{bfs-state}); u \in t\text{-set } (\text{current } \text{bfs-state}) \rrbracket \implies$
 $\text{distance-set } (\text{Graph.digraph-abs } G) (t\text{-set srcs}) v =$
 $\text{distance-set } (\text{Graph.digraph-abs } G) (t\text{-set srcs}) u;$
 $\llbracket v \in t\text{-set } (\text{current } \text{bfs-state});$
 $\text{distance-set } (\text{Graph.digraph-abs } G) (t\text{-set srcs}) v =$
 $\text{distance-set } (\text{Graph.digraph-abs } G) (t\text{-set srcs}) u \rrbracket \implies$
 $u \in t\text{-set } (\text{current } \text{bfs-state}) \rrbracket \implies P$
 $\implies P$
unfolding *invar-3-1-def*
by *blast*

lemma *invar-3-1-intro*[*invar-props-intros*]:
 $(\llbracket \bigwedge u v. \llbracket v \in t\text{-set } (\text{current } \text{bfs-state}); u \in t\text{-set } (\text{current } \text{bfs-state}) \rrbracket \implies$
 $\text{distance-set } (\text{Graph.digraph-abs } G) (t\text{-set srcs}) v =$
 $\text{distance-set } (\text{Graph.digraph-abs } G) (t\text{-set srcs}) u;$
 $\bigwedge u v. \llbracket v \in t\text{-set } (\text{current } \text{bfs-state}); \text{distance-set } (\text{Graph.digraph-abs } G) (t\text{-set$
 $\text{srcs}) v =$
 $\text{distance-set } (\text{Graph.digraph-abs } G) (t\text{-set srcs}) u \rrbracket \implies$
 $u \in t\text{-set } (\text{current } \text{bfs-state}) \rrbracket \implies$
 $\implies \text{invar-3-1 bfs-state}$
unfolding *invar-3-1-def*
by *blast*

lemma *invar-3-2-props*[*elim*]:
invar-3-2 bfs-state \implies
 $(\llbracket \bigwedge v u. \llbracket v \in t\text{-set } (\text{current } \text{bfs-state}); u \in t\text{-set } (\text{visited } \text{bfs-state}) \cup t\text{-set } (\text{current}$
 $\text{bfs-state}) \rrbracket \implies$
 $\text{distance-set } (\text{Graph.digraph-abs } G) (t\text{-set srcs}) u \leq$
 $\text{distance-set } (\text{Graph.digraph-abs } G) (t\text{-set srcs}) v \rrbracket \implies P$
 $\implies P$
unfolding *invar-3-2-def*
by *blast*

lemma *invar-3-2-intro*[*invar-props-intros*]:
 $(\llbracket \bigwedge v u. \llbracket v \in t\text{-set } (\text{current } \text{bfs-state}); u \in t\text{-set } (\text{visited } \text{bfs-state}) \cup t\text{-set } (\text{current}$
 $\text{bfs-state}) \rrbracket \implies$
 $\text{distance-set } (\text{Graph.digraph-abs } G) (t\text{-set srcs}) u \leq$
 $\text{distance-set } (\text{Graph.digraph-abs } G) (t\text{-set srcs}) v \rrbracket \implies$
 $\implies \text{invar-3-2 bfs-state}$
unfolding *invar-3-2-def*
by *blast*

lemma *invar-3-3-props*[*invar-props-elim*s]:
invar-3-3 bfs-state \implies
 $([\bigwedge v. [v \in t\text{-set } (visited\ bfs\text{-state})]] \implies$
 $\quad neighbourhood\ (Graph.digraph\text{-abs } G)\ v \subseteq t\text{-set } (visited\ bfs\text{-state}) \cup t\text{-set}$
 $(current\ bfs\text{-state})] \implies P)$
 $\implies P$
unfolding *invar-3-3-def*
by *blast*

lemma *invar-3-3-intro*[*invar-props-intro*s]:
 $([\bigwedge v. [v \in t\text{-set } (visited\ bfs\text{-state})]] \implies$
 $\quad neighbourhood\ (Graph.digraph\text{-abs } G)\ v \subseteq t\text{-set } (visited\ bfs\text{-state}) \cup t\text{-set}$
 $(current\ bfs\text{-state})]$
 $\implies invar\text{-}3\text{-}3\ bfs\text{-state}$
unfolding *invar-3-3-def*
by *blast*

lemma *invar-dist-bounded-props*[*invar-props-elim*s]:
invar-dist-bounded bfs-state \implies
 $([\bigwedge u\ v. [v \in t\text{-set } (visited\ bfs\text{-state}) \cup t\text{-set } (current\ bfs\text{-state});$
 $\quad distance\text{-set } (Graph.digraph\text{-abs } G)\ (t\text{-set } srcs)\ u \leq distance\text{-set}$
 $(Graph.digraph\text{-abs } G)\ (t\text{-set } srcs)\ v]] \implies$
 $\quad u \in t\text{-set } (visited\ bfs\text{-state}) \cup t\text{-set } (current\ bfs\text{-state})] \implies P)$
 $\implies P$
unfolding *invar-dist-bounded-def*
by *blast*

lemma *invar-dist-bounded-intro*[*invar-props-intro*s]:
 $([\bigwedge u\ v. [v \in t\text{-set } (visited\ bfs\text{-state}) \cup t\text{-set } (current\ bfs\text{-state});$
 $\quad distance\text{-set } (Graph.digraph\text{-abs } G)\ (t\text{-set } srcs)\ u \leq distance\text{-set}$
 $(Graph.digraph\text{-abs } G)\ (t\text{-set } srcs)\ v]] \implies$
 $\quad u \in t\text{-set } (visited\ bfs\text{-state}) \cup t\text{-set } (current\ bfs\text{-state})]$
 $\implies invar\text{-}dist\text{-}bounded\ bfs\text{-state}$
unfolding *invar-dist-bounded-def*
by *blast*

definition *invar-current-reachable bfs-state* =
 $(\forall v \in t\text{-set } (visited\ bfs\text{-state}) \cup t\text{-set } (current\ bfs\text{-state}).$
 $\quad distance\text{-set } (Graph.digraph\text{-abs } G)\ (t\text{-set } srcs)\ v < \infty)$

lemma *invar-current-reachable-props*[*invar-props-elim*s]:
invar-current-reachable bfs-state \implies
 $([\bigwedge v. [v \in t\text{-set } (visited\ bfs\text{-state}) \cup t\text{-set } (current\ bfs\text{-state})]] \implies$
 $\quad distance\text{-set } (Graph.digraph\text{-abs } G)\ (t\text{-set } srcs)\ v < \infty] \implies P)$
 $\implies P$
by (*auto simp: invar-current-reachable-def*)

lemma *invar-current-reachable-intro*[*invar-props-intro*s]:

$\llbracket \wedge v. \llbracket v \in t\text{-set } (\text{visited bfs-state}) \cup t\text{-set } (\text{current bfs-state}) \rrbracket \implies$
 $\text{distance-set } (\text{Graph.digraph-abs } G) (t\text{-set srcs}) v < \infty \rrbracket$
 $\implies \text{invar-current-reachable bfs-state}$
by(*auto simp add: invar-current-reachable-def*)

7.8 Proofs that the Invariants Hold

lemma *invar-current-reachable-holds-upd1* [*invar-holds-intros*]:

$\llbracket \text{BFS-call-1-conds bfs-state; invar-well-formed bfs-state; invar-subsets bfs-state;}$
 $\text{invar-current-reachable bfs-state} \rrbracket$

$\implies \text{invar-current-reachable } (\text{BFS-upd1 bfs-state})$

proof(*rule invar-props-intros, goal-cases*)

case *assms*: (1 *v*)

have *?case* (**is** *?dist v < ∞*) **if** $v \notin t\text{-set } (\text{visited bfs-state}) \cup t\text{-set } (\text{current bfs-state})$

proof–

have $v \in t\text{-set } (\text{current } (\text{BFS-upd1 bfs-state}))$

using *that assms*

by (*auto simp: BFS-upd1-def Let-def elim: invar-props-elims*)

then obtain *u* **where** $v \in t\text{-set } (\mathcal{N}_G u) \ u \in t\text{-set } (\text{current bfs-state})$

using *assms*

by (*auto simp: BFS-upd1-def Let-def elim!: invar-props-elims*)

hence *?dist u < ∞*

using $\langle \text{invar-subsets bfs-state} \rangle \langle \text{invar-current-reachable bfs-state} \rangle$

by (*auto elim!: invar-props-elims*)

hence *?dist v ≤ ?dist u + 1*

using $\langle v \in t\text{-set } (\mathcal{N}_G u) \rangle$

by (*auto intro!: distance-set-neighbourhood*)

thus *?thesis*

using *add-mono1* [*OF* $\langle ?dist u < \infty \rangle$] *linorder-not-less*

by *fastforce*

qed

thus *?case*

using *assms*

by(*force elim!: call-cond-elims invar-props-elims intro!: invar-props-intros simp: BFS-upd1-def Let-def*)

qed

lemma *invar-current-reachable-holds-ret-1* [*invar-holds-intros*]:

$\llbracket \text{BFS-ret-1-conds bfs-state; invar-current-reachable bfs-state} \rrbracket \implies \text{invar-current-reachable}$
 $(\text{BFS-ret1 bfs-state})$

by (*auto simp: intro: invar-props-intros*)

lemma *dist-current-plus-1-new*:

assumes

invar-well-formed bfs-state invar-subsets bfs-state invar-dist-bounded bfs-state

v ∈ neighbourhood (Graph.digraph-abs G) v'

v' ∈ t-set (current bfs-state)

v ∈ t-set (current (BFS-upd1 bfs-state))

shows $\text{distance-set } (\text{Graph.digraph-abs } G) (t\text{-set srcs}) v =$
 $\text{distance-set } (\text{Graph.digraph-abs } G) (t\text{-set srcs}) v' + 1$ (**is** $?dv = ?dv' +$
1)

proof–
have *False* **if** $?dv > ?dv' + 1$
using $\text{distance-set-neighbourhood}[OF \langle v \in \text{neighbourhood } (\text{Graph.digraph-abs } G) v' \rangle]$ *that*
by (*simp add: leD*)

moreover have *False* **if** $?dv < ?dv' + 1$
proof–
have $?dv \leq ?dv'$
using *that eSuc-plus-1 ileI1*
by *force*
moreover have $?dv \neq \infty$
using *that enat-ord-simps(4)*
by *fastforce*
moreover have $v \notin t\text{-set } (\text{visited bfs-state}) \cup t\text{-set } (\text{current bfs-state})$
using *assms*
by (*auto simp: BFS-upd1-def Let-def elim!: invar-well-formed-props invar-subsets-props*)

moreover have $v' \in t\text{-set } (\text{visited bfs-state}) \cup t\text{-set } (\text{current bfs-state})$
using $\langle v' \in t\text{-set } (\text{current bfs-state}) \rangle \langle \text{invar-subsets bfs-state} \rangle$
by (*auto elim!: invar-subsets-props*)

ultimately show *False*
using $\langle \text{invar-dist-bounded bfs-state} \rangle$
apply (*elim invar-props-elim*)
apply (*drule dist-set-not-inf*)
using *dual-order.trans dist-set-mem*
by (*smt (verit, best)*)

qed
ultimately show *?thesis*
by *force*
qed

lemma *plus-lt-enat*: $\llbracket (a::\text{enat}) \neq \infty; b < c \rrbracket \implies a + b < a + c$
using *enat-add-left-cancel-less*
by *presburger*

lemma *plus-one-side-lt-enat*: $\llbracket (a::\text{enat}) \neq \infty; 0 < b \rrbracket \implies a < a + b$
using *plus-lt-enat*
by *auto*

lemma *invar-3-1-holds-upd1-new*[*invar-holds-intros*]:
 $\llbracket \text{BFS-call-1-conds bfs-state}; \text{invar-well-formed bfs-state}; \text{invar-subsets bfs-state}; \text{invar-3-1 bfs-state};$

```

    invar-3-2 bfs-state; invar-dist-bounded bfs-state; invar-current-reachable bfs-state]]
    ==> invar-3-1 (BFS-upd1 bfs-state)
proof(intro invar-props-intros, goal-cases)
  case assms: (1 u v)
  obtain u' v' where
    uv'[intro]: u ∈ neighbourhood (Graph.digraph-abs G) u' u' ∈ t-set (current
    bfs-state)
    v ∈ neighbourhood (Graph.digraph-abs G) v' v' ∈ t-set (current
    bfs-state)
    using assms(1,2,8,9)
    by (auto simp: BFS-upd1-def Let-def elim!: invar-well-formed-props)
  moreover hence distance-set (Graph.digraph-abs G) (t-set srcs) v' =
    distance-set (Graph.digraph-abs G) (t-set srcs) u' (is ?d v' = ?d u')
    using ⟨invar-3-1 bfs-state⟩
    by (auto elim: invar-props-elim)
  moreover have distance-set (Graph.digraph-abs G) (t-set srcs) v = ?d v' + 1
    using assms
    by (auto intro!: dist-current-plus-1-new)
  moreover have distance-set (Graph.digraph-abs G) (t-set srcs) u = ?d u' + 1
    using assms
    by (auto intro!: dist-current-plus-1-new)
  ultimately show ?case
    by auto
next
  case (2 u v)
  then obtain v' where uv'[intro]:
    v ∈ neighbourhood (Graph.digraph-abs G) v' v' ∈ t-set (current bfs-state)
    by (auto simp: BFS-upd1-def Let-def elim!: invar-well-formed-props invar-subsets-props)
  hence distance-set (Graph.digraph-abs G) (t-set srcs) v =
    distance-set (Graph.digraph-abs G) (t-set srcs) v' + 1 (is ?d v = ?d v' +
  -)
    using 2
    by(fastforce intro!: dist-current-plus-1-new)

  show ?case
  proof(cases 0 < ?d u)
    case in-srcs: True
    moreover have ?d v' < ∞
      using ⟨?d v = ?d u⟩ ⟨invar-well-formed bfs-state⟩ ⟨invar-subsets bfs-state⟩ ⟨v'
    ∈ t-set (current bfs-state)⟩
      ⟨invar-current-reachable bfs-state⟩
    by (auto elim!: invar-well-formed-props invar-subsets-props invar-current-reachable-props)
    hence ?d v < ∞
      using ⟨?d v = ?d v' + 1⟩
      by (simp add: plus-eq-infty-iff-enat)
    hence ?d u < ∞
      using ⟨?d v = ?d u⟩
      by auto
    ultimately obtain u' where ?d u' + 1 = ?d u u ∈ neighbourhood (Graph.digraph-abs

```

G) u'

```

using distance-set-parent'
by (metis srcs-invar(1))
hence  $?d\ u' = ?d\ v'$ 
using  $\langle ?d\ v = ?d\ v' + 1 \rangle \langle ?d\ v = ?d\ u \rangle$ 
by auto
hence  $u' \in t\text{-set}\ (current\ bfs\text{-state})$ 
using  $\langle invar\text{-}3\text{-}1\ bfs\text{-state} \rangle$ 
 $\langle v' \in t\text{-set}\ (current\ bfs\text{-state}) \rangle$ 
by (auto elim!: invar-3-1-props)
moreover have  $?d\ u' < ?d\ u$ 
using  $\langle ?d\ u < \infty \rangle$ 
using zero-less-one not-infinity-eq
by (fastforce intro!: plus-one-side-lt-enat simp:  $\langle ?d\ u' + 1 = ?d\ u \rangle$ [symmetric])
hence  $u \notin t\text{-set}\ (visited\ bfs\text{-state}) \cup t\text{-set}\ (current\ bfs\text{-state})$ 
using  $\langle invar\text{-}3\text{-}2\ bfs\text{-state} \rangle \langle u' \in t\text{-set}\ (current\ bfs\text{-state}) \rangle$ 
by (auto elim!: invar-3-2-props dest: leD)
ultimately show ?thesis
using  $\langle invar\text{-well-formed}\ bfs\text{-state} \rangle \langle invar\text{-subsets}\ bfs\text{-state} \rangle \langle u \in neighbourhood$ 
(Graph.digraph-abs G)  $u' \rangle$ 
apply(auto simp: BFS-upd1-def Let-def elim!: invar-well-formed-props in-
var-subsets-props)
by blast
next
case not-in-srcs: False

```

Contradiction because if $u \in srcs$ then distance $srcs$ to a vertex in $srcs$ is > 0 . This is because the distance from $srcs$ to u is the same as that to v .

```

then show ?thesis
using  $\langle ?d\ v = ?d\ u \rangle \langle ?d\ v = ?d\ v' + 1 \rangle$ 
by auto
qed
qed

```

lemma *invar-3-1-holds-ret-1*[*invar-holds-intros*]:
 $\llbracket BFS\text{-ret-1-conds}\ bfs\text{-state}; invar\text{-}3\text{-}3\ bfs\text{-state} \rrbracket \implies invar\text{-}3\text{-}3\ (BFS\text{-ret1}\ bfs\text{-state})$
by (*auto simp: intro: invar-props-intros*)

lemma *invar-3-2-holds-upd1-new*[*invar-holds-intros*]:
 $\llbracket BFS\text{-call-1-conds}\ bfs\text{-state}; invar\text{-well-formed}\ bfs\text{-state}; invar\text{-subsets}\ bfs\text{-state};$
 $invar\text{-}3\text{-}1\ bfs\text{-state};$
 $invar\text{-}3\text{-}2\ bfs\text{-state}; invar\text{-dist-bounded}\ bfs\text{-state}; invar\text{-current-reachable}\ bfs\text{-state} \rrbracket$
 $\implies invar\text{-}3\text{-}2\ (BFS\text{-upd1}\ bfs\text{-state})$
proof(*intro invar-props-intros, goal-cases*)
case *assms: (1 u v)*
show *?case*
proof(*cases v ∈ t-set (current (BFS-upd1 bfs-state))*)
case *v-in-current: True*
moreover have *invar-3-1 (BFS-upd1 bfs-state)*

```

    using assms
    by (auto intro: invar-holds-intros)
  ultimately show ?thesis
    using ⟨u ∈ t-set (current (BFS-upd1 bfs-state))⟩
    by (fastforce elim: invar-props-elim.s)
next
case v-not-in-current: False
hence v ∈ t-set (visited bfs-state) ∪ t-set (current bfs-state)
  using assms(1,2,9)
  by (fastforce simp: BFS-upd1-def Let-def elim!: invar-well-formed-props)
moreover obtain u' where uv'[intro]: u ∈ neighbourhood (Graph.digraph-abs
G) u' u' ∈ t-set (current bfs-state)
  using assms(1,2,8,9)
  by (auto simp: BFS-upd1-def Let-def elim!: invar-well-formed-props)
ultimately have distance-set (Graph.digraph-abs G) (t-set srcs) v ≤
  distance-set (Graph.digraph-abs G) (t-set srcs) u'
  using ⟨invar-3-2 bfs-state⟩
  by (auto elim!: invar-3-2-props)
moreover have distance-set (Graph.digraph-abs G) (t-set srcs) u =
  distance-set (Graph.digraph-abs G) (t-set srcs) u' + 1 (is ?d u = ?d u' +
- )
  using assms
  by(fastforce intro!: dist-current-plus-1-new)
ultimately show ?thesis
  by (metis le-iff-add order.trans)
qed
qed

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)
  by (auto simp: intro: invar-props-intros)

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)
  by(fastforce elim!: call-cond-elim.s invar-well-formed-props invar-subsets-props in-
var-3-3-props intro!: invar-props-intros simp: BFS-upd1-def Let-def)

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)
  by (auto simp: intro: invar-props-intros)

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(intro invar-props-intros, goal-cases)

```

```

case assms: (1 u v)
show ?case
proof(cases ⟨v ∈ t-set (visited bfs-state) ∪ t-set (current bfs-state)⟩)
  case v-visited: True
  hence u ∈ t-set (visited bfs-state) ∪ t-set (current bfs-state)
  using ⟨invar-dist-bounded bfs-state⟩ assms
  by (auto elim!: invar-dist-bounded-props)
  then show ?thesis
    using ⟨invar-well-formed bfs-state⟩ ⟨v ∈ t-set (visited (BFS-upd1 bfs-state))
    ∪ t-set (current (BFS-upd1 bfs-state))⟩
    by (auto simp: BFS-upd1-def Let-def elim: invar-props-elim)
  next
  case v-not-visited: False
  hence v ∈ t-set (current (BFS-upd1 bfs-state))
  using ⟨invar-well-formed bfs-state⟩ ⟨v ∈ t-set (visited (BFS-upd1 bfs-state))
    ∪ t-set (current (BFS-upd1 bfs-state))⟩
  by (auto simp: BFS-upd1-def Let-def elim: invar-props-elim)
  then obtain v' where v'[intro]:
    v ∈ neighbourhood (Graph.digraph-abs G) v' v' ∈ t-set (current bfs-state)
  using ⟨invar-well-formed bfs-state⟩
  by (auto simp: BFS-upd1-def Let-def elim!: invar-well-formed-props)
  have 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)
  using assms ⟨v ∈ t-set (current (BFS-upd1 bfs-state))⟩
  by (auto intro!: dist-current-plus-1-new)
  moreover have u ∈ t-set (visited (BFS-upd1 bfs-state)) ∪ t-set (current
(BFS-upd1 bfs-state))
  if distance-set (Graph.digraph-abs G) (t-set srcs) u ≤ ?dv' (is ?du ≤ ?dv')
  using that ⟨invar-well-formed bfs-state⟩ ⟨invar-subsets bfs-state⟩ ⟨invar-dist-bounded
bfs-state⟩
  by (fastforce simp: BFS-upd1-def Let-def elim!: invar-well-formed-props in-
var-subsets-props invar-dist-bounded-props)
  moreover have u ∈ t-set (visited (BFS-upd1 bfs-state)) ∪ t-set (current
(BFS-upd1 bfs-state))
  if distance-set (Graph.digraph-abs G) (t-set srcs) u = ?dv
proof–
  have invar-3-1 (BFS-upd1 bfs-state)
  by (auto intro: assms invar-holds-intros)
  hence u ∈ t-set (current (BFS-upd1 bfs-state))
  using that ⟨invar-3-1 bfs-state⟩ ⟨v ∈ t-set (current (BFS-upd1 bfs-state))⟩
  by (auto elim!: invar-3-1-props)
  moreover have invar-well-formed (BFS-upd1 bfs-state) invar-subsets (BFS-upd1
bfs-state)
  using ⟨BFS-call-1-conds bfs-state⟩ ⟨invar-well-formed bfs-state⟩ ⟨invar-subsets
bfs-state⟩
  by(auto intro!: invar-well-formed-holds-upd1 invar-subsets-holds-upd1)
  ultimately show ?thesis
  by (auto elim!: invar-props-elim)

```

qed
ultimately show *?thesis*
using $\langle ?du \leq ?dv \rangle$ *ileI1 linorder-not-less plus-1-eSuc(2)*
by *fastforce*
qed
qed

lemma *invar-dist-bounded-holds-ret-1* [*invar-holds-intros*]:
 $\llbracket \text{BFS-ret-1-conds } \text{bfs-state}; \text{invar-dist-bounded } \text{bfs-state} \rrbracket \implies \text{invar-dist-bounded}$
 $(\text{BFS-ret1 } \text{bfs-state})$
by (*auto simp: intro: invar-props-intros*)

lemma *invar-dist-props* [*invar-props-elim*s]:
 $\text{invar-dist } \text{bfs-state} \implies v \in \text{dVs } (\text{Graph.digraph-abs } G) - \text{t-set srcs} \implies$
 \llbracket
 $\llbracket v \in (\text{t-set } (\text{visited } \text{bfs-state}) \cup \text{t-set } (\text{current } \text{bfs-state})) \implies \text{distance-set}$
 $(\text{Graph.digraph-abs } G) (\text{t-set srcs}) v =$
 $\text{distance-set } (\text{Graph.digraph-abs } (\text{parents } \text{bfs-state})) (\text{t-set srcs}) v \rrbracket \implies P$
 \rrbracket
 $\implies P$
unfolding *invar-dist-def*
by *auto*

lemma *invar-dist-intro* [*invar-props-intros*]:
 $\llbracket \bigwedge v. \llbracket v \in \text{dVs } (\text{Graph.digraph-abs } G) - \text{t-set srcs}; v \in \text{t-set } (\text{visited } \text{bfs-state}) \cup$
 $\text{t-set } (\text{current } \text{bfs-state}) \rrbracket \implies$
 $(\text{distance-set } (\text{Graph.digraph-abs } G) (\text{t-set srcs}) v =$
 $\text{distance-set } (\text{Graph.digraph-abs } (\text{parents } \text{bfs-state})) (\text{t-set srcs}) v) \rrbracket$
 $\implies \text{invar-dist } \text{bfs-state}$
unfolding *invar-dist-def*
by *auto*

lemma *invar-goes-through-current-props* [*invar-props-elim*s]:
 $\text{invar-goes-through-current } \text{bfs-state} \implies$
 $\llbracket \llbracket \bigwedge u v p. \llbracket u \in \text{t-set } (\text{visited } \text{bfs-state}) \cup \text{t-set } (\text{current } \text{bfs-state});$
 $v \notin \text{t-set } (\text{visited } \text{bfs-state}) \cup \text{t-set } (\text{current } \text{bfs-state});$
 $V\text{walk.vwalk-bet } (\text{Graph.digraph-abs } G) u p v \rrbracket$
 $\implies \text{set } p \cap \text{t-set } (\text{current } \text{bfs-state}) \neq \{\}$
 $\implies P \rrbracket$
 $\implies P$
unfolding *invar-goes-through-current-def*
by *auto*

lemma *invar-goes-through-current-intro* [*invar-props-intros*]:
 $\llbracket \bigwedge u v p. \llbracket u \in \text{t-set } (\text{visited } \text{bfs-state}) \cup \text{t-set } (\text{current } \text{bfs-state});$
 $v \notin \text{t-set } (\text{visited } \text{bfs-state}) \cup \text{t-set } (\text{current } \text{bfs-state});$
 $V\text{walk.vwalk-bet } (\text{Graph.digraph-abs } G) u p v \rrbracket$

```

    ⇒ set p ∩ t-set (current bfs-state) ≠ {}
    ⇒ invar-goes-through-current bfs-state
unfolding invar-goes-through-current-def
by auto

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(intro invar-props-intros, goal-cases)
  case (1 u v p)
  hence v ∉ t-set (visited bfs-state) ∪ t-set (current bfs-state)
  by (auto simp: Let-def BFS-upd1-def elim!: invar-well-formed-props invar-subsets-props)
  show ?case
  proof(cases u ∈ t-set (visited bfs-state) ∪ t-set (current bfs-state))
    case u-in-visited: True
      have Vwalk.vwalk-bet (Graph.digraph-abs G) u p v set p ∩ t-set (current
bfs-state) ≠ {}
      using ⟨invar-goes-through-current bfs-state⟩
        ⟨v ∉ t-set (visited bfs-state) ∪ t-set (current bfs-state)⟩
        ⟨vwalk-bet (Graph.digraph-abs G) u p v⟩ u-in-visited
      by (auto elim!: invar-goes-through-current-props)
      moreover have u ∈ set p
      using ⟨Vwalk.vwalk-bet (Graph.digraph-abs G) u p v⟩
      by(auto intro: hd-of-vwalk-bet'')
      ultimately have ∃ p1 x p2. p = p1 @ [x] @ p2 ∧
        x ∈ t-set (current bfs-state) ∧
        (∀ y ∈ set p2. y ∉ (t-set (visited bfs-state) ∪ t-set (current
bfs-state))) ∧
        y ∉ (t-set (current bfs-state)))
      using ⟨invar-goes-through-current bfs-state⟩ u-in-visited
        ⟨v ∉ t-set (visited bfs-state) ∪ t-set (current bfs-state)⟩
        ⟨invar-well-formed bfs-state⟩ ⟨invar-subsets bfs-state⟩
      apply (intro list-2-preds[where ?P2.0 = (λx. x ∈ t-set (current bfs-state)),
        simplified list-inter-mem-iff[symmetric]])
      by (fastforce elim!: invar-goes-through-current-props dest!: vwalk-bet-suff
split-vwalk)+

  then obtain p1 x p2 where p = p1 @ x # p2 and
    x ∈ t-set (current bfs-state) and
    unvisited:
    (∧ y. y ∈ set p2 ⇒ y ∉ t-set (visited bfs-state) ∪ t-set (current bfs-state))
  by auto
  moreover have last p = v
  using ⟨vwalk-bet (Graph.digraph-abs G) u p v⟩
  by auto
  ultimately have v ∈ set p2
  using ⟨v ∉ t-set (visited bfs-state) ∪ t-set (current bfs-state)⟩
    ⟨invar-well-formed bfs-state⟩ ⟨invar-subsets bfs-state⟩

```

```

    by (auto elim: invar-props-elim)
  then obtain v' p2' where p2 = v' # p2'
    by (cases p2) auto
  hence v' ∈ neighbourhood (Graph.digraph-abs G) x
    using ⟨p = p1 @ x # p2⟩ ⟨Vwalk.vwalk-bet (Graph.digraph-abs G) u p v⟩
      split-vwalk
    by fastforce
  moreover have v' ∈ set p2
    using ⟨p2 = v' # p2'⟩
    by auto
  ultimately have v' ∈ t-set (current (BFS-upd1 bfs-state))
    using ⟨invar-well-formed bfs-state⟩ ⟨invar-subsets bfs-state⟩ ⟨x ∈ t-set (current
bfs-state)⟩
    by(fastforce simp: BFS-upd1-def Let-def elim!: invar-well-formed-props in-
var-subsets-props dest!: unvisited)
  then show ?thesis
    by(auto intro!: invar-goes-through-current-intro simp: ⟨p = p1 @ x # p2⟩ ⟨p2
= v' # p2'⟩)
next
  case u-not-in-visited: False
  hence u ∈ t-set (current (BFS-upd1 bfs-state))
    using ⟨invar-well-formed bfs-state⟩
    ⟨u ∈ t-set (visited (BFS-upd1 bfs-state)) ∪ t-set (current (BFS-upd1 bfs-state))⟩
  by (auto simp: BFS-upd1-def Let-def elim: invar-props-elim)
  moreover have u ∈ set p
    using ⟨Vwalk.vwalk-bet (Graph.digraph-abs G) u p v⟩
    by (auto intro: hd-of-vwalk-bet'')
  ultimately show ?thesis
    by(auto intro!: invar-goes-through-current-intro)
qed
qed

```

lemma *invar-goes-through-current-holds-ret-1* [*invar-holds-intros*]:

$\llbracket \text{BFS-ret-1-conds } \text{bfs-state}; \text{invar-goes-through-current } \text{bfs-state} \rrbracket \implies \text{invar-goes-through-current } (\text{BFS-ret1 } \text{bfs-state})$

by (auto simp: intro: invar-props-intros)

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*)

using *assms*(2-)

proof (*induction rule: BFS-induct* [*OF* *assms*(1)])

case *IH*: (1 *bfs-state*)

show ?case

apply (*rule* *BFS-cases* [**where** *bfs-state* = *bfs-state*])

by (auto intro!: *IH*(2-) intro: *invar-holds-intros* simp: *BFS-simps* [*OF* *IH*(1)])

qed

```

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(intro invar-props-intros, goal-cases)
  define bfs-state' where bfs-state' = BFS-upd1 bfs-state
  let ?dSrcsG = distance-set (Graph.digraph-abs G) (t-set srcs)
  let ?dSrcsT = distance-set (Graph.digraph-abs (parents bfs-state)) (t-set srcs)
  let ?dSrcsT' = distance-set (Graph.digraph-abs (parents bfs-state')) (t-set srcs)
  let ?dCurrG = distance-set (Graph.digraph-abs G) (t-set (current bfs-state))
  case (1 v)
  then show ?case
  proof(cases distance-set (Graph.digraph-abs G) (t-set srcs) v = ∞)
    case infinite: True
    moreover have ?dSrcsG v ≤ ?dSrcsT' v
      using ⟨invar-well-formed bfs-state⟩ ⟨invar-subsets bfs-state⟩
    by(fastforce simp: bfs-state'-def BFS-upd1-def Let-def intro: distance-set-subset
      elim: invar-props-elims)
    ultimately show ?thesis
      by (simp add: bfs-state'-def)
  next
  case finite: False
  show ?thesis
  proof(cases v ∈ t-set (visited bfs-state) ∪ t-set (current bfs-state))
    case v-in-tree: True
    hence ?dSrcsT v = ?dSrcsG v
      using ⟨invar-dist bfs-state⟩ ⟨invar-well-formed bfs-state⟩ ⟨invar-subsets
bfs-state⟩
      ⟨v ∈ dVs (Graph.digraph-abs G) – t-set srcs⟩
    by (auto elim!: invar-dist-props invar-well-formed-props invar-subsets-props)

    moreover have ?dSrcsT v = ?dSrcsT' v
  proof–
    have ?dSrcsT' v ≤ ?dSrcsT v
      using ⟨invar-well-formed bfs-state⟩
      by(fastforce simp: bfs-state'-def BFS-upd1-def Let-def
        intro: distance-set-subset elim: invar-props-elims)

    moreover have ?dSrcsG v ≤ ?dSrcsT' v
      using ⟨invar-well-formed bfs-state⟩ ⟨invar-subsets bfs-state⟩
    by(fastforce simp: bfs-state'-def BFS-upd1-def Let-def intro: distance-set-subset
      elim: invar-props-elims)

    ultimately show ?thesis
      using ⟨?dSrcsT v = ?dSrcsG v⟩
      by auto
  qed
  ultimately show ?thesis

```

```

    by (auto simp: bfs-state'-def)
next
case v-not-in-tree: False

show ?thesis
proof(rule ccontr, goal-cases)
  case 1
  moreover have ⟨invar-subsets bfs-state'⟩
    using ⟨BFS-call-1-conds bfs-state⟩ ⟨invar-well-formed bfs-state⟩ ⟨in-
var-subsets bfs-state⟩
    by (auto intro!: invar-subsets-holds-upd1 simp: bfs-state'-def)
  hence ⟨Graph.digraph-abs (parents bfs-state') ⊆ Graph.digraph-abs G⟩
    by (auto elim: invar-props-elim)
  ultimately have ?dSrcsG v < ?dSrcsT' v
    by (simp add: distance-set-subset order-less-le bfs-state'-def)
  hence ?dSrcsG v < ?dSrcsT' v

because the tree is a subset of the Graph, which invar?

  by (simp add: distance-set-subset order-less-le bfs-state'-def)

  have v ∈ t-set (current (BFS-upd1 bfs-state))
    using ⟨v ∈ t-set (visited (BFS-upd1 bfs-state)) ∪ t-set (current (BFS-upd1
bfs-state))⟩
    v-not-in-tree ⟨invar-well-formed bfs-state⟩
    by (auto simp: BFS-upd1-def Let-def elim: invar-props-elim)
  moreover then obtain v' where v'[intro]:
    v ∈ neighbourhood (Graph.digraph-abs G) v'
    v' ∈ t-set (current bfs-state)
    v ∈ neighbourhood (Graph.digraph-abs (parents bfs-state')) v'
    using v-not-in-tree ⟨invar-well-formed bfs-state⟩
    by (auto simp: neighbourhoodD BFS-upd1-def Let-def bfs-state'-def elim!:
invar-well-formed-props)
  ultimately have ?dSrcsG v = ?dSrcsG v' + 1
    using ⟨invar-well-formed bfs-state⟩ ⟨invar-dist-bounded bfs-state⟩ ⟨in-
var-subsets bfs-state⟩
    by (auto intro!: dist-current-plus-1-new)
  show False
proof(cases v' ∈ t-set srcs)
  case v'-in-srcs: True
  hence ?dSrcsT' v' = 0
    by (meson dVsI(1) distance-set-0 neighbourhoodI v'(3))
  moreover have ?dSrcsG v' = 0
    using v'-in-srcs
    by (metis ⟨distance-set (Graph.digraph-abs G) (t-set srcs) v = dis-
tance-set (Graph.digraph-abs G) (t-set srcs) v' + 1⟩ add commute add.right-neutral
dist-set-inf dist-set-mem distance-0 enat-add-left-cancel le-iff-add local.finite order-antisym)
  then show ?thesis
  by (metis ⟨distance-set (Graph.digraph-abs G) (t-set srcs) v < distance-set

```

(*Graph.digraph-abs (parents bfs-state')*) (*t-set srcs*) *v* › *distance-set (Graph.digraph-abs G) (t-set srcs) v = distance-set (Graph.digraph-abs G) (t-set srcs) v' + 1* › calculation *distance-set-neighbourhood leD srcs-invar(1) v'(3)*)

next
case *v'-not-in-srcs: False*
have $?dSrcsG\ v = ?dSrcsG\ v' + 1$
using $\langle ?dSrcsG\ v = ?dSrcsG\ v' + 1 \rangle$.
also have $\dots = ?dSrcsT\ v' + 1$

From this current invariant

using $\langle invar-dist\ bfs-state \rangle \langle v' \in t-set\ (current\ bfs-state) \rangle \langle invar-well-formed\ bfs-state \rangle$

$\langle invar-subsets\ bfs-state \rangle\ v'-not-in-srcs$

by (*force elim!:* *invar-well-formed-props invar-subsets-props invar-dist-props*)

also have $\dots = ?dSrcsT'\ v' + 1$

proof –

have $?dSrcsT\ v' = ?dSrcsT'\ v'$

proof –

have $?dSrcsT'\ v' \leq ?dSrcsT\ v'$

using $\langle invar-well-formed\ bfs-state \rangle$

by(*fastforce simp: bfs-state'-def BFS-upd1-def Let-def*

*intro: distance-set-subset elim: invar-props-elim*s)

moreover have $?dSrcsG\ v' \leq ?dSrcsT'\ v'$

using $\langle invar-well-formed\ bfs-state \rangle \langle invar-subsets\ bfs-state \rangle$

by(*fastforce simp: bfs-state'-def BFS-upd1-def Let-def intro: distance-set-subset*

*elim: invar-props-elim*s)

moreover have $\langle ?dSrcsT\ v' = ?dSrcsG\ v' \rangle$

using $\langle invar-dist\ bfs-state \rangle \langle v' \in t-set\ (current\ bfs-state) \rangle \langle invar-well-formed\ bfs-state \rangle$

$\langle invar-subsets\ bfs-state \rangle\ v'-not-in-srcs$

by (*force elim!:* *invar-well-formed-props invar-subsets-props invar-dist-props*)

ultimately show *?thesis*

by *auto*

qed

then show *?thesis*

by *auto*

qed

finally have $?dSrcsG\ v = ?dSrcsT'\ v' + 1$

by *auto*

hence $?dSrcsT'\ v' + 1 < ?dSrcsT'\ v$

using $\langle ?dSrcsG\ v < ?dSrcsT'\ v \rangle$

by *auto*

moreover have $v \in neighbourhood\ (Graph.digraph-abs\ (parents\ bfs-state'))$

v'

using $\langle v \in neighbourhood\ (Graph.digraph-abs\ (parents\ bfs-state'))\ v' \rangle$.

hence $?dSrcsT'\ v \leq ?dSrcsT'\ v' + 1$

by (auto intro!: distance-set-neighbourhood)

ultimately show *False*

From the triangle ineq

by auto

qed

qed

qed

qed

lemma *invar-dist-holds-ret-1* [*invar-holds-intros*]:
 $\llbracket \text{BFS-ret-1-conds } \text{bfs-state}; \text{invar-dist } \text{bfs-state} \rrbracket \implies \text{invar-dist } (\text{BFS-ret1 } \text{bfs-state})$
 by (auto simp: intro: invar-props-intros)

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*)
 using *assms*(2-)

proof (*induction rule: BFS-induct* [*OF* *assms*(1)])
 case *IH*: (1 *bfs-state*)
 show ?case
 apply (*rule* *BFS-cases* [**where** *bfs-state* = *bfs-state*])
 by (auto intro!: *IH*(2-) intro: *invar-holds-intros* simp: *BFS-simps* [*OF* *IH*(1)])

qed

definition *invar-current-no-out* *bfs-state* =
 $(\forall u \in t\text{-set}(\text{current } \text{bfs-state}).$
 $\forall v. (u,v) \notin \text{Graph.digraph-abs } (\text{parents } \text{bfs-state}))$

lemma *invar-current-no-out-props* [*invar-props-elim*s]:
 $\text{invar-current-no-out } \text{bfs-state} \implies$
 $(\llbracket \bigwedge u v. u \in t\text{-set}(\text{current } \text{bfs-state}) \implies (u,v) \notin \text{Graph.digraph-abs } (\text{parents } \text{bfs-state}) \rrbracket$
 $\implies P)$
 $\implies P$
 by (auto simp: *invar-current-no-out-def*)

lemma *invar-current-no-out-intro* [*invar-props-intros*]:
 $\llbracket \bigwedge u v. u \in t\text{-set}(\text{current } \text{bfs-state}) \implies (u,v) \notin \text{Graph.digraph-abs } (\text{parents } \text{bfs-state}) \rrbracket$
 $\implies \text{invar-current-no-out } \text{bfs-state}$
 by (auto simp: *invar-current-no-out-def*)

lemma *invar-current-no-out-holds-upd1* [*invar-holds-intros*]:
 $\llbracket \text{BFS-call-1-conds } \text{bfs-state}; \text{invar-well-formed } \text{bfs-state}; \text{invar-subsets } \text{bfs-state};$
 $\text{invar-current-no-out } \text{bfs-state} \rrbracket$

\implies *invar-current-no-out* (*BFS-upd1* *bfs-state*)
apply(*auto elim!*: *call-cond-elim*s *invar-well-formed-props* *invar-subsets-props* *intro!*: *invar-props-intros* *simp*: *BFS-upd1-def* *Let-def*)
apply *blast+*
done

lemma *invar-current-no-out-holds-ret-1*[*invar-holds-intros*]:
 $\llbracket \text{BFS-ret-1-conds } bfs\text{-state}; \text{invar-current-no-out } bfs\text{-state} \rrbracket \implies \text{invar-current-no-out } (BFS\text{-ret1 } bfs\text{-state})$
by (*auto simp*: *intro*: *invar-props-intros*)

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*)
using *assms*(2-)
proof(*induction* *rule*: *BFS-induct*[*OF* *assms*(1)])
case *IH*: (1 *bfs-state*)
show ?*case*
apply(*rule* *BFS-cases*[**where** *bfs-state* = *bfs-state*])
by (*auto intro!*: *IH*(2-) *intro*: *invar-holds-intros* *simp*: *BFS-simps*[*OF* *IH*(1)])
qed

lemma *invar-parents-shortest-paths-props*[*invar-props-elim*s]:
invar-parents-shortest-paths *bfs-state* \implies
 $(\llbracket \bigwedge u\ p\ v. \llbracket u \in t\text{-set } srcs; \forall walk.vwalk\text{-bet } (Graph.digraph\text{-abs } (parents\ bfs\text{-state}))\ u\ p\ v \rrbracket \implies$
 $\text{length } p - 1 = \text{distance-set } (Graph.digraph\text{-abs } G) (t\text{-set } srcs) v \rrbracket \implies P$
 $\implies P$
by (*auto simp*: *invar-parents-shortest-paths-def*)

lemma *invar-parents-shortest-paths-intro*[*invar-props-intros*]:
 $(\llbracket \bigwedge u\ p\ v. \llbracket u \in t\text{-set } srcs; \forall walk.vwalk\text{-bet } (Graph.digraph\text{-abs } (parents\ bfs\text{-state}))\ u\ p\ v \rrbracket \implies$
 $\text{length } p - 1 = \text{distance-set } (Graph.digraph\text{-abs } G) (t\text{-set } srcs) v \rrbracket$
 $\implies \text{invar-parents-shortest-paths } bfs\text{-state}$
by (*auto simp*: *invar-parents-shortest-paths-def*)

lemma *invar-parents-shortest-paths-holds-upd1*[*invar-holds-intros*]:
 $\llbracket \text{BFS-call-1-conds } bfs\text{-state}; \text{invar-well-formed } bfs\text{-state}; \text{invar-subsets } bfs\text{-state};$
invar-current-no-out *bfs-state*;
invar-dist-bounded *bfs-state*; *invar-parents-shortest-paths* *bfs-state*
 $\implies \text{invar-parents-shortest-paths } (BFS\text{-upd1 } bfs\text{-state})$
proof(*intro* *invar-props-intros*, *goal-cases*)
case *assms*: (1 *u p v*)

define *bfs-state'* **where** *bfs-state'* = *BFS-upd1* *bfs-state*

have *invar-subsets* *bfs-state'*

```

using assms
by (auto intro: invar-holds-intros simp: bfs-state'-def)

hence ?case if length p ≤ 1
  using  $\langle \text{length } p \leq 1 \rangle$  assms(7,8) \langle invar-subsets bfs-state \rangle
by(cases p) (auto elim!: Vwalk.vwalk-bet-props invar-props-elim dest!: dVs-subset
  simp: bfs-state'-def[symmetric] zero-enat-def[symmetric])

have invar-current-no-out bfs-state'
  using assms
by(auto intro: invar-holds-intros simp: bfs-state'-def)

have **:  $u \in t\text{-set } (\text{current } \text{bfs-state}') \vee v \in t\text{-set } (\text{current } \text{bfs-state}')$ 
  if  $(u,v) \in (\text{Graph.digraph-abs } (\text{parents } \text{bfs-state}')) -$ 
   $(\text{Graph.digraph-abs } (\text{parents } \text{bfs-state}'))$  for  $u v$ 
  using  $\langle \text{invar-well-formed } \text{bfs-state} \rangle \langle \text{invar-subsets } \text{bfs-state} \rangle$  dVsI that
by(fastforce dest: dVsI simp: bfs-state'-def dVs-def BFS-upd1-def Let-def
  elim!: invar-well-formed-props invar-subsets-props)

have ?case if length p > 1
proof–
  have  $u \in t\text{-set } (\text{visited } \text{bfs-state}) \cup t\text{-set } (\text{current } \text{bfs-state})$ 
proof(rule ccontr, goal-cases)
  have  $u \in dVs (\text{Graph.digraph-abs } (\text{parents } \text{bfs-state}'))$ 
  using assms(8)
  by (auto simp: bfs-state'-def)
  hence  $u \in t\text{-set } (\text{visited } \text{bfs-state}') \cup t\text{-set } (\text{current } \text{bfs-state}')$ 
  using  $\langle \text{invar-subsets } \text{bfs-state}' \rangle$ 
  by (auto elim: invar-props-elim)
  moreover case 1
  ultimately have  $u \in t\text{-set } (\text{current } \text{bfs-state}')$ 
  using assms
by(auto simp: Let-def bfs-state'-def BFS-upd1-def elim!: invar-well-formed-props
invar-subsets-props)
  moreover obtain  $u'$  where  $(u,u') \in \text{Graph.digraph-abs } (\text{parents } \text{bfs-state}')$ 
  using  $\langle \text{length } p > 1 \rangle$  assms(8) \langle invar-subsets bfs-state \rangle
by (auto elim!: Vwalk.vwalk-bet-props
  simp: eval-nat-numeral Suc-le-length-iff Suc-le-eq[symmetric] bfs-state'-def
  split: if-splits)
  ultimately show ?case
  using  $\langle \text{invar-current-no-out } \text{bfs-state}' \rangle$ 
  by (auto elim!: invar-current-no-out-props)
qed

show ?thesis
proof(cases v ∉ t-set (visited bfs-state) ∪ t-set (current bfs-state))
  case v-not-visited: True
  show ?thesis
proof(rule ccontr, goal-cases)

```

case 1
moreover have *vwalk-bet* (*Graph.digraph-abs G*) *u p v*
by (*metis* \langle *invar-subsets bfs-state'* \rangle *assms*(8) *bfs-state'-def invar-subsets-props vwalk-bet-subset*)

ultimately have *length p - 1 > distance-set* (*Graph.digraph-abs G*) (*t-set srcs*) *v*
using \langle *u* \in *t-set srcs* \rangle 1
apply *auto*
by (*metis* *One-nat-def order-neq-le-trans vwalk-bet-dist-set*)

hence *length p - 2 \geq distance-set* (*Graph.digraph-abs G*) (*t-set srcs*) *v*
using \langle *length p > 1* \rangle
apply (*auto simp: eval-nat-numeral*)
by (*metis* *leD leI Suc-diff-Suc Suc-ile-eq*)
moreover obtain *p' v' where p = p' @ [v', v]*
using \langle *length p > 1* \rangle
apply (*clarsimp*
simp: eval-nat-numeral Suc-le-length-iff Suc-le-eq[symmetric]
split: if-splits)
by (*metis* *append.right-neutral append-butlast-last-cancel assms*(8) *length-Cons length-butlast length-tl list.sel*(3) *list.size*(3) *nat.simps*(3) *vwalk-bet-def*)
have *vwalk-bet* (*Graph.digraph-abs* (*parents bfs-state*)) *u (p' @ [v']) v'*
proof(*rule ccontr, goal-cases*)
case 1
moreover have *vwalk-bet* (*Graph.digraph-abs* (*parents (BFS-upd1 bfs-state)*))
u (p' @ [v']) v'
using *assms*(8) \langle *p = p' @ [v', v]* \rangle
by (*simp add: vwalk-bet-pref*)
ultimately show ?*case*
proof(*elim vwalk-bet-not-vwalk-bet-elim-2[OF - 1], goal-cases*)
case 1
then show ?*case*
by (*metis* \langle *distance-set* (*Graph.digraph-abs G*) (*t-set srcs*) *v \leq enat*
(*length p - 2*) \rangle
 \langle *p = p' @ [v', v]* \rangle \langle *vwalk-bet* (*Graph.digraph-abs G*) *u p v* \rangle
assms(3)
diff-is-0-eq distance-set-0 invar-subsets-def le-zero-eq
length-append-singleton
list.sel(3) *not-less-eq-eq subset-eq v-not-visited vwalk-bet-endpoints*(2)
vwalk-bet-vertex-decompE zero-enat-def)

next
case (2 *u'' v''*)
moreover hence *u'' \notin t-set* (*current bfs-state'*)
using \langle *invar-current-no-out bfs-state'* \rangle \langle *invar-subsets bfs-state'* \rangle
by (*auto simp: bfs-state'-def[symmetric] elim: invar-props-elims*)
ultimately have *v'' \in t-set* (*current bfs-state'*)

```

    using ** ⟨invar-subsets bfs-state'⟩
    by (auto simp: bfs-state'-def[symmetric])
    moreover hence no-outgoing-v'': (v'',u'') ∉ Graph.digraph-abs (parents
bfs-state')
    for u''
    using ⟨invar-current-no-out bfs-state'⟩ that
    by (auto elim: invar-props-elim)
    hence last (p @ [v']) = v''
    using that ⟨vwalk-bet (Graph.digraph-abs (parents (BFS-upd1 bfs-state)))
u (p' @ [v']) v'⟩[simplified bfs-state'-def[symmetric]]
    apply (clarsimp dest: v-in-edge-in-vwalk elim!: vwalk-bet-props intro!:
no-outgoing-last)
    by (metis 2(2) last-snoc no-outgoing-last v-in-edge-in-vwalk(2))
    hence v' = v''
    using that
    by auto
    moreover have (v',v) ∈ Graph.digraph-abs (parents bfs-state')
    using ⟨p = p' @ [v', v]⟩ assms(8)
    by (fastforce simp: bfs-state'-def [symmetric] dest: split-vwalk)
    ultimately show ?case
    using that no-outgoing-v''
    by auto
  qed
qed
  hence length (p' @ [v']) - 1 = distance-set (Graph.digraph-abs G) (t-set
srcs) v'
    using ⟨invar-parents-shortest-paths bfs-state'⟩ ⟨u ∈ t-set srcs⟩
    by (force elim!: invar-props-elim)
  hence length p - 2 = distance-set (Graph.digraph-abs G) (t-set srcs) v' (is
- = ?dist v')
    by (auto simp: ⟨p = p' @ [v', v]⟩)
  hence ?dist v ≤ ?dist v'
    using ⟨?dist v ≤ length p - 2⟩ dual-order.trans
    by presburger
  hence v ∈ t-set (visited bfs-state) ∪ t-set (current bfs-state)
    using ⟨invar-subsets bfs-state'⟩ ⟨invar-dist-bounded bfs-state'⟩ ⟨u ∈ t-set
srcs⟩
    ⟨vwalk-bet (Graph.digraph-abs (parents bfs-state)) u (p' @ [v']) v'⟩
    by (blast elim!: invar-props-elim dest!: vwalk-bet-endpoints(2))
  thus ?case
    using v-not-visited
    by auto
qed
next
case v-visited: False

have Vwalk.vwalk-bet (Graph.digraph-abs (parents bfs-state)) u p v
proof(rule ccontr, goal-cases)
  case 1

```

```

thus ?case
proof(elim vwalk-bet-not-vwalk-bet-elim-2[OF assms(8)], goal-cases)
  case 1
  then show ?case
    using that by auto
  next
  case (2 u' v')
  moreover hence  $u' \notin t\text{-set}$  (current bfs-state')
    using  $\langle \text{invar-current-no-out } \text{bfs-state}' \rangle$   $\langle \text{invar-subsets } \text{bfs-state}' \rangle$ 
    by (auto simp: bfs-state'-def[symmetric] elim: invar-props-elims)
  ultimately have  $v' \in t\text{-set}$  (current bfs-state')
    using **  $\langle \text{invar-subsets } \text{bfs-state}' \rangle$ 
    by (auto simp: bfs-state'-def[symmetric])
  moreover hence  $(v',u') \notin \text{Graph.digraph-abs}$  (parents bfs-state') for u'
    using  $\langle \text{invar-current-no-out } \text{bfs-state}' \rangle$ 
    by (auto elim: invar-props-elims)
  hence last  $p = v'$ 
    using  $\langle \text{vwalk-bet} (\text{Graph.digraph-abs} (\text{parents} (\text{BFS-upd1 } \text{bfs-state}))) \text{ } u \text{ } p \rangle$ 
     $\langle \text{simplified } \text{bfs-state}'\text{-def}[\text{symmetric}] \rangle$ 
     $\langle (u',v') \in \text{set} (\text{edges-of-vwalk } p) \rangle$ 
  by (auto dest: v-in-edge-in-vwalk elim!: vwalk-bet-props intro!: no-outgoing-last)
  hence  $v' = v$ 
    using  $\langle \text{vwalk-bet} (\text{Graph.digraph-abs} (\text{parents} (\text{BFS-upd1 } \text{bfs-state}))) \text{ } u \text{ } p \rangle$ 
   $v \rangle$ 
  by auto
  ultimately show ?case
    using v-visited  $\langle \text{invar-well-formed } \text{bfs-state} \rangle$ 
    by (auto simp: bfs-state'-def BFS-upd1-def Let-def elim: invar-props-elims)
  qed
qed
then show ?thesis
  using  $\langle \text{invar-parents-shortest-paths } \text{bfs-state} \rangle$   $\langle u \in t\text{-set srcs} \rangle$ 
  by (auto elim!: invar-props-elims)
qed
qed
show ?case
  using  $\langle 1 < \text{length } p \implies \text{length } p - 1 = \text{distance-set} (\text{Graph.digraph-abs } G) (t\text{-set srcs}) \text{ } v \rangle$ 
   $\langle \text{length } p \leq 1 \implies \text{length } p - 1 = \text{distance-set} (\text{Graph.digraph-abs } G) (t\text{-set srcs}) \text{ } v \rangle$ 
  by fastforce
qed

```

lemma *invar-parents-shortest-paths-holds-ret-1*[*invar-holds-intros*]:
 $\llbracket \text{BFS-ret-1-conds } \text{bfs-state}; \text{invar-parents-shortest-paths } \text{bfs-state} \rrbracket \implies$
invar-parents-shortest-paths (*BFS-ret1* *bfs-state*)
by (*auto simp: intro: invar-props-intros*)

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)*
using *assms(2-)*
proof(*induction rule: BFS-induct[OF assms(1)]*)
case *IH: (1 bfs-state)*
show *?case*
apply(*rule BFS-cases[where bfs-state = bfs-state]*)
by (*auto intro!: IH(2-) intro: invar-holds-intros simp: BFS-simps[OF IH(1)]*)
qed

lemma *BFS-ret-1[ret-holds-intros]:*
BFS-ret-1-conds (bfs-state) \implies BFS-ret-1-conds (BFS-ret1 bfs-state)
by (*auto simp: elim!: call-cond-elims intro!: call-cond-intros*)

lemma *ret1-holds[ret-holds-intros]:*
assumes *BFS-dom bfs-state*
shows *BFS-ret-1-conds (BFS bfs-state)*
proof(*induction rule: BFS-induct[OF assms(1)]*)
case *IH: (1 bfs-state)*
show *?case*
apply(*rule BFS-cases[where bfs-state = bfs-state]*)
by (*auto intro: ret-holds-intros intro!: IH(2-) simp: BFS-simps[OF IH(1)]*)
qed

lemma *BFS-correct-1-ret-1:*
 \llbracket *invar-subsets bfs-state; invar-goes-through-current bfs-state; BFS-ret-1-conds bfs-state;*
u \in t-set srcs; t \notin t-set (visited bfs-state) \rrbracket
 \implies \nexists *p. vwalk-bet (Graph.digraph-abs G) u p t*
by (*force elim!: call-cond-elims invar-props-elims*)

lemma *BFS-correct-2-ret-1:*
 \llbracket *invar-well-formed bfs-state; invar-subsets bfs-state; invar-dist bfs-state; BFS-ret-1-conds*
bfs-state;
t \in t-set (visited bfs-state) - t-set srcs \rrbracket
 \implies *distance-set (Graph.digraph-abs G) (t-set srcs) t =*
distance-set (Graph.digraph-abs (parents bfs-state)) (t-set srcs) t
apply(*erule invar-props-elims*)
by (*auto elim!: call-cond-elims invar-props-elims*)

lemma *BFS-correct-3-ret-1:*
 \llbracket *invar-parents-shortest-paths bfs-state; BFS-ret-1-conds bfs-state;*
u \in t-set srcs; Vwalk.vwalk-bet (Graph.digraph-abs (parents bfs-state)) u p v \rrbracket
 \implies *length p - 1 = distance-set (Graph.digraph-abs G) (t-set srcs) v*
apply(*erule invar-props-elims*)
by (*auto elim!: call-cond-elims invar-props-elims*)

by (auto elim!: call-cond-elims invar-props-elims)

7.9 Termination

named-theorems *termination-intros*

declare *termination-intros*[intro!]

lemma *in-prod-rell*[intro!,*termination-intros*]:

$\llbracket f1\ a = f1\ a'; (a, a') \in f2\ \langle *mlex* \rangle\ r \rrbracket \implies (a, a') \in (f1\ \langle *mlex* \rangle\ f2\ \langle *mlex* \rangle\ r)$

by (*simp add: mlex-iff*)⁺

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

lemma *wf-less-rel*[intro!]: *wf less-rel*

by(*auto simp: less-rel-def wf-less*)

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

lemma *call-1-measure-nonsym*[*simp*]:

$(call-1-measure-1\ BFS-state, call-1-measure-1\ BFS-state) \notin less-rel$

by (*auto simp: less-rel-def*)

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 $(BFS-upd1\ BFS-state, BFS-state) \in$

$call-1-measure-1\ \langle *mlex* \rangle\ call-1-measure-2\ \langle *mlex* \rangle\ r$

proof(*cases t-set (next-frontier (current BFS-state) (visited BFS-state) \cup_G current BFS-state) = {}*)

case *True*

hence $t-set\ (visited\ (BFS-upd1\ BFS-state)) \cup t-set\ (current\ (BFS-upd1\ BFS-state))$

=

$t-set\ (visited\ BFS-state) \cup t-set\ (current\ BFS-state)$

using $\langle invar-well-formed\ BFS-state \rangle$

by (*fastforce simp: BFS-upd1-def Let-def elim!: invar-props-elims*)

hence $call-1-measure-1\ (BFS-upd1\ BFS-state) = call-1-measure-1\ BFS-state$

by (*auto simp: call-1-measure-1-def*)

moreover have $t-set\ (current\ (BFS-upd1\ BFS-state)) = \{\}$

using *True*

by (*auto simp: BFS-upd1-def Let-def*)

ultimately show *?thesis*

using *assms*

by(*fastforce elim!: invar-props-elims call-cond-elims*

simp add: call-1-measure-2-def

intro!: psubset-card-mono

intro: mlex-less)

```

next
case False
have *: { {v1, v2} | v1 v2. (v1, v2) ∈ [G]G }
      ⊆ (λ(x,y). {x,y}) ‘ {v. ∃ y. lookup G v = Some y} ×
      (⋃ {t-set N | v N. lookup G v = Some N})
  including Graph.adjmap.automation and Graph.vset.set.automation
  apply (clarsimp simp: Graph.digraph-abs-def Graph.neighb-def image-def
        split: option.splits)
  by (metis Graph.graph-invE Graph.vset.set.set-isin graph-inv(1))
moreover have {uu. ∃ v N. uu = t-set N ∧ lookup G v = Some N} =
  ((t-set o the o (lookup G)) ‘ {v | N v. lookup G v = Some N})
  by (force simp: image-def)
hence finite (⋃ {t-set N | v N. lookup G v = Some N})
  using graph-inv(1,2,3)
  apply(subst (asm) Graph.finite-vsets-def)
  by (auto simp: Graph.finite-graph-def Graph.graph-inv-def
      split: option.splits)
ultimately have finite { {v1, v2} | v1 v2. (v1,v2) ∈ [G]G }
  using graph-inv(2)
  by (auto simp: Graph.finite-graph-def intro!: finite-subset[OF *])
moreover have finite {neighbourhood (Graph.digraph-abs G) u | u. u ∈ t-set
  (current BFS-state)}
  using Graph.finite-vsets-def
  by (fastforce simp: )
moreover have t-set (visited (BFS-upd1 BFS-state)) ∪ t-set (current (BFS-upd1
BFS-state)) ⊆ dVs (Graph.digraph-abs G)
  using ⟨invar-well-formed BFS-state⟩ ⟨invar-subsets BFS-state⟩
  by(auto elim!: invar-props-elim call-cond-elim
      simp add: BFS-upd1-def Let-def dVs-def
      intro!: mlex-less psubset-card-mono)
moreover have t-set (visited (BFS-upd1 BFS-state)) ∪ t-set (current (BFS-upd1
BFS-state)) ≠
  t-set (visited BFS-state) ∪ t-set (current BFS-state)
  using ⟨BFS-call-1-conds BFS-state⟩ ⟨invar-well-formed BFS-state⟩ ⟨invar-subsets
BFS-state⟩
  ⟨invar-current-no-out BFS-state⟩ False
  by(fastforce elim!: invar-props-elim call-cond-elim
      simp add: BFS-upd1-def Let-def dVs-def
      intro!: mlex-less psubset-card-mono)

ultimately have **: dVs (Graph.digraph-abs G) – (t-set (visited (BFS-upd1
BFS-state)) ∪
  t-set (current (BFS-upd1 BFS-state))) ≠
  dVs (Graph.digraph-abs G) – (t-set (visited BFS-state) ∪ t-set
  (current BFS-state))
  using ⟨BFS-call-1-conds BFS-state⟩ ⟨invar-well-formed BFS-state⟩ ⟨invar-subsets
BFS-state⟩
  by(auto elim!: invar-props-elim call-cond-elim
      simp add: BFS-upd1-def Let-def dVs-def)

```

intro!: *mlex-less psubset-card-mono*)

hence *call-1-measure-1 (BFS-upd1 BFS-state) < call-1-measure-1 BFS-state*
using *assms*
by(*auto elim!*: *invar-props-elim call-cond-elim*
simp add: call-1-measure-1-def BFS-upd1-def Let-def
intro!: *psubset-card-mono*)
thus *?thesis*
by(*auto intro!*: *mlex-less psubset-card-mono*)
qed

lemma *wf-term-rel: wf BFS-term-rel*
by(*auto simp: wf-mlex BFS-term-rel-def*)

lemma *in-BFS-term-rel[termination-intros]*:
 $\llbracket \text{BFS-call-1-conds BFS-state; invar-well-formed BFS-state; invar-subsets BFS-state; invar-current-no-out BFS-state} \rrbracket \implies$
 $(\text{BFS-upd1 BFS-state, BFS-state}) \in \text{BFS-term-rel}$
by (*simp add: BFS-term-rel-def termination-intros*)+

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*
using *wf-term-rel assms*
proof(*induction rule: wf-induct-rule*)
case (*less x*)
show *?case*
apply (*rule BFS-domintros*)
by (*auto intro: invar-holds-intros intro!: termination-intros less*)
qed

lemma *not-vwalk-bet-empty[simp]*: $\neg \text{Vwalk.vwalk-bet (Graph.digraph-abs empty)}$
 $u \ p \ v$
using *not-vwalk-bet-empty*
by (*force simp add: Graph.digraph-abs-def Graph.neighb-def*)+

lemma *not-edge-in-empty[simp]*: $(u,v) \notin (\text{Graph.digraph-abs empty})$
by (*force simp add: Graph.digraph-abs-def Graph.neighb-def*)+

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 –
  show ?g1
    using graph-inv(3)
    by (fastforce simp: initial-state-def dVs-def Graph.finite-vsets-def
        intro!: invar-props-intros)

  have t-set (visited initial-state) ∪ t-set (current initial-state) ⊆ dVs (Graph.digraph-abs
  G)
    using srcs-in-G
    by (simp add: initial-state-def)
  thus ?g2 ?g3
    by (force simp: initial-state-def dVs-def Graph.digraph-abs-def Graph.neighb-def

        intro!: invar-props-intros)+

  show ?g4
    using ⟨?g1⟩ ⟨?g2⟩ ⟨?g3⟩
    by (auto intro!: termination-intros)

  show ?g5 invar-3-3 initial-state invar-parents-shortest-paths initial-state
    invar-current-no-out initial-state
    by (auto simp add: initial-state-def intro!: invar-props-intros)

  have *: distance-set (Graph.digraph-abs G) (t-set srcs) v = 0 if v ∈ t-set srcs
for v
    using that srcs-in-G
    by (fastforce intro: iffD2[OF distance-set-0[ where G = (Graph.digraph-abs
  G)])])
  moreover have **: v ∈ t-set srcs if distance-set (Graph.digraph-abs G) (t-set
  srcs) v = 0 for v
    using dist-set-inf i0-ne-infinity that
    by (force intro: iffD1[OF distance-set-0[ where G = (Graph.digraph-abs G)])])
  ultimately show invar-3-1 initial-state invar-3-2 initial-state invar-dist-bounded
  initial-state
    invar-current-reachable initial-state
    using dist-set-inf
    by (auto dest: dest: simp add: initial-state-def intro!: invar-props-intros dest!:)

  show invar-goes-through-current initial-state
    by (auto simp add: initial-state-def dest: hd-of-vwalk-bet'' intro!: invar-props-intros)
qed

```

lemma *BFS-correct-1*:

$\llbracket u \in t\text{-set } \text{srcs}; t \notin t\text{-set } (\text{visited } (\text{BFS } \text{initial-state})) \rrbracket$
 $\implies \nexists p. \text{vwalk-bet } (\text{Graph.digraph-abs } G) u p t$
apply(*intro* *BFS-correct-1-ret-1* [**where** *bfs-state* = *BFS initial-state*])
by(*auto intro: invar-holds-intros ret-holds-intros*)

lemma *BFS-correct-2*:

$\llbracket t \in t\text{-set } (\text{visited } (\text{BFS } \text{initial-state})) - t\text{-set } \text{srcs} \rrbracket$
 $\implies \text{distance-set } (\text{Graph.digraph-abs } G) (t\text{-set } \text{srcs}) t =$
 $\text{distance-set } (\text{Graph.digraph-abs } (\text{parents } (\text{BFS } \text{initial-state}))) (t\text{-set } \text{srcs}) t$
apply(*intro* *BFS-correct-2-ret-1* [**where** *bfs-state* = *BFS initial-state*])
by(*auto intro: invar-holds-intros ret-holds-intros*)

lemma *BFS-correct-3*:

$\llbracket u \in t\text{-set } \text{srcs}; \text{Vwalk.vwalk-bet } (\text{Graph.digraph-abs } (\text{parents } (\text{BFS } \text{initial-state})))$
 $u p v \rrbracket$
 $\implies \text{length } p - 1 = \text{distance-set } (\text{Graph.digraph-abs } G) (t\text{-set } \text{srcs}) v$
apply(*intro* *BFS-correct-3-ret-1* [**where** *bfs-state* = *BFS initial-state*])
by(*auto intro: invar-holds-intros ret-holds-intros*)

end

context

end

locale *BFS*

end