Graph Theory

By Lars Noschinski

September 13, 2023

Abstract

This development provides a formalization of directed graphs, supporting (labelled) multi-edges and infinite graphs. A polymorphic edge type allows edges to be treated as pairs of vertices, if multi-edges are not required. Formalized properties are i.a. walks (and related concepts), connectedness and subgraphs and basic properties of isomorphisms.

This formalization is used to prove characterizations of Euler Trails, Shortest Paths and Kuratowski subgraphs.

Definitions and nomenclature are based on [1].

Contents

theory *Rtrancl-On*
imports *Main*
begin

1 Reflexive-Transitive Closure on a Domain

In this section we introduce a variant of the reflexive-transitive closure of a relation which is useful to formalize the reachability relation on digraphs.

inductive-set
rtrancl-on :: *'a set* ⇒ *'a rel* ⇒ *'a rel*
for *F* :: *'a set* **and** *r* :: *'a rel*
where
rtrancl-on-refl [*intro*!, *Pure.intro*!, *simp*]: *a* ∈ *F* ⟹ (*a*, *a*) ∈ *rtrancl-on F r*
| *rtrancl-on-into-rtrancl-on* [*Pure.intro*]:
(*a*, *b*) ∈ *rtrancl-on F r* ⟹ (*b*, *c*) ∈ *r* ⟹ *c* ∈ *F*
⟹ (*a*, *c*) ∈ *rtrancl-on F r*

definition *symcl* :: *'a rel* ⇒ *'a rel* ((-^{s}) [*1000*] *999*) **where**
symcl R = *R* ∪ (λ(*a*,*b*). (*b*,*a*)) ‘ *R*

lemma *in-rtrancl-on-in-F*:
assumes (*a*,*b*) ∈ *rtrancl-on F r* **shows** *a* ∈ *F b* ∈ *F*
using *assms* **by** *induct auto*

lemma *rtrancl-on-induct*[*consumes 1*, *case-names base step*, *induct set*: *rtrancl-on*]:
assumes (*a*, *b*) ∈ *rtrancl-on F r*
and *a* ∈ *F* ⟹ *P a*
⋀*y z*. ⟦(*a*, *y*) ∈ *rtrancl-on F r*; (*y*,*z*) ∈ *r*; *y* ∈ *F*; *z* ∈ *F*; *P y*⟧ ⟹ *P z*
shows *P b*
using *assms* **by** (*induct a b*) (*auto dest*: *in-rtrancl-on-in-F*)

lemma *rtrancl-on-trans*:
assumes (*a*,*b*) ∈ *rtrancl-on F r* (*b*,*c*) ∈ *rtrancl-on F r* **shows** (*a*,*c*) ∈ *rtrancl-on F r*
using *assms*(*2*,*1*)
by *induct* (*auto intro*: *rtrancl-on-into-rtrancl-on*)

lemma *converse-rtrancl-on-into-rtrancl-on*:
assumes (*a*,*b*) ∈ *r* (*b*, *c*) ∈ *rtrancl-on F r a* ∈ *F*

shows $(a,\ c) \in$ *rtrancl-on F r*
proof −
have $b \in F$ **using** ‹$(b,c) \in$ -› **by** (*rule in-rtrancl-on-in-F*)
show *?thesis*
apply (*rule rtrancl-on-trans*)
apply (*rule rtrancl-on-into-rtrancl-on*)
apply (*rule rtrancl-on-refl*)
by *fact*+
qed

lemma *rtrancl-on-converseI*:
assumes $(y,\ x) \in$ *rtrancl-on F r* **shows** $(x,\ y) \in$ *rtrancl-on F* (r^{-1})
using *assms*
proof *induct*
case (*step a b*)
then have $(b,b) \in$ *rtrancl-on F* (r^{-1}) $(b,a) \in r^{-1}$ **by** *auto*
then show *?case* **using** *step*
by (*metis rtrancl-on-trans rtrancl-on-into-rtrancl-on*)
qed *auto*

theorem *rtrancl-on-converseD*:
assumes $(y,\ x) \in$ *rtrancl-on F* (r^{-1}) **shows** $(x,\ y) \in$ *rtrancl-on F r*
using *assms* **by** − (*drule rtrancl-on-converseI*, *simp*)

lemma *converse-rtrancl-on-induct*[*consumes 1*, *case-names base step*, *induct set*: *rtrancl-on*]:
assumes *major*: $(a,\ b) \in$ *rtrancl-on F r*
and *cases*: $b \in F \Longrightarrow P\ b$
$\bigwedge x\ y.\ [\![(x,y) \in r;\ (y,b) \in$ *rtrancl-on F r*$;\ x \in F;\ y \in F;\ P\ y]\!] \Longrightarrow P\ x$
shows $P\ a$
using *rtrancl-on-converseI*[*OF major*] *cases*
by *induct* (*auto intro*: *rtrancl-on-converseD*)

lemma *converse-rtrancl-on-cases*:
assumes $(a,\ b) \in$ *rtrancl-on F r*
obtains (*base*) $a = b\ b \in F$
| (*step*) *c* **where** $(a,c) \in r\ (c,b) \in$ *rtrancl-on F r*
using *assms* **by** *induct auto*

lemma *rtrancl-on-sym*:
assumes *sym r* **shows** *sym* (*rtrancl-on F r*)
using *assms* **by** (*auto simp*: *sym-conv-converse-eq intro*: *symI dest*: *rtrancl-on-converseI*)

lemma *rtrancl-on-mono*:
assumes $s \subseteq r\ F \subseteq G\ (a,b) \in$ *rtrancl-on F s* **shows** $(a,b) \in$ *rtrancl-on G r*
using *assms*(*3*,*1*,*2*)
proof *induct*
case (*step x y*) **show** *?case*
using *step assms* **by** (*intro converse-rtrancl-on-into-rtrancl-on*[*OF - step*(*5*)])

auto
qed *auto*

lemma *rtrancl-consistent-rtrancl-on*:
assumes $(a,b) \in r^*$
and $a \in F$ $b \in F$
and *consistent*: $\bigwedge a\ b.\ [\![\ a \in F;\ (a,b) \in r\ ]\!] \Longrightarrow b \in F$
shows $(a,b) \in$ *rtrancl-on F r*
using *assms*(*1*−*3*)
proof (*induction rule*: *converse-rtrancl-induct*)
case (*step y z*) **then have** $z \in F$ **by** (*rule-tac consistent*) *simp*
with *step* **have** $(z,b) \in$ *rtrancl-on F r* **by** *simp*
with *step.prems* ‹$(y,z) \in r$› ‹$z \in F$› **show** *?case*
using *converse-rtrancl-on-into-rtrancl-on*
by *metis*
qed *simp*

lemma *rtrancl-on-rtranclI*:
$(a,b) \in$ *rtrancl-on F r* $\Longrightarrow (a,b) \in r^*$
by (*induct rule*: *rtrancl-on-induct*) *simp-all*

lemma *rtrancl-on-sub-rtrancl*:
rtrancl-on F r $\subseteq r\widehat{\ }*$
using *rtrancl-on-rtranclI*
by *auto*

end

theory *Stuff*
imports
Main
HOL−*Library.Extended-Real*

begin

2 Additional theorems for base libraries

This section contains lemmas unrelated to graph theory which might be interesting for the Isabelle distribution

lemma *ereal-Inf-finite-Min*:
fixes *S* :: *ereal set*
assumes *finite S* **and** $S \neq \{\}$
shows *Inf S* = *Min S*
using *assms*
by (*induct S rule*: *finite-ne-induct*) (*auto simp*: *min-absorb1*)

lemma *finite-INF-in*:
fixes $f :: 'a \Rightarrow ereal$
assumes *finite S*
assumes $S \neq \{\}$
shows $(INF\ s{\in}\ S.\ f\ s) \in f\ `\ S$
proof −
from *assms*
have *finite* $(f\ `\ S)$ $f\ `\ S \neq \{\}$ **by** *auto*
then show *Inf* $(f\ `\ S) \in f\ `\ S$
using *ereal-Inf-finite-Min* [*of* $f\ `\ S$] **by** *simp*
qed

lemma *not-mem-less-INF*:
fixes $f :: 'a \Rightarrow 'b :: complete\text{-}lattice$
assumes $f\ x < (INF\ s{\in}\ S.\ f\ s)$
assumes $x \in S$
shows *False*
using *assms* **by** (*metis INF-lower less-le-not-le*)

lemma *sym-diff*:
assumes *sym A sym B* **shows** *sym* $(A - B)$
using *assms* **by** (*auto simp*: *sym-def*)

2.1 List

lemmas *list-exhaust2* = *list.exhaust*[*case-product list.exhaust*]

lemma *list-exhaust-NSC*:
obtains (*Nil*) $xs = []$ | (*Single*) x **where** $xs = [x]$ | (*Cons-Cons*) $x\ y\ ys$ **where** $xs = x \#\ y \#\ ys$
by (*metis list.exhaust*)

lemma *tl-rev*:
tl (*rev p*) = *rev* (*butlast p*)
by (*induct p*) *auto*

lemma *butlast-rev*:
butlast (*rev p*) = *rev* (*tl p*)
by (*induct p*) *auto*

lemma *take-drop-take*:
take n xs @ *drop n* (*take m xs*) = *take* (*max n m*) *xs*
proof *cases*
assume $m < n$ **then show** *?thesis* **by** (*auto simp*: *max-def*)
next
assume $\neg m < n$
then have *take n xs* = *take n* (*take m xs*) **by** (*auto simp*: *min-def*)
then show *?thesis* **by** (*simp del*: *take-take add*: *max-def*)

qed

lemma *drop-take-drop*:
drop n (*take m xs*) @ *drop m xs* = *drop* (*min n m*) *xs*
proof *cases*
assume *A*: ¬*m* < *n*
then show *?thesis*
using *drop-append*[*of n take m xs drop m xs*]
by (*cases length xs* < *n*) (*auto simp*: *not-less min-def*)
qed (*auto simp*: *min-def*)

lemma *not-distinct-decomp-min-prefix*:
assumes ¬ *distinct ws*
shows ∃ *xs ys zs y*. *ws* = *xs* @ *y* # *ys* @ *y* # *zs* ∧ *distinct xs* ∧ *y* ∉ *set xs* ∧ *y* ∉ *set ys*
proof −
obtain *xs y ys* **where** *y* ∈ *set xs distinct xs ws* = *xs* @ *y* # *ys*
using *assms* **by** (*auto simp*: *not-distinct-conv-prefix*)
moreover then obtain *xs*′ *ys*′ **where** *xs* = *xs*′ @ *y* # *ys*′ **by** (*auto simp*: *in-set-conv-decomp*)
ultimately show *?thesis* **by** *auto*
qed

lemma *not-distinct-decomp-min-not-distinct*:
assumes ¬ *distinct ws*
shows ∃*xs y ys zs*. *ws* = *xs* @ *y* # *ys* @ *y* # *zs* ∧ *distinct* (*ys* @ [*y*])
using *assms*
proof (*induct ws*)
case (*Cons w ws*)
show *?case*
proof (*cases distinct ws*)
case *True*
then obtain *xs ys* **where** *ws* = *xs* @ *w* # *ys w* ∉ *set xs*
using *Cons.prems* **by** (*fastforce dest*: *split-list-first*)
then have *distinct* (*xs* @ [*w*]) *w* # *ws* = [] @ *w* # *xs* @ *w* # *ys*
using ‹*distinct ws*› **by** *auto*
then show *?thesis* **by** *blast*
next
case *False*
then obtain *xs y ys zs* **where** *ws* = *xs* @ *y* # *ys* @ *y* # *zs* ∧ *distinct* (*ys* @ [*y*])
using *Cons* **by** *auto*
then have *w* # *ws* = (*w* # *xs*) @ *y* # *ys* @ *y* # *zs* ∧ *distinct* (*ys* @ [*y*])
by *simp*
then show *?thesis* **by** *blast*
qed
qed *simp*

lemma *card-Ex-subset*:

$k \leq card\ M \Longrightarrow \exists N.\ N \subseteq M \wedge card\ N = k$
by (*induct rule*: *inc-induct*) (*auto simp*: *card-Suc-eq*)

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

3 NOMATCH simproc

The simplification procedure can be used to avoid simplification of terms of a certain form

definition *NOMATCH* :: $'a \Rightarrow 'a \Rightarrow bool$ **where** *NOMATCH val pat* $\equiv$ *True*
lemma *NOMATCH-cong*[*cong*]: *NOMATCH val pat* = *NOMATCH val pat* **by** (*rule refl*)

simproc-setup *NOMATCH* (*NOMATCH val pat*) = ‹*fn* - => *fn ctxt* => *fn ct* =>
  let
    val thy = *Proof-Context.theory-of ctxt*
    val dest-binop = *Term.dest-comb* #> *apfst* (*Term.dest-comb* #> *snd*)
    val m = *Pattern.matches thy* (*dest-binop* (*Thm.term-of ct*))
  in if m then NONE else SOME @{*thm NOMATCH-def*} *end*
›

This setup ensures that a rewrite rule of the form *NOMATCH val pat* $\Longrightarrow$ *t* is only applied, if the pattern *pat* does not match the value *val*.

end

theory *Digraph*
imports
  Main
  Rtrancl-On
  Stuff
begin

4 Digraphs

record ('*a*,'*b*) *pre-digraph* =
  verts :: $'a\ set$
  arcs :: $'b\ set$
  tail :: $'b \Rightarrow 'a$
  head :: $'b \Rightarrow 'a$

definition *arc-to-ends* :: ('*a*,'*b*) *pre-digraph* $\Rightarrow$ $'b \Rightarrow 'a \times 'a$ **where**
  arc-to-ends G e $\equiv$ (*tail G e*, *head G e*)

locale *pre-digraph* =

fixes $G :: ('a, 'b)$ *pre-digraph* (**structure**)

locale *wf-digraph* = *pre-digraph* +
assumes *tail-in-verts*[*simp*]: $e \in arcs\ G \Longrightarrow tail\ G\ e \in verts\ G$
assumes *head-in-verts*[*simp*]: $e \in arcs\ G \Longrightarrow head\ G\ e \in verts\ G$
begin

lemma *wf-digraph*: *wf-digraph* G **by** *intro-locales*

lemmas *wellformed* = *tail-in-verts head-in-verts*

end

definition *arcs-ends* :: $('a,'b)$ *pre-digraph* $\Rightarrow ('a \times 'a)$ *set* **where**
arcs-ends $G \equiv$ *arc-to-ends* G ‘ *arcs* G

definition *symmetric* :: $('a,'b)$ *pre-digraph* $\Rightarrow$ *bool* **where**
symmetric $G \equiv$ *sym* (*arcs-ends* G)

Matches "pseudo digraphs" from [1], except for allowing the null graph. For a discussion of that topic, see also [3].

locale *fin-digraph* = *wf-digraph* +
assumes *finite-verts*[*simp*]: *finite* (*verts* G)
and *finite-arcs*[*simp*]: *finite* (*arcs* G)

locale *loopfree-digraph* = *wf-digraph* +
assumes *no-loops*: $e \in arcs\ G \Longrightarrow tail\ G\ e \neq head\ G\ e$

locale *nomulti-digraph* = *wf-digraph* +
assumes *no-multi-arcs*: $\bigwedge e1\ e2.\ [\![e1 \in arcs\ G;\ e2 \in arcs\ G;$
$arc\text{-}to\text{-}ends\ G\ e1 = arc\text{-}to\text{-}ends\ G\ e2]\!] \Longrightarrow e1 = e2$

locale *sym-digraph* = *wf-digraph* +
assumes *sym-arcs*[*intro*]: *symmetric* G

locale *digraph* = *fin-digraph* + *loopfree-digraph* + *nomulti-digraph*

We model graphs as symmetric digraphs. This is fine for many purposes, but not for all. For example, the path a, b, a is considered to be a cycle in a digraph (and hence in a symmetric digraph), but not in an undirected graph.

locale *pseudo-graph* = *fin-digraph* + *sym-digraph*

locale *graph* = *digraph* + *pseudo-graph*

lemma (**in** *wf-digraph*) *fin-digraphI*[*intro*]:
assumes *finite* (*verts* G)
assumes *finite* (*arcs* G)
shows *fin-digraph* G

using *assms* **by** *unfold-locales*

lemma (**in** *wf-digraph*) *sym-digraphI*[*intro*]:
assumes *symmetric G*
shows *sym-digraph G*
using *assms* **by** *unfold-locales*

lemma (**in** *digraph*) *graphI*[*intro*]:
assumes *symmetric G*
shows *graph G*
using *assms* **by** *unfold-locales*

definition (**in** *wf-digraph*) *arc* :: *'b* ⇒ *'a* × *'a* ⇒ *bool* **where**
arc e uv ≡ *e* ∈ *arcs G* ∧ *tail G e* = *fst uv* ∧ *head G e* = *snd uv*

lemma (**in** *fin-digraph*) *fin-digraph*: *fin-digraph G*
by *unfold-locales*

lemma (**in** *nomulti-digraph*) *nomulti-digraph*: *nomulti-digraph G* **by** *unfold-locales*

lemma *arcs-ends-conv*: *arcs-ends G* = (λ*e*. (*tail G e*, *head G e*)) ‘ *arcs G*
by (*auto simp*: *arc-to-ends-def arcs-ends-def*)

lemma *symmetric-conv*: *symmetric G* ⟷ (∀ *e1* ∈ *arcs G*. ∃ *e2* ∈ *arcs G*. *tail G e1* = *head G e2* ∧ *head G e1* = *tail G e2*)
unfolding *symmetric-def arcs-ends-conv sym-def* **by** *auto*

lemma *arcs-ends-symmetric*:
assumes *symmetric G*
shows (*u*,*v*) ∈ *arcs-ends G* ⟹ (*v*,*u*) ∈ *arcs-ends G*
using *assms* **unfolding** *symmetric-def sym-def* **by** *auto*

lemma (**in** *nomulti-digraph*) *inj-on-arc-to-ends*:
inj-on (*arc-to-ends G*) (*arcs G*)
by (*rule inj-onI*) (*rule no-multi-arcs*)

4.1 Reachability

abbreviation *dominates* :: (*'a*,*'b*) *pre-digraph* ⇒ *'a* ⇒ *'a* ⇒ *bool* (- →1 - [*100*,*100*] *40*) **where**
dominates G u v ≡ (*u*,*v*) ∈ *arcs-ends G*

abbreviation *reachable1* :: (*'a*,*'b*) *pre-digraph* ⇒ *'a* ⇒ *'a* ⇒ *bool* (- →+1 - [*100*,*100*] *40*) **where**
reachable1 G u v ≡ (*u*,*v*) ∈ (*arcs-ends G*)^+

definition *reachable* :: ('*a*,'*b*) *pre-digraph* ⇒ '*a* ⇒ '*a* ⇒ *bool* (- →*$^*_1$ - [*100*,*100*] *40*) **where**
reachable G u v ≡ (*u*,*v*) ∈ *rtrancl-on* (*verts G*) (*arcs-ends G*)

lemma *reachableE*[*elim*]:
assumes *u* →$_G$ *v*
obtains *e* **where** *e* ∈ *arcs G tail G e* = *u head G e* = *v*
using *assms* **by** (*auto simp add*: *arcs-ends-conv*)

lemma (**in** *loopfree-digraph*) *adj-not-same*:
assumes *a* → *a* **shows** *False*
using *assms* **by** (*rule reachableE*) (*auto dest*: *no-loops*)

lemma *reachable-in-vertsE*:
assumes *u* →*$^*_G$ *v* **obtains** *u* ∈ *verts G v* ∈ *verts G*
using *assms* **unfolding** *reachable-def* **by** *induct auto*

lemma *symmetric-reachable*:
assumes *symmetric G v* →*$^*_G$ *w* **shows** *w* →*$^*_G$ *v*
proof −
have *sym* (*rtrancl-on* (*verts G*) (*arcs-ends G*))
using *assms* **by** (*auto simp add*: *symmetric-def dest*: *rtrancl-on-sym*)
then show *?thesis* **using** *assms* **unfolding** *reachable-def* **by** (*blast elim*: *symE*)
qed

lemma *reachable-rtranclI*:
u →*$^*_G$ *v* ⟹ (*u*, *v*) ∈ (*arcs-ends G*)*
unfolding *reachable-def* **by** (*rule rtrancl-on-rtranclI*)

context *wf-digraph* **begin**

lemma *adj-in-verts*:
assumes *u* →$_G$ *v* **shows** *u* ∈ *verts G v* ∈ *verts G*
using *assms* **unfolding** *arcs-ends-conv* **by** *auto*

lemma *dominatesI*: **assumes** *arc-to-ends G a* = (*u*,*v*) *a* ∈ *arcs G* **shows** *u* →$_G$ *v*
using *assms* **by** (*auto simp*: *arcs-ends-def intro*: *rev-image-eqI*)

lemma *reachable-refl* [*intro*!, *Pure.intro*!, *simp*]: *v* ∈ *verts G* ⟹ *v* →* *v*
unfolding *reachable-def* **by** *auto*

lemma *adj-reachable-trans*[*trans*]:
assumes *a* →$_G$ *b b* →*$^*_G$ *c* **shows** *a* →*$^*_G$ *c*
using *assms* **by** (*auto simp*: *reachable-def intro*: *converse-rtrancl-on-into-rtrancl-on adj-in-verts*)

lemma *reachable-adj-trans*[*trans*]:

assumes $a \rightarrow^*_G b$ $b \rightarrow_G c$ **shows** $a \rightarrow^*_G c$
using *assms* **by** (*auto simp*: *reachable-def intro*: *rtrancl-on-into-rtrancl-on adj-in-verts*)

lemma *reachable-adjI* [*intro*, *simp*]: $u \rightarrow v \Longrightarrow u \rightarrow^* v$
by (*auto intro*: *adj-reachable-trans adj-in-verts*)

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

lemma *reachable-induct*[*consumes 1*, *case-names base step*]:
assumes *major*: $u \rightarrow^*_G v$
and *cases*: $u \in$ *verts* $G \Longrightarrow P\ u$
$\bigwedge x\ y.\ [\![u \rightarrow^*_G x;\ x \rightarrow_G y;\ P\ x]\!] \Longrightarrow P\ y$
shows $P\ v$
using *assms* **unfolding** *reachable-def* **by** (*rule rtrancl-on-induct*) *auto*

lemma *converse-reachable-induct*[*consumes 1*, *case-names base step*, *induct pred*: *reachable*]:
assumes *major*: $u \rightarrow^*_G v$
and *cases*: $v \in$ *verts* $G \Longrightarrow P\ v$
$\bigwedge x\ y.\ [\![x \rightarrow_G y;\ y \rightarrow^*_G v;\ P\ y]\!] \Longrightarrow P\ x$
shows $P\ u$
using *assms* **unfolding** *reachable-def* **by** (*rule converse-rtrancl-on-induct*) *auto*

lemma (**in** *pre-digraph*) *converse-reachable-cases*:
assumes $u \rightarrow^*_G v$
obtains (*base*) $u = v$ $u \in$ *verts* G
| (*step*) w **where** $u \rightarrow_G w$ $w \rightarrow^*_G v$
using *assms* **unfolding** *reachable-def* **by** (*cases rule*: *converse-rtrancl-on-cases*) *auto*

lemma *reachable-in-verts*:
assumes $u \rightarrow^* v$ **shows** $u \in$ *verts* G $v \in$ *verts* G
using *assms* **by** *induct* (*simp-all add*: *adj-in-verts*)

lemma *reachable1-in-verts*:
assumes $u \rightarrow^+ v$ **shows** $u \in$ *verts* G $v \in$ *verts* G
using *assms*
by *induct* (*simp-all add*: *adj-in-verts*)

lemma *reachable1-reachable*[*intro*]:
$v \rightarrow^+ w \Longrightarrow v \rightarrow^* w$
unfolding *reachable-def*
by (*rule rtrancl-consistent-rtrancl-on*) (*simp-all add*: *reachable1-in-verts adj-in-verts*)

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

lemma *reachable-neq-reachable1*[*intro*]:

assumes *reach*: $v \rightarrow^* w$
and *neq*: $v \neq w$
shows $v \rightarrow^+ w$
proof −
from *reach* **have** $(v,w) \in (arcs\text{-}ends\ G)\widehat{}*$ **by** (*rule reachable-rtranclI*)
with *neq* **show** *?thesis* **by** (*auto dest*: *rtranclD*)
qed

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

lemma *reachable1-reachable-trans* [*trans*]:
$u \rightarrow^+ v \Longrightarrow v \rightarrow^* w \Longrightarrow u \rightarrow^+ w$
by (*metis trancl-trans reachable-neq-reachable1*)

lemma *reachable-reachable1-trans* [*trans*]:
$u \rightarrow^* v \Longrightarrow v \rightarrow^+ w \Longrightarrow u \rightarrow^+ w$
by (*metis trancl-trans reachable-neq-reachable1*)

lemma *reachable-conv*:
$u \rightarrow^* v \longleftrightarrow (u,v) \in (arcs\text{-}ends\ G)\widehat{}* \cap (verts\ G \times verts\ G)$
apply (*auto intro*: *reachable-in-verts*)
apply (*induct rule*: *rtrancl-induct*)
apply *auto*
done

lemma *reachable-conv′*:
assumes $u \in verts\ G$
shows $u \rightarrow^* v \longleftrightarrow (u,v) \in (arcs\text{-}ends\ G)^*$ (**is** *?L* = *?R*)
proof
assume *?R* **then show** *?L* **using** *assms* **by** *induct auto*
qed (*auto simp*: *reachable-conv*)

end

lemma (**in** *sym-digraph*) *symmetric-reachable′*:
assumes $v \rightarrow^*{}_G w$ **shows** $w \rightarrow^*{}_G v$
using *sym-arcs assms* **by** (*rule symmetric-reachable*)

4.2 Degrees of vertices

definition *in-arcs* :: ('*a*, '*b*) *pre-digraph* ⇒ '*a* ⇒ '*b set* **where**
in-arcs G v ≡ {*e* ∈ *arcs G*. *head G e* = *v*}

definition *out-arcs* :: ('*a*, '*b*) *pre-digraph* ⇒ '*a* ⇒ '*b set* **where**
out-arcs G v ≡ {*e* ∈ *arcs G*. *tail G e* = *v*}

definition *in-degree* :: ('*a*, '*b*) *pre-digraph* ⇒ '*a* ⇒ *nat* **where**
in-degree G v ≡ *card* (*in-arcs G v*)

definition *out-degree* :: ('*a*, '*b*) *pre-digraph* ⇒ '*a* ⇒ *nat* **where**
out-degree G v ≡ *card* (*out-arcs G v*)

lemma (**in** *fin-digraph*) *finite-in-arcs*[*intro*]:
finite (*in-arcs G v*)
unfolding *in-arcs-def* **by** *auto*

lemma (**in** *fin-digraph*) *finite-out-arcs*[*intro*]:
finite (*out-arcs G v*)
unfolding *out-arcs-def* **by** *auto*

lemma *in-in-arcs-conv*[*simp*]:
e ∈ *in-arcs G v* ⟷ *e* ∈ *arcs G* ∧ *head G e* = *v*
unfolding *in-arcs-def* **by** *auto*

lemma *in-out-arcs-conv*[*simp*]:
e ∈ *out-arcs G v* ⟷ *e* ∈ *arcs G* ∧ *tail G e* = *v*
unfolding *out-arcs-def* **by** *auto*

lemma *inout-arcs-arc-simps*[*simp*]:
assumes *e* ∈ *arcs G*
shows *tail G e* = *u* ⟹ *out-arcs G u* ∩ *insert e E* = *insert e* (*out-arcs G u* ∩ *E*)
tail G e ≠ *u* ⟹ *out-arcs G u* ∩ *insert e E* = *out-arcs G u* ∩ *E*
out-arcs G u ∩ {} = {}
head G e = *u* ⟹ *in-arcs G u* ∩ *insert e E* = *insert e* (*in-arcs G u* ∩ *E*)
head G e ≠ *u* ⟹ *in-arcs G u* ∩ *insert e E* = *in-arcs G u* ∩ *E*
in-arcs G u ∩ {} = {}
using *assms* **by** *auto*

lemma *in-arcs-int-arcs*[*simp*]: *in-arcs G u* ∩ *arcs G* = *in-arcs G u* **and**
out-arcs-int-arcs[*simp*]: *out-arcs G u* ∩ *arcs G* = *out-arcs G u*
by *auto*

lemma *in-arcs-in-arcs*: *x* ∈ *in-arcs G u* ⟹ *x* ∈ *arcs G*
and *out-arcs-in-arcs*: *x* ∈ *out-arcs G u* ⟹ *x* ∈ *arcs G*
by (*auto simp*: *in-arcs-def out-arcs-def*)

4.3 Graph operations

context *pre-digraph* **begin**

definition *add-arc* :: '*b* ⇒ ('*a*,'*b*) *pre-digraph* **where**
add-arc a = ⦇ *verts* = *verts G* ∪ {*tail G a*, *head G a*}, *arcs* = *insert a* (*arcs G*), *tail* = *tail G*, *head* = *head G* ⦈

definition *del-arc* :: '*b* ⇒ ('*a*,'*b*) *pre-digraph* **where**

del-arc a = (| *verts* = *verts G*, *arcs* = *arcs G* − {*a*}, *tail* = *tail G*, *head* = *head G* |)

definition *add-vert* :: *'a* ⇒ (*'a*,*'b*) *pre-digraph* **where**
add-vert v = (| *verts* = *insert v* (*verts G*), *arcs* = *arcs G*, *tail* = *tail G*, *head* = *head G* |)

definition *del-vert* :: *'a* ⇒ (*'a*,*'b*) *pre-digraph* **where**
del-vert v = (| *verts* = *verts G* − {*v*}, *arcs* = {*a* ∈ *arcs G*. *tail G a* ≠ *v* ∧ *head G a* ≠ *v*}, *tail* = *tail G*, *head* = *head G* |)

lemma
verts-add-arc: ⟦ *tail G a* ∈ *verts G*; *head G a* ∈ *verts G* ⟧ ⟹ *verts* (*add-arc a*) = *verts G* **and**
verts-add-arc-conv: *verts* (*add-arc a*) = *verts G* ∪ {*tail G a*, *head G a*} **and**
arcs-add-arc: *arcs* (*add-arc a*) = *insert a* (*arcs G*) **and**
tail-add-arc: *tail* (*add-arc a*) = *tail G* **and**
head-add-arc: *head* (*add-arc a*) = *head G*
by (*auto simp*: *add-arc-def*)

lemmas *add-arc-simps*[*simp*] = *verts-add-arc arcs-add-arc tail-add-arc head-add-arc*

lemma
verts-del-arc: *verts* (*del-arc a*) = *verts G* **and**
arcs-del-arc: *arcs* (*del-arc a*) = *arcs G* − {*a*} **and**
tail-del-arc: *tail* (*del-arc a*) = *tail G* **and**
head-del-arc: *head* (*del-arc a*) = *head G*
by (*auto simp*: *del-arc-def*)

lemmas *del-arc-simps*[*simp*] = *verts-del-arc arcs-del-arc tail-del-arc head-del-arc*

lemma
verts-add-vert: *verts* (*pre-digraph.add-vert G u*) = *insert u* (*verts G*) **and**
arcs-add-vert: *arcs* (*pre-digraph.add-vert G u*) = *arcs G* **and**
tail-add-vert: *tail* (*pre-digraph.add-vert G u*) = *tail G* **and**
head-add-vert: *head* (*pre-digraph.add-vert G u*) = *head G*
by (*auto simp*: *pre-digraph.add-vert-def*)

lemmas *add-vert-simps* = *verts-add-vert arcs-add-vert tail-add-vert head-add-vert*

lemma
verts-del-vert: *verts* (*pre-digraph.del-vert G u*) = *verts G* − {*u*} **and**
arcs-del-vert: *arcs* (*pre-digraph.del-vert G u*) = {*a* ∈ *arcs G*. *tail G a* ≠ *u* ∧ *head G a* ≠ *u*} **and**
tail-del-vert: *tail* (*pre-digraph.del-vert G u*) = *tail G* **and**
head-del-vert: *head* (*pre-digraph.del-vert G u*) = *head G* **and**
ends-del-vert: *arc-to-ends* (*pre-digraph.del-vert G u*) = *arc-to-ends G*
by (*auto simp*: *pre-digraph.del-vert-def arc-to-ends-def*)

lemmas *del-vert-simps* = *verts-del-vert arcs-del-vert tail-del-vert head-del-vert*

lemma *add-add-arc-collapse*[*simp*]: *pre-digraph.add-arc* (*add-arc a*) *a* = *add-arc a*
by (*auto simp*: *pre-digraph.add-arc-def*)

lemma *add-del-arc-collapse*[*simp*]: *pre-digraph.add-arc* (*del-arc a*) *a* = *add-arc a*
by (*auto simp*: *pre-digraph.verts-add-arc-conv pre-digraph.add-arc-simps*)

lemma *del-add-arc-collapse*[*simp*]:
⟦ *tail G a* ∈ *verts G*; *head G a* ∈ *verts G* ⟧ ⟹ *pre-digraph.del-arc* (*add-arc a*) *a* = *del-arc a*
by (*auto simp*: *pre-digraph.add-arc-simps pre-digraph.del-arc-simps*)

lemma *del-del-arc-collapse*[*simp*]: *pre-digraph.del-arc* (*del-arc a*) *a* = *del-arc a*
by (*auto simp*: *pre-digraph.add-arc-simps pre-digraph.del-arc-simps*)

lemma *add-arc-commute*: *pre-digraph.add-arc* (*add-arc b*) *a* = *pre-digraph.add-arc* (*add-arc a*) *b*
by (*auto simp*: *pre-digraph.add-arc-def*)

lemma *del-arc-commute*: *pre-digraph.del-arc* (*del-arc b*) *a* = *pre-digraph.del-arc* (*del-arc a*) *b*
by (*auto simp*: *pre-digraph.del-arc-def*)

lemma *del-arc-in*: *a* ∉ *arcs G* ⟹ *del-arc a* = *G*
by (*rule pre-digraph.equality*) (*auto simp*: *add-arc-def*)

lemma *in-arcs-add-arc-iff*:
in-arcs (*add-arc a*) *u* = (*if head G a* = *u then insert a* (*in-arcs G u*) *else in-arcs G u*)
by *auto*

lemma *out-arcs-add-arc-iff*:
out-arcs (*add-arc a*) *u* = (*if tail G a* = *u then insert a* (*out-arcs G u*) *else out-arcs G u*)
by *auto*

lemma *in-arcs-del-arc-iff*:
in-arcs (*del-arc a*) *u* = (*if head G a* = *u then in-arcs G u* − {*a*} *else in-arcs G u*)
by *auto*

lemma *out-arcs-del-arc-iff*:
out-arcs (*del-arc a*) *u* = (*if tail G a* = *u then out-arcs G u* − {*a*} *else out-arcs G u*)
by *auto*

lemma (**in** *wf-digraph*) *add-arc-in*: *a* ∈ *arcs G* ⟹ *add-arc a* = *G*
by (*rule pre-digraph.equality*) (*auto simp*: *add-arc-def*)

end

context *wf-digraph* **begin**

lemma *wf-digraph-add-arc*[*intro*]:
wf-digraph (*add-arc a*) **by** *unfold-locales* (*auto simp*: *verts-add-arc-conv*)

lemma *wf-digraph-del-arc*[*intro*]:
wf-digraph (*del-arc a*) **by** *unfold-locales* (*auto simp*: *verts-add-arc-conv*)

lemma *wf-digraph-del-vert*: *wf-digraph* (*del-vert u*)
by *standard* (*auto simp*: *del-vert-simps*)

lemma *wf-digraph-add-vert*: *wf-digraph* (*add-vert u*)
by *standard* (*auto simp*: *add-vert-simps*)

lemma *del-vert-add-vert*:
assumes $u \notin$ *verts G*
shows *pre-digraph.del-vert* (*add-vert u*) *u* = *G*
using *assms* **by** (*intro pre-digraph.equality*) (*auto simp*: *pre-digraph.del-vert-def add-vert-def*)

end

context *fin-digraph* **begin**

lemma *in-degree-add-arc-iff*:
in-degree (*add-arc a*) *u* = (*if head G a* = *u* ∧ *a* ∉ *arcs G then in-degree G u* + *1 else in-degree G u*)
proof −
have *a* ∉ *arcs G* ⟹ *a* ∉ *in-arcs G u* **by** (*auto simp*: *in-arcs-def*)
with *finite-in-arcs* **show** *?thesis*
unfolding *in-degree-def* **by** (*auto simp*: *in-arcs-add-arc-iff intro*: *arg-cong*[**where** *f*=*card*])
qed

lemma *out-degree-add-arc-iff*:
out-degree (*add-arc a*) *u* = (*if tail G a* = *u* ∧ *a* ∉ *arcs G then out-degree G u* + *1 else out-degree G u*)
proof −
have *a* ∉ *arcs G* ⟹ *a* ∉ *out-arcs G u* **by** (*auto simp*: *out-arcs-def*)
with *finite-out-arcs* **show** *?thesis*
unfolding *out-degree-def* **by** (*auto simp*: *out-arcs-add-arc-iff intro*: *arg-cong*[**where** *f*=*card*])
qed

lemma *in-degree-del-arc-iff*:
in-degree (*del-arc a*) *u* = (*if head G a* = *u* ∧ *a* ∈ *arcs G then in-degree G u* − *1 else in-degree G u*)
proof −
have *a* ∉ *arcs G* ⟹ *a* ∉ *in-arcs G u* **by** (*auto simp*: *in-arcs-def*)
with *finite-in-arcs* **show** *?thesis*
unfolding *in-degree-def* **by** (*auto simp*: *in-arcs-del-arc-iff intro*: *arg-cong*[**where** *f*=*card*])
qed

lemma *out-degree-del-arc-iff*:
out-degree (*del-arc a*) *u* = (*if tail G a* = *u* ∧ *a* ∈ *arcs G then out-degree G u* − *1 else out-degree G u*)
proof −
have *a* ∉ *arcs G* ⟹ *a* ∉ *out-arcs G u* **by** (*auto simp*: *out-arcs-def*)
with *finite-out-arcs* **show** *?thesis*
unfolding *out-degree-def* **by** (*auto simp*: *out-arcs-del-arc-iff intro*: *arg-cong*[**where** *f*=*card*])
qed

lemma *fin-digraph-del-vert*: *fin-digraph* (*del-vert u*)
by *standard* (*auto simp*: *del-vert-simps*)

lemma *fin-digraph-del-arc*: *fin-digraph* (*del-arc a*)
by *standard* (*auto simp*: *del-vert-simps*)

end

end
theory *Bidirected-Digraph*
imports
Digraph
HOL−*Combinatorics.Permutations*
begin

5 Bidirected Graphs

locale *bidirected-digraph* = *wf-digraph G* **for** *G* +
fixes *arev* :: *'b* ⇒ *'b*
assumes *arev-dom*: ⋀*a*. *a* ∈ *arcs G* ⟷ *arev a* ≠ *a*
assumes *arev-arev-raw*: ⋀*a*. *a* ∈ *arcs G* ⟹ *arev* (*arev a*) = *a*
assumes *tail-arev*[*simp*]: ⋀*a*. *a* ∈ *arcs G* ⟹ *tail G* (*arev a*) = *head G a*

lemma (**in** *wf-digraph*) *bidirected-digraphI*:
assumes *arev-eq*: ⋀*a*. *a* ∉ *arcs G* ⟹ *arev a* = *a*
assumes *arev-neq*: ⋀*a*. *a* ∈ *arcs G* ⟹ *arev a* ≠ *a*
assumes *arev-arev-raw*: ⋀*a*. *a* ∈ *arcs G* ⟹ *arev* (*arev a*) = *a*
assumes *tail-arev*: ⋀*a*. *a* ∈ *arcs G* ⟹ *tail G* (*arev a*) = *head G a*

shows *bidirected-digraph G arev*
using *assms* **by** *unfold-locales* (*auto simp*: *permutes-def*)

context *bidirected-digraph* **begin**

lemma *bidirected-digraph*[*intro*!]: *bidirected-digraph G arev*
by *unfold-locales*

lemma *arev-arev*[*simp*]: *arev* (*arev a*) = *a*
using *arev-dom* **by** (*cases a* ∈ *arcs G*) (*auto simp*: *arev-arev-raw*)

lemma *arev-o-arev*[*simp*]: *arev o arev* = *id*
by (*simp add*: *fun-eq-iff*)

lemma *arev-eq*: *a* ∉ *arcs G* ⟹ *arev a* = *a*
by (*simp add*: *arev-dom*)

lemma *arev-neq*: *a* ∈ *arcs G* ⟹ *arev a* ≠ *a*
by (*simp add*: *arev-dom*)

lemma *arev-in-arcs*[*simp*]: *a* ∈ *arcs G* ⟹ *arev a* ∈ *arcs G*
by (*metis arev-arev arev-dom*)

lemma *head-arev*[*simp*]:
assumes *a* ∈ *arcs G* **shows** *head G* (*arev a*) = *tail G a*
proof −
from *assms* **have** *head G* (*arev a*) = *tail G* (*arev* (*arev a*))
by (*simp only*: *tail-arev arev-in-arcs*)
then show *?thesis* **by** *simp*
qed

lemma *ate-arev*[*simp*]:
assumes *a* ∈ *arcs G* **shows** *arc-to-ends G* (*arev a*) = *prod.swap* (*arc-to-ends G a*)
using *assms* **by** (*auto simp*: *arc-to-ends-def*)

lemma *bij-arev*: *bij arev*
using *arev-arev* **by** (*metis bij-betw-imageI inj-on-inverseI surjI*)

lemma *arev-permutes-arcs*: *arev permutes arcs G*
using *arev-dom bij-arev* **by** (*auto simp*: *permutes-def bij-iff*)

lemma *arev-eq-iff*: ⋀*x y*. *arev x* = *arev y* ⟷ *x* = *y*
by (*metis arev-arev*)

lemma *in-arcs-eq*: *in-arcs G w* = *arev* ‘ *out-arcs G w*
by *auto* (*metis arev-arev arev-in-arcs image-eqI in-out-arcs-conv tail-arev*)

lemma *inj-on-arev*[*intro*!]: *inj-on arev S*

```
    by (metis arev-arev inj-on-inverseI)

  lemma even-card-loops:
    even (card (in-arcs G w ∩ out-arcs G w)) (is even (card ?S))
  proof −
    { assume ¬finite ?S
      then have ?thesis by simp
    }
    moreover
    { assume A:finite ?S
      have card ?S = card (⋃{{a,arev a} | a. a ∈ ?S}) (is - = card (⋃ ?T))
        by (rule arg-cong[where f=card]) (auto intro!: exI[where x={x, arev x}
for x])
      also have ...= sum card ?T
      proof (rule card-Union-disjoint)
        show ⋀A. A∈{{a, arev a} |a. a ∈ ?S} ⟹ finite A by auto
        show pairwise disjnt {{a, arev a} |a. a ∈ in-arcs G w ∩ out-arcs G w}
          unfolding pairwise-def disjnt-def
           by safe (simp-all add: arev-eq-iff)
      qed
      also have ... = sum (λa. 2) ?T
        by (intro sum.cong) (auto simp: card-insert-if dest: arev-neq)
      also have ... = 2 ∗ card ?T by simp
      finally have ?thesis by simp
    }
    ultimately
    show ?thesis by blast
  qed

end


sublocale bidirected-digraph ⊆ sym-digraph
proof (unfold-locales, unfold symmetric-def, intro symI)
  fix u v assume u →_G v
   then obtain a where a ∈ arcs G arc-to-ends G a = (u,v) by (auto simp:
arcs-ends-def)
  then have arev a ∈ arcs G arc-to-ends G (arev a) = (v,u)
    by (auto simp: arc-to-ends-def)
  then show v →_G u by (auto simp: arcs-ends-def intro: rev-image-eqI)
qed



end


theory Arc-Walk
imports
```

Digraph
begin

6 Arc Walks

We represent a walk in a graph by the list of its arcs.

type-synonym *'b awalk = 'b list*

context *pre-digraph* **begin**

The list of vertices of a walk. The additional vertex argument is there to deal with the case of empty walks.

primrec *awalk-verts* :: *'a ⇒ 'b awalk ⇒ 'a list* **where**
  awalk-verts u [] = [u]
| *awalk-verts u (e # es) = tail G e # awalk-verts (head G e) es*

abbreviation *awhd* :: *'a ⇒ 'b awalk ⇒ 'a* **where**
  awhd u p ≡ hd (awalk-verts u p)

abbreviation *awlast*:: *'a ⇒ 'b awalk ⇒ 'a* **where**
  awlast u p ≡ last (awalk-verts u p)

Tests whether a list of arcs is a consistent arc sequence, i.e. a list of arcs, where the head G node of each arc is the tail G node of the following arc.

fun *cas* :: *'a ⇒ 'b awalk ⇒ 'a ⇒ bool* **where**
  cas u [] v = (u = v) |
  cas u (e # es) v = (tail G e = u ∧ cas (head G e) es v)

lemma *cas-simp*:
  assumes *es ≠ []*
  shows *cas u es v ⟷ tail G (hd es) = u ∧ cas (head G (hd es)) (tl es) v*
using *assms* **by** (*cases es*) *auto*

definition *awalk* :: *'a ⇒ 'b awalk ⇒ 'a ⇒ bool* **where**
  awalk u p v ≡ u ∈ verts G ∧ set p ⊆ arcs G ∧ cas u p v

definition (**in** *pre-digraph*) *trail* :: *'a ⇒ 'b awalk ⇒ 'a ⇒ bool* **where**
  trail u p v ≡ awalk u p v ∧ distinct p

definition *apath* :: *'a ⇒'b awalk ⇒ 'a ⇒ bool* **where**
  apath u p v ≡ awalk u p v ∧ distinct (awalk-verts u p)

end

6.1 Basic Lemmas

lemma (**in** *pre-digraph*) *awalk-verts-conv*:

awalk-verts u p = (*if p* = [] *then* [*u*] *else map* (*tail G*) *p* @ [*head G* (*last p*)])
by (*induct p arbitrary*: *u*) *auto*

lemma (**in** *pre-digraph*) *awalk-verts-conv'*:
assumes *cas u p v*
shows *awalk-verts u p* = (*if p* = [] *then* [*u*] *else tail G* (*hd p*) # *map* (*head G*) *p*)
using *assms* **by** (*induct u p v rule*: *cas.induct*) (*auto simp*: *cas-simp*)

lemma (**in** *pre-digraph*) *length-awalk-verts*:
length (*awalk-verts u p*) = *Suc* (*length p*)
by (*simp add*: *awalk-verts-conv*)

lemma (**in** *pre-digraph*) *awalk-verts-ne-eq*:
assumes *p* ≠ []
shows *awalk-verts u p* = *awalk-verts v p*
using *assms* **by** (*auto simp*: *awalk-verts-conv*)

lemma (**in** *pre-digraph*) *awalk-verts-non-Nil*[*simp*]:
awalk-verts u p ≠ []
by (*simp add*: *awalk-verts-conv*)

context *wf-digraph* **begin**

lemma
assumes *cas u p v*
shows *awhd-if-cas*: *awhd u p* = *u* **and** *awlast-if-cas*: *awlast u p* = *v*
using *assms* **by** (*induct p arbitrary*: *u*) *auto*

lemma *awalk-verts-in-verts*:
assumes *u* ∈ *verts G set p* ⊆ *arcs G v* ∈ *set* (*awalk-verts u p*)
shows *v* ∈ *verts G*
using *assms* **by** (*induct p arbitrary*: *u*) (*auto intro*: *wellformed*)

lemma
assumes *u* ∈ *verts G set p* ⊆ *arcs G*
shows *awhd-in-verts*: *awhd u p* ∈ *verts G*
and *awlast-in-verts*: *awlast u p* ∈ *verts G*
using *assms* **by** (*auto elim*: *awalk-verts-in-verts*)

lemma *awalk-conv*:
awalk u p v = (*set* (*awalk-verts u p*) ⊆ *verts G*
∧ *set p* ⊆ *arcs G*
∧ *awhd u p* = *u* ∧ *awlast u p* = *v* ∧ *cas u p v*)
unfolding *awalk-def* **using** *hd-in-set*[*OF awalk-verts-non-Nil*, *of u p*]
by (*auto intro*: *awalk-verts-in-verts awhd-if-cas awlast-if-cas simp del*: *hd-in-set*)

lemma *awalkI*:
assumes *set* (*awalk-verts u p*) ⊆ *verts G set p* ⊆ *arcs G cas u p v*
shows *awalk u p v*

using *assms* **by** (*auto simp*: *awalk-conv awhd-if-cas awlast-if-cas*)

lemma *awalkE*[*elim*]:
assumes *awalk u p v*
obtains *set* (*awalk-verts u p*) ⊆ *verts G set p* ⊆ *arcs G cas u p v*
awhd u p = *u awlast u p* = *v*
using *assms* **by** (*auto simp add*: *awalk-conv*)

lemma *awalk-Nil-iff*:
awalk u [] *v* ⟷ *u* = *v* ∧ *u* ∈ *verts G*
unfolding *awalk-def* **by** *auto*

lemma *trail-Nil-iff*:
trail u [] *v* ⟷ *u* = *v* ∧ *u* ∈ *verts G*
by (*auto simp*: *trail-def awalk-Nil-iff*)

lemma *apath-Nil-iff*: *apath u* [] *v* ⟷ *u* = *v* ∧ *u* ∈ *verts G*
by (*auto simp*: *apath-def awalk-Nil-iff*)

lemma *awalk-hd-in-verts*: *awalk u p v* ⟹ *u* ∈ *verts G*
by (*cases p*) *auto*

lemma *awalk-last-in-verts*: *awalk u p v* ⟹ *v* ∈ *verts G*
unfolding *awalk-conv* **by** *auto*

lemma *hd-in-awalk-verts*:
awalk u p v ⟹ *u* ∈ *set* (*awalk-verts u p*)
apath u p v ⟹ *u* ∈ *set* (*awalk-verts u p*)
by (*case-tac* [!]*p*) (*auto simp*: *apath-def*)

lemma *awalk-Cons-iff*:
awalk u (*e* # *es*) *w* ⟷ *e* ∈ *arcs G* ∧ *u* = *tail G e* ∧ *awalk* (*head G e*) *es w*
by (*auto simp*: *awalk-def*)

lemma *trail-Cons-iff*:
trail u (*e* # *es*) *w* ⟷ *e* ∈ *arcs G* ∧ *u* = *tail G e* ∧ *e* ∉ *set es* ∧ *trail* (*head G e*) *es w*
by (*auto simp*: *trail-def awalk-Cons-iff*)

lemma *apath-Cons-iff*:
apath u (*e* # *es*) *w* ⟷ *e* ∈ *arcs G* ∧ *tail G e* = *u* ∧ *apath* (*head G e*) *es w*
∧ *tail G e* ∉ *set* (*awalk-verts* (*head G e*) *es*) (**is** *?L* ⟷ *?R*)
by (*auto simp*: *apath-def awalk-Cons-iff*)

lemmas *awalk-simps* = *awalk-Nil-iff awalk-Cons-iff*
lemmas *trail-simps* = *trail-Nil-iff trail-Cons-iff*
lemmas *apath-simps* = *apath-Nil-iff apath-Cons-iff*

lemma *arc-implies-awalk*:

$e \in arcs\ G \Longrightarrow awalk\ (tail\ G\ e)\ [e]\ (head\ G\ e)$
by (*simp add*: *awalk-simps*)

lemma *apath-nonempty-ends*:
assumes *apath u p v*
assumes $p \neq []$
shows $u \neq v$
using *assms*
proof (*induct p arbitrary*: *u*)
case (*Cons e es*)
then have *apath* (*head G e*) *es v u* $\notin$ *set* (*awalk-verts* (*head G e*) *es*)
by (*auto simp*: *apath-Cons-iff*)
moreover then have $v \in$ *set* (*awalk-verts* (*head G e*) *es*) **by** (*auto simp*: *apath-def*)
ultimately show $u \neq v$ **by** *auto*
qed *simp*

lemma *awalk-ConsI*:
assumes *awalk v es w*
assumes $e \in arcs\ G$ **and** *arc-to-ends G e* = (*u*,*v*)
shows *awalk u* (*e* # *es*) *w*
using *assms* **by** (*cases es*) (*auto simp*: *awalk-def arc-to-ends-def*)

lemma (**in** *pre-digraph*) *awalkI-apath*:
assumes *apath u p v* **shows** *awalk u p v*
using *assms* **by** (*simp add*: *apath-def*)

lemma *arcE*:
assumes *arc e* (*u*,*v*)
assumes $[\![e \in arcs\ G; tail\ G\ e = u; head\ G\ e = v]\!] \Longrightarrow P$
shows *P*
using *assms* **by** (*auto simp*: *arc-def*)

lemma *in-arcs-imp-in-arcs-ends*:
assumes $e \in arcs\ G$
shows (*tail G e*, *head G e*) $\in$ *arcs-ends G*
using *assms* **by** (*auto simp*: *arcs-ends-conv*)

lemma *set-awalk-verts-cas*:
assumes *cas u p v*
shows *set* (*awalk-verts u p*) = $\{u\} \cup$ *set* (*map* (*tail G*) *p*) $\cup$ *set* (*map* (*head G*) *p*)
using *assms*
proof (*induct p arbitrary*: *u*)
case *Nil* **then show** *?case* **by** *simp*
next

case (*Cons e es*)
then have *set* (*awalk-verts* (*head G e*) *es*)
= {*head G e*} ∪ *set* (*map* (*tail G*) *es*) ∪ *set* (*map* (*head G*) *es*)
by (*auto simp*: *awalk-Cons-iff*)
with *Cons.prems* **show** *?case* **by** *auto*
qed

lemma *set-awalk-verts-not-Nil-cas*:
assumes *cas u p v p* ≠ []
shows *set* (*awalk-verts u p*) = *set* (*map* (*tail G*) *p*) ∪ *set* (*map* (*head G*) *p*)
proof −
have *u* ∈ *set* (*map* (*tail G*) *p*) **using** *assms* **by** (*cases p*) *auto*
with *assms* **show** *?thesis* **by** (*auto simp*: *set-awalk-verts-cas*)
qed

lemma *set-awalk-verts*:
assumes *awalk u p v*
shows *set* (*awalk-verts u p*) = {*u*} ∪ *set* (*map* (*tail G*) *p*) ∪ *set* (*map* (*head G*) *p*)
using *assms* **by** (*intro set-awalk-verts-cas*) *blast*

lemma *set-awalk-verts-not-Nil*:
assumes *awalk u p v p* ≠ []
shows *set* (*awalk-verts u p*) = *set* (*map* (*tail G*) *p*) ∪ *set* (*map* (*head G*) *p*)
using *assms* **by** (*intro set-awalk-verts-not-Nil-cas*) *blast*

lemma
awhd-of-awalk: *awalk u p v* ⟹ *awhd u p* = *u* **and**
awlast-of-awalk: *awalk u p v* ⟹ *NOMATCH* (*awlast u p*) *v* ⟹ *awlast u p* = *v*
unfolding *NOMATCH-def* **by** *auto*
lemmas *awends-of-awalk*[*simp*] = *awhd-of-awalk awlast-of-awalk*

lemma *awalk-verts-arc1*:
assumes *e* ∈ *set p*
shows *tail G e* ∈ *set* (*awalk-verts u p*)
using *assms* **by** (*auto simp*: *awalk-verts-conv*)

lemma *awalk-verts-arc2*:
assumes *awalk u p v e* ∈ *set p*
shows *head G e* ∈ *set* (*awalk-verts u p*)
using *assms* **by** (*simp add*: *set-awalk-verts*)

lemma *awalk-induct-raw*[*case-names Base Cons*]:
assumes *awalk u p v*
assumes ⋀*w1*. *w1* ∈ *verts G* ⟹ *P w1* [] *w1*
assumes ⋀*w1 w2 e es*. *e* ∈ *arcs G* ⟹ *arc-to-ends G e* = (*w1*, *w2*)
⟹ *P w2 es v* ⟹ *P w1* (*e* # *es*) *v*
shows *P u p v*
using *assms*

proof (*induct p arbitrary*: *u v*)
case *Nil* **then show** *?case* **using** *Nil.prems* **by** *auto*
next
case (*Cons e es*)
from *Cons.prems*(*1*) **show** *?case*
by (*intro Cons*) (*auto intro*: *Cons*(*2*−) *simp*: *arc-to-ends-def awalk-Cons-iff*)
qed

6.2 Appending awalks

lemma (**in** *pre-digraph*) *cas-append-iff*[*simp*]:
cas u (*p* @ *q*) *v* ⟷ *cas u p* (*awlast u p*) ∧ *cas* (*awlast u p*) *q v*
by (*induct u p v rule*: *cas.induct*) *auto*

lemma *cas-ends*:
assumes *cas u p v cas u′ p v′*
shows (*p* ≠ [] ∧ *u* = *u′* ∧ *v* = *v′*) ∨ (*p* = [] ∧ *u* = *v* ∧ *u′* = *v′*)
using *assms* **by** (*induct u p v arbitrary*: *u u′ rule*: *cas.induct*) *auto*

lemma *awalk-ends*:
assumes *awalk u p v awalk u′ p v′*
shows (*p* ≠ [] ∧ *u* = *u′* ∧ *v* = *v′*) ∨ (*p* = [] ∧ *u* = *v* ∧ *u′* = *v′*)
using *assms* **by** (*simp add*: *awalk-def cas-ends*)

lemma *awalk-ends-eqD*:
assumes *awalk u p u awalk v p w*
shows *v* = *w*
using *awalk-ends*[*OF assms*(*1*,*2*)] **by** *auto*

lemma *awalk-empty-ends*:
assumes *awalk u* [] *v*
shows *u* = *v*
using *assms* **by** (*auto simp*: *awalk-def*)

lemma *apath-ends*:
assumes *apath u p v* **and** *apath u′ p v′*
shows (*p* ≠ [] ∧ *u* ≠ *v* ∧ *u* = *u′* ∧ *v* = *v′*) ∨ (*p* = [] ∧ *u* = *v* ∧ *u′* = *v′*)
using *assms* **unfolding** *apath-def* **by** (*metis assms*(*2*) *apath-nonempty-ends awalk-ends*)

lemma *awalk-append-iff*[*simp*]:
awalk u (*p* @ *q*) *v* ⟷ *awalk u p* (*awlast u p*) ∧ *awalk* (*awlast u p*) *q v* (**is** *?L*
⟷ *?R*)
by (*auto simp*: *awalk-def intro*: *awlast-in-verts*)

lemma *awlast-append*:
awlast u (*p* @ *q*) = *awlast* (*awlast u p*) *q*
by (*simp add*: *awalk-verts-conv*)

lemma *awhd-append*:

awhd u (p @ q) = awhd (awhd u q) p
by (*simp add*: *awalk-verts-conv*)

declare *awalkE*[*rule del*]

lemma *awalkE'*[*elim*]:
assumes *awalk u p v*
obtains *set* (*awalk-verts u p*) ⊆ *verts G set p* ⊆ *arcs G cas u p v*
awhd u p = u awlast u p = v u ∈ *verts G v* ∈ *verts G*
proof −
have *u* ∈ *set* (*awalk-verts u p*) *v* ∈ *set* (*awalk-verts u p*)
using *assms* **by** (*auto simp*: *hd-in-awalk-verts elim*: *awalkE*)
then show *?thesis* **using** *assms* **by** (*auto elim*: *awalkE intro*: *that*)
qed

lemma *awalk-appendI*:
assumes *awalk u p v*
assumes *awalk v q w*
shows *awalk u* (*p* @ *q*) *w*
using *assms*
proof (*induct p arbitrary*: *u*)
case *Nil* **then show** *?case* **by** *auto*
next
case (*Cons e es*)
from *Cons.prems* **have** *ee-e*: *arc-to-ends G e* = (*u*, *head G e*)
unfolding *arc-to-ends-def* **by** *auto*

have *awalk* (*head G e*) *es v*
using *ee-e Cons*(*2*) *awalk-Cons-iff* **by** *auto*
then show *?case* **using** *Cons ee-e* **by** (*auto simp*: *awalk-Cons-iff*)
qed

lemma *awalk-verts-append-cas*:
assumes *cas u* (*p* @ *q*) *v*
shows *awalk-verts u* (*p* @ *q*) = *awalk-verts u p* @ *tl* (*awalk-verts* (*awlast u p*) *q*)
using *assms*
proof (*induct p arbitrary*: *u*)
case *Nil* **then show** *?case* **by** (*cases q*) *auto*
qed (*auto simp*: *awalk-Cons-iff*)

lemma *awalk-verts-append*:
assumes *awalk u* (*p* @ *q*) *v*
shows *awalk-verts u* (*p* @ *q*) = *awalk-verts u p* @ *tl* (*awalk-verts* (*awlast u p*) *q*)
using *assms* **by** (*intro awalk-verts-append-cas*) *blast*

lemma *awalk-verts-append2*:
assumes *awalk u* (*p* @ *q*) *v*
shows *awalk-verts u* (*p* @ *q*) = *butlast* (*awalk-verts u p*) @ *awalk-verts* (*awlast u p*) *q*

using *assms* **by** (*auto simp*: *awalk-verts-conv*)

lemma *apath-append-iff*:
apath u (*p* @ *q*) *v* ⟷ *apath u p* (*awlast u p*) ∧ *apath* (*awlast u p*) *q v* ∧
set (*awalk-verts u p*) ∩ *set* (*tl* (*awalk-verts* (*awlast u p*) *q*)) = {} (**is** *?L* ⟷ *?R*)
proof
assume *?L*
then have *distinct* (*awalk-verts* (*awlast u p*) *q*) **by** (*auto simp*: *apath-def awalk-verts-append2*)
with ‹*?L*› **show** *?R* **by** (*auto simp*: *apath-def awalk-verts-append*)
next
assume *?R*
then show *?L* **by** (*auto simp*: *apath-def awalk-verts-append dest*: *distinct-tl*)
qed

lemma (**in** *wf-digraph*) *set-awalk-verts-append-cas*:
assumes *cas u p v cas v q w*
shows *set* (*awalk-verts u* (*p* @ *q*)) = *set* (*awalk-verts u p*) ∪ *set* (*awalk-verts v q*)
proof −
from *assms* **have** *cas-pq*: *cas u* (*p* @ *q*) *w*
by (*simp add*: *awlast-if-cas*)
moreover
from *assms* **have** *v* ∈ *set* (*awalk-verts u p*)
by (*metis awalk-verts-non-Nil awlast-if-cas last-in-set*)
ultimately show *?thesis* **using** *assms*
by (*auto simp*: *set-awalk-verts-cas*)
qed

lemma (**in** *wf-digraph*) *set-awalk-verts-append*:
assumes *awalk u p v awalk v q w*
shows *set* (*awalk-verts u* (*p* @ *q*)) = *set* (*awalk-verts u p*) ∪ *set* (*awalk-verts v q*)
proof −
from *assms* **have** *awalk u* (*p* @ *q*) *w* **by** *auto*
moreover
with *assms* **have** *v* ∈ *set* (*awalk-verts u* (*p* @ *q*))
by (*auto simp*: *awalk-verts-append*)
ultimately show *?thesis* **using** *assms*
by (*auto simp*: *set-awalk-verts*)
qed

lemma *cas-takeI*:
assumes *cas u p v awlast u* (*take n p*) = *v′*
shows *cas u* (*take n p*) *v′*
proof −
from *assms* **have** *cas u* (*take n p* @ *drop n p*) *v* **by** *simp*
with *assms* **show** *?thesis* **unfolding** *cas-append-iff* **by** *simp*
qed

lemma *cas-dropI*:
assumes *cas u p v awlast u* (*take n p*) = *u′*
shows *cas u′* (*drop n p*) *v*
proof −
from *assms* **have** *cas u* (*take n p* @ *drop n p*) *v* **by** *simp*
with *assms* **show** *?thesis* **unfolding** *cas-append-iff* **by** *simp*
qed

lemma *awalk-verts-take-conv*:
assumes *cas u p v*
shows *awalk-verts u* (*take n p*) = *take* (*Suc n*) (*awalk-verts u p*)
proof −
from *assms* **have** *cas u* (*take n p*) (*awlast u* (*take n p*)) **by** (*auto intro*: *cas-takeI*)
with *assms* **show** *?thesis*
by (*cases n p rule*: *nat.exhaust*[*case-product list.exhaust*])
(*auto simp*: *awalk-verts-conv′ take-map simp del*: *awalk-verts.simps*)
qed

lemma *awalk-verts-drop-conv*:
assumes *cas u p v*
shows *awalk-verts u′* (*drop n p*) = (*if n* < *length p then drop n* (*awalk-verts u p*) *else* [*u′*])
using *assms* **by** (*auto simp*: *awalk-verts-conv drop-map*)

lemma *awalk-decomp-verts*:
assumes *cas*: *cas u p v* **and** *ev-decomp*: *awalk-verts u p* = *xs* @ *y* # *ys*
obtains *q r* **where** *cas u q y cas y r v p* = *q* @ *r awalk-verts u q* = *xs* @ [*y*] *awalk-verts y r* = *y* # *ys*
using *assms*
proof −
define *q r* **where** *q* = *take* (*length xs*) *p* **and** *r* = *drop* (*length xs*) *p*
then have *p*: *p* = *q* @ *r* **by** *simp*
moreover from *p* **have** *cas u q* (*awlast u q*) *cas* (*awlast u q*) *r v*
using ‹*cas u p v*› **by** *auto*
moreover have *awlast u q* = *y*
using *q-def* **and** *assms* **by** (*auto simp*: *awalk-verts-take-conv*)
moreover have ∗: *awalk-verts u q* = *xs* @ [*awlast u q*]
using *assms q-def* **by** (*auto simp*: *awalk-verts-take-conv*)
moreover from ∗ **have** *awalk-verts y r* = *y* # *ys*
unfolding *q-def r-def* **using** *assms* **by** (*auto simp*: *awalk-verts-drop-conv not-less*)
ultimately show *?thesis* **by** (*intro that*) *auto*
qed

lemma *awalk-decomp*:
assumes *awalk u p v*
assumes *w* ∈ *set* (*awalk-verts u p*)
shows ∃ *q r*. *p* = *q* @ *r* ∧ *awalk u q w* ∧ *awalk w r v*

proof −
from *assms* **have** *cas u p v* **by** *auto*
moreover from *assms* **obtain** *xs ys* **where**
awalk-verts u p = *xs* @ *w* # *ys* **by** (*auto simp*: *in-set-conv-decomp*)
ultimately
obtain *q r* **where** *cas u q w cas w r v p* = *q* @ *r awalk-verts u q* = *xs* @ [*w*]
by (*auto intro*: *awalk-decomp-verts*)
with *assms* **show** *?thesis* **by** *auto*
qed

lemma *awalk-not-distinct-decomp*:
assumes *awalk u p v*
assumes ¬ *distinct* (*awalk-verts u p*)
shows ∃ *q r s*. *p* = *q* @ *r* @ *s* ∧ *distinct* (*awalk-verts u q*)
∧ *0* < *length r*
∧ (∃ *w*. *awalk u q w* ∧ *awalk w r w* ∧ *awalk w s v*)
proof −
from *assms*
obtain *xs ys zs y* **where**
pv-decomp: *awalk-verts u p* = *xs* @ *y* # *ys* @ *y* # *zs*
and *xs-y-props*: *distinct xs y* ∉ *set xs y* ∉ *set ys*
using *not-distinct-decomp-min-prefix* **by** *blast*

obtain *q p′* **where** *cas u q y p* = *q* @ *p′ awalk-verts u q* = *xs* @ [*y*]
and *p′-props*: *cas y p′ v awalk-verts y p′* = (*y* # *ys*) @ *y* # *zs*
using *assms pv-decomp* **by** − (*rule awalk-decomp-verts*, *auto*)
obtain *r s* **where** *cas y r y cas y s v p′* = *r* @ *s*
awalk-verts y r = *y* # *ys* @ [*y*] *awalk-verts y s* = *y* # *zs*
using *p′-props* **by** (*rule awalk-decomp-verts*) *auto*

have *p* = *q* @ *r* @ *s* **using** ‹*p* = *q* @ *p′*› ‹*p′* = *r* @ *s*› **by** *simp*
moreover
have *distinct* (*awalk-verts u q*) **using** ‹*awalk-verts u q* = *xs* @ [*y*]› **and** *xs-y-props*
by *simp*
moreover
have *0* < *length r* **using** ‹*awalk-verts y r* = *y* # *ys* @ [*y*]› **by** *auto*
moreover
from *pv-decomp assms* **have** *y* ∈ *verts G* **by** *auto*
then have *awalk u q y awalk y r y awalk y s v*
using ‹*awalk u p v*› ‹*cas u q y*› ‹*cas y r y*› ‹*cas y s v*› **unfolding** ‹*p* = *q* @ *r*
@ *s*›
by (*auto simp*: *awalk-def*)
ultimately show *?thesis* **by** *blast*
qed

lemma *apath-decomp-disjoint*:
assumes *apath u p v*
assumes *p* = *q* @ *r*

assumes $x \in set\ (awalk\text{-}verts\ u\ q)$ $x \in set\ (tl\ (awalk\text{-}verts\ (awlast\ u\ q)\ r))$
shows *False*
using *assms* **by** (*auto simp*: *apath-def awalk-verts-append*)

6.3 Cycles

definition *closed-w* :: *'b awalk* ⇒ *bool* **where**
closed-w p ≡ ∃ *u*. *awalk u p u* ∧ *0* < *length p*

The definitions of cycles in textbooks vary w.r.t to the minimial length of a cycle.

The definition given here matches [2]. [1] excludes loops from being cycles. Volkmann (Lutz Volkmann: Graphen an allen Ecken und Kanten, 2006 (?)) places no restriction on the length in the definition, but later usage assumes cycles to be non-empty.

definition (**in** *pre-digraph*) *cycle* :: *'b awalk* ⇒ *bool* **where**
cycle p ≡ ∃ *u*. *awalk u p u* ∧ *distinct* (*tl* (*awalk-verts u p*)) ∧ *p* ≠ []

lemma *cycle-altdef*:
cycle p ⟷ *closed-w p* ∧ (∃ *u*. *distinct* (*tl* (*awalk-verts u p*)))
by (*cases p*) (*auto simp*: *closed-w-def cycle-def*)

lemma (**in** *wf-digraph*) *distinct-tl-verts-imp-distinct*:
assumes *awalk u p v*
assumes *distinct* (*tl* (*awalk-verts u p*))
shows *distinct p*
proof (*rule ccontr*)
assume ¬*distinct p*
then obtain *e xs ys zs* **where** *p-decomp*: *p* = *xs* @ *e* # *ys* @ *e* # *zs*
by (*blast dest*: *not-distinct-decomp-min-prefix*)
then show *False*
using *assms p-decomp* **by** (*auto simp*: *awalk-verts-append awalk-Cons-iff set-awalk-verts*)
qed

lemma (**in** *wf-digraph*) *distinct-verts-imp-distinct*:
assumes *awalk u p v*
assumes *distinct* (*awalk-verts u p*)
shows *distinct p*
using *assms* **by** (*blast intro*: *distinct-tl-verts-imp-distinct distinct-tl*)

lemma (**in** *wf-digraph*) *cycle-conv*:
cycle p ⟷ (∃ *u*. *awalk u p u* ∧ *distinct* (*tl* (*awalk-verts u p*)) ∧ *distinct p* ∧ *p* ≠ [])
unfolding *cycle-def* **by** (*auto intro*: *distinct-tl-verts-imp-distinct*)

lemma (**in** *loopfree-digraph*) *cycle-digraph-conv*:
cycle p ⟷ (∃ *u*. *awalk u p u* ∧ *distinct* (*tl* (*awalk-verts u p*)) ∧ *2* ≤ *length p*)
(**is** *?L* ⟷ *?R*)
proof

```
  assume cycle p
  then obtain u where ∗: awalk u p u distinct (tl (awalk-verts u p)) p ≠ []
    unfolding cycle-def by auto
  have 2 ≤ length p
  proof (rule ccontr)
    assume ¬?thesis with ∗ obtain e where p=[e]
      by (cases p) (auto simp: not-le)
    then show False using ∗ by (auto simp: awalk-simps dest: no-loops)
  qed
  then show ?R using ∗ by auto
qed (auto simp: cycle-def)

lemma (in wf-digraph) closed-w-imp-cycle:
  assumes closed-w p shows ∃ p. cycle p
  using assms
proof (induct length p arbitrary: p rule: less-induct)
  case less
  then obtain u where ∗: awalk u p u p ≠ [] by (auto simp: closed-w-def)
  show ?thesis
  proof cases
    assume distinct (tl (awalk-verts u p))
    with less show ?thesis by (auto simp: closed-w-def cycle-altdef)
  next
    assume A: ¬distinct (tl (awalk-verts u p))
    then obtain e es where p = e # es by (cases p) auto
    with A ∗ have ∗∗: awalk (head G e) es u ¬distinct (awalk-verts (head G e) es)
      by (auto simp: awalk-Cons-iff)
    obtain q r s where es = q @ r @ s ∃ w. awalk w r w closed-w r
      using awalk-not-distinct-decomp[OF ∗∗] by (auto simp: closed-w-def)
    then have length r < length p using ‹p = -› by auto
    then show ?thesis using ‹closed-w r› by (rule less)
  qed
qed
```

6.4 Reachability

```
lemma reachable1-awalk:
  u →⁺ v ⟷ (∃ p. awalk u p v ∧ p ≠ [])
proof
  assume u →⁺ v then show ∃ p. awalk u p v ∧ p ≠ []
  proof (induct rule: converse-trancl-induct)
    case (base y) then obtain e where e ∈ arcs G tail G e = y head G e = v by
auto
    with arc-implies-awalk show ?case by auto
  next
    case (step x y)
    then obtain p where awalk y p v p ≠ [] by auto
    moreover
    from ‹x → y› obtain e where tail G e = x head G e = y e ∈ arcs G
```

```
    by auto
  ultimately
  have awalk x (e # p) v
    by (auto simp: awalk-Cons-iff)
  then show ?case by auto
 qed
next
 assume ∃ p. awalk u p v ∧ p ≠ [] then obtain p where awalk u p v p ≠ [] by
auto
 thus u →+ v
 proof (induct p arbitrary: u)
  case (Cons a as) then show ?case
  by (cases as = []) (auto simp: awalk-simps trancl-into-trancl2 dest: in-arcs-imp-in-arcs-ends)
 qed simp
qed

lemma reachable-awalk:
 u →* v ⟷ (∃ p. awalk u p v)
proof cases
 assume u = v
 have u →*u ⟷ awalk u [] u by (auto simp: awalk-Nil-iff reachable-in-verts)
 also have ... ⟷ (∃ p. awalk u p u)
  by (metis awalk-Nil-iff awalk-hd-in-verts)
 finally show ?thesis using ‹u = v› by simp
next
 assume u ≠ v
 then have u →* v ⟷ u →+ v by auto
 also have ... ⟷ (∃ p. awalk u p v)
  using ‹u ≠ v› unfolding reachable1-awalk by force
 finally show ?thesis .
qed

lemma reachable-awalkI[intro?]:
 assumes awalk u p v
 shows u →* v
 unfolding reachable-awalk using assms by auto

lemma reachable1-awalkI:
 awalk v p w ⟹ p ≠ [] ⟹ v →+ w
by (auto simp add: reachable1-awalk)


lemma reachable-arc-trans:
 assumes u →* v arc e (v,w)
 shows u →* w
proof −
 from ‹u →* v› obtain p where awalk u p v
  by (auto simp: reachable-awalk)
 moreover have awalk v [e] w
```

using ‹*arc e* (*v*,*w*)›
by (*auto simp*: *arc-def awalk-def*)
ultimately have *awalk u* (*p* @ [*e*]) *w*
by (*rule awalk-appendI*)
then show *?thesis* **..**
qed

lemma *awalk-verts-reachable-from*:
assumes *awalk u p v w* ∈ *set* (*awalk-verts u p*) **shows** *u* →**G w*
proof −
obtain *s* **where** *awalk u s w* **using** *awalk-decomp*[*OF assms*] **by** *blast*
then show *?thesis* **by** (*metis reachable-awalk*)
qed

lemma *awalk-verts-reachable-to*:
assumes *awalk u p v w* ∈ *set* (*awalk-verts u p*) **shows** *w* →**G v*
proof −
obtain *s* **where** *awalk w s v* **using** *awalk-decomp*[*OF assms*] **by** *blast*
then show *?thesis* **by** (*metis reachable-awalk*)
qed

6.5 Paths

lemma (**in** *fin-digraph*) *length-apath-less*:
assumes *apath u p v*
shows *length p* < *card* (*verts G*)
proof −
have *length p* < *length* (*awalk-verts u p*) **unfolding** *awalk-verts-conv*
by (*auto simp*: *awalk-verts-conv*)
also have *length* (*awalk-verts u p*) = *card* (*set* (*awalk-verts u p*))
using ‹*apath u p v*› **by** (*auto simp*: *apath-def distinct-card*)
also have ... ≤ *card* (*verts G*)
using ‹*apath u p v*› **unfolding** *apath-def awalk-conv*
by (*auto intro*: *card-mono*)
finally show *?thesis* **.**
qed

lemma (**in** *fin-digraph*) *length-apath*:
assumes *apath u p v*
shows *length p* ≤ *card* (*verts G*)
using *length-apath-less*[*OF assms*] **by** *auto*

lemma (**in** *fin-digraph*) *apaths-finite-triple*:
shows *finite* {(*u*,*p*,*v*). *apath u p v*}
proof −
have ⋀*u p v*. *awalk u p v* ⟹ *distinct* (*awalk-verts u p*) ⟹*length p* ≤ *card* (*verts G*)
by (*rule length-apath*) (*auto simp*: *apath-def*)
then have {(*u*,*p*,*v*). *apath u p v*} ⊆ *verts G* × {*es*. *set es* ⊆ *arcs G* ∧ *length es*

```
≤ card (verts G)} × verts G
    by (auto simp: apath-def)
  moreover have finite ...
    using finite-verts finite-arcs
    by (intro finite-cartesian-product finite-lists-length-le)
  ultimately show ?thesis by (rule finite-subset)
qed

lemma (in fin-digraph) apaths-finite:
  shows finite {p. apath u p v}
proof −
  have {p. apath u p v} ⊆ (fst o snd) ` {(u,p,v). apath u p v}
    by force
  with apaths-finite-triple show ?thesis  by (rule finite-surj)
qed

fun is-awalk-cyc-decomp :: 'b awalk =>
  ('b awalk × 'b awalk × 'b awalk) ⇒ bool where
  is-awalk-cyc-decomp p (q,r,s) ⟷ p = q @ r @ s
    ∧ (∃ u v w. awalk u q v ∧ awalk v r v ∧ awalk v s w)
    ∧ 0 <length r
    ∧ (∃ u. distinct (awalk-verts u q))

definition awalk-cyc-decomp :: 'b awalk
    ⇒ 'b awalk × 'b awalk × 'b awalk where
  awalk-cyc-decomp p = (SOME qrs. is-awalk-cyc-decomp p qrs)

function awalk-to-apath :: 'b awalk ⇒ 'b awalk where
  awalk-to-apath p = (if ¬(∃ u. distinct (awalk-verts u p)) ∧ (∃ u v. awalk u p v)
     then (let (q,r,s) = awalk-cyc-decomp p in awalk-to-apath (q @ s))
     else p)
by auto

lemma awalk-cyc-decomp-has-prop:
  assumes awalk u p v and ¬distinct (awalk-verts u p)
  shows is-awalk-cyc-decomp p (awalk-cyc-decomp p)
proof −
  obtain q r s where ∗: p = q @ r @ s ∧ distinct (awalk-verts u q)
      ∧ 0 < length r
      ∧ (∃ w. awalk u q w ∧ awalk w r w ∧ awalk w s v)
    by (atomize-elim) (rule awalk-not-distinct-decomp[OF assms])
  then have ∃ x. is-awalk-cyc-decomp p x
    by (intro exI[where x=(q,r,s)]) auto
  then show ?thesis unfolding awalk-cyc-decomp-def ..
qed

lemma awalk-cyc-decompE:
  assumes dec: awalk-cyc-decomp p = (q,r,s)
  assumes p-props: awalk u p v ¬distinct (awalk-verts u p)
```

obtains $p = q$ @ r @ s *distinct* (*awalk-verts u q*) $\exists w.$ *awalk u q w* $\wedge$ *awalk w r w* $\wedge$ *awalk w s v closed-w r*
proof
show $p = q$ @ r @ s *distinct* (*awalk-verts u q*) *closed-w r*
using *awalk-cyc-decomp-has-prop*[*OF p-props*] **and** *dec*
by (*auto simp*: *closed-w-def awalk-verts-conv*)
then have $p \neq []$ **by** (*auto simp*: *closed-w-def*)

obtain u' w' v' **where** *obt-awalk*: *awalk u' q w' awalk w' r w' awalk w' s v'*
using *awalk-cyc-decomp-has-prop*[*OF p-props*] **and** *dec* **by** *auto*
then have *awalk u' p v'*
using ‹$p = q$ @ r @ s› **by** *simp*
then have $u = u'$ **and** $v = v'$ **using** ‹$p \neq []$› ‹*awalk u p v*› **by** (*metis awalk-ends*)+
then have *awalk u q w' awalk w' r w' awalk w' s v*
using *obt-awalk* **by** *auto*
then show $\exists w.$ *awalk u q w* $\wedge$ *awalk w r w* $\wedge$ *awalk w s v* **by** *auto*
qed

lemma *awalk-cyc-decompE'*:
assumes *p-props*: *awalk u p v* $\neg$*distinct* (*awalk-verts u p*)
obtains *q r s* **where** $p = q$ @ r @ s *distinct* (*awalk-verts u q*) $\exists w.$ *awalk u q w* $\wedge$ *awalk w r w* $\wedge$ *awalk w s v closed-w r*
proof −
obtain *q r s* **where** *awalk-cyc-decomp p* = (*q*,*r*,*s*)
by (*cases awalk-cyc-decomp p*) *auto*
then have $p = q$ @ r @ s *distinct* (*awalk-verts u q*) $\exists w.$ *awalk u q w* $\wedge$ *awalk w r w* $\wedge$ *awalk w s v closed-w r*
using *assms* **by** (*auto elim*: *awalk-cyc-decompE*)
then show *?thesis* **..**
qed

termination *awalk-to-apath*
proof (*relation measure length*)
fix *G p qrs rs q r s*

have *X*: $\bigwedge x\ y.$ *closed-w r* $\Longrightarrow$ *awalk x r y* $\Longrightarrow x = y$
unfolding *closed-w-def* **by** (*blast dest*: *awalk-ends*)

assume $\neg(\exists u.$ *distinct* (*awalk-verts u p*)) $\wedge(\exists u\ v.$ *awalk u p v*)
and $**$:*qrs* = *awalk-cyc-decomp p* (*q*, *rs*) = *qrs* (*r*, *s*) = *rs*
then obtain *u v* **where** $*$: *awalk u p v* $\neg$*distinct* (*awalk-verts u p*)
by (*cases p*) *auto*
then have *awalk-cyc-decomp p* = (*q*,*r*,*s*) **using** $**$ **by** *simp*
then have *is-awalk-cyc-decomp p* (*q*,*r*,*s*)
apply (*rule awalk-cyc-decompE*[*OF* - $*$])
using *X*[*of awlast u q awlast* (*awlast u q*) *r*] $*$(*1*)
by (*auto simp*: *closed-w-def*)
then show (*q* @ *s*, *p*) $\in$ *measure length*

```
    by (auto simp: closed-w-def)
qed simp
declare awalk-to-apath.simps[simp del]

lemma awalk-to-apath-induct[consumes 1, case-names path decomp]:
  assumes awalk: awalk u p v
  assumes dist: ⋀p. awalk u p v ⟹ distinct (awalk-verts u p) ⟹ P p
  assumes dec: ⋀p q r s. ⟦awalk u p v; awalk-cyc-decomp p = (q,r,s);
    ¬distinct (awalk-verts u p); P (q @ s)⟧ ⟹ P p
  shows P p
using awalk
proof (induct length p arbitrary: p rule: less-induct)
  case less
  show ?case
  proof (cases distinct (awalk-verts u p))
    case True then show ?thesis by (auto intro: dist less.prems)
  next
    case False
    obtain q r s where p-cdecomp: awalk-cyc-decomp p = (q,r,s)
      by (cases awalk-cyc-decomp p) auto
    then have is-awalk-cyc-decomp p (q,r,s) p = q @ r @ s
      using awalk-cyc-decomp-has-prop[OF less.prems(1) False] by auto
    then have length (q @ s) < length p awalk u (q @ s) v
      using less.prems by (auto dest!: awalk-ends-eqD)
    then have P (q @ s) by (auto intro: less)

    with p-cdecomp False show ?thesis by (auto intro: dec less.prems)
  qed
qed

lemma step-awalk-to-apath:
  assumes awalk: awalk u p v
    and decomp: awalk-cyc-decomp p = (q, r, s)
    and dist: ¬ distinct (awalk-verts u p)
  shows awalk-to-apath p = awalk-to-apath (q @ s)
proof −
  from dist have ¬(∃ u. distinct (awalk-verts u p))
    by (auto simp: awalk-verts-conv)
  with awalk and decomp show awalk-to-apath p = awalk-to-apath (q @ s)
    by (auto simp: awalk-to-apath.simps)
qed

lemma apath-awalk-to-apath:
  assumes awalk u p v
  shows apath u (awalk-to-apath p) v
using assms
proof (induct rule: awalk-to-apath-induct)
  case (path p)
  then have awalk-to-apath p = p
```

by (*auto simp*: *awalk-to-apath.simps*)
then show *?case* **using** *path* **by** (*auto simp*: *apath-def*)
next
case (*decomp p q r s*)
then show *?case* **using** *step-awalk-to-apath*[*of - p - q r s*] **by** *simp*
qed

lemma (**in** *wf-digraph*) *awalk-to-apath-subset*:
assumes *awalk u p v*
shows *set* (*awalk-to-apath p*) ⊆ *set p*
using *assms*
proof (*induct rule*: *awalk-to-apath-induct*)
case (*path p*)
then have *awalk-to-apath p* = *p*
by (*auto simp*: *awalk-to-apath.simps*)
then show *?case* **by** *simp*
next
case (*decomp p q r s*)
have ∗: ¬(∃ *u*. *distinct* (*awalk-verts u p*)) ∧ (∃ *u v*. *awalk u p v*)
using *decomp* **by** (*cases p*) *auto*
have *set* (*awalk-to-apath* (*q* @ *s*)) ⊆ *set p*
using *decomp* **by** (*auto elim*!: *awalk-cyc-decompE*)
then
show *?case* **by** (*subst awalk-to-apath.simps*) (*simp only*: ∗ *simp-thms if-True decomp Let-def prod.simps*)
qed

lemma *reachable-apath*:
u →* *v* ⟷ (∃ *p*. *apath u p v*)
by (*auto intro*: *awalkI-apath apath-awalk-to-apath simp*: *reachable-awalk*)

lemma *no-loops-in-apath*:
assumes *apath u p v a* ∈ *set p* **shows** *tail G a* ≠ *head G a*
proof −
from ‹*a* ∈ *set p*› **obtain** *p1 p2* **where** *p* = *p1* @ *a* # *p2* **by** (*auto simp*: *in-set-conv-decomp*)
with ‹*apath u p v*› **have** *apath* (*tail G a*) ([*a*] @ *p2*) (*v*)
by (*auto simp*: *apath-append-iff apath-Cons-iff apath-Nil-iff*)
then have *apath* (*tail G a*) [*a*] (*head G a*) **by** − (*drule apath-append-iff*[*THEN iffD1*], *simp*)
then show *?thesis* **by** (*auto simp*: *apath-Cons-iff*)
qed

end

end

theory *Pair-Digraph*
imports
Digraph
Bidirected-Digraph
Arc-Walk
begin

7 Digraphs without Parallel Arcs

If no parallel arcs are desired, arcs can be accurately described as pairs of This is the natural representation for Digraphs without multi-arcs. and *head* *G*, making it easier to deal with multiple related graphs and to modify a graph by adding edges.

This theory introduces such a specialisation of digraphs.

record *'a pair-pre-digraph* = *pverts* :: *'a set parcs* :: *'a rel*

definition *with-proj* :: *'a pair-pre-digraph* ⇒ (*'a*, *'a* × *'a*) *pre-digraph* **where**
with-proj G = (| *verts* = *pverts G*, *arcs* = *parcs G*, *tail* = *fst*, *head* = *snd* |)

declare [[*coercion with-proj*]]

primrec *pawalk-verts* :: *'a* ⇒ (*'a* × *'a*) *awalk* ⇒ *'a list* **where**
pawalk-verts u [] = [*u*] |
pawalk-verts u (*e* # *es*) = *fst e* # *pawalk-verts* (*snd e*) *es*

fun *pcas* :: *'a* ⇒ (*'a* × *'a*) *awalk* ⇒ *'a* ⇒ *bool* **where**
pcas u [] *v* = (*u* = *v*) |
pcas u (*e* # *es*) *v* = (*fst e* = *u* ∧ *pcas* (*snd e*) *es v*)

lemma *with-proj-simps*[*simp*]:
verts (*with-proj G*) = *pverts G*
arcs (*with-proj G*) = *parcs G*
arcs-ends (*with-proj G*) = *parcs G*
tail (*with-proj G*) = *fst*
head (*with-proj G*) = *snd*
by (*auto simp*: *with-proj-def arcs-ends-conv*)

lemma *cas-with-proj-eq*: *pre-digraph.cas* (*with-proj G*) = *pcas*
proof (*unfold fun-eq-iff*, *intro allI*)
fix *u es v* **show** *pre-digraph.cas* (*with-proj G*) *u es v* = *pcas u es v*
by (*induct es arbitrary*: *u*) (*auto simp*: *pre-digraph.cas.simps*)
qed

lemma *awalk-verts-with-proj-eq*: *pre-digraph.awalk-verts* (*with-proj G*) = *pawalk-verts*
proof (*unfold fun-eq-iff*, *intro allI*)
fix *u es* **show** *pre-digraph.awalk-verts* (*with-proj G*) *u es* = *pawalk-verts u es*
by (*induct es arbitrary*: *u*) (*auto simp*: *pre-digraph.awalk-verts.simps*)

qed

locale *pair-pre-digraph* = **fixes** *G* :: *'a pair-pre-digraph*
begin

lemmas [*simp*] = *cas-with-proj-eq awalk-verts-with-proj-eq*

end

locale *pair-wf-digraph* = *pair-pre-digraph* +
  assumes *arc-fst-in-verts*: $\bigwedge e.\ e \in parcs\ G \Longrightarrow fst\ e \in pverts\ G$
  assumes *arc-snd-in-verts*: $\bigwedge e.\ e \in parcs\ G \Longrightarrow snd\ e \in pverts\ G$
begin

lemma *in-arcsD1*: $(u,v) \in parcs\ G \Longrightarrow u \in pverts\ G$
  and *in-arcsD2*: $(u,v) \in parcs\ G \Longrightarrow v \in pverts\ G$
  by (*auto dest*: *arc-fst-in-verts arc-snd-in-verts*)

lemmas *wellformed'* = *in-arcsD1 in-arcsD2*

end

locale *pair-fin-digraph* = *pair-wf-digraph* +
  assumes *pair-finite-verts*: *finite* (*pverts G*)
    and *pair-finite-arcs*: *finite* (*parcs G*)

locale *pair-sym-digraph* = *pair-wf-digraph* +
  assumes *pair-sym-arcs*: *symmetric G*

locale *pair-loopfree-digraph* = *pair-wf-digraph* +
  assumes *pair-no-loops*: $e \in parcs\ G \Longrightarrow fst\ e \neq snd\ e$

locale *pair-bidirected-digraph* = *pair-sym-digraph* + *pair-loopfree-digraph*

locale *pair-pseudo-graph* = *pair-fin-digraph* + *pair-sym-digraph*

locale *pair-digraph* = *pair-fin-digraph* + *pair-loopfree-digraph*

locale *pair-graph* = *pair-digraph* + *pair-pseudo-graph*

sublocale *pair-pre-digraph* $\subseteq$ *pre-digraph with-proj G*
  rewrites *verts G* = *pverts G* **and** *arcs G* = *parcs G* **and** *tail G* = *fst* **and** *head G* = *snd*
    and *arcs-ends G* = *parcs G*
    and *pre-digraph.awalk-verts G* = *pawalk-verts*

and *pre-digraph.cas G = pcas*
by *unfold-locales auto*

sublocale *pair-wf-digraph ⊆ wf-digraph with-proj G*
rewrites *verts G = pverts G* **and** *arcs G = parcs G* **and** *tail G = fst* **and** *head G = snd*
and *arcs-ends G = parcs G*
and *pre-digraph.awalk-verts G = pawalk-verts*
and *pre-digraph.cas G = pcas*
by *unfold-locales* (*auto simp*: *arc-fst-in-verts arc-snd-in-verts*)

sublocale *pair-fin-digraph ⊆ fin-digraph with-proj G*
rewrites *verts G = pverts G* **and** *arcs G = parcs G* **and** *tail G = fst* **and** *head G = snd*
and *arcs-ends G = parcs G*
and *pre-digraph.awalk-verts G = pawalk-verts*
and *pre-digraph.cas G = pcas*
using *pair-finite-verts pair-finite-arcs* **by** *unfold-locales auto*

sublocale *pair-sym-digraph ⊆ sym-digraph with-proj G*
rewrites *verts G = pverts G* **and** *arcs G = parcs G* **and** *tail G = fst* **and** *head G = snd*
and *arcs-ends G = parcs G*
and *pre-digraph.awalk-verts G = pawalk-verts*
and *pre-digraph.cas G = pcas*
using *pair-sym-arcs* **by** *unfold-locales auto*

sublocale *pair-pseudo-graph ⊆ pseudo-graph with-proj G*
rewrites *verts G = pverts G* **and** *arcs G = parcs G* **and** *tail G = fst* **and** *head G = snd*
and *arcs-ends G = parcs G*
and *pre-digraph.awalk-verts G = pawalk-verts*
and *pre-digraph.cas G = pcas*
by *unfold-locales auto*

sublocale *pair-loopfree-digraph ⊆ loopfree-digraph with-proj G*
rewrites *verts G = pverts G* **and** *arcs G = parcs G* **and** *tail G = fst* **and** *head G = snd*
and *arcs-ends G = parcs G*
and *pre-digraph.awalk-verts G = pawalk-verts*
and *pre-digraph.cas G = pcas*
using *pair-no-loops* **by** *unfold-locales auto*

sublocale *pair-digraph ⊆ digraph with-proj G*
rewrites *verts G = pverts G* **and** *arcs G = parcs G* **and** *tail G = fst* **and** *head G = snd*
and *arcs-ends G = parcs G*
and *pre-digraph.awalk-verts G = pawalk-verts*
and *pre-digraph.cas G = pcas*

by *unfold-locales* (*auto simp*: *arc-to-ends-def*)

sublocale *pair-graph* ⊆ *graph with-proj G*
rewrites *verts G* = *pverts G* **and** *arcs G* = *parcs G* **and** *tail G* = *fst* **and** *head G* = *snd*
and *arcs-ends G* = *parcs G*
and *pre-digraph.awalk-verts G* = *pawalk-verts*
and *pre-digraph.cas G* = *pcas*
by *unfold-locales auto*

sublocale *pair-graph* ⊆ *pair-bidirected-digraph* **by** *unfold-locales*

lemma *wf-digraph-wp-iff*: *wf-digraph* (*with-proj G*) = *pair-wf-digraph G* (**is** *?L* ⟷ *?R*)
proof
assume *?L* **then interpret** *wf-digraph with-proj G* **.**
show *?R* **using** *wellformed* **by** *unfold-locales auto*
next
assume *?R* **then interpret** *pair-wf-digraph G* **.**
show *?L* **by** *unfold-locales*
qed

lemma (**in** *pair-fin-digraph*) *pair-fin-digraph*[*intro*!]: *pair-fin-digraph G* **..**

context *pair-digraph* **begin**

lemma *pair-wf-digraph*[*intro*!]: *pair-wf-digraph G* **by** *intro-locales*

lemma *pair-digraph*[*intro*!]: *pair-digraph G* **..**

lemma (**in** *pair-loopfree-digraph*) *no-loops'*:
(*u*,*v*) ∈ *parcs G* ⟹ *u* ≠ *v*
by (*auto dest*: *no-loops*)

end

lemma (**in** *pair-wf-digraph*) *apath-succ-decomp*:
assumes *apath u p v*
assumes (*x*,*y*) ∈ *set p*
assumes *y* ≠ *v*
shows ∃ *p1 z p2*. *p* = *p1* @ (*x*,*y*) # (*y*,*z*) # *p2* ∧ *x* ≠ *z* ∧ *y* ≠ *z*
proof −
from ‹(*x*,*y*) ∈ *set p*› **obtain** *p1 p2* **where** *p-decomp*: *p* = *p1* @ (*x*,*y*) # *p2*
by (*metis* (*no-types*) *in-set-conv-decomp-first*)
from *p-decomp* ‹*apath u p v*› ‹*y* ≠ *v*› **have** *p2* ≠ [] *awalk y p2 v*
by (*auto simp*: *apath-def awalk-Cons-iff*)
then obtain *z p2'* **where** *p2-decomp*: *p2* = (*y*,*z*) # *p2'*
by *atomize-elim* (*cases p2*, *auto simp*: *awalk-Cons-iff*)
then have *x* ≠ *z* ∧ *y* ≠ *z* **using** *p-decomp p2-decomp* ‹*apath u p v*›

by (*auto simp*: *apath-append-iff apath-simps hd-in-awalk-verts*)
with *p-decomp p2-decomp* **have** $p = p1 \ @ \ (x,y) \ \# \ (y,z) \ \# \ p2' \wedge x \neq z \wedge y \neq z$
by *auto*
then show *?thesis* **by** *blast*
qed

lemma (**in** *pair-sym-digraph*) *arcs-symmetric*:
$(a,b) \in parcs\ G \Longrightarrow (b,a) \in parcs\ G$
using *sym-arcs* **by** (*auto simp*: *symmetric-def elim*: *symE*)

lemma (**in** *pair-pseudo-graph*) *pair-pseudo-graph*[*intro*]: *pair-pseudo-graph G* **..**

lemma (**in** *pair-graph*) *pair-graph*[*intro*]: *pair-graph G* **by** *unfold-locales*
lemma (**in** *pair-graph*) *pair-graphD-graph*: *graph G* **by** *unfold-locales*

lemma *pair-graphI-graph*:
assumes *graph* (*with-proj G*) **shows** *pair-graph G*
proof −
interpret *G*: *graph with-proj G* **by** *fact*
show *?thesis*
using *G.wellformed G.finite-arcs G.finite-verts G.no-loops*
by *unfold-locales auto*
qed

lemma *pair-loopfreeI-loopfree*:
assumes *loopfree-digraph* (*with-proj G*) **shows** *pair-loopfree-digraph G*
proof −
interpret *loopfree-digraph with-proj G* **by** *fact*
show *?thesis* **using** *wellformed no-loops* **by** *unfold-locales auto*
qed

7.1 Path reversal for Pair Digraphs

This definition is only meaningful in *Pair-Digraph*

primrec *rev-path* :: $('a \times 'a)\ awalk \Rightarrow ('a \times 'a)\ awalk$ **where**
rev-path [] = [] |
rev-path (*e* # *es*) = *rev-path es* @ [(*snd e*, *fst e*)]

lemma *rev-path-append*[*simp*]: *rev-path* (*p* @ *q*) = *rev-path q* @ *rev-path p*
by (*induct p*) *auto*

lemma *rev-path-rev-path*[*simp*]:
rev-path (*rev-path p*) = *p*
by (*induct p*) *auto*

lemma *rev-path-empty*[*simp*]:
rev-path p = [] ⟷ *p* = []
by (*induct p*) *auto*

lemma *rev-path-eq*: *rev-path p* = *rev-path q* ⟷ *p* = *q*
by (*metis rev-path-rev-path*)

lemma (**in** *pair-sym-digraph*)
assumes *awalk u p v*
shows *awalk-verts-rev-path*: *awalk-verts v* (*rev-path p*) = *rev* (*awalk-verts u p*)
and *awalk-rev-path'*: *awalk v* (*rev-path p*) *u*
using *assms*
proof (*induct p arbitrary*: *u*)
case *Nil* **case** *1* **then show** *?case* **by** *auto*
next
case *Nil* **case** *2* **then show** *?case* **by** (*auto simp*: *awalk-Nil-iff*)
next
case (*Cons e es*) **case** *1*
with *Cons* **have** *walks*: *awalk v* (*rev-path es*) (*snd e*)
awalk (*snd e*) [(*snd e*, *fst e*)] *u*
and *verts*: *awalk-verts v* (*rev-path es*) = *rev* (*awalk-verts* (*snd e*) *es*)
by (*auto simp*: *awalk-simps intro*: *arcs-symmetric*)

from *walks* **have** *awalk v* (*rev-path es* @ [(*snd e*, *fst e*)]) *u*
by *simp*
moreover
have *tl* (*awalk-verts* (*awlast v* (*rev-path es*)) [(*snd e*, *fst e*)]) = [*fst e*]
by *auto*
ultimately
show *?case* **using** *1 verts* **by** (*auto simp*: *awalk-verts-append*)
next
case (*Cons e es*) **case** *2*
with *Cons* **have** *awalk v* (*rev-path es*) (*snd e*)
by (*auto simp*: *awalk-Cons-iff*)
moreover
have *rev-path* (*e* # *es*) = *rev-path es* @ [(*snd e*, *fst e*)]
by *auto*
moreover
from *Cons 2* **have** *awalk* (*snd e*) [(*snd e*, *fst e*)] *u*
by (*auto simp*: *awalk-simps intro*: *arcs-symmetric*)
ultimately show *awalk v* (*rev-path* (*e* # *es*)) *u*
by *simp*
qed

lemma (**in** *pair-sym-digraph*) *awalk-rev-path*[*simp*]:
awalk v (*rev-path p*) *u* = *awalk u p v* (**is** *?L* = *?R*)
by (*metis awalk-rev-path' rev-path-rev-path*)

lemma (**in** *pair-sym-digraph*) *apath-rev-path*[*simp*]:
apath v (*rev-path p*) *u* = *apath u p v*
by (*auto simp*: *awalk-verts-rev-path apath-def*)

7.2 Subdividing Edges

subdivide an edge (=two associated arcs) in graph

fun *subdivide* :: *'a pair-pre-digraph* ⇒ *'a* × *'a* ⇒ *'a* ⇒ *'a pair-pre-digraph* **where**
subdivide G (*u*,*v*) *w* = (|
pverts = *pverts G* ∪ {*w*},
parcs = (*parcs G* − {(*u*,*v*),(*v*,*u*)}) ∪ {(*u*,*w*), (*w*,*u*), (*w*, *v*), (*v*, *w*)}|)

declare *subdivide.simps*[*simp del*]

subdivide an arc in a path

fun *sd-path* :: *'a* × *'a* ⇒ *'a* ⇒ (*'a* × *'a*) *awalk* ⇒ (*'a* × *'a*) *awalk* **where**
sd-path - - [] = []
| *sd-path* (*u*,*v*) *w* (*e* # *es*) = (*if e* = (*u*,*v*)
then [(*u*,*w*),(*w*,*v*)]
else if e = (*v*,*u*)
then [(*v*,*w*),(*w*,*u*)]
else [*e*]) @ *sd-path* (*u*,*v*) *w es*

contract an arc in a path

fun *co-path* :: *'a* × *'a* ⇒ *'a* ⇒ (*'a* × *'a*) *awalk* ⇒ (*'a* × *'a*) *awalk* **where**
co-path - - [] = []
| *co-path* - - [*e*] = [*e*]
| *co-path* (*u*,*v*) *w* (*e1* # *e2* # *es*) = (*if e1* = (*u*,*w*) ∧ *e2* = (*w*,*v*)
then (*u*,*v*) # *co-path* (*u*,*v*) *w es*
else if e1 = (*v*,*w*) ∧ *e2* = (*w*,*u*)
then (*v*,*u*) # *co-path* (*u*,*v*) *w es*
else e1 # *co-path* (*u*,*v*) *w* (*e2* # *es*))

lemma *co-path-simps*[*simp*]:
⟦*e1* ≠ (*fst e*, *w*); *e1* ≠ (*snd e*,*w*)⟧ ⟹ *co-path e w* (*e1* # *es*) = *e1* # *co-path e w es*
⟦*e1* = (*fst e*, *w*); *e2* = (*w*, *snd e*)⟧ ⟹ *co-path e w* (*e1* # *e2* # *es*) = *e* # *co-path e w es*
⟦*e1* = (*snd e*, *w*); *e2* = (*w*, *fst e*)⟧
⟹ *co-path e w* (*e1* # *e2* # *es*) = (*snd e*, *fst e*) # *co-path e w es*
⟦*e1* ≠ (*fst e*, *w*) ∨ *e2* ≠ (*w*, *snd e*); *e1* ≠ (*snd e*, *w*) ∨ *e2* ≠ (*w*, *fst e*)⟧
⟹ *co-path e w* (*e1* # *e2* # *es*) = *e1* # *co-path e w* (*e2* # *es*)
apply (*cases es*; *auto*)
apply (*cases e*; *auto*)
apply (*cases e*; *auto*)
apply (*cases e*; *cases fst e* = *snd e*; *auto*)
apply (*cases e*; *cases fst e* = *snd e*; *auto*)
done

lemma *co-path-nonempty*[*simp*]: *co-path e w p* = [] ⟷ *p* = []
by (*cases e*) (*cases p rule*: *list-exhaust-NSC*, *auto*)

declare *co-path.simps*(*3*)[*simp del*]

lemma *verts-subdivide*[*simp*]: *pverts* (*subdivide G e w*) = *pverts G* ∪ {*w*}
by (*cases e*) (*auto simp*: *subdivide.simps*)

lemma *arcs-subdivide*[*simp*]:
shows *parcs* (*subdivide G* (*u*,*v*) *w*) = (*parcs G* − {(*u*,*v*),(*v*,*u*)}) ∪ {(*u*,*w*), (*w*,*u*), (*w*, *v*), (*v*, *w*)}
by (*auto simp*: *subdivide.simps*)

lemmas *subdivide-simps* = *verts-subdivide arcs-subdivide*

lemma *sd-path-induct*[*case-names empty pass sd sdrev*]:
assumes *A*: *P e* []
and *B*: ⋀*e′ es*. *e′* ≠ *e* ⟹ *e′* ≠ (*snd e* , *fst e*) ⟹ *P e es* ⟹ *P e* (*e′* # *es*)
⋀*es*. *P e es* ⟹ *P e* (*e* # *es*)
⋀*es*. *fst e* ≠ *snd e* ⟹ *P e es* ⟹ *P e* ((*snd e*, *fst e*) # *es*)
shows *P e es*
by (*induct es*) (*rule A*, *metis B prod.collapse*)

lemma *co-path-induct*[*case-names empty single co corev pass*]:
fixes *e* :: *′a* × *′a*
and *w* :: *′a*
and *p* :: (*′a* × *′a*) *awalk*
assumes *Nil*: *P e w* []
and *ConsNil*:⋀*e′*. *P e w* [*e′*]
and *ConsCons1*: ⋀*e1 e2 es*. *e1* = (*fst e*, *w*) ∧ *e2* = (*w*, *snd e*) ⟹ *P e w es* ⟹
P e w (*e1* # *e2* # *es*)
and *ConsCons2*: ⋀*e1 e2 es*. ¬(*e1* = (*fst e*, *w*) ∧ *e2* = (*w*, *snd e*)) ∧
e1 = (*snd e*, *w*) ∧ *e2* = (*w*, *fst e*) ⟹ *P e w es* ⟹
P e w (*e1* # *e2* # *es*)
and *ConsCons3*: ⋀*e1 e2 es*.
¬ (*e1* = (*fst e*, *w*) ∧ *e2* = (*w*, *snd e*)) ⟹
¬ (*e1* = (*snd e*, *w*) ∧ *e2* = (*w*, *fst e*)) ⟹ *P e w* (*e2* # *es*) ⟹
P e w (*e1* # *e2* # *es*)
shows *P e w p*
proof (*induct p rule*: *length-induct*)
case (*1 p*) **then show** *?case*
proof (*cases p rule*: *list-exhaust-NSC*)
case (*Cons-Cons e1 e2 es*)
then have *P e w es P e w* (*e2* # *es*)**using** *1* **by** *auto*
then show *?thesis* **unfolding** *Cons-Cons* **by** (*blast intro*: *ConsCons1 ConsCons2 ConsCons3*)
qed (*auto intro*: *Nil ConsNil*)
qed

lemma *co-sd-id*:
assumes (*u*,*w*) ∉ *set p* (*v*,*w*) ∉ *set p*
shows *co-path* (*u*,*v*) *w* (*sd-path* (*u*,*v*) *w p*) = *p*

using *assms* **by** (*induct p*) *auto*

lemma *sd-path-id*:
assumes (*x*,*y*) ∉ *set p* (*y*,*x*) ∉ *set p*
shows *sd-path* (*x*,*y*) *w p* = *p*
using *assms* **by** (*induct p*) *auto*

lemma (**in** *pair-wf-digraph*) *pair-wf-digraph-subdivide*:
assumes *props*: *e* ∈ *parcs G w* ∉ *pverts G*
shows *pair-wf-digraph* (*subdivide G e w*) (**is** *pair-wf-digraph ?sG*)
proof
obtain *u v* **where** [*simp*]: *e* = (*u*,*v*) **by** (*cases e*) *auto*
fix *e'* **assume** *e'* ∈ *parcs ?sG*
then show *fst e'* ∈ *pverts ?sG snd e'* ∈ *pverts ?sG*
using *props* **by** (*auto dest*: *wellformed*)
qed

lemma (**in** *pair-sym-digraph*) *pair-sym-digraph-subdivide*:
assumes *props*: *e* ∈ *parcs G w* ∉ *pverts G*
shows *pair-sym-digraph* (*subdivide G e w*) (**is** *pair-sym-digraph ?sG*)
proof −
interpret *sdG*: *pair-wf-digraph subdivide G e w* **using** *assms* **by** (*rule pair-wf-digraph-subdivide*)
obtain *u v* **where** [*simp*]: *e* = (*u*,*v*) **by** (*cases e*) *auto*
show *?thesis*
proof
have ⋀*a b*. (*a*, *b*) ∈ *parcs* (*subdivide G e w*) ⟹ (*b*, *a*) ∈ *parcs* (*subdivide G e w*)
unfolding ‹*e* = -› *arcs-subdivide*
by (*elim UnE*, *rule UnI1*, *rule-tac* [*2*] *UnI2*) (*blast intro*: *arcs-symmetric*)+
then show *symmetric ?sG*
unfolding *symmetric-def with-proj-simps* **by** (*rule symI*)
qed
qed

lemma (**in** *pair-loopfree-digraph*) *pair-loopfree-digraph-subdivide*:
assumes *props*: *e* ∈ *parcs G w* ∉ *pverts G*
shows *pair-loopfree-digraph* (*subdivide G e w*) (**is** *pair-loopfree-digraph ?sG*)
proof −
interpret *sdG*: *pair-wf-digraph subdivide G e w* **using** *assms* **by** (*rule pair-wf-digraph-subdivide*)
from *assms* **show** *?thesis*
by *unfold-locales* (*cases e*, *auto dest*: *wellformed no-loops*)
qed

lemma (**in** *pair-bidirected-digraph*) *pair-bidirected-digraph-subdivide*:
assumes *props*: *e* ∈ *parcs G w* ∉ *pverts G*
shows *pair-bidirected-digraph* (*subdivide G e w*) (**is** *pair-bidirected-digraph ?sG*)
proof −
interpret *sdG*: *pair-sym-digraph subdivide G e w* **using** *assms* **by** (*rule pair-sym-digraph-subdivide*)
interpret *sdG*: *pair-loopfree-digraph subdivide G e w* **using** *assms* **by** (*rule*

pair-loopfree-digraph-subdivide)
show *?thesis* **by** *unfold-locales*
qed

lemma (**in** *pair-pseudo-graph*) *pair-pseudo-graph-subdivide*:
assumes *props*: $e \in parcs\ G\ w \notin pverts\ G$
shows *pair-pseudo-graph* (*subdivide G e w*) (**is** *pair-pseudo-graph ?sG*)
proof −
interpret *sdG*: *pair-sym-digraph subdivide G e w* **using** *assms* **by** (*rule pair-sym-digraph-subdivide*)
obtain *u v* **where** [*simp*]: $e = (u,v)$ **by** (*cases e*) *auto*
show *?thesis* **by** *unfold-locales* (*cases e*, *auto*)
qed

lemma (**in** *pair-graph*) *pair-graph-subdivide*:
assumes $e \in parcs\ G\ w \notin pverts\ G$
shows *pair-graph* (*subdivide G e w*) (**is** *pair-graph ?sG*)
proof −
interpret *PPG*: *pair-pseudo-graph subdivide G e w*
using *assms* **by** (*rule pair-pseudo-graph-subdivide*)
interpret *PPG*: *pair-loopfree-digraph subdivide G e w*
using *assms* **by** (*rule pair-loopfree-digraph-subdivide*)
from *assms* **show** *?thesis* **by** *unfold-locales*
qed

lemma *arcs-subdivideD*:
assumes $x \in parcs$ (*subdivide G e w*) $fst\ x \neq w\ snd\ x \neq w$
shows $x \in parcs\ G$
using *assms* **by** (*cases e*) *auto*

context *pair-sym-digraph* **begin**

lemma
assumes *path*: *apath u p v*
assumes *elems*: $e \in parcs\ G\ w \notin pverts\ G$
shows *apath-sd-path*: *pre-digraph.apath* (*subdivide G e w*) *u* (*sd-path e w p*) *v* (**is** *?A*)
and *set-awalk-verts-sd-path*: *set* (*awalk-verts u* (*sd-path e w p*))
$\subseteq$ *set* (*awalk-verts u p*) $\cup$ $\{w\}$ (**is** *?B*)
proof −
obtain *x y* **where** *e-conv*: $e = (x,y)$ **by** (*cases e*) *auto*
define *sG* **where** *sG* = *subdivide G e w*
interpret *S*: *pair-sym-digraph sG*
unfolding *sG-def* **using** *elems* **by** (*rule pair-sym-digraph-subdivide*)

have *ev-sG*: *S.awalk-verts* = *awalk-verts*
by (*auto simp*: *fun-eq-iff pre-digraph.awalk-verts-conv*)
have *w-sG*: $\{(x,w), (y,w), (w,x), (w,y)\} \subseteq parcs\ sG$
by (*auto simp*: *sG-def e-conv*)

from *path* **have** *S.apath u* (*sd-path* (*x*,*y*) *w p*) *v*
and *set* (*S.awalk-verts u* (*sd-path* (*x*,*y*) *w p*)) ⊆ *set* (*awalk-verts u p*) ∪ {*w*}
proof (*induct p arbitrary*: *u rule*: *sd-path-induct*)
case *empty* **case** *1*
moreover have *pverts sG* = *pverts G* ∪ {*w*} **by** (*simp add*: *sG-def*)
ultimately show *?case* **by** (*auto simp*: *apath-Nil-iff S.apath-Nil-iff*)
next
case *empty* **case** *2* **then show** *?case* **by** *simp*
next
case (*pass e′ es*)
{ case *1*
then have *S.apath* (*snd e′*) (*sd-path* (*x*,*y*) *w es*) *v u* ≠ *w fst e′* = *u*
u ∉ *set* (*S.awalk-verts* (*snd e′*) (*sd-path* (*x*,*y*) *w es*))
using *pass elems* **by** (*fastforce simp*: *apath-Cons-iff*)+
moreover then have *e′* ∈ *parcs sG*
using *1 pass* **by** (*auto simp*: *e-conv sG-def S.apath-Cons-iff apath-Cons-iff*)
ultimately show *?case* **using** *pass* **by** (*auto simp*: *S.apath-Cons-iff*) **}**
note *case1* = *this*
{ case *2* **with** *pass 2* **show** *?case* **by** (*simp add*: *apath-Cons-iff*) *blast* **}**
next
{ fix *u es a b*
assume *A*: *apath u* ((*a*,*b*) # *es*) *v*
and *ab*: (*a*,*b*) = (*x*,*y*) ∨ (*a*,*b*) = (*y*,*x*)
and *hyps*: ⋀*u*. *apath u es v* ⟹ *S.apath u* (*sd-path* (*x*, *y*) *w es*) *v*
⋀*u*. *apath u es v* ⟹ *set* (*awalk-verts u* (*sd-path* (*x*, *y*) *w es*)) ⊆ *set* (*awalk-verts u es*) ∪ {*w*}

from *ab A* **have** (*x*,*y*) ∉ *set es* (*y*,*x*) ∉ *set es*
by (*auto simp*: *apath-Cons-iff dest*!: *awalkI-apath dest*: *awalk-verts-arc1 awalk-verts-arc2*)
then have *ev-sd*: *set* (*S.awalk-verts b* (*sd-path* (*x*,*y*) *w es*)) = *set* (*awalk-verts b es*)
by (*simp add*: *sd-path-id*)

from *A ab* **have** [*simp*]: *x* ≠ *y*
by (*simp add*: *apath-Cons-iff*) (*metis awalkI-apath awalk-verts-non-Nil awhd-of-awalk hd-in-set*)

from *A* **have** *S.apath b* (*sd-path* (*x*,*y*) *w es*) *v u* = *a u* ≠ *w*
using *ab hyps elems* **by** (*auto simp*: *apath-Cons-iff wellformed′*)
moreover
then have *S.awalk u* (*sd-path* (*x*, *y*) *w* ((*a*, *b*) # *es*)) *v*
using *ab w-sG* **by** (*auto simp*: *S.apath-def S.awalk-simps S.wellformed′*)
then have *u* ∉ *set* (*S.awalk-verts w* ((*w*,*b*) # *sd-path* (*x*,*y*) *w es*))
using *ab* ‹*u* ≠ *w*› *ev-sd A* **by** (*auto simp*: *apath-Cons-iff S.awalk-def*)
moreover
have *w* ∉ *set* (*awalk-verts b* (*sd-path* (*x*, *y*) *w es*))
using *ab ev-sd A elems* **by** (*auto simp*: *awalk-Cons-iff apath-def*)
ultimately

```
    have path: S.apath u (sd-path (x, y) w ((a, b) # es)) v
      using ab hyps w-sG ‹u = a› by (auto simp: S.apath-Cons-iff ) }
   note path = this
   { case (sd es)
     { case 1 with sd show ?case by (intro path) auto }
     { case 2 show ?case using 2 sd
        by (auto simp: apath-Cons-iff) } }
   { case (sdrev es)
     { case 1 with sdrev show ?case by (intro path) auto }
     { case 2 show ?case using 2 sdrev
        by (auto simp: apath-Cons-iff) } }
 qed
 then show ?A ?B unfolding sG-def e-conv .
qed

lemma
 assumes elems: e ∈ parcs G w ∉ pverts G u ∈ pverts G v ∈ pverts G
 assumes path: pre-digraph.apath (subdivide G e w) u p v
 shows apath-co-path: apath u (co-path e w p) v (is ?thesis-path)
  and set-awalk-verts-co-path: set (awalk-verts u (co-path e w p)) = set (awalk-verts
u p) − {w} (is ?thesis-set)
proof −
 obtain x y where e-conv: e = (x,y) by (cases e) auto
 interpret S: pair-sym-digraph subdivide G e w
   using elems(1,2) by (rule pair-sym-digraph-subdivide)

 have e-w: fst e ≠ w snd e ≠ w using elems by auto

 have S.apath u p v u ≠ w using elems path by auto
 then have co-path: apath u (co-path e w p) v
   ∧ set (awalk-verts u (co-path e w p)) = set (awalk-verts u p) − {w}
 proof (induction p arbitrary: u rule: co-path-induct)
   case empty with elems show ?case
     by (simp add: apath-Nil-iff S.apath-Nil-iff)
 next
   case (single e′) with elems show ?case
    by (auto simp: apath-Cons-iff S.apath-Cons-iff apath-Nil-iff S.apath-Nil-iff
       dest: arcs-subdivideD)
 next
   case (co e1 e2 es)
   then have apath u (co-path e w (e1 # e2 # es)) v using co e-w elems
     by (auto simp: apath-Cons-iff S.apath-Cons-iff)
   moreover
    have set (awalk-verts u (co-path e w (e1 # e2 # es))) = set (awalk-verts u
(e1 # e2 # es)) − {w}
     using co e-w by (auto simp: apath-Cons-iff S.apath-Cons-iff)
   ultimately
   show ?case by fast
 next
```

```
    case (corev e1 e2 es)
   have apath u (co-path e w (e1 # e2 # es)) v using corev(1−3) e-w(1) elems(1)
     by (auto simp: apath-Cons-iff S.apath-Cons-iff  intro: arcs-symmetric)
    moreover
     have set (awalk-verts u (co-path e w (e1 # e2 # es))) = set (awalk-verts u
(e1 # e2 # es)) − {w}
      using corev e-w by (auto simp: apath-Cons-iff S.apath-Cons-iff)
    ultimately
    show ?case by fast
  next
    case (pass e1 e2 es)
    have fst e1 ≠ w using elems pass.prems by (auto simp: S.apath-Cons-iff)
    have snd e1 ≠ w
    proof
      assume snd e1 = w
      then have e1 ∉ parcs G using elems by auto
      then have e1 ∈ parcs (subdivide G e w) − parcs G
        using pass by (auto simp: S.apath-Cons-iff)
      then have e1 = (x,w) ∨ e1 = (y,w)
        using ‹fst e1 ≠ w› e-w by (auto simp add: e-conv)
      moreover
     have fst e2 = w using ‹snd e1 = w› pass.prems by (auto simp: S.apath-Cons-iff)
      then have e2 ∉ parcs G using elems by auto
      then have e2 ∈ parcs (subdivide G e w) − parcs G
        using pass by (auto simp: S.apath-Cons-iff)
      then have e2 = (w,x) ∨ e2 = (w,y)
        using ‹fst e2 = w› e-w by (cases e2) (auto simp add: e-conv)
      ultimately
      have e1 = (x,w) ∧ e2 = (w,x) ∨ e1 = (y,w) ∧ e2 = (w,y)
        using pass.hyps[simplified e-conv] by auto
      then show False
        using pass.prems by (cases es) (auto simp: S.apath-Cons-iff)
    qed
    then have e1 ∈ parcs G
    using ‹fst e1 ≠ w› pass.prems by (auto simp: S.apath-Cons-iff dest: arcs-subdivideD)

    have ih: apath (snd e1) (co-path e w (e2 # es)) v ∧ set (awalk-verts (snd e1)
(co-path e w (e2 # es))) = set (awalk-verts (snd e1) (e2 # es)) − {w}
     using pass.prems ‹snd e1 ≠ w› by (intro pass.IH) (auto simp: apath-Cons-iff
S.apath-Cons-iff)
    then have fst e1 ∉ set (awalk-verts (snd e1) (co-path e w (e2 # es))) fst e1
= u
     using pass.prems by (clarsimp simp: S.apath-Cons-iff)+
    then have apath u (co-path e w (e1 # e2 # es)) v
     using ih pass ‹e1 ∈ parcs G› by (auto simp: apath-Cons-iff S.apath-Cons-iff)[]
    moreover
     have set (awalk-verts u (co-path e w (e1 # e2 # es))) = set (awalk-verts u
(e1 # e2 # es)) − {w}
      using pass.hyps ih ‹fst e1 ≠ w› by auto
```

ultimately show *?case* **by** *fast*
qed
then show *?thesis-set ?thesis-path* **by** *blast+*
qed

end

7.3 Bidirected Graphs

definition (**in** −) *swap-in* :: (′*a* × ′*a*) *set* ⇒ ′*a* × ′*a* ⇒ ′*a* × ′*a* **where**
swap-in S x = (*if x* ∈ *S then prod.swap x else x*)

lemma *bidirected-digraph-rev-conv-pair*:
assumes *bidirected-digraph* (*with-proj G*) *rev-G*
shows *rev-G* = *swap-in* (*parcs G*)
proof −
interpret *bidirected-digraph G rev-G* **by** *fact*
have ⋀*a b*. (*a*, *b*) ∈ *parcs G* ⟹ *rev-G* (*a*, *b*) = (*b*, *a*)
using *tail-arev*[*simplified with-proj-simps*] *head-arev*[*simplified with-proj-simps*]
by (*metis fst-conv prod.collapse snd-conv*)
then show *?thesis* **by** (*auto simp*: *swap-in-def fun-eq-iff arev-eq*)
qed

lemma (**in** *pair-bidirected-digraph*) *bidirected-digraph*:
bidirected-digraph (*with-proj G*) (*swap-in* (*parcs G*))
using *no-loops*′ *arcs-symmetric*
by *unfold-locales* (*auto simp*: *swap-in-def*)

lemma *pair-bidirected-digraphI-bidirected-digraph*:
assumes *bidirected-digraph* (*with-proj G*) (*swap-in* (*parcs G*))
shows *pair-bidirected-digraph G*
proof −
interpret *bidirected-digraph with-proj G swap-in* (*parcs G*) **by** *fact*
{
fix *a* **assume** *a* ∈ *parcs G* **then have** *fst a* ≠ *snd a*
using *arev-neq*[*of a*] *bidirected-digraph-rev-conv-pair*[*OF assms*(*1*)]
by (*cases a*) (*auto simp*: *swap-in-def*)
}
then show *?thesis*
using *tail-in-verts head-in-verts* **by** *unfold-locales auto*
qed

end

theory *Digraph-Component*
imports
Digraph
Arc-Walk

Pair-Digraph
begin

8 Components of (Symmetric) Digraphs

definition *compatible* :: ('a,'b) *pre-digraph* ⇒ ('a,'b) *pre-digraph* ⇒ *bool* **where**
compatible G H ≡ *tail G* = *tail H* ∧ *head G* = *head H*

definition *subgraph* :: ('a,'b) *pre-digraph* ⇒ ('a,'b) *pre-digraph* ⇒ *bool* **where**
subgraph H G ≡ *verts H* ⊆ *verts G* ∧ *arcs H* ⊆ *arcs G* ∧ *wf-digraph G* ∧ *wf-digraph H* ∧ *compatible G H*

definition *induced-subgraph* :: ('a,'b) *pre-digraph* ⇒ ('a,'b) *pre-digraph* ⇒ *bool* **where**
induced-subgraph H G ≡ *subgraph H G* ∧ *arcs H* = {*e* ∈ *arcs G*. *tail G e* ∈ *verts H* ∧ *head G e* ∈ *verts H*}

definition *spanning* :: ('a,'b) *pre-digraph* ⇒ ('a,'b) *pre-digraph* ⇒ *bool* **where**
spanning H G ≡ *subgraph H G* ∧ *verts G* = *verts H*

definition *strongly-connected* :: ('a,'b) *pre-digraph* ⇒ *bool* **where**
strongly-connected G ≡ *verts G* ≠ {} ∧ (∀ *u* ∈ *verts G*. ∀ *v* ∈ *verts G*. *u* →** *G v*)

The following function computes underlying symmetric graph of a digraph and removes parallel arcs.

definition *mk-symmetric* :: ('a,'b) *pre-digraph* ⇒ 'a *pair-pre-digraph* **where**
mk-symmetric G ≡ (| *pverts* = *verts G*, *parcs* = ⋃ *e*∈*arcs G*. {(*tail G e*, *head G e*), (*head G e*, *tail G e*)}|)

definition *connected* :: ('a,'b) *pre-digraph* ⇒ *bool* **where**
connected G ≡ *strongly-connected* (*mk-symmetric G*)

definition *forest* :: ('a,'b) *pre-digraph* ⇒ *bool* **where**
forest G ≡ ¬(∃ *p*. *pre-digraph.cycle G p*)

definition *tree* :: ('a,'b) *pre-digraph* ⇒ *bool* **where**
tree G ≡ *connected G* ∧ *forest G*

definition *spanning-tree* :: ('a,'b) *pre-digraph* ⇒ ('a,'b) *pre-digraph* ⇒ *bool* **where**
spanning-tree H G ≡ *tree H* ∧ *spanning H G*

definition (**in** *pre-digraph*)
max-subgraph :: (('a,'b) *pre-digraph* ⇒ *bool*) ⇒ ('a,'b) *pre-digraph* ⇒ *bool* **where**
max-subgraph P H ≡ *subgraph H G* ∧ *P H* ∧ (∀ *H'*. *H'* ≠ *H* ∧ *subgraph H H'* ⟶ ¬(*subgraph H' G* ∧ *P H'*))

definition (**in** *pre-digraph*) *sccs* :: ('a,'b) *pre-digraph set* **where**

sccs ≡ {*H*. *induced-subgraph H G* ∧ *strongly-connected H* ∧ ¬(∃ *H'*. *induced-subgraph H' G*
∧ *strongly-connected H'* ∧ *verts H* ⊂ *verts H'*)}

definition (**in** *pre-digraph*) *sccs-verts* :: *'a set set* **where**
sccs-verts = {*S*. *S* ≠ {} ∧ (∀ *u* ∈ *S*. ∀ *v* ∈ *S*. *u* →**G v*) ∧ (∀ *u* ∈ *S*. ∀ *v*. *v* ∉ *S* ⟶ ¬*u* →**G v* ∨ ¬*v* →**G u*)}

definition (**in** *pre-digraph*) *scc-of* :: *'a* ⇒ *'a set* **where**
scc-of u ≡ {*v*. *u* →* *v* ∧ *v* →* *u*}

definition *union* :: ('*a*,'*b*) *pre-digraph* ⇒ ('*a*,'*b*) *pre-digraph* ⇒ ('*a*,'*b*) *pre-digraph* **where**
union G H ≡ ⦇ *verts* = *verts G* ∪ *verts H*, *arcs* = *arcs G* ∪ *arcs H*, *tail* = *tail G*, *head* = *head G*⦈

definition (**in** *pre-digraph*) *Union* :: ('*a*,'*b*) *pre-digraph set* ⇒ ('*a*,'*b*) *pre-digraph* **where**
Union gs = ⦇ *verts* = (⋃ *G* ∈ *gs*. *verts G*), *arcs* = (⋃ *G* ∈ *gs*. *arcs G*),
tail = *tail G* , *head* = *head G* ⦈

8.1 Compatible Graphs

lemma *compatible-tail*:
assumes *compatible G H* **shows** *tail G* = *tail H*
using *assms* **by** (*simp add*: *fun-eq-iff compatible-def*)

lemma *compatible-head*:
assumes *compatible G H* **shows** *head G* = *head H*
using *assms* **by** (*simp add*: *fun-eq-iff compatible-def*)

lemma *compatible-cas*:
assumes *compatible G H* **shows** *pre-digraph.cas G* = *pre-digraph.cas H*
proof (*unfold fun-eq-iff*, *intro allI*)
fix *u es v* **show** *pre-digraph.cas G u es v* = *pre-digraph.cas H u es v*
using *assms*
by (*induct es arbitrary*: *u*)
(*simp-all add*: *pre-digraph.cas.simps compatible-head compatible-tail*)
qed

lemma *compatible-awalk-verts*:
assumes *compatible G H* **shows** *pre-digraph.awalk-verts G* = *pre-digraph.awalk-verts H*
proof (*unfold fun-eq-iff*, *intro allI*)
fix *u es* **show** *pre-digraph.awalk-verts G u es* = *pre-digraph.awalk-verts H u es*
using *assms*
by (*induct es arbitrary*: *u*)
(*simp-all add*: *pre-digraph.awalk-verts.simps compatible-head compatible-tail*)

qed

lemma *compatibleI-with-proj*[*intro*]:
shows *compatible* (*with-proj G*) (*with-proj H*)
by (*auto simp*: *compatible-def*)

8.2 Basic lemmas

lemma (**in** *sym-digraph*) *graph-symmetric*:
shows (*u*,*v*) ∈ *arcs-ends G* ⟹ (*v*,*u*) ∈ *arcs-ends G*
using *sym-arcs* **by** (*auto simp add*: *symmetric-def sym-def*)

lemma *strongly-connectedI*[*intro*]:
assumes *verts G* ≠ {} ⋀*u v*. *u* ∈ *verts G* ⟹ *v* ∈ *verts G* ⟹ *u* →** G v*
shows *strongly-connected G*
using *assms* **by** (*simp add*: *strongly-connected-def*)

lemma *strongly-connectedE*[*elim*]:
assumes *strongly-connected G*
assumes (⋀*u v*. *u* ∈ *verts G* ∧ *v* ∈ *verts G* ⟹ *u* →** G v*) ⟹ *P*
shows *P*
using *assms* **by** (*auto simp add*: *strongly-connected-def*)

lemma *subgraph-imp-subverts*:
assumes *subgraph H G*
shows *verts H* ⊆ *verts G*
using *assms* **by** (*simp add*: *subgraph-def*)

lemma *induced-imp-subgraph*:
assumes *induced-subgraph H G*
shows *subgraph H G*
using *assms* **by** (*simp add*: *induced-subgraph-def*)

lemma (**in** *pre-digraph*) *in-sccs-imp-induced*:
assumes *c* ∈ *sccs*
shows *induced-subgraph c G*
using *assms* **by** (*auto simp*: *sccs-def*)

lemma *spanning-tree-imp-tree*[*dest*]:
assumes *spanning-tree H G*
shows *tree H*
using *assms* **by** (*simp add*: *spanning-tree-def*)

lemma *tree-imp-connected*[*dest*]:
assumes *tree G*
shows *connected G*
using *assms* **by** (*simp add*: *tree-def*)

lemma *spanning-treeI*[*intro*]:

assumes *spanning H G*
assumes *tree H*
shows *spanning-tree H G*
using *assms* **by** (*simp add*: *spanning-tree-def*)

lemma *spanning-treeE*[*elim*]:
assumes *spanning-tree H G*
assumes *tree H* ∧ *spanning H G* ⟹ *P*
shows *P*
using *assms* **by** (*simp add*: *spanning-tree-def*)

lemma *spanningE*[*elim*]:
assumes *spanning H G*
assumes *subgraph H G* ∧ *verts G* = *verts H* ⟹ *P*
shows *P*
using *assms* **by** (*simp add*: *spanning-def*)

lemma (**in** *pre-digraph*) *in-sccsI*[*intro*]:
assumes *induced-subgraph c G*
assumes *strongly-connected c*
assumes ¬(∃ *c'*. *induced-subgraph c' G* ∧ *strongly-connected c'* ∧
verts c ⊂ *verts c'*)
shows *c* ∈ *sccs*
using *assms* **by** (*auto simp add*: *sccs-def*)

lemma (**in** *pre-digraph*) *in-sccsE*[*elim*]:
assumes *c* ∈ *sccs*
assumes *induced-subgraph c G* ⟹ *strongly-connected c* ⟹ ¬ (∃ *d*.
induced-subgraph d G ∧ *strongly-connected d* ∧ *verts c* ⊂ *verts d*) ⟹ *P*
shows *P*
using *assms* **by** (*simp add*: *sccs-def*)

lemma *subgraphI*:
assumes *verts H* ⊆ *verts G*
assumes *arcs H* ⊆ *arcs G*
assumes *compatible G H*
assumes *wf-digraph H*
assumes *wf-digraph G*
shows *subgraph H G*
using *assms* **by** (*auto simp add*: *subgraph-def*)

lemma *subgraphE*[*elim*]:
assumes *subgraph H G*
obtains *verts H* ⊆ *verts G arcs H* ⊆ *arcs G compatible G H wf-digraph H*
wf-digraph G
using *assms* **by** (*simp add*: *subgraph-def*)

lemma *induced-subgraphI*[*intro*]:
assumes *subgraph H G*

assumes *arcs H* = {*e* ∈ *arcs G*. *tail G e* ∈ *verts H* ∧ *head G e* ∈ *verts H*}
shows *induced-subgraph H G*
using *assms* **unfolding** *induced-subgraph-def* **by** *safe*

lemma *induced-subgraphE*[*elim*]:
assumes *induced-subgraph H G*
assumes ⟦*subgraph H G*; *arcs H* = {*e* ∈ *arcs G*. *tail G e* ∈ *verts H* ∧ *head G e* ∈ *verts H*}⟧ ⟹ *P*
shows *P*
using *assms* **by** (*auto simp add*: *induced-subgraph-def*)

lemma *pverts-mk-symmetric*[*simp*]: *pverts* (*mk-symmetric G*) = *verts G*
and *parcs-mk-symmetric*:
parcs (*mk-symmetric G*) = (⋃ *e*∈*arcs G*. {(*tail G e*, *head G e*), (*head G e*, *tail G e*)})
by (*auto simp*: *mk-symmetric-def arcs-ends-conv image-UN*)

lemma *arcs-ends-mono*:
assumes *subgraph H G*
shows *arcs-ends H* ⊆ *arcs-ends G*
using *assms* **by** (*auto simp add*: *subgraph-def arcs-ends-conv compatible-tail compatible-head*)

lemma (**in** *wf-digraph*) *subgraph-refl*: *subgraph G G*
by (*auto simp*: *subgraph-def compatible-def*) *unfold-locales*

lemma (**in** *wf-digraph*) *induced-subgraph-refl*: *induced-subgraph G G*
by (*rule induced-subgraphI*) (*auto simp*: *subgraph-refl*)

8.3 The underlying symmetric graph of a digraph

lemma (**in** *wf-digraph*) *wellformed-mk-symmetric*[*intro*]: *pair-wf-digraph* (*mk-symmetric G*)
by *unfold-locales* (*auto simp*: *parcs-mk-symmetric*)

lemma (**in** *fin-digraph*) *pair-fin-digraph-mk-symmetric*[*intro*]: *pair-fin-digraph* (*mk-symmetric G*)
proof −
have *finite* ((λ(*a*,*b*). (*b*,*a*)) ‘ *arcs-ends G*) (**is** *finite ?X*) **by** (*auto simp*: *arcs-ends-conv*)
also have *?X* = {(*a*, *b*). (*b*, *a*) ∈ *arcs-ends G*} **by** *auto*
finally have *X*: *finite* ... .
then show *?thesis*
by *unfold-locales* (*auto simp*: *mk-symmetric-def arcs-ends-conv*)
qed

lemma (**in** *digraph*) *digraph-mk-symmetric*[*intro*]: *pair-digraph* (*mk-symmetric G*)
proof −
have *finite* ((λ(*a*,*b*). (*b*,*a*)) ‘ *arcs-ends G*) (**is** *finite ?X*) **by** (*auto simp*: *arcs-ends-conv*)
also have *?X* = {(*a*, *b*). (*b*, *a*) ∈ *arcs-ends G*} **by** *auto*

finally have *finite ...* **.**
then show *?thesis*
by *unfold-locales* (*auto simp*: *mk-symmetric-def arc-to-ends-def dest*: *no-loops*)
qed

lemma (**in** *wf-digraph*) *reachable-mk-symmetricI*:
assumes $u \rightarrow^* v$ **shows** $u \rightarrow^*_{\textit{mk-symmetric } G} v$
proof −
have *arcs-ends G* ⊆ *parcs* (*mk-symmetric G*)
$(u, v) \in$ *rtrancl-on* (*pverts* (*mk-symmetric G*)) (*arcs-ends G*)
using *assms* **unfolding** *reachable-def* **by** (*auto simp*: *parcs-mk-symmetric*)
then show *?thesis* **unfolding** *reachable-def* **by** (*auto intro*: *rtrancl-on-mono*)
qed

lemma (**in** *wf-digraph*) *adj-mk-symmetric-eq*:
symmetric G ⟹ *parcs* (*mk-symmetric G*) = *arcs-ends G*
by (*auto simp*: *parcs-mk-symmetric in-arcs-imp-in-arcs-ends arcs-ends-symmetric*)

lemma (**in** *wf-digraph*) *reachable-mk-symmetric-eq*:
assumes *symmetric G* **shows** $u \rightarrow^*_{\textit{mk-symmetric } G} v \longleftrightarrow u \rightarrow^* v$ (**is** *?L* ⟷ *?R*)
using *adj-mk-symmetric-eq*[*OF assms*] **unfolding** *reachable-def* **by** *auto*

lemma (**in** *wf-digraph*) *mk-symmetric-awalk-imp-awalk*:
assumes *sym*: *symmetric G*
assumes *walk*: *pre-digraph.awalk* (*mk-symmetric G*) *u p v*
obtains *q* **where** *awalk u q v*
proof −
interpret *S*: *pair-wf-digraph mk-symmetric G* **..**
from *walk* **have** $u \rightarrow^*_{\textit{mk-symmetric } G} v$
by (*simp only*: *S.reachable-awalk*) *rule*
then have $u \rightarrow^* v$ **by** (*simp only*: *reachable-mk-symmetric-eq*[*OF sym*])
then show *?thesis* **by** (*auto simp*: *reachable-awalk intro*: *that*)
qed

lemma *symmetric-mk-symmetric*:
symmetric (*mk-symmetric G*)
by (*auto simp*: *symmetric-def parcs-mk-symmetric intro*: *symI*)

8.4 Subgraphs and Induced Subgraphs

lemma *subgraph-trans*:
assumes *subgraph G H subgraph H I* **shows** *subgraph G I*
using *assms* **by** (*auto simp*: *subgraph-def compatible-def*)

The *digraph* and *fin-digraph* properties are preserved under the (inverse) subgraph relation

lemma (**in** *fin-digraph*) *fin-digraph-subgraph*:
assumes *subgraph H G* **shows** *fin-digraph H*

proof (*intro-locales*)
from *assms* **show** *wf-digraph H* **by** *auto*

have *HG*: *arcs H* ⊆ *arcs G verts H* ⊆ *verts G*
using *assms* **by** *auto*
then have *finite* (*verts H*) *finite* (*arcs H*)
using *finite-verts finite-arcs* **by** (*blast intro*: *finite-subset*)+
then show *fin-digraph-axioms H*
by *unfold-locales*
qed

lemma (**in** *digraph*) *digraph-subgraph*:
assumes *subgraph H G* **shows** *digraph H*
proof
fix *e* **assume** *e*: *e* ∈ *arcs H*
with *assms* **show** *tail H e* ∈ *verts H head H e* ∈ *verts H*
by (*auto simp*: *subgraph-def intro*: *wf-digraph.wellformed*)
from *e* **and** *assms* **have** *e* ∈ *arcs H* ∩ *arcs G* **by** *auto*
with *assms* **show** *tail H e* ≠ *head H e*
using *no-loops* **by** (*auto simp*: *subgraph-def compatible-def arc-to-ends-def*)
next
have *arcs H* ⊆ *arcs G verts H* ⊆ *verts G* **using** *assms* **by** *auto*
then show *finite* (*arcs H*) *finite* (*verts H*)
using *finite-verts finite-arcs* **by** (*blast intro*: *finite-subset*)+
next
fix *e1 e2* **assume** *e1* ∈ *arcs H e2* ∈ *arcs H*
and *eq*: *arc-to-ends H e1* = *arc-to-ends H e2*
with *assms* **have** *e1* ∈ *arcs H* ∩ *arcs G e2* ∈ *arcs H* ∩ *arcs G*
by *auto*
with *eq* **show** *e1* = *e2*
using *no-multi-arcs assms*
by (*auto simp*: *subgraph-def compatible-def arc-to-ends-def*)
qed

lemma (**in** *pre-digraph*) *adj-mono*:
assumes *u* →$_H$ *v subgraph H G*
shows *u* → *v*
using *assms* **by** (*blast dest*: *arcs-ends-mono*)

lemma (**in** *pre-digraph*) *reachable-mono*:
assumes *walk*: *u* →*$^*_H$ *v* **and** *sub*: *subgraph H G*
shows *u* →* *v*
proof −
have *verts H* ⊆ *verts G* **using** *sub* **by** *auto*
with *assms* **show** *?thesis*
unfolding *reachable-def* **by** (*metis arcs-ends-mono rtrancl-on-mono*)
qed

Arc walks and paths are preserved under the subgraph relation.

lemma (**in** *wf-digraph*) *subgraph-awalk-imp-awalk*:
assumes *walk*: *pre-digraph.awalk H u p v*
assumes *sub*: *subgraph H G*
shows *awalk u p v*
using *assms* **by** (*auto simp*: *pre-digraph.awalk-def compatible-cas*)

lemma (**in** *wf-digraph*) *subgraph-apath-imp-apath*:
assumes *path*: *pre-digraph.apath H u p v*
assumes *sub*: *subgraph H G*
shows *apath u p v*
using *assms* **unfolding** *pre-digraph.apath-def*
by (*auto intro*: *subgraph-awalk-imp-awalk simp*: *compatible-awalk-verts*)

lemma *subgraph-mk-symmetric*:
assumes *subgraph H G*
shows *subgraph* (*mk-symmetric H*) (*mk-symmetric G*)
proof (*rule subgraphI*)
let *?wpms* = λG. *mk-symmetric G*
from *assms* **have** *compatible G H* **by** *auto*
with *assms*
show *verts* (*?wpms H*) $\subseteq$ *verts* (*?wpms G*)
and *arcs* (*?wpms H*) $\subseteq$ *arcs* (*?wpms G*)
by (*auto simp*: *parcs-mk-symmetric compatible-head compatible-tail*)
show *compatible* (*?wpms G*) (*?wpms H*) **by** *rule*
interpret *H*: *pair-wf-digraph mk-symmetric H*
using *assms* **by** (*auto intro*: *wf-digraph.wellformed-mk-symmetric*)
interpret *G*: *pair-wf-digraph mk-symmetric G*
using *assms* **by** (*auto intro*: *wf-digraph.wellformed-mk-symmetric*)
show *wf-digraph* (*?wpms H*)
by *unfold-locales*
show *wf-digraph* (*?wpms G*) **by** *unfold-locales*
qed

lemma (**in** *fin-digraph*) *subgraph-in-degree*:
assumes *subgraph H G*
shows *in-degree H v* $\leq$ *in-degree G v*
proof −
have *finite* (*in-arcs G v*) **by** *auto*
moreover
have *in-arcs H v* $\subseteq$ *in-arcs G v*
using *assms* **by** (*auto simp*: *subgraph-def in-arcs-def compatible-head compatible-tail*)
ultimately
show *?thesis* **unfolding** *in-degree-def* **by** (*rule card-mono*)
qed

lemma (**in** *wf-digraph*) *subgraph-cycle*:
assumes *subgraph H G pre-digraph.cycle H p* **shows** *cycle p*
proof −

from *assms* **have** *compatible G H* **by** *auto*
with *assms* **show** *?thesis*
by (*auto simp*: *pre-digraph.cycle-def compatible-awalk-verts intro*: *subgraph-awalk-imp-awalk*)
qed

lemma (**in** *wf-digraph*) *subgraph-del-vert*: *subgraph* (*del-vert u*) *G*
by (*auto simp*: *subgraph-def compatible-def del-vert-simps wf-digraph-del-vert*)
intro-locales

lemma (**in** *wf-digraph*) *subgraph-del-arc*: *subgraph* (*del-arc a*) *G*
by (*auto simp*: *subgraph-def compatible-def del-vert-simps wf-digraph-del-vert*)
intro-locales

8.5 Induced subgraphs

lemma *wf-digraphI-induced*:
assumes *induced-subgraph H G*
shows *wf-digraph H*
proof −
from *assms* **have** *compatible G H* **by** *auto*
with *assms* **show** *?thesis* **by** *unfold-locales* (*auto simp*: *compatible-tail compatible-head*)
qed

lemma (**in** *digraph*) *digraphI-induced*:
assumes *induced-subgraph H G*
shows *digraph H*
proof −
interpret *W*: *wf-digraph H* **using** *assms* **by** (*rule wf-digraphI-induced*)
from *assms* **have** *compatible G H* **by** *auto*
from *assms* **have** *arcs*: *arcs H* ⊆ *arcs G* **by** *blast*
show *?thesis*
proof
from *assms* **have** *verts H* ⊆ *verts G* **by** *blast*
then show *finite* (*verts H*) **using** *finite-verts* **by** (*rule finite-subset*)
next
from *arcs* **show** *finite* (*arcs H*) **using** *finite-arcs* **by** (*rule finite-subset*)
next
fix *e* **assume** *e* ∈ *arcs H*
with *arcs* ‹*compatible G H*› **show** *tail H e* ≠ *head H e*
by (*auto dest*: *no-loops simp*: *compatible-tail*[*symmetric*] *compatible-head*[*symmetric*])
next
fix *e1 e2* **assume** *e1* ∈ *arcs H e2* ∈ *arcs H* **and** *ate*: *arc-to-ends H e1* = *arc-to-ends H e2*
with *arcs* ‹*compatible G H*› **show** *e1* = *e2* **using** *ate*
by (*auto intro*: *no-multi-arcs simp*: *compatible-tail*[*symmetric*] *compatible-head*[*symmetric*] *arc-to-ends-def*)
qed
qed

Computes the subgraph of G induced by vs

definition *induce-subgraph* :: ('a,'b) *pre-digraph* ⇒ 'a *set* ⇒ ('a,'b) *pre-digraph* (**infix** ↾ *67*) **where**
  $G \restriction vs = (\!|\ verts = vs,\ arcs = \{e \in arcs\ G.\ tail\ G\ e \in vs \wedge head\ G\ e \in vs\},$
    $tail = tail\ G,\ head = head\ G\ |\!)$

lemma *induce-subgraph-verts*[*simp*]:
 $verts\ (G \restriction vs) = vs$
by (*auto simp add*: *induce-subgraph-def*)

lemma *induce-subgraph-arcs*[*simp*]:
 $arcs\ (G \restriction vs) = \{e \in arcs\ G.\ tail\ G\ e \in vs \wedge head\ G\ e \in vs\}$
by (*auto simp add*: *induce-subgraph-def*)

lemma *induce-subgraph-tail*[*simp*]:
  $tail\ (G \restriction vs) = tail\ G$
by (*auto simp*: *induce-subgraph-def*)

lemma *induce-subgraph-head*[*simp*]:
  $head\ (G \restriction vs) = head\ G$
by (*auto simp*: *induce-subgraph-def*)

lemma *compatible-induce-subgraph*: *compatible* $(G \restriction S)\ G$
  by (*auto simp*: *compatible-def*)

lemma (**in** *wf-digraph*) *induced-induce*[*intro*]:
  assumes $vs \subseteq verts\ G$
  shows *induced-subgraph* $(G \restriction vs)\ G$
using *assms*
by (*intro subgraphI induced-subgraphI*)
   (*auto simp*: *arc-to-ends-def induce-subgraph-def wf-digraph-def compatible-def*)

lemma (**in** *wf-digraph*) *wellformed-induce-subgraph*[*intro*]:
  wf-digraph $(G \restriction vs)$
  by *unfold-locales auto*

lemma *induced-graph-imp-symmetric*:
  assumes *symmetric G*
  assumes *induced-subgraph H G*
  shows *symmetric H*
proof (*unfold symmetric-conv*, *safe*)
  from *assms* **have** *compatible G H* **by** *auto*

  fix *e1* **assume** *e1* ∈ *arcs H*
  then obtain *e2* **where** *tail G e1* = *head G e2 head G e1* = *tail G e2 e2* ∈ *arcs G*
    using *assms* **by** (*auto simp add*: *symmetric-conv*)
  moreover
  then have *e2* ∈ *arcs H*

using *assms* **and** ‹*e1* ∈ *arcs H*› **by** *auto*
ultimately
show ∃ *e2*∈*arcs H*. *tail H e1* = *head H e2* ∧ *head H e1* = *tail H e2*
using *assms* ‹*e1* ∈ *arcs H*› ‹*compatible G H*›
by (*auto simp*: *compatible-head compatible-tail*)
qed

lemma (**in** *sym-digraph*) *induced-graph-imp-graph*:
assumes *induced-subgraph H G*
shows *sym-digraph H*
proof (*rule wf-digraph.sym-digraphI*)
from *assms* **show** *wf-digraph H* **by** (*rule wf-digraphI-induced*)
next
show *symmetric H*
using *assms sym-arcs* **by** (*auto intro*: *induced-graph-imp-symmetric*)
qed

lemma (**in** *wf-digraph*) *induce-reachable-preserves-paths*:
assumes $u \rightarrow^*_G v$
shows $u \rightarrow^*_{G \restriction \{w.\ u \rightarrow^*_G w\}} v$
using *assms*
proof *induct*
case *base* **then show** *?case* **by** (*auto simp*: *reachable-def*)
next
case (*step u w*)
interpret *iG*: *wf-digraph* $G \restriction \{w.\ u \rightarrow^*_G w\}$
by (*rule wellformed-induce-subgraph*)
from ‹$u \rightarrow w$› **have** $u \rightarrow_{G \restriction \{wa.\ u \rightarrow^*_G wa\}} w$
by (*auto simp*: *arcs-ends-conv reachable-def intro*: *wellformed rtrancl-on-into-rtrancl-on*)
then have $u \rightarrow^*_{G \restriction \{wa.\ u \rightarrow^*_G wa\}} w$
by (*rule iG.reachable-adjI*)
moreover
from *step* **have** $\{x.\ w \rightarrow^* x\} \subseteq \{x.\ u \rightarrow^* x\}$
by (*auto intro*: *adj-reachable-trans*)
then have *subgraph* $(G \restriction \{wa.\ w \rightarrow^* wa\})$ $(G \restriction \{wa.\ u \rightarrow^* wa\})$
by (*intro subgraphI*) (*auto simp*: *arcs-ends-conv compatible-def*)
then have $w \rightarrow^*_{G \restriction \{wa.\ u \rightarrow^* wa\}} v$
by (*rule iG.reachable-mono*[*rotated*]) *fact*
ultimately show *?case* **by** (*rule iG.reachable-trans*)
qed

lemma *induce-subgraph-ends*[*simp*]:
arc-to-ends $(G \restriction S)$ = *arc-to-ends G*
by (*auto simp*: *arc-to-ends-def*)

lemma *dominates-induce-subgraphD*:
assumes $u \rightarrow_{G \restriction S} v$ **shows** $u \rightarrow_G v$
using *assms* **by** (*auto simp*: *arcs-ends-def intro*: *rev-image-eqI*)

context *wf-digraph* **begin**

lemma *reachable-induce-subgraphD*:
assumes $u \rightarrow^*_{G \restriction S} v$ $S \subseteq$ *verts* G **shows** $u \rightarrow^*_G v$
proof −
interpret *GS*: *wf-digraph* $G \restriction S$ **by** *auto*
show *?thesis*
using *assms* **by** *induct* (*auto dest*: *dominates-induce-subgraphD intro*: *adj-reachable-trans*)
qed

lemma *dominates-induce-ss*:
assumes $u \rightarrow_{G \restriction S} v$ $S \subseteq T$ **shows** $u \rightarrow_{G \restriction T} v$
using *assms* **by** (*auto simp*: *arcs-ends-def*)

lemma *reachable-induce-ss*:
assumes $u \rightarrow^*_{G \restriction S} v$ $S \subseteq T$ **shows** $u \rightarrow^*_{G \restriction T} v$
using *assms* **unfolding** *reachable-def*
by *induct* (*auto intro*: *dominates-induce-ss converse-rtrancl-on-into-rtrancl-on*)

lemma *awalk-verts-induce*:
pre-digraph.awalk-verts $(G \restriction S) =$ *awalk-verts*
proof (*intro ext*)
fix *u p* **show** *pre-digraph.awalk-verts* $(G \restriction S)$ *u p* = *awalk-verts u p*
by (*induct p arbitrary*: *u*) (*auto simp*: *pre-digraph.awalk-verts.simps*)
qed

lemma (**in** −) *cas-subset*:
assumes *pre-digraph.cas G u p v subgraph G H*
shows *pre-digraph.cas H u p v*
using *assms*
by (*induct p arbitrary*: *u*) (*auto simp*: *pre-digraph.cas.simps subgraph-def compatible-def*)

lemma *cas-induce*:
assumes *cas u p v set* (*awalk-verts u p*) $\subseteq S$
shows *pre-digraph.cas* $(G \restriction S)$ *u p v*
using *assms*
proof (*induct p arbitrary*: *u S*)
case *Nil* **then show** *?case* **by** (*auto simp*: *pre-digraph.cas.simps*)
next
case (*Cons a as*)
have *pre-digraph.cas* $(G \restriction$ *set* (*awalk-verts* (*head G a*) *as*)) (*head G a*) *as v*
using *Cons* **by** *auto*
then have *pre-digraph.cas* $(G \restriction S)$ (*head G a*) *as v*
using ‹- $\subseteq S$› **by** (*rule-tac cas-subset*) (*auto simp*: *subgraph-def compatible-def*)
then show *?case* **using** *Cons* **by** (*auto simp*: *pre-digraph.cas.simps*)
qed

```
  lemma awalk-induce:
    assumes awalk u p v set (awalk-verts u p) ⊆ S
    shows pre-digraph.awalk (G ↾ S) u p v
  proof −
    interpret GS: wf-digraph G ↾ S by auto
    show ?thesis
      using assms by (auto simp: pre-digraph.awalk-def cas-induce GS.cas-induce
set-awalk-verts)
  qed

  lemma subgraph-induce-subgraphI:
    assumes V ⊆ verts G shows subgraph (G ↾ V) G
    by (metis assms induced-imp-subgraph induced-induce)

end

lemma induced-subgraphI':
  assumes subg:subgraph H G
  assumes max: ⋀H'. subgraph H' G ⟹ (verts H' ≠ verts H ∨ arcs H' ⊆ arcs
H)
  shows induced-subgraph H G
proof −
  interpret H: wf-digraph H using ‹subgraph H G› ..
  define H' where H' = G ↾ verts H
  then have H'-props: subgraph H' G verts H' = verts H
    using subg by (auto intro: wf-digraph.subgraph-induce-subgraphI)
  moreover
  have arcs H' = arcs H
  proof
    show arcs H' ⊆ arcs H using max H'-props by auto
    show arcs H ⊆ arcs H' using subg by (auto simp: H'-def compatible-def)
  qed
  then show induced-subgraph H G by (auto simp: induced-subgraph-def H'-def
subg)
qed

lemma (in pre-digraph) induced-subgraph-altdef:
  induced-subgraph H G ⟷ subgraph H G ∧ (∀ H'. subgraph H' G ⟶ (verts H'
≠ verts H ∨ arcs H' ⊆ arcs H)) (is ?L ⟷ ?R)
proof −
  { fix H' :: ('a,'b) pre-digraph
    assume A: verts H' = verts H subgraph H' G
    interpret H': wf-digraph H' using ‹subgraph H' G› ..
    from ‹subgraph H' G›
    have comp: tail G = tail H' head G = head H' by (auto simp: compatible-def)
    then have ⋀a. a ∈ arcs H' ⟹ tail G a ∈ verts H ⋀a. a ∈ arcs H' ⟹ tail
G a ∈ verts H
      by (auto dest: H'.wellformed simp: A)
    then have arcs H' ⊆ {e ∈ arcs G. tail G e ∈ verts H ∧ head G e ∈ verts H}
```

using ‹*subgraph H' G*› **by** (*auto simp*: *subgraph-def comp A*(*1*)[*symmetric*])
}
then show *?thesis* **using** *induced-subgraphI'*[*of H G*] **by** (*auto simp*: *induced-subgraph-def*)
qed

8.6 Unions of Graphs

lemma
verts-union[*simp*]: *verts* (*union G H*) = *verts G* ∪ *verts H* **and**
arcs-union[*simp*]: *arcs* (*union G H*) = *arcs G* ∪ *arcs H* **and**
tail-union[*simp*]: *tail* (*union G H*) = *tail G* **and**
head-union[*simp*]: *head* (*union G H*) = *head G*
by (*auto simp*: *union-def*)

lemma *wellformed-union*:
assumes *wf-digraph G wf-digraph H compatible G H*
shows *wf-digraph* (*union G H*)
using *assms*
by *unfold-locales*
(*auto simp*: *union-def compatible-tail compatible-head dest*: *wf-digraph.wellformed*)

lemma *subgraph-union-iff*:
assumes *wf-digraph H1 wf-digraph H2 compatible H1 H2*
shows *subgraph* (*union H1 H2*) *G* ⟷ *subgraph H1 G* ∧ *subgraph H2 G*
using *assms* **by** (*fastforce simp*: *compatible-def intro*!: *subgraphI wellformed-union*)

lemma *subgraph-union*[*intro*]:
assumes *subgraph H1 G compatible H1 G*
assumes *subgraph H2 G compatible H2 G*
shows *subgraph* (*union H1 H2*) *G*
proof −
from *assms* **have** *wf-digraph* (*union H1 H2*)
by (*auto intro*: *wellformed-union simp*: *compatible-def*)
with *assms* **show** *?thesis*
by (*auto simp add*: *subgraph-def union-def arc-to-ends-def compatible-def*)
qed

lemma *union-fin-digraph*:
assumes *fin-digraph G fin-digraph H compatible G H*
shows *fin-digraph* (*union G H*)
proof *intro-locales*
interpret *G*: *fin-digraph G* **by** (*rule assms*)
interpret *H*: *fin-digraph H* **by** (*rule assms*)
show *wf-digraph* (*union G H*) **using** *assms*
by (*intro wellformed-union*) *intro-locales*
show *fin-digraph-axioms* (*union G H*)
using *assms* **by** *unfold-locales* (*auto simp*: *union-def*)
qed

lemma *subgraphs-of-union*:
assumes *wf-digraph G wf-digraph G′ compatible G G′*
shows *subgraph G* (*union G G′*)
and *subgraph G′* (*union G G′*)
using *assms* **by** (*auto intro*!: *subgraphI wellformed-union simp*: *compatible-def*)

8.7 Maximal Subgraphs

lemma (**in** *pre-digraph*) *max-subgraph-mp*:
assumes *max-subgraph Q x* ⋀*x*. *P x* ⟹ *Q x P x* **shows** *max-subgraph P x*
using *assms* **by** (*auto simp*: *max-subgraph-def*)

lemma (**in** *pre-digraph*) *max-subgraph-prop*: *max-subgraph P x* ⟹ *P x*
by (*simp add*: *max-subgraph-def*)

lemma (**in** *pre-digraph*) *max-subgraph-subg-eq*:
assumes *max-subgraph P H1 max-subgraph P H2 subgraph H1 H2*
shows *H1* = *H2*
using *assms* **by** (*auto simp*: *max-subgraph-def*)

lemma *subgraph-induce-subgraphI2*:
assumes *subgraph H G* **shows** *subgraph H* (*G* ↾ *verts H*)
using *assms* **by** (*auto simp*: *subgraph-def compatible-def wf-digraph.wellformed wf-digraph.wellformed-induce-subgraph*)

definition *arc-mono* :: (('*a*,'*b*) *pre-digraph* ⇒ *bool*) ⇒ *bool* **where**
arc-mono P ≡ (∀ *H1 H2*. *P H1* ∧ *subgraph H1 H2* ∧ *verts H1* = *verts H2* ⟶ *P H2*)

lemma (**in** *pre-digraph*) *induced-subgraphI-arc-mono*:
assumes *max-subgraph P H*
assumes *arc-mono P*
shows *induced-subgraph H G*
proof −
interpret *wf-digraph G* **using** *assms* **by** (*auto simp*: *max-subgraph-def*)
have *subgraph H* (*G* ↾ *verts H*) *subgraph* (*G* ↾ *verts H*) *G verts H* = *verts* (*G* ↾ *verts H*) *P H*
using *assms* **by** (*auto simp*: *max-subgraph-def subgraph-induce-subgraphI2 subgraph-induce-subgraphI*)
moreover
then have *P* (*G* ↾ *verts H*)
using *assms* **by** (*auto simp*: *arc-mono-def*)
ultimately
have *max-subgraph P* (*G* ↾ *verts H*)
using *assms* **by** (*auto simp*: *max-subgraph-def*) *metis*
then have *H* = *G* ↾ *verts H*
using ‹*max-subgraph P H*› ‹*subgraph H* -›
by (*intro max-subgraph-subg-eq*)
show *?thesis* **using** *assms* **by** (*subst* ‹*H* = -›) (*auto simp*: *max-subgraph-def*)

qed

lemma (**in** *pre-digraph*) *induced-subgraph-altdef2*:
induced-subgraph H G $\longleftrightarrow$ *max-subgraph* ($\lambda H'$. *verts* $H' =$ *verts H*) *H* (**is** *?L* $\longleftrightarrow$ *?R*)
proof
assume *?L*
moreover
{ fix H' **assume** *induced-subgraph H G subgraph H* H' $H \neq H'$
then have $\neg$(*subgraph* H' *G* $\wedge$ *verts* $H' =$ *verts H*)
by (*auto simp*: *induced-subgraph-altdef compatible-def elim*!: *allE*[**where** *x*=H'])
}
ultimately show *max-subgraph* ($\lambda H'$. *verts* $H' =$ *verts H*) *H* **by** (*auto simp*: *max-subgraph-def*)
next
assume *?R*
moreover have *arc-mono* ($\lambda H'$. *verts* $H' =$ *verts H*) **by** (*auto simp*: *arc-mono-def*)
ultimately show *?L* **by** (*rule induced-subgraphI-arc-mono*)
qed

lemma (**in** *pre-digraph*) *max-subgraphI*:
assumes *P x subgraph x G* $\bigwedge y$. $[\![x \neq y$; *subgraph x y*; *subgraph y G*$]\!] \Longrightarrow \neg P\ y$
shows *max-subgraph P x*
using *assms* **by** (*auto simp*: *max-subgraph-def*)

lemma (**in** *pre-digraph*) *subgraphI-max-subgraph*: *max-subgraph P x* $\Longrightarrow$ *subgraph x G*
by (*simp add*: *max-subgraph-def*)

8.8 Connected and Strongly Connected Graphs

context *wf-digraph* **begin**

lemma *in-sccs-verts-conv-reachable*:
$S \in$ *sccs-verts* $\longleftrightarrow S \neq \{\} \wedge (\forall u \in S.\ \forall v \in S.\ u \rightarrow^*_G v) \wedge (\forall u \in S.\ \forall v.\ v \notin S \longrightarrow \neg u \rightarrow^*_G v \vee \neg v \rightarrow^*_G u)$
by (*simp add*: *sccs-verts-def*)

lemma *sccs-verts-disjoint*:
assumes $S \in$ *sccs-verts* $T \in$ *sccs-verts* $S \neq T$ **shows** $S \cap T = \{\}$
using *assms* **unfolding** *in-sccs-verts-conv-reachable* **by** *safe meson+*

lemma *strongly-connected-spanning-imp-strongly-connected*:
assumes *spanning H G*
assumes *strongly-connected H*
shows *strongly-connected G*
proof (*unfold strongly-connected-def*, *intro ballI conjI*)

```
  from assms show verts G ≠ {} unfolding strongly-connected-def spanning-def
by auto
 next
  fix u v assume u ∈ verts G and v ∈ verts G
  then have u →*_H v subgraph H G
   using assms by (auto simp add: strongly-connected-def)
  then show u →* v by (rule reachable-mono)
 qed

 lemma strongly-connected-imp-induce-subgraph-strongly-connected:
  assumes subg: subgraph H G
  assumes sc: strongly-connected H
  shows strongly-connected (G ↾ (verts H))
 proof −
  let ?is-H = G ↾ (verts H)

  interpret H: wf-digraph H
   using subg by (rule subgraphE)
  interpret GrH: wf-digraph ?is-H
   by (rule wellformed-induce-subgraph)

  have verts H ⊆ verts G using assms by auto

  have subgraph H (G ↾ verts H)
   using subg by (intro subgraphI) (auto simp: compatible-def)
  then show ?thesis
   using induced-induce[OF ‹verts H ⊆ verts G›]
    and sc GrH.strongly-connected-spanning-imp-strongly-connected
   unfolding spanning-def by auto
 qed

 lemma in-sccs-vertsI-sccs:
  assumes S ∈ verts ‘ sccs shows S ∈ sccs-verts
  unfolding sccs-verts-def
 proof (intro CollectI conjI allI ballI impI)
  show S ≠ {} using assms by (auto simp: sccs-verts-def sccs-def strongly-connected-def)

  from assms have sc: strongly-connected (G ↾ S) S ⊆ verts G
   apply (auto simp: sccs-verts-def sccs-def)
  by (metis induced-imp-subgraph subgraphE wf-digraph.strongly-connected-imp-induce-subgraph-strongly-con

  {
   fix u v assume A: u ∈ S v ∈ S
   with sc have u →*_{G ↾ S} v by auto
  then show u →*_G v using ‹S ⊆ verts G› by (rule reachable-induce-subgraphD)
  next
   fix u v assume A: u ∈ S v ∉ S
   { assume B: u →*_G v v →*_G u
    from B obtain p-uv where p-uv: awalk u p-uv v by (metis reachable-awalk)
```

from B **obtain** $p\text{-}vu$ **where** $p\text{-}vu$: $awalk\ v\ p\text{-}vu\ u$ **by** $(metis\ reachable\text{-}awalk)$
define T **where** $T = S \cup set\ (awalk\text{-}verts\ u\ p\text{-}uv) \cup set\ (awalk\text{-}verts\ v\ p\text{-}vu)$
have $S \subseteq T$ **by** $(auto\ simp\colon T\text{-}def)$
have $v \in T$ **using** $p\text{-}vu$ **by** $(auto\ simp\colon T\text{-}def\ set\text{-}awalk\text{-}verts)$
then have $T \neq S$ **using** $\langle v \notin S \rangle$ **by** $auto$

interpret T: $wf\text{-}digraph\ G \restriction T$ **by** $auto$

from $p\text{-}uv$ **have** $T\text{-}p\text{-}uv$: $T.awalk\ u\ p\text{-}uv\ v$
by $(rule\ awalk\text{-}induce)\ (auto\ simp\colon T\text{-}def)$
from $p\text{-}vu$ **have** $T\text{-}p\text{-}vu$: $T.awalk\ v\ p\text{-}vu\ u$
by $(rule\ awalk\text{-}induce)\ (auto\ simp\colon T\text{-}def)$

have $uv\text{-}reach$: $u \rightarrow^*_{G \restriction T} v\ v \rightarrow^*_{G \restriction T} u$
using $T\text{-}p\text{-}uv\ T\text{-}p\text{-}vu\ A$ **by** $(metis\ T.reachable\text{-}awalk)+$

{ fix $x\ y$ **assume** $x \in S\ y \in S$
then have $x \rightarrow^*_{G \restriction S} y\ y \rightarrow^*_{G \restriction S} x$
using sc **by** $auto$
then have $x \rightarrow^*_{G \restriction T} y\ y \rightarrow^*_{G \restriction T} x$
using $\langle S \subseteq T \rangle$ **by** $(auto\ intro\colon reachable\text{-}induce\text{-}ss)$
} note $A1 = this$

{ fix x **assume** $x \in T$
moreover
{ assume $x \in S$ **then have** $x \rightarrow^*_{G \restriction T} v$
using $uv\text{-}reach\ A1\ A$ **by** $(auto\ intro\colon T.reachable\text{-}trans[rotated])$
} moreover
{ assume $x \in set\ (awalk\text{-}verts\ u\ p\text{-}uv)$ **then have** $x \rightarrow^*_{G \restriction T} v$
using $T\text{-}p\text{-}uv$ **by** $(auto\ simp\colon awalk\text{-}verts\text{-}induce\ intro\colon T.awalk\text{-}verts\text{-}reachable\text{-}to)$
} moreover
{ assume $x \in set\ (awalk\text{-}verts\ v\ p\text{-}vu)$ **then have** $x \rightarrow^*_{G \restriction T} v$
using $T\text{-}p\text{-}vu$ **by** $(rule\text{-}tac\ T.reachable\text{-}trans)$
$(auto\ simp\colon uv\text{-}reach\ awalk\text{-}verts\text{-}induce\ dest\colon T.awalk\text{-}verts\text{-}reachable\text{-}to)$
} ultimately
have $x \rightarrow^*_{G \restriction T} v$ **by** $(auto\ simp\colon T\text{-}def)$
} note $xv\text{-}reach = this$

{ fix x **assume** $x \in T$
moreover
{ assume $x \in S$ **then have** $v \rightarrow^*_{G \restriction T} x$
using $uv\text{-}reach\ A1\ A$ **by** $(auto\ intro\colon T.reachable\text{-}trans)$
} moreover
{ assume $x \in set\ (awalk\text{-}verts\ v\ p\text{-}vu)$ **then have** $v \rightarrow^*_{G \restriction T} x$
using $T\text{-}p\text{-}vu$ **by** $(auto\ simp\colon awalk\text{-}verts\text{-}induce\ intro\colon T.awalk\text{-}verts\text{-}reachable\text{-}from)$
} moreover
{ assume $x \in set\ (awalk\text{-}verts\ u\ p\text{-}uv)$ **then have** $v \rightarrow^*_{G \restriction T} x$

```
        using T-p-uv by (rule-tac T.reachable-trans[rotated])
       (auto intro: T.awalk-verts-reachable-from uv-reach simp: awalk-verts-induce)
      } ultimately
      have v →*_{G ↾ T} x by (auto simp: T-def)
    } note vx-reach = this

    { fix x y assume x ∈ T y ∈ T then have x →*_{G ↾ T} y
        using xv-reach vx-reach by (blast intro: T.reachable-trans)
    }
    then have strongly-connected (G ↾ T)
      using ‹S ≠ {}› ‹S ⊆ T› by auto
    moreover have induced-subgraph (G ↾ T) G
      using ‹S ⊆ verts G›
       by (auto simp: T-def intro: awalk-verts-reachable-from p-uv p-vu reach-
able-in-verts(2))
    ultimately
   have ∃ T. induced-subgraph (G ↾ T) G ∧ strongly-connected (G ↾ T) ∧ verts
(G ↾ S) ⊂ verts (G ↾ T)
      using ‹S ⊆ T› ‹T ≠ S› by auto
    then have G ↾ S ∉ sccs unfolding sccs-def by blast
    then have S ∉ verts ‘ sccs
     by (metis (erased, opaque-lifting) ‹S ⊆ T› ‹T ≠ S› ‹induced-subgraph (G
↾ T) G› ‹strongly-connected (G ↾ T)›
        dual-order.order-iff-strict image-iff in-sccsE induce-subgraph-verts)
    then have False using assms by metis
  }
  then show ¬u →*_G v ∨ ¬v →*_G u by metis
 }
qed

end

lemma arc-mono-strongly-connected[intro,simp]: arc-mono strongly-connected
 by (auto simp: arc-mono-def) (metis spanning-def subgraphE wf-digraph.strongly-connected-spanning-imp-stro

lemma (in pre-digraph) sccs-altdef2:
  sccs = {H. max-subgraph strongly-connected H} (is ?L = ?R)
proof −
 { fix H H' :: ('a, 'b) pre-digraph
   assume a1: strongly-connected H'
   assume a2: induced-subgraph H' G
   assume a3: max-subgraph strongly-connected H
   assume a4: verts H ⊆ verts H'
   have sg: subgraph H G and ends-G: tail G = tail H  head G = head H
     using a3 by (auto simp: max-subgraph-def compatible-def)
   then interpret H: wf-digraph H by blast
   have arcs H ⊆ arcs H' using a2 a4 sg by (fastforce simp: ends-G)
   then have H = H'
     using a1 a2 a3 a4
```

by (*metis* (*no-types*) *compatible-def induced-imp-subgraph max-subgraph-def subgraph-def*)
} **note** $X = this$

{ **fix** H
assume *a1*: *induced-subgraph H G*
assume *a2*: *strongly-connected H*
assume *a3*: $\forall H'$. *strongly-connected* $H' \longrightarrow$ *induced-subgraph* H' $G \longrightarrow \neg$ *verts* $H \subset$ *verts* H'
interpret *G*: *wf-digraph G* **using** *a1* **by** *auto*
{ **fix** y **assume** $H \neq y$ **and** *subg*: *subgraph H y subgraph y G*
then have *verts H* $\subset$ *verts y*
using *a1* **by** (*auto simp*: *induced-subgraph-altdef2 max-subgraph-def*)
then have $\neg$*strongly-connected y*
using *subg a1 a2 a3*[*THEN spec*, *of G* $\restriction$ *verts y*]
by (*auto simp*: *G.induced-induce G.strongly-connected-imp-induce-subgraph-strongly-connected*)
}
then have *max-subgraph strongly-connected H*
using *a1 a2* **by** (*auto intro*: *max-subgraphI*)
} **note** $Y = this$

show *?thesis* **unfolding** *sccs-def*
by (*auto dest*: *max-subgraph-prop X intro*: *induced-subgraphI-arc-mono Y*)
qed

locale *max-reachable-set* = *wf-digraph* +
fixes S **assumes** *S-in-sv*: $S \in$ *sccs-verts*
begin

lemma *reach-in*: $\bigwedge u\ v.\ [\![u \in S;\ v \in S]\!] \Longrightarrow u \rightarrow^*_G v$
and *not-reach-out*: $\bigwedge u\ v.\ [\![u \in S;\ v \notin S]\!] \Longrightarrow \neg u \rightarrow^*_G v \vee \neg v \rightarrow^*_G u$
and *not-empty*: $S \neq \{\}$
using *S-in-sv* **by** (*auto simp*: *sccs-verts-def*)

lemma *reachable-induced*:
assumes *conn*: $u \in S\ v \in S\ u \rightarrow^*_G v$
shows $u \rightarrow^*_{G \restriction S} v$
proof −
let *?H* = $G \restriction S$
have $S \subseteq$ *verts G* **using** *reach-in* **by** (*auto dest*: *reachable-in-verts*)
then have *induced-subgraph ?H G*
by (*rule induced-induce*)
then interpret *H*: *wf-digraph ?H* **by** (*rule wf-digraphI-induced*)

from *conn* **obtain** *p* **where** *p*: *awalk u p v* **by** (*metis reachable-awalk*)
show *?thesis*
proof (*cases set p* $\subseteq$ *arcs* ($G \restriction S$))
case *True*
with *p conn* **have** *H.awalk u p v*

```
    by (auto simp: pre-digraph.awalk-def compatible-cas[OF compatible-induce-subgraph])
    then show ?thesis by (metis H.reachable-awalk)
   next
    case False
    then obtain a where a ∈ set p a ∉ arcs (G ↾ S) by auto
    moreover
    then have tail G a ∉ S ∨ head G a ∉ S using p by auto
    ultimately
     obtain w where w ∈ set (awalk-verts u p) w ∉ S using p by (auto simp:
set-awalk-verts)
    then have u →*G w w →*G v
     using p by (auto intro: awalk-verts-reachable-from awalk-verts-reachable-to)
    moreover have v →*G u using conn reach-in by auto
    ultimately have u →*G w w →*G u by (auto intro: reachable-trans)
    with ‹w ∉ S› conn not-reach-out have False by blast
    then show ?thesis ..
   qed
  qed

  lemma strongly-connected:
   shows strongly-connected (G ↾ S)
    using not-empty by (intro strongly-connectedI) (auto intro: reachable-induced
reach-in)

  lemma induced-in-sccs: G ↾ S ∈ sccs
  proof −
   let ?H = G ↾ S
   have S ⊆ verts G using reach-in by (auto dest: reachable-in-verts)
   then have induced-subgraph ?H G
     by (rule induced-induce)
   then interpret H: wf-digraph ?H by (rule wf-digraphI-induced)

   { fix T assume S ⊂ T T ⊆ verts G strongly-connected (G ↾ T)
    from ‹S ⊂ T› obtain v where v ∈ T v ∉ S by auto
    from not-empty obtain u where u ∈ S by auto
    then have u ∈ T using ‹S ⊂ T› by auto

    from ‹u ∈ S› ‹v ∉ S› have ¬u →*G v ∨ ¬v →*G u by (rule not-reach-out)
    moreover
    from ‹strongly-connected -› have u →*G ↾ T v v →*G ↾ T u
     using ‹v ∈ T› ‹u ∈ T› by (auto simp: strongly-connected-def)
    then have u →*G v v →*G u
     using ‹T ⊆ verts G› by (auto dest: reachable-induce-subgraphD)
    ultimately have False by blast
   } note psuper-not-sc = this

   have ¬ (∃ c'. induced-subgraph c' G ∧ strongly-connected c' ∧ verts (G ↾ S) ⊂
verts c')
    by (metis induce-subgraph-verts induced-imp-subgraph psuper-not-sc subgraphE
```

strongly-connected-imp-induce-subgraph-strongly-connected)
with ‹$S \subseteq$ -› *not-empty* **show** *?H* $\in$ *sccs* **by** (*intro in-sccsI induced-induce strongly-connected*)
qed
end

context *wf-digraph* **begin**

lemma *in-verts-sccsD-sccs*:
assumes $S \in$ *sccs-verts*
shows $G \restriction S \in$ *sccs*
proof −
from *assms* **interpret** *max-reachable-set* **by** *unfold-locales*
show *?thesis* **by** (*auto simp*: *sccs-verts-def intro*: *induced-in-sccs*)
qed

lemma *sccs-verts-conv*: *sccs-verts* = *verts* ‘ *sccs*
by (*auto intro*: *in-sccs-vertsI-sccs rev-image-eqI dest*: *in-verts-sccsD-sccs*)

lemma *induce-eq-iff-induced*:
assumes *induced-subgraph H G* **shows** *G* ↾ *verts H* = *H*
using *assms* **by** (*auto simp*: *induced-subgraph-def induce-subgraph-def compatible-def*)

lemma *sccs-conv-sccs-verts*: *sccs* = *induce-subgraph G* ‘ *sccs-verts*
by (*auto intro*!: *rev-image-eqI in-sccs-vertsI-sccs dest*: *in-verts-sccsD-sccs*
simp: *sccs-def induce-eq-iff-induced*)

end

lemma *connected-conv*:
shows *connected G* ⟷ *verts G* ≠ {} ∧ (∀ *u* ∈ *verts G*. ∀ *v* ∈ *verts G*. (*u*,*v*) ∈ *rtrancl-on* (*verts G*) ((*arcs-ends G*)^s))
proof −
have *symcl* (*arcs-ends G*) = *parcs* (*mk-symmetric G*)
by (*auto simp*: *parcs-mk-symmetric symcl-def arcs-ends-conv*)
then show *?thesis* **by** (*auto simp*: *connected-def strongly-connected-def reachable-def*)
qed

lemma (**in** *wf-digraph*) *symmetric-connected-imp-strongly-connected*:
assumes *symmetric G connected G*
shows *strongly-connected G*
proof
from ‹*connected G*› **show** *verts G* ≠ {} **unfolding** *connected-def strongly-connected-def*
by *auto*
next
from ‹*connected G*›

```
  have sc-mks: strongly-connected (mk-symmetric G)
    unfolding connected-def by simp

  fix u v assume u ∈ verts G v ∈ verts G
  with sc-mks have u →*mk-symmetric G v
    unfolding strongly-connected-def by auto
  then show u →* v using assms by (simp only: reachable-mk-symmetric-eq)
qed

lemma (in wf-digraph) connected-spanning-imp-connected:
  assumes spanning H G
  assumes connected H
  shows connected G
proof (unfold connected-def strongly-connected-def, intro conjI ballI)
  from assms show verts (mk-symmetric G )≠ {}
    unfolding spanning-def connected-def strongly-connected-def by auto
next
  fix u v
  assume u ∈ verts (mk-symmetric G) and v ∈ verts (mk-symmetric G)
  then have u ∈ pverts (mk-symmetric H) and v ∈ pverts (mk-symmetric H)
    using ‹spanning H G› by (auto simp: mk-symmetric-def)
  with ‹connected H›
 have u →*with-proj (mk-symmetric H) v subgraph (mk-symmetric H) (mk-symmetric
G)
    using ‹spanning H G› unfolding connected-def
    by (auto simp: spanning-def dest: subgraph-mk-symmetric)
  then show u →*mk-symmetric G v by (rule pre-digraph.reachable-mono)
qed

lemma (in wf-digraph) spanning-tree-imp-connected:
  assumes spanning-tree H G
  shows connected G
using assms by (auto intro: connected-spanning-imp-connected)

term LEAST x. P x

lemma (in sym-digraph) induce-reachable-is-in-sccs:
  assumes u ∈ verts G
  shows (G ↾ {v. u →* v}) ∈ sccs
proof −
  let ?c = (G ↾ {v. u →* v})
  have isub-c: induced-subgraph ?c G
    by (auto elim: reachable-in-vertsE)
  then interpret c: wf-digraph ?c by (rule wf-digraphI-induced)

  have sym-c: symmetric (G ↾ {v. u →* v})
    using sym-arcs isub-c by (rule induced-graph-imp-symmetric)

  note ‹induced-subgraph ?c G›
```

```
  moreover
  have strongly-connected ?c
  proof (rule strongly-connectedI)
    show verts ?c ≠ {} using assms by auto
  next
    fix v w assume l-assms: v ∈ verts ?c w ∈ verts ?c
    have u →*_{G ↾ {v. u →* v}} v
      using l-assms by (intro induce-reachable-preserves-paths) auto
    then have v →*_{G ↾ {v. u →* v}} u by (rule symmetric-reachable[OF sym-c])
    also have u →*_{G ↾ {v. u →* v}} w
      using l-assms by (intro induce-reachable-preserves-paths) auto
    finally show v →*_{G ↾ {v. u →* v}} w .
  qed
  moreover
  have ¬(∃ d. induced-subgraph d G ∧ strongly-connected d ∧
    verts ?c ⊂ verts d)
  proof
    assume ∃ d. induced-subgraph d G ∧ strongly-connected d ∧
      verts ?c ⊂ verts d
    then obtain d where induced-subgraph d G strongly-connected d
      verts ?c ⊂ verts d by auto
    then obtain v where v ∈ verts d and v ∉ verts ?c
      by auto

    have u ∈ verts ?c using ‹u ∈ verts G› by auto
    then have u ∈ verts d using ‹verts ?c ⊂ verts d› by auto
    then have u →*_d v
      using ‹strongly-connected d› ‹u ∈ verts d› ‹v ∈ verts d› by auto
    then have u →* v
      using ‹induced-subgraph d G›
      by (auto intro: pre-digraph.reachable-mono)
    then have v ∈ verts ?c by (auto simp: reachable-awalk)
    then show False using ‹v ∉ verts ?c› by auto
  qed
  ultimately show ?thesis unfolding sccs-def by auto
qed

lemma induced-eq-verts-imp-eq:
  assumes induced-subgraph G H
  assumes induced-subgraph G' H
  assumes verts G = verts G'
  shows G = G'
  using assms by (auto simp: induced-subgraph-def subgraph-def compatible-def)

lemma (in pre-digraph) in-sccs-subset-imp-eq:
  assumes c ∈ sccs
  assumes d ∈ sccs
  assumes verts c ⊆ verts d
  shows c = d
```

using *assms* **by** (*blast intro*: *induced-eq-verts-imp-eq*)

context *wf-digraph* **begin**

lemma *connectedI*:
assumes *verts G* ≠ {} ⋀*u v*. *u* ∈ *verts G* ⟹ *v* ∈ *verts G* ⟹ *u* →** mk-symmetric G v
shows *connected G*
using *assms* **by** (*auto simp*: *connected-def*)

lemma *connected-awalkE*:
assumes *connected G u* ∈ *verts G v* ∈ *verts G*
obtains *p* **where** *pre-digraph.awalk* (*mk-symmetric G*) *u p v*
proof −
interpret *sG*: *pair-wf-digraph mk-symmetric G* **..**
from *assms* **have** *u* →** mk-symmetric G v **by** (*auto simp*: *connected-def*)
then obtain *p* **where** *sG.awalk u p v* **by** (*auto simp*: *sG.reachable-awalk*)
then show *?thesis* **..**
qed

lemma *inj-on-verts-sccs*: *inj-on verts sccs*
by (*rule inj-onI*) (*metis in-sccs-imp-induced induced-eq-verts-imp-eq*)

lemma *card-sccs-verts*: *card sccs-verts* = *card sccs*
by (*auto simp*: *sccs-verts-conv intro*: *inj-on-verts-sccs card-image*)

end

lemma *strongly-connected-non-disj*:
assumes *wf*: *wf-digraph G wf-digraph H compatible G H*
assumes *sc*: *strongly-connected G strongly-connected H*
assumes *not-disj*: *verts G* ∩ *verts H* ≠ {}
shows *strongly-connected* (*union G H*)
proof
from *sc* **show** *verts* (*union G H*) ≠ {}
unfolding *strongly-connected-def* **by** *simp*
next
let *?x* = *union G H*
fix *u v w* **assume** *u* ∈ *verts ?x* **and** *v* ∈ *verts ?x*
obtain *w* **where** *w-in-both*: *w* ∈ *verts G w* ∈ *verts H*
using *not-disj* **by** *auto*

interpret *x*: *wf-digraph ?x*
by (*rule wellformed-union*) *fact*+
have *subg*: *subgraph G ?x subgraph H ?x*
by (*rule subgraphs-of-union*[*OF* - -], *fact*+)+
have *reach-uw*: *u* →** ?x *w*
using ‹*u* ∈ *verts ?x*› *subg w-in-both sc*

```
    by (auto intro: pre-digraph.reachable-mono)
  also have reach-wv: w →*_?x v
    using ‹v ∈ verts ?x› subg w-in-both sc
    by (auto intro: pre-digraph.reachable-mono)
  finally (x.reachable-trans) show u →*_?x v .
qed

context wf-digraph begin

  lemma scc-disj:
    assumes scc: c ∈ sccs d ∈ sccs
    assumes c ≠ d
    shows verts c ∩ verts d = {}
  proof (rule ccontr)
    assume contr: ¬?thesis

    let ?x = union c d

    have comp1: compatible G c compatible G d
      using scc by (auto simp: sccs-def)
    then have comp: compatible c d by (auto simp: compatible-def)

    have wf: wf-digraph c wf-digraph d
      and sc: strongly-connected c strongly-connected d
      using scc by (auto intro: in-sccs-imp-induced)
    have compatible c d
      using comp by (auto simp: sccs-def compatible-def)
    from wf comp sc have union-conn: strongly-connected ?x
      using contr by (rule strongly-connected-non-disj)

    have sg: subgraph ?x G
      using scc comp1 by (intro subgraph-union) (auto simp: compatible-def)
   then have v-cd: verts c ⊆ verts G  verts d ⊆ verts G by (auto elim!: subgraphE)
    have wf-digraph ?x by (rule wellformed-union) fact+
    with v-cd sg union-conn
    have induce-subgraph-conn: strongly-connected (G ↾ verts ?x)
        induced-subgraph (G ↾ verts ?x) G
      by − (intro strongly-connected-imp-induce-subgraph-strongly-connected,
        auto simp: subgraph-union-iff)

    from assms have ¬verts c ⊆ verts d and ¬ verts d ⊆ verts c
      by (metis in-sccs-subset-imp-eq)+
    then have psub: verts c ⊂ verts ?x
      by (auto simp: union-def)
    then show False using induce-subgraph-conn
      by (metis ‹c ∈ sccs› in-sccsE induce-subgraph-verts)
  qed

  lemma in-sccs-verts-conv:
```

$S \in$ *sccs-verts* $\longleftrightarrow$ $G \restriction S \in$ *sccs*
by (*auto simp*: *sccs-verts-conv intro*: *rev-image-eqI*)
(*metis in-sccs-imp-induced induce-subgraph-verts induced-eq-verts-imp-eq induced-imp-subgraph induced-induce subgraphE*)

end

lemma (**in** *wf-digraph*) *in-scc-of-self*: $u \in$ *verts* $G \Longrightarrow u \in$ *scc-of* u
by (*auto simp*: *scc-of-def*)

lemma (**in** *wf-digraph*) *scc-of-empty-conv*: *scc-of* $u = \{\} \longleftrightarrow u \notin$ *verts* G
using *in-scc-of-self* **by** (*auto simp*: *scc-of-def reachable-in-verts*)

lemma (**in** *wf-digraph*) *scc-of-in-sccs-verts*:
assumes $u \in$ *verts* G **shows** *scc-of* $u \in$ *sccs-verts*
using *assms* **by** (*auto simp*: *in-sccs-verts-conv-reachable scc-of-def intro*: *reachable-trans exI*[**where** x=u])

lemma (**in** *wf-digraph*) *sccs-verts-subsets*: $S \in$ *sccs-verts* $\Longrightarrow S \subseteq$ *verts* G
by (*auto simp*: *sccs-verts-conv*)

lemma (**in** *fin-digraph*) *finite-sccs-verts*: *finite sccs-verts*
proof −
have *finite* (*Pow* (*verts* G)) **by** *auto*
moreover with *sccs-verts-subsets* **have** *sccs-verts* $\subseteq$ *Pow* (*verts* G) **by** *auto*
ultimately show *?thesis* **by** (*rule rev-finite-subset*)
qed

lemma (**in** *wf-digraph*) *sccs-verts-conv-scc-of*:
sccs-verts = *scc-of* ‘ *verts* G (**is** *?L* = *?R*)
proof (*intro set-eqI iffI*)
fix S **assume** $S \in$ *?R* **then show** $S \in$ *?L*
by (*auto simp*: *in-sccs-verts-conv-reachable scc-of-empty-conv*) (*auto simp*: *scc-of-def intro*: *reachable-trans*)
next
fix S **assume** $S \in$ *?L*
moreover
then obtain u **where** $u \in S$ **by** (*auto simp*: *in-sccs-verts-conv-reachable*)
moreover
then have $u \in$ *verts* G **using** ‹$S \in$ *?L*› **by** (*metis sccs-verts-subsets subsetCE*)
then have *scc-of* $u \in$ *sccs-verts* $u \in$ *scc-of* u
by (*auto intro*: *scc-of-in-sccs-verts in-scc-of-self*)
ultimately
have *scc-of* $u = S$ **using** *sccs-verts-disjoint* **by** *blast*
then show $S \in$ *?R* **using** ‹*scc-of* $u \in$ -› ‹$u \in$ *verts* G› **by** *auto*
qed

lemma (**in** *sym-digraph*) *scc-ofI-reachable*:
assumes $u \rightarrow^* v$ **shows** $u \in$ *scc-of* v

using *assms* **by** (*auto simp*: *scc-of-def symmetric-reachable*[*OF sym-arcs*])

lemma (**in** *sym-digraph*) *scc-ofI-reachable'*:
assumes $v \rightarrow^* u$ **shows** $u \in$ *scc-of* v
using *assms* **by** (*auto simp*: *scc-of-def symmetric-reachable*[*OF sym-arcs*])

lemma (**in** *sym-digraph*) *scc-ofI-awalk*:
assumes *awalk* u p v **shows** $u \in$ *scc-of* v
using *assms* **by** (*metis reachable-awalk scc-ofI-reachable*)

lemma (**in** *sym-digraph*) *scc-ofI-apath*:
assumes *apath* u p v **shows** $u \in$ *scc-of* v
using *assms* **by** (*metis reachable-apath scc-ofI-reachable*)

lemma (**in** *wf-digraph*) *scc-of-eq*: $u \in$ *scc-of* $v \Longrightarrow$ *scc-of* $u =$ *scc-of* v
by (*auto simp*: *scc-of-def intro*: *reachable-trans*)

lemma (**in** *wf-digraph*) *strongly-connected-eq-iff*:
strongly-connected $G \longleftrightarrow$ *sccs* $= \{G\}$ (**is** *?L* $\longleftrightarrow$ *?R*)
proof
assume *?L*
then have $G \in$ *sccs* **by** (*auto simp*: *sccs-def induced-subgraph-refl*)
moreover
{ fix H **assume** $H \in$ *sccs* $G \neq H$
with ‹$G \in$ *sccs*› **have** *verts* $G \cap$ *verts* $H = \{\}$ **by** (*rule scc-disj*)
moreover
from ‹$H \in$ *sccs*› **have** *verts* $H \subseteq$ *verts* G **by** *auto*
ultimately
have *verts* $H = \{\}$ **by** *auto*
with ‹$H \in$ *sccs*› **have** *False* **by** (*auto simp*: *sccs-def strongly-connected-def*)
} ultimately
show *?R* **by** *auto*
qed (*auto simp*: *sccs-def*)

8.9 Components

lemma (**in** *sym-digraph*) *exists-scc*:
assumes *verts* $G \neq \{\}$ **shows** $\exists c.\ c \in$ *sccs*
proof −
from *assms* **obtain** u **where** $u \in$ *verts* G **by** *auto*
then show *?thesis* **by** (*blast dest*: *induce-reachable-is-in-sccs*)
qed

theorem (**in** *sym-digraph*) *graph-is-union-sccs*:
shows *Union sccs* $=$ G
proof −
have $(\bigcup c \in$ *sccs*. *verts* $c) =$ *verts* G
by (*auto intro*: *induce-reachable-is-in-sccs*)
moreover

```
  have (⋃ c ∈ sccs. arcs c) = arcs G
  proof
    show (⋃ c ∈ sccs. arcs c) ⊆ arcs G
      by safe (metis in-sccsE induced-imp-subgraph subgraphE subsetD)
    show arcs G ⊆ (⋃ c ∈ sccs. arcs c)
    proof (safe)
      fix e assume e ∈ arcs G
      define a b where [simp]: a = tail G e and [simp]: b = head G e

      have e ∈ (⋃ x ∈ sccs. arcs x)
      proof cases
        assume ∃ x∈sccs. {a,b } ⊆ verts x
        then obtain c where c ∈ sccs and {a,b} ⊆ verts c
          by auto
        then have e ∈ {e ∈ arcs G. tail G e ∈ verts c
          ∧ head G e ∈ verts c} using ‹e ∈ arcs G› by auto
        then have e ∈ arcs c using ‹c ∈ sccs› by blast
        then show ?thesis using ‹c ∈ sccs› by auto
      next
        assume l-assm: ¬(∃ x∈sccs. {a,b} ⊆ verts x)

        have a →* b using ‹e ∈ arcs G›
          by (metis a-def b-def reachable-adjI in-arcs-imp-in-arcs-ends)
        then have {a,b} ⊆ verts (G ↾ {v. a →* v}) a ∈ verts G
          by (auto elim: reachable-in-vertsE)
        moreover
        have (G ↾ {v. a →* v}) ∈ sccs
          using ‹a ∈ verts G› by (auto intro: induce-reachable-is-in-sccs)
        ultimately
        have False using l-assm by blast
        then show ?thesis by simp
      qed
      then show e ∈ (⋃ c ∈ sccs. arcs c) by auto
    qed
  qed
  ultimately show ?thesis
    by (auto simp add: Union-def)
qed

lemma (in sym-digraph) scc-for-vert-ex:
  assumes u ∈ verts G
  shows ∃ c. c∈sccs ∧ u ∈ verts c
using assms by (auto intro: induce-reachable-is-in-sccs)



lemma (in sym-digraph) scc-decomp-unique:
  assumes S ⊆ sccs verts (Union S) = verts G shows S = sccs
proof (rule ccontr)
```

assume $S \neq sccs$
with *assms* **obtain** c **where** $c \in sccs$ **and** $c \notin S$ **by** *auto*
with *assms* **have** $\bigwedge d.\ d \in S \Longrightarrow verts\ c \cap verts\ d = \{\}$
by (*intro scc-disj*) *auto*
then have $verts\ c \cap verts\ (Union\ S) = \{\}$
by (*auto simp*: *Union-def*)
with *assms* **have** $verts\ c \cap verts\ G = \{\}$ **by** *auto*
moreover from ‹$c \in sccs$› **obtain** u **where** $u \in verts\ c \cap verts\ G$
by (*auto simp*: *sccs-def strongly-connected-def*)
ultimately show *False* **by** *blast*
qed

end

theory *Vertex-Walk*
imports *Arc-Walk*
begin

9 Walks Based on Vertices

These definitions are here mainly for historical purposes, as they do not really work with multigraphs. Consider using Arc Walks instead.

type-synonym $'a\ vwalk = 'a\ list$

Computes the list of arcs belonging to a list of nodes

fun *vwalk-arcs* :: $'a\ vwalk \Rightarrow ('a \times 'a)\ list$ **where**
  vwalk-arcs [] = []
| *vwalk-arcs* [x] = []
| *vwalk-arcs* (x#y#xs) = (x,y) # *vwalk-arcs* (y#xs)

definition *vwalk-length* :: $'a\ vwalk \Rightarrow nat$ **where**
  vwalk-length $p \equiv length\ (vwalk\text{-}arcs\ p)$

lemma *vwalk-length-simp*[*simp*]:
  shows *vwalk-length* $p = length\ p - 1$
by (*induct p rule*: *vwalk-arcs.induct*) (*auto simp*: *vwalk-length-def*)

definition *vwalk* :: $'a\ vwalk \Rightarrow ('a,'b)\ pre\text{-}digraph \Rightarrow bool$ **where**
  vwalk $p\ G \equiv set\ p \subseteq verts\ G \wedge set\ (vwalk\text{-}arcs\ p) \subseteq arcs\text{-}ends\ G \wedge p \neq []$

definition *vpath* :: $'a\ vwalk \Rightarrow ('a,'b)\ pre\text{-}digraph \Rightarrow bool$ **where**
  vpath $p\ G \equiv vwalk\ p\ G \wedge distinct\ p$

For a given vwalk, compute a vpath with the same tail G and end

function *vwalk-to-vpath* :: $'a\ vwalk \Rightarrow 'a\ vwalk$ **where**

```
    vwalk-to-vpath [] = []
  | vwalk-to-vpath (x # xs) = (if (x ∈ set xs)
      then vwalk-to-vpath (dropWhile (λy. y ≠ x) xs)
      else x # vwalk-to-vpath xs)
by pat-completeness auto
termination by (lexicographic-order simp add: length-dropWhile-le)


lemma vwalkI[intro]:
  assumes set p ⊆ verts G
  assumes set (vwalk-arcs p) ⊆ arcs-ends G
  assumes p ≠ []
  shows vwalk p G
using assms by (auto simp add: vwalk-def)

lemma vwalkE[elim]:
  assumes vwalk p G
  assumes set p ⊆ verts G ⟹
    set (vwalk-arcs p) ⊆ arcs-ends G ∧ p ≠ [] ⟹ P
  shows P
using assms by (simp add: vwalk-def)

lemma vpathI[intro]:
  assumes vwalk p G
  assumes distinct p
  shows vpath p G
using assms by (simp add: vpath-def)

lemma vpathE[elim]:
  assumes vpath p G
  assumes vwalk p G ⟹ distinct p ⟹ P
  shows P
using assms by (simp add: vpath-def)


lemma vwalk-consI:
  assumes vwalk p G
  assumes a ∈ verts G
  assumes (a, hd p) ∈ arcs-ends G
  shows vwalk (a # p) G
using assms by (cases p) (auto simp add: vwalk-def)

lemma vwalk-consE:
  assumes vwalk (a # p) G
  assumes p ≠ []
  assumes (a, hd p) ∈ arcs-ends G ⟹ vwalk p G ⟹ P
  shows P
using assms by (cases p) (auto simp add: vwalk-def)
```

lemma *vwalk-induct*[*case-names Base Cons*, *induct pred*: *vwalk*]:
assumes *vwalk p G*
assumes $\bigwedge u.\ u \in verts\ G \Longrightarrow P\ [u]$
assumes $\bigwedge u\ v\ es.\ (u,v) \in arcs\text{-}ends\ G \Longrightarrow P\ (v\ \#\ es) \Longrightarrow P\ (u\ \#\ v\ \#\ es)$
shows *P p*
using *assms*
proof (*induct p*)
case (*Cons u es*)
then show *?case*
proof (*cases es*)
fix *v es′* **assume** *es* = *v* # *es′*
then have (*u*,*v*) ∈ *arcs-ends G* **and** *P* (*v* # *es′*)
using *Cons* **by** (*auto elim*: *vwalk-consE*)
then show *?thesis* **using** ‹*es* = *v* # *es′*› *Cons.prems* **by** *auto*
qed *auto*
qed *auto*

lemma *vwalk-arcs-Cons*[*simp*]:
assumes *p* ≠ []
shows *vwalk-arcs* (*u* # *p*) = (*u*, *hd p*) # *vwalk-arcs p*
using *assms* **by** (*cases p*) *simp*+

lemma *vwalk-arcs-append*:
assumes *p* ≠ [] **and** *q* ≠ []
shows *vwalk-arcs* (*p* @ *q*) = *vwalk-arcs p* @ (*last p*, *hd q*) # *vwalk-arcs q*
proof −
from *assms* **obtain** *a b p′ q′* **where** *p* = *a* # *p′* **and** *q* = *b* # *q′*
by (*auto simp add*: *neq-Nil-conv*)
moreover
have *vwalk-arcs* ((*a* # *p′*) @ (*b* # *q′*))
= *vwalk-arcs* (*a* # *p′*) @ (*last* (*a* # *p′*), *b*) # *vwalk-arcs* (*b* # *q′*)
proof (*induct p′*)
case *Nil* **show** *?case* **by** *simp*
next
case (*Cons a′ p′*) **then show** *?case* **by** (*auto simp add*: *neq-Nil-conv*)
qed
ultimately
show *?thesis* **by** *auto*
qed

lemma *set-vwalk-arcs-append1*:
set (*vwalk-arcs p*) ⊆ *set* (*vwalk-arcs* (*p* @ *q*))
proof (*cases p*)
case (*Cons a p′*) **note** *p-Cons* = *Cons* **then show** *?thesis*
proof (*cases q*)
case (*Cons b q′*)
with *p-Cons* **have** *p* ≠ [] **and** *q* ≠ [] **by** *auto*
then show *?thesis* **by** (*auto simp add*: *vwalk-arcs-append*)
qed *simp*

qed *simp*

lemma *set-vwalk-arcs-append2*:
set (*vwalk-arcs q*) ⊆ *set* (*vwalk-arcs* (*p* @ *q*))
proof (*cases p*)
case (*Cons a p'*) **note** *p-Cons* = *Cons* **then show** *?thesis*
proof (*cases q*)
case (*Cons b q'*)
with *p-Cons* **have** *p* ≠ [] **and** *q* ≠ [] **by** *auto*
then show *?thesis* **by** (*auto simp add*: *vwalk-arcs-append*)
qed *simp*
qed *simp*

lemma *set-vwalk-arcs-cons*:
set (*vwalk-arcs p*) ⊆ *set* (*vwalk-arcs* (*u* # *p*))
by (*cases p*) *auto*

lemma *set-vwalk-arcs-snoc*:
assumes *p* ≠ []
shows *set* (*vwalk-arcs* (*p* @ [*a*]))
= *insert* (*last p*, *a*) (*set* (*vwalk-arcs p*))
using *assms* **proof** (*induct p*)
case *Nil* **then show** *?case* **by** *auto*
next
case (*Cons x xs*)
then show *?case*
proof (*cases xs* = [])
case *True* **then show** *?thesis* **by** *auto*
next
case *False*
have *set* (*vwalk-arcs* ((*x* # *xs*) @ [*a*]))
= *set* (*vwalk-arcs* (*x* # (*xs* @ [*a*])))
by *auto*
then show *?thesis* **using** *Cons* **and** *False*
by (*auto simp add*: *set-vwalk-arcs-cons*)
qed
qed

lemma (**in** *wf-digraph*) *vwalk-wf-digraph-consI*:
assumes *vwalk p G*
assumes (*a*, *hd p*) ∈ *arcs-ends G*
shows *vwalk* (*a* # *p*) *G*
proof
show *a* # *p* ≠ [] **by** *simp*

from *assms* **have** *a* ∈ *verts G* **and** *set p* ⊆ *verts G* **by** *auto*
then show *set* (*a* # *p*) ⊆ *verts G* **by** *auto*

from ‹*vwalk p G*› **have** *p* ≠ [] **by** *auto*

then show *set* (*vwalk-arcs* (*a* # *p*)) ⊆ *arcs-ends G*
using ‹*vwalk p G*› **and** ‹(*a*, *hd p*) ∈ *arcs-ends G*›
by (*auto simp add*: *set-vwalk-arcs-cons*)
qed

lemma *vwalkI-append-l*:
assumes *p* ≠ []
assumes *vwalk* (*p* @ *q*) *G*
shows *vwalk p G*
proof
from *assms* **show** *p* ≠ [] **and** *set p* ⊆ *verts G*
by (*auto elim*!: *vwalkE*)
have *set* (*vwalk-arcs p*) ⊆ *set* (*vwalk-arcs* (*p* @ *q*))
by (*auto simp add*: *set-vwalk-arcs-append1*)
then show *set* (*vwalk-arcs p*) ⊆ *arcs-ends G*
using *assms* **by** *blast*
qed

lemma *vwalkI-append-r*:
assumes *q* ≠ []
assumes *vwalk* (*p* @ *q*) *G*
shows *vwalk q G*
proof
from ‹*vwalk* (*p* @ *q*) *G*› **have** *set* (*p* @ *q*) ⊆ *verts G* **by** *blast*
then show *set q* ⊆ *verts G* **by** *simp*

from ‹*vwalk* (*p* @ *q*) *G*› **have** *set* (*vwalk-arcs* (*p* @ *q*)) ⊆ *arcs-ends G*
by *blast*
then show *set* (*vwalk-arcs q*) ⊆ *arcs-ends G*
by (*metis set-vwalk-arcs-append2 subset-trans*)

from ‹*q* ≠ []› **show** *q* ≠ [] **by** *assumption*
qed

lemma *vwalk-to-vpath-hd*: *hd* (*vwalk-to-vpath xs*) = *hd xs*
proof (*induct xs rule*: *vwalk-to-vpath.induct*)
case (*2 x xs*) **then show** *?case*
proof (*cases x* ∈ *set xs*)
case *True*
then have *hd* (*dropWhile* (λ*y*. *y* ≠ *x*) *xs*) = *x*
using *hd-dropWhile*[**where** *P*=λ*y*. *y* ≠ *x*] **by** *auto*
then show *?thesis* **using** *True* **and** *2* **by** *auto*
qed *auto*
qed *auto*

lemma *vwalk-to-vpath-induct3*[*consumes 0*, *case-names base in-set not-in-set*]:
assumes *P* []
assumes ⋀*x xs*. *x* ∈ *set xs* ⟹ *P* (*dropWhile* (λ*y*. *y* ≠ *x*) *xs*)
⟹ *P* (*x* # *xs*)

assumes $\bigwedge x\ xs.\ x \notin set\ xs \Longrightarrow P\ xs \Longrightarrow P\ (x\ \#\ xs)$
shows $P\ xs$
using *assms* **by** (*induct xs rule*: *vwalk-to-vpath.induct*) *auto*

lemma *vwalk-to-vpath-nonempty*:
assumes $p \neq []$
shows *vwalk-to-vpath* $p \neq []$
using *assms* **by** (*induct p rule*: *vwalk-to-vpath-induct3*) *auto*

lemma *vwalk-to-vpath-last*:
last (*vwalk-to-vpath xs*) = *last xs*
by (*induct xs rule*: *vwalk-to-vpath-induct3*)
(*auto simp add*: *dropWhile-last vwalk-to-vpath-nonempty*)

lemma *vwalk-to-vpath-subset*:
assumes $x \in set$ (*vwalk-to-vpath xs*)
shows $x \in set\ xs$
using *assms* **proof** (*induct xs rule*: *vwalk-to-vpath.induct*)
case (*2 x xs*) **then show** *?case*
by (*cases* $x \in set\ xs$) (*auto dest*: *set-dropWhileD*)
qed *simp-all*

lemma *vwalk-to-vpath-cons*:
assumes $x \notin set\ xs$
shows *vwalk-to-vpath* $(x\ \#\ xs) = x\ \#$ *vwalk-to-vpath xs*
using *assms* **by** *auto*

lemma *vwalk-to-vpath-vpath*:
assumes *vwalk p G*
shows *vpath* (*vwalk-to-vpath p*) *G*
using *assms* **proof** (*induct p rule*: *vwalk-to-vpath-induct3*)
case *base* **then show** *?case* **by** *auto*
next
case (*in-set x xs*)
have *set-neq*: $\bigwedge x\ xs.\ x \notin set\ xs \Longrightarrow \forall x' \in set\ xs.\ x' \neq x$ **by** *metis*
from ‹$x \in set\ xs$› **obtain** *ys zs* **where** $xs = ys\ @\ x\ \#\ zs$ **and** $x \notin set\ ys$
by (*metis in-set-conv-decomp-first*)
then have *vwalk-dW*: *vwalk* (*dropWhile* $(\lambda y.\ y \neq x)$ *xs*) *G*
using *in-set* **and** ‹$xs = ys\ @\ x\ \#\ zs$›
by (*auto simp add*: *dropWhile-append3 set-neq intro*: *vwalkI-append-r*[**where** *p*=$x\ \#\ ys$])
then show *?case* **using** *in-set*
by (*auto simp add*: *vwalk-dW*)
next
case (*not-in-set x xs*)
then have $x \in verts\ G$ **and** *x-notin*: $x \notin set$ (*vwalk-to-vpath xs*)
by (*auto intro*: *vwalk-to-vpath-subset*)

from *not-in-set* **show** *?case*

```
  proof (cases xs)
    case Nil then show ?thesis using not-in-set.prems by auto
  next
    case (Cons x' xs')
    have vpath (vwalk-to-vpath xs) G
      apply (rule not-in-set)
      apply (rule vwalkI-append-r[where p=[x]])
       using Cons not-in-set by auto
    then have vwalk (x # vwalk-to-vpath xs) G
      apply (auto intro!: vwalk-consI simp add: vwalk-to-vpath-hd)
       using not-in-set
       apply −
       apply (erule vwalk-consE)
        using Cons
        apply (auto intro: ‹x ∈ verts G›)
      done
    then have vpath (x # vwalk-to-vpath xs) G
      apply (rule vpathI)
      using ‹vpath (vwalk-to-vpath xs) G›
      using x-notin
      by auto
    then show ?thesis using not-in-set
      by (auto simp add: vwalk-to-vpath-cons)
  qed
qed

lemma vwalk-imp-ex-vpath:
  assumes vwalk p G
  assumes hd p = u
  assumes last p = v
  shows ∃ q. vpath q G ∧ hd q = u ∧ last q = v
by (metis assms vwalk-to-vpath-hd vwalk-to-vpath-last vwalk-to-vpath-vpath)


lemma vwalk-arcs-set-nil:
  assumes x ∈ set (vwalk-arcs p)
  shows p ≠ []
using assms by fastforce

lemma in-set-vwalk-arcs-append1:
  assumes x ∈ set (vwalk-arcs p) ∨ x ∈ set (vwalk-arcs q)
  shows x ∈ set (vwalk-arcs (p @ q))
using assms proof
  assume x ∈ set (vwalk-arcs p)
  then show x ∈ set (vwalk-arcs (p @ q))
    by (cases q = [])
       (auto simp add: vwalk-arcs-append vwalk-arcs-set-nil)
next
  assume x ∈ set (vwalk-arcs q)
```

then show $x \in$ *set* (*vwalk-arcs* (p @ q))
by (*cases* $p = []$)
(*auto simp add*: *vwalk-arcs-append vwalk-arcs-set-nil*)
qed

lemma *in-set-vwalk-arcs-append2*:
assumes *nonempty*: $p \neq []$ $q \neq []$
assumes *disj*: $x \in$ *set* (*vwalk-arcs* p) $\vee$ $x =$ (*last* p, *hd* q)
$\vee$ $x \in$ *set* (*vwalk-arcs* q)
shows $x \in$ *set* (*vwalk-arcs* (p @ q))
using *disj* **proof** (*elim disjE*)
assume $x =$ (*last* p, *hd* q)
then show $x \in$ *set* (*vwalk-arcs* (p @ q))
by (*metis nonempty in-set-conv-decomp vwalk-arcs-append*)
qed (*auto intro*: *in-set-vwalk-arcs-append1*)

lemma *arcs-in-vwalk-arcs*:
assumes $u \in$ *set* (*vwalk-arcs* p)
shows $u \in$ *set* $p \times$ *set* p
using *assms* **by** (*induct p rule*: *vwalk-arcs.induct*) *auto*

lemma *set-vwalk-arcs-rev*:
set (*vwalk-arcs* (*rev* p)) = {(v, u). (u,v) $\in$ *set* (*vwalk-arcs* p)}
proof (*induct* p)
case *Nil* **then show** *?case* **by** *auto*
next
case (*Cons* x xs)
then show *?case*
proof (*cases* $xs = []$)
case *True* **then show** *?thesis* **by** *auto*
next
case *False*
then have *set* (*vwalk-arcs* (*rev* (x # xs))) = {(*hd* xs, x)}
$\cup$ {a. *case a of* (v, u) $\Rightarrow$ (u, v) $\in$ *set* (*vwalk-arcs* xs)}
by (*simp add*: *set-vwalk-arcs-snoc last-rev Cons*)
also have ... = {a. *case a of* (v, u) $\Rightarrow$ (u, v) $\in$ *set* (*vwalk-arcs* (x # xs))}
using *False* **by** (*auto simp add*: *set-vwalk-arcs-cons*)
finally show *?thesis* **by** *assumption*
qed
qed

lemma *vpath-self*:
assumes $u \in$ *verts* G
shows *vpath* [u] G
using *assms* **by** (*intro vpathI vwalkI*, *auto*)

lemma *vwalk-verts-in-verts*:
assumes *vwalk* p G
assumes $u \in$ *set* p

shows $u \in verts\ G$
using *assms* **by** *auto*

lemma *vwalk-arcs-tl*:
$vwalk\text{-}arcs\ (tl\ xs) = tl\ (vwalk\text{-}arcs\ xs)$
by (*induct xs rule*: *vwalk-arcs.induct*) *simp-all*

lemma *vwalk-arcs-butlast*:
$vwalk\text{-}arcs\ (butlast\ xs) = butlast\ (vwalk\text{-}arcs\ xs)$
proof (*induct xs rule*: *rev-induct*)
case (*snoc x xs*) **thus** *?case*
proof (*cases xs* = [])
case *True* **with** *snoc* **show** *?thesis* **by** *simp*
next
case *False*
hence $vwalk\text{-}arcs\ (xs\ @\ [x]) = vwalk\text{-}arcs\ xs\ @\ [(last\ xs, x)]$ **using** *vwalk-arcs-append*
by *force*
with *snoc* **show** *?thesis* **by** *simp*
qed
qed *simp*

lemma *vwalk-arcs-tl-empty*:
$vwalk\text{-}arcs\ xs = [] \Longrightarrow vwalk\text{-}arcs\ (tl\ xs) = []$
by (*induct xs rule*: *vwalk-arcs.induct*) *simp-all*

lemma *vwalk-arcs-butlast-empty*:
$xs \neq [] \Longrightarrow vwalk\text{-}arcs\ xs = [] \Longrightarrow vwalk\text{-}arcs\ (butlast\ xs) = []$
by (*induct xs rule*: *vwalk-arcs.induct*) *simp-all*

definition *joinable* :: *'a vwalk* ⇒ *'a vwalk* ⇒ *bool* **where**
$joinable\ p\ q \equiv last\ p = hd\ q \wedge p \neq [] \wedge q \neq []$

definition *vwalk-join* :: *'a list* ⇒ *'a list* ⇒ *'a list*
(**infixr** ⊕ *65*) **where**
$p \oplus q \equiv p\ @\ tl\ q$

lemma *joinable-Nil-l-iff*[*simp*]: $joinable\ []\ p = False$
and *joinable-Nil-r-iff*[*simp*]: $joinable\ q\ [] = False$
by (*auto simp*: *joinable-def*)

lemma *joinable-Cons-l-iff*[*simp*]: $p \neq [] \Longrightarrow joinable\ (v\ \#\ p)\ q = joinable\ p\ q$
and *joinable-Snoc-r-iff*[*simp*]: $q \neq [] \Longrightarrow joinable\ p\ (q\ @\ [v]) = joinable\ p\ q$
by (*auto simp*: *joinable-def*)

lemma *joinableI*[*intro*,*simp*]:
assumes $last\ p = hd\ q\ p \neq []\ q \neq []$
shows $joinable\ p\ q$
using *assms* **by** (*simp add*: *joinable-def*)

lemma *vwalk-join-non-Nil*[*simp*]:
assumes $p \neq []$
shows $p \oplus q \neq []$
unfolding *vwalk-join-def* **using** *assms* **by** *simp*

lemma *vwalk-join-Cons*[*simp*]:
assumes $p \neq []$
shows $(u \# p) \oplus q = u \# p \oplus q$
unfolding *vwalk-join-def* **using** *assms* **by** *simp*

lemma *vwalk-join-def2*:
assumes *joinable p q*
shows $p \oplus q =$ *butlast p* @ *q*
proof −
from *assms* **have** $p \neq []$ **and** $q \neq []$ **by** (*simp add*: *joinable-def*)+
then have *vwalk-join p q* = *butlast p* @ *last p* # *tl q*
unfolding *vwalk-join-def* **by** *simp*
then show *?thesis* **using** *assms* **by** (*simp add*: *joinable-def*)
qed

lemma *vwalk-join-hd′*:
assumes $p \neq []$
shows *hd* $(p \oplus q) =$ *hd p*
using *assms* **by** (*auto simp add*: *vwalk-join-def*)

lemma *vwalk-join-hd*:
assumes *joinable p q*
shows *hd* $(p \oplus q) =$ *hd p*
using *assms* **by** (*auto simp add*: *vwalk-join-def joinable-def*)

lemma *vwalk-join-last*:
assumes *joinable p q*
shows *last* $(p \oplus q) =$ *last q*
using *assms* **by** (*auto simp add*: *vwalk-join-def2 joinable-def*)

lemma *vwalk-join-Nil*[*simp*]:
$p \oplus [] = p$
by (*simp add*: *vwalk-join-def*)

lemma *vwalk-joinI-vwalk′*:
assumes *vwalk p G*
assumes *vwalk q G*
assumes *last p* = *hd q*
shows *vwalk* $(p \oplus q)$ *G*
proof (*unfold vwalk-join-def*, *rule vwalkI*)
have *set p* ⊆ *verts G* **and** *set q* ⊆ *verts G*
using ‹*vwalk p G*› **and** ‹*vwalk q G*› **by** *blast*+
then show *set* $(p$ @ *tl q*$)$ ⊆ *verts G*

```
    by (cases q) auto
next
  show p @ tl q ≠ [] using ‹vwalk p G› by auto
next
  have pe-p: set (vwalk-arcs p) ⊆ arcs-ends G
    using ‹vwalk p G› by blast
  have pe-q': set (vwalk-arcs (tl q)) ⊆ arcs-ends G
  proof −
    have set (vwalk-arcs (tl q)) ⊆ set (vwalk-arcs q)
      by (cases q) (simp-all add: set-vwalk-arcs-cons)
    then show ?thesis using ‹vwalk q G› by blast
  qed

  show set (vwalk-arcs (p @ tl q)) ⊆ arcs-ends G
  proof (cases tl q)
    case Nil then show ?thesis using pe-p by auto
  next
    case (Cons x xs)
    then have nonempty: p ≠ [] tl q ≠ []
      using ‹vwalk p G› by auto
    moreover
    have (hd q, hd (tl q)) ∈ set (vwalk-arcs q)
      using ‹vwalk q G› Cons by (cases q) auto
    ultimately show ?thesis
      using ‹vwalk q G›
      by (auto simp: pe-p pe-q' ‹last p = hd q› vwalk-arcs-append)
  qed
qed

lemma vwalk-joinI-vwalk:
  assumes vwalk p G
  assumes vwalk q G
  assumes joinable p q
  shows vwalk (p ⊕ q) G
using assms vwalk-joinI-vwalk' by (auto simp: joinable-def)

lemma vwalk-join-split:
  assumes u ∈ set p
  shows ∃ q r. p = q ⊕ r
  ∧ last q = u ∧ hd r = u ∧ q ≠ [] ∧ r ≠ []
proof −
  from ‹u ∈ set p›
  obtain pre-p post-p where p = pre-p @ u # post-p
    by atomize-elim (auto simp add: split-list)
  then have p = (pre-p @ [u]) ⊕ (u # post-p)
    unfolding vwalk-join-def by simp
  then show ?thesis by fastforce
qed
```

lemma *vwalkI-vwalk-join-l*:
assumes $p \neq []$
assumes *vwalk* $(p \oplus q)$ G
shows *vwalk* p G
using *assms* **unfolding** *vwalk-join-def*
by (*auto intro*: *vwalkI-append-l*)

lemma *vwalkI-vwalk-join-r*:
assumes *joinable* p q
assumes *vwalk* $(p \oplus q)$ G
shows *vwalk* q G
using *assms*
by (*auto simp add*: *vwalk-join-def2 joinable-def intro*: *vwalkI-append-r*)

lemma *vwalk-join-assoc'*:
assumes $p \neq []$ $q \neq []$
shows $(p \oplus q) \oplus r = p \oplus q \oplus r$
using *assms* **by** (*simp add*: *vwalk-join-def*)

lemma *vwalk-join-assoc*:
assumes *joinable* p q *joinable* q r
shows $(p \oplus q) \oplus r = p \oplus q \oplus r$
using *assms* **by** (*simp add*: *vwalk-join-def joinable-def*)

lemma *joinable-vwalk-join-r-iff*:
joinable p $(q \oplus r) \longleftrightarrow$ *joinable* p $q \vee (q = [] \wedge$ *joinable* p $(tl\ r))$
by (*cases* q) (*auto simp add*: *vwalk-join-def joinable-def*)

lemma *joinable-vwalk-join-l-iff*:
assumes *joinable* p q
shows *joinable* $(p \oplus q)$ $r \longleftrightarrow$ *joinable* q r (**is** *?L* $\longleftrightarrow$ *?R*)
using *assms* **by** (*auto simp*: *joinable-def vwalk-join-last*)

lemmas *joinable-simps* =
joinable-vwalk-join-l-iff
joinable-vwalk-join-r-iff

lemma *joinable-cyclic-omit*:
assumes *joinable* p q *joinable* q r *joinable* q q
shows *joinable* p r
using *assms* **by** (*metis joinable-def*)

lemma *joinable-non-Nil*:
assumes *joinable* p q
shows $p \neq []$ $q \neq []$
using *assms* **by** (*simp-all add*: *joinable-def*)

lemma *vwalk-join-vwalk-length*[*simp*]:
assumes *joinable* p q

shows *vwalk-length* $(p \oplus q)$ = *vwalk-length p* + *vwalk-length q*
using *assms* **unfolding** *vwalk-join-def*
by (*simp add*: *less-eq-Suc-le*[*symmetric*] *joinable-non-Nil*)

lemma *vwalk-join-arcs*:
assumes *joinable p q*
shows *vwalk-arcs* $(p \oplus q)$ = *vwalk-arcs p* @ *vwalk-arcs q*
using *assms*
proof (*induct p*)
case (*Cons v vs*) **then show** *?case*
by (*cases vs* = []
(*auto simp*: *vwalk-join-hd*, *simp add*: *joinable-def vwalk-join-def*)
qed *simp*

lemma *vwalk-join-arcs1*:
assumes *set* (*vwalk-arcs p*) ⊆ *E*
assumes *p* = *q* ⊕ *r*
shows *set* (*vwalk-arcs q*) ⊆ *E*
by (*metis assms vwalk-join-def set-vwalk-arcs-append1 subset-trans*)

lemma *vwalk-join-arcs2*:
assumes *set* (*vwalk-arcs p*) ⊆ *E*
assumes *joinable q r*
assumes *p* = *q* ⊕ *r*
shows *set* (*vwalk-arcs r*) ⊆ *E*
using *assms* **by** (*simp add*: *vwalk-join-arcs*)

definition *concat-vpath* :: *'a list* ⇒ *'a list* ⇒ *'a list* **where**
concat-vpath p q ≡ *vwalk-to-vpath* $(p \oplus q)$

lemma *concat-vpath-is-vpath*:
assumes *p-props*: *vwalk p G hd p* = *u last p* = *v*
assumes *q-props*: *vwalk q G hd q* = *v last q* = *w*
shows *vpath* (*concat-vpath p q*) *G* ∧ *hd* (*concat-vpath p q*) = *u*
∧ *last* (*concat-vpath p q*) = *w*
proof (*intro conjI*)
have *joinable*: *joinable p q* **using** *assms* **by** *auto*

show *vpath* (*concat-vpath p q*) *G*
unfolding *concat-vpath-def* **using** *assms* **and** *joinable*
by (*auto intro*: *vwalk-to-vpath-vpath vwalk-joinI-vwalk*)

show *hd* (*concat-vpath p q*) = *u last* (*concat-vpath p q*) = *w*
unfolding *concat-vpath-def* **using** *assms* **and** *joinable*
by (*auto simp*: *vwalk-to-vpath-hd vwalk-to-vpath-last*
vwalk-join-hd vwalk-join-last)
qed

lemma *concat-vpath-exists*:
  assumes *p-props*: *vwalk p G hd p = u last p = v*
  assumes *q-props*: *vwalk q G hd q = v last q = w*
  obtains *r* **where** *vpath r G hd r = u last r = w*
using *concat-vpath-is-vpath*[*OF assms*] **by** *blast*

lemma (**in** *fin-digraph*) *vpaths-finite*:
  shows *finite* {*p*. *vpath p G*}
proof −
  have {*p*. *vpath p G*}
    ⊆ {*xs*. *set xs* ⊆ *verts G* ∧ *length xs* ≤ *card* (*verts G*)}
  proof (*clarify*, *rule conjI*)
    fix *p* **assume** *vpath p G*
    then show *set p* ⊆ *verts G* **by** *blast*

    from ‹*vpath p G*› **have** *length p* = *card* (*set p*)
      by (*auto simp add*: *distinct-card*)
    also have . . . ≤ *card* (*verts G*)
      using ‹*vpath p G*›
      by (*auto intro*!: *card-mono elim*!: *vpathE*)
    finally show *length p* ≤ *card* (*verts G*) **.**
  qed
  moreover
  have *finite* {*xs*. *set xs* ⊆ *verts G* ∧ *length xs* ≤ *card* (*verts G*)}
    by (*intro finite-lists-length-le*) *auto*
  ultimately show *?thesis* **by** (*rule finite-subset*)
qed

lemma (**in** *wf-digraph*) *reachable-vwalk-conv*:
  u →**G v* ⟷ (∃ *p*. *vwalk p G* ∧ *hd p* = *u* ∧ *last p* = *v*) (**is** *?L* ⟷ *?R*)
proof
  assume *?L* **then show** *?R*
  proof (*induct rule*: *converse-reachable-induct*)
    case *base* **then show** *?case*
      by (*rule-tac x*=[*v*] **in** *exI*)
        (*auto simp*: *vwalk-def arcs-ends-conv*)
  next
    case (*step u w*)
    then obtain *p* **where** *vwalk p G hd p* = *w last p* = *v* **by** *auto*
    then have *vwalk* (*u*#*p*) *G* ∧ *hd* (*u*#*p*) = *u* ∧ *last* (*u*#*p*) = *v*
      using *step* **by** (*auto intro*!: *vwalk-consI intro*: *adj-in-verts*)
    then show *?case* **..**
  qed
next
  assume *?R*
  then obtain *p* **where** *vwalk p G hd p* = *u last p* = *v* **by** *auto*
  with ‹*vwalk p G*› **show** *?L*
  proof (*induct p arbitrary*: *u rule*: *vwalk-induct*)
    case (*Base u*) **then show** *?case* **by** *auto*

```
  next
    case (Cons w x es)
    then have u →_G x using Cons by auto
    show ?case
      apply (rule adj-reachable-trans)
      apply fact
      apply (rule Cons)
      using Cons by (auto elim: vwalk-consE)
  qed
qed

lemma (in wf-digraph) reachable-vpath-conv:
  u →*_G v ⟷ (∃ p. vpath p G ∧ hd p = u ∧ last p = v) (is ?L ⟷ ?R)
proof
  assume ?L then obtain p where vwalk p G hd p = u last p = v
    by (auto simp: reachable-vwalk-conv)
  then show ?R
    by (auto intro: exI[where x=vwalk-to-vpath p]
      simp: vwalk-to-vpath-hd vwalk-to-vpath-last vwalk-to-vpath-vpath)
qed (auto simp: reachable-vwalk-conv)

lemma in-set-vwalk-arcsE:
  assumes (u,v) ∈ set (vwalk-arcs p)
  obtains u ∈ set p v ∈ set p
using assms
by (induct p rule: vwalk-arcs.induct) auto

lemma vwalk-rev-ex:
  assumes symmetric G
  assumes vwalk p G
  shows vwalk (rev p) G
using assms
proof (induct p)
  case Nil then show ?case by simp
next
  case (Cons x xs)
  then show ?case proof (cases xs = [])
    case True then show ?thesis using Cons by auto
  next
    case False
    then have vwalk xs G using ‹vwalk (x # xs) G›
      by (metis vwalk-consE)
    then have vwalk (rev xs) G using Cons by blast
    have vwalk (rev (x # xs)) G
    proof (rule vwalkI)
      have set (x # xs) ⊆ verts G using ‹vwalk (x # xs) G› by blast
      then show set (rev (x # xs)) ⊆ verts G by auto
    next
      have set (vwalk-arcs (x # xs)) ⊆ arcs-ends G
```

```
      using ‹vwalk (x # xs) G› by auto
    then show set (vwalk-arcs (rev (x # xs))) ⊆ arcs-ends G
      using ‹symmetric G›
      by (simp only: set-vwalk-arcs-rev)
        (auto intro: arcs-ends-symmetric)
  next
    show rev (x # xs) ≠ [] by auto
  qed
  then show vwalk (rev (x # xs)) G by auto
 qed
qed

lemma vwalk-singleton[simp]: vwalk [u] G = (u ∈ verts G)
 by auto

lemma (in wf-digraph) vwalk-Cons-Cons[simp]:
 vwalk (u # v # ws) G = ((u,v) ∈ arcs-ends G ∧ vwalk (v # ws) G)
 by (force elim: vwalk-consE intro: vwalk-consI)

lemma (in wf-digraph) awalk-imp-vwalk:
 assumes awalk u p v shows vwalk (awalk-verts u p) G
 using assms
 by (induct p arbitrary: u rule: vwalk-arcs.induct)
   (force simp: awalk-simps dest: in-arcs-imp-in-arcs-ends)+

end


theory Digraph-Component-Vwalk
imports
 Digraph-Component
 Vertex-Walk
begin
```

10 Lemmas for Vertex Walks

```
lemma vwalkI-subgraph:
 assumes vwalk p H
 assumes subgraph H G
 shows vwalk p G
proof
 show set p ⊆ verts G and p ≠ []
   using assms by (auto simp add: subgraph-def vwalk-def)

 have set (vwalk-arcs p) ⊆ arcs-ends H
   using assms by (simp add: vwalk-def)
 also have ... ⊆ arcs-ends G
   using ‹subgraph H G› by (rule arcs-ends-mono)
 finally show set (vwalk-arcs p) ⊆ arcs-ends G .
```

qed

lemma *vpathI-subgraph*:
  assumes *vpath p G*
  assumes *subgraph G H*
  shows *vpath p H*
using *assms* **by** (*auto intro*: *vwalkI-subgraph*)

lemma (**in** *loopfree-digraph*) *vpathI-arc*:
  assumes (*a*,*b*) ∈ *arcs-ends G*
  shows *vpath* [*a*,*b*] *G*
using *assms*
by (*intro vpathI vwalkI*) (*auto intro*: *adj-in-verts adj-not-same*)

end
theory *Digraph-Isomorphism* **imports**
  Arc-Walk
  Digraph
  Digraph-Component
begin

11 Isomorphisms of Digraphs

record ('*a*,'*b*,'*aa*,'*bb*) *digraph-isomorphism* =
  iso-verts :: '*a* ⇒ '*aa*
  iso-arcs :: '*b* ⇒ '*bb*
  iso-head :: '*bb* ⇒ '*aa*
  iso-tail :: '*bb* ⇒ '*aa*

definition (**in** *pre-digraph*) *digraph-isomorphism* :: ('*a*,'*b*,'*aa*,'*bb*) *digraph-isomorphism*
⇒ *bool* **where**
  digraph-isomorphism hom ≡
    wf-digraph G ∧
    inj-on (*iso-verts hom*) (*verts G*) ∧
    inj-on (*iso-arcs hom*) (*arcs G*) ∧
    (∀ *a* ∈ *arcs G*.
    iso-verts hom (*tail G a*) = *iso-tail hom* (*iso-arcs hom a*) ∧
    iso-verts hom (*head G a*) = *iso-head hom* (*iso-arcs hom a*))

definition (**in** *pre-digraph*) *inv-iso* :: ('*a*,'*b*,'*aa*,'*bb*) *digraph-isomorphism* ⇒ ('*aa*,'*bb*,'*a*,'*b*)
digraph-isomorphism **where**
  inv-iso hom ≡ (|
    iso-verts = *the-inv-into* (*verts G*) (*iso-verts hom*),
    iso-arcs = *the-inv-into* (*arcs G*) (*iso-arcs hom*),
    iso-head = *head G*,
    iso-tail = *tail G*
    |)

definition *app-iso*

:: ('a,'b,'aa,'bb) *digraph-isomorphism* ⇒ ('a,'b) *pre-digraph* ⇒ ('aa,'bb) *pre-digraph*
where
app-iso hom G ≡ (| *verts* = *iso-verts hom* ‘ *verts G*, *arcs* = *iso-arcs hom* ‘ *arcs G*,
tail = *iso-tail hom*, *head* = *iso-head hom* |)

definition *digraph-iso* :: ('a,'b) *pre-digraph* ⇒ ('c,'d) *pre-digraph* ⇒ *bool* **where**
digraph-iso G H ≡ ∃*f*. *pre-digraph.digraph-isomorphism G f* ∧ *H* = *app-iso f G*

lemma *verts-app-iso*: *verts* (*app-iso hom G*) = *iso-verts hom* ‘ *verts G*
and *arcs-app-iso*: *arcs* (*app-iso hom G*) = *iso-arcs hom* ‘*arcs G*
and *tail-app-iso*: *tail* (*app-iso hom G*) = *iso-tail hom*
and *head-app-iso*: *head* (*app-iso hom G*) = *iso-head hom*
by (*auto simp*: *app-iso-def*)

lemmas *app-iso-simps*[*simp*] = *verts-app-iso arcs-app-iso tail-app-iso head-app-iso*

context *pre-digraph* **begin**

lemma
assumes *digraph-isomorphism hom*
shows *iso-verts-inv-iso*: ⋀*u*. *u* ∈ *verts G* ⟹ *iso-verts* (*inv-iso hom*) (*iso-verts hom u*) = *u*
and *iso-arcs-inv-iso*: ⋀*a*. *a* ∈ *arcs G* ⟹ *iso-arcs* (*inv-iso hom*) (*iso-arcs hom a*) = *a*
and *iso-verts-iso-inv*: ⋀*u*. *u* ∈ *verts* (*app-iso hom G*) ⟹ *iso-verts hom* (*iso-verts* (*inv-iso hom*) *u*) = *u*
and *iso-arcs-iso-inv*: ⋀*a*. *a* ∈ *arcs* (*app-iso hom G*) ⟹ *iso-arcs hom* (*iso-arcs* (*inv-iso hom*) *a*) = *a*
and *iso-tail-inv-iso*: *iso-tail* (*inv-iso hom*) = *tail G*
and *iso-head-inv-iso*: *iso-head* (*inv-iso hom*) = *head G*
and *verts-app-inv-iso*:*iso-verts* (*inv-iso hom*) ‘ *iso-verts hom* ‘ *verts G* = *verts G*
and *arcs-app-inv-iso*:*iso-arcs* (*inv-iso hom*) ‘ *iso-arcs hom* ‘ *arcs G* = *arcs G*
using *assms* **by** (*auto simp*: *inv-iso-def digraph-isomorphism-def the-inv-into-f-f*)

lemmas *iso-inv-simps*[*simp*] =
iso-verts-inv-iso iso-verts-iso-inv
iso-arcs-inv-iso iso-arcs-iso-inv
verts-app-inv-iso arcs-app-inv-iso
iso-tail-inv-iso iso-head-inv-iso

lemma *app-iso-inv*[*simp*]:
assumes *digraph-isomorphism hom*
shows *app-iso* (*inv-iso hom*) (*app-iso hom G*) = *G*
using *assms* **by** (*intro pre-digraph.equality*) (*auto intro*: *rev-image-eqI*)

lemma *iso-verts-eq-iff*[*simp*]:
assumes *digraph-isomorphism hom u* ∈ *verts G v* ∈ *verts G*

shows *iso-verts hom u = iso-verts hom v* $\longleftrightarrow$ *u = v*
using *assms* **by** (*auto simp*: *digraph-isomorphism-def dest*: *inj-onD*)

lemma *iso-arcs-eq-iff*[*simp*]:
assumes *digraph-isomorphism hom e1* $\in$ *arcs G e2* $\in$ *arcs G*
shows *iso-arcs hom e1 = iso-arcs hom e2* $\longleftrightarrow$ *e1 = e2*
using *assms* **by** (*auto simp*: *digraph-isomorphism-def dest*: *inj-onD*)

lemma
assumes *digraph-isomorphism hom e* $\in$ *arcs G*
shows *iso-verts-tail*: *iso-tail hom* (*iso-arcs hom e*) = *iso-verts hom* (*tail G e*)
and *iso-verts-head*: *iso-head hom* (*iso-arcs hom e*) = *iso-verts hom* (*head G e*)
using *assms* **unfolding** *digraph-isomorphism-def* **by** *auto*

lemma *digraph-isomorphism-inj-on-arcs*:
digraph-isomorphism hom $\Longrightarrow$ *inj-on* (*iso-arcs hom*) (*arcs G*)
by (*auto simp*: *digraph-isomorphism-def*)

lemma *digraph-isomorphism-inj-on-verts*:
digraph-isomorphism hom $\Longrightarrow$ *inj-on* (*iso-verts hom*) (*verts G*)
by (*auto simp*: *digraph-isomorphism-def*)

end

lemma (**in** *wf-digraph*) *wf-digraphI-app-iso*[*intro?*]:
assumes *digraph-isomorphism hom*
shows *wf-digraph* (*app-iso hom G*)
proof *unfold-locales*
fix *e* **assume** *e* $\in$ *arcs* (*app-iso hom G*)
then obtain *e′* **where** *e′*: *e′* $\in$ *arcs G iso-arcs hom e′ = e*
by *auto*
then have *iso-verts hom* (*head G e′*) $\in$ *verts* (*app-iso hom G*)
iso-verts hom (*tail G e′*) $\in$ *verts* (*app-iso hom G*)
by *auto*
then show *tail* (*app-iso hom G*) *e* $\in$ *verts* (*app-iso hom G*)
head (*app-iso hom G*) *e* $\in$ *verts* (*app-iso hom G*)
using *e′ assms* **by** (*auto simp*: *iso-verts-tail iso-verts-head*)
qed

lemma (**in** *fin-digraph*) *fin-digraphI-app-iso*[*intro?*]:
assumes *digraph-isomorphism hom*
shows *fin-digraph* (*app-iso hom G*)
proof −
interpret *H*: *wf-digraph app-iso hom G* **using** *assms* **..**
show *?thesis* **by** *unfold-locales auto*
qed

context *wf-digraph* **begin**

lemma *digraph-isomorphism-invI*:
assumes *digraph-isomorphism hom* **shows** *pre-digraph.digraph-isomorphism* (*app-iso hom G*) (*inv-iso hom*)
proof (*unfold pre-digraph.digraph-isomorphism-def*, *safe*)
show *inj-on* (*iso-verts* (*inv-iso hom*)) (*verts* (*app-iso hom G*))
inj-on (*iso-arcs* (*inv-iso hom*)) (*arcs* (*app-iso hom G*))
using *assms* **unfolding** *pre-digraph.digraph-isomorphism-def inv-iso-def*
by (*auto intro*: *inj-on-the-inv-into*)
next
show *wf-digraph* (*app-iso hom G*) **using** *assms* **..**
next
fix *a* **assume** *a* ∈ *arcs* (*app-iso hom G*)
then obtain *b* **where** *B*: *a* = *iso-arcs hom b b* ∈ *arcs G*
by *auto*

with *assms* **have** [*simp*]:
iso-tail hom (*iso-arcs hom b*) = *iso-verts hom* (*tail G b*)
iso-head hom (*iso-arcs hom b*) = *iso-verts hom* (*head G b*)
inj-on (*iso-arcs hom*) (*arcs G*)
inj-on (*iso-verts hom*) (*verts G*)
by (*auto simp*: *digraph-isomorphism-def*)

from *B* **show** *iso-verts* (*inv-iso hom*) (*tail* (*app-iso hom G*) *a*)
= *iso-tail* (*inv-iso hom*) (*iso-arcs* (*inv-iso hom*) *a*)
by (*auto simp*: *inv-iso-def the-inv-into-f-f*)
from *B* **show** *iso-verts* (*inv-iso hom*) (*head* (*app-iso hom G*) *a*)
= *iso-head* (*inv-iso hom*) (*iso-arcs* (*inv-iso hom*) *a*)
by (*auto simp*: *inv-iso-def the-inv-into-f-f*)
qed

lemma *awalk-app-isoI*:
assumes *awalk u p v* **and** *hom*: *digraph-isomorphism hom*
shows *pre-digraph.awalk* (*app-iso hom G*) (*iso-verts hom u*) (*map* (*iso-arcs hom*) *p*) (*iso-verts hom v*)
proof −
interpret *H*: *wf-digraph app-iso hom G* **using** *hom* **..**
from *assms* **show** *?thesis*
by (*induct p arbitrary*: *u*)
(*auto simp*: *awalk-simps H.awalk-simps iso-verts-head iso-verts-tail*)
qed

lemma *awalk-app-isoD*:
assumes *w*: *pre-digraph.awalk* (*app-iso hom G*) *u p v* **and** *hom*: *digraph-isomorphism hom*
shows *awalk* (*iso-verts* (*inv-iso hom*) *u*) (*map* (*iso-arcs* (*inv-iso hom*)) *p*) (*iso-verts* (*inv-iso hom*) *v*)
proof −
interpret *H*: *wf-digraph app-iso hom G* **using** *hom* **..**

from *assms* **show** *?thesis*
by (*induct p arbitrary*: *u*)
(*force simp*: *awalk-simps H.awalk-simps iso-verts-head iso-verts-tail*)+
qed

lemma *awalk-verts-app-iso-eq*:
assumes *digraph-isomorphism hom* **and** *awalk u p v*
shows *pre-digraph.awalk-verts* (*app-iso hom G*) (*iso-verts hom u*) (*map* (*iso-arcs hom*) *p*)
= *map* (*iso-verts hom*) (*awalk-verts u p*)
using *assms*
by (*induct p arbitrary*: *u*)
(*auto simp*: *pre-digraph.awalk-verts.simps iso-verts-head iso-verts-tail awalk-Cons-iff*)

lemma *arcs-ends-app-iso-eq*:
assumes *digraph-isomorphism hom*
shows *arcs-ends* (*app-iso hom G*) = (λ(*u*,*v*). (*iso-verts hom u*, *iso-verts hom v*)) ‘ *arcs-ends G*
using *assms* **by** (*auto simp*: *arcs-ends-conv image-image iso-verts-head iso-verts-tail intro*!: *rev-image-eqI*)

lemma *in-arcs-app-iso-eq*:
assumes *digraph-isomorphism hom* **and** *u* ∈ *verts G*
shows *in-arcs* (*app-iso hom G*) (*iso-verts hom u*) = *iso-arcs hom* ‘ *in-arcs G u*
using *assms* **unfolding** *in-arcs-def* **by** (*auto simp*: *iso-verts-head*)

lemma *out-arcs-app-iso-eq*:
assumes *digraph-isomorphism hom* **and** *u* ∈ *verts G*
shows *out-arcs* (*app-iso hom G*) (*iso-verts hom u*) = *iso-arcs hom* ‘ *out-arcs G u*
using *assms* **unfolding** *out-arcs-def* **by** (*auto simp*: *iso-verts-tail*)

lemma *in-degree-app-iso-eq*:
assumes *digraph-isomorphism hom* **and** *u* ∈ *verts G*
shows *in-degree* (*app-iso hom G*) (*iso-verts hom u*) = *in-degree G u*
unfolding *in-degree-def in-arcs-app-iso-eq*[*OF assms*]
proof (*rule card-image*)
from *assms* **show** *inj-on* (*iso-arcs hom*) (*in-arcs G u*)
unfolding *digraph-isomorphism-def* **by** − (*rule subset-inj-on*, *auto*)
qed

lemma *out-degree-app-iso-eq*:
assumes *digraph-isomorphism hom* **and** *u* ∈ *verts G*
shows *out-degree* (*app-iso hom G*) (*iso-verts hom u*) = *out-degree G u*
unfolding *out-degree-def out-arcs-app-iso-eq*[*OF assms*]
proof (*rule card-image*)
from *assms* **show** *inj-on* (*iso-arcs hom*) (*out-arcs G u*)

unfolding *digraph-isomorphism-def* **by** − (*rule subset-inj-on*, *auto*)
qed

lemma *in-arcs-app-iso-eq′*:
assumes *digraph-isomorphism hom* **and** $u \in$ *verts* (*app-iso hom G*)
shows *in-arcs* (*app-iso hom G*) *u* = *iso-arcs hom* ‘ *in-arcs G* (*iso-verts* (*inv-iso hom*) *u*)
using *assms in-arcs-app-iso-eq*[*of hom iso-verts* (*inv-iso hom*) *u*] **by** *auto*

lemma *out-arcs-app-iso-eq′*:
assumes *digraph-isomorphism hom* **and** $u \in$ *verts* (*app-iso hom G*)
shows *out-arcs* (*app-iso hom G*) *u* = *iso-arcs hom* ‘ *out-arcs G* (*iso-verts* (*inv-iso hom*) *u*)
using *assms out-arcs-app-iso-eq*[*of hom iso-verts* (*inv-iso hom*) *u*] **by** *auto*

lemma *in-degree-app-iso-eq′*:
assumes *digraph-isomorphism hom* **and** $u \in$ *verts* (*app-iso hom G*)
shows *in-degree* (*app-iso hom G*) *u* = *in-degree G* (*iso-verts* (*inv-iso hom*) *u*)
using *assms in-degree-app-iso-eq*[*of hom iso-verts* (*inv-iso hom*) *u*] **by** *auto*

lemma *out-degree-app-iso-eq′*:
assumes *digraph-isomorphism hom* **and** $u \in$ *verts* (*app-iso hom G*)
shows *out-degree* (*app-iso hom G*) *u* = *out-degree G* (*iso-verts* (*inv-iso hom*) *u*)
using *assms out-degree-app-iso-eq*[*of hom iso-verts* (*inv-iso hom*) *u*] **by** *auto*

lemmas *app-iso-eq* =
awalk-verts-app-iso-eq
arcs-ends-app-iso-eq
in-arcs-app-iso-eq′
out-arcs-app-iso-eq′
in-degree-app-iso-eq′
out-degree-app-iso-eq′

lemma *reachableI-app-iso*:
assumes *r*: *u* $\rightarrow^*$ *v* **and** *hom*: *digraph-isomorphism hom*
shows (*iso-verts hom u*) $\rightarrow^*_{\textit{app-iso hom G}}$ (*iso-verts hom v*)
proof −
interpret *H*: *wf-digraph app-iso hom G* **using** *hom* **..**
from *r* **obtain** *p* **where** *awalk u p v* **by** (*auto simp*: *reachable-awalk*)
then have *H.awalk* (*iso-verts hom u*) (*map* (*iso-arcs hom*) *p*) (*iso-verts hom v*)
using *hom* **by** (*rule awalk-app-isoI*)
then show *?thesis* **by** (*auto simp*: *H.reachable-awalk*)
qed

lemma *awalk-app-iso-eq*:
assumes *hom*: *digraph-isomorphism hom*
assumes *u* ∈ *iso-verts hom* ‘ *verts G v* ∈ *iso-verts hom* ‘ *verts G set p* ⊆ *iso-arcs hom* ‘ *arcs G*
shows *pre-digraph.awalk* (*app-iso hom G*) *u p v*

$\longleftrightarrow$ *awalk* (*iso-verts* (*inv-iso hom*) *u*) (*map* (*iso-arcs* (*inv-iso hom*)) *p*) (*iso-verts* (*inv-iso hom*) *v*)
proof −
interpret *H*: *wf-digraph app-iso hom G* **using** *hom* **..**
from *assms* **show** *?thesis*
by (*induct p arbitrary*: *u*)
(*auto simp*: *awalk-simps H.awalk-simps iso-verts-head iso-verts-tail*)
qed

lemma *reachable-app-iso-eq*:
assumes *hom*: *digraph-isomorphism hom*
assumes *u* ∈ *iso-verts hom* ‘ *verts G v* ∈ *iso-verts hom* ‘ *verts G*
shows *u* $\rightarrow^*_{app\text{-}iso\ hom\ G}$ *v* $\longleftrightarrow$ *iso-verts* (*inv-iso hom*) *u* $\rightarrow^*$ *iso-verts* (*inv-iso hom*) *v* (**is** *?L* $\longleftrightarrow$ *?R*)
proof −
interpret *H*: *wf-digraph app-iso hom G* **using** *hom* **..**

show *?thesis*
proof
assume *?L*
then obtain *p* **where** *H.awalk u p v* **by** (*auto simp*: *H.reachable-awalk*)
moreover
then have *set p* ⊆ *iso-arcs hom* ‘ *arcs G* **by** (*simp add*: *H.awalk-def*)
ultimately
show *?R* **using** *assms* **by** (*auto simp*: *awalk-app-iso-eq reachable-awalk*)
next
assume *?R*
then obtain *p0* **where** *awalk* (*iso-verts* (*inv-iso hom*) *u*) *p0* (*iso-verts* (*inv-iso hom*) *v*)
by (*auto simp*: *reachable-awalk*)
moreover
then have *set p0* ⊆ *arcs G* **by** (*simp add*: *awalk-def*)
define *p* **where** *p* = *map* (*iso-arcs hom*) *p0*
have *set p* ⊆ *iso-arcs hom* ‘ *arcs G p0* = *map* (*iso-arcs* (*inv-iso hom*)) *p*
using ‹*set p0* ⊆ -› *hom* **by** (*auto simp*: *p-def map-idI subsetD*)
ultimately
show *?L* **using** *assms* **by** (*auto simp*: *awalk-app-iso-eq*[*symmetric*] *H.reachable-awalk*)
qed
qed

lemma *connectedI-app-iso*:
assumes *c*: *connected G* **and** *hom*: *digraph-isomorphism hom*
shows *connected* (*app-iso hom G*)
proof −
have ∗: *symcl* (*arcs-ends* (*app-iso hom G*)) = (λ(*u*,*v*). (*iso-verts hom u*, *iso-verts hom v*)) ‘ *symcl* (*arcs-ends G*)
using *hom* **by** (*auto simp add*: *app-iso-eq symcl-def*)
{ fix *u v* **assume** (*u*,*v*) ∈ *rtrancl-on* (*verts G*) (*symcl* (*arcs-ends G*))
then have (*iso-verts hom u*, *iso-verts hom v*) ∈ *rtrancl-on* (*verts* (*app-iso hom*

```
G)) (symcl (arcs-ends (app-iso hom G)))
    proof induct
      case (step x y)
      have (iso-verts hom x, iso-verts hom y)
          ∈ rtrancl-on (verts (app-iso hom G)) (symcl (arcs-ends (app-iso hom G)))
        using step by (rule-tac rtrancl-on-into-rtrancl-on[where b=iso-verts hom
x]) (auto simp: ∗)
      then show ?case
        by (rule rtrancl-on-trans) (rule step)
    qed auto }
  with c show ?thesis unfolding connected-conv by auto
qed

end

lemma digraph-iso-swap:
  assumes wf-digraph G digraph-iso G H shows digraph-iso H G
proof −
  from assms obtain f where pre-digraph.digraph-isomorphism G f H = app-iso
f G
    unfolding digraph-iso-def by auto
  then have pre-digraph.digraph-isomorphism H (pre-digraph.inv-iso G f) app-iso
(pre-digraph.inv-iso G f) H = G
   using assms by (simp-all add: wf-digraph.digraph-isomorphism-invI pre-digraph.app-iso-inv)
  then show ?thesis unfolding digraph-iso-def by auto
qed

definition
  o-iso :: ('c,'d,'e,'f) digraph-isomorphism ⇒ ('a,'b,'c,'d) digraph-isomorphism ⇒
('a,'b,'e,'f) digraph-isomorphism
where
  o-iso hom2 hom1 = (|
    iso-verts = iso-verts hom2 o iso-verts hom1,
    iso-arcs = iso-arcs hom2 o iso-arcs hom1,
    iso-head = iso-head hom2,
    iso-tail = iso-tail hom2
    |)

lemma digraph-iso-trans[trans]:
  assumes digraph-iso G H digraph-iso H I shows digraph-iso G I
proof −
  from assms obtain hom1 where pre-digraph.digraph-isomorphism G hom1 H
= app-iso hom1 G
    by (auto simp: digraph-iso-def)
  moreover
  from assms obtain hom2 where pre-digraph.digraph-isomorphism H hom2 I =
app-iso hom2 H
    by (auto simp: digraph-iso-def)
  ultimately
```

have *pre-digraph.digraph-isomorphism G* (*o-iso hom2 hom1*) *I* = *app-iso* (*o-iso hom2 hom1*) *G*
apply (*auto simp*: *o-iso-def app-iso-def pre-digraph.digraph-isomorphism-def*)
apply (*rule comp-inj-on*)
apply *auto*
apply (*rule comp-inj-on*)
apply *auto*
done
then show *?thesis* **by** (*auto simp*: *digraph-iso-def*)
qed

lemma (**in** *pre-digraph*) *digraph-isomorphism-subgraphI*:
assumes *digraph-isomorphism hom*
assumes *subgraph H G*
shows *pre-digraph.digraph-isomorphism H hom*
using *assms* **by** (*auto simp*: *pre-digraph.digraph-isomorphism-def subgraph-def compatible-def intro*: *subset-inj-on*)

lemma (**in** *wf-digraph*) *verts-app-inv-iso-subgraph*:
assumes *hom*: *digraph-isomorphism hom* **and** *V* ⊆ *verts G*
shows *iso-verts* (*inv-iso hom*) ‘ *iso-verts hom* ‘ *V* = *V*
proof −
have ⋀*x*. *x* ∈ *V* ⟹ *iso-verts* (*inv-iso hom*) (*iso-verts hom x*) = *x*
using *assms* **by** *auto*
then show *?thesis* **by** (*auto simp*: *image-image cong*: *image-cong*)
qed

lemma (**in** *wf-digraph*) *arcs-app-inv-iso-subgraph*:
assumes *hom*: *digraph-isomorphism hom* **and** *A* ⊆ *arcs G*
shows *iso-arcs* (*inv-iso hom*) ‘ *iso-arcs hom* ‘ *A* = *A*
proof −
have ⋀*x*. *x* ∈ *A* ⟹ *iso-arcs* (*inv-iso hom*) (*iso-arcs hom x*) = *x*
using *assms* **by** *auto*
then show *?thesis* **by** (*auto simp*: *image-image cong*: *image-cong*)
qed

lemma (**in** *pre-digraph*) *app-iso-inv-subgraph*[*simp*]:
assumes *digraph-isomorphism hom subgraph H G*
shows *app-iso* (*inv-iso hom*) (*app-iso hom H*) = *H*
proof −
from *assms* **interpret** *wf-digraph G* **by** *auto*
have ⋀*u*. *u* ∈ *verts H* ⟹ *u* ∈ *verts G* ⋀*a*. *a* ∈ *arcs H* ⟹ *a* ∈ *arcs G*
using *assms* **by** *auto*
with *assms* **show** *?thesis*
by (*intro pre-digraph.equality*) (*auto simp*: *verts-app-inv-iso-subgraph arcs-app-inv-iso-subgraph compatible-def*)

qed

lemma (**in** *wf-digraph*) *app-iso-iso-inv-subgraph*[*simp*]:
assumes *digraph-isomorphism hom*
assumes *subg*: *subgraph H* (*app-iso hom G*)
shows *app-iso hom* (*app-iso* (*inv-iso hom*) *H*) = *H*
proof −
have ⋀*u*. *u* ∈ *verts H* ⟹ *u* ∈ *iso-verts hom* ‘ *verts G* ⋀*a*. *a* ∈ *arcs H* ⟹ *a* ∈ *iso-arcs hom* ‘ *arcs G*
using *assms* **by** (*auto simp*: *subgraph-def*)
with *assms* **show** *?thesis*
by (*intro pre-digraph.equality*) (*auto simp*: *compatible-def image-image cong*: *image-cong*)
qed

lemma (**in** *pre-digraph*) *subgraph-app-isoI′*:
assumes *hom*: *digraph-isomorphism hom*
assumes *subg*: *subgraph H H′ subgraph H′ G*
shows *subgraph* (*app-iso hom H*) (*app-iso hom H′*)
proof −
have *subgraph H G* **using** *subg* **by** (*rule subgraph-trans*)
then have *pre-digraph.digraph-isomorphism H hom pre-digraph.digraph-isomorphism H′ hom*
using *assms* **by** (*auto intro*: *digraph-isomorphism-subgraphI*)
then show *?thesis*
using *assms* **by** (*auto simp*: *subgraph-def wf-digraph.wf-digraphI-app-iso compatible-def*
intro: *digraph-isomorphism-subgraphI*)
qed

lemma (**in** *pre-digraph*) *subgraph-app-isoI*:
assumes *digraph-isomorphism hom*
assumes *subgraph H G*
shows *subgraph* (*app-iso hom H*) (*app-iso hom G*)
using *assms* **by** (*auto intro*: *subgraph-app-isoI′ wf-digraph.subgraph-refl*)

lemma (**in** *pre-digraph*) *app-iso-eq-conv*:
assumes *digraph-isomorphism hom*
assumes *subgraph H1 G subgraph H2 G*
shows *app-iso hom H1* = *app-iso hom H2* ⟷ *H1* = *H2* (**is** *?L* ⟷ *?R*)
proof
assume *?L*
then have *app-iso* (*inv-iso hom*) (*app-iso hom H1*) = *app-iso* (*inv-iso hom*) (*app-iso hom H2*)
by *simp*
with *assms* **show** *?R* **by** *auto*
qed *simp*

lemma *in-arcs-app-iso-cases*:

```
  assumes a ∈ arcs (app-iso hom G)
  obtains a0 where a = iso-arcs hom a0 a0 ∈ arcs G
  using assms by auto

lemma in-verts-app-iso-cases:
  assumes v ∈ verts (app-iso hom G)
  obtains v0 where v = iso-verts hom v0 v0 ∈ verts G
  using assms by auto

lemma (in wf-digraph) max-subgraph-iso:
  assumes hom: digraph-isomorphism hom
  assumes subg: subgraph H (app-iso hom G)
  shows pre-digraph.max-subgraph (app-iso hom G) P H
    ⟷ max-subgraph (P o app-iso hom) (app-iso (inv-iso hom) H)
proof −
  have hom-inv: pre-digraph.digraph-isomorphism (app-iso hom G) (inv-iso hom)
    using hom by (rule digraph-isomorphism-invI)
  interpret aG: wf-digraph app-iso hom G using hom ..

  have ∗: subgraph (app-iso (inv-iso hom) H) G
     using hom pre-digraph.subgraph-app-isoI'[OF hom-inv subg aG.subgraph-refl]
by simp
  define H0 where H0 = app-iso (inv-iso hom) H
  then have H0: H = app-iso hom H0 subgraph H0 G
    using hom subg ‹subgraph - G› by auto

  show ?thesis (is ?L ⟷ ?R)
  proof
    assume ?L then show ?R using assms H0
    by (auto simp: max-subgraph-def aG.max-subgraph-def pre-digraph.subgraph-app-isoI'
        subgraph-refl pre-digraph.app-iso-eq-conv)
  next
    assume ?R
    then show ?L
      using assms hom-inv pre-digraph.subgraph-app-isoI[OF hom-inv]
      apply (auto simp: max-subgraph-def aG.max-subgraph-def)
      apply (erule allE[of - app-iso (inv-iso hom) H' for H'])
     apply (auto simp: pre-digraph.subgraph-app-isoI' pre-digraph.app-iso-eq-conv)
      done
  qed
qed

lemma (in pre-digraph) max-subgraph-cong:
  assumes H = H' ⋀H''. subgraph H' H'' ⟹ subgraph H'' G ⟹ P H'' = P'
H''
  shows max-subgraph P H = max-subgraph P' H'
  using assms by (auto simp: max-subgraph-def intro: wf-digraph.subgraph-refl)

lemma (in pre-digraph) inj-on-app-iso:
```

assumes *hom*: *digraph-isomorphism hom*
assumes $S \subseteq \{H.\ subgraph\ H\ G\}$
shows *inj-on* (*app-iso hom*) *S*
using *assms* **by** (*intro inj-onI*) (*subst* (*asm*) *app-iso-eq-conv*, *auto*)

11.1 Graph Invariants

context
fixes *G hom* **assumes** *hom*: *pre-digraph.digraph-isomorphism G hom*
begin

interpretation *wf-digraph G* **using** *hom* **by** (*auto simp*: *pre-digraph.digraph-isomorphism-def*)

lemma *card-verts-iso*[*simp*]: *card* (*iso-verts hom* ‘ *verts G*) = *card* (*verts G*)
using *hom* **by** (*intro card-image digraph-isomorphism-inj-on-verts*)

lemma *card-arcs-iso*[*simp*]: *card* (*iso-arcs hom* ‘ *arcs G*) = *card* (*arcs G*)
using *hom* **by** (*intro card-image digraph-isomorphism-inj-on-arcs*)

lemma *strongly-connected-iso*[*simp*]: *strongly-connected* (*app-iso hom G*) ⟷ *strongly-connected G*
using *hom* **by** (*auto simp*: *strongly-connected-def reachable-app-iso-eq*)

lemma *subgraph-strongly-connected-iso*:
assumes *subgraph H G*
shows *strongly-connected* (*app-iso hom H*) ⟷ *strongly-connected H*
proof −
interpret *H*: *wf-digraph H* **using** ‹*subgraph H G*› **..**
have *H.digraph-isomorphism hom* **using** *hom assms* **by** (*rule digraph-isomorphism-subgraphI*)
then show *?thesis*
using *assms* **by** (*auto simp*: *strongly-connected-def H.reachable-app-iso-eq*)
qed

lemma *sccs-iso*[*simp*]: *pre-digraph.sccs* (*app-iso hom G*) = *app-iso hom* ‘ *sccs* (**is** *?L* = *?R*)
proof (*intro set-eqI iffI*)
fix *x* **assume** *x* ∈ *?L*
then have *subgraph x* (*app-iso hom G*)
by (*auto simp*: *pre-digraph.sccs-def*)
then show *x* ∈ *?R*
using ‹*x* ∈ *?L*› *hom* **by** (*auto simp*: *pre-digraph.sccs-altdef2 max-subgraph-iso subgraph-strongly-connected-iso cong*: *max-subgraph-cong intro*: *rev-image-eqI*)
next
fix *x* **assume** *x* ∈ *?R*
then obtain *x0* **where** *x0* ∈ *sccs x* = *app-iso hom x0* **by** *auto*
then show *x* ∈ *?L*
using *hom* **by** (*auto simp*: *pre-digraph.sccs-altdef2 max-subgraph-iso subgraph-app-isoI subgraphI-max-subgraph subgraph-strongly-connected-iso cong*: *max-subgraph-cong*)

qed

lemma *card-sccs-iso*[*simp*]: *card* (*app-iso hom* ‘ *sccs*) = *card sccs*
apply (*rule card-image*)
using *hom*
apply (*rule inj-on-app-iso*)
apply *auto*
done

end

end
theory *Auxiliary*
imports
HOL−Library.FuncSet
HOL−Combinatorics.Orbits
begin

lemma *funpow-invs*:
assumes $m \leq n$ **and** *inv*: $\bigwedge x.\ f\ (g\ x) = x$
shows $(f \frown m)\ ((g \frown n)\ x) = (g \frown (n - m))\ x$
using ‹$m \leq n$›
proof (*induction m*)
case (*Suc m*)
moreover then have $n - m = Suc\ (n - Suc\ m)$ **by** *auto*
ultimately show *?case* **by** (*auto simp*: *inv*)
qed *simp*

12 Permutation Domains

definition *has-dom* :: $('a \Rightarrow 'a) \Rightarrow 'a\ set \Rightarrow bool$ **where**
has-dom f S ≡ ∀ *s*. *s* ∉ *S* ⟶ *f s* = *s*

lemma *has-domD*: *has-dom f S* ⟹ *x* ∉ *S* ⟹ *f x* = *x*
by (*auto simp*: *has-dom-def*)

lemma *has-domI*: $(\bigwedge x.\ x \notin S \Longrightarrow f\ x = x) \Longrightarrow$ *has-dom f S*
by (*auto simp*: *has-dom-def*)

lemma *permutes-conv-has-dom*:
f permutes S ⟷ *bij f* ∧ *has-dom f S*
by (*auto simp*: *permutes-def has-dom-def bij-iff*)

13 Segments

inductive-set *segment* :: $('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a\ set$ **for** *f a b* **where**
base: *f a* ≠ *b* ⟹ *f a* ∈ *segment f a b* |
step: *x* ∈ *segment f a b* ⟹ *f x* ≠ *b* ⟹ *f x* ∈ *segment f a b*

lemma *segment-step-2D*:
assumes $x \in$ *segment f a* (*f b*) **shows** $x \in$ *segment f a b* $\vee$ $x = b$
using *assms* **by** *induct* (*auto intro*: *segment.intros*)

lemma *not-in-segment2D*:
assumes $x \in$ *segment f a b* **shows** $x \neq b$
using *assms* **by** *induct auto*

lemma *segment-altdef*:
assumes $b \in$ *orbit f a*
shows *segment f a b* = ($\lambda n.$ (*f* ^^ *n*) *a*) ` {*1*..<*funpow-dist1 f a b*} (**is** *?L* = *?R*)
proof (*intro set-eqI iffI*)
fix *x* **assume** $x \in$ *?L*
have *f a* $\neq b \Longrightarrow$ $b \in$ *orbit f* (*f a*)
using *assms* **by** (*simp add*: *orbit-step*)
then have $*$: *f a* $\neq$ *b* $\Longrightarrow$ *0* < *funpow-dist f* (*f a*) *b*
using *assms* **using** *gr0I funpow-dist-0-eq*[*OF* ‹- $\Longrightarrow$ $b \in$ *orbit f* (*f a*)›] **by** (*simp add*: *orbit.intros*)
from ‹$x \in$ *?L*› **show** $x \in$ *?R*
proof *induct*
case *base* **then show** *?case* **by** (*intro image-eqI*[**where** *x=1*]) (*auto simp*: $*$)
next
case *step* **then show** *?case* **using** *assms funpow-dist1-prop less-antisym*
by (*fastforce intro*!: *image-eqI*[**where** *x=Suc n* **for** *n*])
qed
next
fix *x* **assume** $x \in$ *?R*
then obtain *n* **where** (*f* ^^ *n*) *a* = *x 0* < *n n* < *funpow-dist1 f a b* **by** *auto*
then show $x \in$ *?L*
proof (*induct n arbitrary*: *x*)
case *0* **then show** *?case* **by** *simp*
next
case (*Suc n*)
have (*f* ^^ *Suc n*) *a* $\neq$ *b* **using** *Suc* **by** (*meson funpow-dist1-least*)
with *Suc* **show** *?case* **by** (*cases n* = *0*) (*auto intro*: *segment.intros*)
qed
qed

lemma *segmentD-orbit*:
assumes $x \in$ *segment f y z* **shows** $x \in$ *orbit f y*
using *assms* **by** *induct* (*auto intro*: *orbit.intros*)

lemma *segment1-empty*: *segment f x* (*f x*) = {}
by (*auto simp*: *segment-altdef orbit.base funpow-dist-0*)

lemma *segment-subset*:
assumes $y \in$ *segment f x z*

assumes $w \in$ *segment f x y*
shows $w \in$ *segment f x z*
using *assms* **by** (*induct arbitrary*: *w*) (*auto simp*: *segment1-empty intro*: *segment.intros dest*: *segment-step-2D elim*: *segment.cases*)

lemma *not-in-segment1*:
assumes $y \in$ *orbit f x* **shows** $x \notin$ *segment f x y*
proof
assume $x \in$ *segment f x y*
then obtain *n* **where** *n*: $0 < n$ $n <$ *funpow-dist1 f x y* $(f \hat{\ }\hat{\ } n)$ $x = x$
using *assms* **by** (*auto simp*: *segment-altdef Suc-le-eq*)
then have *neq-y*: $(f \hat{\ }\hat{\ } (\textit{funpow-dist1 f x y} - n))$ $x \neq y$ **by** (*simp add*: *funpow-dist1-least*)

have $(f \hat{\ }\hat{\ } (\textit{funpow-dist1 f x y} - n))$ $x = (f \hat{\ }\hat{\ } (\textit{funpow-dist1 f x y} - n))$ $((f \hat{\ }\hat{\ } n)$ $x)$
using *n* **by** (*simp add*: *funpow-add*)
also have $\ldots = (f \hat{\ }\hat{\ } \textit{funpow-dist1 f x y})$ x
using ‹$n <$ -› **by** (*simp add*: *funpow-add*)
(*metis assms funpow-0 funpow-neq-less-funpow-dist1 n(1) n(3) nat-neq-iff zero-less-Suc*)
also have $\ldots = y$ **using** *assms* **by** (*rule funpow-dist1-prop*)
finally show *False* **using** *neq-y* **by** *contradiction*
qed

lemma *not-in-segment2*: $y \notin$ *segment f x y*
using *not-in-segment2D* **by** *metis*

lemma *in-segmentE*:
assumes $y \in$ *segment f x z* $z \in$ *orbit f x*
obtains $(f \hat{\ }\hat{\ } \textit{funpow-dist1 f x y})$ $x = y$ *funpow-dist1 f x y* $<$ *funpow-dist1 f x z*
proof
from *assms* **show** $(f \hat{\ }\hat{\ } \textit{funpow-dist1 f x y})$ $x = y$
by (*intro segmentD-orbit funpow-dist1-prop*)
moreover
obtain *n* **where** $(f \hat{\ }\hat{\ } n)$ $x = y$ $0 < n$ $n <$ *funpow-dist1 f x z*
using *assms* **by** (*auto simp*: *segment-altdef*)
moreover then have *funpow-dist1 f x y* $\leq n$ **by** (*meson funpow-dist1-least not-less*)
ultimately show *funpow-dist1 f x y* $<$ *funpow-dist1 f x z* **by** *linarith*
qed

lemma *cyclic-split-segment*:
assumes *S*: *cyclic-on f S* $a \in S$ $b \in S$ **and** $a \neq b$
shows $S = \{a,b\} \cup$ *segment f a b* $\cup$ *segment f b a* (**is** *?L* = *?R*)
proof (*intro set-eqI iffI*)

fix c **assume** $c \in ?L$
with S **have** $c \in \mathit{orbit}\ f\ a$ **unfolding** *cyclic-on-alldef* **by** *auto*
then show $c \in ?R$ **by** *induct* (*auto intro*: *segment.intros*)
next
fix c **assume** $c \in ?R$
moreover have $\mathit{segment}\ f\ a\ b \subseteq \mathit{orbit}\ f\ a\ \mathit{segment}\ f\ b\ a \subseteq \mathit{orbit}\ f\ b$
by (*auto dest*: *segmentD-orbit*)
ultimately show $c \in ?L$ **using** S **by** (*auto simp*: *cyclic-on-alldef*)
qed

lemma *segment-split*:
assumes *y-in-seg*: $y \in \mathit{segment}\ f\ x\ z$
shows $\mathit{segment}\ f\ x\ z = \mathit{segment}\ f\ x\ y \cup \{y\} \cup \mathit{segment}\ f\ y\ z$ (**is** $?L = ?R$)
proof (*intro set-eqI iffI*)
fix w **assume** $w \in ?L$ **then show** $w \in ?R$ **by** *induct* (*auto intro*: *segment.intros*)
next
fix w **assume** $w \in ?R$
moreover
{ assume $w \in \mathit{segment}\ f\ x\ y$ **then have** $w \in \mathit{segment}\ f\ x\ z$
using *segment-subset*[*OF y-in-seg*] **by** *auto* **}**
moreover
{ assume $w \in \mathit{segment}\ f\ y\ z$ **then have** $w \in \mathit{segment}\ f\ x\ z$
using *y-in-seg* **by** *induct* (*auto intro*: *segment.intros*) **}**
ultimately
show $w \in ?L$ **using** *y-in-seg* **by** (*auto intro*: *segment.intros*)
qed

lemma *in-segmentD-inv*:
assumes $x \in \mathit{segment}\ f\ a\ b\ x \neq f\ a$
assumes $\mathit{inj}\ f$
shows $\mathit{inv}\ f\ x \in \mathit{segment}\ f\ a\ b$
using *assms* **by** (*auto elim*: *segment.cases*)

lemma *in-orbit-invI*:
assumes $b \in \mathit{orbit}\ f\ a$
assumes $\mathit{inj}\ f$
shows $a \in \mathit{orbit}\ (\mathit{inv}\ f)\ b$
using *assms*(*1*)
apply *induct*
apply (*simp add*: *assms*(*2*) *orbit-eqI*(*1*))
by (*metis assms*(*2*) *inv-f-f orbit.base orbit-trans*)

lemma *segment-step-2*:
assumes *A*: $x \in \mathit{segment}\ f\ a\ b\ b \neq a$ **and** $\mathit{inj}\ f$
shows $x \in \mathit{segment}\ f\ a\ (f\ b)$
using *A* **by** *induct* (*auto intro*: *segment.intros dest*: *not-in-segment2D injD*[*OF* ‹*inj f*›])

lemma *inv-end-in-segment*:
assumes $b \in orbit\ f\ a\ f\ a \neq b\ bij\ f$
shows $inv\ f\ b \in segment\ f\ a\ b$
using *assms*(*1*,*2*)
proof *induct*
case *base* **then show** *?case* **by** *simp*
next
case (*step x*)
moreover
from ‹*bij f*› **have** *inj f* **by** (*rule bij-is-inj*)
moreover
then have $x \neq f\ x \Longrightarrow f\ a = x \Longrightarrow x \in segment\ f\ a\ (f\ x)$ **by** (*meson segment.simps*)
moreover
have $x \neq f\ x$
using *step* ‹*inj f*› **by** (*metis in-orbit-invI inv-f-eq not-in-segment1 segment.base*)
then have $inv\ f\ x \in segment\ f\ a\ (f\ x) \Longrightarrow x \in segment\ f\ a\ (f\ x)$
using ‹*bij f*› ‹*inj f*› **by** (*auto dest*: *segment.step simp*: *surj-f-inv-f bij-is-surj*)
then have $inv\ f\ x \in segment\ f\ a\ x \Longrightarrow x \in segment\ f\ a\ (f\ x)$
using ‹$f\ a \neq f\ x$› ‹*inj f*› **by** (*auto dest*: *segment-step-2 injD*)
ultimately show *?case* **by** (*cases f a = x*) *simp-all*
qed

lemma *segment-overlapping*:
assumes $x \in orbit\ f\ a\ x \in orbit\ f\ b\ bij\ f$
shows $segment\ f\ a\ x \subseteq segment\ f\ b\ x \vee segment\ f\ b\ x \subseteq segment\ f\ a\ x$
using *assms*(*1*,*2*)
proof *induction*
case *base* **then show** *?case* **by** (*simp add*: *segment1-empty*)
next
case (*step x*)
from ‹*bij f*› **have** *inj f* **by** (*simp add*: *bij-is-inj*)
have ∗: $\bigwedge f\ x\ y.\ y \in segment\ f\ x\ (f\ x) \Longrightarrow False$ **by** (*simp add*: *segment1-empty*)
{ fix *y z*
assume *A*: $y \in segment\ f\ b\ (f\ x)\ y \notin segment\ f\ a\ (f\ x)\ z \in segment\ f\ a\ (f\ x)$
from ‹$x \in orbit\ f\ a$› ‹$f\ x \in orbit\ f\ b$› ‹$y \in segment\ f\ b\ (f\ x)$›
have $x \in orbit\ f\ b$
by (*metis* ∗ *inv-end-in-segment*[*OF - -* ‹*bij f*›] *inv-f-eq*[*OF* ‹*inj f*›] *segmentD-orbit*)
moreover
with ‹$x \in orbit\ f\ a$› *step.IH*
have $segment\ f\ a\ (f\ x) \subseteq segment\ f\ b\ (f\ x) \vee segment\ f\ b\ (f\ x) \subseteq segment\ f\ a\ (f\ x)$
apply *auto*
apply (*metis* ∗ *inv-end-in-segment*[*OF - -* ‹*bij f*›] *inv-f-eq*[*OF* ‹*inj f*›] *segment-step-2D segment-subset step.prems subsetCE*)
by (*metis* (*no-types*, *lifting*) ‹*inj f*› ∗ *inv-end-in-segment*[*OF - -* ‹*bij f*›] *inv-f-eq orbit-eqI*(*2*) *segment-step-2D segment-subset subsetCE*)
ultimately

```
    have segment f a (f x) ⊆ segment f b (f x) using A by auto
  } note C = this
  then show ?case by auto
qed

lemma segment-disj:
  assumes a ≠ b bij f
  shows segment f a b ∩ segment f b a = {}
proof (rule ccontr)
  assume ¬?thesis
  then obtain x where x: x ∈ segment f a b x ∈ segment f b a by blast
  then have segment f a b = segment f a x ∪ {x} ∪ segment f x b
      segment f b a = segment f b x ∪ {x} ∪ segment f x a
    by (auto dest: segment-split)
  then have o: x ∈ orbit f a x ∈ orbit f b by (auto dest: segmentD-orbit)

  note * = segment-overlapping[OF o ‹bij f›]
  have inj f using ‹bij f› by (simp add: bij-is-inj)

  have segment f a x = segment f b x
  proof (intro set-eqI iffI)
    fix y assume A: y ∈ segment f b x
    then have y ∈ segment f a x ∨ f a ∈ segment f b a
      using * x(2) by (auto intro: segment.base segment-subset)
    then show y ∈ segment f a x
      using ‹inj f› A by (metis (no-types) not-in-segment2 segment-step-2)
  next
    fix y assume A: y ∈ segment f a x
    then have y ∈ segment f b x ∨ f b ∈ segment f a b
      using * x(1) by (auto intro: segment.base segment-subset)
    then show y ∈ segment f b x
      using ‹inj f› A by (metis (no-types) not-in-segment2 segment-step-2)
  qed
  moreover
  have segment f a x ≠ segment f b x
    by (metis assms bij-is-inj not-in-segment2 segment.base segment-step-2 segment-subset x(1))
  ultimately show False by contradiction
qed

lemma segment-x-x-eq:
  assumes permutation f
  shows segment f x x = orbit f x − {x} (is ?L = ?R)
proof (intro set-eqI iffI)
  fix y assume y ∈ ?L then show y ∈ ?R by (auto dest: segmentD-orbit simp: not-in-segment2)
next
  fix y assume y ∈ ?R
  then have y ∈ orbit f x y ≠ x by auto
```

then show $y \in ?L$ **by** *induct* (*auto intro*: *segment.intros*)
qed

14 Lists of Powers

definition *iterate* :: *nat* ⇒ *nat* ⇒ ('*a* ⇒ '*a*) ⇒ '*a* ⇒ '*a list* **where**
iterate m n f x = *map* (λ*n*. (*f*^^*n*) *x*) [*m*..<*n*]

lemma *set-iterate*:
set (*iterate m n f x*) = (λ*k*. (*f* ^^ *k*) *x*) ' {*m*..<*n*}
by (*auto simp*: *iterate-def*)

lemma *iterate-empty*[*simp*]: *iterate n m f x* = [] ⟷ *m* ≤ *n*
by (*auto simp*: *iterate-def*)

lemma *iterate-length*[*simp*]:
length (*iterate m n f x*) = *n* − *m*
by (*auto simp*: *iterate-def*)

lemma *iterate-nth*[*simp*]:
assumes *k* < *n* − *m* **shows** *iterate m n f x* ! *k* = (*f*^^(*m*+*k*)) *x*
using *assms*
by (*induct k arbitrary*: *m*) (*auto simp*: *iterate-def*)

lemma *iterate-applied*:
iterate n m f (*f x*) = *iterate* (*Suc n*) (*Suc m*) *f x*
by (*induct m arbitrary*: *n*) (*auto simp*: *iterate-def funpow-swap1*)

end
theory *Subdivision*
imports
Arc-Walk
Digraph-Component
Pair-Digraph
Bidirected-Digraph
Auxiliary
begin

15 Subdivision on Digraphs

definition
subdivision-step :: ('*a*, '*b*) *pre-digraph* ⇒ ('*b* ⇒ '*b*) ⇒ ('*a*, '*b*) *pre-digraph* ⇒ ('*b* ⇒ '*b*) ⇒ '*a* × '*a* × '*a* ⇒ '*b* × '*b* × '*b* ⇒ *bool*
where
subdivision-step G rev-G H rev-H ≡ λ(*u*, *v*, *w*) (*uv*, *uw*, *vw*).
bidirected-digraph G rev-G
∧ *bidirected-digraph H rev-H*
∧ *perm-restrict rev-H* (*arcs G*) = *perm-restrict rev-G* (*arcs H*)

$\wedge$ *compatible G H*

$\wedge$ *verts H* = *verts G* $\cup$ $\{w\}$
$\wedge$ $w \notin$ *verts G*

$\wedge$ *arcs H* = $\{uw,$ *rev-H uw*, *vw*, *rev-H vw*$\}$ $\cup$ *arcs G* $-$ $\{uv,$ *rev-G uv*$\}$
$\wedge$ $uv \in$ *arcs G*
$\wedge$ *distinct* [*uw*, *rev-H uw*, *vw*, *rev-H vw*]
$\wedge$ *arc-to-ends G uv* = (*u*,*v*)
$\wedge$ *arc-to-ends H uw* = (*u*,*w*)
$\wedge$ *arc-to-ends H vw* = (*v*,*w*)

inductive *subdivision* :: ('*a*,'*b*) *pre-digraph* $\times$ ('*b* $\Rightarrow$ '*b*) $\Rightarrow$ ('*a*,'*b*) *pre-digraph* $\times$ ('*b* $\Rightarrow$ '*b*) $\Rightarrow$ *bool*
for *biG* **where**
base: *bidirected-digraph* (*fst biG*) (*snd biG*) $\Longrightarrow$ *subdivision biG biG*
| *divide*: $[\![$*subdivision biG biI*; *subdivision-step* (*fst biI*) (*snd biI*) (*fst biH*) (*snd biH*) (*u*,*v*,*w*) (*uv*,*uw*,*vw*)$]\!]$ $\Longrightarrow$ *subdivision biG biH*

lemma *subdivision-induct*[*case-names base divide*, *induct pred*: *subdivision*]:
assumes *subdivision* (*G*, *rev-G*) (*H*, *rev-H*)
and *bidirected-digraph G rev-G* $\Longrightarrow$ *P G rev-G*
and $\bigwedge$*I rev-I H rev-H u v w uv uw vw*.
subdivision (*G*, *rev-G*) (*I*, *rev-I*) $\Longrightarrow$ *P I rev-I* $\Longrightarrow$ *subdivision-step I rev-I H rev-H* (*u*, *v*, *w*) (*uv*, *uw*, *vw*) $\Longrightarrow$ *P H rev-H*
shows *P H rev-H*
using *assms*(*1*) **by** (*induct biH*$\equiv$(*H*, *rev-H*) *arbitrary*: *H rev-H*) (*auto intro*: *assms*(*2*,*3*))

lemma *subdivision-base*:
bidirected-digraph G rev-G $\Longrightarrow$ *subdivision* (*G*, *rev-G*) (*G*, *rev-G*)
by (*rule subdivision.base*) *simp*

lemma *subdivision-step-rev*:
assumes *subdivision-step G rev-G H rev-H* (*u*, *v*, *w*) (*uv*, *uw*, *vw*) *subdivision* (*H*, *rev-H*) (*I*, *rev-I*)
shows *subdivision* (*G*, *rev-G*) (*I*, *rev-I*)
proof $-$
have *bidirected-digraph* (*fst* (*G*, *rev-G*)) (*snd* (*G*, *rev-G*)) **using** *assms* **by** (*auto simp*: *subdivision-step-def*)
with *assms*(*2*,*1*) **show** *?thesis*
using *assms*(*2*,*1*) **by** *induct* (*auto intro*: *subdivision.intros dest*: *subdivision-base*)
qed

lemma *subdivision-trans*:
assumes *subdivision* (*G*, *rev-G*) (*H*, *rev-H*) *subdivision* (*H*, *rev-H*) (*I*, *rev-I*)
shows *subdivision* (*G*, *rev-G*) (*I*, *rev-I*)
using *assms* **by** *induction* (*auto intro*: *subdivision-step-rev*)

locale *subdiv-step* =
  fixes *G rev-G H rev-H u v w uv uw vw*
  assumes *subdiv-step*: *subdivision-step G rev-G H rev-H* (*u*, *v*, *w*) (*uv*, *uw*, *vw*)

sublocale *subdiv-step* ⊆ *G*: *bidirected-digraph G rev-G*
  using *subdiv-step* **unfolding** *subdivision-step-def* **by** *simp*
sublocale *subdiv-step* ⊆ *H*: *bidirected-digraph H rev-H*
  using *subdiv-step* **unfolding** *subdivision-step-def* **by** *simp*

context *subdiv-step* **begin**

  abbreviation (*input*) *vu* ≡ *rev-G uv*
  abbreviation (*input*) *wu* ≡ *rev-H uw*
  abbreviation (*input*) *wv* ≡ *rev-H vw*

  lemma *subdiv-compat*: *compatible G H*
    using *subdiv-step* **by** (*simp add*: *subdivision-step-def*)

  lemma *arc-to-ends-eq*: *arc-to-ends H* = *arc-to-ends G*
    using *subdiv-compat* **by** (*simp add*: *compatible-def arc-to-ends-def fun-eq-iff*)

  lemma *head-eq*: *head H* = *head G*
    using *subdiv-compat* **by** (*simp add*: *compatible-def fun-eq-iff*)

  lemma *tail-eq*: *tail H* = *tail G*
    using *subdiv-compat* **by** (*simp add*: *compatible-def fun-eq-iff*)

  lemma *verts-H*: *verts H* = *verts G* ∪ {*w*}
    using *subdiv-step* **by** (*simp add*: *subdivision-step-def*)

  lemma *verts-G*: *verts G* = *verts H* − {*w*}
    using *subdiv-step* **by** (*auto simp*: *subdivision-step-def*)

  lemma *arcs-H*: *arcs H* = {*uw*, *wu*, *vw*, *wv*} ∪ *arcs G* − {*uv*, *vu*}
    using *subdiv-step* **by** (*simp add*: *subdivision-step-def*)

  lemma *not-in-verts-G*: *w* ∉ *verts G*
    using *subdiv-step* **by** (*simp add*: *subdivision-step-def*)

  lemma *in-arcs-G*: {*uv*, *vu*} ⊆ *arcs G*
    using *subdiv-step* **by** (*simp add*: *subdivision-step-def*)

  lemma *not-in-arcs-H*: {*uv*,*vu*} ∩ *arcs H* = {}
    using *arcs-H* **by** *auto*

  lemma *subdiv-ate*:
      arc-to-ends G uv = (*u*,*v*)

arc-to-ends H uv = (u,v)
arc-to-ends H uw = (u,w)
arc-to-ends H vw = (v,w)
using *subdiv-step subdiv-compat* **by** (*auto simp*: *subdivision-step-def arc-to-ends-def compatible-def*)

lemma *subdiv-ends*[*simp*]:
tail G uv = u head G uv = v tail H uv = u head H uv = v
tail H uw = u head H uw = w tail H vw = v head H vw = w
using *subdiv-ate* **by** (*auto simp*: *arc-to-ends-def*)

lemma *subdiv-ends-G-rev*[*simp*]:
tail G (vu) = v head G (vu) = u tail H (vu) = v head H (vu) = u
using *in-arcs-G* **by** (*auto simp*: *tail-eq head-eq*)

lemma *subdiv-distinct-verts0*: *u* ≠ *w v* ≠ *w*
using *in-arcs-G not-in-verts-G* **using** *subdiv-ate* **by** (*auto simp*: *arc-to-ends-def dest*: *G.wellformed*)

lemma *in-arcs-H*: {*uw*, *wu*, *vw*, *wv*} ⊆ *arcs H*
proof −
{ assume *uv = uw*
then have *arc-to-ends H uv = arc-to-ends H uw* **by** *simp*
then have *v = w* **by** (*simp add*: *arc-to-ends-def*)
} moreover
{ assume *uv = vw*
then have *arc-to-ends H uv = arc-to-ends H vw* **by** *simp*
then have *v = w* **by** (*simp add*: *arc-to-ends-def*)
} moreover
{ assume *vu = uw*
then have *arc-to-ends H (vu) = arc-to-ends H uw* **by** *simp*
then have *u = w* **by** (*simp add*: *arc-to-ends-def*)
} moreover
{ assume *vu = vw*
then have *arc-to-ends H (vu) = arc-to-ends H vw* **by** *simp*
then have *u = w* **by** (*simp add*: *arc-to-ends-def*)
} ultimately
have {*uw*,*vw*} ⊆ *arcs H* **unfolding** *arcs-H* **using** *subdiv-distinct-verts0* **by** *auto*
then show *?thesis* **by** *auto*
qed

lemma *subdiv-ends-H-rev*[*simp*]:
tail H (wu) = w tail H (wv) = w
head H (wu) = u head H (wv) = v
using *in-arcs-H subdiv-ate* **by** *simp-all*

lemma *in-verts-G*: {*u*,*v*} ⊆ *verts G*
using *in-arcs-G* **by** (*auto dest*: *G.wellformed*)

lemma *not-in-arcs-G*: $\{uw, wu, vw, wv\} \cap arcs\ G = \{\}$
proof −
note $X = G.wellformed[simplified\ tail\text{-}eq[symmetric]\ head\text{-}eq[symmetric]]$
show *?thesis* **using** *not-in-verts-G in-arcs-H* **by** (*auto dest*: X)
qed

lemma *subdiv-distinct-arcs*: *distinct* $[uv, vu, uw, wu, vw, wv]$
proof −
have *distinct* $[uw, wu, vw, wv]$
using *subdiv-step* **by** (*simp add*: *subdivision-step-def*)
moreover
have *distinct* $[uv, vu]$ **using** *in-arcs-G G.arev-dom* **by** *auto*
moreover
have $\{uv, vu\} \cap \{uw, wu, vw, wv\} = \{\}$
using *arcs-H in-arcs-H* **by** *auto*
ultimately show *?thesis* **by** *auto*
qed

lemma *arcs-G*: $arcs\ G = arcs\ H \cup \{uv, vu\} - \{uw, wu, vw, wv\}$
using *in-arcs-G not-in-arcs-G* **unfolding** *arcs-H* **by** *auto*

lemma *subdiv-ate-H-rev*:
arc-to-ends H $(wu) = (w,u)$
arc-to-ends H $(wv) = (w,v)$
using *in-arcs-H subdiv-ate* **by** *simp-all*

lemma *adj-with-w*: $u \rightarrow_H w\ w \rightarrow_H u\ v \rightarrow_H w\ w \rightarrow_H v$
using *in-arcs-H subdiv-ate* **by** (*auto intro*: *H.dominatesI*[*rotated*])

lemma *w-reach*: $u \rightarrow^*{}_H w\ w \rightarrow^*{}_H u\ v \rightarrow^*{}_H w\ w \rightarrow^*{}_H v$
using *adj-with-w* **by** *auto*

lemma *G-reach*: $v \rightarrow^*{}_G u\ u \rightarrow^*{}_G v$
using *subdiv-ate in-arcs-G* **by** (*simp add*: *G.dominatesI G.symmetric-reachable′*)+

lemma *out-arcs-w*: *out-arcs H w* $= \{wu, wv\}$
using *subdiv-distinct-verts0 in-arcs-H*
by (*auto simp*: *arcs-H*) (*auto simp*: *tail-eq verts-G dest*: *G.tail-in-verts*)

lemma *out-degree-w*: *out-degree H w* $= 2$
using *subdiv-distinct-arcs* **by** (*auto simp*: *out-degree-def out-arcs-w card-insert-if*)

end

lemma *subdivision-compatible*:
assumes *subdivision* (G, *rev-G*) (H, *rev-H*) **shows** *compatible G H*
using *assms* **by** *induct* (*auto simp*: *compatible-def subdivision-step-def*)

lemma *subdivision-bidir*:
assumes *subdivision* (*G*, *rev-G*) (*H*, *rev-H*)
shows *bidirected-digraph H rev-H*
using *assms* **by** *induct* (*auto simp*: *subdivision-step-def*)

lemma *subdivision-choose-rev*:
assumes *subdivision* (*G*, *rev-G*) (*H*, *rev-H*) *bidirected-digraph H rev-H'*
shows ∃ *rev-G'*. *subdivision* (*G*, *rev-G'*) (*H*, *rev-H'*)
using *assms*
proof (*induction arbitrary*: *rev-H'*)
case *base*
then show *?case* **by** (*auto dest*: *subdivision-base*)
next
case (*divide I rev-I H rev-H u v w uv uw vw*)

interpret *subdiv-step I rev-I H rev-H u v w uv uw vw* **using** *divide* **by** *unfold-locales*
interpret *H'*: *bidirected-digraph H rev-H'* **by** *fact*

define *rev-I'* **where** *rev-I' x* =
(*if x* = *uv then rev-I uv else if x* = *rev-I uv then uv else if x* ∈ *arcs I then rev-H' x else x*)
for *x*

have *rev-H-injD*: ⋀*x y z*. *rev-H' x* = *z* ⟹ *rev-H' y* = *z* ⟹ *x* ≠ *y* ⟹ *False*
by (*metis H'.arev-eq-iff*)

have *rev-H'-simps*: *rev-H' uw* = *rev-H uw* ∧ *rev-H' vw* = *rev-H vw*
∨ *rev-H' uw* = *rev-H vw* ∧ *rev-H' vw* = *rev-H uw*
proof −
have *arc-to-ends H* (*rev-H' uw*) = (*w*,*u*) *arc-to-ends H* (*rev-H' vw*) = (*w*,*v*)
using *in-arcs-H* **by** (*auto simp*: *subdiv-ate*)
moreover
have ⋀*x*. *x* ∈ *arcs H* ⟹ *tail H x* = *w* ⟹ *x* ∈ {*rev-H uw*, *rev-H vw*}
using *subdiv-distinct-verts0 not-in-verts-G* **by** (*auto simp*: *arcs-H*) (*simp add*: *tail-eq*)
ultimately
have *rev-H' uw* ∈ {*rev-H uw*, *rev-H vw*} *rev-H' vw* ∈ {*rev-H uw*, *rev-H vw*}
using *in-arcs-H* **by** *auto*
then show *?thesis* **using** *in-arcs-H* **by** (*auto dest*: *rev-H-injD*)
qed

have *rev-H-uv*: *rev-H' uv* = *uv rev-H'* (*rev-I uv*) = *rev-I uv*
using *not-in-arcs-H* **by** (*auto simp*: *H'.arev-eq*)

have *bd-I'*: *bidirected-digraph I rev-I'*
proof
fix *a*
have ⋀*a*. *a* ≠ *uv* ⟹ *a* ≠ *rev-I uv* ⟹ *a* ∈ *arcs I* ⟹ *a* ∈ *arcs H*

by (*auto simp*: *arcs-H*)
then show (*a* ∈ *arcs I*) = (*rev-I′ a* ≠ *a*)
using *in-arcs-G* **by** (*auto simp*: *rev-I′-def dest*: *G.arev-neq H′.arev-neq*)
next
fix *a*
have ∗: ⋀*a*. *rev-H′ a* = *rev-I uv* ⟷ *a* = *rev-I uv*
by (*metis H′.arev-arev H′.arev-dom insert-disjoint*(*1*) *not-in-arcs-H*)
have ∗∗: ⋀*a*. *uv* = *rev-H′ a* ⟷ *a* = *uv* **using** *H′.arev-eq not-in-arcs-H* **by** *force*
have ∗∗∗: ⋀*a*. *a* ∈ *arcs I* ⟹ *rev-H′ a* ∈ *arcs I*
using *rev-H′-simps* **by** (*case-tac a* ∈ {*uv*,*vu*}) (*fastforce simp*: *rev-H-uv*, *auto simp*: *arcs-G dest*: *rev-H-injD*)
show *rev-I′* (*rev-I′ a*) = *a*
by (*auto simp*: *rev-I′-def H′.arev-eq rev-H-uv* ∗ ∗∗ ∗∗∗)
next
fix *a* **assume** *a* ∈ *arcs I*
then show *tail I* (*rev-I′ a*) = *head I a*
using *in-arcs-G* **by** (*auto simp*: *rev-I′-def tail-eq*[*symmetric*] *head-eq*[*symmetric*] *arcs-H*)
qed
moreover
have ⋀*x*. *rev-H′ x* = *uv* ⟷ *x* = *uv* ⋀*x*. *rev-H′ x* = *rev-I uv* ⟷ *x* = *rev-I uv*
using *not-in-arcs-H* **by** (*auto dest*: *H′.arev-eq*) (*metis H′.arev-arev H′.arev-eq*)
then have *perm-restrict rev-H′* (*arcs I*) = *perm-restrict rev-I′* (*arcs H*)
using *not-in-arcs-H* **by** (*auto simp*: *rev-I′-def perm-restrict-def H′.arev-eq*)
ultimately
have *sds-I′H′*: *subdivision-step I rev-I′ H rev-H′* (*u*, *v*, *w*) (*uv*, *uw*, *vw*)
using *divide*(*2*,*4*) *rev-H′-simps* **unfolding** *subdivision-step-def*
by (*fastforce simp*: *rev-I′-def*)
then have *subdivision* (*I*, *rev-I′*) (*H*, *rev-H′*) ∃ *rev-G′*. *subdivision* (*G*, *rev-G′*) (*I*, *rev-I′*)
using *bd-I′ divide* **by** (*auto intro*: *subdivision.intros dest*: *subdivision-base*)
then show *?case* **by** (*blast intro*: *subdivision-trans*)
qed

lemma *subdivision-verts-subset*:
assumes *subdivision* (*G*,*rev-G*) (*H*,*rev-H*) *x* ∈ *verts G*
shows *x* ∈ *verts H*
using *assms* **by** *induct* (*auto simp*: *subdiv-step.verts-H subdiv-step-def*)

15.1 Subdivision on Pair Digraphs

In this section, we introduce specialized rules for pair digraphs.

abbreviation *subdivision-pair G H* ≡ *subdivision* (*with-proj G*, *swap-in* (*parcs G*)) (*with-proj H*, *swap-in* (*parcs H*))

lemma *arc-to-ends-with-proj*[*simp*]: *arc-to-ends* (*with-proj G*) = *id*
by (*auto simp*: *arc-to-ends-def*)

context
begin

We use the `inductive` command to define an inductive definition pair graphs. This is proven to be equivalent to *subdivision*. This allows us to transfer the rules proven by `inductive` to *subdivision*. To spare the user confusion, we hide this new constant.

private inductive *pair-sd* :: *'a pair-pre-digraph* ⇒ *'a pair-pre-digraph* ⇒ *bool*
for *G* **where**
base: *pair-bidirected-digraph G* ⟹ *pair-sd G G*
| *divide*: ⋀*e w H*. ⟦*e* ∈ *parcs H*; *w* ∉ *pverts H*; *pair-sd G H*⟧
⟹ *pair-sd G* (*subdivide H e w*)

private lemma *bidirected-digraphI-pair-sd*:
assumes *pair-sd G H* **shows** *pair-bidirected-digraph H*
using *assms*
proof *induct*
case *base*
then show *?case* **by** *auto*
next
case (*divide e w H*)
interpret *H*: *pair-bidirected-digraph H* **by** *fact*
from *divide* **show** *?case* **by** (*intro H.pair-bidirected-digraph-subdivide*)
qed

private lemma *subdivision-with-projI*:
assumes *pair-sd G H*
shows *subdivision-pair G H*
using *assms*
proof *induct*
case *base*
then show *?case* **by** (*blast intro*: *pair-bidirected-digraph.bidirected-digraph subdivision-base*)
next
case (*divide e w H*)

obtain *u v* **where** *e* = (*u*,*v*) **by** (*cases e*)

interpret *H*: *pair-bidirected-digraph H*
using *divide*(*3*) **by** (*rule bidirected-digraphI-pair-sd*)
interpret *I*: *pair-bidirected-digraph subdivide H e w*
using *divide*(*1*,*2*) **by** (*rule H.pair-bidirected-digraph-subdivide*)

have *uvw*: *u* ≠ *v u* ≠ *w v* ≠ *w*
using *divide* **by** (*auto simp*: ‹*e* = -› *dest*: *H.adj-not-same H.wellformed*)

have *subdivision* (*with-proj G*, *swap-in* (*parcs G*)) (*H*, *swap-in* (*parcs H*))
using *divide* **by** *auto*
moreover

have $*$: *perm-restrict* (*swap-in* (*parcs* (*subdivide H e w*))) (*parcs H*) = *perm-restrict* (*swap-in* (*parcs H*)) (*parcs* (*subdivide H e w*))
by (*auto simp*: *perm-restrict-def fun-eq-iff swap-in-def*)
have *subdivision-step* (*with-proj H*) (*swap-in* (*arcs H*)) (*with-proj* (*subdivide H e w*)) (*swap-in* (*arcs* (*subdivide H e w*)))
(*u*, *v*, *w*) (*e*, (*u*,*w*), (*v*,*w*))
unfolding *subdivision-step-def*
unfolding *prod.simps with-proj-simps*
using *divide uvw*
by (*intro conjI H.bidirected-digraph I.bidirected-digraph* $*$)
(*auto simp add*: *swap-in-def* ‹*e* = -› *compatibleI-with-proj*)
ultimately
show *?case* **by** (*auto intro*: *subdivision.divide*)
qed

private lemma *subdivision-with-projD*:
assumes *subdivision-pair G H*
shows *pair-sd G H*
using *assms*
proof (*induct with-proj H swap-in* (*parcs H*) *arbitrary*: *H rule*: *subdivision-induct*)
case *base*
interpret *bidirected-digraph with-proj G swap-in* (*parcs G*) **by** *fact*
from *base* **have** *G* = *H* **by** (*simp add*: *with-proj-def*)
with *base* **show** *?case*
by (*auto intro*: *pair-sd.base pair-bidirected-digraphI-bidirected-digraph*)
next
case (*divide I rev-I u v w uv uw vw*)
define *I′* **where** *I′* = ⦇ *pverts* = *verts I*, *parcs* = *arcs I* ⦈
have *compatible G I* **using** ‹*subdivision* (*with-proj G*, -) (*I*, -)›
by (*rule subdivision-compatible*)
then have *tail I* = *fst head I* = *snd* **by** (*auto simp*: *compatible-def*)
then have *I*: *I* = *I′* **by** (*auto simp*: *I′-def*)
moreover
from *I* **have** *rev-I* = *swap-in* (*parcs I′*)
using ‹*subdivision-step* - - - - - -›
by (*simp add*: *subdivision-step-def bidirected-digraph-rev-conv-pair*)
ultimately
have *pd-sd*: *pair-sd G I′* **by** (*auto intro*: *divide.hyps*)

interpret *sd*: *subdiv-step I′ swap-in* (*parcs I′*) *H swap-in* (*parcs H*) *u v w uv uw vw*
using ‹*subdivision-step* - - - - - -› **unfolding** ‹*I* = -› ‹*rev-I* = -› **by** *unfold-locales*

have *ends*: *uv* = (*u*,*v*) *uw* = (*u*,*w*) *vw* = (*v*,*w*)
using *sd.subdiv-ate* **by** *simp-all*
then have *si-ends*: *swap-in* (*parcs H*) (*u*,*w*) = (*w*,*u*) *swap-in* (*parcs H*) (*v*,*w*) = (*w*,*v*)
swap-in (*parcs I′*) (*u*,*v*) = (*v*,*u*)

using *sd.subdiv-ends-H-rev sd.subdiv-ends-G-rev* **by** (*auto simp*: *swap-in-def*)

have *parcs H* = *parcs I'* − {(*u*, *v*), (*v*, *u*)} ∪ {(*u*, *w*), (*w*, *u*), (*w*, *v*), (*v*, *w*)}
using *sd.in-arcs-G sd.not-in-arcs-G sd.arcs-H* **by** (*auto simp*: *si-ends ends*)
then have *H* = *subdivide I' uv w* **using** *sd.verts-H* **by** (*simp add*: *ends subdivide.simps*)
then show *?case*
using *sd.in-arcs-G sd.not-in-verts-G* **by** (*auto intro*: *pd-sd pair-sd.divide*)
qed

private lemma *subdivision-pair-conv*:
pair-sd G H = *subdivision-pair G H*
by (*metis subdivision-with-projD subdivision-with-projI*)

lemmas *subdivision-pair-induct* = *pair-sd.induct*[
unfolded subdivision-pair-conv, *case-names base divide*, *induct pred*: *pair-sd*]

lemmas *subdivision-pair-base* = *pair-sd.base*[*unfolded subdivision-pair-conv*]
lemmas *subdivision-pair-divide* = *pair-sd.divide*[*unfolded subdivision-pair-conv*]

lemmas *subdivision-pair-intros* = *pair-sd.intros*[*unfolded subdivision-pair-conv*]
lemmas *subdivision-pair-cases* = *pair-sd.cases*[*unfolded subdivision-pair-conv*]

lemmas *subdivision-pair-simps* = *pair-sd.simps*[*unfolded subdivision-pair-conv*]

lemmas *bidirected-digraphI-subdivision* = *bidirected-digraphI-pair-sd*[*unfolded subdivision-pair-conv*]

end

lemma (**in** *pair-graph*) *pair-graph-subdivision*:
assumes *subdivision-pair G H*
shows *pair-graph H*
using *assms*
by (*induct rule*: *subdivision-pair-induct*) (*blast intro*: *pair-graph.pair-graph-subdivide divide*)+

end

theory *Euler* **imports**
Arc-Walk
Digraph-Component
Digraph-Isomorphism
begin

16 Euler Trails in Digraphs

In this section we prove the well-known theorem characterizing the existence of an Euler Trail in an directed graph

16.1 Trails and Euler Trails

definition (**in** *pre-digraph*) *euler-trail* :: *'a* ⇒ *'b awalk* ⇒ *'a* ⇒ *bool* **where**
euler-trail u p v ≡ *trail u p v* ∧ *set p* = *arcs G* ∧ *set* (*awalk-verts u p*) = *verts G*

context *wf-digraph* **begin**

lemma *finite-distinct*:
assumes *finite A* **shows** *finite* {*p*. *distinct p* ∧ *set p* ⊆ *A*}
proof −
have {*p*. *distinct p* ∧ *set p* ⊆ *A*} ⊆ {*p*. *set p* ⊆ *A* ∧ *length p* ≤ *card A*}
using *assms* **by** (*auto simp*: *distinct-card*[*symmetric*] *intro*: *card-mono*)
also have *finite* ...
using *assms* **by** (*simp add*: *finite-lists-length-le*)
finally (*finite-subset*) **show** *?thesis* **.**
qed

lemma (**in** *fin-digraph*) *trails-finite*: *finite* {*p*. ∃ *u v*. *trail u p v*}
proof −
have {*p*. ∃ *u v*. *trail u p v*} ⊆ {*p*. *distinct p* ∧ *set p* ⊆ *arcs G*}
by (*auto simp*: *trail-def*)
with *finite-arcs finite-distinct* **show** *?thesis* **by** (*blast intro*: *finite-subset*)
qed

lemma *rotate-awalkE*:
assumes *awalk u p u w* ∈ *set* (*awalk-verts u p*)
obtains *q r* **where** *p* = *q* @ *r awalk w* (*r* @ *q*) *w set* (*awalk-verts w* (*r* @ *q*)) = *set* (*awalk-verts u p*)
proof −
from *assms* **obtain** *q r* **where** *A*: *p* = *q* @ *r* **and** *A'*: *awalk u q w awalk w r u*
by *atomize-elim* (*rule awalk-decomp*)

then have *B*: *awalk w* (*r* @ *q*) *w* **by** *auto*

have *C*: *set* (*awalk-verts w* (*r* @ *q*)) = *set* (*awalk-verts u p*)
using ‹*awalk u p u*› *A A'* **by** (*auto simp*: *set-awalk-verts-append*)

from *A B C* **show** *?thesis* **..**
qed

lemma *rotate-trailE*:

assumes *trail u p u w* ∈ *set* (*awalk-verts u p*)
obtains *q r* **where** *p* = *q* @ *r trail w* (*r* @ *q*) *w set* (*awalk-verts w* (*r* @ *q*)) = *set* (*awalk-verts u p*)
using *assms* **by** − (*rule rotate-awalkE*[**where** *u*=*u* **and** *p*=*p* **and** *w*=*w*], *auto simp*: *trail-def*)

lemma *rotate-trailE′*:
assumes *trail u p u w* ∈ *set* (*awalk-verts u p*)
obtains *q* **where** *trail w q w set q* = *set p set* (*awalk-verts w q*) = *set* (*awalk-verts u p*)
proof −
from *assms* **obtain** *q r* **where** *p* = *q* @ *r trail w* (*r* @ *q*) *w set* (*awalk-verts w* (*r* @ *q*)) = *set* (*awalk-verts u p*)
by (*rule rotate-trailE*)
then have *set* (*r* @ *q*) = *set p* **by** *auto*
show *?thesis* **by** (*rule that*) *fact*+
qed

lemma *sym-reachableI-in-awalk*:
assumes *walk*: *awalk u p v* **and**
w1: *w1* ∈ *set* (*awalk-verts u p*) **and** *w2*: *w2* ∈ *set* (*awalk-verts u p*)
shows *w1* $\rightarrow^*_{mk\text{-}symmetric\ G}$ *w2*
proof −
from *walk w1* **obtain** *q r* **where** *p* = *q* @ *r awalk u q w1 awalk w1 r v*
by (*atomize-elim*) (*rule awalk-decomp*)
then have *w2-in*: *w2* ∈ *set* (*awalk-verts u q*) ∪ *set* (*awalk-verts w1 r*)
using *w2* **by** (*auto simp*: *set-awalk-verts-append*)

show *?thesis*
proof *cases*
assume *A*: *w2* ∈ *set* (*awalk-verts u q*)
obtain *s* **where** *awalk w2 s w1*
using *awalk-decomp*[*OF* ‹*awalk u q w1*› *A*] **by** *blast*
then have *w2* $\rightarrow^*_{mk\text{-}symmetric\ G}$ *w1*
by (*intro reachable-awalkI reachable-mk-symmetricI*)
with *symmetric-mk-symmetric* **show** *?thesis* **by** (*rule symmetric-reachable*)
next
assume *w2* ∉ *set* (*awalk-verts u q*)
then have *A*: *w2* ∈ *set* (*awalk-verts w1 r*)
using *w2-in* **by** *blast*
obtain *s* **where** *awalk w1 s w2*
using *awalk-decomp*[*OF* ‹*awalk w1 r v*› *A*] **by** *blast*
then show *w1* $\rightarrow^*_{mk\text{-}symmetric\ G}$ *w2*
by (*intro reachable-awalkI reachable-mk-symmetricI*)
qed
qed

lemma *euler-imp-connected*:
assumes *euler-trail u p v* **shows** *connected G*

proof −
{ have *verts G* ≠ {} **using** *assms* **unfolding** *euler-trail-def trail-def* **by** *auto* **}**
moreover
{ fix *w1 w2* **assume** *w1* ∈ *verts G w2* ∈ *verts G*
then have *awalk u p v w1* ∈ *set* (*awalk-verts u p*) *w2* ∈ *set* (*awalk-verts u p*)
using *assms* **by** (*auto simp*: *euler-trail-def trail-def*)
then have *w1* $\rightarrow^*_{mk\text{-}symmetric\ G}$ *w2* **by** (*rule sym-reachableI-in-awalk*) **}**
ultimately show *connected G* **by** (*rule connectedI*)
qed

end

16.2 Arc Balance of Walks

context *pre-digraph* **begin**

definition *arc-set-balance* :: *'a* ⇒ *'b set* ⇒ *int* **where**
arc-set-balance w A = *int* (*card* (*in-arcs G w* ∩ *A*)) − *int* (*card* (*out-arcs G w* ∩ *A*))

definition *arc-set-balanced* :: *'a* ⇒ *'b set* ⇒ *'a* ⇒ *bool* **where**
arc-set-balanced u A v ≡
if u = *v then* (∀ *w* ∈ *verts G*. *arc-set-balance w A* = *0*)
else (∀ *w* ∈ *verts G*. (*w* ≠ *u* ∧ *w* ≠ *v*) ⟶ *arc-set-balance w A* = *0*)
∧ *arc-set-balance u A* = −*1*
∧ *arc-set-balance v A* = *1*

abbreviation *arc-balance* :: *'a* ⇒ *'b awalk* ⇒ *int* **where**
arc-balance w p ≡ *arc-set-balance w* (*set p*)

abbreviation *arc-balanced* :: *'a* ⇒ *'b awalk* ⇒ *'a* ⇒ *bool* **where**
arc-balanced u p v ≡ *arc-set-balanced u* (*set p*) *v*

lemma *arc-set-balanced-all*:
arc-set-balanced u (*arcs G*) *v* =
(*if u* = *v then* (∀ *w* ∈ *verts G*. *in-degree G w* = *out-degree G w*)
else (∀ *w* ∈ *verts G*. (*w* ≠ *u* ∧ *w* ≠ *v*) ⟶ *in-degree G w* = *out-degree G w*)
∧ *in-degree G u* + *1* = *out-degree G u*
∧ *out-degree G v* + *1* = *in-degree G v*)
unfolding *arc-set-balanced-def arc-set-balance-def in-degree-def out-degree-def* **by** *auto*

end

context *wf-digraph* **begin**

lemma *arc-balance-Cons*:
assumes *trail u* (*e # es*) *v*
shows *arc-set-balance w* (*insert e* (*set es*)) = *arc-set-balance w* {*e*} + *arc-balance w es*
proof −
from *assms* **have** *e* ∉ *set es e* ∈ *arcs G* **by** (*auto simp*: *trail-def*)

with ‹*e* ∉ *set es*› **show** *?thesis*
apply (*cases w* = *tail G e*)
apply (*case-tac* [!] *w* = *head G e*)
apply (*auto simp*: *arc-set-balance-def*)
done
qed

lemma *arc-balancedI-trail*:
assumes *trail u p v* **shows** *arc-balanced u p v*
using *assms*
proof (*induct p arbitrary*: *u*)
case *Nil* **then show** *?case* **by** (*auto simp*: *arc-set-balanced-def arc-set-balance-def trail-def*)
next
case (*Cons e es*)
then have *arc-balanced* (*head G e*) *es v u* = *tail G e e* ∈ *arcs G*
by (*auto simp*: *awalk-Cons-iff trail-def*)
moreover
have ⋀*w*. *arc-balance w* [*e*] = (*if w* = *tail G e* ∧ *tail G e* ≠ *head G e then* −*1*
else if w = *head G e* ∧ *tail G e* ≠ *head G e then 1 else 0*)
using ‹*e* ∈ -› **by** (*case-tac w* = *tail G e*) (*auto simp*: *arc-set-balance-def*)
ultimately show *?case*
by (*auto simp*: *arc-set-balanced-def arc-balance-Cons*[*OF* ‹*trail u* - -›])
qed

lemma *trail-arc-balanceE*:
assumes *trail u p v*
obtains ⋀*w*. ⟦ *u* = *v* ∨ (*w* ≠ *u* ∧ *w* ≠ *v*); *w* ∈ *verts G* ⟧
⟹ *arc-balance w p* = *0*
and ⟦ *u* ≠ *v* ⟧ ⟹ *arc-balance u p* = − *1*
and ⟦ *u* ≠ *v* ⟧ ⟹ *arc-balance v p* = *1*
using *arc-balancedI-trail*[*OF assms*] **unfolding** *arc-set-balanced-def* **by** (*intro that*) (*metis*,*presburger*+)

end

16.3 Closed Euler Trails

lemma (**in** *wf-digraph*) *awalk-vertex-props*:
assumes *awalk u p v p* ≠ []
assumes ⋀*w*. *w* ∈ *set* (*awalk-verts u p*) ⟹ *P w* ∨ *Q w*
assumes *P u Q v*

```
  shows ∃ e ∈ set p. P (tail G e) ∧ Q (head G e)
  using assms(2,1,3−)
proof (induct p arbitrary: u rule: list-nonempty-induct)
  case (cons e es)
  show ?case
  proof (cases P (tail G e) ∧ Q (head G e))
    case False
    then have P (head G e) ∨ Q (head G e)
      using cons.prems(1) cons.prems(2)[of head G e]
      by (auto simp: awalk-Cons-iff set-awalk-verts)
    then have P (tail G e) ∧ P (head G e)
      using False using cons.prems(1,3) by auto

    then have ∃ e ∈ set es. P (tail G e) ∧ Q (head G e)
      using cons by (auto intro: cons simp: awalk-Cons-iff)
    then show ?thesis by auto
  qed auto
qed (simp add: awalk-simps)

lemma (in wf-digraph) connected-verts:
  assumes connected G arcs G ≠ {}
  shows verts G = tail G ‘ arcs G ∪ head G ‘ arcs G
proof −
  { assume verts G = {} then have ?thesis by (auto dest: tail-in-verts) }
  moreover
  { assume ∃ v. verts G = {v}
    then obtain v where verts G = {v} by (auto simp: card-Suc-eq)
    moreover
    with ‹arcs G ≠ {}› obtain e where e ∈ arcs G tail G e = v head G e = v
      by (auto dest: tail-in-verts head-in-verts)
    moreover have tail G ‘ arcs G ∪ head G ‘ arcs G ⊆ verts G by auto
    ultimately have ?thesis by auto }
  moreover
  { assume A: ∃ u v. u ∈ verts G ∧ v ∈ verts G ∧ u ≠ v
    { fix u assume u ∈ verts G

      interpret S: pair-wf-digraph mk-symmetric G by rule
      from A obtain v where v ∈ verts G u ≠ v by blast
      then obtain p where S.awalk u p v
        using ‹connected G› ‹u ∈ verts G› by (auto elim: connected-awalkE)
      with ‹u ≠ v› obtain e where e ∈ parcs (mk-symmetric G) fst e = u
        by (metis S.awalk-Cons-iff S.awalk-empty-ends list-exhaust2)
      then obtain e′ where tail G e′ = u ∨ head G e′ = u e′ ∈ arcs G
        by (force simp: parcs-mk-symmetric)
      then have u ∈ tail G ‘ arcs G ∪ head G ‘arcs G by auto }
    then have ?thesis by auto }
  ultimately show ?thesis by blast
qed
```

lemma (**in** *wf-digraph*) *connected-arcs-empty*:
assumes *connected G arcs G* = {} *verts G* ≠ {} **obtains** *v* **where** *verts G* = {*v*}
proof (*atomize-elim*, *rule ccontr*)
assume *A*: ¬ (∃ *v*. *verts G* = {*v*})

interpret *S*: *pair-wf-digraph mk-symmetric G* **by** *rule*

from ‹*verts G* ≠ {}› **obtain** *u* **where** *u* ∈ *verts G* **by** *auto*
with *A* **obtain** *v* **where** *v* ∈ *verts G u* ≠ *v* **by** *auto*

from ‹*connected G*› ‹*u* ∈ *verts G*› ‹*v* ∈ *verts G*›
obtain *p* **where** *S.awalk u p v*
using ‹*connected G*› ‹*u* ∈ *verts G*› **by** (*auto elim*: *connected-awalkE*)
with ‹*u* ≠ *v*› **obtain** *e* **where** *e* ∈ *parcs* (*mk-symmetric G*)
by (*metis S.awalk-Cons-iff S.awalk-empty-ends list-exhaust2*)
with ‹*arcs G* = {}› **show** *False*
by (*auto simp*: *parcs-mk-symmetric*)
qed

lemma (**in** *wf-digraph*) *euler-trail-conv-connected*:
assumes *connected G*
shows *euler-trail u p v* ⟷ *trail u p v* ∧ *set p* = *arcs G* (**is** *?L* ⟷ *?R*)
proof
assume *?R* **show** *?L*
proof *cases*
assume *p* = [] **with** *assms* ‹*?R*› **show** *?thesis*
by (*auto simp*: *euler-trail-def trail-def awalk-def elim*: *connected-arcs-empty*)
next
assume *p* ≠ [] **then have** *arcs G* ≠ {} **using** ‹*?R*› **by** *auto*
with *assms* ‹*?R*› ‹*p* ≠ []› **show** *?thesis*
by (*auto simp*: *euler-trail-def trail-def set-awalk-verts-not-Nil connected-verts*)
qed
qed (*simp add*: *euler-trail-def*)

lemma (**in** *wf-digraph*) *awalk-connected*:
assumes *connected G awalk u p v set p* ≠ *arcs G*
shows ∃ *e*. *e* ∈ *arcs G* − *set p* ∧ (*tail G e* ∈ *set* (*awalk-verts u p*) ∨ *head G e* ∈ *set* (*awalk-verts u p*))
proof (*rule ccontr*)
assume *A*: ¬*?thesis*

obtain *e* **where** *e* ∈ *arcs G* − *set p*
using *assms* **by** (*auto simp*: *trail-def*)
with *A* **have** *tail G e* ∉ *set* (*awalk-verts u p*) *tail G e* ∈ *verts G*
by *auto*

interpret *S*: *pair-wf-digraph mk-symmetric G* **..**

```
  have u ∈ verts G using ‹awalk u p v› by (auto simp: awalk-hd-in-verts)
  with ‹tail G e ∈ -› and ‹connected G›
  obtain q where q: S.awalk u q (tail G e)
    by (auto elim: connected-awalkE)

  have u ∈ set (awalk-verts u p)
    using ‹awalk u p v› by (auto simp: set-awalk-verts)

  have q ≠ [] using ‹u ∈ set -› ‹tail G e ∉ -› q by auto

  have ∃ e ∈ set q. fst e ∈ set (awalk-verts u p) ∧ snd e ∉ set (awalk-verts u p)
    by (rule S.awalk-vertex-props[OF ‹S.awalk - - -› ‹q ≠ []›]) (auto simp: ‹u ∈ set
-› ‹tail G e ∉ -›)
  then obtain se' where se': se' ∈ set q fst se' ∈ set (awalk-verts u p) snd se' ∉
set (awalk-verts u p)
    by auto

  from se' have se' ∈ parcs (mk-symmetric G) using q by auto
  then obtain e' where e' ∈ arcs G (tail G e' = fst se' ∧ head G e' = snd se')
∨ (tail G e' = snd se' ∧ head G e' = fst se')
    by (auto simp: parcs-mk-symmetric)
  moreover
  then have e' ∉ set p using se' ‹awalk u p v›
    by (auto dest: awalk-verts-arc2 awalk-verts-arc1)
  ultimately show False using se'
    using A by auto
qed

lemma (in wf-digraph) trail-connected:
  assumes connected G trail u p v set p ≠ arcs G
  shows ∃ e. e ∈ arcs G − set p ∧ (tail G e ∈ set (awalk-verts u p) ∨ head G e ∈
set (awalk-verts u p))
  using assms by (intro awalk-connected) (auto simp: trail-def)

theorem (in fin-digraph) closed-euler1:
  assumes con: connected G
  assumes deg: ⋀u. u ∈ verts G ⟹ in-degree G u = out-degree G u
  shows ∃ u p. euler-trail u p u
proof −
 from con obtain u where u ∈ verts G by (auto simp: connected-def strongly-connected-def)
  then have trail u [] u by (auto simp: trail-def awalk-simps)
  moreover
  { fix u p v assume  trail u p v
    then have ∃ u' p' v'. euler-trail u' p' v'
    proof (induct card (arcs G) − length p arbitrary: u p v)
      case 0
      then have u ∈ verts G by (auto simp: trail-def)

      have set p ⊆ arcs G using ‹trail u p v› by (auto simp: trail-def)
```

```
    with 0 have set p = arcs G
      by (auto simp: trail-def distinct-card[symmetric] card-seteq)
    then have euler-trail u p v
      using 0 by (simp add: euler-trail-conv-connected[OF con])
    then show ?case by blast
  next
    case (Suc n)
    then have neq: set p ≠ arcs G u ∈ verts G
      by (auto simp: trail-def distinct-card[symmetric])

    show ?case
    proof (cases u = v)
      assume u ≠ v
      then have arc-balance u p = -1
        using Suc neq by (auto elim: trail-arc-balanceE)
      then have card (in-arcs G u ∩ set p) < card (out-arcs G u ∩ set p)
        unfolding arc-set-balance-def by auto
      also have ... ≤ card (out-arcs G u)
        by (rule card-mono) auto
      finally have card (in-arcs G u ∩ set p) < card (in-arcs G u)
        using deg[OF ‹u ∈ -›] unfolding out-degree-def in-degree-def by simp
      then have in-arcs G u - set p ≠ {}
        by (auto dest: card-psubset[rotated 2])
      then obtain a where a ∈ arcs G head G a = u a ∉ set p
        by (auto simp: in-arcs-def)
      then have ∗: trail (tail G a) (a # p) v
        using Suc by (auto simp: trail-def awalk-simps)
      then show ?thesis
        using Suc by (intro Suc) auto
    next
      assume u = v
      with neq con Suc
      obtain a where a-in: a ∈ arcs G - set p
        and a-end: (tail G a ∈ set (awalk-verts u p) ∨ head G a ∈ set (awalk-verts
u p))
        by (atomize-elim) (rule trail-connected)
      have trail u p u using Suc ‹u = v› by simp
      show ?case
      proof (cases tail G a ∈ set (awalk-verts u p))
        case True
        with ‹trail u p u› obtain q where q: set p = set q trail (tail G a) q (tail
G a)
          by (rule rotate-trailE') blast
        with True a-in have ∗: trail (tail G a) (q @ [a]) (head G a)
          by (fastforce simp: trail-def awalk-simps )
        moreover
        from q Suc have length q = length p
          by (simp add: trail-def distinct-card[symmetric])
        ultimately
```

```
        show ?thesis using Suc  by (intro Suc) auto
      next
        case False
        with a-end have head G a ∈ set (awalk-verts u p) by blast
         with ‹trail u p u› obtain q where q: set p = set q trail (head G a) q
(head G a)
          by (rule rotate-trailE') blast
        with False a-in have ∗: trail (tail G a) (a # q) (head G a)
          by (fastforce simp: trail-def awalk-simps )
        moreover
        from q Suc have length q = length p
          by (simp add: trail-def distinct-card[symmetric])
        ultimately
        show ?thesis using Suc by (intro Suc) auto
      qed
    qed
  qed }
 ultimately obtain u p v where et: euler-trail u p v by blast
 moreover
 have u = v
 proof −
   have arc-balanced u p v
    using ‹euler-trail u p v› by (auto simp: euler-trail-def dest: arc-balancedI-trail)
   then show ?thesis
     using ‹euler-trail u p v› deg
    by (auto simp add: euler-trail-def trail-def arc-set-balanced-all split: if-split-asm)
 qed
 ultimately show ?thesis by blast
qed

lemma (in wf-digraph) closed-euler-imp-eq-degree:
 assumes euler-trail u p u
 assumes v ∈ verts G
 shows in-degree G v = out-degree G v
proof −
 from assms have arc-balanced u p u set p = arcs G
   unfolding euler-trail-def by (auto dest: arc-balancedI-trail)
 with assms have arc-balance v p = 0
   unfolding arc-set-balanced-def by auto
 moreover
 from ‹set p = -› have in-arcs G v ∩ set p = in-arcs G v out-arcs G v ∩ set p =
out-arcs G v
   by (auto intro: in-arcs-in-arcs out-arcs-in-arcs)
 ultimately
 show ?thesis unfolding arc-set-balance-def in-degree-def out-degree-def by auto
qed
```

theorem (**in** *fin-digraph*) *closed-euler2*:
assumes *euler-trail u p u*
shows *connected G*
and $\bigwedge u.\ u \in verts\ G \Longrightarrow in\text{-}degree\ G\ u = out\text{-}degree\ G\ u$ (**is** $\bigwedge u.$ - $\Longrightarrow$ *?eq-deg u*)
proof −
from *assms* **show** *connected G* **by** (*rule euler-imp-connected*)
next
fix *v* **assume** *A*: $v \in verts\ G$
with *assms* **show** *?eq-deg v* **by** (*rule closed-euler-imp-eq-degree*)
qed

corollary (**in** *fin-digraph*) *closed-euler*:
$(\exists u\ p.\ euler\text{-}trail\ u\ p\ u) \longleftrightarrow connected\ G \wedge (\forall u \in verts\ G.\ in\text{-}degree\ G\ u = out\text{-}degree\ G\ u)$
by (*auto dest*: *closed-euler1 closed-euler2*)

16.4 Open euler trails

Intuitively, a graph has an open euler trail if and only if it is possible to add an arc such that the resulting graph has a closed euler trail. However, this is not true in our formalization, as the arc type $'b$ might be finite:

Consider for example the graph $(\!|verts = \{0::'a,\ 1::'a\},\ arcs = \{()\},\ tail = \lambda\text{-}.\ 0::'a,\ head = \lambda\text{-}.\ 1::'a|\!)$. This graph obviously has an open euler trail, but we cannot add another arc, as we already exhausted the universe.

However, for each *fin-digraph G* there exist an isomorphic graph *H* with arc type $'a \times nat \times 'a$. Hence, we first characterize the existence of euler trail for the infinite arc type $'a \times nat \times 'a$ and transfer that result back to arbitrary arc types.

lemma *open-euler-infinite-label*:
fixes *G* :: $('a, 'a \times nat \times 'a)$ *pre-digraph*
assumes *fin-digraph G*
assumes [*simp*]: *tail G* = *fst head G* = *snd o snd*
assumes *con*: *connected G*
assumes *uv*: $u \in verts\ G\ v \in verts\ G$
assumes *deg*: $\bigwedge w.\ [\![w \in verts\ G;\ u \neq w;\ v \neq w]\!] \Longrightarrow in\text{-}degree\ G\ w = out\text{-}degree\ G\ w$
assumes *deg-in*: *in-degree G u* + *1* = *out-degree G u*
assumes *deg-out*: *out-degree G v* + *1* = *in-degree G v*
shows $\exists p.$ *pre-digraph.euler-trail G u p v*
proof −
define *label* :: $'a \times nat \times 'a \Rightarrow nat$ **where** [*simp*]: *label* = *fst o snd*

interpret *fin-digraph G* **by** *fact*

have *finite* (*label* ‘ *arcs G*) **by** *auto*
moreover have ¬*finite* (*UNIV* :: *nat set*) **by** *blast*

ultimately obtain l **where** $l \notin$ *label* ' *arcs* G **by** *atomize-elim* (*rule ex-new-if-finite*)

from *deg-in deg-out* **have** $u \neq v$ **by** *auto*

let *?e* = (*v*,*l*,*u*)

have *e-notin*:*?e* $\notin$ *arcs* G
  using ‹$l \notin$ -› **by** (*auto simp*: *image-def*)

let *?H* = *add-arc ?e*
  — We define a graph which has an closed euler trail

have [*simp*]: *verts ?H* = *verts* G **using** *uv* **by** *simp*
have [*intro*]: ⋀*a*. *compatible* (*add-arc a*) G **by** (*simp add*: *compatible-def*)

interpret H: *fin-digraph add-arc a*
  rewrites *tail* (*add-arc a*) = *tail* G **and** *head* (*add-arc a*) = *head* G
    and *pre-digraph.cas* (*add-arc a*) = *cas*
    and *pre-digraph.awalk-verts* (*add-arc a*) = *awalk-verts*
   for *a*
  by *unfold-locales* (*auto dest*: *wellformed intro*: *compatible-cas compatible-awalk-verts*
       simp: *verts-add-arc-conv*)

have ∃ *u p*. *H.euler-trail ?e u p u*
proof (*rule H.closed-euler1*)
  show *connected ?H*
  proof (*rule H.connectedI*)
    interpret *sH*: *pair-fin-digraph mk-symmetric ?H* **..**
    fix *u v* **assume** *u* ∈ *verts ?H v* ∈ *verts ?H*
    with *con* **have** *u* →**mk-symmetric G v* **by** (*auto simp*: *connected-def*)
    moreover
    have *subgraph G ?H* **by** (*auto simp*: *subgraph-def*) *unfold-locales*
    ultimately show *u* →**with-proj* (*mk-symmetric ?H*) *v*
      by (*blast intro*: *sH.reachable-mono subgraph-mk-symmetric*)
  qed (*simp add*: *verts-add-arc-conv*)
next
  fix *w* **assume** *w* ∈ *verts ?H*
  then show *in-degree ?H w* = *out-degree ?H w*
    using *deg deg-in deg-out e-notin*
    apply (*cases w* = *u*)
     apply (*case-tac* [!] *w* = *v*)
      by (*auto simp*: *in-degree-add-arc-iff out-degree-add-arc-iff*)
qed

then obtain *w p* **where** *Het*: *H.euler-trail ?e w p w* **by** *blast*
then have *?e* ∈ *set p* **by** (*auto simp*: *pre-digraph.euler-trail-def*)
then obtain *q r* **where** *p-decomp*: *p* = *q* @ [*?e*] @ *r*
  by (*auto simp*: *in-set-conv-decomp*)
  — We show now that removing the additional arc of *add-arc* (*v*, *l*, *u*) from p

yields an euler trail in G

```
  have euler-trail u (r @ q) v
  proof (unfold euler-trail-conv-connected[OF con], intro conjI)
    from Het have Ht': H.trail ?e v (?e # r @ q) v
      unfolding p-decomp H.euler-trail-def H.trail-def
      by (auto simp: p-decomp H.awalk-Cons-iff)
    then have H.trail ?e u (r @ q) v ?e ∉ set (r @ q)
      by (auto simp: H.trail-def H.awalk-Cons-iff)
    then show t': trail u (r @ q) v
      by (auto simp: trail-def H.trail-def awalk-def H.awalk-def)

    show set (r @ q) = arcs G
    proof −
      have arcs G = arcs ?H − {?e} using e-notin by auto
      also have arcs ?H = set p using Het
        by (auto simp: pre-digraph.euler-trail-def pre-digraph.trail-def)
      finally show ?thesis using ‹?e ∉ set -› by (auto simp: p-decomp)
    qed
  qed
  then show ?thesis by blast
qed

context wf-digraph begin

lemma trail-app-isoI:
  assumes t: trail u p v
    and hom: digraph-isomorphism hom
  shows pre-digraph.trail (app-iso hom G) (iso-verts hom u) (map (iso-arcs hom)
p) (iso-verts hom v)
proof −
  interpret H: wf-digraph app-iso hom G using hom ..
  from t hom have i: inj-on (iso-arcs hom) (set p)
    unfolding trail-def digraph-isomorphism-def by (auto dest:subset-inj-on[where
A=set p])
  then have distinct (map (iso-arcs hom) p) = distinct p
    by (auto simp: distinct-map dest: inj-onD)
  with t hom show ?thesis
    by (auto simp: pre-digraph.trail-def awalk-app-isoI)
qed

lemma euler-trail-app-isoI:
  assumes t: euler-trail u p v
    and hom: digraph-isomorphism hom
  shows pre-digraph.euler-trail (app-iso hom G) (iso-verts hom u) (map (iso-arcs
hom) p) (iso-verts hom v)
proof −
  from t have awalk u p v by (auto simp: euler-trail-def trail-def)
  with assms show ?thesis
```

by (*simp add*: *pre-digraph.euler-trail-def trail-app-isoI awalk-verts-app-iso-eq*)
qed

end

context *fin-digraph* **begin**

theorem *open-euler1*:
assumes *connected G*
assumes $u \in$ *verts G* $v \in$ *verts G*
assumes $\bigwedge w.\ [\![w \in \textit{verts } G;\ u \neq w;\ v \neq w]\!] \Longrightarrow \textit{in-degree } G\ w = \textit{out-degree } G\ w$
assumes *in-degree G u + 1 = out-degree G u*
assumes *out-degree G v + 1 = in-degree G v*
shows $\exists p.$ *euler-trail u p v*
proof −
obtain *f* **and** *n* :: *nat* **where** *f ' arcs G* = $\{i.\ i < n\}$
and *i*: *inj-on f* (*arcs G*)
by *atomize-elim* (*rule finite-imp-inj-to-nat-seg*, *auto*)

define *iso-f* **where** *iso-f* =
(| *iso-verts* = *id*, *iso-arcs* = ($\lambda a.$ (*tail G a*, *f a*, *head G a*)),
head = *snd o snd*, *tail* = *fst* |)
have [*simp*]: *iso-verts iso-f* = *id iso-head iso-f* = *snd o snd iso-tail iso-f* = *fst*
unfolding *iso-f-def* **by** *auto*
have *di-iso-f*: *digraph-isomorphism iso-f* **unfolding** *digraph-isomorphism-def iso-f-def*
by (*auto intro*: *inj-onI wf-digraph dest*: *inj-onD*[*OF i*])

let *?iso-g* = *inv-iso iso-f*
have [*simp*]: $\bigwedge u.\ u \in$ *verts G* $\Longrightarrow$ *iso-verts ?iso-g u* = *u*
by (*auto simp*: *inv-iso-def fun-eq-iff the-inv-into-f-eq*)

let *?H* = *app-iso iso-f G*
interpret *H*: *fin-digraph ?H* **using** *di-iso-f* **..**

have $\exists p.$ *H.euler-trail u p v*
using *di-iso-f assms i*
by (*intro open-euler-infinite-label*) (*auto simp*: *connectedI-app-iso app-iso-eq*)
then obtain *p* **where** *Het*: *H.euler-trail u p v* **by** *blast*

have *pre-digraph.euler-trail* (*app-iso ?iso-g ?H*) (*iso-verts ?iso-g u*) (*map* (*iso-arcs ?iso-g*) *p*) (*iso-verts ?iso-g v*)
using *Het* **by** (*intro H.euler-trail-app-isoI digraph-isomorphism-invI di-iso-f*)
then show *?thesis* **using** *di-iso-f* ‹*u* ∈ -› ‹*v* ∈ -› **by** *simp rule*
qed

theorem *open-euler2*:

assumes *et*: *euler-trail u p v* **and** $u \neq v$
shows *connected G* $\wedge$
$(\forall w \in$ *verts G*. $u \neq w \longrightarrow v \neq w \longrightarrow$ *in-degree G w* = *out-degree G w*$) \wedge$
in-degree G u + *1* = *out-degree G u* $\wedge$
out-degree G v + *1* = *in-degree G v*
proof −
from *et* **have** $*$: *trail u p v u* $\in$ *verts G v* $\in$ *verts G*
by (*auto simp*: *euler-trail-def trail-def awalk-hd-in-verts*)

from *et* **have** [*simp*]: $\bigwedge$*u*. *card* (*in-arcs G u* $\cap$ *set p*) = *in-degree G u*
$\bigwedge$*u*. *card* (*out-arcs G u* $\cap$ *set p*) = *out-degree G u*
by (*auto simp*: *in-degree-def out-degree-def euler-trail-def intro*: *arg-cong*[**where** *f*=*card*])

from *assms* $*$ **show** *?thesis*
by (*auto simp*: *arc-set-balance-def elim*: *trail-arc-balanceE*
intro: *euler-imp-connected*)
qed

corollary *open-euler*:
(∃ *u p v*. *euler-trail u p v* $\wedge$ $u \neq v$) $\longleftrightarrow$
connected G $\wedge$ (∃ *u v*. $u \in$ *verts G* $\wedge$ $v \in$ *verts G* $\wedge$
$(\forall w \in$ *verts G*. $u \neq w \longrightarrow v \neq w \longrightarrow$ *in-degree G w* = *out-degree G w*$) \wedge$
in-degree G u + *1* = *out-degree G u* $\wedge$
out-degree G v + *1* = *in-degree G v*) (**is** *?L* $\longleftrightarrow$ *?R*)
proof
assume *?L*
then obtain *u p v* **where** $*$: *euler-trail u p v u* $\neq$ *v*
by *auto*
then have *u* $\in$ *verts G v* $\in$ *verts G*
by (*auto simp*: *euler-trail-def trail-def awalk-hd-in-verts*)
then show *?R* **using** *open-euler2*[*OF* $*$] **by** *blast*
next
assume *?R*
then obtain *u v* **where** $*$:
connected G u $\in$ *verts G v* $\in$ *verts G*
$\bigwedge w.\ [\![w \in$ *verts G*; $u \neq w$; $v \neq w]\!] \Longrightarrow$ *in-degree G w* = *out-degree G w*
in-degree G u + *1* = *out-degree G u*
out-degree G v + *1* = *in-degree G v*
by *blast*
then have $u \neq v$ **by** *auto*
from $*$ **show** *?L* **by** (*metis open-euler1* ‹$u \neq v$›)
qed

end

end

theory *Kuratowski*
imports
Arc-Walk
Digraph-Component
Subdivision
HOL−Library.Rewrite
begin

17 Kuratowski Subgraphs

We consider the underlying undirected graphs. The underlying undirected graph is represented as a symmetric digraph.

17.1 Public definitions

definition *complete-digraph* :: *nat* ⇒ ('*a*,'*b*) *pre-digraph* ⇒ *bool* (*K*_) **where**
complete-digraph n G ≡ *graph G* ∧ *card* (*verts G*) = *n* ∧ *arcs-ends G* = {(*u*,*v*). (*u*,*v*) ∈ *verts G* × *verts G* ∧ *u* ≠ *v*}

definition *complete-bipartite-digraph* :: *nat* ⇒ *nat* ⇒ ('*a*, '*b*) *pre-digraph* ⇒ *bool* (*K*_,_) **where**
complete-bipartite-digraph m n G ≡ *graph G* ∧ (∃ *U V*. *verts G* = *U* ∪ *V* ∧ *U* ∩ *V* = {}
∧ *card U* = *m* ∧ *card V* = *n* ∧ *arcs-ends G* = *U* × *V* ∪ *V* × *U*)

definition *kuratowski-planar* :: ('*a*,'*b*) *pre-digraph* ⇒ *bool* **where**
kuratowski-planar G ≡ ¬(∃ *H*. *subgraph H G* ∧ (∃ *K rev-K rev-H*. *subdivision* (*K*, *rev-K*) (*H*, *rev-H*) ∧ (*K*$_{3,3}$ *K* ∨ *K*$_5$ *K*)))

lemma *complete-digraph-pair-def*: *K*$_n$ (*with-proj G*)
⟷ *finite* (*pverts G*) ∧ *card* (*pverts G*) = *n* ∧ *parcs G* = {(*u*,*v*). (*u*,*v*) ∈ (*pverts G* × *pverts G*) ∧ *u* ≠ *v*} (**is** - = *?R*)
proof
assume *A*: *K*$_n$ *G*
then interpret *graph with-proj G* **by** (*simp add*: *complete-digraph-def*)
show *?R* **using** *A finite-verts* **by** (*auto simp*: *complete-digraph-def*)
next
assume *A*: *?R*
moreover
then have *finite* (*pverts G* × *pverts G*) *parcs G* ⊆ *pverts G* × *pverts G*
by *auto*
then have *finite* (*parcs G*) **by** (*rule rev-finite-subset*)
ultimately interpret *pair-graph G*
by *unfold-locales* (*auto simp*: *symmetric-def split*: *prod.splits intro*: *symI*)
show *K*$_n$ *G* **using** *A finite-verts* **by** (*auto simp*: *complete-digraph-def*)
qed

lemma *complete-bipartite-digraph-pair-def*: $K_{m,n}$ (*with-proj G*) $\longleftrightarrow$ *finite* (*pverts G*)
  $\wedge$ ($\exists$ *U V*. *pverts G* = *U* $\cup$ *V* $\wedge$ *U* $\cap$ *V* = {} $\wedge$ *card U* = *m* $\wedge$ *card V* = *n* $\wedge$ *parcs G* = *U* $\times$ *V* $\cup$ *V* $\times$ *U*) (**is** - = *?R*)
proof
  assume *A*: $K_{m,n}$ *G*
  then interpret *graph G* **by** (*simp add*: *complete-bipartite-digraph-def*)
  show *?R* **using** *A finite-verts* **by** (*auto simp*: *complete-bipartite-digraph-def*)
next
  assume *A*: *?R*
  then interpret *pair-graph G*
    by *unfold-locales* (*fastforce simp*: *complete-bipartite-digraph-def symmetric-def split*: *prod.splits intro*: *symI*)+
  show $K_{m,n}$ *G* **using** *A* **by** (*auto simp*: *complete-bipartite-digraph-def*)
qed

lemma *pair-graphI-complete*:
  assumes K_n (*with-proj G*)
  shows *pair-graph G*
proof −
  from *assms* **interpret** *graph with-proj G* **by** (*simp add*: *complete-digraph-def*)
  show *pair-graph G*
    using *finite-arcs finite-verts sym-arcs wellformed no-loops* **by** *unfold-locales simp-all*
qed

lemma *pair-graphI-complete-bipartite*:
  assumes $K_{m,n}$ (*with-proj G*)
  shows *pair-graph G*
proof −
 from *assms* **interpret** *graph with-proj G* **by** (*simp add*: *complete-bipartite-digraph-def*)
  show *pair-graph G*
    using *finite-arcs finite-verts sym-arcs wellformed no-loops* **by** *unfold-locales simp-all*
qed

17.2 Inner vertices of a walk

context *pre-digraph* **begin**

definition (**in** *pre-digraph*) *inner-verts* :: *'b awalk* ⇒ *'a list* **where**
  inner-verts p ≡ *tl* (*map* (*tail G*) *p*)

lemma *inner-verts-Nil*[*simp*]: *inner-verts* [] = [] **by** (*auto simp*: *inner-verts-def*)

lemma *inner-verts-singleton*[*simp*]: *inner-verts* [*x*] = [] **by** (*auto simp*: *inner-verts-def*)

lemma (**in** *wf-digraph*) *inner-verts-Cons*:
  assumes *awalk u* (*e* # *es*) *v*

shows *inner-verts* (*e # es*) = (*if es* ≠ [] *then head G e # inner-verts es else* [])
using *assms* **by** (*induct es*) (*auto simp*: *inner-verts-def*)

lemma (**in** −) *inner-verts-with-proj-def*:
pre-digraph.inner-verts (*with-proj G*) *p* = *tl* (*map fst p*)
unfolding *pre-digraph.inner-verts-def* **by** *simp*

lemma *inner-verts-conv*: *inner-verts p* = *butlast* (*tl* (*awalk-verts u p*))
unfolding *inner-verts-def awalk-verts-conv* **by** *simp*

lemma (**in** *pre-digraph*) *inner-verts-empty*[*simp*]:
assumes *length p* < *2* **shows** *inner-verts p* = []
using *assms* **by** (*cases p*) (*auto simp*: *inner-verts-def*)

lemma (**in** *wf-digraph*) *set-inner-verts*:
assumes *apath u p v*
shows *set* (*inner-verts p*) = *set* (*awalk-verts u p*) − {*u*,*v*}
proof (*cases length p* < *2*)
case *True* **with** *assms* **show** *?thesis*
by (*cases p*) (*auto simp*: *inner-verts-conv*[*of* - *u*] *apath-def*)
next
case *False*
have *awalk-verts u p* = *u # inner-verts p* @ [*v*]
using *assms False length-awalk-verts*[*of u p*] *inner-verts-conv*[*of p u*]
by (*cases awalk-verts u p*) (*auto simp*: *apath-def awalk-conv*)
then show *?thesis* **using** *assms* **by** (*auto simp*: *apath-def*)
qed

lemma *in-set-inner-verts-appendI-l*:
assumes *u* ∈ *set* (*inner-verts p*)
shows *u* ∈ *set* (*inner-verts* (*p* @ *q*))
using *assms*
by (*induct p*) (*auto simp*: *inner-verts-def*)

lemma *in-set-inner-verts-appendI-r*:
assumes *u* ∈ *set* (*inner-verts q*)
shows *u* ∈ *set* (*inner-verts* (*p* @ *q*))
using *assms*
by (*induct p*) (*auto simp*: *inner-verts-def dest*: *list-set-tl*)

end

17.3 Progressing Walks

We call a walk *progressing* if it does not contain the sequence $[(x, y), (y, x)]$. This concept is relevant in particular for *iapath*s: If all of the inner vertices have degree at most 2 this implies that such a walk is a trail and even a path.

definition *progressing* :: (*'a* × *'a*) *awalk* ⇒ *bool* **where**
progressing p ≡ ∀ *xs x y ys*. *p* ≠ *xs* @ (*x*,*y*) # (*y*,*x*) # *ys*

lemma *progressing-Nil*: *progressing* []
by (*auto simp*: *progressing-def*)

lemma *progressing-single*: *progressing* [*e*]
by (*auto simp*: *progressing-def*)

lemma *progressing-ConsD*:
assumes *progressing* (*e* # *es*) **shows** *progressing es*
using *assms* **unfolding** *progressing-def* **by** (*metis* (*no-types*) *append-eq-Cons-conv*)

lemma *progressing-Cons*:
progressing (*x* # *xs*) ⟷ (*xs* = [] ∨ (*xs* ≠ [] ∧ ¬(*fst x* = *snd* (*hd xs*) ∧ *snd x* = *fst* (*hd xs*)) ∧ *progressing xs*)) (**is** *?L* = *?R*)
proof
assume *?L*
show *?R*
proof (*cases xs*)
case *Nil* **then show** *?thesis* **by** *auto*
next
case (*Cons x' xs'*)
then have ⋀*u v*. (*x* # *x'* # *xs'*) ≠ [] @ (*u*,*v*) # (*v*,*u*) # *xs'* **using** ‹*?L*› **unfolding** *progressing-def* **by** *metis*
then have ¬(*fst x* = *snd x'* ∧ *snd x* = *fst x'*) **by** (*cases x*) (*cases x'*, *auto*)
with *Cons* **show** *?thesis* **using** ‹*?L*› **by** (*auto dest*: *progressing-ConsD*)
qed
next
assume *?R* **then show** *?L* **unfolding** *progressing-def*
by (*auto simp add*: *Cons-eq-append-conv*)
qed

lemma *progressing-Cons-Cons*:
progressing ((*u*,*v*) # (*v*,*w*) # *es*) ⟷ *u* ≠ *w* ∧ *progressing* ((*v*,*w*) # *es*) (**is** *?L* ⟷ *?R*)
by (*auto simp*: *progressing-Cons*)

lemma *progressing-appendD1*:
assumes *progressing* (*p* @ *q*) **shows** *progressing p*
using *assms* **unfolding** *progressing-def* **by** (*metis append-Cons append-assoc*)

lemma *progressing-appendD2*:
assumes *progressing* (*p* @ *q*) **shows** *progressing q*
using *assms* **unfolding** *progressing-def* **by** (*metis append-assoc*)

lemma *progressing-rev-path*:
progressing (*rev-path p*) = *progressing p* (**is** *?L* = *?R*)
proof

```
  assume ?L
  show ?R unfolding progressing-def
  proof (intro allI notI)
    fix xs x y ys l1 l2 assume p = xs @ (x,y) # (y,x) # ys
    then have rev-path p = rev-path ys @ (x,y) # (y,x) # rev-path xs
      by simp
    then show False using ‹?L› unfolding progressing-def by auto
  qed
next
  assume ?R
  show ?L unfolding progressing-def
  proof (intro allI notI)
    fix xs x y ys l1 l2 assume rev-path p = xs @ (x,y) # (y,x) # ys
    then have rev-path (rev-path p) = rev-path ys @ (x,y) # (y,x) # rev-path xs
      by simp
    then show False using ‹?R› unfolding progressing-def by auto
  qed
qed

lemma progressing-append-iff:
  shows progressing (xs @ ys) ⟷ progressing xs ∧ progressing ys
      ∧ (xs ≠ [] ∧ ys ≠ [] ⟶ (fst (last xs) ≠ snd (hd ys) ∨ snd (last xs) ≠ fst (hd
ys)))
proof (induct ys arbitrary: xs)
  case Nil then show ?case by (auto simp: progressing-Nil)
next
  case (Cons y' ys')
  let - = ?R = ?case
  have ∗: xs ≠ [] ⟹ hd (rev-path xs) = prod.swap (last xs) by (induct xs) auto

  have progressing (xs @ y' # ys') ⟷ progressing ((xs @ [y']) @ ys')
    by simp
  also have ... ⟷ progressing (xs @ [y']) ∧ progressing ys' ∧ (ys' ≠ [] ⟶ (fst
y' ≠ snd (hd ys') ∨ snd y' ≠ fst (hd ys')))
    by (subst Cons) simp
  also have ... ⟷ ?R
     by (auto simp: progressing-Cons progressing-Nil progressing-rev-path[where
p=xs @ -,symmetric] ∗ progressing-rev-path prod.swap-def)
  finally show ?case .
qed
```

17.4 Walks with Restricted Vertices

```
definition verts3 :: ('a, 'b) pre-digraph ⇒ 'a set where
  verts3 G ≡ {v ∈ verts G. 2 < in-degree G v}
```

A path were only the end nodes may be in V

```
definition (in pre-digraph) gen-iapath :: 'a set ⇒ 'a ⇒ 'b awalk ⇒ 'a ⇒ bool
where
```

gen-iapath V u p v ≡ *u* ∈ *V* ∧ *v* ∈ *V* ∧ *apath u p v* ∧ *set* (*inner-verts p*) ∩ *V* = {} ∧ *p* ≠ []

abbreviation (**in** *pre-digraph*) (*input*) *iapath* :: *'a* ⇒ *'b awalk* ⇒ *'a* ⇒ *bool* **where**
iapath u p v ≡ *gen-iapath* (*verts3 G*) *u p v*

definition *gen-contr-graph* :: (*'a*,*'b*) *pre-digraph* ⇒ *'a set* ⇒ *'a pair-pre-digraph* **where**
gen-contr-graph G V ≡ (|
pverts = *V*,
parcs = {(*u*,*v*). ∃ *p*. *pre-digraph.gen-iapath G V u p v*}
|)

abbreviation (*input*) *contr-graph* :: *'a pair-pre-digraph* ⇒ *'a pair-pre-digraph* **where**
contr-graph G ≡ *gen-contr-graph G* (*verts3 G*)

17.5 Properties of subdivisions

lemma (**in** *pair-sym-digraph*) *verts3-subdivide*:
assumes *e* ∈ *parcs G w* ∉ *pverts G*
shows*verts3* (*subdivide G e w*) = *verts3 G*
proof −
let *?sG* = *subdivide G e w*
obtain *u v* **where** *e-conv*[*simp*]: *e* = (*u*,*v*) **by** (*cases e*) *auto*

from ‹*w* ∉ *pverts G*› **have** *w-arcs*: (*u*,*w*) ∉ *parcs G* (*v*,*w*) ∉ *parcs G* (*w*,*u*) ∉ *parcs G* (*w*,*v*) ∉ *parcs G*
by (*auto dest*: *wellformed*)
have *G-arcs*: (*u*,*v*) ∈ *parcs G* (*v*,*u*) ∈ *parcs G*
using ‹*e* ∈ *parcs G*› **by** (*auto simp*: *arcs-symmetric*)

have {*v* ∈ *pverts G*. 2 < *in-degree G v*} = {*v* ∈ *pverts G*. 2 < *in-degree ?sG v*}
proof −
{ fix *x* **assume** *x* ∈ *pverts G*
define *card-eq* **where** *card-eq x* ⟷ *in-degree ?sG x* = *in-degree G x* **for** *x*

have *in-arcs ?sG u* = (*in-arcs G u* − {(*v*,*u*)}) ∪ {(*w*,*u*)}
in-arcs ?sG v = (*in-arcs G v* − {(*u*,*v*)}) ∪ {(*w*,*v*)}
using *w-arcs G-arcs* **by** *auto*
then have *card-eq u card-eq v*
unfolding *card-eq-def in-degree-def* **using** *w-arcs G-arcs*
apply −
apply (*cases finite* (*in-arcs G u*); *simp add*: *card-Suc-Diff1 del*: *card-Diff-insert*)
apply (*cases finite* (*in-arcs G v*); *simp add*: *card-Suc-Diff1 del*: *card-Diff-insert*)
done
moreover
have *x* ∉ {*u*,*v*} ⟹ *in-arcs ?sG x* = *in-arcs G x*
using ‹*x* ∈ *pverts G*› ‹*w* ∉ *pverts G*› **by** *auto*
then have *x* ∉ {*u*,*v*} ⟹ *card-eq x* **by** (*simp add*: *in-degree-def card-eq-def*)

```
      ultimately have card-eq x by fast
      then have in-degree G x = in-degree ?sG x
        unfolding card-eq-def by simp }
    then show ?thesis by auto
  qed
  also have ... = {v∈pverts ?sG. 2 < in-degree ?sG v}
  proof −
    have in-degree ?sG w ≤ 2
    proof −
      have in-arcs ?sG w = {(u,w), (v,w)}
        using ‹w ∉ pverts G› G-arcs(1) by (auto simp: wellformed')
      then show ?thesis
        unfolding in-degree-def by (auto simp: card-insert-if)
    qed
    then show ?thesis using G-arcs assms by auto
  qed
  finally show ?thesis by (simp add: verts3-def)
qed

lemma sd-path-Nil-iff:
  sd-path e w p = [] ⟷ p = []
  by (cases (e,w,p) rule: sd-path.cases) auto

lemma (in pair-sym-digraph) gen-iapath-sd-path:
  fixes e :: 'a × 'a and w :: 'a
  assumes elems: e ∈ parcs G w ∉ pverts G
  assumes V: V ⊆ pverts G
  assumes path: gen-iapath V u p v
  shows pre-digraph.gen-iapath (subdivide G e w) V u (sd-path e w p) v
proof −
  obtain x y where e-conv: e = (x,y) by (cases e) auto
  interpret S: pair-sym-digraph subdivide G e w
    using elems by (auto intro: pair-sym-digraph-subdivide)

  from path have apath u p v by (auto simp: gen-iapath-def)
  then have apath-sd: S.apath u (sd-path e w p) v and
    set-ev-sd: set (S.awalk-verts u (sd-path e w p)) ⊆ set (awalk-verts u p) ∪ {w}
    using elems by (rule apath-sd-path set-awalk-verts-sd-path)+
  have w ∉ {u,v} using elems ‹apath u p v›
    by (auto simp: apath-def awalk-hd-in-verts awalk-last-in-verts)

  have set (S.inner-verts (sd-path e w p)) = set (S.awalk-verts u (sd-path e w p))
− {u,v}
    using apath-sd by (rule S.set-inner-verts)
  also have ... ⊆ set (awalk-verts u p) ∪ {w} − {u,v}
    using set-ev-sd by auto
  also have ... = set (inner-verts p) ∪ {w}
    using set-inner-verts[OF ‹apath u p v›] ‹w ∉ {u,v}› by blast
  finally have set (S.inner-verts (sd-path e w p)) ∩ V ⊆ (set (inner-verts p) ∪
```

$\{w\}) \cap V$
using *V* **by** *blast*
also have $\ldots \subseteq \{\}$
using *path elems V* **unfolding** *gen-iapath-def* **by** *auto*
finally show *?thesis*
using *apath-sd elems path* **by** (*auto simp*: *gen-iapath-def S.gen-iapath-def sd-path-Nil-iff*)
qed

lemma (**in** *pair-sym-digraph*)
assumes *elems*: $e \in parcs\ G\ w \notin pverts\ G$
assumes *V*: $V \subseteq pverts\ G$
assumes *path*: *pre-digraph.gen-iapath* (*subdivide G e w*) *V u p v*
shows *gen-iapath-co-path*: *gen-iapath V u* (*co-path e w p*) *v* (**is** *?thesis-path*)
and *set-awalk-verts-co-path'*: *set* (*awalk-verts u* (*co-path e w p*)) = *set* (*awalk-verts u p*) − {*w*} (**is** *?thesis-set*)
proof −
interpret *S*: *pair-sym-digraph subdivide G e w*
using *elems* **by** (*rule pair-sym-digraph-subdivide*)

have *uv*: $u \in pverts\ G\ v \in pverts\ G$ *S.apath u p v* **using** *V path* **by** (*auto simp*: *S.gen-iapath-def*)
note *co* = *apath-co-path*[*OF elems uv*] *set-awalk-verts-co-path*[*OF elems uv*]

show *?thesis-set* **by** (*fact co*)
show *?thesis-path* **using** *co path* **unfolding** *gen-iapath-def S.gen-iapath-def* **using** *elems*
by (*clarsimp simp add*: *set-inner-verts*[*of u*] *S.set-inner-verts*[*of u*]) *blast*
qed

17.6 Pair Graphs

context *pair-sym-digraph* **begin**

lemma *gen-iapath-rev-path*:
gen-iapath V v (*rev-path p*) *u* = *gen-iapath V u p v* (**is** *?L* = *?R*)
proof −
{ fix *u p v* **assume** *gen-iapath V u p v*
then have *butlast* (*tl* (*awalk-verts v* (*rev-path p*))) = *rev* (*butlast* (*tl* (*awalk-verts u p*)))
by (*auto simp*: *tl-rev butlast-rev butlast-tl awalk-verts-rev-path gen-iapath-def apath-def*)
with ‹*gen-iapath V u p v*› **have** *gen-iapath V v* (*rev-path p*) *u*
by (*auto simp*: *gen-iapath-def apath-def inner-verts-conv*[*symmetric*] *awalk-verts-rev-path*)
}
note *RL* = *this*
show *?thesis* **by** (*auto dest*: *RL intro*: *RL*)
qed

lemma *inner-verts-rev-path*:
assumes *awalk u p v*
shows *inner-verts* (*rev-path p*) = *rev* (*inner-verts p*)
by (*metis assms butlast-rev butlast-tl awalk-verts-rev-path inner-verts-conv tl-rev*)

end

context *pair-pseudo-graph* **begin**

lemma *apath-imp-progressing*:
assumes *apath u p v* **shows** *progressing p*
proof (*rule ccontr*)
assume ¬*?thesis*
then obtain *xs x y ys* **where** ∗: *p* = *xs* @ (*x*,*y*) # (*y*,*x*) # *ys*
unfolding *progressing-def* **by** *auto*
then have ¬*apath u p v*
by (*simp add*: *apath-append-iff apath-simps hd-in-awalk-verts*)
then show *False* **using** *assms* **by** *auto*
qed

lemma *awalk-Cons-deg2-unique*:
assumes *awalk u p v p* ≠ []
assumes *in-degree G u* ≤ *2*
assumes *awalk u1* (*e1* # *p*) *v awalk u2* (*e2* # *p*) *v*
assumes *progressing* (*e1* # *p*) *progressing* (*e2* # *p*)
shows *e1* = *e2*
proof (*cases p*)
case (*Cons e es*)
show *?thesis*
proof (*rule ccontr*)
assume *e1* ≠ *e2*
define *x* **where** *x* = *snd e*
then have *e-unf*:*e* = (*u*,*x*) **using** ‹*awalk u p v*› *Cons* **by** (*auto simp*: *awalk-simps*)
then have *ei-unf*: *e1* = (*u1*, *u*) *e2* = (*u2*, *u*)
using *Cons assms* **by** (*auto simp*: *apath-simps prod-eqI*)
with *Cons assms* ‹*e* = (*u*,*x*)› ‹*e1* ≠ *e2*› **have** *u1* ≠ *u2 x* ≠ *u1 x* ≠ *u2*
by (*auto simp*: *progressing-Cons-Cons*)
moreover have {(*u1*, *u*), (*u2*, *u*), (*x*,*u*)} ⊆ *parcs G*
using *e-unf ei-unf Cons assms* **by** (*auto simp*: *awalk-simps intro*: *arcs-symmetric*)
then have *finite* (*in-arcs G u*)
and {(*u1*, *u*), (*u2*, *u*), (*x*,*u*)} ⊆ *in-arcs G u* **by** *auto*
then have *card* ({(*u1*, *u*), (*u2*, *u*), (*x*,*u*)}) ≤ *in-degree G u*
unfolding *in-degree-def* **by** (*rule card-mono*)
ultimately show *False* **using** ‹*in-degree G u* ≤ *2*› **by** *auto*
qed
qed (*simp add*: ‹*p* ≠ []›)

lemma *same-awalk-by-same-end*:
assumes *V*: *verts3 G* ⊆ *V V* ⊆ *pverts G*

```
    and walk: awalk u p v awalk u q w hd p = hd q p ≠ [] q ≠ []
    and progress: progressing p progressing q
    and tail: v ∈ V w ∈ V
    and inner-verts: set (inner-verts p) ∩ V = {}
      set (inner-verts q) ∩ V = {}
  shows p = q
  using walk progress inner-verts
proof (induct p q arbitrary: u rule: list-induct2'[case-names Nil-Nil Cons-Nil Nil-Cons
Cons-Cons])
  case (Cons-Cons a as b bs)
  from ‹hd (a # -) = hd ->› have a = b by simp

  { fix a as v b bs w
    assume A: awalk u (a # as) v awalk u (b # bs) w
        set (inner-verts (b # bs)) ∩ V = {} v ∈ V a = b as = []
   then have bs = [] by - (rule ccontr, auto simp: inner-verts-Cons awalk-simps)
  } note Nil-imp-Nil = this

  show ?case
  proof (cases as = [])
    case True
   then have bs = [] using Cons-Cons.prems ‹a = b› tail by (metis Nil-imp-Nil)
    then show ?thesis using True ‹a = b› by simp
  next
    case False
   then have bs ≠ [] using Cons-Cons.prems ‹a = b› tail by (metis Nil-imp-Nil)

    obtain a' as' where as = a' # as' using ‹as ≠ []› by (cases as) simp
    obtain b' bs' where bs = b' # bs' using ‹bs ≠ []› by (cases bs) simp

    let ?arcs = {(fst a, snd a), (snd a', snd a), (snd b', snd a)}

    have card {fst a, snd a', snd b'} = card (fst ` ?arcs) by auto
    also have ... = card ?arcs by (rule card-image) (cases a, auto)
    also have ... ≤ in-degree G (snd a)
    proof -
      have ?arcs ⊆ in-arcs G (snd a)
        using ‹progressing (a # as)› ‹progressing (b # bs)› ‹awalk - (a # as) ->
‹awalk - (b # bs) ->
       unfolding ‹a = b› ‹as = ->› ‹bs = ->
       by (cases b; cases a') (auto simp: progressing-Cons-Cons awalk-simps intro:
arcs-symmetric)
      with -show ?thesis unfolding in-degree-def by (rule card-mono) auto
    qed
    also have ... ≤ 2
    proof -
      have snd a ∉ V snd a ∈ pverts G
        using Cons-Cons.prems ‹as ≠ []› by (auto simp: inner-verts-Cons)
      then show ?thesis using V by (auto simp: verts3-def)
```

```
    qed
    finally have fst a = snd a' ∨ fst a = snd b' ∨ snd a' = snd b'
      by (auto simp: card-insert-if split: if-splits)
    then have hd as = hd bs
     using ‹progressing (a # as)› ‹progressing (b # bs)› ‹awalk - (a # as) -› ‹awalk
- (b # bs) -›
      unfolding ‹a = b› ‹as = -› ‹bs = -›
     by (cases b, cases a', cases b') (auto simp: progressing-Cons-Cons awalk-simps)
    then show ?thesis
      using ‹as ≠ []› ‹bs ≠ []› Cons-Cons.prems
      by (auto dest: progressing-ConsD simp: awalk-simps inner-verts-Cons intro!:
Cons-Cons)
  qed
qed simp-all

lemma same-awalk-by-common-arc:
  assumes V: verts3 G ⊆ V V ⊆ pverts G
  assumes walk: awalk u p v awalk w q x
  assumes progress: progressing p progressing q
  assumes iv-not-in-V: set (inner-verts p) ∩ V = {} set (inner-verts q) ∩ V =
{}
  assumes ends-in-V: {u,v,w,x} ⊆ V
  assumes arcs: e ∈ set p e ∈ set q
  shows p = q
proof −
 from arcs obtain p1 p2 where p-decomp: p = p1 @ e # p2 by (metis in-set-conv-decomp-first)
 from arcs obtain q1 q2 where q-decomp: q = q1 @ e # q2 by (metis in-set-conv-decomp-first)

  { define p1' q1' where p1' = rev-path (p1 @ [e]) and q1' = rev-path (q1 @
[e])
    then have decomp: p = rev-path p1' @ p2 q = rev-path q1' @ q2
      and awlast u (rev-path p1') = snd e awlast w (rev-path q1') = snd e
     using p-decomp q-decomp walk by (auto simp: awlast-append awalk-verts-rev-path)
    then have walk': awalk (snd e) p1' u awalk (snd e) q1' w
      using walk by auto
     moreover have hd p1' = hd q1' p1' ≠ [] q1' ≠ [] by (auto simp: p1'-def
q1'-def)
    moreover have progressing p1' progressing q1'
      using progress unfolding decomp by (auto dest: progressing-appendD1 simp:
progressing-rev-path)
    moreover
    have set (inner-verts (rev-path p1')) ∩ V = {} set (inner-verts (rev-path q1'))
∩ V = {}
      using iv-not-in-V unfolding decomp
      by (auto intro: in-set-inner-verts-appendI-l in-set-inner-verts-appendI-r)
    then have u ∈ V w ∈ V set (inner-verts p1') ∩ V = {} set (inner-verts q1')
∩ V = {}
      using ends-in-V iv-not-in-V walk unfolding decomp
      by (auto simp: inner-verts-rev-path)
```

ultimately have *p1' = q1'* **by** (*rule same-awalk-by-same-end*[*OF V*]) **}**
moreover
{ define *p2' q2'* **where** *p2' = e # p2* **and** *q2' = e # q2*
then have *decomp*: *p = p1 @ p2' q = q1 @ q2'*
using *p-decomp q-decomp* **by** (*auto simp*: *awlast-append*)
moreover
have *awlast u p1 = fst e awlast w q1 = fst e*
using *p-decomp q-decomp walk* **by** *auto*
ultimately
have ∗: *awalk* (*fst e*) *p2' v awalk* (*fst e*) *q2' x*
using *walk* **by** *auto*
moreover have *hd p2' = hd q2' p2' ≠ [] q2' ≠ []* **by** (*auto simp*: *p2'-def q2'-def*)
moreover have *progressing p2' progressing q2'*
using *progress* **unfolding** *decomp* **by** (*auto dest*: *progressing-appendD2*)
moreover
have *v ∈ V x ∈ V set* (*inner-verts p2'*) ∩ *V* = {} *set* (*inner-verts q2'*) ∩ *V* = {}
using *ends-in-V iv-not-in-V* **unfolding** *decomp*
by (*auto intro*: *in-set-inner-verts-appendI-l in-set-inner-verts-appendI-r*)
ultimately have *p2' = q2'* **by** (*rule same-awalk-by-same-end*[*OF V*]) **}**
ultimately
show *p = q* **using** *p-decomp q-decomp* **by** (*auto simp*: *rev-path-eq*)
qed

lemma *same-gen-iapath-by-common-arc*:
assumes *V*: *verts3 G* ⊆ *V V* ⊆ *pverts G*
assumes *path*: *gen-iapath V u p v gen-iapath V w q x*
assumes *arcs*: *e ∈ set p e ∈ set q*
shows *p = q*
proof −
from *path* **have** *awalk*: *awalk u p v awalk w q x progressing p progressing q*
and *in-V*: *set* (*inner-verts p*) ∩ *V* = {} *set* (*inner-verts q*) ∩ *V* = {} {*u*,*v*,*w*,*x*} ⊆ *V*
by (*auto simp*: *gen-iapath-def apath-imp-progressing apath-def*)
from *V awalk in-V arcs* **show** *?thesis* **by** (*rule same-awalk-by-common-arc*)
qed

end

17.7 Slim graphs

We define the notion of a slim graph. The idea is that for a slim graph G, G is a subdivision of *gen-contr-graph* (*with-proj G*) (*verts3* (*with-proj G*)).

context *pair-pre-digraph* **begin**

definition (**in** *pair-pre-digraph*) *is-slim* :: *'a set* ⇒ *bool* **where**
is-slim V ≡

$(\forall\, v \in \mathit{pverts}\ G.\ v \in V \vee$
$\mathit{in\text{-}degree}\ G\ v \leq 2 \wedge (\exists\, x\ p\ y.\ \mathit{gen\text{-}iapath}\ V\ x\ p\ y \wedge v \in \mathit{set}\ (\mathit{awalk\text{-}verts}\ x\ p)))$
$\wedge$
$(\forall\, e \in \mathit{parcs}\ G.\ \mathit{fst}\ e \neq \mathit{snd}\ e \wedge (\exists\, x\ p\ y.\ \mathit{gen\text{-}iapath}\ V\ x\ p\ y \wedge e \in \mathit{set}\ p)) \wedge$
$(\forall\, u\ v\ p\ q.\ (\mathit{gen\text{-}iapath}\ V\ u\ p\ v \wedge \mathit{gen\text{-}iapath}\ V\ u\ q\ v) \longrightarrow p = q) \wedge$
$V \subseteq \mathit{pverts}\ G$

definition *direct-arc* :: $'a \times 'a \Rightarrow 'a \times 'a$ **where**
direct-arc uv $\equiv$ *SOME e.* $\{\mathit{fst}\ uv\ ,\ \mathit{snd}\ uv\} = \{\mathit{fst}\ e,\ \mathit{snd}\ e\}$

definition *choose-iapath* :: $'a \Rightarrow 'a \Rightarrow ('a \times 'a)\ \mathit{awalk}$ **where**
choose-iapath u v $\equiv$ (*let*
chosen-path = $(\lambda u\ v.\ \mathit{SOME}\ p.\ \mathit{iapath}\ u\ p\ v)$
in if direct-arc $(u,v) = (u,v)$ *then chosen-path u v else rev-path* (*chosen-path v u*))

definition *slim-paths* :: $('a \times ('a \times 'a)\ \mathit{awalk} \times 'a)\ \mathit{set}$ **where**
slim-paths $\equiv (\lambda e.\ (\mathit{fst}\ e,\ \mathit{choose\text{-}iapath}\ (\mathit{fst}\ e)\ (\mathit{snd}\ e),\ \mathit{snd}\ e))$ ‘ *parcs* (*contr-graph G*)

definition *slim-verts* :: $'a\ \mathit{set}$ **where**
slim-verts $\equiv$ *verts3 G* $\cup$ $(\bigcup (u,p,\text{-}) \in \mathit{slim\text{-}paths}.\ \mathit{set}\ (\mathit{awalk\text{-}verts}\ u\ p))$

definition *slim-arcs* :: $'a\ \mathit{rel}$ **where**
slim-arcs $\equiv \bigcup (\text{-},p,\text{-}) \in \mathit{slim\text{-}paths}.\ \mathit{set}\ p$

Computes a slim subgraph for an arbitrary *pair-digraph*

definition *slim* :: $'a$ *pair-pre-digraph* **where**
slim $\equiv (\!|\ \mathit{pverts} = \mathit{slim\text{-}verts},\ \mathit{parcs} = \mathit{slim\text{-}arcs}\ |\!)$

end

lemma (**in** *wf-digraph*) *iapath-dist-ends*: $\bigwedge u\ p\ v.\ \mathit{iapath}\ u\ p\ v \Longrightarrow u \neq v$
unfolding *pre-digraph.gen-iapath-def* **by** (*metis apath-ends*)

context *pair-sym-digraph* **begin**

lemma *choose-iapath*:
assumes $\exists\, p.\ \mathit{iapath}\ u\ p\ v$
shows *iapath u* (*choose-iapath u v*) *v*
proof (*cases direct-arc* $(u,v) = (u,v)$)
define *chosen* **where** *chosen u v* = (*SOME p. iapath u p v*) **for** *u v*
{ case *True*
have *iapath u* (*chosen u v*) *v*
unfolding *chosen-def* **by** (*rule someI-ex*) (*rule assms*)
then show *?thesis* **using** *True* **by** (*simp add*: *choose-iapath-def chosen-def*) **}**

```
  { case False
    from assms obtain p where iapath u p v by auto
    then have iapath v (rev-path p) u
      by (simp add: gen-iapath-rev-path)
    then have iapath v (chosen v u) u
      unfolding chosen-def by (rule someI)
    then show ?thesis using False
      by (simp add: choose-iapath-def chosen-def gen-iapath-rev-path) }
qed

lemma slim-simps: pverts slim = slim-verts parcs slim = slim-arcs
  by (auto simp: slim-def)

lemma slim-paths-in-G-E:
  assumes (u,p,v) ∈ slim-paths obtains iapath u p v u ≠ v
  using assms choose-iapath
  by (fastforce simp: gen-contr-graph-def slim-paths-def dest: iapath-dist-ends)

lemma verts-slim-in-G: pverts slim ⊆ pverts G
  by (auto simp: slim-simps slim-verts-def verts3-def gen-iapath-def apath-def
    elim!: slim-paths-in-G-E elim!: awalkE)

lemma verts3-in-slim-G[simp]:
  assumes x ∈ verts3 G shows x ∈ pverts slim
using assms by (auto simp: slim-simps slim-verts-def)

lemma arcs-slim-in-G: parcs slim ⊆ parcs G
  by (auto simp: slim-simps slim-arcs-def gen-iapath-def apath-def
      elim!: slim-paths-in-G-E elim!: awalkE)

lemma slim-paths-in-slimG:
  assumes (u,p,v) ∈ slim-paths
  shows pre-digraph.gen-iapath slim (verts3 G) u p v ∧ p ≠ []
proof −
  from assms have arcs: ⋀e. e ∈ set p ⟹ e ∈ parcs slim
    by (auto simp: slim-simps slim-arcs-def)
  moreover
  from assms have gen-iapath (verts3 G) u p v and p ≠ []
    by (auto simp: gen-iapath-def elim!: slim-paths-in-G-E)
  ultimately show ?thesis
   by (auto simp: pre-digraph.gen-iapath-def pre-digraph.apath-def pre-digraph.awalk-def
      inner-verts-with-proj-def)
qed

lemma direct-arc-swapped:
  direct-arc (u,v) = direct-arc (v,u)
by (simp add: direct-arc-def insert-commute)

lemma direct-arc-chooses:
```

fixes *u v* :: *'a* **shows** *direct-arc* (*u*,*v*) = (*u*,*v*) ∨ *direct-arc* (*u*,*v*) = (*v*,*u*)
proof −
define *f* :: *'a set* ⇒ *'a* × *'a*
where *f X* = (*SOME e*. *X* = {*fst e*,*snd e*}) **for** *X*

have ∃ *p*::*'a* × *'a*. {*u*,*v*} = {*fst p*, *snd p*} **by** (*rule exI*[**where** *x*=(*u*,*v*)]) *auto*
then have {*u*,*v*} = {*fst* (*f* {*u*,*v*}), *snd* (*f* {*u*,*v*})}
unfolding *f-def* **by** (*rule someI-ex*)
then have *f* {*u*,*v*} = (*u*,*v*) ∨ *f* {*u*,*v*} = (*v*,*u*)
by (*auto simp*: *doubleton-eq-iff prod-eq-iff*)
then show *?thesis* **by** (*auto simp*: *direct-arc-def f-def*)
qed

lemma *rev-path-choose-iapath*:
assumes *u* ≠ *v*
shows *rev-path* (*choose-iapath u v*) = *choose-iapath v u*
using *assms direct-arc-chooses*[*of u v*]
by (*auto simp*: *choose-iapath-def direct-arc-swapped*)

lemma *no-loops-in-iapath*: *gen-iapath V u p v* ⟹ *a* ∈ *set p* ⟹ *fst a* ≠ *snd a*
by (*auto simp*: *gen-iapath-def no-loops-in-apath*)

lemma *pair-bidirected-digraph-slim*: *pair-bidirected-digraph slim*
proof
fix *e* **assume** *A*: *e* ∈ *parcs slim*
then obtain *u p v* **where** (*u*,*p*,*v*) ∈ *slim-paths e* ∈ *set p* **by** (*auto simp*: *slim-simps slim-arcs-def*)
with *A* **have** *iapath u p v* **by** (*auto elim*: *slim-paths-in-G-E*)
with ‹*e* ∈ *set p*› **have** *fst e* ∈ *set* (*awalk-verts u p*) *snd e* ∈ *set* (*awalk-verts u p*)
by (*auto simp*: *set-awalk-verts gen-iapath-def apath-def*)
moreover
from ‹- ∈ *slim-paths*› **have** *set* (*awalk-verts u p*) ⊆ *pverts slim*
by (*auto simp*: *slim-simps slim-verts-def*)
ultimately
show *fst e* ∈ *pverts slim snd e* ∈ *pverts slim* **by** *auto*

show *fst e* ≠ *snd e*
using ‹*iapath u p v*› ‹*e* ∈ *set p* › **by** (*auto dest*: *no-loops-in-iapath*)
next
{ fix *e* **assume** *e* ∈ *parcs slim*
then obtain *u p v* **where** (*u*,*p*,*v*) ∈ *slim-paths* **and** *e* ∈ *set p*
by (*auto simp*: *slim-simps slim-arcs-def*)
moreover
then have *iapath u p v* **and** *p* ≠ [] **and** *u* ≠ *v* **by** (*auto elim*: *slim-paths-in-G-E*)
then have *iapath v* (*rev-path p*) *u* **and** *rev-path p* ≠ [] **and** *v* ≠ *u*
by (*auto simp*: *gen-iapath-rev-path*)
then have (*v*,*u*) ∈ *parcs* (*contr-graph G*)
by (*auto simp*: *gen-contr-graph-def*)
moreover

```
    from ‹iapath u p v› have u ≠ v
      by (auto simp: gen-iapath-def dest: apath-nonempty-ends)
    ultimately
    have (v, rev-path p, u) ∈ slim-paths
      by (auto simp: slim-paths-def rev-path-choose-iapath intro: rev-image-eqI)
    moreover
    from ‹e ∈ set p› have (snd e, fst e) ∈ set (rev-path p)
      by (induct p) auto
    ultimately have (snd e, fst e) ∈ parcs slim
     by (auto simp: slim-simps slim-arcs-def) }
  then show symmetric slim
    unfolding symmetric-conv by simp (metis fst-conv snd-conv)
qed


lemma (in pair-pseudo-graph) pair-graph-slim: pair-graph slim
proof −
  interpret slim: pair-bidirected-digraph slim by (rule pair-bidirected-digraph-slim)
  show ?thesis
  proof
    show finite (pverts slim)
      using verts-slim-in-G finite-verts by (rule finite-subset)
    show finite (parcs slim)
      using arcs-slim-in-G finite-arcs by (rule finite-subset)
  qed
qed

lemma subgraph-slim: subgraph slim G
proof (rule subgraphI)
  interpret H: pair-bidirected-digraph slim
    by (rule pair-bidirected-digraph-slim) intro-locales

  show verts slim ⊆ verts G arcs slim ⊆ arcs G
    by (auto simp: verts-slim-in-G arcs-slim-in-G)
  show compatible G slim ..
  show wf-digraph slim wf-digraph G
    by unfold-locales
qed

lemma giapath-if-slim-giapath:
  assumes pre-digraph.gen-iapath slim (verts3 G) u p v
  shows gen-iapath (verts3 G) u p v
using assms verts-slim-in-G arcs-slim-in-G
by (auto simp: pre-digraph.gen-iapath-def pre-digraph.apath-def pre-digraph.awalk-def
  inner-verts-with-proj-def)

lemma slim-giapath-if-giapath:
assumes gen-iapath (verts3 G) u p v
  shows ∃ p. pre-digraph.gen-iapath slim (verts3 G) u p v (is ∃ p. ?P p)
```

proof
from *assms* **have** *choose-arcs*: $\bigwedge e.\ e \in set\ (choose\text{-}iapath\ u\ v) \Longrightarrow e \in parcs\ slim$
by (*fastforce simp*: *slim-simps slim-arcs-def slim-paths-def gen-contr-graph-def*)
moreover
from *assms* **have** *choose*: *iapath u* (*choose-iapath u v*) *v*
by (*intro choose-iapath*) (*auto simp*: *gen-iapath-def*)
ultimately show *?P* (*choose-iapath u v*)
by (*auto simp*: *pre-digraph.gen-iapath-def pre-digraph.apath-def pre-digraph.awalk-def inner-verts-with-proj-def*)
qed

lemma *contr-graph-slim-eq*:
gen-contr-graph slim (*verts3 G*) = *contr-graph G*
using *giapath-if-slim-giapath slim-giapath-if-giapath* **by** (*fastforce simp*: *gen-contr-graph-def*)

end

context *pair-pseudo-graph* **begin**

lemma *verts3-slim-in-verts3*:
assumes *v* ∈ *verts3 slim* **shows** *v* ∈ *verts3 G*
proof −
from *assms* **have** *2* < *in-degree slim v* **by** (*auto simp*: *verts3-def*)
also have ... ≤ *in-degree G v* **using** *subgraph-slim* **by** (*rule subgraph-in-degree*)
finally show *?thesis* **using** *assms subgraph-slim* **by** (*fastforce simp*: *verts3-def*)
qed

lemma *slim-is-slim*:
pair-pre-digraph.is-slim slim (*verts3 G*)
proof (*unfold pair-pre-digraph.is-slim-def*, *safe*)
interpret *S*: *pair-graph slim* **by** (*rule pair-graph-slim*)
{ fix *v* **assume** *v* ∈ *pverts slim v* ∉ *verts3 G*
then have *in-degree G v* ≤ *2*
using *verts-slim-in-G* **by** (*auto simp*: *verts3-def*)
then show *in-degree slim v* ≤ *2*
using *subgraph-in-degree*[*OF subgraph-slim*, *of v*] **by** *fastforce*
next
fix *w* **assume** *w* ∈ *pverts slim w* ∉ *verts3 G*
then obtain *u p v* **where** *upv*: (*u*, *p*, *v*) ∈ *slim-paths w* ∈ *set* (*awalk-verts u p*)
by (*auto simp*: *slim-simps slim-verts-def*)
moreover
then have *S.gen-iapath* (*verts3 G*) *u p v*
using *slim-paths-in-slimG* **by** *auto*
ultimately
show ∃ *x q y*. *S.gen-iapath* (*verts3 G*) *x q y*
∧ *w* ∈ *set* (*awalk-verts x q*)
by *auto*
next

```
    fix u v assume (u,v) ∈ parcs slim
    then obtain x p y where (x, p, y) ∈ slim-paths (u,v) ∈ set p
      by (auto simp: slim-simps slim-arcs-def)
    then have S.gen-iapath (verts3 G) x p y ∧ (u,v) ∈ set p
      using slim-paths-in-slimG by auto
    then show ∃ x p y. S.gen-iapath (verts3 G) x p y ∧ (u,v) ∈ set p
      by blast
  next
    fix u v assume (u,v) ∈ parcs slim fst (u,v) = snd (u,v)
    then show False by (auto simp: S.no-loops')
  next
    fix u v p q
    assume paths: S.gen-iapath (verts3 G) u p v
          S.gen-iapath (verts3 G) u q v

    have V: verts3 slim ⊆ verts3 G verts3 G ⊆ pverts slim
      by (auto simp: verts3-slim-in-verts3)

    have p = [] ∨ q = [] ⟹ p = q using paths
      by (auto simp: S.gen-iapath-def dest: S.apath-ends)
    moreover
    { assume p ≠ [] q ≠ []
      { fix u p v assume p ≠ [] and path: S.gen-iapath (verts3 G) u p v
        then obtain e where e ∈ set p by (metis last-in-set)
          then have e ∈ parcs slim using path by (auto simp: S.gen-iapath-def
S.apath-def)
        then obtain x r y where (x,r,y) ∈ slim-paths e ∈ set r
          by (auto simp: slim-simps slim-arcs-def)
        then have S.gen-iapath (verts3 G) x r y by (metis slim-paths-in-slimG)
        with ‹e ∈ set r› ‹e ∈ set p› path have p = r
          by (auto intro: S.same-gen-iapath-by-common-arc[OF V])
       then have x = u y = v using path ‹S.gen-iapath (verts3 G) x r y› ‹p = r›
‹p ≠ []›
          by (auto simp: S.gen-iapath-def S.apath-def dest: S.awalk-ends)
         then have (u,p,v) ∈ slim-paths using ‹p = r› ‹(x,r,y) ∈ slim-paths› by
simp }
      note obt = this
      from ‹p ≠ []› ‹q ≠ []› paths have (u,p,v) ∈ slim-paths (u,q,v) ∈ slim-paths
        by (auto intro: obt)
      then have p = q by (auto simp: slim-paths-def)
    }
    ultimately show p = q by metis
  }
qed auto

end

context pair-sym-digraph begin
```

lemma
  assumes *p*: *gen-iapath* (*pverts G*) *u p v*
  shows *gen-iapath-triv-path*: *p* = [(*u*,*v*)]
    and *gen-iapath-triv-arc*: (*u*,*v*) ∈ *parcs G*
proof −
  have *set* (*inner-verts p*) = {}
  proof −
    have ∗: ⋀*A B* :: *'a set*. ⟦*A* ⊆ *B*; *A* ∩ *B* = {}⟧ ⟹ *A* = {} **by** *blast*
    have *set* (*inner-verts p*) = *set* (*awalk-verts u p*) − {*u*, *v*}
      using *p* **by** (*simp add*: *set-inner-verts gen-iapath-def*)
    also have ... ⊆ *pverts G*
      using *p* **unfolding** *gen-iapath-def apath-def awalk-conv* **by** *auto*
    finally show *?thesis*
      using *p* **by** (*rule-tac* ∗) (*auto simp*: *gen-iapath-def*)
  qed
  then have *inner-verts p* = [] **by** *simp*
  then show *p* = [(*u*,*v*)] **using** *p*
   by (*cases p*) (*auto simp*: *gen-iapath-def apath-def inner-verts-def split*: *if-split-asm*)
  then show (*u*,*v*) ∈ *parcs G*
    using *p* **by** (*auto simp*: *gen-iapath-def apath-def*)
qed

lemma *gen-contr-triv*:
  assumes *is-slim V pverts G* = *V* **shows** *gen-contr-graph G V* = *G*
proof −
  let *?gcg* = *gen-contr-graph G V*

  from *assms* **have** *pverts ?gcg* = *pverts G*
    by (*auto simp*: *gen-contr-graph-def is-slim-def*)
  moreover
  have *parcs ?gcg* = *parcs G*
  proof (*rule set-eqI*, *safe*)
    fix *u v* **assume** (*u*,*v*) ∈ *parcs ?gcg*
    then obtain *p* **where** *gen-iapath V u p v*
      by (*auto simp*: *gen-contr-graph-def*)
    then show (*u*,*v*) ∈ *parcs G*
      using *gen-iapath-triv-arc* ‹*pverts G* = *V*› **by** *auto*
  next
    fix *u v* **assume** (*u*,*v*) ∈ *parcs G*
    with *assms* **obtain** *x p y* **where** *path*: *gen-iapath V x p y* (*u*,*v*) ∈ *set p u* ≠ *v*
      by (*auto simp*: *is-slim-def*)
    with ‹*pverts G* = *V*› **have** *p* = [(*x*,*y*)] **by** (*intro gen-iapath-triv-path*) *auto*
    then show (*u*,*v*) ∈ *parcs ?gcg*
      using *path* **by** (*auto simp*: *gen-contr-graph-def*)
  qed
  ultimately
  show *?gcg* = *G* **by** *auto*
qed

lemma *is-slim-no-loops*:
assumes *is-slim V a* ∈ *arcs G* **shows** *fst a* ≠ *snd a*
using *assms* **by** (*auto simp*: *is-slim-def*)

end

17.8 Contraction Preserves Kuratowski-Subgraph-Property

lemma (**in** *pair-pseudo-graph*) *in-degree-contr*:
assumes *v* ∈ *V* **and** *V*: *verts3 G* ⊆ *V V* ⊆ *verts G*
shows *in-degree* (*gen-contr-graph G V*) *v* ≤ *in-degree G v*
proof −
have *fin*: *finite* {(*u*, *p*). *gen-iapath V u p v*}
proof −
have {(*u*, *p*). *gen-iapath V u p v*} ⊆ (λ(*u*,*p*,-). (*u*,*p*)) ‘ {(*u*,*p*,*v*). *apath u p v*}
by (*force simp*: *gen-iapath-def*)
with *apaths-finite-triple* **show** *?thesis* **by** (*rule finite-surj*)
qed

have *io-snd*: *inj-on snd* {(*u*,*p*). *gen-iapath V u p v*}
by (*rule inj-onI*) (*auto simp*: *gen-iapath-def apath-def dest*: *awalk-ends*)

have *io-last*: *inj-on last* {*p*. ∃ *u*. *gen-iapath V u p v*}
proof (*rule inj-onI*, *safe*)
fix *u1 u2 p1 p2*
assume *A*: *last p1* = *last p2* **and** *B*: *gen-iapath V u1 p1 v gen-iapath V u2 p2 v*
from *B* **have** *last p1* ∈ *set p1 last p2* ∈ *set p2* **by** (*auto simp*: *gen-iapath-def*)
with *A* **have** *last p1* ∈ *set p1 last p1* ∈ *set p2* **by** *simp-all*
with *V*[*simplified*] *B* **show** *p1* = *p2* **by** (*rule same-gen-iapath-by-common-arc*)
qed

have *in-degree* (*gen-contr-graph G V*) *v* = *card* ((λ(*u*,-). (*u*,*v*)) ‘ {(*u*,*p*). *gen-iapath V u p v*})
proof −
have *in-arcs* (*gen-contr-graph G V*) *v* = (λ(*u*,-). (*u*,*v*)) ‘ {(*u*,*p*). *gen-iapath V u p v*}
by (*auto simp*: *gen-contr-graph-def*)
then show *?thesis* **unfolding** *in-degree-def* **by** *simp*
qed
also have ... ≤ *card* {(*u*,*p*). *gen-iapath V u p v*}
using *fin* **by** (*rule card-image-le*)
also have ... = *card* (*snd* ‘ {(*u*,*p*). *gen-iapath V u p v*})
using *io-snd* **by** (*rule card-image*[*symmetric*])
also have *snd* ‘ {(*u*,*p*). *gen-iapath V u p v*} = {*p*. ∃ *u*. *gen-iapath V u p v*}
by (*auto intro*: *rev-image-eqI*)
also have *card* ... = *card* (*last* ‘ ...)
using *io-last* **by** (*rule card-image*[*symmetric*])
also have ... ≤ *in-degree G v*

```
    unfolding in-degree-def
  proof (rule card-mono)
    show last ` {p. ∃ u. gen-iapath V u p v} ⊆ in-arcs G v
    proof −
      have ⋀u p. awalk u p v ⟹ p ≠ [] ⟹ last p ∈ parcs G
        by (auto simp: awalk-def)
      moreover
      { fix u p assume awalk u p v p ≠ []
          then have snd (last p) = v by (induct p arbitrary: u) (auto simp:
awalk-simps) }
      ultimately
     show ?thesis unfolding in-arcs-def by (auto simp: gen-iapath-def apath-def)
    qed
  qed auto
  finally show ?thesis .
qed

lemma (in pair-graph) contracted-no-degree2-simp:
  assumes subd: subdivision-pair G H
  assumes two-less-deg2: verts3 G = pverts G
  shows contr-graph H = G
  using subd
proof (induct rule: subdivision-pair-induct)
  case base

  { fix e assume e ∈ parcs G
    then have gen-iapath (pverts G) (fst e) [(fst e, snd e)] (snd e) e ∈ set [(fst e,
snd e)]
      using no-loops[of (fst e, snd e)] by (auto simp: gen-iapath-def apath-simps )
    then have ∃ u p v. gen-iapath (pverts G) u p v ∧ e ∈ set p by blast }
  moreover
  { fix u p v assume gen-iapath (pverts G) u p v
    from ‹gen-iapath - u p v› have p = [(u,v)]
      unfolding gen-iapath-def apath-def
      by safe (cases p, case-tac [2] list, auto simp: awalk-simps inner-verts-def) }
  ultimately have is-slim (verts3 G) unfolding is-slim-def two-less-deg2
    by (blast dest: no-loops-in-iapath)
  then show ?case by (simp add: gen-contr-triv two-less-deg2)
next
  case (divide e w H)
  let ?sH = subdivide H e w
  from ‹subdivision-pair G H› interpret H: pair-bidirected-digraph H
    by (rule bidirected-digraphI-subdivision)
 from divide(1,2) interpret S: pair-sym-digraph ?sH by (rule H.pair-sym-digraph-subdivide)
  obtain u v where e-conv:e = (u,v) by (cases e) auto
  have contr-graph ?sH = contr-graph H
  proof −
    have V-cond: verts3 H ⊆ pverts H by (auto simp: verts3-def)
    have verts3 H = verts3 ?sH
```

```
    using divide by (simp add: H.verts3-subdivide)
   then have v: pverts (contr-graph ?sH) = pverts (contr-graph H)
    by (auto simp: gen-contr-graph-def)
   moreover
   then have parcs (contr-graph ?sH) = parcs (contr-graph H)
    unfolding gen-contr-graph-def
    by (auto dest: H.gen-iapath-co-path[OF divide(1,2) V-cond]
        H.gen-iapath-sd-path[OF divide(1,2) V-cond])
   ultimately show ?thesis by auto
  qed
  then show ?case using divide by simp
qed


lemma verts3-K33:
  assumes K_{3,3} (with-proj G)
  shows verts3 G = verts G
proof −
  { fix v assume v ∈ pverts G
   from assms obtain U V where cards: card U = 3 card V=3
    and UV: U ∩ V = {} pverts G = U ∪ V parcs G = U × V ∪ V × U
    unfolding complete-bipartite-digraph-pair-def by blast
   have 2 < in-degree G v
   proof (cases v ∈ U)
    case True
    then have in-arcs G v = V × {v} using UV by fastforce
   then show ?thesis using cards by (auto simp: card-cartesian-product in-degree-def)
   next
    case False
    then have in-arcs G v = U × {v} using ‹v ∈ -› UV by fastforce
   then show ?thesis using cards by (auto simp: card-cartesian-product in-degree-def)
   qed }
  then show ?thesis by (auto simp: verts3-def)
qed


lemma verts3-K5:
  assumes K_5 (with-proj G)
  shows verts3 G = verts G
proof −
  interpret pgG: pair-graph G using assms by (rule pair-graphI-complete)
  { fix v assume v ∈ pverts G
   have 2 < (4 :: nat) by simp
   also have 4 = card (pverts G − {v})
    using assms ‹v ∈ pverts G› unfolding complete-digraph-def by auto
   also have pverts G − {v} = {u ∈ pverts G. u ≠ v}
    by auto
   also have card ... = card ({u ∈ pverts G. u ≠ v} × {v}) (is - = card ?A)
    by auto
```

also have *?A = in-arcs G v*
using *assms ‹v ∈ pverts G›* **unfolding** *complete-digraph-def* **by** *safe auto*
also have *card ... = in-degree G v*
unfolding *in-degree-def* **..**
finally have *2 < in-degree G v* **. }**
then show *?thesis* **unfolding** *verts3-def* **by** *auto*
qed

lemma *K33-contractedI*:
assumes *subd*: *subdivision-pair G H*
assumes *k33*: $K_{3,3}$ *G*
shows $K_{3,3}$ (*contr-graph H*)
proof −
interpret *pgG*: *pair-graph G* **using** *k33* **by** (*rule pair-graphI-complete-bipartite*)
show *?thesis*
using *assms* **by** (*auto simp*: *pgG.contracted-no-degree2-simp verts3-K33*)
qed

lemma *K5-contractedI*:
assumes *subd*: *subdivision-pair G H*
assumes *k5*: K_5 *G*
shows K_5 (*contr-graph H*)
proof −
interpret *pgG*: *pair-graph G* **using** *k5* **by** (*rule pair-graphI-complete*)
show *?thesis*
using *assms* **by** (*auto simp add*: *pgG.contracted-no-degree2-simp verts3-K5*)
qed

17.9 Final proof

context *pair-sym-digraph* **begin**

lemma *gcg-subdivide-eq*:
assumes *mem*: *e ∈ parcs G w ∉ pverts G*
assumes *V*: *V ⊆ pverts G*
shows *gen-contr-graph* (*subdivide G e w*) *V* = *gen-contr-graph G V*
proof −
interpret *sdG*: *pair-sym-digraph subdivide G e w*
using *mem* **by** (*rule pair-sym-digraph-subdivide*)
{ fix *u p v* **assume** *sdG.gen-iapath V u p v*
have *gen-iapath V u* (*co-path e w p*) *v*
using *mem V ‹sdG.gen-iapath V u p v›* **by** (*rule gen-iapath-co-path*)
then have ∃ *p*. *gen-iapath V u p v* **..**
} **note** *A* = *this*
moreover
{ fix *u p v* **assume** *gen-iapath V u p v*
have *sdG.gen-iapath V u* (*sd-path e w p*) *v*
using *mem V ‹gen-iapath V u p v›* **by** (*rule gen-iapath-sd-path*)

```
    then have ∃ p. sdG.gen-iapath V u p v ..
  } note B = this
  ultimately show ?thesis using assms by (auto simp: gen-contr-graph-def)
qed

lemma co-path-append:
  assumes [last p1, hd p2] ∉ {[(fst e,w),(w,snd e)], [(snd e,w),(w,fst e)]}
  shows co-path e w (p1 @ p2) = co-path e w p1 @ co-path e w p2
using assms
proof (induct p1 rule: co-path-induct)
  case single then show ?case by (cases p2) auto
next
  case (co e1 e2 es) then show ?case by (cases es) auto
next
  case (corev e1 e2 es) then show ?case by (cases es) auto
qed auto

lemma exists-co-path-decomp1:
  assumes mem: e ∈ parcs G w ∉ pverts G
  assumes p: pre-digraph.apath (subdivide G e w) u p v (fst e, w) ∈ set p w ≠ v
  shows ∃ p1 p2. p = p1 @ (fst e, w) # (w, snd e) # p2
proof −
  let ?sdG = subdivide G e w
  interpret sdG: pair-sym-digraph ?sdG
    using mem by (rule pair-sym-digraph-subdivide)
  obtain p1 p2 z where p-decomp: p = p1 @ (fst e, w) # (w, z) # p2 fst e ≠ z
w ≠ z
    by atomize-elim (rule sdG.apath-succ-decomp[OF p])
  then have (fst e,w) ∈ parcs ?sdG (w, z) ∈ parcs ?sdG
    using p by (auto simp: sdG.apath-def)
  with ‹fst e ≠ z› have z = snd e
    using mem by (cases e) (auto simp: wellformed')
  with p-decomp show ?thesis by fast
qed

lemma is-slim-if-subdivide:
  assumes pair-pre-digraph.is-slim (subdivide G e w) V
  assumes mem1: e ∈ parcs G w ∉ pverts G and mem2: w ∉ V
  shows is-slim V
proof −
  let ?sdG = subdivide G e w
  interpret sdG: pair-sym-digraph subdivide G e w
    using mem1 by (rule pair-sym-digraph-subdivide)
  obtain u v where e = (u,v) by (cases e) auto
  with mem1 have u ∈ pverts G v ∈ pverts G by (auto simp: wellformed')
  with mem1  have u ≠ w v ≠ w by auto

  let ?w-parcs = {(u,w), (v,w), (w,u), (w, v)}
  have sdg-new-parcs: ?w-parcs ⊆ parcs ?sdG
```

```
    using ‹e = (u,v)› by auto
  have sdg-no-parcs: (u,v) ∉ parcs ?sdG (v,u) ∉ parcs ?sdG
    using ‹e = (u,v)› ‹u ≠ w› ‹v ≠ w› by auto

  { fix z assume A: z ∈ pverts G
    have in-degree ?sdG z = in-degree G z
    proof −
      { assume z ≠ u z ≠ v
        then have in-arcs ?sdG z = in-arcs G z
          using ‹e = (u,v)› mem1 A by auto
       then have in-degree ?sdG z = in-degree G z by (simp add: in-degree-def) }
      moreover
      { assume z = u
        then have in-arcs G z = in-arcs ?sdG z ∪ {(v,u)} − {(w,u)}
         using ‹e = (u,v)› mem1 by (auto simp: intro: arcs-symmetric wellformed')
        moreover
        have card (in-arcs ?sdG z ∪ {(v,u)} − {(w,u)}) = card (in-arcs ?sdG z)
          using sdg-new-parcs sdg-no-parcs ‹z = u› by (cases finite (in-arcs ?sdG
z)) (auto simp: in-arcs-def)
      ultimately have in-degree ?sdG z= in-degree G z by (simp add: in-degree-def)
}
      moreover
      { assume z = v
        then have in-arcs G z = in-arcs ?sdG z ∪ {(u,v)} − {(w,v)}
          using ‹e = (u,v)› mem1 A by (auto simp: wellformed')
        moreover
        have card (in-arcs ?sdG z ∪ {(u,v)} − {(w,v)}) = card (in-arcs ?sdG z)
          using sdg-new-parcs sdg-no-parcs ‹z = v› by (cases finite (in-arcs ?sdG
z)) (auto simp: in-arcs-def)
      ultimately have in-degree ?sdG z= in-degree G z by (simp add: in-degree-def)
}
      ultimately show ?thesis by metis
    qed }
  note in-degree-same = this

  have V-G: V ⊆ pverts G verts3 G ⊆ V
  proof −
    have V ⊆ pverts ?sdG pverts ?sdG = pverts G ∪ {w} verts3 ?sdG ⊆ V verts3
G ⊆ verts3 ?sdG
      using ‹sdG.is-slim V› ‹e = (u,v)› in-degree-same mem1
      unfolding sdG.is-slim-def verts3-def
      by (fast, simp, fastforce, force)
    then show V ⊆ pverts G verts3 G ⊆ V using ‹w ∉ V› by auto
  qed

  have pverts: ∀ v∈pverts G. v ∈ V ∨ in-degree G v ≤ 2 ∧ (∃ x p y. gen-iapath V
x p y ∧ v ∈ set (awalk-verts x p))
  proof −
    { fix z assume A: z ∈ pverts G z ∉ V
```

```
    have z ∈ pverts ?sdG using ‹e = (u,v)› A mem1 by auto
    then have in-degree ?sdG z ≤ 2
      using ‹sdG.is-slim V› A by (auto simp: sdG.is-slim-def)
    with in-degree-same[OF ‹z ∈ pverts G›] have idg: in-degree G z ≤ 2 by auto

    from A have z ∈ pverts ?sdG z ∉ V using ‹e = (u,v)› mem1 by auto
    then obtain x' q y' where sdG.gen-iapath V x' q y' z ∈ set (sdG.awalk-verts
x' q)
      using ‹sdG.is-slim V› unfolding sdG.is-slim-def by metis
    then have gen-iapath V x' (co-path e w q) y' z ∈ set (awalk-verts x' (co-path
e w q))
    using A mem1 V-G by (auto simp: set-awalk-verts-co-path' intro: gen-iapath-co-path)
     with idg have in-degree G z ≤ 2 ∧ (∃ x p y. gen-iapath V x p y ∧ z ∈ set
(awalk-verts x p))
      by metis }
   then show ?thesis by auto
  qed

  have parcs: ∀ e∈parcs G. fst e ≠ snd e ∧ (∃ x p y. gen-iapath V x p y ∧ e ∈ set
p)
  proof (intro ballI conjI)
    fix e' assume e' ∈ parcs G

    show (∃ x p y. gen-iapath V x p y ∧ e' ∈ set p)
    proof (cases e' ∈ parcs ?sdG)
      case True
      then obtain x p y where sdG.gen-iapath V x p y e' ∈ set p
        using ‹sdG.is-slim V› by (auto simp: sdG.is-slim-def)
      with ‹e ∈ parcs G› ‹w ∉ pverts G› V-G have gen-iapath V x (co-path e w p)
y
        by (auto intro: gen-iapath-co-path)

       from ‹e' ∈ parcs G› have e' ∉ ?w-parcs using mem1 by (auto simp:
wellformed')
      with ‹e' ∈ set p› have e' ∈ set (co-path e w p)
        by (induct p rule: co-path-induct) (force simp: ‹e = (u,v)›)+
      then show ∃ x p y. gen-iapath V x p y ∧ e' ∈ set p
        using ‹gen-iapath V x (co-path e w p) y› by fast
    next
      assume e' ∉ parcs ?sdG
      define a b where a = fst e' and b = snd e'
      then have e' = (a,b) and ab: (a,b) = (u,v) ∨ (a,b) = (v,u)
        using ‹e' ∈ parcs G› ‹e' ∉ parcs ?sdG› ‹e = (u,v)› mem1 by auto
      obtain x p y where sdG.gen-iapath V x p y (a,w) ∈ set p
        using ‹sdG.is-slim V› sdg-new-parcs ab by (auto simp: sdG.is-slim-def)
      with ‹e ∈ parcs G› ‹w ∉ pverts G› V-G have gen-iapath V x (co-path e w p)
y
        by (auto intro: gen-iapath-co-path)
```

```
    have (a,b) ∈ parcs G subdivide G (a,b) w = subdivide G e w
      using mem1 ‹e = (u,v)› ‹e' = (a,b)› ab
      by (auto intro: arcs-symmetric simp: subdivide.simps)
    then have pre-digraph.apath (subdivide G (a,b) w) x p y w ≠ y
      using mem2 ‹sdG.gen-iapath V x p y› by (auto simp: sdG.gen-iapath-def)
    then obtain p1 p2 where p: p = p1 @ (a,w) # (w,b) # p2
     using exists-co-path-decomp1 ‹(a,b) ∈ parcs G› ‹w ∉ pverts G› ‹(a,w) ∈ set
p› ‹w ≠ y›
      by atomize-elim auto
    moreover
    from p have co-path e w ((a,w) # (w,b) # p2) = (a,b) # co-path e w p2
      unfolding ‹e = (u,v)› using ab by auto
    ultimately
    have (a,b) ∈ set (co-path e w p)
      unfolding ‹e = (u,v)› using ab ‹u ≠ w› ‹v ≠ w›
      by (induct p rule: co-path-induct) (auto simp: co-path-append)
    then show ?thesis
      using ‹gen-iapath V x (co-path e w p) y› ‹e' = (a,b)› by fast
  qed
  then show fst e' ≠ snd e' by (blast dest: no-loops-in-iapath)
 qed

 have unique: ∀ u v p q. (gen-iapath V u p v ∧ gen-iapath V u q v) ⟶ p = q
 proof safe
  fix x y p q assume A: gen-iapath V x p y gen-iapath V x q y
  then have set p ⊆ parcs G set q ⊆ parcs G
    by (auto simp: gen-iapath-def apath-def)
  then have w-p: (u,w) ∉ set p (v,w) ∉ set p and w-q: (u,w) ∉ set q (v,w) ∉ set
q
    using mem1 by (auto simp: wellformed')

  from A have sdG.gen-iapath V x (sd-path e w p) y sdG.gen-iapath V x (sd-path
e w q) y
    using mem1 V-G by (auto intro: gen-iapath-sd-path)
  then have sd-path e w p = sd-path e w q
    using ‹sdG.is-slim V› unfolding sdG.is-slim-def by metis
  then have co-path e w (sd-path e w p) = co-path e w (sd-path e w q) by simp
  then show p = q using w-p w-q ‹e = (u,v)› by (simp add: co-sd-id)
 qed

 from pverts parcs V-G unique show ?thesis by (auto simp: is-slim-def)
qed

end

context pair-pseudo-graph begin

lemma subdivision-gen-contr:
 assumes is-slim V
```

```
  shows subdivision-pair (gen-contr-graph G V) G
using assms using pair-pseudo-graph
proof (induct card (pverts G − V) arbitrary: G)
  case 0
  interpret G: pair-pseudo-graph G by fact
  have pair-bidirected-digraph G
    using G.pair-sym-arcs 0 by unfold-locales (auto simp: G.is-slim-def)
  with 0 show ?case
    by (auto intro: subdivision-pair-intros simp: G.gen-contr-triv G.is-slim-def)
next
  case (Suc n)
  interpret G: pair-pseudo-graph G by fact

  from ‹Suc n = card (pverts G − V)›
  have pverts G − V ≠ {}
   by (metis Nat.diff-le-self Suc-n-not-le-n card-Diff-subset-Int diff-Suc-Suc empty-Diff
finite.emptyI inf-bot-left)
  then obtain w where w ∈ pverts G − V by auto
  then obtain x q y where q: G.gen-iapath V x q y w ∈ set (G.awalk-verts x q)
in-degree G w ≤ 2
    using ‹G.is-slim V› by (auto simp: G.is-slim-def)
  then have w ≠ x w ≠ y w ∉ V using ‹w ∈ pverts G − V› by (auto simp:
G.gen-iapath-def)
  then obtain e where e ∈ set q snd e = w
    using ‹w ∈ pverts G − V› q
    unfolding G.gen-iapath-def G.apath-def G.awalk-conv
    by (auto simp: G.awalk-verts-conv')
  moreover define u where u = fst e
  ultimately obtain q1 q2 v where q-decomp: q = q1 @ (u, w) # (w, v) # q2 u
≠ v w ≠ v
   using q ‹w ≠ y› unfolding G.gen-iapath-def by atomize-elim (rule G.apath-succ-decomp,
auto)
  with q have qi-walks: G.awalk x q1 u G.awalk v q2 y
    by (auto simp: G.gen-iapath-def G.apath-def G.awalk-Cons-iff)

  from q q-decomp have uvw-arcs1: (u,w) ∈ parcs G (w,v) ∈ parcs G
    by (auto simp: G.gen-iapath-def G.apath-def)
  then have uvw-arcs2: (w,u) ∈ parcs G (v,w) ∈ parcs G
    by (blast intro: G.arcs-symmetric)+

  have u ≠ w v ≠ w using q-decomp q
    by (auto simp: G.gen-iapath-def G.apath-append-iff G.apath-simps)

  have in-arcs: in-arcs G w = {(u,w), (v,w)}
  proof −
    have {(u,w), (v,w)} ⊆ in-arcs G w
      using uvw-arcs1 uvw-arcs2 by auto
    moreover note ‹in-degree G w ≤ 2›
    moreover have card {(u,w), (v,w)} = 2 using ‹u ≠ v› by auto
```

```
    ultimately
    show ?thesis by − (rule card-seteq[symmetric], auto simp: in-degree-def)
  qed
  have out-arcs: out-arcs G w ⊆ {(w,u), (w,v)} (is ?L ⊆ ?R)
  proof
    fix e assume e ∈ out-arcs G w
    then have (snd e, fst e) ∈ in-arcs G w
      by (auto intro: G.arcs-symmetric)
    then show e ∈ {(w, u), (w, v)} using in-arcs by auto
  qed

  have (u,v) ∉ parcs G
  proof
    assume (u,v) ∈ parcs G
    have G.gen-iapath V x (q1 @ (u,v) # q2) y
    proof −
      have awalk': G.awalk x (q1 @ (u,v) # q2) y
        using qi-walks ‹(u,v) ∈ parcs G›
        by (auto simp: G.awalk-simps)

    have G.awalk x q y using ‹G.gen-iapath V x q y› by (auto simp: G.gen-iapath-def
G.apath-def)

      have distinct (G.awalk-verts x (q1 @ (u,v) # q2))
        using awalk' ‹G.gen-iapath V x q y› unfolding q-decomp
        by (auto simp: G.gen-iapath-def G.apath-def G.awalk-verts-append)
      moreover
      have set (G.inner-verts (q1 @ (u,v) # q2)) ⊆ set (G.inner-verts q)
        using awalk' ‹G.awalk x q y› unfolding q-decomp
      by (auto simp: butlast-append G.inner-verts-conv[of - x] G.awalk-verts-append
          intro: in-set-butlast-appendI)
      then have set (G.inner-verts (q1 @ (u,v) # q2)) ∩ V = {}
        using ‹G.gen-iapath V x q y› by (auto simp: G.gen-iapath-def)
     ultimately show ?thesis using awalk' ‹G.gen-iapath V x q y› by (simp add:
G.gen-iapath-def G.apath-def)
    qed
    then have (q1 @ (u,v) # q2) = q
     using ‹G.gen-iapath V x q y› ‹G.is-slim V› unfolding G.is-slim-def by metis
    then show False unfolding q-decomp by simp
  qed
  then have (v,u) ∉ parcs G by (auto intro: G.arcs-symmetric)

  define G' where G' = ⦇pverts = pverts G − {w},
      parcs = {(u,v), (v,u)} ∪ (parcs G − {(u,w), (w,u), (v,w), (w,v)})⦈

  have mem-G': (u,v) ∈ parcs G' w ∉ pverts G' by (auto simp: G'-def)

  interpret pd-G': pair-fin-digraph G'
  proof
```

fix e **assume** A: $e \in parcs\ G'$
have $e \in parcs\ G \wedge e \neq (u,\ w) \wedge e \neq (w,\ u) \wedge e \neq (v,\ w) \wedge e \neq (w,\ v) \Longrightarrow fst\ e \neq w$
$e \in parcs\ G \wedge e \neq (u,\ w) \wedge e \neq (w,\ u) \wedge e \neq (v,\ w) \wedge e \neq (w,\ v) \Longrightarrow snd\ e \neq w$
using *out-arcs in-arcs* **by** *auto*
with A *uvw-arcs1* **show** *fst* $e \in pverts\ G'$ *snd* $e \in pverts\ G'$
using ‹$u \neq w$› ‹$v \neq w$› **by** (*auto simp*: G'*-def G.wellformed'*)
next
qed (*auto simp*: G'*-def arc-to-ends-def*)

interpret *spd-G'*: *pair-pseudo-graph* G'
proof (*unfold-locales*, *simp add*: *symmetric-def*)
have *sym* $\{(u,v),\ (v,u)\}$ *sym* (*parcs* G) *sym* $\{(u,\ w),\ (w,\ u),\ (v,\ w),\ (w,\ v)\}$
using *G.sym-arcs* **by** (*auto simp*: *symmetric-def sym-def*)
then have *sym* $(\{(u,v),\ (v,u)\} \cup (parcs\ G - \{(u,w),\ (w,u),\ (v,w),\ (w,v)\}))$
by (*intro sym-Un*) (*auto simp*: *sym-diff*)
then show *sym* (*parcs* G') **unfolding** G'*-def* **by** *simp*
qed

have *card-G'*: $n = card\ (pverts\ G' - V)$
proof −
have $pverts\ G - V = insert\ w\ (pverts\ G' - V)$
using ‹$w \in pverts\ G - V$› **by** (*auto simp*: G'*-def*)
then show *?thesis* **using** ‹$Suc\ n = card\ (pverts\ G - V)$› *mem-G'* **by** *simp*
qed

have *G-is-sd*: $G = subdivide\ G'\ (u,v)\ w$ (**is** - = *?sdG'*)
using ‹$w \in pverts\ G - V$› ‹$(u,v) \notin parcs\ G$› ‹$(v,u) \notin parcs\ G$› *uvw-arcs1 uvw-arcs2*
by (*intro pair-pre-digraph.equality*) (*auto simp*: G'*-def*)

have *gcg-sd*: *gen-contr-graph* (*subdivide* G' (u,v) w) V = *gen-contr-graph* G' V
proof −
have $V \subseteq pverts\ G$
using ‹*G.is-slim* V› **by** (*auto simp*: *G.is-slim-def verts3-def*)
moreover
have *verts3* G' = *verts3* G
by (*simp only*: *G-is-sd spd-G'.verts3-subdivide*[*OF* ‹$(u,v) \in parcs\ G'$› ‹$w \notin pverts\ G'$›])
ultimately
have V: $V \subseteq pverts\ G'$
using ‹$w \in pverts\ G - V$› **by** (*auto simp*: G'*-def*)
with *mem-G'* **show** *?thesis* **by** (*rule spd-G'.gcg-subdivide-eq*)
qed

have *is-slim-G'*: *pd-G'.is-slim* V **using** ‹*G.is-slim* V› *mem-G'* ‹$w \notin V$›
unfolding *G-is-sd* **by** (*rule spd-G'.is-slim-if-subdivide*)
with *mem-G'* **have** *subdivision-pair* (*gen-contr-graph* G' V) (*subdivide* G' $(u,\ v)$

w)
by (*intro Suc card-G' subdivision-pair-intros*) *auto*
then show *?case* **by** (*simp add*: *gcg-sd G-is-sd*)
qed

lemma *contr-is-subgraph-subdivision*:
shows $\exists H.$ *subgraph* (*with-proj H*) *G* $\wedge$ *subdivision-pair* (*contr-graph G*) *H*
proof −
interpret *sG*: *pair-graph slim* **by** (*rule pair-graph-slim*)

have *subdivision-pair* (*gen-contr-graph slim* (*verts3 G*)) *slim*
by (*rule sG.subdivision-gen-contr*) (*rule slim-is-slim*)
then show *?thesis* **unfolding** *contr-graph-slim-eq* **by** (*blast intro*: *subgraph-slim*)
qed

theorem *kuratowski-contr*:
fixes *K* :: *'a pair-pre-digraph*
assumes *subgraph-K*: *subgraph K G*
assumes *spd-K*: *pair-pseudo-graph K*
assumes *kuratowski*: $K_{3,3}$ (*contr-graph K*) $\vee$ K_5 (*contr-graph K*)
shows ¬*kuratowski-planar G*
proof −
interpret *spd-K*: *pair-pseudo-graph K* **by** (*fact spd-K*)
obtain *H* **where** *subgraph-H*: *subgraph* (*with-proj H*) *K*
and *subdiv-H*:*subdivision-pair* (*contr-graph K*) *H*
by *atomize-elim* (*rule spd-K.contr-is-subgraph-subdivision*)
have *grI*: $\bigwedge K.$ ($K_{3,3}$ *K* $\vee$ K_5 *K*) $\Longrightarrow$ *graph K*
by (*auto simp*: *complete-digraph-def complete-bipartite-digraph-def*)
from *subdiv-H* **and** *kuratowski*
have $\exists K.$ *subdivision-pair K H* $\wedge$ ($K_{3,3}$ *K* $\vee$ K_5 *K*) **by** *blast*
then have $\exists K$ *rev-K rev-H. subdivision* (*K*, *rev-K*) (*H*, *rev-H*) $\wedge$ ($K_{3,3}$ *K* $\vee$ K_5 *K*)
by (*auto intro*: *grI pair-graphI-graph*)
then show *?thesis* **using** *subgraph-H subgraph-K*
unfolding *kuratowski-planar-def* **by** (*auto intro*: *subgraph-trans*)
qed

theorem *certificate-characterization*:
defines *kuratowski* ≡ λG :: *'a pair-pre-digraph.* $K_{3,3}$ *G* $\vee$ K_5 *G*
shows *kuratowski* (*contr-graph G*)
$\longleftrightarrow$ ($\exists H.$ *kuratowski H* $\wedge$ *subdivision-pair H slim* $\wedge$ *verts3 G* = *verts3 slim*)
(**is** *?L* $\longleftrightarrow$ *?R*)
proof
assume *?L*
interpret *S*: *pair-graph slim* **by** (*rule pair-graph-slim*)
have *subdivision-pair* (*contr-graph G*) *slim*
proof −
have ∗: *S.is-slim* (*verts3 G*) **by** (*rule slim-is-slim*)
show *?thesis* **using** *contr-graph-slim-eq S.subdivision-gen-contr*[*OF* ∗] **by** *auto*

```
  qed
  moreover
  have verts3 slim = verts3 G (is ?l = ?r)
  proof safe
    fix v assume v ∈ ?l then show v ∈ ?r
      using verts-slim-in-G verts3-slim-in-verts3 by auto
  next
    fix v assume v ∈ ?r
    have v ∈ verts3 (contr-graph G)
    proof −
      have v ∈ verts (contr-graph G)
        using ‹v ∈ ?r› by (auto simp: verts3-def gen-contr-graph-def)
      then show ?thesis
       using ‹?L› unfolding kuratowski-def by (auto simp: verts3-K33 verts3-K5)
    qed
   then have v ∈ verts3 (gen-contr-graph slim (verts3 G)) unfolding contr-graph-slim-eq
.
    then have 2 < in-degree (gen-contr-graph slim (verts3 G)) v
      unfolding verts3-def by auto
    also have ... ≤ in-degree slim v
      using ‹v ∈ ?r› verts3-slim-in-verts3 by (auto intro: S.in-degree-contr)
    finally show v ∈ verts3 slim
      using verts3-in-slim-G ‹v ∈ ?r› unfolding verts3-def by auto
  qed
  ultimately show ?R using ‹?L› by auto
next
  assume ?R
  then have kuratowski (gen-contr-graph slim (verts3 G))
    unfolding kuratowski-def
    by (auto intro: K33-contractedI K5-contractedI)
  then show ?L unfolding contr-graph-slim-eq .
qed

definition (in pair-pre-digraph) certify :: 'a pair-pre-digraph ⇒ bool where
  certify cert ≡ let C = contr-graph cert in subgraph cert G ∧ (K_{3,3} C ∨ K_5 C)

theorem certify-complete:
  assumes pair-pseudo-graph cert
  assumes subgraph cert G
  assumes ∃ H. subdivision-pair H cert ∧ (K_{3,3} H ∨ K_5 H)
  shows certify cert
  unfolding certify-def
  using assms by (auto simp: Let-def intro: K33-contractedI K5-contractedI)

theorem certify-sound:
  assumes pair-pseudo-graph cert
  assumes certify cert
  shows ¬kuratowski-planar G
  using assms by (intro kuratowski-contr) (auto simp: certify-def Let-def)
```

theorem *certify-characterization*:
assumes *pair-pseudo-graph cert*
shows *certify cert* $\longleftrightarrow$ *subgraph cert G* $\wedge$ *verts3 cert* = *verts3* (*pair-pre-digraph.slim cert*)
$\wedge$($\exists$ *H*. ($K_{3,3}$ (*with-proj H*) $\vee$ K_5 *H*) $\wedge$ *subdivision-pair H* (*pair-pre-digraph.slim cert*))
(**is** *?L* $\longleftrightarrow$ *?R*)
by (*auto simp only*: *simp-thms certify-def Let-def pair-pseudo-graph.certificate-characterization*[*OF assms*])

end

end

theory *Weighted-Graph*
imports
Digraph
Arc-Walk
Complex-Main
begin

18 Weighted Graphs

type-synonym *'b weight-fun* = *'b* $\Rightarrow$ *real*

context *wf-digraph* **begin**

definition *awalk-cost* :: *'b weight-fun* $\Rightarrow$ *'b awalk* $\Rightarrow$ *real* **where**
awalk-cost f es = *sum-list* (*map f es*)

lemma *awalk-cost-Nil*[*simp*]: *awalk-cost f* [] = *0*
unfolding *awalk-cost-def* **by** *simp*

lemma *awalk-cost-Cons*[*simp*]: *awalk-cost f* (*x* # *xs*) = *f x* + *awalk-cost f xs*
unfolding *awalk-cost-def* **by** *simp*

lemma *awalk-cost-append*[*simp*]:
awalk-cost f (*xs* @ *ys*) = *awalk-cost f xs* + *awalk-cost f ys*
unfolding *awalk-cost-def* **by** *simp*

end

end

theory *Shortest-Path* **imports**

Arc-Walk
Weighted-Graph
HOL−Library.Extended-Real
begin

19 Shortest Paths

context *wf-digraph* **begin**

definition μ **where**
$\mu\ f\ u\ v \equiv INF\ p{\in}\ \{p.\ awalk\ u\ p\ v\}.\ ereal\ (awalk\text{-}cost\ f\ p)$

lemma *shortest-path-inf*:
assumes $\neg(u \rightarrow^* v)$
shows $\mu\ f\ u\ v = \infty$
proof −
have $*$: $\{p.\ awalk\ u\ p\ v\} = \{\}$
using *assms* **by** (*auto simp*: *reachable-awalk*)
show $\mu\ f\ u\ v = \infty$ **unfolding** μ*-def* $*$
by (*simp add*: *top-ereal-def*)
qed

lemma *min-cost-le-walk-cost*:
assumes *awalk u p v*
shows $\mu\ c\ u\ v \leq awalk\text{-}cost\ c\ p$
using *assms* **unfolding** μ*-def* **by** (*auto intro*: *INF-lower2*)

lemma *pos-cost-pos-awalk-cost*:
assumes *awalk u p v*
assumes *pos-cost*: $\bigwedge e.\ e \in arcs\ G \Longrightarrow c\ e \geq 0$
shows *awalk-cost c p* ≥ 0
using *assms* **by** (*induct p arbitrary*: *u*) (*auto simp*: *awalk-Cons-iff*)

fun *mk-cycles-path* :: *nat*
$\Rightarrow$ *'b awalk* $\Rightarrow$ *'b awalk* **where**
mk-cycles-path 0 c = []
| *mk-cycles-path* (*Suc n*) *c* = *c* @ (*mk-cycles-path n c*)

lemma *mk-cycles-path-awalk*:
assumes *awalk u c u*
shows *awalk u* (*mk-cycles-path n c*) *u*
using *assms* **by** (*induct n*) (*auto simp*: *awalk-Nil-iff*)

lemma *mk-cycles-awalk-cost*:
assumes *awalk u p u*
shows *awalk-cost c* (*mk-cycles-path n p*) = *n* $*$ *awalk-cost c p*
using *assms* **proof** (*induct rule*: *mk-cycles-path.induct*)
case *1* **show** *?case* **by** *simp*

```
next
  case (2 n p)
  have awalk-cost c (mk-cycles-path (Suc n) p)
      = awalk-cost c (p @ (mk-cycles-path n p))
    by simp
  also have ... = awalk-cost c p + real n * awalk-cost c p
  proof (cases n)
    case 0 then show ?thesis by simp
  next
    case (Suc n') then show ?thesis
      using 2 by simp
  qed
  also have ... = real (Suc n) * awalk-cost c p
    by (simp add: algebra-simps)
  finally show ?case .
qed

lemma inf-over-nats:
  fixes a c :: real
  assumes c < 0
  shows (INF (i :: nat). ereal (a + i * c)) = - ∞
proof (rule INF-eqI)
  fix i :: nat show - ∞ ≤ a + real i * c by simp
next
  fix y :: ereal
  assume ⋀(i :: nat). i ∈ UNIV ⟹ y ≤ a + real i * c
  then have l-assm: ⋀i::nat. y ≤ a + real i * c by simp

  show y ≤ - ∞
  proof (subst ereal-infty-less-eq, rule ereal-bot)
    fix B :: real
    obtain real-x where a + real-x * c ≤ B using ‹c < 0›
      by atomize-elim
         (rule exI[where x=(- abs B - a)/c], auto simp: field-simps)
    obtain x :: nat where a + x * c ≤ B
    proof (atomize-elim, intro exI[where x=nat(ceiling real-x)] conjI)
      have real (nat(ceiling real-x)) * c ≤ real-x * c
        using ‹c < 0› by (simp add: real-nat-ceiling-ge)
      then show a + nat(ceiling real-x) * c ≤ B
        using ‹a + real-x * c ≤ B› by simp
    qed
    then show y ≤ ereal B
    proof -
      have ereal (a + x * c) ≤ ereal B
        using ‹a + x * c ≤ B› by simp
      with l-assm show ?thesis by (rule order-trans)
    qed
  qed
qed
```

lemma *neg-cycle-imp-inf-μ*:
assumes *walk-p*: *awalk u p v*
assumes *walk-c*: *awalk w c w*
assumes *w-in-p*: *w ∈ set* (*awalk-verts u p*)
assumes *awalk-cost f c < 0*
shows *μ f u v = −∞*
proof −
from *w-in-p* **obtain** *xs ys* **where** *pv-decomp*: *awalk-verts u p = xs @ w # ys*
by (*auto simp*: *in-set-conv-decomp*)

define *q r* **where** *q = take* (*length xs*) *p* **and** *r = drop* (*length xs*) *p*
define *ext-p* **where** *ext-p n = q @ mk-cycles-path n c @ r* **for** *n*

have *ext-p-cost*: *⋀n. awalk-cost f* (*ext-p n*)
= awalk-cost f q + awalk-cost f r + *n ∗ awalk-cost f c*
using ‹*awalk w c w*›
by (*auto simp*: *ext-p-def intro*: *mk-cycles-awalk-cost*)

from *q-def r-def* **have** *awlast u q = w*
using *pv-decomp walk-p* **by** (*auto simp*: *awalk-verts-take-conv elim*!: *awalkE*)
moreover
from *q-def r-def* **have** *awalk u* (*q @ r*) *v*
using *walk-p* **by** *simp*
ultimately
have *awalk u q w awalk w r v ⋀n. awalk w* (*mk-cycles-path n c*) *w*
using *walk-c*
by (*auto simp*: *intro*: *mk-cycles-path-awalk*)
then have *⋀n. awalk u* (*ext-p n*) *v*
unfolding *ext-p-def* **by** (*blast intro*: *awalk-appendI*)
then have {*ext-p i*|*i*. *i ∈ UNIV*} ⊆ {*p*. *awalk u p v*}
by *auto*
then have (*INF p*∈{*p*. *awalk u p v*}. *ereal* (*awalk-cost f p*))
≤ (*INF p*∈ {*ext-p i*|*i*. *i ∈ UNIV*}. *ereal* (*awalk-cost f p*))
by (*auto intro*: *INF-superset-mono*)
also have *. . .* = (*INF i*∈ *UNIV*. *ereal* (*awalk-cost f* (*ext-p i*)))
by (*rule arg-cong*[**where** *f*=*Inf*], *auto*)
also have *. . .* = *−* ∞ **unfolding** *ext-p-cost*
by (*rule inf-over-nats*[*OF* ‹*awalk-cost f c < 0*›])
finally show *?thesis* **unfolding** *μ-def* **by** *simp*
qed

lemma *walk-cheaper-path-imp-neg-cyc*:
assumes *p-props*: *awalk u p v*
assumes *less-path-μ*: *awalk-cost f p < (INF p*∈ {*p*. *apath u p v*}. *ereal* (*awalk-cost f p*))
shows ∃ *w c*. *awalk w c w* ∧ *w ∈ set* (*awalk-verts u p*) ∧ *awalk-cost f c < 0*
proof −
define *path-μ* **where** *path-μ = (INF p*∈ {*p*. *apath u p v*}. *ereal* (*awalk-cost f p*))

```
  then have awalk u p v and awalk-cost f p < path-μ
    using p-props less-path-μ by simp-all
  then show ?thesis
  proof (induct rule: awalk-to-apath-induct)
    case (path p) then have apath u p v by (auto simp: apath-def)
   then show ?case using path.prems by (auto simp: path-μ-def dest: not-mem-less-INF)
  next
    case (decomp p q r s)
    then obtain w where p-props: p = q @ r @ s awalk u q w awalk w r w awalk
w s v
      by (auto elim: awalk-cyc-decompE)
    then have awalk u (q @ s) v
      using ‹awalk u p v› by (auto simp: awalk-appendI)
    then have verts-ss: set (awalk-verts u (q @ s)) ⊆ set (awalk-verts u p)
      using ‹awalk u p v› ‹p = q @ r @ s› by (auto simp: set-awalk-verts)

    show ?case
    proof (cases ereal (awalk-cost f (q @ s)) < path-μ)
      case True then have ∃ w c. awalk w c w ∧ w ∈ set (awalk-verts u (q @ s))
∧ awalk-cost f c < 0
        by (rule decomp)
      then show ?thesis using verts-ss by auto
    next
      case False
      note ‹awalk-cost f p < path-μ›
      also have path-μ ≤ awalk-cost f (q @ s)
        using False by simp
      finally have awalk-cost f r < 0 using ‹p = q @ r @ s› by simp
      moreover
      have w ∈ set (awalk-verts u q) using ‹awalk u q w› by auto
      then have w ∈ set (awalk-verts u p)
        using ‹awalk u p v› ‹awalk u q w› ‹p = q @ r @ s›
        by (auto simp: set-awalk-verts)
      ultimately
      show ?thesis using ‹awalk w r w› by auto
    qed
  qed
qed

lemma (in fin-digraph) neg-inf-imp-neg-cyc:
  assumes inf-mu: μ f u v = − ∞
  shows ∃ p. awalk u p v ∧ (∃ w c. awalk w c w ∧ w ∈ set (awalk-verts u p) ∧
awalk-cost f c < 0)
proof −
  define path-μ where path-μ = (INF s∈{p. apath u p v}. ereal (awalk-cost f s))

  have awalks-ne: {p. awalk u p v} ≠ {}
    using inf-mu unfolding μ-def by safe (simp add: top-ereal-def)
  then have paths-ne: {p. apath u p v} ~= {}
```

```
    by (auto intro: apath-awalk-to-apath)

  obtain p where apath u p v awalk-cost f p = path-μ
  proof −
    obtain p where p ∈ {p. apath u p v} awalk-cost f p = path-μ
    using finite-INF-in[OF apaths-finite paths-ne, of awalk-cost f]
      by (auto simp: path-μ-def)
    then show ?thesis using that by auto
  qed
  then have path-μ ≠ −∞ by auto
  then have μ f u v < path-μ using inf-mu by simp
  then obtain pw where p-def: awalk u pw v awalk-cost f pw < path-μ
    by atomize-elim (auto simp: μ-def INF-less-iff)
  then have ∃ w c. awalk w c w ∧ w ∈ set (awalk-verts u pw) ∧ awalk-cost f c < 0
    by (intro walk-cheaper-path-imp-neg-cyc) (auto simp: path-μ-def)
  with ‹awalk u pw v› show ?thesis by auto
qed

lemma (in fin-digraph) no-neg-cyc-imp-no-neg-inf:
  assumes no-neg-cyc: ⋀p. awalk u p v
    ⟹ ¬(∃ w c. awalk w c w ∧ w ∈ set (awalk-verts u p) ∧ awalk-cost f c < 0)
  shows − ∞ < μ f u v
proof (intro ereal-MInfty-lessI notI)
  assume μ f u v = − ∞
  then obtain p where p-props: awalk u p v
    and ex-cyc: ∃ w c. awalk w c w ∧ w ∈ set (awalk-verts u p) ∧ awalk-cost f c <
0
    by atomize-elim (rule neg-inf-imp-neg-cyc)
  then show False using no-neg-cyc by blast
qed

lemma μ-reach-conv:
  μ f u v < ∞ ⟷ u →* v
proof
  assume μ f u v < ∞
  then have {p. awalk u p v} ≠ {}
    unfolding μ-def by safe (simp add: top-ereal-def)
  then show u →* v by (simp add: reachable-awalk)
next
  assume u →* v
  then obtain p where p-props: apath u p v
    by (metis reachable-awalk apath-awalk-to-apath)
  then have {p} ⊆ {p. apath u p v} by simp
  then have μ f u v ≤ (INF p∈ {p}. ereal (awalk-cost f p))
    unfolding μ-def by (intro INF-superset-mono) (auto simp: apath-def)
  also have ... < ∞ by (simp add: min-def)
  finally show μ f u v < ∞ .
qed
```

lemma *awalk-to-path-no-neg-cyc-cost*:
assumes *p-props*:*awalk u p v*
assumes *no-neg-cyc*: ¬ (∃ *w c*. *awalk w c w* ∧ *w* ∈ *set* (*awalk-verts u p*) ∧ *awalk-cost f c* < *0*)
shows *awalk-cost f* (*awalk-to-apath p*) ≤ *awalk-cost f p*
using *assms*
proof (*induct rule*: *awalk-to-apath-induct*)
case *path* **then show** *?case* **by** (*auto simp*: *awalk-to-apath.simps*)
next
case (*decomp p q r s*)
from *decomp*(*2*,*3*) **have** *is-awalk-cyc-decomp p* (*q*,*r*,*s*)
using *awalk-cyc-decomp-has-prop*[*OF decomp*(*1*)] **by** *auto*
then have *decomp-props*: *p* = *q* @ *r* @ *s* ∃ *w*. *awalk w r w* **by** *auto*

have *awalk-cost f* (*awalk-to-apath p*) = *awalk-cost f* (*awalk-to-apath* (*q* @ *s*))
using *decomp* **by** (*auto simp*: *step-awalk-to-apath*[*of - p - q r s*])
also have ... ≤ *awalk-cost f* (*q* @ *s*)
proof −
have *awalk u* (*q* @ *s*) *v*
using ‹*awalk u p v*› *decomp-props* **by** (*auto dest*!: *awalk-ends-eqD*)
then have *set* (*awalk-verts u* (*q* @ *s*)) ⊆ *set* (*awalk-verts u p*)
using ‹*awalk u p v*› ‹*p* = *q* @ *r* @ *s*›
by (*auto simp add*: *set-awalk-verts*)
then show *?thesis* **using** *decomp.prems* **by** (*intro decomp.hyps*) *auto*
qed
also have ... ≤ *awalk-cost f p*
proof −
obtain *w* **where** *awalk u q w awalk w r w awalk w s v*
using *decomp* **by** (*auto elim*: *awalk-cyc-decompE*)
then have *w* ∈ *set* (*awalk-verts u q*) **by** *auto*
then have *w* ∈ *set* (*awalk-verts u p*)
using ‹*p* = *q* @ *r* @ *s*› ‹*awalk u p v*› ‹*awalk u q w*›
by (*auto simp add*: *set-awalk-verts*)
then have *0* ≤ *awalk-cost f r* **using** ‹*awalk w r w*›
using *decomp.prems* **by** (*auto simp*: *not-less*)
then show *?thesis* **using** ‹*p* = *q* @ *r* @ *s*› **by** *simp*
qed
finally show *?case* **.**
qed

lemma (**in** *fin-digraph*) *no-neg-cyc-reach-imp-path*:
assumes *reach*: *u* →* *v*
assumes *no-neg-cyc*: ⋀*p*. *awalk u p v*
⟹ ¬(∃ *w c*. *awalk w c w* ∧ *w* ∈ *set* (*awalk-verts u p*) ∧ *awalk-cost f c* < *0*)
shows ∃ *p*. *apath u p v* ∧ *μ f u v* = *awalk-cost f p*
proof −
define *set-walks* **where** *set-walks* = {*p*. *awalk u p v*}
define *set-paths* **where** *set-paths* = {*p*. *apath u p v*}

```
  have set-paths ≠ {}
  proof −
    obtain p where apath u p v
      using reach by (metis apath-awalk-to-apath reachable-awalk)
    then show ?thesis unfolding set-paths-def by blast
  qed

  have μ f u v = (INF p∈ set-walks. ereal (awalk-cost f p))
    unfolding μ-def set-walks-def by simp
  also have ... = (INF p∈ set-paths. ereal (awalk-cost f p))
  proof (rule antisym)
    have awalk-to-apath ‘ set-walks ⊆ set-paths
      unfolding set-walks-def set-paths-def
      by (intro subsetI) (auto elim: apath-awalk-to-apath)
    then have (INF p∈ set-paths. ereal (awalk-cost f p))
      ≤ (INF p∈ awalk-to-apath ‘ set-walks. ereal (awalk-cost f p))
      by (rule INF-superset-mono) simp
    also have ... = (INF p∈ set-walks. ereal (awalk-cost f (awalk-to-apath p)))
      by (simp add: image-comp)
    also have ... ≤ (INF p∈ set-walks. ereal (awalk-cost f p))
    proof −
      { fix p assume p ∈ set-walks
        then have awalk u p v by (auto simp: set-walks-def)
        then have awalk-cost f (awalk-to-apath p) ≤ awalk-cost f p
          using no-neg-cyc
          using no-neg-cyc and awalk-to-path-no-neg-cyc-cost
          by auto }
      then show ?thesis by (intro INF-mono) auto
    qed
    finally show
      (INF p∈ set-paths. ereal (awalk-cost f p))
      ≤ (INF p∈ set-walks. ereal (awalk-cost f p)) by simp

    have set-paths ⊆ set-walks
      unfolding set-paths-def set-walks-def by (auto simp: apath-def)
    then show (INF p∈ set-walks. ereal (awalk-cost f p))
      ≤ (INF p∈ set-paths. ereal (awalk-cost f p))
      by (rule INF-superset-mono) simp
  qed
  also have ... ∈ (λp. ereal (awalk-cost f p)) ‘ set-paths
    using apaths-finite ‹set-paths ≠ {}›
    by (intro finite-INF-in) (auto simp: set-paths-def)
  finally show ?thesis
    by (simp add: set-paths-def image-def)
qed

lemma (in fin-digraph) min-cost-awalk:
  assumes reach: u →* v
  assumes pos-cost: ⋀e. e ∈ arcs G ⟹ c e ≥ 0
```

```
  shows ∃p. apath u p v ∧ μ c u v = awalk-cost c p
proof −
  have pc: ⋀u p v. awalk u p v ⟹ 0 ≤ awalk-cost c p
    using pos-cost-pos-awalk-cost pos-cost by auto

  from reach show ?thesis
    by (rule no-neg-cyc-reach-imp-path) (auto simp: not-less intro: pc)
qed

lemma (in fin-digraph) pos-cost-mu-triangle:
  assumes pos-cost: ⋀e. e ∈ arcs G ⟹ c e ≥ 0
  assumes e-props: arc-to-ends G e = (u,v) e ∈ arcs G
  shows μ c s v ≤ μ c s u + c e
proof cases
  assume μ c s u = ∞ then show ?thesis by simp
next
  assume μ c s u ≠ ∞
  then have {p. awalk s p u} ≠ {}
    unfolding μ-def by safe (simp add: top-ereal-def)
  then have s →* u by (simp add: reachable-awalk)
  with pos-cost
  obtain p where p-props: apath s p u
      and p-cost: μ c s u = awalk-cost c p
    by (metis min-cost-awalk)

  have awalk u [e] v
    using e-props by (auto simp: arc-to-ends-def awalk-simps)
  with ‹apath s p u›
  have awalk s (p @ [e]) v
    by (auto simp: apath-def awalk-appendI)
  then have μ c s v ≤ awalk-cost c (p @ [e])
    by (rule min-cost-le-walk-cost)
  also have ... ≤ awalk-cost c p + c e by simp
  also have ... ≤ μ c s u + c e using p-cost by simp
  finally show ?thesis .
qed

lemma (in fin-digraph) mu-exact-triangle:
  assumes v ≠ s
  assumes s →* v
  assumes nonneg-arcs: ⋀e. e ∈ arcs G ⟹ 0 ≤ c e
  obtains u e where μ c s v = μ c s u + c e and arc e (u,v)
proof −
  obtain p where p-path: apath s p v
    and p-cost: μ c s v = awalk-cost c p
    using assms by (metis min-cost-awalk)
  then obtain e p' where p'-props: p = p' @ [e] using ‹v ≠ s›
    by (cases p rule: rev-cases) (auto simp: apath-def)
  then obtain u where awalk s p' u awalk u [e] v
```

```
    using ‹apath s p v› by (auto simp: apath-def)
  then have mu-le: μ c s v ≤ μ c s u + c e and arc: arc e (u,v)
    using nonneg-arcs by (auto intro!: pos-cost-mu-triangle simp: arc-to-ends-def
arc-def)

  have μ c s u + c e ≤ ereal (awalk-cost c p') + ereal (c e)
    using ‹awalk s p' u›
    by (fast intro: add-right-mono min-cost-le-walk-cost)
  also have ... = awalk-cost c p using p'-props by simp
  also have ... = μ c s v using p-cost by simp
  finally
  have μ c s v = μ c s u + c e using mu-le by auto
  then show ?thesis using arc ..
qed

lemma (in fin-digraph) mu-exact-triangle-Ex:
  assumes v ≠ s
  assumes s →* v
  assumes ⋀e. e ∈ arcs G ⟹ 0 ≤ c e
  shows ∃ u e. μ c s v = μ c s u + c e ∧ arc e (u,v)
using assms by (metis mu-exact-triangle)

lemma (in fin-digraph) mu-Inf-triangle:
  assumes v ≠ s
  assumes ⋀e. e ∈ arcs G ⟹ 0 ≤ c e
  shows μ c s v = Inf {μ c s u + c e | u e. arc e (u, v)} (is - = Inf ?S)
proof cases
  assume s →* v
  then obtain u e where μ c s v = μ c s u + c e arc e (u,v)
    using assms by (metis mu-exact-triangle)
  then have Inf ?S ≤ μ c s v by (auto intro: Complete-Lattices.Inf-lower)
  also have ... ≤ Inf ?S using assms(2)
    by (auto intro!: Complete-Lattices.Inf-greatest pos-cost-mu-triangle
      simp: arc-def arc-to-ends-def)
  finally show ?thesis by simp
next
  assume ¬s →* v
  then have μ c s v = ∞ by (metis shortest-path-inf)
  define S where S = ?S
  show μ c s v = Inf S
  proof cases
    assume S = {}
    then show ?thesis unfolding ‹μ c s v = ∞›
      by (simp add: top-ereal-def)
  next
    assume S ≠ {}
    { fix x assume x ∈ S
      then obtain u e where arc e (u,v) and x-val: x = μ c s u + c e
        unfolding S-def by auto
```

```
    then have ¬s →* u using ‹¬ s →* v› by (metis reachable-arc-trans)
    then have μ c s u + c e= ∞ by (simp add: shortest-path-inf)
    then have x = ∞ using x-val by simp }
  then have S = {∞} using ‹S ≠ {}› by auto
  then show ?thesis using ‹μ c s v = ∞› by (simp add: min-def)
 qed
qed

end

end
```

```
theory Graph-Theory
imports
  Digraph
  Bidirected-Digraph
  Arc-Walk

  Digraph-Component
  Digraph-Component-Vwalk
  Digraph-Isomorphism
  Pair-Digraph
  Vertex-Walk
  Subdivision

  Euler
  Kuratowski
  Shortest-Path

begin

end
```

References

[1] J. Bang-Jensen and G. Z. Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer, 2 edition, 2009.

[2] R. Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer, 4 edition, 2010. http://diestel-graph-theory.com.

[3] F. Harary and R. Read. Is the null-graph a pointless concept? In R. Bari and F. Harary, editors, *Graphs and Combinatorics*, volume 406 of *Lecture Notes in Mathematics*, pages 37–44. Springer Berlin Heidelberg, 1974.