

# Graph Theory

By Lars Noschinski

February 23, 2021

## 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

<b>1</b>	<b>Reflexive-Transitive Closure on a Domain</b>	<b>3</b>
<b>2</b>	<b>Additional theorems for base libraries</b>	<b>6</b>
2.1	List . . . . .	6
<b>3</b>	<b>NOMATCH simproc</b>	<b>8</b>
<b>4</b>	<b>Digraphs</b>	<b>9</b>
4.1	Reachability . . . . .	11
4.2	Degrees of vertices . . . . .	14
4.3	Graph operations . . . . .	15
<b>5</b>	<b>Bidirected Graphs</b>	<b>18</b>
<b>6</b>	<b>Arc Walks</b>	<b>21</b>
6.1	Basic Lemmas . . . . .	22
6.2	Appending awalks . . . . .	26
6.3	Cycles . . . . .	31
6.4	Reachability . . . . .	32
6.5	Paths . . . . .	34

<b>7</b>	<b>Digraphs without Parallel Arcs</b>	<b>39</b>
7.1	Path reversal for Pair Digraphs . . . . .	43
7.2	Subdividing Edges . . . . .	45
7.3	Bidirected Graphs . . . . .	52
<b>8</b>	<b>Components of (Symmetric) Digraphs</b>	<b>53</b>
8.1	Compatible Graphs . . . . .	54
8.2	Basic lemmas . . . . .	55
8.3	The underlying symmetric graph of a digraph . . . . .	57
8.4	Subgraphs and Induced Subgraphs . . . . .	58
8.5	Induced subgraphs . . . . .	61
8.6	Unions of Graphs . . . . .	66
8.7	Maximal Subgraphs . . . . .	67
8.8	Connected and Strongly Connected Graphs . . . . .	68
8.9	Components . . . . .	80
<b>9</b>	<b>Walks Based on Vertices</b>	<b>82</b>
<b>10</b>	<b>Lemmas for Vertex Walks</b>	<b>97</b>
<b>11</b>	<b>Isomorphisms of Digraphs</b>	<b>98</b>
11.1	Graph Invariants . . . . .	109
<b>12</b>	<b>Auxiliary Lemmas about <math>(\sim)</math></b>	<b>110</b>
<b>13</b>	<b>Function-power distance between values</b>	<b>113</b>
<b>14</b>	<b>Cyclic Permutations</b>	<b>113</b>
14.1	Orbits . . . . .	119
14.2	Decomposition of Arbitrary Permutations . . . . .	120
14.3	Funpow + Orbit . . . . .	122
14.4	Permutation Domains . . . . .	126
<b>15</b>	<b>Segments</b>	<b>126</b>
<b>16</b>	<b>Lists of Powers</b>	<b>131</b>
<b>17</b>	<b>Subdivision on Digraphs</b>	<b>132</b>
17.1	Subdivision on Pair Digraphs . . . . .	138
<b>18</b>	<b>Euler Trails in Digraphs</b>	<b>141</b>
18.1	Trails and Euler Trails . . . . .	141
18.2	Arc Balance of Walks . . . . .	143
18.3	Closed Euler Trails . . . . .	145
18.4	Open euler trails . . . . .	150

<b>19 Kuratowski Subgraphs</b>	<b>155</b>
19.1 Public definitions . . . . .	155
19.2 Inner vertices of a walk . . . . .	157
19.3 Progressing Walks . . . . .	158
19.4 Walks with Restricted Vertices . . . . .	160
19.5 Properties of subdivisions . . . . .	160
19.6 Pair Graphs . . . . .	162
19.7 Slim graphs . . . . .	167
19.8 Contraction Preserves Kuratowski-Subgraph-Property . . . . .	174
19.9 Final proof . . . . .	177
<b>20 Weighted Graphs</b>	<b>187</b>
<b>21 Shortest Paths</b>	<b>188</b>

```

theory Rtranc1-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**

```

rtranc1-on :: 'a set => 'a rel => 'a rel
for F :: 'a set and r :: 'a rel

```

**where**

```

rtranc1-on-refl [intro!, Pure.intro!, simp]: a ∈ F ==> (a, a) ∈ rtranc1-on F r
| rtranc1-on-into-rtranc1-on [Pure.intro]:
  (a, b) ∈ rtranc1-on F r ==> (b, c) ∈ r ==> c ∈ F
  ==> (a, c) ∈ rtranc1-on F r

```

**definition** *symcl* :: 'a rel => 'a rel ((-<sup>s</sup>) [1000] 999) **where**

```

symcl R = R ∪ (λ(a,b). (b,a)) ‘ R

```

**lemma** *in-rtranc1-on-in-F*:

```

assumes (a,b) ∈ rtranc1-on F r shows a ∈ F b ∈ F
using assms by induct auto

```

**lemma** *rtranc1-on-induct*[*consumes 1, case-names base step, induct set: rtranc1-on*]:

```

assumes (a, b) ∈ rtranc1-on F r
and a ∈ F ==> P a
  ∧ y z. [(a, y) ∈ rtranc1-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-rtranc1-on-in-F)

```

**lemma** *rtrancl-on-trans*:  
**assumes**  $(a,b) \in rtrancl\text{-on } F r$   $(b,c) \in rtrancl\text{-on } F r$  **shows**  $(a,c) \in rtrancl\text{-on } F r$   
**using** *assms(2,1)*  
**by** *induct (auto intro: rtrancl-on-into-rtrancl-on)*

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

**theorem** *rtrancl-on-converseD*:  
**assumes**  $(y,x) \in rtrancl\text{-on } F (r^{-1})$  **shows**  $(x,y) \in rtrancl\text{-on } F r$   
**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*:  $(a,b) \in rtrancl\text{-on } F r$   
**and cases**:  $b \in F \implies P b$   
 $\bigwedge x y. \llbracket (x,y) \in r; (y,b) \in rtrancl\text{-on } F r; x \in F; y \in F; P y \rrbracket \implies P x$   
**shows**  $P a$   
**using** *rtrancl-on-converseI[OF major] cases*  
**by** *induct (auto intro: rtrancl-on-converseD)*

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

**lemma** *rtrancl-on-sym*:  
**assumes** *sym*  $r$  **shows** *sym*  $(rtrancl\text{-on } F r)$

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

**lemma** *rtrancl-on-mono*:

**assumes**  $s \subseteq r \ F \subseteq G \ (a,b) \in rtrancl\text{-on } F \ s$  **shows**  $(a,b) \in rtrancl\text{-on } G \ r$

**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*:  $\bigwedge a \ b. \llbracket a \in F; (a,b) \in r \rrbracket \implies b \in F$

**shows**  $(a,b) \in rtrancl\text{-on } F \ r$

**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**  $(z,b) \in rtrancl\text{-on } F \ r$  **by** *simp*

**with** *step.prem*s  $\langle (y,z) \in r \ \langle z \in F \rangle$  **show** *?case*

**using** *converse-rtrancl-on-into-rtrancl-on*

**by** *metis*

**qed** *simp*

**lemma** *rtrancl-on-rtranclI*:

$(a,b) \in rtrancl\text{-on } F \ r \implies (a,b) \in r^*$

**by** (*induct rule: rtrancl-on-induct*) *simp-all*

**lemma** *rtrancl-on-sub-rtrancl*:

$rtrancl\text{-on } F \ r \subseteq r^*$

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

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

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

**lemma** *sym-diff*:  
  **assumes**  $\text{sym } A \text{ and } \text{sym } B$  **shows**  $\text{sym } (A - B)$   
**using** *assms* **by** (*auto simp: sym-def*)

### 2.1 List

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

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

**lemma** *tl-rev*:  
   $\text{tl } (\text{rev } p) = \text{rev } (\text{butlast } p)$   
**by** (*induct p*) *auto*

**lemma** *butlast-rev*:  
   $\text{butlast } (\text{rev } p) = \text{rev } (\text{tl } p)$

**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**  $A: \neg m < n$

**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**  $\neg distinct\ ws$

**shows**  $\exists\ xs\ ys\ zs\ y. ws = xs\ @\ y\ \#\ ys\ @\ y\ \#\ zs \wedge distinct\ xs \wedge y \notin set\ xs \wedge y \notin set\ ys$

**proof**  $-$

**obtain**  $xs\ y\ ys$  **where**  $y \in set\ xs$  *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**  $\neg distinct\ ws$

**shows**  $\exists\ xs\ y\ ys\ zs. ws = xs\ @\ y\ \#\ ys\ @\ y\ \#\ zs \wedge distinct\ (ys\ @\ [y])$

**using** *assms*

**proof** (*induct ws*)

**case** (*Cons w ws*)

**show** *?case*

**proof** (*cases distinct ws*)

**case** *True*

**then obtain**  $xs\ ys$  **where**  $ws = xs\ @\ w\ \#\ ys$   $w \notin set\ xs$

**using** *Cons.prem*s **by** (*fastforce dest: split-list-first*)

**then have**  $distinct\ (xs\ @\ [w])\ w\ \#\ ws = []\ @\ w\ \#\ xs\ @\ w\ \#\ ys$

**using**  $\langle distinct\ ws \rangle$  **by** *auto*

**then show** *?thesis* **by** *blast*

**next**

**case** *False*

```

then obtain  $xs\ y\ ys\ zs$  where  $ws = xs @ y \# ys @ y \# zs \wedge distinct\ (ys @ [y])$ 
using Cons by auto
then have  $w \# ws = (w \# xs) @ y \# ys @ y \# zs \wedge distinct\ (ys @ [y])$ 
by simp
then show ?thesis by blast
qed
qed simp

```

```

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

```

```

lemma list-set-tl:  $x \in set\ (tl\ xs) \implies x \in set\ xs$ 
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) =  $\langle fn\ - \implies fn\ ctxt \implies fn\ ct$ 
 $\implies$ 
  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
   $\rangle$ 

```

This setup ensures that a rewrite rule of the form  $NOMATCH\ val\ pat \implies t$  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  $\implies$  tail G e  $\in$  verts G
  assumes head-in-verts[simp]: e  $\in$  arcs G  $\implies$  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 [2].

```
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  $\implies$  tail G e  $\neq$  head G e
```

```
locale nomulti-digraph = wf-digraph +
  assumes no-multi-arcs:  $\bigwedge e1 e2. \llbracket e1 \in \textit{arcs} G; e2 \in \textit{arcs} G; \textit{arc-to-ends} G e1 = \textit{arc-to-ends} G e2 \rrbracket \implies e1 = e2$ 
```

```
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  $\Rightarrow$  'a  $\times$  'a  $\Rightarrow$  bool **where**

*arc*  $e\ uv \equiv e \in \text{arcs } G \wedge \text{tail } G\ e = \text{fst } uv \wedge \text{head } G\ e = \text{snd } uv$

**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*  $G = (\lambda e. (\text{tail } G\ e, \text{head } G\ e)) \text{ ` arcs } G$

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

**lemma** *symmetric-conv*: *symmetric*  $G \longleftrightarrow (\forall e1 \in \text{arcs } G. \exists e2 \in \text{arcs } G. \text{tail } G\ e1 = \text{head } G\ e2 \wedge \text{head } G\ e1 = \text{tail } G\ e2)$

**unfolding** *symmetric-def arcs-ends-conv sym-def* **by** *auto*

**lemma** *arcs-ends-symmetric*:

**assumes** *symmetric*  $G$

**shows**  $(u, v) \in \text{arcs-ends } G \implies (v, u) \in \text{arcs-ends } G$

**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**  $\text{dominates} :: ('a, 'b) \text{ pre-digraph} \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{bool} (- \rightarrow_1 - [100,100]$   
 $40)$  **where**

$\text{dominates } G \ u \ v \equiv (u, v) \in \text{arcs-ends } G$

**abbreviation**  $\text{reachable1} :: ('a, 'b) \text{ pre-digraph} \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{bool} (- \rightarrow^+_1 - [100,100]$   
 $40)$  **where**

$\text{reachable1 } G \ u \ v \equiv (u, v) \in (\text{arcs-ends } G)^{\wedge+}$

**definition**  $\text{reachable} :: ('a, 'b) \text{ pre-digraph} \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{bool} (- \rightarrow^*_1 - [100,100]$   
 $40)$  **where**

$\text{reachable } G \ u \ v \equiv (u, v) \in \text{rtrancl-on } (\text{verts } G) (\text{arcs-ends } G)$

**lemma**  $\text{reachableE}[\text{elim}]$ :

**assumes**  $u \rightarrow_G v$

**obtains**  $e$  **where**  $e \in \text{arcs } G \ \text{tail } G \ e = u \ \text{head } G \ e = v$

**using**  $\text{assms}$  **by**  $(\text{auto simp add: arcs-ends-conv})$

**lemma**  $(\text{in loopfree-digraph}) \ \text{adj-not-same}$ :

**assumes**  $a \rightarrow a$  **shows**  $\text{False}$

**using**  $\text{assms}$  **by**  $(\text{rule reachableE}) (\text{auto dest: no-loops})$

**lemma**  $\text{reachable-in-vertsE}$ :

**assumes**  $u \rightarrow^*_G v$  **obtains**  $u \in \text{verts } G \ v \in \text{verts } G$

**using**  $\text{assms}$  **unfolding**  $\text{reachable-def}$  **by**  $\text{induct auto}$

**lemma**  $\text{symmetric-reachable}$ :

**assumes**  $\text{symmetric } G \ v \rightarrow^*_G w$  **shows**  $w \rightarrow^*_G v$

**proof** –

**have**  $\text{sym } (\text{rtrancl-on } (\text{verts } G) (\text{arcs-ends } G))$

**using**  $\text{assms}$  **by**  $(\text{auto simp add: symmetric-def dest: rtrancl-on-sym})$

**then show**  $?thesis$  **using**  $\text{assms}$  **unfolding**  $\text{reachable-def}$  **by**  $(\text{blast elim: symE})$

**qed**

**lemma**  $\text{reachable-rtranclI}$ :

$u \rightarrow^*_G v \implies (u, v) \in (\text{arcs-ends } G)^*$

**unfolding**  $\text{reachable-def}$  **by**  $(\text{rule rtrancl-on-rtranclI})$

**context**  $\text{wf-digraph}$  **begin**

**lemma**  $\text{adj-in-verts}$ :

**assumes**  $u \rightarrow_G v$  **shows**  $u \in \text{verts } G \ v \in \text{verts } G$

**using**  $\text{assms}$  **unfolding**  $\text{arcs-ends-conv}$  **by**  $\text{auto}$

**lemma**  $\text{dominatesI}$ : **assumes**  $\text{arc-to-ends } G \ a = (u, v) \ a \in \text{arcs } G$  **shows**  $u \rightarrow_G v$

**using**  $\text{assms}$  **by**  $(\text{auto simp: arcs-ends-def intro: rev-image-eqI})$

**lemma**  $\text{reachable-refl}$   $[\text{intro!}, \text{Pure.intro!}, \text{simp}]$ :  $v \in \text{verts } G \implies v \rightarrow^*_G v$

**unfolding** *reachable-def* **by** *auto*

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

**assumes**  $a \rightarrow_G b$   $b \rightarrow^*_G c$  **shows**  $a \rightarrow^*_G c$

**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*]:  $u \rightarrow v \implies u \rightarrow^* v$

**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**:  $u \in \text{verts } G \implies P u$

$\bigwedge x y. \llbracket u \rightarrow^*_G x; x \rightarrow_G y; P x \rrbracket \implies P y$

**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**:  $v \in \text{verts } G \implies P v$

$\bigwedge x y. \llbracket x \rightarrow_G y; y \rightarrow^*_G v; P y \rrbracket \implies P x$

**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*)  $u = v$   $u \in \text{verts } G$

| (*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**  $u \in \text{verts } G$   $v \in \text{verts } G$

**using** *assms* **by** *induct (simp-all add: adj-in-verts)*

**lemma** *reachable1-in-verts*:

**assumes**  $u \rightarrow^+ v$  **shows**  $u \in \text{verts } G$   $v \in \text{verts } G$

**using** *assms*

**by** *induct (simp-all add: adj-in-verts)*

**lemma** *reachable1-reachable*[*intro*]:  
 $v \rightarrow^+ w \implies v \rightarrow^* w$   
**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**  $(v,w) \in (\text{arcs-ends } G) \hat{\ }^*$  **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*]:  
 $u \rightarrow^+ v \implies v \rightarrow^* w \implies u \rightarrow^+ w$   
**by** (*metis trancl-trans reachable-neq-reachable1*)

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

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

**lemma** *reachable-conv'*:  
**assumes**  $u \in \text{verts } G$   
**shows**  $u \rightarrow^* v \iff (u,v) \in (\text{arcs-ends } G)^*$  (**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]:  
 $e \in \text{in-arcs } G \ v \longleftrightarrow e \in \text{arcs } G \wedge \text{head } G \ e = v$   
**unfolding** *in-arcs-def* by *auto*

**lemma** *in-out-arcs-conv*[simp]:  
 $e \in \text{out-arcs } G \ v \longleftrightarrow e \in \text{arcs } G \wedge \text{tail } G \ e = v$   
**unfolding** *out-arcs-def* by *auto*

**lemma** *inout-arcs-arc-simps*[simp]:  
**assumes**  $e \in \text{arcs } G$   
**shows**  $\text{tail } G \ e = u \implies \text{out-arcs } G \ u \cap \text{insert } e \ E = \text{insert } e \ (\text{out-arcs } G \ u \cap E)$   
 $\text{tail } G \ e \neq u \implies \text{out-arcs } G \ u \cap \text{insert } e \ E = \text{out-arcs } G \ u \cap E$   
 $\text{out-arcs } G \ u \cap \{\} = \{\}$   
 $\text{head } G \ e = u \implies \text{in-arcs } G \ u \cap \text{insert } e \ E = \text{insert } e \ (\text{in-arcs } G \ u \cap E)$   
 $\text{head } G \ e \neq u \implies \text{in-arcs } G \ u \cap \text{insert } e \ E = \text{in-arcs } G \ u \cap E$   
 $\text{in-arcs } G \ u \cap \{\} = \{\}$   
**using** *assms* by *auto*

**lemma** *in-arcs-int-arcs*[simp]:  $\text{in-arcs } G \ u \cap \text{arcs } G = \text{in-arcs } G \ u$  **and**  
*out-arcs-int-arcs*[simp]:  $\text{out-arcs } G \ u \cap \text{arcs } G = \text{out-arcs } G \ u$   
**by** *auto*

**lemma** *in-arcs-in-arcs*:  $x \in \text{in-arcs } G \ u \implies x \in \text{arcs } G$   
**and** *out-arcs-in-arcs*:  $x \in \text{out-arcs } G \ u \implies x \in \text{arcs } G$   
**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:**  $\text{verts } (\text{pre-digraph.del-vert } G \ u) = \text{verts } G - \{u\}$  **and**  
**arcs-del-vert:**  $\text{arcs } (\text{pre-digraph.del-vert } G \ u) = \{a \in \text{arcs } G. \text{tail } G \ a \neq u \wedge \text{head } G \ a \neq u\}$  **and**  
**tail-del-vert:**  $\text{tail } (\text{pre-digraph.del-vert } G \ u) = \text{tail } G$  **and**  
**head-del-vert:**  $\text{head } (\text{pre-digraph.del-vert } G \ u) = \text{head } G$  **and**  
**ends-del-vert:**  $\text{arc-to-ends } (\text{pre-digraph.del-vert } G \ u) = \text{arc-to-ends } G$   
**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]:*  
 $\llbracket \text{tail } G \ a \in \text{verts } G; \text{head } G \ a \in \text{verts } G \rrbracket \implies \text{pre-digraph.del-arc } (\text{add-arc } a) \ a = \text{del-arc } a$   
**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 \notin \text{arcs } G \implies \text{del-arc } a = G*  
**by** (*rule pre-digraph.equality (auto simp: add-arc-def)*)

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

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

**lemma** *in-arcs-del-arc-iff:*  
 $\text{in-arcs } (\text{del-arc } a) \ u = (\text{if } \text{head } G \ a = u \text{ then } \text{in-arcs } G \ u - \{a\} \text{ else } \text{in-arcs } G \ u)$   
**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**  $u \notin \text{verts } G$   
**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**  $a \notin \text{arcs } G \implies a \notin \text{in-arcs } G \text{ } u$  **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**  $a \notin \text{arcs } G \implies a \notin \text{out-arcs } G$   $u$  **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  
 $f=\text{card}$ ])  
**qed**

**lemma** in-degree-del-arc-iff:  
  in-degree (del-arc  $a$ )  $u$  = (if head  $G$   $a$  =  $u \wedge a \in \text{arcs } G$  then in-degree  $G$   $u$  – 1  
  else in-degree  $G$   $u$ )  
**proof** –  
  **have**  $a \notin \text{arcs } G \implies a \notin \text{in-arcs } G$   $u$  **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  
 $f=\text{card}$ ])  
**qed**

**lemma** out-degree-del-arc-iff:  
  out-degree (del-arc  $a$ )  $u$  = (if tail  $G$   $a$  =  $u \wedge a \in \text{arcs } G$  then out-degree  $G$   $u$  –  
  1 else out-degree  $G$   $u$ )  
**proof** –  
  **have**  $a \notin \text{arcs } G \implies a \notin \text{out-arcs } G$   $u$  **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  
 $f=\text{card}$ ])  
**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-Library.Permutations

**begin**

## 5 Bidirected Graphs

**locale** bidirected-digraph = wf-digraph  $G$  **for**  $G$  +  
  **fixes** arev :: 'b  $\Rightarrow$  'b  
  **assumes** arev-dom:  $\bigwedge a. a \in \text{arcs } G \iff \text{arev } a \neq a$   
  **assumes** arev-arev-raw:  $\bigwedge a. a \in \text{arcs } G \implies \text{arev } (\text{arev } a) = a$

**assumes** *tail-arev*[*simp*]:  $\bigwedge a. a \in \text{arcs } G \implies \text{tail } G (\text{arev } a) = \text{head } G a$

**lemma** (in *wf-digraph*) *bidirected-digraphI*:

**assumes** *arev-eq*:  $\bigwedge a. a \notin \text{arcs } G \implies \text{arev } a = a$

**assumes** *arev-neq*:  $\bigwedge a. a \in \text{arcs } G \implies \text{arev } a \neq a$

**assumes** *arev-arev-raw*:  $\bigwedge a. a \in \text{arcs } G \implies \text{arev } (\text{arev } a) = a$

**assumes** *tail-arev*:  $\bigwedge a. a \in \text{arcs } G \implies \text{tail } G (\text{arev } a) = \text{head } G a$

**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*]:  $\text{arev } (\text{arev } a) = a$

**using** *arev-dom* **by** (*cases*  $a \in \text{arcs } G$ ) (*auto simp: arev-arev-raw*)

**lemma** *arev-o-arev*[*simp*]:  $\text{arev } o \text{ arev} = \text{id}$

**by** (*simp add: fun-eq-iff*)

**lemma** *arev-eq*:  $a \notin \text{arcs } G \implies \text{arev } a = a$

**by** (*simp add: arev-dom*)

**lemma** *arev-neq*:  $a \in \text{arcs } G \implies \text{arev } a \neq a$

**by** (*simp add: arev-dom*)

**lemma** *arev-in-arcs*[*simp*]:  $a \in \text{arcs } G \implies \text{arev } a \in \text{arcs } G$

**by** (*metis arev-arev arev-dom*)

**lemma** *head-arev*[*simp*]:

**assumes**  $a \in \text{arcs } G$  **shows**  $\text{head } G (\text{arev } a) = \text{tail } G a$

**proof** –

**from** *assms* **have**  $\text{head } G (\text{arev } a) = \text{tail } G (\text{arev } (\text{arev } a))$

**by** (*simp only: tail-arev arev-in-arcs*)

**then show** *?thesis* **by** *simp*

**qed**

**lemma** *ate-arev*[*simp*]:

**assumes**  $a \in \text{arcs } G$  **shows**  $\text{arc-to-ends } G (\text{arev } a) = \text{prod.swap } (\text{arc-to-ends } G a)$

**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:  $\bigwedge x y. \text{arev } x = \text{arev } y \iff x = y$ 
  by (metis arev-arev)

lemma in-arcs-eq:  $\text{in-arcs } G w = \text{arev } \text{ ` } \text{out-arcs } G w$ 
  by auto (metis arev-arev arev-in-arcs image-eqI in-out-arcs-conv tail-arev)

lemma inj-on-arev[intro!]:  $\text{inj-on arev } S$ 
  by (metis arev-arev inj-on-inverseI)

lemma even-card-loops:
  even (card (in-arcs  $G w \cap$  out-arcs  $G w$ )) (is even (card  $?S$ ))
proof -
  { assume  $\neg$ finite  $?S$ 
    then have  $?thesis$  by simp
  }
  moreover
  { assume  $A$ :finite  $?S$ 
    have card  $?S = \text{card } (\bigcup \{ \{a, \text{arev } a\} \mid a. a \in ?S \})$  (is - = card  $(\bigcup ?T)$ )
      by (rule arg-cong[where  $f = \text{card}$ ]) (auto intro!: exI[where  $x = \{x, \text{arev } x\}$ 
for  $x$ ])
    also have ... = sum card  $?T$ 
    proof (rule card-Union-disjoint)
      show  $\bigwedge A. A \in \{ \{a, \text{arev } a\} \mid a. a \in ?S \} \implies \text{finite } A$  by auto
      show pairwise disjnt  $\{ \{a, \text{arev } a\} \mid a. a \in \text{in-arcs } G w \cap \text{out-arcs } G w \}$ 
        unfolding pairwise-def disjnt-def
        by safe (simp-all add: arev-eq-iff)
    qed
    also have ... = sum  $(\lambda a. 2)$   $?T$ 
      by (intro sum.cong) (auto simp: card-insert-if dest: arev-neq)
    also have ... =  $2 * \text{card } ?T$  by simp
    finally have  $?thesis$  by simp
  }
  ultimately
  show  $?thesis$  by blast
qed

end

sublocale bidirected-digraph  $\subseteq$  sym-digraph
proof (unfold-locales, unfold symmetric-def, intro symI)
  fix  $u v$  assume  $u \rightarrow_G v$ 
  then obtain  $a$  where  $a \in \text{arcs } G$  arc-to-ends  $G a = (u, v)$  by (auto simp:
arcs-ends-def)
  then have  $\text{arev } a \in \text{arcs } G$  arc-to-ends  $G (\text{arev } a) = (v, u)$ 
    by (auto simp: arc-to-ends-def)
  then show  $v \rightarrow_G u$  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**  $apath :: 'a \Rightarrow 'b \Rightarrow 'a \Rightarrow bool$  **where**  
 $apath\ u\ p\ v \equiv awalk\ u\ p\ v \wedge distinct\ (awalk\text{-verts}\ u\ p)$

**end**

## 6.1 Basic Lemmas

**lemma** (**in** *pre-digraph*) *awalk-verts-conv*:  
 $awalk\text{-verts}\ u\ p = (if\ p = []\ then\ [u]\ else\ map\ (tail\ G)\ p\ @\ [head\ G\ (last\ p)])$   
**by** (*induct p arbitrary: u*) *auto*

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

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

**lemma** (**in** *pre-digraph*) *awalk-verts-ne-eq*:  
**assumes**  $p \neq []$   
**shows**  $awalk\text{-verts}\ u\ p = awalk\text{-verts}\ v\ p$   
**using** *assms* **by** (*auto simp: awalk-verts-conv*)

**lemma** (**in** *pre-digraph*) *awalk-verts-non-Nil[simp]*:  
 $awalk\text{-verts}\ u\ p \neq []$   
**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**  $u \in verts\ G\ set\ p \subseteq arcs\ G\ v \in set\ (awalk\text{-verts}\ u\ p)$   
**shows**  $v \in verts\ G$   
**using** *assms* **by** (*induct p arbitrary: u*) (*auto intro: wellformed*)

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

**lemma** *awalk-conv*:  
 $awalk\ u\ p\ v = (set\ (awalk\text{-verts}\ u\ p) \subseteq verts\ G)$

$\wedge \text{set } p \subseteq \text{arcs } G$   
 $\wedge \text{awhd } u \text{ } p = u \wedge \text{awlast } u \text{ } p = v \wedge \text{cas } u \text{ } p \text{ } v$   
**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)  $\subseteq$  verts G set p  $\subseteq$  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)  $\subseteq$  verts G set p  $\subseteq$  arcs G cas u p v*  
 $\text{awhd } u \text{ } p = u \text{ } \text{awlast } u \text{ } p = v$   
**using** *assms by (auto simp add: awalk-conv)*

**lemma** *awalk-Nil-iff*:

$\text{awalk } u \text{ } [] \text{ } v \longleftrightarrow u = v \wedge u \in \text{verts } G$   
**unfolding** *awalk-def* **by** *auto*

**lemma** *trail-Nil-iff*:

$\text{trail } u \text{ } [] \text{ } v \longleftrightarrow u = v \wedge u \in \text{verts } G$   
**by** (*auto simp: trail-def awalk-Nil-iff*)

**lemma** *apath-Nil-iff*:  $\text{apath } u \text{ } [] \text{ } v \longleftrightarrow u = v \wedge u \in \text{verts } G$

**by** (*auto simp: apath-def awalk-Nil-iff*)

**lemma** *awalk-hd-in-verts*:  $\text{awalk } u \text{ } p \text{ } v \implies u \in \text{verts } G$

**by** (*cases p*) *auto*

**lemma** *awalk-last-in-verts*:  $\text{awalk } u \text{ } p \text{ } v \implies v \in \text{verts } G$

**unfolding** *awalk-conv* **by** *auto*

**lemma** *hd-in-awalk-verts*:

$\text{awalk } u \text{ } p \text{ } v \implies u \in \text{set (awalk-verts u p)}$   
 $\text{apath } u \text{ } p \text{ } v \implies u \in \text{set (awalk-verts u p)}$   
**by** (*case-tac [!p]*) (*auto simp: apath-def*)

**lemma** *awalk-Cons-iff*:

$\text{awalk } u \text{ } (e \# es) \text{ } w \longleftrightarrow e \in \text{arcs } G \wedge u = \text{tail } G \text{ } e \wedge \text{awalk (head } G \text{ } e) \text{ } es \text{ } w$   
**by** (*auto simp: awalk-def*)

**lemma** *trail-Cons-iff*:

$\text{trail } u \text{ } (e \# es) \text{ } w \longleftrightarrow e \in \text{arcs } G \wedge u = \text{tail } G \text{ } e \wedge e \notin \text{set } es \wedge \text{trail (head } G \text{ } e) \text{ } es \text{ } w$   
**by** (*auto simp: trail-def awalk-Cons-iff*)

**lemma** *apath-Cons-iff*:

$\text{apath } u \text{ } (e \# es) \text{ } w \longleftrightarrow e \in \text{arcs } G \wedge \text{tail } G \text{ } e = u \wedge \text{apath (head } G \text{ } e) \text{ } es \text{ } w$

$\wedge \text{tail } G \ e \notin \text{set } (\text{awalk-verts } (\text{head } G \ e) \ es)$  (is ?L  $\longleftrightarrow$  ?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:

$e \in \text{arcs } G \implies \text{awalk } (\text{tail } G \ e) \ [e] \ (\text{head } G \ e)$

**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  $(\text{head } G \ e) \ es \ v \ u \notin \text{set } (\text{awalk-verts } (\text{head } G \ e) \ es)$

**by** (auto simp: apath-Cons-iff)

**moreover then have**  $v \in \text{set } (\text{awalk-verts } (\text{head } G \ e) \ es)$  **by** (auto simp: apath-def)

**ultimately show**  $u \neq v$  **by** auto

**qed** simp

**lemma** awalk-ConsI:

**assumes** awalk  $v \ es \ w$

**assumes**  $e \in \text{arcs } G$  **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-apat:

**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**  $\llbracket e \in \text{arcs } G; \text{tail } G \ e = u; \text{head } G \ e = v \rrbracket \implies P$

**shows**  $P$

**using** assms **by** (auto simp: arc-def)

**lemma** in-arcs-imp-in-arcs-ends:

**assumes**  $e \in \text{arcs } G$

**shows**  $(\text{tail } G \ e, \text{head } G \ e) \in \text{arcs-ends } G$

**using** assms **by** (auto simp: arcs-ends-conv)



**lemma** *set-awalk-verts-cas*:  
**assumes** *cas u p v*  
**shows**  $\text{set } (\text{awalk-verts } u \ p) = \{u\} \cup \text{set } (\text{map } (\text{tail } G) \ p) \cup \text{set } (\text{map } (\text{head } G) \ p)$   
**using** *assms*  
**proof** (*induct p arbitrary: u*)  
**case Nil then show ?case by simp**  
**next**  
**case (Cons e es)**  
**then have**  $\text{set } (\text{awalk-verts } (\text{head } G \ e) \ es)$   
 $= \{\text{head } G \ e\} \cup \text{set } (\text{map } (\text{tail } G) \ es) \cup \text{set } (\text{map } (\text{head } G) \ es)$   
**by** (*auto simp: awalk-Cons-iff*)  
**with Cons.premis show ?case by auto**  
**qed**

**lemma** *set-awalk-verts-not-Nil-cas*:  
**assumes**  $\text{cas } u \ p \ v \ p \neq []$   
**shows**  $\text{set } (\text{awalk-verts } u \ p) = \text{set } (\text{map } (\text{tail } G) \ p) \cup \text{set } (\text{map } (\text{head } G) \ p)$   
**proof** –  
**have**  $u \in \text{set } (\text{map } (\text{tail } G) \ p)$  **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**  $\text{set } (\text{awalk-verts } u \ p) = \{u\} \cup \text{set } (\text{map } (\text{tail } G) \ p) \cup \text{set } (\text{map } (\text{head } G) \ p)$   
**using** *assms* **by** (*intro set-awalk-verts-cas*) *blast*

**lemma** *set-awalk-verts-not-Nil*:  
**assumes**  $\text{awalk } u \ p \ v \ p \neq []$   
**shows**  $\text{set } (\text{awalk-verts } u \ p) = \text{set } (\text{map } (\text{tail } G) \ p) \cup \text{set } (\text{map } (\text{head } G) \ p)$   
**using** *assms* **by** (*intro set-awalk-verts-not-Nil-cas*) *blast*

**lemma**  
*awhd-of-awalk*:  $\text{awalk } u \ p \ v \implies \text{awhd } u \ p = u$  **and**  
*awlast-of-awalk*:  $\text{awalk } u \ p \ v \implies \text{NOMATCH } (\text{awlast } u \ p) \ v \implies \text{awlast } u \ p = v$   
**unfolding** *NOMATCH-def* **by** *auto*  
**lemmas** *awends-of-awalk[simp]* = *awhd-of-awalk awlast-of-awalk*

**lemma** *awalk-verts-arc1*:  
**assumes**  $e \in \text{set } p$   
**shows**  $\text{tail } G \ e \in \text{set } (\text{awalk-verts } u \ p)$   
**using** *assms* **by** (*auto simp: awalk-verts-conv*)

**lemma** *awalk-verts-arc2*:  
**assumes**  $\text{awalk } u \ p \ v \ e \in \text{set } p$   
**shows**  $\text{head } G \ e \in \text{set } (\text{awalk-verts } u \ p)$   
**using** *assms* **by** (*simp add: set-awalk-verts*)

**lemma** *awalk-induct-raw*[*case-names Base Cons*]:  
**assumes** *awalk u p v*  
**assumes**  $\bigwedge w1. w1 \in \text{verts } G \implies P w1 [] w1$   
**assumes**  $\bigwedge w1 w2 e \text{ es}. e \in \text{arcs } G \implies \text{arc-to-ends } G e = (w1, w2)$   
 $\implies P w2 \text{ es } v \implies P w1 (e \# \text{ es}) v$   
**shows**  $P u p v$   
**using** *assms*  
**proof** (*induct p arbitrary: u v*)  
**case** *Nil* **then show** *?case* **using** *Nil.prem*s **by** *auto*  
**next**  
**case** (*Cons e es*)  
**from** *Cons.prem*s(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*]:  
 $\text{cas } u (p @ q) v \longleftrightarrow \text{cas } u p (\text{awlast } u p) \wedge \text{cas } (\text{awlast } u p) q v$   
**by** (*induct u p v rule: cas.induct*) *auto*

**lemma** *cas-ends*:  
**assumes**  $\text{cas } u p v \text{ cas } u' p v'$   
**shows**  $(p \neq [] \wedge u = u' \wedge v = v') \vee (p = [] \wedge u = v \wedge u' = v')$   
**using** *assms* **by** (*induct u p v arbitrary: u u' rule: cas.induct*) *auto*

**lemma** *awalk-ends*:  
**assumes**  $\text{awalk } u p v \text{ awalk } u' p v'$   
**shows**  $(p \neq [] \wedge u = u' \wedge v = v') \vee (p = [] \wedge u = v \wedge u' = v')$   
**using** *assms* **by** (*simp add: awalk-def cas-ends*)

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

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

**lemma** *apath-ends*:  
**assumes**  $\text{apath } u p v$  **and**  $\text{apath } u' p v'$   
**shows**  $(p \neq [] \wedge u \neq v \wedge u = u' \wedge v = v') \vee (p = [] \wedge u = v \wedge u' = v')$   
**using** *assms* **unfolding** *apath-def* **by** (*metis assms(2) apath-nonempty-ends awalk-ends*)

**lemma** *awalk-append-iff*[*simp*]:  
 $\text{awalk } u (p @ q) v \longleftrightarrow \text{awalk } u p (\text{awlast } u p) \wedge \text{awalk } (\text{awlast } u p) q v$  (**is** *?L*)

$\longleftrightarrow ?R$   
**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**  $set\ (awalk-verts\ u\ p) \subseteq verts\ G\ set\ p \subseteq arcs\ G\ cas\ u\ p\ v$   
 $awhd\ u\ p = u\ awlast\ u\ p = v\ u \in verts\ G\ v \in verts\ G$   
**proof** –  
**have**  $u \in set\ (awalk-verts\ u\ p)\ v \in set\ (awalk-verts\ u\ p)$   
**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.prem*s **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**  $awalk\text{-}verts\ u\ (p\ @\ q) = awalk\text{-}verts\ u\ p\ @\ tl\ (awalk\text{-}verts\ (awlast\ u\ p)\ q)$   
**using** *assms* **by** (*intro awalk-verts-append-cas*) *blast*

**lemma** *awalk-verts-append2*:

**assumes**  $awalk\ u\ (p\ @\ q)\ v$   
**shows**  $awalk\text{-}verts\ u\ (p\ @\ q) = butlast\ (awalk\text{-}verts\ u\ p)\ @\ awalk\text{-}verts\ (awlast\ u\ p)\ q$   
**using** *assms* **by** (*auto simp: awalk-verts-conv*)

**lemma** *apath-append-iff*:

$apath\ u\ (p\ @\ q)\ v \longleftrightarrow apath\ u\ p\ (awlast\ u\ p) \wedge apath\ (awlast\ u\ p)\ q\ v \wedge$   
 $set\ (awalk\text{-}verts\ u\ p) \cap set\ (tl\ (awalk\text{-}verts\ (awlast\ u\ p)\ q)) = \{\}$  (**is**  $?L \longleftrightarrow$   
 $?R$ )

**proof**

**assume**  $?L$

**then have** *distinct*  $(awalk\text{-}verts\ (awlast\ u\ p)\ q)$  **by** (*auto simp: apath-def awalk-verts-append2*)

**with**  $\langle ?L \rangle$  **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**  $set\ (awalk\text{-}verts\ u\ (p\ @\ q)) = set\ (awalk\text{-}verts\ u\ p) \cup set\ (awalk\text{-}verts\ v\ q)$

**proof** –

**from** *assms* **have** *cas-pq*:  $cas\ u\ (p\ @\ q)\ w$

**by** (*simp add: awlast-if-cas*)

**moreover**

**from** *assms* **have**  $v \in set\ (awalk\text{-}verts\ u\ p)$

**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**  $set\ (awalk\text{-}verts\ u\ (p\ @\ q)) = set\ (awalk\text{-}verts\ u\ p) \cup set\ (awalk\text{-}verts\ v\ q)$

**proof** –

**from** *assms* **have**  $awalk\ u\ (p\ @\ q)\ w$  **by** *auto*

**moreover**

**with** *assms* **have**  $v \in set\ (awalk\text{-}verts\ u\ (p\ @\ q))$

**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**  $\langle cas\ u\ p\ v \rangle$  **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**  $w \in \text{set } (\text{awalk-verts } u\ p)$

**shows**  $\exists q\ r. p = q @ r \wedge \text{awalk } u\ q\ w \wedge \text{awalk } w\ r\ v$

**proof** –

**from** *assms* **have**  $\text{cas } u\ p\ v$  **by** *auto*

**moreover from** *assms* **obtain**  $xs\ ys$  **where**

$\text{awalk-verts } u\ p = xs @ w \# ys$  **by** (*auto simp: in-set-conv-decomp*)

**ultimately**

**obtain**  $q\ r$  **where**  $\text{cas } u\ q\ w\ \text{cas } w\ r\ v\ p = q @ r$   $\text{awalk-verts } u\ q = xs @ [w]$

**by** (*auto intro: awalk-decomp-verts*)

**with** *assms* **show** *?thesis* **by** *auto*

qed

**lemma** *awalk-not-distinct-decomp*:

**assumes** *awalk*  $u\ p\ v$

**assumes**  $\neg \text{distinct } (\text{awalk-verts } u\ p)$

**shows**  $\exists q\ r\ s. p = q @ r @ s \wedge \text{distinct } (\text{awalk-verts } u\ q)$

$\wedge 0 < \text{length } r$

$\wedge (\exists w. \text{awalk } u\ q\ w \wedge \text{awalk } w\ r\ w \wedge \text{awalk } w\ s\ v)$

**proof** –

**from** *assms*

**obtain**  $xs\ ys\ zs\ y$  **where**

*pv-decomp*:  $\text{awalk-verts } u\ p = xs @ y \# ys @ y \# zs$

**and** *xs-y-props*:  $\text{distinct } xs\ y \notin \text{set } xs\ y \notin \text{set } ys$

**using** *not-distinct-decomp-min-prefix* **by** *blast*

**obtain**  $q\ p'$  **where**  $\text{cas } u\ q\ y\ p = q @ p'$   $\text{awalk-verts } u\ q = xs @ [y]$

**and** *p'-props*:  $\text{cas } y\ p'\ v\ \text{awalk-verts } y\ p' = (y \# ys) @ y \# zs$

**using** *assms pv-decomp* **by** – (*rule awalk-decomp-verts, auto*)

**obtain**  $r\ s$  **where**  $\text{cas } y\ r\ y\ \text{cas } y\ s\ v\ p' = r @ s$

$\text{awalk-verts } y\ r = y \# ys @ [y]\ \text{awalk-verts } y\ s = y \# zs$

**using** *p'-props* **by** (*rule awalk-decomp-verts*) *auto*

**have**  $p = q @ r @ s$  **using**  $\langle p = q @ p' \rangle \langle p' = r @ s \rangle$  **by** *simp*

**moreover**

**have**  $\text{distinct } (\text{awalk-verts } u\ q)$  **using**  $\langle \text{awalk-verts } u\ q = xs @ [y] \rangle$  **and** *xs-y-props*

**by** *simp*

**moreover**

**have**  $0 < \text{length } r$  **using**  $\langle \text{awalk-verts } y\ r = y \# ys @ [y] \rangle$  **by** *auto*

**moreover**

**from** *pv-decomp assms* **have**  $y \in \text{verts } G$  **by** *auto*

**then** **have**  $\text{awalk } u\ q\ y\ \text{awalk } y\ r\ y\ \text{awalk } y\ s\ v$

**using**  $\langle \text{awalk } u\ p\ v \rangle \langle \text{cas } u\ q\ y \rangle \langle \text{cas } y\ r\ y \rangle \langle \text{cas } y\ s\ v \rangle$  **unfolding**  $\langle p = q @ r @ s \rangle$

**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**  $x \in \text{set } (\text{awalk-verts } u\ q)\ x \in \text{set } (\text{tl } (\text{awalk-verts } (\text{awlast } u\ q)\ r))$

**shows** *False*

**using** *assms* **by** (*auto simp: apath-def awalk-verts-append*)

### 6.3 Cycles

**definition** *closed-w* :: '*b* *awalk*  $\Rightarrow$  *bool* **where**

*closed-w*  $p \equiv \exists u. \text{awalk } u\ p\ u \wedge 0 < \text{length } p$

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

The definition given here matches [?]. [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*  $\Rightarrow$  *bool* **where**

*cycle*  $p \equiv \exists u. \text{awalk } u\ p\ u \wedge \text{distinct } (\text{tl } (\text{awalk-verts } u\ p)) \wedge p \neq []$

**lemma** *cycle-altdef*:

*cycle*  $p \iff \text{closed-w } p \wedge (\exists u. \text{distinct } (\text{tl } (\text{awalk-verts } u\ p)))$

**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*  $(\text{tl } (\text{awalk-verts } u\ p))$

**shows** *distinct*  $p$

**proof** (*rule ccontr*)

**assume**  $\neg \text{distinct } p$

**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*  $(\text{awalk-verts } u\ p)$

**shows** *distinct*  $p$

**using** *assms* **by** (*blast intro: distinct-tl-verts-imp-distinct distinct-tl*)

**lemma** (**in** *wf-digraph*) *cycle-conv*:

*cycle*  $p \iff (\exists u. \text{awalk } u\ p\ u \wedge \text{distinct } (\text{tl } (\text{awalk-verts } u\ p)) \wedge \text{distinct } p \wedge p \neq [])$

[])

**unfolding** *cycle-def* **by** (*auto intro: distinct-tl-verts-imp-distinct*)

**lemma** (in *loopfree-digraph*) *cycle-digraph-conv*:  
 $cycle\ p \longleftrightarrow (\exists u. awalk\ u\ p\ u \wedge distinct\ (tl\ (awalk\ -verts\ u\ p)) \wedge 2 \leq length\ p)$   
(is  $?L \longleftrightarrow ?R$ )  
**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**  $2 \leq length\ p$   
**proof** (rule *ccontr*)  
**assume**  $\neg ?thesis$  **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**  $\exists p. cycle\ p$   
**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*  $\exists w. awalk\ w\ r\ w\ closed-w\ r$   
**using** *awalk-not-distinct-decomp[OF \*\*]* **by** (auto simp: *closed-w-def*)  
**then have**  $length\ r < length\ p$  **using**  $\langle p = \rightarrow \rangle$  **by** *auto*  
**then show**  $?thesis$  **using**  $\langle closed-w\ r \rangle$  **by** (rule *less*)  
**qed**  
**qed**

## 6.4 Reachability

**lemma** *reachable1-awalk*:  
 $u \rightarrow^+ v \longleftrightarrow (\exists p. awalk\ u\ p\ v \wedge p \neq [])$   
**proof**  
**assume**  $u \rightarrow^+ v$  **then show**  $\exists p. awalk\ u\ p\ v \wedge p \neq []$   
**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**  $awalk\ y\ p\ v\ p \neq []$  **by** *auto*  
**moreover**  
**from**  $\langle x \rightarrow y \rangle$  **obtain**  $e$  **where**  $tail\ G\ e = x\ head\ G\ e = y\ e \in arcs\ G$   
**by** *auto*  
**ultimately**  
**have**  $awalk\ x\ (e \# p)\ v$   
**by** (*auto simp: awalk-Cons-iff*)  
**then show** *?case* **by** *auto*  
**qed**  
**next**  
**assume**  $\exists p. awalk\ u\ p\ v \wedge p \neq []$  **then obtain**  $p$  **where**  $awalk\ u\ p\ v\ p \neq []$  **by**  
*auto*  
**thus**  $u \rightarrow^+ v$   
**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*:  
 $u \rightarrow^* v \longleftrightarrow (\exists p. awalk\ u\ p\ v)$   
**proof** *cases*  
**assume**  $u = v$   
**have**  $u \rightarrow^* u \longleftrightarrow awalk\ u\ []\ u$  **by** (*auto simp: awalk-Nil-iff reachable-in-verts*)  
**also have**  $\dots \longleftrightarrow (\exists p. awalk\ u\ p\ u)$   
**by** (*metis awalk-Nil-iff awalk-hd-in-verts*)  
**finally show** *?thesis* **using**  $\langle u = v \rangle$  **by** *simp*  
**next**  
**assume**  $u \neq v$   
**then have**  $u \rightarrow^* v \longleftrightarrow u \rightarrow^+ v$  **by** *auto*  
**also have**  $\dots \longleftrightarrow (\exists p. awalk\ u\ p\ v)$   
**using**  $\langle u \neq v \rangle$  **unfolding** *reachable1-awalk* **by** *force*  
**finally show** *?thesis* .  
**qed**

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

**lemma** *reachable1-awalkI*:  
 $awalk\ v\ p\ w \implies p \neq [] \implies v \rightarrow^+ w$   
**by** (*auto simp add: reachable1-awalk*)

**lemma** *reachable-arc-trans*:  
**assumes**  $u \rightarrow^* v\ arc\ e\ (v, w)$

**shows**  $u \rightarrow^* w$   
**proof** –  
**from**  $\langle u \rightarrow^* v \rangle$  **obtain**  $p$  **where**  $awalk\ u\ p\ v$   
**by**  $(auto\ simp:\ reachable-awalk)$   
**moreover have**  $awalk\ v\ [e]\ w$   
**using**  $\langle arc\ e\ (v,w) \rangle$   
**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**  $awalk\ u\ p\ v\ w \in set\ (awalk-verts\ u\ p)$  **shows**  $u \rightarrow^*_G w$   
**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**  $awalk\ u\ p\ v\ w \in set\ (awalk-verts\ u\ p)$  **shows**  $w \rightarrow^*_G v$   
**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**  $\langle apath\ u\ p\ v \rangle$  **by**  $(auto\ simp:\ apath-def\ distinct-card)$   
**also have**  $\dots \leq card\ (verts\ G)$   
**using**  $\langle apath\ u\ p\ v \rangle$  **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**  $length\ p \leq card\ (verts\ G)$   
**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**  $\bigwedge u p v. \text{awalk } u p v \implies \text{distinct } (\text{awalk-verts } u p) \implies \text{length } p \leq \text{card } (\text{verts } G)$   
**by** (*rule length-apath*) (*auto simp: apath-def*)  
**then have**  $\{(u,p,v). \text{apath } u p v\} \subseteq \text{verts } G \times \{\text{es. set } es \subseteq \text{arcs } G \wedge \text{length } es \leq \text{card } (\text{verts } G)\} \times \text{verts } G$   
**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*  $\{p. \text{apath } u p v\}$

**proof** –

**have**  $\{p. \text{apath } u p v\} \subseteq (\text{fst } o \text{snd}) \text{ ` } \{(u,p,v). \text{apath } u p v\}$

**by** *force*

**with** *apaths-finite-triple* **show** *?thesis* **by** (*rule finite-surj*)

**qed**

**fun** *is-awalk-cyc-decomp* :: *'b awalk* =>

*('b awalk*  $\times$  *'b awalk*  $\times$  *'b awalk*)  $\Rightarrow$  *bool* **where**

*is-awalk-cyc-decomp*  $p (q,r,s) \longleftrightarrow p = q @ r @ s$

$\wedge (\exists u v w. \text{awalk } u q v \wedge \text{awalk } v r v \wedge \text{awalk } v s w)$

$\wedge 0 < \text{length } r$

$\wedge (\exists u. \text{distinct } (\text{awalk-verts } u q))$

**definition** *awalk-cyc-decomp* :: *'b awalk*

$\Rightarrow$  *'b awalk*  $\times$  *'b awalk*  $\times$  *'b awalk* **where**

*awalk-cyc-decomp*  $p = (\text{SOME } qrs. \text{is-awalk-cyc-decomp } p qrs)$

**function** *awalk-to-apath* :: *'b awalk*  $\Rightarrow$  *'b awalk* **where**

*awalk-to-apath*  $p = (\text{if } \neg(\exists u. \text{distinct } (\text{awalk-verts } u p)) \wedge (\exists u v. \text{awalk } u p v)$

*then* (*let*  $(q,r,s) = \text{awalk-cyc-decomp } p$  *in* *awalk-to-apath*  $(q @ s)$ )

*else*  $p$ )

**by** *auto*

**lemma** *awalk-cyc-decomp-has-prop*:

**assumes** *awalk*  $u p v$  **and**  $\neg \text{distinct } (\text{awalk-verts } u p)$

**shows** *is-awalk-cyc-decomp*  $p (\text{awalk-cyc-decomp } p)$

**proof** –

**obtain**  $q r s$  **where**  $*$ :  $p = q @ r @ s \wedge \text{distinct } (\text{awalk-verts } u q)$

$\wedge 0 < \text{length } r$

$\wedge (\exists w. \text{awalk } u q w \wedge \text{awalk } w r w \wedge \text{awalk } w s v)$

**by** (*atomize-elim*) (*rule awalk-not-distinct-decomp[OF assms]*)

**then have**  $\exists x. \text{is-awalk-cyc-decomp } p x$

**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*  $u\ p\ v \neg distinct\ (awalk-verts\ u\ p)$

**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**  $\langle p = q @ r @ s \rangle$  **by** *simp*

**then have**  $u = u'$  **and**  $v = v'$  **using**  $\langle p \neq [] \rangle \langle awalk\ u\ p\ v \rangle$  **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-apat*

**proof** (*relation measure length*)

**fix**  $G\ p\ qrs\ rs\ q\ r\ s$

**have**  $X: \bigwedge x\ y. closed-w\ r \implies awalk\ x\ r\ y \implies x = y$

**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  $(q @ s, p) \in \text{measure length}$ 
  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:  $\bigwedge p. \text{awalk } u \ p \ v \implies \text{distinct } (\text{awalk-verts } u \ p) \implies P \ p$ 
  assumes dec:  $\bigwedge p \ q \ r \ s. \llbracket \text{awalk } u \ p \ v; \text{awalk-cyc-decomp } p = (q, r, s);$ 
     $\neg \text{distinct } (\text{awalk-verts } u \ p); P \ (q @ s) \rrbracket \implies P \ p$ 
  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  $(q @ s) < \text{length } p$  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:  $\neg \text{distinct } (\text{awalk-verts } u \ p)$ 
  shows awalk-to-apath  $p = \text{awalk-to-apath } (q @ s)$ 
proof -
  from dist have  $\neg(\exists u. \text{distinct } (\text{awalk-verts } u \ p))$ 
    by (auto simp: awalk-verts-conv)
  with awalk and decomp show awalk-to-apath  $p = \text{awalk-to-apath } (q @ s)$ 
    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)  $\subseteq$  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 *:  $\neg(\exists u. \text{distinct } (\text{awalk-verts } u \ p)) \wedge (\exists u \ v. \text{awalk } u \ p \ v)$ 
    using decomp by (cases p) auto
  have set (awalk-to-apath (q @ s))  $\subseteq$  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:
   $u \rightarrow^* v \iff (\exists p. \text{apath } u \ p \ v)$ 
  by (auto intro: awalkI-apath apath-awalk-to-apath simp: reachable-awalk)

```

```

lemma no-loops-in-apath:
  assumes apath u p v a  $\in$  set p shows tail G a  $\neq$  head G a
proof -
  from  $\langle a \in \text{set } p \rangle$  obtain p1 p2 where  $p = p1 \ @ \ a \ \# \ p2$  by (auto simp:
in-set-conv-decomp)
  with  $\langle \text{apath } u \ p \ v \rangle$  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  $\Rightarrow$  ('a, 'a  $\times$  'a) pre-digraph where
  with-proj  $G$  = ( $\lambda$  verts = pverts  $G$ , arcs = parcs  $G$ , tail = fst, head = snd  $\lambda$ )
```

```
declare [[coercion with-proj]]
```

```
primrec pawalk-verts :: 'a  $\Rightarrow$  ('a  $\times$  'a) awalk  $\Rightarrow$  'a list where
  pawalk-verts  $u$  [] = [ $u$ ] |
  pawalk-verts  $u$  ( $e$  #  $es$ ) = fst  $e$  # pawalk-verts (snd  $e$ )  $es$ 
```

```
fun pcas :: 'a  $\Rightarrow$  ('a  $\times$  'a) awalk  $\Rightarrow$  'a  $\Rightarrow$  bool where
  pcas  $u$  []  $v$  = ( $u$  =  $v$ ) |
  pcas  $u$  ( $e$  #  $es$ )  $v$  = (fst  $e$  =  $u$   $\wedge$  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*:  $\bigwedge e. e \in \text{parcs } G \implies \text{fst } e \in \text{pverts } G$   
  **assumes** *arc-snd-in-verts*:  $\bigwedge e. e \in \text{parcs } G \implies \text{snd } e \in \text{pverts } G$   
**begin**

**lemma** *in-arcsD1*:  $(u,v) \in \text{parcs } G \implies u \in \text{pverts } G$   
  **and** *in-arcsD2*:  $(u,v) \in \text{parcs } G \implies v \in \text{pverts } G$   
  **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*:  $e \in \text{parcs } G \implies \text{fst } e \neq \text{snd } e$

**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*  $\subseteq$  *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*  $\subseteq$  *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*  $\subseteq$  *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*  $\subseteq$  *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*  $\subseteq$  *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*  $\subseteq$  *digraph with-proj*  $G$

**rewrites**  $verts\ G = pverts\ G$  **and**  $arcs\ G = parcs\ G$  **and**  $tail\ G = fst$  **and**  $head\ G = snd$   
**and**  $arcs\text{-}ends\ G = parcs\ G$   
**and**  $pre\text{-}digraph.\text{awalk}\text{-}verts\ G = \text{pawalk}\text{-}verts$   
**and**  $pre\text{-}digraph.\text{cas}\ G = \text{pcas}$   
**by**  $unfold\text{-}locales$  ( $auto\ simp: arc\text{-}to\text{-}ends\text{-}def$ )

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

**sublocale**  $pair\text{-}graph \subseteq pair\text{-}bidirected\text{-}digraph$  **by**  $unfold\text{-}locales$

**lemma**  $wf\text{-}digraph\text{-}wp\text{-}iff$ :  $wf\text{-}digraph\ (with\text{-}proj\ G) = pair\text{-}wf\text{-}digraph\ G$  (**is**  $?L \longleftrightarrow ?R$ )

**proof**

**assume**  $?L$  **then interpret**  $wf\text{-}digraph\ with\text{-}proj\ G$  .

**show**  $?R$  **using**  $wellformed$  **by**  $unfold\text{-}locales\ auto$

**next**

**assume**  $?R$  **then interpret**  $pair\text{-}wf\text{-}digraph\ G$  .

**show**  $?L$  **by**  $unfold\text{-}locales$

**qed**

**lemma** (**in**  $pair\text{-}fin\text{-}digraph$ )  $pair\text{-}fin\text{-}digraph[intro!]$ :  $pair\text{-}fin\text{-}digraph\ G ..$

**context**  $pair\text{-}digraph$  **begin**

**lemma**  $pair\text{-}wf\text{-}digraph[intro!]$ :  $pair\text{-}wf\text{-}digraph\ G$  **by**  $intro\text{-}locales$

**lemma**  $pair\text{-}digraph[intro!]$ :  $pair\text{-}digraph\ G ..$

**lemma** (**in**  $pair\text{-}loopfree\text{-}digraph$ )  $no\text{-}loops'$ :

$(u,v) \in parcs\ G \implies u \neq v$

**by** ( $auto\ dest: no\text{-}loops$ )

**end**

**lemma** (**in**  $pair\text{-}wf\text{-}digraph$ )  $apath\text{-}succ\text{-}decomp$ :

**assumes**  $apath\ u\ p\ v$

**assumes**  $(x,y) \in set\ p$

**assumes**  $y \neq v$

**shows**  $\exists p1\ z\ p2. p = p1 @ (x,y) \# (y,z) \# p2 \wedge x \neq z \wedge y \neq z$

**proof** –

**from**  $\langle (x,y) \in set\ p \rangle$  **obtain**  $p1\ p2$  **where**  $p\text{-}decomp$ :  $p = p1 @ (x,y) \# p2$

**by** ( $metis\ (no\text{-}types)\ in\text{-}set\text{-}conv\text{-}decomp\text{-}first$ )

**from**  $p$ -decomp  $\langle$ apath  $u$   $p$   $v$   $\rangle$   $\langle$  $y \neq v$   $\rangle$  **have**  $p2 \neq []$   $awalk$   $y$   $p2$   $v$   
**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**  $x \neq z \wedge y \neq z$  **using**  $p$ -decomp  $p2$ -decomp  $\langle$ apath  $u$   $p$   $v$   $\rangle$   
**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  $G \implies (b,a) \in$  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 ::  $('a \times 'a)$   $awalk \implies ('a \times 'a)$   $awalk$  **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]*:  
 $rev\text{-}path\ p = [] \longleftrightarrow p = []$   
**by** (*induct p*) *auto*

**lemma** *rev-path-eq*:  $rev\text{-}path\ p = rev\text{-}path\ q \longleftrightarrow p = q$   
**by** (*metis rev-path-rev-path*)

**lemma** (*in pair-sym-digraph*)  
**assumes** *awalk u p v*  
**shows** *awalk-verts-rev-path*:  $awalk\text{-}verts\ v\ (rev\text{-}path\ p) = rev\ (awalk\text{-}verts\ u\ p)$   
**and** *awalk-rev-path'*:  $awalk\ v\ (rev\text{-}path\ p)\ u$   
**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**:  $awalk\ v\ (rev\text{-}path\ es)\ (snd\ e)$   
 $awalk\ (snd\ e)\ [(snd\ e,\ fst\ e)]\ u$   
**and** *verts*:  $awalk\text{-}verts\ v\ (rev\text{-}path\ es) = rev\ (awalk\text{-}verts\ (snd\ e)\ es)$   
**by** (*auto simp: awalk-simps intro: arcs-symmetric*)

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

**lemma** (*in pair-sym-digraph*) *awalk-rev-path[simp]*:  
 $awalk\ v\ (rev\text{-}path\ p)\ u = awalk\ u\ p\ v$  (**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*  $\Rightarrow$  '*a*  $\times$  '*a*  $\Rightarrow$  '*a*  $\Rightarrow$  '*a* *pair-pre-digraph* **where**  
*subdivide G (u,v) w = []*  
*pverts = pverts G  $\cup$  {w},*  
*parcs = (parcs G - {(u,v),(v,u)})  $\cup$  {(u,w), (w,u), (w,v), (v,w)}*

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

subdivide an arc in a path

**fun** *sd-path* :: '*a*  $\times$  '*a*  $\Rightarrow$  '*a*  $\Rightarrow$  ('*a*  $\times$  '*a*) *awalk*  $\Rightarrow$  ('*a*  $\times$  '*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*  $\times$  '*a*  $\Rightarrow$  '*a*  $\Rightarrow$  ('*a*  $\times$  '*a*) *awalk*  $\Rightarrow$  ('*a*  $\times$  '*a*) *awalk* **where**  
*co-path - - [] = []*  
| *co-path - - [e] = [e]*  
| *co-path (u,v) w (e1 # e2 # es) = (if e1 = (u,w)  $\wedge$  e2 = (w,v)*  
*then (u,v) # co-path (u,v) w es*  
*else if e1 = (v,w)  $\wedge$  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*]:

$\llbracket e1 \neq (fst\ e, w); e1 \neq (snd\ e, w) \rrbracket \Longrightarrow co-path\ e\ w\ (e1\ \#\ es) = e1\ \#\ co-path\ e\ w\ es$

$\llbracket e1 = (fst\ e, w); e2 = (w, snd\ e) \rrbracket \Longrightarrow co-path\ e\ w\ (e1\ \#\ e2\ \#\ es) = e\ \#\ co-path\ e\ w\ es$

$\llbracket e1 = (snd\ e, w); e2 = (w, fst\ e) \rrbracket$

$\Longrightarrow co-path\ e\ w\ (e1\ \#\ e2\ \#\ es) = (snd\ e, fst\ e)\ \#\ co-path\ e\ w\ es$

$\llbracket e1 \neq (fst\ e, w) \vee e2 \neq (w, snd\ e); e1 \neq (snd\ e, w) \vee e2 \neq (w, fst\ e) \rrbracket$

$\Longrightarrow co-path\ e\ w\ (e1\ \#\ e2\ \#\ es) = e1\ \#\ co-path\ e\ w\ (e2\ \#\ es)$

**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]:  $co\text{-}path\ e\ w\ p = [] \longleftrightarrow p = []$   
**by** (*cases e*) (*cases p rule: list-exhaust-NSC, auto*)

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

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

**lemma** *arcs-subdivide*[simp]:  
**shows**  $parcs\ (subdivide\ G\ (u,v)\ w) = (parcs\ G - \{(u,v),(v,u)\}) \cup \{(u,w), (w,u), (w,v), (v,w)\}$   
**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**  $B: \bigwedge e' es. e' \neq e \implies e' \neq (snd\ e, fst\ e) \implies P\ e\ es \implies P\ e\ (e' \# es)$   
 $\bigwedge es. P\ e\ es \implies P\ e\ (e \# es)$   
 $\bigwedge es. fst\ e \neq snd\ e \implies P\ e\ es \implies P\ e\ ((snd\ e, fst\ e) \# es)$   
**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**  $e :: 'a \times 'a$   
**and**  $w :: 'a$   
**and**  $p :: ('a \times 'a) \text{ awalk}$   
**assumes**  $Nil: P\ e\ w\ []$   
**and**  $ConsNil: \bigwedge e'. P\ e\ w\ [e']$   
**and**  $ConsCons1: \bigwedge e1\ e2\ es. e1 = (fst\ e, w) \wedge e2 = (w, snd\ e) \implies P\ e\ w\ es$   
 $\implies$   
 $P\ e\ w\ (e1 \# e2 \# es)$   
**and**  $ConsCons2: \bigwedge e1\ e2\ es. \neg(e1 = (fst\ e, w) \wedge e2 = (w, snd\ e)) \wedge$   
 $e1 = (snd\ e, w) \wedge e2 = (w, fst\ e) \implies P\ e\ w\ es \implies$   
 $P\ e\ w\ (e1 \# e2 \# es)$   
**and**  $ConsCons3: \bigwedge e1\ e2\ es.$   
 $\neg(e1 = (fst\ e, w) \wedge e2 = (w, snd\ e)) \implies$   
 $\neg(e1 = (snd\ e, w) \wedge e2 = (w, fst\ e)) \implies P\ e\ w\ (e2 \# es) \implies$   
 $P\ e\ w\ (e1 \# e2 \# es)$   
**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**  $(u,w) \notin \text{set } p$   $(v,w) \notin \text{set } p$   
**shows**  $\text{co-path } (u,v) \ w \ (\text{sd-path } (u,v) \ w \ p) = p$   
**using** *assms* **by** (*induct p*) *auto*

**lemma** *sd-path-id*:  
**assumes**  $(x,y) \notin \text{set } p$   $(y,x) \notin \text{set } p$   
**shows**  $\text{sd-path } (x,y) \ w \ p = p$   
**using** *assms* **by** (*induct p*) *auto*

**lemma** (*in pair-wf-digraph*) *pair-wf-digraph-subdivide*:  
**assumes** *props*:  $e \in \text{parcs } G$   $w \notin \text{pverts } G$   
**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**  $e' \in \text{parcs } ?sG$   
**then show**  $\text{fst } e' \in \text{pverts } ?sG$   $\text{snd } e' \in \text{pverts } ?sG$   
**using** *props* **by** (*auto dest: wellformed*)  
**qed**

**lemma** (*in pair-sym-digraph*) *pair-sym-digraph-subdivide*:  
**assumes** *props*:  $e \in \text{parcs } G$   $w \notin \text{pverts } G$   
**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**  $\bigwedge a \ b. (a, b) \in \text{parcs } (\text{subdivide } G \ e \ w) \implies (b, a) \in \text{parcs } (\text{subdivide } G \ e \ w)$   
**unfolding**  $\langle e = \cdot \rangle \text{ arcs-subdivide}$   
**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*:  $e \in \text{parcs } G$   $w \notin \text{pverts } G$   
**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*:  $e \in \text{parcs } G \ w \notin \text{pverts } G$   
 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*:  $e \in \text{parcs } G \ w \notin \text{pverts } G$   
 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  $e \in \text{parcs } G \ w \notin \text{pverts } G$   
 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  $x \in \text{parcs} \ ( \text{subdivide } G \ e \ w) \ \text{fst } x \neq w \ \text{snd } x \neq w$   
 shows  $x \in \text{parcs } G$   
 using *assms* by (cases  $e$ ) *auto*

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

**lemma**  
 assumes *path*: *apath*  $u \ p \ v$   
 assumes *elems*:  $e \in \text{parcs } G \ w \notin \text{pverts } G$   
 shows *apath-sd-path*: *pre-digraph.apath* (subdivide  $G \ e \ w$ )  $u \ (\text{sd-path } e \ w \ p) \ v$  (is  
 ?A)  
 and *set-awalk-verts-sd-path*: *set* (*awalk-verts*  $u \ (\text{sd-path } e \ w \ p)$ )  
 $\subseteq \text{set } (\text{awalk-verts } u \ p) \cup \{w\}$  (is ?B)  
**proof** –  
 obtain  $x \ y$  where *e-conv*:  $e = (x, y)$  by (cases  $e$ ) *auto*  
 define *sG* where  $sG = \text{subdivide } G \ e \ w$   
 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:  $\{(x,w), (y,w), (w,x), (w,y)\} \subseteq \text{parcs } sG$ 
  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))  $\subseteq \text{set } (\text{awalk-verts } u p) \cup \{w\}$ 
proof (induct p arbitrary: u rule: sd-path-induct)
  case empty case 1
  moreover have pverts sG = pverts G  $\cup \{w\}$  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  $u \neq w$  fst e' = u
       $u \notin \text{set } (\text{S.awalk-verts } (\text{snd } e') (\text{sd-path } (x,y) w es))$ 
      using pass elems by (fastforce simp: apath-Cons-iff)+
      moreover then have  $e' \in \text{parcs } sG$ 
      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:  $(a,b) = (x,y) \vee (a,b) = (y,x)$ 
      and hyps:  $\bigwedge u. \text{apath } u \text{ es } v \implies \text{S.apath } u (\text{sd-path } (x,y) w es) v$ 
         $\bigwedge u. \text{apath } u \text{ es } v \implies \text{set } (\text{awalk-verts } u (\text{sd-path } (x,y) w es)) \subseteq \text{set } (\text{awalk-verts } u es) \cup \{w\}$ 

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

      from A ab have [simp]:  $x \neq y$ 
        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  $u = a$   $u \neq w$ 
        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  $u \notin \text{set } (S.\text{awalk-verts } w ((w,b) \# \text{sd-path } (x,y) w \text{ es}))$ 
  using  $ab \langle u \neq w \rangle \text{ ev-sd } A$  by (auto simp:  $\text{apath-Cons-iff } S.\text{awalk-def}$ )
moreover
have  $w \notin \text{set } (\text{awalk-verts } b (\text{sd-path } (x, y) w \text{ es}))$ 
  using  $ab \text{ ev-sd } A \text{ elems}$  by (auto simp:  $\text{awalk-Cons-iff } \text{apath-def}$ )
ultimately
have  $\text{path}: S.\text{apath } u (\text{sd-path } (x, y) w ((a, b) \# \text{es})) v$ 
  using  $ab \text{ hyps } w\text{-s}G \langle u = a \rangle$  by (auto simp:  $S.\text{apath-Cons-iff}$  ) }
note  $\text{path} = \text{this}$ 
{ case (sd es)
  { case 1 with sd show ?case by (intro path) auto }
  { case 2 show ?case using 2 sd
    by (auto simp:  $\text{apath-Cons-iff}$ ) } }
{ case (sdrev es)
  { case 1 with sdrev show ?case by (intro path) auto }
  { case 2 show ?case using 2 sdrev
    by (auto simp:  $\text{apath-Cons-iff}$ ) } }
qed
then show ?A ?B unfolding  $sG\text{-def } e\text{-conv}$  .
qed

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

have  $e\text{-w}: \text{fst } e \neq w \ \text{snd } e \neq w$  using  $\text{elems}$  by auto

have  $S.\text{apath } u \ p \ v \ u \neq w$  using  $\text{elems path}$  by auto
then have  $\text{co-path}: \text{apath } u (\text{co-path } e \ w \ p) \ v$ 
   $\wedge \text{set } (\text{awalk-verts } u (\text{co-path } e \ w \ p)) = \text{set } (\text{awalk-verts } u \ p) - \{w\}$ 
proof (induction p arbitrary: u rule:  $\text{co-path-induct}$ )
  case empty with  $\text{elems}$  show ?case
    by (simp add:  $\text{apath-Nil-iff } S.\text{apath-Nil-iff}$ )
next
  case (single e') with  $\text{elems}$  show ?case
    by (auto simp:  $\text{apath-Cons-iff } S.\text{apath-Cons-iff } \text{apath-Nil-iff } S.\text{apath-Nil-iff}$ 
       $\text{dest: arcs-subdivideD}$ )
next
  case (co e1 e2 es)
then have  $\text{apath } u (\text{co-path } e \ w (e1 \# \ e2 \# \ es)) \ v$  using  $\text{co } e\text{-w } \text{elems}$ 
  by (auto simp:  $\text{apath-Cons-iff } S.\text{apath-Cons-iff}$ )
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.prem1 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.prem2 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.prem3 by (cases es) (auto simp: S.apath-Cons-iff)
  qed
  then have e1 ∈ parcs G
  using ⟨fst e1 ≠ w⟩ pass.prem1 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.prem1 ⟨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.prem1 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* ::  $('a \times 'a) \text{ set} \Rightarrow 'a \times 'a \Rightarrow 'a \times 'a$  **where**  
*swap-in*  $S$   $x = (\text{if } x \in S \text{ then } \text{prod.swap } x \text{ else } x)$

**lemma** *bidirected-digraph-rev-conv-pair*:  
**assumes** *bidirected-digraph* (*with-proj*  $G$ ) *rev-G*  
**shows**  $\text{rev-G} = \text{swap-in } (\text{parcs } G)$   
**proof** –  
**interpret** *bidirected-digraph*  $G$  *rev-G* **by fact**  
**have**  $\bigwedge a b. (a, b) \in \text{parcs } G \Longrightarrow \text{rev-G } (a, b) = (b, a)$   
**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**  $a \in \text{parcs } G$  **then have**  $\text{fst } a \neq \text{snd } a$   
**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  $\Rightarrow$  ('a,'b) pre-digraph  $\Rightarrow$  bool **where**  
*compatible* G H  $\equiv$  tail G = tail H  $\wedge$  head G = head H

**definition** *subgraph* :: ('a,'b) pre-digraph  $\Rightarrow$  ('a,'b) pre-digraph  $\Rightarrow$  bool **where**  
*subgraph* H G  $\equiv$  verts H  $\subseteq$  verts G  $\wedge$  arcs H  $\subseteq$  arcs G  $\wedge$  wf-digraph G  $\wedge$   
wf-digraph H  $\wedge$  compatible G H

**definition** *induced-subgraph* :: ('a,'b) pre-digraph  $\Rightarrow$  ('a,'b) pre-digraph  $\Rightarrow$  bool **where**  
**where**  
*induced-subgraph* H G  $\equiv$  subgraph H G  $\wedge$  arcs H = {e  $\in$  arcs G. tail G e  $\in$  verts  
H  $\wedge$  head G e  $\in$  verts H}

**definition** *spanning* :: ('a,'b) pre-digraph  $\Rightarrow$  ('a,'b) pre-digraph  $\Rightarrow$  bool **where**  
*spanning* H G  $\equiv$  subgraph H G  $\wedge$  verts G = verts H

**definition** *strongly-connected* :: ('a,'b) pre-digraph  $\Rightarrow$  bool **where**  
*strongly-connected* G  $\equiv$  verts G  $\neq$  {}  $\wedge$  ( $\forall u \in$  verts G.  $\forall v \in$  verts G.  $u \rightarrow^*_G v$ )

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

**definition** *mk-symmetric* :: ('a,'b) pre-digraph  $\Rightarrow$  'a pair-pre-digraph **where**  
*mk-symmetric* G  $\equiv$  ( $\emptyset$  pverts = verts G, parcs =  $\bigcup_{e \in$  arcs G. {(tail G e, head G  
e), (head G e, tail G e)}))

**definition** *connected* :: ('a,'b) pre-digraph  $\Rightarrow$  bool **where**  
*connected* G  $\equiv$  strongly-connected (mk-symmetric G)

**definition** *forest* :: ('a,'b) pre-digraph  $\Rightarrow$  bool **where**  
*forest* G  $\equiv$   $\neg(\exists p.$  pre-digraph.cycle G p)

**definition** *tree* :: ('a,'b) pre-digraph  $\Rightarrow$  bool **where**  
*tree* G  $\equiv$  connected G  $\wedge$  forest G

**definition** *spanning-tree* :: ('a,'b) pre-digraph  $\Rightarrow$  ('a,'b) pre-digraph  $\Rightarrow$  bool **where**  
*spanning-tree* H G  $\equiv$  tree H  $\wedge$  spanning H G

**definition** (in pre-digraph)

*max-subgraph* :: (*'a, 'b*) *pre-digraph*  $\Rightarrow$  *bool*  $\Rightarrow$  (*'a, 'b*) *pre-digraph*  $\Rightarrow$  *bool*  
**where**

*max-subgraph* *P H*  $\equiv$  *subgraph H G*  $\wedge$  *P H*  $\wedge$  ( $\forall H'. H' \neq H \wedge$  *subgraph H H'*  
 $\longrightarrow \neg(\text{subgraph } H' G \wedge P H')$ )

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

*sccs*  $\equiv$  {*H. induced-subgraph H G*  $\wedge$  *strongly-connected H*  $\wedge$   $\neg(\exists H'. \text{induced-subgraph } H' G$   
 $\wedge$  *strongly-connected H'*  $\wedge$  *verts H*  $\subset$  *verts H')}*

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

*sccs-verts* = {*S. S*  $\neq$  {}  $\wedge$  ( $\forall u \in S. \forall v \in S. u \rightarrow_G^* v$ )  $\wedge$  ( $\forall u \in S. \forall v. v \notin S$   
 $\longrightarrow \neg u \rightarrow_G^* v \vee \neg v \rightarrow_G^* u$ )}

**definition** (in *pre-digraph*) *scc-of* :: *'a*  $\Rightarrow$  *'a set* **where**

*scc-of u*  $\equiv$  {*v. u*  $\rightarrow_G^* v$   $\wedge$  *v*  $\rightarrow_G^* u$ }

**definition** *union* :: (*'a, 'b*) *pre-digraph*  $\Rightarrow$  (*'a, 'b*) *pre-digraph*  $\Rightarrow$  (*'a, 'b*) *pre-digraph*  
**where**

*union G H*  $\equiv$  ( $\lfloor$  *verts* = *verts G*  $\cup$  *verts H*, *arcs* = *arcs G*  $\cup$  *arcs H*, *tail* = *tail G*,  
*head* = *head G*  $\rfloor$ )

**definition** (in *pre-digraph*) *Union* :: (*'a, 'b*) *pre-digraph set*  $\Rightarrow$  (*'a, 'b*) *pre-digraph*  
**where**

*Union gs* = ( $\lfloor$  *verts* = ( $\bigcup G \in gs. \text{verts } G$ ), *arcs* = ( $\bigcup G \in gs. \text{arcs } G$ ),  
*tail* = *tail G*, *head* = *head G*  $\rfloor$ )

## 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*

*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**  $(u,v) \in \text{arcs-ends } G \implies (v,u) \in \text{arcs-ends } G$   
**using** *sym-arcs* **by** (*auto simp add: symmetric-def sym-def*)

**lemma** *strongly-connectedI[intro]*:  
**assumes**  $\text{verts } G \neq \{\}$   $\bigwedge u v. u \in \text{verts } G \implies v \in \text{verts } G \implies u \rightarrow^*_G v$   
**shows** *strongly-connected G*  
**using** *assms* **by** (*simp add: strongly-connected-def*)

**lemma** *strongly-connectedE[elim]*:  
**assumes** *strongly-connected G*  
**assumes**  $(\bigwedge u v. u \in \text{verts } G \wedge v \in \text{verts } G \implies u \rightarrow^*_G v) \implies P$   
**shows** *P*  
**using** *assms* **by** (*auto simp add: strongly-connected-def*)

**lemma** *subgraph-imp-subverts*:  
**assumes** *subgraph H G*  
**shows**  $\text{verts } H \subseteq \text{verts } G$   
**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**  $c \in \text{sccs}$   
**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  $\wedge$  spanning H G  $\implies$  P*  
**shows** *P*  
**using** *assms* **by** (*simp add: spanning-tree-def*)

**lemma** *spanningE*[*elim*]:  
**assumes** *spanning H G*  
**assumes** *subgraph H G  $\wedge$  verts G = verts H  $\implies$  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**  $\neg(\exists c'. \textit{induced-subgraph } c' G \wedge \textit{strongly-connected } c' \wedge \textit{verts } c \subset \textit{verts } c')$   
**shows**  $c \in \textit{sccs}$   
**using** *assms* **by** (*auto simp add: sccs-def*)

**lemma** (**in** *pre-digraph*) *in-sccsE*[*elim*]:  
**assumes**  $c \in \textit{sccs}$   
**assumes** *induced-subgraph c G  $\implies$  strongly-connected c  $\implies$   $\neg(\exists d. \textit{induced-subgraph } d G \wedge \textit{strongly-connected } d \wedge \textit{verts } c \subset \textit{verts } d)$*   
**shows** *P*  
**using** *assms* **by** (*simp add: sccs-def*)

**lemma** *subgraphI*:  
**assumes**  $\textit{verts } H \subseteq \textit{verts } G$   
**assumes**  $\textit{arcs } H \subseteq \textit{arcs } G$   
**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**  $\text{verts } H \subseteq \text{verts } G$   $\text{arcs } H \subseteq \text{arcs } G$  *compatible*  $G H$  *wf-digraph*  $H$

**using** *assms* **by** (*simp* *add*: *subgraph-def*)

**lemma** *induced-subgraphI*[*intro*]:

**assumes** *subgraph*  $H G$

**assumes**  $\text{arcs } H = \{e \in \text{arcs } G. \text{tail } G e \in \text{verts } H \wedge \text{head } G e \in \text{verts } H\}$

**shows** *induced-subgraph*  $H G$

**using** *assms* **unfolding** *induced-subgraph-def* **by** *safe*

**lemma** *induced-subgraphE*[*elim*]:

**assumes** *induced-subgraph*  $H G$

**assumes**  $\llbracket \text{subgraph } H G; \text{arcs } H = \{e \in \text{arcs } G. \text{tail } G e \in \text{verts } H \wedge \text{head } G e \in \text{verts } H\} \rrbracket \implies P$

**shows**  $P$

**using** *assms* **by** (*auto* *simp* *add*: *induced-subgraph-def*)

**lemma** *pverts-mk-symmetric*[*simp*]:  $\text{pverts } (\text{mk-symmetric } G) = \text{verts } G$

**and** *parcs-mk-symmetric*:

$\text{parcs } (\text{mk-symmetric } G) = (\bigcup e \in \text{arcs } G. \{( \text{tail } G e, \text{head } G e), (\text{head } G e, \text{tail } G e)\})$

**by** (*auto* *simp*: *mk-symmetric-def* *arcs-ends-conv* *image-UN*)

**lemma** *arcs-ends-mono*:

**assumes** *subgraph*  $H G$

**shows** *arcs-ends*  $H \subseteq \text{arcs-ends } G$

**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*  $((\lambda(a,b). (b,a)) \text{ ‘arcs-ends } G)$  (**is** *finite*  $?X$ ) **by** (*auto* *simp*: *arcs-ends-conv*)

**also have**  $?X = \{(a, b). (b, a) \in \text{arcs-ends } G\}$  **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* (( $\lambda(a,b). (b,a)$ ) ‘*arcs-ends*  $G$ ) (**is finite** ? $X$ ) **by** (*auto simp: arcs-ends-conv*)  
  **also have** ? $X = \{(a, b). (b, a) \in \text{arcs-ends } G\}$  **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**  $u \rightarrow^* \text{mk-symmetric } G v$   
**proof** –  
  **have** *arcs-ends*  $G \subseteq \text{parcs}$  (*mk-symmetric*  $G$ )  
    ( $u, v \in \text{rtrancl-on}$  (*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*  $G \implies \text{parcs}$  (*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**  $u \rightarrow^* \text{mk-symmetric } G v \iff u \rightarrow^* v$  (**is** ? $L \iff$   
  ? $R$ )  
  **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**  $u \rightarrow^* \text{mk-symmetric } G v$   
    **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  $\subseteq$  arcs G* *verts H  $\subseteq$  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  $\in$  arcs H*  
**with** *assms* **show** *tail H e  $\in$  verts H* *head H e  $\in$  verts H*  
**by** (*auto simp: subgraph-def intro: wf-digraph.wellformed*)  
**from** *e* **and** *assms* **have** *e  $\in$  arcs H  $\cap$  arcs G* **by** *auto*  
**with** *assms* **show** *tail H e  $\neq$  head H e*  
**using** *no-loops* **by** (*auto simp: subgraph-def compatible-def arc-to-ends-def*)

**next**

**have** *arcs H  $\subseteq$  arcs G* *verts H  $\subseteq$  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  $\in$  arcs H* *e2  $\in$  arcs H*  
**and** *eq: arc-to-ends H e1 = arc-to-ends H e2*  
**with** *assms* **have** *e1  $\in$  arcs H  $\cap$  arcs G* *e2  $\in$  arcs H  $\cap$  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  $\rightarrow_H$  v* *subgraph H G*  
**shows** *u  $\rightarrow$  v*  
**using** *assms* **by** (*blast dest: arcs-ends-mono*)

**lemma** (*in pre-digraph*) *reachable-mono*:  
**assumes** *walk: u  $\rightarrow^*_H$  v* **and** *sub: subgraph H G*  
**shows** *u  $\rightarrow^*$  v*  
**proof** –

**have**  $\text{verts } H \subseteq \text{verts } G$  **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-apatk-imp-apatk*:  
**assumes** *path: pre-digraph.apatk H u p v*  
**assumes** *sub: subgraph H G*  
**shows** *apatk u p v*  
**using** *assms* **unfolding** *pre-digraph.apatk-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 =  $\lambda G. \text{mk-symmetric } G$*   
**from** *assms* **have** *compatible G H* **by** *auto*  
**with** *assms*  
**show**  $\text{verts } (?wpms \ H) \subseteq \text{verts } (?wpms \ G)$   
**and**  $\text{arcs } (?wpms \ H) \subseteq \text{arcs } (?wpms \ G)$   
**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**  $\text{in-degree } H \ v \leq \text{in-degree } G \ v$   
**proof** –  
**have** *finite (in-arcs G v)* **by** *auto*  
**moreover**  
**have**  $\text{in-arcs } H \ v \subseteq \text{in-arcs } G \ v$   
**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  $\subseteq$  arcs G* **by** *blast*  
**show** *?thesis*  
**proof**  
**from** *assms* **have** *verts H  $\subseteq$  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  $\in$  arcs H*  
**with** *arcs (compatible G H)* **show** *tail H e  $\neq$  head H e*  
**by** (*auto dest: no-loops simp: compatible-tail[symmetric] compatible-head[symmetric]*)  
**next**  
**fix** *e1 e2* **assume** *e1  $\in$  arcs H e2  $\in$  arcs H* **and** *ate: arc-to-ends H e1 =*

*arc-to-ends*  $H\ e2$   
**with** *arcs*  $\langle \text{compatible } G\ H \rangle$  **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*  $\Rightarrow$   $'a$  *set*  $\Rightarrow$   $( 'a, 'b )$  *pre-digraph*  
**(infix**  $\uparrow$  67) **where**  
 $G \uparrow vs = (\ \backslash \text{verts} = vs, \text{arcs} = \{e \in \text{arcs } G. \text{tail } G\ e \in vs \wedge \text{head } G\ e \in vs\},$   
 $\text{tail} = \text{tail } G, \text{head} = \text{head } G \ \backslash)$

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

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

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

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

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

**lemma** (**in** *wf-digraph*) *induced-induce*[*intro*]:  
**assumes**  $vs \subseteq \text{verts } G$   
**shows** *induced-subgraph*  $(G \uparrow vs)\ G$   
**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*]:  
 $\text{wf-digraph } (G \uparrow vs)$   
**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**  $e1 \in \text{arcs } H$   
**then obtain**  $e2$  **where**  $\text{tail } G \ e1 = \text{head } G \ e2 \ \text{head } G \ e1 = \text{tail } G \ e2 \ e2 \in \text{arcs } G$   
**using**  $\text{assms}$  **by** ( $\text{auto simp add: symmetric-conv}$ )  
**moreover**  
**then have**  $e2 \in \text{arcs } H$   
**using**  $\text{assms}$  **and**  $\langle e1 \in \text{arcs } H \rangle$  **by**  $\text{auto}$   
**ultimately**  
**show**  $\exists e2 \in \text{arcs } H. \text{tail } H \ e1 = \text{head } H \ e2 \wedge \text{head } H \ e1 = \text{tail } H \ e2$   
**using**  $\text{assms}$   $\langle e1 \in \text{arcs } H \rangle$   $\langle \text{compatible } G \ H \rangle$   
**by** ( $\text{auto simp: compatible-head compatible-tail}$ )  
**qed**

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

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

```

lemma induce-subgraph-ends[simp]:
  arc-to-ends (G ↯ S) = arc-to-ends G
  by (auto simp: arc-to-ends-def)

lemma dominates-induce-subgraphD:
  assumes u →G ↯ S v shows u →G v
  using assms by (auto simp: arcs-ends-def intro: rev-image-eqI)

context wf-digraph begin

  lemma reachable-induce-subgraphD:
    assumes u →*G ↯ S v S ⊆ verts G shows u →*G v
  proof –
    interpret GS: wf-digraph G ↯ 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 →G ↯ S v S ⊆ T shows u →G ↯ T v
    using assms by (auto simp: arcs-ends-def)

  lemma reachable-induce-ss:
    assumes u →*G ↯ S v S ⊆ T shows u →*G ↯ 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 ↯ S) = awalk-verts
  proof (intro ext)
    fix u p show pre-digraph.awalk-verts (G ↯ 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) ⊆ S
    shows pre-digraph.cas (G ↯ 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 ( $G \upharpoonright \text{set } (\text{awalk-verts } (\text{head } G \ a) \ as))$ ) ( $\text{head } G \ a$ ) as v
  using Cons by auto
then have pre-digraph.cas ( $G \upharpoonright S$ ) ( $\text{head } G \ a$ ) as v
  using  $\langle - \subseteq S \rangle$  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)  $\subseteq S$ 
  shows pre-digraph.awalk (G  $\upharpoonright S$ ) u p v
proof –
  interpret GS: wf-digraph G  $\upharpoonright S$  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  $V \subseteq \text{verts } G$  shows subgraph (G  $\upharpoonright V$ ) G
  by (metis assms induced-imp-subgraph induced-induce)

end

lemma induced-subgraphI':
  assumes subg:subgraph H G
  assumes max:  $\bigwedge H'. \text{subgraph } H' G \implies (\text{verts } H' \neq \text{verts } H \vee \text{arcs } H' \subseteq \text{arcs } H)$ 
  shows induced-subgraph H G
proof –
  interpret H: wf-digraph H using  $\langle \text{subgraph } H \ G \rangle$  ..
  define H' where  $H' = G \upharpoonright \text{verts } H$ 
  then have H'-props: subgraph H' G  $\text{verts } H' = \text{verts } H$ 
    using subg by (auto intro: wf-digraph.subgraph-induce-subgraphI)
  moreover
  have  $\text{arcs } H' = \text{arcs } H$ 
  proof
    show  $\text{arcs } H' \subseteq \text{arcs } H$  using max H'-props by auto
    show  $\text{arcs } H \subseteq \text{arcs } H'$  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  $\longleftrightarrow$  subgraph H G  $\wedge$  ( $\forall H'. \text{subgraph } H' G \implies (\text{verts } H' \neq \text{verts } H \vee \text{arcs } H' \subseteq \text{arcs } H)$ ) (is ?L  $\longleftrightarrow$  ?R)
proof –
  { fix H' :: ('a,'b) pre-digraph
    assume A: verts H' = verts H subgraph H' G

```

```

interpret  $H'$ : wf-digraph  $H'$  using  $\langle \text{subgraph } H' \ G \rangle$  ..
from  $\langle \text{subgraph } H' \ G \rangle$ 
have comp:  $\text{tail } G = \text{tail } H'$   $\text{head } G = \text{head } H'$  by (auto simp: compatible-def)
then have  $\bigwedge a. a \in \text{arcs } H' \implies \text{tail } G \ a \in \text{verts } H \ \bigwedge a. a \in \text{arcs } H' \implies \text{tail}$ 
 $G \ a \in \text{verts } H$ 
by (auto dest: H'.wellformed simp: A)
then have  $\text{arcs } H' \subseteq \{e \in \text{arcs } G. \text{tail } G \ e \in \text{verts } H \ \wedge \ \text{head } G \ e \in \text{verts } H\}$ 
using  $\langle \text{subgraph } H' \ G \rangle$  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]:  $\text{verts } (\text{union } G \ H) = \text{verts } G \cup \text{verts } H$  and
arcs-union[simp]:  $\text{arcs } (\text{union } G \ H) = \text{arcs } G \cup \text{arcs } H$  and
tail-union[simp]:  $\text{tail } (\text{union } G \ H) = \text{tail } G$  and
head-union[simp]:  $\text{head } (\text{union } G \ H) = \text{head } G$ 
by (auto simp: union-def)

```

**lemma** *wellformed-union*:

```

assumes wf-digraph  $G$  wf-digraph  $H$  compatible  $G \ H$ 
shows wf-digraph  $(\text{union } G \ H)$ 
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  $(\text{union } H1 \ H2) \ G \longleftrightarrow \text{subgraph } H1 \ G \ \wedge \ \text{subgraph } H2 \ G$ 
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  $(\text{union } H1 \ H2) \ G$ 

```

**proof** –

```

from assms have wf-digraph  $(\text{union } H1 \ H2)$ 
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  $(\text{union } G \ H)$ 

```

**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*  $Q$   $x \wedge x. P x \implies Q x$  **shows** *max-subgraph*  $P$   $x$   
**using** *assms* **by** (*auto simp: max-subgraph-def*)

**lemma** (*in pre-digraph*) *max-subgraph-prop*: *max-subgraph*  $P$   $x \implies P$   $x$   
**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*  $H$  ( $G \upharpoonright \text{verts } H$ )  
**using** *assms* **by** (*auto simp: subgraph-def compatible-def wf-digraph.wellformed wf-digraph.wellformed-induce-subgraph*)

**definition** *arc-mono* :: ( $'a, 'b$ ) *pre-digraph*  $\Rightarrow$  *bool*  $\Rightarrow$  *bool* **where**  
*arc-mono*  $P \equiv (\forall H1 H2. P H1 \wedge \text{subgraph } H1 H2 \wedge \text{verts } H1 = \text{verts } H2 \longrightarrow P H2)$

**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*  $H$  ( $G \upharpoonright \text{verts } H$ ) *subgraph* ( $G \upharpoonright \text{verts } H$ )  $G$  *verts*  $H = \text{verts}$  ( $G \upharpoonright \text{verts } H$ )  $P$   $H$   
**using** *assms* **by** (*auto simp: max-subgraph-def subgraph-induce-subgraphI2 subgraph-induce-subgraphI*)

**moreover**

**then have**  $P$  ( $G \upharpoonright \text{verts } H$ )  
**using** *assms* **by** (*auto simp: arc-mono-def*)

**ultimately**  
**have**  $\text{max-subgraph } P (G \upharpoonright \text{verts } H)$   
**using**  $\text{assms by (auto simp: max-subgraph-def)metis}$   
**then have**  $H = G \upharpoonright \text{verts } H$   
**using**  $\langle \text{max-subgraph } P H \rangle \langle \text{subgraph } H \text{ -} \rangle$   
**by**  $(\text{intro max-subgraph-subg-eq})$   
**show**  $?thesis$  **using**  $\text{assms by (subst } \langle H = \text{-} \rangle \text{) (auto simp: max-subgraph-def)}$   
**qed**

**lemma**  $(\text{in pre-digraph})$   $\text{induced-subgraph-altdef2}$ :  
 $\text{induced-subgraph } H G \longleftrightarrow \text{max-subgraph } (\lambda H'. \text{verts } H' = \text{verts } H) H$   $(\text{is } ?L \longleftrightarrow ?R)$

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

**lemma**  $(\text{in pre-digraph})$   $\text{max-subgraphI}$ :  
**assumes**  $P x \text{ subgraph } x G \wedge y. \llbracket x \neq y; \text{subgraph } x y; \text{subgraph } y G \rrbracket \implies \neg P y$   
**shows**  $\text{max-subgraph } P x$   
**using**  $\text{assms by (auto simp: max-subgraph-def)}$

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

## 8.8 Connected and Strongly Connected Graphs

**context**  $\text{wf-digraph}$  **begin**

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

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

```

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  $\text{verts } G \neq \{\}$  unfolding strongly-connected-def spanning-def
by auto
next
  fix  $u\ v$  assume  $u \in \text{verts } G$  and  $v \in \text{verts } G$ 
  then have  $u \rightarrow^*_H v$  subgraph  $H\ G$ 
    using assms by (auto simp add: strongly-connected-def)
  then show  $u \rightarrow^* v$  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 ( $G \upharpoonright (\text{verts } H)$ )
proof –
  let  $?is-H = G \upharpoonright (\text{verts } H)$ 

```

```

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

```

```

  have  $\text{verts } H \subseteq \text{verts } G$  using assms by auto

```

```

  have subgraph  $H\ (G \upharpoonright \text{verts } H)$ 
    using subg by (intro subgraphI) (auto simp: compatible-def)
  then show thesis
    using induced-induce[OF  $\langle \text{verts } H \subseteq \text{verts } G \rangle$ ]
    and sc  $GrH$ .strongly-connected-spanning-imp-strongly-connected
    unfolding spanning-def by auto
qed

```

```

lemma in-sccs-vertsI-sccs:
  assumes  $S \in \text{verts } 'sccs$  shows  $S \in \text{sccs-verts}$ 
  unfolding sccs-verts-def
proof (intro CollectI conjI allI ballI impI)
show  $S \neq \{\}$  using assms by (auto simp: sccs-verts-def sccs-def strongly-connected-def)

```

```

  from assms have sc: strongly-connected ( $G \upharpoonright S$ )  $S \subseteq \text{verts } G$ 
  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  $A: u \in S\ v \in S$ 

```

```

with sc have  $u \rightarrow^* G \upharpoonright S v$  by auto
then show  $u \rightarrow^* G v$  using  $\langle S \subseteq \text{verts } G \rangle$  by (rule reachable-induce-subgraphD)
next
fix u v assume A:  $u \in S v \notin S$ 
{ assume B:  $u \rightarrow^* G v v \rightarrow^* G u$ 
  from B obtain p-uv where p-uv: awalk u p-uv v by (metis reachable-awalk)
  from B obtain p-vu where p-vu: awalk v p-vu u by (metis reachable-awalk)
  define T where  $T = S \cup \text{set } (\text{awalk-verts } u \text{ p-uv}) \cup \text{set } (\text{awalk-verts } v \text{ p-vu})$ 
  have  $S \subseteq T$  by (auto simp: T-def)
  have  $v \in T$  using p-vu by (auto simp: T-def set-awalk-verts)
  then have  $T \neq S$  using  $\langle v \notin S \rangle$  by auto

interpret T: wf-digraph G  $\upharpoonright$  T by auto

from p-uv have T-p-uv: T.awalk u p-uv v
  by (rule awalk-induce) (auto simp: T-def)
from p-vu have T-p-vu: T.awalk v p-vu u
  by (rule awalk-induce) (auto simp: T-def)

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

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

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

{ fix x assume  $x \in T$ 
  moreover
  { assume  $x \in S$  then have  $v \rightarrow^* G \upharpoonright T x$ 
    using uv-reach A1 A by (auto intro: T.reachable-trans)
  }

```

```

    } moreover
    { assume  $x \in \text{set } (\text{awalk-verts } v \text{ } p\text{-}vu)$  then have  $v \rightarrow^* G \upharpoonright T x$ 
    using  $T\text{-}p\text{-}vu$  by (auto simp:  $\text{awalk-verts-}induce$  intro:  $T.\text{awalk-verts-reachable-from}$ )
    } moreover
    { assume  $x \in \text{set } (\text{awalk-verts } u \text{ } p\text{-}uw)$  then have  $v \rightarrow^* G \upharpoonright T x$ 
      using  $T\text{-}p\text{-}uw$  by (rule-tac  $T.\text{reachable-trans}$ [rotated])
      (auto intro:  $T.\text{awalk-verts-reachable-from}$   $uw\text{-}reach$  simp:  $\text{awalk-verts-}induce$ )
    } ultimately
    have  $v \rightarrow^* G \upharpoonright T x$  by (auto simp:  $T\text{-}def$ )
  } note  $vx\text{-}reach = \text{this}$ 

  { fix  $x y$  assume  $x \in T y \in T$  then have  $x \rightarrow^* G \upharpoonright T y$ 
    using  $xv\text{-}reach vx\text{-}reach$  by (blast intro:  $T.\text{reachable-trans}$ )
  }
  then have  $\text{strongly-connected } (G \upharpoonright T)$ 
    using  $\langle S \neq \{\} \rangle \langle S \subseteq T \rangle$  by auto
  moreover have  $\text{induced-subgraph } (G \upharpoonright T) G$ 
    using  $\langle S \subseteq \text{verts } G \rangle$ 
    by (auto simp:  $T\text{-}def$  intro:  $\text{awalk-verts-reachable-from } p\text{-}uw \text{ } p\text{-}vu \text{ } \text{reachable-in-verts}(2)$ )
  ultimately
  have  $\exists T. \text{induced-subgraph } (G \upharpoonright T) G \wedge \text{strongly-connected } (G \upharpoonright T) \wedge \text{verts } (G \upharpoonright S) \subset \text{verts } (G \upharpoonright T)$ 
    using  $\langle S \subseteq T \rangle \langle T \neq S \rangle$  by auto
  then have  $G \upharpoonright S \notin \text{sccs}$  unfolding  $\text{sccs-def}$  by blast
  then have  $S \notin \text{verts } 'sccs$ 
    by (metis (erased, hide-lams)  $\langle S \subseteq T \rangle \langle T \neq S \rangle \langle \text{induced-subgraph } (G \upharpoonright T) G \rangle \langle \text{strongly-connected } (G \upharpoonright T) \rangle$ 
       $\text{dual-order.order-iff-strict image-iff in-sccsE induce-subgraph-verts}$ )
  then have  $\text{False}$  using  $\text{assms}$  by metis
  }
  then show  $\neg u \rightarrow^* G v \vee \neg v \rightarrow^* G u$  by metis
  }
qed

```

end

**lemma**  $\text{arc-mono-strongly-connected}$ [intro,simp]:  $\text{arc-mono strongly-connected}$   
 by (auto simp:  $\text{arc-mono-def}$ ) (metis  $\text{spanning-def subgraphE wf-digraph.strongly-connected-spanning-imp-strongly-connected}$ )

**lemma** (in  $\text{pre-digraph}$ )  $\text{sccs-altdef2}$ :

$\text{sccs} = \{H. \text{max-subgraph strongly-connected } H\}$  (is ?L = ?R)

**proof** –

```

{ fix  $H H' :: ('a, 'b) \text{pre-digraph}$ 
  assume  $a1: \text{strongly-connected } H'$ 
  assume  $a2: \text{induced-subgraph } H' G$ 
  assume  $a3: \text{max-subgraph strongly-connected } H$ 
  assume  $a4: \text{verts } H \subseteq \text{verts } H'$ 
  have  $sg: \text{subgraph } H G$  and  $\text{ends-}G: \text{tail } G = \text{tail } H \text{ head } G = \text{head } H$ 

```

```

    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: ∀ H'. strongly-connected H' ⟶ induced-subgraph H' G ⟶ ¬ verts
H ⊂ verts H'
  interpret G: wf-digraph G using a1 by auto
  { fix y assume H ≠ y and subg: subgraph H y subgraph y G
    then have verts H ⊂ verts y
      using a1 by (auto simp: induced-subgraph-altdef2 max-subgraph-def)
    then have ¬strongly-connected y
      using subg a1 a2 a3 [THEN spec, of G ↯ 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 ∈ sccs-verts
begin

lemma reach-in: ∧ u v. [u ∈ S; v ∈ S] ⟹ u →*G v
  and not-reach-out: ∧ u v. [u ∈ S; v ∉ S] ⟹ ¬u →*G v ∨ ¬v →*G u
  and not-empty: S ≠ {}
  using S-in-sv by (auto simp: sccs-verts-def)

lemma reachable-induced:
  assumes conn: u ∈ S v ∈ S u →*G v
  shows u →*G ↯ S v
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)

```



```

from conn obtain p where p: awalk u p v by (metis reachable-awalk)
show ?thesis
proof (cases set p  $\subseteq$  arcs (G  $\upharpoonright$  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  $\in$  set p a  $\notin$  arcs (G  $\upharpoonright$  S) by auto
    moreover
    then have tail G a  $\notin$  S  $\vee$  head G a  $\notin$  S using p by auto
    ultimately
    obtain w where w  $\in$  set (awalk-verts u p) w  $\notin$  S using p by (auto simp:
set-awalk-verts)
    then have u  $\rightarrow^*_G$  w w  $\rightarrow^*_G$  v
    using p by (auto intro: awalk-verts-reachable-from awalk-verts-reachable-to)
    moreover have v  $\rightarrow^*_G$  u using conn reach-in by auto
    ultimately have u  $\rightarrow^*_G$  w w  $\rightarrow^*_G$  u by (auto intro: reachable-trans)
    with  $\langle w \notin S \rangle$  conn not-reach-out have False by blast
    then show ?thesis ..
  qed
qed

```

```

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

```

```

lemma induced-in-sccs: G  $\upharpoonright$  S  $\in$  sccs

```

```

proof –

```

```

  let ?H = G  $\upharpoonright$  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)

```

```

  { fix T assume S  $\subset$  T T  $\subseteq$  verts G strongly-connected (G  $\upharpoonright$  T)

```

```

    from  $\langle S \subset T \rangle$  obtain v where v  $\in$  T v  $\notin$  S by auto

```

```

    from not-empty obtain u where u  $\in$  S by auto

```

```

    then have u  $\in$  T using  $\langle S \subset T \rangle$  by auto

```

```

    from  $\langle u \in S \rangle$   $\langle v \notin S \rangle$  have  $\neg u \rightarrow^*_G v \vee \neg v \rightarrow^*_G u$  by (rule not-reach-out)

```

```

    moreover

```

```

    from (strongly-connected  $\rightarrow$ ) have u  $\rightarrow^*_G$   $\upharpoonright$  T v v  $\rightarrow^*_G$   $\upharpoonright$  T u

```

```

      using  $\langle v \in T \rangle$   $\langle u \in T \rangle$  by (auto simp: strongly-connected-def)

```

```

    then have u  $\rightarrow^*_G$  v v  $\rightarrow^*_G$  u

```

```

      using  $\langle T \subseteq \text{verts } G \rangle$  by (auto dest: reachable-induce-subgraphD)

```

```

    ultimately have False by blast

```

```

} note psuper-not-sc = this

have  $\neg (\exists c'. \text{induced-subgraph } c' G \wedge \text{strongly-connected } c' \wedge \text{verts } (G \upharpoonright S) \subset \text{verts } c')$ 
by (metis induce-subgraph-verts induced-imp-subgraph psuper-not-sc subgraphE strongly-connected-imp-induce-subgraph-strongly-connected)
with  $\langle S \subseteq \rightarrow \text{not-empty} \rangle$  show  $?H \in \text{sccs}$  by (intro in-sccsI induced-induce strongly-connected)
qed
end

```

**context** *wf-digraph* **begin**

```

lemma in-verts-sccsD-sccs:
  assumes  $S \in \text{sccs-verts}$ 
  shows  $G \upharpoonright S \in \text{sccs}$ 
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:  $\text{sccs-verts} = \text{verts } ' \text{sccs}$ 
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  $G \upharpoonright \text{verts } H = H$ 
  using assms by (auto simp: induced-subgraph-def induce-subgraph-def compatible-def)

```

```

lemma sccs-conv-sccs-verts:  $\text{sccs} = \text{induce-subgraph } G ' \text{sccs-verts}$ 
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  $\text{connected } G \longleftrightarrow \text{verts } G \neq \{\} \wedge (\forall u \in \text{verts } G. \forall v \in \text{verts } G. (u,v) \in \text{rtrancl-on } (\text{verts } G) ((\text{arcs-ends } G)^s))$ 
proof –
  have  $\text{symcl } (\text{arcs-ends } G) = \text{parcs } (\text{mk-symmetric } G)$ 
  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  $\text{verts } G \neq \{\}$  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  $u \in \text{verts } G\ v \in \text{verts } G$ 
  with sc-mks have  $u \rightarrow^* \text{mk-symmetric } G\ v$ 
    unfolding strongly-connected-def by auto
  then show  $u \rightarrow^* v$  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  $\text{verts } (\text{mk-symmetric } G) \neq \{\}$ 
    unfolding spanning-def connected-def strongly-connected-def by auto
next
  fix  $u\ v$ 
  assume  $u \in \text{verts } (\text{mk-symmetric } G)$  and  $v \in \text{verts } (\text{mk-symmetric } G)$ 
  then have  $u \in \text{pverts } (\text{mk-symmetric } H)$  and  $v \in \text{pverts } (\text{mk-symmetric } H)$ 
    using ⟨spanning H G⟩ by (auto simp: mk-symmetric-def)
  with ⟨connected H⟩
  have  $u \rightarrow^* \text{with-proj } (\text{mk-symmetric } H)\ v$  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  $u \rightarrow^* \text{mk-symmetric } G\ v$  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  $u \in \text{verts } G$ 
  shows  $(G \upharpoonright \{v. u \rightarrow^* v\}) \in \text{sccs}$ 
proof –
  let  $?c = (G \upharpoonright \{v. u \rightarrow^* v\})$ 
  have isub-c: induced-subgraph ?c G
    by (auto elim: reachable-in-vertsE)
  then interpret  $c: \text{wf-digraph } ?c$  by (rule wf-digraphI-induced)

```

```

have sym-c: symmetric ( $G \upharpoonright \{v. u \rightarrow^* v\}$ )
  using sym-arcs isub-c by (rule induced-graph-imp-symmetric)

note  $\langle \text{induced-subgraph } ?c \ G \rangle$ 
moreover
have strongly-connected ?c
proof (rule strongly-connectedI)
  show verts ?c  $\neq \{\}$  using assms by auto
next
  fix v w assume l-assms:  $v \in \text{verts } ?c \ w \in \text{verts } ?c$ 
  have  $u \rightarrow^* G \upharpoonright \{v. u \rightarrow^* v\} \ v$ 
    using l-assms by (intro induce-reachable-preserves-paths) auto
  then have  $v \rightarrow^* G \upharpoonright \{v. u \rightarrow^* v\} \ u$  by (rule symmetric-reachable[OF sym-c])
  also have  $u \rightarrow^* G \upharpoonright \{v. u \rightarrow^* v\} \ w$ 
    using l-assms by (intro induce-reachable-preserves-paths) auto
  finally show  $v \rightarrow^* G \upharpoonright \{v. u \rightarrow^* v\} \ w \cdot$ 
qed
moreover
have  $\neg(\exists d. \text{induced-subgraph } d \ G \wedge \text{strongly-connected } d \wedge$ 
   $\text{verts } ?c \subset \text{verts } d)$ 
proof
  assume  $\exists d. \text{induced-subgraph } d \ G \wedge \text{strongly-connected } d \wedge$ 
   $\text{verts } ?c \subset \text{verts } d$ 
  then obtain d where induced-subgraph d G strongly-connected d
   $\text{verts } ?c \subset \text{verts } d$  by auto
  then obtain v where  $v \in \text{verts } d$  and  $v \notin \text{verts } ?c$ 
  by auto

  have  $u \in \text{verts } ?c$  using  $\langle u \in \text{verts } G \rangle$  by auto
  then have  $u \in \text{verts } d$  using  $\langle \text{verts } ?c \subset \text{verts } d \rangle$  by auto
  then have  $u \rightarrow^*_d \ v$ 
    using  $\langle \text{strongly-connected } d \rangle \langle u \in \text{verts } d \rangle \langle v \in \text{verts } d \rangle$  by auto
  then have  $u \rightarrow^* \ v$ 
    using  $\langle \text{induced-subgraph } d \ G \rangle$ 
    by (auto intro: pre-digraph.reachable-mono)
  then have  $v \in \text{verts } ?c$  by (auto simp: reachable-awalk)
  then show False using  $\langle v \notin \text{verts } ?c \rangle$  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  $\text{verts } G = \text{verts } G'$ 
  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  $c \in sccs$ 
  assumes  $d \in sccs$ 
  assumes  $verts\ c \subseteq verts\ d$ 
  shows  $c = d$ 
using assms by (blast intro: induced-eq-verts-imp-eq)

context wf-digraph begin

  lemma connectedI:
    assumes  $verts\ G \neq \{\}$   $\wedge u\ v.\ u \in verts\ G \implies v \in verts\ G \implies u \rightarrow^*_{mk\text{-}symmetric\ G} v$ 
    shows connected  $G$ 
    using assms by (auto simp: connected-def)

  lemma connected-awalkE:
    assumes connected  $G$   $u \in verts\ G$   $v \in verts\ G$ 
    obtains  $p$  where pre-digraph.awalk (mk-symmetric  $G$ )  $u\ p\ v$ 
    proof –
      interpret  $sG$ : pair-wf-digraph mk-symmetric  $G$  ..
      from assms have  $u \rightarrow^*_{mk\text{-}symmetric\ G} v$  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  $G \cap verts\ H \neq \{\}$ 
  shows strongly-connected (union  $G\ H$ )
proof
  from sc show  $verts\ (union\ G\ H) \neq \{\}$ 
  unfolding strongly-connected-def by simp
next
  let  $?x = union\ G\ H$ 
  fix  $u\ v\ w$  assume  $u \in verts\ ?x$  and  $v \in verts\ ?x$ 
  obtain  $w$  where w-in-both: w  $\in verts\ G$   $w \in verts\ H$ 
  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:  $c \in \text{sccs}$   $d \in \text{sccs}$

assumes  $c \neq d$

shows  $\text{verts } c \cap \text{verts } d = \{\}$

proof (rule ccontr)

assume contr:  $\neg ?thesis$

let  $?x = \text{union } c \ d$

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:  $\text{verts } c \subseteq \text{verts } G$   $\text{verts } d \subseteq \text{verts } G$  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  $\upharpoonright$  verts ?x)

induced-subgraph (G  $\upharpoonright$  verts ?x) G

by – (intro strongly-connected-imp-induce-subgraph-strongly-connected,  
auto simp: subgraph-union-iff)

from assms have  $\neg \text{verts } c \subseteq \text{verts } d$  and  $\neg \text{verts } d \subseteq \text{verts } c$

by (metis in-sccs-subset-imp-eq)+

then have psub:  $\text{verts } c \subset \text{verts } ?x$

by (auto simp: union-def)

**then show** *False* **using** *induce-subgraph-conn*  
**by** (*metis*  $\langle c \in \text{sccs} \rangle$  *in-sccsE induce-subgraph-verts*)  
**qed**

**lemma** *in-sccs-verts-conv*:  
 $S \in \text{sccs-verts} \longleftrightarrow G \upharpoonright S \in \text{sccs}$   
**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*:  $u \in \text{verts } G \implies u \in \text{scc-of } u$   
**by** (*auto simp: scc-of-def*)

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

**lemma** (*in wf-digraph*) *scc-of-in-sccs-verts*:  
**assumes**  $u \in \text{verts } G$  **shows**  $u \in \text{sccs-verts}$   
**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*:  $S \in \text{sccs-verts} \implies S \subseteq \text{verts } G$   
**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**  $\text{sccs-verts} \subseteq \text{Pow } (\text{verts } G)$  **by** *auto*  
**ultimately show** *?thesis* **by** (*rule rev-finite-subset*)  
**qed**

**lemma** (*in wf-digraph*) *sccs-verts-conv-scc-of*:  
 $\text{sccs-verts} = \text{scc-of } \cdot \text{verts } G$  (**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**  $u \in \text{verts } G$  **using**  $\langle S \in ?L \rangle$  **by** (*metis sccs-verts-subsets subsetCE*)  
**then have**  $\text{scc-of } u \in \text{sccs-verts}$   $u \in \text{scc-of } u$   
**by** (*auto intro: scc-of-in-sccs-verts in-scc-of-self*)  
**ultimately**  
**have**  $\text{scc-of } u = S$  **using** *sccs-verts-disjoint* **by** *blast*

then show  $S \in ?R$  using  $\langle scc\text{-of } u \in \cdot \rangle \langle u \in \text{verts } G \rangle$  by auto  
qed

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

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

lemma (in *wf-digraph*) *scc-of-eq*:  $u \in scc\text{-of } v \implies scc\text{-of } u = scc\text{-of } v$   
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  $\langle G \in sccs \rangle$  have  $\text{verts } G \cap \text{verts } H = \{\}$  by (*rule scc-disj*)

moreover

from  $\langle H \in sccs \rangle$  have  $\text{verts } H \subseteq \text{verts } G$  by auto

ultimately

have  $\text{verts } H = \{\}$  by auto

with  $\langle H \in sccs \rangle$  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  $\text{verts } G \neq \{\}$  shows  $\exists c. c \in sccs$

proof –

from *assms* obtain  $u$  where  $u \in \text{verts } G$  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**  $(\bigcup c \in sccs. arcs\ c) = arcs\ G$   
**proof**  
**show**  $(\bigcup c \in sccs. arcs\ c) \subseteq arcs\ G$   
**by** *safe* (*metis in-sccsE induced-imp-subgraph subgraphE subsetD*)  
**show**  $arcs\ G \subseteq (\bigcup c \in sccs. arcs\ c)$   
**proof** (*safe*)  
**fix**  $e$  **assume**  $e \in arcs\ G$   
**define**  $a\ b$  **where** [*simp*]:  $a = tail\ G\ e$  **and** [*simp*]:  $b = head\ G\ e$   
  
**have**  $e \in (\bigcup x \in sccs. arcs\ x)$   
**proof** *cases*  
**assume**  $\exists x \in sccs. \{a, b\} \subseteq verts\ x$   
**then obtain**  $c$  **where**  $c \in sccs$  **and**  $\{a, b\} \subseteq verts\ c$   
**by** *auto*  
**then have**  $e \in \{e \in arcs\ G. tail\ G\ e \in verts\ c$   
 $\wedge head\ G\ e \in verts\ c\}$  **using**  $\langle e \in arcs\ G \rangle$  **by** *auto*  
**then have**  $e \in arcs\ c$  **using**  $\langle c \in sccs \rangle$  **by** *blast*  
**then show** *?thesis* **using**  $\langle c \in sccs \rangle$  **by** *auto*  
**next**  
**assume**  $l\text{-assm}: \neg(\exists x \in sccs. \{a, b\} \subseteq verts\ x)$   
  
**have**  $a \rightarrow^* b$  **using**  $\langle e \in arcs\ G \rangle$   
**by** (*metis a-def b-def reachable-adjI in-arcs-imp-in-arcs-ends*)  
**then have**  $\{a, b\} \subseteq verts\ (G \upharpoonright \{v. a \rightarrow^* v\})$   $a \in verts\ G$   
**by** (*auto elim: reachable-in-vertsE*)  
**moreover**  
**have**  $(G \upharpoonright \{v. a \rightarrow^* v\}) \in sccs$   
**using**  $\langle a \in verts\ G \rangle$  **by** (*auto intro: induce-reachable-is-in-sccs*)  
**ultimately**  
**have** *False* **using**  $l\text{-assm}$  **by** *blast*  
**then show** *?thesis* **by** *simp*  
**qed**  
**then show**  $e \in (\bigcup c \in sccs. arcs\ c)$  **by** *auto*  
**qed**  
**qed**  
**ultimately show** *?thesis*  
**by** (*auto simp add: Union-def*)  
**qed**  
  
**lemma** (*in sym-digraph*) *scc-for-vert-ex*:  
**assumes**  $u \in verts\ G$   
**shows**  $\exists c. c \in sccs \wedge u \in verts\ c$   
**using** *assms* **by** (*auto intro: induce-reachable-is-in-sccs*)

```

lemma (in sym-digraph) scc-decomp-unique:
  assumes  $S \subseteq \text{sccs } \text{verts } (Union\ S) = \text{verts } G$  shows  $S = \text{sccs}$ 
proof (rule ccontr)
  assume  $S \neq \text{sccs}$ 
  with assms obtain  $c$  where  $c \in \text{sccs}$  and  $c \notin S$  by auto
  with assms have  $\bigwedge d. d \in S \implies \text{verts } c \cap \text{verts } d = \{\}$ 
    by (intro scc-disj) auto
  then have  $\text{verts } c \cap \text{verts } (Union\ S) = \{\}$ 
    by (auto simp: Union-def)
  with assms have  $\text{verts } c \cap \text{verts } G = \{\}$  by auto
  moreover from ( $c \in \text{sccs}$ ) obtain  $u$  where  $u \in \text{verts } c \cap \text{verts } G$ 
    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-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-digraph  $\Rightarrow$  bool where
  vwalk p G  $\equiv$  set p  $\subseteq$  verts G  $\wedge$  set (vwalk-arcs p)  $\subseteq$  arcs-ends G  $\wedge$  p  $\neq$  []

```

```

definition vpath :: 'a vwalk  $\Rightarrow$  ('a,'b) pre-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  $\in$  set xs)  
then *vwalk-to-vpath* (dropWhile ( $\lambda y. y \neq x$ ) 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  $\subseteq$  *verts G*  
**assumes** set (*vwalk-arcs p*)  $\subseteq$  *arcs-ends G*  
**assumes** p  $\neq$  []  
**shows** *vwalk p G*  
**using** *assms* **by** (*auto simp add: vwalk-def*)

**lemma** *vwalkE[elim]*:  
**assumes** *vwalk p G*  
**assumes** set p  $\subseteq$  *verts G*  $\implies$   
set (*vwalk-arcs p*)  $\subseteq$  *arcs-ends G*  $\wedge$  p  $\neq$  []  $\implies$  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*  $\implies$  *distinct p*  $\implies$  P  
**shows** P  
**using** *assms* **by** (*simp add: vpath-def*)

**lemma** *vwalk-consI*:  
**assumes** *vwalk p G*  
**assumes** a  $\in$  *verts G*  
**assumes** (a, *hd p*)  $\in$  *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  $\neq$  []

```

assumes  $(a, hd\ p) \in arcs\text{-}ends\ G \implies vwalk\ p\ G \implies P$ 
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  $\bigwedge u. u \in verts\ G \implies P\ [u]$ 
assumes  $\bigwedge u\ v\ es. (u,v) \in arcs\text{-}ends\ G \implies P\ (v\ \# \ es) \implies P\ (u\ \# \ v\ \# \ es)$ 
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  $(u,v) \in arcs\text{-}ends\ G$  and  $P\ (v\ \# \ es')$ 
  using Cons by (auto elim: vwalk-consE)
  then show ?thesis using  $\langle es = v\ \# \ es' \rangle$  Cons.prem by auto
qed auto
qed auto

lemma vwalk-arcs-Cons[simp]:
assumes  $p \neq []$ 
shows  $vwalk\text{-}arcs\ (u\ \# \ p) = (u, hd\ p) \# \ vwalk\text{-}arcs\ p$ 
using assms by (cases p) simp+

lemma vwalk-arcs-append:
assumes  $p \neq []$  and  $q \neq []$ 
shows  $vwalk\text{-}arcs\ (p\ @\ q) = vwalk\text{-}arcs\ p\ @\ (last\ p, hd\ q) \# \ vwalk\text{-}arcs\ q$ 
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  $vwalk\text{-}arcs\ ((a\ \# \ p')\ @\ (b\ \# \ q'))$ 
   $= vwalk\text{-}arcs\ (a\ \# \ p')\ @\ (last\ (a\ \# \ p'), b) \# \ vwalk\text{-}arcs\ (b\ \# \ q')$ 
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  $(vwalk\text{-}arcs\ p) \subseteq set\ (vwalk\text{-}arcs\ (p\ @\ q))$ 
proof (cases p)
case (Cons a p') note  $p\text{-}Cons = Cons$  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
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  $a \in \text{verts } G$  and  $\text{set } p \subseteq \text{verts } G$  by auto
then show  $\text{set } (a \# p) \subseteq \text{verts } G$  by auto

from  $\langle \text{vwalk } p \ G \rangle$  have  $p \neq []$  by auto
then show  $\text{set } (\text{vwalk-args } (a \# p)) \subseteq \text{arcs-ends } G$ 
  using  $\langle \text{vwalk } p \ G \rangle$  and  $\langle (a, \text{hd } p) \in \text{arcs-ends } G \rangle$ 
  by (auto simp add: set-vwalk-args-cons)
qed

lemma vwalkI-append-l:
  assumes  $p \neq []$ 
  assumes  $\text{vwalk } (p @ q) \ G$ 
  shows  $\text{vwalk } p \ G$ 
proof
  from assms show  $p \neq []$  and  $\text{set } p \subseteq \text{verts } G$ 
    by (auto elim!: vwalkE)
  have  $\text{set } (\text{vwalk-args } p) \subseteq \text{set } (\text{vwalk-args } (p @ q))$ 
    by (auto simp add: set-vwalk-args-append1)
  then show  $\text{set } (\text{vwalk-args } p) \subseteq \text{arcs-ends } G$ 
    using assms by blast
qed

lemma vwalkI-append-r:
  assumes  $q \neq []$ 
  assumes  $\text{vwalk } (p @ q) \ G$ 
  shows  $\text{vwalk } q \ G$ 
proof
  from  $\langle \text{vwalk } (p @ q) \ G \rangle$  have  $\text{set } (p @ q) \subseteq \text{verts } G$  by blast
  then show  $\text{set } q \subseteq \text{verts } G$  by simp

  from  $\langle \text{vwalk } (p @ q) \ G \rangle$  have  $\text{set } (\text{vwalk-args } (p @ q)) \subseteq \text{arcs-ends } G$ 
    by blast
  then show  $\text{set } (\text{vwalk-args } q) \subseteq \text{arcs-ends } G$ 
    by (metis set-vwalk-args-append2 subset-trans)

  from  $\langle q \neq [] \rangle$  show  $q \neq []$  by assumption
qed

lemma vwalk-to-vpath-hd:  $\text{hd } (\text{vwalk-to-vpath } xs) = \text{hd } xs$ 
proof (induct xs rule: vwalk-to-vpath.induct)
  case ( $2 \ x \ xs$ ) then show ?case
  proof (cases x \in set xs)
    case True
    then have  $\text{hd } (\text{dropWhile } (\lambda y. y \neq x) \ xs) = x$ 
      using hd-dropWhile [where  $P = \lambda y. y \neq x$ ] 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  $\bigwedge x xs. x \in \text{set } xs \implies P (\text{dropWhile } (\lambda y. y \neq x) xs)$ 
     $\implies P (x \# xs)$ 
  assumes  $\bigwedge x xs. x \notin \text{set } xs \implies P xs \implies P (x \# xs)$ 
  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  $x \in \text{set } (\text{vwalk-to-vpath } xs)$ 
  shows  $x \in \text{set } xs$ 
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  $x \notin \text{set } xs$ 
  shows vwalk-to-vpath  $(x \# xs) = x \# \text{vwalk-to-vpath } xs$ 
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:  $\bigwedge x xs. x \notin \text{set } xs \implies \forall x' \in \text{set } xs. x' \neq x$  by metis
  from  $\langle x \in \text{set } xs \rangle$  obtain  $ys\ zs$  where  $xs = ys @ x \# zs$  and  $x \notin \text{set } ys$ 
    by (metis in-set-conv-decomp-first)
  then have vwalk-dW: vwalk (dropWhile  $(\lambda y. y \neq x) xs$ )  $G$ 
    using in-set and  $\langle xs = ys @ x \# zs \rangle$ 
    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  $x \in \text{verts } G$  and  $x\text{-notin: } x \notin \text{set } (\text{vwalk-to-vpath } xs)$ 
  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.prem1 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

lemma vwalk-imp-ex-vpath:
  assumes vwalk p G
  assumes hd p = u
  assumes last p = v
  shows  $\exists q. \text{vpath } q \text{ } G \wedge \text{hd } q = u \wedge \text{last } q = v$ 
  by (metis assms vwalk-to-vpath-hd vwalk-to-vpath-last vwalk-to-vpath-vpath)

lemma vwalk-arcs-set-nil:
  assumes  $x \in \text{set } (\text{vwalk-arcs } p)$ 
  shows  $p \neq []$ 
  using assms by fastforce

lemma in-set-vwalk-arcs-append1:
  assumes  $x \in \text{set } (\text{vwalk-arcs } p) \vee x \in \text{set } (\text{vwalk-arcs } q)$ 
  shows  $x \in \text{set } (\text{vwalk-arcs } (p @ q))$ 
  using assms proof
  assume  $x \in \text{set } (\text{vwalk-arcs } p)$ 
  then show  $x \in \text{set } (\text{vwalk-arcs } (p @ q))$ 

```



```

    by (cases q = [])
      (auto simp add: vwalk-arcs-append vwalk-arcs-set-nil)
next
  assume x ∈ set (vwalk-arcs q)
  then show x ∈ 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 ≠ [] q ≠ []
  assumes disj: x ∈ set (vwalk-arcs p) ∨ x = (last p, hd q)
    ∨ x ∈ set (vwalk-arcs q)
  shows x ∈ set (vwalk-arcs (p @ q))
using disj proof (elim disjE)
  assume x = (last p, hd q)
  then show x ∈ 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 ∈ set (vwalk-arcs p)
  shows u ∈ set p × set p
using asms by (induct p rule: vwalk-arcs.induct) auto

```

```

lemma set-vwalk-arcs-rev:
  set (vwalk-arcs (rev p)) = {(v, u). (u, v) ∈ 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)}
      ∪ {a. case a of (v, u) ⇒ (u, v) ∈ set (vwalk-arcs xs)}
      by (simp add: set-vwalk-arcs-snoc last-rev Cons)
    also have ... = {a. case a of (v, u) ⇒ (u, v) ∈ 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 ∈ verts G
  shows vpath [u] G
using asms by (intro vpathI vwalkI, auto)

```

**lemma** *vwalk-verts-in-verts*:

**assumes** *vwalk*  $p$   $G$   
**assumes**  $u \in \text{set } p$   
**shows**  $u \in \text{verts } G$   
**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*:

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

**lemma** *vwalk-arcs-butlast-empty*:

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

**definition** *joinable* :: ' $a$  *vwalk*  $\Rightarrow$  ' $a$  *vwalk*  $\Rightarrow$  *bool* **where**

$joinable\ p\ q \equiv last\ p = hd\ q \wedge p \neq [] \wedge q \neq []$

**definition** *vwalk-join* :: ' $a$  *list*  $\Rightarrow$  ' $a$  *list*  $\Rightarrow$  ' $a$  *list*

(*infixr*  $\oplus$  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]*:  $p \neq [] \implies joinable\ (v \# p)\ q = joinable\ p\ q$

**and** *joinable-Snoc-r-iff[simp]*:  $q \neq [] \implies joinable\ p\ (q @ [v]) = joinable\ p\ q$

**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**  $vwalk\text{-length } (p \oplus q) = vwalk\text{-length } p + vwalk\text{-length } q$

**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**  $vwalk\text{-arcs } (p \oplus q) = vwalk\text{-arcs } p @ vwalk\text{-arcs } q$

**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**  $set (vwalk\text{-arcs } p) \subseteq E$

**assumes**  $p = q \oplus r$

**shows**  $set (vwalk\text{-arcs } q) \subseteq E$

**by** (*metis assms vwalk-join-def set-vwalk-arcs-append1 subset-trans*)

**lemma** *vwalk-join-arcs2*:

**assumes**  $set (vwalk\text{-arcs } p) \subseteq E$

**assumes** *joinable q r*

**assumes**  $p = q \oplus r$

**shows**  $set (vwalk\text{-arcs } r) \subseteq E$

**using** *assms by (simp add: vwalk-join-arcs)*

**definition** *concat-vpath* :: *'a list*  $\Rightarrow$  *'a list*  $\Rightarrow$  *'a list* **where**

$concat\text{-vpath } p \ q \equiv vwalk\text{-to-vpath } (p \oplus q)$

**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**  $vpath (concat\text{-vpath } p \ q) \ G \wedge hd (concat\text{-vpath } p \ q) = u$

$\wedge last (concat\text{-vpath } p \ q) = w$

**proof** (*intro conjI*)

**have** *joinable: joinable p q using assms by auto*

**show**  $vpath (concat\text{-vpath } p \ q) \ G$

**unfolding** *concat-vpath-def using assms and joinable*

**by** (*auto intro: vwalk-to-vpath-vpath vwalk-joinI-vwalk*)

**show**  $hd (concat\text{-vpath } p \ q) = u \ last (concat\text{-vpath } p \ q) = w$

**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*:  
 $u \rightarrow^* G v \iff (\exists p. \text{vwalk } p \ G \wedge \text{hd } p = u \wedge \text{last } p = v)$  (is ?*L*  $\iff$  ?*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  $\langle vwalk\ p\ G \rangle$  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  $u \rightarrow_G x$  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:
   $u \rightarrow^*_G v \iff (\exists p. vpath\ p\ G \wedge hd\ p = u \wedge last\ p = v)$  (is  $?L \iff ?R$ )
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  $(u,v) \in set\ (vwalk\ arcs\ p)$ 
  obtains  $u \in set\ p\ v \in set\ p$ 
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  $\langle vwalk\ (x\ \# \ xs)\ G \rangle$ 
    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**  $\dots \subseteq \text{arcs-ends } G$   
**using**  $\langle \text{subgraph } H \ G \rangle$  **by** (*rule arcs-ends-mono*)  
**finally show**  $\text{set } (\text{vwalk-arcs } p) \subseteq \text{arcs-ends } G$  .  
**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**  $(a,b) \in \text{arcs-ends } G$   
**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* ::  $'a \Rightarrow 'aa$   
*iso-arcs* ::  $'b \Rightarrow 'bb$   
*iso-head* ::  $'bb \Rightarrow 'aa$   
*iso-tail* ::  $'bb \Rightarrow 'aa$

**definition** (**in** *pre-digraph*) *digraph-isomorphism* ::  $( 'a, 'b, 'aa, 'bb )$  *digraph-isomorphism*  
 $\Rightarrow \text{bool}$  **where**

*digraph-isomorphism* *hom*  $\equiv$   
*wf-digraph*  $G \wedge$   
*inj-on* (*iso-verts* *hom*) (*verts*  $G$ )  $\wedge$   
*inj-on* (*iso-arcs* *hom*) (*arcs*  $G$ )  $\wedge$   
 $(\forall a \in \text{arcs } G.$   
*iso-verts* *hom* (*tail*  $G \ a$ ) = *iso-tail* *hom* (*iso-arcs* *hom*  $a$ )  $\wedge$   
*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*  $\Rightarrow ( 'aa, 'bb, 'a, 'b )$   
*digraph-isomorphism* **where**

*inv-iso* *hom*  $\equiv ()$   
*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 ↔ u = v
  using assms by (auto simp: digraph-isomorphism-def dest: inj-onD)

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

lemma
  assumes digraph-isomorphism hom e ∈ 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 ⇒ inj-on (iso-arcs hom) (arcs G)
  by (auto simp: digraph-isomorphism-def)

lemma digraph-isomorphism-inj-on-verts:
  digraph-isomorphism hom ⇒ 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 ∈ arcs (app-iso hom G)
  then obtain e' where e' ∈ arcs G iso-arcs hom e' = e
  by auto
  then have iso-verts hom (head G e') ∈ verts (app-iso hom G)
  iso-verts hom (tail G e') ∈ verts (app-iso hom G)
  by auto
  then show tail (app-iso hom G) e ∈ verts (app-iso hom G)
  head (app-iso hom G) e ∈ 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**  $arcs-ends (app-iso hom G) = (\lambda(u,v). (iso-verts hom u, iso-verts hom v))$

$' arcs-ends G$

**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**  $u \in verts G$

**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**  $u \in verts G$

**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**  $u \in verts G$

**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**  $u \in verts G$

**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**  $u \in \text{verts } (\text{app-iso } \text{hom } G)$   
**shows**  $\text{in-arcs } (\text{app-iso } \text{hom } G) u = \text{iso-arcs } \text{hom } ' \text{in-arcs } G (\text{iso-verts } (\text{inv-iso } \text{hom}) u)$   
**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**  $u \in \text{verts } (\text{app-iso } \text{hom } G)$   
**shows**  $\text{out-arcs } (\text{app-iso } \text{hom } G) u = \text{iso-arcs } \text{hom } ' \text{out-arcs } G (\text{iso-verts } (\text{inv-iso } \text{hom}) u)$   
**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**  $u \in \text{verts } (\text{app-iso } \text{hom } G)$   
**shows**  $\text{in-degree } (\text{app-iso } \text{hom } G) u = \text{in-degree } G (\text{iso-verts } (\text{inv-iso } \text{hom}) u)$   
**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**  $u \in \text{verts } (\text{app-iso } \text{hom } G)$   
**shows**  $\text{out-degree } (\text{app-iso } \text{hom } G) u = \text{out-degree } G (\text{iso-verts } (\text{inv-iso } \text{hom}) u)$   
**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** *digraph-isomorphism hom*  
**shows**  $(\text{iso-verts } \text{hom } u) \rightarrow^* \text{app-iso } \text{hom } G (\text{iso-verts } \text{hom } v)$   
**proof** –  
**interpret**  $H: \text{wf-digraph } \text{app-iso } \text{hom } G$  **using** *hom* ..  
**from**  $r$  **obtain**  $p$  **where**  $\text{awalk } u p v$  **by** (*auto simp: reachable-awalk*)  
**then have**  $H.\text{awalk } (\text{iso-verts } \text{hom } u) (\text{map } (\text{iso-arcs } \text{hom}) p) (\text{iso-verts } \text{hom } v)$   
**using** *hom* **by** (*rule awalk-app-isoI*)  
**then show** *?thesis* **by** (*auto simp: H.reachable-awalk*)  
**qed**

**lemma** *awalk-app-iso-eq*:  
**assumes** *digraph-isomorphism hom*

**assumes**  $u \in \text{iso-verts } \text{hom} \text{ ' } \text{verts } G$   $v \in \text{iso-verts } \text{hom} \text{ ' } \text{verts } G$   $\text{set } p \subseteq \text{iso-arcs } \text{hom} \text{ ' } \text{arcs } G$   
**shows**  $\text{pre-digraph.awalk } (\text{app-iso } \text{hom } G) u p v$   
 $\longleftrightarrow \text{awalk } (\text{iso-verts } (\text{inv-iso } \text{hom}) u) (\text{map } (\text{iso-arcs } (\text{inv-iso } \text{hom})) p) (\text{iso-verts } (\text{inv-iso } \text{hom}) v)$   
**proof** –  
**interpret**  $H: \text{wf-digraph } \text{app-iso } \text{hom } G$  **using**  $\text{hom} \dots$   
**from**  $\text{assms}$  **show**  $?thesis$   
**by** ( $\text{induct } p$   $\text{arbitrary: } u$ )  
 $(\text{auto simp: awalk-simps } H.\text{awalk-simps } \text{iso-verts-head } \text{iso-verts-tail})$   
**qed**

**lemma**  $\text{reachable-app-iso-eq}$ :

**assumes**  $\text{hom: digraph-isomorphism } \text{hom}$   
**assumes**  $u \in \text{iso-verts } \text{hom} \text{ ' } \text{verts } G$   $v \in \text{iso-verts } \text{hom} \text{ ' } \text{verts } G$   
**shows**  $u \xrightarrow{*} \text{app-iso } \text{hom } G v \longleftrightarrow \text{iso-verts } (\text{inv-iso } \text{hom}) u \xrightarrow{*} \text{iso-verts } (\text{inv-iso } \text{hom}) v$  (**is**  $?L \longleftrightarrow ?R$ )  
**proof** –  
**interpret**  $H: \text{wf-digraph } \text{app-iso } \text{hom } G$  **using**  $\text{hom} \dots$

**show**  $?thesis$

**proof**

**assume**  $?L$

**then obtain**  $p$  **where**  $H.\text{awalk } u p v$  **by** ( $\text{auto simp: } H.\text{reachable-awalk}$ )

**moreover**

**then have**  $\text{set } p \subseteq \text{iso-arcs } \text{hom} \text{ ' } \text{arcs } G$  **by** ( $\text{simp add: } H.\text{awalk-def}$ )

**ultimately**

**show**  $?R$  **using**  $\text{assms}$  **by** ( $\text{auto simp: awalk-app-iso-eq } \text{reachable-awalk}$ )

**next**

**assume**  $?R$

**then obtain**  $p0$  **where**  $\text{awalk } (\text{iso-verts } (\text{inv-iso } \text{hom}) u) p0 (\text{iso-verts } (\text{inv-iso } \text{hom}) v)$

**by** ( $\text{auto simp: } \text{reachable-awalk}$ )

**moreover**

**then have**  $\text{set } p0 \subseteq \text{arcs } G$  **by** ( $\text{simp add: awalk-def}$ )

**define**  $p$  **where**  $p = \text{map } (\text{iso-arcs } \text{hom}) p0$

**have**  $\text{set } p \subseteq \text{iso-arcs } \text{hom} \text{ ' } \text{arcs } G$   $p0 = \text{map } (\text{iso-arcs } (\text{inv-iso } \text{hom})) p$

**using**  $\langle \text{set } p0 \subseteq \rightarrow \text{hom} \text{ by } (\text{auto simp: } p\text{-def } \text{map-idI } \text{subsetD}) \rangle$

**ultimately**

**show**  $?L$  **using**  $\text{assms}$  **by** ( $\text{auto simp: awalk-app-iso-eq[symmetric]} H.\text{reachable-awalk}$ )

**qed**

**qed**

**lemma**  $\text{connectedI-app-iso}$ :

**assumes**  $c: \text{connected } G$  **and**  $\text{hom: digraph-isomorphism } \text{hom}$

**shows**  $\text{connected } (\text{app-iso } \text{hom } G)$

**proof** –

**have**  $*$ :  $\text{symcl } (\text{arcs-ends } (\text{app-iso } \text{hom } G)) = (\lambda(u,v). (\text{iso-verts } \text{hom } u, \text{iso-verts } \text{hom } v)) \text{ ' } \text{symcl } (\text{arcs-ends } G)$



```

    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  $\Rightarrow$  ('a,'b,'c,'d) digraph-isomorphism  $\Rightarrow$  ('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**  $\bigwedge u. u \in \text{verts } H \implies u \in \text{iso-verts } \text{hom} \text{ ' } \text{verts } G \bigwedge a. a \in \text{arcs } H \implies a \in$   
*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 com-*  
*patible-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  $\longleftrightarrow$  H1 = H2 (is ?L  $\longleftrightarrow$  ?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  $a \in \text{arcs}$  (app-iso hom  $G$ )

obtains  $a0$  where  $a = \text{iso-arcs hom } a0$   $a0 \in \text{arcs } G$

using *assms* by *auto*

lemma in-verts-app-iso-cases:

assumes  $v \in \text{verts}$  (app-iso hom  $G$ )

obtains  $v0$  where  $v = \text{iso-verts hom } v0$   $v0 \in \text{verts } G$

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  $\text{pre-digraph.max-subgraph}$  (app-iso hom  $G$ )  $P$   $H$

$\longleftrightarrow \text{max-subgraph}$  ( $P$  o app-iso hom) (app-iso (inv-iso hom)  $H$ )

proof -

have *hom-inv*:  $\text{pre-digraph.digraph-isomorphism}$  (app-iso hom  $G$ ) (inv-iso hom)

using *hom* by (rule  $\text{digraph-isomorphism-invI}$ )

interpret  $aG$ : wf-digraph app-iso hom  $G$  using *hom* ..

have \*: subgraph (app-iso (inv-iso hom)  $H$ )  $G$

using *hom*  $\text{pre-digraph.subgraph-app-isoI}$ [ $OF$  *hom-inv* *subg*  $aG$ .*subgraph-refl*]

by *simp*

define  $H0$  where  $H0 = \text{app-iso}$  (inv-iso hom)  $H$

then have  $H0$ :  $H = \text{app-iso hom } H0$  subgraph  $H0$   $G$

using *hom* *subg* (subgraph -  $G$ ) by (auto *simp*: )

show *thesis* (is  $?L \longleftrightarrow ?R$ )

proof

assume  $?L$  then show  $?R$  using *assms*  $H0$

by (auto *simp*:  $\text{max-subgraph-def}$   $aG$ . $\text{max-subgraph-def}$   $\text{pre-digraph.subgraph-app-isoI}$ '  
 $\text{subgraph-refl}$   $\text{pre-digraph.app-iso-eq-conv}$ )

next

assume  $?R$

then show  $?L$

using *assms* *hom-inv*  $\text{pre-digraph.subgraph-app-isoI}$ [ $OF$  *hom-inv*]

apply (auto *simp*:  $\text{max-subgraph-def}$   $aG$ . $\text{max-subgraph-def}$ )

apply (erule  $\text{allE}$ [of - app-iso (inv-iso hom)  $H'$  for  $H'$ ])

apply (auto *simp*:  $\text{pre-digraph.subgraph-app-isoI}$ '  $\text{pre-digraph.app-iso-eq-conv}$ )

done

qed

qed

lemma (in pre-digraph) max-subgraph-cong:

assumes  $H = H' \wedge H''$ . subgraph  $H' H'' \implies \text{subgraph } H'' G \implies P H'' = P'$   
 $H''$

shows  $\text{max-subgraph } P H = \text{max-subgraph } P' H'$

**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**  $S \subseteq \{H. \text{subgraph } H \ G\}$   
**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]*:  $\text{card } (\text{iso-verts } \text{hom} \text{ ' } \text{verts } G) = \text{card } (\text{verts } G)$   
**using** *hom* **by** (*intro card-image digraph-isomorphism-inj-on-verts*)

**lemma** *card-arcs-iso[simp]*:  $\text{card } (\text{iso-arcs } \text{hom} \text{ ' } \text{arcs } G) = \text{card } (\text{arcs } G)$   
**using** *hom* **by** (*intro card-image digraph-isomorphism-inj-on-arcs*)

**lemma** *strongly-connected-iso[simp]*:  $\text{strongly-connected } (\text{app-iso } \text{hom} \ G) \longleftrightarrow$   
*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**  $\text{strongly-connected } (\text{app-iso } \text{hom} \ H) \longleftrightarrow \text{strongly-connected } H$

**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]*:  $\text{pre-digraph.sccs } (\text{app-iso } \text{hom} \ G) = \text{app-iso } \text{hom} \text{ ' } \text{sccs } (\text{is } ?L = ?R)$

**proof** (*intro set-eqI iffI*)

**fix** *x* **assume**  $x \in ?L$

**then have** *subgraph x (app-iso hom G)*

**by** (*auto simp: pre-digraph.sccs-def*)

**then show**  $x \in ?R$

**using** ( $x \in ?L$ ) *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**  $x \in ?R$

**then obtain** *x0* **where**  $x0 \in \text{sccs } x = \text{app-iso } \text{hom} \ x0$  **by** *auto*

**then show**  $x \in ?L$

```

using hom by (auto simp: pre-digraph.sccs-altdef2 max-subgraph-iso sub-
graph-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** *Funpow*

**imports**

*HOL-Library.FuncSet*

*HOL-Library.Permutations*

**begin**

## 12 Auxiliary Lemmas about $(\sim)$

```

lemma funpow-simp-l: f ((f  $\sim$  n) x) = (f  $\sim$  Suc n) x
by (metis comp-apply funpow.simps(2))

```

```

lemma funpow-add-app: (f  $\sim$  n) ((f  $\sim$  m) x) = (f  $\sim$  (n + m)) x
by (metis comp-apply funpow-add)

```

**lemma** *funpow-mod-eq*:

**assumes**  $(f \sim n) x = x$   $0 < n$  **shows**  $(f \sim (m \bmod n)) x = (f \sim m) x$

**proof** (*induct m rule: less-induct*)

**case** (*less m*)

{ **assume**  $m < n$  **then have** *?case* **by** *simp* }

**moreover**

{ **assume**  $m = n$  **then have** *?case* **by** (*simp add: (- = x)*) }

**moreover**

{ **assume**  $n < m$

**then have**  $m - n < m$   $0 < m - n$  **using**  $0 < n$  **by** *arith+*

**have**  $(f \sim (m \bmod n)) x = (f \sim ((m - n) \bmod n)) x$

**using**  $0 < m - n$  **by** (*simp add: mod-geq*)

**also have**  $\dots = (f \sim (m - n)) x$

**using**  $m - n < m$  **by** (*rule less*)

**also have**  $\dots = (f \sim (m - n)) ((f \sim n) x)$

**by** (*simp add: assms*)

**also have**  $\dots = (f \sim m) x$

**using**  $0 < m - n$  **by** (*simp add: funpow-add-app*)

**finally have** *?case* . }

ultimately show ?case by (metis linorder-neqE-nat)  
qed

lemma id-funpow-id:  
assumes  $f x = x$  shows  $(f \hat{\sim} n) x = x$   
using *assms* by (induct n) auto

lemma inv-id-abs[simp]:  $inv (\lambda a. a) = id$  unfolding id-def[symmetric] by simp

lemma inj-funpow:  
fixes  $f :: 'a \Rightarrow 'a$   
assumes *inj f* shows *inj (f  $\hat{\sim}$  n)*  
proof (induct n)  
case 0 then show ?case by (auto simp: id-def[symmetric])  
next  
case (Suc n) with *assms* show ?case unfolding funpow.simps by (rule inj-compose)  
qed

lemma funpow-inj-finite:  
assumes *inj p finite*  $\{(p \hat{\sim} n) x \mid n. True\}$   
shows  $\exists n > 0. (p \hat{\sim} n) x = x$   
proof -  
have  $\neg finite \{0::nat..\}$  by simp  
moreover  
have  $\{(p \hat{\sim} n) x \mid n. True\} = (\lambda n. (p \hat{\sim} n) x) \text{ ` } \{0..\}$  by auto  
with *assms* have *finite ...* by simp  
ultimately have  $\exists n \in \{0..\}. \neg finite \{m \in \{0..\}. (p \hat{\sim} m) x = (p \hat{\sim} n) x\}$   
by (rule pigeonhole-infinite)  
then obtain *n* where  $\neg finite \{m. (p \hat{\sim} m) x = (p \hat{\sim} n) x\}$  by auto  
then have  $\neg finite (\{m. (p \hat{\sim} m) x = (p \hat{\sim} n) x\} - \{n\})$  by auto  
then have  $(\{m. (p \hat{\sim} m) x = (p \hat{\sim} n) x\} - \{n\}) \neq \{\}$   
by (metis finite.emptyI)  
then obtain *m* where  $m: (p \hat{\sim} m) x = (p \hat{\sim} n) x$   $m \neq n$  by auto

{ fix *m n* assume  $(p \hat{\sim} n) x = (p \hat{\sim} m) x$   $m < n$   
have  $(p \hat{\sim} (n - m)) x = inv (p \hat{\sim} m) ((p \hat{\sim} m) ((p \hat{\sim} (n - m)) x))$   
using <inj p> by (simp add: inv-f-f inj-funpow)  
also have  $((p \hat{\sim} m) ((p \hat{\sim} (n - m)) x)) = (p \hat{\sim} n) x$   
using <m < n> by (simp add: funpow-add-app)  
also have  $inv (p \hat{\sim} m) \dots = x$   
using <inj p> by (simp add: <(p  $\hat{\sim}$  n) x =  $\rightarrow$  inj-funpow)>  
finally have  $(p \hat{\sim} (n - m)) x = x$   $0 < n - m$   
using <m < n> by auto }  
note *general = this*

show ?thesis  
proof (cases *m n* rule: linorder-cases)  
case less  
then show ?thesis using *general m* by metis

```

next
  case equal
  then show ?thesis using m by metis
next
  case greater
  then show ?thesis using general m by metis
qed
qed

lemma permutes-in-funpow-image:
  assumes f permutes S x ∈ S
  shows (f  $\sim$  n) x ∈ S
  using assms by (induct n) (auto simp: permutes-in-image)

lemma permutation-self:
  assumes permutation p shows  $\exists n > 0. (p \sim n) x = x$ 
proof cases
  assume p x = x then show ?thesis by auto
next
  assume p x  $\neq$  x
  from assms have inj p by (intro permutation-bijective bij-is-inj)
  { fix n
    from  $\langle p x \neq x \rangle$  have (p  $\sim$  Suc n) x  $\neq$  (p  $\sim$  n) x
    proof (induct n arbitrary: x)
      case 0 then show ?case by simp
    next
      case (Suc n)
      have p (p x)  $\neq$  p x
      proof (rule notI)
        assume p (p x) = p x
        then show False using  $\langle p x \neq x \rangle$   $\langle inj p \rangle$  by (simp add: inj-eq)
      qed
      have (p  $\sim$  Suc (Suc n)) x = (p  $\sim$  Suc n) (p x)
        by (metis funpow-simp-l funpow-swap1)
      also have ...  $\neq$  (p  $\sim$  n) (p x)
        by (rule Suc) fact
      also have (p  $\sim$  n) (p x) = (p  $\sim$  Suc n) x
        by (metis funpow-simp-l funpow-swap1)
      finally show ?case by simp
    qed }
  then have {(p  $\sim$  n) x | n. True}  $\subseteq$  {x. p x  $\neq$  x}
    by auto
  then have finite {(p  $\sim$  n) x | n. True}
    using permutation-finite-support[OF assms] by (rule finite-subset)
  with  $\langle inj p \rangle$  show ?thesis by (rule funpow-inj-finite)
qed

```



**lemma** (in  $-$ ) *funpow-invs*:  
**assumes**  $m \leq n$  **and** *inv*:  $\bigwedge x. f (g x) = x$   
**shows**  $(f \overset{\sim}{\sim} m) ((g \overset{\sim}{\sim} n) x) = (g \overset{\sim}{\sim} (n - m)) x$   
**using**  $\langle m \leq n \rangle$   
**proof** (*induction m*)  
**case** (*Suc m*)  
**moreover then have**  $n - m = \text{Suc } (n - \text{Suc } m)$  **by** *auto*  
**ultimately show** *?case* **by** (*auto simp: inv*)  
**qed** *simp*

### 13 Function-power distance between values

**definition** *funpow-dist* ::  $('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{nat}$  **where**  
*funpow-dist*  $f x y \equiv \text{LEAST } n. (f \overset{\sim}{\sim} n) x = y$

**abbreviation** *funpow-dist1* ::  $('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a \Rightarrow \text{nat}$  **where**  
*funpow-dist1*  $f x y \equiv \text{Suc } (\text{funpow-dist } f (f x) y)$

**lemma** *funpow-dist-0*:  
**assumes**  $x = y$  **shows** *funpow-dist*  $f x y = 0$   
**using** *assms* **unfolding** *funpow-dist-def* **by** (*intro Least-eq-0*) *simp*

**lemma** *funpow-dist-least*:  
**assumes**  $n < \text{funpow-dist } f x y$  **shows**  $(f \overset{\sim}{\sim} n) x \neq y$   
**proof** (*rule notI*)  
**assume**  $(f \overset{\sim}{\sim} n) x = y$   
**then have** *funpow-dist*  $f x y \leq n$  **unfolding** *funpow-dist-def* **by** (*rule Least-le*)  
**with** *assms* **show** *False* **by** *linarith*  
**qed**

**lemma** *funpow-dist1-least*:  
**assumes**  $0 < n < \text{funpow-dist1 } f x y$  **shows**  $(f \overset{\sim}{\sim} n) x \neq y$   
**proof** (*rule notI*)  
**assume**  $(f \overset{\sim}{\sim} n) x = y$   
**then have**  $(f \overset{\sim}{\sim} (n - 1)) (f x) = y$   
**using**  $\langle 0 < n \rangle$  **by** (*cases n*) (*simp-all add: funpow-swap1*)  
**then have** *funpow-dist*  $f (f x) y \leq n - 1$  **unfolding** *funpow-dist-def* **by** (*rule Least-le*)  
**with** *assms* **show** *False* **by** *simp*  
**qed**

### 14 Cyclic Permutations

**inductive-set** *orbit* ::  $('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a$  **set for**  $f x$  **where**  
*base*:  $f x \in \text{orbit } f x$  |  
*step*:  $y \in \text{orbit } f x \implies f y \in \text{orbit } f x$

**definition** *cyclic-on* ::  $('a \Rightarrow 'a) \Rightarrow 'a$  **set**  $\Rightarrow \text{bool}$  **where**

$cyclic\text{-}on\ f\ S \longleftrightarrow (\exists s \in S. S = orbit\ f\ s)$

**lemma** *orbit-altdef*:  $orbit\ f\ x = \{(f \smallfrown n)\ x \mid n. 0 < n\}$  (**is**  $?L = ?R$ )

**proof** (*intro set-eqI iffI*)

**fix**  $y$  **assume**  $y \in ?L$  **then show**  $y \in ?R$

**by** (*induct rule: orbit.induct*) (*auto simp: exI[where x=1] exI[where x=Suc n for n]*)

**next**

**fix**  $y$  **assume**  $y \in ?R$

**then obtain**  $n$  **where**  $y = (f \smallfrown n)\ x$   $0 < n$  **by** *blast*

**then show**  $y \in ?L$

**proof** (*induction n arbitrary: y*)

**case** (*Suc n*) **then show**  $?case$  **by** (*cases n = 0*) (*auto intro: orbit.intros*)

**qed** *simp*

**qed**

**lemma** *orbit-trans*:

**assumes**  $s \in orbit\ f\ t$   $t \in orbit\ f\ u$  **shows**  $s \in orbit\ f\ u$

**using** *assms* **by** *induct* (*auto intro: orbit.intros*)

**lemma** *orbit-subset*:

**assumes**  $s \in orbit\ f\ (f\ t)$  **shows**  $s \in orbit\ f\ t$

**using** *assms* **by** (*induct*) (*auto intro: orbit.intros*)

**lemma** *orbit-sim-step*:

**assumes**  $s \in orbit\ f\ t$  **shows**  $f\ s \in orbit\ f\ (f\ t)$

**using** *assms* **by** *induct* (*auto intro: orbit.intros*)

**lemma** *orbit-step*:

**assumes**  $y \in orbit\ f\ x$   $f\ x \neq y$  **shows**  $y \in orbit\ f\ (f\ x)$

**using** *assms*

**proof** *induction*

**case** (*step y*) **then show**  $?case$  **by** (*cases x = y*) (*auto intro: orbit.intros*)

**qed** *simp*

**lemma** *self-in-orbit-trans*:

**assumes**  $s \in orbit\ f\ s$   $t \in orbit\ f\ s$  **shows**  $t \in orbit\ f\ t$

**using** *assms*(2,1) **by** *induct* (*auto intro: orbit-sim-step*)

**lemma** *orbit-swap*:

**assumes**  $s \in orbit\ f\ s$   $t \in orbit\ f\ s$  **shows**  $s \in orbit\ f\ t$

**using** *assms*(2,1)

**proof** *induction*

**case** *base* **then show**  $?case$  **by** (*cases f s = s*) (*auto intro: orbit-step*)

**next**

**case** (*step x*) **then show**  $?case$  **by** (*cases f x = s*) (*auto intro: orbit-step*)

**qed**

**lemma** *permutation-self-in-orbit*:

**assumes** *permutation f* **shows**  $s \in \text{orbit } f \ s$   
**unfolding** *orbit-altdef* **using** *permutation-self[OF assms, of s]* **by** *simp metis*

**lemma** *orbit-altdef-self-in*:

**assumes**  $s \in \text{orbit } f \ s$  **shows**  $\text{orbit } f \ s = \{(f \ \overset{\sim}{\sim} \ n) \ s \mid n. \ \text{True}\}$

**proof** (*intro set-eqI iffI*)

**fix**  $x$  **assume**  $x \in \{(f \ \overset{\sim}{\sim} \ n) \ s \mid n. \ \text{True}\}$

**then obtain**  $n$  **where**  $x = (f \ \overset{\sim}{\sim} \ n) \ s$  **by** *auto*

**then show**  $x \in \text{orbit } f \ s$  **using** *assms* **by** (*cases n = 0*) (*auto simp: orbit-altdef*)

**qed** (*auto simp: orbit-altdef*)

**lemma** *orbit-altdef-permutation*:

**assumes** *permutation f* **shows**  $\text{orbit } f \ s = \{(f \ \overset{\sim}{\sim} \ n) \ s \mid n. \ \text{True}\}$

**using** *assms* **by** (*intro orbit-altdef-self-in permutation-self-in-orbit*)

**lemma** *orbit-altdef-bounded*:

**assumes**  $(f \ \overset{\sim}{\sim} \ n) \ s = s \ 0 < n$  **shows**  $\text{orbit } f \ s = \{(f \ \overset{\sim}{\sim} \ m) \ s \mid m. \ m < n\}$

**proof** –

**from** *assms* **have**  $s \in \text{orbit } f \ s$  **unfolding** *orbit-altdef* **by** *auto metis*

**then have**  $\text{orbit } f \ s = \{(f \ \overset{\sim}{\sim} \ m) \ s \mid m. \ \text{True}\}$  **by** (*rule orbit-altdef-self-in*)

**also have**  $\dots = \{(f \ \overset{\sim}{\sim} \ m) \ s \mid m. \ m < n\}$

**using** *assms* **by** (*auto simp: funpow-mod-eq intro: exI[where x=m mod n for m]*)

**finally show** *?thesis* .

**qed**

**lemma** *funpow-in-orbit*:

**assumes**  $s \in \text{orbit } f \ t$  **shows**  $(f \ \overset{\sim}{\sim} \ n) \ s \in \text{orbit } f \ t$

**using** *assms* **by** (*induct n*) (*auto intro: orbit.intros*)

**lemma** *finite-orbit*:

**assumes**  $s \in \text{orbit } f \ s$  **shows** *finite (orbit f s)*

**proof** –

**from** *assms* **obtain**  $n$  **where**  $n: \ 0 < n \ (f \ \overset{\sim}{\sim} \ n) \ s = s$  **by** (*auto simp: orbit-altdef*)

**then show** *?thesis* **by** (*auto simp: orbit-altdef-bounded*)

**qed**

**lemma** *self-in-orbit-step*:

**assumes**  $s \in \text{orbit } f \ s$  **shows**  $\text{orbit } f \ (f \ s) = \text{orbit } f \ s$

**proof** (*intro set-eqI iffI*)

**fix**  $t$  **assume**  $t \in \text{orbit } f \ s$  **then show**  $t \in \text{orbit } f \ (f \ s)$

**using** *assms* **by** (*auto intro: orbit-step orbit-sim-step*)

**qed** (*auto intro: orbit-subset*)

**lemma** *permutation-orbit-step*:

**assumes** *permutation f* **shows**  $\text{orbit } f \ (f \ s) = \text{orbit } f \ s$

**using** *assms* **by** (*intro self-in-orbit-step permutation-self-in-orbit*)

**lemma** *orbit-nonempty*:

```

orbit f s ≠ {}
using orbit.base by fastforce

lemma orbit-inv-eq:
  assumes permutation f
  shows orbit (inv f) x = orbit f x (is ?L = ?R)
proof -
  { fix g y assume A: permutation g y ∈ orbit (inv g) x
    have y ∈ orbit g x
    proof -
      have inv-g:  $\bigwedge y. x = g y \implies inv\ g\ x = y \bigwedge y. inv\ g\ (g\ y) = y$ 
      by (metis A(1) bij-inv-eq-iff permutation-bijective)+

      { fix y assume y ∈ orbit g x
        then have inv g y ∈ orbit g x
          by (cases) (simp-all add: inv-g A(1) permutation-self-in-orbit)
        } note inv-g-in-orb = this

      from A(2) show ?thesis
        by induct (simp-all add: inv-g-in-orb A permutation-self-in-orbit)
      qed
    } note orb-inv-ss = this

  have inv (inv f) = f
  by (simp add: assms inv-inv-eq permutation-bijective)
  then show ?thesis
  using orb-inv-ss[OF assms] orb-inv-ss[OF permutation-inverse[OF assms]] by
auto
qed

lemma cyclic-on-alldef:
  cyclic-on f S  $\longleftrightarrow$  S ≠ {}  $\wedge$  ( $\forall s \in S. S = orbit\ f\ s$ )
  unfolding cyclic-on-def by (auto intro: orbit.step orbit-swap orbit-trans)

lemma cyclic-on-funpow-in:
  assumes cyclic-on f S s ∈ S shows (f~n) s ∈ S
  using assms unfolding cyclic-on-def by (auto intro: funpow-in-orbit)

lemma finite-cyclic-on:
  assumes cyclic-on f S shows finite S
  using assms by (auto simp: cyclic-on-def finite-orbit)

lemma cyclic-on-singleI:
  assumes s ∈ S S = orbit f s shows cyclic-on f S
  using assms unfolding cyclic-on-def by blast

lemma inj-on-funpow-least:
  assumes (f~n) s = s  $\wedge$  m.  $\llbracket m < n; 0 < m \rrbracket \implies (f~m) s \neq s$ 

```

```

shows inj-on ( $\lambda k. (f \sim k) s$ ) { $0..<n$ }
proof -
{ fix k l assume A:  $k < n \wedge l < n \wedge k \neq l \implies (f \sim k) s = (f \sim l) s$ 
  define k' l' where  $k' = \min k l$  and  $l' = \max k l$ 
  with A have A':  $k' < l' \implies (f \sim k') s = (f \sim l') s \wedge l' < n$ 
  by (auto simp: min-def max-def)

  have  $s = (f \sim ((n - l') + l')) s$  using assms ( $l' < n$ ) by simp
  also have  $\dots = (f \sim (n - l')) ((f \sim l') s)$  by (simp add: funpow-add)
  also have  $(f \sim l') s = (f \sim k') s$  by (simp add: A')
  also have  $(f \sim (n - l')) \dots = (f \sim (n - l' + k')) s$  by (simp add: funpow-add)
  finally have  $(f \sim (n - l' + k')) s = s$  by simp
  moreover have  $n - l' + k' < n \wedge 0 < n - l' + k'$  using A' by linarith+
  ultimately have False using assms(2) by auto
}
then show ?thesis by (intro inj-onI) auto
qed

lemma cyclic-on-inI:
  assumes cyclic-on f S s  $s \in S$  shows  $f s \in S$ 
  using assms by (auto simp: cyclic-on-def intro: orbit.intros)

lemma bij-betw-funpow:
  assumes bij-betw f S S shows bij-betw  $(f \sim n) S S$ 
proof (induct n)
  case 0 then show ?case by (auto simp: id-def[symmetric])
next
  case (Suc n)
  then show ?case unfolding funpow.simps using assms by (rule bij-betw-trans)
qed

lemma orbit-FOO:
  assumes self:  $a \in \text{orbit } g a$ 
  and eq:  $\bigwedge x. x \in \text{orbit } g a \implies g' (f x) = f (g x)$ 
  shows  $f ' \text{orbit } g a = \text{orbit } g' (f a)$  (is ?L = ?R)
proof (intro set-eqI iffI)
  fix x assume  $x \in ?L$ 
  then obtain x0 where  $x0 \in \text{orbit } g a \wedge x = f x0$  by auto
  then show  $x \in ?R$ 
proof (induct arbitrary: x)
  case base then show ?case by (auto simp: self orbit.base eq[symmetric])
next
  case step then show ?case by cases (auto simp: eq[symmetric] orbit.intros)
qed
next
  fix x assume  $x \in ?R$ 
  then show  $x \in ?L$ 
proof (induct arbitrary: )

```

```

    case base then show ?case by (auto simp: self orbit.base eq)
  next
    case step then show ?case by cases (auto simp: eq orbit.intros)
  qed
qed

```

```

lemma cyclic-on-FOO:
  assumes cyclic-on f S
  assumes  $\bigwedge x. x \in S \implies g (h x) = h (f x)$ 
  shows cyclic-on g (h ` S)
  using assms by (auto simp: cyclic-on-def) (meson orbit-FOO)

```

```

lemma cyclic-on-f-in:
  assumes f permutes S cyclic-on f A f x ∈ A
  shows x ∈ A
proof -
  from assms have fx-in-orb: f x ∈ orbit f (f x) by (auto simp: cyclic-on-alldef)
  from assms have A = orbit f (f x) by (auto simp: cyclic-on-alldef)
  moreover
  then have ... = orbit f x using ⟨f x ∈ A⟩ by (auto intro: orbit-step orbit-subset)
  ultimately
  show ?thesis by (metis (no-types) orbit.simps permutes-inverses(2)[OF assms(1)])
qed

```

```

lemma permutes-not-in:
  assumes f permutes S x ∉ S shows f x = x
  using assms by (auto simp: permutes-def)

```

```

lemma orbit-cong0:
  assumes  $x \in A f \in A \rightarrow A \bigwedge y. y \in A \implies f y = g y$  shows orbit f x = orbit g x
proof -
  { fix n have (f  $\sim$  n) x = (g  $\sim$  n) x  $\wedge$  (f  $\sim$  n) x ∈ A
    by (induct n rule: nat.induct) (insert assms, auto)
  } then show ?thesis by (auto simp: orbit-altdef)
qed

```

```

lemma orbit-cong:
  assumes self-in: t ∈ orbit f t and eq:  $\bigwedge s. s \in orbit f t \implies g s = f s$ 
  shows orbit g t = orbit f t
  using assms(1) - assms(2) by (rule orbit-cong0) (auto simp: orbit.step eq)

```

```

lemma cyclic-cong:
  assumes  $\bigwedge s. s \in S \implies f s = g s$  shows cyclic-on f S = cyclic-on g S
proof -
  have ( $\exists s \in S. orbit f s = orbit g s$ )  $\implies$  cyclic-on f S = cyclic-on g S
    by (metis cyclic-on-alldef cyclic-on-def)
  then show ?thesis by (metis assms orbit-cong cyclic-on-def)
qed

```

**lemma** *permutes-comp-preserves-cyclic1*:  
**assumes**  $g$  permutes  $B$  cyclic-on  $f$   $C$   
**assumes**  $A \cap B = \{\}$   $C \subseteq A$   
**shows** cyclic-on  $(f \circ g)$   $C$   
**proof** –  
**have** \*:  $\bigwedge c. c \in C \implies f (g c) = f c$   
**using** *assms* **by** (*subst permutes-not-in*[**where**  $f=g$ ]) *auto*  
**with** *assms*(2) **show** ?thesis **by** (*simp cong: cyclic-cong*)  
**qed**

**lemma** *permutes-comp-preserves-cyclic2*:  
**assumes**  $f$  permutes  $A$  cyclic-on  $g$   $C$   
**assumes**  $A \cap B = \{\}$   $C \subseteq B$   
**shows** cyclic-on  $(f \circ g)$   $C$   
**proof** –  
**obtain**  $c$  **where**  $c: c \in C$   $C = \text{orbit } g c$   $c \in \text{orbit } g c$   
**using**  $\langle \text{cyclic-on } g C \rangle$  **by** (*auto simp: cyclic-on-def*)  
**then have**  $\bigwedge c. c \in C \implies f (g c) = g c$   
**using** *assms*  $c$  **by** (*subst permutes-not-in*[**where**  $f=g$ ]) (*auto intro: orbit.intros*)  
**with** *assms*(2) **show** ?thesis **by** (*simp cong: cyclic-cong*)  
**qed**

## 14.1 Orbits

**lemma** *permutes-orbit-subset*:  
**assumes**  $f$  permutes  $S$   $x \in S$  **shows** orbit  $f$   $x \subseteq S$   
**proof**  
**fix**  $y$  **assume**  $y \in \text{orbit } f x$   
**then show**  $y \in S$  **by** *induct* (*auto simp: permutes-in-image assms*)  
**qed**

**lemma** *cyclic-on-orbit'*:  
**assumes** permutation  $f$  **shows** cyclic-on  $f$  ( $\text{orbit } f x$ )  
**unfolding** *cyclic-on-alldef* **using** *orbit-nonempty*[ $\text{of } f x$ ]  
**by** (*auto intro: assms orbit-swap orbit-trans permutation-self-in-orbit*)

**lemma** *cyclic-on-orbit*:  
**assumes**  $f$  permutes  $S$  finite  $S$  **shows** cyclic-on  $f$  ( $\text{orbit } f x$ )  
**using** *assms* **by** (*intro cyclic-on-orbit'*) (*auto simp: permutation-permutes*)

**lemma** *orbit-cyclic-eq3*:  
**assumes** cyclic-on  $f$   $S$   $y \in S$  **shows** orbit  $f$   $y = S$   
**using** *assms* **unfolding** *cyclic-on-alldef* **by** *simp*

**lemma** *orbit-eq-singleton-iff*: orbit  $f$   $x = \{x\} \iff f x = x$  (**is** ? $L \iff$  ? $R$ )  
**proof**  
**assume**  $A: ?R$

```

{ fix y assume y ∈ orbit f x then have y = x
  by induct (auto simp: A)
} then show ?L by (metis orbit-nonempty singletonI subsetI subset-singletonD)
next
assume A: ?L
then have  $\bigwedge y. y \in \text{orbit } f \ x \implies f \ x = y$ 
  by - (erule orbit.cases, simp-all)
then show ?R using A by blast
qed

```

```

lemma eq-on-cyclic-on-iff1:
  assumes cyclic-on f S x ∈ S
  obtains f x ∈ S f x = x  $\longleftrightarrow$  card S = 1
proof
  from assms show f x ∈ S by (auto simp: cyclic-on-def intro: orbit.intros)
  from assms have S = orbit f x by (auto simp: cyclic-on-alldef)
  then have f x = x  $\longleftrightarrow$  S = {x} by (metis orbit-eq-singleton-iff)
  then show f x = x  $\longleftrightarrow$  card S = 1 using ⟨x ∈ S⟩ by (auto simp: card-Suc-eq)
qed

```

## 14.2 Decomposition of Arbitrary Permutations

**definition** *perm-restrict* ::  $(a \Rightarrow 'a) \Rightarrow 'a \text{ set} \Rightarrow (a \Rightarrow 'a)$  **where**  
*perm-restrict* f S x  $\equiv$  if x ∈ S then f x else x

```

lemma perm-restrict-comp:
  assumes A ∩ B = {} cyclic-on f B
  shows perm-restrict f A o perm-restrict f B = perm-restrict f (A ∪ B)
proof -
  have  $\bigwedge x. x \in B \implies f \ x \in B$  using ⟨cyclic-on f B⟩ by (rule cyclic-on-inI)
  with assms show ?thesis by (auto simp: perm-restrict-def fun-eq-iff)
qed

```

```

lemma perm-restrict-simps:
  x ∈ S  $\implies$  perm-restrict f S x = f x
  x ∉ S  $\implies$  perm-restrict f S x = x
  by (auto simp: perm-restrict-def)

```

```

lemma perm-restrict-perm-restrict:
  perm-restrict (perm-restrict f A) B = perm-restrict f (A ∩ B)
  by (auto simp: perm-restrict-def)

```

```

lemma perm-restrict-union:
  assumes perm-restrict f A permutes A perm-restrict f B permutes B A ∩ B = {}
  shows perm-restrict f A o perm-restrict f B = perm-restrict f (A ∪ B)
  using assms by (auto simp: fun-eq-iff perm-restrict-def permutes-def) (metis
  Diff-iff Diff-triv)

```



```

lemma perm-restrict-id[simp]:
  assumes  $f$  permutes  $S$  shows perm-restrict  $f$   $S = f$ 
  using assms by (auto simp: permutes-def perm-restrict-def)

lemma cyclic-on-perm-restrict:
  cyclic-on (perm-restrict  $f$   $S$ )  $S \longleftrightarrow$  cyclic-on  $f$   $S$ 
  by (simp add: perm-restrict-def cong: cyclic-cong)

lemma perm-restrict-diff-cyclic:
  assumes  $f$  permutes  $S$  cyclic-on  $f$   $A$ 
  shows perm-restrict  $f$  ( $S - A$ ) permutes ( $S - A$ )
proof -
  { fix  $y$ 
    have  $\exists x. \text{perm-restrict } f (S - A) x = y$ 
    proof cases
      assume  $A: y \in S - A$ 
      with  $\langle f \text{ permutes } S \rangle$  obtain  $x$  where  $f x = y$   $x \in S$ 
      unfolding permutes-def by auto metis
      moreover
      with  $A$  have  $x \notin A$  by (metis Diff-iff assms(2) cyclic-on-inI)
      ultimately
      have perm-restrict  $f$  ( $S - A$ )  $x = y$  by (simp add: perm-restrict-simps)
      then show ?thesis ..
    }
  next
    assume  $y \notin S - A$ 
    then have perm-restrict  $f$  ( $S - A$ )  $y = y$  by (simp add: perm-restrict-simps)
    then show ?thesis ..
  }
  qed
} note  $X = \text{this}$ 

{ fix  $x y$  assume perm-restrict  $f$  ( $S - A$ )  $x = \text{perm-restrict } f (S - A) y$ 
  with assms have  $x = y$ 
  by (auto simp: perm-restrict-def permutes-def split: if-splits intro: cyclic-on-f-in)
} note  $Y = \text{this}$ 

show ?thesis by (auto simp: permutes-def perm-restrict-simps  $X$  intro:  $Y$ )
qed

lemma orbit-eqI:
   $y = f x \implies y \in \text{orbit } f x$ 
   $z = f y \implies y \in \text{orbit } f x \implies z \in \text{orbit } f x$ 
  by (metis orbit.base) (metis orbit.step)

lemma permutes-decompose:
  assumes  $f$  permutes  $S$  finite  $S$ 
  shows  $\exists C. (\forall c \in C. \text{cyclic-on } f c) \wedge \bigcup C = S \wedge (\forall c1 \in C. \forall c2 \in C. c1 \neq c2 \implies c1 \cap c2 = \{\})$ 
  using assms(2,1)
proof (induction arbitrary:  $f$  rule: finite-psubset-induct)

```

```

case (psubset S)

show ?case
proof (cases S = {})
  case True then show ?thesis by (intro exI[where x={}]) auto
next
  case False
  then obtain s where s ∈ S by auto
  with ⟨f permutes S⟩ have orbit f s ⊆ S
    by (rule permutes-orbit-subset)
  have cyclic-orbit: cyclic-on f (orbit f s)
    using ⟨f permutes S⟩ ⟨finite S⟩ by (rule cyclic-on-orbit)

  let ?f' = perm-restrict f (S - orbit f s)

  have f s ∈ S using ⟨f permutes S⟩ ⟨s ∈ S⟩ by (auto simp: permutes-in-image)
  then have S - orbit f s ⊂ S using orbit.base[of f s] ⟨s ∈ S⟩ by blast
  moreover
  have ?f' permutes (S - orbit f s)
    using ⟨f permutes S⟩ cyclic-orbit by (rule perm-restrict-diff-cyclic)
  ultimately
  obtain C where C: ∧c. c ∈ C ⇒ cyclic-on ?f' c ∪ C = S - orbit f s
    ∀c1 ∈ C. ∀c2 ∈ C. c1 ≠ c2 → c1 ∩ c2 = {}
    using psubset.IH by metis

  { fix c assume c ∈ C
    then have *: ∧x. x ∈ c ⇒ perm-restrict f (S - orbit f s) x = f x
      using C(2) ⟨f permutes S⟩ by (auto simp add: perm-restrict-def)
    then have cyclic-on f c using C(1)[OF ⟨c ∈ C⟩] by (simp cong: cyclic-cong
add: *)
  } note in-C-cyclic = this

  have Un-ins: ∪ (insert (orbit f s) C) = S
    using ⟨∪ C = -⟩ ⟨orbit f s ⊆ S⟩ by blast

  have Disj-ins: (∀c1 ∈ insert (orbit f s) C. ∀c2 ∈ insert (orbit f s) C. c1 ≠ c2
→ c1 ∩ c2 = {})
    using C by auto

  show ?thesis
  by (intro conjI Un-ins Disj-ins exI[where x=insert (orbit f s) C])
    (auto simp: cyclic-orbit in-C-cyclic)
qed
qed

```

### 14.3 Funpow + Orbit

**lemma** funpow-dist-prop:

$$y \in \text{orbit } f \ x \implies (f \overset{\sim}{\sim} \text{funpow-dist } f \ x \ y) \ x = y$$

**unfolding** *funpow-dist-def* **by** (*rule LeastI-ex*) (*auto simp: orbit-altdef*)

**lemma** *funpow-dist-0-eq*:

**assumes**  $y \in \text{orbit } f \ x$  **shows**  $\text{funpow-dist } f \ x \ y = 0 \longleftrightarrow x = y$   
**using** *assms* **by** (*auto simp: funpow-dist-0 dest: funpow-dist-prop*)

**lemma** *funpow-dist-step*:

**assumes**  $x \neq y \ y \in \text{orbit } f \ x$  **shows**  $\text{funpow-dist } f \ x \ y = \text{Suc } (\text{funpow-dist } f \ (f \ x) \ y)$

**proof** –

**from**  $\langle y \in \cdot \rangle$  **obtain**  $n$  **where**  $(f \ \sim^n) \ x = y$  **by** (*auto simp: orbit-altdef*)  
**with**  $\langle x \neq y \rangle$  **obtain**  $n'$  **where**  $[simp]: n = \text{Suc } n'$  **by** (*cases n*) *auto*

**show** *?thesis*

**unfolding** *funpow-dist-def*

**proof** (*rule Least-Suc2*)

**show**  $(f \ \sim^n) \ x = y$  **by** *fact*

**then show**  $(f \ \sim^{n'}) \ (f \ x) = y$  **by** (*simp add: funpow-swap1*)

**show**  $(f \ \sim^0) \ x \neq y$  **using**  $\langle x \neq y \rangle$  **by** *simp*

**show**  $\forall k. ((f \ \sim^{\text{Suc } k}) \ x = y) = ((f \ \sim^k) \ (f \ x) = y)$

**by** (*simp add: funpow-swap1*)

**qed**

**qed**

**lemma** *funpow-dist1-prop*:

**assumes**  $y \in \text{orbit } f \ x$  **shows**  $(f \ \sim^{\text{funpow-dist1 } f \ x \ y}) \ x = y$

**by** (*metis assms funpow.simps(1) funpow-dist-0 funpow-dist-prop funpow-simp-1 funpow-swap1 id-apply orbit-step*)

**lemma** *funpow-neq-less-funpow-dist*:

**assumes**  $y \in \text{orbit } f \ x \ m \leq \text{funpow-dist } f \ x \ y \ n \leq \text{funpow-dist } f \ x \ y \ m \neq n$

**shows**  $(f \ \sim^m) \ x \neq (f \ \sim^n) \ x$

**proof** (*rule notI*)

**assume**  $A: (f \ \sim^m) \ x = (f \ \sim^n) \ x$

**define**  $m' \ n'$  **where**  $m' = \min \ m \ n$  **and**  $n' = \max \ m \ n$

**with**  $A$  *assms* **have**  $A': m' < n' \ (f \ \sim^{m'}) \ x = (f \ \sim^{n'}) \ x \ n' \leq \text{funpow-dist } f \ x \ y$

**by** (*auto simp: min-def max-def*)

**have**  $y = (f \ \sim^{\text{funpow-dist } f \ x \ y}) \ x$

**using**  $\langle y \in \cdot \rangle$  **by** (*simp only: funpow-dist-prop*)

**also have**  $\dots = (f \ \sim^{((\text{funpow-dist } f \ x \ y - n') + n')}) \ x$

**using**  $\langle n' \leq \cdot \rangle$  **by** *simp*

**also have**  $\dots = (f \ \sim^{((\text{funpow-dist } f \ x \ y - n') + m')}) \ x$

**by** (*simp add: funpow-add*  $\langle (f \ \sim^{m'}) \ x = \cdot \rangle$ )

**also have**  $(f \ \sim^{((\text{funpow-dist } f \ x \ y - n') + m')}) \ x \neq y$

**using**  $A'$  **by** (*intro funpow-dist-least*) *linarith*

**finally show** *False* **by** *simp*

qed

**lemma** *funpow-neq-less-funpow-dist1*:

**assumes**  $y \in \text{orbit } f \ x \ m < \text{funpow-dist1 } f \ x \ y \ n < \text{funpow-dist1 } f \ x \ y \ m \neq n$

**shows**  $(f \ \sim^m) \ x \neq (f \ \sim^n) \ x$

**proof** (rule *notI*)

**assume**  $A: (f \ \sim^m) \ x = (f \ \sim^n) \ x$

**define**  $m' \ n'$  **where**  $m' = \min \ m \ n$  **and**  $n' = \max \ m \ n$

**with**  $A$  **assms** **have**  $A': m' < n' \ (f \ \sim^{m'}) \ x = (f \ \sim^{n'}) \ x \ n' < \text{funpow-dist1 } f \ x$

$y$

**by** (auto simp: *min-def max-def*)

**have**  $y = (f \ \sim^{\text{funpow-dist1 } f \ x \ y}) \ x$

**using**  $\langle y \in \cdot \rangle$  **by** (simp only: *funpow-dist1-prop*)

**also have**  $\dots = (f \ \sim^{((\text{funpow-dist1 } f \ x \ y - n') + n')}) \ x$

**using**  $\langle n' < \cdot \rangle$  **by** simp

**also have**  $\dots = (f \ \sim^{((\text{funpow-dist1 } f \ x \ y - n') + m')}) \ x$

**by** (simp add: *funpow-add*  $\langle (f \ \sim^{m'}) \ x = \cdot \rangle$ )

**also have**  $(f \ \sim^{((\text{funpow-dist1 } f \ x \ y - n') + m')}) \ x \neq y$

**using**  $A'$  **by** (intro *funpow-dist1-least*) *linarith+*

**finally show** *False* **by** simp

qed

**lemma** *inj-on-funpow-dist*:

**assumes**  $y \in \text{orbit } f \ x$  **shows** *inj-on*  $(\lambda n. (f \ \sim^n) \ x) \ \{0.. \text{funpow-dist } f \ x \ y\}$

**using** *funpow-neq-less-funpow-dist*[*OF assms*] **by** (intro *inj-onI*) auto

**lemma** *inj-on-funpow-dist1*:

**assumes**  $y \in \text{orbit } f \ x$  **shows** *inj-on*  $(\lambda n. (f \ \sim^n) \ x) \ \{0.. < \text{funpow-dist1 } f \ x \ y\}$

**using** *funpow-neq-less-funpow-dist1*[*OF assms*] **by** (intro *inj-onI*) auto

**lemma** *orbit-conv-funpow-dist1*:

**assumes**  $x \in \text{orbit } f \ x$

**shows**  $\text{orbit } f \ x = (\lambda n. (f \ \sim^n) \ x) \ \{0.. < \text{funpow-dist1 } f \ x \ x\}$  (is ? $L = ?R$ )

**using** *funpow-dist1-prop*[*OF assms*]

**by** (auto simp: *orbit-altdef-bounded*[**where**  $n = \text{funpow-dist1 } f \ x \ x$ ])

**lemma** *funpow-dist1-prop1*:

**assumes**  $(f \ \sim^n) \ x = y \ 0 < n$  **shows**  $(f \ \sim^{\text{funpow-dist1 } f \ x \ y}) \ x = y$

**proof** –

**from** *assms* **have**  $y \in \text{orbit } f \ x$  **by** (auto simp: *orbit-altdef*)

**then show** ?*thesis* **by** (rule *funpow-dist1-prop*)

qed

**lemma** *funpow-dist1-dist*:

**assumes**  $\text{funpow-dist1 } f \ x \ y < \text{funpow-dist1 } f \ x \ z$

**assumes**  $\{y, z\} \subseteq \text{orbit } f \ x$

**shows**  $\text{funpow-dist1 } f x z = \text{funpow-dist1 } f x y + \text{funpow-dist1 } f y z$  (**is**  $?L = ?R$ )  
**proof** –  
**have**  $x-z$ :  $(f \sim \text{funpow-dist1 } f x z) x = z$  **using** *assms* **by** (*blast intro: funpow-dist1-prop*)  
**have**  $x-y$ :  $(f \sim \text{funpow-dist1 } f x y) x = y$  **using** *assms* **by** (*blast intro: funpow-dist1-prop*)  
  
**have**  $(f \sim (\text{funpow-dist1 } f x z - \text{funpow-dist1 } f x y)) y$   
 $= (f \sim (\text{funpow-dist1 } f x z - \text{funpow-dist1 } f x y)) ((f \sim \text{funpow-dist1 } f x y) x)$   
**using**  $x-y$  **by** *simp*  
**also have**  $\dots = z$   
**using** *assms*  $x-z$  **by** (*simp del: funpow.simps add: funpow-add-app*)  
**finally have**  $y-z\text{-diff}$ :  $(f \sim (\text{funpow-dist1 } f x z - \text{funpow-dist1 } f x y)) y = z$  .  
**then have**  $(f \sim \text{funpow-dist1 } f y z) y = z$   
**using** *assms* **by** (*intro funpow-dist1-prop1*) *auto*  
**then have**  $(f \sim \text{funpow-dist1 } f y z) ((f \sim \text{funpow-dist1 } f x y) x) = z$   
**using**  $x-y$  **by** *simp*  
**then have**  $(f \sim (\text{funpow-dist1 } f y z + \text{funpow-dist1 } f x y)) x = z$   
**by** (*simp del: funpow.simps add: funpow-add-app*)  
  
**show** *?thesis*  
**proof** (*rule antisym*)  
**from**  $y-z\text{-diff}$  **have**  $(f \sim \text{funpow-dist1 } f y z) y = z$   
**using** *assms* **by** (*intro funpow-dist1-prop1*) *auto*  
**then have**  $(f \sim \text{funpow-dist1 } f y z) ((f \sim \text{funpow-dist1 } f x y) x) = z$   
**using**  $x-y$  **by** *simp*  
**then have**  $(f \sim (\text{funpow-dist1 } f y z + \text{funpow-dist1 } f x y)) x = z$   
**by** (*simp del: funpow.simps add: funpow-add-app*)  
**then have**  $\text{funpow-dist1 } f x z \leq \text{funpow-dist1 } f y z + \text{funpow-dist1 } f x y$   
**using** *funpow-dist1-least not-less* **by** *fastforce*  
**then show**  $?L \leq ?R$  **by** *presburger*  
**next**  
**have**  $\text{funpow-dist1 } f y z \leq \text{funpow-dist1 } f x z - \text{funpow-dist1 } f x y$   
**using**  $y-z\text{-diff}$  *assms*(1) **by** (*metis not-less zero-less-diff funpow-dist1-least*)  
**then show**  $?R \leq ?L$  **by** *linarith*  
**qed**  
**qed**  
  
**lemma** *funpow-dist1-le-self*:  
**assumes**  $(f \sim m) x = x \ 0 < m \ y \in \text{orbit } f x$   
**shows**  $\text{funpow-dist1 } f x y \leq m$   
**proof** (*cases*  $x = y$ )  
**case** *True* **with** *assms* **show** *?thesis* **by** (*auto dest!: funpow-dist1-least*)  
**next**  
**case** *False*  
**have**  $(f \sim \text{funpow-dist1 } f x y) x = (f \sim (\text{funpow-dist1 } f x y \text{ mod } m)) x$   
**using** *assms* **by** (*simp add: funpow-mod-eq*)  
**with** *False*  $\langle y \in \text{orbit } f x \rangle$  **have**  $\text{funpow-dist1 } f x y \leq \text{funpow-dist1 } f x y \text{ mod } m$   
**by** *auto* (*metis funpow-dist-least funpow-dist-prop funpow-dist-step funpow-simp-l*)

*not-less*)  
**with**  $\langle m > 0 \rangle$  **show** *?thesis*  
**by** (*auto intro: order-trans*)  
**qed**

## 14.4 Permutation Domains

**definition** *has-dom* ::  $('a \Rightarrow 'a) \Rightarrow 'a \text{ set} \Rightarrow \text{bool}$  **where**  
*has-dom*  $f S \equiv \forall s. s \notin S \longrightarrow f s = s$

**lemma** *permutes-conv-has-dom*:  
 $f \text{ permutes } S \longleftrightarrow \text{bij } f \wedge \text{has-dom } f S$   
**by** (*auto simp: permutes-def has-dom-def bij-iff*)

## 15 Segments

**inductive-set** *segment* ::  $('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a \text{ set}$  **for**  $f a b$  **where**  
*base*:  $f a \neq b \Longrightarrow f a \in \text{segment } f a b \mid$   
*step*:  $x \in \text{segment } f a b \Longrightarrow f x \neq b \Longrightarrow f x \in \text{segment } f a b$

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

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

**lemma** *segment-altdef*:  
**assumes**  $b \in \text{orbit } f a$   
**shows**  $\text{segment } f a b = (\lambda n. (f \overset{\sim}{\sim} n) a) \cdot \{1..<\text{funpow-dist1 } f a b\}$  (**is**  $?L = ?R$ )  
**proof** (*intro set-eqI iffI*)  
**fix**  $x$  **assume**  $x \in ?L$   
**have**  $f a \neq b \Longrightarrow b \in \text{orbit } f (f a)$   
**using** *assms* **by** (*simp add: orbit-step*)  
**then have**  $*$ :  $f a \neq b \Longrightarrow 0 < \text{funpow-dist } f (f a) b$   
**using** *assms* **using** *gr0I funpow-dist-0-eq[OF  $\langle - \Longrightarrow b \in \text{orbit } f (f a) \rangle$ ]* **by** (*simp add: orbit.intros*)  
**from**  $\langle x \in ?L \rangle$  **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  $x=\text{Suc } n$  for  $n$ ]*)  
**qed**  
**next**  
**fix**  $x$  **assume**  $x \in ?R$   
**then obtain**  $n$  **where**  $(f \overset{\sim}{\sim} n) a = x$   $0 < n$   $n < \text{funpow-dist1 } f a b$  **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  $\sim$  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  $\sim$  n) x = x
using assms by (auto simp: segment-altdef Suc-le-eq)
then have neq-y: (f  $\sim$  (funpow-dist1 f x y - n)) x  $\neq$  y by (simp add: funpow-dist1-least)

```

```

have (f  $\sim$  (funpow-dist1 f x y - n)) x = (f  $\sim$  (funpow-dist1 f x y - n)) ((f  $\sim$  n) x)

```

```

using n by (simp add: funpow-add-app)

```

```

also have ... = (f  $\sim$  funpow-dist1 f x y) x

```

```

using (n < -) by (simp add: funpow-add-app)

```

```

also have ... = 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**  $(f \overset{\sim}{\sim} \text{funpow-dist1 } f x y) x = y \text{ funpow-dist1 } f x y < \text{funpow-dist1 } f x z$   
**proof**  
**from** *assms* **show**  $(f \overset{\sim}{\sim} \text{funpow-dist1 } f x y) x = y$   
**by** (*intro segmentD-orbit funpow-dist1-prop*)  
**moreover**  
**obtain** *n* **where**  $(f \overset{\sim}{\sim} n) x = y \ 0 < n \ n < \text{funpow-dist1 } f x z$   
**using** *assms* **by** (*auto simp: segment-altdef*)  
**moreover then have**  $\text{funpow-dist1 } f x y \leq n$  **by** (*meson funpow-dist1-least not-less*)  
**ultimately show**  $\text{funpow-dist1 } f x y < \text{funpow-dist1 } f x z$  **by** *linarith*  
**qed**

**lemma** *cyclic-split-segment*:  
**assumes** *S*: *cyclic-on* *f* *S*  $a \in S \ b \in S$  **and**  $a \neq b$   
**shows**  $S = \{a, b\} \cup \text{segment } f a b \cup \text{segment } f b a$  (**is**  $?L = ?R$ )  
**proof** (*intro set-eqI iffI*)  
**fix** *c* **assume**  $c \in ?L$   
**with** *S* **have**  $c \in \text{orbit } f a$  **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**  $\text{segment } f a b \subseteq \text{orbit } f a \ \text{segment } f b a \subseteq \text{orbit } f b$   
**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*:  $y \in \text{segment } f x z$   
**shows**  $\text{segment } f x z = \text{segment } f x y \cup \{y\} \cup \text{segment } f y z$  (**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**  $w \in \text{segment } f x y$  **then have**  $w \in \text{segment } f x z$   
**using** *segment-subset[OF y-in-seg]* **by** *auto* **}**  
**moreover**  
**{ assume**  $w \in \text{segment } f y z$  **then have**  $w \in \text{segment } f x z$   
**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**  $x \in \text{segment } f a b \ x \neq f a$   
**assumes** *inj* *f*  
**shows**  $\text{inv } f x \in \text{segment } f a b$



```

using assms by (auto elim: segment.cases)

lemma in-orbit-invI:
  assumes  $b \in \text{orbit } f \ a$ 
  assumes inj f
  shows  $a \in \text{orbit } (\text{inv } f) \ b$ 
  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  $A: x \in \text{segment } f \ a \ b \neq a$  and inj f
  shows  $x \in \text{segment } f \ a \ (f \ b)$ 
  using  $A$  by induct (auto intro: segment.intros dest: not-in-segment2D injD[OF
  <inj f>])

lemma inv-end-in-segment:
  assumes  $b \in \text{orbit } f \ a \ f \ a \neq b$  bij f
  shows  $\text{inv } f \ b \in \text{segment } f \ a \ b$ 
  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  $x \neq f \ x \implies f \ a = x \implies x \in \text{segment } f \ a \ (f \ x)$  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  $\text{inv } f \ x \in \text{segment } f \ a \ (f \ x) \implies x \in \text{segment } f \ a \ (f \ x)$ 
    using <bij f> <inj f> by (auto dest: segment.step simp: surj-f-inv-f bij-is-surj)
  then have  $\text{inv } f \ x \in \text{segment } f \ a \ x \implies x \in \text{segment } f \ a \ (f \ x)$ 
    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  $x \in \text{orbit } f \ a \ x \in \text{orbit } f \ b$  bij f
  shows  $\text{segment } f \ a \ x \subseteq \text{segment } f \ b \ x \vee \text{segment } f \ b \ x \subseteq \text{segment } f \ a \ x$ 
  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 *:  $\bigwedge f \ x \ y. y \in \text{segment } f \ x \ (f \ x) \implies \text{False}$  by (simp add: segment1-empty)

```

```

{ fix y z
  assume A: y ∈ segment f b (f x) y ∉ segment f a (f x) z ∈ segment f a (f x)
  from ⟨x ∈ orbit f a⟩ ⟨f x ∈ orbit f b⟩ ⟨y ∈ segment f b (f x)⟩
  have x ∈ orbit f b
    by (metis * inv-end-in-segment[OF - - ⟨bij f⟩] inv-f-eq[OF ⟨inj f⟩] segmentD-orbit)
  moreover
  with ⟨x ∈ orbit f a⟩ step.IH
  have segment f a (f x) ⊆ segment f b (f x) ∨ segment f b (f x) ⊆ 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.premis 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 ∈ ?L* **by** *induct (auto intro: segment.intros)*  
**qed**

## 16 Lists of Powers

**definition** *iterate* :: *nat ⇒ nat ⇒ ('a ⇒ 'a) ⇒ 'a ⇒ 'a list* **where**  
*iterate m n f x = map (λn. (f<sup>~</sup>n) x) [m..*n*]*

**lemma** *set-iterate*:  
*set (iterate m n f x) = (λk. (f<sup>~</sup>k) x) ‘ {m..*n*}*  
**by** (*auto simp: iterate-def*)

**lemma** *iterate-empty[simp]*: *iterate n m f x = []*  $\longleftrightarrow$  *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<sup>~</sup>(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*  
*Funpow*  
**begin**

## 17 Subdivision on Digraphs

**definition**

*subdivision-step* :: ('a, 'b) pre-digraph  $\Rightarrow$  ('b  $\Rightarrow$  'b)  $\Rightarrow$  ('a, 'b) pre-digraph  $\Rightarrow$  ('b  $\Rightarrow$  'b)  $\Rightarrow$  'a  $\times$  'a  $\times$  'a  $\Rightarrow$  'b  $\times$  'b  $\times$  'b  $\Rightarrow$  bool

**where**

*subdivision-step* *G* *rev-G* *H* *rev-H*  $\equiv$   $\lambda(u, v, w) (uv, uw, vw)$ .

*bidirected-digraph* *G* *rev-G*

$\wedge$  *bidirected-digraph* *H* *rev-H*

$\wedge$  *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* - {*uw*, *rev-G* *uw*}

$\wedge$  *uw*  $\in$  *arcs* *G*

$\wedge$  *distinct* [*uw*, *rev-H* *uw*, *vw*, *rev-H* *vw*]

$\wedge$  *arc-to-ends* *G* *uw* = (*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*:  $\llbracket$ *subdivision* *biG* *biI*; *subdivision-step* (*fst* *biI*) (*snd* *biI*) (*fst* *biH*) (*snd* *biH*) (*u,v,w*) (*uv,uw,vw*) $\rrbracket \Longrightarrow$  *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*  
 $(H, \text{rev-}H)$   $(I, \text{rev-}I)$   
**shows** *subdivision*  $(G, \text{rev-}G)$   $(I, \text{rev-}I)$   
**proof** –  
**have** *bidirected-digraph*  $(\text{fst } (G, \text{rev-}G))$   $(\text{snd } (G, \text{rev-}G))$  **using** *assms* **by**  $(\text{auto simp: subdivision-step-def})$   
**with** *assms*(2,1) **show** ?thesis  
**using** *assms*(2,1) **by** *induct*  $(\text{auto intro: subdivision.intros dest: subdivision-base})$   
**qed**

**lemma** *subdivision-trans*:  
**assumes** *subdivision*  $(G, \text{rev-}G)$   $(H, \text{rev-}H)$  *subdivision*  $(H, \text{rev-}H)$   $(I, \text{rev-}I)$   
**shows** *subdivision*  $(G, \text{rev-}G)$   $(I, \text{rev-}I)$   
**using** *assms* **by** *induction*  $(\text{auto intro: subdivision-step-rev})$

**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*  $\subseteq G$ : *bidirected-digraph*  $G$  *rev-G*  
**using** *subdiv-step unfolding subdivision-step-def* **by** *simp*

**sublocale** *subdiv-step*  $\subseteq H$ : *bidirected-digraph*  $H$  *rev-H*  
**using** *subdiv-step unfolding subdivision-step-def* **by** *simp*

**context** *subdiv-step* **begin**

**abbreviation**  $(\text{input})$   $vu \equiv \text{rev-}G$   $uv$   
**abbreviation**  $(\text{input})$   $wu \equiv \text{rev-}H$   $uw$   
**abbreviation**  $(\text{input})$   $wv \equiv \text{rev-}H$   $vw$

**lemma** *subdiv-compat: compatible*  $G$   $H$   
**using** *subdiv-step* **by**  $(\text{simp add: subdivision-step-def})$

**lemma** *arc-to-ends-eq: arc-to-ends*  $H = \text{arc-to-ends } G$   
**using** *subdiv-compat* **by**  $(\text{simp add: compatible-def arc-to-ends-def fun-eq-iff})$

**lemma** *head-eq: head*  $H = \text{head } G$   
**using** *subdiv-compat* **by**  $(\text{simp add: compatible-def fun-eq-iff})$

**lemma** *tail-eq: tail*  $H = \text{tail } G$   
**using** *subdiv-compat* **by**  $(\text{simp add: compatible-def fun-eq-iff})$

**lemma** *verts-H: verts*  $H = \text{verts } G \cup \{w\}$   
**using** *subdiv-step* **by**  $(\text{simp add: subdivision-step-def})$

**lemma** *verts-G: verts*  $G = \text{verts } H - \{w\}$   
**using** *subdiv-step* **by**  $(\text{auto simp: subdivision-step-def})$

**lemma** *arcs-H*:  $\text{arcs } H = \{uw, wu, vw, wv\} \cup \text{arcs } G - \{uv, vu\}$   
**using** *subdiv-step* **by** (*simp add: subdivision-step-def*)

**lemma** *not-in-verts-G*:  $w \notin \text{verts } G$   
**using** *subdiv-step* **by** (*simp add: subdivision-step-def*)

**lemma** *in-arcs-G*:  $\{uv, vu\} \subseteq \text{arcs } G$   
**using** *subdiv-step* **by** (*simp add: subdivision-step-def*)

**lemma** *not-in-arcs-H*:  $\{uv, vu\} \cap \text{arcs } H = \{\}$   
**using** *arcs-H* **by** *auto*

**lemma** *subdiv-ate*:  
 $\text{arc-to-ends } G \text{ } uv = (u, v)$   
 $\text{arc-to-ends } H \text{ } uv = (u, v)$   
 $\text{arc-to-ends } H \text{ } uw = (u, w)$   
 $\text{arc-to-ends } H \text{ } vw = (v, w)$   
**using** *subdiv-step subdiv-compat* **by** (*auto simp: subdivision-step-def arc-to-ends-def compatible-def*)

**lemma** *subdiv-ends[*simp*]*:  
 $\text{tail } G \text{ } uv = u \text{ head } G \text{ } uv = v \text{ tail } H \text{ } uv = u \text{ head } H \text{ } uv = v$   
 $\text{tail } H \text{ } uw = u \text{ head } H \text{ } uw = w \text{ tail } H \text{ } vw = v \text{ head } H \text{ } vw = w$   
**using** *subdiv-ate* **by** (*auto simp: arc-to-ends-def*)

**lemma** *subdiv-ends-G-rev[*simp*]*:  
 $\text{tail } G \text{ } (vu) = v \text{ head } G \text{ } (vu) = u \text{ tail } H \text{ } (vu) = v \text{ head } H \text{ } (vu) = u$   
**using** *in-arcs-G* **by** (*auto simp: tail-eq head-eq*)

**lemma** *subdiv-distinct-verts0*:  $u \neq w \text{ } v \neq w$   
**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*:  $\{uw, wu, vw, wv\} \subseteq \text{arcs } H$   
**proof** –  
{ **assume**  $uv = uw$   
**then have**  $\text{arc-to-ends } H \text{ } uv = \text{arc-to-ends } H \text{ } uw$  **by** *simp*  
**then have**  $v = w$  **by** (*simp add: arc-to-ends-def*)  
} **moreover**  
{ **assume**  $uv = vw$   
**then have**  $\text{arc-to-ends } H \text{ } uv = \text{arc-to-ends } H \text{ } vw$  **by** *simp*  
**then have**  $v = w$  **by** (*simp add: arc-to-ends-def*)  
} **moreover**  
{ **assume**  $vu = uw$   
**then have**  $\text{arc-to-ends } H \text{ } (vu) = \text{arc-to-ends } H \text{ } uw$  **by** *simp*  
**then have**  $u = w$  **by** (*simp add: arc-to-ends-def*)  
} **moreover**  
{ **assume**  $vu = vw$

**then have**  $\text{arc-to-ends } H (vu) = \text{arc-to-ends } H vw$  **by** *simp*  
**then have**  $u = w$  **by** (*simp add: arc-to-ends-def*)  
**} ultimately**  
**have**  $\{uw, vw\} \subseteq \text{arcs } H$  **unfolding** *arcs-H* **using** *subdiv-distinct-verts0* **by**  
*auto*  
**then show** *?thesis* **by** *auto*  
**qed**

**lemma** *subdiv-ends-H-rev[simp]*:  
 $\text{tail } H (wu) = w$   $\text{tail } H (wv) = w$   
 $\text{head } H (wu) = u$   $\text{head } H (wv) = v$   
**using** *in-arcs-H subdiv-ate* **by** *simp-all*

**lemma** *in-verts-G*:  $\{u, v\} \subseteq \text{verts } G$   
**using** *in-arcs-G* **by** (*auto dest: G.wellformed*)

**lemma** *not-in-arcs-G*:  $\{uw, wu, vw, wv\} \cap \text{arcs } G = \{\}$   
**proof** –  
**note**  $X = G.wellformed[\text{simplified tail-eq}[\text{symmetric}] \text{ head-eq}[\text{symmetric}]]$   
**show** *?thesis* **using** *not-in-verts-G in-arcs-H* **by** (*auto dest: X*)  
**qed**

**lemma** *subdiv-distinct-arcs*:  $\text{distinct } [uv, vu, uw, wu, vw, wv]$   
**proof** –  
**have**  $\text{distinct } [uw, wu, vw, wv]$   
**using** *subdiv-step* **by** (*simp add: subdivision-step-def*)  
**moreover**  
**have**  $\text{distinct } [uv, vu]$  **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*:  $\text{arcs } G = \text{arcs } H \cup \{uv, vu\} - \{uw, wu, vw, wv\}$   
**using** *in-arcs-G not-in-arcs-G* **unfolding** *arcs-H* **by** *auto*

**lemma** *subdiv-ate-H-rev*:  
 $\text{arc-to-ends } H (wu) = (w, u)$   
 $\text{arc-to-ends } H (wv) = (w, v)$   
**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  $\exists$  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  $\in$  arcs I then rev-H'
    x else x)
  for x

  have rev-H-injD:  $\bigwedge x y z. \text{rev-H}' x = z \implies \text{rev-H}' y = z \implies x \neq y \implies \text{False}$ 
  by (metis H'.arev-eq-iff)

  have rev-H'-simps: rev-H' uw = rev-H uw  $\wedge$  rev-H' vw = rev-H vw
   $\vee$  rev-H' uw = rev-H vw  $\wedge$  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  $\bigwedge x. x \in \text{arcs } H \implies \text{tail } H x = w \implies x \in \{\text{rev-H } uw, \text{rev-H } vw\}$ 

```



```

    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  $\bigwedge a. a \neq uv \implies a \neq \text{rev-I } uv \implies a \in \text{arcs } I \implies a \in \text{arcs } H$ 
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 *:  $\bigwedge a. \text{rev-H}' a = \text{rev-I } uv \longleftrightarrow a = \text{rev-I } uv$ 
by (metis H'.arev-arev H'.arev-dom insert-disjoint(1) not-in-arcs-H)
have **:  $\bigwedge a. uv = \text{rev-H}' a \longleftrightarrow a = uv$  using H'.arev-eq not-in-arcs-H by
force
have ***:  $\bigwedge a. a \in \text{arcs } I \implies \text{rev-H}' a \in \text{arcs } I$ 
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  $\bigwedge x. \text{rev-H}' x = uv \longleftrightarrow x = uv \bigwedge x. \text{rev-H}' x = \text{rev-I } uv \longleftrightarrow x = \text{rev-I } uv$ 
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, uv, 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')  $\exists$  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* ( $G, rev-G$ ) ( $H, rev-H$ )  $x \in verts\ G$   
**shows**  $x \in verts\ H$   
**using** *assms* **by** *induct* (*auto simp: subdiv-step.verts-H subdiv-step-def*)

## 17.1 Subdivision on Pair Digraphs

In this section, we introduce specialized rules for pair digraphs.

**abbreviation** *subdivision-pair*  $G\ H \equiv$  *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*  $\Rightarrow$  '*a* *pair-pre-digraph*  $\Rightarrow$  *bool*  
**for**  $G$  **where**  
*base*: *pair-bidirected-digraph*  $G \Longrightarrow$  *pair-sd*  $G\ G$   
| *divide*:  $\bigwedge e\ w\ H. \llbracket e \in \text{parcs}\ H; w \notin \text{pverts}\ H; \text{pair-sd}\ G\ H \rrbracket$   
 $\Longrightarrow$  *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  $\langle subdivision-step - - - - - \rangle$ 
  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  $\langle subdivision-step - - - - - \rangle$  unfolding  $\langle I = - \rangle\ \langle rev-I = - \rangle$  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  $parcs\ H = parcs\ I' - \{(u,v), (v,u)\} \cup \{(u,w), (w,u), (w,v), (v,w)\}$ 
  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  $bidirected-digraphI-subdivision = bidirected-digraphI-pair-sd[unfolded\ sub-$ 
 $division-pair-conv]$ 

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

## 18 Euler Trails in Digraphs

In this section we prove the well-known theorem characterizing the existence of an Euler Trail in an directed graph

### 18.1 Trails and Euler Trails

**definition** (in *pre-digraph*) *euler-trail* :: 'a  $\Rightarrow$  'b *awalk*  $\Rightarrow$  'a  $\Rightarrow$  bool **where**  
*euler-trail* u p v  $\equiv$  trail u p v  $\wedge$  set p = arcs G  $\wedge$  set (awalk-verts u p) = verts G

**context** *wf-digraph* **begin**

**lemma** *finite-distinct*:

assumes *finite* A **shows** *finite* {p. distinct p  $\wedge$  set p  $\subseteq$  A}

**proof** –

have {p. distinct p  $\wedge$  set p  $\subseteq$  A}  $\subseteq$  {p. set p  $\subseteq$  A  $\wedge$  length p  $\leq$  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.  $\exists$  u v. trail u p v}

**proof** –

have {p.  $\exists$  u v. trail u p v}  $\subseteq$  {p. distinct p  $\wedge$  set p  $\subseteq$  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  $\in$  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**  $A': \text{awalk } u\ q\ w\ \text{awalk } w\ r\ u$   
**by** *atomize-elim* (*rule awalk-decomp*)

**then have**  $B: \text{awalk } w\ (r @ q)\ w$  **by** *auto*

**have**  $C: \text{set } (\text{awalk-verts } w\ (r @ q)) = \text{set } (\text{awalk-verts } u\ p)$   
**using**  $\langle \text{awalk } u\ p\ w\ A\ A' \rangle$  **by** (*auto simp: set-awalk-verts-append*)

**from**  $A\ B\ C$  **show** *?thesis* ..

**qed**

**lemma** *rotate-trailE*:

**assumes**  $\text{trail } u\ p\ u\ w \in \text{set } (\text{awalk-verts } u\ p)$   
**obtains**  $q\ r$  **where**  $p = q @ r$   $\text{trail } w\ (r @ q)\ w \in \text{set } (\text{awalk-verts } w\ (r @ q)) =$   
 $\text{set } (\text{awalk-verts } u\ p)$   
**using** *assms* **by** – (*rule rotate-awalkE*[**where**  $u=u$  **and**  $p=p$  **and**  $w=w$ ], *auto simp: trail-def*)

**lemma** *rotate-trailE'*:

**assumes**  $\text{trail } u\ p\ u\ w \in \text{set } (\text{awalk-verts } u\ p)$   
**obtains**  $q$  **where**  $\text{trail } w\ q\ w \in \text{set } q = \text{set } p \in \text{set } (\text{awalk-verts } w\ q) = \text{set } (\text{awalk-verts } u\ p)$

**proof** –

**from** *assms* **obtain**  $q\ r$  **where**  $p = q @ r$   $\text{trail } w\ (r @ q)\ w \in \text{set } (\text{awalk-verts } w\ (r @ q)) = \text{set } (\text{awalk-verts } u\ p)$

**by** (*rule rotate-trailE*)

**then have**  $\text{set } (r @ q) = \text{set } p$  **by** *auto*

**show** *?thesis* **by** (*rule that*) *fact+*

**qed**

**lemma** *sym-reachableI-in-awalk*:

**assumes** *walk*:  $\text{awalk } u\ p\ v$  **and**

$w1: w1 \in \text{set } (\text{awalk-verts } u\ p)$  **and**  $w2: w2 \in \text{set } (\text{awalk-verts } u\ p)$

**shows**  $w1 \rightarrow^*_{\text{mk-symmetric } G} w2$

**proof** –

**from** *walk*  $w1$  **obtain**  $q\ r$  **where**  $p = q @ r$   $\text{awalk } u\ q\ w1\ \text{awalk } w1\ r\ v$

**by** (*atomize-elim*) (*rule awalk-decomp*)

**then have**  $w2\text{-in}: w2 \in \text{set } (\text{awalk-verts } u\ q) \cup \text{set } (\text{awalk-verts } w1\ r)$

**using**  $w2$  **by** (*auto simp: set-awalk-verts-append*)

**show** *?thesis*

**proof** *cases*

**assume**  $A: w2 \in \text{set } (\text{awalk-verts } u\ q)$

**obtain**  $s$  **where**  $\text{awalk } w2\ s\ w1$

**using** *awalk-decomp*[*OF*  $\langle \text{awalk } u\ q\ w1 \rangle A$ ] **by** *blast*

**then have**  $w2 \rightarrow^*_{\text{mk-symmetric } G} w1$

**by** (*intro reachable-awalkI reachable-mk-symmetricI*)

**with** *symmetric-mk-symmetric* **show** *?thesis* **by** (*rule symmetric-reachable*)

```

next
  assume  $w2 \notin \text{set } (\text{awalk-verts } u \ q)$ 
  then have  $A: w2 \in \text{set } (\text{awalk-verts } w1 \ r)$ 
    using  $w2\text{-in}$  by blast
  obtain  $s$  where  $\text{awalk } w1 \ s \ w2$ 
    using  $\text{awalk-decomp}[OF \ (\text{awalk } w1 \ r \ v) \ A]$  by blast
  then show  $w1 \rightarrow^* \text{mk-symmetric } G \ w2$ 
    by (intro  $\text{reachable-awalkI}$   $\text{reachable-mk-symmetricI}$ )
qed
qed

```

**lemma** *euler-imp-connected*:

```

assumes  $\text{euler-trail } u \ p \ v$  shows  $\text{connected } G$ 
proof -
  { have  $\text{verts } G \neq \{\}$  using  $\text{assms}$  unfolding  $\text{euler-trail-def}$   $\text{trail-def}$  by auto }
  moreover
  { fix  $w1 \ w2$  assume  $w1 \in \text{verts } G \ w2 \in \text{verts } G$ 
    then have  $\text{awalk } u \ p \ v \ w1 \in \text{set } (\text{awalk-verts } u \ p) \ w2 \in \text{set } (\text{awalk-verts } u \ p)$ 
      using  $\text{assms}$  by (auto simp:  $\text{euler-trail-def}$   $\text{trail-def}$ )
    then have  $w1 \rightarrow^* \text{mk-symmetric } G \ w2$  by (rule  $\text{sym-reachableI-in-awalk}$ ) }
  ultimately show  $\text{connected } G$  by (rule  $\text{connectedI}$ )
qed

end

```

## 18.2 Arc Balance of Walks

**context** *pre-digraph* begin

**definition** *arc-set-balance* ::  $'a \Rightarrow 'b \ \text{set} \Rightarrow \text{int}$  **where**

$\text{arc-set-balance } w \ A = \text{int } (\text{card } (\text{in-arcs } G \ w \ \cap \ A)) - \text{int } (\text{card } (\text{out-arcs } G \ w \ \cap \ A))$

**definition** *arc-set-balanced* ::  $'a \Rightarrow 'b \ \text{set} \Rightarrow 'a \Rightarrow \text{bool}$  **where**

$\text{arc-set-balanced } u \ A \ v \equiv$   
 if  $u = v$  then  $(\forall w \in \text{verts } G. \ \text{arc-set-balance } w \ A = 0)$   
 else  $(\forall w \in \text{verts } G. \ (w \neq u \wedge w \neq v) \longrightarrow \text{arc-set-balance } w \ A = 0)$   
 $\wedge \text{arc-set-balance } u \ A = -1$   
 $\wedge \text{arc-set-balance } v \ A = 1$

**abbreviation** *arc-balance* ::  $'a \Rightarrow 'b \ \text{awalk} \Rightarrow \text{int}$  **where**

$\text{arc-balance } w \ p \equiv \text{arc-set-balance } w \ (\text{set } p)$

**abbreviation** *arc-balanced* ::  $'a \Rightarrow 'b \ \text{awalk} \Rightarrow 'a \Rightarrow \text{bool}$  **where**

$\text{arc-balanced } u \ p \ v \equiv \text{arc-set-balanced } u \ (\text{set } p) \ v$

**lemma** *arc-set-balanced-all*:

$\text{arc-set-balanced } u \ (\text{arcs } G) \ v =$

(if  $u = v$  then  $(\forall w \in \text{verts } G. \text{in-degree } G w = \text{out-degree } G w)$   
 else  $(\forall w \in \text{verts } G. (w \neq u \wedge w \neq v) \longrightarrow \text{in-degree } G w = \text{out-degree } G w)$   
 $\wedge \text{in-degree } G u + 1 = \text{out-degree } G u$   
 $\wedge \text{out-degree } G v + 1 = \text{in-degree } G v$   
**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**  $\text{arc-set-balance } w (\text{insert } e (\text{set } es)) = \text{arc-set-balance } w \{e\} + \text{arc-balance } w es$

**proof** –

**from** *assms* **have**  $e \notin \text{set } es \ e \in \text{arcs } G$  **by** (*auto simp: trail-def*)

**with**  $\langle e \notin \text{set } es \rangle$  **show** *?thesis*

**apply** (*cases*  $w = \text{tail } G e$ )

**apply** (*case-tac*  $[\!|] w = \text{head } G e$ )

**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* ( $\text{head } G e$ )  $es v u = \text{tail } G e \ e \in \text{arcs } G$

**by** (*auto simp: awalk-Cons-iff trail-def*)

**moreover**

**have**  $\bigwedge w. \text{arc-balance } w [e] = (\text{if } w = \text{tail } G e \wedge \text{tail } G e \neq \text{head } G e \text{ then } -1$   
 else if  $w = \text{head } G e \wedge \text{tail } G e \neq \text{head } G e$  then 1 else 0)

**using**  $\langle e \in \cdot \rangle$  **by** (*case-tac*  $w = \text{tail } G e$ ) (*auto simp: arc-set-balance-def*)

**ultimately show** *?case*

**by** (*auto simp: arc-set-balanced-def arc-balance-Cons[OF  $\langle \text{trail } u \ - \ \rangle$ ]*)

**qed**

**lemma** *trail-arc-balanceE:*

**assumes** *trail*  $u p v$

**obtains**  $\bigwedge w. \llbracket u = v \vee (w \neq u \wedge w \neq v); w \in \text{verts } G \rrbracket$

$\implies \text{arc-balance } w p = 0$



**and**  $\llbracket u \neq v \rrbracket \implies \text{arc-balance } u \ p = - 1$   
**and**  $\llbracket u \neq v \rrbracket \implies \text{arc-balance } v \ p = 1$   
**using** *arc-balancedI-trail*[*OF assms*] **unfolding** *arc-set-balanced-def* **by** (*intro that*) (*metis,presburger+*)

**end**

### 18.3 Closed Euler Trails

**lemma** (*in wf-digraph*) *awalk-vertex-props*:  
**assumes** *awalk*  $u \ p \ v \ p \neq \llbracket$   
**assumes**  $\bigwedge w. w \in \text{set } (\text{awalk-verts } u \ p) \implies P \ w \ \vee \ Q \ w$   
**assumes**  $P \ u \ Q \ v$   
**shows**  $\exists e \in \text{set } p. P \ (\text{tail } G \ e) \wedge Q \ (\text{head } G \ e)$   
**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)  $\wedge$  Q (head G e)*)  
**case** *False*  
**then have**  $P \ (\text{head } G \ e) \ \vee \ Q \ (\text{head } G \ e)$   
**using** *cons.prem*s(1) *cons.prem*s(2)[*of head G e*]  
**by** (*auto simp: awalk-Cons-iff set-awalk-verts*)  
**then have**  $P \ (\text{tail } G \ e) \wedge P \ (\text{head } G \ e)$   
**using** *False using cons.prem*s(1,3) **by** *auto*  
  
**then have**  $\exists e \in \text{set } es. P \ (\text{tail } G \ e) \wedge Q \ (\text{head } G \ e)$   
**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*  $G \ \text{arcs } G \neq \{\}$   
**shows**  $\text{verts } G = \text{tail } G \ \text{' arcs } G \cup \text{head } G \ \text{' arcs } G$   
**proof** –  
{ **assume**  $\text{verts } G = \{\}$  **then have** *?thesis* **by** (*auto dest: tail-in-verts*) }  
**moreover**  
{ **assume**  $\exists v. \text{verts } G = \{v\}$   
**then obtain**  $v$  **where**  $\text{verts } G = \{v\}$  **by** (*auto simp: card-Suc-eq*)  
**moreover**  
**with**  $\langle \text{arcs } G \neq \{\} \rangle$  **obtain**  $e$  **where**  $e \in \text{arcs } G \ \text{tail } G \ e = v \ \text{head } G \ e = v$   
**by** (*auto dest: tail-in-verts head-in-verts*)  
**moreover have**  $\text{tail } G \ \text{' arcs } G \cup \text{head } G \ \text{' arcs } G \subseteq \text{verts } G$  **by** *auto*  
**ultimately have** *?thesis* **by** *auto* }  
**moreover**  
{ **assume**  $A: \exists u \ v. u \in \text{verts } G \wedge v \in \text{verts } G \wedge u \neq v$   
{ **fix**  $u$  **assume**  $u \in \text{verts } G$

**interpret**  $S$ : *pair-wf-digraph mk-symmetric*  $G$  **by rule**  
**from**  $A$  **obtain**  $v$  **where**  $v \in \text{verts } G \ u \neq v$  **by blast**  
**then obtain**  $p$  **where**  $S.\text{awalk } u \ p \ v$   
**using**  $\langle \text{connected } G \rangle \langle u \in \text{verts } G \rangle$  **by**  $(\text{auto elim: connected-awalkE})$   
**with**  $\langle u \neq v \rangle$  **obtain**  $e$  **where**  $e \in \text{parcs } (mk\text{-symmetric } G)$   $\text{fst } e = u$   
**by**  $(\text{metis } S.\text{awalk-Cons-iff } S.\text{awalk-empty-ends list-exhaust2})$   
**then obtain**  $e'$  **where**  $\text{tail } G \ e' = u \vee \text{head } G \ e' = u \ e' \in \text{arcs } G$   
**by**  $(\text{force simp: parcs-mk-symmetric})$   
**then have**  $u \in \text{tail } G \ \cup \ \text{arcs } G \cup \text{head } G \ \cup \ \text{arcs } G$  **by auto** }  
**then have**  $?thesis$  **by auto** }  
**ultimately show**  $?thesis$  **by blast**  
**qed**

**lemma** (*in wf-digraph*) *connected-arcs-empty*:  
**assumes**  $\text{connected } G \ \text{arcs } G = \{\}$   $\text{verts } G \neq \{\}$  **obtains**  $v$  **where**  $\text{verts } G = \{v\}$   
**proof**  $(\text{atomize-elim, rule ccontr})$   
**assume**  $A$ :  $\neg (\exists v. \text{verts } G = \{v\})$

**interpret**  $S$ : *pair-wf-digraph mk-symmetric*  $G$  **by rule**

**from**  $\langle \text{verts } G \neq \{\} \rangle$  **obtain**  $u$  **where**  $u \in \text{verts } G$  **by auto**  
**with**  $A$  **obtain**  $v$  **where**  $v \in \text{verts } G \ u \neq v$  **by auto**

**from**  $\langle \text{connected } G \rangle \langle u \in \text{verts } G \rangle \langle v \in \text{verts } G \rangle$   
**obtain**  $p$  **where**  $S.\text{awalk } u \ p \ v$   
**using**  $\langle \text{connected } G \rangle \langle u \in \text{verts } G \rangle$  **by**  $(\text{auto elim: connected-awalkE})$   
**with**  $\langle u \neq v \rangle$  **obtain**  $e$  **where**  $e \in \text{parcs } (mk\text{-symmetric } G)$   
**by**  $(\text{metis } S.\text{awalk-Cons-iff } S.\text{awalk-empty-ends list-exhaust2})$   
**with**  $\langle \text{arcs } G = \{\} \rangle$  **show**  $\text{False}$   
**by**  $(\text{auto simp: parcs-mk-symmetric})$

**qed**

**lemma** (*in wf-digraph*) *euler-trail-conv-connected*:

**assumes**  $\text{connected } G$   
**shows**  $\text{euler-trail } u \ p \ v \longleftrightarrow \text{trail } u \ p \ v \wedge \text{set } p = \text{arcs } G$  (**is**  $?L \longleftrightarrow ?R$ )

**proof**

**assume**  $?R$  **show**  $?L$

**proof cases**

**assume**  $p = []$  **with**  $\text{assms } \langle ?R \rangle$  **show**  $?thesis$

**by**  $(\text{auto simp: euler-trail-def trail-def awalk-def elim: connected-arcs-empty})$

**next**

**assume**  $p \neq []$  **then have**  $\text{arcs } G \neq \{\}$  **using**  $\langle ?R \rangle$  **by auto**

**with**  $\text{assms } \langle ?R \rangle \langle p \neq [] \rangle$  **show**  $?thesis$

**by**  $(\text{auto simp: euler-trail-def trail-def set-awalk-verts-not-Nil connected-verts})$

**qed**

**qed**  $(\text{simp add: euler-trail-def})$

**lemma** (*in wf-digraph*) *awalk-connected*:

**assumes** *connected*  $G$  *awalk*  $u$   $p$   $v$  *set*  $p \neq \text{arcs } G$   
**shows**  $\exists e. e \in \text{arcs } G - \text{set } p \wedge (\text{tail } G e \in \text{set } (\text{awalk-verts } u p) \vee \text{head } G e \in \text{set } (\text{awalk-verts } u p))$   
**proof** (*rule ccontr*)  
**assume**  $A: \neg ?thesis$

**obtain**  $e$  **where**  $e \in \text{arcs } G - \text{set } p$   
**using** *assms* **by** (*auto simp: trail-def*)  
**with**  $A$  **have**  $\text{tail } G e \notin \text{set } (\text{awalk-verts } u p) \wedge \text{tail } G e \in \text{verts } G$   
**by** *auto*

**interpret**  $S$ : *pair-wf-digraph mk-symmetric*  $G$  ..

**have**  $u \in \text{verts } G$  **using**  $\langle \text{awalk } u p v \rangle$  **by** (*auto simp: awalk-hd-in-verts*)  
**with**  $\langle \text{tail } G e \in \cdot \rangle$  **and**  $\langle \text{connected } G \rangle$   
**obtain**  $q$  **where**  $q: S.\text{awalk } u q (\text{tail } G e)$   
**by** (*auto elim: connected-awalkE*)

**have**  $u \in \text{set } (\text{awalk-verts } u p)$   
**using**  $\langle \text{awalk } u p v \rangle$  **by** (*auto simp: set-awalk-verts*)

**have**  $q \neq []$  **using**  $\langle u \in \text{set } \cdot \rangle \langle \text{tail } G e \notin \cdot \rangle q$  **by** *auto*

**have**  $\exists e \in \text{set } q. \text{fst } e \in \text{set } (\text{awalk-verts } u p) \wedge \text{snd } e \notin \text{set } (\text{awalk-verts } u p)$   
**by** (*rule S.awalk-vertex-props[OF  $\langle S.\text{awalk } - - \cdot \rangle \langle q \neq [] \rangle]$ )* (*auto simp:  $\langle u \in \text{set } \cdot \rangle \langle \text{tail } G e \notin \cdot \rangle$* )  
**then obtain**  $se'$  **where**  $se': se' \in \text{set } q \wedge \text{fst } se' \in \text{set } (\text{awalk-verts } u p) \wedge \text{snd } se' \notin \text{set } (\text{awalk-verts } u p)$   
**by** *auto*

**from**  $se'$  **have**  $se' \in \text{parcs } (\text{mk-symmetric } G)$  **using**  $q$  **by** *auto*  
**then obtain**  $e'$  **where**  $e' \in \text{arcs } G \wedge (\text{tail } G e' = \text{fst } se' \wedge \text{head } G e' = \text{snd } se') \vee (\text{tail } G e' = \text{snd } se' \wedge \text{head } G e' = \text{fst } se')$   
**by** (*auto simp: parcs-mk-symmetric*)  
**moreover**  
**then have**  $e' \notin \text{set } p$  **using**  $se'$   $\langle \text{awalk } u p v \rangle$   
**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*  $p \neq \text{arcs } G$   
**shows**  $\exists e. e \in \text{arcs } G - \text{set } p \wedge (\text{tail } G e \in \text{set } (\text{awalk-verts } u p) \vee \text{head } G e \in \text{set } (\text{awalk-verts } u p))$   
**using** *assms* **by** (*intro awalk-connected*) (*auto simp: trail-def*)

**theorem** (*in fin-digraph*) *closed-euler1*:

**assumes** *con: connected*  $G$

```

assumes deg:  $\bigwedge u. u \in \text{verts } G \implies \text{in-degree } G \ u = \text{out-degree } G \ u$ 
shows  $\exists u \ p. \text{euler-trail } u \ p \ u$ 
proof –
from con obtain  $u$  where  $u \in \text{verts } G$  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  $\exists u' \ p' \ v'. \text{euler-trail } u' \ p' \ v'$ 
  proof (induct card (arcs  $G$ ) – length  $p$  arbitrary:  $u \ p \ v$ )
    case 0
    then have  $u \in \text{verts } G$  by (auto simp: trail-def)

    have set  $p \subseteq \text{arcs } G$  using (trail  $u \ p \ v$ ) by (auto simp: trail-def)
    with 0 have set  $p = \text{arcs } G$ 
      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 neg: set  $p \neq \text{arcs } G \ u \in \text{verts } G$ 
  by (auto simp: trail-def distinct-card[symmetric])

  show ?case
  proof (cases  $u = v$ )
    assume  $u \neq v$ 
    then have arc-balance  $u \ p = -1$ 
      using Suc neg by (auto elim: trail-arc-balanceE)
    then have card (in-arcs  $G \ u \cap \text{set } p$ ) < card (out-arcs  $G \ u \cap \text{set } p$ )
      unfolding arc-set-balance-def by auto
    also have ...  $\leq$  card (out-arcs  $G \ u$ )
      by (rule card-mono) auto
    finally have card (in-arcs  $G \ u \cap \text{set } p$ ) < card (in-arcs  $G \ u$ )
      using deg[OF  $\langle u \in \cdot \rangle$ ] unfolding out-degree-def in-degree-def by simp
    then have in-arcs  $G \ u - \text{set } p \neq \{\}$ 
      by (auto dest: card-psubset[rotated 2])
    then obtain  $a$  where  $a \in \text{arcs } G \ \text{head } G \ a = u \ a \notin \text{set } p$ 
      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 neg con Suc
obtain  $a$  where  $a\text{-in}: a \in \text{arcs } G - \text{set } p$ 
  and  $a\text{-end}: (\text{tail } G \ a \in \text{set } (\text{awalk-verts } u \ p) \vee \text{head } G \ a \in \text{set } (\text{awalk-verts } u \ p))$ 
  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 w⟩ 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 w⟩ 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  $\bigwedge u. u \in \text{verts } G \implies \text{in-degree } G u = \text{out-degree } G u$  (is  $\bigwedge u. - \implies ?eq\text{-deg } u$ )
proof -
  from assms show connected G by (rule euler-imp-connected)
next
  fix v assume A: v ∈ verts G
  with assms show ?eq-deg v by (rule closed-euler-imp-eq-degree)
qed

```

```

corollary (in fin-digraph) closed-euler:
   $(\exists u p. \text{euler-trail } u p u) \iff \text{connected } G \wedge (\forall u \in \text{verts } G. \text{in-degree } G u = \text{out-degree } G u)$ 
  by (auto dest: closed-euler1 closed-euler2)

```

## 18.4 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  $(\text{verts} = \{0::'a, 1::'a\}, \text{arcs} = \{()\}, \text{tail} = \lambda-. 0::'a, \text{head} = \lambda-. 1::'a)$ . 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  $'a \times \text{nat} \times 'a$ . Hence, we first characterize the existence of euler trail for the infinite arc type  $'a \times \text{nat} \times 'a$  and transfer that result back to arbitrary arc types.

```

lemma open-euler-infinite-label:
  fixes G :: ('a, 'a × nat × 'a) pre-digraph
  assumes fin-digraph G
  assumes [simp]: tail G = fst head G = snd o snd
  assumes con: connected G

```

```

assumes uv:  $u \in \text{verts } G \ v \in \text{verts } G$ 
assumes deg:  $\bigwedge w. \llbracket w \in \text{verts } G; u \neq w; v \neq w \rrbracket \implies \text{in-degree } G \ w = \text{out-degree } G \ w$ 
assumes deg-in:  $\text{in-degree } G \ u + 1 = \text{out-degree } G \ u$ 
assumes deg-out:  $\text{out-degree } G \ v + 1 = \text{in-degree } G \ v$ 
shows  $\exists p. \text{pre-digraph.euler-trail } G \ u \ p \ v$ 
proof –
define label ::  $'a \times \text{nat} \times 'a \Rightarrow \text{nat}$  where [simp]: label = fst o snd

interpret fin-digraph G by fact

have finite (label ‘ arcs G) by auto
moreover have  $\neg \text{finite } (UNIV :: \text{nat set})$  by blast
ultimately obtain l where  $l \notin \text{label } ' \text{arcs } G$  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:  $?e \notin \text{arcs } G$ 
using  $l \notin \rightarrow$  by (auto simp: image-def)

let ?H = add-arc ?e
– We define a graph which has an closed euler trail

have [simp]:  $\text{verts } ?H = \text{verts } G$  using uv by simp
have [intro]:  $\bigwedge a. \text{compatible } (\text{add-arc } a) \ G$  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  $\exists u \ p. \ H.\text{euler-trail } ?e \ u \ p \ u$ 
proof (rule H.closed-euler1)
show connected ?H
proof (rule H.connectedI)
interpret sH: pair-fin-digraph mk-symmetric ?H ..
fix u v assume  $u \in \text{verts } ?H \ v \in \text{verts } ?H$ 
with con have  $u \rightarrow^* \text{mk-symmetric } G \ v$  by (auto simp: connected-def)
moreover
have subgraph G ?H by (auto simp: subgraph-def) unfold-locales
ultimately show  $u \rightarrow^* \text{with-proj } (\text{mk-symmetric } ?H) \ v$ 
by (blast intro: sH.reachable-mono subgraph-mk-symmetric)
qed (simp add: verts-add-arc-conv)
next

```

```

fix  $w$  assume  $w \in \text{verts } ?H$ 
then show  $\text{in-degree } ?H w = \text{out-degree } ?H w$ 
  using  $\text{deg deg-in deg-out e-notin}$ 
  apply ( $\text{cases } w = u$ )
  apply ( $\text{case-tac } [!]$   $w = v$ )
  by ( $\text{auto simp: in-degree-add-arc-iff out-degree-add-arc-iff}$ )
qed

then obtain  $w p$  where  $\text{Het: } H.\text{euler-trail } ?e w p$  by  $\text{blast}$ 
then have  $?e \in \text{set } p$  by ( $\text{auto simp: pre-digraph.euler-trail-def}$ )
then obtain  $q r$  where  $p\text{-decomp: } p = q \text{ @ } [?e] \text{ @ } r$ 
  by ( $\text{auto simp: in-set-conv-decomp}$ )
  — We show now that removing the additional arc of  $\text{add-arc } (v, l, u)$  from  $p$ 
  yields an euler trail in  $G$ 

have  $\text{euler-trail } u (r \text{ @ } q) v$ 
proof ( $\text{unfold euler-trail-conv-connected}[OF con], \text{intro conjI}$ )
  from  $\text{Het}$  have  $\text{Ht': } H.\text{trail } ?e v (?e \# r \text{ @ } q) v$ 
  unfolding  $p\text{-decomp } H.\text{euler-trail-def } H.\text{trail-def}$ 
  by ( $\text{auto simp: p-decomp } H.\text{awalk-Cons-iff}$ )
then have  $H.\text{trail } ?e u (r \text{ @ } q) v$   $?e \notin \text{set } (r \text{ @ } q)$ 
  by ( $\text{auto simp: } H.\text{trail-def } H.\text{awalk-Cons-iff}$ )
then show  $t': \text{trail } u (r \text{ @ } q) v$ 
  by ( $\text{auto simp: trail-def } H.\text{trail-def } \text{awalk-def } H.\text{awalk-def}$ )

show  $\text{set } (r \text{ @ } q) = \text{arcs } G$ 
proof —
  have  $\text{arcs } G = \text{arcs } ?H - \{?e\}$  using  $e\text{-notin}$  by  $\text{auto}$ 
  also have  $\text{arcs } ?H = \text{set } p$  using  $\text{Het}$ 
  by ( $\text{auto simp: pre-digraph.euler-trail-def pre-digraph.trail-def}$ )
  finally show  $?thesis$  using  $\langle ?e \notin \text{set } \rightarrow \rangle$  by ( $\text{auto simp: p-decomp}$ )
qed
qed
then show  $?thesis$  by  $\text{blast}$ 
qed

context  $\text{wf-digraph}$  begin

lemma  $\text{trail-app-isoI}$ :
  assumes  $t: \text{trail } u p v$ 
  and  $\text{hom: digraph-isomorphism } \text{hom}$ 
  shows  $\text{pre-digraph.trail } (\text{app-iso } \text{hom } G) (\text{iso-verts } \text{hom } u) (\text{map } (\text{iso-arcs } \text{hom}) p) (\text{iso-verts } \text{hom } v)$ 
proof —
  interpret  $H: \text{wf-digraph app-iso } \text{hom } G$  using  $\text{hom} \dots$ 
  from  $t$   $\text{hom}$  have  $i: \text{inj-on } (\text{iso-arcs } \text{hom}) (\text{set } p)$ 
  unfolding  $\text{trail-def digraph-isomorphism-def}$  by ( $\text{auto dest:subset-inj-on}[\text{where } A=\text{set } p]$ )
  then have  $\text{distinct } (\text{map } (\text{iso-arcs } \text{hom}) p) = \text{distinct } p$ 

```



```

    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 ∈ verts G v ∈ verts G
  assumes  $\bigwedge w. [w \in \text{verts } G; u \neq w; v \neq w] \implies \text{in-degree } G \ w = \text{out-degree } G \ w$ 
  assumes in-degree G u + 1 = out-degree G u
  assumes out-degree G v + 1 = in-degree G v
  shows  $\exists p. \text{euler-trail } u \ p \ v$ 
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 = ( $\lambda a. (\text{tail } G \ a, f \ a, \text{head } G \ a)$ ),
      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]:  $\bigwedge u. u \in \text{verts } G \implies \text{iso-verts } ?\text{iso-g } u = u$ 
  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*  $\langle u \in \cdot \rangle \langle v \in \cdot \rangle$  **by** *simp rule*  
**qed**

**theorem** *open-euler2*:  
**assumes** *et: euler-trail u p v* **and**  $u \neq v$   
**shows** *connected G*  $\wedge$   
 $(\forall w \in \text{verts } G. u \neq w \longrightarrow v \neq w \longrightarrow \text{in-degree } G\ w = \text{out-degree } G\ w) \wedge$   
 $\text{in-degree } G\ u + 1 = \text{out-degree } G\ u \wedge$   
 $\text{out-degree } G\ v + 1 = \text{in-degree } G\ v$

**proof** –  
**from** *et* **have**  $*$ : *trail u p v*  $u \in \text{verts } G\ v \in \text{verts } G$   
**by** (*auto simp: euler-trail-def trail-def awalk-hd-in-verts*)

**from** *et* **have** [*simp*]:  $\bigwedge u. \text{card } (\text{in-arcs } G\ u \cap \text{set } p) = \text{in-degree } G\ u$   
 $\bigwedge u. \text{card } (\text{out-arcs } G\ u \cap \text{set } p) = \text{out-degree } G\ u$   
**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*:  
 $(\exists u\ p\ v. \text{euler-trail } u\ p\ v \wedge u \neq v) \longleftrightarrow$   
 $\text{connected } G \wedge (\exists u\ v. u \in \text{verts } G \wedge v \in \text{verts } G \wedge$   
 $(\forall w \in \text{verts } G. u \neq w \longrightarrow v \neq w \longrightarrow \text{in-degree } G\ w = \text{out-degree } G\ w) \wedge$   
 $\text{in-degree } G\ u + 1 = \text{out-degree } G\ u \wedge$   
 $\text{out-degree } G\ v + 1 = \text{in-degree } G\ v)$  (**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**  $u \in \text{verts } G\ v \in \text{verts } G$   
**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**  $*$ :  
 $\text{connected } G\ u \in \text{verts } G\ v \in \text{verts } G$

```

 $\bigwedge w. \llbracket w \in \text{verts } G; u \neq w; v \neq w \rrbracket \implies \text{in-degree } G \ w = \text{out-degree } G \ w$ 
 $\text{in-degree } G \ u + 1 = \text{out-degree } G \ u$ 
 $\text{out-degree } G \ v + 1 = \text{in-degree } G \ v$ 
  by blast
  then have  $u \neq v$  by auto
  from * show ?L by (metis open-euler1 (u ≠ v))
qed

end

end

```

```

theory Kuratowski
imports
  Arc-Walk
  Digraph-Component
  Subdivision
  HOL-Library.Rewrite
begin

```

## 19 Kuratowski Subgraphs

We consider the underlying undirected graphs. The underlying undirected graph is represented as a symmetric digraph.

### 19.1 Public definitions

**definition** *complete-digraph* ::  $\text{nat} \Rightarrow ('a, 'b) \text{ pre-digraph} \Rightarrow \text{bool}$  ( $K_{\cdot}$ ) **where**  
 $\text{complete-digraph } n \ G \equiv \text{graph } G \wedge \text{card } (\text{verts } G) = n \wedge \text{arcs-ends } G = \{(u, v). (u, v) \in \text{verts } G \times \text{verts } G \wedge u \neq v\}$

**definition** *complete-bipartite-digraph* ::  $\text{nat} \Rightarrow \text{nat} \Rightarrow ('a, 'b) \text{ pre-digraph} \Rightarrow \text{bool}$  ( $K_{\cdot, \cdot}$ ) **where**  
 $\text{complete-bipartite-digraph } m \ n \ G \equiv \text{graph } G \wedge (\exists U \ V. \text{verts } G = U \cup V \wedge U \cap V = \{\})$   
 $\wedge \text{card } U = m \wedge \text{card } V = n \wedge \text{arcs-ends } G = U \times V \cup V \times U$

**definition** *kuratowski-planar* ::  $('a, 'b) \text{ pre-digraph} \Rightarrow \text{bool}$  **where**  
 $\text{kuratowski-planar } G \equiv \neg(\exists H. \text{subgraph } H \ G \wedge (\exists K \ \text{rev-}K \ \text{rev-}H. \text{subdivision } (K, \text{rev-}K) (H, \text{rev-}H) \wedge (K_{3,3} \ K \vee K_5 \ K)))$

**lemma** *complete-digraph-pair-def*:  $K_n$  (*with-proj*  $G$ )  
 $\longleftrightarrow \text{finite } (\text{pverts } G) \wedge \text{card } (\text{pverts } G) = n \wedge \text{parcs } G = \{(u, v). (u, v) \in (\text{pverts } G \times \text{pverts } G) \wedge u \neq v\}$  (**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* ( $pverts\ G \times pverts\ G$ ) *parcs*  $G \subseteq pverts\ G \times pverts\ G$   
**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* ( $pverts\ G$ )  
 $G$

$\wedge (\exists U\ V. pverts\ G = U \cup V \wedge U \cap V = \{\}) \wedge card\ U = m \wedge card\ V = n \wedge$   
*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**

## 19.2 Inner vertices of a walk

**context** *pre-digraph* **begin**

**definition** (**in** *pre-digraph*) *inner-verts* :: 'b *awalk*  $\Rightarrow$  'a list **where**  
*inner-verts* *p*  $\equiv$  *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*  $\neq$  [] 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-append-l*:

**assumes** *u*  $\in$  *set* (*inner-verts* *p*)

**shows** *u*  $\in$  *set* (*inner-verts* (*p* @ *q*))

**using** *assms*

**by** (*induct* *p*) (*auto simp: inner-verts-def*)

**lemma** *in-set-inner-verts-append-r*:

**assumes** *u*  $\in$  *set* (*inner-verts* *q*)

**shows** *u*  $\in$  *set* (*inner-verts* (*p* @ *q*))

```

using assms
by (induct p) (auto simp: inner-verts-def dest: list-set-tl)

end

```

### 19.3 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 *iapaths*: 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  $\bigwedge u v. (x \# x' \# xs') \neq [] @ (u,v) \# (v,u) \# xs'$  using ⟨?L⟩
  unfolding progressing-def by metis
    then have  $\neg(\text{fst } x = \text{snd } x' \wedge \text{snd } x = \text{fst } x')$  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)  $\longleftrightarrow$  progressing xs  $\wedge$  progressing ys
     $\wedge$  (xs  $\neq$  []  $\wedge$  ys  $\neq$  []  $\longrightarrow$  (fst (last xs)  $\neq$  snd (hd ys)  $\vee$  snd (last xs)  $\neq$  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  $\neq$  []  $\implies$  hd (rev-path xs) = prod.swap (last xs) by (induct xs) auto

    have progressing (xs @ y' # ys')  $\longleftrightarrow$  progressing ((xs @ [y']) @ ys')
      by simp
    also have ...  $\longleftrightarrow$  progressing (xs @ [y'])  $\wedge$  progressing ys'  $\wedge$  (ys'  $\neq$  []  $\longrightarrow$  (fst
y'  $\neq$  snd (hd ys')  $\vee$  snd y'  $\neq$  fst (hd ys')))
      by (subst Cons) simp
    also have ...  $\longleftrightarrow$  ?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**

## 19.4 Walks with Restricted Vertices

**definition**  $verts3 :: ('a, 'b) pre-digraph \Rightarrow 'a set$  **where**  
 $verts3 G \equiv \{v \in verts G. 2 < in-degree G v\}$

A path were only the end nodes may be in  $V$

**definition** (in  $pre-digraph$ )  $gen-iapath :: 'a set \Rightarrow 'a \Rightarrow 'b awalk \Rightarrow 'a \Rightarrow bool$   
**where**  
 $gen-iapath V u p v \equiv u \in V \wedge v \in V \wedge apath u p v \wedge set (inner-verts p) \cap V = \{\} \wedge p \neq []$

**abbreviation** (in  $pre-digraph$ ) (input)  $iapath :: 'a \Rightarrow 'b awalk \Rightarrow 'a \Rightarrow bool$  **where**  
 $iapath u p v \equiv gen-iapath (verts3 G) u p v$

**definition**  $gen-contr-graph :: ('a, 'b) pre-digraph \Rightarrow 'a set \Rightarrow 'a pair-pre-digraph$   
**where**  
 $gen-contr-graph G V \equiv ($   
 $pverts = V,$   
 $parcs = \{(u,v). \exists p. pre-digraph.gen-iapath G V u p v\}$   
 $)$

**abbreviation** (input)  $contr-graph :: 'a pair-pre-digraph \Rightarrow 'a pair-pre-digraph$   
**where**  
 $contr-graph G \equiv gen-contr-graph G (verts3 G)$

## 19.5 Properties of subdivisions

**lemma** (in  $pair-sym-digraph$ )  $verts3-subdivide$ :

**assumes**  $e \in parcs G w \notin pverts G$   
**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**  $\langle w \notin pverts G \rangle$

**have**  $w$ -arcs:  $(u,w) \notin parcs G (v,w) \notin parcs G (w,u) \notin parcs G (w,v) \notin parcs G$   
**by** (auto dest: wellformed)

**have**  $G$ -arcs:  $(u,v) \in parcs G (v,u) \in parcs G$

**using**  $\langle e \in parcs G \rangle$  **by** (auto simp: arcs-symmetric)

**have**  $\{v \in pverts G. 2 < in-degree G v\} = \{v \in pverts G. 2 < in-degree ?sG v\}$

**proof** –

**{ fix**  $x$  **assume**  $x \in pverts G$

**define**  $card-eq$  **where**  $card-eq x \longleftrightarrow in-degree ?sG x = in-degree G x$  **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 simp: )
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

```

$set\text{-}ev\text{-}sd: set (S.awalk\text{-}verts\ u\ (sd\text{-}path\ e\ w\ p)) \subseteq set (awalk\text{-}verts\ u\ p) \cup \{w\}$   
**using** *elems* **by** (rule *apath-sd-path set-awalk-verts-sd-path*)  
**have**  $w \notin \{u,v\}$  **using** *elems* (*apath u p v*)  
**by** (*auto simp: apath-def awalk-hd-in-verts awalk-last-in-verts*)

**have**  $set (S.inner\text{-}verts\ (sd\text{-}path\ e\ w\ p)) = set (S.awalk\text{-}verts\ u\ (sd\text{-}path\ e\ w\ p)) - \{u,v\}$   
**using** *apath-sd* **by** (rule *S.set-inner-verts*)  
**also have**  $\dots \subseteq set (awalk\text{-}verts\ u\ p) \cup \{w\} - \{u,v\}$   
**using** *set-ev-sd* **by** *auto*  
**also have**  $\dots = set (inner\text{-}verts\ p) \cup \{w\}$   
**using** *set-inner-verts[OF apath u p v]* ( $w \notin \{u,v\}$ ) **by** *blast*  
**finally have**  $set (S.inner\text{-}verts\ (sd\text{-}path\ e\ w\ p)) \cap V \subseteq (set (inner\text{-}verts\ p) \cup \{w\}) \cap V$   
**using** *V* **by** *blast*  
**also have**  $\dots \subseteq \{\}$   
**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 ∈ parcs G w ∉ pverts G*  
**assumes** *V: V ⊆ 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 ∈ pverts G v ∈ 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**

## 19.6 Pair Graphs

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

**lemma** *gen-iapath-rev-path:*  
 $gen\text{-}iapath\ V\ v\ (rev\text{-}path\ p)\ u = gen\text{-}iapath\ V\ u\ p\ v$  (**is**  $?L = ?R$ )

```

proof –
  { fix  $u\ p\ v$  assume  $\text{gen-iapath } V\ u\ p\ v$ 
    then have  $\text{butlast } (\text{tl } (\text{awalk-verts } v\ (\text{rev-path } p))) = \text{rev } (\text{butlast } (\text{tl } (\text{awalk-verts } u\ p)))$ 
    by ( $\text{auto simp: tl-rev butlast-rev butlast-tl awalk-verts-rev-path gen-iapath-def apath-def}$ )
    with  $\langle \text{gen-iapath } V\ u\ p\ v \rangle$  have  $\text{gen-iapath } V\ v\ (\text{rev-path } p)\ u$ 
    by ( $\text{auto simp: gen-iapath-def apath-def inner-verts-conv[symmetric] awalk-verts-rev-path}$ )
  }
  note  $RL = \text{this}$ 
  show  $?thesis$  by ( $\text{auto dest: RL intro: RL}$ )
qed

```

```

lemma  $\text{inner-verts-rev-path}$ :
  assumes  $\text{awalk } u\ p\ v$ 
  shows  $\text{inner-verts } (\text{rev-path } p) = \text{rev } (\text{inner-verts } p)$ 
by ( $\text{metis assms butlast-rev butlast-tl awalk-verts-rev-path inner-verts-conv tl-rev}$ )

end

```

**context**  $\text{pair-pseudo-graph}$  **begin**

```

lemma  $\text{apath-imp-progressing}$ :
  assumes  $\text{apath } u\ p\ v$  shows  $\text{progressing } p$ 
proof ( $\text{rule ccontr}$ )
  assume  $\neg ?thesis$ 
  then obtain  $xs\ x\ y\ ys$  where  $*$ :  $p = xs @ (x,y) \# (y,x) \# ys$ 
  unfolding  $\text{progressing-def}$  by  $\text{auto}$ 
  then have  $\neg \text{apath } u\ p\ v$ 
  by ( $\text{simp add: apath-append-iff apath-simps hd-in-awalk-verts}$ )
  then show  $\text{False}$  using  $\text{assms}$  by  $\text{auto}$ 
qed

```

```

lemma  $\text{awalk-Cons-deg2-unique}$ :
  assumes  $\text{awalk } u\ p\ v\ p \neq []$ 
  assumes  $\text{in-degree } G\ u \leq 2$ 
  assumes  $\text{awalk } u1\ (e1 \# p)\ v\ \text{awalk } u2\ (e2 \# p)\ v$ 
  assumes  $\text{progressing } (e1 \# p)\ \text{progressing } (e2 \# p)$ 
  shows  $e1 = e2$ 
proof ( $\text{cases } p$ )
  case ( $\text{Cons } e\ es$ )
  show  $?thesis$ 
proof ( $\text{rule ccontr}$ )
  assume  $e1 \neq e2$ 
  define  $x$  where  $x = \text{snd } e$ 
  then have  $e\text{-unf}: e = (u,x)$  using  $\langle \text{awalk } u\ p\ v \rangle$   $\text{Cons}$  by ( $\text{auto simp: awalk-simps}$ )
  then have  $e1\text{-unf}: e1 = (u1, u)\ e2 = (u2, u)$ 
  using  $\text{Cons assms}$  by ( $\text{auto simp: apath-simps prod-eqI}$ )
  with  $\text{Cons assms } \langle e = (u,x) \rangle \langle e1 \neq e2 \rangle$  have  $u1 \neq u2\ x \neq u1\ x \neq u2$ 

```

by (auto simp: progressing-Cons-Cons)  
 moreover have  $\{(u1, u), (u2, u), (x, u)\} \subseteq \text{parcs } G$   
 using e-unf ei-unf Cons assms by (auto simp: awalk-simps intro: arcs-symmetric)  
 then have finite (in-arcs G u)  
 and  $\{(u1, u), (u2, u), (x, u)\} \subseteq \text{in-arcs } G \text{ u}$  by auto  
 then have  $\text{card } (\{(u1, u), (u2, u), (x, u)\}) \leq \text{in-degree } G \text{ u}$   
 unfolding in-degree-def by (rule card-mono)  
 ultimately show False using  $\langle \text{in-degree } G \text{ u} \leq 2 \rangle$  by auto  
 qed  
 qed (simp add:  $\langle p \neq [] \rangle$ )

lemma same-awalk-by-same-end:

assumes  $V: \text{verts3 } G \subseteq V \ V \subseteq \text{pverts } G$   
 and walk:  $\text{awalk } u \ p \ v \ \text{awalk } u \ q \ w \ \text{hd } p = \text{hd } q \ p \neq [] \ q \neq []$   
 and progress:  $\text{progressing } p \ \text{progressing } q$   
 and tail:  $v \in V \ w \in V$   
 and inner-verts:  $\text{set } (\text{inner-verts } p) \cap V = \{\}$   
 $\text{set } (\text{inner-verts } q) \cap V = \{\}$   
 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  $\langle \text{hd } (a \# -) = \text{hd } - \rangle$  have  $a = b$  by simp

{ fix a as v b bs w  
 assume A:  $\text{awalk } u \ (a \# as) \ v \ \text{awalk } u \ (b \# bs) \ w$   
 $\text{set } (\text{inner-verts } (b \# bs)) \cap V = \{\} \ v \in V \ a = b \ as = []$   
 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.prem  $\langle a = b \rangle$  tail by (metis Nil-imp-Nil)  
 then show ?thesis using True  $\langle a = b \rangle$  by simp

next

case False

then have  $bs \neq []$  using Cons-Cons.prem  $\langle a = b \rangle$  tail by (metis Nil-imp-Nil)

obtain  $a' \ as'$  where  $as = a' \# as'$  using  $\langle as \neq [] \rangle$  by (cases as) simp

obtain  $b' \ bs'$  where  $bs = b' \# bs'$  using  $\langle bs \neq [] \rangle$  by (cases bs) simp

let  $?arcs = \{(\text{fst } a, \text{snd } a), (\text{snd } a', \text{snd } a), (\text{snd } b', \text{snd } a)\}$

have  $\text{card } \{\text{fst } a, \text{snd } a', \text{snd } b'\} = \text{card } (\text{fst } ' ?arcs)$  by auto

also have  $\dots = \text{card } ?arcs$  by (rule card-image) (cases a, auto)

also have  $\dots \leq \text{in-degree } G \ (\text{snd } a)$

proof  $-$

```

have ?arcs  $\subseteq$  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 ...  $\leq 2$ 
proof -
  have snd  $a \notin V$  snd  $a \in pverts$   $G$ 
    using Cons-Cons.premis ⟨ $as \neq []$ ⟩ by (auto simp: inner-verts-Cons)
  then show ?thesis using  $V$  by (auto simp: verts3-def)
qed
finally have fst  $a =$  snd  $a' \vee$  fst  $a =$  snd  $b' \vee$  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 ⟨ $as \neq []$ ⟩ ⟨ $bs \neq []$ ⟩ Cons-Cons.premis
  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  $G \subseteq V$   $V \subseteq pverts$   $G$ 
assumes walk: awalk  $u$   $p$   $v$  awalk  $w$   $q$   $x$ 
assumes progress: progressing  $p$  progressing  $q$ 
assumes iv-not-in- $V$ : set (inner-verts  $p$ )  $\cap V = \{\}$  set (inner-verts  $q$ )  $\cap V = \{\}$ 
assumes ends-in- $V$ :  $\{u, v, w, x\} \subseteq V$ 
assumes arcs:  $e \in$  set  $p$   $e \in$  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  $q1'$   $p1' \neq []$   $q1' \neq []$  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'))  $\cap$  V = {} set (inner-verts (rev-path q1'))
 $\cap$  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  $\in$  V w  $\in$  V set (inner-verts p1')  $\cap$  V = {} set (inner-verts q1')
 $\cap$  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'  $\neq$  [] q2'  $\neq$  [] 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  $\in$  V x  $\in$  V set (inner-verts p2')  $\cap$  V = {} set (inner-verts q2')  $\cap$  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  $\subseteq$  V V  $\subseteq$  pverts G
  assumes path: gen-iapath V u p v gen-iapath V w q x
  assumes arcs: e  $\in$  set p e  $\in$  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)  $\cap$  V = {} set (inner-verts q)  $\cap$  V = {} {u,v,w,x}
 $\subseteq$  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

## 19.7 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  $\Rightarrow$  bool **where**

*is-slim*  $V \equiv$   
( $\forall v \in pverts\ G. v \in V \vee$   
  *in-degree*  $G\ v \leq 2 \wedge (\exists x\ p\ y. gen-iapath\ V\ x\ p\ y \wedge v \in set\ (awalk-verts\ x\ p))$ )  
 $\wedge$   
( $\forall e \in parcs\ G. fst\ e \neq snd\ e \wedge (\exists x\ p\ y. gen-iapath\ V\ x\ p\ y \wedge e \in set\ p)$ )  $\wedge$   
( $\forall u\ v\ p\ q. (gen-iapath\ V\ u\ p\ v \wedge gen-iapath\ V\ u\ q\ v) \longrightarrow p = q$ )  $\wedge$   
   $V \subseteq pverts\ G$ )

**definition** *direct-arc* :: 'a  $\times$  'a  $\Rightarrow$  'a  $\times$  'a **where**

*direct-arc*  $uv \equiv SOME\ e. \{fst\ uv, snd\ uv\} = \{fst\ e, snd\ e\}$

**definition** *choose-iapath* :: 'a  $\Rightarrow$  'a  $\Rightarrow$  ('a  $\times$  'a) *awalk* **where**

*choose-iapath*  $u\ v \equiv (let$   
  *chosen-path* = ( $\lambda u\ v. SOME\ p. iapath\ u\ p\ v$ )  
  *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) *awalk*  $\times$  'a) set **where**

*slim-paths*  $\equiv (\lambda e. (fst\ e, choose-iapath\ (fst\ e)\ (snd\ e), snd\ e))\ 'parcs\ (contr-graph\ G)$

**definition** *slim-verts* :: 'a set **where**

*slim-verts*  $\equiv verts3\ G \cup (\bigcup (u, p, -) \in slim-paths. set\ (awalk-verts\ u\ p))$

**definition** *slim-arcs* :: 'a rel **where**

*slim-arcs*  $\equiv \bigcup (-, p, -) \in slim-paths. set\ p$

Computes a slim subgraph for an arbitrary *pair-digraph*

**definition** *slim* :: 'a *pair-pre-digraph* **where**

*slim*  $\equiv (\mid pverts = slim-verts, parcs = slim-arcs \mid)$

end

**lemma** (**in** *wf-digraph*) *iapath-dist-ends*:  $\bigwedge u\ p\ v. iapath\ u\ p\ v \Longrightarrow u \neq v$

**unfolding** *pre-digraph.gen-iapath-def* **by** (*metis* *apath-ends*)

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

```

lemma choose-iapath:
  assumes  $\exists p. \text{iapath } u \ p \ v$ 
  shows  $\text{iapath } u \ (\text{choose-iapath } u \ v) \ v$ 
proof (cases direct-arc (u,v) = (u,v))
  define chosen where  $\text{chosen } u \ v = (\text{SOME } p. \text{iapath } u \ p \ v)$  for  $u \ v$ 
  { case True
    have  $\text{iapath } u \ (\text{chosen } u \ v) \ v$ 
      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  $\text{iapath } u \ p \ v$  by auto
    then have  $\text{iapath } v \ (\text{rev-path } p) \ u$ 
      by (simp add: gen-iapath-rev-path)
    then have  $\text{iapath } v \ (\text{chosen } v \ u) \ u$ 
      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  $(u,p,v) \in \text{slim-paths}$  obtains  $\text{iapath } u \ p \ v \ u \neq v$ 
  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  $\subseteq$  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  $x \in \text{verts3 } G$  shows  $x \in \text{pverts slim}$ 
using assms by (auto simp: slim-simps slim-verts-def)

lemma arcs-slim-in-G: parcs slim  $\subseteq$  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  $(u,p,v) \in \text{slim-paths}$ 
  shows  $\text{pre-digraph.gen-iapath slim (verts3 } G) \ u \ p \ v \wedge p \neq []$ 
proof –
  from assms have arcs:  $\bigwedge e. e \in \text{set } p \implies e \in \text{parcs slim}$ 
    by (auto simp: slim-simps slim-arcs-def)
  moreover
  from assms have  $\text{gen-iapath (verts3 } G) \ u \ p \ v$  and  $p \neq []$ 

```



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)  $\vee$  *direct-arc* (u,v) = (v,u)  
**proof** –  
 define f :: 'a set  $\Rightarrow$  'a  $\times$  'a  
 where f X = (SOME e. X = {fst e, snd e}) for X

have  $\exists p::'a \times 'a. \{u,v\} = \{fst p, snd p\}$  **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)  $\vee$  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  $\neq$  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  $\Longrightarrow$  a  $\in$  set p  $\Longrightarrow$  fst a  $\neq$  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  $\in$  *parcs slim*  
 then obtain u p v **where** (u,p,v)  $\in$  *slim-paths* e  $\in$  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  $\langle e \in$  set p  $\rangle$  have fst e  $\in$  set (awalk-verts u p) snd e  $\in$  set (awalk-verts u p)  
**by** (auto simp: *set-awalk-verts gen-iapath-def apath-def*)  
 moreover  
 from  $\langle - \in$  *slim-paths*  $\rangle$  have set (awalk-verts u p)  $\subseteq$  pverts slim  
**by** (auto simp: *slim-simps slim-verts-def*)  
 ultimately  
 show fst e  $\in$  pverts slim snd e  $\in$  pverts slim **by** auto

show fst e  $\neq$  snd e  
 using  $\langle$  *iapath* u p v  $\rangle$   $\langle e \in$  set p  $\rangle$  **by** (auto dest: *no-loops-in-iapath*)

**next**  
 { **fix**  $e$  **assume**  $e \in \text{parcs } \textit{slim}$   
   **then obtain**  $u \ p \ v$  **where**  $(u,p,v) \in \textit{slim-paths}$  **and**  $e \in \textit{set } p$   
     **by** (*auto simp: slim-simps slim-arcs-def*)  
   **moreover**  
   **then have**  $\textit{iapath } u \ p \ v$  **and**  $p \neq []$  **and**  $u \neq v$  **by** (*auto elim: slim-paths-in-G-E*)  
   **then have**  $\textit{iapath } v \ (\textit{rev-path } p) \ u$  **and**  $\textit{rev-path } p \neq []$  **and**  $v \neq u$   
     **by** (*auto simp: gen-iapath-rev-path*)  
   **then have**  $(v,u) \in \textit{parcs } (\textit{contr-graph } G)$   
     **by** (*auto simp: gen-contr-graph-def*)  
   **moreover**  
   **from**  $\langle \textit{iapath } u \ p \ v \rangle$  **have**  $u \neq v$   
     **by** (*auto simp: gen-iapath-def dest: apath-nonempty-ends*)  
   **ultimately**  
   **have**  $(v, \textit{rev-path } p, u) \in \textit{slim-paths}$   
     **by** (*auto simp: slim-paths-def rev-path-choose-iapath intro: rev-image-eqI*)  
   **moreover**  
   **from**  $\langle e \in \textit{set } p \rangle$  **have**  $(\textit{snd } e, \textit{fst } e) \in \textit{set } (\textit{rev-path } p)$   
     **by** (*induct p*) *auto*  
   **ultimately have**  $(\textit{snd } e, \textit{fst } e) \in \textit{parcs } \textit{slim}$   
     **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**  $H: \textit{pair-bidirected-digraph } \textit{slim}$   
     **by** (*rule pair-bidirected-digraph-slim*) *intro-locales*  
  
   **show**  $\textit{verts } \textit{slim} \subseteq \textit{verts } G$   $\textit{arcs } \textit{slim} \subseteq \textit{arcs } G$   
     **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  $\exists p. \text{pre-digraph.gen-iapath slim (verts3 G) u p v}$  (is  $\exists p. ?P p$ )
proof
  from assms have choose-arcs:  $\bigwedge e. e \in \text{set (choose-iapath u v)} \implies e \in \text{parcs slim}$ 
  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  $v \in \text{verts3 slim}$  shows  $v \in \text{verts3 G}$ 
proof –
  from assms have  $2 < \text{in-degree slim } v$  by (auto simp: verts3-def)
  also have  $\dots \leq \text{in-degree } G v$  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  $v \in \text{pverts slim}$   $v \notin \text{verts3 } G$ 
    then have  $\text{in-degree } G v \leq 2$ 
    using verts-slim-in-G by (auto simp: verts3-def)
    then show  $\text{in-degree slim } v \leq 2$ 
    using subgraph-in-degree[OF subgraph-slim, of v] by fastforce
  }
next

```

```

fix  $w$  assume  $w \in pverts\ slim\ w \notin verts3\ G$ 
then obtain  $u\ p\ v$  where  $upv: (u, p, v) \in slim\ paths\ w \in set\ (awalk\ vert\ u\ p)$ 
  by  $(auto\ simp: slim\ simps\ slim\ vert\ def)$ 
moreover
then have  $S.gen\ iapath\ (verts3\ G)\ u\ p\ v$ 
  using  $slim\ paths\ in\ slim\ G$  by  $auto$ 
ultimately
show  $\exists x\ q\ y. S.gen\ iapath\ (verts3\ G)\ x\ q\ y$ 
   $\wedge w \in set\ (awalk\ vert\ x\ q)$ 
  by  $auto$ 
next
fix  $u\ v$  assume  $(u, v) \in parcs\ slim$ 
then obtain  $x\ p\ y$  where  $(x, p, y) \in slim\ paths\ (u, v) \in set\ p$ 
  by  $(auto\ simp: slim\ simps\ slim\ arcs\ def)$ 
then have  $S.gen\ iapath\ (verts3\ G)\ x\ p\ y \wedge (u, v) \in set\ p$ 
  using  $slim\ paths\ in\ slim\ G$  by  $auto$ 
then show  $\exists x\ p\ y. S.gen\ iapath\ (verts3\ G)\ x\ p\ y \wedge (u, v) \in set\ p$ 
  by  $blast$ 
next
fix  $u\ v$  assume  $(u, v) \in parcs\ slim\ fst\ (u, v) = snd\ (u, v)$ 
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  $V: verts3\ slim \subseteq verts3\ G\ verts3\ G \subseteq pverts\ slim$ 
  by  $(auto\ simp: verts3\ slim\ in\ verts3)$ 

have  $p = [] \vee q = [] \implies p = q$  using  $paths$ 
  by  $(auto\ simp: S.gen\ iapath\ def\ dest: S.apath\ ends)$ 
moreover
{ assume  $p \neq []\ q \neq []$ 
  { fix  $u\ p\ v$  assume  $p \neq []$  and  $path: S.gen\ iapath\ (verts3\ G)\ u\ p\ v$ 
    then obtain  $e$  where  $e \in set\ p$  by  $(metis\ last\ in\ set)$ 
    then have  $e \in parcs\ slim$  using  $path$  by  $(auto\ simp: S.gen\ iapath\ def\ S.apath\ def)$ 
    then obtain  $x\ r\ y$  where  $(x, r, y) \in slim\ paths\ e \in set\ r$ 
      by  $(auto\ simp: slim\ simps\ slim\ arcs\ def)$ 
    then have  $S.gen\ iapath\ (verts3\ G)\ x\ r\ y$  by  $(metis\ slim\ paths\ in\ slim\ G)$ 
    with  $\langle e \in set\ r \rangle \langle e \in set\ p \rangle\ path$  have  $p = r$ 
      by  $(auto\ intro: S.same\ gen\ iapath\ by\ common\ arc[OF\ V])$ 
    then have  $x = u\ y = v$  using  $path\ \langle S.gen\ iapath\ (verts3\ G)\ x\ r\ y \rangle \langle p = r \rangle$ 
     $\langle p \neq [] \rangle$ 
    by  $(auto\ simp: S.gen\ iapath\ def\ S.apath\ def\ dest: S.awalk\ ends)$ 
    then have  $(u, p, v) \in slim\ paths$  using  $\langle p = r \rangle \langle (x, r, y) \in slim\ paths \rangle$  by  $simp$ 
  }
}

note  $obt = this$ 
from  $\langle p \neq [] \rangle \langle q \neq [] \rangle\ paths$  have  $(u, p, v) \in slim\ paths\ (u, q, v) \in slim\ paths$ 

```

```

    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:  $(u,v) \in \text{parcs } G$ 
proof -
  have set (inner-verts  $p$ ) = {}
  proof -
    have *:  $\bigwedge A B :: 'a \text{ set. } [A \subseteq B; A \cap B = \{\}] \implies A = \{\}$  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 ...  $\subseteq$  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  $(u,v) \in \text{parcs } G$ 
    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  $(u,v) \in \text{parcs } ?gcg$ 
    then obtain  $p$  where gen-iapath  $V$   $u$   $p$   $v$ 
      by (auto simp: gen-contr-graph-def)
    then show  $(u,v) \in \text{parcs } G$ 
      using gen-iapath-triv-arc (pverts  $G = V$ ) by auto
  next

```

```

fix  $u\ v$  assume  $(u,v) \in \text{parcs } G$ 
with  $\text{assms}$  obtain  $x\ p\ y$  where  $\text{path}: \text{gen-iapath } V\ x\ p\ y\ (u,v) \in \text{set } p\ u \neq v$ 
  by  $(\text{auto simp: is-slim-def})$ 
with  $\langle p \text{verts } G = V \rangle$  have  $p = [(x,y)]$  by  $(\text{intro gen-iapath-triv-path})\ \text{auto}$ 
then show  $(u,v) \in \text{parcs } ?gcg$ 
  using  $\text{path}$  by  $(\text{auto simp: gen-contr-graph-def})$ 
qed
ultimately
show  $?gcg = G$  by  $\text{auto}$ 
qed

```

```

lemma  $\text{is-slim-no-loops}$ :
  assumes  $\text{is-slim } V\ a \in \text{arcs } G$  shows  $\text{fst } a \neq \text{snd } a$ 
  using  $\text{assms}$  by  $(\text{auto simp: is-slim-def})$ 

```

**end**

## 19.8 Contraction Preserves Kuratowski-Subgraph-Property

```

lemma  $(\text{in pair-pseudo-graph})\ \text{in-degree-contr}$ :
  assumes  $v \in V$  and  $V: \text{verts3 } G \subseteq V\ V \subseteq \text{verts } G$ 
  shows  $\text{in-degree } (\text{gen-contr-graph } G\ V)\ v \leq \text{in-degree } G\ v$ 
proof –
  have  $\text{fin}: \text{finite } \{(u, p). \text{gen-iapath } V\ u\ p\ v\}$ 
  proof –
    have  $\{(u, p). \text{gen-iapath } V\ u\ p\ v\} \subseteq (\lambda(u,p,-). (u,p))\ \text{' } \{(u,p,v). \text{apath } u\ p\ v\}$ 
    by  $(\text{force simp: gen-iapath-def})$ 
    with  $\text{apaths-finite-triple}$  show  $?thesis$  by  $(\text{rule finite-surj})$ 
  qed

  have  $\text{io-snd}: \text{inj-on } \text{snd } \{(u,p). \text{gen-iapath } V\ u\ p\ v\}$ 
  by  $(\text{rule inj-onI})\ (\text{auto simp: gen-iapath-def apath-def dest: awalk-ends})$ 

  have  $\text{io-last}: \text{inj-on } \text{last } \{p. \exists u. \text{gen-iapath } V\ u\ p\ v\}$ 
proof  $(\text{rule inj-onI}, \text{ safe})$ 
  fix  $u1\ u2\ p1\ p2$ 
  assume  $A: \text{last } p1 = \text{last } p2$  and  $B: \text{gen-iapath } V\ u1\ p1\ v\ \text{gen-iapath } V\ u2\ p2\ v$ 
  from  $B$  have  $\text{last } p1 \in \text{set } p1\ \text{last } p2 \in \text{set } p2$  by  $(\text{auto simp: gen-iapath-def})$ 
  with  $A$  have  $\text{last } p1 \in \text{set } p1\ \text{last } p1 \in \text{set } p2$  by  $\text{simp-all}$ 
  with  $V[\text{simplified}]\ B$  show  $p1 = p2$  by  $(\text{rule same-gen-iapath-by-common-arc})$ 
qed

  have  $\text{in-degree } (\text{gen-contr-graph } G\ V)\ v = \text{card } ((\lambda(u,-). (u,v))\ \text{' } \{(u,p). \text{gen-iapath } V\ u\ p\ v\})$ 
proof –
  have  $\text{in-arcs } (\text{gen-contr-graph } G\ V)\ v = (\lambda(u,-). (u,v))\ \text{' } \{(u,p). \text{gen-iapath } V\ u\ p\ v\}$ 
  by  $(\text{auto simp: gen-contr-graph-def})$ 
  then show  $?thesis$  unfolding  $\text{in-degree-def}$  by  $\text{simp}$ 

```

```

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 K3,3 (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  $\text{verts3 } G = \text{verts } G$ 
proof –
  interpret  $\text{pg}G$ : pair-graph  $G$  using assms by (rule pair-graphI-complete)
  { fix  $v$  assume  $v \in \text{pverts } G$ 
    have  $2 < (4 :: \text{nat})$  by simp
    also have  $4 = \text{card } (\text{pverts } G - \{v\})$ 
      using assms  $\langle v \in \text{pverts } G \rangle$  unfolding complete-digraph-def by auto
    also have  $\text{pverts } G - \{v\} = \{u \in \text{pverts } G. u \neq v\}$ 
      by auto
    also have  $\text{card } \dots = \text{card } (\{u \in \text{pverts } G. u \neq v\} \times \{v\})$  (is  $=$   $\text{card } ?A$ )
      by auto
    also have  $?A = \text{in-arcs } G v$ 
      using assms  $\langle v \in \text{pverts } G \rangle$  unfolding complete-digraph-def by safe auto
    also have  $\text{card } \dots = \text{in-degree } G v$ 
      unfolding in-degree-def ..
    finally have  $2 < \text{in-degree } G v$  . }
  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  $\text{pg}G$ : 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  $\text{pg}G$ : 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

```

## 19.9 Final proof

```

context pair-sym-digraph begin

```

```

lemma gcg-subdivide-eq:
  assumes mem:  $e \in \text{parcs } G$   $w \notin \text{pverts } G$ 
  assumes  $V$ :  $V \subseteq \text{pverts } G$ 
  shows gen-contr-graph (subdivide  $G e w$ )  $V = \text{gen-contr-graph } G V$ 

```

```

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  $\exists p. \text{gen-iapath } V u p v ..$ 
  } 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  $\exists p. \text{sdG.gen-iapath } V u p v ..$ 
  } 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]  $\notin \{[(fst\ e, w), (w, snd\ e)], [(snd\ e, w), (w, fst\ e)]\}$ 
  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  $\exists p1\ p2. p = p1 @ (fst\ e, w) \# (w, snd\ e) \# p2$ 
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
  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  $\langle \text{fst } e \neq z \rangle$  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:  $e \in \text{parcs } G$   $w \notin \text{pverts } G$  and mem2:  $w \notin V$ 
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  $u \in \text{pverts } G$   $v \in \text{pverts } G$  by (auto simp: wellformed')
  with mem1 have  $u \neq w$   $v \neq w$  by auto

  let ?w-parcs =  $\{(u,w), (v,w), (w,u), (w,v)\}$ 
  have sdg-new-parcs: ?w-parcs  $\subseteq$  parcs ?sdG
    using  $\langle e = (u,v) \rangle$  by auto
  have sdg-no-parcs:  $(u,v) \notin \text{parcs } ?sdG$   $(v,u) \notin \text{parcs } ?sdG$ 
    using  $\langle e = (u,v) \rangle$   $\langle u \neq w \rangle$   $\langle v \neq w \rangle$  by auto

  { fix  $z$  assume  $A$ :  $z \in \text{pverts } G$ 
    have in-degree ?sdG  $z = \text{in-degree } G$   $z$ 
    proof –
      { assume  $z \neq u$   $z \neq v$ 
        then have in-arcs ?sdG  $z = \text{in-arcs } G$   $z$ 
          using  $\langle e = (u,v) \rangle$  mem1  $A$  by auto
        then have in-degree ?sdG  $z = \text{in-degree } G$   $z$  by (simp add: in-degree-def) }
      moreover
      { assume  $z = u$ 
        then have in-arcs  $G$   $z = \text{in-arcs } ?sdG$   $z \cup \{(v,u)\} - \{(w,u)\}$ 
          using  $\langle e = (u,v) \rangle$  mem1 by (auto simp: intro: arcs-symmetric wellformed')
        moreover
        have card (in-arcs ?sdG  $z \cup \{(v,u)\} - \{(w,u)\}$ ) = card (in-arcs ?sdG  $z$ )
          using sdg-new-parcs sdg-no-parcs  $\langle z = u \rangle$  by (cases finite (in-arcs ?sdG
 $z)$ ) (auto simp: in-arcs-def)
        ultimately have in-degree ?sdG  $z = \text{in-degree } G$   $z$  by (simp add: in-degree-def)
      }
    }
    moreover
    { assume  $z = v$ 
      then have in-arcs  $G$   $z = \text{in-arcs } ?sdG$   $z \cup \{(u,v)\} - \{(w,v)\}$ 
        using  $\langle e = (u,v) \rangle$  mem1  $A$  by (auto simp: wellformed')
      moreover
      have card (in-arcs ?sdG  $z \cup \{(u,v)\} - \{(w,v)\}$ ) = card (in-arcs ?sdG  $z$ )
        using sdg-new-parcs sdg-no-parcs  $\langle z = v \rangle$  by (cases finite (in-arcs ?sdG
 $z)$ ) (auto simp: in-arcs-def)
      ultimately have in-degree ?sdG  $z = \text{in-degree } G$   $z$  by (simp add: in-degree-def)
    }
  }
  ultimately show ?thesis by metis
  qed }
note in-degree-same = this

have V-G:  $V \subseteq \text{pverts } G$   $\text{verts3 } G \subseteq V$ 
proof –

```

**have**  $V \subseteq pverts \text{ ?sdG } pverts \text{ ?sdG} = pverts \ G \cup \{w\} \text{ verts3 } \text{ ?sdG} \subseteq V \text{ verts3 } G \subseteq \text{verts3 } \text{ ?sdG}$   
**using**  $\langle sdG.is-slim \ V \rangle \langle e = (u,v) \rangle \text{ in-degree-same mem1}$   
**unfolding**  $sdG.is-slim-def \text{ verts3-def}$   
**by**  $(fast, simp, fastforce, force)$   
**then show**  $V \subseteq pverts \ G \text{ verts3 } G \subseteq V$  **using**  $\langle w \notin V \rangle$  **by** *auto*  
**qed**

**have**  $pverts: \forall v \in pverts \ G. v \in V \vee \text{in-degree } G \ v \leq 2 \wedge (\exists x \ p \ y. \text{gen-iapath } V \ x \ p \ y \wedge v \in \text{set } (awalk-verts \ x \ p))$   
**proof** –  
**{ fix**  $z$  **assume**  $A: z \in pverts \ G \ z \notin V$   
**have**  $z \in pverts \ \text{?sdG}$  **using**  $\langle e = (u,v) \rangle \ A \ \text{mem1}$  **by** *auto*  
**then have**  $\text{in-degree } \text{ ?sdG } \ z \leq 2$   
**using**  $\langle sdG.is-slim \ V \rangle \ A$  **by**  $(auto \ simp: \ sdG.is-slim-def)$   
**with**  $\text{in-degree-same}[OF \ \langle z \in pverts \ G \rangle]$  **have**  $\text{idg: in-degree } G \ z \leq 2$  **by** *auto*

**from**  $A$  **have**  $z \in pverts \ \text{?sdG} \ z \notin V$  **using**  $\langle e = (u,v) \rangle \ \text{mem1}$  **by** *auto*  
**then obtain**  $x' \ q \ y'$  **where**  $sdG.gen-iapath \ V \ x' \ q \ y' \ z \in \text{set } (sdG.awalk-verts \ x' \ q)$   
**using**  $\langle sdG.is-slim \ V \rangle$  **unfolding**  $sdG.is-slim-def$  **by** *metis*  
**then have**  $\text{gen-iapath } V \ x' \ (\text{co-path } e \ w \ q) \ y' \ z \in \text{set } (awalk-verts \ x' \ (\text{co-path } e \ w \ q))$   
**using**  $A \ \text{mem1} \ V-G$  **by**  $(auto \ simp: \ \text{set-awalk-verts-co-path}' \ \text{intro: } \ \text{gen-iapath-co-path})$   
**with**  $\text{idg}$  **have**  $\text{in-degree } G \ z \leq 2 \wedge (\exists x \ p \ y. \text{gen-iapath } V \ x \ p \ y \wedge z \in \text{set } (awalk-verts \ x \ p))$   
**by** *metis* **}**  
**then show**  $\text{?thesis}$  **by** *auto*  
**qed**

**have**  $\text{parcs: } \forall e \in \text{parcs } \ G. \ \text{fst } e \neq \text{snd } e \wedge (\exists x \ p \ y. \text{gen-iapath } V \ x \ p \ y \wedge e \in \text{set } p)$   
**proof**  $(intro \ \text{ballI} \ \text{conjI})$   
**fix**  $e'$  **assume**  $e' \in \text{parcs } \ G$

**show**  $(\exists x \ p \ y. \text{gen-iapath } V \ x \ p \ y \wedge e' \in \text{set } p)$   
**proof**  $(cases \ e' \in \text{parcs } \ \text{?sdG})$   
**case** *True*  
**then obtain**  $x \ p \ y$  **where**  $sdG.gen-iapath \ V \ x \ p \ y \ e' \in \text{set } p$   
**using**  $\langle sdG.is-slim \ V \rangle$  **by**  $(auto \ simp: \ sdG.is-slim-def)$   
**with**  $\langle e \in \text{parcs } \ G \rangle \langle w \notin pverts \ G \rangle \ V-G$  **have**  $\text{gen-iapath } V \ x \ (\text{co-path } e \ w \ p) \ y$   
**by**  $(auto \ \text{intro: } \ \text{gen-iapath-co-path})$

**from**  $\langle e' \in \text{parcs } \ G \rangle$  **have**  $e' \notin \text{?w-parcs}$  **using**  $\text{mem1}$  **by**  $(auto \ simp: \ \text{wellformed}' )$   
**with**  $\langle e' \in \text{set } p \rangle$  **have**  $e' \in \text{set } (\text{co-path } e \ w \ p)$   
**by**  $(induct \ p \ \text{rule: } \ \text{co-path-induct}) \ (\text{force } \ \text{simp: } \langle e = (u,v) \rangle )+$   
**then show**  $\exists x \ p \ y. \text{gen-iapath } V \ x \ p \ y \wedge e' \in \text{set } p$   
**using**  $\langle \text{gen-iapath } V \ x \ (\text{co-path } e \ w \ p) \ y \rangle$  **by** *fast*

**next**  
**assume**  $e' \notin \text{parcs } ?sdG$   
**define**  $a\ b$  **where**  $a = \text{fst } e'$  **and**  $b = \text{snd } e'$   
**then have**  $e' = (a,b)$  **and**  $ab: (a,b) = (u,v) \vee (a,b) = (v,u)$   
**using**  $\langle e' \in \text{parcs } G \rangle \langle e' \notin \text{parcs } ?sdG \rangle \langle e = (u,v) \rangle \text{mem1}$  **by** *auto*  
**obtain**  $x\ p\ y$  **where**  $sdG.\text{gen-iapath } V\ x\ p\ y\ (a,w) \in \text{set } p$   
**using**  $\langle sdG.\text{is-slim } V \rangle \text{sdg-new-parcs } ab$  **by**  $(\text{auto simp: } sdG.\text{is-slim-def})$   
**with**  $\langle e \in \text{parcs } G \rangle \langle w \notin \text{pverts } G \rangle V-G$  **have**  $\text{gen-iapath } V\ x\ (\text{co-path } e\ w\ p)\ y$   
**by**  $(\text{auto intro: } \text{gen-iapath-co-path})$

**have**  $(a,b) \in \text{parcs } G$  *subdivide*  $G\ (a,b)\ w = \text{subdivide } G\ e\ w$   
**using**  $\text{mem1 } \langle e = (u,v) \rangle \langle e' = (a,b) \rangle ab$   
**by**  $(\text{auto intro: arcs-symmetric simp: subdivide.simps})$   
**then have**  $\text{pre-digraph.apath } (\text{subdivide } G\ (a,b)\ w)\ x\ p\ y\ w \neq y$   
**using**  $\text{mem2 } \langle sdG.\text{gen-iapath } V\ x\ p\ y \rangle$  **by**  $(\text{auto simp: } sdG.\text{gen-iapath-def})$   
**then obtain**  $p1\ p2$  **where**  $p: p = p1 \ @\ (a,w) \ #\ (w,b) \ #\ p2$   
**using**  $\text{exists-co-path-decomp1 } \langle (a,b) \in \text{parcs } G \rangle \langle w \notin \text{pverts } G \rangle \langle (a,w) \in \text{set } p \rangle \langle w \neq y \rangle$   
**by** *atomize-elim auto*  
**moreover**  
**from**  $p$  **have**  $\text{co-path } e\ w\ ((a,w) \ #\ (w,b) \ #\ p2) = (a,b) \ #\ \text{co-path } e\ w\ p2$   
**unfolding**  $\langle e = (u,v) \rangle$  **using**  $ab$  **by** *auto*  
**ultimately**  
**have**  $(a,b) \in \text{set } (\text{co-path } e\ w\ p)$   
**unfolding**  $\langle e = (u,v) \rangle$  **using**  $ab\ \langle u \neq w \rangle \langle v \neq w \rangle$   
**by**  $(\text{induct } p\ \text{rule: co-path-induct}) (\text{auto simp: co-path-append})$   
**then show** *?thesis*  
**using**  $\langle \text{gen-iapath } V\ x\ (\text{co-path } e\ w\ p)\ y \rangle \langle e' = (a,b) \rangle$  **by** *fast*  
**qed**  
**then show**  $\text{fst } e' \neq \text{snd } e'$  **by**  $(\text{blast dest: no-loops-in-iapath})$   
**qed**

**have** *unique*:  $\forall u\ v\ p\ q. (\text{gen-iapath } V\ u\ p\ v \wedge \text{gen-iapath } V\ u\ q\ v) \longrightarrow p = q$   
**proof** *safe*  
**fix**  $x\ y\ p\ q$  **assume**  $A: \text{gen-iapath } V\ x\ p\ y\ \text{gen-iapath } V\ x\ q\ y$   
**then have**  $\text{set } p \subseteq \text{parcs } G\ \text{set } q \subseteq \text{parcs } G$   
**by**  $(\text{auto simp: gen-iapath-def apath-def})$   
**then have**  $w-p: (u,w) \notin \text{set } p\ (v,w) \notin \text{set } p$  **and**  $w-q: (u,w) \notin \text{set } q\ (v,w) \notin \text{set } q$   
**using**  $\text{mem1}$  **by**  $(\text{auto simp: wellformed'})$

**from**  $A$  **have**  $sdG.\text{gen-iapath } V\ x\ (\text{sd-path } e\ w\ p)\ y\ sdG.\text{gen-iapath } V\ x\ (\text{sd-path } e\ w\ q)\ y$   
**using**  $\text{mem1 } V-G$  **by**  $(\text{auto intro: gen-iapath-sd-path})$   
**then have**  $\text{sd-path } e\ w\ p = \text{sd-path } e\ w\ q$   
**using**  $\langle sdG.\text{is-slim } V \rangle$  **unfolding**  $sdG.\text{is-slim-def}$  **by** *metis*  
**then have**  $\text{co-path } e\ w\ (\text{sd-path } e\ w\ p) = \text{co-path } e\ w\ (\text{sd-path } e\ w\ q)$  **by** *simp*  
**then show**  $p = q$  **using**  $w-p\ w-q\ \langle e = (u,v) \rangle$  **by**  $(\text{simp add: co-sd-id})$   
**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 simp: )
  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**  $(v,u) \notin \text{parcs } G$  **by** (*auto intro: G.arcs-symmetric*)

**define**  $G'$  **where**  $G' = (\text{pverts} = \text{pverts } G - \{w\},$   
 $\text{parcs} = \{(u,v), (v,u)\} \cup (\text{parcs } G - \{(u,w), (w,u), (v,w), (w,v)\}))$

**have**  $\text{mem-}G'$ :  $(u,v) \in \text{parcs } G'$   $w \notin \text{pverts } G'$  **by** (*auto simp: G'-def*)

**interpret**  $\text{pd-}G'$ : *pair-fin-digraph*  $G'$   
**proof**  
**fix**  $e$  **assume**  $A: e \in \text{parcs } G'$   
**have**  $e \in \text{parcs } G \wedge e \neq (u, w) \wedge e \neq (w, u) \wedge e \neq (v, w) \wedge e \neq (w, v) \implies$   
 $\text{fst } e \neq w$   
 $e \in \text{parcs } G \wedge e \neq (u, w) \wedge e \neq (w, u) \wedge e \neq (v, w) \wedge e \neq (w, v) \implies \text{snd } e$   
 $\neq w$   
**using** *out-arcs in-arcs* **by** *auto*  
**with**  $A$  *uvw-arcs1* **show**  $\text{fst } e \in \text{pverts } G'$   $\text{snd } e \in \text{pverts } G'$   
**using**  $\langle u \neq w \rangle \langle v \neq w \rangle$  **by** (*auto simp: G'-def G.wellformed'*)  
**next**  
**qed** (*auto simp: G'-def arc-to-ends-def*)

**interpret**  $\text{spd-}G'$ : *pair-pseudo-graph*  $G'$   
**proof** (*unfold-locales, simp add: symmetric-def*)  
**have**  $\text{sym } \{(u,v), (v,u)\} \text{ sym } (\text{parcs } G) \text{ sym } \{(u, w), (w, u), (v, w), (w, v)\}$   
**using**  $G.\text{sym-arcs}$  **by** (*auto simp: symmetric-def sym-def*)  
**then have**  $\text{sym } (\{(u,v), (v,u)\} \cup (\text{parcs } G - \{(u,w), (w,u), (v,w), (w,v)\}))$   
**by** (*intro sym-Un*) (*auto simp: sym-diff*)  
**then show**  $\text{sym } (\text{parcs } G')$  **unfolding**  $G'\text{-def}$  **by** *simp*  
**qed**

**have**  $\text{card-}G'$ :  $n = \text{card } (\text{pverts } G' - V)$   
**proof** –  
**have**  $\text{pverts } G - V = \text{insert } w (\text{pverts } G' - V)$   
**using**  $\langle w \in \text{pverts } G - V \rangle$  **by** (*auto simp: G'-def*)  
**then show** *?thesis* **using**  $\langle \text{Suc } n = \text{card } (\text{pverts } G - V) \rangle$   $\text{mem-}G'$  **by** *simp*  
**qed**

**have**  $G\text{-is-sd}$ :  $G = \text{subdivide } G' (u,v) w$  (*is - = ?sdG'*)  
**using**  $\langle w \in \text{pverts } G - V \rangle \langle (u,v) \notin \text{parcs } G \rangle \langle (v,u) \notin \text{parcs } G \rangle$  *uvw-arcs1*  
*uvw-arcs2*  
**by** (*intro pair-pre-digraph.equality*) (*auto simp: G'-def*)

**have**  $\text{gcg-sd}$ : *gen-contr-graph* (*subdivide*  $G' (u,v) w$ )  $V = \text{gen-contr-graph } G' V$   
**proof** –  
**have**  $V \subseteq \text{pverts } G$   
**using**  $\langle G.\text{is-slim } V \rangle$  **by** (*auto simp: G.is-slim-def verts3-def*)  
**moreover**  
**have**  $\text{verts3 } G' = \text{verts3 } G$   
**by** (*simp only: G-is-sd spd-G'.verts3-subdivide[OF \langle (u,v) \in \text{parcs } G' \rangle \langle w \notin*



*pverts G'*])  
**ultimately**  
**have**  $V: V \subseteq \text{pverts } G'$   
**using**  $\langle w \in \text{pverts } G - V \rangle$  **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**  $\langle G.\text{is-slim } V \rangle$  *mem-G'*  $\langle w \notin V \rangle$   
**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**  $\exists H. \text{subgraph (with-proj } H) G \wedge \text{subdivision-pair (contr-graph } G) H$   
**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**  $K :: 'a \text{ pair-pre-digraph}$   
**assumes** *subgraph-K: subgraph K G*  
**assumes** *spd-K: pair-pseudo-graph K*  
**assumes** *kuratowski:  $K_{3,3} \text{ (contr-graph } K) \vee K_5 \text{ (contr-graph } K)$*   
**shows**  $\neg \text{kuratowski-planar } G$   
**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:  $\bigwedge K. (K_{3,3} K \vee K_5 K) \implies \text{graph } K$*   
**by** (*auto simp: complete-digraph-def complete-bipartite-digraph-def*)  
**from** *subdiv-H* **and** *kuratowski*  
**have**  $\exists K. \text{subdivision-pair } K H \wedge (K_{3,3} K \vee K_5 K)$  **by** *blast*  
**then have**  $\exists K \text{ rev-}K \text{ rev-}H. \text{subdivision } (K, \text{rev-}K) (H, \text{rev-}H) \wedge (K_{3,3} K \vee K_5 K)$   
**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*  $\equiv \lambda G :: 'a \text{ pair-pre-digraph. } K_{3,3} G \vee K_5 G$

```

shows kuratowski (contr-graph G)
   $\longleftrightarrow (\exists H. \text{kuratowski } H \wedge \text{subdivision-pair } H \text{ slim} \wedge \text{verts3 } G = \text{verts3 slim})$ 
(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  $\in$  ?l then show v  $\in$  ?r
      using verts-slim-in-G verts3-slim-in-verts3 by auto
  next
    fix v assume v  $\in$  ?r
    have v  $\in$  verts3 (contr-graph G)
    proof -
      have v  $\in$  verts (contr-graph G)
      using  $\langle v \in ?r \rangle$  by (auto simp: verts3-def gen-contr-graph-def)
      then show ?thesis
      using  $\langle ?L \rangle$  unfolding kuratowski-def by (auto simp: verts3-K33 verts3-K5)
    qed
  then have v  $\in$  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 ...  $\leq$  in-degree slim v
    using  $\langle v \in ?r \rangle$  verts3-slim-in-verts3 by (auto intro: S.in-degree-contr)
  finally show v  $\in$  verts3 slim
    using verts3-in-slim-G  $\langle v \in ?r \rangle$  unfolding verts3-def by auto
  qed
  ultimately show ?R using  $\langle ?L \rangle$  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  $\Rightarrow$  bool where
  certify cert  $\equiv$  let C = contr-graph cert in subgraph cert G  $\wedge$  (K3,3 C  $\vee$  K5C)

theorem certify-complete:
  assumes pair-pseudo-graph cert
  assumes subgraph cert G

```

**assumes**  $\exists H. \text{subdivision-pair } H \text{ cert} \wedge (K_{3,3} H \vee K_5 H)$   
**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**  $\neg \text{kuratowski-planar } G$   
**using** *assms* **by** (*intro kuratowski-contr*) (*auto simp: certify-def Let-def*)

**theorem** *certify-characterization*:  
**assumes** *pair-pseudo-graph cert*  
**shows**  $\text{certify cert} \longleftrightarrow \text{subgraph cert } G \wedge \text{verts3 cert} = \text{verts3 } (\text{pair-pre-digraph.slim cert})$   
 $\wedge (\exists H. (K_{3,3} (\text{with-proj } H) \vee K_5 H) \wedge \text{subdivision-pair } H (\text{pair-pre-digraph.slim cert}))$   
**(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**

## 20 Weighted Graphs

**type-synonym**  $'b \text{ weight-fun} = 'b \Rightarrow \text{real}$

**context** *wf-digraph* **begin**

**definition** *awalk-cost*  $:: 'b \text{ weight-fun} \Rightarrow 'b \text{ awalk} \Rightarrow \text{real}$  **where**  
 $\text{awalk-cost } f \text{ es} = \text{sum-list } (\text{map } f \text{ es})$

**lemma** *awalk-cost-Nil[simp]*:  $\text{awalk-cost } f [] = 0$   
**unfolding** *awalk-cost-def* **by** *simp*

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

```

## 21 Shortest Paths

```

context wf-digraph begin

```

```

definition  $\mu$  where
   $\mu f u v \equiv \text{INF } p \in \{p. \text{awalk } u p v\}. \text{ereal } (\text{awalk-cost } f p)$ 

```

```

lemma shortest-path-inf:
  assumes  $\neg(u \rightarrow^* v)$ 
  shows  $\mu f u v = \infty$ 

```

```

proof –
  have *:  $\{p. \text{awalk } u p v\} = \{\}$ 
    using assms by (auto simp: reachable-awalk)
  show  $\mu f u v = \infty$  unfolding  $\mu$ -def *
    by (simp add: top-ereal-def)
qed

```

```

lemma min-cost-le-walk-cost:
  assumes awalk u p v
  shows  $\mu c u v \leq \text{awalk-cost } c p$ 
  using assms unfolding  $\mu$ -def by (auto intro: INF-lower2)

```

```

lemma pos-cost-pos-awalk-cost:
  assumes awalk u p v
  assumes pos-cost:  $\bigwedge e. e \in \text{arcs } G \implies c e \geq 0$ 
  shows  $\text{awalk-cost } c p \geq 0$ 
using assms by (induct p arbitrary: u) (auto simp: awalk-Cons-iff)

```

```

fun mk-cycles-path :: nat
   $\Rightarrow 'b \text{ awalk} \Rightarrow 'b \text{ awalk}$  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  $\bigwedge (i :: nat). i \in UNIV \implies y \leq a + real i * c$ 
  then have l-assm:  $\bigwedge i::nat. y \leq a + real i * c$  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
  end
end

```

```

    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⟩])
  qed

```

**finally show** *?thesis unfolding  $\mu$ -def by simp*  
**qed**

**lemma** *walk-cheaper-path-imp-neg-cyc:*  
**assumes** *p-props: awalk u p v*  
**assumes** *less-path- $\mu$ : awalk-cost f p < (INF p $\in$  {p. apath u p v}. ereal (awalk-cost f p))*  
**shows**  $\exists w c. awalk w c w \wedge w \in set (awalk-verts u p) \wedge awalk-cost f c < 0$   
**proof** –  
**define** *path- $\mu$  where path- $\mu = (INF p \in \{p. apath u p v\}. ereal (awalk-cost f p))$*   
**then have** *awalk u p v and awalk-cost f p < path- $\mu$*   
**using** *p-props less-path- $\mu$  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.prem by (auto simp: path- $\mu$ -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**  $\langle awalk u p v \rangle$  **by** (*auto simp: awalk-appendI*)  
**then have** *verts-ss: set (awalk-verts u (q @ s))  $\subseteq$  set (awalk-verts u p)*  
**using**  $\langle awalk u p v \rangle \langle p = q @ r @ s \rangle$  **by** (*auto simp: set-awalk-verts*)  
  
**show** *?case*  
**proof** (*cases ereal (awalk-cost f (q @ s)) < path- $\mu$* )  
**case** *True then have*  $\exists w c. awalk w c w \wedge w \in set (awalk-verts u (q @ s)) \wedge awalk-cost f c < 0$   
**by** (*rule decomp*)  
**then show** *?thesis using verts-ss by auto*  
**next**  
**case** *False*  
**note**  $\langle awalk-cost f p < path-\mu \rangle$   
**also have** *path- $\mu \leq awalk-cost f (q @ s)$*   
**using** *False by simp*  
**finally have** *awalk-cost f r < 0 using*  $\langle p = q @ r @ s \rangle$  **by** *simp*  
**moreover**  
**have**  $w \in set (awalk-verts u q)$  **using**  $\langle awalk u q w \rangle$  **by** *auto*  
**then have**  $w \in set (awalk-verts u p)$   
**using**  $\langle awalk u p v \rangle \langle awalk u q w \rangle \langle p = q @ r @ s \rangle$   
**by** (*auto simp: set-awalk-verts*)  
**ultimately**  
**show** *?thesis using*  $\langle awalk w r w \rangle$  **by** *auto*  
**qed**  
**qed**  
**qed**

**lemma** (in *fin-digraph*) *neg-inf-imp-neg-cyc*:  
**assumes** *inf-mu*:  $\mu f u v = -\infty$   
**shows**  $\exists p. \text{awalk } u p v \wedge (\exists w c. \text{awalk } w c w \wedge w \in \text{set } (\text{awalk-verts } u p) \wedge \text{awalk-cost } f c < 0)$   
**proof** –  
**define** *path- $\mu$*  **where** *path- $\mu$*  =  $(\text{INF } s \in \{p. \text{apath } u p v\}. \text{ereal } (\text{awalk-cost } f s))$   
  
**have** *awalks-ne*:  $\{p. \text{awalk } u p v\} \neq \{\}$   
**using** *inf-mu* **unfolding**  *$\mu$ -def* **by** *safe* (*simp add: top-ereal-def*)  
**then have** *paths-ne*:  $\{p. \text{apath } u p v\} \sim = \{\}$   
**by** (*auto intro: apath-awalk-to-apath*)  
  
**obtain** *p* **where** *apath* *u p v* *awalk-cost* *f p* = *path- $\mu$*   
**proof** –  
**obtain** *p* **where**  $p \in \{p. \text{apath } u p v\}$  *awalk-cost* *f p* = *path- $\mu$*   
**using** *finite-INF-in[OF apaths-finite paths-ne, of awalk-cost f]*  
**by** (*auto simp: path- $\mu$ -def*)  
**then show** *?thesis* **using** *that* **by** *auto*  
**qed**  
**then have** *path- $\mu$*   $\neq -\infty$  **by** *auto*  
**then have**  $\mu f u v < \text{path-}\mu$  **using** *inf-mu* **by** *simp*  
**then obtain** *pw* **where** *p-def*: *awalk* *u pw v* *awalk-cost* *f pw*  $< \text{path-}\mu$   
**by** *atomize-elim* (*auto simp:  $\mu$ -def INF-less-iff*)  
**then have**  $\exists w c. \text{awalk } w c w \wedge w \in \text{set } (\text{awalk-verts } u pw) \wedge \text{awalk-cost } f c < 0$   
**by** (*intro walk-cheaper-path-imp-neg-cyc*) (*auto simp: path- $\mu$ -def*)  
**with**  $\langle \text{awalk } u pw v \rangle$  **show** *?thesis* **by** *auto*  
**qed**

**lemma** (in *fin-digraph*) *no-neg-cyc-imp-no-neg-inf*:  
**assumes** *no-neg-cyc*:  $\bigwedge p. \text{awalk } u p v$   
 $\implies \neg(\exists w c. \text{awalk } w c w \wedge w \in \text{set } (\text{awalk-verts } u p) \wedge \text{awalk-cost } f c < 0)$   
**shows**  $-\infty < \mu f u v$   
**proof** (*intro ereal-MInfty-lessI notI*)  
**assume**  $\mu f u v = -\infty$   
**then obtain** *p* **where** *p-props*: *awalk* *u p v*  
**and** *ex-cyc*:  $\exists w c. \text{awalk } w c w \wedge w \in \text{set } (\text{awalk-verts } u p) \wedge \text{awalk-cost } f c < 0$   
**by** *atomize-elim* (*rule neg-inf-imp-neg-cyc*)  
**then show** *False* **using** *no-neg-cyc* **by** *blast*  
**qed**

**lemma**  *$\mu$ -reach-conv*:  
 $\mu f u v < \infty \iff u \rightarrow^* v$   
**proof**  
**assume**  $\mu f u v < \infty$   
**then have**  $\{p. \text{awalk } u p v\} \neq \{\}$   
**unfolding**  *$\mu$ -def* **by** *safe* (*simp add: top-ereal-def*)  
**then show**  $u \rightarrow^* v$  **by** (*simp add: reachable-awalk*)  
**next**  
**assume**  $u \rightarrow^* v$



**then obtain**  $p$  **where**  $p$ -props:  $\text{apath } u \ p \ v$   
**by** (*metis reachable-awalk apath-awalk-to-apath*)  
**then have**  $\{p\} \subseteq \{p. \text{apath } u \ p \ v\}$  **by** *simp*  
**then have**  $\mu \ f \ u \ v \leq (\text{INF } p \in \{p\}. \text{ereal } (\text{awalk-cost } f \ p))$   
**unfolding**  $\mu$ -def **by** (*intro INF-superset-mono*) (*auto simp: apath-def*)  
**also have**  $\dots < \infty$  **by** (*simp add: min-def*)  
**finally show**  $\mu \ f \ u \ v < \infty$  .  
**qed**

**lemma** *awalk-to-path-no-neg-cyc-cost*:  
**assumes**  $p$ -props:  $\text{awalk } u \ p \ v$   
**assumes** *no-neg-cyc*:  $\neg (\exists w \ c. \text{awalk } w \ c \ w \wedge w \in \text{set } (\text{awalk-verts } u \ p) \wedge \text{awalk-cost } f \ c < 0)$   
**shows**  $\text{awalk-cost } f \ (\text{awalk-to-apath } p) \leq \text{awalk-cost } f \ p$   
**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:  $p = q \ @ \ r \ @ \ s \ \exists w. \text{awalk } w \ r \ w$  **by** *auto*

**have**  $\text{awalk-cost } f \ (\text{awalk-to-apath } p) = \text{awalk-cost } f \ (\text{awalk-to-apath } (q \ @ \ s))$   
**using** *decomp* **by** (*auto simp: step-awalk-to-apath*[*of - p - q r s*])  
**also have**  $\dots \leq \text{awalk-cost } f \ (q \ @ \ s)$   
**proof** –  
**have**  $\text{awalk } u \ (q \ @ \ s) \ v$   
**using**  $\langle \text{awalk } u \ p \ v \rangle$  *decomp*-props **by** (*auto dest!: awalk-ends-eqD*)  
**then have**  $\text{set } (\text{awalk-verts } u \ (q \ @ \ s)) \subseteq \text{set } (\text{awalk-verts } u \ p)$   
**using**  $\langle \text{awalk } u \ p \ v \rangle$   $\langle p = q \ @ \ r \ @ \ s \rangle$   
**by** (*auto simp add: set-awalk-verts*)  
**then show** ?thesis **using** *decomp.prem*s **by** (*intro decomp.hyps*) *auto*  
**qed**

**also have**  $\dots \leq \text{awalk-cost } f \ p$   
**proof** –  
**obtain**  $w$  **where**  $\text{awalk } u \ q \ w \ \text{awalk } w \ r \ w \ \text{awalk } w \ s \ v$   
**using** *decomp* **by** (*auto elim: awalk-cyc-decompE*)  
**then have**  $w \in \text{set } (\text{awalk-verts } u \ q)$  **by** *auto*  
**then have**  $w \in \text{set } (\text{awalk-verts } u \ p)$   
**using**  $\langle p = q \ @ \ r \ @ \ s \rangle$   $\langle \text{awalk } u \ p \ v \rangle$   $\langle \text{awalk } u \ q \ w \rangle$   
**by** (*auto simp add: set-awalk-verts*)  
**then have**  $0 \leq \text{awalk-cost } f \ r$  **using**  $\langle \text{awalk } w \ r \ w \rangle$   
**using** *decomp.prem*s **by** (*auto simp: not-less*)  
**then show** ?thesis **using**  $\langle p = q \ @ \ r \ @ \ s \rangle$  **by** *simp*  
**qed**

**finally show** ?case .  
**qed**

**lemma** (in *fin-digraph*) *no-neg-cyc-reach-imp-path*:  
**assumes** *reach*:  $u \rightarrow^* v$   
**assumes** *no-neg-cyc*:  $\bigwedge p. \text{awalk } u \ p \ v$   
 $\implies \neg(\exists w \ c. \text{awalk } w \ c \ w \wedge w \in \text{set } (\text{awalk-verts } u \ p) \wedge \text{awalk-cost } f \ c < 0)$   
**shows**  $\exists p. \text{apath } u \ p \ v \wedge \mu \ f \ u \ v = \text{awalk-cost } f \ p$   
**proof** –  
**define** *set-walks* **where** *set-walks* =  $\{p. \text{awalk } u \ p \ v\}$   
**define** *set-paths* **where** *set-paths* =  $\{p. \text{apath } u \ p \ v\}$   
  
**have** *set-paths*  $\neq \{\}$   
**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**  $\mu \ f \ u \ v = (\text{INF } p \in \text{set-walks}. \text{ereal } (\text{awalk-cost } f \ p))$   
**unfolding**  $\mu$ -*def* *set-walks-def* **by** *simp*  
**also have**  $\dots = (\text{INF } p \in \text{set-paths}. \text{ereal } (\text{awalk-cost } f \ p))$   
**proof** (*rule antisym*)  
**have** *awalk-to-apath* ‘*set-walks*  $\subseteq$  *set-paths*  
**unfolding** *set-walks-def* *set-paths-def*  
**by** (*intro* *subsetI*) (*auto* *elim*: *apath-awalk-to-apath*)  
**then have**  $(\text{INF } p \in \text{set-paths}. \text{ereal } (\text{awalk-cost } f \ p))$   
 $\leq (\text{INF } p \in \text{awalk-to-apath } ' \text{set-walks}. \text{ereal } (\text{awalk-cost } f \ p))$   
**by** (*rule* *INF-superset-mono*) *simp*  
**also have**  $\dots = (\text{INF } p \in \text{set-walks}. \text{ereal } (\text{awalk-cost } f \ (\text{awalk-to-apath } p)))$   
**by** (*simp* *add*: *image-comp*)  
**also have**  $\dots \leq (\text{INF } p \in \text{set-walks}. \text{ereal } (\text{awalk-cost } f \ p))$   
**proof** –  
**{** **fix** *p* **assume**  $p \in \text{set-walks}$   
**then have** *awalk* *u p v* **by** (*auto* *simp*: *set-walks-def*)  
**then have**  $\text{awalk-cost } f \ (\text{awalk-to-apath } p) \leq \text{awalk-cost } f \ p$   
**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**  
 $(\text{INF } p \in \text{set-paths}. \text{ereal } (\text{awalk-cost } f \ p))$   
 $\leq (\text{INF } p \in \text{set-walks}. \text{ereal } (\text{awalk-cost } f \ p))$  **by** *simp*  
  
**have** *set-paths*  $\subseteq$  *set-walks*  
**unfolding** *set-paths-def* *set-walks-def* **by** (*auto* *simp*: *apath-def*)  
**then show**  $(\text{INF } p \in \text{set-walks}. \text{ereal } (\text{awalk-cost } f \ p))$   
 $\leq (\text{INF } p \in \text{set-paths}. \text{ereal } (\text{awalk-cost } f \ p))$   
**by** (*rule* *INF-superset-mono*) *simp*  
**qed**  
**also have**  $\dots \in (\lambda p. \text{ereal } (\text{awalk-cost } f \ p)) ' \text{set-paths}$

**using** *apaths-finite*  $\langle \text{set-paths} \neq \{\} \rangle$   
**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*:  $u \rightarrow^* v$   
**assumes** *pos-cost*:  $\bigwedge e. e \in \text{arcs } G \implies c\ e \geq 0$   
**shows**  $\exists p. \text{apath } u\ p\ v \wedge \mu\ c\ u\ v = \text{awalk-cost } c\ p$   
**proof** –  
**have** *pc*:  $\bigwedge u\ p\ v. \text{awalk } u\ p\ v \implies 0 \leq \text{awalk-cost } c\ p$   
**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*:  $\bigwedge e. e \in \text{arcs } G \implies c\ e \geq 0$   
**assumes** *e-props*:  $\text{arc-to-ends } G\ e = (u, v)\ e \in \text{arcs } G$   
**shows**  $\mu\ c\ s\ v \leq \mu\ c\ s\ u + c\ e$   
**proof** *cases*  
**assume**  $\mu\ c\ s\ u = \infty$  **then show** *?thesis* **by** *simp*  
**next**  
**assume**  $\mu\ c\ s\ u \neq \infty$   
**then have**  $\{p. \text{awalk } s\ p\ u\} \neq \{\}$   
**unfolding** *μ-def* **by** *safe* (*simp add: top-ereal-def*)  
**then have**  $s \rightarrow^* u$  **by** (*simp add: reachable-awalk*)  
**with** *pos-cost*  
**obtain** *p* **where** *p-props*:  $\text{apath } s\ p\ u$   
**and** *p-cost*:  $\mu\ c\ s\ u = \text{awalk-cost } c\ p$   
**by** (*metis min-cost-awalk*)

**have**  $\text{awalk } u\ [e]\ v$   
**using** *e-props* **by** (*auto simp: arc-to-ends-def awalk-simps*)  
**with**  $\langle \text{apath } s\ p\ u \rangle$   
**have**  $\text{awalk } s\ (p @ [e])\ v$   
**by** (*auto simp: apath-def awalk-appendI*)  
**then have**  $\mu\ c\ s\ v \leq \text{awalk-cost } c\ (p @ [e])$   
**by** (*rule min-cost-le-walk-cost*)  
**also have**  $\dots \leq \text{awalk-cost } c\ p + c\ e$  **by** *simp*  
**also have**  $\dots \leq \mu\ c\ s\ u + c\ e$  **using** *p-cost* **by** *simp*  
**finally show** *?thesis* .  
**qed**

**lemma** (*in fin-digraph*) *mu-exact-triangle*:  
**assumes**  $v \neq s$   
**assumes**  $s \rightarrow^* v$

**assumes** *nonneg-arcs*:  $\bigwedge e. e \in \text{arcs } G \implies 0 \leq c e$   
**obtains**  $u e$  **where**  $\mu c s v = \mu c s u + c e$  **and**  $\text{arc } e (u, v)$   
**proof** –  
**obtain**  $p$  **where**  $p\text{-path}$ :  $\text{apath } s p v$   
**and**  $p\text{-cost}$ :  $\mu c s v = \text{awalk-cost } c p$   
**using** *assms* **by** (*metis min-cost-awalk*)  
**then obtain**  $e p'$  **where**  $p'\text{-props}$ :  $p = p' @ [e]$  **using**  $\langle v \neq s \rangle$   
**by** (*cases p rule: rev-cases*) (*auto simp: apath-def*)  
**then obtain**  $u$  **where**  $\text{awalk } s p' u \text{ awalk } u [e] v$   
**using**  $\langle \text{apath } s p v \rangle$  **by** (*auto simp: apath-def*)  
**then have**  $\mu\text{-le}$ :  $\mu c s v \leq \mu c s u + c e$  **and**  $\text{arc}$ :  $\text{arc } e (u, v)$   
**using** *nonneg-arcs* **by** (*auto intro!: pos-cost-mu-triangle simp: arc-to-ends-def arc-def*)

**have**  $\mu c s u + c e \leq \text{ereal } (\text{awalk-cost } c p') + \text{ereal } (c e)$   
**using**  $\langle \text{awalk } s p' u \rangle$   
**by** (*fast intro: add-right-mono min-cost-le-walk-cost*)  
**also have**  $\dots = \text{awalk-cost } c p$  **using**  $p'\text{-props}$  **by** *simp*  
**also have**  $\dots = \mu c s v$  **using**  $p\text{-cost}$  **by** *simp*  
**finally**  
**have**  $\mu c s v = \mu c s u + c e$  **using**  $\mu\text{-le}$  **by** *auto*  
**then show** *?thesis* **using**  $\text{arc}$  ..

**qed**

**lemma** (*in fin-digraph*) *mu-exact-triangle-Ex*:

**assumes**  $v \neq s$   
**assumes**  $s \rightarrow^* v$   
**assumes**  $\bigwedge e. e \in \text{arcs } G \implies 0 \leq c e$   
**shows**  $\exists u e. \mu c s v = \mu c s u + c e \wedge \text{arc } e (u, v)$   
**using** *assms* **by** (*metis mu-exact-triangle*)

**lemma** (*in fin-digraph*) *mu-Inf-triangle*:

**assumes**  $v \neq s$   
**assumes**  $\bigwedge e. e \in \text{arcs } G \implies 0 \leq c e$   
**shows**  $\mu c s v = \text{Inf } \{ \mu c s u + c e \mid u e. \text{arc } e (u, v) \}$  (*is - = Inf ?S*)

**proof** *cases*

**assume**  $s \rightarrow^* v$   
**then obtain**  $u e$  **where**  $\mu c s v = \mu c s u + c e$   $\text{arc } e (u, v)$   
**using** *assms* **by** (*metis mu-exact-triangle*)  
**then have**  $\text{Inf } ?S \leq \mu c s v$  **by** (*auto intro: Complete-Lattices.Inf-lower*)  
**also have**  $\dots \leq \text{Inf } ?S$  **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**  $\neg s \rightarrow^* v$   
**then have**  $\mu c s v = \infty$  **by** (*metis shortest-path-inf*)  
**define**  $S$  **where**  $S = ?S$   
**show**  $\mu c s v = \text{Inf } S$

```

proof cases
  assume  $S = \{\}$ 
  then show ?thesis unfolding  $\langle \mu \ c \ s \ v = \infty \rangle$ 
    by (simp add: top-ereal-def)
next
  assume  $S \neq \{\}$ 
  { fix  $x$  assume  $x \in S$ 
    then obtain  $u \ e$  where arc e (u,v) and x-val: x =  $\mu \ c \ s \ u + c \ e$ 
      unfolding S-def by auto
      then have  $\neg s \rightarrow^* u$  using  $\langle \neg s \rightarrow^* v \rangle$  by (metis reachable-arc-trans)
      then have  $\mu \ c \ s \ u + c \ e = \infty$  by (simp add: shortest-path-inf)
      then have  $x = \infty$  using x-val by simp }
  then have  $S = \{\infty\}$  using  $\langle S \neq \{\} \rangle$  by auto
  then show ?thesis using  $\langle \mu \ c \ s \ v = \infty \rangle$  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] 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.