Graph Theory

By Lars Noschinski

September 13, 2023

Abstract

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

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

Definitions and nomenclature are based on [1].

Contents

theory *Rtrancl-On*
imports *Main*
begin

1 Reflexive-Transitive Closure on a Domain

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

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

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

lemma *in-rtrancl-on-in-F*:
assumes (*a*,*b*) ∈ *rtrancl-on F r* **shows** *a* ∈ *F b* ∈ *F*
⟨*proof*⟩

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

lemma *rtrancl-on-trans*:
assumes (*a*,*b*) ∈ *rtrancl-on F r* (*b*,*c*) ∈ *rtrancl-on F r* **shows** (*a*,*c*) ∈ *rtrancl-on F r*
⟨*proof*⟩

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

⟨*proof*⟩

lemma *rtrancl-on-converseI*:
assumes $(y, x) \in$ *rtrancl-on* $F\ r$ **shows** $(x, y) \in$ *rtrancl-on* $F\ (r^{-1})$
⟨*proof*⟩

theorem *rtrancl-on-converseD*:
assumes $(y, x) \in$ *rtrancl-on* $F\ (r^{-1})$ **shows** $(x, y) \in$ *rtrancl-on* $F\ r$
⟨*proof*⟩

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

lemma *converse-rtrancl-on-cases*:
assumes $(a, b) \in$ *rtrancl-on* $F\ r$
obtains (*base*) $a = b\ b \in F$
| (*step*) c **where** $(a,c) \in r\ (c,b) \in$ *rtrancl-on* $F\ r$
⟨*proof*⟩

lemma *rtrancl-on-sym*:
assumes *sym* r **shows** *sym* (*rtrancl-on* $F\ r$)
⟨*proof*⟩

lemma *rtrancl-on-mono*:
assumes $s \subseteq r\ F \subseteq G\ (a,b) \in$ *rtrancl-on* $F\ s$ **shows** $(a,b) \in$ *rtrancl-on* $G\ r$
⟨*proof*⟩

lemma *rtrancl-consistent-rtrancl-on*:
assumes $(a,b) \in r^*$
and $a \in F\ b \in F$
and *consistent*: $\bigwedge a\ b.\ [\![\ a \in F; (a,b) \in r\ ]\!] \Longrightarrow b \in F$
shows $(a,b) \in$ *rtrancl-on* $F\ r$
⟨*proof*⟩

lemma *rtrancl-on-rtranclI*:
$(a,b) \in$ *rtrancl-on* $F\ r \Longrightarrow (a,b) \in r^*$
⟨*proof*⟩

lemma *rtrancl-on-sub-rtrancl*:
rtrancl-on $F\ r \subseteq r\widehat{}*$
⟨*proof*⟩

end

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

begin

2 Additional theorems for base libraries

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

lemma *ereal-Inf-finite-Min*:
fixes S :: *ereal set*
assumes *finite* S **and** $S \neq \{\}$
shows *Inf* S = *Min* S
⟨*proof*⟩

lemma *finite-INF-in*:
fixes $f :: 'a \Rightarrow ereal$
assumes *finite* S
assumes $S \neq \{\}$
shows $(INF\ s{\in}\ S.\ f\ s) \in f\ `\ S$
⟨*proof*⟩

lemma *not-mem-less-INF*:
fixes $f :: 'a \Rightarrow 'b$ :: *complete-lattice*
assumes $f\ x < (INF\ s{\in}\ S.\ f\ s)$
assumes $x \in S$
shows *False*
⟨*proof*⟩

lemma *sym-diff*:
assumes *sym A sym B* **shows** *sym* $(A - B)$
⟨*proof*⟩

2.1 List

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

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

lemma *tl-rev*:

tl (*rev p*) = *rev* (*butlast p*)
⟨*proof*⟩

lemma *butlast-rev*:
butlast (*rev p*) = *rev* (*tl p*)
⟨*proof*⟩

lemma *take-drop-take*:
take n xs @ *drop n* (*take m xs*) = *take* (*max n m*) *xs*
⟨*proof*⟩

lemma *drop-take-drop*:
drop n (*take m xs*) @ *drop m xs* = *drop* (*min n m*) *xs*
⟨*proof*⟩

lemma *not-distinct-decomp-min-prefix*:
assumes ¬ *distinct ws*
shows ∃ *xs ys zs y*. *ws* = *xs* @ *y* # *ys* @ *y* # *zs* ∧ *distinct xs* ∧ *y* ∉ *set xs* ∧ *y* ∉ *set ys*
⟨*proof*⟩

lemma *not-distinct-decomp-min-not-distinct*:
assumes ¬ *distinct ws*
shows ∃*xs y ys zs*. *ws* = *xs* @ *y* # *ys* @ *y* # *zs* ∧ *distinct* (*ys* @ [*y*])
⟨*proof*⟩

lemma *card-Ex-subset*:
k ≤ *card M* ⟹ ∃ *N*. *N* ⊆ *M* ∧ *card N* = *k*
⟨*proof*⟩

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

3 NOMATCH simproc

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

definition *NOMATCH* :: *'a* ⇒ *'a* ⇒ *bool* **where** *NOMATCH val pat* ≡ *True*
lemma *NOMATCH-cong*[*cong*]: *NOMATCH val pat* = *NOMATCH val pat* ⟨*proof*⟩

⟨*ML*⟩

This setup ensures that a rewrite rule of the form *NOMATCH val pat* ⟹ *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 ⇒ 'a
head :: 'b ⇒ 'a

definition *arc-to-ends* :: ('a,'b) *pre-digraph* ⇒ 'b ⇒ 'a × 'a **where**
arc-to-ends G e ≡ (*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* ∈ *arcs G* ⟹ *tail G e* ∈ *verts G*
assumes *head-in-verts*[*simp*]: *e* ∈ *arcs G* ⟹ *head G e* ∈ *verts G*
begin

lemma *wf-digraph*: *wf-digraph G* ⟨*proof*⟩

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

end

definition *arcs-ends* :: ('a,'b) *pre-digraph* ⇒ ('a × 'a) *set* **where**
arcs-ends G ≡ *arc-to-ends G* ‘ *arcs G*

definition *symmetric* :: ('a,'b) *pre-digraph* ⇒ *bool* **where**
symmetric G ≡ *sym* (*arcs-ends G*)

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

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

locale *loopfree-digraph* = *wf-digraph* +
assumes *no-loops*: *e* ∈ *arcs G* ⟹ *tail G e* ≠ *head G e*

locale *nomulti-digraph* = *wf-digraph* +
assumes *no-multi-arcs*: ⋀*e1 e2*. ⟦*e1* ∈ *arcs G*; *e2* ∈ *arcs G*;
arc-to-ends G e1 = *arc-to-ends G e2*⟧ ⟹ *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*
⟨*proof*⟩

lemma (**in** *wf-digraph*) *sym-digraphI*[*intro*]:
  assumes *symmetric G*
  shows *sym-digraph G*
⟨*proof*⟩

lemma (**in** *digraph*) *graphI*[*intro*]:
  assumes *symmetric G*
  shows *graph G*
⟨*proof*⟩

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

lemma (**in** *fin-digraph*) *fin-digraph*: *fin-digraph G*
  ⟨*proof*⟩

lemma (**in** *nomulti-digraph*) *nomulti-digraph*: *nomulti-digraph G* ⟨*proof*⟩

lemma *arcs-ends-conv*: *arcs-ends G* = (λ*e*. (*tail G e*, *head G e*)) ‘ *arcs G*
  ⟨*proof*⟩

lemma *symmetric-conv*: *symmetric G* ⟷ (∀ *e1* ∈ *arcs G*. ∃ *e2* ∈ *arcs G*. *tail G e1* = *head G e2* ∧ *head G e1* = *tail G e2*)
  ⟨*proof*⟩

lemma *arcs-ends-symmetric*:

assumes *symmetric G*
shows (*u*,*v*) ∈ *arcs-ends G* ⟹ (*v*,*u*) ∈ *arcs-ends G*
⟨*proof*⟩

lemma (**in** *nomulti-digraph*) *inj-on-arc-to-ends*:
inj-on (*arc-to-ends G*) (*arcs G*)
⟨*proof*⟩

4.1 Reachability

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

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

definition *reachable* :: ('*a*,'*b*) *pre-digraph* ⇒ '*a* ⇒ '*a* ⇒ *bool* (- →*1 - [*100*,*100*] *40*) **where**
reachable G u v ≡ (*u*,*v*) ∈ *rtrancl-on* (*verts G*) (*arcs-ends G*)

lemma *reachableE*[*elim*]:
assumes *u* →*G v*
obtains *e* **where** *e* ∈ *arcs G tail G e* = *u head G e* = *v*
⟨*proof*⟩

lemma (**in** *loopfree-digraph*) *adj-not-same*:
assumes *a* → *a* **shows** *False*
⟨*proof*⟩

lemma *reachable-in-vertsE*:
assumes *u* →*G v* **obtains** *u* ∈ *verts G v* ∈ *verts G*
⟨*proof*⟩

lemma *symmetric-reachable*:
assumes *symmetric G v* →*G *w* **shows** *w* →*G *v*
⟨*proof*⟩

lemma *reachable-rtranclI*:
u →*G *v* ⟹ (*u*, *v*) ∈ (*arcs-ends G*)*
⟨*proof*⟩

context *wf-digraph* **begin**

lemma *adj-in-verts*:
assumes *u* →*G v* **shows** *u* ∈ *verts G v* ∈ *verts G*
⟨*proof*⟩

lemma *dominatesI*: **assumes** *arc-to-ends G a* = (*u*,*v*) *a* ∈ *arcs G* **shows** *u* →$_G$ *v*
⟨*proof*⟩

lemma *reachable-refl* [*intro*!, *Pure.intro*!, *simp*]: *v* ∈ *verts G* ⟹ *v* →* *v*
⟨*proof*⟩

lemma *adj-reachable-trans*[*trans*]:
assumes *a* →$_G$ *b b* →$^*{}_G$ *c* **shows** *a* →$^*{}_G$ *c*
⟨*proof*⟩

lemma *reachable-adj-trans*[*trans*]:
assumes *a* →$^*{}_G$ *b b* →$_G$ *c* **shows** *a* →$^*{}_G$ *c*
⟨*proof*⟩

lemma *reachable-adjI* [*intro*, *simp*]: *u* → *v* ⟹ *u* →* *v*
⟨*proof*⟩

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

lemma *reachable-induct*[*consumes 1*, *case-names base step*]:
assumes *major*: *u* →$^*{}_G$ *v*
and *cases*: *u* ∈ *verts G* ⟹ *P u*
⋀*x y*. ⟦*u* →$^*{}_G$ *x*; *x* →$_G$ *y*; *P x*⟧ ⟹ *P y*
shows *P v*
⟨*proof*⟩

lemma *converse-reachable-induct*[*consumes 1*, *case-names base step*, *induct pred*: *reachable*]:
assumes *major*: *u* →$^*{}_G$ *v*
and *cases*: *v* ∈ *verts G* ⟹ *P v*
⋀*x y*. ⟦*x* →$_G$ *y*; *y* →$^*{}_G$ *v*; *P y*⟧ ⟹ *P x*
shows *P u*
⟨*proof*⟩

lemma (**in** *pre-digraph*) *converse-reachable-cases*:
assumes *u* →$^*{}_G$ *v*
obtains (*base*) *u* = *v u* ∈ *verts G*
| (*step*) *w* **where** *u* →$_G$ *w w* →$^*{}_G$ *v*
⟨*proof*⟩

lemma *reachable-in-verts*:
assumes *u* →* *v* **shows** *u* ∈ *verts G v* ∈ *verts G*
⟨*proof*⟩

lemma *reachable1-in-verts*:

assumes $u \rightarrow^+ v$ **shows** $u \in \mathit{verts}\ G\ v \in \mathit{verts}\ G$
⟨*proof*⟩

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

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*⟩

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

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

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

lemma *reachable-conv*:
$u \rightarrow^* v \longleftrightarrow (u,v) \in (\mathit{arcs\text{-}ends}\ G)\widehat{\ }* \cap (\mathit{verts}\ G \times \mathit{verts}\ G)$
⟨*proof*⟩

lemma *reachable-conv'*:
assumes $u \in \mathit{verts}\ G$
shows $u \rightarrow^* v \longleftrightarrow (u,v) \in (\mathit{arcs\text{-}ends}\ G)^*$ (**is** *?L* = *?R*)
⟨*proof*⟩

end

lemma (**in** *sym-digraph*) *symmetric-reachable'*:
assumes $v \rightarrow^*{}_G w$ **shows** $w \rightarrow^*{}_G v$
⟨*proof*⟩

4.2 Degrees of vertices

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

definition *out-arcs* :: ('*a*, '*b*) *pre-digraph* ⇒ '*a* ⇒ '*b set* **where**
out-arcs $G\ v \equiv \{e \in \mathit{arcs}\ G.\ \mathit{tail}\ G\ e = v\}$

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*)
⟨*proof*⟩

lemma (**in** *fin-digraph*) *finite-out-arcs*[*intro*]:
finite (*out-arcs G v*)
⟨*proof*⟩

lemma *in-in-arcs-conv*[*simp*]:
e ∈ *in-arcs G v* ⟷ *e* ∈ *arcs G* ∧ *head G e* = *v*
⟨*proof*⟩

lemma *in-out-arcs-conv*[*simp*]:
e ∈ *out-arcs G v* ⟷ *e* ∈ *arcs G* ∧ *tail G e* = *v*
⟨*proof*⟩

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

lemma *in-arcs-int-arcs*[*simp*]: *in-arcs G u* ∩ *arcs G* = *in-arcs G u* **and**
out-arcs-int-arcs[*simp*]: *out-arcs G u* ∩ *arcs G* = *out-arcs G u*
⟨*proof*⟩

lemma *in-arcs-in-arcs*: *x* ∈ *in-arcs G u* ⟹ *x* ∈ *arcs G*
and *out-arcs-in-arcs*: *x* ∈ *out-arcs G u* ⟹ *x* ∈ *arcs G*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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

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

⟨*proof*⟩

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*
⟨*proof*⟩

lemma *add-del-arc-collapse*[*simp*]: *pre-digraph.add-arc* (*del-arc a*) *a* = *add-arc a*
⟨*proof*⟩

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

lemma *del-del-arc-collapse*[*simp*]: *pre-digraph.del-arc* (*del-arc a*) *a* = *del-arc a*
⟨*proof*⟩

lemma *add-arc-commute*: *pre-digraph.add-arc* (*add-arc b*) *a* = *pre-digraph.add-arc* (*add-arc a*) *b*
⟨*proof*⟩

lemma *del-arc-commute*: *pre-digraph.del-arc* (*del-arc b*) *a* = *pre-digraph.del-arc* (*del-arc a*) *b*
⟨*proof*⟩

lemma *del-arc-in*: *a* ∉ *arcs G* ⟹ *del-arc a* = *G*
⟨*proof*⟩

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

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

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

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*)
⟨*proof*⟩

lemma (**in** *wf-digraph*) *add-arc-in*: $a \in arcs\ G \Longrightarrow add\text{-}arc\ a = G$
⟨*proof*⟩

end

context *wf-digraph* **begin**

lemma *wf-digraph-add-arc*[*intro*]:
wf-digraph (*add-arc a*) ⟨*proof*⟩

lemma *wf-digraph-del-arc*[*intro*]:
wf-digraph (*del-arc a*) ⟨*proof*⟩

lemma *wf-digraph-del-vert*: *wf-digraph* (*del-vert u*)
⟨*proof*⟩

lemma *wf-digraph-add-vert*: *wf-digraph* (*add-vert u*)
⟨*proof*⟩

lemma *del-vert-add-vert*:
assumes $u \notin verts\ G$
shows *pre-digraph.del-vert* (*add-vert u*) *u* = *G*
⟨*proof*⟩

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*⟩

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*⟩

lemma *in-degree-del-arc-iff*:
in-degree (*del-arc a*) *u* = (*if head G a* = *u* ∧ *a* ∈ *arcs G then in-degree G u* − *1 else in-degree G u*)
⟨*proof*⟩

lemma *out-degree-del-arc-iff*:
out-degree (*del-arc a*) *u* = (*if tail G a* = *u* ∧ *a* ∈ *arcs G then out-degree G u* − *1 else out-degree G u*)

⟨*proof*⟩

lemma *fin-digraph-del-vert*: *fin-digraph* (*del-vert u*)
⟨*proof*⟩

lemma *fin-digraph-del-arc*: *fin-digraph* (*del-arc a*)
⟨*proof*⟩

end

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

5 Bidirected Graphs

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

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

context *bidirected-digraph* **begin**

lemma *bidirected-digraph*[*intro*!]: *bidirected-digraph G arev*
⟨*proof*⟩

lemma *arev-arev*[*simp*]: *arev* (*arev a*) = *a*
⟨*proof*⟩

lemma *arev-o-arev*[*simp*]: *arev o arev* = *id*
⟨*proof*⟩

lemma *arev-eq*: *a* ∉ *arcs G* ⟹ *arev a* = *a*
⟨*proof*⟩

lemma *arev-neq*: *a* ∈ *arcs G* ⟹ *arev a* ≠ *a*
⟨*proof*⟩

lemma *arev-in-arcs*[*simp*]: $a \in arcs\ G \Longrightarrow arev\ a \in arcs\ G$
⟨*proof*⟩

lemma *head-arev*[*simp*]:
assumes $a \in arcs\ G$ **shows** $head\ G\ (arev\ a) = tail\ G\ a$
⟨*proof*⟩

lemma *ate-arev*[*simp*]:
assumes $a \in arcs\ G$ **shows** $arc\text{-}to\text{-}ends\ G\ (arev\ a) = prod.swap\ (arc\text{-}to\text{-}ends\ G\ a)$
⟨*proof*⟩

lemma *bij-arev*: *bij arev*
⟨*proof*⟩

lemma *arev-permutes-arcs*: *arev permutes arcs G*
⟨*proof*⟩

lemma *arev-eq-iff*: $\bigwedge x\ y.\ arev\ x = arev\ y \longleftrightarrow x = y$
⟨*proof*⟩

lemma *in-arcs-eq*: *in-arcs G w* = *arev* ‘ *out-arcs G w*
⟨*proof*⟩

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

lemma *even-card-loops*:
even (*card* (*in-arcs G w* ∩ *out-arcs G w*)) (**is** *even* (*card ?S*))
⟨*proof*⟩

end

sublocale *bidirected-digraph* ⊆ *sym-digraph*
⟨*proof*⟩

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*
⟨*proof*⟩

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

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

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

end

6.1 Basic Lemmas

lemma (**in** *pre-digraph*) *awalk-verts-conv*:
  awalk-verts u p = (if p = [] then [u] else map (tail G) p @ [head G (last p)])
⟨*proof*⟩

lemma (**in** *pre-digraph*) *awalk-verts-conv'*:
assumes *cas u p v*
shows *awalk-verts u p* = (*if p* = [] *then* [*u*] *else tail G* (*hd p*) # *map* (*head G*) *p*)
⟨*proof*⟩

lemma (**in** *pre-digraph*) *length-awalk-verts*:
length (*awalk-verts u p*) = *Suc* (*length p*)
⟨*proof*⟩

lemma (**in** *pre-digraph*) *awalk-verts-ne-eq*:
assumes *p* ≠ []
shows *awalk-verts u p* = *awalk-verts v p*
⟨*proof*⟩

lemma (**in** *pre-digraph*) *awalk-verts-non-Nil*[*simp*]:
awalk-verts u p ≠ []
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *awalk-verts-in-verts*:
assumes *u* ∈ *verts G set p* ⊆ *arcs G v* ∈ *set* (*awalk-verts u p*)
shows *v* ∈ *verts G*
⟨*proof*⟩

lemma
assumes *u* ∈ *verts G set p* ⊆ *arcs G*
shows *awhd-in-verts*: *awhd u p* ∈ *verts G*
and *awlast-in-verts*: *awlast u p* ∈ *verts G*
⟨*proof*⟩

lemma *awalk-conv*:
awalk u p v = (*set* (*awalk-verts u p*) ⊆ *verts G*
∧ *set p* ⊆ *arcs G*
∧ *awhd u p* = *u* ∧ *awlast u p* = *v* ∧ *cas u p v*)
⟨*proof*⟩

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

lemma *awalkE*[*elim*]:
assumes *awalk u p v*

obtains *set (awalk-verts u p) ⊆ verts G set p ⊆ arcs G cas u p v*
awhd u p = u awlast u p = v
⟨*proof*⟩

lemma *awalk-Nil-iff*:
awalk u [] v ⟷ u = v ∧ u ∈ verts G
⟨*proof*⟩

lemma *trail-Nil-iff*:
trail u [] v ⟷ u = v ∧ u ∈ verts G
⟨*proof*⟩

lemma *apath-Nil-iff*: *apath u [] v ⟷ u = v ∧ u ∈ verts G*
⟨*proof*⟩

lemma *awalk-hd-in-verts*: *awalk u p v ⟹ u ∈ verts G*
⟨*proof*⟩

lemma *awalk-last-in-verts*: *awalk u p v ⟹ v ∈ verts G*
⟨*proof*⟩

lemma *hd-in-awalk-verts*:
awalk u p v ⟹ u ∈ set (awalk-verts u p)
apath u p v ⟹ u ∈ set (awalk-verts u p)
⟨*proof*⟩

lemma *awalk-Cons-iff*:
awalk u (e # es) w ⟷ e ∈ arcs G ∧ u = tail G e ∧ awalk (head G e) es w
⟨*proof*⟩

lemma *trail-Cons-iff*:
trail u (e # es) w ⟷ e ∈ arcs G ∧ u = tail G e ∧ e ∉ set es ∧ trail (head G e) es w
⟨*proof*⟩

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

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 ∈ arcs G ⟹ awalk (tail G e) [e] (head G e)
⟨*proof*⟩

lemma *apath-nonempty-ends*:

assumes *apath u p v*
assumes $p \neq []$
shows $u \neq v$
⟨*proof*⟩

lemma *awalk-ConsI*:
assumes *awalk v es w*
assumes $e \in arcs\ G$ **and** *arc-to-ends G e* = (*u*,*v*)
shows *awalk u* (*e* # *es*) *w*
⟨*proof*⟩

lemma (**in** *pre-digraph*) *awalkI-apath*:
assumes *apath u p v* **shows** *awalk u p v*
⟨*proof*⟩

lemma *arcE*:
assumes *arc e* (*u*,*v*)
assumes $[\![e \in arcs\ G;\ tail\ G\ e = u;\ head\ G\ e = v]\!] \Longrightarrow P$
shows *P*
⟨*proof*⟩

lemma *in-arcs-imp-in-arcs-ends*:
assumes $e \in arcs\ G$
shows (*tail G e*, *head G e*) ∈ *arcs-ends G*
⟨*proof*⟩

lemma *set-awalk-verts-cas*:
assumes *cas u p v*
shows *set* (*awalk-verts u p*) = {*u*} ∪ *set* (*map* (*tail G*) *p*) ∪ *set* (*map* (*head G*) *p*)
⟨*proof*⟩

lemma *set-awalk-verts-not-Nil-cas*:
assumes *cas u p v* $p \neq []$
shows *set* (*awalk-verts u p*) = *set* (*map* (*tail G*) *p*) ∪ *set* (*map* (*head G*) *p*)
⟨*proof*⟩

lemma *set-awalk-verts*:
assumes *awalk u p v*
shows *set* (*awalk-verts u p*) = {*u*} ∪ *set* (*map* (*tail G*) *p*) ∪ *set* (*map* (*head G*) *p*)
⟨*proof*⟩

lemma *set-awalk-verts-not-Nil*:
assumes *awalk u p v* $p \neq []$
shows *set* (*awalk-verts u p*) = *set* (*map* (*tail G*) *p*) ∪ *set* (*map* (*head G*) *p*)

⟨*proof*⟩

lemma
awhd-of-awalk: *awalk u p v* ⟹ *awhd u p* = *u* **and**
awlast-of-awalk: *awalk u p v* ⟹ *NOMATCH* (*awlast u p*) *v* ⟹ *awlast u p* = *v*
⟨*proof*⟩
lemmas *awends-of-awalk*[*simp*] = *awhd-of-awalk awlast-of-awalk*

lemma *awalk-verts-arc1*:
assumes *e* ∈ *set p*
shows *tail G e* ∈ *set* (*awalk-verts u p*)
⟨*proof*⟩

lemma *awalk-verts-arc2*:
assumes *awalk u p v e* ∈ *set p*
shows *head G e* ∈ *set* (*awalk-verts u p*)
⟨*proof*⟩

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

6.2 Appending awalks

lemma (**in** *pre-digraph*) *cas-append-iff*[*simp*]:
cas u (*p* @ *q*) *v* ⟷ *cas u p* (*awlast u p*) ∧ *cas* (*awlast u p*) *q v*
⟨*proof*⟩

lemma *cas-ends*:
assumes *cas u p v cas u′ p v′*
shows (*p* ≠ [] ∧ *u* = *u′* ∧ *v* = *v′*) ∨ (*p* = [] ∧ *u* = *v* ∧ *u′* = *v′*)
⟨*proof*⟩

lemma *awalk-ends*:
assumes *awalk u p v awalk u′ p v′*
shows (*p* ≠ [] ∧ *u* = *u′* ∧ *v* = *v′*) ∨ (*p* = [] ∧ *u* = *v* ∧ *u′* = *v′*)
⟨*proof*⟩

lemma *awalk-ends-eqD*:
assumes *awalk u p u awalk v p w*
shows *v* = *w*
⟨*proof*⟩

lemma *awalk-empty-ends*:
assumes *awalk u* [] *v*

shows $u = v$
⟨*proof*⟩

lemma *apath-ends*:
assumes *apath u p v* **and** *apath u′ p v′*
shows $(p \neq [] \wedge u \neq v \wedge u = u' \wedge v = v') \vee (p = [] \wedge u = v \wedge u' = v')$
⟨*proof*⟩

lemma *awalk-append-iff*[*simp*]:
awalk u (*p* @ *q*) *v* ⟷ *awalk u p* (*awlast u p*) ∧ *awalk* (*awlast u p*) *q v* (**is** *?L* ⟷ *?R*)
⟨*proof*⟩

lemma *awlast-append*:
awlast u (*p* @ *q*) = *awlast* (*awlast u p*) *q*
⟨*proof*⟩

lemma *awhd-append*:
awhd u (*p* @ *q*) = *awhd* (*awhd u q*) *p*
⟨*proof*⟩

declare *awalkE*[*rule del*]

lemma *awalkE′*[*elim*]:
assumes *awalk u p v*
obtains *set* (*awalk-verts u p*) ⊆ *verts G set p* ⊆ *arcs G cas u p v*
awhd u p = *u awlast u p* = *v u* ∈ *verts G v* ∈ *verts G*
⟨*proof*⟩

lemma *awalk-appendI*:
assumes *awalk u p v*
assumes *awalk v q w*
shows *awalk u* (*p* @ *q*) *w*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *awalk-verts-append*:
assumes *awalk u* (*p* @ *q*) *v*
shows *awalk-verts u* (*p* @ *q*) = *awalk-verts u p* @ *tl* (*awalk-verts* (*awlast u p*) *q*)
⟨*proof*⟩

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

⟨*proof*⟩

lemma *apath-append-iff*:
apath u (p @ q) v ⟷ apath u p (awlast u p) ∧ apath (awlast u p) q v ∧
set (awalk-verts u p) ∩ set (tl (awalk-verts (awlast u p) q)) = {} (**is** *?L ⟷ ?R*)
⟨*proof*⟩

lemma (**in** *wf-digraph*) *set-awalk-verts-append-cas*:
assumes *cas u p v cas v q w*
shows *set (awalk-verts u (p @ q)) = set (awalk-verts u p) ∪ set (awalk-verts v q)*
⟨*proof*⟩

lemma (**in** *wf-digraph*) *set-awalk-verts-append*:
assumes *awalk u p v awalk v q w*
shows *set (awalk-verts u (p @ q)) = set (awalk-verts u p) ∪ set (awalk-verts v q)*
⟨*proof*⟩

lemma *cas-takeI*:
assumes *cas u p v awlast u (take n p) = v′*
shows *cas u (take n p) v′*
⟨*proof*⟩

lemma *cas-dropI*:
assumes *cas u p v awlast u (take n p) = u′*
shows *cas u′ (drop n p) v*
⟨*proof*⟩

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*⟩

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′])*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *awalk-decomp*:
assumes *awalk u p v*

assumes $w \in set\ (awalk\text{-}verts\ u\ p)$
shows $\exists\, q\ r.\ p = q\ @\ r \wedge awalk\ u\ q\ w \wedge awalk\ w\ r\ v$
⟨*proof*⟩

lemma *awalk-not-distinct-decomp*:
assumes *awalk u p v*
assumes $\neg$ *distinct* (*awalk-verts u p*)
shows $\exists\, q\ r\ s.\ p = q\ @\ r\ @\ s \wedge distinct\ (awalk\text{-}verts\ u\ q)$
$\wedge\ 0 < length\ r$
$\wedge\ (\exists\, w.\ awalk\ u\ q\ w \wedge awalk\ w\ r\ w \wedge awalk\ w\ s\ v)$
⟨*proof*⟩

lemma *apath-decomp-disjoint*:
assumes *apath u p v*
assumes $p = q\ @\ r$
assumes $x \in set\ (awalk\text{-}verts\ u\ q)\ x \in set\ (tl\ (awalk\text{-}verts\ (awlast\ u\ q)\ r))$
shows *False*
⟨*proof*⟩

6.3 Cycles

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

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

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

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

lemma *cycle-altdef*:
cycle p ⟷ *closed-w p* ∧ (∃ *u. distinct* (*tl* (*awalk-verts u p*)))
⟨*proof*⟩

lemma (**in** *wf-digraph*) *distinct-tl-verts-imp-distinct*:
assumes *awalk u p v*
assumes *distinct* (*tl* (*awalk-verts u p*))
shows *distinct p*
⟨*proof*⟩

lemma (**in** *wf-digraph*) *distinct-verts-imp-distinct*:
assumes *awalk u p v*
assumes *distinct* (*awalk-verts u p*)
shows *distinct p*
⟨*proof*⟩

lemma (**in** *wf-digraph*) *cycle-conv*:
cycle p $\longleftrightarrow$ ($\exists$ *u*. *awalk u p u* $\wedge$ *distinct* (*tl* (*awalk-verts u p*)) $\wedge$ *distinct p* $\wedge$ *p* $\neq$ []) ⟨*proof*⟩

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*⟩

lemma (**in** *wf-digraph*) *closed-w-imp-cycle*:
assumes *closed-w p* **shows** $\exists$ *p*. *cycle p* ⟨*proof*⟩

6.4 Reachability

lemma *reachable1-awalk*:
u $\rightarrow^+$ *v* $\longleftrightarrow$ ($\exists$ *p*. *awalk u p v* $\wedge$ *p* $\neq$ []) ⟨*proof*⟩

lemma *reachable-awalk*:
u $\rightarrow^*$ *v* $\longleftrightarrow$ ($\exists$ *p*. *awalk u p v*) ⟨*proof*⟩

lemma *reachable-awalkI*[*intro?*]:
assumes *awalk u p v*
shows *u* $\rightarrow^*$ *v* ⟨*proof*⟩

lemma *reachable1-awalkI*:
awalk v p w $\Longrightarrow$ *p* $\neq$ [] $\Longrightarrow$ *v* $\rightarrow^+$ *w* ⟨*proof*⟩

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

lemma *awalk-verts-reachable-from*:
assumes *awalk u p v w* $\in$ *set* (*awalk-verts u p*) **shows** *u* $\rightarrow^*{}_G$ *w* ⟨*proof*⟩

lemma *awalk-verts-reachable-to*:
assumes *awalk u p v w* $\in$ *set* (*awalk-verts u p*) **shows** *w* $\rightarrow^*{}_G$ *v* ⟨*proof*⟩

6.5 Paths

lemma (**in** *fin-digraph*) *length-apath-less*:
assumes *apath u p v*
shows *length p* < *card* (*verts G*)
⟨*proof*⟩

lemma (**in** *fin-digraph*) *length-apath*:
assumes *apath u p v*
shows *length p* ≤ *card* (*verts G*)
⟨*proof*⟩

lemma (**in** *fin-digraph*) *apaths-finite-triple*:
shows *finite* {(*u*,*p*,*v*). *apath u p v*}
⟨*proof*⟩

lemma (**in** *fin-digraph*) *apaths-finite*:
shows *finite* {*p*. *apath u p v*}
⟨*proof*⟩

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

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

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

lemma *awalk-cyc-decomp-has-prop*:
assumes *awalk u p v* **and** ¬*distinct* (*awalk-verts u p*)
shows *is-awalk-cyc-decomp p* (*awalk-cyc-decomp p*)
⟨*proof*⟩

lemma *awalk-cyc-decompE*:
assumes *dec*: *awalk-cyc-decomp p* = (*q*,*r*,*s*)
assumes *p-props*: *awalk u p v* ¬*distinct* (*awalk-verts u p*)
obtains *p* = *q* @ *r* @ *s distinct* (*awalk-verts u q*) ∃ *w*. *awalk u q w* ∧ *awalk w r w* ∧ *awalk w s v closed-w r*
⟨*proof*⟩

lemma *awalk-cyc-decompE'*:

assumes *p-props*: *awalk u p v* ¬*distinct* (*awalk-verts u p*)
obtains *q r s* **where** *p* = *q* @ *r* @ *s distinct* (*awalk-verts u q*) ∃ *w*. *awalk u q w* ∧ *awalk w r w* ∧ *awalk w s v closed-w r*
⟨*proof*⟩

termination *awalk-to-apath*
⟨*proof*⟩
declare *awalk-to-apath.simps*[*simp del*]

lemma *awalk-to-apath-induct*[*consumes 1*, *case-names path decomp*]:
assumes *awalk*: *awalk u p v*
assumes *dist*: ⋀*p*. *awalk u p v* ⟹ *distinct* (*awalk-verts u p*) ⟹ *P p*
assumes *dec*: ⋀*p q r s*. ⟦*awalk u p v*; *awalk-cyc-decomp p* = (*q*,*r*,*s*); ¬*distinct* (*awalk-verts u p*); *P* (*q* @ *s*)⟧ ⟹ *P p*
shows *P p*
⟨*proof*⟩

lemma *step-awalk-to-apath*:
assumes *awalk*: *awalk u p v*
and *decomp*: *awalk-cyc-decomp p* = (*q*, *r*, *s*)
and *dist*: ¬ *distinct* (*awalk-verts u p*)
shows *awalk-to-apath p* = *awalk-to-apath* (*q* @ *s*)
⟨*proof*⟩

lemma *apath-awalk-to-apath*:
assumes *awalk u p v*
shows *apath u* (*awalk-to-apath p*) *v*
⟨*proof*⟩

lemma (**in** *wf-digraph*) *awalk-to-apath-subset*:
assumes *awalk u p v*
shows *set* (*awalk-to-apath p*) ⊆ *set p*
⟨*proof*⟩

lemma *reachable-apath*:
u →* *v* ⟷ (∃ *p*. *apath u p v*)
⟨*proof*⟩

lemma *no-loops-in-apath*:
assumes *apath u p v a* ∈ *set p* **shows** *tail G a* ≠ *head G a*
⟨*proof*⟩

end

end

theory *Pair-Digraph*

imports
Digraph
Bidirected-Digraph
Arc-Walk
begin

7 Digraphs without Parallel Arcs

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

This theory introduces such a specialisation of digraphs.

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

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

declare [[*coercion with-proj*]]

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

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

lemma *with-proj-simps*[*simp*]:
verts (*with-proj G*) = *pverts G*
arcs (*with-proj G*) = *parcs G*
arcs-ends (*with-proj G*) = *parcs G*
tail (*with-proj G*) = *fst*
head (*with-proj G*) = *snd*
⟨*proof*⟩

lemma *cas-with-proj-eq*: *pre-digraph.cas* (*with-proj G*) = *pcas*
⟨*proof*⟩

lemma *awalk-verts-with-proj-eq*: *pre-digraph.awalk-verts* (*with-proj G*) = *pawalk-verts*
⟨*proof*⟩

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

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

end

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

lemma *in-arcsD1*: $(u,v) \in parcs\ G \Longrightarrow u \in pverts\ G$
  and *in-arcsD2*: $(u,v) \in parcs\ G \Longrightarrow v \in pverts\ G$
  ⟨*proof*⟩

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

end

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

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

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

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

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

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

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

sublocale *pair-pre-digraph* ⊆ *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*
  ⟨*proof*⟩

sublocale *pair-wf-digraph* ⊆ *wf-digraph with-proj G*
  rewrites *verts G* = *pverts G* **and** *arcs G* = *parcs G* **and** *tail G* = *fst* **and** *head G* = *snd*
    and *arcs-ends G* = *parcs G*

and *pre-digraph.awalk-verts G* = *pawalk-verts*
and *pre-digraph.cas G* = *pcas*
⟨*proof*⟩

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

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

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

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

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

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

and *pre-digraph.cas G = pcas*
⟨*proof*⟩

sublocale *pair-graph* ⊆ *pair-bidirected-digraph* ⟨*proof*⟩

lemma *wf-digraph-wp-iff*: *wf-digraph* (*with-proj G*) = *pair-wf-digraph G* (**is** *?L* ⟷ *?R*)
⟨*proof*⟩

lemma (**in** *pair-fin-digraph*) *pair-fin-digraph*[*intro*!]: *pair-fin-digraph G* ⟨*proof*⟩

context *pair-digraph* **begin**

lemma *pair-wf-digraph*[*intro*!]: *pair-wf-digraph G* ⟨*proof*⟩

lemma *pair-digraph*[*intro*!]: *pair-digraph G* ⟨*proof*⟩

lemma (**in** *pair-loopfree-digraph*) *no-loops′*:
(*u*,*v*) ∈ *parcs G* ⟹ *u* ≠ *v*
⟨*proof*⟩

end

lemma (**in** *pair-wf-digraph*) *apath-succ-decomp*:
assumes *apath u p v*
assumes (*x*,*y*) ∈ *set p*
assumes *y* ≠ *v*
shows ∃ *p1 z p2*. *p* = *p1* @ (*x*,*y*) # (*y*,*z*) # *p2* ∧ *x* ≠ *z* ∧ *y* ≠ *z*
⟨*proof*⟩

lemma (**in** *pair-sym-digraph*) *arcs-symmetric*:
(*a*,*b*) ∈ *parcs G* ⟹ (*b*,*a*) ∈ *parcs G*
⟨*proof*⟩

lemma (**in** *pair-pseudo-graph*) *pair-pseudo-graph*[*intro*]: *pair-pseudo-graph G* ⟨*proof*⟩

lemma (**in** *pair-graph*) *pair-graph*[*intro*]: *pair-graph G* ⟨*proof*⟩
lemma (**in** *pair-graph*) *pair-graphD-graph*: *graph G* ⟨*proof*⟩

lemma *pair-graphI-graph*:
assumes *graph* (*with-proj G*) **shows** *pair-graph G*
⟨*proof*⟩

lemma *pair-loopfreeI-loopfree*:
assumes *loopfree-digraph* (*with-proj G*) **shows** *pair-loopfree-digraph G*
⟨*proof*⟩

7.1 Path reversal for Pair Digraphs

This definition is only meaningful in *Pair-Digraph*

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

lemma *rev-path-append*[*simp*]: *rev-path* (*p* @ *q*) = *rev-path q* @ *rev-path p*
  ⟨*proof*⟩

lemma *rev-path-rev-path*[*simp*]:
  rev-path (*rev-path p*) = *p*
  ⟨*proof*⟩

lemma *rev-path-empty*[*simp*]:
  rev-path p = [] ⟷ *p* = []
  ⟨*proof*⟩

lemma *rev-path-eq*: *rev-path p* = *rev-path q* ⟷ *p* = *q*
  ⟨*proof*⟩

lemma (**in** *pair-sym-digraph*)
  assumes *awalk u p v*
  shows *awalk-verts-rev-path*: *awalk-verts v* (*rev-path p*) = *rev* (*awalk-verts u p*)
    and *awalk-rev-path'*: *awalk v* (*rev-path p*) *u*
⟨*proof*⟩

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

lemma (**in** *pair-sym-digraph*) *apath-rev-path*[*simp*]:
  apath v (*rev-path p*) *u* = *apath u p v*
⟨*proof*⟩

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* ∪ {*w*},
    parcs = (*parcs G* − {(*u*,*v*),(*v*,*u*)}) ∪ {(*u*,*w*), (*w*,*u*), (*w*, *v*), (*v*, *w*)}|)

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

subdivide an arc in a path

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

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

lemma *co-path-nonempty*[*simp*]: *co-path e w p* = [] ⟷ *p* = []
⟨*proof*⟩

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

lemma *verts-subdivide*[*simp*]: *pverts* (*subdivide G e w*) = *pverts G* ∪ {*w*}
⟨*proof*⟩

lemma *arcs-subdivide*[*simp*]:
shows *parcs* (*subdivide G* (*u*,*v*) *w*) = (*parcs G* − {(*u*,*v*),(*v*,*u*)}) ∪ {(*u*,*w*), (*w*,*u*), (*w*, *v*), (*v*, *w*)}
⟨*proof*⟩

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

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

lemma *co-path-induct*[*case-names empty single co corev pass*]:
fixes $e :: 'a \times 'a$
and $w :: 'a$
and $p :: ('a \times 'a)$ *awalk*
assumes *Nil*: $P\ e\ w\ []$
and *ConsNil*:$\bigwedge e'.\ P\ e\ w\ [e']$
and *ConsCons1*: $\bigwedge e1\ e2\ es.\ e1 = (fst\ e,\ w) \wedge e2 = (w,\ snd\ e) \Longrightarrow P\ e\ w\ es \Longrightarrow$
$P\ e\ w\ (e1\ \#\ e2\ \#\ es)$
and *ConsCons2*: $\bigwedge e1\ e2\ es.\ \neg(e1 = (fst\ e,\ w) \wedge e2 = (w,\ snd\ e)) \wedge$
$e1 = (snd\ e,\ w) \wedge e2 = (w,\ fst\ e) \Longrightarrow P\ e\ w\ es \Longrightarrow$
$P\ e\ w\ (e1\ \#\ e2\ \#\ es)$
and *ConsCons3*: $\bigwedge e1\ e2\ es.$
$\neg\ (e1 = (fst\ e,\ w) \wedge e2 = (w,\ snd\ e)) \Longrightarrow$
$\neg\ (e1 = (snd\ e,\ w) \wedge e2 = (w,\ fst\ e)) \Longrightarrow P\ e\ w\ (e2\ \#\ es) \Longrightarrow$
$P\ e\ w\ (e1\ \#\ e2\ \#\ es)$
shows $P\ e\ w\ p$
⟨*proof*⟩

lemma *co-sd-id*:
assumes $(u,w) \notin set\ p\ (v,w) \notin set\ p$
shows *co-path* $(u,v)\ w$ (*sd-path* $(u,v)\ w\ p) = p$
⟨*proof*⟩

lemma *sd-path-id*:
assumes $(x,y) \notin set\ p\ (y,x) \notin set\ p$
shows *sd-path* $(x,y)\ w\ p = p$
⟨*proof*⟩

lemma (**in** *pair-wf-digraph*) *pair-wf-digraph-subdivide*:
assumes *props*: $e \in parcs\ G\ w \notin pverts\ G$
shows *pair-wf-digraph* (*subdivide G e w*) (**is** *pair-wf-digraph ?sG*)
⟨*proof*⟩

lemma (**in** *pair-sym-digraph*) *pair-sym-digraph-subdivide*:
assumes *props*: $e \in parcs\ G\ w \notin pverts\ G$
shows *pair-sym-digraph* (*subdivide G e w*) (**is** *pair-sym-digraph ?sG*)
⟨*proof*⟩

lemma (**in** *pair-loopfree-digraph*) *pair-loopfree-digraph-subdivide*:
assumes *props*: $e \in parcs\ G\ w \notin pverts\ G$
shows *pair-loopfree-digraph* (*subdivide G e w*) (**is** *pair-loopfree-digraph ?sG*)
⟨*proof*⟩

lemma (**in** *pair-bidirected-digraph*) *pair-bidirected-digraph-subdivide*:
assumes *props*: $e \in parcs\ G\ w \notin pverts\ G$
shows *pair-bidirected-digraph* (*subdivide G e w*) (**is** *pair-bidirected-digraph ?sG*)
⟨*proof*⟩

lemma (**in** *pair-pseudo-graph*) *pair-pseudo-graph-subdivide*:
assumes *props*: $e \in parcs\ G\ w \notin pverts\ G$
shows *pair-pseudo-graph* (*subdivide G e w*) (**is** *pair-pseudo-graph ?sG*)
⟨*proof*⟩

lemma (**in** *pair-graph*) *pair-graph-subdivide*:
assumes $e \in parcs\ G\ w \notin pverts\ G$
shows *pair-graph* (*subdivide G e w*) (**is** *pair-graph ?sG*)
⟨*proof*⟩

lemma *arcs-subdivideD*:
assumes $x \in parcs$ (*subdivide G e w*) *fst* $x \neq w$ *snd* $x \neq w$
shows $x \in parcs\ G$
⟨*proof*⟩

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

lemma
assumes *path*: *apath u p v*
assumes *elems*: $e \in parcs\ G\ w \notin pverts\ G$
shows *apath-sd-path*: *pre-digraph.apath* (*subdivide G e w*) *u* (*sd-path e w p*) *v* (**is** *?A*)
and *set-awalk-verts-sd-path*: *set* (*awalk-verts u* (*sd-path e w p*))
$\subseteq$ *set* (*awalk-verts u p*) $\cup$ $\{w\}$ (**is** *?B*)
⟨*proof*⟩

lemma
assumes *elems*: $e \in parcs\ G\ w \notin pverts\ G\ u \in pverts\ G\ v \in pverts\ G$
assumes *path*: *pre-digraph.apath* (*subdivide G e w*) *u p v*
shows *apath-co-path*: *apath u* (*co-path e w p*) *v* (**is** *?thesis-path*)
and *set-awalk-verts-co-path*: *set* (*awalk-verts u* (*co-path e w p*)) = *set* (*awalk-verts u p*) − $\{w\}$ (**is** *?thesis-set*)
⟨*proof*⟩

end

7.3 Bidirected Graphs

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

lemma *bidirected-digraph-rev-conv-pair*:
assumes *bidirected-digraph* (*with-proj G*) *rev-G*
shows *rev-G* = *swap-in* (*parcs G*)
⟨*proof*⟩

lemma (**in** *pair-bidirected-digraph*) *bidirected-digraph*:
bidirected-digraph (*with-proj G*) (*swap-in* (*parcs G*))

⟨*proof*⟩

lemma *pair-bidirected-digraphI-bidirected-digraph*:
assumes *bidirected-digraph* (*with-proj G*) (*swap-in* (*parcs G*))
shows *pair-bidirected-digraph G*
⟨*proof*⟩

end

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

8 Components of (Symmetric) Digraphs

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

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

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

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

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

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

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

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

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

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

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

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

definition (**in** *pre-digraph*) *sccs* :: ('*a*,'*b*) *pre-digraph set* **where**
sccs ≡ {*H*. *induced-subgraph H G* ∧ *strongly-connected H* ∧ ¬(∃ *H'*. *induced-subgraph*
H' G
∧ *strongly-connected H'* ∧ *verts H* ⊂ *verts H'*)}

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

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

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

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

8.1 Compatible Graphs

lemma *compatible-tail*:
assumes *compatible G H* **shows** *tail G* = *tail H*
⟨*proof*⟩

lemma *compatible-head*:
assumes *compatible G H* **shows** *head G* = *head H*
⟨*proof*⟩

lemma *compatible-cas*:
assumes *compatible G H* **shows** *pre-digraph.cas G* = *pre-digraph.cas H*
⟨*proof*⟩

lemma *compatible-awalk-verts*:
assumes *compatible G H* **shows** *pre-digraph.awalk-verts G* = *pre-digraph.awalk-verts H*
⟨*proof*⟩

lemma *compatibleI-with-proj*[*intro*]:
shows *compatible* (*with-proj G*) (*with-proj H*)
⟨*proof*⟩

8.2 Basic lemmas

lemma (**in** *sym-digraph*) *graph-symmetric*:
shows (*u*,*v*) ∈ *arcs-ends G* ⟹ (*v*,*u*) ∈ *arcs-ends G*
⟨*proof*⟩

lemma *strongly-connectedI*[*intro*]:
assumes *verts G* ≠ {} ⋀*u v*. *u* ∈ *verts G* ⟹ *v* ∈ *verts G* ⟹ *u* →**G v*
shows *strongly-connected G*
⟨*proof*⟩

lemma *strongly-connectedE*[*elim*]:
assumes *strongly-connected G*
assumes (⋀*u v*. *u* ∈ *verts G* ∧ *v* ∈ *verts G* ⟹ *u* →**G v*) ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *subgraph-imp-subverts*:
assumes *subgraph H G*
shows *verts H* ⊆ *verts G*
⟨*proof*⟩

lemma *induced-imp-subgraph*:
assumes *induced-subgraph H G*
shows *subgraph H G*
⟨*proof*⟩

lemma (**in** *pre-digraph*) *in-sccs-imp-induced*:
assumes *c* ∈ *sccs*
shows *induced-subgraph c G*
⟨*proof*⟩

lemma *spanning-tree-imp-tree*[*dest*]:
assumes *spanning-tree H G*
shows *tree H*
⟨*proof*⟩

lemma *tree-imp-connected*[*dest*]:
assumes *tree G*

shows *connected G*
⟨*proof*⟩

lemma *spanning-treeI*[*intro*]:
assumes *spanning H G*
assumes *tree H*
shows *spanning-tree H G*
⟨*proof*⟩

lemma *spanning-treeE*[*elim*]:
assumes *spanning-tree H G*
assumes *tree H* ∧ *spanning H G* ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *spanningE*[*elim*]:
assumes *spanning H G*
assumes *subgraph H G* ∧ *verts G* = *verts H* ⟹ *P*
shows *P*
⟨*proof*⟩

lemma (**in** *pre-digraph*) *in-sccsI*[*intro*]:
assumes *induced-subgraph c G*
assumes *strongly-connected c*
assumes ¬(∃ *c'*. *induced-subgraph c' G* ∧ *strongly-connected c'* ∧
verts c ⊂ *verts c'*)
shows *c* ∈ *sccs*
⟨*proof*⟩

lemma (**in** *pre-digraph*) *in-sccsE*[*elim*]:
assumes *c* ∈ *sccs*
assumes *induced-subgraph c G* ⟹ *strongly-connected c* ⟹ ¬ (∃ *d*.
induced-subgraph d G ∧ *strongly-connected d* ∧ *verts c* ⊂ *verts d*) ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *subgraphI*:
assumes *verts H* ⊆ *verts G*
assumes *arcs H* ⊆ *arcs G*
assumes *compatible G H*
assumes *wf-digraph H*
assumes *wf-digraph G*
shows *subgraph H G*
⟨*proof*⟩

lemma *subgraphE*[*elim*]:
assumes *subgraph H G*
obtains *verts H* ⊆ *verts G arcs H* ⊆ *arcs G compatible G H wf-digraph H wf-digraph G*

⟨*proof*⟩

lemma *induced-subgraphI*[*intro*]:
assumes *subgraph H G*
assumes *arcs H* = {*e* ∈ *arcs G*. *tail G e* ∈ *verts H* ∧ *head G e* ∈ *verts H*}
shows *induced-subgraph H G*
⟨*proof*⟩

lemma *induced-subgraphE*[*elim*]:
assumes *induced-subgraph H G*
assumes ⟦*subgraph H G*; *arcs H* = {*e* ∈ *arcs G*. *tail G e* ∈ *verts H* ∧ *head G e* ∈ *verts H*}⟧ ⟹ *P*
shows *P*
⟨*proof*⟩

lemma *pverts-mk-symmetric*[*simp*]: *pverts* (*mk-symmetric G*) = *verts G*
and *parcs-mk-symmetric*:
parcs (*mk-symmetric G*) = (⋃ *e*∈*arcs G*. {(*tail G e*, *head G e*), (*head G e*, *tail G e*)})
⟨*proof*⟩

lemma *arcs-ends-mono*:
assumes *subgraph H G*
shows *arcs-ends H* ⊆ *arcs-ends G*
⟨*proof*⟩

lemma (**in** *wf-digraph*) *subgraph-refl*: *subgraph G G*
⟨*proof*⟩

lemma (**in** *wf-digraph*) *induced-subgraph-refl*: *induced-subgraph G G*
⟨*proof*⟩

8.3 The underlying symmetric graph of a digraph

lemma (**in** *wf-digraph*) *wellformed-mk-symmetric*[*intro*]: *pair-wf-digraph* (*mk-symmetric G*)
⟨*proof*⟩

lemma (**in** *fin-digraph*) *pair-fin-digraph-mk-symmetric*[*intro*]: *pair-fin-digraph* (*mk-symmetric G*)
⟨*proof*⟩

lemma (**in** *digraph*) *digraph-mk-symmetric*[*intro*]: *pair-digraph* (*mk-symmetric G*)
⟨*proof*⟩

lemma (**in** *wf-digraph*) *reachable-mk-symmetricI*:
assumes *u* →* *v* **shows** *u* →*$_{mk\text{-}symmetric\ G}$ *v*
⟨*proof*⟩

lemma (**in** *wf-digraph*) *adj-mk-symmetric-eq*:
symmetric G $\Longrightarrow$ *parcs* (*mk-symmetric G*) = *arcs-ends G*
⟨*proof*⟩

lemma (**in** *wf-digraph*) *reachable-mk-symmetric-eq*:
assumes *symmetric G* **shows** $u \rightarrow^*_{\textit{mk-symmetric } G} v \longleftrightarrow u \rightarrow^* v$ (**is** *?L* $\longleftrightarrow$ *?R*)
⟨*proof*⟩

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*⟩

lemma *symmetric-mk-symmetric*:
symmetric (*mk-symmetric G*)
⟨*proof*⟩

8.4 Subgraphs and Induced Subgraphs

lemma *subgraph-trans*:
assumes *subgraph G H subgraph H I* **shows** *subgraph G I*
⟨*proof*⟩

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*⟩

lemma (**in** *digraph*) *digraph-subgraph*:
assumes *subgraph H G* **shows** *digraph H*
⟨*proof*⟩

lemma (**in** *pre-digraph*) *adj-mono*:
assumes $u \rightarrow_H v$ *subgraph H G*
shows $u \rightarrow v$
⟨*proof*⟩

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

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*
⟨*proof*⟩

lemma (**in** *wf-digraph*) *subgraph-apath-imp-apath*:
assumes *path*: *pre-digraph.apath H u p v*
assumes *sub*: *subgraph H G*
shows *apath u p v*
⟨*proof*⟩

lemma *subgraph-mk-symmetric*:
assumes *subgraph H G*
shows *subgraph* (*mk-symmetric H*) (*mk-symmetric G*)
⟨*proof*⟩

lemma (**in** *fin-digraph*) *subgraph-in-degree*:
assumes *subgraph H G*
shows *in-degree H v* ≤ *in-degree G v*
⟨*proof*⟩

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

lemma (**in** *wf-digraph*) *subgraph-del-vert*: *subgraph* (*del-vert u*) *G*
⟨*proof*⟩

lemma (**in** *wf-digraph*) *subgraph-del-arc*: *subgraph* (*del-arc a*) *G*
⟨*proof*⟩

8.5 Induced subgraphs

lemma *wf-digraphI-induced*:
assumes *induced-subgraph H G*
shows *wf-digraph H*
⟨*proof*⟩

lemma (**in** *digraph*) *digraphI-induced*:
assumes *induced-subgraph H G*
shows *digraph H*
⟨*proof*⟩

Computes the subgraph of G induced by *vs*

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

lemma *induce-subgraph-verts*[*simp*]:
verts (*G* ↾ *vs*) = *vs*

⟨*proof*⟩

lemma *induce-subgraph-arcs*[*simp*]:
arcs (*G* ↾ *vs*) = {*e* ∈ *arcs G*. *tail G e* ∈ *vs* ∧ *head G e* ∈ *vs*}
⟨*proof*⟩

lemma *induce-subgraph-tail*[*simp*]:
tail (*G* ↾ *vs*) = *tail G*
⟨*proof*⟩

lemma *induce-subgraph-head*[*simp*]:
head (*G* ↾ *vs*) = *head G*
⟨*proof*⟩

lemma *compatible-induce-subgraph*: *compatible* (*G* ↾ *S*) *G*
⟨*proof*⟩

lemma (**in** *wf-digraph*) *induced-induce*[*intro*]:
assumes *vs* ⊆ *verts G*
shows *induced-subgraph* (*G* ↾ *vs*) *G*
⟨*proof*⟩

lemma (**in** *wf-digraph*) *wellformed-induce-subgraph*[*intro*]:
wf-digraph (*G* ↾ *vs*)
⟨*proof*⟩

lemma *induced-graph-imp-symmetric*:
assumes *symmetric G*
assumes *induced-subgraph H G*
shows *symmetric H*
⟨*proof*⟩

lemma (**in** *sym-digraph*) *induced-graph-imp-graph*:
assumes *induced-subgraph H G*
shows *sym-digraph H*
⟨*proof*⟩

lemma (**in** *wf-digraph*) *induce-reachable-preserves-paths*:
assumes *u* →**G v*
shows *u* →**G* ↾ {*w*. *u* →**G w*} *v*
⟨*proof*⟩

lemma *induce-subgraph-ends*[*simp*]:
arc-to-ends (*G* ↾ *S*) = *arc-to-ends G*
⟨*proof*⟩

lemma *dominates-induce-subgraphD*:
assumes *u* →*G* ↾ *S v* **shows** *u* →*G v*
⟨*proof*⟩

context *wf-digraph* **begin**

lemma *reachable-induce-subgraphD*:
assumes $u \rightarrow^*_{G \restriction S} v$ $S \subseteq$ *verts* G **shows** $u \rightarrow^*_G v$
⟨*proof*⟩

lemma *dominates-induce-ss*:
assumes $u \rightarrow_{G \restriction S} v$ $S \subseteq T$ **shows** $u \rightarrow_{G \restriction T} v$
⟨*proof*⟩

lemma *reachable-induce-ss*:
assumes $u \rightarrow^*_{G \restriction S} v$ $S \subseteq T$ **shows** $u \rightarrow^*_{G \restriction T} v$
⟨*proof*⟩

lemma *awalk-verts-induce*:
pre-digraph.awalk-verts $(G \restriction S) =$ *awalk-verts*
⟨*proof*⟩

lemma (**in** −) *cas-subset*:
assumes *pre-digraph.cas* G u p v *subgraph* G H
shows *pre-digraph.cas* H u p v
⟨*proof*⟩

lemma *cas-induce*:
assumes *cas* u p v *set* (*awalk-verts* u p) $\subseteq S$
shows *pre-digraph.cas* $(G \restriction S)$ u p v
⟨*proof*⟩

lemma *awalk-induce*:
assumes *awalk* u p v *set* (*awalk-verts* u p) $\subseteq S$
shows *pre-digraph.awalk* $(G \restriction S)$ u p v
⟨*proof*⟩

lemma *subgraph-induce-subgraphI*:
assumes $V \subseteq$ *verts* G **shows** *subgraph* $(G \restriction V)$ G
⟨*proof*⟩

end

lemma *induced-subgraphI'*:
assumes *subg*:*subgraph* H G
assumes *max*: $\bigwedge H'$. *subgraph* H' $G \Longrightarrow$ (*verts* $H' \neq$ *verts* $H \vee$ *arcs* $H' \subseteq$ *arcs* H)
shows *induced-subgraph* H G
⟨*proof*⟩

lemma (**in** *pre-digraph*) *induced-subgraph-altdef*:
induced-subgraph H $G \longleftrightarrow$ *subgraph* H $G \wedge$ ($\forall H'$. *subgraph* H' $G \longrightarrow$ (*verts* H'

$\neq$ *verts H* $\vee$ *arcs H'* $\subseteq$ *arcs H*)) (**is** *?L* $\longleftrightarrow$ *?R*)
⟨*proof*⟩

8.6 Unions of Graphs

lemma
verts-union[*simp*]: *verts* (*union G H*) = *verts G* $\cup$ *verts H* **and**
arcs-union[*simp*]: *arcs* (*union G H*) = *arcs G* $\cup$ *arcs H* **and**
tail-union[*simp*]: *tail* (*union G H*) = *tail G* **and**
head-union[*simp*]: *head* (*union G H*) = *head G*
⟨*proof*⟩

lemma *wellformed-union*:
assumes *wf-digraph G wf-digraph H compatible G H*
shows *wf-digraph* (*union G H*)
⟨*proof*⟩

lemma *subgraph-union-iff*:
assumes *wf-digraph H1 wf-digraph H2 compatible H1 H2*
shows *subgraph* (*union H1 H2*) *G* $\longleftrightarrow$ *subgraph H1 G* $\wedge$ *subgraph H2 G*
⟨*proof*⟩

lemma *subgraph-union*[*intro*]:
assumes *subgraph H1 G compatible H1 G*
assumes *subgraph H2 G compatible H2 G*
shows *subgraph* (*union H1 H2*) *G*
⟨*proof*⟩

lemma *union-fin-digraph*:
assumes *fin-digraph G fin-digraph H compatible G H*
shows *fin-digraph* (*union G H*)
⟨*proof*⟩

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'*)
⟨*proof*⟩

8.7 Maximal Subgraphs

lemma (**in** *pre-digraph*) *max-subgraph-mp*:
assumes *max-subgraph Q x* $\bigwedge x$. *P x* $\Longrightarrow$ *Q x P x* **shows** *max-subgraph P x*
⟨*proof*⟩

lemma (**in** *pre-digraph*) *max-subgraph-prop*: *max-subgraph P x* $\Longrightarrow$ *P x*
⟨*proof*⟩

lemma (**in** *pre-digraph*) *max-subgraph-subg-eq*:
assumes *max-subgraph P H1 max-subgraph P H2 subgraph H1 H2*

shows *H1 = H2*
⟨*proof*⟩

lemma *subgraph-induce-subgraphI2*:
assumes *subgraph H G* **shows** *subgraph H* (*G* ↾ *verts H*)
⟨*proof*⟩

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

lemma (**in** *pre-digraph*) *induced-subgraphI-arc-mono*:
assumes *max-subgraph P H*
assumes *arc-mono P*
shows *induced-subgraph H G*
⟨*proof*⟩

lemma (**in** *pre-digraph*) *induced-subgraph-altdef2*:
induced-subgraph H G ⟷ *max-subgraph* (λ*H'*. *verts H'* = *verts H*) *H* (**is** *?L* ⟷ *?R*)
⟨*proof*⟩

lemma (**in** *pre-digraph*) *max-subgraphI*:
assumes *P x subgraph x G* ⋀*y*. ⟦*x* ≠ *y*; *subgraph x y*; *subgraph y G*⟧ ⟹ ¬*P y*
shows *max-subgraph P x*
⟨*proof*⟩

lemma (**in** *pre-digraph*) *subgraphI-max-subgraph*: *max-subgraph P x* ⟹ *subgraph x G*
⟨*proof*⟩

8.8 Connected and Strongly Connected Graphs

context *wf-digraph* **begin**

lemma *in-sccs-verts-conv-reachable*:
S ∈ *sccs-verts* ⟷ *S* ≠ {} ∧ (∀ *u* ∈ *S*. ∀ *v* ∈ *S*. *u* →**G v*) ∧ (∀ *u* ∈ *S*. ∀ *v*. *v* ∉ *S* ⟶ ¬*u* →**G v* ∨ ¬*v* →**G u*)
⟨*proof*⟩

lemma *sccs-verts-disjoint*:
assumes *S* ∈ *sccs-verts T* ∈ *sccs-verts S* ≠ *T* **shows** *S* ∩ *T* = {}
⟨*proof*⟩

lemma *strongly-connected-spanning-imp-strongly-connected*:
assumes *spanning H G*
assumes *strongly-connected H*
shows *strongly-connected G*

⟨*proof*⟩

lemma *strongly-connected-imp-induce-subgraph-strongly-connected*:
assumes *subg*: *subgraph H G*
assumes *sc*: *strongly-connected H*
shows *strongly-connected* (*G* ↾ (*verts H*))
⟨*proof*⟩

lemma *in-sccs-vertsI-sccs*:
assumes $S \in$ *verts* ‘ *sccs* **shows** $S \in$ *sccs-verts*
⟨*proof*⟩

end

lemma *arc-mono-strongly-connected*[*intro*,*simp*]: *arc-mono strongly-connected*
⟨*proof*⟩

lemma (**in** *pre-digraph*) *sccs-altdef2*:
sccs = {*H*. *max-subgraph strongly-connected H*} (**is** *?L* = *?R*)
⟨*proof*⟩

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

lemma *reach-in*: $\bigwedge u\ v.\ [\![u \in S;\ v \in S]\!] \Longrightarrow u \rightarrow^*_G v$
and *not-reach-out*: $\bigwedge u\ v.\ [\![u \in S;\ v \notin S]\!] \Longrightarrow \neg u \rightarrow^*_G v \vee \neg v \rightarrow^*_G u$
and *not-empty*: $S \neq \{\}$
⟨*proof*⟩

lemma *reachable-induced*:
assumes *conn*: $u \in S\ v \in S\ u \rightarrow^*_G v$
shows $u \rightarrow^*_{G \restriction S} v$
⟨*proof*⟩

lemma *strongly-connected*:
shows *strongly-connected* (*G* ↾ *S*)
⟨*proof*⟩

lemma *induced-in-sccs*: $G \restriction S \in$ *sccs*
⟨*proof*⟩
end

context *wf-digraph* **begin**

lemma *in-verts-sccsD-sccs*:
assumes $S \in$ *sccs-verts*
shows $G \restriction S \in$ *sccs*
⟨*proof*⟩

lemma *sccs-verts-conv*: *sccs-verts* = *verts* ‘ *sccs*
⟨*proof*⟩

lemma *induce-eq-iff-induced*:
assumes *induced-subgraph H G* **shows** *G* ↾ *verts H* = *H*
⟨*proof*⟩

lemma *sccs-conv-sccs-verts*: *sccs* = *induce-subgraph G* ‘ *sccs-verts*
⟨*proof*⟩

end

lemma *connected-conv*:
shows *connected G* ⟷ *verts G* ≠ {} ∧ (∀ *u* ∈ *verts G*. ∀ *v* ∈ *verts G*. (*u*,*v*) ∈ *rtrancl-on* (*verts G*) ((*arcs-ends G*)^s))
⟨*proof*⟩

lemma (**in** *wf-digraph*) *symmetric-connected-imp-strongly-connected*:
assumes *symmetric G connected G*
shows *strongly-connected G*
⟨*proof*⟩

lemma (**in** *wf-digraph*) *connected-spanning-imp-connected*:
assumes *spanning H G*
assumes *connected H*
shows *connected G*
⟨*proof*⟩

lemma (**in** *wf-digraph*) *spanning-tree-imp-connected*:
assumes *spanning-tree H G*
shows *connected G*
⟨*proof*⟩

term *LEAST x*. *P x*

lemma (**in** *sym-digraph*) *induce-reachable-is-in-sccs*:
assumes *u* ∈ *verts G*
shows (*G* ↾ {*v*. *u* →* *v*}) ∈ *sccs*
⟨*proof*⟩

lemma *induced-eq-verts-imp-eq*:
assumes *induced-subgraph G H*
assumes *induced-subgraph G′ H*
assumes *verts G* = *verts G′*
shows *G* = *G′*
⟨*proof*⟩

lemma (**in** *pre-digraph*) *in-sccs-subset-imp-eq*:
assumes $c \in sccs$
assumes $d \in sccs$
assumes $verts\ c \subseteq verts\ d$
shows $c = d$
⟨*proof*⟩

context *wf-digraph* **begin**

lemma *connectedI*:
assumes $verts\ G \neq \{\}$ $\bigwedge u\ v.\ u \in verts\ G \Longrightarrow v \in verts\ G \Longrightarrow u \rightarrow^*_{mk\text{-}symmetric\ G} v$
shows *connected G*
⟨*proof*⟩

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*⟩

lemma *inj-on-verts-sccs*: *inj-on verts sccs*
⟨*proof*⟩

lemma *card-sccs-verts*: *card sccs-verts* = *card sccs*
⟨*proof*⟩

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*⟩

context *wf-digraph* **begin**

lemma *scc-disj*:
assumes *scc*: $c \in sccs\ d \in sccs$
assumes $c \neq d$
shows $verts\ c \cap verts\ d = \{\}$
⟨*proof*⟩

lemma *in-sccs-verts-conv*:
$S \in sccs\text{-}verts \longleftrightarrow G \restriction S \in sccs$
⟨*proof*⟩

end

lemma (**in** *wf-digraph*) *in-scc-of-self*: $u \in \mathit{verts}\ G \Longrightarrow u \in \mathit{scc\text{-}of}\ u$
⟨*proof*⟩

lemma (**in** *wf-digraph*) *scc-of-empty-conv*: $\mathit{scc\text{-}of}\ u = \{\} \longleftrightarrow u \notin \mathit{verts}\ G$
⟨*proof*⟩

lemma (**in** *wf-digraph*) *scc-of-in-sccs-verts*:
assumes $u \in \mathit{verts}\ G$ **shows** $\mathit{scc\text{-}of}\ u \in \mathit{sccs\text{-}verts}$
⟨*proof*⟩

lemma (**in** *wf-digraph*) *sccs-verts-subsets*: $S \in \mathit{sccs\text{-}verts} \Longrightarrow S \subseteq \mathit{verts}\ G$
⟨*proof*⟩

lemma (**in** *fin-digraph*) *finite-sccs-verts*: *finite sccs-verts*
⟨*proof*⟩

lemma (**in** *wf-digraph*) *sccs-verts-conv-scc-of*:
$\mathit{sccs\text{-}verts} = \mathit{scc\text{-}of}\ `\ \mathit{verts}\ G$ (**is** *?L* = *?R*)
⟨*proof*⟩

lemma (**in** *sym-digraph*) *scc-ofI-reachable*:
assumes $u \rightarrow^* v$ **shows** $u \in \mathit{scc\text{-}of}\ v$
⟨*proof*⟩

lemma (**in** *sym-digraph*) *scc-ofI-reachable′*:
assumes $v \rightarrow^* u$ **shows** $u \in \mathit{scc\text{-}of}\ v$
⟨*proof*⟩

lemma (**in** *sym-digraph*) *scc-ofI-awalk*:
assumes $\mathit{awalk}\ u\ p\ v$ **shows** $u \in \mathit{scc\text{-}of}\ v$
⟨*proof*⟩

lemma (**in** *sym-digraph*) *scc-ofI-apath*:
assumes $\mathit{apath}\ u\ p\ v$ **shows** $u \in \mathit{scc\text{-}of}\ v$
⟨*proof*⟩

lemma (**in** *wf-digraph*) *scc-of-eq*: $u \in \mathit{scc\text{-}of}\ v \Longrightarrow \mathit{scc\text{-}of}\ u = \mathit{scc\text{-}of}\ v$
⟨*proof*⟩

lemma (**in** *wf-digraph*) *strongly-connected-eq-iff*:
$\mathit{strongly\text{-}connected}\ G \longleftrightarrow \mathit{sccs} = \{G\}$ (**is** *?L* ⟷ *?R*)
⟨*proof*⟩

8.9 Components

lemma (**in** *sym-digraph*) *exists-scc*:
assumes $\mathit{verts}\ G \neq \{\}$ **shows** $\exists\, c.\ c \in \mathit{sccs}$
⟨*proof*⟩

theorem (**in** *sym-digraph*) *graph-is-union-sccs*:
shows *Union sccs* = *G*
⟨*proof*⟩

lemma (**in** *sym-digraph*) *scc-for-vert-ex*:
assumes *u* ∈ *verts G*
shows ∃ *c*. *c*∈*sccs* ∧ *u* ∈ *verts c*
⟨*proof*⟩

lemma (**in** *sym-digraph*) *scc-decomp-unique*:
assumes *S* ⊆ *sccs verts* (*Union S*) = *verts G* **shows** *S* = *sccs*
⟨*proof*⟩

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* ⇒ (*'a* × *'a*) *list* **where**
vwalk-arcs [] = []
| *vwalk-arcs* [*x*] = []
| *vwalk-arcs* (*x*#*y*#*xs*) = (*x*,*y*) # *vwalk-arcs* (*y*#*xs*)

definition *vwalk-length* :: *'a vwalk* ⇒ *nat* **where**
vwalk-length p ≡ *length* (*vwalk-arcs p*)

lemma *vwalk-length-simp*[*simp*]:
shows *vwalk-length p* = *length p* − *1*
⟨*proof*⟩

definition *vwalk* :: *'a vwalk* ⇒ (*'a*,*'b*) *pre-digraph* ⇒ *bool* **where**
vwalk p G ≡ *set p* ⊆ *verts G* ∧ *set* (*vwalk-arcs p*) ⊆ *arcs-ends G* ∧ *p* ≠ []

definition *vpath* :: *'a vwalk* ⇒ (*'a*,*'b*) *pre-digraph* ⇒ *bool* **where**
vpath p G ≡ *vwalk p G* ∧ *distinct p*

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

function *vwalk-to-vpath* :: *'a vwalk* ⇒ *'a vwalk* **where**
  vwalk-to-vpath [] = []
| *vwalk-to-vpath* (*x* # *xs*) = (*if* (*x* ∈ *set xs*)
    then vwalk-to-vpath (*dropWhile* (λ*y*. *y* ≠ *x*) *xs*)
    else x # *vwalk-to-vpath xs*)
⟨*proof*⟩
termination ⟨*proof*⟩

lemma *vwalkI*[*intro*]:
  assumes *set p* ⊆ *verts G*
  assumes *set* (*vwalk-arcs p*) ⊆ *arcs-ends G*
  assumes *p* ≠ []
  shows *vwalk p G*
⟨*proof*⟩

lemma *vwalkE*[*elim*]:
  assumes *vwalk p G*
  assumes *set p* ⊆ *verts G* ⟹
    set (*vwalk-arcs p*) ⊆ *arcs-ends G* ∧ *p* ≠ [] ⟹ *P*
  shows *P*
⟨*proof*⟩

lemma *vpathI*[*intro*]:
  assumes *vwalk p G*
  assumes *distinct p*
  shows *vpath p G*
⟨*proof*⟩

lemma *vpathE*[*elim*]:
  assumes *vpath p G*
  assumes *vwalk p G* ⟹ *distinct p* ⟹ *P*
  shows *P*
⟨*proof*⟩

lemma *vwalk-consI*:
  assumes *vwalk p G*
  assumes *a* ∈ *verts G*
  assumes (*a*, *hd p*) ∈ *arcs-ends G*
  shows *vwalk* (*a* # *p*) *G*
⟨*proof*⟩

lemma *vwalk-consE*:
  assumes *vwalk* (*a* # *p*) *G*
  assumes *p* ≠ []
  assumes (*a*, *hd p*) ∈ *arcs-ends G* ⟹ *vwalk p G* ⟹ *P*
  shows *P*

⟨*proof*⟩

lemma *vwalk-induct*[*case-names Base Cons*, *induct pred*: *vwalk*]:
assumes *vwalk p G*
assumes $\bigwedge u.\ u \in verts\ G \Longrightarrow P\ [u]$
assumes $\bigwedge u\ v\ es.\ (u,v) \in arcs\text{-}ends\ G \Longrightarrow P\ (v\ \#\ es) \Longrightarrow P\ (u\ \#\ v\ \#\ es)$
shows *P p*
⟨*proof*⟩

lemma *vwalk-arcs-Cons*[*simp*]:
assumes $p \neq []$
shows *vwalk-arcs* $(u\ \#\ p) = (u,\ hd\ p)\ \#$ *vwalk-arcs p*
⟨*proof*⟩

lemma *vwalk-arcs-append*:
assumes $p \neq []$ **and** $q \neq []$
shows *vwalk-arcs* $(p\ @\ q) =$ *vwalk-arcs p* $@$ $(last\ p,\ hd\ q)\ \#$ *vwalk-arcs q*
⟨*proof*⟩

lemma *set-vwalk-arcs-append1*:
$set\ (vwalk\text{-}arcs\ p) \subseteq set\ (vwalk\text{-}arcs\ (p\ @\ q))$
⟨*proof*⟩

lemma *set-vwalk-arcs-append2*:
$set\ (vwalk\text{-}arcs\ q) \subseteq set\ (vwalk\text{-}arcs\ (p\ @\ q))$
⟨*proof*⟩

lemma *set-vwalk-arcs-cons*:
$set\ (vwalk\text{-}arcs\ p) \subseteq set\ (vwalk\text{-}arcs\ (u\ \#\ p))$
⟨*proof*⟩

lemma *set-vwalk-arcs-snoc*:
assumes $p \neq []$
shows $set\ (vwalk\text{-}arcs\ (p\ @\ [a]))$
$= insert\ (last\ p,\ a)\ (set\ (vwalk\text{-}arcs\ p))$
⟨*proof*⟩

lemma (**in** *wf-digraph*) *vwalk-wf-digraph-consI*:
assumes *vwalk p G*
assumes $(a,\ hd\ p) \in arcs\text{-}ends\ G$
shows $vwalk\ (a\ \#\ p)\ G$
⟨*proof*⟩

lemma *vwalkI-append-l*:
assumes $p \neq []$
assumes $vwalk\ (p\ @\ q)\ G$
shows *vwalk p G*
⟨*proof*⟩

lemma *vwalkI-append-r*:
assumes $q \neq []$
assumes *vwalk* $(p\ @\ q)$ G
shows *vwalk* q G
⟨*proof*⟩

lemma *vwalk-to-vpath-hd*: $hd\ (vwalk\text{-}to\text{-}vpath\ xs) = hd\ xs$
⟨*proof*⟩

lemma *vwalk-to-vpath-induct3*[*consumes 0*, *case-names base in-set not-in-set*]:
assumes $P\ []$
assumes $\bigwedge x\ xs.\ x \in set\ xs \Longrightarrow P\ (dropWhile\ (\lambda y.\ y \neq x)\ xs)$
$\Longrightarrow P\ (x\ \#\ xs)$
assumes $\bigwedge x\ xs.\ x \notin set\ xs \Longrightarrow P\ xs \Longrightarrow P\ (x\ \#\ xs)$
shows $P\ xs$
⟨*proof*⟩

lemma *vwalk-to-vpath-nonempty*:
assumes $p \neq []$
shows $vwalk\text{-}to\text{-}vpath\ p \neq []$
⟨*proof*⟩

lemma *vwalk-to-vpath-last*:
$last\ (vwalk\text{-}to\text{-}vpath\ xs) = last\ xs$
⟨*proof*⟩

lemma *vwalk-to-vpath-subset*:
assumes $x \in set\ (vwalk\text{-}to\text{-}vpath\ xs)$
shows $x \in set\ xs$
⟨*proof*⟩

lemma *vwalk-to-vpath-cons*:
assumes $x \notin set\ xs$
shows $vwalk\text{-}to\text{-}vpath\ (x\ \#\ xs) = x\ \#\ vwalk\text{-}to\text{-}vpath\ xs$
⟨*proof*⟩

lemma *vwalk-to-vpath-vpath*:
assumes *vwalk* p G
shows *vpath* $(vwalk\text{-}to\text{-}vpath\ p)$ G
⟨*proof*⟩

lemma *vwalk-imp-ex-vpath*:
assumes *vwalk* p G
assumes $hd\ p = u$
assumes $last\ p = v$
shows $\exists q.\ vpath\ q\ G \wedge hd\ q = u \wedge last\ q = v$
⟨*proof*⟩

lemma *vwalk-arcs-set-nil*:
assumes $x \in set\ (vwalk\text{-}arcs\ p)$
shows $p \neq []$
⟨*proof*⟩

lemma *in-set-vwalk-arcs-append1*:
assumes $x \in set\ (vwalk\text{-}arcs\ p) \vee x \in set\ (vwalk\text{-}arcs\ q)$
shows $x \in set\ (vwalk\text{-}arcs\ (p\ @\ q))$
⟨*proof*⟩

lemma *in-set-vwalk-arcs-append2*:
assumes *nonempty*: $p \neq []\ q \neq []$
assumes *disj*: $x \in set\ (vwalk\text{-}arcs\ p) \vee x = (last\ p,\ hd\ q)$
$\vee\ x \in set\ (vwalk\text{-}arcs\ q)$
shows $x \in set\ (vwalk\text{-}arcs\ (p\ @\ q))$
⟨*proof*⟩

lemma *arcs-in-vwalk-arcs*:
assumes $u \in set\ (vwalk\text{-}arcs\ p)$
shows $u \in set\ p \times set\ p$
⟨*proof*⟩

lemma *set-vwalk-arcs-rev*:
$set\ (vwalk\text{-}arcs\ (rev\ p)) = \{(v,\ u).\ (u,v) \in set\ (vwalk\text{-}arcs\ p)\}$
⟨*proof*⟩

lemma *vpath-self*:
assumes $u \in verts\ G$
shows $vpath\ [u]\ G$
⟨*proof*⟩

lemma *vwalk-verts-in-verts*:
assumes $vwalk\ p\ G$
assumes $u \in set\ p$
shows $u \in verts\ G$
⟨*proof*⟩

lemma *vwalk-arcs-tl*:
$vwalk\text{-}arcs\ (tl\ xs) = tl\ (vwalk\text{-}arcs\ xs)$
⟨*proof*⟩

lemma *vwalk-arcs-butlast*:
$vwalk\text{-}arcs\ (butlast\ xs) = butlast\ (vwalk\text{-}arcs\ xs)$
⟨*proof*⟩

lemma *vwalk-arcs-tl-empty*:
$vwalk\text{-}arcs\ xs = [] \Longrightarrow vwalk\text{-}arcs\ (tl\ xs) = []$
⟨*proof*⟩

lemma *vwalk-arcs-butlast-empty*:
xs ≠ [] ⟹ *vwalk-arcs xs* = [] ⟹ *vwalk-arcs* (*butlast xs*) = []
⟨*proof*⟩

definition *joinable* :: *'a vwalk* ⇒ *'a vwalk* ⇒ *bool* **where**
joinable p q ≡ *last p* = *hd q* ∧ *p* ≠ [] ∧ *q* ≠ []

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

lemma *joinable-Nil-l-iff*[*simp*]: *joinable* [] *p* = *False*
and *joinable-Nil-r-iff*[*simp*]: *joinable q* [] = *False*
⟨*proof*⟩

lemma *joinable-Cons-l-iff*[*simp*]: *p* ≠ [] ⟹ *joinable* (*v* # *p*) *q* = *joinable p q*
and *joinable-Snoc-r-iff*[*simp*]: *q* ≠ [] ⟹ *joinable p* (*q* @ [*v*]) = *joinable p q*
⟨*proof*⟩

lemma *joinableI*[*intro*,*simp*]:
assumes *last p* = *hd q p* ≠ [] *q* ≠ []
shows *joinable p q*
⟨*proof*⟩

lemma *vwalk-join-non-Nil*[*simp*]:
assumes *p* ≠ []
shows *p* ⊕ *q* ≠ []
⟨*proof*⟩

lemma *vwalk-join-Cons*[*simp*]:
assumes *p* ≠ []
shows (*u* # *p*) ⊕ *q* = *u* # *p* ⊕ *q*
⟨*proof*⟩

lemma *vwalk-join-def2*:
assumes *joinable p q*
shows *p* ⊕ *q* = *butlast p* @ *q*
⟨*proof*⟩

lemma *vwalk-join-hd'*:
assumes *p* ≠ []
shows *hd* (*p* ⊕ *q*) = *hd p*
⟨*proof*⟩

lemma *vwalk-join-hd*:
assumes *joinable p q*
shows *hd* (*p* ⊕ *q*) = *hd p*
⟨*proof*⟩

lemma *vwalk-join-last*:
assumes *joinable p q*
shows *last* $(p \oplus q) =$ *last q*
⟨*proof*⟩

lemma *vwalk-join-Nil*[*simp*]:
$p \oplus [] = p$
⟨*proof*⟩

lemma *vwalk-joinI-vwalk'*:
assumes *vwalk p G*
assumes *vwalk q G*
assumes *last p* = *hd q*
shows *vwalk* $(p \oplus q)$ *G*
⟨*proof*⟩

lemma *vwalk-joinI-vwalk*:
assumes *vwalk p G*
assumes *vwalk q G*
assumes *joinable p q*
shows *vwalk* $(p \oplus q)$ *G*
⟨*proof*⟩

lemma *vwalk-join-split*:
assumes $u \in$ *set p*
shows $\exists\, q\ r.\ p = q \oplus r$
$\wedge$ *last q* = $u \wedge$ *hd r* = $u \wedge q \neq [] \wedge r \neq []$
⟨*proof*⟩

lemma *vwalkI-vwalk-join-l*:
assumes $p \neq []$
assumes *vwalk* $(p \oplus q)$ *G*
shows *vwalk p G*
⟨*proof*⟩

lemma *vwalkI-vwalk-join-r*:
assumes *joinable p q*
assumes *vwalk* $(p \oplus q)$ *G*
shows *vwalk q G*
⟨*proof*⟩

lemma *vwalk-join-assoc'*:
assumes $p \neq [] \ q \neq []$
shows $(p \oplus q) \oplus r = p \oplus q \oplus r$
⟨*proof*⟩

lemma *vwalk-join-assoc*:
assumes *joinable p q joinable q r*

shows $(p \oplus q) \oplus r = p \oplus q \oplus r$
⟨*proof*⟩

lemma *joinable-vwalk-join-r-iff*:
joinable p $(q \oplus r) \longleftrightarrow$ *joinable p q* $\vee$ $(q = [] \wedge$ *joinable p* $(tl\ r))$
⟨*proof*⟩

lemma *joinable-vwalk-join-l-iff*:
assumes *joinable p q*
shows *joinable* $(p \oplus q)\ r \longleftrightarrow$ *joinable q r* (**is** *?L* $\longleftrightarrow$ *?R*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *joinable-non-Nil*:
assumes *joinable p q*
shows $p \neq [] \ q \neq []$
⟨*proof*⟩

lemma *vwalk-join-vwalk-length*[*simp*]:
assumes *joinable p q*
shows *vwalk-length* $(p \oplus q)$ = *vwalk-length p* + *vwalk-length q*
⟨*proof*⟩

lemma *vwalk-join-arcs*:
assumes *joinable p q*
shows *vwalk-arcs* $(p \oplus q)$ = *vwalk-arcs p* @ *vwalk-arcs q*
⟨*proof*⟩

lemma *vwalk-join-arcs1*:
assumes *set* (*vwalk-arcs p*) $\subseteq E$
assumes $p = q \oplus r$
shows *set* (*vwalk-arcs q*) $\subseteq E$
⟨*proof*⟩

lemma *vwalk-join-arcs2*:
assumes *set* (*vwalk-arcs p*) $\subseteq E$
assumes *joinable q r*
assumes $p = q \oplus r$
shows *set* (*vwalk-arcs r*) $\subseteq E$
⟨*proof*⟩

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

lemma *concat-vpath-is-vpath*:
assumes *p-props*: *vwalk p G hd p* = *u last p* = *v*
assumes *q-props*: *vwalk q G hd q* = *v last q* = *w*
shows *vpath* (*concat-vpath p q*) *G* ∧ *hd* (*concat-vpath p q*) = *u*
∧ *last* (*concat-vpath p q*) = *w*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma (**in** *fin-digraph*) *vpaths-finite*:
shows *finite* {*p*. *vpath p G*}
⟨*proof*⟩

lemma (**in** *wf-digraph*) *reachable-vwalk-conv*:
u →**G v* ⟷ (∃ *p*. *vwalk p G* ∧ *hd p* = *u* ∧ *last p* = *v*) (**is** *?L* ⟷ *?R*)
⟨*proof*⟩

lemma (**in** *wf-digraph*) *reachable-vpath-conv*:
u →**G v* ⟷ (∃ *p*. *vpath p G* ∧ *hd p* = *u* ∧ *last p* = *v*) (**is** *?L* ⟷ *?R*)
⟨*proof*⟩

lemma *in-set-vwalk-arcsE*:
assumes (*u*,*v*) ∈ *set* (*vwalk-arcs p*)
obtains *u* ∈ *set p v* ∈ *set p*
⟨*proof*⟩

lemma *vwalk-rev-ex*:
assumes *symmetric G*
assumes *vwalk p G*
shows *vwalk* (*rev p*) *G*
⟨*proof*⟩

lemma *vwalk-singleton*[*simp*]: *vwalk* [*u*] *G* = (*u* ∈ *verts G*)
⟨*proof*⟩

lemma (**in** *wf-digraph*) *vwalk-Cons-Cons*[*simp*]:
vwalk (*u* # *v* # *ws*) *G* = ((*u*,*v*) ∈ *arcs-ends G* ∧ *vwalk* (*v* # *ws*) *G*)
⟨*proof*⟩

lemma (**in** *wf-digraph*) *awalk-imp-vwalk*:
assumes *awalk u p v* **shows** *vwalk* (*awalk-verts u p*) *G*

⟨*proof*⟩

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*⟩

lemma *vpathI-subgraph*:
assumes *vpath p G*
assumes *subgraph G H*
shows *vpath p H*
⟨*proof*⟩

lemma (**in** *loopfree-digraph*) *vpathI-arc*:
assumes (*a*,*b*) ∈ *arcs-ends G*
shows *vpath* [*a*,*b*] *G*
⟨*proof*⟩

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

11 Isomorphisms of Digraphs

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

definition (**in** *pre-digraph*) *digraph-isomorphism* :: ('*a*,'*b*,'*aa*,'*bb*) *digraph-isomorphism*
⇒ *bool* **where**
digraph-isomorphism hom ≡

wf-digraph G ∧
inj-on (*iso-verts hom*) (*verts G*) ∧
inj-on (*iso-arcs hom*) (*arcs G*) ∧
(∀ *a* ∈ *arcs G*.
iso-verts hom (*tail G a*) = *iso-tail hom* (*iso-arcs hom a*) ∧
iso-verts hom (*head G a*) = *iso-head hom* (*iso-arcs hom a*))

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

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

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

lemma *verts-app-iso*: *verts* (*app-iso hom G*) = *iso-verts hom* ‘ *verts G*
and *arcs-app-iso*: *arcs* (*app-iso hom G*) = *iso-arcs hom* ‘*arcs G*
and *tail-app-iso*: *tail* (*app-iso hom G*) = *iso-tail hom*
and *head-app-iso*: *head* (*app-iso hom G*) = *iso-head hom*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *digraph-isomorphism-inj-on-arcs*:
digraph-isomorphism hom ⟹ *inj-on* (*iso-arcs hom*) (*arcs G*)
⟨*proof*⟩

lemma *digraph-isomorphism-inj-on-verts*:
digraph-isomorphism hom ⟹ *inj-on* (*iso-verts hom*) (*verts G*)
⟨*proof*⟩

end

lemma (**in** *wf-digraph*) *wf-digraphI-app-iso*[*intro?*]:
assumes *digraph-isomorphism hom*
shows *wf-digraph* (*app-iso hom G*)
⟨*proof*⟩

lemma (**in** *fin-digraph*) *fin-digraphI-app-iso*[*intro?*]:
assumes *digraph-isomorphism hom*
shows *fin-digraph* (*app-iso hom G*)

⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*)
⟨*proof*⟩

lemma *arcs-ends-app-iso-eq*:
assumes *digraph-isomorphism hom*
shows *arcs-ends* (*app-iso hom G*) = (λ(*u*,*v*). (*iso-verts hom u*, *iso-verts hom v*)) ‘ *arcs-ends G*
⟨*proof*⟩

lemma *in-arcs-app-iso-eq*:
assumes *digraph-isomorphism hom* **and** *u* ∈ *verts G*
shows *in-arcs* (*app-iso hom G*) (*iso-verts hom u*) = *iso-arcs hom* ‘ *in-arcs G u*
⟨*proof*⟩

lemma *out-arcs-app-iso-eq*:
assumes *digraph-isomorphism hom* **and** *u* ∈ *verts G*
shows *out-arcs* (*app-iso hom G*) (*iso-verts hom u*) = *iso-arcs hom* ‘ *out-arcs G u*
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

lemma *in-arcs-app-iso-eq′*:
assumes *digraph-isomorphism hom* **and** $u \in$ *verts* (*app-iso hom* G)
shows *in-arcs* (*app-iso hom* G) u = *iso-arcs hom* ‘ *in-arcs* G (*iso-verts* (*inv-iso hom*) u)
⟨*proof*⟩

lemma *out-arcs-app-iso-eq′*:
assumes *digraph-isomorphism hom* **and** $u \in$ *verts* (*app-iso hom* G)
shows *out-arcs* (*app-iso hom* G) u = *iso-arcs hom* ‘ *out-arcs* G (*iso-verts* (*inv-iso hom*) u)
⟨*proof*⟩

lemma *in-degree-app-iso-eq′*:
assumes *digraph-isomorphism hom* **and** $u \in$ *verts* (*app-iso hom* G)
shows *in-degree* (*app-iso hom* G) u = *in-degree* G (*iso-verts* (*inv-iso hom*) u)
⟨*proof*⟩

lemma *out-degree-app-iso-eq′*:
assumes *digraph-isomorphism hom* **and** $u \in$ *verts* (*app-iso hom* G)
shows *out-degree* (*app-iso hom* G) u = *out-degree* G (*iso-verts* (*inv-iso hom*) u)
⟨*proof*⟩

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

lemma *reachableI-app-iso*:
assumes *r*: $u \rightarrow^* v$ **and** *hom*: *digraph-isomorphism hom*
shows (*iso-verts hom* u) $\rightarrow^*_{\textit{app-iso hom } G}$ (*iso-verts hom* v)
⟨*proof*⟩

lemma *awalk-app-iso-eq*:
assumes *hom*: *digraph-isomorphism hom*
assumes $u \in$ *iso-verts hom* ‘ *verts* G $v \in$ *iso-verts hom* ‘ *verts* G *set* $p \subseteq$ *iso-arcs hom* ‘ *arcs* G

shows *pre-digraph.awalk* (*app-iso hom G*) *u p v*
⟷ *awalk* (*iso-verts* (*inv-iso hom*) *u*) (*map* (*iso-arcs* (*inv-iso hom*)) *p*) (*iso-verts* (*inv-iso hom*) *v*)
⟨*proof*⟩

lemma *reachable-app-iso-eq*:
assumes *hom*: *digraph-isomorphism hom*
assumes *u* ∈ *iso-verts hom* ‘ *verts G v* ∈ *iso-verts hom* ‘ *verts G*
shows *u* →**app-iso hom G v* ⟷ *iso-verts* (*inv-iso hom*) *u* →* *iso-verts* (*inv-iso hom*) *v* (**is** *?L* ⟷ *?R*)
⟨*proof*⟩

lemma *connectedI-app-iso*:
assumes *c*: *connected G* **and** *hom*: *digraph-isomorphism hom*
shows *connected* (*app-iso hom G*)
⟨*proof*⟩

end

lemma *digraph-iso-swap*:
assumes *wf-digraph G digraph-iso G H* **shows** *digraph-iso H G*
⟨*proof*⟩

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

lemma *digraph-iso-trans*[*trans*]:
assumes *digraph-iso G H digraph-iso H I* **shows** *digraph-iso G I*
⟨*proof*⟩

lemma (**in** *pre-digraph*) *digraph-isomorphism-subgraphI*:
assumes *digraph-isomorphism hom*
assumes *subgraph H G*
shows *pre-digraph.digraph-isomorphism H hom*
⟨*proof*⟩

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*⟩

lemma (**in** *wf-digraph*) *arcs-app-inv-iso-subgraph*:
assumes *hom*: *digraph-isomorphism hom* **and** $A \subseteq$ *arcs G*
shows *iso-arcs* (*inv-iso hom*) ‘ *iso-arcs hom* ‘ $A = A$
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*)
⟨*proof*⟩

lemma (**in** *pre-digraph*) *app-iso-eq-conv*:
assumes *digraph-isomorphism hom*
assumes *subgraph H1 G subgraph H2 G*
shows *app-iso hom H1* = *app-iso hom H2* ⟷ *H1* = *H2* (**is** *?L* ⟷ *?R*)
⟨*proof*⟩

lemma *in-arcs-app-iso-cases*:
assumes *a* ∈ *arcs* (*app-iso hom G*)
obtains *a0* **where** *a* = *iso-arcs hom a0 a0* ∈ *arcs G*
⟨*proof*⟩

lemma *in-verts-app-iso-cases*:
assumes *v* ∈ *verts* (*app-iso hom G*)
obtains *v0* **where** *v* = *iso-verts hom v0 v0* ∈ *verts G*
⟨*proof*⟩

lemma (**in** *wf-digraph*) *max-subgraph-iso*:
assumes *hom*: *digraph-isomorphism hom*

assumes *subg*: *subgraph H* (*app-iso hom G*)
shows *pre-digraph.max-subgraph* (*app-iso hom G*) *P H*
⟷ *max-subgraph* (*P o app-iso hom*) (*app-iso* (*inv-iso hom*) *H*)
⟨*proof*⟩

lemma (**in** *pre-digraph*) *max-subgraph-cong*:
assumes *H* = *H'* ⋀*H''*. *subgraph H' H''* ⟹ *subgraph H'' G* ⟹ *P H''* = *P' H''*
shows *max-subgraph P H* = *max-subgraph P' H'*
⟨*proof*⟩

lemma (**in** *pre-digraph*) *inj-on-app-iso*:
assumes *hom*: *digraph-isomorphism hom*
assumes *S* ⊆ {*H*. *subgraph H G*}
shows *inj-on* (*app-iso hom*) *S*
⟨*proof*⟩

11.1 Graph Invariants

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

interpretation *wf-digraph G* ⟨*proof*⟩

lemma *card-verts-iso*[*simp*]: *card* (*iso-verts hom* ‘ *verts G*) = *card* (*verts G*)
⟨*proof*⟩

lemma *card-arcs-iso*[*simp*]: *card* (*iso-arcs hom* ‘ *arcs G*) = *card* (*arcs G*)
⟨*proof*⟩

lemma *strongly-connected-iso*[*simp*]: *strongly-connected* (*app-iso hom G*) ⟷ *strongly-connected G*
⟨*proof*⟩

lemma *subgraph-strongly-connected-iso*:
assumes *subgraph H G*
shows *strongly-connected* (*app-iso hom H*) ⟷ *strongly-connected H*
⟨*proof*⟩

lemma *sccs-iso*[*simp*]: *pre-digraph.sccs* (*app-iso hom G*) = *app-iso hom* ‘ *sccs* (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *card-sccs-iso*[*simp*]: *card* (*app-iso hom* ‘ *sccs*) = *card sccs*
⟨*proof*⟩

end

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

lemma *funpow-invs*:
assumes $m \leq n$ **and** *inv*: $\bigwedge x.\ f\ (g\ x) = x$
shows $(f \frown\frown m)\ ((g \frown\frown n)\ x) = (g \frown\frown (n - m))\ x$
⟨*proof*⟩

12 Permutation Domains

definition *has-dom* :: $('a \Rightarrow 'a) \Rightarrow 'a\ set \Rightarrow bool$ **where**
has-dom f S ≡ $\forall s.\ s \notin S \longrightarrow f\ s = s$

lemma *has-domD*: *has-dom f S* $\Longrightarrow x \notin S \Longrightarrow f\ x = x$
⟨*proof*⟩

lemma *has-domI*: $(\bigwedge x.\ x \notin S \Longrightarrow f\ x = x) \Longrightarrow$ *has-dom f S*
⟨*proof*⟩

lemma *permutes-conv-has-dom*:
f permutes S $\longleftrightarrow$ *bij f* $\wedge$ *has-dom f S*
⟨*proof*⟩

13 Segments

inductive-set *segment* :: $('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a\ set$ **for** *f a b* **where**
base: *f a* $\neq$ *b* $\Longrightarrow$ *f a* $\in$ *segment f a b* |
step: *x* $\in$ *segment f a b* $\Longrightarrow$ *f x* $\neq$ *b* $\Longrightarrow$ *f x* $\in$ *segment f a b*

lemma *segment-step-2D*:
assumes *x* $\in$ *segment f a* (*f b*) **shows** *x* $\in$ *segment f a b* $\vee$ *x* = *b*
⟨*proof*⟩

lemma *not-in-segment2D*:
assumes *x* $\in$ *segment f a b* **shows** *x* $\neq$ *b*
⟨*proof*⟩

lemma *segment-altdef*:
assumes *b* $\in$ *orbit f a*
shows *segment f a b* = $(\lambda n.\ (f \frown\frown n)\ a)$ ' {*1..<funpow-dist1 f a b*} (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *segmentD-orbit*:

assumes $x \in$ *segment f y z* **shows** $x \in$ *orbit f y*
⟨*proof*⟩

lemma *segment1-empty*: *segment f x* (*f x*) = {}
⟨*proof*⟩

lemma *segment-subset*:
assumes $y \in$ *segment f x z*
assumes $w \in$ *segment f x y*
shows $w \in$ *segment f x z*
⟨*proof*⟩

lemma *not-in-segment1*:
assumes $y \in$ *orbit f x* **shows** $x \notin$ *segment f x y*
⟨*proof*⟩

lemma *not-in-segment2*: $y \notin$ *segment f x y*
⟨*proof*⟩

lemma *in-segmentE*:
assumes $y \in$ *segment f x z z* $\in$ *orbit f x*
obtains (*f* ^^ *funpow-dist1 f x y*) *x* = *y funpow-dist1 f x y* < *funpow-dist1 f x z*
⟨*proof*⟩

lemma *cyclic-split-segment*:
assumes *S*: *cyclic-on f S a* ∈ *S b* ∈ *S* **and** *a* ≠ *b*
shows *S* = {*a*,*b*} ∪ *segment f a b* ∪ *segment f b a* (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *segment-split*:
assumes *y-in-seg*: $y \in$ *segment f x z*
shows *segment f x z* = *segment f x y* ∪ {*y*} ∪ *segment f y z* (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *in-segmentD-inv*:
assumes $x \in$ *segment f a b x* ≠ *f a*
assumes *inj f*
shows *inv f x* ∈ *segment f a b*
⟨*proof*⟩

lemma *in-orbit-invI*:
assumes $b \in$ *orbit f a*
assumes *inj f*
shows *a* ∈ *orbit* (*inv f*) *b*
⟨*proof*⟩

lemma *segment-step-2*:
assumes *A*: $x \in segment\ f\ a\ b\ b \neq a$ **and** *inj f*
shows $x \in segment\ f\ a\ (f\ b)$
⟨*proof*⟩

lemma *inv-end-in-segment*:
assumes $b \in orbit\ f\ a\ f\ a \neq b$ *bij f*
shows $inv\ f\ b \in segment\ f\ a\ b$
⟨*proof*⟩

lemma *segment-overlapping*:
assumes $x \in orbit\ f\ a\ x \in orbit\ f\ b$ *bij f*
shows $segment\ f\ a\ x \subseteq segment\ f\ b\ x \vee segment\ f\ b\ x \subseteq segment\ f\ a\ x$
⟨*proof*⟩

lemma *segment-disj*:
assumes $a \neq b$ *bij f*
shows $segment\ f\ a\ b \cap segment\ f\ b\ a = \{\}$
⟨*proof*⟩

lemma *segment-x-x-eq*:
assumes *permutation f*
shows $segment\ f\ x\ x = orbit\ f\ x - \{x\}$ (**is** *?L* = *?R*)
⟨*proof*⟩

14 Lists of Powers

definition *iterate* :: $nat \Rightarrow nat \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ list$ **where**
$iterate\ m\ n\ f\ x = map\ (\lambda n.\ (f\widehat{\ }n)\ x)\ [m..<n]$

lemma *set-iterate*:
$set\ (iterate\ m\ n\ f\ x) = (\lambda k.\ (f\ \widehat{\ }\ k)\ x)\ `\ \{m..<n\}$
⟨*proof*⟩

lemma *iterate-empty*[*simp*]: $iterate\ n\ m\ f\ x = [] \longleftrightarrow m \leq n$
⟨*proof*⟩

lemma *iterate-length*[*simp*]:
$length\ (iterate\ m\ n\ f\ x) = n - m$
⟨*proof*⟩

lemma *iterate-nth*[*simp*]:
assumes $k < n - m$ **shows** $iterate\ m\ n\ f\ x\ !\ k = (f\widehat{\ }(m+k))\ x$
⟨*proof*⟩

lemma *iterate-applied*:
$iterate\ n\ m\ f\ (f\ x) = iterate\ (Suc\ n)\ (Suc\ m)\ f\ x$
⟨*proof*⟩

end
theory *Subdivision*
imports
Arc-Walk
Digraph-Component
Pair-Digraph
Bidirected-Digraph
Auxiliary
begin

15 Subdivision on Digraphs

definition
subdivision-step :: ('*a*, '*b*) *pre-digraph* ⇒ ('*b* ⇒ '*b*) ⇒ ('*a*, '*b*) *pre-digraph* ⇒ ('*b* ⇒ '*b*) ⇒ '*a* × '*a* × '*a* ⇒ '*b* × '*b* × '*b* ⇒ *bool*
where
subdivision-step G rev-G H rev-H ≡ λ(*u*, *v*, *w*) (*uv*, *uw*, *vw*).
bidirected-digraph G rev-G
∧ *bidirected-digraph H rev-H*
∧ *perm-restrict rev-H* (*arcs G*) = *perm-restrict rev-G* (*arcs H*)
∧ *compatible G H*

∧ *verts H* = *verts G* ∪ {*w*}
∧ *w* ∉ *verts G*

∧ *arcs H* = {*uw*, *rev-H uw*, *vw*, *rev-H vw*} ∪ *arcs G* − {*uv*, *rev-G uv*}
∧ *uv* ∈ *arcs G*
∧ *distinct* [*uw*, *rev-H uw*, *vw*, *rev-H vw*]
∧ *arc-to-ends G uv* = (*u*,*v*)
∧ *arc-to-ends H uw* = (*u*,*w*)
∧ *arc-to-ends H vw* = (*v*,*w*)

inductive *subdivision* :: ('*a*,'*b*) *pre-digraph* × ('*b* ⇒ '*b*) ⇒ ('*a*,'*b*) *pre-digraph* × ('*b* ⇒ '*b*) ⇒ *bool*
for *biG* **where**
base: *bidirected-digraph* (*fst biG*) (*snd biG*) ⟹ *subdivision biG biG*
| *divide*: ⟦*subdivision biG biI*; *subdivision-step* (*fst biI*) (*snd biI*) (*fst biH*) (*snd biH*) (*u*,*v*,*w*) (*uv*,*uw*,*vw*)⟧ ⟹ *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* ⟹ *P G rev-G*
and ⋀*I rev-I H rev-H u v w uv uw vw*.
subdivision (*G*, *rev-G*) (*I*, *rev-I*) ⟹ *P I rev-I* ⟹ *subdivision-step I rev-I H rev-H* (*u*, *v*, *w*) (*uv*, *uw*, *vw*) ⟹ *P H rev-H*
shows *P H rev-H*
⟨*proof*⟩

lemma *subdivision-base*:
bidirected-digraph G rev-G $\Longrightarrow$ *subdivision* (*G*, *rev-G*) (*G*, *rev-G*)
⟨*proof*⟩

lemma *subdivision-step-rev*:
assumes *subdivision-step G rev-G H rev-H* (*u*, *v*, *w*) (*uv*, *uw*, *vw*) *subdivision* (*H*, *rev-H*) (*I*, *rev-I*)
shows *subdivision* (*G*, *rev-G*) (*I*, *rev-I*)
⟨*proof*⟩

lemma *subdivision-trans*:
assumes *subdivision* (*G*, *rev-G*) (*H*, *rev-H*) *subdivision* (*H*, *rev-H*) (*I*, *rev-I*)
shows *subdivision* (*G*, *rev-G*) (*I*, *rev-I*)
⟨*proof*⟩

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*
⟨*proof*⟩
sublocale *subdiv-step* $\subseteq$ *H*: *bidirected-digraph H rev-H*
⟨*proof*⟩

context *subdiv-step* **begin**

abbreviation (*input*) *vu* $\equiv$ *rev-G uv*
abbreviation (*input*) *wu* $\equiv$ *rev-H uw*
abbreviation (*input*) *wv* $\equiv$ *rev-H vw*

lemma *subdiv-compat*: *compatible G H*
⟨*proof*⟩

lemma *arc-to-ends-eq*: *arc-to-ends H* = *arc-to-ends G*
⟨*proof*⟩

lemma *head-eq*: *head H* = *head G*
⟨*proof*⟩

lemma *tail-eq*: *tail H* = *tail G*
⟨*proof*⟩

lemma *verts-H*: *verts H* = *verts G* $\cup$ {*w*}
⟨*proof*⟩

lemma *verts-G*: *verts G* = *verts H* − {*w*}
⟨*proof*⟩

lemma *arcs-H*: *arcs H* = {*uw*, *wu*, *vw*, *wv*} ∪ *arcs G* − {*uv*, *vu*}
⟨*proof*⟩

lemma *not-in-verts-G*: *w* ∉ *verts G*
⟨*proof*⟩

lemma *in-arcs-G*: {*uv*, *vu*} ⊆ *arcs G*
⟨*proof*⟩

lemma *not-in-arcs-H*: {*uv*,*vu*} ∩ *arcs H* = {}
⟨*proof*⟩

lemma *subdiv-ate*:
arc-to-ends G uv = (*u*,*v*)
arc-to-ends H uv = (*u*,*v*)
arc-to-ends H uw = (*u*,*w*)
arc-to-ends H vw = (*v*,*w*)
⟨*proof*⟩

lemma *subdiv-ends*[*simp*]:
tail G uv = *u head G uv* = *v tail H uv* = *u head H uv* = *v*
tail H uw = *u head H uw* = *w tail H vw* = *v head H vw* = *w*
⟨*proof*⟩

lemma *subdiv-ends-G-rev*[*simp*]:
tail G (*vu*) = *v head G* (*vu*) = *u tail H* (*vu*) = *v head H* (*vu*) = *u*
⟨*proof*⟩

lemma *subdiv-distinct-verts0*: *u* ≠ *w v* ≠ *w*
⟨*proof*⟩

lemma *in-arcs-H*: {*uw*, *wu*, *vw*, *wv*} ⊆ *arcs H*
⟨*proof*⟩

lemma *subdiv-ends-H-rev*[*simp*]:
tail H (*wu*) = *w tail H* (*wv*) = *w*
head H (*wu*) = *u head H* (*wv*) = *v*
⟨*proof*⟩

lemma *in-verts-G*: {*u*,*v*} ⊆ *verts G*
⟨*proof*⟩

lemma *not-in-arcs-G*: {*uw*, *wu*, *vw*, *wv*} ∩ *arcs G* = {}
⟨*proof*⟩

lemma *subdiv-distinct-arcs*: *distinct* [*uv*, *vu*, *uw*, *wu*, *vw*, *wv*]
⟨*proof*⟩

lemma *arcs-G*: *arcs G* = *arcs H* ∪ {*uv*, *vu*} − {*uw*, *wu*, *vw*, *wv*}
⟨*proof*⟩

lemma *subdiv-ate-H-rev*:
arc-to-ends H (*wu*) = (*w*,*u*)
arc-to-ends H (*wv*) = (*w*,*v*)
⟨*proof*⟩

lemma *adj-with-w*: *u* →*H w w* →*H u v* →*H w w* →*H v*
⟨*proof*⟩

lemma *w-reach*: *u* →**H w w* →**H u v* →**H w w* →**H v*
⟨*proof*⟩

lemma *G-reach*: *v* →**G u u* →**G v*
⟨*proof*⟩

lemma *out-arcs-w*: *out-arcs H w* = {*wu*, *wv*}
⟨*proof*⟩

lemma *out-degree-w*: *out-degree H w* = *2*
⟨*proof*⟩

end

lemma *subdivision-compatible*:
assumes *subdivision* (*G*, *rev-G*) (*H*, *rev-H*) **shows** *compatible G H*
⟨*proof*⟩

lemma *subdivision-bidir*:
assumes *subdivision* (*G*, *rev-G*) (*H*, *rev-H*)
shows *bidirected-digraph H rev-H*
⟨*proof*⟩

lemma *subdivision-choose-rev*:
assumes *subdivision* (*G*, *rev-G*) (*H*, *rev-H*) *bidirected-digraph H rev-H′*
shows ∃ *rev-G′*. *subdivision* (*G*, *rev-G′*) (*H*, *rev-H′*)
⟨*proof*⟩

lemma *subdivision-verts-subset*:
assumes *subdivision* (*G*,*rev-G*) (*H*,*rev-H*) *x* ∈ *verts G*
shows *x* ∈ *verts H*
⟨*proof*⟩

15.1 Subdivision on Pair Digraphs

In this section, we introduce specialized rules for pair digraphs.

abbreviation *subdivision-pair G H* ≡ *subdivision* (*with-proj G*, *swap-in* (*parcs G*)) (*with-proj H*, *swap-in* (*parcs H*))

lemma *arc-to-ends-with-proj*[*simp*]: *arc-to-ends* (*with-proj G*) = *id*
⟨*proof*⟩

context
begin

We use the `inductive` command to define an inductive definition pair graphs. This is proven to be equivalent to *subdivision*. This allows us to transfer the rules proven by `inductive` to *subdivision*. To spare the user confusion, we hide this new constant.

private inductive *pair-sd* :: *'a pair-pre-digraph* ⇒ *'a pair-pre-digraph* ⇒ *bool*
for *G* **where**
base: *pair-bidirected-digraph G* ⟹ *pair-sd G G*
| *divide*: ⋀*e w H*. ⟦*e* ∈ *parcs H*; *w* ∉ *pverts H*; *pair-sd G H*⟧
⟹ *pair-sd G* (*subdivide H e w*)

private lemma *bidirected-digraphI-pair-sd*:
assumes *pair-sd G H* **shows** *pair-bidirected-digraph H*
⟨*proof*⟩ **lemma** *subdivision-with-projI*:
assumes *pair-sd G H*
shows *subdivision-pair G H*
⟨*proof*⟩ **lemma** *subdivision-with-projD*:
assumes *subdivision-pair G H*
shows *pair-sd G H*
⟨*proof*⟩ **lemma** *subdivision-pair-conv*:
pair-sd G H = *subdivision-pair G H*
⟨*proof*⟩

lemmas *subdivision-pair-induct* = *pair-sd.induct*[
unfolded subdivision-pair-conv, *case-names base divide*, *induct pred*: *pair-sd*]

lemmas *subdivision-pair-base* = *pair-sd.base*[*unfolded subdivision-pair-conv*]
lemmas *subdivision-pair-divide* = *pair-sd.divide*[*unfolded subdivision-pair-conv*]

lemmas *subdivision-pair-intros* = *pair-sd.intros*[*unfolded subdivision-pair-conv*]
lemmas *subdivision-pair-cases* = *pair-sd.cases*[*unfolded subdivision-pair-conv*]

lemmas *subdivision-pair-simps* = *pair-sd.simps*[*unfolded subdivision-pair-conv*]

lemmas *bidirected-digraphI-subdivision* = *bidirected-digraphI-pair-sd*[*unfolded subdivision-pair-conv*]

end

lemma (**in** *pair-graph*) *pair-graph-subdivision*:
assumes *subdivision-pair G H*
shows *pair-graph H*
⟨*proof*⟩

end

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

16 Euler Trails in Digraphs

In this section we prove the well-known theorem characterizing the existence of an Euler Trail in an directed graph

16.1 Trails and Euler Trails

definition (**in** *pre-digraph*) *euler-trail* :: $'a \Rightarrow 'b\ awalk \Rightarrow 'a \Rightarrow bool$ **where**
euler-trail u p v ≡ *trail u p v* ∧ *set p* = *arcs G* ∧ *set* (*awalk-verts u p*) = *verts G*

context *wf-digraph* **begin**

lemma *finite-distinct*:
assumes *finite A* **shows** *finite* {*p*. *distinct p* ∧ *set p* ⊆ *A*}
⟨*proof*⟩

lemma (**in** *fin-digraph*) *trails-finite*: *finite* {*p*. ∃ *u v*. *trail u p v*}
⟨*proof*⟩

lemma *rotate-awalkE*:
assumes *awalk u p u w* ∈ *set* (*awalk-verts u p*)
obtains *q r* **where** *p* = *q* @ *r awalk w* (*r* @ *q*) *w set* (*awalk-verts w* (*r* @ *q*)) = *set* (*awalk-verts u p*)
⟨*proof*⟩

lemma *rotate-trailE*:
assumes *trail u p u w* ∈ *set* (*awalk-verts u p*)
obtains *q r* **where** *p* = *q* @ *r trail w* (*r* @ *q*) *w set* (*awalk-verts w* (*r* @ *q*)) = *set* (*awalk-verts u p*)
⟨*proof*⟩

lemma *rotate-trailE'*:
assumes *trail u p u w* ∈ *set* (*awalk-verts u p*)
obtains *q* **where** *trail w q w set q* = *set p set* (*awalk-verts w q*) = *set* (*awalk-verts u p*)

⟨*proof*⟩

lemma *sym-reachableI-in-awalk*:
assumes *walk*: *awalk u p v* **and**
w1: *w1* ∈ *set* (*awalk-verts u p*) **and** *w2*: *w2* ∈ *set* (*awalk-verts u p*)
shows *w1* $\rightarrow^*_{\textit{mk-symmetric } G}$ *w2*
⟨*proof*⟩

lemma *euler-imp-connected*:
assumes *euler-trail u p v* **shows** *connected G*
⟨*proof*⟩

end

16.2 Arc Balance of Walks

context *pre-digraph* **begin**

definition *arc-set-balance* :: *'a* ⇒ *'b set* ⇒ *int* **where**
arc-set-balance w A = *int* (*card* (*in-arcs G w* ∩ *A*)) − *int* (*card* (*out-arcs G w* ∩ *A*))

definition *arc-set-balanced* :: *'a* ⇒ *'b set* ⇒ *'a* ⇒ *bool* **where**
arc-set-balanced u A v ≡
if u = *v then* (∀ *w* ∈ *verts G*. *arc-set-balance w A* = *0*)
else (∀ *w* ∈ *verts G*. (*w* ≠ *u* ∧ *w* ≠ *v*) ⟶ *arc-set-balance w A* = *0*)
∧ *arc-set-balance u A* = −*1*
∧ *arc-set-balance v A* = *1*

abbreviation *arc-balance* :: *'a* ⇒ *'b awalk* ⇒ *int* **where**
arc-balance w p ≡ *arc-set-balance w* (*set p*)

abbreviation *arc-balanced* :: *'a* ⇒ *'b awalk* ⇒ *'a* ⇒ *bool* **where**
arc-balanced u p v ≡ *arc-set-balanced u* (*set p*) *v*

lemma *arc-set-balanced-all*:
arc-set-balanced u (*arcs G*) *v* =
(*if u* = *v then* (∀ *w* ∈ *verts G*. *in-degree G w* = *out-degree G w*)
else (∀ *w* ∈ *verts G*. (*w* ≠ *u* ∧ *w* ≠ *v*) ⟶ *in-degree G w* = *out-degree G w*)
∧ *in-degree G u* + *1* = *out-degree G u*
∧ *out-degree G v* + *1* = *in-degree G v*)
⟨*proof*⟩

end

context *wf-digraph* **begin**

lemma *arc-balance-Cons*:
assumes *trail u* (*e* # *es*) *v*
shows *arc-set-balance w* (*insert e* (*set es*)) = *arc-set-balance w* {*e*} + *arc-balance w es*
⟨*proof*⟩

lemma *arc-balancedI-trail*:
assumes *trail u p v* **shows** *arc-balanced u p v*
⟨*proof*⟩

lemma *trail-arc-balanceE*:
assumes *trail u p v*
obtains ⋀*w*. ⟦ *u* = *v* ∨ (*w* ≠ *u* ∧ *w* ≠ *v*); *w* ∈ *verts G* ⟧
⟹ *arc-balance w p* = *0*
and ⟦ *u* ≠ *v* ⟧ ⟹ *arc-balance u p* = − *1*
and ⟦ *u* ≠ *v* ⟧ ⟹ *arc-balance v p* = *1*
⟨*proof*⟩

end

16.3 Closed Euler Trails

lemma (**in** *wf-digraph*) *awalk-vertex-props*:
assumes *awalk u p v p* ≠ []
assumes ⋀*w*. *w* ∈ *set* (*awalk-verts u p*) ⟹ *P w* ∨ *Q w*
assumes *P u Q v*
shows ∃ *e* ∈ *set p*. *P* (*tail G e*) ∧ *Q* (*head G e*)
⟨*proof*⟩

lemma (**in** *wf-digraph*) *connected-verts*:
assumes *connected G arcs G* ≠ {}
shows *verts G* = *tail G* ‘ *arcs G* ∪ *head G* ‘ *arcs G*
⟨*proof*⟩

lemma (**in** *wf-digraph*) *connected-arcs-empty*:
assumes *connected G arcs G* = {} *verts G* ≠ {} **obtains** *v* **where** *verts G* = {*v*}
⟨*proof*⟩

lemma (**in** *wf-digraph*) *euler-trail-conv-connected*:
assumes *connected G*
shows *euler-trail u p v* ⟷ *trail u p v* ∧ *set p* = *arcs G* (**is** *?L* ⟷ *?R*)
⟨*proof*⟩

lemma (**in** *wf-digraph*) *awalk-connected*:
assumes *connected G awalk u p v set p* ≠ *arcs G*
shows ∃ *e*. *e* ∈ *arcs G* − *set p* ∧ (*tail G e* ∈ *set* (*awalk-verts u p*) ∨ *head G e* ∈ *set* (*awalk-verts u p*))

⟨*proof*⟩

lemma (**in** *wf-digraph*) *trail-connected*:
assumes *connected G trail u p v set p* ≠ *arcs G*
shows ∃ *e*. *e* ∈ *arcs G* − *set p* ∧ (*tail G e* ∈ *set* (*awalk-verts u p*) ∨ *head G e* ∈ *set* (*awalk-verts u p*))
⟨*proof*⟩

theorem (**in** *fin-digraph*) *closed-euler1*:
assumes *con*: *connected G*
assumes *deg*: ⋀*u*. *u* ∈ *verts G* ⟹ *in-degree G u* = *out-degree G u*
shows ∃ *u p*. *euler-trail u p u*
⟨*proof*⟩

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*⟩

theorem (**in** *fin-digraph*) *closed-euler2*:
assumes *euler-trail u p u*
shows *connected G*
and ⋀*u*. *u* ∈ *verts G* ⟹ *in-degree G u* = *out-degree G u* (**is** ⋀*u*. - ⟹ *?eq-deg u*)
⟨*proof*⟩

corollary (**in** *fin-digraph*) *closed-euler*:
(∃ *u p*. *euler-trail u p u*) ⟷ *connected G* ∧ (∀ *u* ∈ *verts G*. *in-degree G u* = *out-degree G u*)
⟨*proof*⟩

16.4 Open euler trails

Intuitively, a graph has an open euler trail if and only if it is possible to add an arc such that the resulting graph has a closed euler trail. However, this is not true in our formalization, as the arc type *'b* might be finite:

Consider for example the graph (|*verts* = {*0*::*'a*, *1*::*'a*}, *arcs* = {()}, *tail* = λ-. *0*::*'a*, *head* = λ-. *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* × *nat* × *'a*. Hence, we first characterize the existence of euler trail for the infinite arc type *'a* × *nat* × *'a* and transfer that result back to arbitrary arc types.

lemma *open-euler-infinite-label*:

fixes $G :: ('a, 'a \times nat \times 'a)$ *pre-digraph*
assumes *fin-digraph G*
assumes [*simp*]: *tail G = fst head G = snd o snd*
assumes *con*: *connected G*
assumes *uv*: $u \in verts\ G\ v \in verts\ G$
assumes *deg*: $\bigwedge w.\ [\![w \in verts\ G;\ u \neq w;\ v \neq w]\!] \Longrightarrow in\text{-}degree\ G\ w = out\text{-}degree\ G\ w$
assumes *deg-in*: *in-degree G u + 1 = out-degree G u*
assumes *deg-out*: *out-degree G v + 1 = in-degree G v*
shows $\exists p.$ *pre-digraph.euler-trail G u p v*
⟨*proof*⟩

context *wf-digraph* **begin**

lemma *trail-app-isoI*:
assumes *t*: *trail u p v*
and *hom*: *digraph-isomorphism hom*
shows *pre-digraph.trail* (*app-iso hom G*) (*iso-verts hom u*) (*map* (*iso-arcs hom*) *p*) (*iso-verts hom v*)
⟨*proof*⟩

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*⟩

end

context *fin-digraph* **begin**

theorem *open-euler1*:
assumes *connected G*
assumes $u \in verts\ G\ v \in verts\ G$
assumes $\bigwedge w.\ [\![w \in verts\ G;\ u \neq w;\ v \neq w]\!] \Longrightarrow in\text{-}degree\ G\ w = out\text{-}degree\ G\ w$
assumes *in-degree G u + 1 = out-degree G u*
assumes *out-degree G v + 1 = in-degree G v*
shows $\exists p.$ *euler-trail u p v*
⟨*proof*⟩

theorem *open-euler2*:
assumes *et*: *euler-trail u p v* **and** $u \neq v$
shows *connected G* ∧
$(\forall w \in verts\ G.\ u \neq w \longrightarrow v \neq w \longrightarrow in\text{-}degree\ G\ w = out\text{-}degree\ G\ w) \wedge$
in-degree G u + 1 = out-degree G u ∧
out-degree G v + 1 = in-degree G v

⟨*proof*⟩

corollary *open-euler*:
(∃ *u p v*. *euler-trail u p v* ∧ *u* ≠ *v*) ⟷
connected G ∧ (∃ *u v*. *u* ∈ *verts G* ∧ *v* ∈ *verts G* ∧
(∀ *w* ∈ *verts G*. *u* ≠ *w* ⟶ *v* ≠ *w* ⟶ *in-degree G w* = *out-degree G w*) ∧
in-degree G u + *1* = *out-degree G u* ∧
out-degree G v + *1* = *in-degree G v*) (**is** *?L* ⟷ *?R*)
⟨*proof*⟩

end

end

theory *Kuratowski*
imports
Arc-Walk
Digraph-Component
Subdivision
HOL−Library.Rewrite
begin

17 Kuratowski Subgraphs

We consider the underlying undirected graphs. The underlying undirected graph is represented as a symmetric digraph.

17.1 Public definitions

definition *complete-digraph* :: *nat* ⇒ (*'a*,*'b*) *pre-digraph* ⇒ *bool* (*K*_) **where**
complete-digraph n G ≡ *graph G* ∧ *card* (*verts G*) = *n* ∧ *arcs-ends G* = {(*u*,*v*). (*u*,*v*) ∈ *verts G* × *verts G* ∧ *u* ≠ *v*}

definition *complete-bipartite-digraph* :: *nat* ⇒ *nat* ⇒ (*'a*, *'b*) *pre-digraph* ⇒ *bool* (*K*_,_) **where**
complete-bipartite-digraph m n G ≡ *graph G* ∧ (∃ *U V*. *verts G* = *U* ∪ *V* ∧ *U* ∩ *V* = {}
∧ *card U* = *m* ∧ *card V* = *n* ∧ *arcs-ends G* = *U* × *V* ∪ *V* × *U*)

definition *kuratowski-planar* :: (*'a*,*'b*) *pre-digraph* ⇒ *bool* **where**
kuratowski-planar G ≡ ¬(∃ *H*. *subgraph H G* ∧ (∃ *K rev-K rev-H*. *subdivision* (*K*, *rev-K*) (*H*, *rev-H*) ∧ (*K*$_{3,3}$ *K* ∨ *K*$_5$ *K*)))

lemma *complete-digraph-pair-def*: *K*$_n$ (*with-proj G*)
⟷ *finite* (*pverts G*) ∧ *card* (*pverts G*) = *n* ∧ *parcs G* = {(*u*,*v*). (*u*,*v*) ∈ (*pverts G* × *pverts G*) ∧ *u* ≠ *v*} (**is** - = *?R*)
⟨*proof*⟩

lemma *complete-bipartite-digraph-pair-def*: $K_{m,n}$ (*with-proj G*) $\longleftrightarrow$ *finite* (*pverts G*)
$\wedge$ ($\exists$ *U V*. *pverts G* = *U* $\cup$ *V* $\wedge$ *U* $\cap$ *V* = {} $\wedge$ *card U* = *m* $\wedge$ *card V* = *n* $\wedge$ *parcs G* = *U* $\times$ *V* $\cup$ *V* $\times$ *U*) (**is** - = *?R*)
⟨*proof*⟩

lemma *pair-graphI-complete*:
assumes K_n (*with-proj G*)
shows *pair-graph G*
⟨*proof*⟩

lemma *pair-graphI-complete-bipartite*:
assumes $K_{m,n}$ (*with-proj G*)
shows *pair-graph G*
⟨*proof*⟩

17.2 Inner vertices of a walk

context *pre-digraph* **begin**

definition (**in** *pre-digraph*) *inner-verts* :: *'b awalk* ⇒ *'a list* **where**
inner-verts p ≡ *tl* (*map* (*tail G*) *p*)

lemma *inner-verts-Nil*[*simp*]: *inner-verts* [] = [] ⟨*proof*⟩

lemma *inner-verts-singleton*[*simp*]: *inner-verts* [*x*] = [] ⟨*proof*⟩

lemma (**in** *wf-digraph*) *inner-verts-Cons*:
assumes *awalk u* (*e* # *es*) *v*
shows *inner-verts* (*e* # *es*) = (*if es* ≠ [] *then head G e* # *inner-verts es else* [])
⟨*proof*⟩

lemma (**in** −) *inner-verts-with-proj-def*:
pre-digraph.inner-verts (*with-proj G*) *p* = *tl* (*map fst p*)
⟨*proof*⟩

lemma *inner-verts-conv*: *inner-verts p* = *butlast* (*tl* (*awalk-verts u p*))
⟨*proof*⟩

lemma (**in** *pre-digraph*) *inner-verts-empty*[*simp*]:
assumes *length p* < *2* **shows** *inner-verts p* = []
⟨*proof*⟩

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*⟩

lemma *in-set-inner-verts-appendI-l*:
assumes $u \in$ *set* (*inner-verts* p)
shows $u \in$ *set* (*inner-verts* (p @ q))
⟨*proof*⟩

lemma *in-set-inner-verts-appendI-r*:
assumes $u \in$ *set* (*inner-verts* q)
shows $u \in$ *set* (*inner-verts* (p @ q))
⟨*proof*⟩

end

17.3 Progressing Walks

We call a walk *progressing* if it does not contain the sequence $[(x, y), (y, x)]$. This concept is relevant in particular for *iapath*s: If all of the inner vertices have degree at most 2 this implies that such a walk is a trail and even a path.

definition *progressing* :: ('a × 'a) *awalk* ⇒ *bool* **where**
progressing $p \equiv \forall$ *xs x y ys*. $p \neq xs$ @ (x,y) # (y,x) # *ys*

lemma *progressing-Nil*: *progressing* []
⟨*proof*⟩

lemma *progressing-single*: *progressing* [e]
⟨*proof*⟩

lemma *progressing-ConsD*:
assumes *progressing* (e # *es*) **shows** *progressing es*
⟨*proof*⟩

lemma *progressing-Cons*:
progressing (x # *xs*) ⟷ (*xs* = [] ∨ (*xs* ≠ [] ∧ ¬(*fst x* = *snd* (*hd xs*) ∧ *snd x* = *fst* (*hd xs*)) ∧ *progressing xs*)) (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *progressing-Cons-Cons*:
progressing ((u,v) # (v,w) # *es*) ⟷ $u \neq w$ ∧ *progressing* ((v,w) # *es*) (**is** *?L* ⟷ *?R*)
⟨*proof*⟩

lemma *progressing-appendD1*:
assumes *progressing* (p @ q) **shows** *progressing p*
⟨*proof*⟩

lemma *progressing-appendD2*:
assumes *progressing* (p @ q) **shows** *progressing q*
⟨*proof*⟩

lemma *progressing-rev-path*:
progressing (*rev-path p*) = *progressing p* (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *progressing-append-iff*:
shows *progressing* (*xs* @ *ys*) ⟷ *progressing xs* ∧ *progressing ys*
∧ (*xs* ≠ [] ∧ *ys* ≠ [] ⟶ (*fst* (*last xs*) ≠ *snd* (*hd ys*) ∨ *snd* (*last xs*) ≠ *fst* (*hd ys*)))
⟨*proof*⟩

17.4 Walks with Restricted Vertices

definition *verts3* :: ('*a*, '*b*) *pre-digraph* ⇒ '*a set* **where**
verts3 G ≡ {*v* ∈ *verts G*. 2 < *in-degree G v*}

A path were only the end nodes may be in V

definition (**in** *pre-digraph*) *gen-iapath* :: '*a set* ⇒ '*a* ⇒ '*b awalk* ⇒ '*a* ⇒ *bool* **where**
gen-iapath V u p v ≡ *u* ∈ *V* ∧ *v* ∈ *V* ∧ *apath u p v* ∧ *set* (*inner-verts p*) ∩ *V* = {} ∧ *p* ≠ []

abbreviation (**in** *pre-digraph*) (*input*) *iapath* :: '*a* ⇒ '*b awalk* ⇒ '*a* ⇒ *bool* **where**
iapath u p v ≡ *gen-iapath* (*verts3 G*) *u p v*

definition *gen-contr-graph* :: ('*a*,'*b*) *pre-digraph* ⇒ '*a set* ⇒ '*a pair-pre-digraph* **where**
gen-contr-graph G V ≡ (|
pverts = *V*,
parcs = {(*u*,*v*). ∃ *p*. *pre-digraph.gen-iapath G V u p v*}
|)

abbreviation (*input*) *contr-graph* :: '*a pair-pre-digraph* ⇒ '*a pair-pre-digraph* **where**
contr-graph G ≡ *gen-contr-graph G* (*verts3 G*)

17.5 Properties of subdivisions

lemma (**in** *pair-sym-digraph*) *verts3-subdivide*:
assumes *e* ∈ *parcs G w* ∉ *pverts G*
shows*verts3* (*subdivide G e w*) = *verts3 G*
⟨*proof*⟩

lemma *sd-path-Nil-iff*:
sd-path e w p = [] ⟷ *p* = []
⟨*proof*⟩

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*: $V \subseteq$ *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*⟩

lemma (**in** *pair-sym-digraph*)
assumes *elems*: $e \in$ *parcs G w* $\notin$ *pverts G*
assumes *V*: $V \subseteq$ *pverts G*
assumes *path*: *pre-digraph.gen-iapath* (*subdivide G e w*) *V u p v*
shows *gen-iapath-co-path*: *gen-iapath V u* (*co-path e w p*) *v* (**is** *?thesis-path*)
and *set-awalk-verts-co-path'*: *set* (*awalk-verts u* (*co-path e w p*)) = *set* (*awalk-verts u p*) − {*w*} (**is** *?thesis-set*)
⟨*proof*⟩

17.6 Pair Graphs

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

lemma *gen-iapath-rev-path*:
gen-iapath V v (*rev-path p*) *u* = *gen-iapath V u p v* (**is** *?L* = *?R*)
⟨*proof*⟩

lemma *inner-verts-rev-path*:
assumes *awalk u p v*
shows *inner-verts* (*rev-path p*) = *rev* (*inner-verts p*)
⟨*proof*⟩

end

context *pair-pseudo-graph* **begin**

lemma *apath-imp-progressing*:
assumes *apath u p v* **shows** *progressing p*
⟨*proof*⟩

lemma *awalk-Cons-deg2-unique*:
assumes *awalk u p v p* ≠ []
assumes *in-degree G u* ≤ *2*
assumes *awalk u1* (*e1* # *p*) *v awalk u2* (*e2* # *p*) *v*
assumes *progressing* (*e1* # *p*) *progressing* (*e2* # *p*)
shows *e1* = *e2*
⟨*proof*⟩

lemma *same-awalk-by-same-end*:
assumes *V*: *verts3 G* ⊆ *V V* ⊆ *pverts G*
and *walk*: *awalk u p v awalk u q w hd p* = *hd q p* ≠ [] *q* ≠ []
and *progress*: *progressing p progressing q*
and *tail*: *v* ∈ *V w* ∈ *V*
and *inner-verts*: *set* (*inner-verts p*) ∩ *V* = {}

set (*inner-verts q*) ∩ *V* = {}
shows *p* = *q*
⟨*proof*⟩

lemma *same-awalk-by-common-arc*:
assumes *V*: *verts3 G* ⊆ *V V* ⊆ *pverts G*
assumes *walk*: *awalk u p v awalk w q x*
assumes *progress*: *progressing p progressing q*
assumes *iv-not-in-V*: *set* (*inner-verts p*) ∩ *V* = {} *set* (*inner-verts q*) ∩ *V* = {}
assumes *ends-in-V*: {*u*,*v*,*w*,*x*} ⊆ *V*
assumes *arcs*: *e* ∈ *set p e* ∈ *set q*
shows *p* = *q*
⟨*proof*⟩

lemma *same-gen-iapath-by-common-arc*:
assumes *V*: *verts3 G* ⊆ *V V* ⊆ *pverts G*
assumes *path*: *gen-iapath V u p v gen-iapath V w q x*
assumes *arcs*: *e* ∈ *set p e* ∈ *set q*
shows *p* = *q*
⟨*proof*⟩

end

17.7 Slim graphs

We define the notion of a slim graph. The idea is that for a slim graph *G*, *G* is a subdivision of *gen-contr-graph* (*with-proj G*) (*verts3* (*with-proj G*)).

context *pair-pre-digraph* **begin**

definition (**in** *pair-pre-digraph*) *is-slim* :: *'a set* ⇒ *bool* **where**
is-slim V ≡
(∀ *v* ∈ *pverts G*. *v* ∈ *V* ∨
in-degree G v ≤ *2* ∧ (∃ *x p y*. *gen-iapath V x p y* ∧ *v* ∈ *set* (*awalk-verts x p*)))
∧
(∀ *e* ∈ *parcs G*. *fst e* ≠ *snd e* ∧ (∃ *x p y*. *gen-iapath V x p y* ∧ *e* ∈ *set p*)) ∧
(∀ *u v p q*. (*gen-iapath V u p v* ∧ *gen-iapath V u q v*) ⟶ *p* = *q*) ∧
V ⊆ *pverts G*

definition *direct-arc* :: *'a* × *'a* ⇒ *'a* × *'a* **where**
direct-arc uv ≡ *SOME e*. {*fst uv* , *snd uv*} = {*fst e*, *snd e*}

definition *choose-iapath* :: *'a* ⇒ *'a* ⇒ (*'a* × *'a*) *awalk* **where**
choose-iapath u v ≡ (*let*
chosen-path = (λ*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* × ('*a* × '*a*) *awalk* × '*a*) *set* **where**
slim-paths ≡ (λ*e*. (*fst e*, *choose-iapath* (*fst e*) (*snd e*), *snd e*)) ‘ *parcs* (*contr-graph G*)

definition *slim-verts* :: '*a set* **where**
slim-verts ≡ *verts3 G* ∪ (⋃(*u*,*p*,-) ∈ *slim-paths*. *set* (*awalk-verts u p*))

definition *slim-arcs* :: '*a rel* **where**
slim-arcs ≡ ⋃(-,*p*,-) ∈ *slim-paths*. *set p*

Computes a slim subgraph for an arbitrary *pair-digraph*

definition *slim* :: '*a pair-pre-digraph* **where**
slim ≡ (| *pverts* = *slim-verts*, *parcs* = *slim-arcs* |)

end

lemma (**in** *wf-digraph*) *iapath-dist-ends*: ⋀*u p v*. *iapath u p v* ⟹ *u* ≠ *v*
⟨*proof*⟩

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

lemma *choose-iapath*:
assumes ∃ *p*. *iapath u p v*
shows *iapath u* (*choose-iapath u v*) *v*
⟨*proof*⟩

lemma *slim-simps*: *pverts slim* = *slim-verts parcs slim* = *slim-arcs*
⟨*proof*⟩

lemma *slim-paths-in-G-E*:
assumes (*u*,*p*,*v*) ∈ *slim-paths* **obtains** *iapath u p v u* ≠ *v*
⟨*proof*⟩

lemma *verts-slim-in-G*: *pverts slim* ⊆ *pverts G*
⟨*proof*⟩

lemma *verts3-in-slim-G*[*simp*]:
assumes *x* ∈ *verts3 G* **shows** *x* ∈ *pverts slim*
⟨*proof*⟩

lemma *arcs-slim-in-G*: *parcs slim* ⊆ *parcs G*
⟨*proof*⟩

lemma *slim-paths-in-slimG*:
assumes (*u*,*p*,*v*) ∈ *slim-paths*
shows *pre-digraph.gen-iapath slim* (*verts3 G*) *u p v* ∧ *p* ≠ []
⟨*proof*⟩

lemma *direct-arc-swapped*:
direct-arc (*u*,*v*) = *direct-arc* (*v*,*u*)
⟨*proof*⟩

lemma *direct-arc-chooses*:
fixes *u v* :: *'a* **shows** *direct-arc* (*u*,*v*) = (*u*,*v*) ∨ *direct-arc* (*u*,*v*) = (*v*,*u*)
⟨*proof*⟩

lemma *rev-path-choose-iapath*:
assumes *u* ≠ *v*
shows *rev-path* (*choose-iapath u v*) = *choose-iapath v u*
⟨*proof*⟩

lemma *no-loops-in-iapath*: *gen-iapath V u p v* ⟹ *a* ∈ *set p* ⟹ *fst a* ≠ *snd a*
⟨*proof*⟩

lemma *pair-bidirected-digraph-slim*: *pair-bidirected-digraph slim*
⟨*proof*⟩

lemma (**in** *pair-pseudo-graph*) *pair-graph-slim*: *pair-graph slim*
⟨*proof*⟩

lemma *subgraph-slim*: *subgraph slim G*
⟨*proof*⟩

lemma *giapath-if-slim-giapath*:
assumes *pre-digraph.gen-iapath slim* (*verts3 G*) *u p v*
shows *gen-iapath* (*verts3 G*) *u p v*
⟨*proof*⟩

lemma *slim-giapath-if-giapath*:
assumes *gen-iapath* (*verts3 G*) *u p v*
shows ∃ *p*. *pre-digraph.gen-iapath slim* (*verts3 G*) *u p v* (**is** ∃ *p*. *?P p*)
⟨*proof*⟩

lemma *contr-graph-slim-eq*:
gen-contr-graph slim (*verts3 G*) = *contr-graph G*
⟨*proof*⟩

end

context *pair-pseudo-graph* **begin**

lemma *verts3-slim-in-verts3*:
assumes *v* ∈ *verts3 slim* **shows** *v* ∈ *verts3 G*
⟨*proof*⟩

lemma *slim-is-slim*:
pair-pre-digraph.is-slim slim (*verts3 G*)
⟨*proof*⟩

end

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

lemma
assumes *p*: *gen-iapath* (*pverts G*) *u p v*
shows *gen-iapath-triv-path*: *p* = [(*u*,*v*)]
and *gen-iapath-triv-arc*: (*u*,*v*) ∈ *parcs G*
⟨*proof*⟩

lemma *gen-contr-triv*:
assumes *is-slim V pverts G* = *V* **shows** *gen-contr-graph G V* = *G*
⟨*proof*⟩

lemma *is-slim-no-loops*:
assumes *is-slim V a* ∈ *arcs G* **shows** *fst a* ≠ *snd a*
⟨*proof*⟩

end

17.8 Contraction Preserves Kuratowski-Subgraph-Property

lemma (**in** *pair-pseudo-graph*) *in-degree-contr*:
assumes *v* ∈ *V* **and** *V*: *verts3 G* ⊆ *V V* ⊆ *verts G*
shows *in-degree* (*gen-contr-graph G V*) *v* ≤ *in-degree G v*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *verts3-K33*:
assumes $K_{3,3}$ (*with-proj G*)
shows *verts3 G* = *verts G*
⟨*proof*⟩

lemma *verts3-K5*:
assumes K_5 (*with-proj G*)
shows *verts3 G* = *verts G*
⟨*proof*⟩

lemma *K33-contractedI*:
assumes *subd*: *subdivision-pair G H*
assumes *k33*: $K_{3,3}$ *G*
shows $K_{3,3}$ (*contr-graph H*)
⟨*proof*⟩

lemma *K5-contractedI*:
assumes *subd*: *subdivision-pair G H*
assumes *k5*: K_5 *G*
shows K_5 (*contr-graph H*)
⟨*proof*⟩

17.9 Final proof

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

lemma *gcg-subdivide-eq*:
assumes *mem*: *e* ∈ *parcs G w* ∉ *pverts G*
assumes *V*: *V* ⊆ *pverts G*
shows *gen-contr-graph* (*subdivide G e w*) *V* = *gen-contr-graph G V*
⟨*proof*⟩

lemma *co-path-append*:
assumes [*last p1*, *hd p2*] ∉ {[(*fst e*,*w*),(*w*,*snd e*)], [(*snd e*,*w*),(*w*,*fst e*)]}
shows *co-path e w* (*p1* @ *p2*) = *co-path e w p1* @ *co-path e w p2*
⟨*proof*⟩

lemma *exists-co-path-decomp1*:
assumes *mem*: *e* ∈ *parcs G w* ∉ *pverts G*
assumes *p*: *pre-digraph.apath* (*subdivide G e w*) *u p v* (*fst e*, *w*) ∈ *set p w* ≠ *v*
shows ∃ *p1 p2*. *p* = *p1* @ (*fst e*, *w*) # (*w*, *snd e*) # *p2*
⟨*proof*⟩

lemma *is-slim-if-subdivide*:
assumes *pair-pre-digraph.is-slim* (*subdivide G e w*) *V*
assumes *mem1*: *e* ∈ *parcs G w* ∉ *pverts G* **and** *mem2*: *w* ∉ *V*
shows *is-slim V*
⟨*proof*⟩

end

context *pair-pseudo-graph* **begin**

lemma *subdivision-gen-contr*:
assumes *is-slim V*
shows *subdivision-pair* (*gen-contr-graph G V*) *G*
⟨*proof*⟩

lemma *contr-is-subgraph-subdivision*:
shows $\exists H.$ *subgraph* (*with-proj H*) *G* $\wedge$ *subdivision-pair* (*contr-graph G*) *H*
⟨*proof*⟩

theorem *kuratowski-contr*:
fixes *K* :: *'a pair-pre-digraph*
assumes *subgraph-K*: *subgraph K G*
assumes *spd-K*: *pair-pseudo-graph K*
assumes *kuratowski*: $K_{3,3}$ (*contr-graph K*) $\vee$ K_5 (*contr-graph K*)
shows ¬*kuratowski-planar G*
⟨*proof*⟩

theorem *certificate-characterization*:
defines *kuratowski* $\equiv \lambda G$ :: *'a pair-pre-digraph.* $K_{3,3}$ *G* $\vee$ K_5 *G*
shows *kuratowski* (*contr-graph G*)
$\longleftrightarrow$ ($\exists H.$ *kuratowski H* $\wedge$ *subdivision-pair H slim* $\wedge$ *verts3 G* = *verts3 slim*)
(**is** *?L* $\longleftrightarrow$ *?R*)
⟨*proof*⟩

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* $\wedge$ ($K_{3,3}$ *C* $\vee$ $K_5 C$)

theorem *certify-complete*:
assumes *pair-pseudo-graph cert*
assumes *subgraph cert G*
assumes $\exists H.$ *subdivision-pair H cert* $\wedge$ ($K_{3,3}$ *H* $\vee$ K_5 *H*)
shows *certify cert*
⟨*proof*⟩

theorem *certify-sound*:
assumes *pair-pseudo-graph cert*
assumes *certify cert*
shows ¬*kuratowski-planar G*
⟨*proof*⟩

theorem *certify-characterization*:
assumes *pair-pseudo-graph cert*
shows *certify cert* $\longleftrightarrow$ *subgraph cert G* $\wedge$ *verts3 cert* = *verts3* (*pair-pre-digraph.slim cert*)
$\wedge$($\exists H.$ ($K_{3,3}$ (*with-proj H*) $\vee$ K_5 *H*) $\wedge$ *subdivision-pair H* (*pair-pre-digraph.slim cert*))
(**is** *?L* $\longleftrightarrow$ *?R*)
⟨*proof*⟩

end

end

theory *Weighted-Graph*
imports
Digraph
Arc-Walk
Complex-Main
begin

18 Weighted Graphs

type-synonym *'b weight-fun* = *'b* ⇒ *real*

context *wf-digraph* **begin**

definition *awalk-cost* :: *'b weight-fun* ⇒ *'b awalk* ⇒ *real* **where**
awalk-cost f es = *sum-list* (*map f es*)

lemma *awalk-cost-Nil*[*simp*]: *awalk-cost f* [] = *0*
⟨*proof*⟩

lemma *awalk-cost-Cons*[*simp*]: *awalk-cost f* (*x* # *xs*) = *f x* + *awalk-cost f xs*
⟨*proof*⟩

lemma *awalk-cost-append*[*simp*]:
awalk-cost f (*xs* @ *ys*) = *awalk-cost f xs* + *awalk-cost f ys*
⟨*proof*⟩

end

end

theory *Shortest-Path* **imports**
Arc-Walk
Weighted-Graph
HOL−Library.Extended-Real
begin

19 Shortest Paths

context *wf-digraph* **begin**

definition *μ* **where**
μ f u v ≡ *INF p*∈ {*p*. *awalk u p v*}. *ereal* (*awalk-cost f p*)

lemma *shortest-path-inf*:
assumes ¬(*u* →* *v*)

shows $\mu\ f\ u\ v = \infty$
⟨*proof*⟩

lemma *min-cost-le-walk-cost*:
assumes *awalk u p v*
shows $\mu\ c\ u\ v \leq$ *awalk-cost c p*
⟨*proof*⟩

lemma *pos-cost-pos-awalk-cost*:
assumes *awalk u p v*
assumes *pos-cost*: $\bigwedge e.\ e \in$ *arcs G* $\Longrightarrow$ *c e* ≥ 0
shows *awalk-cost c p* ≥ 0
⟨*proof*⟩

fun *mk-cycles-path* :: *nat*
$\Rightarrow$ *'b awalk* $\Rightarrow$ *'b awalk* **where**
mk-cycles-path 0 c = []
| *mk-cycles-path* (*Suc n*) *c* = *c* @ (*mk-cycles-path n c*)

lemma *mk-cycles-path-awalk*:
assumes *awalk u c u*
shows *awalk u* (*mk-cycles-path n c*) *u*
⟨*proof*⟩

lemma *mk-cycles-awalk-cost*:
assumes *awalk u p u*
shows *awalk-cost c* (*mk-cycles-path n p*) = *n* $*$ *awalk-cost c p*
⟨*proof*⟩

lemma *inf-over-nats*:
fixes *a c* :: *real*
assumes *c* < 0
shows (*INF* (*i* :: *nat*). *ereal* (*a* + *i* $*$ *c*)) = $-\infty$
⟨*proof*⟩

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* $\in$ *set* (*awalk-verts u p*)
assumes *awalk-cost f c* < 0
shows $\mu\ f\ u\ v = -\infty$
⟨*proof*⟩

lemma *walk-cheaper-path-imp-neg-cyc*:
assumes *p-props*: *awalk u p v*
assumes *less-path-μ*: *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*⟩

lemma (**in** *fin-digraph*) *neg-inf-imp-neg-cyc*:
assumes *inf-mu*: $\mu\ f\ u\ v = -\infty$
shows $\exists\, p.\ awalk\ u\ p\ v \wedge (\exists\, w\ c.\ awalk\ w\ c\ w \wedge w \in set\ (awalk\text{-}verts\ u\ p) \wedge awalk\text{-}cost\ f\ c < 0)$
⟨*proof*⟩

lemma (**in** *fin-digraph*) *no-neg-cyc-imp-no-neg-inf*:
assumes *no-neg-cyc*: $\bigwedge p.\ awalk\ u\ p\ v \Longrightarrow \neg(\exists\, w\ c.\ awalk\ w\ c\ w \wedge w \in set\ (awalk\text{-}verts\ u\ p) \wedge awalk\text{-}cost\ f\ c < 0)$
shows $-\infty < \mu\ f\ u\ v$
⟨*proof*⟩

lemma *μ-reach-conv*:
$\mu\ f\ u\ v < \infty \longleftrightarrow u \rightarrow^* v$
⟨*proof*⟩

lemma *awalk-to-path-no-neg-cyc-cost*:
assumes *p-props*:*awalk u p v*
assumes *no-neg-cyc*: $\neg\ (\exists\, w\ c.\ awalk\ w\ c\ w \wedge w \in set\ (awalk\text{-}verts\ u\ p) \wedge awalk\text{-}cost\ f\ c < 0)$
shows *awalk-cost f* (*awalk-to-apath p*) ≤ *awalk-cost f p*
⟨*proof*⟩

lemma (**in** *fin-digraph*) *no-neg-cyc-reach-imp-path*:
assumes *reach*: $u \rightarrow^* v$
assumes *no-neg-cyc*: $\bigwedge p.\ awalk\ u\ p\ v \Longrightarrow \neg(\exists\, w\ c.\ awalk\ w\ c\ w \wedge w \in set\ (awalk\text{-}verts\ u\ p) \wedge awalk\text{-}cost\ f\ c < 0)$
shows $\exists\, p.\ apath\ u\ p\ v \wedge \mu\ f\ u\ v = awalk\text{-}cost\ f\ p$
⟨*proof*⟩

lemma (**in** *fin-digraph*) *min-cost-awalk*:
assumes *reach*: $u \rightarrow^* v$
assumes *pos-cost*: $\bigwedge e.\ e \in arcs\ G \Longrightarrow c\ e \geq 0$
shows $\exists\, p.\ apath\ u\ p\ v \wedge \mu\ c\ u\ v = awalk\text{-}cost\ c\ p$
⟨*proof*⟩

lemma (**in** *fin-digraph*) *pos-cost-mu-triangle*:
assumes *pos-cost*: $\bigwedge e.\ e \in arcs\ G \Longrightarrow c\ e \geq 0$
assumes *e-props*: *arc-to-ends G e* = (*u*,*v*) $e \in arcs\ G$
shows $\mu\ c\ s\ v \leq \mu\ c\ s\ u + c\ e$
⟨*proof*⟩

lemma (**in** *fin-digraph*) *mu-exact-triangle*:
assumes $v \neq s$
assumes $s \rightarrow^* v$
assumes *nonneg-arcs*: $\bigwedge e.\ e \in arcs\ G \Longrightarrow 0 \leq c\ e$
obtains $u\ e$ **where** $\mu\ c\ s\ v = \mu\ c\ s\ u + c\ e$ **and** *arc e* (*u*,*v*)
⟨*proof*⟩

lemma (**in** *fin-digraph*) *mu-exact-triangle-Ex*:
assumes $v \neq s$
assumes $s \rightarrow^* v$
assumes $\bigwedge e.\ e \in arcs\ G \Longrightarrow 0 \leq c\ e$
shows $\exists u\ e.\ \mu\ c\ s\ v = \mu\ c\ s\ u + c\ e \wedge arc\ e\ (u,v)$
⟨*proof*⟩

lemma (**in** *fin-digraph*) *mu-Inf-triangle*:
assumes $v \neq s$
assumes $\bigwedge e.\ e \in arcs\ G \Longrightarrow 0 \leq c\ e$
shows $\mu\ c\ s\ v = Inf\ \{\mu\ c\ s\ u + c\ e \mid u\ e.\ arc\ e\ (u,\ v)\}$ (**is** - = *Inf ?S*)
⟨*proof*⟩

end

end

theory *Graph-Theory*
imports
Digraph
Bidirected-Digraph
Arc-Walk

Digraph-Component
Digraph-Component-Vwalk
Digraph-Isomorphism
Pair-Digraph
Vertex-Walk
Subdivision

Euler
Kuratowski
Shortest-Path

begin

end

References

[1] J. Bang-Jensen and G. Z. Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer, 2 edition, 2009.

[2] R. Diestel. *Graph Theory*, volume 173 of *Graduate Texts in Mathematics*. Springer, 4 edition, 2010. http://diestel-graph-theory.com.

[3] F. Harary and R. Read. Is the null-graph a pointless concept? In R. Bari and F. Harary, editors, *Graphs and Combinatorics*, volume 406 of *Lecture Notes in Mathematics*, pages 37–44. Springer Berlin Heidelberg, 1974.