

Undirected Graph Theory

Chelsea Edmonds

January 31, 2023

Abstract

This entry presents a general library for undirected graph theory - enabling reasoning on simple graphs and undirected graphs with loops. It primarily builds off Noschinski's basic ugraph definition [4], however generalises it in a number of ways and significantly expands on the range of basic graph theory definitions formalised. Notably, this library removes the constraint of vertices being a type synonym with the natural numbers which causes issues in more complex mathematical reasoning using graphs, such as the Balog Szemerédi Gowers theorem which this library is used for. Secondly this library also presents a locale-centric approach, enabling more concise, flexible, and reusable modelling of different types of graphs. Using this approach enables easy links to be made with more expansive formalisations of other combinatorial structures, such as incidence systems, as well as various types of formal representations of graphs. Further inspiration is also taken from Noschinski's [5] Directed Graph library for some proofs and definitions on walks, paths and cycles, however these are much simplified using the set based representation of graphs, and also extended on in this formalisation.

Contents

1	Undirected Graph Theory Basics	3
1.1	Miscellaneous Extras	3
1.2	Initial Set up	4
1.3	Graph System Locale	8
1.4	Undirected Graph with Loops	9
1.5	Edge Density	17
1.6	Simple Graphs	19
1.7	Subgraph Basics	21
2	Walks, Paths and Cycles	24
2.1	Walks	25
2.2	Paths	32
2.3	Cycles	33

3	Connectivity	37
3.1	Connecting Walks and Paths	37
3.2	Vertex Connectivity	41
3.3	Graph Properties on Connectivity	42
3.4	We define a connected graph as a non-empty graph (the empty set is not usually considered connected by convention), where the vertex set is connected	47
4	Girth and Independence	52
5	Triangles in Graph	55
5.1	Preliminaries on Triangles in Graphs	56
6	Bipartite Graphs	60
6.1	Bipartite Set Up	60
6.2	Bipartite Graph Locale	62
7	Graph Theory Inheritance	69
7.1	Design Inheritance	69
7.2	Adjacency Relation Definition	70

Acknowledgements

Chelsea Edmonds is jointly funded by the Cambridge Trust (Cambridge Australia Scholarship) and a Cambridge Department of Computer Science and Technology Premium Research Studentship. The ALEXANDRIA project is funded by the European Research Council, Advanced Grant GA 742178.

This library aims to present a general theory for undirected graphs. The formalisation approach models edges as sets with two elements, and is inspired in part by the graph theory basics defined by Lars Noschinski in [4] which are used in [2, 1]. Crucially this library makes the definition more flexible by removing the type synonym from vertices to natural numbers. This is limiting in more advanced mathematical applications, where it is common for vertices to represent elements of some other set. It additionally extends significantly on basic graph definitions.

The approach taken in this formalisation is the "locale-centric" approach for modelling different graph properties, which has been successfully used in other combinatorial structure formalisations.

1 Undirected Graph Theory Basics

This first theory focuses on the basics of graph theory (vertices, edges, degree, incidence, neighbours etc), as well as defining a number of different types of basic graphs. This theory draws inspiration from [4, 2, 1]

theory *Undirected-Graph-Basics* **imports** *Main HOL-Library.Multiset HOL-Library.Disjoint-Sets*

HOL-Library.Extended-Real Girth-Chromatic.Girth-Chromatic-Misc
begin

1.1 Miscellaneous Extras

Useful concepts on lists and sets

lemma *distinct-tl-rev*:

assumes *hd xs = last xs*

shows *distinct (tl xs) \longleftrightarrow distinct (tl (rev xs))*

using *assms*

proof (*induct xs*)

case *Nil*

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

next

case (*Cons a xs*)

then show *?case* **proof** (*cases xs = []*)

case *True*

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

next

case *False*

then have *a = last xs*

using *Cons.prem* **by** *auto*

then obtain *xs'* **where** *xs = xs' @ [last xs]*

by (*metis False append-butlast-last-id*)

then have *tleq: tl (rev xs) = rev (xs')*

by (*metis butlast-rev butlast-snoc rev-rev-ident*)

have *distinct (tl (a # xs)) \longleftrightarrow distinct xs* **by** *simp*

also have ... \longleftrightarrow $\text{distinct } (\text{rev } xs') \wedge a \notin \text{set } (\text{rev } xs')$
by (*metis False Nil-is-rev-conv* $\langle a = \text{last } xs \rangle$ *distinct.simps(2)* *distinct-rev*
hd-rev list.exhaust-sel tleq)
finally show $\text{distinct } (\text{tl } (a \# xs)) \longleftrightarrow \text{distinct } (\text{tl } (\text{rev } (a \# xs)))$
using *tleq* **by** (*simp add: False*)
qed
qed

lemma *last-in-list-set*: $\text{length } xs \geq 1 \implies \text{last } xs \in \text{set } (xs)$
using *dual-order.strict-trans1 last-in-set* **by** *blast*

lemma *last-in-list-tl-set*:
assumes $\text{length } xs \geq 2$
shows $\text{last } xs \in \text{set } (\text{tl } xs)$
using *assms* **by** (*induct xs*) *auto*

lemma *length-list-decomp-lt*: $ys \neq [] \implies \text{length } (xs @ zs) < \text{length } (xs @ ys @ zs)$
using *length-append* **by** *simp*

lemma *obtains-Max*:
assumes *finite A* **and** $A \neq \{\}$
obtains x **where** $x \in A$ **and** $\text{Max } A = x$
using *assms Max-in* **by** *blast*

lemma *obtains-MAX*:
assumes *finite A* **and** $A \neq \{\}$
obtains x **where** $x \in A$ **and** $\text{Max } (f \text{ ` } A) = f x$
using *obtains-Max*
by (*metis (mono-tags, opaque-lifting) assms(1) assms(2) empty-is-image finite-imageI image-iff*)

lemma *obtains-Min*:
assumes *finite A* **and** $A \neq \{\}$
obtains x **where** $x \in A$ **and** $\text{Min } A = x$
using *assms Min-in* **by** *blast*

lemma *obtains-MIN*:
assumes *finite A* **and** $A \neq \{\}$
obtains x **where** $x \in A$ **and** $\text{Min } (f \text{ ` } A) = f x$
using *obtains-Min assms empty-is-image finite-imageI image-iff*
by (*metis (mono-tags, opaque-lifting)*)

1.2 Initial Set up

For convenience and readability, some functions and type synonyms are defined outside locale context

fun *mk-triangle-set* :: $('a \times 'a \times 'a) \Rightarrow 'a \text{ set}$
where $\text{mk-triangle-set } (x, y, z) = \{x, y, z\}$

type-synonym $'a$ edge = $'a$ set

type-synonym $'a$ pregraph = $('a$ set) \times ($'a$ edge set)

abbreviation $gverts :: 'a$ pregraph $\Rightarrow 'a$ set **where**
 $gverts H \equiv fst H$

abbreviation $gedges :: 'a$ pregraph $\Rightarrow 'a$ edge set **where**
 $gedges H \equiv snd H$

fun $mk-edge :: 'a \times 'a \Rightarrow 'a$ edge **where**
 $mk-edge (u,v) = \{u,v\}$

All edges is simply the set of subsets of a set S of size 2

definition $all-edges S \equiv \{e . e \subseteq S \wedge card e = 2\}$

Note, this is a different definition to Noschinski's [4] ugraph which uses the $mk-edge$ function unnecessarily

Basic properties of these functions

lemma $all-edges-mono$:
 $vs \subseteq ws \implies all-edges vs \subseteq all-edges ws$
unfolding $all-edges-def$ **by** $auto$

lemma $all-edges-alt$: $all-edges S = \{\{x, y\} \mid x y . x \in S \wedge y \in S \wedge x \neq y\}$
unfolding $all-edges-def$

proof ($intro$ $subset-antisym$ $subsetI$)

fix x **assume** $x \in \{e . e \subseteq S \wedge card e = 2\}$

then obtain $u v$ **where** $x = \{u, v\}$ **and** $card \{u, v\} = 2$ **and** $\{u, v\} \subseteq S$

by ($metis$ ($mono-tags$, $lifting$) $card-2-iff$ $mem-Collect-eq$)

then show $x \in \{\{x, y\} \mid x y . x \in S \wedge y \in S \wedge x \neq y\}$

by $fastforce$

next

show $\bigwedge x . x \in \{\{x, y\} \mid x y . x \in S \wedge y \in S \wedge x \neq y\} \implies x \in \{e . e \subseteq S \wedge card e = 2\}$

by $auto$

qed

lemma $all-edges-alt-pairs$: $all-edges S = mk-edge \{ \{uv \in S \times S . fst uv \neq snd uv\}$
unfolding $all-edges-alt$

proof ($intro$ $subset-antisym$)

have $img: mk-edge \{ \{uv \in S \times S . fst uv \neq snd uv\} = \{mk-edge (u, v) \mid u v . (u, v) \in S \times S \wedge u \neq v\}$

by (smt ($z3$) $Collect-cong$ $fst-conv$ $prod.collapse$ $setcompr-eq-image$ $snd-conv$)

then show $mk-edge \{ \{uv \in S \times S . fst uv \neq snd uv\} \subseteq \{\{x, y\} \mid x y . x \in S \wedge y \in S \wedge x \neq y\}$

by $auto$

show $\{\{x, y\} \mid x \in S \wedge y \in S \wedge x \neq y\} \subseteq \text{mk-edge } \{uv \in S \times S. \text{fst } uv \neq \text{snd } uv\}$
using *img* **by** *simp*
qed

lemma *all-edges-subset-Pow*: $\text{all-edges } A \subseteq \text{Pow } A$
by (*auto simp: all-edges-def*)

lemma *all-edges-disjoint*: $S \cap T = \{\} \implies \text{all-edges } S \cap \text{all-edges } T = \{\}$
by (*auto simp add: all-edges-def disjoint-iff subset-eq*)

lemma *card-all-edges*: $\text{finite } A \implies \text{card } (\text{all-edges } A) = \text{card } A \text{ choose } 2$
using *all-edges-def* **by** (*metis (full-types) n-subsets*)

lemma *finite-all-edges*: $\text{finite } S \implies \text{finite } (\text{all-edges } S)$
by (*meson all-edges-subset-Pow finite-Pow-iff finite-subset*)

lemma *in-mk-edge-img*: $(a,b) \in A \vee (b,a) \in A \implies \{a,b\} \in \text{mk-edge } A$
by (*auto intro: rev-image-eqI*)

thm *in-mk-edge-img*

lemma *in-mk-uedge-img-iff*: $\{a,b\} \in \text{mk-edge } A \longleftrightarrow (a,b) \in A \vee (b,a) \in A$
by (*auto simp: doubleton-eq-iff intro: rev-image-eqI*)

lemma *inj-on-mk-edge*: $X \cap Y = \{\} \implies \text{inj-on } \text{mk-edge } (X \times Y)$
by (*auto simp: inj-on-def doubleton-eq-iff*)

definition *complete-graph* :: 'a set \Rightarrow 'a pregraph **where**
complete-graph $S \equiv (S, \text{all-edges } S)$

definition *all-edges-loops*:: 'a set \Rightarrow 'a edge set**where**
all-edges-loops $S \equiv \text{all-edges } S \cup \{\{v\} \mid v. v \in S\}$

lemma *all-edges-loops-alt*: $\text{all-edges-loops } S = \{e. e \subseteq S \wedge (\text{card } e = 2 \vee \text{card } e = 1)\}$

proof –

have *1*: $\{\{v\} \mid v. v \in S\} = \{e. e \subseteq S \wedge \text{card } e = 1\}$

by (*metis One-nat-def card.empty card-Suc-eq empty-iff empty-subsetI insert-subset is-singleton-altdef is-singleton-the-elem*)

have $\{e. e \subseteq S \wedge (\text{card } e = 2 \vee \text{card } e = 1)\} = \{e. e \subseteq S \wedge \text{card } e = 2\} \cup \{e. e \subseteq S \wedge \text{card } e = 1\}$

by *auto*

then have $\{e. e \subseteq S \wedge (\text{card } e = 2 \vee \text{card } e = 1)\} = \text{all-edges } S \cup \{\{v\} \mid v. v \in S\}$

by (*simp add: all-edges-def 1*)

then show *?thesis* **unfolding** *all-edges-loops-def* **by** *simp*

qed

lemma *loops-disjoint*: $\text{all-edges } S \cap \{\{v\} \mid v. v \in S\} = \{\}$

unfolding *all-edges-def* **using** *card-2-iff*
by *fastforce*

lemma *all-edges-loops-ss*: $\text{all-edges } S \subseteq \text{all-edges-loops } S \ \{\{v\} \mid v. v \in S\} \subseteq \text{all-edges-loops } S$
by (*simp-all add: all-edges-loops-def*)

lemma *finite-singletons*: $\text{finite } S \implies \text{finite } (\{\{v\} \mid v. v \in S\})$
by (*auto*)

lemma *card-singletons*:
assumes *finite S* **shows** $\text{card } \{\{v\} \mid v. v \in S\} = \text{card } S$
using *assms*
proof (*induct S rule: finite-induct*)
case *empty*
then show *?case* **by** *simp*
next
case (*insert x F*)
then have *disj*: $\{\{x\}\} \cap \{\{v\} \mid v. v \in F\} = \{\}$ **by** *auto*
have $\{\{v\} \mid v. v \in \text{insert } x F\} = (\{\{x\}\} \cup \{\{v\} \mid v. v \in F\})$ **by** *auto*
then have $\text{card } \{\{v\} \mid v. v \in \text{insert } x F\} = \text{card } (\{\{x\}\} \cup \{\{v\} \mid v. v \in F\})$ **by**
simp
also have $\dots = \text{card } \{\{x\}\} + \text{card } \{\{v\} \mid v. v \in F\}$ **using** *card-Un-disjoint disj*
assms finite-subset
using *insert.hyps(1)* **by** *force*
also have $\dots = 1 + \text{card } \{\{v\} \mid v. v \in F\}$ **using** *is-singleton-altdef* **by** *simp*
also have $\dots = 1 + \text{card } F$ **using** *insert.hyps* **by** *auto*
finally show *?case* **using** *insert.hyps(1) insert.hyps(2)* **by** *force*
qed

lemma *finite-all-edges-loops*: $\text{finite } S \implies \text{finite } (\text{all-edges-loops } S)$
unfolding *all-edges-loops-def* **using** *finite-all-edges finite-singletons* **by** *auto*

lemma *card-all-edges-loops*:
assumes *finite S*
shows $\text{card } (\text{all-edges-loops } S) = (\text{card } S) \text{ choose } 2 + \text{card } S$
proof –
have $\text{card } (\text{all-edges-loops } S) = \text{card } (\text{all-edges } S \cup \{\{v\} \mid v. v \in S\})$
by (*simp add: all-edges-loops-def*)
also have $\dots = \text{card } (\text{all-edges } S) + \text{card } \{\{v\} \mid v. v \in S\}$
using *loops-disjoint assms card-Un-disjoint[of all-edges S {\{v\} \mid v. v \in S}]*
all-edges-loops-ss finite-all-edges-loops finite-subset **by** *fastforce*
also have $\dots = (\text{card } S) \text{ choose } 2 + \text{card } \{\{v\} \mid v. v \in S\}$ **by** (*simp add: card-all-edges assms*)
finally show *?thesis* **using** *assms card-singletons* **by** *auto*
qed

1.3 Graph System Locale

A generic incidence set system re-labeled to graph notation, where repeated edges are not allowed. All the definitions here do not need the "edge" size to be constrained to make sense.

```
locale graph-system =  
  fixes vertices :: 'a set (V)  
  fixes edges :: 'a edge set (E)  
  assumes wellformed:  $e \in E \implies e \subseteq V$   
begin
```

```
abbreviation order :: nat where  
order  $\equiv$  card (V)
```

```
abbreviation graph-size :: nat where  
graph-size  $\equiv$  card E
```

```
definition incident :: 'a  $\Rightarrow$  'a edge  $\Rightarrow$  bool where  
incident  $v\ e \equiv v \in e$ 
```

```
lemma incident-edge-in-wf:  $e \in E \implies \text{incident } v\ e \implies v \in V$   
using wellformed incident-def by auto
```

```
definition incident-edges :: 'a  $\Rightarrow$  'a edge set where  
incident-edges  $v \equiv \{e . e \in E \wedge \text{incident } v\ e\}$ 
```

```
lemma incident-edges-empty:  $\neg (v \in V) \implies \text{incident-edges } v = \{\}$   
using incident-edges-def incident-edge-in-wf by auto
```

```
lemma finite-incident-edges:  $\text{finite } E \implies \text{finite } (\text{incident-edges } v)$   
by (simp add: incident-edges-def)
```

```
definition edge-adj :: 'a edge  $\Rightarrow$  'a edge  $\Rightarrow$  bool where  
edge-adj  $e1\ e2 \equiv e1 \cap e2 \neq \{\} \wedge e1 \in E \wedge e2 \in E$ 
```

```
lemma edge-adj-inE:  $\text{edge-adj } e1\ e2 \implies e1 \in E \wedge e2 \in E$   
using edge-adj-def by auto
```

```
lemma edge-adjacent-alt-def:  $e1 \in E \implies e2 \in E \implies \exists x . x \in V \wedge x \in e1 \wedge x \in e2 \implies \text{edge-adj } e1\ e2$   
unfolding edge-adj-def by auto
```

```
lemma wellformed-alt-fst:  $\{x, y\} \in E \implies x \in V$   
using wellformed by auto
```

```
lemma wellformed-alt-snd:  $\{x, y\} \in E \implies y \in V$   
using wellformed by auto  
end
```


Simple constraints on a graph system may include finite and non-empty constraints

```
locale fin-graph-system = graph-system +
  assumes finV: finite V
begin
```

```
lemma fin-edges: finite E
  using wellformed finV
  by (meson PowI finite-Pow-iff finite-subset subsetI)
```

```
end
```

```
locale ne-graph-system = graph-system +
  assumes not-empty:  $V \neq \{\}$ 
```

1.4 Undirected Graph with Loops

This formalisation models a loop by a singleton set. In this case a graph has the edge size criteria if it has edges of size 1 or 2. Notably this removes the option for an edge to be empty

```
locale ulgraph = graph-system +
  assumes edge-size:  $e \in E \implies \text{card } e > 0 \wedge \text{card } e \leq 2$ 
```

```
begin
```

```
lemma alt-edge-size:  $e \in E \implies \text{card } e = 1 \vee \text{card } e = 2$ 
  using edge-size by fastforce
```

```
definition is-loop:: 'a edge  $\Rightarrow$  bool where
is-loop e  $\equiv \text{card } e = 1$ 
```

```
definition is-sedge :: 'a edge  $\Rightarrow$  bool where
is-sedge e  $\equiv \text{card } e = 2$ 
```

```
lemma is-edge-or-loop:  $e \in E \implies \text{is-loop } e \vee \text{is-sedge } e$ 
  using alt-edge-size is-loop-def is-sedge-def by simp
```

```
lemma edges-split-loop:  $E = \{e \in E . \text{is-loop } e\} \cup \{e \in E . \text{is-sedge } e\}$ 
  using is-edge-or-loop by auto
```

```
lemma edges-split-loop-inter-empty:  $\{\} = \{e \in E . \text{is-loop } e\} \cap \{e \in E . \text{is-sedge } e\}$ 
  unfolding is-loop-def is-sedge-def by auto
```

```
definition vert-adj :: 'a  $\Rightarrow$  'a  $\Rightarrow$  bool where — Neighbor in graph from Roth [1]
vert-adj v1 v2  $\equiv \{v1, v2\} \in E$ 
```

```
lemma vert-adj-sym:  $\text{vert-adj } v1 v2 \longleftrightarrow \text{vert-adj } v2 v1$ 
```

unfolding *vert-adj-def* **by** (*simp-all add: insert-commute*)

lemma *vert-adj-imp-inV*: $\text{vert-adj } v1 \ v2 \implies v1 \in V \wedge v2 \in V$
using *vert-adj-def wellformed* **by** *auto*

lemma *vert-adj-inc-edge-iff*: $\text{vert-adj } v1 \ v2 \longleftrightarrow \text{incident } v1 \ \{v1, v2\} \wedge \text{incident } v2 \ \{v1, v2\} \wedge \{v1, v2\} \in E$
unfolding *vert-adj-def incident-def* **by** *auto*

lemma *not-vert-adj[simp]*: $\neg \text{vert-adj } v \ u \implies \{v, u\} \notin E$
by (*simp add: vert-adj-def*)

definition *neighborhood* :: $'a \Rightarrow 'a \text{ set}$ **where** — Neighbors in Roth Development [1]
 $\text{neighborhood } x \equiv \{v \in V . \text{vert-adj } x \ v\}$

lemma *neighborhood-incident*: $u \in \text{neighborhood } v \longleftrightarrow \{u, v\} \in \text{incident-edges } v$
unfolding *neighborhood-def incident-edges-def*
by (*smt (verit) incident-def insert-commute insert-subset mem-Collect-eq subset-insertI vert-adj-def wellformed*)

definition *neighbors-ss* :: $'a \Rightarrow 'a \text{ set} \Rightarrow 'a \text{ set}$ **where**
 $\text{neighbors-ss } x \ Y \equiv \{y \in Y . \text{vert-adj } x \ y\}$

lemma *vert-adj-edge-iff2*:
assumes $v1 \neq v2$
shows $\text{vert-adj } v1 \ v2 \longleftrightarrow (\exists e \in E . \text{incident } v1 \ e \wedge \text{incident } v2 \ e)$
proof (*intro iffI*)
show $\text{vert-adj } v1 \ v2 \implies \exists e \in E . \text{incident } v1 \ e \wedge \text{incident } v2 \ e$ **using** *vert-adj-inc-edge-iff*
by *blast*
assume $\exists e \in E . \text{incident } v1 \ e \wedge \text{incident } v2 \ e$
then obtain e **where** $\text{ein}: e \in E$ **and** $\text{incident } v1 \ e$ **and** $\text{incident } v2 \ e$
using *vert-adj-inc-edge-iff* *assms alt-edge-size* **by** *auto*
then have $e = \{v1, v2\}$ **using** *alt-edge-size assms*
by (*smt (verit) card-1-singletonE card-2-iff incident-def insertE insert-commute singletonD*)
then show $\text{vert-adj } v1 \ v2$ **using** ein *vert-adj-def*
by *simp*
qed

Incident simple edges, i.e. excluding loops

definition *incident-sedges* :: $'a \Rightarrow 'a \text{ edge set}$ **where**
 $\text{incident-sedges } v \equiv \{e \in E . \text{incident } v \ e \wedge \text{card } e = 2\}$

lemma *finite-inc-sedges*: $\text{finite } E \implies \text{finite } (\text{incident-sedges } v)$
by (*simp add: incident-sedges-def*)

lemma *incident-sedges-empty[simp]*: $v \notin V \implies \text{incident-sedges } v = \{\}$
unfolding *incident-sedges-def* **using** *incident-def wellformed* **by** *fastforce*

definition *has-loop* :: 'a ⇒ bool **where**

has-loop v ≡ {v} ∈ E

lemma *has-loop-in-verts*: *has-loop* v ⇒ v ∈ V

using *has-loop-def wellformed* **by** *auto*

lemma *is-loop-set-alt*: { {v} | v . *has-loop* v } = { e ∈ E . *is-loop* e }

proof (*intro subset-antisym subsetI*)

fix x **assume** x ∈ { {v} | v . *has-loop* v }

then obtain v **where** x = {v} **and** *has-loop* v

by *blast*

then show x ∈ { e ∈ E . *is-loop* e } **using** *has-loop-def is-loop-def* **by** *auto*

next

fix x **assume** a: x ∈ { e ∈ E . *is-loop* e }

then have *is-loop* x **by** *blast*

then obtain v **where** x = {v} **and** {v} ∈ E **using** *is-loop-def a*

by (*metis card-1-singletonE mem-Collect-eq*)

thus x ∈ { {v} | v . *has-loop* v } **using** *has-loop-def* **by** *simp*

qed

definition *incident-loops* :: 'a ⇒ 'a edge set **where**

incident-loops v ≡ { e ∈ E . e = {v} }

lemma *card1-incident-imp-vert*: *incident* v e ∧ *card* e = 1 ⇒ e = {v}

by (*metis card-1-singletonE incident-def singleton-iff*)

lemma *incident-loops-alt*: *incident-loops* v = { e ∈ E . *incident* v e ∧ *card* e = 1 }

unfolding *incident-loops-def* **using** *card1-incident-imp-vert incident-def* **by** *auto*

lemma *incident-loops-simp*: *has-loop* v ⇒ *incident-loops* v = { {v} } ∩ *has-loop* v
⇒ *incident-loops* v = { }

unfolding *incident-loops-def has-loop-def* **by** *auto*

lemma *incident-loops-union*: $\bigcup (\textit{incident-loops} \textit{ ` } V) = \{ e \in E . \textit{is-loop} \ e \}$

proof –

have V = { v ∈ V . *has-loop* v } ∪ { v ∈ V . ¬ *has-loop* v }

by *auto*

then have $\bigcup (\textit{incident-loops} \textit{ ` } V) = \bigcup (\textit{incident-loops} \textit{ ` } \{ v \in V . \textit{has-loop} \ v \})$

∪

$\bigcup (\textit{incident-loops} \textit{ ` } \{ v \in V . \neg \textit{has-loop} \ v \})$ **by** *auto*

also have ... = $\bigcup (\textit{incident-loops} \textit{ ` } \{ v \in V . \textit{has-loop} \ v \})$ **using** *incident-loops-simp(2)*

by *simp*

also have ... = $\bigcup (\{ \{ v \} \} \mid v . \textit{has-loop} \ v)$ **using** *has-loop-in-verts incident-loops-simp(1)* **by** *auto*

also have ... = ({ {v} } | v . *has-loop* v) **by** *auto*

finally show *?thesis* **using** *is-loop-set-alt* **by** *simp*

qed

lemma *finite-incident-loops*: *finite (incident-loops v)*
using *incident-loops-simp* **by** (*cases has-loop v*) *auto*

lemma *incident-loops-card*: *card (incident-loops v) ≤ 1*
by (*cases has-loop v*) (*simp-all add: incident-loops-simp*)

lemma *incident-edges-union*: *incident-edges v = incident-sedges v ∪ incident-loops v*
unfolding *incident-edges-def incident-sedges-def incident-loops-alt* **using** *alt-edge-size*
by *auto*

lemma *incident-edges-sedges[simp]*: $\neg \text{has-loop } v \implies \text{incident-edges } v = \text{incident-sedges } v$
using *incident-edges-union incident-loops-simp* **by** *auto*

lemma *incident-sedges-union*: $\bigcup (\text{incident-sedges } `V) = \{e \in E . \text{is-sedge } e\}$
proof (*intro subset-antisym subsetI*)
fix *x* **assume** $x \in \bigcup (\text{incident-sedges } `V)$
then obtain *v* **where** $x \in \text{incident-sedges } v$ **by** *blast*
then show $x \in \{e \in E . \text{is-sedge } e\}$ **using** *incident-sedges-def is-sedge-def* **by**
auto
next
fix *x* **assume** $x \in \{e \in E . \text{is-sedge } e\}$
then have *xin*: $x \in E$ **and** *c2*: *card x = 2* **using** *is-sedge-def* **by** *auto*
then obtain *v* **where** $v \in x$ **and** *vin*: $v \in V$ **using** *wellformed*
by (*meson card-2-iff' subsetD*)
then have $x \in \text{incident-sedges } v$ **unfolding** *incident-sedges-def incident-def*
using *xin c2* **by** *auto*
then show $x \in \bigcup (\text{incident-sedges } `V)$ **using** *vin* **by** *auto*
qed

lemma *empty-not-edge*: $\{\} \notin E$
using *edge-size* **by** *fastforce*

The degree definition is complicated by loops - each loop contributes two to degree. This is required for basic counting properties on the degree to hold

definition *degree* :: $'a \Rightarrow \text{nat}$ **where**
degree v $\equiv \text{card } (\text{incident-sedges } v) + 2 * (\text{card } (\text{incident-loops } v))$

lemma *degree-no-loops[simp]*: $\neg \text{has-loop } v \implies \text{degree } v = \text{card } (\text{incident-edges } v)$
using *incident-edges-sedges degree-def incident-loops-simp(2)* **by** *auto*

lemma *degree-none[simp]*: $\neg v \in V \implies \text{degree } v = 0$
using *degree-def degree-no-loops has-loop-in-verts incident-edges-sedges incident-sedges-empty*
by *auto*

lemma *degree0-inc-edges-empt-iff*:

assumes *finite E*
shows $\text{degree } v = 0 \longleftrightarrow \text{incident-edges } v = \{\}$
proof (*intro iffI*)
assume $\text{degree } v = 0$
then have $\text{card } (\text{incident-sedges } v) + 2 * (\text{card } (\text{incident-loops } v)) = 0$ **using**
degree-def by simp
then have $\text{incident-sedges } v = \{\}$ **and** $\text{incident-loops } v = \{\}$
using *degree-def incident-edges-union assms finite-incident-edges finite-incident-loops*
by auto
thus $\text{incident-edges } v = \{\}$ **using** *incident-edges-union by auto*
next
show $\text{incident-edges } v = \{\} \implies \text{degree } v = 0$ **using** *incident-edges-union degree-def*
by simp
qed

lemma *incident-edges-neighbors-img*: $\text{incident-edges } v = (\lambda u . \{v, u\}) \text{ ` } (\text{neighborhood } v)$
proof (*intro subset-antisym subsetI*)
fix x **assume** $a: x \in \text{incident-edges } v$
then have $x \in E$ **and** $v \in x$ **using** *incident-edges-def incident-def by auto*
then obtain u **where** $x = \{u, v\}$ **using** *alt-edge-size*
by (*smt (verit, best) card-1-singletonE card-2-iff insertE insert-absorb2 insert-commute singletonD*)
then have $u \in \text{neighborhood } v$
using *a neighborhood-incident by blast*
then show $x \in (\lambda u . \{v, u\}) \text{ ` } \text{neighborhood } v$ **using** $\langle x = \{u, v\} \rangle$ **by blast**
next
fix x **assume** $x \in (\lambda u . \{v, u\}) \text{ ` } \text{neighborhood } v$
then obtain u' **where** $x = \{v, u'\}$ **and** $u' \in \text{neighborhood } v$
by blast
then show $x \in \text{incident-edges } v$
by (*simp add: insert-commute neighborhood-incident*)
qed

lemma *card-incident-sedges-neighborhood*: $\text{card } (\text{incident-edges } v) = \text{card } (\text{neighborhood } v)$
proof –
have *bij-betw* $(\lambda u . \{v, u\}) (\text{neighborhood } v) (\text{incident-edges } v)$
by (*intro bij-betw-imageI inj-onI, simp-all add:incident-edges-neighbors-img*)(*metis doubleton-eq-iff*)
thus *?thesis*
by (*metis bij-betw-same-card*)
qed

lemma *degree0-neighborhood-empt-iff*:
assumes *finite E*
shows $\text{degree } v = 0 \longleftrightarrow \text{neighborhood } v = \{\}$

using *degree0-inc-edges-empt-iff incident-edges-neighbors-imp*
by (*simp add: assms*)

definition *is-isolated-vertex*:: 'a \Rightarrow bool **where**
is-isolated-vertex v \equiv v \in V \wedge (\forall u \in V . \neg vert-adj u v)

lemma *is-isolated-vertex-edge*: *is-isolated-vertex* v \Longrightarrow (\bigwedge e. e \in E \Longrightarrow \neg (incident v e))

unfolding *is-isolated-vertex-def*

by (*metis (full-types) all-not-in-conv incident-def insert-absorb insert-iff mk-disjoint-insert*

vert-adj-def vert-adj-edge-iff2 vert-adj-imp-inV)

lemma *is-isolated-vertex-no-loop*: *is-isolated-vertex* v \Longrightarrow \neg has-loop v

unfolding *has-loop-def is-isolated-vertex-def vert-adj-def* **by** auto

lemma *is-isolated-vertex-degree0*: *is-isolated-vertex* v \Longrightarrow degree v = 0

proof –

assume *assm*: *is-isolated-vertex* v

then have \neg has-loop v **using** *is-isolated-vertex-no-loop* **by** *simp*

then have degree v = card (incident-edges v) **using** *degree-no-loops* **by** auto

moreover have \bigwedge e. e \in E \Longrightarrow \neg (incident v e)

using *is-isolated-vertex-edge* *assm* **by** auto

then have (incident-edges v) = {} **unfolding** *incident-edges-def* **by** auto

ultimately show degree v = 0 **by** *simp*

qed

lemma *iso-vertex-empty-neighborhood*: *is-isolated-vertex* v \Longrightarrow neighborhood v = {}

using *is-isolated-vertex-def neighborhood-def*

by (*metis (mono-tags, lifting) Collect-empty-eq is-isolated-vertex-edge vert-adj-inc-edge-iff*)

definition *max-degree* :: nat **where**

max-degree \equiv Max {degree v | v. v \in V}

definition *min-degree* :: nat **where**

min-degree \equiv Min {degree v | v . v \in V}

definition *is-edge-between* :: 'a set \Rightarrow 'a set \Rightarrow 'a edge \Rightarrow bool **where**

is-edge-between X Y e \equiv \exists x y. e = {x, y} \wedge x \in X \wedge y \in Y

All edges between two sets of vertices, X and Y, in a graph, G. Inspired by Szemerédi development [2] and generalised here

definition *all-edges-between* :: 'a set \Rightarrow 'a set \Rightarrow ('a \times 'a) set **where**

all-edges-between X Y \equiv {(x, y) . x \in X \wedge y \in Y \wedge {x, y} \in E}

lemma *all-edges-betw-D3*: (x, y) \in *all-edges-between* X Y \Longrightarrow {x, y} \in E

by (*simp add: all-edges-between-def*)

lemma *all-edges-betw-I*: $x \in X \implies y \in Y \implies \{x, y\} \in E \implies (x, y) \in \text{all-edges-between } X Y$
by (*simp add: all-edges-between-def*)

lemma *all-edges-between-subset*: $\text{all-edges-between } X Y \subseteq X \times Y$
by (*auto simp: all-edges-between-def*)

lemma *all-edges-between-E-ss*: $\text{mk-edge } \text{all-edges-between } X Y \subseteq E$
by (*auto simp add: all-edges-between-def*)

lemma *all-edges-between-rem-wf*: $\text{all-edges-between } X Y = \text{all-edges-between } (X \cap V) (Y \cap V)$
using *wellformed* **by** (*simp add: all-edges-between-def*) *blast*

lemma *all-edges-between-empty* [*simp*]:
 $\text{all-edges-between } \{ \} Z = \{ \}$ $\text{all-edges-between } Z \{ \} = \{ \}$
by (*auto simp: all-edges-between-def*)

lemma *all-edges-between-disjnt1*: $\text{disjnt } X Y \implies \text{disjnt } (\text{all-edges-between } X Z) (\text{all-edges-between } Y Z)$
by (*auto simp: all-edges-between-def disjnt-iff*)

lemma *all-edges-between-disjnt2*: $\text{disjnt } Y Z \implies \text{disjnt } (\text{all-edges-between } X Y) (\text{all-edges-between } X Z)$
by (*auto simp: all-edges-between-def disjnt-iff*)

lemma *max-all-edges-between*:
assumes *finite X finite Y*
shows $\text{card } (\text{all-edges-between } X Y) \leq \text{card } X * \text{card } Y$
by (*metis assms card-mono finite-SigmaI all-edges-between-subset card-cartesian-product*)

lemma *all-edges-between-Un1*:
 $\text{all-edges-between } (X \cup Y) Z = \text{all-edges-between } X Z \cup \text{all-edges-between } Y Z$
by (*auto simp: all-edges-between-def*)

lemma *all-edges-between-Un2*:
 $\text{all-edges-between } X (Y \cup Z) = \text{all-edges-between } X Y \cup \text{all-edges-between } X Z$
by (*auto simp: all-edges-between-def*)

lemma *finite-all-edges-between*:
assumes *finite X finite Y*
shows *finite (all-edges-between X Y)*
by (*meson all-edges-between-subset assms finite-cartesian-product finite-subset*)

lemma *all-edges-between-Union1*:
 $\text{all-edges-between } (\text{Union } \mathcal{X}) Y = \bigcup_{X \in \mathcal{X}} \text{all-edges-between } X Y$
by (*auto simp: all-edges-between-def*)

lemma *all-edges-between-Union2*:

all-edges-between X (*Union* \mathcal{Y}) = $(\bigcup Y \in \mathcal{Y}. \text{all-edges-between } X Y)$

by (*auto simp: all-edges-between-def*)

lemma *all-edges-between-disjoint1*:

assumes *disjoint* R

shows *disjoint* $((\lambda X. \text{all-edges-between } X Y) \text{ ' } R)$

using *assms by (auto simp: all-edges-between-def disjoint-def)*

lemma *all-edges-between-disjoint2*:

assumes *disjoint* R

shows *disjoint* $((\lambda Y. \text{all-edges-between } X Y) \text{ ' } R)$

using *assms by (auto simp: all-edges-between-def disjoint-def)*

lemma *all-edges-between-disjoint-family-on1*:

assumes *disjoint* R

shows *disjoint-family-on* $(\lambda X. \text{all-edges-between } X Y) R$

by (*metis (no-types, lifting) all-edges-between-disjnt1 assms disjnt-def disjoint-family-on-def pairwiseD*)

lemma *all-edges-between-disjoint-family-on2*:

assumes *disjoint* R

shows *disjoint-family-on* $(\lambda Y. \text{all-edges-between } X Y) R$

by (*metis (no-types, lifting) all-edges-between-disjnt2 assms disjnt-def disjoint-family-on-def pairwiseD*)

lemma *all-edges-between-mono1*:

$Y \subseteq Z \implies \text{all-edges-between } Y X \subseteq \text{all-edges-between } Z X$

by (*auto simp: all-edges-between-def*)

lemma *all-edges-between-mono2*:

$Y \subseteq Z \implies \text{all-edges-between } X Y \subseteq \text{all-edges-between } X Z$

by (*auto simp: all-edges-between-def*)

lemma *inj-on-mk-edge*: $X \cap Y = \{\} \implies \text{inj-on mk-edge } (\text{all-edges-between } X Y)$

by (*auto simp: inj-on-def doubleton-eq-iff all-edges-between-def*)

lemma *all-edges-between-subset-times*: $\text{all-edges-between } X Y \subseteq (X \cap \bigcup E) \times (Y \cap \bigcup E)$

by (*auto simp: all-edges-between-def*)

lemma *all-edges-betw-prod-def-neighbors*: $\text{all-edges-between } X Y = \{(x, y) \in X \times Y . \text{vert-adj } x y\}$

by (*auto simp: vert-adj-def all-edges-between-def*)

lemma *all-edges-betw-sigma-neighbor*:

$\text{all-edges-between } X Y = (\text{SIGMA } x:X. \text{neighbors-ss } x Y)$

by (*auto simp add: all-edges-between-def neighbors-ss-def vert-adj-def*)

lemma *card-all-edges-betw-neighbor*:
assumes *finite X finite Y*
shows $\text{card } (\text{all-edges-between } X \ Y) = (\sum x \in X. \text{card } (\text{neighbors-ss } x \ Y))$
using *all-edges-betw-sigma-neighbor assms* **by** (*simp add: neighbors-ss-def*)

lemma *all-edges-between-swap*:
 $\text{all-edges-between } X \ Y = (\lambda(x,y). (y,x)) \ ` (\text{all-edges-between } Y \ X)$
unfolding *all-edges-between-def*
by (*auto simp add: insert-commute image-iff split: prod.split*)

lemma *card-all-edges-between-commute*:
 $\text{card } (\text{all-edges-between } X \ Y) = \text{card } (\text{all-edges-between } Y \ X)$
proof –
have *inj-on* $(\lambda(x, y). (y, x)) \ A$ **for** $A :: (\text{nat} * \text{nat}) \text{set}$
by (*auto simp: inj-on-def*)
then show *?thesis* **using** *all-edges-between-swap [of X Y] card-image*
by (*metis swap-inj-on*)
qed

lemma *all-edges-between-set: mk-edge* ‘ $\text{all-edges-between } X \ Y = \{\{x, y\} \mid x \ y. x \in X \wedge y \in Y \wedge \{x, y\} \in E\}$
unfolding *all-edges-between-def*
proof (*intro subset-antisym subsetI*)
fix e **assume** $e \in \text{mk-edge } \{ \{(x, y). x \in X \wedge y \in Y \wedge \{x, y\} \in E\}$
then obtain $x \ y$ **where** $e = \text{mk-edge } (x, y)$ **and** $x \in X$ **and** $y \in Y$ **and** $\{x, y\} \in E$
by *blast*
then show $e \in \{\{x, y\} \mid x \ y. x \in X \wedge y \in Y \wedge \{x, y\} \in E\}$
by *auto*
next
fix e **assume** $e \in \{\{x, y\} \mid x \ y. x \in X \wedge y \in Y \wedge \{x, y\} \in E\}$
then obtain $x \ y$ **where** $e = \{x, y\}$ **and** $x \in X$ **and** $y \in Y$ **and** $\{x, y\} \in E$
by *blast*
then have $e = \text{mk-edge } (x, y)$
by *auto*
then show $e \in \text{mk-edge } \{ \{(x, y). x \in X \wedge y \in Y \wedge \{x, y\} \in E\}$
using $\langle x \in X \rangle \langle y \in Y \rangle \langle \{x, y\} \in E \rangle$ **by** *blast*
qed

1.5 Edge Density

The edge density between two sets of vertices, X and Y , in G . This is the same definition as taken in the Szemerédi development, generalised here [2]

definition *edge-density* $X \ Y \equiv \text{card } (\text{all-edges-between } X \ Y) / (\text{card } X * \text{card } Y)$

lemma *edge-density-ge0*: $\text{edge-density } X \ Y \geq 0$
by (*auto simp: edge-density-def*)

lemma *edge-density-le1*: $\text{edge-density } X \ Y \leq 1$
proof (*cases finite X ∧ finite Y*)

```

case True
then show ?thesis
  using of-nat-mono [OF max-all-edges-between, of X Y]
  by (fastforce simp add: edge-density-def divide-simps)
qed (auto simp: edge-density-def)

lemma edge-density-zero:  $Y = \{\}$   $\implies$   $\text{edge-density } X \ Y = 0$ 
  by (simp add: edge-density-def)

lemma edge-density-commute:  $\text{edge-density } X \ Y = \text{edge-density } Y \ X$ 
  by (simp add: edge-density-def card-all-edges-between-commute mult.commute)

lemma edge-density-Un:
  assumes disjnt X1 X2 finite X1 finite X2 finite Y
  shows  $\text{edge-density } (X1 \cup X2) \ Y = (\text{edge-density } X1 \ Y * \text{card } X1 + \text{edge-density } X2 \ Y * \text{card } X2) / (\text{card } X1 + \text{card } X2)$ 
  using assms unfolding edge-density-def
  by (simp add: all-edges-between-disjnt1 all-edges-between-Un1 finite-all-edges-between card-Un-disjnt divide-simps)

lemma edge-density-eq0:
  assumes all-edges-between A B =  $\{\}$  and  $X \subseteq A$   $Y \subseteq B$ 
  shows  $\text{edge-density } X \ Y = 0$ 
proof –
  have all-edges-between X Y =  $\{\}$ 
  by (metis all-edges-between-mono1 all-edges-between-mono2 assms subset-empty)
  then show ?thesis
  by (auto simp: edge-density-def)
qed

end

```

A number of lemmas are limited to a finite graph

```

locale fin-ulgraph = ulgraph + fin-graph-system
begin

```

```

lemma card-is-has-loop-eq:  $\text{card } \{e \in E . \text{is-loop } e\} = \text{card } \{v \in V . \text{has-loop } v\}$ 
proof –
  have  $\bigwedge e . e \in E \implies \text{is-loop } e \longleftrightarrow (\exists v . e = \{v\})$  using is-loop-def
  using is-singleton-altdef is-singleton-def by blast
  define  $f :: 'a \Rightarrow 'a \text{ set}$  where  $f = (\lambda v . \{v\})$ 
  have  $\text{feq}: f \text{ ` } \{v \in V . \text{has-loop } v\} = \{\{v\} \mid v . \text{has-loop } v\}$  using has-loop-in-verts
  f-def by auto
  have inj-on  $f \text{ ` } \{v \in V . \text{has-loop } v\}$  by (simp add: f-def)
  then have  $\text{card } \{v \in V . \text{has-loop } v\} = \text{card } (f \text{ ` } \{v \in V . \text{has-loop } v\})$ 
  using card-image by fastforce
  also have  $\dots = \text{card } \{\{v\} \mid v . \text{has-loop } v\}$  using feq by simp
  finally have  $\text{card } \{v \in V . \text{has-loop } v\} = \text{card } \{e \in E . \text{is-loop } e\}$  using
is-loop-set-alt by simp

```

thus $\text{card } \{e \in E . \text{is-loop } e\} = \text{card } \{v \in V . \text{has-loop } v\}$ **by** *simp*
qed

lemma *finite-all-edges-between'*: $\text{finite } (\text{all-edges-between } X Y)$
using *finV wellformed*
by (*metis all-edges-between-rem-wf finite-Int finite-all-edges-between*)

lemma *card-all-edges-between*:
assumes *finite Y*
shows $\text{card } (\text{all-edges-between } X Y) = (\sum y \in Y. \text{card } (\text{all-edges-between } X \{y\}))$
proof –
have $\text{all-edges-between } X Y = (\bigcup y \in Y. \text{all-edges-between } X \{y\})$
by (*auto simp: all-edges-between-def*)
moreover have *disjoint-family-on* $(\lambda y. \text{all-edges-between } X \{y\}) Y$
unfolding *disjoint-family-on-def*
by (*auto simp: disjoint-family-on-def all-edges-between-def*)
ultimately show *?thesis*
by (*simp add: card-UN-disjoint' assms finite-all-edges-between'*)
qed

end

1.6 Simple Graphs

A simple graph (or sgraph) constrains edges to size of two. This is the classic definition of an undirected graph

locale *sgraph* = *graph-system* +
assumes *two-edges*: $e \in E \implies \text{card } e = 2$
begin

lemma *wellformed-all-edges*: $E \subseteq \text{all-edges } V$
unfolding *all-edges-def* **using** *wellformed two-edges* **by** *auto*

lemma *e-in-all-edges*: $e \in E \implies e \in \text{all-edges } V$
using *wellformed-all-edges* **by** *auto*

lemma *e-in-all-edges-ss*: $e \in E \implies e \subseteq V' \implies V' \subseteq V \implies e \in \text{all-edges } V'$
unfolding *all-edges-def* **using** *wellformed two-edges* **by** *auto*

lemma *singleton-not-edge*: $\{x\} \notin E$ — Suggested by Mantas Baksys
using *two-edges* **by** *fastforce*

end

It is easy to proof that *sgraph* is a sublocale of *ulgraph*. By using indirect inheritance, we avoid two unneeded cardinality conditions

sublocale *sgraph* \subseteq *ulgraph* $V E$
by (*unfold-locales*)(*simp add: two-edges*)

locale *fin-sgraph* = *sgraph* + *fin-graph-system*
begin

lemma *fin-neighborhood*: *finite (neighborhood x)*
unfolding *neighborhood-def* **using** *finV* **by** *simp*

lemma *fin-all-edges*: *finite (all-edges V)*
unfolding *all-edges-def* **by** (*simp add: finV*)

lemma *max-edges-graph*: $\text{card } E \leq (\text{card } V)^2$

proof –

have $\text{card } E \leq \text{card } V$ *choose 2*

by (*metis fin-all-edges finV card-all-edges card-mono wellformed-all-edges*)

thus *?thesis*

by (*metis binomial-le-pow le0 neq0-conv order.trans zero-less-binomial-iff*)

qed

end

sublocale *fin-sgraph* \subseteq *fin-ulgraph*
by (*unfold-locales*)

context *sgraph*
begin

lemma *no-loops*: $v \in V \implies \neg \text{has-loop } v$
using *has-loop-def two-edges* **by** *fastforce*

Ideally, we'd redefine degree in the context of a simple graph. However, this requires a named loop locale, which complicates notation unnecessarily. This is the lemma that should always be used when unfolding the degree definition in a simple graph context

lemma *alt-degree-def[simp]*: $\text{degree } v = \text{card } (\text{incident-edges } v)$

using *no-loops degree-no-loops degree-none incident-edges-empty* **by** (*cases v \in V*) *simp-all*

lemma *alt-deg-neighborhood*: $\text{degree } v = \text{card } (\text{neighborhood } v)$

using *card-incident-sedges-neighborhood* **by** *simp*

definition *degree-set* :: 'a set \Rightarrow nat **where**
degree-set vs $\equiv \text{card } \{e \in E. \text{vs} \subseteq e\}$

definition *is-complete-n-graph*:: nat \Rightarrow bool **where**
is-complete-n-graph n $\equiv \text{order} = n \wedge E = \text{all-edges } V$

The complement of a graph is a basic concept

definition *is-complement* :: 'a pregraph \Rightarrow bool **where**
is-complement G $\equiv V = \text{gverts } G \wedge \text{gedges } G = \text{all-edges } V - E$

definition *complement-edges* :: 'a edge set **where**
complement-edges \equiv *all-edges* $V - E$

lemma *is-complement-edges*: *is-complement* (V', E') $\longleftrightarrow V = V' \wedge$ *complement-edges* $= E'$

unfolding *is-complement-def complement-edges-def* **by** *auto*

interpretation *G-comp*: *sgraph* V *complement-edges*

by (*unfold-locales*)(*auto simp add: complement-edges-def all-edges-def*)

lemma *is-complement-edge-iff*: $e \subseteq V \implies e \in$ *complement-edges* $\longleftrightarrow e \notin E \wedge$
card $e = 2$

unfolding *complement-edges-def all-edges-def* **by** *auto*

end

A complete graph is a simple graph

lemma *complete-sgraph*: *sgraph* S (*all-edges* S)

unfolding *all-edges-def* **by** (*unfold-locales*) (*simp-all*)

interpretation *comp-sgraph*: *sgraph* S (*all-edges* S)

using *complete-sgraph* **by** *auto*

lemma *complete-fin-sgraph*: *finite* $S \implies$ *fin-sgraph* S (*all-edges* S)

using *complete-sgraph*

by (*intro-locales*) (*auto simp add: sgraph.axioms(1) sgraph-def fin-graph-system-axioms-def*)

1.7 Subgraph Basics

A subgraph is defined as a graph where the vertex and edge sets are subsets of the original graph. Note that using the locale approach, we require each graph to be wellformed. This is interestingly omitted in a number of other formal definitions.

locale *subgraph* = H : *graph-system* V_H :: 'a set E_H + G : *graph-system* V_G :: 'a set E_G **for** $V_H E_H V_G E_G$ +

assumes *verts-ss*: $V_H \subseteq V_G$

assumes *edges-ss*: $E_H \subseteq E_G$

lemma *is-subgraphI[intro]*: $V' \subseteq V \implies E' \subseteq E \implies$ *graph-system* $V' E' \implies$
graph-system $V E \implies$ *subgraph* $V' E' V E$

using *graph-system-def* **by** (*unfold-locales*)

(*auto simp add: graph-system.incident-def graph-system.incident-edge-in-wf*)

context *subgraph*

begin

Note: it could also be useful to have similar rules in *ulgraph* locale etc with subgraph assumption

lemma *is-subgraph-ulgraph*:
assumes *ulgraph* $V_G E_G$
shows *ulgraph* $V_H E_H$
using *assms* *ulgraph.edge-size* [of $V_G E_G$] *edges-ss* **by** (*unfold-locales*) *auto*

lemma *is-simp-subgraph*:
assumes *sgraph* $V_G E_G$
shows *sgraph* $V_H E_H$
using *assms* *sgraph.two-edges* *edges-ss* **by** (*unfold-locales*) *auto*

lemma *is-finite-subgraph*:
assumes *fin-graph-system* $V_G E_G$
shows *fin-graph-system* $V_H E_H$
using *assms* *verts-ss*
by (*unfold-locales*) (*simp* *add*: *fin-graph-system.finV* *finite-subset*)

lemma (*in graph-system*) *subgraph-reft*: *subgraph* $V E V E$
by (*simp* *add*: *graph-system-axioms* *is-subgraphI*)

lemma *subgraph-trans*:
assumes *graph-system* $V E$
assumes *graph-system* $V' E'$
assumes *graph-system* $V'' E''$
shows *subgraph* $V'' E'' V' E' \implies$ *subgraph* $V' E' V E \implies$ *subgraph* $V'' E'' V E$
by (*meson* *assms*(1) *assms*(3) *is-subgraphI* *subgraph.edges-ss* *subgraph.verts-ss* *subset-trans*)

lemma *subgraph-antisym*: *subgraph* $V' E' V E \implies$ *subgraph* $V E V' E' \implies$ $V = V' \wedge E = E'$
by (*simp* *add*: *dual-order.eq-iff* *subgraph.edges-ss* *subgraph.verts-ss*)

end

lemma (*in sgraph*) *subgraph-complete*: *subgraph* $V E V$ (*all-edges* V)

proof –

interpret *comp*: *sgraph* V (*all-edges* V)

using *complete-sgraph* **by** *auto*

show *?thesis* **by** (*unfold-locales*) (*simp-all* *add*: *wellformed-all-edges*)

qed

We are often interested in the set of subgraphs. This is still very possible using locale definitions. Interesting Note - random graphs [3] has a different definition for the well formed constraint to be added in here instead of in the main subgraph definition

definition (*in graph-system*) *subgraphs*:: 'a *pregraph* set **where**
subgraphs $\equiv \{G . \text{subgraph } (gverts\ G) (gedges\ G) V E\}$

Induced subgraph - really only affects edges

definition (in *graph-system*) *induced-edges*:: 'a set \Rightarrow 'a edge set **where**
induced-edges $V' \equiv \{e \in E. e \subseteq V'\}$

lemma (in *sgraph*) *induced-edges-alt*: *induced-edges* $V' = E \cap$ *all-edges* V'
unfolding *induced-edges-def* *all-edges-def* **using** *two-edges* **by** *blast*

lemma (in *sgraph*) *induced-edges-self*: *induced-edges* $V = E$
unfolding *induced-edges-def*
by (*simp add: subsetI subset-antisym wellformed*)

context *graph-system*
begin

lemma *induced-edges-ss*: $V' \subseteq V \Longrightarrow$ *induced-edges* $V' \subseteq E$
unfolding *induced-edges-def* **by** *auto*

lemma *induced-is-graph-sys*: *graph-system* V' (*induced-edges* V')
by (*unfold-locales*) (*simp add: induced-edges-def*)

interpretation *induced-graph*: *graph-system* V' (*induced-edges* V')
using *induced-is-graph-sys* **by** *simp*

lemma *induced-is-subgraph*: $V' \subseteq V \Longrightarrow$ *subgraph* V' (*induced-edges* V') $V E$
using *induced-edges-ss* **by** (*unfold-locales*) *auto*

lemma *induced-edges-union*:
assumes $VH1 \subseteq S$ $VH2 \subseteq T$
assumes *graph-system* $VH1$ $EH1$ *graph-system* $VH2$ $EH2$
assumes $EH1 \cup EH2 \subseteq$ (*induced-edges* $(S \cup T)$)
shows $EH1 \subseteq$ (*induced-edges* S)
proof (*intro subsetI, simp add: induced-edges-def, intro conjI*)
show $\bigwedge x. x \in EH1 \Longrightarrow x \in E$ **using** *assms(5)*
by (*simp add: induced-edges-def subset-iff*)
show $\bigwedge x. x \in EH1 \Longrightarrow x \subseteq S$
using *assms(1) assms(3) graph-system.wellformed* **by** *blast*
qed

lemma *induced-edges-union-subgraph-single*:
assumes $VH1 \subseteq S$ $VH2 \subseteq T$
assumes *graph-system* $VH1$ $EH1$ *graph-system* $VH2$ $EH2$
assumes *subgraph* $(VH1 \cup VH2)$ $(EH1 \cup EH2)$ $(S \cup T)$ (*induced-edges* $(S \cup T)$)
shows *subgraph* $VH1$ $EH1$ S (*induced-edges* S)
proof –
interpret *ug*: *subgraph* $(VH1 \cup VH2)$ $(EH1 \cup EH2)$ $(S \cup T)$ (*induced-edges* $(S \cup T)$)
using *assms(5)* **by** *simp*
show *subgraph* $VH1$ $EH1$ S (*induced-edges* S)

using *assms(3) graph-system-def*
by (*unfold-locales*) (*blast, simp add: assms(1), meson assms induced-edges-union ug.edges-ss*)
qed

lemma *induced-union-subgraph*:
assumes $VH1 \subseteq S$ **and** $VH2 \subseteq T$
assumes *graph-system VH1 EH1 graph-system VH2 EH2*
shows *subgraph VH1 EH1 S (induced-edges S) \wedge subgraph VH2 EH2 T (induced-edges T) \longleftrightarrow*
subgraph (VH1 \cup VH2) (EH1 \cup EH2) (S \cup T) (induced-edges (S \cup T))
proof (*intro iffI conjI, elim conjE*)
show *subgraph (VH1 \cup VH2) (EH1 \cup EH2) (S \cup T) (induced-edges (S \cup T))*
 \implies *subgraph VH1 EH1 S (induced-edges S)*
using *induced-edges-union-subgraph-single assms by simp*
show *subgraph (VH1 \cup VH2) (EH1 \cup EH2) (S \cup T) (induced-edges (S \cup T))*
 \implies *subgraph VH2 EH2 T (induced-edges T)*
using *induced-edges-union-subgraph-single assms by (simp add: Un-commute)*
assume *a1: subgraph VH1 EH1 S (induced-edges S) and a2: subgraph VH2 EH2 T (induced-edges T)*
then interpret *h1: subgraph VH1 EH1 S (induced-edges S)*
by *simp*
interpret *h2: subgraph VH2 EH2 T (induced-edges T) using a2 by simp*
show *subgraph (VH1 \cup VH2) (EH1 \cup EH2) (S \cup T) (induced-edges (S \cup T))*
using *h1.H.wellformed h2.H.wellformed h1.verts-ss h2.verts-ss h1.edges-ss h2.edges-ss*
by (*unfold-locales*) (*auto simp add: induced-edges-def*)
qed

end
end

theory *Undirected-Graph-Walks imports Undirected-Graph-Basics*
begin

2 Walks, Paths and Cycles

The definition of walks, paths, cycles, and related concepts are foundations of graph theory, yet there can be some differences in literature between definitions. This formalisation draws inspiration from Noschinski's Graph Library [5], however focuses on an undirected graph context compared to a directed graph context, and extends on some definitions, as required to formalise Balog Szemerédi Gowers theorem.

context *ulgraph*
begin

2.1 Walks

This definition is taken from the directed graph library, however edges are undirected

```
fun walk-edges :: 'a list  $\Rightarrow$  'a edge list where
  walk-edges [] = []
| walk-edges [x] = []
| walk-edges (x # y # ys) = {x,y} # walk-edges (y # ys)
```

```
lemma walk-edges-app: walk-edges (xs @ [y, x]) = walk-edges (xs @ [y]) @ [{y, x}]
by (induct xs rule: walk-edges.induct, simp-all)
```

```
lemma walk-edges-tl-ss: set (walk-edges (tl xs))  $\subseteq$  set (walk-edges xs)
by (induct xs rule: walk-edges.induct) auto
```

```
lemma walk-edges-rev: rev (walk-edges xs) = walk-edges (rev xs)
```

```
proof (induct xs rule: walk-edges.induct, simp-all)
```

```
  fix x y ys assume assm: rev (walk-edges (y # ys)) = walk-edges (rev ys @ [y])
```

```
  then show walk-edges (rev ys @ [y]) @ [{x, y}] = walk-edges (rev ys @ [y, x])
```

```
    using walk-edges-app by fastforce
```

```
qed
```

```
lemma walk-edges-append-ss1: set (walk-edges (ys))  $\subseteq$  set (walk-edges (xs@ys))
```

```
proof (induct xs rule: walk-edges.induct)
```

```
  case 1
```

```
    then show ?case by simp
```

```
next
```

```
  case (2 x)
```

```
    then show ?case
```

```
      using walk-edges-tl-ss by fastforce
```

```
next
```

```
  case (3 x y ys)
```

```
    then show ?case by (simp add: subset-iff)
```

```
qed
```

```
lemma walk-edges-append-ss2: set (walk-edges (xs))  $\subseteq$  set (walk-edges (xs@ys))
```

```
by (induct xs rule: walk-edges.induct) auto
```

```
lemma walk-edges-singleton-app: ys  $\neq$  []  $\implies$  walk-edges ([x]@ys) = {x, hd ys} #
walk-edges ys
```

```
  using list.exhaust-sel walk-edges.simps(3) by (metis Cons-eq-appendI eq-Nil-appendI)
```

```
lemma walk-edges-append-union: xs  $\neq$  []  $\implies$  ys  $\neq$  []  $\implies$ 
```

```
  set (walk-edges (xs@ys)) = set (walk-edges (xs))  $\cup$  set (walk-edges ys)  $\cup$  {{last
xs, hd ys}}
```

```
  using walk-edges-singleton-app by (induct xs rule: walk-edges.induct) auto
```

lemma *walk-edges-decomp-ss*: $set (walk-edges (xs@[y]@zs)) \subseteq set (walk-edges (xs@[y]@ys@[y]@zs))$
proof –
 have *half-ss*: $set (walk-edges (xs@[y])) \subseteq set (walk-edges (xs@[y]@ys@[y]))$
 using *walk-edges-append-ss2* **by** *fastforce*
 thus *?thesis* **proof** (*cases* $zs = []$)
 case *True*
 then show *?thesis* **using** *half-ss* **by** *auto*
next
 case *False*
 then have *decomp1*: $set (walk-edges (xs@[y]@zs)) = set (walk-edges (xs@[y]))$
 $\cup set (walk-edges (zs)) \cup \{\{y, hd\ zs\}\}$
 using *walk-edges-append-union*
 by (*metis* *append-assoc* *append-is-Nil-conv* *last-snoc* *neq-Nil-conv*)
 have $set (walk-edges (xs@[y]@ys@[y]@zs)) = set (walk-edges (xs@[y]@ys@[y]))$
 $\cup set (walk-edges (zs)) \cup \{\{y, hd\ zs\}\}$
 using *walk-edges-append-union* *False*
 by (*metis* *append-assoc* *append-is-Nil-conv* *empty-iff* *empty-set* *last-snoc* *list.set-intros*(1))
 then show *?thesis* **using** *decomp1* *half-ss* **by** *auto*
qed
qed

definition *walk-length* :: 'a list \Rightarrow nat **where**
walk-length $p \equiv length (walk-edges\ p)$

lemma *walk-length-conv*: $walk-length\ p = length\ p - 1$
by (*induct* p *rule*: *walk-edges.induct*) (*auto* *simp*: *walk-length-def*)

lemma *walk-length-rev*: $walk-length\ p = walk-length (rev\ p)$
using *walk-edges-rev* *walk-length-def*
by (*metis* *length-rev*)

lemma *walk-length-app*: $xs \neq [] \implies ys \neq [] \implies walk-length (xs @ ys) = walk-length\ xs + walk-length\ ys + 1$
apply (*induct* xs *rule*: *walk-edges.induct*)
 apply (*simp-all* *add*: *walk-length-def*)
 using *walk-edges-singleton-app* **by** *force*

lemma *walk-length-app-ineq*: $walk-length (xs @ ys) \geq walk-length\ xs + walk-length\ ys \wedge$
 $walk-length (xs @ ys) \leq walk-length\ xs + walk-length\ ys + 1$
proof (*cases* $xs = [] \vee ys = []$)
 case *True*
 then show *?thesis* **using** *walk-length-def* **by** *auto*
next
 case *False*
 then show *?thesis*
 by (*simp* *add*: *walk-length-app*)
qed

Note that while the trivial walk is allowed, the empty walk is not

definition *is-walk* :: 'a list \Rightarrow bool **where**

is-walk *xs* \equiv set *xs* \subseteq *V* \wedge set (walk-edges *xs*) \subseteq *E* \wedge *xs* \neq []

lemma *is-walkI*: set *xs* \subseteq *V* \Longrightarrow set (walk-edges *xs*) \subseteq *E* \Longrightarrow *xs* \neq [] \Longrightarrow *is-walk* *xs*

using *is-walk-def* **by** *simp*

lemma *is-walk-wf*: *is-walk* *xs* \Longrightarrow set *xs* \subseteq *V*

by (*simp add: is-walk-def*)

lemma *is-walk-wf-hd*: *is-walk* *xs* \Longrightarrow hd *xs* \in *V*

using *is-walk-wf hd-in-set is-walk-def* **by** *blast*

lemma *is-walk-wf-last*: *is-walk* *xs* \Longrightarrow last *xs* \in *V*

using *is-walk-wf last-in-set is-walk-def* **by** *blast*

lemma *is-walk-singleton*: *u* \in *V* \Longrightarrow *is-walk* [*u*]

unfolding *is-walk-def* **using** *walk-edges.simps* **by** *simp*

lemma *is-walk-not-empty*: *is-walk* *xs* \Longrightarrow *xs* \neq []

unfolding *is-walk-def* **by** *simp*

lemma *is-walk-not-empty2*: *is-walk* [] = *False*

unfolding *is-walk-def* **by** *simp*

Reasoning on transformations of a walk

lemma *is-walk-rev*: *is-walk* *xs* \longleftrightarrow *is-walk* (*rev xs*)

unfolding *is-walk-def* **using** *walk-edges-rev*

by (*metis rev-is-Nil-conv set-rev*)

lemma *is-walk-tl*: length *xs* \geq 2 \Longrightarrow *is-walk* *xs* \Longrightarrow *is-walk* (*tl xs*)

using *walk-edges-tl-ss is-walk-def in-mono list.set-sel(2) tl-Nil* **by** *fastforce*

lemma *is-walk-append*:

assumes *is-walk* *xs*

assumes *is-walk* *ys*

assumes last *xs* = hd *ys*

shows *is-walk* (*xs* @ (*tl ys*))

proof (*intro is-walkI subsetI*)

show *xs* @ *tl ys* \neq [] **using** *is-walk-def assms* **by** *auto*

show $\bigwedge x. x \in$ set (*xs* @ *tl ys*) \Longrightarrow *x* \in *V* **using** *assms is-walk-def is-walk-wf*

by (*metis Un-iff in-mono list-set-tl set-append*)

next

fix *x* **assume** *xin*: *x* \in set (walk-edges (*xs* @ *tl ys*))

show *x* \in *E* **proof** (*cases tl ys = []*)

case *True*

then show *?thesis* **using** *assms(1) is-walk-def xin* **by** *auto*

next

```

case False
then have xin2:  $x \in (\text{set } (\text{walk-edges } xs) \cup \text{set } (\text{walk-edges } (\text{tl } ys))) \cup \{\{ \text{last } xs, \text{hd } (\text{tl } ys) \}\}$ 
  using walk-edges-append-union is-walk-not-empty assms xin by auto
have 1:  $\text{set } (\text{walk-edges } xs) \subseteq E$  using assms(1) is-walk-def
  by simp
have 2:  $\text{set } (\text{walk-edges } (\text{tl } ys)) \subseteq E$  using assms(2) is-walk-def
  by (meson dual-order.trans walk-edges-tl-ss)
have  $\{\text{last } xs, \text{hd } (\text{tl } ys)\} \in E$  using is-walk-def assms(2) assms(3)
  by (metis False hd-Cons-tl insert-subset list.simps(15) walk-edges.simps(3))
then show ?thesis using 1 2 xin2 by auto
qed
qed

```

```

lemma is-walk-decomp:
  assumes is-walk ( $xs @ [y] @ ys @ [y] @ zs$ ) (is is-walk ?w)
  shows is-walk ( $xs @ [y] @ zs$ )
proof (intro is-walkI)
  show  $\text{set } (xs @ [y] @ zs) \subseteq V$  using assms is-walk-def by simp
  show  $xs @ [y] @ zs \neq []$  by simp
  show  $\text{set } (\text{walk-edges } (xs @ [y] @ zs)) \subseteq E$ 
    using walk-edges-decomp-ss assms(1) is-walk-def by blast
qed

```

```

lemma is-walk-hd-tl:
  assumes is-walk ( $y \# ys$ )
  assumes  $\{x, y\} \in E$ 
  shows is-walk ( $x \# y \# ys$ )
proof (intro is-walkI)
  show  $\text{set } (x \# y \# ys) \subseteq V$ 
    using assms by (simp add: is-walk-def wellformed-alt-fst)
  show  $\text{set } (\text{walk-edges } (x \# y \# ys)) \subseteq E$ 
    using walk-edges.simps assms is-walk-def by simp
  show  $x \# y \# ys \neq []$  by simp
qed

```

```

lemma is-walk-drop-hd:
  assumes  $ys \neq []$ 
  assumes is-walk ( $y \# ys$ )
  shows is-walk ys
proof (intro is-walkI)
  show  $\text{set } ys \subseteq V$ 
    using assms is-walk-wf by fastforce
  show  $\text{set } (\text{walk-edges } ys) \subseteq E$ 
    using assms is-walk-def walk-edges-tl-ss by force
  show  $ys \neq []$  using assms by simp
qed

```

```

lemma walk-edges-index:

```

```

assumes  $i \geq 0$   $i < \text{walk-length } w$ 
assumes  $\text{is-walk } w$ 
shows  $(\text{walk-edges } w) ! i \in E$ 
using  $\text{assms}$ 
proof ( $\text{induct } w$  arbitrary:  $i$  rule:  $\text{walk-edges.induct}$ ,  $\text{simp add: is-walk-not-empty2}$ ,

   $\text{simp add: walk-length-def}$ )
case  $(\exists x y \text{ys})$ 
then show  $?case$  proof ( $\text{cases } i = 0$ )
  case  $\text{True}$ 
  then show  $?thesis$ 
  using  $\exists.\text{prems}(\exists)$   $\text{is-walk-def}$  by  $\text{fastforce}$ 
next
case  $\text{False}$ 
have  $gt: 0 \leq i - 1$  using  $\text{False}$  by  $\text{simp}$ 
have  $lt: i - 1 < \text{walk-length } (y \# \text{ys})$ 
  using  $\exists.\text{prems}(2)$   $\text{False}$   $\text{walk-length-conv}$  by  $\text{auto}$ 
have  $\text{is-walk } (y \# \text{ys})$ 
  using  $\exists.\text{prems}(3)$   $\text{is-walk-def}$  by  $\text{fastforce}$ 
then show  $?thesis$  using  $\exists.\text{hyps}[\text{of } i - 1]$ 
by ( $\text{metis } \exists.\text{prems}(1)$   $\text{False}$   $gt$   $lt$   $\text{le-neq-implies-less}$   $\text{nth-Cons-pos}$   $\text{walk-edges.simps}(3)$ )

```

qed
qed

lemma is-walk-index :

```

assumes  $i \geq 0$   $\text{Suc } i < (\text{length } w)$ 
assumes  $\text{is-walk } w$ 
shows  $\{w ! i, w ! (i + 1)\} \in E$ 
using  $\text{assms}$  proof ( $\text{induct } w$  arbitrary:  $i$  rule:  $\text{walk-edges.induct}$ ,  $\text{simp}$ ,  $\text{simp}$ )
fix  $x y \text{ys } i$ 
assume  $\text{IH: } \bigwedge j. 0 \leq j \implies \text{Suc } j < \text{length } (y \# \text{ys}) \implies \text{is-walk } (y \# \text{ys}) \implies$ 
 $\{(y \# \text{ys}) ! j, (y \# \text{ys}) ! (j + 1)\} \in E$ 
assume  $1: 0 \leq i$  and  $2: \text{Suc } i < \text{length } (x \# y \# \text{ys})$  and  $3: \text{is-walk } (x \# y \#$ 
 $\text{ys})$ 
show  $\{(x \# y \# \text{ys}) ! i, (x \# y \# \text{ys}) ! (i + 1)\} \in E$ 
proof ( $\text{cases } i = 0$ )
  case  $\text{True}$ 
  then show  $?thesis$  using  $\exists$   $\text{is-walk-def}$ 
  by  $\text{simp}$ 
next
case  $\text{False}$ 
have  $\text{is-walk } (y \# \text{ys})$  using  $\text{is-walk-def } \exists$  by  $\text{fastforce}$ 
then show  $?thesis$  using  $2$   $\text{IH}[\text{of } i - 1]$ 
by ( $\text{simp add: False nat-less-le}$ )

```

qed
qed

lemma is-walk-take :

```

assumes is-walk w
assumes  $n > 0$ 
assumes  $n \leq \text{length } w$ 
shows is-walk (take n w)
using assms proof (induct w arbitrary: n rule: walk-edges.induct)
case 1
then show ?case by simp
next
case (2 x)
then have  $n = 1$  using 2 by auto
then show ?case by (simp add: 2.prem(1))
next
case (3 x y ys)
then show ?case proof (cases n = 1)
  case True
    then have  $\text{take } n (x \# y \# ys) = [x]$ 
    by simp
    then show ?thesis using is-walk-def 3.prem(1) by simp
  next
  case False
    then have ngt: n ≥ 2 using 3.prem(2) by auto
    then have tk-split1: take n (x # y # ys) = x # take (n - 1) (y # ys) using 3
    by (simp add: take-Cons')
    then have tk-split: take n (x # y # ys) = x # y # (take (n - 2) ys)
    using 3 ngt take-Cons'[of n - 1 y ys]
    by (metis False diff-diff-left less-one nat-neq-iff one-add-one zero-less-diff)
    have w: is-walk (y # ys) using is-walk-tl
    using 3.prem(1) is-walk-def by force
    have  $n - 1 \leq \text{length } (y \# ys)$  using 3.prem(3) by simp
    then have w-tl: is-walk (take (n - 1) (y # ys)) using 3.hyps[of n - 1] w
    3.prem ngt
    by linarith
    have  $\{x, y\} \in E$  using is-walk-def walk-edges.simps 3.prem(1) by auto
    then show ?thesis using is-walk-hd-tl[of y (take (n - 2) ys) x] tk-split
    using tk-split1 w-tl by force
  qed
qed

lemma is-walk-drop:
assumes is-walk w
assumes  $n < \text{length } w$ 
shows is-walk (drop n w)
using assms proof (induct w arbitrary: n rule: walk-edges.induct)
case 1
then show ?case by simp
next
case (2 x)
then have  $n = 0$  using 2 by auto
then show ?case by (simp add: 2.prem(1))

```

```

next
case (∃ x y ys)
then show ?case proof (cases n ≥ 2)
  case True
  then have ngt: n ≥ 2 using 3.prem1(2) by auto
  then have tk-split1: drop n (x # y # ys) = drop (n - 1) (y # ys) using 3
    by (simp add: drop-Cons')
  then have tk-split: drop n (x # y # ys) = (drop (n - 2) ys)
    using 3 ngt drop-Cons'[of n - 1 y ys] True
    by (metis Suc-1 Suc-le-eq diff-diff-left less-not-refl nat-1-add-1 zero-less-diff)
  have w: is-walk (y # ys) using is-walk-tl
    using 3.prem1(1) is-walk-def by force
  have n - 1 < length (y # ys) using 3.prem1(2) by simp
  then have w-tl: is-walk (drop (n - 1) (y # ys)) using 3.hyps[of n - 1] w
    3.prem1 ngt
    by linarith
  have {x, y} ∈ E using is-walk-def walk-edges.simps 3.prem1(1) by auto
  then show ?thesis using is-walk-hd-tl[of y (take (n - 2) ys) x] tk-split
    using tk-split1 w-tl by force
next
case False
then have or: n = 0 ∨ n = 1
  by auto
  have walk: is-walk (y # ys) using is-walk-drop-hd 3 by blast
  have n0: n = 0 ⇒ (drop n (x # y # ys)) = (x # y # ys) by simp
  have n1: n = 1 ⇒ (drop n (x # y # ys)) = y # ys by simp
  then show ?thesis using n0 3 walk or by auto
qed
qed

```

definition *walks* :: 'a list set **where**

walks ≡ {p. is-walk p}

definition *is-open-walk* :: 'a list ⇒ bool **where**

is-open-walk xs ≡ is-walk xs ∧ hd xs ≠ last xs

lemma *is-open-walk-rev*: *is-open-walk* xs ⇔ *is-open-walk* (rev xs)

unfolding *is-open-walk-def* **using** *is-walk-rev*
by (metis hd-rev last-rev)

definition *is-closed-walk* :: 'a list ⇒ bool **where**

is-closed-walk xs ≡ is-walk xs ∧ hd xs = last xs

lemma *is-closed-walk-rev*: *is-closed-walk* xs ⇔ *is-closed-walk* (rev xs)

unfolding *is-closed-walk-def* **using** *is-walk-rev*
by (metis hd-rev last-rev)

definition *is-trail* :: 'a list ⇒ bool **where**

is-trail xs ≡ is-walk xs ∧ distinct (walk-edges xs)

lemma *is-trail-rev*: $is-trail\ xs \longleftrightarrow is-trail\ (rev\ xs)$
unfolding *is-trail-def* **using** *is-walk-rev*
by (*metis distinct-rev walk-edges-rev*)

2.2 Paths

There are two common definitions of a path. The first, given below, excludes the case where a path is a cycle. Note this also excludes the trivial path $[x]$

definition *is-path* :: 'a list \Rightarrow bool **where**
is-path $xs \equiv (is-open-walk\ xs \wedge distinct\ (xs))$

lemma *is-path-rev*: $is-path\ xs \longleftrightarrow is-path\ (rev\ xs)$
unfolding *is-path-def* **using** *is-open-walk-rev*
by (*metis distinct-rev*)

lemma *is-path-walk*: $is-path\ xs \implies is-walk\ xs$
unfolding *is-path-def is-open-walk-def* **by** *auto*

definition *paths* :: 'a list set **where**
paths $\equiv \{p \cdot is-path\ p\}$

lemma *paths-ss-walk*: $paths \subseteq walks$
unfolding *paths-def walks-def is-path-def is-open-walk-def* **by** *auto*

A more generic definition of a path - used when a cycle is considered a path, and therefore includes the trivial path $[x]$

definition *is-gen-path*:: 'a list \Rightarrow bool **where**
is-gen-path $p \equiv is-walk\ p \wedge ((distinct\ (tl\ p) \wedge hd\ p = last\ p) \vee distinct\ p)$

lemma *is-path-gen-path*: $is-path\ p \implies is-gen-path\ p$
unfolding *is-path-def is-gen-path-def is-open-walk-def* **by** (*auto simp add: distinct-tl*)

lemma *is-gen-path-rev*: $is-gen-path\ p \longleftrightarrow is-gen-path\ (rev\ p)$
unfolding *is-gen-path-def* **using** *is-walk-rev distinct-tl-rev*
by (*metis distinct-rev hd-rev last-rev*)

lemma *is-gen-path-distinct*: $is-gen-path\ p \implies hd\ p \neq last\ p \implies distinct\ p$
unfolding *is-gen-path-def* **by** *auto*

lemma *is-gen-path-distinct-tl*:
assumes *is-gen-path* p **and** $hd\ p = last\ p$
shows $distinct\ (tl\ p)$

proof (*cases length p > 1*)

case *True*

then show *?thesis*

using *assms(1) distinct-tl is-gen-path-def* **by** *auto*

next


```

case False
then show ?thesis
  using assms(1) distinct-tl is-gen-path-def by auto
qed

```

```

lemma is-gen-path-trivial:  $x \in V \implies is-gen-path\ x$ 
  unfolding is-gen-path-def is-walk-def by simp

```

```

definition gen-paths :: 'a list where
gen-paths  $\equiv \{p . is-gen-path\ p\}$ 

```

```

lemma gen-paths-ss-walks:  $gen-paths \subseteq walks$ 
  unfolding gen-paths-def walks-def is-gen-path-def by auto

```

2.3 Cycles

Note, a cycle must be non trivial (i.e. have an edge), but as we let a loop by a cycle we broaden the definition in comparison to Noschinski [5] for a cycle to be of length greater than 1 rather than 3

```

definition is-cycle :: 'a list  $\Rightarrow$  bool where
is-cycle  $xs \equiv is-closed-walk\ xs \wedge walk-length\ xs \geq 1 \wedge distinct\ (tl\ xs)$ 

```

```

lemma is-gen-path-cycle:  $is-cycle\ p \implies is-gen-path\ p$ 
  unfolding is-cycle-def is-gen-path-def is-closed-walk-def by auto

```

```

lemma is-cycle-alt-gen-path:  $is-cycle\ xs \longleftrightarrow is-gen-path\ xs \wedge walk-length\ xs \geq 1 \wedge hd\ xs = last\ xs$ 

```

```

proof (intro iffI)

```

```

  show  $is-cycle\ xs \implies is-gen-path\ xs \wedge 1 \leq walk-length\ xs \wedge hd\ xs = last\ xs$ 

```

```

    using is-gen-path-cycle is-cycle-def is-closed-walk-def

```

```

    by auto

```

```

  show  $is-gen-path\ xs \wedge 1 \leq walk-length\ xs \wedge hd\ xs = last\ xs \implies is-cycle\ xs$ 

```

```

    using distinct-tl is-closed-walk-def is-cycle-def is-gen-path-def by blast

```

```

qed

```

```

lemma is-cycle-alt:  $is-cycle\ xs \longleftrightarrow is-walk\ xs \wedge distinct\ (tl\ xs) \wedge walk-length\ xs \geq 1 \wedge hd\ xs = last\ xs$ 

```

```

proof (intro iffI)

```

```

  show  $is-cycle\ xs \implies is-walk\ xs \wedge distinct\ (tl\ xs) \wedge 1 \leq walk-length\ xs \wedge hd\ xs = last\ xs$ 

```

```

    using is-cycle-alt-gen-path is-cycle-def is-gen-path-def by blast

```

```

  show  $is-walk\ xs \wedge distinct\ (tl\ xs) \wedge 1 \leq walk-length\ xs \wedge hd\ xs = last\ xs \implies is-cycle\ xs$ 

```

```

    by (simp add: is-cycle-alt-gen-path is-gen-path-def)

```

```

qed

```

```

lemma is-cycle-rev:  $is-cycle\ xs \longleftrightarrow is-cycle\ (rev\ xs)$ 

```

```

proof –

```

```

  have len:  $1 \leq walk-length\ xs \longleftrightarrow 1 \leq walk-length\ (rev\ xs)$ 

```

by (metis length-rev walk-edges-rev walk-length-def)
 have $hd\ xs = last\ xs \implies distinct\ (tl\ xs) \longleftrightarrow distinct\ (tl\ (rev\ xs))$
 using distinct-tl-rev by blast
 then show ?thesis using len is-cycle-def
 using is-closed-walk-def is-closed-walk-rev by auto
 qed

lemma cycle-tl-is-path: $is-cycle\ xs \wedge walk-length\ xs \geq 3 \implies is-path\ (tl\ xs)$
proof (simp add: is-cycle-def is-path-def is-open-walk-def is-closed-walk-def walk-length-conv,

elim conjE, intro conjI, simp add: is-walk-tl)
 assume $w: is-walk\ xs$ and $eq: hd\ xs = last\ xs$ and $3 \leq length\ xs - Suc\ 0$ and
 $dis: distinct\ (tl\ xs)$
 then have $len: 4 \leq length\ xs$
 by linarith
 then have $lentl: 3 \leq length\ (tl\ xs)$ by simp
 then have $lentltl: 2 \leq length\ (tl\ (tl\ xs))$ by simp
 have $last\ (tl\ (tl\ xs)) = last\ (tl\ xs)$
 by (metis One-nat-def Suc-1 $3 \leq length\ xs - Suc\ 0$ diff-is-0-eq' is-walk-def
 is-walk-tl last-tl
 $lentl\ not-less-eq-eq\ numeral-le-one-iff\ one-le-numeral\ order.trans\ semiring-norm(70)\ w$)
 then have $last\ (tl\ xs) \in set\ (tl\ (tl\ xs))$
 using last-in-list-tl-set lentltl by (metis last-in-set list.sel(2))
 moreover have $hd\ (tl\ xs) \notin set\ (tl\ (tl\ xs))$ using dis lentltl
 by (metis distinct.simps(2) hd-Cons-tl list.sel(2) list.size(3) not-numeral-le-zero)

 ultimately show $hd\ (tl\ xs) \neq last\ (tl\ xs)$ by fastforce
 qed

lemma is-gen-path-path:
 assumes $is-gen-path\ p$ and $walk-length\ p > 0$ and $(\neg is-cycle\ p)$
 shows $is-path\ p$
proof (simp add: is-gen-path-def is-path-def is-open-walk-def, intro conjI)
 show $is-walk\ p$ using is-gen-path-def assms(1) by simp
 show $ne: hd\ p \neq last\ p$
 using assms(1) assms(2) assms(3) is-cycle-alt-gen-path by auto
 have $((distinct\ (tl\ p) \wedge hd\ p = last\ p) \vee distinct\ p)$ using is-gen-path-def assms(1)
 by auto
 thus $distinct\ p$ using ne by auto
 qed

lemma is-gen-path-options: $is-gen-path\ p \longleftrightarrow is-cycle\ p \vee is-path\ p \vee (\exists\ v \in V.\ p = [v])$
proof (intro iffI)
 assume $a: is-gen-path\ p$
 then have $p \neq []$ unfolding is-gen-path-def is-walk-def by auto
 then have $(\forall\ v \in V.\ p \neq [v]) \implies walk-length\ p > 0$ using walk-length-def
 by (metis a is-gen-path-def is-walk-wf-hd length-greater-0-conv list.collapse)

```

list.distinct(1) walk-edges.simps( $\beta$ )
  then show is-cycle  $p \vee$  is-path  $p \vee (\exists v \in V. p = [v])$ 
    using a is-gen-path-path by auto
next
show is-cycle  $p \vee$  is-path  $p \vee (\exists v \in V. p = [v]) \implies$  is-gen-path  $p$ 
  using is-gen-path-cycle is-path-gen-path is-gen-path-trivial by auto
qed

```

definition *cycles* :: 'a list set **where**

cycles $\equiv \{p. \text{is-cycle } p\}$

lemma *cycles-ss-gen-paths*: *cycles* \subseteq *gen-paths*

unfolding *cycles-def* *gen-paths-def* **using** *is-gen-path-cycle* **by** *auto*

lemma *gen-paths-ss*: *gen-paths* \subseteq *cycles* \cup *paths* $\cup \{[v] \mid v. v \in V\}$

unfolding *gen-paths-def* *cycles-def* *paths-def* **using** *is-gen-path-options* **by** *auto*

Walk edges are distinct in a path and cycle

lemma *distinct-edgesI*:

assumes *distinct* p **shows** *distinct* (*walk-edges* p)

proof –

from *assms* **have** *?thesis* $\wedge u. u \notin \text{set } p \implies (\wedge v. u \neq v \implies \{u, v\} \notin \text{set}$
(*walk-edges* p))

by (*induct* p *rule*: *walk-edges.induct*) *auto*

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

qed

lemma *scycles-distinct-edges*:

assumes $c \in \text{cycles}$ $3 \leq \text{walk-length } c$ **shows** *distinct* (*walk-edges* c)

proof –

from *assms* **have** *c-props*: *distinct* (*tl* c) $4 \leq \text{length } c$ $\text{hd } c = \text{last } c$

by (*auto* *simp* *add*: *cycles-def* *is-cycle-def* *is-closed-walk-def* *walk-length-conv*)

then have $\{\text{hd } c, \text{hd } (\text{tl } c)\} \notin \text{set } (\text{walk-edges } (\text{tl } c))$

proof (*induct* c *rule*: *walk-edges.induct*)

case ($\exists x y ys$)

then have $\text{hd } ys \neq \text{last } ys$ **by** (*cases* ys) *auto*

moreover

from \exists **have** $\text{walk-edges } (y \# ys) = \{y, \text{hd } ys\} \# \text{walk-edges } ys$

by (*cases* ys) *auto*

moreover

{ fix xs **have** $\text{set } (\text{walk-edges } xs) \subseteq \text{Pow } (\text{set } xs)$

by (*induct* xs *rule*: *walk-edges.induct*) *auto* }

ultimately

show *?case* **using** \exists **by** *auto*

qed *simp-all*

moreover

from *assms* **have** *distinct* (*walk-edges* (*tl* c))

by (*intro* *distinct-edgesI*) (*simp* *add*: *cycles-def* *is-cycle-def*)

ultimately

```

show ?thesis by (cases c, simp-all)
  (metis distinct.simps(1) distinct.simps(2) list.sel(1) list.sel(3) walk-edges.elims)
qed

end

context fin-ulgraph
begin

lemma finite-paths: finite paths
proof –
  have ss: paths  $\subseteq$  {xs. set xs  $\subseteq$  V  $\wedge$  length xs  $\leq$  (card (V))}
  proof (rule, simp, intro conjI)
    show 1:  $\bigwedge x. x \in \text{paths} \implies \text{set } x \subseteq V$ 
    unfolding paths-def is-path-def is-open-walk-def is-walk-def by simp
    fix x assume a: x  $\in$  paths
    then have distinct x
    using paths-def is-path-def by simp-all
    then have eq: length x = card (set x)
    by (simp add: distinct-card)
    then show length x  $\leq$  order using a 1
    by (simp add: card-mono finV)
  qed
  have finite {xs. set xs  $\subseteq$  V  $\wedge$  length xs  $\leq$  (card (V))}
    using finV by (simp add: finite-lists-length-le)
  thus ?thesis using ss finite-subset by auto
qed

lemma finite-cycles: finite (cycles)
proof –
  have cycles  $\subseteq$  {xs. set xs  $\subseteq$  V  $\wedge$  length xs  $\leq$  Suc (card (V))}
  proof (rule, simp)
    fix p assume p  $\in$  cycles
    then have distinct (tl p) and set p  $\subseteq$  V
    unfolding cycles-def walks-def is-cycle-def is-closed-walk-def is-walk-def
    by (simp-all)
    then have set (tl p)  $\subseteq$  V
    by (cases p) auto
    with finV have card (set (tl p))  $\leq$  card (V)
    by (rule card-mono)
    then have length (p)  $\leq$  1 + card (V)
    using distinct-card[OF  $\langle$ distinct (tl p) $\rangle$ ] by auto
    then show set p  $\subseteq$  V  $\wedge$  length p  $\leq$  Suc (card (V))
    by (simp add:  $\langle$ set p  $\subseteq$  V $\rangle$ )
  qed
moreover
  have finite {xs. set xs  $\subseteq$  V  $\wedge$  length xs  $\leq$  Suc (card (V))}
    using finV by (rule finite-lists-length-le)
  ultimately

```

show *?thesis* **by** (*rule finite-subset*)
qed

lemma *finite-gen-paths: finite (gen-paths)*

proof –

have *finite* ($\{[v] \mid v . v \in V\}$) **using** *finV* **by** *auto*

thus *?thesis* **using** *gen-paths-ss finite-cycles finite-paths finite-subset* **by** *auto*
qed

end

end

3 Connectivity

This theory defines concepts around the connectivity of a graph and its vertices, as well as graph properties that depend on connectivity definitions, such as shortest path, radius, diameter, and eccentricity

theory *Connectivity* **imports** *Undirected-Graph-Walks*
begin

context *ulgraph*
begin

3.1 Connecting Walks and Paths

definition *connecting-walk* :: $'a \Rightarrow 'a \Rightarrow 'a \text{ list} \Rightarrow \text{bool}$ **where**
connecting-walk $u\ v\ xs \equiv \text{is-walk } xs \wedge \text{hd } xs = u \wedge \text{last } xs = v$

lemma *connecting-walk-rev: connecting-walk* $u\ v\ xs \longleftrightarrow \text{connecting-walk } v\ u\ (\text{rev } xs)$

unfolding *connecting-walk-def* **using** *is-walk-rev*
by (*auto simp add: hd-rev last-rev*)

lemma *connecting-walk-wf: connecting-walk* $u\ v\ xs \implies u \in V \wedge v \in V$
using *is-walk-wf-hd is-walk-wf-last* **by** (*auto simp add: connecting-walk-def*)

lemma *connecting-walk-self: u* $\in V \implies \text{connecting-walk } u\ u\ [u] = \text{True}$
unfolding *connecting-walk-def* **by** (*simp add: is-walk-singleton*)

We define two definitions of connecting paths. The first uses the *gen-path* definition, which allows for trivial paths and cycles, the second uses the stricter definition of a path which requires it to be an open walk

definition *connecting-path* :: $'a \Rightarrow 'a \Rightarrow 'a \text{ list} \Rightarrow \text{bool}$ **where**
connecting-path $u\ v\ xs \equiv \text{is-gen-path } xs \wedge \text{hd } xs = u \wedge \text{last } xs = v$

definition *connecting-path-str* :: $'a \Rightarrow 'a \Rightarrow 'a \text{ list} \Rightarrow \text{bool}$ **where**
connecting-path-str $u\ v\ xs \equiv \text{is-path } xs \wedge \text{hd } xs = u \wedge \text{last } xs = v$

lemma *connecting-path-rev*: $\text{connecting-path } u \ v \ xs \longleftrightarrow \text{connecting-path } v \ u \ (\text{rev } xs)$

unfolding *connecting-path-def* **using** *is-gen-path-rev*
by (*auto simp add: hd-rev last-rev*)

lemma *connecting-path-walk*: $\text{connecting-path } u \ v \ xs \implies \text{connecting-walk } u \ v \ xs$
unfolding *connecting-path-def* *connecting-walk-def* **using** *is-gen-path-def* **by** *auto*

lemma *connecting-path-str-gen*: $\text{connecting-path-str } u \ v \ xs \implies \text{connecting-path } u \ v \ xs$

unfolding *connecting-path-def* *connecting-path-str-def* *is-gen-path-def* *is-path-def*
by (*simp add: is-open-walk-def*)

lemma *connecting-path-gen-str*: $\text{connecting-path } u \ v \ xs \implies (\neg \text{is-cycle } xs) \implies \text{walk-length } xs > 0 \implies \text{connecting-path-str } u \ v \ xs$

unfolding *connecting-path-def* *connecting-path-str-def* **using** *is-gen-path-path* **by** *auto*

lemma *connecting-path-alt-def*: $\text{connecting-path } u \ v \ xs \longleftrightarrow \text{connecting-walk } u \ v \ xs \wedge \text{is-gen-path } xs$

proof –

have $\text{is-gen-path } xs \implies \text{is-walk } xs$

by (*simp add: is-gen-path-def*)

then have $(\text{is-walk } xs \wedge \text{hd } xs = u \wedge \text{last } xs = v) \wedge \text{is-gen-path } xs \longleftrightarrow (\text{hd } xs = u \wedge \text{last } xs = v) \wedge \text{is-gen-path } xs$

by *blast*

thus *?thesis*

by (*auto simp add: connecting-path-def connecting-walk-def*)

qed

lemma *connecting-path-length-bound*: $u \neq v \implies \text{connecting-path } u \ v \ p \implies \text{walk-length } p \geq 1$

using *walk-length-def*

by (*metis connecting-path-def is-gen-path-def is-walk-not-empty2 last-ConsL le-refl length-0-conv less-one list.exhaust-sel nat-less-le nat-neq-iff neq-Nil-conv walk-edges.simps(3)*)

lemma *connecting-path-self*: $u \in V \implies \text{connecting-path } u \ u \ [u] = \text{True}$

unfolding *connecting-path-alt-def* **using** *connecting-walk-self*

by (*simp add: is-gen-path-def is-walk-singleton*)

lemma *connecting-path-singleton*: $\text{connecting-path } u \ v \ xs \implies \text{length } xs = 1 \implies u = v$

by (*metis cancel-comm-monoid-add-class.diff-cancel connecting-path-def fact-1 fact-nonzero*

last-rev length-0-conv neq-Nil-conv singleton-rev-conv walk-edges.simps(3) walk-length-conv walk-length-def)

```

lemma connecting-walk-path:
  assumes connecting-walk u v xs
  shows  $\exists ys . \text{connecting-path } u v ys \wedge \text{walk-length } ys \leq \text{walk-length } xs$ 
proof (cases u = v)
  case True
  then show ?thesis
    using assms connecting-path-self connecting-walk-wf
    by (metis bot-nat-0.extremum list.size(3) walk-edges.simps(2) walk-length-def)
next
  case False
  then have walk-length xs  $\neq$  0 using assms connecting-walk-def is-walk-def
  by (metis last-ConsL length-0-conv list.distinct(1) list.exhaust-sel walk-edges.simps(3))
  walk-length-def
  then show ?thesis using assms False proof (induct walk-length xs arbitrary: xs
rule: less-induct)
    fix xs assume IH: ( $\wedge xsa . \text{walk-length } xsa < \text{walk-length } xs \implies \text{walk-length } xsa$ 
 $\neq 0 \implies$ 
connecting-walk u v xsa  $\implies u \neq v \implies \exists ys . \text{connecting-path } u v ys \wedge \text{walk-length}$ 
 $ys \leq \text{walk-length } xsa$ )
    assume assm: connecting-walk u v xs and ne:  $u \neq v$  and n0: walk-length xs  $\neq$ 
0
    then show  $\exists ys . \text{connecting-path } u v ys \wedge \text{walk-length } ys \leq \text{walk-length } xs$ 
    proof (cases walk-length xs  $\leq$  1) — Base Cases
      case True
      then have walk-length xs = 1
      using n0 by auto
      then show ?thesis using ne assm cancel-comm-monoid-add-class.diff-cancel
connecting-path-alt-def connecting-walk-def
      distinct-length-2-or-more distinct-singleton hd-Cons-tl is-gen-path-def
is-walk-def last-ConsL
      last-ConsR length-0-conv length-tl walk-length-conv
      by (metis True)
    next
    case False
    then show ?thesis
    proof (cases distinct xs)
      case True
      then show ?thesis
      using assm connecting-path-alt-def connecting-walk-def is-gen-path-def by
auto
    next
    case False
    then obtain ws ys zs y where xs-decomp:  $xs = ws@[y]@ys@[y]@zs$  using
not-distinct-decomp
    by blast
    let ?rs = ws@[y]@zs
    have hd: hd ?rs = u using xs-decomp assm connecting-walk-def
    by (metis hd-append list.distinct(1))

```

```

    have lst: last ?rs = v using xs-decomp assm connecting-walk-def by simp
    have wl: walk-length ?rs ≠ 0 using hd lst ne walk-length-conv by auto
    have set ?rs ⊆ V using assm connecting-walk-def is-walk-def xs-decomp by
auto
    have cw: connecting-walk u v ?rs unfolding connecting-walk-def is-walk-decomp
    using assm connecting-walk-def hd is-walk-decomp lst xs-decomp by blast
    have ys@[y] ≠ [] by simp
    then have length ?rs < length xs using xs-decomp length-list-decomp-lt by
auto
    have walk-length ?rs < walk-length xs using walk-length-conv xs-decomp by
force
    then show ?thesis using IH[of ?rs] using cw ne wl le-trans less-or-eq-imp-le
by blast
    qed
    qed
    qed
    qed

```

lemma *connecting-walk-split*:

```

    assumes connecting-walk u v xs assumes connecting-walk v z ys
    shows connecting-walk u z (xs @ (tl ys))
    using connecting-walk-def is-walk-append
    by (metis append.right-neutral assms(1) assms(2) connecting-walk-self connect-
ing-walk-wf hd-append2 is-walk-not-empty last-appendR last-tl list.collapse)

```

lemma *connecting-path-split*:

```

    assumes connecting-path u v xs connecting-path v z ys
    obtains p where connecting-path u z p and walk-length p ≤ walk-length (xs @
(tl ys))
    using connecting-walk-split connecting-walk-path connecting-path-walk assms(1)
assms(2) by blast

```

lemma *connecting-path-split-length*:

```

    assumes connecting-path u v xs connecting-path v z ys
    obtains p where connecting-path u z p and walk-length p ≤ walk-length xs +
walk-length ys
proof –
    have connecting-walk u z (xs @ (tl ys))
    using connecting-walk-split assms connecting-path-walk by blast
    have walk-length (xs @ (tl ys)) ≤ walk-length xs + walk-length ys
    using walk-length-app-ineq
    by (simp add: le-diff-conv walk-length-conv)
    thus ?thesis using connecting-path-split
    by (metis (full-types) assms(1) assms(2) dual-order.trans that)
qed

```


3.2 Vertex Connectivity

Two vertices are defined to be connected if there exists a connecting path. Note that the more general version of a connecting path is again used as a vertex should be considered as connected to itself

definition *vert-connected* :: 'a ⇒ 'a ⇒ bool **where**
vert-connected u v ≡ ∃ xs . *connecting-path* u v xs

lemma *vert-connected-rev*: *vert-connected* u v ⟷ *vert-connected* v u
unfolding *vert-connected-def* **using** *connecting-path-rev* **by** *auto*

lemma *vert-connected-id*: u ∈ V ⟹ *vert-connected* u u = True
unfolding *vert-connected-def* **using** *connecting-path-self* **by** *auto*

lemma *vert-connected-trans*: *vert-connected* u v ⟹ *vert-connected* v z ⟹ *vert-connected* u z
unfolding *vert-connected-def* **using** *connecting-path-split*
by *meson*

lemma *vert-connected-wf*: *vert-connected* u v ⟹ u ∈ V ∧ v ∈ V
using *vert-connected-def* *connecting-path-walk* *connecting-walk-wf* **by** *blast*

definition *vert-connected-n* :: 'a ⇒ 'a ⇒ nat ⇒ bool **where**
vert-connected-n u v n ≡ ∃ p . *connecting-path* u v p ∧ *walk-length* p = n

lemma *vert-connected-n-imp*: *vert-connected-n* u v n ⟹ *vert-connected* u v
by (*auto simp add: vert-connected-def vert-connected-n-def*)

lemma *vert-connected-n-rev*: *vert-connected-n* u v n ⟷ *vert-connected-n* v u n
unfolding *vert-connected-n-def* **using** *walk-length-rev*
by (*metis connecting-path-rev*)

definition *connecting-paths* :: 'a ⇒ 'a ⇒ 'a list set **where**
connecting-paths u v ≡ {xs . *connecting-path* u v xs}

lemma *connecting-paths-self*: u ∈ V ⟹ [u] ∈ *connecting-paths* u u
unfolding *connecting-paths-def* **using** *connecting-path-self* **by** *auto*

lemma *connecting-paths-empty-iff*: *vert-connected* u v ⟷ *connecting-paths* u v ≠ {}
unfolding *connecting-paths-def* *vert-connected-def* **by** *auto*

lemma *elem-connecting-paths*: p ∈ *connecting-paths* u v ⟹ *connecting-path* u v p
using *connecting-paths-def* **by** *blast*

lemma *connecting-paths-ss-gen*: *connecting-paths* u v ⊆ *gen-paths*
unfolding *connecting-paths-def* *gen-paths-def* *connecting-path-def* **by** *auto*

lemma *connecting-paths-sym*: xs ∈ *connecting-paths* u v ⟷ rev xs ∈ *connect-*

ing-paths v u

unfolding *connecting-paths-def* **using** *connecting-path-rev* **by** *simp*

A set is considered to be connected, if all the vertices within that set are pairwise connected

definition *is-connected-set* :: 'a set \Rightarrow bool **where**

is-connected-set $V' \equiv (\forall u v . u \in V' \longrightarrow v \in V' \longrightarrow \text{vert-connected } u v)$

lemma *is-connected-set-empty*: *is-connected-set* {}

unfolding *is-connected-set-def* **by** *simp*

lemma *is-connected-set-singleton*: $x \in V \Longrightarrow \text{is-connected-set } \{x\}$

unfolding *is-connected-set-def* **by** (*auto simp add: vert-connected-id*)

lemma *is-connected-set-wf*: *is-connected-set* $V' \Longrightarrow V' \subseteq V$

unfolding *is-connected-set-def*

by (*meson connecting-path-walk connecting-walk-wf subsetI vert-connected-def*)

lemma *is-connected-setD*: *is-connected-set* $V' \Longrightarrow u \in V' \Longrightarrow v \in V' \Longrightarrow \text{vert-connected } u v$

by (*simp add: is-connected-set-def*)

lemma *not-connected-set*: $\neg \text{is-connected-set } V' \Longrightarrow u \in V' \Longrightarrow \exists v \in V' . \neg \text{vert-connected } u v$

using *is-connected-setD* **by** (*meson is-connected-set-def vert-connected-rev vert-connected-trans*)

3.3 Graph Properties on Connectivity

The shortest path is defined to be the infimum of the set of connecting path walk lengths. Drawing inspiration from [4], we use the infimum and enats as this enables more natural reasoning in a non-finite setting, while also being useful for proofs of a more probabilistic or analysis nature

definition *shortest-path* :: 'a \Rightarrow 'a \Rightarrow enat **where**

shortest-path $u v \equiv \text{INF } p \in \text{connecting-paths } u v . \text{enat } (\text{walk-length } p)$

lemma *shortest-path-walk-length*: *shortest-path* $u v = n \Longrightarrow p \in \text{connecting-paths } u v \Longrightarrow \text{walk-length } p \geq n$

using *shortest-path-def INF-lower*[*of p connecting-paths u v λ p . enat (walk-length p)*]

by *auto*

lemma *shortest-path-lte*: $\bigwedge p . p \in \text{connecting-paths } u v \Longrightarrow \text{shortest-path } u v \leq \text{walk-length } p$

unfolding *shortest-path-def* **by** (*simp add: Inf-lower*)

lemma *shortest-path-obtains*:

assumes *shortest-path* $u v = n$

assumes $n \neq \text{top}$

obtains p **where** $p \in \text{connecting-paths } u \ v$ **and** $\text{walk-length } p = n$
using *enat-in-INF shortest-path-def* **by** (*metis* *assms(1)* *assms(2)* *the-enat.simps*)

lemma *shortest-path-intro*:
assumes $n \neq \text{top}$
assumes $(\exists p \in \text{connecting-paths } u \ v . \text{walk-length } p = n)$
assumes $(\bigwedge p. p \in \text{connecting-paths } u \ v \implies n \leq \text{walk-length } p)$
shows $\text{shortest-path } u \ v = n$
proof (*rule ccontr*)
assume $a: \text{shortest-path } u \ v \neq \text{enat } n$
then have $\text{shortest-path } u \ v < n$
by (*metis antisym-conv2* *assms(2)* *shortest-path-lte*)
then have $\exists p \in \text{connecting-paths } u \ v . \text{walk-length } p < n$
using *shortest-path-def* **by** (*simp* *add: INF-less-iff*)
thus *False* **using** *assms(3)*
using *le-antisym less-imp-le-nat* **by** *blast*
qed

lemma *shortest-path-self*:
assumes $u \in V$
shows $\text{shortest-path } u \ u = 0$
proof –
have $[u] \in \text{connecting-paths } u \ u$
using *connecting-paths-self* **by** (*simp* *add: assms*)
then have $\text{walk-length } [u] = 0$
using *walk-length-def* *walk-edges.simps* **by** *auto*
thus *?thesis* **using** *shortest-path-def*
by (*metis* $\langle [u] \in \text{connecting-paths } u \ u \rangle$ *le-zero-eq* *shortest-path-lte* *zero-enat-def*)

qed

lemma *connecting-paths-sym-length*: $i \in \text{connecting-paths } u \ v \implies \exists j \in \text{connecting-paths } v \ u. (\text{walk-length } j) = (\text{walk-length } i)$
using *connecting-paths-sym* **by** (*metis* *walk-length-rev*)

lemma *shortest-path-sym*: $\text{shortest-path } u \ v = \text{shortest-path } v \ u$
unfolding *shortest-path-def*
by (*intro* *INF-eq*)(*metis* *add.right-neutral* *le-iff-add* *connecting-paths-sym-length*)**+**

lemma *shortest-path-inf*: $\neg \text{vert-connected } u \ v \implies \text{shortest-path } u \ v = \infty$
using *connecting-paths-empty-iff* *shortest-path-def* **by** (*simp* *add: top-enat-def*)

lemma *shortest-path-not-inf*:
assumes $\text{vert-connected } u \ v$
shows $\text{shortest-path } u \ v \neq \infty$
proof –
have $\bigwedge p. \text{connecting-path } u \ v \ p \implies \text{enat } (\text{walk-length } p) \neq \infty$

using *connecting-path-def is-gen-path-def* **by** *auto*
thus *?thesis unfolding shortest-path-def connecting-paths-def*
by (*metis assms connecting-paths-def infinity-ileE mem-Collect-eq shortest-path-def shortest-path-lte vert-connected-def*)
qed

lemma *shortest-path-obtains2*:
assumes *vert-connected u v*
obtains *p* **where** $p \in \text{connecting-paths } u \ v$ **and** $\text{walk-length } p = \text{shortest-path } u \ v$
proof –
have $\text{connecting-paths } u \ v \neq \{\}$ **using** *assms connecting-paths-empty-iff* **by** *auto*
have $\text{shortest-path } u \ v \neq \infty$ **using** *assms shortest-path-not-inf* **by** *simp*
thus *?thesis using shortest-path-def enat-in-INF*
by (*metis that top-enat-def*)
qed

lemma *shortest-path-split*: $\text{shortest-path } x \ y \leq \text{shortest-path } x \ z + \text{shortest-path } z \ y$

proof (*cases vert-connected x y \wedge vert-connected x z*)
case *True*
show *?thesis*
proof (*rule ccontr*)
assume $\neg \text{shortest-path } x \ y \leq \text{shortest-path } x \ z + \text{shortest-path } z \ y$
then have $c: \text{shortest-path } x \ y > \text{shortest-path } x \ z + \text{shortest-path } z \ y$ **by** *simp*
have *vert-connected z y* **using** *True vert-connected-trans vert-connected-rev* **by** *blast*
then obtain $p1 \ p2$ **where** $\text{connecting-path } x \ z \ p1$ **and** $\text{connecting-path } z \ y \ p2$
and
 $s1: \text{shortest-path } x \ z = \text{walk-length } p1$ **and** $s2: \text{shortest-path } z \ y = \text{walk-length } p2$
using *True shortest-path-obtains2 connecting-paths-def elem-connecting-paths*
by *metis*
then obtain $p3$ **where** $\text{connecting-path } x \ y \ p3$ **and** $\text{walk-length } p1 + \text{walk-length } p2 \geq \text{walk-length } p3$
using *connecting-path-split-length* **by** *blast*
then have $\text{shortest-path } x \ z + \text{shortest-path } z \ y \geq \text{walk-length } p3$ **using** $s1 \ s2$
by *simp*
then have $lt: \text{shortest-path } x \ y > \text{walk-length } p3$ **using** c **by** *auto*
have $p3 \in \text{connecting-paths } x \ y$ **using** cp *connecting-paths-def* **by** *auto*
then show *False* **using** *shortest-path-def shortest-path-obtains2*
by (*metis True enat-ord-simps(1) enat-ord-simps(2) le-Suc-ex lt not-add-less1 shortest-path-lte*)
qed
next
case *False*
then show *?thesis*
by (*metis enat-ord-code(3) plus-enat-simps(2) plus-enat-simps(3) shortest-path-inf vert-connected-trans*)

qed

lemma *shortest-path-invalid-v*: $v \notin V \vee u \notin V \implies \text{shortest-path } u \ v = \infty$
using *shortest-path-inf vert-connected-wf* **by** *blast*

lemma *shortest-path-lb*:

assumes $u \neq v$

assumes *vert-connected* $u \ v$

shows $\text{shortest-path } u \ v > 0$

proof –

have $\bigwedge p. \text{connecting-path } u \ v \ p \implies \text{enat } (\text{walk-length } p) > 0$

using *connecting-path-length-bound* *assms* **by** *fastforce*

thus *?thesis* **unfolding** *shortest-path-def*

by (*metis elem-connecting-paths shortest-path-def shortest-path-obtains2* *assms(2)*)

qed

Eccentricity of a vertex v is the furthest distance between it and a (different) vertex

definition *eccentricity* :: $'a \Rightarrow \text{enat}$ **where**

eccentricity $v \equiv \text{SUP } u \in V - \{v\}. \text{shortest-path } v \ u$

lemma *eccentricity-empty-vertices*: $V = \{\} \implies \text{eccentricity } v = 0$

$V = \{v\} \implies \text{eccentricity } v = 0$

unfolding *eccentricity-def* **using** *bot-enat-def* **by** *simp-all*

lemma *eccentricity-bot-iff*: $\text{eccentricity } v = 0 \iff V = \{\} \vee V = \{v\}$

proof (*intro iffI*)

assume $a: \text{eccentricity } v = 0$

show $V = \{\} \vee V = \{v\}$

proof (*rule ccontr, simp*)

assume $a2: V \neq \{\} \wedge V \neq \{v\}$

have $\text{eq0}: \forall u \in V - \{v\}. \text{shortest-path } v \ u = 0$

using *SUP-bot-conv(1)*[*of* $\lambda u. \text{shortest-path } v \ u \ V - \{v\}$] *a* *eccentricity-def*

bot-enat-def **by** *simp*

have $\text{nc}: \forall u \in V - \{v\}. \neg \text{vert-connected } v \ u \longrightarrow \text{shortest-path } v \ u = \infty$

using *shortest-path-inf* **by** *simp*

have $\forall u \in V - \{v\}. \text{vert-connected } v \ u \longrightarrow \text{shortest-path } v \ u > 0$

using *shortest-path-lb* **by** *auto*

then show *False* **using** *eq0 a2 nc*

by *auto*

qed

next

show $V = \{\} \vee V = \{v\} \implies \text{eccentricity } v = 0$ **using** *eccentricity-empty-vertices*

by *auto*

qed

lemma *eccentricity-invalid-v*:

assumes $v \notin V$

assumes $V \neq \{\}$

shows *eccentricity* $v = \infty$
proof –
have $\bigwedge u. \text{shortest-path } v \ u = \infty$ **using** *assms shortest-path-invalid-v* **by** *blast*
have $V - \{v\} = V$ **using** *assms* **by** *simp*
then have *eccentricity* $v = (\text{SUP } u \in V . \text{shortest-path } v \ u)$ **by** (*simp add: eccentricity-def*)
thus *?thesis* **using** *eccentricity-def shortest-path-invalid-v assms* **by** *simp*
qed

lemma *eccentricity-gt-shortest-path*:
assumes $u \in V$
shows *eccentricity* $v \geq \text{shortest-path } v \ u$
proof (*cases* $u \in V - \{v\}$)
case *True*
then show *?thesis* **unfolding** *eccentricity-def* **by** (*simp add: SUP-upper*)
next
case *f1: False*
then have $u = v$ **using** *assms* **by** *auto*
then have *shortest-path* $u \ v = 0$ **using** *shortest-path-self assms* **by** *auto*
then show *?thesis* **by** (*simp add: ⟨u = v⟩*)
qed

lemma *eccentricity-disconnected-graph*:
assumes $\neg \text{is-connected-set } V$
assumes $v \in V$
shows *eccentricity* $v = \infty$
proof –
obtain u **where** *uin*: $u \in V$ **and** *nvc*: $\neg \text{vert-connected } v \ u$
using *not-connected-set assms* **by** *auto*
then have $u \neq v$ **using** *vert-connected-id* **by** *auto*
then have $u \in V - \{v\}$ **using** *uin* **by** *simp*
moreover have *shortest-path* $v \ u = \infty$ **using** *nvc shortest-path-inf* **by** *auto*
thus *?thesis* **using** *eccentricity-gt-shortest-path*
by (*metis enat-ord-simps(5) uin*)
qed

The diameter is the largest distance between any two vertices

definition *diameter* $:: \text{enat}$ **where**
diameter $\equiv \text{SUP } v \in V . \text{eccentricity } v$

lemma *diameter-gt-eccentricity*: $v \in V \implies \text{diameter} \geq \text{eccentricity } v$
using *diameter-def* **by** (*simp add: SUP-upper*)

lemma *diameter-disconnected-graph*:
assumes $\neg \text{is-connected-set } V$
shows *diameter* $= \infty$
unfolding *diameter-def* **using** *eccentricity-disconnected-graph*
by (*metis SUP-eq-const assms is-connected-set-empty*)

lemma *diameter-empty*: $V = \{\}$ \implies *diameter* = 0
unfolding *diameter-def* **using** *Sup-empty bot-enat-def* **by** *simp*

lemma *diameter-singleton*: $V = \{v\}$ \implies *diameter* = *eccentricity v*
unfolding *diameter-def* **by** *simp*

The radius is the smallest "shortest" distance between any two vertices

definition *radius* :: *enat* **where**
radius \equiv *INF* $v \in V$. *eccentricity v*

lemma *radius-lt-eccentricity*: $v \in V \implies$ *radius* \leq *eccentricity v*
using *radius-def* **by** (*simp add: INF-lower*)

lemma *radius-disconnected-graph*: \neg *is-connected-set V* \implies *radius* = ∞
unfolding *radius-def* **using** *eccentricity-disconnected-graph*
by (*metis INF-eq-const is-connected-set-empty*)

lemma *radius-empty*: $V = \{\}$ \implies *radius* = ∞
unfolding *radius-def* **using** *Inf-empty top-enat-def* **by** *simp*

lemma *radius-singleton*: $V = \{v\}$ \implies *radius* = *eccentricity v*
unfolding *radius-def* **by** *simp*

The centre of the graph is all vertices whose eccentricity equals the radius

definition *centre* :: '*a set* **where**
centre \equiv $\{v \in V$. *eccentricity v* = *radius* $\}$

lemma *centre-disconnected-graph*: \neg *is-connected-set V* \implies *centre* = V
unfolding *centre-def* **using** *radius-disconnected-graph eccentricity-disconnected-graph*
by *auto*

end

lemma (*in fin-ulgraph*) *fin-connecting-paths*: *finite* (*connecting-paths u v*)
using *connecting-paths-ss-gen finite-gen-paths finite-subset* **by** *fastforce*

3.4 We define a connected graph as a non-empty graph (the empty set is not usually considered connected by convention), where the vertex set is connected

locale *connected-ulgraph* = *ulgraph* + *ne-graph-system* +
assumes *connected: is-connected-set V*
begin

lemma *vertices-connected*: $u \in V \implies v \in V \implies$ *vert-connected u v*
using *is-connected-set-def connected* **by** *auto*

lemma *vertices-connected-path*: $u \in V \implies v \in V \implies \exists p$. *connecting-path u v p*
using *vertices-connected* **by** (*simp add: vert-connected-def*)

lemma *connecting-paths-not-empty*: $u \in V \implies v \in V \implies \text{connecting-paths } u \ v \neq \{\}$
using *connected not-empty connecting-paths-empty-iff is-connected-setD* **by** *blast*

lemma *min-shortest-path*: **assumes** $u \in V \ v \in V \ u \neq v$
shows *shortest-path* $u \ v > 0$
using *shortest-path-lb assms vertices-connected* **by** *auto*

The eccentricity, diameter, radius, and centre definitions tend to be only used in a connected context, as otherwise they are the INF/SUP value. In these contexts, we can obtain the vertex responsible

lemma *eccentricity-obtains-inf*:
assumes $V \neq \{v\}$
shows *eccentricity* $v = \infty \vee (\exists u \in (V - \{v\}) . \text{shortest-path } v \ u = \text{eccentricity } v)$
proof (*cases finite* $((\lambda u . \text{shortest-path } v \ u) \ ' (V - \{v\})))$
case *True*
then have $e : \text{eccentricity } v = \text{Max } ((\lambda u . \text{shortest-path } v \ u) \ ' (V - \{v\}))$
unfolding *eccentricity-def* **using** *Sup-enat-def*
using *assms not-empty* **by** *auto*
have $(V - \{v\}) \neq \{\}$ **using** *assms not-empty* **by** *auto*
then have $((\lambda u . \text{shortest-path } v \ u) \ ' (V - \{v\})) \neq \{\}$ **by** *simp*
then obtain n **where** $n \in ((\lambda u . \text{shortest-path } v \ u) \ ' (V - \{v\}))$ **and** $n = \text{eccentricity } v$
using *Max-in e True* **by** *auto*
then obtain u **where** $u \in (V - \{v\})$ **and** *shortest-path* $v \ u = \text{eccentricity } v$
by *blast*
then show *?thesis* **by** *auto*
next
case *False*
then have *eccentricity* $v = \infty$ **unfolding** *eccentricity-def* **using** *Sup-enat-def*
by (*metis (mono-tags, lifting) cSup-singleton empty-iff finite-insert insert-iff*)
then show *?thesis* **by** *simp*
qed

lemma *diameter-obtains*: *diameter* $= \infty \vee (\exists v \in V . \text{eccentricity } v = \text{diameter})$
proof (*cases is-singleton V*)
case *True*
then obtain v **where** $V = \{v\}$
using *is-singletonE* **by** *auto*
then show *?thesis* **using** *diameter-singleton*
by *simp*
next
case *f1: False*
then show *?thesis* **proof** (*cases finite* $((\lambda v . \text{eccentricity } v) \ ' V)$)
case *True*
then have *diameter* $= \text{Max } ((\lambda v . \text{eccentricity } v) \ ' V)$ **unfolding** *diameter-def*


```

using Sup-enat-def not-empty
  by simp
  then obtain  $n$  where  $n \in ((\lambda v. \text{eccentricity } v) \text{ ` } V)$  and  $\text{diameter} = n$  using
Max-in True
  using not-empty by auto
  then obtain  $u$  where  $u \in V$  and  $\text{eccentricity } u = \text{diameter}$ 
  by fastforce
  then show ?thesis by auto
next
  case False
  then have  $\text{diameter} = \infty$  unfolding diameter-def using Sup-enat-def by auto
  then show ?thesis by simp
qed
qed

```

```

lemma radius-diameter-singleton-eq: assumes  $\text{card } V = 1$  shows  $\text{radius} = \text{diameter}$ 
proof –
  obtain  $v$  where  $V = \{v\}$  using assms card-1-singletonE by auto
  thus ?thesis unfolding radius-def diameter-def by auto
qed

```

end

```

locale fin-connected-ulgraph = connected-ulgraph + fin-ulgraph
begin

```

In a finite context the supremum/infimum are equivalent to the Max/Min of the sets respectively. This can make reasoning easier

```

lemma shortest-path-Min-alt:
  assumes  $u \in V$   $v \in V$ 
  shows  $\text{shortest-path } u \ v = \text{Min } ((\lambda p. \text{enat } (\text{walk-length } p)) \text{ ` } (\text{connecting-paths } u \ v))$ 
  (is  $\text{shortest-path } u \ v = \text{Min } ?A$ )
proof –
  have  $ne: ?A \neq \{\}$ 
  using connecting-paths-not-empty assms by auto
  have finite (connecting-paths  $u \ v$ )
  by (simp add: fin-connecting-paths)
  then have fin: finite  $?A$ 
  by simp
  have  $\text{shortest-path } u \ v = \text{Inf } ?A$  unfolding shortest-path-def by simp
  thus ?thesis using Min-Inf ne
  by (metis fin)
qed

```

```

lemma eccentricity-Max-alt:
  assumes  $v \in V$ 
  assumes  $V \neq \{v\}$ 
  shows  $\text{eccentricity } v = \text{Max } ((\lambda u. \text{shortest-path } v \ u) \text{ ` } (V - \{v\}))$ 

```

unfolding *eccentricity-def* **using** *assms Sup-enat-def finV not-empty*
by *auto*

lemma *diameter-Max-alt*: $diameter = Max ((\lambda v. eccentricity v) \text{ ‘ } V)$
unfolding *diameter-def* **using** *Sup-enat-def finV not-empty* **by** *auto*

lemma *radius-Min-alt*: $radius = Min ((\lambda v. eccentricity v) \text{ ‘ } V)$
unfolding *radius-def* **using** *Min-Inf finV not-empty*
by (*metis (no-types, opaque-lifting) empty-is-image finite-imageI*)

lemma *eccentricity-obtains*:

assumes $v \in V$

assumes $V \neq \{v\}$

obtains u **where** $u \in V$ **and** $u \neq v$ **and** $shortest-path\ u\ v = eccentricity\ v$

proof –

have *ni*: $\bigwedge u. u \in V - \{v\} \implies u \neq v \wedge u \in V$ **by** *auto*

have *ne*: $V - \{v\} \neq \{\}$ **using** *assms not-empty* **by** *auto*

have $eccentricity\ v = Max ((\lambda u. shortest-path\ v\ u) \text{ ‘ } (V - \{v\}))$ **using** *eccentricity-Max-alt* **assms** **by** *simp*

then obtain u **where** *ui*: $u \in V - \{v\}$ **and** *eq*: $shortest-path\ v\ u = eccentricity\ v$

using *obtains-MAX* *assms finV ne* **by** (*metis finite-Diff*)

then have *neq*: $u \neq v$ **by** *blast*

have *uin*: $u \in V$ **using** *ui* **by** *auto*

thus *?thesis* **using** *neq eq that[of u] shortest-path-sym* **by** *simp*

qed

lemma *radius-obtains*:

obtains v **where** $v \in V$ **and** $radius = eccentricity\ v$

proof –

have $radius = Min ((\lambda v. eccentricity\ v) \text{ ‘ } V)$ **using** *radius-Min-alt* **by** *simp*

then obtain v **where** $v \in V$ **and** $radius = eccentricity\ v$

using *obtains-MIN*[of $V (\lambda v. eccentricity\ v)$] *not-empty finV* **by** *auto*

thus *?thesis*

by (*simp add: that*)

qed

lemma *radius-obtains-path-vertices*:

assumes $card\ V \geq 2$

obtains $u\ v$ **where** $u \in V$ **and** $v \in V$ **and** $u \neq v$ **and** $radius = shortest-path\ u\ v$

proof –

obtain v **where** *vin*: $v \in V$ **and** *e*: $radius = eccentricity\ v$

using *radius-obtains* **by** *blast*

then have $V \neq \{v\}$ **using** *assms* **by** *auto*

then obtain u **where** $u \in V$ **and** $u \neq v$ **and** $shortest-path\ u\ v = radius$

using *eccentricity-obtains vin e* **by** *auto*

thus *?thesis* **using** *vin*

by (*simp add: that*)

qed

lemma *diameter-obtains*:

obtains v where $v \in V$ and $diameter = eccentricity\ v$

proof –

have $diameter = Max ((\lambda v. eccentricity\ v) \text{ ‘ } V)$ using *diameter-Max-alt* by *simp*

then obtain v where $v \in V$ and $diameter = eccentricity\ v$

using *obtains-MAX*[of $V (\lambda v. eccentricity\ v)$] *not-empty finV* by *auto*

thus *?thesis*

by (*simp add: that*)

qed

lemma *diameter-obtains-path-vertices*:

assumes $card\ V \geq 2$

obtains $u\ v$ where $u \in V$ and $v \in V$ and $u \neq v$ and $diameter = shortest-path\ u\ v$

proof –

obtain v where $vin: v \in V$ and $e: diameter = eccentricity\ v$

using *diameter-obtains* by *blast*

then have $V \neq \{v\}$ using *assms* by *auto*

then obtain u where $u \in V$ and $u \neq v$ and $shortest-path\ u\ v = diameter$

using *eccentricity-obtains vin e* by *auto*

thus *?thesis* using *vin*

by (*simp add: that*)

qed

lemma *radius-diameter-bounds*:

shows $radius \leq diameter$ $diameter \leq 2 * radius$

proof –

show $radius \leq diameter$ unfolding *radius-def diameter-def*

by (*simp add: INF-le-SUP not-empty*)

next

show $diameter \leq 2 * radius$

proof (*cases card V ≥ 2*)

case *True*

then obtain $x\ y$ where $xin: x \in V$ and $ysin: y \in V$ and $d: shortest-path\ x\ y = diameter$

using *diameter-obtains-path-vertices* by *metis*

obtain z where $zin: z \in V$ and $e: eccentricity\ z = radius$ using *radius-obtains* by *metis*

have $shortest-path\ x\ z \leq eccentricity\ z$

using *eccentricity-gt-shortest-path xin shortest-path-sym* by *simp*

have $shortest-path\ x\ y \leq shortest-path\ x\ z + shortest-path\ z\ y$ using *shortest-path-split* by *simp*

also have $\dots \leq eccentricity\ z + eccentricity\ z$

using *eccentricity-gt-shortest-path shortest-path-sym zin xin yin* by (*simp add: add-mono*)

also have $\dots \leq radius + radius$ using *e* by *simp*

```

    finally show ?thesis using d by (simp add: mult-2)
next
  case False
  have card V ≠ 0 using not-empty finV by auto
  then have card V = 1 using False by simp
  then show ?thesis using radius-diameter-singleton-eq by (simp add: mult-2)
qed
qed

end

```

We define various subclasses of the general connected graph, using the functor locale pattern

```

locale connected-sgraph = sgraph + ne-graph-system +
  assumes connected: is-connected-set V

```

```

sublocale connected-sgraph ⊆ connected-ulgraph
by (unfold-locales) (simp add: connected)

```

```

locale fin-connected-sgraph = connected-sgraph + fin-sgraph

```

```

sublocale fin-connected-sgraph ⊆ fin-connected-ulgraph
by (unfold-locales)

```

```

end
theory Girth-Independence imports Connectivity
begin

```

4 Girth and Independence

We translate and extend on a number of definitions and lemmas on girth and independence from Noschinski's ugraph representation [4].

```

context sgraph
begin

```

```

definition girth :: enat where
  girth ≡ INF p∈ cycles. enat (walk-length p)

```

```

lemma girth-acyclic: cycles = {} ⇒ girth = ∞
unfolding girth-def using top-enat-def by simp

```

```

lemma girth-lte: c ∈ cycles ⇒ girth ≤ walk-length c
using girth-def INF-lower by auto

```

```

lemma girth-obtains:
  assumes girth ≠ top
  obtains c where c ∈ cycles and walk-length c = girth
  using enat-in-INF girth-def assms by (metis (full-types) the-enat.simps)

```

lemma *girthI*:
assumes $c' \in \text{cycles}$
assumes $\bigwedge c. c \in \text{cycles} \implies \text{walk-length } c' \leq \text{walk-length } c$
shows $\text{girth} = \text{walk-length } c'$
proof (*rule ccontr*)
assume $\text{girth} \neq \text{walk-length } c'$
then have $\text{girth} < \text{walk-length } c'$
using *assms girth-lte by fastforce*
then obtain c **where** $c \in \text{cycles}$ **and** $\text{walk-length } c < \text{walk-length } c'$
using *girth-def by (metis enat-ord-simps(2) girth-obtains infinity-ilessE top-enat-def)*

thus *False* **using** *assms(2) less-imp-le-nat le-antisym*
by *fastforce*

qed

lemma (*in fin-sgraph*) *girth-min-alt*:
assumes $\text{cycles} \neq \{\}$
shows $\text{girth} = \text{Min } ((\lambda c. \text{enat } (\text{walk-length } c)) \text{ `cycles})$ (**is** $\text{girth} = \text{Min } ?A$)
unfolding *girth-def* **using** *finite-cycles assms Min-Inf*
by (*metis (full-types) INF-le-SUP bot-enat-def ccInf-empty ccSup-empty enat-ord-code(5) finite-imageI top-enat-def zero-enat-def*)

definition *is-independent-set* :: $'a \text{ set} \Rightarrow \text{bool}$ **where**
is-independent-set $vs \equiv vs \subseteq V \wedge (\text{all-edges } vs) \cap E = \{\}$

A More mathematical way of thinking about it

lemma *is-independent-alt*: $\text{is-independent-set } vs \longleftrightarrow vs \subseteq V \wedge (\forall v \in vs. \forall u \in vs. \neg \text{vert-adj } v \ u)$
unfolding *is-independent-set-def*
proof (*auto*)
fix $v \ u$ **assume** $ss: vs \subseteq V$ **and** *inter*: $\text{all-edges } vs \cap E = \{\}$ **and** *vin*: $v \in vs$
and *uin*: $u \in vs$ **and** *adj*: $\text{vert-adj } v \ u$
then have $\text{inE}: \{v, u\} \in E$ **using** *vert-adj-def* **by** *simp*
then have $\text{imp}: \{v, u\} \in \text{all-edges } vs$ **using** *vin uin e-in-all-edges-ss vin uin*
by (*simp add: ss*)
then show *False*
using inE inter **by** *blast*

next
fix x **assume** $vs \subseteq V \forall v \in vs. \forall u \in vs. \neg \text{vert-adj } v \ u$ $x \in \text{all-edges } vs$ $x \in E$
then have $\bigwedge u \ v. \{u, v\} \subseteq vs \implies \{u, v\} \notin E$ **by** (*simp add: vert-adj-def*)
then have $\bigwedge x. x \subseteq vs \implies \text{card } x = 2 \implies x \notin E$ **by** (*metis card-2-iff*)
then show *False* **using** *all-edges-def*
by (*metis (mono-tags, lifting) ⟨x ∈ E⟩ ⟨x ∈ all-edges vs⟩ mem-Collect-eq*)

qed

lemma *singleton-independent-set*: $v \in V \implies \text{is-independent-set } \{v\}$
by (*metis empty-subsetI insert-absorb2 insert-subset is-independent-alt singletonD singleton-not-edge vert-adj-def*)

definition *independent-sets* :: 'a set set **where**

independent-sets \equiv {vs. *is-independent-set* vs}

definition *independence-number* :: enat **where**

independence-number \equiv SUP vs \in *independent-sets*. enat (card vs)

abbreviation $\alpha \equiv$ *independence-number*

lemma *independent-sets-mono*:

vs \in *independent-sets* \implies us \subseteq vs \implies us \in *independent-sets*

using *Int-mono*[*OF all-edges-mono*, of us vs *E E*]

unfolding *independent-sets-def is-independent-set-def* **by** *auto*

lemma *le-independence-iff*:

assumes $0 < k$

shows $k \leq \alpha \iff k \in \text{card } \text{'independent-sets}$ (**is** ?L \iff ?R)

proof

assume ?L

then obtain vs **where** vs \in *independent-sets* **and** klt: $k \leq \text{card } vs$

using *assms* **unfolding** *independence-number-def enat-le-Sup-iff* **by** *auto*

moreover

obtain us **where** us \subseteq vs **and** $k = \text{card } us$

using *card-Ex-subset* klt **by** *auto*

ultimately

have us \in *independent-sets* **by** (*auto intro: independent-sets-mono*)

then show ?R **using** $\langle k = \text{card } us \rangle$ **by** *auto*

qed (*auto intro: SUP-upper simp: independence-number-def*)

lemma *zero-less-independence*:

assumes $V \neq \{\}$

shows $0 < \alpha$

proof –

from *assms* **obtain** a **where** a \in V **by** *auto*

then have $0 < \text{enat } (\text{card } \{a\})$ $\{a\} \in$ *independent-sets*

using *independent-sets-def is-independent-set-def all-edges-def singleton-independent-set*

by *simp-all*

then show ?thesis **unfolding** *independence-number-def less-SUP-iff* ..

qed

end

context *fin-sgraph*

begin

lemma *fin-independent-sets*: *finite (independent-sets)*

unfolding *independent-sets-def is-independent-set-def* **using** *finV* **by** *auto*

lemma *independence-le-card*:

shows $\alpha \leq \text{card } V$

```

proof –
  { fix  $x$  assume  $x \in \text{independent-sets}$ 
    then have  $x \subseteq V$  by (auto simp: independent-sets-def is-independent-set-def)
  }
  with  $\text{fin } V$  show ?thesis unfolding independence-number-def
    by (intro SUP-least) (auto intro: card-mono)
qed

```

```

lemma independence-fin:  $\alpha \neq \infty$ 
  using independence-le-card by (cases  $\alpha$ ) auto

```

```

lemma independence-max-alt:  $V \neq \{\}$   $\implies \alpha = \text{Max } ((\lambda \text{ vs } . \text{enat } (\text{card } \text{vs})) \text{ ‘ }
\text{independent-sets})$ 
  unfolding independence-number-def using Sup-enat-def zero-less-independence
  by (metis i0-less independence-fin independence-number-def)

```

```

lemma independent-sets-ne:
  assumes  $V \neq \{\}$ 
  shows independent-sets  $\neq \{\}$ 
proof –
  from assms obtain  $a$  where  $a \in V$  by auto
  then have  $\{a\} \in \text{independent-sets}$  using independent-sets-def singleton-independent-set
by simp
  thus ?thesis by blast
qed

```

```

lemma independence-obtains:
  assumes  $V \neq \{\}$ 
  obtains  $vs$  where is-independent-set  $vs$  and  $\text{card } vs = \alpha$ 
proof –
  have  $\alpha = \text{Max } ((\lambda \text{ vs } . \text{enat } (\text{card } \text{vs})) \text{ ‘ } \text{independent-sets})$  using independence-max-alt assms by simp
  then obtain  $vs$  where  $vs \in \text{independent-sets}$  and  $\text{enat } (\text{card } vs) = \alpha$ 
  using obtains-MIN[of independent-sets  $\lambda \text{ vs } . \text{enat } (\text{card } \text{vs})$ ] assms fin-independent-sets
independent-sets-ne
  by (metis (no-types, lifting) Max-in finite-imageI imageE image-is-empty)
  thus ?thesis using independent-sets-def that by simp
qed
end
end

```

5 Triangles in Graph

Triangles are an important tool in graph theory. This theory presents a number of basic definitions/lemmas which are useful for general reasoning using triangles. The definitions and lemmas in this theory are adapted from previous less general work in [2] and [1]

```

theory Graph-Triangles imports Undirected-Graph-Basics

```

begin

Triangles don't make as much sense in a loop context, hence we restrict this to simple graphs

context *sgraph*

begin

definition *triangle-in-graph* :: 'a ⇒ 'a ⇒ 'a ⇒ bool **where**
triangle-in-graph x y z ≡ ({x,y} ∈ E) ∧ ({y,z} ∈ E) ∧ ({x,z} ∈ E)

lemma *triangle-in-graph-edge-empty*: E = {} ⇒ ¬ *triangle-in-graph* x y z
using *triangle-in-graph-def* **by** *auto*

definition *triangle-triples* **where**
triangle-triples X Y Z ≡ {(x,y,z) ∈ X × Y × Z. *triangle-in-graph* x y z }

definition

unique-triangles
 ≡ ∀ e ∈ E. ∃! T. ∃ x y z. T = {x,y,z} ∧ *triangle-in-graph* x y z ∧ e ⊆ T

definition *triangle-set* :: 'a set set
where *triangle-set* ≡ { {x,y,z} | x y z. *triangle-in-graph* x y z }

5.1 Preliminaries on Triangles in Graphs

lemma *card-triangle-triples-rotate*: card (*triangle-triples* X Y Z) = card (*triangle-triples* Y Z X)

proof –

have *triangle-triples* Y Z X = (λ(x,y,z). (y,z,x)) ‘ *triangle-triples* X Y Z
by (*auto simp: triangle-triples-def case-prod-unfold image-iff insert-commute triangle-in-graph-def*)
moreover have *inj-on* (λ(x, y, z). (y, z, x)) (*triangle-triples* X Y Z)
by (*auto simp: inj-on-def*)
ultimately show ?thesis
by (*simp add: card-image*)

qed

lemma *triangle-commu1*:
assumes *triangle-in-graph* x y z
shows *triangle-in-graph* y x z
using *assms triangle-in-graph-def* **by** (*auto simp add: insert-commute*)

lemma *triangle-vertices-distinct1*:
assumes *tri: triangle-in-graph* x y z
shows x ≠ y

proof (*rule ccontr*)
assume a: ¬ x ≠ y
have card {x, y} = 2 **using** *tri triangle-in-graph-def*

using *wellformed* **by** (*simp add: two-edges*)
thus *False* **using** *a* **by** *simp*
qed

lemma *triangle-vertices-distinct2*:
assumes *triangle-in-graph x y z*
shows $y \neq z$
by (*metis assms triangle-vertices-distinct1 triangle-in-graph-def*)

lemma *triangle-vertices-distinct3*:
assumes *triangle-in-graph x y z*
shows $z \neq x$
by (*metis assms triangle-vertices-distinct1 triangle-in-graph-def*)

lemma *triangle-in-graph-edge-point*: $\text{triangle-in-graph } x \ y \ z \longleftrightarrow \{y, z\} \in E \wedge \text{vert-adj } x \ y \wedge \text{vert-adj } x \ z$
by (*auto simp add: triangle-in-graph-def vert-adj-def*)

lemma *edge-vertices-not-equal*:
assumes $\{x, y\} \in E$
shows $x \neq y$
using *assms two-edges* **by** *fastforce*

lemma *edge-btw-vertices-not-equal*:
assumes $(x, y) \in \text{all-edges-between } X \ Y$
shows $x \neq y$
using *edge-vertices-not-equal all-edges-between-def*
by (*metis all-edges-betw-D3 assms*)

lemma *mk-triangle-from-ss-edges*:
assumes $(x, y) \in \text{all-edges-between } X \ Y$ **and** $(x, z) \in \text{all-edges-between } X \ Z$ **and**
 $(y, z) \in \text{all-edges-between } Y \ Z$
shows $(\text{triangle-in-graph } x \ y \ z)$
by (*meson all-edges-betw-D3 assms triangle-in-graph-def*)

lemma *triangle-in-graph-verts*:
assumes *triangle-in-graph x y z*
shows $x \in V \ y \in V \ z \in V$
proof –
show $x \in V$ **using** *triangle-in-graph-def wellformed-alt-fst assms* **by** *blast*
show $y \in V$ **using** *triangle-in-graph-def wellformed-alt-snd assms* **by** *blast*
show $z \in V$ **using** *triangle-in-graph-def wellformed-alt-snd assms* **by** *blast*
qed

lemma *convert-triangle-rep-ss*:
assumes $X \subseteq V$ **and** $Y \subseteq V$ **and** $Z \subseteq V$
shows $\text{mk-triangle-set } \{(x, y, z) \in X \times Y \times Z . (\text{triangle-in-graph } x \ y \ z)\} \subseteq \text{triangle-set}$
by (*auto simp add: subsetI triangle-set-def*) (*auto*)

```

lemma (in fin-sgraph) finite-triangle-set: finite (triangle-set)
proof -
  have triangle-set  $\subseteq$  Pow V
  using insert-iff wellformed triangle-in-graph-def triangle-set-def by auto
  then show ?thesis
    by (meson fin V finite-Pow-iff infinite-super)
qed

lemma card-triangle-3:
  assumes  $t \in$  triangle-set
  shows card t = 3
  using assms by (auto simp: triangle-set-def edge-vertices-not-equal triangle-in-graph-def)

lemma triangle-set-power-set-ss: triangle-set  $\subseteq$  Pow V
  by (auto simp add: triangle-set-def triangle-in-graph-def wellformed-alt-fst well-
  formed-alt-snd)

lemma triangle-in-graph-ss:
  assumes  $E' \subseteq E$ 
  assumes sgraph.triangle-in-graph  $E' x y z$ 
  shows triangle-in-graph  $x y z$ 
proof -
  interpret gnew: sgraph V  $E'$ 
  apply (unfold-locales)
  using assms wellformed two-edges by auto
  have  $\{x, y\} \in E$  using assms gnew.triangle-in-graph-def by auto
  have  $\{y, z\} \in E$  using assms gnew.triangle-in-graph-def by auto
  have  $\{x, z\} \in E$  using assms gnew.triangle-in-graph-def by auto
  thus ?thesis
    by (simp add:  $\langle \{x, y\} \in E \rangle \langle \{y, z\} \in E \rangle$  triangle-in-graph-def)
qed

lemma triangle-set-graph-edge-ss:
  assumes  $E' \subseteq E$ 
  shows (sgraph.triangle-set  $E'$ )  $\subseteq$  (triangle-set)
proof (intro subsetI)
  interpret gnew: sgraph V  $E'$ 
  using assms wellformed two-edges by (unfold-locales) auto
  fix t assume  $t \in$  gnew.triangle-set
  then obtain  $x y z$  where  $t = \{x, y, z\}$  and gnew.triangle-in-graph  $x y z$ 
  using gnew.triangle-set-def assms mem-Collect-eq by auto
  then have triangle-in-graph  $x y z$  using assms triangle-in-graph-ss by simp
  thus  $t \in$  triangle-set using triangle-set-def assms
  using  $\langle t = \{x, y, z\} \rangle$  by auto
qed

lemma (in fin-sgraph) triangle-set-graph-edge-ss-bound:
  assumes  $E' \subseteq E$ 

```

```

shows  $\text{card } (\text{triangle-set}) \geq \text{card } (\text{sgraph.triangle-set } E')$ 
using triangle-set-graph-edge-ss finite-triangle-set
by (simp add: assms card-mono)

end

locale triangle-free-graph = sgraph +
  assumes tri-free:  $\neg(\exists x y z. \text{triangle-in-graph } x y z)$ 

lemma triangle-free-graph-empty:  $E = \{\} \implies \text{triangle-free-graph } V E$ 
apply (unfold-locales, simp-all)
using sgraph.triangle-in-graph-edge-empty
by (metis Int-absorb all-edges-disjoint complete-sgraph)

context fin-sgraph
begin

  Converting between ordered and unordered triples for reasoning on cardinality

lemma card-convert-triangle-rep:
  assumes  $X \subseteq V$  and  $Y \subseteq V$  and  $Z \subseteq V$ 
  shows  $\text{card } (\text{triangle-set}) \geq 1/6 * \text{card } \{(x, y, z) \in X \times Y \times Z . (\text{triangle-in-graph } x y z)\}$ 
  (is  $\geq 1/6 * \text{card } ?TT$ )
proof –
  define tofl where  $\text{tofl} \equiv \lambda l::'a \text{ list. } (\text{hd } l, \text{hd}(\text{tl } l), \text{hd}(\text{tl}(\text{tl } l)))$ 
  have in-tofl:  $(x, y, z) \in \text{tofl } \text{'permutations-of-set } \{x,y,z\}$  if  $x \neq y$   $y \neq z$   $x \neq z$  for  $x$ 
   $y z$ 
  proof –
  have distinct[ $x,y,z$ ]
  using that by simp
  then show ?thesis
  unfolding tofl-def image-iff
  by (smt (verit, best) list.sel(1) list.sel(3) list.simps(15) permutations-of-setI set-empty)
  qed
  have  $?TT \subseteq \{(x, y, z). (\text{triangle-in-graph } x y z)\}$ 
  by auto
  also have  $\dots \subseteq (\bigcup t \in \text{triangle-set. } \text{tofl } \text{'permutations-of-set } t)$ 
  proof (clarsimp simp: triangle-set-def)
  fix  $u v w$ 
  assume  $t: \text{triangle-in-graph } u v w$ 
  then have  $(u, v, w) \in \text{tofl } \text{'permutations-of-set } \{u,v,w\}$ 
  by (metis in-tofl triangle-commu1 triangle-vertices-distinct1 triangle-vertices-distinct2)
  with  $t$  show  $\exists t. (\exists x y z. t = \{x, y, z\} \wedge \text{triangle-in-graph } x y z) \wedge (u, v, w)$ 
   $\in \text{tofl } \text{'permutations-of-set } t$ 
  by blast
  qed
  finally have  $?TT \subseteq (\bigcup t \in \text{triangle-set. } \text{tofl } \text{'permutations-of-set } t)$  .

```

```

then have card ?TT ≤ card(∪ t ∈ triangle-set. tofl ‘ permutations-of-set t)
  by (intro card-mono finite-UN-I finite-triangle-set) (auto simp: assms)
also have ... ≤ (∑ t ∈ triangle-set. card (tofl ‘ permutations-of-set t))
  using card-UN-le finV finite-triangle-set wellformed by blast
also have ... ≤ (∑ t ∈ triangle-set. card (permutations-of-set t))
  by (meson card-image-le finite-permutations-of-set sum-mono)
also have ... ≤ (∑ t ∈ triangle-set. fact 3)
  by(rule sum-mono) (metis card.infinite card-permutations-of-set card-triangle-3
eq-refl nat.simps(3) numeral-3-eq-3)
also have ... = 6 * card (triangle-set)
  by (simp add: eval-nat-numeral)
finally have card ?TT ≤ 6 * card (triangle-set) .
then show ?thesis
  by (simp add: divide-simps)
qed

```

lemma *card-convert-triangle-rep-bound*:

```

fixes t :: real
assumes card {(x, y, z) ∈ X × Y × Z . (triangle-in-graph x y z)} ≥ t
assumes X ⊆ V and Y ⊆ V and Z ⊆ V
shows card (triangle-set) ≥ 1/6 * t
proof -
  define t' where t' ≡ card {(x, y, z) ∈ X × Y × Z . (triangle-in-graph x y z)}
  have t' ≥ t using assms t'-def by simp
  then have tgt: 1/6 * t' ≥ 1/6 * t by simp
  have card (triangle-set) ≥ 1/6 * t' using t'-def card-convert-triangle-rep assms
  by simp
  thus ?thesis using tgt by linarith
qed
end
end
theory Bipartite-Graphs imports Undirected-Graph-Walks
begin

```

6 Bipartite Graphs

An introductory library for reasoning on bipartite graphs.

6.1 Bipartite Set Up

All "edges", i.e. pairs, between any two sets

definition *all-bi-edges* :: 'a set ⇒ 'a set ⇒ 'a edge set **where**
all-bi-edges X Y ≡ *mk-edge* ‘ (X × Y)

lemma *all-bi-edges-alt*:

```

assumes X ∩ Y = {}
shows all-bi-edges X Y = {e . card e = 2 ∧ e ∩ X ≠ {} ∧ e ∩ Y ≠ {}}
unfolding all-bi-edges-def

```

proof (*intro subset-antisym subsetI*)
fix e **assume** $e \in \text{mk-edge } \langle X \times Y \rangle$
then obtain $v1\ v2$ **where** $e = \{v1, v2\}$ **and** $v1 \in X$ **and** $v2 \in Y$
by *auto*
then show $e \in \{e. \text{card } e = 2 \wedge e \cap X \neq \{\}\ \wedge e \cap Y \neq \{\}\}$ **using** *assms*
using *card-2-iff* **by** *blast*
next
fix e' **assume** *assm*: $e' \in \{e. \text{card } e = 2 \wedge e \cap X \neq \{\}\ \wedge e \cap Y \neq \{\}\}$
then obtain $v1$ **where** $v1 \text{in}$: $v1 \in e'$ **and** $v1 \in X$
by *blast*
moreover obtain $v2$ **where** $v2 \text{in}$: $v2 \in e'$ **and** $v2 \in Y$ **using** *assm* **by** *blast*
then have ne : $v1 \neq v2$
using *assms calculation(2)* **by** *blast*
have $\text{card } e' = 2$ **using** *assm* **by** *blast*
have $\{v1, v2\} \subseteq e'$ **using** $v1 \text{in } v2 \text{in}$ **by** *blast*
then have $e' = \{v1, v2\}$ **using** *assm v1in v2in*
by (*metis (no-types, opaque-lifting) <card e' = 2> card-2-iff' insertCI ne subsetI subset-antisym*)
then show $e' \in \text{mk-edge } \langle X \times Y \rangle$
by (*simp add: <v2 \in Y> calculation(2) in-mk-edge-img*)
qed

lemma *all-bi-edges-alt2*: $\text{all-bi-edges } X\ Y = \{\{x, y\} \mid x\ y. x \in X \wedge y \in Y\}$
unfolding *all-bi-edges-def*

proof (*intro subset-antisym subsetI*)
fix x **assume** $x \in \text{mk-edge } \langle X \times Y \rangle$
then obtain $a\ b$ **where** $(a, b) \in (X \times Y)$ **and** $x \text{eq}$: $x = \text{mk-edge } (a, b)$ **by** *blast*
then show $x \in \{\{x, y\} \mid x\ y. x \in X \wedge y \in Y\}$
by *auto*
next
fix x **assume** $x \in \{\{x, y\} \mid x\ y. x \in X \wedge y \in Y\}$
then obtain $a\ b$ **where** $x \text{eq}$: $x = \{a, b\}$ **and** $a \in X$ **and** $b \in Y$
by *blast*
then have $(a, b) \in (X \times Y)$ **by** *auto*
then show $x \in \text{mk-edge } \langle X \times Y \rangle$ **using** *in-mk-edge-img xeq* **by** *metis*
qed

lemma *all-bi-edges-wf*: $e \in \text{all-bi-edges } X\ Y \implies e \subseteq X \cup Y$
by (*auto simp add: all-bi-edges-alt2*)

lemma *all-bi-edges-2*: $X \cap Y = \{\} \implies e \in \text{all-bi-edges } X\ Y \implies \text{card } e = 2$
using *card-2-iff* **by** (*auto simp add: all-bi-edges-alt2*)

lemma *all-bi-edges-main*: $X \cap Y = \{\} \implies \text{all-bi-edges } X\ Y \subseteq \text{all-edges } (X \cup Y)$
unfolding *all-edges-def* **using** *all-bi-edges-wf all-bi-edges-2* **by** *blast*

lemma *all-bi-edges-finite*: $\text{finite } X \implies \text{finite } Y \implies \text{finite } (\text{all-bi-edges } X\ Y)$
by (*simp add: all-bi-edges-def*)

lemma *all-bi-edges-not-ssX*: $X \cap Y = \{\}$ $\implies e \in \text{all-bi-edges } X \ Y \implies \neg e \subseteq X$
by (*auto simp add: all-bi-edges-alt*)

lemma *all-bi-edges-sym*: $\text{all-bi-edges } X \ Y = \text{all-bi-edges } Y \ X$
by (*auto simp add: all-bi-edges-alt2*)

lemma *all-bi-edges-not-ssY*: $X \cap Y = \{\}$ $\implies e \in \text{all-bi-edges } X \ Y \implies \neg e \subseteq Y$
by (*auto simp add: all-bi-edges-alt*)

lemma *card-all-bi-edges*:
assumes *finite X finite Y*
assumes $X \cap Y = \{\}$
shows $\text{card } (\text{all-bi-edges } X \ Y) = \text{card } X * \text{card } Y$
proof –
have $\text{card } (\text{all-bi-edges } X \ Y) = \text{card } (X \times Y)$
unfolding *all-bi-edges-def* **using** *inj-on-mk-edge assms card-image* **by** *blast*
thus *?thesis* **using** *card-cartesian-product* **by** *auto*
qed

lemma (*in sgraph*) *all-edges-between-bi-subset*: *mk-edge* ‘*all-edges-between* $X \ Y \subseteq$
all-bi-edges $X \ Y$
by (*auto simp: all-edges-between-def all-bi-edges-def*)

6.2 Bipartite Graph Locale

For reasoning purposes, it is useful to explicitly label the two sets of vertices as X and Y . These are parameters in the locale

locale *bipartite-graph* = *graph-system* +
fixes $X \ Y :: 'a \ \text{set}$
assumes *partition: partition-on* $V \ \{X, Y\}$
assumes *ne*: $X \neq Y$
assumes *edge-betw*: $e \in E \implies e \in \text{all-bi-edges } X \ Y$
begin

lemma *part-intersect-empty*: $X \cap Y = \{\}$
using *partition-onD2 partition disjointD ne*
by *blast*

lemma *X-not-empty*: $X \neq \{\}$
using *partition partition-onD3* **by** *auto*

lemma *Y-not-empty*: $Y \neq \{\}$
using *partition partition-onD3* **by** *auto*

lemma *XY-union*: $X \cup Y = V$
using *partition partition-onD1* **by** *auto*

lemma *card-edges-two*: $e \in E \implies \text{card } e = 2$
using *edge-betw all-bi-edges-alt part-intersect-empty* **by** *auto*

lemma *partitions-ss*: $X \subseteq V \ Y \subseteq V$
using *XY-union* **by** *auto*

end

By definition, we say an edge must be between X and Y, i.e. contains two vertices

sublocale *bipartite-graph* \subseteq *sgraph*
using *card-edges-two* **by** (*unfold-locales*)

context *bipartite-graph*
begin

abbreviation *density* \equiv *edge-density* *X* *Y*

lemma *bipartite-sym*: *bipartite-graph* *V* *E* *Y* *X*
using *partition ne edge-betw all-bi-edges-sym*
by (*unfold-locales*) (*auto simp add: insert-commute*)

lemma *X-verts-not-adj*:
assumes $x1 \in X \ x2 \in X$
shows \neg *vert-adj* *x1* *x2*
proof (*rule ccontr, simp add: vert-adj-def*)
assume $\{x1, x2\} \in E$
then have $\neg \{x1, x2\} \subseteq X$
using *all-bi-edges-not-ssX edge-betw part-intersect-empty* **by** *auto*
then show *False* **using** *assms* **by** *auto*
qed

lemma *Y-verts-not-adj*:
assumes $y1 \in Y \ y2 \in Y$
shows \neg *vert-adj* *y1* *y2*
proof –
interpret *sym*: *bipartite-graph* *V* *E* *Y* *X* **using** *bipartite-sym* **by** *simp*
show *?thesis* **using** *sym.X-verts-not-adj*
by (*simp add: assms(1) assms(2)*)
qed

lemma *X-vert-adj-Y*: $x \in X \implies$ *vert-adj* *x* *y* $\implies y \in Y$
using *X-verts-not-adj XY-union vert-adj-imp-inV* **by** *blast*

lemma *Y-vert-adj-X*: $y \in Y \implies$ *vert-adj* *y* *x* $\implies x \in X$
using *Y-verts-not-adj XY-union vert-adj-imp-inV* **by** *blast*

lemma *neighbors-ss-eq-neighborhoodX*: $v \in X \implies$ *neighborhood* *v* $=$ *neighbors-ss* *v* *Y*
unfolding *neighborhood-def neighbors-ss-def*
by(*auto simp add: X-vert-adj-Y vert-adj-imp-inV*)

lemma *neighbors-ss-eq-neighborhoodY*: $v \in Y \implies \text{neighborhood } v = \text{neighbors-ss } v X$

unfolding *neighborhood-def neighbors-ss-def*
by(*auto simp add: Y-vert-adj-X vert-adj-imp-in V*)

lemma *neighborhood-subset-oppX*: $v \in X \implies \text{neighborhood } v \subseteq Y$
using *neighbors-ss-eq-neighborhoodX neighbors-ss-def* **by** *auto*

lemma *neighborhood-subset-oppY*: $v \in Y \implies \text{neighborhood } v \subseteq X$
using *neighbors-ss-eq-neighborhoodY neighbors-ss-def* **by** *auto*

lemma *degree-neighbors-ssX*: $v \in X \implies \text{degree } v = \text{card } (\text{neighbors-ss } v Y)$
using *neighbors-ss-eq-neighborhoodX alt-deg-neighborhood* **by** *auto*

lemma *degree-neighbors-ssY*: $v \in Y \implies \text{degree } v = \text{card } (\text{neighbors-ss } v X)$
using *neighbors-ss-eq-neighborhoodY alt-deg-neighborhood* **by** *auto*

definition *is-bicomplete*:: *bool* **where**
is-bicomplete $\equiv E = \text{all-bi-edges } X Y$

lemma *edge-betw-indiv*:

assumes $e \in E$

obtains $x y$ **where** $x \in X \wedge y \in Y \wedge e = \{x, y\}$

proof –

have $e \in \{\{x, y\} \mid x y. x \in X \wedge y \in Y\}$

using *edge-betw all-bi-edges-alt2 assms* **by** *blast*

thus *?thesis*

using *that* **by** *auto*

qed

lemma *edges-between-equals-edge-set*: *mk-edge* ‘ $(\text{all-edges-between } X Y) = E$

by (*simp add: all-edges-between-set, intro subset-antisym subsetI, auto*) (*metis edge-betw-indiv*)

Lemmas for reasoning on walks and paths in a bipartite graph

lemma *walk-alternates*:

assumes *is-walk* w

assumes $\text{Suc } i < \text{length } w \wedge i \geq 0$

shows $w ! i \in X \longleftrightarrow w ! (i + 1) \in Y$

proof –

have $\{w ! i, w ! (i + 1)\} \in E$ **using** *is-walk-index assms* **by** *auto*

then show *?thesis*

using *X-vert-adj-Y not-vert-adj Y-vert-adj-X vert-adj-sym* **by** *blast*

qed

A useful reasoning pattern to mimic "wlog" statements for properties that are symmetric is to interpret the symmetric bipartite graph and then directly apply the lemma proven earlier


```

lemma walk-alternates-sym:
  assumes is-walk w
  assumes Suc i < length w i ≥ 0
  shows  $w ! i \in Y \iff w ! (i + 1) \in X$ 
proof –
  interpret sym: bipartite-graph V E Y X using bipartite-sym by simp
  show ?thesis using sym.walk-alternates assms by simp
qed

lemma walk-length-even:
  assumes is-walk w
  assumes hd w ∈ X and last w ∈ X
  shows even (walk-length w)
  using assms
proof (induct length w arbitrary: w rule: nat-induct2)
  case 0
  then show ?case by (auto simp add: is-walk-def)
next
  case 1
  then have walk-length w = 0 using walk-length-conv by auto
  then show ?case by simp
next
  case (step n)
  then show ?case proof (cases n = 0)
    case True
    then have length w = 2 using step by simp
    then have hd w ∈ X ⇒ last w ∈ Y using walk-alternates hd-conv-nth
last-conv-nth
    by (metis add-0 add-diff-cancel-right' less-2-cases-iff list.size(3) nat-1-add-1
step.prem(1)
    zero-le zero-neq-numeral)
    then show ?thesis
    using part-intersect-empty step.prem(2) step.prem(3) by blast
  next
  case False
  have IH: (∧w. n = length w ⇒ is-walk w ⇒ hd w ∈ X ⇒ last w ∈ X ⇒
even (walk-length w))
  using step by simp
  obtain w1 w2 where weq: w = w1@w2 and w1: w1 = take n w and w2: w2
  = drop n w
  by simp
  then have ne: w1 ≠ [] using False is-walk-not-empty2 step.prem(1) by fastforce

  then have w1-walk: is-walk w1 using w1 is-walk-take False
  by (metis nat-le-linear neq0-conv step.prem(1) take-all)
  have hdw1: hd w1 ∈ X using step ne weq by auto
  then have w1n: length w1 = n using step length-take w1 by auto
  then have length w2 = 2 using step length-drop
  by (simp add: w2)

```

```

have last w = w ! (n + 1) using step last-conv-nth is-walk-not-empty
  by (metis add.left-commute diff-add-inverse nat-1-add-1)
then have w ! n ∈ Y using step by (simp add: walk-alternates-sym)
then have w ! (n - 1) ∈ X using False walk-alternates step by simp
then have last w1 ∈ X using step last-conv-nth[of w1] ne w1n
  by (metis last-list-update list-update-id take-update-swap w1)
then have even (walk-length w1) using w1-walk w1n hdw1 IH[of w1] by simp
then have even (walk-length w1 + 2) by simp
then show ?thesis using walk-length-conv weq step
  by (simp add: False w1n)
qed
qed

```

lemma *walk-length-even-sym*:

```

assumes is-walk w
assumes hd w ∈ Y
assumes last w ∈ Y
shows even (walk-length w)
proof -
interpret sym: bipartite-graph V E Y X using bipartite-sym by simp
show ?thesis using sym.walk-length-even assms by auto
qed

```

lemma *walk-length-odd*:

```

assumes is-walk w
assumes hd w ∈ X and last w ∈ Y
shows odd (walk-length w)
using assms
proof (cases length w ≥ 2)
case True
then have hdin: hd (tl w) ∈ Y using walk-alternates hd-conv-nth
  by (metis (mono-tags, lifting) Suc-1 Suc-less-eq2 assms(1) assms(2) is-walk-not-empty2
is-walk-tl
le-neq-implies-less le-numeral-extra(3) length-greater-0-conv less-Suc-eq nth-tl
numeral-1-eq-Suc-0 numerals(1) plus-nat.add-0)
have w: is-walk (tl w) using assms True is-walk-tl by auto
have last: last (tl w) ∈ Y using assms(3) by (simp add: is-walk-not-empty last-tl
w)
then have ev: even (walk-length (tl w)) using hdin w walk-length-even-sym[of
tl w] by auto
then have walk-length w = walk-length (tl w) + 1 using True walk-length-conv
by auto
then show ?thesis using ev by simp
next
case False
have length w ≠ 0 using is-walk-not-empty assms by simp
then have length w = 1 using False by linarith
then have hd w = last w

```

using $\langle \text{length } w \neq 0 \rangle$ *hd-conv-nth last-conv-nth* **by** *fastforce*
then have $hd\ w \in X \implies last\ w \notin Y$ **using** *part-intersect-empty* **by** *auto*
then show *?thesis* **using** *assms* **by** *simp*
qed

lemma *walk-length-odd-sym*:

assumes *is-walk w*
assumes $hd\ w \in Y$ **and** $last\ w \in X$
shows *odd (walk-length w)*
proof –
interpret *sym: bipartite-graph V E Y X* **using** *bipartite-sym* **by** *simp*
show *?thesis* **using** *assms sym.walk-length-odd* **by** *simp*
qed

lemma *walk-length-even-iff*:

assumes *is-walk w*
shows $even\ (walk\ length\ w) \iff (hd\ w \in X \wedge last\ w \in X) \vee (hd\ w \in Y \wedge last\ w \in Y)$
proof (*intro iffI*)
assume *ev: even (walk-length w)*
show $hd\ w \in X \wedge last\ w \in X \vee hd\ w \in Y \wedge last\ w \in Y$
proof (*rule ccontr*)
assume $\neg ((hd\ w \in X \wedge last\ w \in X) \vee (hd\ w \in Y \wedge last\ w \in Y))$
then have $(hd\ w \notin X \vee last\ w \notin X) \wedge (hd\ w \notin Y \vee last\ w \notin Y)$ **by** *simp*
then have $(hd\ w \in Y \vee last\ w \in Y) \wedge (hd\ w \in X \vee last\ w \in X)$ **using** *part-intersect-empty*
using *XY-union assms is-walk-wf-hd is-walk-wf-last* **by** *auto*
then have *split: (hd w ∈ X ∧ last w ∈ Y) ∨ (hd w ∈ Y ∧ last w ∈ X)*
using *part-intersect-empty* **by** *auto*
have *o1: (hd w ∈ X ∧ last w ∈ Y) ⇒ odd (walk-length w)* **using** *walk-length-odd assms* **by** *auto*
have $(hd\ w \in Y \wedge last\ w \in X) \implies odd\ (walk\ length\ w)$ **using** *walk-length-odd-sym assms* **by** *auto*
then show *False* **using** *split ev o1* **by** *auto*
qed
next
show $(hd\ w \in X \wedge last\ w \in X) \vee (hd\ w \in Y \wedge last\ w \in Y) \implies even\ (walk\ length\ w)$
using *walk-length-even walk-length-even-sym assms* **by** *auto*
qed

lemma *walk-length-odd-iff*:

assumes *is-walk w*
shows $odd\ (walk\ length\ w) \iff (hd\ w \in X \wedge last\ w \in Y) \vee (hd\ w \in Y \wedge last\ w \in X)$
proof (*intro iffI*)
assume *o: odd (walk-length w)*
show $(hd\ w \in X \wedge last\ w \in Y) \vee (hd\ w \in Y \wedge last\ w \in X)$
proof (*rule ccontr*)

```

assume  $\neg ((hd\ w \in X \wedge last\ w \in Y) \vee (hd\ w \in Y \wedge last\ w \in X))$ 
then have  $(hd\ w \notin X \vee last\ w \notin Y) \wedge (hd\ w \notin Y \vee last\ w \notin X)$  by simp
then have  $(hd\ w \in Y \vee last\ w \in X) \wedge (hd\ w \in X \vee last\ w \in Y)$  using
part-intersect-empty
using XY-union assms is-walk-wf-hd is-walk-wf-last by auto
then have split:  $(hd\ w \in X \wedge last\ w \in X) \vee (hd\ w \in Y \wedge last\ w \in Y)$ 
using part-intersect-empty by auto
have e1:  $(hd\ w \in X \wedge last\ w \in X) \implies even\ (walk-length\ w)$  using walk-length-even
assms by auto
have  $(hd\ w \in Y \wedge last\ w \in Y) \implies even\ (walk-length\ w)$  using walk-length-even-sym
assms by auto
then show False using split o e1 by auto
qed
next
show  $(hd\ w \in X \wedge last\ w \in Y) \vee (hd\ w \in Y \wedge last\ w \in X) \implies odd\ (walk-length\ w)$ 
using walk-length-odd walk-length-odd-sym assms by auto
qed

```

Classic basic theorem that a bipartite graph must not have any cycles with an odd length

lemma *no-odd-cycles*:

```

assumes is-walk w
assumes odd (walk-length w)
shows  $\neg is-cycle\ w$ 

```

proof –

```

have  $(hd\ w \in X \wedge last\ w \in Y) \vee (hd\ w \in Y \wedge last\ w \in X)$  using assms
walk-length-odd-iff by auto
then have  $hd\ w \neq last\ w$  using part-intersect-empty by auto
thus ?thesis using is-cycle-def is-closed-walk-def by simp
qed

```

end

A few properties rely on cardinality definitions that require the vertex sets to be finite

locale *fin-bipartite-graph = bipartite-graph + fin-graph-system*

begin

lemma *fin-bipartite-sym*: *fin-bipartite-graph V E Y X*

by (*intro-locales*) (*simp add: bipartite-sym bipartite-graph.axioms(2)*)

lemma *partitions-finite*: *finite X finite Y*

using *partitions-ss finite-subset finV* **by** *auto*

lemma *card-edges-between-set*: *card (all-edges-between X Y) = card E*

proof –

```

have  $card\ (all-edges-between\ X\ Y) = card\ (mk-edge\ ' (all-edges-between\ X\ Y))$ 
using inj-on-mk-edge using partitions-finite card-image

```

```

    by (metis inj-on-mk-edge part-intersect-empty)
  then show ?thesis by (simp add: edges-between-equals-edge-set)
qed

lemma density-simp: density = card (E) / ((card X) * (card Y))
  unfolding edge-density-def using card-edges-between-set by auto

lemma edge-size-degree-sumY: card E = (∑ y ∈ Y . degree y)
proof -
  have (∑ y ∈ Y . degree y) = (∑ y ∈ Y . card(neighbors-ss y X))
    using degree-neighbors-ssY by (simp)
  also have ... = card (all-edges-between X Y)
    using card-all-edges-betw-neighbor
  by (metis card-all-edges-between-commute partitions-finite(1) partitions-finite(2))

  finally show ?thesis
    by (simp add: card-edges-between-set)
qed

lemma edge-size-degree-sumX: card E = (∑ y ∈ X . degree y)
proof -
  interpret sym: fin-bipartite-graph V E Y X
    using fin-bipartite-sym by simp
  show ?thesis using sym.edge-size-degree-sumY by simp
qed

end
end

```

7 Graph Theory Inheritance

This theory aims to demonstrate the use of locales to transfer theorems between different graph/combinatorial structure representations

```
theory Graph-Theory-Relations imports Undirected-Graph-Basics Bipartite-Graphs
```

```
Design-Theory.Block-Designs Design-Theory.Group-Divisible-Designs
begin
```

7.1 Design Inheritance

A graph is a type of incidence system, and more specifically a type of combinatorial design. This section demonstrates the correspondence between designs and graphs

```
sublocale graph-system ⊆ inc: incidence-system V mset-set E
  by (unfold-locales) (metis wellformed elem-mset-set ex-in-conv infinite-set-mset-mset-set)
```

sublocale *fin-graph-system* \subseteq *finc: finite-incidence-system* *V mset-set E*
using *finV* **by** *unfold-locales*

sublocale *fin-ulgraph* \subseteq *d: design* *V mset-set E*
using *edge-size empty-not-edge fin-edges* **by** *unfold-locales auto*

sublocale *fin-ulgraph* \subseteq *d: simple-design* *V mset-set E*
by *unfold-locales (simp add: fin-edges)*

locale *graph-has-edges* = *graph-system* +
assumes *edges-nempty: E \neq {}*

locale *fin-sgraph-wedges* = *fin-sgraph* + *graph-has-edges*

The simple graph definition of degree overlaps with the definition of a point replication number

sublocale *fin-sgraph-wedges* \subseteq *bd: block-design* *V mset-set E 2*
rewrites *point-replication-number (mset-set E) x = degree x*
and *points-index (mset-set E) vs = degree-set vs*

proof (*unfold-locales*)

show *inc.b \neq 0* **by** (*simp add: edges-nempty fin-edges*)

show $\bigwedge bl. bl \in \# mset-set E \implies card\ bl = 2$ **by** (*simp add: fin-edges two-edges*)

show *mset-set E index vs = degree-set vs*

unfolding *degree-set-def points-index-def* **by** (*simp add: fin-edges*)

next

have *size {#b \in # (mset-set E) . x \in b#} = card (incident-edges x)*

unfolding *incident-edges-def incident-def*

by (*simp add: fin-edges*)

then show *mset-set E rep x = degree x* **using** *alt-degree-def point-replication-number-def*
by *metis*

qed

locale *fin-bipartite-graph-wedges* = *fin-bipartite-graph* + *fin-sgraph-wedges*

sublocale *fin-bipartite-graph-wedges* \subseteq *group-design* *V mset-set E {X, Y}*
by *unfold-locales (simp-all add: partition ne)*

7.2 Adjacency Relation Definition

Another common formal representation of graphs is as a vertex set and an adjacency relation This is a useful representation in some contexts - we use locales to enable the transfer of results between the two representations, specifically the mutual sublocales approach

locale *graph-rel* =

fixes *vertices :: 'a set (V)*

fixes *adj-rel :: 'a rel*

assumes *wf: $\bigwedge u\ v. (u, v) \in adj-rel \implies u \in V \wedge v \in V$*

begin

abbreviation $adj\ u\ v \equiv (u, v) \in adj\text{-}rel$

lemma *wf-alt*: $adj\ u\ v \implies (u, v) \in V \times V$
using *wf* by *blast*

end

locale *ulgraph-rel* = *graph-rel* +
assumes *sym-adj*: *sym adj-rel*
begin

This definition makes sense in the context of an undirected graph

definition *edge-set*:: 'a edge set **where**
 $edge\text{-}set \equiv \{\{u, v\} \mid u\ v.\ adj\ u\ v\}$

lemma *obtain-edge-pair-adj*:
assumes $e \in edge\text{-}set$
obtains $u\ v$ **where** $e = \{u, v\}$ **and** $adj\ u\ v$
using *assms edge-set-def mem-Collect-eq*
by *fastforce*

lemma *adj-to-edge-set-card*:
assumes $e \in edge\text{-}set$
shows $card\ e = 1 \vee card\ e = 2$

proof –

obtain $u\ v$ **where** $e = \{u, v\}$ **and** $adj\ u\ v$ using *obtain-edge-pair-adj assms* by
blast

then show *?thesis* by (*cases u = v, simp-all*)

qed

lemma *adj-to-edge-set-card-lim*:
assumes $e \in edge\text{-}set$
shows $card\ e > 0 \wedge card\ e \leq 2$

proof –

obtain $u\ v$ **where** $e = \{u, v\}$ **and** $adj\ u\ v$ using *obtain-edge-pair-adj assms* by
blast

then show *?thesis* by (*cases u = v, simp-all*)

qed

lemma *edge-set-wf*: $e \in edge\text{-}set \implies e \subseteq V$
using *obtain-edge-pair-adj wf* by (*metis insert-iff singletonD subsetI*)

lemma *is-graph-system*: *graph-system* $V\ edge\text{-}set$
by (*unfold-locales*) (*simp add: edge-set-wf*)

lemma *sym-alt*: $adj\ u\ v \longleftrightarrow adj\ v\ u$
using *sym-adj* by (*meson symE*)

```

lemma is-ulgraph: ulgraph V edge-set
  using ulgraph-axioms-def is-graph-system adj-to-edge-set-card-lim
  by (intro-locales) auto

end

context ulgraph
begin

definition adj-relation :: 'a rel where
adj-relation  $\equiv \{(u, v) \mid u \ v \ . \ \text{vert-adj } u \ v\}$ 

lemma adj-relation-wf: (u, v) ∈ adj-relation ⇒ {u, v} ⊆ V
  unfolding adj-relation-def using vert-adj-imp-in V by auto

lemma adj-relation-sym: sym adj-relation
  unfolding adj-relation-def sym-def using vert-adj-sym by auto

lemma is-ulgraph-rel: ulgraph-rel V adj-relation
  using adj-relation-wf adj-relation-sym by (unfold-locales) auto

  Temporary interpretation - mutual sublocale setup
interpretation ulgraph-rel V adj-relation by (rule is-ulgraph-rel)

lemma vert-adj-rel-iff:
  assumes  $u \in V \ v \in V$ 
  shows  $\text{vert-adj } u \ v \longleftrightarrow \text{adj } u \ v$ 
  using adj-relation-def by auto

lemma edges-rel-is: E = edge-set
proof –
  have  $E = \{\{u, v\} \mid u \ v \ . \ \text{vert-adj } u \ v\}$ 
  proof (intro subset-antisym subsetI)
    show  $\bigwedge x. x \in \{\{u, v\} \mid u \ v \ . \ \text{vert-adj } u \ v\} \implies x \in E$ 
    using vert-adj-def by fastforce
  next
    fix  $x$  assume  $x \in E$ 
    then have  $x \subseteq V$  and  $\text{card } x > 0$  and  $\text{card } x \leq 2$  using wellformed edge-size
  by auto
    then obtain  $u \ v$  where  $x = \{u, v\}$  and  $\{u, v\} \in E$ 
    by (metis  $\langle x \in E \rangle$  alt-edge-size card-1-singletonE card-2-iff insert-absorb2)
    then show  $x \in \{\{u, v\} \mid u \ v \ . \ \text{vert-adj } u \ v\}$  unfolding vert-adj-def by blast
  qed
  then have  $E = \{\{u, v\} \mid u \ v \ . \ \text{adj } u \ v\}$  using vert-adj-rel-iff Collect-cong
  by (smt (verit) local.wf vert-adj-imp-in V)
  thus ?thesis using edge-set-def by simp
qed

```



```

end

context ulgraph-rel
begin

  Temporary interpretation - mutual sublocale setup
  interpretation ulgraph V edge-set by (rule is-ulgraph)

  lemma rel-vert-adj-iff: vert-adj u v  $\longleftrightarrow$  adj u v
  proof (intro iffI)
    assume vert-adj u v
    then have  $\{u, v\} \in \text{edge-set}$  by (simp add: vert-adj-def)
    then show adj u v using edge-set-def
      by (metis (no-types, lifting) doubleton-eq-iff obtain-edge-pair-adj sym-alt)
  next
    assume adj u v
    then have  $\{u, v\} \in \text{edge-set}$  using edge-set-def by auto
    then show vert-adj u v by (simp add: vert-adj-def)
  qed

  lemma rel-item-is:  $(u, v) \in \text{adj-rel} \longleftrightarrow (u, v) \in \text{adj-relation}$ 
    unfolding adj-relation-def using rel-vert-adj-iff by auto

  lemma rel-edges-is:  $\text{adj-rel} = \text{adj-relation}$ 
    using rel-item-is by auto

end

sublocale ulgraph-rel  $\subseteq$  ulgraph V edge-set
  rewrites ulgraph.adj-relation edge-set = adj-rel
  using local.is-ulgraph rel-edges-is by simp-all

sublocale ulgraph  $\subseteq$  ulgraph-rel V adj-relation
  rewrites ulgraph-rel.edge-set adj-relation = E
  using is-ulgraph-rel edges-rel-is by simp-all

locale sgraph-rel = ulgraph-rel +
  assumes irreft-adj: irreft adj-rel
begin

  lemma irreft-alt:  $\text{adj } u \ v \implies u \neq v$ 
    using irreft-adj irreft-def by fastforce

  lemma edge-is-card2:
    assumes  $e \in \text{edge-set}$ 
    shows  $\text{card } e = 2$ 
  proof -
    obtain u v where eq:  $e = \{u, v\}$  and adj u v using assms edge-set-def by blast
    then have  $u \neq v$  using irreft-alt by simp
  qed

```

```

    thus ?thesis using eq by simp
qed

lemma is-sgraph: sgraph V edge-set
  using is-graph-system edge-is-card2 sgraph-axioms-def by (intro-locales) auto

end

context sgraph
begin

lemma is-rel-irrefl-alt:
  assumes  $(u, v) \in \text{adj-relation}$ 
  shows  $u \neq v$ 
proof -
  have  $\text{vert-adj } u \ v$  using adj-relation-def assms by blast
  then have  $\{u, v\} \in E$  using vert-adj-def by simp
  then have  $\text{card } \{u, v\} = 2$  using two-edges by simp
  thus ?thesis by auto
qed

lemma is-rel-irrefl: irrefl adj-relation
  using irrefl-def is-rel-irrefl-alt by auto

lemma is-sgraph-rel: sgraph-rel V adj-relation
  by (unfold-locales) (simp add: is-rel-irrefl)

end

sublocale sgraph-rel  $\subseteq$  sgraph V edge-set
  rewrites ulgraph.adj-relation edge-set = adj-rel
  using is-sgraph rel-edges-is by simp-all

sublocale sgraph  $\subseteq$  sgraph-rel V adj-relation
  rewrites ulgraph-rel.edge-set adj-relation = E
  using is-sgraph-rel edges-rel-is by simp-all

end
theory Undirected-Graphs-Root imports
  Undirected-Graph-Basics
  Undirected-Graph-Walks
  Connectivity
  Girth-Independence
  Graph-Triangles
  Bipartite-Graphs
  Graph-Theory-Relations
begin
end

```

References

- [1] C. Edmonds, A. Koutsoukou-Argyaki, and L. C. Paulson. Roth's Theorem on Arithmetic Progressions. *Archive of Formal Proofs*, Dec. 2021.
- [2] C. Edmonds, A. Koutsoukou-Argyaki, and L. C. Paulson. Szemerédi's Regularity Lemma. *Archive of Formal Proofs*, Nov. 2021.
- [3] L. Hupel. Properties of random graphs – subgraph containment. *Archive of Formal Proofs*, February 2014. https://isa-afp.org/entries/Random_Graph_Subgraph_Threshold.html, Formal proof development.
- [4] L. Noschinski. Proof Pearl: A Probabilistic Proof for the Girth-Chromatic Number Theorem. In *Interactive Theorem Proving. ITP 2012.*, volume 7406 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012.
- [5] L. Noschinski. A Graph Library for Isabelle. *Mathematics in Computer Science*, 9(1):23–39, Mar. 2015. <http://link.springer.com/10.1007/s11786-014-0183-z>.