Undirected Graph Theory

Chelsea Edmonds

October 27, 2022

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 Szemeredi 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

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*))
  ⟨*proof*⟩

lemma *last-in-list-set*: *length xs* $\geq$ *1* $\Longrightarrow$ *last xs* $\in$ *set* (*xs*)
  ⟨*proof*⟩

lemma *last-in-list-tl-set*:
  assumes *length xs* $\geq$ *2*
  shows *last xs* $\in$ *set* (*tl xs*)
  ⟨*proof*⟩

lemma *length-list-decomp-lt*: *ys* $\neq$ [] $\Longrightarrow$ *length* (*xs* @*zs*) < *length* (*xs*@*ys*@*zs*)
  ⟨*proof*⟩

lemma *obtains-Max*:
  assumes *finite A* **and** *A* $\neq$ {}
  obtains *x* **where** *x* $\in$ *A* **and** *Max A* = *x*
  ⟨*proof*⟩

lemma *obtains-MAX*:
  assumes *finite A* **and** *A* $\neq$ {}
  obtains *x* **where** *x* $\in$ *A* **and** *Max* (*f* ' *A*) = *f x*
  ⟨*proof*⟩

lemma *obtains-Min*:
  assumes *finite A* **and** *A* $\neq$ {}
  obtains *x* **where** *x* $\in$ *A* **and** *Min A* = *x*
  ⟨*proof*⟩

lemma *obtains-MIN*:
  assumes *finite A* **and** *A* $\neq$ {}
  obtains *x* **where** *x* $\in$ *A* **and** *Min* (*f* ' *A*) = *f x*
  ⟨*proof*⟩

1.2 Initial Set up

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

fun *mk-triangle-set* :: ('*a* × '*a* × '*a*) ⇒ '*a set*
  where *mk-triangle-set* (*x*, *y*, *z*) = {*x*,*y*,*z*}

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

type-synonym '*a pregraph* = ('*a set*) × ('*a edge set*)

abbreviation *gverts* :: '*a pregraph* ⇒ '*a set* **where**

gverts H ≡ fst H

abbreviation *gedges* :: *'a pregraph ⇒ 'a edge set* **where**
gedges H ≡ snd H

fun *mk-edge* :: *'a × 'a ⇒ '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 ≡ {e . e ⊆ S ∧ 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 ⊆ ws ⟹ all-edges vs ⊆ all-edges ws
⟨*proof*⟩

lemma *all-edges-alt*: *all-edges S = {{x, y} | x y . x ∈ S ∧ y ∈ S ∧ x ≠ y}*
⟨*proof*⟩

lemma *all-edges-alt-pairs*: *all-edges S = mk-edge ` {uv ∈ S × S. fst uv ≠ snd uv}*
⟨*proof*⟩

lemma *all-edges-subset-Pow*: *all-edges A ⊆ Pow A*
⟨*proof*⟩

lemma *all-edges-disjoint*: *S ∩ T = {} ⟹ all-edges S ∩ all-edges T = {}*
⟨*proof*⟩

lemma *card-all-edges*: *finite A ⟹ card (all-edges A) = card A choose 2*
⟨*proof*⟩

lemma *finite-all-edges*: *finite S ⟹ finite (all-edges S)*
⟨*proof*⟩

lemma *in-mk-edge-img*: *(a,b) ∈ A ∨ (b,a) ∈ A ⟹ {a,b} ∈ mk-edge ` A*
⟨*proof*⟩

thm *in-mk-edge-img*
lemma *in-mk-uedge-img-iff*: *{a,b} ∈ mk-edge ` A ⟷ (a,b) ∈ A ∨ (b,a) ∈ A*
⟨*proof*⟩

lemma *inj-on-mk-edge*: *X ∩ Y = {} ⟹ inj-on mk-edge (X × Y)*
⟨*proof*⟩

definition *complete-graph* :: *'a set ⇒ 'a pregraph* **where**
complete-graph S ≡ (S, all-edges S)

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

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

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

lemma *all-edges-loops-ss*: $all\text{-}edges\ S \subseteq all\text{-}edges\text{-}loops\ S\ \{\{v\} \mid v.\ v \in S\} \subseteq all\text{-}edges\text{-}loops\ S$
⟨*proof*⟩

lemma *finite-singletons*: $finite\ S \Longrightarrow finite\ (\{\{v\} \mid v.\ v \in S\})$
⟨*proof*⟩

lemma *card-singletons*:
assumes $finite\ S$ **shows** $card\ \{\{v\} \mid v.\ v \in S\} = card\ S$
⟨*proof*⟩

lemma *finite-all-edges-loops*: $finite\ S \Longrightarrow finite\ (all\text{-}edges\text{-}loops\ S)$
⟨*proof*⟩

lemma *card-all-edges-loops*:
assumes $finite\ S$
shows $card\ (all\text{-}edges\text{-}loops\ S) = (card\ S)\ choose\ 2 + card\ S$
⟨*proof*⟩

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*: $e \in E \Longrightarrow e \subseteq V$
begin

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

abbreviation *graph-size* :: *nat* **where**
$graph\text{-}size \equiv card\ E$

definition *incident* :: $'a \Rightarrow 'a\ edge \Rightarrow bool$ **where**

incident v e $\equiv$ *v* $\in$ *e*

lemma *incident-edge-in-wf*: *e* $\in$ *E* $\Longrightarrow$ *incident v e* $\Longrightarrow$ *v* $\in$ *V*
  ⟨*proof*⟩

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

lemma *incident-edges-empty*: $\neg$ (*v* $\in$ *V*) $\Longrightarrow$ *incident-edges v* = \{\}
  ⟨*proof*⟩

lemma *finite-incident-edges*: *finite E* $\Longrightarrow$ *finite* (*incident-edges v*)
  ⟨*proof*⟩

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*: *edge-adj e1 e2* $\Longrightarrow$ *e1* $\in$ *E* $\wedge$ *e2* $\in$ *E*
  ⟨*proof*⟩

lemma *edge-adjacent-alt-def*: *e1* $\in$ *E* $\Longrightarrow$ *e2* $\in$ *E* $\Longrightarrow$ $\exists$ *x* . *x* $\in$ *V* $\wedge$ *x* $\in$ *e1* $\wedge$ *x* $\in$ *e2* $\Longrightarrow$ *edge-adj e1 e2*
  ⟨*proof*⟩

lemma *wellformed-alt-fst*: \{*x*, *y*\} $\in$ *E* $\Longrightarrow$ *x* $\in$ *V*
  ⟨*proof*⟩

lemma *wellformed-alt-snd*: \{*x*, *y*\} $\in$ *E* $\Longrightarrow$ *y* $\in$ *V*
  ⟨*proof*⟩
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*
  ⟨*proof*⟩

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*: $e \in E \Longrightarrow card\ e > 0 \wedge card\ e \leq 2$

begin

lemma *alt-edge-size*: $e \in E \Longrightarrow card\ e = 1 \vee card\ e = 2$
  ⟨*proof*⟩

definition *is-loop*:: *'a edge* ⇒ *bool* **where**
is-loop e ≡ *card e* = *1*

definition *is-sedge* :: *'a edge* ⇒ *bool* **where**
is-sedge e ≡ *card e* = *2*

lemma *is-edge-or-loop*: $e \in E \Longrightarrow$ *is-loop e* ∨ *is-sedge e*
  ⟨*proof*⟩

lemma *edges-split-loop*: $E = \{e \in E\ .\ is\text{-}loop\ e\ \} \cup \{e \in E\ .\ is\text{-}sedge\ e\}$
  ⟨*proof*⟩

lemma *edges-split-loop-inter-empty*: $\{\} = \{e \in E\ .\ is\text{-}loop\ e\ \} \cap \{e \in E\ .\ is\text{-}sedge\ e\}$
  ⟨*proof*⟩

definition *vert-adj* :: *'a* ⇒ *'a* ⇒ *bool* **where** — Neighbor in graph from Roth [1]
vert-adj v1 v2 ≡ {*v1*, *v2*} ∈ *E*

lemma *vert-adj-sym*: *vert-adj v1 v2* ⟷ *vert-adj v2 v1*
  ⟨*proof*⟩

lemma *vert-adj-imp-inV*: *vert-adj v1 v2* ⟹ *v1* ∈ *V* ∧ *v2* ∈ *V*
  ⟨*proof*⟩

lemma *vert-adj-inc-edge-iff*: *vert-adj v1 v2* ⟷ *incident v1* {*v1*, *v2*} ∧ *incident v2* {*v1*, *v2*} ∧ {*v1*, *v2*} ∈ *E*
  ⟨*proof*⟩

lemma *not-vert-adj*[*simp*]: ¬ *vert-adj v u* ⟹ {*v*, *u*} ∉ *E*
  ⟨*proof*⟩

definition *neighborhood* :: *'a* ⇒ *'a set* **where** — Neighbors in Roth Development [1]
neighborhood x ≡ {*v* ∈ *V* . *vert-adj x v*}

lemma *neighborhood-incident*: *u* ∈ *neighborhood v* ⟷ {*u*, *v*} ∈ *incident-edges v*
  ⟨*proof*⟩

definition *neighbors-ss* :: *'a* ⇒ *'a set* ⇒ *'a set* **where**

neighbors-ss x Y ≡ {*y* ∈ *Y* . *vert-adj x y*}

lemma *vert-adj-edge-iff2*:
assumes *v1* ≠ *v2*
shows *vert-adj v1 v2* ⟷ (∃ *e* ∈ *E* . *incident v1 e* ∧ *incident v2 e*)
⟨*proof*⟩

Incident simple edges, i.e. excluding loops

definition *incident-sedges* :: *'a* ⇒ *'a edge set* **where**
incident-sedges v ≡ {*e* ∈ *E* . *incident v e* ∧ *card e* = *2*}

lemma *finite-inc-sedges*: *finite E* ⟹ *finite* (*incident-sedges v*)
⟨*proof*⟩

lemma *incident-sedges-empty*[*simp*]: *v* ∉ *V* ⟹ *incident-sedges v* = {}
⟨*proof*⟩

definition *has-loop* :: *'a* ⇒ *bool* **where**
has-loop v ≡ {*v*} ∈ *E*

lemma *has-loop-in-verts*: *has-loop v* ⟹ *v* ∈ *V*
⟨*proof*⟩

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

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*}
⟨*proof*⟩

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

lemma *incident-loops-simp*: *has-loop v* ⟹ *incident-loops v* = {{*v*}} ¬ *has-loop v* ⟹ *incident-loops v* = {}
⟨*proof*⟩

lemma *incident-loops-union*: ⋃ (*incident-loops* ‘ *V*) = {*e* ∈ *E* . *is-loop e*}
⟨*proof*⟩

lemma *finite-incident-loops*: *finite* (*incident-loops v*)
⟨*proof*⟩

lemma *incident-loops-card*: *card* (*incident-loops v*) ≤ *1*
⟨*proof*⟩

lemma *incident-edges-union*: *incident-edges v* = *incident-sedges v* ∪ *incident-loops*

v
⟨proof⟩

lemma *incident-edges-sedges*[*simp*]: $\neg$ *has-loop* $v \Longrightarrow$ *incident-edges* v = *incident-sedges* v
⟨proof⟩

lemma *incident-sedges-union*: $\bigcup$ (*incident-sedges* ‘ V) = $\{e \in E$. *is-sedge* $e\}$
⟨proof⟩

lemma *empty-not-edge*: $\{\} \notin E$
⟨proof⟩

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

lemma *degree-no-loops*[*simp*]: $\neg$ *has-loop* $v \Longrightarrow$ *degree* v = *card* (*incident-edges* v)
⟨proof⟩

lemma *degree-none*[*simp*]: $\neg$ $v \in V \Longrightarrow$ *degree* v = *0*
⟨proof⟩

lemma *degree0-inc-edges-empt-iff*:
assumes *finite* E
shows *degree* v = *0* $\longleftrightarrow$ *incident-edges* $v = \{\}$
⟨proof⟩

lemma *incident-edges-neighbors-img*: *incident-edges* v = (λ u . $\{v, u\}$) ‘ (*neighborhood* v)
⟨proof⟩

lemma *card-incident-sedges-neighborhood*: *card* (*incident-edges* v) = *card* (*neighborhood* v)
⟨proof⟩

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

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))
⟨proof⟩

lemma *is-isolated-vertex-no-loop*: *is-isolated-vertex v* $\Longrightarrow \neg$ *has-loop v*
⟨*proof*⟩

lemma *is-isolated-vertex-degree0*: *is-isolated-vertex v* $\Longrightarrow$ *degree v* = *0*
⟨*proof*⟩

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

definition *max-degree* :: *nat* **where**
max-degree ≡ *Max* {*degree v* | *v*. *v* ∈ *V*}

definition *min-degree* :: *nat* **where**
min-degree ≡ *Min* {*degree v* | *v* . *v* ∈ *V*}

definition *is-edge-between* :: *'a set* ⇒ *'a set* ⇒ *'a edge* ⇒ *bool* **where**
is-edge-between X Y e ≡ ∃ *x y*. *e* = {*x*, *y*} ∧ *x* ∈ *X* ∧ *y* ∈ *Y*

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

definition *all-edges-between* :: *'a set* ⇒ *'a set* ⇒ (*'a* × *'a*) *set* **where**
all-edges-between X Y ≡ {(*x*, *y*) . *x* ∈ *X* ∧ *y* ∈ *Y* ∧ {*x*, *y*} ∈ *E*}

lemma *all-edges-betw-D3*: (*x*, *y*) ∈ *all-edges-between X Y* $\Longrightarrow$ {*x*, *y*} ∈ *E*
⟨*proof*⟩

lemma *all-edges-betw-I*: *x* ∈ *X* $\Longrightarrow$ *y* ∈ *Y* $\Longrightarrow$ {*x*, *y*} ∈ *E* $\Longrightarrow$ (*x*, *y*) ∈ *all-edges-between X Y*
⟨*proof*⟩

lemma *all-edges-between-subset*: *all-edges-between X Y* ⊆ *X*×*Y*
⟨*proof*⟩

lemma *all-edges-between-E-ss*: *mk-edge* ‘ *all-edges-between X Y* ⊆ *E*
⟨*proof*⟩

lemma *all-edges-between-rem-wf*: *all-edges-between X Y* = *all-edges-between* (*X* ∩ *V*) (*Y* ∩ *V*)
⟨*proof*⟩

lemma *all-edges-between-empty* [*simp*]:
all-edges-between {} *Z* = {} *all-edges-between Z* {} = {}
⟨*proof*⟩

lemma *all-edges-between-disjnt1*: *disjnt X Y* $\Longrightarrow$ *disjnt* (*all-edges-between X Z*) (*all-edges-between Y Z*)
⟨*proof*⟩

lemma *all-edges-between-disjnt2*: *disjnt* Y Z $\Longrightarrow$ *disjnt* (*all-edges-between* X Y) (*all-edges-between* X Z)
⟨*proof*⟩

lemma *max-all-edges-between*:
assumes *finite* X *finite* Y
shows *card* (*all-edges-between* X Y) $\leq$ *card* X $*$ *card* Y
⟨*proof*⟩

lemma *all-edges-between-Un1*:
all-edges-between ($X \cup Y$) Z = *all-edges-between* X Z $\cup$ *all-edges-between* Y Z
⟨*proof*⟩

lemma *all-edges-between-Un2*:
all-edges-between X ($Y \cup Z$) = *all-edges-between* X Y $\cup$ *all-edges-between* X Z
⟨*proof*⟩

lemma *finite-all-edges-between*:
assumes *finite* X *finite* Y
shows *finite* (*all-edges-between* X Y)
⟨*proof*⟩

lemma *all-edges-between-Union1*:
all-edges-between (*Union* $\mathcal{X}$) Y = ($\bigcup X \in \mathcal{X}$. *all-edges-between* X Y)
⟨*proof*⟩

lemma *all-edges-between-Union2*:
all-edges-between X (*Union* $\mathcal{Y}$) = ($\bigcup Y \in \mathcal{Y}$. *all-edges-between* X Y)
⟨*proof*⟩

lemma *all-edges-between-disjoint1*:
assumes *disjoint* R
shows *disjoint* ((λX. *all-edges-between* X Y) ‘ R)
⟨*proof*⟩

lemma *all-edges-between-disjoint2*:
assumes *disjoint* R
shows *disjoint* ((λY. *all-edges-between* X Y) ‘ R)
⟨*proof*⟩

lemma *all-edges-between-disjoint-family-on1*:
assumes *disjoint* R
shows *disjoint-family-on* (λX. *all-edges-between* X Y) R
⟨*proof*⟩

lemma *all-edges-between-disjoint-family-on2*:
assumes *disjoint* R
shows *disjoint-family-on* (λY. *all-edges-between* X Y) R

⟨*proof*⟩

lemma *all-edges-between-mono1*:
$Y \subseteq Z \Longrightarrow$ *all-edges-between* Y $X \subseteq$ *all-edges-between* Z X
⟨*proof*⟩

lemma *all-edges-between-mono2*:
$Y \subseteq Z \Longrightarrow$ *all-edges-between* X $Y \subseteq$ *all-edges-between* X Z
⟨*proof*⟩

lemma *inj-on-mk-edge*: $X \cap Y = \{\} \Longrightarrow$ *inj-on mk-edge* (*all-edges-between* X Y)
⟨*proof*⟩

lemma *all-edges-between-subset-times*: *all-edges-between* X $Y \subseteq (X \cap \bigcup E) \times (Y \cap \bigcup E)$
⟨*proof*⟩

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

lemma *all-edges-betw-sigma-neighbor*:
all-edges-between X Y = (*SIGMA* x:X. *neighbors-ss* x Y)
⟨*proof*⟩

lemma *card-all-edges-betw-neighbor*:
assumes *finite* X *finite* Y
shows *card* (*all-edges-between* X Y) = ($\sum x \in X$. *card* (*neighbors-ss* x Y))
⟨*proof*⟩

lemma *all-edges-between-swap*:
all-edges-between X Y = ($\lambda(x,y)$. (y,x)) ' (*all-edges-between* Y X)
⟨*proof*⟩

lemma *card-all-edges-between-commute*:
card (*all-edges-between* X Y) = *card* (*all-edges-between* Y X)
⟨*proof*⟩

lemma *all-edges-between-set*: *mk-edge* ' *all-edges-between* X $Y = \{\{x, y\} | \ x\ y.\ x \in X \wedge y \in Y \wedge \{x, y\} \in E\}$
⟨*proof*⟩

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 Szemeredi development, generalised here [2]

definition *edge-density* X $Y \equiv$ *card* (*all-edges-between* X Y)/(*card* X $*$ *card* Y)
lemma *edge-density-ge0*: *edge-density* X $Y \geq 0$
⟨*proof*⟩

lemma *edge-density-le1*: *edge-density* X $Y \leq 1$
⟨*proof*⟩

lemma *edge-density-zero*: $Y = \{\} \Longrightarrow$ *edge-density* X $Y = 0$
⟨*proof*⟩

lemma *edge-density-commute*: *edge-density* X Y = *edge-density* Y X
⟨*proof*⟩

lemma *edge-density-Un*:
assumes *disjnt X1 X2 finite X1 finite X2 finite Y*
shows *edge-density* $(X1 \cup X2)$ Y = (*edge-density* $X1$ Y $*$ *card* $X1$ + *edge-density* $X2$ Y $*$ *card* $X2$) / (*card* $X1$ + *card* $X2$)
⟨*proof*⟩

lemma *edge-density-eq0*:
assumes *all-edges-between* A $B = \{\}$ **and** $X \subseteq A$ $Y \subseteq B$
shows *edge-density* X $Y = 0$
⟨*proof*⟩

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*: *card* $\{e \in E$. *is-loop* $e\}$ = *card* $\{v \in V$. *has-loop* $v\}$
⟨*proof*⟩

lemma *finite-all-edges-between′*: *finite* (*all-edges-between* X Y)
⟨*proof*⟩

lemma *card-all-edges-between*:
assumes *finite* Y
shows *card* (*all-edges-between* X Y) = $(\sum y \in Y.$ *card* (*all-edges-between* X $\{y\}$))
⟨*proof*⟩

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*: $e \in E \Longrightarrow$ *card* $e = 2$
begin

lemma *wellformed-all-edges*: $E \subseteq$ *all-edges* V

⟨*proof*⟩

lemma *e-in-all-edges*: $e \in E \Longrightarrow e \in$ *all-edges* V
⟨*proof*⟩

lemma *e-in-all-edges-ss*: $e \in E \Longrightarrow e \subseteq V' \Longrightarrow V' \subseteq V \Longrightarrow e \in$ *all-edges* V'
⟨*proof*⟩

lemma *singleton-not-edge*: $\{x\} \notin E$ — Suggested by Mantas Baksys
⟨*proof*⟩

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
⟨*proof*⟩

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

lemma *fin-neighbourhood*: *finite* (*neighborhood* x)
⟨*proof*⟩

lemma *fin-all-edges*: *finite* (*all-edges* V)
⟨*proof*⟩

lemma *max-edges-graph*: *card* $E \leq$ (*card* V)^2
⟨*proof*⟩

end

sublocale *fin-sgraph* $\subseteq$ *fin-ulgraph*
⟨*proof*⟩

context *sgraph*
begin

lemma *no-loops*: $v \in V \Longrightarrow \neg$ *has-loop* v
⟨*proof*⟩

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*]: *degree* v = *card* (*incident-edges* v)
⟨*proof*⟩

lemma *alt-deg-neighborhood*: *degree* v = *card* (*neighborhood* v)

⟨*proof*⟩

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

definition *is-complete-n-graph*:: *nat* ⇒ *bool* **where**
is-complete-n-graph n ≡ *order* = *n* ∧ *E* = *all-edges V*

The complement of a graph is a basic concept

definition *is-complement* :: $'a$ *pregraph* ⇒ *bool* **where**
is-complement G ≡ *V* = *gverts G* ∧ *gedges G* = *all-edges V* − *E*

definition *complement-edges* :: $'a$ *edge set* **where**
complement-edges ≡ *all-edges V* − *E*

lemma *is-complement-edges*: *is-complement* (V', E') ⟷ $V = V'$ ∧ *complement-edges* = E'
⟨*proof*⟩

interpretation *G-comp*: *sgraph V complement-edges*
⟨*proof*⟩

lemma *is-complement-edge-iff*: $e \subseteq V \Longrightarrow e \in$ *complement-edges* ⟷ $e \notin E$ ∧ *card e* = *2*
⟨*proof*⟩

end

A complete graph is a simple graph

lemma *complete-sgraph*: *sgraph S* (*all-edges S*)
⟨*proof*⟩

interpretation *comp-sgraph*: *sgraph S* (*all-edges S*)
⟨*proof*⟩

lemma *complete-fin-sgraph*: *finite S* ⟹ *fin-sgraph S* (*all-edges S*)
⟨*proof*⟩

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*]: $V' \subseteq V \Longrightarrow E' \subseteq E \Longrightarrow$ *graph-system* V' E' $\Longrightarrow$ *graph-system* V E $\Longrightarrow$ *subgraph* V' E' V E
⟨*proof*⟩

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
⟨*proof*⟩

lemma *is-simp-subgraph*:
assumes *sgraph* V_G E_G
shows *sgraph* V_H E_H
⟨*proof*⟩

lemma *is-finite-subgraph*:
assumes *fin-graph-system* V_G E_G
shows *fin-graph-system* V_H E_H
⟨*proof*⟩

lemma (**in** *graph-system*) *subgraph-refl*: *subgraph* V E V E
⟨*proof*⟩

lemma *subgraph-trans*:
assumes *graph-system* V E
assumes *graph-system* V' E'
assumes *graph-system* V'' E''
shows *subgraph* V'' E'' V' E' $\Longrightarrow$ *subgraph* V' E' V E $\Longrightarrow$ *subgraph* V'' E'' V E
⟨*proof*⟩

lemma *subgraph-antisym*: *subgraph* V' E' V E $\Longrightarrow$ *subgraph* V E V' E' $\Longrightarrow V = V' \wedge E = E'$
⟨*proof*⟩

end

lemma (**in** *sgraph*) *subgraph-complete*: *subgraph* V E V (*all-edges* V)
⟨*proof*⟩

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* . *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'
⟨*proof*⟩

lemma (**in** *sgraph*) *induced-edges-self*: *induced-edges* $V = E$
⟨*proof*⟩

context *graph-system*
begin

lemma *induced-edges-ss*: $V' \subseteq V \Longrightarrow$ *induced-edges* $V' \subseteq E$
⟨*proof*⟩

lemma *induced-is-graph-sys*: *graph-system* V' (*induced-edges* V')
⟨*proof*⟩

interpretation *induced-graph*: *graph-system* V' (*induced-edges* V')
⟨*proof*⟩

lemma *induced-is-subgraph*: $V' \subseteq V \Longrightarrow$ *subgraph* V' (*induced-edges* V') *V E*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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 [**?**], however focuses on an undirected graph context compared to a directed graph context, and extends on some definitions, as required to formalise Balog Szemeredi 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* $(xs\ @\ [y])\ @\ [\{y, x\}]$
 ⟨*proof*⟩

lemma *walk-edges-tl-ss*: *set* (*walk-edges* (*tl xs*)) $\subseteq$ *set* (*walk-edges xs*)
 ⟨*proof*⟩

lemma *walk-edges-rev*: *rev* (*walk-edges xs*) = *walk-edges* (*rev xs*)
⟨*proof*⟩

lemma *walk-edges-append-ss1*: *set* (*walk-edges* (*ys*)) $\subseteq$ *set* (*walk-edges* (*xs@ys*))
⟨*proof*⟩

lemma *walk-edges-append-ss2*: *set* (*walk-edges* (*xs*)) $\subseteq$ *set* (*walk-edges* (*xs@ys*))
 ⟨*proof*⟩

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

⟨*proof*⟩

lemma *walk-edges-append-union*: $xs \neq [] \Longrightarrow ys \neq [] \Longrightarrow$
$set\ (walk\text{-}edges\ (xs@ys)) = set\ (walk\text{-}edges\ (xs)) \cup set\ (walk\text{-}edges\ ys) \cup \{\{last\ xs,\ hd\ ys\}\}$
⟨*proof*⟩

lemma *walk-edges-decomp-ss*: $set\ (walk\text{-}edges\ (xs@[y]@zs)) \subseteq set\ (walk\text{-}edges\ (xs@[y]@ys@[y]@zs))$
⟨*proof*⟩

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

lemma *walk-length-conv*: $walk\text{-}length\ p = length\ p - 1$
⟨*proof*⟩

lemma *walk-length-rev*: $walk\text{-}length\ p = walk\text{-}length\ (rev\ p)$
⟨*proof*⟩

lemma *walk-length-app*: $xs \neq [] \Longrightarrow ys \neq [] \Longrightarrow walk\text{-}length\ (xs\ @\ ys) = walk\text{-}length\ xs + walk\text{-}length\ ys + 1$
⟨*proof*⟩

lemma *walk-length-app-ineq*: $walk\text{-}length\ (xs\ @\ ys) \geq walk\text{-}length\ xs + walk\text{-}length\ ys\ \wedge$
$walk\text{-}length\ (xs\ @\ ys) \leq walk\text{-}length\ xs + walk\text{-}length\ ys + 1$
⟨*proof*⟩

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

definition *is-walk* :: $'a\ list \Rightarrow bool$ **where**
$is\text{-}walk\ xs \equiv set\ xs \subseteq V \wedge set\ (walk\text{-}edges\ xs) \subseteq E \wedge xs \neq []$

lemma *is-walkI*: $set\ xs \subseteq V \Longrightarrow set\ (walk\text{-}edges\ xs) \subseteq E \Longrightarrow xs \neq [] \Longrightarrow is\text{-}walk\ xs$
⟨*proof*⟩

lemma *is-walk-wf*: $is\text{-}walk\ xs \Longrightarrow set\ xs \subseteq V$
⟨*proof*⟩

lemma *is-walk-wf-hd*: $is\text{-}walk\ xs \Longrightarrow hd\ xs \in V$
⟨*proof*⟩

lemma *is-walk-wf-last*: $is\text{-}walk\ xs \Longrightarrow last\ xs \in V$
⟨*proof*⟩

lemma *is-walk-singleton*: $u \in V \Longrightarrow is\text{-}walk\ [u]$
⟨*proof*⟩

lemma *is-walk-not-empty*: $is\text{-}walk\ xs \Longrightarrow xs \neq []$

⟨*proof*⟩

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

Reasoning on transformations of a walk

lemma *is-walk-rev*: *is-walk xs* $\longleftrightarrow$ *is-walk* (*rev xs*)
⟨*proof*⟩

lemma *is-walk-tl*: *length xs* $\geq$ *2* $\Longrightarrow$ *is-walk xs* $\Longrightarrow$ *is-walk* (*tl xs*)
⟨*proof*⟩

lemma *is-walk-append*:
assumes *is-walk xs*
assumes *is-walk ys*
assumes *last xs* = *hd ys*
shows *is-walk* (*xs* @ (*tl ys*))
⟨*proof*⟩

lemma *is-walk-decomp*:
assumes *is-walk* (*xs*@[*y*]@*ys*@[*y*]@*zs*) (**is** *is-walk ?w*)
shows *is-walk* (*xs*@[*y*]@*zs*)
⟨*proof*⟩

lemma *is-walk-hd-tl*:
assumes *is-walk* (*y* # *ys*)
assumes {*x*, *y*} $\in$ *E*
shows *is-walk* (*x* # *y* # *ys*)
⟨*proof*⟩

lemma *is-walk-drop-hd*:
assumes *ys* $\neq$ []
assumes *is-walk* (*y* # *ys*)
shows *is-walk ys*
⟨*proof*⟩

lemma *walk-edges-index*:
assumes *i* $\geq$ *0 i* < *walk-length w*
assumes *is-walk w*
shows (*walk-edges w*) ! *i* $\in$ *E*
⟨*proof*⟩

lemma *is-walk-index*:
assumes *i* $\geq$ *0 Suc i* < (*length w*)
assumes *is-walk w*
shows {*w* ! *i*, *w* ! (*i* + *1*)} $\in$ *E*
⟨*proof*⟩

lemma *is-walk-take*:

assumes *is-walk w*
assumes $n > 0$
assumes $n \leq$ *length w*
shows *is-walk* (*take n w*)
⟨*proof*⟩

lemma *is-walk-drop*:
assumes *is-walk w*
assumes $n <$ *length w*
shows *is-walk* (*drop n w*)
⟨*proof*⟩

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*)
⟨*proof*⟩

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*)
⟨*proof*⟩

definition *is-trail* :: *'a list* ⇒ *bool* **where**
is-trail xs ≡ *is-walk xs* ∧ *distinct* (*walk-edges xs*)

lemma *is-trail-rev*: *is-trail xs* ⟷ *is-trail* (*rev xs*)
⟨*proof*⟩

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* ⇒ *bool* **where**
is-path xs ≡ (*is-open-walk xs* ∧ *distinct* (*xs*))

lemma *is-path-rev*: *is-path xs* ⟷ *is-path* (*rev xs*)
⟨*proof*⟩

lemma *is-path-walk*: *is-path xs* ⟹ *is-walk xs*
⟨*proof*⟩

definition *paths* :: *'a list set* **where**
paths ≡ {*p* . *is-path p*}

lemma *paths-ss-walk*: $paths \subseteq walks$
⟨*proof*⟩

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**
$is\text{-}gen\text{-}path\ p \equiv is\text{-}walk\ p \wedge ((distinct\ (tl\ p) \wedge hd\ p = last\ p) \vee distinct\ p)$

lemma *is-path-gen-path*: $is\text{-}path\ p \Longrightarrow is\text{-}gen\text{-}path\ p$
⟨*proof*⟩

lemma *is-gen-path-rev*: $is\text{-}gen\text{-}path\ p \longleftrightarrow is\text{-}gen\text{-}path\ (rev\ p)$
⟨*proof*⟩

lemma *is-gen-path-distinct*: $is\text{-}gen\text{-}path\ p \Longrightarrow hd\ p \neq last\ p \Longrightarrow distinct\ p$
⟨*proof*⟩

lemma *is-gen-path-distinct-tl*:
assumes $is\text{-}gen\text{-}path\ p$ **and** $hd\ p = last\ p$
shows $distinct\ (tl\ p)$
⟨*proof*⟩

lemma *is-gen-path-trivial*: $x \in V \Longrightarrow is\text{-}gen\text{-}path\ [x]$
⟨*proof*⟩

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

lemma *gen-paths-ss-walks*: $gen\text{-}paths \subseteq walks$
⟨*proof*⟩

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**
$is\text{-}cycle\ xs \equiv is\text{-}closed\text{-}walk\ xs \wedge walk\text{-}length\ xs \geq 1 \wedge distinct\ (tl\ xs)$

lemma *is-gen-path-cycle*: $is\text{-}cycle\ p \Longrightarrow is\text{-}gen\text{-}path\ p$
⟨*proof*⟩

lemma *is-cycle-alt-gen-path*: $is\text{-}cycle\ xs \longleftrightarrow is\text{-}gen\text{-}path\ xs \wedge walk\text{-}length\ xs \geq 1 \wedge hd\ xs = last\ xs$
⟨*proof*⟩

lemma *is-cycle-alt*: $is\text{-}cycle\ xs \longleftrightarrow is\text{-}walk\ xs \wedge distinct\ (tl\ xs) \wedge walk\text{-}length\ xs \geq 1 \wedge hd\ xs = last\ xs$
⟨*proof*⟩

lemma *is-cycle-rev*: *is-cycle xs* $\longleftrightarrow$ *is-cycle* (*rev xs*)
⟨*proof*⟩

lemma *cycle-tl-is-path*: *is-cycle xs* $\wedge$ *walk-length xs* $\geq$ *3* $\Longrightarrow$ *is-path* (*tl xs*)
⟨*proof*⟩

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*⟩

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*⟩

definition *cycles* :: *'a list set* **where**
cycles $\equiv$ {*p*. *is-cycle p*}

lemma *cycles-ss-gen-paths*: *cycles* $\subseteq$ *gen-paths*
⟨*proof*⟩

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

Walk edges are distinct in a path and cycle

lemma *distinct-edgesI*:
assumes *distinct p* **shows** *distinct* (*walk-edges p*)
⟨*proof*⟩

lemma *scycles-distinct-edges*:
assumes *c* $\in$ *cycles 3* $\leq$ *walk-length c* **shows** *distinct* (*walk-edges c*)
⟨*proof*⟩

end

context *fin-ulgraph*
begin

lemma *finite-paths*: *finite paths*
⟨*proof*⟩

lemma *finite-cycles*: *finite* (*cycles*)
⟨*proof*⟩

lemma *finite-gen-paths*: *finite* (*gen-paths*)
⟨*proof*⟩

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

lemma *connecting-walk-rev*: *connecting-walk u v xs* $\longleftrightarrow$ *connecting-walk v u* (*rev xs*)
⟨*proof*⟩

lemma *connecting-walk-wf*: *connecting-walk u v xs* $\Longrightarrow u \in V \wedge v \in V$
⟨*proof*⟩

lemma *connecting-walk-self*: $u \in V \Longrightarrow$ *connecting-walk u u* $[u]$ $=$ *True*
⟨*proof*⟩

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

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

lemma *connecting-path-rev*: *connecting-path u v xs* $\longleftrightarrow$ *connecting-path v u* (*rev xs*)
⟨*proof*⟩

lemma *connecting-path-walk*: *connecting-path u v xs* $\Longrightarrow$ *connecting-walk u v xs*
⟨*proof*⟩

lemma *connecting-path-str-gen*: *connecting-path-str u v xs* $\Longrightarrow$ *connecting-path u v xs*
⟨*proof*⟩

lemma *connecting-path-gen-str*: *connecting-path u v xs* $\Longrightarrow$ ($\neg$ *is-cycle xs*) $\Longrightarrow$ *walk-length xs* $>$ *0* $\Longrightarrow$ *connecting-path-str u v xs*
⟨*proof*⟩

lemma *connecting-path-alt-def*: *connecting-path u v xs* $\longleftrightarrow$ *connecting-walk u v xs* $\wedge$ *is-gen-path xs*
⟨*proof*⟩

lemma *connecting-path-length-bound*: $u \neq v \Longrightarrow$ *connecting-path u v p* $\Longrightarrow$ *walk-length* $p \geq 1$
⟨*proof*⟩

lemma *connecting-path-self*: $u \in V \Longrightarrow$ *connecting-path u u* $[u]$ = *True*
⟨*proof*⟩

lemma *connecting-path-singleton*: *connecting-path u v xs* $\Longrightarrow$ *length xs* = *1* $\Longrightarrow$ $u = v$
⟨*proof*⟩

lemma *connecting-walk-path*:
assumes *connecting-walk u v xs*
shows $\exists$ *ys* . *connecting-path u v ys* $\wedge$ *walk-length ys* $\leq$ *walk-length xs*
⟨*proof*⟩

lemma *connecting-walk-split*:
assumes *connecting-walk u v xs* **assumes** *connecting-walk v z ys*
shows *connecting-walk u z* (*xs* @ (*tl ys*))
⟨*proof*⟩

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* $\leq$ *walk-length* (*xs* @ (*tl ys*))
⟨*proof*⟩

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* $\leq$ *walk-length xs* + *walk-length ys*
⟨*proof*⟩

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* :: $'a \Rightarrow 'a \Rightarrow$ *bool* **where**
vert-connected u v $\equiv \exists$ *xs* . *connecting-path u v xs*

lemma *vert-connected-rev*: *vert-connected u v* $\longleftrightarrow$ *vert-connected v u*
⟨*proof*⟩

lemma *vert-connected-id*: $u \in V \Longrightarrow$ *vert-connected u u* = *True*
⟨*proof*⟩

lemma *vert-connected-trans*: *vert-connected u v* $\Longrightarrow$ *vert-connected v z* $\Longrightarrow$ *vert-connected u z*
⟨*proof*⟩

lemma *vert-connected-wf*: *vert-connected u v* $\Longrightarrow u \in V \wedge v \in V$
⟨*proof*⟩

definition *vert-connected-n* :: $'a \Rightarrow 'a \Rightarrow nat \Rightarrow bool$ **where**
vert-connected-n u v n $\equiv \exists$ *p. connecting-path u v p* $\wedge$ *walk-length p* = *n*

lemma *vert-connected-n-imp*: *vert-connected-n u v n* $\Longrightarrow$ *vert-connected u v*
⟨*proof*⟩

lemma *vert-connected-n-rev*: *vert-connected-n u v n* $\longleftrightarrow$ *vert-connected-n v u n*
⟨*proof*⟩

definition *connecting-paths* :: $'a \Rightarrow 'a \Rightarrow 'a$ *list set* **where**
connecting-paths u v $\equiv$ {*xs* . *connecting-path u v xs*}

lemma *connecting-paths-self*: $u \in V \Longrightarrow [u] \in$ *connecting-paths u u*
⟨*proof*⟩

lemma *connecting-paths-empty-iff*: *vert-connected u v* $\longleftrightarrow$ *connecting-paths u v* $\neq$ {}
⟨*proof*⟩

lemma *elem-connecting-paths*: $p \in$ *connecting-paths u v* $\Longrightarrow$ *connecting-path u v p*
⟨*proof*⟩

lemma *connecting-paths-ss-gen*: *connecting-paths u v* $\subseteq$ *gen-paths*
⟨*proof*⟩

lemma *connecting-paths-sym*: *xs* $\in$ *connecting-paths u v* $\longleftrightarrow$ *rev xs* $\in$ *connecting-paths v u*
⟨*proof*⟩

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 $V' \equiv (\forall\ u\ v\ .\ u \in V' \longrightarrow v \in V' \longrightarrow$ *vert-connected u v*)

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

$\langle proof \rangle$

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

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

lemma *is-connected-setD*: *is-connected-set* $V' \Longrightarrow u \in V' \Longrightarrow v \in V' \Longrightarrow$ *vert-connected* $u\ v$
$\langle proof \rangle$

lemma *not-connected-set*: $\neg$ *is-connected-set* $V' \Longrightarrow u \in V' \Longrightarrow \exists\ v \in V'$. $\neg$ *vert-connected* $u\ v$
$\langle proof \rangle$

3.3 Graph Properties on Connectivity

The shortest path is defined to be the infinum of the set of connecting path walk lengths. Drawing inspiration from [?], we use the infinum 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 $u\ v \equiv$ *INF* $p\in$ *connecting-paths* $u\ v$. *enat* (*walk-length* p)

lemma *shortest-path-walk-length*: *shortest-path* $u\ v = n \Longrightarrow p \in$ *connecting-paths* $u\ v \Longrightarrow$ *walk-length* $p \geq n$
$\langle proof \rangle$

lemma *shortest-path-lte*: $\bigwedge\ p\ .\ p \in$ *connecting-paths* $u\ v \Longrightarrow$ *shortest-path* $u\ v \leq$ *walk-length* p
$\langle proof \rangle$

lemma *shortest-path-obtains*:
assumes *shortest-path* $u\ v = n$
assumes $n \neq$ *top*
obtains p **where** $p \in$ *connecting-paths* $u\ v$ **and** *walk-length* $p = n$
$\langle proof \rangle$

lemma *shortest-path-intro*:
assumes $n \neq$ *top*
assumes ($\exists\ p \in$ *connecting-paths* $u\ v$. *walk-length* $p = n$)
assumes ($\bigwedge\ p.\ p \in$ *connecting-paths* $u\ v \Longrightarrow n \leq$ *walk-length* p)
shows *shortest-path* $u\ v = n$
$\langle proof \rangle$

lemma *shortest-path-self*:
assumes $u \in V$
shows *shortest-path* $u\ u = 0$

$\langle proof \rangle$

lemma *connecting-paths-sym-length*: $i \in$ *connecting-paths* $u\ v \Longrightarrow \exists j \in$*connecting-paths* $v\ u$. (*walk-length* j) = (*walk-length* i)
$\langle proof \rangle$

lemma *shortest-path-sym*: *shortest-path* $u\ v$ = *shortest-path* $v\ u$
$\langle proof \rangle$

lemma *shortest-path-inf*: $\neg$ *vert-connected* $u\ v \Longrightarrow$ *shortest-path* $u\ v = \infty$
$\langle proof \rangle$

lemma *shortest-path-not-inf*:
assumes *vert-connected* $u\ v$
shows *shortest-path* $u\ v \neq \infty$
$\langle proof \rangle$

lemma *shortest-path-obtains2*:
assumes *vert-connected* $u\ v$
obtains p **where** $p \in$ *connecting-paths* $u\ v$ **and** *walk-length* p = *shortest-path* $u\ v$
$\langle proof \rangle$

lemma *shortest-path-split*: *shortest-path* $x\ y \leq$ *shortest-path* $x\ z$ + *shortest-path* $z\ y$
$\langle proof \rangle$

lemma *shortest-path-invalid-v*: $v \notin V \vee u \notin V \Longrightarrow$ *shortest-path* $u\ v = \infty$
$\langle proof \rangle$

lemma *shortest-path-lb*:
assumes $u \neq v$
assumes *vert-connected* $u\ v$
shows *shortest-path* $u\ v > 0$
$\langle proof \rangle$

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

definition *eccentricity* :: $'a \Rightarrow$ *enat* **where**
eccentricity $v \equiv$ *SUP* $u \in V - \{v\}$. *shortest-path* $v\ u$

lemma *eccentricity-empty-vertices*: $V = \{\} \Longrightarrow$ *eccentricity* $v = 0$
$V = \{v\} \Longrightarrow$ *eccentricity* $v = 0$
$\langle proof \rangle$

lemma *eccentricity-bot-iff*: *eccentricity* $v = 0 \longleftrightarrow V = \{\} \vee V = \{v\}$
$\langle proof \rangle$

lemma *eccentricity-invalid-v*:

assumes $v \notin V$
assumes $V \neq \{\}$
shows *eccentricity* $v = \infty$
⟨*proof*⟩

lemma *eccentricity-gt-shortest-path*:
assumes $u \in V$
shows *eccentricity* $v \geq$ *shortest-path* v u
⟨*proof*⟩

lemma *eccentricity-disconnected-graph*:
assumes $\neg$ *is-connected-set* V
assumes $v \in V$
shows *eccentricity* $v = \infty$
⟨*proof*⟩

The diameter is the largest distance between any two vertices

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

lemma *diameter-gt-eccentricity*: $v \in V \Longrightarrow$ *diameter* $\geq$ *eccentricity* v
⟨*proof*⟩

lemma *diameter-disconnected-graph*:
assumes $\neg$ *is-connected-set* V
shows *diameter* $= \infty$
⟨*proof*⟩

lemma *diameter-empty*: $V = \{\} \Longrightarrow$ *diameter* $=$ *0*
⟨*proof*⟩

lemma *diameter-singleton*: $V = \{v\} \Longrightarrow$ *diameter* $=$ *eccentricity* v
⟨*proof*⟩

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*: $v \in V \Longrightarrow$ *radius* $\leq$ *eccentricity* v
⟨*proof*⟩

lemma *radius-disconnected-graph*: $\neg$ *is-connected-set* $V \Longrightarrow$ *radius* $= \infty$
⟨*proof*⟩

lemma *radius-empty*: $V = \{\} \Longrightarrow$ *radius* $= \infty$
⟨*proof*⟩

lemma *radius-singleton*: $V = \{v\} \Longrightarrow$ *radius* $=$ *eccentricity* v
⟨*proof*⟩

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

definition *centre* :: *'a set* **where**
centre ≡ {v ∈ V. *eccentricity* v = *radius* }

lemma *centre-disconnected-graph*: ¬ *is-connected-set* V ⟹ *centre* = V
⟨*proof*⟩

end

lemma (**in** *fin-ulgraph*) *fin-connecting-paths*: *finite* (*connecting-paths* u v)
⟨*proof*⟩

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 ∈ V ⟹ v ∈ V ⟹ *vert-connected* u v
⟨*proof*⟩

lemma *vertices-connected-path*: u ∈ V ⟹ v ∈ V ⟹ ∃ p. *connecting-path* u v p
⟨*proof*⟩

lemma *connecting-paths-not-empty*: u ∈ V ⟹ v ∈ V ⟹ *connecting-paths* u v ≠ {}
⟨*proof*⟩

lemma *min-shortest-path*: **assumes** u ∈ V v ∈ V u ≠ v
shows *shortest-path* u v > *0*
⟨*proof*⟩

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 ≠ {v}
shows *eccentricity* v = ∞ ∨ (∃ u ∈ (V − {v}) . *shortest-path* v u = *eccentricity* v)
⟨*proof*⟩

lemma *diameter-obtains*: *diameter* = ∞ ∨ (∃ v ∈ V . *eccentricity* v = *diameter*)
⟨*proof*⟩

lemma *radius-diameter-singleton-eq*: **assumes** *card* V = *1* **shows** *radius* = *diameter*
⟨*proof*⟩

end

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

In a finite context the supremum/infinum 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 *shortest-path u v* = *Min* ((λ *p. enat* (*walk-length p*)) ' (*connecting-paths u v*)) (**is** *shortest-path u v* = *Min ?A*)
⟨*proof*⟩

lemma *eccentricity-Max-alt*:
assumes $v \in V$
assumes $V \neq \{v\}$
shows *eccentricity v* = *Max* ((λ *u. shortest-path v u*) ' ($V - \{v\}$))
⟨*proof*⟩

lemma *diameter-Max-alt*: *diameter* = *Max* ((λ *v. eccentricity v*) ' *V*)
⟨*proof*⟩

lemma *radius-Min-alt*: *radius* = *Min* ((λ *v. eccentricity v*) ' *V*)
⟨*proof*⟩

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*⟩

lemma *radius-obtains*:
obtains *v* **where** $v \in V$ **and** *radius* = *eccentricity v*
⟨*proof*⟩

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*⟩

lemma *diameter-obtains*:
obtains *v* **where** $v \in V$ **and** *diameter* = *eccentricity v*
⟨*proof*⟩

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*⟩

lemma *radius-diameter-bounds*:
  shows *radius* $\leq$ *diameter diameter* $\leq$ *2* $*$ *radius*
⟨*proof*⟩

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* $\subseteq$ *connected-ulgraph*
  ⟨*proof*⟩

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

sublocale *fin-connected-sgraph* $\subseteq$ *fin-connected-ulgraph*
  ⟨*proof*⟩

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 $\equiv$ *INF p*$\in$ *cycles. enat* (*walk-length p*)

lemma *girth-acyclic*: *cycles* = {} $\Longrightarrow$ *girth* = ∞
  ⟨*proof*⟩

lemma *girth-lte*: *c* $\in$ *cycles* $\Longrightarrow$ *girth* $\leq$ *walk-length c*
  ⟨*proof*⟩

lemma *girth-obtains*:
  assumes *girth* $\neq$ *top*
  obtains *c* **where** *c* $\in$ *cycles* **and** *walk-length c* = *girth*
  ⟨*proof*⟩

lemma *girthI*:

assumes $c' \in$ *cycles*
assumes $\bigwedge$ *c* . $c \in$ *cycles* $\Longrightarrow$ *walk-length* $c' \leq$ *walk-length c*
shows *girth* = *walk-length* c'
$\langle proof \rangle$

lemma (**in** *fin-sgraph*) *girth-min-alt*:
assumes *cycles* $\neq \{\}$
shows *girth* = *Min* ((λ *c* . *enat* (*walk-length c*)) ' *cycles*) (**is** *girth* = *Min ?A*)
$\langle proof \rangle$

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

A More mathematical way of thinking about it

lemma *is-independent-alt*: *is-independent-set vs* $\longleftrightarrow$ *vs* $\subseteq$ *V* $\wedge$ ($\forall$ *v* $\in$ *vs*. $\forall$ *u* $\in$ *vs*. $\neg$ *vert-adj v u*)
$\langle proof \rangle$

lemma *singleton-independent-set*: $v \in V \Longrightarrow$ *is-independent-set* $\{v\}$
$\langle proof \rangle$

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* $\Longrightarrow$ *us* $\subseteq$ *vs* $\Longrightarrow$ *us* $\in$ *independent-sets*
$\langle proof \rangle$

lemma *le-independence-iff*:
assumes $0 < k$
shows $k \leq \alpha \longleftrightarrow k \in$ *card* ' *independent-sets* (**is** *?L* $\longleftrightarrow$ *?R*)
$\langle proof \rangle$

lemma *zero-less-independence*:
assumes $V \neq \{\}$
shows $0 < \alpha$
$\langle proof \rangle$

end

context *fin-sgraph*
begin
lemma *fin-independent-sets*: *finite* (*independent-sets*)
$\langle proof \rangle$

lemma *independence-le-card*:
  shows $\alpha \ \leq$ *card* V
$\langle proof \rangle$

lemma *independence-fin*: $\alpha \neq \infty$
  $\langle proof \rangle$

lemma *independence-max-alt*: $V \neq \{\} \Longrightarrow \alpha = Max\ ((\lambda\ vs\ .\ enat\ (card\ vs))$ ' *independent-sets*)
  $\langle proof \rangle$

lemma *independent-sets-ne*:
  assumes $V \neq \{\}$
  shows *independent-sets* $\neq \{\}$
$\langle proof \rangle$

lemma *independence-obtains*:
  assumes $V \neq \{\}$
  obtains *vs* **where** *is-independent-set vs* **and** *card vs* $= \alpha$
$\langle proof \rangle$
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*
    HOL−Combinatorics.Multiset-Permutations
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* :: $'a \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ **where**
triangle-in-graph x y z $\equiv (\{x,y\} \in E) \wedge (\{y,z\} \in E) \wedge (\{x,z\} \in E)$

lemma *triangle-in-graph-edge-empty*: $E = \{\} \Longrightarrow \neg$ *triangle-in-graph x y z*
  $\langle proof \rangle$

definition *triangle-triples* **where**
triangle-triples $X\ Y\ Z \equiv \{(x,y,z) \in X \times Y \times Z.$ *triangle-in-graph x y z* $\}$

definition
  unique-triangles
    $\equiv \forall\, e \in E.\ \exists! T.\ \exists\, x\ y\ z.\ T = \{x,y,z\} \wedge$ *triangle-in-graph* $x\ y\ z\ \wedge e \subseteq T$

definition *triangle-set* :: *'a set set*
  where *triangle-set* $\equiv \{\ \{x,y,z\} \mid x\ y\ z.$ *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*⟩

lemma *triangle-commu1*:
  assumes *triangle-in-graph* $x\ y\ z$
  shows *triangle-in-graph* $y\ x\ z$
  ⟨*proof*⟩

lemma *triangle-vertices-distinct1*:
  assumes *tri*: *triangle-in-graph* $x\ y\ z$
  shows $x \neq y$
⟨*proof*⟩

lemma *triangle-vertices-distinct2*:
  assumes *triangle-in-graph* $x\ y\ z$
  shows $y \neq z$
  ⟨*proof*⟩

lemma *triangle-vertices-distinct3*:
  assumes *triangle-in-graph* $x\ y\ z$
  shows $z \neq x$
  ⟨*proof*⟩

lemma *triangle-in-graph-edge-point*: *triangle-in-graph* $x\ y\ z \longleftrightarrow \{y,\ z\} \in E\ \wedge$ *vert-adj* $x\ y\ \wedge$ *vert-adj* $x\ z$
  ⟨*proof*⟩

lemma *edge-vertices-not-equal*:
  assumes $\{x,y\} \in E$
  shows $x \neq y$
  ⟨*proof*⟩

lemma *edge-btw-vertices-not-equal*:
  assumes $(x,\ y) \in$ *all-edges-between* $X\ Y$
  shows $x \neq y$
  ⟨*proof*⟩

lemma *mk-triangle-from-ss-edges*:
assumes $(x,\ y) \in$ *all-edges-between* $X\ Y$ **and** $(x,\ z) \in$ *all-edges-between* $X\ Z$ **and**

$(y, z) \in$ *all-edges-between Y Z*
shows (*triangle-in-graph x y z*)
⟨*proof*⟩

lemma *triangle-in-graph-verts*:
assumes *triangle-in-graph x y z*
shows $x \in V$ $y \in V$ $z \in V$
⟨*proof*⟩

lemma *convert-triangle-rep-ss*:
assumes $X \subseteq V$ **and** $Y \subseteq V$ **and** $Z \subseteq V$
shows *mk-triangle-set* ‘ $\{(x, y, z) \in X \times Y \times Z$. (*triangle-in-graph x y z*)$\} \subseteq$ *triangle-set*
⟨*proof*⟩

lemma (**in** *fin-sgraph*) *finite-triangle-set*: *finite* (*triangle-set*)
⟨*proof*⟩

lemma *card-triangle-3*:
assumes $t \in$ *triangle-set*
shows *card* $t = 3$
⟨*proof*⟩

lemma *triangle-set-power-set-ss*: *triangle-set* $\subseteq$ *Pow V*
⟨*proof*⟩

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*⟩

lemma *triangle-set-graph-edge-ss*:
assumes $E' \subseteq E$
shows (*sgraph.triangle-set* E') $\subseteq$ (*triangle-set*)
⟨*proof*⟩

lemma (**in** *fin-sgraph*) *triangle-set-graph-edge-ss-bound*:
assumes $E' \subseteq E$
shows *card* (*triangle-set*) $\geq$ *card* (*sgraph.triangle-set* E')
⟨*proof*⟩

end

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

lemma *triangle-free-graph-empty*: $E = \{\} \Longrightarrow$ *triangle-free-graph V E*
⟨*proof*⟩

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 $card\ (triangle\text{-}set) \geq 1/6 * card\ \{(x, y, z) \in X \times Y \times Z\ .\ (triangle\text{-}in\text{-}graph\ x\ y\ z)\}$
(**is** $- \geq 1/6 * card\ ?TT$)
$\langle proof \rangle$

lemma *card-convert-triangle-rep-bound*:
fixes $t :: real$
assumes $card\ \{(x, y, z) \in X \times Y \times Z\ .\ (triangle\text{-}in\text{-}graph\ x\ y\ z)\} \geq t$
assumes $X \subseteq V$ **and** $Y \subseteq V$ **and** $Z \subseteq V$
shows $card\ (triangle\text{-}set) \geq 1/6 *t$
$\langle proof \rangle$
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 $all\text{-}bi\text{-}edges :: 'a\ set \Rightarrow 'a\ set \Rightarrow 'a\ edge\ set$ **where**
$all\text{-}bi\text{-}edges\ X\ Y \equiv mk\text{-}edge\ `\ (X \times Y)$

lemma *all-bi-edges-alt*:
assumes $X \cap Y = \{\}$
shows $all\text{-}bi\text{-}edges\ X\ Y = \{e\ .\ card\ e = 2 \wedge e \cap X \neq \{\} \wedge e \cap Y \neq \{\}\}$
$\langle proof \rangle$

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

lemma *all-bi-edges-wf*: $e \in all\text{-}bi\text{-}edges\ X\ Y \Longrightarrow e \subseteq X \cup Y$
$\langle proof \rangle$

lemma *all-bi-edges-2*: $X \cap Y = \{\} \Longrightarrow e \in all\text{-}bi\text{-}edges\ X\ Y \Longrightarrow card\ e = 2$
$\langle proof \rangle$

lemma *all-bi-edges-main*: $X \cap Y = \{\} \Longrightarrow$ *all-bi-edges* $X\ Y \subseteq$ *all-edges* $(X \cup Y)$
⟨*proof*⟩

lemma *all-bi-edges-finite*: *finite* $X \Longrightarrow$ *finite* $Y \Longrightarrow$ *finite* (*all-bi-edges* $X\ Y$)
⟨*proof*⟩

lemma *all-bi-edges-not-ssX*: $X \cap Y = \{\} \Longrightarrow e \in$ *all-bi-edges* $X\ Y \Longrightarrow \neg\ e \subseteq X$
⟨*proof*⟩

lemma *all-bi-edges-sym*: *all-bi-edges* $X\ Y =$ *all-bi-edges* $Y\ X$
⟨*proof*⟩

lemma *all-bi-edges-not-ssY*: $X \cap Y = \{\} \Longrightarrow e \in$ *all-bi-edges* $X\ Y \Longrightarrow \neg\ e \subseteq Y$
⟨*proof*⟩

lemma *card-all-bi-edges*:
assumes *finite* X *finite* Y
assumes $X \cap Y = \{\}$
shows *card* (*all-bi-edges* $X\ Y$) $=$ *card* $X *$ *card* Y
⟨*proof*⟩

lemma (**in** *sgraph*) *all-edges-between-bi-subset*: *mk-edge* ‘ *all-edges-between* $X\ Y \subseteq$ *all-bi-edges* $X\ Y$
⟨*proof*⟩

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 set*
assumes *partition*: *partition-on* V $\{X, Y\}$
assumes *ne*: $X \neq Y$
assumes *edge-betw*: $e \in E \Longrightarrow e \in$ *all-bi-edges* $X\ Y$
begin

lemma *part-intersect-empty*: $X \cap Y = \{\}$
⟨*proof*⟩

lemma *X-not-empty*: $X \neq \{\}$
⟨*proof*⟩

lemma *Y-not-empty*: $Y \neq \{\}$
⟨*proof*⟩

lemma *XY-union*: $X \cup Y = V$
⟨*proof*⟩

lemma *card-edges-two*: $e \in E \Longrightarrow$ *card* $e = 2$

⟨*proof*⟩

lemma *partitions-ss*: $X \subseteq V$ $Y \subseteq V$
⟨*proof*⟩

end

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

sublocale *bipartite-graph* $\subseteq$ *sgraph*
⟨*proof*⟩

context *bipartite-graph*
begin

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

lemma *bipartite-sym*: *bipartite-graph V E Y X*
⟨*proof*⟩

lemma *X-verts-not-adj*:
assumes $x1 \in X$ $x2 \in X$
shows $\neg$ *vert-adj x1 x2*
⟨*proof*⟩

lemma *Y-verts-not-adj*:
assumes $y1 \in Y$ $y2 \in Y$
shows $\neg$ *vert-adj y1 y2*
⟨*proof*⟩

lemma *X-vert-adj-Y*: $x \in X \Longrightarrow$ *vert-adj x y* $\Longrightarrow y \in Y$
⟨*proof*⟩

lemma *Y-vert-adj-X*: $y \in Y \Longrightarrow$ *vert-adj y x* $\Longrightarrow x \in X$
⟨*proof*⟩

lemma *neighbors-ss-eq-neighborhoodX*: $v \in X \Longrightarrow$ *neighborhood v* = *neighbors-ss v Y*
⟨*proof*⟩

lemma *neighbors-ss-eq-neighborhoodY*: $v \in Y \Longrightarrow$ *neighborhood v* = *neighbors-ss v X*
⟨*proof*⟩

lemma *neighborhood-subset-oppX*: $v \in X \Longrightarrow$ *neighborhood v* $\subseteq Y$
⟨*proof*⟩

lemma *neighborhood-subset-oppY*: $v \in Y \Longrightarrow$ *neighborhood v* $\subseteq X$
⟨*proof*⟩

lemma *degree-neighbors-ssX*: $v \in X \Longrightarrow$ *degree* $v =$ *card* (*neighbors-ss* v Y)
  ⟨*proof*⟩

lemma *degree-neighbors-ssY*: $v \in Y \Longrightarrow$ *degree* $v =$ *card* (*neighbors-ss* v X)
  ⟨*proof*⟩

definition *is-bicomplete*:: *bool* **where**
is-bicomplete $\equiv E =$ *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*⟩

lemma *edges-between-equals-edge-set*: *mk-edge* ‘ (*all-edges-between* X Y) $= E$
  ⟨*proof*⟩

Lemmas for reasoning on walks and paths in a bipartite graph

lemma *walk-alternates*:
  assumes *is-walk* w
  assumes *Suc* $i <$ *length* w $i \geq 0$
  shows w ! $i \in X \longleftrightarrow w$! $(i + 1) \in Y$
⟨*proof*⟩

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 $i \geq 0$
  shows w ! $i \in Y \longleftrightarrow w$! $(i + 1) \in X$
⟨*proof*⟩

lemma *walk-length-even*:
  assumes *is-walk* w
  assumes *hd* $w \in X$ **and** *last* $w \in X$
  shows *even* (*walk-length* w)
  ⟨*proof*⟩

lemma *walk-length-even-sym*:
  assumes *is-walk* w
  assumes *hd* $w \in Y$
  assumes *last* $w \in Y$
  shows *even* (*walk-length* w)
⟨*proof*⟩

lemma *walk-length-odd*:
  assumes *is-walk* w

assumes *hd* $w \in X$ **and** *last* $w \in Y$
shows *odd* (*walk-length* w)
⟨*proof*⟩

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*⟩

lemma *walk-length-even-iff*:
assumes *is-walk* w
shows *even* (*walk-length* w) $\longleftrightarrow$ (*hd* $w \in X \wedge$ *last* $w \in X$) $\vee$ (*hd* $w \in Y \wedge$ *last* $w \in Y$)
⟨*proof*⟩

lemma *walk-length-odd-iff*:
assumes *is-walk* w
shows *odd* (*walk-length* w) $\longleftrightarrow$ (*hd* $w \in X \wedge$ *last* $w \in Y$) $\vee$ (*hd* $w \in Y \wedge$ *last* $w \in X$)
⟨*proof*⟩

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*⟩

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* $V\ E\ Y\ X$
⟨*proof*⟩

lemma *partitions-finite*: *finite* X *finite* Y
⟨*proof*⟩

lemma *card-edges-between-set*: *card* (*all-edges-between* $X\ Y$) = *card* E
⟨*proof*⟩

lemma *density-simp*: *density* = *card* (E) / ((*card* X) $*$ (*card* Y))
⟨*proof*⟩

lemma *edge-size-degree-sumY*: *card E* = ($\sum$ *y* ∈ *Y* . *degree y*)
⟨*proof*⟩

lemma *edge-size-degree-sumX*: *card E* = ($\sum$ *y* ∈ *X* . *degree y*)
⟨*proof*⟩

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*
⟨*proof*⟩

sublocale *fin-graph-system* ⊆ *finc*: *finite-incidence-system V mset-set E*
⟨*proof*⟩

sublocale *fin-ulgraph* ⊆ *d*: *design V mset-set E*
⟨*proof*⟩

sublocale *fin-ulgraph* ⊆ *d*: *simple-design V mset-set E*
⟨*proof*⟩

locale *graph-has-edges* = *graph-system* +
assumes *edges-nempty*: *E* ≠ {}

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* ⊆ *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*⟩

locale *fin-bipartite-graph-wedges* = *fin-bipartite-graph* + *fin-sgraph-wedges*

sublocale *fin-bipartite-graph-wedges* ⊆ *group-design V mset-set E* $\{X, Y\}$
⟨*proof*⟩

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* $\Longrightarrow u \in V \wedge v \in V$
begin

abbreviation *adj u v* ≡ $(u, v) \in$ *adj-rel*

lemma *wf-alt*: *adj u v* $\Longrightarrow (u, v) \in V \times V$
⟨*proof*⟩

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-set ≡ $\{\{u, v\} \mid u\ v.\ adj\ u\ v\}$

lemma *obtain-edge-pair-adj*:
assumes $e \in$ *edge-set*
obtains *u v* **where** $e = \{u, v\}$ **and** *adj u v*
⟨*proof*⟩

lemma *adj-to-edge-set-card*:
assumes $e \in$ *edge-set*
shows *card e* = *1* ∨ *card e* = *2*
⟨*proof*⟩

lemma *adj-to-edge-set-card-lim*:
assumes $e \in$ *edge-set*
shows *card e* > *0* ∧ *card e* ≤ *2*
⟨*proof*⟩

lemma *edge-set-wf*: $e \in$ *edge-set* $\Longrightarrow e \subseteq V$

⟨*proof*⟩

lemma *is-graph-system*: *graph-system V edge-set*
⟨*proof*⟩

lemma *sym-alt*: *adj u v* ⟷ *adj v u*
⟨*proof*⟩

lemma *is-ulgraph*: *ulgraph V edge-set*
⟨*proof*⟩

end

context *ulgraph*
begin

definition *adj-relation* :: *'a rel* **where**
adj-relation ≡ {(*u*, *v*) | *u v* . *vert-adj u v*}

lemma *adj-relation-wf*: (*u*, *v*) ∈ *adj-relation* ⟹ {*u*, *v*} ⊆ *V*
⟨*proof*⟩

lemma *adj-relation-sym*: *sym adj-relation*
⟨*proof*⟩

lemma *is-ulgraph-rel*: *ulgraph-rel V adj-relation*
⟨*proof*⟩

Temporary interpretation - mutual sublocale setup

interpretation *ulgraph-rel V adj-relation* ⟨*proof*⟩

lemma *vert-adj-rel-iff*:
assumes *u* ∈ *V v* ∈ *V*
shows *vert-adj u v* ⟷ *adj u v*
⟨*proof*⟩

lemma *edges-rel-is*: *E* = *edge-set*
⟨*proof*⟩

end

context *ulgraph-rel*
begin

Temporary interpretation - mutual sublocale setup

interpretation *ulgraph V edge-set* ⟨*proof*⟩

lemma *rel-vert-adj-iff*: *vert-adj u v* ⟷ *adj u v*
⟨*proof*⟩

lemma *rel-item-is*: $(u,\ v) \in$ *adj-rel* $\longleftrightarrow$ $(u,\ v) \in$ *adj-relation*
⟨*proof*⟩

lemma *rel-edges-is*: *adj-rel* = *adj-relation*
⟨*proof*⟩

end

sublocale *ulgraph-rel* $\subseteq$ *ulgraph V edge-set*
rewrites *ulgraph.adj-relation edge-set* = *adj-rel*
⟨*proof*⟩

sublocale *ulgraph* $\subseteq$ *ulgraph-rel V adj-relation*
rewrites *ulgraph-rel.edge-set adj-relation* = *E*
⟨*proof*⟩

locale *sgraph-rel* = *ulgraph-rel* +
assumes *irrefl-adj*: *irrefl adj-rel*
begin

lemma *irrefl-alt*: *adj u v* $\Longrightarrow$ $u \neq v$
⟨*proof*⟩

lemma *edge-is-card2*:
assumes $e \in$ *edge-set*
shows *card e* = *2*
⟨*proof*⟩

lemma *is-sgraph*: *sgraph V edge-set*
⟨*proof*⟩

end

context *sgraph*
begin

lemma *is-rel-irrefl-alt*:
assumes $(u,\ v) \in$ *adj-relation*
shows $u \neq v$
⟨*proof*⟩

lemma *is-rel-irrefl*: *irrefl adj-relation*
⟨*proof*⟩

lemma *is-sgraph-rel*: *sgraph-rel V adj-relation*
⟨*proof*⟩

end

sublocale *sgraph-rel* ⊆ *sgraph V edge-set*
rewrites *ulgraph.adj-relation edge-set* = *adj-rel*
⟨*proof*⟩

sublocale *sgraph* ⊆ *sgraph-rel V adj-relation*
rewrites *ulgraph-rel.edge-set adj-relation* = *E*
⟨*proof*⟩

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-Argyraki, and L. C. Paulson. Roth's Theorem on Arithmetic Progressions. *Archive of Formal Proofs*, Dec. 2021.

[2] C. Edmonds, A. Koutsoukou-Argyraki, 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.