

Menger's Theorem

Christoph Dittmann
isabelle@christoph-d.de

November 29, 2023

We present a formalization of Menger's Theorem for directed and undirected graphs in Isabelle/HOL. This well-known result shows that if two non-adjacent distinct vertices u, v in a directed graph have no separator smaller than n , then there exist n internally vertex-disjoint paths from u to v .

The version for undirected graphs follows immediately because undirected graphs are a special case of directed graphs.

Contents

1	Introduction	3
2	Relation to Min-Cut Max-Flow	3
3	Helpers	3
4	Graphs	4
4.1	Walks	5
4.2	Paths	6
4.3	The Set of All Paths	6
4.4	Edges of Walks	8
4.5	The First Edge of a Walk	11
4.6	Distance	12
4.7	Subgraphs	12
4.8	Two Distinguished Distinct Non-adjacent Vertices.	14
4.9	Undirected Graphs	14
5	Separations	15
6	Internally Vertex-Disjoint Paths	16
6.1	Basic Properties	16
6.2	Second Vertices	17
7	One More Path	18
7.1	Characterizing the New Path	19

7.2	The Last Vertex of the New Path	19
7.3	Removing the Last Vertex	20
7.4	A New Path Following the Other Paths	21
8	Induction of Menger's Theorem	22
8.1	No Small Separations	22
8.2	Choosing Paths Avoiding <i>new_last</i>	23
8.3	Finding a Path Avoiding Q	24
8.4	Decomposing P_k	25
9	The case $y = new_last$	27
10	The case $y \neq new_last$	31
11	Menger's Theorem	35
11.1	Menger's Theorem	37
11.2	Self-contained Statement of the Main Theorem	39
	Bibliography	41

1 Introduction

Given two non-adjacent distinct vertices u, v in a finite directed graph, a u - v -separator is a set of vertices S with $u \notin S, v \notin S$ such that every u - v -path visits a vertex of S . Two u - v -paths are *internally vertex-disjoint* if their intersection is exactly $\{u, v\}$.

A famous classical result of graph theory relates the size of a minimum separator to the maximal number of internally vertex-disjoint paths.

Theorem 1 (Menger [Men27]) *Let u, v be two non-adjacent distinct vertices. Then the size of a minimum u - v -separator equals the maximal number of pairwise internally vertex-disjoint u - v -paths.*

This theorem has many proofs, but as far as the author is aware, there was no formalized proof. We follow a proof given by William McCuaig, who calls it “A simple proof of Menger’s theorem” [McC84]. His proof is roughly one page in length. Our formalization is significantly longer than that because we had to fill in a lot of details.

Most of the work goes into showing the following theorem, which proves one direction of Theorem 1.

Theorem 2 *Let u, v be two non-adjacent distinct vertices. If every u - v -separator has size at least n , then there exists n pairwise internally vertex-disjoint u - v -paths.*

Compared to this, the other direction of Theorem 1 is easy because the existence of n internally vertex-disjoint paths implies that every separator needs to cut at least these paths, so every separator needs to have size at least n .

2 Relation to Min-Cut Max-Flow

Another famous result of graph theory is the Min-Cut Max-Flow Theorem, stating that the size of a minimum u - v -cut equals the value of a maximum u - v -flow. There exists a formalization of a very general version of this theorem for countable graphs in the Archive of Formal Proofs, written by Andreas Lochbihler [Loc16].

Technically, our version of Menger’s Theorem should follow from Lochbihler’s very general result. However, the author was of the opinion that a fresh formalization of Menger’s Theorem was warranted given the complexity of the Min-Cut Max-Flow formalization. Our formalization is about a sixth of the size of the Min-Cut Max-Flow formalization (not counting comments). It may also be easier to grasp by readers who are unfamiliar with the intricacies of countable networks.

Let us also note that the Min-Cut Max-Flow Theorem considers *edge cuts* whereas Menger’s Theorem works with *vertex cuts*. This is a minor difference because one can be reduced to the other, but it makes Menger’s Theorem not a trivial corollary of the Min-Cut Max-Flow formalization.

3 Helpers

```
theory Helpers imports Main begin
```

First, we will prove a few lemmas unrelated to graphs or Menger's Theorem. These lemmas will simplify some of the other proof steps.

If two finite sets have different cardinality, then there exists an element in the larger set that is not in the smaller set.

```

lemma card-finite-less-ex:
  assumes finite-A: finite A
    and finite-B: finite B
    and card-AB: card A < card B
  shows  $\exists b \in B. b \notin A$ 
proof-
  have card (B - A) > 0 using finite-A finite-B card-AB
    by (meson Diff-eq-empty-iff card-eq-0-iff card-mono finite-Diff gr0I leD)
  then show ?thesis using finite-B
    by (metis Diff-eq-empty-iff card-0-eq finite-Diff neq-iff subsetI)
qed

```

The cardinality of the union of two disjoint finite sets is the sum of their cardinalities even if we intersect everything with a fixed set X .

```

lemma card-intersect-sum-disjoint:
  assumes finite B finite C A = B  $\cup$  C B  $\cap$  C = {}
  shows card (A  $\cap$  X) = card (B  $\cap$  X) + card (C  $\cap$  X)
  by (metis (no-types, lifting) Un-Diff-Int assms card-Un-disjoint finite-Int inf.commute inf-sup-distrib2 sup-eq-bot-iff)

```

If x is in a list xs but is not its last element, then it is also in *butlast* xs .

```

lemma set-butlast:  $\llbracket x \in \text{set } xs; x \neq \text{last } xs \rrbracket \implies x \in \text{set } (\text{butlast } xs)$ 
  by (metis butlast.simps(2) in-set-butlast-appendI last.simps last-appendR list.set-intros(1) split-list-first)

```

If a property P is satisfiable and if we have a weight measure mapping into the natural numbers, then there exists an element of minimum weight satisfying P because the natural numbers are well-ordered.

```

lemma arg-min-ex:
  fixes P :: 'a  $\Rightarrow$  bool' and weight :: 'a  $\Rightarrow$  nat'
  assumes  $\exists x. P x$ 
  obtains x where P x  $\wedge$   $\forall y. P y \implies \text{weight } x \leq \text{weight } y$ 
proof (cases  $\exists x. P x \wedge \text{weight } x = 0$ )
  case True then show ?thesis using that by auto
next
  case False then show ?thesis
    using that ex-least-nat-le[ $\text{of } \lambda n. \exists x. P x \wedge \text{weight } x = n$ ] assms by (metis not-le-imp-less)
qed
end

```

4 Graphs

```

theory Graph imports Main begin

```

Let us now define digraphs, graphs, walks, paths, and related concepts.

'a is the vertex type.

type-synonym 'a Edge = 'a × 'a

type-synonym 'a Walk = 'a list

record 'a Graph =

 verts :: 'a set (V1)

 arcs :: 'a Edge set (E1)

abbreviation is-arc :: ('a, 'b) Graph-scheme ⇒ 'a ⇒ 'a ⇒ bool (**infixl** →₁ 60) **where**

 v →_G w ≡ (v,w) ∈ E_G

We consider directed and undirected finite graphs. Our graphs do not have multi-edges.

locale Digraph =

fixes G :: ('a, 'b) Graph-scheme (**structure**)

assumes finite-vertex-set: finite V

and valid-edge-set: E ⊆ V × V

context Digraph **begin**

lemma finite-edge-set [simp]: finite E **using** finite-vertex-set valid-edge-set

by (simp add: finite-subset)

lemma edges-are-in-V: **assumes** v→w **shows** v ∈ V w ∈ V

using assms valid-edge-set **by** blast+

4.1 Walks

A walk is sequence of vertices connected by edges.

inductive walk :: 'a Walk ⇒ bool **where**

 Nil [simp]: walk []

 | Singleton [simp]: v ∈ V ⇒ walk [v]

 | Cons: v→w ⇒ walk (w # vs) ⇒ walk (v # w # vs)

Show a few composition/decomposition lemmas for walks. These will greatly simplify the proofs that follow.

lemma walk-2 [simp]: v→w ⇒ walk [v,w] **by** (simp add: edges-are-in-V(2) walk.intros(3))

lemma walk-comp: [walk xs; walk ys; xs = Nil ∨ ys = Nil ∨ last xs→hd ys] ⇒ walk (xs @ ys)

by (induct rule: walk.induct, simp-all add: walk.intros(3))

 (metis list.exhaust-sel walk.intros(2) walk.intros(3))

lemma walk-tl: walk xs ⇒ walk (tl xs) **by** (induct rule: walk.induct) simp-all

lemma walk-drop: walk xs ⇒ walk (drop n xs) **by** (induct n, simp) (metis drop-Suc tl-drop walk-tl)

lemma walk-take: walk xs ⇒ walk (take n xs)

by (induct arbitrary: n rule: walk.induct)

 (simp, metis Digraph.walk.simps Digraph-axioms take-Cons' take-eq-Nil,

 metis Digraph.walk.simps Digraph-axioms edges-are-in-V(1) take-Cons')

lemma walk-decomp: **assumes** walk (xs @ ys) **shows** walk xs walk ys

using assms append-eq-conv-conj[of xs ys xs @ ys] walk-take walk-drop **by** metis+

lemma walk-in-V: walk xs ⇒ set xs ⊆ V **by** (induct rule: walk.induct; simp add: edges-are-in-V)

lemma walk-first-edge: walk (v # w # xs) ⇒ v→w **using** walk.cases **by** fastforce

lemma walk-first-edge': [walk (v # xs); xs ≠ Nil] ⇒ v→hd xs

using *walk-first-edge* **by** (*metis list.exhaust-sel*)
lemma *walk-middle-edge*: $walk (xs @ v \# w \# ys) \implies v \rightarrow w$
by (*induct xs @ v \# w \# ys arbitrary: xs rule: walk.induct, simp, simp*)
(*metis list.sel(1,3) self-append-conv2 tl-append2*)
lemma *walk-last-edge*: $\llbracket walk (xs @ ys); xs \neq Nil; ys \neq Nil \rrbracket \implies last\ xs \rightarrow hd\ ys$
using *walk-middle-edge* [*of butlast xs last xs hd ys tl ys*]
by (*metis Cons-eq-appendI append-butlast-last-id append-eq-append-conv2 list.exhaust-sel self-append-conv*)

4.2 Paths

A path is a walk without repeated vertices. This is simple enough, so most of the above lemmas transfer directly to paths.

abbreviation *path* :: 'a Walk \Rightarrow bool **where** *path xs* $\equiv walk\ xs \wedge distinct\ xs$

lemma *path-singleton* [*simp*]: $v \in V \implies path\ [v]$ **by** *simp*
lemma *path-2* [*simp*]: $\llbracket v \rightarrow w; v \neq w \rrbracket \implies path\ [v,w]$ **by** *simp*
lemma *path-cons*: $\llbracket path\ xs; xs \neq Nil; v \rightarrow hd\ xs; v \notin set\ xs \rrbracket \implies path\ (v \# xs)$
by (*metis distinct.simps(2) list.exhaust-sel walk.Cons*)
lemma *path-comp*: $\llbracket walk\ xs; walk\ ys; xs = Nil \vee ys = Nil \vee last\ xs \rightarrow hd\ ys; distinct\ (xs @ ys) \rrbracket$
 $\implies path\ (xs @ ys)$ **using** *walk-comp* **by** *blast*
lemma *path-tl*: $path\ xs \implies path\ (tl\ xs)$ **by** (*simp add: distinct-tl walk-tl*)
lemma *path-drop*: $path\ xs \implies path\ (drop\ n\ xs)$ **by** (*simp add: walk-drop*)
lemma *path-take*: $path\ xs \implies path\ (take\ n\ xs)$ **by** (*simp add: walk-take*)
lemma *path-decomp*: **assumes** $path\ (xs @ ys)$ **shows** $path\ xs\ path\ ys$
using *walk-decomp* *assms distinct-append* **by** *blast+*
lemma *path-decomp'*: $path\ (xs @ x \# ys) \implies path\ (xs @ [x])$
by (*metis Singleton distinct.simps(2) distinct1-rotate edges-are-in-V(1) list.discI list.sel(1)*)
not-distinct-conv-prefix path-decomp(1) rotate1.simps(2) walk-comp walk-decomp(2)
walk-first-edge' walk-last-edge)
lemma *path-in-V*: $path\ xs \implies set\ xs \subseteq V$ **by** (*simp add: walk-in-V*)
lemma *path-length*: $path\ xs \implies length\ xs \leq card\ V$
by (*metis card-mono distinct-card finite-vertex-set path-in-V*)
lemma *path-first-edge*: $path\ (v \# w \# xs) \implies v \rightarrow w$ **using** *walk-first-edge* **by** *blast*
lemma *path-first-edge'*: $\llbracket path\ (v \# xs); xs \neq Nil \rrbracket \implies v \rightarrow hd\ xs$ **using** *walk-first-edge'* **by** *blast*
lemma *path-middle-edge*: $path\ (xs @ v \# w \# ys) \implies v \rightarrow w$ **using** *walk-middle-edge* **by** *blast*
lemma *path-first-vertex*: $path\ (x \# xs) \implies x \notin set\ xs$ **by** *simp*
lemma *path-disjoint*: $\llbracket path\ (xs @ ys); xs \neq Nil; x \in set\ xs \rrbracket \implies x \notin set\ ys$ **by** *auto*

4.3 The Set of All Paths

definition *all-paths* **where** *all-paths* $\equiv \{ xs \mid xs.\ path\ xs \}$

Because paths have no repeated vertices, every graph has at most finitely many distinct paths. This will be useful later to easily derive that any set of paths is finite.

lemma *finitely-many-paths*: *finite all-paths* **proof**—
have $all-paths \subseteq \{xs.\ set\ xs \subseteq V \wedge length\ xs \leq card\ V\}$
unfolding *all-paths-def* **using** *path-length* **by** (*simp add: Collect-mono path-in-V*)
thus *?thesis* **using** *finite-lists-length-le* [*OF finite-vertex-set*] *walk-in-V infinite-super* **by** *blast*
qed

end — context Digraph

We introduce shorthand notation for a path connecting two vertices.

definition *path-from-to* :: ('a, 'b) Graph-scheme \Rightarrow 'a \Rightarrow 'a Walk \Rightarrow 'a \Rightarrow bool
 (- \rightsquigarrow \rightsquigarrow_1 - [71, 71, 71] 70) **where**
path-from-to G v xs w \equiv Digraph.path G xs \wedge xs \neq Nil \wedge hd xs = v \wedge last xs = w

context Digraph **begin**

lemma *path-from-toI* [intro]: \llbracket path xs; xs \neq Nil; hd xs = v; last xs = w $\rrbracket \Longrightarrow$ v \rightsquigarrow xs \rightsquigarrow w
and *path-from-toE* [dest]: v \rightsquigarrow xs \rightsquigarrow w \Longrightarrow path xs \wedge xs \neq Nil \wedge hd xs = v \wedge last xs = w
unfolding *path-from-to-def* **by** blast+

lemma *path-from-to-ends*: v \rightsquigarrow (xs @ w # ys) \rightsquigarrow w \Longrightarrow ys = Nil
by (metis *path-from-toE distinct.simps(2) last.simps last-appendR last-in-set list.discI path-decomp(2)*)

lemma *path-from-to-combine*:

assumes v \rightsquigarrow (xs @ x # xs') \rightsquigarrow w v' \rightsquigarrow (ys @ x # ys') \rightsquigarrow w' set xs \cap set ys' = {}
shows v \rightsquigarrow (xs @ x # ys') \rightsquigarrow w'

proof

show path (xs @ x # ys')
by (metis *path-from-toE assms(1,2,3) disjoint-insert(1) distinct-append list.sel(1) list.set(2) list.simps(3) path-decomp(2) walk-comp walk-decomp(1) walk-last-edge*)
show hd (xs @ x # ys') = v **by** (metis *path-from-toE assms(1) hd-append list.sel(1)*)
show last (xs @ x # ys') = w' **using** *assms(2)* **by** auto

qed simp

lemma *path-from-to-first*: v \rightsquigarrow xs \rightsquigarrow w \Longrightarrow v \notin set (tl xs)
by (metis *path-from-toE list.collapse path-first-vertex*)

lemma *path-from-to-first'*: v \rightsquigarrow (xs @ x # xs') \rightsquigarrow w \Longrightarrow v \notin set xs'
by (metis *path-from-toE append-eq-append-conv2 distinct.simps(2) hd-append list.exhaust-sel list.sel(3) list.set-sel(1,2) list.simps(3) path-disjoint self-append-conv*)

lemma *path-from-to-last*: v \rightsquigarrow xs \rightsquigarrow w \Longrightarrow w \notin set (butlast xs)
by (metis *path-from-toE append-butlast-last-id distinct-append not-distinct-conv-prefix*)

lemma *path-from-to-last'*: v \rightsquigarrow (xs @ x # xs') \rightsquigarrow w \Longrightarrow w \notin set xs
by (metis *path-from-toE be-empty last-appendR last-in-set list.set(1) list.simps(3) path-disjoint*)

Every walk contains a path connecting the same vertices.

lemma *walk-to-path*:

assumes walk xs xs \neq Nil hd xs = v last xs = w
shows \exists ys. v \rightsquigarrow ys \rightsquigarrow w \wedge set ys \subseteq set xs

proof -

We prove this by removing loops from xs until xs is a path. We want to perform induction over length xs, but xs in set ys \subseteq set xs should not be part of the induction hypothesis. To accomplish this, we hide set xs behind a definition for this specific part of the goal.

define *target-set* **where** target-set \equiv set xs

hence set xs \subseteq target-set **by** simp

thus \exists ys. v \rightsquigarrow ys \rightsquigarrow w \wedge set ys \subseteq target-set

using *assms* **proof** (induct length xs arbitrary: xs rule: infinite-descent0)

case (*smaller n*)
then obtain xs **where**
 $xs: n = \text{length } xs \text{ walk } xs \text{ } xs \neq \text{Nil } hd \text{ } xs = v \text{ last } xs = w \text{ set } xs \subseteq \text{target-set}$ **and**
 $hyp: \neg(\exists ys. v \rightsquigarrow ys \rightsquigarrow w \wedge \text{set } ys \subseteq \text{target-set})$ **by** *blast*

If xs is not a path, then xs is not distinct and we can decompose it.

then obtain ys $rest$ u
where $xs\text{-decomp}: u \in \text{set } ys \text{ distinct } ys \text{ } xs = ys @ u \# rest$
using *not-distinct-conv-prefix* **by** (*metis path-from-toI*)

u appears in ys , so we have a loop in xs starting from an occurrence of u in ys ending in the vertex u in $u \# rest$. We define zs as xs without this loop.

obtain ys' $ys\text{-suffix}$ **where**
 $ys\text{-decomp}: ys = ys' @ u \# ys\text{-suffix}$ **by** (*meson split-list xs-decomp(1)*)
define zs **where** $zs \equiv ys' @ u \# rest$
have $walk \text{ } zs$ **unfolding** $zs\text{-def}$ **using** $xs(2)$ $xs\text{-decomp}(3)$ $ys\text{-decomp}$
by (*metis walk-decomp list.sel(1) list.simps(3) walk-comp walk-last-edge*)
moreover have $\text{length } zs < n$ **unfolding** $zs\text{-def}$ **by** (*simp add: xs(1) xs-decomp(3) ys-decomp*)
moreover have $hd \text{ } zs = v$ **unfolding** $zs\text{-def}$
by (*metis append-is-Nil-conv hd-append list.sel(1) xs(4) xs-decomp(3) ys-decomp*)
moreover have $\text{last } zs = w$ **unfolding** $zs\text{-def}$ **using** $xs(5)$ $xs\text{-decomp}(3)$ **by** *auto*
moreover have $\text{set } zs \subseteq \text{target-set}$ **unfolding** $zs\text{-def}$ **using** $xs(6)$ $xs\text{-decomp}(3)$ $ys\text{-decomp}$ **by**
auto
ultimately show $?case$ **using** $zs\text{-def}$ hyp **by** *blast*
qed *simp*
qed

4.4 Edges of Walks

The set of edges on a walk. Note that this is empty for walks of length 0 or 1.

definition $edges\text{-of-walk} :: 'a \text{ Walk} \Rightarrow 'a \text{ Edge set}$ **where**
 $edges\text{-of-walk } xs = \{ (v,w) \mid v w \text{ } xs\text{-pre } xs\text{-post}. xs = xs\text{-pre} @ v \# w \# xs\text{-post} \}$

lemma $edges\text{-of-walk}E: (v,w) \in edges\text{-of-walk } xs \Longrightarrow \exists xs\text{-pre } xs\text{-post}. xs = xs\text{-pre} @ v \# w \# xs\text{-post}$
unfolding $edges\text{-of-walk}\text{-def}$ **by** *blast*

lemma $edges\text{-of-walk-in-E}: walk \text{ } xs \Longrightarrow edges\text{-of-walk } xs \subseteq E$
unfolding $edges\text{-of-walk}\text{-def}$ **using** $walk\text{-middle-edge}$ **by** *auto*

lemma $edges\text{-of-walk-finite}: walk \text{ } xs \Longrightarrow finite (edges\text{-of-walk } xs)$
using $edges\text{-of-walk-in-E}$ $finite\text{-edge-set}$ $finite\text{-subset}$ **by** *blast*

lemma $edges\text{-of-walk-empty}: edges\text{-of-walk } [] = \{ \}$ $edges\text{-of-walk } [v] = \{ \}$
unfolding $edges\text{-of-walk}\text{-def}$ **by** *simp-all*

lemma $edges\text{-of-walk-2}: edges\text{-of-walk } [v,w] = \{(v,w)\}$ **proof**
 $\{$
fix $v' w'$ **assume** $(v', w') \in edges\text{-of-walk } [v,w]$
then obtain $xs\text{-pre } xs\text{-post}$ **where** $xs\text{-decomp}: [v,w] = xs\text{-pre} @ v' \# w' \# xs\text{-post}$
using $edges\text{-of-walk}E[of \ v' \ w' \ [v,w]]$ **by** *blast*

```

then have  $xs\text{-pre} = Nil$ 
  by (metis Nil-is-append-conv butlast.simps(2) butlast-append list.discI)
  then have  $(v',w') \in \{(v,w)\}$  using  $xs\text{-decomp}$  by simp
}
then show  $edges\text{-of-walk } [v, w] \subseteq \{(v, w)\}$  by (simp add: subrelI)
show  $\{(v, w)\} \subseteq edges\text{-of-walk } [v, w]$  unfolding  $edges\text{-of-walk-def}$  by blast
qed

```

```

lemma  $edges\text{-of-walk-edge}$ :  $\llbracket walk\ xs; (v,w) \in edges\text{-of-walk } xs \rrbracket \implies v \rightarrow w$ 
  using  $edges\text{-of-walkE walk-middle-edge}$  by fastforce

```

```

lemma  $edges\text{-of-walk-middle}$  [simp]:  $(v,w) \in edges\text{-of-walk } (xs @ v \# w \# xs')$ 
  unfolding  $edges\text{-of-walk-def}$  by blast

```

```

lemma  $edges\text{-of-comp1}$ :  $edges\text{-of-walk } xs \subseteq edges\text{-of-walk } (xs @ ys)$ 
  unfolding  $edges\text{-of-walk-def}$  by force

```

```

lemma  $edges\text{-of-comp2}$ :  $edges\text{-of-walk } ys \subseteq edges\text{-of-walk } (xs @ ys)$  proof –
{
  fix  $v\ w$  assume  $(v,w) \in edges\text{-of-walk } ys$ 
  then have  $\exists ys\text{-pre } ys\text{-post. } ys = ys\text{-pre} @ v \# w \# ys\text{-post}$  by (meson edges-of-walkE)
  then have  $(v,w) \in edges\text{-of-walk } (xs @ ys)$ 
    by (metis (mono-tags, lifting) append.assoc edges-of-walk-def mem-Collect-eq)
}
then show ?thesis by (simp add: subrelI)
qed

```

```

lemma  $walk\text{-edges-decomp-simple}$ :
   $edges\text{-of-walk } (v \# w \# xs) = \{(v,w)\} \cup edges\text{-of-walk } (w \# xs)$  (is  $?A = ?B$ )
proof
  have  $edges\text{-of-walk } (w \# xs) \subseteq ?A$  using  $edges\text{-of-comp2}[of\ w \# xs\ [v]]$  by simp
  moreover have  $(v,w) \in ?A$  by (metis append-eq-Cons-conv edges-of-walk-middle)
  ultimately show  $?B \subseteq ?A$  by blast
{
  fix  $v'\ w'$  assume  $(v',w') \in ?A$ 
  then obtain  $xs\text{-pre } xs\text{-post}$  where  $xs\text{-decomp}: v \# w \# xs = xs\text{-pre} @ v' \# w' \# xs\text{-post}$ 
    using  $edges\text{-of-walkE}$  by blast
  have  $(v',w') \in ?B$  proof (cases)
    assume  $xs\text{-pre} = Nil$  then show ?thesis using  $xs\text{-decomp}$  by auto
  next
    assume  $xs\text{-pre} \neq Nil$  then show ?thesis
      by (metis Cons-eq-append-conv UnI2 edges-of-walk-middle xs-decomp)
  }
then show  $?A \subseteq ?B$  by auto
qed

```

```

lemma  $walk\text{-edges-decomp}$ :
   $edges\text{-of-walk } (xs @ x \# xs') = edges\text{-of-walk } (xs @ [x]) \cup edges\text{-of-walk } (x \# xs')$ 
proof (induct xs)
  case (Cons v xs)
  show ?case proof (cases)
    assume  $xs = Nil$ 

```

```

    then show ?thesis using edges-of-walk-2 walk-edges-decomp-simple by auto
  next
    assume  $xs \neq Nil$ 
    then obtain  $w$   $xs\text{-post}$  where  $xs = w \# xs\text{-post}$  using list.exhaust-sel by blast
    then show ?thesis using Cons.hyps walk-edges-decomp-simple by auto
  qed
qed (simp add: edges-of-walk-empty(2))

lemma walk-edges-decomp':
  edges-of-walk ( $xs @ v \# w \# xs'$ ) = edges-of-walk ( $xs @ [v]$ )  $\cup$   $\{(v,w)\}$   $\cup$  edges-of-walk ( $w \# xs'$ )
  using walk-edges-decomp walk-edges-decomp-simple by (metis sup.assoc)

lemma walk-edges-vertices: assumes  $(v, w) \in$  edges-of-walk  $xs$  shows  $v \in$  set  $xs$   $w \in$  set  $xs$ 
  using assms edges-of-walkE by force+

lemma walk-edges-subset:
  assumes edges-subsets: edges-of-walk  $xs \subseteq$  edges-of-walk  $ys$ 
    and non-trivial:  $tl\ xs \neq Nil$ 
  shows set  $xs \subseteq$  set  $ys$ 
proof
  fix  $v$  assume  $v \in$  set  $xs$ 
  then obtain  $xs\text{-pre}$   $xs\text{-post}$  where
     $xs\text{-decomp}$ :  $xs = xs\text{-pre} @ v \# xs\text{-post}$  by (meson split-list)
  show  $v \in$  set  $ys$  proof (cases)
    assume  $xs\text{-pre} = Nil$ 
    then have  $xs\text{-post} \neq Nil$  using  $xs\text{-decomp}$  non-trivial by auto
    then have  $xs = xs\text{-pre} @ v \# hd\ xs\text{-post} \# tl\ xs\text{-post}$  by (simp add:  $xs\text{-decomp}$ )
    then have  $(v, hd\ xs\text{-post}) \in$  edges-of-walk  $xs$  using edges-of-walk-def by auto
    then show ?thesis using walk-edges-vertices(1) edges-subsets by fastforce
  next
    assume  $xs\text{-pre} \neq Nil$ 
    then have  $xs = butlast\ xs\text{-pre} @ last\ xs\text{-pre} \# v \# xs\text{-post}$  by (simp add:  $xs\text{-decomp}$ )
    then have  $(last\ xs\text{-pre}, v) \in$  edges-of-walk  $xs$  using edges-of-walk-def by auto
    then show ?thesis using walk-edges-vertices(2) edges-subsets by fastforce
  qed
qed

```

A path has no repeated vertices, so if we split a path at an edge we find that the two pieces do not contain this edge any more.

```

lemma path-edges:
  assumes path  $xs$   $(v,w) \in$  edges-of-walk  $xs$ 
  shows  $\exists$   $xs\text{-pre}$   $xs\text{-post}$ .  $xs = xs\text{-pre} @ v \# w \# xs\text{-post}$ 
     $\wedge (v,w) \notin$  edges-of-walk ( $xs\text{-pre} @ [v]$ )
     $\wedge (v,w) \notin$  edges-of-walk ( $w \# xs\text{-post}$ )
proof-
  obtain  $xs\text{-pre}$   $xs\text{-post}$  where
     $xs\text{-decomp}$ :  $xs = xs\text{-pre} @ v \# w \# xs\text{-post}$  by (meson assms(2) edges-of-walkE)
  then have  $(v,w) \notin$  edges-of-walk ( $xs\text{-pre} @ [v]$ ) using assms(1) edges-of-walkE
    by (metis path-from-to-ends list.discI path-decomp' path-from-toI snoc-eq-iff-butlast)
  moreover have  $(v,w) \notin$  edges-of-walk ( $w \# xs\text{-post}$ ) using assms(1)
    by (metis edges-of-walkE in-set-conv-decomp path-decomp(2) path-first-vertex  $xs\text{-decomp}$ )

```

ultimately show *?thesis* using *xs-decomp* by *blast*
qed

lemma *path-edges-remove-prefix*:

assumes *path* (*xs @ x # xs'*)

shows *edges-of-walk* (*xs @ [x]*) = *edges-of-walk* (*xs @ x # xs'*) - *edges-of-walk* (*x # xs'*)

proof-

{

fix *v w* assume *: (*v,w*) ∈ *edges-of-walk* (*xs @ [x]*)

then have 1: (*v,w*) ∈ *edges-of-walk* (*xs @ x # xs'*)

using *walk-edges-decomp*[*of xs x xs'*] by *force*

moreover have (*v,w*) ∉ *edges-of-walk* (*x # xs'*) **proof**

assume *contra*: (*v,w*) ∈ *edges-of-walk* (*x # xs'*)

then have *w* ∈ *set* (*x # xs'*) by (*meson walk-edges-vertices*(2))

moreover have *w* ≠ *x* using *assms contra * 1*

by (*metis path-decomp*(2) *UnE edges-of-walkE edges-of-walk-edge list.set-intros*(1)

path-2 path-disjoint path-first-vertex self-append-conv2 set-append walk-edges-vertices(1))

moreover have *w* ∈ *set* (*xs @ [x]*) by (*meson * walk-edges-vertices*(2))

ultimately show *False* using *assms* by *auto*

qed

ultimately have (*v,w*) ∈ *edges-of-walk* (*xs @ x # xs'*) - *edges-of-walk* (*x # xs'*) by *blast*

}

then show *?thesis* using *walk-edges-decomp*[*of xs x xs'*] by *auto*

qed

4.5 The First Edge of a Walk

In the proof of Menger's Theorem, we will often talk about the first edge of a path. Let us define this concept.

fun *first-edge-of-walk* where

first-edge-of-walk (*v # w # xs*) = (*v, w*)

| *first-edge-of-walk* [*v*] = *undefined*

| *first-edge-of-walk* [] = *undefined*

lemma *first-edge-in-edges*: *tl xs* ≠ *Nil* ⇒ *first-edge-of-walk xs* ∈ *edges-of-walk xs*

unfolding *edges-of-walk-def* by (*induct rule: first-edge-of-walk.induct*) *auto*

lemma *first-edge-hd-tl*: [*v* ∼*xs* ∼*w*; *tl xs* ≠ *Nil*] ⇒ *first-edge-of-walk xs* = (*v, hd (tl xs)*)

by (*induct xs rule: first-edge-of-walk.induct*) *auto*

lemma *first-edge-first*:

assumes *v* ∼*xs* ∼*w* (*v,w'*) ∈ *edges-of-walk xs*

shows *first-edge-of-walk xs* = (*v,w'*)

using *assms* **proof** (*induct rule: first-edge-of-walk.induct*)

case (1 *v w xs*)

then show *?case*

by (*metis path-decomp*(1) *append-self-conv2 edges-of-walkE first-edge-of-walk.simps*(1)

hd-append hd-in-set not-distinct-conv-prefix path-from-toE)

next

case (2 *v*)

then show *?case* using *path-edges* by *fastforce*

qed *blast*

4.6 Distance

The distance between two vertices is the minimum length of a path. Note that this is not a symmetric function because we are on digraphs.

definition *distance* :: 'a ⇒ 'a ⇒ nat **where**
distance v w ≡ Min { length xs | xs. v ↪ xs ↪ w }

The *Min* operator applies only to finite sets, so let us prove that this is the case.

lemma *distance-lengths-finite*: finite { length xs | xs. v ↪ xs ↪ w } **proof**–
have { length xs | xs. v ↪ xs ↪ w } ⊆ { n | n. n ≤ card V } **using** *path-length* **by** *blast*
then show ?thesis **using** *finite-Collect-le-nat* **by** (*meson finite-subset*)
 qed

If we have a concrete path from *v* to *w*, then the length of this path bounds the distance from *v* to *w*.

lemma *distance-upper-bound*: v ↪ xs ↪ w ⇒ distance v w ≤ length xs
unfolding *distance-def* **using** *Min-le[OF distance-lengths-finite]* **by** *blast*

Another characterization of *distance*: If we have a concrete minimal path from *v* to *w*, this defines the distance.

lemma *distance-witness*:
assumes xs: v ↪ xs ↪ w
and xs-min: ⋀ xs'. v ↪ xs' ↪ w ⇒ length xs ≤ length xs'
shows distance v w = length xs

proof–
have ⋀ d. d ∈ { length xs | xs. v ↪ xs ↪ w } ⇒ length xs ≤ d **using** *xs-min* **by** *blast*
then show ?thesis **unfolding** *distance-def* **using** *Min-eqI*
by (*metis (mono-tags, lifting) distance-lengths-finite xs mem-Collect-eq*)
 qed

4.7 Subgraphs

We only need one kind of subgraph: The subgraph obtained by removing a single vertex.

definition *remove-vertex* :: 'a ⇒ ('a, 'b) Graph-scheme **where**
remove-vertex x ≡ G(| *verts* := V - {x}, *arcs* := Restr E (V - {x}) |)

lemma *remove-vertex-V*: V_remove-vertex x = V - {x} **unfolding** *remove-vertex-def* **by** *auto*
lemma *remove-vertex-V'*: V_remove-vertex x ⊆ V **unfolding** *remove-vertex-def* **by** *auto*
lemma *remove-vertex-E*: E_remove-vertex x = Restr E (V - {x}) **unfolding** *remove-vertex-def* **by** *simp*
lemma *remove-vertex-E'*: v →_remove-vertex x w ⇒ v → w **by** (*simp add: remove-vertex-E*)
lemma *remove-vertex-E''*: [v → w; v ≠ x; w ≠ x] ⇒ v →_remove-vertex x w
by (*simp add: edges-are-in-V remove-vertex-E*)

Of course, this is still a digraph.

lemma *remove-vertex-Digraph*: Digraph (remove-vertex v) **proof**
let ?V = V_remove-vertex v **let** ?E = E_remove-vertex v

```

show finite ?V unfolding remove-vertex-def using finite-vertex-set by simp
show ?E ⊆ ?V × ?V proof
  fix e assume e ∈ ?E
  then have e ∈ (V - {v}) × (V - {v}) by (metis Int-iff remove-vertex-E)
  then show e ∈ ?V × ?V using remove-vertex-V by auto
qed
have  $\bigwedge x y. \llbracket (x,y) \in ?E; (x,y) \notin E \rrbracket \implies (y,x) \in ?E$  unfolding remove-vertex-def by simp
qed

```

We are also going to need a few lemmas about how walks and paths behave when we remove a vertex.

First, if we remove a vertex that is not on a walk xs , then xs is still a walk after removing this vertex.

lemma *remove-vertex-walk*:

assumes *walk xs x ∉ set xs*

shows *Digraph.walk (remove-vertex x) xs*

proof –

interpret *H: Digraph remove-vertex x* **using** *remove-vertex-Digraph* **by** *blast*

show ?thesis **using** *assms* **proof** (*induct rule: walk.induct*)

case (*Singleton v*)

then have *v* ∈ V - {x} **by** *simp*

then show ?case **using** *remove-vertex-V* **by** *simp*

next

case (*Cons v w vs*)

then have *v* →_{*remove-vertex x*} *w* **using** *remove-vertex-E''* **by** *auto*

then show ?case

by (*meson Cons.hyps(3) Cons.prem(1) H.Cons assms(2) list.set-intros(2)*)

qed *simp*

qed

The same holds for paths.

lemma *remove-vertex-path-from-to*:

$\llbracket v \rightsquigarrow_{xs} w; x \in V; x \notin \text{set } xs \rrbracket \implies v \rightsquigarrow_{xs} \text{remove-vertex } x \ w$

using *path-from-to-def remove-vertex-walk* **by** *fastforce*

Conversely, if something was a walk or a path in the subgraph, then it is also a walk or a path in the supergraph.

lemma *remove-vertex-walk-add*:

assumes *Digraph.walk (remove-vertex x) xs*

shows *walk xs*

proof –

interpret *H: Digraph remove-vertex x* **using** *remove-vertex-Digraph* **by** *blast*

show ?thesis **using** *assms* **proof** (*induct rule: H.walk.induct*)

case (*Singleton v*)

then show ?case **by** (*meson Digraph.Singleton Digraph-axioms remove-vertex-V' subsetD*)

next

case (*Cons v w vs*)

then show ?case **by** (*meson Digraph.Cons Digraph-axioms remove-vertex-E'*)

qed *simp*

qed

lemma *remove-vertex-path-from-to-add*: $v \rightsquigarrow xs \rightsquigarrow \text{remove-vertex } x \ w \implies v \rightsquigarrow xs \rightsquigarrow w$
using *path-from-to-def* *remove-vertex-walk-add* **by** *fastforce*

end — context *Digraph*

4.8 Two Distinguished Distinct Non-adjacent Vertices.

The setup for Menger’s Theorem requires two distinguished distinct non-adjacent vertices $v0$ and $v1$. Let us pin down this concept with the following locale.

locale *v0-v1-Digraph* = *Digraph* +
fixes $v0 \ v1 :: 'a$
assumes $v0-V: v0 \in V$ **and** $v1-V: v1 \in V$
and $v0\text{-nonadj-}v1: \neg v0 \rightarrow v1$
and $v0\text{-neq-}v1: v0 \neq v1$

The only lemma we need about *v0-v1-Digraph* for now is that it is closed under removing a vertex that is not $v0$ or $v1$.

lemma (**in** *v0-v1-Digraph*) *remove-vertices-v0-v1-Digraph*:
assumes $v \neq v0 \ v \neq v1$
shows *v0-v1-Digraph* (*remove-vertex* v) $v0 \ v1$
proof (*rule v0-v1-Digraph.intro*)
show *v0-v1-Digraph-axioms* (*remove-vertex* v) $v0 \ v1$
using *assms v0-nonadj-v1 v0-neq-v1 v0-V v1-V remove-vertex-V remove-vertex-E'*
by *unfold-locales blast+*
qed (*simp add: remove-vertex-Digraph*)

4.9 Undirected Graphs

We represent undirected graphs as a special case of digraphs where every undirected edge is represented as an edge in both directions. We also exclude loops because loops are uncommon in undirected graphs.

As we will explain in the next paragraph, all of this has no bearing on the validity of Menger’s Theorem for undirected graphs.

locale *Graph* = *Digraph* +
assumes *undirected*: $v \rightarrow w = w \rightarrow v$
and *no-loops*: $\neg v \rightarrow v$

We observe that this makes *Digraph* a sublocale of *Graph*, meaning that every theorem we prove for digraphs automatically holds for undirected graphs, although it may not make sense because for example “connectedness” (if we were to define it) would need different definitions for directed and undirected graphs.

Fortunately, the notions of “separator” and “internally vertex-disjoint paths” on directed graphs are the same for undirected graphs. So Menger’s Theorem, when we eventually prove it in the *Digraph* locale, will apply automatically to the *Graph* locale without any additional work.

For this reason we will not use the *Graph* locale again in this proof development and it exists merely to show that undirected graphs are covered as a special case by our definitions.

end

5 Separations

theory *Separations* imports *Helpers Graph* begin

locale *Separation* = *v0-v1-Digraph* +

fixes $S :: 'a \text{ set}$

assumes $S\text{-}V: S \subseteq V$

and $v0\text{-notin-}S: v0 \notin S$

and $v1\text{-notin-}S: v1 \notin S$

and $S\text{-separates}: \bigwedge xs. v0 \rightsquigarrow xs \rightsquigarrow v1 \implies \text{set } xs \cap S \neq \{\}$

lemma (in *Separation*) *finite-S* [*simp*]: *finite S* using $S\text{-}V$ *finite-subset finite-vertex-set* by *auto*

lemma (in *v0-v1-Digraph*) *subgraph-separation-extend*:

assumes $v \neq v0$ $v \neq v1$ $v \in V$

and *Separation* (*remove-vertex v*) $v0$ $v1$ S

shows *Separation* G $v0$ $v1$ (*insert v S*)

proof (rule *Separation.intro*)

interpret G : *Separation remove-vertex v v0 v1 S* using *assms(4)* .

show *v0-v1-Digraph* G $v0$ $v1$ using *v0-v1-Digraph-axioms* .

show *Separation-axioms* G $v0$ $v1$ (*insert v S*) **proof**

show $\text{insert } v \ S \subseteq V$ by (*meson G.S-V assms(3) insert-subsetI remove-vertex-V' subset-trans*)

show $v0 \notin \text{insert } v \ S$ using $G.v0\text{-notin-}S$ *assms(1)* by *blast*

show $v1 \notin \text{insert } v \ S$ using $G.v1\text{-notin-}S$ *assms(2)* by *blast*

next

fix xs assume $v0 \rightsquigarrow xs \rightsquigarrow v1$

show $\text{set } xs \cap \text{insert } v \ S \neq \{\}$ **proof** (*cases*)

assume $v \notin \text{set } xs$

then have $v0 \rightsquigarrow xs \rightsquigarrow \text{remove-vertex } v \ v1$

using *remove-vertex-path-from-to* ($v0 \rightsquigarrow xs \rightsquigarrow v1$) *assms(3)* by *blast*

then show *?thesis* by (*simp add: G.S-separates*)

qed *simp*

qed

qed

lemma (in *v0-v1-Digraph*) *subgraph-separation-min-size*:

assumes $v \neq v0$ $v \neq v1$ $v \in V$

and *no-small-separation*: $\bigwedge S. \text{Separation } G \ v0 \ v1 \ S \implies \text{card } S \geq \text{Suc } n$

and *Separation* (*remove-vertex v*) $v0$ $v1$ S

shows $\text{card } S \geq n$

using *subgraph-separation-extend*

by (*metis Separation.finite-S Suc-leD assms card-insert-disjoint insert-absorb not-less-eq-eq*)

lemma (in *v0-v1-Digraph*) *path-exists-if-no-separation*:

assumes $S \subseteq V$ $v0 \notin S$ $v1 \notin S$ $\neg \text{Separation } G \ v0 \ v1 \ S$

shows $\exists xs. v0 \rightsquigarrow xs \rightsquigarrow v1 \wedge \text{set } xs \cap S = \{\}$

by (*meson assms Separation.intro Separation-axioms.intro v0-v1-Digraph-axioms*)

end

6 Internally Vertex-Disjoint Paths

theory *DisjointPaths* **imports** *Separations* **begin**

Menger's Theorem talks about internally vertex-disjoint v_0 - v_1 -paths. Let us define this concept.

locale *DisjointPaths* = *v0-v1-Digraph* +

fixes *paths* :: 'a Walk set

assumes *paths*:

$\bigwedge xs. xs \in paths \implies v_0 \rightsquigarrow xs \rightsquigarrow v_1$

and *paths-disjoint*: $\bigwedge xs\ ys\ v.$

$\llbracket xs \in paths; ys \in paths; xs \neq ys; v \in set\ xs; v \in set\ ys \rrbracket \implies v = v_0 \vee v = v_1$

6.1 Basic Properties

The empty set of paths trivially satisfies the conditions.

lemma (**in** *v0-v1-Digraph*) *DisjointPaths-empty*: *DisjointPaths* *G* *v0* *v1* {}

by (*simp* *add*: *DisjointPaths.intro* *DisjointPaths-axioms-def* *v0-v1-Digraph-axioms*)

Re-adding a deleted vertex is fine.

lemma (**in** *v0-v1-Digraph*) *DisjointPaths-supergraph*:

assumes *DisjointPaths* (*remove-vertex* *v*) *v0* *v1* *paths*

shows *DisjointPaths* *G* *v0* *v1* *paths*

proof

interpret *H*: *DisjointPaths* *remove-vertex* *v* *v0* *v1* *paths* **using** *assms* .

show $\bigwedge xs. xs \in paths \implies v_0 \rightsquigarrow xs \rightsquigarrow v_1$ **using** *remove-vertex-path-from-to-add* *H.paths* **by** *blast*

show $\bigwedge xs\ ys\ v. \llbracket xs \in paths; ys \in paths; xs \neq ys; v \in set\ xs; v \in set\ ys \rrbracket \implies v = v_0 \vee v = v_1$

by (*meson* *DisjointPaths.paths-disjoint* *H.DisjointPaths-axioms*)

qed

context *DisjointPaths* **begin**

lemma *paths-in-all-paths*: *paths* \subseteq *all-paths* **unfolding** *all-paths-def* **using** *paths* **by** *blast*

lemma *finite-paths*: *finite* *paths*

using *finitely-many-paths* *infinite-super* *paths-in-all-paths* **by** *blast*

lemma *paths-edge-finite*: *finite* (\bigcup (*edges-of-walk* ' *paths*)) **proof** –

have \bigcup (*edges-of-walk* ' *paths*) \subseteq *E* **using** *edges-of-walk-in-E* *paths* **by** *fastforce*

then show *?thesis* **by** (*meson* *finite-edge-set* *finite-subset*)

qed

lemma *paths-tl-notnil*: *xs* \in *paths* \implies *tl* *xs* \neq *Nil*

by (*metis* *path-from-toE* *hd-Cons-tl* *last-ConsL* *paths* *v0-neq-v1*)

lemma *paths-second-in-V*: *xs* \in *paths* \implies *hd* (*tl* *xs*) \in *V*

by (*metis* *paths* *edges-are-in-V*(2) *list.exhaust-sel* *path-from-toE* *paths-tl-notnil* *walk-first-edge*)

lemma *paths-second-not-v0*: *xs* \in *paths* \implies *hd* (*tl* *xs*) \neq *v0*

by (*metis* *distinct.simps*(2) *hd-in-set* *list.exhaust-sel* *path-from-to-def* *paths* *paths-tl-notnil*)

lemma *paths-second-not-v1*: *xs* \in *paths* \implies *hd* (*tl* *xs*) \neq *v1*

using *paths paths-tl-notnil v0-nonadj-v1 walk-first-edge'* by *fastforce*

lemma *paths-second-disjoint*: $\llbracket xs \in paths; ys \in paths; xs \neq ys \rrbracket \implies hd (tl xs) \neq hd (tl ys)$
by (*metis paths-disjoint Nil-tl hd-in-set list.set-sel(2)*)
paths-second-not-v0 paths-second-not-v1 paths-tl-notnil)

lemma *paths-edge-disjoint*:

assumes $xs \in paths$ $ys \in paths$ $xs \neq ys$

shows $edges-of-walk xs \cap edges-of-walk ys = \{\}$

proof (*rule ccontr*)

assume $edges-of-walk xs \cap edges-of-walk ys \neq \{\}$

then obtain $v w$ **where** $v-w: (v,w) \in edges-of-walk xs$ $(v,w) \in edges-of-walk ys$ **by** *auto*

then have $v \in set xs$ $w \in set xs$ $v \in set ys$ $w \in set ys$ **by** (*meson walk-edges-vertices*)⁺

then have $v = v0 \vee v = v1$ $w = v0 \vee w = v1$ **using** *assms paths-disjoint* **by** *blast*⁺

then show *False* **using** $v-w(1)$ *assms(1) v0-nonadj-v1 edges-of-walk-edge path-edges*

by (*metis distinct-length-2-or-more path-decomp(2) path-from-to-def path-from-to-ends paths*)

qed

Specify the conditions for adding a new disjoint path to the set of disjoint paths.

lemma *DisjointPaths-extend*:

assumes *P-path*: $v0 \rightsquigarrow P \rightsquigarrow v1$

and *P-disjoint*: $\bigwedge xs v. \llbracket xs \in paths; xs \neq P; v \in set xs; v \in set P \rrbracket \implies v = v0 \vee v = v1$

shows *DisjointPaths* G $v0$ $v1$ (*insert P paths*)

proof

fix xs ys v

assume $xs \in insert P paths$ $ys \in insert P paths$ $xs \neq ys$ $v \in set xs$ $v \in set ys$

then show $v = v0 \vee v = v1$

by (*metis DisjointPaths.paths-disjoint DisjointPaths-axioms P-disjoint insert-iff*)

next

show $\bigwedge xs. xs \in insert P paths \implies v0 \rightsquigarrow xs \rightsquigarrow v1$

using *P-path paths* **by** *blast*

qed

lemma *DisjointPaths-reduce*:

assumes $paths' \subseteq paths$

shows *DisjointPaths* G $v0$ $v1$ $paths'$

proof

fix xs **assume** $xs \in paths'$ **then show** $v0 \rightsquigarrow xs \rightsquigarrow v1$ **using** *assms paths* **by** *blast*

next

fix xs ys v **assume** $xs \in paths'$ $ys \in paths'$ $xs \neq ys$ $v \in set xs$ $v \in set ys$

then show $v = v0 \vee v = v1$ **by** (*meson assms paths-disjoint subsetCE*)

qed

6.2 Second Vertices

Let us now define the set of second vertices of the paths. We are going to need this in order to find a path avoiding the old paths on its first edge.

definition *second-vertex* **where** *second-vertex* $\equiv \lambda xs :: 'a$ *Walk*. $hd (tl xs)$

definition *second-vertices* **where** *second-vertices* $\equiv second-vertex \text{ ` } paths$

lemma *second-vertex-inj*: *inj-on second-vertex paths*

unfolding *second-vertex-def* **using** *paths-second-disjoint* **by** (*meson inj-onI*)

lemma *second-vertices-card*: $\text{card } \text{second-vertices} = \text{card } \text{paths}$

unfolding *second-vertices-def* **using** *finite-paths card-image second-vertex-inj* **by** *blast*

lemma *second-vertices-in-V*: $\text{second-vertices} \subseteq V$

unfolding *second-vertex-def second-vertices-def* **using** *paths-second-in-V* **by** *blast*

lemma *v0-v1-notin-second-vertices*: $v0 \notin \text{second-vertices } v1 \notin \text{second-vertices}$

unfolding *second-vertices-def second-vertex-def*

using *paths-second-not-v0 paths-second-not-v1* **by** *blast+*

lemma *second-vertices-new-path*: $\text{hd } (\text{tl } xs) \notin \text{second-vertices} \implies xs \notin \text{paths}$

by (*metis image-iff second-vertex-def second-vertices-def*)

lemma *second-vertices-first-edge*:

$\llbracket xs \in \text{paths}; \text{first-edge-of-walk } xs = (v,w) \rrbracket \implies w \in \text{second-vertices}$

unfolding *second-vertices-def second-vertex-def*

using *first-edge-hd-tl paths paths-tl-notnil* **by** *fastforce*

If we have no small separations, then the set of second vertices is not a separator and we can find a path avoiding this set.

lemma *disjoint-paths-new-path*:

assumes *no-small-separations*: $\bigwedge S. \text{Separation } G \ v0 \ v1 \ S \implies \text{card } S \geq \text{Suc } (\text{card } \text{paths})$

shows $\exists P\text{-new}. v0 \rightsquigarrow P\text{-new} \rightsquigarrow v1 \wedge \text{set } P\text{-new} \cap \text{second-vertices} = \{\}$

proof –

have $\neg \text{Separation } G \ v0 \ v1 \ \text{second-vertices}$

using *no-small-separations second-vertices-card* **by** *force*

then show *?thesis*

by (*simp add: path-exists-if-no-separation second-vertices-in-V v0-v1-notin-second-vertices*)

qed

We need the following predicate to find the first vertex on a new path that hits one of the other paths. We add the condition $x = v1$ to cover the case $\text{paths} = \{\}$.

definition *hitting-paths* **where**

$\text{hitting-paths} \equiv \lambda x. x \neq v0 \wedge ((\exists xs \in \text{paths}. x \in \text{set } xs) \vee x = v1)$

end — *DisjointPaths*

7 One More Path

Let us define a set of disjoint paths with one more path. Except for the first and last vertex, the new path must be disjoint from all other paths. The first vertex must be $v0$ and the last vertex must be on some other path. In the ideal case, the last vertex will be $v1$, in which case we are already done because we have found a new disjoint path between $v0$ and $v1$.

locale *DisjointPathsPlusOne* = *DisjointPaths* +

fixes *P-new* :: 'a *Walk*

assumes *P-new*:

$v0 \rightsquigarrow P\text{-new} \rightsquigarrow (\text{last } P\text{-new})$

and *tl-P-new*:

$tl\ P\text{-new} \neq Nil$
 $hd\ (tl\ P\text{-new}) \notin second\text{-vertices}$
and $last\text{-}P\text{-new}$:
 $hitting\text{-}paths\ (last\ P\text{-new})$
 $\bigwedge v. v \in set\ (butlast\ P\text{-new}) \implies \neg hitting\text{-}paths\ v$
begin

7.1 Characterizing the New Path

lemma $P\text{-new-hd-disjoint}$: $\bigwedge xs. xs \in paths \implies hd\ (tl\ P\text{-new}) \neq hd\ (tl\ xs)$
using $tl\text{-}P\text{-new}(2)$ **unfolding** $second\text{-vertices-def}$ $second\text{-vertex-def}$ **by** $blast$

lemma $P\text{-new-new}$: $P\text{-new} \notin paths$ **using** $P\text{-new-hd-disjoint}$ **by** $auto$

definition $paths\text{-with-new}$ **where** $paths\text{-with-new} \equiv insert\ P\text{-new}\ paths$

lemma $card\text{-}paths\text{-with-new}$: $card\ paths\text{-with-new} = Suc\ (card\ paths)$
unfolding $paths\text{-with-new-def}$ **using** $P\text{-new-new}$ **by** $(simp\ add: finite\text{-}paths)$

lemma $paths\text{-with-new-no-}Nil$: $Nil \notin paths\text{-with-new}$
using $P\text{-new}\ paths\text{-tl-notnil}$ $paths\text{-with-new-def}$ **by** $fastforce$

lemma $paths\text{-with-new-path}$: $xs \in paths\text{-with-new} \implies path\ xs$
using $P\text{-new}\ paths\ paths\text{-with-new-def}$ **by** $auto$

lemma $paths\text{-with-new-start-in-v0}$: $xs \in paths\text{-with-new} \implies hd\ xs = v0$
using $P\text{-new}\ paths\ paths\text{-with-new-def}$ **by** $auto$

7.2 The Last Vertex of the New Path

McCuaig in [McC84] calls the last vertex of $P\text{-new}$ by the name x . However, this name is somewhat confusing because it is so short and it will be visible in most places from now on, so let us give this vertex the more descriptive name of $new\text{-last}$.

definition $new\text{-pre}$ **where** $new\text{-pre} \equiv butlast\ P\text{-new}$

definition $new\text{-last}$ **where** $new\text{-last} \equiv last\ P\text{-new}$

lemma $P\text{-new-decomp}$: $P\text{-new} = new\text{-pre} @ [new\text{-last}]$
by $(metis\ new\text{-pre-def}\ append\text{-butlast-last-id}\ list.sel(2)\ tl\text{-}P\text{-new}(1)\ new\text{-last-def})$

lemma $new\text{-pre-not-}Nil$: $new\text{-pre} \neq Nil$ **using** $P\text{-new}(1)$ $hitting\text{-}paths\text{-def}$
by $(metis\ P\text{-new-decomp}\ list.sel(3)\ self\text{-append-conv2}\ tl\text{-}P\text{-new}(1))$

lemma $new\text{-pre-hitting}$: $x' \in set\ new\text{-pre} \implies \neg hitting\text{-}paths\ x'$
by $(simp\ add: new\text{-pre-def}\ last\text{-}P\text{-new}(2))$

lemma $P\text{-hit}$: $hitting\text{-}paths\ new\text{-last}$
by $(simp\ add: last\text{-}P\text{-new}(1)\ new\text{-last-def})$

lemma $new\text{-last-neq-v0}$: $new\text{-last} \neq v0$ **using** $hitting\text{-}paths\text{-def}$ $P\text{-hit}$ **by** $force$

lemma $new\text{-last-in-}V$: $new\text{-last} \in V$ **using** $P\text{-new}\ new\text{-last-def}\ path\text{-in-}V$ **by** $fastforce$

```

lemma new-last-to-v1:  $\exists R. \text{new-last} \rightsquigarrow R \rightsquigarrow \text{remove-vertex } v0 \ v1$ 
proof (cases)
  assume new-last = v1
  then have new-last  $\rightsquigarrow [v1] \rightsquigarrow \text{remove-vertex } v0 \ v1$ 
    by (metis last.simps list.sel(1) list.set(1) list.simps(15) list.simps(3) path-from-to-def
      path-singleton remove-vertex-path-from-to singletonD v0-V v0-neq-v1 v1-V)
  then show ?thesis by blast
next
  assume new-last  $\neq v1$ 
  then obtain xs where xs: xs  $\in$  paths new-last  $\in$  set xs
    using hitting-paths-def last-P-new(1) new-last-def by auto
  then obtain xs-pre xs-post where xs-decomp: xs = xs-pre @ new-last # xs-post
    by (meson split-list)
  then have new-last  $\rightsquigarrow$  (new-last # xs-post)  $\rightsquigarrow v1$  using  $\langle xs \in paths \rangle$ 
    by (metis paths last-appendR list.sel(1) list.simps(3) path-decomp(2) path-from-to-def)
  then have new-last  $\rightsquigarrow$  (new-last # xs-post)  $\rightsquigarrow \text{remove-vertex } v0 \ v1$ 
    using remove-vertex-path-from-to
    by (metis paths Set.set-insert xs-decomp xs(1) disjoint-insert(1) distinct-append hd-append
      hitting-paths-def last-P-new(1) list.set-sel(1) path-from-to-def v0-V new-last-def)
  then show ?thesis by blast
qed

```

```

lemma paths-plus-one-disjoint:
  assumes xs  $\in$  paths-with-new ys  $\in$  paths-with-new xs  $\neq$  ys v  $\in$  set xs v  $\in$  set ys
  shows v = v0  $\vee$  v = v1  $\vee$  v = new-last
proof –
  have xs  $\in$  paths  $\vee$  ys  $\in$  paths using assms(1,2,3) paths-with-new-def by auto
  then have hitting-paths v  $\vee$  v = v0 using assms(1,2,4,5) unfolding hitting-paths-def by blast
  then show ?thesis using assms last-P-new(2) set-butlast paths-disjoint
    by (metis insert-iff paths-with-new-def new-last-def)
qed

```

If the new path is disjoint, we are happy.

```

lemma P-new-solves-if-disjoint:
  new-last = v1  $\implies$   $\exists paths'. \text{DisjointPaths } G \ v0 \ v1 \ paths' \wedge \text{card } paths' = \text{Suc } (\text{card } paths)$ 
  using DisjointPaths-extend P-new(1) paths-plus-one-disjoint card-paths-with-new
  unfolding paths-with-new-def new-last-def by blast

```

7.3 Removing the Last Vertex

```

definition H-x where H-x  $\equiv$  remove-vertex new-last

```

```

lemma H-x-Digraph: Digraph H-x unfolding H-x-def using remove-vertex-Digraph .

```

```

lemma H-x-v0-v1-Digraph: new-last  $\neq v1$   $\implies$  v0-v1-Digraph H-x v0 v1 unfolding H-x-def
  using remove-vertices-v0-v1-Digraph hitting-paths-def P-hit by (simp add: H-x-def)

```

7.4 A New Path Following the Other Paths

The following lemma is one of the most complicated technical lemmas in the proof of Menger's Theorem.

Suppose we have a non-trivial path whose edges are all in the edge set of *path-with-new* and whose first edge equals the first edge of some $P \in \text{path-with-new}$. Also suppose that the path does not contain $v1$ or new-last . Then it follows by induction that this path is an initial segment of P .

Note that McCuaig does not mention this statement at all in his proof because it looks so obvious.

lemma *new-path-follows-old-paths*:

assumes $xs: v0 \rightsquigarrow xs \rightsquigarrow w$ $tl\ xs \neq Nil$ $v1 \notin set\ xs$ $\text{new-last} \notin set\ xs$
and $P: P \in \text{paths-with-new}$ $hd\ (tl\ xs) = hd\ (tl\ P)$
and *edges-subset*: $edges\text{-of-walk}\ xs \subseteq \bigcup (edges\text{-of-walk}\ 'paths\text{-with-new})$
shows $edges\text{-of-walk}\ xs \subseteq edges\text{-of-walk}\ P$
using $xs\ P(2)$ *edges-subset* **proof** (*induct length xs arbitrary: xs w*)
case 0
then show ?*case* **using** $xs(1)$ **by** *auto*
next
case (*Suc n xs w*)
have $n \neq 0$ **using** *Suc.hyps(2)* *Suc.prem(1,2)*
by (*metis path-from-toE Nitpick.size-list-simp(2) Suc-inject length-0-conv*)
show ?*case* **proof** (*cases*)
assume $n = \text{Suc } 0$
then obtain $v\ w$ **where** $v\text{-}w: xs = [v, w]$
by (*metis (full-types) Suc.hyps(2) length-0-conv length-Suc-conv*)
then have $v = v0$ **using** *Suc.prem(1)* **by** *auto*
moreover have $w = hd\ (tl\ P)$ **using** *Suc.prem(5)* $v\text{-}w$ **by** *auto*
moreover have $edges\text{-of-walk}\ xs = \{(v, w)\}$ **using** $v\text{-}w$ *edges-of-walk-2* **by** *simp*
moreover have $(v0, hd\ (tl\ P)) \in edges\text{-of-walk}\ P$ **using** $P\ tl\text{-}P\text{-new}(1)$ *P-new paths*
by (*metis first-edge-hd-tl first-edge-in-edges insert-iff paths-tl-notnil paths-with-new-def*)
ultimately show ?*thesis* **by** *auto*
next
assume $n \neq \text{Suc } 0$
obtain $xs'\ x$ **where** $xs': xs = xs' @ [x]$
by (*metis path-from-toE Suc.prem(1) append-butlast-last-id*)
then have $n = length\ xs'$ **using** xs' **using** *Suc.hyps(2)* **by** *auto*
moreover have $xs'\text{-path}: v0 \rightsquigarrow xs' \rightsquigarrow last\ xs'$
using xs' *Suc.prem(1)* $\langle tl\ xs \neq Nil \rangle$ *walk-decomp(1)*
by (*metis distinct-append hd-append list.sel(3) path-from-to-def self-append-conv2*)
moreover have $tl\ xs' \neq []$ **using** $\langle n \neq \text{Suc } 0 \rangle$
by (*metis path-from-toE Nitpick.size-list-simp(2) calculation(1,2)*)
moreover have $v1 \notin set\ xs'$ **using** xs' *Suc.prem(3)* **by** *auto*
moreover have $\text{new-last} \notin set\ xs'$ **using** xs' *Suc.prem(4)* **by** *auto*
moreover have $hd\ (tl\ xs') = hd\ (tl\ P)$
using xs' $\langle tl\ xs' \neq [] \rangle$ *Suc.prem(5)* *calculation(2)* **by** *auto*
moreover have $edges\text{-of-walk}\ xs' \subseteq \bigcup (edges\text{-of-walk}\ 'paths\text{-with-new})$
using xs' *Suc.prem(6)* *edges-of-comp1* **by** *blast*
ultimately have $xs'\text{-edges}: edges\text{-of-walk}\ xs' \subseteq edges\text{-of-walk}\ P$ **using** *Suc.hyps(1)* **by** *blast*
moreover have $edges\text{-of-walk}\ xs = edges\text{-of-walk}\ xs' \cup \{(last\ xs', x)\}$

```

using  $xs'$  using walk-edges-decomp [of butlast  $xs'$  last  $xs'$   $x$  Nil]  $xs'$ -path
by (metis path-from-toE Un-empty-right append-assoc append-butlast-last-id butlast.simps(2)
    edges-of-walk-empty(2) last-ConsL last-ConsR list.distinct(1))
moreover have  $(last\ xs', x) \in edges-of-walk\ P$  proof (rule ccontr)
assume contra:  $(last\ xs', x) \notin edges-of-walk\ P$ 
have  $xs\text{-last-edge}$ :  $(last\ xs', x) \in edges-of-walk\ xs$ 
    using  $xs'$  calculation(2) by blast
then obtain  $P'$  where
     $P'$ :  $P' \in paths-with-new\ (last\ xs', x) \in edges-of-walk\ P'$ 
    using Suc.prems(6) by auto
then have  $P \neq P'$  using contra by blast
moreover have  $last\ xs' \in set\ P$  using  $xs\text{-last-edge}$   $xs'\text{-edges}$   $\langle tl\ xs' \neq [] \rangle$   $xs'$ -path
    by (metis path-from-toE last-in-set subsetCE walk-edges-subset)
moreover have  $last\ xs' \in set\ P'$  using  $P'$ (2) by (meson walk-edges-vertices(1))
ultimately have  $last\ xs' = v0 \vee last\ xs' = v1 \vee last\ xs' = new\text{-last}$ 
    using paths-plus-one-disjoint  $P'$ (1)  $P$  paths-with-new-def by auto
then show False using Suc.prems(3)  $\langle new\text{-last} \notin set\ xs' \rangle$   $\langle tl\ xs' \neq [] \rangle$   $xs'$   $xs'$ -path
    by (metis path-from-toE butlast-snoc in-set-butlastD last-in-set last-tl path-from-to-first)
qed
ultimately show ?thesis by simp
qed
qed
end — locale DisjointPathsPlusOne
end

```

8 Induction of Menger's Theorem

theory *MengerInduction* **imports** *Separations DisjointPaths* **begin**

8.1 No Small Separations

In this section we set up the general structure of the proof of Menger's Theorem. The proof is based on induction over *sep-size* (called n in McCuaig's proof), the minimum size of a separator.

```

locale NoSmallSeparationsInduct = v0-v1-Digraph +
  fixes  $sep\text{-size} :: nat$ 
  — The size of a minimum separator.
assumes no-small-separations:  $\bigwedge S. Separation\ G\ v0\ v1\ S \implies card\ S \geq Suc\ sep\text{-size}$ 
  — The induction hypothesis.
and no-small-separations-hyp:  $\bigwedge G' :: ('a, 'b)\ Graph\ scheme.$ 
     $(\bigwedge S. Separation\ G'\ v0\ v1\ S \implies card\ S \geq sep\text{-size})$ 
     $\implies v0\text{-}v1\text{-}Digraph\ G'\ v0\ v1$ 
     $\implies \exists paths. DisjointPaths\ G'\ v0\ v1\ paths \wedge card\ paths = sep\text{-size}$ 

```

Next, we want to combine this with *DisjointPathsPlusOne*.

If a minimum separator has size at least $Suc\ sep\text{-size}$, then it follows immediately from the induction hypothesis that we have $sep\text{-size}$ many disjoint paths. We then observe that

second-vertices of these paths is not a separator because $\text{card } \textit{second-vertices} = \textit{sep-size}$. So there exists a new path from $v0$ to $v1$ whose second vertex is not in *second-vertices*.

If this path is disjoint from the other paths, we have found $\textit{Sep-size}$ many disjoint paths, so assume it is not disjoint. Then there exist a vertex x on the new path that is not $v0$ or $v1$ such that *new-last* hits one of the other paths. Let *P-new* be the initial segment of the new path up to x . We call x , the last vertex of *P-new*, now *new-last*.

We then assume that *paths* and *P-new* have been chosen in such a way that $\text{distance } \textit{new-last } v1$ is minimal.

First, we define a locale that expresses that we have no small separators (with the corresponding induction hypothesis) as well as $\textit{sep-size}$ many internally vertex-disjoint paths (with $\textit{sep-size} \neq 0$ because the other case is trivial) and also one additional path that starts in $v1$, whose second vertex is not among *second-vertices* and whose last vertex is *new-last*.

We will add the assumption $\textit{new-last} \neq v1$ soon.

```

locale ProofStepInduct =
  NoSmallSeparationsInduct G v0 v1 sep-size + DisjointPathsPlusOne G v0 v1 paths P-new
  for G (structure) and v0 v1 paths P-new sep-size +
  assumes sep-size-not0: sep-size  $\neq$  0
  and paths-sep-size: card paths = sep-size

```

```

lemma (in ProofStepInduct) hitting-paths-v1: hitting-paths v1
  unfolding hitting-paths-def using paths v0-neq-v1 by force

```

8.2 Choosing Paths Avoiding *new_last*

Let us now consider only the non-trivial case that $\textit{new-last} \neq v1$.

```

locale ProofStepInduct-NonTrivial = ProofStepInduct +
  assumes new-last-neq-v1:  $\textit{new-last} \neq v1$ 
begin

```

The next step is the observation that in the graph *remove-vertex new-last*, which we called *H-x*, there are also $\textit{sep-size}$ many internally vertex-disjoint paths, again by the induction hypothesis.

```

lemma Q-exists:  $\exists Q. \text{DisjointPaths } H-x v0 v1 Q \wedge \text{card } Q = \textit{sep-size}$ 

```

```

proof–
  have  $\bigwedge S. \text{Separation } H-x v0 v1 S \implies \text{card } S \geq \textit{sep-size}$ 
  using subgraph-separation-min-size paths walk-in-V P-hit new-last-neq-v1 no-small-separations
  by (metis H-x-def new-last-in-V new-last-neq-v0)
  then show ?thesis using H-x-v0-v1-Digraph new-last-neq-v1 by (meson no-small-separations-hyp)
qed

```

We want to choose these paths in a clever way, too. Our goal is to choose these paths such that the number of edges in $\bigcup (\textit{edges-of-walk } ' Q) \cap (E - \bigcup (\textit{edges-of-walk } ' \textit{paths-with-new}))$ is minimal.

```

definition B where B  $\equiv E - \bigcup (\textit{edges-of-walk } ' \textit{paths-with-new})$ 

```

```

definition Q-weight where Q-weight  $\equiv \lambda Q. \text{card } (\bigcup (\textit{edges-of-walk } ' Q) \cap B)$ 

```

definition *Q-good* **where** $Q\text{-good} \equiv \lambda Q. \text{DisjointPaths } H\text{-}x \ v0 \ v1 \ Q \wedge \text{card } Q = \text{sep-size} \wedge$
 $(\forall Q'. \text{DisjointPaths } H\text{-}x \ v0 \ v1 \ Q' \wedge \text{card } Q' = \text{sep-size} \longrightarrow Q\text{-weight } Q \leq Q\text{-weight } Q')$

definition *Q* **where** $Q \equiv \text{SOME } Q. Q\text{-good } Q$

It is easy to show that such a *Q* exists.

lemma *Q*: $\text{DisjointPaths } H\text{-}x \ v0 \ v1 \ Q \ \text{card } Q = \text{sep-size}$

and *Q-min*: $\bigwedge Q'. \text{DisjointPaths } H\text{-}x \ v0 \ v1 \ Q' \wedge \text{card } Q' = \text{sep-size} \implies Q\text{-weight } Q \leq Q\text{-weight } Q'$

proof–

obtain *Q'* **where** $\text{DisjointPaths } H\text{-}x \ v0 \ v1 \ Q' \ \text{card } Q' = \text{sep-size}$

$\bigwedge Q''. \text{DisjointPaths } H\text{-}x \ v0 \ v1 \ Q'' \wedge \text{card } Q'' = \text{sep-size} \implies Q\text{-weight } Q' \leq Q\text{-weight } Q''$

using *arg-min-ex*[of $\lambda Q. \text{DisjointPaths } H\text{-}x \ v0 \ v1 \ Q \wedge \text{card } Q = \text{sep-size} \ Q\text{-weight}$]

new-last-neq-v1 *Q-exists* **by** *metis*

then have *Q-good* *Q'* **unfolding** *Q-good-def* **by** *blast*

then show $\text{DisjointPaths } H\text{-}x \ v0 \ v1 \ Q \ \text{card } Q = \text{sep-size}$

$\bigwedge Q'. \text{DisjointPaths } H\text{-}x \ v0 \ v1 \ Q' \wedge \text{card } Q' = \text{sep-size} \implies Q\text{-weight } Q \leq Q\text{-weight } Q'$

using *someI*[of *Q-good*] **by** (*simp-all* *add*: *Q-def* *Q-good-def*)

qed

sublocale *Q*: $\text{DisjointPaths } H\text{-}x \ v0 \ v1 \ Q$ **using** *Q(1)* .

8.3 Finding a Path Avoiding *Q*

Because *Q* contains only *sep-size* many paths, we have $\text{card } Q.\text{second-vertices} = \text{sep-size}$. So there exists a path *P-k* among the *Suc sep-size* many paths in *paths-with-new* such that the second vertex of *P-k* is not among *Q.second-vertices*.

definition *P-k* **where**

$P\text{-}k \equiv \text{SOME } P\text{-}k. P\text{-}k \in \text{paths-with-new} \wedge \text{hd } (\text{tl } P\text{-}k) \notin Q.\text{second-vertices}$

lemma *P-k*: $P\text{-}k \in \text{paths-with-new} \ \text{hd } (\text{tl } P\text{-}k) \notin Q.\text{second-vertices}$ **proof**–

obtain *y* **where** $y \in \text{insert } (\text{hd } (\text{tl } P\text{-}k)) \ \text{second-vertices} \ y \notin Q.\text{second-vertices}$ **proof**–

have $\text{hd } (\text{tl } P\text{-}k) \notin \text{second-vertices}$ **using** *P-new-decomp* *tl-P-new(2)* **by** *simp*

moreover have $\text{card } \text{second-vertices} = \text{card } Q.\text{second-vertices}$ **using** *Q(2)* *paths-sep-size*

using *Q.second-vertices-card* *second-vertices-card* **by** (*simp* *add*: *new-last-neq-v1*)

ultimately have $\text{card } (\text{insert } (\text{hd } (\text{tl } P\text{-}k)) \ \text{second-vertices}) = \text{Suc } (\text{card } Q.\text{second-vertices})$

using *finite-paths* *second-vertices-def* **by** *auto*

then show *?thesis*

using *that* *card-finite-less-ex*

by (*metis* *Q.finite-paths* *Q.second-vertices-def* *Zero-not-Suc* *card.infinite* *finite-imageI* *lessI*)

qed

then have $\exists P\text{-}k. P\text{-}k \in \text{paths-with-new} \wedge \text{hd } (\text{tl } P\text{-}k) \notin Q.\text{second-vertices}$

by (*metis* (*mono-tags*, *lifting*) *image-iff* *insertCI* *insertE* *paths-with-new-def* *second-vertex-def* *second-vertices-def*)

then show $P\text{-}k \in \text{paths-with-new} \ \text{hd } (\text{tl } P\text{-}k) \notin Q.\text{second-vertices}$

using *someI*[of $\lambda P\text{-}k. P\text{-}k \in \text{paths-with-new} \wedge \text{hd } (\text{tl } P\text{-}k) \notin Q.\text{second-vertices}$] *P-k-def* **by** *auto*

qed

lemma *path-P-k* [*simp*]: *path* *P-k* **by** (*simp* *add*: *P-k(1)* *paths-with-new-path*)

lemma *hd-P-k-v0* [*simp*]: $\text{hd } P\text{-}k = v0$ **by** (*simp* *add*: *P-k(1)* *paths-with-new-start-in-v0*)

definition *hitting-Q-or-new-last* **where**

hitting-Q-or-new-last $\equiv \lambda y. y \neq v0 \wedge (y = \text{new-last} \vee (\exists Q\text{-hit} \in Q. y \in \text{set } Q\text{-hit}))$

P-k hits a vertex in *Q* or it hits *new-last* because it either ends in *v1* or in *new-last*.

lemma *P-k-hits-Q*: $\exists y \in \text{set } P\text{-k}. \text{hitting-Q-or-new-last } y$ **proof** (*cases*)

assume *P-k* \neq *P-new*

then have *v1* \in *set P-k*

by (*metis P-k(1) insertE last-in-set path-from-toE paths paths-with-new-def*)

moreover have $\exists Q\text{-witness}. Q\text{-witness} \in Q$ **using** *Q(2) sep-size-not0 finite.simps* **by** *fastforce*

ultimately show *?thesis*

using *Q.paths path-from-toE hitting-Q-or-new-last-def v0-neq-v1* **by** *fastforce*

qed (*metis P-new new-last-neq-v0 hitting-Q-or-new-last-def last-in-set path-from-toE new-last-def*)

end — locale *ProofStepInduct-NonTrivial*

8.4 Decomposing P_k

Having established with the previous lemma that *P-k* hits *Q* or *new-last*, let *y* be the first such vertex on *P-k*. Then we can split *P-k* at this vertex.

locale *ProofStepInduct-NonTrivial-P-k-pre* = *ProofStepInduct-NonTrivial* +

fixes *P-k-pre* *y* *P-k-post*

assumes *P-k-decomp*: *P-k* = *P-k-pre* @ *y* # *P-k-post*

and *y*: *hitting-Q-or-new-last y*

and *y-min*: $\bigwedge y'. y' \in \text{set } P\text{-k-pre} \implies \neg \text{hitting-Q-or-new-last } y'$

We can always go from *ProofStepInduct-NonTrivial* to *ProofStepInduct-NonTrivial-P-k-pre*.

lemma (**in** *ProofStepInduct-NonTrivial*) *ProofStepInduct-NonTrivial-P-k-pre-exists*:

shows $\exists P\text{-k-pre } y P\text{-k-post}$.

ProofStepInduct-NonTrivial-P-k-pre G v0 v1 paths P-new sep-size P-k-pre y P-k-post

proof –

obtain *y P-k-pre P-k-post* **where**

P-k = *P-k-pre* @ *y* # *P-k-post* *hitting-Q-or-new-last y*

$\bigwedge y'. y' \in \text{set } P\text{-k-pre} \implies \neg \text{hitting-Q-or-new-last } y'$

using *P-k-hits-Q split-list-first-prop[of P-k hitting-Q-or-new-last]* **by** *blast*

then have *ProofStepInduct-NonTrivial-P-k-pre G v0 v1 paths P-new sep-size P-k-pre y P-k-post*

by *unfold-locales*

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

qed

context *ProofStepInduct-NonTrivial-P-k-pre* **begin**

lemma *y-neq-v0*: *y* \neq *v0* **using** *hitting-Q-or-new-last-def y* **by** *auto*

lemma *P-k-pre-not-Nil*: *P-k-pre* \neq *Nil*

using *P-k-decomp hd-P-k-v0 hitting-Q-or-new-last-def y* **by** *auto*

lemma *second-P-k-pre-not-in-Q*: *hd (tl (P-k-pre @ [y]))* \notin *Q.second-vertices*

using *P-k(2) P-k-decomp P-k-pre-not-Nil*

by (*metis append-eq-append-conv2 append-self-conv hd-append2 list.sel(1) tl-append2*)

definition *H* **where** *H* \equiv *remove-vertex v0*

sublocale H : *Digraph H unfolding H-def using remove-vertex-Digraph* .

lemma $y\text{-eq-}v1\text{-implies-}P\text{-k-neq-}P\text{-new}$: **assumes** $y = v1$ **shows** $P\text{-k} \neq P\text{-new}$ **proof**
assume *contra*: $P\text{-k} = P\text{-new}$
have $v0 \rightsquigarrow_{(new\text{-pre} \text{ @ } [new\text{-last}])} \rightsquigarrow new\text{-last}$
using $P\text{-new}(1)$ $P\text{-new-decomp}$ $new\text{-last-def}$ **by** *auto*
then have $v0 \rightsquigarrow_{P\text{-k}} \rightsquigarrow new\text{-last}$ **using** $P\text{-new-decomp}$ *contra* **by** *auto*
moreover have $P\text{-k} = P\text{-k-pre} \text{ @ } v1 \# P\text{-k-post}$ **using** $P\text{-k-decomp}$ $assms(1)$ **by** *blast*
ultimately have $v0 \rightsquigarrow_{(P\text{-k-pre} \text{ @ } v1 \# P\text{-k-post})} \rightsquigarrow new\text{-last}$ **by** *simp*
then have $v1 \in set\ P\text{-new}$ **by** (*metis* $assms\ contra\ P\text{-k-decomp}\ in\text{-set-conv-decomp}$)
then have $new\text{-last} = v1$
using $hitting\text{-paths-}v1\ assms\ last\text{-}P\text{-new}(2)$ $set\text{-butlast}$ $new\text{-last-def}$ **by** *fastforce*
then show *False* **using** $new\text{-last-neq-}v1$ **by** *blast*
qed

If $y = v1$, then we are done.

lemma $y\text{-eq-}v1\text{-solves}$:
assumes $y = v1$
shows $\exists paths. DisjointPaths\ G\ v0\ v1\ paths \wedge card\ paths = Suc\ sep\text{-size}$
proof–
have $P\text{-k} \neq P\text{-new}$ **using** $y\text{-eq-}v1\text{-implies-}P\text{-k-neq-}P\text{-new}$ $assms$ **by** *blast*
then have $P\text{-k} = P\text{-k-pre} \text{ @ } [y]$
using $P\text{-k}(1)$ $P\text{-k-decomp}$ $paths\ assms\ paths\text{-with-}new\text{-def}$ **by** *fastforce*
then have $v0 \rightsquigarrow_{(P\text{-k-pre} \text{ @ } [y])} \rightsquigarrow v1$
using $paths\ P\text{-k}(1)$ $\langle P\text{-k} \neq P\text{-new} \rangle$ **by** (*simp* *add*: $paths\text{-with-}new\text{-def}$)
moreover have $new\text{-last} \notin set\ P\text{-k-pre}$
using $hitting\text{-}Q\text{-or-}new\text{-last-def}$ $y\text{-min}$ $new\text{-last-neq-}v0$ **by** *auto*
ultimately have $v0 \rightsquigarrow_{(P\text{-k-pre} \text{ @ } [y])} \rightsquigarrow_{H\text{-}x} v1$ **using** $remove\text{-vertex-path-from-to}$
by (*simp* *add*: $H\text{-}x\text{-def}$ $assms\ new\text{-last-in-}V\ new\text{-last-neq-}v1$)
moreover {
fix $xs\ v$ **assume** $xs \in Q\ v \in set\ xs\ v \in set\ (P\text{-k-pre} \text{ @ } [y])\ v \neq v0\ v \neq v1$
then have $v \in set\ P\text{-k-pre}$ **using** $assms$ **by** *simp*
then have $\neg hitting\text{-}Q\text{-or-}new\text{-last}\ v$ **using** $y\text{-min}$ **by** *blast*
then have *False* **using** $\langle v \in set\ xs \rangle\ \langle xs \in Q \rangle\ hitting\text{-}Q\text{-or-}new\text{-last-def}\ \langle v \neq v0 \rangle$ **by** *auto*
}
ultimately have $DisjointPaths\ H\text{-}x\ v0\ v1\ (insert\ (P\text{-k-pre} \text{ @ } [y])\ Q)$
using $Q.DisjointPaths\text{-extend}$ **by** *blast*
then have $DisjointPaths\ G\ v0\ v1\ (insert\ (P\text{-k-pre} \text{ @ } [y])\ Q)$
using $DisjointPaths\text{-supergraph}\ H\text{-}x\text{-def}\ new\text{-last-in-}V\ new\text{-last-neq-}v0\ new\text{-last-neq-}v1$ **by** *auto*
moreover have $card\ (insert\ (P\text{-k-pre} \text{ @ } [y])\ Q) = Suc\ sep\text{-size}$ **proof**–
have $P\text{-k-pre} \text{ @ } [y] \notin Q$
by (*metis* $P\text{-k}(2)$ $Q.second\text{-vertices-def}\ \langle P\text{-k} = P\text{-k-pre} \text{ @ } [y] \rangle\ image\text{-iff}\ second\text{-vertex-def}$)
then show $?thesis$ **by** (*simp* *add*: $Q(2)\ Q.finite\text{-paths}$)
qed
ultimately show $?thesis$ **by** *blast*
qed
end — locale $ProofStepInduct\ NonTrivial\ P\text{-k-pre}$
end

9 The case $y = \text{new_last}$

theory *Y-eq-new-last* **imports** *MengerInduction* **begin**

We may assume $y \neq v1$ now because $\llbracket \text{ProofStepInduct-NonTrivial-P-k-pre } ?G \text{ } ?v0.0 \text{ } ?v1.0 \text{ } ?paths \text{ } ?P\text{-new} \text{ } ?sep\text{-size} \text{ } ?P\text{-k-pre} \text{ } ?y \text{ } ?P\text{-k-post}; ?y = ?v1.0 \rrbracket \implies \exists paths. \text{DisjointPaths } ?G \text{ } ?v0.0 \text{ } ?v1.0 \text{ } paths \wedge \text{card } paths = \text{Suc } ?sep\text{-size}$ shows that $y = v1$ already gives us *Suc sep-size* many disjoint paths.

We also assume that we have chosen the previous paths optimally in the sense that the distance from *new-last* to *v1* is minimal.

locale *ProofStepInduct-y-eq-new-last* = *ProofStepInduct-NonTrivial-P-k-pre* +
assumes *y-neq-v1*: $y \neq v1$ **and** *y-eq-new-last*: $y = \text{new_last}$
and *optimal-paths*: $\bigwedge paths' P\text{-new}'.$
ProofStepInduct *G* *v0* *v1* *paths'* *P-new'* *sep-size*
 $\implies H.\text{distance } (\text{last } P\text{-new}) \text{ } v1 \leq H.\text{distance } (\text{last } P\text{-new}') \text{ } v1$

begin

Let *R* be a shortest path from *new-last* to *v1*.

definition *R* **where** $R \equiv$

SOME *R*. $\text{new_last} \rightsquigarrow_{R \rightsquigarrow_H} v1 \wedge (\forall R'. \text{new_last} \rightsquigarrow_{R' \rightsquigarrow_H} v1 \longrightarrow \text{length } R \leq \text{length } R')$

lemma *R*: $\text{new_last} \rightsquigarrow_{R \rightsquigarrow_H} v1 \wedge R'. \text{new_last} \rightsquigarrow_{R' \rightsquigarrow_H} v1 \implies \text{length } R \leq \text{length } R'$ **proof**–
obtain *R'* **where**

R': $\text{new_last} \rightsquigarrow_{R' \rightsquigarrow_H} v1 \wedge R''. \text{new_last} \rightsquigarrow_{R'' \rightsquigarrow_H} v1 \implies \text{length } R' \leq \text{length } R''$

using *arg-min-ex*[*OF* *new-last-to-v1*] **unfolding** *H-def* **by** *blast*

then show $\text{new_last} \rightsquigarrow_{R \rightsquigarrow_H} v1 \wedge R'. \text{new_last} \rightsquigarrow_{R' \rightsquigarrow_H} v1 \implies \text{length } R \leq \text{length } R'$

using *someI*[*of* $\lambda R. \text{new_last} \rightsquigarrow_{R \rightsquigarrow_H} v1 \wedge (\forall R'. \text{new_last} \rightsquigarrow_{R' \rightsquigarrow_H} v1 \longrightarrow \text{length } R \leq \text{length } R')$]

R-def **by** *auto*

qed

lemma *v1-in-Q*: $\exists Q\text{-hit} \in Q. v1 \in \text{set } Q\text{-hit}$ **proof**–

obtain *xs* **where** $xs \in Q$ **using** *Q*(*?*) *sep-size-not0* **by** *fastforce*

then show *?thesis* **using** *Q.paths* *last-in-set* **by** *blast*

qed

lemma *R-hits-Q*: $\exists z \in \text{set } R. Q.\text{hitting-paths } z$ **proof**–

have $v1 \in \text{set } R$ **using** *R*(1) *last-in-set* **by** (*metis* *path-from-to-def*)

then show *?thesis* **unfolding** *Q.hitting-paths-def* **using** *v0-neq-v1* **by** *auto*

qed

lemma *R-decomp-exists*:

obtains *R-pre* *z* *R-post*

where $R = R\text{-pre} \ @ \ z \ \# \ R\text{-post}$

and *Q.hitting-paths* *z*

and $\bigwedge z'. z' \in \text{set } R\text{-pre} \implies \neg Q.\text{hitting-paths } z'$

using *R-hits-Q* *split-list-first-prop*[*of* *R* *Q.hitting-paths*] **by** *blast*

We open an anonymous context in order to hide all but the final lemma. This also gives us the decomposition of *R* whose existence we established above.

```

context fixes  $R\text{-pre } z R\text{-post}$ 
assumes  $R\text{-decomp}: R = R\text{-pre} @ z \# R\text{-post}$ 
and  $z: Q.\text{hitting-paths } z$ 
and  $z\text{-min}: \bigwedge z'. z' \in \text{set } R\text{-pre} \implies \neg Q.\text{hitting-paths } z'$ 
begin
private lemma  $z\text{-neq-}v0: z \neq v0$  using  $z Q.\text{hitting-paths-def}$  by auto

private lemma  $z\text{-neq-new-last}: z \neq \text{new-last}$  proof
assume  $z = \text{new-last}$ 
then obtain  $Q\text{-hit}$  where  $Q\text{-hit}: Q\text{-hit} \in Q \text{ new-last} \in \text{set } Q\text{-hit}$ 
using  $z Q.\text{hitting-paths-def } y\text{-eq-new-last } y\text{-neq-}v1$  by auto
then have  $Q.\text{path } Q\text{-hit}$  by (meson  $Q.\text{paths path-from-to-def}$ )
then have  $\text{set } Q\text{-hit} \subseteq V - \{\text{new-last}\}$  using  $Q.\text{walk-in-}V H\text{-x-def remove-vertex-}V$  by simp
then show False using  $Q\text{-hit}(2)$  by blast
qed

private lemma  $R\text{-pre-neq-}Nil: R\text{-pre} \neq Nil$  using  $z\text{-neq-new-last } R\text{-decomp } R(1)$  by auto

private lemma  $z\text{-closer-than-new-last}: H.\text{distance } z v1 < H.\text{distance } \text{new-last } v1$  proof–
have  $H.\text{distance } \text{new-last } v1 = \text{length } R$  using  $H.\text{distance-witness } R$  by auto
moreover have  $z \rightsquigarrow (z \# R\text{-post}) \rightsquigarrow_H v1$  using  $R\text{-decomp } R(1)$ 
by (metis  $H.\text{walk-decomp}(2) \text{ distinct-append last-append}R \text{ list.sel}(1)$ 
 $\text{list.simps}(3) \text{ path-from-to-def}$ )
moreover have  $\text{length } R > \text{length } (z \# R\text{-post})$ 
unfolding  $R\text{-decomp}$  using  $R\text{-pre-neq-}Nil$  by simp
ultimately show ?thesis using  $H.\text{distance-upper-bound}$  by fastforce
qed

private definition  $R'\text{-walk}$  where  $R'\text{-walk} \equiv P\text{-k-pre} @ R\text{-pre} @ [z]$ 

private lemma  $R'\text{-walk-not-}Nil: R'\text{-walk} \neq Nil$  using  $R'\text{-walk-def } R(1)$  by simp

private lemma  $R'\text{-walk-no-}Q: \llbracket v \in \text{set } R'\text{-walk}; v \neq z \rrbracket \implies \neg Q.\text{hitting-paths } v$  proof–
fix  $v$  assume  $v \in \text{set } R'\text{-walk } v \neq z$ 
moreover have  $v \in \text{set } P\text{-k-pre} \implies \neg Q.\text{hitting-paths } v$ 
using  $Q.\text{hitting-paths-def hitting-}Q\text{-or-new-last-def } y\text{-min } v1\text{-in-}Q$  by auto
moreover have  $v \in \text{set } R\text{-pre} \implies \neg Q.\text{hitting-paths } v$  using  $z\text{-min}$  by simp
ultimately show  $\neg Q.\text{hitting-paths } v$  unfolding  $R'\text{-walk-def}$  using  $R'\text{-walk-def}$  by auto
qed

```

The original proof goes like this: “Let z be the first vertex of R on some path in Q . Then the distance in H from z to $v1$ is less than the distance from new-last to $v1$. This contradicts the choice of paths and $P\text{-new}$.”

It does not say exactly why it contradicts the choice of paths and $P\text{-new}$. It seems we can choose Q together with $R'\text{-walk}$ as our new paths plus extrapath. But this seems to be wrong because we cannot show that $R'\text{-walk}$ is a path: $P\text{-k-pre}$ and $R\text{-pre}$ could intersect. So we use $\llbracket \text{walk } ?xs; ?xs \neq []; \text{hd } ?xs = ?v; \text{last } ?xs = ?w \rrbracket \implies \exists ys. ?v \rightsquigarrow ys \rightsquigarrow ?w \wedge \text{set } ys \subseteq \text{set } ?xs$ to transform $R'\text{-walk}$ into a path R' .

```

private definition  $R'$  where
 $R' \equiv \text{SOME } R'. \text{hd } (\text{tl } R'\text{-walk}) \rightsquigarrow_{R'} z \wedge \text{set } R' \subseteq \text{set } (\text{tl } R'\text{-walk})$ 

```

private lemma R' : $hd (tl R'\text{-walk}) \rightsquigarrow R' \rightsquigarrow z$ $set R' \subseteq set (tl R'\text{-walk})$ **proof**–
have $tl R'\text{-walk} \neq Nil$ **by** (*simp add: P-k-pre-not-Nil R'\text{-walk-def}*)
moreover have $last R'\text{-walk} = z$ **unfolding** $R'\text{-walk-def}$ **by** *simp*
moreover have $walk (tl R'\text{-walk})$
by (*metis (no-types, lifting) path-from-toE walk-tl H-def P-k-decomp R'\text{-walk-def R(1) R-decomp path-P-k y-eq-new-last hd-append list.sel(1) list.simps(3) path-decomp' remove-vertex-path-from-to-add walk-comp walk-decomp(1) walk-last-edge*)
ultimately obtain R'' **where** $hd (tl R'\text{-walk}) \rightsquigarrow R'' \rightsquigarrow z$ $set R'' \subseteq set (tl R'\text{-walk})$
using $walk\text{-to-path}[of\ tl\ R'\text{-walk}\ hd\ (tl\ R'\text{-walk})\ z]$ $last\text{-tl}$ **by** *force*
then show $hd (tl R'\text{-walk}) \rightsquigarrow R' \rightsquigarrow z$ $set R' \subseteq set (tl R'\text{-walk})$ **unfolding** $R'\text{-def}$
using $someI[of\ \lambda R'.\ hd\ (tl\ R'\text{-walk})\ \rightsquigarrow R' \rightsquigarrow z \wedge set\ R' \subseteq set\ (tl\ R'\text{-walk})]$ **by** *auto*
qed

private lemma $hd\text{-}R'$: $hd R' = hd (tl P\text{-}k)$ **proof**–
have $hd (tl R'\text{-walk}) = hd (tl P\text{-}k)$ **proof** (*cases*)
assume $tl P\text{-}k\text{-pre} = Nil$
then show *?thesis* **unfolding** $R'\text{-walk-def}$ **using** $P\text{-}k\text{-decomp}\ R(1)$ $P\text{-}k\text{-pre-not-Nil}$ $y\text{-eq-new-last}$
by (*metis H.path-from-toE R-decomp hd-append list.sel(1) tl-append2*)
next
assume $tl P\text{-}k\text{-pre} \neq Nil$
then show *?thesis* **unfolding** $R'\text{-walk-def}$ **using** $P\text{-}k\text{-pre-not-Nil}$ **by** (*simp add: P-k-decomp*)
qed
then show *?thesis* **using** $R'(1)$ **by** *auto*
qed

private lemma $R'\text{-no-Q}$: $\llbracket v \in set R'; v \neq z \rrbracket \implies \neg Q.hitting\text{-paths}\ v$
using $R'\text{-walk-no-Q}$ **by** (*meson R'(2) R'\text{-walk-not-Nil list.set-sel(2) subsetCE*)

private lemma $v0\text{-}R'\text{-path}$: $v0 \rightsquigarrow (v0 \# R') \rightsquigarrow z$ **proof**–
have $v0 \rightarrow hd R'$ **using** $hd\text{-}R'$ $hd\text{-}P\text{-}k\text{-}v0$
by (*metis Nil-is-append-conv P-k-decomp P-k-pre-not-Nil path-P-k list.distinct(1) list.exhaust-sel path-first-edge' tl-append2*)
moreover have $v0 \notin set R'$ **proof**–
have $v0 \notin set R$ **using** $R(1)$ $H\text{-def}$ $H.path\text{-in-}V$ $remove\text{-vertex-}V$
by (*simp add: path-from-to-def subset-Diff-insert*)
then have $v0 \notin set R\text{-pre}$ **using** $R\text{-decomp}$ **by** *simp*
moreover have $v0 \notin set (tl P\text{-}k\text{-pre})$ **using** $hd\text{-}P\text{-}k\text{-}v0$ $path\text{-}P\text{-}k$ $path\text{-}first\text{-}vertex$
by (*metis P-k-decomp P-k-pre-not-Nil hd-append list.exhaust-sel path-decomp(1)*)
ultimately show *?thesis* **using** $R'(2)$ **unfolding** $R'\text{-walk-def}$
using $P\text{-}k\text{-pre-not-Nil}$ $z\text{-neq-}v0$ **by** *auto*
qed
ultimately show *?thesis* **using** $path\text{-cons}$
by (*metis R'(1) last.simps list.sel(1) list.simps(3) path-from-to-def*)
qed

private corollary $z\text{-last-}R'$: $z = last (v0 \# R')$ **using** $v0\text{-}R'\text{-path}$ **by** *auto*

private lemma $z\text{-eq-}v1\text{-solves}$:
assumes $z = v1$
shows $\exists paths. DisjointPaths\ G\ v0\ v1\ paths \wedge card\ paths = Suc\ sep\text{-size}$
proof–

```

interpret Q': DisjointPaths G v0 v1 Q
  using DisjointPaths-supergraph H-x-def Q.DisjointPaths-axioms by auto
have v0  $\rightsquigarrow$  (v0 # R')  $\rightsquigarrow$  v1 using assms v0-R'-path by auto
moreover {
  fix xs v assume xs  $\in$  Q xs  $\neq$  v0 # R' v  $\in$  set xs v  $\in$  set (v0 # R')
  then have v = v0  $\vee$  v = v1 using R'-no-Q Q.hitting-paths-def  $\langle z = v1 \rangle$  by auto
}
ultimately have DisjointPaths G v0 v1 (insert (v0 # R') Q)
  using Q'.DisjointPaths-extend by blast
moreover have card (insert (v0 # R') Q) = Suc sep-size
  by (simp add: P-k(2) Q(2) Q.finite-paths Q.second-vertices-new-path hd-R')
ultimately show ?thesis by blast
qed

private lemma z-neq-v1-solves:
  assumes z  $\neq$  v1
  shows  $\exists$  paths. DisjointPaths G v0 v1 paths  $\wedge$  card paths = Suc sep-size
proof -
  have ProofStepInduct G v0 v1 Q (v0 # R') sep-size proof (rule ProofStepInduct.intro)
  show DisjointPathsPlusOne G v0 v1 Q (v0 # R') proof (rule DisjointPathsPlusOne.intro)
  show DisjointPaths G v0 v1 Q
    using DisjointPaths-supergraph H-x-def Q.DisjointPaths-axioms by auto
  show DisjointPathsPlusOne-axioms G v0 v1 Q (v0 # R') proof
  show v0  $\rightsquigarrow$  (v0 # R')  $\rightsquigarrow$  last (v0 # R') using v0-R'-path by blast
  show tl (v0 # R')  $\neq$  [] using R'(1) by auto
  show hd (tl (v0 # R'))  $\notin$  Q.second-vertices using hd-R' P-k(2) by auto
  show Q.hitting-paths (last (v0 # R')) using z z-last-R' by auto
  next
  fix v assume v  $\in$  set (butlast (v0 # R'))
  then show  $\neg$  Q.hitting-paths v using R'-no-Q path-from-to-last[OF v0-R'-path]
    by (metis Q.hitting-paths-def in-set-butlastD set-ConsD)
  qed
  qed
  show ProofStepInduct-axioms Q sep-size using sep-size-not0 Q(2) by unfold-locales
  qed (insert NoSmallSeparationsInduct-axioms)
  then have H.distance (last P-new) v1  $\leq$  H.distance (last (v0 # R')) v1
    using H-def optimal-paths[of Q v0 # R'] by blast
  then have False using z-last-R' new-last-def z-closer-than-new-last by simp
  then show ?thesis by blast
qed

corollary with-optimal-paths-solves':
  shows  $\exists$  paths. DisjointPaths G v0 v1 paths  $\wedge$  card paths = Suc sep-size
  using optimal-paths z-eq-v1-solves z-neq-v1-solves by blast
end — anonymous context

corollary with-optimal-paths-solves:
   $\exists$  paths. DisjointPaths G v0 v1 paths  $\wedge$  card paths = Suc sep-size
  using optimal-paths with-optimal-paths-solves' R-decomp-exists by blast

end — locale ProofStepInduct-y-eq-new-last
end

```

10 The case $y \neq \text{new_last}$

theory *Y-neq-new-last* **imports** *MengerInduction* **begin**

Let us now consider the case that $y \neq v1 \wedge y \neq \text{new-last}$. Our goal is to show that this is inconsistent: The following locale will be unsatisfiable, proving that $y = v1 \vee y = \text{new-last}$ holds.

locale *ProofStepInduct-y-neq-new-last* = *ProofStepInduct-NonTrivial-P-k-pre* +
assumes *y-neq-v1*: $y \neq v1$ **and** *y-neq-new-last*: $y \neq \text{new-last}$
begin

lemma *Q-hit-exists*: **obtains** *Q-hit Q-hit-pre Q-hit-post* **where**
 $Q\text{-hit} \in Q$ $y \in \text{set } Q\text{-hit}$ $Q\text{-hit} = Q\text{-hit-pre} @ y \# Q\text{-hit-post}$
proof –
obtain *Q-hit* **where** $Q\text{-hit} \in Q$ $y \in \text{set } Q\text{-hit}$
using *hitting-Q-or-new-last-def y y-neq-v1 y-neq-new-last* **by** *auto*
then show *?thesis* **using** *that* **by** (*meson split-list*)
qed

We open an anonymous context because we do not want to export any lemmas except the final lemma proving the contradiction. This is also an easy way to get the decomposition of *Q-hit*, whose existence we have established above.

context
fixes *Q-hit Q-hit-pre Q-hit-post*
assumes *Q-hit*: $Q\text{-hit} \in Q$ $y \in \text{set } Q\text{-hit}$
and *Q-hit-decomp*: $Q\text{-hit} = Q\text{-hit-pre} @ y \# Q\text{-hit-post}$
begin
private lemma *Q-hit-v0-v1*: $v0 \rightsquigarrow_{Q\text{-hit}} \rightsquigarrow_{H-x} v1$ **using** *Q.paths Q-hit(1)* **by** *blast*

private lemma *Q-hit-vertices*: $\text{set } Q\text{-hit} \subseteq V - \{\text{new-last}\}$
using *Q.walk-in-V H-x-def path-from-to-def remove-vertex-V Q-hit-v0-v1* **by** *fastforce*

private lemma *Q-hit-pre-not-Nil*: $Q\text{-hit-pre} \neq \text{Nil}$
using *Q-hit-v0-v1 y-neq-v0* **unfolding** *Q-hit-decomp* **by** *auto*

private lemma *tl-Q-hit-pre*: $\text{tl } (Q\text{-hit-pre} @ [y]) \neq \text{Nil}$ **using** *Q-hit-pre-not-Nil* **by** *simp*

private lemma *Q-hit-pre-edges*: $\text{edges-of-walk } (Q\text{-hit-pre} @ [y]) \cap B \neq \{\}$ **proof**
assume $\text{edges-of-walk } (Q\text{-hit-pre} @ [y]) \cap B = \{\}$
moreover have $\text{edges-of-walk } (Q\text{-hit-pre} @ [y]) \subseteq E$
by (*metis Q.paths H-x-def Q-hit(1) Q-hit-decomp edges-of-walk-in-E path-decomp'*
path-from-to-def remove-vertex-walk-add)
ultimately have *Q-hit-pre-edges*:
 $\text{edges-of-walk } (Q\text{-hit-pre} @ [y]) \subseteq \bigcup (\text{edges-of-walk } \text{'paths-with-new'})$
unfolding *B-def* **by** *blast*
then have ***: $\text{first-edge-of-walk } (Q\text{-hit-pre} @ [y]) \in \bigcup (\text{edges-of-walk } \text{'paths-with-new'})$
using *tl-Q-hit-pre first-edge-in-edges* **by** *blast*

define *v'* **where** $v' \equiv \text{hd } (\text{tl } (Q\text{-hit-pre} @ [y]))$
then have *v'*: $(v0, v') = \text{first-edge-of-walk } (Q\text{-hit-pre} @ [y])$
using *first-edge-hd-tl Q-hit-pre-not-Nil tl-Q-hit-pre*

by (metis *Q.path-from-toE Q-hit-decomp Q-hit-v0-v1 first-edge-of-walk.simps(1)*
hd-Cons-tl hd-append snoc-eq-iff-butlast)

then obtain *P-i* where

P-i: *P-i* ∈ *paths-with-new* (*v0*, *v'*) ∈ *edges-of-walk P-i* using * by auto
then have *P-i-first*: *first-edge-of-walk P-i* = (*v0*, *v'*)
using *first-edge-first paths-with-new-def paths P-new* by (metis *insert-iff*)
moreover have *first-edge-of-walk P-k* = (*v0*, *hd (tl P-k)*)
by (metis *P-k-decomp P-k-pre-not-Nil append-is-Nil-conv first-edge-of-walk.simps(1)*
hd-P-k-v0 list.distinct(1) list.exhaust-sel tl-append2)
ultimately have *P-i* ≠ *P-k*
by (metis *Q.first-edge-first P-k(2) Q.second-vertices-first-edge Q-hit(1) Q-hit-decomp*
Q-hit-v0-v1 Un-iff v' tl-Q-hit-pre first-edge-in-edges walk-edges-decomp)

Then *P-k* and *P-i* intersect in *y*, which is not one of *v0*, *v1*, or *new-last*. So we get a contradiction because these two paths should be disjoint on all other vertices.

moreover have *v1* ∉ *set (Q-hit-pre @ [y])*
using *Q-hit-v0-v1 Q-hit-decomp y-neq-v1* by (simp add: *Q.path-from-to-last'*)
moreover have *new-last* ∉ *set (Q-hit-pre @ [y])* proof –
have *new-last* ∉ *set Q-hit-pre* using *Q-hit-decomp Q-hit-vertices* by auto
then show ?thesis using *y-neq-new-last* by auto
qed
moreover have *hd (tl (Q-hit-pre @ [y]))* = *hd (tl P-i)* proof –
have *hd (tl P-i)* = *v'* using *P-i-first P-i(1) tl-P-new(1)*
by (metis *Pair-inject first-edge-of-walk.simps(1) insert-iff list.collapse*
paths-tl-notnil paths-with-new-def tl-Nil)
then show ?thesis using *v'-def* by simp
qed
moreover have *v0* ∼ (*Q-hit-pre @ [y]*) ∼ *y*
by (metis *Q.path-decomp' H-x-def Q-hit-decomp Q-hit-v0-v1 Q-hit-pre-not-Nil*
hd-append2 path-from-to-def remove-vertex-walk-add snoc-eq-iff-butlast)
ultimately have *edges-of-walk (Q-hit-pre @ [y])* ⊆ *edges-of-walk P-i*
using *new-path-follows-old-paths tl-Q-hit-pre P-i(1) Q-hit-pre-edges* by blast
from *walk-edges-subset[OF this]* have *y* ∈ *set P-i* by (simp add: *tl-Q-hit-pre*)
moreover have *y* ∈ *set P-k* using *P-k-decomp* by auto
ultimately show *False*
using *y-neq-v0 y-neq-v1 y-neq-new-last ⟨P-i ≠ P-k⟩*
paths-plus-one-disjoint[OF P-i(1), of P-k y] P-k(1) P-new-decomp by auto
qed

private lemma *P-k-pre-edges*: *edges-of-walk (P-k-pre @ [y])* ∩ *B* = {} proof –
have *edges-of-walk (P-k-pre @ [y])* ⊆ ∪ (*edges-of-walk ' paths-with-new*)
proof (cases)
assume *P-k* = *P-new*
then have *edges-of-walk (P-k-pre @ [y])* ⊆ *edges-of-walk P-new*
using *P-k-decomp edges-of-comp1* by force
then show ?thesis unfolding *paths-with-new-def* by blast
next
assume *P-k* ≠ *P-new*
then have *P-k* ∈ *paths* using *P-k(1) paths-with-new-def* by blast
then have *edges-of-walk (P-k-pre @ [y])* ⊆ ∪ (*edges-of-walk ' paths*)

using *edges-of-comp1*[of *P-k-pre* @ *[y]*] *P-k-decomp* by *auto*
 then show *?thesis unfolding paths-with-new-def* by *blast*
 qed
 then show *?thesis unfolding B-def* by *blast*
 qed

private definition *Q-hit'* where $Q\text{-hit}' \equiv P\text{-k-pre} \ @ \ y \ \# \ Q\text{-hit-post}$

private lemma *Q-hit'-v0-v1*: $v0 \rightsquigarrow Q\text{-hit}' \rightsquigarrow v1$ **proof**–
 {
 fix *v* assume $v \in \text{set } P\text{-k-pre}$
 then have $\neg Q.\text{hitting-paths } v$ using *Q.paths Q-hit(1) y-min*
 by (*metis Q.hitting-paths-def hitting-Q-or-new-last-def last-in-set path-from-to-def*)
 moreover have $v0 \notin \text{set } Q\text{-hit-post}$ using *Q.path-from-to-first' Q-hit-v0-v1*
 unfolding *Q-hit-decomp* by *blast*
 ultimately have $v \notin \text{set } Q\text{-hit-post}$ unfolding *Q.hitting-paths-def*
 using *Q-hit(1) Q-hit-decomp* by *auto*
 }
 then have $\text{set } P\text{-k-pre} \cap \text{set } Q\text{-hit-post} = \{\}$ by *blast*
 then show *?thesis unfolding Q-hit'-def* using *path-from-to-combine*
 by (*metis H-x-def P-k-decomp P-k-pre-not-Nil Q-hit-decomp Q-hit-v0-v1 append-is-Nil-conv*
hd-P-k-v0 path-P-k path-from-toI remove-vertex-path-from-to-add)
 qed

private lemma *Q-hit'-v0-v1-H-x*: $v0 \rightsquigarrow Q\text{-hit}' \rightsquigarrow_{H-x} v1$ **proof**–
 have *new-last* $\notin \text{set } P\text{-k-pre}$ using *new-last-neq-v0 hitting-Q-or-new-last-def y-min* by *auto*
 moreover have *new-last* $\notin \text{set } Q\text{-hit-post}$ using *Q-hit-vertices unfolding Q-hit-decomp* by *auto*
 ultimately have *new-last* $\notin \text{set } Q\text{-hit}'$ using *y-neq-new-last Q-hit'-def* by *auto*
 then show *?thesis* using *remove-vertex-path-from-to[OF Q-hit'-v0-v1] H-x-def new-last-in-V*
 by *simp*
 qed

private definition *Q'* where $Q' \equiv \text{insert } Q\text{-hit}' (Q - \{Q\text{-hit}\})$

private lemma *Q-hit-edges-disjoint*:
 $\bigcup (\text{edges-of-walk } ' (Q - \{Q\text{-hit}\})) \cap \text{edges-of-walk } Q\text{-hit} = \{\}$
 using *DiffD1 Q.paths-edge-disjoint Q-hit(1)* by *fastforce*

private lemma *Q-hit'-notin-Q-minus-Q-hit*: $Q\text{-hit}' \notin Q - \{Q\text{-hit}\}$ **proof**–
 have *hd (tl Q-hit')* $\notin Q.\text{second-vertices}$ using *P-k(2) P-k-decomp*
 by (*metis P-k-pre-not-Nil Q-hit'-def append-eq-append-conv2 append-self-conv hd-append2*
list.sel(1) tl-append2)
 then show *?thesis* using *Q.second-vertices-new-path[of Q-hit']* by *blast*
 qed

private lemma *Q-weight-smaller*: $Q\text{-weight } Q' < Q\text{-weight } Q$ **proof**–
 define *Q-edges* where $Q\text{-edges} \equiv \bigcup (\text{edges-of-walk } ' Q) \cap B$
 define *Q'-edges* where $Q'\text{-edges} \equiv \bigcup (\text{edges-of-walk } ' Q') \cap B$
 {
 fix *v w* assume $*$: $(v,w) \in Q'\text{-edges}$ $(v,w) \notin Q\text{-edges}$
 then have *v-w-in-B*: $(v,w) \in B$ unfolding *Q'-edges-def* by *blast*
 }

```

obtain  $Q'-v-w-witness$  where  $Q'-v-w-witness$ :
   $Q'-v-w-witness \in Q'$   $(v,w) \in edges-of-walk\ Q'-v-w-witness$ 
  using  $*(1)$  unfolding  $Q'-edges-def$  by blast

have  $Q'-v-w-witness \neq Q-hit'$  proof
  assume  $Q'-v-w-witness = Q-hit'$ 
  then have  $edges-of-walk\ Q'-v-w-witness =$ 
     $edges-of-walk\ (P-k-pre\ @\ [y]) \cup edges-of-walk\ (y\ \# \ Q-hit-post)$ 
  unfolding  $Q-hit'-def$  using  $walk-edges-decomp[of\ P-k-pre\ y\ Q-hit-post]$  by simp
  moreover have  $(v,w) \notin edges-of-walk\ (P-k-pre\ @\ [y])$ 
  using  $P-k-pre-edges\ v-w-in-B$  by blast
  moreover have  $(v,w) \notin edges-of-walk\ (y\ \# \ Q-hit-post)$  proof
  assume  $(v,w) \in edges-of-walk\ (y\ \# \ Q-hit-post)$ 
  then have  $(v,w) \in edges-of-walk\ Q-hit$ 
  unfolding  $Q-hit-decomp$  by  $(metis\ UnCI\ walk-edges-decomp)$ 
  then show False using  $*(2)$   $v-w-in-B\ Q-hit(1)$  unfolding  $Q-edges-def$  by blast
  qed
  ultimately show False using  $Q'-v-w-witness(2)$  by blast
qed
then have  $Q'-v-w-witness \in Q$  using  $Q'-v-w-witness(1)$  unfolding  $Q'-def$  by blast
then have False using  $*(2)$   $v-w-in-B\ Q'-v-w-witness(2)$  unfolding  $Q-edges-def$  by blast
}
moreover have  $\exists e \in Q-edges.\ e \notin Q'-edges$  proof–
obtain  $v\ w$  where  $v-w:$   $(v,w) \in edges-of-walk\ (Q-hit-pre\ @\ [y]) \cap B$ 
  using  $Q-hit-pre-edges$  by auto
then have  $v-w-in-Q-hit:$   $(v,w) \in edges-of-walk\ Q-hit \cap B$  unfolding  $Q-hit-decomp$ 
  by  $(metis\ Int-iff\ UnCI\ walk-edges-decomp)$ 
then have  $(v,w) \in Q-edges$  unfolding  $Q-edges-def$  using  $Q-hit(1)$  by blast
moreover have  $(v,w) \notin Q'-edges$  proof
  assume  $(v,w) \in Q'-edges$ 
  then have  $(v,w) \in edges-of-walk\ Q-hit'$  unfolding  $Q'-edges-def\ Q'-def$ 
  using  $IntD1\ v-w-in-Q-hit\ Q-hit-edges-disjoint$  by auto
  then have  $(v,w) \in edges-of-walk\ (y\ \# \ Q-hit-post)$  unfolding  $Q-hit'-def$ 
  using  $v-w\ P-k-pre-edges$ 
  by  $(metis\ (no-types,\ lifting)\ IntD2\ UnE\ disjoint-iff-not-equal\ walk-edges-decomp)$ 
  then show False using  $v-w\ Q-hit(1)\ Q.paths\ Q-hit-decomp$ 
  by  $(metis\ DiffE\ Q.path-edges-remove-prefix\ IntD1\ path-from-to-def)$ 
qed
ultimately show ?thesis by blast
qed
moreover have finite  $Q-edges$  unfolding  $Q-edges-def\ B-def$  by simp
moreover have finite  $Q'-edges$  unfolding  $Q'-edges-def\ B-def$  by simp
ultimately have  $card\ Q'-edges < card\ Q-edges$  by  $(metis\ card-seteq\ not-le\ subrelI)$ 
then have  $card\ (\bigcup (edges-of-walk\ 'Q') \cap B) < card\ (\bigcup (edges-of-walk\ 'Q) \cap B)$ 
  unfolding  $Q-edges-def\ Q'-edges-def$  by blast
then show ?thesis unfolding  $Q-weight-def$  by blast
qed

private lemma  $DisjointPaths-Q'$ :  $DisjointPaths\ H-x\ v0\ v1\ Q'$  proof–
interpret  $Q-reduced:$   $DisjointPaths\ H-x\ v0\ v1\ Q - \{Q-hit\}$ 
  using  $Q.DisjointPaths-reduce[of\ Q - \{Q-hit\}]$  by blast
{

```

```

fix xs v
assume xs: xs ∈ Q − {Q-hit}
  and v: v ∈ set xs v ∈ set Q-hit' v ≠ v0 v ≠ v1
have v ∉ set P-k-pre proof
  assume v ∈ set P-k-pre
  then have ¬hitting-Q-or-new-last v using y-min by blast
  moreover have v ≠ new-last using v(2) calculation hitting-Q-or-new-last-def v(3) by auto
  ultimately show False unfolding hitting-Q-or-new-last-def using v(1,3) xs by blast
qed
moreover have v ≠ y
  by (metis DiffE Q.paths-disjoint Q-hit y-neq-v0 y-neq-v1 insert-iff v(1) xs)
moreover have v ∉ set Q-hit-post proof
  assume v ∈ set Q-hit-post
  then have v ∈ set Q-hit unfolding Q-hit-decomp by simp
  then show False using Q.paths-disjoint[of Q-hit xs] xs Q-hit(1) v by blast
qed
ultimately have False using v(2) unfolding Q-hit'-def by simp
}
then show ?thesis using Q-reduced.DisjointPaths-extend Q-hit'-v0-v1-H-x
unfolding Q'-def by blast
qed

private lemma card-Q': card Q' = sep-size proof–
  have Suc (card (Q − {Q-hit})) = card Q
  using Q-hit(1) Q.finite-paths by (meson card-Suc-Diff1)
  then show ?thesis using Q(2) Q.finite-paths Q-hit'-notin-Q-minus-Q-hit
  unfolding Q'-def by simp
qed

lemma contradiction': False using Q-weight-smaller DisjointPaths-Q' card-Q' Q-min
using Suc-leI not-less-eq-eq by blast
end — anonymous context

corollary contradiction: False using Q-hit-exists contradiction' by blast

end — locale ProofStepInduct-y-neq-new-last
end

```

11 Menger's Theorem

theory Menger **imports** Y-eq-new-last Y-neq-new-last **begin**

In this section, we combine the cases and finally prove Menger's Theorem.

locale ProofStepInductOptimalPaths = ProofStepInduct +

assumes optimal-paths:

$$\begin{aligned} & \bigwedge \text{paths}' P\text{-new}'. \text{ProofStepInduct } G \ v0 \ v1 \ \text{paths}' \ P\text{-new}' \ \text{sep-size} \\ & \implies \text{Digraph.distance (remove-vertex } v0) \ (\text{last } P\text{-new}') \ v1 \\ & \leq \text{Digraph.distance (remove-vertex } v0) \ (\text{last } P\text{-new}') \ v1 \end{aligned}$$

begin

lemma one-more-paths-exists-trivial:

$new-last = v1 \implies \exists paths. DisjointPaths G v0 v1 paths \wedge card paths = Suc sep-size$
using $P-new-solves-if-disjoint paths-sep-size$ **by** $blast$

lemma *one-more-paths-exists-nontrivial*:

assumes $new-last \neq v1$

shows $\exists paths. DisjointPaths G v0 v1 paths \wedge card paths = Suc sep-size$

proof–

interpret $ProofStepInduct-NonTrivial G v0 v1 paths P-new sep-size$

using $assms new-last-def$ **by** $unfold-locales simp$

obtain $P-k-pre y P-k-post$ **where**

$ProofStepInduct-NonTrivial-P-k-pre G v0 v1 paths P-new sep-size P-k-pre y P-k-post$

using $ProofStepInduct-NonTrivial-P-k-pre-exists$ **by** $blast$

then interpret $ProofStepInduct-NonTrivial-P-k-pre G v0 v1 paths P-new sep-size P-k-pre y P-k-post$.

{

assume $y \neq v1 y = new-last$

then interpret $ProofStepInduct-y-eq-new-last G v0 v1 paths P-new sep-size P-k-pre y P-k-post$

using $optimal-paths[folded H-def]$ **by** $unfold-locales$

have $?thesis$ **using** $with-optimal-paths-solves$ **by** $blast$

} **moreover** {

assume $y \neq v1 y \neq new-last$

then interpret $ProofStepInduct-y-neq-new-last G v0 v1 paths P-new sep-size P-k-pre y P-k-post$

by $unfold-locales$

have $?thesis$ **using** $contradiction$ **by** $blast$

}

ultimately show $?thesis$ **using** $y-eq-v1-solves$ **by** $blast$

qed

corollary *one-more-paths-exists*:

shows $\exists paths. DisjointPaths G v0 v1 paths \wedge card paths = Suc sep-size$

using $one-more-paths-exists-trivial one-more-paths-exists-nontrivial$ **by** $blast$

end

lemma (in $ProofStepInduct$) *one-more-paths-exists*:

$\exists paths. DisjointPaths G v0 v1 paths \wedge card paths = Suc sep-size$

proof–

define $paths-weight$ **where** $paths-weight \equiv$

$\lambda(paths' :: 'a Walk set, P-new'). Digraph.distance (remove-vertex v0) (last P-new') v1$

define $paths-good$ **where** $paths-good \equiv$

$\lambda(paths', P-new'). ProofStepInduct G v0 v1 paths' P-new' sep-size$

have $\exists paths' P-new'. paths-good (paths', P-new')$

unfolding $paths-good-def$ **using** $ProofStepInduct-axioms$ **by** $auto$

then obtain P' **where**

$P': paths-good P' \wedge P''. paths-good P'' \implies paths-weight P' \leq paths-weight P''$

using $arg-min-ex[of paths-good paths-weight]$ **by** $blast$

then obtain $paths' P-new'$ **where** $P'-decomp: P' = (paths', P-new')$ **by** (meson surj-pair)

have $optimal-paths-good: ProofStepInduct G v0 v1 paths' P-new' sep-size$

using $P'(1) P'-decomp$ **unfolding** $paths-good-def$ **by** $auto$

have $\bigwedge \text{paths}'' P\text{-new}''$. *paths-good* (*paths''*, *P-new''*)
 \implies *paths-weight* $P' \leq$ *paths-weight* (*paths''*, *P-new''*) **by** (*simp add: P'(2)*)
then have *optimal-paths-min*: $\bigwedge \text{paths}'' P\text{-new}''$. *ProofStepInduct* $G v0 v1 \text{paths}'' P\text{-new}'' \text{sep-size}$
 \implies *Digraph.distance* (*remove-vertex v0*) (*last P-new'*) $v1$
 \leq *Digraph.distance* (*remove-vertex v0*) (*last P-new''*) $v1$
unfolding *paths-good-def paths-weight-def* **by** (*simp add: P'-decomp*)

interpret G : *ProofStepInductOptimalPaths* $G v0 v1 \text{paths}' P\text{-new}' \text{sep-size}$
using *optimal-paths-good optimal-paths-min*
by (*simp add: ProofStepInductOptimalPaths.intro ProofStepInductOptimalPaths-axioms.intro*)
show *?thesis* **using** G .*one-more-paths-exists* **by** *blast*
qed

11.1 Menger's Theorem

theorem (*in v0-v1-Digraph*) *menger*:
assumes $\bigwedge S$. *Separation* $G v0 v1 S \implies \text{card } S \geq n$
shows $\exists \text{paths}$. *DisjointPaths* $G v0 v1 \text{paths} \wedge \text{card paths} = n$
using *assms v0-v1-Digraph-axioms* **proof** (*induct n arbitrary: G*)
case ($0 G$)
then show *?case* **using** *v0-v1-Digraph.DisjointPaths-empty[of G]* *card.empty* **by** *blast*
next
case (*Suc n G*)
interpret G : *v0-v1-Digraph* $G v0 v1$ **using** *Suc(3)* .
have $\bigwedge S$. *Separation* $G v0 v1 S \implies n \leq \text{card } S$ **using** *Suc.prem* *Suc-leD* **by** *blast*
then obtain *paths* **where** P : *DisjointPaths* $G v0 v1 \text{paths}$ *card paths* $= n$ **using** *Suc(1,3)* **by**
blast
interpret G : *DisjointPaths* $G v0 v1 \text{paths}$ **using** *P(1)* .

obtain *P-new* **where**
 $P\text{-new}$: $v0 \rightsquigarrow P\text{-new} \rightsquigarrow_G v1$ *set* $P\text{-new} \cap G$.*second-vertices* $= \{\}$
using G .*disjoint-paths-new-path* *P(2)* *Suc.prem*(1) **by** *blast*
have $P\text{-new-new}$: $P\text{-new} \notin \text{paths}$
by (*metis G.paths-tl-notnil G.second-vertex-def G.second-vertices-def G.path-from-toE IntI*
 $P\text{-new}$ *empty-iff image-eqI list.set-sel(1) list.set-sel(2)*)

have G .*hitting-paths* $v1$ **unfolding** G .*hitting-paths-def* **using** $v0\text{-neq-}v1$ **by** *blast*
then have $\exists x \in \text{set } P\text{-new}$. G .*hitting-paths* x **using** $P\text{-new}(1)$ **by** *fastforce*
then obtain *new-pre* x *new-post* **where**
 $P\text{-new-decomp}$: $P\text{-new} = \text{new-pre} @ x \# \text{new-post}$
and x : G .*hitting-paths* x
 $\bigwedge y$. $y \in \text{set } \text{new-pre} \implies \neg G$.*hitting-paths* y
by (*metis split-list-first-prop*)

have 1 : *DisjointPathsPlusOne* $G v0 v1 \text{paths}$ ($\text{new-pre} @ [x]$) **proof**
show $v0 \rightsquigarrow (\text{new-pre} @ [x]) \rightsquigarrow_G \text{last} (\text{new-pre} @ [x])$ **using** $P\text{-new}(1)$
by (*metis G.path-decomp' P-new-decomp append-is-Nil-conv hd-append2 list.distinct(1)*
 $\text{list.sel}(1)$ *path-from-to-def self-append-conv2*)
then show $\text{tl} (\text{new-pre} @ [x]) \neq []$
by (*metis DisjointPaths.hitting-paths-def G.DisjointPaths-axioms G.path-from-toE*
 $\text{butlast.simps}(1)$ *butlast-snoc list.distinct(1) list.sel(1) self-append-conv2*
 $\text{tl-append2 } x(1)$)

```

have new-pre ≠ Nil using G.hitting-paths-def P-new(1) P-new-decomp x(1) by auto
then have hd (tl (new-pre @ [x])) = hd (tl P-new) by (simp add: P-new-decomp hd-append)
then show hd (tl (new-pre @ [x])) ∉ G.second-vertices
  by (metis P-new(2) P-new-decomp ⟨new-pre ≠ []⟩ append-is-Nil-conv disjoint-iff-not-equal
    list.distinct(1) list.set-sel(1) list.set-sel(2) tl-append2)
show G.hitting-paths (last (new-pre @ [x])) using x(1) by auto
show  $\bigwedge v. v \in \text{set } (\text{butlast } (\text{new-pre } @ [x])) \implies \neg G.\text{hitting-paths } v$  by (simp add: x(2))
qed

```

```

have 2: NoSmallSeparationsInduct G v0 v1 n
by (simp add: G.v0-v1-Digraph-axioms NoSmallSeparationsInduct.intro
  NoSmallSeparationsInduct-axioms-def Suc.hyps Suc.prem(1))

```

```

show ?case proof (rule ccontr)
  assume not-case: ¬?case
  have x ≠ v1 proof
    assume x = v1
    define paths' where paths' = insert P-new paths
    {
      fix xs v
      assume ∗: xs ∈ paths v ∈ set xs v ∈ set P-new v ≠ v0 v ≠ v1
      have v ∈ set new-pre
        by (metis ∗(3,5) G.path-from-to-ends G.path-from-toE P-new(1) P-new-decomp
          ⟨x = v1⟩ butlast-snoc set-butlast)
      then have False using ∗(1,2,4) G.hitting-paths-def x(2) by auto
    }
    then have DisjointPaths G v0 v1 paths' unfolding paths'-def
      using G.DisjointPaths-extend P-new(1) by blast
    moreover have card paths' = Suc n
      using P-new-new by (simp add: G.finite-paths P(2) paths'-def)
    ultimately show False using not-case by blast
  qed
have ProofStepInduct-axioms paths n proof
  show n ≠ 0
    using G.DisjointPaths-extend G.finite-paths P(2) P-new(1) not-case card-insert-disjoint
    by fastforce
  qed (insert P(2))
  then have ProofStepInduct G v0 v1 paths (new-pre @ [x]) n
    using 1 2 by (simp add: ProofStepInduct.intro)
  then show False using ProofStepInduct.one-more-paths-exists not-case by metis
qed
qed

```

The previous theorem was the difficult direction of Menger's Theorem. Let us now prove the other direction: If we have n disjoint paths, than every separator must contain at least n vertices. This direction is rather trivial because every separator needs to separate at least the n paths, so we do not need induction or an elaborate setup to prove this.

```

theorem (in v0-v1-Digraph) menger-trivial:
  assumes DisjointPaths G v0 v1 paths card paths = n
  shows  $\bigwedge S. \text{Separation } G v0 v1 S \implies \text{card } S \geq n$ 
proof –

```

```

interpret DisjointPaths G v0 v1 paths using assms(1) .
fix S assume Separation G v0 v1 S
then interpret S: Separation G v0 v1 S .

```

Our plan is to show $n \leq \text{card } S$ by defining an injective function from *paths* into *S*. Because we have $\text{card } \textit{paths} = n$, the result follows.

For the injective function, we simply use the observation stated above: Every path needs to be separated by *S* at some vertex, so we can choose such a vertex.

```

define f where f  $\equiv \lambda xs. \text{SOME } v. v \in S \wedge v \in \text{set } xs$ 

have f-good:  $\bigwedge xs. xs \in \textit{paths} \implies f \textit{ xs} \in S \wedge f \textit{ xs} \in \text{set } xs$  proof -
  fix xs assume xs  $\in \textit{paths}$ 
  then obtain v where  $v \in \text{set } xs \cap S$  using S.S-separates paths by fastforce
  then show  $f \textit{ xs} \in S \wedge f \textit{ xs} \in \text{set } xs$  unfolding f-def
    using someI[of  $\lambda v. v \in S \wedge v \in \text{set } xs$  v] by blast
qed

```

This *f* is injective because no two paths intersect in the same vertex.

```

have inj-on f paths proof
  fix xs ys
  assume  $*$ :  $xs \in \textit{paths} \text{ } ys \in \textit{paths} \text{ } f \textit{ xs} = f \textit{ ys}$ 
  then obtain v where  $v \in S \text{ } v \in \text{set } xs \text{ } v \in \text{set } ys$ 
    using f-good by fastforce
  then show  $xs = ys$  using  $*(1,2)$  paths-disjoint S.v0-notin-S S.v1-notin-S by fastforce
qed

```

```

then show  $\text{card } S \geq n$  using assms(2) f-good
  by (metis S.finite-S finite-paths image-subsetI inj-on-iff-card-le)
qed

```

11.2 Self-contained Statement of the Main Theorem

Let us state both directions of Menger's Theorem again in a more self-contained way in the *Digraph* locale. Stating the theorems in a self-contained way helps avoiding mistakes due to wrong definitions hidden in one of the numerous locales we used and also significantly reduces the work needed to review this formalization.

With the statements below, all you need to do in order to verify that this formalization actually expresses Menger's Theorem (and not something else), is to look into the assumptions and definitions of the *Digraph* locale.

```

theorem (in Digraph) menger:
  fixes v0 v1 :: 'a and n :: nat
  assumes v0-V:  $v0 \in V$ 
    and v1-V:  $v1 \in V$ 
    and v0-nonadj-v1:  $\neg v0 \rightarrow v1$ 
    and v0-neq-v1:  $v0 \neq v1$ 
    and no-small-separators:  $\bigwedge S.$ 
       $\llbracket S \subseteq V; v0 \notin S; v1 \notin S; \bigwedge xs. v0 \rightsquigarrow xs \rightsquigarrow v1 \implies \text{set } xs \cap S \neq \{\} \rrbracket \implies \text{card } S \geq n$ 
  shows  $\exists \textit{paths}. \text{card } \textit{paths} = n \wedge (\forall xs \in \textit{paths}.$ 
     $v0 \rightsquigarrow xs \rightsquigarrow v1 \wedge (\forall ys \in \textit{paths} - \{xs\}. (\forall v \in \text{set } xs \cap \text{set } ys. v = v0 \vee v = v1)))$ 

```

proof-
interpret $v0-v1$ -Digraph $G v0 v1$ **using** $v0-V v1-V v0-nonadj-v1 v0-neq-v1$ **by** *unfold-locales*
have $\bigwedge S. Separation\ G\ v0\ v1\ S \implies n \leq card\ S$ **using** *no-small-separators*
by (*simp add: Separation.S-V Separation.S-separates Separation.v0-notin-S Separation.v1-notin-S*)
then obtain *paths* **where**
paths: DisjointPaths G v0 v1 paths card paths = n **using** *no-small-separators menger* **by** *blast*
then show *?thesis*
by (*metis DiffD1 DiffD2 DisjointPaths.paths DisjointPaths.paths-disjoint IntD1 IntD2 singletonI*)
qed

theorem (in *Digraph*) *menger-trivial*:

fixes $v0 v1 :: 'a$ **and** $n :: nat$

assumes $v0-V: v0 \in V$

and $v1-V: v1 \in V$

and $v0-nonadj-v1: \neg v0 \rightarrow v1$

and $v0-neq-v1: v0 \neq v1$

and $n-paths: card\ paths = n$

and $paths-disjoint: \forall xs \in paths.$

$v0 \rightsquigarrow_{xs} v1 \wedge (\forall ys \in paths - \{xs\}. (\forall v \in set\ xs \cap set\ ys. v = v0 \vee v = v1))$

shows $\bigwedge S. \llbracket S \subseteq V; v0 \notin S; v1 \notin S; \bigwedge xs. v0 \rightsquigarrow_{xs} v1 \implies set\ xs \cap S \neq \{\} \rrbracket \implies card\ S \geq n$

proof-

interpret $v0-v1$ -Digraph $G v0 v1$ **using** $v0-V v1-V v0-nonadj-v1 v0-neq-v1$ **by** *unfold-locales*

interpret *DisjointPaths G v0 v1 paths* **proof**

show $\bigwedge xs. xs \in paths \implies v0 \rightsquigarrow_{xs} v1$ **using** *paths-disjoint* **by** *simp*

next

fix $xs\ ys\ v$ **assume** $xs \in paths\ ys \in paths\ xs \neq ys\ v \in set\ xs\ v \in set\ ys$

then have $xs \in paths\ ys \in paths - \{xs\}\ v \in set\ xs \cap set\ ys$ **by** *blast+*

then show $v = v0 \vee v = v1$ **using** *paths-disjoint* **by** *blast*

qed

fix S **assume** $S \subseteq V\ v0 \notin S\ v1 \notin S\ \bigwedge xs. v0 \rightsquigarrow_{xs} v1 \implies set\ xs \cap S \neq \{\}$

then interpret *Separation G v0 v1 S* **by** *unfold-locales*

show $card\ S \geq n$ **using** *menger-trivial DisjointPaths-axioms Separation-axioms n-paths* **by** *blast*

qed

end

References

- [Loc16] Andreas Lochbihler. A formal proof of the max-flow min-cut theorem for countable networks. *Archive of Formal Proofs*, May 2016. http://isa-afp.org/entries/MFMC_Countable.shtml, Formal proof development.
- [McC84] William McCuaig. A simple proof of Menger's theorem. *Journal of Graph Theory*, 8(3):427–429, 1984. doi:10.1002/jgt.3190080311.
- [Men27] Karl Menger. Zur allgemeinen Kurventheorie. *Fundamenta Mathematicae*, 10(1):96–115, 1927. URL: <http://eudml.org/doc/211191>.