Menger's Theorem

Christoph Dittmann
isabelle@christoph-d.de

March 17, 2025

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

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*: $[\![\ x \in set\ xs;\ x \neq last\ xs\ ]\!] \Longrightarrow x \in set\ (butlast\ xs)$
  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** $P\ x$ $\bigwedge y.\ P\ y \Longrightarrow weight\ x \leq weight\ y$
proof (*cases* $\exists\, x.\ P\ x \wedge weight\ x = 0$)
  case *True* **then show** *?thesis* **using** *that* **by** *auto*
next
  case *False* **then show** *?thesis*
    using *that ex-least-nat-le*[*of* $\lambda n.\ \exists\, x.\ P\ x \wedge weight\ x = n$] *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* (‹*V*ı›)
  arcs :: *'a Edge set* (‹*E*ı›)
abbreviation *is-arc* :: (*'a*, *'b*) *Graph-scheme* ⇒ *'a* ⇒ *'a* ⇒ *bool* (**infixl** ‹→ı› *60*) **where**
  $v \rightarrow_G w \equiv (v,w) \in E_G$

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*) $\Longrightarrow$ *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*: $[\![$ *walk* (*xs* @ *ys*); *xs* $\neq$ *Nil*; *ys* $\neq$ *Nil* $]\!]$ $\Longrightarrow$ *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* $\Longrightarrow$ *path* [*v*] **by** *simp*
lemma *path-2* [*simp*]: $[\![$ *v*$\rightarrow$*w*; *v* $\neq$ *w* $]\!]$ $\Longrightarrow$ *path* [*v*,*w*] **by** *simp*
lemma *path-cons*: $[\![$ *path xs*; *xs* $\neq$ *Nil*; *v*$\rightarrow$*hd xs*; *v* $\notin$ *set xs* $]\!]$ $\Longrightarrow$ *path* (*v* # *xs*)
by (*metis distinct.simps*(*2*) *list.exhaust-sel walk.Cons*)
lemma *path-comp*: $[\![$ *walk xs*; *walk ys*; *xs* = *Nil* $\vee$ *ys* = *Nil* $\vee$ *last xs*$\rightarrow$*hd ys*; *distinct* (*xs* @ *ys*) $]\!]$
$\Longrightarrow$ *path* (*xs* @ *ys*) **using** *walk-comp* **by** *blast*
lemma *path-tl*: *path xs* $\Longrightarrow$ *path* (*tl xs*) **by** (*simp add*: *distinct-tl walk-tl*)
lemma *path-drop*: *path xs* $\Longrightarrow$ *path* (*drop n xs*) **by** (*simp add*: *walk-drop*)
lemma *path-take*: *path xs* $\Longrightarrow$ *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*) $\Longrightarrow$ *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* $\Longrightarrow$ *set xs* $\subseteq$ *V* **by** (*simp add*: *walk-in-V*)
lemma *path-length*: *path xs* $\Longrightarrow$ *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*) $\Longrightarrow$ *v*$\rightarrow$*w* **using** *walk-first-edge* **by** *blast*
lemma *path-first-edge'*: $[\![$ *path* (*v* # *xs*); *xs* $\neq$ *Nil* $]\!]$ $\Longrightarrow$ *v*$\rightarrow$*hd xs* **using** *walk-first-edge'* **by** *blast*
lemma *path-middle-edge*: *path* (*xs* @ *v* # *w* # *ys*) $\Longrightarrow$ *v* $\rightarrow$ *w* **using** *walk-middle-edge* **by** *blast*
lemma *path-first-vertex*: *path* (*x* # *xs*) $\Longrightarrow$ *x* $\notin$ *set xs* **by** *simp*
lemma *path-disjoint*: $[\![$ *path* (*xs* @ *ys*); *xs* $\neq$ *Nil*; *x* $\in$ *set xs* $]\!]$ $\Longrightarrow$ *x* $\notin$ *set ys* **by** *auto*

4.3 The Set of All Paths

definition *all-paths* **where** *all-paths* $\equiv$ { *xs* | *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*
(‹- ⇝-⇝ı -› [*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*]: ⟦ *path xs*; *xs* $\neq$ *Nil*; *hd xs* $=$ *v*; *last xs* $=$ *w* ⟧ $\Longrightarrow$ *v* ⇝*xs*⇝ *w*
and *path-from-toE* [*dest*]: *v* ⇝*xs*⇝ *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* ⇝(*xs* @ *w* # *ys*)⇝ *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* ⇝(*xs* @ *x* # *xs'*)⇝ *w* *v'* ⇝(*ys* @ *x* # *ys'*)⇝ *w'* *set xs* $\cap$ *set ys'* $=$ {}
shows *v* ⇝(*xs* @ *x* # *ys'*) ⇝ *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* ⇝*xs*⇝ *w* $\Longrightarrow$ *v* $\notin$ *set* (*tl xs*)
by (*metis path-from-toE list.collapse path-first-vertex*)

lemma *path-from-to-first'*: *v* ⇝(*xs* @ *x* # *xs'*)⇝ *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* ⇝*xs*⇝ *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* ⇝(*xs* @ *x* # *xs'*)⇝ *w* $\Longrightarrow$ *w* $\notin$ *set xs*
by (*metis path-from-toE bex-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* ⇝*ys*⇝ *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* ⇝*ys*⇝ *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: $n = length\ xs$ *walk xs* $xs \neq Nil$ $hd\ xs = v$ $last\ xs = w$ $set\ xs \subseteq target\text{-}set$ **and**
hyp: $\neg(\exists\, ys.\ v \leadsto ys \leadsto w \wedge set\ ys \subseteq target\text{-}set)$ **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-decomp*: $u \in set\ ys$ *distinct ys* $xs = ys\ @\ u\ \#\ rest$
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-suffix* **where**
ys-decomp: $ys = ys'\ @\ u\ \#\ ys\text{-}suffix$ **by** (*meson split-list xs-decomp*(*1*))
define *zs* **where** $zs \equiv ys'\ @\ u\ \#\ rest$
have *walk zs* **unfolding** *zs-def* **using** *xs*(*2*) *xs-decomp*(*3*) *ys-decomp*
by (*metis walk-decomp list.sel*(*1*) *list.simps*(*3*) *walk-comp walk-last-edge*)
moreover have $length\ zs < n$ **unfolding** *zs-def* **by** (*simp add*: *xs*(*1*) *xs-decomp*(*3*) *ys-decomp*)
moreover have $hd\ zs = v$ **unfolding** *zs-def*
by (*metis append-is-Nil-conv hd-append list.sel*(*1*) *xs*(*4*) *xs-decomp*(*3*) *ys-decomp*)
moreover have $last\ zs = w$ **unfolding** *zs-def* **using** *xs*(*5*) *xs-decomp*(*3*) **by** *auto*
moreover have $set\ zs \subseteq target\text{-}set$ **unfolding** *zs-def* **using** *xs*(*6*) *xs-decomp*(*3*) *ys-decomp* **by** *auto*
ultimately show *?case* **using** *zs-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-of-walk* :: *′a Walk* ⇒ *′a Edge set* **where**
$edges\text{-}of\text{-}walk\ xs = \{ (v,w) \mid v\ w\ xs\text{-}pre\ xs\text{-}post.\ xs = xs\text{-}pre\ @\ v\ \#\ w\ \#\ xs\text{-}post \}$

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

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

lemma *edges-of-walk-finite*: *walk xs* ⟹ *finite* (*edges-of-walk xs*)
using *edges-of-walk-in-E finite-edge-set finite-subset* **by** *blast*

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

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

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

lemma edges-of-walk-edge: ⟦ walk xs; (v,w) ∈ edges-of-walk xs ⟧ ⟹ v→w
  using edges-of-walkE walk-middle-edge by fastforce

lemma edges-of-walk-middle [simp]: (v,w) ∈ edges-of-walk (xs @ v # w # xs′)
  unfolding edges-of-walk-def by blast

lemma edges-of-comp1: edges-of-walk xs ⊆ edges-of-walk (xs @ ys)
  unfolding edges-of-walk-def by force
lemma edges-of-comp2: edges-of-walk ys ⊆ edges-of-walk (xs @ ys) proof−
  {
    fix v w assume (v,w) ∈ edges-of-walk ys
    then have ∃ ys-pre ys-post. ys = ys-pre @ v # w # ys-post by (meson edges-of-walkE)
    then have (v,w) ∈ edges-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-edges-decomp-simple:
  edges-of-walk (v # w # xs) = {(v,w)} ∪ edges-of-walk (w # xs) (is ?A = ?B)
proof
  have edges-of-walk (w # xs) ⊆ ?A using edges-of-comp2[of w # xs [v]] by simp
  moreover have (v,w) ∈ ?A by (metis append-eq-Cons-conv edges-of-walk-middle)
  ultimately show ?B ⊆ ?A by blast
  {
    fix v′ w′ assume (v′,w′) ∈ ?A
    then obtain xs-pre xs-post where xs-decomp: v # w # xs = xs-pre @ v′ # w′ # xs-post
      using edges-of-walkE by blast
    have (v′,w′) ∈ ?B proof (cases)
      assume xs-pre = Nil then show ?thesis using xs-decomp by auto
    next
      assume xs-pre ≠ Nil then show ?thesis
        by (metis Cons-eq-append-conv UnI2 edges-of-walk-middle xs-decomp)
    qed
  }
  then show ?A ⊆ ?B by auto
qed

lemma walk-edges-decomp:
  edges-of-walk (xs @ x # xs′) = edges-of-walk (xs @ [x]) ∪ edges-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-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-pre xs-post* **where**
xs-decomp: $xs = xs\text{-}pre\ @\ v \# xs\text{-}post$ **by** (*meson split-list*)
show $v \in$ *set ys* **proof** (*cases*)
assume *xs-pre* $=$ *Nil*
then have *xs-post* $\neq$ *Nil* **using** *xs-decomp non-trivial* **by** *auto*
then have $xs = xs\text{-}pre\ @\ v \#$ *hd xs-post* $\#$ *tl xs-post* **by** (*simp add*: *xs-decomp*)
then have $(v,$ *hd xs-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-pre* $\neq$ *Nil*
then have $xs =$ *butlast xs-pre* $@$ *last xs-pre* $\# v \#$ *xs-post* **by** (*simp add*: *xs-decomp*)
then have (*last xs-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-pre xs-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-pre xs-post* **where**
xs-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-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* :: $'a \Rightarrow 'a \Rightarrow nat$ **where**
distance $v\ w \equiv Min\ \{\ length\ xs \mid xs.\ v \leadsto xs \leadsto w\ \}$

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

lemma *distance-lengths-finite*: *finite* $\{\ length\ xs \mid xs.\ v \leadsto xs \leadsto w\ \}$ **proof**−
have $\{\ length\ xs \mid xs.\ v \leadsto xs \leadsto w\ \} \subseteq \{\ n \mid n.\ n \leq card\ V\ \}$ **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*: $v \leadsto xs \leadsto w \Longrightarrow distance\ v\ w \leq length\ xs$
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*: $v \leadsto xs \leadsto w$
and *xs-min*: $\bigwedge xs'.\ v \leadsto xs' \leadsto w \Longrightarrow length\ xs \leq length\ xs'$
shows $distance\ v\ w = length\ xs$
proof−
have $\bigwedge d.\ d \in \{length\ xs \mid xs.\ v \leadsto xs \leadsto w\} \Longrightarrow length\ xs \leq d$ **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* :: $'a \Rightarrow ('a, 'b)$ *Graph-scheme* **where**
remove-vertex $x \equiv G(\!|\ verts := V - \{x\},\ arcs := Restr\ E\ (V - \{x\})\ |\!)$

lemma *remove-vertex-V*: $V_{remove\text{-}vertex\ x} = V - \{x\}$ **unfolding** *remove-vertex-def* **by** *auto*
lemma *remove-vertex-V'*: $V_{remove\text{-}vertex\ x} \subseteq V$ **unfolding** *remove-vertex-def* **by** *auto*
lemma *remove-vertex-E*: $E_{remove\text{-}vertex\ x} = Restr\ E\ (V - \{x\})$ **unfolding** *remove-vertex-def* **by** *simp*
lemma *remove-vertex-E'*: $v \rightarrow_{remove\text{-}vertex\ x} w \Longrightarrow v \rightarrow w$ **by** (*simp add*: *remove-vertex-E*)
lemma *remove-vertex-E''*: $[\![\ v \rightarrow w;\ v \neq x;\ w \neq x\ ]\!] \Longrightarrow v \rightarrow_{remove\text{-}vertex\ x} w$
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 $?V = V_{remove\text{-}vertex\ v}$ **let** $?E = E_{remove\text{-}vertex\ v}$

show *finite ?V* **unfolding** *remove-vertex-def* **using** *finite-vertex-set* **by** *simp*
show $?E \subseteq ?V \times ?V$ **proof**
fix *e* **assume** $e \in ?E$
then have $e \in (V - \{v\}) \times (V - \{v\})$ **by** (*metis Int-iff remove-vertex-E*)
then show $e \in ?V \times ?V$ **using** *remove-vertex-V* **by** *auto*
qed
have $\bigwedge x\ y.\ [\![(x,y) \in ?E; (x,y) \notin E]\!] \Longrightarrow (y,x) \in ?E$ **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* $x \notin$ *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 $v \in V - \{x\}$ **by** *simp*
then show *?case* **using** *remove-vertex-V* **by** *simp*
next
case (*Cons v w vs*)
then have $v \rightarrow_{remove\text{-}vertex\ x} w$ **using** *remove-vertex-E''* **by** *auto*
then show *?case*
by (*meson Cons.hyps*(*3*) *Cons.prems*(*1*) *H.Cons assms*(*2*) *list.set-intros*(*2*))
qed *simp*
qed

The same holds for paths.

lemma *remove-vertex-path-from-to*:
$[\![v \leadsto xs \leadsto w; x \in V; x \notin set\ xs]\!] \Longrightarrow v \leadsto xs \leadsto_{remove\text{-}vertex\ x} w$
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*: $v \rightsquigarrow xs \rightsquigarrow_{remove\text{-}vertex\ x}\ w \Longrightarrow v \rightsquigarrow xs \rightsquigarrow w$
  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-nonadj-v1*: $\neg v0 \rightarrow v1$
    and *v0-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 undirecteded 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 set*
  assumes *S-V*: $S \subseteq V$
    and *v0-notin-S*: $v0 \notin S$
    and *v1-notin-S*: $v1 \notin S$
    and *S-separates*: $\bigwedge xs.\ v0 \leadsto xs \leadsto v1 \Longrightarrow set\ xs \cap S \neq \{\}$

lemma (**in** *Separation*) *finite-S* [*simp*]: *finite S* **using** *S-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 *insert v S* $\subseteq$ *V* **by** (*meson G.S-V assms*(*3*) *insert-subsetI remove-vertex-V' subset-trans*)
    show $v0 \notin$ *insert v S* **using** *G.v0-notin-S assms*(*1*) **by** *blast*
    show $v1 \notin$ *insert v S* **using** *G.v1-notin-S assms*(*2*) **by** *blast*
  next
    fix *xs* **assume** $v0 \leadsto xs \leadsto v1$
    show *set xs* $\cap$ *insert v S* $\neq \{\}$ **proof** (*cases*)
      assume $v \notin$ *set xs*
      then have $v0 \leadsto xs \leadsto_{remove\text{-}vertex\ v} v1$
        using *remove-vertex-path-from-to* ‹$v0 \leadsto xs \leadsto v1$› *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*: $\bigwedge S.$ *Separation G v0 v1 S* $\Longrightarrow$ *card S* $\geq$ *Suc n*
    and *Separation* (*remove-vertex v*) *v0 v1 S*
  shows *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 $S \subseteq V$ $v0 \notin S$ $v1 \notin S$ $\neg$*Separation G v0 v1 S*
  shows $\exists xs.\ v0 \leadsto xs \leadsto v1 \wedge set\ xs \cap S = \{\}$
  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 *v0*-*v1*-paths. Let us define this concept.

locale *DisjointPaths* = *v0-v1-Digraph* +
  fixes *paths* :: *'a Walk set*
  assumes *paths*:
    $\bigwedge xs.\ xs \in paths \Longrightarrow v0 \leadsto xs \leadsto v1$
  and *paths-disjoint*: $\bigwedge xs\ ys\ v.$
    $[\![\ xs \in paths;\ ys \in paths;\ xs \neq ys;\ v \in set\ xs;\ v \in set\ ys\ ]\!] \Longrightarrow v = v0 \lor v = v1$

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 $\bigwedge xs.\ xs \in paths \Longrightarrow v0 \leadsto xs \leadsto v1$ **using** *remove-vertex-path-from-to-add H.paths* **by** *blast*
  show $\bigwedge xs\ ys\ v.\ [\![\ xs \in paths;\ ys \in paths;\ xs \neq ys;\ v \in set\ xs;\ v \in set\ ys\ ]\!] \Longrightarrow v = v0 \lor v = v1$
    by (*meson DisjointPaths.paths-disjoint H.DisjointPaths-axioms*)
qed

context *DisjointPaths* **begin**

lemma *paths-in-all-paths*: *paths* ⊆ *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* (⋃(*edges-of-walk* ‘ *paths*)) **proof**−
  have ⋃(*edges-of-walk* ‘ *paths*) ⊆ *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*: $xs \in paths \Longrightarrow tl\ xs \neq Nil$
  by (*metis path-from-toE hd-Cons-tl last-ConsL paths v0-neq-v1*)

lemma *paths-second-in-V*: $xs \in paths \Longrightarrow hd\ (tl\ xs) \in V$
  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*: $xs \in paths \Longrightarrow hd\ (tl\ xs) \neq v0$
  by (*metis distinct.simps(2) hd-in-set list.exhaust-sel path-from-to-def paths paths-tl-notnil*)

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

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

lemma *paths-second-disjoint*: $[\![$ *xs* $\in$ *paths*; *ys* $\in$ *paths*; *xs* $\neq$ *ys* $]\!] \Longrightarrow$ *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*$\leadsto$*P*$\leadsto$*v1*
and *P-disjoint*: $\bigwedge$*xs v*. $[\![$ *xs* $\in$ *paths*; *xs* $\neq$ *P*; *v* $\in$ *set xs*; *v* $\in$ *set P* $]\!] \Longrightarrow$ *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* $\Longrightarrow$ *v0* $\leadsto$*xs*$\leadsto$ *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* $\leadsto$*xs*$\leadsto$ *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* $\equiv$ λ*xs* :: *'a Walk*. *hd* (*tl xs*)
definition *second-vertices* **where** *second-vertices* $\equiv$ *second-vertex* ` *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*: *card second-vertices* = *card paths*
unfolding *second-vertices-def* **using** *finite-paths card-image second-vertex-inj* **by** *blast*

lemma *second-vertices-in-V*: *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* $\notin$ *second-vertices v1* $\notin$ *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*: *hd* (*tl xs*) $\notin$ *second-vertices* $\Longrightarrow$ *xs* $\notin$ *paths*
by (*metis image-iff second-vertex-def second-vertices-def*)

lemma *second-vertices-first-edge*:
$[\![$ *xs* $\in$ *paths*; *first-edge-of-walk xs* = (*v*,*w*) $]\!]$ $\Longrightarrow$ *w* $\in$ *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*. *Separation G v0 v1 S* $\Longrightarrow$ *card S* $\geq$ *Suc* (*card paths*)
shows $\exists$ *P-new*. *v0*$\rightsquigarrow$*P-new*$\rightsquigarrow$*v1* $\wedge$ *set P-new* $\cap$ *second-vertices* = {}
proof−
have $\neg$*Separation G v0 v1 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 *paths* = {}.

definition *hitting-paths* **where**
hitting-paths $\equiv$ λ*x*. *x* $\neq$ *v0* $\wedge$ (($\exists$ *xs* $\in$ *paths*. *x* $\in$ *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-new*$\rightsquigarrow$ (*last P-new*)
and *tl-P-new*:

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

7.1 Characterizing the New Path

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

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

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

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

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

lemma *paths-with-new-path*: *xs* $\in$ *paths-with-new* $\Longrightarrow$ *path xs*
using *P-new paths paths-with-new-def* **by** *auto*

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

7.2 The Last Vertex of the New Path

McCuaig in [McC84] calls the last vertex of *P-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-last*.

definition *new-pre* **where** *new-pre* $\equiv$ *butlast P-new*
definition *new-last* **where** *new-last* $\equiv$ *last P-new*

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

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

lemma *new-pre-hitting*: $x' \in$ *set new-pre* $\Longrightarrow$ $\neg$*hitting-paths* x'
by (*simp add*: *new-pre-def last-P-new*(*2*))

lemma *P-hit*: *hitting-paths new-last*
by (*simp add*: *last-P-new*(*1*) *new-last-def*)

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

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

lemma *new-last-to-v1*: $\exists R.\ \textit{new-last} \leadsto R \leadsto_{\textit{remove-vertex v0}} \textit{v1}$
proof (*cases*)
  assume *new-last* = *v1*
  then have $\textit{new-last} \leadsto [\textit{v1}] \leadsto_{\textit{remove-vertex v0}} \textit{v1}$
    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* ≠ *v1*
  then obtain *xs* **where** *xs*: *xs* ∈ *paths new-last* ∈ *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 $\textit{new-last} \leadsto (\textit{new-last}\ \#\ \textit{xs-post}) \leadsto \textit{v1}$ **using** ‹*xs* ∈ *paths*›
    by (*metis paths last-appendR list.sel*(*1*) *list.simps*(*3*) *path-decomp*(*2*) *path-from-to-def*)
  then have $\textit{new-last} \leadsto (\textit{new-last}\ \#\ \textit{xs-post}) \leadsto_{\textit{remove-vertex v0}} \textit{v1}$
    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* ∈ *paths-with-new ys* ∈ *paths-with-new xs* ≠ *ys v* ∈ *set xs v* ∈ *set ys*
  shows *v* = *v0* ∨ *v* = *v1* ∨ *v* = *new-last*
proof−
  have *xs* ∈ *paths* ∨ *ys* ∈ *paths* **using** *assms*(*1*,*2*,*3*) *paths-with-new-def* **by** *auto*
  then have *hitting-paths v* ∨ *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*:
  $\textit{new-last} = \textit{v1} \Longrightarrow \exists \textit{paths}'.\ \textit{DisjointPaths G v0 v1 paths}' \land \textit{card paths}' = \textit{Suc}\ (\textit{card paths})$
  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* ≡ *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* ≠ *v1* ⟹ *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 $P \in$ *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* $\leadsto$*xs*$\leadsto$ *w tl xs* $\neq$ *Nil v1* $\notin$ *set xs new-last* $\notin$ *set xs*
    and *P*: $P \in$ *paths-with-new hd* (*tl xs*) = *hd* (*tl P*)
    and *edges-subset*: *edges-of-walk xs* $\subseteq \bigcup$(*edges-of-walk* ' *paths-with-new*)
   shows *edges-of-walk xs* $\subseteq$ *edges-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.prems*(*1*,*2*)
   by (*metis path-from-toE Nitpick.size-list-simp*(*2*) *Suc-inject length-0-conv*)
  show *?case* **proof** (*cases*)
   assume $n = Suc\ 0$
   then obtain *v w* **where** *v-w*: *xs* = [*v*,*w*]
    by (*metis* (*full-types*) *Suc.hyps*(*2*) *length-0-conv length-Suc-conv*)
   then have $v = v0$ **using** *Suc.prems*(*1*) **by** *auto*
   moreover have *w* = *hd* (*tl P*) **using** *Suc.prems*(*5*) *v-w* **by** *auto*
   moreover have *edges-of-walk xs* = {(*v*, *w*)} **using** *v-w edges-of-walk-2* **by** *simp*
   moreover have (*v0*, *hd* (*tl P*)) $\in$ *edges-of-walk P* **using** *P tl-P-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 $n \neq Suc\ 0$
   obtain *xs' x* **where** *xs'*: *xs* = *xs'* @ [*x*]
    by (*metis path-from-toE Suc.prems*(*1*) *append-butlast-last-id*)
   then have *n* = *length xs'* **using** *xs'* **using** *Suc.hyps*(*2*) **by** *auto*
   moreover have *xs'-path*: *v0* $\leadsto$*xs'*$\leadsto$ *last xs'*
    using *xs' Suc.prems*(*1*) ‹*tl xs* $\neq$ *Nil*› *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** ‹$n \neq Suc\ 0$›
    by (*metis path-from-toE Nitpick.size-list-simp*(*2*) *calculation*(*1*,*2*))
   moreover have *v1* $\notin$ *set xs'* **using** *xs' Suc.prems*(*3*) **by** *auto*
   moreover have *new-last* $\notin$ *set xs'* **using** *xs' Suc.prems*(*4*) **by** *auto*
   moreover have *hd* (*tl xs'*) = *hd* (*tl P*)
    using *xs'* ‹*tl xs'* $\neq$ []› *Suc.prems*(*5*) *calculation*(*2*) **by** *auto*
   moreover have *edges-of-walk xs'* $\subseteq \bigcup$(*edges-of-walk* ' *paths-with-new*)
    using *xs' Suc.prems*(*6*) *edges-of-comp1* **by** *blast*
   ultimately have *xs'-edges*: *edges-of-walk xs'* $\subseteq$ *edges-of-walk P* **using** *Suc.hyps*(*1*) **by** *blast*
   moreover have *edges-of-walk xs* = *edges-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-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-last-edge xs′-edges* ‹*tl xs′* $\neq$ []› *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-last*
using *paths-plus-one-disjoint P′*(*1*) *P paths-with-new-def* **by** *auto*
then show *False* **using** *Suc.prems*(*3*) ‹*new-last* $\notin$ *set xs′*› ‹*tl xs′* $\neq$ []› *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-size* :: *nat*
— The size of a minimum separator.
assumes *no-small-separations*: $\bigwedge S$. *Separation G v0 v1 S* $\Longrightarrow$ *card S* $\geq$ *Suc sep-size*
— The induction hypothesis.
and *no-small-separations-hyp*: $\bigwedge G'$ :: (*′a*, *′b*) *Graph-scheme*.
($\bigwedge S$. *Separation G′ v0 v1 S* $\Longrightarrow$ *card S* $\geq$ *sep-size*)
$\Longrightarrow$ *v0-v1-Digraph G′ v0 v1*
$\Longrightarrow$ $\exists$ *paths*. *DisjointPaths G′ v0 v1 paths* $\wedge$ *card paths* = *sep-size*

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

If a minimum separator has size at least *Suc sep-size*, then it follows immediately from the induction hypothesis that we have *sep-size* many disjoint paths. We then observe that *second-vertices* of these paths is not a separator because *card second-vertices = 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 *Suc 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 *distance 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 *sep-size* many internally vertex-disjoint paths (with *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 *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 *new-last* $\neq$ *v1*.

locale *ProofStepInduct-NonTrivial* = *ProofStepInduct* +
  assumes *new-last-neq-v1*: *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 *sep-size* many internally vertex-disjoint paths, again by the induction hypothesis.

lemma *Q-exists*: $\exists$ *Q*. *DisjointPaths H-x v0 v1 Q* $\wedge$ *card Q = sep-size*
proof−
  have $\bigwedge$*S*. *Separation H-x v0 v1 S* $\Longrightarrow$ *card S* $\geq$ *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 $\bigcup$ (*edges-of-walk* ‘ *Q*) $\cap$ (*E* − $\bigcup$ (*edges-of-walk* ‘ *paths-with-new*)) is minimal.

definition *B* **where** *B* $\equiv$ *E* − $\bigcup$(*edges-of-walk* ‘ *paths-with-new*)

definition *Q-weight* **where** *Q-weight* $\equiv$ λ*Q*. *card* ($\bigcup$(*edges-of-walk* ‘ *Q*) $\cap$ *B*)

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

definition *Q* **where** *Q* $\equiv$ *SOME Q. Q-good Q*

It is easy to show that such a Q exists.

lemma *Q*: *DisjointPaths H-x v0 v1 Q card Q* = *sep-size*
and *Q-min*: $\bigwedge Q'$. *DisjointPaths H-x v0 v1* Q' $\wedge$ *card* Q' = *sep-size* $\Longrightarrow$ *Q-weight Q* $\leq$ *Q-weight* Q'
proof−
obtain Q' **where** *DisjointPaths H-x v0 v1* Q' *card* Q' = *sep-size*
$\bigwedge Q''$. *DisjointPaths H-x v0 v1* Q'' $\wedge$ *card* Q'' = *sep-size* $\Longrightarrow$ *Q-weight* Q' $\leq$ *Q-weight* Q''
using *arg-min-ex*[*of* λQ. *DisjointPaths H-x v0 v1 Q* $\wedge$ *card Q* = *sep-size Q-weight*]
new-last-neq-v1 Q-exists **by** *metis*
then have *Q-good* Q' **unfolding** *Q-good-def* **by** *blast*
then show *DisjointPaths H-x v0 v1 Q card Q* = *sep-size*
$\bigwedge Q'$. *DisjointPaths H-x v0 v1* Q' $\wedge$ *card* Q' = *sep-size* $\Longrightarrow$ *Q-weight Q* $\leq$ *Q-weight* Q'
using *someI*[*of Q-good*] **by** (*simp-all add*: *Q-def Q-good-def*)
qed

sublocale *Q*: *DisjointPaths H-x v0 v1 Q* **using** *Q*(*1*) **.**

8.3 Finding a Path Avoiding Q

Because Q contains only *sep-size* many paths, we have *card Q.second-vertices* = *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-k $\equiv$ *SOME P-k. P-k* $\in$ *paths-with-new* $\wedge$ *hd* (*tl P-k*) $\notin$ *Q.second-vertices*

lemma *P-k*: *P-k* $\in$ *paths-with-new hd* (*tl P-k*) $\notin$ *Q.second-vertices* **proof**−
obtain *y* **where** *y* $\in$ *insert* (*hd* (*tl P-new*)) *second-vertices y* $\notin$ *Q.second-vertices* **proof**−
have *hd* (*tl P-new*) $\notin$ *second-vertices* **using** *P-new-decomp tl-P-new*(*2*) **by** *simp*
moreover have *card second-vertices* = *card Q.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 *card* (*insert* (*hd* (*tl P-new*)) *second-vertices*) = *Suc* (*card Q.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-k. P-k* $\in$ *paths-with-new* $\wedge$ *hd* (*tl P-k*) $\notin$ *Q.second-vertices*
by (*metis* (*mono-tags*, *lifting*) *image-iff insertCI insertE paths-with-new-def second-vertex-def second-vertices-def*)
then show *P-k* $\in$ *paths-with-new hd* (*tl P-k*) $\notin$ *Q.second-vertices*
using *someI*[*of* λ*P-k. P-k* $\in$ *paths-with-new* $\wedge$ *hd* (*tl P-k*) $\notin$ *Q.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*]: *hd P-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 ≡ λ*y*. *y* ≠ *v0* ∧ (*y* = *new-last* ∨ (∃ *Q-hit* ∈ *Q*. *y* ∈ *set Q-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*: ∃ *y* ∈ *set P-k*. *hitting-Q-or-new-last y* **proof** (*cases*)
assume *P-k* ≠ *P-new*
then have *v1* ∈ *set P-k*
by (*metis P-k*(*1*) *insertE last-in-set path-from-toE paths paths-with-new-def*)
moreover have ∃ *Q-witness*. *Q-witness* ∈ *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*: ⋀*y*′. *y*′ ∈ *set P-k-pre* ⟹ ¬*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 ∃ *P-k-pre y P-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*
⋀*y*′. *y*′ ∈ *set P-k-pre* ⟹ ¬*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* ≠ *v0* **using** *hitting-Q-or-new-last-def y* **by** *auto*

lemma *P-k-pre-not-Nil*: *P-k-pre* ≠ *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*])) ∉ *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* ≡ *remove-vertex v0*

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

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

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

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

end

9 The case $y = new\_last$

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

We may assume $y \neq v1$ now because $[\![ProofStepInduct\text{-}NonTrivial\text{-}P\text{-}k\text{-}pre\ ?G\ ?v0.0\ ?v1.0\ ?paths\ ?P\text{-}new\ ?sep\text{-}size\ ?P\text{-}k\text{-}pre\ ?y\ ?P\text{-}k\text{-}post;\ ?y = ?v1.0]\!] \Longrightarrow \exists\, paths.\ DisjointPaths\ ?G\ ?v0.0\ ?v1.0\ paths \wedge card\ paths = Suc\ ?sep\text{-}size$ 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*: $y = new\text{-}last$
    and *optimal-paths*: $\bigwedge paths'\ P\text{-}new'.$
      $ProofStepInduct\ G\ v0\ v1\ paths'\ P\text{-}new'\ sep\text{-}size$
      $\Longrightarrow H.distance\ (last\ P\text{-}new)\ v1 \leq H.distance\ (last\ P\text{-}new')\ v1$
begin

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

definition R **where** $R \equiv$
  $SOME\ R.\ new\text{-}last \leadsto R \leadsto_H v1 \wedge (\forall R'.\ new\text{-}last \leadsto R' \leadsto_H v1 \longrightarrow length\ R \leq length\ R')$

lemma R: $new\text{-}last \leadsto R \leadsto_H v1\ \bigwedge R'.\ new\text{-}last \leadsto R' \leadsto_H v1 \Longrightarrow length\ R \leq length\ R'$ **proof**−
  obtain R' **where**
    R': $new\text{-}last \leadsto R' \leadsto_H v1\ \bigwedge R''.\ new\text{-}last \leadsto R'' \leadsto_H v1 \Longrightarrow length\ R' \leq length\ R''$
    using *arg-min-ex*[*OF new-last-to-v1*] **unfolding** *H-def* **by** *blast*
  then show $new\text{-}last \leadsto R \leadsto_H v1\ \bigwedge R'.\ new\text{-}last \leadsto R' \leadsto_H v1 \Longrightarrow length\ R \leq length\ R'$
    using *someI*[$of\ \lambda R.\ new\text{-}last \leadsto R \leadsto_H v1 \wedge (\forall R'.\ new\text{-}last \leadsto R' \leadsto_H v1 \longrightarrow length\ R \leq length\ R')$]
      R-def **by** *auto*
qed

lemma *v1-in-Q*: $\exists\, Q\text{-}hit \in Q.\ v1 \in set\ Q\text{-}hit$ **proof**−
  obtain *xs* **where** $xs \in Q$ **using** $Q(2)$ *sep-size-not0* **by** *fastforce*
  then show *?thesis* **using** *Q.paths last-in-set* **by** *blast*
qed

lemma *R-hits-Q*: $\exists\, z \in set\ R.\ Q.hitting\text{-}paths\ z$ **proof**−
  have $v1 \in set\ R$ **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 $Q.hitting\text{-}paths\ z$
      and $\bigwedge z'.\ z' \in set\ R\text{-}pre \Longrightarrow \neg Q.hitting\text{-}paths\ z'$
  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-pre z R-post*
assumes *R-decomp*: $R = R\text{-}pre\ @\ z\ \#\ R\text{-}post$
and *z*: *Q.hitting-paths z*
and *z-min*: $\bigwedge z'.\ z' \in set\ R\text{-}pre \Longrightarrow \neg Q.hitting\text{-}paths\ z'$
begin
private lemma *z-neq-v0*: $z \neq v0$ **using** *z Q.hitting-paths-def* **by** *auto*

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

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

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

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

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

private lemma *R′-walk-no-Q*: $[\![\ v \in set\ R'\text{-}walk;\ v \neq z\ ]\!] \Longrightarrow \neg Q.hitting\text{-}paths\ v$ **proof**−
fix *v* **assume** $v \in set\ R'\text{-}walk$ $v \neq z$
moreover have $v \in set\ P\text{-}k\text{-}pre \Longrightarrow \neg Q.hitting\text{-}paths\ v$
using *Q.hitting-paths-def hitting-Q-or-new-last-def y-min v1-in-Q* **by** *auto*
moreover have $v \in set\ R\text{-}pre \Longrightarrow \neg Q.hitting\text{-}paths\ v$ **using** *z-min* **by** *simp*
ultimately show $\neg Q.hitting\text{-}paths\ v$ **unfolding** *R′-walk-def* **using** *R′-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-new*.”

It does not say exactly why it contradicts the choice of *paths* and *P-new*. It seems we can choose Q together with *R′-walk* as our new paths plus extrapath. But this seems to be wrong because we cannot show that *R′-walk* is a path: *P-k-pre* and *R-pre* could intersect.

So we use $[\![walk\ ?xs;\ ?xs \neq [];\ hd\ ?xs = ?v;\ last\ ?xs = ?w]\!] \Longrightarrow \exists\, ys.\ ?v \leadsto ys \leadsto\ ?w \wedge set\ ys \subseteq set\ ?xs$ to transform *R′-walk* into a path R'.

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

private lemma *R′*: *hd* (*tl R′-walk*) ⇝*R′*⇝ *z set R′* ⊆ *set* (*tl R′-walk*) **proof**−
  have *tl R′-walk* ≠ *Nil* **by** (*simp add*: *P-k-pre-not-Nil R′-walk-def*)
  moreover have *last R′-walk* = *z* **unfolding** *R′-walk-def* **by** *simp*
  moreover have *walk* (*tl R′-walk*)
    by (*metis* (*no-types*, *lifting*) *path-from-toE walk-tl H-def P-k-decomp R′-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′-walk*) ⇝*R″*⇝ *z set R″* ⊆ *set* (*tl R′-walk*)
    using *walk-to-path*[*of tl R′-walk hd* (*tl R′-walk*) *z*] *last-tl* **by** *force*
  then show *hd* (*tl R′-walk*) ⇝*R′*⇝ *z set R′* ⊆ *set* (*tl R′-walk*) **unfolding** *R′-def*
    using *someI*[*of* λ*R′*. *hd* (*tl R′-walk*) ⇝*R′*⇝ *z* ∧ *set R′* ⊆ *set* (*tl R′-walk*)] **by** *auto*
qed

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

private lemma *R′-no-Q*: ⟦ *v* ∈ *set R′*; *v* ≠ *z* ⟧ ⟹ ¬*Q.hitting-paths v*
  using *R′-walk-no-Q* **by** (*meson R′*(*2*) *R′-walk-not-Nil list.set-sel*(*2*) *subsetCE*)

private lemma *v0-R′-path*: *v0* ⇝(*v0* # *R′*)⇝ *z* **proof**−
  have *v0*→*hd R′* **using** *hd-R′ hd-P-k-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* ∉ *set R′* **proof**−
    have *v0* ∉ *set R* **using** *R*(*1*) *H-def H.path-in-V remove-vertex-V*
      by (*simp add*: *path-from-to-def subset-Diff-insert*)
    then have *v0* ∉ *set R-pre* **using** *R-decomp* **by** *simp*
    moreover have *v0* ∉ *set* (*tl P-k-pre*) **using** *hd-P-k-v0 path-P-k path-first-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′-walk-def*
      using *P-k-pre-not-Nil z-neq-v0* **by** *auto*
  qed
  ultimately show *?thesis* **using** *path-cons*
    by (*metis R′*(*1*) *last.simps list.sel*(*1*) *list.simps*(*3*) *path-from-to-def*)
qed

private corollary *z-last-R′*: *z* = *last* (*v0* # *R′*) **using** *v0-R′-path* **by** *auto*

private lemma *z-eq-v1-solves*:
  assumes *z* = *v1*
  shows ∃*paths*. *DisjointPaths G v0 v1 paths* ∧ *card paths* = *Suc sep-size*
proof−

```
    interpret Q′: DisjointPaths G v0 v1 Q
      using DisjointPaths-supergraph H-x-def Q.DisjointPaths-axioms by auto
    have v0 ⇝(v0 # R′)⇝ v1 using assms v0-R′-path by auto
    moreover {
      fix xs v assume xs ∈ Q xs ≠ v0 # R′ v ∈ set xs v ∈ set (v0 # R′)
      then have v = v0 ∨ v = v1 using R′-no-Q Q.hitting-paths-def ‹z = v1› 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 ≠ v1
    shows ∃paths. DisjointPaths G v0 v1 paths ∧ 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 ⇝(v0 # R′)⇝ last (v0 # R′) using v0-R′-path by blast
          show tl (v0 # R′) ≠ [] using R′(1) by auto
          show hd (tl (v0 # R′)) ∉ 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 ∈ set (butlast (v0 # R′))
          then show ¬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 ≤ 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 ∃paths. DisjointPaths G v0 v1 paths ∧ 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:
  ∃paths. DisjointPaths G v0 v1 paths ∧ 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 $y \neq new\_last$

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

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

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

lemma *Q-hit-exists*: **obtains** *Q-hit Q-hit-pre Q-hit-post* **where**
  $Q\text{-}hit \in Q$ $y \in set\ Q\text{-}hit$ $Q\text{-}hit = Q\text{-}hit\text{-}pre\ @\ y\ \#\ Q\text{-}hit\text{-}post$
proof−
  obtain *Q-hit* **where** $Q\text{-}hit \in Q$ $y \in set\ Q\text{-}hit$
    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*: $Q\text{-}hit \in Q$ $y \in set\ Q\text{-}hit$
    and *Q-hit-decomp*: $Q\text{-}hit = Q\text{-}hit\text{-}pre\ @\ y\ \#\ Q\text{-}hit\text{-}post$
begin
  private lemma *Q-hit-v0-v1*: $v0 \leadsto Q\text{-}hit \leadsto_{H\text{-}x} v1$ **using** *Q.paths Q-hit*(*1*) **by** *blast*

  private lemma *Q-hit-vertices*: $set\ Q\text{-}hit \subseteq V - \{new\text{-}last\}$
    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*: $Q\text{-}hit\text{-}pre \neq Nil$
    using *Q-hit-v0-v1 y-neq-v0* **unfolding** *Q-hit-decomp* **by** *auto*

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

  private lemma *Q-hit-pre-edges*: $edges\text{-}of\text{-}walk\ (Q\text{-}hit\text{-}pre\ @\ [y]) \cap B \neq \{\}$ **proof**
    assume $edges\text{-}of\text{-}walk\ (Q\text{-}hit\text{-}pre\ @\ [y]) \cap B = \{\}$
    moreover have $edges\text{-}of\text{-}walk\ (Q\text{-}hit\text{-}pre\ @\ [y]) \subseteq E$
      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*:
      $edges\text{-}of\text{-}walk\ (Q\text{-}hit\text{-}pre\ @\ [y]) \subseteq \bigcup(edges\text{-}of\text{-}walk\ ‘\ paths\text{-}with\text{-}new)$
      unfolding *B-def* **by** *blast*
    then have ∗: $first\text{-}edge\text{-}of\text{-}walk\ (Q\text{-}hit\text{-}pre\ @\ [y]) \in \bigcup(edges\text{-}of\text{-}walk\ ‘\ paths\text{-}with\text{-}new)$
      using *tl-Q-hit-pre first-edge-in-edges* **by** *blast*

    define v' **where** $v' \equiv hd\ (tl\ (Q\text{-}hit\text{-}pre\ @\ [y]))$
    then have v': $(v0,\ v') = first\text{-}edge\text{-}of\text{-}walk\ (Q\text{-}hit\text{-}pre\ @\ [y])$
      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-hit′* ≡ *P-k-pre* @ *y* # *Q-hit-post*

private lemma *Q-hit′-v0-v1*: *v0* ⇝*Q-hit′*⇝ *v1* **proof**−
{
fix *v* **assume** *v* ∈ *set P-k-pre*
then have ¬*Q.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* ∉ *set Q-hit-post* **using** *Q.path-from-to-first′ Q-hit-v0-v1*
unfolding *Q-hit-decomp* **by** *blast*
ultimately have *v* ∉ *set Q-hit-post* **unfolding** *Q.hitting-paths-def*
using *Q-hit*(*1*) *Q-hit-decomp* **by** *auto*
}
then have *set P-k-pre* ∩ *set Q-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* ⇝*Q-hit′*⇝$_{H\text{-}x}$ *v1* **proof**−
have *new-last* ∉ *set P-k-pre* **using** *new-last-neq-v0 hitting-Q-or-new-last-def y-min* **by** *auto*
moreover have *new-last* ∉ *set Q-hit-post* **using** *Q-hit-vertices* **unfolding** *Q-hit-decomp* **by** *auto*
ultimately have *new-last* ∉ *set Q-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′* ≡ *insert Q-hit′* (*Q* − {*Q-hit*})

private lemma *Q-hit-edges-disjoint*:
⋃(*edges-of-walk* ‘ (*Q* − {*Q-hit*})) ∩ *edges-of-walk Q-hit* = {}
using *DiffD1 Q.paths-edge-disjoint Q-hit*(*1*) **by** *fastforce*

private lemma *Q-hit′-notin-Q-minus-Q-hit*: *Q-hit′* ∉ *Q* − {*Q-hit*} **proof**−
have *hd* (*tl Q-hit′*) ∉ *Q.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-weight Q′* < *Q-weight Q* **proof**−
define *Q-edges* **where** *Q-edges* ≡ ⋃(*edges-of-walk* ‘ *Q*) ∩ *B*
define *Q′-edges* **where** *Q′-edges* ≡ ⋃(*edges-of-walk* ‘ *Q′*) ∩ *B*
{
fix *v w* **assume** ∗: (*v*,*w*) ∈ *Q′-edges* (*v*,*w*) ∉ *Q-edges*
then have *v-w-in-B*: (*v*,*w*) ∈ *B* **unfolding** *Q′-edges-def* **by** *blast*

```
    obtain Q′-v-w-witness where Q′-v-w-witness:
      Q′-v-w-witness ∈ Q′ (v,w) ∈ edges-of-walk Q′-v-w-witness
      using ∗(1) unfolding Q′-edges-def by blast

    have Q′-v-w-witness ≠ 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]) ∪ 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) ∉ edges-of-walk (P-k-pre @ [y])
        using P-k-pre-edges v-w-in-B by blast
      moreover have (v,w) ∉ edges-of-walk (y # Q-hit-post) proof
        assume (v,w) ∈ edges-of-walk (y # Q-hit-post)
        then have (v,w) ∈ 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 ∈ 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 ∃ e ∈ Q-edges. e ∉ Q′-edges proof−
    obtain v w where v-w: (v,w) ∈ edges-of-walk (Q-hit-pre @ [y]) ∩ B
      using Q-hit-pre-edges by auto
    then have v-w-in-Q-hit: (v,w) ∈ edges-of-walk Q-hit ∩ B unfolding Q-hit-decomp
      by (metis Int-iff UnCI walk-edges-decomp)
    then have (v,w) ∈ Q-edges unfolding Q-edges-def using Q-hit(1) by blast
    moreover have (v,w) ∉ Q′-edges proof
      assume (v,w) ∈ Q′-edges
      then have (v,w) ∈ 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) ∈ 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 (⋃(edges-of-walk ‘ Q′) ∩ B) < card (⋃(edges-of-walk ‘ Q) ∩ 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:
    ⋀paths′ P-new′. ProofStepInduct G v0 v1 paths′ P-new′ sep-size
      ⟹ Digraph.distance (remove-vertex v0) (last P-new) v1
       ≤ Digraph.distance (remove-vertex v0) (last P-new′) v1
begin

lemma one-more-paths-exists-trivial:
```

new-last = *v1* ⟹ ∃ *paths*. *DisjointPaths G v0 v1 paths* ∧ *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* ≠ *v1*
shows ∃ *paths*. *DisjointPaths G v0 v1 paths* ∧ *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* ≠ *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* ≠ *v1 y* ≠ *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 ∃ *paths*. *DisjointPaths G v0 v1 paths* ∧ *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*:
∃ *paths*. *DisjointPaths G v0 v1 paths* ∧ *card paths* = *Suc sep-size*
proof−
define *paths-weight* **where** *paths-weight* ≡
λ(*paths*′ :: ′*a Walk set*, *P-new*′). *Digraph.distance* (*remove-vertex v0*) (*last P-new*′) *v1*
define *paths-good* **where** *paths-good* ≡
λ(*paths*′, *P-new*′). *ProofStepInduct G v0 v1 paths*′ *P-new*′ *sep-size*
have ∃ *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*′ ⋀*P*′′. *paths-good P*′′ ⟹ *paths-weight P*′ ≤ *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 $\bigwedge$*paths″ P-new″. paths-good* (*paths″*, *P-new″*)
$\Longrightarrow$ *paths-weight P′* $\leq$ *paths-weight* (*paths″*, *P-new″*) **by** (*simp add*: *P′*(*2*))
then have *optimal-paths-min*: $\bigwedge$*paths″ P-new″. ProofStepInduct G v0 v1 paths″ P-new″ sep-size*
$\Longrightarrow$ *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 paths′ P-new′ 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* $\Longrightarrow$ *card S* $\geq$ *n*
shows $\exists$ *paths. DisjointPaths G v0 v1 paths* $\wedge$ *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* $\Longrightarrow$ *n* $\leq$ *card S* **using** *Suc.prems Suc-leD* **by** *blast*
then obtain *paths* **where** *P*: *DisjointPaths G v0 v1 paths card paths* = *n* **using** *Suc*(*1*,*3*) **by** *blast*
interpret *G*: *DisjointPaths G v0 v1 paths* **using** *P*(*1*) **.**

obtain *P-new* **where**
P-new: *v0* $\leadsto$*P-new*$\leadsto_G$ *v1 set P-new* $\cap$ *G.second-vertices* = {}
using *G.disjoint-paths-new-path P*(*2*) *Suc.prems*(*1*) **by** *blast*
have *P-new-new*: *P-new* $\notin$ *paths*
by (*metis G.paths-tl-notnil G.second-vertex-def G.second-vertices-def G.path-from-toE IntI P-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-neq-v1* **by** *blast*
then have $\exists$ *x* $\in$ *set P-new. G.hitting-paths x* **using** *P-new*(*1*) **by** *fastforce*
then obtain *new-pre x new-post* **where**
P-new-decomp: *P-new* = *new-pre* @ *x* # *new-post*
and *x*: *G.hitting-paths x*
$\bigwedge$*y. y* $\in$ *set new-pre* $\Longrightarrow$ $\neg$*G.hitting-paths y*
by (*metis split-list-first-prop*)

have *1*: *DisjointPathsPlusOne G v0 v1 paths* (*new-pre* @ [*x*]) **proof**
show *v0* $\leadsto$(*new-pre* @ [*x*])$\leadsto_G$ *last* (*new-pre* @ [*x*]) **using** *P-new*(*1*)
by (*metis G.path-decomp′ P-new-decomp append-is-Nil-conv hd-append2 list.distinct*(*1*) *list.sel*(*1*) *path-from-to-def self-append-conv2*)
then show *tl* (*new-pre* @ [*x*]) $\neq$ []
by (*metis DisjointPaths.hitting-paths-def G.DisjointPaths-axioms G.path-from-toE butlast.simps*(*1*) *butlast-snoc list.distinct*(*1*) *list.sel*(*1*) *self-append-conv2 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 ⋀v. v ∈ set (butlast (new-pre @ [x])) ⟹ ¬G.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.prems(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 ⋀S. Separation G v0 v1 S ⟹ card S ≥ 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 $n \leq$ *card S* by defining an injective function from *paths* into *S*. Because we have *card 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** $f \equiv \lambda xs.\ SOME\ v.\ v \in S \wedge v \in set\ xs$

have *f-good*: $\bigwedge xs.\ xs \in paths \Longrightarrow f\ xs \in S \wedge f\ xs \in set\ xs$ **proof**−
  fix *xs* **assume** $xs \in paths$
  then obtain *v* **where** $v \in set\ xs \cap S$ **using** *S.S-separates paths* **by** *fastforce*
  then show $f\ xs \in S \wedge f\ xs \in set\ xs$ **unfolding** *f-def*
    using $someI[of\ \lambda v.\ v \in S \wedge v \in set\ xs\ v]$ **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 $*$: $xs \in paths\ ys \in paths\ f\ xs = f\ ys$
  then obtain *v* **where** $v \in S\ v \in set\ xs\ v \in set\ ys$
    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 *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*: $\bigwedge S.$
      $[\![\ S \subseteq V;\ v0 \notin S;\ v1 \notin S;\ \bigwedge xs.\ v0 \leadsto xs \leadsto v1 \Longrightarrow set\ xs \cap S \neq \{\}\ ]\!] \Longrightarrow card\ S \geq n$
  shows $\exists\, paths.\ card\ paths = n \wedge (\forall\, xs \in paths.$
    $v0 \leadsto xs \leadsto v1 \wedge (\forall\, ys \in paths - \{xs\}.\ (\forall\, v \in set\ xs \cap set\ ys.\ v = v0 \vee v = v1)))$

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* $\Longrightarrow$ *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*$\rightsquigarrow$ *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*$\rightsquigarrow$ *v1* $\Longrightarrow$ *set xs* $\cap$ *S* $\neq$ {} $\rrbracket$ $\Longrightarrow$ *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* $\Longrightarrow$ *v0* $\rightsquigarrow$*xs*$\rightsquigarrow$ *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*$\rightsquigarrow$ *v1* $\Longrightarrow$ *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.