

Tree Decompositions

Christoph Dittmann
christoph.dittmann@tu-berlin.de

April 14, 2026

We formalize tree decompositions and tree width in Isabelle/HOL, proving that trees have treewidth 1. We also show that every edge of a tree decomposition is a separation of the underlying graph. As an application of this theorem we prove that complete graphs of size n have treewidth $n - 1$.

Contents

1	Introduction	2
1.1	Avoid List Indices	2
1.2	Future Work	2
2	Graphs	3
2.1	Walks	3
2.2	Connectivity	5
2.3	Paths	5
2.4	Cycles	7
3	Trees	9
3.1	Unique Connecting Path	9
3.2	Separations	12
3.3	Rooted Trees	15
4	Tree Decompositions	16
4.1	Width of a Tree Decomposition	16
4.2	Treewidth of a Graph	17
4.3	Separations	18
5	Treewidth of Trees	21
6	Treewidth of Complete Graphs	23
7	Example Instantiations	25
	Bibliography	27

1 Introduction

We follow [1] in terms of the definition of tree decompositions and treewidth. We write a fairly minimal formalization of graphs and trees and then go straight to tree decompositions.

Let $G = (V, E)$ be a graph and (\mathcal{T}, β) be a tree decomposition, where \mathcal{T} is a tree and $\beta : V(\mathcal{T}) \rightarrow 2^V$ maps bags to sets of vertices. Our main theorem is that if $(s, t) \in V(\mathcal{T})$ is an edge of the tree decomposition, then $\beta(s) \cap \beta(t)$ is a separator of G , separating

$$\bigcup_{u \in V(\mathcal{T}) \text{ is in the left subtree of } \mathcal{T} \setminus (s, t)} \beta(u)$$

and

$$\bigcup_{u \in V(\mathcal{T}) \text{ is in the right subtree of } \mathcal{T} \setminus (s, t)} \beta(u).$$

As an application of this theorem we show that if K_n is the complete graph on n vertices, then the treewidth of K_n is $n - 1$.

Independent of this theorem, relying only on the basic definitions of tree decompositions, we also prove that trees have treewidth 1 if they have at least one edge (and treewidth 0 otherwise, which is trivial and holds for all graphs).

1.1 Avoid List Indices

While this will be obvious for more experienced Isabelle/HOL users, what we learned in this work is that working with lists becomes significantly easier if we avoid indices. It turns out that indices often trip up Isabelle's automatic proof methods. Rewriting a proof with list indices to a proof without often reduced the length of the proof by 50% or more.

For example, instead of saying “let $n \in \mathbb{N}$ be maximal such that the first n elements of the list all satisfy property P ”, it is better to say “let ps be a maximal prefix such that all elements of ps satisfy P ”.

1.2 Future Work

We have several ideas for future work. Let us enumerate them in order of ascending difficulty (subjectively, of course).

1. The easiest would be a formalization of the fact that treewidth is closed under minors and disjoint union, and that adding a single edge increases the treewidth by at most one. There are probably many more theorems similar to these.
2. A more interesting project would be a formalization of the cops and robber game for treewidth, where the number of cops is equivalent to the treewidth plus one. See [2] for a survey on these games.
3. Another interesting project would be a formal proof that the treewidth of a square grid is large. It seems reasonable to expect that this could profit from a formalization of cops and robber games, but it is no prerequisite.

4. An ambitious long-term project would be a full formalization of the grid theorem by Robertson and Seymour [4]. They showed that there exists a function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that for every $k \in \mathbb{N}$ it holds that if a graph has treewidth at least $f(k)$, then it contains a $k \times k$ grid as a minor.

Another more technical point would be to evaluate whether it would be good to use the “Graph Theory” library [3] from the Archive of Formal Proofs instead of reimplementing graphs here. At first glance it seems that the graph theory library would provide a lot of helpful lemmas. On the other hand, it would be a non-trivial dependency with its own idiosyncrasies, which could complicate the development of tree decomposition proofs. The author feels that overall it is probably a good idea to base this work on the graph theory library, but it needs further consideration.

2 Graphs

theory *Graph*

imports *Main* **begin**

'a is the vertex type.

type-synonym *'a Edge* = *'a* × *'a*

type-synonym *'a Walk* = *'a list*

record *'a Graph* =

verts :: *'a set* ($\langle V \rangle$)

arcs :: *'a Edge set* ($\langle E \rangle$)

abbreviation *is-arc* :: (*'a, 'b*) *Graph-scheme* \Rightarrow *'a* \Rightarrow *'a* \Rightarrow *bool* (**infixl** $\langle \rightarrow \rangle$ 60) **where**

$v \rightarrow_G w \equiv (v, w) \in E_G$

We only consider undirected finite simple graphs, that is, graphs without multi-edges and without loops.

locale *Graph* =

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

assumes *finite-vertex-set*: *finite V*

and *valid-edge-set*: $E \subseteq V \times V$

and *undirected*: $v \rightarrow w = w \rightarrow v$

and *no-loops*: $\neg v \rightarrow v$

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** $v \rightarrow w$ **shows** $v \in V$ $w \in V$

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

2.1 Walks

A walk is sequence of vertices connected by edges.

inductive *walk* :: *'a Walk* \Rightarrow *bool* **where**

Nil [*simp*]: *walk* []

| *Singleton* [*simp*]: $v \in V \Longrightarrow$ *walk* [*v*]

| *Cons*: $v \rightarrow w \Longrightarrow$ *walk* ($w \# vs$) \Longrightarrow *walk* ($v \# w \# vs$)

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

lemma *walk-2* [*simp*]: $v \rightarrow w \implies \text{walk } [v, w]$ **by** (*simp add: edges-are-in-V(2) walk.intros(3)*)

lemma *walk-comp*: $\llbracket \text{walk } xs; \text{walk } ys; xs = \text{Nil} \vee ys = \text{Nil} \vee \text{last } xs \rightarrow \text{hd } ys \rrbracket \implies \text{walk } (xs @ ys)$
by (*induct rule: walk.induct, simp-all add: walk.intros(3)*)
(*metis list.exhaust-sel walk.intros(2) walk.intros(3)*)

lemma *walk-tl*: $\text{walk } xs \implies \text{walk } (\text{tl } xs)$ **by** (*induct rule: walk.induct simp-all*)

lemma *walk-drop*: $\text{walk } xs \implies \text{walk } (\text{drop } n \text{ } xs)$ **by** (*induct n, simp*) (*metis drop-Suc tl-drop walk-tl*)

lemma *walk-take*: $\text{walk } xs \implies \text{walk } (\text{take } n \text{ } xs)$

by (*induct arbitrary: n rule: walk.induct*)
(*simp, metis Graph.walk.simps Graph-axioms take-Cons' take-eq-Nil,*
metis Graph.walk.simps Graph-axioms edges-are-in-V(1) take-Cons')

lemma *walk-rev*: $\text{walk } xs \implies \text{walk } (\text{rev } xs)$

by (*induct rule: walk.induct, simp, simp*)
(*metis Singleton edges-are-in-V(1) last-ConsL last-appendR list.sel(1)*
not-Cons-self2 rev.simps(2) undirected walk-comp)

lemma *walk-decomp*: **assumes** $\text{walk } (xs @ ys)$ **shows** $\text{walk } xs \text{ walk } ys$

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

lemma *walk-dropWhile*: $\text{walk } xs \implies \text{walk } (\text{dropWhile } f \text{ } xs)$ **by** (*simp add: walk-drop dropWhile-eq-drop*)

lemma *walk-takeWhile*: $\text{walk } xs \implies \text{walk } (\text{takeWhile } f \text{ } xs)$ **using** *walk-take takeWhile-eq-take* **by** *metis*

lemma *walk-in-V*: $\text{walk } xs \implies \text{set } xs \subseteq V$ **by** (*induct rule: walk.induct; simp add: edges-are-in-V*)

lemma *walk-first-edge*: $\text{walk } (v \# w \# xs) \implies v \rightarrow w$ **using** *walk.cases* **by** *fastforce*

lemma *walk-first-edge'*: $\llbracket \text{walk } (v \# xs); xs \neq \text{Nil} \rrbracket \implies v \rightarrow \text{hd } xs$
using *walk-first-edge* **by** (*metis list.exhaust-sel*)

lemma *walk-middle-edge*: $\text{walk } (xs @ v \# w \# ys) \implies v \rightarrow w$

by (*induct xs @ v \# w \# ys arbitrary: xs rule: walk.induct, simp, simp*)
(*metis list.sel(1,3) self-append-conv2 tl-append2*)

lemma *walk-last-edge*: $\llbracket \text{walk } (xs @ ys); xs \neq \text{Nil}; ys \neq \text{Nil} \rrbracket \implies \text{last } xs \rightarrow \text{hd } ys$

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*)

lemma *walk-takeWhile-edge*:

assumes $\text{walk } (xs @ [v]) \text{ } xs \neq \text{Nil} \text{ } \text{hd } xs \neq v$
shows $\text{last } (\text{takeWhile } (\lambda x. x \neq v) \text{ } xs) \rightarrow v$ (**is** *last ?xs → v*)

proof –

obtain xs' **where** $xs' : xs = ?xs @ xs'$ **by** (*metis takeWhile-dropWhile-id*)

thus *?thesis* **proof** (*cases*)

assume $xs' = \text{Nil}$ **thus** *?thesis* **using** xs' *assms(1,2) walk-last-edge* **by** *force*

next

assume $xs' \neq \text{Nil}$

hence $\text{hd } xs' = v$ **by** (*metis (full-types) hd-dropWhile same-append-eq takeWhile-dropWhile-id*

xs')

thus *?thesis* **by** (*metis <xs' ≠ []> append-Nil assms(1,3) walk-decomp(1) walk-last-edge xs'*)

qed

qed

2.2 Connectivity

definition *connected* :: 'a ⇒ 'a ⇒ bool (**infixl** <→* > 60) **where**

connected v w ≡ ∃ xs. walk xs ∧ xs ≠ Nil ∧ hd xs = v ∧ last xs = w

lemma *connectedI* [*intro*]: $\llbracket \text{walk } xs; xs \neq \text{Nil}; \text{hd } xs = v; \text{last } xs = w \rrbracket \implies v \rightarrow^* w$

unfolding *connected-def* **by** *blast*

lemma *connectedE*:

assumes $v \rightarrow^* w$

obtains *xs* **where** walk xs xs ≠ Nil hd xs = v last xs = w

using *assms* **that** **unfolding** *connected-def* **by** *blast*

lemma *connected-in-V*: **assumes** $v \rightarrow^* w$ **shows** $v \in V \ w \in V$

using *assms* **unfolding** *connected-def* **by** (*meson* *hd-in-set last-in-set subsetCE walk-in-V*)**+**

lemma *connected-refl*: $v \in V \implies v \rightarrow^* v$ **by** (*rule* *connectedI*[*of* [v]]) *simp-all*

lemma *connected-edge*: $v \rightarrow w \implies v \rightarrow^* w$ **by** (*rule* *connectedI*[*of* [v,w]]) *simp-all*

lemma *connected-trans*:

assumes $u \rightarrow v$ **and** $v \rightarrow w$

shows $u \rightarrow^* w$

proof–

obtain *xs* **where** $xs: \text{walk } xs \ xs \neq \text{Nil} \ \text{hd } xs = u \ \text{last } xs = v$ **using** *u-v connectedE* **by** *blast*

obtain *ys* **where** $ys: \text{walk } ys \ ys \neq \text{Nil} \ \text{hd } ys = v \ \text{last } ys = w$ **using** *v-w connectedE* **by** *blast*

let $?R = xs \ @ \ \text{tl } ys$

show *?thesis* **proof**

show walk ?R **using** *walk-comp*[*OF* *xs*(1)] **by** (*metis* *xs*(4) *ys*(1,2,3) *list.sel*(1,3) *walk.simps*)

show ?R ≠ Nil **by** (*simp* *add*: *xs*(2))

show hd ?R = u **by** (*simp* *add*: *xs*(2,3))

show last ?R = w **using** *xs*(2,4) *ys*(2,3,4)

by (*metis* *append-butlast-last-id last-append last-tl list.exhaust-sel*)

qed

qed

2.3 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 ⇒ bool **where** *path* xs ≡ walk xs ∧ *distinct* xs

lemma *path-singleton* [*simp*]: $v \in V \implies \text{path } [v]$ **by** *simp*

lemma *path-2* [*simp*]: $\llbracket v \rightarrow w; v \neq w \rrbracket \implies \text{path } [v,w]$ **by** *simp*

lemma *path-cons*: $\llbracket \text{path } xs; xs \neq \text{Nil}; v \rightarrow \text{hd } xs; v \notin \text{set } xs \rrbracket \implies \text{path } (v \ # \ xs)$

by (*metis* *distinct.simps*(2) *list.exhaust-sel walk.Cons*)

lemma *path-comp*: $\llbracket \text{walk } xs; \text{walk } ys; xs = \text{Nil} \vee ys = \text{Nil} \vee \text{last } xs \rightarrow \text{hd } ys; \text{distinct } (xs \ @ \ ys) \rrbracket$

$\implies \text{path } (xs \ @ \ ys)$ **using** *walk-comp* **by** *blast*

lemma *path-tl*: $\text{path } xs \implies \text{path } (\text{tl } xs)$ **by** (*simp* *add*: *distinct-tl walk-tl*)

lemma *path-drop*: $\text{path } xs \implies \text{path } (\text{drop } n \ xs)$ **by** (*simp* *add*: *walk-drop*)

lemma *path-take*: $\text{path } xs \implies \text{path } (\text{take } n \ xs)$ **by** (*simp* *add*: *walk-take*)

lemma *path-rev*: $\text{path } xs \implies \text{path } (\text{rev } xs)$ **by** (*simp* *add*: *walk-rev*)

lemma *path-decomp*: **assumes** $\text{path } (xs \ @ \ ys)$ **shows** $\text{path } xs \ \text{path } ys$

using *walk-decomp* *assms* *distinct-append* **by** *blast***+**

lemma *path-dropWhile*: $\text{path } xs \implies \text{path } (\text{dropWhile } f \ xs)$ **by** (*simp* *add*: *walk-dropWhile*)

lemma *path-takeWhile*: $\text{path } xs \implies \text{path } (\text{takeWhile } f \ xs)$ **by** (*simp* *add*: *walk-takeWhile*)

lemma *path-in-V*: $\text{path } xs \implies \text{set } xs \subseteq V$ **by** (*simp add: walk-in-V*)
lemma *path-first-edge*: $\text{path } (v \# w \# xs) \implies v \rightarrow w$ **using** *walk-first-edge* **by** *blast*
lemma *path-first-edge'*: $\llbracket \text{path } (v \# xs); xs \neq \text{Nil} \rrbracket \implies v \rightarrow \text{hd } xs$ **using** *walk-first-edge'* **by** *blast*
lemma *path-middle-edge*: $\text{path } (xs @ v \# w \# ys) \implies v \rightarrow w$ **using** *walk-middle-edge* **by** *blast*
lemma *path-takeWhile-edge*: $\llbracket \text{path } (xs @ [v]); xs \neq \text{Nil}; \text{hd } xs \neq v \rrbracket$
 $\implies \text{last } (\text{takeWhile } (\lambda x. x \neq v) xs) \rightarrow v$ **using** *walk-takeWhile-edge* **by** *blast*

end

We introduce shorthand notation for a path connecting two vertices.

definition *path-from-to* :: $(\text{'a}, \text{'b}) \text{ Graph-scheme} \Rightarrow \text{'a} \Rightarrow \text{'a} \text{ Walk} \Rightarrow \text{'a} \Rightarrow \text{bool}$
 $(\langle _ \rightsquigarrow _ \rightsquigarrow 1 _ \rangle [71, 71, 71] 70)$ **where**
 $\text{path-from-to } G \ v \ xs \ w \equiv \text{Graph.path } G \ xs \wedge xs \neq \text{Nil} \wedge \text{hd } xs = v \wedge \text{last } xs = w$

context *Graph* **begin**

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

Every walk contains a path connecting the same vertices.

lemma *walk-to-path*:
assumes $\text{walk } xs \ xs \neq \text{Nil} \ \text{hd } xs = v \ \text{last } xs = w$
shows $\exists ys. v \rightsquigarrow ys \rightsquigarrow w \wedge \text{set } ys \subseteq \text{set } xs$
proof–

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

define *target-set* **where** $\text{target-set} = \text{set } xs$
hence $\text{set } xs \subseteq \text{target-set}$ **by** *simp*
thus $\exists ys. v \rightsquigarrow ys \rightsquigarrow w \wedge \text{set } ys \subseteq \text{target-set}$
using *assms*
proof (*induct length xs arbitrary: xs rule: infinite-descent0*)
case (*smaller n*)
then obtain xs **where**
 $xs: n = \text{length } xs \ \text{walk } xs \ xs \neq \text{Nil} \ \text{hd } xs = v \ \text{last } xs = w \ \text{set } xs \subseteq \text{target-set}$ **and**
 $\text{hyp}: \neg(\exists ys. v \rightsquigarrow ys \rightsquigarrow w \wedge \text{set } ys \subseteq \text{target-set})$ **by** *blast*

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

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

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

obtain ys' *ys-suffix* **where**
 $ys\text{-decomp}: ys = ys' @ u \# ys\text{-suffix}$ **by** (*meson split-list xs-decomp(1)*)
define zs' **where** $zs' = ys' @ u \# zs$
have $\text{walk } zs'$ **unfolding** $zs'\text{-def}$ **using** $xs(2)$ $xs\text{-decomp}(3)$ $ys\text{-decomp}$
by (*metis walk-decomp list.sel(1) list.simps(3) walk-comp walk-last-edge*)
moreover have $\text{length } zs' < n$ **unfolding** $zs'\text{-def}$ **by** (*simp add: xs(1) xs-decomp(3) ys-decomp*)

```

moreover have  $hd\ zs' = v$  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  $set\ zs' \subseteq target-set$  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

```

corollary *connected-by-path*:

```

assumes  $v \rightarrow^* w$ 
obtains  $xs$  where  $v \rightsquigarrow xs \rightsquigarrow w$ 
using assms connected-def walk-to-path by blast

```

2.4 Cycles

A cycle in an undirected graph is a closed path with at least 3 different vertices. Closed paths with 0 or 1 vertex do not exist (graphs are loop-free), and paths with 2 vertices are not considered loops in undirected graphs.

definition *cycle* :: 'a Walk \Rightarrow bool **where**

```

cycle xs  $\equiv path\ xs \wedge length\ xs > 2 \wedge last\ xs \rightarrow hd\ xs$ 

```

lemma *cycleI* [*intro*]: $\llbracket path\ xs; length\ xs > 2; last\ xs \rightarrow hd\ xs \rrbracket \Longrightarrow cycle\ xs$

```

unfolding cycle-def by blast

```

lemma *cycleE*: $cycle\ xs \Longrightarrow path\ xs \wedge xs \neq Nil \wedge length\ xs > 2 \wedge last\ xs \rightarrow hd\ xs$

```

unfolding cycle-def by auto

```

We can now show a lemma that explains how to construct cycles from certain paths. If two paths both starting from v diverge immediately and meet again on their last vertices, then the graph contains a cycle with v on it.

Note that if two paths do not diverge immediately but only eventually, then *maximal-common-prefix* can be used to remove the common prefix.

lemma *meeting-paths-produce-cycle*:

```

assumes  $xs: path\ (v \# xs)\ xs \neq Nil$ 

```

```

  and  $ys: path\ (v \# ys)\ ys \neq Nil$ 

```

```

  and  $meet: last\ xs = last\ ys$ 

```

```

  and  $diverge: hd\ xs \neq hd\ ys$ 

```

```

shows  $\exists zs. cycle\ zs \wedge hd\ zs = v$ 

```

proof–

```

have  $set\ xs \cap set\ ys \neq \{\}$  using  $meet\ xs(2)\ ys(2)\ last-in-set$  by fastforce

```

```

then obtain  $xs'\ x\ xs''$  where  $xs: xs = xs' @ x \# xs''$   $set\ xs' \cap set\ ys = \{\}$   $x \in set\ ys$ 

```

```

  using split-list-first-prop[of xs  $\lambda x. x \in set\ ys$ ] by (metis disjoint-iff-not-equal)

```

```

then obtain  $ys'\ ys''$  where  $ys: ys = ys' @ x \# ys''$   $x \notin set\ ys'$ 

```

```

  using split-list-first-prop[of ys  $\lambda y. y = x$ ] by blast

```

```

let  $?zs = v \# xs' @ x \# (rev\ ys')$ 

```

```

have  $last\ ?zs \rightarrow hd\ ?zs$ 

```

```

  using undirected walk-first-edge walk-first-edge'  $ys'(1)\ ys(1)$  by (fastforce simp: last-rev)

```

```

moreover have  $path\ ?zs$  proof

```

```

  have  $walk\ (x \# rev\ ys')$  proof(cases)

```

```

    assume  $ys' = Nil$  thus  $?thesis$  using  $\langle last\ ?zs \rightarrow hd\ ?zs \rangle$  edges-are-in-V(1) by auto
next
    assume  $ys' \neq Nil$ 
    moreover hence  $last\ ys' \rightarrow x$  using walk-last-edge walk-tl ys'(1) ys(1) by fastforce
    moreover have  $hd\ (rev\ ys') = last\ ys'$  by (simp add:  $\langle ys' \neq [] \rangle$  hd-rev)
    moreover have  $walk\ (rev\ ys')$  by (metis list.sel(3) walk-decomp(1) walk-rev walk-tl ys'(1))
 $ys(1)$ 
    ultimately show  $walk\ (x \# rev\ ys')$  using path-cons undirected ys'(1) ys(1) by auto
qed
thus  $walk\ (v \# xs' @ x \# rev\ ys')$  using  $xs'(1)\ xs(1)$ 
by (metis append-Cons list.sel(1) list.simps(3) walk-comp walk-decomp(1) walk-last-edge)
next
show  $distinct\ (v \# xs' @ x \# rev\ ys')$  unfolding distinct-append distinct.simps(2) set-append
using  $xs'(1,2)\ xs(1)\ ys'(1)\ ys(1)$  by auto
qed
moreover have  $length\ ?zs \neq 2$  using diverge xs'(1) ys'(1) by auto
ultimately show  $?thesis$  using cycleI[of ?zs] by auto
qed

```

A graph with unique paths between every pair of connected vertices has no cycles.

lemma *unique-paths-implies-no-cycles:*

assumes *unique-paths:* $\bigwedge v\ w.\ v \rightarrow^* w \implies \exists !xs.\ v \rightsquigarrow xs \rightsquigarrow w$

shows $\bigwedge xs.\ \neg cycle\ xs$

proof

fix xs **assume** *cycle xs*

let $?v = hd\ xs$

let $?w = last\ xs$

let $?ys = [?v, ?w]$

define *good* **where** $good\ xs \iff ?v \rightsquigarrow xs \rightsquigarrow ?w$ **for** xs

have *path ?ys* **using** $\langle cycle\ xs \rangle$ *cycle-def no-loops undirected* **by** *auto*

hence *good ?ys* **unfolding** *good-def* **by** (*simp add: path-from-toI*)

moreover **have** *good xs* **unfolding** *good-def* **by** (*simp add: path-from-toI $\langle cycle\ xs \rangle$ cycleE*)

moreover **have** $?ys \neq xs$ **using** $\langle cycle\ xs \rangle$

by (*metis One-nat-def Suc-1 cycleE length-Cons less-not-refl list.size(3)*)

ultimately **have** $\neg(\exists !xs.\ good\ xs)$ **by** *blast*

moreover **have** *connected ?v ?w* **using** $\langle cycle\ xs \rangle$ *cycleE* **by** *blast*

ultimately **show** *False* **unfolding** *good-def* **using** *unique-paths* **by** *blast*

qed

A graph without cycles (also called a forest) has a unique path between every pair of connected vertices.

lemma *no-cycles-implies-unique-paths:*

assumes *no-cycles:* $\bigwedge xs.\ \neg cycle\ xs$ **and** *connected:* $v \rightarrow^* w$

shows $\exists !xs.\ v \rightsquigarrow xs \rightsquigarrow w$

proof (*rule ex-ex1I*)

show $\exists xs.\ v \rightsquigarrow xs \rightsquigarrow w$ **using** *connected connected-by-path* **by** *blast*

next

fix $xs\ ys$

assume $v \rightsquigarrow xs \rightsquigarrow w\ v \rightsquigarrow ys \rightsquigarrow w$

hence *xs-valid:* $path\ xs\ xs \neq Nil\ hd\ xs = v\ last\ xs = w$

and *ys-valid:* $path\ ys\ ys \neq Nil\ hd\ ys = v\ last\ ys = w$ **by** *blast+*

```

show  $xs = ys$  proof (rule ccontr)
  assume  $xs \neq ys$ 
  hence  $\exists ps\ xs'\ ys'. xs = ps @ xs' \wedge ys = ps @ ys' \wedge (xs' = Nil \vee ys' = Nil \vee hd\ xs' \neq hd\ ys')$ 
    by (induct  $xs\ ys$  rule: list-induct2', blast, blast, blast)
    (metis (no-types, opaque-lifting) append-Cons append-Nil list.sel(1))
  then obtain  $ps\ xs'\ ys'$  where
     $ps: xs = ps @ xs'\ ys = ps @ ys'\ xs' = Nil \vee ys' = Nil \vee hd\ xs' \neq hd\ ys'$  by blast

  have  $last\ xs \in set\ ps$  if  $xs' = Nil$  using  $xs\ valid(2)\ ps(1)$  by (simp add: that)
  hence  $xs\ not\ nil: xs' \neq Nil$  using  $\langle xs \neq ys \rangle\ ys\ valid(1,4)\ ps(1,2)\ xs\ valid(4)$  by auto

  have  $last\ ys \in set\ ps$  if  $ys' = Nil$  using  $ys\ valid(2)\ ps(2)$  by (simp add: that)
  hence  $ys\ not\ nil: ys' \neq Nil$  using  $\langle xs \neq ys \rangle\ xs\ valid(1,4)\ ps(1,2)\ ys\ valid(4)$  by auto

  have  $\exists zs. cycle\ zs$  proof—
    let  $?v = last\ ps$ 
    have  $*$ :  $ps \neq Nil$  using  $xs\ valid(2,3)\ ys\ valid(2,3)\ ps(1,2,3)$  by auto
    have  $path\ (?v \# xs')$  using  $xs\ valid(1)\ ps(1) * walk\ decomp(2)$ 
      by (metis append-Cons append-assoc append-butlast-last-id distinct-append self-append-conv2)
    moreover have  $path\ (?v \# ys')$  using  $ys\ valid(1)\ ps(2) * walk\ decomp(2)$ 
      by (metis append-Cons append-assoc append-butlast-last-id distinct-append self-append-conv2)
    moreover have  $last\ xs' = last\ ys'$ 
      using  $xs\ valid(4)\ ys\ valid(4)\ xs\ not\ nil\ ys\ not\ nil\ ps(1,2)$  by auto
    ultimately show  $?thesis$  using  $ps(3)\ meeting\ paths\ produce\ cycle\ xs\ not\ nil\ ys\ not\ nil$  by
blast
  qed
  thus  $False$  using  $no\ cycles$  by blast
qed
qed
end — locale Graph
end

```

3 Trees

```

theory Tree
imports Graph begin

```

A tree is a connected graph without cycles.

```

locale Tree = Graph +
  assumes  $connected: \llbracket v \in V; w \in V \rrbracket \implies v \rightarrow^* w$  and  $no\ cycles: \neg cycle\ xs$ 
begin

```

3.1 Unique Connecting Path

For every pair of vertices in a tree, there exists a unique path connecting these two vertices.

```

lemma  $unique\ connecting\ path: \llbracket v \in V; w \in V \rrbracket \implies \exists! xs. v \rightsquigarrow_{xs} w$ 
  using  $connected\ no\ cycles\ no\ cycles\ implies\ unique\ paths$  by blast

```

Let us define a function mapping pair of vertices to their unique connecting path.

end — locale Tree

definition *unique-connecting-path* :: ('a, 'b) Graph-scheme \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a Walk
(**infix** $\langle \rightsquigarrow_1 \rangle$ 71) **where** *unique-connecting-path* G v w \equiv THE xs. v \rightsquigarrow_{xs} w

We defined this outside the locale in order to be able to use the index in the shorthand syntax $v \rightsquigarrow_{\text{some-index}} w$.

context Tree **begin**

lemma *unique-connecting-path-set*:

assumes v \in V w \in V

shows v \in set (v \rightsquigarrow w) w \in set (v \rightsquigarrow w)

using theI'[OF *unique-connecting-path*[OF *assms*], *folded unique-connecting-path-def*]
hd-in-set last-in-set **by** fastforce+

lemma *unique-connecting-path-properties*:

assumes v \in V w \in V

shows path (v \rightsquigarrow w) v \rightsquigarrow w \neq Nil hd (v \rightsquigarrow w) = v last (v \rightsquigarrow w) = w

using theI'[OF *unique-connecting-path*[OF *assms*], *folded unique-connecting-path-def*] **by** blast+

lemma *unique-connecting-path-unique*:

assumes v \rightsquigarrow_{xs} w

shows xs = v \rightsquigarrow w

proof —

have v \in V w \in V **using** *assms connected-in-V* **by** blast+

with *unique-connecting-path-properties*[OF *this*] **show** ?thesis

using *assms unique-connecting-path* **by** blast

qed

corollary *unique-connecting-path-connects*: $\llbracket v \in V; w \in V \rrbracket \Longrightarrow v \rightsquigarrow (v \rightsquigarrow w) \rightsquigarrow w$

using *unique-connecting-path unique-connecting-path-unique* **by** blast

lemma *unique-connecting-path-rev*:

assumes v \in V w \in V

shows v \rightsquigarrow w = rev (w \rightsquigarrow v)

proof —

have v \rightsquigarrow (rev (w \rightsquigarrow v)) \rightsquigarrow w **using** *assms*

by (*simp add: unique-connecting-path-properties walk-rev hd-rev last-rev path-from-toI*)

thus ?thesis **using** *unique-connecting-path-unique* **by** *simp*

qed

lemma *unique-connecting-path-decomp*:

assumes v \in V w \in V v \rightsquigarrow w = ps @ u # ps'

shows ps @ [u] = v \rightsquigarrow u u # ps' = u \rightsquigarrow w

proof —

have hd (ps @ [u]) = v

by (*metis append-Nil assms hd-append2 list.sel(1) unique-connecting-path-properties(3)*)

moreover **have** path (ps @ [u]) **using** *unique-connecting-path-properties(1)*[OF *assms(1,2)*]

unfolding *assms(3)*

by (*metis distinct.simps(2) distinct1-rotate list.sel(1) list.simps(3) not-distinct-conv-prefix*

path-decomp(1) rev.simps(2) rotate1.simps(2) walk-comp walk-decomp(2) walk-last-edge

walk-rev)

moreover **have** last (ps @ [u]) = u ps @ [u] \neq Nil **by** *simp-all*

ultimately show $ps @ [u] = v \rightsquigarrow u$ **using** *unique-connecting-path-unique* **by** *blast*
next
have $last (u \# ps') = w$
using *assms unique-connecting-path-properties(4)* **by** *fastforce*
moreover have $path (u \# ps')$ **using** *unique-connecting-path-properties(1)[OF assms(1,2)]*
unfolding *assms(3)* **using** *path-decomp(2)* **by** *blast*
moreover have $hd (u \# ps') = u \ u \# ps' \neq Nil$ **by** *simp-all*
ultimately show $u \# ps' = u \rightsquigarrow w$ **using** *unique-connecting-path-unique* **by** *blast*
qed

lemma *unique-connecting-path-tl:*

assumes $v \in V \ u \in set (w \rightsquigarrow v) \ u \rightarrow w$

shows $tl (w \rightsquigarrow v) = u \rightsquigarrow v$

proof (*rule ccontr*)

assume *contra: $\neg ?thesis$*

from *assms(2)* **obtain** $ps \ ps'$ **where**

$ps: w \rightsquigarrow v = ps @ u \# ps'$ **by** (*meson split-list*)

have $cycle (ps @ [u])$ **proof**

show $path (ps @ [u])$ **using** *unique-connecting-path-decomp assms(1,3) ps*

by (*metis edges-are-in-V unique-connecting-path-properties(1)*)

show $length (ps @ [u]) > 2$ **proof** (*rule ccontr*)

assume *$\neg ?thesis$*

moreover have $u \neq w$ **using** *assms(3) no-loops* **by** *blast*

ultimately have $length (ps @ [u]) = 2$

by (*metis edges-are-in-V(2) assms(1,3) hd-append length-0-conv length-append-singleton less-2-cases linorder-neqE-nat list.sel(1) nat.simps(1) ps snoc-eq-iff-butlast unique-connecting-path-properties(3)*)

hence $tl (w \rightsquigarrow v) = u \# ps'$

by (*metis One-nat-def Suc-1 append-Nil diff-Suc-1 length-0-conv length-Cons length-append-singleton list.collapse nat.simps(3) ps tl-append2*)

moreover have $u \# ps' = u \rightsquigarrow v$

using *unique-connecting-path-decomp assms(1,3) edges-are-in-V(2) ps* **by** *blast*

ultimately show *False* **using** *contra* **by** *simp*

qed

show $last (ps @ [u]) \rightarrow hd (ps @ [u])$ **using** *assms(3)*

by (*metis edges-are-in-V(2) unique-connecting-path-properties(3) assms(1) hd-append list.sel(1) ps snoc-eq-iff-butlast*)

qed

thus *False* **using** *no-cycles* **by** *auto*

qed

Every tree with at least two vertices contains an edge.

lemma *tree-has-edge:*

assumes $card \ V > 1$

shows $\exists v \ w. \ v \rightarrow w$

proof–

obtain v **where** $v: v \in V$ **using** *assms*

by (*metis List.finite-set One-nat-def card.empty card-mono empty-set less-le-trans linear not-less subsetI zero-less-Suc*)

then obtain w **where** $w \in V \ v \neq w$ **using** *assms*

by (*metis (no-types, lifting) One-nat-def card.empty card.insert distinct.simps(2) empty-set finite.intros(1) finite-distinct-list finite-vertex-set hd-in-set last.simps last-in-set*)

```

    less-or-eq-imp-le list.exhaust-sel list.simps(15) not-less path-singleton)
  hence  $v \rightarrow hd (tl (v \rightsquigarrow w))$  using  $v$ 
  by (metis unique-connecting-path-properties last.simps list.exhaust-sel walk-first-edge')
  thus ?thesis by blast
qed

```

3.2 Separations

Removing a single edge always splits a tree into two subtrees. Here we define the set of vertices of the left subtree. The definition may not be obvious at first glance, but we will soon prove that it behaves as expected. We say that a vertex u is in the left subtree if and only if the unique path from u to t visits s .

```

definition left-tree :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'a set where
  left-tree s t  $\equiv$  {  $u \in V. s \in set (u \rightsquigarrow t)$  }
lemma left-treeI [intro]:  $\llbracket u \in V; s \in set (u \rightsquigarrow t) \rrbracket \Longrightarrow u \in left-tree s t$ 
  unfolding left-tree-def by blast
lemma left-treeE:  $u \in left-tree s t \Longrightarrow u \in V \wedge s \in set (u \rightsquigarrow t)$ 
  unfolding left-tree-def by blast

```

```

lemma left-tree-in-V:  $left-tree s t \subseteq V$  unfolding left-tree-def by blast
lemma left-tree-initial:  $\llbracket s \in V; t \in V \rrbracket \Longrightarrow s \in left-tree s t$ 
  unfolding left-tree-def by (simp add: unique-connecting-path-set(1))
lemma left-tree-initial':  $\llbracket s \in V; t \in V; s \neq t \rrbracket \Longrightarrow t \notin left-tree s t$ 
  by (metis distinct.simps(2) last.simps left-treeE list.discI list.sel(1) path-from-toI
    path-singleton set-ConsD unique-connecting-path-unique)
lemma left-tree-initial-edge:  $s \rightarrow t \Longrightarrow t \notin left-tree s t$ 
  using edges-are-in-V(1) left-tree-initial' no-loops undirected by blast

```

The union of the left and right subtree is V .

```

lemma left-tree-union-V:
  assumes  $s \rightarrow t$ 
  shows  $left-tree s t \cup left-tree t s = V$ 
proof
  show  $left-tree s t \cup left-tree t s \subseteq V$  using left-tree-in-V by auto
  {
    have  $s: s \in V$  and  $t: t \in V$  using assms using edges-are-in-V by blast+
  }

```

Assume to the contrary that $u \in V$ is in neither part.

```

fix  $u$  assume  $u: u \in V \ u \notin left-tree s t \ u \notin left-tree t s$ 

```

Then we can construct two different paths from s to u , which, in a tree, is a contradiction. First, we get paths from s to u and from t to u .

```

let ?xs =  $s \rightsquigarrow u$ 
let ?ys =  $t \rightsquigarrow u$ 
have  $t \notin set ?xs$  using  $u(1,3)$  unfolding left-tree-def
  by (metis (no-types, lifting) unique-connecting-path-rev mem-Collect-eq s set-rev)
have  $s \notin set ?ys$  using  $u(1,2)$  unfolding left-tree-def
  by (metis (no-types, lifting) unique-connecting-path-rev mem-Collect-eq set-rev t)

```

Now we can define two different paths from s to u .

```

define  $xs'$  where [simp]:  $xs' = ?xs$ 
define  $ys'$  where [simp]:  $ys' = s \# ?ys$ 

have  $path\ ys'$  using  $path-cons\ \langle s \notin set\ ?ys \rangle\ assms$ 
  by (simp add: unique-connecting-path-properties(1-3)) t u(1)
moreover have  $path\ xs'\ xs' \neq []\ ys' \neq []\ hd\ xs' = s\ last\ xs' = u$ 
  by (simp-all add: unique-connecting-path-properties s u(1))
moreover have  $hd\ ys' = s\ last\ ys' = u$ 
  by simp (simp add: unique-connecting-path-properties(2,4)) t u(1)
moreover have  $xs' \neq ys'$  using unique-connecting-path-set(1)  $\langle t \notin set\ ?xs \rangle\ t\ u(1)$  by auto

```

The existence of two different paths is a contradiction.

```

  ultimately have False using unique-connecting-path-unique by blast
}
thus  $V \subseteq left-tree\ s\ t \cup left-tree\ t\ s$  by blast
qed

```

The left and right subtrees are disjoint.

```

lemma left-tree-disjoint:
  assumes  $s \rightarrow t$ 
  shows  $left-tree\ s\ t \cap left-tree\ t\ s = \{\}$ 
proof (rule ccontr)
  assume  $\neg ?thesis$ 
  then obtain  $u$  where  $u: u \in V\ s \in set\ (u \rightsquigarrow t)\ t \in set\ (u \rightsquigarrow s)$  using left-treeE by blast

```

```

have  $s: s \in V$  and  $t: t \in V$  using assms edges-are-in-V by blast+

```

```

obtain  $ps\ ps'$  where  $ps: u \rightsquigarrow t = ps @ s \# ps'$  by (meson split-list u(2))
hence  $ps' \neq Nil$ 
  using assms last-snoc no-loops unique-connecting-path-properties(4)[OF u(1) t] by auto
hence *:  $length\ (ps @ [s]) < length\ (u \rightsquigarrow t)$  by (simp add: ps)

```

```

have  $ps': ps @ [s] = u \rightsquigarrow s$  using ps unique-connecting-path-decomp t u(1) by blast

```

```

then obtain  $qs\ qs'$  where  $qs: ps @ [s] = qs @ t \# qs'$  using split-list[OF u(3)] by auto
hence  $qs' \neq Nil$  using assms last-snoc no-loops by auto
hence **:  $length\ (qs @ [t]) < length\ (ps @ [s])$  by (simp add: qs)

```

```

have  $qs @ [t] = u \rightsquigarrow t$  using qs ps' unique-connecting-path-decomp s u(1) by metis
thus False using less-trans[OF ** *] by simp

```

qed

The path from a vertex in the left subtree to a vertex in the right subtree goes through s . In other words, an edge $s \rightarrow t$ is a separator in a tree.

```

theorem left-tree-separates:
  assumes  $st: s \rightarrow t$  and  $u: u \in left-tree\ s\ t$  and  $u': u' \in left-tree\ t\ s$ 
  shows  $s \in set\ (u \rightsquigarrow u')$ 
proof (rule ccontr)
  assume  $\neg ?thesis$ 
  with assms have  $set\ (u \rightsquigarrow u') \subseteq left-tree\ s\ t$ 
  proof(induct  $u \rightsquigarrow u'$  arbitrary:  $u\ u'$ )

```

```

case Nil thus ?case using unique-connecting-path-properties(2) by auto
next
case (Cons x xs u u')
have x = u using Cons.hyps(2) Cons.prem(2,3)
by (metis left-treeE list.sel(1) unique-connecting-path-properties(3))
hence u → hd xs using Cons.hyps(2) Cons.prem(2,3) st
by (metis IntI left-tree-disjoint distinct.simps(2) last.simps left-treeE list.set(1)
    unique-connecting-path-properties(1,4) walk-first-edge')
hence u ∈ V hd xs ∈ V using edges-are-in-V by blast+
have *: xs = hd xs ∼ u'
by (metis Cons.hyps(2) Cons.prem(2,3) IntI left-tree-disjoint distinct.simps(2) last.simps
    left-treeE list.sel(1,3) list.set(1) path-from-toI st
    unique-connecting-path-properties(1,3,4) unique-connecting-path-unique walk-tl)
moreover hence s ∉ set (hd xs ∼ u') using Cons.hyps(2) Cons.prem(4)
by (metis list.set-intros(2))
moreover have hd xs ∈ left-tree s t proof (rule ccontr)
  assume ¬?thesis
  hence hd xs ∈ left-tree t s using ⟨hd xs ∈ V⟩ st left-tree-union-V by fastforce
  hence t ∈ set (hd xs ∼ s) using left-treeE by blast
  let ?ys' = hd xs ∼ s
  let ?ys = u # ?ys'
  have u ∉ set ?ys' proof
    assume u ∈ set ?ys'
    hence tl ?ys' = u ∼ s
      using unique-connecting-path-tl ⟨u → hd xs⟩ edges-are-in-V(1) st by auto
  moreover have t ≠ hd xs proof
    let ?ys = [u, hd xs]
    have t ≠ u using Cons.prem(2) left-tree-initial-edge st by blast
    assume t = hd xs
    hence ?ys = u ∼ t
      using unique-connecting-path-unique[of u ?ys hd xs] ⟨u → hd xs⟩ ⟨t ≠ u⟩
      by (simp add: path-from-toI)
    hence s ∉ set (u ∼ t)
      by (metis Cons.hyps(2) Cons.prem(4) ⟨t = hd xs⟩ ⟨x = u⟩ distinct.simps(2)
        distinct-singleton list.set-intros(1) no-loops set-ConsD st)
  thus False using Cons.prem(2) left-treeE by blast
qed
ultimately have t ∈ set (u ∼ s) using ⟨t ∈ set ?ys'⟩ ⟨hd xs ∈ V⟩ st
by (metis edges-are-in-V(1) unique-connecting-path-properties(2,3) list.collapse set-ConsD)
thus False using Cons.prem(2) st ⟨u ∈ V⟩
by (meson left-tree-disjoint disjoint-iff-not-equal left-treeI)
qed
hence path ?ys using path-cons ⟨u → hd xs⟩
by (metis unique-connecting-path-properties(1-3) edges-are-in-V st)
moreover have ?ys ≠ Nil hd ?ys = u by simp-all
moreover have last ?ys = s using st unique-connecting-path-properties(2,4) ⟨hd xs ∈ V⟩
by (simp add: edges-are-in-V(1))
ultimately have ?ys = u ∼ s using unique-connecting-path-unique by blast
hence t ∈ set (u ∼ s) by (metis ⟨t ∈ set ?ys'⟩ list.set-intros(2))
thus False using Cons.prem(2) ⟨u ∈ V⟩ st
by (meson left-tree-disjoint disjoint-iff-not-equal left-treeI)
qed

```

```

ultimately have set (hd xs  $\rightsquigarrow$  u')  $\subseteq$  left-tree s t
  using Cons.hyps(1) st Cons.prem(3) by blast
hence set xs  $\subseteq$  left-tree s t using * by simp
thus ?case using Cons.hyps(2) Cons.prem(2,3)
  by (metis insert-subset left-treeE list.sel(1) list.set(2) unique-connecting-path-properties(3))
qed
hence u'  $\in$  left-tree s t using left-treeE u u' unique-connecting-path-set(2) by auto
thus False by (meson left-tree-disjoint disjoint-iff-not-equal st u')
qed

```

By symmetry, the path also visits t .

```

corollary left-tree-separates':
  assumes s  $\rightarrow$  t u  $\in$  left-tree s t u'  $\in$  left-tree t s
  shows t  $\in$  set (u  $\rightsquigarrow$  u')
  using assms left-tree-separates by (metis left-treeE set-rev undirected unique-connecting-path-rev)

```

end — locale Tree

3.3 Rooted Trees

A rooted tree is a tree with a distinguished vertex called root.

```

locale RootedTree = Tree +
  fixes root :: 'a
  assumes root-in-V: root  $\in$  V
begin

```

In a rooted tree, we can define the parent relation.

```

definition parent :: 'a  $\Rightarrow$  'a where
  parent v  $\equiv$  hd (tl (v  $\rightsquigarrow$  root))

```

```

lemma parent-edge:  $\llbracket v \in V; v \neq \text{root} \rrbracket \Longrightarrow v \rightarrow \text{parent } v$  unfolding parent-def
  by (metis last.simps list.exhaust-sel root-in-V unique-connecting-path-properties walk-first-edge')

```

```

lemma parent-edge-root: v  $\rightarrow$  root  $\Longrightarrow$  parent v = root unfolding parent-def
  by (metis edges-are-in-V(1) path-from-toE undirected unique-connecting-path
    unique-connecting-path-set(2) unique-connecting-path-tl unique-connecting-path-unique)

```

```

lemma parent-in-V:  $\llbracket v \in V; v \neq \text{root} \rrbracket \Longrightarrow \text{parent } v \in V$ 
  using parent-edge edges-are-in-V(2) by blast

```

```

lemma parent-edge-cases: v  $\rightarrow$  w  $\Longrightarrow$  w = parent v  $\vee$  v = parent w unfolding parent-def
  by (metis Un-iff edges-are-in-V(1) left-tree-initial left-tree-separates' left-tree-union-V
    root-in-V undirected unique-connecting-path-properties(3) unique-connecting-path-tl)

```

```

lemma sibling-path:
  assumes v: v  $\in$  V v  $\neq$  root and w: w  $\in$  V w  $\neq$  root and vw: v  $\neq$  w parent v = parent w
  shows v  $\rightsquigarrow$  w = [v, parent v, w] (is - = ?xs)

```

proof—

```

  have path ?xs using v w vw

```

```

    by (metis distinct-length-2-or-more distinct-singleton no-loops parent-edge undirected
      walk.Cons walk-2)

```

```

    thus ?thesis using unique-connecting-path-unique by fastforce

```

qed

end — locale RootedTree

end

4 Tree Decompositions

theory *TreeDecomposition*
imports *Tree* **begin**

A tree decomposition of a graph.

locale *TreeDecomposition* = *Graph* *G* + *T*: *Tree* *T*
for *G* :: ('a, 'b) *Graph-scheme* (**structure**) **and** *T* :: ('c, 'd) *Graph-scheme* +
fixes *bag* :: 'c \Rightarrow 'a *set*
assumes
— Every vertex appears somewhere
bags-union: $\bigcup \{ \text{bag } t \mid t. t \in V_T \} = V$
— Every edge is covered
and *bags-edges*: $v \rightarrow w \implies \exists t \in V_T. v \in \text{bag } t \wedge w \in \text{bag } t$
— Every vertex appearing in *s* and *u* also appears in every bag on the path connecting *s* and *u*
and *bags-continuous*: $\llbracket s \in V_T; u \in V_T; t \in \text{set } (s \rightsquigarrow_T u) \rrbracket \implies \text{bag } s \cap \text{bag } u \subseteq \text{bag } t$
begin

Following the usual literature, we will call elements of *V* vertices and elements of *V_T* bags (or nodes) from now on.

4.1 Width of a Tree Decomposition

We define the width of this tree decomposition as the size of the largest bag minus 1.

abbreviation *bag-cards* $\equiv \{ \text{card } (\text{bag } t) \mid t. t \in V_T \}$
definition *max-bag-card* $\equiv \text{Max } \text{bag-cards}$

We need a special case for $V_T = \{\}$ because in this case *max-bag-card* is not well-defined.

definition *width* $\equiv \text{if } V_T = \{\} \text{ then } 0 \text{ else } \text{max-bag-card} - 1$

lemma *bags-in-V*: $t \in V_T \implies \text{bag } t \subseteq V$ **using** *bags-union* *Sup-upper mem-Collect-eq* **by** *blast*
lemma *bag-finite*: $t \in V_T \implies \text{finite } (\text{bag } t)$ **using** *bags-in-V* *finite-subset* *finite-vertex-set* **by** *blast*
lemma *bag-bound-V*: $t \in V_T \implies \text{card } (\text{bag } t) \leq \text{card } V$ **by** (*simp add: bags-in-V* *card-mono* *finite-vertex-set*)
lemma *bag-bound-V-empty*: $\llbracket V = \{\}; t \in V_T \rrbracket \implies \text{card } (\text{bag } t) = 0$ **using** *bag-bound-V* **by** *auto*
lemma *empty-tree-empty-V*: $V_T = \{\} \implies V = \{\}$ **using** *bags-union* **by** *simp*
lemma *bags-exist*: $v \in V \implies \exists t \in V_T. v \in \text{bag } t$ **using** *bags-union* **using** *UnionE* *mem-Collect-eq* **by** *auto*

The width is never larger than the number of vertices, and if there is at least one vertex in the graph, then it is always smaller. This is trivially true because a bag contains at most all of *V*. However, the proof is not fully trivial because we also need to show that *width* is well-defined.

lemma *bag-cards-finite*: *finite* *bag-cards* **using** *T.finite-vertex-set* **by** *simp*
lemma *bag-cards-nonempty*: $V \neq \{\} \implies \text{bag-cards} \neq \{\}$
using *bag-cards-finite* *empty-tree-empty-V* *empty-Collect-eq* *ex-in-conv* **by** *blast*

lemma *max-bag-card-in-bag-cards*: $V \neq \{\}$ \implies $\text{max-bag-card} \in \text{bag-cards}$ **unfolding** *max-bag-card-def*
using *Max-in bag-cards-finite bag-cards-nonempty* **by** *auto*
lemma *max-bag-card-lower-bound-bag*: $t \in V_T \implies \text{max-bag-card} \geq \text{card } (\text{bag } t)$
by (*metis (mono-tags, lifting) Max-ge bag-cards-finite max-bag-card-def mem-Collect-eq*)
lemma *max-bag-card-lower-bound-1*: **assumes** $V \neq \{\}$ **shows** $\text{max-bag-card} > 0$ **proof**–
have $\exists v \in V. \exists t \in V_T. v \in \text{bag } t$ **using** $\langle V \neq \{\} \rangle$ *bags-union* **by** *blast*
thus $\text{max-bag-card} > 0$ **unfolding** *max-bag-card-def* **using** *bag-finite*
card-gt-0-iff emptyE Max-gr-iff[OF bag-cards-finite bag-cards-nonempty[OF assms]] **by** *auto*
qed
lemma *max-bag-card-upper-bound-V*: $V \neq \{\} \implies \text{max-bag-card} \leq \text{card } V$ **unfolding** *max-bag-card-def*
using *Max-le-iff[OF bag-cards-finite bag-cards-nonempty] bag-bound-V* **by** *blast*

lemma *width-upper-bound-V*: $V \neq \{\} \implies \text{width} < \text{card } V$ **unfolding** *width-def*
using *max-bag-card-upper-bound-V max-bag-card-lower-bound-1*
diff-less empty-tree-empty-V le-neq-implies-less less-imp-diff-less zero-less-one **by** *presburger*
lemma *width-V-empty*: $V = \{\} \implies \text{width} = 0$ **unfolding** *width-def max-bag-card-def*
using *bag-bound-V-empty T.finite-vertex-set* **by** (*cases* $V_T = \{\}$) *auto*
lemma *width-bound-V-le*: $\text{width} \leq \text{card } V - 1$
using *width-upper-bound-V width-V-empty* **by** (*cases* $V = \{\}$) *auto*
lemma *width-lower-bound-1*:
assumes $v \rightarrow w$
shows $\text{width} \geq 1$
proof–
obtain t **where** $t: t \in V_T v \in \text{bag } t w \in \text{bag } t$ **using** *bags-edges assms* **by** *blast*
have $\text{card } (\text{bag } t) \neq 0$ **using** $t(1,2)$ *bag-finite card-0-eq empty-iff* **by** *blast*
moreover **have** $\text{card } (\text{bag } t) \neq 1$ **using** $t(2,3)$ *assms no-loops*
by (*metis One-nat-def card-Suc-eq empty-iff insertE*)
ultimately **have** $\text{card } (\text{bag } t) \geq 2$ **by** *simp*
hence $\text{max-bag-card} > 1$ **using** $t(1)$ *max-bag-card-lower-bound-bag* **by** *fastforce*
thus *?thesis* **unfolding** *width-def* **using** $t(1)$ **by** *fastforce*
qed
end — locale *TreeDecomposition*

4.2 Treewidth of a Graph

context *Graph* **begin**

The treewidth of a graph is the minimum treewidth over all its tree decompositions. Here we assume without loss of generality that the universe of the vertices of the tree is *nat*. Because trees are finite, *nat* always contains enough elements.

abbreviation *treewidth-cards* $:: \text{nat set}$ **where** *treewidth-cards* \equiv

$\{ \text{TreeDecomposition.width } T \text{ bag} \mid (T :: \text{'c Graph}) \text{ bag. TreeDecomposition } G \ T \ \text{bag} \}$

definition *treewidth* $:: \text{nat}$ **where** *treewidth* $\equiv \text{Min } \text{treewidth-cards}$

Every graph has a trivial tree decomposition consisting of a single bag containing all of V .

proposition *tree-decomposition-exists*: $\exists (T :: \text{'c Graph}) \text{ bag. TreeDecomposition } G \ T \ \text{bag}$ **proof**–

obtain x **where** $x \in (\text{UNIV} :: \text{'c set})$ **by** *blast*

define T **where** [*simp*]: $T = (\downarrow \text{verts} = \{x\}, \text{arcs} = \{\})$

define bag **where** [*simp*]: $\text{bag} = (\lambda _ :: \text{'c. } V)$

have *Graph* T **by** *unfold-locales simp-all*

then interpret T : *Graph* T .
have $\bigwedge xs. \neg T.cycle\ xs$ **using** $T.cycleE$ **by** *auto*
moreover have $\bigwedge v\ w. v \in V_T \implies w \in V_T \implies T.connected\ v\ w$ **using** $T.connected-refl$ **by**
auto
ultimately have *Tree* T **by** *unfold-locales*
then interpret T : *Tree* T .
have $TreeDecomposition\ G\ T\ bag$ **by** *unfold-locales* (*simp-all* *add: edges-are-in-V*)
thus *?thesis* **by** *blast*
qed

corollary *treewidth-cards-upper-bound-V*: $n \in treewidth-cards \implies n \leq card\ V - 1$
using $TreeDecomposition.width-bound-V-le$ **by** *blast*
corollary *treewidth-cards-finite*: *finite* *treewidth-cards*
using *treewidth-cards-upper-bound-V* *finite-nat-set-iff-bounded-le* **by** *auto*
corollary *treewidth-cards-nonempty*: $treewidth-cards \neq \{\}$ **by** (*simp* *add: tree-decomposition-exists*)

lemma *treewidth-cards-treewidth*:
 $\exists (T :: nat\ Graph)\ bag. TreeDecomposition\ G\ T\ bag \wedge treewidth = TreeDecomposition.width\ T\ bag$
using *Min-in* *treewidth-cards-finite* *treewidth-cards-nonempty* *treewidth-def* **by** *fastforce*

corollary *treewidth-upper-bound-V*: $treewidth \leq card\ V - 1$ **unfolding** *treewidth-def*
using *treewidth-cards-nonempty* *Min-in* *treewidth-cards-finite* *treewidth-cards-upper-bound-V* **by**
auto

corollary *treewidth-upper-bound-0*: $V = \{\} \implies treewidth = 0$ **using** *treewidth-upper-bound-V* **by**
simp

corollary *treewidth-upper-bound-1*: $card\ V = 1 \implies treewidth = 0$ **using** *treewidth-upper-bound-V*
by *simp*

corollary *treewidth-lower-bound-1*: $v \rightarrow w \implies treewidth \geq 1$
using $TreeDecomposition.width-lower-bound-1$ *treewidth-cards-treewidth* **by** *fastforce*

lemma *treewidth-upper-bound-ex*:
 $\llbracket TreeDecomposition\ G\ (T :: nat\ Graph)\ bag; TreeDecomposition.width\ T\ bag \leq n \rrbracket \implies treewidth \leq n$
unfolding *treewidth-def*
by (*metis* (*mono-tags*, *lifting*) *Min-le* *dual-order.trans* *mem-Collect-eq* *treewidth-cards-finite*)

end — locale *Graph*

4.3 Separations

context *TreeDecomposition* **begin**

Every edge $s \rightarrow_T t$ in T separates T . In a tree decomposition, this edge also separates G . Proving this is our goal. First, let us define the set of vertices appearing in the left subtree when separating the tree at $s \rightarrow_T t$.

definition *left-part* :: $'c \Rightarrow 'c \Rightarrow 'a$ *set* **where**
 $left-part\ s\ t \equiv \bigcup \{ bag\ u \mid u. u \in T.left-tree\ s\ t \}$

lemma *left-partI* [*intro*]: $\llbracket v \in bag\ u; u \in T.left-tree\ s\ t \rrbracket \implies v \in left-part\ s\ t$
unfolding *left-part-def* **by** *blast*

lemma *left-part-in-V*: $left-part\ s\ t \subseteq V$ **unfolding** *left-part-def*

using $T.left-tree-in-V$ $bags-in-V$ by *blast*

Let us define the subgraph of T induced by a vertex of G .

definition $vertex-subtree :: 'a \Rightarrow 'c$ set **where**

$vertex-subtree\ v \equiv \{ t \in V_T. v \in bag\ t \}$

lemma $vertex-subtreeI$ [*intro*]: $\llbracket t \in V_T; v \in bag\ t \rrbracket \implies t \in vertex-subtree\ v$

unfolding $vertex-subtree-def$ by *blast*

The suggestive name $vertex-subtree$ is correct: Because T is a tree decomposition, $vertex-subtree\ v$ is a subtree (it is connected).

lemma $vertex-subtree-connected$:

assumes $v: v \in V$ **and** $s: s \in vertex-subtree\ v$ **and** $t: t \in vertex-subtree\ v$

and $xs: s \rightsquigarrow xs \rightsquigarrow_T t$

shows set $xs \subseteq vertex-subtree\ v$

using *assms* **proof** (*induct xs arbitrary: s*)

case ($Cons\ x\ xs$)

show *?case* **proof** (*cases*)

assume $xs = []$ **thus** *?thesis* **using** $Cons.prem\ s(3,4)$ by *auto*

next

assume $xs \neq []$

moreover **hence** $last\ xs = t$ **using** $Cons.prem\ s(4)$ *last.simps* by *auto*

moreover **have** $T.path\ xs$ **using** $Cons.prem\ s(4)$ $T.walk-tl$ by *fastforce*

moreover **have** $hd\ xs \in vertex-subtree\ v$ **proof**

have $hd\ xs \in set\ (s \rightsquigarrow_T t)$ **using** $T.unique-connecting-path-unique$

using $Cons.prem\ s(4)$ $\langle xs \neq [] \rangle$ by *auto*

hence $bag\ s \cap bag\ t \subseteq bag\ (hd\ xs)$

using $bags-continuous$ $Cons.prem\ s(4)$ $T.connected-in-V$ by *blast*

thus $v \in bag\ (hd\ xs)$ **using** $Cons.prem\ s(2,3)$ **unfolding** $vertex-subtree-def$ by *blast*

show $hd\ xs \in V_T$ **using** $T.connected-in-V(1)$ $\langle xs \neq [] \rangle$ $\langle T.path\ xs \rangle$ by *blast*

qed

ultimately **have** set $xs \subseteq vertex-subtree\ v$ **using** $Cons.hyps$ $Cons.prem\ s(1,3)$ by *blast*

thus *?thesis* **using** $Cons.prem\ s(2,4)$ by *auto*

qed

qed *simp*

corollary $vertex-subtree-unique-path-connected$:

assumes $v \in V$ $s \in vertex-subtree\ v$ $t \in vertex-subtree\ v$

shows set $(s \rightsquigarrow_T t) \subseteq vertex-subtree\ v$

using *assms* $vertex-subtree-connected$ $T.unique-connecting-path-properties$

by (*metis* (*no-types*, *lifting*) $T.unique-connecting-path$ $T.unique-connecting-path-unique$ *mem-Collect-eq* $vertex-subtree-def$)

In order to prove that edges in T are separations in G , we need one key lemma. If a vertex appears on both sides of a separation, then it also appears in the separation.

lemma $vertex-in-separator$:

assumes $st: s \rightarrow_T t$ **and** $v: v \in left-part\ s\ t$ $v \in left-part\ t\ s$

shows $v \in bag\ s$ $v \in bag\ t$

proof–

obtain $u\ u'$ **where** $u: v \in bag\ u$ $u \in T.left-tree\ s\ t$ $v \in bag\ u'$ $u' \in T.left-tree\ t\ s$

using v **unfolding** $left-part-def$ by *blast*

have $s \in set\ (u \rightsquigarrow_T u')$ **using** $T.left-tree-separates\ st\ u$ by *blast*

thus $v \in \text{bag } s$ using *bags-continuous* u by (*meson* *IntI* *T.left-treeE subsetCE*)
have $t \in \text{set } (u \rightsquigarrow_T u')$ using *T.left-tree-separates'* st u by *blast*
thus $v \in \text{bag } t$ using *bags-continuous* u by (*meson* *IntI* *T.left-treeE subsetCE*)

qed

Now we can show the main theorem: For every edge $s \rightarrow_T t$ in T , the set $\text{bag } s \cap \text{bag } t$ is a separator of G . That is, every path from the left part to the right part goes through $\text{bag } s \cap \text{bag } t$.

theorem *bags-separate*:

assumes $st: s \rightarrow_T t$ and $v: v \in \text{left-part } s$ t and $w: w \in \text{left-part } t$ s and $xs: v \rightsquigarrow xs \rightsquigarrow w$
shows $\text{set } xs \cap \text{bag } s \cap \text{bag } t \neq \{\}$

proof (*rule ccontr*)

assume $\neg ?thesis$

{

fix u assume $u \in \text{set } xs$

with xs $v \leftarrow ?thesis$ have $\text{vertex-subtree } u \subseteq T.\text{left-tree } s$ t

proof (*induct* xs *arbitrary: v*)

case (*Cons* x xs v)

hence *contra*: $v \notin \text{bag } s \vee v \notin \text{bag } t$ by (*metis* *path-from-toE* *IntI* *empty-iff* *hd-in-set*)

{

assume $x = u$ $\neg \text{vertex-subtree } u \subseteq T.\text{left-tree } s$ t

then obtain z where $z: z \in \text{vertex-subtree } u$ $z \notin T.\text{left-tree } s$ t by *blast*

hence $z \in \text{vertex-subtree } v$ using *Cons.prem* $s(1,3)$ $\langle x = u \rangle$

by (*metis* *list.sel(1)* *path-from-to-def*)

hence $v \in \text{left-part } t$ s **unfolding** *vertex-subtree-def*

using *T.left-tree-union-V* z st by *auto*

hence *False* using *vertex-in-separator* *contra* st *Cons.prem* $s(2)$ by *blast*

}

moreover {

assume $x \neq u$

hence $u \in \text{set } xs$ using *Cons.prem* $s(4)$ by *auto*

moreover hence $xs \neq \text{Nil}$ using *empty-iff* *list.set(1)* by *auto*

moreover hence $\text{last } xs = w$ using *Cons.prem* $s(1)$ by *auto*

moreover have *path* xs using *Cons.prem* $s(1)$ *walk-tl* by *force*

moreover have $\text{hd } xs \in \text{left-part } s$ t **proof**–

have $v \rightarrow \text{hd } xs$ using *Cons.prem* $s(1,3)$ $\langle xs \neq \text{Nil} \rangle$ *walk-first-edge'* by *auto*

then obtain u' where $u': u' \in V_T$ $v \in \text{bag } u'$ $\text{hd } xs \in \text{bag } u'$

using *bags-edges* by *blast*

hence $u' \in T.\text{left-tree } s$ t

using *contra* *vertex-in-separator* st *T.left-tree-union-V* *Cons.prem* $s(2)$ by *blast*

thus *?thesis* using $u'(3)$ **unfolding** *left-part-def* by *blast*

qed

moreover have $\neg \text{set } xs \cap \text{bag } s \cap \text{bag } t \neq \{\}$ using *Cons.prem* $s(3)$

IntI *disjoint-iff-not-equal* *inf-le1* *inf-le2* *set-subset-Cons* *subsetCE* by *auto*

ultimately have $\text{vertex-subtree } u \subseteq T.\text{left-tree } s$ t using *Cons.hyps* by *blast*

}

ultimately show *?case* by *blast*

qed *simp*

}

hence $\text{vertex-subtree } w \subseteq T.\text{left-tree } s$ t using xs *last-in-set* by *blast*

moreover have $\text{vertex-subtree } w \cap T.\text{left-tree } t$ $s \neq \{\}$ using w

unfolding *left-part-def T.left-tree-def* **by** *blast*
ultimately show *False* **using** *T.left-tree-disjoint st* **by** *blast*
qed

It follows that vertices cannot be dropped from a bag if they have a neighbor that has not been visited yet (that is, a neighbor that is strictly in the right part of the separation).

corollary *bag-no-drop*:

assumes *st: s →_T t* **and** *vw: v→w* **and** *v: v ∈ bag s* **and** *w: w ∉ bag s w ∈ left-part t s*
shows *v ∈ bag t*

proof–

have *v ∼_[v,w] w* **using** *v vw w(1)* **by** *auto*
hence *set [v,w] ∩ bag s ∩ bag t ≠ {}* **using** *st v w(2)*
by (*meson T.edges-are-in-V T.left-tree-initial bags-separate left-partI*)
thus *?thesis* **using** *w(1)* **by** *auto*

qed

end — locale *TreeDecomposition*
end

5 Treewidth of Trees

theory *TreewidthTree*

imports *TreeDecomposition* **begin**

The treewidth of a tree is 1 if the tree has at least one edge, otherwise it is 0.

For simplicity and without loss of generality, we assume that the vertex set of the tree is a subset of the natural numbers because this is what we use in the definition of *Graph.treewidth*.

While it would be nice to lift this restriction, removing it would entail defining isomorphisms between graphs in order to map the tree decomposition to a tree decomposition over the natural numbers. This is outside the scope of this theory and probably not terribly interesting by itself.

theorem *treewidth-tree*:

fixes *G :: nat Graph (structure)*
assumes *Tree G*
shows *Graph.treewidth G ≤ 1*

proof–

interpret *Tree G* **using** *assms* .

{

assume *V ≠ {}*

then obtain *root* **where** *root: root ∈ V* **by** *blast*

then interpret *RootedTree G root* **by** *unfold-locales*

define *bag* **where** *bag v = (if v = root then {v} else {v, parent v})* **for** *v*

have *v-in-bag: ∧v. v ∈ bag v* **unfolding** *bag-def* **by** *simp*

have *bag-in-V: ∧v. v ∈ V ⇒ bag v ⊆ V* **unfolding** *bag-def*

using *parent-in-V empty-subsetI insert-subset* **by** *auto*

have *TreeDecomposition G G bag* **proof**

show $\bigcup \{ \text{bag } t \mid t. t \in V \} = V$ **using** *bag-in-V v-in-bag* **by** *blast*

next

fix *v w* **assume** *v→w*

```

moreover have  $\bigwedge v' w'. \llbracket v' \rightarrow w'; v' \neq \text{root} \rrbracket \implies w' \in \text{bag } v' \vee v' \in \text{bag } w'$  unfolding bag-def
  by (metis insertI2 parent-edge-cases parent-edge-root singletonI)
ultimately have  $v \in \text{bag } w \vee w \in \text{bag } v$  using no-loops undirected by blast
thus  $\exists t \in V. v \in \text{bag } t \wedge w \in \text{bag } t$  using  $\langle v \rightarrow w \rangle$  edges-are-in-V v-in-bag by blast
next
fix  $s u t$  assume  $s: s \in V$  and  $u: u \in V$  and  $t: t \in \text{set } (s \rightsquigarrow u)$ 
have  $t \in V$  using  $t$  by (meson s subsetCE u unique-connecting-path-properties(1) walk-in-V)
hence  $s = u \implies t = s$  using left-tree-initial'  $s t$  by blast
moreover have  $s \rightarrow u \implies t = s \vee t = u$  using  $s t u \langle t \in V \rangle$ 
  by (metis insertE left-treeI left-tree-initial' list.exhaust-sel list.simps(15))
  undirected unique-connecting-path-properties(2,3) unique-connecting-path-set(2)
  unique-connecting-path-tl)
moreover {
  assume  $*$ :  $s \neq u \neg s \rightarrow u$ 
  have  $s = \text{root} \implies \text{bag } s \cap \text{bag } u = \{\}$  unfolding bag-def
    using  $*(1,2)$  parent-edge u undirected by fastforce
  moreover have  $u = \text{root} \implies \text{bag } s \cap \text{bag } u = \{\}$  unfolding bag-def
    using  $*(1,2)$  parent-edge s by fastforce
  moreover have  $\llbracket s \neq \text{root}; u \neq \text{root}; \text{parent } s \neq \text{parent } u \rrbracket \implies \text{bag } s \cap \text{bag } u = \{\}$ 
    unfolding bag-def using  $*(2)$  parent-edge s u undirected by fastforce
  moreover {
    assume  $**$ :  $s \neq \text{root } u \neq \text{root } \text{parent } s = \text{parent } u t \neq s t \neq u$ 
    have  $\text{bag } s \cap \text{bag } u = \{\text{parent } s\}$  unfolding bag-def using  $*(1) ** (1-3)$ 
      Int-insert-left inf.orderE insertE insert-absorb subset-insertI by auto
    moreover have  $t = \text{parent } s$ 
      using sibling-path[OF s **(1) u **(2) *(1) **(3)] t **(4,5) by auto
    ultimately have  $\text{bag } s \cap \text{bag } u \subseteq \text{bag } t$  by (simp add: v-in-bag)
  }
  ultimately have  $\text{bag } s \cap \text{bag } u \subseteq \text{bag } t$  by blast
}
ultimately show  $\text{bag } s \cap \text{bag } u \subseteq \text{bag } t$  by blast
qed
then interpret TreeDecomposition  $G G \text{ bag}$  .
{
fix  $v$ 
have  $\text{card } \{v, \text{parent } v\} \leq 2$ 
  by (metis card.insert card.empty finite.emptyI finite-insert insert-absorb insert-not-empty)
  lessI less-or-eq-imp-le numerals(2))
hence  $\text{card } (\text{bag } v) \leq 2$  unfolding bag-def by simp
}
hence  $\text{max-bag-card} \leq 2$  using  $\langle V \neq \{\} \rangle$  max-bag-card-in-bag-cards by auto
hence  $\text{width} \leq 1$  unfolding width-def by (simp add: \langle V \neq \{\} \rangle)
hence  $\exists \text{ bag}. \text{TreeDecomposition } G G \text{ bag} \wedge \text{TreeDecomposition.width } G \text{ bag} \leq 1$ 
  using TreeDecomposition-axioms by blast
}
thus ?thesis by (metis TreeDecomposition.width-V-empty le-0-eq linear)
treewidth-cards-treewidth treewidth-upper-bound-ex)
qed

```

If the tree is non-trivial, that is, if it contains more than one vertex, then its treewidth is exactly 1.

corollary *treewidth-tree-exact*:
fixes $G :: \text{nat Graph}$ (**structure**)
assumes $\text{Tree } G \text{ card } V_G > 1$
shows $\text{Graph.treewidth } G = 1$
using *assms Graph.treewidth-lower-bound-1 Tree.tree-has-edge Tree-def treewidth-tree*
by *fastforce*

end

6 Treewidth of Complete Graphs

theory *TreewidthCompleteGraph*
imports *TreeDecomposition* **begin**

As an application of the separator theorem *bags-separate*, or more precisely its corollary *bag-no-drop*, we show that a complete graph of size n (a clique) has treewidth $n - 1$.

theorem (**in** *Graph*) *treewidth-complete-graph*:
assumes $\bigwedge v w. \llbracket v \in V; w \in V; v \neq w \rrbracket \implies v \rightarrow w$
shows $\text{treewidth} = \text{card } V - 1$

proof –

{
assume $V \neq \{\}$
obtain T *bag* **where**
 $T: \text{TreeDecomposition } G$ ($T :: \text{nat Graph}$) *bag* $\text{treewidth} = \text{TreeDecomposition.width } T$ *bag*
using *treewidth-cards-treewidth* **by** *blast*
interpret $\text{TreeDecomposition } G$ T *bag* **using** $T(1)$.

assume $\neg ?thesis$
hence $\text{width} \neq \text{card } V - 1$ **by** (*simp add: T(2)*)

Let s be a bag of maximal size.

moreover obtain s **where** $s: s \in V_T$ $\text{card } (\text{bag } s) = \text{max-bag-card}$
using *max-bag-card-in-bag-cards* $\langle V \neq \{\} \rangle$ **by** *fastforce*

The treewidth cannot be larger than $\text{card } V - 1$, so due to our assumption $\text{width} \neq \text{card } V - 1$ it must be smaller, hence $\text{card } (\text{bag } s) < \text{card } V$.

ultimately have $\text{card } (\text{bag } s) < \text{card } V$ **unfolding** *width-def*
using $\langle V \neq \{\} \rangle$ *empty-tree-empty-V le-eq-less-or-eq max-bag-card-upper-bound-V* **by** *presburger*
then obtain v **where** $v: v \in V$ $v \notin \text{bag } s$ **by** (*meson bag-finite card-mono not-less s(1) subsetI*)

There exists a bag containing v . We consider the path from s to t and find that somewhere along this path there exists a bag containing $\text{insert } v$ ($\text{bag } s$), which is a contradiction because such a bag would be too big.

obtain t **where** $t: t \in V_T$ $v \in \text{bag } t$ **using** *bags-exist v(1)* **by** *blast*
with s **have** $\exists t \in V_T. \text{insert } v$ ($\text{bag } s$) $\subseteq \text{bag } t$ **proof** (*induct s \rightsquigarrow_T t arbitrary: s*)
case *Nil* **thus** *?case* **using** *T.unique-connecting-path-properties(2)* **by** *fastforce*
next
case (*Cons x xs s*)
show *?case* **proof** (*cases*)
assume $v \in \text{bag } s$ **thus** *?thesis* **using** *t Cons.prem(1)* **by** *blast*

```

next
  assume  $v \notin \text{bag } s$ 
  hence  $s \neq t$  using  $t(2)$  by blast
  hence  $xs \neq \text{Nil}$  using  $\text{Cons.hyps}(2)$   $\text{Cons.prem}(1,3)$ 
    by (metis  $T.\text{unique-connecting-path-properties}(3,4)$   $\text{last-ConsL}$   $\text{list.sel}(1)$ )
  moreover have  $x = s$  using  $\text{Cons.hyps}(2)$   $\text{Cons.prem}(1)$   $t(1)$ 
    by (metis  $T.\text{unique-connecting-path-properties}(3)$   $\text{list.sel}(1)$ )
  ultimately obtain  $s' xs'$  where  $s': s \# s' \# xs' = s \rightsquigarrow_T t$ 
    using  $\text{Cons.hyps}(2)$   $\text{list.exhaust}$  by metis
  moreover have  $st\text{-path}: T.\text{path } (s \rightsquigarrow_T t)$ 
    by (simp add:  $\text{Cons.prem}(1)$   $T.\text{unique-connecting-path-properties}(1)$   $t(1)$ )
  ultimately have  $s' \in V_T$  by (metis  $T.\text{edges-are-in-}V(2)$   $T.\text{path-first-edge}$ )

```

Bags can never drop vertices because every vertex has a neighbor in G which has not yet been visited.

```

have  $s\text{-in-}s': \text{bag } s \subseteq \text{bag } s'$  proof
  fix  $w$  assume  $w \in \text{bag } s$ 
  moreover have  $s \rightarrow_T s'$  using  $s'$   $st\text{-path}$  by (metis  $T.\text{walk-first-edge}$ )
  moreover have  $v \in \text{left-part } s' s$  using  $\text{Cons.prem}(1,4)$   $s' t(1)$ 
    by (metis  $T.\text{left-treeI}$   $T.\text{unique-connecting-path-rev}$   $\text{insert-subset}$   $\text{left-partI}$ 
       $\text{list.simps}(15)$   $\text{set-rev}$   $\text{subsetI}$ )
  ultimately show  $w \in \text{bag } s'$ 
    using  $\text{bag-no-drop}$   $\text{Cons.prem}(1,4)$   $\langle v \notin \text{bag } s \rangle$   $\text{assms}$   $\text{bags-in-}V$   $v(1)$  by blast
qed

```

Bags can never gain vertices because we started with a bag of maximal size.

```

moreover have  $\text{card } (\text{bag } s') \leq \text{card } (\text{bag } s)$  proof-
  have  $\text{card } (\text{bag } s') \leq \text{max-bag-card}$  unfolding  $\text{max-bag-card-def}$ 
    using  $\text{Max-ge}$   $\langle s' \in V_T \rangle$   $\text{bag-cards-finite}$  by blast
  thus ?thesis using  $\text{Cons.prem}(2)$  by auto
qed
ultimately have  $\text{bag } s' = \text{bag } s$  using  $\langle s' \in V_T \rangle$   $\text{bag-finite}$   $\text{card-seteq}$  by blast
thus ?thesis
  using  $\text{Cons.hyps}$   $\text{Cons.prem}(1,2)$   $\langle s' \in V_T \rangle$   $t$   $s'$   $st\text{-path}$   $\langle xs \neq [] \rangle$ 
  by (metis  $T.\text{path-from-toI}$   $T.\text{path-tl}$   $T.\text{unique-connecting-path-properties}(4)$ 
     $T.\text{unique-connecting-path-unique}$   $\text{last.simps}$   $\text{list.sel}(1,3)$ )
qed
qed
hence  $\exists t \in V_T. \text{card } (\text{bag } s) < \text{card } (\text{bag } t)$  using  $v(2)$ 
  by (metis  $\text{bag-finite}$   $\text{card-seteq}$   $\text{insert-subset}$   $\text{not-le}$ )
hence False using  $s$   $\text{Max.coboundedI}$   $\text{bag-cards-finite}$   $\text{not-le}$  unfolding  $\text{max-bag-card-def}$  by
auto
}
thus ?thesis using  $\text{treewidth-upper-bound-}V$   $\text{card.empty}$   $\text{diff-diff-cancel}$   $\text{zero-diff}$  by fastforce
qed
end

```

7 Example Instantiations

This section provides a few example instantiations for the locales to show that they are not empty.

```
theory ExampleInstantiations
imports TreewidthCompleteGraph begin
```

```
datatype Vertices = u0 | v0 | w0
```

The empty graph is a tree.

```
definition T1 ≡ (| verts = {}, arcs = {} |)
```

```
interpretation Graph-T1: Graph T1 unfolding T1-def by standard simp-all
```

```
interpretation Tree-T1: Tree T1
```

```
  by (rule Tree.intro, simp add: Graph-T1.Graph-axioms, standard, unfold T1-def, simp)
    (metis T1-def Graph-T1.cycle-def equals0D_simps(2))
```

The complete graph with 2 vertices.

```
definition T2 ≡ (| verts = {u0, v0}, arcs = {(u0,v0),(v0,u0)} |)
```

```
lemma Graph-T2: Graph T2 unfolding T2-def by standard auto
```

```
lemma Tree-T2: Tree T2
```

```
proof -
```

```
  interpret Graph T2 using Graph-T2 .
```

```
  show ?thesis proof
```

```
    fix v w assume v ∈ VT2 w ∈ VT2 thus connected v w
```

```
    by (metis T2-def connected-def connected-edge empty-iff insert-iff last.simps list.discI
      list.sel(1) path-singleton_simps(1,2))
```

```
  next
```

```
    fix xs :: Vertices list
```

```
    {
```

```
      fix x y
```

```
      assume cycle xs and xy: (x = v0 ∧ y = u0) ∨ (x = u0 ∧ y = v0) and hd xs = x
```

```
      hence last xs = y
```

```
      by (metis T2-def cycleE distinct_simps(2) distinct-singleton insert-iff list.set(1)
        prod.inject_simps(2))
```

```
      moreover have  $\bigwedge v. v \in \text{set } xs \implies v = x \vee v = y$  using ⟨cycle xs⟩ xy
```

```
      by (metis cycle-def walk-in-V T2-def empty-iff insertE insert-absorb insert-subset
        select-convs(1))
```

```
      ultimately have xs = [x,y] using ⟨cycle xs⟩ xy
```

```
      by (metis cycleE distinct-length-2-or-more last.simps list.exhaust-sel list.set-sel(1)
        list.set-sel(2) no-loops)
```

```
      hence False using ⟨cycle xs⟩ unfolding cycle-def by simp
```

```
    }
```

```
    thus ¬cycle xs by (metis T2-def cycleE empty-iff insertE prod.inject_simps(2))
```

```
  qed
```

```
qed
```

As expected, the treewidth of the complete graph with 2 vertices is 1.

Note that we use *Graph.treewidth-complete-graph* here and not *treewidth-tree*. This is because *treewidth-tree* requires the vertex set of the graph to be a set of natural numbers, which is not the case here.

lemma *T2-complete*: $\llbracket v \in V_{T2}; w \in V_{T2}; v \neq w \rrbracket \implies v \rightarrow_{T2} w$ **unfolding** *T2-def* **by** *auto*
lemma *treewidth-T2*: *Graph.treewidth T2 = 1*
using *Graph.treewidth-complete-graph[OF Graph-T2]* *T2-complete* **unfolding** *T2-def* **by** *simp*

The complete graph with 3 vertices.

definition *T3* \equiv ($\text{verts} = \{u0, v0, w0\}$, $\text{arcs} = \{(u0, v0), (v0, u0), (v0, w0), (w0, v0), (w0, u0), (u0, w0)\}$)
 \Downarrow

lemma *Graph-T3*: *Graph T3* **unfolding** *T3-def* **by** *standard auto*

$[u0, v0, w0]$ is a cycle in *T3*, so *T3* is not a tree.

lemma *Not-Tree-T3*: $\neg \text{Tree } T3$ **proof**

assume *Tree T3* **then interpret** *Tree T3* .

let $?xs = [u0, v0, w0]$

have *path ?xs* **by** (*metis T3-def Vertices.distinct(1,3,5)*)

distinct-length-2-or-more distinct-singleton insert-iff_simps(2) walk.Cons walk-2)

moreover have (*hd ?xs, last ?xs*) $\in \text{arcs } T3$ **by** (*simp add: T3-def*)

ultimately show *False* **using** *meeting-paths-produce-cycle no-cycles walk-2*

by (*metis distinct-length-2-or-more last-ConsL last-ConsR list.sel(1)*)

qed

lemma *T3-complete*: $\llbracket v \in V_{T3}; w \in V_{T3}; v \neq w \rrbracket \implies v \rightarrow_{T3} w$ **unfolding** *T3-def* **by** *auto*

lemma *treewidth-T3*: *Graph.treewidth T3 = 2*

using *Graph.treewidth-complete-graph[OF Graph-T3]* *T3-complete* **unfolding** *T3-def* **by** *simp*

We omit a concrete example for the *TreeDecomposition* locale because *tree-decomposition-exists* already shows that it is non-empty.

end

References

- [1] Reinhard Diestel. *Graph Theory, 3rd Edition*, volume 173 of *Graduate texts in mathematics*. Springer, 2006.
- [2] Fedor V. Fomin and Dimitrios M. Thilikos. An annotated bibliography on guaranteed graph searching. *Theor. Comput. Sci.*, 399(3):236–245, 2008.
- [3] Lars Noschinski. Graph theory. *Archive of Formal Proofs*, April 2013. http://isa-afp.org/entries/Graph_Theory.shtml, Formal proof development.
- [4] Neil Robertson and Paul D. Seymour. Graph minors. v. excluding a planar graph. *J. Comb. Theory, Ser. B*, 41(1):92–114, 1986.