Tree Decompositions

Christoph Dittmann
christoph.dittmann@tu-berlin.de

March 17, 2025

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

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 $(\mathcal{T}, \beta)$ be a tree decomposition, where $\mathcal{T}$ is a tree and $\beta : V(\mathcal{T}) \to 2^V$ 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

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

and

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

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 $f : \mathbb{N} \to \mathbb{N}$ 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* (‹*V*ı›)
  arcs :: *'a Edge set* (‹*E*ı›)
abbreviation *is-arc* :: (*'a*, *'b*) *Graph-scheme* ⇒ *'a* ⇒ *'a* ⇒ *bool* (**infixl** ‹→ı› *60*) **where**
  $v \to_G w \equiv (v,w) \in E_G$

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*: $v{\to}w = w{\to}v$
    and *no-loops*: $\neg v{\to}v$
begin
lemma *finite-edge-set* [*simp*]: *finite E* ⟨*proof*⟩
lemma *edges-are-in-V*: **assumes** $v{\to}w$ **shows** $v \in V$ $w \in V$
  ⟨*proof*⟩

2.1 Walks

A walk is sequence of vertices connected by edges.

inductive *walk* :: *'a Walk* ⇒ *bool* **where**
Nil [*simp*]: *walk* []
| *Singleton* [*simp*]: $v \in V \Longrightarrow$ *walk* [*v*]
| *Cons*: $v{\to}w \Longrightarrow$ *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*]: $v \rightarrow w \Longrightarrow$ *walk* $[v,w]$ ⟨*proof*⟩
lemma *walk-comp*: ⟦ *walk xs*; *walk ys*; *xs* = *Nil* ∨ *ys* = *Nil* ∨ *last xs*→*hd ys* ⟧ ⟹ *walk* (*xs* @ *ys*)
⟨*proof*⟩
lemma *walk-tl*: *walk xs* ⟹ *walk* (*tl xs*) ⟨*proof*⟩
lemma *walk-drop*: *walk xs* ⟹ *walk* (*drop n xs*) ⟨*proof*⟩
lemma *walk-take*: *walk xs* ⟹ *walk* (*take n xs*)
⟨*proof*⟩
lemma *walk-rev*: *walk xs* ⟹ *walk* (*rev xs*)
⟨*proof*⟩
lemma *walk-decomp*: **assumes** *walk* (*xs* @ *ys*) **shows** *walk xs walk ys*
⟨*proof*⟩
lemma *walk-dropWhile*: *walk xs* ⟹ *walk* (*dropWhile f xs*) ⟨*proof*⟩
lemma *walk-takeWhile*: *walk xs* ⟹ *walk* (*takeWhile f xs*) ⟨*proof*⟩

lemma *walk-in-V*: *walk xs* ⟹ *set xs* ⊆ *V* ⟨*proof*⟩

lemma *walk-first-edge*: *walk* (*v* # *w* # *xs*) ⟹ *v*→*w* ⟨*proof*⟩
lemma *walk-first-edge'*: ⟦ *walk* (*v* # *xs*); *xs* ≠ *Nil* ⟧ ⟹ *v*→*hd xs*
⟨*proof*⟩
lemma *walk-middle-edge*: *walk* (*xs* @ *v* # *w* # *ys*) ⟹ *v*→*w*
⟨*proof*⟩
lemma *walk-last-edge*: ⟦ *walk* (*xs* @ *ys*); *xs* ≠ *Nil*; *ys* ≠ *Nil* ⟧ ⟹ *last xs*→*hd ys*
⟨*proof*⟩

lemma *walk-takeWhile-edge*:
assumes *walk* (*xs* @ [*v*]) *xs* ≠ *Nil hd xs* ≠ *v*
shows *last* (*takeWhile* (λ*x*. *x* ≠ *v*) *xs*)→*v* (**is** *last ?xs*→*v*)
⟨*proof*⟩

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*]: ⟦ *walk xs*; *xs* ≠ *Nil*; *hd xs* = *v*; *last xs* = *w* ⟧ ⟹ *v* →* *w*
⟨*proof*⟩
lemma *connectedE*:
assumes *v* →* *w*
obtains *xs* **where** *walk xs xs* ≠ *Nil hd xs* = *v last xs* = *w*
⟨*proof*⟩

lemma *connected-in-V*: **assumes** *v* →* *w* **shows** *v* ∈ *V w* ∈ *V*
⟨*proof*⟩
lemma *connected-refl*: *v* ∈ *V* ⟹ *v* →* *v* ⟨*proof*⟩
lemma *connected-edge*: *v*→*w* ⟹ *v* →* *w* ⟨*proof*⟩
lemma *connected-trans*:
assumes *u-v*: *u* →* *v* **and** *v-w*: *v* →* *w*
shows *u* →* *w*
⟨*proof*⟩

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* $\Rightarrow$ *bool* **where** *path xs* $\equiv$ *walk xs* $\wedge$ *distinct xs*

lemma *path-singleton* [*simp*]: $v \in V \Longrightarrow$ *path* [v] ⟨*proof*⟩
lemma *path-2* [*simp*]: ⟦ $v \to w$; $v \neq w$ ⟧ $\Longrightarrow$ *path* [v,w] ⟨*proof*⟩
lemma *path-cons*: ⟦ *path xs*; *xs* $\neq$ *Nil*; $v \to$ *hd xs*; $v \notin$ *set xs* ⟧ $\Longrightarrow$ *path* (v # *xs*)
⟨*proof*⟩
lemma *path-comp*: ⟦ *walk xs*; *walk ys*; *xs* = *Nil* $\vee$ *ys* = *Nil* $\vee$ *last xs* $\to$ *hd ys*; *distinct* (*xs* @ *ys*) ⟧
$\Longrightarrow$ *path* (*xs* @ *ys*) ⟨*proof*⟩
lemma *path-tl*: *path xs* $\Longrightarrow$ *path* (*tl xs*) ⟨*proof*⟩
lemma *path-drop*: *path xs* $\Longrightarrow$ *path* (*drop n xs*) ⟨*proof*⟩
lemma *path-take*: *path xs* $\Longrightarrow$ *path* (*take n xs*) ⟨*proof*⟩
lemma *path-rev*: *path xs* $\Longrightarrow$ *path* (*rev xs*) ⟨*proof*⟩
lemma *path-decomp*: **assumes** *path* (*xs* @ *ys*) **shows** *path xs path ys*
⟨*proof*⟩
lemma *path-dropWhile*: *path xs* $\Longrightarrow$ *path* (*dropWhile f xs*) ⟨*proof*⟩
lemma *path-takeWhile*: *path xs* $\Longrightarrow$ *path* (*takeWhile f xs*) ⟨*proof*⟩
lemma *path-in-V*: *path xs* $\Longrightarrow$ *set xs* $\subseteq V$ ⟨*proof*⟩
lemma *path-first-edge*: *path* (v # w # *xs*) $\Longrightarrow v \to w$ ⟨*proof*⟩
lemma *path-first-edge'*: ⟦ *path* (v # *xs*); *xs* $\neq$ *Nil* ⟧ $\Longrightarrow v \to$ *hd xs* ⟨*proof*⟩
lemma *path-middle-edge*: *path* (*xs* @ v # w # *ys*) $\Longrightarrow v \to w$ ⟨*proof*⟩
lemma *path-takeWhile-edge*: ⟦ *path* (*xs* @ [v]); *xs* $\neq$ *Nil*; *hd xs* $\neq v$ ⟧
$\Longrightarrow$ *last* (*takeWhile* ($\lambda x.\ x \neq v$) *xs*) $\to v$ ⟨*proof*⟩

end

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 $\equiv$ *Graph.path G xs* $\wedge$ *xs* $\neq$ *Nil* $\wedge$ *hd xs* = v $\wedge$ *last xs* = w
context *Graph* **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
⟨*proof*⟩

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*⟩

corollary *connected-by-path*:
assumes $v \to^* w$
obtains *xs* **where** v ⇝*xs*⇝ w
⟨*proof*⟩

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* ⇒ *bool* **where**
  cycle xs ≡ *path xs* ∧ *length xs* > *2* ∧ *last xs* → *hd xs*

lemma *cycleI* [*intro*]: ⟦ *path xs*; *length xs* > *2*; *last xs*→*hd xs* ⟧ ⟹ *cycle xs*
  ⟨*proof*⟩
lemma *cycleE*: *cycle xs* ⟹ *path xs* ∧ *xs* ≠ *Nil* ∧ *length xs* > *2* ∧ *last xs*→*hd xs*
  ⟨*proof*⟩

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* ≠ *Nil*
    and *ys*: *path* (*v* # *ys*) *ys* ≠ *Nil*
    and *meet*: *last xs* = *last ys*
    and *diverge*: *hd xs* ≠ *hd ys*
  shows ∃ *zs*. *cycle zs* ∧ *hd zs* = *v*
⟨*proof*⟩

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

lemma *unique-paths-implies-no-cycles*:
  assumes *unique-paths*: ⋀*v w*. *v* →* *w* ⟹ ∃!*xs*. *v* ⇝*xs*⇝ *w*
  shows ⋀*xs*. ¬*cycle xs*
⟨*proof*⟩

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*: ⋀*xs*. ¬*cycle xs* **and** *connected*: *v* →* *w*
  shows ∃!*xs*. *v* ⇝*xs*⇝ *w*
⟨*proof*⟩

end — locale Graph
end

3 Trees

theory *Tree*
imports *Graph* **begin**

A tree is a connected graph without cycles.

locale *Tree* = *Graph* +

assumes *connected*: $[\![v \in V; w \in V]\!] \Longrightarrow v \rightarrow^{*} w$ **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*: $[\![v \in V; w \in V]\!] \Longrightarrow \exists!xs.\ v \leadsto xs \leadsto w$
⟨*proof*⟩

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** ‹⇝ı› *71*) **where** *unique-connecting-path G v w* $\equiv$ *THE xs.* $v \leadsto xs \leadsto_G w$

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

context *Tree* **begin**

lemma *unique-connecting-path-set*:
assumes $v \in V$ $w \in V$
shows $v \in set\ (v \leadsto w)$ $w \in set\ (v \leadsto w)$
⟨*proof*⟩

lemma *unique-connecting-path-properties*:
assumes $v \in V$ $w \in V$
shows $path\ (v \leadsto w)$ $v \leadsto w \neq Nil$ $hd\ (v \leadsto w) = v$ $last\ (v \leadsto w) = w$
⟨*proof*⟩

lemma *unique-connecting-path-unique*:
assumes $v \leadsto xs \leadsto w$
shows $xs = v \leadsto w$
⟨*proof*⟩
corollary *unique-connecting-path-connects*: $[\![v \in V; w \in V]\!] \Longrightarrow v \leadsto (v \leadsto w) \leadsto w$
⟨*proof*⟩

lemma *unique-connecting-path-rev*:
assumes $v \in V$ $w \in V$
shows $v \leadsto w = rev\ (w \leadsto v)$
⟨*proof*⟩

lemma *unique-connecting-path-decomp*:
assumes $v \in V$ $w \in V$ $v \leadsto w = ps\ @\ u\ \#\ ps'$
shows $ps\ @\ [u] = v \leadsto u$ $u\ \#\ ps' = u \leadsto w$
⟨*proof*⟩

lemma *unique-connecting-path-tl*:
assumes $v \in V$ $u \in set\ (w \leadsto v)$ $u \rightarrow w$
shows $tl\ (w \leadsto v) = u \leadsto v$
⟨*proof*⟩

Every tree with at least two vertices contains an edge.

lemma *tree-has-edge*:
  assumes *card* $V > 1$
  shows $\exists\, v\ w.\ v \rightarrow w$
$\langle proof \rangle$

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*]: $[\![\ u \in V;\ s \in set\ (u \rightsquigarrow t)\ ]\!] \Longrightarrow u \in$ *left-tree* $s\ t$
  $\langle proof \rangle$
lemma *left-treeE*: $u \in$ *left-tree* $s\ t \Longrightarrow u \in V \wedge s \in set\ (u \rightsquigarrow t)$
  $\langle proof \rangle$

lemma *left-tree-in-V*: *left-tree* $s\ t \subseteq V$ $\langle proof \rangle$
lemma *left-tree-initial*: $[\![\ s \in V;\ t \in V\ ]\!] \Longrightarrow s \in$ *left-tree* $s\ t$
  $\langle proof \rangle$
lemma *left-tree-initial'*: $[\![\ s \in V;\ t \in V;\ s \neq t\ ]\!] \Longrightarrow t \notin$ *left-tree* $s\ t$
  $\langle proof \rangle$
lemma *left-tree-initial-edge*: $s \rightarrow t \Longrightarrow t \notin$ *left-tree* $s\ t$
  $\langle proof \rangle$

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* $t\ s = V$
$\langle proof \rangle$

The left and right subtrees are disjoint.

lemma *left-tree-disjoint*:
  assumes $s \rightarrow t$
  shows *left-tree* $s\ t \cap$ *left-tree* $t\ s = \{\}$
$\langle proof \rangle$

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 $s \in set\ (u \rightsquigarrow u')$
$\langle proof \rangle$

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 $t \in set\ (u \rightsquigarrow u')$

⟨*proof*⟩

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**
$parent\ v \equiv hd\ (tl\ (v \leadsto root))$

lemma *parent-edge*: $[\![v \in V; v \neq root]\!] \Longrightarrow v \rightarrow parent\ v$ ⟨*proof*⟩
lemma *parent-edge-root*: $v \rightarrow root \Longrightarrow parent\ v = root$ ⟨*proof*⟩
lemma *parent-in-V*: $[\![v \in V; v \neq root]\!] \Longrightarrow parent\ v \in V$
⟨*proof*⟩
lemma *parent-edge-cases*: $v \rightarrow w \Longrightarrow w = parent\ v \vee v = parent\ w$ ⟨*proof*⟩

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 $v \leadsto w = [v, parent\ v, w]$ (**is** - = *?xs*)
⟨*proof*⟩
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: $\bigcup \{ bag\ t \mid t.\ t \in V_T \} = V$
— Every edge is covered
and *bags-edges*: $v \rightarrow w \Longrightarrow \exists t \in V_T.\ v \in bag\ t \wedge w \in bag\ t$
— Every vertex appearing in s and u also appears in every bag on the path connecting s and u
and *bags-continuous*: $[\![s \in V_T; u \in V_T; t \in set\ (s \leadsto_T u)]\!] \Longrightarrow bag\ s \cap bag\ u \subseteq bag\ t$
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* $\equiv$ { *card* (*bag t*) | *t*. $t \in V_T$ }
definition *max-bag-card* $\equiv$ *Max bag-cards*

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

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

lemma *bags-in-V*: $t \in V_T \Longrightarrow$ *bag t* $\subseteq V$ ⟨*proof*⟩
lemma *bag-finite*: $t \in V_T \Longrightarrow$ *finite* (*bag t*) ⟨*proof*⟩
lemma *bag-bound-V*: $t \in V_T \Longrightarrow$ *card* (*bag t*) $\leq$ *card V* ⟨*proof*⟩
lemma *bag-bound-V-empty*: $[\![$ $V = \{\}$; $t \in V_T$ $]\!] \Longrightarrow$ *card* (*bag t*) $=$ *0* ⟨*proof*⟩
lemma *empty-tree-empty-V*: $V_T = \{\} \Longrightarrow V = \{\}$ ⟨*proof*⟩
lemma *bags-exist*: $v \in V \Longrightarrow \exists t \in V_T.$ $v \in$ *bag t* ⟨*proof*⟩

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* ⟨*proof*⟩
lemma *bag-cards-nonempty*: $V \neq \{\} \Longrightarrow$ *bag-cards* $\neq \{\}$
⟨*proof*⟩
lemma *max-bag-card-in-bag-cards*: $V \neq \{\} \Longrightarrow$ *max-bag-card* $\in$ *bag-cards* ⟨*proof*⟩
lemma *max-bag-card-lower-bound-bag*: $t \in V_T \Longrightarrow$ *max-bag-card* $\geq$ *card* (*bag t*)
⟨*proof*⟩
lemma *max-bag-card-lower-bound-1*: **assumes** $V \neq \{\}$ **shows** *max-bag-card* $>$ *0* ⟨*proof*⟩
lemma *max-bag-card-upper-bound-V*: $V \neq \{\} \Longrightarrow$ *max-bag-card* $\leq$ *card V* ⟨*proof*⟩

lemma *width-upper-bound-V*: $V \neq \{\} \Longrightarrow$ *width* $<$ *card V* ⟨*proof*⟩
lemma *width-V-empty*: $V = \{\} \Longrightarrow$ *width* $=$ *0* ⟨*proof*⟩
lemma *width-bound-V-le*: *width* $\leq$ *card V* $-$ *1*
⟨*proof*⟩
lemma *width-lower-bound-1*:
assumes $v \rightarrow w$
shows *width* $\geq$ *1*
⟨*proof*⟩

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* :: *nat set* **where** *treewidth-cards* $\equiv$
{ *TreeDecomposition.width T bag* | (*T* :: *nat Graph*) *bag*. *TreeDecomposition G T bag* }
definition *treewidth* :: *nat* **where** *treewidth* $\equiv$ *Min treewidth-cards*

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

proposition *tree-decomposition-exists*: $\exists (T :: 'c\ Graph)\ bag.\ TreeDecomposition\ G\ T\ bag$ ⟨*proof*⟩

corollary *treewidth-cards-upper-bound-V*: $n \in treewidth\text{-}cards \Longrightarrow n \leq card\ V - 1$
⟨*proof*⟩
corollary *treewidth-cards-finite*: *finite treewidth-cards*
⟨*proof*⟩
corollary *treewidth-cards-nonempty*: $treewidth\text{-}cards \neq \{\}$ ⟨*proof*⟩

lemma *treewidth-cards-treewidth*:
$\exists (T :: nat\ Graph)\ bag.\ TreeDecomposition\ G\ T\ bag \wedge treewidth = TreeDecomposition.width\ T\ bag$
⟨*proof*⟩

corollary *treewidth-upper-bound-V*: $treewidth \leq card\ V - 1$ ⟨*proof*⟩
corollary *treewidth-upper-bound-0*: $V = \{\} \Longrightarrow treewidth = 0$ ⟨*proof*⟩
corollary *treewidth-upper-bound-1*: $card\ V = 1 \Longrightarrow treewidth = 0$ ⟨*proof*⟩
corollary *treewidth-lower-bound-1*: $v \rightarrow w \Longrightarrow treewidth \geq 1$
⟨*proof*⟩

lemma *treewidth-upper-bound-ex*:
$[\![\ TreeDecomposition\ G\ (T :: nat\ Graph)\ bag;\ TreeDecomposition.width\ T\ bag \leq n\ ]\!] \Longrightarrow treewidth \leq n$
⟨*proof*⟩

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**
$left\text{-}part\ s\ t \equiv \bigcup \{\ bag\ u \mid u.\ u \in T.left\text{-}tree\ s\ t\ \}$
lemma *left-partI* [*intro*]: $[\![\ v \in bag\ u;\ u \in T.left\text{-}tree\ s\ t\ ]\!] \Longrightarrow v \in left\text{-}part\ s\ t$
⟨*proof*⟩

lemma *left-part-in-V*: $left\text{-}part\ s\ t \subseteq V$ ⟨*proof*⟩

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

definition *vertex-subtree* $:: 'a \Rightarrow 'c\ set$ **where**
$vertex\text{-}subtree\ v \equiv \{\ t \in V_T.\ v \in bag\ t\ \}$
lemma *vertex-subtreeI* [*intro*]: $[\![\ t \in V_T;\ v \in bag\ t\ ]\!] \Longrightarrow t \in vertex\text{-}subtree\ v$
⟨*proof*⟩

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: $s \in vertex\text{-}subtree\ v$ **and** t: $t \in vertex\text{-}subtree\ v$

and *xs*: $s \leadsto xs \leadsto_T t$
shows *set xs* $\subseteq$ *vertex-subtree v*
⟨*proof*⟩

corollary *vertex-subtree-unique-path-connected*:
assumes $v \in V$ $s \in$ *vertex-subtree v* $t \in$ *vertex-subtree v*
shows *set* $(s \leadsto_T t) \subseteq$ *vertex-subtree v*
⟨*proof*⟩

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*⟩

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

theorem *bags-separate*:
assumes *st*: $s \rightarrow_T t$ **and** *v*: $v \in$ *left-part s t* **and** *w*: $w \in$ *left-part t s* **and** *xs*: $v \leadsto xs \leadsto w$
shows *set xs* $\cap$ *bag s* $\cap$ *bag t* $\neq \{\}$
⟨*proof*⟩

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*: $s \rightarrow_T t$ **and** *vw*: $v \rightarrow w$ **and** *v*: $v \in$ *bag s* **and** *w*: $w \notin$ *bag s* $w \in$ *left-part t s*
shows $v \in$ *bag t*
⟨*proof*⟩

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* $\leq$ *1*
⟨*proof*⟩

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* :: *nat Graph* (**structure**)
assumes *Tree G card* $V_G > 1$
shows *Graph.treewidth G* = *1*
⟨*proof*⟩

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 $\bigwedge v\ w.\ [\![\ v \in V;\ w \in V;\ v \neq w\ ]\!] \Longrightarrow v \rightarrow w$
shows *treewidth* = *card V* − *1*
⟨*proof*⟩

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* ⟨*proof*⟩
interpretation *Tree-T1*: *Tree T1*
⟨*proof*⟩

The complete graph with 2 vertices.

definition *T2* ≡ ⦇ *verts* = {*u0*, *v0*}, *arcs* = {(*u0*,*v0*),(*v0*,*u0*)} ⦈
lemma *Graph-T2*: *Graph T2* ⟨*proof*⟩
lemma *Tree-T2*: *Tree T2*
⟨*proof*⟩

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*: $[\![\ v \in V_{T2};\ w \in V_{T2};\ v \neq w\ ]\!] \Longrightarrow v \rightarrow_{T2} w$ $\langle proof \rangle$
lemma *treewidth-T2*: *Graph.treewidth T2* $= 1$
$\langle proof \rangle$

The complete graph with 3 vertices.

definition *T3* $\equiv (\!|$ *verts* $= \{u0,\ v0,\ w0\}$, *arcs* $= \{(u0,v0),(v0,u0),(v0,w0),(w0,v0),(w0,u0),(u0,w0)\}$
$|\!)$
lemma *Graph-T3*: *Graph T3* $\langle proof \rangle$

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

lemma *Not-Tree-T3*: $\neg$*Tree T3* $\langle proof \rangle$

lemma *T3-complete*: $[\![\ v \in V_{T3};\ w \in V_{T3};\ v \neq w\ ]\!] \Longrightarrow v \rightarrow_{T3} w$ $\langle proof \rangle$
lemma *treewidth-T3*: *Graph.treewidth T3* $= 2$
$\langle proof \rangle$

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.