# Tree Decompositions 

Christoph Dittmann<br>christoph.dittmann@tu-berlin.de

September 13, 2023

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 ..... 4
2.3 Paths ..... 5
2.4 Cycles ..... 6
3 Trees ..... 6
3.1 Unique Connecting Path ..... 7
3.2 Separations ..... 8
3.3 Rooted Trees ..... 9
4 Tree Decompositions ..... 9
4.1 Width of a Tree Decomposition ..... 10
4.2 Treewidth of a Graph ..... 10
4.3 Separations ..... 11
5 Treewidth of Trees ..... 12
6 Treewidth of Complete Graphs ..... 13
7 Example Instantiations ..... 13
Bibliography ..... 15

## 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 $\beta: V(\mathcal{T}) \rightarrow 2^{V}$ 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

and

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

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 $p s$ be a maximal prefix such that all elements of $p s$ 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 }\times\mp@subsup{}{}{\prime}
type-synonym 'a Walk = 'a list
record 'a Graph =
    verts :: 'a set (V1)
    arcs :: 'a Edge set (E1)
abbreviation is-arc :: ('a,'b) Graph-scheme = ' }a>\mp@subsup{|}{}{\prime}a=>\mathrm{ bool (infixl }->\mathbf{1}60)\mathrm{ where
    v>G}\mp@subsup{G}{}{w}\equiv(v,w)\in\mp@subsup{E}{G}{
```

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

```
locale Graph =
    fixes \(G::\left({ }^{\prime} a,{ }^{\prime} b\right)\) 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 \(E\langle p r o o f\rangle\)
lemma edges-are-in- \(V\) : assumes \(v \rightarrow w\) shows \(v \in V w \in V\)
    \(\langle p r o o f\rangle\)
```


### 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 $[v]$
$\mid$ Cons: $v \rightarrow w \Longrightarrow$ walk $(w \# v s) \Longrightarrow$ walk $(v \# w \# v s)$

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

```
lemma walk-2 \([\operatorname{simp}]: v \rightarrow w \Longrightarrow\) walk \([v, w]\langle p r o o f\rangle\)
lemma walk-comp: 【walk xs; walk ys; xs = Nil \(\vee y s=N i l \vee l a s t x s \rightarrow h d y s \rrbracket \Longrightarrow\) walk \((x s @ y s)\)
        〈proof〉
lemma walk-tl: walk xs \(\Longrightarrow\) walk (tl xs) \(\langle p r o o f\rangle\)
lemma walk-drop: walk xs \(\Longrightarrow\) walk (drop n xs) 〈proof〉
lemma walk-take: walk \(x s \Longrightarrow\) walk (take \(n\) xs)
        〈proof〉
lemma walk-rev: walk \(x s \Longrightarrow\) walk (rev xs)
    〈proof〉
lemma walk-decomp: assumes walk (xs @ ys) shows walk xs walk ys
    〈proof〉
lemma walk-dropWhile: walk xs \(\Longrightarrow\) walk (dropWhile \(f\) xs) \(\langle p r o o f\rangle\)
lemma walk-takeWhile: walk \(x s \Longrightarrow\) walk (takeWhile \(f x s\) ) \(\langle\) proof \(\rangle\)
lemma walk-in- \(V\) : walk \(x s \Longrightarrow\) set \(x s \subseteq V\langle\) proof \(\rangle\)
lemma walk-first-edge: walk \((v \# w \# x s) \Longrightarrow v \rightarrow w\langle p r o o f\rangle\)
lemma walk-first-edge': 【walk \((v \# x s) ; x s \neq N i l \rrbracket \Longrightarrow v \rightarrow h d x s\)
    〈proof〉
lemma walk-middle-edge: walk (xs @ v\#w\#ys) \(\Longrightarrow v \rightarrow w\)
    〈proof〉
lemma walk-last-edge: 【walk (xs @ ys); xs \(\neq\) Nil; ys \(\neq\) Nil 】 \(\Longrightarrow\) last \(x s \rightarrow h d y s\)
    \(\langle p r o o f\rangle\)
lemma walk-takeWhile-edge:
    assumes walk (xs @ \([v]\) ) xs \(\neq\) Nil hd \(x s \neq v\)
    shows last (takeWhile \((\lambda x . x \neq v) x s) \rightarrow v\) (is last ? \(x s \rightarrow v\) )
\(\langle p r o o f\rangle\)
```


## 2．2 Connectivity

definition connected $::$＇$a \Rightarrow$＇$a \Rightarrow$ bool（infixl $\rightarrow$＊60）where connected $v w \equiv \exists x s$ ．walk $x s \wedge x s \neq$ Nil $\wedge h d x s=v \wedge$ last $x s=w$
lemma connectedI $[$ intro $]: \llbracket$ walk $x s ; x s \neq N i l ; h d x s=v ;$ last $x s=w \rrbracket \Longrightarrow v \rightarrow^{*} w$〈proof〉
lemma connectedE：
assumes $v \rightarrow{ }^{*} w$
obtains $x s$ where walk xs $x s \neq$ Nil hd xs $=v$ last $x s=w$
$\langle p r o o f\rangle$
lemma connected－in－$V$ ：assumes $v \rightarrow^{*} w$ shows $v \in V w \in V$
〈proof〉
lemma connected－refl：$v \in V \Longrightarrow v \rightarrow^{*} v\langle p r o o f\rangle$
lemma connected－edge：$v \rightarrow w \Longrightarrow v \rightarrow^{*} w\langle$ proof $\rangle$
lemma connected－trans：
assumes $u-v: u \rightarrow^{*} v$ and $v-w: v \rightarrow^{*} w$
shows $u \rightarrow^{*} w$
$\langle$ proof $\rangle$

## 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 $[v]\langle$ proof $\rangle$
lemma path－2［simp］：$\llbracket v \rightarrow w ; v \neq w \rrbracket \Longrightarrow$ path $[v, w]\langle$ proof $\rangle$
lemma path－cons：【path $x s ; x s \neq N i l ; v \rightarrow h d x s ; v \notin$ set $x s \rrbracket \Longrightarrow$ path $(v \# x s)$ $\langle p r o o f\rangle$
lemma path－comp：【walk xs；walk ys；xs＝Nil $\vee y s=N i l \vee$ last $x s \rightarrow h d y s ; \operatorname{distinct}(x s @ y s) \rrbracket$

$$
\Longrightarrow \text { path (xs @ ys) 〈proof〉 }
$$

lemma path－tl：path $x s \Longrightarrow$ path（tl xs）〈proof〉
lemma path－drop：path $x s \Longrightarrow$ path（drop $n x s$ ）$\langle$ proof $\rangle$
lemma path－take：path $x s \Longrightarrow$ path（take $n x s$ ）$\langle$ proof $\rangle$
lemma path－rev：path $x s \Longrightarrow$ path（rev xs）$\langle$ proof $\rangle$
lemma path－decomp：assumes path（xs＠ys）shows path xs path ys $\langle p r o o f\rangle$
lemma path－dropWhile：path $x s \Longrightarrow$ path（drop While $f x s$ ）$\langle$ proof $\rangle$
lemma path－takeWhile：path $x s \Longrightarrow$ path（takeWhile $f$ xs）〈proof〉
lemma path－in－$V$ ：path $x s \Longrightarrow$ set $x s \subseteq V\langle$ proof $\rangle$
lemma path－first－edge：path $(v \# w \# x s) \Longrightarrow v \rightarrow w\langle$ proof $\rangle$
lemma path－first－edge $: \llbracket$ path $(v \# x s) ; x s \neq$ Nil 】 $\Longrightarrow v \rightarrow h d x s\langle p r o o f\rangle$
lemma path－middle－edge：path（xs＠$v \# w \# y s) \Longrightarrow v \rightarrow w\langle$ proof $\rangle$
lemma path－takeWhile－edge：【path（xs＠$[v]) ; x s \neq N i l ; h d x s \neq v \rrbracket$
$\Longrightarrow$ last（takeWhile $(\lambda x . x \neq v) x s) \rightarrow v\langle$ proof $\rangle$
end
We introduce shorthand notation for a path connecting two vertices．

```
definition path-from-to :: (' \(a\), ' \(b\) ) Graph-scheme \(\Rightarrow{ }^{\prime} a \Rightarrow{ }^{\prime} a\) Walk \(\Rightarrow{ }^{\prime} a \Rightarrow\) bool
    (- \(-\gg 1-[71,71,71]\) 70) where
    path-from-to G vxs \(w \equiv\) Graph.path \(G x s \wedge x s \neq\) Nil \(\wedge h d x s=v \wedge\) last \(x s=w\)
context Graph begin
lemma path-from-toI [intro]: 【 path \(x s ; x s \neq N i l ; h d x s=v\); last \(x s=w \rrbracket \Longrightarrow v \rightsquigarrow x s \rightsquigarrow w\)
    and path-from-toE [dest]: \(v \rightsquigarrow x s \rightsquigarrow w \Longrightarrow\) path \(x s \wedge x s \neq\) Nil \(\wedge h d x s=v \wedge\) last \(x s=w\)
    〈proof〉
```

Every walk contains a path connecting the same vertices．
lemma walk－to－path：
assumes walk xs $x s \neq$ Nil hd $x s=v$ last $x s=w$
shows $\exists y s . v \rightsquigarrow y s \rightsquigarrow w \wedge$ set $y s \subseteq$ set $x s$
$\langle p r o o f\rangle$
corollary connected－by－path：
assumes $v \rightarrow^{*} w$
obtains $x s$ where $v \rightsquigarrow x s \rightsquigarrow w$
$\langle p r o o f\rangle$

## 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 ^ length xs > 2 ^ last xs -> hd xs
```

lemma cycleI［intro］：【 path xs；length $x s>2$ ；last $x s \rightarrow h d$ xs $\rrbracket \Longrightarrow$ cycle xs $\langle$ proof $\rangle$
lemma cycleE：cycle $x s \Longrightarrow$ path $x s \wedge x s \neq$ Nil $\wedge$ length $x s>2 \wedge$ last $x s \rightarrow h d x s$ $\langle p r o o f\rangle$

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 \(x s\) : path \((v \# x s) x s \neq\) Nil
        and ys: path \((v \# y s) y s \neq N i l\)
        and meet: last \(x s=\) last ys
        and diverge: \(h d x s \neq h d y s\)
    shows \(\exists\) zs. cycle zs \(\wedge h d z s=v\)
\(\langle p r o o f\rangle\)
```

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

```
lemma unique-paths-implies-no-cycles:
    assumes unique-paths: \(\bigwedge v w . v \rightarrow^{*} w \Longrightarrow \exists!x s . v \rightsquigarrow x s \rightsquigarrow w\)
    shows \(\bigwedge x s\). \(\neg\) cycle xs
\(\langle p r o o f\rangle\)
```

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：$\bigwedge x s$ ．$\neg$ cycle $x s$ and connected：$v \rightarrow^{*} w$
shows $\exists$ ！$x s . v \rightsquigarrow x s \rightsquigarrow w$
〈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：$\llbracket v \in V ; w \in V \rrbracket \Longrightarrow v \rightarrow^{*} w$ and no－cycles：$\neg$ cycle $x s$ 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：$\llbracket v \in V ; w \in V \rrbracket \Longrightarrow \exists!x s . v \rightsquigarrow x s \rightsquigarrow w$
〈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 $\Rightarrow{ }^{\prime} a \Rightarrow{ }^{\prime} a \Rightarrow^{\prime} a$ Walk
（infix $\rightsquigarrow 1$ 71）where unique－connecting－path $G v w \equiv$ THE $x s . v \rightsquigarrow x s \rightsquigarrow G$ w
We defined this outside the locale in order to be able to use the index in the shorthand syntax $v \rightsquigarrow$ some－index $w$ ．
context Tree begin
lemma unique－connecting－path－set：
assumes $v \in V w \in V$
shows $v \in \operatorname{set}(v \rightsquigarrow w) w \in \operatorname{set}(v \rightsquigarrow w)$
$\langle$ proof $\rangle$
lemma unique－connecting－path－properties：
assumes $v \in V w \in V$
shows path $(v \rightsquigarrow w) v \rightsquigarrow w \neq \operatorname{Nilhd}(v \rightsquigarrow w)=v$ last $(v \rightsquigarrow w)=w$〈proof〉
lemma unique－connecting－path－unique：
assumes $v \rightsquigarrow x s \rightsquigarrow w$
shows $x s=v \rightsquigarrow w$
$\langle$ proof $\rangle$
corollary unique－connecting－path－connects：$\llbracket v \in V ; w \in V \rrbracket \Longrightarrow v \rightsquigarrow(v \rightsquigarrow w) \rightsquigarrow w$〈proof〉
lemma unique－connecting－path－rev：
assumes $v \in V w \in V$
shows $v \rightsquigarrow w=\operatorname{rev}(w \rightsquigarrow v)$
〈proof〉
lemma unique－connecting－path－decomp：
assumes $v \in V w \in V v \rightsquigarrow w=p s @ u \# p s^{\prime}$
shows $p s @[u]=v \rightsquigarrow u u \# p s^{\prime}=u \rightsquigarrow w$
$\langle$ proof $\rangle$
lemma unique－connecting－path－tl：
assumes $v \in V u \in \operatorname{set}(w \rightsquigarrow v) u \rightarrow w$
shows $t l(w \rightsquigarrow v)=u \rightsquigarrow v$
$\langle p r o o f\rangle$
Every tree with at least two vertices contains an edge．

```
lemma tree-has-edge:
    assumes card \(V>1\)
    shows \(\exists v w, v \rightarrow w\)
\(\langle p r o o f\rangle\)
```


## 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 :: ' \(a \Rightarrow{ }^{\prime} a \Rightarrow{ }^{\prime} a\) set where
    left-tree \(s t \equiv\{u \in V . s \in \operatorname{set}(u \rightsquigarrow t)\}\)
lemma left-treeI \([\) intro \(]: \llbracket u \in V ; s \in \operatorname{set}(u \rightsquigarrow t) \rrbracket \Longrightarrow u \in\) left-tree st
    〈proof〉
lemma left-treeE: \(u \in\) left-tree \(s t \Longrightarrow u \in V \wedge s \in \operatorname{set}(u \rightsquigarrow t)\)
        \(\langle p r o o f\rangle\)
lemma left-tree-in- \(V\) : left-tree st \(\subseteq V\langle\) proof \(\rangle\)
lemma left-tree-initial: \(\llbracket s \in V ; t \in V \rrbracket \Longrightarrow s \in\) left-tree \(s t\)
\(\langle p r o o f\rangle\)
lemma left-tree-initial': \(\llbracket s \in V ; t \in V ; s \neq t \rrbracket \Longrightarrow t \notin\) left-tree s \(t\)
    〈proof〉
lemma left-tree-initial-edge: \(s \rightarrow t \Longrightarrow t \notin\) left-tree s \(t\)
    \(\langle p r o o f\rangle\)
```

The union of the left and right subtree is $V$ ．
lemma left－tree－union－$V$ ：
assumes $s \rightarrow t$
shows left-tree st left-tree $t s=V$
$\langle p r o o f\rangle$

The left and right subtrees are disjoint．
lemma left－tree－disjoint：
assumes $s \rightarrow t$
shows left－tree s $t \cap$ left－tree $t s=\{ \}$
$\langle p r o o f\rangle$
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->t and u:u\inleft-tree s t and }\mp@subsup{u}{}{\prime}:\mp@subsup{u}{}{\prime}\inl=left-tree t 
    shows }s\in\operatorname{set}(u\rightsquigarrow\mp@subsup{u}{}{\prime}
<proof>
```

By symmetry，the path also visits $t$ ．
corollary left－tree－separates＇：
assumes $s \rightarrow t u \in$ left－tree $s t u^{\prime} \in$ left－tree $t s$ shows $t \in \operatorname{set}\left(u \rightsquigarrow u^{\prime}\right)$

```
    <proof\rangle
```

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 \(v \equiv h d(t l(v \rightsquigarrow r o o t))\)
lemma parent-edge: \(\llbracket v \in V ; v \neq\) root \(\rrbracket \Longrightarrow v \rightarrow\) parent \(v\langle\) proof \(\rangle\)
lemma parent-edge-root: \(v \rightarrow\) root \(\Longrightarrow\) parent \(v=\) root \(\langle\) proof \(\rangle\)
lemma parent-in- \(V: \llbracket v \in V ; v \neq\) root \(\rrbracket \Longrightarrow\) parent \(v \in V\)
        〈proof〉
lemma parent-edge-cases: \(v \rightarrow w \Longrightarrow w=\) parent \(v \vee v=\) parent \(w\langle\) proof \(\rangle\)
lemma sibling-path:
        assumes \(v: v \in V v \neq\) root and \(w: w \in V w \neq\) root and \(v w: v \neq w\) parent \(v=\) parent \(w\)
        shows \(v \rightsquigarrow w=[v\), parent \(v, w]\) (is \(-=\) ? \(x s\) )
    \(\langle p r o o f\rangle\)
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 $G::\left({ }^{\prime} a,{ }^{\prime} b\right)$ Graph-scheme (structure) and $T::\left({ }^{\prime} c,{ }^{\prime} d\right)$ Graph-scheme +
fixes $b a g:: ' c \Rightarrow$ 'a set
assumes

- Every vertex appears somewhere
bags-union: $\bigcup\left\{\right.$ bag $\left.t \mid t . t \in V_{T}\right\}=V$
- Every edge is covered
and bags-edges: $v \rightarrow w \Longrightarrow \exists t \in V_{T} . v \in \operatorname{bag} t \wedge w \in \operatorname{bag} t$
- Every vertex appearing in $s$ and $u$ also appears in every bag on the path connecting $s$ and $u$ and bags-continuous: $\llbracket s \in V_{T} ; u \in V_{T} ; t \in \operatorname{set}\left(s \rightsquigarrow T_{T} u\right) \rrbracket \Longrightarrow b a g s \cap$ bag $u \subseteq b a g t$
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 $\equiv\left\{\operatorname{card}(b a g t) \mid t . t \in V_{T}\right\}$
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 $t \subseteq V\langle p r o o f\rangle$
lemma bag－finite：$t \in V_{T} \Longrightarrow$ finite（bag $t$ ）$\langle$ proof $\rangle$
lemma bag－bound－$V: t \in V_{T} \Longrightarrow$ card（bag $t$ ）$\leq$ card $V\langle$ proof $\rangle$
lemma bag－bound－$V$－empty：$\llbracket V=\{ \} ; t \in V_{T} \rrbracket \Longrightarrow$ card（bag $t$ ）$=0\langle$ proof $\rangle$
lemma empty－tree－empty－$V: V_{T}=\{ \} \Longrightarrow V=\{ \}\langle$ proof $\rangle$
lemma bags－exist：$v \in V \Longrightarrow \exists t \in V_{T} . v \in$ bag $t\langle$ proof $\rangle$
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 $\neq\{ \}$
$\langle p r o o f\rangle$
lemma max－bag－card－in－bag－cards：$V \neq\{ \} \Longrightarrow$ max－bag－card $\in$ bag－cards $\langle$ proof $\rangle$
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 $>0\langle p r o o f\rangle$
lemma max－bag－card－upper－bound－$V: V \neq\{ \} \Longrightarrow$ max－bag－card $\leq$ card $V\langle$ proof $\rangle$
lemma width－upper－bound－$V: V \neq\{ \} \Longrightarrow$ width $<$ card $V\langle$ proof $\rangle$
lemma width－V－empty：$V=\{ \} \Longrightarrow$ width $=0\langle$ proof $\rangle$
lemma width－bound－V－le：width $\leq$ card $V-1$
〈proof〉
lemma width－lower－bound－1：
assumes $v \rightarrow w$
shows width $\geq 1$
$\langle$ proof $\rangle$
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 $\mid$（ $T$ ：：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：$\exists\left(T::{ }^{\prime} c\right.$ Graph $)$ bag．TreeDecomposition $G T$ bag $\langle p r o o f\rangle$

```
corollary treewidth-cards-upper-bound-V: \(n \in\) treewidth-cards \(\Longrightarrow n \leq\) card \(V-1\)
    〈proof〉
corollary treewidth-cards-finite: finite treewidth-cards
    〈proof〉
corollary treewidth-cards-nonempty: treewidth-cards \(\neq\{ \}\langle\) proof \(\rangle\)
lemma treewidth-cards-treewidth:
    \(\exists(T\) :: nat Graph ) bag. TreeDecomposition \(G T\) bag \(\wedge\) treewidth \(=\) TreeDecomposition.width \(T\) bag
    \(\langle p r o o f\rangle\)
corollary treewidth-upper-bound- \(V\) : treewidth \(\leq\) card \(V-1\langle\) proof \(\rangle\)
corollary treewidth-upper-bound- \(0: V=\{ \} \Longrightarrow\) treewidth \(=0\langle\) proof \(\rangle\)
corollary treewidth-upper-bound-1: card \(V=1 \Longrightarrow\) treewidth \(=0\langle\) proof \(\rangle\)
corollary treewidth-lower-bound-1: \(v \rightarrow w \Longrightarrow\) treewidth \(\geq 1\)
    \(\langle p r o o f\rangle\)
lemma treewidth-upper-bound-ex:
    【TreeDecomposition \(G(T::\) nat Graph \()\) bag; TreeDecomposition.width \(T\) bag \(\leq n \rrbracket \Longrightarrow\) treewidth
\(\leq n\)
    \(\langle p r o o f\rangle\)
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 ：：＇$c \Rightarrow^{\prime} c \Rightarrow^{\prime} a$ set where left－part st $\equiv \bigcup\{$ 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 $t \rrbracket \Longrightarrow v \in$ left－part s $t$ $\langle p r o o f\rangle$
lemma left－part－in－$V$ ：left－part s $t \subseteq V\langle$ proof $\rangle$
Let us define the subgraph of $T$ induced by a vertex of $G$ ．
definition vertex－subtree ：：＇$a \Rightarrow{ }^{\prime} c$ set where vertex－subtree $v \equiv\left\{t \in V_{T} . v \in \operatorname{bag} t\right\}$
lemma vertex－subtreeI［intro］：$\llbracket t \in V_{T} ; v \in$ bag $t \rrbracket \Longrightarrow t \in$ vertex－subtree $v$〈proof〉

The suggestive name vertex－subtree is correct：Because $T$ is a tree decomposition，ver－ tex－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 $x s: s \rightsquigarrow x s \rightsquigarrow T^{t}$
shows set $x s \subseteq$ vertex－subtree $v$
$\langle p r o o f\rangle$
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 $v$
〈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 蚆t and v:v\inleft-part s t v\inleft-part t s
    shows v\inbag s v\inbagt
<proof\rangle
```

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 st and $w: w \in$ left－part $t s$ and $x s: v \rightsquigarrow x s \rightsquigarrow w$ shows set $x s \cap$ 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 $v w: v \rightarrow w$ and $v: v \in b a g s$ and $w: w \notin b a g s w \in$ left－part $t s$
shows $v \in b a g t$
$\langle p r o o f\rangle$
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 isomor－ phisms 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
<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 <br> 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 $\bigwedge v w . \llbracket v \in V ; w \in V ; v \neq w \rrbracket \Longrightarrow v \rightarrow w$
shows treewidth $=$ card $V-1$
$\langle p r o o f\rangle$
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 $=u 0|v 0| w 0$
The empty graph is a tree．
definition $T 1 \equiv($ verts $=\{ \}$ ，arcs $=\{ \}$ ）
interpretation Graph－T1：Graph T1〈proof〉
interpretation Tree－T1：Tree T1
〈proof〉
The complete graph with 2 vertices．
definition $T 2 \equiv 0$ verts $=\{u 0, v 0\}$ ，arcs $=\{(u 0, v 0),(v 0, u 0)\} D$
lemma Graph－T2：Graph T2 〈proof〉
lemma Tree－T2：Tree T2
$\langle p r o o f\rangle$

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 be－ cause 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：$\llbracket v \in V_{T 2} ; w \in V_{T 2} ; v \neq w \rrbracket \Longrightarrow v \rightarrow{ }_{T 2} w\langle$ proof $\rangle$
lemma treewidth－T2：Graph．treewidth T2 $=1$
〈proof〉
The complete graph with 3 vertices．
definition $T 3 \equiv 0$ verts $=\{u 0, v 0, w 0\}$, arcs $=\{(u 0, v 0),(v 0, u 0),(v 0, w 0),(w 0, v 0),(w 0, u 0),(u 0, w 0)\}$ ）
lemma Graph－T3：Graph T3 〈proof〉
$[u 0, v 0, w 0]$ is a cycle in $T 3$ ，so $T 3$ is not a tree．
lemma Not－Tree－T3：$\neg$ Tree T3 $\langle$ proof〉
lemma T3－complete：$\llbracket v \in V_{T 3} ; w \in V_{T 3} ; v \neq w \rrbracket \Longrightarrow v \rightarrow{ }_{T 3} w\langle$ proof $\rangle$
lemma treewidth－T3：Graph．treewidth T3＝ 2 $\langle p r o o f\rangle$

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.

