# 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 ..... 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 $(\mathcal{T}, \beta)$ 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\mathrm{ is the vertex type.
type-synonym 'a Edge = 'a < 'a
type-synonym 'a Walk = 'a list
record 'a Graph =
    verts :: 'a set (V1)
    arcs :: 'a Edge set (E1)
abbreviation is-arc :: ('a,'b) Graph-scheme = ' }a>\mp@subsup{|}{}{\prime}a=>\mathrm{ bool (infixl }->160\mathrm{ ) where
    v>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 \(E\) 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 $[v]$
$\mid$ Cons: $v \rightarrow w \Longrightarrow$ walk $(w \#$ vs $) \Longrightarrow$ 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 $[v, w]$ by (simp add: edges-are-in-V(2) walk.intros(3))
lemma walk-comp: 【 walk xs; walk ys; xs = Nil $\vee y s=N i l \vee$ last $x s \rightarrow h d y s \rrbracket \Longrightarrow$ walk $(x s @ y s)$
by (induct rule: walk.induct, simp-all add: walk.intros(3))
(metis list.exhaust-sel walk.intros(2) walk.intros(3))
lemma walk-tl: walk $x s \Longrightarrow$ walk ( $t l x s$ ) by (induct rule: walk.induct) simp-all
lemma walk-drop: walk $x s \Longrightarrow$ walk (drop $n$ xs) by (induct $n$, simp) (metis drop-Suc tl-drop walk-tl)
lemma walk-take: walk xs $\Longrightarrow$ walk (take $n$ 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: walk xs $\Longrightarrow$ walk (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 walk (xs @ ys) shows walk xs walk ys
using assms append-eq-conv-conj[of xs ys xs @ ys] walk-take walk-drop by metis+
lemma walk-drop While: walk xs $\Longrightarrow$ walk (dropWhile fxs) by (simp add: walk-drop dropWhile-eq-drop)
lemma walk-takeWhile: walk $x s \Longrightarrow$ walk (takeWhile $f$ xs) using walk-take takeWhile-eq-take by metis
lemma walk-in- $V$ : walk $x s \Longrightarrow$ set $x s \subseteq V$ by (induct rule: walk.induct; simp add: edges-are-in- $V$ )
lemma walk-first-edge: walk $(v \# w \# x s) \Longrightarrow v \rightarrow w$ using walk.cases by fastforce
lemma walk-first-edge': $\llbracket$ walk $(v \# x s) ; x s \neq N i l \rrbracket \Longrightarrow v \rightarrow h d x s$
using walk-first-edge by (metis list.exhaust-sel)
lemma walk-middle-edge: walk (xs @ $v \# w \# y s) \Longrightarrow v \rightarrow w$
by (induct xs @ $v \# w \#$ ys arbitrary: xs rule: walk.induct, simp, simp)
(metis list.sel(1,3) self-append-conv2 tl-append2)
lemma walk-last-edge: 【 walk (xs @ ys); xs $\neq$ Nil; ys $\neq$ Nil $\rrbracket ~ l a s t ~ x s ~ \rightarrow h d ~ y s ~$
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 walk (xs @ \([v])\) xs \(\neq\) Nil hd \(x s \neq v\)
    shows last \((\) take While \((\lambda x . x \neq v) x s) \rightarrow v(\) is last ? \(x s \rightarrow v)\)
proof-
    obtain \(x s^{\prime}\) where \(x s^{\prime}: x s=\) ? \(x s\) @ \(x s^{\prime}\) by (metis takeWhile-drop While-id)
    thus ?thesis proof (cases)
        assume \(x s^{\prime}=N i l\) thus ?thesis using \(x s^{\prime}\) assms \((1,2)\) walk-last-edge by force
    next
        assume \(x s^{\prime} \neq\) Nil
        hence \(h d x s^{\prime}=v\) by (metis (full-types) hd-dropWhile same-append-eq takeWhile-dropWhile-id
    \(x s^{\prime}\) )
        thus ?thesis by (metis \(\left\langle x s^{\prime} \neq[]\right\rangle\) append-Nil assms(1,3) walk-decomp(1) walk-last-edge xs')
    qed
qed
```


## 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 connected $I[$ intro $]$ ：$\llbracket$ walk $x s ; x s \neq N i l ; h d x s=v ;$ last $x s=w \rrbracket \Longrightarrow v \rightarrow^{*} w$ unfolding connected－def by blast
lemma connectedE：
assumes $v \rightarrow^{*} w$
obtains $x s$ where walk xs $x s \neq$ Nil hd xs $=v$ last $x s=w$ 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：$v \in V \Longrightarrow v \rightarrow^{*} v$ by（rule connectedI［of［v］］）simp－all
lemma connected－edge：$v \rightarrow w \Longrightarrow v \rightarrow^{*} w$ by（rule connectedI［of $\left.[v, w]\right]$ ）simp－all
lemma connected－trans：
assumes $u-v: u \rightarrow^{*} v$ and $v-w: v \rightarrow^{*} w$
shows $u \rightarrow^{*} w$
proof－
obtain $x s$ where $x s$ ：walk $x s x s \neq$ Nil hd $x s=u$ last $x s=v$ using $u$－v connectedE by blast
obtain ys where ys：walk ys ys $\neq$ Nil hd ys $=v$ last $y s=w$ using $v$－$w$ connectedE by blast
let ？$R=x s @ t l y s$
show ？thesis proof
show walk ？R using walk－comp $[O F x s(1)]$ by（metis $x s(4) y s(1,2,3)$ list．sel $(1,3)$ walk．simps $)$
show $? R \neq$ Nil by（simp add：xs（2））
show $h d ? R=u$ by（simp add：$x s(2,3))$
show last ？$R=w$ using $x s(2,4) y s(2,3,4)$
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 $\Rightarrow$ bool where path $x s \equiv$ walk $x s \wedge$ distinct $x s$
lemma path－singleton $[$ simp $]: v \in V \Longrightarrow$ path $[v]$ by simp
lemma path－2［simp］：【v $u w ; v \neq w \rrbracket \Longrightarrow$ path $[v, w]$ by simp
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（ $v \# x s$ ）
by（metis distinct．simps（2）list．exhaust－sel walk．Cons）
lemma path－comp：【walk xs；walk ys；xs＝Nil $\vee y s=N i l \vee$ last $x s \rightarrow h d y s ; \operatorname{distinct}(x s @ y s) \rrbracket$
$\Longrightarrow$ path（xs＠ys）using walk－comp by blast
lemma path－tl：path $x s \Longrightarrow$ path（ $t l$ xs）by（simp add：distinct－tl walk－tl）
lemma path－drop：path $x s \Longrightarrow$ path（drop $n$ xs）by（simp add：walk－drop）
lemma path－take：path $x s \Longrightarrow$ path（take $n$ xs）by（simp add：walk－take）
lemma path－rev：path $x s \Longrightarrow$ path（rev xs）by（simp add：walk－rev）
lemma path－decomp：assumes path（xs＠ys）shows path xs path ys
using walk－decomp assms distinct－append by blast＋
lemma path－dropWhile：path $x s \Longrightarrow$ path（dropWhile $f$ xs）by（simp add：walk－dropWhile）
lemma path－takeWhile：path $x s \Longrightarrow$ path（takeWhile $f$ xs）by（simp add：walk－takeWhile）
lemma path-in- $V$ : path $x s \Longrightarrow$ set $x s \subseteq V$ by (simp add: walk-in- $V$ )
lemma path-first-edge: path $(v \# w \# x s) \Longrightarrow v \rightarrow w$ using walk-first-edge by blast
lemma path-first-edge': 【 path ( $v \# x s) ; x s \neq N i l \rrbracket \Longrightarrow v \rightarrow h d x s$ using walk-first-edge' by blast
lemma path-middle-edge: path $(x s @ v \# w \# y s) \Longrightarrow v \rightarrow w$ using walk-middle-edge by blast
lemma path-takeWhile-edge: 【path (xs @ $[v]) ; x s \neq N i l ; h d x s \neq v \rrbracket$
$\Longrightarrow$ last (takeWhile $(\lambda x . x \neq v) x s) \rightarrow v$ using walk-takeWhile-edge by blast
end
We introduce shorthand notation for a path connecting two vertices.

```
definition path-from-to :: ('a, 'b) Graph-scheme = ' }a=>\mathrm{ ' ' Walk = ' }a=>\mathrm{ bool
    (- \rightsquigarrow-\rightsquigarrow1-[71, 71, 71] 70) where
    path-from-to G v xs w\equivGraph.path Gxs ^ xs \not=Nil^hd xs=v^ last xs =w
context Graph begin
lemma path-from-toI [intro]:\llbracket path xs; xs \not=Nil; hd xs =v; last xs =w\rrbracket\Longrightarrowv v wxs\rightsquigarroww
    and path-from-toE [dest]:v\rightsquigarrowxs\rightsquigarroww\Longrightarrow path xs ^xs F Nil ^hd xs=v^ last xs=w
    unfolding path-from-to-def by blast+
```

Every walk contains a path connecting the same vertices.
lemma walk-to-path:
assumes walk xs xs $\neq$ Nil hd xs $=v$ last $x s=w$
shows $\exists y s . v \rightsquigarrow y s \rightsquigarrow w \wedge$ set $y s \subseteq$ set $x s$
proof-
We prove this by removing loops from $x s$ until $x s$ is a path. We want to perform induction over length $x s$, but $x s$ in set $y s \subseteq$ set $x s$ should not be part of the induction hypothesis. To accomplish this, we hide set $x s$ behind a definition for this specific part of the goal.

```
define target-set where target-set \(=\) set \(x s\)
hence set \(x s \subseteq\) target-set by simp
thus \(\exists y s . v \rightsquigarrow y s \rightsquigarrow w \wedge\) set \(y s \subseteq\) target-set
    using assms
proof (induct length xs arbitrary: xs rule: infinite-descent0)
    case (smaller \(n\) )
    then obtain \(x s\) where
```

    xs: \(n=\) length \(x s\) walk xs \(x s \neq\) Nil hd \(x s=v\) last \(x s=w\) set \(x s \subseteq\) target-set and
    hyp: \(\neg(\exists y s . v \rightsquigarrow y s \rightsquigarrow w \wedge\) set \(y s \subseteq\) target-set \()\) by blast
    If $x s$ is not a path, then $x s$ is not distinct and we can decompose it.

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

$u$ appears in $x s$, so we have a loop in $x s$ starting from an occurrence of $u$ in $x s$ ending in the vertex $u$ in $u \# y s$. We define $z s$ as $x s$ without this loop.

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

```
    moreover have hd zs' = v unfolding zs'-def
        by (metis append-is-Nil-conv hd-append list.sel(1) xs(4) xs-decomp(3) ys-decomp)
    moreover have last zs' = w unfolding zs'-def using xs(5) xs-decomp(3) by auto
    moreover have set zs'\subseteqtarget-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 ->* w
    obtains xs where v}\rightsquigarrowxs\rightsquigarrow
    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 $x s \equiv$ path $x s \wedge$ length $x s>2 \wedge$ last $x s \rightarrow h d x s$
lemma cycleI [intro]: 【 path xs; length $x s>2 ;$ last $x s \rightarrow h d$ xs $\rrbracket \Longrightarrow$ cycle xs unfolding cycle-def by blast
lemma cycleE: cycle $x s \Longrightarrow$ path $x s \wedge x s \neq$ Nil $\wedge$ length $x s>2 \wedge$ last $x s \rightarrow h d x s$ 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 \(x s\) : path \((v \# x s) x s \neq\) Nil
        and ys: path \((v \# y s) y s \neq\) Nil
        and meet: last \(x s=\) last ys
        and diverge: \(h d x s \neq h d\) ys
    shows \(\exists\) zs. cycle zs \(\wedge h d z s=v\)
proof-
    have set \(x s \cap\) set \(y s \neq\{ \}\) using meet xs(2) ys(2) last-in-set by fastforce
    then obtain \(x s^{\prime} x x s^{\prime \prime}\) where \(x s^{\prime}: x s=x s^{\prime} @ x \# x s^{\prime \prime}\) set \(x s^{\prime} \cap\) set \(y s=\{ \} x \in\) set \(y s\)
        using split-list-first-prop[of xs \(\lambda x . x \in\) set \(y s]\) by (metis disjoint-iff-not-equal)
    then obtain \(y s^{\prime} y s^{\prime \prime}\) where \(y s^{\prime}: y s=y s^{\prime} @ x \# y s^{\prime \prime} x \notin\) set \(y s^{\prime}\)
        using split-list-first-prop[of ys \(\lambda y . y=x]\) by blast
    let? \(z s=v \# x s^{\prime} @ x \#\left(r e v y s^{\prime}\right)\)
    have last? zs \(\rightarrow h d\) ? zs
        using undirected walk-first-edge walk-first-edge' ys \(^{\prime}(1)\) ys(1) by (fastforce simp: last-rev)
    moreover have path? zs proof
        have walk ( \(x\) \# rev ys') proof(cases)
```

assume $y s^{\prime}=$ Nil thus ？thesis using〈last ？zs $\rightarrow h d$ ？zs〉 edges－are－in－$V(1)$ by auto
next
assume $y s^{\prime} \neq$ Nil
moreover hence last $y s^{\prime} \rightarrow x$ using walk－last－edge walk－tl ys＇（1）ys（1）by fastforce
moreover have $h d\left(\right.$ rev ys $\left.s^{\prime}\right)=$ last ys ${ }^{\prime}$ by（simp add：$\left\langle y s^{\prime} \neq[]\right\rangle$ hd－rev $)$
moreover have walk（rev ys＇）by（metis list．sel（3）walk－decomp（1）walk－rev walk－tl ys＇（1） $y s(1))$
ultimately show walk（ $x$ \＃rev ys＇）using path－cons undirected $y s^{\prime}(1) y s(1)$ by auto qed
thus walk（ $v \# x s^{\prime}$＠$x \#$ rev $\left.y s^{\prime}\right)$ using $x s^{\prime}(1) x s(1)$
by（metis append－Cons list．sel（1）list．simps（3）walk－comp walk－decomp（1）walk－last－edge）
next
show distinct（ $v \# x s^{\prime} @ x \#$ rev ys＇）unfolding distinct－append distinct．simps（2）set－append using $x s^{\prime}(1,2) x s(1) y s^{\prime}(1) y s(1)$ by auto
qed
moreover have length ？zs $\neq 2$ using diverge $x s^{\prime}(1) y s^{\prime}(1)$ by auto
ultimately show ？thesis using cycleI $[o f$ ？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：$\bigwedge v w . v \rightarrow^{*} w \Longrightarrow \exists!x s . v \rightsquigarrow x s \rightsquigarrow w$
shows $\bigwedge x s$ ．$\neg$ cycle $x s$
proof
fix $x s$ assume cycle xs
let ？$v=h d x s$
let ？$w=$ last $x s$
let $? y s=[? v, ? w]$
define good where good $x s \longleftrightarrow$ ？$v \rightsquigarrow x s \rightsquigarrow$ ？$w$ for $x s$
have path？？ys using «cycle xs〉 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〈cycle xs〉 cycleE）
moreover have ？$y s \neq x s$ using $\langle$ cycle $x s\rangle$
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 «cycle xs〉 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：$\bigwedge x s . \neg$ cycle $x s$ and connected：$v \rightarrow^{*} w$
shows $\exists$ ！$x s . v \rightsquigarrow x s \rightsquigarrow w$
proof（rule ex－ex1I）
show $\exists x s . v \rightsquigarrow x s \rightsquigarrow w$ using connected connected－by－path by blast
next
fix $x s$ ys
assume $v \rightsquigarrow x s \rightsquigarrow w v \rightsquigarrow y s \rightsquigarrow w$
hence $x s$－valid：path xs $x s \neq$ Nil hd $x s=v$ last $x s=w$
and ys－valid：path ys ys $\neq$ Nil hd ys $=v$ last $y s=w$ by blast +

```
    show xs = ys proof (rule ccontr)
    assume xs \not= ys
    hence \existspsx\mp@subsup{s}{}{\prime}y\mp@subsup{s}{}{\prime}.xs=ps@ x\mp@subsup{s}{}{\prime}^ys=ps@ys'^(x\mp@subsup{s}{}{\prime}=Nil\veey\mp@subsup{s}{}{\prime}=Nil\veehdx\mp@subsup{s}{}{\prime}\not==hdy\mp@subsup{s}{}{\prime})
        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' x\mp@subsup{s}{}{\prime}=Nil\veeys'=Nil\veehd x\mp@subsup{s}{}{\prime}\not=hdy\mp@subsup{s}{}{\prime}}\mathbf{b}\mathrm{ 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'}\not=N\mathrm{ Nil using <xs }\not=ys>ys-valid(1,4) ps(1,2) xs-valid(4) by aut
    have last ys \in set ps if ys' = Nil using ys-valid(2) ps(2) by (simp add: that)
    hence ys-not-nil: ys' = Nil using «xs \not= ys` xs-valid(1,4) ps(1,2) ys-valid(4) by auto
    have \existszs.cycle zs proof-
        let ?v = last ps
        have *: ps \not=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: $\llbracket v \in V ; w \in V \rrbracket \Longrightarrow v \rightarrow^{*} w$ 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:\llbracketv\inV;w\inV\rrbracket\Longrightarrow\exists!xs.v\rightsquigarrowxs\rightsquigarroww
    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 $\Rightarrow{ }^{\prime} a \Rightarrow{ }^{\prime} a \Rightarrow{ }^{\prime} a$ Walk
(infix $\rightsquigarrow 1$ 71) where unique-connecting-path $G v w \equiv T H E x s . v \rightsquigarrow x s \rightsquigarrow G$ 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 $v \in \operatorname{set}(v \rightsquigarrow w) w \in \operatorname{set}(v \rightsquigarrow w)$
using theI'[OF unique-connecting-path[OF assms], folded unique-connecting-path-def] $h d$-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 the $I^{\prime}[$ OF unique-connecting-path[ OF assms], folded unique-connecting-path-def] by blast+
lemma unique-connecting-path-unique:
assumes $v \rightsquigarrow x s \rightsquigarrow w$
shows $x s=v \rightsquigarrow w$
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 $v \rightsquigarrow w=\operatorname{rev}(w \rightsquigarrow v)$
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 $v \in V w \in V v \rightsquigarrow w=p s @ u \# p s^{\prime}$
shows $p s @[u]=v \rightsquigarrow u u \# p s^{\prime}=u \rightsquigarrow w$
proof-
have $h d(p s @[u])=v$
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 $(p s @[u])=u p s @[u] \neq$ Nil by simp-all

```
    ultimately show \(p s\) @ \([u]=v \rightsquigarrow u\) using unique-connecting-path-unique by blast
next
    have last ( \(u \# p s^{\prime}\) ) \(=w\)
        using assms unique-connecting-path-properties(4) by fastforce
    moreover have path ( \(u \# p s^{\prime}\) ) using unique-connecting-path-properties(1)[OF \(\left.\operatorname{assms}(1,2)\right]\)
    unfolding assms(3) using path-decomp(2) by blast
    moreover have \(h d\left(u \# p s^{\prime}\right)=u u \# p s^{\prime} \neq\) Nil by simp-all
    ultimately show \(u \# p s^{\prime}=u \rightsquigarrow w\) using unique-connecting-path-unique by blast
qed
lemma unique-connecting-path-tl:
    assumes \(v \in V u \in \operatorname{set}(w \rightsquigarrow v) u \rightarrow w\)
    shows \(t l(w \rightsquigarrow v)=u \rightsquigarrow v\)
proof (rule ccontr)
    assume contra: \(\neg\) ?thesis
    from \(\operatorname{assms}\) (2) obtain \(p s p s^{\prime}\) where
        \(p s: w \rightsquigarrow v=p s @ u \# p s^{\prime}\) by (meson split-list)
    have cycle ( \(p s\) @ [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 \(t l(w \rightsquigarrow v)=u \# p s^{\prime}\)
            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 \(u \# p s^{\prime}=u \rightsquigarrow v\)
                    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]) \(\rightarrow h d(p s\) @ [u]) using \(\operatorname{assms(3)}\)
            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}->hd(tl(v\rightsquigarroww))\mathrm{ 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 :: ' \(a \Rightarrow{ }^{\prime} a \Rightarrow{ }^{\prime} a\) 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
    unfolding left-tree-def by blast
lemma left-treeE: \(u \in\) left-tree \(s t \Longrightarrow u \in V \wedge s \in \operatorname{set}(u \rightsquigarrow t)\)
    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 st $u \notin$ 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 ? $x s=s \rightsquigarrow u$
let ? $y s=t \rightsquigarrow u$
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 $u(1, \mathcal{Q})$ 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 \(x s^{\prime}\) where \([s i m p]: x s^{\prime}=\) ? \(x s\)
define \(y s^{\prime}\) where \([s i m p]: y s^{\prime}=s \#\) ?ys
have path ys' using path-cons «s \(\notin\) set ? ys \(\downarrow\) assms
    by (simp add: unique-connecting-path-properties(1-3) tu(1))
moreover have path \(x s^{\prime} x s^{\prime} \neq[] y^{\prime} \neq[] \quad h d x s^{\prime}=s\) last \(x s^{\prime}=u\)
    by (simp-all add: unique-connecting-path-properties s u(1))
moreover have \(h d y s^{\prime}=s\) last \(y s^{\prime}=u\)
    by simp (simp add: unique-connecting-path-properties(2,4) tu(1))
    moreover have \(x s^{\prime} \neq y s^{\prime}\) using unique-connecting-path-set(1) \(\langle t \notin\) set ? \(x s\rangle t u(1)\) 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 \(u: u \in V s \in \operatorname{set}(u \rightsquigarrow t) t \in \operatorname{set}(u \rightsquigarrow s)\) 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: \(u \rightsquigarrow t=p s\) @ \(s \# p s^{\prime}\) by (meson split-list \(u(2)\) )
    hence \(p s^{\prime} \neq\) Nil
        using assms last-snoc no-loops unique-connecting-path-properties(4) \([\) OF u(1) t] by auto
    hence \(*\) : length ( \(p s\) @ \([s]\) ) < length ( \(u \rightsquigarrow t\) ) by (simp add: ps)
    have $p s^{\prime}: p s$ @ $[s]=u \rightsquigarrow s$ using $p s$ unique-connecting-path-decomp $t u(1)$ by blast
then obtain $q s q s^{\prime}$ where $q s$ : $p s$ @ $[s]=q s$ @ $t \# q s^{\prime}$ using split-list $[O F u(3)]$ by auto
hence $q s^{\prime} \neq$ Nil using assms last-snoc no-loops by auto
hence $* *$ : length ( $q s$ @ $[t])<$ length ( $p s$ @ $[s]$ ) by (simp add: $q s$ )
have $q s$ @ $[t]=u \rightsquigarrow t$ using $q s p s^{\prime}$ 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->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 (rule ccontr)
    assume \neg?thesis
    with assms have set ( }u\rightsquigarrow\mp@subsup{u}{}{\prime})\subseteql\mathrm{ left-tree s t
    proof (induct u}\rightsquigarrow\mp@subsup{u}{}{\prime}\mathrm{ arbitrary: }u\mp@subsup{u}{}{\prime}
```

```
case Nil thus ?case using unique-connecting-path-properties(2) by auto
next
    case (Cons x xs u u')
    have }x=u\mathrm{ using Cons.hyps(2) Cons.prems(2,3)
        by (metis left-treeE list.sel(1) unique-connecting-path-properties(3))
    hence }u->hd\mathrm{ xs using Cons.hyps(2) Cons.prems(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\inVhdxs\inV using edges-are-in-V by blast
    have *: xs = hd xs \rightsquigarrow u'
        by (metis Cons.hyps(2) Cons.prems(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\not\inset (hd xs \rightsquigarrow u') using Cons.hyps(2) Cons.prems(4)
        by (metis list.set-intros(2))
    moreover have hd xs \inleft-tree st proof (rule ccontr)
        assume }\neg\mathrm{ ?thesis
        hence hd xs \in left-tree ts using <hd xs \inV> st left-tree-union-V by fastforce
        hence t\in set (hd xs \rightsquigarrows) using left-treeE by blast
        let ?ys' = hd xs }\rightsquigarrow
        let ?ys = u# ?ys'
        have u\not\in set ?ys' proof
            assume u\in set ?ys'
            hence tl ?ys' = u\rightsquigarrows
                using unique-connecting-path-tl <u->hd xs` edges-are-in-V(1) st by auto
            moreover have t\not=hd xs proof
                let ?ys = [u,hd xs]
                have t\not=u using Cons.prems(2) left-tree-initial-edge st by blast
                assume t=hd xs
                hence ?ys = u\rightsquigarrowt
                    using unique-connecting-path-unique[of u ?ys hd xs] <u->hd xs\rangle\langlet\not=u\rangle
                by (simp add: path-from-toI)
            hence s}\not\inset ( u\rightsquigarrowt
                        by (metis Cons.hyps(2) Cons.prems(4) <t = hd xs\rangle\langlex=u\rangledistinct.simps(2)
                        distinct-singleton list.set-intros(1) no-loops set-ConsD st)
            thus False using Cons.prems(2) left-treeE by blast
        qed
        ultimately have t\in set (u\rightsquigarrows) using <t \in set ?ys'\rangle\langlehd xs \inV>st
        by (metis edges-are-in-V(1) unique-connecting-path-properties(2,3) list.collapse set-ConsD)
        thus False using Cons.prems(2) st }\langleu\inV
            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 }\not=\mathrm{ Nil hd ?ys =u by simp-all
    moreover have last ?ys =s using st unique-connecting-path-properties(2,4)<hd xs \inV`
        by (simp add: edges-are-in-V(1))
    ultimately have ?ys =u\rightsquigarrows using unique-connecting-path-unique by blast
    hence t\inset (u\rightsquigarrows) by (metis <t \in set ?ys'> list.set-intros(2))
    thus False using Cons.prems(2) <u\inV\ranglest
        by (meson left-tree-disjoint disjoint-iff-not-equal left-treeI)
    qed
```

```
    ultimately have set (hd xs \rightsquigarrow u')\subseteq left-tree st
        using Cons.hyps(1) st Cons.prems(3) by blast
    hence set xs\subseteqleft-tree s t using * by simp
    thus ?case using Cons.hyps(2) Cons.prems(2,3)
        by (metis insert-subset left-treeE list.sel(1) list.set(2) unique-connecting-path-properties(3))
    qed
    hence }\mp@subsup{u}{}{\prime}\inleft-tree s t using left-treeE u u' unique-connecting-path-set(2) by aut
    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 \(s t u^{\prime} \in\) left-tree \(t s\)
shows \(t \in\) set ( \(u \rightsquigarrow u^{\prime}\) )
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 \(v \equiv h d(t l(v \rightsquigarrow r o o t))\)
    lemma parent-edge: \(\llbracket v \in V ; v \neq\) root \(\rrbracket \Longrightarrow v \rightarrow\) parent \(v\) 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- \(V: \llbracket v \in V ; v \neq\) root \(\rrbracket \Longrightarrow\) 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 \(v w: v \neq w\) parent \(v=\) parent \(w\)
        shows \(v \rightsquigarrow w=[v\), parent \(v, w]\) (is \(-=\) ? \(x s\) )
    proof -
        have path ?xs using \(v w v w\)
            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<br>imports Tree begin

A tree decomposition of a graph.
locale TreeDecomposition $=$ Graph $G+T$ : Tree $T$
for $G::(' a, ~ ' b)$ Graph-scheme (structure) and $T::\left({ }^{\prime} c,{ }^{\prime} d\right)$ Graph-scheme +
fixes $b a y:: ' c \Rightarrow$ '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: $\llbracket s \in V_{T} ; u \in V_{T} ; t \in \operatorname{set}(s \rightsquigarrow T u) \rrbracket \Longrightarrow b a g s \cap b a g u \subseteq b a g t$
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 $t \subseteq V$ using bags-union Sup-upper mem-Collect-eq by blast lemma bag-finite: $t \in V_{T} \Longrightarrow$ finite (bag t) using bags-in- $V$ finite-subset finite-vertex-set by blast lemma bag-bound- $V: t \in V_{T} \Longrightarrow$ card (bag $t$ ) $\leq$ card $V$ by (simp add: bags-in- $V$ card-mono finite-vertex-set)
lemma bag-bound- $V$-empty: $\llbracket V=\{ \} ; t \in V_{T} \rrbracket \Longrightarrow$ card (bag $t$ ) $=0$ using bag-bound- $V$ by auto lemma empty-tree-empty- $V: V_{T}=\{ \} \Longrightarrow V=\{ \}$ using bags-union by simp
lemma bags-exist: $v \in V \Longrightarrow \exists t \in V_{T} . v \in$ 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: $V \neq\{ \} \Longrightarrow$ 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: $V \neq\{ \} \Longrightarrow$ max-bag-card $\in$ 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: $t \in V_{T} \Longrightarrow$ max-bag-card $\geq$ card (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 max-bag-card $>0$ proof-
have $\exists v \in V . \exists t \in V_{T} . v \in$ bag $t$ using $\langle V \neq\{ \}\rangle$ bags-union by blast
thus max-bag-card $>0$ unfolding max-bag-card-def using bag-finite
card-gt-O-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$\neq\{ \} \Longrightarrow$ max-bag-card $\leq$ 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- $V: V \neq\{ \} \Longrightarrow$ width $<$ 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: $V=\{ \} \Longrightarrow$ 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: width $\leq$ 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 width $\geq 1$
proof-
obtain $t$ where $t: t \in V_{T} v \in$ bag $t w \in$ bag $t$ using bags-edges assms by blast
have card (bag $t) \neq 0$ using $t(1,2)$ bag-finite card- 0 -eq empty-iff by blast
moreover have card $(\operatorname{bag} t) \neq 1$ using $t(2,3)$ assms no-loops
by (metis One-nat-def card-Suc-eq empty-iff insertE)
ultimately have card (bag t) $\geq 2$ by $\operatorname{simp}$
hence 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 :: 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 proof-
obtain $x$ where $x \in(U N I V ~:: ~ ' c ~ s e t) ~ b y ~ b l a s t ~$
define $T$ where $[$ simp $]: T=\{$ verts $=\{x\}$, arcs $=\{ \}$ D
define $b a g$ where $[$ simp $]: b a g=\left(\lambda-::{ }^{\prime} c . V\right)$
have Graph $T$ by unfold-locales simp-all

```
    then interpret T: Graph T.
    have \xs. }\neg\mathrm{ T.cycle xs using T.cycleE by auto
    moreover have }\vw.v\in\mp@subsup{V}{T}{}\Longrightarroww\in\mp@subsup{V}{T}{}\Longrightarrow\mathrm{ 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 treewidth-cards \Longrightarrown\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 }\not={}\mathrm{ by (simp add: tree-decomposition-exists)
lemma treewidth-cards-treewidth:
    \exists(T :: nat Graph) bag. TreeDecomposition G T bag ^ 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 = {}\Longrightarrow treewidth = 0 using treewidth-upper-bound-V by
simp
corollary treewidth-upper-bound-1: card V = 1 \Longrightarrow treewidth = 0 using treewidth-upper-bound-V
by simp
corollary treewidth-lower-bound-1:v->w \Longrightarrow treewidth \geq1
    using TreeDecomposition.width-lower-bound-1 treewidth-cards-treewidth by fastforce
lemma treewidth-upper-bound-ex:
    \llbracketTreeDecomposition G (T :: nat Graph) bag; TreeDecomposition.width T bag \leqn\rrbracket\Longrightarrow treewidth
\leqn
    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 : : ' $c \Rightarrow^{\prime} c \Rightarrow^{\prime} a$ 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 $t \rrbracket \Longrightarrow v \in$ left-part st
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 :: ' \(a \Rightarrow{ }^{\prime} c\) set where
    vertex-subtree \(v \equiv\left\{t \in V_{T} . v \in \operatorname{bag} t\right\}\)
lemma vertex-subtree \(I\) [intro]: \(\llbracket t \in V_{T} ; v \in\) bag \(t \rrbracket \Longrightarrow t \in\) vertex-subtree \(v\)
    unfolding vertex-subtree-def by blast
```

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 \(v\)
using assms proof (induct xs arbitrary: s)
    case (Cons \(x\) xs)
    show ?case proof (cases)
        assume \(x s=[]\) thus ?thesis using Cons.prems \((3,4)\) by auto
    next
        assume \(x s \neq[]\)
        moreover hence last \(x s=t\) using Cons.prems(4) last.simps by auto
        moreover have T.path xs using Cons.prems(4) T.walk-tl by fastforce
        moreover have \(h d x s \in\) vertex-subtree \(v\) proof
            have \(h d x s \in \operatorname{set}(s \rightsquigarrow T t)\) using T.unique-connecting-path-unique
                using Cons.prems(4) <xs \(\neq[]\rangle\) by auto
            hence bag \(s \cap\) bag \(t \subseteq b a g(h d x s)\)
                using bags-continuous Cons.prems(4) T.connected-in-V by blast
            thus \(v \in\) bag ( \(h d x s\) ) using Cons.prems (2,3) unfolding vertex-subtree-def by blast
            show hd \(x s \in V_{T}\) using T.connected-in- \(V(1)\langle x s \neq[]\rangle\langle T\). path \(x s\rangle\) by blast
        qed
        ultimately have set \(x s \subseteq\) vertex-subtree \(v\) using Cons.hyps Cons.prems \((1,3)\) by blast
        thus ?thesis using Cons.prems \((2,4)\) 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 \(v \in b a g\) s \(v \in b a g t\)
proof-
    obtain \(u u^{\prime}\) where \(u: v \in\) bag \(u u \in\) T.left-tree st \(v \in b a g u^{\prime} u^{\prime} \in T\).left-tree \(t s\)
        using \(v\) unfolding left-part-def by blast
    have \(s \in \operatorname{set}\left(u \rightsquigarrow T^{\prime} u^{\prime}\right)\) using T.left-tree-separates st \(u\) by blast
```

```
    thus }v\inbag s using bags-continuous u by (meson IntI T.left-treeE subsetCE)
    have t\in set ( }u\rightsquigarrowT\mp@subsup{T}{}{\prime})\mathrm{ using T.left-tree-separates' st u by blast
    thus v}\in\mathrm{ 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 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 }\mp@subsup{->}{T}{}t\mathrm{ and v:v lelft-part st and w:w left-part t s and xs:v}\rightsquigarrowxs\rightsquigarrow
    shows set xs \capbag s \cap bag t\not={}
proof (rule ccontr)
    assume }\neg\mathrm{ ?thesis
    {
        fix u}\mathrm{ assume ú set xs
        with xs v<~?thesis` have vertex-subtree u\subseteqT.left-tree st
        proof (induct xs arbitrary:v)
        case (Cons x xs v)
        hence contra: v & bag s \veev\not\in bag t by (metis path-from-toE IntI empty-iff hd-in-set)
        {
            assume x =u \negvertex-subtree u\subseteqT.left-tree st
            then obtain z}\mathrm{ where z:z z| vertex-subtree u z& T.left-tree s t by blast
            hence z\in vertex-subtree v using Cons.prems(1,3) \langlex=u\rangle
                by (metis list.sel(1) path-from-to-def)
            hence v\inleft-part ts unfolding vertex-subtree-def
                    using T.left-tree-union-V z st by auto
            hence False using vertex-in-separator contra st Cons.prems(2) by blast
        }
        moreover {
            assume x\not=u
            hence u\in set xs using Cons.prems(4) by auto
            moreover hence xs \not= Nil using empty-iff list.set(1) by auto
            moreover hence last xs = w using Cons.prems(1) by auto
            moreover have path xs using Cons.prems(1) walk-tl by force
            moreover have hd xs \inleft-part st proof-
                have v->hd xs using Cons.prems(1,3) <xs \not= Nil` walk-first-edge' by auto
                    then obtain }\mp@subsup{u}{}{\prime}\mathrm{ where }\mp@subsup{u}{}{\prime}:\mp@subsup{u}{}{\prime}\in\mp@subsup{V}{T}{}v\in\mathrm{ bag }\mp@subsup{u}{}{\prime}hdxs\in\mathrm{ bag }\mp@subsup{u}{}{\prime
                        using bags-edges by blast
                    hence }\mp@subsup{u}{}{\prime}\inT.l\mathrm{ left-tree st
                        using contra vertex-in-separator st T.left-tree-union-V Cons.prems(2) by blast
                    thus ?thesis using u'(3) unfolding left-part-def by blast
            qed
            moreover have \negset xs \cap bag s\capbagt\not={} using Cons.prems(3)
                    IntI disjoint-iff-not-equal inf-le1 inf-le2 set-subset-Cons subsetCE by auto
            ultimately have vertex-subtree u\subseteqT.left-tree st using Cons.hyps by blast
        }
        ultimately show ?case by blast
        qed simp
    }
    hence vertex-subtree w\subseteq T.left-tree st using xs last-in-set by blast
    moreover have vertex-subtree w\capT.left-tree t s\not={} 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: $s \rightarrow_{T} t$ and $v w: v \rightarrow w$ and $v: v \in b a g s$ and $w: w \notin$ bag $s w \in$ left-part $t s$
shows $v \in b a g t$
proof-
have $v \rightsquigarrow[v, w] \rightsquigarrow w$ using $v v w w(1)$ by auto
hence set $[v, w] \cap$ bag $s \cap$ bag $t \neq\{ \}$ 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}\leq
proof-
    interpret Tree G using assms .
    {
        assume V\not={}
        then obtain root where root: root }\inV\mathrm{ 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: \bigwedgev.v\inbag v unfolding bag-def by simp
        have bag-in-V: \bigwedgev.v\inV\Longrightarrow bag v\subseteqV unfolding bag-def
            using parent-in-V empty-subsetI insert-subset by auto
        have TreeDecomposition G G bag proof
            show \bigcup{bag t | t.t\inV}=V using bag-in-V v-in-bag by blast
        next
            fix v w assume v}->
```

```
    moreover have }\bigwedge\mp@subsup{v}{}{\prime}\mp@subsup{w}{}{\prime}.\llbracket\mp@subsup{v}{}{\prime}->\mp@subsup{w}{}{\prime};\mp@subsup{v}{}{\prime}\not=root\rrbracket\Longrightarrow\mp@subsup{w}{}{\prime}\inbag v'\vee v'\inbag w' unfolding bag-de
        by (metis insertI2 parent-edge-cases parent-edge-root singletonI)
    ultimately have }v\inbag w\veew\inbag v using no-loops undirected by blas
    thus \existst\inV.v\inbag t ^ w\in bag t using <v->w\rangle edges-are-in-V v-in-bag by blast
    next
        fix s ut assume s:s\inV and u:u\inV and t:t\in set (s\rightsquigarrowu)
    have t\inV using t by (meson s subsetCE u unique-connecting-path-properties(1) walk-in-V)
    hence s=u\Longrightarrowt=s using left-tree-initial' s t by blast
    moreover have s->u\Longrightarrowt=s\veet=u using s tu<t\inV〉
        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\not=u\negs->u
        have s=root \Longrightarrowbag s\cap bag u={} unfolding bag-def
            using *(1,2) parent-edge u undirected by fastforce
        moreover have }u=\mathrm{ root Cbag s }\cap\mathrm{ bag u}={}\mathrm{ unfolding bag-def
        using *(1,2) parent-edge s by fastforce
        moreover have \llbrackets\not= root; u\not= root; parent s \not= parent u\rrbracket\Longrightarrow bag s \cap bag u={}
            unfolding bag-def using *(2) parent-edge s u undirected by fastforce
        moreover {
            assume **:s\not= root u\not= root parent s= parent ut\not=st\not=u
            have bag s\cap bag u = { parent s } unfolding bag-def using *(1)**(1-3)
                Int-insert-left inf.orderE insertE insert-absorb subset-insertI by auto
            moreover have t= parent s
                using sibling-path[OF s**(1) u**(2) *(1) **(3)] t **(4,5) by auto
            ultimately have bag s \cap bag u\subseteq bag t by (simp add: v-in-bag)
        }
        ultimately have bag s\cap bag u\subseteqbag t by blast
    }
        ultimately show bag s\capbag u\subseteqbag t by blast
    qed
    then interpret TreeDecomposition G G bag.
    {
        fix v
        have card {v, parent v}\leq2
            by (metis card.insert card.empty finite.emptyI finite-insert insert-absorb insert-not-empty
                lessI less-or-eq-imp-le numerals(2))
        hence card (bag v) \leq2 unfolding bag-def by simp
    }
    hence max-bag-card \leq 2 using <V F {}> max-bag-card-in-bag-cards by auto
    hence width \leq1 unfolding width-def by (simp add: <V = {}>)
    hence \existsbag.TreeDecomposition G G bag ^ TreeDecomposition.width G 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 :: nat Graph (structure)
    assumes Tree G card V}\mp@subsup{V}{G}{}>
    shows Graph.treewidth G}=
    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 \(\bigwedge v w . \llbracket v \in V ; w \in V ; v \neq w \rrbracket \Longrightarrow v \rightarrow w\)
    shows treewidth \(=\) card \(V-1\)
proof-
    \{
        assume \(V \neq\{ \}\)
        obtain \(T\) bag where
            T: TreeDecomposition \(G\) ( \(T\) :: nat Graph) bag treewidth \(=\) TreeDecomposition.width \(T\) bag
            using treewidth-cards-treewidth by blast
    interpret TreeDecomposition \(G T\) bag using \(T(1)\).
    assume \(\neg\) ?thesis
    hence width \(\neq\) card \(V-1\) by (simp add: \(T(2)\) )
```

Let $s$ be a bag of maximal size.

```
moreover obtain \(s\) where \(s: s \in V_{T}\) card (bag \(\left.s\right)=\) max-bag-card
using max-bag-card-in-bag-cards \(\langle V \neq\{ \}\rangle\) by fastforce
```

The treewidth cannot be larger than card $V-1$, so due to our assumption width $\neq$ card $V-1$ it must be smaller, hence card (bag $s)<\operatorname{card} V$.
ultimately have card (bag s) < 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 $v: v \in V v \notin$ 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 insert $v$ (bag s), which is a contradiction because such a bag would be too big.

```
obtain \(t\) where \(t: t \in V_{T} v \in\) bag \(t\) using bags-exist \(v(1)\) by blast
with \(s\) have \(\exists t \in V_{T}\). insert \(v(b a g s) \subseteq\) 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 \(v \in\) bag \(s\) thus ?thesis using \(t\) Cons.prems(1) by blast
```

```
next
    assume v}\not\inbag
    hence s\not=t using t(2) by blast
    hence xs \not=Nil using Cons.hyps(2) Cons.prems(1,3)
        by (metis T.unique-connecting-path-properties(3,4) last-ConsL list.sel(1))
    moreover have x=s using Cons.hyps(2) Cons.prems(1) t(1)
        by (metis T.unique-connecting-path-properties(3) list.sel(1))
    ultimately obtain s' xs' where s':s# s'# x\mp@subsup{s}{}{\prime}=s\rightsquigarrow}\mp@subsup{s}{}{\prime}\mp@subsup{}{}{t
        using Cons.hyps(2) list.exhaust by metis
    moreover have st-path:T.path (s\rightsquigarrowTt)
        by (simp add: Cons.prems(1) T.unique-connecting-path-properties(1) t(1))
    ultimately have s'\in V T
```

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

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

Bags can never gain vertices because we started with a bag of maximal size.
moreover have card (bag s') $\leq$ card (bag s) proof-
have card (bag s') $\leq$ max-bag-card unfolding max-bag-card-def
using Max-ge $\left\langle s^{\prime} \in V_{T}\right\rangle$ bag-cards-finite by blast
thus ?thesis using Cons.prems(2) by auto
qed
ultimately have bag $s^{\prime}=b a g s$ using $\left\langle s^{\prime} \in V_{T}\right\rangle$ bag-finite card-seteq by blast
thus ?thesis
using Cons.hyps Cons.prems (1,2) $\left\langle s^{\prime} \in V_{T^{\prime}} t s^{\prime}\right.$ st-path $\langle x s \neq[]\rangle$
by (metis T.path-from-toI T.path-tl T.unique-connecting-path-properties(4)
T.unique-connecting-path-unique last.simps list.sel( $(1,3))$
qed
qed
hence $\exists t \in V_{T}$. card (bag s) < card (bag t) using $v($ (2)
by (metis bag-finite card-seteq insert-subset not-le)
hence False using s Max.coboundedI bag-cards-finite not-le unfolding max-bag-card-def by
auto
\}
thus ?thesis using treewidth-upper-bound-V card.empty diff-diff-cancel 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 $=u 0|v 0| w 0$
The empty graph is a tree．
definition $T 1 \equiv 0$ 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 \(T \mathcal{Z} \equiv 0\) verts \(=\{u 0, v 0\}, \operatorname{arcs}=\{(u 0, v 0),(v 0, u 0)\})\)
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 \(v \in V_{T 2} w \in V_{T 2}\) 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 \(\operatorname{simps}(1,2))\)
    next
        fix \(x s::\) Vertices list
        \{
            fix \(x y\)
            assume cycle \(x s\) and \(x y:(x=v 0 \wedge y=u 0) \vee(x=u 0 \wedge y=v 0)\) and \(h d x s=x\)
            hence last \(x s=y\)
                by (metis T2-def cycleE distinct.simps(2) distinct-singleton insert-iff list.set(1)
                    prod.inject \(\operatorname{simps}(2))\)
            moreover have \(\bigwedge v . v \in\) set \(x s \Longrightarrow v=x \vee v=y\) using 〈cycle \(x s\) 〉 \(x y\)
                by (metis cycle-def walk-in-V T2-def empty-iff insertE insert-absorb insert-subset
                select-convs(1))
            ultimately have \(x s=[x, y]\) using «cycle \(x s\rangle x y\)
                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 \(\neg\) 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 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: $\llbracket v \in V_{T 2} ; w \in V_{T 2} ; v \neq w \rrbracket \Longrightarrow v \rightarrow_{T 2} w$ 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 $T 3 \equiv 0$ 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 unfolding T3-def by standard auto
$[u 0, v 0, w 0]$ is a cycle in $T 3$, so $T 3$ is not a tree.
lemma Not-Tree-T3: $\neg$ Tree T3 proof
assume Tree T3 then interpret Tree T3 .
let $? x s=[u 0, v 0, w 0]$
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) $\in$ 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: $\llbracket v \in V_{T 3} ; w \in V_{T 3} ; v \neq w \rrbracket \Longrightarrow v \rightarrow_{\text {T3 }} w$ 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.

