Verified Enumeration of Trees

Nils Cremer

March 17, 2025

Abstract

This thesis presents the verification of enumeration algorithms for trees. The first algorithm is based on the well known Prüfer-correspondence and allows the enumeration of all possible labeled trees over a fixed finite set of vertices. The second algorithm enumerates rooted, unlabeled trees of a specified size up to graph isomorphisms. It allows for the efficient enumeration without the use of an intermediate encoding of the trees with level sequences, unlike the algorithm by Beyer and Hedetniemi [1] it is based on. Both algorithms are formalized and verified in Isabelle/HOL. The formalization of trees and other graph theoretic results is also presented.

Contents

1 Graphs and Trees **2**
1.1 Miscellaneous 2
1.2 Degree 4
1.3 Walks 7
1.4 Paths 7
1.5 Cycles 8
1.6 Subgraphs 9
1.7 Connectivity 10
1.8 Connected components 15
1.9 Trees 21
1.10 Graph Isomorphism 28

2 Enumeration of Labeled Trees **30**
2.1 Algorithm 30
2.2 Correctness 30
2.3 Totality 33
2.4 Distinction 36

3 Rooted Trees **39**
3.1 Rooted Graphs 46
3.2 Rooted Graph Isomorphism 47
3.3 Conversion between unlabeled, ordered, rooted trees and tree graphs 48
3.4 Injectivity with respect to isomorphism 65

4 Enumeration of Rooted Trees **71**
4.1 Enumeration is monotonically decreasing 73
4.2 Size preservation 73
4.3 Setup for termination proof 74
4.4 Algorithms for enumeration 75
4.5 Regularity 76
4.6 Totality 78
4.7 Distinctness 87

1 Graphs and Trees

theory *Tree-Graph*
imports *Undirected-Graph-Theory.Undirected-Graphs-Root*
begin

1.1 Miscellaneous

definition (**in** *ulgraph*) *loops* :: *'a edge set* **where**
loops = {*e*∈*E*. *is-loop e*}

definition (**in** *ulgraph*) *sedges* :: *'a edge set* **where**
sedges = {*e*∈*E*. *is-sedge e*}

lemma (**in** *ulgraph*) *union-loops-sedges*: *loops* ∪ *sedges* = *E*
unfolding *loops-def sedges-def is-loop-def is-sedge-def* **using** *alt-edge-size* **by** *blast*

lemma (**in** *ulgraph*) *disjnt-loops-sedges*: *disjnt loops sedges*
unfolding *disjnt-def loops-def sedges-def is-loop-def is-sedge-def* **by** *auto*

lemma (**in** *fin-ulgraph*) *finite-loops*: *finite loops*
unfolding *loops-def* **using** *fin-edges* **by** *auto*

lemma (**in** *fin-ulgraph*) *finite-sedges*: *finite sedges*
unfolding *sedges-def* **using** *fin-edges* **by** *auto*

lemma (**in** *ulgraph*) *edge-incident-vert*: *e* ∈ *E* ⟹ ∃ *v*∈*V*. *vincident v e*
using *edge-size wellformed* **by** (*metis empty-not-edge equals0I vincident-def incident-edge-in-wf*)

lemma (**in** *ulgraph*) *Union-incident-edges*: ($\bigcup v \in V$. *incident-edges v*) = *E*
unfolding *incident-edges-def* **using** *edge-incident-vert* **by** *auto*

lemma (**in** *ulgraph*) *induced-edges-mono*: $V_1 \subseteq V_2 \Longrightarrow$ *induced-edges* $V_1 \subseteq$ *induced-edges* V_2
using *induced-edges-def* **by** *auto*

definition (**in** *graph-system*) *remove-vertex* :: *'a* ⇒ *'a pregraph* **where**
remove-vertex v = ($V - \{v\}$, $\{e \in E.$ ¬ *vincident v e*$\}$)

lemma (**in** *ulgraph*) *ex-neighbor-degree-not-0*:
assumes *degree-non-0*: *degree v* ≠ *0*
shows ∃ *u*∈*V*. *vert-adj v u*
proof−
have ∃ *e*∈*E*. *v* ∈ *e* **using** *degree-non-0 elem-exists-non-empty-set*
unfolding *degree-def incident-sedges-def incident-loops-def vincident-def* **by** *auto*
then show *?thesis*
by (*metis degree-non-0 in-mono is-isolated-vertex-def is-isolated-vertex-degree0 vert-adj-sym wellformed*)
qed

lemma (**in** *ulgraph*) *ex1-neighbor-degree-1*:
assumes *degree-1*: *degree v* = *1*
shows ∃!*u*. *vert-adj v u*
proof−
have *card* (*incident-loops v*) = *0* **using** *degree-1* **unfolding** *degree-def* **by** *auto*
then have *incident-loops*: *incident-loops v* = {} **by** (*simp add*: *finite-incident-loops*)
then have *card-incident-sedges*: *card* (*incident-sedges v*) = *1* **using** *degree-1* **unfolding** *degree-def* **by** *simp*
obtain *u* **where** *vert-adj*: *vert-adj v u* **using** *degree-1 ex-neighbor-degree-not-0* **by** *force*
then have *u* ≠ *v* **using** *incident-loops* **unfolding** *incident-loops-def vert-adj-def* **by** *blast*
then have *u-incident*: {*v*,*u*} ∈ *incident-sedges v* **using** *vert-adj* **unfolding** *incident-sedges-def vert-adj-def vincident-def* **by** *simp*
then have *incident-sedges*: *incident-sedges v* = {{*v*,*u*}} **using** *card-incident-sedges*
by (*simp add*: *comp-sgraph.card1-incident-imp-vert comp-sgraph.vincident-def*)
have *vert-adj v u'* ⟹ *u'* = *u* **for** *u'*
proof−
assume *v-u'-adj*: *vert-adj v u'*
then have *u'* ≠ *v* **using** *incident-loops* **unfolding** *incident-loops-def vert-adj-def* **by** *blast*
then have {*v*,*u'*} ∈ *incident-sedges v* **using** *v-u'-adj* **unfolding** *incident-sedges-def vert-adj-def vincident-def* **by** *simp*
then show *u'* = *u* **using** *incident-sedges* **by** *force*
qed
then show *?thesis* **using** *vert-adj* **by** *blast*

qed

lemma (**in** *ulgraph*) *degree-1-edge-partition*:
assumes *degree-1*: *degree v* = *1*
shows $E = \{\{THE\ u.\ vert\text{-}adj\ v\ u,\ v\}\} \cup \{e \in E.\ v \notin e\}$
proof−
have *card* (*incident-loops v*) = *0* **using** *degree-1* **unfolding** *degree-def* **by** *auto*
then have *incident-loops*: *incident-loops v* = {} **by** (*simp add*: *finite-incident-loops*)
then have *card* (*incident-sedges v*) = *1* **using** *degree-1* **unfolding** *degree-def* **by** *simp*
then have *card-incident-edges*: *card* (*incident-edges v*) = *1* **using** *incident-loops incident-edges-union* **by** *simp*
obtain *u* **where** *vert-adj*: *vert-adj v u* **using** *ex1-neighbor-degree-1 degree-1* **by** *blast*
then have $\{v,\ u\} \in \{e \in E.\ v \in e\}$ **unfolding** *vert-adj-def* **by** *blast*
then have *edges-incident-v*: $\{e \in E.\ v \in e\} = \{\{v,\ u\}\}$ **using** *card-incident-edges card-1-singletonE singletonD*
unfolding *incident-edges-def vincident-def* **by** *metis*
have *u*: *u* = (*THE u. vert-adj v u*) **using** *vert-adj ex1-neighbor-degree-1 degree-1*
by (*simp add*: *the1-equality*)
show *?thesis* **using** *edges-incident-v u* **by** *blast*
qed

lemma (**in** *sgraph*) *vert-adj-not-eq*: *vert-adj u v* ⟹ $u \neq v$
unfolding *vert-adj-def* **using** *edge-vertices-not-equal* **by** *blast*

1.2 Degree

lemma (**in** *ulgraph*) *empty-E-degree-0*: *E* = {} ⟹ *degree v* = *0*
using *incident-edges-empty degree0-inc-edges-empt-iff* **unfolding** *incident-edges-def* **by** *simp*

lemma (**in** *fin-ulgraph*) *handshaking*: $(\sum v{\in}V.\ degree\ v) = 2 * card\ E$
using *fin-edges fin-ulgraph-axioms*
proof (*induction E*)
case *empty*
then interpret *g*: *fin-ulgraph V* {} **.**
show *?case* **using** *g.empty-E-degree-0* **by** *simp*
next
case (*insert e E′*)
then interpret *g′*: *fin-ulgraph V insert e E′* **by** *blast*
interpret *g*: *fin-ulgraph V E′* **using** *g′.wellformed g′.edge-size finV* **by** (*unfold-locales*, *auto*)
show *?case*
proof (*cases is-loop e*)
case *True*
then obtain *u* **where** *e*: *e* = {*u*} **using** *card-1-singletonE is-loop-def* **by** *blast*
then have *inc-sedges*: ⋀*v. g′.incident-sedges v* = *g.incident-sedges v* **unfolding** *g′.incident-sedges-def g.incident-sedges-def* **by** *auto*

have $\bigwedge v.\ v \neq u \Longrightarrow g'.incident\text{-}loops\ v = g.incident\text{-}loops\ v$ **unfolding** $g'.incident\text{-}loops\text{-}def$ $g.incident\text{-}loops\text{-}def$ **using** e **by** *auto*
then have *degree-not-u*: $\bigwedge v.\ v \neq u \Longrightarrow g'.degree\ v = g.degree\ v$ **using** *inc-sedges* **unfolding** $g'.degree\text{-}def$ $g.degree\text{-}def$ **by** *auto*
have $g'.incident\text{-}loops\ u = g.incident\text{-}loops\ u \cup \{e\}$ **unfolding** $g'.incident\text{-}loops\text{-}def$ $g.incident\text{-}loops\text{-}def$ **using** e **by** *auto*
then have *degree-u*: $g'.degree\ u = g.degree\ u + 2$ **using** *inc-sedges insert(2)* *g.finite-incident-loops g.incident-loops-def* **unfolding** $g'.degree\text{-}def$ $g.degree\text{-}def$ **by** *auto*
have $u \in V$ **using** e $g'.wellformed$ **by** *blast*
then have $(\sum v{\in}V.\ g'.degree\ v) = g'.degree\ u + (\sum v{\in}V{-}\{u\}.\ g'.degree\ v)$
by (*simp add*: *finV sum.remove*)
also have $\ldots = (\sum v{\in}V.\ g.degree\ v) + 2$ **using** *degree-not-u degree-u sum.remove*[*OF finV* ‹$u{\in}V$›, *of g.degree*] **by** *auto*
also have $\ldots = 2 * card\ (insert\ e\ E')$ **using** *insert g.fin-ulgraph-axioms* **by** *auto*
finally show *?thesis* **.**
next
case *False*
obtain u w **where** e: $e = \{u,w\}$ **using** $g'.obtain\text{-}edge\text{-}pair\text{-}adj$ **by** *fastforce*
then have *card-e*: $card\ e = 2$ **using** *False* $g'.alt\text{-}edge\text{-}size$ *is-loop-def* **by** *auto*
then have $u \neq w$ **using** *card-2-iff* **using** e **by** *fastforce*
have *inc-loops*: $\bigwedge v.\ g'.incident\text{-}loops\ v = g.incident\text{-}loops\ v$
unfolding $g'.incident\text{-}loops\text{-}alt$ $g.incident\text{-}loops\text{-}alt$ **using** *False is-loop-def* **by** *auto*
have $\bigwedge v.\ v \neq u \Longrightarrow v \neq w \Longrightarrow g'.incident\text{-}sedges\ v = g.incident\text{-}sedges\ v$
unfolding $g'.incident\text{-}sedges\text{-}def$ $g.incident\text{-}sedges\text{-}def$ $g.vincident\text{-}def$ **using** e **by** *auto*
then have *degree-not-u-w*: $\bigwedge v.\ v \neq u \Longrightarrow v \neq w \Longrightarrow g'.degree\ v = g.degree\ v$
unfolding $g'.degree\text{-}def$ $g.degree\text{-}def$ **using** *inc-loops* **by** *auto*
have $g'.incident\text{-}sedges\ u = g.incident\text{-}sedges\ u \cup \{e\}$
unfolding $g'.incident\text{-}sedges\text{-}def$ $g.incident\text{-}sedges\text{-}def$ $g.vincident\text{-}def$ **using** e *card-e* **by** *auto*
then have *degree-u*: $g'.degree\ u = g.degree\ u + 1$
using *inc-loops insert(2) g.fin-edges g.finite-inc-sedges g.incident-sedges-def*
unfolding $g'.degree\text{-}def$ $g.degree\text{-}def$ **by** *auto*
have $g'.incident\text{-}sedges\ w = g.incident\text{-}sedges\ w \cup \{e\}$
unfolding $g'.incident\text{-}sedges\text{-}def$ $g.incident\text{-}sedges\text{-}def$ $g.vincident\text{-}def$ **using** e *card-e* **by** *auto*
then have *degree-w*: $g'.degree\ w = g.degree\ w + 1$
using *inc-loops insert(2) g.fin-edges g.finite-inc-sedges g.incident-sedges-def*
unfolding $g'.degree\text{-}def$ $g.degree\text{-}def$ **by** *auto*
have *inV*: $u \in V$ $w \in V{-}\{u\}$ **using** e $g'.wellformed$ ‹$u{\neq}w$› **by** *auto*
then have $(\sum v{\in}V.\ g'.degree\ v) = g'.degree\ u + g'.degree\ w + (\sum v{\in}V{-}\{u\}{-}\{w\}.\ g'.degree\ v)$
using *sum.remove finV* **by** (*metis add.assoc finite-Diff*)
also have $\ldots = g.degree\ u + g.degree\ w + (\sum v{\in}V{-}\{u\}{-}\{w\}.\ g.degree\ v) + 2$
using *degree-not-u-w degree-u degree-w* **by** *simp*

also have ... = ($\sum v \in V.$ *g.degree v*) + *2* **using** *sum.remove finV inV* **by** (*metis add.assoc finite-Diff*)
also have ... = *2* ∗ *card* (*insert e E′*) **using** *insert g.fin-ulgraph-axioms* **by** *auto*
finally show *?thesis* **.**
qed
qed

lemma (**in** *fin-ulgraph*) *degree-remove-adj-ne-vert*:
assumes $u \neq v$
and *vert-adj*: *vert-adj u v*
and *remove-vertex*: *remove-vertex u* = (*V′*,*E′*)
shows *ulgraph.degree E′ v* = *degree v* − *1*
proof−
interpret *G′*: *fin-ulgraph V′ E′* **using** *remove-vertex wellformed edge-size finV* **unfolding** *remove-vertex-def vincident-def*
by (*unfold-locales*, *auto*)
have *E′*: *E′* = {*e* ∈ *E*. *u* ∉ *e*} **using** *remove-vertex* **unfolding** *remove-vertex-def vincident-def* **by** *simp*
have *incident-loops′*: *G′.incident-loops v* = *incident-loops v* **unfolding** *incident-loops-def*
using ‹*u*≠*v*› *E′ G′.incident-loops-def* **by** *auto*
have *uv-incident*: {*u*,*v*} ∈ *incident-sedges v* **using** *vert-adj* ‹*u*≠*v*› **unfolding** *vert-adj-def incident-sedges-def vincident-def* **by** *simp*
have *uv-incident′*: {*u*, *v*} ∉ *G′.incident-sedges v* **unfolding** *G′.incident-sedges-def vincident-def* **using** *E′* **by** *blast*
have *e* ∈ *E* ⟹ *u* ∈ *e* ⟹ *v* ∈ *e* ⟹ *card e* = *2* ⟹ *e* = {*u*,*v*} **for** *e*
using ‹*u*≠*v*› *obtain-edge-pair-adj* **by** *blast*
then have {*e* ∈ *E*. *u* ∈ *e* ∧ *v* ∈ *e* ∧ *card e* = *2*} = {{*u*,*v*}} **using** *uv-incident* **unfolding** *incident-sedges-def* **by** *blast*
then have *incident-sedges v* = *G′.incident-sedges v* ∪ {{*u*,*v*}} **unfolding** *G′.incident-sedges-def incident-sedges-def vincident-def* **using** *E′* **by** *blast*
then show *?thesis* **unfolding** *G′.degree-def degree-def* **using** *incident-loops′ uv-incident′ G′.finite-inc-sedges G′.fin-edges* **by** *auto*
qed

lemma (**in** *ulgraph*) *degree-remove-non-adj-vert*:
assumes $u \neq v$
and *vert-non-adj*: ¬ *vert-adj u v*
and *remove-vertex*: *remove-vertex u* = (*V′*, *E′*)
shows *ulgraph.degree E′ v* = *degree v*
proof−
interpret *G′*: *ulgraph V′ E′* **using** *remove-vertex wellformed edge-size* **unfolding** *remove-vertex-def vincident-def*
by (*unfold-locales*, *auto*)
have *E′*: *E′* = {*e* ∈ *E*. *u* ∉ *e*} **using** *remove-vertex* **unfolding** *remove-vertex-def vincident-def* **by** *simp*
have *incident-loops′*: *G′.incident-loops v* = *incident-loops v* **unfolding** *incident-loops-def*

  using ‹$u{\neq}v$› E' G'*.incident-loops-def* **by** *auto*
have G'*.incident-sedges v* = *incident-sedges v* **unfolding** G'*.incident-sedges-def incident-sedges-def vincident-def*
  using E' ‹$u{\neq}v$› *vincident-def vert-adj-edge-iff2 vert-non-adj* **by** *auto*
then show *?thesis* **using** *incident-loops'* **unfolding** G'*.degree-def degree-def* **by** *simp*
qed

1.3 Walks

lemma (**in** *ulgraph*) *walk-edges-induced-edges*: *is-walk p* $\Longrightarrow$ *set* (*walk-edges p*) $\subseteq$ *induced-edges* (*set p*)
unfolding *induced-edges-def is-walk-def* **by** (*induction p rule*: *walk-edges.induct*) *auto*

lemma (**in** *ulgraph*) *walk-edges-in-verts*: $e \in$ *set* (*walk-edges xs*) $\Longrightarrow$ $e \subseteq$ *set xs*
by (*induction xs rule*: *walk-edges.induct*) *auto*

lemma (**in** *ulgraph*) *is-walk-prefix*: *is-walk* (*xs*@*ys*) $\Longrightarrow$ *xs* $\neq$ [] $\Longrightarrow$ *is-walk xs*
unfolding *is-walk-def* **using** *walk-edges-append-ss2* **by** *fastforce*

lemma (**in** *ulgraph*) *split-walk-edge*: $\{x,y\} \in$ *set* (*walk-edges p*) $\Longrightarrow$
$\exists$ *xs ys*. *p* = *xs* @ *x* # *y* # *ys* $\vee$ *p* = *xs* @ *y* # *x* # *ys*
by (*induction p rule*: *walk-edges.induct*) (*auto*, *metis append-Nil doubleton-eq-iff*, (*metis append-Cons*)+)

1.4 Paths

lemma (**in** *ulgraph*) *is-gen-path-wf*: *is-gen-path p* $\Longrightarrow$ *set p* $\subseteq$ *V*
unfolding *is-gen-path-def* **using** *is-walk-wf* **by** *auto*

lemma (**in** *ulgraph*) *path-wf*: *is-path p* $\Longrightarrow$ *set p* $\subseteq$ *V*
by (*simp add*: *is-path-walk is-walk-wf*)

lemma (**in** *fin-ulgraph*) *length-gen-path-card-V*: *is-gen-path p* $\Longrightarrow$ *walk-length p* $\leq$ *card V*
by (*metis card-mono distinct-card distinct-tl finV is-gen-path-def is-walk-def length-tl list.exhaust-sel order-trans set-subset-Cons walk-length-conv*)

lemma (**in** *fin-ulgraph*) *length-path-card-V*: *is-path p* $\Longrightarrow$ *length p* $\leq$ *card V*
by (*metis path-wf card-mono distinct-card finV is-path-def*)

lemma (**in** *ulgraph*) *is-gen-path-prefix*: *is-gen-path* (*xs*@*ys*) $\Longrightarrow$ *xs* $\neq$ [] $\Longrightarrow$ *is-gen-path* (*xs*)
unfolding *is-gen-path-def* **using** *is-walk-prefix*
by (*auto*, *metis Int-iff distinct.simps*(*2*) *emptyE last-appendL last-appendR last-in-set list.collapse*)

lemma (**in** *ulgraph*) *connecting-path-append*: *connecting-path u w* (*xs*@*ys*) $\Longrightarrow$ *xs* $\neq$ [] $\Longrightarrow$ *connecting-path u* (*last xs*) *xs*

unfolding *connecting-path-def* **using** *is-gen-path-prefix* **by** *auto*

lemma (**in** *ulgraph*) *connecting-path-tl*: *connecting-path u v* (*u* # *w* # *xs*) $\Longrightarrow$ *connecting-path w v* (*w* # *xs*)
unfolding *connecting-path-def is-gen-path-def* **using** *is-walk-drop-hd distinct-tl* **by** *auto*

lemma (**in** *fin-ulgraph*) *obtain-longest-path*:
assumes $e \in E$
and *sedge*: *is-sedge e*
obtains *p* **where** *is-path p* $\forall$ *s. is-path s* $\longrightarrow$ *length s* $\leq$ *length p*
proof−
let *?longest-path* = *ARG-MAX length p. is-path p*
obtain *u v* **where** *e*: $u \neq v$ $e = \{u,v\}$ **using** *sedge card-2-iff* **unfolding** *is-sedge-def* **by** *metis*
then have *inV*: $u \in V$ $v \in V$ **using** ‹e∈E› *wellformed* **by** *auto*
then have *path-ex*: *is-path* [*u*,*v*] **using** *e* ‹e∈E› **unfolding** *is-path-def is-open-walk-def is-walk-def* **by** *simp*
obtain *p* **where** *p-is-path*: *is-path p* **and** *p-longest-path*: $\forall$ *s. is-path s* $\longrightarrow$ *length s* $\leq$ *length p*
using *path-ex length-path-card-V ex-has-greatest-nat*[*of is-path* [*u*,*v*] *length gorder*] **by** *force*
then show *?thesis* **..**
qed

1.5 Cycles

context *ulgraph*
begin

definition *is-cycle2* :: *'a list* $\Rightarrow$ *bool* **where**
is-cycle2 xs $\longleftrightarrow$ *is-cycle xs* $\wedge$ *distinct* (*walk-edges xs*)

lemma *loop-is-cycle2*: $\{v\} \in E \Longrightarrow$ *is-cycle2* [*v*, *v*]
unfolding *is-cycle2-def is-cycle-alt is-walk-def* **using** *wellformed walk-length-conv* **by** *auto*

end

lemma (**in** *sgraph*) *cycle2-min-length*:
assumes *cycle*: *is-cycle2 c*
shows *walk-length c* $\geq$ *3*
proof−
consider *c* = [] | $\exists$ *v1*. *c* = [*v1*] | $\exists$ *v1 v2*. *c* = [*v1*, *v2*] | $\exists$ *v1 v2 v3*. *c* = [*v1*, *v2*, *v3*] | $\exists$ *v1 v2 v3 v4 vs*. *c* = *v1*#*v2*#*v3*#*v4*#*vs*
by (*metis list.exhaust-sel*)
then show *?thesis* **using** *cycle walk-length-conv singleton-not-edge* **unfolding** *is-cycle2-def is-cycle-alt is-walk-def* **by** (*cases*, *auto*)
qed

lemma (**in** *fin-ulgraph*) *length-cycle-card-V*: *is-cycle c* $\Longrightarrow$ *walk-length c* $\leq$ *Suc* (*card V*)
using *length-gen-path-card-V* **unfolding** *is-gen-path-def is-cycle-alt* **by** *fastforce*

lemma (**in** *ulgraph*) *is-cycle-connecting-path*: *is-cycle* (*u#v#xs*) $\Longrightarrow$ *connecting-path v u* (*v#xs*)
unfolding *is-cycle-def connecting-path-def is-closed-walk-def is-gen-path-def* **using** *is-walk-drop-hd* **by** *auto*

lemma (**in** *ulgraph*) *cycle-edges-notin-tl*: *is-cycle2* (*u#v#xs*) $\Longrightarrow$ {*u*,*v*} $\notin$ *set* (*walk-edges* (*v#xs*))
unfolding *is-cycle2-def* **by** *simp*

1.6 Subgraphs

locale *ulsubgraph* = *subgraph* V_H E_H V_G E_G +
G: *ulgraph* V_G E_G **for** V_H E_H V_G E_G
begin

interpretation *H*: *ulgraph* V_H E_H
using *is-subgraph-ulgraph G.ulgraph-axioms* **by** *auto*

lemma *is-walk*: *H.is-walk xs* $\Longrightarrow$ *G.is-walk xs*
unfolding *H.is-walk-def G.is-walk-def* **using** *verts-ss edges-ss* **by** *blast*

lemma *is-closed-walk*: *H.is-closed-walk xs* $\Longrightarrow$ *G.is-closed-walk xs*
unfolding *H.is-closed-walk-def G.is-closed-walk-def* **using** *is-walk* **by** *blast*

lemma *is-gen-path*: *H.is-gen-path p* $\Longrightarrow$ *G.is-gen-path p*
unfolding *H.is-gen-path-def G.is-gen-path-def* **using** *is-walk* **by** *blast*

lemma *connecting-path*: *H.connecting-path u v p* $\Longrightarrow$ *G.connecting-path u v p*
unfolding *H.connecting-path-def G.connecting-path-def* **using** *is-gen-path* **by** *blast*

lemma *is-cycle*: *H.is-cycle c* $\Longrightarrow$ *G.is-cycle c*
unfolding *H.is-cycle-def G.is-cycle-def* **using** *is-closed-walk* **by** *blast*

lemma *is-cycle2*: *H.is-cycle2 c* $\Longrightarrow$ *G.is-cycle2 c*
unfolding *H.is-cycle2-def G.is-cycle2-def* **using** *is-cycle* **by** *blast*

lemma *vert-connected*: *H.vert-connected u v* $\Longrightarrow$ *G.vert-connected u v*
unfolding *H.vert-connected-def G.vert-connected-def* **using** *connecting-path* **by** *blast*

lemma *is-connected-set*: *H.is-connected-set V'* $\Longrightarrow$ *G.is-connected-set V'*
unfolding *H.is-connected-set-def G.is-connected-set-def* **using** *vert-connected* **by** *blast*

end

lemma (**in** *graph-system*) *subgraph-remove-vertex*: *remove-vertex* $v = (V', E') \Longrightarrow$ *subgraph* V' E' V E
using *wellformed* **unfolding** *remove-vertex-def vincident-def* **by** (*unfold-locales*, *auto*)

1.7 Connectivity

lemma (**in** *ulgraph*) *connecting-path-connected-set*:
assumes *conn-path*: *connecting-path u v p*
shows *is-connected-set* (*set p*)
proof −
have ∀ *w*∈*set p*. *vert-connected u w*
proof
fix *w* **assume** *w* ∈ *set p*
then obtain *xs ys* **where** *p* = *xs*@[*w*]@*ys* **using** *split-list* **by** *fastforce*
then have *connecting-path u w* (*xs*@[*w*]) **using** *conn-path* **unfolding** *connecting-path-def* **using** *is-gen-path-prefix* **by** (*auto simp*: *hd-append*)
then show *vert-connected u w* **unfolding** *vert-connected-def* **by** *blast*
qed
then show *?thesis* **using** *vert-connected-rev vert-connected-trans* **unfolding** *is-connected-set-def* **by** *blast*
qed

lemma (**in** *ulgraph*) *vert-connected-neighbors*:
assumes {*v*,*u*} ∈ *E*
shows *vert-connected v u*
proof −
have *connecting-path v u* [*v*,*u*] **unfolding** *connecting-path-def is-gen-path-def is-walk-def* **using** *assms wellformed* **by** *auto*
then show *?thesis* **unfolding** *vert-connected-def* **by** *auto*
qed

lemma (**in** *ulgraph*) *connected-empty-E*:
assumes *empty*: *E* = {}
and *connected*: *vert-connected u v*
shows $u = v$
proof (*rule ccontr*)
assume $u \neq v$
then obtain *p* **where** *conn-path*: *connecting-path u v p* **using** *connected* **unfolding** *vert-connected-def* **by** *blast*
then obtain *e* **where** *e* ∈ *set* (*walk-edges p*) **using** ‹*u*≠*v*› *connecting-path-length-bound* **unfolding** *walk-length-def* **by** *fastforce*
then have *e* ∈ *E* **using** *conn-path* **unfolding** *connecting-path-def is-gen-path-def is-walk-def* **by** *blast*
then show *False* **using** *empty* **by** *blast*
qed

lemma (**in** *fin-ulgraph*) *degree-0-not-connected*:
assumes *degree-0*: *degree v* = *0*
and *u* ≠ *v*
shows ¬ *vert-connected v u*
proof
assume *connected*: *vert-connected v u*
then obtain *p* **where** *conn-path*: *connecting-path v u p* **unfolding** *vert-connected-def* **by** *blast*
then have *walk-length p* ≥ *1* **using** ‹*u*≠*v*› *connecting-path-length-bound* **by** *metis*
then have *length p* ≥ *2* **using** *walk-length-conv* **by** *simp*
then obtain *w p'* **where** *p* = *v*#*w*#*p'* **using** *walk-length-conv conn-path* **unfolding** *connecting-path-def*
by (*metis assms*(*2*) *is-gen-path-def is-walk-not-empty2 last-ConsL list.collapse*)
then have *inE*: {*v*,*w*} ∈ *E* **using** *conn-path* **unfolding** *connecting-path-def is-gen-path-def is-walk-def* **by** *simp*
then have {*v*,*w*} ∈ *incident-edges v* **unfolding** *incident-edges-def vincident-def* **by** *simp*
then show *False* **using** *degree0-inc-edges-empt-iff fin-edges degree-0* **by** *blast*
qed

lemma (**in** *fin-connected-ulgraph*) *degree-not-0*:
assumes *card V* ≥ *2*
and *inV*: *v* ∈ *V*
shows *degree v* ≠ *0*
proof−
obtain *u* **where** *u* ∈ *V* **and** *u* ≠ *v* **using** *assms*
by (*metis card-eq-0-iff card-le-Suc0-iff-eq less-eq-Suc-le nat-less-le not-less-eq-eq numeral-2-eq-2*)
then show *?thesis* **using** *degree-0-not-connected inV vertices-connected* **by** *blast*
qed

lemma (**in** *connected-ulgraph*) *V-E-empty*: *E* = {} ⟹ ∃ *v*. *V* = {*v*}
using *connected-empty-E connected not-empty* **unfolding** *is-connected-set-def*
by (*metis ex-in-conv insert-iff mk-disjoint-insert*)

lemma (**in** *connected-ulgraph*) *vert-connected-remove-edge*:
assumes *e*: {*u*,*v*} ∈ *E*
shows ∀ *w*∈*V*. *ulgraph.vert-connected V* (*E* − {{*u*,*v*}}) *w u* ∨ *ulgraph.vert-connected V* (*E* − {{*u*,*v*}}) *w v*
proof
fix *w* **assume** *w*∈*V*
interpret *g'*: *ulgraph V E* − {{*u*,*v*}} **using** *wellformed edge-size* **by** (*unfold-locales*, *auto*)
have *inV*: *u* ∈ *V v* ∈ *V* **using** *e wellformed* **by** *auto*
obtain *p* **where** *conn-path*: *connecting-path w v p* **using** *connected inV* ‹*w*∈*V*› **unfolding** *is-connected-set-def vert-connected-def* **by** *blast*
then show *g'.vert-connected w u* ∨ *g'.vert-connected w v*
proof (*cases* {*u*,*v*} ∈ *set* (*walk-edges p*))

case *True*
assume *walk-edge*: $\{u,v\} \in$ *set* (*walk-edges p*)
then show *?thesis*
proof (*cases* $w = v$)
case *True*
then show *?thesis* **using** *inV g'.vert-connected-id* **by** *blast*
next
case *False*
then have *distinct*: *distinct p* **using** *conn-path* **by** (*simp add*: *connecting-path-def is-gen-path-distinct*)
have $u \in$ *set p* **using** *walk-edge walk-edges-in-verts* **by** *blast*
obtain *xs ys* **where** *p-split*: $p = xs @ u \# v \# ys \vee p = xs @ v \# u \# ys$ **using** *split-walk-edge*[*OF walk-edge*] **by** *blast*
have *v-notin-ys*: $v \notin$ *set ys* **using** *distinct p-split* **by** *auto*
have *last* $p = v$ **using** *conn-path* **unfolding** *connecting-path-def* **by** *simp*
then have *p*: $p = (xs@[u]) @ [v]$ **using** *v-notin-ys p-split last-in-set last-appendR*
by (*metis append.assoc append-Cons last.simps list.discI self-append-conv2*)
then have *conn-path-u*: *connecting-path w u* (*xs*@[*u*]) **using** *connecting-path-append conn-path* **by** *fastforce*
have $v \notin$ *set* (*xs*@[*u*]) **using** *p distinct* **by** *auto*
then have $\{u,v\} \notin$ *set* (*walk-edges* (*xs*@[*u*])) **using** *walk-edges-in-verts* **by** *blast*
then have *g'.connecting-path w u* (*xs*@[*u*]) **using** *conn-path-u*
unfolding *g'.connecting-path-def connecting-path-def g'.is-gen-path-def is-gen-path-def g'.is-walk-def is-walk-def* **by** *blast*
then show *?thesis* **unfolding** *g'.vert-connected-def* **by** *blast*
qed
next
case *False*
then have *g'.connecting-path w v p* **using** *conn-path*
unfolding *g'.connecting-path-def connecting-path-def g'.is-gen-path-def is-gen-path-def g'.is-walk-def is-walk-def* **by** *blast*
then show *?thesis* **unfolding** *g'.vert-connected-def* **by** *blast*
qed
qed

lemma (**in** *ulgraph*) *vert-connected-remove-cycle-edge*:
assumes *cycle*: *is-cycle2* (*u#v#xs*)
shows *ulgraph.vert-connected V* $(E - \{\{u,v\}\})$ *u v*
proof−
interpret *g'*: *ulgraph V* $E - \{\{u,v\}\}$ **using** *wellformed edge-size* **by** (*unfold-locales*, *auto*)
have *conn-path*: *connecting-path v u* (*v#xs*) **using** *cycle is-cycle-connecting-path* **unfolding** *is-cycle2-def* **by** *blast*
have $\{u,v\} \notin$ *set* (*walk-edges* (*v#xs*)) **using** *cycle* **unfolding** *is-cycle2-def* **by** *simp*
then have *g'.connecting-path v u* (*v#xs*) **using** *conn-path*
unfolding *g'.connecting-path-def connecting-path-def g'.is-gen-path-def is-gen-path-def g'.is-walk-def is-walk-def* **by** *blast*

then show *?thesis* **using** *g′.vert-connected-rev* **unfolding** *g′.vert-connected-def*
by *blast*
qed

lemma (**in** *connected-ulgraph*) *connected-remove-cycle-edges*:
assumes *cycle*: *is-cycle2* (*u#v#xs*)
shows *connected-ulgraph* V (E − {{u,v}})
proof−
interpret *g′*: *ulgraph* V E − {{u,v}} **using** *wellformed edge-size* **by** (*unfold-locales*, *auto*)
have *g′.vert-connected* x y **if** *inV*: $x \in V$ $y \in V$ **for** x y
proof−
have *e*: {u,v} ∈ E **using** *cycle* **unfolding** *is-cycle2-def is-cycle-alt is-walk-def*
by *auto*
show *?thesis* **using** *vert-connected-remove-cycle-edge*[*OF cycle*] *vert-connected-remove-edge*[*OF e*] *g′.vert-connected-trans g′.vert-connected-rev inV* **by** *metis*
qed
then show *?thesis* **using** *not-empty* **by** (*unfold-locales*, *auto simp*: *g′.is-connected-set-def*)
qed

lemma (**in** *connected-ulgraph*) *connected-remove-leaf*:
assumes *degree*: *degree* l = *1*
and *remove-vertex*: *remove-vertex* l = (V', E')
shows *ulgraph.is-connected-set* V' E' V'
proof−
interpret *g′*: *ulgraph* V' E' **using** *remove-vertex wellformed edge-size*
unfolding *remove-vertex-def vincident-def* **by** (*unfold-locales*, *auto*)
have V': $V' = V - \{l\}$ **using** *remove-vertex* **unfolding** *remove-vertex-def* **by** *simp*
have E': $E' = \{e \in E.\ l \notin e\}$ **using** *remove-vertex* **unfolding** *remove-vertex-def vincident-def* **by** *simp*
have $u \in V' \Longrightarrow v \in V' \Longrightarrow$ *g′.vert-connected* u v **for** u v
proof−
assume *inV′*: $u \in V'$ $v \in V'$
then have *inV*: $u \in V$ $v \in V$ **using** *remove-vertex* **unfolding** *remove-vertex-def*
by *auto*
then obtain p **where** *conn-path*: *connecting-path* u v p **using** *vertices-connected-path*
by *blast*
show *?thesis*
proof (*cases* $u = v$)
case *True*
then show *?thesis* **using** *g′.vert-connected-id inV′* **by** *simp*
next
case *False*
then have *distinct*: *distinct* p **using** *conn-path* **unfolding** *connecting-path-def is-gen-path-def* **by** *blast*
have *l-notin-p*: $l \notin$ *set* p
proof
assume *l-in-p*: $l \in$ *set* p

```
        then obtain xs ys where p: p = xs @ l # ys by (meson split-list)
        have l ≠ u l ≠ v using inV' remove-vertex unfolding remove-vertex-def
by auto
        then have xs ≠ [] using p conn-path unfolding connecting-path-def by
fastforce
        then obtain x where last-xs: last xs = x by simp
        then have x ≠ l using distinct p ‹xs≠[]› by auto
          have {x,l} ∈ set (walk-edges p) using walk-edges-append-union ‹xs≠[]›
unfolding p
          by (simp add: walk-edges-append-union last-xs)
        then have xl-incident: {x,l} ∈ incident-sedges l using conn-path ‹x≠l›
        unfolding connecting-path-def is-gen-path-def is-walk-def incident-sedges-def
vincident-def by auto

        have ys ≠ [] using ‹l≠v› p conn-path unfolding connecting-path-def by
fastforce
        then obtain y ys' where ys: ys = y # ys' by (meson list.exhaust)
        then have y ≠ l using distinct p by auto
      then have {y,l} ∈ set (walk-edges p) using p ys conn-path walk-edges-append-ss1
by fastforce
        then have yl-incident: {y,l} ∈ incident-sedges l using conn-path ‹y≠l›
        unfolding connecting-path-def is-gen-path-def is-walk-def incident-sedges-def
vincident-def by auto

         have card-loops: card (incident-loops l) = 0 using degree unfolding de-
gree-def by auto
        have x ≠ y using distinct last-xs ‹xs≠[]› unfolding p ys by fastforce
        then have {x,l} ≠ {y,l} by (metis doubleton-eq-iff)
        then have card (incident-sedges l) ≠ 1 using xl-incident yl-incident
          by (metis card-1-singletonE singletonD)
        then have degree l ≠ 1 using card-loops unfolding degree-def by simp
        then show False using degree ..
      qed
      then have set (walk-edges p) ⊆ E' using walk-edges-in-verts conn-path E'
unfolding connecting-path-def is-gen-path-def is-walk-def by blast
      then have g'.connecting-path u v p using conn-path V' l-notin-p
           unfolding g'.connecting-path-def connecting-path-def g'.is-gen-path-def
is-gen-path-def g'.is-walk-def is-walk-def by blast
      then show ?thesis unfolding g'.vert-connected-def by blast
    qed
  qed
  then show ?thesis unfolding g'.is-connected-set-def by blast
qed

lemma (in connected-sgraph) connected-two-graph-edges:
  assumes u ≠ v
    and V: V = {u,v}
  shows E = {{u,v}}
proof−
```

obtain *p* **where** *conn-path*: *connecting-path u v p* **using** *V vertices-connected-path* **by** *blast*
then obtain p' **where** *p*: $p = u \# p' @ [v]$ **using** ‹$u \neq v$› **unfolding** *connecting-path-def is-gen-path-def*
by (*metis append-Nil is-walk-not-empty2 list.exhaust-sel list.sel(1) snoc-eq-iff-butlast tl-append2*)
have *distinct p* **using** *conn-path* ‹$u \neq v$› **unfolding** *connecting-path-def is-gen-path-def* **by** *auto*
then have $p' = []$ **using** *V conn-path is-gen-path-wf append-is-Nil-conv last-in-set self-append-conv2*
unfolding *connecting-path-def p* **by** *fastforce*
then have *edge-in-E*: $\{u,v\} \in E$ **using** ‹$u \neq v$› *conn-path*
unfolding *p connecting-path-def is-gen-path-def is-walk-def* **by** *simp*
have $E \subseteq \{\{\}, \{u\}, \{v\}, \{u,v\}\}$ **using** *wellformed V* **by** *blast*
then show *?thesis* **using** *two-edges edge-in-E* **by** *fastforce*
qed

1.8 Connected components

context *ulgraph*
begin

abbreviation *vert-connected-rel* $\equiv \{(u,v).$ *vert-connected u v*$\}$

definition *connected-components* :: *'a set set* **where**
connected-components = *V* // *vert-connected-rel*

definition *connected-component-of* :: *'a* ⇒ *'a set* **where**
connected-component-of v = *vert-connected-rel* ‘‘ $\{v\}$

lemma *vert-connected-rel-on-V*: *vert-connected-rel* $\subseteq V \times V$
using *vert-connected-wf* **by** *auto*

lemma *vert-connected-rel-refl*: *refl-on V vert-connected-rel*
unfolding *refl-on-def* **using** *vert-connected-rel-on-V vert-connected-id* **by** *simp*

lemma *vert-connected-rel-sym*: *sym vert-connected-rel*
unfolding *sym-def* **using** *vert-connected-rev* **by** *simp*

lemma *vert-connected-rel-trans*: *trans vert-connected-rel*
unfolding *trans-def* **using** *vert-connected-trans* **by** *blast*

lemma *equiv-vert-connected*: *equiv V vert-connected-rel*
unfolding *equiv-def* **using** *vert-connected-rel-refl vert-connected-rel-sym vert-connected-rel-trans* **by** *blast*

lemma *connected-component-non-empty*: $V' \in$ *connected-components* $\Longrightarrow V' \neq \{\}$
unfolding *connected-components-def* **using** *equiv-vert-connected in-quotient-imp-non-empty*

by *auto*

lemma *connected-component-connected*: $V' \in$ *connected-components* $\Longrightarrow$ *is-connected-set* V'
unfolding *connected-components-def is-connected-set-def* **using** *quotient-eq-iff* [*OF equiv-vert-connected*, *of* V' V'] **by** *simp*

lemma *connected-component-wf*: $V' \in$ *connected-components* $\Longrightarrow$ $V' \subseteq V$
by (*simp add*: *connected-component-connected is-connected-set-wf*)

lemma *connected-component-of-self*: $v \in V \Longrightarrow v \in$ *connected-component-of* v
unfolding *connected-component-of-def* **using** *vert-connected-id* **by** *blast*

lemma *conn-comp-of-conn-comps*: $v \in V \Longrightarrow$ *connected-component-of* $v \in$ *connected-components*
unfolding *connected-components-def quotient-def connected-component-of-def* **by** *blast*

lemma *Un-connected-components*: *connected-components* = *connected-component-of* ` V
unfolding *connected-components-def connected-component-of-def quotient-def* **by** *blast*

lemma *connected-component-subgraph*: $V' \in$ *connected-components* $\Longrightarrow$ *subgraph* V' (*induced-edges* V') V E
using *induced-is-subgraph connected-component-wf* **by** *simp*

lemma *connected-components-connected2*:
assumes *conn-comp*: $V' \in$ *connected-components*
shows *ulgraph.is-connected-set* V' (*induced-edges* V') V'
proof−
interpret *subg*: *subgraph* V' *induced-edges* V' V E **using** *connected-component-subgraph conn-comp* **by** *simp*
interpret g': *ulgraph* V' *induced-edges* V' **using** *subg.is-subgraph-ulgraph ulgraph-axioms* **by** *simp*
have $\bigwedge u\ v.\ u \in V' \Longrightarrow v \in V' \Longrightarrow g'$*.vert-connected* u v
proof−
fix u v **assume** $u \in V'$ $v \in V'$
then obtain p **where** *conn-path*: *connecting-path* u v p **using** *connected-component-connected conn-comp* **unfolding** *is-connected-set-def vert-connected-def* **by** *blast*
then have *u-in-p*: $u \in$ *set* p **unfolding** *connecting-path-def is-gen-path-def is-walk-def* **by** *force*
then have *set-p*: *set* $p \subseteq V'$ **using** *connecting-path-connected-set*[*OF conn-path*]
in-quotient-imp-closed[*OF equiv-vert-connected*] *conn-comp* ‹$u \in V'$›
unfolding *is-connected-set-def connected-components-def* **by** *blast*
then have *set* (g'*.walk-edges* p) $\subseteq$ *induced-edges* V'
using *walk-edges-induced-edges induced-edges-mono conn-path* **unfolding** *connecting-path-def is-gen-path-def* **by** *blast*
then have g'*.connecting-path* u v p

using *set-p conn-path*
unfolding *g′.connecting-path-def g′.connecting-path-def g′.is-gen-path-def g′.is-walk-def*
unfolding *connecting-path-def connecting-path-def is-gen-path-def is-walk-def*
by *auto*
then show *g′.vert-connected u v* **unfolding** *g′.vert-connected-def* **by** *blast*
qed
then show *?thesis* **unfolding** *g′.is-connected-set-def* **by** *blast*
qed

lemma *vert-connected-connected-component*: $C \in$ *connected-components* $\Longrightarrow u \in C \Longrightarrow$ *vert-connected u v* $\Longrightarrow v \in C$
unfolding *connected-components-def* **using** *equiv-vert-connected in-quotient-imp-closed*
by *fastforce*

lemma *connected-components-connected-ulgraphs*:
assumes *conn-comp*: $V' \in$ *connected-components*
shows *connected-ulgraph* V' (*induced-edges* V')
proof−
interpret *subg*: *subgraph* V' *induced-edges* V' V E **using** *connected-component-subgraph conn-comp* **by** *simp*
interpret *g′*: *ulgraph* V' *induced-edges* V' **using** *subg.is-subgraph-ulgraph ulgraph-axioms* **by** *simp*
show *?thesis* **using** *conn-comp connected-component-non-empty connected-components-connected2*
by (*unfold-locales*, *auto*)
qed

lemma *connected-components-partition-on-V*: *partition-on V connected-components*
using *partition-on-quotient equiv-vert-connected* **unfolding** *connected-components-def*
by *blast*

lemma *Union-connected-components*: $\bigcup$ *connected-components* = V
using *connected-components-partition-on-V* **unfolding** *partition-on-def* **by** *blast*

lemma *disjoint-connected-components*: *disjoint connected-components*
using *connected-components-partition-on-V* **unfolding** *partition-on-def* **by** *blast*

lemma *Union-induced-edges-connected-components*: $\bigcup$ (*induced-edges* ‘ *connected-components*) = E
proof−
have $\exists$ C∈*connected-components*. $e \in$ *induced-edges* C **if** $e \in E$ **for** e
proof−
obtain u v **where** e: $e = \{u,v\}$ **by** (*meson* ‹$e \in E$› *obtain-edge-pair-adj*)
then have *vert-connected u v* **using** *that vert-connected-neighbors* **by** *blast*
then have $v \in$ *connected-component-of u* **unfolding** *connected-component-of-def*
by *simp*
then have $e \in$ *induced-edges* (*connected-component-of u*) **using** *connected-component-of-self wellformed* ‹e∈E› **unfolding** *e induced-edges-def* **by** *auto*
then show *?thesis* **using** *conn-comp-of-conn-comps e wellformed* ‹e∈E› **by**

auto
qed
then show *?thesis* **using** *connected-component-wf induced-edges-ss* **by** *blast*
qed

lemma *connected-components-empty-E*:
assumes *empty*: $E = \{\}$
shows *connected-components* $= \{\{v\} \mid v.\ v \in V\}$
proof−
have $\forall v \in V.$ *vert-connected-rel*''$\{v\} = \{v\}$ **using** *vert-connected-id connected-empty-E empty* **by** *auto*
then show *?thesis* **unfolding** *connected-components-def quotient-def* **by** *auto*
qed

lemma *connected-iff-connected-components*:
assumes *non-empty*: $V \neq \{\}$
shows *is-connected-set* $V \longleftrightarrow$ *connected-components* $= \{V\}$
proof
assume *is-connected-set* V
then have $\forall v \in V.$ *connected-component-of* $v = V$ **unfolding** *connected-component-of-def is-connected-set-def* **using** *vert-connected-wf* **by** *blast*
then show *connected-components* $= \{V\}$ **unfolding** *quotient-def connected-component-of-def connected-components-def* **using** *non-empty* **by** *auto*
next
show *connected-components* $= \{V\} \Longrightarrow$ *is-connected-set* V
using *connected-component-connected* **unfolding** *connected-components-def is-connected-set-def* **by** *auto*
qed

end

lemma (**in** *connected-ulgraph*) *connected-components*[*simp*]: *connected-components* $= \{V\}$
using *connected connected-iff-connected-components not-empty* **by** *simp*

lemma (**in** *fin-ulgraph*) *finite-connected-components*: *finite connected-components*
unfolding *connected-components-def* **using** *finV vert-connected-rel-on-V finite-quotient* **by** *blast*

lemma (**in** *fin-ulgraph*) *finite-connected-component*: $C \in$ *connected-components* $\Longrightarrow$ *finite* C
using *connected-component-wf finV finite-subset* **by** *blast*

lemma (**in** *connected-ulgraph*) *connected-components-remove-edges*:
assumes *edge*: $\{u,v\} \in E$
shows *ulgraph.connected-components* V $(E - \{\{u,v\}\}) =$
$\{$*ulgraph.connected-component-of* V $(E - \{\{u,v\}\})$ u, *ulgraph.connected-component-of* V $(E - \{\{u,v\}\})$ $v\}$
proof−

interpret *g'*: *ulgraph V E* − {{*u*,*v*}} **using** *wellformed edge-size* **by** (*unfold-locales*, *auto*)
have *inV*: *u* ∈ *V v* ∈ *V* **using** *edge wellformed* **by** *auto*
have ∀ *w*∈*V*. *g'*.*connected-component-of w* = *g'*.*connected-component-of u* ∨ *g'*.*connected-component-of w* = *g'*.*connected-component-of v*
using *vert-connected-remove-edge*[*OF edge*] *g'*.*equiv-vert-connected equiv-class-eq* **unfolding** *g'*.*connected-component-of-def* **by** *fast*
then show *?thesis* **unfolding** *g'*.*connected-components-def quotient-def g'*.*connected-component-of-def* **using** *inV* **by** *auto*
qed

lemma (**in** *ulgraph*) *connected-set-connected-component*:
assumes *conn-set*: *is-connected-set C*
and *non-empty*: *C* ≠ {}
and ⋀*u v*. {*u*,*v*} ∈ *E* ⟹ *u* ∈ *C* ⟹ *v* ∈ *C*
shows *C* ∈ *connected-components*
proof−
have *walk-subset-C*: *is-walk xs* ⟹ *hd xs* ∈ *C* ⟹ *set xs* ⊆ *C* **for** *xs*
proof (*induction xs rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *auto*
next
case (*snoc x xs*)
then show *?case*
proof (*cases xs rule*: *rev-exhaust*)
case *Nil*
then show *?thesis* **using** *snoc* **by** *auto*
next
fix *ys y* **assume** *xs*: *xs* = *ys* @ [*y*]
then have *is-walk xs* **using** *is-walk-prefix snoc*(*2*) **by** *blast*
then have *set-xs-C*: *set xs* ⊆ *C* **using** *snoc xs is-walk-not-empty2 hd-append2* **by** *metis*
have *yx-E*: {*y*,*x*} ∈ *E* **using** *snoc*(*2*) *walk-edges-app* **unfolding** *xs is-walk-def* **by** *simp*
have *x* ∈ *C* **using** *assms*(*3*)[*OF yx-E*] *set-xs-C* **unfolding** *xs* **by** *simp*
then show *?thesis* **using** *set-xs-C* **by** *simp*
qed
qed
obtain *u* **where** *u* ∈ *C* **using** *non-empty* **by** *blast*
then have *u* ∈ *V* **using** *conn-set is-connected-set-wf* **by** *blast*
have *v* ∈ *C* **if** *vert-connected*: *vert-connected u v* **for** *v*
proof−
obtain *p* **where** *connecting-path u v p* **using** *vert-connected* **unfolding** *vert-connected-def* **by** *blast*
then show *?thesis* **using** *walk-subset-C*[*of p*] ‹*u*∈*C*› *is-walk-def last-in-set* **unfolding** *connecting-path-def is-gen-path-def* **by** *auto*
qed
then have *connected-component-of u* = *C* **using** *assms* ‹*u*∈*C*› **unfolding** *connected-component-of-def is-connected-set-def* **by** *auto*

then show *?thesis* **using** *conn-comp-of-conn-comps* ‹$u \in V$› **by** *blast*
qed

lemma (**in** *ulgraph*) *subset-conn-comps-if-Union*:
assumes *A-subset-conn-comps*: $A \subseteq$ *connected-components*
and *Un-A*: $\bigcup A = V$
shows $A =$ *connected-components*
proof (*rule ccontr*)
assume $A \neq$ *connected-components*
then obtain C **where** *C-conn-comp*: $C \in$ *connected-components* $C \notin A$ **using** *A-subset-conn-comps* **by** *blast*
then obtain v **where** $v \in C$ **using** *connected-component-non-empty* **by** *blast*
then have $v \notin V$ **using** *A-subset-conn-comps Un-A connected-components-partition-on-V C-conn-comp*
using *partition-onD4* **by** *fastforce*
then show *False* **using** *C-conn-comp connected-component-wf* ‹$v \in C$› **by** *auto*
qed

lemma (**in** *connected-ulgraph*) *exists-adj-vert-removed*:
assumes $v \in V$
and *remove-vertex*: *remove-vertex* $v = (V',E')$
and *conn-component*: $C \in$ *ulgraph.connected-components* V' E'
shows $\exists u \in C$. *vert-adj* v u
proof−
have V': $V' = V - \{v\}$ **and** E': $E' = \{e \in E.\ v \notin e\}$ **using** *remove-vertex* **unfolding** *remove-vertex-def vincident-def* **by** *auto*
interpret *subg*: *subgraph* $V - \{v\}$ $\{e \in E.\ v \notin e\}$ V E **using** *subgraph-remove-vertex remove-vertex* V' E' **by** *metis*
interpret g': *ulgraph* $V - \{v\}$ $\{e \in E.\ v \notin e\}$ **using** *subg.is-subgraph-ulgraph ulgraph-axioms* **by** *blast*
obtain c **where** $c \in C$ **using** g'*.connected-component-non-empty conn-component* V' E' **by** *blast*
then have $c \in V'$ **using** g'*.connected-component-wf conn-component* V' E' **by** *blast*
then have $c \in V$ **using** *subg.verts-ss* V' **by** *blast*
then obtain p **where** *conn-path*: *connecting-path* v c p **using** ‹$v \in V$› *vertices-connected-path* **by** *blast*
have $v \neq c$ **using** ‹$c \in V'$› *remove-vertex* **unfolding** *remove-vertex-def* **by** *blast*
then obtain u p' **where** p: $p = v \# u \# p'$ **using** *conn-path*
by (*metis connecting-path-def is-gen-path-def is-walk-def last.simps list.exhaust-sel*)
then have *conn-path-uc*: *connecting-path* u c $(u \# p')$ **using** *conn-path connecting-path-tl* **unfolding** p **by** *blast*
have *v-notin-p'*: $v \notin$ *set* $(u \# p')$ **using** *conn-path* ‹$v \neq c$› **unfolding** p *connecting-path-def is-gen-path-def* **by** *auto*
then have g'*.connecting-path* u c $(u \# p')$ **using** *conn-path-uc v-notin-p' walk-edges-in-verts*
unfolding g'*.connecting-path-def connecting-path-def* g'*.is-gen-path-def is-gen-path-def* g'*.is-walk-def is-walk-def*
by *blast*
then have g'*.vert-connected* u c **unfolding** g'*.vert-connected-def* **by** *blast*

then have *u* ∈ *C* **using** ‹*c*∈*C*› *conn-component g'.vert-connected-connected-component g'.vert-connected-rev* **unfolding** *V' E'* **by** *blast*
have *vert-adj v u* **using** *conn-path* **unfolding** *p connecting-path-def is-gen-path-def is-walk-def vert-adj-def* **by** *auto*
then show *?thesis* **using** ‹*u*∈*C*› **by** *blast*
qed

1.9 Trees

locale *tree* = *fin-connected-ulgraph* +
assumes *no-cycles*: ¬ *is-cycle2 c*
begin

sublocale *fin-connected-sgraph*
using *alt-edge-size no-cycles loop-is-cycle2 card-1-singletonE connected*
by (*unfold-locales*, *metis*, *simp*)

end

locale *spanning-tree* = *ulgraph V E* + *T*: *tree V T* **for** *V E T* +
assumes *subgraph*: *T* ⊆ *E*

lemma (**in** *fin-connected-ulgraph*) *has-spanning-tree*: ∃ *T*. *spanning-tree V E T*
using *fin-connected-ulgraph-axioms*
proof (*induction card E arbitrary*: *E*)
case *0*
then interpret *g*: *fin-connected-ulgraph V edges* **by** *blast*
have *edges*: *edges* = {} **using** *g.fin-edges 0* **by** *simp*
then obtain *v* **where** *V*: *V* = {*v*} **using** *g.V-E-empty* **by** *blast*
interpret *g'*: *fin-connected-sgraph V edges* **using** *g.connected edges* **by** (*unfold-locales*, *auto*)
interpret *t*: *tree V edges* **using** *g.length-cycle-card-V g'.cycle2-min-length g.is-cycle2-def V* **by** (*unfold-locales*, *fastforce*)
have *spanning-tree V edges edges* **by** (*unfold-locales*, *auto*)
then show *?case* **by** *blast*
next
case (*Suc m*)
then interpret *g*: *fin-connected-ulgraph V edges* **by** *blast*
show *?case*
proof (*cases* ∀ *c*. ¬*g.is-cycle2 c*)
case *True*
then have *spanning-tree V edges edges* **by** (*unfold-locales*, *auto*)
then show *?thesis* **by** *blast*
next
case *False*
then obtain *c* **where** *cycle*: *g.is-cycle2 c* **by** *blast*
then have *length c* ≥ *2* **unfolding** *g.is-cycle2-def g.is-cycle-alt walk-length-conv* **by** *auto*
then obtain *u v xs* **where** *c*: *c* = *u#v#xs* **by** (*metis Suc-le-length-iff nu-*

meral-2-eq-2)
then have *g'*: *fin-connected-ulgraph V* (*edges* − {{*u*,*v*}}) **using** *finV g.connected-remove-cycle-edges*
by (*metis connected-ulgraph-def cycle fin-connected-ulgraph-def fin-graph-system.intro fin-graph-system-axioms.intro fin-ulgraph.intro ulgraph-def*)
have {*u*,*v*} ∈ *edges* **using** *cycle* **unfolding** *c g.is-cycle2-def g.is-cycle-alt g.is-walk-def* **by** *auto*
then obtain *T* **where** *spanning-tree V* (*edges* − {{*u*,*v*}}) *T* **using** *Suc card-Diff-singleton g'* **by** *fastforce*
then have *spanning-tree V edges T* **unfolding** *spanning-tree-def spanning-tree-axioms-def* **using** *g.ulgraph-axioms* **by** *blast*
then show *?thesis* **by** *blast*
qed
qed

context *tree*
begin

definition *leaf* :: *'a* ⇒ *bool* **where**
leaf v ⟷ *degree v* = *1*

definition *leaves* :: *'a set* **where**
leaves = {*v*. *leaf v*}

definition *non-trivial* :: *bool* **where**
non-trivial ⟷ *card V* ≥ *2*

lemma *obtain-2-verts*:
assumes *non-trivial*
obtains *u v* **where** *u* ∈ *V v* ∈ *V u* ≠ *v*
using *assms* **unfolding** *non-trivial-def*
by (*meson diameter-obtains-path-vertices*)

lemma *leaf-in-V*: *leaf v* ⟹ *v* ∈ *V*
unfolding *leaf-def* **using** *degree-none* **by** *force*

lemma *exists-leaf*:
assumes *non-trivial*
shows ∃ *v*∈*V*. *leaf v*
proof−
obtain *p* **where** *is-path*: *is-path p* **and** *longest-path*: ∀ *s*. *is-path s* ⟶ *length s* ≤ *length p*
using *obtain-longest-path*
by (*metis One-nat-def assms connected connected-sgraph-axioms connected-sgraph-def degree-0-not-connected*
is-connected-setD is-edge-or-loop is-isolated-vertex-def is-isolated-vertex-degree0 is-loop-def
n-not-Suc-n numeral-2-eq-2 obtain-2-verts sgraph.two-edges vert-adj-def)
then obtain *l v xs* **where** *p*: *p* = *l*#*v*#*xs*
by (*metis is-open-walk-def is-path-def is-walk-not-empty2 last-ConsL list.exhaust-sel*)

```
  then have lv-incident: {l,v} ∈ incident-edges l using is-path
   unfolding incident-edges-def vincident-def is-path-def is-open-walk-def is-walk-def
by simp
  have ⋀e. e∈E ⟹ e ≠ {l,v} ⟹ e ∉ incident-edges l
  proof
    fix e
    assume e-in-E: e ∈ E
      and not-lv: e ≠ {l,v}
      and incident: e ∈ incident-edges l
    obtain u where e: e = {l,u} using e-in-E obtain-edge-pair-adj incident
      unfolding incident-edges-def vincident-def by auto
    then have u ≠ l using e-in-E edge-vertices-not-equal by blast
    have u ≠ v using e not-lv by auto
    have u-in-V: u ∈ V using e-in-E e wellformed by blast
    then show False
    proof (cases u ∈ set p)
      case True
      then have u ∈ set xs using ‹u≠l› ‹u≠v› p by simp
      then obtain ys zs where xs = ys@u#zs by (meson split-list)
      then have is-cycle2 (u#l#v#ys@[u])
          using is-path ‹u≠l› ‹u≠v› e-in-E distinct-edgesI walk-edges-append-ss2
walk-edges-in-verts
      unfolding is-cycle2-def is-cycle-def p is-path-def is-closed-walk-def is-open-walk-def
is-walk-def e walk-length-conv
        by (auto, metis insert-commute, fastforce+)
      then show ?thesis using no-cycles by blast
    next
      case False
      then have is-path (u#p) using is-path u-in-V e-in-E
          unfolding is-path-def is-open-walk-def is-walk-def e p by (auto, (metis
insert-commute)+)
      then show False using longest-path by auto
    qed
  qed
 then have incident-edges l = {{l,v}} using lv-incident unfolding incident-edges-def
by blast
  then have leaf: leaf l unfolding leaf-def alt-degree-def by simp
  then show ?thesis using leaf-in-V by blast
qed

lemma tree-remove-leaf:
  assumes leaf: leaf l
    and remove-vertex: remove-vertex l = (V',E')
  shows tree V' E'
proof−
  interpret g': ulgraph V' E' using remove-vertex wellformed edge-size unfolding
remove-vertex-def vincident-def
    by (unfold-locales, auto)
 interpret subg: ulsubgraph V' E' V E using subgraph-remove-vertex ulgraph-axioms
```

remove-vertex
unfolding *ulsubgraph-def* **by** *blast*
have V': $V' = V - \{l\}$ **using** *remove-vertex* **unfolding** *remove-vertex-def* **by** *blast*
have E': $E' = \{e{\in}E.\ l \notin e\}$ **using** *remove-vertex* **unfolding** *remove-vertex-def vincident-def* **by** *blast*
have $\exists v{\in}V.\ v \neq l$ **using** *leaf* **unfolding** *leaf-def*
by (*metis One-nat-def is-independent-alt is-isolated-vertex-def is-isolated-vertex-degree0 n-not-Suc-n radius-obtains singletonI singleton-independent-set*)
then have $V' \neq \{\}$ **using** *remove-vertex* **unfolding** *remove-vertex-def vincident-def* **by** *blast*
then have *g'.is-connected-set* V' **using** *connected-remove-leaf leaf remove-vertex* **unfolding** *leaf-def* **by** *blast*
then show *?thesis* **using** ‹$V'{\neq}\{\}$› *finV subg.is-cycle2* V' E' *no-cycles* **by** (*unfold-locales*, *auto*)
qed

end

lemma *tree-induct* [*case-names singolton insert*, *induct set*: *tree*]:
assumes *tree*: *tree V E*
and *trivial*: $\bigwedge v.$ *tree* $\{v\}$ $\{\} \Longrightarrow P$ $\{v\}$ $\{\}$
and *insert*: $\bigwedge l\ v\ V\ E.$ *tree* $V\ E \Longrightarrow P\ V\ E \Longrightarrow l \notin V \Longrightarrow v \in V \Longrightarrow \{l,v\} \notin E \Longrightarrow$ *tree.leaf* (*insert* $\{l,v\}$ E) $l \Longrightarrow P$ (*insert* l V) (*insert* $\{l,v\}$ E)
shows *P V E*
using *tree*
proof (*induction card V arbitrary*: *V E*)
case *0*
then interpret *tree V E* **by** *simp*
have $V = \{\}$ **using** *finV 0(1)* **by** *simp*
then show *?case* **using** *not-empty* **by** *blast*
next
case (*Suc n*)
then interpret *t*: *tree V E* **by** *simp*
show *?case*
proof (*cases card* $V = 1$)
case *True*
then obtain *v* **where** *V*: $V = \{v\}$ **using** *card-1-singletonE* **by** *blast*
then have $E = \{\}$
using *True subset-antisym t.edge-incident-vert t.vincident-def t.singleton-not-edge t.wellformed*
by *fastforce*
then show *?thesis* **using** *trivial t.tree-axioms V* **by** *simp*
next
case *False*
then have *card-V*: *card* $V \geq 2$ **using** *Suc* **by** *simp*
then obtain *l* **where** *leaf*: *t.leaf l* **using** *t.exists-leaf t.non-trivial-def* **by** *blast*
then obtain *e* **where** *inc-edges*: *t.incident-edges* $l = \{e\}$
unfolding *t.leaf-def t.alt-degree-def* **using** *card-1-singletonE* **by** *blast*

then have *e-in-E*: $e \in E$ **unfolding** *t.incident-edges-def* **by** *blast*
then obtain *u* **where** *e*: $e = \{l,u\}$ **using** *t.two-edges card-2-iff inc-edges* **unfolding** *t.incident-edges-def t.vincident-def*
by (*metis* (*no-types*, *lifting*) *empty-iff insert-commute insert-iff mem-Collect-eq*)
then have $l \neq u$ **using** *e-in-E t.edge-vertices-not-equal* **by** *blast*
have $u \in V$ **using** *e e-in-E t.wellformed* **by** *blast*
let $?V' = V - \{l\}$
let $?E' = E - \{\{l,u\}\}$
have *remove-vertex*: *t.remove-vertex l* = (*?V′*, *?E′*)
using *inc-edges e* **unfolding** *t.remove-vertex-def t.incident-edges-def* **by** *blast*
then have *t′*: *tree ?V′ ?E′* **using** *t.tree-remove-leaf leaf* **by** *blast*
have $l \in V$ **using** *leaf t.leaf-in-V* **by** *blast*
then have *P′*: *P ?V′ ?E′* **using** *Suc t′* **by** *auto*
show *?thesis* **using** *insert*[*OF t′ P′*] *Suc leaf* ‹u∈V› ‹l≠u› ‹l ∈ V› *e e-in-E* **by** (*auto*, *metis insert-Diff*)
qed
qed

context *tree*
begin

lemma *card-V-card-E*: *card V* = *Suc* (*card E*)
using *tree-axioms*
proof (*induction V E*)
case (*singolton v*)
then show *?case* **by** *auto*
next
case (*insert l v V′ E′*)
then interpret *t′*: *tree V′ E′* **by** *simp*
show *?case* **using** *t′.finV t′.fin-edges insert* **by** *simp*
qed

end

lemma *card-E-treeI*:
assumes *fin-conn-sgraph*: *fin-connected-ulgraph V E*
and *card-V-E*: *card V* = *Suc* (*card E*)
shows *tree V E*
proof−
interpret *G*: *fin-connected-ulgraph V E* **using** *fin-conn-sgraph* **.**
obtain *T* **where** *T*: *spanning-tree V E T* **using** *G.has-spanning-tree* **by** *blast*
show *?thesis*
proof (*cases E* = *T*)
case *True*
then show *?thesis* **using** *T* **unfolding** *spanning-tree-def* **by** *blast*
next
case *False*
then have *card E* > *card T* **using** *T G.fin-edges* **unfolding** *spanning-tree-def spanning-tree-axioms-def*

by (*simp add*: *psubsetI psubset-card-mono*)
then show *?thesis* **using** *tree.card-V-card-E T card-V-E* **unfolding** *spanning-tree-def* **by** *fastforce*
qed
qed

context *tree*
begin

lemma *add-vertex-tree*:
assumes $v \notin V$
and $w \in V$
shows *tree* (*insert v V*) (*insert* $\{v,w\}$ *E*)
proof −
let $?V' = insert\ v\ V$ **and** $?E' = insert\ \{v,w\}\ E$

have *cardV*: *card* $\{v,w\} = 2$ **using** *card-2-iff assms* **by** *auto*
then interpret *t'*: *ulgraph ?V' ?E'*
using *wellformed assms two-edges* **by** (*unfold-locales*, *auto*)

interpret *subg*: *ulsubgraph V E ?V' ?E'* **by** (*unfold-locales*, *auto*)

have *connected*: *t'.is-connected-set ?V'*
unfolding *t'.is-connected-set-def*
using *subg.vert-connected t'.vert-connected-neighbors t'.vert-connected-trans t'.vert-connected-id vertices-connected t'.ulgraph-axioms ulgraph-axioms assms t'.vert-connected-rev*
by *simp metis*

then have *fin-connected-ulgraph*: *fin-connected-ulgraph ?V' ?E'* **using** *finV* **by** (*unfold-locales*, *auto*)

from *assms* **have** $\{v,w\} \notin E$ **using** *wellformed-alt-fst* **by** *auto*
then have *card ?E'* = *Suc* (*card E*) **using** *fin-edges card-insert-if* **by** *auto*
then have *card ?V'* = *Suc* (*card ?E'*) **using** *card-V-card-E assms wellformed-alt-fst finV card-insert-if* **by** *auto*

then show *?thesis* **using** *card-E-treeI fin-connected-ulgraph* **by** *auto*
qed

lemma *tree-connected-set*:
assumes *non-empty*: $V' \neq \{\}$
and *subg*: $V' \subseteq V$
and *connected-V'*: *ulgraph.is-connected-set V'* (*induced-edges V'*) *V'*
shows *tree V'* (*induced-edges V'*)
proof−
interpret *subg*: *subgraph V' induced-edges V' V E* **using** *induced-is-subgraph subg* **by** *simp*
interpret *g'*: *ulgraph V' induced-edges V'* **using** *subg.is-subgraph-ulgraph ul-*

graph-axioms **by** *blast*
interpret *subg*: *ulsubgraph* V' *induced-edges* V' V E **by** *unfold-locales*
show *?thesis* **using** *connected-V'* *subg.is-cycle2* *no-cycles* *finV* *subg* *non-empty* *rev-finite-subset* **by** (*unfold-locales*) (*auto*, *blast*)
qed

lemma *unique-adj-vert-removed*:
assumes $v \in V$
and *remove-vertex*: *remove-vertex* $v = (V',E')$
and *conn-component*: $C \in$ *ulgraph.connected-components* V' E'
shows $\exists!u\in C.$ *vert-adj* v u
proof−
interpret *subg*: *ulsubgraph* V' E' V E **using** *remove-vertex* *subgraph-remove-vertex* *ulgraph-axioms* *ulsubgraph.intro* **by** *metis*
interpret g': *ulgraph* V' E' **using** *subg.is-subgraph-ulgraph* *ulgraph-axioms* **by** *simp*
obtain u **where** $u \in C$ **and** *adj-vu*: *vert-adj* v u **using** *exists-adj-vert-removed* **using** *assms* **by** *blast*
have $w = u$ **if** $w \in C$ **and** *adj-vw*: *vert-adj* v w **for** w
proof (*rule ccontr*)
assume $w \neq u$
obtain p **where** *g'-conn-path*: *g'.connecting-path* w u p **using** ‹$u\in C$› ‹$w\in C$› *conn-component*
g'.connected-component-connected *g'.is-connected-setD* *g'.vert-connected-def* **by** *blast*
then have *v-notin-p*: $v \notin$ *set* p **using** *remove-vertex* **unfolding** *g'.connecting-path-def* *g'.is-gen-path-def* *g'.is-walk-def* *remove-vertex-def* **by** *blast*
have *conn-path*: *connecting-path* w u p **using** *g'-conn-path* *subg.connecting-path* **by** *simp*
then obtain p' **where** p: $p = w \# p'$ @ $[u]$ **unfolding** *connecting-path-def* **using** ‹$w\neq u$›
by (*metis hd-Cons-tl last.simps last-rev rev-is-Nil-conv snoc-eq-iff-butlast*)
then have *walk-edges* $(v\#p$@$[v]) = \{v,w\}$ # *walk-edges* $((w \# p')$ @ $[u,v])$ **by** *simp*
also have ... $= \{v,w\}$ # *walk-edges* p @ $[\{u,v\}]$ **unfolding** p **using** *walk-edges-app* **by** (*metis Cons-eq-appendI*)
finally have *walk-edges*: *walk-edges* $(v\#p$@$[v]) = \{v,w\}$ # *walk-edges* p @ $[\{v,u\}]$ **by** (*simp add*: *insert-commute*)
then have *is-cycle* $(v\#p$@$[v])$ **using** *conn-path* *adj-vu* *adj-vw* ‹$w\neq u$› ‹$v\in V$› *g'.walk-length-conv* *singleton-not-edge* *v-notin-p*
unfolding *connecting-path-def* *is-cycle-def* *is-gen-path-def* *is-closed-walk-def* *is-walk-def* *p* *vert-adj-def* **by** *auto*
then have *is-cycle2* $(v\#p$@$[v])$ **using** ‹$w\neq u$› *v-notin-p* *walk-edges-in-verts* **unfolding** *is-cycle2-def* *walk-edges*
by (*auto simp*: *doubleton-eq-iff* *is-cycle-alt* *distinct-edgesI*)
then show *False* **using** *no-cycles* **by** *blast*
qed
then show *?thesis* **using** ‹$u\in C$› *adj-vu* **by** *blast*
qed

```
lemma non-trivial-card-E: non-trivial ⟹ card E ≥ 1
  using card-V-card-E unfolding non-trivial-def by simp

lemma V-Union-E: non-trivial ⟹ V = ⋃ E
  using tree-axioms
proof (induction V E)
  case (singolton v)
  then interpret t: tree {v} {} by simp
  show ?case using singolton unfolding t.non-trivial-def by simp
next
  case (insert l v V' E')
  then interpret t: tree V' E' by simp
  show ?case
  proof (cases card V' = 1)
    case True
    then have V: V' = {v} using insert(3) card-1-singletonE by blast
    then have E: E' = {} using t.fin-edges t.card-V-card-E by fastforce
    then show ?thesis unfolding E V by simp
  next
    case False
    then have t.non-trivial using t.card-V-card-E unfolding t.non-trivial-def by
simp
    then show ?thesis using insert by blast
  qed
qed

end

lemma singleton-tree: tree {v} {}
proof−
  interpret g: fin-ulgraph {v} {} by (unfold-locales, auto)
  show ?thesis using g.is-walk-def g.walk-length-def by (unfold-locales, auto simp:
g.is-connected-set-singleton g.is-cycle2-def g.is-cycle-alt)
qed

lemma tree2:
  assumes u ≠ v
    shows tree {u,v} {{u,v}}
proof−
  interpret ulgraph {u,v} {{u,v}} using ‹u≠v› by unfold-locales auto
  have fin-connected-ulgraph {u,v} {{u,v}} by unfold-locales
   (auto simp: is-connected-set-def vert-connected-id vert-connected-neighbors vert-connected-rev)
  then show ?thesis using card-E-treeI ‹u≠v› by fastforce
qed
```

1.10 Graph Isomorphism

```
locale graph-isomorphism =
```

G: *graph-system* V_G E_G **for** V_G E_G +
fixes V_H E_H f
assumes *bij-f*: *bij-betw* f V_G V_H
and *edge-preserving*: $((`)\ f)$ ` $E_G = E_H$
begin

lemma *inj-f*: *inj-on* f V_G
using *bij-f* **unfolding** *bij-betw-def* **by** *blast*

lemma V_H*-def*: $V_H = f$ ` V_G
using *bij-f* **unfolding** *bij-betw-def* **by** *blast*

definition *inv-iso* $\equiv$ *the-inv-into* V_G f

lemma *graph-system-H*: *graph-system* V_H E_H
using *G.wellformed edge-preserving bij-f bij-betw-imp-surj-on* **by** *unfold-locales blast*

interpretation H: *graph-system* V_H E_H **using** *graph-system-H* **.**

lemma *graph-isomorphism-inv*: *graph-isomorphism* V_H E_H V_G E_G *inv-iso*
proof (*unfold-locales*)
show *bij-betw inv-iso* V_H V_G **unfolding** *inv-iso-def* **using** *bij-betw-the-inv-into bij-f* **by** *blast*
next
have $\forall v \in V_G$. *the-inv-into* V_G f $(f\ v) = v$ **using** *bij-f* **by** (*simp add*: *bij-betw-imp-inj-on the-inv-into-f-f*)
then have $\forall e \in E_G$. $(\lambda v.$ *the-inv-into* V_G f $(f\ v))$ ` $e = e$ **using** *G.wellformed*
by (*simp add*: *subset-iff*)
then show $((`)$ *inv-iso*$)$ ` $E_H = E_G$ **unfolding** *inv-iso-def* **by** (*simp add*: *edge-preserving*[*symmetric*] *image-comp*)
qed

interpretation *inv-iso*: *graph-isomorphism* V_H E_H V_G E_G *inv-iso* **using** *graph-isomorphism-inv* **.**

end

fun *graph-isomorph* :: *'a pregraph* $\Rightarrow$ *'b pregraph* $\Rightarrow$ *bool* (**infix** ‹≃› *50*) **where**
$(V_G,E_G) \simeq (V_H,E_H) \longleftrightarrow (\exists f.$ *graph-isomorphism* V_G E_G V_H E_H $f)$

lemma (**in** *graph-system*) *graph-isomorphism-id*: *graph-isomorphism* V E V E *id*
by *unfold-locales auto*

lemma (**in** *graph-system*) *graph-isomorph-refl*: $(V,E) \simeq (V,E)$
using *graph-isomorphism-id* **by** *auto*

lemma *graph-isomorph-sym*: *symp* $(\simeq)$
using *graph-isomorphism.graph-isomorphism-inv* **unfolding** *symp-def* **by** *fast-*

force

lemma *graph-isomorphism-trans*: *graph-isomorphism* V_G E_G V_H E_H *f* $\Longrightarrow$ *graph-isomorphism* V_H E_H V_F E_F *g* $\Longrightarrow$ *graph-isomorphism* V_G E_G V_F E_F (*g o f*)
unfolding *graph-isomorphism-def graph-isomorphism-axioms-def* **using** *bij-betw-trans*
by (*auto*, *blast*)

lemma *graph-isomorph-trans*: *transp* ($\simeq$)
using *graph-isomorphism-trans* **unfolding** *transp-def* **by** *fastforce*

end

2 Enumeration of Labeled Trees

theory *Labeled-Tree-Enumeration*
imports *Tree-Graph*
begin

definition *labeled-trees* :: *'a set* ⇒ *'a pregraph set* **where**
labeled-trees V = {(*V*,*E*)| *E*. *tree V E*}

2.1 Algorithm

Prüfer sequence to tree

definition *prufer-sequences* :: *'a list* ⇒ *'a list set* **where**
prufer-sequences verts = {*xs*. *length xs* = *length verts* − *2* ∧ *set xs* ⊆ *set verts*}

fun *tree-edges-of-prufer-seq* :: *'a list* ⇒ *'a list* ⇒ *'a edge set* **where**
tree-edges-of-prufer-seq [*u*,*v*] [] = {{*u*,*v*}}
| *tree-edges-of-prufer-seq verts* (*b*#*seq*) =
(*case find* (λ*x*. *x* ∉ *set* (*b*#*seq*)) *verts of*
Some a ⇒ *insert* {*a*,*b*} (*tree-edges-of-prufer-seq* (*remove1 a verts*) *seq*))

definition *tree-of-prufer-seq* :: *'a list* ⇒ *'a list* ⇒ *'a pregraph* **where**
tree-of-prufer-seq verts seq = (*set verts*, *tree-edges-of-prufer-seq verts seq*)

definition *labeled-tree-enum* :: *'a list* ⇒ *'a pregraph list* **where**
labeled-tree-enum verts = *map* (*tree-of-prufer-seq verts*) (*List.n-lists* (*length verts* − *2*) *verts*)

2.2 Correctness

Tree to Prüfer sequence

definition *remove-vertex-edges* :: *'a* ⇒ *'a edge set* ⇒ *'a edge set* **where**
remove-vertex-edges v E = {*e*∈*E*. ¬ *graph-system.vincident v e*}

lemma *find-in-list*[*termination-simp*]: *find P verts* = *Some v* $\Longrightarrow$ *v* ∈ *set verts*
by (*metis find-Some-iff nth-mem*)

lemma [*termination-simp*]: *find P verts = Some v* $\Longrightarrow$ *length verts − Suc 0 < length verts*
by (*meson diff-Suc-less length-pos-if-in-set find-in-list*)

fun *prufer-seq-of-tree* :: *'a list* ⇒ *'a edge set* ⇒ *'a list* **where**
prufer-seq-of-tree verts E =
(*if length verts* ≤ *2 then* []
else (*case find* (*tree.leaf E*) *verts of*
Some leaf ⇒ (*THE v. ulgraph.vert-adj E leaf v*) # *prufer-seq-of-tree* (*remove1 leaf verts*) (*remove-vertex-edges leaf E*)))

locale *valid-verts* =
fixes *verts*
assumes *length-verts*: *length verts* ≥ *2*
and *distinct-verts*: *distinct verts*

locale *tree-of-prufer-seq-ctx* = *valid-verts* +
fixes *seq*
assumes *prufer-seq*: *seq* ∈ *prufer-sequences verts*

lemma (**in** *valid-verts*) *card-verts*: *card* (*set verts*) = *length verts*
using *length-verts distinct-verts distinct-card* **by** *blast*

lemma *length-gt-find-not-in-ys*:
assumes *length xs* > *length ys*
and *distinct xs*
shows ∃ *x. find* (λ*x. x* ∉ *set ys*) *xs* = *Some x*
proof−
have *card* (*set xs*) > *card* (*set ys*)
by (*metis assms card-length distinct-card le-neq-implies-less order-less-trans*)
then have ∃ *x*∈*set xs. x* ∉ *set ys*
by (*meson finite-set card-subset-not-gt-card subsetI*)
then show *?thesis* **by** (*metis find-None-iff2 not-Some-eq*)
qed

lemma (**in** *tree-of-prufer-seq-ctx*) *tree-edges-of-prufer-seq-induct'*:
assumes ⋀*u v. P* [*u*, *v*] []
and ⋀*verts b seq a.*
find (λ*x. x* ∉ *set* (*b* # *seq*)) *verts* = *Some a*
$\Longrightarrow$ *a* ∈ *set verts* $\Longrightarrow$ *a* ∉ *set* (*b* # *seq*) $\Longrightarrow$ *seq* ∈ *prufer-sequences* (*remove1 a verts*)
$\Longrightarrow$ *tree-of-prufer-seq-ctx* (*remove1 a verts*) *seq* $\Longrightarrow$ *P* (*remove1 a verts*) *seq* $\Longrightarrow$ *P verts* (*b* # *seq*)
shows *P verts seq*
using *tree-of-prufer-seq-ctx-axioms*
proof (*induction verts seq rule*: *tree-edges-of-prufer-seq.induct*)
case (*2 verts b seq*)
then interpret *tree-of-prufer-seq-ctx verts b* # *seq* **by** *simp*

obtain *a* **where** *a-find*: *find* (λ*x*. *x* ∉ *set* (*b* # *seq*)) *verts* = *Some a*
using *length-gt-find-not-in-ys*[*of b#seq verts*] *distinct-verts prufer-seq*
unfolding *prufer-sequences-def* **by** *fastforce*
then have *a-in-verts*: *a* ∈ *set verts* **by** (*simp add*: *find-in-list*)
have *a-not-in-seq*: *a* ∉ *set* (*b#seq*) **using** *a-find* **by** (*metis find-Some-iff*)
have *prufer-seq′*: *seq* ∈ *prufer-sequences* (*remove1 a verts*)
using *prufer-seq a-in-verts set-remove1-eq length-verts a-not-in-seq distinct-verts*
unfolding *prufer-sequences-def* **by** (*auto simp*: *length-remove1*)
have *length verts* ≥ *3* **using** *prufer-seq* **unfolding** *prufer-sequences-def* **by** *auto*
then have *length* (*remove1 a verts*) ≥ *2* **by** (*auto simp*: *length-remove1*)
then have *valid-verts-seq′*: *tree-of-prufer-seq-ctx* (*remove1 a verts*) *seq*
using *prufer-seq′ distinct-verts* **by** *unfold-locales auto*
then show *?case* **using** *a-find assms*(*2*) *a-in-verts a-not-in-seq prufer-seq′ 2*(*1*)
by *blast*
qed (*auto simp*: *assms tree-of-prufer-seq-ctx-def tree-of-prufer-seq-ctx-axioms-def valid-verts-def prufer-sequences-def*)

lemma (**in** *tree-of-prufer-seq-ctx*) *tree-edges-of-prufer-seq-tree*:
shows *tree* (*set verts*) (*tree-edges-of-prufer-seq verts seq*)
using *tree-of-prufer-seq-ctx-axioms*
proof (*induction rule*: *tree-edges-of-prufer-seq-induct′*)
case (*1 u v*)
then show *?case* **using** *tree2* **unfolding** *tree-of-prufer-seq-ctx-def valid-verts-def*
by *fastforce*
next
case (*2 verts b seq a*)
interpret *tree-of-prufer-seq-ctx verts b* # *seq* **using** *2*(*7*) **.**
interpret *tree set* (*remove1 a verts*) *tree-edges-of-prufer-seq* (*remove1 a verts*) *seq*
using *2*(*5*,*6*) **by** *simp*
have *a-not-in-verts′*: *a* ∉ *set* (*remove1 a verts*) **using** *distinct-verts* **by** *simp*
have *a* ≠ *b* **using** *2* **by** *auto*
then have *b-in-verts′*: *b* ∈ *set* (*remove1 a verts*) **using** *prufer-seq* **unfolding** *prufer-sequences-def* **by** *auto*
then show *?case* **using** *a-not-in-verts′ add-vertex-tree*[*OF a-not-in-verts′ b-in-verts′*] *2*(*1*,*2*) *distinct-verts*
by (*auto simp*: *insert-absorb insert-commute*)
qed

lemma (**in** *tree-of-prufer-seq-ctx*) *tree-of-prufer-seq-tree*: (*V*,*E*) = *tree-of-prufer-seq verts seq* ⟹ *tree V E*
unfolding *tree-of-prufer-seq-def* **using** *tree-edges-of-prufer-seq-tree* **by** *auto*

lemma (**in** *valid-verts*) *labeled-tree-enum-trees*:
assumes *VE-in-labeled-tree-enum*: (*V*,*E*) ∈ *set* (*labeled-tree-enum verts*)
shows *tree V E*
proof−
obtain *seq* **where** *seq* ∈ *set* (*List.n-lists* (*length verts* − *2*) *verts*) **and** *tree-of-seq*: *tree-of-prufer-seq verts seq* = (*V*,*E*)

using *VE-in-labeled-tree-enum* **unfolding** *labeled-tree-enum-def* **by** *auto*
then interpret *tree-of-prufer-seq-ctx verts seq*
using *List.set-n-lists* **by** (*unfold-locales*) (*auto simp*: *prufer-sequences-def*)
show *?thesis* **using** *tree-of-prufer-seq-tree* **using** *tree-of-seq* **by** *simp*
qed

2.3 Totality

locale *prufer-seq-of-tree-context* =
valid-verts verts + *tree set verts E* **for** *verts E*
begin

lemma *prufer-seq-of-tree-induct′*:
assumes ⋀*u v*. *P* [*u*,*v*] {{*u*,*v*}}
and ⋀*verts E l*. ¬ *length verts* ≤ *2* ⟹ *find* (*tree.leaf E*) *verts* = *Some l* ⟹ *tree.leaf E l*
⟹ *l* ∈ *set verts* ⟹ *prufer-seq-of-tree-context* (*remove1 l verts*) (*remove-vertex-edges l E*)
⟹ *P* (*remove1 l verts*) (*remove-vertex-edges l E*) ⟹ *P verts E*
shows *P verts E*
using *prufer-seq-of-tree-context-axioms*
proof (*induction verts E rule*: *prufer-seq-of-tree.induct*)
case (*1 verts E*)
then interpret *ctx*: *prufer-seq-of-tree-context verts E* **by** *simp*
show *?case*
proof (*cases length verts* ≤ *2*)
case *True*
then have *length-verts*: *length verts* = *2* **using** *ctx.length-verts* **by** *simp*
then obtain *u w* **where** *verts*: *verts* = [*u*,*w*]
unfolding *numeral-2-eq-2* **by** (*metis length-0-conv length-Suc-conv*)
then have *E* = {{*u*,*w*}} **using** *ctx.connected-two-graph-edges ctx.distinct-verts* **by** *simp*
then show *?thesis* **using** *assms*(*1*) *verts* **by** *blast*
next
case *False*
then have *ctx.non-trivial* **using** *ctx.distinct-verts distinct-card*
unfolding *ctx.non-trivial-def* **by** *fastforce*
then obtain *l* **where** *l*: *find ctx.leaf verts* = *Some l* **using** *ctx.exists-leaf*
by (*metis find-None-iff2 not-Some-eq*)
then have *leaf-l*: *ctx.leaf l* **by** (*metis find-Some-iff*)
then have *l-in-verts*: *l* ∈ *set verts* **using** *ctx.leaf-in-V* **by** *simp*
then have *length-verts′*: *length* (*remove1 l verts*) ≥ *2* **using** *False* **unfolding** *length-remove1* **by** *simp*
have *tree* (*set* (*remove1 l verts*)) (*remove-vertex-edges l E*) **using** *ctx.tree-remove-leaf*[*OF leaf-l*]
unfolding *ctx.remove-vertex-def remove-vertex-edges-def* **using** *ctx.distinct-verts* **by** *simp*
then have *ctx′*: *prufer-seq-of-tree-context* (*remove1 l verts*) (*remove-vertex-edges l E*)

unfolding *prufer-seq-of-tree-context-def valid-verts-def*
using *ctx.distinct-verts length-verts'* **by** *simp*
then have *P* (*remove1 l verts*) (*remove-vertex-edges l E*) **using** *1 False l* **by** *simp*
then show *?thesis* **using** *assms*(*2*)[*OF False l leaf-l l-in-verts ctx'*] **by** *simp*
qed
qed

lemma *prufer-seq-of-tree-wf*: *set* (*prufer-seq-of-tree verts E*) ⊆ *set verts*
using *prufer-seq-of-tree-context-axioms*
proof (*induction rule*: *prufer-seq-of-tree-induct'*)
case (*1 u v*)
then show *?case* **by** *simp*
next
case (*2 verts E l*)
then interpret *ctx*: *prufer-seq-of-tree-context verts E* **by** *simp*
let *?u* = *THE u. ctx.vert-adj l u*
have *l-u-adj*: *ctx.vert-adj l ?u* **using** *ctx.ex1-neighbor-degree-1 2*(*3*) **unfolding** *ctx.leaf-def* **by** (*metis theI*)
then have *?u* ∈ *set verts* **unfolding** *ctx.vert-adj-def* **using** *ctx.wellformed-alt-snd* **by** *blast*
then show *?case* **using** *2 ctx.ex1-neighbor-degree-1 2*(*3*)
by (*auto*, *meson in-mono notin-set-remove1*)
qed

lemma *length-prufer-seq-of-tree*: *length* (*prufer-seq-of-tree verts E*) = *length verts* − *2*
proof (*induction rule*: *prufer-seq-of-tree-induct'*)
case (*1 u v*)
then show *?case* **by** *simp*
next
case (*2 verts E l*)
then show *?case* **unfolding** *prufer-seq-of-tree.simps*[*of verts*] **by** (*simp add*: *length-remove1*)
qed

lemma *prufer-seq-of-tree-prufer-seq*: *prufer-seq-of-tree verts E* ∈ *prufer-sequences verts*
using *prufer-seq-of-tree-wf length-prufer-seq-of-tree* **unfolding** *prufer-sequences-def* **by** *blast*

lemma *count-list-prufer-seq-degree*: *v* ∈ *set verts* ⟹ *Suc* (*count-list* (*prufer-seq-of-tree verts E*) *v*) = *degree v*
using *prufer-seq-of-tree-context-axioms*
proof (*induction rule*: *prufer-seq-of-tree-induct'*)
case (*1 u v*)
then interpret *ctx*: *prufer-seq-of-tree-context* [*u*, *v*] {{*u*, *v*}} **by** *simp*
show *?case* **using** *1*(*1*) **unfolding** *ctx.alt-degree-def ctx.incident-edges-def ctx.vincident-def*
by (*simp add*: *Collect-conv-if*)

next
  case (*2 verts E l*)
  then interpret *ctx*: *prufer-seq-of-tree-context verts E* **by** *simp*
  interpret *ctx'*: *prufer-seq-of-tree-context remove1 l verts remove-vertex-edges l E*
using *2*(*5*) **by** *simp*
  let *?u* = *THE u. ctx.vert-adj l u*
  have *l-u-adj*: *ctx.vert-adj l ?u* **using** *ctx.ex1-neighbor-degree-1 2*(*3*) **unfolding**
ctx.leaf-def **by** (*metis theI*)
  show *?case*
  proof (*cases v* = *?u*)
    case *True*
    then have $v \neq l$ **using** *l-u-adj ctx.vert-adj-not-eq* **by** *blast*
    then have *count-list* (*prufer-seq-of-tree verts E*) *v* = *ulgraph.degree* (*remove-vertex-edges*
l E) *v*
      using *2 True* **by** *simp*
    then show *?thesis* **using** *2 ctx.degree-remove-adj-ne-vert* ‹*v*≠*l*› *True l-u-adj*
    unfolding *ctx.remove-vertex-def remove-vertex-edges-def prufer-seq-of-tree.simps*[*of*
verts] **by** *simp*
  next
    case *False*
    then show *?thesis*
    proof (*cases v* = *l*)
      case *True*
      then have $l \notin$ *set* (*remove1 l verts*) **using** *ctx.distinct-verts* **by** *simp*
      then have $l \notin$ *set* (*prufer-seq-of-tree* (*remove1 l verts*) (*remove-vertex-edges*
l E)) **using** *ctx'.prufer-seq-of-tree-wf* **by** *blast*
    then show *?thesis* **using** *2 False True* **unfolding** *ctx.leaf-def prufer-seq-of-tree.simps*[*of*
verts] **by** *simp*
    next
      case *False*
      then have ¬ *ctx.vert-adj l v* **using** ‹*v*≠*?u*› *ctx.ex1-neighbor-degree-1 2*(*3*)
l-u-adj
        unfolding *ctx.leaf-def* **by** *blast*
      then show *?thesis* **using** *False 2* ‹*v*≠*?u*› *ctx.degree-remove-non-adj-vert*
      unfolding *prufer-seq-of-tree.simps*[*of verts*] *ctx'.remove-vertex-def remove-vertex-edges-def*
ctx.remove-vertex-def **by** *auto*
    qed
  qed
qed

lemma *not-in-prufer-seq-iff-leaf*: $v \in$ *set verts* $\Longrightarrow v \notin$ *set* (*prufer-seq-of-tree verts*
E) $\longleftrightarrow$ *leaf v*
  using *count-list-prufer-seq-degree*[*symmetric*] **unfolding** *leaf-def* **by** (*simp add*:
count-list-0-iff)

lemma *tree-edges-of-prufer-seq-of-tree*: *tree-edges-of-prufer-seq verts* (*prufer-seq-of-tree*
verts E) = *E*
  using *prufer-seq-of-tree-context-axioms*
proof (*induction rule*: *prufer-seq-of-tree-induct'*)

case (*1 u v*)
then show *?case* **by** *simp*
next
case (*2 verts E l*)
then interpret *ctx*: *prufer-seq-of-tree-context verts E* **by** *simp*
have *tree-edges-of-prufer-seq verts* (*prufer-seq-of-tree verts E*)
= *tree-edges-of-prufer-seq verts* ((*THE v. ctx.vert-adj l v*) # *prufer-seq-of-tree* (*remove1 l verts*) (*remove-vertex-edges l E*)) **using** *2* **by** *simp*
have *find* (*λx. x* ∉ *set* (*prufer-seq-of-tree verts E*)) *verts* = *Some l* **using** *ctx.not-in-prufer-seq-iff-leaf 2*(*2*)
by (*metis* (*no-types*, *lifting*) *find-cong*)
then have *tree-edges-of-prufer-seq verts* (*prufer-seq-of-tree verts E*)
= *insert* {*The* (*ctx.vert-adj l*), *l*} (*tree-edges-of-prufer-seq* (*remove1 l verts*) (*prufer-seq-of-tree* (*remove1 l verts*) (*remove-vertex-edges l E*)))
using *2* **by** *auto*
also have ... = *E* **using** *2 ctx.degree-1-edge-partition* **unfolding** *remove-vertex-edges-def vincident-def ctx.leaf-def* **by** *simp*
finally show *?case* **.**
qed

lemma *tree-in-labeled-tree-enum*: (*set verts*, *E*) ∈ *set* (*labeled-tree-enum verts*)
using *prufer-seq-of-tree-prufer-seq tree-edges-of-prufer-seq-of-tree List.set-n-lists*
unfolding *prufer-sequences-def labeled-tree-enum-def tree-of-prufer-seq-def* **by** *fastforce*

end

lemma (**in** *valid-verts*) *V-labeled-tree-enum-verts*: (*V*,*E*) ∈ *set* (*labeled-tree-enum verts*) ⟹ *V* = *set verts*
unfolding *labeled-tree-enum-def* **by** (*metis Pair-inject ex-map-conv tree-of-prufer-seq-def*)

theorem (**in** *valid-verts*) *labeled-tree-enum-correct*: *set* (*labeled-tree-enum verts*) = *labeled-trees* (*set verts*)
using *labeled-tree-enum-trees V-labeled-tree-enum-verts prufer-seq-of-tree-context.tree-in-labeled-tree-enum valid-verts-axioms*
unfolding *labeled-trees-def prufer-seq-of-tree-context-def* **by** *fast*

2.4 Distinction

lemma (**in** *tree-of-prufer-seq-ctx*) *count-prufer-seq-degree*:
assumes *v-in-verts*: *v* ∈ *set verts*
shows *Suc* (*count-list seq v*) = *ulgraph.degree* (*tree-edges-of-prufer-seq verts seq*) *v*
using *v-in-verts tree-of-prufer-seq-ctx-axioms*
proof (*induction rule*: *tree-edges-of-prufer-seq-induct'*)
case (*1 u w*)
then interpret *tree-of-prufer-seq-ctx* [*u*, *w*] [] **by** *simp*
interpret *tree* {*u*,*w*} {{*u*,*w*}} **using** *tree-edges-of-prufer-seq-tree* **by** *simp*
show *?case* **using** *1*(*1*) **by** (*auto simp add*: *incident-edges-def vincident-def Col-*

lect-conv-if)
next
  case (*2 verts b seq a*)
  interpret *tree-of-prufer-seq-ctx verts b # seq* **using** *2*(*8*) **.**
  interpret *tree set verts tree-edges-of-prufer-seq verts* (*b#seq*)
    using *tree-edges-of-prufer-seq-tree* **by** *simp*
  interpret *ctx'*: *tree-of-prufer-seq-ctx remove1 a verts seq* **using** *2*(*5*) **.**
  interpret *T'*: *tree set* (*remove1 a verts*) *tree-edges-of-prufer-seq* (*remove1 a verts*) *seq*
    using *ctx'.tree-edges-of-prufer-seq-tree* **by** *simp*
  show *?case*
  proof (*cases v = b*)
    case *True*
    have *ab-not-in-T'*: {*a*, *b*} ∉ *tree-edges-of-prufer-seq* (*remove1 a verts*) *seq*
      using *T'.wellformed-alt-snd distinct-verts* **by** (*auto*, *metis doubleton-eq-iff*)
    have *incident-edges v = insert* {*a*,*b*} {*e* ∈ *tree-edges-of-prufer-seq* (*remove1 a verts*) *seq*. *v* ∈ *e*}
      unfolding *incident-edges-def vincident-def* **using** *2*(*1*) *True* **by** *auto*
    then have *degree v = Suc* (*T'.degree v*)
      unfolding *T'.alt-degree-def alt-degree-def T'.incident-edges-def vincident-def*
      using *ab-not-in-T' T'.fin-edges* **by** (*simp del*: *tree-edges-of-prufer-seq.simps*)
    then show *?thesis* **using** *2 True* **by** *auto*
  next
    case *False*
    then show *?thesis*
    proof (*cases v = a*)
      case *True*
      also have *incident-edges a* = {{*a*,*b*}} **unfolding** *incident-edges-def vincident-def*
        using *2*(*1*) *T'.wellformed distinct-verts* **by** *auto*
      then show *?thesis* **unfolding** *alt-degree-def True* **using** *2*(*3*) **by** *auto*
    next
      case *False*
      then have *incident-edges v = T'.incident-edges v*
       unfolding *incident-edges-def T'.incident-edges-def vincident-def* **using** *2*(*1*) ‹*v* ≠ *b*› **by** *auto*
      then show *?thesis* **using** *False* ‹*v* ≠ *b*› *2* **unfolding** *alt-degree-def* **by** *simp*
    qed
  qed
qed

lemma (**in** *tree-of-prufer-seq-ctx*) *notin-prufer-seq-iff-leaf*:
  assumes *v* ∈ *set verts*
  shows *v* ∉ *set seq* ⟷ *tree.leaf* (*tree-edges-of-prufer-seq verts seq*) *v*
proof−
  interpret *tree set verts tree-edges-of-prufer-seq verts seq*
    using *tree-edges-of-prufer-seq-tree* **by** *auto*
  show *?thesis* **using** *count-prufer-seq-degree assms count-list-0-iff* **unfolding** *leaf-def* **by** *fastforce*

```
qed

lemma (in valid-verts) inj-tree-edges-of-prufer-seq: inj-on (tree-edges-of-prufer-seq
verts) (prufer-sequences verts)
proof
  fix seq1 seq2
  assume prufer-seq1: seq1 ∈ prufer-sequences verts
  assume prufer-seq2: seq2 ∈ prufer-sequences verts
  assume trees-eq: tree-edges-of-prufer-seq verts seq1 = tree-edges-of-prufer-seq
verts seq2
  interpret tree-of-prufer-seq-ctx verts seq1 using prufer-seq1 by unfold-locales
simp
  have length-eq: length seq1 = length seq2 using prufer-seq1 prufer-seq2 unfolding prufer-sequences-def by simp
  show seq1 = seq2
    using prufer-seq1 prufer-seq2 trees-eq length-eq tree-of-prufer-seq-ctx-axioms
  proof (induction arbitrary: seq2 rule: tree-edges-of-prufer-seq-induct')
    case (1 u v)
    then show ?case by simp
  next
    case (2 verts b seq a)
    then interpret ctx1: tree-of-prufer-seq-ctx verts b # seq by simp
    interpret ctx2: tree-of-prufer-seq-ctx verts seq2 using 2 by unfold-locales blast
      obtain b' seq2' where seq2: seq2 = b' # seq2' using 2(10) by (metis
length-Suc-conv)
    then have find (λx. x ∉ set seq2) verts = Some a
     using ctx2.notin-prufer-seq-iff-leaf 2(9) 2(1) ctx1.notin-prufer-seq-iff-leaf[symmetric]
find-cong by force
     then have edges-eq: insert {a, b} (tree-edges-of-prufer-seq (remove1 a verts)
seq)
        = insert {a, b'} (tree-edges-of-prufer-seq (remove1 a verts) seq2')
      using 2 seq2 by simp
    interpret ctx1': tree-of-prufer-seq-ctx remove1 a verts seq using 2(5) .
     interpret T1: tree set (remove1 a verts) tree-edges-of-prufer-seq (remove1 a
verts) seq
      using ctx1'.tree-edges-of-prufer-seq-tree by blast
   have a ∉ set seq2' using seq2 2 ctx1.notin-prufer-seq-iff-leaf ctx2.notin-prufer-seq-iff-leaf
by auto
    then interpret ctx2': tree-of-prufer-seq-ctx remove1 a verts seq2'
      using seq2 2(8) 2(2) ctx1.distinct-verts
      by unfold-locales (auto simp: length-remove1 prufer-sequences-def)
     interpret T2: tree set (remove1 a verts) tree-edges-of-prufer-seq (remove1 a
verts) seq2'
      using ctx2'.tree-edges-of-prufer-seq-tree by blast

     have a-notin-verts': a ∉ set (remove1 a verts) using ctx1.distinct-verts by
simp
    then have ab'-notin-edges: {a,b'} ∉ tree-edges-of-prufer-seq (remove1 a verts)
seq using T1.wellformed by blast
```

```
    then have b = b′ using edges-eq by (metis doubleton-eq-iff insert-iff)

   have {a,b} ∉ tree-edges-of-prufer-seq (remove1 a verts) seq2′ using T2.wellformed
a-notin-verts′ by blast
   then have (tree-edges-of-prufer-seq (remove1 a verts) seq) = tree-edges-of-prufer-seq
(remove1 a verts) seq2′
      using edges-eq ab′-notin-edges
      by (simp add: ‹b = b′› insert-eq-iff)
     then have seq = seq2′ using 2.IH[of seq2′] ctx1′.prufer-seq ctx2′.prufer-seq
2(10) ctx1′.tree-of-prufer-seq-ctx-axioms
      unfolding seq2 by simp
    then show ?case using ‹b = b′› seq2 by simp
  qed
qed

theorem (in valid-verts) distinct-labeld-tree-enum: distinct (labeled-tree-enum verts)
  using inj-tree-edges-of-prufer-seq distinct-n-lists distinct-verts
  unfolding labeled-tree-enum-def prufer-sequences-def tree-of-prufer-seq-def
  by (auto simp add: distinct-map set-n-lists inj-on-def)

lemma (in valid-verts) cayleys-formula: card (labeled-trees (set verts)) = length
verts ^(length verts − 2)
proof−
  have card (labeled-trees (set verts)) = length (labeled-tree-enum verts)
    using distinct-labeld-tree-enum labeled-tree-enum-correct distinct-card by fast-
force
  also have ... = length verts ^(length verts − 2) unfolding labeled-tree-enum-def
using length-n-lists by auto
  finally show ?thesis .
qed

end
```

3 Rooted Trees

```
theory Rooted-Tree
imports Tree-Graph HOL−Library.FSet
begin

datatype tree = Node tree list

fun tree-size :: tree ⇒ nat where
  tree-size (Node ts) = Suc (∑ t←ts. tree-size t)

fun height :: tree ⇒ nat where
  height (Node []) = 0
| height (Node ts) = Suc (Max (height ‘ set ts))
```

Convenient case splitting and induction for trees

lemma *tree-cons-exhaust*[*case-names Nil Cons*]:
(*t* = *Node* [] ⟹ *P*) ⟹ (⋀*r ts*. *t* = *Node* (*r* # *ts*) ⟹ *P*) ⟹ *P*
by (*cases t*) (*metis list.exhaust*)

lemma *tree-rev-exhaust*[*case-names Nil Snoc*]:
(*t* = *Node* [] ⟹ *P*) ⟹ (⋀*ts r*. *t* = *Node* (*ts* @ [*r*]) ⟹ *P*) ⟹ *P*
by (*cases t*) (*metis rev-exhaust*)

lemma *tree-cons-induct*[*case-names Nil Cons*]:
assumes *P* (*Node* [])
and ⋀*t ts*. *P t* ⟹ *P* (*Node ts*) ⟹ *P* (*Node* (*t*#*ts*))
shows *P t*
proof (*induction size-tree t arbitrary*: *t rule*: *less-induct*)
case *less*
then show *?case* **using** *assms* **by** (*cases t rule*: *tree-cons-exhaust*) *auto*
qed

fun *lexord-tree* **where**
lexord-tree t (*Node* []) ⟷ *False*
| *lexord-tree* (*Node* []) *r* ⟷ *True*
| *lexord-tree* (*Node* (*t*#*ts*)) (*Node* (*r*#*rs*)) ⟷ *lexord-tree t r* ∨ (*t* = *r* ∧ *lexord-tree* (*Node ts*) (*Node rs*))

fun *mirror* :: *tree* ⇒ *tree* **where**
mirror (*Node ts*) = *Node* (*map mirror* (*rev ts*))

instantiation *tree* :: *linorder*
begin

definition
tree-less-def: (*t*::*tree*) < *r* ⟷ *lexord-tree* (*mirror t*) (*mirror r*)

definition
tree-le-def: (*t* :: *tree*) ≤ *r* ⟷ *t* < *r* ∨ *t* = *r*

lemma *lexord-tree-empty2*[*simp*]: *lexord-tree* (*Node* []) *r* ⟷ *r* ≠ *Node* []
by (*cases r rule*: *tree-cons-exhaust*) *auto*

lemma *mirror-empty*[*simp*]: *mirror t* = *Node* [] ⟷ *t* = *Node* []
by (*cases t*) *auto*

lemma *mirror-not-empty*[*simp*]: *mirror t* ≠ *Node* [] ⟷ *t* ≠ *Node* []
by (*cases t*) *auto*

lemma *tree-le-empty*[*simp*]: *Node* [] ≤ *t*
unfolding *tree-le-def tree-less-def* **using** *mirror-not-empty* **by** *auto*

lemma *tree-less-empty-iff*: *Node* [] < *t* ⟷ *t* ≠ *Node* []
unfolding *tree-less-def* **by** *simp*

lemma *not-tree-less-empty*[*simp*]: ¬ t < *Node* []
unfolding *tree-less-def* **by** *simp*

lemma *tree-le-empty2-iff* [*simp*]: $t \leq$ *Node* [] ⟷ t = *Node* []
unfolding *tree-le-def* **by** *simp*

lemma *lexord-tree-antisym*: *lexord-tree t r* ⟹ ¬ *lexord-tree r t*
by (*induction r t rule*: *lexord-tree.induct*) *auto*

lemma *tree-less-antisym*: (*t*::*tree*) < *r* ⟹ ¬ *r* < *t*
unfolding *tree-less-def* **using** *lexord-tree-antisym* **by** *blast*

lemma *lexord-tree-not-eq*: *lexord-tree t r* ⟹ $t \neq r$
by (*induction r t rule*: *lexord-tree.induct*) *auto*

lemma *tree-less-not-eq*: (*t*::*tree*) < *r* ⟹ $t \neq r$
unfolding *tree-less-def* **using** *lexord-tree-not-eq* **by** *blast*

lemma *lexord-tree-irrefl*: ¬ *lexord-tree t t*
using *lexord-tree-not-eq* **by** *blast*

lemma *tree-less-irrefl*: ¬ (*t*::*tree*) < *t*
unfolding *tree-less-def* **using** *lexord-tree-irrefl* **by** *blast*

lemma *lexord-tree-eq-iff*: ¬ *lexord-tree t r* ∧ ¬ *lexord-tree r t* ⟷ *t* = *r*
using *lexord-tree-empty2* **by** (*induction t r rule*: *lexord-tree.induct*, *fastforce*+)

lemma *mirror-mirror*: *mirror* (*mirror t*) = *t*
by (*induction t rule*: *mirror.induct*) (*simp add*: *map-idI rev-map*)

lemma *mirror-inj*: *mirror t* = *mirror r* ⟹ *t* = *r*
using *mirror-mirror* **by** *metis*

lemma *tree-less-eq-iff*: ¬ (*t*::*tree*) < *r* ∧ ¬ *r* < *t* ⟷ *t* = *r*
unfolding *tree-less-def* **using** *lexord-tree-eq-iff mirror-inj* **by** *blast*

lemma *lexord-tree-trans*: *lexord-tree t r* ⟹ *lexord-tree r s* ⟹ *lexord-tree t s*
proof (*induction t s arbitrary*: *r rule*: *lexord-tree.induct*)
case (*1 t*)
then show *?case* **by** *auto*
next
case (*2 va vb*)
then show *?case* **by** *auto*
next
case (*3 t ts s ss*)
then show *?case* **by** (*cases r rule*: *tree-cons-exhaust*) *auto*
qed

instance
proof
fix *t r s* :: *tree*
show $t < r \longleftrightarrow t \leq r \land \neg\, r \leq t$ **unfolding** *tree-le-def* **using** *tree-less-antisym tree-less-irrefl* **by** *auto*
show $t \leq t$ **unfolding** *tree-le-def* **by** *simp*
show $t \leq r \Longrightarrow r \leq t \Longrightarrow t = r$ **unfolding** *tree-le-def* **using** *tree-less-antisym* **by** *blast*
show $t \leq r \lor r \leq t$ **unfolding** *tree-le-def* **using** *tree-less-eq-iff* **by** *blast*
show $t \leq r \Longrightarrow r \leq s \Longrightarrow t \leq s$ **unfolding** *tree-le-def tree-less-def* **using** *lexord-tree-trans* **by** *blast*
qed

end

lemma *tree-size-children*: *tree-size* (*Node ts*) = *Suc n* $\Longrightarrow$ *t* $\in$ *set ts* $\Longrightarrow$ *tree-size t* $\leq$ *n*
by (*auto simp*: *le-add1 sum-list-map-remove1*)

lemma *tree-size-ge-1*: *tree-size t* $\geq$ *1*
by (*cases t*) *auto*

lemma *tree-size-ne-0*: *tree-size t* $\neq$ *0*
by (*cases t*) *auto*

lemma *tree-size-1-iff*: *tree-size t* = *1* $\longleftrightarrow$ *t* = *Node* []
using *tree-size-ne-0* **by** (*cases t rule*: *tree-cons-exhaust*) *auto*

lemma *length-children*: *tree-size* (*Node ts*) = *Suc n* $\Longrightarrow$ *length ts* $\leq$ *n*
by (*induction ts arbitrary*: *n*, *auto*, *metis add-mono plus-1-eq-Suc tree-size-ge-1*)

lemma *height-Node-cons*: *height* (*Node* (*t#ts*)) $\geq$ *Suc* (*height t*)
by *auto*

lemma *height-0-iff*: *height t* = *0* $\Longrightarrow$ *t* = *Node* []
using *height.elims* **by** *blast*

lemma *height-children*: *height* (*Node ts*) = *Suc n* $\Longrightarrow$ *t* $\in$ *set ts* $\Longrightarrow$ *height t* $\leq$ *n*
by (*metis List.finite-set Max-ge diff-Suc-1 finite-imageI height.elims imageI nat.simps*(*3*) *tree.inject*)

lemma *height-children-le-height*: $\forall$ *t* $\in$ *set ts*. *height t* $\leq$ *n* $\Longrightarrow$ *height* (*Node ts*) $\leq$ *Suc n*
by (*cases ts*) *auto*

lemma *mirror-iff*: *mirror t* = *Node ts* $\longleftrightarrow$ *t* = *Node* (*map mirror* (*rev ts*))
by (*metis mirror.simps mirror-mirror*)

lemma *mirror-append*: *mirror* (*Node* (*ts*@*rs*)) = *Node* (*map mirror* (*rev rs*) @ *map mirror* (*rev ts*))
by (*induction ts*) *auto*

lemma *lexord-tree-snoc*: *lexord-tree* (*Node ts*) (*Node* (*ts*@[*t*]))
by (*induction ts*) *auto*

lemma *tree-less-cons*: *Node ts* < *Node* (*t*#*ts*)
unfolding *tree-less-def* **using** *lexord-tree-snoc* **by** *simp*

lemma *tree-le-cons*: *Node ts* ≤ *Node* (*t*#*ts*)
unfolding *tree-le-def* **using** *tree-less-cons* **by** *simp*

lemma *tree-less-cons'*: *t* ≤ *Node rs* ⟹ *t* < *Node* (*r*#*rs*)
using *tree-less-cons* **by** (*simp add*: *order-le-less-trans*)

lemma *tree-less-snoc2-iff*[*simp*]: *Node* (*ts*@[*t*]) < *Node* (*rs*@[*r*]) ⟷ *t* < *r* ∨ (*t* = *r* ∧ *Node ts* < *Node rs*)
unfolding *tree-less-def* **using** *mirror-inj* **by** *auto*

lemma *tree-le-snoc2-iff*[*simp*]: *Node* (*ts*@[*t*]) ≤ *Node* (*rs*@[*r*]) ⟷ *t* < *r* ∨ (*t* = *r* ∧ *Node ts* ≤ *Node rs*)
unfolding *tree-le-def* **by** *auto*

lemma *lexord-tree-cons2*[*simp*]: *lexord-tree* (*Node* (*ts*@[*t*])) (*Node* (*ts*@[*r*])) ⟷ *lexord-tree t r*
by (*induction ts*) (*auto simp*: *lexord-tree-irrefl*)

lemma *tree-less-cons2*[*simp*]: *Node* (*t*#*ts*) < *Node* (*r*#*ts*) ⟷ *t* < *r*
unfolding *tree-less-def* **using** *lexord-tree-cons2* **by** *simp*

lemma *tree-le-cons2*[*simp*]: *Node* (*t*#*ts*) ≤ *Node* (*r*#*ts*) ⟷ *t* ≤ *r*
unfolding *tree-le-def* **using** *tree-less-cons2* **by** *blast*

lemma *tree-less-sorted-snoc*: *sorted* (*ts*@[*r*]) ⟹ *Node ts* < *Node* (*ts*@[*r*])
unfolding *tree-less-def* **by** (*induction ts rule*: *rev-induct*, *auto*,
metis leD lexord-tree-eq-iff sorted2 sorted-wrt-append tree-less-def,
metis dual-order.strict-iff-not list.set-intros(*2*) *nle-le sorted2 sorted-append tree-less-def*)

lemma *lexord-tree-comm-prefix*[*simp*]: *lexord-tree* (*Node* (*ss*@*ts*)) (*Node* (*ss*@*rs*)) ⟷ *lexord-tree* (*Node ts*) (*Node rs*)
using *lexord-tree-antisym* **by** (*induction ss*) *auto*

lemma *less-tree-comm-suffix*[*simp*]: *Node* (*ts*@*ss*) < *Node* (*rs*@*ss*) ⟷ *Node ts* < *Node rs*
unfolding *tree-less-def* **by** *simp*

lemma *tree-le-comm-suffix*[*simp*]: *Node* (*ts*@*ss*) ≤ *Node* (*rs*@*ss*) ⟷ *Node ts* ≤ *Node rs*
  unfolding *tree-le-def* **by** *simp*

lemma *tree-less-comm-suffix2*: *t* < *r* ⟹ *Node* (*ts*@*t*#*ss*) < *Node* (*r*#*ss*)
  unfolding *tree-less-def* **using** *lexord-tree-comm-prefix* **by** *simp*

lemma *lexord-tree-append*[*simp*]: *lexord-tree* (*Node ts*) (*Node* (*ts*@*rs*)) ⟷ *rs* ≠ []
  using *lexord-tree-irrefl* **by** (*induction ts*) *auto*

lemma *tree-less-append*[*simp*]: *Node ts* < *Node* (*rs*@*ts*) ⟷ *rs* ≠ []
  unfolding *tree-less-def* **by** *simp*

lemma *tree-le-append*: *Node ts* ≤ *Node* (*ss*@*ts*)
  unfolding *tree-le-def* **by** *simp*

lemma *tree-less-singleton-iff*[*simp*]: *Node* (*ts*@[*t*]) < *Node* [*r*] ⟷ *t* < *r*
  unfolding *tree-less-def* **by** *simp*

lemma *tree-le-singleton-iff*[*simp*]: *Node* (*ts*@[*t*]) ≤ *Node* [*r*] ⟷ *t* < *r* ∨ (*t* = *r* ∧ *ts* = [])
  unfolding *tree-le-def* **by** *auto*

lemma *lexord-tree-nested*: *lexord-tree t* (*Node* [*t*])
proof (*induction t rule*: *tree-cons-induct*)
  case *Nil*
  then show *?case* **by** *auto*
next
  case (*Cons t ts*)
  then show *?case* **by** (*cases t rule*: *tree-cons-exhaust*) *auto*
qed

lemma *tree-less-nested*: *t* < *Node* [*t*]
  unfolding *tree-less-def* **using** *lexord-tree-nested* **by** *auto*

lemma *tree-le-nested*: *t* ≤ *Node* [*t*]
  unfolding *tree-le-def* **using** *tree-less-nested* **by** *auto*

lemma *lexord-tree-iff*:
  lexord-tree t r ⟷ (∃ *ts t′ ss rs r′*. *t* = *Node* (*ss* @ *t′* # *ts*) ∧ *r* = *Node* (*ss* @ *r′* # *rs*) ∧ *lexord-tree t′ r′*) ∨ (∃ *ts rs*. *rs* ≠ [] ∧ *t* = *Node ts* ∧ *r* = *Node* (*ts* @ *rs*))
(**is** *?l* ⟷ *?r*)
proof
  show *?l* ⟹ *?r*
  proof−
    assume *lexord*: *lexord-tree t r*
    obtain *ts* **where** *ts*: *t* = *Node ts* **by** (*cases t*) *auto*
    obtain *rs* **where** *rs*: *r* = *Node rs* **by** (*cases r*) *auto*

obtain *ss ts′ rs′* **where** *prefix*: $ts = ss @ ts' \wedge rs = ss @ rs' \wedge (ts' = [] \vee rs' = [] \vee hd\ ts' \neq hd\ rs')$ **using** *longest-common-prefix* **by** *blast*
then have $ts' = [] \vee$ *lexord-tree* (*hd ts′*) (*hd rs′*) **using** *lexord* **unfolding** *ts rs*
by (*auto*, *metis lexord-tree.simps*(*1*) *lexord-tree.simps*(*3*) *list.exhaust-sel*)
then show *?thesis* **using** *prefix*
by (*metis append.right-neutral lexord lexord-tree.simps*(*1*) *lexord-tree-comm-prefix list.exhaust-sel rs ts*)
qed
show *?r* ⟹ *?l* **by** *auto*
qed

lemma *tree-less-iff*: $t < r \longleftrightarrow (\exists ts\ t'\ ss\ rs\ r'.\ t = Node\ (ts @ t' \# ss) \wedge r = Node\ (rs @ r' \# ss) \wedge t' < r') \vee (\exists ts\ rs.\ rs \neq [] \wedge t = Node\ ts \wedge r = Node\ (rs @ ts))$ (**is** *?l* ⟷ *?r*)
proof
show *?l* ⟹ *?r*
unfolding *tree-less-def* **using** *lexord-tree-iff*[*of mirror t mirror r*, *unfolded mirror-iff*]
by (*simp*, *metis append-Nil lexord-tree-eq-iff mirror-mirror*)
next
show *?r* ⟹ *?l*
by (*auto simp*: *order-le-neq-trans tree-le-append*,
meson dual-order.strict-trans1 tree-le-append tree-less-comm-suffix2)
qed

lemma *tree-empty-cons-lt-le*: $r < Node\ (Node\ [] \# ts) \Longrightarrow r \leq Node\ ts$
proof (*induction ts arbitrary*: *r rule*: *rev-induct*)
case *Nil*
then show *?case* **by** (*cases r rule*: *tree-rev-exhaust*) *auto*
next
case (*snoc x xs*)
then show *?case*
proof (*cases r rule*: *tree-rev-exhaust*)
case *Nil*
then show *?thesis* **by** *auto*
next
case (*Snoc rs r1*)
then show *?thesis* **using** *snoc* **by** (*auto*, (*metis append-Cons tree-less-snoc2-iff*)+)
qed
qed

fun *regular* :: *tree* ⇒ *bool* **where**
regular (*Node ts*) ⟷ *sorted ts* ∧ (∀ *t*∈*set ts*. *regular t*)

definition *n-trees* :: *nat* ⇒ *tree set* **where**
n-trees n = {*t*. *tree-size t* = *n*}

definition *regular-n-trees* :: *nat* ⇒ *tree set* **where**

regular-n-trees $n = \{t.$ *tree-size* $t = n \wedge$ *regular* $t\}$

3.1 Rooted Graphs

type-synonym *'a rpregraph* = (*'a set*) × (*'a edge set*) × *'a*

locale *rgraph* = *graph-system* +
  fixes *r*
  assumes *root-wf*: $r \in V$

locale *rtree* = *tree* + *rgraph*
begin

definition *subtrees* :: *'a rpregraph set* **where**
  subtrees =
    (*let* (V',E') = *remove-vertex r*
    in (λC. (C, *graph-system.induced-edges* E' C, *THE* r'. $r' \in C \wedge$ *vert-adj r r'*))
' *ulgraph.connected-components* V' E')

lemma *rtree-subtree*:
  assumes *subtree*: $(S,E_S,r_S) \in$ *subtrees*
  shows *rtree* S E_S r_S
proof−
  obtain V' E' **where** *remove-vertex*: *remove-vertex* $r = (V', E')$ **by** *fastforce*
  interpret *subg*: *ulsubgraph* V' E' V E **unfolding** *ulsubgraph-def* **using** *subgraph-remove-vertex subtree ulgraph-axioms remove-vertex* **by** *blast*
  interpret g': *fin-ulgraph* V' E'
    by (*simp add*: *fin-graph-system-axioms fin-ulgraph-def subg.is-finite-subgraph subg.is-subgraph-ulgraph ulgraph-axioms*)
  have *conn-component*: $S \in g'$*.connected-components* **using** *subtree remove-vertex* **unfolding** *subtrees-def* **by** *auto*
  then interpret *subg'*: *subgraph* S E_S V' E' **using** g'*.connected-component-subgraph subtree remove-vertex* **unfolding** *subtrees-def* **by** *auto*
  interpret *subg'*: *ulsubgraph* S E_S V' E' **by** *unfold-locales*
  interpret S: *connected-ulgraph* S E_S **using** g'*.connected-components-connected-ulgraphs conn-component subtree remove-vertex* **unfolding** *subtrees-def* **by** *auto*
  interpret S: *fin-connected-ulgraph* S E_S **using** *subg'.verts-ss* g'*.finV* **by** *unfold-locales* (*simp add*: *finite-subset*)
  interpret S: *tree* S E_S **using** *subg.is-cycle2 subg'.is-cycle2 no-cycles* **by** (*unfold-locales*, *blast*)
  show *?thesis* **using** *theI'*[*OF unique-adj-vert-removed*[*OF root-wf remove-vertex conn-component*]]
    subtree remove-vertex **by** *unfold-locales* (*auto simp*: *subtrees-def*)
qed

lemma *finite-subtrees*: *finite subtrees*
proof−
  obtain V' E' **where** *remove-vertex*: *remove-vertex* $r = (V', E')$ **by** *fastforce*
  then interpret *subg*: *subgraph* V' E' V E **using** *subgraph-remove-vertex* **by** *auto*

interpret g': *fin-ulgraph* V' E'
by (*simp add*: *fin-graph-system-axioms fin-ulgraph-def subg.is-finite-subgraph subg.is-subgraph-ulgraph ulgraph-axioms*)
show *?thesis* **using** g'*.finite-connected-components remove-vertex* **unfolding** *subtrees-def* **by** *simp*
qed

lemma *remove-root-subtrees*:
assumes *remove-vertex*: *remove-vertex* $r = (V',E')$
and *conn-component*: $C \in$ *ulgraph.connected-components* V' E'
shows *rtree* C (*graph-system.induced-edges* E' C) (*THE* r'. $r' \in C \wedge$ *vert-adj* r r')
proof−
interpret *subg*: *ulsubgraph* V' E' V E **unfolding** *ulsubgraph-def* **using** *subgraph-remove-vertex remove-vertex ulgraph-axioms* **by** *blast*
interpret g': *fin-ulgraph* V' E'
by (*simp add*: *fin-graph-system-axioms fin-ulgraph-def subg.is-finite-subgraph subg.is-subgraph-ulgraph ulgraph-axioms*)
interpret *subg'*: *ulsubgraph* C *graph-system.induced-edges* E' C V' E'
by (*simp add*: *conn-component* g'*.connected-component-subgraph* g'*.ulgraph-axioms ulsubgraph.intro*)
interpret C: *fin-connected-ulgraph* C *graph-system.induced-edges* E' C
by (*simp add*: *fin-connected-ulgraph.intro fin-ulgraph.intro* g'*.fin-graph-system-axioms* g'*.ulgraph-axioms subg'.is-finite-subgraph subg'.is-subgraph-ulgraph conn-component* g'*.connected-components-connected-ulgraphs*)
interpret C: *tree* C *graph-system.induced-edges* E' C **using** *subg.is-cycle2 subg'.is-cycle2 no-cycles* **by** (*unfold-locales*, *blast*)
show *?thesis* **using** *theI'*[*OF unique-adj-vert-removed*[*OF root-wf remove-vertex conn-component*]] **by** *unfold-locales simp*
qed

end

3.2 Rooted Graph Isomorphism

fun *app-rgraph-isomorphism* :: $('a \Rightarrow 'b) \Rightarrow 'a$ *rpregraph* $\Rightarrow 'b$ *rpregraph* **where**
app-rgraph-isomorphism f $(V,E,r) = (f$ ‘ V, ((‘) f) ‘ E, f $r)$

locale *rgraph-isomorphism* =
G: *rgraph* V_G E_G r_G + *graph-isomorphism* V_G E_G V_H E_H f **for** V_G E_G r_G V_H E_H r_H f +
assumes *root-preserving*: $f\ r_G = r_H$
begin

interpretation H: *graph-system* V_H E_H **using** *graph-system-H* **.**

lemma *rgraph-H*: *rgraph* V_H E_H r_H
using *root-preserving bij-f G.root-wf* V_H*-def* **by** *unfold-locales blast*

interpretation *H*: *rgraph* V_H E_H r_H **using** *rgraph-H* **.**

lemma *rgraph-isomorphism-inv*: *rgraph-isomorphism* V_H E_H r_H V_G E_G r_G *inv-iso*

proof−
interpret *iso*: *graph-isomorphism* V_H E_H V_G E_G *inv-iso* **using** *graph-isomorphism-inv*
.
show *?thesis* **using** *G.root-wf inj-f inv-iso-def root-preserving the-inv-into-f-f*
by *unfold-locales fastforce*
qed

end

fun *rgraph-isomorph* :: *'a rpregraph* ⇒ *'b rpregraph* ⇒ *bool* (**infix** ‹$\simeq_r$› *50*) **where**
$(V_G, E_G, r_G) \simeq_r (V_H, E_H, r_H) \longleftrightarrow (\exists f.$ *rgraph-isomorphism* V_G E_G r_G V_H E_H r_H $f)$

lemma (**in** *rgraph*) *rgraph-isomorphism-id*: *rgraph-isomorphism V E r V E r id*
using *graph-isomorphism-id rgraph-isomorphism.intro rgraph-axioms*
unfolding *rgraph-isomorphism-axioms-def* **by** *fastforce*

lemma (**in** *rgraph*) *rgraph-isomorph-refl*: $(V,E,r) \simeq_r (V,E,r)$
using *rgraph-isomorphism-id* **by** *auto*

lemma *rgraph-isomorph-sym*: $G \simeq_r H \Longrightarrow H \simeq_r G$
using *rgraph-isomorphism.rgraph-isomorphism-inv* **by** (*cases G*, *cases H*) *fastforce*

lemma *rgraph-isomorphism-trans*: *rgraph-isomorphism* V_G E_G r_G V_H E_H r_H f $\Longrightarrow$ *rgraph-isomorphism* V_H E_H r_H V_F E_F r_F g $\Longrightarrow$ *rgraph-isomorphism* V_G E_G r_G V_F E_F r_F $(g \ o \ f)$
using *graph-isomorphism-trans* **unfolding** *rgraph-isomorphism-def rgraph-isomorphism-axioms-def*
by *fastforce*

lemma *rgraph-isomorph-trans*: *transp* $(\simeq_r)$
using *rgraph-isomorphism-trans* **unfolding** *transp-def* **by** *fastforce*

lemma (**in** *rtree*) *rgraph-isomorphis-app-iso*: *inj-on f V* $\Longrightarrow$ *app-rgraph-isomorphism* f $(V,E,r) = (V',E',r')$ $\Longrightarrow$ *rgraph-isomorphism* V E r V' E' r' f
by *unfold-locales* (*auto simp*: *bij-betw-def*)

lemma (**in** *rtree*) *rgraph-isomorph-app-iso*: *inj-on f V* $\Longrightarrow (V, E, r) \simeq_r$ *app-rgraph-isomorphism* f (V, E, r)
using *rgraph-isomorphis-app-iso* **by** *fastforce*

3.3 Conversion between unlabeled, ordered, rooted trees and tree graphs

datatype *'a ltree* = *LNode 'a 'a ltree list*

fun *ltree-size* :: *'a ltree* ⇒ *nat* **where**
ltree-size (*LNode r ts*) = *Suc* (∑ *t*←*ts*. *ltree-size t*)

fun *root-ltree* :: *'a ltree* ⇒ *'a* **where**
root-ltree (*LNode r ts*) = *r*

fun *nodes-ltree* :: *'a ltree* ⇒ *'a set* **where**
nodes-ltree (*LNode r ts*) = {*r*} ∪ (⋃ *t*∈*set ts*. *nodes-ltree t*)

fun *relabel-ltree* :: (*'a* ⇒ *'b*) ⇒ *'a ltree* ⇒ *'b ltree* **where**
relabel-ltree f (*LNode r ts*) = *LNode* (*f r*) (*map* (*relabel-ltree f*) *ts*)

fun *distinct-ltree-nodes* :: *'a ltree* ⇒ *bool* **where**
distinct-ltree-nodes (*LNode a ts*) ⟷ (∀ *t*∈*set ts*. *a* ∉ *nodes-ltree t*) ∧ *distinct ts* ∧ *disjoint-family-on nodes-ltree* (*set ts*) ∧ (∀ *t*∈*set ts*. *distinct-ltree-nodes t*)

fun *postorder-label-aux* :: *nat* ⇒ *tree* ⇒ *nat* × *nat ltree* **where**
postorder-label-aux n (*Node* []) = (*n*, *LNode n* [])
| *postorder-label-aux n* (*Node* (*t*#*ts*)) =
(*let* (*n'*, *t'*) = *postorder-label-aux n t in*
case postorder-label-aux (*Suc n'*) (*Node ts*) *of*
(*n''*, *LNode r ts'*) ⇒ (*n''*, *LNode r* (*t'*#*ts'*)))

definition *postorder-label* :: *tree* ⇒ *nat ltree* **where**
postorder-label t = *snd* (*postorder-label-aux 0 t*)

fun *tree-ltree* :: *'a ltree* ⇒ *tree* **where**
tree-ltree (*LNode r ts*) = *Node* (*map tree-ltree ts*)

fun *regular-ltree* :: *'a ltree* ⇒ *bool* **where**
regular-ltree (*LNode r ts*) ⟷ *sorted-wrt* (λ*t s*. *tree-ltree t* ≤ *tree-ltree s*) *ts* ∧ (∀ *t*∈*set ts*. *regular-ltree t*)

datatype *'a stree* = *SNode 'a 'a stree fset*

lemma *stree-size-child-lt*[*termination-simp*]: *t* |∈| *ts* ⟹ *size t* < *Suc* (∑ *s*∈*fset ts*. *Suc* (*size s*))
using *sum-nonneg-leq-bound zero-le finite-fset Suc-le-eq less-SucI* **by** *metis*

lemma *stree-size-child-lt'*[*termination-simp*]: *t* ∈ *fset ts* ⟹ *size t* < *Suc* (∑ *s*∈*fset ts*. *Suc* (*size s*))
using *stree-size-child-lt* **by** *metis*

fun *stree-size* :: *'a stree* ⇒ *nat* **where**
stree-size (*SNode r ts*) = *Suc* (*fsum stree-size ts*)

definition *n-strees* :: *nat* ⇒ *'a stree set* **where**
n-strees n = {*t*. *stree-size t* = *n*}

fun *root-stree* :: $'a$ *stree* $\Rightarrow$ $'a$ **where**
root-stree (*SNode a ts*) = *a*

fun *nodes-stree* :: $'a$ *stree* $\Rightarrow$ $'a$ *set* **where**
nodes-stree (*SNode a ts*) = $\{a\} \cup (\bigcup t{\in}$*fset ts*. *nodes-stree t*)

fun *tree-graph-edges* :: $'a$ *stree* $\Rightarrow$ $'a$ *edge set* **where**
tree-graph-edges (*SNode a ts*) = ((λt. {*a*, *root-stree t*}) ‘ *fset ts*) $\cup$ ($\bigcup t{\in}$*fset ts*. *tree-graph-edges t*)

fun *distinct-stree-nodes* :: $'a$ *stree* $\Rightarrow$ *bool* **where**
distinct-stree-nodes (*SNode a ts*) $\longleftrightarrow$ ($\forall t{\in}$*fset ts*. *a* $\notin$ *nodes-stree t*) $\wedge$ *disjoint-family-on nodes-stree* (*fset ts*) $\wedge$ ($\forall t{\in}$*fset ts*. *distinct-stree-nodes t*)

fun *ltree-stree* :: $'a$ *stree* $\Rightarrow$ $'a$ *ltree* **where**
ltree-stree (*SNode r ts*) = *LNode r* (*SOME xs*. *fset-of-list xs* = *ltree-stree* |‘| *ts* $\wedge$ *distinct xs* $\wedge$ *sorted-wrt* (λt *s*. *tree-ltree t* $\leq$ *tree-ltree s*) *xs*)

fun *stree-ltree* :: $'a$ *ltree* $\Rightarrow$ $'a$ *stree* **where**
stree-ltree (*LNode r ts*) = *SNode r* (*fset-of-list* (*map stree-ltree ts*))

definition *tree-graph-stree* :: $'a$ *stree* $\Rightarrow$ $'a$ *rpregraph* **where**
tree-graph-stree t = (*nodes-stree t*, *tree-graph-edges t*, *root-stree t*)

function *stree-of-graph* :: $'a$ *rpregraph* $\Rightarrow$ $'a$ *stree* **where**
stree-of-graph (*V*,*E*,*r*) =
(*if* $\neg$ *rtree V E r then undefined else*
SNode r (*Abs-fset* (*stree-of-graph* ‘ *rtree.subtrees V E r*)))
by *pat-completeness auto*

termination
proof (*relation measure* (λp. *card* (*fst p*)), *auto*)
fix *r* :: $'a$ **and** *V* :: $'a$ *set* **and** *E* :: $'a$ *edge set* **and** *S* :: $'a$ *set* **and** E_S :: $'a$ *edge set* **and** r_S :: $'a$
assume *rtree*: *rtree V E r*
assume *subtree*: (S, E_S, r_S) $\in$ *rtree.subtrees V E r*
interpret *rtree V E r* **using** *rtree* **.**
obtain V' E' **where** *remove-vertex*: *remove-vertex r* = (V', E') **by** *fastforce*
then interpret *subg*: *subgraph* V' E' *V E* **using** *subgraph-remove-vertex* **by** *simp*
interpret g': *fin-ulgraph* V' E' **using** *fin-ulgraph.intro subg.is-finite-subgraph fin-graph-system-axioms subg.is-subgraph-ulgraph ulgraph-axioms* **by** *blast*
have $S \in g'$*.connected-components* **using** *subtree remove-vertex* **unfolding** *subtrees-def* **by** *auto*
then have *card-C-*V':*card S* $\leq$ *card* V' **using** g'*.connected-component-wf* g'*.finV card-mono* **by** *metis*
have *card* V' < *card V* **using** *remove-vertex root-wf finV card-Diff1-less* **unfolding** *remove-vertex-def* **by** *fast*

then show *card S* < *card V* **using** *card-C-V′* **by** *simp*
qed

definition *tree-graph* :: *tree* ⇒ *nat rpregraph* **where**
tree-graph t = *tree-graph-stree* (*stree-ltree* (*postorder-label t*))

fun *relabel-stree* :: (′*a* ⇒ ′*b*) ⇒ ′*a stree* ⇒ ′*b stree* **where**
relabel-stree f (*SNode r ts*) = *SNode* (*f r*) ((*relabel-stree f*) |‘| *ts*)

lemma *root-ltree-wf*: *root-ltree t* ∈ *nodes-ltree t*
by (*cases t*) *auto*

lemma *root-relabel-ltree*[*simp*]: *root-ltree* (*relabel-ltree f t*) = *f* (*root-ltree t*)
by (*cases t*) *simp*

lemma *nodes-relabel-ltree*[*simp*]: *nodes-ltree* (*relabel-ltree f t*) = *f* ‘ *nodes-ltree t*
by (*induction t*) *auto*

lemma *finite-nodes-ltree*: *finite* (*nodes-ltree t*)
by (*induction t*) *auto*

lemma *root-stree-wf*: *root-stree t* ∈ *nodes-stree t*
by (*cases t*) *auto*

lemma *tree-graph-edges-wf*: *e* ∈ *tree-graph-edges t* ⟹ *e* ⊆ *nodes-stree t*
using *root-stree-wf* **by** (*induction t rule*: *tree-graph-edges.induct*) *auto*

lemma *card-tree-graph-edges-distinct*: *distinct-stree-nodes t* ⟹ *e* ∈ *tree-graph-edges t* ⟹ *card e* = *2*
using *root-stree-wf card-2-iff* **by** (*induction t rule*: *tree-graph-edges.induct*) (*auto*, *fast*+)

lemma *nodes-stree-non-empty*: *nodes-stree t* ≠ {}
by (*cases t rule*: *nodes-stree.cases*) *auto*

lemma *finite-nodes-stree*: *finite* (*nodes-stree t*)
by (*induction t rule*: *nodes-stree.induct*) *auto*

lemma *finite-tree-graph-edges*: *finite* (*tree-graph-edges t*)
by (*induction t rule*: *tree-graph-edges.induct*) *auto*

lemma *root-relabel-stree*[*simp*]: *root-stree* (*relabel-stree f t*) = *f* (*root-stree t*)
by (*cases t*) *auto*

lemma *nodes-stree-relabel-stree*[*simp*]: *nodes-stree* (*relabel-stree f t*) = *f* ‘ *nodes-stree t*
by (*induction t*) *auto*

lemma *tree-graph-edges-relabel-stree*[*simp*]: *tree-graph-edges* (*relabel-stree f t*) =

((‘) *f*) ‘ *tree-graph-edges t*
by (*induction t*) (*simp add*: *image-image image-Un image-Union*)

lemma *nodes-stree-ltree*[*simp*]: *nodes-stree* (*stree-ltree t*) = *nodes-ltree t*
by (*induction t*) (*auto simp*: *fset-of-list.rep-eq*)

lemma *distinct-sorted-wrt-list*: ∃ *xs*. *fset-of-list xs* = *A* ∧ *distinct xs* ∧ *sorted-wrt* (λ*t s*. (*f t* :: ′*b*::*linorder*) ≤ *f s*) *xs*
proof−
obtain *xs* **where** *fset-of-list xs* = *A* ∧ *distinct xs*
by (*metis finite-distinct-list finite-fset fset-cong fset-of-list.rep-eq*)
then have *fset-of-list* (*sort-key f xs*) = *A* ∧ *distinct* (*sort-key f xs*) ∧ *sorted-wrt* (λ*t s*. *f t* ≤ *f s*) (*sort-key f xs*)
using *sorted-sort-key sorted-wrt-map* **by** (*simp add*: *fset-of-list.abs-eq*, *blast*)
then show *?thesis* **by** *blast*
qed

abbreviation *ltree-stree-subtrees ts* ≡ *SOME xs*. *fset-of-list xs* = *ltree-stree* |‘| *ts* ∧ *distinct xs* ∧ *sorted-wrt* (λ*t s*. *tree-ltree t* ≤ *tree-ltree s*) *xs*

lemma *fset-of-list-ltree-stree-subtrees*[*simp*]: *fset-of-list* (*ltree-stree-subtrees ts*) = *ltree-stree* |‘| *ts*
using *someI-ex*[*OF distinct-sorted-wrt-list*] **by** *fast*

lemma *set-ltree-stree-subtrees*[*simp*]: *set* (*ltree-stree-subtrees ts*) = *ltree-stree* ‘ *fset ts*
using *fset-of-list-ltree-stree-subtrees* **by** (*metis* (*mono-tags*, *lifting*) *fset.set-map fset-of-list.rep-eq*)

lemma *distinct-ltree-stree-subtrees*: *distinct* (*ltree-stree-subtrees ts*)
using *someI-ex*[*OF distinct-sorted-wrt-list*] **by** *blast*

lemma *sorted-wrt-ltree-stree-subtrees*: *sorted-wrt* (λ*t s*. *tree-ltree t* ≤ *tree-ltree s*) (*ltree-stree-subtrees ts*)
using *someI-ex*[*OF distinct-sorted-wrt-list*] **by** *blast*

lemma *nodes-ltree-stree*[*simp*]: *nodes-ltree* (*ltree-stree t*) = *nodes-stree t*
by (*induction t*) *auto*

lemma *stree-ltree-stree*[*simp*]: *stree-ltree* (*ltree-stree t*) = *t*
by (*induction t*) (*simp add*: *fset.map-ident-strong*)

lemma *nodes-tree-graph-stree*: *tree-graph-stree t* = (*V*, *E*, *r*) ⟹ *V* = *nodes-stree t*
by (*induction t*) (*simp add*: *tree-graph-stree-def*)

lemma *relabel-stree-stree-ltree*: *relabel-stree f* (*stree-ltree t*) = *stree-ltree* (*relabel-ltree f t*)
by (*induction t*) (*auto simp add*: *fset-of-list-elem*)

lemma *relabel-stree-relabel-ltree*: *relabel-ltree f t1* = *t2* $\Longrightarrow$ *relabel-stree f* (*stree-ltree t1*) = *stree-ltree t2*
using *relabel-stree-stree-ltree* **by** *blast*

lemma *app-rgraph-iso-tree-graph-stree*: *app-rgraph-isomorphism f* (*tree-graph-stree t*) = *tree-graph-stree* (*relabel-stree f t*)
unfolding *tree-graph-stree-def* **using** *image-iff mk-disjoint-insert*
by (*induction t*) (*auto*, *fastforce*+)

lemma (**in** *rtree*) *root-stree-of-graph*[*simp*]: *root-stree* (*stree-of-graph* (*V*,*E*,*r*)) = *r*
using *rtree-axioms* **by** (*simp split*: *prod.split*)

lemma (**in** *rtree*) *nodes-stree-stree-of-graph*[*simp*]: *nodes-stree* (*stree-of-graph* (*V*,*E*,*r*)) = *V*
using *rtree-axioms*
proof (*induction* (*V*,*E*,*r*) *arbitrary*: *V E r rule*: *stree-of-graph.induct*)
case (*1* V_T E_T *r*)
then interpret *t*: *rtree* V_T E_T *r* **by** *simp*
obtain V' E' **where** VE': *t.remove-vertex r* = (V', E') **by** (*simp add*: *t.remove-vertex-def*)
interpret *subg*: *subgraph* V' E' V_T E_T **using** *t.subgraph-remove-vertex* VE' **by** *metis*
interpret g': *fin-ulgraph* V' E' **using** *fin-ulgraph.intro subg.is-finite-subgraph t.fin-graph-system-axioms subg.is-subgraph-ulgraph t.ulgraph-axioms* **by** *blast*

have *finite* (*stree-of-graph* ' *t.subtrees*) **using** *t.finite-subtrees* **by** *blast*
then have *nodes-stree* (*stree-of-graph* (V_T, E_T, *r*)) = $\{r\} \cup V'$
using *1* **using** VE' *t.rtree-subtree* g'*.Union-connected-components* **by** (*simp add*: *Abs-fset-inverse t.subtrees-def*)
then show *?case* **using** VE' *t.root-wf* **unfolding** *t.remove-vertex-def* **by** *auto*
qed

lemma (**in** *rtree*) *tree-graph-edges-stree-of-graph*[*simp*]: *tree-graph-edges* (*stree-of-graph* (*V*,*E*,*r*)) = *E*
using *rtree-axioms*
proof (*induction* (*V*,*E*,*r*) *arbitrary*: *V E r rule*: *stree-of-graph.induct*)
case (*1* V_T E_T *r*)
then interpret *t*: *rtree* V_T E_T *r* **by** *simp*
obtain V' E' **where** VE': *t.remove-vertex r* = (V', E') **by** (*simp add*: *t.remove-vertex-def*)
interpret *subg*: *subgraph* V' E' V_T E_T **using** *t.subgraph-remove-vertex* VE' **by** *metis*
interpret g': *fin-ulgraph* V' E' **using** *fin-ulgraph.intro subg.is-finite-subgraph t.fin-graph-system-axioms subg.is-subgraph-ulgraph t.ulgraph-axioms* **by** *blast*

have *finite* (*stree-of-graph* ' *t.subtrees*) **using** *t.finite-subtrees* **by** *blast*
then have *fset-Abs-fset-subtrees*[*simp*]: *fset* (*Abs-fset* (*stree-of-graph* ' *t.subtrees*)) = *stree-of-graph* ' *t.subtrees* **by** (*simp add*: *Abs-fset-inverse*)

have *root-edges*: (λx. {*r*, *root-stree x*}) ' *stree-of-graph* ' *t.subtrees* = {*e*∈E_T. *r* ∈ *e*} (**is** *?l* = *?r*)
proof−
have *e* ∈ *?l* **if** *e* ∈ *?r* **for** *e*
proof−
obtain *r′* **where** *e*: *e* = {*r*, *r′*} **using** ‹*e*∈*?r*›
by (*metis* (*no-types*, *lifting*) *CollectD insert-commute insert-iff singleton-iff t.obtain-edge-pair-adj*)
then have *r′* ≠ *r* **using** *t.singleton-not-edge* ‹*e*∈*?r*› **by** *force*
then have *r′* ∈ *V′* **using** *e* ‹*e*∈*?r*› *VE′ t.remove-vertex-def t.wellformed-alt-snd* **by** *fastforce*
then obtain *C* **where** *C-conn-component*: *C* ∈ *g′.connected-components* **and** *r′* ∈ *C* **using** *g′.Union-connected-components* **by** *auto*
have *t.vert-adj r r′* **unfolding** *t.vert-adj-def* **using** ‹*e*∈*?r*› *e* **by** *blast*
then have (*THE r′*. *r′* ∈ *C* ∧ *t.vert-adj r r′*) = *r′* **using** *t.unique-adj-vert-removed*[*OF t.root-wf VE′ C-conn-component*] ‹*r′*∈*C*› **by** *auto*
then show *?thesis* **using** *e* ‹*r′*∈*C*› *C-conn-component rtree.root-stree-of-graph t.rtree-subtree VE′* **unfolding** *t.subtrees-def* **by** (*auto simp*: *image-comp*)
qed
then show *?thesis* **using** *t.unique-adj-vert-removed*[*OF t.root-wf VE′*] *t.rtree-subtree VE′*
unfolding *t.subtrees-def t.vert-adj-def* **by** (*auto*, *metis* (*no-types*, *lifting*) *theI*)
qed
have (⋃*S*∈*t.subtrees*. *tree-graph-edges* (*stree-of-graph S*)) = *E′*
using *1 VE′ t.rtree-subtree g′.Union-induced-edges-connected-components*
unfolding *t.subtrees-def* **by** *simp*
then have *tree-graph-edges* (*stree-of-graph* (V_T,E_T,*r*)) = {*e*∈E_T. *r* ∈ *e*} ∪ *E′*
using *root-edges 1*(*2*) **by** *simp*
then show *?case* **using** *VE′* **unfolding** *t.remove-vertex-def t.vincident-def* **by** *blast*
qed

lemma (**in** *rtree*) *tree-graph-stree-of-graph*[*simp*]: *tree-graph-stree* (*stree-of-graph* (*V*,*E*,*r*)) = (*V*,*E*,*r*)
using *nodes-stree-stree-of-graph tree-graph-edges-stree-of-graph root-stree-of-graph* **unfolding** *tree-graph-stree-def* **by** *blast*

lemma *postorder-label-aux-mono*: *fst* (*postorder-label-aux n t*) ≥ *n*
by (*induction n t rule*: *postorder-label-aux.induct*) (*auto split*: *prod.split ltree.split*, *fastforce*)

lemma *nodes-postorder-label-aux-ge*: *postorder-label-aux n t* = (*n′*, *t′*) ⟹ *v* ∈ *nodes-ltree t′* ⟹ *v* ≥ *n*
by (*induction n t arbitrary*: *n′ t′ rule*: *postorder-label-aux.induct*,
auto split: *prod.splits ltree.splits*,
(*metis fst-conv le-SucI order.trans postorder-label-aux-mono*)+)

lemma *nodes-postorder-label-aux-le*: *postorder-label-aux n t* = (*n′*, *t′*) ⟹ *v* ∈

nodes-ltree t′ ⟹ *v* ≤ *n′*
by (*induction n t arbitrary*: *n′ t′ rule*: *postorder-label-aux.induct*,
auto split: *prod.splits ltree.splits*,
metis Suc-leD fst-conv order-trans postorder-label-aux-mono,
blast)

lemma *distinct-nodes-postorder-label-aux*: *distinct-ltree-nodes* (*snd* (*postorder-label-aux n t*))
proof (*induction n t rule*: *postorder-label-aux.induct*)
case (*1 n*)
then show *?case* **by** (*simp add*: *disjoint-family-on-def*)
next
case (*2 n t ts*)
obtain *n′ t′* **where** *t′*: *postorder-label-aux n t* = (*n′*, *t′*) **by** *fastforce*
obtain *n″ r ts′* **where** *ts′*: *postorder-label-aux* (*Suc n′*) (*Node ts*) = (*n″*, *LNode r ts′*) **by** (*metis eq-snd-iff ltree.exhaust*)
then have *r* ≥ *Suc n′* **using** *nodes-postorder-label-aux-ge* **by** *auto*
then have *r-notin-t′*: *r* ∉ *nodes-ltree t′* **using** *nodes-postorder-label-aux-le*[*OF t′*] **by** *fastforce*
have *distinct-subtrees*: *distinct* (*t′#ts′*) **using** *2 t′ ts′ nodes-postorder-label-aux-le*[*OF t′*]
nodes-postorder-label-aux-ge[*OF ts′*] **by** (*auto*, *meson not-less-eq-eq root-ltree-wf*)
have *disjoint-family-on nodes-ltree* (*set* (*t′#ts′*)) **using** *2 t′ ts′ nodes-postorder-label-aux-le*[*OF t′*]
nodes-postorder-label-aux-ge[*OF ts′*] **by** (*simp add*: *disjoint-family-on-def*, *meson disjoint-iff not-less-eq-eq*)
then show *?case* **using** *2 t′ ts′ r-notin-t′ distinct-subtrees* **by** *simp*
qed

lemma *distinct-nodes-postorder-label*: *distinct-ltree-nodes* (*postorder-label t*)
unfolding *postorder-label-def* **using** *distinct-nodes-postorder-label-aux* **by** *simp*

lemma *distinct-nodes-stree-ltree*: *distinct-ltree-nodes t* ⟹ *distinct-stree-nodes* (*stree-ltree t*)
by (*induction t*) (*auto simp*: *fset-of-list.rep-eq disjoint-family-on-def*, *fast*)

fun *distinct-edges* :: *′a stree* ⇒ *bool* **where**
distinct-edges (*SNode a ts*) ⟷ *inj-on* (λ*t*. {*a*, *root-stree t*}) (*fset ts*)
∧ (∀ *t*∈*fset ts*. *disjnt* ((λ*t*. {*a*, *root-stree t*}) ‘ *fset ts*) (*tree-graph-edges t*))
∧ *disjoint-family-on tree-graph-edges* (*fset ts*)
∧ (∀ *t*∈*fset ts*. *distinct-edges t*)

lemma *distinct-nodes-inj-on-root-stree*: *distinct-stree-nodes* (*SNode r ts*) ⟹ *inj-on root-stree* (*fset ts*)
by (*auto simp*: *disjoint-family-on-def*, *metis IntI emptyE inj-onI root-stree-wf*)

lemma *distinct-nodes-disjoint-edges*:
assumes *distinct-nodes*: *distinct-stree-nodes* (*SNode a ts*)
shows *disjoint-family-on tree-graph-edges* (*fset ts*)

proof −
  have *tree-graph-edges t1* ∩ *tree-graph-edges t2* = {}
    if *t1-in-ts*: *t1* ∈ *fset ts* **and** *t2-in-ts*: *t2* ∈ *fset ts* **and** *t1* ≠ *t2* **for** *t1 t2*
  proof −
    have ∀ *e*∈*tree-graph-edges t1*. *e* ∉ *tree-graph-edges t2*
    proof
      fix *e* **assume** *e-in-edges-t1*: *e* ∈ *tree-graph-edges t1*
      then have *e* ≠ {} **using** *t1-in-ts card-tree-graph-edges-distinct distinct-nodes* **by** *fastforce*
      then have ∃ *v*∈*nodes-stree t1*. *v* ∈ *e* **using** *tree-graph-edges-wf e-in-edges-t1* **by** *blast*
      then show *e* ∉ *tree-graph-edges t2* **using** ‹*t1*≠*t2*› *distinct-nodes t1-in-ts t2-in-ts tree-graph-edges-wf*
        by (*auto simp*: *disjoint-family-on-def*, *blast*)
    qed
    then show *?thesis* **by** *blast*
  qed
  then show *?thesis* **unfolding** *disjoint-family-on-def* **by** *blast*
qed

lemma *card-nodes-edges*: *distinct-stree-nodes t* ⟹ *card* (*nodes-stree t*) = *Suc* (*card* (*tree-graph-edges t*))
proof (*induction t rule*: *tree-graph-edges.induct*)
  case (*1 a ts*)
  let *?t* = *SNode a ts*
  have *inj-on* (λ*t*. {*a*, *root-stree t*}) (*fset ts*) **using** *distinct-nodes-inj-on-root-stree*[*OF 1*(*2*)]
    unfolding *inj-on-def doubleton-eq-iff* **by** *blast*
  then have *card-root-edges*: *card* ((λ*t*. {*a*, *root-stree t*}) ‘ *fset ts*) = *card* (*fset ts*)
    using *card-image* **by** *blast*
  have *finite-Un*: *finite* (⋃ *t*∈*fset ts*. *nodes-stree t*) **using** *finite-Union finite-nodes-stree finite-fset* **by** *auto*
  then have *card* (*nodes-stree ?t*) = *Suc* (*card* (⋃ *t*∈*fset ts*. *nodes-stree t*)) **using** *1*(*2*) *card-insert-disjoint finite-Un* **by** *simp*
  also have ... = *Suc* (∑ *t*∈*fset ts*. *card* (*nodes-stree t*)) **using** *1*(*2*) *card-UN-disjoint′ finite-nodes-stree finite-fset* **by** *fastforce*
  also have ... = *Suc* (∑ *t*∈*fset ts*. *Suc* (*card* (*tree-graph-edges t*))) **using** *1* **by** *simp*
  also have ... = *Suc* (*card* (*fset ts*) + (∑ *t*∈*fset ts*. *card* (*tree-graph-edges t*))) **by** (*metis add.commute sum-Suc*)
  also have ... = *Suc* (*card* ((λ*t*. {*a*, *root-stree t*}) ‘ *fset ts*) + (∑ *t*∈*fset ts*. *card* (*tree-graph-edges t*)))
    using *card-root-edges* **by** *simp*
  also have ... = *Suc* (*card* ((λ*x*. {*a*, *root-stree x*}) ‘ *fset ts*) + *card* (⋃ (*tree-graph-edges* ‘ *fset ts*)))
    using *distinct-nodes-disjoint-edges*[*OF 1*(*2*)] *card-UN-disjoint′ finite-tree-graph-edges* **by** *fastforce*
  also have ... = *Suc* (*card* ((λ*x*. {*a*, *root-stree x*}) ‘ *fset ts* ∪ (⋃ (*tree-graph-edges* ‘ *fset ts*)))) (**is** *Suc* (*card ?r* + *card ?Un*) = *Suc* (*card* (*?r* ∪ *?Un*)))

proof−
have ∀ *t* ∈ *fset ts*. ∀ *e* ∈ *tree-graph-edges t*. *a* ∉ *e* **using** *1*(*2*) *tree-graph-edges-wf* **by** *auto*
then have *disjnt*: *disjnt ?r ?Un* **using** *disjoint-UN-iff* **by** (*auto simp*: *disjnt-def*)
show *?thesis* **using** *card-Un-disjnt*[*OF - - disjnt*] *finite-tree-graph-edges* **by** *fastforce*
qed
finally show *?case* **by** *simp*
qed

lemma *tree-tree-graph-edges*: *distinct-stree-nodes t* ⟹ *tree* (*nodes-stree t*) (*tree-graph-edges t*)
proof (*induction t rule*: *tree-graph-edges.induct*)
case (*1 a ts*)
let *?t* = *SNode a ts*
have ⋀*e*. *e* ∈ *tree-graph-edges ?t* ⟹ *0* < *card e* ∧ *card e* ≤ *2* **using** *card-tree-graph-edges-distinct 1* **by** (*metis order-refl pos2*)
then interpret *g*: *fin-ulgraph nodes-stree ?t tree-graph-edges ?t* **using** *tree-graph-edges-wf finite-nodes-stree* **by** (*unfold-locales*) *blast+*
have *g.vert-connected a v* **if** *t*: *t* ∈ *fset ts* **and** *v*: *v* ∈ *nodes-stree t* **for** *t v*
proof−
interpret *t*: *tree nodes-stree t tree-graph-edges t* **using** *1 t* **by** *auto*
interpret *subg*: *ulsubgraph nodes-stree t tree-graph-edges t nodes-stree ?t tree-graph-edges ?t* **using** *t* **by** *unfold-locales auto*
have *conn-root-v*: *g.vert-connected* (*root-stree t*) *v* **using** *subg.vert-connected v root-stree-wf t.vertices-connected* **by** *blast*
have {*a*, *root-stree t*} ∈ *tree-graph-edges ?t* **using** *t* **by** *auto*
then have *g.vert-connected a* (*root-stree t*) **using** *g.vert-connected-neighbors* **by** *blast*
then show *?thesis* **using** *conn-root-v g.vert-connected-trans* **by** *blast*
qed
then have ∀ *v*∈*nodes-stree ?t*. *g.vert-connected a v* **using** *g.vert-connected-id* **by** *auto*
then have *g.is-connected-set* (*nodes-stree ?t*) **using** *g.vert-connected-trans g.vert-connected-rev* **unfolding** *g.is-connected-set-def* **by** *blast*
then interpret *g*: *fin-connected-ulgraph nodes-stree ?t tree-graph-edges ?t* **by** *unfold-locales auto*
show *?case* **using** *card-E-treeI card-nodes-edges 1*(*2*) *g.fin-connected-ulgraph-axioms* **by** *blast*
qed

lemma *rtree-tree-graph-edges*:
assumes *distinct-nodes*: *distinct-stree-nodes t*
shows *rtree* (*nodes-stree t*) (*tree-graph-edges t*) (*root-stree t*)
proof−
interpret *tree nodes-stree t tree-graph-edges t* **using** *distinct-nodes tree-tree-graph-edges* **by** *blast*
show *?thesis* **using** *root-stree-wf* **by** *unfold-locales blast*
qed

lemma *rtree-tree-graph-stree*: *distinct-stree-nodes t* $\Longrightarrow$ *tree-graph-stree t* = (*V*,*E*,*r*) $\Longrightarrow$ *rtree V E r*
using *rtree-tree-graph-edges* **unfolding** *tree-graph-stree-def* **by** *blast*

lemma *rtree-tree-graph*: *tree-graph t* = (*V*,*E*,*r*) $\Longrightarrow$ *rtree V E r*
unfolding *tree-graph-def* **using** *distinct-nodes-postorder-label rtree-tree-graph-stree distinct-nodes-stree-ltree* **by** *fast*

Cardinality of the resulting rooted tree is correct

lemma *ltree-size-postorder-label-aux*: *ltree-size* (*snd* (*postorder-label-aux n t*)) = *tree-size t*
by (*induction n t rule*: *postorder-label-aux.induct*) (*auto split*: *prod.split ltree.split*)

lemma *ltree-size-postorder-label*: *ltree-size* (*postorder-label t*) = *tree-size t*
unfolding *postorder-label-def* **using** *ltree-size-postorder-label-aux* **by** *blast*

lemma *distinct-nodes-ltree-size-card-nodes*: *distinct-ltree-nodes t* $\Longrightarrow$ *ltree-size t* = *card* (*nodes-ltree t*)
proof (*induction t*)
case (*LNode r ts*)
have *finite* ($\bigcup$ (*nodes-ltree* ' *set ts*)) **using** *finite-nodes-ltree* **by** *blast*
then show *?case* **using** *LNode disjoint-family-on-disjoint-image*
by (*auto simp*: *sum-list-distinct-conv-sum-set card-UN-disjoint'*)
qed

lemma *distinct-nodes-stree-size-card-nodes*: *distinct-stree-nodes t* $\Longrightarrow$ *stree-size t* = *card* (*nodes-stree t*)
proof (*induction t*)
case (*SNode r ts*)
have *finite* ($\bigcup$ (*nodes-stree* ' *fset ts*)) **using** *finite-nodes-stree* **by** *auto*
then show *?case* **using** *SNode disjoint-family-on-disjoint-image*
by (*auto simp*: *fsum.F.rep-eq card-UN-disjoint'*)
qed

lemma *stree-size-stree-ltree*: *distinct-ltree-nodes t* $\Longrightarrow$ *stree-size* (*stree-ltree t*) = *ltree-size t*
by (*simp add*: *distinct-nodes-ltree-size-card-nodes distinct-nodes-stree-ltree distinct-nodes-stree-size-card-nodes*)

lemma *card-tree-graph-stree*: *distinct-stree-nodes t* $\Longrightarrow$ *tree-graph-stree t* = (*V*,*E*,*r*) $\Longrightarrow$ *card V* = *stree-size t*
by (*simp add*: *distinct-nodes-stree-size-card-nodes*) (*metis nodes-tree-graph-stree*)

lemma *card-tree-graph*: *tree-graph t* = (*V*,*E*,*r*) $\Longrightarrow$ *card V* = *tree-size t*
unfolding *tree-graph-def* **using** *ltree-size-postorder-label stree-size-stree-ltree card-tree-graph-stree*
by (*metis distinct-nodes-postorder-label distinct-nodes-stree-ltree*)

lemma [*termination-simp*]: (*t*, *s*) ∈ *set* (*zip ts ss*) ⟹ *size t* < *Suc* (*size-list size ts*)
by (*metis less-not-refl not-less-eq set-zip-leftD size-list-estimation*)

fun *obtain-ltree-isomorphism* :: *'a ltree* ⇒ *'b ltree* ⇒ (*'a* ⇀ *'b*) **where**
obtain-ltree-isomorphism (*LNode r1 ts*) (*LNode r2 ss*) = *fold* (++) (*map2 obtain-ltree-isomorphism ts ss*) [*r1*↦*r2*]

fun *postorder-relabel-aux* :: *nat* ⇒ *'a ltree* ⇒ *nat* × (*nat* ⇀ *'a*) **where**
postorder-relabel-aux n (*LNode r* []) = (*n*, [*n* ↦ *r*])
| *postorder-relabel-aux n* (*LNode r* (*t*#*ts*)) =
(*let* (*n'*, f_t) = *postorder-relabel-aux n t*;
(*n''*, f_{ts}) = *postorder-relabel-aux* (*Suc n'*) (*LNode r ts*) *in*
(*n''*, f_t ++ f_{ts}))

definition *postorder-relabel* :: *'a ltree* ⇒ (*nat* ⇀ *'a*) **where**
postorder-relabel t = *snd* (*postorder-relabel-aux 0 t*)

lemma *fst-postorder-label-aux-tree-ltree*: *fst* (*postorder-label-aux n* (*tree-ltree t*)) = *fst* (*postorder-relabel-aux n t*)
by (*induction n t rule*: *postorder-relabel-aux.induct*) (*auto split*: *prod.split ltree.split*)

lemma *dom-postorder-relabel-aux*: *dom* (*snd* (*postorder-relabel-aux n t*)) = *nodes-ltree* (*snd* (*postorder-label-aux n* (*tree-ltree t*)))
proof (*induction n t rule*: *postorder-relabel-aux.induct*)
case (*1 n r*)
then show *?case* **by** (*auto split*: *if-splits*)
next
case (*2 n r t ts*)
obtain *n' f-t* **where** *f-t*: *postorder-relabel-aux n t* = (*n'*, *f-t*) **by** *fastforce*
then obtain *t'* **where** *t'*: *postorder-label-aux n* (*tree-ltree t*) = (*n'*, *t'*)
using *fst-postorder-label-aux-tree-ltree* **by** (*metis fst-eqD prod.exhaust-sel*)
obtain *n'' f-ts* **where** *f-ts*: *postorder-relabel-aux* (*Suc n'*) (*LNode r ts*) = (*n''*, *f-ts*) **by** *fastforce*
then obtain *ts' r'* **where** *ts'*: *postorder-label-aux* (*Suc n'*) (*tree-ltree* (*LNode r ts*)) = (*n''*, *LNode r' ts'*)
using *fst-postorder-label-aux-tree-ltree* **by** (*metis fst-eqD prod.exhaust-sel ltree.exhaust*)
show *?case* **using** *2 f-t f-ts t' ts'* **by** *auto*
qed

lemma *ran-postorder-relabel-aux*: *ran* (*snd* (*postorder-relabel-aux n t*)) = *nodes-ltree t*
proof (*induction n t rule*: *postorder-relabel-aux.induct*)
case (*1 n r*)
then show *?case* **by** (*simp add*: *ran-def*)
next
case (*2 n r t ts*)
obtain *n' f-t* **where** *f-t*: *postorder-relabel-aux n t* = (*n'*, *f-t*) **by** *fastforce*
obtain *n'' f-ts* **where** *f-ts*: *postorder-relabel-aux* (*Suc n'*) (*LNode r ts*) = (*n''*,

f-ts) **by** *fastforce*
have *dom f-t* ∩ *dom f-ts* = {} **using** *dom-postorder-relabel-aux f-t f-ts*
by (*metis disjoint-iff fst-eqD fst-postorder-label-aux-tree-ltree nodes-postorder-label-aux-ge nodes-postorder-label-aux-le not-less-eq-eq prod.exhaust-sel snd-conv*)
then show *?case* **using** *2 f-t f-ts* **by** (*simp add*: *ran-map-add*)
qed

lemma *relabel-ltree-eq*: ∀ *v*∈*nodes-ltree t*. *f v* = *g v* ⟹ *relabel-ltree f t* = *relabel-ltree g t*
by (*induction t*) *auto*

lemma *relabel-postorder-relabel-aux*: *relabel-ltree* (*the o snd* (*postorder-relabel-aux n t*)) (*snd* (*postorder-label-aux n* (*tree-ltree t*))) = *t*
proof (*induction n t rule*: *postorder-relabel-aux.induct*)
case (*1 n r*)
then show *?case* **by** *auto*
next
case (*2 n r t ts*)
obtain *n′ f-t* **where** *f-t*: *postorder-relabel-aux n t* = (*n′*, *f-t*) **by** *fastforce*
then obtain *t′* **where** *t′*: *postorder-label-aux n* (*tree-ltree t*) = (*n′*, *t′*)
using *fst-postorder-label-aux-tree-ltree* **by** (*metis fst-eqD prod.exhaust-sel*)
obtain *n′′ f-ts* **where** *f-ts*: *postorder-relabel-aux* (*Suc n′*) (*LNode r ts*) = (*n′′*, *f-ts*) **by** *fastforce*
then obtain *ts′ r′* **where** *ts′*: *postorder-label-aux* (*Suc n′*) (*tree-ltree* (*LNode r ts*)) = (*n′′*, *LNode r′ ts′*)
using *fst-postorder-label-aux-tree-ltree* **by** (*metis fst-eqD prod.exhaust-sel ltree.exhaust*)
have *ts′-in-f-ts*: ∀ *v*∈*nodes-ltree* (*LNode r′ ts′*). *v* ∈ *dom f-ts* **using** *f-ts ts′ dom-postorder-relabel-aux*
by (*metis snd-conv*)
have ∀ *v*∈*nodes-ltree t′*. *v* ∉ *dom f-ts* **using** *f-ts t′ ts′ f-t dom-postorder-relabel-aux*
by (*metis nodes-postorder-label-aux-ge nodes-postorder-label-aux-le not-less-eq-eq snd-conv*)
then show *?case* **using** *2 f-t f-ts t′ ts′ ts′-in-f-ts*
by (*auto intro*!: *relabel-ltree-eq simp*: *map-add-dom-app-simps*(*3*) *map-add-dom-app-simps*(*1*), *smt* (*verit*, *ccfv-threshold*) *map-add-dom-app-simps*(*1*) *map-eq-conv relabel-ltree-eq*)
qed

lemma *relabel-postorder-relabel*: *relabel-ltree* (*the o postorder-relabel t*) (*postorder-label* (*tree-ltree t*)) = *t*
unfolding *postorder-relabel-def postorder-label-def* **using** *relabel-postorder-relabel-aux*
by *auto*

lemma *relabel-postorder-aux-inj*: *distinct-ltree-nodes t* ⟹ *inj-on* (*the o snd* (*postorder-relabel-aux n t*)) (*nodes-ltree* (*snd* (*postorder-label-aux n* (*tree-ltree t*))))
proof (*induction n t rule*: *postorder-relabel-aux.induct*)
case (*1 n r*)
then show *?case* **by** *auto*
next

case (*2 n r t ts*)
have *disjoint-family-on-ts*: *disjoint-family-on nodes-ltree* (*set ts*) **using** *2*(*3*) **by** (*simp add*: *disjoint-family-on-def*)
obtain *n′ f-t* **where** *f-t*: *postorder-relabel-aux n t* = (*n′*, *f-t*) **by** *fastforce*
then obtain *t′* **where** *t′*: *postorder-label-aux n* (*tree-ltree t*) = (*n′*, *t′*)
using *fst-postorder-label-aux-tree-ltree* **by** (*metis fst-eqD prod.exhaust-sel*)
obtain *n′′ f-ts* **where** *f-ts*: *postorder-relabel-aux* (*Suc n′*) (*LNode r ts*) = (*n′′*, *f-ts*) **by** *fastforce*
then obtain *ts′ r′* **where** *ts′*: *postorder-label-aux* (*Suc n′*) (*tree-ltree* (*LNode r ts*)) = (*n′′*, *LNode r′ ts′*)
using *fst-postorder-label-aux-tree-ltree* **by** (*metis fst-eqD prod.exhaust-sel ltree.exhaust*)

have *t′-in-dom-f-t*: *nodes-ltree t′* ⊆ *dom f-t* **using** *f-t t′ dom-postorder-relabel-aux*
by (*metis order-refl snd-conv*)
have ∀ *v*∈*nodes-ltree t′*. *v* ∉ *dom f-ts* **using** *f-ts ts′ t′ dom-postorder-relabel-aux*
by (*metis nodes-postorder-label-aux-ge nodes-postorder-label-aux-le not-less-eq-eq snd-conv*)
then have *f-t′*: ∀ *v*∈*nodes-ltree t′*. *the* ((*f-t* ++ *f-ts*) *v*) = *the* (*f-t v*)
by (*simp add*: *map-add-dom-app-simps*(*3*))
have *inj-on* (λ*v*. *the* (*f-t v*)) (*nodes-ltree t′*) **using** *2 ts′ f-ts f-t t′ disjoint-family-on-ts* **by** *auto*
then have *inj-on-t′*: *inj-on* (λ*v*. *the* ((*f-t* ++ *f-ts*) *v*)) (*nodes-ltree t′*)
by (*metis* (*mono-tags*, *lifting*) *inj-on-cong f-t′*)
have *ts′-in-dom-f-ts*: ∀ *v*∈*nodes-ltree* (*LNode r′ ts′*). *v* ∈ *dom f-ts* **using** *f-ts ts′ dom-postorder-relabel-aux*
by (*metis snd-conv*)
then have *f-ts′*: ∀ *v*∈*nodes-ltree* (*LNode r′ ts′*). *the* ((*f-t* ++ *f-ts*) *v*) = *the* (*f-ts v*)
by (*simp add*: *map-add-dom-app-simps*(*1*))
have *inj-on* (λ*v*. *the* (*f-ts v*)) (*nodes-ltree* (*LNode r′ ts′*)) **using** *2 ts′ f-ts f-t disjoint-family-on-ts* **by** *simp*
then have *inj-on-ts′*: *inj-on* (λ*v*. *the* ((*f-t* ++ *f-ts*) *v*)) (*nodes-ltree* (*LNode r′ ts′*)) **using** *f-ts′ inj-on-cong* **by** *fast*

have (λ*v*. *the* ((*f-t* ++ *f-ts*) *v*)) ‘ *nodes-ltree t′* ∩ (λ*v*. *the* ((*f-t* ++ *f-ts*) *v*)) ‘ *nodes-ltree* (*LNode r′ ts′*) = {}
proof−
have (λ*v*. *the* ((*f-t* ++ *f-ts*) *v*)) ‘ *nodes-ltree t′* = (λ*v*. *the* (*f-t v*)) ‘ *nodes-ltree t′* **using** *f-t′* **by** *simp*
also have . . . ⊆ *ran f-t* **using** *t′-in-dom-f-t ran-def* **by** *fastforce*
also have . . . = *nodes-ltree t* **by** (*metis f-t ran-postorder-relabel-aux snd-conv*)
finally have *f-nodes-t′*: (λ*v*. *the* ((*f-t* ++ *f-ts*) *v*)) ‘ *nodes-ltree t′* ⊆ *nodes-ltree t* .

have (λ*v*. *the* ((*f-t* ++ *f-ts*) *v*)) ‘ *nodes-ltree* (*LNode r′ ts′*) = (λ*v*. *the* (*f-ts v*)) ‘ *nodes-ltree* (*LNode r′ ts′*)
using *f-ts′* **by** (*simp del*: *nodes-ltree.simps*)
also have . . . ⊆ *ran f-ts* **using** *ts′-in-dom-f-ts ran-def* **by** *fastforce*
also have . . . = *nodes-ltree* (*LNode r ts*) **by** (*metis f-ts ran-postorder-relabel-aux*

snd-conv)
finally have *f-nodes-ts'*: (λv. *the* ((*f-t* ++ *f-ts*) *v*)) ' *nodes-ltree* (*LNode* *r'* *ts'*) $\subseteq$ *nodes-ltree* (*LNode* *r* *ts*) **.**

have *nodes-ltree* *t* $\cap$ *nodes-ltree* (*LNode* *r* *ts*) = {} **using** *2*(*3*) **by** (*auto simp add*: *disjoint-family-on-def*)
then show *?thesis* **using** *f-nodes-t'* *f-nodes-ts'* **by** *blast*
qed
then have *inj-on* (λv. *the* ((*f-t* ++ *f-ts*) *v*)) (*nodes-ltree* *t'* $\cup$ *nodes-ltree* (*LNode* *r'* *ts'*)) **using** *inj-on-t'* *inj-on-ts'* *inj-on-Un* **by** *fast*
then show *?case* **using** *f-t* *t'* *f-ts* *ts'* **by** *simp*
qed

lemma *relabel-postorder-inj*: *distinct-ltree-nodes* *t* $\Longrightarrow$ *inj-on* (*the o postorder-relabel* *t*) (*nodes-ltree* (*postorder-label* (*tree-ltree* *t*)))
unfolding *postorder-relabel-def* *postorder-label-def* **using** *relabel-postorder-aux-inj* **by** *blast*

lemma (**in** *rtree*) *distinct-nodes-stree-of-graph*: *distinct-stree-nodes* (*stree-of-graph* (*V*,*E*,*r*))
using *rtree-axioms*
proof (*induction* (*V*,*E*,*r*) *arbitrary*: *V E r rule*: *stree-of-graph.induct*)
case (*1* V_T E_T *r*)
then interpret *t*: *rtree* V_T E_T *r* **by** *simp*
obtain *V'* *E'* **where** *VE'*: *t.remove-vertex* *r* = (*V'*, *E'*) **by** (*simp add*: *t.remove-vertex-def*)
interpret *subg*: *subgraph* *V'* *E'* V_T E_T **using** *t.subgraph-remove-vertex* *VE'* **by** *metis*
interpret *g'*: *fin-ulgraph* *V'* *E'* **using** *fin-ulgraph.intro* *subg.is-finite-subgraph* *t.fin-graph-system-axioms* *subg.is-subgraph-ulgraph* *t.ulgraph-axioms* **by** *blast*

have *finite* (*stree-of-graph* ' *t.subtrees*) **using** *t.finite-subtrees* **by** *blast*
then have *fset-Abs-fset-subtrees*[*simp*]: *fset* (*Abs-fset* (*stree-of-graph* ' *t.subtrees*)) = *stree-of-graph* ' *t.subtrees* **by** (*simp add*: *Abs-fset-inverse*)

have *r-notin-subtrees*: $\forall$ *s*$\in$*t.subtrees*. *r* $\notin$ *nodes-stree* (*stree-of-graph* *s*)
proof
fix *s* **assume** *subtree*: *s* $\in$ *t.subtrees*
then obtain *S* E_S r_S **where** *s*: *s* = (*S*,E_S,r_S) **using** *prod.exhaust* **by** *metis*
then interpret *s*: *rtree* *S* E_S r_S **using** *t.rtree-subtree* *subtree* **by** *blast*
have *S* $\in$ *g'.connected-components* **using** *subtree* *VE'* **unfolding** *s* *t.subtrees-def* **by** *auto*
then have *nodes-stree* (*stree-of-graph* (*S*,E_S,r_S)) $\subseteq$ *V'* **using** *s.nodes-stree-stree-of-graph* *g'.connected-component-wf* **by** *auto*
then show *r* $\notin$ *nodes-stree* (*stree-of-graph* *s*) **using** *VE'* **unfolding** *s* *t.remove-vertex-def* **by** *blast*
qed

have *nodes-stree* (*stree-of-graph* *s1*) $\cap$ *nodes-stree* (*stree-of-graph* *s2*) = {}
if *s1-subtree*: *s1* $\in$ *t.subtrees* **and** *s2-subtree*: *s2* $\in$ *t.subtrees* **and** *ne*: *stree-of-graph*

s1 ≠ *stree-of-graph s2* **for** *s1 s2*
proof−
obtain *V1 E1 r1* **where** *s1*: *s1* = (*V1*,*E1*,*r1*) **using** *prod.exhaust* **by** *metis*
then interpret *s1*: *rtree V1 E1 r1* **using** *t.rtree-subtree s1-subtree* **by** *blast*
have *V1-conn-comp*: *V1* ∈ *g′.connected-components* **using** *s1-subtree VE′* **unfolding** *t.subtrees-def s1* **by** *auto*
then have *s1-conn-comp*: *nodes-stree* (*stree-of-graph s1*) ∈ *g′.connected-components* **unfolding** *s1* **using** *s1.nodes-stree-stree-of-graph* **by** *auto*
obtain *V2 E2 r2* **where** *s2*: *s2* = (*V2*,*E2*,*r2*) **using** *prod.exhaust* **by** *metis*
then interpret *s2*: *rtree V2 E2 r2* **using** *t.rtree-subtree s2-subtree* **by** *blast*
have *V2-conn-comp*: *V2* ∈ *g′.connected-components* **using** *s2-subtree VE′* **unfolding** *t.subtrees-def s2* **by** *auto*
have *V1* ≠ *V2* **using** *s1 s2 s1-subtree s2-subtree VE′ ne* **unfolding** *t.subtrees-def* **by** *auto*
then have *V1* ∩ *V2* = {} **using** *V1-conn-comp V2-conn-comp g′.disjoint-connected-components* **unfolding** *disjoint-def* **by** *blast*
then show *?thesis* **using** *s1 s2 s1.nodes-stree-stree-of-graph s2.nodes-stree-stree-of-graph* **by** *simp*
qed
then have *disjoint-family-on nodes-stree* (*stree-of-graph* ‘ *t.subtrees*)
unfolding *disjoint-family-on-def* **by** *blast*
then show *?case* **using** *1 t.rtree-subtree r-notin-subtrees* **by** *auto*
qed

lemma *disintct-nodes-ltree-stree*: *distinct-stree-nodes t* ⟹ *distinct-ltree-nodes* (*ltree-stree t*)
using *distinct-ltree-stree-subtrees* **by** (*induction t*) (*auto simp*: *disjoint-family-on-def*, *metis disjoint-iff*)

lemma (**in** *rtree*) *tree-graph-tree-of-graph*: *tree-graph* (*tree-ltree* (*ltree-stree* (*stree-of-graph* (*V*,*E*,*r*)))) $\simeq_r$ (*V*,*E*,*r*)
proof−
define *t* **where** *t* = (*V*,*E*,*r*)
define *s* **where** *s* = *stree-of-graph t*
define *l* **where** *l* = *ltree-stree s*
define *l′* **where** *l′* = *postorder-label* (*tree-ltree l*)
define *s′* **where** *s′* = *stree-ltree l′*
define *t′* **where** *t′* = *tree-graph-stree s′*
obtain *V′ E′ r′* **where** *t′*: *t′* = (*V′*,*E′*,*r′*) **using** *prod.exhaust* **by** *metis*
interpret *t′*: *rtree V′ E′ r′* **using** *t′ rtree-tree-graph* **unfolding** *tree-graph-def t′-def s′-def l′-def* **by** *simp*
have *distinct-ltree-nodes l* **using** *distinct-nodes-stree-of-graph disintct-nodes-ltree-stree*
unfolding *l-def s-def t-def* **by** *blast*
then obtain *f* **where** *inj-on-l′*: *inj-on f* (*nodes-ltree l′*) **and** *relabel-l′*: *relabel-ltree f l′* = *l*
unfolding *l′-def* **using** *relabel-postorder-relabel relabel-postorder-inj* **by** *blast*
then have *relabel-stree f s′* = *s* **unfolding** *l-def s′-def*
using *relabel-stree-relabel-ltree* **by** *fastforce*
then have *app-rgraph-iso*: *app-rgraph-isomorphism f t′* = *t* **unfolding** *s-def t′-def*

t-def
using *t′ tree-graph-stree-of-graph* **by** (*simp add*: *app-rgraph-iso-tree-graph-stree*)
have *inj-on f* (*nodes-stree s′*) **unfolding** *s′-def* **using** *inj-on-l′* **by** *simp*
then have *inj-on-V′*: *inj-on f V′* **using** *t′ nodes-tree-graph-stree* **unfolding** *t′-def* **by** *fast*
have (V',E',r') $\simeq_r$ (V,E,r) **using** *app-rgraph-iso t′.rgraph-isomorph-app-iso inj-on-V′* **unfolding** *t′ t-def* **by** *auto*
then show *?thesis* **using** *t′* **unfolding** *tree-graph-def t-def s-def l-def l′-def s′-def t′-def* **by** *auto*
qed

lemma (**in** *rtree*) *stree-size-stree-of-graph*[*simp*]: *stree-size* (*stree-of-graph* (V,E,r)) = *card V*
using *distinct-nodes-stree-of-graph* **by** (*simp add*: *distinct-nodes-stree-size-card-nodes del*: *stree-of-graph.simps*)

lemma *inj-ltree-stree*: *inj ltree-stree*
proof
fix *t1* :: *′a stree*
and *t2* :: *′a stree*
assume *ltree-stree t1* = *ltree-stree t2*
then show *t1* = *t2*
proof (*induction t1 arbitrary*: *t2*)
case (*SNode r1 ts1*)
obtain *r2 ts2* **where** *t2*: *t2* = *SNode r2 ts2* **using** *stree.exhaust* **by** *blast*
then show *?case* **using** *SNode* **by** (*simp*, *metis SNode.prems stree.inject stree-ltree-stree*)
qed
qed

lemma *ltree-size-ltree-stree*[*simp*]: *ltree-size* (*ltree-stree t*) = *stree-size t*
using *inj-ltree-stree* **by** (*induction t*) (*auto simp*: *sum-list-distinct-conv-sum-set*[*OF distinct-ltree-stree-subtrees*] *fsum.F.rep-eq*,
smt (*verit*, *best*) *inj-on-def stree-ltree-stree sum.reindex-cong*)

lemma *tree-size-tree-ltree*[*simp*]: *tree-size* (*tree-ltree t*) = *ltree-size t*
by (*induction t*) (*auto*, *metis comp-eq-dest-lhs map-cong*)

lemma *regular-ltree-stree*: *regular-ltree* (*ltree-stree t*)
using *sorted-wrt-ltree-stree-subtrees* **by** (*induction t*) *auto*

lemma *regular-tree-ltree*: *regular-ltree t* $\Longrightarrow$ *regular* (*tree-ltree t*)
by (*induction t*) (*auto simp*: *sorted-map*)

lemma (**in** *rtree*) *tree-of-graph-regular-n-tree*: *tree-ltree* (*ltree-stree* (*stree-of-graph* (V,E,r))) $\in$ *regular-n-trees* (*card V*) (**is** *?t* $\in$ *?A*)
proof−
have *size-t*: *tree-size ?t* = *card V* **by** (*simp del*: *stree-of-graph.simps*)
have *regular ?t* **using** *regular-ltree-stree regular-tree-ltree* **by** *blast*

then show *?thesis* **using** *size-t* **unfolding** *regular-n-trees-def* **by** *blast*
qed

lemma (**in** *rtree*) *ex-regular-n-tree*: $\exists$ *t*$\in$*regular-n-trees* (*card V*). *tree-graph t* $\simeq_r$ (*V*,*E*,*r*)
using *tree-graph-tree-of-graph tree-of-graph-regular-n-tree* **by** *blast*

3.4 Injectivity with respect to isomorphism

lemma *app-rgraph-isomorphism-relabel-stree*: *app-rgraph-isomorphism f* (*tree-graph-stree t*) = *tree-graph-stree* (*relabel-stree f t*)
unfolding *tree-graph-stree-def* **by** *simp*

Lemmas relating the connected components of the tree graph with the root removed to the subtrees of an stree.

context
fixes *t r ts V′ E′*
assumes *t*: *t* = *SNode r ts*
assumes *distinct-nodes*: *distinct-stree-nodes t*
and *remove-vertex*: *graph-system.remove-vertex* (*nodes-stree t*) (*tree-graph-edges t*) *r* = (*V′*,*E′*)
begin

interpretation *t*: *rtree nodes-stree t tree-graph-edges t r* **using** *rtree-tree-graph-edges*[*OF distinct-nodes*] **unfolding** *t* **by** *simp*

interpretation *subg*: *ulsubgraph V′ E′ nodes-stree t tree-graph-edges t* **using** *remove-vertex t.subgraph-remove-vertex t.ulgraph-axioms ulsubgraph-def t* **by** *blast*

interpretation *g′*: *ulgraph V′ E′* **using** *subg.is-subgraph-ulgraph t.ulgraph-axioms* **by** *blast*

lemma *neighborhood-root*: *t.neighborhood r* = *root-stree ‘ fset ts*
unfolding *t.neighborhood-def t.vert-adj-def* **using** *distinct-nodes tree-graph-edges-wf root-stree-wf t*
by (*auto*, *blast*, *fastforce*, *blast*, *blast*)

lemma *V′*: *V′* = *nodes-stree t* − {*r*}
using *remove-vertex distinct-nodes* **unfolding** *t.remove-vertex-def* **by** *blast*

lemma *E′*: *E′* = $\bigcup$ (*tree-graph-edges ‘ fset ts*)
using *tree-graph-edges-wf distinct-nodes remove-vertex t* **unfolding** *t.remove-vertex-def t.vincident-def* **by** *auto*

lemma *subtrees-not-connected*:
assumes *s-in-ts*: *s* $\in$ *fset ts*
and *e*: {*u*, *v*} $\in$ *E′*
and *u-in-s*: *u* $\in$ *nodes-stree s*
shows *v* $\in$ *nodes-stree s*

proof –
have {*u*,*v*} ∈ *tree-graph-edges s* **using** *e u-in-s tree-graph-edges-wf s-in-ts distinct-nodes t* **unfolding** *E'*
by (*auto simp*: *disjoint-family-on-def*,
smt (*verit*, *del-insts*) *insert-absorb insert-disjoint*(*2*) *insert-subset tree-graph-edges-wf*)
then show *?thesis* **using** *tree-graph-edges-wf u-in-s* **by** *blast*
qed

lemma *subtree-connected-components*:
assumes *s-in-ts*: *s* ∈ *fset ts*
shows *nodes-stree s* ∈ *g'.connected-components*
proof –
interpret *s*: *rtree nodes-stree s tree-graph-edges s root-stree s* **using** *rtree-tree-graph-edges distinct-nodes s-in-ts t* **by** *auto*
interpret *subg'*: *ulsubgraph nodes-stree s tree-graph-edges s V' E'* **using** *distinct-nodes s-in-ts t* **by** *unfold-locales* (*auto simp*: *V' E'*)
have *conn-set*: *g'.is-connected-set* (*nodes-stree s*) **using** *s.connected subg'.is-connected-set* **by** *blast*
then show *?thesis* **using** *subtrees-not-connected s-in-ts g'.connected-set-connected-component nodes-stree-non-empty* **by** *fast*
qed

lemma *connected-components-subtrees*: *g'.connected-components* = *nodes-stree* ‘ *fset ts*
proof –
have *nodes-ts-ss-conn-comps*: *nodes-stree* ‘ *fset ts* ⊆ *g'.connected-components* **using** *subtree-connected-components* **by** *blast*
have *Un-nodes-ts*: ⋃(*nodes-stree* ‘ *fset ts*) = *V'* **unfolding** *V'* **using** *distinct-nodes t* **by** *auto*
show *?thesis* **using** *g'.subset-conn-comps-if-Union*[*OF nodes-ts-ss-conn-comps Un-nodes-ts*] **by** *simp*
qed

lemma *induced-edges-subtree*:
assumes *s-in-ts*: *s* ∈ *fset ts*
shows *graph-system.induced-edges E'* (*nodes-stree s*) = *tree-graph-edges s*
proof –
have *graph-system.induced-edges E'* (*nodes-stree s*) = {*e* ∈ ⋃ (*tree-graph-edges* ‘ *fset ts*). *e* ⊆ *nodes-stree s*} **using** *subg.H.induced-edges-def E'* **by** *auto*
also have ... = *tree-graph-edges s*
using *s-in-ts distinct-nodes tree-graph-edges-wf t*
by (*auto simp*: *disjoint-family-on-def*,
metis card.empty card-tree-graph-edges-distinct inf.bounded-iff nat.simps(*3*) *numeral-2-eq-2 subset-empty*)
finally show *?thesis* **.**
qed

lemma *root-subtree*:
assumes *s-in-ts*: *s* ∈ *fset ts*

shows (*THE r′. r′ ∈ (nodes-stree s) ∧ t.vert-adj r r′*) = *root-stree s*
proof
show *root-stree s ∈ nodes-stree s ∧ t.vert-adj r (root-stree s)* **unfolding** *t.vert-adj-def*
using *t root-stree-wf s-in-ts* **by** *auto*
next
fix *r′*
assume *r′: r′ ∈ nodes-stree s ∧ t.vert-adj r r′*
then have *edge-in-root-edges: {r, r′} ∈ (λt. {r, root-stree t}) ‘ fset ts*
unfolding *t.vert-adj-def* **using** *distinct-nodes tree-graph-edges-wf t* **by** *fastforce*
have *∀ s′∈fset ts. s′ ≠ s ⟶ r′ ∉ nodes-stree s′*
using *distinct-nodes s-in-ts r′* **unfolding** *t* **by** (*auto simp: disjoint-family-on-def*)
then show *r′ = root-stree s* **using** *edge-in-root-edges root-stree-wf* **by** (*smt* (*verit*)
doubleton-eq-iff image-iff)
qed

lemma *subtrees-tree-subtrees: t.subtrees = tree-graph-stree ‘ fset ts*
unfolding *t.subtrees-def tree-graph-stree-def* **using** *remove-vertex*
by (*simp add: connected-components-subtrees image-comp induced-edges-subtree*
root-subtree)

end

lemma *stree-of-graph-tree-graph-stree[simp]: distinct-stree-nodes t ⟹ stree-of-graph*
(*tree-graph-stree t*) = *t*
proof (*induction t*)
case (*SNode r ts*)
define *t* **where** *t: t = SNode r ts*
then have *root-t[simp]: root-stree t = r* **by** *simp*
have *distinct-t: distinct-stree-nodes t* **using** *SNode(2) t* **by** *blast*
interpret *t: rtree nodes-stree t tree-graph-edges t r* **using** *SNode(2) rtree-tree-graph-edges*
t **by** (*metis root-stree.simps*)
obtain *V′ E′* **where** *remove-vertex: t.remove-vertex r = (V′,E′)* **by** *fastforce*

have *stree-of-graph (tree-graph-stree t) = SNode r ts* **unfolding** *tree-graph-stree-def*
using *SNode t.rtree-axioms t.rtree-subtree*
by (*simp add: subtrees-tree-subtrees[OF t distinct-t remove-vertex] image-comp*
fset-inverse)
then show *?case* **unfolding** *t* **.**
qed

lemma *distinct-nodes-relabel: distinct-stree-nodes t ⟹ inj-on f (nodes-stree t)*
⟹ distinct-stree-nodes (relabel-stree f t)
by (*induction t*) (*auto simp: image-UN disjoint-family-on-def inj-on-def*, *metis*
IntI empty-iff)

lemma *relabel-stree-app-rgraph-isomorphism*:
assumes *distinct-stree-nodes t*
and *inj-on f (nodes-stree t)*
shows *relabel-stree f t = stree-of-graph (app-rgraph-isomorphism f (tree-graph-stree*

t))
 using *assms* **by** (*auto simp*: *app-rgraph-isomorphism-relabel-stree distinct-nodes-relabel*)

lemma (**in** *rgraph-isomorphism*) *app-rgraph-isomorphism-G*: *app-rgraph-isomorphism f* (V_G,E_G,r_G) = (V_H,E_H,r_H)
 using *bij-f edge-preserving root-preserving* **unfolding** *bij-betw-def* **by** *simp*

lemma *tree-graphs-iso-strees-iso*:
 assumes *tree-graph-stree t1* $\simeq_r$ *tree-graph-stree t2*
 and *distinct-t1*: *distinct-stree-nodes t1*
 and *distinct-t2*: *distinct-stree-nodes t2*
 shows ∃*f*. *inj-on f* (*nodes-stree t1*) ∧ *relabel-stree f t1* = *t2*
proof−
 obtain *f* **where** *rgraph-isomorphism* (*nodes-stree t1*) (*tree-graph-edges t1*) (*root-stree t1*) (*nodes-stree t2*) (*tree-graph-edges t2*) (*root-stree t2*) *f*
 using *assms* **unfolding** *tree-graph-stree-def* **by** *auto*
 then interpret *rgraph-isomorphism nodes-stree t1 tree-graph-edges t1 root-stree t1 nodes-stree t2 tree-graph-edges t2 root-stree t2 f* **.**
 have *inj*: *inj-on f* (*nodes-stree t1*) **using** *bij-f bij-betw-imp-inj-on* **by** *blast*
 have *relabel-stree f t1* = *t2*
 unfolding *relabel-stree-app-rgraph-isomorphism*[*OF distinct-t1 inj*] *tree-graph-stree-def app-rgraph-isomorphism-G*
 using *stree-of-graph-tree-graph-stree*[*OF distinct-t2*, *unfolded tree-graph-stree-def*] **by** *blast*
 then show *?thesis* **using** *inj* **by** *blast*
qed

Skip the ltree representation as it introduces complications with the proofs

fun *tree-stree* :: *'a stree* ⇒ *tree* **where**
 tree-stree (*SNode r ts*) = *Node* (*sorted-list-of-multiset* (*image-mset tree-stree* (*mset-set* (*fset ts*))))

fun *postorder-label-stree-aux* :: *nat* ⇒ *tree* ⇒ *nat* × *nat stree* **where**
 postorder-label-stree-aux n (*Node* []) = (*n*, *SNode n* {||})
| *postorder-label-stree-aux n* (*Node* (*t*#*ts*)) =
 (*let* (*n'*, *t'*) = *postorder-label-stree-aux n t in*
 case postorder-label-stree-aux (*Suc n'*) (*Node ts*) *of*
 (*n''*, *SNode r ts'*) ⇒ (*n''*, *SNode r* (*finsert t' ts'*)))

definition *postorder-label-stree* :: *tree* ⇒ *nat stree* **where**
 postorder-label-stree t = *snd* (*postorder-label-stree-aux 0 t*)

lemma *fst-postorder-label-stree-aux-eq*: *fst* (*postorder-label-stree-aux n t*) = *fst* (*postorder-label-aux n t*)
 by (*induction n t rule*: *postorder-label-stree-aux.induct*) (*auto split*: *prod.split stree.split ltree.split*)

lemma *postorder-label-stree-aux-eq*: *snd* (*postorder-label-stree-aux n t*) = *stree-ltree* (*snd* (*postorder-label-aux n t*))

by (*induction n t rule*: *postorder-label-aux.induct*) (*simp*, *simp split*: *prod.split stree.split ltree.split*,
metis fset-of-list-map fst-conv fst-postorder-label-stree-aux-eq sndI stree.inject stree-ltree.simps)

lemma *postorder-label-stree-eq*: *postorder-label-stree t* = *stree-ltree* (*postorder-label t*)
using *postorder-label-stree-aux-eq* **unfolding** *postorder-label-stree-def postorder-label-def*
by *blast*

lemma *postorder-label-stree-aux-mono*: *fst* (*postorder-label-stree-aux n t*) $\geq$ *n*
by (*induction n t rule*: *postorder-label-stree-aux.induct*) (*auto split*: *prod.split stree.split*, *fastforce*)

lemma *nodes-postorder-label-stree-aux-ge*: *postorder-label-stree-aux n t* = (n', t') $\Longrightarrow$ $v \in$ *nodes-stree* t' $\Longrightarrow$ $v \geq n$
by (*induction n t arbitrary*: n' t' *rule*: *postorder-label-stree-aux.induct*,
auto split: *prod.splits stree.splits*,
(*metis fst-conv le-SucI order.trans postorder-label-stree-aux-mono*)+)

lemma *nodes-postorder-label-stree-aux-le*: *postorder-label-stree-aux n t* = (n', t') $\Longrightarrow$ $v \in$ *nodes-stree* t' $\Longrightarrow$ $v \leq n'$
by (*induction n t arbitrary*: n' t' *rule*: *postorder-label-stree-aux.induct*,
auto split: *prod.splits stree.splits*,
metis Suc-leD fst-conv order-trans postorder-label-stree-aux-mono,
blast)

lemma *distinct-nodes-postorder-label-stree-aux*: *distinct-stree-nodes* (*snd* (*postorder-label-stree-aux n t*))
proof (*induction n t rule*: *postorder-label-stree-aux.induct*)
case (*1 n*)
then show *?case* **by** (*simp add*: *disjoint-family-on-def*)
next
case (*2 n t ts*)
obtain n' t' **where** t': *postorder-label-stree-aux n t* = (n', t') **by** *fastforce*
obtain n'' *r* ts' **where** ts': *postorder-label-stree-aux* (*Suc* n') (*Node ts*) = (n'', *SNode r* ts')
by (*metis eq-snd-iff stree.exhaust*)
then have $r \geq$ *Suc* n' **using** *nodes-postorder-label-stree-aux-ge* **by** *auto*
then have *r-notin-t'*: $r \notin$ *nodes-stree* t' **using** *nodes-postorder-label-stree-aux-le*[*OF* t'] **by** *fastforce*
have *disjoint-family-on nodes-stree* (*insert* t' (*fset* ts'))
using *2* t' ts' *nodes-postorder-label-stree-aux-le*[*OF* t'] *nodes-postorder-label-stree-aux-ge*[*OF* ts']
by (*auto simp add*: *disjoint-family-on-def*, *fastforce*+)
then show *?case* **using** *2* t' ts' *r-notin-t'* **by** *simp*
qed

lemma *distinct-nodes-postorder-label-stree*: *distinct-stree-nodes* (*postorder-label-stree*

t)
unfolding *postorder-label-stree-def* **using** *distinct-nodes-postorder-label-stree-aux*
by *simp*

lemma *tree-stree-postorder-label-stree-aux*: *regular t* $\Longrightarrow$ *tree-stree* (*snd* (*postorder-label-stree-aux n t*)) = *t*
proof (*induction t rule*: *postorder-label-stree-aux.induct*)
case (*1 n*)
then show *?case* **by** *auto*
next
case (*2 n t ts*)
obtain *n′ t′* **where** *nt′*: *postorder-label-stree-aux n t* = (*n′*, *t′*) **by** *fastforce*
obtain *n″ r ts′* **where** *nt″*: *postorder-label-stree-aux* (*Suc n′*) (*Node ts*) = (*n″*, *SNode r ts′*)
using *stree.exhaust prod.exhaust* **by** *metis*
have *t′* $\notin$ *fset ts′* **using** *nodes-postorder-label-stree-aux-le*[*OF nt′*] *nodes-postorder-label-stree-aux-ge*[*OF nt″*]
by (*auto*, *meson not-less-eq-eq root-stree-wf*)
then show *?case* **using** *2 nt′ nt″* **by** (*auto simp*: *insort-is-Cons*)
qed

lemma *tree-ltree-postorder-label-stree*[*simp*]: *regular t* $\Longrightarrow$ *tree-stree* (*postorder-label-stree t*) = *t*
using *tree-stree-postorder-label-stree-aux* **unfolding** *postorder-label-stree-def* **by** *blast*

lemma *inj-relabel-subtrees*:
assumes *distinct-nodes*: *distinct-stree-nodes* (*SNode r ts*)
and *inj-on-nodes*: *inj-on f* (*nodes-stree* (*SNode r ts*))
shows *inj-on* (*relabel-stree f*) (*fset ts*)
proof
fix *t1 t2*
assume *t1-subtree*: *t1* $\in$ *fset ts*
and *t2-subtree*: *t2* $\in$ *fset ts*
and *relabel-eq*: *relabel-stree f t1* = *relabel-stree f t2*
then have *nodes-stree* (*relabel-stree f t1*) = *nodes-stree* (*relabel-stree f t2*) **by** *simp*
then have *f* ‘ *nodes-stree t1* = *f* ‘ *nodes-stree t2* **by** *simp*
then have *nodes-stree t1* = *nodes-stree t2* **using** *inj-on-nodes t1-subtree t2-subtree inj-on-image*[*of f nodes-stree* ‘ *fset ts*]
by (*simp*, *meson image-eqI inj-onD*)
then show *t1* = *t2* **using** *distinct-nodes nodes-stree-non-empty t1-subtree t2-subtree*
by (*auto simp add*: *disjoint-family-on-def*, *force*)
qed

lemma *inj-on-subtree*: *inj-on f* (*nodes-stree* (*SNode r ts*)) $\Longrightarrow$ *t* $\in$ *fset ts* $\Longrightarrow$ *inj-on f* (*nodes-stree t*)
unfolding *inj-on-def* **by** *simp*

lemma *tree-stree-relabel-stree*: *distinct-stree-nodes t* $\Longrightarrow$ *inj-on f* (*nodes-stree t*) $\Longrightarrow$ *tree-stree* (*relabel-stree f t*) = *tree-stree t*
proof (*induction t*)
  case (*SNode r ts*)
  then have *IH*: $\forall t \in \#$ *mset-set* (*fset ts*). *tree-stree* (*relabel-stree f t*) = *tree-stree t*
    using *inj-on-subtree*[*OF SNode*(*3*)] *elem-mset-set finite-fset* **by** *auto*
  show *?case* **using** *inj-relabel-subtrees*[*OF SNode*(*2*) *SNode*(*3*)]
    by (*auto simp add*: *mset-set-image-inj*, *metis IH image-mset-cong*)
qed

lemma *tree-ltree-relabel-ltree-postorder-label-stree*: *regular t* $\Longrightarrow$ *inj-on f* (*nodes-stree* (*postorder-label-stree t*)) $\Longrightarrow$ *tree-stree* (*relabel-stree f* (*postorder-label-stree t*)) = *t*
  using *tree-stree-relabel-stree distinct-nodes-postorder-label-stree* **by** *fastforce*

lemma *postorder-label-stree-inj*: *regular t1* $\Longrightarrow$ *regular t2* $\Longrightarrow$ *inj-on f* (*nodes-stree* (*postorder-label-stree t1*)) $\Longrightarrow$ *relabel-stree f* (*postorder-label-stree t1*) = *postorder-label-stree t2* $\Longrightarrow$ *t1* = *t2*
  using *tree-ltree-relabel-ltree-postorder-label-stree* **by** *fastforce*

lemma *tree-graph-inj-iso*: *regular t1* $\Longrightarrow$ *regular t2* $\Longrightarrow$ *tree-graph t1* $\simeq_r$ *tree-graph t2* $\Longrightarrow$ *t1* = *t2*
  using *postorder-label-stree-inj tree-graphs-iso-strees-iso distinct-nodes-postorder-label distinct-nodes-stree-ltree postorder-label-stree-eq* **unfolding** *tree-graph-def* **by** *metis*

lemma *tree-graph-inj*:
  assumes *regular-t1*: *regular t1*
    and *regular-t2*: *regular t2*
    and *tree-graph-eq*: *tree-graph t1* = *tree-graph t2*
  shows *t1* = *t2*
proof−
  obtain *V E r* **where** *g*: *tree-graph t1* = (*V*,*E*,*r*) **using** *prod.exhaust* **by** *metis*
  then interpret *rtree V E r* **using** *rtree-tree-graph* **by** *auto*
  have *tree-graph t1* $\simeq_r$ *tree-graph t2* **using** *tree-graph-eq g rgraph-isomorph-refl* **by** *simp*
  then show *?thesis* **using** *tree-graph-inj-iso regular-t1 regular-t2* **by** *simp*
qed

end

4 Enumeration of Rooted Trees

theory *Rooted-Tree-Enumeration*
  imports *Rooted-Tree*
begin

Algorithm inspired by works of Beyer and Hedetniemi [1], performing the same operations but directly on a recursive tree data structure instead of level sequences.

```
definition n-rtree-graphs :: nat ⇒ nat rpregraph set where
  n-rtree-graphs n = {(V,E,r). rtree V E r ∧ card V = n}
```

Recursive definition on the tree structure without using level sequences

```
fun trim-tree :: nat ⇒ tree ⇒ nat × tree where
  trim-tree 0 t = (0, t)
| trim-tree (Suc 0) t = (0, Node [])
| trim-tree (Suc n) (Node []) = (n, Node [])
| trim-tree n (Node (t#ts)) =
  (case trim-tree n (Node ts) of
    (0, t') ⇒ (0, t') |
    (n1, Node ts') ⇒
      let (n2, t') = trim-tree n1 t
      in (n2, Node (t'#ts')))

lemma fst-trim-tree-lt[termination-simp]: n ≠ 0 ⟹ fst (trim-tree n t) < n
  by (induction n t rule: trim-tree.induct, auto split: prod.split nat.split tree.split,
fastforce)

fun fill-tree :: nat ⇒ tree ⇒ tree list where
  fill-tree 0 - = []
| fill-tree n t =
    (let (n', t') = trim-tree n t
    in fill-tree n' t' @ [t'])

fun next-tree-aux :: nat ⇒ tree ⇒ tree option where
  next-tree-aux n (Node []) = None
| next-tree-aux n (Node (Node [] # ts)) = next-tree-aux (Suc n) (Node ts)
| next-tree-aux n (Node (Node (Node [] # rs) # ts)) = Some (Node (fill-tree (Suc
n) (Node rs) @ (Node rs) # ts))
| next-tree-aux n (Node (t # ts)) = Some (Node (the (next-tree-aux n t) # ts))

fun next-tree :: tree ⇒ tree option where
  next-tree t = next-tree-aux 0 t


lemma next-tree-aux-None-iff: next-tree-aux n t = None ⟷ height t < 2
proof (induction n t rule: next-tree-aux.induct)
  case (1 n)
  then show ?case by auto
next
  case (2 n ts)
  then show ?case by (cases ts) auto
next
  case (3 n rs ts)
  then show ?case by (auto simp: Max-gr-iff)
next
  case (4 n vc vd vb ts)
```

then show *?case*
by (*metis One-nat-def Suc-n-not-le-n dual-order.trans height-Node-cons le-add1 less-2-cases*
next-tree-aux.simps(*4*) *option.simps*(*3*) *plus-1-eq-Suc*)
qed

lemma *next-tree-Some-iff*: ($\exists t'$. *next-tree* t = *Some* t') $\longleftrightarrow$ *height* $t \geq 2$
using *next-tree-aux-None-iff* **by** (*metis linorder-not-less next-tree.simps not-Some-eq*)

4.1 Enumeration is monotonically decreasing

lemma *trim-id*: *trim-tree* n t = (*Suc* n', t') $\Longrightarrow t = t'$
by (*induction* n t *arbitrary*: n' t' *rule*: *trim-tree.induct*) (*auto split*: *prod.splits nat.splits tree.splits*)

lemma *trim-tree-le*: (n', t') = *trim-tree* n $t \Longrightarrow t' \leq t$
using *trim-id* **by** (*induction* n t *arbitrary*: n' t' *rule*: *trim-tree.induct*)
(*auto split*: *prod.splits tree.splits nat.splits simp*: *order-less-imp-le tree-less-cons'*, *fastforce*)

lemma *fill-tree-le*: $r \in$ *set* (*fill-tree* n t) $\Longrightarrow r \leq t$
using *trim-tree-le* **by** (*induction* n t *rule*: *fill-tree.induct*) (*auto*, *fastforce*)

lemma *next-tree-aux-lt*: *height* $t \geq 2 \Longrightarrow$ *the* (*next-tree-aux* n t) $< t$
proof (*induction* n t *rule*: *next-tree-aux.induct*)
case (*1* n)
then show *?case* **by** *auto*
next
case (*2* n *ts*)
then show *?case* **using** *tree-less-cons'* **by** (*cases ts*) *auto*
next
case (*3* n *rs ts*)
then show *?case* **using** *tree-less-comm-suffix2 tree-less-cons* **by** *simp*
next
case (*4* n *vc vd vb ts*)
have *height* (*Node* (*Node* (*vc* # *vd*) # *vb*)) $\geq$ *2* **unfolding** *numeral-2-eq-2*
by (*metis dual-order.antisym height-Node-cons less-eq-nat.simps*(*1*) *not-less-eq-eq*)
then show *?case* **using** *4 tree-less-cons2* **by** *simp*
qed

lemma *next-tree-lt*: *height* $t \geq 2 \Longrightarrow$ *the* (*next-tree* t) $< t$
using *next-tree-aux-lt* **by** *simp*

lemma *next-tree-lt'*: *next-tree* t = *Some* $t' \Longrightarrow t' < t$
using *next-tree-lt next-tree-Some-iff* **by** *fastforce*

4.2 Size preservation

lemma *size-trim-tree*: $n \neq 0 \Longrightarrow$ *trim-tree* n t = (n', t') $\Longrightarrow n'$ + *tree-size* $t' = n$

by (*induction n t arbitrary*: *n′ t′ rule*: *trim-tree.induct*) (*auto split*: *prod.splits nat.splits tree.splits*)

lemma *size-fill-tree*: *sum-list* (*map tree-size* (*fill-tree n t*)) = *n*
using *size-trim-tree* **by** (*induction n t rule*: *fill-tree.induct*) (*auto split*: *prod.split*)

lemma *size-next-tree-aux*: *height t* ≥ *2* ⟹ *tree-size* (*the* (*next-tree-aux n t*)) = *tree-size t* + *n*
proof (*induction n t rule*: *next-tree-aux.induct*)
case (*1 n*)
then show *?case* **by** *auto*
next
case (*2 n ts*)
then show *?case* **by** (*cases ts*) *auto*
next
case (*3 n rs ts*)
then show *?case* **using** *size-fill-tree* **by** (*auto simp del*: *fill-tree.simps*)
next
case (*4 n vc vd vb ts*)
have *height-t*: *height* (*Node* (*Node* (*vc* # *vd*) # *vb*)) ≥ *2* **unfolding** *numeral-2-eq-2*
by (*metis dual-order.antisym height-Node-cons less-eq-nat.simps*(*1*) *not-less-eq-eq*)
then show *?case* **using** *4* **by** *auto*
qed

lemma *size-next-tree*: *height t* ≥ *2* ⟹ *tree-size* (*the* (*next-tree t*)) = *tree-size t*
using *size-next-tree-aux* **by** *simp*

lemma *size-next-tree′*: *next-tree t* = *Some t′* ⟹ *tree-size t′* = *tree-size t*
using *size-next-tree next-tree-Some-iff* **by** *fastforce*

4.3 Setup for termination proof

definition *lt-n-trees n* ≡ {*t*. *tree-size t* ≤ *n*}

lemma *n-trees-eq*: *n-trees n* = *Node* ‘ {*ts*. *tree-size* (*Node ts*) = *n*}
proof−
have *n-trees n* = {*Node ts* | *ts*. *tree-size* (*Node ts*) = *n*} **unfolding** *n-trees-def* **by** (*metis tree-size.cases*)
then show *?thesis* **by** *blast*
qed

lemma *lt-n-trees-eq*: *lt-n-trees* (*Suc n*) = *Node* ‘ {*ts*. *tree-size* (*Node ts*) ≤ *Suc n*}
proof−
have *lt-n-trees* (*Suc n*) = {*Node ts* | *ts*. *tree-size* (*Node ts*) ≤ *Suc n*} **unfolding** *lt-n-trees-def* **by** (*metis tree-size.cases*)
then show *?thesis* **by** *blast*
qed

lemma *finite-lt-n-trees*: *finite* (*lt-n-trees n*)

proof (*induction n*)
  case *0*
  then show *?case* **unfolding** *lt-n-trees-def* **using** *not-finite-existsD not-less-eq-eq tree-size-ge-1* **by** *auto*
next
  case (*Suc n*)
  have ∀ *ts*∈{*ts. tree-size* (*Node ts*) ≤ *Suc n*}. *set ts* ⊆ *lt-n-trees n* **unfolding** *lt-n-trees-def* **using** *tree-size-children* **by** *fastforce*

  have {*ts. tree-size* (*Node ts*) ≤ *Suc n*} = {*ts. tree-size* (*Node ts*) ≤ *Suc n* ∧ *set ts* ⊆ *lt-n-trees n* ∧ *length ts* ≤ *n*} **unfolding** *lt-n-trees-def* **using** *tree-size-children length-children* **by** *fastforce*
  then have *finite* {*ts. tree-size* (*Node ts*) ≤ *Suc n*} **using** *finite-lists-length-le*[*OF Suc.IH*] **by** *auto*
  then show *?case* **unfolding** *lt-n-trees-eq* **by** *blast*
qed

lemma *n-trees-subset-lt-n-trees*: *n-trees n* ⊆ *lt-n-trees n*
  unfolding *n-trees-def lt-n-trees-def* **by** *blast*

lemma *finite-n-trees*: *finite* (*n-trees n*)
  using *n-trees-subset-lt-n-trees finite-lt-n-trees rev-finite-subset* **by** *metis*

4.4 Algorithms for enumeration

fun *greatest-tree* :: *nat* ⇒ *tree* **where**
  greatest-tree (*Suc 0*) = *Node* []
| *greatest-tree* (*Suc n*) = *Node* [*greatest-tree n*]

function *n-tree-enum-aux* :: *tree* ⇒ *tree list* **where**
  n-tree-enum-aux t =
    (*case next-tree t of None* ⇒ [*t*] | *Some t′* ⇒ *t* # *n-tree-enum-aux t′*)
  by *pat-completeness auto*

fun *n-tree-enum* :: *nat* ⇒ *tree list* **where**
  n-tree-enum 0 = []
| *n-tree-enum n* = *n-tree-enum-aux* (*greatest-tree n*)

termination *n-tree-enum-aux*
proof (*relation measure* (λ*t. card* {*r. r* < *t* ∧ *tree-size r* = *tree-size t*}), *auto*)
  fix *t t′* **assume** *t-t′*: *next-tree-aux 0 t* = *Some t′*
  then have *height-t*: *height t* ≥ *2* **using** *next-tree-Some-iff* **by** *auto*
  then have *t′* < *t* **using** *t-t′ next-tree-lt* **by** *fastforce*
  have *size-t′-t*: *tree-size t′* = *tree-size t* **using** *size-next-tree height-t t-t′* **by** *fastforce*
  let *?meas-t′* = {*r. r* < *t′* ∧ *tree-size r* = *tree-size t′*}
  let *?meas-t* = {*r. r* < *t* ∧ *tree-size r* = *tree-size t*}
  have *fin*: *finite ?meas-t* **using** *finite-n-trees* **unfolding** *n-trees-def* **by** *auto*
  have *?meas-t′* ⊆ *?meas-t* **using** ‹*t′* < *t*› *size-t′-t* **by** *auto*

then show *card* {*r*. *r* < *t′* ∧ *tree-size* *r* = *tree-size* *t′*} < *card* {*r*. *r* < *t* ∧ *tree-size* *r* = *tree-size* *t*}
using *fin* ‹*t′* < *t*› *psubset-card-mono* *size-t′-t* **by** *auto*
qed

definition *n-rtree-graph-enum* :: *nat* ⇒ *nat rpregraph list* **where**
n-rtree-graph-enum n = *map tree-graph* (*n-tree-enum n*)

4.5 Regularity

lemma *regular-trim-tree*: *regular t* ⟹ *regular* (*snd* (*trim-tree n t*))
by (*induction n t rule*: *trim-tree.induct*, *auto split*: *prod.split nat.split tree.split*, *metis dual-order.trans tree.inject trim-id trim-tree-le*)

lemma *regular-trim-tree′*: *regular t* ⟹ (*n′*, *t′*) = *trim-tree n t* ⟹ *regular t′*
using *regular-trim-tree* **by** (*metis snd-eqD*)

lemma *sorted-fill-tree*: *sorted* (*fill-tree n t*)
using *fill-tree-le* **by** (*induction n t rule*: *fill-tree.induct*) (*auto simp*: *sorted-append split*: *prod.split*)

lemma *regular-fill-tree*: *regular t* ⟹ *r* ∈ *set* (*fill-tree n t*) ⟹ *regular r*
using *regular-trim-tree′* **by** (*induction n t rule*: *fill-tree.induct*) *auto*

lemma *regular-next-tree-aux*: *regular t* ⟹ *height t* ≥ *2* ⟹ *regular* (*the* (*next-tree-aux n t*))
proof (*induction n t rule*: *next-tree-aux.induct*)
case (*1 n*)
then show *?case* **by** *auto*
next
case (*2 n ts*)
then show *?case* **by** (*cases ts*) *auto*
next
case (*3 n rs ts*)
then have *regular-rs*: *regular* (*Node rs*) **by** *simp*
have ∀ *t* ∈ *set ts*. *Node* (*rs*) < *t* **using** *3*(*1*) *tree-less-cons*[*of rs Node* []] **by** *auto*
then show *?case* **using** *3 sorted-fill-tree regular-fill-tree*[*OF regular-rs*] *fill-tree-le*
by (*auto simp del*: *fill-tree.simps simp*: *sorted-append*, *meson dual-order.trans tree-le-cons*)
next
case (*4 n vc vd vb ts*)
have *height-t*: *height* (*Node* (*Node* (*vc* # *vd*) # *vb*)) ≥ *2* **unfolding** *numeral-2-eq-2*
by (*metis dual-order.antisym height-Node-cons less-eq-nat.simps*(*1*) *not-less-eq-eq*)
then show *?case* **using** *4* **by** (*auto*, *meson height-t dual-order.strict-trans1 next-tree-aux-lt nless-le*)
qed

lemma *regular-next-tree*: *regular t* ⟹ *height t* ≥ *2* ⟹ *regular* (*the* (*next-tree t*))
using *regular-next-tree-aux* **by** *simp*

lemma *regular-next-tree'*: *regular t* ⟹ *next-tree t* = *Some t'* ⟹ *regular t'*
  using *regular-next-tree next-tree-Some-iff* **by** *fastforce*

lemma *regular-n-tree-enum-aux*: *regular t* ⟹ *r* ∈ *set* (*n-tree-enum-aux t*) ⟹ *regular r*
proof (*induction t rule*: *n-tree-enum-aux.induct*)
  case (*1 t*)
  then show *?case*
  proof (*cases next-tree-aux 0 t*)
    case *None*
    then show *?thesis* **using** *1* **by** *auto*
  next
    case (*Some a*)
    then show *?thesis* **using** *1 regular-next-tree'* **by** *auto*
  qed
qed

lemma *regular-n-tree-greatest-tree*: *n* ≠ *0* ⟹ *greatest-tree n* ∈ *regular-n-trees n*
proof (*induction n*)
  case *0*
  then show *?case* **by** *auto*
next
  case (*Suc n*)
  then show *?case* **unfolding** *regular-n-trees-def n-trees-def* **by** (*cases n*) *auto*
qed

lemma *regular-n-tree-enum*: *t* ∈ *set* (*n-tree-enum n*) ⟹ *regular t*
  using *regular-n-tree-enum-aux regular-n-tree-greatest-tree* **unfolding** *regular-n-trees-def*
by (*cases n*) *auto*

lemma *size-n-tree-enum-aux*: *n* ≠ *0* ⟹ *r* ∈ *set* (*n-tree-enum-aux t*) ⟹ *tree-size r* = *tree-size t*
proof (*induction t rule*: *n-tree-enum-aux.induct*)
  case (*1 t*)
  then show *?case*
  proof (*cases next-tree-aux 0 t*)
    case *None*
    then show *?thesis* **using** *1* **by** *auto*
  next
    case (*Some a*)
    then show *?thesis* **using** *1 size-next-tree'* **by** *auto*
  qed
qed

lemma *size-greatest-tree*[*simp*]: *n* ≠ *0* ⟹ *tree-size* (*greatest-tree n*) = *n*
  by (*induction n rule*: *greatest-tree.induct*) *auto*

lemma *size-n-tree-enum*: $t \in set\ (n\text{-}tree\text{-}enum\ n) \Longrightarrow tree\text{-}size\ t = n$
using *size-n-tree-enum-aux size-greatest-tree* **by** (*cases n*, *auto*, *fastforce*)

4.6 Totality

lemma *set* (*n-tree-enum n*) ⊆ *regular-n-trees n*
using *regular-n-tree-enum size-n-tree-enum* **unfolding** *regular-n-trees-def n-trees-def*
by *blast*

lemma *greatest-tree-lt-Suc*: $n \neq 0 \Longrightarrow greatest\text{-}tree\ n < greatest\text{-}tree\ (Suc\ n)$
by (*induction n rule*: *greatest-tree.induct*) (*auto simp*: *tree-less-nested*)

lemma *greatest-tree-ge*: $tree\text{-}size\ t \leq n \Longrightarrow t \leq greatest\text{-}tree\ n$
proof (*induction n arbitrary*: *t rule*: *greatest-tree.induct*)
case *1*
then show *?case* **by** (*cases t rule*: *tree-cons-exhaust*) (*auto simp*: *tree-size-ne-0*)
next
case (*2 v*)
then show *?case*
proof (*cases t rule*: *tree-rev-exhaust*)
case *Nil*
then show *?thesis* **by** *simp*
next
case (*Snoc ts r*)
then have *r-le-greatest-Suc-v*: $r \leq greatest\text{-}tree\ (Suc\ v)$ **using** *2* **by** *auto*
then show *?thesis*
proof (*cases* $r = greatest\text{-}tree\ (Suc\ v)$)
case *True*
then have $ts = []$ **using** *2*(*2*) *Snoc* **by** (*simp add*: *tree-size-ne-0*)
then show *?thesis* **using** *Snoc r-le-greatest-Suc-v* **by** *auto*
next
case *False*
then show *?thesis* **using** *r-le-greatest-Suc-v Snoc* **by** *auto*
qed
qed
next
case *3*
then show *?case* **by** (*simp add*: *tree-size-ne-0*)
qed

fun *least-tree* :: *nat* ⇒ *tree* **where**
least-tree (*Suc n*) = *Node* (*replicate n* (*Node* []))

lemma *regular-n-tree-least-tree*: $n \neq 0 \Longrightarrow least\text{-}tree\ n \in regular\text{-}n\text{-}trees\ n$
proof (*induction n*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc n*)

then show *?case* **unfolding** *regular-n-trees-def n-trees-def* **by** (*cases n*) *auto*
qed

lemma *height-lt-2-least-tree*: $t \in$ *regular-n-trees* $n \Longrightarrow$ *height* $t < 2 \Longrightarrow t =$ *least-tree n*
proof (*induction n arbitrary*: *t*)
case *0*
have *regular-n-trees* $0 = \{\}$ **unfolding** *regular-n-trees-def n-trees-def* **using** *tree-size.elims* **by** *auto*
then show *?case* **using** *0* **by** *blast*
next
case (*Suc n*)
then show *?case*
proof (*cases* $n = 0$)
case *True*
then show *?thesis* **using** *Suc tree-size.elims* **unfolding** *regular-n-trees-def n-trees-def*
by (*auto*, *metis leD length-children length-greater-0-conv*)
next
case *False*
then have *t-non-empty*: $t \neq$ *Node* [] **using** *Suc*(*2*) **unfolding** *regular-n-trees-def n-trees-def* **by** *auto*
then have *height-t*: *height* $t = 1$ **using** *Suc*(*3*)
by (*metis One-nat-def gr0-conv-Suc height.elims less-2-cases less-numeral-extra*(*3*))
obtain *s ts* **where** *s-ts*: $t =$ *Node* (*s* # *ts*) **using** *t-non-empty* **by** (*meson height.elims*)
then have *height* $s = 0$ **by** (*metis Suc-le-eq height-Node-cons less-one height-t*)
then have *s*: $s =$ *Node* [] **using** *height-0-iff* **by** *simp*
then have *regular-ts*: *Node ts* $\in$ *regular-n-trees n* **using** *Suc*(*2*) **unfolding** *s-ts regular-n-trees-def n-trees-def* **by** *auto*
have *height* (*Node ts*) < 2 **using** *height-t height-children height-children-le-height* **unfolding** *s-ts One-nat-def* **by** *fastforce*
then have *Node ts* = *least-tree n* **using** *Suc*(*1*) *regular-ts* **by** *blast*
then show *?thesis* **using** *False gr0-conv-Suc s s-ts* **by** *auto*
qed
qed

lemma *least-tree-le*: $n \neq 0 \Longrightarrow$ *tree-size* $t \geq n \Longrightarrow$ *least-tree* $n \leq t$
proof (*induction n arbitrary*: *t rule*: *less-induct*)
case (*less n*)
then obtain n' **where** *n*: $n =$ *Suc* n' **using** *least-tree.cases* **by** *blast*
then obtain *ts* **where** *t*: $t =$ *Node ts* **by** (*cases t*) *auto*
then show *?case*
proof (*cases* n')
case *0*
then show *?thesis* **using** *n* **by** *simp*
next
case (*Suc* n'')
then show *?thesis*

```
    proof (cases ts rule: rev-exhaust)
      case Nil
      then show ?thesis using less t n by auto
    next
      case (snoc rs r)
      then show ?thesis
      proof (cases r = Node [])
        case True
        then have tree-size (Node rs) ≥ n'' using less(3) unfolding n t Suc snoc
by auto
        then show ?thesis using less True unfolding n t Suc snoc
          by (auto simp: simp: replicate-append-same[symmetric], force)
      next
        case False
        then show ?thesis using less False unfolding n t Suc snoc
          by (auto simp: replicate-append-same[symmetric] tree-less-empty-iff)
      qed
    qed
  qed
qed

lemma trim-id': n ≥ tree-size t ⟹ trim-tree n t = (n', t') ⟹ t' = t
proof (induction n t arbitrary: n' t' rule: trim-tree.induct)
  case (1 t)
  then show ?case by auto
next
  case (2 t)
  then have t = Node [] using le-Suc-eq tree-size-1-iff tree-size-ne-0 by simp
  then show ?case using 2 by auto
next
  case (3 v)
  then show ?case by auto
next
  case (4 va t ts)
  then show ?case using size-trim-tree[OF - 4(4)] size-trim-tree
    by (auto split: prod.splits nat.splits simp: tree-size-ne-0, fastforce)
qed

lemma tree-ge-lt-suffix: Node ts ≤ r ⟹ r < Node (t#ts) ⟹ ∃ ss. r = Node (ss
@ ts)
proof (induction ts arbitrary: r rule: rev-induct)
  case Nil
  then show ?case by (cases r rule: tree-rev-exhaust) auto
next
  case (snoc x xs)
  then show ?case using tree-le-empty2-iff
    by (cases r rule: tree-rev-exhaust)
    (simp-all, metis Cons-eq-appendI tree.inject tree-less-antisym tree-less-snoc2-iff)
qed
```

lemma *trim-tree-0-iff*: *fst* (*trim-tree n t*) = *0* ⟷ *n* ≤ *tree-size t*
using *size-trim-tree trim-id tree-size-ge-1*
by (*induction n t rule*: *trim-tree.induct*, *auto split*: *prod.split nat.split tree.split*, *fastforce*+)

lemma *trim-tree-greatest-le*: *tree-size r* ≤ *n* ⟹ *r* ≤ *t* ⟹ *r* ≤ *snd* (*trim-tree n t*)
proof (*induction n t arbitrary*: *r rule*: *trim-tree.induct*)
case (*1 t*)
then show *?case* **by** *auto*
next
case (*2 t*)
then show *?case* **using** *tree-size-ne-0 tree-size-1-iff* **by** (*simp add*: *le-Suc-eq*)
next
case (*3 v*)
then show *?case* **by** *auto*
next
case (*4 va t ts*)
obtain *n1 t1* **where** *nt1*: *trim-tree* (*Suc* (*Suc va*)) (*Node ts*) = (*n1*, *t1*) **by** *fastforce*
then show *?case*
proof (*cases n1*)
case *0*
then show *?thesis*
proof (*cases r* ≤ *Node ts*)
case *True*
then show *?thesis* **using** *4 0 nt1* **by** *simp*
next
case *False*
then obtain *ss s* **where** *r*: *r* = *Node* (*ss* @ *s* # *ts*) **using** *4*(*4*) *tree-ge-lt-suffix*
by (*metis append.assoc append-Cons append-Nil nle-le rev-exhaust tree-le-def*)
then have *tree-size* (*Node ts*) ≥ *Suc* (*Suc va*) **using** *nt1 trim-tree-0-iff* **unfolding** *0* **by** *fastforce*
then have *tree-size r* > *Suc* (*Suc va*) **using** *tree-size-ne-0* **unfolding** *r*
by (*auto simp*: *add-strict-increasing trans-less-add2*)
then show *?thesis* **using** *4*(*3*) **by** *auto*
qed
next
case (*Suc nat*)
then have *t1*: *t1* = *Node ts* **using** *trim-id nt1* **by** *blast*
then obtain *n2 t2* **where** *nt2*: *trim-tree n1 t* = (*n2*, *t2*) **by** *fastforce*
then show *?thesis*
proof (*cases r* ≤ *Node ts*)
case *True*
then show *?thesis* **using** *4 Suc nt1 t1*
by (*auto split*: *prod.split simp*: *tree-le-cons*, *meson dual-order.trans tree-le-cons*)
next
case *False*

```
    then obtain ss s where r: r = Node (ss @ s # ts) using 4(4) tree-ge-lt-suffix
      by (metis append.assoc append-Cons append-Nil nle-le rev-exhaust tree-le-def)
    have size-s: tree-size s ≤ Suc nat using 4(3) Suc size-trim-tree[OF - nt1] t1
unfolding r by auto
     have s ≤ t using 4(4) unfolding r by (meson order.trans tree-le-append
tree-le-cons2)
      have s ≤ t2 using 4.IH(2)[OF nt1[symmetric] Suc t1 size-s ‹s≤t›] nt2
unfolding Suc by auto
    then show ?thesis
    proof (cases s = t2)
      case True
      then have ss = []
      proof (cases t2 = t)
        case True
        then show ?thesis using 4(4) nle-le tree-le-append unfolding r ‹s=t2›
True by auto
      next
        case False
        then have n2 = 0 using nt2 trim-id by (cases n2) auto
        then show ?thesis using size-trim-tree[OF - nt1] size-trim-tree[OF - nt2]
Suc 4(3) tree-size-ne-0 unfolding r t1 ‹s=t2› by auto
      qed
      then show ?thesis using nt1 Suc t1 nt2 unfolding r True by auto
    next
      case False
      then show ?thesis using ‹s≤t2› nt1 nt2 t1 Suc unfolding r
        by (auto simp: order-less-imp-le tree-less-comm-suffix2)
    qed
  qed
 qed
qed

lemma fill-tree-next-smallest: tree-size (Node rs) ≤ Suc n ⟹ ∀r∈set rs. r ≤ t
⟹ Node rs ≤ Node (fill-tree n t)
proof (induction n t arbitrary: rs rule: fill-tree.induct)
  case (1 uu)
  have rs = [] using tree-size-1-iff 1(1) tree.inject by fastforce
  then show ?case by auto
next
  case (2 v t)
  obtain n' t' where nt': trim-tree (Suc v) t = (n', t') by fastforce
  then show ?case
  proof (cases rs rule: rev-exhaust)
    case Nil
    then show ?thesis by auto
  next
    case (snoc rs' r')
    then show ?thesis
    proof (cases n')
```

case *0*
then show *?thesis*
proof (*cases* $r' = t'$)
case *True*
then have $rs' = []$ **using** *0 2*(*2*) *size-trim-tree*[*OF - nt'*] **unfolding** *snoc* **by** (*auto simp*: *tree-size-ne-0*)
then show *?thesis* **using** *nt' 0* **unfolding** *snoc True* **by** *simp*
next
case *False*
then show *?thesis* **using** *2 trim-tree-greatest-le nt' 0 tree-less-comm-suffix2* **unfolding** *snoc*
by (*auto*, *metis nless-le not-less-eq-eq snd-eqD trans-le-add2*)
qed
next
case (*Suc nat*)
then show *?thesis* **using** *2 nt' trim-id*[*OF nt'*[*unfolded Suc*]] *size-trim-tree*[*OF - nt'*] **unfolding** *snoc* **by** *auto*
qed
qed
qed

fun *fill-twos* :: *nat* ⇒ *tree* ⇒ *tree* **where**
fill-twos n (*Node ts*) = *Node* (*replicate n* (*Node* []) @ *ts*)

lemma *size-fill-twos*: *tree-size* (*fill-twos n t*) = *n* + *tree-size t*
by (*cases t*) (*auto simp*: *sum-list-replicate*)

lemma *regular-fill-twos*: *regular t* ⟹ *regular* (*fill-twos n t*)
by (*cases t*) (*auto simp*: *sorted-append*)

lemma *fill-twos-lt*: $n \neq 0 \Longrightarrow t <$ *fill-twos n t*
using *tree-less-append* **by** (*cases t*) *auto*

lemma *fill-twos-less*: $r <$ *Node* (*t#ts*) ⟹ $t \neq$ *Node* [] ⟹ *fill-twos n r* < *Node* (*t#ts*)
proof (*induction n*)
case *0*
then show *?case* **by** (*cases r*) *auto*
next
case (*Suc n*)
then show *?case* **by** (*cases r rule*: *tree.exhaust*, *simp*,
meson leD linorder-less-linear list.inject tree.inject tree-empty-cons-lt-le)
qed

lemma *next-tree-aux-successor*: *tree-size r* = *tree-size t* + *n* ⟹ *regular r* ⟹ $r < t$ ⟹ *height t* ≥ *2* ⟹ $r \leq$ *the* (*next-tree-aux n t*)
proof (*induction n t arbitrary*: *r rule*: *next-tree-aux.induct*)
case (*1 n*)
then show *?case* **by** *auto*

next
  case (*2 n ts*)
  have *size-r*: *tree-size r* ≤ *tree-size* (*Node ts*) + *Suc n* **using** *2*(*2*) **by** *auto*
  have *height-ts*: *height* (*Node ts*) ≥ *2* **using** *2*(*5*) **by** (*cases ts*) *auto*
  then show *?case* **using** *2 size-r tree-empty-cons-lt-le* **by** *fastforce*
next
  case (*3 n rs ts*)
  then show *?case*
  proof (*cases r* < *Node ts*)
    case *True*
    then show *?thesis* **by** (*auto*, *meson dual-order.trans order.strict-implies-order tree-le-append tree-le-cons*)
  next
    case *False*
    then obtain *ss* **where** *r*: *r* = *Node* (*ss* @ *ts*) **using** *3*(*3*) *tree-ge-lt-suffix* **by** *fastforce*
    show *?thesis*
    proof (*cases ss rule*: *rev-exhaust*)
      case *Nil*
      then show *?thesis* **unfolding** *r* **by** (*simp*, *meson order-trans tree-le-append tree-le-cons*)
    next
      case (*snoc ss′ s′*)
      have *s′-le-rs*: *s′* ≤ *Node rs* **using** *3*(*3*) *tree-empty-cons-lt-le* **unfolding** *r snoc*
        by (*metis* (*mono-tags*, *lifting*) *append.assoc append-Cons append-self-conv2*
          dual-order.order-iff-strict linorder-not-less order-less-le-trans tree-le-append tree-less-cons2)
      show *?thesis*
      proof (*cases s′* = *Node rs*)
        case *True*
        then show *?thesis* **using** *3*(*1*,*2*) *fill-tree-next-smallest* **unfolding** *r snoc*
          by (*auto simp del*: *fill-tree.simps simp*: *sorted-append*)
      next
        case *False*
        then show *?thesis* **using** *s′-le-rs* **unfolding** *r snoc* **by** (*auto*, *meson tree-le-def tree-less-iff*)
      qed
    qed
  qed
next
  case (*4 n vc vd vb ts*)
  define *t* **where** *t* = *Node* (*Node* (*vc* # *vd*) # *vb*)
  have *height-t*: *height t* ≥ *2* **unfolding** *numeral-2-eq-2 t-def*
   by (*metis dual-order.antisym height-Node-cons less-eq-nat.simps*(*1*) *not-less-eq-eq*)
  then show *?case*
  proof (*cases r* < *Node ts*)
    case *True*
    then show *?thesis* **by** (*auto*, *meson dual-order.trans order.strict-implies-order tree-le-append tree-le-cons*)

next
case *False*
then obtain *ss* **where** *r*: *r* = *Node* (*ss* @ *ts*) **using** *4*(*4*) *tree-ge-lt-suffix* **by** *fastforce*
then show *?thesis*
proof (*cases ss rule*: *rev-exhaust*)
case *Nil*
then show *?thesis* **using** *tree-le-cons* **unfolding** *r* **by** *auto*
next
case (*snoc ss′ s′*)
have $s' < t$ **using** *4*(*4*)[*folded t-def*] **unfolding** *r snoc*
by (*auto*, *metis antisym-conv3 append.left-neutral dual-order.strict-trans less-tree-comm-suffix not-tree-less-empty tree-less-cons2*)
show *?thesis*
proof (*cases tree-size s′* = *tree-size t* + *n*)
case *True*
then have *ss′* = [] **using** *4*(*2*)[*folded t-def*] *tree-size-ne-0* **unfolding** *r snoc* **by** *auto*
then show *?thesis* **using** *4.IH True 4*(*3*) ‹*s′<t*› *height-t tree-le-cons2* **unfolding** *r snoc t-def* **by** *auto*
next
case *False*
obtain *us* **where** *s′*: *s′* = *Node us* **using** *tree.exhaust* **by** *blast*
— s” is greater than s’ but has the same size as t so the IH can be used on it.
define *s″* **where** *s″* = *fill-twos* (*tree-size t* + *n* − *tree-size s′*) *s′*
have *size-s′*: *tree-size s′* ≤ *tree-size t* + *n* **using** *4*(*2*)[*folded t-def*] **unfolding** *r snoc* **by** *simp*
then have *size-s″*: *tree-size s″* = *tree-size t* + *n* **unfolding** *s″-def* **using** *size-fill-twos* **by** *auto*
have *regular-s″*: *regular s″* **using** *regular-fill-twos 4*(*3*) **unfolding** *s″-def r snoc* **by** *auto*
have $s'' < t$ **using** *fill-twos-less* ‹*s′<t*› **unfolding** *t-def s″-def* **by** *auto*
have $s' < s''$ **using** *fill-twos-lt False size-fill-twos size-s″* **unfolding** *s″-def* **by** *auto*
then show *?thesis* **using** *4.IH*[*folded t-def*, *OF size-s″ regular-s″* ‹*s″<t*› *height-t*]
unfolding *r snoc t-def* **by** (*simp add*: *order-less-imp-le tree-less-comm-suffix2*)
qed
qed
qed
qed

lemma *next-tree-successor*: *tree-size r* = *tree-size t* ⟹ *regular r* ⟹ *r* < *t* ⟹ *next-tree t* = *Some t′* ⟹ *r* ≤ *t′*
using *next-tree-aux-successor next-tree-Some-iff* **by** *force*

lemma *set-n-tree-enum-aux*: *t* ∈ *regular-n-trees n* ⟹ *set* (*n-tree-enum-aux t*) = {*r*∈*regular-n-trees n*. *r* ≤ *t*}
proof (*induction t rule*: *n-tree-enum-aux.induct*)

case (*1 t*)
then show *?case*
proof (*cases next-tree t*)
case *None*
have *n* ≠ *0* **using** *1*(*2*) *tree-size-ne-0* **unfolding** *regular-n-trees-def n-trees-def*
by *auto*
have *t* = *least-tree n* **using** *height-lt-2-least-tree next-tree-aux-None-iff 1 None*
by *simp*
then show *?thesis* **using** *next-tree-Some-iff 1 None least-tree-le* ‹*n*≠*0*›
unfolding *regular-n-trees-def n-trees-def* **by** (*auto simp*: *antisym*)
next
case (*Some t′*)
then have *set* (*n-tree-enum-aux t*) = *insert t* {*r* ∈ *regular-n-trees n*. *r* ≤ *t′*}
using *1 regular-next-tree′ size-next-tree′* **unfolding** *regular-n-trees-def n-trees-def*
by *auto*
also have ... = {*r*∈*regular-n-trees n*. *r* ≤ *t*} **using** *next-tree-successor 1*(*2*)
Some **unfolding** *regular-n-trees-def n-trees-def*
by (*auto*, *meson Some less-le-not-le next-tree-lt′ order.trans*)
finally show *?thesis* **.**
qed
qed

theorem *set-n-tree-enum*: *set* (*n-tree-enum n*) = *regular-n-trees n*
proof (*cases n*)
case *0*
then show *?thesis* **unfolding** *regular-n-trees-def n-trees-def* **using** *tree-size-ne-0*
by *simp*
next
case (*Suc nat*)
then show *?thesis* **using** *set-n-tree-enum-aux regular-n-tree-greatest-tree greatest-tree-ge*
unfolding *regular-n-trees-def n-trees-def* **by** *auto*
qed

theorem *n-rtree-graph-enum-n-rtree-graphs*: *G* ∈ *set* (*n-rtree-graph-enum n*) ⟹
G ∈ *n-rtree-graphs n*
using *set-n-tree-enum rtree-tree-graph card-tree-graph*
unfolding *n-rtree-graph-enum-def n-rtree-graphs-def regular-n-trees-def n-trees-def*
by (*auto*, *metis*)

theorem *n-rtree-graph-enum-surj*:
assumes *n-rtree-graph*: *G* ∈ *n-rtree-graphs n*
shows ∃ *G′* ∈ *set* (*n-rtree-graph-enum n*). $G' \simeq_r G$
proof−
obtain *V E r* **where** *G* = (*V*,*E*,*r*) **using** *prod.exhaust* **by** *metis*
then show *?thesis* **using** *n-rtree-graph set-n-tree-enum rtree.ex-regular-n-tree*
unfolding *n-rtree-graphs-def n-rtree-graph-enum-def* **by** (*auto simp*: *rtree.ex-regular-n-tree*)
qed

4.7 Distinctness

lemma *n-tree-enum-aux-le*: $r \in set$ (*n-tree-enum-aux t*) $\Longrightarrow r \leq t$
proof (*induction t rule*: *n-tree-enum-aux.induct*)
  case (*1 t*)
  then show *?case*
  proof (*cases next-tree t*)
    case *None*
    then show *?thesis* **using** *1* **by** *auto*
  next
    case (*Some a*)
    then show *?thesis* **using** *next-tree-lt′ 1* **by** *fastforce*
  qed
qed

lemma *sorted-n-tree-enum-aux*: *sorted-wrt* (>) (*n-tree-enum-aux t*)
proof (*induction t rule*: *n-tree-enum-aux.induct*)
  case (*1 t*)
  then show *?case*
  proof (*cases next-tree t*)
    case *None*
    then show *?thesis* **by** *simp*
  next
    case (*Some a*)
    then show *?thesis* **using** *1 Some next-tree-lt′ n-tree-enum-aux-le* **by** *fastforce*
  qed
qed

lemma *distinct-n-tree-enum-aux*: *distinct* (*n-tree-enum-aux t*)
  using *sorted-n-tree-enum-aux strict-sorted-iff distinct-rev sorted-wrt-rev* **by** *blast*

theorem *distinct-n-tree-enum*: *distinct* (*n-tree-enum n*)
  using *distinct-n-tree-enum-aux* **by** (*cases n*) *auto*

theorem *distinct-n-rtree-graph-enum*: *distinct* (*n-rtree-graph-enum n*)
 using *tree-graph-inj distinct-n-tree-enum set-n-tree-enum* **unfolding** *n-rtree-graph-enum-def regular-n-trees-def*
  by (*simp add*: *distinct-map inj-on-def*)

theorem *inj-iso-n-rtree-graph-enum*:
  assumes *G-in-n-rtree-graph-enum*: $G \in set$ (*n-rtree-graph-enum n*)
    and *H-in-n-rtree-graph-enum*: $H \in set$ (*n-rtree-graph-enum n*)
    and $G \simeq_r H$
  shows $G = H$
proof−
 obtain t_G **where** *t-G*: *regular* t_G *tree-graph* $t_G = G$ **using** *G-in-n-rtree-graph-enum regular-n-tree-enum*
    unfolding *n-rtree-graph-enum-def* **by** *auto*
 obtain t_H **where** *t-H*: *regular* t_H *tree-graph* $t_H = H$ **using** *H-in-n-rtree-graph-enum regular-n-tree-enum*

unfolding *n-rtree-graph-enum-def* **by** *auto*
then show *?thesis* **using** *t-G tree-graph-inj-iso* ‹$G \simeq_r H$› **by** *auto*
qed

theorem *ex1-iso-n-rtree-graph-enum*: $G \in$ *n-rtree-graphs* $n \Longrightarrow \exists!G' \in$ *set* (*n-rtree-graph-enum* n). $G' \simeq_r G$
using *inj-iso-n-rtree-graph-enum rgraph-isomorph-trans rgraph-isomorph-sym n-rtree-graph-enum-surj*
unfolding *transp-def* **by** *blast*

end

References

[1] T. Beyer and S. M. Hedetniemi. Constant time generation of rooted trees. *SIAM Journal on Computing*, 9(4):706–712, 1980.