Tree-Enumeration

nils

March 8, 2023

Contents

1 Trees

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

1.1 Misc

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

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

lemma (**in** *ulgraph*) *union-loops-sedges*: *loops* $\cup$ *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*: $e \in E \Longrightarrow \exists\, v{\in}V.$ *incident v e*
using *edge-size wellformed* **by** (*metis empty-not-edge equals0I incident-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* :: $'a \Rightarrow\ 'a$ *pregraph* **where**
remove-vertex $v = (V - \{v\}, \{e{\in}E.\ \neg$ *incident v e*$\})$

1.2 Degree

lemma (**in** *ulgraph*) *empty-E-degree-0*: $E = \{\} \Longrightarrow$ *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*: $\bigwedge v.$ *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-loops-def g.incident-loops-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-def g.degree-def* **by** *auto*
have $g'.incident\text{-}loops\ u = g.incident\text{-}loops\ u \cup \{e\}$ **unfolding** g'*.incident-loops-def g.incident-loops-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-def g.degree-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-edge-pair-adj* **by** *fastforce*
then have *card-e*: $card\ e = 2$ **using** *False* g'*.alt-edge-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-loops-alt g.incident-loops-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-sedges-def g.incident-sedges-def g.incident-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-def g.degree-def* **using** *inc-loops* **by** *auto*
have $g'.incident\text{-}sedges\ u = g.incident\text{-}sedges\ u \cup \{e\}$
unfolding g'*.incident-sedges-def g.incident-sedges-def g.incident-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-def g.degree-def* **by** *auto*
have $g'.incident\text{-}sedges\ w = g.incident\text{-}sedges\ w \cup \{e\}$
unfolding g'*.incident-sedges-def g.incident-sedges-def g.incident-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-def g.degree-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

1.3 Walks

lemma (**in** *ulgraph*) *walk-edges-induced-edges*: *is-walk p* ⟹ *set* (*walk-edges p*) ⊆ *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* ∈ *set* (*walk-edges xs*) ⟹ *e* ⊆ *set xs*
by (*induction xs rule*: *walk-edges.induct*) *auto*

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

lemma (**in** *ulgraph*) *split-walk-edge*: {*x*,*y*} ∈ *set* (*walk-edges p*) ⟹
∃ *xs ys*. *p* = *xs* @ *x* # *y* # *ys* ∨ *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* ⟹ *set p* ⊆ *V*
unfolding *is-gen-path-def* **using** *is-walk-wf* **by** *auto*

lemma (**in** *ulgraph*) *path-wf*: *is-path p* ⟹ *set p* ⊆ *V*
by (*simp add*: *is-path-walk is-walk-wf*)

lemma (**in** *fin-ulgraph*) *length-gen-path-card-V*: *is-gen-path p* ⟹ *walk-length p* ≤ *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* ⟹ *length p* ≤ *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*) ⟹ *xs* ≠ [] ⟹ *is-gen-path* (*xs*)
unfolding *is-gen-path-def* **using** *is-walk-prefix* **apply** *auto*
by (*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 order*] **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 incident-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 $\forall$ *w*$\in$*set p*. *vert-connected u w*
proof
fix *w* **assume** $w \in$ *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 $\{v,u\} \in E$
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** $e \in$ *set* (*walk-edges p*) **using** ‹*u*≠*v*› *connecting-path-length-bound* **unfolding** *walk-length-def* **by** *fastforce*
then have $e \in E$ **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 $u \neq v$
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* $\geq$ *1* **using** ‹$u \neq v$› *connecting-path-length-bound* **by** *metis*
then have *length p* $\geq$ *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*: $\{v,w\} \in E$ **using** *conn-path* **unfolding** *connecting-path-def is-gen-path-def is-walk-def* **by** *simp*
then have $\{v,w\} \in$ *incident-edges v* **unfolding** *incident-edges-def incident-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* $\geq$ *2*
and *inV*: $v \in V$
shows *degree v* $\neq$ *0*
proof−
obtain u **where** $u \in V$ **and** $u \neq v$ **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*: $E = \{\} \Longrightarrow \exists\, v.\ V = \{v\}$
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*: $\{u,v\} \in E$
shows $\forall\, w \in V.$ *ulgraph.vert-connected* V $(E - \{\{u,v\}\})$ w u $\vee$ *ulgraph.vert-connected* V $(E - \{\{u,v\}\})$ w v
proof
fix w **assume** $w \in V$
interpret g': *ulgraph* V $E - \{\{u,v\}\}$ **using** *wellformed edge-size* **by** (*unfold-locales*, *auto*)
have *inV*: $u \in V$ $v \in V$ **using** *e wellformed* **by** *auto*
obtain *p* **where** *conn-path*: *connecting-path w v p* **using** *connected inV* ‹$w \in V$› **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*} ∈ *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* ∈ *set p* **using** *walk-edge walk-edges-in-verts* **by** *blast*
obtain *xs ys* **where** *p-split*: *p* = *xs* @ *u* # *v* # *ys* ∨ *p* = *xs* @ *v* # *u* # *ys* **using** *split-walk-edge*[*OF walk-edge*] **by** *blast*
have *v-notin-ys*: *v* ∉ *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* ∉ *set* (*xs*@[*u*]) **using** *p distinct* **by** *auto*
then have {*u*,*v*} ∉ *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*} ∉ *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*: $\{u,v\} \in E$ **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 incident-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 incident-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 ∈ 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 xs' x where xs: xs = xs'@[x] by (meson rev-exhaust)
  then have x ≠ l using distinct p by simp
  have {x,l} ∈ set (walk-edges p) using conn-path walk-edges-append-union p
xs
  by (smt (verit) Un-insert-right ‹xs ≠ []› comp-sgraph.walk-edges-append-union
insert-iff
      last-snoc list.discI list.sel(1))
  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
incident-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
incident-def by auto

   have card-loops: card (incident-loops l) = 0 using degree unfolding de-
gree-def by auto
  have x ≠ y using distinct unfolding p xs ys by simp
  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
```

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* $\Rightarrow$ *'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* ∈ *induced-edges C* **if** *e* ∈ *E* **for** *e*
proof –
obtain *u v* **where** *e*: *e* = {*u*,*v*} **by** (*meson* ‹*e* ∈ *E*› *obtain-edge-pair-adj*)
then have *vert-connected u v* **using** *that vert-connected-neighbors* **by** *blast*
then have *v* ∈ *connected-component-of u* **unfolding** *connected-component-of-def* **by** *simp*
then have *e* ∈ *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*} | *v*. *v*∈*V*}
proof –
have ∀ *v*∈*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* ≠ {}
shows *is-connected-set V* ⟷ *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*: $u \in V$ $v \in V$ **using** *edge wellformed* **by** *auto*
have $\forall\, w{\in}V.$ g'*.connected-component-of w* = g'*.connected-component-of u* $\vee$ 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*: $C \neq \{\}$
and $\bigwedge u\ v.\ \{u,v\} \in E \Longrightarrow u \in C \Longrightarrow v \in C$
shows $C \in$ *connected-components*
proof −
have *walk-subset-C*: *is-walk xs* $\Longrightarrow$ *hd xs* $\in C \Longrightarrow$ *set xs* $\subseteq C$ **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* $\subseteq$ *C* **using** *snoc xs is-walk-not-empty2 hd-append2* **by** *metis*
    have *yx-E*: $\{y,x\} \in E$ **using** *snoc*(*2*) *walk-edges-app* **unfolding** *xs is-walk-def* **by** *simp*
    have $x \in C$ **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** $u \in C$ **using** *non-empty* **by** *blast*
then have $u \in V$ **using** *conn-set is-connected-set-wf* **by** *blast*
have $v \in C$ **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*] ‹$u \in C$› *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* ‹$u \in C$› **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 have $C = \{\}$ **using** *A-subset-conn-comps Un-A connected-components-partition-on-V* **unfolding** *partition-on-def*
    by (*auto*, *smt* (*verit*, *best*) *UnionE UnionI disjnt-iff pairwise-def subset-iff*)
  then show *False* **using** *connected-components-partition-on-V C-conn-comp* **unfolding** *partition-on-def* **by** *blast*
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 incident-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 $u \in C$ **using** ‹$c{\in}C$› *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** ‹$u{\in}C$› **by** *blast*
qed

1.9 Trees

locale *tree* = *fin-connected-ulgraph* +
assumes *no-cycles*: $\neg$ *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 = fin-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.fin-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. 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 incident-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 incident-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 l* **unfolding** *leaf-def alt-degree-def* **by** *simp*
then show *?thesis* **..**
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 incident-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 incident-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 incident-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'≠{}› *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.incident-def t.singleton-not-edge t.wellformed*
by *fastforce*
then show *?thesis* **using** *trivial t.tree-axioms V* **by** *simp*
next
case *False*
thm *graph-system.incident-edges-def*
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.incident-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* ‹$u \in V$› ‹$l \neq u$› ‹$l \in V$› *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 span-
ning-tree-def by fastforce
  qed
qed

context tree
begin

lemma add-vertex-tree:
  assumes v ∉ V
    and w ∈ 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 (*auto*, *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 ulgraph-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 $\ldots = \{v,w\} \#$ *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* $\Longrightarrow$ *card E* ≥ 1
using *card-V-card-E* **unfolding** *non-trivial-def* **by** *simp*

lemma *V-Union-E*: *non-trivial* $\Longrightarrow V = \bigcup E$
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

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*: $((\text{'})\ f)$ ' $E_G = E_H$
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* ≡ *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 $\forall e \in E_G$. (λv. *the-inv-into* V_G f $(f\ v)$) ' $e = e$ **using** *G.wellformed*
by (*simp add*: *subset-iff*)

then show $((`)\ inv\text{-}iso)`\ E_H = E_G$ **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* ⇒ *'b pregraph* ⇒ *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* (≃)
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* (≃)
using *graph-isomorphism-trans* **unfolding** *transp-def* **by** *fastforce*

end

2 Labeled Trees

theory *Labeled-Tree-Enumeration*
imports *Tree-Graph Combinatorial-Enumeration-Algorithms.n-Sequences*
begin

2.1 Definition

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

2.2 Algorithm

Prüfer sequence to tree

definition *prufer-sequences* :: *'a list* ⇒ *'a list set* **where**
prufer-sequences verts = *n-sequences* (*set verts*) (*length verts* − *2*)

fun *prufer-seq-to-tree-edges* :: $'a$ *list* $\Rightarrow$ $'a$ *list* $\Rightarrow$ $('a \times 'a)$ *list* **where**
prufer-seq-to-tree-edges $[v,w]$ $[]$ $=$ $[(v,w)]$
| *prufer-seq-to-tree-edges verts* $(a\#seq)$ $=$
 (*case find* $(\lambda x.\ x \notin$ *set* $(a\#seq))$ *verts of*
 Some $b \Rightarrow (a,b)$ $\#$ *prufer-seq-to-tree-edges* (*remove1 b verts*) *seq*)

definition *edges-of-edge-list* :: $('a \times 'a)$ *list* $\Rightarrow$ $'a$ *edge set* **where**
edges-of-edge-list edge-list $\equiv$ *mk-edge* ‘ *set edge-list*

definition *prufer-seq-to-tree* :: $'a$ *list* $\Rightarrow$ $'a$ *list* $\Rightarrow$ $'a$ *pregraph* **where**
prufer-seq-to-tree verts seq $=$ (*set verts*, *edges-of-edge-list* (*prufer-seq-to-tree-edges verts seq*))

definition *labeled-tree-enum* :: $'a$ *list* $\Rightarrow$ $'a$ *pregraph list* **where**
labeled-tree-enum verts $=$ *map* (*prufer-seq-to-tree verts*) (*n-sequence-enum verts* (*length verts* $-$ *2*))

2.3 Correctness

Tree to Prüfer sequence

definition *incident-edges* :: $'a \Rightarrow ('a \times 'a)$ *list* $\Rightarrow$ $('a \times 'a)$ *list* **where**
incident-edges v edge-list $=$ *filter* $(\lambda(u,w).\ u = v \vee w = v)$ *edge-list*

abbreviation *degree v edge-list* $\equiv$ *length* (*incident-edges v edge-list*)

fun *neighbor* :: $'a \Rightarrow ('a \times 'a)$ *list* $\Rightarrow$ $'a$ **where**
neighbor v $[]$ $=$ *undefined*
| *neighbor v* $((u,w)\#edges)$ $=$ (*if* $v = u$ *then* w *else if* $v = w$ *then* u *else neighbor v edges*)

definition *remove-vertex* :: $'a \Rightarrow ('a \times 'a)$ *list* $\Rightarrow$ $('a \times 'a)$ *list* **where**
remove-vertex v $=$ *filter* $(\lambda(u,w).\ u \neq v \wedge w \neq v)$

lemma *find-in-list*[*termination-simp*]: *find P verts* $=$ *Some v* $\Longrightarrow$ $v \in$ *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 *tree-to-prufer-seq* :: $'a$ *list* $\Rightarrow$ $('a \times 'a)$ *list* $\Rightarrow$ $'a$ *list* **where**
tree-to-prufer-seq verts $[]$ $=$ *undefined*
| *tree-to-prufer-seq verts* $[(u,w)]$ $=$ $[]$
| *tree-to-prufer-seq verts edges* $=$
 (*case find* $(\lambda v.$ *degree v edges* $=$ *1*$)$ *verts of*
 Some leaf $\Rightarrow$ *neighbor leaf edges* $\#$ *tree-to-prufer-seq* (*remove1 leaf verts*) (*remove-vertex leaf edges*))

lemma *remove-vertex*: *edges-of-edge-list* (*remove-vertex v edge-list*) = {*e* ∈ *edges-of-edge-list edge-list*. *v* ∉ *e*}
  unfolding *remove-vertex-def* **by** (*auto simp*: *edges-of-edge-list-def*)

lemma *neighbor-ne*: ∀(*u*,*w*)∈*set edge-list*. *u* ≠ *w* ⟹ *degree v edge-list* ≥ *1* ⟹ *neighbor v edge-list* ≠ *v*
  unfolding *incident-edges-def* **by** (*induction edge-list rule*: *neighbor.induct*) *auto*

lemma *degree-remove-vertex-0*[*simp*]: *degree v* (*remove-vertex v edge-list*) = *0*
  unfolding *incident-edges-def remove-vertex-def*
  by (*smt* (*verit*, *best*) *filter-False list.size*(*3*) *mem-Collect-eq set-filter split-def*)

lemma *degree-0-remove-vertex*:
  assumes *degree-0*: *degree v edge-list* = *0*
  shows *remove-vertex v edge-list* = *edge-list*
proof−
  have ∀(*u*,*w*) ∈ *set edge-list*. *u* ≠ *v* ∧ *w* ≠ *v* **using** *degree-0* **unfolding** *incident-edges-def*
    by (*simp add*: *filter-empty-conv split-def*)
  then show *?thesis* **unfolding** *remove-vertex-def* **by** *simp*
qed

lemma *degree-length-filter*: *degree v edge-list* = *length* (*filter* (λ*e*. *v* ∈ *e*) (*map mk-edge edge-list*))
proof−
  have (λ(*u*, *w*). *u* = *v* ∨ *w* = *v*) = (∈) *v* ∘ *mk-edge* **by** *auto*
  then have *1*: *map mk-edge* (*filter* (λ(*u*, *w*). *u* = *v* ∨ *w* = *v*) *edge-list*) = *filter* ((∈) *v*) (*map mk-edge edge-list*) **using** *filter-map* **by** *metis*
  have *length* (*filter* (λ(*u*, *w*). *u* = *v* ∨ *w* = *v*) *edge-list*) = *length* (*map mk-edge* (*filter* (λ(*u*, *w*). *u* = *v* ∨ *w* = *v*) *edge-list*)) **by** *simp*
  then show *?thesis* **unfolding** *incident-edges-def* **using** *1* **by** *argo*
qed

lemma *degree-neighbor-remove-vertex*: *degree v edge-list* = *1* ⟹ *Suc* (*degree* (*neighbor v edge-list*) (*remove-vertex v edge-list*)) = *degree* (*neighbor v edge-list*) *edge-list*
proof (*induction v edge-list rule*: *neighbor.induct*)
  case (*1 v*)
  then show *?case* **unfolding** *incident-edges-def remove-vertex-def* **by** *simp*
next
  case (*2 v u w edges*)
  assume *degree-1*: *degree v* ((*u*, *w*) # *edges*) = *1*
  consider *u* = *v* ∧ *w* = *v* | *u* ≠ *v* ∧ *w* = *v* | *u* = *v* ∧ *w* ≠ *v* | *u* ≠ *v* ∧ *w* ≠ *v* **by** *blast*
  then show *?case*
  proof *cases*
    case *1*
    then show *?thesis* **using** *2* **by** *simp*
  next
    case *2*

then have *degree v edges* = *0* **using** *degree-1* **unfolding** *incident-edges-def* **by** *auto*
then show *?thesis* **using** *2 degree-0-remove-vertex* **unfolding** *remove-vertex-def incident-edges-def* **by** *fastforce*
next
case *3*
then have *degree v edges* = *0* **using** *degree-1* **unfolding** *incident-edges-def* **by** *auto*
then show *?thesis* **using** *3 degree-0-remove-vertex* **unfolding** *remove-vertex-def incident-edges-def* **by** *fastforce*
next
case *4*
then have *degree v edges* = *1* **using** *2*(*2*) **unfolding** *incident-edges-def* **by** *auto*
then show *?thesis* **using** *4 2.IH* **unfolding** *remove-vertex-def incident-edges-def* **by** *auto*
qed
qed

lemma *distinct-remove-vertex*[*simp*]: *distinct* (*map mk-edge edge-list*) ⟹ *distinct* (*map mk-edge* (*remove-vertex leaf edge-list*))
unfolding *remove-vertex-def* **using** *distinct-map-filter* **by** *fast*

lemma *neighbor-edge-in-edges*: *degree v edge-list* ≥ *1* ⟹ {*neighbor v edge-list*, *v*} ∈ *edges-of-edge-list edge-list*
unfolding *incident-edges-def edges-of-edge-list-def* **by** (*induction v edge-list rule*: *neighbor.induct*) *auto*

lemma *insert-remove-leaf*:
assumes *degree-leaf*: *degree leaf edge-list* = *1*
shows *insert* {*neighbor leaf edge-list*, *leaf*} (*edges-of-edge-list* (*remove-vertex leaf edge-list*)) = *edges-of-edge-list edge-list*
proof−
let *?leaf-edges* = *filter* (λ(*u*,*w*). *u* = *leaf* ∨ *w* = *leaf*) *edge-list*
have *length-leaf-edges*: *length ?leaf-edges* = *1* **using** *degree-leaf* **unfolding** *incident-edges-def* **by** *simp*
have {*neighbor leaf edge-list*, *leaf*} ∈ *edges-of-edge-list edge-list* **using** *neighbor-edge-in-edges degree-leaf* **by** *force*
then have (*neighbor leaf edge-list*, *leaf*) ∈ *set edge-list* ∨ (*leaf*, *neighbor leaf edge-list*) ∈ *set edge-list* **by** (*simp add*: *edges-of-edge-list-def in-mk-uedge-img-iff*)
then have (*neighbor leaf edge-list*, *leaf*) ∈ *set ?leaf-edges* ∨ (*leaf*, *neighbor leaf edge-list*) ∈ *set ?leaf-edges* **by** *simp*
then have *?leaf-edges* = [(*neighbor leaf edge-list*, *leaf*)] ∨ *?leaf-edges* = [(*leaf*, *neighbor leaf edge-list*)] **using** *length-leaf-edges*
by (*smt* (*verit*) *One-nat-def empty-iff empty-set length-0-conv length-Suc-conv list.inject list.set-cases*)
then have *leaf-edges*: *edges-of-edge-list ?leaf-edges* = {{*neighbor leaf edge-list*, *leaf*}} **unfolding** *edges-of-edge-list-def* **by** *fastforce*

have *edges-of-edge-list edge-list* = *edges-of-edge-list ?leaf-edges* ∪ *edges-of-edge-list* (*remove-vertex leaf edge-list*) **unfolding** *remove-vertex-def edges-of-edge-list-def* **by** *auto*
then show *?thesis* **using** *leaf-edges* **by** *auto*
qed

lemma *find-Some*: *find P xs* = *Some x* ⟹ *P x*
by (*metis find-Some-iff*)

definition *verts-of-edges* :: (′*a* × ′*a*) *list* ⇒ ′*a set* **where**
verts-of-edges edges = {*v* | *v e*. *v* ∈ *e* ∧ *e* ∈ *edges-of-edge-list edges*}

locale *prufer-seq-to-tree-context* =
fixes *verts* :: ′*a list*
assumes *verts-length*: *length verts* ≥ *2*
and *distinct-verts*: *distinct verts*
begin

lemma *card-verts*: *card* (*set verts*) ≥ *2*
using *verts-length distinct-verts distinct-card* **by** *fastforce*

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 *obtain-b-prufer-seq-to-tree-edges*:
assumes (*a* # *seq*) ∈ *prufer-sequences verts*
obtains *b*
where *find* (λ*x*. *x* ∉ *set* (*a* # *seq*)) *verts* = *Some b*
and *b* ∈ *set verts*
and *b* ∉ *set* (*a* # *seq*)
and *seq* ∈ *prufer-sequences* (*remove1 b verts*)
and *length* (*remove1 b verts*) ≥ *2*
and *distinct* (*remove1 b verts*)
proof−
obtain *b* **where** *b-find*: *find* (λ*x*. *x* ∉ *set* (*a*#*seq*)) *verts* = *Some b*
using *assms length-gt-find-not-in-ys*[*of a*#*seq verts*] *distinct-verts*
unfolding *prufer-sequences-def n-sequences-def*
by *fastforce*
have *b-in-verts*: *b* ∈ *set verts* **using** *b-find*

```
    by (metis find-Some-iff nth-mem)
  have b-not-in-seq: b ∉ set (a#seq) using b-find
    by (metis find-Some-iff)
  have seq-prufer-verts': seq ∈ prufer-sequences (remove1 b verts)
    using assms b-in-verts set-remove1-eq verts-length b-not-in-seq distinct-verts
    unfolding prufer-sequences-def n-sequences-def
    by (auto simp: length-remove1)
 have length verts ≥ 3 using assms unfolding prufer-sequences-def n-sequences-def
by auto
 then have length-verts': length (remove1 b verts) ≥ 2 by (auto simp: length-remove1)
   have distinct: distinct (remove1 b verts) using distinct-remove1 assms dis-
tinct-verts by fast
  from b-find b-in-verts b-not-in-seq seq-prufer-verts' length-verts' distinct show
?thesis ..
qed

lemma verts-of-edges-prufer-to-tree[simp]:
  seq ∈ prufer-sequences verts ⟹
    verts-of-edges (prufer-seq-to-tree-edges verts seq) = set verts
  using verts-length distinct-verts
proof (induction verts seq rule: prufer-seq-to-tree-edges.induct)
  case (1 v w)
  then show ?case unfolding verts-of-edges-def edges-of-edge-list-def by auto
next
  case (2 verts a seq)
  then interpret contxt: prufer-seq-to-tree-context verts by unfold-locales
  obtain b
    where b-find: find (λx. x ∉ set (a # seq)) verts = Some b
      and seq-in-verts': seq ∈ prufer-sequences (remove1 b verts)
      and len-verts': 2 ≤ length (remove1 b verts)
      and distinct-verts': distinct (remove1 b verts)
      and b-in-verts: b ∈ set verts
    using contxt.obtain-b-prufer-seq-to-tree-edges 2 by metis
  then have verts-of-edges (prufer-seq-to-tree-edges verts (a # seq))
    = verts-of-edges ((a,b) # prufer-seq-to-tree-edges (remove1 b verts) seq)
    by auto
 also have ... = {a,b} ∪ verts-of-edges (prufer-seq-to-tree-edges (remove1 b verts)
seq)
    unfolding verts-of-edges-def edges-of-edge-list-def by auto
  also have ... = {a,b} ∪ (set verts − {b}) using 2.IH[OF b-find seq-in-verts'
len-verts' distinct-verts'] b-in-verts by fastforce
 also have ... = set verts using 2.prems(1) b-in-verts unfolding prufer-sequences-def
n-sequences-def by auto
  finally show ?case .
qed (auto simp: prufer-sequences-def n-sequences-def)

lemma prufer-seq-to-tree-edges-tree:
  assumes seq ∈ prufer-sequences verts
  shows tree (verts-of-edges (prufer-seq-to-tree-edges verts seq)) (edges-of-edge-list
```

```
(prufer-seq-to-tree-edges verts seq))
    (is tree (?V verts seq) (?E verts seq))
  using assms verts-length distinct-verts
proof(induction verts seq rule: prufer-seq-to-tree-edges.induct)
  case (1 v w)
  have [simp]: verts-of-edges [(v,w)] = {v,w}
    unfolding verts-of-edges-def edges-of-edge-list-def using 1 by auto

  interpret ulgraph ?V [v,w] [] ?E [v,w] []
   by (unfold-locales, auto simp: card-insert-if verts-of-edges-def edges-of-edge-list-def)

  have connecting-path v w [v,w]
    unfolding connecting-path-def  is-gen-path-def is-walk-def
    by (auto simp: verts-of-edges-def edges-of-edge-list-def)
  then have vert-connected v w vert-connected w v
    unfolding vert-connected-def using connecting-path-rev by auto
  then have connected: is-connected-set (?V [v,w] [])
    unfolding is-connected-set-def using vert-connected-id by auto
   then have fin-connected-ulgraph: fin-connected-ulgraph (?V [v,w] []) (?E [v,w]
[])
    using 1 unfolding verts-of-edges-def edges-of-edge-list-def by (unfold-locales,
auto)

  then show ?case using fin-connected-ulgraph 1 unfolding edges-of-edge-list-def
by (auto intro: card-E-treeI)
next
  case (2 verts a seq)
  then interpret contxt: prufer-seq-to-tree-context verts by unfold-locales
  obtain b
    where b-find: find (λx. x ∉ set (a # seq)) verts = Some b
      and b-in-verts: b ∈ set verts
      and b-notin-seq: b ∉ set (a # seq)
      and seq-pruf-verts': seq ∈ prufer-sequences (remove1 b verts)
      and length-verts': length (remove1 b verts) ≥ 2
      and distinct-verts': distinct (remove1 b verts)
    using contxt.obtain-b-prufer-seq-to-tree-edges 2 by metis
  then interpret tree': tree ?V (remove1 b verts) seq ?E (remove1 b verts) seq
    using 2 seq-pruf-verts' distinct-remove1 b-find b-in-verts by auto

  interpret contxt': prufer-seq-to-tree-context remove1 b verts using length-verts'
distinct-verts' by unfold-locales

  have V'[simp]: ?V (remove1 b verts) seq = set verts − {b}
     using contxt'.verts-of-edges-prufer-to-tree seq-pruf-verts' set-remove1-eq 2(4)
by metis
  have V-V': ?V verts (a # seq) = insert b (?V (remove1 b verts) seq)
    using contxt.verts-of-edges-prufer-to-tree 2 V' b-in-verts by blast
  have edges: ?E verts (a # seq) = insert {a,b} (?E (remove1 b verts) seq)
    unfolding edges-of-edge-list-def using b-find by simp
```

have *b-notin-V'*: $b \notin$ *?V* (*remove1 b verts*) *seq* **using** *V'* **by** *blast*
have *a-in-V'*: $a \in$ *?V* (*remove1 b verts*) *seq*
using *V' b-notin-seq 2*(*2*) **unfolding** *prufer-sequences-def n-sequences-def* **by** *auto*

show *?case* **using** *V-V' edges tree'.add-vertex-tree*[*OF b-notin-V' a-in-V'*] *insert-commute* **by** *metis*
qed (*auto simp*: *prufer-sequences-def n-sequences-def*)

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

lemma *labeled-tree-enum-tree*: (*V*,*E*) ∈ *set* (*labeled-tree-enum verts*) ⟹ *tree V E*
using *prufer-seq-to-tree-tree n-sequence-enum-correct* **unfolding** *labeled-tree-enum-def prufer-sequences-def* **by** *fastforce*

lemma *prufer-seq-to-tree-edges-wf*:
assumes *pruf-seq*: *seq* ∈ *prufer-sequences verts*
and *edge*: *e* ∈ *edges-of-edge-list* (*prufer-seq-to-tree-edges verts seq*)
shows *e* ⊆ *set verts*
using *prufer-seq-to-tree-context-axioms assms*
proof (*induction seq arbitrary*: *verts*)
case *Nil*
then interpret *prufer-seq-to-tree-context verts* **by** *simp*
obtain *u v* **where** *verts* = [*u*,*v*] **using** *Nil verts-length* **unfolding** *prufer-sequences-def n-sequences-def* **apply** *auto*
by (*metis* (*no-types*, *opaque-lifting*) *One-nat-def Suc-1 length-0-conv length-Suc-conv*)
then show *?case* **using** *Nil* **unfolding** *edges-of-edge-list-def* **by** *simp*
next
case (*Cons a seq*)
then interpret *prufer-seq-to-tree-context verts* **by** *simp*
obtain *leaf* **where** *find-leaf*: *find* (λ*v*. *v* ∉ *set* (*a*#*seq*)) *verts* = *Some leaf*
and *pruf-seq'*: *seq* ∈ *prufer-sequences* (*remove1 leaf verts*)
and *leaf-in-verts*: *leaf* ∈ *set verts*
and *length* (*remove1 leaf verts*) ≥ *2*
and *distinct* (*remove1 leaf verts*) **using** *Cons obtain-b-prufer-seq-to-tree-edges* **by** *blast*
then have *contxt'*: *prufer-seq-to-tree-context* (*remove1 leaf verts*) **by** (*unfold-locales*, *simp*)
have *a-in-verts*: *a* ∈ *set verts* **using** *Cons*(*3*) **unfolding** *prufer-sequences-def n-sequences-def* **by** *simp*
show *?case* **using** *Cons*(*4*) *Cons.IH*[*OF contxt' pruf-seq'*] *find-leaf a-in-verts leaf-in-verts*
unfolding *edges-of-edge-list-def* **by** (*auto*, (*meson in-mk-uedge-img-iff notin-set-remove1*)+)
qed

lemma *distinct-prufer-seq-to-tree*: *seq* ∈ *prufer-sequences verts* ⟹ *distinct* (*map mk-edge* (*prufer-seq-to-tree-edges verts seq*))
using *prufer-seq-to-tree-context-axioms*
proof (*induction seq arbitrary*: *verts*)
case *Nil*
then interpret *prufer-seq-to-tree-context verts* **by** *simp*
obtain *u v* **where** *verts* = [*u*,*v*] **using** *Nil verts-length* **unfolding** *prufer-sequences-def n-sequences-def* **apply** *auto*
by (*metis* (*no-types*, *opaque-lifting*) *One-nat-def Suc-1 length-0-conv length-Suc-conv*)
then show *?case* **by** *auto*
next
case (*Cons a seq*)
then interpret *prufer-seq-to-tree-context verts* **by** *simp*
obtain *leaf* **where** *find-leaf*: *find* (λ*v*. *v* ∉ *set* (*a*#*seq*)) *verts* = *Some leaf*
and *pruf-seq′*: *seq* ∈ *prufer-sequences* (*remove1 leaf verts*)
and *length* (*remove1 leaf verts*) ≥ *2*
and *distinct* (*remove1 leaf verts*) **using** *Cons obtain-b-prufer-seq-to-tree-edges* **by** *blast*
then interpret *contxt′*: *prufer-seq-to-tree-context remove1 leaf verts* **by** (*unfold-locales*, *simp*)
have *leaf* ∉ *set* (*remove1 leaf verts*) **using** *distinct-verts set-remove1-eq* **by** *simp*
then have {*a*, *leaf*} ∉ *edges-of-edge-list* (*prufer-seq-to-tree-edges* (*remove1 leaf verts*) *seq*)
using *contxt′.prufer-seq-to-tree-edges-wf pruf-seq′* **by** *blast*
then show *?case* **using** *find-leaf Cons pruf-seq′ contxt′.prufer-seq-to-tree-context-axioms*
unfolding *edges-of-edge-list-def* **by** *simp*
qed

end

locale *tree-to-prufer-seq-context* =
fixes *verts* :: *′a list*
and *edge-list* :: (*′a* × *′a*) *list*
assumes *distinct-verts*: *distinct verts*
and *card-V*: *card* (*set verts*) ≥ *2*
and *tree*: *tree* (*set verts*) (*edges-of-edge-list edge-list*)
and *distinct-edges*: *distinct* (*map mk-edge edge-list*)
begin

sublocale *t*: *tree set verts edges-of-edge-list edge-list* **using** *tree* **.**

lemma *non-trivial*: *t.non-trivial*
using *card-V* **unfolding** *t.non-trivial-def* **.**

lemma *length-verts*: *length verts* ≥ *2*
using *card-V distinct-verts distinct-card* **by** *fastforce*

sublocale *prufer-seq-to-tree-context verts* **using** *length-verts distinct-verts prufer-seq-to-tree-context.intro* **by** *blast*

lemma *edge-ne*: (*u*,*v*) ∈ *set edge-list* ⟹ *u* ≠ *v*
using *t.two-edges tree* **unfolding** *edges-of-edge-list-def* **by** *fastforce*

lemma *distinct-edge-list*: *distinct edge-list*
using *distinct-edges* **by** (*simp add*: *distinct-map*)

lemma *length-varts-edge-list*: *length verts* = *Suc* (*length edge-list*)
using *distinct-verts t.card-V-card-E distinct-card distinct-edges edges-of-edge-list-def length-map list.set-map* **by** *metis*

lemma *incident-edges-correct*: *edges-of-edge-list* (*incident-edges v edge-list*) = *t.incident-edges v*
unfolding *t.incident-edges-def t.incident-def* **by** (*auto simp*: *edges-of-edge-list-def incident-edges-def*)

lemma *degree-correct*: *degree v edge-list* = *t.degree v*
proof−
have *distinct-incident-edges*: *distinct* (*map mk-edge* (*incident-edges v edge-list*)) **unfolding** *incident-edges-def* **using** *distinct-map-filter distinct-edges* **by** *blast*
have *degree v edge-list* = *length* (*map mk-edge* (*incident-edges v edge-list*)) **using** *distinct-edges* **by** *simp*
also have ... = *card* (*edges-of-edge-list* (*incident-edges v edge-list*)) **unfolding** *edges-of-edge-list-def* **using** *distinct-incident-edges distinct-card* **by** *fastforce*
also have ... = *card* (*t.incident-edges v*) **using** *incident-edges-correct* **by** *simp*
finally show *?thesis* **by** *simp*
qed

lemma *obtain-leaf-tree-to-prufer-seq*:
assumes *length-edge-list*: *length edge-list* ≥ *2*
obtains *leaf*
where *find* (*λv. degree v edge-list* = *1*) *verts* = *Some leaf*
and *t.leaf leaf*
and *leaf* ∈ *set verts*
and *tree-to-prufer-seq-context* (*remove1 leaf verts*) (*remove-vertex leaf edge-list*)
proof−
obtain *leaf* **where** *leaf-find*: *find* (*λv. degree v edge-list* = *1*) *verts* = *Some leaf*
using *find-None-iff2 t.leaf-in-V degree-correct t.leaf-def t.exists-leaf non-trivial* **by** *fastforce*
then have *degree leaf edge-list* = *1*
by (*metis* (*mono-tags*, *lifting*) *find-Some-iff*)
then have *leaf*: *t.leaf leaf* **using** *degree-correct t.leaf-def* **by** *auto*
have *in-verts*: *leaf* ∈ *set verts* **by** (*simp add*: *leaf t.leaf-in-V*)
let *?verts′* = *remove1 leaf verts*
let *?edge-list′* = *remove-vertex leaf edge-list*
have *distinct-verts′*: *distinct ?verts′* **using** *distinct-verts distinct-remove1* **by** *auto*
have *card* (*edges-of-edge-list edge-list*) ≥ *2* **unfolding** *edges-of-edge-list-def* **using** *length-edge-list distinct-edges distinct-card* **by** *fastforce*
then have *card* (*set verts*) ≥ *3* **using** *t.card-V-card-E* **by** *simp*

then have *card-verts'*: *card* (*set ?verts'*) $\geq$ *2* **by** (*simp add*: *distinct-verts in-verts*)
then interpret *t'*: *tree set ?verts' edges-of-edge-list ?edge-list'*
using *t.tree-remove-leaf leaf tree distinct-verts* **by** (*auto simp*: *remove-vertex t.remove-vertex-def t.incident-def*)
have *distinct-edges'*: *distinct* (*map mk-edge ?edge-list'*) **using** *distinct-edges distinct-remove-vertex* **by** *simp*
then have *tree-to-prufer-seq-context ?verts' ?edge-list'* **using** *distinct-verts' card-verts'* **by** (*unfold-locales*, *auto*)
then show *?thesis* **using** *that leaf-find leaf in-verts* **by** *auto*
qed

lemma *length-edge-list*: *length edge-list* $\geq$ *1*
proof−
have *length edge-list* = *card* (*edges-of-edge-list edge-list*) **unfolding** *edges-of-edge-list-def* **using** *distinct-edges distinct-card* **by** *force*
then show *?thesis* **using** *t.card-V-card-E length-verts distinct-verts distinct-card* **by** *fastforce*
qed

lemma *pruf-seq-tree-to-prufer-seq*: *tree-to-prufer-seq verts edge-list* $\in$ *prufer-sequences verts*
using *tree-to-prufer-seq-context-axioms*
proof (*induction verts edge-list rule*: *tree-to-prufer-seq.induct*)
case (*1 verts*)
then interpret *contxt*: *tree-to-prufer-seq-context verts* []
using *tree-to-prufer-seq-context.intro* **by** *blast*
show *?case* **using** *contxt.length-edge-list* **by** *auto*
next
case (*2 verts u w*)
then interpret *contxt*: *tree-to-prufer-seq-context verts* [(*u*,*w*)]
using *tree-to-prufer-seq-context.intro* **by** *blast*
show *?case* **using** *contxt.length-varts-edge-list* **unfolding** *prufer-sequences-def n-sequences-def* **by** *auto*
next
case (*3 verts e1 e2 edges*)
let *?edge-list* = *e1#e2#edges*
interpret *contxt*: *tree-to-prufer-seq-context verts ?edge-list*
using *tree-to-prufer-seq-context.intro 3* **by** *blast*
have *length-edge-list*: *length ?edge-list* $\geq$ *2* **by** *simp*
then obtain *leaf*
where *find-leaf*: *find* (λv. *degree v ?edge-list* = *1*) *verts* = *Some leaf*
and *contxt'*: *tree-to-prufer-seq-context* (*remove1 leaf verts*) (*remove-vertex leaf ?edge-list*)
using *contxt.obtain-leaf-tree-to-prufer-seq 3* **by** *blast*
then interpret *contxt'*: *tree-to-prufer-seq-context remove1 leaf verts remove-vertex leaf ?edge-list* **by** *simp*

let *?neigh* = *neighbor leaf ?edge-list*
have *degree*: *degree leaf ?edge-list* $\geq$ *1* **using** *find-Some find-leaf* **by** *fastforce*

have *?neigh* ∈ *set verts* **using** *neighbor-edge-in-edges*[*OF degree*] *contxt.t.wellformed-alt-fst*
by *blast*
then show *?case* **using** *find-leaf 3.IH contxt′* **unfolding** *prufer-sequences-def n-sequences-def*
apply *auto*
apply (*meson notin-set-remove1 subset-code*(*1*))
by (*metis Suc-diff-le Suc-length-remove1 contxt′.verts-length contxt.obtain-leaf-tree-to-prufer-seq find-leaf length-edge-list option.simps*(*1*))
qed

lemma *prufer-seq-in-verts*: $v \in$ *set* (*tree-to-prufer-seq verts edge-list*) $\Longrightarrow v \in$ *set verts*
using *pruf-seq-tree-to-prufer-seq* **unfolding** *prufer-sequences-def n-sequences-def*
by *auto*

lemma *degree-remove-vertex-non-adjacent*:
assumes $v \neq u$
and *non-adjacent*: $\{v,u\} \notin$ *edges-of-edge-list edge-list*
shows *degree u* (*remove-vertex v edge-list*) = *degree u edge-list*
proof −
have $(v,u) \notin$ *set edge-list* ∧ $(u,v) \notin$ *set edge-list* **using** *non-adjacent* **unfolding** *edges-of-edge-list-def* **by** *force*
then have *set* (*incident-edges u* (*remove-vertex v edge-list*)) = *set* (*incident-edges u edge-list*) **unfolding** *incident-edges-def edges-of-edge-list-def remove-vertex-def*
using *filter-filter* ‹$v \neq u$› **by** *auto*
then show *?thesis* **using** *distinct-edges distinct-remove-vertex distinct-card distinct-filter distinct-map incident-edges-def* **by** *metis*
qed

lemma *count-list-pruf-seq-degree*:
assumes *v-in-verts*: $v \in$ *set verts*
shows *Suc* (*count-list* (*tree-to-prufer-seq verts edge-list*) *v*) = *degree v edge-list*
using *v-in-verts tree-to-prufer-seq-context-axioms*
proof (*induction verts edge-list rule*: *tree-to-prufer-seq.induct*)
case (*1 verts*)
then interpret *contxt*: *tree-to-prufer-seq-context verts* [] **using** *tree-to-prufer-seq-context.intro*
by *blast*
show *?case* **using** *contxt.length-edge-list* **by** *auto*
next
case (*2 verts u w*)
then interpret *contxt*: *tree-to-prufer-seq-context verts* [(*u*,*w*)] **by** *simp*
interpret *tr*: *tree set verts* {{*u*,*w*}} **using** *contxt.tree* **unfolding** *edges-of-edge-list-def*
by *simp*
have *set verts* = {*u*,*w*} **using** *tr.V-Union-E contxt.non-trivial* **by** *blast*
then show *?case* **unfolding** *incident-edges-def* **using** *2* **by** *auto*
next
case (*3 verts e1 e2 edges*)
let *?edge-list* = *e1*#*e2*#*edges*
interpret *contxt*: *tree-to-prufer-seq-context verts ?edge-list* **using** *tree-to-prufer-seq-context.intro*

3 **by** *blast*
have *length ?edge-list* ≥ *2* **by** *simp*
then obtain *leaf*
where *find-leaf*: *find* (*λv. degree v ?edge-list* = *1*) *verts* = *Some leaf*
and *leaf*: *contxt.t.leaf leaf*
and *leaf-in-verts*: *leaf* ∈ *set verts*
and *contxt′*: *tree-to-prufer-seq-context* (*remove1 leaf verts*) (*remove-vertex leaf ?edge-list*)
using *contxt.obtain-leaf-tree-to-prufer-seq 3* **by** *blast*
then interpret *contxt′*: *tree-to-prufer-seq-context remove1 leaf verts remove-vertex leaf ?edge-list* **using** *tree-to-prufer-seq-context.intro* **by** *blast*
let *?neigh* = *neighbor leaf ?edge-list*
have *degree-leaf*: *degree leaf ?edge-list* = *1* **using** *find-leaf find-Some* **by** *fast*
show *?case*
proof (*cases v* = *leaf*)
case *True*
have *leaf* ∉ *set* (*remove1 leaf verts*) **using** *contxt.distinct-verts set-remove1-eq* **by** *auto*
then have *leaf-notin-pruf-seq′*: *leaf* ∉ *set* (*tree-to-prufer-seq* (*remove1 leaf verts*) (*remove-vertex leaf* (*e1* # *e2* # *edges*)))
using *contxt′.prufer-seq-in-verts True* **by** *blast*

have *neighbor leaf ?edge-list* ≠ *leaf*
using *degree-leaf* **by** (*simp add*: *contxt.t.edge-vertices-not-equal neighbor-edge-in-edges*)
then show *?thesis* **using** *find-leaf True leaf-notin-pruf-seq′ degree-leaf* **by** *auto*
next
case *False*
then have *v* ∈ *set* (*remove1 leaf verts*) **using** *3 set-remove1-eq* **by** *auto*
then have *IH*: *Suc* (*count-list* (*tree-to-prufer-seq* (*remove1 leaf verts*) (*remove-vertex leaf ?edge-list*)) *v*)
= *degree v* (*remove-vertex leaf ?edge-list*) **using** *3.IH find-leaf contxt′* **by** *blast*
then show *?thesis*
proof (*cases v* = *?neigh*)
case *True*
then show *?thesis* **using** *degree-neighbor-remove-vertex*[*OF degree-leaf*] *find-leaf IH* **by** *auto*
next
case *False*
have {*leaf*, *v*} ∉ *edges-of-edge-list ?edge-list*
proof
assume {*leaf*, *v*} ∈ *edges-of-edge-list ?edge-list*
then have *leaf-v-edge*: {*leaf*, *v*} ∈ *edges-of-edge-list* (*incident-edges leaf ?edge-list*)
unfolding *contxt.incident-edges-correct contxt.t.incident-edges-def contxt.t.incident-def* **by** *simp*
have {*?neigh*, *leaf*} ∈ *edges-of-edge-list ?edge-list* **using** *neighbor-edge-in-edges degree-leaf degree-length-filter* **by** *force*
then have {*?neigh*, *leaf*} ∈ *edges-of-edge-list* (*incident-edges leaf ?edge-list*)
unfolding *contxt.incident-edges-correct contxt.t.incident-edges-def con-*

txt.t.incident-def **by** *simp*
then show *False* **using** *leaf-v-edge degree-leaf*
by (*metis False One-nat-def card-le-Suc0-iff-eq contxt.degree-correct contxt.incident-edges-correct*
contxt.t.alt-degree-def contxt.t.fin-edges contxt.t.finite-incident-edges insert-iff le-numeral-extra(*4*) *singletonD*)
qed
then show *?thesis* **using** *False find-leaf IH find-leaf* ‹$v \neq$ *leaf*› *contxt.degree-remove-vertex-non-adjacent* **by** *auto*
qed
qed
qed

lemma *notin-set-tree-to-prufer-seq*:
assumes *v-in-verts*: $v \in$ *set verts*
shows $v \notin$ *set* (*tree-to-prufer-seq verts edge-list*) $\longleftrightarrow$ *degree v edge-list* = *1*
using *count-list-pruf-seq-degree assms count-list-zero-not-elem* **by** *force*

lemma *find-Some-impl-eq*: *find P xs* = *Some x* $\Longrightarrow \forall x.\ Q\ x \longrightarrow P\ x \Longrightarrow Q\ x \Longrightarrow$ *find Q xs* = *Some x*
by (*induction xs*) (*auto split*: *if-splits*)

lemma *pruf-seq-to-tree-to-pruf-seq*: *edges-of-edge-list* (*prufer-seq-to-tree-edges verts* (*tree-to-prufer-seq verts edge-list*)) = *edges-of-edge-list edge-list*
using *tree-to-prufer-seq-context-axioms*
proof (*induction verts edge-list rule*: *tree-to-prufer-seq.induct*)
case (*1 verts*)
then interpret *contxt*: *tree-to-prufer-seq-context verts* [] **using** *tree-to-prufer-seq-context.intro* **by** *blast*
show *?case* **using** *contxt.length-edge-list* **by** *auto*
next
case (*2 verts u w*)
then interpret *contxt*: *tree-to-prufer-seq-context verts* [(*u*, *w*)] **by** *simp*
interpret *tr*: *tree set verts* {{*u*,*w*}} **using** *contxt.tree* **unfolding** *edges-of-edge-list-def* **by** *simp*
have *card-verts*: *card* (*set verts*) = *2* **using** *tr.card-V-card-E* **by** *force*
then have *set-verts*: *set verts* = {*u*,*w*} **using** *tr.V-Union-E contxt.non-trivial* **by** *simp*
have *length verts* = *Suc* (*Suc 0*) **using** *contxt.distinct-verts card-verts distinct-card* **by** *fastforce*
then have $\exists\, a\ b.$ *verts* = [*a*,*b*] **by** (*metis length-0-conv length-Suc-conv*)
then show *?case* **unfolding** *edges-of-edge-list-def* **using** *set-verts* **by** *force*
next
case (*3 verts e1 e2 es*)
let *?edge-list* = *e1*#*e2*#*es*
interpret *contxt*: *tree-to-prufer-seq-context verts ?edge-list* **using** *3 tree-to-prufer-seq-context.intro* **by** *blast*
have *length ?edge-list* $\geq$ *2* **by** *simp*

then obtain *leaf*
where *find-leaf*: *find* ($\lambda v.$ *degree v ?edge-list* = *1*) *verts* = *Some leaf*
and *leaf*: *contxt.t.leaf leaf*
and *leaf-in-verts*: *leaf* ∈ *set verts*
and *contxt'*: *tree-to-prufer-seq-context* (*remove1 leaf verts*) (*remove-vertex leaf ?edge-list*)
using *contxt.obtain-leaf-tree-to-prufer-seq 3* **by** *blast*
then interpret *contxt'*: *tree-to-prufer-seq-context remove1 leaf verts remove-vertex leaf ?edge-list* **by** *simp*

have *degree-leaf*: *degree leaf ?edge-list* = *1* **using** *find-leaf find-Some* **by** *fast*
have *find-not-in-seq*: *find* ($\lambda v.$ *v* ∉ *set* (*tree-to-prufer-seq verts ?edge-list*)) *verts* = *Some leaf*
using *find-leaf contxt.notin-set-tree-to-prufer-seq find-cong* **by** *force*
show *?case* **using** *find-not-in-seq find-leaf 3.IH find-leaf contxt' insert-remove-leaf*[*OF degree-leaf*]
unfolding *edges-of-edge-list-def* **by** *simp*
qed

end

context *prufer-seq-to-tree-context*
begin

lemma *tree-labeled-tree-enum*:
assumes *t*: *tree* (*set verts*) *E*
shows (*set verts*, *E*) ∈ *set* (*labeled-tree-enum verts*)
proof−
interpret *t*: *tree set verts E* **using** *t* **.**
obtain *edges* **where** *set-edges*: *set edges* = *E* **and** *distinct-edges*: *distinct edges*
using *finite-distinct-list t.fin-edges* **by** *blast*
let *?edge-list* = *map* ($\lambda e.$ *SOME uv. mk-edge uv* = *e*) *edges*
have ∀ *e*∈*E*. ∃ *uv. mk-edge uv* = *e* **using** *t.two-edges card-2-iff* **by** (*metis mk-edge.simps*)
then have $\bigwedge e.$ *e* ∈ *E* ⟹ (*mk-edge o* ($\lambda e.$ *SOME uv. mk-edge uv* = *e*)) *e* = *e*
using *someI-ex*
by (*smt* (*verit*, *del-insts*) *comp-apply*)
then have *map-edges*: *map mk-edge ?edge-list* = *edges* **unfolding** *map-map*
using *map-idI set-edges* **by** *blast*
then have *edge-list*: *edges-of-edge-list ?edge-list* = *E* **unfolding** *edges-of-edge-list-def*
using *set-edges set-map* **by** *metis*
have *distinct-edge-list*: *distinct* (*map mk-edge ?edge-list*) **using** *distinct-edges map-edges* **by** *metis*

then interpret *contxt*: *tree-to-prufer-seq-context verts ?edge-list* **using** *t tree-to-prufer-seq-context.intro distinct-verts edge-list card-verts* **by** *blast*
show *?thesis*
using *contxt.pruf-seq-tree-to-prufer-seq contxt.pruf-seq-to-tree-to-pruf-seq n-sequence-enum-correct distinct-edge-list edge-list*
unfolding *prufer-sequences-def prufer-seq-to-tree-def labeled-tree-enum-def* **by**

auto
qed

lemma *V-labeled-tree-enum-verts*: $(V,E) \in$ *set* (*labeled-tree-enum verts*) $\Longrightarrow V =$ *set verts*
unfolding *labeled-tree-enum-def* **by** (*metis Pair-inject ex-map-conv prufer-seq-to-tree-def*)

theorem *labeled-tree-enum-correct*: *set* (*labeled-tree-enum verts*) = *labeled-trees* (*set verts*)
using *labeled-tree-enum-tree V-labeled-tree-enum-verts tree-labeled-tree-enum* **unfolding** *labeled-trees-def* **by** *auto*

2.4 Distinctness

lemma *count-list-degree*: *seq* $\in$ *prufer-sequences verts* $\Longrightarrow v \in$ *set verts* $\Longrightarrow$ *Suc* (*count-list seq v*) = *degree v* (*prufer-seq-to-tree-edges verts seq*)
using *verts-length distinct-verts*
proof (*induction verts seq rule*: *prufer-seq-to-tree-edges.induct*)
case (*1 u w*)
then show *?case* **unfolding** *incident-edges-def* **by** *auto*
next
case (*2 verts a seq*)
then interpret *contxt*: *prufer-seq-to-tree-context verts* **by** *unfold-locales*
obtain *leaf*
where *leaf-find*: *find* ($\lambda x.\ x \notin$ *set* (*a # seq*)) *verts* = *Some leaf*
and *leaf-not-in-seq*: *leaf* $\notin$ *set* (*a#seq*)
and *seq-in-verts'*: *seq* $\in$ *prufer-sequences* (*remove1 leaf verts*)
and *len-verts'*: $2 \leq$ *length* (*remove1 leaf verts*)
and *distinct-verts'*: *distinct* (*remove1 leaf verts*)
and *leaf-in-verts*: *leaf* $\in$ *set verts* **using** *contxt.obtain-b-prufer-seq-to-tree-edges 2* **by** *blast*
interpret *contxt'*: *prufer-seq-to-tree-context remove1 leaf verts* **using** *len-verts' distinct-verts'* **by** *unfold-locales*
show *?case*
proof (*cases v = leaf*)
case *True*
then have $a \neq$ *leaf* **using** *2 leaf-not-in-seq* **by** *auto*
interpret *t*: *tree set* (*remove1 leaf verts*) *edges-of-edge-list* (*prufer-seq-to-tree-edges* (*remove1 leaf verts*) *seq*)
using *contxt'.prufer-seq-to-tree-edges-tree seq-in-verts'* **by** *auto*
have [*simp*]: *set* (*remove1 leaf verts*) = *set verts* − {*leaf*} **using** *set-remove1-eq 2* **by** *auto*
then have $\forall (u,w) \in$ *set* (*prufer-seq-to-tree-edges* (*remove1 leaf verts*) *seq*). $u \neq$ *leaf* $\wedge\ w \neq$ *leaf*
using *t.wellformed in-mk-edge-img* **unfolding** *edges-of-edge-list-def* **apply** *auto* **by** *fast+*
then have *degree v* (*prufer-seq-to-tree-edges* (*remove1 leaf verts*) *seq*) = *0*
unfolding *incident-edges-def filter-False True* **by** (*auto split*: *prod.splits*)
then show *?thesis* **using** ‹$a \neq$*leaf*› *True leaf-find leaf-not-in-seq* **unfolding**

incident-edges-def **by** *simp*
  next
    case *False*
    then show *?thesis* **using** *2 leaf-find seq-in-verts′ len-verts′* **unfolding** *incident-edges-def* **by** *auto*
  qed
qed (*auto simp*: *prufer-sequences-def n-sequences-def*)

lemma *vert-notin-pruf-seq-leaf*: *seq* ∈ *prufer-sequences verts* ⟹ *v* ∈ *set verts* ⟹ *v* ∉ *set seq* ⟷ *degree v* (*prufer-seq-to-tree-edges verts seq*) = *1*
  using *count-list-degree count-list-zero-not-elem* **by** *fastforce*

lemma *inj-prufer-seq-to-tree-edges*:
  assumes *pruf-seq1*: *seq1* ∈ *prufer-sequences verts*
    and *pruf-seq2*: *seq2* ∈ *prufer-sequences verts*
    and *seq-ne*: *seq1* ≠ *seq2*
  shows *edges-of-edge-list* (*prufer-seq-to-tree-edges verts seq1*) ≠ *edges-of-edge-list* (*prufer-seq-to-tree-edges verts seq2*) (**is** *?l* ≠ *?r*)
proof
  assume *trees-eq*: *?l* = *?r*
  have *length seq1* = *length seq2* **using** *pruf-seq1 pruf-seq2* **unfolding** *prufer-sequences-def n-sequences-def* **by** *simp*
  then show *False*
    using *assms* ‹*?l* = *?r*› *prufer-seq-to-tree-context-axioms*
  proof (*induction seq1 seq2 arbitrary*: *verts rule*: *list-induct2*)
    case *Nil*
    then show *?case* **by** *simp*
  next
    case (*Cons x xs y ys*)
    then interpret *prufer-seq-to-tree-context verts* **by** *simp*
      interpret *t1*: *tree set verts edges-of-edge-list* (*prufer-seq-to-tree-edges verts* (*x#xs*)) **using** *Cons*(*3*) *prufer-seq-to-tree-edges-tree* **by** *fastforce*
      interpret *t2*: *tree set verts edges-of-edge-list* (*prufer-seq-to-tree-edges verts* (*y#ys*)) **using** *Cons*(*4*) *prufer-seq-to-tree-edges-tree* **by** *fastforce*
    obtain *leaf* **where** *find-leaf*: *find* (λ*v*. *v* ∉ *set* (*x#xs*)) *verts* = *Some leaf*
      and *pruf-seq1′*: *xs* ∈ *prufer-sequences* (*remove1 leaf verts*)
      and *length* (*remove1 leaf verts*) ≥ *2*
        and *distinct* (*remove1 leaf verts*) **using** *obtain-b-prufer-seq-to-tree-edges Cons*(*3*) **by** *blast*
   then interpret *contxt′*: *prufer-seq-to-tree-context remove1 leaf verts* **by** (*unfold-locales*, *simp*)
    obtain *leaf2* **where** *find-leaf2*: *find* (λ*v*. *v* ∉ *set* (*y#ys*)) *verts* = *Some leaf2*
     and *pruf-seq2′*: *ys* ∈ *prufer-sequences* (*remove1 leaf2 verts*) **using** *obtain-b-prufer-seq-to-tree-edges Cons*(*4*) **by** *blast*
     interpret *ttps-contxt1*: *tree-to-prufer-seq-context verts prufer-seq-to-tree-edges verts* (*x#xs*)
       using *distinct-verts verts-length distinct-prufer-seq-to-tree*[*OF Cons*(*3*)] **by** (*unfold-locales*, *auto simp*: *distinct-card*)
     interpret *ttps-contxt2*: *tree-to-prufer-seq-context verts prufer-seq-to-tree-edges*

verts (*y#ys*)
using *distinct-verts verts-length distinct-prufer-seq-to-tree*[*OF Cons*(*4*)] **by** (*unfold-locales*, *auto simp*: *distinct-card*)
have *1*: *find* (*λv. v* ∉ *set* (*x#xs*)) *verts* = *find* (*λv. t1.leaf v*) *verts* **using** *vert-notin-pruf-seq-leaf*[*OF Cons*(*3*)] *ttps-contxt1.degree-correct find-cong* **unfolding** *t1.leaf-def* **by** *force*
have *2*: *find* (*λv. v* ∉ *set* (*y#ys*)) *verts* = *find* (*λv. t2.leaf v*) *verts* **using** *vert-notin-pruf-seq-leaf*[*OF Cons*(*4*)] *ttps-contxt2.degree-correct find-cong* **unfolding** *t2.leaf-def* **by** *force*
have *find* (*λv. v* ∉ *set* (*x#xs*)) *verts* = *find* (*λv. v* ∉ *set* (*y#ys*)) *verts* **using** *Cons*(*6*) *1 2* **unfolding** *t1.leaf-def t2.leaf-def* **by** *simp*
have *leafs-eq*: *leaf2* = *leaf* **using** *Cons*(*6*) *1 2 find-leaf find-leaf2* **unfolding** *t1.leaf-def t2.leaf-def* **by** *simp*
have *leaf-not-in-verts′*: *leaf* ∉ *set* (*remove1 leaf verts*) **using** *distinct-verts set-remove1-eq* **by** *simp*
show *False*
proof (*cases y* = *x*)
case *True*
then have *xs* ≠ *ys* **using** *Cons* **by** *simp*
have *1*: {*x*, *leaf*} ∉ *edges-of-edge-list* (*prufer-seq-to-tree-edges* (*remove1 leaf verts*) *xs*) **using** *contxt′.prufer-seq-to-tree-edges-wf pruf-seq1′ leaf-not-in-verts′* **by** *blast*
have *2*: {*x*, *leaf*} ∉ *edges-of-edge-list* (*prufer-seq-to-tree-edges* (*remove1 leaf verts*) *ys*) **using** *contxt′.prufer-seq-to-tree-edges-wf pruf-seq2′ leaf-not-in-verts′ True leafs-eq* **by** *blast*
then have *edges-of-edge-list* (*prufer-seq-to-tree-edges* (*remove1 leaf verts*) *xs*) = *edges-of-edge-list* (*prufer-seq-to-tree-edges* (*remove1 leaf verts*) *ys*)
using *Cons*(*6*) *find-leaf find-leaf2 leafs-eq True insert-ident*[*OF 1 2*] **unfolding** *edges-of-edge-list-def* **by** *simp*
then show *?thesis* **using** *True leafs-eq Cons.IH pruf-seq1′ pruf-seq2′ leafs-eq Cons*(*6*) *find-leaf*
find-leaf2 ‹*xs* ≠ *ys*› *contxt′.prufer-seq-to-tree-context-axioms* **unfolding** *edges-of-edge-list-def* **by** *auto*
next
case *False*
then have {*x*, *leaf*} ∉ *edges-of-edge-list* (*prufer-seq-to-tree-edges* (*remove1 leaf verts*) *ys*) **using** *find-leaf2 leafs-eq contxt′.prufer-seq-to-tree-edges-wf pruf-seq2′ leaf-not-in-verts′* **by** *auto*
then show *?thesis* **using** *Cons*(*6*) *find-leaf find-leaf2 leafs-eq False* **unfolding** *edges-of-edge-list-def*
by (*auto*, *metis* (*no-types*, *lifting*) *doubleton-eq-iff insert-iff*)
qed
qed
qed

lemma *inj-on-prufer-seq-to-tree*: *inj-on* (*prufer-seq-to-tree verts*) (*prufer-sequences verts*)
unfolding *inj-on-def prufer-seq-to-tree-def* **using** *inj-prufer-seq-to-tree-edges* **by** *auto*

theorem *labeled-tree-enum-distinct*: *distinct* (*labeled-tree-enum verts*)
unfolding *labeled-tree-enum-def* **using** *inj-on-prufer-seq-to-tree*
by (*simp add*: *distinct-map n-sequence-enum-correct n-sequence-enum-distinct prufer-sequences-def distinct-verts*)

end

end