Kruskal’s Algorithm for Minimum Spanning Forest

Maximilian P.L. Haslbeck, Peter Lammich, Julian Biendarra

June 11, 2019

Abstract

This Isabelle/HOL formalization defines a greedy algorithm for finding a minimum weight basis on a weighted matroid and proves its correctness. This algorithm is an abstract version of Kruskal’s algorithm.

We interpret the abstract algorithm for the cycle matroid (i.e. forests in a graph) and refine it to imperative executable code using an efficient union-find data structure.

Our formalization can be instantiated for different graph representations. We provide instantiations for undirected graphs and symmetric directed graphs.

Contents

1 Minimum Weight Basis 2
  1.1 Preparations .............................. 3
     1.1.1 Weight restricted set .................. 4
     1.1.2 The greedy idea ...................... 4
  1.2 Minimum Weight Basis algorithm ........ 6
  1.3 The heart of the argument ................ 6
  1.4 The Invariant ............................. 8
  1.5 Invariant proofs ........................... 9
  1.6 The refinement lemma .................... 10

2 Kruskal interface 10
  2.1 Derived facts ............................. 11
  2.2 The edge set and forest form the cycle matroid .. 13

3 Refine Kruskal 15
  3.1 Refinement I: cycle check by connectedness ........ 15
  3.2 Refinement II: connectedness by PER operation .... 16
1 Minimum Weight Basis

theory MinWeightBasis
begin

For a matroid together with a weight function, assigning each element of the carrier set an weight, we construct a greedy algorithm that determines a minimum weight basis.

locale weighted-matroid = matroid carrier indep for carrier::'a set and indep +
fixes weight :: 'a ⇒ 'b::{linorder, ordered-comm-monoid-add}
begin
definition minBasis where
  minBasis B ≡ basis B ∧ (∀ B'. basis B' → sum weight B ≤ sum weight B')
1.1 Preparations

fun in-sort-edge where
  in-sort-edge x [] = [x]
  | in-sort-edge x (y#ys) = (if weight x ≤ weight y then x#y#ys else y# in-sort-edge x ys)

lemma [simp]: set (in-sort-edge x L) = insert x (set L) by (induct L, auto)

lemma in-sort-edge: sorted-wrt (λe1 e2. weight e1 ≤ weight e2) L
  ⇒ sorted-wrt (λe1 e2. weight e1 ≤ weight e2) (in-sort-edge x L)
  by (induct L, auto)

lemma in-sort-edge-distinct: x /∈ set L ⇒ distinct L ⇒ distinct (in-sort-edge x L)
  by (induct L, auto)

lemma finite-sorted-edge-distinct:
  assumes finite S
  obtains L where distinct L sorted-wrt (λe1 e2. weight e1 ≤ weight e2) L S = set L
  proof –
  { have ∃L. distinct L ∧ sorted-wrt (λe1 e2. weight e1 ≤ weight e2) L ∧ S = set L
    using assms
    apply(induct S)
    apply(clarsimp)
    apply(clarsimp)
    subgoal for x L apply(rule exI[where x=in-sort-edge x L])
    by (auto simp: in-sort-edge in-sort-edge-distinct)
    done
  }
  with that show thesis by blast
qed

abbreviation usorted == sorted-wrt (λe1 e2. weight e1 ≤ weight e2)

lemma sum-list-map-cons:
  sum-list (map weight (y # ys)) = weight y + sum-list (map weight ys)
  by auto

lemma exists-greater:
  assumes len: length F = length F'
  and sum: sum-list (map weight F) > sum-list (map weight F')
  shows ∃i<length F. weight (F ! i) > weight (F' ! i)
  using len sum
  proof (induct rule: list-induct2)
  case (Cons x xs y ys)
  from Cons(3)
  qed
have *: \sim weight y < weight x \implies \text{sum-list (map weight ys)} < \text{sum-list (map weight xs)}
  by (metis add-mono not-less sum-list-map-cons)
show \(?case
  using Cons *
  by (cases weight y < weight x, auto)
qed simp

lemma wsorted-nth-mono; assumes wsorted L i \leq j < \text{length } L
shows weight (L[i]) \leq weight (L[j])
using assms by (induct L arbitrary: i j rule: list.induct, auto simp: nth-Cons*)

1.1.1 Weight restricted set

\(\text{limi } T g\) is the set \(T\) restricted to elements only with weight strictly smaller than \(g\).
definition \(\text{limi } T g \equiv \{ e . e \in T \land weight e < g \}\)

lemma limi-subset: \(\text{limi } T g \subseteq T\) by (auto simp: limi-def)

lemma limi-mono: \(A \subseteq B \implies \text{limi } A g \subseteq \text{limi } B g\) by (auto simp: limi-def)

1.1.2 The greedy idea
definition \(\text{no-smallest-element-skipped } E F\)
  \(= (\forall e \in \text{carrier} - E. \forall g > weight e. \text{indep (insert } e \text{ (limi } F g)))) \implies (e \in \text{limi } F g))\)

let \(F\) be a set of elements \(\text{limi } F g\) is \(F\) restricted to elements with weight smaller than \(g\) let \(E\) be a set of elements we want to exclude.

\(\text{no-smallest-element-skipped } E F\) expresses, that going greedily over \(\text{carrier} - E\), every element that did not render the accumulated set dependent, was added to the set \(F\).

lemma no-smallest-element-skipped-empty[simp]: no-smallest-element-skipped carrier {}
by (auto simp: no-smallest-element-skipped-def)

lemma no-smallest-element-skippedD:
assumes no-smallest-element-skipped E F e \in carrier - E
  weight e < g (indep (insert e (limi F g)))
shows e \in limi F g
using assms by (auto simp: no-smallest-element-skipped-def)

lemma no-smallest-element-skipped-skip:
assumes createsCycle: \neg indep (insert e F)
and I: no-smallest-element-skipped (E \cup \{e\}) F
and sorted: (\forall x \in F. \forall y \in (E \cup \{e\}). weight x \leq weight y)
shows no-smallest-element-skipped E F

unfolding no-smallest-element-skipped-def

proof (clarsimp)

fix x g

assume x: x ∈ carrier x /∈ E weight x < g

assume f: indep (insert x (limi F g))

show (x ∈ limi F g)

proof (cases x=e)

case True

from True have limi F g = F

unfolding limi-def using ⟨weight x < g⟩ sorted by fastforce

with createsCycle f True have False by auto

then show ?thesis by simp

next

case False

show ?thesis

apply (rule I [THEN no-smallest-element-skippedD, OF - ⟨weight x < g⟩])

using x f False

by auto

qed

qed

lemma no-smallest-element-skipped-add:

assumes I: no-smallest-element-skipped (E∪{e}) F

shows no-smallest-element-skipped E (insert e F)

unfolding no-smallest-element-skipped-def

proof (clarsimp)

fix x g

assume xc: x ∈ carrier

assume x: x /∈ E

assume wx: weight x < g

assume f: indep (insert x (limi (insert e F) g))

show (x ∈ limi (insert e F) g)

proof (cases x=e)

case True

then show ?thesis unfolding limi-def

using wx by blast

next

case False

have ind: indep (insert x (limi F g))

apply (rule indep-subset[OF f]) using limi-mono by blast

have indep (insert x (limi F g)) ⇒ x ∈ limi F g

apply (rule I [THEN no-smallest-element-skippedD]) using False xc wx x by auto

with ind show ?thesis using limi-mono by blast

qed

qed
1.2 Minimum Weight Basis algorithm

definition obtain-sorted-carrier ≡ SPEC (λl. wsorted l ∧ set l = carrier)

abbreviation empty-basis ≡ {}

To compute a minimum weight basis one obtains a list of the carrier set sorted ascendingly by the weight function. Then one iterates over the list and adds an elements greedily to the independent set if it does not render the set dependent.

definition minWeightBasis where
minWeightBasis ≡ do
  l ← obtain-sorted-carrier;
  ASSERT (set l = carrier);
  T ← nfoldli l (λ-. True)
  (λe T. do
    ASSERT (indep T ∧ e∈carrier ∧ T⊆carrier);
    if indep (insert e T) then
      RETURN (insert e T)
    else
      RETURN T
  ) empty-basis;
RETURN T

1.3 The heart of the argument

The algorithmic idea above is correct, as an independent set, which is inclusion maximal and has not skipped any smaller element, is a minimum weight basis.

lemma greedy-approach-leads-to-minBasis: assumes indep: indep F and inclmax: ∀e∈carrier − F. ¬indep (insert e F) and no-smallest-element-skipped {} F shows minBasis F

proof (rule contr)
— from our assumptions we have that F is a basis
from indep inclmax have bF: basis F using indep-not-basis by blast
— towards a contradiction, assume F is not a minimum Basis
assume notmin: ¬ minBasis F
— then we can get a smaller Basis B
from bF notmin[unfolded minBasis-def] obtain B
where bB: basis B and sum: sum weight B < sum weight F
by force
— lets us obtain two sorted lists for the bases F and B
from bF basis-finite finite-sorted-edge-distinct
obtain FL where dF[simp]: distinct FL and wF[simp]: wsorted FL
  and sF[simp]: F = set FL
by blast
from bB basis-finite finite-sorted-edge-distinct
**obtain BL where** dB[simp]; distinct BL and wB[simp]; wsorted BL

**and** sB[simp]; B = set BL

**by** blast

— as basis F has more total weight than basis B (and the basis have the same length) ...

**from** sum have saml: sum-list (map weight BL) < sum-list (map weight FL)

**by**(simp add: sum.distinct-set-conv-list[symmetric])

**from** bB bF have card B = card F using basis-card by blast

**then have** l: length FL = length BL by (simp add: distinct-card)

— ... there exists an index i such that the ith element of the BL is strictly smaller than the ith element of FL

**from** exists-greater[OF l saml] obtain i where i: i<length FL

**and gr: weight (BL ! i) < weight (FL ! i)**

**by** auto

**let** ?FL-restricted = limi (set FL) (weight (FL ! i))

— now let us look at the two independent sets X and Y: let X and Y be the set if we take the first i-1 elements of BL and the first i elements of FL respectively. We want to use the augment property of Matroids in order to show that we must have skipped and optimal element, which then contradicts our assumption.

**let** ?X = take i FL

**have** X-size: card (set ?X) = i using i

**by** (simp add: distinct-card)

**have** X-indep: indep (set ?X) using bF

**using** indep-iff-subset-basis set-take-subset by force

**let** ?Y = take (Suc i) BL

**have** Y-size: card (set ?Y) = Suc i using i l

**by** (simp add: distinct-card)

**have** Y-indep: indep (set ?Y) using bB

**using** indep-iff-subset-basis set-take-subset by force

**have** card (set ?X) < card (set ?Y) using X-size Y-size by simp

— X and Y are independent and X is smaller than Y, thus we can augment X with some element x

**with** Y-indep X-indep

**obtain** x where x: x∈set (take (Suc i) BL) − set ?X

**and indexp: indep (insert x (set ?X))**

**using** augment by auto

— we know many things about x now, i.e. x weights strictly less than the ith element of FL ...

**have** x∈carrier using indexp indep-subset-carrier by blast

**from** x have xs: x∈set (take (Suc i) BL) and xnX: x /∈ set ?X by auto

**from** xs obtain j where x=(take (Suc i) BL)!j and ij: j≤i

**by** (metis i in-set-conv-nth l length-take less-Suc-eq-le min-Suc-gt(2))

**then have** x: x=BL!j by auto

**have** il: i < length BL using i l by simp
have weight \( x \leq \text{weight} (BL ! i) \)

unfolding \( x \) apply (rule wsorted-nth-mono) by fact+

then have \( k: \text{weight} \ x < \text{weight} (FL ! i) \) using \( \text{gr} \) by auto

— ... and that adding \( x \) to \( X \) gives us an independent set

have \( ?FL\text{-restricted} \subseteq \text{set} \ ?X \)

unfolding \( \text{limi-def} \) apply safe

by (metis (no-types, lifting) \( i \) in-set-cone-nth \ length-take
min-simps (2) not-less nth-take \( wF \) wsorted-nth-mono)

have \( z': \text{insert} \ x \ ?FL\text{-restricted} \subseteq \text{insert} \ x \ (\text{set} \ ?X) \)

using \( \text{xnX} \ (\?FL\text{-restricted} \subseteq \text{set} \ (\text{take} \ i \ FL)) \) by auto
from indep-subset[OE indepX \( z' \)] have add-x-stay-indep: indep (\text{insert} \ x \ ?FL\text{-restricted})

— ... finally this means that we must have taken the element during our greedy algorithm
from \( \langle \\text{no-smallest-element-skipped} \ \{\} \ F\rangle \)
\( \langle x \in \text{carrier} \rangle \ \text{weight} \ x < \text{weight} (FL ! i) \) add-x-stay-indep

have \( x \in ?FL\text{-restricted} \) by (auto \( \text{dest} \): \( \text{no-smallest-element-skippedD} \))
with \( \langle ?FL\text{-restricted} \subseteq \text{set} \ ?X \rangle \) have \( x \in \text{set} \ ?X \) by auto

— ... but we actually didn’t. This finishes our proof by contradiction.

with \( \text{xnX} \) show \( \text{False} \) by auto

qed

1.4 The Invariant

The following predicate is invariant during the execution of the minimum weight basis algorithm, and implies that its result is a minimum weight basis.

definition I-minWeightBasis where
I-minWeightBasis == \( \lambda (T, E). \text{indep} \ T \)
\( \land \ T \subseteq \text{carrier} \)
\( \land \ E \subseteq \text{carrier} \)
\( \land (\forall x \in T. \forall y \in E. \ \text{weight} \ x \leq \text{weight} \ y) \)
\( \land (\forall e \in \text{carrier} - E - T. \sim \text{indep} (\text{insert} \ e \ T)) \)
\( \land \text{no-smallest-element-skipped} \ E \ T \)

lemma I-minWeightBasisD:
assumes I-minWeightBasis \( (T, E) \)
shows indep \( T \land e. e \in \text{carrier} - E - T \implies \sim \text{indep} (\text{insert} \ e \ T) \)
\( E \subseteq \text{carrier} \land x \ y. x \in T \implies y \in E \implies \text{weight} \ x \leq \text{weight} \ y \ T \subseteq \text{carrier} \)
\( \text{no-smallest-element-skipped} \ E \ T \)
using assms by (auto simp: no-smallest-element-skipped-def I-minWeightBasis-def)

lemma I-minWeightBasisI:
assumes indep \( T \land e. e \in \text{carrier} - E - T \implies \sim \text{indep} (\text{insert} \ e \ T) \)
\( E \subseteq \text{carrier} \land x \ y. x \in T \implies y \in E \implies \text{weight} \ x \leq \text{weight} \ y \ T \subseteq \text{carrier} \)
\( \text{no-smallest-element-skipped} \ E \ T \)
shows \( \text{I-minWeightBasis} (T, E) \)
using assms by (auto simp: no-smallest-element-skipped-def I-minWeightBasis-def)

**Lemma I-minWeightBasisG**: \( \text{I-minWeightBasis} (T, E) \Rightarrow \) no-smallest-element-skipped \( E \ T \)
by (auto simp: I-minWeightBasis-def)

**Lemma I-minWeightBasis-sorted**: \( \text{I-minWeightBasis} (T, E) \Rightarrow (\forall x \in T. \forall y \in E. \text{weight } x \leq \text{weight } y) \)
by (auto simp: I-minWeightBasis-def)

### 1.5 Invariant proofs

**Lemma I-minWeightBasis-empty**: \( \text{I-minWeightBasis} (\{\}, \text{carrier}) \)
by (auto simp: I-minWeightBasis-def)

**Lemma I-minWeightBasis-final**: \( \text{I-minWeightBasis} (T, \{\}) \Rightarrow \text{minBasis } T \)
by (auto simp: greedy-approach-leads-to-minBasis I-minWeightBasis-def)

**Lemma indep-aux**:
assumes \( e \in E \forall e \in \text{carrier} - E - F. \neg \text{indep} (\text{insert } e F) \)
and \( x \in \text{carrier} - (E - \{e\}) - \text{insert } e F \)
shows \( \neg \text{indep} (\text{insert } x (\text{insert } e F)) \)
using assms indep-iff-subset-basis by auto

**Lemma preservation-if**: \( \text{wsorted } x \Rightarrow \text{set } x = \text{carrier} \Rightarrow \)
x \( = l1 @ xa \# l2 \Rightarrow \text{I-minWeightBasis} (\sigma, \text{set } (xa \# l2)) \Rightarrow \text{indep } \sigma \)
\( \Rightarrow xa \in \text{carrier} \Rightarrow \text{indep} (\text{insert } xa \sigma) \Rightarrow \text{I-minWeightBasis} (\text{insert } xa \sigma, \text{set } l2) \)
apply (rule I-minWeightBasisI)
subgoal by simp
subgoal unfolding I-minWeightBasis-def by (rule indep-aux[where \( E=\text{set } (xa \# l2)\)])
by simp-all
subgoal by auto
subgoal by (metis insert-iff list.set(2) I-minWeightBasis-sorted sorted-wrt-append sorted-wrt.simps(2))
subgoal by (auto simp: I-minWeightBasis-def)
subgoal apply (rule no-smallest-element-skipped-add)
by (auto intro!: simp: I-minWeightBasis-def)
done

**Lemma preservation-else**: \( \text{set } x = \text{carrier} \Rightarrow \)
x \( = l1 @ xa \# l2 \Rightarrow \text{I-minWeightBasis} (\sigma, \text{set } (xa \# l2)) \)
\( \Rightarrow \text{indep } \sigma \Rightarrow \neg \text{indep } (\text{insert } xa \sigma) \Rightarrow \text{I-minWeightBasis} (\sigma, \text{set } l2) \)
apply (rule I-minWeightBasisI)
subgoal by simp
subgoal by (auto simp: DiffD2 I-minWeightBasis-def)
subgoal by auto

9
subgoal by (auto simp: I-minWeightBasis-def)
subgoal by (auto simp: I-minWeightBasis-def)
subgoal apply (rule no-smallest-element-skipped-skip)
  by (auto intro!: simp: I-minWeightBasis-def)
done

1.6 The refinement lemma

theorem minWeightBasis-refine: (minWeightBasis, SPEC minBasis) ∈ ⟨Id⟩res-rel
unfolding minWeightBasis-def obtain-sorted-carrier-def
apply (refine-vcg nfoldli-rule[where I = λl1 l2 s. I-minWeightBasis (s, set l2)])
subgoal by auto
subgoal by (auto simp: I-minWeightBasis-empty)
  — asserts
subgoal by (auto simp: I-minWeightBasis-def)
subgoal by (auto simp: I-minWeightBasis-def)
subgoal by (auto simp: I-minWeightBasis-def)
  — branches
subgoal apply (rule preservation-if) by auto
subgoal apply (rule preservation-else) by auto
  — final
subgoal by auto
subgoal by (auto simp: I-minWeightBasis-final)
done

end — locale minWeightBasis
end

2 Kruskal interface

theory Kruskal
imports Kruskal-Misc MinWeightBasis
begin

In order to instantiate Kruskal’s algorithm for different graph formalizations we provide an interface consisting of the relevant concepts needed for the algorithm, but hiding the concrete structure of the graph formalization. We thus enable using both undirected graphs and symmetric directed graphs.

Based on the interface, we show that the set of edges together with the predicate of being cycle free (i.e. a forest) forms the cycle matroid. Together with a weight function on the edges we obtain a weighted-matroid and thus an instance of the minimum weight basis algorithm, which is an abstract version of Kruskal.

locale Kruskal-interface =
fixes E :: 'edge set
  and V :: 'a set
and vertices :: 'edge ⇒ 'a set
and joins :: 'a ⇒ 'a ⇒ 'edge ⇒ bool
and forest :: 'edge set ⇒ bool
and connected :: 'edge set ⇒ ('a*'a) set
and weight :: 'edge ⇒ 'b::{linorder, ordered-comm-monoid-add}

assumes
finiteE[simp]: finite E
and forest-subE: forest E' ⟹ E' ⊆ E
and forest-empty: forest {}
and forest-mono: forest X ⟹ Y ⊆ X ⟹ forest Y
and connected-same: (u,v) ∈ connected {} ⟹ u=v ∧ v∈V
and findaugmenting-aux: E1 ⊆ E ⟹ E2 ⊆ E ⟹ (u,v) ∈ connected E1 ⟹
(u,v)≠ connected E2
⟹ ∃ a b e. (a,b) ≠ connected E2 ∧ e ≠ E2 ∧ e ∈ E1 ∧ joins a b e
and augment-forest: forest F ⟹ e ∈ E−F ⟹ joins u v e
⟹ forest (insert e F) ⟹ (u,v) ≠ connected F
and equiv: F ⊆ E ⟹ equiv V (connected F)
and connected-in: F ⊆ E ⟹ connected F ⊆ V × V
and insert-reachable: x ∈ V ⟹ y ∈ V ⟹ F ⊆ E ⟹ e∈E ⟹ joins x y e
⟹ connected (insert e F) = per-union (connected F) x y
and exhaust: ∃ x. x∈E ⟹ ∃ a b. joins a b x
and vertices-constr: ∃ a b e. joins a b e ⟹ \{a,b\} ⊆ vertices e
and joins-sym: ∃ a b e. joins a b e ⟹ joins b a e
and selfloop-no-forest: ∃ e. e∈E ⟹ joins a a e ⟹ ¬ forest (insert e F)
and finite-vertices: ∃ a b e. e∈E ⟹ finite (vertices e)

and edgesinvertices: ∪{ vertices ' E) ⊆ V
and finiteV[simp]: finite V
and joins-connected: joins a b e ⟹ T⊆E ⟹ e∈T ⟹ (a,b) ∈ connected T

begin

2.1 Derived facts

lemma joins-in-V: joins a b e ⟹ e∈E ⟹ a∈V ∧ b∈V
apply(frule vertices-constr) using edgesinvertices by blast

lemma finiteE-finiteV: finite E ⟹ finite V
using finite-vertices by auto

lemma E-inV: ∃ e. e∈E ⟹ vertices e ⊆ V
using edgesinvertices by auto

definition CC E' x = (connected E')''[x]

lemma sameCC-reachable: E' ⊆ E ⟹ u∈V ⟹ v∈V ⟹ CC E' u = CC E' v
⇐ (u,v) ∈ connected E'
unfolding CC-def using equiv-class-eq-iff[OF equiv] by auto

11
definition \( CCs E' = \) quotient \( V \) (connected \( E' \))

lemma quotient \( V \) \( Id = \{ \{ v \} | v \in V \} \)

unfolding quotient-def by auto

lemma \( CCs \)-empty: \( CCs \{ \} = \{ \{ v \} | v \in V \} \)

unfolding CCs-def unfolding quotient-def using connected-same by auto

lemma \( CCs \)-empty-card: \( card \ (CCs \{ \}) = card V \)

proof
have \( i: \{ \{ v \} | v \in V \} = (\lambda v. \{ v \})'V \)
  by blast
have \( card \ (CCs \{ \}) = card \ \{ \{ v \} | v \in V \} \)
  using CCs-empty by auto
also have \( \ldots = card ((\lambda v. \{ v \})'V) \)
  by (simp only: \( i \))
also have \( \ldots = card V \)
apply (rule card-image)
unfolding inj-on-def by auto
finally show \( \text{thesis} \).
qed

lemma \( CCs \)-image-CC: \( CCs F = (CC F)'V \)

unfolding CCs-def CC-def quotient-def
by blast

lemma union-eqclass-decreases-components:
assumes \( CC F x \neq CC F y e \notin F x \in V y \in V F \subseteq E e \in E \) joins \( x \ y \ e \)
shows \( Suc (card (CCs (insert e F))) = card (CCs F) \)

proof
from assms(1) have \( xny \): \( x \neq y \) by blast
show \( \text{thesis} \) unfolding CCs-def
apply (simp only: insert-reachable[OF assms(3–7)])
apply (rule unify2EquivClasses-alt)
apply (fact assms(1)[unfolded CC-def])
apply fact+
apply (rule connected-in)
apply fact
apply (rule equiv)
apply fact
by (fact finiteV)
qed

lemma forest-CCs: assumes forest \( E' \) shows \( card \ (CCs E') + card E' = card V \)

proof
from assms have \( finite E' \) using forest-subE
  using finiteE finite-subset by blast
from this assms show \( \text{thesis} \)
proof (induct \( E' \))
  case (insert \( x \) \( F \))
then have $x \in E$ using forest-subE by auto
from this obtain $a \neq b$ where $xab$ joins $a \neq b$ using exhaust by blast
{ assume $a = b$
  with $x \in E$ self-loop-no-forest insert(4) have False by auto
}
then have $xab'$: $a \neq b$ by auto
from insert(4) forest-mono have $fF$: forest $F$ by auto
with insert(3) have eq: card (CCs $F$) + card $F$ = card $V$ by auto
from insert(4) forest-subE have $k$: $F \subseteq E$ by auto
from $xab$ $xab'$ have $abV$: $a \in V \ b \in V$ using vertices-constr $E \text{-in} V \ x \in E$ by fast-force+
  have $(a, b) \notin$ connected $F$
  apply(subst augment-forest[symmetric])
  apply(rule $fF$)
  using $x \in E$ $xab$ $xab$ insert by auto
  with $k abV$ sameCC-reachable have $CC F a \neq CC F b$ by auto
  have $Suc$: card (CCs (insert $x F$)) = card (CCs $F$)
  apply(rule union-eqclass-decreases-components)
  by fact+
  then show $\exists case$ using $xab$ insert(1,2) eq by auto
qed (simp add: CCs-empty-card)

lemma pigeonhole-CCs:
  assumes finiteV: finite $V$ and cardlt: card (CCs $E1$) < card (CCs $E2$)
  shows $(\exists u v. u \in V \land v \in V \land CC E1 u = CC E1 v \land CC E2 u \neq CC E2 v)$
proof (rule ccontr, clarsimp)
  assume $\forall u. u \in V \longrightarrow (\forall v. CC E1 u = CC E1 v \longrightarrow v \in V \longrightarrow CC E2 u = CC E2 v)$
  then have $\forall u v. u \in V \Longrightarrow v \in V \Longrightarrow CC E1 u = CC E1 v \Longrightarrow CC E2 u = CC E2 v$ by blast
    with coarser[OF finiteV] have card (CCs (insert $x E1$) \ $V$) \ ge card (CCs (insert $x E2$) \ $V$) by blast
    with CCs-imageCC cardlt show False by auto
qed

2.2 The edge set and forest form the cycle matroid

theorem assumes $f1$: forest $E1$
  and $f2$: forest $E2$
  and $c$: card $E1 >$ card $E2$
  shows augment: $\exists e \in E1 - E2$. forest (insert $e E2$)
proof
  — as $E1$ and $E2$ are both forests, and $E1$ has more edges than $E2$, $E2$ has more connected components than $E1$
from forest-CCs[OF f1] forest-CCs[OF f2] c have card (CCs E1) < card (CCs E2) by linarith

— by an pigeonhole argument, we can obtain two vertices u and v that are in the same components of E1, but in different components of E2

then obtain \( u \), \( v \) where sameCCinE1: CC E1 u = CC E1 v \( \) and

diffCCinE2: CC E2 u \( \neq \) CC E2 v \( \) and \( k: u \in V \), \( v \in V \)

using pigeonhole-CCs[OF finiteV] by blast

from diffCCinE2 have \( u \neq v \) by auto

— this means that there is a path from \( u \) to \( v \) in E1 ...

from f1 forest-subE have e1: E1 \( \subseteq \) E by auto

with sameCC-reachable \( k \) sameCCinE1 have pathinE1: \((u, v) \in connected E1 \)

by auto

— ... but none in E2

from f2 forest-subE have e2: E2 \( \subseteq \) E by auto

with sameCC-reachable \( k \) diffCCinE2

have noPathinE2: \((u, v) \notin connected E2 \)

by auto

— hence, we can find vertices a and b that are not connected in E2, but are connected by an edge in E1

obtain a b e where pe: \((a, b) \notin connected E2 \) and abE2: \( e \notin E2 \)

and abE1: \( e \in E1 \) \( \) and joins a b e

using findaugmenting-aux[OF e1 e2 pathinE1 noPathinE2] by auto

with forest-subE[OF f1] have e \( \in \) E by auto

from abE1 abE2 have abdiff: \( e \in E1 \) \( - \) E2 by auto

with e1 have e \( \in \) E \( - \) E2 by auto

— we can savely add this edge between a and b to E2 and obtain a bigger forest

have forest (insert e E2) apply(subst augment-forest)

by fact+

then show \( \exists e \in E1 \) \( - \) E2. forest (insert e E2) using abdiff

by blast

qed

sublocale weighted-matroid E forest weight

proof

have forest \( \{ \} \) using forest-empty by auto

then show \( \exists X. \) forest X by blast

qed (auto simp: forest-subE forest-mono augment)

end — locale Kruskal-interface

end
3  Refine Kruskal

theory Kruskal-Refine
imports Kruskal SeprefUF
begin

3.1  Refinement I: cycle check by connectedness
As a first refinement step, the check for introduction of a cycle when adding
an edge $e$ can be replaced by checking whether the edge’s endpoints are
already connected. By this we can shift from an edge-centric perspective to
a vertex-centric perspective.

context Kruskal-interface
begin

abbreviation empty-forest ≡ {}

abbreviation a-endpoints e ≡ SPEC (λ(a,b). joins a b e )

definition kruskal0
  where kruskal0 ≡ do { 
    l ← obtain-sorted-carrier;
    spanning-forest ← nfoldli l (λ-. True)
    (λe T. do { 
      ASSERT (e ∈ E);
      (a,b) ← a-endpoints e;
      ASSERT (joins a b e ∧ forest T ∧ e∈E ∧ T ⊆ E);
      if ¬ (a,b) ∈ connected T then
        do { 
          ASSERT (e∉T);
          RETURN (insert e T)
        }
      else
        RETURN T
    }) empty-forest;
  RETURN spanning-forest
}

lemma if-subst: (if indep (insert e T) then
  RETURN (insert e T)
else
  RETURN T)
= (if e∉ T ∧ indep (insert e T) then
  RETURN (insert e T)
else
  RETURN T)

by auto
lemma kruskal0-refine: \((\text{kruskal0}, \text{minWeightBasis}) \in \langle \text{Id} \rangle \text{nres-rel}\)

unfolding kruskal0-def minWeightBasis-def
apply(subst if-subst)
apply refine-vcg
apply refine-dref-type
apply (all (auto; fail)?)
apply clarsimp
apply (auto simp: augment-forest)
using augment-forest joins-connected by blast+

3.2 Refinement II: connectedness by PER operation

Connectedness in the subgraph spanned by a set of edges is a partial equivalence relation and can be represented in a disjoint sets. This data structure is maintained while executing Kruskal’s algorithm and can be used to efficiently check for connectedness (per-compare).

definition corresponding-union-find :: 'a per ⇒ 'edge set ⇒ bool where
corresponding-union-find uf T ≡ (∀ a∈V. ∀ b∈V. per-compare uf a b ←→ ((a,b)∈ connected T))

definition uf-graph-invar uf-T ≡ case uf-T of (uf, T) ⇒ corresponding-union-find uf T ∧ Domain uf = V

lemma uf-graph-invarD: uf-graph-invar (uf, T) ⇒ corresponding-union-find uf T
unfolding uf-graph-invar-def by simp

definition uf-graph-rel ≡ br snd uf-graph-invar

lemma uf-graph-relsndD: ((a,b),c) ∈ uf-graph-rel ⇒ b=c
by(auto simp: uf-graph-rel-def in-br-conv)

lemma uf-graph-relD: ((a,b),c) ∈ uf-graph-rel ⇒ b=c ∧ uf-graph-invar (a,b)
by(auto simp: uf-graph-rel-def in-br-conv)

definition kruskal1
where kruskal1 ≡ do {
l ← obtain-sorted-carrier;
let initial-union-find = per-init V;
(per, spanning-forest) ← nfoldli l (λ- True)
(λe (uf, T). do {
  ASSERT (e ∈ E);
  (a,b) ← a-endpoints e;
  ASSERT (a∈V ∧ b∈V ∧ a ∈ Domain uf ∧ b ∈ Domain uf ∧ T⊆E);
  if ¬ per-compare uf a b then
do {
    let uf = per-union uf a b;
    ASSERT (e∈T);
    RETURN (uf, insert e T)
})
lemma corresponding-union-find-empty:
shows corresponding-union-find (per-init V) empty-forest
by (auto simp: corresponding-union-find-def connected-same per-init-def)

lemma empty-forest-refine: ((per-init V, empty-forest), empty-forest) ∈ uf-graph-rel
using corresponding-union-find-empty
unfolding uf-graph-rel-def uf-graph-invar-def
by (auto simp: in-br-conv per-init-def)

lemma uf-graph-invar-preserve:
assumes uf-graph-invar (uf, T) a ∈ V b ∈ V
joins a b e e ∈ E T ⊆ E
shows uf-graph-invar (per-union uf a b, insert e T)
using assms
by (auto simp add: uf-graph-invar-def corresponding-union-find-def
insert-reachable per-union-def)

theorem kruskal1-refine: (kruskal1, kruskal0) ∈ ⟨Id⟩ nres-rel
unfolding kruskal1-def kruskal0-def Let-def
apply (refine-rcg empty-forest-refine)
apply refine-dref-type
apply (auto dest: uf-graph-relD E-inV uf-graph-invarD
simp: corresponding-union-find-def uf-graph-rel-def
simp: in-br-conv uf-graph-invar-preserve)
by (auto simp: uf-graph-invar-def dest: joins-in-V)

end

end

4 Kruskal Implementation

theory Kruskal-Impl
imports Kruskal-Refine Refine-Imperative-HOL IICF
begin

4.1 Refinement III: concrete edges

Given a concrete representation of edges and their endpoints as a pair, we refine Kruskal’s algorithm to work on these concrete edges.
locale Kruskal-concrete = Kruskal-interface E V vertices joins forest connected

for E V vertices joins forest connected and weight :: 'edge ⇒ int +

fixes
  α :: 'edge ⇒ 'edge
  and endpoints :: 'edge ⇒ ('a*'a) nres

assumes
  endpoints-refine: α xi = x ⇒ endpoints xi ≤ ⇓ Id (a-endpoints x)

begin

definition wsorted' where wsorted' == sorted-wrt (λx y. weight (α x) ≤ weight (α y))

lemma wsorted-map[simp]: wsorted' s ⇒ wsorted (map α s)
  by(auto simp:wsorted'-def(sorted-wrt-map))

definition obtain-sorted-carrier' == SPEC (λL. wsorted' L ∧ α ' set L = E)

abbreviation concrete-edge-rel :: ('edge × 'edge) set where
  concrete-edge-rel ≡ br α (λ-. True)

lemma obtain-sorted-carrier'-refine:
  (obtain-sorted-carrier', obtain-sorted-carrier) ∈ ((concrete-edge-rel)list-rel)nres-rel

unfolding obtain-sorted-carrier'-def obtain-sorted-carrier-def
apply refine-vcg
apply (auto intro!: RES-refine simp:)
subgoal for s apply(rule exI[where x=map α s])
  by(auto simp:map-in-list-rel-conv in-br-conv)
done

definition kruskal2
  where kruskal2 ≡ do { l ← obtain-sorted-carrier';
    let initial-union-find = per-init V;
    (per, spanning-forest) ← nfoldli l (λ-. True)
    (λce (uf, T). do {
      ASSERT (α ce ∈ E);
      (a,b) ← endpoints ce;
      ASSERT (a∈V ∧ b∈V ∧ a ∈ Domain uf ∧ b ∈ Domain uf);
      if ¬ per-compare uf a b then
        do {
          let uf = per-union uf a b;
          ASSERT (ce ∉ set T);
          RETURN (uf, T@[ce])
        }
      else
        RETURN (uf,T)
    }) (initial-union-find, []);
    RETURN spanning-forest
  )
lemma lst-graph-rel-empty[simp]: ([], { }) ∈ ⟨concrete-edge-rel⟩ list-set-rel
unfolding list-set-rel-def apply (rule relcompI [where b = []])
by (auto simp add: in-br-conv)

lemma loop-initial-rel:
((per-init V, []), per-init V, { }) ∈ Id ×r ⟨concrete-edge-rel⟩ list-set-rel
by simp

lemma concrete-edge-rel-list-set-rel:
(a, b) ∈ ⟨concrete-edge-rel⟩ list-set-rel =⇒ \( \alpha ' \) (set) = b
by (auto simp: in-br-conv list-set-rel-def dest: list-relD2)

theorem kruskal2-refine: (kruskal2, kruskal1) ∈ ⟨⟨concrete-edge-rel⟩ list-set-rel⟩ nres-rel
unfolding kruskal1-def kruskal2-def Let-def
apply (refine-reg obtain-sorted-carrier 'refine [THEN nres-relD])
endpoints-refine loop-initial-rel)
by (auto intro!: list-set-rel-append dest: concrete-edge-rel-list-set-rel simp: in-br-conv)

end

4.2 Refinement to Imperative/HOL with Sepref-Tool

Given implementations for the operations of getting a list of concrete edges and getting the endpoints of a concrete edge we synthesize Kruskal in Imperative/HOL.

locale Kruskal-Impl = Kruskal-concrete E V vertices joins forest connected weight α endpoints
for E V vertices joins forest connected and weight :: 'edge ⇒ int
and α and endpoints :: nat × int × nat ⇒ (nat × nat) nres
+ fixes getEdges :: (nat × int × nat) list nres
and getEdges-impl :: (nat × int × nat) list Heap
and superE :: (nat × int × nat) set
and endpoints-impl :: (nat × int × nat) ⇒ (nat × nat) Heap
assumes
getEdges-refine: getEdges ≤ SPEC (λL. α ' set L = E
\& (\forall (a,wv,b)\in set L. weight (α (a,wv,b)) = wv) \& set L ⊆ superE)
and
getEdges-impl: (uncurry0 getEdges-impl, uncurry0 getEdges)
∈ unit-assn \rightarrow_a list-assn (nat-assn ×_a int-assn ×_a nat-assn)
and
max-node-is-Max-V: E = α ' set la =⇒ max-node la = Max (insert 0 V)
and
endpoints-impl: ( endpoints-impl, endpoints)
\[ \in (\text{nat-assn} \times_a \text{int-assn} \times_a \text{nat-assn})^k \rightarrow_a (\text{nat-assn} \times_a \text{nat-assn}) \]

**Lemma** this-loc: Kruskal-Impl E V vertices joins forest connected weight \( \alpha \) endpoints getEdges getEdges-impl superE endpoints-impl by unfold-locale

### 4.2.1 Refinement IV: given an edge set

We now assume to have an implementation of the operation to obtain a list of the edges of a graph. By sorting this list we refine obtain-sorted-carrier'.

**Definition** obtain-sorted-carrier'' = do \\
\[ l \leftarrow \text{SPEC} (\lambda L. \alpha \; \text{set} \; L = E \land (\forall (a,wv,b) \in \text{set} \; L. \; \text{weight} \; (\alpha \; (a,wv,b)) = wv) \land \text{set} \; L \subseteq \text{superE}); \]
\[ \text{SPEC} (\lambda L. \text{sorted-wrt edges-less-eq} \; L \land \text{set} \; L = \text{set} \; l) \]

**Lemma** wsorted'-sorted-wrt-edges-less-eq:  
**Assumes** \( \forall (a,wv,b) \in \text{set} \; s. \; \text{weight} \; (\alpha \; (a,wv,b)) = wv \)
**Shows** wsorted' \( s \)
**Using** assms apply – 
**Unfolding** usorted'-def 
**Unfolding** edges-less-eq-def 
**Apply** (rule sorted-wrt-mono-rel ) 
**By** (auto simp: case-prod-beta)

**Lemma** obtain-sorted-carrier''-refine: 
(\text{obtain-sorted-carrier''}, \text{obtain-sorted-carrier'} \in (\text{Id})\text{nres-rel}
**Unfolding** obtain-sorted-carrier''-def obtain-sorted-carrier'-def
**Apply** refine-vcg 
**Apply** (auto simp: in-br-conv usorted'-sorted-wrt-edges-less-eq 
distinct-map map-in-list-rel-conv)
**Done**

**Definition** obtain-sorted-carrier'''' = do \\
\[ l \leftarrow \text{getEdges}; \]
\[ \text{RETURN} \; (\text{quicksort-by-rel edges-less-eq} \; [] \; l, \text{max-node} \; l) \]

**Definition** add-size-rel = br fst (\( \lambda \) (l,n). \( n = \text{Max} \; (\text{insert} \; 0 \; V) \))

**Lemma** obtain-sorted-carrier''''-refine: 
(\text{obtain-sorted-carrier''''}, \text{obtain-sorted-carrier'''} \in (\text{add-size-rel})\text{nres-rel}
**Unfolding** obtain-sorted-carrier''''-def obtain-sorted-carrier'''-def
**Apply** (refine-reg getEdges-refine)
**By** (auto intro!: RETURN-SPEC-refine simp: quicksort-by-rel-distinct sort-edges-correct add-size-rel-def in-br-conv max-node-is-Max-V)
dest!: distinct-mapI)

lemmas osc-refine = obtain-sorted-carrier""-refine[FCOMP obtain-sorted-carrier""-refine, to-foparam, simplified]

definition kruskal3 :: (nat × int × nat) list nres
  where kruskal3 ≡ do {
    (sl,mn) ← obtain-sorted-carrier"";
    let initial-union-find = per-init' (mn + 1);
    (per, spanning-forest) ← nfoldli sl (λ-. True)
      (λce (uf, T). do {
        ASSERT (α ce ∈ E);
        (a,b) ← endpoints ce;
        ASSERT (a ∈ Domain uf ∧ b ∈ Domain uf);
        if ¬ per-compare uf a b then
          do {
            let uf = per-union uf a b;
            ASSERT (ce∈set T);
            RETURN (uf, T@[ce])
          }
        else
          RETURN (uf, T)
      }) (initial-union-find, []);
    RETURN spanning-forest
  }

lemma endpoints-spec: endpoints ce ≤ SPEC (λ-. True)
  by (rule order.trans[OF endpoints-refine], auto)

lemma kruskal3-subset:
  shows kruskal3 ≤ n SPEC (λT. distinct T ∧ set T ⊆ superE )
unfolding kruskal3-def obtain-sorted-carrier""-def
apply (refine-vcg getEdges-refine[THEN leaf-lift] endpoints-spec[THEN leaf-lift] nfoldli-leaf-rule[where I=λ-. (-, T), distinct T ∧ set T ⊆ superE ])
apply auto
subgoal
  by (metis append-self-conv in-set-conv-decomp set-quicksort-by-rel subset-iff)
subgoal
  by blast
done

definition per-supset-rel :: ('a per × 'a per) set where
  per-supset-rel
≡ {(p1,p2). p1 ∩ Domain p2 × Domain p2 = p2 ∧ p1 − (Domain p2 × Domain p2) ⊆ Id}

lemma per-supset-rel-dom: (p1, p2) ∈ per-supset-rel ⇒ Domain p1 ⊇ Domain

21
by (auto simp: per-supset-rel-def)

lemma per-supset-compare:
  \[(p_1, p_2) \in \text{per-supset-rel} \implies x_1 \in \text{Domain } p_2 \implies x_2 \in \text{Domain } p_2 \]
  \[\implies \text{per-compare } p_1 \ x_1 \ x_2 \iff \text{per-compare } p_2 \ x_1 \ x_2 \]
by (auto simp: per-supset-rel-def)

lemma per-supset-union:
  \[(p_1, p_2) \in \text{per-supset-rel} \implies x_1 \in \text{Domain } p_2 \implies x_2 \in \text{Domain } p_2 \implies \]
  \[(\text{per-union } p_1 \ x_1 \ x_2, \text{per-union } p_2 \ x_1 \ x_2) \in \text{per-supset-rel} \]
apply (clarsimp simp: per-supset-rel-def per-union-def Domain-unfold)
apply (intro subsetI conjI)
apply blast
apply force
done

lemma per-initN-refine:
  \[(\text{per-init'} (\text{Max } (\text{insert } 0 \ V) + 1), \text{per-init } V) \in \text{per-supset-rel} \]
unfolding per-supset-rel-def per-init\'_def per-init-def max-node-def
by (auto simp: less-Suc-eq-le)

theorem kruskal3-refine:
  \[(\text{kruskal3}, \text{kruskal2}) \in \langle \text{Id} \rangle \text{nres-rel} \]
unfolding kruskal2-def kruskal3-def Let-def
apply (refine-reg osc-refine [THEN nres-relD])
supply RELATESI [where R=\text{per-supset-rel}: (\text{nat } \times -) \text{ set},
refine-dref-RELATES]
apply refine-dref-type
subgoal by (simp add: add-size-rel-def in-br-conv)
subgoal using per-initN-refine by (simp add: add-size-rel-def in-br-conv)
by (auto simp add: add-size-rel-def in-br-conv per-supset-compare per-supset-union
dest: per-supset-rel-dom
simp del: per-compare-def)

4.2.2 Synthesis of Kruskal by SepRef

lemma [sepref-import-param]: \((\text{sort-edges}, \text{sort-edges}) \in \langle \text{Id } \times, \text{Id } \times, \text{Id} \rangle \text{list-rel} \to \langle \text{Id } \times, \text{Id } \times, \text{Id} \rangle \text{list-rel}\)
by simp

lemma [sepref-import-param]: \((\text{max-node}, \text{max-node}) \in \langle \text{Id } \times, \text{Id } \times, \text{Id} \rangle \text{list-rel} \to \text{nat-rel}\)
by simp

sepref-register getEdges :: (nat \times \text{ int } \times \text{ nat}) \text{ list nres}
sepref-register endpoints :: (nat \times \text{ int } \times \text{ nat}) \Rightarrow (\text{nat } \times \text{nat}) \text{ nres}

declare getEdges-impl [sepref-fr-rules]
declare endpoints-impl [sepref-fr-rules]

schematic-goal kruskal-impl:
  \((\text{uncurry0 } \text{tc}, \text{uncurry0 } \text{kruskal3}) \in (\text{unit-assn})^k \to \text{a list-assn } (\text{nat-assn } \times \text{a})\)
unfolding kruskal3-def obtain-sorted-carrier''-def
unfolding sort-edges-def[symmetric]
apply (rewrite at nfoldli - - - (-,rewrite-HOLE) HOL-list.fold-custom-empty)
by sepref

concrete-definition (in –) kruskal uses Kruskal-Impl.kruskal-impl
prepare-code-thms (in –) kruskal-def
lemmas kruskal-refine = kruskal.refine[OF this-loc]

abbreviation MSF == minBasis
abbreviation SpanningForest == basis
lemmas SpanningForest-def = basis-def
lemmas MSF-def = minBasis-def

lemmas kruskal3-ref-spec = kruskal3-refine[FCOMP kruskal2-refine, FCOMP kruskal1-refine,
FCOMP kruskal0-refine,
FCOMP minWeightBasis-refine]

lemma kruskal3-ref-spec'::
(uncurry0 kruskal3, uncurry0 (SPEC (λr. MSF (α ' set r)))) ∈ unit-rel →f
(Id)nres-rel
  unfolding fref-def
  apply auto
  apply(rule nres-relI)
  apply(rule order.trans[OF kruskal3-ref-spec-[unfolded fref-def, simplified, THEN nres-relD]])
  by (auto simp: conc-fun-def list-set-rel-def in-br-conv dest: list-relD2)

lemma kruskal3-ref-spec:
(uncurry0 kruskal3, uncurry0 (SPEC (λr. distinct r ∧ set r ⊆ superE ∧ MSF (α ' set r))))
∈ unit-rel →f (Id)nres-rel
  unfolding fref-def
  apply auto
  apply(rule nres-relI)
  apply simp
  using SPEC-rule-conj-leofI2[OF kruskal3-subset kruskal3-ref-spec']
  [unfolded fref-def, simplified, THEN nres-relD, simplified]]
  by simp

lemma [fcomp-norm-simps]: list-assn (nat-assn ×a int-assn ×a nat-assn) = id-assn
  by (auto simp: list-assn-pure-conv)

lemmas kruskal-ref-spec = kruskal-refine[FCOMP kruskal3-ref-spec]
The final correctness lemma for Kruskal’s algorithm.

**lemma** `kruskal-correct-forest`:

**shows**

\[
<\lambda r. \uparrow (\text{distinct } r \land \text{set } r \subseteq \text{super } E \land \text{MSF (set (map } \alpha r))\rangle, \leq
\]

**proof** –

- **show** `?thesis`
  - **using** `kruskal-ref-spec[to-hnr]`
  - **unfolding** `hn-refine-def`
  - **apply** `clar simp`
  - **apply** `(erule cons-post-rule)
    - by `(sep-auto simp: hn-ctxt-def pure-def list-set-rel-def in-br-conv dest: list-relD)`

**qed**

end — locale `Kruskal-Impl`

end

5 UGraph - undirected graph with Uprod edges

**theory** `UGraph`

**imports**

- `Automatic-Refinement.Misc`
- `Collections.Partial-Equivalence-Relation`
- `HOL-Library.Uprod`

**begin**

5.1 Edge path

**fun** `epath` :: `'a uprod set ⇒ 'a ⇒ ('a uprod) list ⇒ 'a ⇒ bool`

where

\[
\begin{align*}
\text{epath } E & \; u \; \_ \; v = (u = v) \\
\text{epath } E & \; x \# x s \; v \leftrightarrow (\exists w. u \neq w \land Upair u w = x \land \text{epath } E \; w \; x \; s \; v) \land x \in E
\end{align*}
\]

**lemma** `[simp, intro!]`: `epath E u \; \_ \; u` `by simp`

**lemma** `epath-subset-E`: `epath E u p v` `⇒` `set p ⊆ E`

**apply**(induct `p` arbitrary: `u`) `by auto`

**lemma** `path-append-conv[simp]`: `epath E u (p@q) v` `↔` `(∃ w. epath E u p w \land epath E w q v)`

**apply**(induct `p` arbitrary: `u`) `by auto`

**lemma** `epath-rev[simp]`: `epath E y (rev p) x = epath E x p y`

**apply**(induct `p` arbitrary: `x`) `by auto`

**lemma** `epath-mono` `: E ⊆ E'` `⇒` `epath E u p v` `⇒` `epath E' u p v`

24
apply (induct p arbitrary: u) by auto

lemma epath-restrict: set p ⊆ I ⇒ epath E u p v ⇒ epath (E ∩ I) u p v
apply (induct p arbitrary: u) by auto

lemma assumes A ⊆ A' ∼ epath A u p v epath A' u p v
shows epath-diff-edge: (∃ e. e ∈ set p - A)
proof (rule ccontr)
assume ¬ (∃ e. e ∈ set p - A)
then have i: set p ⊆ A by auto
have ii: A = A' ∩ A using assms (1) by auto
have epath A u p v apply (subst ii)
apply (rule epath-restrict ) by fact+
with assms (2) show False by auto
qed

lemma epath-restrict': epath (insert e E) u p v ⇒ e ∉ set p ⇒ epath E u p v
proof
assume a: epath (insert e E) u p v and e ∉ set p
then have b: set p ⊆ E by (auto dest: epath-subset-E)
have e: insert e E ∩ E = E by auto
show ?thesis apply (rule epath-restrict [where I = E and E = insert e E, simplified e])
  using a b by auto
qed

lemma epath-not-direct:
assumes ep: epath E u p v and unv: u ≠ v
and edge-notin: Upair u v ∉ E
shows length p ≥ 2
proof (rule ccontr)
from ep have setp: set p ⊆ E using epath-subset-E by fast
assume ¬ length p ≥ 2
then have length p < 2 by auto
moreover
{
  assume length p = 0
  then have p = [] by auto
  with ep unv have False by auto
}
moreover
{
  assume length p = 1
  then obtain e where p: p = [e]
  using list-decomp-1 by blast
  with ep have i: e = Upair u v by auto
  from p i setp and edge-notin have False by auto
}
ultimately show False by linarith

\textbf{lemma} epath-decompose:
\textbf{assumes} e: epath G v p v'
\textbf{and} elem : Upair a b \in set p
\textbf{shows} \exists u u' p'' p'''. u \in \{a, b\} \land u' \in \{a, b\} \land epath G v p' u \land epath G u' p'' v'' \land length p' < length p \land length p''' < length p

\textbf{proof} –
\textbf{from} elem \textbf{obtain} p' p''' \textbf{where} p: p = p' \# (Upair a b) \# p''' \textbf{using} in-set-conv-decomp \textbf{by} metis
\textbf{from} p \textbf{have} epath G v (p' \# (Upair a b) \# p''') v' using e \textbf{by} auto
\textbf{then} \textbf{obtain} z z' \textbf{where} \textbf{pr}: epath G v p' z epath G z' p''' v' and u: Upair z z' = Upair a b \textbf{by} auto
\textbf{from} u \textbf{have} u': z \in \{a, b\} \land z' \in \{a, b\} \textbf{by} auto
\textbf{have} \textbf{len}: length p' < length p length p''' < length p \textbf{using} p \textbf{by} auto
\textbf{from} len pr u' \textbf{show} \textbf{thesis} \textbf{by} auto
\textbf{qed}

\textbf{lemma} epath-decompose' :
\textbf{assumes} e: epath G v p v'
\textbf{and} elem : Upair a b \in set p
\textbf{shows} \exists u u' p'' . Upair a b = Upair u u' \land epath G v p' u \land epath G u' p'' v'' \land length p' < length p \land length p''' < length p

\textbf{proof} –
\textbf{from} elem \textbf{obtain} p' p''' \textbf{where} p: p = p' \# (Upair a b) \# p''' \textbf{using} in-set-conv-decomp \textbf{by} metis
\textbf{from} p \textbf{have} epath G v (p' \# (Upair a b) \# p''') v' using e \textbf{by} auto
\textbf{then} \textbf{obtain} z z' \textbf{where} \textbf{pr}: epath G v p' z epath G z' p''' v' and u: Upair z z' = Upair a b \textbf{by} auto
\textbf{from} u \textbf{have} u': z \in \{a, b\} \land z' \in \{a, b\} \textbf{by} auto
\textbf{have} \textbf{len}: length p' < length p length p''' < length p \textbf{using} p \textbf{by} auto
\textbf{from} len pr u' \textbf{show} \textbf{thesis} \textbf{by} auto
\textbf{qed}

\textbf{lemma} epath-split-distinct:
\textbf{assumes} epath G v p v'
\textbf{assumes} Upair a b \in set p
\textbf{shows} \exists p'' p''' . epath G v p'' u \land epath G a' p''' v' \land length p' < length p \land length p'' < length p \land (u \in \{a, b\} \land a' \in \{a, b\}) \land Upair a b \notin set p' \land Upair a b \notin set p''
using assms
proof (induction $n == \text{length } p$ arbitrary \(p \ v \ v'\) rule: nat-less-induct)

case 1

obtain $u \ u' \ p' \ p''$ where $u : u \in \{a, b\} \land u' \in \{a, b\}$
and $p' : \text{epath } G v p' u$ and $p'' : \text{epath } G u' p'' v'$
and $\text{len-p'} : \text{length } p' < \text{length } p$ and $\text{len-p''} : \text{length } p'' < \text{length } p$
using $\text{epath-decompose}[\text{OF } 1(2,3)]$ by blast

\begin{align*}
1 \ \text{len-p' p'} \ & \text{have } \
\text{Upair a b} \in \text{set } p' \rightarrow (\exists p'2 \ u2). \
\text{epath } G v p'2 u2 \land \
\text{length } p'2 < \text{length } p' \land \
\ u2 \in \{a, b\} \land \
\text{Upair a b} \notin \text{set } p'2 \\
\text{by metis}
\end{align*}

\begin{align*}
\text{with } \text{len-p' p'} u & \text{ have } p' : \exists p' u. \text{epath } G v p' u \land \text{length } p' < \text{length } p \land \\
& u \in \{a,b\} \land \text{Upair a b} \notin \text{set } p' \land \text{Upair a b} \notin \text{set } p' \\
\text{by fastforce}
\end{align*}

\begin{align*}
\text{from } 1 \ \text{len-p'' p''} \ & \text{have } \
\text{Upair a b} \in \text{set } p'' \rightarrow (\exists p''2 \ u'2). \
\text{epath } G u'2 p''2 v'' \land \
\text{length } p''2 < \text{length } p'' \land \
\ u'2 \in \{a, b\} \land \
\text{Upair a b} \notin \text{set } p''2 \land \text{Upair a b} \notin \text{set } p''2 \\
\text{by metis}
\end{align*}

\begin{align*}
\text{with } \text{len-p'' p'' u} & \text{ have } \exists p'' u'. \text{epath } G u' p'' u' \land \text{length } p'' < \text{length } p \land \\
& u' \in \{a,b\} \land \text{Upair a b} \notin \text{set } p'' \land \text{Upair a b} \notin \text{set } p'' \\
\text{by fastforce}
\end{align*}

\begin{align*}
\text{with } p' & \text{ show } \exists \text{case by auto}
\end{align*}

qed

5.2 Distinct edge path

definition $\text{depath } E \ u \ dp \ v \equiv \text{epath } E \ u \ dp \ v \land \text{distinct } dp$

lemma $\text{epath-to-depath: set } p \subseteq I \Rightarrow \text{epath } E \ u \ p \ v \Rightarrow \exists dp. \text{depath } E \ u \ dp \ v \land \\
\text{set } dp \subseteq I$

proof (induction $p$ rule: length-induct)

case (1 $p$)

\begin{align*}
hence IH: \& p'. &[\text{length } p' < \text{length } p; \text{set } p' \subseteq I; \text{epath } E \ u \ p' \ v] \\
\implies \exists p'. \text{depath } E \ u \ p' \ v \land \text{set } p' \subseteq I \\
\text{and PATH: epath } E \ u \ p \ v \\
\text{and set: set } p \subseteq I \\
\text{by auto}
\end{align*}

\begin{align*}
\exists \ p. \text{depath } E \ u \ p \ v \land \text{set } p \subseteq I
\end{align*}

proof cases

assume distinct $p$

thus $\exists \text{thesis using } \text{PATH set by } (\text{auto simp: } \text{depath-def})$

next

assume $\neg$(distinct $p$)

then obtain $pv1 \ pv2 \ pv3$ where $p : p = pv1 \at \ w \# pv2 @ w \# pv3$

\begin{align*}
\text{by } (\text{auto dest: not-distinct-decomp})
\end{align*}
with PATH obtain a where 1: epath E u pv1 a and 2: epath E a (w#pv2@w#pv3) v by auto
then obtain b where ab: w=Upair a b a≠b by auto
with 2 have epath E b (pv2@w#pv3) v by auto
then obtain c where 3: epath E b pv2 c and 4: epath E c (w#pv3) v by auto
then have cw: c∈set-uprod w by auto

moreover
{ assume c≠a

with ab cw have c=b by auto
with 4 ab have epath E a pv3 v by auto
then have length (pv1@w#pv3) < length p set (pv1@w#pv3) ⊆ I epath E u (pv1@w#pv3) v using 1 4 p set by auto
hence ∃p’, depath E u p’ v ∧ set p’ ⊆ I by (rule IH)
}

ultimately show ?case by auto
qed

lemma epath-to-depath': epath E u p v ⇒ ∃dp. depath E u dp v
using epath-to-depath[where I=set p] by blast

definition decycle E u p == epath E u p u ∧ length p > 2 ∧ distinct p

5.3 Connectivity in undirected Graphs

definition uconnected E ≡ {(u,v). ∃p. epath E u p v}

lemma uconnectedempty: uconnected {} = {(a,a)|a. True}
using epath.elims(2) by fastforce

lemma uconnected-refl: refl (uconnected E)
by(auto simp: refl-on-def uconnected-def)

lemma uconnected-sym: sym (uconnected E)
apply(clarsimp simp: sym-def uconnected-def)
subgoal for x y p apply (rule exI[where x=rev p]) by (auto) done

lemma uconnected-trans: trans (uconnected E)
apply(clarsimp simp: trans-def uconnected-def)
subgoal for x y p z q by (rule exI[where x=p@q], auto) done

lemma uconnected-symI: (u,v) ∈ uconnected E ⇒ (v,u) ∈ uconnected E
using uconnected-sym sym-def by fast

lemma equiv UNIV (uconnected E)
  apply (rule equivI)
  subgoal by (auto simp: refl-on-def uconnected-def)
  subgoal apply (clarsimp simp: sym-def uconnected-def) subgoal for x y p apply (rule exI[where x=rev p]) by auto done
  by (fact uconnected-trans)

lemma uconnected-refcl: (uconnected E)^* = (uconnected E)="
  apply (rule trans-rtrancl-eq-reflcl)
  by (fact uconnected-trans)

lemma uconnected-transcl: (uconnected E)^* = uconnected E
  apply (simp only: uconnected-refcl)
  by (auto simp: uconnected-def)

lemma uconnected-mono: A ⊆ A' ⇒ uconnected A ⊆ uconnected A'
  unfolding uconnected-def
  apply (auto)
  using epath-mono by metis

lemma findaugmenting-edge: assumes epath E1 u p v
  and ¬(∃ p. epath E2 u p v)
  shows ∃ a b. (a,b) /∈ uconnected E2 ∧ Upair a b /∈ E2 ∧ Upair a b ∈ E1
  using assms
  proof (induct p arbitrary: u)
  case Nil
  then show ?case by auto
  next
  case (Cons a p)
  then obtain w where axy: a=Upair u w u≠w and e': epath E1 w p v
  and uwE1: Upair u w ∈ E1 by auto
  show ?case
  proof (cases a∈E2)
  case True
  have e2': ¬(∃ p. epath E2 w p v)
  proof (rule ccontr, clarsimp)
    fix p2
    assume epath E2 w p2 v
    with True axy have epath E2 u (a#p2) v by auto
    with Cons(3) show False by blast
  qed
  from Cons(1)[OF e' e2'] show ?thesis .
  next
  case False
  {
assume $e'_2$: $\neg(\exists p. epath E_2 w p v)$
from $Cons(1)[OF e' e'_2]$ have \thesis.
}

moreover {
assume $e'_2$: $\exists p. epath E_2 w p v$
then obtain $p1$ where $p1$: $epath E_2 w p1 v$ by auto
}

moreover { have (u,w) $\notin$ uconnected $E_2$
proof (rule ccontr, auto simp add: uconnected-def)
fix $p2$
assume $epath E_2 u p2 w$
with $p1$ have $epath E_2 u (p2@p1) v$ by auto
then show False using $Cons(3)$ by blast
qed
moreover
note uwE1
ultimately have \thesis by auto
}

ultimately show \thesis by auto
qed
qed

5.4 Forest
definition forest $E$ $\equiv$ $\neg(\exists u p. decycle E u p)$

lemma forest-mono: $Y \subseteq X$ $\Longrightarrow$ forest $X$ $\Longrightarrow$ forest $Y$
unfolding forest-def decycle-def apply (auto) using epath-mono by metis

lemma forest2-E: assumes $(u,v) \in$ uconnected $E$
and $Upair u v \notin E$
and $u \neq v$
shows $\neg$ forest $(insert (Upair u v) E)$
proof --
from assms[unfolded uconnected-def] obtain $p'$ where $epath E u p' v$ by blast
then obtain $p$ where $ep$: $epath E u p v$ and $dep$: distinct $p$ using epath-to-depath'$
unfolding depath-def by fast
from $ep$ have setp: set $p \subseteq E$ using epath-subset-E by fast

have $lengthp$: $length p \geq 2$ apply(rule epath-not-direct) by fact+
from epath-mono[OF - $ep$] have $ep'$: $epath$ $(insert (Upair u v) E) u p v$ by auto

have $epath$ $(insert (Upair u v) E) v ((Upair u v)#p) v length ((Upair u v)#p)$
$> 2$ distinct $((Upair u v)#p)$
using $ep'$ assms(3) $lengthp$ dep setp assms(2) by auto
then have decycle $(insert (Upair u v) E) v ((Upair u v)#p)$ unfolding decycle-def by auto

30
then show ?thesis unfolding forest-def by auto

qed

lemma insert-stays-forest-means-not-connected: assumes forest (insert (Upair u v) E)
  and (Upair u v) ∉ E
  and u ≠ v
shows ¬(u, v) ∈ uconnected E
  using forest2-E assms by metis

lemma epath-singleton: epath F a [e] b ⟷ e = Upair a b
  by auto

lemma forest-alt1:
  assumes Upair a b ∈ F forest F \(\forall e. e\in F \implies \) proper-uprod e
  shows (a, b) ∉ uconnected (F − {Upair a b})
proof (rule ccontr)
  from assms(1,3) have anb: a≠b by force
  assume ¬(a, b) ∉ uconnected (F − {Upair a b})
  then obtain p where epath (F − {Upair a b}) a p b unfolding uconnected-def
  by blast
  then obtain p’ where dp: depath (F − {Upair a b}) a p b using epath-to-depath'
  by force
  then have ab: Upair a b ∉ set p' by(auto simp: depath-def dest: epath-subset-E)
  from ab dp have n0: length p' ≠ 0 by (auto simp: depath-def)
  from ab dp have n1: length p' ≠ 1 by (auto simp: depath-def simp del: One-nat-def)
  then have l: length p' ≥ 2 by linarith
  from dp have epath F a p' b by (auto intro: epath-mono simp: depath-def)
  then have e: epath F b (Upair a b#p') b using assms(1) anb by auto
  from dp ab dp have d: distinct (Upair a b#p') by (auto simp: depath-def)
  from d e l have decycle F b (Upair a b#p') by (auto simp: decycle-def)
  with assms(2) show False by (simp add: forest-def)
qed

lemma forest-alt2:
  assumes \(\forall e. e\in F \implies \) proper-uprod e
  and \(\forall a b. \quad\) Upair a b ∈ F \(\implies \) (a, b) ∉ uconnected (F − {Upair a b})
  shows forest F
proof (rule ccontr)
  assume ¬ forest F
  then obtain a p where e: epath F a p a length p > 2 distinct p
  unfolding decycle-def forest-def by auto
  then obtain b p' where p': p = Upair a b # p'
  by (metis Suc-1 epath.simps(2) less_imp_not_less list.size(3) neg-NilE zero_less_Suc)
  then have a: Upair a b∈ F using e(1) by auto
  then have F: (insert (Upair a b) F) = F by auto
  have epath (F − {Upair a b}) b p' a
apply (rule epath-restrict [where e = Upair a b]) using e p' by (auto simp: F)
then have epath (F − {Upair a b}) a (rev p') b by auto
with \ assms\(2)\{OF u\}
show False unfolding uconnected-def by blast
qed

lemma forest-alt:
  assumes \(\forall e. e\in F \implies \text{proper-uprod } e\)
  shows forest F ←→ (\(\forall a b. \text{Upair } a b \in F \implies (a,b) \notin uconnected (F − \{\text{Upair } a b\})\))
  using \ assms\ forest-alt1 forest-alt2
  by metis

lemma augment-forest-overedges:
  assumes \(F \subseteq E\) forest F (Upair u v) ∈ E (u,v) /∈ uconnected F
  and notsame: u≠v
  shows forest (insert (Upair u v) F) unfolding forest-def
proof (rule ccontr, clarsimp simp: decycle-def )
fix w p
assume d: distinct p and v: epath (insert (Upair u v) F) w p w and p: 2 < length p

have setep: set p ⊆ insert (Upair u v) F using epath-subset-E v
  by metis

have upF: (Upair u v)∉ F
proof (rule ccontr, clarsimp)
  assume (Upair u v) ∈ F
  then have epath F u [(Upair u v)] v using notsame by auto
  then have (u,v) ∈ uconnected F unfolding uconnected-def by blast
  then show False using assms(4) by auto
qed

have k: insert (Upair u v) F ∩ F = F by auto

show False
proof (cases)
  assume (Upair u v) ∈ set p
  then obtain as bs where ep: p = as @ (Upair u v) # bs using in-set-cone-decomp
  by metis
  then have epath (insert (Upair u v) F) w (as @ (Upair u v) # bs) w using v by auto
  then obtain z where pr: epath (insert (Upair u v) F) w as z epath (insert (Upair u v) F) z ((Upair u v) # bs) w by auto
  from d ep have wus: (Upair u v) ∉ set (as@bs) by auto
  then have setasbs: set (bs@as) ⊆ F using ep setep by auto
{ assume \( z = u \)
  with \( \text{pr} \) have \( \text{epath} \ (\text{insert} \ (\text{Upair} \ u \ v) \ F) \ w \ as \ u \) \( \text{epath} \ (\text{insert} \ (\text{Upair} \ u \ v) \ F) \ v \ bs \ w \) by auto
  then have \( \text{epath} \ (\text{insert} \ (\text{Upair} \ u \ v) \ F) \ v \ (bs@as) \ u \) by auto
  from \( \text{epath-restrict} [\text{where} \ I=F, \ OF \ \text{setasbs} \ this] \) have \( \text{epath} \ F \ v \ (bs@as) \ u \) by auto
using \( uvF \) by auto
  then have \( (v,u) \in \text{uconnected} \ F \) using \( \text{uconnected-def} \)
  by blast
  then have \( (u,v) \in \text{uconnected} \ F \) by (rule \( \text{uconnected-symI} \))
} moreover
{ assume \( z \neq u \)
  then have \( z = v \) using \( \text{pr}(2) \) by auto
  with \( \text{pr} \) have \( \text{epath} \ (\text{insert} \ (\text{Upair} \ u \ v) \ F) \ w \ as \ v \) \( \text{epath} \ (\text{insert} \ (\text{Upair} \ u \ v) \ F) \ u \ bs \ w \) by auto
  then have \( \text{epath} \ (\text{insert} \ (\text{Upair} \ u \ v) \ F) \ u \ (bs@as) \ v \) by auto
  from \( \text{epath-restrict} [\text{where} \ I=F, \ OF \ \text{setasbs} \ this] \) have \( \text{epath} \ F \ u \ (bs@as) \ v \) using \( uvF \) by auto
  then have \( (u,v) \in \text{uconnected} \ F \) using \( \text{uconnected-def} \)
  by fast
}
ultimately have \( (u,v) \in \text{uconnected} \ F \) by auto
then show \( \text{False} \) using \( \text{assms} \) by auto
next
  assume \( (\text{Upair} \ u \ v) \notin \text{set} \ p \)
  with \( \text{setep} \) have \( \text{set} \ p \subseteq \ F \) by auto
  then have \( \text{epath} \ (\text{insert} \ (\text{Upair} \ u \ v) \ F \cap F) \ w \ p \ w \) using \( \text{epath-restrict} [\text{OF} - \ v, \ \text{where} \ I=F] \) by auto
  then have \( \text{epath} \ F \ w \ p \ w \) using \( k \) by auto
  with \( \langle \text{forest} \ F \rangle \) show \( \text{False} \) unfolding \( \text{forest-def} \ \text{decycle-def} \) using \( p \ d \)
  by auto
qed
qed

5.5 \( \text{uGraph} \) locale
locale \( \text{uGraph} \) =
fixes \( E :: 'a \uprod \text{set} \)
  and \( w :: 'a \uprod \Rightarrow 'c::\{\text{linorder, ordered-comm-monoid-add}\} \)
assumes \( \text{ecard2}: \ \bigwedge e. \ e \in E \implies \text{proper-uprod} \ e \)
  and \( \text{finiteE}[\text{simp}]: \ \text{finite} \ E \)
begin
abbreviation \( \text{uconnected-on} \ E' V \equiv \text{uconnected} \ E' \cap (V \times V) \)
abbreviation \( \text{verts} \equiv \bigcup (\text{set-uprod} \ i \ E) \)
lemma \( \text{set-uprod-nonemptyY}[\text{simp}]: \ \text{set-uprod} \ x \neq \{\} \) apply(cases \( x \)) by auto

33
abbreviation \( u\text{connected} V E' \equiv \text{Restr} (u\text{connected} E') \text{ verts} \)

lemma \( \text{equiv-unconnected-on}: \text{equiv} V (u\text{connected-on} E' V) \)
apply (rule equivI)
subgoal by (auto simp: refl-on-def uconnected-def)

subgoal apply (clarsimp simp: sym-def uconnected-def) subgoal for \( x y p \)
apply (rule exI\[where \( x = \text{rev} p \)]\) by auto done

lemma \( u\text{connected} V\text{-refl}: E' \subseteq E \implies \text{refl-on} \text{ verts} (u\text{connected} V E') \)
by (auto simp: refl-on-def uconnected-def)

lemma \( u\text{connected} V\text{-trans}: \text{trans} (u\text{connected} V E') \)
apply (clarsimp simp: trans-def uconnected-def) subgoal for \( x y z p a b c q \)
apply (rule exI\[where \( x = p@q \)]\) by auto done

lemma \( u\text{connected} V\text{-sym}: \text{sym} (u\text{connected} V E') \)
apply (clarsimp simp: sym-def uconnected-def) subgoal for \( x y p \)
apply (rule exI\[where \( x = \text{rev} p \)]\) by (auto)

lemma \( u\text{connected} V\text{-r}\text{connected}: \text{equiv} \text{ verts} (u\text{connected} V E') \)
using \( \text{equiv-unconnected-on} \) by auto

lemma \( u\text{connected} V\text{-r}\text{transcl}: (u\text{connected} V F)^+ = (u\text{connected} V F)^\text{\textast} \)
apply (rule trans-rtrancl-eq-refl)
by (fact uconnectedV-trans)

lemma \( u\text{connected} V\text{-r}\text{cl}: (u\text{connected} V F)^+ = (u\text{connected} V F) \)
apply (rule trancl-id)
by (fact uconnectedV-trans)

lemma \( u\text{connected} V\text{-r}\text{Restrcl}: \text{Restr} ((u\text{connected} V F)^+) \text{ verts} = (u\text{connected} V F) \)
apply simp\ only: uconnectedV-tracl
apply auto unfolding \( u\text{connected-def} \) by auto

lemma \( \text{restr-ucon}: F \subseteq E \implies \text{uconnected} \text{ F = uconnected} V F \cup \text{Id} \)
unfolding \( u\text{connected-def} \) apply auto

proof (goal-cases)
case \( (1 \ a \ b \ p) \)
then have \( p \neq [] \) by auto
then obtain \( e \ es \ where \ p = e@es \)
using list.exhaust by blast
with \( 1(2) \) have \( a \in \text{set-uprod} e \ e \in F \) by auto
then show ?case using \( 1(1) \)
by blast

34
next
  case (2 a b p)
  then have \( \text{rev } p \neq [] \) \( \text{epath } F b (\text{rev } p) a \) \text{ by auto}
  then obtain \( e es \) \text{ where rev } p = e \# es
  using list.exhaust \text{ by metis}
with \( 2(2) \) have \( b \in \text{ set-cond } e \) \( e \in F \) \text{ by auto}
  then show \( \text{?case using } 2(1) \)
  by blast
qed

lemma relI:
  assumes \( \forall a. b. (a,b) \in F \implies (a,b) \in G \)
  and \( \forall a. b. (a,b) \in G \implies (a,b) \in F \)
  shows \( F = G \)
  using assms \text{ by auto}

lemma in-per-union: \( u \in \{ x, y \} \implies u' \in \{ x, y \} \implies x \in V \implies y \in V \)
  \text{ refl-on } V R \implies \text{ part-equiv } R \implies (u, u') \in \text{ per-union } R x y
  by \( \text{ (auto simp: per-union-def dest: refl-onD) } \)

lemma uconnectedV-mono: \( (a,b) \in \text{ uconnectedV } F \implies F \subseteq F' \implies (a,b) \in \text{ uconnectedV } F' \)
  unfolding \text{ uconnected-def by \text{(auto intro: epath-mono) } }

lemma per-union-subs: \( x \in S \implies y \in S \implies R \subseteq S \times S \implies \text{ per-union } R x y \subseteq S \times S \)
  unfolding \text{ per-union-def by auto }

lemma insert-uconnectedV-per:
  assumes \( x \neq y \) \text{ and } \text{ inV: } x \in \text{ verts } y \in \text{ verts and } subE: F \subseteq E
  shows \( \text{ uconnectedV } (\text{ insert } (\text{ Upair } x y) F) = \text{ per-union } (\text{ uconnectedV } F) x y \)
  (is \( \text{ uconnectedV } ?F' = \text{ per-union } ?uf x y \))
  proof
  have \( \text{ PER: part-equiv } (\text{ uconnectedV } F) \) \text{ unfolding part-equiv-def}
  using \( \text{ uconnectedV-sym } \text{ uconnectedV-trans by auto } \)
  from \( \text{ PER } \) have \( \text{ PER': part-equiv } (\text{ per-union } (\text{ uconnectedV } F) x y) \)
  by \( \text{ (auto simp: union-part-eqiup) } \)
  have \( \text{ ref: refl-on verts } (\text{ uconnectedV } F) \) \text{ using } \( \text{ uconnectedV-refl } \) \text{ assms(4) by auto } \)
  show \( \text{?thesis } \)
  proof \( \text{ (rule relI) } \)
    fix \( a \) \( b \)
    assume \( (a,b) \in \text{ uconnectedV } ?F' \)
    then obtain \( p \) \text{ where } p: \text{ epath } ?F' a p b \) \text{ and } \( ab: a \in \text{ verts } b \in \text{ verts } \)
    unfolding \( \text{ uconnected-def } \)
    by \text{ blast }
    show \( (a,b) \in \text{ per-union } (\text{ uconnectedV } F) x y \)
  proof \( \text{ (cases } \text{ Upair } x y \in \text{ set } p ) \)
case \( \text{True} \)

obtain \( p' \) \( p'' \) \( u' \) \( u'' \) where
\[ \text{epath } ?F' \text{ a } p' \text{ u epath } ?F' \text{ u' } p'' \text{ band} \]
\[ u: u \in \{x, y\} \land u': u' \in \{x, y\} \text{ and} \]
\[ \text{Upair } x \ y \notin \text{ set } p' \text{ Upair } x \ y \notin \text{ set } p'' \]
using epath-split-distinct[of \( p \) \( \text{True} \)] by blast
then have epath \( F \) \( a \) \( p' \) \( u \) epath \( F \) \( u' \) \( p'' \) \( b \) by (auto intro: epath-restrict')
then have \( a: (a, u) \in (\text{unconnectedV } F) \) and \( b: (u', b) \in (\text{unconnectedV } F) \)
unfolding unconnected-def using \( u \) \( ab \) assms by blast

then have epath \( F \) \( a \) \( p \) \( b \) by (auto intro: epath-restrict')
then have \( (a, u) \in \text{per-union } ?uf \) \( x \) \( y \) apply (rule in-per-union) using \( u \) \( \text{inV ref} \)
PER by auto
also (part-equiv-trans[of \( \text{OF } \) \( \text{PER} \)']) have \( (u', b) \in \text{per-union } ?uf \) \( x \) \( y \) using \( b \) by (auto simp: per-union-def)
finally (part-equiv-trans[of \( \text{OF } \) \( \text{PER} \)']) show \( (a, b) \in \text{per-union } ?uf \) \( x \) \( y \).

next

case \( \text{False} \)
with \( p \) have epath \( F \) \( a \) \( p \) \( b \) by (auto intro: epath-restrict')
then have \( (a, b) \in \text{unconnectedV } F \) using \( ab \) by (auto simp: unconnected-def)
then show ?thesis unfolding per-union-def by auto
qed

next

fix \( a \) \( b \)
assume asm: \( (a, b) \in \text{per-union } ?uf \) \( x \) \( y \)
have per-union ?uf \( x \) \( y \) \( \subseteq \) verts \( \times \) verts apply (rule per-union-subs)
using \( \text{inV ref} \) by auto
with asm have \( ab \): \( a \in \text{verts} \) \( b \in \text{verts} \) by auto
have \( \text{Upair } x \ y \in ?F' \) by simp
show \( (a, b) \in \text{unconnectedV } ?F' \)
proof (cases \( (a, b) \in ?uf \))


case \( \text{True} \)
then show ?thesis using unconnectedV-mono by blast

next

case \( \text{False} \)
with asm part-equiv-sym[of \( \text{OF } \) \( \text{PER} \)] have \( (a, x) \in ?uf \land (y, b) \in ?uf \lor (a, y) \in ?uf \land (x, b) \in ?uf \)
by (auto simp: per-union-def)
with \( \text{assms}(1) \): \( x \in \text{verts} \) \( y \in \text{verts} \) in \( V \) obtain \( p \) \( q \) \( p' \) \( q' \)
where epath \( F \) \( a \) \( p \) \( x \) \( \land \) epath \( F \) \( y \) \( q \) \( b \) \lor epath \( F \) \( a \) \( p' \) \( y \) \( \land \) epath \( F \) \( x \) \( q' \) \( b \)
unfolding unconnected-def
by fastforce
then have epath ?F' \( a \) \( p \) \( x \) \( \land \) epath ?F' \( y \) \( q \) \( b \) \lor epath ?F' \( a \) \( p' \) \( y \) \( \land \) epath ?F' \( x \) \( q' \) \( b \)
by (auto intro: epath-mono)
then have 2: epath ?F' \( a \) \( (p \@ \text{Upair } x \ y \# q) \) \( b \) \lor epath ?F' \( a \) \( (p' \@ \text{Upair } x \ y \# q) \) \( b \)
by (auto simp: per-union-def)
lemma epath-filter-selfloop: epath (insert (Upair x x) F) a p b \implies \exists p. epath F a p b
proof (induction n == length p arbitrary: p rule: nat-less-induct)
case 1
  from 1(1) have indhyp:
    \forall xa. length xa < length p \implies epath (insert (Upair x x) F) a xa b \implies (\exists p. epath F a p b) by auto
  from 1(2) have k: set p \subseteq (insert (Upair x x) F) using epath-subset-E by fast
  { assume a: set p \subseteq F
    have F: (insert (Upair x x) F \cap F) = F by auto
    from epath-restrict[OF a 1(2)] F have epath F a p b by simp
    then have (\exists p. epath F a p b) by auto
  }
  moreover
  { assume \neg set p \subseteq F
    with k have Upair x x \in set p by auto
    then obtain xs ys where p: p = xs @ Upair x x # ys
      by (meson split-list-last)
    then have epath (insert (Upair x x) F) a xs x epath (insert (Upair x x) F) x ys b
      using 1.prems by auto
    then have epath (insert (Upair x x) F) a (xs@ys) b by auto
    from indhyp[OF this] p have (\exists p. epath F a p b) by simp
  }
  ultimately show \?thesis by auto
qed

lemma equiv-selfloop-per-union-id: equiv S F \implies x \in S \implies per-union F x x = F
apply rule
subgoal unfolding equiv-def apply auto
subgoal unfolding per-union-def apply auto
using equiv-class-eq-iff by fastforce
subgoal unfolding per-union-def by auto
done

lemma insert-uconnectedV-per-eq:
assumes inV: x∈verts and subE: F⊆E
shows uconnectedV (insert (Upair x x) F) = per-union (uconnectedV F) x x
using assms
by(simp add: uconnectedV-insert-selfloop equiv-selfloop-per-union-id[OF equiv-vert-uconnected])

lemma insert-uconnectedV-per′:  
assumes inV: x∈verts y∈verts and subE: F⊆E
shows uconnectedV (insert (Upair x y) F) = per-union (uconnectedV F) x y
apply(cases x=y)
subgoal using assms insert-uconnectedV-per-eq by simp
subgoal using assms insert-uconnectedV-per by simp
done

definition subforest F ≡ forest F ∧ F ⊆ E

definition spanningForest where spanningForest X ←→ subforest X ∧ (∀ x ∈ E − X. ¬ subforest (insert x X))

definition minSpanningForest F ≡ spanningForest F ∧ (∀ F′. spanningForest F′ →→ sum w F ≤ sum w F′)

end

end

6 Kruskal on UGraphs

theory UGraph-Impl
imports
Kruskal-Impl UGraph
begin

definition α = (λ(u,w,v). Upair u v)

6.1 Interpreting Kruskal-Impl with a UGraph

abbreviation (in uGraph)
getEdges-SPEC csuper-E
≡ (SPEC (λL. distinct (map α L) ∧ α' set L = E
∧ (∀ (a,wv,b)∈set L w (α (a, wv, b)) = wv) ∧ set L ⊆ csuper-E))

locale uGraph-impl = uGraph E w for E :: nat uprod set and w :: nat uprod ⇒
int +
fixes getEdges-impl :: (nat × int × nat) list Heap and csuper-E :: (nat × int × nat) set
assumes getEdges-impl:
(uncurry0 getEdges-impl, uncurry0 (getEdges-SPEC csuper-E))
∈ unit-assn k →a list-assn (nat-assn ×a int-assn ×a nat-assn)

begin

abbreviation V ≡ (UNION E set-uprod)

lemma max-node-is-Max-V: E = α · set la ⇒ max-node la = Max (insert 0 V)
proof –
assume E: E = α · set la
have *: fst · set la ∪ (snd ◦ snd) · set la = (∪x∈set la. case x of (x1, x1a, x2a) ⇒ {x1, x2a})
by auto force
show ?thesis
unfolding E using *
by (auto simp add: α-def max-node-def prod.case-distrib)
qed

sublocale s: Kruskal-Impl E ∪(set-uprod · E) set-uprod λu v e. Upair u v = e
subforest aconnectedV w α PR-CONST (λ(u,w,v). RETURN (u,v))
PR-CONST (getEdges-SPEC csuper-E)
getEdges-impl csuper-E (λ(u,w,v). return (u,v))
unfolding subforest-def
proof (unfold-locales, goal-cases)
show finite E by simp
next
fix E’
assume forest E’ ∧ E’ ⊆ E
then show E’ ⊆ E by auto
next
show forest {} ∧ {} ⊆ E apply (auto simp: decycle-def forest-def)
using epath.elims(2) by fastforce
next
fix X Y
assume forest X ∧ X ⊆ E Y ⊆ X
then show forest Y ∧ Y ⊆ E using forest-mono by auto
next
case (5 u v)
then show ?case unfolding uconnected-def apply auto
using epath.elims(2) by force

next
case (6 E1 E2 u v)
then have (u, v) ∈ (uconnected E1) and uv: u ∈ (UNION E set-uprod) v∈(UNION E set-uprod)
  by auto
then obtain p where 1: epath E1 u p v unfolding uconnected-def by auto
from 6 uv have 2: ¬(∃ p. epath E2 u p v) unfolding uconnected-def by auto
from 1 2 have ∃ a b. (a, b) ∉ uconnected E2
  ∧ Upair a b ∉ E2 ∧ Upair a b ∈ E1 by(rule findaugmenting-edge)
then show ?case by auto

next
case (7 F e u v)

note f = ⟨forest F ∧ F ⊆ E⟩

note notin = ⟨e ∈ E − F⟩ (Upair u v = e)

from notin ecard2 have unv: u ≠ v by fastforce

show (forest (insert e F) ∧ insert e F ⊆ E) = ((u, v) ∉ uconnectedV F)

proof
  assume a: forest (insert e F) ∧ insert e F ⊆ E
  have (u, v) ∉ uconnected F apply(rule insert-stays-forest-means-not-connected)
    using notin a unv by auto
  then show ( ((u, v) ∉ Restr (uconnected F) (UNION E set-uprod)) ) by auto

next
  assume a: (u, v) ∉ Restr (uconnected F) (UNION E set-uprod)
  have forest (insert (Upair u v) F) apply(rule augment-forest-overedges[where E=E])
    using notin f a unv by auto
  moreover have insert e F ⊆ E
    using notin f by auto
  ultimately show forest (insert e F) ∧ insert e F ⊆ E using notin by auto
qed

next
fix F

assume F ⊆ E

show equiv V (uconnectedV F) by(rule equiv-vert-uconnected)

next
case (9 F)
then show ?case by auto

next
case (10 x y F)
then show ?case using insert-uconnectedV-per′ by metis

next
case (11 x)
then show ?case apply(cases x) by auto

next
case (12 u v e)
then show ?case by auto
next
case (13 u v e)
then show ?case by auto
next
case (14 a F e)
then show ?case using ecard2 by force
next
case (15 v)
then show ?case using ecard2 by auto
next
case 16
show V ⊆ V by auto
next
case 17
show finite V by simp
next
case (18 a b e T)
then show ?case
apply auto
subgoal unfolding uconnected-def apply auto apply (rule exI [where x = e])
apply simp
using ecard2 by force
subgoal by force
subgoal by force
done
next
case (19 xi x)
then show ?case by (auto split: prod.splits simp: α-def)
next
case 20
show ?case by auto
next
case 21
show ?case using getEdges-impl by simp
next
case (22 l)
from max-node-is-Max-V[OF 22] show max-node l = Max (insert 0 V).
next
case (23)
then show ?case
apply sepref-to-hoare by sep-auto
qed

lemma spanningForest-eq-basis: spanningForest = s.basis
unfolding spanningForest-def s.basis-def by auto

lemma minSpanningForest-eq-minbasis: minSpanningForest = s.minBasis
unfolding minSpanningForest-def s.MSF-def spanningForest-eq-basis by auto

lemma kruskal-correct':
<emp> kruskal getEdges-impl (λ(u, w, v). return (u, v)) ()
\[
<\lambda r. \uparrow (\text{distinct } r \land \text{set } r \subseteq \text{csuper-E} \land \text{s.MSF } (\text{set } (\text{map } \alpha r)))>_t
\]

using \text{s.kruskal-correct-forest} by auto

lemma \text{kruskal-correct}:
\[
<\lambda r. \uparrow (\text{distinct } r \land \text{set } r \subseteq \text{csuper-E} \land \text{minSpanningForest } (\text{set } (\text{map } \alpha r)))>_t
\]

using \text{s.kruskal-correct-forest minSpanningForest-eq-minbasis} by auto

end

6.2 Kruskal on UGraph from list of concrete edges

definition \text{uGraph-from-list-\(\alpha\)-weight} \(L\ e = (\text{THE } w. \exists a' b'. \text{Upair } a' b' = e \land (a', w, b') \in \text{set } L)\)

abbreviation \text{uGraph-from-list-\(\alpha\)-edges} \(L \equiv \alpha' \text{ set } L\)

locale \text{fromlist} = fixes
\(L :: (\text{nat } \times \text{int } \times \text{nat}) \text{ list}\)
assumes dist: \text{distinct } (\text{map } \alpha L) and \text{no-selfloop: } \forall u w v. (u, w, v) \in \text{set } L \rightarrow u \neq v

begin

lemma \text{not-distinct-map}: \(a \in \text{set } l \Rightarrow b \in \text{set } l \Rightarrow a \neq b \Rightarrow \alpha a = \alpha b \Rightarrow \neg \text{distinct } (\text{map } \alpha l)\)

by (meson \text{distinct-map-eq})

lemma \text{ii}: \((a, aa, b) \in \text{set } L \Rightarrow \text{uGraph-from-list-\(\alpha\)-weight } L (\text{Upair } a b) = aa\)

unfolding \text{uGraph-from-list-\(\alpha\)-weight-def}

apply rule
subgoal by auto
apply clarify
subgoal for \(w a' b'\)
apply(auto)
subgoal using \text{distinct-map-eq}[OF dist, of \((a, aa, b) (a, w, b)\)]

unfolding \text{\(\alpha\)-def} by auto
subgoal using \text{distinct-map-eq}[OF dist, of \((a, aa, b) (a', w, b')\)]

unfolding \text{\(\alpha\)-def} by fastforce
done
done

sublocale \text{uGraph-impl} \(\alpha' \text{ set } L\) \text{uGraph-from-list-\(\alpha\)-weight} \(L\ \text{return } L \text{ set } L\)

proof (unfold-locales)
fix \(e\) assume \(\ast : e \in \alpha' \text{ set } L\)

from \(\ast\) obtain \(u w v\) where \((u, w, v) \in \text{set } L e = \alpha (u, w, v)\) by auto
then show \text{proper-aprod } e using \text{no-selfloop unfolding } \text{\(\alpha\)-def} by auto
next

show finite \((\alpha' \text{ set } L)\) by auto
next

42
show \( (\text{uncurry} \theta \ (\text{return} \ L), \text{uncurry} \theta ((\text{SPEC} (X \alpha. \ \text{distinct} \ (\text{map} \ \alpha \ X) \land \ \alpha \ \cdot \ \text{set} \ X = \ \alpha \ \cdot \ \text{set} \ L \land (\forall (aa, vv, ba) \in \text{set} \ X. \ \text{uGraph-from-list-\alpha-weight} \ L (\alpha \ (aa, vv, ba)) = vv) \land \ \text{set} \ X \subseteq \text{set} \ L))) \in \text{unit-assn} k \rightarrow \text{a list-assn}) (\alpha \rightarrow \text{nat-assn} \times \alpha \rightarrow \text{int-assn} \times \alpha \rightarrow \text{nat-assn})) \)

apply sepref-to-hoare using dist apply sep-auto
subgoal using ii unfolding \( \alpha \)-def by auto
subgoal by simp
subgoal by (auto simp: pure-fold list-assn-emp)
done
qed

lemmas kruskal-correct = kruskal-correct

definition \((\text{in} -)\) kruskal-algo \( L \) = kruskal (\( \text{return} \ L \)) (\( \lambda(u,w,v). \ \text{return} \ (u,v) \)) ()

end

6.3 Outside the locale

definition \( \text{uGraph-from-list-invar} :: \ (\text{nat} \times \text{int} \times \text{nat}) \ \text{list} \Rightarrow \text{bool} \) where
\( \text{uGraph-from-list-invar} \ L = (\text{distinct} \ (\text{map} \ \alpha \ L) \land (\forall p \in \text{set} \ L. \ \text{case} \ p \ o \ (\alpha, w, v) \Rightarrow u \neq v)) \)

lemma \( \text{uGraph-from-list-invar-conv}: \text{uGraph-from-list-invar} \ L = \text{fromlist} \ L \)
by (auto simp add: \( \text{uGraph-from-list-invar-def} \) fromlist-def)

lemma \( \text{uGraph-from-list-invar-subset}: \text{uGraph-from-list-invar} \ L \Rightarrow \text{set} \ L \subseteq \text{set} \ L \Rightarrow \text{distinct} \ L' \Rightarrow \text{uGraph-from-list-invar} \ L' \)
unfolding \( \text{uGraph-from-list-invar-def} \) by (auto simp: distinct-map inj-on-subset)

lemma \( \text{uGraph-from-list-\alpha-inj-on}: \text{uGraph-from-list-invar} \ E \Rightarrow \text{inj-on} \ \alpha \ (\text{set} \ E) \)
by (auto simp: distinct-map \( \text{uGraph-from-list-invar-def} \) )

lemma \( \text{sum-easier}: \text{uGraph-from-list-invar} \ L \)
\( \Rightarrow \text{set} \ E \subseteq \text{set} \ L \)
\( \Rightarrow \text{sum} \ (\text{uGraph-from-list-\alpha-weight} \ L) \ (\text{uGraph-from-list-\alpha-edges} \ E) = \text{sum} \ (\lambda(u,w,v). \ w) \ (\text{set} \ E) \)
proof -
assume \( a: \text{uGraph-from-list-invar} \ L \)
assume \( b: \text{set} \ E \subseteq \text{set} \ L \)

have \( \ast: \ \land e. \ e \in \text{set} \ E \Rightarrow (\langle \lambda e. \ \text{THE} \ w. \ \exists a' \ b'. \ \text{Upair} \ a' \ b' = e \land (a', w, b') \in \text{set} \ L \rangle \circ \alpha) \ e \Rightarrow (\text{case} \ e \ o \ (u, w, v) \Rightarrow w) \)
apply simp

43
apply (rule the-equality)
subgoal using b by (auto simp: α-def split: prod.splits)
subgoal using a b apply (auto simp: uGraph-from-list-invar-def distinct-map
  split: prod.splits)
  using α-def
    by (smt α-def inj-onD old.prod.case prod.inject set-mp)
done

have inj-on-E: inj-on α (set E)
  apply (rule inj-on-subset)
  apply (rule uGraph-from-list-α-inj-on) by fact+
show ?thesis
  unfolding uGraph-from-list-α-weight-def
  apply (subst sum.reindex[OF inj-on-E])
    by auto
  using *
  qed

lemma corr: uGraph-from-list-invar L \implies
  <\emptyset> kruskal-algo L
    \(\lambda F. \uparrow (uGraph-from-list-invar F \land set F \subseteq set L \land
        uGraph.minSpanningForest (uGraph-from-list-α-edges L)
            (uGraph-from-list-α-weight L) (uGraph-from-list-α-edges F))>_{\lambda}
  apply(sep-auto heap: fromlist.kruskal-correct
    simp: uGraph-from-list-invar-cone kruskal-algo-def )
  using uGraph-from-list-invar-subset uGraph-from-list-invar-conv by simp

lemma uGraph-from-list-invar L \implies
  <\emptyset> kruskal-algo L
    \(\lambda F. \uparrow (uGraph-from-list-invar F \land set F \subseteq set L \land
        uGraph.spanningForest (uGraph-from-list-α-edges L) (uGraph-from-list-α-edges F)
            \land (\forall F'. uGraph.spanningForest (uGraph-from-list-α-edges L) (uGraph-from-list-α-edges F')
            \rightarrow set F' \subseteq set L \rightarrow sum (\lambda(u,w,v). w) (set F') \leq sum (\lambda(u,w,v). w)\)
    (set F'))>_{\lambda}
proof -
  assume a: uGraph-from-list-invar L
then interpret fromlist L apply unfold-locales by (auto simp: uGraph-from-list-invar-def)
from a show ?thesis
  by(sep-auto heap: corr simp: minSpanningForest-def sum-easier)
qed
6.4 Kruskal with input check

**definition** \(\text{kruskal}' L = \text{kruskal} (\lambda(u,w,v). \text{return} (u,v)) ()\)

**definition** \(\text{kruskal-checked} L = \text{if } \text{uGraph-from-list-invar} L \text{ then do } \{ F \leftarrow \text{kruskal}' L; \text{return} \ (\text{Some} \ F) \} \text{ else return None} \)

**lemma** <emp> \(\text{kruskal-checked} L < \lambda\)
\(\text{Some} \ F \Rightarrow \uparrow (\text{uGraph-from-list-invar} L \land \text{set} F \subseteq \text{set} L)
\land \text{uGraph.minSpanningForest} (\text{uGraph-from-list-\alpha-edges} L) (\text{uGraph-from-list-\alpha-weight} L)
\)
\((\text{uGraph-from-list-\alpha-edges} F))
\| \text{None} \Rightarrow \uparrow (\neg \text{uGraph-from-list-invar} L)\rangle_t

**unfolding** \(\text{kruskal-checked-def}\)
\(\text{apply} (\text{cases uGraph-from-list-invar} L) \text{ apply simp-all}\)

**subgoal proof** –
\(\text{assume} \ [\text{simp}]: \text{uGraph-from-list-invar} L\)
\(\text{then interpret fromlist} L \text{ apply unfold-locales by} (\text{auto simp: uGraph-from-list-invar-def})\)
\(\text{show} \ ?\text{thesis unfolding kruskal'-def by} (\text{sep-auto heap: kruskal-correct})\)

**qed**
**subgoal by** \(\text{sep-auto}\)
**done**

6.5 Code export

**export-code** \text{uGraph-from-list-invar checking SML-imp}

**export-code** \text{kruskal-checked checking SML-imp}

**ML-val (**
\(\text{val export-nat} = @\{\text{code integer-of-nat}\}\)
\(\text{val import-nat} = @\{\text{code nat-of-integer}\}\)
\(\text{val export-int} = @\{\text{code integer-of-int}\}\)
\(\text{val import-int} = @\{\text{code int-of-integer}\}\)
\(\text{val import-list} = \text{map} \ (\text{fn} \ (\text{a,b,c}) \Rightarrow (\text{import-nat} \ a, (\text{import-int} \ b, \text{import-nat} \ c)))\)
\(\text{val export-list} = \text{map} \ (\text{fn} \ (\text{a,b,c}) \Rightarrow (\text{export-nat} \ a, (\text{export-int} \ b, \text{export-nat} \ c)))\)
\(\text{val export-Some-list} = (\text{fn} \ \text{SOME} \ l \Rightarrow \text{SOME} \ (\text{export-list} \ l) \mid \text{NONE} \Rightarrow \text{NONE})\)

\(\text{fun} \ \text{kruskal} \ l = @\{\text{code kruskal}\} \ (\text{fn} \ () \Rightarrow \text{import-list} \ l) \ (\text{fn} \ (\cdot,c)) \Rightarrow \text{fn} ()\)
\(\Rightarrow (a,c)) () ()\)
\(\mid > \text{export-list}\)
\(\text{fun} \ \text{kruskal-checked} \ l = @\{\text{code kruskal-checked}\} \ (\text{import-list} \ l) () \mid > \text{export-Some-list}\)

\(\text{val result} = \text{kruskal} [(\text{1,~9,2}),(\text{2,~3,3}),(\text{3,~4,1})]\)
\(\text{val result4} = \text{kruskal} [(\text{1,~100,4}), (\text{3,64,5}), (\text{1,13,2}), (\text{3,20,2}), (\text{2,5,5}), (\text{4,80,3}), (\text{4,40,5})]\)
val result' = kruskal-checked [(1,~9,2),(2,~3,3),(3,~4,1)]
val result1' = kruskal-checked [(1,~9,2),(2,~3,3),(3,~4,1),(1,5,3)]
val result2' = kruskal-checked [(1,~9,2),(2,~3,3),(3,~4,1),(3,~4,1)]
val result3' = kruskal-checked [(1,~9,2),(2,~3,3),(3,~4,1),(1,~4,1)]
val result4' = kruskal-checked [(1,~100,4),(3,64,5),(1,13,2),(3,20,2),
(2,5,1),(4,80,3),(4,40,5)]
⟩

end

7 Undirected Graphs as symmetric directed graphs

theory Graph-Definition
imports
    Dijkstra-Shortest-Path, Graph
begin

7.1 Definition

fun is-path-undir :: ('v,'w) graph ⇒ 'v ⇒ ('v,'w) path ⇒ 'v ⇒ bool
  where
  is-path-undir G v [] v' ←→ v = v' ∧ v' ∈ nodes G
  is-path-undir G v [(v1,w,v2)#p] v'
    ←→ v = v1 ∧ ((v1,w,v2) ∈ edges G ∨ (v2,w,v1) ∈ edges G) ∧ is-path-undir G v2 p v'

abbreviation nodes-connected G a b ≡ ∃p. is-path-undir G a p b

definition degree :: ('v,'w) graph ⇒ 'v ⇒ nat
  where
  degree G v = card { e ∈ edges G. fst e = v ∨ snd (snd e) = v }

locale forest = valid-graph G
  for G :: ('v,'w) graph +
  assumes cycle-free:
    ∀(a,w,b) ∈ E. ¬ nodes-connected (delete-edge a w b G) a b

locale connected-graph = valid-graph G
  for G :: ('v,'w) graph +
  assumes connected:
    ∀v ∈ V. ∀ v' ∈ V. nodes-connected G v v'

locale tree = forest + connected-graph

locale finite-graph = valid-graph G
  for G :: ('v,'w) graph +
  assumes finite-E: finite E and
          finite-V: finite V

46
locale finite-weighted-graph = finite-graph G
for G :: ('v, 'w::weight) graph

definition subgraph :: ('v, 'w) graph ⇒ ('v, 'w) graph ⇒ bool where
subgraph G H ≡ nodes G = nodes H ∧ edges G ⊆ edges H

definition edge-weight :: ('v, 'w) graph ⇒ 'w::weight where
edge-weight G ≡ sum (fst o snd) (edges G)

definition edges-less-eq :: ('a × 'w::weight × 'a) ⇒ ('a × 'w × 'a) ⇒ bool where
edges-less-eq a b ≡ fst (snd a) ≤ fst (snd b)

definition maximally-connected :: ('v, 'w) graph ⇒ ('v, 'w) graph ⇒ bool where
maximally-connected H G ≡ ∀ v ∈ nodes G. ∀ v' ∈ nodes G.
(nodes-connected G v v') −→ (nodes-connected H v v')

definition spanning-forest :: ('v, 'w) graph ⇒ ('v, 'w) graph ⇒ bool where
spanning-forest F G ≡ forest F ∧ maximally-connected F G ∧ subgraph F G

definition optimal-forest :: ('v, 'w::weight) graph ⇒ ('v, 'w) graph ⇒ bool where
optimal-forest F G ≡∀ F'::('v, 'w) graph.
spanning-forest F' G −→ edge-weight F ≤ edge-weight F'

definition minimum-spanning-forest :: ('v, 'w::weight) graph ⇒ ('v, 'w) graph ⇒ bool where
minimum-spanning-forest F G ≡ spanning-forest F G ∧ optimal-forest F G

definition spanning-tree :: ('v, 'w) graph ⇒ ('v, 'w) graph ⇒ bool where
spanning-tree F G ≡ tree F ∧ subgraph F G

definition optimal-tree :: ('v, 'w::weight) graph ⇒ ('v, 'w) graph ⇒ bool where
optimal-tree F G ≡∀ F'::('v, 'w) graph.
spanning-tree F' G −→ edge-weight F ≤ edge-weight F'

definition minimum-spanning-tree :: ('v, 'w::weight) graph ⇒ ('v, 'w) graph ⇒ bool where
minimum-spanning-tree F G ≡ spanning-tree F G ∧ optimal-tree F G

7.2 Helping lemmas

lemma nodes-delete-edge[simp]:
nodes (delete-edge v e v' G) = nodes G
by (simp add: delete-edge-def)

lemma edges-delete-edge[simp]:
edges (delete-edge v e v' G) = edges G - {(v,e,v')}
by (simp add: delete-edge-def)

lemma subgraph-node:
assumes subgraph H G
shows v ∈ nodes G ⟷ v ∈ nodes H
using assms
unfolding subgraph-def
by simp

lemma delete-add-edge:
assumes a ∈ nodes H
assumes c ∈ nodes H
assumes (a, w, c) /∈ edges H
shows delete-edge a w c (add-edge a w c H) = H
using assms unfolding delete-edge-def add-edge-def
by (simp add: insert-absorb)

lemma swap-delete-add-edge:
assumes (a, b, c) ≠ (x, y, z)
shows delete-edge a b c (add-edge x y z H) = add-edge x y z (delete-edge a b c H)
using assms unfolding delete-edge-def add-edge-def
by auto

lemma swap-delete-edges: delete-edge a b c (delete-edge x y z H) = delete-edge x y z (delete-edge a b c H)
unfolding delete-edge-def
by auto

cancel context valid-graph
begin

lemma valid-subgraph:
assumes subgraph H G
shows valid-graph H
using assms E-valid unfolding subgraph-def valid-graph-def
by blast

lemma is-path-undir-simps[simp, intro!]:
is-path-undir G v [] v ⟷ v ∈ V
is-path-undir G v [(v,w,v')] v' ⟷ (v,w,v') ∈ E ∨ (v',w,v) ∈ E
by (auto dest: E-validD)

lemma is-path-undir-memb[simp]:
is-path-undir G v p v' ⟷ v ∈ V ∧ v' ∈ V
apply (induct p arbitrary: v)
apply (auto dest: E-validD)
done

lemma is-path-undir-memb-edges:
assumes is-path-undir G v p v'
shows ∀(a,w,b) ∈ set p. (a,w,b) ∈ E ∨ (b,w,a) ∈ E
using assms
by (induct p arbitrary: v) fastforce+

lemma is-path-undir-split:
is-path-undir G v (p1@[p2]) v' \iff (\exists u. is-path-undir G v p1 u \land is-path-undir G u p2 v')
by (induct p1 arbitrary: v) auto

lemma is-path-undir-split'[simp]:
is-path-undir G v (p1 @(u,w,u')#p2) v'
\iff is-path-undir G v p1 u \land ((u,w,u')\in E \lor (u',w,u)\in E) \land is-path-undir G u' p2 v'
by (auto simp add: is-path-undir-split)

lemma is-path-undir-sym:
assumes is-path-undir G v p v'
shows is-path-undir G v' (rev (map (\lambda (u, w, u'). (u', w, u)) p)) v
using assms
by (induct p arbitrary: v) (auto simp: E-validD)

lemma is-path-undir-subgraph:
assumes is-path-undir H x p y
assumes subgraph H G
shows is-path-undir G x p y
using assms is-path-undir.simps
unfolding subgraph-def
by (induction p arbitrary: x y) auto

lemma no-path-in-empty-graph:
assumes E = {}
assumes p \neq []
shows \neg is-path-undir G v p v
using assms
by (cases p) auto

lemma is-path-undir-split-distinct:
assumes is-path-undir G v p v'
assumes (a, w, b) \in set p \lor (b, w, a) \in set p
shows (\exists p'' u a'. is-path-undir G v p' u \land is-path-undir G u' p'' v' \land
length p' < length p \land length p'' < length p \land
(u \in \{a, b\} \land u' \in \{a, b\} \land
(a, w, b) \notin set p' \land (b, w, a) \notin set p' \land
(a, w, b) \notin set p'' \land (b, w, a) \notin set p''))
using assms
proof (induction n == length p arbitrary: p v v' rule: nat-less-induct)
case 1
then obtain u u' where (u, w, u') \in set p and u : u \in \{a, b\} \land u' \in \{a, b\}
by blast
with split-list obtain p' p''
where p: p = p' @ (u, w, u') \# p''

49
by fast
then have len-p': length p' < length p and len-p'': length p'' < length p
by auto
from 1 p have p': is-path-undir G v p' u and p'': is-path-undir G u' p'' v'
by auto
from 1 len-p' p' have (a, w, b) ∈ set p' ∨ (b, w, a) ∈ set p' → (∃ p'' u2. is-path-undir G v p'' u2 ∧ length p'' < length p' ∧ u2 ∈ {a, b} ∧ (a, w, b) /∈ set p'' ∧ (b, w, a) /∈ set p'')
by metis
with len-p' p' u have p': ∃ p' u. is-path-undir G v p' u ∧ length p' < length p ∧ u ∈ {a, b} ∧ (a, w, b) /∈ set p' ∧ (b, w, a) /∈ set p'
by fastforce
from 1 len-p'' p'' have (a, w, b) ∈ set p'' ∨ (b, w, a) ∈ set p'' → (∃ p'' u2. is-path-undir G u2 p'' v' ∧ length p'' < length p'' ∧ u2 ∈ {a, b} ∧ (a, w, b) /∈ set p'' ∧ (b, w, a) /∈ set p'')
by metis
with len-p'' p'' u have p'': ∃ p'' u'. is-path-undir G u' p'' v' ∧ length p'' < length p'' ∧ u' ∈ {a, b} ∧ (a, w, b) /∈ set p'' ∧ (b, w, a) /∈ set p''
by fastforce
with p' show ?case by auto
qed

lemma add-edge-is-path:
assumes is-path-undir G x p y
shows is-path-undir (add-edge a b c G) x p y
proof –
from E-valid have valid-graph (add-edge a b c G)
unfolding valid-graph-def add-edge-def
by auto
with assms is-path-undir.simps[of add-edge a b c G]
show is-path-undir (add-edge a b c G) x p y
by (induction p arbitrary: x y) auto
qed

lemma add-edge-was-path:
assumes is-path-undir (add-edge a b c G) x p y
assumes (a, b, c) /∈ set p
assumes (c, b, a) /∈ set p
assumes a ∈ V
assumes c ∈ V
shows is-path-undir G x p y
proof –
from E-valid have valid-graph (add-edge a b c G)
unfolding valid-graph-def add-edge-def
by auto
with assms is-path-undir.simps[of add-edge a b c G]
show is-path-undir G x p y
  by (induction p arbitrary; x y) auto
qed

lemma delete-edge-is-path:
  assumes is-path-undir G x p y
  assumes (a, b, c) ∉ set p
  assumes (c, b, a) ∉ set p
  shows is-path-undir (delete-edge a b c G) x p y
proof –
  from E-valid have valid-graph (delete-edge a b c G)
    unfolding valid-graph-def delete-edge-def
    by auto
  with assms is-path-undir.simps[of delete-edge a b c G]
  show ?thesis
    by (induction p arbitrary; x y) auto
qed

lemma delete-node-is-path:
  assumes is-path-undir G x p y
  assumes x ≠ v
  assumes v /∈ fst'set p ∪ snd'snd'set p
  shows is-path-undir (delete-node v G) x p y
  using assms
    unfolding delete-node-def
    by (induction p arbitrary; x y) auto

lemma delete-edge-was-path:
  assumes is-path-undir (delete-edge a b c G) x p y
  shows is-path-undir G x p y
  using assms
    by (induction p arbitrary; x y) auto

lemma subset-was-path:
  assumes is-path-undir H x p y
  assumes edges H ⊆ E
  assumes nodes H ⊆ V
  shows is-path-undir G x p y
  using assms
    by (induction p arbitrary; x y) auto

lemma delete-node-was-path:
  assumes is-path-undir (delete-node v G) x p y
  shows is-path-undir G x p y
  using assms
    unfolding delete-node-def
by \((\text{induction } p \text{ arbitrary: } x \ y) \text{ auto}\)

**Lemma add-edge-preserve-subgraph:**

- **Assumes** subgraph \(H \ G\)
- **Assumes** \((a, w, b) \in E\)
- **Shows** subgraph \((\text{add-edge } a \ w \ b \ H) \ G\)

**Proof**
- from assms E-validD have \(a \in \text{nodes } H \land b \in \text{nodes } H\)
- unfolding subgraph-def by simp
- with assms show \(\text{thesis}\)
- unfolding subgraph-def
- by auto

qed

**Lemma delete-edge-preserve-subgraph:**

- **Assumes** subgraph \(H \ G\)
- **Shows** subgraph \((\text{delete-edge } a \ w \ b \ H) \ G\)
- using assms
- unfolding subgraph-def
- by auto

**Lemma add-delete-edge:**

- **Assumes** \((a, w, c) \in E\)
- **Shows** add-edge \(a \ w \ c\) \((\text{delete-edge } a \ w \ c \ G) = G\)
- using assms E-validD unfolding delete-edge-def add-edge-def
- by \((\text{simp add: } \text{insert-absorb})\)

**Lemma swap-add-edge-in-path:**

- **Assumes** is-path-undir \((\text{add-edge } a \ w \ b \ G) \ v \ p \ v'\)
- **Assumes** \((a,w',a') \in E \lor (a',w',a) \in E\)
- **Shows** \(\exists p. \text{is-path-undir} \ (\text{add-edge } a' \ w'' b \ G) \ v \ p \ v'\)
- using assms(1)

**Proof** (induction \(p\) arbitrary: \(v\))
- case Nil
- with assms(2) E-validD
- have is-path-undir \((\text{add-edge } a' \ w'' b \ G) \ v \ v'\)
- by auto
- then show \(\text{?case}\)
- by blast

next
- case \((\text{Cons } e \ p')\)
- then obtain \(v2 \ x \ e\text{-}w\text{ where } e = (v2, e\text{-}w, x)\)
- using prod-cases3 by blast
- with Cons(2)
- have \(e: e = (v, e\text{-}w, x)\text{ and}\)
  - edge-e: \((v, e\text{-}w, x) \in \text{edges} \ (\text{add-edge } a \ w \ b \ G)\)
  - \(\lor (x, e\text{-}w, v) \in \text{edges} \ (\text{add-edge } a \ w \ b \ G)\) and
  - \(p': \text{is-path-undir} \ (\text{add-edge } a \ w \ b \ G) \ x \ p' \ v'\)
- by auto

52
have \( \exists p. \text{is-path-undir} (\text{add-edge} a' w'' b G) v p x \)

proof (cases \( e = (a, w, b) \) \( \lor e = (b, w, a) \))

  case True
  from True e assms (2) E-validD
  have is-path-undir (add-edge a' w'' b G) v [(a,w',a'), (a',w'',b)] x
    \( \lor \) is-path-undir (add-edge a' w'' b G) v [(b,w'',a'), (a',w',a)] x
    by auto
  then show ?thesis
    by blast

next
  case False
  with edge-e e
  have is-path-undir (add-edge a' w'' b G) v [e] x
  by (auto simp: E-validD)
  then show ?thesis
  by blast

qed

with \( p' \) Cons.IH

and valid-graph.is-path-undir-split[OF add-edge-valid[OF valid-graph.intro[OF E-valid]]]

show ?case
by blast

qed

lemma induce-maximally-connected:

assumes subgraph H G
assumes \( \forall (a,w,b) \in E. \text{nodes-connected} H a b \)
shows maximally-connected H G

proof -
from valid-subgraph[OF ⟨subgraph H G⟩]
have valid-H: valid-graph H ,

have (nodes-connected G v v') \( \longrightarrow \) (nodes-connected H v v') (is ?lhs \( \longrightarrow \) ?rhs)
  if \( v \in V \) and \( v' \in V \) for v v'

proof
  assume ?lhs
  then obtain p where is-path-undir G v p v'
  by blast
  then show ?rhs
  proof (induction p arbitrary: v v')
    case Nil
    with subgraph-node[OF assms(1)] show ?case
    by (metis is-path-undir.simps(1))

next
  case (Cons e p)
  from prod-cases3 obtain a b where awb: e = (a, w, b) ,
  with assms Cons.prems valid-graph.is-path-undir-sym[OF valid-H, of b - a]
  obtain p' where p': is-path-undir H a p' b
  by fastforce
  from assms awb Cons.prems Cons.IH[of b v']

53
obtain \( p'' \) where is-path-undir \( H b p'' v' \)

unfolding subgraph-def by auto

with Cons.prems awb assms p' valid-graph.is-path-undir-split[OF valid-H]

have is-path-undir \( H v (p' @ p'') v' \)

by auto

then show \(?case ..\)

qed

qed with assms show \(?thesis\)

unfolding maximally-connected-def

by auto

qed

lemma add-edge-maximally-connected:

assumes maximally-connected \( H G \)

assumes subgraph \( H G \)

assumes \((a, w, b) \in E\)

shows maximally-connected \((add-edge a w b H) G\)

proof –

have \((\text{nodes-connected \( G v v' \)} \rightarrow (\text{nodes-connected \((add-edge a w b H) v v'\)}) \)

(is \(?lhs \rightarrow ?rhs\)) if \(vv': v \in V v' \in V\) for \(v v'\)

proof

assume \(?lhs\)

with \(\langle \text{maximally-connected \( H G\)} \rangle vv'\) obtain \( p \) where is-path-undir \( H v p v' \)

unfolding maximally-connected-def

by auto

with valid-graph.add-edge-is-path[OF valid-subgraph[OF \(\langle \text{subgraph \( H G\)} \rangle\) this]

show \(?rhs\)

by auto

qed

then show \(?thesis\)

unfolding maximally-connected-def

by auto

qed

lemma delete-edge-maximally-connected:

assumes maximally-connected \( H G \)

assumes subgraph \( H G \)

assumes \(pab: \text{is-path-undir \((delete-edge a w b H) a pab b\)}\)

shows maximally-connected \((delete-edge a w b H) G\)

proof –

from valid-subgraph[OF \(\langle \text{subgraph \( H G\)} \rangle\)]

have valid-H: valid-graph \( H\).

have \((\text{nodes-connected \( G v v' \)} \rightarrow (\text{nodes-connected \((delete-edge a w b H) v v'\)}) \)

(is \(?lhs \rightarrow ?rhs\)) if \(vv': v \in V v' \in V\) for \(v v'\)

proof

assume \(?lhs\)

with \(\langle \text{maximally-connected \( H G\)} \rangle vv'\) obtain \( p \) where \(p: \text{is-path-undir \( H v p\)}\)
unfolding \texttt{maximally-connected-def} \\
by \texttt{auto} \\
show \texttt{?rhs} \\
proof (cases \((a, w, b) \in \text{set } p \lor (b, w, a) \in \text{set } p\)) \\
\texttt{case True} \\
with \(p\) \texttt{valid-graph.is-path-undir-split-distinct[OF valid-H p, of a w b]} \obtain \(p'\) \(p''\) \(u\) \(u'\) \\
where \(\text{is-path-undir } H v p' u \land \text{is-path-undir } H u' p'' v'\) and \\
\((u \in \{a, b\} \land u' \in \{a, b\})\) and \\
\((a, w, b) \notin \text{set } p' \land (b, w, a) \notin \text{set } p'\) \land \\
\((a, w, b) \notin \text{set } p'' \land (b, w, a) \notin \text{set } p''\) \\
by \texttt{auto} \\
with \texttt{valid-graph.delete-edge-is-path[OF valid-H]} \obtain \(p'\) \(p''\) \\
where \(p': \text{is-path-undir } (\text{delete-edge } a w b H) v p' u \land \)
\(\text{is-path-undir } (\text{delete-edge } a w b H) u' p'' v'\) \\
by \texttt{blast} \\
note \texttt{dev-H = delete-edge-valid[OF valid-H]} \\
note \texttt{* = valid-graph.is-path-undir-split[OF dev-H, of a w b v]} \\
from \texttt{valid-graph.is-path-undir-sym[OF delete-edge-valid[OF valid-H] pab]} \obtain \(\text{pab}'\) \\
where \(\text{is-path-undir } (\text{delete-edge } a w b H) b \text{pab}' a\) \\
by \texttt{auto} \\
with \texttt{assms u p' valid-graph.is-path-undir-split[OF dev-H, of a w v v' p'']} \\
\([\text{of } p' \text{ pab } b] \ast [\text{of } p' @ \text{pab} p'' v'] \ast [\text{of } p' \text{ pab}' a] \ast [\text{of } p' @ \text{pab}' p'' v']\) \\
show \texttt{?thesis} by \texttt{auto} \\
next \\
\texttt{case False} \\
with \texttt{valid-graph.delete-edge-is-path[OF valid-H p]} \show \texttt{?thesis} \\
by \texttt{auto} \\
qed \\
qed \\
then show \texttt{?thesis} \\
unfolding \texttt{maximally-connected-def} \\
by \texttt{auto} \\
qed \\

lemma \texttt{connected-impl-maximally-connected}: \\
\texttt{assumes connected-graph H} \\
\texttt{assumes subgraph: subgraph H G} \\
\texttt{shows maximally-connected H G} \\
using \texttt{assms} \\
unfolding \texttt{connected-graph-def connected-graph-axioms-def maximally-connected-def subgraph-def} \\
by \texttt{blast} \\

lemma \texttt{add-edge-is-connected}: \\
\texttt{nodes-connected (add-edge a b c G) a c}
nodes-connected (add-edge a b c G) c a

using valid-graph.is-path-undir-simps(2)[OF
  add-edge-valid[OF valid-graph-axioms], of a b c a b c]
valid-graph.is-path-undir-simps(2)[OF
  add-edge-valid[OF valid-graph-axioms], of a b c b a]

by fastforce+

lemma swap-edges:
  assumes nodes-connected (add-edge a w b G) v v'
  assumes a ∈ V
  assumes b ∈ V
  assumes ¬ nodes-connected G v v'
  shows nodes-connected (add-edge v w' v' G) a b

proof −
  from assms(1) obtain p where p: is-path-undir (add-edge a w b G) v p v'
    by auto
  have awb: (a, w, b) ∈ set p ∨ (b, w, a) ∈ set p
  proof (rule ccontr)
    assume ¬ ((a, w, b) ∈ set p ∨ (b, w, a) ∈ set p)
    with add-edge-was-path[OF p - - assms(2,3)] assms(4)
    show False by auto
  qed

  from valid-graph.is-path-undir-split-distinct[OF
    add-edge-valid[OF valid-graph-axioms] p awb]
  obtain p'' a u' where
    is-path-undir (add-edge a w b G) v p' u ∧
    is-path-undir (add-edge a w b G) u' p'' v' and
    u: u ∈ {a, b} ∧ u' ∈ {a, b} and
    (a, w, b) /∈ set p ∧ (b, w, a) /∈ set p ∧
    (a, w, b) /∈ set p'' ∧ (b, w, a) /∈ set p''
    by auto
  with assms(2,3) add-edge-was-path
  have paths: is-path-undir G v p' u ∧
    is-path-undir G u' p'' v'
    by blast
  with is-path-undir-split[of v p' p'' v'] assms(4)
  have u ≠ u'
    by blast
  from paths assms add-edge-is-path
  have paths': is-path-undir (add-edge v w' v' G) v p' u ∧
    is-path-undir (add-edge v w' v' G) u' p'' v'
    by blast
  note * = add-edge-valid[OF valid-graph-axioms]
  from add-edge-is-connected obtain p''' where
    is-path-undir (add-edge v w' v' G) v' p''' v
    by blast
  with paths' valid-graph.is-path-undir-split[OF *, of v w' v' u' p'' p''' v]
  have is-path-undir (add-edge v w' v' G) u' (p''@[p''']) v
by auto
with paths\textsuperscript{′} valid-graph.is-path-undir-split[OF \star, of v w’ v’ u’ p''@p''@p’ u]
have is-path-undir (add-edge v w’ v’ G) u’ (p''@p''@p’ u)
  by auto
with u (u \neq u’) valid-graph.is-path-undir-sym[OF \star this]
show \?thesis
  by auto
qed

lemma subgraph-impl-connected:
  assumes connected-graph H
  assumes subgraph: subgraph H G
  shows connected-graph G
  using assms is-path-undir-subgraph[OF - subgraph] valid-graph-axioms
  unfolding connected-graph-def connected-graph-axioms-def maximally-connected-def
  subgraph-def
  by blast

lemma add-node-connected:
  assumes \(\forall a \in V - \{v\}. \forall b \in V - \{v\}.\) nodes-connected G a b
  assumes (v, w, v’) \(\in E \lor (v’, w, v) \in E\)
  assumes v \(\neq v’\)
  shows \(\forall a \in V, \forall b \in V.\) nodes-connected G a b
proof =
  have nodes-connected G a b if a: a \(\in V\) and b: b \(\in V\) for a b
  proof (cases a = v)
    case True
    show \?thesis
    proof (cases b = v)
      case True
      with (a = v) a is-path-undir-simps(1) show \?thesis
      by blast
    next
    case False
    from assms(2) have v’ \(\in V\)
      by (auto simp: E-validD)
    with b assms(1) (b \(\neq v\) \(\lor v \neq v’\)) have nodes-connected G v’ b
      by blast
    with assms(2) \(a = v\) is-path-undir-simps(2)[of G v v w v’ - b]
    show \?thesis
      by blast
  qed
next
  case False
  show \?thesis
  proof (cases b = v)
    case True
    from assms(2) have v’ \(\in V\)
      by (auto simp: E-validD)
  qed

57
with a assms(1) \(a \neq v\) \(v \neq v'\) have nodes-connected \(G a v'\)
  by blast
with assms(2) \(b = v\) is-path-undir.simps(2)[of \(G v v v' a\)]
  is-path-undir-sym
  show \(?thesis\)
  by blast
next
case False
with \(a \neq v\) assms(1) a b show ?thesis
  by simp
qed
qed
then show ?thesis by simp
qed
qed
end

context connected-graph
begin
lemma maximally-connected-impl-connected:
  assumes maximally-connected \(H G\)
  assumes subgraph: subgraph \(H G\)
  shows connected-graph \(H\)
  using assms connected-graph-axioms valid-subgraph[OF subgraph]
  unfolding connected-graph-def connected-graph-axioms-def
  maximally-connected-def subgraph-def
  by auto
end

context forest
begin
lemmas delete-edge-valid' = delete-edge-valid[OF valid-graph-axioms]

lemma delete-edge-from-path:
  assumes nodes-connected \(G a b\)
  assumes subgraph \(H G\)
  assumes \(\neg\) nodes-connected \(H a b\)
  shows \(\exists (x, w, y) \in E - \text{edges } H. \ (\neg\text{nodes-connected } (\text{delete-edge } x w y G) a b) \land\)
  \(\text{nodes-connected } (\text{add-edge } a w' b \ (\text{delete-edge } x w y G)) x y\)
  proof
    from assms(1) obtain p where is-path-undir \(G a p b\)
      by auto
    from this assms(3) show ?thesis
    proof (induction \(n == \text{length } p\) arbitrary: \(p a b\) rule: nat-less-induct)
      case 1
      from valid-subgraph[OF assms(2)] have valid-H: valid-graph \(H\).
      show ?case
      proof (cases \(p\) )

58
case Nil
with 1(2) have \( a = b \)
  by simp
with 1(2) assms(2) have is-path-undir \( H \ a \emptyset b \)
  unfolding subgraph-def
  by auto
with 1(3) show \( \text{?thesis} \)
  by blast
next
  case \( (\text{Cons} \ e \ p') \)
  obtain \( a_2 \ a' w \) where \( e = (a_2, w, a') \)
    using prod-cases3 by blast
  with 1(2) Cons have \( c: c = (a, w, a') \)
    by simp
  with 1(2) Cons obtain \( e_1 e_2 \) where \( e_12: e = (e_1, w, e_2) \lor e = (e_2, w, e_1) \)
  and
    edge-e12: \( (e_1, w, e_2) \in E \)
    by auto
  from 1(2) Cons e have is-path-undir \( G \ a' p' b \)
    by simp
  with is-path-undir-split-distinct[OF this, of \( a \ w a' \)] Cons
  obtain \( p'-\text{dst} \ u' \) where \( p'-\text{dst} : \) is-path-undir \( G \ u' p'-\text{dst} b \land u' \in \{ a, a' \} \)
  and
    e-not-in-p': \( (a, w, a') \notin \text{set} p'-\text{dst} \land (a', w, a) \notin \text{set} p'-\text{dst} \)
    and
    len-p': length \( p'-\text{dst} < \) length \( p \)
    by fastforce
  show \( \text{?thesis} \)
proof (cases \( u' = a' \))
  case False
  with 1 len-p' p'-\text{dst} show \( \text{?thesis} \)
    by auto
next
  case True
  with p'-\text{dst} have path-p': is-path-undir \( G \ a' p'-\text{dst} b \)
    by auto
  show \( \text{?thesis} \)
proof (cases \( (e1, w, e2) \in \text{edges} \ H \))
  case True
  have \( \neg \) nodes-connected \( H \ a' b \)
    proof
      assume nodes-connected \( H \ a' b \)
      then obtain \( p-H \) where is-path-undir \( H \ a' p-H b \)
        by auto
      with True e12 e have is-path-undir \( H \ a \ (e\#p-H) b \)
        by auto
      with 1(3) show False
        by simp
    qed
  with path-p' 1(1) len-p' obtain \( x z y \) where \( xy: (x, z, y) \in E \setminus \text{edges} \)

\( H \text{ and} \)

\[ IH1: (\neg \text{nodes-connected } (\text{delete-edge } x z y G) \ a' \ b) \text{ and} \]

\[ IH2: (\text{nodes-connected } (\text{add-edge } a' \ w' \ b \ (\text{delete-edge } x z y G)) \ x y) \]

by blast

with True have \( xy \neq e \): \((x, z, y) \neq (e_1, w, e_2)\)

by auto

have \( \text{thm1}: \neg \text{nodes-connected } (\text{delete-edge } x z y G) \ a \ b \)

proof

assume \( \text{nodes-connected } (\text{delete-edge } x z y G) \ a \ b \)

then obtain \( p-e \) where \( \text{is-path-undir } (\text{delete-edge } x z y G) \ a \ p-e \ b \)

by auto

with \( \text{edge-e12} e_{12} \ e \ x y \neq e \)

have \( \text{is-path-undir } (\text{delete-edge } x z y G) \ a' ((a', w, a)\#p-e) \ b \)

by auto

with \( IH1 \) show False

by blast

qed

from \( IH2 \) obtain \( p-xy \)

where \( \text{is-path-undir } (\text{add-edge } a' \ w' \ b \ (\text{delete-edge } x z y G)) \ x \ p-xy \ y \)

by auto

from valid-graph.swap-add-edge-in-path[OF delete-edge-valid this, of \( w a w' \)] edge-e12

\( e_{12} \ e \ \text{edges-delete-edge}[\text{of } x z y G] \ xy \neq e \)

have \( \text{thm2}: \text{nodes-connected } (\text{add-edge } a w' b \ (\text{delete-edge } x z y G)) \ x y \)

by blast

with \( \text{thm1} \) show \( \neg \text{thesis} \)

using \( xy \) by auto

next

case False

have \( \text{thm1}: \neg \text{nodes-connected } (\text{delete-edge } e_1 w e_2 G) \ a \ b \)

proof

assume \( \text{nodes-connected } (\text{delete-edge } e_1 w e_2 G) \ a \ b \)

then obtain \( p-e \) where \( p-e: \text{is-path-undir } (\text{delete-edge } e_1 w e_2 G) \ a \)

\( p-e b \)

by auto

from delete-edge-is-path[OF path-p', of \( e_1 w e_2 \)] e-not-in-p' e_{12} e

have \( \text{is-path-undir } (\text{delete-edge } e_1 w e_2 G) \ a' \ p'\text{-dst} b \)

by auto

with valid-graph.is-path-undir-sym[OF delete-edge-valid this]

obtain \( p\text{-rev} \) where \( \text{is-path-undir } (\text{delete-edge } e_1 w e_2 G) \ b \ p\text{-rev} a' \)

by auto

with \( p-e \) valid-graph.is-path-undir-split[OF delete-edge-valid]

have \( \text{is-path-undir } (\text{delete-edge } e_1 w e_2 G) \ a \ (p-e \& p\text{-rev}) \ a' \)

by auto

with cycle-free edge-e12 e_{12} e

and valid-graph.is-path-undir-sym[OF delete-edge-valid this]

show False

unfolding valid-graph-def

by auto

60
qed

note ** = delete-edge-is-path[OF path-p', of e1 w e2]
from valid-graph.is-path-undir-split[OF add-edge-valid[OF delete-edge-valid']]
valid-graph.add-edge-is-path[OF delete-edge-valid' ** , of a w' b]
valid-graph.is-path-undir-simps(2)[OF add-edge-valid[OF delete-edge-valid'],
of a w' b e1 w e2 b w' a]

\[ e \in \text{not-in-p'} \]
e12 e
have is-path-undir (add-edge a w' b (delete-edge e1 w e2 G)) a'
(p'-dst @(b, w', a)) a
by auto
with valid-graph.is-path-undir-sym[OF add-edge-valid[OF delete-edge-valid']]
this]
e12 e
have nodes-connected (add-edge a w' b (delete-edge e1 w e2 G)) e1 e2
by blast
with thm1 show ?thesis
using False edge-e12 by auto
qed
qed
qed
qed
qed

lemma forest-add-edge:
assumes a ∈ V
assumes b ∈ V
assumes \( \neg \) nodes-connected G a b
shows forest (add-edge a w b G)
proof -
from asms(3) have \( \neg \) is-path-undir G a [(a, w, b)] b
by blast
with asms(2) have awb: (a, w, b) \notin E \land (b, w, a) \notin E
by auto
have \( \neg \) nodes-connected (delete-edge v w' v' (add-edge a w b G)) v v'
if e: (v, w', v') ∈ edges (add-edge a w b G) for v w' v'
proof (cases (v, w', v') = (a, w, b))
case True
with asms awb delete-add-edge[of a G b w]
show ?thesis by simp
next
case False
with e have e': (v, w', v') ∈ edges G
by auto
show ?thesis
proof
assume asm: nodes-connected (delete-edge v w' v' (add-edge a w b G)) v v'
with swap-delete-add-edge[OF False, OF G]
valid-graph.swap-edges[OF delete-edge-valid', of a w b v w' v' v v' w]
add-delete-edge[OF e'] cycle-free asms(1,2) e'

have nodes-connected $G$ $a$ $b$
  by force
with assms show False
  by simp
qed
qed
with cycle-free add-edge-valid[OF valid-graph-axioms] show \( ?\)thesis
 unfolding forest-def forest-axioms-def by auto
qed

lemma forest-subsets:
  assumes valid-graph $H$
  assumes edges $H$ $\subseteq$ $E$
  assumes nodes $H$ $\subseteq$ $V$
  shows forest $H$
proof
  have \( \neg \) nodes-connected (delete-edge $a$ $w$ $b$ $H$) $a$ $b$
    if $e$ $:\ (a,\ w,\ b) \in$ edges $H$ for $a$ $w$ $b$
  proof
    assume asm: nodes-connected (delete-edge $a$ $w$ $b$ $H$) $a$ $b$
    from \( \langle \) edges $H$ $\subseteq$ $E$\) have edges: edges (delete-edge $a$ $w$ $b$ $H$) $\subseteq$ edges (delete-edge $a$ $w$ $b$ $G$)
      by auto
    from \( \langle \) nodes $H$ $\subseteq$ $V$\) have nodes: nodes (delete-edge $a$ $w$ $b$ $H$) $\subseteq$ nodes (delete-edge $a$ $w$ $b$ $G$)
      by auto
    from asm valid-graph.subset-was-path[OF delete-edge-valid' - edges nodes] have nodes-connected (delete-edge $a$ $w$ $b$ $G$) $a$ $b$
      by auto
    with cycle-free $e$ \( \langle \) edges $H$ $\subseteq$ $E$\) show False
      by blast
  qed
with assms(1) show \( ?\)thesis
 unfolding forest-def forest-axioms-def by auto
qed

lemma subgraph-forest:
  assumes subgraph $H$ $G$
  shows forest $H$
  using assms forest-subsets valid-subgraph
 unfolding subgraph-def by simp

lemma forest-delete-edge: forest (delete-edge $a$ $w$ $c$ $G$)
  using forest-subsets[OF delete-edge-valid']
 unfolding delete-edge-def by auto

62
lemma forest-delete-node: forest (delete-node n G)
    using forest-subsets[OF delete-node-valid[OF valid-graph-axioms]]
    unfolding delete-node-def
    by auto
end

context finite-graph
begin

lemma finite-subgraphs: finite {T. subgraph T G}
    proof –
    from finite-E have finite {E’. E’ ⊆ E}
        by simp
    then have finite {[(nodes = V, edges = E’)| E’. E’ ⊆ E} by simp
    also have {[(nodes = V, edges = E’)| E’. E’ ⊆ E} = {T. subgraph T G}
        unfolding subgraph-def
        by (metis (mono-tags, lifting) old.unit.exhaust select-convs(1) select-convs(2)
            surjective)
    finally show ?thesis .
    qed
end

lemma minimum-spanning-forest-impl-tree:
    assumes minimum-spanning-forest F G
    assumes valid-G: valid-graph G
    assumes connected-graph F
    shows minimum-spanning-tree F G
    using assms valid-graph.connected-impl-maximally-connected[OF valid-G]
    unfolding minimum-spanning-forest-def minimum-spanning-tree-def
        spanning-forest-def spanning-tree-def tree-def
        optimal-forest-def optimal-tree-def
    by auto

lemma minimum-spanning-forest-impl-tree2:
    assumes minimum-spanning-forest F G
    assumes connected-G: connected-graph G
    shows minimum-spanning-tree F G
    using assms connected-graph.maximally-connected-impl-connected[OF connected-G]
    unfolding minimum-spanning-forest-impl-tree connected-graph.axioms(1)[OF connected-G]
    by auto
end
7.3 Auxiliary lemmas for graphs

theory Graph-Definition-Aux
imports Graph-Definition SeprefUF
begin

context valid-graph
begin

lemma nodes-connected-sym: nodes-connected G a b = nodes-connected G b a
using is-path-undir-sym by auto

lemma Domain-nodes-connected: Domain \{ (x, y) | x y. nodes-connected G x y \} = V
apply auto subgoal for x apply(\text{rule exI} [\text{where } x=x]) apply(\text{rule exI} [\text{where } x=[]]) by auto
done

lemma Range-nodes-connected: Range \{ (x, y) | x y. nodes-connected G x y \} = V
apply auto subgoal for x apply(\text{rule exI} [\text{where } x=x]) apply(\text{rule exI} [\text{where } x=[]]) by auto
done

— adaptation of a proof by Julian Biendarra

lemma nodes-connected-insert-per-union:
(nodes-connected (add-edge a w b H) x y) \iff (x,y) \in \text{per-union} \{ (x,y) | x y. nodes-connected H x y \}
if subgraph H G and PER: part-equiv \{ (x,y) | x y. nodes-connected H x y \}
and V: a \in V b \in V for x y
proof
let \(uf = \{ (x,y) | x y. nodes-connected H x y \}\)
from valid-subgraph[OF (subgraph H G)]
have valid-H: valid-graph H .
from (subgraph H G)
have nodes-H: nodes H = V
  unfolding subgraph-def ..
with \(\{a \in V \} \& \{ b \in V \}\)
have nodes-add-H: (add-edge a w b H) = nodes H
by auto
have Domain \(uf = nodes H\) using valid-graph.Domin-nodes-connected[OF valid-H] .
show \(?thesis
proof
assume nodes-connected (add-edge a w b H) x y
then obtain p where p: is-path-undir (add-edge a w b H) x p y
by blast
from \(\{a \in V \} \& \{ b \in V \}\) Domain \{ (x,y) | x y. nodes-connected H x y \} = nodes H
 nodes-H
have [simp]: a \in Domain (per-union \(uf a b\) b \in Domain (per-union \(uf a b\)
by auto
from PER have PER': part-equiv (per-union \(uf a b\)

64
by (auto simp: union-part-equivp)
show \( (x,y) \in \text{per-union } \text{uf } a \ b \)
proof (cases \( (a, w, b) \in \text{set } p \lor (b, w, a) \in \text{set } p \))
case True
from valid-graph.is-path-undir-split-distinct[OF add-edge-valid[OF valid-H] p]

obtain \( p', p'' \) \( u \ u' \) where
  is-path-undir \( \text{add-edge } a \ w \ b \ H \) \( x \ p' u \wedge \)
  is-path-undir \( \text{add-edge } a \ w \ b \ H \) \( u' \ p'' y \) and
  \( u \in \{a,b\} \land u' \in \{a,b\} \) and
  \( (a, w, b) \notin \text{set } p' \land (b, w, a) \notin \text{set } p' \land \)
  \( (a, w, b) \notin \text{set } p'' \land (b, w, a) \notin \text{set } p'' \)
  by auto
with \( \langle a \in V \rangle \langle b \in V \rangle \langle \text{Domain } \text{uf } = \text{nodes } H \rangle \langle \text{subgraph } H G \rangle \)
valid-graph.add-edge-was-path[OF valid-H]

have is-path-undir \( H x \ p' u \wedge \) is-path-undir \( H u' p'' y \)

unfolding subgraph-def by auto
with \( \langle a \in V \rangle \langle b \in V \rangle \langle \text{Domain } \text{uf } = \text{nodes } H \rangle \langle \text{nodes } H \rangle \)
have \( \text{comps } : \langle x, u \rangle \in \text{uf } \land \langle u', y \rangle \in \text{uf } \) by auto
from comp show \( (x,u) \in \text{per-union } \text{uf } a \ b \) apply(intro per-union-impl)
by auto
also from \( u \langle a \in V \rangle \langle b \in V \rangle \langle \text{Domain } \text{uf } = \text{nodes } H \rangle \langle \text{nodes } H \rangle \)
part-equiv-refl[OF PER'[ \( a \in \text{Domain } \langle \text{per-union } \text{uf } a \ b \rangle \)]]
part-equiv-refl[OF PER'[ \( b \in \text{Domain } \langle \text{per-union } \text{uf } a \ b \rangle \)]] part-equiv-sym[OF PER'

PER'

per-union-related[OF PER]

have \( (u,u') \in \text{per-union } \text{uf } a \ b \)
by auto
also (part-equiv-trans[OF PER']) from comp
have \( (u',y) \in \text{per-union } \text{uf } a \ b \) apply(intro per-union-impl)
by auto
finally (part-equiv-trans[OF PER']) show ?thesis by simp
next
case False
with \( \langle a \in V \rangle \langle b \in V \rangle \langle \text{nodes } H \rangle \langle \text{valid-graph.add-edge-was-path } \langle \text{OF valid-H } p(1) \rangle \rangle \)
have is-path-undir \( H x \ p \ y \)
by auto
with \( \langle x,y \rangle \in \text{uf } \) by auto
from per-union-impl[OF this] show ?thesis .
qed
next
assume asm: \( (x, y) \in \text{per-union } \text{uf } a \ b \)
show \( \text{nodes-connected } \langle \text{add-edge } a \ w \ b \ H \rangle x y \)
proof (cases \( (x, y) \in \text{uf } \))
case True
with \( \langle \text{nodes-add-H } \rangle \langle \text{nodes-connected } H x \ y \rangle \)
by auto
with \( \langle \text{valid-graph.add-edge-is-path } \langle \text{OF valid-H } \rangle \rangle \)
show ?thesis by blast
next

65
case False
with asm part-equiv-sym[OF PER]
have \((x,a) \in ?uf \land (b,y) \in ?uf\) ∨
  \((x,b) \in ?uf \land (a,y) \in ?uf\)
  unfolding per-union-def by auto
with \(\langle a \in V \rangle \langle b \in V \rangle\) nodes-H nodes-add-H obtain p q p' q'
where is-path-undir H x p a ∧ is-path-undir H b q y ∨
  is-path-undir H x p' b ∧ is-path-undir H a q' y
  by fastforce
with valid-graph.add-edge-is-path[OF valid-H]
have is-path-undir (add-edge a w b H) x p a ∧
  is-path-undir (add-edge a w b H) b q y ∨
  is-path-undir (add-edge a w b H) x p' b ∧
  is-path-undir (add-edge a w b H) a q' y
  by blast
with valid-graph.is-path-undir-split[OF add-edge-valid][OF valid-H]
have is-path-undir (add-edge a w b H) x (p @ (a, w, b)) ≠ q y ∨
  is-path-undir (add-edge a w b H) x (p' @ (b, w, a)) ≠ q' y
  by auto
with valid-graph.is-path-undir-sym[OF add-edge-valid][OF valid-H]
show ?thesis by blast
qed

lemma is-path-undir-append: is-path-undir G v p1 u =⇒ is-path-undir G u p2 w
  =⇒ is-path-undir G v (p1 @ p2) w
using is-path-undir-split by auto

lemma augment-edge:
assumes sq: subgraph G1 G subgraph G2 G and
  p: \((u, v) \in \{(a, b) \mid a b. nodes-connected G1 a b\}\)
  and notinE2: \((u, v) \notin \{(a, b) \mid a b. nodes-connected G2 a b\}\)
shows \(\exists a b e. (a, b) \notin \{(a, b) \mid a b. nodes-connected G2 a b\} \land e \notin edges G2 \land e \in edges G1 \land (case e of \(\text{case } e \text{ of } \text{case } e \text{ of } e a a = ba \land b = a a\))\)
proof –
  from sg have [simp]: nodes G1 = nodes G nodes G2 = nodes G unfolding subgraph-def by auto
  from p obtain p where a: is-path-undir G1 u p v by blast
  from notinE2 have b: \(~(\exists p \text{. is-path-undir G2 } u p v)\) by auto
  from a b show ?thesis
  proof (induct p arbitrary: u)
    case Nil

66
then have \( u = v \in \text{nodes } G_1 \) by auto
then have \( \text{is-path-undir } G_2 \ u \ \| \ v \) by auto
have \((u, v) \in \{(a, b) \mid a \ b. \ \text{nodes-connected } G_2 \ a \ b\}\)
apply auto
apply((rule exI[where \( x=[[\]\)]) by fact
with Nil(2) show \(?case\) by blast
next
case \((\text{Cons } a \ p)\)
from Cons(2) obtain \( w \ x \ u' \) where \( \text{arg: } a=(u,w,u') \) and \( 2: (x=u \land y=u') \)
\( \lor (x=u' \land y=u) \) and \( e': \text{is-path-undir } G_1 \ u' \ p \ v \)
and \( uwE1: \ (x,w,y) \in \text{edges } G_1 \) apply(cases a) by auto
show \(?case\)
proof (cases \((x,w,y)\in\text{edges } G_2 \lor (y,w,x)\in\text{edges } G_2\))
case True
have \( e2': \sim(\exists p. \text{is-path-undir } G_2 \ u' \ p \ v) \)
proof (rule ccontr, clarsimp)
fix \( p2 \)
assume \( \text{is-path-undir } G_2 \ u' \ p2 \ v \)
with True axy 2 have \( \text{is-path-undir } G_2 \ u' \ p \ v \) by auto
with Cons(3) show False by blast
qed
from Cons(1)[OF \( e' \ e2' \)] show \(?thesis\).
next
case False
\{ 
assume \( e2': \sim(\exists p. \text{is-path-undir } G_2 \ u' \ p \ v) \)
from Cons(1)[OF \( e' \ e2' \)] have \(?thesis\).
\} moreover \{ 
assume \( e2': \exists p. \text{is-path-undir } G_2 \ u' \ p \ v \)
then obtain \( p1 \) where \( p1: \text{is-path-undir } G_2 \ u' \ p1 \ v \) by auto
from False axy have \((x, w, y)\in\text{edges } G_2 \) by auto
moreover
have \((u,u') \notin \{(a, b) \mid a \ b. \ \text{nodes-connected } G_2 \ a \ b\}\)
proof (rule ccontr, auto simp add: )
fix \( p2 \)
assume \( \text{is-path-undir } G_2 \ u \ p2 \ u' \)
with \( p1 \) have \( \text{is-path-undir } G_2 \ u \ (p2@p1) \ v \)
using valid-graph.is-path-undir-append[OF valid-subgraph[OF assms(2)]] by auto
then show False using Cons(3) by blast
qed
moreover
note uwE1
ultimately have \(?thesis\)
apply --
apply((rule exI[where \( x=[[\]\)])
apply((rule exI[where \( x=[x]\)])
apply((rule exI[where \( x=(x,w,y)\)])

67
using 2 by fastforce
}
ultimately show thesis by auto
qed
qed

lemma nodes-connected-refl: a ∈ V ⇒ nodes-connected G a a
apply (rule exI [where x=[]]) by auto

lemma assumes sg: subgraph H G
shows connected-VV: \{ (x, y) | x y. nodes-connected H x y \} ⊆ V×V
and connected-refl: refl-on V \{ (x, y) | x y. nodes-connected H x y \}
and connected-trans: trans \{ (x, y) | x y. nodes-connected H x y \}
and connected-sym: sym \{ (x, y) | x y. nodes-connected H x y \}
and connected-equiv: equiv V \{ (x, y) | x y. nodes-connected H x y \}
proof –
have ∗: \( \forall R S. \text{Domain } R \subseteq S \Rightarrow \text{Range } R \subseteq S \subseteq S×S \) by auto
from sg have [simp]: nodes H = V by (auto simp: subgraph-def)
from sg valid-subgraph have v: valid-graph H by auto
from valid-graph. Domain-nodes-connected[OF this] valid-graph.Range-nodes-connected[OF this]
show i: \{ (x, y) | x y. nodes-connected H x y \} ⊆ V×V
apply (intro ∗) by auto

have ii: \( \forall x. x \in V \Rightarrow (x, x) \in \{ (x, y) | x y. nodes-connected H x y \} \)
using valid-graph.nodes-connected-refl[OF v] by auto
show refl-on V \{ (x, y) | x y. nodes-connected H x y \}
apply (rule refl-onI) by fact+

from valid-graph. is-path-undir-append[OF v]
show trans \{ (x, y) | x y. nodes-connected H x y \} unfolding trans-def by fast

from valid-graph. nodes-connected-sym[OF v]
show sym \{ (x, y) | x y. nodes-connected H x y \} unfolding sym-def by fast

show equiv V \{ (x, y) | x y. nodes-connected H x y \} apply (rule equivI) by fact+
qed

lemma forest-maximally-connected-incl-max1:
assumes forest H
subgraph H G
shows (\( \forall (a,w,b) \in \text{edges } G - \text{edges } H. \neg (\text{forest } (\text{add-edge } a \ w \ b \ H)) \)) \Rightarrow
maximally-connected H G
proof –
from assms(2) have V[simp]: nodes H = nodes G unfolding subgraph-def by
auto

assume pff: (\forall (a,w,b) \in E - edges H. \neg (forest (add-edge a w b H)))
{ fix u v
  assume uv: v \in V u \in V
  assume nodes-connected G u v
  then have i: ((a, v) \in \{(a, b) | a. b. nodes-connected G a b\} by auto

  have nodes-connected H u v
  proof (rule ccontr)
    assume \neg nodes-connected H u v
    then have ii: (u, v) \notin \{(a, b) | a. b. nodes-connected H a b\} by auto
    have subgraph G G by (auto simp: subgraph-def)
    from augment-edge[OF this assms(2) i ii] obtain e a b where
    k: (a, b) \notin \{(a, b) | a. b. nodes-connected H a b\}
    and nn: e \notin edges H e \in E and ee: (case e of (aa, w, ba) \Rightarrow a=aa \land b=ba
    \lor a=ba \land b=aa)
    by blast
    obtain x w y where e: e=(x,w,y) apply(cases e) by auto
    from e ee have x=a \land y=b \lor x=b \land y=a by auto
    with k have k': \neg nodes-connected H x y
    using valid-graph.nodes-connected-sym[OF valid-subgraph[OF assms(2)]] by auto
    have xy: x \in V y \in V using nn(2) by (auto dest: E-validD)
    then have nxy: x \in nodes H y \in nodes H by auto
    from forest.forest-add-edge[OF assms(1) nxy k'] have
    forest (add-edge x w y H).
    moreover have (x, w, y) \in E - edges H using nn e by auto
    ultimately show False using pff by blast
  qed
}
then show maximally-connected H G
  unfolding maximally-connected-def by auto
qed

lemma forest-maximally-connected-incl-max2:
assumes
  forest H
  subgraph H G
shows maximally-connected H G \Rightarrow (\forall (a,w,b) \in E - edges H. \neg (forest (add-edge a w b H)))
proof -
  from assms(2) have V[simp]: nodes H = nodes G unfolding subgraph-def by auto

  assume mc: maximally-connected H G
  then have k: \\\land v v'. v \in V \Rightarrow v' \in V \Rightarrow
                       nodes-connected G v v' \Rightarrow nodes-connected H v v'
  unfolding maximally-connected-def by auto
show \( (\forall (a,w,b) \in E - \text{edges } H. \neg (\text{forest} \ (\text{add-edge} \ a \ w \ b \ H))) \)

proof (safe)
  fix \(x \ w \ y\)
  assume \(i: (x, w, y) \in E \text{ and } ni: (x, w, y) \notin \text{edges } H\)
  and \(f: \text{forest} \ (\text{add-edge} \ x \ w \ y \ H)\)
  from \(i\) have \(xy: x \in V \ w \ y \in V\) by (auto dest: E-validD)
  from \(f\) have \(\forall (a,wa,b) \in \text{insert} \ (x, w, y) \ (\text{edges } H). \neg \text{nodes-connected} \ (\text{delete-edge} \ a \ wa \ b \ (\text{add-edge} \ x \ w \ y \ H)) \ a \ b\)
    unfolding forest-def forest-axioms-def by auto
  then have \(\neg \text{nodes-connected} \ (\text{delete-edge} \ x \ w \ y \ (\text{add-edge} \ x \ w \ y \ H)) \ x \ y\)
    by auto
  moreover have \((\text{delete-edge} \ x \ w \ y \ (\text{add-edge} \ x \ w \ y \ H)) = H\)
    using \(ni \ xy\) by (auto simp: add-edge-def delete-edge-def insert-absorb)
  ultimately have \(\neg \text{nodes-connected } H \ x \ y\) by auto
  moreover from \(i\) have \(\text{nodes-connected} \ G \ x \ y\) apply - apply (rule exI[where \(x=([x,w,y])\)]
    by (auto dest: E-validD)
  ultimately show False using \(k[OF \ xy]\) by simp
qed
qed

lemma forest-maximally-connected-incl-max-conv:
  assumes \(\text{forest } H\)
  subgraph \(H \ G\)
  shows \(\text{maximally-connected} \ H \ G = (\forall (a,w,b) \in E - \text{edges } H. \neg (\text{forest} \ (\text{add-edge} \ a \ w \ b \ H)))\)
  using asms forest-maximally-connected-incl-max2 forest-maximally-connected-incl-max1
by blast

end

end

8 Kruskal on Symmetric Directed Graph

theory Graph-Definition-Impl
imports
Kruskal-Impl Graph-Definition-Aux
begin

8.1 Interpreting Kruskal-Impl
locale fromlist = fixes
  \(L :: (\text{nat} \times \text{int} \times \text{nat})\) \(\text{list}\)
begin

70
abbreviation $E \equiv \text{set } L$
abbreviation $V \equiv \text{fst } E \cup (\text{snd} \circ \text{snd}) \cdot E$
abbreviation $\text{ind } (E'::(\text{nat} \times \text{int} \times \text{nat}) \text{ set}) \equiv (\text{nodes}=V, \text{edges}=E')$
abbreviation $\text{subforest } E' \equiv \text{forest } (\text{ind } E') \land \text{subgraph } (\text{ind } E') \cdot (\text{ind } E)$

lemma max-node-is-Max-V: $E = \text{set } la \implies \text{max-node } la = \text{Max } (\text{insert } 0 V)$
proof
  assume $E: E = \text{set } la$
  have $*: \text{fst } \text{set } la \cup (\text{snd} \circ \text{snd}) \cdot \text{set } la$
    $= (\bigcup x \in \text{set } la. \text{case } x \text{ of } (x1, x1a, x2a) \Rightarrow \{x1, x2a\})$
    by auto force
  show $?\text{thesis}$
    unfolding $E$
    by (auto simp add: max-node-def prod.case-distrib $*$)
qed

lemma ind-valid-graph: $\forall E'. E' \subseteq E \implies \text{valid-graph } (\text{ind } E')$
  unfolding valid-graph-def by force

lemma vE: valid-graph (ind $E$) apply(rule ind-valid-graph) by simp

lemma ind-valid-graph': $\forall E'. \text{subgraph } (\text{ind } E') \cdot (\text{ind } E) \implies \text{valid-graph } (\text{ind } E')$
  apply(rule ind-valid-graph) by(auto simp: subgraph-def)

lemma add-edge-ind: $(a,w,b) \in E \implies \text{add-edge } a \ w \ b \ (\text{ind } F) = \text{ind } (\text{insert } (a,w,b) \ F)$
  unfolding add-edge-def by force

lemma nodes-connected-ind-sym: $F \subseteq E \implies \text{sym } \{(x, y) | x \ y. \text{nodes-connected } (\text{ind } F) \ x \ y\}$
  apply(rule ind-valid-graph)
  unfolding sym-def using valid-graph.nodes-connected-sym by fast

lemma nodes-connected-ind-trans: $F \subseteq E \implies \text{trans } \{(x, y) | x \ y. \text{nodes-connected } (\text{ind } F) \ x \ y\}$
  apply(rule ind-valid-graph)
  unfolding trans-def using valid-graph.is-path-undir-append by fast

lemma part-equiv-nodes-connected-ind:
  $F \subseteq E \implies \text{part-equiv } \{(x, y) | x \ y. \text{nodes-connected } (\text{ind } F) \ x \ y\}$
  apply(rule) using nodes-connected-ind-trans nodes-connected-ind-sym by auto

sublocale s: Kruskal-Impl $E \ V$
  λe. (fst e, snd (snd e)) λu v (a,w,b). u=a ∧ v=b ∨ u=b ∧ v=a
subforest
λE′. { (a,b) | a b. nodes-connected (ind E′) a b}
λ(u,w,v). w id PR-CONST (λ(u,w,v). RETURN (u,v))
PR-CONST (RETURN L) return L set L (λ(u,w,v). return (u,v))

proof (unfold-locals, goal-cases)
  show finite E by simp
next
  fix E′
  assume forest (ind E′) ∧ subgraph (ind E′) {nodes=V, edges=E′}
  then show E′ ⊆ E unfolding subgraph-def by auto
next
  show subforest {} by (auto simp: subgraph-def forest-def valid-graph-def forest-axioms-def)
next
  case (4 X Y)
  then have *: subgraph (ind Y) (ind X) subgraph (ind Y) (ind E)
    unfolding subgraph-def by auto
  with 4 show ?case using forest.subgraph-forest by auto
next
  case (5 u v)
  have k: valid-graph (ind {}) apply(rule ind-valid-graph) by simp
  show ?case
  apply auto
  subgoal for p apply(cases p) by auto
  subgoal for p apply(cases p) by auto
  subgoal apply(rule exI[where x=\_]) by auto
  subgoal apply(rule exI[where x=\_]) by force done
next
  case (6 E1 E2 u v)
  have *: valid-graph (ind E) apply(rule ind-valid-graph) by simp
  from 6 show ?case using valid-graph.augment-edge[of ind E ind E1 ind E2 u v, OF *]
    unfolding subgraph-def by simp
next
  case (7 F e u v)
  then have f: forest (ind F) and s: subgraph (ind F) (ind E) by auto
  from 7 have uv: u∈V v∈V by force+
  obtain a w b where e: e=(a,w,b) apply(cases e) by auto
  from e 7(3) have abuv: u=a ∧ v=b ∨ u=b ∧ v=a by auto
  show ?case
proof
  assume forest (ind (insert e F)) ∧ subgraph (ind (insert e F)) (ind E)
  then have (∀ (a, w, b)∈ insert e F. ¬nodes-connected (delete-edge a w b (ind (insert e F))) a b)
    unfolding forest-def forest-axioms-def by auto
  with e have i: ¬ nodes-connected (delete-edge a w b (ind (insert e F))) a b
  by auto
  have ii: (delete-edge a w b (ind (insert e F))) = ind F
    using 7(2) e by (auto simp: delete-edge-def)
from i have ¬ nodes-connected (ind F) a b using ii by auto
then show (u, v) ∉ {(a, b) | a b. nodes-connected (ind F) a b}
  using 7(3) valid-graph.nodes-connected-sym[OF ind-valid-graph "OF s" e]
by auto
next
from s 7(2) have sg: subgraph (ind (insert e F)) (ind E)
  unfolding subgraph-def by auto
assume: (u, v) ∉ {(a, b) | a b. nodes-connected (ind F) a b}
with above have (a, b) ∉ {(a, b) | a b. nodes-connected (ind F) a b}
  using valid-graph.nodes-connected-sym[OF ind-valid-graph "OF s"]
by auto
then have mn: ¬ nodes-connected (ind F) a b by auto
have forest (add-edge a w b (ind F)) apply(rule forest.forest-add-edge[OF f
- - mn])
  using uv abw by auto
then have f': forest (ind (insert e F)) using 7(2) add-edge-ind by (auto
simp add: e)
from f' sg show forest (ind (insert e F)) ∧ subgraph (ind (insert e F)) (ind E)
  by auto
qed
next
from 10 have xy: x∈V y∈V by force+
obtain a w b where: e = (a, w, b) apply(cases e) by auto
from 10(4) have ad-eq: add-edge a w b (ind F) = ind (insert e F)
  using e unfolding add-edge-def by (auto simp add: rev-image-eqI)
have s: ∀ x y. nodes-connected (add-edge a w b (ind F)) x y
  = ((x, y) ∈ per-union {(x, y) | x y. nodes-connected (ind F) x y} a b)
  apply(rule valid-graph.nodes-connected-insert-per-union[OF ind E])
  subgoal apply(rule ind-valid-graph) by simp
subgoal using 10(3) by (auto simp: subgraph-def)
subgoal apply(rule part-equiv-nodes-connected-ind) by fact
using xy e 10(5) by auto
show ?case
  using 10(5) e * ad-eq by auto
next
case 11
then show ?case by auto
next
case 12
then show \( \text{?case by auto} \)
next
  case 13
then show \( \text{?case by auto} \)
next
  case (14 a F e)
then obtain \( w \) where \( e=(a,w,a) \) by auto
with 14 have \( a \in V \) and \( p: (a,w,a): edges (\text{ind } \text{insert } e F) \) by auto
then have \( *: \text{nodes-connected} (\text{delete-edge } a w a (\text{ind } \text{insert } e F)) \) a a
apply (intro exI[where \( x=[] \)]) by simp
have \( \exists (a, w, b) \in \text{edges } (\text{ind } \text{insert } e F)). \text{nodes-connected} (\text{delete-edge } a w b (\text{ind } \text{insert } e F)) \) a b
apply (rule bexI[where \( x=(a,w,a) \)])
using \( * \) \( p \) by auto
then
have \( \neg \text{forest } (\text{ind } \text{insert } e F) \)
unfolding \( \text{forest-def forest-axioms-def} \) by blast
then show \( \text{?case by auto} \)
next
  case (15 e)
then show \( \text{?case by auto} \)
next
  case 16
thus \( \text{?case by force} \)
next
  case 17
thus \( \text{?case by auto} \)
next
  case (18 a b)
then show \( \text{?case apply auto} \)
subgoal for \( w \) apply (rule exI[where \( x=[(a, w, b)] \)]) by force
subgoal for \( w \) apply (rule exI[where \( x=[(a, w, b)] \)]) apply simp by blast
done
next
  case 19
thus \( \text{?case by (auto split: prod.split )} \)
next
  case 20
thus \( \text{?case by auto} \)
next
  case 21
thus \( \text{?case apply sepref-to-hoare apply sep-auto by (auto simp: pure-fold list-assn-emp)} \)
next
  case (22 l)
then show \( \text{?case using max-node-is-Max-V by auto} \)
next
  case 23

74
then show ?case apply sepref-to-hoare by sep-auto
qed

8.2 Showing the equivalence of minimum spanning forest definitions

As the definition of the minimum spanning forest from the minWeightBasis algorithm differs from the one of our graph formalization, we new show their equivalence.

lemma spanning-forest-eq: s.SpanningForest E′ = spanning-forest (ind E′) (ind E)
proof rule
assume t: s.SpanningForest E′
have f: (forest (ind E′)) and sub: subgraph (ind E′) (ind E) and
  n: (∀ x∈E − E′. ¬ (forest (ind (insert x E′)) ∧ subgraph (ind (insert x E′)) (ind E)))
  using t [unfolded s.SpanningForest-def ] by auto
have vE: valid-graph (ind E) apply (rule ind-valid-graph) by simp
have (∀ x. x∈E−E′ ⇒ subgraph (ind (insert x E′)) (ind E)
  using sub unfolding subgraph-def by auto
with n have (∀ x∈E − E′. ¬ (forest (ind (insert x E′)))) by blast
then have n′: (∀ a,w,b∈edges (ind E) − edges (ind E′). ¬ (forest (add-edge a w b (ind E′))))
  using valid-graph.E-validD[OF vE] by (auto simp: add-edge-def insert-absorb)
have mc: maximally-connected (ind E′) (ind E)
  apply (rule valid-graph.forest-maximally-connected-incl-max1) by fact+
show spanning-forest (ind E′) (ind E)
  unfolding spanning-forest-def using f sub mc by blast
next
assume t: spanning-forest (ind E′) (ind E)
have f: (forest (ind E′)) and sub: subgraph (ind E′) (ind E) and
  n: maximally-connected (ind E′) (ind E) using t [unfolded spanning-forest-def ]
by auto
have i: (∀ x. x∈E−E′ ⇒ subgraph (ind (insert x E′)) (ind E)
  using sub unfolding subgraph-def by auto
have vE: valid-graph (ind E) apply (rule ind-valid-graph) by simp
have (∀ a, w, b∈edges (ind E) − edges (ind E′). ¬ forest (add-edge a w b (ind E′))
  apply (rule valid-graph.forest-maximally-connected-incl-max2) by fact+
then have t: (∀ a, w, b∈edges (ind E) − edges (ind E′)
  ⇒ ¬ forest (add-edge a w b (ind E′))
by blast
have ii: (\( \forall x \in E - E' \). \neg (\text{forest}\ (\text{ind}\ (\text{insert}\ x\ E'))))
apply (auto simp: add-edge-def)
subgoal for a w b using \([\text{of}\ a\ w\ b]\) \text{valid-graph.E-validD[OF vE]}
  by (auto simp: add-edge-def insert-absorb)
done

from i ii have
  iii: (\( \forall x \in E - E' \). \neg (\text{forest}\ (\text{ind}\ (\text{insert}\ x\ E')) \land \text{subgraph}\ (\text{ind}\ (\text{insert}\ x\ E'))\))
  (ind E))
by blast

show s.SpanningForest E'
  unfolding s.SpanningForest-def using iii f sub
  by blast
qed

lemma edge-weight-alt: edge-weight G = \( \sum (\lambda (u,w,v). w)\) (edges G)
proof -
  have \( f: \text{fst} \circ \text{snd} = (\lambda (u,w,v). w)\) by auto
  show \( ?\text{thesis}\) unfolding edge-weight-def f
    by (auto cong: )
qed

lemma MSF-eq: s.MSF E' = minimum-spanning-forest (ind E') (ind E)
unfolding s.MSF-def minimum-spanning-forest-def optimal-forest-def
unfolding spanning-forest-eq edge-weight-alt
proof safe
  fix F'
  assume spanning-forest (ind E') (ind E)
  and B: (\( \forall B'.\) spanning-forest (ind B') (ind E))
    \( \rightarrow \) (\( \sum (u, w, v)\in E'.\) w) \( \leq \) (\( \sum (u, w, v)\in B'.\) w))
  and sf: spanning-forest F' (ind E)
from sf have subgraph F' (ind E) by (auto simp: spanning-forest-def)
then have F' = \( \text{ind}\ (\text{edges}\ F')\) unfolding subgraph-def by auto
  with B sf show (\( \sum (u, w, v)\in \text{edges}\ (ind E').\) w) \( \leq \) (\( \sum (u, w, v)\in \text{edges}\ F'.\) w)
    by auto
qed auto

lemma kruskal-correct:
  \(<\text{emp}>\) kruskal (return L) (\( \lambda (u,w,v).\) return (u,v)) ()
  \(<\lambda F.\) ↑ (distinct F \( \land \) set F \( \subseteq \) E \( \land \) minimum-spanning-forest (ind (set F))
  (ind E)))\( \rightarrow_t\)
  using s.kruskal-correct-forest unfolding MSF-eq by auto

  definition (in -) kruskal-algo L = kruskal (return L) (\( \lambda (u,w,v).\) return (u,v)) ()

end
8.3 Outside the locale

definition GD-from-list-α-weight L e = (case e of (u,w,v) ⇒ w)
abbreviation GD-from-list-α-graph G L ≡ (nodes=fst (set G) ∪ (snd o snd) ↓ (set G), edges=set L)

lemma corr:
<emp> kruskal-algo L
<λF. ↑ (set F ⊆ set L ∧ minimum-spanning-forest (GD-from-list-α-graph L F) (GD-from-list-α-graph L L))>₁
by(sep-auto heap: fromlist.kruskal-correct simp: kruskal-algo-def )

lemma kruskal-correct: <emp> kruskal-algo L
<λF. ↑ (set F ⊆ set L ∧ spanning-forest (GD-from-list-α-graph L F) (GD-from-list-α-graph L L) ∧ (∀ F'. spanning-forest (GD-from-list-α-graph L F') (GD-from-list-α-graph L L)) → sum (λ(u,w,v). w) (set F) ≤ sum (λ(u,w,v). w) (set F'))>₁
proof –
interpret fromlist L by unfold-locales
have *: (∃ F'. edge-weight (ind F') = sum (λ(u,w,v). w) F')
unfolding edge-weight-def apply auto by (metis fn-snd-conv fst-def)
show ?thesis using *
by (sep-auto heap: corr simp: minimum-spanning-forest-def optimal-forest-def)
qed

8.4 Code export

export-code kruskal-algo checking SML-imp

ML-val {
val export-nat = @ {code integer-of-nat}
val import-nat = @ {code nat-of-integer}
val export-int = @ {code integer-of-int}
val import-int = @ {code int-of-integer}
val import-list = map (fn (a,b,c) => (import-nat a, (import-int b, import-nat c)))
val export-list = map (fn (a,(b,c)) => (export-nat a, export-int b, export-nat c))
val export-Some-list = (fn SOME l => SOME (export-list l) | NONE => NONE)

fun kruskal l = @ {code kruskal} (fn () => import-list l) (fn (a,(c)) => fn () => (a,c)) () ()
|> export-list
fun kruskal-algo l = @ {code kruskal-algo} (import-list l) () |> export-list

val result = kruskal ((1,9,2),(2,3,3),(3,4,1)]
val result4 = kruskal ((1,100,4), (3,64,5), (1,13,2), (3,20,2), (2,5,5), (4,80,3), (4,40,5)]
val result' = kruskal-algo [(1, 2), (2, 3, 3), (3, 4, 1)]
val result1' = kruskal-algo [(1, 2), (2, 3, 3), (3, 4, 1), (1, 5, 3)]
val result2' = kruskal-algo [(1, 2), (2, 3, 3), (3, 4, 1), (1, 4, 3)]
val result3' = kruskal-algo [(1, 2), (2, 3, 3), (3, 4, 1), (1, 4, 1)]
val result4' = kruskal-algo [(1, 100, 4), (3, 64, 5), (1, 13, 2), (3, 20, 2),
(2, 5, 5), (4, 80, 3), (4, 40, 5)]