Kruskal's Algorithm for Minimum Spanning Forest

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

May 26, 2024

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

theory *MinWeightBasis*
imports *Refine-Monadic.Refine-Monadic Matroids.Matroid*
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**
$minBasis\ B \equiv basis\ B \wedge (\forall B'.\ basis\ B' \longrightarrow sum\ weight\ B \leq sum\ weight\ B')$

1.1 Preparations

fun *in-sort-edge* **where**
  in-sort-edge x [] = [x]
| *in-sort-edge* x (y#ys) = (*if weight* $x \leq$ *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) ⟨*proof*⟩

lemma *in-sort-edge*: *sorted-wrt* ($\lambda e1\ e2.$ *weight* $e1 \leq$ *weight* $e2$) L
  $\Longrightarrow$ *sorted-wrt* ($\lambda e1\ e2.$ *weight* $e1 \leq$ *weight* $e2$) (*in-sort-edge* x L)
  ⟨*proof*⟩

lemma *in-sort-edge-distinct*: $x \notin$ *set* $L \Longrightarrow$ *distinct* $L \Longrightarrow$ *distinct* (*in-sort-edge* x L)
  ⟨*proof*⟩

lemma *finite-sorted-edge-distinct*:
  assumes *finite* S
  obtains L **where** *distinct* L *sorted-wrt* ($\lambda e1\ e2.$ *weight* $e1 \leq$ *weight* $e2$) L S = *set* L
⟨*proof*⟩

abbreviation *wsorted* == *sorted-wrt* ($\lambda e1\ e2.$ *weight* $e1 \leq$ *weight* $e2$)

lemma *sum-list-map-cons*:
  sum-list (*map weight* (y # ys)) = *weight* y + *sum-list* (*map weight* ys)
  ⟨*proof*⟩

lemma *exists-greater*:
  assumes *len*: *length* F = *length* F'
    and *sum*: *sum-list* (*map weight* F) > *sum-list* (*map weight* F')
   shows $\exists\, i<$*length* F. *weight* (F ! i) > *weight* (F' ! i)
⟨*proof*⟩

lemma *wsorted-nth-mono*: **assumes** *wsorted* L $i \leq j$ $j<$*length* L
  shows *weight* (L!i) $\leq$ *weight* (L!j)
  ⟨*proof*⟩

1.1.1 Weight restricted set

limi T g is the set T restricted to elements only with weight strictly smaller than g.

definition *limi* T g == $\{e.\ e \in T \wedge$ *weight* $e < g\}$

lemma *limi-subset*: *limi* T $g \subseteq T$ ⟨*proof*⟩

lemma *limi-mono*: $A \subseteq B \Longrightarrow$ *limi* A $g \subseteq$ *limi* B g ⟨*proof*⟩

1.1.2 The greedy idea

definition *no-smallest-element-skipped E F*
$= (\forall\, e \in carrier - E.\ \forall\, g > weight\ e.\ indep\ (insert\ e\ (limi\ F\ g)) \longrightarrow (e \in limi\ F\ g))$

let F be a set of elements *limi F g* is F restricted to elements with weight smaller than g let E be a set of elements we want to exclude.

no-smallest-element-skipped E F expresses, that going greedily over *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* {}
⟨*proof*⟩

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 $e \in limi\ F\ g$
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *no-smallest-element-skipped-add*:
assumes *I*: *no-smallest-element-skipped* $(E \cup \{e\})$ *F*
shows *no-smallest-element-skipped E* (*insert e F*)
⟨*proof*⟩

1.2 Minimum Weight Basis algorithm

definition *obtain-sorted-carrier* $\equiv$ *SPEC* $(\lambda L.\ wsorted\ L \wedge set\ L = carrier)$

abbreviation *empty-basis* $\equiv$ {}

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 dependet.

definition *minWeightBasis* **where**
minWeightBasis $\equiv$ *do* {
$l \leftarrow$ *obtain-sorted-carrier*;
ASSERT (*set l = carrier*);
$T \leftarrow$ *nfoldli l* (λ-. *True*)
(λ*e 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⟩
```

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 == λ(T,E). indep T
              ∧ T ⊆ carrier
              ∧ E ⊆ carrier
              ∧ (∀ x∈T.∀ y∈E. weight x ≤ weight y)
              ∧ (∀ e∈carrier−E−T. ~indep (insert e T))
              ∧ no-smallest-element-skipped E T
```

```
lemma I-minWeightBasisD:
  assumes
   I-minWeightBasis (T,E)
 showsindep T ⋀e. e∈carrier−E−T ⟹ ~indep (insert e T)
   E ⊆ carrier ⋀x y. x∈T ⟹ y∈E ⟹ weight x ≤ weight y  T ⊆ carrier
   no-smallest-element-skipped E T
  ⟨proof⟩
```

```
lemma I-minWeightBasisI:
  assumes indep T ⋀e. e∈carrier−E−T ⟹ ~indep (insert e T)
   E ⊆ carrier ⋀x y. x∈T ⟹ y∈E ⟹ weight x ≤ weight y  T ⊆ carrier
   no-smallest-element-skipped E T
  shows I-minWeightBasis (T,E)
  ⟨proof⟩
```

lemma *I-minWeightBasisG*: *I-minWeightBasis* $(T,E) \Longrightarrow$ *no-smallest-element-skipped* E T
⟨*proof*⟩

lemma *I-minWeightBasis-sorted*: *I-minWeightBasis* $(T,E) \Longrightarrow (\forall x \in T. \forall y \in E.$ *weight* $x \leq$ *weight* $y)$
⟨*proof*⟩

1.5 Invariant proofs

lemma *I-minWeightBasis-empty*: *I-minWeightBasis* ({}, *carrier*)
⟨*proof*⟩

lemma *I-minWeightBasis-final*: *I-minWeightBasis* (T, {}) $\Longrightarrow$ *minBasis* T
⟨*proof*⟩

lemma *indep-aux*:
assumes $e \in E$ $\forall e \in carrier - E - F.$ ¬ *indep* (*insert* e F)
and $x \in carrier - (E - \{e\}) -$ *insert* e F
shows ¬ *indep* (*insert* x (*insert* e F))
⟨*proof*⟩

lemma *preservation-if*: *wsorted* $x \Longrightarrow$ *set* x = *carrier* $\Longrightarrow$
$x = l1$ @ xa # $l2 \Longrightarrow$ *I-minWeightBasis* (σ, *set* (xa # $l2$)) $\Longrightarrow$ *indep* σ
$\Longrightarrow xa \in$ *carrier* $\Longrightarrow$ *indep* (*insert* xa σ) $\Longrightarrow$ *I-minWeightBasis* (*insert* xa σ, *set* $l2$)
⟨*proof*⟩

lemma *preservation-else*: *set* x = *carrier* $\Longrightarrow$
$x = l1$ @ xa # $l2 \Longrightarrow$ *I-minWeightBasis* (σ, *set* (xa # $l2$))
$\Longrightarrow$ *indep* σ $\Longrightarrow$ ¬ *indep* (*insert* xa σ) $\Longrightarrow$ *I-minWeightBasis* (σ, *set* $l2$)
⟨*proof*⟩

1.6 The refinement lemma

theorem *minWeightBasis-refine*: (*minWeightBasis*, *SPEC minBasis*)∈⟨*Id*⟩*nres-rel*
⟨*proof*⟩

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*: ⋀*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*
  ⟨*proof*⟩

lemma *finiteE-finiteV*: *finite E* $\Longrightarrow$ *finite V*
⟨*proof*⟩

lemma *E-inV*: $\bigwedge e.\ e \in E \Longrightarrow$ *vertices e* $\subseteq V$
⟨*proof*⟩

definition $CC\ E'\ x = (connected\ E')``\{x\}$

lemma *sameCC-reachable*: $E' \subseteq E \Longrightarrow u \in V \Longrightarrow v \in V \Longrightarrow CC\ E'\ u = CC\ E'\ v \longleftrightarrow (u,v) \in connected\ E'$
⟨*proof*⟩

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

lemma $quotient\ V\ Id = \{\{v\}|v.\ v \in V\}$ ⟨*proof*⟩

lemma *CCs-empty*: $CCs\ \{\} = \{\{v\}|v.\ v \in V\}$
⟨*proof*⟩

lemma *CCs-empty-card*: $card\ (CCs\ \{\}) = card\ V$
⟨*proof*⟩

lemma *CCs-imageCC*: $CCs\ F = (CC\ F)\ `\ V$
⟨*proof*⟩

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*⟩

lemma *forest-CCs*: **assumes** *forest E'* **shows** $card\ (CCs\ E') + card\ E' = card\ V$
⟨*proof*⟩

lemma *pigeonhole-CCs*:
assumes *finiteV*: *finite V* **and** *cardlt*: $card\ (CCs\ E1) < card\ (CCs\ E2)$
shows $(\exists\ u\ v.\ u \in V \wedge v \in V \wedge CC\ E1\ u = CC\ E1\ v \wedge CC\ E2\ u \neq CC\ E2\ v)$
⟨*proof*⟩

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*⟩

sublocale *weighted-matroid E forest weight*
⟨*proof*⟩

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)
```

$= (if\ e\notin\ T \wedge indep\ (insert\ e\ T)\ then$
$RETURN\ (insert\ e\ T)$
$else$
$RETURN\ T)$
⟨*proof*⟩

lemma *kruskal0-refine*: $(kruskal0,\ minWeightBasis) \in \langle Id\rangle nres\text{-}rel$
⟨*proof*⟩

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*
⟨*proof*⟩

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

lemma *uf-graph-relsndD*: ((*a*,*b*),*c*) ∈ *uf-graph-rel* ⟹ *b*=*c*
⟨*proof*⟩

lemma *uf-graph-relD*: ((*a*,*b*),*c*) ∈ *uf-graph-rel* ⟹ *b*=*c* ∧ *uf-graph-invar* (*a*,*b*)
⟨*proof*⟩

definition *kruskal1*
where *kruskal1* ≡ *do* {
l ← *obtain-sorted-carrier*;
let initial-union-find = *per-init V*;
(*per*, *spanning-forest*) ← *nfoldli l* (λ-. *True*)
(λ*e* (*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*)
}

else
  RETURN (*uf*,*T*)
}) (*initial-union-find*, *empty-forest*);
RETURN spanning-forest
}

lemma *corresponding-union-find-empty*:
shows *corresponding-union-find* (*per-init V*) *empty-forest*
⟨*proof*⟩

lemma *empty-forest-refine*: ((*per-init V*, *empty-forest*), *empty-forest*)∈*uf-graph-rel*
⟨*proof*⟩

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*)
⟨*proof*⟩

theorem *kruskal1-refine*: (*kruskal1*, *kruskal0*)∈⟨*Id*⟩*nres-rel*
⟨*proof*⟩

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 weight*
for *E V vertices joins forest connected* **and** *weight* :: ′*edge* ⇒ *int* +
fixes
α :: ′*cedge* ⇒ ′*edge*
and *endpoints* :: ′*cedge* ⇒ (′*a*∗′*a*) *nres*
assumes
endpoints-refine: α *xi* = *x* ⟹ *endpoints xi* ≤ ⇓ *Id* (*a-endpoints x*)
begin

definition *wsorted'* **where** *wsorted'* == *sorted-wrt* ($\lambda x\ y.\ weight\ (\alpha\ x) \leq weight\ (\alpha\ y)$)

lemma *wsorted-map*α[*simp*]: *wsorted' s* $\Longrightarrow$ *wsorted* (*map* α *s*)
⟨*proof*⟩

definition *obtain-sorted-carrier'* == *SPEC* ($\lambda L.\ wsorted'\ L \wedge \alpha$ ' *set* $L = E$)

abbreviation *concrete-edge-rel* :: (*'cedge* × *'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*
⟨*proof*⟩

definition *kruskal2*
where *kruskal2* ≡ *do* {
  l ← *obtain-sorted-carrier'*;
  let initial-union-find = *per-init V*;
  (*per*, *spanning-forest*) ← *nfoldli l* (λ-. *True*)
    (λ*ce* (*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*
⟨*proof*⟩

lemma *loop-initial-rel*:
((*per-init V*, []), *per-init V*, {}) ∈ *Id* $\times_r$ ⟨*concrete-edge-rel*⟩*list-set-rel*
⟨*proof*⟩

lemma *concrete-edge-rel-list-set-rel*:
(*a*, *b*) ∈ ⟨*concrete-edge-rel*⟩*list-set-rel* $\Longrightarrow$ α ' (*set a*) = *b*
⟨*proof*⟩

theorem *kruskal2-refine*: (*kruskal2*, *kruskal1*)∈⟨⟨*concrete-edge-rel*⟩*list-set-rel*⟩*nres-rel*
⟨*proof*⟩

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* $\Rightarrow$ *int*
    and α **and** *endpoints* :: *nat* $\times$ *int* $\times$ *nat* $\Rightarrow$ (*nat* $\times$ *nat*) *nres*
    +
  fixes *getEdges* :: (*nat* $\times$ *int* $\times$ *nat*) *list nres*
    and *getEdges-impl* :: (*nat* $\times$ *int* $\times$ *nat*) *list Heap*
    and *superE* :: (*nat* $\times$ *int* $\times$ *nat*) *set*
    and *endpoints-impl* :: (*nat* $\times$ *int* $\times$ *nat*) $\Rightarrow$ (*nat* $\times$ *nat*) *Heap*
  assumes
    getEdges-refine: *getEdges* $\leq$ *SPEC* ($\lambda L.$ α ' *set L* = *E*
        $\wedge$ ($\forall$ (*a*,*wv*,*b*)$\in$*set L*. *weight* (α (*a*,*wv*,*b*)) = *wv*) $\wedge$ *set L* $\subseteq$ *superE*)
    and
    getEdges-impl: (*uncurry0 getEdges-impl*, *uncurry0 getEdges*)
        $\in$ *unit-assn*^{k} $\rightarrow_a$ *list-assn* (*nat-assn* $\times_a$ *int-assn* $\times_a$ *nat-assn*)
    and
    max-node-is-Max-V: *E* = α ' *set la* $\Longrightarrow$ *max-node la* = *Max* (*insert 0 V*)
    and
    endpoints-impl: (*endpoints-impl*, *endpoints*)
        $\in$ (*nat-assn* $\times_a$ *int-assn* $\times_a$ *nat-assn*)^k $\rightarrow_a$ (*nat-assn* $\times_a$ *nat-assn*)
begin

  lemma *this-loc*: *Kruskal-Impl E V vertices joins forest connected weight*
      α *endpoints getEdges getEdges-impl superE endpoints-impl* $\langle$*proof*$\rangle$

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 $\leftarrow$ *SPEC* ($\lambda L.$ α ' *set L* = *E*
        $\wedge$ ($\forall$ (*a*,*wv*,*b*)$\in$*set L*. *weight* (α (*a*,*wv*,*b*)) = *wv*) $\wedge$ *set L* $\subseteq$ *superE*);
    SPEC ($\lambda L.$ *sorted-wrt edges-less-eq L* $\wedge$ *set L* = *set l*)
  }

  lemma *wsorted'-sorted-wrt-edges-less-eq*:
    assumes $\forall$ (*a*,*wv*,*b*)$\in$*set s*. *weight* (α (*a*,*wv*,*b*)) = *wv*
      sorted-wrt edges-less-eq s

shows *wsorted′ s*
⟨*proof*⟩

lemma *obtain-sorted-carrier′′-refine*:
(*obtain-sorted-carrier′′*, *obtain-sorted-carrier′*) ∈ ⟨*Id*⟩*nres-rel*
⟨*proof*⟩

definition *obtain-sorted-carrier′′′* =
do {
l ← *getEdges*;
RETURN (*quicksort-by-rel edges-less-eq* [] *l*, *max-node l*)
}

definition *add-size-rel* = *br fst* (λ(*l*,*n*). *n*= *Max* (*insert 0 V*))

lemma *obtain-sorted-carrier′′′-refine*:
(*obtain-sorted-carrier′′′*, *obtain-sorted-carrier′′*) ∈ ⟨*add-size-rel*⟩*nres-rel*
⟨*proof*⟩

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*)
(λ*ce* (*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*)
⟨*proof*⟩

lemma *kruskal3-subset*:
shows *kruskal3* $\leq_n$ *SPEC* (λ*T*. *distinct T* ∧ *set T* ⊆ *superE*)
⟨*proof*⟩

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 p2*
⟨*proof*⟩

lemma *per-supset-compare*:
(*p1*, *p2*) ∈ *per-supset-rel* ⟹ *x1*∈*Domain p2* ⟹ *x2*∈*Domain p2*
⟹ *per-compare p1 x1 x2* ⟷ *per-compare p2 x1 x2*
⟨*proof*⟩

lemma *per-supset-union*: (*p1*, *p2*) ∈ *per-supset-rel* ⟹ *x1*∈*Domain p2* ⟹ *x2*∈*Domain p2* ⟹
(*per-union p1 x1 x2*, *per-union p2 x1 x2*) ∈ *per-supset-rel*
⟨*proof*⟩

lemma *per-initN-refine*: (*per-init'* (*Max* (*insert 0 V*) + *1*), *per-init V*) ∈ *per-supset-rel*
⟨*proof*⟩

theorem *kruskal3-refine*: (*kruskal3*, *kruskal2*)∈⟨*Id*⟩*nres-rel*
⟨*proof*⟩

4.2.2 Synthesis of Kruskal by SepRef

lemma [*sepref-import-param*]: (*sort-edges*,*sort-edges*)∈⟨$Id \times_r Id \times_r Id$⟩*list-rel* →⟨$Id \times_r Id \times_r Id$⟩*list-rel*
⟨*proof*⟩
lemma [*sepref-import-param*]: (*max-node*, *max-node*) ∈ ⟨$Id \times_r Id \times_r Id$⟩*list-rel* → *nat-rel* ⟨*proof*⟩

sepref-register *getEdges* :: (*nat* × *int* × *nat*) *list nres*
sepref-register *endpoints* :: (*nat* × *int* × *nat*) ⇒ (*nat*∗*nat*) *nres*

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

schematic-goal *kruskal-impl*:
(*uncurry0 ?c*, *uncurry0 kruskal3*) ∈ $(unit\text{-}assn)^k \rightarrow_a$ *list-assn* (*nat-assn* $\times_a$ *int-assn* $\times_a$ *nat-assn*)
⟨*proof*⟩

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'*:
$(uncurry0\ kruskal3,\ uncurry0\ (SPEC\ (\lambda r.\ MSF\ (\alpha\ `\ set\ r)))) \in unit\text{-}rel \rightarrow_f \langle Id\rangle nres\text{-}rel$
⟨*proof*⟩

lemma *kruskal3-ref-spec*:
$(uncurry0\ kruskal3,$
$uncurry0\ (SPEC\ (\lambda r.\ distinct\ r \wedge set\ r \subseteq superE \wedge\ MSF\ (\alpha\ `\ set\ r))))$
$\in unit\text{-}rel \rightarrow_f \langle Id\rangle nres\text{-}rel$
⟨*proof*⟩

lemma [*fcomp-norm-simps*]: $list\text{-}assn\ (nat\text{-}assn \times_a int\text{-}assn \times_a nat\text{-}assn) = id\text{-}assn$
⟨*proof*⟩

lemmas *kruskal-ref-spec* = *kruskal-refine*[*FCOMP kruskal3-ref-spec*]

The final correctness lemma for Kruskal's algorithm.

lemma *kruskal-correct-forest*:
shows <*emp*> *kruskal getEdges-impl endpoints-impl* ()
$<\lambda r.\ \uparrow(\ distinct\ r \wedge set\ r \subseteq superE \wedge MSF\ (set\ (map\ \alpha\ r)))>_t$
⟨*proof*⟩

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* $\Rightarrow$ $'a \Rightarrow$ ($'a$ *uprod*) *list* $\Rightarrow$ $'a \Rightarrow$ *bool* **where**
epath E u [] *v* = (*u* = *v*)
| *epath E u* (*x*#*xs*) *v* $\longleftrightarrow$ ($\exists$ *w*. *u*$\neq$*w* $\wedge$ *Upair u w* = *x* $\wedge$ *epath E w xs v*) $\wedge$ *x*$\in$*E*

lemma [*simp*,*intro*!]: *epath E u* [] *u* ⟨*proof*⟩

lemma *epath-subset-E*: *epath E u p v* $\Longrightarrow$ *set p* $\subseteq$ *E*
⟨*proof*⟩

lemma *path-append-conv*[*simp*]: *epath E u* (*p*@*q*) *v* $\longleftrightarrow$ ($\exists$ *w*. *epath E u p w* $\wedge$ *epath E w q v*)
⟨*proof*⟩

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

lemma *epath E x p y* $\Longrightarrow$ $\exists$ *p*. *epath E y p x*
⟨*proof*⟩

lemma *epath-mono*: *E* $\subseteq$ *E'* $\Longrightarrow$ *epath E u p v* $\Longrightarrow$ *epath E' u p v*
⟨*proof*⟩

lemma *epath-restrict*: *set p* $\subseteq$ *I* $\Longrightarrow$ *epath E u p v* $\Longrightarrow$ *epath* (*E*$\cap$*I*) *u p v*
⟨*proof*⟩

lemma assumes *A*$\subseteq$*A'* ~ *epath A u p v epath A' u p v*
shows *epath-diff-edge*: ($\exists$ *e*. *e*$\in$*set p* − *A*)
⟨*proof*⟩

lemma *epath-restrict'*: *epath* (*insert e E*) *u p v* $\Longrightarrow$ *e*$\notin$*set p* $\Longrightarrow$ *epath E u p v*
⟨*proof*⟩

lemma *epath-not-direct*:
assumes *ep*: *epath E u p v* **and** *unv*: *u* $\neq$ *v*
and *edge-notin*: *Upair u v* $\notin$ *E*
shows *length p* $\geq$ *2*
⟨*proof*⟩

lemma *epath-decompose*:
assumes *e*: *epath G v p v'*
and *elem* :*Upair a b* $\in$ *set p*
shows $\exists$ *u u' p' p''* . *u* $\in$ {*a*, *b*} $\wedge$ *u'* $\in$ {*a*, *b*} $\wedge$ *epath G v p' u* $\wedge$ *epath G u' p'' v'* $\wedge$
length p' < *length p* $\wedge$ *length p''* < *length p*
⟨*proof*⟩

lemma *epath-decompose'*:
assumes *e*: *epath G v p v'*
and *elem* :*Upair a b* ∈ *set p*
shows ∃ *u u' p' p''* . *Upair a b* = *Upair u u'* ∧ *epath G v p' u* ∧ *epath G u' p'' v'* ∧
length p' < *length p* ∧ *length p''* < *length p*
⟨*proof*⟩

lemma *epath-split-distinct*:
assumes *epath G v p v'*
assumes *Upair a b* ∈ *set p*
shows (∃ *p' p'' u u'*.
epath G v p' u ∧ *epath G u' p'' v'* ∧
length p' < *length p* ∧ *length p''* < *length p* ∧
(*u* ∈ {*a*, *b*} ∧ *u'* ∈ {*a*, *b*}) ∧
Upair a b ∉ *set p'* ∧ *Upair a b* ∉ *set p''*)
⟨*proof*⟩

5.2 Distinct edge path

definition *depath E u dp v* ≡ *epath E u dp v* ∧ *distinct dp*

lemma *epath-to-depath*: *set p* ⊆ *I* ⟹ *epath E u p v* ⟹ ∃ *dp*. *depath E u dp v* ∧ *set dp* ⊆ *I*
⟨*proof*⟩

lemma *epath-to-depath'*: *epath E u p v* ⟹ ∃ *dp*. *depath E u dp v*
⟨*proof*⟩

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*}
⟨*proof*⟩

lemma *uconnected-refl*: *refl* (*uconnected E*)
⟨*proof*⟩

lemma *uconnected-sym*: *sym* (*uconnected E*)
⟨*proof*⟩
lemma *uconnected-trans*: *trans* (*uconnected E*)
⟨*proof*⟩

lemma *uconnected-symI*: (*u*,*v*) ∈ *uconnected E* ⟹ (*v*,*u*) ∈ *uconnected E*
⟨*proof*⟩

lemma *equiv UNIV* (*uconnected E*)
⟨*proof*⟩

lemma *uconnected-refcl*: $(uconnected\ E)^* = (uconnected\ E)^=$
⟨*proof*⟩

lemma *uconnected-transcl*: $(uconnected\ E)^* = uconnected\ E$
⟨*proof*⟩

lemma *uconnected-mono*: $A \subseteq A' \Longrightarrow uconnected\ A \subseteq uconnected\ A'$
⟨*proof*⟩

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*
⟨*proof*⟩

5.4 Forest

definition *forest E* ≡ ~(∃ *u p. decycle E u p*)

lemma *forest-mono*: $Y \subseteq X \Longrightarrow forest\ X \Longrightarrow forest\ Y$
⟨*proof*⟩

lemma *forrest2-E*: **assumes** (*u*,*v*) ∈ *uconnected E*
and *Upair u v* ∉ *E*
and $u \neq v$
shows ~ *forest* (*insert* (*Upair u v*) *E*)
⟨*proof*⟩

lemma *insert-stays-forest-means-not-connected*: **assumes** *forest* (*insert* (*Upair u v*) *E*)
and (*Upair u v*) ∉ *E*
and $u \neq v$
shows ~ (*u*,*v*) ∈ *uconnected E*
⟨*proof*⟩

lemma *epath-singleton*: *epath F a* [*e*] *b* ⟹ *e* = *Upair a b*
⟨*proof*⟩

lemma *forest-alt1*:
assumes *Upair a b* ∈ *F forest F* ⋀*e. e*∈*F* ⟹ *proper-uprod e*
shows (*a*,*b*) ∉ *uconnected* (*F* − {*Upair a b*})
⟨*proof*⟩

lemma *forest-alt2*:
  assumes $\bigwedge e.\ e{\in}F \Longrightarrow$ *proper-uprod e*
    and $\bigwedge a\ b.$ *Upair a b* $\in F \Longrightarrow (a,b) \notin$ *uconnected* $(F - \{$*Upair a b*$\})$
  shows *forest F*
⟨*proof*⟩

lemma *forest-alt*:
  assumes $\bigwedge e.\ e{\in}F \Longrightarrow$ *proper-uprod e*
  shows *forest F* $\longleftrightarrow (\forall\ a\ b.$ *Upair a b* $\in F \longrightarrow (a,b) \notin$ *uconnected* $(F - \{$*Upair a b*$\}))$
  ⟨*proof*⟩

lemma *augment-forest-overedges*:
  assumes $F{\subseteq}E$ *forest F* (*Upair u v*) $\in E$ $(u,v) \notin$ *uconnected F*
    and *notsame*: $u{\neq}v$
  shows *forest* (*insert* (*Upair u v*) *F*)
  ⟨*proof*⟩

5.5 uGraph locale

locale *uGraph* =
  fixes *E* :: *'a uprod set*
    and *w* :: *'a uprod* ⇒ *'c*::{*linorder*, *ordered-comm-monoid-add*}
  assumes *ecard2*: $\bigwedge e.\ e{\in}E \Longrightarrow$ *proper-uprod e*
    and *finiteE*[*simp*]: *finite E*
begin

abbreviation *uconnected-on E' V* ≡ *uconnected E'* ∩ (*V*×*V*)

abbreviation *verts* ≡ ⋃(*set-uprod* ‘ *E*)

lemma *set-uprod-nonemptyY*[*simp*]: *set-uprod x* ≠ {} ⟨*proof*⟩

abbreviation *uconnectedV E'* ≡ *Restr* (*uconnected E'*) *verts*

lemma *equiv-unconnected-on*: *equiv V* (*uconnected-on E' V*)
  ⟨*proof*⟩

lemma *uconnectedV-refl*: *E'*⊆*E* ⟹ *refl-on verts* (*uconnectedV E'*)
  ⟨*proof*⟩

lemma *uconnectedV-trans*: *trans* (*uconnectedV E'*)
  ⟨*proof*⟩

lemma *uconnectedV-sym*: *sym* (*uconnectedV* E')
⟨*proof*⟩

lemma *equiv-vert-uconnected*: *equiv verts* (*uconnectedV* E')
⟨*proof*⟩

lemma *uconnectedV-tracl*: $(uconnectedV\ F)^* = (uconnectedV\ F)^=$
⟨*proof*⟩

lemma *uconnectedV-cl*: $(uconnectedV\ F)^+ = (uconnectedV\ F)$
⟨*proof*⟩

lemma *uconnectedV-Restrcl*: *Restr* $((uconnectedV\ F)^*)$ *verts* = (*uconnectedV* *F*)
⟨*proof*⟩

lemma *restr-ucon*: $F \subseteq E \Longrightarrow uconnected\ F = uconnectedV\ F \cup Id$
⟨*proof*⟩

lemma *relI*:
assumes $\bigwedge a\ b.\ (a,b) \in F \Longrightarrow (a,b) \in G$
and $\bigwedge a\ b.\ (a,b) \in G \Longrightarrow (a,b) \in F$ **shows** $F=G$
⟨*proof*⟩

lemma *in-per-union*: $u \in \{x,\ y\} \Longrightarrow\ u' \in \{x,\ y\} \Longrightarrow\ x\in V \Longrightarrow y\in V \Longrightarrow$
refl-on $V\ R \Longrightarrow$ *part-equiv* $R\ \Longrightarrow (u,\ u') \in$ *per-union* $R\ x\ y$
⟨*proof*⟩

lemma *uconnectedV-mono*: $(a,b)\in uconnectedV\ F \Longrightarrow F\subseteq F' \Longrightarrow (a,b)\in uconnectedV$
F'
⟨*proof*⟩

lemma *per-union-subs*: $x \in S \Longrightarrow y\in S \Longrightarrow R\subseteq S \times S \Longrightarrow$ *per-union* $R\ x\ y \subseteq S$
$\times\ S$
⟨*proof*⟩

lemma *insert-uconnectedV-per*:
assumes $x\neq y$ **and** *inV*: $x\in verts\ y\in verts$ **and** *subE*: $F\subseteq E$
shows *uconnectedV* (*insert* (*Upair* *x y*) *F*) = *per-union* (*uconnectedV* *F*) *x y*
(**is** *uconnectedV* *?F'* = *per-union* *?uf x y*)
⟨*proof*⟩

lemma *epath-filter-selfloop*: *epath* (*insert* (*Upair* *x x*) *F*) *a p b* $\Longrightarrow \exists\, p.$ *epath F a*
p b
⟨*proof*⟩

lemma *uconnectedV-insert-selfloop*: $x \in verts \Longrightarrow uconnectedV\ (insert\ (Upair\ x\ x)\ F) = uconnectedV\ F$
⟨*proof*⟩

lemma *equiv-selfloop-per-union-id*: $equiv\ S\ F \Longrightarrow x \in S \Longrightarrow per\text{-}union\ F\ x\ x = F$
⟨*proof*⟩

lemma *insert-uconnectedV-per-eq*:
assumes *inV*: $x \in verts$ **and** *subE*: $F \subseteq E$
shows $uconnectedV\ (insert\ (Upair\ x\ x)\ F) = per\text{-}union\ (uconnectedV\ F)\ x\ x$
⟨*proof*⟩

lemma *insert-uconnectedV-per'*:
assumes *inV*: $x \in verts$ $y \in verts$ **and** *subE*: $F \subseteq E$
shows $uconnectedV\ (insert\ (Upair\ x\ y)\ F) = per\text{-}union\ (uconnectedV\ F)\ x\ y$
⟨*proof*⟩

definition $subforest\ F \equiv forest\ F \wedge F \subseteq E$

definition *spanningForest* **where** $spanningForest\ X \longleftrightarrow subforest\ X \wedge (\forall x \in E - X.\ \neg\ subforest\ (insert\ x\ X))$

definition $minSpanningForest\ F \equiv spanningForest\ F \wedge (\forall F'.\ spanningForest\ F' \longrightarrow sum\ w\ F \leq sum\ w\ F')$

end

end

6 Kruskal on UGraphs

theory *UGraph-Impl*
imports
Kruskal-Impl UGraph
begin

definition $\alpha = (\lambda(u,w,v).\ Upair\ u\ v)$

6.1 Interpreting *Kruskl-Impl* with a UGraph

abbreviation (**in** *uGraph*)
getEdges-SPEC csuper-E
$\equiv (SPEC\ (\lambda L.\ distinct\ (map\ \alpha\ L) \wedge \alpha\ `\ set\ L = E$
$\wedge (\forall (a,\ wv,\ b) \in set\ L.\ w\ (\alpha\ (a,\ wv,\ b)) = wv) \wedge set\ L \subseteq csuper\text{-}E))$

locale *uGraph-impl* = *uGraph E w* **for** $E :: nat\ uprod\ set$ **and** $w :: nat\ uprod \Rightarrow$

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} $\rightarrow_a$ *list-assn* (*nat-assn* $\times_a$ *int-assn* $\times_a$ *nat-assn*)
begin

abbreviation $V \equiv \bigcup$ (*set-uprod* ' *E*)

lemma *max-node-is-Max-V*: *E* = α ' *set la* ⟹ *max-node la* = *Max* (*insert 0 V*)
⟨*proof*⟩

sublocale *s*: *Kruskal-Impl E* ⋃(*set-uprod* ' *E*) *set-uprod* λ*u v e*. *Upair u v* = *e*
subforest uconnectedV 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*))
⟨*proof*⟩

lemma *spanningForest-eq-basis*: *spanningForest* = *s.basis*
⟨*proof*⟩

lemma *minSpanningForest-eq-minbasis*: *minSpanningForest* = *s.minBasis*
⟨*proof*⟩

lemma *kruskal-correct'*:
<*emp*> *kruskal getEdges-impl* (λ(*u*,*w*,*v*). *return* (*u*,*v*)) ()
<λ*r*. ↑ (*distinct r* ∧ *set r* ⊆ *csuper-E* ∧ *s.MSF* (*set* (*map* α *r*)))>$_t$
⟨*proof*⟩

lemma *kruskal-correct*:
<*emp*> *kruskal getEdges-impl* (λ(*u*,*w*,*v*). *return* (*u*,*v*)) ()
<λ*r*. ↑ (*distinct r* ∧ *set r* ⊆ *csuper-E* ∧ *minSpanningForest* (*set* (*map* α *r*)))>$_t$
⟨*proof*⟩

end

6.2 Kruskal on UGraph from list of concrete edges

definition *uGraph-from-list-α-weight L e* = (*THE w*. ∃ *a' b'*. *Upair a' b'* = *e* ∧ (*a'*, *w*, *b'*) ∈ *set L*)
abbreviation *uGraph-from-list-α-edges L* ≡ α ' *set L*

locale *fromlist* = **fixes**
  L :: (*nat* × *int* × *nat*) *list*
assumes *dist*: *distinct* (*map* α L) **and** *no-selfloop*: ∀ *u w v*. (*u*,*w*,*v*)∈*set* L ⟶ *u*≠*v*
begin

lemma *not-distinct-map*: *a*∈*set* *l* ⟹ *b*∈*set* *l* ⟹ *a*≠*b* ⟹ α *a* = α *b* ⟹ ¬ *distinct* (*map* α *l*)

  ⟨*proof*⟩

lemma *ii*: (*a*, *aa*, *b*) ∈ *set* L ⟹ *uGraph-from-list-α-weight* L (*Upair a b*) = *aa*
  ⟨*proof*⟩

sublocale *uGraph-impl* α ‘ *set* L *uGraph-from-list-α-weight* L *return* L *set* L
⟨*proof*⟩

lemmas *kruskal-correct* = *kruskal-correct*

definition (**in** −) *kruskal-algo* L = *kruskal* (*return* L) (λ(*u*,*w*,*v*). *return* (*u*,*v*)) ()

end

6.3 Outside the locale

definition *uGraph-from-list-invar* :: (*nat*×*int*×*nat*) *list* ⇒ *bool* **where**
  uGraph-from-list-invar L = (*distinct* (*map* α L) ∧ (∀ *p*∈*set* L. *case p of* (*u*,*w*,*v*) ⇒*u*≠*v*))

lemma *uGraph-from-list-invar-conv*: *uGraph-from-list-invar* L = *fromlist* L
  ⟨*proof*⟩

lemma *uGraph-from-list-invar-subset*:
  uGraph-from-list-invar L ⟹ *set* L'⊆ *set* L ⟹ *distinct* L' ⟹ *uGraph-from-list-invar* L'
  ⟨*proof*⟩

lemma *uGraph-from-list-α-inj-on*: *uGraph-from-list-invar* E ⟹ *inj-on* α (*set* E)
  ⟨*proof*⟩

lemma *sum-easier*: *uGraph-from-list-invar* L
    ⟹ *set* E ⊆ *set* L
    ⟹ *sum* (*uGraph-from-list-α-weight* L) (*uGraph-from-list-α-edges* E) = *sum* (λ(*u*,*w*,*v*). *w*) (*set* E)
 ⟨*proof*⟩

lemma *corr*: *uGraph-from-list-invar L* ⟹
<*emp*> *kruskal-algo L*
<λ*F*. ↑ (*uGraph-from-list-invar F* ∧ *set F* ⊆ *set L* ∧
uGraph.minSpanningForest (*uGraph-from-list-α-edges L*)
(*uGraph-from-list-α-weight L*) (*uGraph-from-list-α-edges F*))>$_t$
⟨*proof*⟩

lemma *uGraph-from-list-invar L* ⟹
<*emp*> *kruskal-algo L*
<λ*F*. ↑ (*uGraph-from-list-invar F* ∧ *set F* ⊆ *set L* ∧
uGraph.spanningForest (*uGraph-from-list-α-edges L*) (*uGraph-from-list-α-edges F*)
∧ (∀ *F*′. *uGraph.spanningForest* (*uGraph-from-list-α-edges L*) (*uGraph-from-list-α-edges F*′)
⟶ *set F*′ ⊆ *set L* ⟶ *sum* (λ(*u*,*w*,*v*). *w*) (*set F*) ≤ *sum* (λ(*u*,*w*,*v*). *w*) (*set F*′)))>$_t$
⟨*proof*⟩

6.4 Kruskal with input check

definition *kruskal*′ *L* = *kruskal* (*return L*) (λ(*u*,*w*,*v*). *return* (*u*,*v*)) ()

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

lemma <*emp*> *kruskal-checked L* <λ
Some F ⇒ ↑ (*uGraph-from-list-invar L* ∧ *set F* ⊆ *set L*
∧ *uGraph.minSpanningForest* (*uGraph-from-list-α-edges L*) (*uGraph-from-list-α-weight L*)
(*uGraph-from-list-α-edges F*))
| *None* ⇒ ↑ (¬ *uGraph-from-list-invar L*)>$_t$
⟨*proof*⟩

6.5 Code export

export-code *uGraph-from-list-invar* **checking** *SML-imp*
export-code *kruskal-checked* **checking** *SML-imp*

⟨*ML*⟩

end

7 Undirected Graphs as symmetric directed graphs

theory *Graph-Definition*
imports
Dijkstra-Shortest-Path.Graph
Dijkstra-Shortest-Path.Weight
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*

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*
⟨*proof*⟩

lemma *edges-delete-edge*[*simp*]:
edges (*delete-edge v e v' G*) = *edges G* − {(*v*,*e*,*v'*)}
⟨*proof*⟩

lemma *subgraph-node*:
assumes *subgraph H G*
shows *v* ∈ *nodes G* ⟷ *v* ∈ *nodes H*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *swap-delete-add-edge*:

assumes $(a,\ b,\ c) \neq (x,\ y,\ z)$
shows *delete-edge a b c* (*add-edge x y z H*) = *add-edge x y z* (*delete-edge a b c H*)
⟨*proof*⟩

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*)
⟨*proof*⟩

context *valid-graph*
begin
lemma *valid-subgraph*:
assumes *subgraph H G*
shows *valid-graph H*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *is-path-undir-memb*[*simp*]:
is-path-undir G v p v' ⟹ *v*∈*V* ∧ *v'*∈*V*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *is-path-undir-split*:
is-path-undir G v (*p1*@*p2*) *v'* ⟷ (∃ *u*. *is-path-undir G v p1 u* ∧ *is-path-undir G u p2 v'*)
⟨*proof*⟩

lemma *is-path-undir-split'*[*simp*]:
is-path-undir G v (*p1*@(*u*,*w*,*u'*)#*p2*) *v'*
⟷ *is-path-undir G v p1 u* ∧ ((*u*,*w*,*u'*)∈*E* ∨ (*u'*,*w*,*u*)∈*E*) ∧ *is-path-undir G u' p2 v'*
⟨*proof*⟩

lemma *is-path-undir-sym*:
assumes *is-path-undir G v p v'*
shows *is-path-undir G v'* (*rev* (*map* (λ(*u*, *w*, *u'*). (*u'*, *w*, *u*)) *p*)) *v*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *no-path-in-empty-graph*:
assumes $E = \{\}$
assumes $p \neq []$
shows ¬*is-path-undir G v p v*
⟨*proof*⟩

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 $(\exists p'\ p''\ u\ u'.$
is-path-undir G v p′ u ∧ *is-path-undir G u′ p″ v′* ∧
$length\ p' < length\ p \land length\ p'' < length\ p \land$
$(u \in \{a, b\} \land u' \in \{a, b\}) \land$
$(a, w, b) \notin set\ p' \land (b, w, a) \notin set\ p' \land$
$(a, w, b) \notin set\ p'' \land (b, w, a) \notin set\ p'')$
⟨*proof*⟩

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*⟩

lemma *add-edge-was-path*:
assumes *is-path-undir* (*add-edge a b c G*) *x p y*
assumes $(a, b, c) \notin set\ p$
assumes $(c, b, a) \notin set\ p$
assumes $a \in V$
assumes $c \in V$
shows *is-path-undir G x p y*
⟨*proof*⟩

lemma *delete-edge-is-path*:
assumes *is-path-undir G x p y*
assumes $(a, b, c) \notin set\ p$
assumes $(c, b, a) \notin set\ p$
shows *is-path-undir* (*delete-edge a b c G*) *x p y*
⟨*proof*⟩

lemma *delete-node-is-path*:
assumes *is-path-undir G x p y*
assumes $x \neq v$
assumes $v \notin fst`set\ p \cup snd`snd`set\ p$
shows *is-path-undir* (*delete-node v G*) *x p y*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *subset-was-path*:
assumes *is-path-undir H x p y*
assumes *edges H* $\subseteq$ *E*
assumes *nodes H* $\subseteq$ *V*
shows *is-path-undir G x p y*
⟨*proof*⟩

lemma *delete-node-was-path*:
assumes *is-path-undir* (*delete-node v G*) *x p y*
shows *is-path-undir G x p y*
⟨*proof*⟩

lemma *add-edge-preserve-subgraph*:
assumes *subgraph H G*
assumes $(a, w, b) \in E$
shows *subgraph* (*add-edge a w b H*) *G*
⟨*proof*⟩

lemma *delete-edge-preserve-subgraph*:
assumes *subgraph H G*
shows *subgraph* (*delete-edge a w b H*) *G*
⟨*proof*⟩

lemma *add-delete-edge*:
assumes $(a, w, c) \in E$
shows *add-edge a w c* (*delete-edge a w c G*) = *G*
⟨*proof*⟩

lemma *swap-add-edge-in-path*:
assumes *is-path-undir* (*add-edge a w b G*) $v\ p\ v'$
assumes $(a,w',a') \in E \lor (a',w',a) \in E$
shows $\exists p.$ *is-path-undir* (*add-edge* $a'\ w''\ b\ G$) $v\ p\ v'$
⟨*proof*⟩

lemma *induce-maximally-connected*:
assumes *subgraph H G*
assumes $\forall (a,w,b) \in E.$ *nodes-connected H a b*
shows *maximally-connected H G*
⟨*proof*⟩

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*⟩

lemma *delete-edge-maximally-connected*:
assumes *maximally-connected H G*
assumes *subgraph H G*
assumes *pab*: *is-path-undir* (*delete-edge a w b H*) *a pab b*
shows *maximally-connected* (*delete-edge a w b H*) *G*
⟨*proof*⟩

lemma *connected-impl-maximally-connected*:
assumes *connected-graph H*
assumes *subgraph*: *subgraph H G*
shows *maximally-connected H G*
⟨*proof*⟩

lemma *add-edge-is-connected*:
nodes-connected (*add-edge a b c G*) *a c*
nodes-connected (*add-edge a b c G*) *c a*
⟨*proof*⟩

lemma *swap-edges*:
assumes *nodes-connected* (*add-edge a w b G*) $v\ v'$
assumes $a \in V$
assumes $b \in V$
assumes $\neg$ *nodes-connected* $G\ v\ v'$
shows *nodes-connected* (*add-edge* $v\ w'\ v'\ G$) *a b*
⟨*proof*⟩

lemma *subgraph-impl-connected*:
assumes *connected-graph H*
assumes *subgraph*: *subgraph H G*
shows *connected-graph G*
⟨*proof*⟩

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*⟩
end

context *connected-graph*
begin
lemma *maximally-connected-impl-connected*:
assumes *maximally-connected H G*
assumes *subgraph*: *subgraph H G*
shows *connected-graph H*
⟨*proof*⟩

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 ¬ *nodes-connected H a b*
shows ∃ (*x*, *w*, *y*) ∈ *E* − *edges H*. (¬ *nodes-connected* (*delete-edge x w y G*) *a b*) ∧
(*nodes-connected* (*add-edge a w' b* (*delete-edge x w y G*)) *x y*)
⟨*proof*⟩

lemma *forest-add-edge*:
assumes *a* ∈ *V*
assumes *b* ∈ *V*
assumes ¬ *nodes-connected G a b*
shows *forest* (*add-edge a w b G*)
⟨*proof*⟩

lemma *forest-subsets*:
assumes *valid-graph H*
assumes *edges H* ⊆ *E*
assumes *nodes H* ⊆ *V*
shows *forest H*
⟨*proof*⟩

lemma *subgraph-forest*:
assumes *subgraph H G*
shows *forest H*
⟨*proof*⟩

lemma *forest-delete-edge*: *forest* (*delete-edge a w c G*)
⟨*proof*⟩

lemma *forest-delete-node*: *forest* (*delete-node n G*)
⟨*proof*⟩
end

context *finite-graph*
begin

lemma *finite-subgraphs*: *finite* {*T*. *subgraph T G*}
⟨*proof*⟩

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*
 ⟨*proof*⟩

lemma *minimum-spanning-forest-impl-tree2*:
 assumes *minimum-spanning-forest F G*
 assumes *connected-G*: *connected-graph G*
 shows *minimum-spanning-tree F G*
 ⟨*proof*⟩

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*
 ⟨*proof*⟩

lemma *Domain-nodes-connected*: *Domain* {(*x*, *y*) |*x y*. *nodes-connected G x y*} = *V*
 ⟨*proof*⟩
lemma *Range-nodes-connected*: *Range* {(*x*, *y*) |*x y*. *nodes-connected G x y*} = *V*
 ⟨*proof*⟩
lemma *nodes-connected-insert-per-union*:
 (*nodes-connected* (*add-edge a w b H*) *x y*) ⟷ (*x*,*y*) ∈ *per-union* {(*x*,*y*)| *x y*. *nodes-connected H x y*} *a b*
 if *subgraph H G* **and** *PER*: *part-equiv* {(*x*,*y*)| *x y*. *nodes-connected H x y*}
  and *V*: *a*∈*V* *b*∈*V* **for** *x y*
⟨*proof*⟩

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*
 ⟨*proof*⟩

lemma
 augment-edge:

assumes *sg*: *subgraph G1 G subgraph G2 G* **and**
p: $(u, v) \in \{(a, b) \mid a\ b.\ nodes\text{-}connected\ G1\ a\ b\}$
and *notinE2*: $(u, v) \notin \{(a, b) \mid a\ b.\ nodes\text{-}connected\ G2\ a\ b\}$

shows $\exists\, a\ b\ e.\ (a, b) \notin \{(a, b) \mid a\ b.\ nodes\text{-}connected\ G2\ a\ b\} \wedge e \notin edges\ G2 \wedge e \in edges\ G1 \wedge (case\ e\ of\ (aa, w, ba) \Rightarrow a{=}aa \wedge b{=}ba \vee a{=}ba \wedge b{=}aa)$
⟨*proof*⟩

lemma *nodes-connected-refl*: $a \in V \Longrightarrow nodes\text{-}connected\ G\ a\ a$
⟨*proof*⟩

lemma assumes *sg*: *subgraph H G*
shows *connected-VV*: $\{(x, y) \mid x\ y.\ nodes\text{-}connected\ H\ x\ y\} \subseteq V \times V$
and *connected-refl*: $refl\text{-}on\ V\ \{(x, y) \mid x\ y.\ nodes\text{-}connected\ H\ x\ y\}$
and *connected-trans*: $trans\ \{(x, y) \mid x\ y.\ nodes\text{-}connected\ H\ x\ y\}$
and *connected-sym*: $sym\ \{(x, y) \mid x\ y.\ nodes\text{-}connected\ H\ x\ y\}$
and *connected-equiv*: $equiv\ V\ \{(x, y) \mid x\ y.\ nodes\text{-}connected\ H\ x\ y\}$
⟨*proof*⟩

lemma *forest-maximally-connected-incl-max1*:
assumes
forest H
subgraph H G
shows $(\forall (a,w,b) \in edges\ G - edges\ H.\ \neg\ (forest\ (add\text{-}edge\ a\ w\ b\ H))) \Longrightarrow maximally\text{-}connected\ H\ G$
⟨*proof*⟩

lemma *forest-maximally-connected-incl-max2*:
assumes
forest H
subgraph H G
shows $maximally\text{-}connected\ H\ G \Longrightarrow (\forall (a,w,b) \in E - edges\ H.\ \neg\ (forest\ (add\text{-}edge\ a\ w\ b\ H)))$
⟨*proof*⟩

lemma *forest-maximally-connected-incl-max-conv*:
assumes
forest H
subgraph H G
shows $maximally\text{-}connected\ H\ G = (\forall (a,w,b) \in E - edges\ H.\ \neg\ (forest\ (add\text{-}edge\ a\ w\ b\ H)))$
⟨*proof*⟩

end

end

8 Kruskal on Symmetric Directed Graph

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

8.1 Interpreting *Kruskl-Impl*

locale *fromlist* = **fixes**
L :: (*nat* × *int* × *nat*) *list*
begin

abbreviation $E \equiv$ *set* L
abbreviation $V \equiv$ *fst* ‘ $E \cup$ (*snd* ∘ *snd*) ‘ E
abbreviation *ind* (E'::(*nat* × *int* × *nat*) *set*) ≡ (|*nodes*=V, *edges*=E'|)
abbreviation *subforest* E' ≡ *forest* (*ind* E') ∧ *subgraph* (*ind* E') (*ind* E)

lemma *max-node-is-Max-V*: E = *set la* ⟹ *max-node la* = *Max* (*insert 0* V)
⟨*proof*⟩

lemma *ind-valid-graph*: ⋀E'. $E' \subseteq E$ ⟹ *valid-graph* (*ind* E')
⟨*proof*⟩

lemma *vE*: *valid-graph* (*ind* E) ⟨*proof*⟩

lemma *ind-valid-graph'*: ⋀E'. *subgraph* (*ind* E') (*ind* E) ⟹ *valid-graph* (*ind* E')
⟨*proof*⟩

lemma *add-edge-ind*: $(a,w,b) \in E$ ⟹ *add-edge a w b* (*ind F*) = *ind* (*insert* (a,w,b) F)
⟨*proof*⟩

lemma *nodes-connected-ind-sym*: $F \subseteq E$ ⟹ *sym* {(x, y) |x y. *nodes-connected* (*ind F*) x y}
⟨*proof*⟩
lemma *nodes-connected-ind-trans*: $F \subseteq E$ ⟹ *trans* {(x, y) |x y. *nodes-connected* (*ind F*) x y}
⟨*proof*⟩

lemma *part-equiv-nodes-connected-ind*:
$F \subseteq E$ ⟹ *part-equiv* {(x, y) |x y. *nodes-connected* (*ind F*) x y}
⟨*proof*⟩

sublocale *s*: *Kruskal-Impl E V*

$\lambda e.\ \{fst\ e,\ snd\ (snd\ e)\}\ \lambda u\ v\ (a,w,b).\ u{=}a \wedge v{=}b \vee u{=}b \wedge v{=}a$
subforest
$\lambda E'.\ \{\ (a,b)\ |a\ b.\ nodes\text{-}connected\ (ind\ E')\ a\ b\}$
$\lambda(u,w,v).\ w\ id\ \ PR\text{-}CONST\ (\lambda(u,w,v).\ RETURN\ (u,v))$
$PR\text{-}CONST\ (RETURN\ L)\ return\ L\ set\ L\ (\lambda(u,w,v).\ return\ (u,v))$
⟨*proof*⟩

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*: $s.SpanningForest\ E' = spanning\text{-}forest\ (ind\ E')\ (ind\ E)$
⟨*proof*⟩

lemma *edge-weight-alt*: $edge\text{-}weight\ G = sum\ (\lambda(u,w,v).\ w)\ (edges\ G)$
⟨*proof*⟩

lemma *MSF-eq*: $s.MSF\ E' = minimum\text{-}spanning\text{-}forest\ (ind\ E')\ (ind\ E)$
⟨*proof*⟩

lemma *kruskal-correct*:
<*emp*> $kruskal\ (return\ L)\ (\lambda(u,w,v).\ return\ (u,v))\ ()$
<$\lambda F.\ \uparrow (distinct\ F \wedge set\ F \subseteq E \wedge minimum\text{-}spanning\text{-}forest\ (ind\ (set\ F))\ (ind\ E))$>$_t$
⟨*proof*⟩

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

end

8.3 Outside the locale

definition $GD\text{-}from\text{-}list\text{-}\alpha\text{-}weight\ L\ e = (case\ e\ of\ (u,w,v) \Rightarrow w)$
abbreviation $GD\text{-}from\text{-}list\text{-}\alpha\text{-}graph\ G\ L \equiv (\!|nodes{=}fst\ `\ (set\ G) \cup (snd \circ snd)\ `\ (set\ G),\ edges{=}set\ L|\!)$

lemma *corr*:
<*emp*> *kruskal-algo L*
<$\lambda F.\ \uparrow (set\ F \subseteq set\ L \wedge$
$minimum\text{-}spanning\text{-}forest\ (GD\text{-}from\text{-}list\text{-}\alpha\text{-}graph\ L\ F)\ (GD\text{-}from\text{-}list\text{-}\alpha\text{-}graph\ L\ L))$>$_t$
⟨*proof*⟩

lemma *kruskal-correct*: <*emp*> *kruskal-algo L*

$<\lambda F.\ \uparrow$ (*set* $F \subseteq$ *set* L $\wedge$
spanning-forest (*GD-from-list-α-graph* L F) (*GD-from-list-α-graph* L L)
$\wedge$ ($\forall F'$. *spanning-forest* (*GD-from-list-α-graph* L F') (*GD-from-list-α-graph* L L)
$\longrightarrow$ *sum* ($\lambda(u,w,v)$. w) (*set* F) $\leq$ *sum* ($\lambda(u,w,v)$. w) (*set* F')))$>_t$
⟨*proof*⟩

8.4 Code export

export-code *kruskal-algo* **checking** *SML-imp*

⟨*ML*⟩

end