

Kruskal’s Algorithm for Minimum Spanning Forest

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

December 14, 2021

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	1
1.1	Preparations	1
1.1.1	Weight restricted set	2
1.1.2	The greedy idea	2
1.2	Minimum Weight Basis algorithm	3
1.3	The heart of the argument	4
1.4	The Invariant	4
1.5	Invariant proofs	5
1.6	The refinement lemma	5
2	Kruskal interface	5
2.1	Derived facts	6
2.2	The edge set and forest form the cycle matroid	7
3	Refine Kruskal	8
3.1	Refinement I: cycle check by connectedness	8
3.2	Refinement II: connectedness by PER operation	9

4	Kruskal Implementation	10
4.1	Refinement III: concrete edges	10
4.2	Refinement to Imperative/HOL with Sepref-Tool	12
4.2.1	Refinement IV: given an edge set	12
4.2.2	Synthesis of Kruskal by SepRef	14
5	UGraph - undirected graph with Uprod edges	15
5.1	Edge path	15
5.2	Distinct edge path	17
5.3	Connectivity in undirected Graphs	17
5.4	Forest	18
5.5	uGraph locale	19
6	Kruskal on UGraphs	21
6.1	Interpreting <i>Kruskl-Impl</i> with a UGraph	21
6.2	Kruskal on UGraph from list of concrete edges	22
6.3	Outside the locale	23
6.4	Kruskal with input check	24
6.5	Code export	24
7	Undirected Graphs as symmetric directed graphs	24
7.1	Definition	25
7.2	Helping lemmas	26
7.3	Auxiliary lemmas for graphs	32
8	Kruskal on Symmetric Directed Graph	33
8.1	Interpreting <i>Kruskl-Impl</i>	34
8.2	Showing the equivalence of minimum spanning forest definitions	35
8.3	Outside the locale	35
8.4	Code export	36

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 $B \equiv \text{basis } B \wedge (\forall B'. \text{basis } B' \longrightarrow \text{sum weight } B \leq \text{sum weight } B')$

1.1 Preparations

fun *in-sort-edge* **where**

in-sort-edge $x [] = [x]$

| *in-sort-edge* $x (y\#ys) = (\text{if } \text{weight } x \leq \text{weight } y \text{ then } x\#y\#ys \text{ else } y\# \text{in-sort-edge } x \text{ } ys)$

lemma [*simp*]: $\text{set } (\text{in-sort-edge } x \ L) = \text{insert } x \ (\text{set } L)$ $\langle \text{proof} \rangle$

lemma *in-sort-edge*: $\text{sorted-wrt } (\lambda e1 \ e2. \text{weight } e1 \leq \text{weight } e2) \ L$
 $\implies \text{sorted-wrt } (\lambda e1 \ e2. \text{weight } e1 \leq \text{weight } e2) \ (\text{in-sort-edge } x \ L)$
 $\langle \text{proof} \rangle$

lemma *in-sort-edge-distinct*: $x \notin \text{set } L \implies \text{distinct } L \implies \text{distinct } (\text{in-sort-edge } x \ L)$
 $\langle \text{proof} \rangle$

lemma *finite-sorted-edge-distinct*:

assumes *finite* S

obtains L **where** $\text{distinct } L \ \text{sorted-wrt } (\lambda e1 \ e2. \text{weight } e1 \leq \text{weight } e2) \ L \ S = \text{set } L$

$\langle \text{proof} \rangle$

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

lemma *sum-list-map-cons*:

$\text{sum-list } (\text{map } \text{weight } (y \# \text{ys})) = \text{weight } y + \text{sum-list } (\text{map } \text{weight } \text{ys})$

$\langle \text{proof} \rangle$

lemma *exists-greater*:

assumes *len*: $\text{length } F = \text{length } F'$

and *sum*: $\text{sum-list } (\text{map } \text{weight } F) > \text{sum-list } (\text{map } \text{weight } F')$

shows $\exists i < \text{length } F. \text{weight } (F ! i) > \text{weight } (F' ! i)$

$\langle \text{proof} \rangle$

lemma *wsorted-nth-mono*: **assumes** *wsorted* $L \ i \leq j < \text{length } L$

shows $\text{weight } (L ! i) \leq \text{weight } (L ! j)$

$\langle \text{proof} \rangle$

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

lemma *limi-subset*: $\text{limi } T \ g \subseteq T$ $\langle \text{proof} \rangle$

lemma *limi-mono*: $A \subseteq B \implies \text{limi } A \ g \subseteq \text{limi } B \ g$ $\langle \text{proof} \rangle$

1.1.2 The greedy idea

definition *no-smallest-element-skipped* $E F$

$$= (\forall e \in \text{carrier} - E. \forall g > \text{weight } e. \text{indep} (\text{insert } e (\text{limi } F g)) \longrightarrow (e \in \text{limi } F g))$$

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.

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* $E F$ $e \in \text{carrier} - E$
 $\text{weight } e < g$ (*indep* (*insert* e (*limi* $F g$)))

shows $e \in \text{limi } F g$

<proof>

lemma *no-smallest-element-skipped-skip*:

assumes *createsCycle*: $\neg \text{indep} (\text{insert } e F)$

and I : *no-smallest-element-skipped* $(E \cup \{e\}) F$

and *sorted*: $(\forall x \in F. \forall y \in (E \cup \{e\}). \text{weight } x \leq \text{weight } y)$

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* $\equiv \text{SPEC } (\lambda L. \text{wsorted } L \wedge \text{set } L = \text{carrier})$

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 dependent.

definition *minWeightBasis* **where**

minWeightBasis $\equiv \text{do } \{$
 $l \leftarrow \text{obtain-sorted-carrier};$
 $\text{ASSERT } (\text{set } l = \text{carrier});$
 $T \leftarrow \text{nfoldli } l (\lambda-. \text{True})$
 $(\lambda e T. \text{do } \{$

```

    ASSERT (indep T  $\wedge$   $e \in \text{carrier} \wedge T \subseteq \text{carrier}$ );
    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*: $\forall e \in \text{carrier} - F. \neg \text{indep} (\text{insert } e F)$
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**
 $I\text{-minWeightBasis} == \lambda(T, E). \text{indep } T$
 $\wedge T \subseteq \text{carrier}$
 $\wedge E \subseteq \text{carrier}$
 $\wedge (\forall x \in T. \forall y \in E. \text{weight } x \leq \text{weight } y)$
 $\wedge (\forall e \in \text{carrier} - E - T. \sim \text{indep} (\text{insert } e T))$
 $\wedge \text{no-smallest-element-skipped } E T$

lemma *I-minWeightBasisD*:
assumes
 $I\text{-minWeightBasis } (T, E)$
shows $\text{indep } T \wedge e. e \in \text{carrier} - E - T \implies \sim \text{indep} (\text{insert } e T)$
 $E \subseteq \text{carrier} \wedge x y. x \in T \implies y \in E \implies \text{weight } x \leq \text{weight } y \quad T \subseteq \text{carrier}$
 $\text{no-smallest-element-skipped } E T$
 ⟨*proof*⟩

lemma *I-minWeightBasisI*:
assumes $\text{indep } T \wedge e. e \in \text{carrier} - E - T \implies \sim \text{indep} (\text{insert } e T)$
 $E \subseteq \text{carrier} \wedge x y. x \in T \implies y \in E \implies \text{weight } x \leq \text{weight } y \quad T \subseteq \text{carrier}$
 $\text{no-smallest-element-skipped } E T$
shows $I\text{-minWeightBasis } (T, E)$
 ⟨*proof*⟩

lemma *I-minWeightBasisG*: $I\text{-minWeightBasis } (T, E) \implies \text{no-smallest-element-skipped } E\ T$

<proof>

lemma *I-minWeightBasis-sorted*: $I\text{-minWeightBasis } (T, E) \implies (\forall x \in T. \forall y \in E. \text{weight } x \leq \text{weight } y)$

<proof>

1.5 Invariant proofs

lemma *I-minWeightBasis-empty*: $I\text{-minWeightBasis } (\{\}, \text{carrier})$

<proof>

lemma *I-minWeightBasis-final*: $I\text{-minWeightBasis } (T, \{\}) \implies \text{minBasis } T$

<proof>

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 $\neg \text{indep } (\text{insert } x\ (\text{insert } e\ F))$

<proof>

lemma *preservation-if*: $\text{wsorted } x \implies \text{set } x = \text{carrier} \implies$

$x = l1 @ xa \# l2 \implies I\text{-minWeightBasis } (\sigma, \text{set } (xa \# l2)) \implies \text{indep } \sigma$

$\implies xa \in \text{carrier} \implies \text{indep } (\text{insert } xa\ \sigma) \implies I\text{-minWeightBasis } (\text{insert } xa\ \sigma, \text{set } l2)$

<proof>

lemma *preservation-else*: $\text{set } x = \text{carrier} \implies$

$x = l1 @ xa \# l2 \implies I\text{-minWeightBasis } (\sigma, \text{set } (xa \# l2))$

$\implies \text{indep } \sigma \implies \neg \text{indep } (\text{insert } xa\ \sigma) \implies I\text{-minWeightBasis } (\sigma, \text{set } l2)$

<proof>

1.6 The refinement lemma

theorem *minWeightBasis-refine*: $(\text{minWeightBasis}, \text{SPEC } \text{minBasis}) \in \langle \text{Id} \rangle \text{nres-rel}$

<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  $\Rightarrow$  'a set
    and joins :: 'a  $\Rightarrow$  'a  $\Rightarrow$  'edge  $\Rightarrow$  bool
    and forest :: 'edge set  $\Rightarrow$  bool
    and connected :: 'edge set  $\Rightarrow$  ('a*'a) set
    and weight :: 'edge  $\Rightarrow$  'b::{linorder, ordered-comm-monoid-add}
assumes
  finiteE[simp]: finite E
  and forest-subE: forest E'  $\Longrightarrow$  E'  $\subseteq$  E
  and forest-empty: forest {}
  and forest-mono: forest X  $\Longrightarrow$  Y  $\subseteq$  X  $\Longrightarrow$  forest Y
  and connected-same: (u,v)  $\in$  connected {}  $\longleftrightarrow$  u=v  $\wedge$  v  $\in$  V
  and findaugmenting-aux: E1  $\subseteq$  E  $\Longrightarrow$  E2  $\subseteq$  E  $\Longrightarrow$  (u,v)  $\in$  connected E1  $\Longrightarrow$ 
(u,v)  $\notin$  connected E2
     $\Longrightarrow$   $\exists$  a b e. (a,b)  $\notin$  connected E2  $\wedge$  e  $\notin$  E2  $\wedge$  e  $\in$  E1  $\wedge$  joins a b e
  and augment-forest: forest F  $\Longrightarrow$  e  $\in$  E-F  $\Longrightarrow$  joins u v e
     $\Longrightarrow$  forest (insert e F)  $\longleftrightarrow$  (u,v)  $\notin$  connected F
  and equiv: F  $\subseteq$  E  $\Longrightarrow$  equiv V (connected F)
  and connected-in: F  $\subseteq$  E  $\Longrightarrow$  connected F  $\subseteq$  V  $\times$  V
  and insert-reachable: x  $\in$  V  $\Longrightarrow$  y  $\in$  V  $\Longrightarrow$  F  $\subseteq$  E  $\Longrightarrow$  e  $\in$  E  $\Longrightarrow$  joins x y e
     $\Longrightarrow$  connected (insert e F) = per-union (connected F) x y
  and exhaust:  $\bigwedge$  x. x  $\in$  E  $\Longrightarrow$   $\exists$  a b. joins a b x
  and vertices-constr:  $\bigwedge$  a b e. joins a b e  $\Longrightarrow$  {a,b}  $\subseteq$  vertices e
  and joins-sym:  $\bigwedge$  a b e. joins a b e = joins b a e
  and selfloop-no-forest:  $\bigwedge$  e. e  $\in$  E  $\Longrightarrow$  joins a a e  $\Longrightarrow$   $\sim$ forest (insert e F)
  and finite-vertices:  $\bigwedge$  e. e  $\in$  E  $\Longrightarrow$  finite (vertices e)

  and edgesinvertices:  $\bigcup$  (vertices ' E)  $\subseteq$  V
  and finiteV[simp]: finite V
  and joins-connected: joins a b e  $\Longrightarrow$  T  $\subseteq$  E  $\Longrightarrow$  e  $\in$  T  $\Longrightarrow$  (a,b)  $\in$  connected T

```

begin

2.1 Derived facts

lemma joins-in-V: joins a b e \Longrightarrow e \in E \Longrightarrow a \in V \wedge b \in V
 <proof>

lemma *finiteE-finiteV*: $\text{finite } E \implies \text{finite } V$
<proof>

lemma *E-inV*: $\bigwedge e. e \in E \implies \text{vertices } e \subseteq V$
<proof>

definition *CC E' x* = $(\text{connected } E')^{\{x\}}$

lemma *sameCC-reachable*: $E' \subseteq E \implies u \in V \implies v \in V \implies \text{CC } E' u = \text{CC } E' v$
 $\iff (u,v) \in \text{connected } E'$
<proof>

definition *CCs E'* = $\text{quotient } V (\text{connected } E')$

lemma *quotient V Id* = $\{\{v\} | v. v \in V\}$ *<proof>*

lemma *CCs-empty*: $\text{CCs } \{\} = \{\{v\} | v. v \in V\}$
<proof>

lemma *CCs-empty-card*: $\text{card } (\text{CCs } \{\}) = \text{card } V$
<proof>

lemma *CCs-imageCC*: $\text{CCs } F = (\text{CC } F)^{\cdot} V$
<proof>

lemma *union-eqclass-decreases-components*:

assumes $\text{CC } F x \neq \text{CC } F y \implies e \notin F \implies x \in V \implies y \in V \implies F \subseteq E \implies e \in E \text{ joins } x \ y$

shows $\text{Suc } (\text{card } (\text{CCs } (\text{insert } e \ F))) = \text{card } (\text{CCs } F)$

<proof>

lemma *forest-CCs*: **assumes** *forest E'* **shows** $\text{card } (\text{CCs } E') + \text{card } E' = \text{card } V$
<proof>

lemma *pigeonhole-CCs*:

assumes *finiteV*: $\text{finite } V$ **and** *cardlt*: $\text{card } (\text{CCs } E1) < \text{card } (\text{CCs } E2)$

shows $(\exists u \ v. u \in V \wedge v \in V \wedge \text{CC } E1 \ u = \text{CC } E1 \ v \wedge \text{CC } E2 \ u \neq \text{CC } E2 \ v)$

<proof>

2.2 The edge set and forest form the cycle matroid

theorem **assumes** *f1*: *forest E1*

and *f2*: *forest E2*

and *c*: $\text{card } E1 > \text{card } E2$

shows *augment*: $\exists e \in E1 - E2. \text{forest } (\text{insert } e \ E2)$

<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* $\equiv \{\}$

abbreviation *a-endpoints* $e \equiv SPEC (\lambda(a,b). joins\ a\ b\ e)$

definition *kruskal0*

```
where kruskal0  $\equiv do \{$   
   $l \leftarrow obtain\ sorted\ carrier;$   
   $spanning\ forest \leftarrow nfoldli\ l\ (\lambda-. True)$   
   $(\lambda e\ T. do \{$   
     $ASSERT\ (e \in E);$   
     $(a,b) \leftarrow a\ endpoints\ e;$   
     $ASSERT\ (joins\ a\ b\ e \wedge forest\ T \wedge e \in E \wedge T \subseteq E);$   
     $if\ \neg\ (a,b) \in connected\ T\ then$   
       $do \{$   
         $ASSERT\ (e \notin T);$   
         $RETURN\ (insert\ e\ T)$   
       $\}$   
     $else$   
       $RETURN\ T$   
   $\})\ empty\ forest;$   
   $RETURN\ spanning\ forest$   
 $\}$ 
```

lemma *if-subst*: $(if\ indep\ (insert\ e\ T)\ then$
 $RETURN\ (insert\ e\ T)$
 $else$
 $RETURN\ T)$

$$= (\text{if } e \notin T \wedge \text{indep } (\text{insert } e \ T) \text{ then}$$

$$\quad \text{RETURN } (\text{insert } e \ T)$$

$$\quad \text{else}$$

$$\quad \text{RETURN } T)$$
 <proof>

lemma *kruskal0-refine*: $(\text{kruskal0}, \text{minWeightBasis}) \in \langle \text{Id} \rangle \text{nres-rel}$
 <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 \Rightarrow 'edge set \Rightarrow bool **where**
corresponding-union-find uf T $\equiv (\forall a \in V. \forall b \in V. \text{per-compare } uf \ a \ b \longleftrightarrow ((a,b) \in \text{connected } T))$

definition *uf-graph-invar* uf-T
 $\equiv \text{case } uf\text{-}T \text{ of } (uf, T) \Rightarrow \text{corresponding-union-find } uf \ T \wedge \text{Domain } uf = V$

lemma *uf-graph-invarD*: $uf\text{-}graph\text{-}invar \ (uf, T) \Longrightarrow \text{corresponding-union-find } uf \ T$
 <proof>

definition *uf-graph-rel* $\equiv \text{br snd } uf\text{-}graph\text{-}invar$

lemma *uf-graph-relsndD*: $((a,b),c) \in uf\text{-}graph\text{-}rel \Longrightarrow b=c$
 <proof>

lemma *uf-graph-relD*: $((a,b),c) \in uf\text{-}graph\text{-}rel \Longrightarrow b=c \wedge uf\text{-}graph\text{-}invar \ (a,b)$
 <proof>

definition *kruskal1*

where *kruskal1* $\equiv \text{do } \{$
 $l \leftarrow \text{obtain-sorted-carrier};$
 $\text{let } \text{initial-union-find} = \text{per-init } V;$
 $(\text{per}, \text{spanning-forest}) \leftarrow \text{nfoldli } l \ (\lambda_. \text{True})$
 $(\lambda e \ (uf, T). \text{do } \{$
 $\quad \text{ASSERT } (e \in E);$
 $\quad (a,b) \leftarrow \text{a-endpoints } e;$
 $\quad \text{ASSERT } (a \in V \wedge b \in V \wedge a \in \text{Domain } uf \wedge b \in \text{Domain } uf \wedge T \subseteq E);$
 $\quad \text{if } \neg \text{per-compare } uf \ a \ b \text{ then}$
 $\quad \text{do } \{$
 $\quad \quad \text{let } uf = \text{per-union } uf \ a \ b;$
 $\quad \quad \text{ASSERT } (e \notin T);$
 $\quad \quad \text{RETURN } (uf, \text{insert } e \ T)$
 $\quad \}$
 $\}$

```

      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]*: $wsorted' s \implies wsorted (map \alpha s)$
<proof>

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

abbreviation *concrete-edge-rel* $:: ('cedge \times 'edge) set$ **where**
concrete-edge-rel $\equiv br \alpha (\lambda-. True)$

lemma *obtain-sorted-carrier'-refine*:
 $(obtain-sorted-carrier', obtain-sorted-carrier) \in \langle \langle concrete-edge-rel \rangle list-rel \rangle nres-rel$
<proof>

definition *kruskal2*

where *kruskal2* $\equiv do \{$
 $l \leftarrow obtain-sorted-carrier'$;
 $let initial-union-find = per-init V$;
 $(per, spanning-forest) \leftarrow nfoldli l (\lambda-. True)$
 $(\lambda ce (uf, T). do \{$
 $ASSERT (\alpha ce \in E)$;
 $(a,b) \leftarrow endpoints ce$;
 $ASSERT (a \in V \wedge b \in V \wedge a \in Domain uf \wedge b \in Domain uf)$;
 $if \neg per-compare uf a b then$
 $do \{$
 $let uf = per-union uf a b$;
 $ASSERT (ce \notin set T)$;
 $RETURN (uf, T@[ce])$
 $\}$
 $else$
 $RETURN (uf, T)$
 $\}) (initial-union-find, [])$;
 $RETURN spanning-forest$
 $\}$

lemma *lst-graph-rel-empty[simp]*: $([], \{\}) \in \langle concrete-edge-rel \rangle list-set-rel$
<proof>

lemma *loop-initial-rel*:
 $((per-init V, []), per-init V, \{\}) \in Id \times_r \langle concrete-edge-rel \rangle list-set-rel$
<proof>

lemma *concrete-edge-rel-list-set-rel*:
 $(a, b) \in \langle concrete-edge-rel \rangle list-set-rel \implies \alpha ' (set a) = b$
<proof>

theorem *kruskal2-refine*: $(kruskal2, kruskal1) \in \langle \langle concrete-edge-rel \rangle list-set-rel \rangle nres-rel$
<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 ($\lambda L. \alpha$ ' set $L = E$

$\wedge (\forall (a, wv, b) \in \text{set } L. \text{weight } (\alpha (a, wv, b)) = wv) \wedge \text{set } L \subseteq$

superE)

and

getEdges-impl: (uncurry0 *getEdges-impl*, uncurry0 *getEdges*)

$\in \text{unit-assn}^k \rightarrow_a \text{list-assn } (\text{nat-assn } \times_a \text{int-assn } \times_a \text{nat-assn})$

and

max-node-is-Max-V: $E = \alpha$ ' set $la \implies \text{max-node } la = \text{Max } (\text{insert } 0 V)$

and

endpoints-impl: (*endpoints-impl*, *endpoints*)

$\in (\text{nat-assn } \times_a \text{int-assn } \times_a \text{nat-assn})^k \rightarrow_a (\text{nat-assn } \times_a \text{nat-assn})$

begin

lemma *this-loc*: *Kruskal-Impl* *E V vertices joins forest connected weight*

α *endpoints* *getEdges* *getEdges-impl* *superE* *endpoints-impl* <proof>

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 {

$l \leftarrow \text{SPEC } (\lambda L. \alpha$ ' set $L = E$

$\wedge (\forall (a, wv, b) \in \text{set } L. \text{weight } (\alpha (a, wv, b)) = wv) \wedge \text{set } L \subseteq$

superE);

SPEC ($\lambda L. \text{sorted-wrt edges-less-eq } L \wedge \text{set } L = \text{set } l$)

}

lemma *wsorted'-sorted-wrt-edges-less-eq*:

assumes $\forall (a, wv, b) \in \text{set } s. \text{weight } (\alpha (a, wv, b)) = wv$

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* ($\lambda(l,n). n = \text{Max} (\text{insert } 0 V)$)

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* ($\lambda-. \text{True}$)
 ($\lambda ce (uf, T). \text{do}$ {
 ASSERT ($\alpha ce \in E$);
 (*a,b*) ← *endpoints ce*;
 ASSERT ($a \in \text{Domain } uf \wedge b \in \text{Domain } uf$);
 if $\neg \text{per-compare } uf \ a \ b$ then
 do {
 let *uf* = *per-union uf a b*;
 ASSERT ($ce \notin \text{set } T$);
 RETURN (*uf, T@[ce]*)
 }
 }
 else
 RETURN (*uf, T*)
 }) (*initial-union-find*, []);
 RETURN *spanning-forest*
 }

lemma *endpoints-spec*: *endpoints ce* ≤ *SPEC* ($\lambda-. \text{True}$)
 ⟨*proof*⟩

lemma *kruskal3-subset*:
shows *kruskal3* ≤_{*n*} *SPEC* ($\lambda T. \text{distinct } T \wedge \text{set } T \subseteq \text{superE}$)
 ⟨*proof*⟩

definition *per-supset-rel* :: ('a per × 'a per) set **where**
per-supset-rel
 $\equiv \{(p1, p2). p1 \cap \text{Domain } p2 \times \text{Domain } p2 = p2 \wedge p1 - (\text{Domain } p2 \times \text{Domain } p2) \subseteq \text{Id}\}$

lemma *per-supset-rel-dom*: $(p1, p2) \in \text{per-supset-rel} \implies \text{Domain } p1 \supseteq \text{Domain } p2$
 ⟨proof⟩

lemma *per-supset-compare*:
 $(p1, p2) \in \text{per-supset-rel} \implies x1 \in \text{Domain } p2 \implies x2 \in \text{Domain } p2$
 $\implies \text{per-compare } p1 \ x1 \ x2 \longleftrightarrow \text{per-compare } p2 \ x1 \ x2$
 ⟨proof⟩

lemma *per-supset-union*: $(p1, p2) \in \text{per-supset-rel} \implies x1 \in \text{Domain } p2 \implies x2 \in \text{Domain } p2 \implies$
 $(\text{per-union } p1 \ x1 \ x2, \text{per-union } p2 \ x1 \ x2) \in \text{per-supset-rel}$
 ⟨proof⟩

lemma *per-initN-refine*: $(\text{per-init}' (\text{Max } (\text{insert } 0 \ V) + 1), \text{per-init } V) \in \text{per-supset-rel}$
 ⟨proof⟩

theorem *kruskal3-refine*: $(\text{kruskal3}, \text{kruskal2}) \in \langle \text{Id} \rangle \text{nres-rel}$
 ⟨proof⟩

4.2.2 Synthesis of Kruskal by SepRef

lemma [*sepref-import-param*]: $(\text{sort-edges}, \text{sort-edges}) \in \langle \text{Id} \times_r \text{Id} \times_r \text{Id} \rangle \text{list-rel} \rightarrow \langle \text{Id} \times_r \text{Id} \times_r \text{Id} \rangle \text{list-rel}$
 ⟨proof⟩

lemma [*sepref-import-param*]: $(\text{max-node}, \text{max-node}) \in \langle \text{Id} \times_r \text{Id} \times_r \text{Id} \rangle \text{list-rel} \rightarrow \text{nat-rel}$ ⟨proof⟩

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

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

schematic-goal *kruskal-impl*:
 $(\text{uncurry0 } ?c, \text{uncurry0 } \text{kruskal3}) \in (\text{unit-assn})^k \rightarrow_a \text{list-assn } (\text{nat-assn} \times_a \text{int-assn} \times_a \text{nat-assn})$
 ⟨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 $kruskal3-ref-spec'$:
 $(uncurry0 kruskal3, uncurry0 (SPEC (\lambda r. MSF (\alpha ' set r)))) \in unit-rel \rightarrow_f$
 $\langle Id \rangle nres-rel$
 $\langle proof \rangle$

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

lemma $[fcomp-norm-simps]: list-assn (nat-assn \times_a int-assn \times_a nat-assn) =$
 $id-assn$
 $\langle proof \rangle$

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

The final correctness lemma for Kruskal's algorithm.

lemma $kruskal-correct-forest$:
shows $\langle emp \rangle kruskal getEdges-impl endpoints-impl ()$
 $\langle \lambda r. \uparrow (distinct r \wedge set r \subseteq superE \wedge MSF (set (map \alpha r))) \rangle_t$
 $\langle proof \rangle$

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 ($\exists w. u \neq w \wedge \text{Upair } u \ w = x \wedge \text{epath } E \ w \ xs \ v$) $\wedge x \in E$

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 ($\exists w. \text{epath } E \ u \ p \ w \ \wedge \text{epath } E \ w \ q \ v$)
<proof>

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

lemma *epath* *E* *x* *p* *y* \Longrightarrow $\exists p. \text{epath } E \ y \ p \ x$
<proof>

lemma *epath-mono*: $E \subseteq E' \Longrightarrow \text{epath } E \ u \ p \ v \Longrightarrow \text{epath } E' \ u \ p \ v$
<proof>

lemma *epath-restrict*: set *p* $\subseteq I \Longrightarrow \text{epath } E \ u \ p \ v \Longrightarrow \text{epath } (E \cap I) \ u \ p \ v$
<proof>

lemma **assumes** $A \subseteq A' \sim \text{epath } A \ u \ p \ v \ \text{epath } A' \ u \ p \ v$
shows *epath-diff-edge*: ($\exists e. e \in \text{set } p - A$)
<proof>

lemma *epath-restrict'*: *epath* (*insert* *e* *E*) *u* *p* *v* $\Longrightarrow e \notin \text{set } p \Longrightarrow \text{epath } E \ u \ p \ v$
<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* ≥ 2
<proof>

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

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

lemma *epath-split-distinct*:
assumes $\text{epath } G \ v \ p \ v'$
assumes $\text{Upair } a \ b \in \text{set } p$
shows $(\exists p' \ p'' \ u \ u' .$
 $\text{epath } G \ v \ p' \ u \wedge \text{epath } G \ u' \ p'' \ v' \wedge$
 $\text{length } p' < \text{length } p \wedge \text{length } p'' < \text{length } p \wedge$
 $(u \in \{a, b\} \wedge u' \in \{a, b\}) \wedge$
 $\text{Upair } a \ b \notin \text{set } p' \wedge \text{Upair } a \ b \notin \text{set } p'')$
 $\langle \text{proof} \rangle$

5.2 Distinct edge path

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

lemma *epath-to-depath*: $\text{set } p \subseteq I \implies \text{epath } E \ u \ p \ v \implies \exists dp. \text{depath } E \ u \ dp \ v \wedge \text{set } dp \subseteq I$
 $\langle \text{proof} \rangle$

lemma *epath-to-depath'*: $\text{epath } E \ u \ p \ v \implies \exists dp. \text{depath } E \ u \ dp \ v$
 $\langle \text{proof} \rangle$

definition $\text{decycle } E \ u \ p \equiv \text{epath } E \ u \ p \ u \wedge \text{length } p > 2 \wedge \text{distinct } p$

5.3 Connectivity in undirected Graphs

definition $\text{unconnected } E \equiv \{(u,v). \exists p. \text{epath } E \ u \ p \ v\}$

lemma *unconnectedempty*: $\text{unconnected } \{ \} = \{(a,a) \mid a. \text{True}\}$
 $\langle \text{proof} \rangle$

lemma *unconnected-refl*: $\text{refl } (\text{unconnected } E)$
 $\langle \text{proof} \rangle$

lemma *unconnected-sym*: $\text{sym } (\text{unconnected } E)$
 $\langle \text{proof} \rangle$

lemma *unconnected-trans*: $\text{trans } (\text{unconnected } E)$
 $\langle \text{proof} \rangle$

lemma *unconnected-symI*: $(u,v) \in \text{unconnected } E \implies (v,u) \in \text{unconnected } E$
 $\langle \text{proof} \rangle$

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*: $A \subseteq A' \implies \text{uconnected } A \subseteq \text{uconnected } A'$

<proof>

lemma *findaugmenting-edge*: **assumes** *epath E1 u p v*

and $\neg(\exists p. \text{epath } E2 \ u \ p \ v)$

shows $\exists a \ b. (a, b) \notin \text{uconnected } E2 \wedge \text{Upair } a \ b \notin E2 \wedge \text{Upair } a \ b \in E1$

<proof>

5.4 Forest

definition *forest E* $\equiv \sim(\exists u \ p. \text{decycle } E \ u \ p)$

lemma *forest-mono*: $Y \subseteq X \implies \text{forest } X \implies \text{forest } Y$

<proof>

lemma *forrest2-E*: **assumes** $(u, v) \in \text{uconnected } E$

and $\text{Upair } u \ v \notin E$

and $u \neq v$

shows $\sim \text{forest } (\text{insert } (\text{Upair } u \ v) \ E)$

<proof>

lemma *insert-stays-forest-means-not-connected*: **assumes** *forest (insert (Upair u v) E)*

and $(\text{Upair } u \ v) \notin E$

and $u \neq v$

shows $\sim (u, v) \in \text{uconnected } E$

<proof>

lemma *epath-singleton*: *epath F a [e] b* $\implies e = \text{Upair } a \ b$

<proof>

lemma *forest-alt1*:

assumes $\text{Upair } a \ b \in F \ \text{forest } F \ \bigwedge e. e \in F \implies \text{proper-uprod } e$

shows $(a, b) \notin \text{uconnected } (F - \{\text{Upair } a \ b\})$

<proof>

lemma *forest-alt2*:

assumes $\bigwedge e. e \in F \implies \text{proper-uprod } e$
and $\bigwedge a b. \text{Upair } a b \in F \implies (a,b) \notin \text{unconnected } (F - \{\text{Upair } a b\})$
shows *forest* F
<proof>

lemma *forest-alt*:

assumes $\bigwedge e. e \in F \implies \text{proper-uprod } e$
shows *forest* $F \iff (\forall a b. \text{Upair } a b \in F \longrightarrow (a,b) \notin \text{unconnected } (F - \{\text{Upair } a b\}))$
<proof>

lemma *augment-forest-overedges*:

assumes $F \subseteq E$ *forest* F $(\text{Upair } u v) \in E$ $(u,v) \notin \text{unconnected } F$
and *notsame*: $u \neq v$
shows *forest* $(\text{insert } (\text{Upair } u v) F)$
<proof>

5.5 uGraph locale

locale *uGraph* =

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

abbreviation *unconnected-on* $E' V \equiv \text{unconnected } E' \cap (V \times V)$

abbreviation *verts* $\equiv \bigcup (\text{set-uprod } ' E)$

lemma *set-uprod-nonempty* $Y[\text{simp}]$: *set-uprod* $x \neq \{\}$ *<proof>*

abbreviation *unconnectedV* $E' \equiv \text{Restr } (\text{unconnected } E') \text{ verts}$

lemma *equiv-unconnected-on*: *equiv* V $(\text{unconnected-on } E' V)$
<proof>

lemma *unconnectedV-refl*: $E' \subseteq E \implies \text{refl-on } \text{verts } (\text{unconnectedV } E')$
<proof>

lemma *unconnectedV-trans*: *trans* $(\text{unconnectedV } E')$
<proof>

lemma *uconnectedV-sym*: $\text{sym } (\text{uconnectedV } E')$
<proof>

lemma *equiv-vert-uconnected*: $\text{equiv verts } (\text{uconnectedV } E')$
<proof>

lemma *uconnectedV-tracl*: $(\text{uconnectedV } F)^* = (\text{uconnectedV } F)^=$
<proof>

lemma *uconnectedV-cl*: $(\text{uconnectedV } F)^+ = (\text{uconnectedV } F)$
<proof>

lemma *uconnectedV-Restrcl*: $\text{Restr } ((\text{uconnectedV } F)^*) \text{ verts} = (\text{uconnectedV } F)$
<proof>

lemma *restr-ucon*: $F \subseteq E \implies \text{uconnected } F = \text{uconnectedV } F \cup \text{Id}$
<proof>

lemma *relI*:
assumes $\bigwedge a b. (a,b) \in F \implies (a,b) \in G$
and $\bigwedge a b. (a,b) \in G \implies (a,b) \in F$ **shows** $F=G$
<proof>

lemma *in-per-union*: $u \in \{x, y\} \implies u' \in \{x, y\} \implies x \in V \implies y \in V \implies$
 $\text{refl-on } V \ R \implies \text{part-equiv } R \implies (u, u') \in \text{per-union } R \ x \ y$
<proof>

lemma *uconnectedV-mono*: $(a,b) \in \text{uconnectedV } F \implies F \subseteq F' \implies (a,b) \in \text{uconnectedV } F'$
<proof>

lemma *per-union-subs*: $x \in S \implies y \in S \implies R \subseteq S \times S \implies \text{per-union } R \ x \ y \subseteq S \times S$
<proof>

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

lemma *epath-filter-selfloop*: $\text{epath } (\text{insert } (\text{Upair } x \ x) \ F) \ a \ p \ b \implies \exists p. \text{epath } F \ a \ p \ b$
<proof>

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

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

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

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

definition *subforest* $F \equiv \text{forest } F \wedge F \subseteq E$

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

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

end

end

6 Kruskal on UGraphs

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

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

6.1 Interpreting *Kruskal-Impl* with a UGraph

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

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

$int +$
fixes $getEdges-impl :: (nat \times int \times nat) list Heap$ **and** $csuper-E :: (nat \times int \times nat) set$
assumes $getEdges-impl:$
 $(uncurry0\ getEdges-impl, uncurry0\ (getEdges-SPEC\ csuper-E))$
 $\in unit-assn^k \rightarrow_a list-assn (nat-assn \times_a int-assn \times_a nat-assn)$
begin

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

lemma $max-node-is-Max-V: E = \alpha\ 'set\ la \implies max-node\ la = Max\ (insert\ 0\ V)$
 $\langle proof \rangle$

sublocale $s: Kruskal-Impl\ E \bigcup (set-uprod\ 'E) set-uprod\ \lambda u\ v\ e. Upair\ u\ v = e$
 $subforest\ uconnectedV\ w\ \alpha\ PR-CONST\ (\lambda(u,w,v). RETURN\ (u,v))$
 $PR-CONST\ (getEdges-SPEC\ csuper-E)$
 $getEdges-impl\ csuper-E\ (\lambda(u,w,v). return\ (u,v))$
 $\langle proof \rangle$

lemma $spanningForest-eq-basis: spanningForest = s.basis$
 $\langle proof \rangle$

lemma $minSpanningForest-eq-minbasis: minSpanningForest = s.minBasis$
 $\langle proof \rangle$

lemma $kruskal-correct':$
 $\langle emp \rangle kruskal\ getEdges-impl\ (\lambda(u,w,v). return\ (u,v))\ ()$
 $\langle \lambda r. \uparrow (distinct\ r \wedge set\ r \subseteq csuper-E \wedge s.MSF\ (set\ (map\ \alpha\ r))) \rangle_t$
 $\langle proof \rangle$

lemma $kruskal-correct:$
 $\langle emp \rangle kruskal\ getEdges-impl\ (\lambda(u,w,v). return\ (u,v))\ ()$
 $\langle \lambda r. \uparrow (distinct\ r \wedge set\ r \subseteq csuper-E \wedge minSpanningForest\ (set\ (map\ \alpha\ r))) \rangle_t$
 $\langle proof \rangle$

end

6.2 Kruskal on UGraph from list of concrete edges

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

abbreviation $uGraph-from-list-\alpha-edges\ L \equiv \alpha\ 'set\ L$

locale *fromlist* = **fixes**

$L :: (\text{nat} \times \text{int} \times \text{nat}) \text{ list}$

assumes *dist*: $\text{distinct} (\text{map } \alpha L)$ **and** *no-selfloop*: $\forall u w v. (u,w,v) \in \text{set } L \longrightarrow u \neq v$
begin

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

<proof>

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

<proof>

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

<proof>

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

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

end

6.3 Outside the locale

definition *uGraph-from-list-invar* $:: (\text{nat} \times \text{int} \times \text{nat}) \text{ list} \Rightarrow \text{bool}$ **where**

$\text{uGraph-from-list-invar } L = (\text{distinct} (\text{map } \alpha L) \wedge (\forall p \in \text{set } L. \text{case } p \text{ of } (u,w,v) \Rightarrow u \neq v))$

lemma *uGraph-from-list-invar-conv*: $\text{uGraph-from-list-invar } L = \text{fromlist } L$

<proof>

lemma *uGraph-from-list-invar-subset*:

$\text{uGraph-from-list-invar } L \implies \text{set } L' \subseteq \text{set } L \implies \text{distinct } L' \implies \text{uGraph-from-list-invar } L'$

<proof>

lemma *uGraph-from-list-}\alpha\text{-inj-on}*: $\text{uGraph-from-list-invar } E \implies \text{inj-on } \alpha (\text{set } E)$

<proof>

lemma *sum-easier*: $\text{uGraph-from-list-invar } L$

$\implies \text{set } E \subseteq \text{set } L$

$\implies \text{sum} (\text{uGraph-from-list-}\alpha\text{-weight } L) (\text{uGraph-from-list-}\alpha\text{-edges } E) = \text{sum} (\lambda(u,w,v). w) (\text{set } E)$

<proof>

lemma *corr*: $uGraph\text{-from-list-invar } L \implies$
 $\langle emp \rangle$ *kruskal-algo* L
 $\langle \lambda F. \uparrow (uGraph\text{-from-list-invar } F \wedge set\ F \subseteq set\ L \wedge$
 $uGraph.minSpanningForest\ (uGraph\text{-from-list-}\alpha\text{-edges } L)$
 $(uGraph\text{-from-list-}\alpha\text{-weight } L)\ (uGraph\text{-from-list-}\alpha\text{-edges } F)) \rangle_t$
 $\langle proof \rangle$

lemma $uGraph\text{-from-list-invar } L \implies$
 $\langle emp \rangle$ *kruskal-algo* L
 $\langle \lambda F. \uparrow (uGraph\text{-from-list-invar } F \wedge set\ F \subseteq set\ L \wedge$
 $uGraph.spanningForest\ (uGraph\text{-from-list-}\alpha\text{-edges } L)\ (uGraph\text{-from-list-}\alpha\text{-edges}$
 $F)$
 $\wedge (\forall F'. uGraph.spanningForest\ (uGraph\text{-from-list-}\alpha\text{-edges } L)\ (uGraph\text{-from-list-}\alpha\text{-edges}$
 $F')) \implies set\ F' \subseteq set\ L \implies sum\ (\lambda(u,w,v). w)\ (set\ F) \leq sum\ (\lambda(u,w,v). w)$
 $(set\ F')) \rangle_t$
 $\langle proof \rangle$

6.4 Kruskal with input check

definition $kruskal' L = kruskal\ (return\ L)\ (\lambda(u,w,v). return\ (u,v))\ ()$

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

lemma $\langle emp \rangle$ $kruskal\text{-checked } L \langle \lambda$
 $Some\ F \implies \uparrow (uGraph\text{-from-list-invar } L \wedge set\ F \subseteq set\ L$
 $\wedge uGraph.minSpanningForest\ (uGraph\text{-from-list-}\alpha\text{-edges } L)\ (uGraph\text{-from-list-}\alpha\text{-weight}$
 $L)$
 $(uGraph\text{-from-list-}\alpha\text{-edges } F))$
 $| None \implies \uparrow (\neg uGraph\text{-from-list-invar } L) \rangle_t$
 $\langle proof \rangle$

6.5 Code export

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

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

$\langle ML \rangle$

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 \Rightarrow 'v \Rightarrow ('v, 'w) path \Rightarrow 'v \Rightarrow bool **where**
is-path-undir G v [] v' \longleftrightarrow v=v' \wedge v' \in nodes G |
is-path-undir G v ((v1,w,v2)#p) v'
 \longleftrightarrow v=v1 \wedge ((v1,w,v2) \in edges G \vee (v2,w,v1) \in edges G) \wedge *is-path-undir* G v2 p v'

abbreviation *nodes-connected* G a b \equiv \exists p. *is-path-undir* G a p b

definition *degree* :: ('v, 'w) graph \Rightarrow 'v \Rightarrow nat **where**
degree G v = card {e \in edges G. fst e = v \vee snd (snd e) = v}

locale *forest* = *valid-graph* G

for G :: ('v, 'w) graph +

assumes *cycle-free*:

\forall (a,w,b) \in E. \neg *nodes-connected* (delete-edge a w b G) a b

locale *connected-graph* = *valid-graph* G

for G :: ('v, 'w) graph +

assumes *connected*:

\forall v \in V. \forall v' \in 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 \Rightarrow ('v, 'w) graph \Rightarrow bool **where**

subgraph G H \equiv nodes G = nodes H \wedge edges G \subseteq edges H

definition *edge-weight* :: ('v, 'w) graph \Rightarrow 'w::weight **where**

edge-weight G \equiv sum (fst o snd) (edges G)

definition *edges-less-eq* :: ('a \times 'w::weight \times 'a) \Rightarrow ('a \times 'w \times 'a) \Rightarrow bool

where *edges-less-eq* a b \equiv fst(snd a) \leq 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 $\text{delete-edge } a \ b \ c \ (\text{add-edge } x \ y \ z \ H) = \text{add-edge } x \ y \ z \ (\text{delete-edge } a \ b \ c \ H)$
 $\langle \text{proof} \rangle$

lemma *swap-delete-edges*: $\text{delete-edge } a \ b \ c \ (\text{delete-edge } x \ y \ z \ H) = \text{delete-edge } x \ y \ z \ (\text{delete-edge } a \ b \ c \ H)$
 $\langle \text{proof} \rangle$

context *valid-graph*
begin

lemma *valid-subgraph*:
assumes *subgraph* $H \ G$
shows *valid-graph* H
 $\langle \text{proof} \rangle$

lemma *is-path-undir-simps*[*simp*, *intro!*]:
 $\text{is-path-undir } G \ v \ [] \ v \longleftrightarrow v \in V$
 $\text{is-path-undir } G \ v \ [(v, w, v')] \ v' \longleftrightarrow (v, w, v') \in E \vee (v', w, v) \in E$
 $\langle \text{proof} \rangle$

lemma *is-path-undir-memb*[*simp*]:
 $\text{is-path-undir } G \ v \ p \ v' \implies v \in V \wedge v' \in V$
 $\langle \text{proof} \rangle$

lemma *is-path-undir-memb-edges*:
assumes $\text{is-path-undir } G \ v \ p \ v'$
shows $\forall (a, w, b) \in \text{set } p. (a, w, b) \in E \vee (b, w, a) \in E$
 $\langle \text{proof} \rangle$

lemma *is-path-undir-split*:
 $\text{is-path-undir } G \ v \ (p1 @ p2) \ v' \longleftrightarrow (\exists u. \text{is-path-undir } G \ v \ p1 \ u \wedge \text{is-path-undir } G \ u \ p2 \ v')$
 $\langle \text{proof} \rangle$

lemma *is-path-undir-split'*[*simp*]:
 $\text{is-path-undir } G \ v \ (p1 @ (u, w, u') \# p2) \ v'$
 $\longleftrightarrow \text{is-path-undir } G \ v \ p1 \ u \wedge ((u, w, u') \in E \vee (u', w, u) \in E) \wedge \text{is-path-undir } G \ u' \ p2 \ v'$
 $\langle \text{proof} \rangle$

lemma *is-path-undir-sym*:
assumes $\text{is-path-undir } G \ v \ p \ v'$
shows $\text{is-path-undir } G \ v' \ (\text{rev } (\text{map } (\lambda(u, w, u'). (u', w, u)) \ p)) \ v$
 $\langle \text{proof} \rangle$

lemma *is-path-undir-subgraph*:
assumes $\text{is-path-undir } H \ x \ p \ y$
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 \neg *is-path-undir* G v p v
<proof>

lemma *is-path-undir-split-distinct:*

assumes *is-path-undir* G v p v'
assumes $(a, w, b) \in \text{set } p \vee (b, w, a) \in \text{set } p$
shows $(\exists p' p'' u u'.$
 is-path-undir G v $p' u \wedge$ *is-path-undir* G $u' p'' v' \wedge$
 $\text{length } p' < \text{length } p \wedge \text{length } p'' < \text{length } p \wedge$
 $(u \in \{a, b\} \wedge u' \in \{a, b\}) \wedge$
 $(a, w, b) \notin \text{set } p' \wedge (b, w, a) \notin \text{set } p' \wedge$
 $(a, w, b) \notin \text{set } p'' \wedge (b, w, a) \notin \text{set } p'')$
<proof>

lemma *add-edge-is-path:*

assumes *is-path-undir* G x p y
shows *is-path-undir* $(\text{add-edge } a$ b c $G)$ x p y
<proof>

lemma *add-edge-was-path:*

assumes *is-path-undir* $(\text{add-edge } a$ b c $G)$ x p y
assumes $(a, b, c) \notin \text{set } p$
assumes $(c, b, a) \notin \text{set } p$
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 $(a, b, c) \notin \text{set } p$
assumes $(c, b, a) \notin \text{set } p$
shows *is-path-undir* $(\text{delete-edge } a$ 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 $v \notin \text{fst}'\text{set } p \cup \text{snd}'\text{snd}'\text{set } p$
shows *is-path-undir* $(\text{delete-node } v$ $G)$ 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 $(a, w', a') \in E \vee (a', w', a) \in E$
shows $\exists p. \textit{is-path-undir}$ (*add-edge* $a'\ w''\ b\ G$) $v\ p\ v'$
<proof>

lemma *induce-maximally-connected*:
assumes *subgraph* $H\ G$
assumes $\forall (a, w, b) \in E. \textit{nodes-connected}\ H\ a\ b$
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* $a w b H$) $a p a b b$

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 $\forall a \in V - \{v\}. \forall b \in V - \{v\}. \textit{nodes-connected } G a b$

assumes $(v, w, v') \in E \vee (v', w, v) \in E$

assumes $v \neq v'$

shows $\forall a \in V. \forall b \in V. \textit{nodes-connected } G a b$

<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 \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) \wedge

(nodes-connected (add-edge a w' b (delete-edge x w y G)) x y)

<proof>

lemma *forest-add-edge*:

assumes *a* $\in V$

assumes *b* $\in V$

assumes \neg *nodes-connected* *G a b*

shows *forest* (*add-edge a w b G*)

<proof>

lemma *forest-subsets*:

assumes *valid-graph* *H*

assumes *edges* $H \subseteq E$

assumes *nodes* $H \subseteq V$

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* $G\ a\ b = \text{nodes-connected } G\ b\ a$
 ⟨*proof*⟩

lemma *Domain-nodes-connected*: *Domain* $\{(x, y) \mid x\ y.\ \text{nodes-connected } G\ x\ y\} = V$
 ⟨*proof*⟩

lemma *Range-nodes-connected*: *Range* $\{(x, y) \mid x\ y.\ \text{nodes-connected } G\ x\ y\} = V$
 ⟨*proof*⟩

lemma *nodes-connected-insert-per-union*:
 $(\text{nodes-connected } (\text{add-edge } a\ w\ b\ H)\ x\ y) \longleftrightarrow (x, y) \in \text{per-union } \{(x, y) \mid x\ y.\ \text{nodes-connected } H\ x\ y\}\ a\ b$
 if *subgraph* $H\ G$ **and** *PER*: *part-equiv* $\{(x, y) \mid x\ y.\ \text{nodes-connected } H\ x\ y\}$
 and $V: a \in V\ b \in V$ **for** $x\ y$
 ⟨*proof*⟩

lemma *is-path-undir-append*: *is-path-undir* $G\ v\ p1\ u \implies \text{is-path-undir } G\ u\ p2\ w$
 $\implies \text{is-path-undir } G\ v\ (p1@p2)\ w$
 ⟨*proof*⟩

lemma
 augment-edge:

assumes *sg: subgraph G1 G subgraph G2 G and*
p: (u, v) ∈ {(a, b) | a b. nodes-connected G1 a b}
and *notinE2: (u, v) ∉ {(a, b) | a b. nodes-connected G2 a b}*

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

lemma *nodes-connected-refl: a ∈ V ⇒ nodes-connected G a a*
 ⟨proof⟩

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⟩

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

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

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

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 :: (nat × int × nat) list
begin

```

abbreviation $E \equiv \text{set } L$

abbreviation $V \equiv \text{fst } \text{' } E \cup (\text{snd} \circ \text{snd}) \text{' } E$

abbreviation $\text{ind } (E' :: (\text{nat} \times \text{int} \times \text{nat}) \text{ set}) \equiv (\text{nodes} = V, \text{edges} = E')$

abbreviation $\text{subforest } E' \equiv \text{forest } (\text{ind } E') \wedge \text{subgraph } (\text{ind } E') \text{ (ind } E)$

lemma *max-node-is-Max-V*: $E = \text{set } la \implies \text{max-node } la = \text{Max } (\text{insert } 0 V)$
 ⟨*proof*⟩

lemma *ind-valid-graph*: $\bigwedge E'. E' \subseteq E \implies \text{valid-graph } (\text{ind } E')$
 ⟨*proof*⟩

lemma *vE*: $\text{valid-graph } (\text{ind } E)$ ⟨*proof*⟩

lemma *ind-valid-graph'*: $\bigwedge E'. \text{subgraph } (\text{ind } E') \text{ (ind } E) \implies \text{valid-graph } (\text{ind } E')$
 ⟨*proof*⟩

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

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

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

lemma *part-equiv-nodes-connected-ind*:
 $F \subseteq E \implies \text{part-equiv } \{(x, y) \mid x \ y. \text{nodes-connected } (\text{ind } F) \ x \ y\}$
 ⟨*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
 $\lambda E'. \{ (a, b) \mid a\ b.\ nodes\text{-}connected\ (ind\ E')\ a\ b\}$
 $\lambda(u, w, v). w\ id\ PR\text{-}CONST\ (\lambda(u, w, v). RETURN\ (u, v))$
 $PR\text{-}CONST\ (RETURN\ L)\ return\ L\ set\ L\ (\lambda(u, w, v). return\ (u, v))$
<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 now 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*:
 $\langle emp \rangle\ kruskal\ (return\ L)\ (\lambda(u, w, v). return\ (u, v))\ ()$
 $\langle \lambda F. \uparrow (distinct\ F \wedge set\ F \subseteq E \wedge minimum\text{-}spanning\text{-}forest\ (ind\ (set\ F))\ (ind\ E)) \rangle_t$
<proof>

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* $L\ e = (case\ e\ of\ (u, w, v) \Rightarrow w)$

abbreviation *GD-from-list- α -graph* $G\ L \equiv (\{nodes=fst\ ' (set\ G) \cup (snd \circ snd)\ ' (set\ G),\ edges=set\ L\})$

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

lemma *kruskal-correct*: $\langle emp \rangle\ kruskal\text{-}algo\ L$

$\langle \lambda F. \uparrow (\text{set } F \subseteq \text{set } L \wedge$
 $\text{spanning-forest } (GD\text{-from-list-}\alpha\text{-graph } L \ F) \ (GD\text{-from-list-}\alpha\text{-graph } L \ L)$
 $\wedge (\forall F'. \text{spanning-forest } (GD\text{-from-list-}\alpha\text{-graph } L \ F') \ (GD\text{-from-list-}\alpha\text{-graph } L$
 $L)) \rightarrow \text{sum } (\lambda(u,w,v). w) \ (\text{set } F) \leq \text{sum } (\lambda(u,w,v). w) \ (\text{set } F')) \rangle_t$
 $\langle \text{proof} \rangle$

8.4 Code export

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

$\langle ML \rangle$

end