Department of Informatics

Technical University of Munich

Master's Thesis in Informatics

Implementation and Verification of Efficient Minimal Spanning Tree Algorithms using Refinement in Isabelle/HOL

Julian Biendarra

Department of Informatics

Technical University of Munich

Master's Thesis in Informatics

Implementation and Verification of Efficient Minimal Spanning Tree Algorithms using Refinement in Isabelle/HOL

Implementierung und Verifikation von effizienten Minimalen Spannbaum-Algorithmen mit Hilfe von Verfeinerung in Isabelle/HOL

Author:	Julian Biendarra
Supervisor:	Prof. Dr. Tobias Nipkow
Advisor:	Dr. Peter Lammich
Submission date:	July 15, 2018

Declaration

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Garching, July 11, 2018 Julian Biendarra

Abstract

A spanning tree of a connected graph is an acyclic subgraph that connects all nodes in the graph. In a weighted graph a minimum spanning tree is a spanning tree of minimum weight. The problem of finding minimum spanning trees is well studied and has many applications.

In this thesis I focus on the Kruskal algorithm – one of the most famous algorithms for minimal spanning trees. I give a specification and verify it in Isabelle/HOL. I refine the specification with the refinement framework in Isabelle/HOL to get an executable algorithm. Finally, we have an ML program that is verified in Isabelle/HOL.

Since the Kruskal algorithm works on non-connected graphs as well, I use it as a minimum spanning forest algorithm and provide an additional algorithm for the case if we explicitly need a minimum spanning tree as output and we do not know whether our input graph is connected.

Contents

1 Introduction

To get electricity from a power plant to the customers, the power supplier constructs an electric grid of cables connecting the customers with the power plant. The company wants to connect the customers and the plant with a minimal total length of cables in order to reduce costs. The network of cables will be an acyclic graph. Otherwise, one cable in the cycle would be redundant and could be removed to reduce costs. Since the goal is to connect all customers and the power plant, the resulting network is a connected graph. Thus, the desired network is a tree that spans the graph of customers and power plant, i. e. a *spanning tree*. The problem is to find such a spanning tree of minimum weight (in our case of minimum length). This problem is called *minimum spanning tree problem*.

The problem was first mentioned by Otakar Borůvka in 1926 [1, 2] (translation of the papers: [10]). Borůvka studied the problem for electric grids in the South Moravian region in Czech Republic and developed an algorithm to solve the problem.

There are many more applications where the minimum spanning tree problem occurs, for example when producing electric circuits: The goal is to connect a set of pins on a circuit board with wires s. t. all pins are connected to each other. Unless there is any other condition, the desired goal is a connection with a minimum length of wire. This can be modeled with the minimum spanning tree problem [4].

Over the years the problem was studied many times and several algorithms were developed. One of the most famous algorithms is the algorithm by Joseph B. Kruskal proposed in 1956 [6].

In this thesis I will have a deeper look at the Kruskal algorithm. The input graph to the algorithm must be connected, otherwise the graph has no minimum spanning tree, which can be constructed by the algorithm. However, the Kruskal algorithm – like some other minimum spanning tree algorithms – also works for non-connected graphs. Then the algorithm constructs a minimum spanning forest, i. e. a forest of minimum weight that spans all components of the initial graph. Since this is more general (a minimum spanning tree is also a minimum spanning forest), I will not assume that the input graphs are connected and prove that the Kruskal algorithm constructs a minimum spanning forest. Moreover, I will prove that if the input graph is connected the algorithm constructs a minimum spanning tree.

The goal of the thesis is to specify and verify the Kruskal algorithm in *Isabelle/HOL* – a theorem prover for Higher-Order Logic [11] – and to use refinement techniques to change the algorithm step by step to get an efficient executable program in the end.

In the refinement process I will change one aspect of the algorithm in each step. Then I will prove that the changed algorithm computes the same as the previous algorithm. For this process I will use the Imperative Refinement Framework in Isabelle/HOL [8]. With this framework the last refinement step is generated and proven automatically based on previously proven refinements of the functions used. Additionally, the framework generates executable ML code from the verified algorithm.

After a few definitions and helping lemmas (chapter 2), I will describe the Kruskal algorithm in detail (chapter 3.1), give a formal specification of the algorithm (chapter 3.2) and verify the correctness of the algorithm (chapter 3.3). In chapter 4 I will refine the algorithm step by step to get an executable program (chapters 4.1 to 4.5) and prove the overall correctness theorems (chapter 4.6).

Since the specified Kruskal algorithm might return a forest that is not connected, I will define another algorithm that only returns a result (namely a minimum spanning tree) if the input graph is connected (chapter 5). Hence, we can use this algorithm if we need a minimum spanning tree and we are not sure whether the input graph is connected or not. Finally, I will write wrappers around the generated ML code to get executable command line programs for both algorithms (chapter 6).

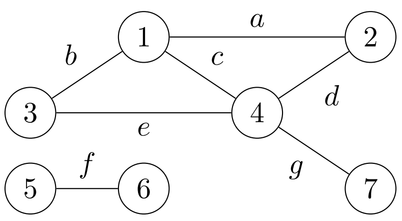
2 Basic Definitions and Helping Lemmas

2.1 Graph Definitions

To model the minimum spanning forest and the minimum spanning tree problem we first need a few graph definitions.

Definition 2.1.1 (Graphs)**.** A graph is a tuple $G = (V, E)$ of a set of nodes V and a set of edges E, where the edges are pairs of vertices with labels $E \subseteq V \times L \times V$ for a set of labels L.

Example 1 (Graphs)**.** Let $G = (V, E)$ be the following graph:



In this case the nodes are $V = \{1, 2, 3, 4, 5, 6, 7\}$ and the edges are $E = \{(1, a, 2), (1, b, 3), (1, c, 4), (2, d, 4), (3, e, 4), (5, f, 6), (4, g, 7)\}$. Then the labels are $L = \{a, b, c, d, e, f, g\}$.

We only consider finite graphs:

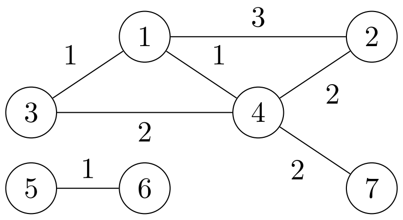
Definition 2.1.2 (Finite Graphs)**.** A graph $G = (V, E)$ is finite iff the set of nodes V is finite and the set of edges E is finite.

We also need an edge weight for every edge to compare the edges by their weight:

Definition 2.1.3 (Weighted Graphs)**.** In a weighted graph $G = (V, E)$ each edge $e \in E$ has an edge weight w_e.

Remark. We will model the weights as edge labels, i. e. the edge $(a, w, b) \in E$ has the weight w.

Example 2 (Weighted Graphs)**.** If we introduce weights for the graph in example 1, we will get a modified graph $G = (V, E)$ where we replace the labels by edge weights:



In this case the edges are $E = \{(1,3,2), (1,1,3), (1,1,4), (2,2,4), (3,2,4), (5,1,6), (4,2,7)\}$, i. e. each edge contains its weight as the label.

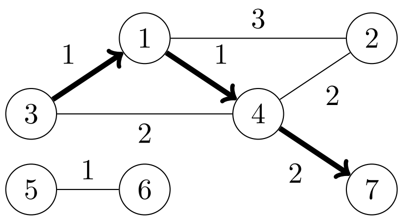
A basic concept is the concept of paths. Paths are lists of edges forming a connection between two nodes.

Definition 2.1.4 (Paths)**.** Let $G = (V, E)$ be a graph, $a, b \in V$. A path p from a to b in G is a list of edges $[e_1, e_2, \ldots, e_n]$ (with $e_i = (a_i, l_i, b_i)$ and $(a_i, l_i, b_i) \in E \vee (b_i, l_i, a_i) \in E$ for $i \in \{1, \ldots, n\}$) such that adjacent edges share incident nodes:

$$a = a_1 \wedge (\forall i \in \{1, \ldots, n-1\}.\ b_i = a_{i+1}) \wedge b_n = b.$$

Remark. We only consider undirected graphs, i. e. there is a path from a to b iff there is a path from b to a. Therefore, we interpret the edges in E as undirected edges.

Example 3 (Paths)**.** We take the same graph $G = (V, E)$ as in example 2, we have a look at a path from 3 to 7:



Then $[(3,1,1), (1,1,4), (4,2,7)]$ is a path from 3 to 7 in G, since $(1,1,3), (1,1,4), (4,2,7) \in E$.

For the notion of trees we need the definition of connected graphs:

Definition 2.1.5 (Connected Graphs)**.** A graph $G = (V, E)$ is connected iff for all pairs of nodes there is a path between them:

$$\forall a, b \in V.\ \exists \text{path } p \text{ from } a \text{ to } b \text{ in graph } G.$$

Example 4 (Connected Graphs)**.** The graph from example 2 is not connected because there is for example no path from 3 to 5 (5 and 6 are in a different component than the other nodes). If we, for example, add an edge from 6 to 7, the new graph will be connected.

In example 4 we have seen that our example graph is not connected. In this case we cannot find a spanning tree, but we can find a spanning forest. For the notion of spanning forests we need a weaker connectedness property:

Definition 2.1.6 (Maximally Connected Graphs)**.** A graph $G' = (V', E')$ is maximally connected w. r. t. a graph $G = (V, E)$ iff all nodes that are connected in G are connected in G':

$$\forall a, b \in V.\ \ (\exists \text{path } p \text{ from } a \text{ to } b \text{ in } G) \longrightarrow (\exists \text{path } p \text{ from } a \text{ to } b \text{ in } G').$$

Now we can define forests and trees:

Definition 2.1.7 (Forests)**.** A graph $G = (V, E)$ is a forest iff it is cycle-free, i. e. if any edge is removed from the graph, the incident nodes are not connected in the new graph:

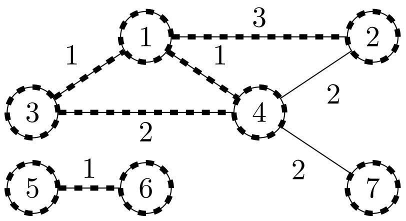
$$\forall (a, l, b) \in E.\ \neg\exists \text{path } p \text{ from } a \text{ to } b \text{ in the graph } (V, E \setminus \{(a, l, b)\})$$

Definition 2.1.8 (Trees)**.** A graph $G = (V, E)$ is a tree iff it is a connected graph and a forest.

We define subgraphs over the subset relation on edges. The nodes need to be the same. Hence, we can easily combine this definition with the definition of connected graphs.

Definition 2.1.9 (Subgraphs)**.** A graph $G' = (V', E')$ is a subgraph of a graph $G = (V, E)$ iff the nodes are equal ($V' = V$) and the edges are a subset ($E' \subseteq E$).

Example 5 (Subgraphs)**.** We consider the graph $G = (V, E)$ from example 2:



The graph $G' = (V' := V,\ E' := \{(1,3,2), (1,1,3), (1,1,4), (3,2,4), (5,1,6)\})$ is a subgraph of G', since $V' = V$ and $E' \subseteq E$.

We will also need an auxiliary function to get the element set of a list:

Algorithm 2.1.10: `set`

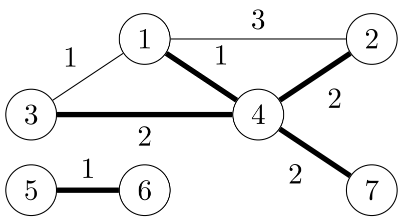
```
Input   : list l
Output: the set of elements in l
set [] = {}
set (a#as) = {a} ∪ set as
```

2.2 Definitions of Minimum Spanning Forests and Trees

Now we can define (minimum) spanning forests and trees as well as the corresponding optimization problems:

Definition 2.2.1 (Spanning Forests). A graph F is a spanning forest of a graph G iff F is a forest, a subgraph of G and maximally connected w. r. t. G.

Example 6 (Spanning Forests). We take the graph $G = (V, E)$ from example 2:



The graph $F = (V, E_F)$ with $E_F = \{(1,1,4), (2,2,4), (3,2,4), (5,1,6), (4,2,7)\}$ is a spanning forest of G.

Definition 2.2.2 (Minimum Spanning Forests). A graph $F = (V, E_F)$ is a minimum spanning forest of a weighted graph $G = (V, E)$ iff F is a spanning forest of G and for all spanning forests $F' = (V, E_{F'})$ of G the total edge weight of F is less or equal to the total edge weight of F':

$$\forall F'.\ F' \text{ is a spanning forest of } G \longrightarrow \sum_{(a,w,b)\in E_F} w \leq \sum_{(a,w,b)\in E_{F'}} w.$$

Definition 2.2.3 (The Minimum Spanning Forest Problem). The minimum spanning forest problem is the following problem:

Given a weighted graph G find a minimum spanning forest F of G.

Definition 2.2.4 (Spanning Trees). A graph T is a spanning tree of a graph G iff T is a tree and a subgraph of G.

Definition 2.2.5 (Minimum Spanning Trees). A graph $T = (V, E_T)$ is a minimum spanning tree of a weighted graph $G = (V, E)$ iff T is a spanning tree of G and for all spanning trees $T' = (V, E_{T'})$ of G the total edge weight of T is less or equal to the total edge weight of T':

$$\forall T'.\ T' \text{ is a spanning tree of } G \longrightarrow \sum_{(a,w,b)\in E_T} w \leq \sum_{(a,w,b)\in E_{T'}} w.$$

Definition 2.2.6 (The Minimum Spanning Tree Problem). The minimum spanning tree problem is the following problem:

Given a weighted, connected graph G find a minimum spanning tree T of G.

2.3 Refinement

We will use multiple refinement steps to get from the abstract specification to the concrete implementation. Refinement is a relation between two algorithms. One algorithm refines the other when it computes the same output as the other given the same input.

Since the algorithms can be non-deterministic, we need to be a little bit more general. We consider the set of possible output values for a given input. A refined algorithm might return a subset of the output values of the original algorithm.

In the proof below we will first specify a variable to be a sorted list, later we will define the specific deterministic sorting algorithm that will be used to get the sorted list. The first specification has multiple possible outputs, as it defines not how elements with equal keys are sorted. The concrete sorting algorithm is deterministic, i. e. it produces one specific output (for a given input). If the sorting algorithm is correct, this output is part of the output set of the specification, so the second algorithm refines the specification.

Sometimes we want to change the input or output interface of an algorithm. In the proof we will first model graphs as two sets (the set of nodes V and the set of edges E as in definition 2.1.1) and later we will represent graphs as lists of edges. Then we cannot say anymore that the refined algorithm computes a subset of the output set. Hence, we define the refinement a bit more general: If the inputs of both algorithms are related by a given input relation, then the output values in the output sets must be related by a given output relation.

In our example the input relation would be the relation that relates the graphs modeled by sets with the graphs modeled by edge lists. This relation will be defined in definition 4.2.2.

Definition 2.3.1 (Refinement)**.** Assume that given an input i the algorithm A computes the set of possible outputs O and given an input i' the algorithm A' computes the output set O'.

The algorithm A' refines the algorithm A under the input relation R_I and under the output relation R_O iff the following condition hold: If the inputs i' and i are related ($(i', i) \in R_I$), then the sets of output values O' and O are related ($\forall o' \in O'.\ \exists o \in O.\ (o', o) \in R_O$).

2.4 Helping Lemmas

In this section we will show a few helping lemmas that we will need later.

Lemma 2.4.1 (Path Preservation When Adding Edge)**.** *Let $G = (V, E)$. Let p be a path from x to y in G. Then p is a path from x to y in the graph $G_{add} = (V, E_{G_{add}})$ with $E_{G_{add}} = E \cup \{(a, w, b)\}$.*

Proof. Proof by induction on p:

Base Case $p = []$

Then $x = y$ and therefore p is also a path from x to y in G_{add}.

Induction Step $p = (v, w', v')\#p'$

Then $(v, w', v') \in E \vee (v', w', v) \in E$, $v = x$ and p' is a path from v' to y in G, since p is a valid path from x to y in G. Then $(v, w', v') \in E_{G_{\text{add}}} \vee (v', w', v) \in E_{G_{\text{add}}}$ and by induction hypothesis p' is a path from v' to y in G_{add}. Hence, p is a valid path from x to y in G_{add}.

□

Lemma 2.4.2 (Path Preservation When Deleting Edge)**.** *Let $G = (V, E)$. Let p be a path from x to y in G with $(a, w, b) \notin$ `set` p and $(b, w, a) \notin$ `set` p. Then p is a path from x to y in the graph $G_{del} = (V, E_{G_{del}})$ with $E_{G_{del}} = E \setminus \{(a, w, b)\}$.*

Proof. Proof by induction on p:

Base Case $p = []$

Then $x = y$ and therefore p is also a path from x to y in G_{del}.

Induction Step $p = (v, w', v')\#p'$

Then $(v, w', v') \in E \vee (v', w', v) \in E$, $v = x$ and p' is a path from v' to y in G, since p is a valid path from x to y in G. As $(a, w, b) \notin$ `set` p and $(b, w, a) \notin$ `set` p and $(v, w', v') \in$ `set` p, $(v, w', v') \in E_{G_{\text{del}}} \vee (v', w', v) \in E_{G_{\text{del}}}$. Moreover, p' is a path from v' to y in G_{del} by induction hypothesis. Hence, p is a valid path from x to y in G_{del}.

□

Lemma 2.4.3 (Path Preservation When Removing Added Edge). *Let $G_{add} = (V, E \cup \{(a, w, b)\})$. Let p be a path from x to y in G_{add} with $(a, w, b) \notin$ `set` p and $(b, w, a) \notin$ `set` p. Then p is a path from x to y in the graph $G = (V, E)$.*

Proof. Proof by case distinction:

Case 1 $(a, w, b) \in E$

Then $G_{\text{add}} = G$ and the lemma trivially holds.

Case 2 $(a, w, b) \notin E$

Then by lemma 2.4.2 p is a path from x to y in the graph $(V, E \cup \{(a, w, b)\} \setminus \{(a, w, b)\}) = (V, E) = G$.

□

Lemma 2.4.4 (Path Preservation in Subgraphs). *Let $G = (V, E)$ and $H = (V_H, E_H)$ a subgraph of G. Let p be a path from x to y in H. Then p is a path from x to y in G.*

Proof. Since H is a subgraph of G, $V_H = V$ and $E_H \subseteq E$ by the definition of subgraphs (2.1.9).

Proof by induction on p:

Base Case $p = []$

Then $x = y$ and therefore p is also a path from x to y in G.

Induction Step $p = (v, w', v')\#p'$

Then $(v, w', v') \in E_H \vee (v', w', v) \in E_H$, $v = x$ and p' is a path from v' to y in H, due to the fact that p is a valid path from x to y in H. As $E_H \subseteq E$, $(v, w', v') \in E \vee (v', w', v) \in E$. Moreover, p' is a path from v' to y in G by induction hypothesis. Hence, p is a valid path from x to y in G.

□

Lemma 2.4.5 (Split Path by Edge). *Let $G = (V, E)$ be a graph, let $v, v' \in V$, let p be a path from v to v' in G with $(a, w, b) \in$ `set` $p \vee (b, w, a) \in$ `set` p.*

Then there are paths p' and p'' and $u, u' \in \{a, b\}$ s. t. p' is a path from v to u in G with `length` $p' <$ `length` p *and $(a, w, b) \notin$ `set` $p' \wedge (b, w, a) \notin$ `set` p'. Furthermore, p'' is a path from u' to v' in G with* `length` $p'' <$ `length` p *and $(a, w, b) \notin$ `set` $p'' \wedge (b, w, a) \notin$ `set` p''.*

Proof. Proof by induction on the length of p (for arbitrary nodes $v, v' \in V$): Let $n \in \mathbb{N}$, assume that the goal holds for all paths p with length $l < n$ (*induction hypothesis*). Proof the goal for an arbitrary path p from v to v' in G with length n.

Let $u, u' \in \{a, b\}$ s. t. $(u, w, u') \in \texttt{set}\ p$. We can split p into $p = p'@[(u, w, u')]@p''$ where p' and p'' are lists. Then the lengths of p' and p'' are less than n. Moreover, p' is a path from v to u in G and p'' is a path from u' to v' in G.

If $(a, w, b) \in \texttt{set}\ p' \vee (b, w, a) \in \texttt{set}\ p'$, by induction hypothesis we have that there is a path p_2' and $u_2 \in \{a, b\}$ s. t. p_2' is a path from v to u_2 in G with $\texttt{length}\ p_2' \leq \texttt{length}\ p' < \texttt{length}\ p$ and $(a, w, b) \notin \texttt{set}\ p_2' \wedge (b, w, a) \notin \texttt{set}\ p_2'$. Otherwise, if $(a, w, b) \notin \texttt{set}\ p' \wedge (b, w, a) \notin \texttt{set}\ p'$, then $p_2' = p'$ and $u_2 = u$ fulfill the same proposition.

For p'' we get analogously: There is a path p_2'' and $u_2' \in \{a, b\}$ s. t. p_2'' is a path from u_2'' to v' in G with $\texttt{length}\ p_2'' < \texttt{length}\ p$ and $(a, w, b) \notin \texttt{set}\ p_2'' \wedge (b, w, a) \notin \texttt{set}\ p_2''$. ☐

Lemma 2.4.6 (Swap Edges). *Let $G = (V, E)$ be a graph. Let $a, b, v, v' \in V$. Assume there is a path p from v to v' in the graph $G_{awb} = (V, E \cup \{(a, w, b)\})$, but there is no path from v to v' in G.*

Then we can swap the edge (a, w, b) by an edge (v, w', v'): There is a path from a to b in the graph $G_{vw'v'} = (V, E \cup \{(v, w', v')\})$.

Proof. We have $(a, w, b) \in \texttt{set}\ p \vee (b, w, a) \in \texttt{set}\ p$, otherwise p would be a valid path from v to v' in G as well by lemma 2.4.3 which is in contradiction with the assumption that there is no such path in G.

By lemma 2.4.5 for G_{awb}, we have for $u, u' \in \{a, b\}$ that there is a path p' from v to u in G_{awb} with $(a, w, b) \notin \texttt{set}\ p' \wedge (b, w, a) \notin \texttt{set}\ p'$ and there is a path p'' from u' to v' in G_{awb} with $(a, w, b) \notin \texttt{set}\ p'' \wedge (b, w, a) \notin \texttt{set}\ p''$.

Then p' is a path from v to u in G by lemma 2.4.3. Analogously, p'' is a path from u' to v' in G.

It holds that $u \neq u'$, otherwise the paths p' and p'' together would be a path from v to v' in G which is in contradiction with the assumption that there is no such path in G.

p' is also a path from v to u in $G_{vw'v'}$ by lemma 2.4.1. Analogously, p'' is a path from u' to v' in $G_{vw'v'}$. Furthermore, $[(v', w', v)]$ is a valid path from v' to v in $G_{vw'v'}$, since $(v, w', v') \in E \cup \{(v, w', v')\}$.

Then the three paths p'', $[(v', w', v)]$ and p' form a path from u' to u in $G_{vw'v'}$. As $u, u' \in \{a, b\}$ and $u \neq u'$, the path is from a to b or from b to a. In the first case we get the goal directly, in the second case we get the goal by symmetry due to the fact that the graph is undirected: There is a path from a to b in $G_{vw'v'}$. □

Lemma 2.4.7 (Forest Add Edge). *Let $G = (V, E)$ be a forest, let $a, b \in V$. If there is no path between a and b in G, the graph $G' = (V, E')$ with $E' = E \cup \{(a, w, b)\}$ is a forest.*

Proof. Assume there is no path between a and b in G. Then $(a, w, b) \notin E$, otherwise $[(a, w, b)]$ would be a path from a to b in G.

Using the definition of forests (2.1.7), we need to prove for every edge $(v, w', v') \in E'$ that there is no path from v to v' in the graph $G'_{\text{del}} = (V, E' \setminus \{(v, w', v')\})$.

Let $(v, w', v') \in E'$, we distinguish two cases:

Case 1 $(v, w', v') = (a, w, b)$

Then $G'_{\text{del}} = (V, E' \setminus \{(v, w', v')\}) = (V, E) = G$ because $(a, w, b) \notin E$. By assumption, there is no path from v to v' in G.

Case 2 $(v, w', v') \neq (a, w, b)$

Proof by contradiction: Assume that there is a path p from v to v' in the graph G'_{del}.

From the assumption $(v, w', v') \neq (a, w, b)$ and $(v, w', v') \in E' = E \cup \{(a, w, b\}$ we get $(v, w', v') \in E$.

We will apply lemma 2.4.6 to the graph $G_{\text{del}} = (V, E \setminus \{(v, w', v')\})$. There is a path from v to v' in the graph $(V, E \setminus \{(v, w', v')\} \cup \{(a, w, b)\}) = G'_{\text{del}}$ by assumption due to $(v, w', v') \neq (a, w, b)$. And there is no path from v to v' in the graph G_{del} by the definition of forests (2.1.7), since G is a forest and $(v, w', v') \in E$. With these two assumptions we get by lemma 2.4.6 that there is a path from a to b in the graph $(V, E \setminus \{(v, w', v')\} \cup \{(v, w', v')\}) = G$ because $(v, w', v') \in E$.

This is in contradiction with the assumption of the lemma. Thus, our assumption that there is a path from v to v' in the graph G'_{del} is false, i. e. there is no such path.

□

Lemma 2.4.8 (Existence of a Forest Connecting All Edges). *Let $G = (V, E)$ be a finite graph. There is a forest F s. t. F is a subgraph of G and for every edge $(a, w, b) \in E$ there is a path from a to b in F.*

Proof. We will prove this by induction on the cardinality of E.

Base Case $|E| = 0$

G is a forest by the definition of forests (2.1.7), since it has no edges, and it is a subgraph of itself by the definition of subgraphs (2.1.9). Finally, the third condition also holds for G due to the fact that G has no edges.

Induction Step Let $n \in \mathbb{N}$. Assume that the goal holds for all graphs $G = (V, E)$ with $|E| = n$ (*induction hypothesis*). Prove the goal for all graphs $G = (V, E)$ with $|E| = n + 1$.

Let $G = (V, E)$ with $|E| = n + 1$. As $n \in \mathbb{N}$, $E \neq \{\}$. Hence, there is an edge $(a, w, b) \in E$. We define the graph $G' = (V, E')$ with $E' = E \setminus \{(a, w, b)\}$. Due to the fact that $|E'| = n$, there is a forest $F = (V, E_F)$ by induction hypothesis s. t. F is a subgraph of G' and for every edge $(c, w', d) \in E'$ there is a path from c to d in F. Since $E_F \subseteq E' = E \setminus \{(a, w, b)\} \subseteq E$ by the definition of subgraphs (2.1.9), F is a subgraph of G.

We distinguish two cases to prove the goal for G:

Case 1 There is a path p from a to b in F.

Then for every edge $(c, w', d) \in E$ there is a path from c to d in F. Thus, F fulfills the goal for G.

Case 2 There is no path from a to b in F.

We will prove that $F' = (V, E_{F'})$ with $E_{F'} = E_F \cup \{(a, w, b)\}$ fulfills the goal for G.

F' is a forest by lemma 2.4.7, since there is no path from a to b in F.

F' is a subgraph of G by the definition of subgraphs, as $(a, w, b) \in E$ and F is a subgraph of G.

Finally, we prove by case distinction that for every edge $(c, w', d) \in E$ there is a path from c to d in F':

Case 2.1 $(c, w', d) = (a, w, b)$

Then $[(a, w, b)]$ is a path from c to d in F' due to the fact that $(a, w, b) \in E_{F'}$.

Case 2.2 $(c, w', d) \neq (a, w, b)$

Then $(c, w', d) \in E'$ and therefore there is a path p from c to d in F. By lemma 2.4.1, p is also a valid path from c to d in F'.

□

Lemma 2.4.9 (Existence of a Spanning Forest). *Let $G = (V, E)$ be a finite graph. Then G has a spanning forest.*

Proof. By lemma 2.4.8, there is a forest F where F is a subgraph of G and for every edge $(a, w, b) \in E$ there is a path from a to b in F.

We will prove that F is maximally connected w. r. t. G. Let $v, v' \in V$ and assume there is a path p from v to v' in G. We prove by induction on p that there is a path p' from v to v' in F.

Base Case $p = []$

Then $v = v'$ and therefore $p' = []$ is a path from v to v' in F, since F is a subgraph of G.

Induction Step $p = (x, w', y)\#p_2$

Then $(x, w', y) \in E \lor (y, w', x) \in E$, $x = v$ and p_2 is a path from y to v' in G, as p is a valid path from v to v' in G. Due to the fact that for every edge $(a, w, b) \in E$ there is a path from a to b in F, there is a path p_1' from $x = v$ to y in F. By induction hypothesis for p_2, there is a path p_2' from y to v' in graph F. Then $p_1'@p_2'$ is a path from v to v' in F.

Thus, F is maximally connected w. r. t. G by the definition of maximal connected graphs (2.1.6). Hence, F is a spanning forest by the definition of spanning forests (2.2.1), since it is also a forest and a subgraph of G. □

Lemma 2.4.10 (Existence of a Minimum Spanning Forest). *Let $G = (V, E)$ be a finite, weighted graph. Then G has a minimum spanning forest.*

Proof. Define the set $W = \{\texttt{edge\_weight}\ F.\ F \text{ is a spanning forest of } G\}$, where `edge_weight` F for a graph $F = (V_F, E_F)$ is defined as the sum of weights of all edges:

$$\texttt{edge\_weight}\ F = \sum_{(a,w,b) \in E_F} w.$$

Due to the fact that G is a finite graph, the set of edges E in G is finite by the definition of finite graphs (2.1.2). Then the set of subsets of edges in G, i. e. $\{E'.\ E' \subseteq E\}$, is finite as well. Hence, the set of subgraphs of G is also finite, since a subgraph consists of the node set V and a subset of edges $E' \subseteq E$ by the definition of subgraphs (2.1.9). As all spanning forests of G are subgraphs of G by the definition of spanning forests (2.2.1), W is finite as well.

By lemma 2.4.9, G has a spanning forest F. Then `edge_weight` $F \in W$, i. e. $W \neq \{\}$. Thus, there is a minimum w_{min} of the set W with $\forall w \in W.\ w_{\text{min}} \leq w$ and $w_{\text{min}} \in W$. Let F' be a spanning forest of G s. t. `edge_weight` $F' = w_{\text{min}}$. Then F' is a minimum spanning forest of G by the definition of minimum spanning forests (2.2.2). □

Lemma 2.4.11 (Swap Add Edge in Path)**.** *Let $G = (V, E)$ be a graph. Let $v, v' \in V$. Let p be a path from v to v' in the graph $G' = (V, E \cup \{(a, w, b)\})$. Let $(a, w', a') \in E \vee (a', w', a) \in E$ be an edge. Then there is a path from v to v' in the graph $G'' = (V, E \cup \{(a', w'', b)\})$.*

Proof. Proof by induction on p for an arbitrary node v:

Base Case $p = []$

Then $v = v'$ and therefore $p' = []$ is a path from v to v' in G''.

Induction Step $p = e \# p'$

As p is a path from v to v' in G', $e = (v, w_e, x)$ (for some label w_e and some node $x \in V$), $(v, w_e, x) \in E \cup \{(a, w, b)\} \vee (x, w_e, v) \in E \cup \{(a, w, b)\}$ and p' is a path from x to v' in G'. By induction hypothesis, there is a path $p_{xv'}$ from x to v' in G''. It remains to show that there is a path from v to x in G''. With $p_{xv'}$ we get a path form v to v' in G''.

If $(v, w_e, x) \neq (a, w, b) \wedge (v, w_e, x) \neq (b, w, a)$, then $[(v, w_e, x)]$ is a path from v to x in G''.

So we assume $(v, w_e, x) = (a, w, b) \vee (v, w_e, x) = (b, w, a)$. Moreover, $[(a, w', a'),$ $(a', w'', b)]$ is a path from a to b in G'', since $(a, w', a') \in E \vee (a', w', a) \in E$ and $(a', w'', b) \in E \cup (a', w'', b)$. By symmetry of undirected graphs, there is also a path from b to a in G''. Thus, there is a path from v to x in G'' by assumption.

□

Lemma 2.4.12 (Delete Edge From Path). *Let $G = (V, E)$ be a forest and $H = (V, E_H)$ a subgraph of G, let $a, b \in V$, let p be a path from a to b in G. If there is no path from a to b in H, there is an edge $(x, w, y) \in E \setminus E_H$ s. t. there is no path from a to b in the graph $G_{del} = (V, E_{G_{del}})$ with $E_{G_{del}} = E \setminus \{(x, w, y)\})$ and there is a path from x to y in the graph $G' = (V, E_{G'})$ with $E_{G'} = E \setminus \{(x, w, y)\} \cup \{(a, w', b)\}$ for an arbitrary label w'.*

Proof. Proof by induction on the length of p (for arbitrary nodes $a, b \in V$): Let $n \in \mathbb{N}$. Assume that the goal holds for all paths p with length $l < n$ (*induction hypothesis*). Proof the goal for an arbitrary path p of length n.

If $p = []$ (i. e. $n = 0$), then $a = b$, since p is a path from a to b in G. Then p is also a path from a to b in H, as $a, b \in V$. This is in contradiction with the assumption that there is no such path in H. Thus, $p \neq []$.

Then there is an edge e and a path p' s. t. $p = e\#p'$. Due to the fact that p is a path from a to b in G, $e = (a, w, a')$ (for some w, a'), $(a, w, a') \in E \vee (a', w, a) \in E$ and p' is a path from a' to b in G.

If $(a, w, a') \in \texttt{set}\ p' \vee (a', w, a) \in \texttt{set}\ p'$, then we have by lemma 2.4.5 for G that there is a path p'_{dst} and $u' \in \{a, a'\}$ s. t. p'_{dst} is a path from u' to b in G with $\texttt{length}\ p'_{\text{dst}} \leq \texttt{length}\ p' < \texttt{length}\ p$ and $(a, w, a') \notin \texttt{set}\ p'_{\text{dst}} \wedge (a', w, a) \notin \texttt{set}\ p'_{\text{dst}}$. Otherwise, if $(a, w, a') \notin \texttt{set}\ p' \wedge (a', w, a) \notin \texttt{set}\ p'$, then $p'_{\text{dst}} = p'$ and $u' = a'$ fulfill the same proposition.

If $u' = a$, then p'_{dst} is a path from a to b in G with $\texttt{length}\ p'_{\text{dst}} < \texttt{length}\ p$. The goal follows directly if we apply the induction hypothesis. Thus, we only consider $u' = a'$, i. e. p'_{dst} is a path from a' to b in G.

Let $e_1, e_2 \in V$ s. t. $e = (e_1, w, e_2) \vee e = (e_1, w, e_2)$ and $(e_1, w, e_2) \in E$. We make a case distinction on (e_1, w, e_2):

Case 1 $(e_1, w, e_2) \in E_H$

Then there is no path from a' to b in H, otherwise with the edge (e_1, w, e_2) there would be a path from a to b in H which is in contradiction with the lemma assumption.

With $\texttt{length}\ p'_{\text{dst}} < \texttt{length}\ p$ and p'_{dst} is a path from a' to b, we get by induction hypothesis that there is an edge $(x, z, y) \in E \setminus E_H$ s. t. there is no path from a' to b in the graph $G_{\text{del}} = (V, E_{G_{\text{del}}})$ with $E_{G_{\text{del}}} = E \setminus \{(x, z, y)\})$ and there is a path p_{xy} from x to y in the graph $G' = (V, E_{G'})$ with $E_{G'} = E \setminus \{(x, z, y)\} \cup \{(a', w', b)\}$.

We first prove by contradiction that there is no path from a to b in G_{del} (the first part of the goal):

Assume that there is a path p_e from a to b in G_{del}. As $(x, z, y) \in E \setminus E_H$ and $(e_1, w, e_2) \in E_H$, $(e_1, w, e_2) \neq (x, z, y)$ and therefore $(e_1, w, e_2) \in E_{G_{\text{del}}}$. By definition of e_1 and e_2, we have that $[(a', w, a)]$ is a path from a' to a in G_{del}. The paths $[(a', w, a)]$ and p_e together form a path from a' to b in G_{del} which is in contradiction with the fact that there is no such path.

By lemma 2.4.11 for p_{xy}, we have a path from x to y in the graph $G'' = (V, E_{G''})$ with $E_{G''} = E \setminus \{(x, z, y)\} \cup \{(a, w', b)\}$, since $(e_1, w, e_2) \in E \setminus \{(x, z, y)\}$.

With the fact that there is no path from a to b in G_{del}, we have the goal.

Case 2 $(e_1, w, e_2) \notin E_H$

We will prove that (e_1, w, e_2) is the desired edge. Let $G_{\text{del}} = (V, E_{G_{\text{del}}})$ with $E_{G_{\text{del}}} = E \setminus \{(e_1, w, e_2)\})$ and $G' = (V, E_{G'})$ with $E_{G'} = E \setminus \{(e_1, w, e_2)\} \cup \{(a, w', b)\}$.

By definition of e_1 and e_2, we have $(e_1, w, e_2) \notin \texttt{set}\ p'_{\text{dst}} \wedge (e_1, w, e_2) \notin \texttt{set}\ p'_{\text{dst}}$. Due to the fact that p'_{dst} is a path from a' to b in G, it is also a path from a' to b in G_{del} by lemma 2.4.2.

We first prove by contradiction that there is no path from a to b in G_{del} (the first part of the goal):

Assume that there is a path p_e from a to b in G_{del}. As p'_{dst} is a path from a' to b in G_{del}, by symmetry in undirected graphs there is also a path p_{rev} from b to a' in G_{del}. Then p_e and p_{rev} form a path from a to a' in G_{del}. By symmetry, there is also a path from a' to a in G_{del}. Hence, there is a path from e_1 to e_2 in G_{del} by definition of e_1, e_2. This is in contradiction with the fact that G is a forest by the definition of forests (2.1.7).

Furthermore, p'_{dst} is a path from a' to b in G' by lemma 2.4.1, since it is a path from a' to b in G_{del}. Moreover, we have $(a, w', b) \in E_{G'}$. Therefore, $[(b, w', a)]$ is a path from b to a in G'.

Thus, the paths p'_{dst} and $[(b, w', a)]$ form a path from a' to a in G'. By symmetry in undirected graphs, there is also a path from a to a' in G'. Hence, there is a path from e_1 to e_2 in G' by definition of e_1 and e_2.

With the fact that there is no path from a to b in G_{del}, we have the goal.

$\square$

Lemma 2.4.13 (Add Edge Maximally Connected)**.** *Let $G = (V, E)$ be a graph and $H = (V, E_H)$ be a subgraph of G that is maximally connected w. r. t. G. Let $(a, w, b) \in E$ be an edge in G. Then the graph $H' = (V, E_{H'})$ with $E_{H'} = E_H \cup \{(a, w, b)\}$ is maximally connected w. r. t. G.*

Proof. Let $v, v' \in V$ and assume there is a path from v to v' in G. By the definition of maximally connected graphs (2.1.6), we need to prove that there is a path from v to v' in H'.

Since H is maximally connected w.r.t. G, there is a path p from v to v' in H by definition. By lemma 2.4.1, the path p is a valid path from v to v' in H' as well. $\square$

Lemma 2.4.14 (Delete Edge Maximally Connected)**.** *Let $G = (V, E)$ be a graph and $H = (V, E_H)$ be a subgraph of G that is maximally connected w. r. t. G. Let $H' = (V, E_{H'})$ with $E_{H'} = E_H \setminus \{(a, w, b)\}$. Let p_{ab} be a path from a to b in H'. Then H' is maximally connected w. r. t. G.*

Proof. Let $v, v' \in V$ and assume there is a path from v to v' in G. By the definition of maximally connected graphs (2.1.6), we need to prove that there is a path from v to v' in H'.

As H is maximally connected w. r. t. G, there is a path p from v to v' in H by definition.

We prove the goal by case distinction:

Case 1 $(a, w, b) \in \texttt{set}\ p \lor (b, w, a) \in \texttt{set}\ p$

By lemma 2.4.5 for H, we have that for $u, u' \in \{a, b\}$ there is a path p' from v to u in H with $(a, w, b) \notin \texttt{set}\ p' \land (b, w, a) \notin \texttt{set}\ p'$ and there is a path p'' from u' to v' in H with $(a, w, b) \notin \texttt{set}\ p'' \land (b, w, a) \notin \texttt{set}\ p''$.

Then p' and p'' are valid paths in H' as well by lemma 2.4.2.

If $u = u'$, p' and p'' together form a path from v to v' in H'. Otherwise, by assumption there is a path p_{ab} from a to b in H'. By symmetry of undirected graphs, there is also a path p'_{ab} from b to a in H'. Together with p' and p'' we can form a path from v to v' in H' (depending on the values of u and u' we will use p_{ab} or p'_{ab} between p' and p'').

Case 2 $(a, w, b) \notin \texttt{set}\ p \land (b, w, a) \notin \texttt{set}\ p$

Then p is also a valid path from v to v' in H' by lemma 2.4.2.

□

Lemma 2.4.15 (Maximally Connected w. r. t. a Connected Graph Implies Connected)**.** *Let $G = (V, E)$ be a connected graph. Let $H = (V, E_H)$ be a subgraph of G that is maximally connected w. r. t. G. Then H is a connected graph.*

Proof. Let $a, b \in V$ be arbitrary nodes. Since G is connected, there is a path p from a to b in G by the definition of connected graphs (2.1.5). Due to the fact that H is maximally connected w. r. t. G, there is a path p from a to b in H by the definition of maximally connected graphs (2.1.6). Thus, H is connected by definition. □

Lemma 2.4.16 (Connected Minimum Spanning Forest Implies Tree)**.** *Let F be a minimum spanning forest of G. If F is connected, F is a minimum spanning tree of G.*

Proof. F is a forest and a subgraph of G by the definitions of minimum spanning forests (2.2.2) and spanning forests (2.2.1). As F is also connected, F is a spanning tree of G by the definitions of trees (2.1.8) and spanning trees (2.2.4).

If a graph T is connected, then it is also maximally connected w. r. t. any graph U that has the same nodes by the definition of maximally connected graphs (2.1.6), since all pairs of nodes in T are connected by paths by the definition of connected graphs (2.1.5), i. e. especially all pairs of nodes that are connected by a path in U are connected by a path in T.

Therefore, a spanning tree is always a spanning forest by the definitions of trees (2.1.8), spanning trees (2.2.4) and spanning forests (2.2.1). Thus, due to the fact that F is a minimum spanning forest of G and a spanning tree, F is a minimum spanning tree of G by the definition of minimum spanning trees (2.2.5). Otherwise, if there was a spanning tree T of G that has a smaller edge weight than F, T would also be a spanning forest of G and then this is in contradiction with the fact that H is a minimum spanning forest of G by the definition of minimum spanning forests (2.2.2). □

Lemma 2.4.17 (Minimum Spanning Forest Exists for Updated Forest)**.** *Let $G = (V, E)$, let $H = (V, E_H)$ be a subgraph of G. Let $F = (V, E_F)$ be a minimum spanning forest of G s. t. H is a subgraph of F. Let $E' \subseteq E$ be a set of edges that are connected in H*

(i. e. $\forall (x, w', y) \in E'$. $\exists$ path p from x to y in H). Let $(a, w, b) \in E$ an edge that is not connected yet, i. e. there is no path from a to b in H. The weights of all edges in $E \setminus E'$ are greater or equal to the weight of (a, w, b), i. e. $\forall (x, w', y) \in E \setminus E'$. $w \leq w'$.

Then there is a minimum spanning forest F' of G s. t. $H' = (V, E_{H'})$ with $E_{H'} = E_H \cup \{(a, w, b)\}$ is a subgraph of F'.

Proof. If $(a, w, b) \in E_F$, then H' is a subgraph of F by the definition of subgraphs (2.1.9). Thus, F fulfills the property for H'.

So we assume $(a, w, b) \notin E_F$.

Due to the fact that $(a, w, b) \in E$, $[(a, w, b)]$ is a path from a to b in G. Hence, there is a path p from a to b in F by the definition of maximally connected graphs (2.1.6), as F is maximally connected w. r. t. G by the definitions of minimum spanning forests (2.2.2) and spanning forests (2.2.1).

We will apply lemma 2.4.12 to F and H: F is a forest by definitions of minimum spanning forests and spanning forests, H is a subgraph of F, p is a path from a to b in F and there is no path from a to b in H. With the lemma we have:

There is an edge $(x, w', y) \in E_F \setminus E_H$ s. t. there is no path from a to b in $F_{\text{del}} = (V, E_{F_{\text{del}}})$ with $E_{F_{\text{del}}} = E_F \setminus \{(x, w', y)\}$ and there is a path from x to y in $F' = (V, E_{F'})$ with $E_{F'} = E_F \setminus \{(x, w', y)\} \cup \{(a, w, b)\}$.

We will prove that F' is a minimum spanning forest of G and H' is a subgraph of F'.

Since H is a subgraph of F and $(x, w', y) \notin E_H$, we have $E_H \subseteq E_F \setminus \{(x, w', y)\} = E_{F_{\text{del}}}$, i. e. H is a subgraph of F_{del} by the definition of subgraphs (2.1.9). Then $E_{H'} = E_H \cup \{(a, w, b)\} \subseteq E_F \setminus \{(x, w', y)\} \cup \{(a, w, b)\} = E_{F'}$. Hence, H' is a subgraph of F'.

Due to the fact that $(a, w, b) \in E$ and F is a subgraph of G by the definitions of minimum spanning forests and spanning forests, F' is a subgraph of G as well by the definition of subgraphs.

As F is a forest and F_{del} consits of the same edges without the edge (x, w', y), F_{del} is also a forest by the definition of forests (2.1.7). With the fact that there is no path from a to b in F_{del} we get that F' is a forest as well by lemma 2.4.7.

Since F is a subgraph of G and is maximally connected w. r. t. G and since $(a, w, b) \in E$, the graph $F_{\text{add}} = (V, E_{F_{\text{add}}})$ with $E_{F_{\text{add}}} = E_F \cup \{(a, w, b)\}$ is maximally connected w. r. t. G by lemma 2.4.13. From $(a, w, b) \in E$, we can also infer that F_{add} is a subgraph of G by the definition of subgraphs. Moreover, $(x, w', y) \neq (a, w, b)$, as $(a, w, b) \notin E_F$ and $(x, w', y) \in E_F \setminus E_H$. Thus, $E_{F_{\text{add}}} \setminus \{(x, w', y)\} = E_F \cup \{(a, w, b)\} \setminus \{(x, w', y)\} =$

$E_F \setminus \{(x, w', y)\} \cup \{(a, w, b)\} = E_{F'}$, i. e. $(V,\ E_{F_{\text{add}}} \setminus \{(x, w', y)\}) = F'$. With the fact that there is a path from x to y in F' we get that F' is maximally connected w. r. t. G by lemma 2.4.14.

Therefore, F' is a spanning forest by the definition of spanning forests (2.2.1).

Due to the fact that F is a forest and $(x, w', y) \in E_F$, there is no path from x to y in F_{del} by the definition of forests. Since H is a subgraph of F_{del}, there is no path from x to y in H either. Otherwise, such a path would be also valid in F_{del} by lemma 2.4.4.

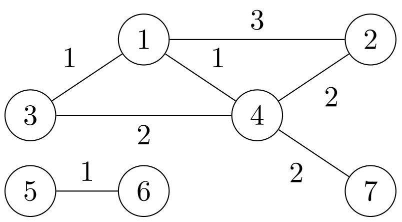
Then $(x, w', y) \notin E'$ by assumption. As F is a subgraph of G and $(x, w', y) \in E_F$, $(x, w', y) \in E \setminus E'$ by the definition of subgraphs (2.1.9). Hence, $w \leq w'$ by assumption. Then the total edge weight of F' is less or equal to the total edge weight of F, since F' contains the same edges as F with the edge (x, w', y) removed and the (a, w, b) added instead. Hence, F' is a minimum spanning forest by the definition of minimum spanning forests (2.2.2), as it is a spanning forest and F is a minimum spanning forest. □

3 Specification and Verification of the Kruskal Algorithm

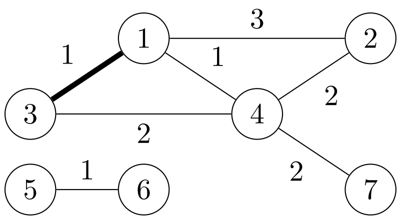
3.1 The Kruskal Algorithm

The Kruskal algorithm [4, 5] is a greedy algorithm, it considers the edges in increasing order of their weight. We start with an empty forest, i. e. with all nodes from the input graph, but no edges. For each considered edge the algorithm checks whether the edge creates a cycle in the current forest. If not, the edge can be safely added to the forest, otherwise the edge is not added. When all edges have been considered, we get a spanning forest of the initial graph. We will prove that this spanning forest is optimal. First we apply the algorithm to our example graph:

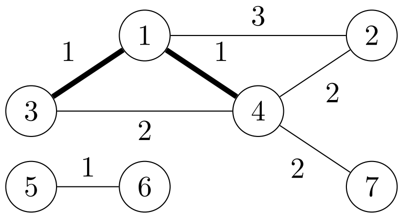
Example 7 (Kruskal Algorithm)**.** We take the graph $G = (V, E)$ from example 2. We sort the edges in G by weight and get the following order: $(1, 1, 3)$, $(1, 1, 4)$, $(5, 1, 6)$, $(2, 2, 4)$, $(3, 2, 4)$, $(4, 2, 7)$, $(1, 3, 2)$. Initially, we start with an empty forest $H = (V,\ E_H := \{\})$.



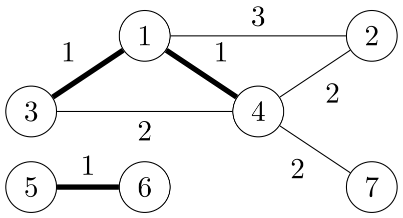
Edge $(1, 1, 3)$**:**
Since the graph H is empty, the edge $(1, 1, 3)$ does not create a cycle. Hence, we add it to H.



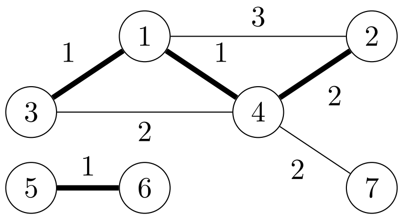
Edge $(1, 1, 4)$**:**
In H the nodes 1 and 4 are not connected, so $(1, 1, 4)$ can be safely added to H without violating the forest property.



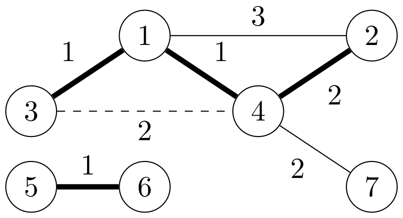
Edge $(5, 1, 6)$**:**
The nodes 5 and 6 are not connected in H, so $(5, 1, 6)$ can be safely added to H.



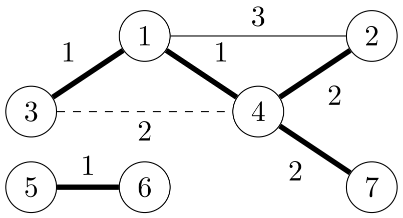
Edge $(2, 2, 4)$**:**
The nodes 2 and 4 are not connected in H, so $(2, 2, 4)$ can be safely added to H.



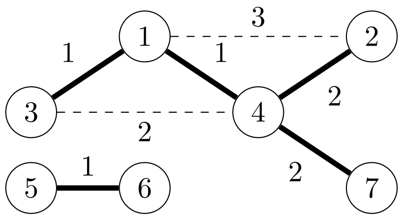
Edge $(3, 2, 4)$**:**
The nodes 3 and 4 are connected in H by the path $[(3, 1, 1), (1, 1, 4)]$, so we do not add $(3, 2, 4)$ to H.



Edge $(4, 2, 7)$**:**
The nodes 4 and 7 are not connected in H, so $(4, 2, 7)$ can be safely added to H.



Edge $(1, 3, 2)$**:**
The nodes 1 and 2 are connected in H by the path $[(1, 1, 4), (4, 2, 2)]$, so we do not add $(1, 3, 2)$ to H.



There are no more edges to consider, so $H = (V, E_H)$ with $E_H = \{(1, 1, 3), (1, 1, 4), (5, 1, 6), (2, 2, 4), (4, 2, 7)\}$ is a minimum spanning forest of G.

3.2 Specification

This is the formal specification of the described algorithm (below I will explain the steps in detail):

Algorithm 3.2.1: Kruskal Algorithm

```
   Input  : finite, weighted graph $G = (V, E)$
   Output: a minimum spanning forest $H$ of $G$
 1 $H := (V,\ E_H := \{\});$
 2 let $l$ be a list of the edges in $G$ (set $l = E$) which is sorted by increasing edge
     weight;
 3 foreach $(a, w, b)$ in $l$ do
 4     assert $H$ is a subgraph of $G$;
 5     assert $a, b \in V$;
 6     if $\neg\ \exists$ path $p$ from $a$ to $b$ in $H$ then
 7         assert $(a, w, b) \in E \setminus E_H$;
 8         $E_H \leftarrow E_H \cup \{(a, w, b)\}$;
 9     end
10 end
11 return $H$
```

The Kruskal algorithm only works for finite, weighted graphs. It computes a minimum spanning forest of the given graph (this will be proven in the correctness theorem 3.3.1).

First we create an empty forest H using the nodes of G (line 1).

We want to consider the edges of G in increasing order of the edge weights. Thus, we sort the edges by weight and construct a list of edges l with the sorted edges (line 2). For this specification we only need to know the result, that the edges are sorted, later we will define a specific sorting algorithm.

We then consider each edge in l individually in the sorted order using the foreach loop in line 3.

For each edge (a, w, b) we check whether there is already a path from a to b in our current forest H (line 6). If there is a path, the edge is not added to H, otherwise we would create a cycle in H. If there is no path, we will add the edge to H (line 8).

Finally, we have a few **assert** statements in the code (lines 4, 5 and 7). An **assert** statement is an annotation for the refinement process. It propagates knowledge from an abstract level to a more concrete level. The knowledge is usually easily provable in the step, in which we introduce the **assert** statement. In a later step, when we use the statement for the proof that the algorithm in that step refines the previous algorithm, we need it as explicit statement, if we cannot conclude the fact from the previous algorithm (without the **assert** statement).

Therefore, if we prove the correctness of the refinement step where the **assert** statement is introduced, we need to prove the **assert** statement. In our case we will do this in the correctness theorem 3.3.1.

For example the statement in line 7 states that the edge (a, w, b), which will be added to the forest H, is not already in H. This can be proven with the fact that there is no path from a to b in H. In the third refinement step (in theorem 4.3.3) we will use the statement to prove that the list of edges, which we use to model the forest H, is still distinct after adding the edge. Since the if condition will be different in the third refinement step, we need the **assert** statement to prove this fact.

3.3 Correctness Theorem

In this section the following correctness theorem for the Kruskal algorithm in 3.2.1 will be shown:

Theorem 3.3.1 (Correctness of the Kruskal Algorithm)**.** *Given a finite, weighted graph $G = (V, E)$ the Kruskal algorithm in 3.2.1 computes a minimum spanning forest H of G.*

To show this we will prove the following invariants for the loop:

Definition 3.3.2 (Loop Invariants for Algorithm 3.2.1)**.** For the input graph $G = (V, E)$, the current forest H and the set of edges $E' \subseteq E$ which were already considered in the loop we define the following loop invariants:

- **Forest:** H is a forest.
- **Subgraph:** H is a subgraph of G.
- **Previous Edges Connected:** $\forall (a, w, b) \in E'.\ \exists$ path p from a to b in H.
- **Minimum Spanning Forest Exists[1]:** There is a minimum spanning forest F of G s. t. H is a subgraph of F.

First we prove the invariants for the empty graph:

Lemma 3.3.3 (Loop Invariants for Empty Graph)**.** *The loop invariants in 3.3.2 initially hold for the input graph $G = (V, E)$, the empty forest $H = (V_H := V, E_H := \{\})$ and the case of no considered edges $E' = \{\}$.*

Proof. We prove the invariants separately:

- **Forest:** As the set of edges E_H is empty, the condition in the definition of forests (2.1.7) holds for every edge $(a, w, b) \in E_H$, thus H is a forest.
- **Subgraph:** H is a subgraph G by the definition of subgraphs (2.1.9), since $V_H = V$ and $E_H = \{\} \subseteq E$.
- **Previous Edges Connected:** The set of already considered edges E' is empty because the loop has not run yet. Therefore, the condition trivially holds.
- **Minimum Spanning Forest Exists:** Due to the fact that G is a finite, weighted graph, there is a minimum spanning forest $F = (V_F, E_F)$ of G by lemma 2.4.10. By the definitions of minimum spanning forests (2.2.2) and spanning forests (2.2.1), F is a subgraph of G. Then $V_F = V = V_H$ by the definition of subgraphs (2.1.9). As the empty set is a subset of every set, $E_H = \{\} \subseteq E_F$. Thus, H is a subgraph of F.

□

Lemma 3.3.4 (Loop Invariants Preserved by Loop Step)**.** *Assume the loop invariants in 3.3.2 hold at the beginning of a loop step for the input graph $G = (V, E)$, the current forest $H = (V, E_H)$ and the already considered edges $E' \subseteq E$. Let $(a, w, b) \in E$ be the edge that is considered in the current step. Then the invariants hold after the loop step for the updated graph H' and the edge set $E' \cup \{(a, w, b)\}$.*

Proof. We distinguish two cases:

Case 1 $\exists$ path p from a to b in H

In this case the graph is not changed, i. e. $H' = H$. Therefore, the invariants **Forest**, **Subgraph**, **Minimum Spanning Forest Exists** hold by assumption.

[1]This invariant is inspired by the loop invariant in the correctness proof by Cormen et al. [4, p. 625-628].

The invariant **Previous Edges Connected** holds for $E' \cup \{(a, w, b)\}$ as well, since it holds by assumption for all edges in E' and with the case assumption it also holds for (a, w, b).

Case 2 $\neg\, \exists$ path from a to b in H

In this case the edge (a, w, b) is added to the current graph: $H' = (V,\ E_H \cup \{(a, w, b)\})$.

- **Forest:** With the case assumption we get the goal by lemma 2.4.7.
- **Subgraph:** H' is a subgraph of G because $V_{H'} = V_H = V$ and $E_{H'} = E_H \cup \{(a, w, b)\} \subseteq E$ using the **Subgraph** property for H and $(a, w, b) \in E$ for the last step.
- **Previous Edges Connected:** The invariant for H holds for all edges in E' by assumption. By lemma 2.4.1, the same nodes are still connected in H'. Furthermore, the nodes a and b are connected in H' by the direct edge (a, w, b). Hence, the invariant holds for all edges in $E' \cup \{(a, w, b)\}$.
- **Minimum Spanning Forest Exists:** By **Subgraph**, H is a subgraph of G. By **Minimum Spanning Forest Exists**, we obtain a minimum spanning forest $F = (V, E_F)$ of G s. t. H is a subgraph of F. By **Previous Edges Connected**, $\forall (x, w', y) \in E'.\ \exists$ path p from x to y in H. As (a, w, b) is the next edge that is considered after the edges in E' and the edges are sorted by edge weight, $\forall (x, w', y) \in E \setminus E'.\ w \leq w'$.

  Then with $(a, w, b) \in E$ and the case assumption, we get the goal by lemma 2.4.17.

$\square$

Lemma 3.3.5 (Loop Invariants Imply Correctness). *Assume the loop invariants in 3.3.2 hold for the final graph $H = (V, E_H)$ and the set of considered edges $E' = E$. Then H is a minimum spanning forest of $G = (V, E)$.*

Proof. From the invariant **Minimum Spanning Forest Exists** we obtain a minimal spanning forest $F = (V, E_F)$ of G and H is a subgraph of F.

We prove that $F = H$. We already have $E_H \subseteq E_F$ by the definition of subgraphs (2.1.9), since H is a subgraph of F. So we need to prove $E_F \subseteq E_H$ which we will show by contradiction.

Assume $(a, w, b) \in E_F$ and $(a, w, b) \notin E_H$. Due to the fact that F is a subgraph of G by the definitions of minimum spanning forests (2.2.2) and spanning forests (2.2.1), $(a, w, b) \in E$ by the definition of subgraphs. By **Previous Edges Connected**, there is a path p from a to b in H. As $(a, w, b) \notin E_H$ and H is a subgraph of F, then H is also a subgraph of $F' = (V, E_F \setminus \{(a, w, b)\})$ by the definition of subgraphs. Then p is also a path from a to b in F' by lemma 2.4.4. This is in contradiction with the fact that F is a forest by the definition of forests (2.1.7).

Thus, $H = F$ is a minimum spanning forest of G. □

Proof of the Correctness Theorem 3.3.1. Initially the loop invariants in 3.3.2 hold by lemma 3.3.3. The loop preserves the invariants by induction on the number of loop steps and the lemma 3.3.4 for the induction step. Thus, the invariants hold at the end of the loop. With lemma 3.3.5 H is a minimum spanning forest of G.

It remains to show the correctness of the assert statements in the loop body. We know that the loop invariants in 3.3.2 hold at the beginning of a loop step for the current graph $H = (V, E_H)$ and the already considered edges set E'. Let $(a, w, b) \in E$ be the edge that is considered in the current step. Then we need to show three goals, one for each assert statement:

Goal 1 H is a subgraph of G.

The goal follows directly from the invariant **Subgraph**.

Goal 2 $a, b \in V$.

This follows directly from $(a, w, b) \in E$.

Goal 3 Assume there is no path from a to b in H, then $(a, w, b) \in E \setminus E_H$.

Proof by contradiction. Assume $(a, w, b) \notin E \setminus E_H$. Then $(a, w, b) \in E_H$ because $(a, w, b) \in E$. Now $[(a, w, b)]$ is a path from a to b in H which is in contradiction with the assumption. Hence, $(a, w, b) \in E \setminus E_H$.

□

4 Refinement Steps

4.1 First Refinement Step: Partial Equivalence Relation

To implement the Kruskal algorithm efficiently we need an efficient way to check if there is a path between two edges. Therefore, we need a data structure where we can efficiently check if two objects are in the same component and where we can efficiently join two components (when we add an edge to the graph). We can use a union find data structure for this.

4.1.1 Specification of the Union Find Data Structure

A *union find data structure*[2] consists of three functions: `init`, `compare` and `union`. The function `init` initializes the data structure with a set of objects. At the beginning each, object is in its own component. With `compare` we check whether two objects are in the same component. Finally, `union` joins the components of two given objects.

Lammich and Meis specified and verified a union find data structure in Isabelle\HOL [9]. The following description is based on their specification. The concrete data structure will be introduced in the fifth refinement step in chapter 4.5.1.

For the specification we use a relation as data structure: Two objects are related by the relation if and only if they are in the same component.

This relation is symmetric and transitive because the same component relation is symmetric and transitive. Thus, the relation is a partial equivalence relation:

Definition 4.1.1 (Partial Equivalence Relation)**.** A relation $R \subseteq A \times A$ is a partial equivalence relation iff

- R is symmetric, i. e. $\forall a, b.\ (a, b) \in R \longrightarrow (b, a) \in R$ and

[2]Union find data structures are called *disjoint set data structures* in some literature, since they represent disjoint components of elements.

- R is transitive, i. e. $\forall a, b, c.\ (a, b) \in R \wedge (b, c) \in R \longrightarrow (a, c) \in R$.

Remark. We do not have an equivalence relation here, as we do not have reflexivity. We do have reflexivity on the domain of the relation (see definition below) due to symmetry and transitivity: For $(a, b) \in R$, we have $(b, a) \in R$ by symmetry and $(a, a) \in R$ and $(b, b) \in R$ by transitivity. We do not have a predefined base set for the relation that could act as base set for the reflexivity relation, since we use the relation as data structure: All objects that are in the domain of the relation are stored in the data structure, all others are not stored. That is why we only have a partial equivalence relation.

Additionally, we need a function to get the objects in our relation:

Definition 4.1.2 (Domain of a Relation)**.** Let R be a relation. The domain of R is

$$\texttt{Domain}\ R = \{a.\ \exists b.\ (a, b) \in R\}.$$

Remark. Due to the fact that a partial equivalence relation is symmetric, the domain of a partial equivalence relation is the set of all elements in the relation, i. e. $R \subseteq \texttt{Domain}\ R \times \texttt{Domain}\ R$ for a partial equivalence relation R.

Using a partial equivalence relation as data structure we can give specifications of the three functions `per_init`, `per_compare` and `per_union`:

Algorithm 4.1.3: `per_init`: Initialize a new union find data structure from a set of objects.

```
Input  : set of objects $D$
Output: empty union find data structure with domain $\{0, \dots, n\}$
1 return $\{(i, i).\ i \in D\}$
```

Algorithm 4.1.4: `per_compare`: Check whether two objects are in the same component.

```
Input  : union find data structure $R$, objects $a$ and $b$
Output: Boolean whether $a$ and $b$ are in the same component
1 return $(a, b) \in R$
```

Algorithm 4.1.5: `per_union`: Join two components.

```
Input  : union find data structure $R$, objects $a$ and $b$
Output: updated $R$ with joined components
1 return $R \cup \{(x, y).\ (x, a) \in R \wedge (y, b) \in R\} \cup \{(y, x).\ (x, a) \in R \wedge (y, b) \in R\}$
```

We will prove that `per_init` produces a partial equivalence relation and `per_union` preserves it.

Lemma 4.1.6 (Partial Equivalence `per_init`)**.** `per_init` D *is a partial equivalence relation with domain* D*.*

Proof. Let $R =$ `per_init` D.

R is symmetric: Let $(a, b) \in R$, then $a = b$ by the definition of `per_init` (4.1.3). Thus, $(b, a) \in R$.

R is transitive: Let $(a, b) \in R$ and $(b, c) \in R$, then $a = b = c$ by the definition of `per_init`. Thus, $(a, c) \in R$.

Hence, R is a partial equivalence relation by the definition of partial equivalence relations (4.1.1).

Moreover, `Domain` $R = D$ by the definitions of `per_init` and `Domain` (4.1.2). $\square$

Lemma 4.1.7 (Partial Equivalence `per_union`)**.** *Let R be a partial equivalence relation with domain D. Then* `per_union` $R\ a\ b$ *is a partial equivalence relation with domain D.*

Proof. Let $R' =$ `per_union` $R\ a\ b$. Then $R' = R \cup \{(x, y).\ (x, a) \in R \wedge (y, b) \in R\} \cup \{(y, x).\ (x, a) \in R \wedge (y, b) \in R\}$ by the definition of `per_union` (4.1.5).

R' is symmetric: Let $(x, y) \in R'$, then $(x, y) \in R \vee ((x, a) \in R \wedge (y, b) \in R) \vee ((y, a) \in R \wedge (x, b) \in R)$. By symmetry of R and commutativity of $\vee$, we have $(y, x) \in R \vee ((y, a) \in R \wedge (x, b) \in R) \vee ((x, a) \in R \wedge (y, b) \in R)$. Then $(y, x) \in R$ by definition of R'.

R' is transitive: Let $(x, y) \in R'$ and $(y, z) \in R'$, then $(x, y) \in R \vee ((x, a) \in R \wedge (y, b) \in R) \vee ((y, a) \in R \wedge (x, b) \in R)$ and $(y, z) \in R \vee ((y, a) \in R \wedge (z, b) \in R) \vee ((z, a) \in R \wedge (y, b) \in R)$. Then $(x, z) \in R \vee ((x, a) \in R \wedge (z, b) \in R) \vee ((z, a) \in R \wedge (x, b) \in R)$ by symmetry and transitivity of R. Hence, $(x, z) \in R'$ by definition of R'.

Thus, R' is a partial equivalence relation by the definition of partial equivalence relations (4.1.1).

Moreover, `Domain` $R' =$ `Domain` $R = D$ by the definitions of R' and `Domain` (4.1.2). $\square$

4.1.2 The Refined Algorithm

Now we can use the union find data structure to refine the Kruskal algorithm in 3.2.1 (the changed lines are marked in gray):

Algorithm 4.1.8: Refined Kruskal Algorithm

```
Input  : finite, weighted graph $G = (V, E)$
Output: a minimum spanning forest $H$ of $G$
1  $uf :=$ per_init $V$;
2  $H := (V,\ E_H := \{\})$;
3  let $l$ be a list of the edges in $G$ (set $l = E$) which is sorted by increasing edge
     weight;
4  foreach $(a, w, b)$ in $l$ do
5      assert $a, b \in$ Domain $uf$;
6      if $\neg$ per_compare $uf\ a\ b$ then
7          $uf \leftarrow$ per_union $uf\ a\ b$;
8          assert $(a, w, b) \in E \setminus E_H$;
9          $E_H \leftarrow E_H \cup \{(a, w, b)\}$;
10     end
11 end
12 return $H$
```

4.1.3 Refinement Theorem

Theorem 4.1.9 (First Refinement Step)**.** *The Kruskal algorithm in 4.1.8 refines the algorithm in 3.2.1 under the identity as input and output relation.*

We introduce a relation between the tuple (uf, H) (for a union find data structure uf and a graph H) and the graph H. For this we first need another definition:

Definition 4.1.10 (Corresponding Union Find Data Structure)**.** A corresponding union find data structure uf for a graph $H = (V, E_H)$ is a union find data structure uf where two nodes $a, b \in V$ are in the same component in uf iff a and b are connected by a path in H:

$$\forall a, b \in V.\ \texttt{per\_compare}\ uf\ a\ b \longleftrightarrow (\exists \text{ path } p \text{ from } a \text{ to } b \text{ in } H)$$

Now we can define the relation:

Definition 4.1.11 (Union Find Graph Relation)**.** Let `uf_graph_rel` be the relation

$$\texttt{uf\_graph\_rel} = \{((uf, H),\ H').\ H = H' \wedge \texttt{uf\_graph\_invar}\ uf\ H\}$$

where `uf_graph_invar` uf H is defined as:

- uf is a partial equivalence relation,
- `Domain` $uf = V$ (for $H = (V, E)$) and
- uf is a corresponding union find data structure for H.

We first prove a helping lemma:

Lemma 4.1.12 (Corresponding Union Find Data Structure for Updated Graph)**.** *Let $H = (V, E_H)$ be a graph. Let uf be a corresponding union find data structure for H where uf is a partial equivalence relation with* `Domain` $uf = V$*. Let $a, b \in V$.*

Then `per_union` uf a b *is a corresponding union find data structure for the graph $H_{add} = (V, E_{H_{add}})$ with $E_{H_{add}} = E_H \cup \{(a, w, b)\}$.*

Proof. By the definition of corresponding union find data structures (4.1.10), we need to prove that for $x, y \in V$: `per_compare` (`per_union` uf a b) x y $\longleftrightarrow$ there is a path p from x to y in H_{add}.

We show the two directions separately:

"$\longrightarrow$": Assume `per_compare` (`per_union` uf a b) x y.

Then $(x, y) \in$ `per_union` uf a b by the definition of `per_compare` (4.1.4). Moreover, due to the fact that uf is a partial equivalence relation, uf is symmetric by the definition of partial equivalence relations (4.1.1). Then $(x, y) \in uf$ or $(x, a) \in uf \wedge (b, y) \in uf$ or $(x, b) \in uf \wedge (a, y) \in uf$ by the definition of `per_union` (4.1.5).

As uf is a corresponding union find data structure for H, there is a path p from x to y in H or there are paths p' and q' between x and a resp. b and y in H or there are paths p'' and q'' between x and b resp. a and y in H.

By lemma 2.4.1, these paths are also valid paths in H_{add}. Moreover, $[(a, w, b)]$ is a valid path from a to b in H_{add} and $[(b, w, a)]$ a path from b to a in H_{add}, since $(a, w, b) \in E_H \cup \{(a, w, b)\}$.

Then the paths p or (p', $[(a, w, b)]$ and q') or (p'', $[(b, w, a)]$ and q'') together form a path from x to y in H_{add}.

"$\longleftarrow$": Assume there is a path p from x to y in H_{add}.

Proof by case distinction:

Case 1 $(a, w, b) \in \texttt{set}\ p \vee (b, w, a) \in \texttt{set}\ p$

By lemma 2.4.5 for H_{add}, we have for $u, u' \in \{a, b\}$ that there is a path p' from x to u in H_{add} with $(a, w, b) \notin \texttt{set}\ p' \wedge (b, w, a) \notin \texttt{set}\ p'$ and there is a path p'' from u' to y in H_{add} with $(a, w, b) \notin \texttt{set}\ p'' \wedge (b, w, a) \notin \texttt{set}\ p''$.

By lemma 2.4.3, p' and p'' are also valid paths in H.

As uf is a corresponding union find data structure for H, per_compare $uf\ x\ u$ and per_compare $uf\ u'\ y$. By the definitions of per_compare (4.1.4) and per_union (4.1.5), we get per_compare (per_union $uf\ a\ b$) $x\ u$ and per_compare (per_union $uf\ a\ b$) $u'\ y$.

$(a, b) \in$ per_union $uf\ a\ b$ by the definition of per_union and reflexivity on the domain of uf by the definition of partial equivalence relations (4.1.1), since uf is a partial equivalence relation. Then per_compare (per_union $uf\ a\ b$) $u\ u'$ by the definition of per_compare, due to the fact that $u, u' \in \{a, b\}$ and per_union $uf\ a\ b$ is reflexive on the domain and symmetric by the definition of partial equivalence relations because it is a partial equivalence relation by lemma 4.1.7.

By the definition of per_compare and the transitivity of per_union $uf\ a\ b$ from the definition of partial equivalence relations, we have the goal, i. e. per_compare (per_union $uf\ a\ b$) $x\ y$.

Case 2 $(a, w, b) \notin \texttt{set}\ p \wedge (b, w, a) \notin \texttt{set}\ p$

Then p is also a path from x to y in H by lemma 2.4.3. As uf is a corresponding union find data structure for H, per_compare $uf\ x\ y$. By the definitions of per_compare (4.1.4) and per_union (4.1.5), we get per_compare (per_union $uf\ a\ b$) $x\ y$.

□

Proof of the First Refinement Step Theorem 4.1.9. When we unfold the definitions of algorithm 4.1.8 and algorithm 3.2.1 and use the relation uf_graph_rel (definition 4.1.11), we get the following proof goals:

Goal 1 ((`per_init` V, $(V, \{\})$), $(V, \{\})) \in$ `uf_graph_rel`.

By lemma 4.1.6, `per_init` V is a partial equivalence relation with the domain V. With $(V, \{\}) = (V, \{\})$ it remains to be proven that `per_init` V is a corresponding union find data structure for $(V, \{\})$. Let $a, b \in V$, then we need to prove that `per_compare` (`per_init` V) a b iff there is a path from a to b in $(V, \{\})$.

Since the edge set is empty, there is a path from a to b iff $a = b$. As $a, b \in V$, $a = b$ iff `per_compare` (`per_init` V) a b by the definitions of `per_init` (4.1.3) and `per_compare` (4.1.4).

Goal 2 Assume $((uf, H),\ H') \in$ `uf_graph_rel` with $H = (V, E_H)$ and $a, b \in V$, then $a, b \in$ `Domain` uf.

By the definition of `uf_graph_rel` (4.1.11), `Domain` $uf = V$. Thus, the conclusion follows from the assumption $a, b \in V$.

Goal 3 Assume $((uf, H),\ H') \in$ `uf_graph_rel` and $a, b \in V$, then `per_compare` uf a b iff there is a path p from a to b in H'.

The goal follows directly from the definition of `uf_graph_rel` (4.1.11) and the definition of corresponding union find data structures (4.1.10).

Goal 4 Let $H = (V, E_H)$ and $H' = (V, E_{H'})$, assume $((uf, H),\ H') \in$ `uf_graph_rel` and $(a, w, b) \in E$,
let $H_{\text{add}} = (V, E_H \cup \{(a, w, b)\})$ and $H'_{\text{add}} = (V, E_{H'} \cup \{(a, w, b)\})$,
then ((`per_union` uf a b, H_{add}), $H'_{\text{add}}) \in$ `uf_graph_rel`.

By definition of `uf_graph_rel` (4.1.11), we have that $H = H'$, uf is a partial equivalence relation with domain V and uf is a corresponding union find data structure for H.

Therefore, $H_{\text{add}} = H'_{\text{add}}$ and by lemma 4.1.7 `per_union` uf a b is a partial equivalence relation with domain V.

Moreover, `per_union` uf a b is a corresponding union find data structure for H_{add} by lemma 4.1.12.

Goal 5 Assume $((uf, H),\ H') \in$ `uf_graph_rel`, then $H = H'$.

The goal follows directly from the definition of `uf_graph_rel` (4.1.11).

□

4.2 Second Refinement Step: Input Graph Representation as List

We introduce a new representation for graphs: a list of edges. We can reconstruct the set of nodes by selecting the incident nodes for each edge. For a list of edges l the corresponding graph G is

$$G = (V := \{a.\ \exists w, b.\ (a, w, b) \in \texttt{set}\ l\} \cup \{b.\ \exists a, w.\ (a, w, b) \in \texttt{set}\ l\},\ E := \texttt{set}\ l).$$

In order to get a sorted list of edges we can now sort the list of edges by an arbitrary sorting algorithm. For our use case we will use the quicksort algorithm, since it is efficient in practice. The quicksort algorithm was formalized and proven to be correct in [7]. The formalization implements a function `quicksort_by_rel` that takes a relation and a list. It sorts the list under this relation. In our case this relation is the $\leq$ relation on the weights of two edges.

4.2.1 The Refined Algorithm

Algorithm 4.2.1: Refined Kruskal Algorithm

```
Input  : list of edges $l$ forming a graph $G = (V, E)$
Output: a minimum spanning forest $H$ of $G$
1  $uf = \texttt{per\_init}\ V$;
2  $H = (V,\ E_H := \{\})$;
3  $l \leftarrow \texttt{quicksort\_by\_rel}\ (\lambda(a, w, b)\ (a', w', b').\ w \leq w')\ l$;
4  foreach $(a, w, b)$ in $l$ do
5      assert $a, b \in \texttt{Domain}\ uf$;
6      if $\neg$ per_compare $uf\ a\ b$ then
7          $uf \leftarrow \texttt{per\_union}\ uf\ a\ b$;
8          assert $(a, w, b) \in E \setminus E_H$;
9          $E_H \leftarrow E_H \cup \{(a, w, b)\}$;
10     end
11 end
12 return $H$
```

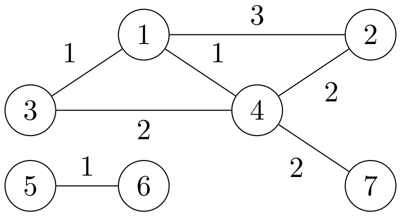
4.2.2 Refinement Theorem

We introduce a relation between the edge list representation l and the graph G:

Definition 4.2.2 (List Graph Relation)**.** Let `lst_graph_rel` be the relation

$$
\begin{aligned}
&\texttt{lst\_graph\_rel} = \{(l, G).\ G = (\\
&\quad V := \{a.\ \exists w, b.\ (a, w, b) \in \texttt{set}\ l\} \cup \{b.\ \exists a, w.\ (a, w, b) \in \texttt{set}\ l\}, \\
&\quad E := \texttt{set}\ l \\
&)\}.
\end{aligned}
$$

Example 8 (List Graph Relation)**.** We take the graph $G = (V, E)$ from example 2:



Then $V = \{1, 2, 3, 4, 5, 6, 7\}$ and $E = \{(1, 3, 2), (1, 1, 3), (1, 1, 4), (2, 2, 4), (3, 2, 4), (5, 1, 6), (4, 2, 7)\}$. Hence, the list $l = [(4, 2, 7), (1, 1, 3), (2, 2, 4), (1, 1, 4), (5, 1, 6), (1, 3, 2), (3, 2, 4), (1, 1, 3)]$ is a representation of G, i. e. $(l, G) \in$ `lst_graph_rel`.

Theorem 4.2.3 (Second Refinement Step)**.** *The Kruskal algorithm in 4.2.1 refines the algorithm in 4.1.8 under the input relation* `lst_graph_rel` *and the identity as the output relation.*

Proof. Assume $(l, G) \in$ `lst_graph_rel` with $G = (V, E)$.

We need to prove that `set` $l = E$ and `quicksort_by_rel` for the given relation $(\lambda(a, w, b)\ (a', w', b').\ w \leq w')$ returns a list that is sorted by increasing edge weight.

`set` $l = E$ holds by the definition of `lst_graph_rel` (4.2.2).

The relation $(\lambda(a, w, b)\ (a', w', b').\ w \leq w')$ is the $\leq$ relation on the weights of two edges. By the correctness of `quicksort_by_rel` [7], the resulting list is sorted by increasing edge weight. □

4.3 Third Refinement Step: Result Graph Representation as List

In this step we use an edge list l_H to represent the resulting spanning forest. The initial forest is the empty list. New edges are inserted in front of the list with the cons operator #. The nodes of the forest are the nodes of the input graph G, so there is no need to save them explicitly.

4.3.1 The Refined Algorithm

Algorithm 4.3.1: Refined Kruskal Algorithm

```
Input  : list of edges $l$ forming a graph $G = (V, E)$
Output: list of edges $l_H$ forming a minimum spanning forest of $G$
1  $uf$ := per_init $V$;
2  $l_H := []$;
3  $l \leftarrow$ quicksort_by_rel $(\lambda(a, w, b)\ (a', w', b').\ w \le w')\ l$;
4  foreach $(a, w, b)$ in $l$ do
5      assert $a, b \in$ Domain $uf$;
6      if $\neg$ per_compare $uf\ a\ b$ then
7          $uf \leftarrow$ per_union $uf\ a\ b$;
8          $l_H \leftarrow (a, w, b)\#l_H$;
9      end
10 end
11 return $l_H$
```

4.3.2 Refinement Theorem

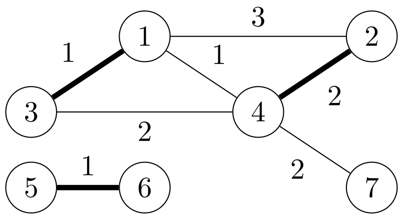
We introduce a relation between the edge list l_H and the graph representation H of the forest:

Definition 4.3.2 (List Subgraph Relation)**.** Let `lst_subgraph_rel` G (with $G = (V, E)$) be the relation

$$\texttt{lst\_subgraph\_rel}\ G = \{(l_H, H).\ H = (V, \texttt{set}\ l_H) \wedge \texttt{distinct}\ l\}$$

where `distinct` means that no two elements in the list are equal.

Example 9 (List Subgraph Relation)**.** We take the graph $G = (V, E)$ from example 2 and a subgraph H of G:



Let $H = (V,\ \{(1, 1, 3), (2, 2, 4), (5, 1, 6)\})$. Then the list $l_H = [(2, 2, 4), (5, 1, 6), (1, 1, 3)]$ is a representation of H, i. e. $(l_H, H) \in$ `lst_subgraph_rel`.

Theorem 4.3.3 (Third Refinement Step)**.** *Let $G = (V, E)$ be a graph s. t. $(l, G) \in$* `lst_graph_rel` *with the input list l.*

The Kruskal algorithm in 4.3.1 refines the algorithm in 4.2.1 under the identity as the input relation and the output relation `lst_subgraph_rel` *G.*

Proof. When we unfold the definitions of algorithm 4.3.1 and algorithm 4.2.1 and use the relation `lst_subgraph_rel` (definition 4.3.2), we get the following proof goals:

Goal 1 $([], (V, \{\})) \in$ `lst_subgraph_rel` G.

Follows directly from the definition of `lst_subgraph_rel` (4.3.2), since the empty list is distinct and `set` $[] = \{\}$ by the definition of `set` (2.1.10).

Goal 2 Assume $(l_H, H) \in$ `lst_subgraph_rel` G with $H = (V, E_H)$ and $(a, w, b) \in E \backslash E_H$, then $((a, w, b)\#l_H, H_{\text{add}}) \in$ `lst_subgraph_rel` G with $H_{\text{add}} = (V, E_H \cup \{(a, w, b)\})$.

From $(l_H, H) \in$ `lst_subgraph_rel` G we have `set` $l_H = E_H$ and `distinct` l_H by the definition of `lst_subgraph_rel` (4.3.2).

Then we have `set` $((a, w, b)\#l_H) =$ `set` $l_H \cup \{(a, w, b)\} = E_H \cup \{(a, w, b)\}$ by the definition of `set` (2.1.10). Thus, $H_{\text{add}} = (V,$ `set` $((a, w, b)\#l_H))$.

From $(a, w, b) \in E \backslash E_H$ we have $(a, w, b) \notin E_H =$ `set` l_H. Therefore, $(a, w, b)\#l_H$ is distinct because l_H is distinct and $(a, w, b) \notin$ `set` l_H. Hence, $((a, w, b)\#l_H, H_{\text{add}}) \in$ `lst_subgraph_rel` G.

□

4.4 Fourth Refinement Step: Restriction to Natural and Integer Numbers

Later we will implement the union find data structure using arrays. For this we need that our nodes are natural numbers, since we will use them as indices of the arrays. Moreover, we will restrict the weights to integer numbers to get an efficient, but not too restrictive representation.

Now an upper bound on the required size of the union find data structure can be obtained by using the highest number of a node. If we use this as the size, the data structure might be too large because not all indices between 0 and the highest number might be present, but it removes the restriction for the input that all nodes must be in a range of natural numbers without gaps. In our case we only have the restriction that we need natural numbers.

4.4.1 The Refined Algorithm

We first need an alternative specification of `per_init` that takes a size and not a domain as input and an auxiliary function that computes the maximum node in an edge list:

Algorithm 4.4.1: `per_init'`: Initialize a new union find data structure with a given size.

```
Input  : size $n \in \mathbb{N}$
Output: empty union find data structure of with domain $\{0, \dots, n-1\}$
1 return $\{(i,i).\ 0 \leq i < n\}$
```

Algorithm 4.4.2: `max_node`: Maximum node in a list of edges

```
Input  : list of edges $l$
Output: maximum node in $l$
1 return  $\max(\{0\} \cup \{a.\ \exists w, b.\ (a,w,b) \in \mathit{set}\ l\} \cup \{b.\ \exists a, w.\ (a,w,b) \in \mathit{set}\ l\})$
```

Algorithm 4.4.3: Refined Kruskal Algorithm

```
Input  : list of edges $l$ of type (nat × int × nat) list forming a graph $G$
Output: list of edges $l_H$ forming a minimum spanning forest of $G$
1  $uf$ := per_init' (max_node $l + 1$);
2  $l_H$ := [];
3  $l \leftarrow$ quicksort_by_rel ($\lambda(a,w,b)\ (a',w',b').\ w \leq w'$) $l$;
4  foreach $(a,w,b)$ in $l$ do
5      assert $a, b \in$ Domain $uf$;
6      if $\neg$ per_compare $uf$ $a$ $b$ then
7          $uf \leftarrow$ per_union $uf$ $a$ $b$;
8          $l_H \leftarrow (a,w,b)\#l_H$;
9      end
10 end
11 return $l_H$
```

4.4.2 Refinement Theorem

Theorem 4.4.4 (Fourth Refinement Step)**.** *The Kruskal algorithm in 4.4.3 refines the algorithm in 4.3.1 under the identity as input and output relation.*

We introduce a relation between the old and the new union find data structure:

Definition 4.4.5 (Partial Equivalence Relations Superset Relation)**.**
Let `per_supset_rel` be the following relation between two partial equivalence relations:

$$\texttt{per\_supset\_rel} = \{(p_1, p_2).\ p_1 \cap (\texttt{Domain}\ p_2 \times \texttt{Domain}\ p_2) = p_2\ \wedge$$
$$p_1 \setminus (\texttt{Domain}\ p_2 \times \texttt{Domain}\ p_2) \subseteq \mathit{Id}\}$$

where Id is the identity relation for an arbitrary type.

A new partial equivalence relation p_1 is related by `per_supset_rel` to an old relation p_2 iff p_1 restricted on the objects of p_2 is the relation p_2 and the remaining part of the relation p_1 is an identity relation.

Proof of the Fourth Refinement Step Theorem 4.4.4. When we unfold the definitions of algorithm 4.4.3 and algorithm 4.3.1 and use the relation `per_supset_rel` (definition 4.4.5), we get the following proof goals where $(l, G) \in$ `lst_graph_rel` with $G = (V, E)$:

Goal 1 (`per_init'` (`max_node` $l + 1$), `per_init` V) $\in$ `per_supset_rel`.

By the definitions of `per_init` (4.1.3), `per_init'` (4.4.1), `max_node` (4.4.2) and `lst_graph_rel` (4.2.2), we have `per_init'` (`max_node` l + 1) $= \{(i, i).\ 0 \leq i < \max(\{0\} \cup \{a.\ \exists w, b.\ (a, w, b) \in \texttt{set}\ l\} \cup \{b.\ \exists a, w.\ (a, w, b) \in \texttt{set}\ l\}) + 1\} = \{(i, i).\ 0 \leq i < \max(\{0\} \cup V) + 1\}$ and `per_init` $V = \{(i, i).\ i \in V\}$. By lemma 4.1.6, we have `Domain` (`per_init` V) $= V$.

By the definition of `per_supset_rel` we need to prove: $\{(i, i).\ 0 \leq i < \max(\{0\} \cup V) + 1\} \cap (V \times V) = \{(i, i).\ i \in V\}$ and $\{(i, i).\ 0 \leq i < \max(\{0\} \cup V) + 1\} \setminus (V \times V) \subseteq \mathit{Id}$. Trivially, $\{(i, i).\ 0 \leq i < \max(\{0\} \cup V) + 1\} \subseteq \mathit{Id}$. The second goal follows immediately. Due to the fact that $\mathit{Id} \cap (V \times V) = \{(i, i).\ i \in V\}$, we have $\{(i, i).\ 0 \leq i < \max(\{0\} \cup V) + 1\} \cap (V \times V) \subseteq \{(i, i).\ i \in V\}$.

So the other direction remains to be proven: $\{(i, i).\ 0 \leq i < \max(\{0\} \cup V) + 1\} \cap (V \times V) \supseteq \{(i, i).\ i \in V\}$. Let $(i, i) \in \{(i, i).\ i \in V\}$, i. e. $i \in V$. As the nodes are natural numbers by the Kruskal algorithm in 4.4.3, $0 \leq i$. Moreover, $i \leq \max(\{0\} \cup V)$ by the definition of max. Thus, we have $0 \leq i < \max(\{0\} \cup V) + 1$. Hence, $(i, i) \in \{(i, i).\ 0 \leq i < \max(\{0\} \cup V) + 1\}$.

Goal 2 Assume $(uf', uf) \in$ `per_supset_rel` and $a, b \in$ `Domain` uf, then $a, b \in$ `Domain` uf'.

By the definition of `per_supset_rel`, we have $uf \subseteq uf'$. Then `Domain` $uf \subseteq$ `Domain` uf' by the definition of `Domain` (4.1.2) and therefore $a, b \in$ `Domain` uf', since $a, b \in$ `Domain` uf.

Goal 3 Assume $(uf', uf) \in$ `per_supset_rel` and $a, b \in$ `Domain` uf, then `per_compare` $uf'\ a\ b =$ `per_compare` $uf\ a\ b$.

By the definition of `per_supset_rel`, uf' behaves like uf on `Domain` $uf \times$ `Domain` uf. With $a, b \in$ `Domain` uf we get the goal.

Goal 4 Assume $(uf', uf) \in$ `per_supset_rel` and $a, b \in$ `Domain` uf, then (`per_union` $uf'\ a\ b$, `per_union` $uf\ a\ b$) $\in$ `per_supset_rel`.

Let $I_{uf} =$ `Domain` $uf \times$ `Domain` uf.

Then $uf' \cap I_{uf} = uf$ and $uf' \setminus I_{uf} \subseteq Id$ by definition of `per_supset_rel`.

Let $uf_{\text{union}} =$ `per_union` $uf\ a\ b = uf \cup \{(x, y).\ (x, a) \in uf \wedge (y, b) \in uf\} \cup \{(y, x).\ (x, a) \in uf \wedge (y, b) \in uf\}$ and $uf'_{\text{union}} =$ `per_union` $uf'\ a\ b = uf' \cup \{(x, y).\ (x, a) \in uf' \wedge (y, b) \in uf'\} \cup \{(y, x).\ (x, a) \in uf' \wedge (y, b) \in uf'\}$ by the definition of `per_union` (4.1.5).

By lemma 4.1.7, `Domain` $uf_{\text{union}} =$ `Domain` uf.

The proof goal is: $(uf'_{\text{union}}, uf_{\text{union}}) \in \{(p_1, p_2).\ p_1 \cap$ (`Domain` $p_2 \times$ `Domain` $p_2) = p_2\ \wedge p_1 \setminus$ (`Domain` $p_2 \times$ `Domain` $p_2) \subseteq Id\}$, i. e. we have to show $uf'_{\text{union}} \cap I_{uf} = uf_{\text{union}}$ and $uf'_{\text{union}} \setminus I_{uf} \subseteq Id$.

With $uf' \cap I_{uf} = uf$ we can prove the first part:

$$\begin{aligned}
&uf'_{\text{union}} \cap I_{uf}\\
&\quad = uf' \cap I_{uf}\\
&\qquad \cup \{(x,y).\ (x,a) \in uf' \wedge (y,b) \in uf'\} \cap I_{uf}\\
&\qquad \cup \{(y,x).\ (x,a) \in uf' \wedge (y,b) \in uf'\} \cap I_{uf}\\
&\quad = uf' \cap I_{uf}\\
&\qquad \cup \{(x,y).\ (x,a) \in uf' \cap I_{uf} \wedge (y,b) \in uf' \cap I_{uf}\}\\
&\qquad \cup \{(y,x).\ (x,a) \in uf' \cap I_{uf} \wedge (y,b) \in uf' \cap I_{uf}\}\\
&\quad = uf \cup \{(x,y).\ (x,a) \in uf \wedge (y,b) \in uf\} \cup \{(y,x).\ (x,a) \in uf \wedge (y,b) \in uf\}\\
&\quad = uf_{\text{union}}.
\end{aligned}$$

For the second part we get:

$$\begin{aligned}
uf'_{\text{union}} \setminus I_{uf} \ = \ & uf' \setminus I_{uf} \cup \{(x,y).\ (x,a) \in uf' \wedge (y,b) \in uf'\} \setminus I_{uf}\\
& \cup \{(y,x).\ (x,a) \in uf' \wedge (y,b) \in uf'\} \setminus I_{uf}
\end{aligned}$$

We have a look at the set $X = \{(x,y).\ (x,a) \in uf' \wedge (y,b) \in uf'\} \setminus I_{uf}$. For tuples $(x,y) \in X$, we have $x \notin \texttt{Domain}\ uf$ or $y \notin \texttt{Domain}\ uf$, otherwise $(x,y) \in I_{uf} = (\texttt{Domain}\ uf \times \texttt{Domain}\ uf)$. W.l.o.g. we assume $x \notin \texttt{Domain}\ uf$. Then $(x,a) \in uf' \setminus I_{uf}$. Since $uf' \setminus I_{uf} \subseteq Id$, we have $x = a$. With $a \in \texttt{Domain}\ uf$ we get $x \in \texttt{Domain}\ uf$ which is in contradiction with $x \notin \texttt{Domain}\ uf$. Hence, there is no such tuple $(x,y) \in X$, i.e. X is empty. Analogously, $\{(y,x).\ (x,a) \in uf' \wedge (y,b) \in uf'\} \setminus I_{uf} = \{\}$.

Thus, $uf'_{\text{union}} \setminus I_{uf} = uf' \setminus I_{uf} \subseteq Id$.

$\square$

4.5 Fifth Refinement Step: Union Find Implementation

The last refinement step is automatically performed and proven correct by the Imperative Refinement Framework in Isabelle/HOL [8]. We have a look at the most important change in this step: the refinement of the union find data structure from the specification using partial equivalence relations to a concrete implementation on arrays.

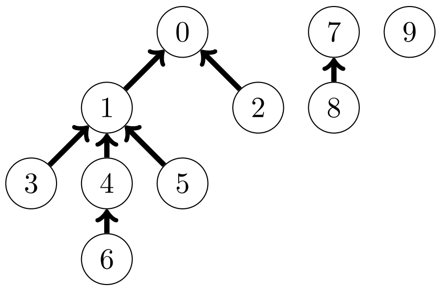
4.5.1 The Union Find Implementation

The following implementation of the union find data structure is based on [9]. The components of the data structure can be modeled as trees. Two objects that are related by the partial equivalence relation (i. e. they are in the same component) are connected by a path, i. e. they are in the same tree. Each tree has a distinguished root node. We call these nodes the representatives of the components. The component trees together form a forest. Then joining two components results in merging the trees.

To store these trees we store a reference for each object that points to the parent node. The root nodes have a parent reference to themselves. These references are stored in an array p (the parent array). Therefore, we need to restrict our objects to natural numbers between 0 and $n - 1$ (if we have n objects), so we can use the access by index on the array. Thus, the parent of an object i is stored in $p[i]$.

In another array s (the size array) we store the sizes of the trees. The size of a tree is an upper bound on the depth of the tree. To gain an efficient data structure we want to have trees that are as flat as possible. So with the size information we can decide what the new root will be when we join two trees: We will choose the root of the larger tree to be the new root of the joined tree, since it is probably the higher tree.

Example 10 (Union Find Data Structure).



The components are $\{0, 1, 2, 3, 4, 5, 6\}$, $\{7, 8\}$ and $\{9\}$ with the representatives 0, 7 and 9. The arrays p and s are:

i	0	1	2	3	4	5	6	7	8	9
$p[i]$	0	0	0	1	1	1	4	7	7	9
$s[i]$	7	5	1	1	2	1	1	2	1	1

Initially, when we construct a union find data structure of size n, $p[i] = i$ and $s[i] = 1$ for each object $i \in \{0, \ldots, n - 1\}$, since each objects is in its own component by the definition of `per_init` (4.1.3):

Algorithm 4.5.1: `uf_init`: Initialize a new union find data structure from a set of objects

```
Input  : number of objects $n$
Output: empty union find data structure
1 $p := [0, \ldots, n - 1]$;
```

```
2 $s := [1, \ldots, 1]$;
3 return $(s, p)$
```

Before we can define the function `uf_cmp`, we need a few auxiliary functions first, as we will not only check if the two objects are in the same component, but also update the parent references on the way such that they aim directly at the representative of the component. Thus, we gain shorter access times for future requests on these objects. Since we reduce the length of paths in the tree, this technique is called *path compression.*

The function `uf_rep_of` looks up the representative of the component of a given object (without path compression):

Algorithm 4.5.2: `uf_rep_of`: Look up the representative of the component of an object

```
Input  : union find parent array $p$, object $i$
Output: representative of the component of $i$
1 $n \leftarrow p[i]$;
2 if $n = i$ then
3     return $i$;
4 else
5     uf_rep_of $p$ $n$;
6 end
```

Assume we now have the representative *ci* of the component of an object i. Then we can go up the path from i to *ci* again, but this time we compress the path on our way, i. e. we set the parent of each node on the path to *ci*:

Algorithm 4.5.3: `uf_compress`: Compress the path from an object to the representative of its component

```
Input  : object $i$, representative $ci$ of the component of $i$, union find parent
         array $p$
Output: parent array with compressed paths
1 if $i \neq ci$ then
2     $ni \leftarrow p[i]$;
3     uf_compress $ni$ $ci$ $p$;
4     $p[i] \leftarrow ci$;
5 end
```

Just as the function `uf_rep_of`, the function `uf_rep_of_c` returns the representative of an object, but this time with path compression via `uf_compress`:

Algorithm 4.5.4: `uf_rep_of_c`: Look up the representative of the component of an object and perform path compression

```
Input  : union find parent array $p$, object $i$
Output: representative of the component of $i$
1 $ci \leftarrow$ uf_rep_of $p$ $i$;
2 uf_compress $i$ $ci$ $p$;
3 return $ci$;
```

Now we can implement the compare function `uf_cmp` with path compression:

Algorithm 4.5.5: `uf_cmp`: Check whether two objects are in the same component (and perform path compression)

```
Input  : union find data structure $(s, p)$, objects $i$ and $j$
Output: boolean whether $i$ and $j$ are in the same component
1 $n \leftarrow$ length $p$;
2 if $i \geq n \vee j \geq n$ then
3     return $False$;
4 else
5     $ci \leftarrow$ uf_rep_of_c $p$ $i$;
6     $cj \leftarrow$ uf_rep_of_c $p$ $j$;
7     return $ci = cj$;
8 end
```

Finally, we can implement the union function `uf_union`. Here we will use the size information to decide which representative will be the representative of the joined component (as described above):

Algorithm 4.5.6: `uf_union`: Join two components

```
Input  : union find data structure $(s, p)$, objects $i$ and $j$
Output: updated $(s, p)$ with joined components
1 $ci \leftarrow$ uf_rep_of $p$ $i$;
2 $cj \leftarrow$ uf_rep_of $p$ $j$;
3 if $ci = cj$ then
4     return $(s, p)$;
5 else
6     $si \leftarrow s[ci]$;
```

```
    $sj \leftarrow s[cj]$;
    **if** $si < sj$ **then**
        $p[ci] \leftarrow cj$;
        $s[cj] \leftarrow si + sj$;
        **return** $(s, p)$;
    **else**
        $p[cj] \leftarrow ci$;
        $s[ci] \leftarrow si + sj$;
        **return** $(s, p)$;
    **end**
**end**
```

4.5.2 The Final Algorithm

Algorithm 4.5.7: Final Kruskal Algorithm

```
**Input** : list of edges $l$ of type (nat × int × nat) list forming a graph $G$
**Output:** list of edges $l_H$ forming a minimum spanning forest of $G$
$uf$ := uf_init (max_node $l + 1$);
$l_H$ := [];
$l \leftarrow$ quicksort_by_rel $(\lambda(a, w, b)\ (a', w', b').\ w \leq w')\ l$;
**foreach** $(a, w, b)$ *in* $l$ **do**
    $cmp$ := uf_cmp $uf\ a\ b$;
    **if** $\neg\ cmp$ **then**
        $uf \leftarrow$ uf_union $uf\ a\ b$;
        $l_H \leftarrow (a, w, b)\#l_H$;
    **end**
**end**
**return** $l_H$
```

4.5.3 Refinement Theorem

Theorem 4.5.8 (Fifth Refinement Step). *The Kruskal algorithm in 4.5.7 refines the algorithm in 4.4.3 under the identity as input and output relation.*

Proof. The correctness follows immediately from the correctness of the refinement of the union find functions proven in [9]. □

4.6 Overall Correctness Theorems

Theorem 4.6.1 (Overall Correctness of the Final Kruskal Algorithm)**.** *Given a list of edges l forming a weighted graph $G = (V := \{a.\ \exists w, b.\ (a, w, b) \in \texttt{set}\ l\} \cup \{b.\ \exists a, w.\ (a, w, b) \in \texttt{set}\ l\},\ E := \texttt{set}\ l)$, the Kruskal algorithm in 4.5.7 computes a distinct list of edges l_H forming a minimum spanning forest $H = (V, E_H := \texttt{set}\ l_H)$ of G, i. e. the Kruskal algorithm in 4.5.7 solves the minimum spanning forest problem (cf. definition 2.2.3).*

Proof. From the refinement theorems 4.1.9, 4.2.3, 4.3.3, 4.4.4 and 4.5.8 we get that the Kruskal algorithm in 4.5.7 refines the original algorithm in 3.2.1 under the input relation `lst_graph_rel` and the output relation `lst_subgraph_rel` G. Finally, by the correctness theorem 3.3.1 and the definitions of `lst_graph_rel` (4.2.2) and `lst_subgraph_rel` (4.3.2) we have the correctness of algorithm 4.5.7. □

If we assume that the initial graph is connected, we can prove that the Kruskal algorithm constructs a minimum spanning tree:

Theorem 4.6.2 (Correctness of Kruskal Algorithm for Minimum Spanning Trees)**.** *Given a list of edges l forming a connected, weighted graph $G = (V := \{a.\ \exists w, b.\ (a, w, b) \in \texttt{set}\ l\} \cup \{b.\ \exists a, w.\ (a, w, b) \in \texttt{set}\ l\},\ E := \texttt{set}\ l)$, the Kruskal algorithm in 4.5.7 computes a distinct list of edges l_H forming a minimum spanning tree $H = (V, E_H := \texttt{set}\ l_H)$ of G, i. e. the Kruskal algorithm in 4.5.7 solves the minimum spanning tree problem (cf. definition 2.2.6).*

Proof. By theorem 4.6.1, the algorithm 4.5.7 constructs a list of edges l_H forming a minimum spanning forest $H = (V, E_H := \texttt{set}\ l_H)$ of G.

We will prove that H is also a minimum spanning tree of G.

By definitions of minimum spanning forests (2.2.2) and spanning forests (2.2.1), H is a subgraph of G that is maximally connected w. r. t. G. Due to the fact that G is connected, H is connected as well by lemma 2.4.15.

As H is a minimum spanning forest of G and H is connected, H is a minimum spanning tree of G by lemma 2.4.16. □

5 A Kruskal Algorithm for Minimum Spanning Trees

In chapter 3 and 4 I specified the Kruskal algorithm and verified that it constructs a minimum spanning forest (theorem 4.6.1). I decided to verify it as a minimum spanning forest algorithm instead of a minimum spanning tree algorithm, since in the first case we do not need the condition that the input graph is connected. Furthermore, we can easily prove that the output of the Kruskal algorithm is a minimum spanning tree if the input graph is connected (as proven in theorem 4.6.2).

Thus, the implemented algorithm is useful if we want to compute a minimum spanning forest or if we want to compute a minimum spanning tree and we know that the input graph is connected. For the case, if we need a minimum spanning tree and we do not know whether the input graph is connected, we will need another algorithm.

In this chapter I will specify and verify this alternative algorithm as a wrapper around the already verified Kruskal algorithm. The wrapper takes the computed minimum spanning forest and checks whether it is connected. If it is connected, it is a minimum spanning tree and we can return it. Otherwise, if it is not connected, the input graph is not connected, i. e. there is no minimum spanning tree. For the connectedness check I will use the fact that a forest is connected iff the number of edges in the forest is equal to the number of nodes minus one (lemma 5.3.2). The idea is that this check is more efficient than a connectedness check on the input graph which is in general not a forest. Thus, the provided algorithm is more efficient than the original Kruskal algorithm plus a connectedness check of the input graph.

I will call this algorithm *Kruskal algorithm for minimum spanning trees*, since it is basically a Kruskal algorithm with a connectedness check afterwards and returns minimum spanning trees.

5.1 Specification

First we specify the algorithm. The algorithm should return a minimum spanning tree if the given graph is connected, otherwise the algorithm should return nothing. We will use the option type with constructors `Some` x and `None` to model the output value.

Algorithm 5.1.1: Specification of the Kruskal Algorithm for Minimum Spanning Trees

```
Input  : finite graph $G = (V, E)$
Output: a minimum spanning tree $H$ of $G$ or nothing
1 if $G$ is connected then
2 |   let $H$ be a minimum spanning tree of $G$;
3 |   return Some $H$;
4 else
5 |   return None;
6 end
```

5.2 First Refinement Step

We refine the specification so that it first finds a minimum spanning forest (later we will use the Kruskal algorithm for this). Then it checks whether this forest is connected. It is connected iff the initial graph is connected (this will be proven in theorem 5.2.2).

Algorithm 5.2.1: Refined Kruskal Algorithm for Minimum Spanning Trees

```
Input  : finite graph $G = (V, E)$
Output: a minimum spanning tree $H$ of $G$ or nothing
1 let $H = (V_H, E_H)$ be a minimum spanning forest of $G$;
2 assert $H$ is a finite graph;
3 if $H$ is connected then
4 |   return Some $H$;
5 else
6 |   return None;
7 end
```

Theorem 5.2.2 (First Refinement Step)**.** *The Kruskal algorithm for minimum spanning trees in 5.2.1 refines the specification in 5.1.1 under the identity as the input and the* *output relation.*

Proof. Let $G = (V, E)$ be a finite graph.

Goal 1 Let $H = (V_H, E_H)$ be a minimum spanning forest of G, then H is a finite graph.

By the definitions of minimum spanning forests (2.2.2) and spanning forests (2.2.1), H is a subgraph of G, i. e. $V_H = V$ and $E_H \subseteq E$ by the definition of subgraphs (2.1.9). Due to the fact that G is a finite graph, V and E are finite by the definition of finite graphs (2.1.2). Then V_H and E_H are finite as well, i. e. H is a finite graph.

Goal 2 Let $H = (V_H, E_H)$ be a minimum spanning forest of G, then H is connected $\longleftrightarrow$ G is connected.

We show the two directions separately:

"$\longrightarrow$": Assume H is connected. Prove that G is connected.

By the definitions of minimum spanning forests (2.2.2) and spanning forests (2.2.1), H is a subgraph of G, i. e. $V_H = V$ by the definition of subgraphs (2.1.9).

Let $a, b \in V$. By the definition of connected graphs (2.1.5), we need to show that there is a path from a to b in G.

As H is connected and $a, b \in V = V_H$, there is a path from a to b in H. As H is a subgraph of G, there is a path from a to b in G by lemma 2.4.4.

"$\longleftarrow$": Assume G is connected. Prove that H is connected.

By the definitions of minimum spanning forests and spanning forests, H is a subgraph of G and H is maximally connected w. r. t. G. Since G is connected, H is connected as well by lemma 2.4.15.

Goal 3 Let H be a minimum spanning forest of G and let H be connected, then H is a minimum spanning tree of G.

The goal follows directly by lemma 2.4.16.

□

5.3 Second Refinement Step

Before we define a refined algorithm, we will introduce a function to compute whether a forest is connected.

5.3.1 Function to Compute Connectedness of a Forest

A tree, i. e. a connected forest by the definition of trees (2.1.8), has the property that the number of edges is equal to the number of nodes minus one. The other direction holds as well: If the property holds for a forest, then the forest is connected (cf. lemma 5.3.2). We define the function `is_connected` that uses the property to implement a connectedness check for forests:

Algorithm 5.3.1: `is_connected`: Check whether a forest is connected

```
   Input  : finite forest $F = (V_F, E_F)$
   Output: Boolean whether $F$ is connected
1  return $|E_F| = |V_F| - 1$;
```

Lemma 5.3.2 (Correctness of `is_connected`)**.** *Given a finite forest $F = (V_F, E_F)$ the function* `is_connected` *(5.3.1) computes whether F is connected.*

Since the proof is very long, we will not repeat it here. The whole proof can be found in the Isabelle sources.

As our algorithm should work on lists of edges, we first refine `is_connected`:

Algorithm 5.3.3: `is_connected_list`: Refined `is_connected` function that takes a list of edges

```
   Input  : list of edges $l$ forming a graph $G$ and
            list of edges $l_F$ forming a forest $F$
   Output: Boolean whether $F$ is connected
1  return length $l_F$ = length (remdups((map fst $l$) @
      (map (snd ∘ snd) $l$))) − 1;
```

`remdups` removes the duplicates of a list. Thus, `remdups` returns a distinct list and keeps the set of elements: `distinct` (`remdups` l') and `set` (`remdups` l') = `set` l' (for a list l').

`fst` and `snd` return the first resp. second element of a pair. In Isabelle/HOL all tuples are modeled as pairs, i. e. $(a, b, c) = (a, (b, c))$. Then `fst` $(a, b, c) = a$ and `snd` (`snd` $(a, b, c)) = c$.

Lemma 5.3.4 (Refinement of `is_connected`). *The function* `is_connected_list` *(5.3.3) refines the function* `is_connected` *(5.3.1) under the input relation* $\{((l, l_F), F).\ \exists G.\ (l, G) \in \texttt{lst\_graph\_rel} \land (l_F, F) \in \texttt{lst\_subgraph\_rel}\ G\}$ *and the identity as the output relation.*

Proof. We get the following goal, when we unfold the definitions of `is_connected` (5.3.1) and `is_connected_list` (5.3.3):

Let $G = (V, E)$ and $F = (V_F, E_F)$. Assume that $(l,\ G) \in \texttt{lst\_graph\_rel}$ and $(l_F,\ F) \in \texttt{lst\_subgraph\_rel}\ G$, then $\texttt{length}\ l_F = \texttt{length}\ (\texttt{remdups}\ ((\texttt{map fst}\ l)\ @\ (\texttt{map}\ (\texttt{snd} \circ \texttt{snd})\ l))) - 1$ iff $|E_F| = |V_F| - 1$.

By the definition of `lst_subgraph_rel` (4.3.2), l_F is distinct, $V_F = V$ and $E_F = \texttt{set}\ l_F$. Then the cardinality of E_F is equal to the length of l_F. Due to the fact that `remdups` returns a distinct list and $\texttt{set}\ (\texttt{remdups}\ l') = \texttt{set}\ l'$ for a list l', the cardinality of $\texttt{set}\ l'$ is equal to the length of $\texttt{remdups}\ l'$. As $V_F = V = \texttt{set}\ ((\texttt{map fst}\ l)\ @\ (\texttt{map}\ (\texttt{snd} \circ \texttt{snd})\ l))$ by the definition of `lst_graph_rel` (4.2.2), the cardinality of V_F is equal to $\texttt{length}\ (\texttt{remdups}\ ((\texttt{map fst}\ l)\ @\ (\texttt{map}\ (\texttt{snd} \circ \texttt{snd})\ l)))$. Thus, $\texttt{length}\ l_F = \texttt{length}\ (\texttt{remdups}\ ((\texttt{map fst}\ l)\ @\ (\texttt{map}\ (\texttt{snd} \circ \texttt{snd})\ l))) - 1$ iff $|E_F| = |V_F| - 1$. ☐

5.3.2 The Refined Algorithm and the Refinement Theorem

Like the final Kruskal algorithm the refined algorithm takes a list of edges with nodes as natural numbers and weights as integer numbers. For the connectedness check the previously defined function `is_connected_list` is used.

Algorithm 5.3.5: Refined Kruskal Algorithm for Minimum Spanning Trees

```
Input  : list of edges $l$ of type (nat × int × nat) list forming a graph $G$
Output: list of edges $l_H$ forming a minimum spanning tree of $G$ or nothing
1 $l_H \leftarrow$ kruskal $l$ (cf. algorithm 4.5.7);
2 if is_connected_list $l$ $l_H$ then
3     return Some $l_H$;
4 else
5     return None;
6 end
```

Theorem 5.3.6 (Second Refinement Step)**.** *The Kruskal algorithm for minimum spanning trees in 5.3.5 refines the algorithm in 5.2.1 under* `lst_graph_rel` *as the input relation and* $\{($`Some` l_F, `Some` $F)$. $(l_F,\ F) \in$ `lst_subgraph_rel`$\} \cup \{($`None`, `None`$)\}$[3] *as the output relation.*

Proof. If we unfold the definitions of algorithm 5.3.5 and algorithm 5.2.1, we get the following proof goals:

Goal 1 The Kruskal algorithm in 4.5.7 computes a list of edges l_F forming a minimum spanning forest of G.

This follows directly from the correctness theorem 4.6.1.

Goal 2 Assume $(l, G) \in$ `lst_graph_rel`, $(l_F, F) \in$ `lst_subgraph_rel` G and $F = (V, E_F)$ is a finite graph and a minimum spanning forest of G.
Then `is_connected_list` $l\ l_F$ iff F is a connected graph.

By the definitions of minimum spanning forests (2.2.2) and spanning forests (2.2.1), F is a forest. Since F is finite, by lemma 5.3.2, the function `is_connected` computes, whether F is connected. By lemma 5.3.4, `is_connected_list` refines `is_connected` under the input relation

$\{((l, l_F), F).\ \exists G.\ (l, G) \in$ `lst_graph_rel` $\wedge\ (l_F, F) \in$ `lst_subgraph_rel` $G\}$

and the identity as the output relation. Thus, `is_connected_list` $l\ l_F$ computes whether F is connected.

□

5.4 Overall Correctness Theorem

Finally, we can prove the correctness of the Kruskal algorithm for minimum spanning trees:

Theorem 5.4.1 (Overall Correctness of the Kruskal Algorithm for Minimum Spanning Trees)**.** *Given a list of edges l forming a weighted graph $G = (V := \{a.\ \exists w, b.\ (a, w, b) \in$ `set` $l\} \cup \{b.\ \exists a, w.\ (a, w, b) \in$ `set` $l\},\ E :=$ `set` $l)$, the Kruskal algorithm for minimum spanning trees in 5.3.5 computes a distinct list of edges l_H forming a minimum spanning tree $H = (V, E_H :=$ `set` $l_H)$ of G if G is connected. Otherwise, it returns nothing.*

[3]This relation is the relation `lst_subgraph_rel` lifted to the option type.

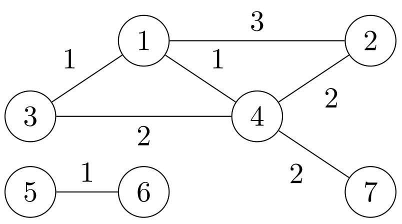
Proof. From the refinement theorems 5.2.2 and 5.3.6, we get that the Kruskal algorithm for minimum spanning trees in 5.3.5 refines the specification in 5.1.1 under the input relation `lst_graph_rel` and the output relation $\{(\texttt{Some}\ l_F, \texttt{Some}\ F).\ (l_F, F) \in$ `lst_subgraph_rel`$\} \cup \{(\texttt{None}, \texttt{None})\}$. The goal follows from the definition of the specification in 5.1.1. $\square$

6 Code Generation

The Imperative Refinement Framework [8] provides a code generation mechanism that generates ML code from the refined algorithm. I wrote command line programs in ML around the code of both algorithms, the Kruskal algorithm in 4.5.7 and the Kruskal algorithm for minimum spanning trees in 5.3.5.

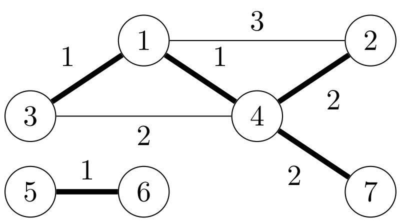
The programs take a list of edges as input and return the list of edges of a minimum spanning forest (resp. minimum spanning tree or nothing – for the second algorithm). The programs take a filename as an argument. This file should be a text file containing the edges – one in each line. The format of a line is $a\ b\ w$ for an edge (a, w, b).

Example 11 (Executable Program)**.** We take the graph $G = (V, E)$ from example 2:



Then the input for the graph G is

```
1 2 3
1 3 1
1 4 1
2 4 2
3 4 2
5 6 1
4 7 2
```



The output of the algorithm, i. e. the edges of the minimum spanning forest[4] are

```
4 7 2
2 4 2
1 3 1
5 6 1
1 4 1
```

[4]This is the same forest as in example 7. In fact, in this case the minimum spanning forest is unique.

As the edges are inserted at the beginning of the list, we can see the order in which the edges were added if we revert the list. First the edges with weight 1 are added and then the edges with weight 2, except $(3, 2, 4)$, since it creates a cycle. Finally, the edge $(1, 3, 2)$ is not added because the nodes 1 and 2 are already connected.

7 Conclusion

In this thesis I had a look at the Kruskal algorithm – an algorithm for minimum spanning trees and forests. I verified a specification of the Kruskal algorithm in Isabelle/HOL using loop invariants. Then I refined the specification step by step using the Imperative Refinement Framework to an imperative program. In this process I changed the interface for the input and output to lists of edges instead of sets for nodes and edges. To get an efficient implementation I used the union find data structure from [9] to check whether two nodes are connected. Additionally, I created a modified Kruskal algorithm that only returns minimum spanning trees. This alternative algorithm is useful if we explicitly need a tree and not a forest as outcome and we do not know whether the input graph is connected. Using the code generation of the framework I created executable command line programs for both algorithms.

As far as I know this is the first time the Kruskal algorithm was verified in a theorem prover. The union find data structure (as in [9]), for example, has been verified in Coq by Charguéraud and Pottier [3]. They also state that future work can verify the Kruskal algorithm, but this has not been done so far.

In this thesis I gave a verification of the correctness of the Kruskal algorithm. I refined it to an efficient algorithm using a union find data structure, but I did not prove the efficiency of the algorithm. This was beyond the scope of this thesis which was the verification of the algorithm and is open to future work. Moreover, one could verify the Prim algorithm – another algorithm for minimum spanning trees – which is similar to the Kruskal algorithm as shown in [4].

Bibliography

[1] Otakar Borůvka. O jistém problému minimálním. *Práce Moravské přírodovědecké společnosti*, 3:37–58, 1926.

[2] Otakar Borůvka. Příspěvek k řešení otázky ekonomické stavby elektrovodních sítí. *Elektronický Obzor*, 15:153–154, 1926.

[3] Arthur Charguéraud and François Pottier. Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. *Journal of Automated Reasoning*, Sep 2017.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[5] M.T. Goodrich, R. Tamassia, and M.H. Goldwasser. *Data Structures and Algorithms in Java.* Wiley, 6th edition, 2014.

[6] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[7] Peter Lammich. Automatic data refinement. *Archive of Formal Proofs*, October 2013. `http://isa-afp.org/entries/Automatic_Refinement.html`, Formal proof development.

[8] Peter Lammich. The imperative refinement framework. *Archive of Formal Proofs*, August 2016. `http://isa-afp.org/entries/Refine_Imperative_HOL.html`, Formal proof development.

[9] Peter Lammich and Rene Meis. A separation logic framework for imperative HOL. *Archive of Formal Proofs*, November 2012. `http://isa-afp.org/entries/Separation_Logic_Imperative_HOL.html`, Formal proof development.

[10] Jaroslav Nešetřil, Eva Milková, and Helena Nešetřilová. Otakar Borůvka on minimum spanning tree problem translation of both the 1926 papers, comments, history. *Discrete Mathematics*, 233(1):3–36, 2001.

[11] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994.