10. Minimal aufspannende Bäume

- Problemstellung
- Algorithmus von Prim
- Algorithmus von Kruskal

Problemstellung (1)

Definition

Sei G = (V,E) ein <u>ungerichteter, gewichteter Graph</u>.

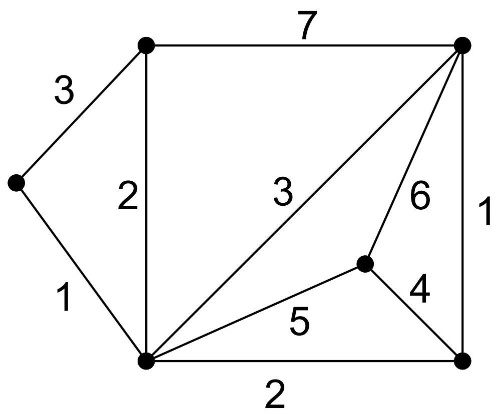
Dann ist B ein minimal aufspannender Baum, falls B folgende Eigenschaften erfüllt:

(1) B = (V, E') mit $E' \subseteq E$;
d.h. B ist ein Teilgraph von G mit gleicher Knotenmenge

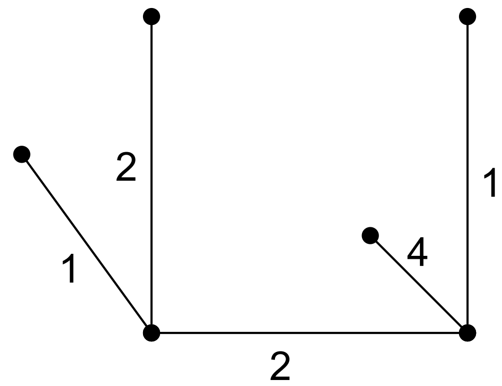
(2) B ist ein Baum;
d.h. ein azyklischer, zusammenhängender Graph

(3) Die Summe der Kantengewichte von B ist minimal;
d.h. es gibt keinen anderen Baum, der Eigenschaften (1) und (2) erfüllt und eine kleinere Kantengewichtssumme hat.

Beispiel



Ungerichteter Graph G

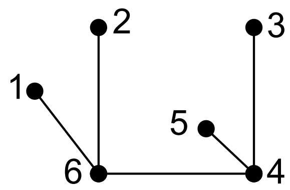


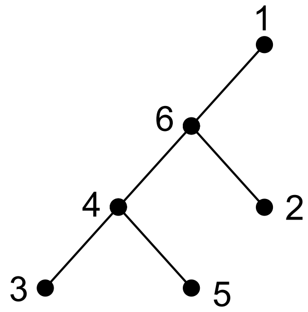
Minimal aufspannender Baum von G

Problemstellung (2)

Bemerkung

- Ein azyklischer (d.h. kreisloser) zusammenhängender Graph ist ein Baum:
  man nehme einfach einen beliebigen Knoten als Wurzel und „verschiebe die anderen Knoten nach unten“.





- Ein Baum garantiert, dass es zwischen je zwei Knoten immer genau einen Weg gibt. Er spannt damit die Knotenmenge auf.
- Der minimal aufspannende Baum muss nicht eindeutig sein.

Typische Anwendung

- Zwischen n Orten soll ein Versorgungsnetz (Kommunikation, Strom, Wasser, etc.) aufgebaut werden, so dass je 2 Orte direkt oder indirekt miteinander verbunden sind. Die Verbindungskosten zwischen 2 Orte seien bekannt.
- Gesucht ist ein Versorgungsnetz mit den geringsten Kosten.

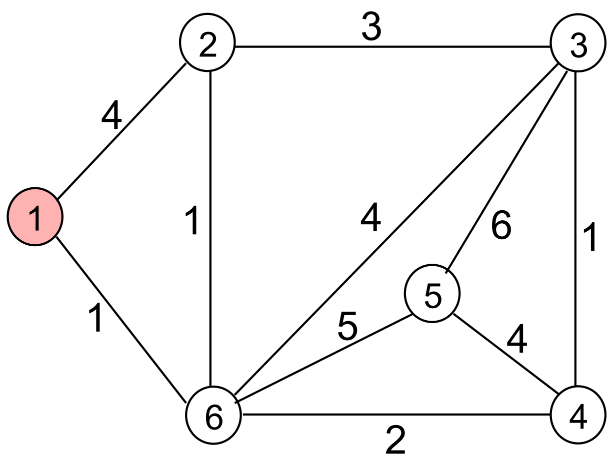
10. Minimal aufspannende Bäume

- Problemstellung
- Algorithmus von Prim
- Algorithmus von Kruskal

Idee des Algorithmus von Prim

- Der Algorithmus von Prim ist eine leichte Modifikation des Algorithmus von Dijkstra.
- Zu jedem Zeitpunkt bilden die bereits besuchten Knoten einen minimal aufspannenden Baum.
  Der minimal aufspannende Baum wird wie bei den Algorithmen für kürzeste Wege in einem Vorgängerfeld p gehalten.
- Alle Knoten, die als nächstes besucht werden können, werden in einer Kandidatenliste gehalten. Es wird mit irgendeinem Knoten begonnen.
- Von den Kandidaten wird derjenige Knoten als nächster besucht, der über die billigste Kante zu erreichen ist. Daher werden die Kosten c für jeden Knoten verwaltet.
- Beachte den Unterschied zum Dijkstra-Algorithmus:
  Beim Algorithmus von Djikstra wird immer der Knoten mit der kürzesten Distanz zum Startknoten s besucht.

Beispiel zu Algorithmus von Prim (1)



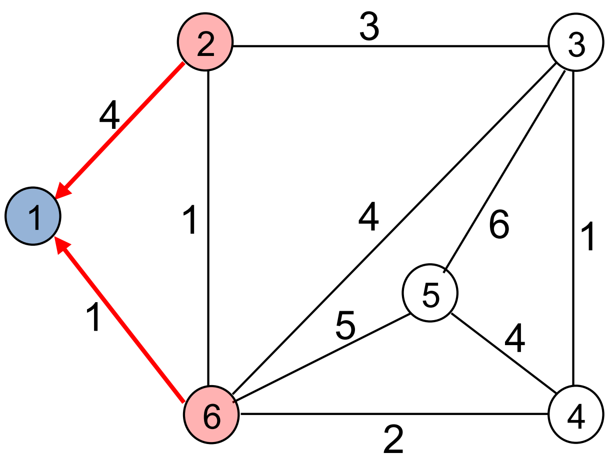
Startpunkt ist Knoten 1, der damit einziger Kandidat ist.

Knoten v	1	2	3	4	5	6
Vorgängerfeld p[v]	-	-	-	-	-	-
Kosten c[v]	0	∞	∞	∞	∞	∞

Kandidaten (rot).

Bereits besuchte Knoten bilden den minimal aufspannenden Baum (blau)

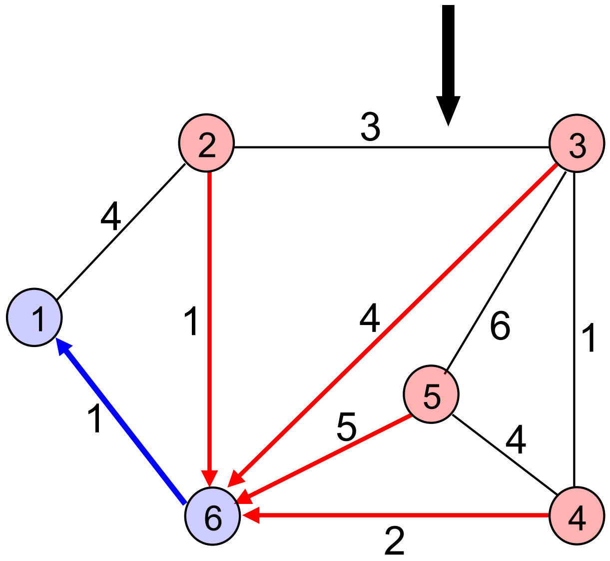
Knoten 1 wird besucht



Alle noch nicht besuchten Nachbarn von Knoten 1 werden Kandidaten.

Knoten v	1	2	3	4	5	6
Vorgängerfeld p[v]	-	1	-	-	-	1
Kosten c[v]	0	4	∞	∞	∞	1

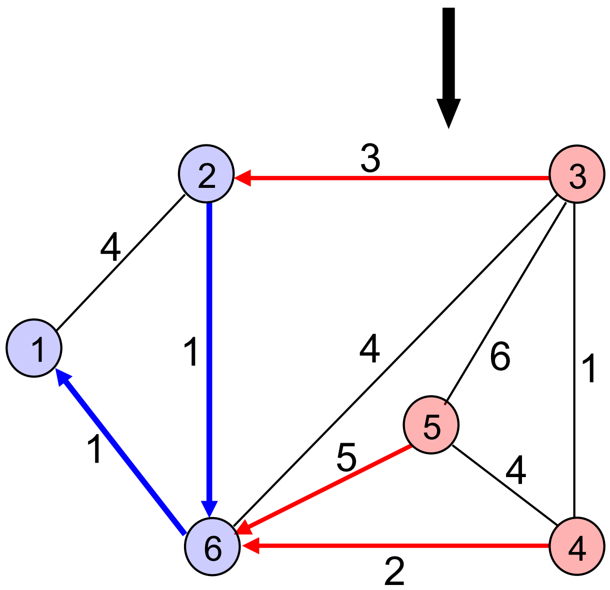
Beispiel zu Algorithmus von Prim (2)



Knoten 6 wird besucht.

Knoten v	1	2	3	4	5	6
Vorgängerfeld p[v]	-	6	6	6	6	1
Kosten c[v]	0	1	4	2	5	1

Beachte: Kosten für Knoten 2 verbessert sich.

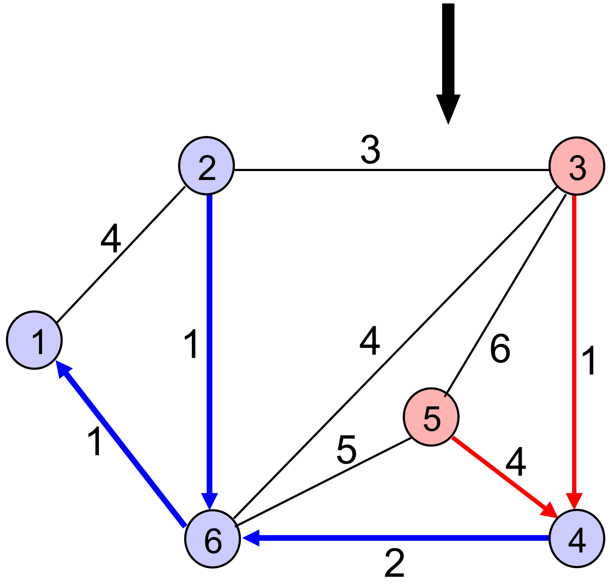


Knoten 2 wird besucht.

Knoten v	1	2	3	4	5	6
Vorgängerfeld p[v]	-	6	2	6	6	1
Kosten c[v]	0	1	3	2	5	1

Beachte: Kosten für Knoten 3 verbessert sich.

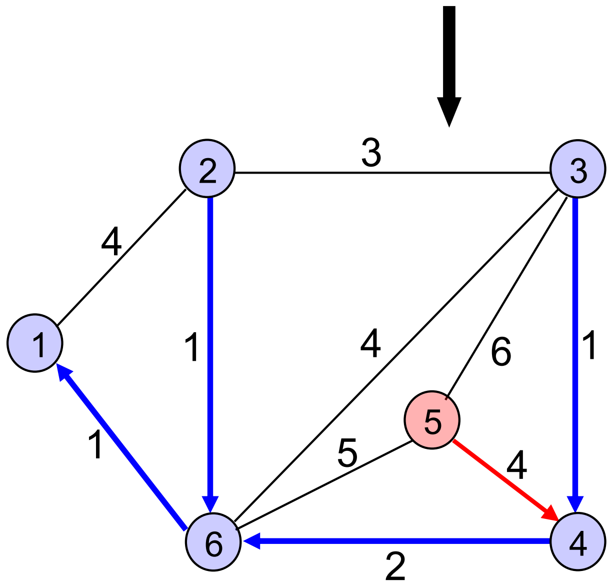
Beispiel zu Algorithmus von Prim (3)



Knoten 4 wird besucht.

Knoten v	1	2	3	4	5	6
Vorgängerfeld p[v]	-	6	4	6	4	1
Kosten c[v]	0	1	1	2	4	1

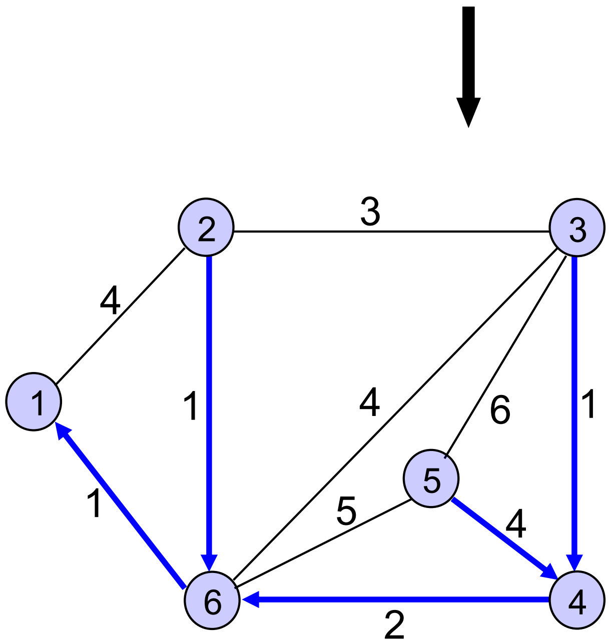
Beachte: Kosten für Knoten 3 und 5 verbessern sich.



Knoten 3 wird besucht.

Knoten v	1	2	3	4	5	6
Vorgängerfeld p[v]	-	6	4	6	4	1
Kosten c[v]	0	1	1	2	4	1

Beispiel zu Algorithmus von Prim (4)



Knoten 5 wird besucht.

Knoten v	1	2	3	4	5	6
Vorgängerfeld p[v]	-	6	4	6	4	1
Kosten c[v]	0	1	1	2	4	1

Fertig, da alle Knoten besucht!

Algorithmus von Prim

```
void minimumSpanningTree(Graph G, Vertex[ ] p ) {
        kl = ∅;    // leere Kandidatenliste
        Set<Vertex> baumKnoten = ∅;   // Knoten im aufspannenden Baum
        for (jeden Knoten v) {
                c[v] = ∞; p[v] = undef;
        }
        s = 1; c[s] = 0;    // irgendeinen Startknoten wählen
        kl.insert(s, 0);

        while (! kl.empty() ) {
                v = kl.delMin();  // lösche Knoten v aus kl mit c[v] minimal;
                baumKnoten.add(v);
                for (jeden Nachbarknoten w von v)
                        if ( ! baumKnoten.contains(w))  {
                                if (c[w] == ∞)    { // w noch nicht in Kandidatenliste
                                        p[w] = v;
                                        c[w] = c(v,w);
                                        kl.insert(w , c[w]);
                                } else if ( c(v,w) < c[w] ) { // c[w] verbessert sich
                                        p[w] = v;
                                        c[w] = c(v,w);
                                        kl.change(w , c[w]);
                                }
                        }
                }
        }
        if (baumKnoten.size() < Anzahl Knoten im Graph g)
                println("es existiert kein aufspannder Baum);
}
```

Eingabe: Gewichteter Graph G.

Ausgabe: Vorgängerfeld p, das den minimal aufspannenden Baum darstellt.

- Kandidatenliste kl enthält alle als nächstes besuchbaren Knoten v mit ihren Kosten c[v].
- c[v] gibt das Gewicht der billigsten Kante an, mit der v von den bereits besuchten Knoten erreicht werden kann.
- Für die noch nicht besuchten Knoten, die noch keine Kandidaten sind, ist c[w] = ∞.

c(v,w) = Gewicht der Kante (v,w).

Analyse des Algorithmus von Prim

- Wie beim Algorithmus von Dijkstra:

  (1) Kandidatenliste als einfaches (unsortiertes) Feld

  $$T = O(|V|^2)$$

  (2) Kandidatenliste als Index-Heap

  $$T = O(|E| \log|V|)$$

Fazit:

- Ist der Graph dicht besetzt, d.h. $|E| = O(|V|^2)$, dann ist die Variante (1) besser.
- Ist der Graph dünn besetzt, d.h. $|E| = O(|V|)$, dann ist die Variante (2) besser.

10. Minimal aufspannende Bäume

- Problemstellung
- Algorithmus von Prim
- Algorithmus von Kruskal

Idee des Algorithmus von Kruskal

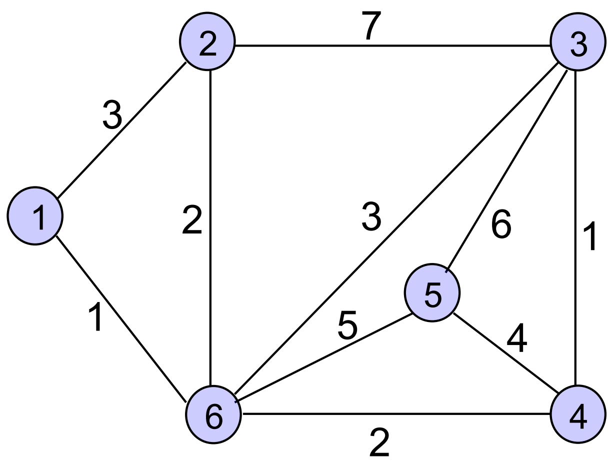
- Es wird ein Wald von minimal aufspannenden Bäumen verwaltet.
  (Wald = Menge von Bäumen)

- Start:
  Zu Anfang besteht der Wald aus allen Bäumen mit jeweils genau einem Knoten.

- Vereinigungsschritt:
  – Suche die billigste Kante k, die zwei unterschiedliche Bäume aus dem Wald verbindet.
    Vereinige die 2 Bäume mit der Kante k zu einem größeren Baum.
  – Es werden so lange Vereinigungsschritte durchgeführt, bis nur noch ein Baum übrig bleibt. Das ist dann der minimal aufspannende Baum.

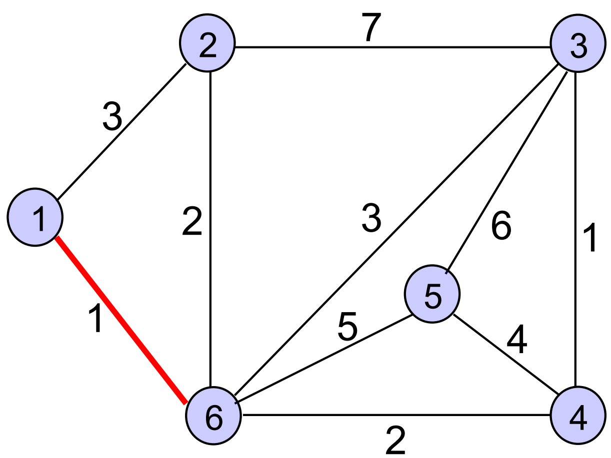
– Der Wald von Bäumen lässt sich besonders effizient mit einer so genannten Union-Find-Struktur darstellen (siehe Kapitel 14).

Beispiel zu Algorithmus von Kruskal (1)



Am Anfang besteht der
Wald aus 6 Bäumen
mit jeweils genau einem Knoten.

Kante k = (1,6) mit Gewicht 1 wird gewählt

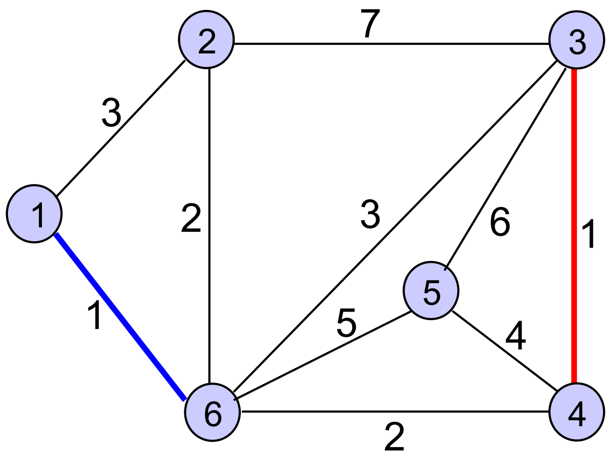


Die beiden Bäumen, die aus dem
Knoten 1 bzw. 6 bestehen, werden zu
einem Baum vereinigt.

Der Wald besteht damit nun aus
5 Bäumen.

Beispiel zu Algorithmus von Kruskal (2)

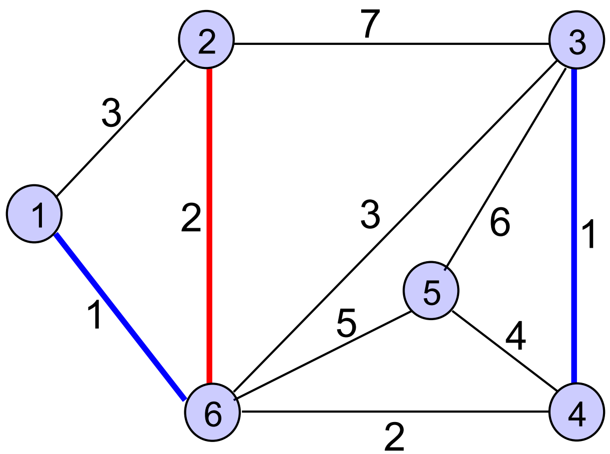
Kante k = (3,4) mit Gewicht 1 wird gewählt



Die beiden Bäumen, die aus dem Knoten 3 bzw. 4 bestehen, werden zu einem Baum vereinigt.

Der Wald besteht damit nun aus 4 Bäumen.

Kante k = (2,6) mit Gewicht 2 wird gewählt

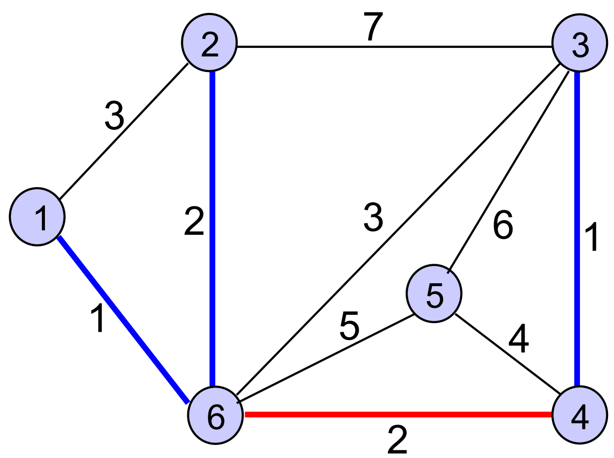


Der Baum mit den Knoten {1, 6} und der Baum mit dem Knoten 2 werden zu einem Baum vereinigt.

Der Wald besteht damit nun aus 3 Bäumen.

Beispiel zu Algorithmus von Kruskal (3)

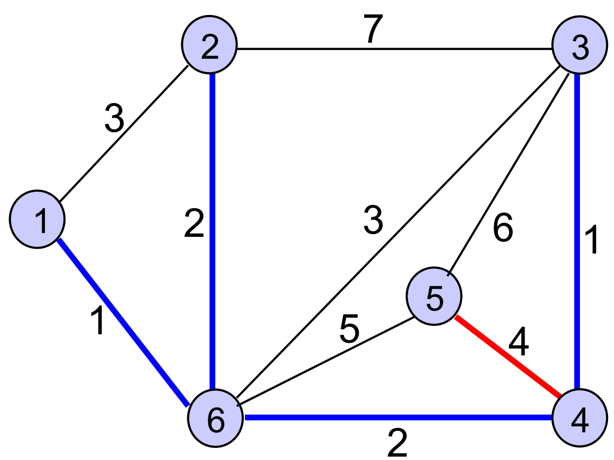
Kante k = (4,6) mit Gewicht 2 wird gewählt



Der Baum mit den Knoten {1,2,6} und der Baum mit den Knoten {3,4} werden zu einem Baum vereinigt.

Der Wald besteht damit nun aus 2 Bäumen.

Kante k = (4,5) mit Gewicht 4 wird gewählt



Der letzte Vereinigungsschritt führt zu einem Wald mit genau einem Baum.

⇒ Minimal aufspannender Baum.

Algorithmus von Kruskal

Rückgabe: Minimal aufspannender Baum

Eingabe: Graph G = (V,E).

```
List<Edge> minimumSpanningTree(Graph G) {

        UnionFind forest = { {v} / v ∊ V};
(1)     PriorityQueue<Edge> edges = E;
        List<Edge> minSpanTree;

        while ( forest.size() != 1 &&  ! edges.isEmpty() )  {
(2)             (v,w) = edges.delMin();
(3)             t1 = forest.find(v);
(4)             t2 = forest.find(w);
                if (t1 != t2) {
(5)                 forest.union(t1,t2);
(6)                 minSpanTree.add(v,w);
                }
        }

        if (forest.size() != 1)
                return “es existiert kein aufspannender Baum”;
        else
                return minSpanTree;
}
```

forest ist eine Menge von Bäumen, die anfangs aus jeweils genau einem Knoten aus V bestehen. Der Datentyp UnionFind wird in Kap. 14 erklärt.

Alle Kanten werden mit ihren Gewichten in einer Prioritätsliste gespeichert, so dass effizient auf die Kante mit dem kleinsten Gewicht zugegriffen werden kann.

Solange der Wald noch mehr als ein Baum enthält und es noch Kanten gibt.

Wähle Kante mit kleinstem Gewicht, die 2 Bäume aus dem Wald verbindet.

forest.find(v) liefert denjenigen Baum zurück, in dem v enthalten ist.

Vereinige die beiden Bäume zu einem Baum.

Analyse des Algorithmus von Kruskal

- Mit dem Datentyp UnionFind lässt sich eine Menge (hier: Wald) von disjunkten Mengen (hier Bäume) effizient verwalten mit den Operationen find und union (siehe Kap. 14).
  Damit:
  - find in (3) und (4): O(log|V|)
  - union in (5): O(log|V|)
- Eine Prioritätsliste PriorityQueue läßt sich mit einer Heap-Struktur implementieren (oder PriorityQueue aus der Java API). Damit:
  - Aufbau einer PriorityQueue in Zeile (1): O(|E|)
  - delMin in Zeile (2): O(log|E|)
- Der minimal aufpannende Baum minSpanTree ist als einfache Kanten-Liste realisiert. Damit ist (6) in O(1) durchführbar.
- Insgesamt:
  Im schlechtesten Fall wird delMin (Zeile (2)) gefolgt von Zeile (3), (4) und evtl. (5) und (6) |E|-mal durchgeführt.

  $T = O(|E|(\log|E| + \log |V|))$

  Also:

  $T = O(|E| \log|V|)$.