

# Einführung in minimale Spann­bäume und deren Berechnung

*Johannes Diemke Nils Sommer*

Universität Oldenburg, FK II - Department für Informatik  
Vortrag im Rahmen des Proseminars 2006

**Zusammenfassung:** In der folgenden Ausarbeitung behandeln wir minimale Spann­bäume und die zu ihrer Berechnung nötigen Algorithmen. Nach einer kurzen Einführung in die Thematik folgt zunächst ein Abschnitt in dem wir grundlegende Begriffe und Definitionen wiederholen. Darauf schliesst dann ein Ausblick auf unmittelbare Anwendungsgebiete des Spannbaumproblems an. Im Hauptteil wenden wir uns schliesslich den Algorithmen von Kruskal und Prim zum finden minimaler Spann­bäume zu.

## 1.1 Einleitung

Oft stellt sich die Frage wie man mit möglichst wenigen Verbindungen eine bestimmte Menge von Objekten, zum Beispiel Computer, miteinander verbinden kann. Die Lösung dieses Problems ist in der Graphentheorie ein spannender Baum. In der Praxis kann man diese Verbindungen jedoch oft bezüglich ihrer Länge, Kosten oder anderer Merkmale bewerten. Diese Bewertung bezeichnen wir im folgenden als Gewichtung. Wenn man nun die bestmögliche Verbindung aller Komponenten findet nennt man dies einen minimal spannenden Baum.

Im Folgenden werden wir minimale Spann­bäume noch formal definieren und näher auf deren praktische und theoretische Bedeutung eingehen. Im Hauptteil werden wir dann schliesslich zwei Greedy-Algorithmen zur Berechnung minimaler Spann­bäume vorstellen. Zunächst den Algorithmus von Kruskal welcher erstmals 1956 in der Zeitschrift Proceedings of the American Mathematical Society von dem US-amerikanischen Mathematiker und Statistiker Joseph Bernard Kruskal vorgestellt wurde [Kru56]. Daraufhin dann den Algorithmus von Prim welcher von Robert C. Prim erfunden wurde [Pri57]. Die Algorithmen werden jeweils zum besseren Verständnis in einem java-ähnlichem Pseudocode ausformuliert.

## 1.2 Spann­bäume

In der Graphentheorie bezeichnet ein Spannbaum  $T = (V, E')$  mit  $E' \subseteq E$  einen Teilgraphen eines ungerichteten, zusammenhängenden Graphen  $G = (V, E)$  welcher alle seine

Knoten enthält und ein Baum, also insbesondere zusammenhängend und keine Zyklen enthält, ist [HS81].

Abbildung 1.1 zeigt einen Graphen und einen seiner zugehörigen Spannbaume (Darstellung in fetten Linien und Knoten). Der vorherige Satz impliziert schon, dass ein ungerichteter, zusammenhängender Graph durchaus mehrere zugehörige Spannbaume besitzen kann.

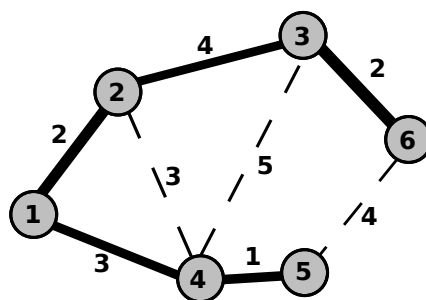
Für kantengewichtete Graphen  $G = (V, E)$  die über eine Kantengewichtungsfunktion  $w : E \rightarrow \mathbb{R}$  jeder Kante eindeutig ein Gewicht zuordnen ist das Gewicht des Graphen  $w(G)$  als die Summe aller seiner, den Kanten zugeordneten, Kantengewichte definiert:

$$w(G) = \sum_{e \in E} w(e)$$

Anschaulich gesprochen werden also die Werte an den Kanten zusammengezählt und bilden so das Gewicht des Graphen. Ein Spannbaum  $T = (V, E')$  ist nun genau dann minimal wenn kein anderer zugehöriger Spannbaum mit geringerem Gewicht existiert [HS84, Sed02]. Also wenn für alle Spannbaume  $T'$  von  $G$  gilt:

$$w(T') \geq w(T)$$

Intuitiv ist ein minimaler Spannbaum zu einem ungerichteten, zusammenhängenden, gewichteten Graphen  $G = (V, E)$  also derjenige Baum welcher alle Knoten aus  $V$  so verbindet, dass das Gewicht des Baumes minimal ist dennoch aber jeder dieser Knoten, wenn auch nur indirekt, erreichbar ist. Der minimale Spannbaum bleibt folglich zusammenhängend, reduziert jedoch sein Gewicht auf das Minimum. Hier wird deutlich welche Relevanz minimale Spannbaume in der Praxis besitzen. So tritt das Spannbaumproblem zum Beispiel in Computernetzen in Form der Frage nach der kürzeste Verdrahtung aller zu verbindenden Rechner auf [HS84]. Interpretiert man zum Beispiel die Kanten in Abbildung 1.1 als Verdrahtung und die Knoten als Computer so wird schnell klar, dass der minimale Spannbaum es immer noch erlaubt jeden Computer zu erreichen jedoch die Verdrahtungslänge drastisch sinkt und somit wahrscheinlich auch die Kosten für ein solches Netz.



**Abbildung 1.1:** Graph mit seinem zugehörigem (minimalen) Spannbaum

## 1.3 Praktische und Theoretische Bedeutung Minimaler Spannbäume

Das Minimale Spannbaum Problem ist oft repräsentativ für viele andere Probleme im Zusammenhang mit Graphen. So können die bekannten Lösungsstrategien für minimale Spannbäume oft auf diese Probleme übertragen werden um diese wiederum zu lösen [Sed02]. Bei Der Minimum-Spanning-Tree-Heuristik wird zum Beispiel zuerst ein minimaler Spannbaum erstellt um dann das Problem des Handlungsreisenden zu lösen. Auch auf Bereiche ausserhalb der Theorie kann man das Problem übertragen. So kann man sich zum Beispiel vorstellen, dass die Knoten eines Graphen elektrische Komponenten sind und die Kanten der Draht zum Verbinden. Die Gewichtung der Kanten wäre dann der Preis oder die Länge des Drahtes. Ein minimaler Spannbaum ist die optimale Verdrahtung aller Komponenten. Leicht lässt sich dies auch auf Computernetzwerke übertragen. Ein anderer praktischer Einsatz für minimale Spannbäume ist ein Scheduling Chart. Die Knoten sind hierbei die Aufträge und die gewichteten Kanten stellen die Übergänge mit dem nötigen Zeitaufwand zum Bearbeiten dar. In besseren Routern werden Spannbäume verwendet um das Routing zu verbessern und Zyklenfreiheit zu gewährleisten. Die Gewichtung der Kanten kann hierbei Kosten und Geschwindigkeit sein [Sys, Sed02].

## 1.4 Greedy Algorithmen

Die vorgestellten Algorithmen zur Konstruktion minimaler Spannbäume sind jeweils gierige Algorithmen (engl.: *greedy algorithms*) die infolge ihres Zustandes die zu einem Zeitpunkt bestmögliche Entscheidung zur Erreichung ihres Zieles auf Grund einer Bewertungsfunktion treffen. Solche Algorithmen sind im Vergleich zu backtrackenden Algorithmen vergleichsweise effizient. Gierige Algorithmen neigen ausserdem dazu schnell zu einer guten Lösung zu führen die jedoch nicht immer ein Optimum darstellt [HS81, HS84]. Eine Besonderheit sind jedoch Lösungen von Problemen welche die Struktur eines Matroiden aufweisen [Oxl92]. Für diese sind gierige Algorithmen in der Lage optimale Lösungen zu finden. Das Spannbaumproblem besitzt die Struktur eines Matroiden und somit finden die Algorithmen von Kruskal und Prim jeweils optimale Lösungen.

## 1.5 Algorithmus von Kruskal

Der Algorithmus von Kruskal konstruiert einen minimalen Spannbaum  $T = (V, E')$  aus einem ungerichteten, zusammenhängenden, gewichteten Graphen  $G = (v, E)$ . Dazu wird wie folgt beschrieben vorgegangen:

Zu Beginn enthält  $T$  genau die Knoten aus  $G$ . Seine Kantenmenge ist jedoch gleich der leeren Menge  $E' = \emptyset$ . Nun werden sukzessiv die Kanten  $e \in E$  aus dem Graphen  $G$  in die Kantenmenge  $E'$  des Baumes  $T$  eingefügt. Dabei werden alle Kanten  $e \in E$  in aufsteigender Reihenfolge ihrer Gewichte durchlaufen und in Abhängigkeit einer Auswahlbedingung entweder verworfen oder aber zu  $E'$  hinzugefügt. Die Auswahlbedingung prüft ob das Einfügen der Kante zu Zyklen führt. Führt das Einfügen der Kante  $e \in E$  in

$T$  zu Zyklen so wird die Kante  $e$  verworfen und mit der nächsten Kante fortgefahren. Ansonsten wird die Kante in die Kantenmenge  $E'$  aus  $T$  eingefügt. Sind alle Kanten von  $T$  in aufsteigender Reihenfolge ihrer Gewichte durchlaufen so terminiert dieser Algorithmus und der minimale Spannbaum  $T$  ist gefunden [GD03, HS84, Kru56].

Eine alternative Auswahlbedingung besteht darin zu prüfen ob das Einfügen einer Kante  $e \in E$  in die Kantenmenge  $E'$  des Graphen  $T$  dazu führt, dass paarweise disjunkte Zusammenhangskomponenten von  $T$  zu einer einzigen Zusammenhangskomponente „verschmelzen“ beziehungsweise ob die Kante  $e$  inzident mit zwei Knoten aus paarweise verschiedenen Zusammenhangskomponenten ist. Ist dies der Fall so wird die Kante  $e$  der Kantenmenge  $E'$  hinzugefügt, andernfalls wird sie verworfen. Es ist einleuchtend das beide Bedingungen zu einem äquivalenten Auswahlverhalten führen [GD03].

Folgender Pseudocode veranschaulicht dies:

```
while(!isConnectedGraph(T)) {
    edge = getEdgeWithMinimumCost(E);
    deleteEdgeFromGraph(E, edge);

    if(!addingEdgeToGraphCreatesCycle(T, edge)) {
        addEdgeToGraph(T, edge)
    }
}
```

**isConnectedGraph(Graph g)** prüft ob der Graph  $g$  zusammenhängend ist.

**getEdgeWithMinimumCost(EdgeSet set)** gibt die Kante mit dem kleinsten Gewicht aus der Kantenmenge  $set$  zurück.

**delteEdgeFromGraph(EdgeSet set, Edge edge)** entfernt aus der Kantenmenge  $set$  die Kante  $edge$ .

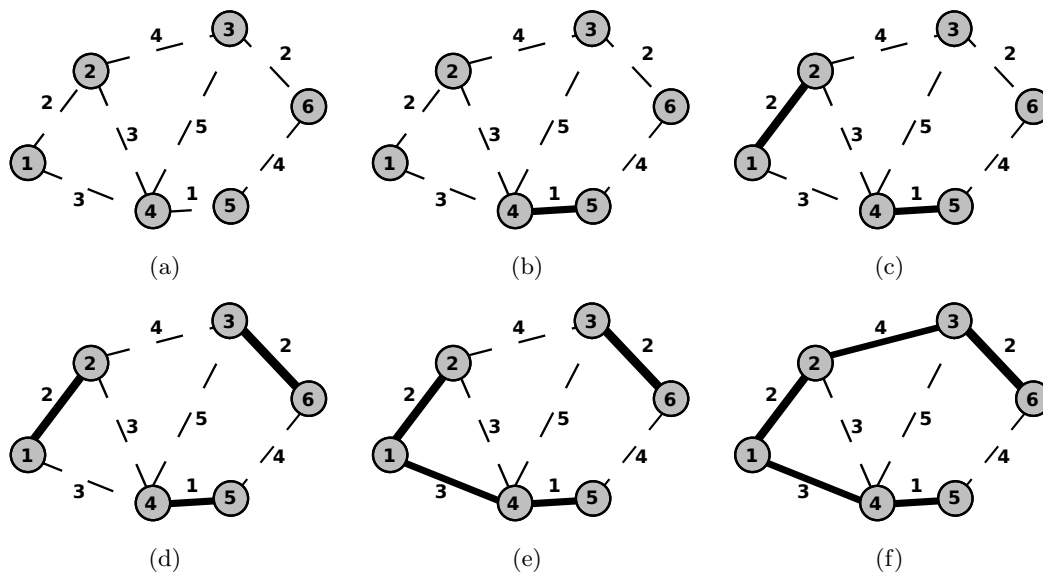
**addingEdgeToGraphCraesCycle(Graph graph, Edge edge)** gibt `true` zurück genau dann wenn das Hinzufügen der Kante  $edge$  zu dem Graphen  $g$  zu einem Zyklus führt.

**addEdgeToGraph(Graph graph, Edge edge)** fügt dem Graphen  $graph$  die Kante  $edge$  hinzu.

Der minimale Spannbaum aus Abbildung 1.1 resultiert aus der Anwendung des Algorithmus mit der Annahme, dass die Kanten in folgender Reihenfolge vorsortiert vorliegen:

$(4, 5), (1, 2), (3, 6), (1, 4), (2, 4), (2, 3), (5, 6), (4, 3)$

Abbildung 1.2 skizziert die einzelnen Schritte die der Algortihmus durchläuft während er aus dem Graphen in Abbildung 1.1 mit genannter Kantensortierung den zugehörigen minimalen Spannbaum berechnet.



**Abbildung 1.2:** Einzelne Phasen bei der Berechnung des minimalen Spannbaums (von links oben nach rechts unten)

Ein Beweis dafür, dass dieser Algorithmus auch tatsächlich für jeden ungerichteten, zusammenhängenden, gewichteten Graphen einen seiner minimalen Spannbaume liefert wird in dem Buch von Güntig und Dierker [GD03] geliefert.

In konkreten Implementierungen wird das Finden der nächste Kante mit minimalen Kosten durch eine Prioritätswarteschlange (engl.: Priority-Queue) verwirklicht, welche in aufsteigender Reihenfolge mit den Kanten des Graphen für den wir einen minimalen Spannbaum suchen initialisiert werden muss. Das Entfernen der Kante aus der Kantenmenge  $E$  wird ebenfalls durch die Priority-Queue implementiert. Eine Partition in Form einer Merge-Find-Struktur wird für das effiziente Erkennen von Zyklen genutzt. Also das Erkennen ob die Endknoten der einzufügenden Kante in der selben Zusammenhangskomponente des Graphen liegen, so dass das Einfügen der Kante zu einem Zyklus führen würde. Der Hauptaufwand des Algorithmus resultiert aus der Verwaltung der Kanten in der Prioritätswarteschlange. Insgesamt ergibt sich daraus ein Gesamtaufwand von  $O(e \log e)$  wobei  $e$  die Anzahl der Kanten in dem Graphen  $G = (V, E)$  mit  $e = |E|$  ist [Sed02, GD03].

## 1.6 Prim's Algorithmus

Dieser Algorithmus wurde schon 1957 von Robert C. Prim in Proceedings of the American Mathematical Society. 7 vorgestellt [Pri57]. Wie auch Kruskals Algorithmus geht Prim's Algorithmus nach der Greedy Methode vor. Um aus dem Graphen  $G$  einen minimalen Spannbaum  $T$  zu berechnen geht der Algorithmus wie folgt vor.

$T$  wird als ein trivialer Graph mit einem beliebigen Knoten aus  $G$  initialisiert. Im

nächsten Schritt wird dem Graphen  $T$  eine Kante aus  $G$  hinzugefügt. Diese Kante muss einen neuen Knoten aus  $G$ , der nicht schon in  $T$  enthalten ist, mit  $T$  verbinden. Ausserdem muss die Kante am niedrigsten gewichtet sein von allen Kanten, die die erste Bedingung erfüllen. Wenn diese Kante zu  $T$  hinzugefügt wurde, wird auch der Knoten, den sie verbindet, zu  $T$  hinzugefügt. Dieser Vorgang wird wiederholt bis alle Knoten aus  $G$  auch in  $T$  enthalten sind.  $T$  ist dann ein minimaler Spannbaum von  $G$ .

Im Pseudocode sieht dann ein Algorithmus, der diese Aufgabe erfüllt, wie folgt aus:

```
public static Graph primMinSpanTree(Graph input) {
    Graph T = new Graph();
    T.addNode(input.getAnyNode());
    while(!T.nodesEqual(input)) {
        Edge edges[] = getEdgesWhichAddNewNode(input, T);
        edges.sortByWeight();
        T.addEdgeWithNode(edges[0]);
    }
    return T;
}
```

**Graph** repräsentiert einen gewichteten Graphen mit Knoten und Kanten.

**Edge** hält eine gewichtete Kante in einem Graphen.

**addNode()** fügt einen Knoten zum Graphen hinzu.

**getAnyNode()** liefert einen beliebigen Knoten des Graphen zurück.

**nodesEqual(Graph)** überprüft ob die Menge der Knoten gleich derer des Arguments ist.

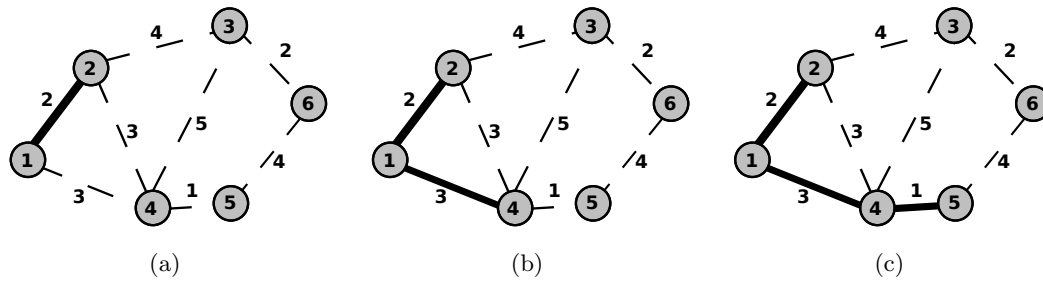
**getEdgesWhichAddNewNode(Graph1, Graph2)** gibt ein Array der Kanten zurück die einen neuen Knoten aus Graph1 zu Graph2 hinzufügen würden und somit die erste Bedingung des Prim Algorithmus erfüllen.

**sortByWeight()** sortiert ein Array von Kanten nach deren Gewichtung.

**addEdgeWithNode()** fügt dem Graphen eine neue Kante, sowie den dazugehörigen Knoten der noch nicht in dem Graphen enthalten ist, hinzu.

Dieser Algorithmus liegt in der Komplexitätsklasse  $O(n^2)$ . Dies kann man leicht anhand des Pseudocodes erkennen. Die while-Schleife benötigt  $n - 1$  Iterationen, da der entstehende Spannbaum  $n - 1$  Kanten haben wird und in jeder Iteration eine Kante hinzugefügt wird. Die Kanten mit **getEdgesWhichAddNewNode** zu ermitteln braucht im Mittel  $\frac{n}{2}$  Rechenschritte. Bei dem Sortieren der in Frage kommenden Kanten kann man auch von  $\frac{n}{2}$  Rechenschritten im Mittel ausgehen. Insgesamt ergeben sich  $(n - 1) * (\frac{n}{2} + \frac{n}{2})$  also die Klasse  $O(n^2)$  [HS84].

Abbildung 1.3 zeigt die ersten drei Schritte des Prim Algorithmus angesetzt auf den Graphen aus Kapitel 3. Als Startknoten hat der Algorithmus in diesem Fall den ersten



**Abbildung 1.3:** Drei Iterationen des Prim Algorithmus

Knoten. In dem zweiten Schritt hat der Algorithmus zwei Kanten mit gleicher Gewichtung zur Auswahl. Welche hier bevorzugt wird kann nicht deterministisch entschieden werden.

## 1.7 Resümee

Wir haben nun in die Thematik der minimalen Spannbäume eingeführt und zwei bekannte Algorithmen zur effizienten Lösung vorgestellt. Wenn man gute Datenstrukturen und Techniken in diesen Algorithmen verwendet sind sie auch von der Effizienz optimal.

Da schon in den fünfziger Jahren viel Forschung in dem Bereich betrieben wurde ist dieses Gebiet weitestgehend abgeschlossen und in der Theorie nur noch zur Lehre oder zur Lösung komplexerer Probleme relevant. In der Praxis hingegen finden die Algorithmen noch nach wie vor weiten Einsatz.

# Literaturverzeichnis

- [GD03] Ralf Hartmut Gültling und Stefan Dieker. *Datenstrukturen und Algorithmen*. Teubner, 2003.
- [HS81] Ellis Horowitz und Sartaj Sahni. *Algorithmen, Entwurf und Analyse*. Springer Verlag, 1981.
- [HS84] Ellis Horowitz und Sartaj Sahni. *Fundamentals of Computer Algorithms*. Computer Sci. P, 1984.
- [Kru56] Joseph Bernard Kruskal. On the shortest spanning subtree and the traveling salesman problem. In: *Proceedings of the American Mathematical Society*. 7, 1956.
- [Oxl92] James G. Oxley. *Matroid Theory*. Oxford Science Publications, 1992.
- [Pri57] Robert C. Prim. Shortest connection networks and some generalisations. In: *Bell System Technical Journal*, 36, 1957.
- [Sed02] Robert Sedgewick. *Algorithmen*. 2. Ausgabe, 2002.
- [Sys] Cisco System. *Understanding Spanning-Tree Protocol* - [http://www.cisco.com/univercd/cc/td/doc/product/rtrmgmt/sw\\_ntman/cwsi2/cwsiug2/vlan2/stpapp.htm](http://www.cisco.com/univercd/cc/td/doc/product/rtrmgmt/sw_ntman/cwsi2/cwsiug2/vlan2/stpapp.htm).