1 Minimale Spannbäume

Es sei ein ungerichteter, zusammenhängender Graph $G = (V, E)$ mit $|V| = n$ Knoten, $|E| = m$ Kanten und Kantengewichten $c : E \to \mathbb{R}^+$ gegeben. Ein Spannbaum von G ist ein Teilgraph $T = (V, E')$ von G mit $E' \subseteq E$, der zusammenhängend ist und keine Kreise enthält. Das Gewicht eines Spannbaums ist gleich der Summe der Gewichte aller Kanten in dem Spannbaum, also $\sum_{e \in E'} c(e)$. Ein Spannbaum T ist ein *minimaler Spannbaum*, falls kein anderer Spannbaum von G ein kleineres Gewicht hat als T. Ein Graph kann mehrere unterschiedliche minimale Spannbäume haben.

Ein Problem, das durch die Berechnung eines minimalen Spannbaums gelöst werden kann, ist das folgende: Es ist ein Kommunikationsnetzwerk aufzubauen, das eine gegebene Menge von Knotenpunkten zusammenhängend verbindet. Dabei kann zwischen bestimmten Paaren von Knotenpunkten eine Leitung eingerichtet werden, deren Kosten von den Endpunkten der Leitung abhängen. Das Problem, eine möglichst billige Menge von einzurichtenden Leitungen auszuwählen, die aber ausreicht, das Netzwerk zusammenhängend zu machen, entspricht genau dem Problem, einen minimalen Spannbaum zu berechnen. Dabei repräsentiert V die Menge der Knotenpunkte, E die Menge der möglichen Verbindungsleitungen und $c(\{v, w\})$ die Kosten für das Einrichten einer Verbindung von v nach w.

Die Algorithmen zur Berechnung eines minimalen Spannbaums, die wir betrachten, beginnen mit einer leeren Kantenmenge $E' = \emptyset$ und fügen nacheinander Kanten in E' ein, bis schließlich E' genau die Kantenmenge eines minimalen Spannbaums ist. Dabei werden zu jedem Zeitpunkt die Knoten des Graphen G durch die Kanten, die schon in E' eingefügt wurden, in eine Menge von Teilbäumen partitioniert. Anfangs, wenn $E' = \emptyset$ gilt, besteht jeder dieser Teilbäume aus einem einzigen Knoten. Durch jedes Einfügen einer Kante e in E' werden die zwei Teilbäume, die die Endknoten von e enthalten, zu einem einzigen Teilbaum vereinigt. Damit keine Kreise entstehen, dürfen nur Kanten in E' eingefügt werden, die Knoten aus verschiedenen Teilbäumen verbinden. Am Ende, d.h. nach $n - 1$ Einfügungen, bleibt genau ein Baum übrig. Die Regel, nach der entschieden wird, welche Kanten jeweils in E' eingefügt werden sollen, muss so gewählt sein, dass der resultierende Spannbaum tatsächlich minimales Gewicht hat.

Satz 1 *Sei $E' \subset E$ so, dass es einen minimalen Spannbaum T von G gibt, der alle Kanten in E' enthält. Sei T' einer der Teilbäume, die durch E' gegeben sind. Sei e eine Kante mit minimalem Gewicht unter allen Kanten, die einen Knoten aus T' mit einem Knoten aus $G \setminus T'$ verbinden. Dann gibt es auch einen minimalen Spannbaum, der alle Kanten aus $E' \cup \{e\}$ enthält.*

Beweis: Wie im Satz bezeichne e die aktuell eingefügte Kante, E' die aktuelle Kantenmenge vor Einfügen von e, T' den aktuellen Teilbaum, aus dem die Kante e herausführt, und T einen minimalen Spannbaum von G, der E' enthält. Es ist zu zeigen, dass es auch einen minimalen Spannbaum von G gibt, der $E' \cup \{e\}$ enthält.

Falls e in T enthalten ist, so ist T der gesuchte minimale Spannbaum und wir sind fertig. Nehmen wir also an, dass e nicht in T enthalten ist. Dann entsteht durch Einfügen von e in T ein Kreis k, der sowohl Knoten in T' als auch Knoten in $G \setminus T'$ enthält. Folglich muss k außer e noch mindestens eine weitere Kante e' enthalten, die einen Knoten in T' mit einem Knoten

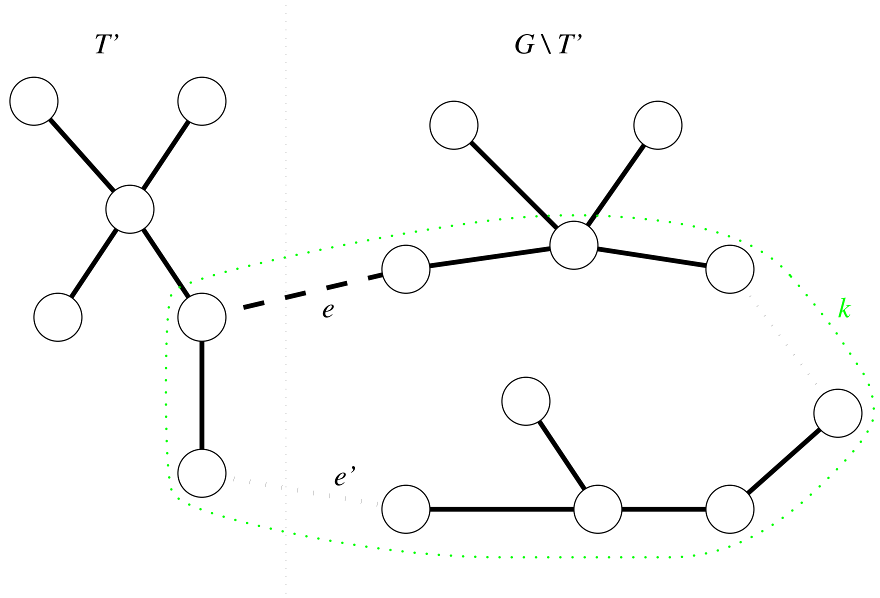

Abbildung 1: Skizze zum Beweis von Satz 1

in $G \setminus T'$ verbindet, also aus T' herausführt, siehe Abbildung 1. Da e minimales Gewicht unter allen solchen Kanten hatte, muss $c(e') \geq c(e)$ gelten. Wenn wir nun aus T die Kante e' entfernen und die Kante e einfügen, so erhalten wir einen Spannbaum T'', dessen Gewicht nicht größer als das von T ist. Da T ein minimaler Spannbaum ist, ist das Gewicht von T'' gleich dem Gewicht von T. Da T'' außerdem $E' \cup \{e\}$ enthält, ist T'' der gesuchte minimale Spannbaum. □

Dieser Satz sagt uns, dass wir einen minimalen Spannbaum berechnen können, indem wir mit einer leeren Kantenmenge E' beginnen (diese ist auf jeden Fall Teilmenge der Kantenmenge eines minimalen Spannbaums) und in jedem Schritt eine Kante e auswählen und in E' aufnehmen, die unter allen Kanten, die aus einem der aktuellen Teilbäume herausführen, minimales Gewicht hat.

1.1 Algorithmus von Kruskal

Der Algorithmus von Kruskal betrachtet die Kanten des Graphen in der Reihenfolge aufsteigenden Gewichts (bei Kanten mit gleichem Gewicht ist die Reihenfolge egal). Verbindet die aktuell betrachtete Kante e zwei Knoten, die schon im selben Teilbaum sind, so wird die Kante verworfen und nicht in E' eingefügt. Verbindet e dagegen zwei Knoten aus verschiedenen Teilbäumen, so wird e in E' eingefügt und damit zwei Teilbäume zu einem verbunden.

Da der Algorithmus nur solche Kanten e in E' einfügt, die unter allen Kanten, die noch für einen Spannbaum in Frage kommen, minimales Gewicht haben, wird insbesondere die Bedingung von Satz 1 erfüllt und der Algorithmus liefert einen minimalen Spannbaum.

Für eine effiziente Implementierung des Algorithmus von Kruskal gibt es zwei zeitkritische Stellen: erstens ist zu überlegen, auf welche Weise die Kanten in Reihenfolge aufsteigender Gewichte betrachtet werden können; zweitens ist nicht klar, wie für die jeweils aktuelle Kante e entschieden werden soll, ob die beiden Endpunkte in verschiedenen Teilbäumen liegen oder nicht.

Zur Lösung des ersten Problems bieten sich zwei Methoden an. Erstens kann man die Kanten des Graphen einfach am Anfang mit Zeitaufwand $O(m \log m) = O(m \log n)$ nach ihrem Gewicht sortieren. Zweitens kann eine Priority-Queue (Prioritätswarteschlange) verwendet werden. Eine Priority-Queue ist eine Datenstruktur Q, die zumindest die folgenden Operationen effizient realisiert:

- INSERT(e,p): füge Element e mit Priorität p in Q ein
- DECREASEPRIORITY(e,p): erniedrige die Priorität des bereits in Q befindlichen Elements e auf p
- DELETEMIN(): liefere das Element aus Q mit niedrigster Priorität zurück und lösche es aus Q

Werden Priority-Queues mit Fibonacci-Heaps implementiert, so ist der Zeitaufwand für INSERT und DECREASEPRIORITY (amortisiert) $O(1)$ und für DELETEMIN $O(\log n)$, wenn Q n Elemente enthält. In LEDA stehen solche Priority-Queues als `p_queue` oder als `node_pq` (speziell für Knoten eines Graphen) zur Verfügung.

Zu Anfang des Algorithmus von Kruskal können alle Kanten mit ihrem Gewicht als Priorität in Zeit $O(m)$ in die Priority-Queue eingefügt werden. Um dann jeweils die nächste aktuelle Kante zu bestimmen, wird die DELETEMIN-Operation der Priority-Queue benutzt. Jedes DELETEMIN benötigt $O(\log m) = O(\log n)$ Zeit. Wenn der minimale Spannbaum nach der Bearbeitung von ℓ Kanten fertig berechnet ist, ist der Zeitaufwand für die Priority-Queue-Operationen also $O(m + \ell \log n)$. Im Worst-Case ist das ebenfalls $O(m \log n)$, nämlich wenn die Kante mit dem größten Gewicht im Spannbaum ist und daher alle Kanten bearbeitet werden müssen. Die Implementierung mit Priority-Queue ist jedoch vorteilhaft, wenn weniger Kanten bearbeitet werden müssen.

Das zweite Problem war die Entscheidung, ob die zwei Endknoten einer Kante im selben oder in verschiedenen Teilbäumen liegen. Dies ist ein so genanntes Union/Find-Problem: man will eine Partition einer Grundmenge in disjunkte Teilmengen dynamisch verwalten, wobei nur die Operationen FIND ("zu welcher Teilmenge gehört ein bestimmtes Element?") und UNION ("vereinige zwei Teilmengen") ausgeführt werden sollen. Dazu existieren sehr effiziente Verfahren, die für eine Grundmenge von n Elementen $k \geq n$ FIND- und UNION-Operationen in Zeit $O(k\alpha(k, n))$ ausführen, wobei α die inverse Ackermann-Funktion ist und extrem langsam wächst. Bei unserer Anwendung müssen wir $n - 1$ UNION-Operationen und höchstens $2m$ FIND-Operationen ausführen. Der Gesamtaufwand dafür ist also $O(m\alpha(m, n))$. In LEDA stehen Union/Find-Strukturen als `partition` bzw. `node_partition` (speziell für Partitionen der Knotenmenge eines Graphen) zur Verfügung.

Da $\alpha(m, n)$ viel langsamer wächst als $\log m$ bzw. $\log n$, ist der Gesamtaufwand für den Algorithmus von Kruskal mit den beschriebenen Implementierungen $O(m \log n)$.

1.2 Algorithmus von Prim

Der Algorithmus von Prim wählt einen beliebigen Knoten des Graphen G als Startknoten. Dann wird immer derjenige Teilbaum betrachtet, der den Startknoten enthält. Unter allen Kanten, die aus diesem Teilbaum herausführen, wird eine mit minimalem Gewicht ausgewählt und in E' eingefügt. Dadurch wird der Teilbaum, der den Startknoten enthält, um eine Kante und einen Knoten größer. Zu jedem Zeitpunkt gibt es nur einen Teilbaum, der mehr als einen Knoten enthält. Dieser eine Teilbaum wächst in $n - 1$ Schritten zu einem minimalen Spannbaum heran.

Offensichtlich erfüllt diese Auswahlregel zum Einfügen von Kanten in E' die Bedingung von Satz 1. Folglich liefert der Algorithmus von Prim einen minimalen Spannbaum.

Bei der Implementierung des Algorithmus von Prim ist zu überlegen, wie in jedem Schritt die in E' einzufügende Kante effizient berechnet werden kann. Diese Kante muss minimales Gewicht haben unter allen Kanten, die einen Knoten aus dem aktuellen Teilbaum T mit einem Knoten aus $G \setminus T$ verbinden. Auch hier erweist sich eine Priority-Queue als Datenstruktur sehr nützlich: wir speichern in der Priority-Queue Q alle Knoten aus $G \setminus T$, die über eine Kante aus T heraus erreichbar sind. Sei v ein solcher Knoten aus $G \setminus T$ und sei e_v eine Kante mit minimalem Gewicht unter allen Kanten, die einen Knoten aus T mit v verbinden. Dann soll die Priorität von v in Q gleich $c(e_v)$ sein. Zusätzlich merken wir uns bei Knoten v, dass e_v seine billigste Kante zum Spannbaum ist. Am Anfang initialisieren wir Q, indem wir alle Nachbarknoten des Startknotens in Q einfügen und ihnen als Priorität das Gewicht der betreffenden zum Startknoten inzidenten Kante geben. Außerdem merken wir uns bei jedem solchen Knoten eben diese Kante als vorläufig günstigste Kante, um den Knoten vom Startknoten aus zu erreichen.

Wenn nun die nächste Kante zum Einfügen in E' ausgewählt werden soll, führen wir einfach die DELETEMIN-Operation der Priority-Queue aus; dadurch bekommen wir den Knoten v aus $G \setminus T$, der von T aus über eine Kante mit minimalem Gewicht, nämlich die Kante e_v, erreichbar ist. Wir fügen e_v in E' ein und müssen zusätzlich einige Daten aktualisieren; da v nämlich jetzt in T ist, kann es Nachbarn w von v geben, die noch nicht in T enthalten sind und die entweder vorher überhaupt nicht von T aus über eine Kante erreichbar waren oder nur über eine Kante mit höherem Gewicht als $c(\{v, w\})$. In ersterem Fall fügen wir w mit Priorität $c(\{v, w\})$ in Q ein; in letzterem Fall erniedrigen wir die Priorität von w auf $c(\{v, w\})$ mittels einer DECREASEPRIORITY-Operation. In beiden Fällen merken wir uns bei Knoten w, dass w nun von T aus am günstigsten über die Kante $e_w = \{v, w\}$ erreicht werden kann.

Der Algorithmus von Prim berechnet den minimalen Spannbaum in $n - 1$ Schritten, wobei in jedem Schritt durch eine DELETEMIN-Operation ein neuer Spannbaumknoten v gewählt wird und dann alle Nachbarn von v ggf. durch eine INSERT- oder DECREASEPRIORITY-Operation in die Queue eingefügt oder bzgl. ihrer Priorität aktualisiert werden müssen. Ohne Berücksichtigung der Operationen auf der Priority-Queue ergibt sich damit ein Zeitaufwand von $O(n + m) = O(m)$. Da die INSERT- und DECREASEPRIORITY-Operationen (amortisiert) Zeitbedarf $O(1)$ und die DELETEMIN-Operation Zeitbedarf $O(\log n)$ haben und $n - 1$ INSERT-Operationen, $n - 1$ DELETEMIN-Operationen und $O(m)$ DECREASEPRIORITY-Operationen ausgeführt werden, ergibt sich insgesamt eine Laufzeit von $O(m + n \log n)$.