

Teil 5: Algorithmen auf Graphen

Minimale Spannbäume

Definition (Spannbaum, minimaler Spannbaum)

$G = (V, E, c)$ sei ein ungerichteter, gewichteter, zusammenhängender Graph mit **Kostenfunktion** $c: E \rightarrow \mathbb{R}_{\geq 0}$.

Ein **Spannbaum** T von G ist ein Baum $T = (V, E')$ mit $E' \subseteq E$, d.h. T ist ein azyklischer, zusammenhängender Teilgraph von G , der alle Knoten enthält.

Das **Gewicht** oder die **Kosten** $c(T)$ eines Spannbaumes $T = (V, E')$ sind die Summe seiner Kantengewichte:

$$c(T) := \sum_{e \in E'} c(e).$$

Ein **minimaler Spannbaum** ist ein Spannbaum T mit minimalem Gewicht, d.h.

$$c(T) \leq c(T') \text{ für alle Spann bäume } T' \text{ von } G.$$

Minimale Spannbäume

Mittels der Tiefen- und Breitensuche haben wir bereits Spannbäume konstruiert. Allerdings haben wir dabei das Gewicht des Baumes vollkommen ignoriert.

Im Folgenden wollen wir einige Verfahren vorstellen, mit denen man minimale Spannbäume konstruieren kann.

Teil 5: Algorithmen auf Graphen
Minimale Spannbäume

Der Algorithmus von Kruskal

Der Algorithmus von Kruskal

Grundprinzip des Algorithmus von Kruskal

Füge nach und nach die günstigste Kante zu dem bislang konstruierten Teilgraphen hinzu, die keine Kreise erzeugt.

Dabei handelt es sich um einen so genannten **Greedy-Algorithmus** (greedy = gierig).

Um genau zu sein, wird anfänglich jeder Knoten als ein einzelner Teilbaum von G betrachtet.

Nach und nach werden nun die Kanten, sortiert nach aufsteigendem Gewicht, darauf überprüft, ob sie zwischen zwei Bäumen liegen oder zwei Knoten eines bereits vorhandenen Baumes verbinden.

Im zweiten Fall wird die Kante einfach ignoriert.

Im ersten Fall wird die Kante hinzugefügt und die beiden Bäume zusammengelegt. Dadurch sinkt die Anzahl der Bäume um eins.

Verwaltung der Bäume und der Zugriff auf die Kanten

Die Bäume werden einfach als Knotenmengen beschrieben (die Kanten werden unabhängig davon gesammelt).

Dafür verwenden wir eine Union-Find-Datenstruktur

Eine Möglichkeit den Algorithmus von Kruskal einfach realisieren zu können, besteht darin die Kanten vor dem eigentlichen Algorithmus nach dem Gewicht zu sortieren.

Eine andere Möglichkeit besteht in der Verwendung einer Priority Queue. Dabei werden zu Beginn alle Kanten in die Schlange eingefügt und anschließend Schritt für Schritt das Minimum extrahiert.

Da die Verwendung einer Sortierten Liste äquivalent zur Verwendung einer Prioritätswarteschlange ist (auch was die asymptotischen Laufzeiten angeht), werden wir die zweite Variante beschreiben.

Algorithmus (Kruskal)

Eingabe: Ein ungerichteter Graph $G = (V, E)$ mit nicht-negativen Kantengewichten $c: E \rightarrow \mathbb{R}_{\geq 0}$.

Ausgabe: Ein minimaler Spannbaum T von G , repräsentiert als Kantenliste.

Daten: Eine Union-Find-Datenstruktur U , eine Priority Queue Q .

$T := \emptyset$

für alle $e \in E$ **tue** $Q.insert(e)$

für alle $v \in V$ **tue** $U.make_set(v)$

solange $Q \neq \emptyset$ **tue**

$e = (u, v) := Q.delete_min();$ // Hole nächste Kante

wenn $U.find(u) \neq U.find(v)$ **dann** // Bäume verschieden?

$U.union(u, v)$ // Vereinige Bäume

$T := (T, e)$ // Füge Kante zu Baum hinzu

Ende

Ende

Die Laufzeit

Die Initialisierung benötigt, je nach verwendeter Priority Queue $O(|E|t_{\text{insert}})$ für das Einfügen der Kanten, und $O(|V|)$ für das Anlegen der Knotenmengen.

Anschließend wird jede Kante genau einmal entnommen, was zu $O(|E|t_{\text{delete_min}})$ führt.

Für jede Kante werden zwei find-Operationen durchgeführt und höchstens $(n - 1)$ -mal union, was zu einer Laufzeit von $O(2|E|t_{\text{find}} + |V|t_{\text{union}})$ für die Schleife führt.

Insgesamt somit

$$O(|E|(t_{\text{insert}} + t_{\text{delete_min}} + 2t_{\text{find}}) + |V|(t_{\text{union}} + 1))$$

Die Laufzeit

$$O(|E|(t_{\text{insert}} + t_{\text{delete_min}} + 2t_{\text{find}}) + |V|(t_{\text{union}} + 1))$$

Die Laufzeiten t_{insert} und $t_{\text{delete_min}}$ hängen dabei von der verwendeten Prioritätswarteschlange, und die Laufzeiten $t_{\text{make_set}}$ und t_{union} von der verwendeten Union-find-Datenstruktur ab.

Da $t_{\text{delete_min}}$ in allen vorgestellten Varianten $O(\log |E|)$ ist, können wir auch $t_{\text{insert}} = O(\log |E|)$ annehmen.

Verwenden wir die vorgestellte Union-Find-Datenstruktur, ergeben sich die amortisierten Kosten von `union` und `find` als $O(\alpha(|V|)) = O(\log |V|)$ und von `make_set` als $O(1)$.

$$\begin{aligned} O(|E| (t_{\text{insert}} + t_{\text{delete_min}} + 2t_{\text{find}}) + |V|(t_{\text{union}} + 1)) \\ &= (|E|(\log |E| + \alpha(|V|)) + |V|O(\alpha(|V|))) \\ &= (|E|(\log |E| + \log |V|) + |V|O(\log |V|)) \end{aligned}$$

Die Laufzeit

Da G zusammenhängend ist, gilt $|V| - 1 \leq |E| \leq \frac{|V|^2 - |V|}{2}$ (Warum? Übungsaufgabe).

Damit gilt auch $\log |E| = O(\log |V|)$ und $|V| = O(|E|)$ und somit ergibt sich die Gesamtlaufzeit als

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

Es bleibt die Korrektheit zu zeigen.

Satz

Der Algorithmus von Kruskal berechnet in $O(|E| \log |V|)$ einen minimalen Spannbaum eines zusammenhängenden, ungerichteten Graphen $G = (V, E)$ mit nicht-negativen Kantengewichten.

Beweis (Algorithmus von Kruskal)

Sei E_i die Menge der nach dem i -ten Schleifendurchlauf bereits für den Spannbaum ausgewählte Menge von Kanten.

Wir zeigen per Induktion über i , dass E_i stets zu einem minimalen Spannbaum erweitert werden kann, d.h. dass ein Spannbaum $T = (V, E')$ existiert mit $E_i \subseteq E'$.

Für $i = 0$ ist $E_0 = \emptyset$ und die Behauptung trivialerweise erfüllt.

Nehmen wir an die Behauptung gilt für $0 \leq j \leq i - 1$ und $i \geq 1$.

Falls die i -te Kante nicht zu E_{i-1} hinzugefügt wird, gilt $E_i = E_{i-1}$ und somit ist die Behauptung für E_i immer noch erfüllt.

Nehmen wir nun an, dass die i -te Kante $e_i = (u_i, v_i)$ hinzugefügt wird, d.h. es gilt $\text{find}(u_i) \neq \text{find}(v_i)$.

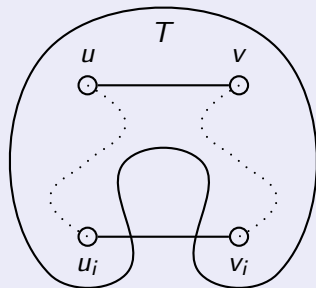
Beweis (Algorithmus von Kruskal - Fortsetzung)

Sei nun $T = (V, E')$ eine Erweiterung von E_{i-1} . Falls $e_i \in E'$ ist T auch eine Erweiterung von E_i .

Also sei $e_i \notin E'$. Da T bereits einen Weg von u_i nach v_i enthält, gibt es in dem Graphen $G' = (V, E' \cup \{e_i\})$ einen Kreis, auf dem e_i liegt.

Nehmen wir nun an, dass jede der Kanten auf diesem Kreis in E_i liegt. Dann muss der Weg von u_i nach v_i in T bereits in E_{i-1} liegen. Damit würde aber $\text{find}(u_i) = \text{find}(v_i)$ gelten.

Damit existiert auf dem Kreis eine Kante $e = (u, v)$, die nicht in E_i liegt.



Beweis (Algorithmus von Kruskal - Fortsetzung)

Sei nun $c(e) < c(e_i)$, d.h. e wurde bereits bearbeitet. Da $e \notin E_{i-1}$ gilt, muss vor dem i -ten Durchlauf, nämlich bei der Bearbeitung von e , bereits $\text{find}(u) = \text{find}(v)$ gegolten haben. Damit sind u und v bereits in E_{i-1} verbunden.

Damit erzeugt e zusammen mit E_{i-1} einen Kreis und somit kann, wegen $E_{i-1} \subseteq E'$, T kein Spannbaum sein.

Dies steht im direkten Widerspruch zur Wahl von T . Damit muss $c(e) \geq c(e_i)$ gelten.

Nun ist aber $T' := (V, E' \setminus \{e\} \cup \{e_i\})$ wieder ein Spannbaum, denn er enthält keine Kreise und ist zusammenhängend, und es gilt

$$c(T') = c(T) + c(e_i) - c(e) \leq c(T) \leq C(T').$$

Damit ist T' ein minimaler Spannbaum der E_i erweitert.

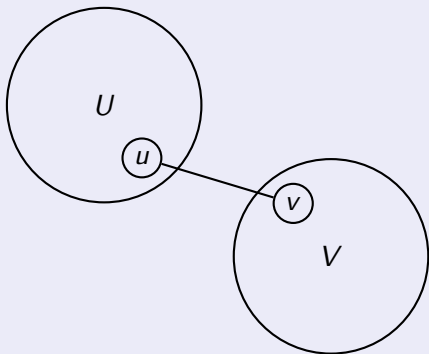
Beweis (Algorithmus von Kruskal - Fortsetzung)

Nehmen wir zum Schluss an, dass am Ende des Algorithmus mehr als eine Knotenmenge verbleibt.

Dann gibt es mindestens noch zwei Knotenmengen U und V . Da G aber zusammenhängend ist, gibt es mindestens einen Weg zwischen ihnen und somit auch eine Kante $e = (u, v)$ mit $u \in U$ und $v \in V$.

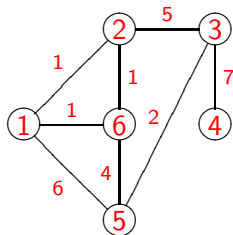
Diese Kante wurde im Laufe des Algorithmus betrachtet. Da zum Schluss $\text{find}(u) \neq \text{find}(v)$ gilt, galt dies auch zu dem Zeitpunkt zu dem e bearbeitet wurde. Damit wäre e zum Baum hinzugefügt worden.

Das führt direkt zu einem Widerspruch. Daher kann am Ende nur eine Knotenmenge vorliegen. □

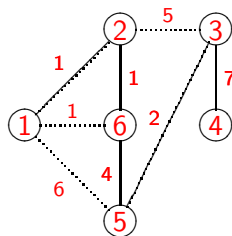


Ein Beispiel

Ursprünglicher Graph:



Konstruierte Bäume:



Gewicht des Minimalen Spannbaumes: 15

Teil 5: Algorithmen auf Graphen
Minimale Spannbäume

Der Algorithmus von Prim

Der Algorithmus von Prim

Beim Algorithmus von Kruskal beginnt man mit einem **Spannwald** mit $|V|$ Bäumen. Durch die Hinzunahme von Kanten wird die Anzahl der Bäume nach und nach um eins gesenkt, bis am Ende ein Spannbaum vorliegt.

Beim **Algorithmus von Prim** geht man etwas anders vor. Hier hat man zu jedem Zeitpunkt nur **einen** Baum, der Teil des späteren minimalen Spannbaumes ist.

In jedem Schritt wird dieser Baum um einen Knoten erweitert, indem eine Kante hinzugefügt wird, die zu einem Knoten des Baumes und zu einem Knoten inzident ist, der noch nicht im Baum liegt.

Da das Gesamtgewicht der Kanten minimal sein soll, wird dabei immer die mögliche Kante mit kleinstem Gewicht gewählt. D.h. wir haben wieder einen Greedy-Algorithmus.

Algorithmus (Naive Realisierung des Algorithmus von Prim)

Eingabe: Ein ungerichteter Graph $G = (V, E)$ mit nicht-negativen Kantengewichten $c: E \rightarrow \mathbb{R}_{\geq 0}$.

Ausgabe: Ein minimaler Spannbaum T von G , repräsentiert als Kantenliste.

Daten: Eine Kantenmenge M und eine Knotenmenge U .

```
 $T := \emptyset, M := \emptyset$  //  $O(1)$   
Wähle  $v \in V$  beliebig und setze  $U := \{v\}$  //  $O(1)$   
solange  $B \neq V$  tue //  $|V| - 1$  Durchläufe  
    für alle  $(v, u) \in E$  tue //  $O(|E|)$   
        wenn  $|\{v, u\} \cap U| = 1$  dann  $M := M \cup \{(v, u)\}$   
    Ende  
    Suche eine Kante  $e = (x, y) \in M$  mit minimalem  $c(e)$  //  $O(|E|)$   
     $U := U \cup \{x, y\}$   
     $T := T \cup \{e\}$   
     $M := \emptyset$  //  $O(1)$   
Ende
```

Laufzeit der primitiven Implementation.

Wie bereits gesehen, benötigt die Initialisierung konstante Zeit.

Da in jedem Durchlauf der äußeren Schleife ein Knoten hinzugefügt wird, wird sie für jeden Knoten, außer dem ersten, genau einmal durchlaufen, d.h. insgesamt $|V| - 1$ Mal.

In jedem Durchlauf werden alle Kanten überprüft, ob genau eines ihrer Enden unter den bereits hinzugefügten Knoten ist. Dies benötigt, zusammen mit der Suche nach der Kante mit minimalen Kosten unter den möglichen, $O(|E|)$ Zeit.

Der Rest schlägt mit konstanter Zeit pro Durchlauf zu Buche.

Satz

Die naive Implementierung der Algorithmus von Prim berechnet zu einem gegebenen zusammenhängenden, ungerichteten Graphen $G = (V, E)$ mit nicht-negativen Kantengewichten, einen minimalen Spannbaum in Zeit $O(|V| \cdot |E|)$.

Beweis (Korrektheit des Algorithmus von Prim)

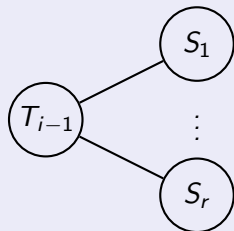
Ähnlich zum Beweis der Korrektheit des Algorithmus von Kruskal zeigen wir, dass für jedes $i \geq 1$ der nach dem Hinzufügen des i -ten Knoten vorliegende Baum T_i zu einem minimalen Spannbaum ergänzt werden kann.

Für $i = 1$, d.h. nach dem Hinzufügen des ersten willkürlich gewählten Knotens, ist die Behauptung offensichtlich erfüllt.

Sei nun $i > 1$ und v_1, \dots, v_i seien die Knoten und e_2, \dots, e_i die Kanten in T_i . Dabei sei $e_i = (v_j, v_i)$, mit $1 \leq j < i$, die zuletzt hinzugefügte Kante. Nach Induktionsvoraussetzung kann der Baum T_{i-1} , mit den Knoten v_1, \dots, v_{i-1} und den Kanten e_2, \dots, e_{i-1} , zu einem minimalen Spannbaum T' ergänzt werden.

Ist e_i in T' enthalten, so ist T' auch eine Ergänzung von T_i .

Beweis (Korrektheit des Algorithmus von Prim - Fortsetzung)



Sei nun $e_i \notin T'$. Da T_{i-1} ein Teilbaum von T' ist, besteht $T \setminus T_{i-1}$ aus einer Reihe S_1, \dots, S_r von Bäumen, von denen jeder über genau eine Kante mit T_{i-1} verbunden ist. Untereinander sind sie nicht verbunden.

O.B.d.A. sei $v_i \in S_1$ und $e = (u, v)$ sei die Verbindung zwischen T_{i-1} und S_1 mit $v \in T_{i-1}$ und $u \in S_1$. Insbesondere gilt auch $e \neq e_i$.

Da v_{i-1} und v in T_{i-1} verbunden sind und u und v_i in S_1 , ergibt sich ein Kreis, auf dem sowohl e_i als auch e liegt.

Beweis (Korrektheit des Algorithmus von Prim - Fortsetzung)

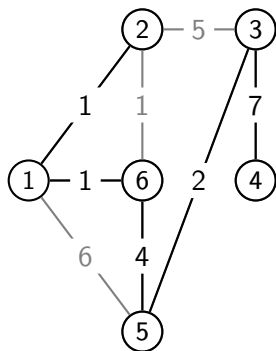
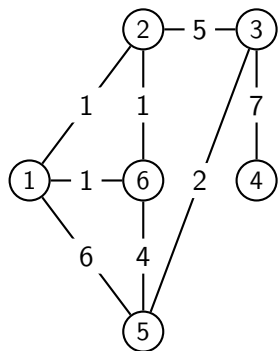
Offensichtlich ist $T'' := (T \setminus \{e\}) \cup \{e_i\}$ ebenfalls ein Spannbaum von G und es gilt

$$c(T'') = c(T) - c(e) + c(e_i).$$

Da e_i mit minimalem Gewicht unter allen von T_{i-1} ausgehenden Kanten gewählt wurde, gilt $c(e_i) \leq c(e)$ und somit $c(T) \leq c(T'') \leq c(T)$.

Damit ist T'' ein minimaler Spannbaum, der T_i ergänzt. □

Ein Beispiel



Eine bessere Implementierung von Prim's Algorithmus

Die vorgestellte Implementierung muss für jeden Knoten alle Kanten des Graphen überprüfen, ob sie ein Ende in dem bereits konstruierten Baum und eines außerhalb haben. Dieses Vorgehen benötigt unnötig viel Zeit.

Besser wäre es für jeden Knoten mitzuführen, ob er zum bisher konstruierten Baum benachbart ist, und welche Kante diese Verbindung mit minimalen Kosten herstellt.

Dann wäre es in jeder Runde nur nötig den erreichbaren Knoten mit minimalem Gewicht zu bestimmen und zum Baum hinzuzufügen.

Zusätzlich müssten nach dem Hinzufügen eines Knotens lediglich alle Nachbarn, die noch nicht zum Baum hinzugefügt worden sind, aktualisiert werden.

Für die Verwaltung dieser Knoten bietet sich eine **Priority Queue** an.

Algorithmus (Der Algorithmus von Prim)

Eingabe: Ein ungerichteter Graph $G = (V, E)$ mit nicht-negativen Kantengewichten $c: E \rightarrow \mathbb{R}_{\geq 0}$.

Ausgabe: Ein minimaler Spannbaum T von G , repräsentiert als Kantenliste.

Daten: Eine Priority Queue Q mit Kanten als Schlüsseln, geordnet nach Gewicht.

```
 $T := \emptyset$   
für alle  $v \in V$  tue  $Q.insert(v, \infty)$  // Alle Knoten in die Queue  
solange  $Q \neq \emptyset$  tue  
   $v := Q.min()$  // Hole den nächsten Knoten  
   $e := Q.key(v)$  // Hole die verbindende Kante  
   $Q.delete\_min()$   
   $T := T \cup \{e\}$   
  für alle  $u \in adj(v)$  tue // Prüfe alle Nachbarn  
    wenn  $u \in Q$  und  $w(u, v) < w(Q.key(u))$  dann  
       $Q.decrease\_key(u, (u, v))$   
    Ende  
  Ende  
Ende
```

Laufzeit des Algorithmus von Prim

Wenn wir die Priority Queue mittels eines **Fibonacci Heaps** realisieren, können wir die Laufzeit des Algorithmus von Prim über die amortisierten Kosten bestimmen.

Wir fügen zu Beginn insgesamt $|V|$ Knoten ein, was zu amortisierten Kosten von $O(|V|)$ führt.

Anschließend führen wir insgesamt $|V|$ -mal `delete_min` aus, was amortisierte Kosten $O(|V| \log |V|)$ verursacht.

Da jede Kante von zwei Enden betrachtet wird, aber nur einmal pro Kante ein `decrease_key` ausgeführt wird, erhalten wir amortisierte Kosten von $O(|E|)$ für die Aktualisierungen der Schlüssel.

Satz

Unter Verwendung eines Fibonacci Heaps benötigt der Algorithmus von Prim $O(|E| + |V| \log |V|)$ Zeit zur Berechnung eines minimalen Spannbaumes eines ungerichteten, zusammenhängenden Graphen $G = (V, E)$ mit nicht-negativen Kantengewichten.