Vorlesung Datenstrukturen

Minimale Spannbäume

Maike Buchin

18.7., 20.7.2017

Einführung

Motivation: Verbinde Inseln mit Fähren oder Städte mit Schienen und verbrauche dabei möglichst wenig Länge.

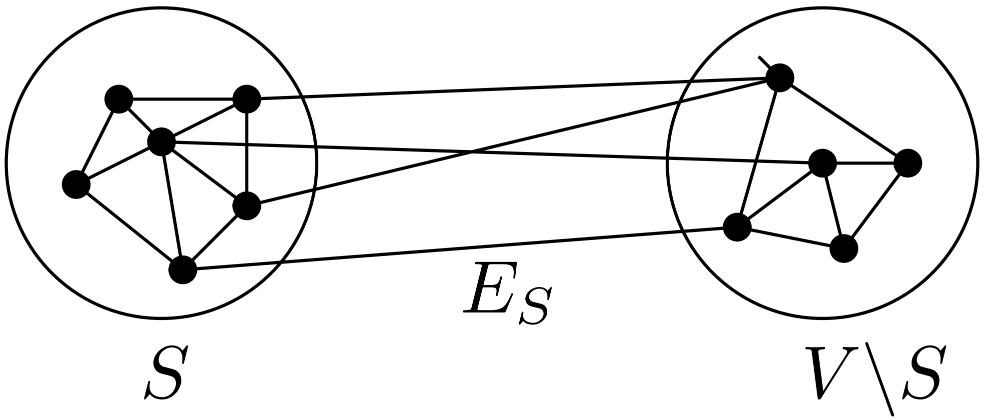
Problem: Gegeben ist ein zusammenhängender ungerichteter Graph $G = (V, E)$ mit positiven reellen Kantenkosten $c\colon E \to \mathbb{R}^+$. Gesucht ist ein minimaler Spannbaum (MST) von G, d.h. eine Menge $T \subseteq E$, so dass $G' = (V, T)$ ein zusammenhängender Baum mit minimalen Kosten $c(T) = \sum_{e \in T} c(e)$ ist.

Vorgehen: Wir leiten zunächst zwei grundlegende Eigenschaften her, die Grundlage der folgenden Algorithmen sein werden.

11.1 Schnitteigenschaft und Kreiseigenschaft

Wir zeigen zwei Lemmata. Eines erlaubt, Kanten in einen entstehenden MST aufzunehmen, das andere Kanten auszuschliessen.

Definition: Ein Schnitt in einem Graphen ist eine Partitionierung $S, V \backslash S$ der Knotenmenge V in zwei nicht-leere Teile. Dazu gehört die Kantenmenge $E_S = \{\{u, v\} \in E \,|\, u \in S, v \in V \backslash S\}$.

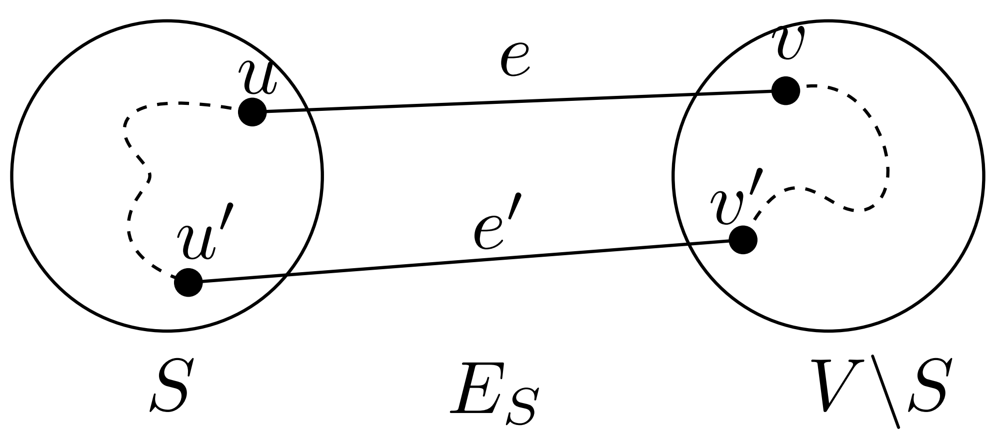


11.1 Schnitteigenschaft und Kreiseigenschaft

RUB

Lemma (Schnitteigenschaft): Sei $S, V\backslash S$ ein Schnitt mit Kantenmenge E_S und $e \in E_S$ eine Kante mit minimalen Kosten in E_S. Sei weiter T' eine Kantenmenge, die in einem MST enthalten ist und keine Kante aus E_S enthält.

Beweis: Betrachte MST T von G mit $T' \subseteq T$. Die Endpunkte von e seien $u \in S$ und $v \in V\backslash S$. Da T MST ist, gibt es einen Weg P von u nach v in T. P muss eine Kante $e' = (u', v') \in E_S$ enthalten. Nach Voraussetzung $e' \notin T'$. Dann bildet $T'' := (T\backslash\{e'\}) \cup \{e\}$ ebenfalls einen Spannbaum. Da $c(e)$ in E_S minimal ist, folgt $c(T'') \leq c(T)$. Also ist T'' ebenfalls MST (mit $c(T'') = c(T)$) und $T' \cup \{e\} \subseteq T''$.



11.1 Schnitteigenschaft und Kreiseigenschaft

RUB

Lemma (Kreiseigenschaft): Es sei C ein einfacher Kreis in G und es sei e ein Kante in C mit maximalen Kosten. Dann ist jeder MST von $G' = (V, E \backslash \{e\})$ auch ein MST von G.

Beweis: Ein Spannbaum von G' ist ebenfalls Spannbaum von G.
Also genügt zu zeigen, dass sich minimale Kosten ebenfalls übertragen.
Betrachte dazu MST T von G. Wenn $e \notin T$, folgt sofort, dass alle MSTs von G' die gleichen Kosten wie T haben.
Nehmen wir nun an $e = (u, v) \in T$. Wenn man e aus T entfernt, entstehen zwei Teilbäume T_u und T_v mit $u \in T_u$ und $v \in T_v$. Da C ein Kreis, gibt es eine Kante $e' = (u', v') \neq e$ auf C mit $u' \in T_u$ und $v' \in T_v$.
Dann ist $T' := (T \backslash \{e'\}) \cup \{e\}$ ebenfalls ein Spannbaum in G' mit $c(T') = c(T) - c(e) + c(e) \leq c(T)$.
Also folgt, dass alle MSTs in G' die gleichen Kosten wie T haben.

11.1 Schnitteigenschaft und Kreiseigenschaft

RUB

Die **Schnitteigenschaft** führt zu einer einfachen Greedy-Strategie: beginne mit der leeren Kantenmenge $T' = \emptyset$. Solange T' kein Spannbaum ist, wiederhole folgenden Schritt: Suche einen Schnitt $S, V \setminus S$, so dass keine Kante aus T' die beiden Seiten verbindet. Füge eine Kante aus E_S mit minimalen Kanten hinzu.
Wir betrachten zwei Algorithmen, die diese Strategie umsetzen: Jarnik-Prim und Kruskal.

Die **Kreiseigenschaft** führt ebenfalls zu einer einfachen Strategie: Beginne mit der gesamten Kantenmenge und finde wiederholt Kreise und entferne deren teuerste Kante. Allerdings ist keine effiziente Umsetzung dieses Ansatzes bekannt.

11.2 Der Algorithmus von Jarnik-Prim

Der Algorithmus ähnelt dem Algorithmus von Dijkstra.

Vorgehen: Starte mit $S = \{s\}$ und $T = \emptyset$ für beliebigen Knoten s und baue MST auf, indem nach und nach Knoten zu S und Kanten zu T hinzugefügt werden.
Betrachte beliebige Iterationsrunde mit Knoten $S \subseteq V$ und Baumkanten T. Verwende den Schnitt $(S, V \backslash S)$. Wähle also Kante aus E_S mit minimalen Kosten, füge diese zu T und den Zielknoten zu S hinzu.
Um diese Kante in jedem Schritt effizient zu finden, verwende PrioWS Q für Knoten in $V \backslash S$ mit minimalen Kantenkosten zu S. Verwende ebenfalls Arrays d und p für Abstände und Vorgängerzeiger. Wenn ein Knoten zu S hinzugefügt wird, erniedrige ggfs. die Abstände zu inzidenten Knoten in $V \backslash S$.

Pseudocode:

Function $jpMST$: Set **of** $Edge$

$d = \langle \infty, \dots, \infty \rangle : NodeArray[1..n]$ **of** $\mathbb{R} \cup \{\infty\}$ // $d[u]$ gibt geringste Kosten // einer Kante von S nach u an

$parent : NodeArray$ **of** $NodeId$ // $\{parent[u], u\}$ ist eine billigste Kante von S nach u

$Q : NodePQ$ // benutzt $d[\cdot]$ als Priorität

$Q.insert(s)$ für ein beliebiges $s \in V$

while $Q \neq \emptyset$ **do**

  $u := Q.deleteMin$

  $d[u] := 0$ // $d[u] = 0$ kodiert $u \in S$

  foreach $edge\ e = \{u, w\} \in E$ **do**

    if $c(e) < d[w]$ **then** // $c(e) < d[w]$ impliziert $d[w] > 0$ und daher $w \notin S$

      $d[w] := c(e)$

      $parent[w] := u$

      if $w \in Q$ **then** $Q.decreaseKey(w)$ **else** $Q.insert(w)$

  invariant $\forall w \in Q : d[w] = \min\{c(\{v, w\}) : \{v, w\} \in E \wedge v \in S\}$

return $\{\{parent[v], v\} : v \in V \setminus \{s\}\}$

Laufzeit: überträgt sich von Dijkstra, dh. $O(n \log n + m)$ bei Verwendung von Fibonacci-heaps.

11.3 Der Algorithmus von Kruskal

Vorgehen: Betrachte Kanten in der Reihenfolge aufsteigender Kosten. Führe eine kreisfreie Kantenmenge T, einen Wald, mit, welche anfangs leer ist. Invariante: T kann zu einem MST erweitert werden. Wenn eine Kante zwei Komponenten verbindet, wird sie zu T hinzugenommen, sonst nicht. Sei (u, v) die Kante, S_u die Komponente, die u enthält, dann verwende Schnitt $(S_u, V \backslash S_u)$

Pseudocode:

Function *kruskalMST*(V, E, c) : *Set* **of** *Edge*
    $T := \emptyset$
    foreach $\{u, v\} \in E$ in der Reihenfolge steigender Kosten **do**
$\star$ **if** u und v liegen in verschiedenen Bäumen von T **then**
            $T := T \cup \{\{u, v\}\}$ // vereinige zwei Bäume
        invariant T ist ein Wald, der in einem MST enthalten ist
    return T

Laufzeit: Wir werden sehen, dass sich Test $\star$ so effizient implementieren lässt, dass das Sortieren der Kanten $O(m \log m)$ dominiert.

11.4 Die Union-Find-Datenstruktur

Im Algorithmus von Kruskal wird die Knotenmenge V durch den Wald T in Blöcke ($=$ Zushgs.komponenten) partitioniert.

Die Union-Find Datenstruktur verwaltet eine Partitionierung einer Menge $1 \dots n$ unter den Operationen Find und Union:

- Find(v) gibt Repräsentanten des Blockes von v wider
- Union(r,s) vereinigt die Blöcke mit Repräsentanten r, s

Umsetzung der Union-Find Datenstruktur:
Jeder Block als Baum mit Wurzel als Repräsentant.
Speichere Vorgänger im Baum in Array parent.

Diese Umsetzung wird effizient bei Verwendung von

- Vereinigung nach Rang
- Pfadkompression

Analyse: siehe Buch