

3.5 Prim's Algorithmus, zweite Variante

Die Idee der folgenden Variante von Prim's Algorithmus ist:

Lasse die Priority Queues nicht zu groß werden.

Seien dazu $G = (V, E)$, $|V| = n$, $|E| = m$, w Gewichtsfunktion, und k ein Parameter, dessen Wert wir erst später festlegen werden.

Der Algorithmus arbeitet nun in **Phasen** wie folgt:

- 1 Initialisiere eine Schlange von Bäumen, jeder Baum anfangs ein (Super-) Knoten. Zu jedem Baum initialisiere eine Priority Queue (Fibonacci-Heap) mit den Nachbarn der Knoten im Baum, die selbst nicht im Baum sind, als Elementen und jeweils dem Gewicht einer leichtesten Kante zu einem Knoten im Baum als Schlüssel.
- 2 Markiere jeden Baum in der Schlange mit der Nummer der laufenden Phase.

3

while vorderster Baum in der Schlange hat laufende Phasennummer **do**

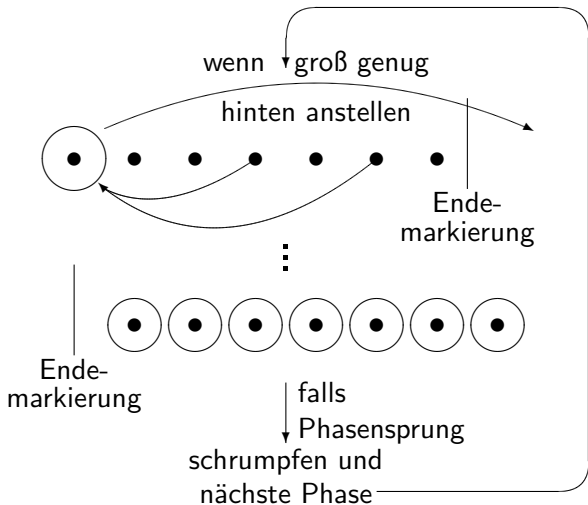
lasse ihn wachsen, solange seine Priority Queue höchstens k Elemente enthält (und noch etwas zu tun ist)

if Priority Queue zu groß (mehr als k Elemente) **then**
füge Baum mit inkrementierter Phasennummer am Ende der Schlange ein

fi

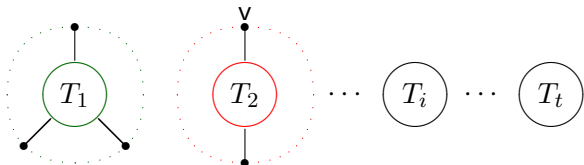
od

- 4 Falls „Phasensprung“: Schrumpfe alle Bäume zu Superknoten, reduziere Kantenmenge (d.h., behalte zwischen zwei Knoten jeweils nur die leichteste Kante)
- 5 Beginne nächste Phase mit Schritt 1.!



Analyse des Zeitbedarfs:

Sei t die Anzahl der Bäume in der Schlange zu Beginn einer Phase.
Betrachte die Schlange von Bäumen:



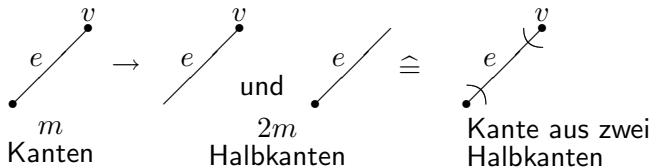
Abgesehen von den Operationen, die bei der Vereinigung von Superknoten anfallen, beträgt der Zeitaufwand pro Phase $\mathcal{O}(m)$.

Vereinigung zweier Superknoten, z.B. T_1 und T_2 :

Für jeden (Super-)Knoten v in T_2 's Priority Queue:

- 1 $v \in T_1$: \surd (nichts zu tun)
- 2 v in Priority Queue von T_1 : *DecreaseKey*. Hilfsdatenstruktur: für alle Knoten in den Priority Queues ein Zeiger auf den Superknoten, in dessen Priority Queue der Knoten letztmals am Anfang der Schlange stand.
- 3 sonst: Einfügen

Betrachte Knoten mit „Halbkanten“: jede Halbkante kommt nur 1x in allen Bäumen der Queue vor. Mit m Kanten ergeben sich $2m$ Halbkanten.



Zeitaufwand pro Phase (mit Bildung der Superknoten zu Beginn):

- Initialisierung: $\mathcal{O}(m)$
- *ExtractMin*: $< t$ Operationen
- sonstige Priority Queue-Operationen, Overhead: Zeit $\mathcal{O}(m)$

Da die Priority Queues höchstens k Elemente enthalten, wenn darauf eine „teure“ Priority Queue-Operation durchgeführt wird, sind die Kosten pro Phase

$$\mathcal{O}(t \log k + m).$$

Setze $k = 2^{\frac{2m}{t}}$ (damit $t \log k = 2m$). Damit sind die Kosten pro Phase $\mathcal{O}(m)$.

Wieviele Phasen führt der Algorithmus aus?

t ist die Anzahl der Superknoten am Anfang einer Phase, t' sei diese Zahl zu Beginn der nächsten Phase. Sei a die durchschnittliche Anzahl ursprünglicher Knoten in jeder der t Priority Queues zu Anfang der Phase, a' entsprechend zu Beginn der nächsten Phase.

Wir haben:

- 1 $a = \frac{2m}{t}$
- 2 $t' \leq \frac{2m}{k}$ (mit Ausnahme ev. der letzten Phase)

Also:

$$a' = \frac{2m}{t'} \geq k = 2^{\frac{2m}{t}} = 2^a$$

Für die erste Phase ist $a = \frac{2m}{n}$, für jede Phase ist $a \leq n - 1$. Also ist die Anzahl der Phasen

$$\leq 1 + \min \left\{ i; \log_2^{(i)}(n-1) \leq \frac{2m}{n} \right\}.$$

Setzen wir $\beta(m, n) := \min \left\{ i; \log_2^{(i)} n \leq \frac{m}{n} \right\}$, dann gilt

$$\beta(m, n) \leq \log^* n \text{ für } n \leq m \leq \binom{n}{2}.$$

Satz 106

Für gewichtete, zusammenhängende (ungerichtete) Graphen mit n Knoten, m Kanten kann ein minimaler Spannbaum in Zeit $\mathcal{O}(\min\{m \cdot \beta(m, n), m + n \log n\})$ bestimmt werden.

3.6 Erweiterungen

Euklidische minimale Spannbäume stellen ein Problem dar, für das es speziellere Algorithmen gibt. Literatur hierzu:



Andrew Chih-Chi Yao:

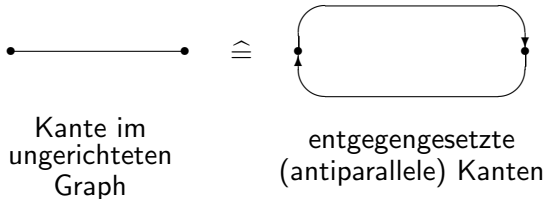
On constructing minimum spanning trees in k -dimensional spaces and related problems

SIAM J. Comput. **11**(4), pp. 721–736 (1982)

Kapitel V Kürzeste Pfade

1. Grundlegende Begriffe

Betrachte Digraph $G = (V, A)$ oder Graph $G = (V, E)$.



Distanzfunktion:

$$d : A \longrightarrow \mathbb{R}^+ \text{ (bzw. } \longrightarrow \mathbb{R})$$

O.B.d.A.: $A = V \times V$, $d(x, y) = +\infty$ für Kanten, die eigentlich nicht vorhanden sind.

$\text{dis}(v, w) :=$ Länge eines kürzesten Pfades von v nach w
 $\in \mathbb{R}^+ \cup \{+\infty\}$.

Arten von Kürzeste-Pfade-Problemen:

- 1 single-pair-shortest-path (**spsp**). Beispiel: Kürzeste Entfernung von München nach Frankfurt.
- 2 single-source-shortest-path: gegeben G , d und $s \in V$, bestimme für alle $v \in V$ die Länge eines kürzesten Pfades von s nach v (bzw. einen kürzesten Pfad von s nach v) (**sssp**).
Beispiel: Kürzeste Entfernung von München nach allen anderen Großstädten.
- 3 all-pairs-shortest-path (**apsp**). Beispiel: Kürzeste Entfernung zwischen allen Großstädten.

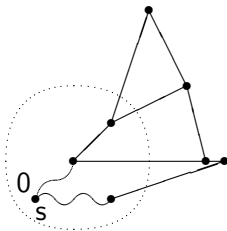
Bemerkung: Es gibt keinen Algorithmus, der das single-pair-shortest-path berechnet, ohne nicht gleichzeitig das single-source-shortest-path-Problem zu lösen.

2. Das single-source-shortest-path-Problem

Zunächst nehmen wir an, dass $d \geq 0$ ist. Alle kürzesten Pfade von a nach b sind o.B.d.A. einfache Pfade.

2.1 Dijkstra's Algorithmus

Gegeben: $G = (V, A)$, ($A = V \times V$),
Distanzfunktion $d : A \rightarrow \mathbb{R}^+ \cup \{+\infty\}$,
Startknoten s , G durch Adjazenzlisten dargestellt.



algorithm sssp:=

$S := \{s\}$; $\text{dis}[s] := 0$; initialisiere eine Priority Queue PQ , die alle Knoten $v \in V \setminus \{s\}$ enthält mit Schlüssel $\text{dis}[v] := d(s, v)$

for alle $v \in V - \{s\}$ **do** $\text{from}[v] := s$ **od**

while $S \neq V$ **do**

$v := \text{ExtractMin}(PQ)$

$S := S \cup \{v\}$

for alle $w \in V \setminus S$, $d(v, w) < \infty$ **do**

if $\text{dis}[v] + d(v, w) < \text{dis}[w]$ **then**

$\text{DecreaseKey}(w, \text{dis}[v] + d(v, w))$

co DecreaseKey aktualisiert $\text{dis}[w]$ **oc**

$\text{from}[w] := v$

fi

od

od

Seien $n = |V|$ und $m =$ die Anzahl der wirklichen Kanten in G .
Laufzeit (mit Fibonacci-Heaps):

Initialisierung:	$\mathcal{O}(n)$
<i>ExtractMin</i> :	$n \cdot \mathcal{O}(\log n)$
Sonstiger Aufwand:	$m \cdot \mathcal{O}(1)$ (z.B. <i>DecreaseKey</i>)

⇒ Zeitbedarf also: $\mathcal{O}(m + n \log n)$

Korrektheit: Wir behaupten, dass in dem Moment, in dem ein $v \in V \setminus \{s\}$ Ergebnis der *ExtractMin* Operation ist, der Wert $\text{dis}[v]$ des Schlüssels von v gleich der Länge eines kürzesten Pfades von s nach v ist.

Beweis:

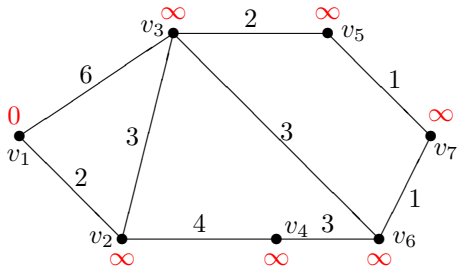
[durch Widerspruch] Sei $v \in V \setminus \{s\}$ der erste Knoten, für den diese Behauptung nicht stimmt, und sei



ein kürzester Pfad von s nach v , mit einer Länge $< \text{dis}[v]$. Dabei sind $s_1, \dots, s_r \in S$, $v_1 \notin S$ [$r = 0$ und/oder $q = 0$ ist möglich].

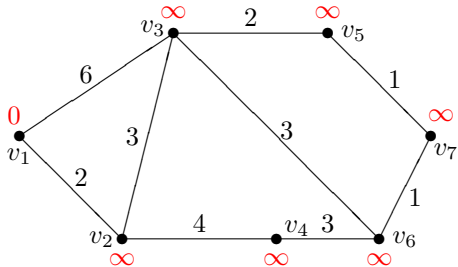
Betrachte den Pfad $s \rightarrow s_1 \rightarrow \dots \rightarrow s_r \rightarrow v_1$; seine Länge ist $< \text{dis}[v]$, für $q \geq 1$ (ebenso für $q = 0$) ist also $\text{dis}[v_1] < \text{dis}[v]$, im Widerspruch zur Wahl von v . □

Beispiel 107 (Dijkstra's Algorithmus)

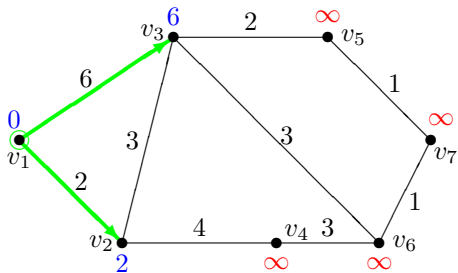


gegeben Graph G ;
 v_1 ist der Startknoten;
setze v_1 als Bezugsknoten;
setze $\text{dis}[v_1] = 0$;
setze $\text{dis}[\text{Rest}] = +\infty$;

Beispiel 107 (Dijkstra's Algorithmus)

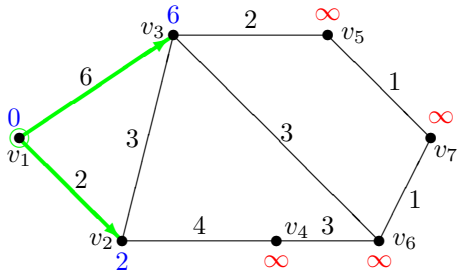


gegeben Graph G ;
 v_1 ist der Startknoten;
 setze v_1 als Bezugsknoten;
 setze $dis[v_1] = 0$;
 setze $dis[\text{Rest}] = +\infty$;

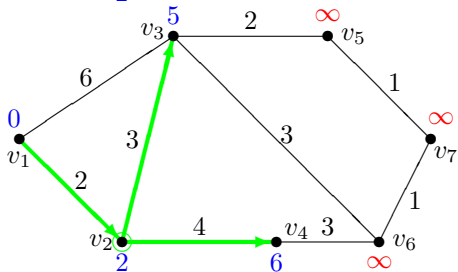


setze $dis[v_2] = 2$;
 markiere (v_1, v_2) ;
 setze $dis[v_3] = 6$;
 markiere (v_1, v_3) ;
 setze v_2 als Bezugsknoten,
 da $dis[v_2]$ minimal;

Beispiel 107 (Dijkstra's Algorithmus)

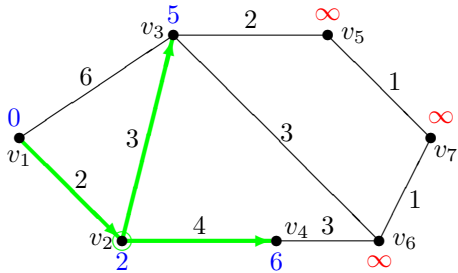


setze $\text{dis}[v_2] = 2$;
 markiere (v_1, v_2) ;
 setze $\text{dis}[v_3] = 6$;
 markiere (v_1, v_3) ;
 setze v_2 als Bezugsknoten,
 da $\text{dis}[v_2]$ minimal;

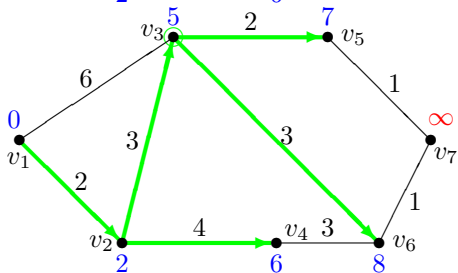


setze $\text{dis}[v_3] = 2 + 3 = 5$;
 markiere (v_2, v_3) ;
 unmarkiere (v_1, v_3) ;
 setze $\text{dis}[v_4] = 2 + 4 = 6$;
 markiere (v_2, v_4) ;
 setze v_3 als Bezugsknoten,
 da $\text{dis}[v_3]$ minimal;

Beispiel 107 (Dijkstra's Algorithmus)

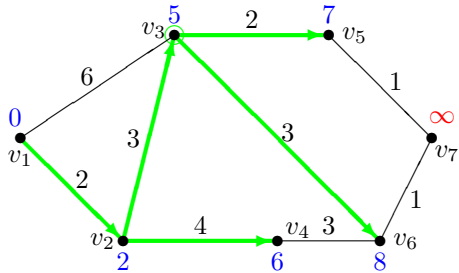


setze $\text{dis}[v_3] = 2 + 3 = 5$;
 markiere (v_2, v_3) ;
 unmarkiere (v_1, v_3) ;
 setze $\text{dis}[v_4] = 2 + 4 = 6$;
 markiere (v_2, v_4) ;
 setze v_3 als Bezugsknoten,
 da $\text{dis}[v_3]$ minimal;

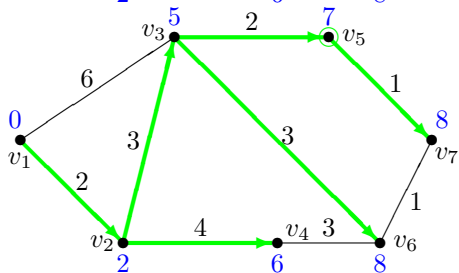


setze $\text{dis}[v_5] = 5 + 2 = 7$;
 markiere (v_3, v_5) ;
 setze $\text{dis}[v_6] = 5 + 3 = 8$;
 markiere (v_3, v_6) ;
 setze v_4 , dann v_5
 als Bezugsknoten;

Beispiel 107 (Dijkstra's Algorithmus)



setze $\text{dis}[v_5] = 5 + 2 = 7$;
markiere (v_3, v_5) ;
setze $\text{dis}[v_6] = 5 + 3 = 8$;
markiere (v_3, v_6) ;
setze v_4 , dann v_5
als Bezugsknoten;



setze $\text{dis}[v_7] := 7 + 1 = 8$;
markiere (v_5, v_7) ;
alle Knoten wurden erreicht:
 \Rightarrow Algorithmus zu Ende

Beobachtung:

- *ExtractMin* liefert eine (schwach) monoton steigende Folge von Schlüsseln $\text{dis}[\cdot]$;
- Die Schlüssel in PQ sind stets $\leq \text{dis}[v] + C$, wobei v das Ergebnis der vorangehenden *ExtractMin*-Operation (bzw. s zu Beginn) und $C := \max_{u,v \in A} \{\text{dis}(u, v)\}$ ist.


Satz 108

Dijkstra's Algorithmus (mit Fibonacci-Heaps) löst das single-source-shortest-path-Problem in Zeit $\mathcal{O}(m + n \log n)$.

Bemerkung:

- 1 Verwendet man Dijkstra's Algorithmus mit d -Heaps, so erhält man Laufzeit

$$\mathcal{O}(m \log_{2+\frac{m}{n}} n)$$

- 2  **Mikkel Thorup:**
Undirected single-source shortest paths with positive integer weights in linear time
J. ACM **46**(3), pp. 362–394 (1999)