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.

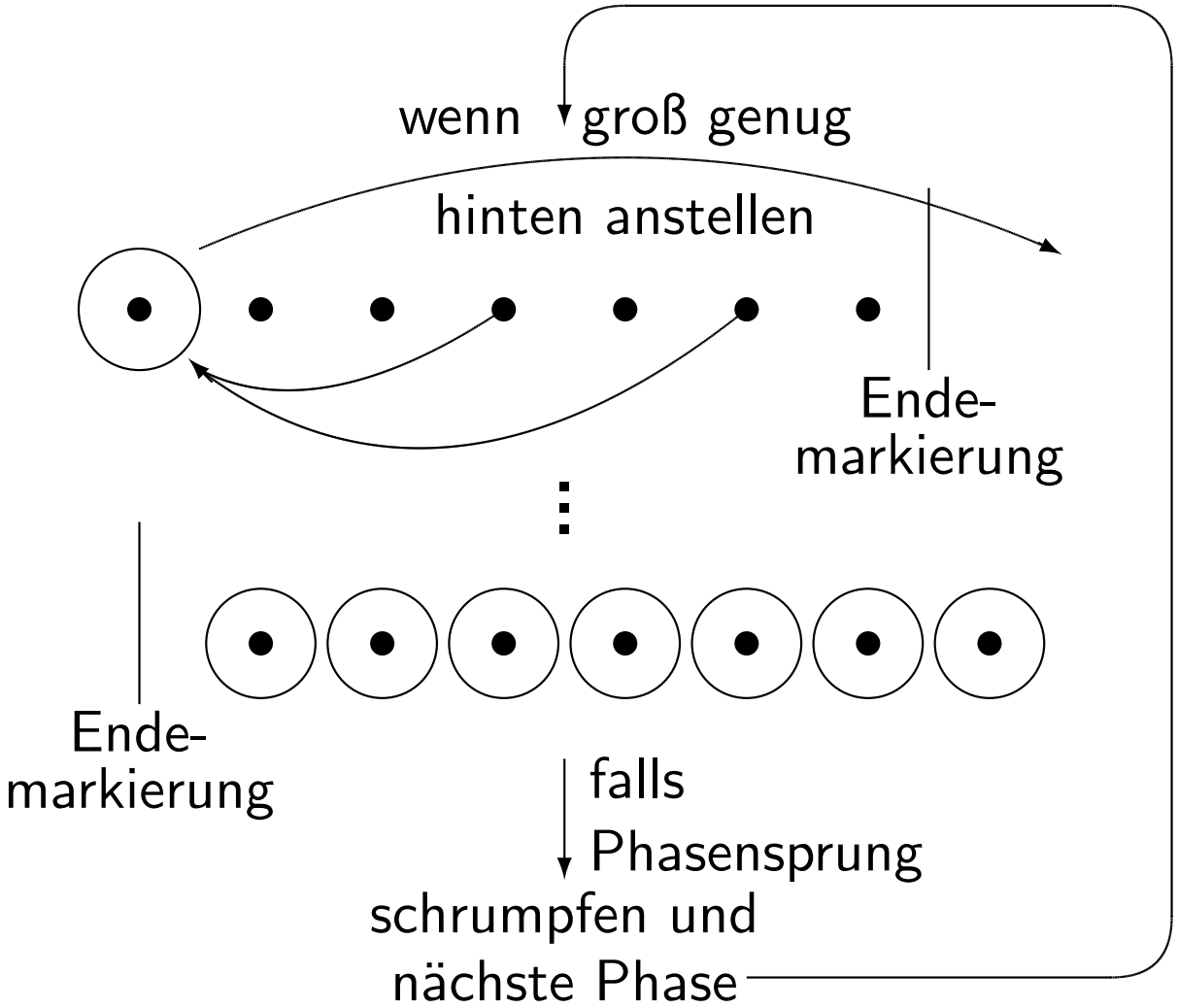
Der Algorithmus arbeitet nun in Phasen wie folgt:

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. Bestimme k für die Phase
4.

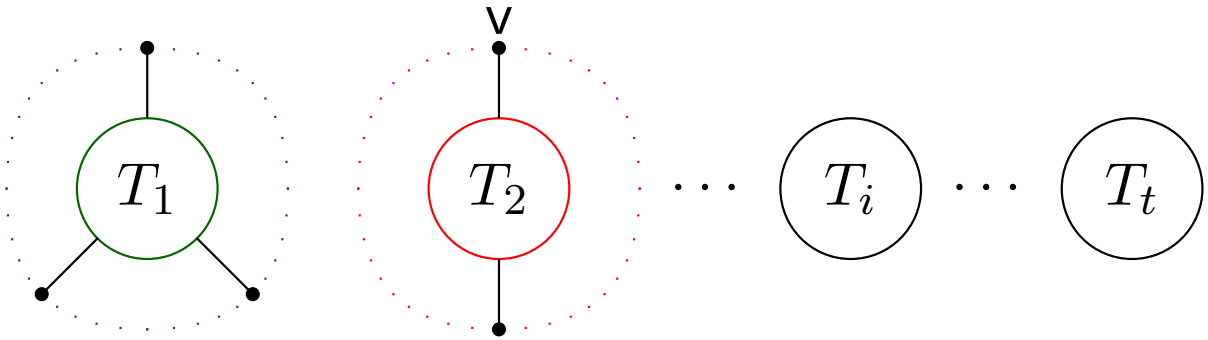
while vorderster Baum in der Schlange hat laufende Phasennummer **do**
    lasse ihn wachsen, solange seine Priority Queue höchstens k Elemente enthält (und noch etwas zu tun ist)
    if Priority Queue zu groß (mehr als k Elemente) **then**
        füge Baum mit inkrementierter Phasennummer am Ende der Schlange ein
    fi
od

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


wenn groß genug
hinten anstellen
Ende-
markierung
Ende-
markierung
falls
Phasensprung
schrumpfen und
nächste Phase

Analyse des Zeitbedarfs:

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

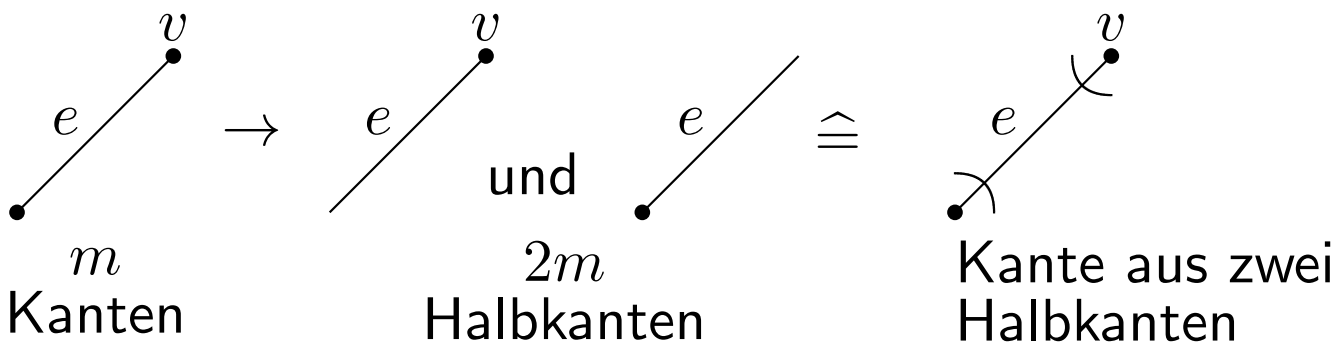


Abgesehen von den Operationen, die bei der Vereinigung von Superknoten anfallen, beträgt der Zeitaufwand pro Phase $\mathcal{O}(m)$.

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 107

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

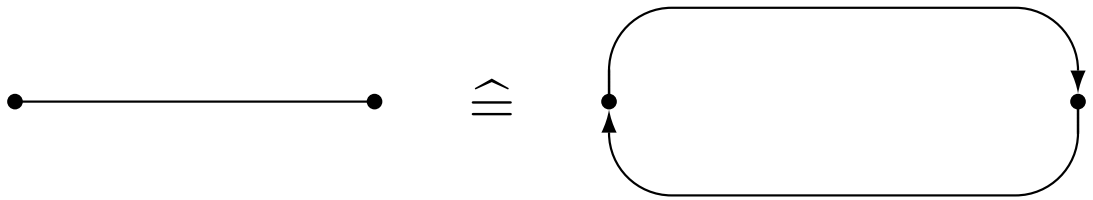
Euklidische minimale Spannbäume stellen ein Problem dar, für das es speziellere Algorithmen gibt. Literatur hierzu:

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 $G = (V, E)$.



Kante im ungerichteten Graph

entgegengesetzte (antiparallele) Kanten

Distanzfunktion: $\mathrm{d} : A \longrightarrow \mathbb{R}^+$ (bzw. $\longrightarrow \mathbb{R}$)
O.B.d.A.: $A = V \times V$, $\mathrm{d}(x, y) = +\infty$ für Kanten, die eigentlich nicht vorhanden sind.
$\mathrm{dis}(v, w) :=$ 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: Wir kennen keinen Algorithmus, der das single-pair-shortest-path berechnet, ohne nicht gleichzeitig (im worst-case) das single-source-shortest-path-Problem zu lösen.