Technische Universität München
Institut für Informatik

Effiziente Algorithmen und Datenstrukturen
WS 1997/98 - Kapitel 3

Stand: 10. März 1998

Dozent:

Prof. Dr. Ernst W. Mayr

Mitschrift:

Martin Stumpf
Thomas Schöpf
Martin Hans
Guido Wimmel

Inhaltsverzeichnis

Kapitel 3

Minimale Spannbäume

3.1 Grundlagen

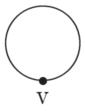
Graphen: ungerichtet, einfach, zusammenhängend, gewichtet.

$$G = (V, E)$$

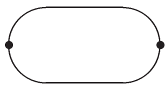
Knoten Kanten

$V = \{x, y, z, \dots\}$ $\{v_i, v_j\}; i \neq j$

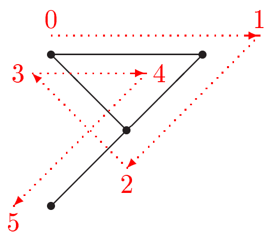
$\{v_1, \dots, v_n\}$



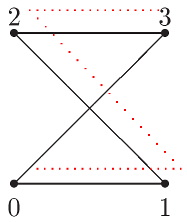
<u>keine</u> Schleifen erlaubt



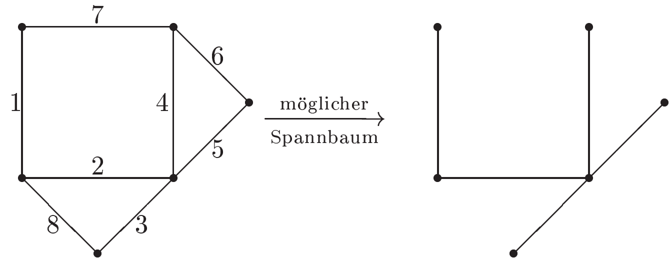
<u>keine</u> doppelten Kanten erlaubt



Weg 0-5 nicht knoteneinfach und nicht kanteneinfach



Weg 0-3 knoteneinfach und kanteneinfach



$G = (V, E)$
$T = (V, E')$ Baum
$E' \subseteq E$

Anwendungen:

- Telekom: Telefonvermittlungen
- Leiterplatinen

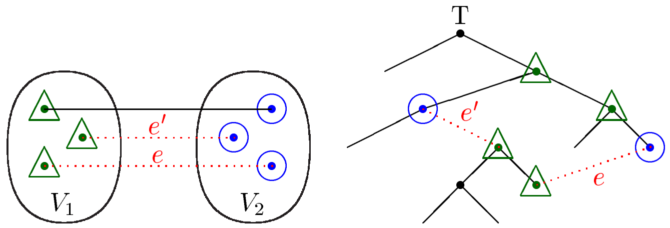
Zwei Prinzipien für die Konstruktion von minimalen Spannbäumen: „blaue“ Regel und „rote“ Regel (Tarjan).

Satz: Sei $G = (V, E)$ ein zusammenhängender, ungerichteter Graph, w: $E \to \mathbb{R}$ eine Gewichtsfunktion, $C = (V_1, V_2)$ ein Schnitt (d.h. $V = V_1 \cup V_2$, $V_1 \cap V_2 = \emptyset$, $V_1 \neq \emptyset \neq V_2$). Sei $E_C = E \cap (V_1 \times V_2)$ die Menge der Kanten „über den Schnitt hinweg“. Dann gilt: (**„blaue“ Regel**)

1. Ist $e \in E_C$ die einzige Kante minimalen Gewichts (über alle Kanten in E_C), dann ist e in jedem minimalen Spannbaum für $(G$,w$)$ enthalten.
2. Hat $e \in E_C$ minimales Gewicht (über alle Kanten in E_C), dann gibt es einen minimalen Spannbaum von $(G$,w$)$, der e enthält.

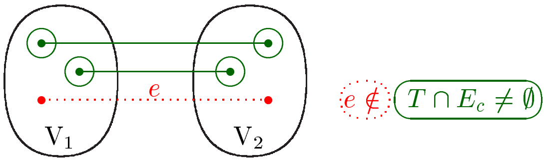
Beweis: durch Widerspruch

1. Sei T ein minimaler Spannbaum von $(G$,w$)$, sei $e \in E_C$ die minimale Kante. Annahme: $e \notin T$. Da T Spannbaum $\Rightarrow T \cap E_C \neq \emptyset$.
   Sei $T \cap E_C = \{e_1, e_2, \ldots, e_k\}, k \geq 1$. Dann enthält $T \cup \{e\}$ einen eindeutig bestimmten Kreis (den sogenannten durch e bzgl. T bestimmten Fundamentalkreis). Dieser Kreis muß mindestens eine Kante $\in E_c \cap T$ enthalten, da die beiden Endpunkte von e auf verschiedenen Seiten des Schnitts C liegen.

   Sei $e' \in E_c \cap T$. Dann gilt nach Vorraussetzung w(e') >w(e). Also ist $T' := T - \{e'\} \cup \{e\}$ ein Spannbaum von G, der echt kleineres Gewicht als T hat, Widerspruch zu T ist minimaler Spannbaum.

2. Sei $e \in E_c$ minimal. Annahme: e kommt in <u>keinem</u> minimalen Spannbaum vor. Sei T ein beliebiger minimaler Spannbaum von $(G$,w$)$.



$e \notin T \cap E_c \neq \emptyset$. Sei $e' \in E_c \cap T$ eine Kante auf dem durch e bezüglich T erzeugten Fundamentalkreis. Dann ist $T' = T - \{e'\} \cup \{e\}$ wieder ein Spannbaum von G, und es ist $\mathrm{w}(T') \leq \mathrm{w}(T)$. Also T' minimaler Spannbaum und $e \in T'$.

□

Satz: Sei $G = (V, E)$, $w : E \to \mathbb{R}$, ein gewichteter, zusammenhängender Graph. Dann gilt: (**„rote“ Regel**)

1. Gibt es zu $e \in E$ einen Kreis C in G, der e enthält und $w(e) > w(e') \quad \forall e' \in C \setminus \{e\}$ erfüllt, dann kommt e in keinem minimalen Spannbaum vor.
2. Ist $c_1 = e_1, \ldots, e_k$ ein Kreis in G und $w(e_i) = \max\{w(e_j) | 1 \leq j \leq k\}$, dann gibt es einen minimalen Spannbaum, der e_i nicht enthält.

Beweis:

1. Nehmen wir an, da es einen minimalen Spannbaum T gibt, der $e = \{v_1, v_2\}$ enthält. Wenn wir e aus T entfernen, so zerfällt T in zwei nicht zusammenhängende Teilbäume T_1 und T_2 mit $v_i \in T_i$, $i = 1, 2$. Da aber e auf einem Kreis in G liegt, muß es einen Weg von v_1 nach v_2 geben, der e nicht benützt. Mithin gibt es eine Kante $\hat{e} \neq e$ auf diesem Weg, die einen Knoten in T_1 mit T_2 verbindet. Verbinden wir T_1 und T_2 entlang $\hat{e}$, so erhalten wir einen von T verschiedenen Spannbaum $\hat{T}$. Wegen $w(\hat{e}) < w(e)$ folgt $w(\hat{T}) < w(T)$. Ein Widerspruch.

2. Wir nehmen an, e_i liege in jedem minimalen Spannbaum (MSB) von G, und zeigen die Behauptung durch Widerspruch.
   Sei T ein beliebiger MSB von G. Entfernen wir e_i aus T, so zerfällt T in zwei nicht zusammenhängende Teilbäume T_1 und T_2. Da e_i auf einem Kreis $c_1 = e_1, \ldots, e_k$ in G liegt, können wir wie unter 1. e_i durch eine Kante e_j des Kreises c_1 ersetzen, die T_1 und T_2 verbindet. Dadurch erhalten wir einen von T verschiedenen Spannbaum $\tilde{T}$, der e_i nicht enthält. Da nach Voraussetzung $w(e_j) \leq w(e_i)$ gilt, folgt $w(\tilde{T}) \leq w(T)$ (und sogar $w(\tilde{T}) = w(T)$, da T nach Annahme ein MSB ist). Also ist $\tilde{T}$ ein MSB von G, der e_i nicht enthält, im Widerspruch zur Annahme, e_i liege in jedem MSB von G.

□

Literatur: R.E. Tarjan: “Data Structures and Network Algorithms”, SIAM CBMS-NSF Reihe Bd. 44

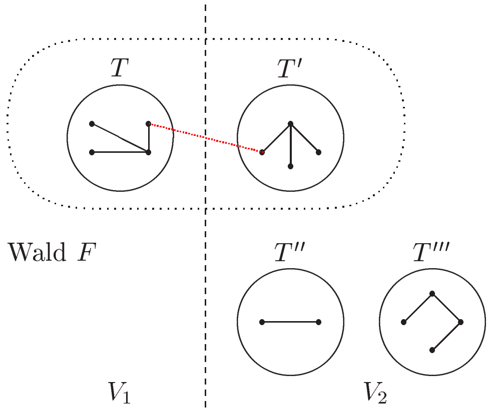
<u>Generischer minimaler Spannbaum-Algorithmus:</u>

Initialisiere Wald F von Bäumen, jedes $v \in V$ bildet einen Baum.

<u>while</u> F enthält mehr als einen Baum <u>do</u>

- wähle einen Baum $T \in F$ aus

- bestimme eine leichteste Kante $e = \{v, w\}$ aus T heraus
- sei $v \in T, w \in T'$
- vereinige T und T', füge e zum minimalen Spannbaum hinzu.

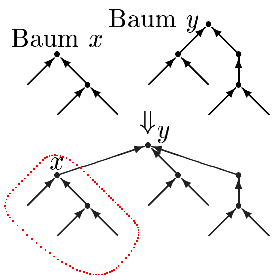


3.2 Kruskals Algorithmus

algorithm Kruskal (G,w):=

sortiere die Kanten nach aufsteigendem Gewicht in eine Liste L;
initialisiere $F = \{T_i, i = 1, \ldots, n\}$ Wald mit $T_i = \{v_i\}$;
  co $G = (V, E)$, $V = \{v_1, \ldots, v_n\}$ oc
co implementiere T_i's als in-Trees mit den jeweiligen Wurzeln als
  Repräsentanten; allgemein: UNION-FIND-Problem oc
MSB:=$\emptyset$;
for $i := 1$ to Länge(L) do
  $\{v, w\} := L_i$;
  $x :=$ Baum $\in F$, der v enthält; co $x :=$ FIND(v) oc
  $y :=$ Baum $\in F$, der w enthält; co $y :=$ FIND(w) oc
  if $x \neq y$ then
    MSB:=MSB $\cup$ $\{\{v, w\}\}$;
    if $|x| \leq |y|$ then
      co Baum x enthält $\leq$ Knoten als Baum y oc

      vereinige x und y, indem die Wurzel von x zu einem neuen Kind der Wurzel von y wird;



      co $y :=$ UNION(x, y) oc
    else
      vereinige x und y, mit y neues Kind der Wurzel von x;
    fi
  fi
od

Korrektheit: $\surd$

Zeitkomplexität: (mit $n = |V|, m = |E|$)

$$\mathcal{O}(m \log m + m \cdot \text{Kosten für } \underbrace{\text{FIND}}_{\mathcal{O}(\log n)} \text{ und } \underbrace{\text{UNION}}_{\mathcal{O}(1)}) = \mathcal{O}(m \log n)$$

(wegen $m \geq n - 1$)

Tiefe des in-Trees $\leq \log n$:

Sei $T'' = \text{UNION}(T, T')$, $|T| \leq |T'|$.

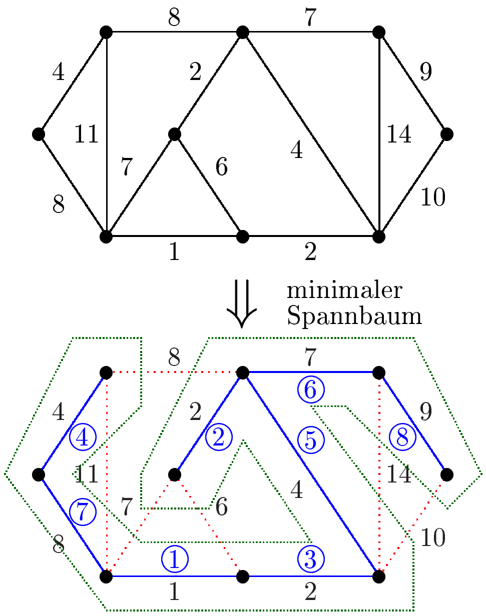
Beweis: durch Induktion

ok für $|T| = 1$

$$\begin{aligned}
\text{Tiefe}(T'') &\leq \max\{\text{Tiefe}(T'), \text{Tiefe}(T) + 1\} \\
&\leq \max\{\log |T'|, 1 + \log |T|\} \\
&\leq \log(|T| + |T'|) \\
&= \log |T''|.
\end{aligned}$$

$\square$

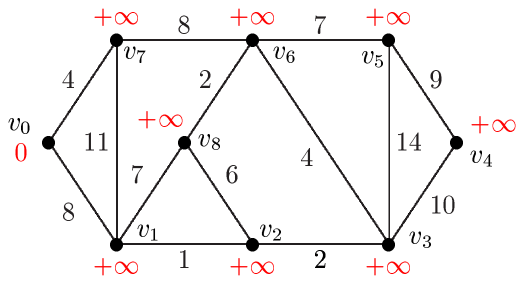
Beispiel für Kruskals Algorithmus:



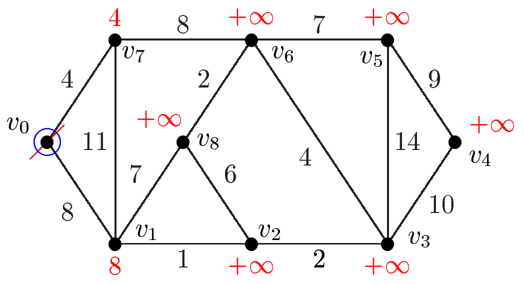
3.3 Prim's Algorithmus, erste Variante

```
algorithm PRIM-MSB (G,w) :=
⌜
    initialisiere Priority Queue PQ mit Knotenmenge V und Schlüssel +∞, ∀v ∈ V;
    wähle Knoten r als Wurzel (beliebig);
    Schlüssel[r] := 0;
    Vorgänger[r] := nil;
    while PQ ≠ ∅ do
        u := Find_Min(PQ);
        Delete_Min(PQ);
        for alle Knoten v, die in G zu u benachbart sind do
            if v ∈ PQ and w({u,v}) < Schlüssel[v] then
                Vorgänger[v] := u;
                Schlüssel[v] := w({u,v});
            fi
        od
    od
⌞
```

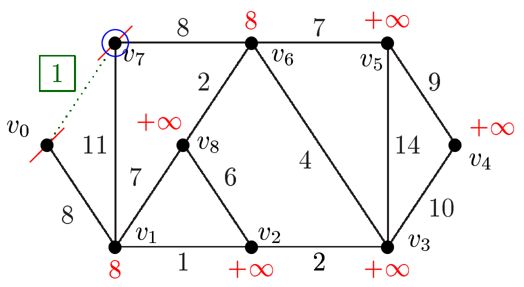
Beispiel für Prim's Algorithmus (erste Variante):



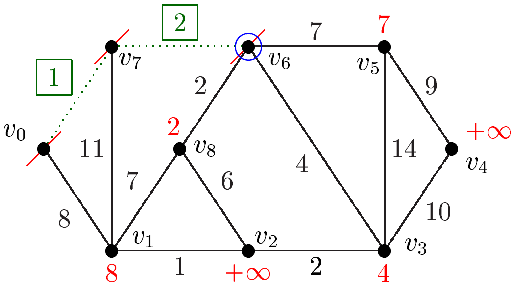
Ausgangszustand:
alle Schlüssel $= +\infty$
aktueller Knoten $(= u)$: ⦿
Startknoten: r $(= v_0)$



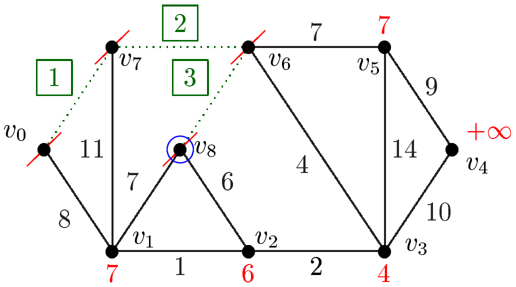
suche u := Find_Min(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn
in PQ mit
w($\{u, v\}$) < Schlüssel[v]:
$(v_1 = 8, v_7 = 4)$



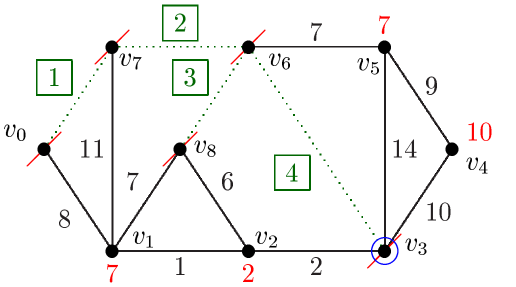
suche u := Find_Min(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn
in PQ mit
w($\{u, v\}$) < Schlüssel[v]:
$(v_6 = 8)$



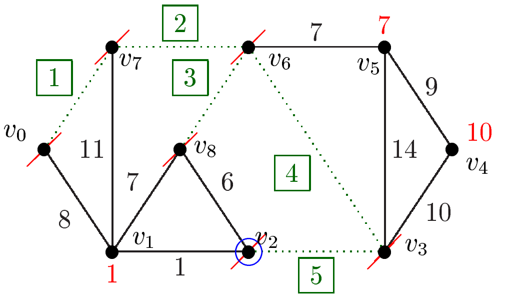
suche u := Find_Min(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit
w($\{u, v\}$) < Schlüssel[v]:
($v_3 = 4, v_5 = 7, v_8 = 2$)



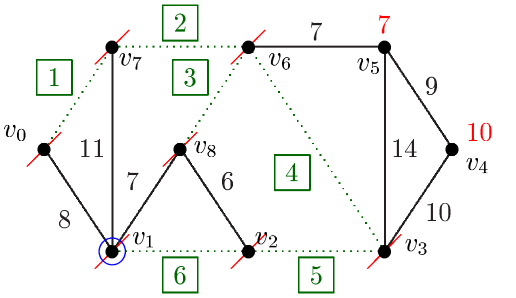
suche u := Find_Min(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit
w($\{u, v\}$) < Schlüssel[v]:
($v_1 = 7, v_2 = 6$)



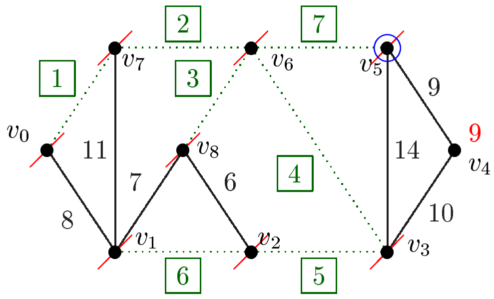
suche u := Find_Min(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit
w($\{u, v\}$) < Schlüssel[v]:
($v_2 = 2, v_4 = 10$)



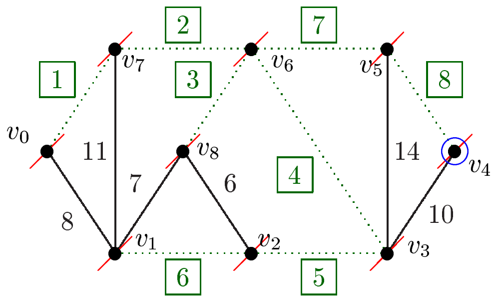
suche u := Find_Min(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit
w($\{u, v\}$) < Schlüssel[v]:
($v_1 = 1$)



suche u := Find_Min(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn in PQ mit
w($\{u, v\}$) < Schlüssel[v]:
solche Nachbarn existieren nicht



suche u := Find_Min(PQ)
und entferne u aus PQ
setze Schlüssel der Nachbarn
in PQ mit
w($\{u, v\}$) $<$ Schlüssel[v]:
($v_4 = 9$)



Endzustand:
suche u := Find_Min(PQ)
und entferne u aus PQ,
damit ist PQ leer und der
Algorithmus beendet

Implementierung der Priority Queue mittels Fibonacci-Heaps
Initialisierung: $\mathcal{O}(n)$
Find + Delete_Mins: $\mathcal{O}(n \log n)$ $\quad$ ($\leq n$ Stück)
Decrease_Keys: $\mathcal{O}(m)$ $\quad$ ($\leq m$ Stück)
$\rightsquigarrow$ Laufzeit: $\mathcal{O}(n) + \mathcal{O}(n \log(n)) + m \cdot \mathcal{O}(1) + m \cdot \underbrace{\mathcal{O}(1)}_{\text{sonstiger Overhead}} = \mathcal{O}(m + n \log\, n)$

Satz: Sei $G = (V, E)$ ein ungerichteter Graph (zusammenhängend, einfach) mit Kantengewichten w. Prim's Algorithmus berechnet, wenn mit Fibonacci-Heaps implementiert, einen minimalen Spannbaum von (G, w) in Zeit $\mathcal{O}(m+n \log n)$ (hier: $n = |V|$, $m = |E|$). Dies ist für $m = \Omega(n \log n)$ asymptotisch optimal.

3.4 Prim's Algorithmus, zweite Variante

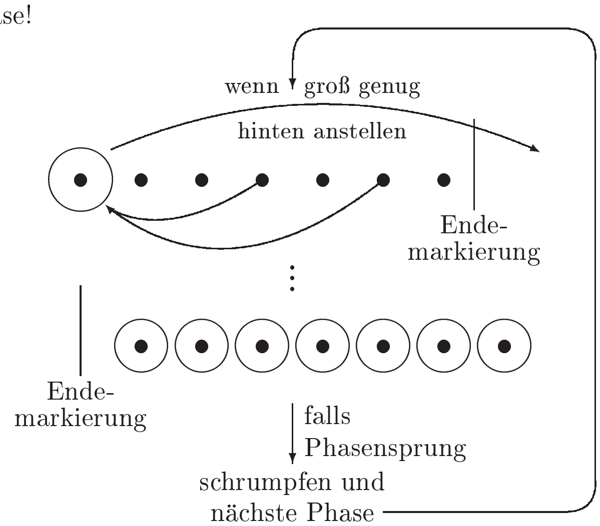
Idee: Lasse die Priority Queues nicht zu groß werden.
Sei $G = (V, E)$, $|V| = n$, $|E| = m$, w Gewichtsfunktion, k Parameter, der unten festgelegt wird.
Der Algorithmus wird in den folgenden Phasen abgearbeitet:

1. Initialisiere eine Schlange von Bäumen, jeder Baum anfangs (Super-) Knoten. Zu jedem Baum: Priority Queue (Fibonacci-Heap) mit den Nachbarn der Knoten im Baum, die selbst nicht im Baum sind. Schlüssel: Gewicht einer leichtesten Kante zu einem Knoten im Baum.

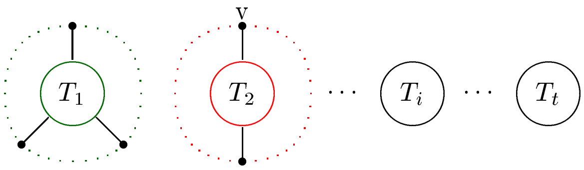
2. Markiere jeden Baum in der Schlange mit der Nummer der laufenden Phase

3. <u>while</u> vorderster Baum in der Schlange hat laufende Phasennummer <u>do</u>
   lasse ihn wachsen, solange seine Priority Queue weniger als
   k Elemente enthält (und noch was zu tun ist)
   <u>if</u> Priority Queue zu groß <u>then</u>
   füge Baum mit Phasennummer + 1 am Ende der Schlange ein
   <u>fi</u>
   <u>od</u>

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



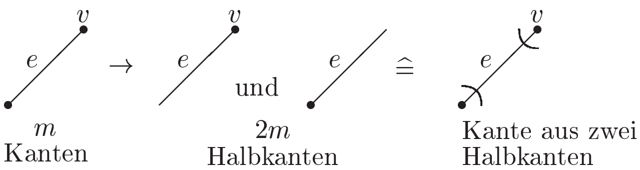
Betrachte Schlange von Bäumen:



(Arbeit pro Phase $\sim$ # Knoten, außer Vereinigung zweier Superknoten 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, Einfügen bzw. conditional Decrease_Key. 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 den Bäumen vor. Mit m Kanten ergeben sich $2m$ Halbkanten, d.h. Anzahl proportional zu m.



Zeitaufwand pro Phase: (+ Superknoten zu Beginn)

- Initialisierung: $\mathcal{O}(m)$
- Delete und Delete_Min: $\leq t$
- sonstige Priority Queue-Operationen, Overhead: $\mathcal{O}(m)$

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

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

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

Wieviele Phasen? Sei t die Anzahl der Superknoten am Anfang einer Phase, t' diese Zahl zu Beginn der nächsten Phase. Sei a die durchschnittliche Anzahl ursprünglicher Knoten in den 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 $a = \frac{2m}{n}$, a ist für jede Phase $\leq n-1$. 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

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

(Bemerkung: $\log^* n = \min\{i; log_2^{(i)} \leq 1\}$, z.B. $\log^* 65536 = \log^* 2^{2^{2^2}} = 4$)

Satz: Für gewichtete, zusammenhängende 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.

Euklidische minimale Spannbäume stellen ein Problem dar, für das es speziellere Algorithmen gibt. Literatur hierzu:
A.C.-C. Yao: Constructing Minimum Spanning Trees in k-dimensional spaces and related problems. SIAM Journal on Computing <u>11</u>(4) p. 721–736 (1982)