Effiziente Algorithmen und Datenstrukturen I

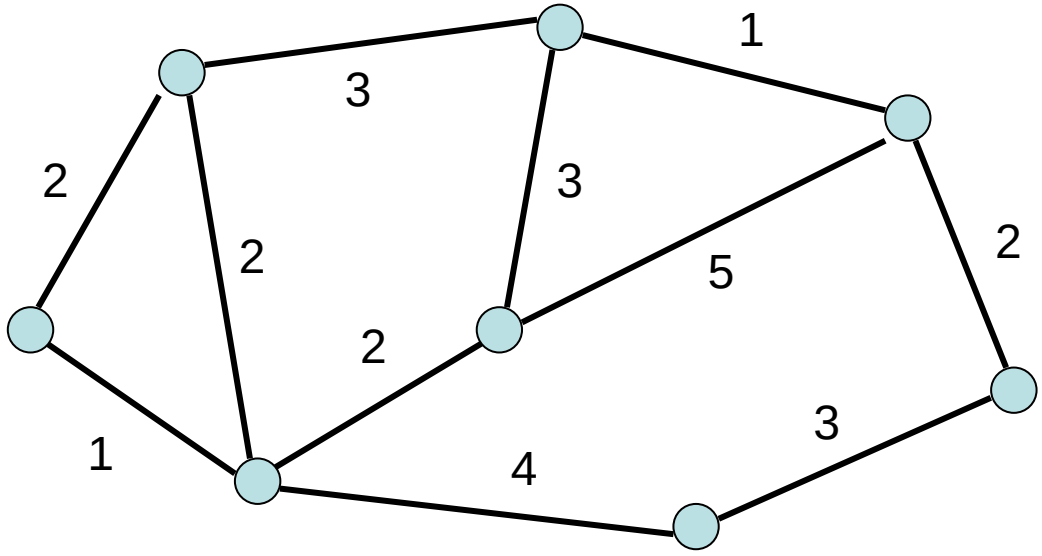
Kapitel 9: Minimale Spannbäume

Christian Scheideler

WS 2008

Minimaler Spannbaum

Zentrale Frage: Welche Kanten muss ich nehmen, um mit minimalen Kosten alle Knoten zu verbinden?



Minimaler Spannbaum

Eingabe:

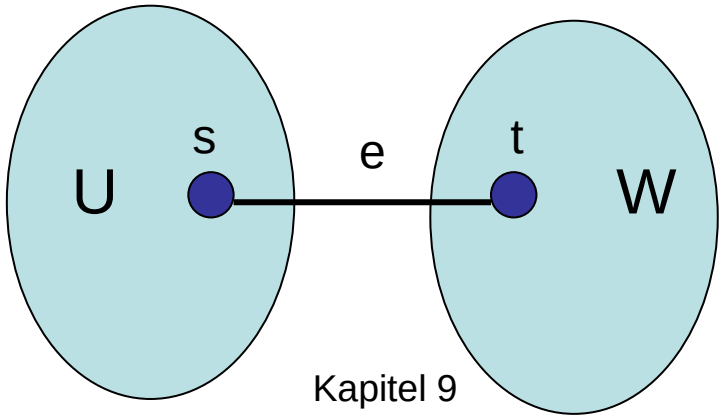
- ungerichteter Graph $G=(V,E)$
- Kantenkosten $c:E \rightarrow \mathbb{R}_+$

Ausgabe:

- Teilmenge $T \subset E$, so dass Graph (V,T) verbunden und $c(T)=\sum_{e \in T} c(e)$ minimal
- T formt immer einen Baum (wenn c positiv).
- Baum über alle Knoten in V mit minimalen Kosten: minimaler Spannbaum (MSB)

Minimaler Spannbaum

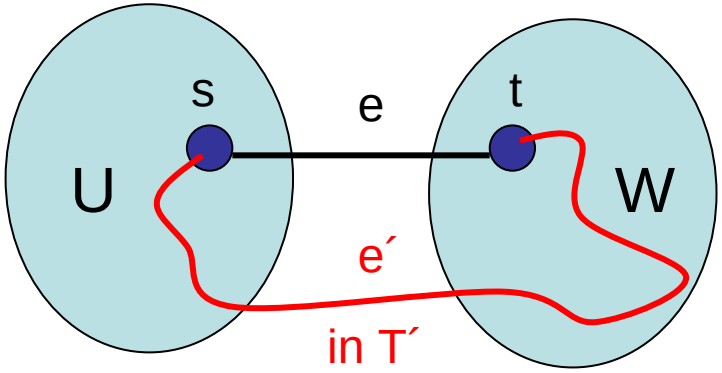
Lemma 9.1: Sei (U,W) eine Partition von V (d.h. $U \cup W = V$ und $U \cap W = \emptyset$) und $e=\{s,t\}$ eine Kante mit minimalen Kosten mit $s \in U$ und $t \in W$. Dann gibt es einen minimalen Spannbaum (MSB) T, der e enthält.



Minimaler Spannbaum

Beweis von Lemma 9.1:

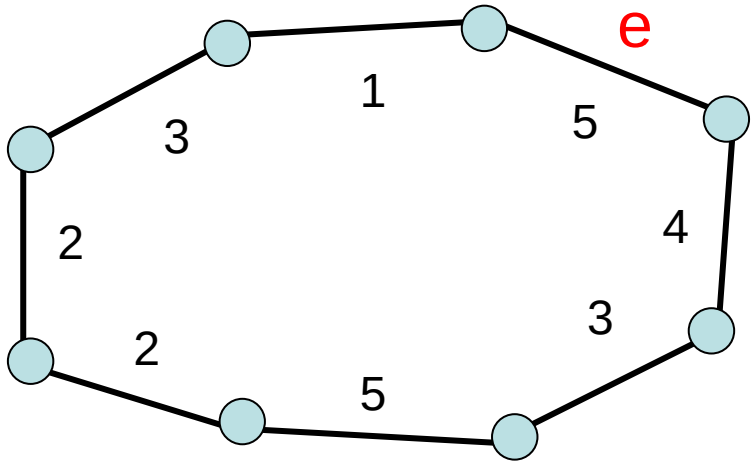
- Betrachte beliebigen MSB T´
- e={s,t}: (U,W)-Kante minimaler Kosten



- Ersetzung von e´ durch e führt zu Baum T´´, der höchstens Kosten von MSB T´ hat, also MSB ist

Minimaler Spannbaum

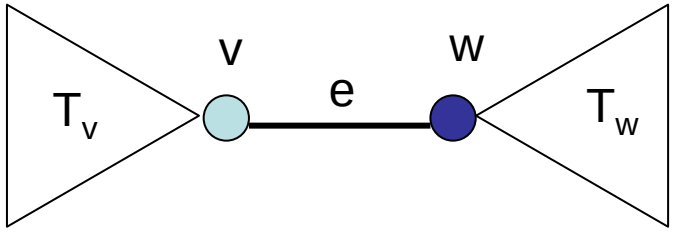
Lemma 9.2: Betrachte beliebigen Kreis C in G und sei e Kante in C mit maximalen Kosten. Dann ist jeder MSB in G ohne e auch ein MSB in G.



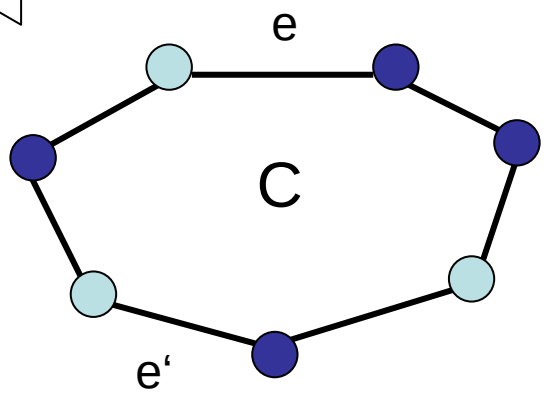
Minimaler Spannbaum

Beweis von Lemma 9.2:

- Betrachte beliebigen MSB T in G
- Angenommen, T enthalte e



e maximal für C

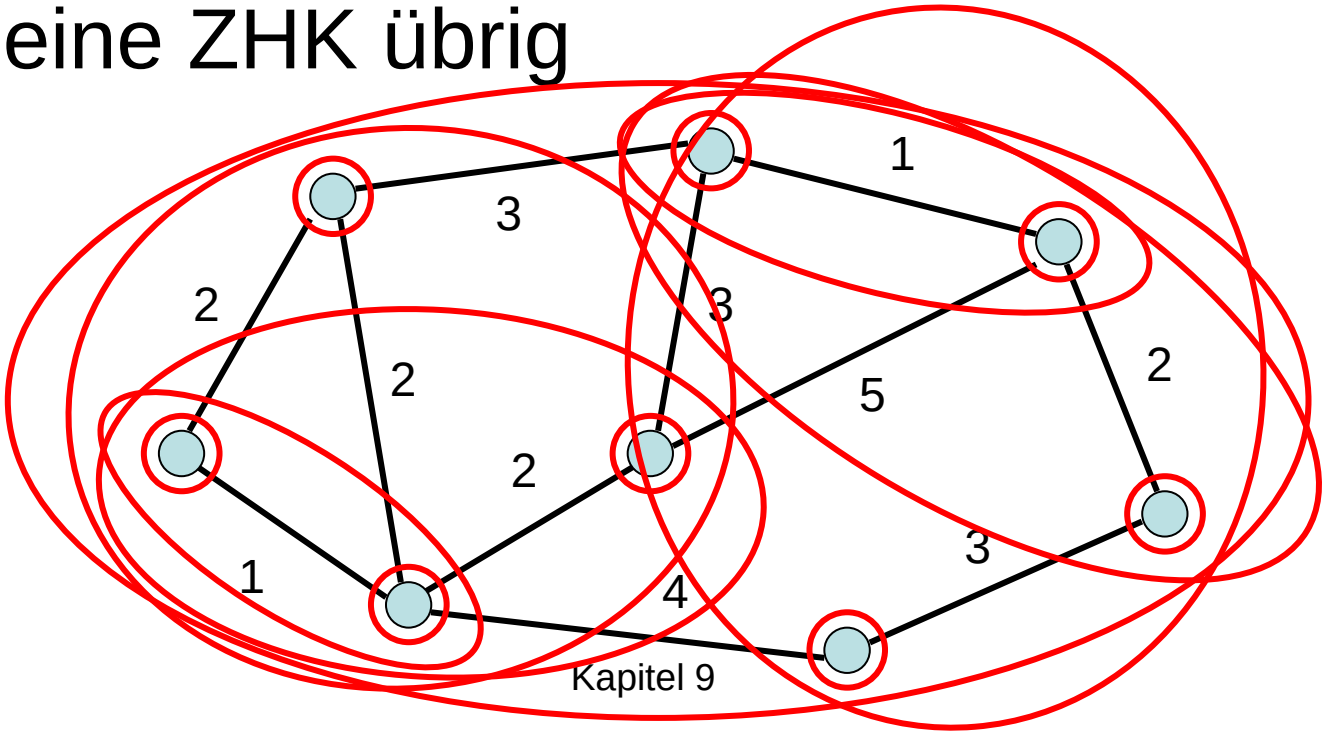


- ○: zu T_v, ●: zu T_w
  - es gibt e´ von T_v nach T_w
  - $e \rightarrow e´$ ergibt MSB T´ ohne e

Minimaler Spannbaum

Regel aus Lemma 9.1:

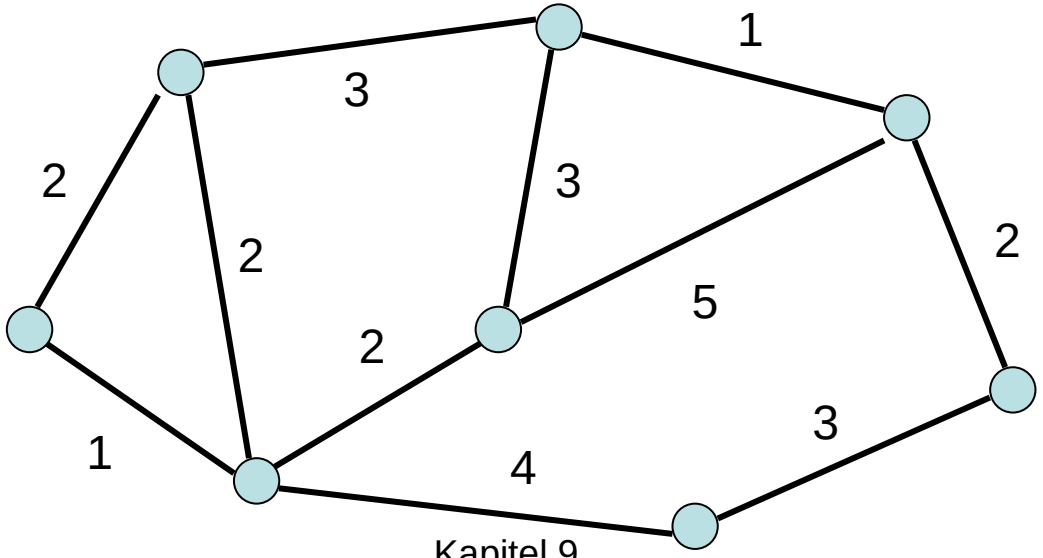
Wähle wiederholt Kante mit minimalen Kosten, die verschiedene ZHKs verbindet, bis eine ZHK übrig



Minimaler Spannbaum

Regel aus Lemma 9.2:

Lösche wiederholt Kante mit maximalen Kosten, die Zusammenhang nicht gefährdet, bis ein Baum übrig



Minimaler Spannbaum

Problem: Wie implementiert man die Regeln effizient?

Strategie aus Lemma 9.1:

- Setze $T=\emptyset$ und sortiere die Kanten aufsteigend nach ihren Kosten
- Für jede Kante (u,v) in der sortierten Liste, teste, ob u und v bereits im selben Baum in T sind. Falls nicht, füge (u,v) zu T hinzu.

benötigt Union-Find DS

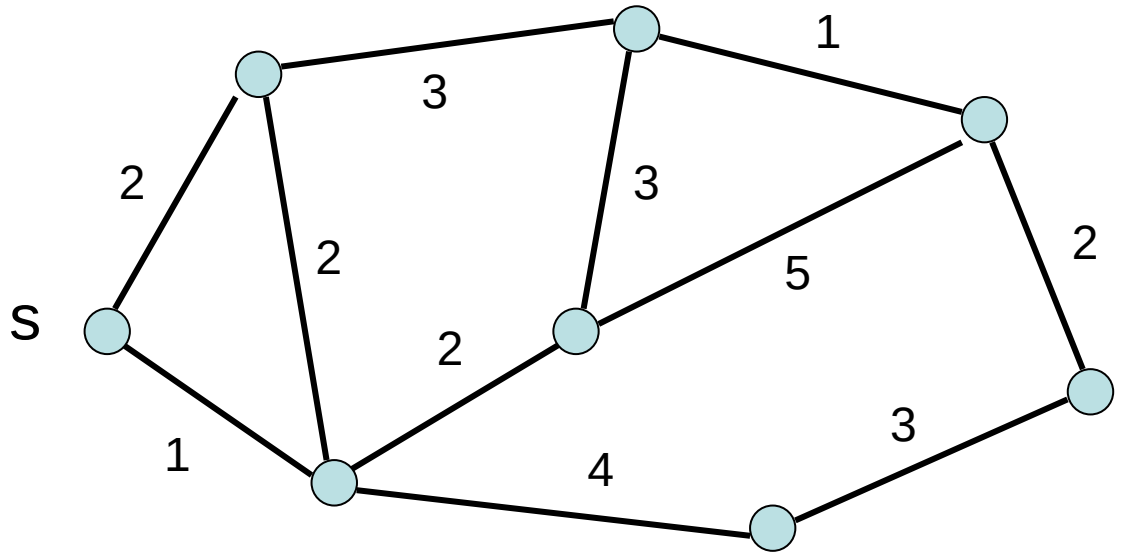
Erinnerung: Union-Find DS

Operationen:

- Union(x_1, x_2): vereinigt die Elemente in den Teilmengen T_1 und T_2, zu denen die Elemente x_1 und x_2 gehören, zu $T=T_1 \cup T_2$
- Find(x): gibt (eindeutigen) Repräsentanten der Teilmenge aus, zu der Element x gehört

Minimaler Spannbaum

Beispiel: (—: Kanten im MSB)



Kruskal Algorithmus

```
Function KruskalMST(V,E,c): Set of Edge
   T:=∅
   Init(V) // initialisiere einelem. Mengen für V
   S:=Mergesort(E)  // aufsteigend sortiert
   foreach {u,v}∈ S do
      if Find(u)≠Find(v) then // versch. Mengen
         T:=T ∪ { {u,v} }
         Union(u, v)  // u und v in einer Menge
   return T
```

Kruskal Algorithmus

Laufzeit:

- Mergesort: $O(m \log n)$ Zeit
- $2m$ Find-Operationen und $n-1$ Union-Operationen: $O(\alpha(m,n) \cdot m)$ Zeit

Insgesamt Zeit $O(m \log n)$.

Mit LinearSort auf $O(\alpha(m,n) \cdot m)$ reduzierbar.

Minimaler Spannbaum

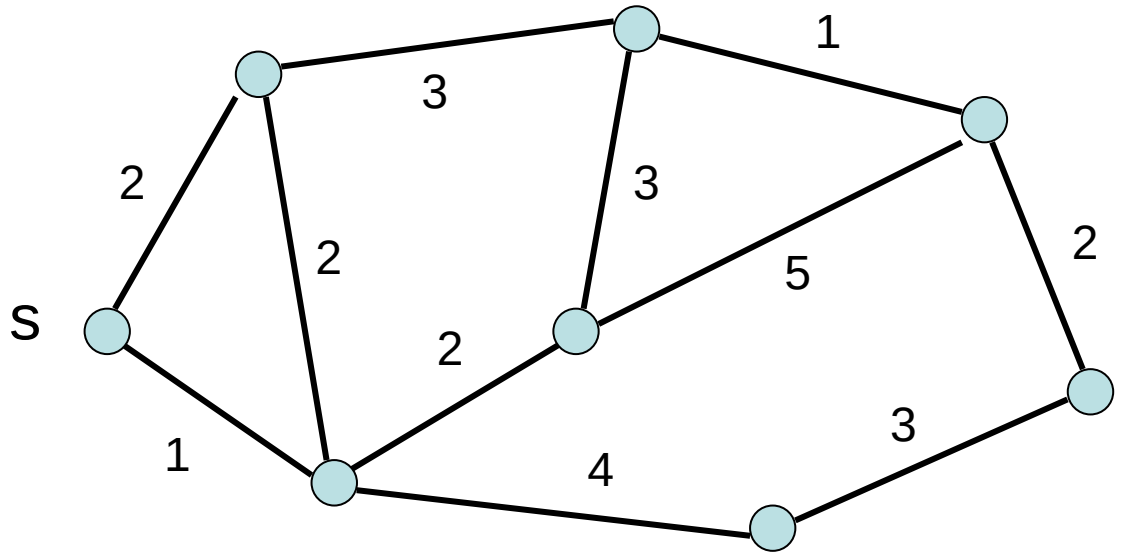
Problem: Wie implementiert man die Regeln effizient?

Alternative Strategie aus Lemma 9.1:

- Starte bei beliebigem Knoten s, MSB T besteht anfangs nur aus s
- Ergänze T durch günstigste Kante zu äußerem Knoten w und füge w zu T hinzu bis T alle Knoten im Graphen umfasst

Minimaler Spannbaum

Beispiel:



Jarnik-Prim Algorithmus

```
Procedure JarnikPrim(s: NodeId)
    d=<∞,…,∞>: NodeArray of IR ∪ {-∞,∞}
    parent=<⊥,…,⊥>: NodeArray of NodeId
    d[s]:=0; parent[s]:=s // T anfangs nur aus s
    q=<s>: NodePQ
    while q ≠ ∅ do
        u:=q.deleteMin() // u: min. Distanz zu T in q
        foreach e={u,v} ∈ E mit v∉MSB(s) do
            if c(e) < d[v] then // aktualisiere d[v] zu T
                if d[v]=∞ then q.insert(v) // v schon in q?
                d[v]:=c(e); parent[v]:=u
                q.decreaseKey(v)
```

Jarnik-Prim Algorithmus

Laufzeit:

$T_{JP} = O(n(T_{DeleteMin}(n)+T_{Insert}(n)) + m \cdot T_{decreaseKey}(n))$

Binärer Heap: alle Operationen $O(\log n)$, also

$T_{JP} = O((m+n)\log n)$

Fibonacci Heap:

- $T_{DeleteMin}(n)=T_{Insert}(n)=O(\log n)$
- $T_{decreaseKey}(n)=O(1)$
- Damit $T_{JP} = O(n \log n + m)$

Verbesserter JP-Algorithmus

Sei $G=(V,E)$ ein ungerichteter Graph mit $|V|=n$, $|E|=m$ und $m=o(n \log n)$ (sonst verwende ursprünglichen JP Algorithmus)

Idee: lasse Priority Queues nicht zu groß werden.

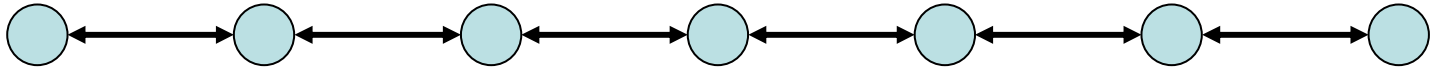
Der verbesserte JP-Algorithmus arbeitet dazu in Phasen, wobei in Phase i ein Parameter k_i verwendet wird, dessen Wert wir erst später festlegen werden.

Verbesserter JP-Algorithmus

Phase 1 startet mit Graph $G_1=(V_1,E_1)=G$. Die Funktion $g:E_1 \cup V_1 \rightarrow IR_+$ weist jeder Kante Gewicht 1 zu und jedem Knoten in G_1 ein Gewicht zu, das seinem Grad entspricht.

Phase $i \geqslant 1$ arbeitet wie folgt:

- Schritt 1: Forme eine Liste L aller Knoten in G_i.

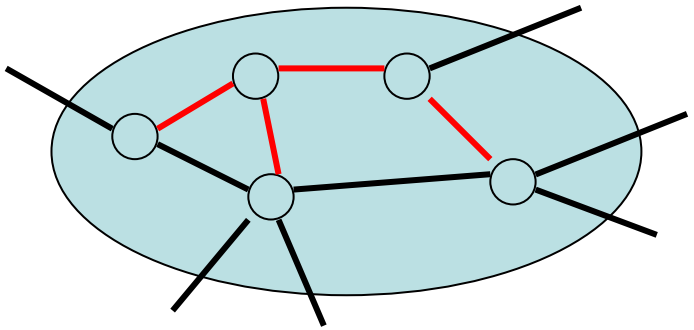


Verbesserter JP-Algorithmus

Hintergrund zu Schritt 2:

- Jeder (Super-)Knoten repräsentiert einen MSB im ursprünglichen Graphen G.
- Das Gewicht eines (Super-)Knotens ist die Summe der Gewichte der Knoten in G, die in seinem MSB enthalten sind.

Superknoten v



— : MSB-Kante

Verbesserter JP-Algorithmus

- Schritt 2: solange L nicht leer ist:
  - Nimm den vordersten Knoten j in L und führe den JP-Algo auf j aus, um einen minimalen Spannbaum MSB_j mit Wurzel j aufzubauen (wobei Knoten aus L, die zum MSB_j hinzugefügt werden, aus L gelöscht werden), bis eines der folgenden Ereignisse eintritt:
    - PQ $Q_j=\emptyset$ oder $\sum_{v \in MSB_j} g(v) \geqslant k_i$. Dann entferne j aus L.
    - Ein Knoten v wird zu MSB_j hinzugefügt wird, der bereits zu einem MSB mit Wurzel j' gehört. Dann vereinen wir die MSBs und die PQs von j und j' zu einem MSB mit Wurzel j' mittels einer speziellen Union-Operation und entfernen j aus L.

Verbesserter JP-Algorithmus

Union-Operation:

- T_1, Q_1: MSB und PQ von Knoten j',
  T_2, Q_2: MSB und PQ von Knoten j,
  Vereinigung aufgrund Kante $\{v,w\}$ mit $v \in T_1$, $w \in T_2$.
  Ziel: $T_1 := T_1 \cup T_2 \cup \{v,w\}$, $Q_1 := PQ(Q_1,Q_2)$
- Für alle Knoten $u \in T_2$ mit $u \in Q_1$:
  lösche u aus Q_1 (einfaches Delete)
- Für alle Knoten $u \in Q_2$:
  - $u \in T_1$: u nicht in Q_1 einfügen (einfach löschen)
  - $u \in Q_1$: update Wert von u in Q_1 (decreaseKey)
  - Sonst füge u in Q_1 ein (Insert).

Verbesserter JP-Algorithmus

- Schritt 3: (mindestens eine nichtleere PQ übrig)
  - Schrumpfe alle MSBs zu Superknoten (dessen Gewicht die Summe der Gewichte all seiner Knoten in G ist, die er repräsentiert).
  - Verwende die PQs zu den MSBs, um Superknoten miteinander zu verbinden. (D.h. für jeden MSB T_1 mit Kanten zu einem MSB T_2 erhalten die entsprechenden Superknoten eine Kante e, dessen Wert den minimalen Kosten einer Kante von T_1 nach T_2 entspricht und dessen Gewicht $g(e)$ die Summe der Gewichte der Kanten von T_1 nach T_2 ist.) Dafür reicht ein linearer Scan der entsprechenden PQs.
  - Daraus resultierender Graph ist G_{i+1}.

Verbesserter JP-Algorithmus

- Schritt 3: (alle PQs leer)
  Falls nur eine PQ übrig, dann gibt es nur noch einen MSB mit einer Wurzel. Damit haben wir einen MSB über alle Superkonten in G_i (und damit über alle Knoten in G).
  Falls mehrere PQs übrig, dann ist G nicht verbunden. Fehler!

Verbesserter JP-Algorithmus

Korrektheit:

- Die MSB-Bildung ist korrekt, da JP-Algo verwendet wird, um MSBs aufzubauen, welcher Lemma 9.1 erfüllt.

Verbesserter JP-Algorithmus

Laufzeitanalyse:

- Schritt 1: $O(n_i)$ Zeit, wobei $n_i=|V_i|$ ist.
- Schritt 2: (Übung)
  - $O(n_i)$ deleteMin Operationen
  - $O(m)$ Delete Operationen
  - $O(m)$ Insert Operationen
  - $O(m)$ decreaseKey Operationen

  Mit Fibonacci-Heaps Laufzeit $O(n_i \log k_i + m)$

Verbesserter JP-Algorithmus

- Schritt 3 (mind. eine nichtleere PQ): Laufzeit $O(n_i+m)$ zur Bildung von G_{i+1}
- Schritt 3 (alle PQs leer): fertig

Also pro Phase Laufzeit $O(n_i \log k_i + m)$.

Noch offen:

- Definition von k_i (damit Schranke für n_i).
- Maximale Anzahl Phasen.

Verbesserter JP-Algorithmus

Setze $k_i = 2^{2m/n_i}$.

$\Rightarrow n_i \log k_i = 2m$,

$\Rightarrow$ Kosten pro Phase $O(m)$.

Anzahl Phasen:

- n: Anzahl Superknoten zu Beginn einer Phase
- $2m$: Summe aller Knotengewichte zu Beginn der Phase
- d: durschnittliches Gewicht eines Superknotens zu Beginn der Phase
- d': durchschnittliches Gewicht eines Superknotens am Ende der Phase

Verbesserter JP-Algorithmus

- Es gilt $d' \geqslant k = 2^{2m/n} = 2^d$.
- Für die erste Phase ist $d=2m/n$ und für alle anderen Phasen ist $d \leqslant 2m$.
- Also ist die Anzahl der Phasen höchstens

$$1+ \min\{i: \log^{(i)} 2m \leqslant 2m/n\}$$

Setzen wir $\beta(m,n):=\min\{i: \log^{(i)} 2m \leqslant 2m/n\}$, dann gilt $\beta(m,n) \leqslant \log^* m$ für alle $m \geqslant n$.

Verbesserter JP-Algorithmus

Theorem 9.3: Für beliebige gewichtete ungerichtete Graphen mit n Knoten und m Kanten kann ein MSB in Zeit

$$O(\min\{m \cdot \beta(m,n), m + n \log n\})$$

bestimmt werden.

Ausblick

Weiter mit linearer Algebra…