Grundlagen der Algorithmen und Datenstrukturen

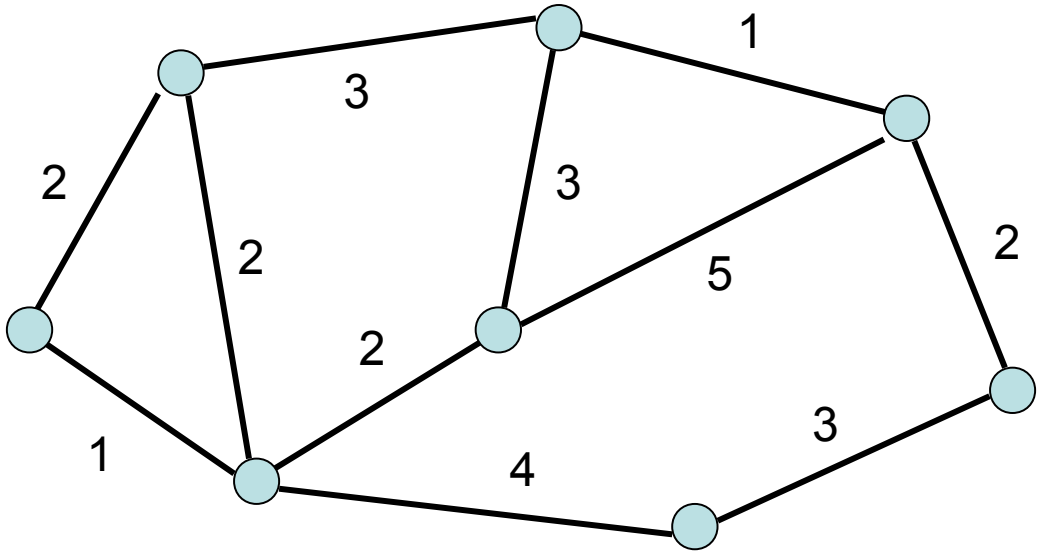
Kapitel 11

Christian Scheideler + Helmut Seidl

SS 2009

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 $c:E \rightarrow \mathbb{R}_+$

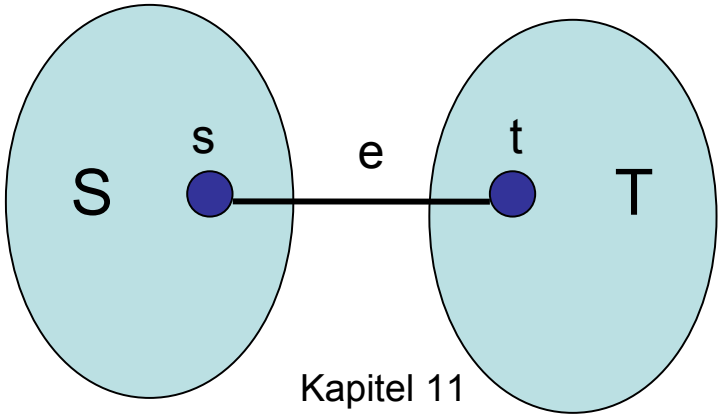
Ausgabe:

- Teilmenge $T \subseteq E$, 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

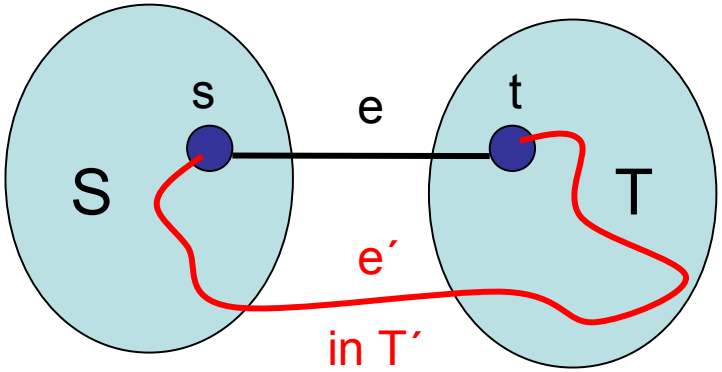
Lemma 11.1: Sei (S,T) eine Partition von V (d.h. $S \cup T = V$ und $S \cap T = \emptyset$) und $e=\{s,t\}$ eine Kante mit minimalen Kosten mit $s \in S$ und $t \in T$. Dann gibt es einen minimalen Spannbaum (MSB) T, der e enthält.



Minimaler Spannbaum

Beweis von Lemma 11.1:

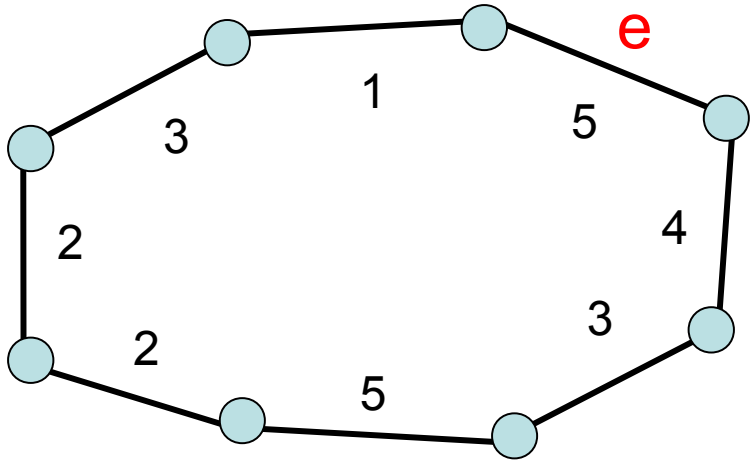
- Betrachte beliebigen MSB T´
- e={s,t}: (S,T)-Kante minimaler Kosten



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

Minimaler Spannbaum

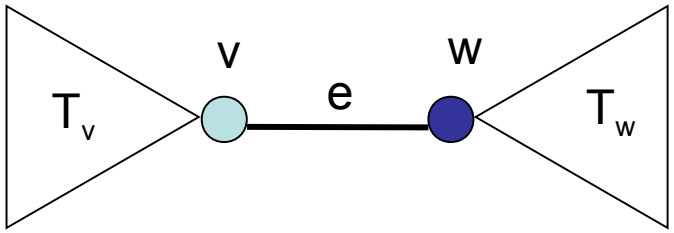
Lemma 11.2: Betrachte beliebigen Kreis C in G und sei e Kante in C mit maximalen Kosten. Dann ist jeder MSB in G ohne e auch ein MSB in G.



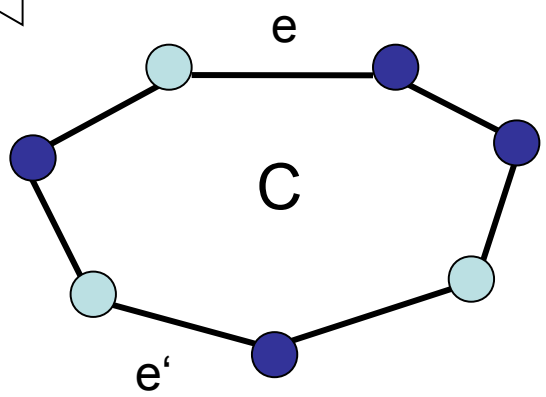
Minimaler Spannbaum

Beweis von Lemma 11.2:

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



e maximal für C

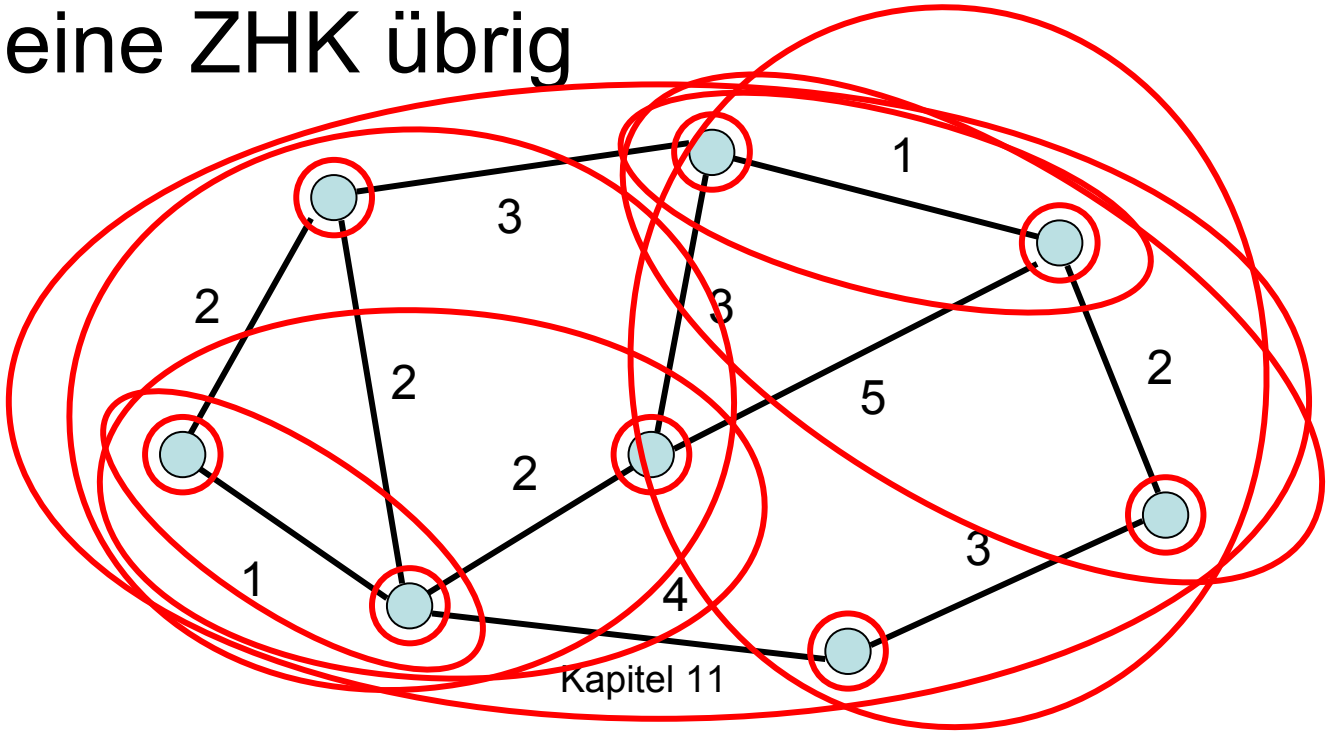


- ○: zu T_v, ●: zu T_w
  - es gibt e´ von T_v nach T_w
  - e → e´ ergibt MSB T´ ohne e

Minimaler Spannbaum

Regel aus Lemma 11.1:

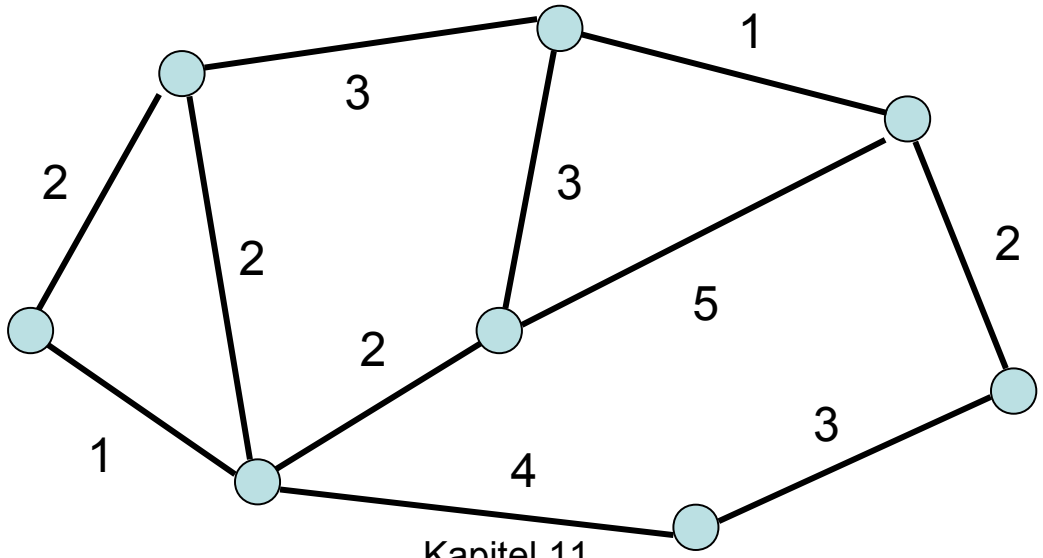
Wähle wiederholt Kante mit minimalen Kosten, die verschiedene ZHKs verbindet, bis eine ZHK übrig



Minimaler Spannbaum

Regel aus Lemma 11.2:

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



Minimaler Spannbaum

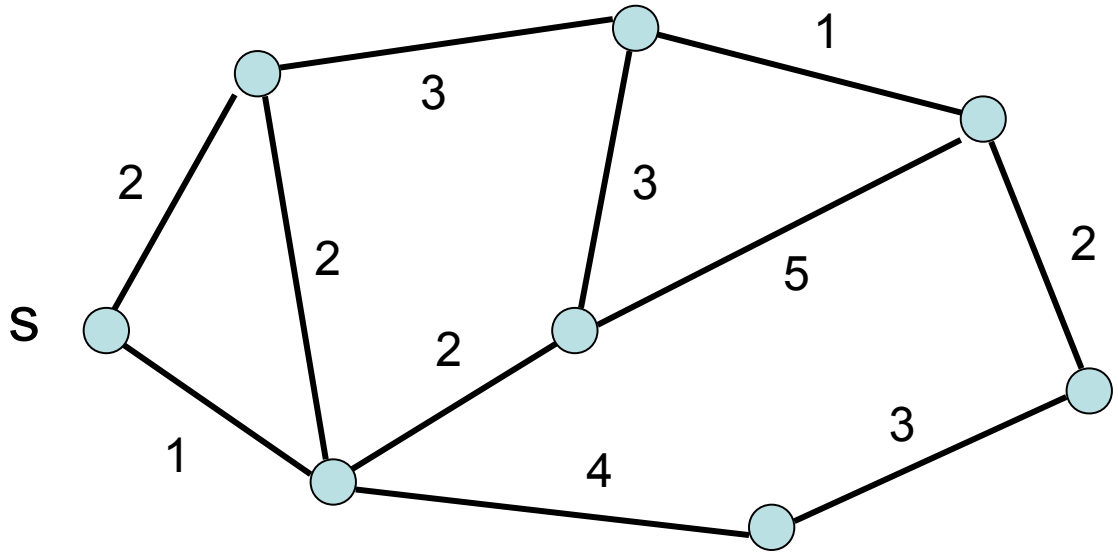
Problem: Wie implementiert man die Regeln effizient?

Strategie aus Lemma 11.1:

- 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.

Minimaler Spannbaum

Beispiel: (—: Kanten im MSB)



Kruskal Algorithmus

```
Set<Edge> KruskalMST(V,E,c) {
   T = ∅;
   S = sort(E);  // aufsteigend sortiert
   foreach ((u,v) ∈ S)
      if (u, v in verschiedenen Bäumen in T)
         T = T ∪ {(u,v)};
   return T;
}
```

benötigt effiziente Datenstruktur

Union-Find Datenstruktur

- Knoten seien nummeriert von 0 bis n-1
- Feld int parent[n]
- Anfangs: parent[i] = i; für alle i

```
int find(int I) {
   if (parent[i] == i) return i; // i Wurzel des Baums?
   else {  // nein:
      i' = find(parent[i]);  // suche Wurzel des Baums
      parent[i] = i';        // lass i direkt darauf zeigen
      return i';             // gib Wurzel zurück
}}
```

Union-Find Datenstruktur

- Knoten seien nummeriert von 0 bis n-1
- Feld int parent[n]
- Anfangs: parent[i]=i; für alle i

```
void union(int i, int j) {
   int k = find(i); int l = find(j);  // suche Wurzeln
   // falls verschieden, dann zusammen
   if (k!=l) parent[k] = l;
}
```

Kruskal Algorithmus

```
Set<Edge> KruskalMST(Nodes V,Edges E) {
   T = ∅; S = sort(E);  // aufsteigend sortiert
   for (int i=1; i<|V|; i++) parent[i] = i;
   foreach ((u,v)∈ S)
      if (find(u) != find(v)) { // nicht verbunden
         T = T ∪ {(u,v)};
         union(u,v);  // u und v in einen Baum
   }
   return T;
}
```

Union-Find Datenstruktur

- Die Kosten von find() hängen von der Tiefe der Datenstruktur ab. Wiederholtes Finden wird billiger!
- Um Kosten weiter zu verringern, hängen wir stets den kleineren Baum unter den größeren:

```
void union(int i, int j) {
   int k = find(i); int l = find(j);  // suche Wurzeln
   // falls verschieden, dann zusammen
   if (k!=l) if (size[k] <= size[l]) { parent[k] = l;
                                       size[l] = size[k]+size[l];
            } else { parent[l] = k; size[k] = size[k]+size[l];}
}
```

Union-Find Datenstruktur

- Die Kosten von find wachsen nun amortisiert proportional wie log* n !!!
- Die Kosten von union haben sich dabei nur unwesentlich erhöht.

Insgesamt ergeben sich dann Kosten:

$$O(m \log n + m \log^* n).$$

Minimaler Spannbaum

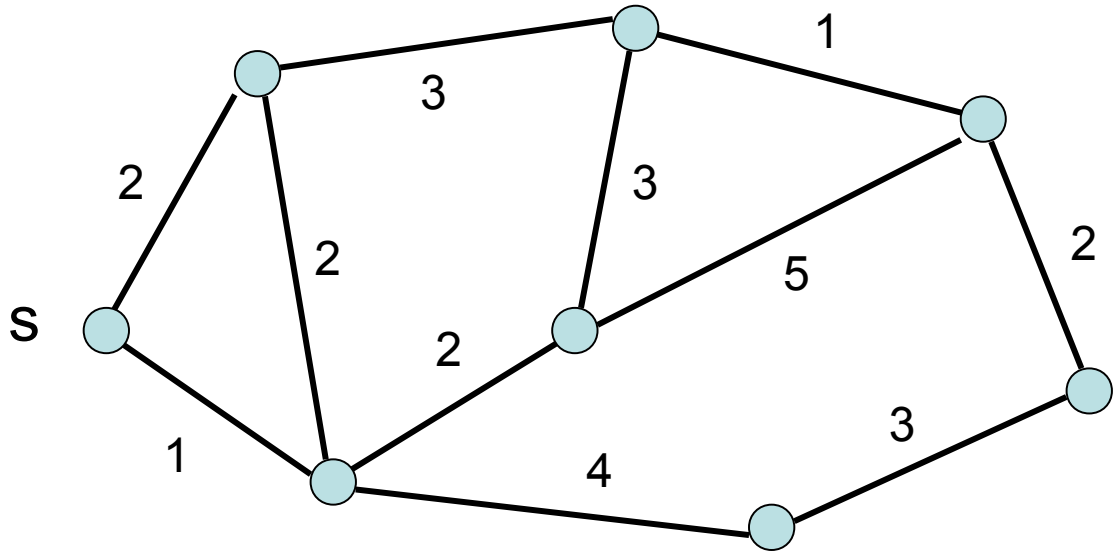
Problem: Wie implementiert man die Regeln effizient?

Alternative Strategie aus Lemma 11.1:

- 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

```
void JarnikPrim(Node s) {
   double [] d={∞,…,∞};
   Node [] parent = new Node [n];
   d[s] =0.; parent[s] = s; // T anfangs nur aus s
   PQ<Node> q = <s>;
   while (q !=<>) {
      u = q.deleteMin();  // u: min. Distanz zu T in q
      foreach (e=(u,v) ∈ E)
         if (c(e) < d[v]) { // aktualisiere d[v] zu T
            if (d[v] == ∞) q.insert(v); // v schon in q?
            If (v ∈ 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 (nicht in Vorlesung behandelt):

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