Script generated by TTT

Title: TÃ¤ubig: GAD (16.07.2013)

Date: Tue Jul 16 14:16:50 CEST 2013

Duration: 77:55 min

Pages: 46

Johnson-Algorithmus für APSP

- füge neuen Knoten s und Kanten (s, v) für alle v hinzu mit $c(s, v) = 0$

$\Rightarrow$ alle Knoten erreichbar

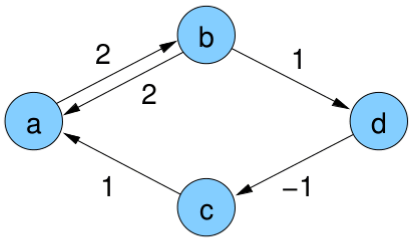
- berechne $d(s, v)$ mit Bellman-Ford-Algorithmus
- setze $\Phi(v) = d(s, v)$ für alle v
- berechne modifizierte Kosten $\bar{c}(e)$

$\Rightarrow$ $\bar{c}(e) \geq 0$, kürzeste Wege sind noch die gleichen

- berechne für alle Knoten v die Distanzen $\bar{d}(v, w)$ mittels Dijkstra-Algorithmus mit modifizierten Kantenkosten auf dem Graph ohne Knoten s
- berechne korrekte Distanzen $d(v, w) = \bar{d}(v, w) + \Phi(w) - \Phi(v)$

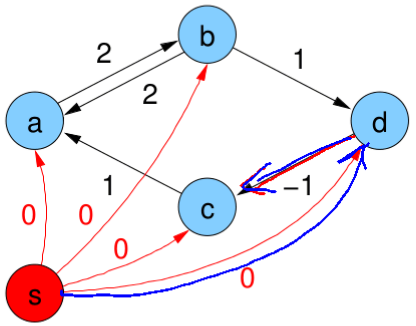
Johnson-Algorithmus für APSP

Beispiel:



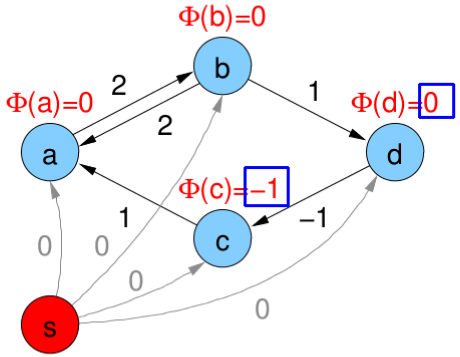
Johnson-Algorithmus für APSP

1. künstlicher Startknoten s:



Johnson-Algorithmus für APSP

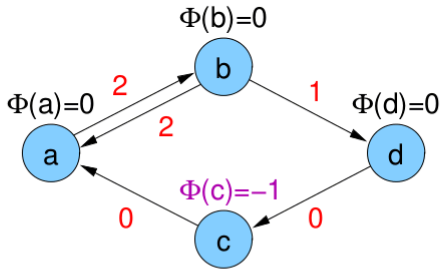
2. Bellman-Ford-Algorithmus auf s:

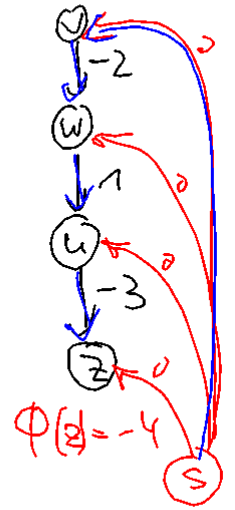


Johnson-Algorithmus für APSP

3. $\bar{c}(e)$-Werte für alle $e = (v, w)$ berechnen:

$$\bar{c}(e) = \Phi(v) + c(e) - \Phi(w)$$

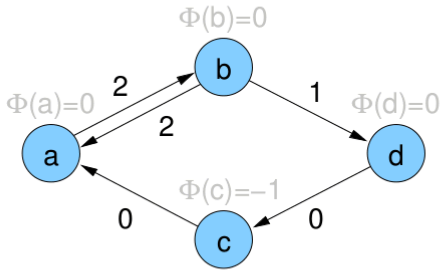




Johnson-Algorithmus für APSP

4. Distanzen $\bar{d}$ mit modifizierten Kantengewichten via Dijkstra:

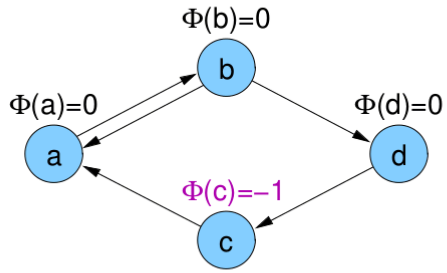
d	*a*	*b*	*c*	*d*
a	0	2	3	3
b	1	0	1	1
c	0	2	0	3
d	0	2	0	0



Johnson-Algorithmus für APSP

5. korrekte Distanzen berechnen: $d(v, w) = \bar{d}(v, w) + \Phi(w) - \Phi(v)$

d	*a*	*b*	*c*	*d*
a	0	2	2	3
b	1	0	0	1
c	1	3	0	4
d	0	2	−1	0



Johnson-Algorithmus für APSP

$(n+1)(m+n) = n \cdot m + n + n^2 \in O(n^2)$

Laufzeit:

$$\begin{aligned} T_{\text{Johnson}}(n,m) &= O(T_{\text{Bellman-Ford}}(n+1, m+n) + n \cdot T_{\text{Dijkstra}}(n,m)) \\ &= O((m+n)\cdot(n+1) + n \cdot (n \log n + m)) \\ &= O(m \cdot n + n^2 \log n) \end{aligned}$$

(bei Verwendung von Fibonacci Heaps)

Floyd-Warshall-Algorithmus für APSP

Grundlage:

- geht der kürzeste Weg von u nach w über v, dann sind auch die beiden Teile von u nach v und von v nach w kürzeste Pfade zwischen diesen Knoten
- Annahme: alle kürzesten Wege bekannt, die nur über Zwischenknoten mit Index kleiner als k gehen

$\Rightarrow$ kürzeste Wege über Zwischenknoten mit Indizes bis einschließlich k können leicht berechnet werden:

- entweder der schon bekannte Weg über Knoten mit Indizes kleiner als k
- oder über den Knoten mit Index k (hier im Algorithmus der Knoten v)

Floyd-Warshall-Algorithmus für APSP

Floyd-Warshall APSP Algorithmus

Input : Graph $G = (V, E)$, $c : E \mapsto R$
Output : Distanzen $d(u, v)$ zwischen allen $u, v \in V$

```
for u, v ∈ V do
    d(u, v) = ∞;   pred(u, v) = ⊥;
for v ∈ V do  d(v, v) = 0;
for (u, v) ∈ E do
    d(u, v) = c(u, v);   pred(u, v) = u;
for v ∈ V do
    for {u, w} ∈ V × V do
        if d(u, w) > d(u, v) + d(v, w) then
            d(u, w) = d(u, v) + d(v, w);
            pred(u, w) = pred(v, w);
```

$O(n^2)$ $O(n)$ $O(m) \in O(n^2)$ $\leftarrow n^2$ $\leftarrow n$ $O(n^3)$

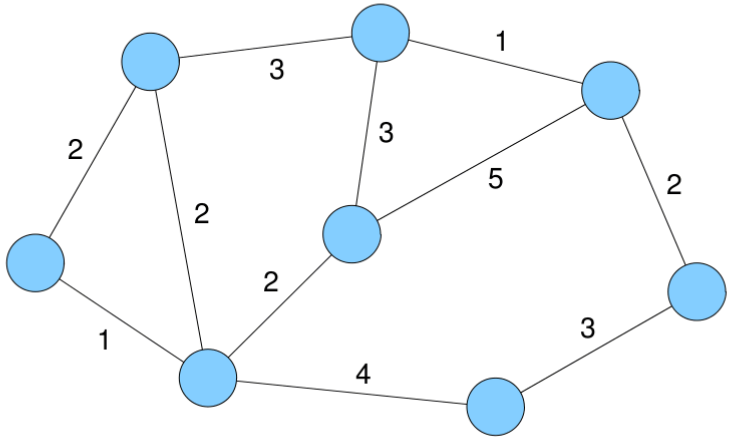
Floyd-Warshall-Algorithmus für APSP

- Komplexität: $O(n^3)$
- funktioniert auch, wenn Kanten mit negativem Gewicht existieren
- Kreise negativer Länge werden nicht direkt erkannt und verfälschen das Ergebnis, sind aber indirekt am Ende an negativen Diagonaleinträgen der Distanzmatrix erkennbar

Übersicht

Minimaler Spannbaum

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



Minimaler Spannbaum

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

Ausgabe:
- Kantenteilmenge $T \subseteq E$, so dass Graph (V, T) verbunden und $c(T) = \sum_{e \in T} c(e)$ minimal

Beobachtung:
- T formt immer einen Baum (wenn Kantengewichte echt positiv)

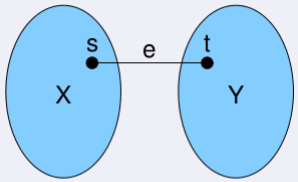
$\Rightarrow$ Minimaler Spannbaum (MSB) / Minimum Spanning Tree (MST)

Minimaler Spannbaum

Lemma

Sei
- (X, Y) *eine Partition von* V *(d.h.* $X \cup Y = V$ *und* $X \cap Y = \emptyset$*) und*
- $e = \{s, t\}$ *eine Kante mit minimalen Kosten mit* $s \in X$ *und* $t \in Y$*.*

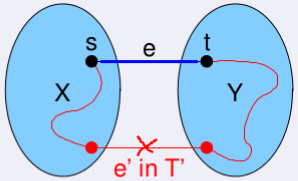
Dann gibt es einen minimalen Spannbaum T*, der* e *enthält.*



Minimaler Spannbaum

Beweis.

- gegeben X, Y und $e = \{s, t\}$: (X, Y)-Kante minimaler Kosten
- betrachte beliebigen MSB T', der e nicht enthält
- betrachte Verbindung zwischen s und t in T', darin muss es mindestens eine Kante e' zwischen X und Y geben



- Ersetzung von e' durch e führt zu Baum T'', der höchstens Kosten von MSB T' hat (also auch ein MSB ist)

□

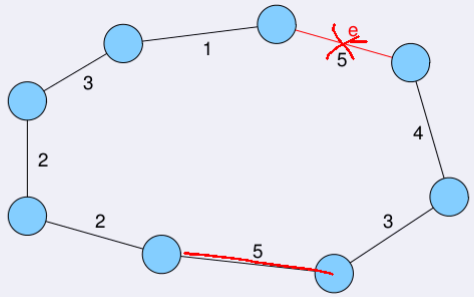
Minimaler Spannbaum

Lemma

Betrachte

- *beliebigen Kreis C in G*
- *eine Kante e in C mit maximalen Kosten*

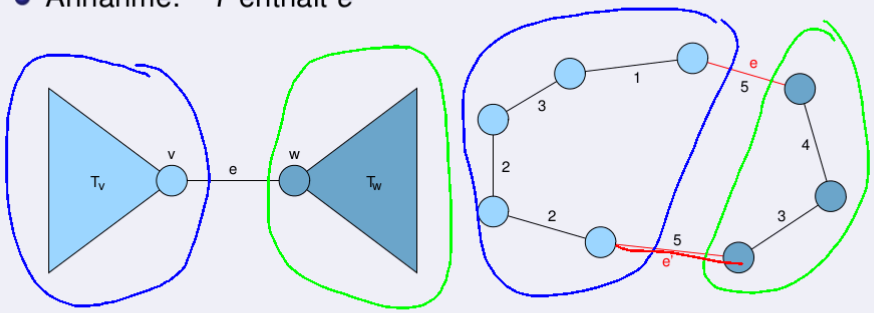
Dann ist jeder MSB in G ohne e auch ein MSB in G



Minimaler Spannbaum

Beweis.

- betrachte beliebigen MSB T in G
- Annahme: T enthält e

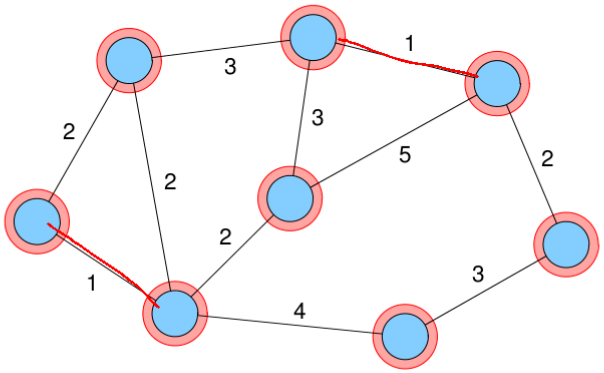


- es muss (mindestens) eine weitere Kante e' in C geben, die einen Knoten aus T_v mit einem Knoten aus T_w verbindet
- Ersetzen von e durch e' ergibt einen Baum T' dessen Gewicht nicht größer sein kann als das von T, also ist T' auch MSB

Minimaler Spannbaum

Regel:

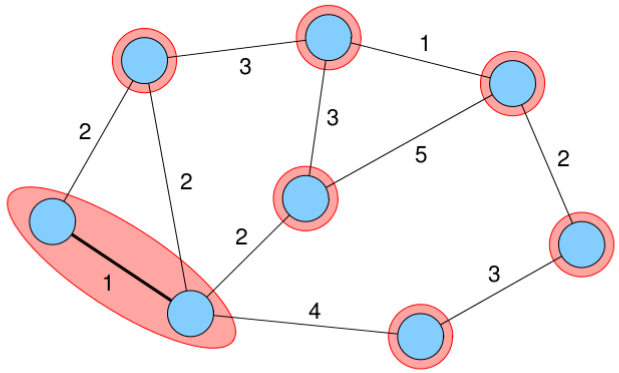
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

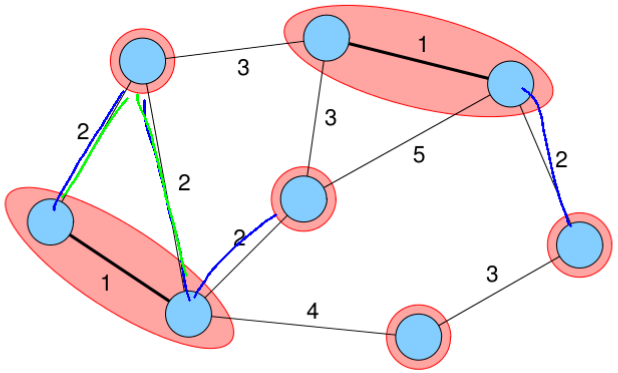
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

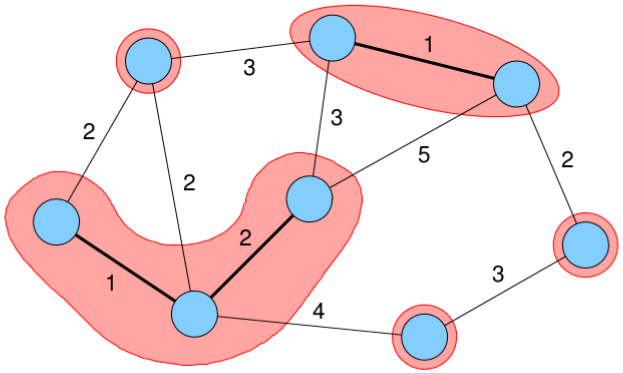
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

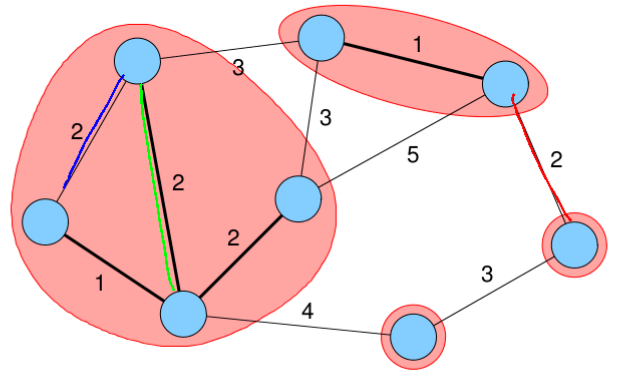
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

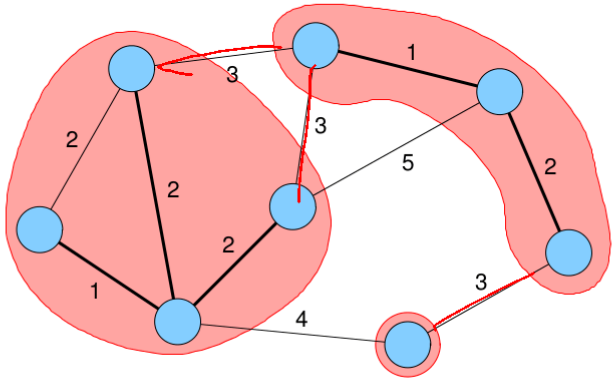
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

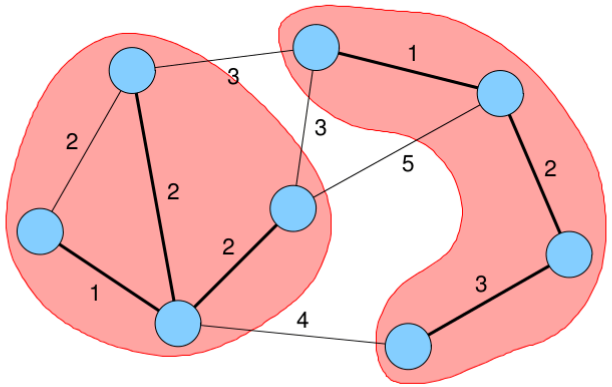
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

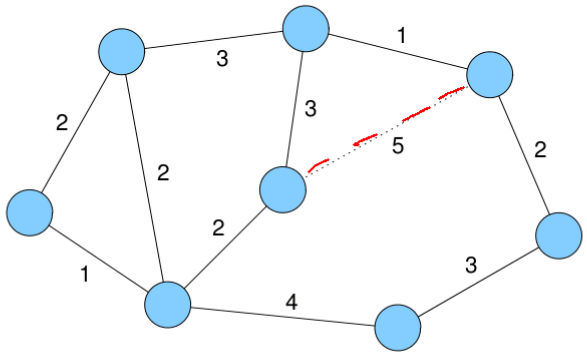
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

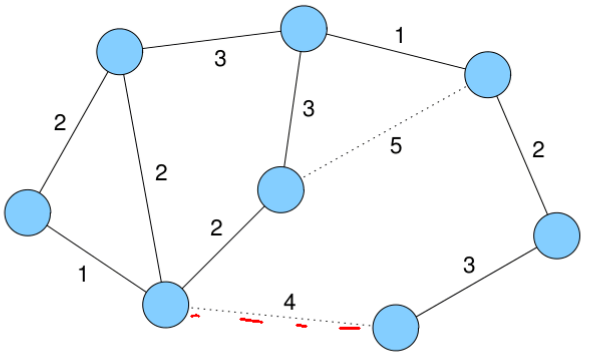
- lösche wiederholt Kante mit maximalen Kosten, so dass Zusammenhang nicht zerstört
- bis ein Baum übrig ist



Minimaler Spannbaum

Regel:

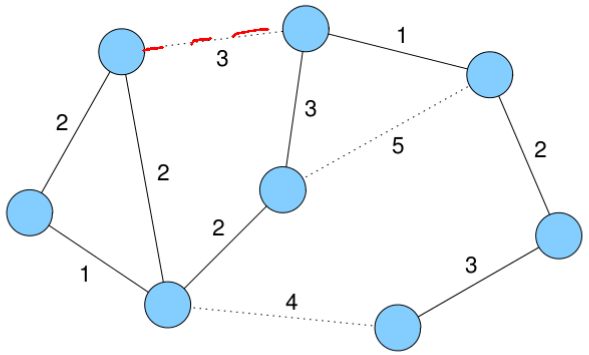
- lösche wiederholt Kante mit maximalen Kosten, so dass Zusammenhang nicht zerstört
- bis ein Baum übrig ist



Minimaler Spannbaum

Regel:

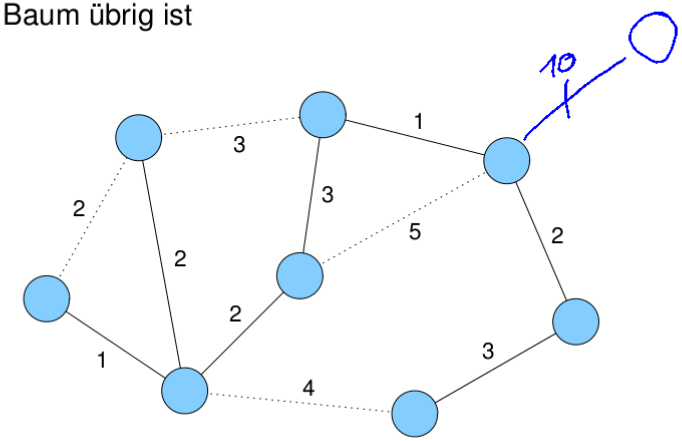
- lösche wiederholt Kante mit maximalen Kosten, so dass Zusammenhang nicht zerstört
- bis ein Baum übrig ist



Minimaler Spannbaum

Regel:

- lösche wiederholt Kante mit maximalen Kosten, so dass Zusammenhang nicht zerstört
- bis ein Baum übrig ist



Minimaler Spannbaum

Problem: Wie implementiert man die Regeln effizient?

Strategie aus dem ersten Lemma:

- sortiere Kanten aufsteigend nach ihren Kosten
- setze $T = \emptyset$ (leerer Baum)
- teste für jede Kante $\{u, v\}$ (in aufsteigender Reihenfolge), ob u und v schon in einer Zusammenhangskomponente (also im gleichen Baum) sind
- falls nicht, füge $\{u, v\}$ zu T hinzu (nun sind u und v im gleichen Baum)

Algorithmus von Kruskal

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

Problem:

- Umsetzung des Tests auf gleiche / unterschiedliche Zusammenhangskomponente

Union-Find-Datenstruktur

Union-Find-Problem:

- gegeben sind (disjunkte) Mengen von Elementen
- jede Menge hat genau einen Repräsentanten
- union soll zwei Mengen vereinigen, die durch ihren jeweiligen Repräsentanten gegeben sind
- find soll zu einem gegebenen Element die zugehörige Menge in Form des Repräsentanten finden

Anwendung:

- Knoten seien nummeriert von 0 bis $n-1$
- Array int parent[n], Einträge verweisen Richtung Repräsentant
- anfangs parent[i]=i für alle i

Union-Find-Datenstruktur

```
int find(int i) {
  if (parent[i] == i)   return i;   // ist i Wurzel des Baums?
  else {   // nein
    k = find( parent[i] );   // suche Wurzel
    parent[i] = k;   // zeige direkt auf Wurzel
    return k;   // gibt Wurzel zurück
  }
}

union(int i, int j) {
  int ri = find(i);
  int rj = find(j);   // suche Wurzeln
  if (ri ≠ rj)
    parent[ri] = rj;   // vereinigen
}
```


Algorithmus von Kruskal

```
Set<Edge> MST_Kruskal (V, E, c) {
  T = ∅;
  S = sort(E);   // aufsteigend sortieren
  for (int i = 0; i < |V|; i++)
    parent[i] = i;
  foreach (e = {u, v} ∈ S)
    if (find(u) ≠ find(v)) {
      T = T ∪ e;
      union(u, v);   // Bäume von u und v vereinigen
    }
  return T;
}
```


Gewichtete union-Operation mit Pfadkompression

- Laufzeit von find hängen von der Höhe des Baums ab
- deshalb wird am Ende von find jeder Knoten auf dem Suchpfad direkt unter die Wurzel gehängt, damit die Suche beim nächsten Mal direkt zu diesem Knoten kommt (Pfadkompression)
- weiterhin sollte bei union der niedrigere Baum unter die Wurzel des höheren gehängt werden (gewichtete Vereinigung)

$\Rightarrow$ Höhe des Baums ist dann $O(\log n)$

Gewichtete union-Operation

```
union(int i, int j) {
  int ri = find(i);
  int rj = find(j);   // suche Wurzeln
  if (ri ≠ rj)
    if (height[ri] < height[rk])
      parent[ri] = rj;
    else {
      parent[rj] = ri;
      if (height[ri] == height[rk])
        height[ri]++;
    }
}
```


union / find - Kosten

Situation:

- Folge von union / find -Operationen auf einer Partition von n Elementen, darunter $n-1$ union-Operationen

Komplexität:

- amortisiert $\log^* n$ pro Operation, wobei

$$\log^* n = \min\{i \geq 1 : \underbrace{\log \log \ldots \log}_{i\text{-mal}} n \leq 1\}$$

- bessere obere Schranke: mit inverser Ackermannfunktion (Vorlesung Effiziente Algorithmen und Datenstrukturen I)
- Gesamtkosten für Kruskal-Algorithmus: $O(m \log n)$ (Sortieren)

Algorithmus von Prim

Problem: Wie implementiert man die Regeln effizient?

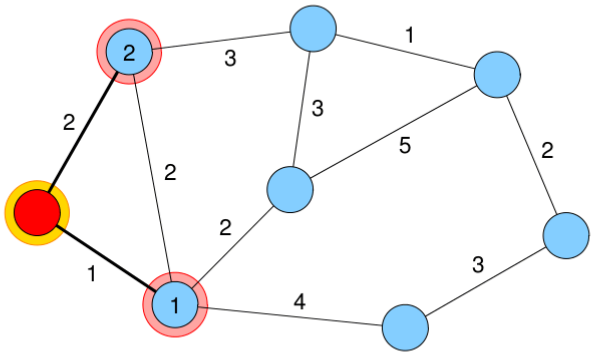
Alternative Strategie aus dem ersten Lemma:

- betrachte wachsenden Baum T, anfangs bestehend aus beliebigem einzelnen Knoten s
- füge zu T eine Kante mit minimalem Gewicht von einem Baumknoten zu einem Knoten außerhalb des Baums ein (bei mehreren Möglichkeiten egal welche)

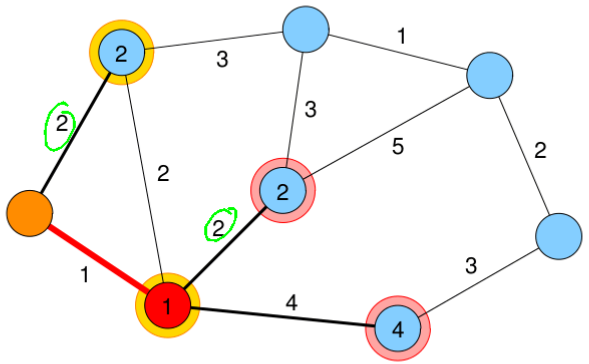
$\Rightarrow$ Baum umfasst jetzt 1 Knoten / Kante mehr

- wiederhole Auswahl bis alle n Knoten im Baum

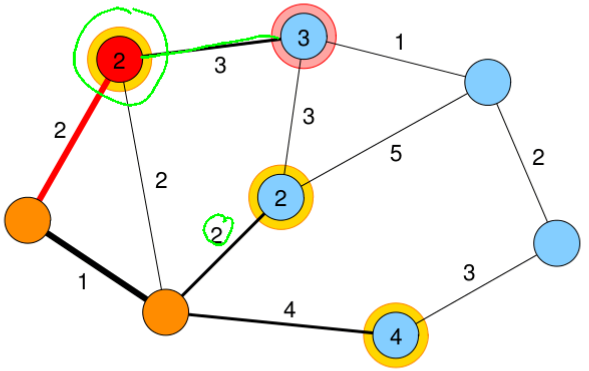
Algorithmus von Prim



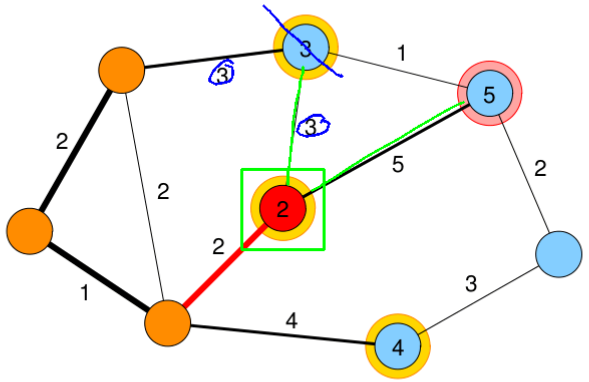
Algorithmus von Prim



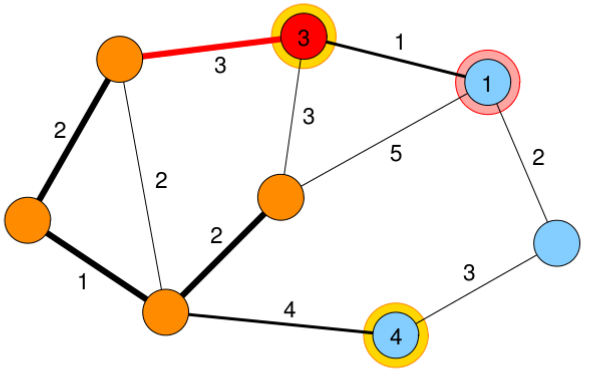
Algorithmus von Prim



Algorithmus von Prim



Algorithmus von Prim



Jarnik-Prim-Algorithmus für MSB

Input : $G = (V, E)$, $c : E \to \mathbb{R}_+$, $s \in V$
Output : Minimaler Spannbaum zwischen allen $v \in V$ (in Array pred)

```
d[v] = ∞ for all v ∈ V \ s;
d[s] = 0;   pred[s] = ⊥;
pq = ⟨⟩;   pq.insert(s,0);
while ¬pq.empty() do
    v = pq.deleteMin();
    forall the {v,w} ∈ E do
        newWeight = c(v,w);
        if newWeight < d[w] then
            pred[w] = v;
            if d[w] == ∞ then pq.insert(w,newWeight);
            else if w ∈ pq then
                pq.decreaseKey(w,newWeight);
            d[w] = newWeight;
```


Jarnik-Prim-Algorithmus

Laufzeit:

$$O\Big(n \cdot (T_{\text{insert}}(n) + T_{\text{deletMin}}(n)) + m \cdot T_{\text{decreaseKey}}(n)\Big)$$

Binärer Heap:

- alle Operationen $O(\log n)$, also
- gesamt: $O((m+n)\log n)$

Fibonacci-Heap: amortisierte Kosten

- $O(1)$ für insert und decreaseKey,
- $O(\log n)$ deleteMin
- gesamt: $O(m + n\log n)$

Übersicht