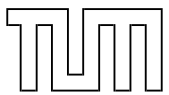
Grundlagen: Algorithmen und Datenstrukturen

Prof. Dr. Hanjo Täubig

Lehrstuhl für Effiziente Algorithmen
(Prof. Dr. Ernst W. Mayr)
Institut für Informatik
Technische Universität München

Sommersemester 2010



Übersicht

APSP / Floyd-Warshall-Algorithmus

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 kürzeste 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)

APSP / Floyd-Warshall-Algorithmus

Algorithmus 16 : Floyd-Warshall APSP Algorithmus

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

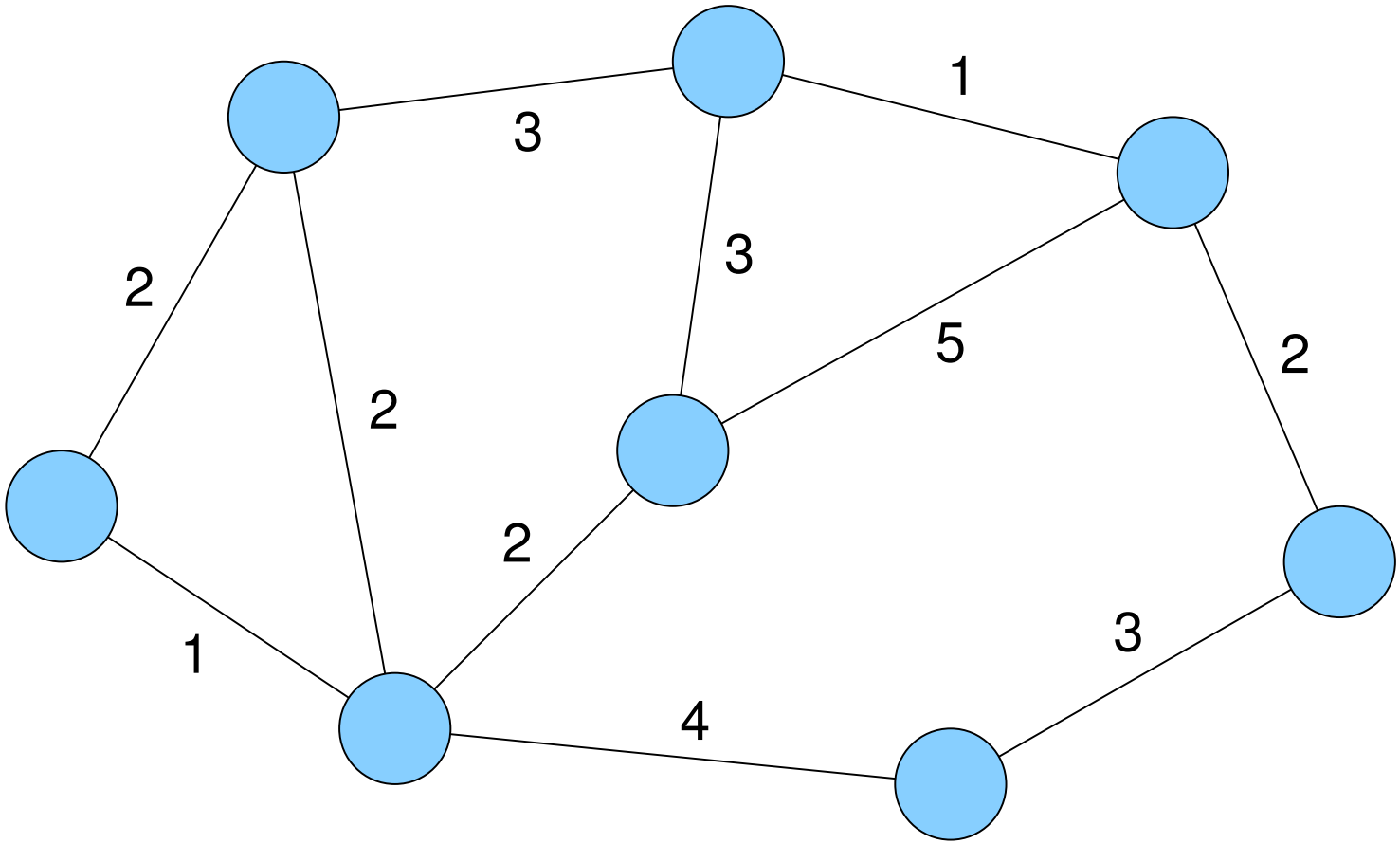
for $u, v \in V$ **do**
  $d(u, v) = \infty$; $\text{pred}(u, v) = 0$;
for $v \in V$ **do** $d(v, v) = 0$;
for $\{u, v\} \in E$ **do**
  $d(u, v) = c(u, v)$; $\text{pred}(u, v) = u$;
for $v \in V$ **do**
  for $\{u, w\} \in V \times V$ **do**
    if $d(u, w) > d(u, v) + d(v, w)$ **then**
      $d(u, w) = d(u, v) + d(v, w)$;
      $\text{pred}(u, w) = \text{pred}(v, w)$;

APSP / Floyd-Warshall-Algorithmus

- 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

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)

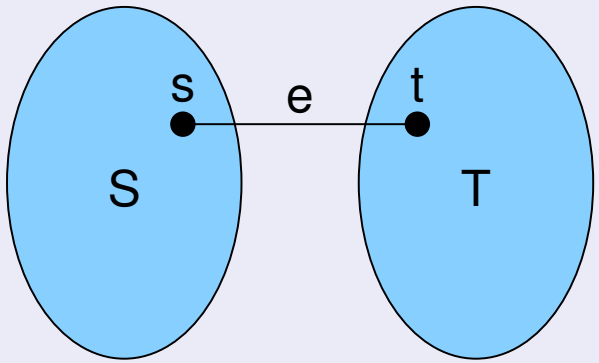
Minimaler Spannbaum

Lemma

Sei

- (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* $t \in T$*.*

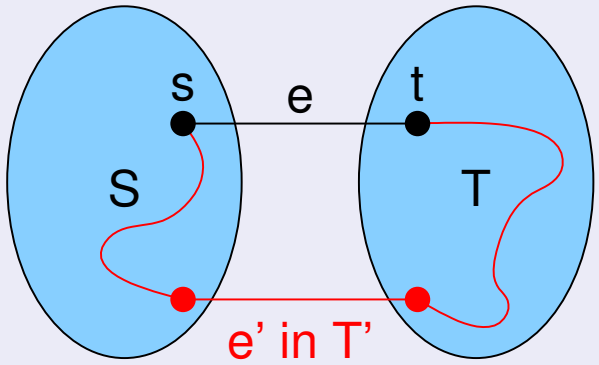
Dann gibt es einen minimalen Spannbaum T, der e enthält.



Minimaler Spannbaum

Beweis.

- betrachte beliebigen MSB T'
- $e = \{s, t\}$: (S, T)-Kante minimaler Kosten
- betrachte Verbindung zwischen s und t in T' mit Kante e' zwischen S und T



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

□

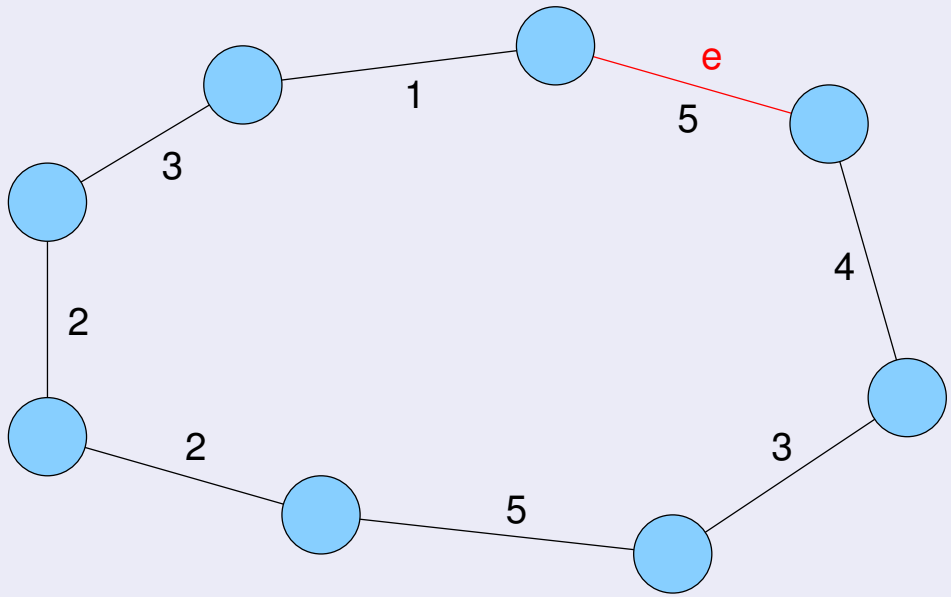
Minimaler Spannbaum

Lemma

Betrachte

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

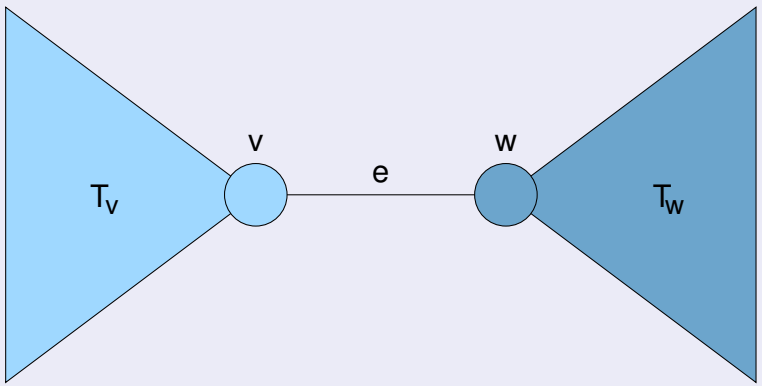
Dann ist jeder MSB in G ohne e auch ein MSB in G

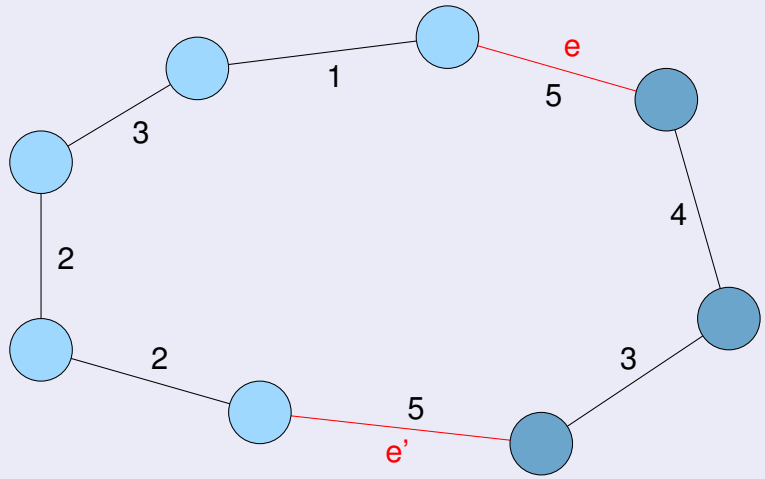


Minimaler Spannbaum

Beweis.

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



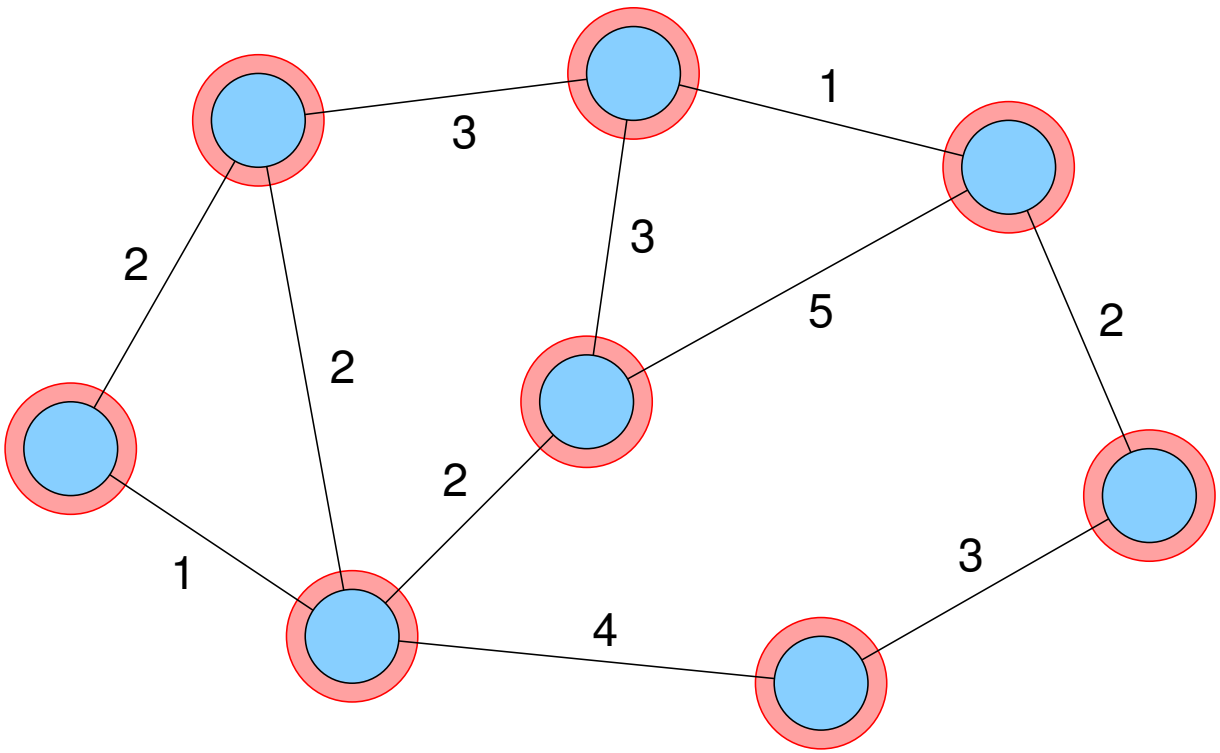


- es muss (mindestens) eine weitere Kante e' in C geben, die einen Knoten aus T_v mit einem Knoten aus T_w verbindet
- Ersetzen von e durch e' ergibt einen Baum T' dessen Gewicht nicht größer sein kann als das von T, also ist T' auch MSB

Minimaler Spannbaum

Regel:

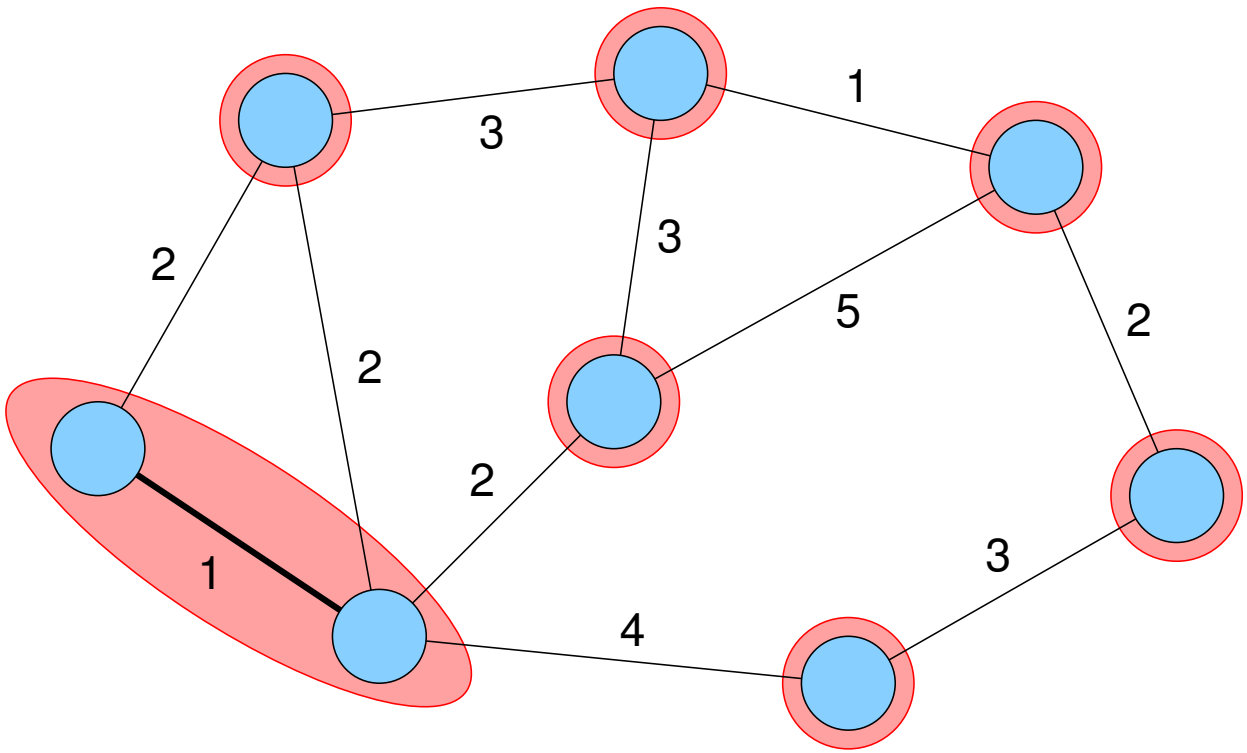
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

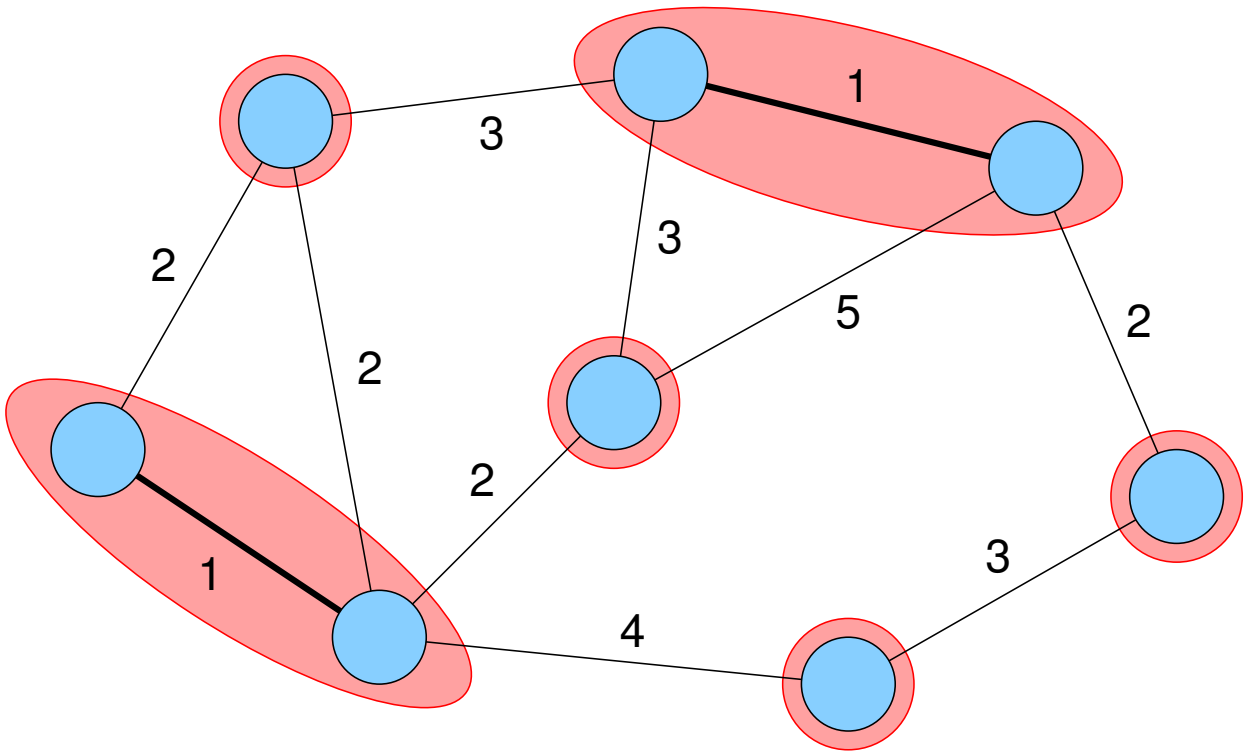
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

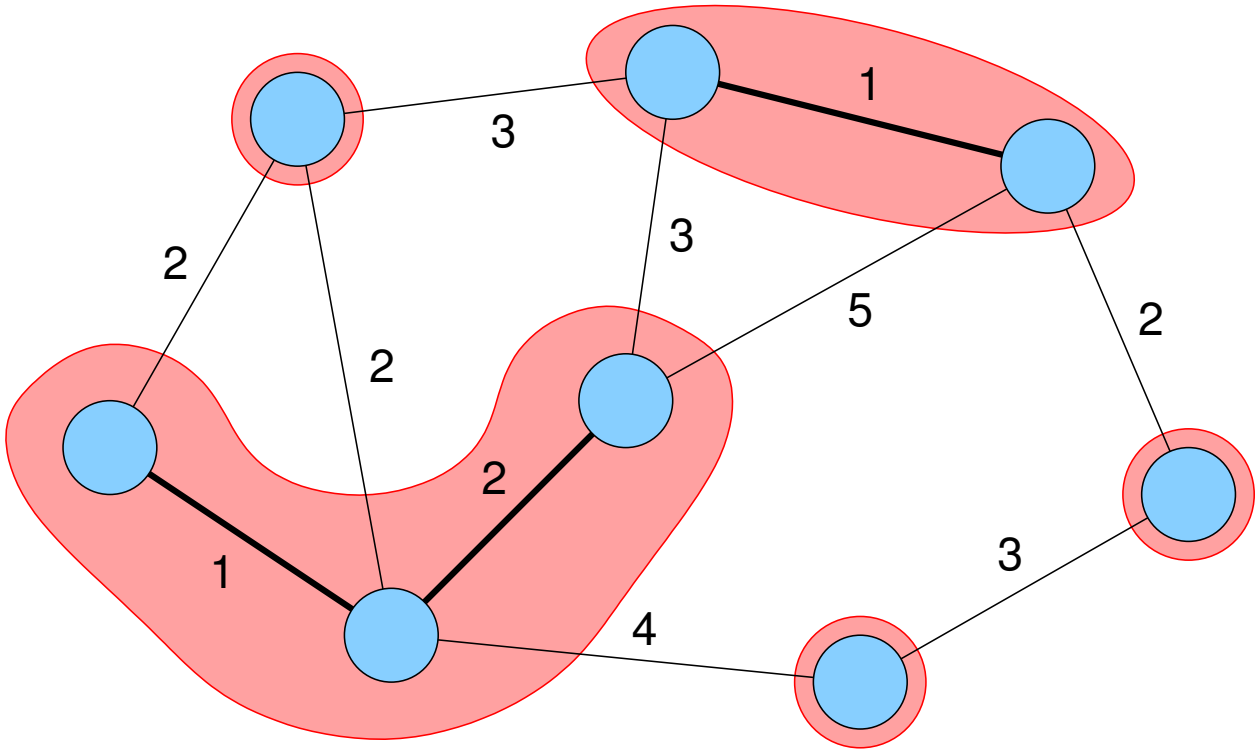
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

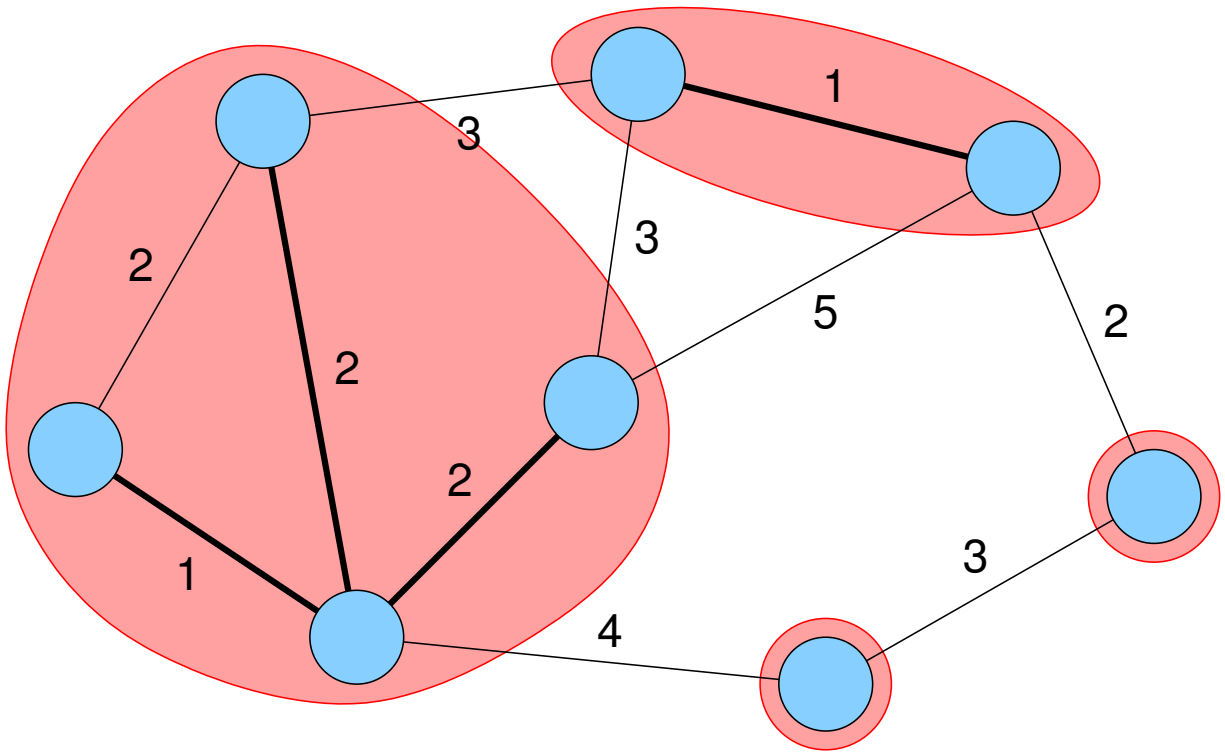
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

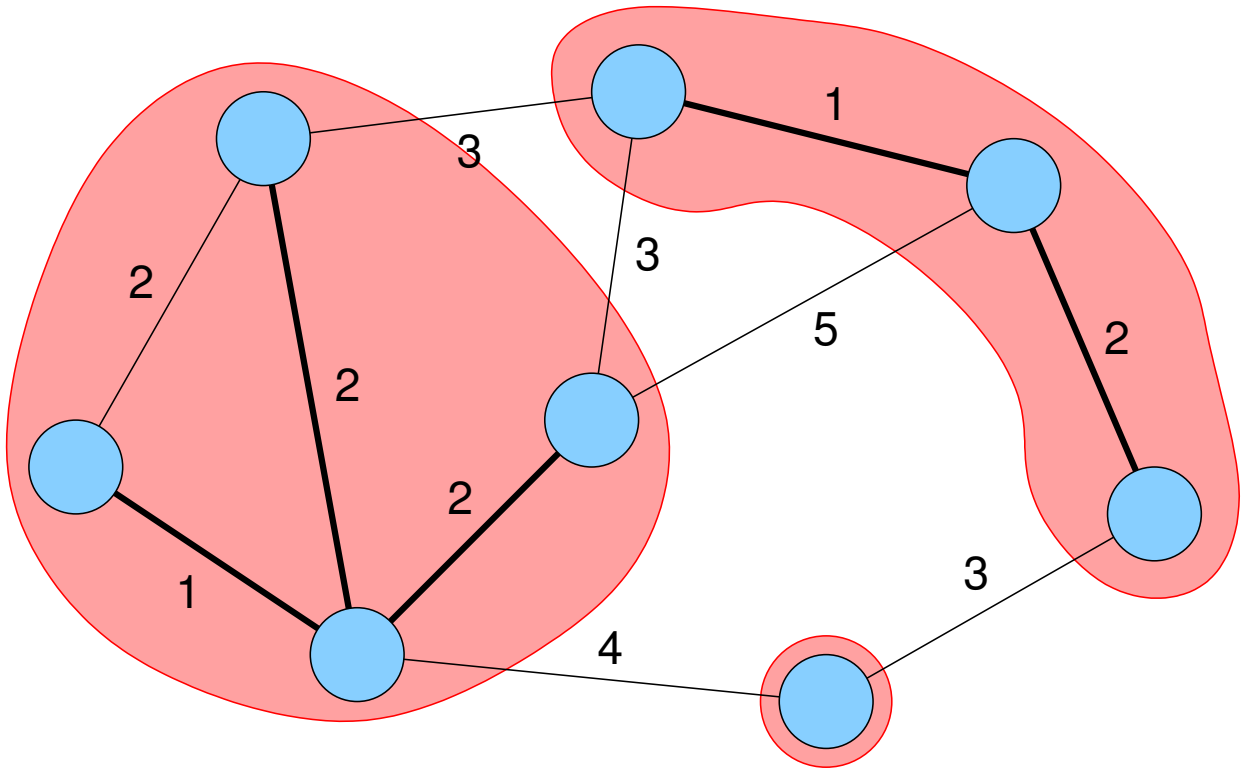
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

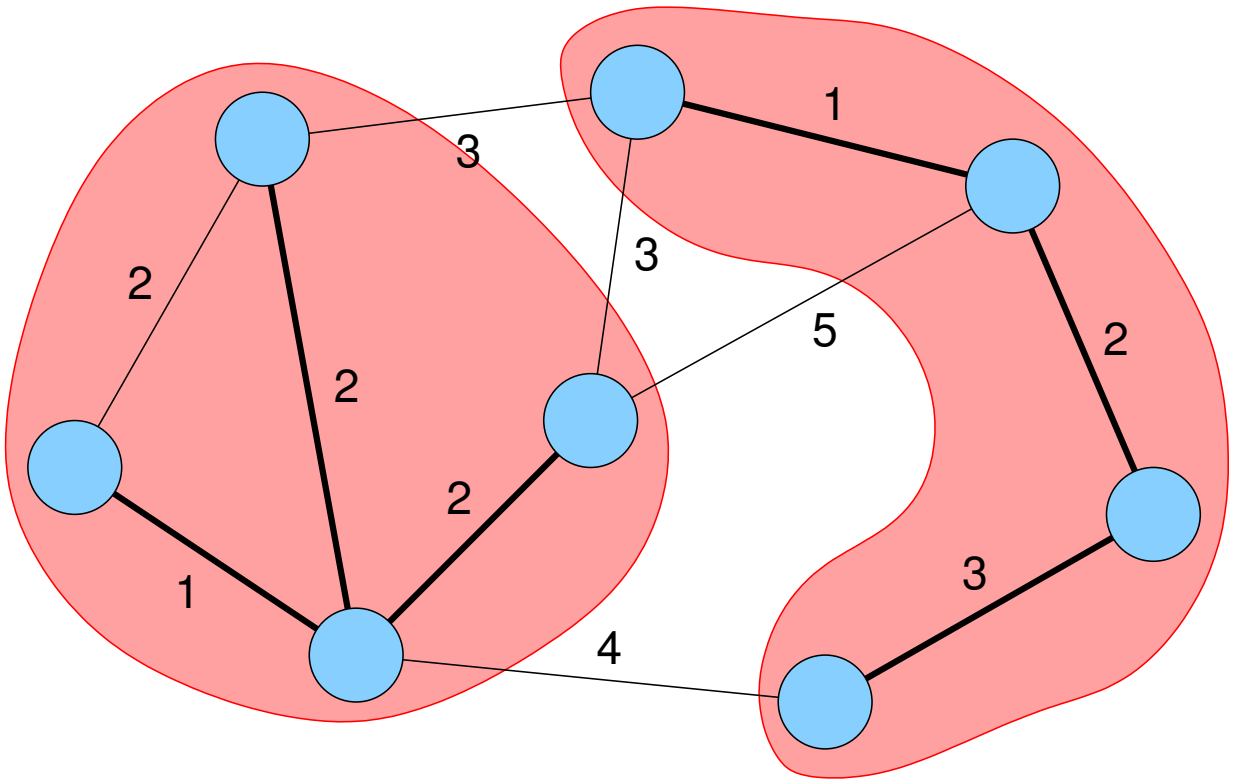
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

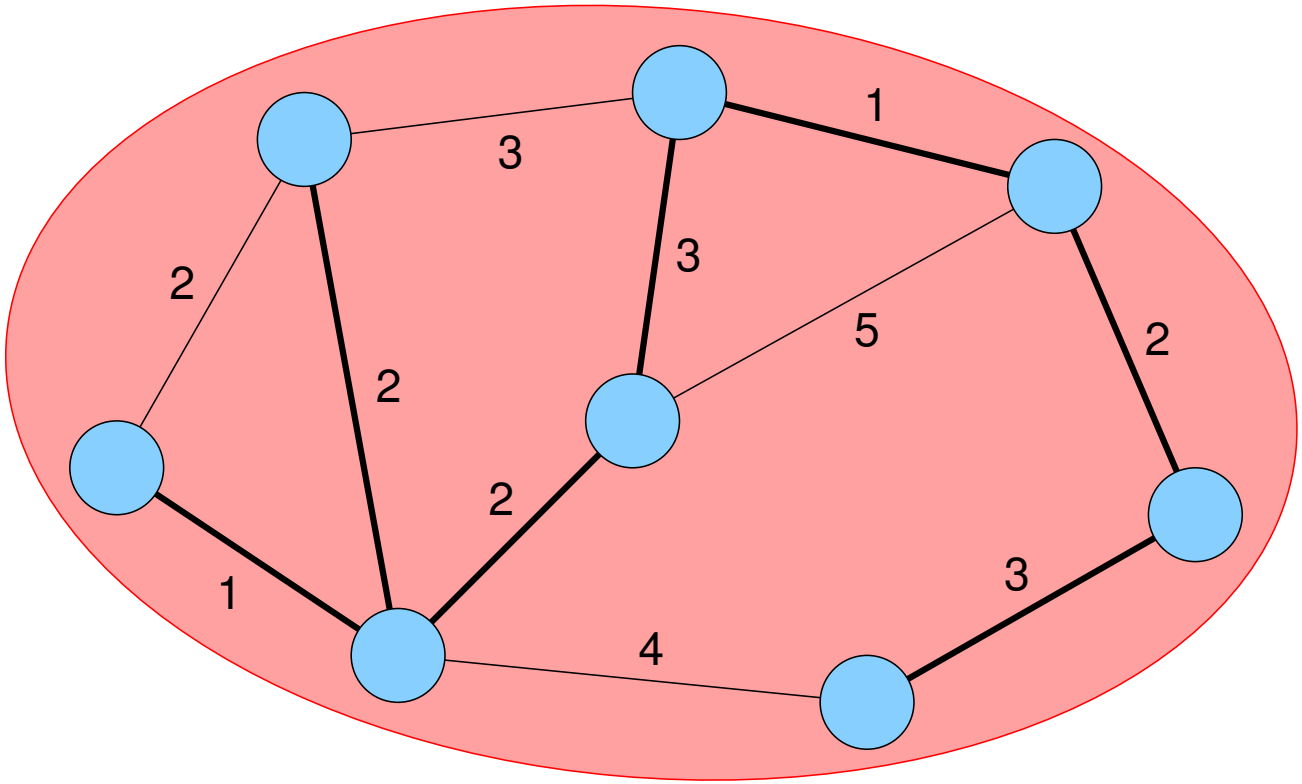
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

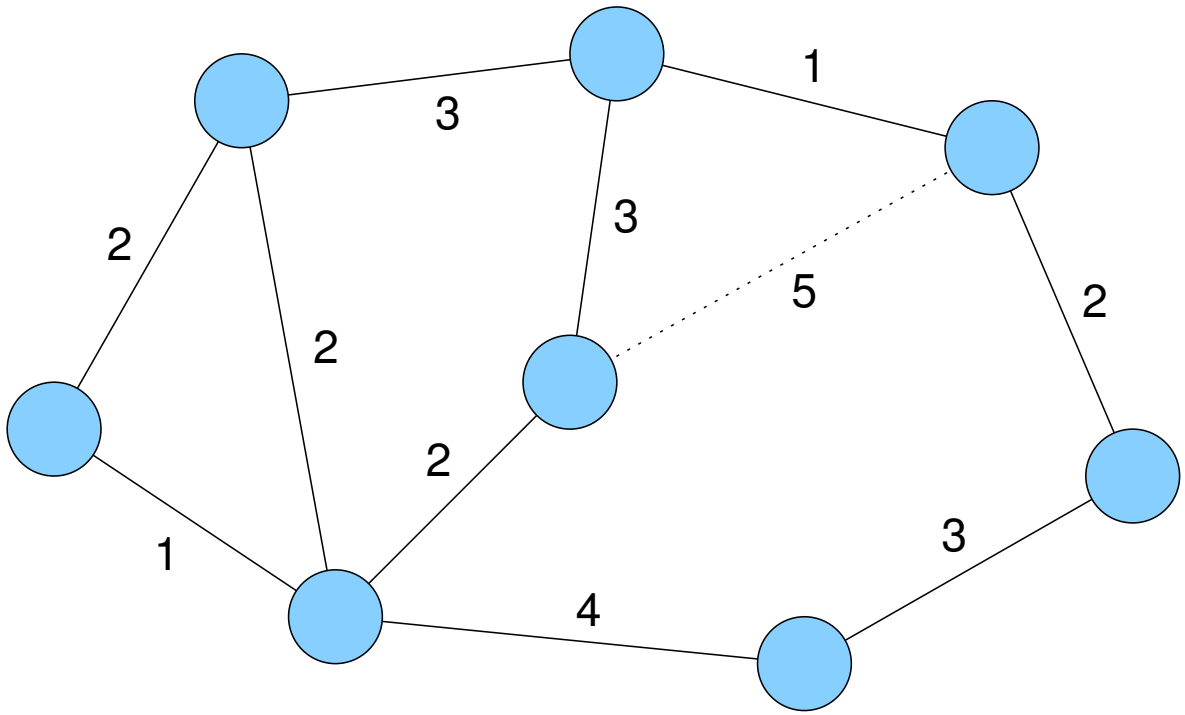
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

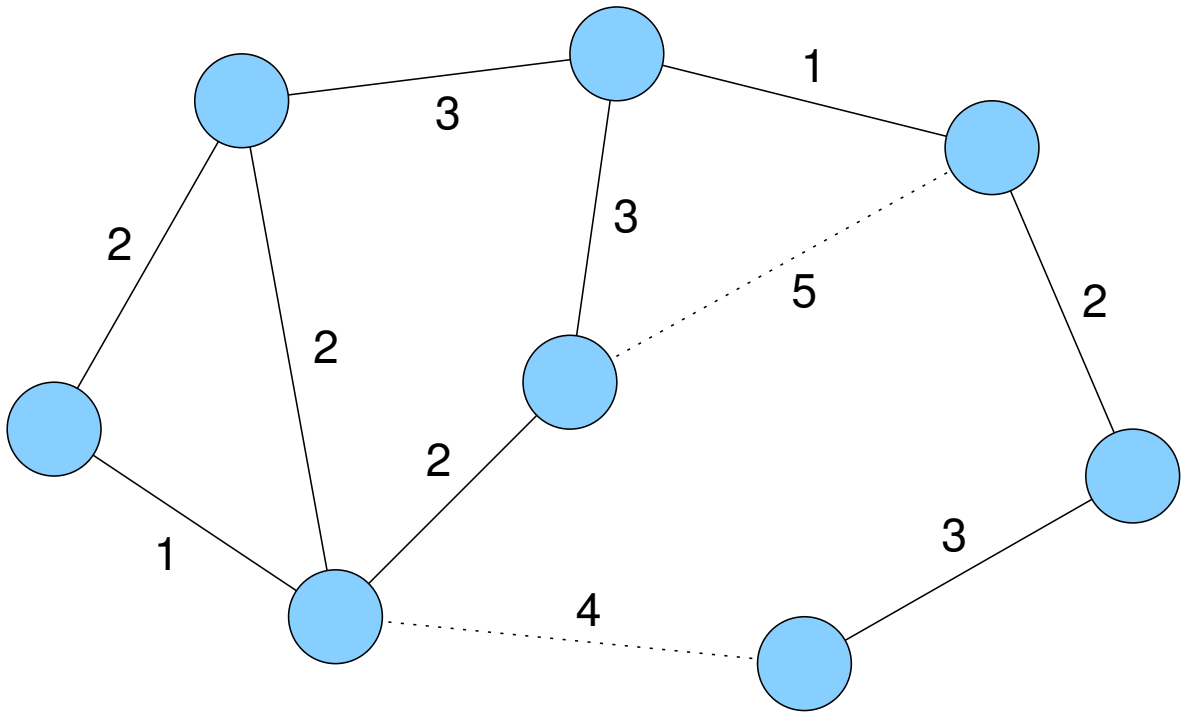
- lösche wiederholt Kante mit maximalen Kosten, so dass Zusammenhang nicht zerstört
- bis ein Baum übrig ist



Minimaler Spannbaum

Regel:

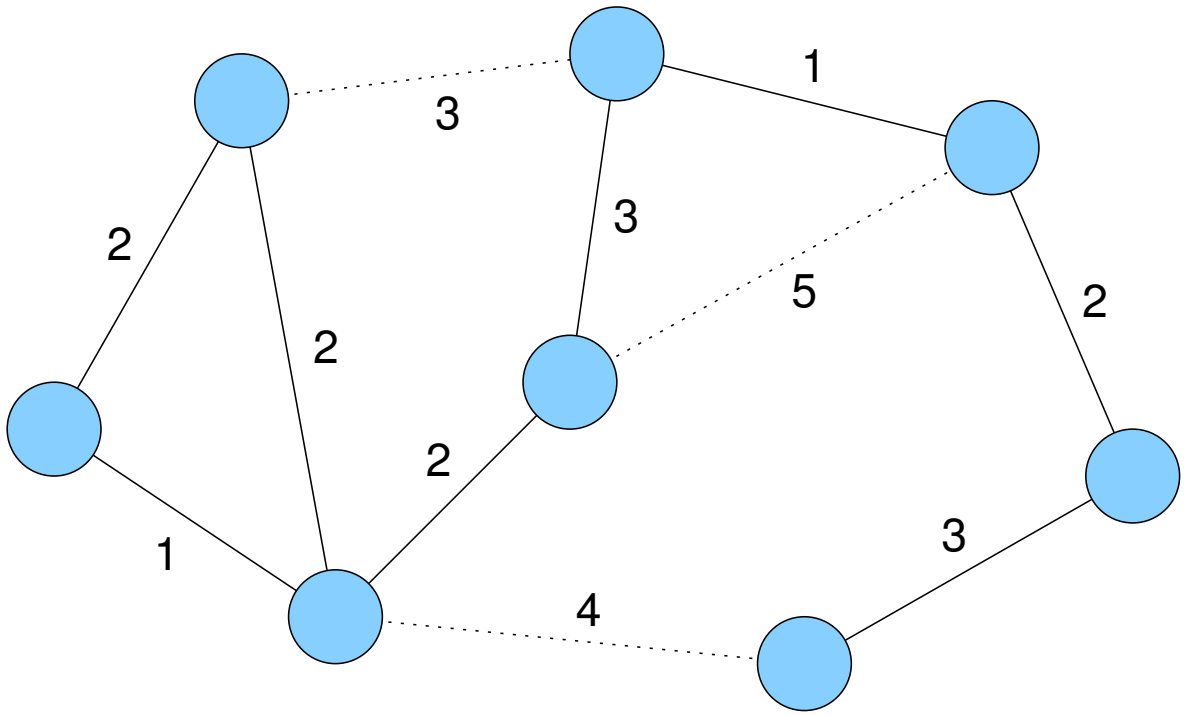
- lösche wiederholt Kante mit maximalen Kosten, so dass Zusammenhang nicht zerstört
- bis ein Baum übrig ist



Minimaler Spannbaum

Regel:

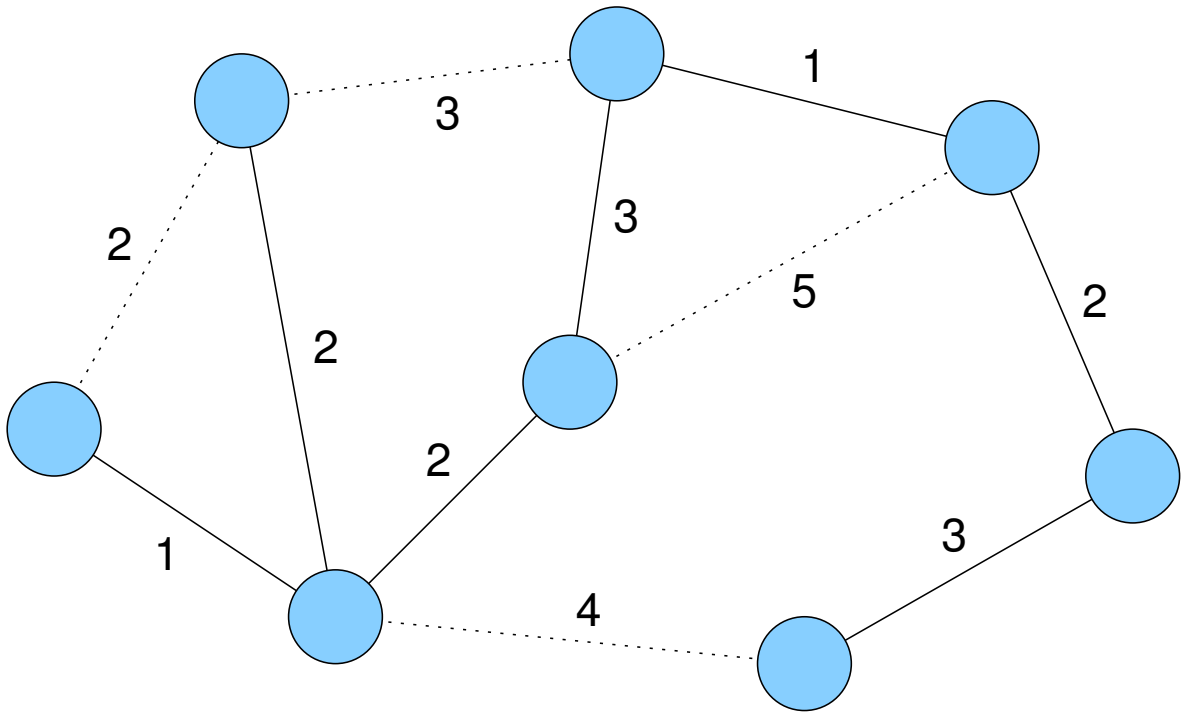
- lösche wiederholt Kante mit maximalen Kosten, so dass Zusammenhang nicht zerstört
- bis ein Baum übrig ist



Minimaler Spannbaum

Regel:

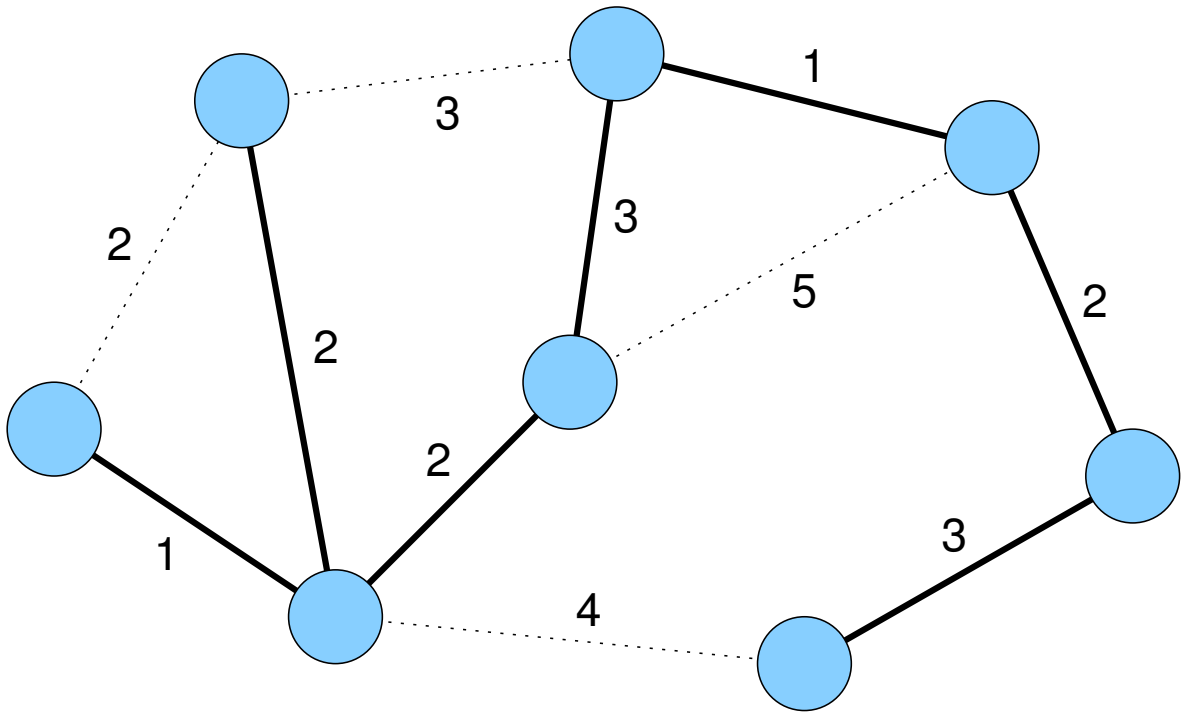
- lösche wiederholt Kante mit maximalen Kosten, so dass Zusammenhang nicht zerstört
- bis ein Baum übrig ist



Minimaler Spannbaum

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
    }
}

void 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

```
void 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: $\mathcal{O}(m \log n)$ (Sortieren)

Algorithmus von Prim

Problem: Wie implementiert man die Regeln effizient?

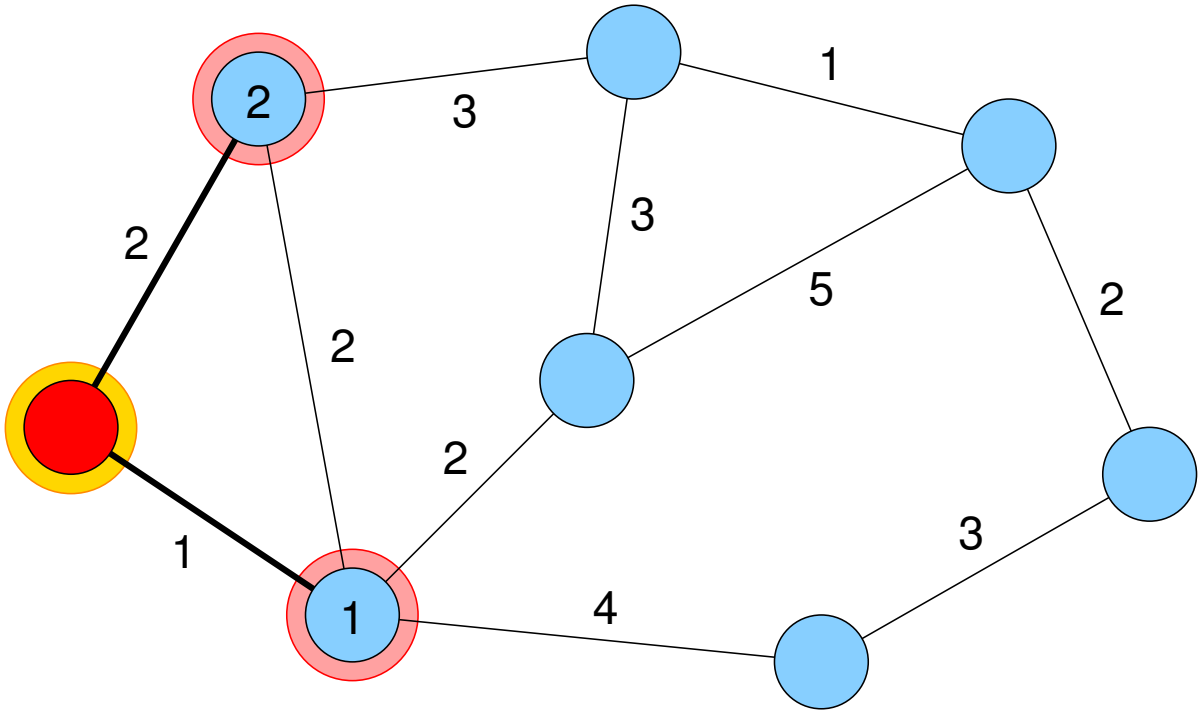
Alternative Strategie aus dem ersten Lemma:

- 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)

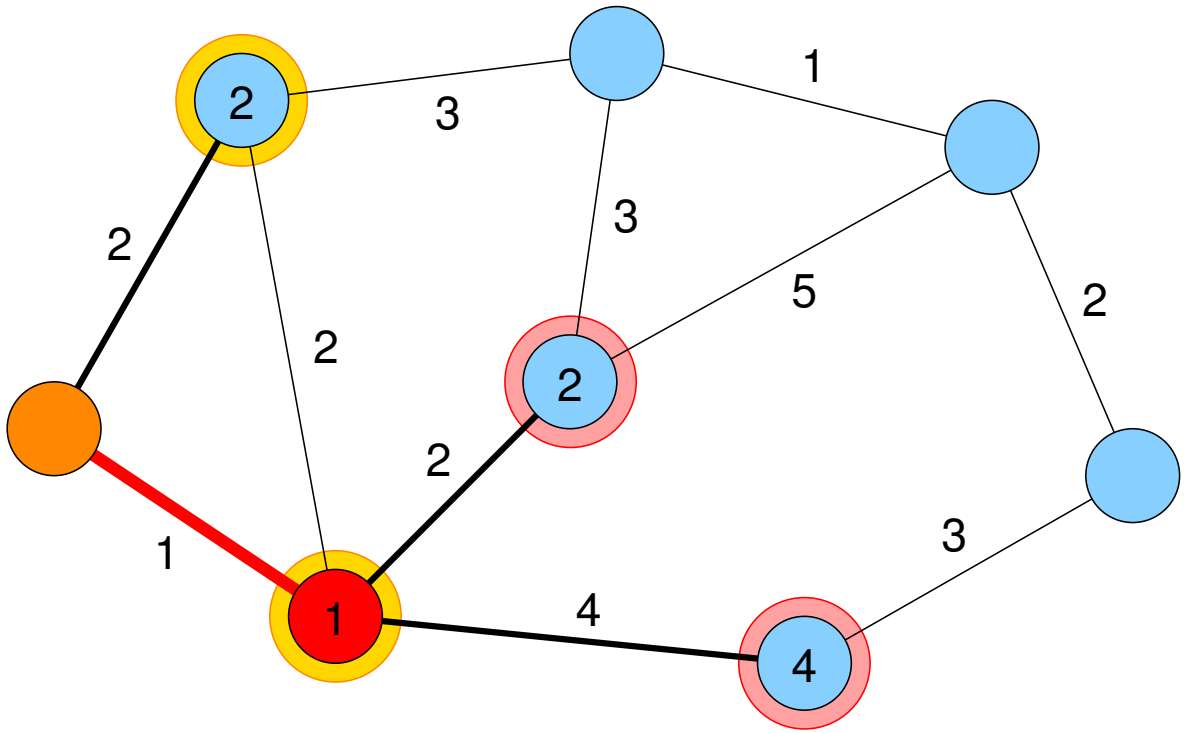
$\Rightarrow$ Baum umfasst jetzt 1 Knoten / Kante mehr

- wiederhole Auswahl bis alle n Knoten im Baum

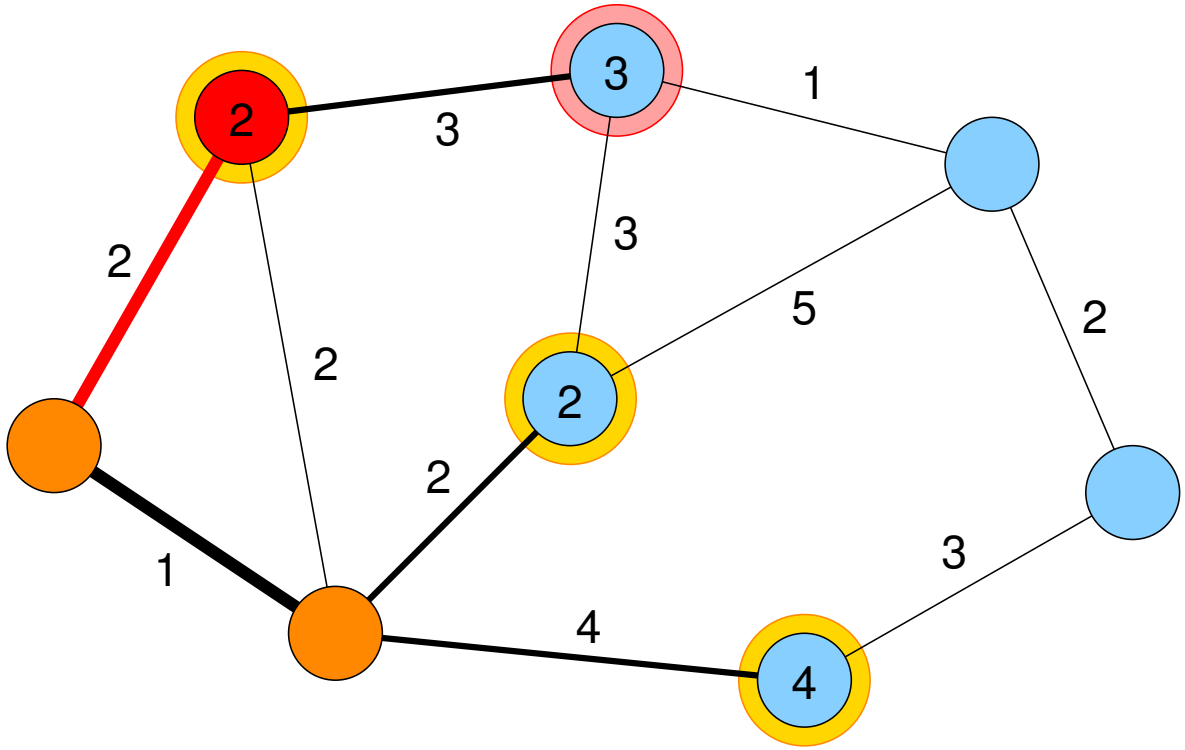
Algorithmus von Prim



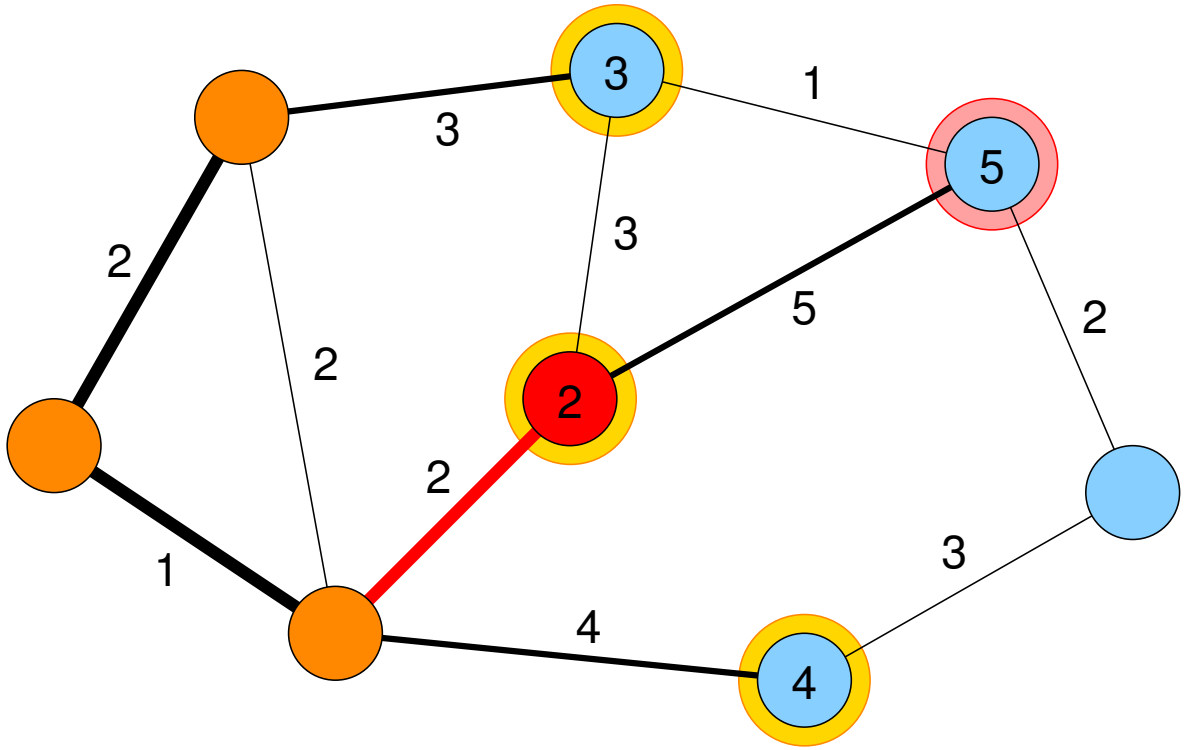
Algorithmus von Prim



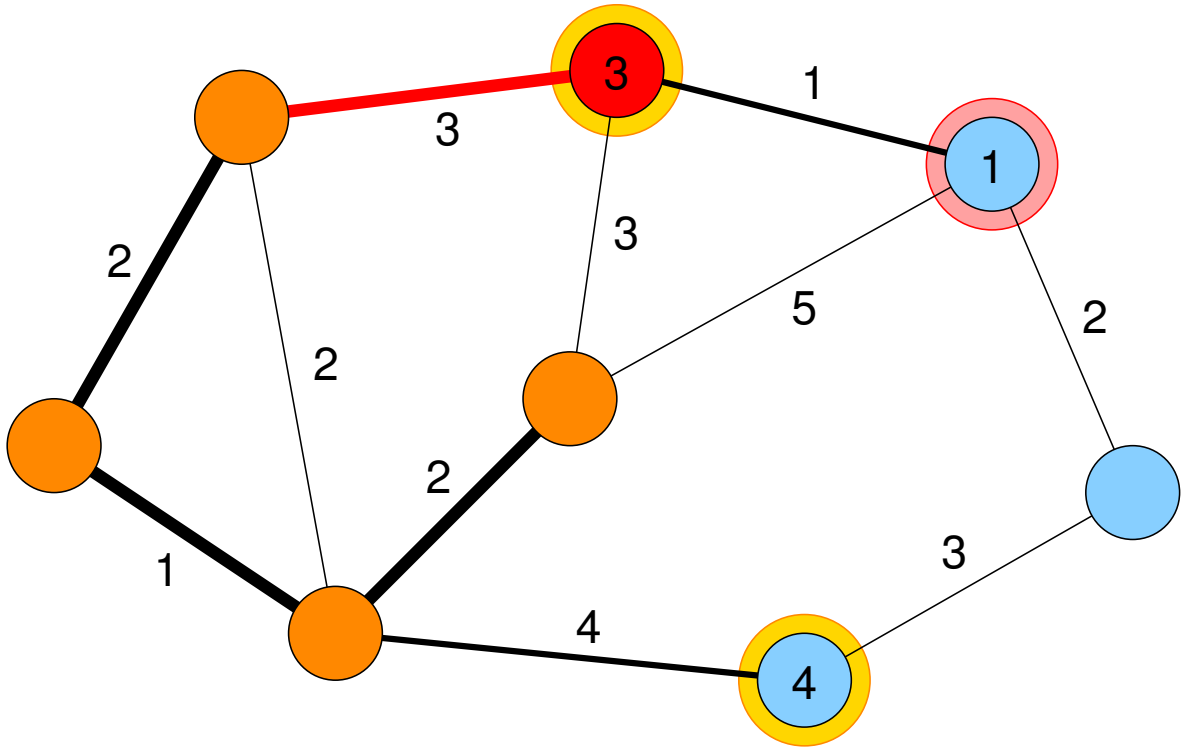
Algorithmus von Prim



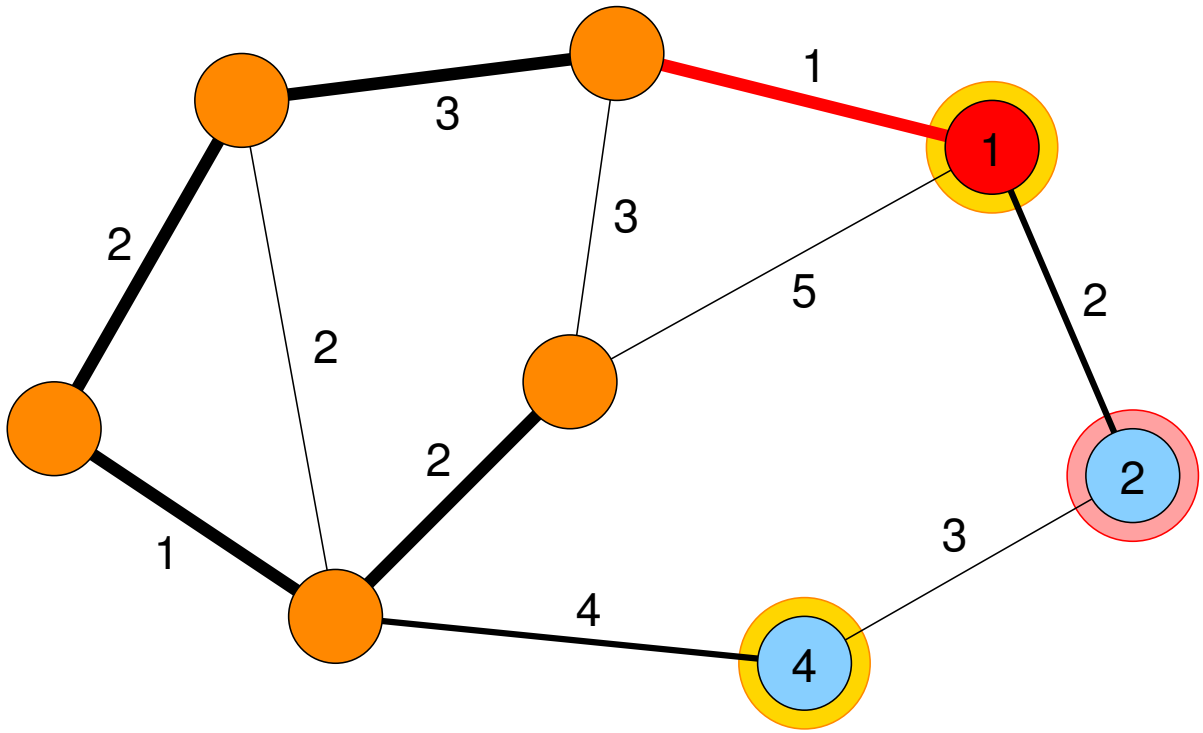
Algorithmus von Prim



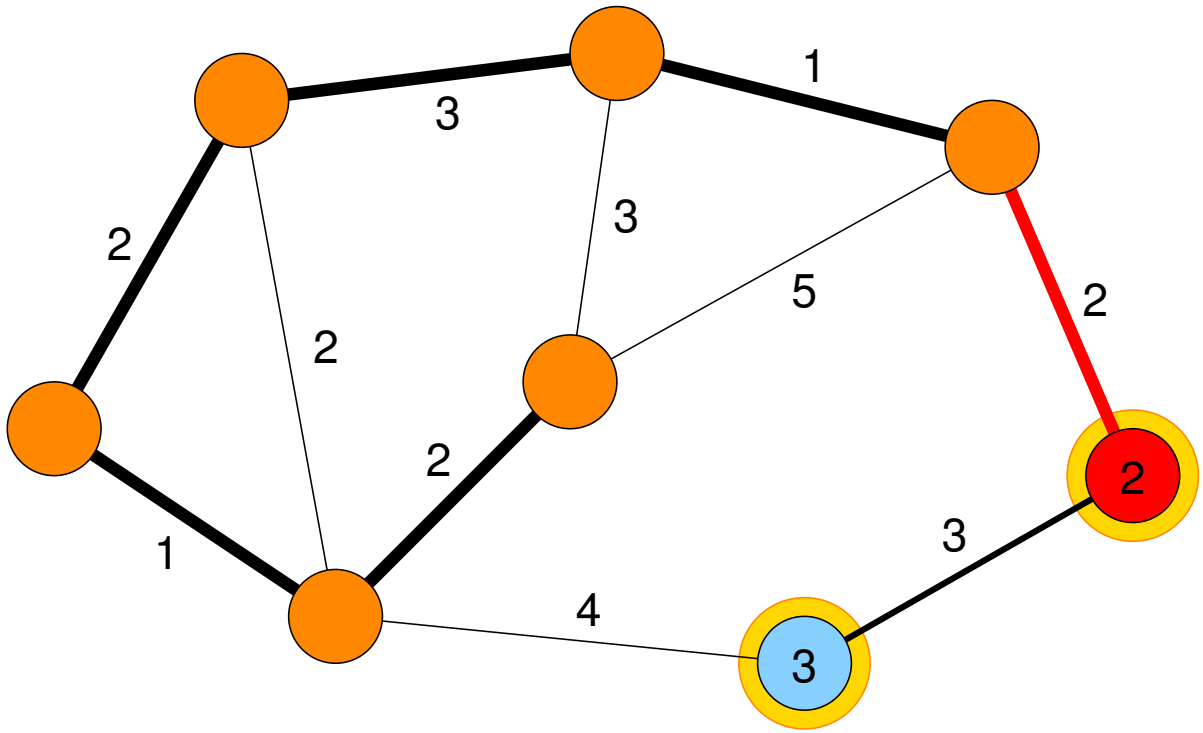
Algorithmus von Prim



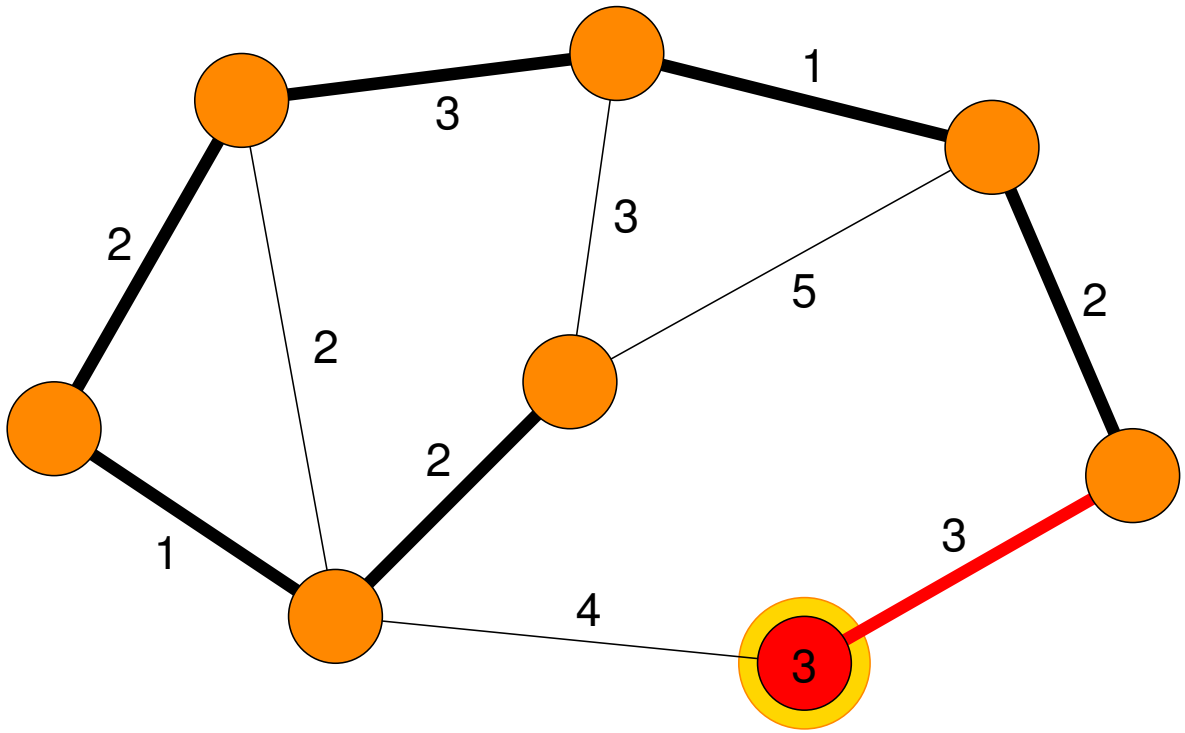
Algorithmus von Prim



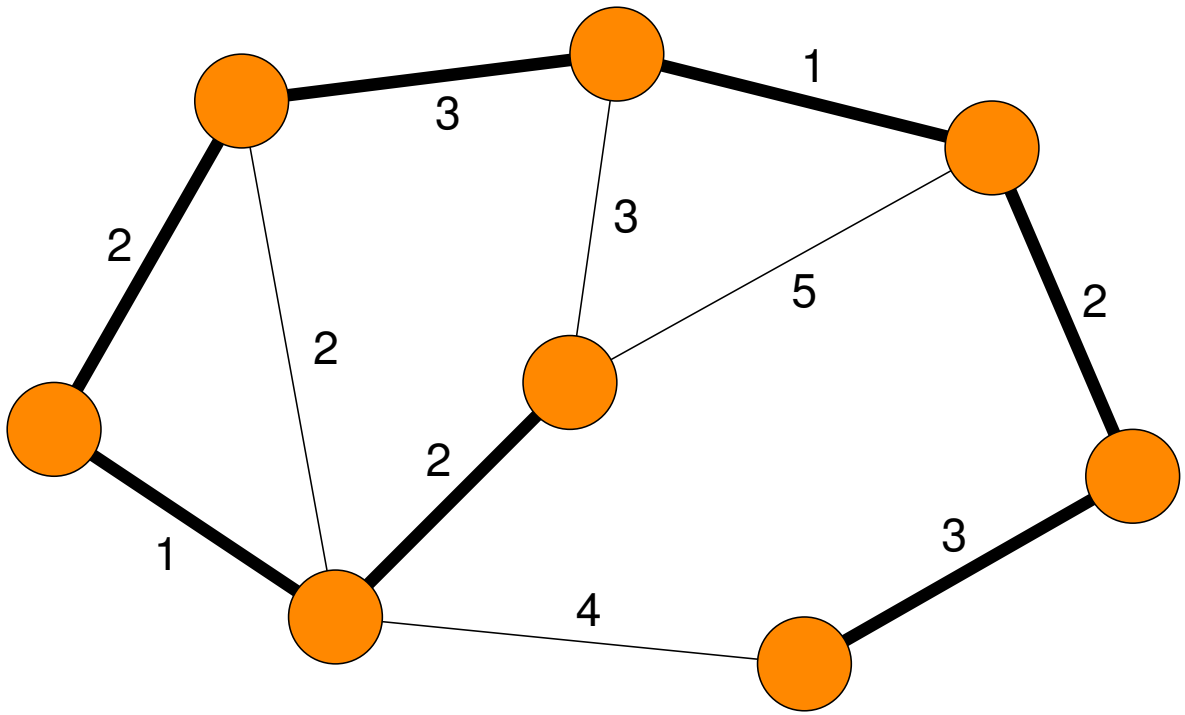
Algorithmus von Prim



Algorithmus von Prim



Algorithmus von Prim



Algorithmus 17 : 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:

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

Binärer Heap:

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

Fibonacci-Heap: amortisierte Kosten

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