Script generated by TTT

Title: Seidl: GAD (28.06.2016)

Date: Tue Jun 28 14:22:48 CEST 2016

Duration: 91:58 min

Pages: 47

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)

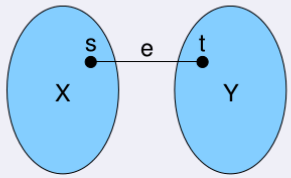
Minimaler Spannbaum

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* $t \in Y$.

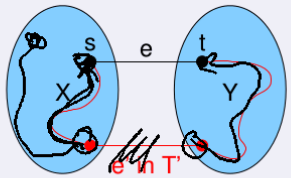
Dann gibt es einen minimalen Spannbaum T, der e enthält.



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)

□

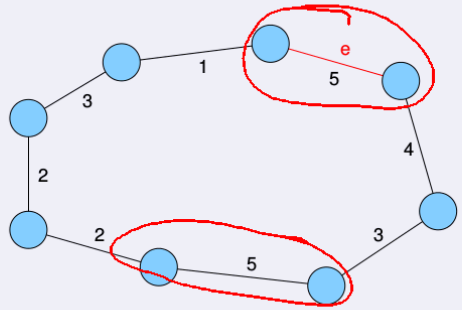
Minimaler Spannbaum

Lemma

Betrachte

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

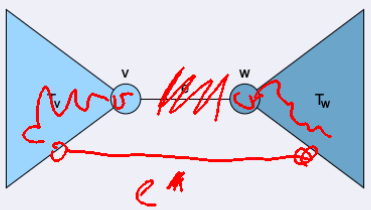
Dann ist jeder MSB in G ohne e auch ein MSB in G

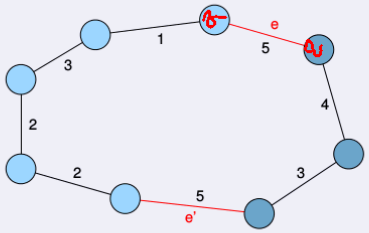


Minimaler Spannbaum

Beweis.

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



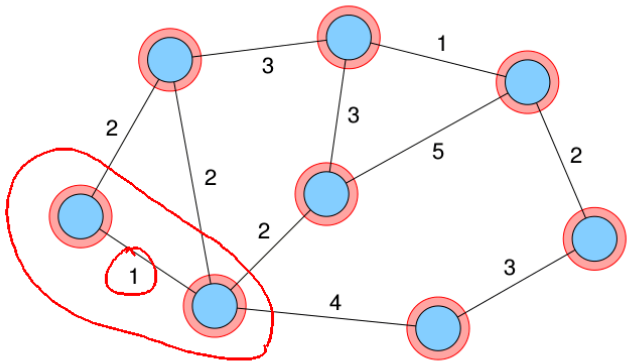


- es muss (mindestens) eine weitere Kante e' in C geben, die einen Knoten aus T_v mit einem Knoten aus T_w verbindet
- Ersetzen von e durch e' ergibt einen Baum T' dessen Gewicht nicht größer sein kann als das von T, also ist T' auch MSB

Minimaler Spannbaum

Regel:

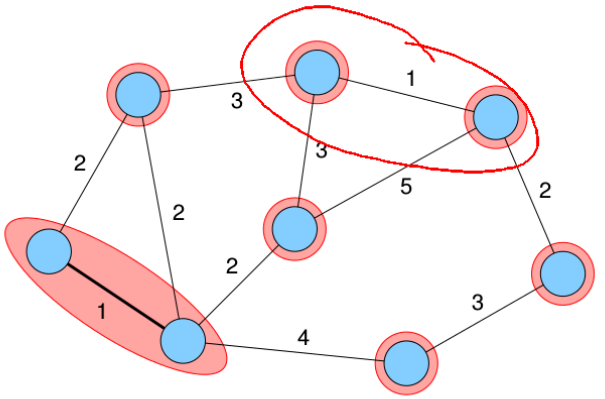
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

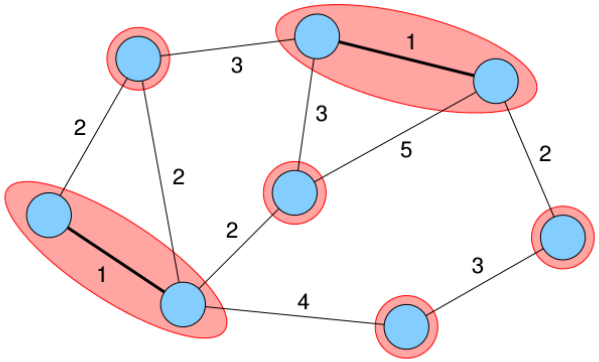
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

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

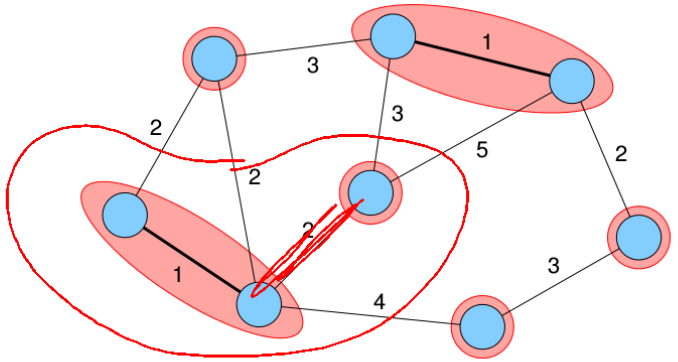


Graphen Minimale Spannbäume

Minimaler Spannbaum

Regel:

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



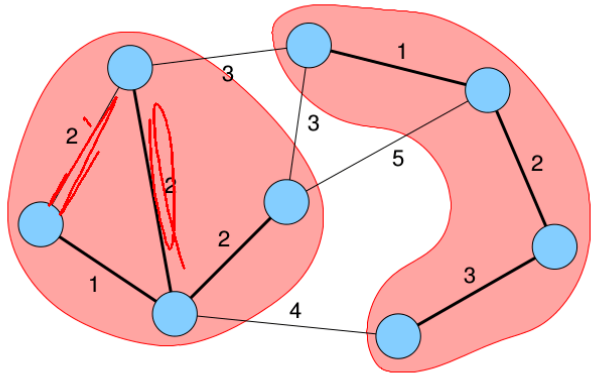
H. Seidl (TUM) GAD SS'16 529

Graphen Minimale Spannbäume

Minimaler Spannbaum

Regel:

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



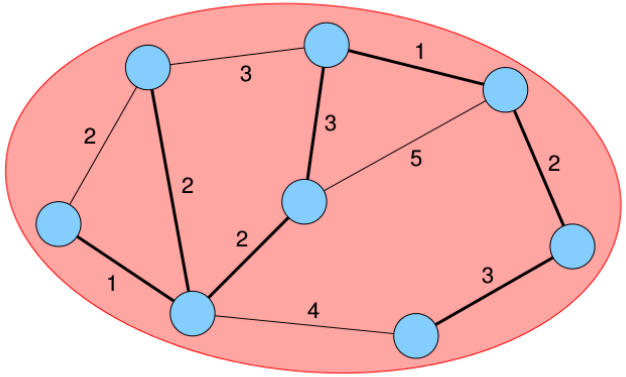
H. Seidl (TUM) GAD SS'16 533

Graphen Minimale Spannbäume

Minimaler Spannbaum

Regel:

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

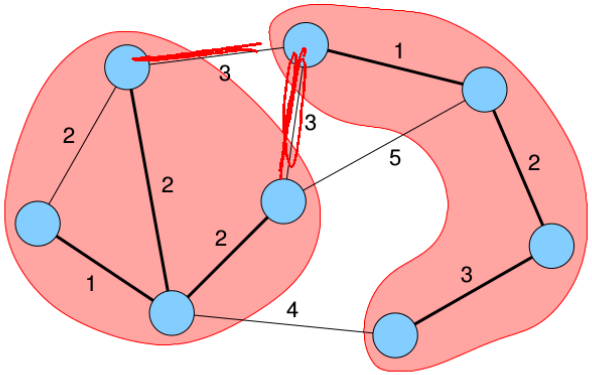


H. Seidl (TUM) GAD SS'16 534

Minimaler Spannbaum

Regel:

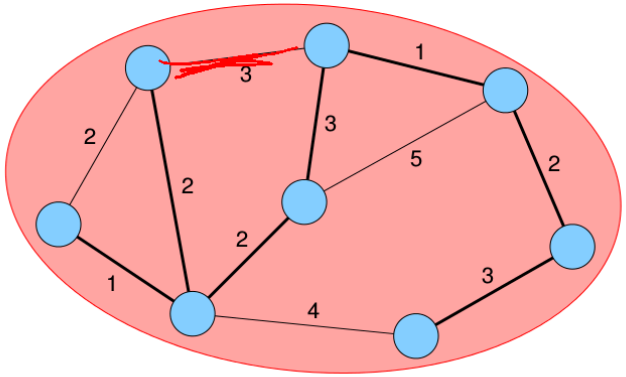
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

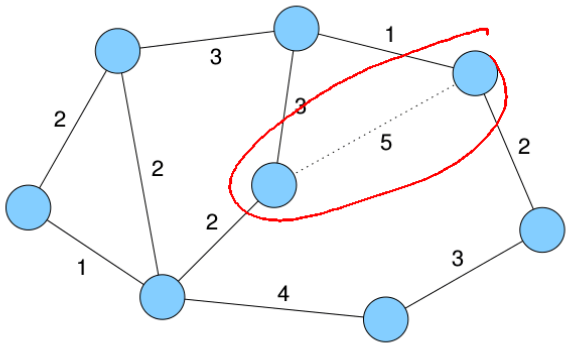
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

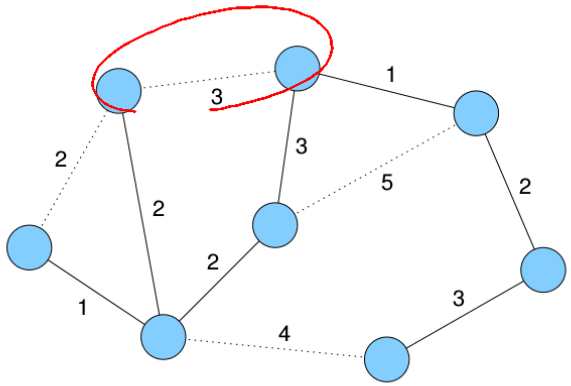
- lösche wiederholt Kante mit maximalen Kosten, so dass Zusammenhang nicht zerstört
- bis ein Baum übrig ist



Minimaler Spannbaum

Regel:

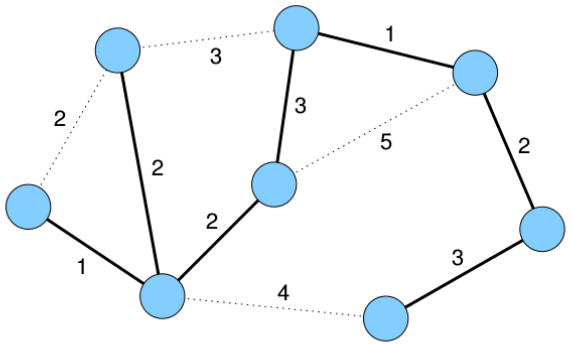
- lösche wiederholt Kante mit maximalen Kosten, so dass Zusammenhang nicht zerstört
- bis ein Baum übrig ist



Minimaler Spannbaum

Regel:

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



Minimaler Spannbaum

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)

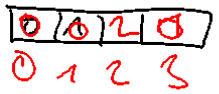
Algorithmus von Kruskal

```
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:

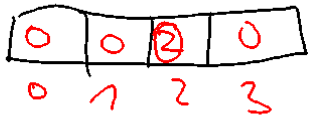
- Knoten seien nummeriert von 0 bis $n - 1$
- Array int parent[n], Einträge verweisen Richtung Repräsentant
- anfangs parent[i]=i für alle i

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
}
```

path compression



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 größeren 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) {
      new_size = size[ri] + size[rj];
      if (size[ri] < size[rj]) {
         parent[ri] = rj;
         size[rj] = new_size;
      } else {
         parent[rj] = ri;
         size[ri] = new_size;
      }
}
```

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: $O(m \log m)$ (Sortieren)

Algorithmus von Prim

Problem: Wie implementiert man die Regeln effizient?

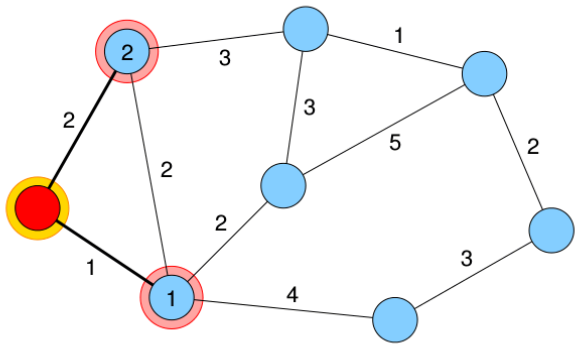
Alternative Strategie aus dem ersten Lemma:

- 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)

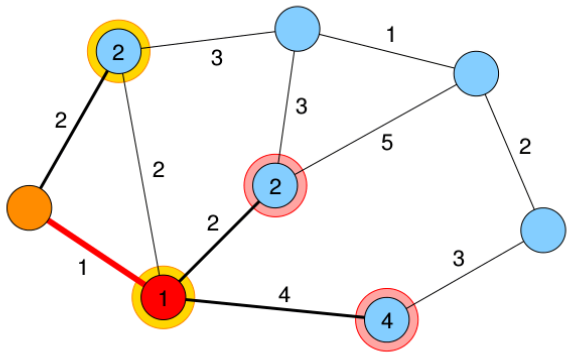
$\Rightarrow$ Baum umfasst jetzt 1 Knoten / Kante mehr

- wiederhole Auswahl bis alle n Knoten im Baum

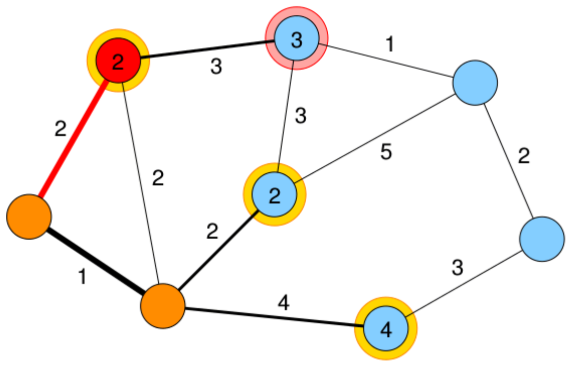
Algorithmus von Prim



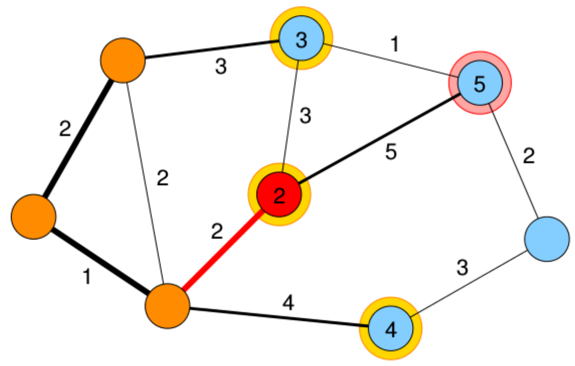
Algorithmus von Prim



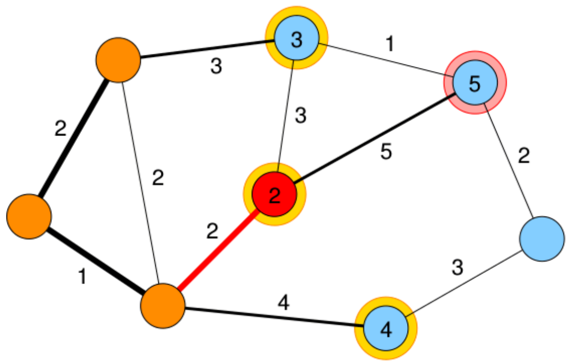
Algorithmus von Prim



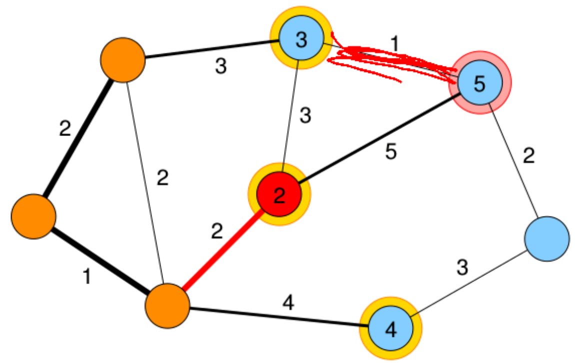
Algorithmus von Prim



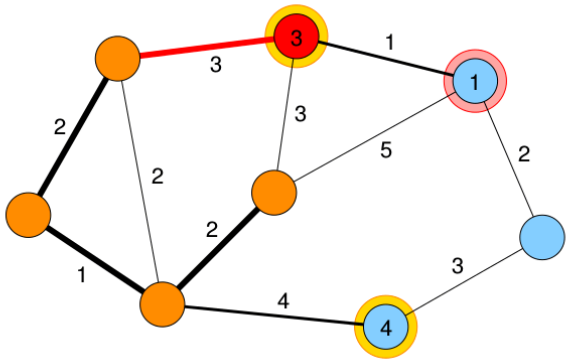
Algorithmus von Prim



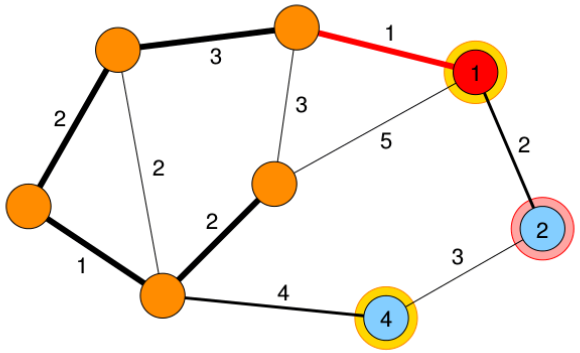
Algorithmus von Prim



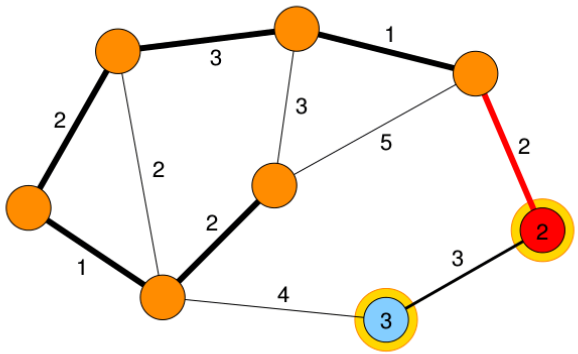
Algorithmus von Prim



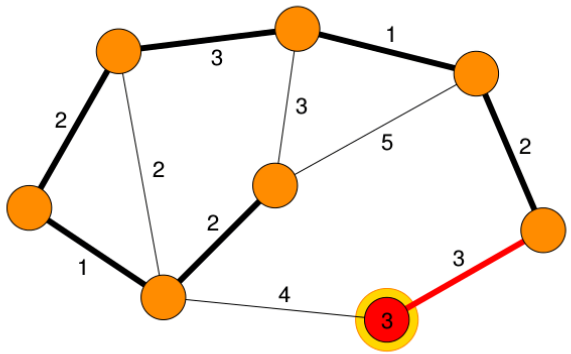
Algorithmus von Prim



Algorithmus von Prim



Algorithmus von Prim



Algorithmus Jarník-Prim: findet minimalen Spannbaum

Eingabe : $G = (V, E)$, $c : E \mapsto \mathbb{R}_+$, $s \in V$
Ausgabe : Minimaler Spannbaum 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 {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;
```


Jarník-Prim-Algorithmus

Laufzeit:

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

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: $O(m + n\log n)$

Jarník-Prim-Algorithmus

Laufzeit:

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

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: $O(m + n\log n)$

Alphabet, Wörter, Wortlänge, Wortmengen

Definition

Ein Alphabet Σ ist eine endliche Menge von Symbolen.

Wörter über Σ sind endliche Folgen von Symbolen aus Σ
(meist $w = w_0 \cdots w_{n-1}$ oder $w = w_1 \cdots w_n$).

Notation:

$\lvert w\rvert$	Länge des Wortes w (Anzahl der Zeichen in w)
ε	leeres Wort (Wort der Länge 0)
Σ^*	Menge aller Wörter über Σ
Σ^+	Menge aller Wörter der Länge ≥ 1 über Σ ($\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$)
Σ^k	Menge aller Wörter über Σ der Länge k

Präfix, Suffix, Teilwort

Definition

$[a : b] := \{n \in \mathbb{Z} \mid a \leq n \ \wedge \ n \leq b\}$ für $a, b \in \mathbb{Z}$

Sei $w = w_1 \cdots w_n$ ein Wort der Länge n über Σ, dann heißt

- w' Präfix von w, wenn $w' = w_1 \cdots w_\ell$ mit $\ell \in [0 : n]$
- w' Suffix von w, wenn $w' = w_\ell \cdots w_n$ mit $\ell \in [1 : n+1]$
- w' Teilwort von w, wenn $w' = w_i \cdots w_j$ mit $i, j \in [1 : n]$

Für $w' = w_i \cdots w_j$ mit $i > j$ soll gelten $w' = \varepsilon$.

Das leere Wort ε ist also Präfix, Suffix und Teilwort eines jeden Wortes.

Textsuche

Problem:

Gegeben: Text $t \in \Sigma^*$; $|t| = n$;
Suchwort $s \in \Sigma^*$; $|s| = m \leq n$

Gesucht: $\exists\, i \in [0 : n-m]$ mit $t_i \cdots t_{i+m-1} = s$?
(bzw. alle solchen Positionen i)

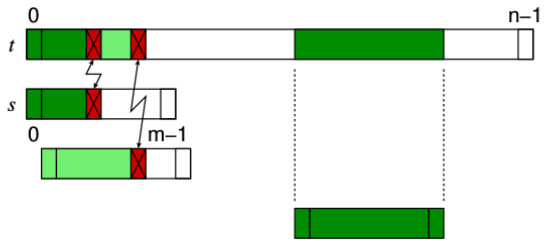
Textsuche

Problem:

Gegeben: Text $t \in \Sigma^*$; $|t| = n$;
Suchwort $s \in \Sigma^*$; $|s| = m \leq n$

Gesucht: $\exists\, i \in [0 : n-m]$ mit $t_i \cdots t_{i+m-1} = s$?
(bzw. alle solchen Positionen i)

Naiver Algorithmus



- Suchwort s Zeichen für Zeichen mit Text t vergleichen
- wenn zwei Zeichen nicht übereinstimmen (Mismatch), dann s um eine Position „nach rechts“ schieben und erneut s mit t vergleichen
- Vorgang wiederholen, bis s in t gefunden wird oder bis klar ist, dass s in t nicht enthalten ist

Naiver Algorithmus: Beispiele

```
t = a a a a a a a a a a
    a a a b
      a a a b
        a a a b
          a a a b
```


Naiver Algorithmus: Beispiele

```
t = a a a a a a a a a a
    a a a b
      a a a b
        a a a b
          a a a b
```


Naiver Algorithmus: Beispiele

```
t = a a a a a a a a a a
    a a a b
      a a a b
        a a a b
          a a a b
```

```
t = a a b a a b a a b a a b a a b
    a a b a a a
      a a b a a a
        a a b a a a
          a a b a a a
```


Naiver Algorithmus: Beispiele

```
t = a a a a a a a a a a
    a a a b
      a a a b
        a a a b
          a a a b
```