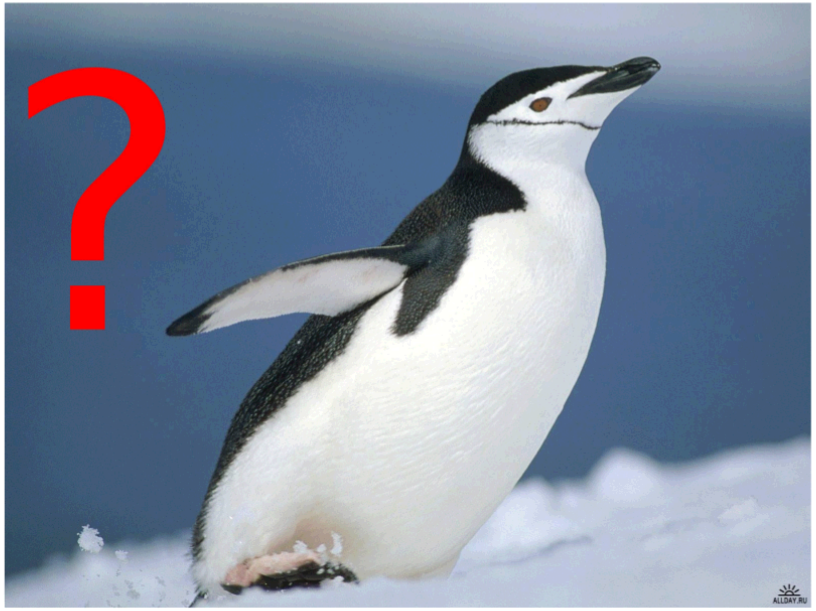
Script **generated by TTT**

Title: Seidl: GAD (14.07.2015)

Date: Tue Jul 14 13:48:17 CEST 2015

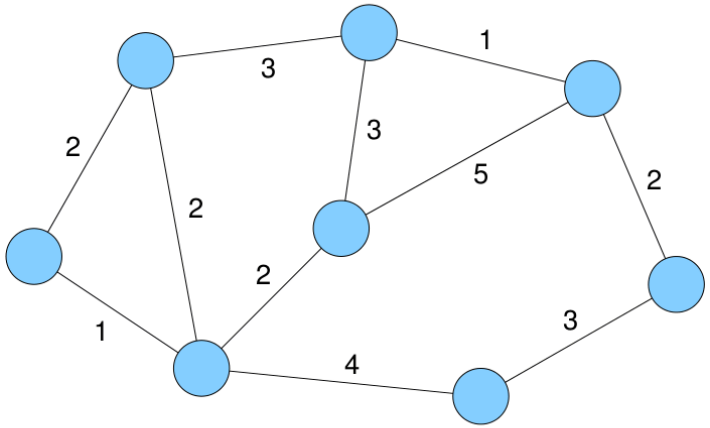
Duration: 147:32 min

Pages: 105



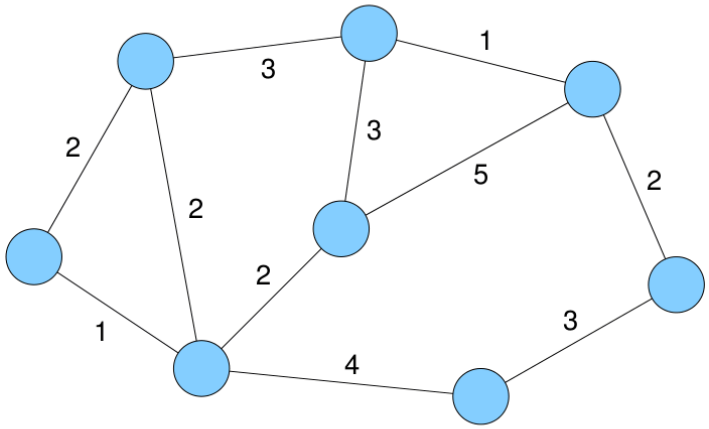
Minimaler Spannbaum

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



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)

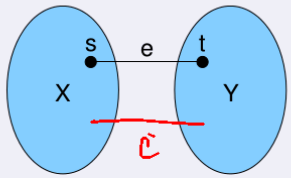
Minimaler Spannbaum

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.



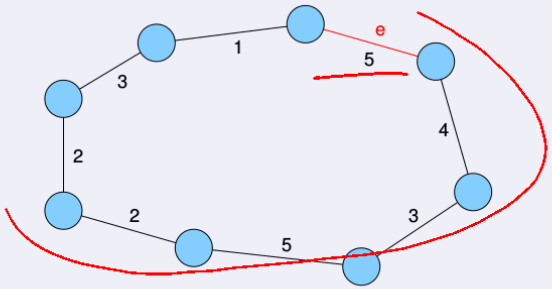
Minimaler Spannbaum

Lemma

Betrachte

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

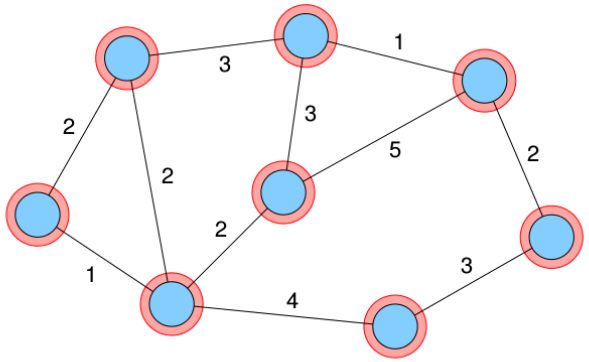
Dann ist jeder MSB in G ohne e auch ein MSB in G



Minimaler Spannbaum

Regel:

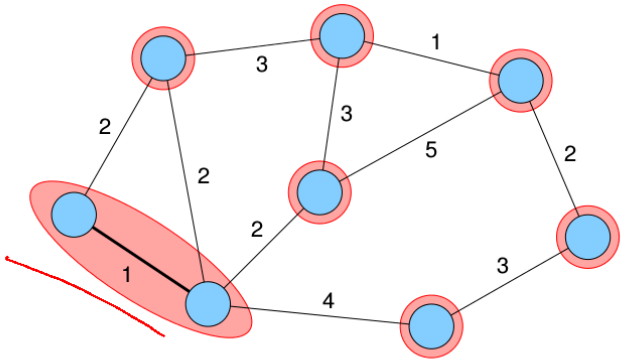
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

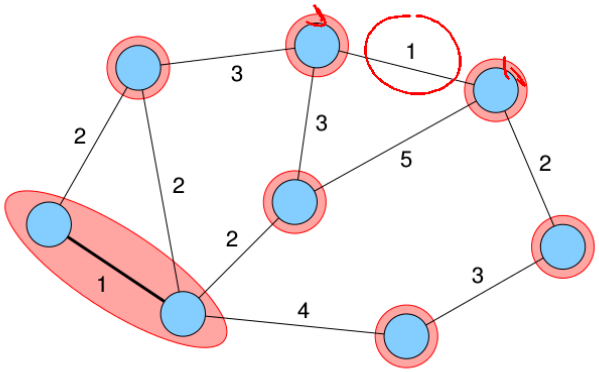
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

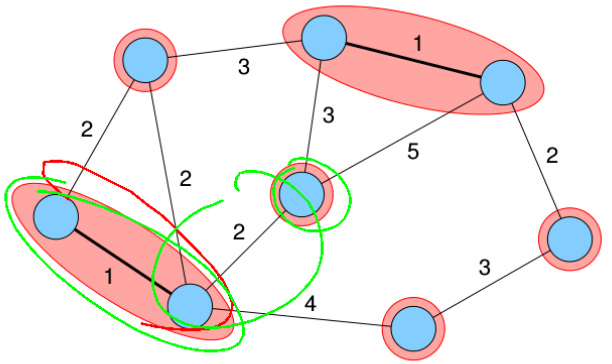
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

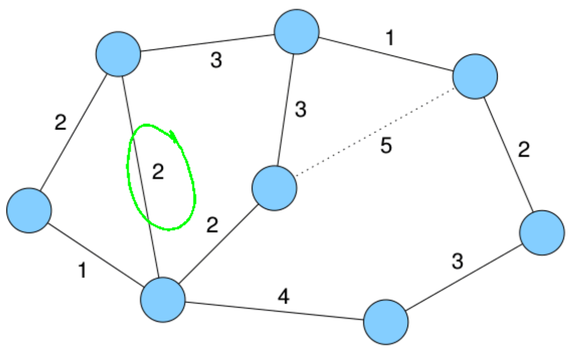
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

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

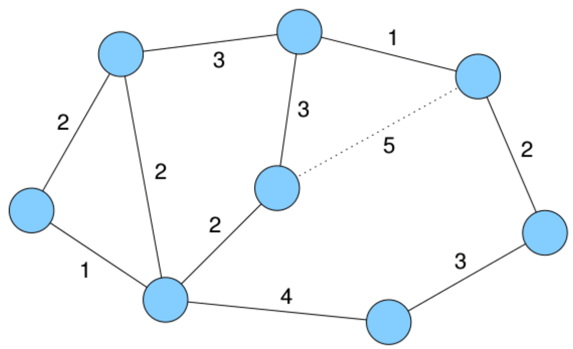


Graphen | Minimale Spannbäume

Minimaler Spannbaum

Regel:

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

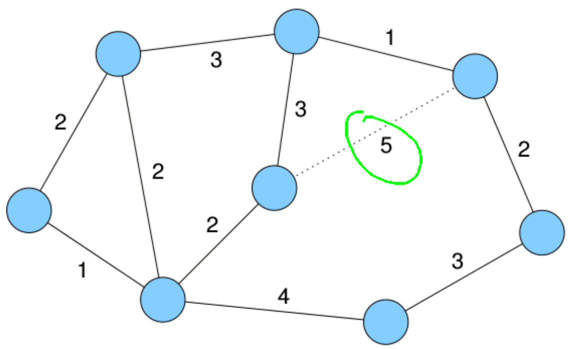


Graphen | Minimale Spannbäume

Minimaler Spannbaum

Regel:

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

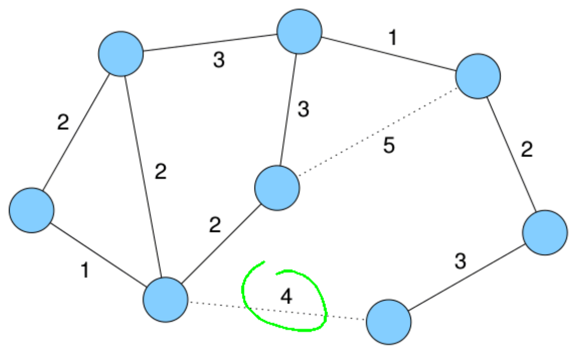


Graphen | Minimale Spannbäume

Minimaler Spannbaum

Regel:

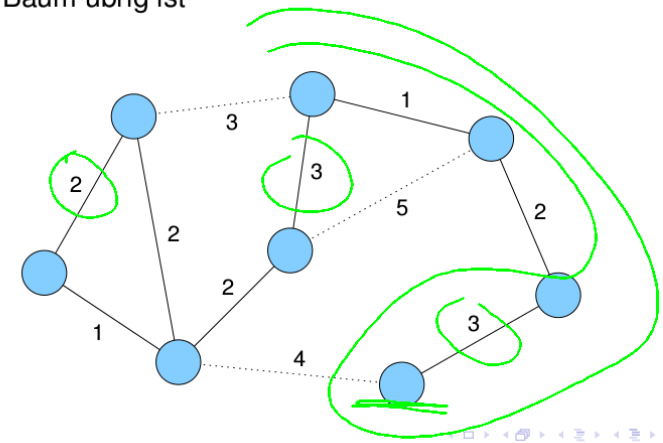
- lösche wiederholt Kante mit maximalen Kosten, so dass Zusammenhang nicht zerstört
- bis ein Baum übrig ist



Minimaler Spannbaum

Regel:

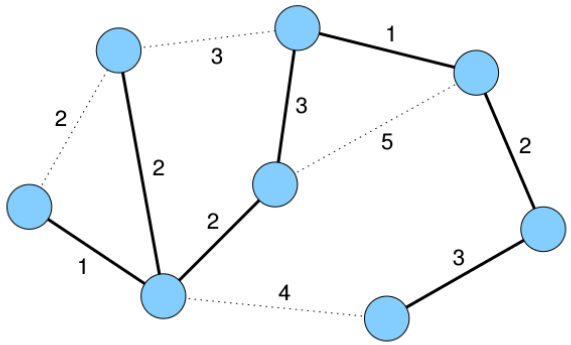
- lösche wiederholt Kante mit maximalen Kosten, so dass Zusammenhang nicht zerstört
- bis ein Baum übrig ist



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)

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

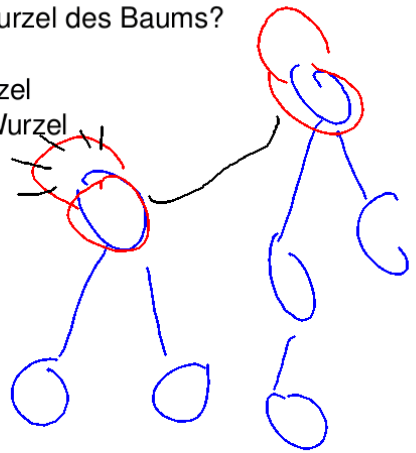
Anwendung:

- 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
}
```



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 $O(\log n)$

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
}
```


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
}
```

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 $O(\log n)$

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
}
```


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 $O(\log n)$

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
}
```

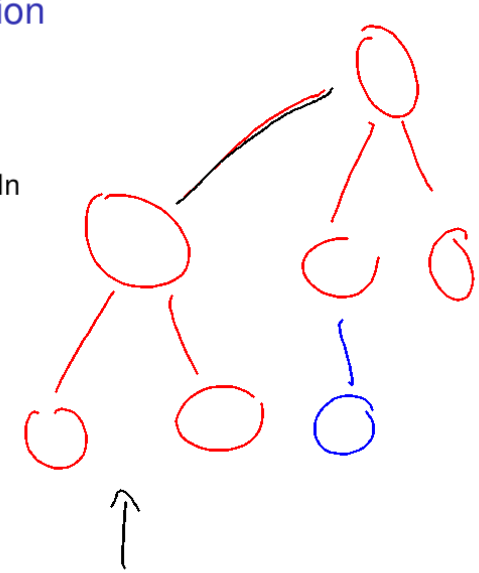
Gewichtete union-Operation mit Pfadkompression

- 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[rj])
      parent[ri] = rj;
    else {
      parent[rj] = ri;
      if (height[ri] == height[rj])
        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: $O(m \log m)$ (Sortieren)

Graphen Minimale Spannbäume

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)

H. Seidl (TUM) GAD SS'15 596

Algorithmus von Prim

Problem: Wie implementiert man die Regeln effizient?

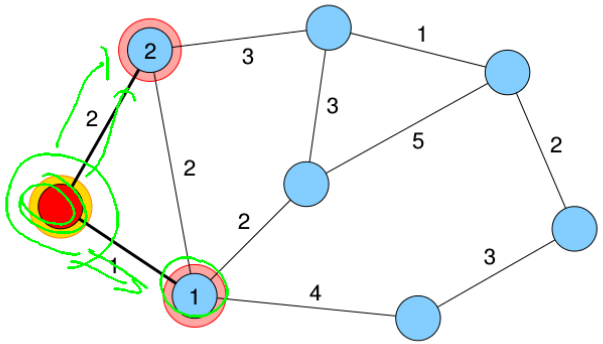
Alternative Strategie aus dem ersten Lemma:

- 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)

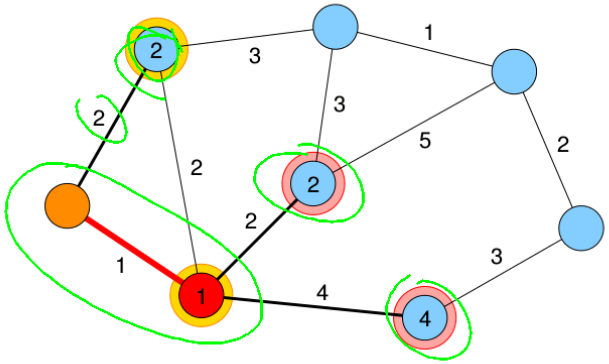
$\Rightarrow$ Baum umfasst jetzt 1 Knoten / Kante mehr

- wiederhole Auswahl bis alle n Knoten im Baum

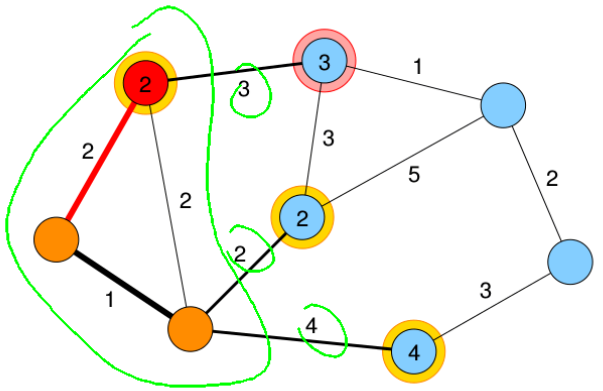
Algorithmus von Prim



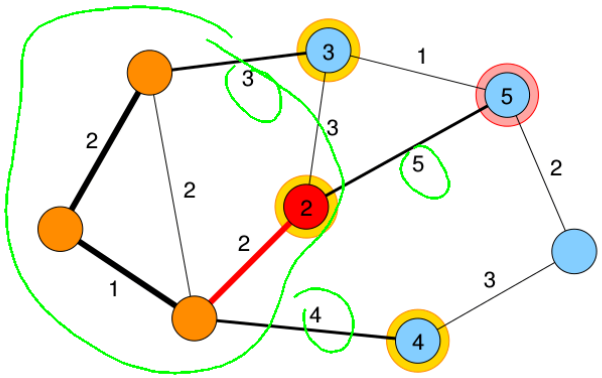
Algorithmus von Prim



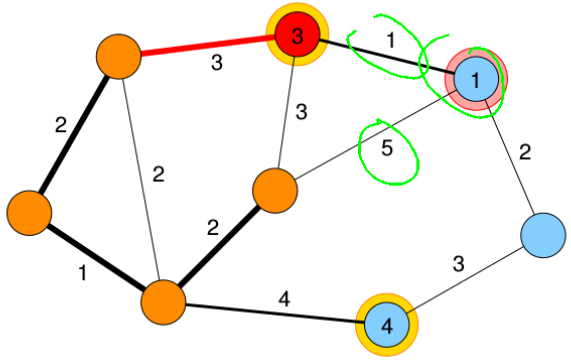
Algorithmus von Prim



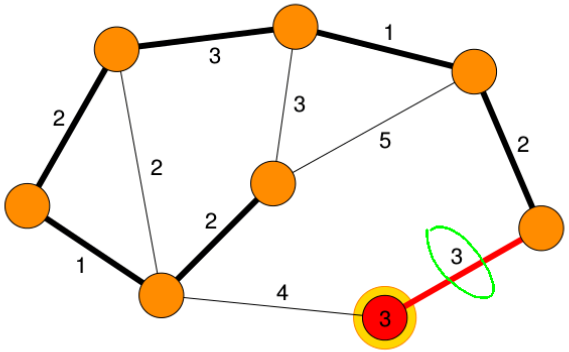
Algorithmus von Prim



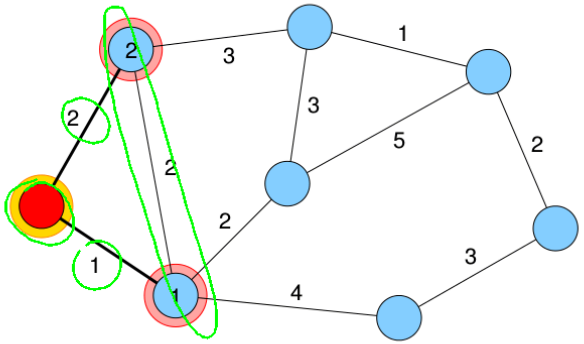
Algorithmus von Prim



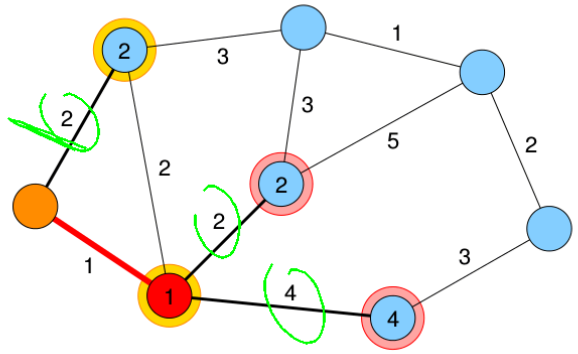
Algorithmus von Prim



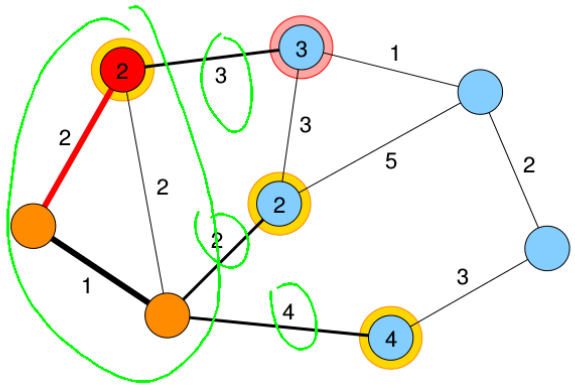
Algorithmus von Prim



Algorithmus von Prim



Algorithmus von Prim

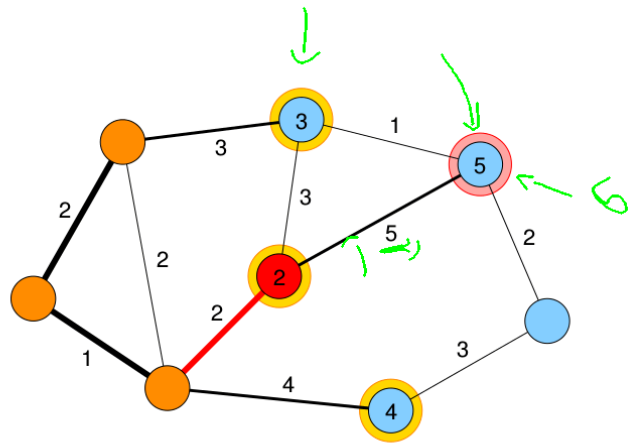


Algorithmus Jarník-Prim: findet minimalen Spannbaum

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 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;
```


Algorithmus von Prim

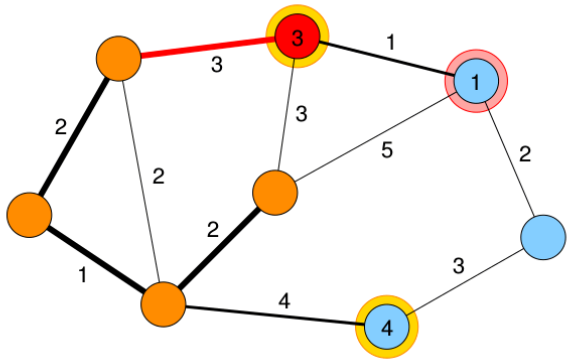


Algorithmus Jarník-Prim: findet minimalen Spannbaum

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 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;
```


Algorithmus von Prim



Graphen | Minimale Spannbäume

Algorithmus Jarník-Prim: findet minimalen Spannbaum

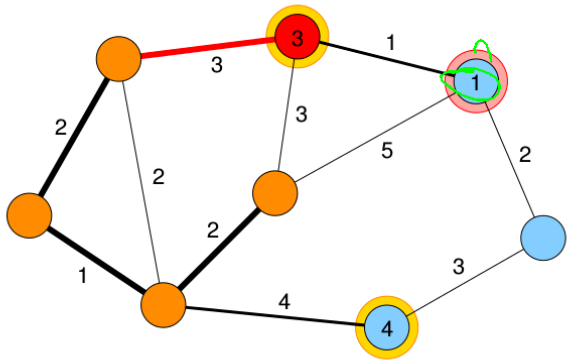
Eingabe : $G = (V, E)$, $c : E \mapsto \mathbb{R}_+$, $s \in V$
Ausgabe : Minimaler Spannbaum in Array *pred*

```
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 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;
```

H. Seidl (TUM) GAD SS'15 607

Graphen | Minimale Spannbäume

Algorithmus von Prim



H. Seidl (TUM) GAD SS'15 602

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 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;
```


Graphen | Minimale Spannbäume

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 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;
```

H. Seidl (TUM) GAD SS'15 607

Graphen | Minimale Spannbäume

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

Graphen | Minimale Spannbäume

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 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;
```

H. Seidl (TUM) GAD SS'15 607

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

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 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;
```


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

Übersicht

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:

- $|w|$ 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

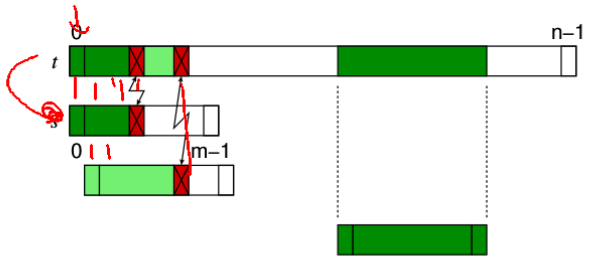
Problem:

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)

Übersicht

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
```


Naiver Algorithmus: Beispiele

```
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 b a a a
          a a b a a a
```


Naiver Algorithmus: Implementation

Funktion NaiveSearch(char $t[\,]$, int n, char $s[\,]$, int m)

```
int i := 0, j := 0;
while (i ≤ n − m) do
    while (t[i + j] = s[j]) do
        j++;
        if (j = m) then
            return TRUE;
    i++;
    j := 0;
return FALSE;
```

Analyse des naiven Algorithmus

- zähle Vergleiche von Zeichen,
- äußere Schleife wird $(n - m + 1)$-mal durchlaufen,
- die innere Schleife wird maximal m-mal durchlaufen.
- maximale Anzahl von Vergleichen: $(n - m + 1)m$,
- Laufzeit: $O(nm)$

Übersicht

Bessere Idee

- frühere erfolgreiche Zeichenvergleiche ausnutzen

- Idee:

  Suchwort so weit nach rechts verschieben, dass in dem Bereich von t, in dem bereits beim vorherigen Versuch erfolgreiche Zeichenvergleiche durchgeführt wurden, nun nach dem Verschieben auch wieder die Zeichen in diesem Bereich übereinstimmen

Rand und eigentlicher Rand

Definition

Ein Wort r heißt Rand eines Wortes w, wenn r Präfix und Suffix von w ist. (Für jedes Wort w ist das leere Wort ε ein Rand von w, genau wie w selbst.)

Ein Rand r eines Wortes w heißt eigentlicher Rand, wenn $r \neq w$ und wenn es außer w selbst keinen längeren Rand gibt.

Rand und eigentlicher Rand

Beispiel

Das Wort $w =$ aabaabaa besitzt folgende Ränder:

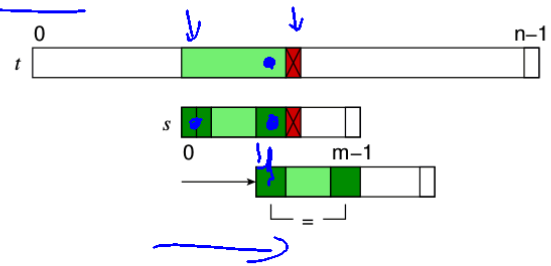
- ε
- a
- aa
- aabaa
- aabaabaa$= w$.

Der eigentliche Rand ist aabaa.

Beachte: bei der Darstellung des Rands im Wort können sich Präfix und Suffix in der Wortmitte überlappen.

Shift-Idee

- Pattern s so verschieben, dass im bereits gematchten Bereich wieder Übereinstimmung herrscht.
- Dazu müssen überlappendes Präfix und Suffix dieses Bereichs übereinstimmen.



Shifts und sichere Shifts

Definition

Eine Verschiebung der Anfangsposition i des zu suchenden Wortes (also eine Indexerhöhung $i \to i'$) heißt Shift.

Ein Shift von $i \to i'$ heißt sicher, wenn s nicht als Teilwort von t an der Position $k \in [i+1 : i'-1]$ vorkommt, d.h., $s \neq t_k \cdots t_{k+m-1}$ für alle $k \in [i+1 : i'-1]$.

- Sinn eines sicheren Shifts:
  dass man beim Verschieben des Suchworts kein eventuell vorhandenes Vorkommen von s in t überspringt

Shifts und sichere Shifts

Definition

Eine Verschiebung der Anfangsposition i des zu suchenden Wortes (also eine Indexerhöhung $i \to i'$) heißt Shift.

Ein Shift von $i \to i'$ heißt sicher, wenn s nicht als Teilwort von t an der Position $k \in [i+1 : i'-1]$ vorkommt, d.h., $s \neq t_k \cdots t_{k+m-1}$ für alle $k \in [i+1 : i'-1]$.

- Sinn eines sicheren Shifts:
  dass man beim Verschieben des Suchworts kein eventuell vorhandenes Vorkommen von s in t überspringt

Sichere Shifts

Definition

Sei $\partial(s)$ der eigentliche Rand von s und sei

$$border[j] = \begin{cases} -1 & \text{für } j = 0 \\ |\partial(s_0 \cdots s_{j-1})| & \text{für } j \geq 1 \end{cases}$$

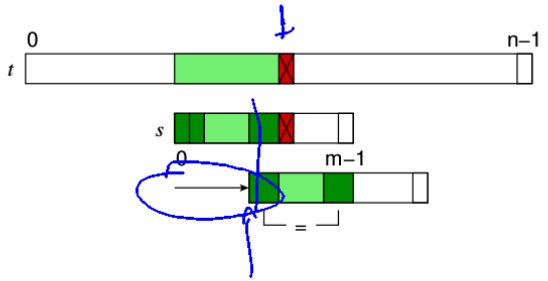
die Länge des eigentlichen Rands des Präfixes der Länge j.

Lemma

Ist das Präfix der Länge j gematcht (also gilt $s_k = t_{i+k}$ für alle $k \in [0 : j-1]$) und haben wir ein Mismatch an der nächsten Position j ($s_j \neq t_{i+j}$), dann ist der Shift $i \to i + j - border[j]$ sicher.

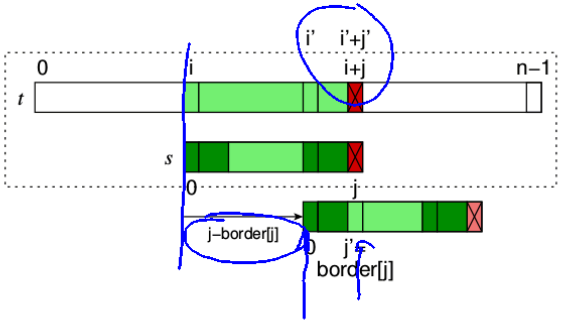
Shift-Idee

- Pattern s so verschieben, dass im bereits gematchten Bereich wieder Übereinstimmung herrscht.
- Dazu müssen überlappendes Präfix und Suffix dieses Bereichs übereinstimmen.



Sichere Shifts

Shift um $j - border[j]$



Sichere Shifts

Beweis.

- (siehe Skizze)

$$\begin{array}{rcl} s_0 \cdots s_{j-1} & = & t_i \cdots t_{i+j-1}, \\ s_j & \neq & t_{i+j} \end{array}$$

- Der eigentliche Rand von $s_0 \cdots s_{j-1}$ hat die Länge $\mathrm{border}[j]$.
- Verschiebt man s um $j - \mathrm{border}[j]$ nach rechts, so liegt der linke Rand von $s_0 \cdots s_{j-1}$ nun genau da, wo vorher der rechte Rand lag, d.h. im Präfix/Suffix-Überlappungsbereich besteht Übereinstimmung zwischen Präfix, Suffix und Text.
- Da es keinen längeren Rand von $s_0 \cdots s_{j-1}$ als diesen gibt (außer $s_0 \cdots s_{j-1}$ selbst), ist dieser Shift sicher.

□

Sichere Shifts

Beweis.

- (siehe Skizze)

$$\begin{aligned} s_0 \cdots s_{j-1} &= t_i \cdots t_{i+j-1}, \\ s_j &\neq t_{i+j} \end{aligned}$$

- Der eigentliche Rand von $s_0 \cdots s_{j-1}$ hat die Länge $\text{border}[j]$.
- Verschiebt man s um $j - \text{border}[j]$ nach rechts, so liegt der linke Rand von $s_0 \cdots s_{j-1}$ nun genau da, wo vorher der rechte Rand lag, d.h. im Präfix/Suffix-Überlappungsbereich besteht Übereinstimmung zwischen Präfix, Suffix und Text.
- Da es keinen längeren Rand von $s_0 \cdots s_{j-1}$ als diesen gibt (außer $s_0 \cdots s_{j-1}$ selbst), ist dieser Shift sicher.

□

Pattern Matching | Knuth-Morris-Pratt-Algorithmus

Sichere Shifts

Beweis.

- (siehe Skizze)

$$\begin{aligned} s_0 \cdots s_{j-1} &= t_i \cdots t_{i+j-1}, \\ s_j &\neq t_{i+j} \end{aligned}$$

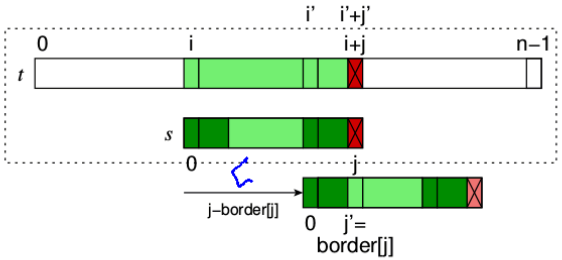
- Der eigentliche Rand von $s_0 \cdots s_{j-1}$ hat die Länge $\text{border}[j]$.
- Verschiebt man s um $j - \text{border}[j]$ nach rechts, so liegt der linke Rand von $s_0 \cdots s_{j-1}$ nun genau da, wo vorher der rechte Rand lag, d.h. im Präfix/Suffix-Überlappungsbereich besteht Übereinstimmung zwischen Präfix, Suffix und Text.
- Da es keinen längeren Rand von $s_0 \cdots s_{j-1}$ als diesen gibt (außer $s_0 \cdots s_{j-1}$ selbst), ist dieser Shift sicher.

□

H. Seidl (TUM) GAD SS'15 626

Sichere Shifts

Shift um $j - \text{border}[j]$



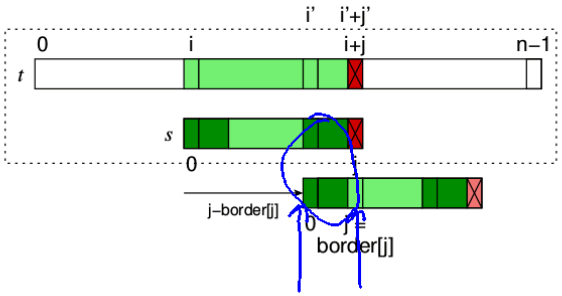
KMP-Algorithmus

Funktion KMP($t[\,]$, n, $s[\,]$, m)

```
int border[m + 1];
computeBorders(border, m, s);
int i := 0, j := 0;
while i ≤ n − m do
    while t[i + j] = s[j] do
        j++;
        if j = m then
            return TRUE;
    i := i + (j − border[j]) ;          // Es gilt j − border[j] > 0
    j := max{0, border[j]};
return FALSE;
```

Sichere Shifts

Shift um j – border[j]



KMP-Algorithmus

Funktion KMP($t[\,]$, n, $s[\,]$, m)

int $border[m+1]$;
computeBorders($border$, m, s);
int $i := 0$, $j := 0$;
while $i \leq n - m$ **do**
  while $t[i+j] = s[j]$ **do**
    j++;
    if $j = m$ **then**
      return TRUE;
  $i := i + (j - border[j])$; // Es gilt $j - border[j] > 0$
  $j := \max\{0, border[j]\}$;
return FALSE;

Laufzeit des KMP-Algorithmus: erfolglose Vergleiche

Nach erfolglosem Vergleich (Mismatch) wird $(i + j)$ nie kleiner:

- Seien dazu i und j die Werte vor einem erfolglosen Vergleich und i' und j' die Werte nach einem erfolglosen Vergleich.
- Wert vor dem Vergleich: $i + j$
- Wert nach dem Vergleich:
  $i' + j' = (i + j - border[j]) + (max\{0, border[j]\})$.
- Fallunterscheidung: $border[j]$ negativ oder nicht.
  - $border[j] < 0$, also $border[j] = -1$, dann muss $j = 0$ sein.
    Das bedeutet $i' + j' = i' + 0 = (i + 0 - (-1)) + 0 = i + 1$.
  - $border[j] \geq 0$, dann gilt $i' + j' = i + j$
- Also wird $i + j$ nach einem erfolglosen Vergleich nicht kleiner.

Laufzeit des KMP-Algorithmus: erfolglose Vergleiche

Nach erfolglosem Vergleich (Mismatch) wird $(i + j)$ nie kleiner:

- Seien dazu i und j die Werte vor einem erfolglosen Vergleich und i' und j' die Werte nach einem erfolglosen Vergleich.
- Wert vor dem Vergleich: $i + j$
- Wert nach dem Vergleich:
  $i' + j' = (i + j - border[j]) + (max\{0, border[j]\})$.
- Fallunterscheidung: $border[j]$ negativ oder nicht.
  - $border[j] < 0$, also $border[j] = -1$, dann muss $j = 0$ sein.
    Das bedeutet $i' + j' = i' + 0 = (i + 0 - (-1)) + 0 = i + 1$.
  - $border[j] \geq 0$, dann gilt $i' + j' = i + j$
- Also wird $i + j$ nach einem erfolglosen Vergleich nicht kleiner.

Laufzeit des KMP-Algorithmus

- Nach jedem erfolglosen Vergleich wird $i \in [0 : n - m]$ erhöht.
- i wird nie verkleinert.

$\Rightarrow$ maximal $n - m + 1$ erfolglose Vergleiche

- Nach einem erfolgreichen Vergleich wird $i + j$ um 1 erhöht.
- maximal n erfolgreiche Vergleiche, da $i + j \in [0 : n - 1]$.

- insgesamt maximal $2n - m + 1$ Vergleiche

Pattern Matching | Knuth-Morris-Pratt-Algorithmus

Laufzeit des KMP-Algorithmus

- Nach jedem erfolglosen Vergleich wird $i \in [0 : n - m]$ erhöht.
- i wird nie verkleinert.

$\Rightarrow$ maximal $n - m + 1$ erfolglose Vergleiche

- Nach einem erfolgreichen Vergleich wird $i + j$ um 1 erhöht.
- maximal n erfolgreiche Vergleiche, da $i + j \in [0 : n - 1]$.

- insgesamt maximal $2n - m + 1$ Vergleiche

H. Seidl (TUM) GAD SS'15 629

Berechnung der border-Tabelle

- *border*[ ]-Tabelle:
  speichert für jedes Präfix $s_0 \cdots s_{j-1}$ der Länge $j \in \{0 \ldots m\}$ von Suchstring s die Größe des eigentlichen Rands

- Initialisierung: $border[0] = -1$ und $border[1] = 0$

- Annahme: $border[0], \ldots, border[j-1]$ sind schon berechnet

- Ziel: Berechnung von $border[j]$
  (Länge des eigentlichen Randes von Präfix der Länge j)

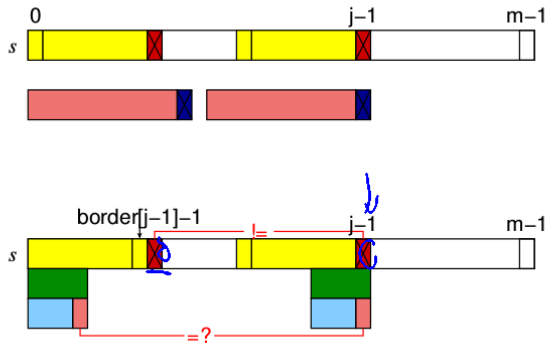
Pattern Matching | Knuth-Morris-Pratt-Algorithmus

Berechnung der border-Tabelle

- Der eigentliche Rand $s_0 \cdots s_k$ von $s_0 \cdots s_{j-1}$ kann um maximal ein Zeichen länger sein als der eigentliche Rand von $s_0 \cdots s_{j-2}$, denn $s_0 \cdots s_{k-1}$ ist auch ein Rand von $s_0 \cdots s_{j-2}$ (oberer Teil der Abbildung).
- Ist $s_{border[j-1]} = s_{j-1}$, so ist $border[j] = border[j-1] + 1$.
- Andernfalls müssen wir ein kürzeres Präfix von $s_0 \cdots s_{j-2}$ finden, das auch ein Suffix von $s_0 \cdots s_{j-2}$ ist.
- Der nächstkürzere Rand eines Wortes ist offensichtlich der eigentliche Rand des zuletzt betrachteten Randes dieses Wortes.
- Nach Konstruktion der Tabelle *border* ist das nächstkürzere Präfix mit dieser Eigenschaft das der Länge $border[border[j-1]]$.

H. Seidl (TUM) GAD SS'15 632

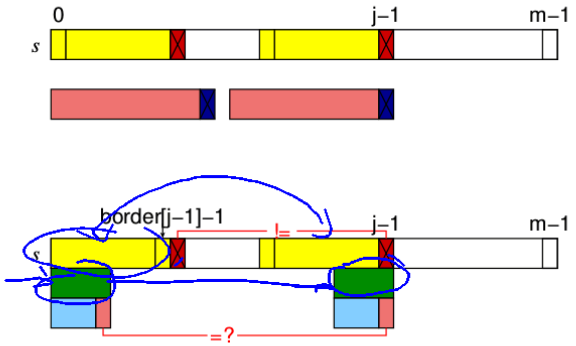
Berechnung der border-Tabelle



Berechnung der border-Tabelle

- Der eigentliche Rand $s_0 \cdots s_k$ von $s_0 \cdots s_{j-1}$ kann um maximal ein Zeichen länger sein als der eigentliche Rand von $s_0 \cdots s_{j-2}$, denn $s_0 \cdots s_{k-1}$ ist auch ein Rand von $s_0 \cdots s_{j-2}$ (oberer Teil der Abbildung).
- Ist $s_{border[j-1]} = s_{j-1}$, so ist $border[j] = border[j-1] + 1$.
- Andernfalls müssen wir ein kürzeres Präfix von $s_0 \cdots s_{j-2}$ finden, das auch ein Suffix von $s_0 \cdots s_{j-2}$ ist.
- Der nächstkürzere Rand eines Wortes ist offensichtlich der eigentliche Rand des zuletzt betrachteten Randes dieses Wortes.
- Nach Konstruktion der Tabelle *border* ist das nächstkürzere Präfix mit dieser Eigenschaft das der Länge $border[border[j-1]]$.

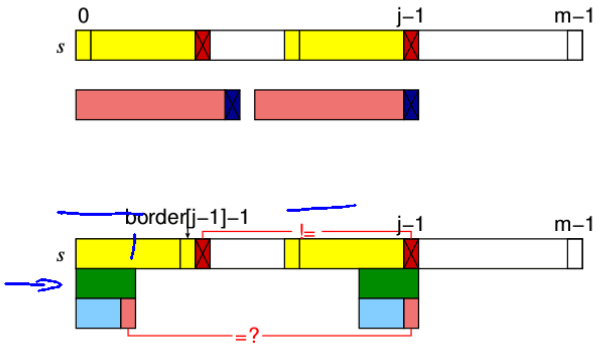
Berechnung der border-Tabelle



Berechnung der border-Tabelle

- Der eigentliche Rand $s_0 \cdots s_k$ von $s_0 \cdots s_{j-1}$ kann um maximal ein Zeichen länger sein als der eigentliche Rand von $s_0 \cdots s_{j-2}$, denn $s_0 \cdots s_{k-1}$ ist auch ein Rand von $s_0 \cdots s_{j-2}$ (oberer Teil der Abbildung).
- Ist $s_{border[j-1]} = s_{j-1}$, so ist $border[j] = border[j-1] + 1$.
- Andernfalls müssen wir ein kürzeres Präfix von $s_0 \cdots s_{j-2}$ finden, das auch ein Suffix von $s_0 \cdots s_{j-2}$ ist.
- Der nächstkürzere Rand eines Wortes ist offensichtlich der eigentliche Rand des zuletzt betrachteten Randes dieses Wortes.
- Nach Konstruktion der Tabelle *border* ist das nächstkürzere Präfix mit dieser Eigenschaft das der Länge $border[border[j-1]]$.

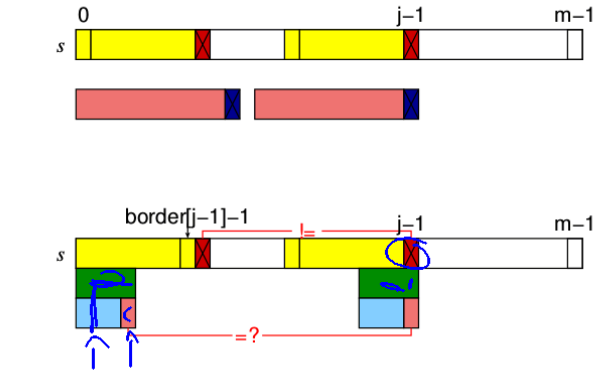
Berechnung der border-Tabelle



Berechnung der border-Tabelle

- Der eigentliche Rand $s_0 \cdots s_k$ von $s_0 \cdots s_{j-1}$ kann um maximal ein Zeichen länger sein als der eigentliche Rand von $s_0 \cdots s_{j-2}$, denn $s_0 \cdots s_{k-1}$ ist auch ein Rand von $s_0 \cdots s_{j-2}$ (oberer Teil der Abbildung).
- Ist $s_{border[j-1]} = s_{j-1}$, so ist $border[j] = border[j-1] + 1$.
- Andernfalls müssen wir ein kürzeres Präfix von $s_0 \cdots s_{j-2}$ finden, das auch ein Suffix von $s_0 \cdots s_{j-2}$ ist.
- Der nächstkürzere Rand eines Wortes ist offensichtlich der eigentliche Rand des zuletzt betrachteten Randes dieses Wortes.
- Nach Konstruktion der Tabelle *border* ist das nächstkürzere Präfix mit dieser Eigenschaft das der Länge $border[border[j-1]]$.

Berechnung der border-Tabelle



Berechnung der border-Tabelle

- teste nun, ob sich dieser Rand von $s_0 \cdots s_{j-2}$ zu einem eigentlichen Rand von $s_0 \cdots s_{j-1}$ erweitern lässt
- solange wiederholen, bis wir einen Rand gefunden haben, der sich zu einem Rand von $s_0 \cdots s_{j-1}$ erweitern lässt
- Falls sich kein Rand von $s_0 \cdots s_{j-2}$ zu einem Rand von $s_0 \cdots s_{j-1}$ erweitern lässt, so ist der eigentliche Rand von $s_0 \cdots s_{j-1}$ das leere Wort und wir setzen $border[j] = 0$.

Algorithmus zur Berechnung der border-Tabelle

Prozedur computeBorders(int *border*[ ], int *m*, char *s*[ ])

```
border[0] := −1;
border[1] := 0;
int i := 0;
for (int j := 2; j ≤ m; j++) do
    // Hier gilt: i = border[j − 1]
    while (i ≥ 0) && (s[i] ≠ s[j − 1]) do
        i := border[i];
    i++;
    border[j] := i;
```


Pattern Matching | Knuth-Morris-Pratt-Algorithmus

Algorithmus zur Berechnung der border-Tabelle

Prozedur computeBorders(int *border*[ ], int *m*, char *s*[ ])

```
border[0] := −1;
border[1] := 0;
int i := 0;
for (int j := 2; j ≤ m; j++) do
    // Hier gilt: i = border[j − 1]
    while (i ≥ 0) && (s[i] ≠ s[j − 1]) do
        i := border[i];
    i++;
    border[j] := i;
```

H. Seidl (TUM) GAD SS'15 634

Pattern Matching | Knuth-Morris-Pratt-Algorithmus

Laufzeit der Berechnung der border-Tabelle

- maximal $m-1$ erfolgreiche Vergleiche, da jedes Mal $j \in [2:m]$ um 1 erhöht und nie erniedrigt wird
- Betrachte für die Anzahl erfolgloser Vergleiche den Wert i.
  Zu Beginn ist $i = 0$.
- i wird genau $(m-1)$ Mal um 1 erhöht, da die for-Schleife $(m-1)$ Mal durchlaufen wird.
- Bei einem erfolglosen Vergleich wird i um mindestens 1 erniedrigt.
- i kann maximal $(m-1)+1 = m$ Mal erniedrigt werden, da immer $i \geq -1$ gilt. Es gibt also höchstens m erfolglose Vergleiche.
- Gesamtzahl der Vergleiche $\leq 2m-1$

Pattern Matching | Knuth-Morris-Pratt-Algorithmus

Laufzeit der Berechnung der border-Tabelle

- maximal $m-1$ erfolgreiche Vergleiche, da jedes Mal $j \in [2:m]$ um 1 erhöht und nie erniedrigt wird
- Betrachte für die Anzahl erfolgloser Vergleiche den Wert i.
  Zu Beginn ist $i = 0$.
- i wird genau $(m-1)$ Mal um 1 erhöht, da die for-Schleife $(m-1)$ Mal durchlaufen wird.
- Bei einem erfolglosen Vergleich wird i um mindestens 1 erniedrigt.
- i kann maximal $(m-1)+1 = m$ Mal erniedrigt werden, da immer $i \geq -1$ gilt. Es gibt also höchstens m erfolglose Vergleiche.
- Gesamtzahl der Vergleiche $\leq 2m-1$

H. Seidl (TUM) GAD SS'15 635

Laufzeit des KMP-Algorithmus

Theorem

Der Algorithmus von Knuth, Morris und Pratt benötigt maximal $2n + m$ Vergleiche, um festzustellen, ob ein Muster s der Länge m in einem Text t der Länge n enthalten ist.

Der Algorithmus lässt sich leicht derart modifizieren, dass er alle Positionen der Vorkommen von s in t ausgibt, ohne dabei die asymptotische Laufzeit zu erhöhen.

Donald E. Knuth, James H. Morris, Jr. and Vaughan R. Pratt
Fast Pattern Matching in Strings
SIAM Journal on Computing 6(2):323–350, 1977.

Übersicht