**Script** generated by TTT

Title: Täubig: GAD (08.07.2014)

Date: Tue Jul 08 13:52:40 CEST 2014

Duration: 117:01 min

Pages: 74



## Monotone Priority Queues

Beobachtung:

- aktuelles Distanz-Minimum der verbleibenden Knoten ist beim Dijkstra-Algorithmus monoton wachsend

Monotone Priority Queue

- Folge der entnommenen Elemente hat monoton steigende Werte
- effizientere Implementierung möglich, falls Kantengewichte ganzzahlig

Annahme: alle Kantengewichte im Bereich $[0, C]$

Konsequenz für Dijkstra-Algorithmus:

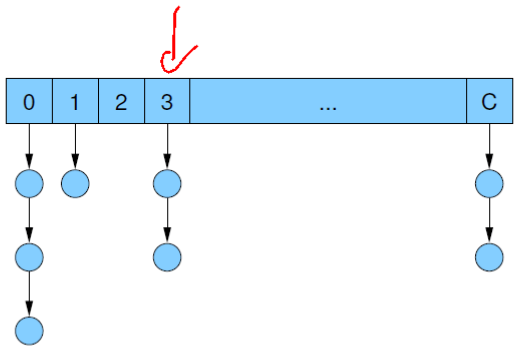
$\Rightarrow$ enthaltene Distanzwerte immer im Bereich $[d, d + C]$





## Bucket Queue

- Array B aus $C + 1$ Listen
- Variable $d_{min}$ für aktuelles Distanzminimum mod$(C + 1)$

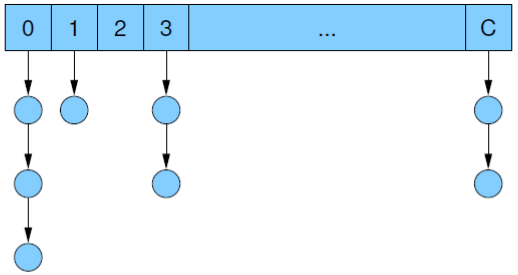






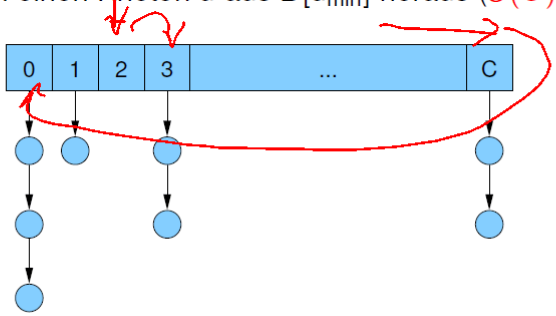

## Bucket Queue

- jeder Knoten $v$ mit aktueller Distanz $d[v]$ in Liste $B[d[v] \bmod (C + 1)]$
- alle Knoten in Liste $B[d]$ haben dieselbe Distanz, weil alle aktuellen Distanzen im Bereich $[d, d + C]$ liegen



# Bucket Queue / Operationen

- insert($v$): fügt $v$ in Liste $B[d[v] \bmod (C+1)]$ ein ($O(1)$)
- decreaseKey($v$): entfernt $v$ aus momentaner Liste ($O(1)$ falls Handle auf Listenelement in $v$ gespeichert) und fügt $v$ in Liste $B[d[v] \bmod (C+1)]$ ein ($O(1)$)
- deleteMin(): solange $B[d_{\min}] = \emptyset$, setze $d_{\min} = (d_{\min}+1) \bmod (C+1)$. Nimm dann einen Knoten $u$ aus $B[d_{\min}]$ heraus ($O(C)$)








# Dijkstra mit Bucket Queue

- insert, decreaseKey: $O(1)$
- deleteMin: $O(C)$
- Dijkstra: $O(m + C \cdot n)$
- lässt sich mit Radix Heaps noch verbessern
- verwendet exponentiell wachsende Bucket-Größen
- Details in der Vorlesung Effiziente Algorithmen und Datenstrukturen
- Laufzeit ist dann $O(m + n \log C)$





# Beliebige Graphen mit beliebigen Gewichten

Gegeben:

- beliebiger Graph mit beliebigen Kantengewichten

$\Rightarrow$ Anhängen einer Kante an einen Weg kann zur Verkürzung des Weges (Kantengewichtssumme) führen (wenn Kante negatives Gewicht hat)

$\Rightarrow$ es kann negative Kreise und Knoten mit Distanz $-\infty$ geben

Problem:

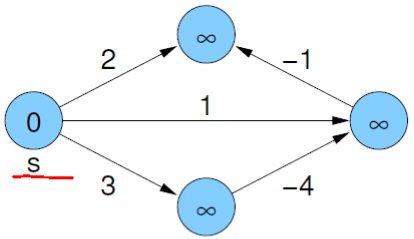
- besuche Knoten eines kürzesten Weges in der richtigen Reihenfolge
- Dijkstra kann nicht mehr verwendet werden, weil Knoten nicht unbedingt in der Reihenfolge der kürzesten Distanz zum Startknoten $s$ besucht werden





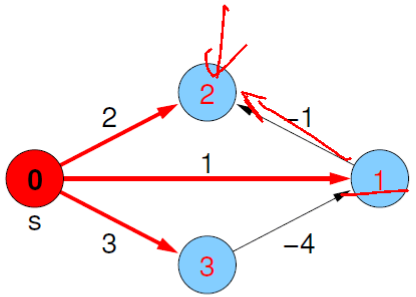
# Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:



# Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:

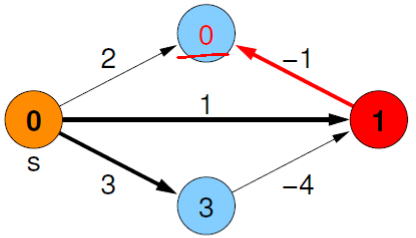







# Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:

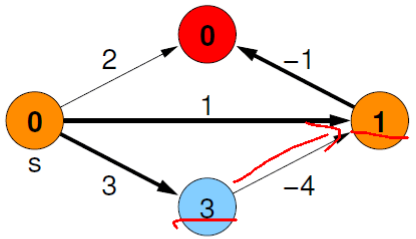







# Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:

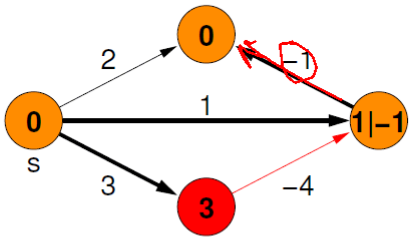






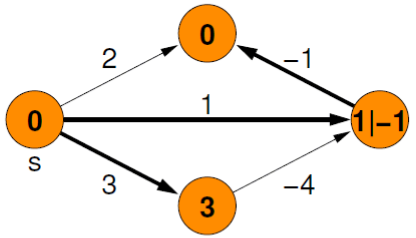

# Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:



# Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:

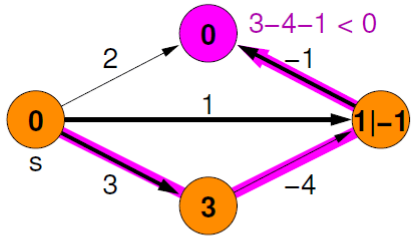







# Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:








# Beliebige Graphen mit beliebigen Gewichten

**Lemma**

*Für jeden von $s$ erreichbaren Knoten $v$ mit $d(s,v) > -\infty$ gibt es einen einfachen Pfad (ohne Kreis) von $s$ nach $v$ der Länge $d(s,v)$.*

**Beweis.**

Betrachte kürzesten Weg mit Kreis(en):

- Kreis mit Kantengewichtssumme $> 0$ nicht enthalten:
  Entfernen des Kreises würde Kosten verringern
- Kreis mit Kantengewichtssumme $= 0$:
  Entfernen des Kreises lässt Kosten unverändert
- Kreis mit Kantengewichtssumme $< 0$:
  Distanz von $s$ ist $-\infty$

□





# Bellman-Ford-Algorithmus

**Folgerung**

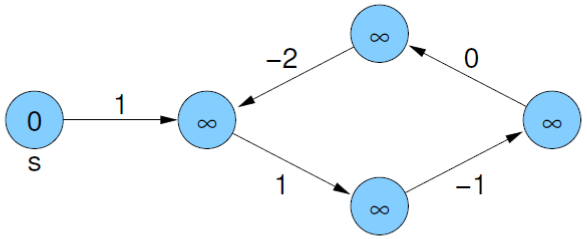
*In einem Graph mit $n$ Knoten gibt es für jeden erreichbaren Knoten $v$ mit $d(s,v) > -\infty$ einen kürzesten Weg bestehend aus $< n$ Kanten zwischen $s$ und $v$.*

Strategie:

- anstatt kürzeste Pfade in Reihenfolge wachsender Gewichtssumme zu berechnen, betrachte sie in Reihenfolge steigender Kantenanzahl
- durchlaufe (n-1)-mal alle Kanten im Graph und aktualisiere die Distanz
- dann alle kürzesten Wege berücksichtigt

# Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise

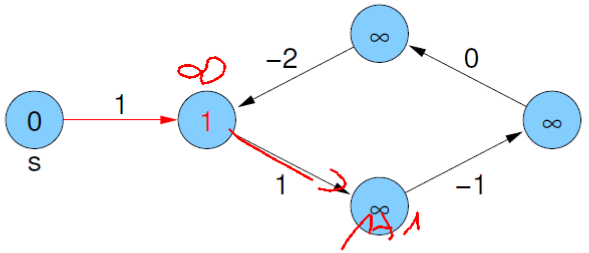







# Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise

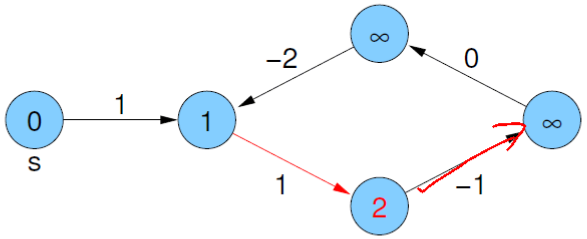







# Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise

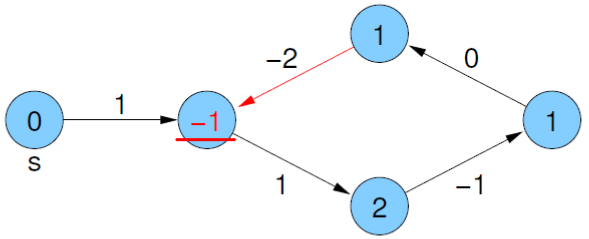






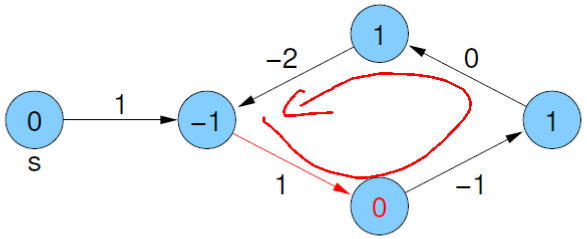

# Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise



# Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise








# Bellman-Ford-Algorithmus

Keine Distanzverringerung mehr möglich:

- Annahme: zu einem Zeitpunkt gilt für alle Kanten $(v, w)$ $d[v] + c(v, w) \geq d[w]$

$\Rightarrow$ (per Induktion) für alle Knoten $w$ und jeden Weg $p$ von $s$ nach $w$ gilt: $d[s] + c(p) \geq d[w]$

- falls sichergestellt, dass zu jedem Zeitpunkt für kürzesten Weg $p$ von $s$ nach $w$ gilt $d[w] \geq c(p)$, dann ist $d[w]$ zum Schluss genau die Länge eines kürzesten Pfades von $s$ nach $w$ (also korrekte Distanz)





# Bellman-Ford-Algorithmus

Zusammenfassung:

- keine Distanzverringerung mehr möglich
  ($d[v] + c(v, w) \geq d[w]$ für alle $w$):
  fertig, alle $d[w]$ korrekt für alle $w$

- Distanzverringerung möglich selbst noch in $n$-ter Runde
  ($d[v] + c(v, w) < d[w]$ für ein $w$):
  Es gibt einen negativen Kreis, also Knoten $w$ mit Distanz $-\infty$.





# Bellman-Ford-Algorithmus

```
BellmanFord(Node s) {
  foreach (v ∈ V) d[v] = ∞;
  d[s] = 0;   parent[s] = s;
  for (int i = 0; i < n − 1; i++) {   // n − 1 Runden
    foreach (e = (v, w) ∈ E)
      if (d[v] + c(e) < d[w]) {   // kürzerer Weg?
        d[w] = d[v] + c(e);
        parent[w] = v;
      }
  }
  foreach (e = (v, w) ∈ E)
    if (d[v] + c(e) < d[w]) {   // kürzerer Weg in n-ter Runde?
      infect(w);
}
```

# Bellman-Ford-Algorithmus

```
infect(Node v) {   // −∞-Knoten
  if (d[v] > −∞) {
    d[v] = −∞;
    foreach (e = (v, w) ∈ E)
      infect(w);
  }
}
```

Gesamtlaufzeit: $O(m \cdot n)$





# Bellman-Ford-Algorithmus

```
BellmanFord(Node s) {
  foreach (v ∈ V) d[v] = ∞;
  d[s] = 0;   parent[s] = s;
  for (int i = 0; i < n − 1; i++) {   // n − 1 Runden
    foreach (e = (v, w) ∈ E)
      if (d[v] + c(e) < d[w]) {   // kürzerer Weg?
        d[w] = d[v] + c(e);
        parent[w] = v;
      }
  }
  foreach (e = (v, w) ∈ E)
    if (d[v] + c(e) < d[w]) {   // kürzerer Weg in n-ter Runde?
      infect(w);
}
```





# Bellman-Ford-Algorithmus

```
infect(Node v) {   // −∞-Knoten
  if (d[v] > −∞) {
    d[v] = −∞;
    foreach (e = (v, w) ∈ E)
      infect(w);
  }
}
```

Gesamtlaufzeit: $O(m \cdot n)$





# Bellman-Ford-Algorithmus

Bestimmung der Knoten mit Distanz $-\infty$:

- betrachte alle Knoten, die in der $n$-ten Phase noch Distanzverbesserung erfahren
- aus jedem Kreis mit negativem Gesamtgewicht muss mindestens ein Knoten dabei sein
- jeder von diesen Knoten aus erreichbare Knoten muss Distanz $-\infty$ bekommen
- das erledigt hier die infect-Funktion
- wenn ein Knoten zweimal auftritt (d.h. der Wert ist schon $-\infty$), wird die Rekursion abgebrochen

# Bellman-Ford-Algorithmus

Bestimmung eines negativen Zyklus:

- bei den oben genannten Knoten sind vielleicht auch Knoten, die nur an negativen Kreisen über ausgehende Kanten angeschlossen sind, die selbst aber nicht Teil eines negativen Kreises sind
- Rückwärtsverfolgung der parent-Werte, bis sich ein Knoten wiederholt
- Kanten vom ersten bis zum zweiten Auftreten bilden einen negativen Zyklus





# Bellman-Ford-Algorithmus

Idee der Updates vorläufiger Distanzwerte: Lester R. Ford Jr.

Verbesserung (Richard E. Bellman / Edward F. Moore):

- benutze Queue von Knoten, zu denen ein kürzerer Pfad gefunden wurde und deren Nachbarn an ausgehenden Kanten noch auf kürzere Wege geprüft werden müssen
- wiederhole: nimm ersten Knoten aus der Queue und prüfe für jede ausgehende Kante die Distanz des Nachbarn

  falls kürzerer Weg gefunden, aktualisiere Distanzwert des Nachbarn und hänge ihn an Queue an (falls nicht schon enthalten)
- Phase besteht immer aus Bearbeitung der Knoten, die am Anfang des Algorithmus (bzw. der Phase) in der Queue sind (dabei kommen während der Phase schon neue Knoten ans Ende der Queue) $\Rightarrow$ $\leq n-1$ Phasen





# Kürzeste einfache Pfade bei beliebigen Kantengewichten

Achtung!

**Fakt**

*Die Suche nach kürzesten einfachen Pfaden (also ohne Knotenwiederholungen / Kreise) in Graphen mit beliebigen Kantengewichten (also möglichen negativen Kreisen) ist ein $\mathcal{NP}$-vollständiges Problem.*

*(Man könnte Hamilton-Pfad-Suche damit lösen.)*





# All Pairs Shortest Paths (APSP)

gegeben:

- Graph mit beliebigen Kantengewichten, der aber keine negativen Kreise enthält

gesucht:

- Distanzen / kürzeste Pfade zwischen allen Knotenpaaren

Naive Strategie:

- $n$-mal Bellman-Ford-Algorithmus (jeder Knoten einmal als Startknoten)

$\Rightarrow$ $O(n^2 \cdot m)$

# APSP / Kantengewichte

Bessere Strategie:

- reduziere $n$ Aufrufe des Bellman-Ford-Algorithmus auf $n$ Aufrufe des Dijkstra-Algorithmus

Problem:

- Dijkstra-Algorithmus funktioniert nur für nichtnegative Kantengewichte

Lösung:

- Umwandlung in nichtnegative Kantenkosten ohne Verfälschung der kürzesten Wege



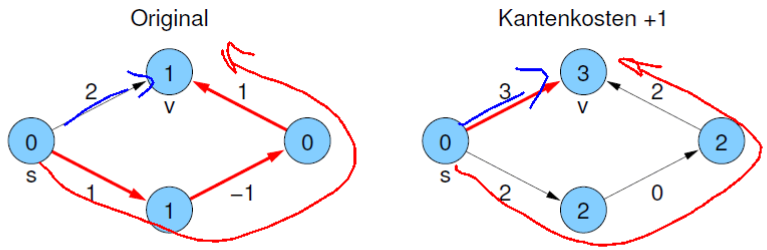


# Naive Modifikation der Kantengewichte

Naive Idee:

- negative Kantengewichte eliminieren, indem auf jedes Kantengewicht der gleiche Wert $c$ addiert wird

$\Rightarrow$ verfälscht kürzeste Pfade








# Knotenpotential

Sei $\Phi : V \mapsto \mathbb{R}$ eine Funktion, die jedem Knoten ein Potential zuordnet.

Modifizierte Kantenkosten von $e = (v, w)$:

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

**Lemma**

*Seien $p$ und $q$ Wege von $v$ nach $w$ in $G$.*

*$c(p)$ und $c(q)$ bzw. $\bar{c}(p)$ und $\bar{c}(q)$ seien die aufsummierten Kosten bzw. modifizierten Kosten der Kanten des jeweiligen Pfads.*

*Dann gilt für jedes Potential $\Phi$:*

$$\bar{c}(p) < \bar{c}(q) \quad \Leftrightarrow \quad c(p) < c(q)$$





# Knotenpotential

**Beweis.**

Sei $p = (v_1, \ldots, v_k)$ beliebiger Weg und $\forall i : \; e_i = (v_i, v_{i+1}) \in E$

Es gilt:

$$\begin{aligned} \bar{c}(p) &= \sum_{i=1}^{k-1} \bar{c}(e_i) \\ &= \sum_{i=1}^{k-1} (\Phi(v_i) + c(e_i) - \Phi(v_{i+1})) \\ &= \Phi(v_1) + c(p) - \Phi(v_k) \end{aligned}$$

d.h. modifizierte Kosten eines Pfads hängen nur von ursprünglichen Pfadkosten und vom Potential des Anfangs- und Endknotens ab.
(Im Lemma ist $v_1 = v$ und $v_k = w$) $\square$

## Potential für nichtnegative Kantengewichte

**Lemma**

*Annahme:*

- *Graph hat keine negativen Kreise*
- *alle Knoten von $s$ aus erreichbar*

*Sei für alle Knoten $v$ das Potential $\Phi(v) = d(s,v)$.*

*Dann gilt für alle Kanten $e$:* $\bar{c}(e) \geq 0$

**Beweis.**

- für alle Knoten $v$ gilt nach Annahme: $d(s,v) \in \mathbb{R}$ (also $\neq \pm\infty$)
- für jede Kante $e = (v,w)$ ist

$$d(s,v) + c(e) \geq d(s,w)$$

$$d(s,v) + c(e) - d(s,w) \geq 0$$

$$\bar{c}(e) = \Phi(v) + c(e) - \Phi(w) \geq 0$$

□





## 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

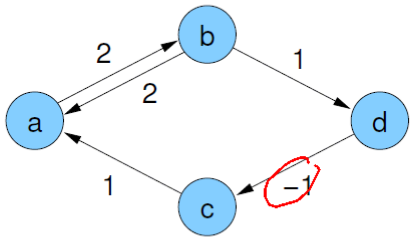
- berechne für alle Knoten $v$ die Distanzen $\bar{d}(v,w)$ mittels Dijkstra-Algorithmus mit modifizierten Kantenkosten auf dem Graph ohne Knoten $s$
- berechne korrekte Distanzen $d(v,w) = \bar{d}(v,w) + \Phi(w) - \Phi(v)$





## Johnson-Algorithmus für APSP

Beispiel:

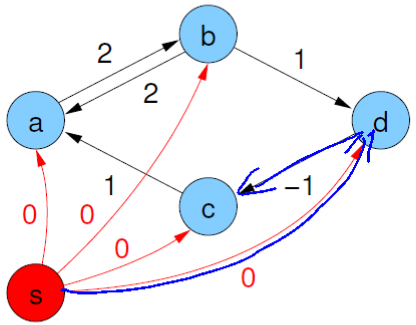






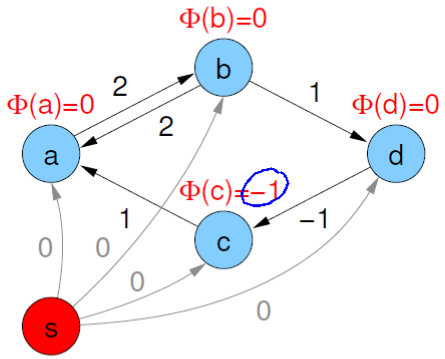

## Johnson-Algorithmus für APSP

1. künstlicher Startknoten $s$:



# Johnson-Algorithmus für APSP

2. Bellman-Ford-Algorithmus auf $s$:



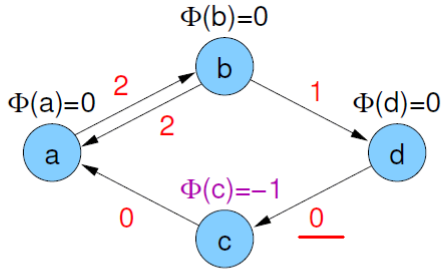





# Johnson-Algorithmus für APSP

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

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

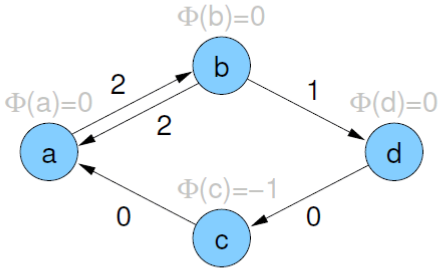







# Johnson-Algorithmus für APSP

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



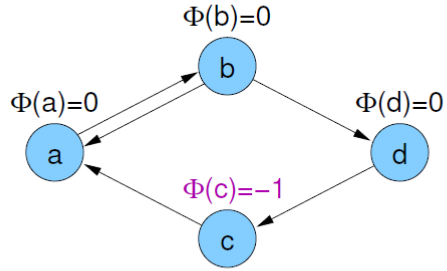

| **d** | *a* | *b* | *c* | *d* |
|---|---|---|---|---|
| *a* | 0 | 2 | 3 | 3 |
| *b* | 1 | 0 | 1 | 1 |
| *c* | 0 | 2 | 0 | 3 |
| *d* | 0 | 2 | 0 | 0 |





# 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

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 $v$)



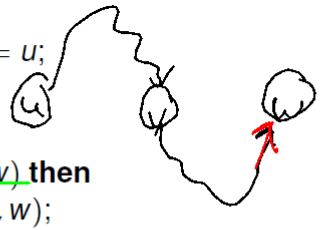


# Floyd-Warshall-Algorithmus für APSP

**Algorithmus Floyd-Warshall:** löst APSP-Problem

**Eingabe** : Graph $G = (V, E)$, $c : E \mapsto \mathbb{R}$
**Ausgabe** : Distanzen $d(u, v)$ zwischen allen $u, v \in V$

**for** $u, v \in V$ **do**
  $d(u, v) = \infty$; $\text{pred}(u, v) = \perp$;
**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**
      $d(u, w) = d(u, v) + d(v, w)$;
      $\text{pred}(u, w) = \text{pred}(v, w)$;








# Floyd-Warshall-Algorithmus für APSP

**Algorithmus Floyd-Warshall:** löst APSP-Problem

**Eingabe** : Graph $G = (V, E)$, $c : E \mapsto \mathbb{R}$
**Ausgabe** : Distanzen $d(u, v)$ zwischen allen $u, v \in V$

**for** $u, v \in V$ **do**
  $d(u, v) = \infty$; $\text{pred}(u, v) = \perp$;
**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**
      $d(u, w) = d(u, v) + d(v, w)$;
      $\text{pred}(u, w) = \text{pred}(v, w)$;

# 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

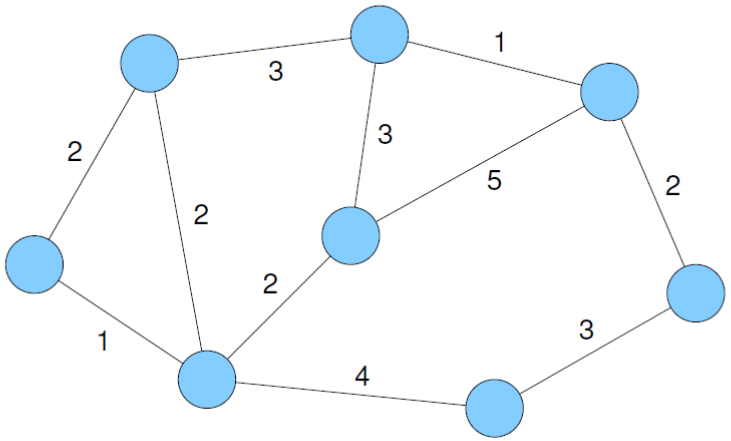






# Minimaler Spannbaum

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)

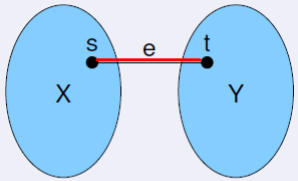
# Minimaler Spannbaum

## 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.*



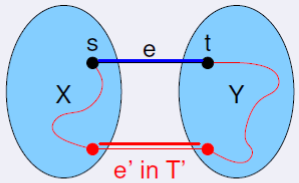





# Minimaler Spannbaum

## 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)

□





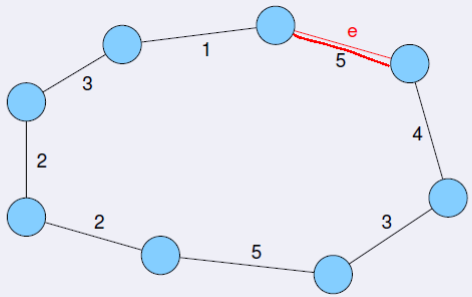
# Minimaler Spannbaum

## 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 $G$*



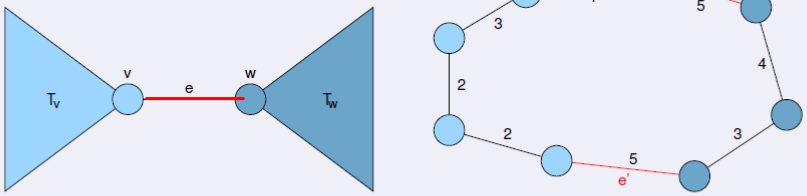





# Minimaler Spannbaum

## Beweis.

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

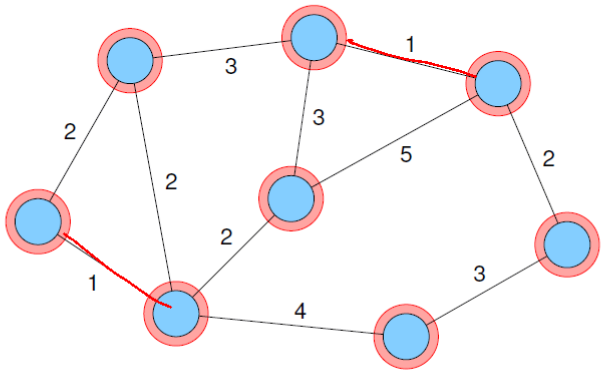



- es muss (mindestens) eine weitere Kante $e'$ in $C$ geben, die einen Knoten aus $T_v$ mit einem Knoten aus $T_w$ verbindet
- Ersetzen von $e$ durch $e'$ ergibt einen Baum $T'$ dessen Gewicht nicht größer sein kann als das von $T$, also ist $T'$ auch MSB

# Minimaler Spannbaum

Regel:

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



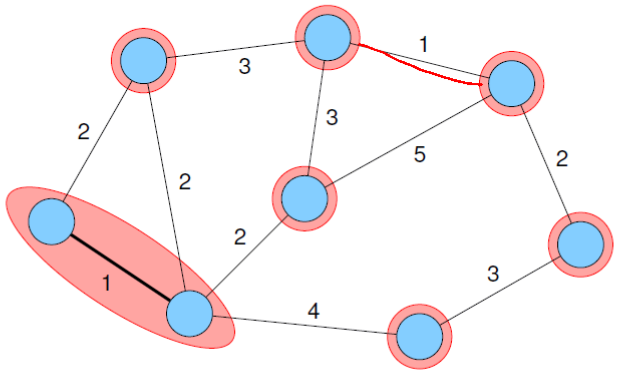





# Minimaler Spannbaum

Regel:

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



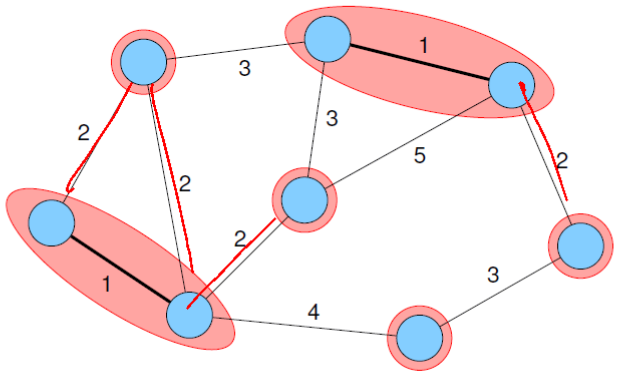





# Minimaler Spannbaum

Regel:

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



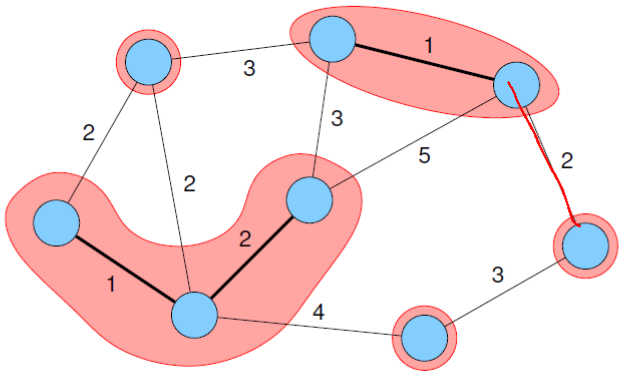





# Minimaler Spannbaum

Regel:

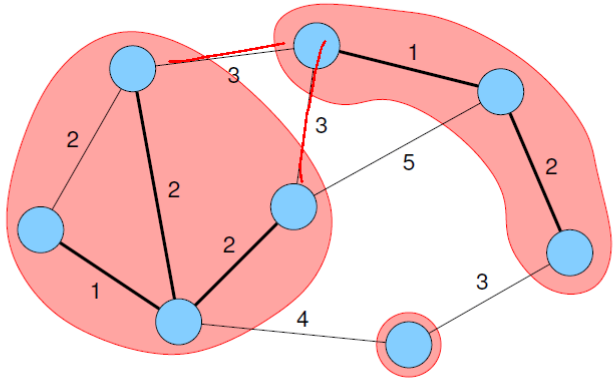
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



# Minimaler Spannbaum

Regel:

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



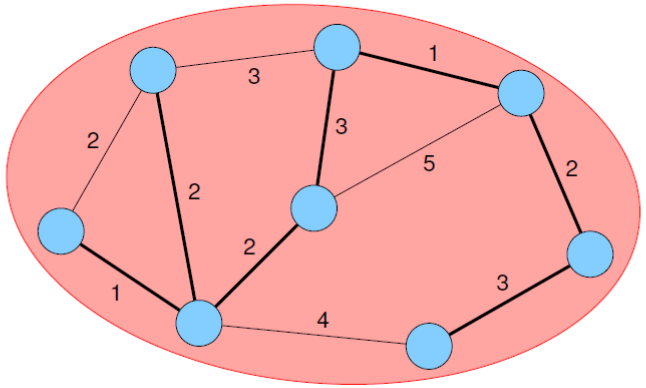





# Minimaler Spannbaum

Regel:

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



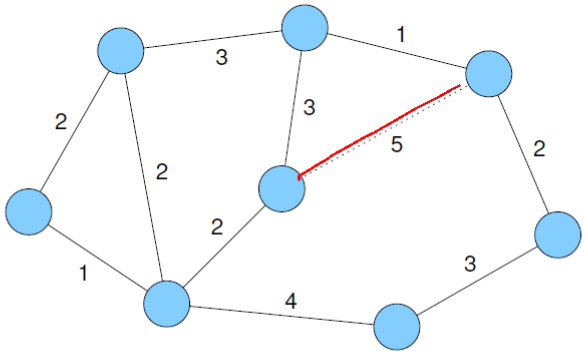





# Minimaler Spannbaum

Regel:

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



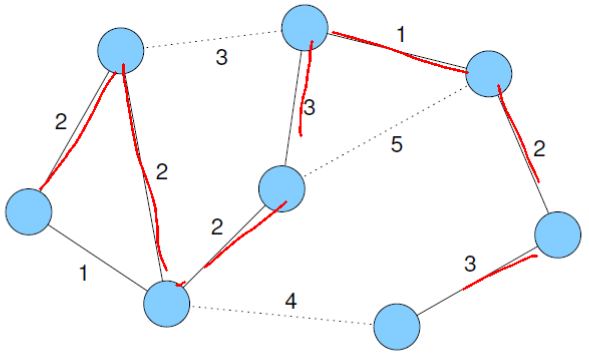





# Minimaler Spannbaum

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

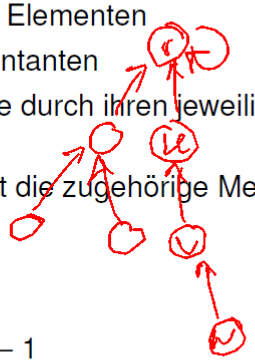




# Union-Find-Datenstruktur

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)

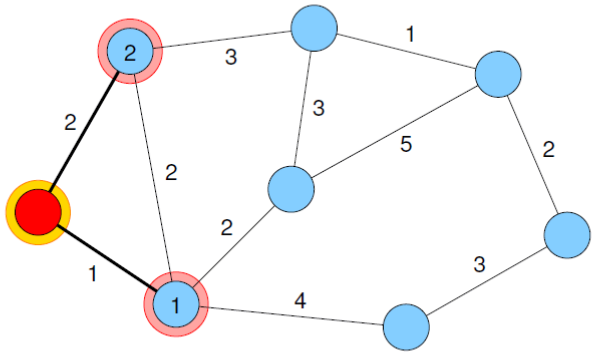
$\Rightarrow$ Baum umfasst jetzt 1 Knoten / Kante mehr

- wiederhole Auswahl bis alle $n$ Knoten im Baum





## Algorithmus von Prim








## Jarník-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: $O(m + n\log n)$