Script generated by TTT

Title: Seidl: GAD (07.07.2015)

Date: Tue Jul 07 13:44:42 CEST 2015

Duration: 150:47 min

Pages: 86

Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Topologische Sortierung – warum funktioniert das?

- betrachte einen kürzesten Weg von s nach v
- der ganze Pfad beachtet die topologische Sortierung
- d.h., die Distanzen werden in der Reihenfolge der Knoten vom Anfang des Pfades zum Ende hin betrachtet
- damit ergibt sich für v der richtige Distanzwert

- ein Knoten x kann auch nie einen Wert erhalten, der echt kleiner als seine Distanz zu s ist
- die Kantenfolge von s zu x, die jeweils zu den Distanzwerten an den Knoten geführt hat, wäre dann ein kürzerer Pfad (Widerspruch)

Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Allgemeine Strategie:

- Anfang: setze $d(s) = 0$ und
  für alle anderen Knoten v setze $d(v) = \infty$
- besuche Knoten in einer Reihenfolge, die sicherstellt, dass mindestens ein kürzester Weg von s zu jedem v in der Reihenfolge seiner Knoten besucht wird
- für jeden besuchten Knoten v aktualisiere die Distanzen der Knoten w mit $(v, w) \in E$, d.h. setze

$$d(w) = \min\{\, d(w),\ d(v) + c(v, w) \,\}$$

Kürzeste Wege in DAGs

Topologische Sortierung

- verwende FIFO-Queue q
- verwalte für jeden Knoten einen Zähler für die noch nicht markierten eingehenden Kanten
- initialisiere q mit allen Knoten, die keine eingehende Kante haben (Quellen)
- nimm nächsten Knoten v aus q und
  markiere alle $(v, w) \in E$, d.h. dekrementiere Zähler für w
- falls der Zähler von w dabei Null wird, füge w in q ein
- wiederhole das, bis q leer wird

Kürzeste Wege in DAGs

DAG-Strategie

1. Topologische Sortierung der Knoten

   Laufzeit $\mathcal{O}(n+m)$

2. Aktualisierung der Distanzen gemäß der topologischen Sortierung

   Laufzeit $\mathcal{O}(n+m)$

Gesamtlaufzeit: $\mathcal{O}(n+m)$

Beliebige Graphen mit nicht-negativen Gewichten

Gegeben:

- beliebiger Graph
  (gerichtet oder ungerichtet, muss diesmal kein DAG sein)
- mit nicht-negativen Kantengewichten

$\Rightarrow$ keine Knoten mit Distanz $-\infty$

Problem:

- besuche Knoten eines kürzesten Weges in der richtigen Reihenfolge
- wie bei Breitensuche, jedoch diesmal auch mit Distanzen $\neq 1$

Lösung:

- besuche Knoten in der Reihenfolge der kürzesten Distanz zum Startknoten s

Kürzeste Pfade: SSSP / Dijkstra

Algorithmus Dijkstra1: löst SSSP-Problem

Eingabe : $G = (V, E)$, $c: E \mapsto \mathbb{R}$, $s \in V$
Ausgabe : Distanzen $d(s, v)$ zu allen $v \in V$

$P = \emptyset$; $T = V$;
forall the $v \in V \setminus \{s\}$ **do**
  $d(s, v) = \infty$
;
$d(s, s) = 0$; $pred(s) = \perp$;
while $(P \neq V)$ **do**
  $v = \text{argmin}_{v \in T}\{d(s, v)\}$;
  $P = P \cup v$; $T = T \setminus v$;
  forall the $(v, w) \in E$ **do**
    if $d(s, w) > d(s, v) + c(v, w)$ **then**
      $d(s, w) = d(s, v) + c(v, w)$;
      $pred(w) = v$;

Graphen | Kürzeste Wege

Kürzeste Pfade: SSSP / Dijkstra

Algorithmus Dijkstra1: löst SSSP-Problem

Eingabe : $G = (V, E)$, $c: E \mapsto \mathbb{R}$, $s \in V$
Ausgabe : Distanzen $d(s, v)$ zu allen $v \in V$

$P = \emptyset$; $T = V$;
forall the $v \in V \setminus \{s\}$ **do**
  $d(s, v) = \infty$
;
$d(s, s) = 0$; $pred(s) = \perp$;
while $(P \neq V)$ **do**
  $v = \text{argmin}_{v \in T}\{d(s, v)\}$;
  $P = P \cup v$; $T = T \setminus v$;
  forall the $(v, w) \in E$ **do**
    if $d(s, w) > d(s, v) + c(v, w)$ **then**
      $d(s, w) = d(s, v) + c(v, w)$;
      $pred(w) = v$;

H. Seidl (TUM) GAD SS'15 503

Algorithmus Dijkstra2: löst SSSP-Problem

Eingabe : $G = (V, E)$, $c : E \mapsto \mathbb{R}_{\geq 0}$, $s \in V$
Ausgabe : Distanzen $d[v]$ von s zu allen $v \in V$

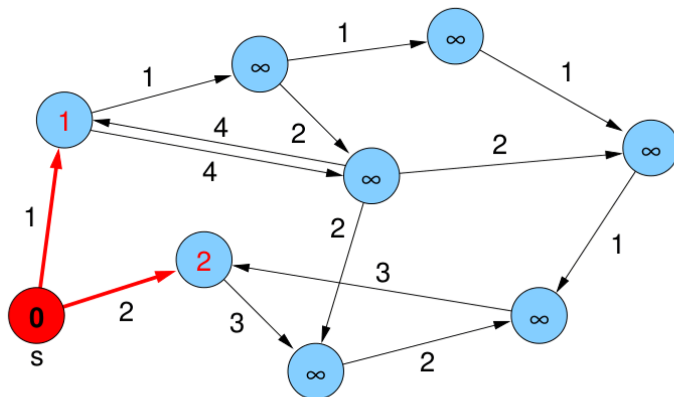
```
forall the v ∈ V \ s do
    d[v] = ∞
;
d[s] = 0;   pred[s] = ⊥;
pq = ⟨⟩;   pq.insert(s, 0);
while (¬pq.empty()) do
    v = pq.deleteMin();
    forall the (v, w) ∈ E do
        newDist = d[v] + c(v, w);
        if (newDist < d[w]) then
            pred[w] = v;
            if (d[w] == ∞) then  pq.insert(w,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

Dijkstra-Algorithmus

- setze Startwert $d(s, s) = 0$ und zunächst $d(s, v) = \infty$
- verwende Prioritätswarteschlange, um die Knoten zusammen mit ihren aktuellen Distanzen zuspeichern
- am Anfang nur Startknoten (mit Distanz 0) in Priority Queue
- dann immer nächsten Knoten v (mit kleinster Distanz) entnehmen, endgültige Distanz dieses Knotens v steht nun fest
- betrachte alle Nachbarn von v,
  füge sie ggf. in die PQ ein bzw.
  aktualisiere deren Priorität in der PQ

Dijkstra-Algorithmus

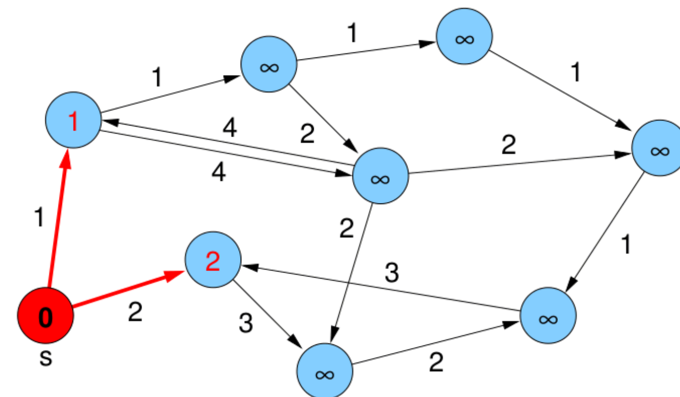
Beispiel:



Graphen Kürzeste Wege

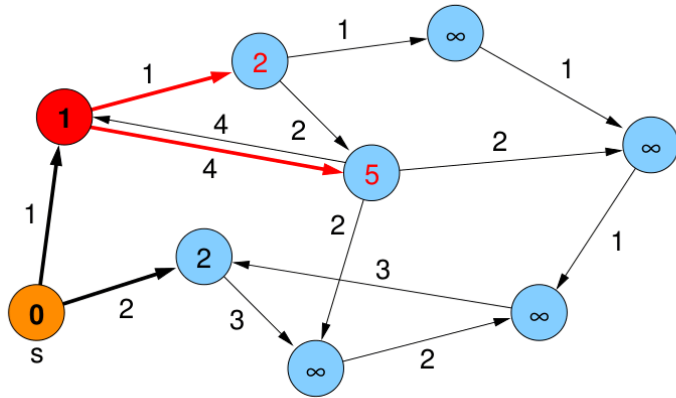
Dijkstra-Algorithmus

Beispiel:



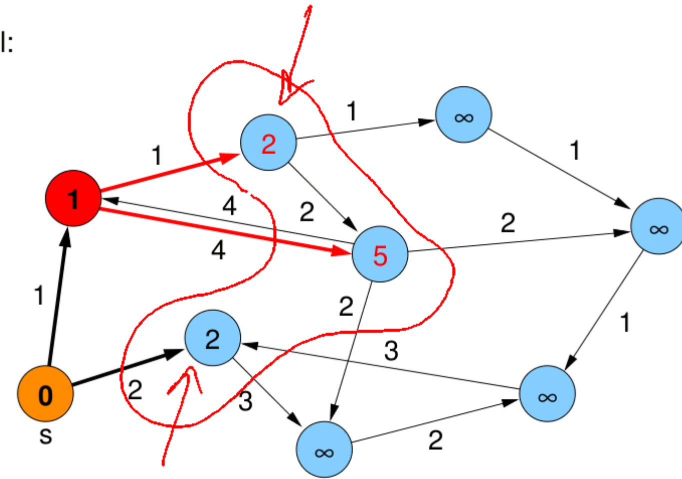
Dijkstra-Algorithmus

Beispiel:



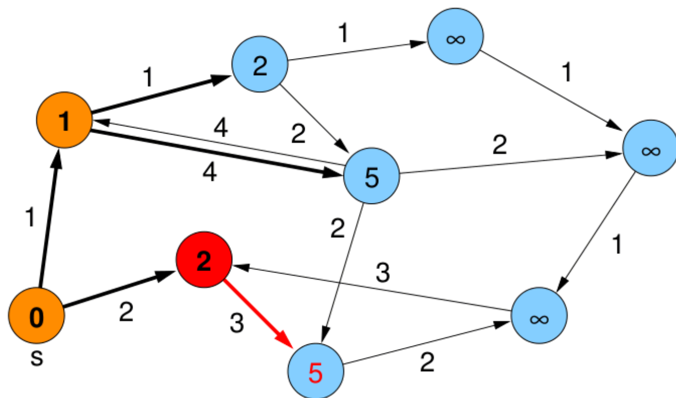
Dijkstra-Algorithmus

Beispiel:



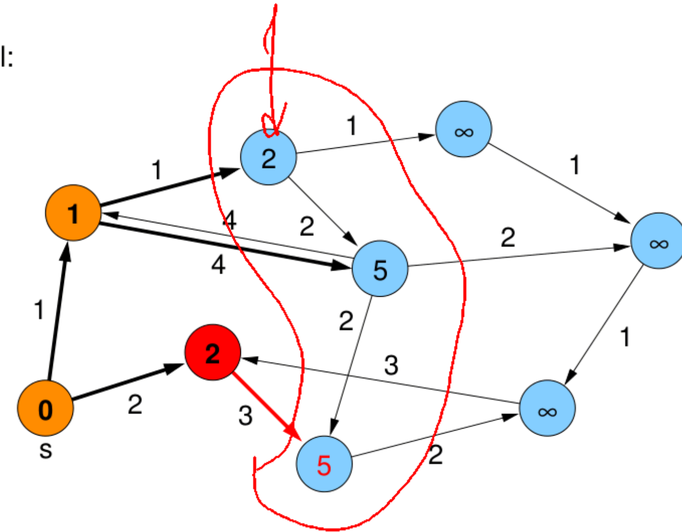
Dijkstra-Algorithmus

Beispiel:



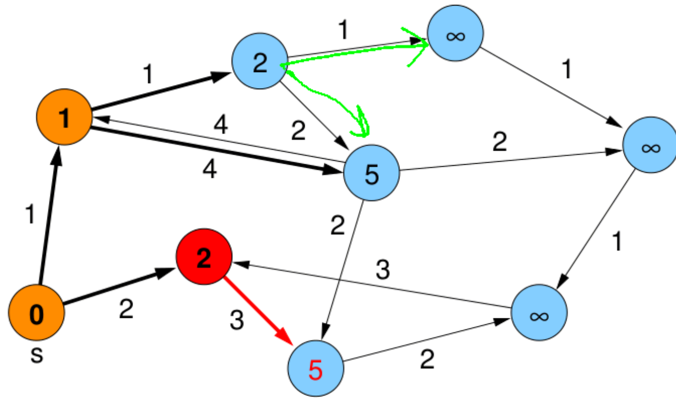
Dijkstra-Algorithmus

Beispiel:



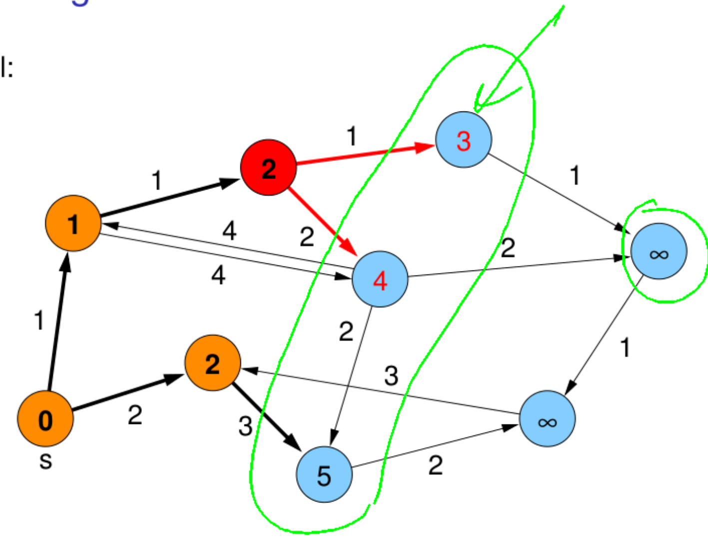
Dijkstra-Algorithmus

Beispiel:



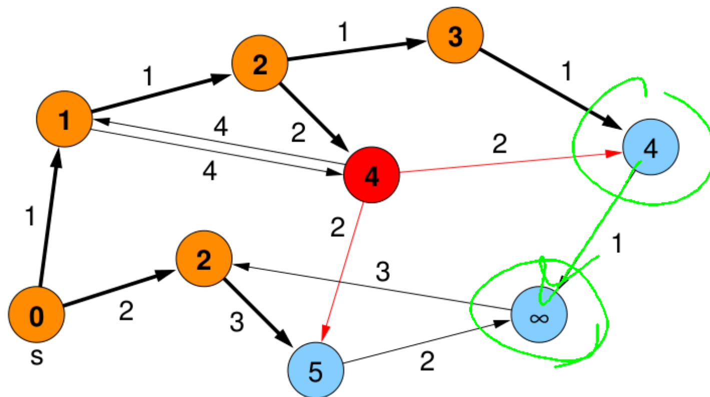
Dijkstra-Algorithmus

Beispiel:



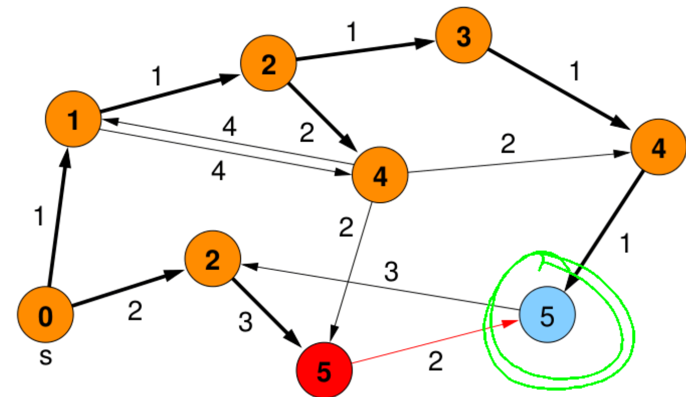
Dijkstra-Algorithmus

Beispiel:



Dijkstra-Algorithmus

Beispiel:



Dijkstra-Algorithmus

Korrektheit:

- Annahme: Algorithmus liefert für w einen zu kleinen Wert $d(s, w)$
- sei w der erste Knoten, für den die Distanz falsch festgelegt wird (kann nicht s sein, denn die Distanz $d(s, s)$ bleibt immer 0)
- kann nicht sein, weil $d(s, w)$ nur dann aktualisiert wird, wenn man über einen von s schon erreichten Knoten v mit Distanz $d(s, v)$ den Knoten w über die Kante (v, w) mit Distanz $d(s, v) + c(v, w)$ erreichen kann
- d.h. $d(s, v)$ müsste schon falsch gewesen sein (Widerspruch zur Annahme, dass w der erste Knoten mit falscher Distanz war)

Dijkstra-Algorithmus

- Annahme: Algorithmus liefert für w einen zu großen Wert $d(s, w)$
- sei w der Knoten mit der kleinsten (wirklichen) Distanz, für den der Wert $d(s, w)$ falsch festgelegt wird (wenn es davon mehrere gibt, der Knoten, für den die Distanz zuletzt festgelegt wird)
- kann nicht sein, weil $d(s, w)$ immer aktualisiert wird, wenn man über einen von s schon erreichten Knoten v mit Distanz $d(s, v)$ den Knoten w über die Kante (v, w) mit Distanz $d(s, v) + c(v, w)$ erreichen kann (dabei steht $d(s, v)$ immer schon fest, so dass auch die Länge eines kürzesten Wegs über v zu w richtig berechnet wird)
- d.h., entweder wurde auch der Wert von v falsch berechnet (Widerspruch zur Def. von w) oder die Distanz von v wurde noch nicht festgesetzt
- weil die berechneten Distanzwerte monoton wachsen, kann letzteres nur passieren, wenn v die gleiche Distanz hat wie w (auch Widerspruch zur Def. von w)

Dijkstra-Algorithmus

- Datenstruktur: Prioritätswarteschlange (z.B. Fibonacci Heap: amortisierte Komplexität $O(1)$ für `insert` und `decreaseKey`, $O(\log n)$ `deleteMin`)
- Komplexität:
  - $n \times O(1)$ `insert`
  - $n \times O(\log n)$ `deleteMin`
  - $m \times O(1)$ `decreaseKey`

  $\Rightarrow O(m + n \log n)$
- aber: nur für nichtnegative Kantengewichte(!)

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

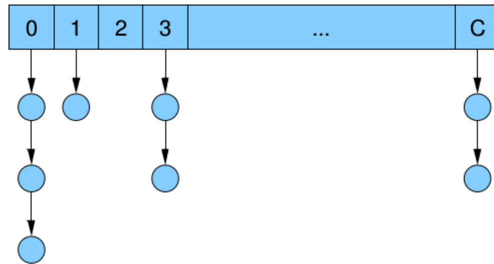
Annahme: alle Kantengewichte im Bereich $[0, C]$

Konsequenz für Dijkstra-Algorithmus:

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

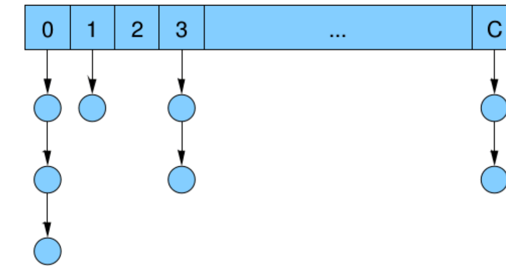
Bucket Queue

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



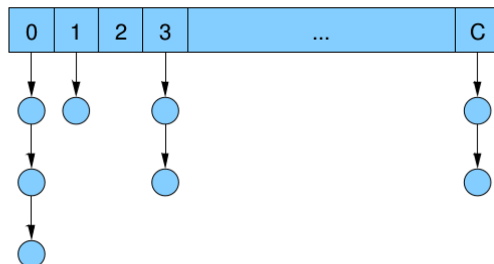
Bucket Queue

- 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

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

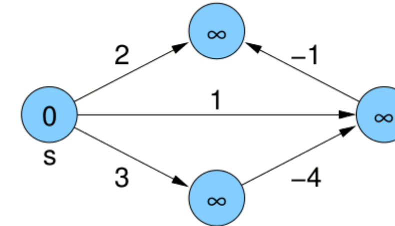
⇒ es kann negative Kreise und Knoten mit Distanz $-\infty$ geben

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

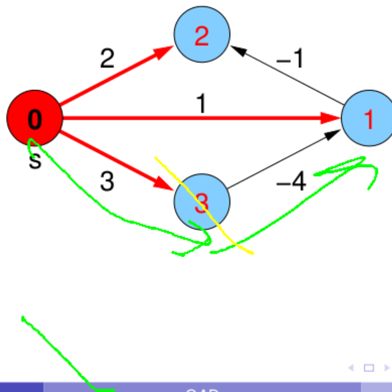
Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:



Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:



Beliebige Graphen mit beliebigen Gewichten

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

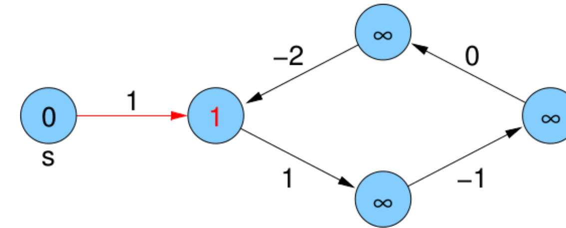
In einem Graph mit n Knoten gibt es für jeden erreichbaren Knoten v mit $d(s, v) > -\infty$ einen kürzesten Weg bestehend aus < n Kanten zwischen s und v.

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

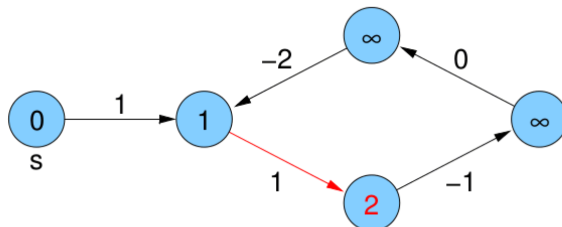
Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise



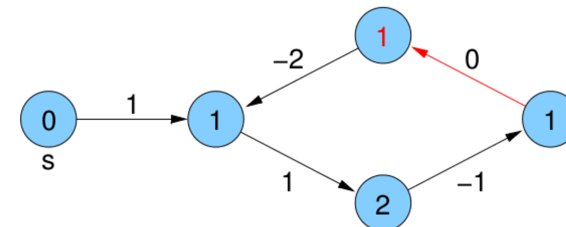
Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise



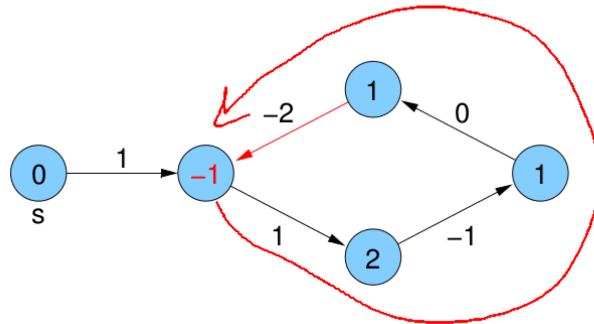
Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise



Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise



Bellman-Ford-Algorithmus

Folgerung

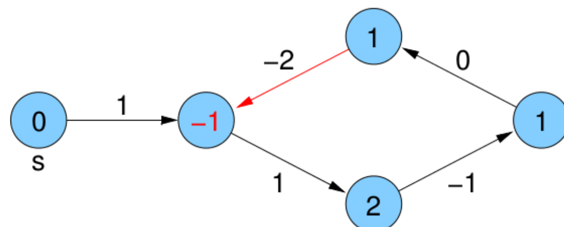
In einem Graph mit n Knoten gibt es für jeden erreichbaren Knoten v mit $d(s, v) > -\infty$ einen kürzesten Weg bestehend aus $< n$ Kanten zwischen s und v.

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

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

```
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

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);
}
```

$O(m + n)$

Bellman-Ford-Algorithmus

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

$O(n + n \cdot m + m + n)$

Gesamtlaufzeit: $O(m \cdot n)$

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

Graphen Kürzeste Wege

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

H. Seidl (TUM) GAD SS'15 547

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

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

Graphen | Kürzeste Wege

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

H. Seidl (TUM) GAD SS'15 548

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

Graphen Kürzeste Wege

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

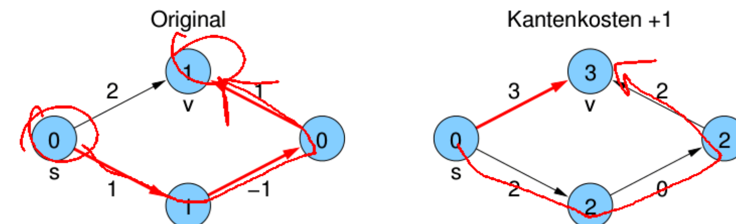
H. Seidl (TUM) GAD SS'15 551

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 Φ:

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

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 $v_k = w$) □

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

$$\begin{aligned} d(s, v) + c(e) &\geq d(s, w) \\ d(s, v) + c(e) - d(s, w) &\geq 0 \end{aligned}$$

□

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

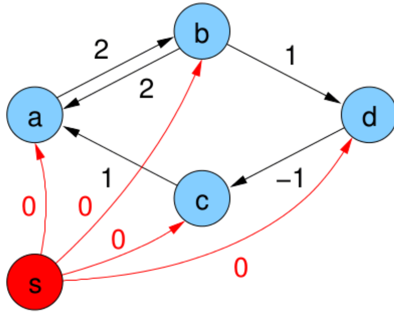
- berechne $d(s, v)$ mit Bellman-Ford-Algorithmus
- setze $\Phi(v) = d(s, v)$ für alle v
- berechne modifizierte Kosten $\bar{c}(e)$

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

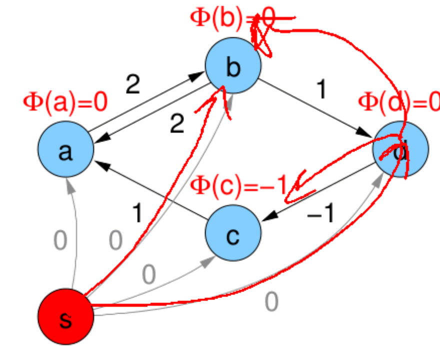
Johnson-Algorithmus für APSP

1. künstlicher Startknoten *s*:



Johnson-Algorithmus für APSP

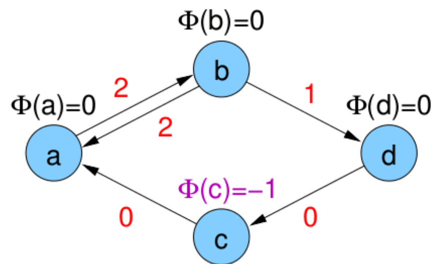
2. Bellman-Ford-Algorithmus auf *s*:



Johnson-Algorithmus für APSP

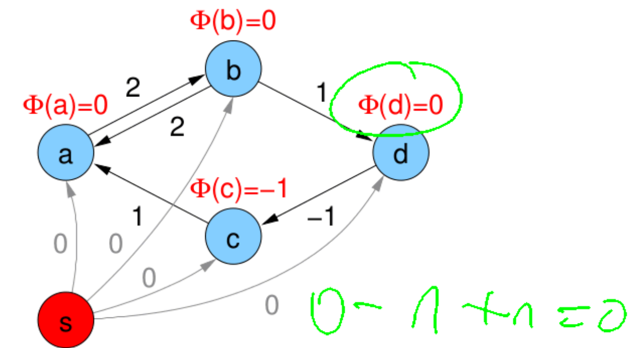
3. $\bar{c}(e)$-Werte für alle $e = (v, w)$ berechnen:

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



Johnson-Algorithmus für APSP

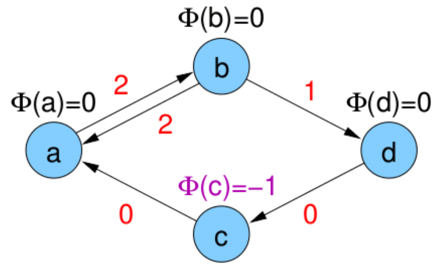
2. Bellman-Ford-Algorithmus auf *s*:



Johnson-Algorithmus für APSP

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

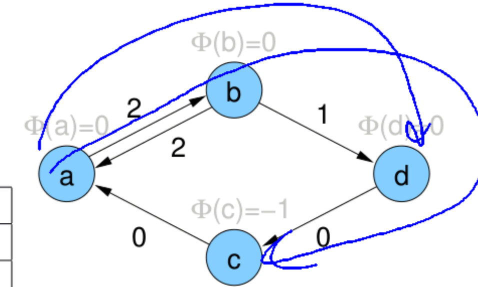
$$\bar{c}(e) = \Phi(v) + c(e) - \Phi(w)$$



Johnson-Algorithmus für APSP

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

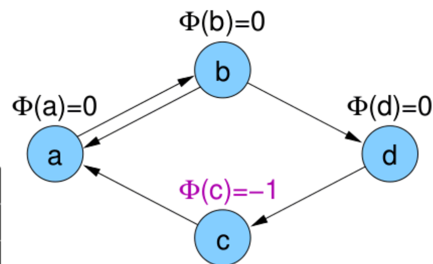
$\bar{\mathbf{d}}$	a	b	c	d
a	0	2	3	3
b	1	0	1	1
c	0	2	0	3
d	0	2	0	0



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



$d(a,c) = \bar{d}(a,c) - \Phi(a) + \Phi(c)$

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

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

n^2
n
$n \leq n^2$
$n \cdot n^2 = n^3$

Floyd-Warshall-Algorithmus für APSP

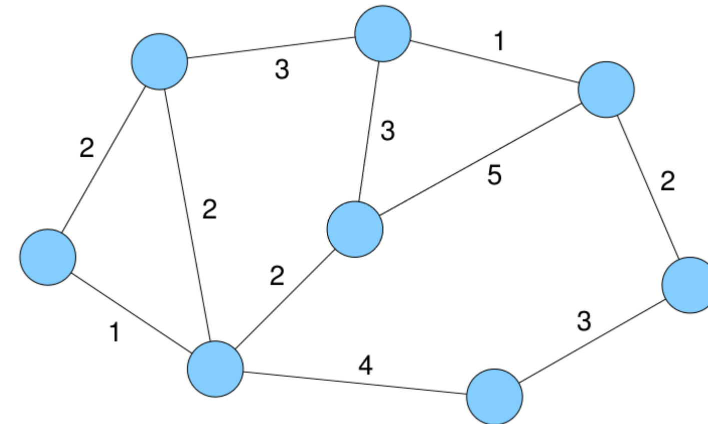
- Komplexität: $O(n^3)$
- funktioniert auch, wenn Kanten mit negativem Gewicht existieren
- Kreise negativer Länge werden nicht direkt erkannt und verfälschen das Ergebnis, sind aber indirekt am Ende an negativen Diagonaleinträgen der Distanzmatrix erkennbar

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

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:

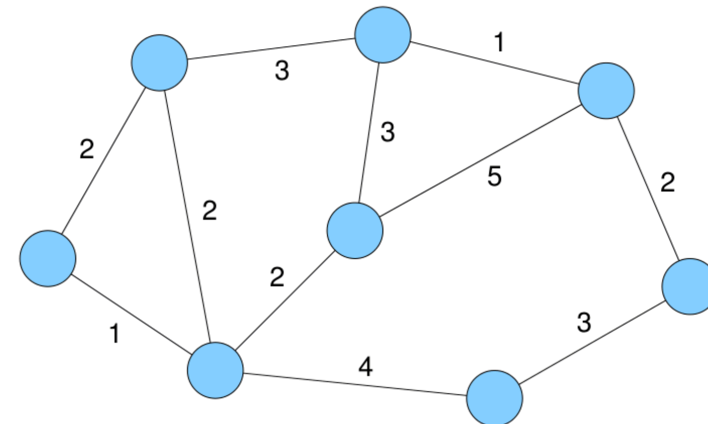
- T formt immer einen Baum (wenn Kantengewichte echt positiv)

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

Graphen | Minimale Spannbäume

Minimaler Spannbaum

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



H. Seidl (TUM) GAD SS'15 568

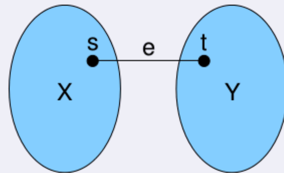
Minimaler Spannbaum

Lemma

Sei

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

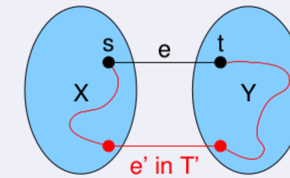
Dann gibt es einen minimalen Spannbaum T, der e enthält.



Minimaler Spannbaum

Beweis.

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



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

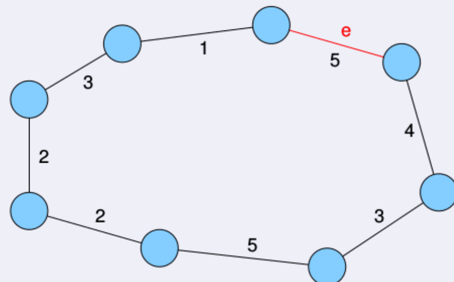
Minimaler Spannbaum

Lemma

Betrachte

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

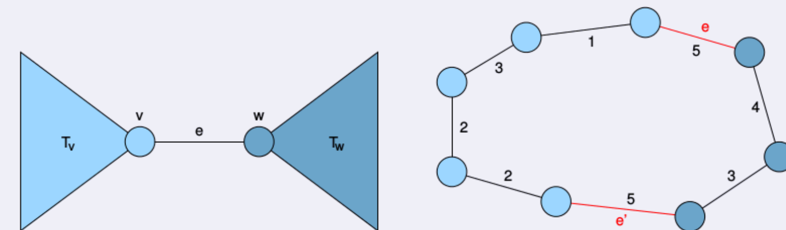
Dann ist jeder MSB in G ohne e auch ein MSB in G



Minimaler Spannbaum

Beweis.

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

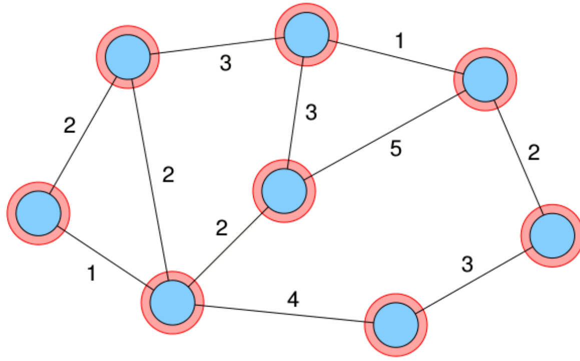


- es muss (mindestens) eine weitere Kante e' in C geben, die einen Knoten aus T_v mit einem Knoten aus T_w verbindet
- Ersetzen von e durch e' ergibt einen Baum T' dessen Gewicht nicht größer sein kann als das von T, also ist T' auch MSB

Minimaler Spannbaum

Regel:

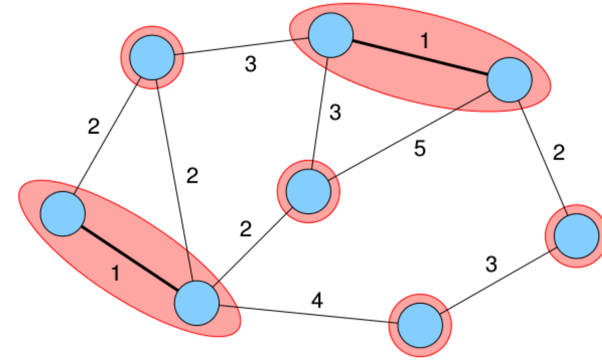
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

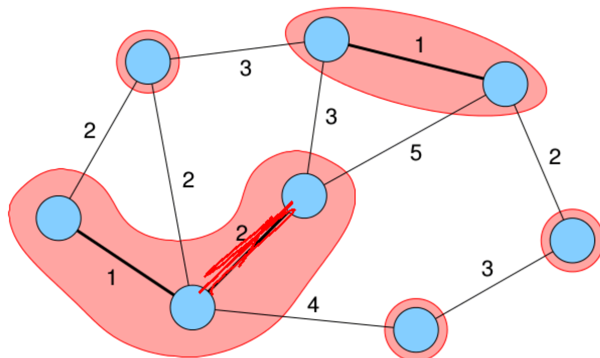
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

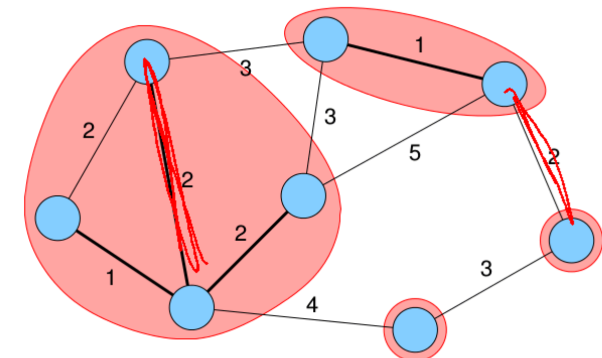
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

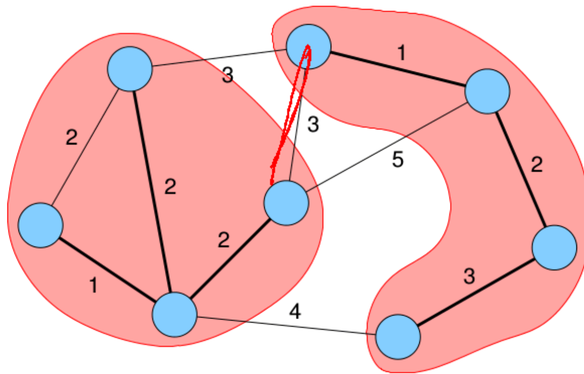
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

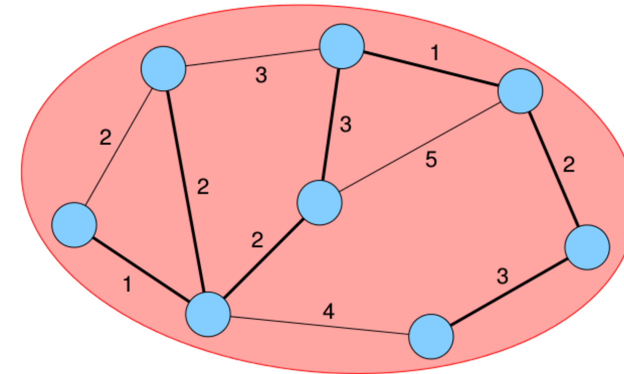
- wähle wiederholt Kante mit minimalen Kosten, die zwei Zusammenhangskomponenten verbindet
- bis nur noch eine Zusammenhangskomponente übrig ist



Minimaler Spannbaum

Regel:

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



Minimaler Spannbaum

Regel:

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