

Script generated by TTT

Title: H. Täubig: GAD (09.07.2013)

Date: Tue Jul 09 14:15:34 CEST 2013

Duration: 91:07 min

Pages: 54

Übersicht

- 9 Graphen
 - Netzwerke und Graphen
 - Graphrepräsentation
 - Graphtraversierung
 - Kürzeste Wege**
 - Minimale Spannbäume

Kürzeste Wege

Zentrale Frage: Wie kommt man am schnellsten von A nach B?

Kürzeste Wege

Zentrale Frage: Wie kommt man am schnellsten von A nach B?

Fälle:

- Kantenkosten 1
- DAG, beliebige Kantenkosten
- beliebiger Graph, positive Kantenkosten
- beliebiger Graph, beliebige Kantenkosten

Kürzeste-Wege-Problem

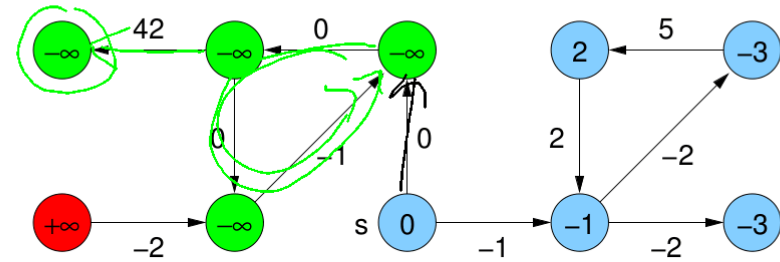
gegeben:

- gerichteter Graph $G = (V, E)$
- Kantenkosten $c : E \mapsto \mathbb{R}$

2 Varianten:

- SSSP (single source shortest paths):
kürzeste Wege von einer Quelle zu allen anderen Knoten
- APSP (all pairs shortest paths):
kürzeste Wege zwischen allen Paaren

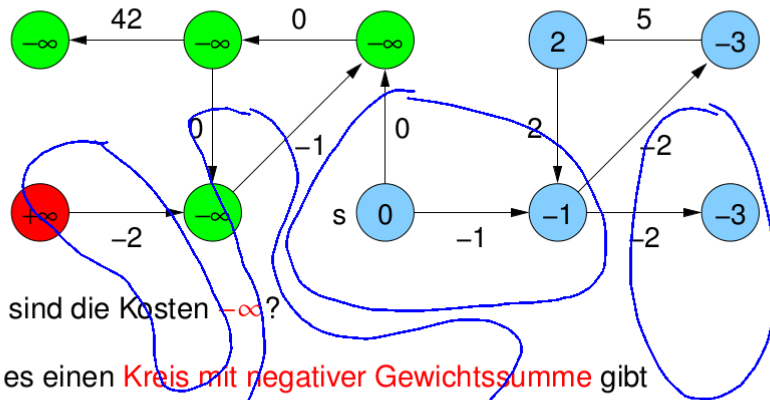
Distanzen



$\mu(s, v)$: Distanz von s nach v

$$\mu(s, v) = \begin{cases} +\infty & \text{kein Weg von } s \text{ nach } v \\ -\infty & \text{Weg beliebig kleiner Kosten von } s \text{ nach } v \\ \min\{c(p) : p \text{ ist Weg von } s \text{ nach } v\} & \end{cases}$$

Distanzen



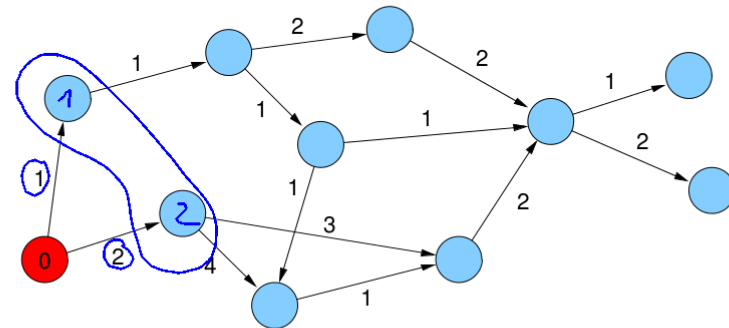
Wann sind die Kosten $-\infty$?

wenn es einen **Kreis mit negativer Gewichtssumme** gibt
(hinreichende und notwendige Bedingung)

Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

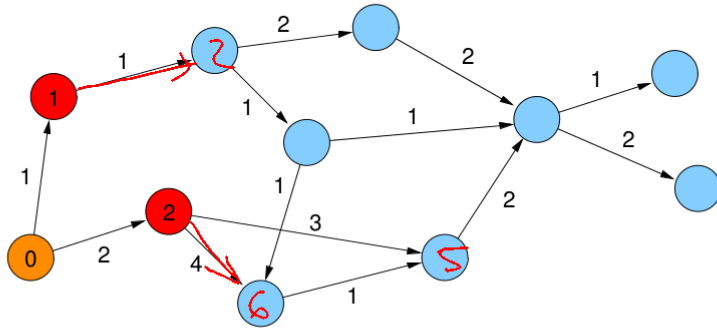
Einfache Breitensuche funktioniert nicht.



Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

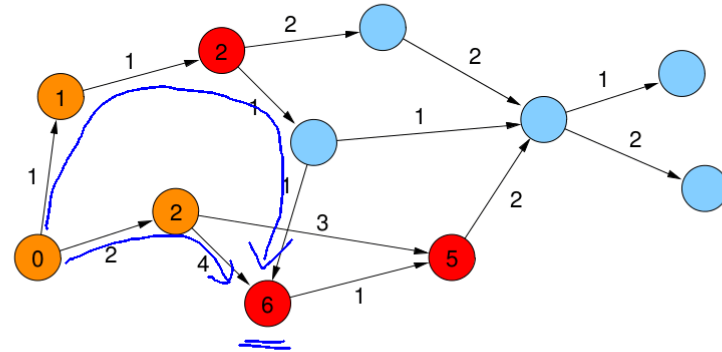
Einfache Breitensuche funktioniert nicht.



Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

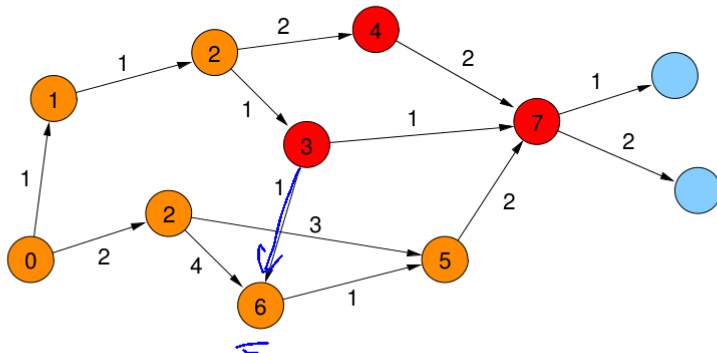
Einfache Breitensuche funktioniert nicht.



Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

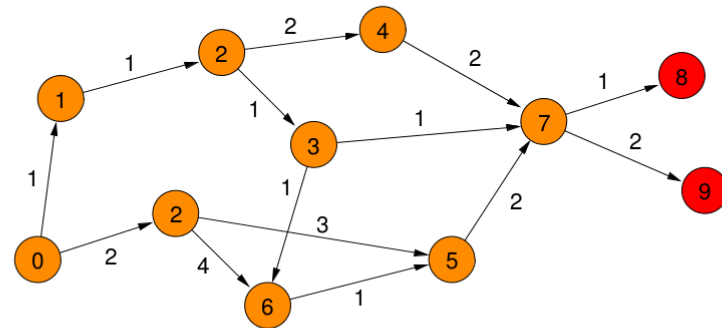
Einfache Breitensuche funktioniert nicht.



Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

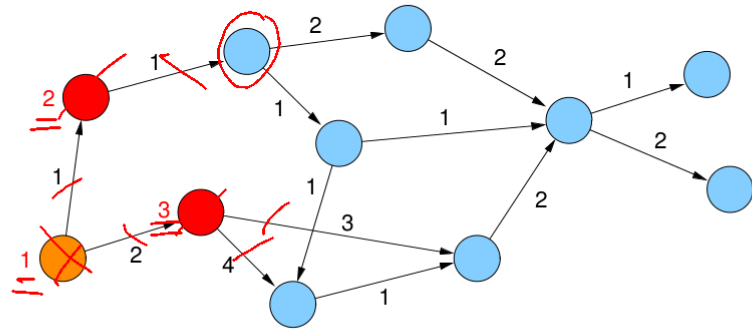
Einfache Breitensuche funktioniert nicht.



Topologische Sortierung in DAGs

Beliebige Kantengewichte in DAGs

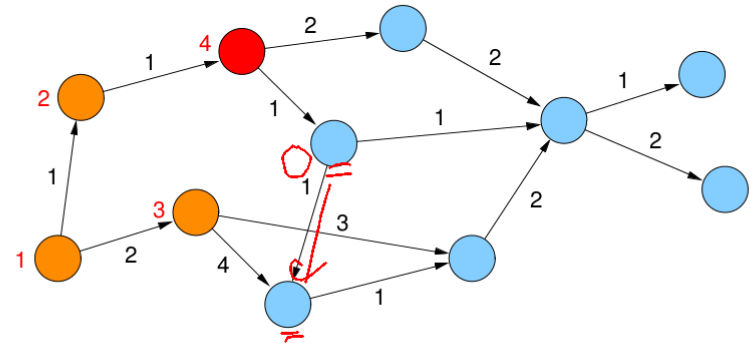
Strategie: in DAGs gibt es topologische Sortierung
 (für alle Kanten $e=(v,w)$ gilt $\text{topoNum}(v) < \text{topoNum}(w)$)



Topologische Sortierung in DAGs

Beliebige Kantengewichte in DAGs

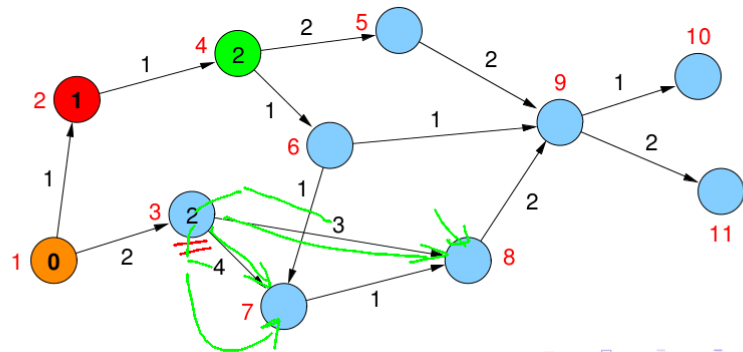
Strategie: in DAGs gibt es topologische Sortierung
 (für alle Kanten $e=(v,w)$ gilt $\text{topoNum}(v) < \text{topoNum}(w)$)



Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

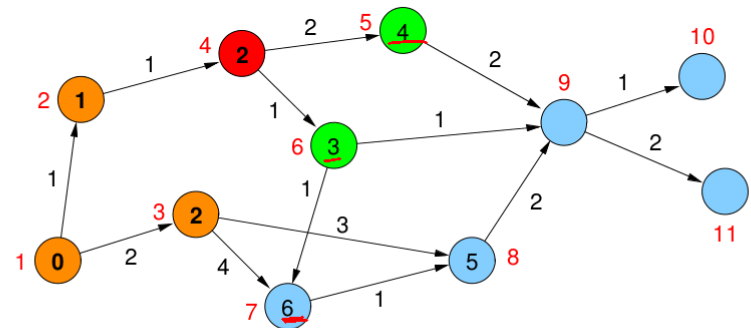
- Strategie:
- betrachte Knoten in Reihenfolge der topologischen Sortierung
 - aktualisiere Distanzwerte



Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

- Strategie:
- betrachte Knoten in Reihenfolge der topologischen Sortierung
 - aktualisiere Distanzwerte

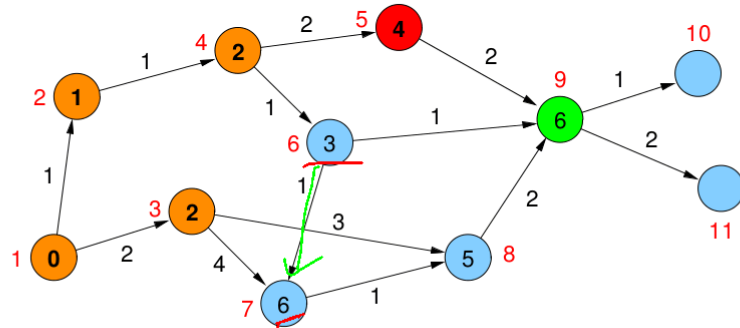


Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

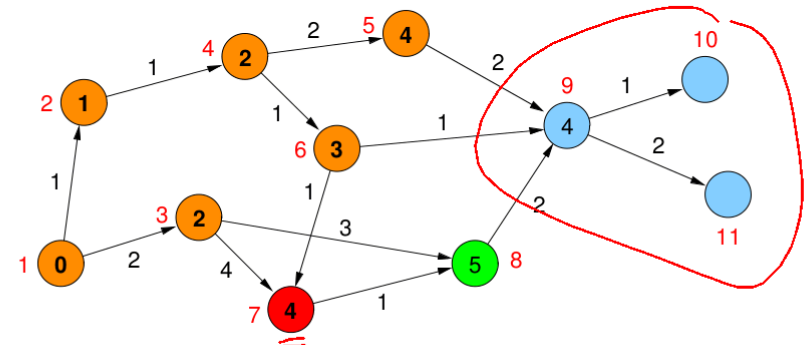


Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs

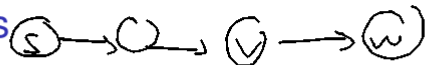
Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte



Kürzeste Wege in DAGs

Beliebige Kantengewichte in DAGs



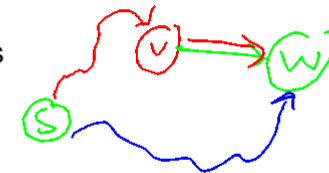
Topologische Sortierung – warum funktioniert das?

- 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 $O(n + m)$
- 2 Aktualisierung der Distanzen gemäß der topologischen Sortierung
Laufzeit $O(n + m)$

Gesamtlaufzeit: $O(n + m)$

Kürzeste Wege in DAGs

Topologische Sortierung

Korrektheit

- Knoten wird erst dann nummeriert, wenn alle Vorgänger nummeriert sind

Laufzeit

- für die Anfangswerte der Zähler muss der Graph einmal traversiert werden $O(n + m)$
 - danach wird jede Kante genau einmal betrachtet
- ⇒ gesamt: $O(n + m)$

Test auf DAG-Eigenschaft

- topologische Sortierung erfasst genau dann **alle** Knoten, wenn der Graph ein **DAG** ist
- bei gerichteten Kreisen erhalten diese Knoten keine Nummer

Beliebige Graphen mit nicht-negativen Gewichten

Gegeben:

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

Dijkstra-Algorithmus

Input : $G = (V, E)$, $c : E \rightarrow \mathbb{R}$, $s \in V$

Output : Distanzen $d(s, v)$ zu allen $v \in V$

$P = \emptyset$; $T = V$;

$d(s, v) = \infty$ for all $v \in V \setminus s$;

$d(s, s) = 0$; $pred(s) = \perp$;

while $P \neq V$ **do**

$v = \operatorname{argmin}_{v \in T} \{d(s, v)\}$;

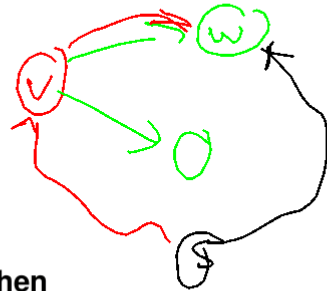
$P = P \cup v$; $T = T \setminus v$;

forall the $(v, w) \in E$ **do**

if $d(s, w) > d(s, v) + c(v, w)$ **then**

$d(s, w) = d(s, v) + c(v, w)$;

$pred(w) = v$;



Dijkstra-Algorithmus für SSSP

Input : $G = (V, E)$, $c : E \rightarrow \mathbb{R}_{\geq 0}$, $s \in V$

Output : Distanzen $d[v]$ von s zu allen $v \in V$

$d[v] = \infty$ for all $v \in V \setminus s$;

$d[s] = 0$; $pred[s] = \perp$;

$pq = \langle \rangle$; $pq.insert(s, 0)$;

while $\neg pq.empty()$ **do**

$v = pq.deleteMin()$;

forall the $(v, w) \in E$ **do**

$newDist = d[v] + c(v, w)$;

if $newDist < d[w]$ **then**

$pred[w] = v$;

if $d[w] == \infty$ **then** $pq.insert(w, newDist)$;

else $pq.decreaseKey(w, newDist)$;

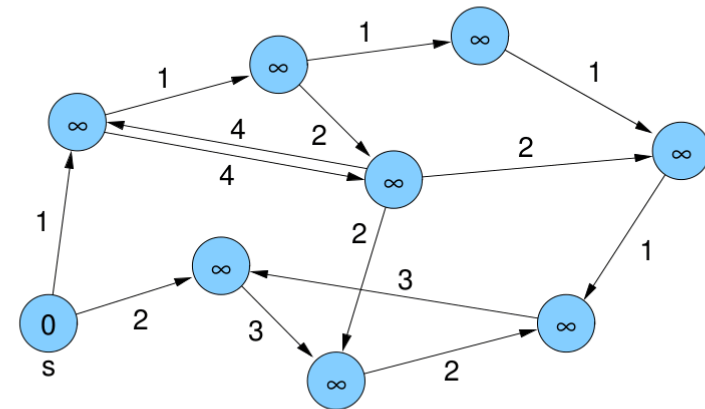
$d[w] = newDist$;

Dijkstra-Algorithmus

- setze Startwert $d(s, s) = 0$ und zunächst $d(s, v) = \infty$
- verwende **Prioritätswarteschlange**, um die Knoten zusammen mit ihren aktuellen Distanzen zu speichern
- 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

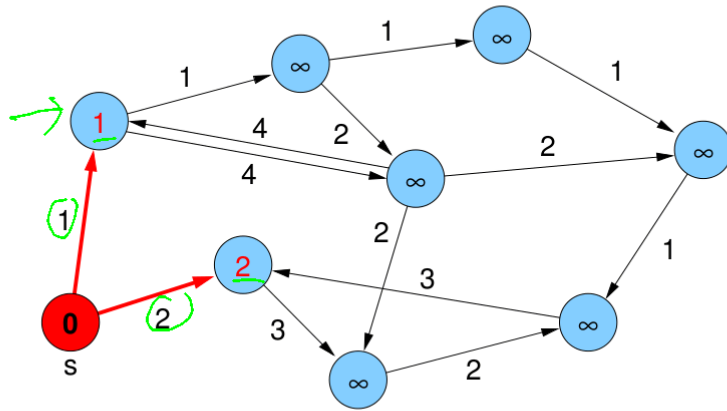
Dijkstra-Algorithmus

Beispiel:



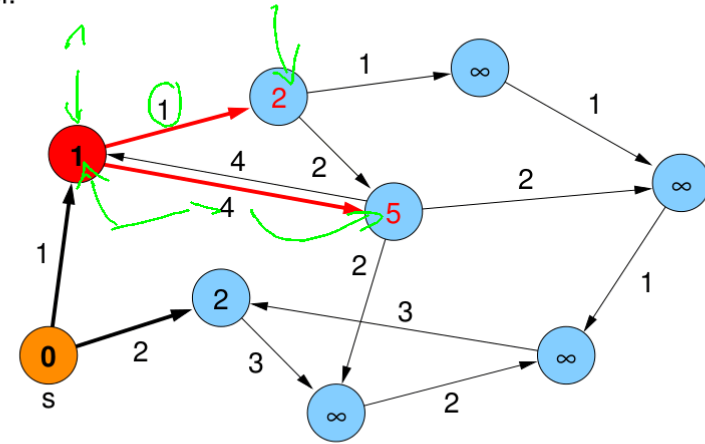
Dijkstra-Algorithmus

Beispiel:



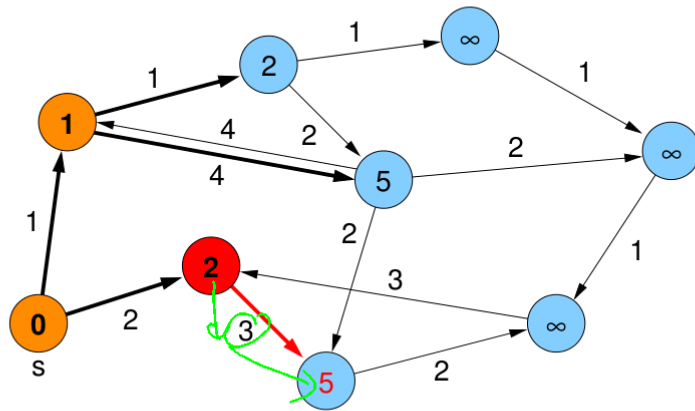
Dijkstra-Algorithmus

Beispiel:



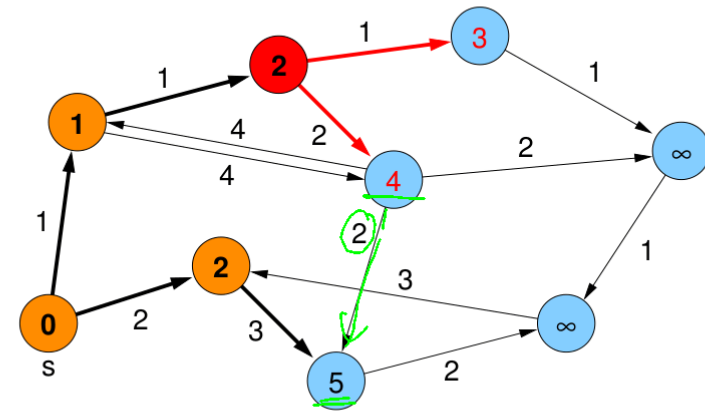
Dijkstra-Algorithmus

Beispiel:



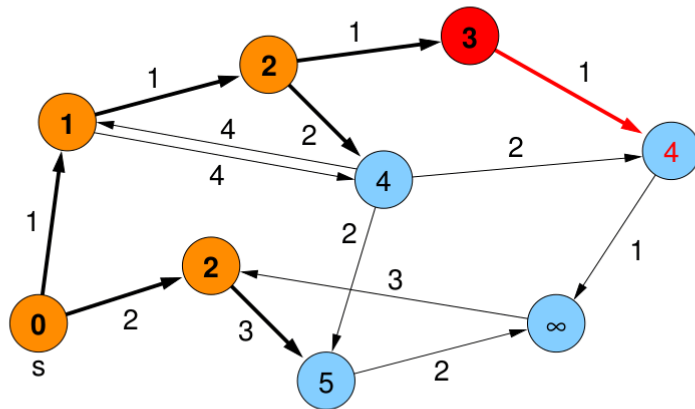
Dijkstra-Algorithmus

Beispiel:



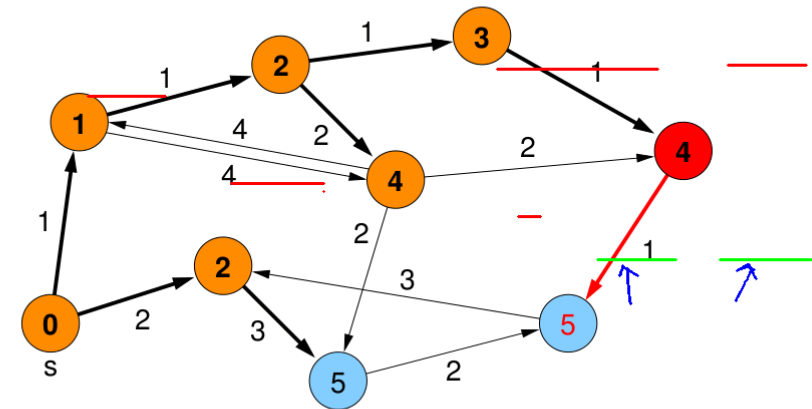
Dijkstra-Algorithmus

Beispiel:



Dijkstra-Algorithmus

Beispiel:



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
 - ⇒ $O(m + n \log n)$
- 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**

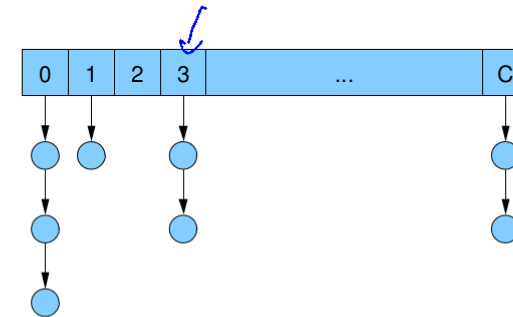
Annahme: alle **Kantengewichte** im Bereich $[0, C]$

Konsequenz für Dijkstra-Algorithmus:

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

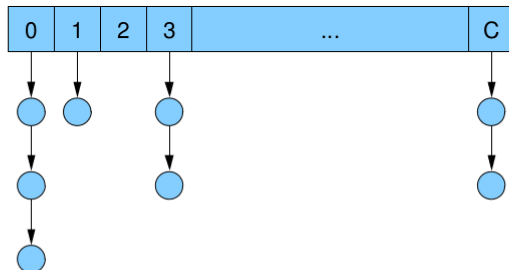
Bucket Queue

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



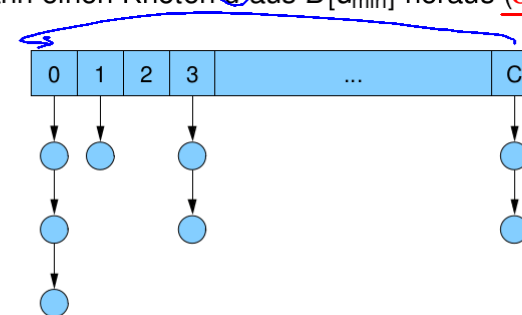
Bucket Queue

- 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:

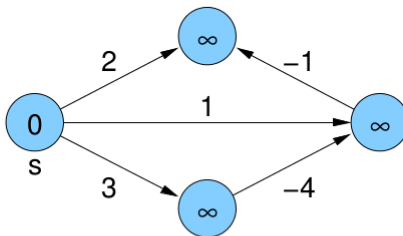
- **beliebiger** Graph mit **beliebigen** Kantengewichten
- ⇒ Anhängen einer Kante an einen Weg kann zur Verkürzung des Weges (Kantengewichtssumme) führen (wenn Kante negatives Gewicht hat)
- ⇒ es kann negative Kreise und Knoten mit Distanz $-\infty$ geben

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

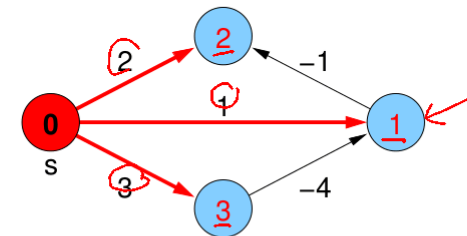
Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:



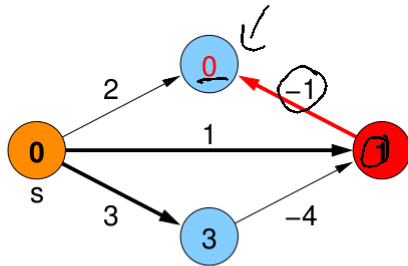
Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:



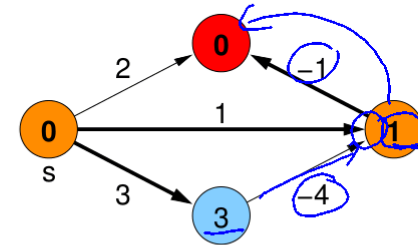
Beliebige Graphen mit beliebigen Gewichten

Gegenbeispiel für Dijkstra-Algorithmus:



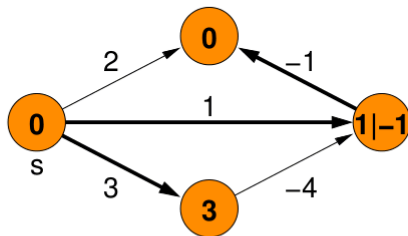
Beliebige Graphen mit beliebigen Gewichten

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 .

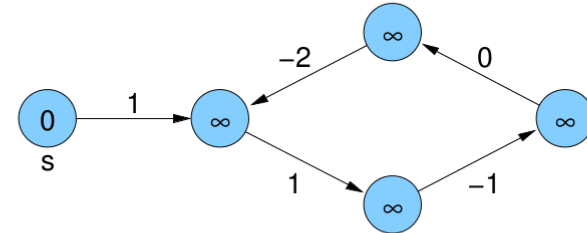
$$d[v] + c(v,w) < d[w]$$

Strategie:

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

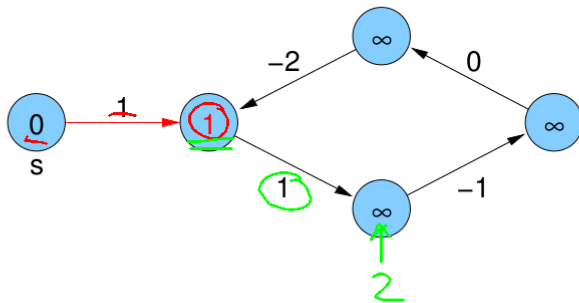
Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise



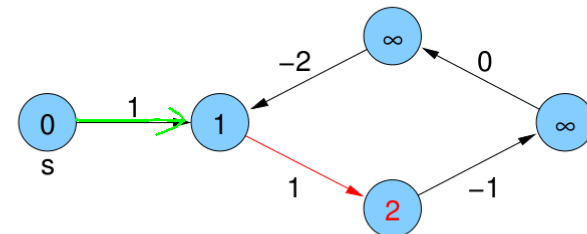
Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise



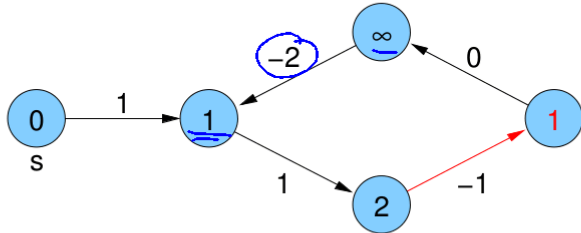
Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise



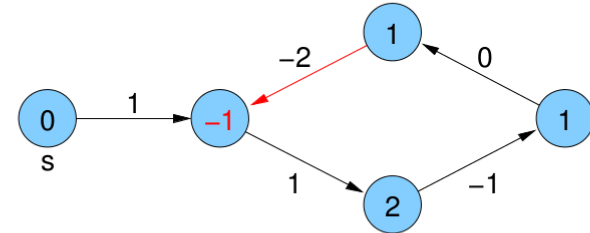
Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise



Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise



Bellman-Ford-Algorithmus

Problem: Erkennung negativer Kreise

