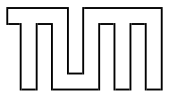
Fortgeschrittene Netzwerk- und Graph-Algorithmen

Prof. Dr. Hanjo Täubig

Lehrstuhl für Effiziente Algorithmen
(Prof. Dr. Ernst W. Mayr)
Institut für Informatik
Technische Universität München

Wintersemester 2010/11



Übersicht

Knoten- und Kanten-Stresszentralität

Lemma

In einem gerichteten Graphen $G = (V, E)$ sind die Stresszentralitäten von Knoten und Kanten wie folgt miteinander verknüpft:

$$c_S(v) = \frac{1}{2}\left(\sum_{e \in \Gamma(v)} c_S(e) - \sum_{s \in V,\, s \neq v} \sigma_{sv} - \sum_{t \in V,\, t \neq v} \sigma_{vt}\right)$$

$\Gamma(v)$: Menge der Kanten mit Endknoten v

σ_{xy}: Anzahl kürzester Wege von x nach y

Knoten- und Kanten-Stresszentralität

Beweis.

- Betrachte einen kürzesten Pfad zwischen $s, t \in V$.

$\Rightarrow$ trägt genau 1 zum Stress jedes inneren Knotens und jeder Kante bei

- Innere Knoten sind jeweils zu zwei Kanten des Pfads benachbart.
- Anfangs- und Endknoten sind immer zu einer Kante des Pfads benachbart. (Während dieser kürzeste Pfad dann für die Kante zählt, soll er beim Anfangs-/Endknoten nicht mitgezählt werden.)

- Wie oft ist v Anfangs- bzw. Endknoten eines kürzesten Pfades?

$$\sum_{s \in V, s \neq v} \sigma_{sv} + \sum_{t \in V, t \neq v} \sigma_{vt}$$

Shortest-Path Betweenness

- eine Art relative Stress-Zentralität
- Anzahl kürzester Pfade zwischen $s, t \in V$: σ_{st}
- Anzahl kürzester Pfade zwischen s, t, die v enthalten: $\sigma_{st}(v)$
- Anteil der kürzesten Pfade zwischen s und t, die v enthalten:

$$\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$$

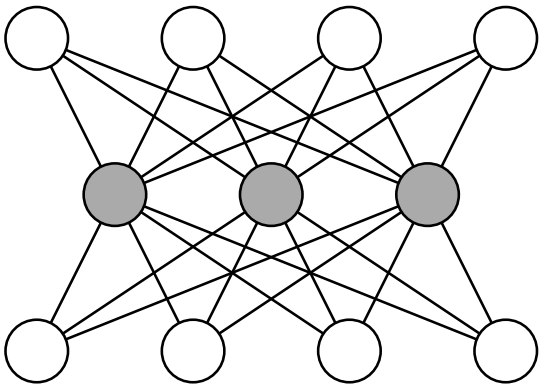
$\Rightarrow$ interpretiert als Anteil oder Wahrscheinlichkeit der Kommunikation

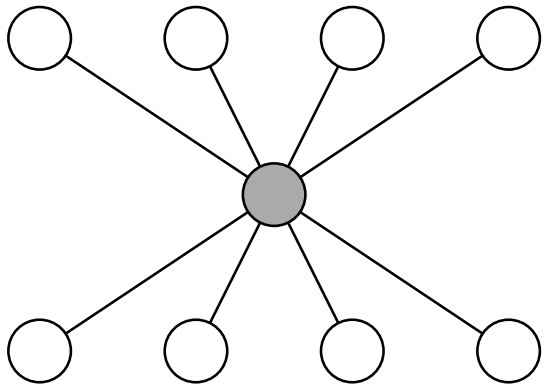
- Betweenness-Zentralität:

$$c_B(v) = \sum_{s \neq v \in V} \sum_{t \neq v \in V} \delta_{st}(v)$$

Vorteile von Betweenness

- Vorteil gegenüber Closeness:
  funktioniert auch bei nicht zusammenhängenden Graphen
- Vorteil gegenüber Stress (siehe Abb.): Knoten der mittleren Schicht haben in beiden Graphen gleiche Stresszentralität
  links: mehrere Alternativen
  rechts: mittlerer Knoten für gesamte Kommunikation notwendig





- Normierung: Teilen durch Anzahl Paare
  $(n-1)(n-2)$ bzw. $(n-1)(n-2)/2$

Betweenness für Kanten

- Anteil der Kante e am Informationsfluss (auf allen kürzesten Pfaden) zwischen den Knoten s und t

$$\delta_{st}(e) = \frac{\sigma_{st}(e)}{\sigma_{st}}$$

- Betweenness-Zentralität der Kante e:

$$c_B(e) = \sum_{s \in V} \sum_{t \in V} \delta_{st}(e)$$

Knoten- und Kanten-Betweenness-Zentralität

Lemma

In einem gerichteten Graphen $G = (V, E)$ ist die (Kürzeste-Pfade-)Betweenness-Zentralität für Knoten und Kanten wie folgt miteinander verknüpft:

$$c_B(v) = \left(\sum_{e \in \Gamma^+(v)} c_B(e) \right) - (n-1) = \left(\sum_{e \in \Gamma^-(v)} c_B(e) \right) - (n-1)$$

$\Gamma^+(v)$: Menge der Kanten mit Startknoten v
$\Gamma^-(v)$: Menge der Kanten mit Zielknoten v

Knoten- und Kanten-Betweenness-Zentralität

Beweis.

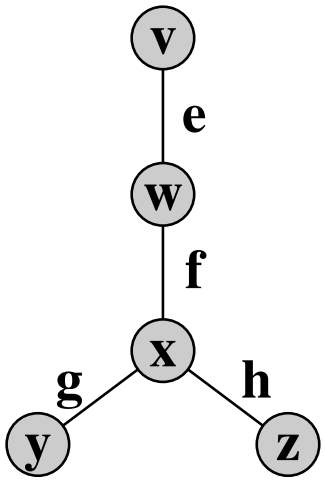
- Betrachte einen kürzesten Pfad zwischen $s, t \in V$.

$\Rightarrow$ trägt genau $1/\sigma_{st}$ zur Betweenness jedes inneren Knotens und jeder Kante bei

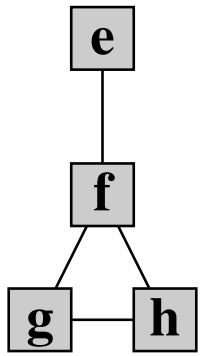
- Innere Knoten sind jeweils zu einer eingehenden und einer ausgehenden Kante des Pfads benachbart.
- Anfangs- und Endknoten sind immer zu genau einer Kante des Pfads benachbart. (Während dieser kürzeste Pfad dann $1/\sigma_{st}$ zur Betweenness der Kante beiträgt, soll er beim Anfangs-/Endknoten nicht mitgezählt werden.)
- Insgesamt summieren sich die Anteile eines Knotens als Anfangs- oder Endknoten zu $(n-1)$, die also von der Summe der angrenzenden Kanten abzuziehen sind.

Abgeleitete Kantenzentralitäten

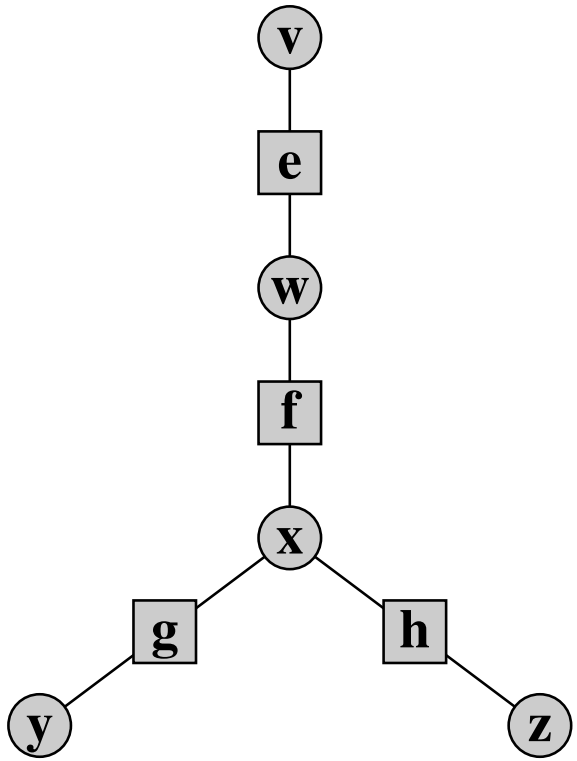
Wie kann man aus Knotenzentralitäten automatisch Definitionen von Kantenzentralität ableiten?



Graph



Kantengraph



Inzidenzgraph

Kantenzentralität als Knotenzentralität im Kantengraph

Definition

Kantengraph von Graph $G = (V, E)$ ist $G' = (E, K)$, wobei K die Menge der Kanten $e = ((x, y), (y, z))$ mit $(x, y) \in E$ und $(y, z) \in E$ ist.

Zwei Knoten im Kantengraph sind also benachbart, wenn die entsprechenden Kanten im ursprünglichen Graphen einen Knoten gemeinsam haben (im gerichteten Fall als Zielknoten der einen Kante und Startknoten der anderen).

$\Rightarrow$ Wende Knotenzentralität auf den Kantengraph an
Nachteile:

- Größe des Kantengraphen kann quadratisch in der Größe des ursprünglichen Graphen sein,
- Kantengewichte in G werden zu Knotengewichten in G' (Nutzung unklar)
- keine natürliche Interpretation / Generalisierung

Kantenzentralität als Knotenzentralität im Inzidenzgraph

Definition

Inzidenzgraph von Graph $G = (V, E)$ ist $G'' = (V'', E'')$, wobei

$$V'' = V \cup E \quad \text{und} \quad E'' = \{(v, e) \mid \exists w : e = (v, w) \in E\} \cup \{(e, w) \mid \exists v : e = (v, w) \in E\}.$$

Im Inzidenzgraphen sind also ein 'echter Knoten' und ein 'Kantenknoten' benachbart, wenn der entsprechende Knoten und die entsprechende Kante im ursprünglichen Graphen inzident sind.

$\Rightarrow$ Wende Knotenzentralität auf den Inzidenzgraph an, wobei nur die Pfade zwischen 'echten Knoten' als relevant betrachtet werden

Vitalität

Vitalitätsmaße bewerten die Wichtigkeit eines Knotens oder einer Kante anhand eines Qualitätsverlusts durch das Löschen des Knotens bzw. der Kante.

Definition (Vitalitätsindex)

Sei $\mathcal{G}$ die Menge aller einfachen ungerichteten ungewichteten Graphen $G = (V, E)$ und $f : \mathcal{G} \to \mathbb{R}$ eine reellwertige Funktion auf $G \in \mathcal{G}$.
Dann ist ein Vitalitätsindex $\mathcal{V}(G, x)$ definiert als die Differenz der Werte von f auf G und $G \setminus \{x\}$:

$$\mathcal{V}(G, x) = f(G) - f(G \setminus \{x\})$$

Flow Betweenness Vitality

- ähnlich zu Shortest Path Betweenness
- Motivation: Information in einem Kommunikationsnetzwerk muss sich nicht unbedingt auf kürzesten Pfaden bewegen.
- Maß: Abhängigkeit des maximalen Flusses zwischen zwei Knoten von der Existenz des betrachteten Knotens

Flow Betweenness Vitality

- f_{st}: Maximum-Fluss zwischen Knoten s und t unter Berücksichtigung der Kantenkapazitäten
- $f_{st}(v)$: Fluss zwischen Knoten s und t, der bei Maximum-Fluss von s nach t durch v gehen muss
- $\tilde{f}_{st}(v)$: Maximum-Fluss von s nach t in $G - v$

$$c_{mf}(v) = \sum_{\substack{s,t \in V \\ v \neq s, v \neq t \\ f_{st} > 0}} \frac{f_{st}(v)}{f_{st}} = \sum_{\substack{s,t \in V \\ v \neq s, v \neq t \\ f_{st} > 0}} \frac{f_{st} - \tilde{f}_{st}(v)}{f_{st}}$$

Closeness Vitality

- Wiener Index:

$$I_W(G) = \sum_{v \in V} \sum_{w \in V} d(v, w)$$

- oder in Abhängigkeit der Closeness-Zentralitäten:

$$I_W(G) = \sum_{v \in V} \frac{1}{c_C(v)}$$

- Closeness Vitality für Knoten/Kante x:

$$c_{CV}(x) = I_W(G) - I_W(G - x)$$

- Interpretation: Um wieviel steigen die Gesamtkommunikationskosten, wenn jeder Knoten mit jedem kommuniziert?

Closeness Vitality - Durchschnitt

- Durchschnittliche Distanz:

$$\overline{d}(G) = \frac{I_W(G)}{n(n-1)}$$

- Das Vitalitätsmaß auf der Basis $f(G) = \overline{d}(G)$ misst die durchschnittliche Verschlechterung der Entfernung zwischen zwei Knoten beim Entfernen eines Knotens / einer Kante x.

- Vorsicht! Beim Entfernen eines Artikulationsknotens (cut vertex) oder einer Brücke (cut edge) x ist $c_{CV}(x) = -\infty$.

Shortcut-Werte

- kein echter Vitalitätsindex im Sinne der Definition
- maximale Erhöhung eines Distanzwerts, wenn Kante $e = (v, w)$ entfernt wird

$\Rightarrow$ nur relevant für Knoten, bei denen alle kürzesten Pfade über e laufen

- maximale Erhöhung tritt direkt zwischen Knoten v und w auf
- alternativ: maximale relative Erhöhung
- Berechnung: mit $m = |E|$ Single Source Shortest Path Instanzen (und Vergleich mit der jeweiligen Kante)
- später: mit $n = |V|$ SSSP-Bäumen
- Anwendung auch auf Knotenlöschungen möglich

Stress-Zentralität als Vitalitätsindex

- Stress-Zentralität zählt die Anzahl der kürzesten Pfade, an denen ein Knoten oder eine Kante beteiligt ist

$\Rightarrow$ kann als Vitalitätsmaß betrachtet werden

- Aber: Anzahl der kürzesten Pfade kann sich durch Löschung erhöhen (wenn sich die Distanz zwischen zwei Knoten erhöht)

$\Rightarrow$ Längere kürzeste Pfade müssen ignoriert werden

Stress-Zentralität als Vitalitätsindex

- $f(G \setminus \{v\})$ muss ersetzt werden durch

$$\sum_{s \in V} \sum_{t \in V} \sigma_{st}[d_G(s,t) = d_{G \setminus \{v\}}(s,t)]$$

- Ausdruck in Klammern ist 1 (wahr) oder 0 (falsch)
- $f(G \setminus \{e\})$ analog:

$$\sum_{s \in V} \sum_{t \in V} \sigma_{st}[d_G(s,t) = d_{G \setminus \{e\}}(s,t)]$$

- Also: $f(G) - f(G - x) = \sum_{s \in V} \sum_{t \in V} \sigma_{st}(x)$
- kein echter Vitalitätsindex im Sinne der Definition, weil der Index-Wert nach der Löschung von der Distanz vor der Löschung abhängt

Übersicht

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:

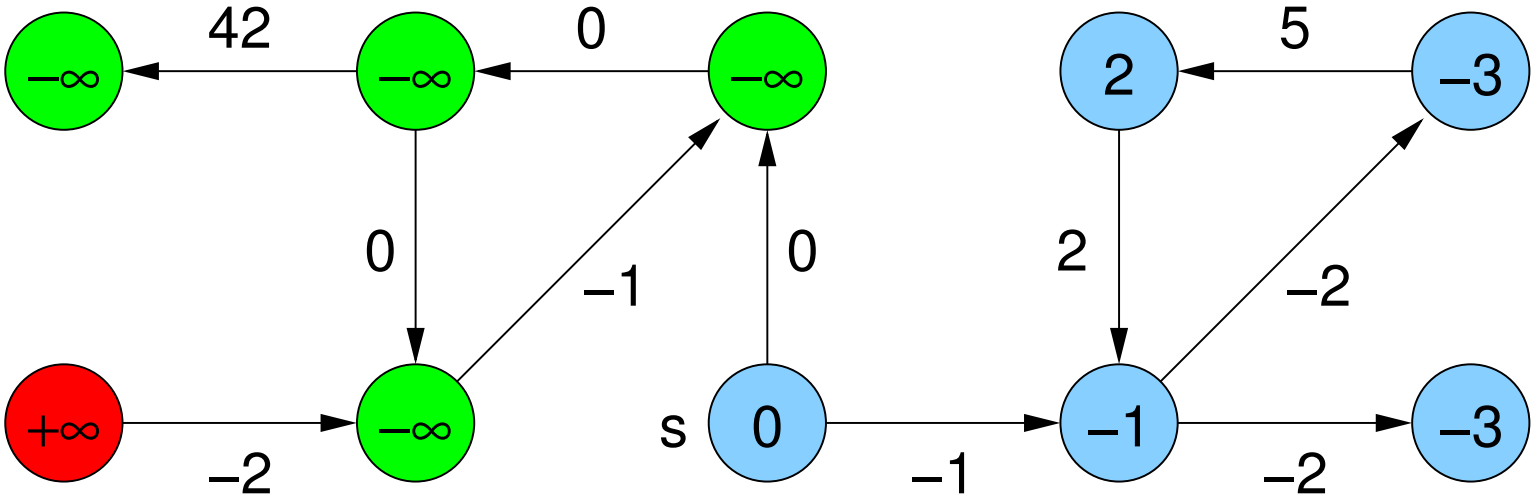
- SSSP (single source shortest paths):

  kürzeste Wege von einer Quelle zu allen anderen Knoten

- APSP (all pairs shorstest paths):

  kürzeste Wege zwischen allen Paaren

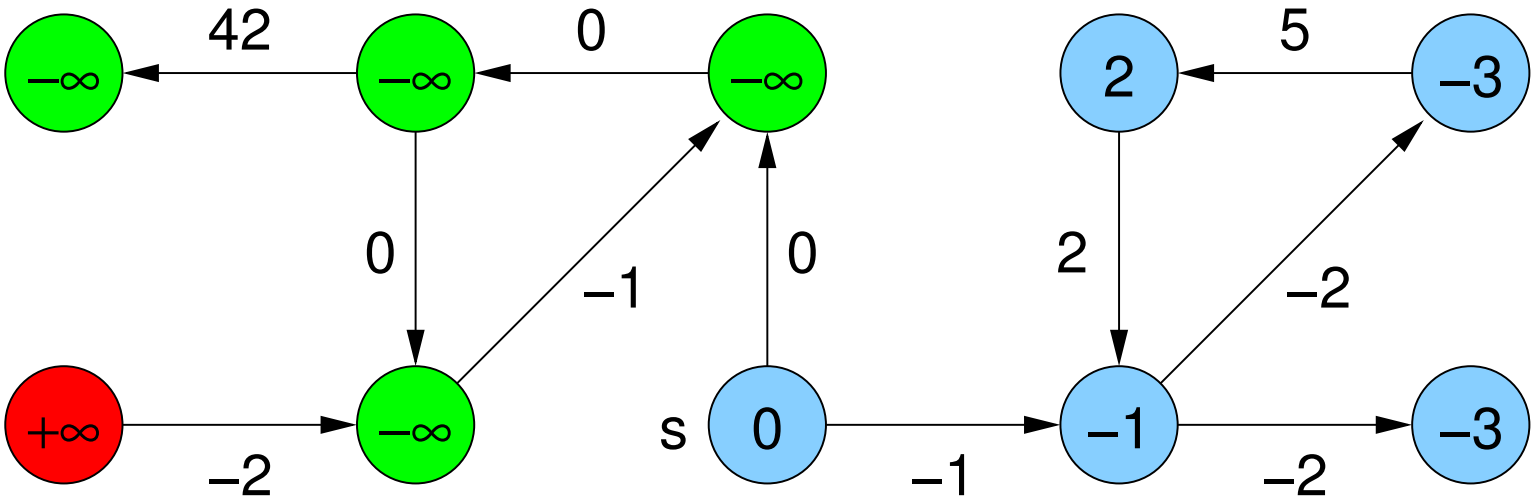
Distanzen



$\mu(s, v)$: 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



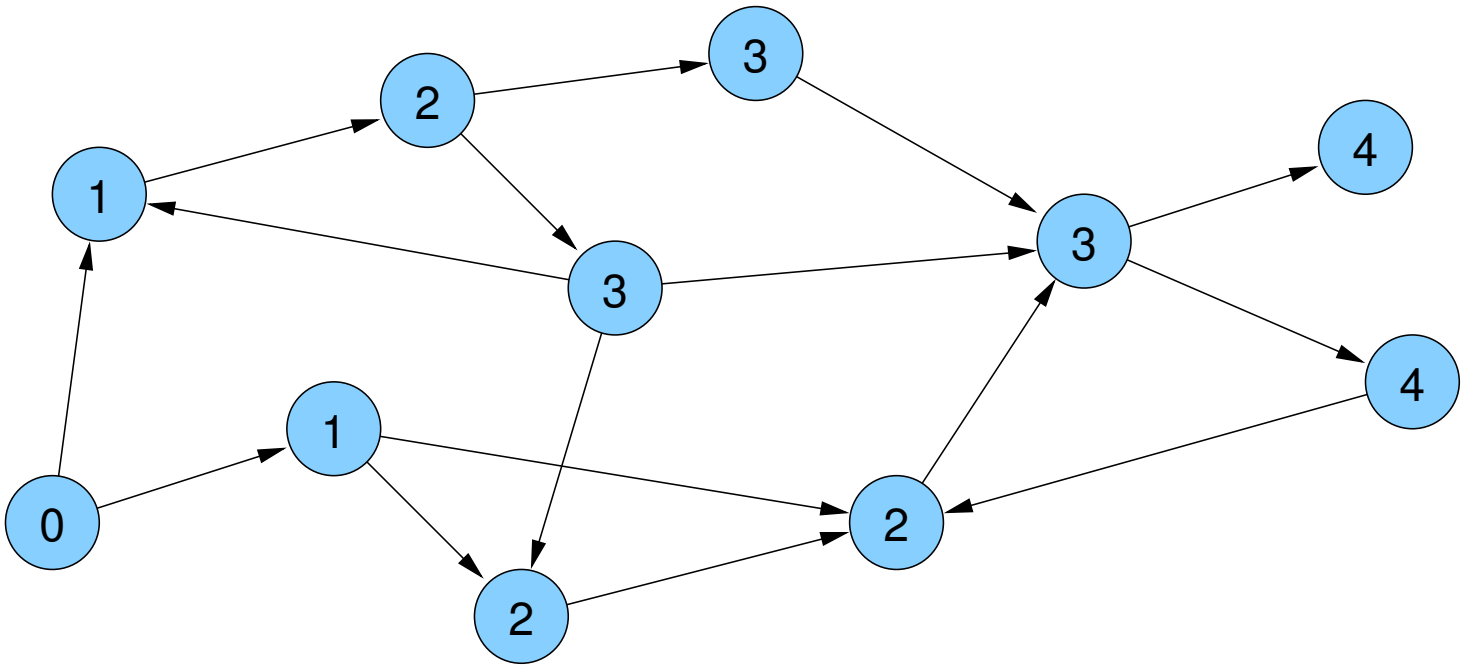
Wann sind die Kosten $-\infty$?

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

Kürzeste Wege

Graph mit Kantenkosten 1:

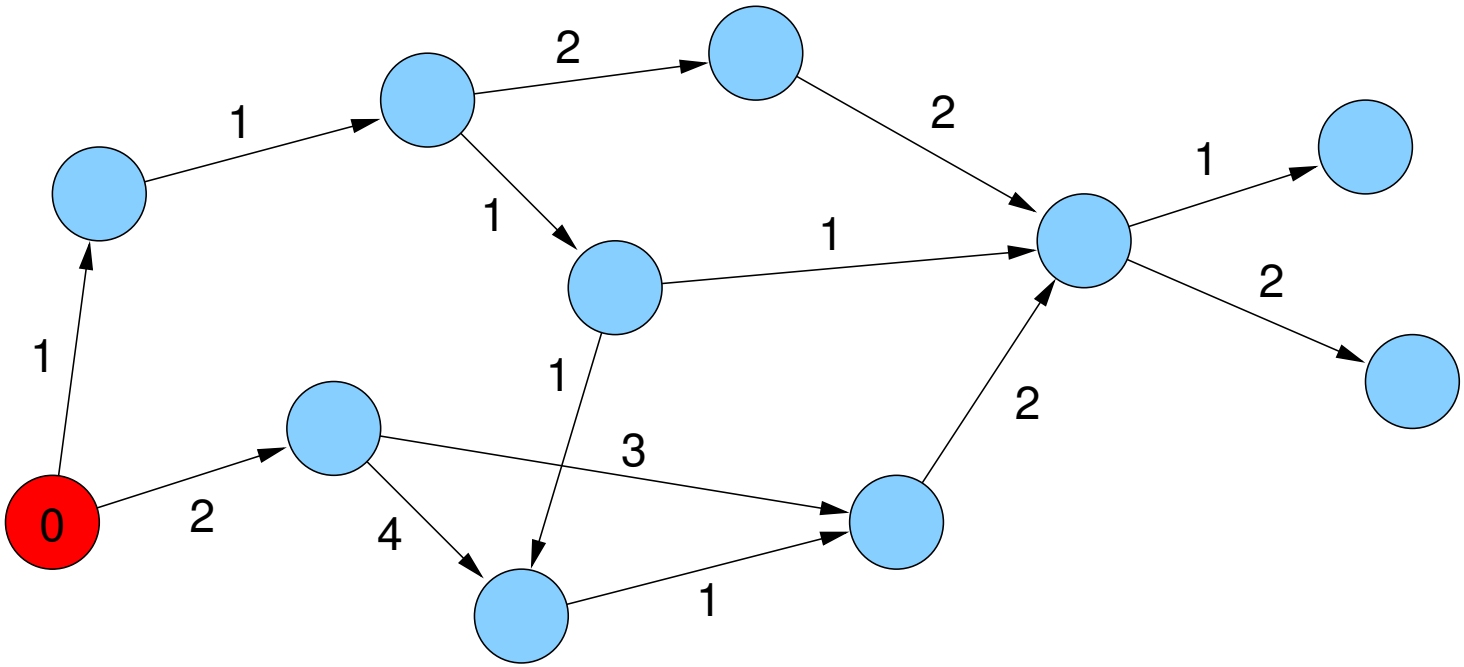
$\Rightarrow$ Breitensuche (BFS)



Kürzeste Wege

Beliebige Kantengewichte in DAGs

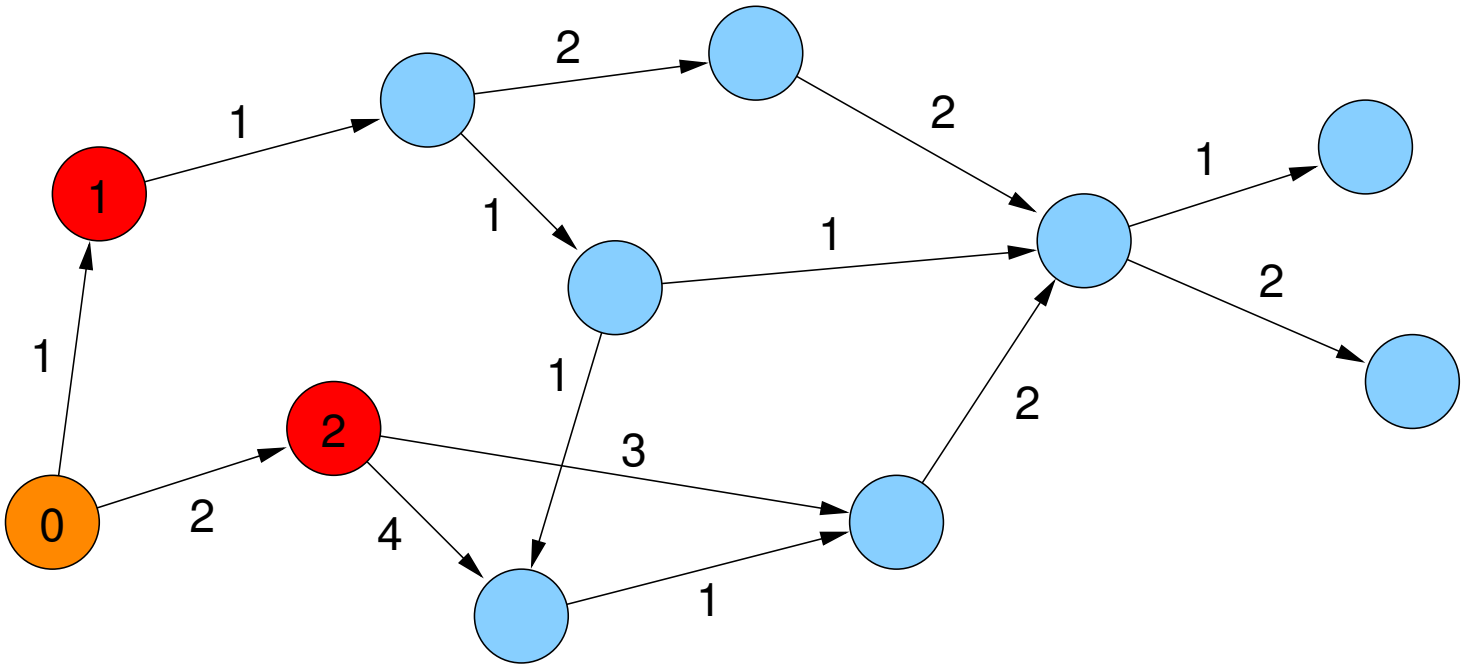
Einfache Breitensuche funktioniert nicht.



Kürzeste Wege

Beliebige Kantengewichte in DAGs

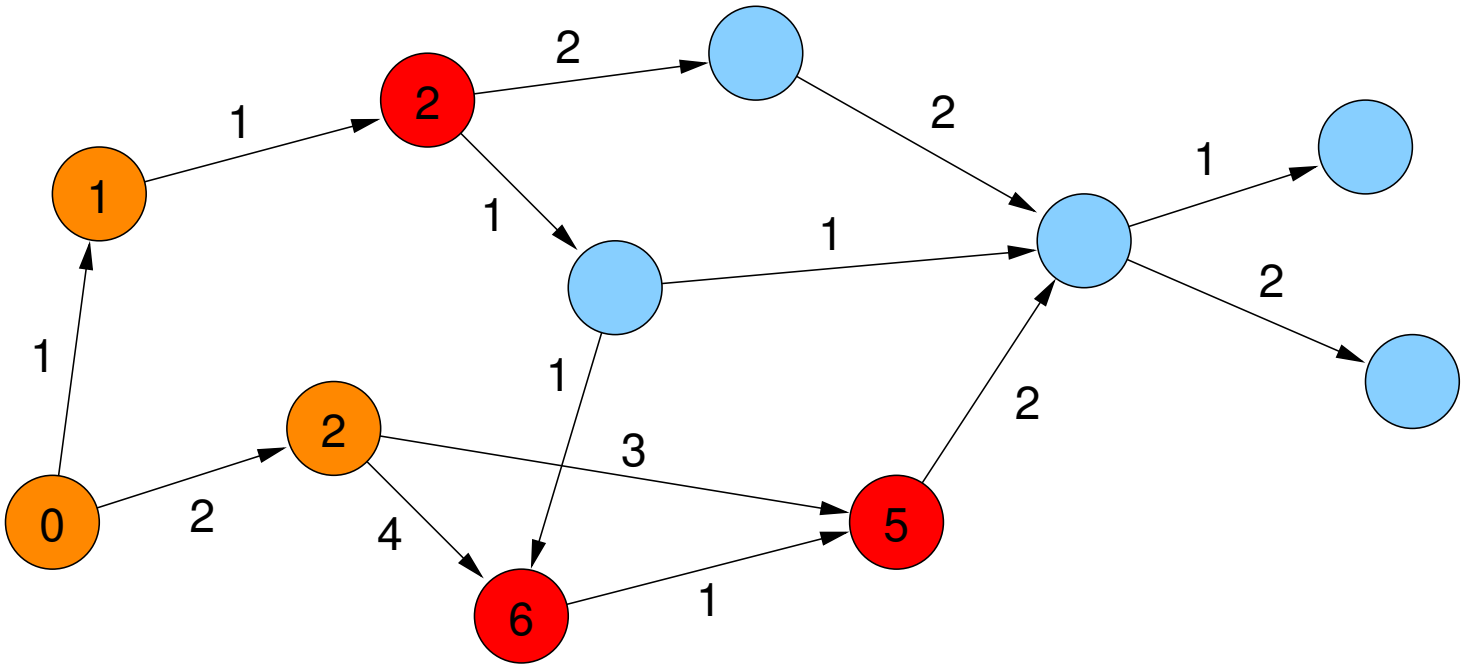
Einfache Breitensuche funktioniert nicht.



Kürzeste Wege

Beliebige Kantengewichte in DAGs

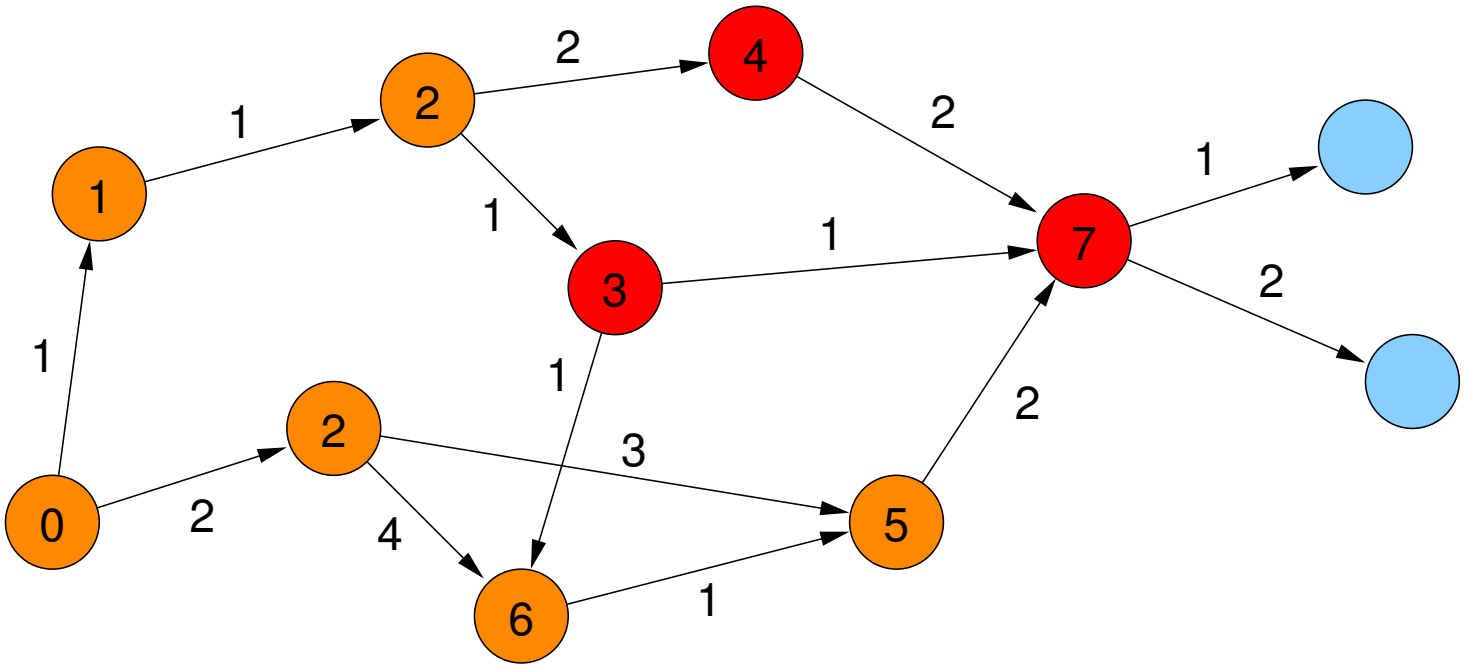
Einfache Breitensuche funktioniert nicht.



Kürzeste Wege

Beliebige Kantengewichte in DAGs

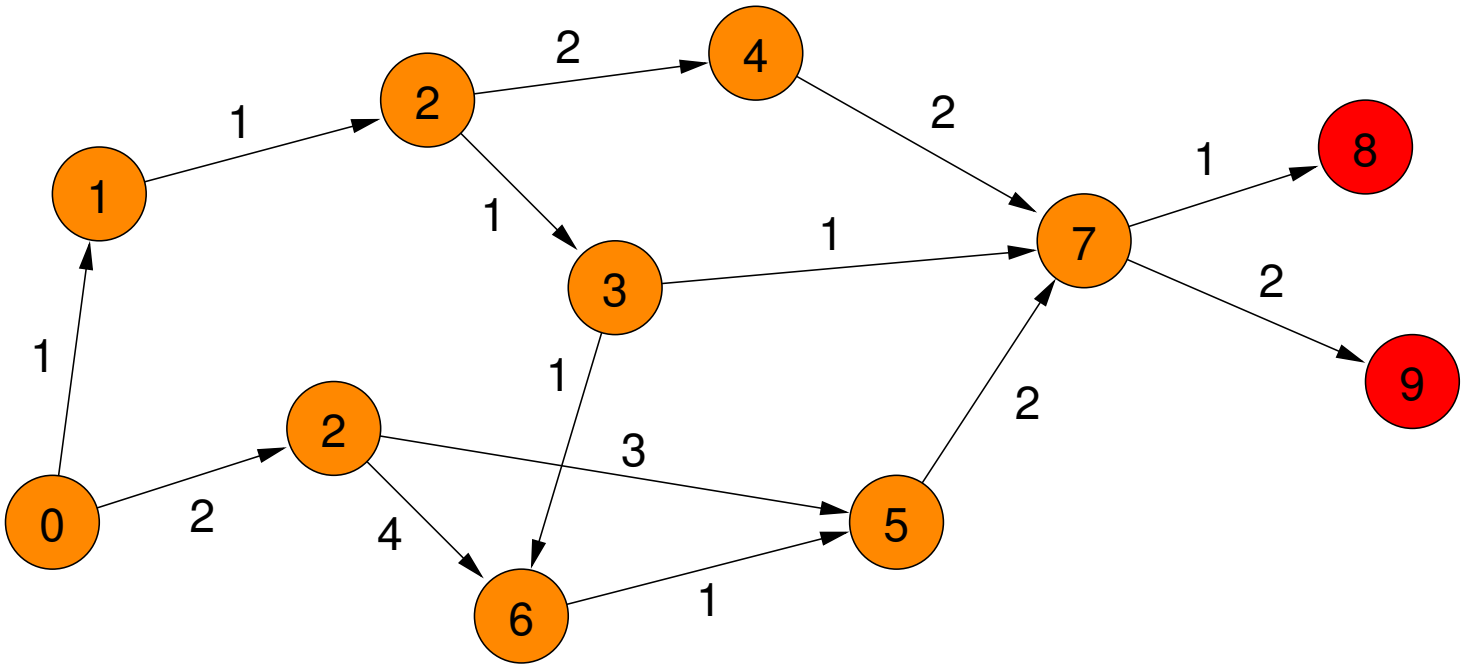
Einfache Breitensuche funktioniert nicht.



Kürzeste Wege

Beliebige Kantengewichte in DAGs

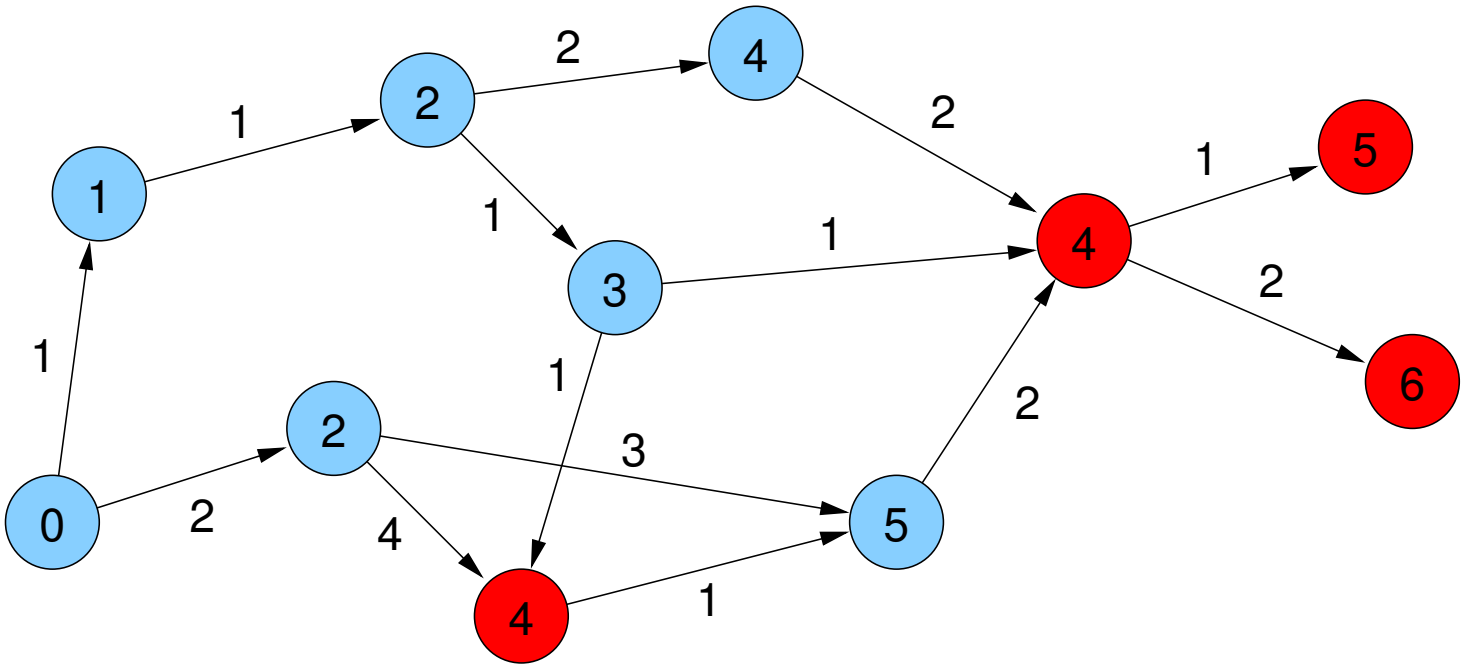
Einfache Breitensuche funktioniert nicht.



Kürzeste Wege

Beliebige Kantengewichte in DAGs

Einfache Breitensuche funktioniert nicht.

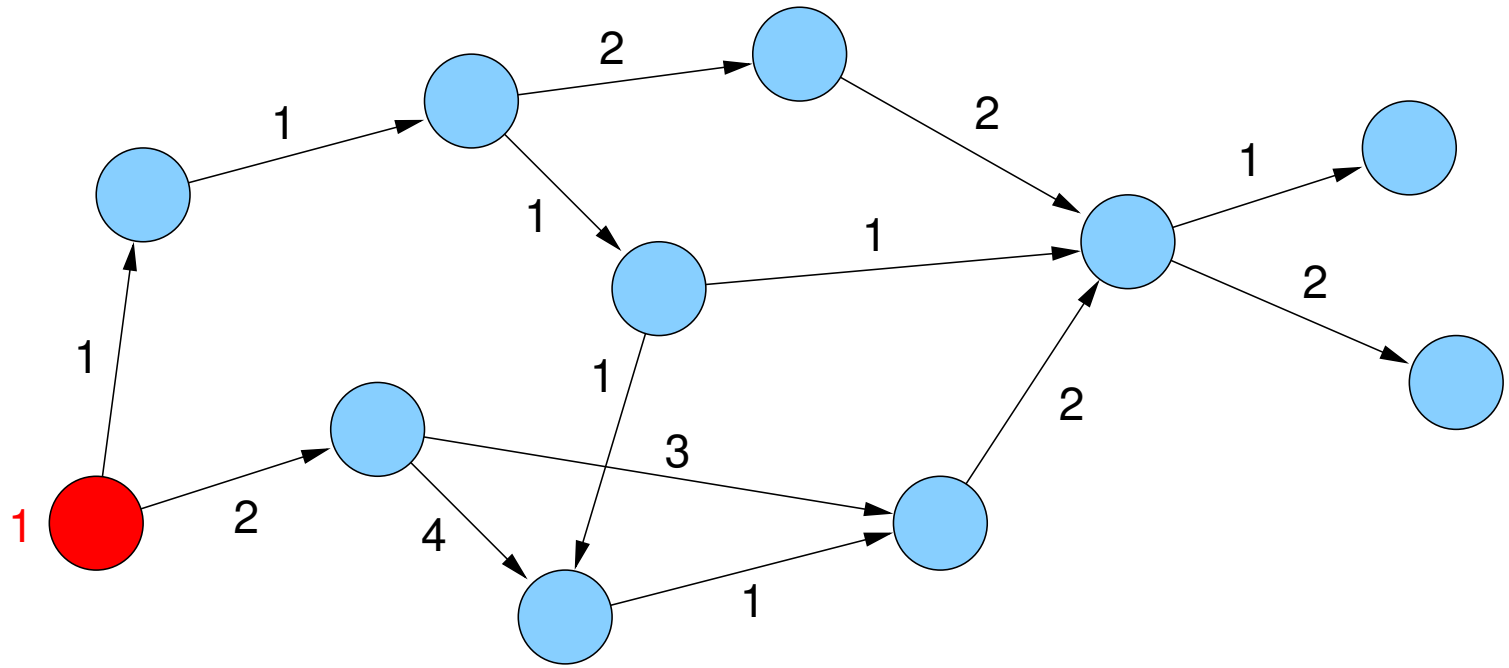


Kürzeste Wege

Beliebige Kantengewichte in DAGs

Strategie: in DAGs gibt es topologische Sortierung

(für alle Kanten e=(v,w) gilt topoNum(v) $<$ topoNum(w))

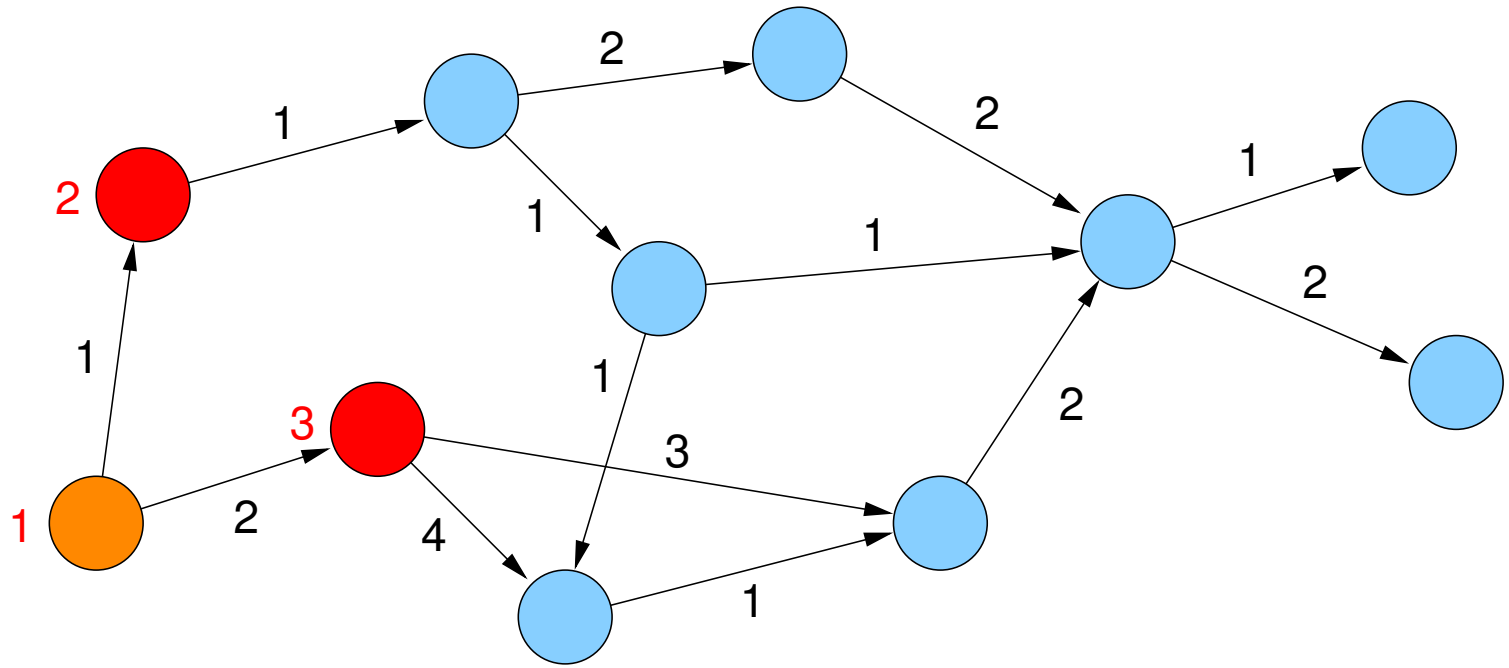


Kürzeste Wege

Beliebige Kantengewichte in DAGs

Strategie: in DAGs gibt es topologische Sortierung

(für alle Kanten e=(v,w) gilt topoNum(v) $<$ topoNum(w))

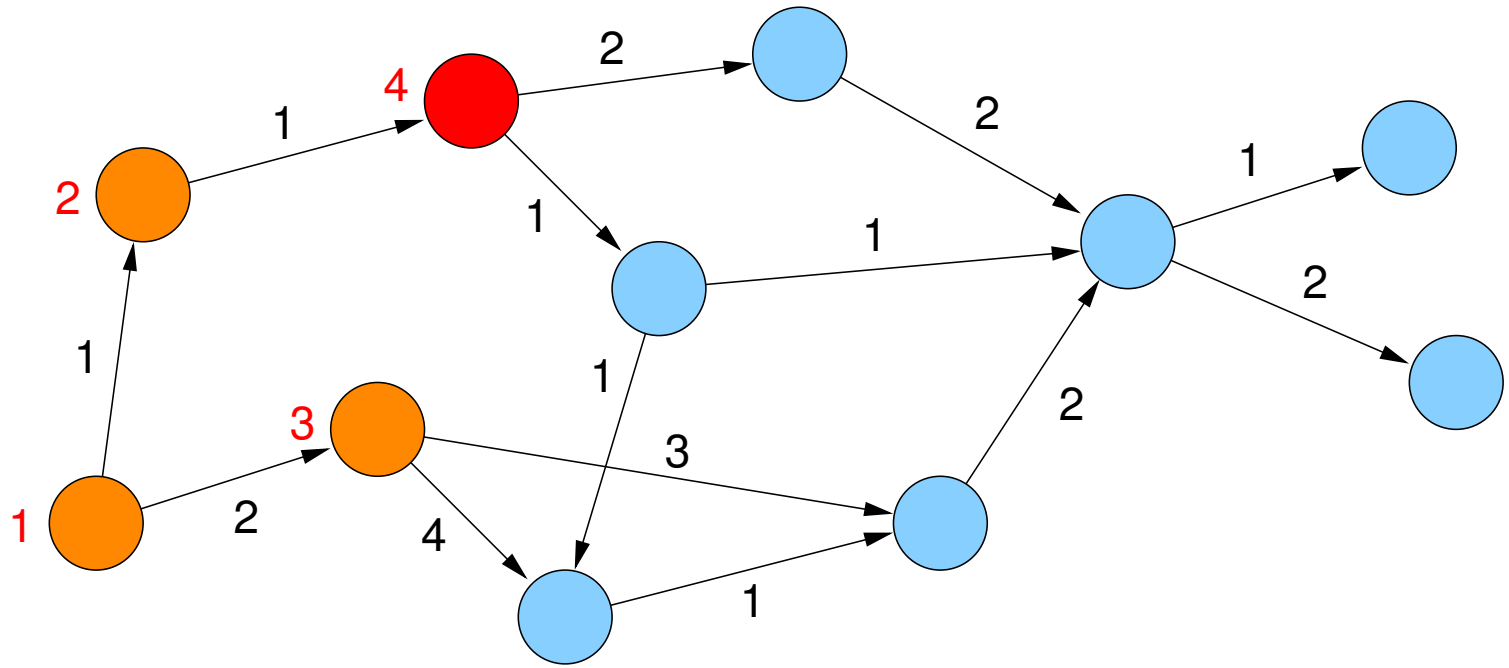


Kürzeste Wege

Beliebige Kantengewichte in DAGs

Strategie: in DAGs gibt es topologische Sortierung

(für alle Kanten e=(v,w) gilt topoNum(v) $<$ topoNum(w))

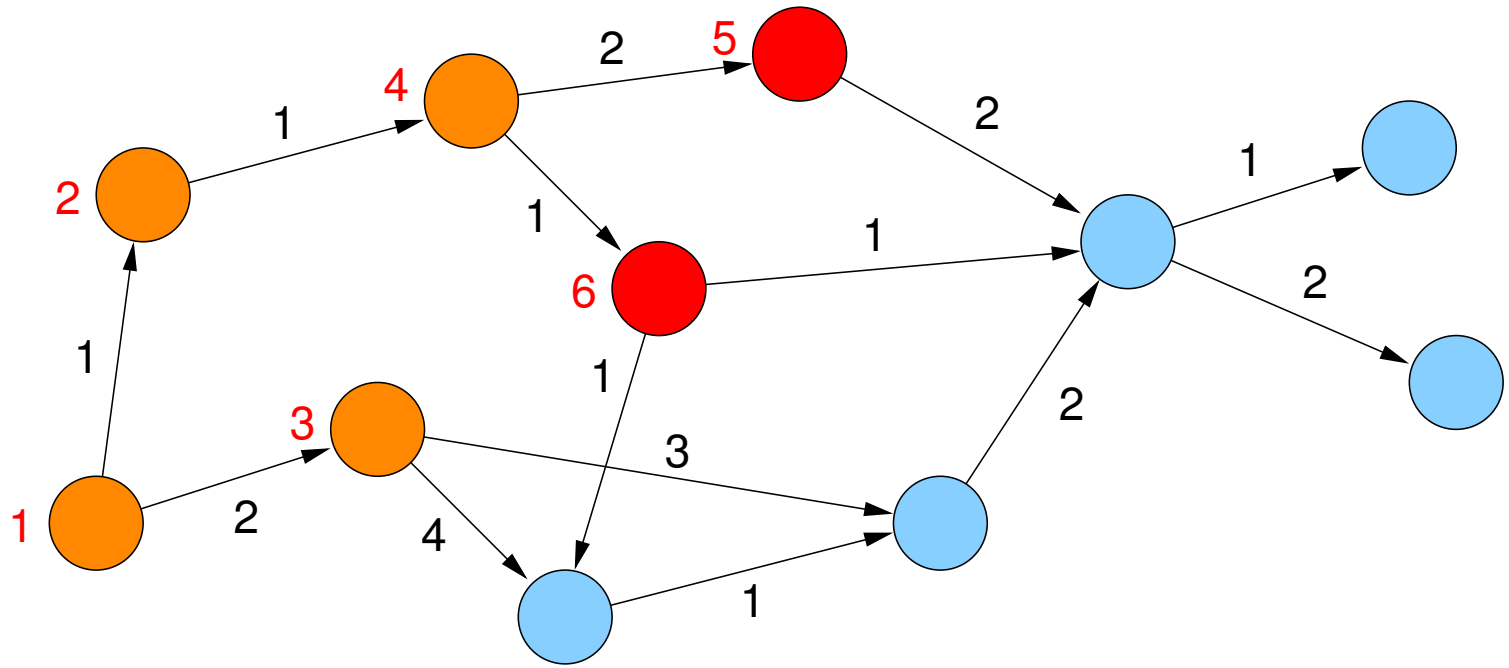


Kürzeste Wege

Beliebige Kantengewichte in DAGs

Strategie: in DAGs gibt es topologische Sortierung

(für alle Kanten e=(v,w) gilt topoNum(v) $<$ topoNum(w))

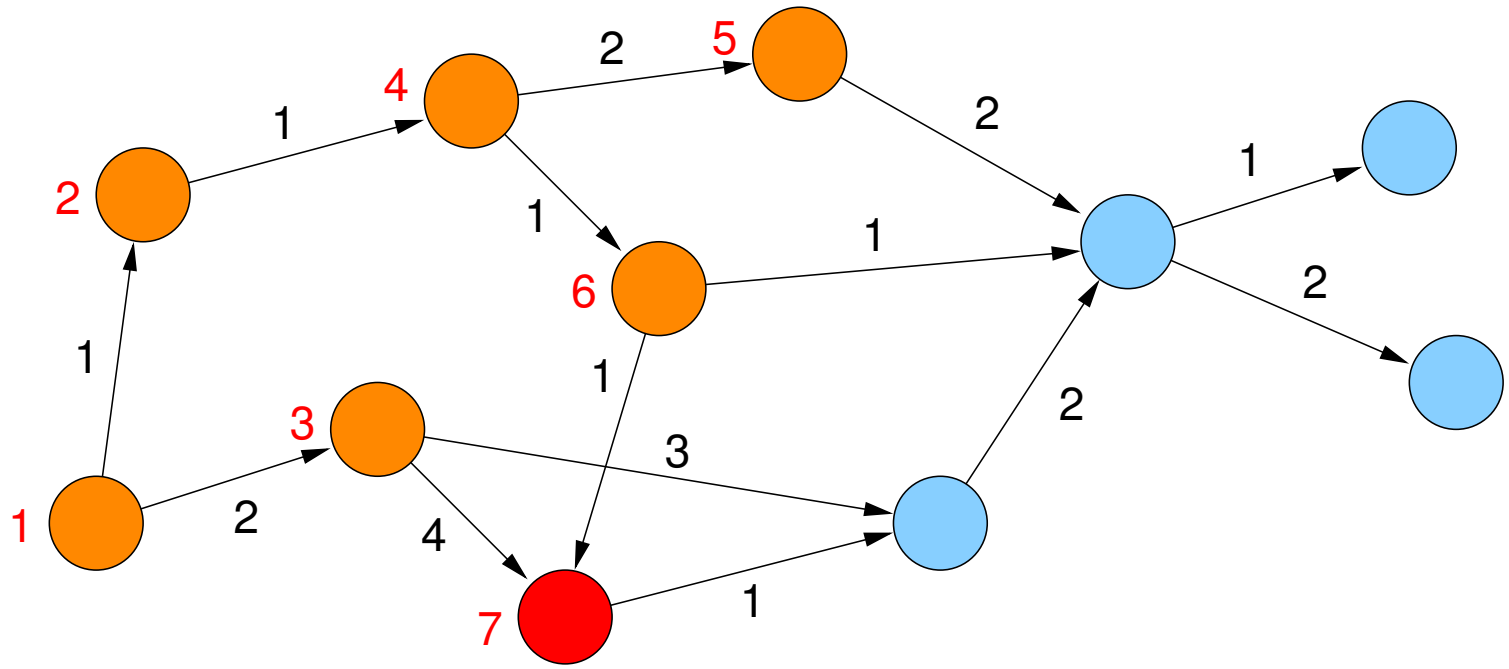


Kürzeste Wege

Beliebige Kantengewichte in DAGs

Strategie: in DAGs gibt es topologische Sortierung

(für alle Kanten e=(v,w) gilt topoNum(v) $<$ topoNum(w))

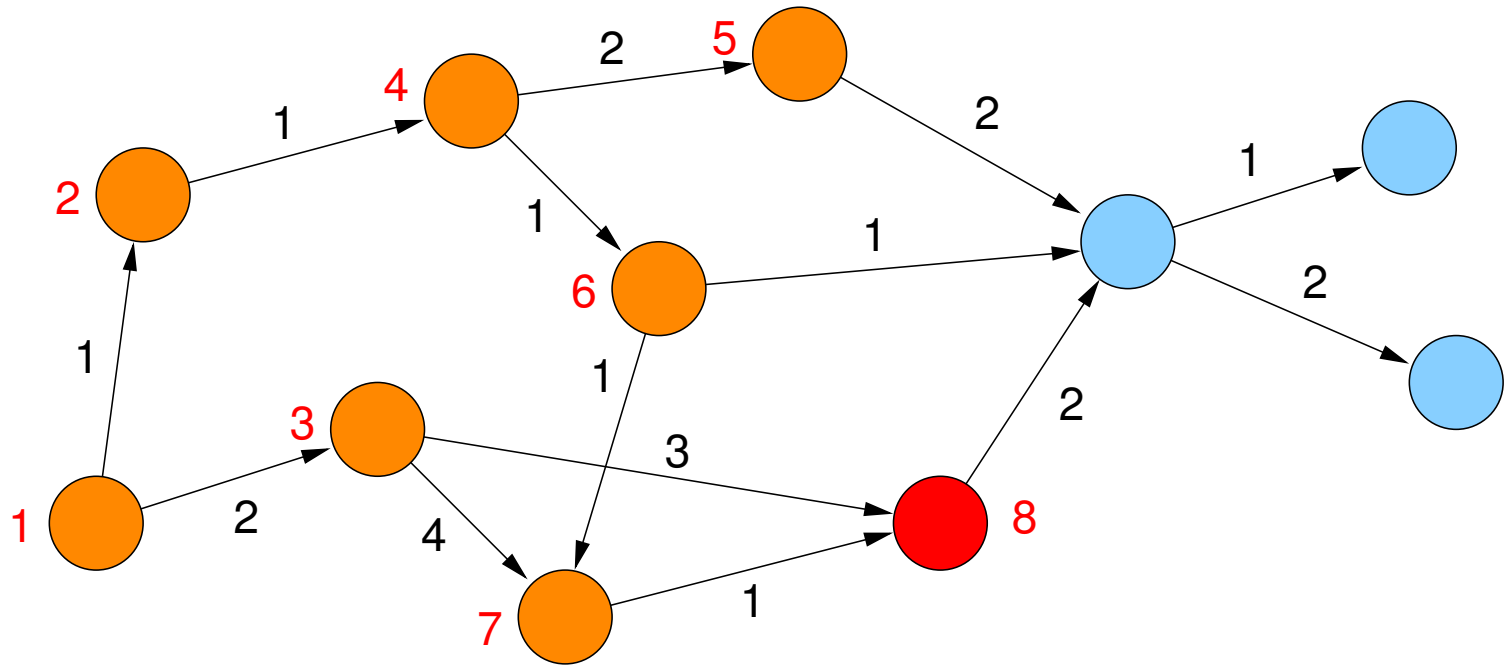


Kürzeste Wege

Beliebige Kantengewichte in DAGs

Strategie: in DAGs gibt es topologische Sortierung

(für alle Kanten e=(v,w) gilt topoNum(v) $<$ topoNum(w))

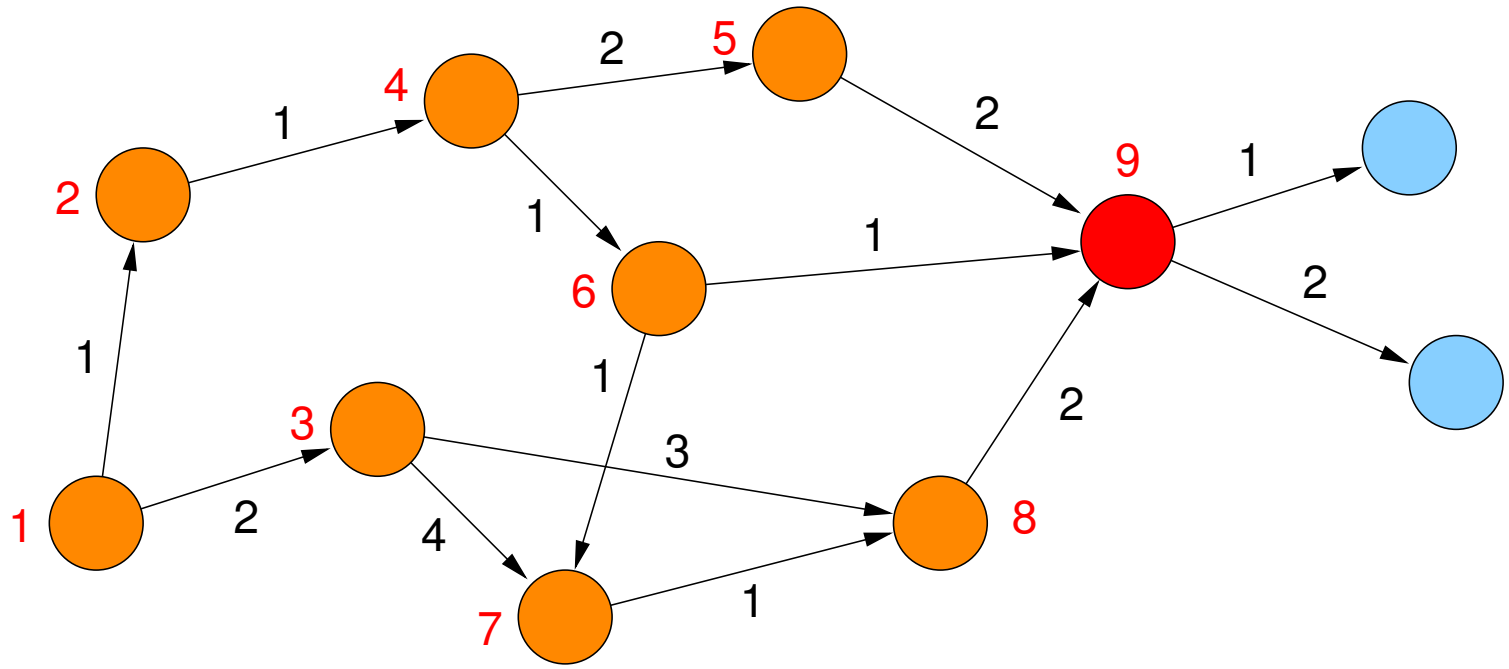


Kürzeste Wege

Beliebige Kantengewichte in DAGs

Strategie: in DAGs gibt es topologische Sortierung

(für alle Kanten e=(v,w) gilt topoNum(v) $<$ topoNum(w))

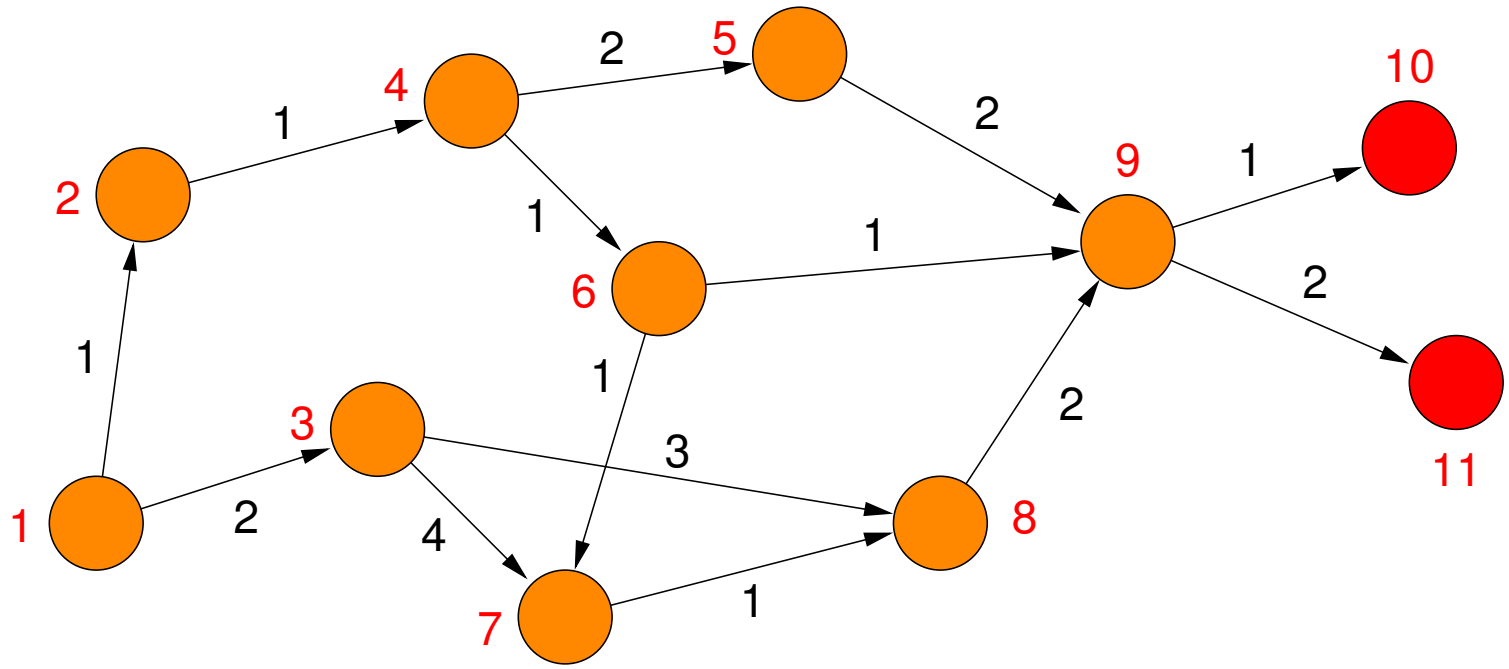


Kürzeste Wege

Beliebige Kantengewichte in DAGs

Strategie: in DAGs gibt es topologische Sortierung

(für alle Kanten e=(v,w) gilt topoNum(v) $<$ topoNum(w))

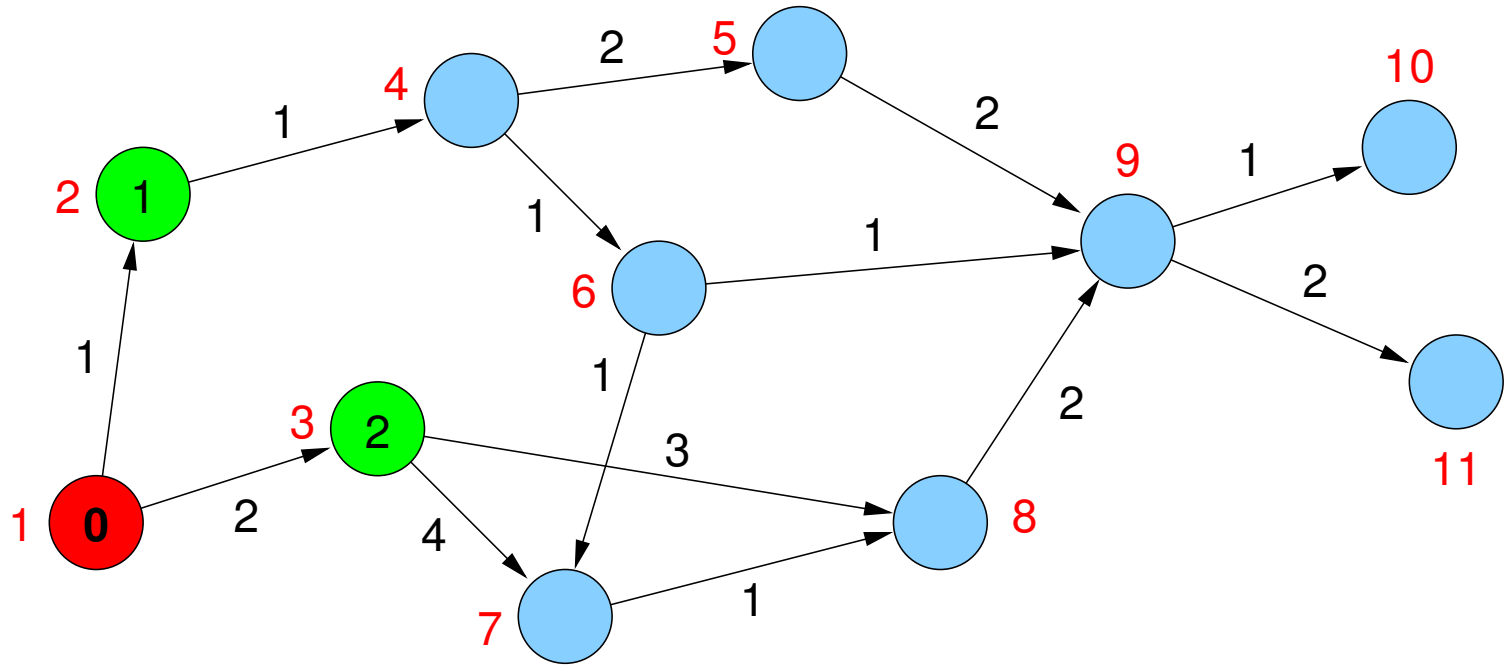


Kürzeste Wege

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

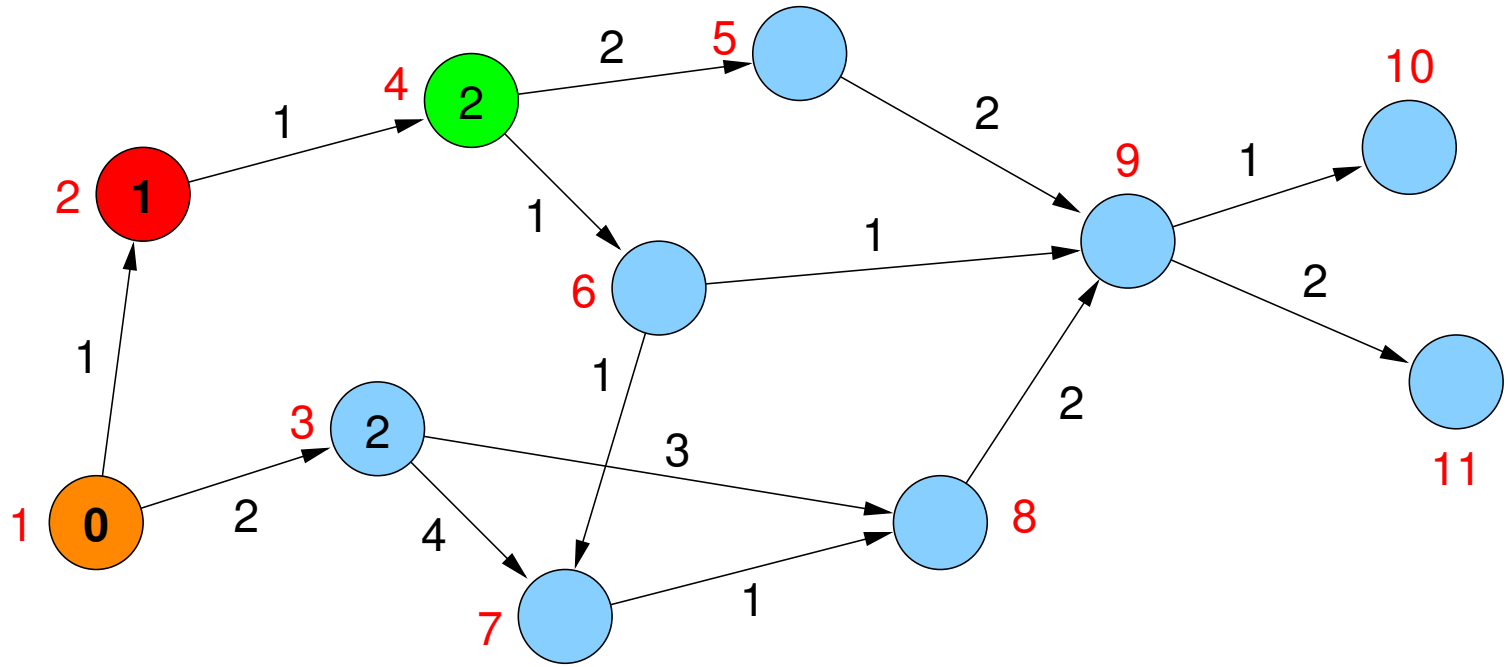


Kürzeste Wege

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

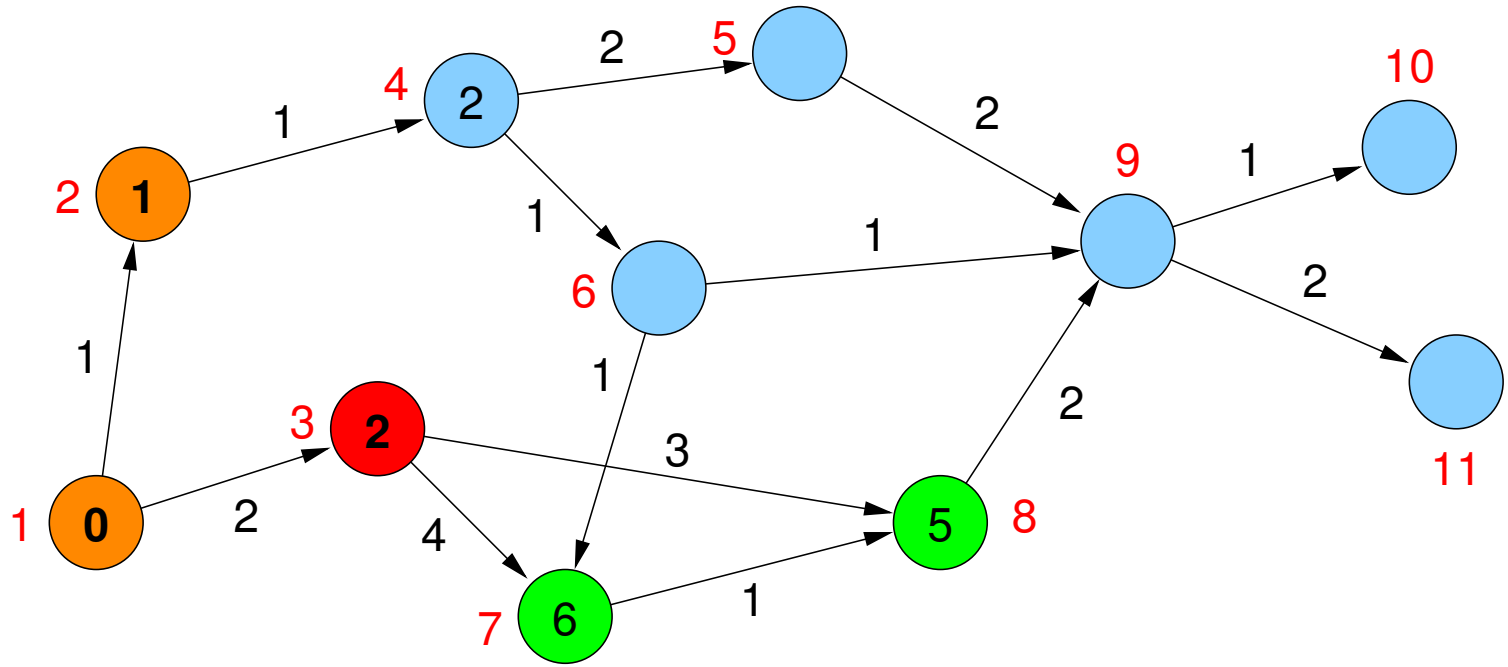


Kürzeste Wege

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

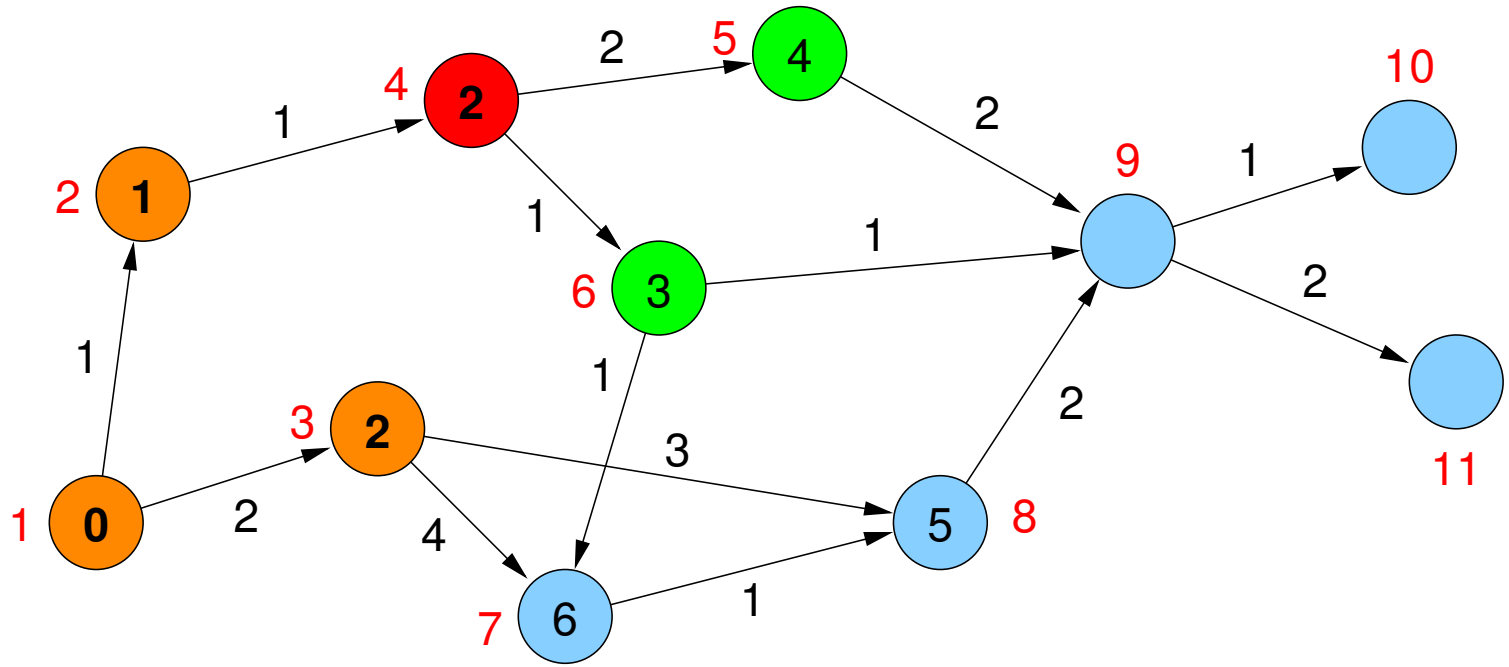


Kürzeste Wege

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

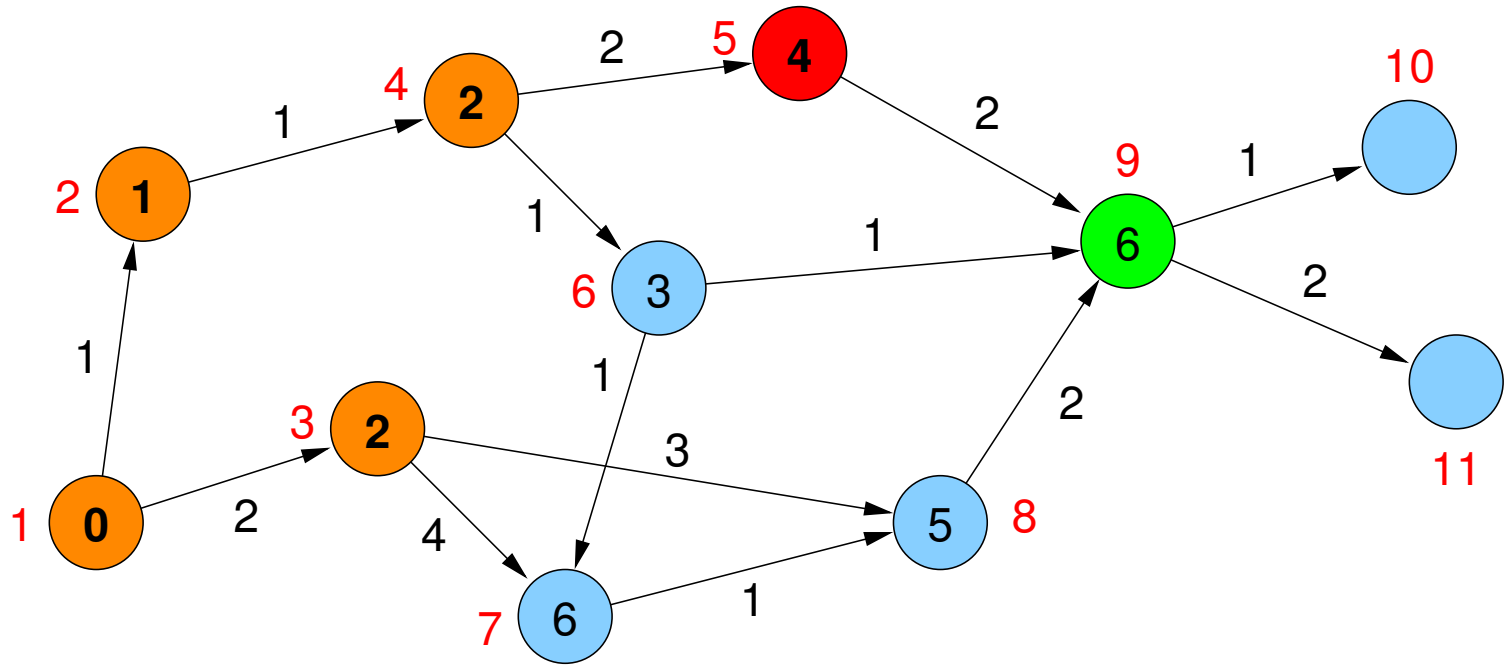


Kürzeste Wege

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

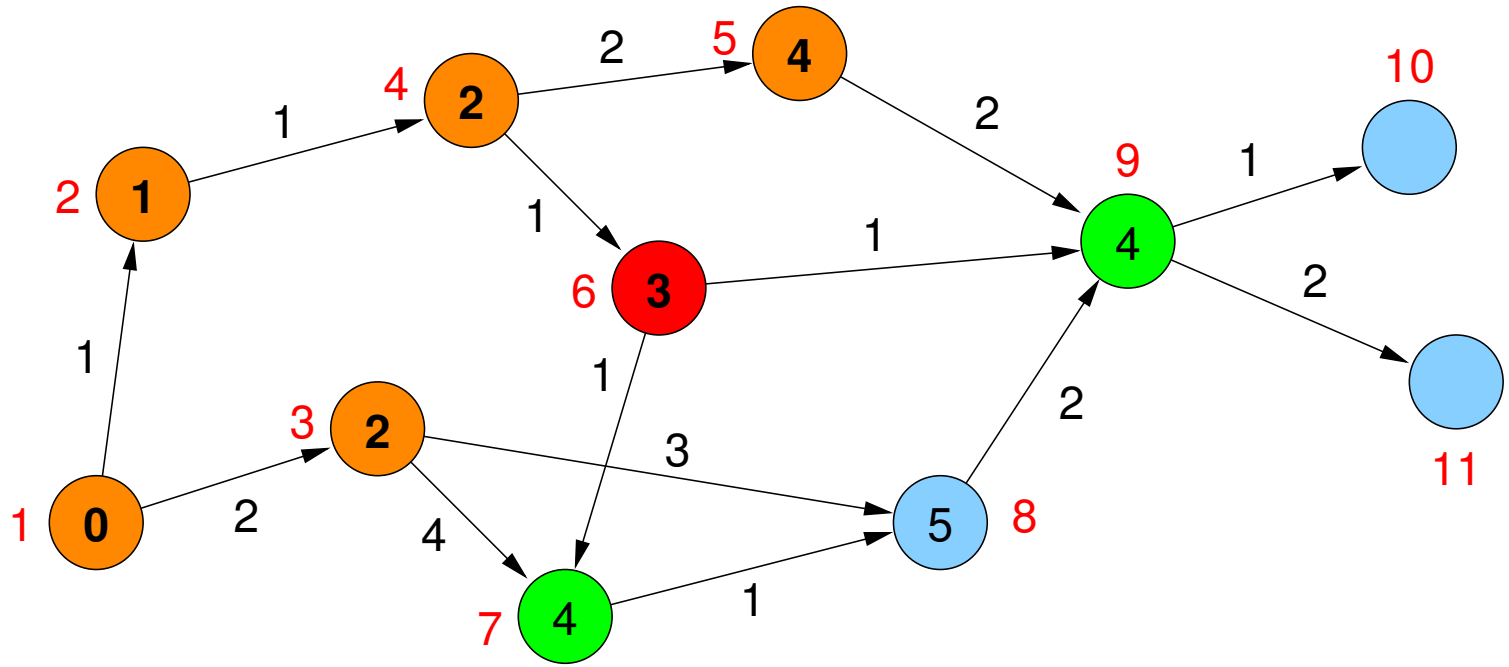


Kürzeste Wege

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

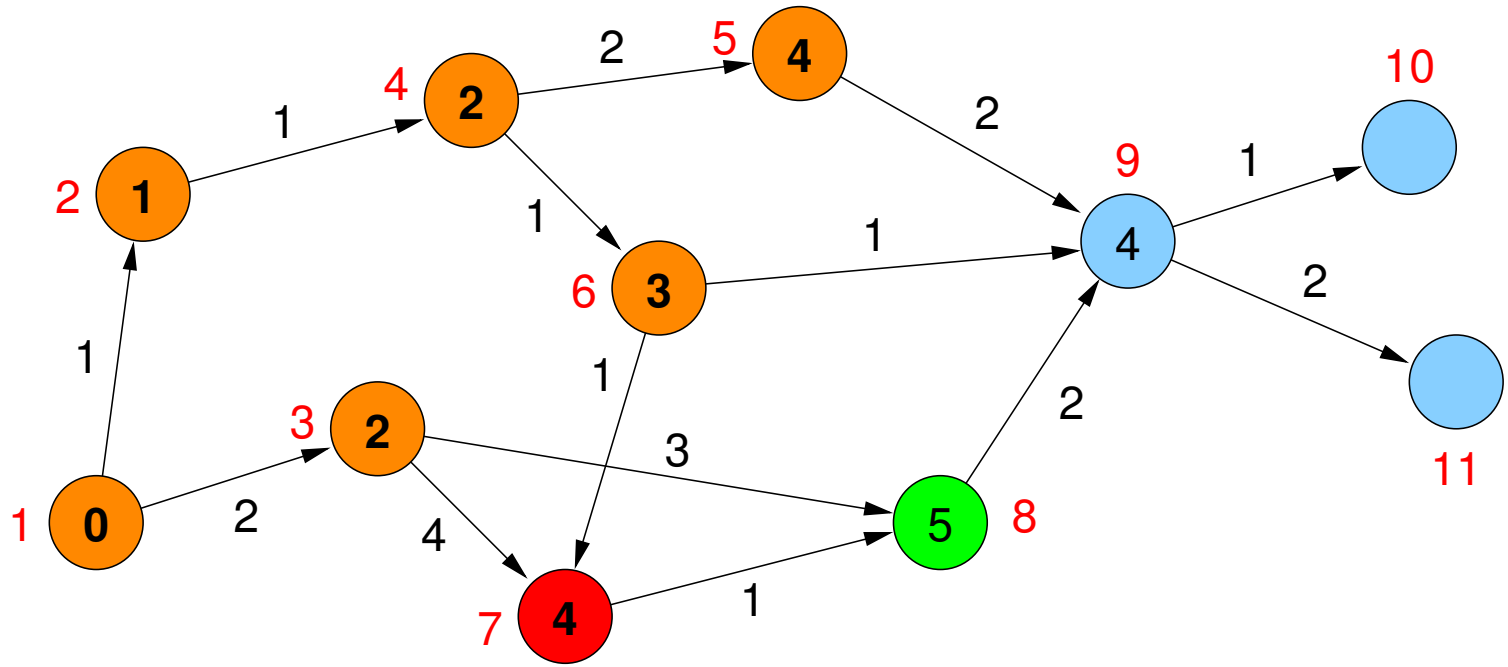


Kürzeste Wege

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

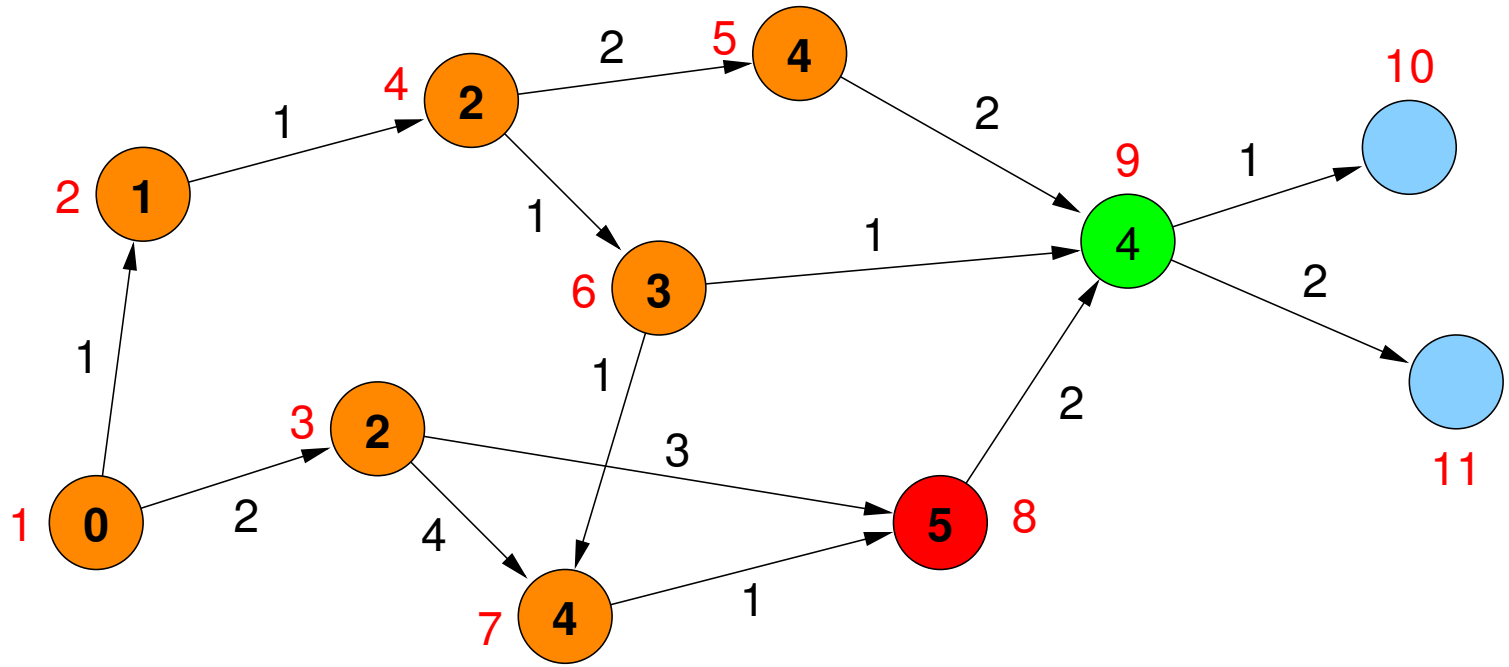


Kürzeste Wege

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

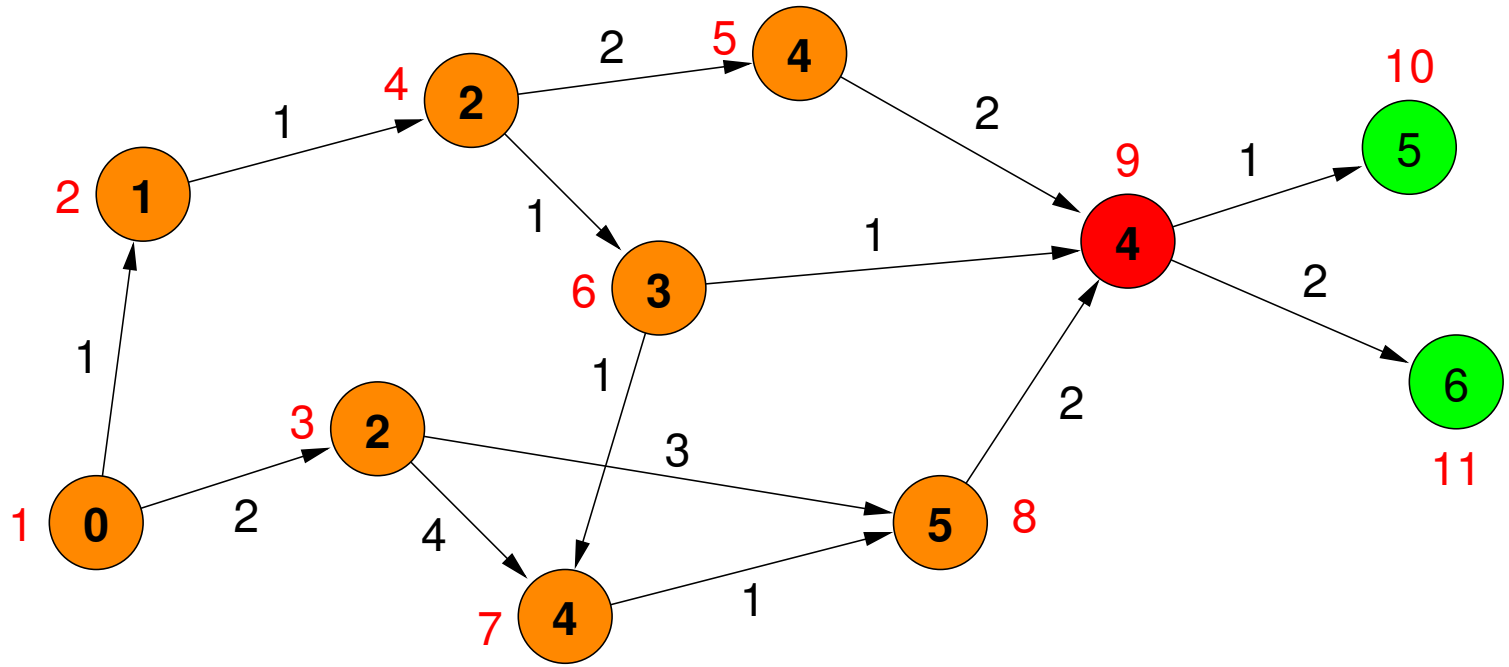


Kürzeste Wege

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

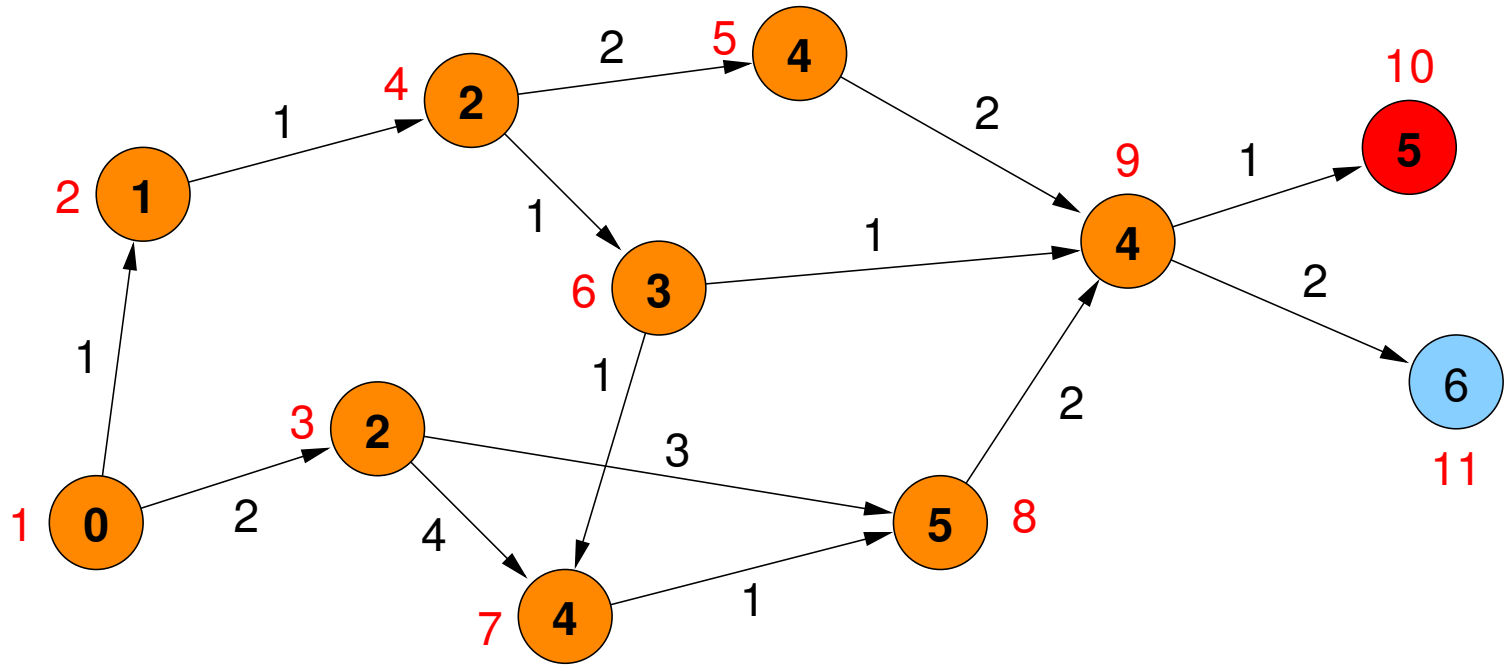


Kürzeste Wege

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte

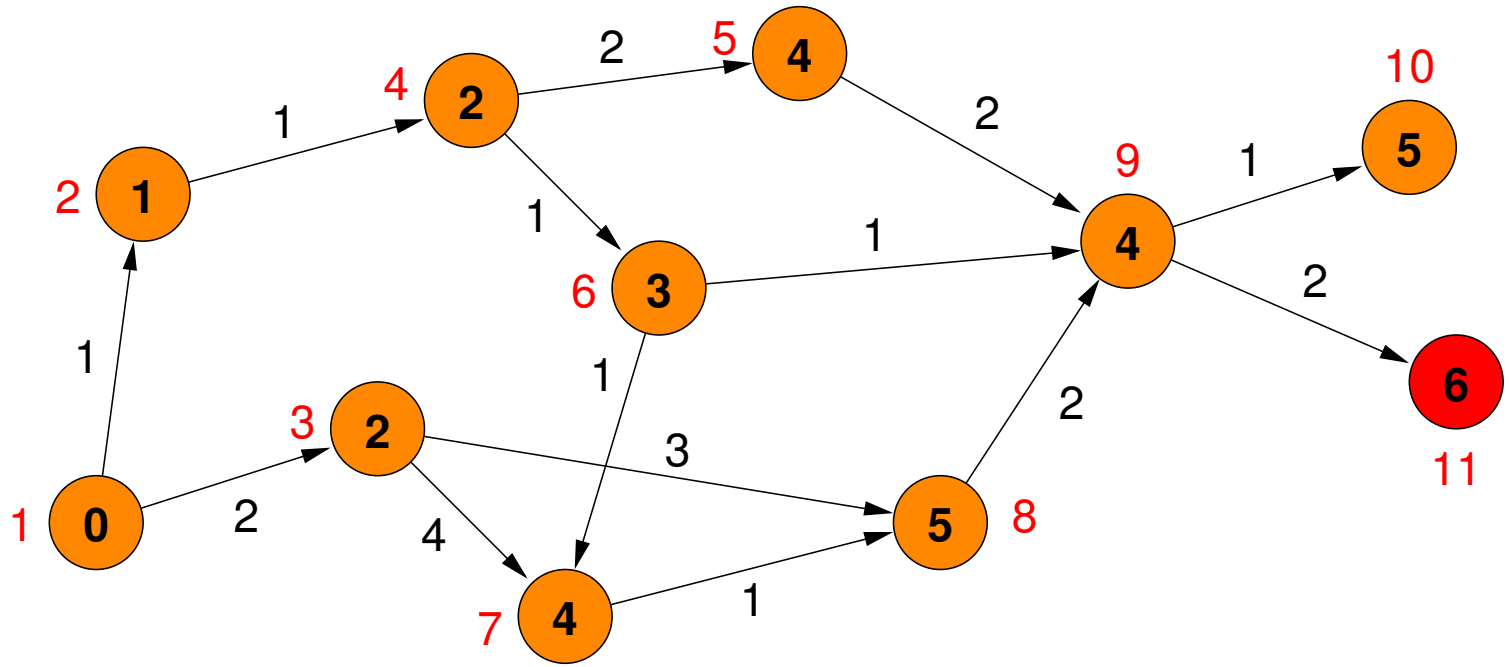


Kürzeste Wege

Beliebige Kantengewichte in DAGs

Strategie:

- betrachte Knoten in Reihenfolge der topologischen Sortierung
- aktualisiere Distanzwerte



Kürzeste Wege

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

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

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

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 $\mathcal{O}(n+m)$
- danach wird jede Kante genau einmal betrachtet

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

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

Kürzeste Wege

DAG-Strategie

1. Topologische Sortierung der Knoten

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

2. Aktualisierung der Distanzengemäß der topologischen Sortierung

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

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

Kürzeste Wege für 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 1 : Dijkstra-Algorithmus

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

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

$P = \emptyset$; $T = V$;
$d(s, v) = \infty$ for all $v \in V \setminus s$;
$d(s, s) = 0$; $pred(s) = 0$;
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**
      $d(s, w) = d(s, v) + c(v, w)$;
      $pred(w) = v$;

Algorithmus 2 : Dijkstra-Algorithmus für SSSP

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

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

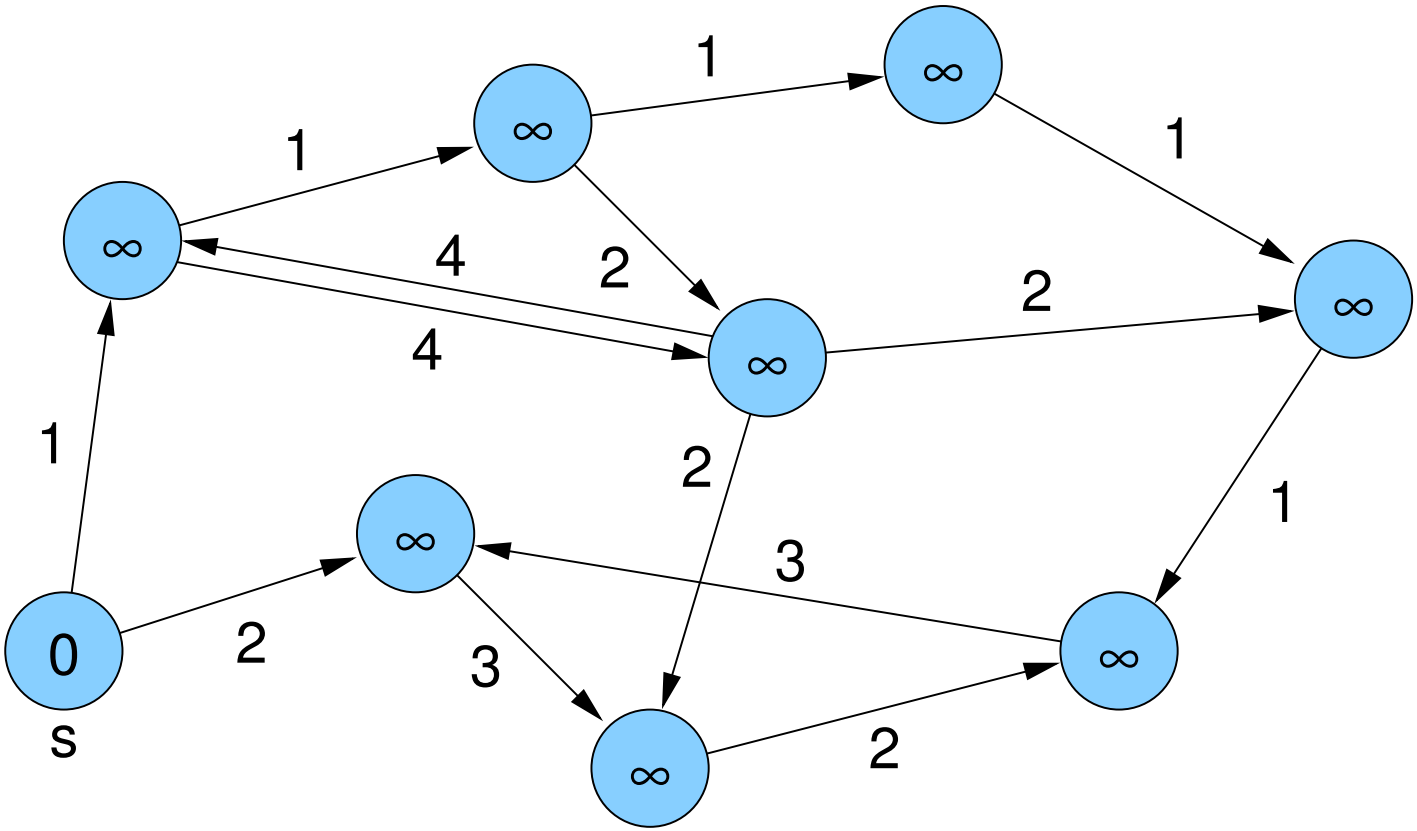
$d[v] = \infty$ for all $v \in V \setminus s$;
$d[s] = 0$; $pred[s] = \perp$;
$pq = \langle\rangle$; pq.insert$(s, 0)$;
while $\neg pq.empty()$ **do**
  $v = pq$.deleteMin();
  forall the $(v, w) \in E$ **do**
    newDist $= d[v] + c(v, w)$;
    if $newDist < d[w]$ **then**
      $pred[w] = v$;
      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 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

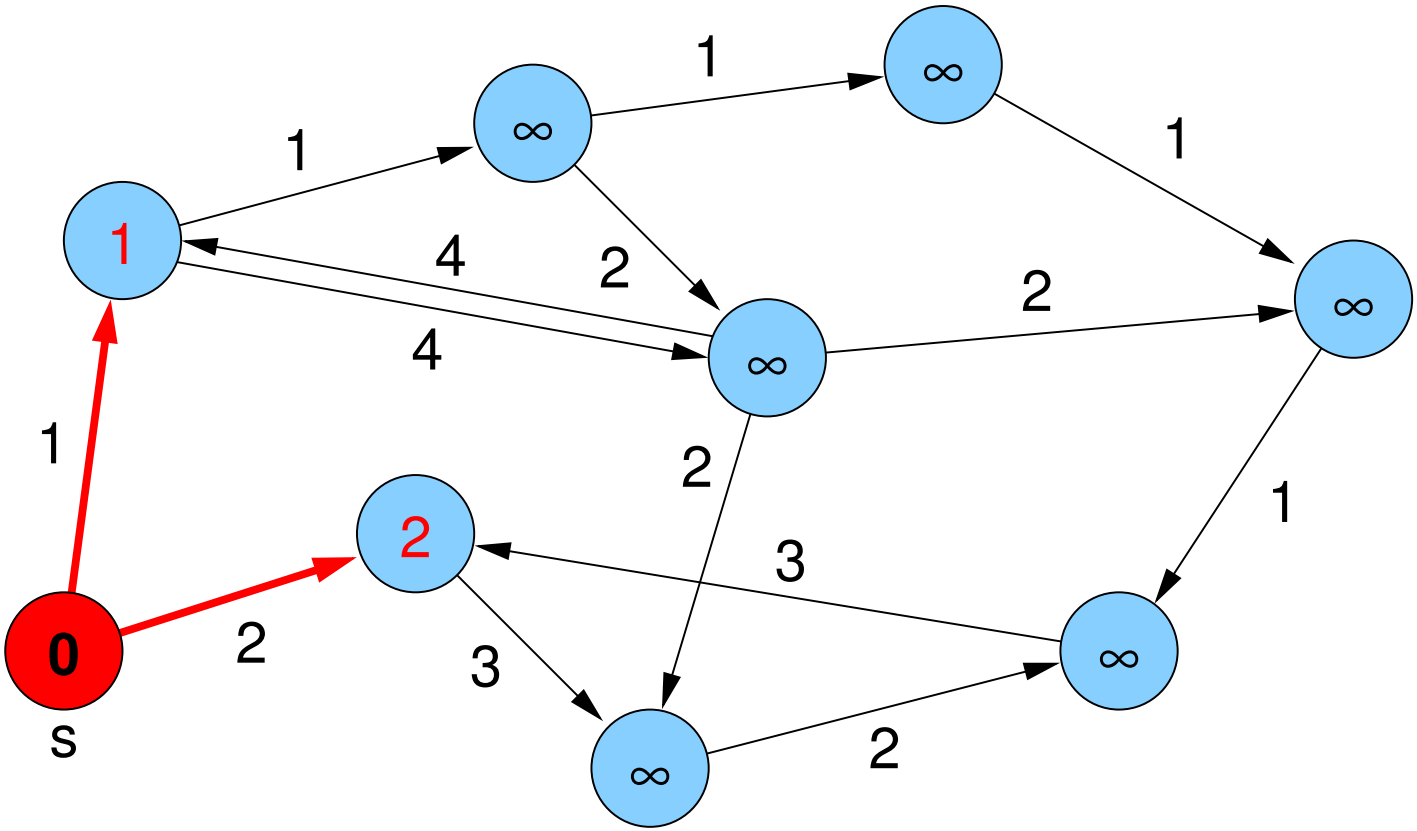
Dijkstra-Algorithmus

Beispiel:



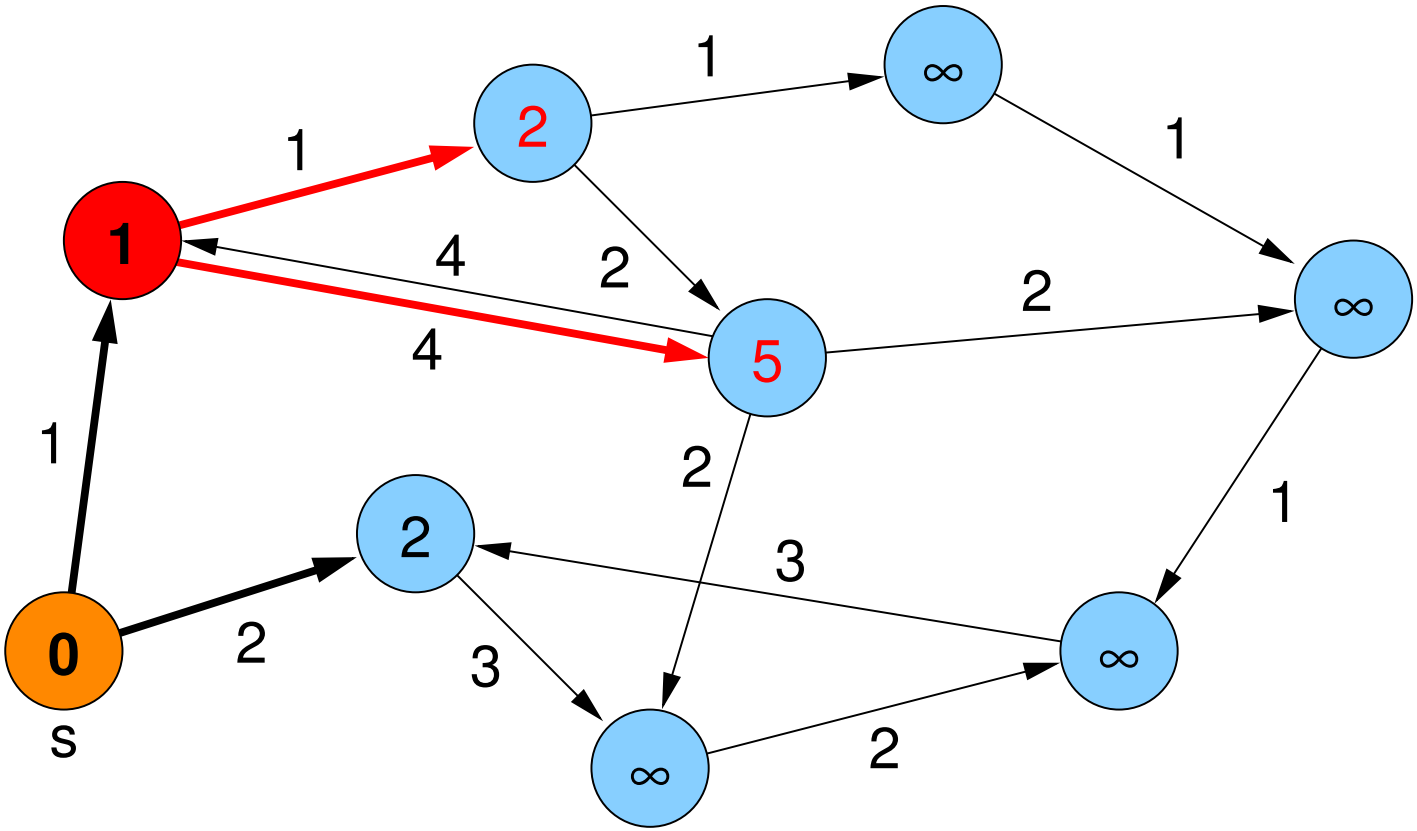
Dijkstra-Algorithmus

Beispiel:



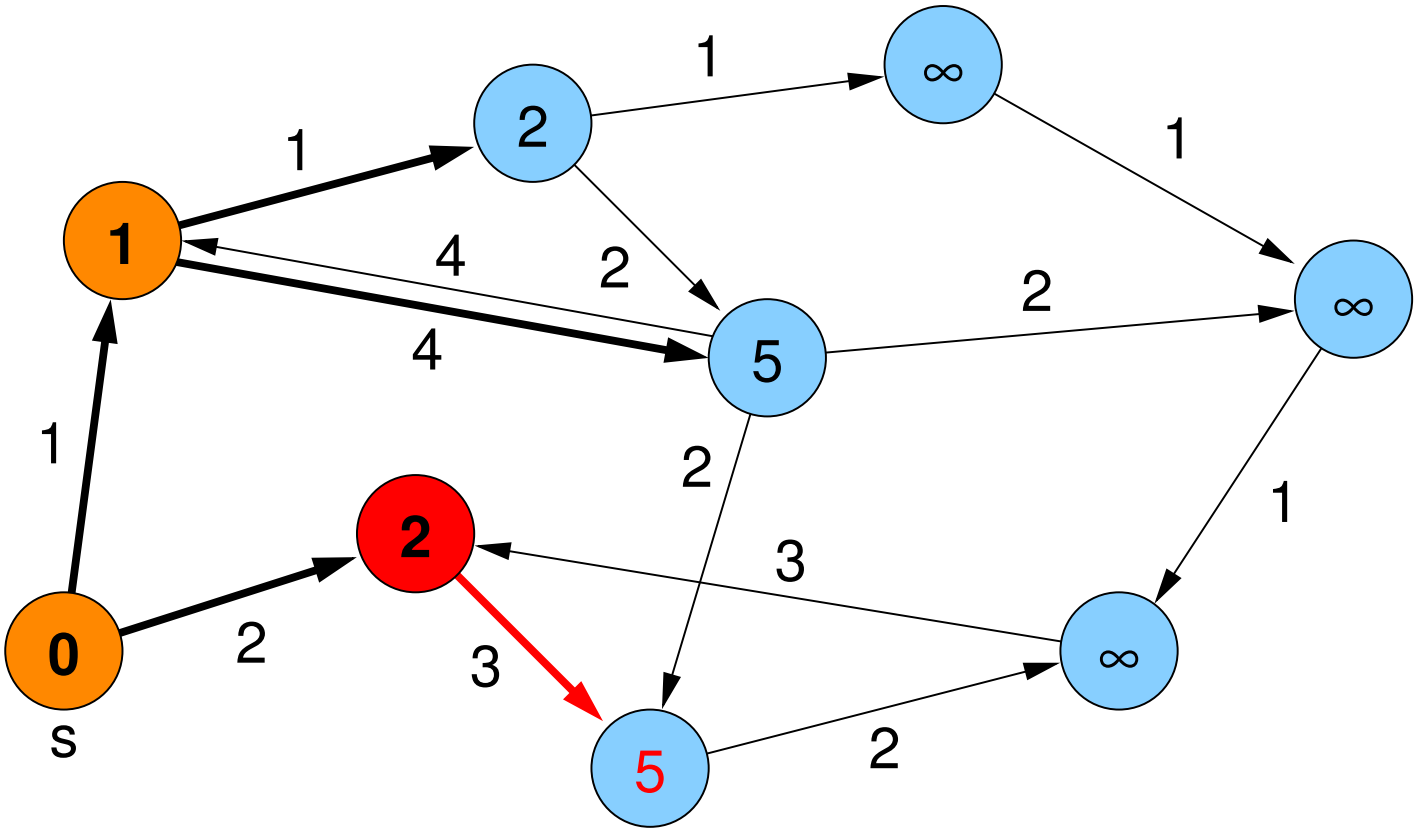
Dijkstra-Algorithmus

Beispiel:



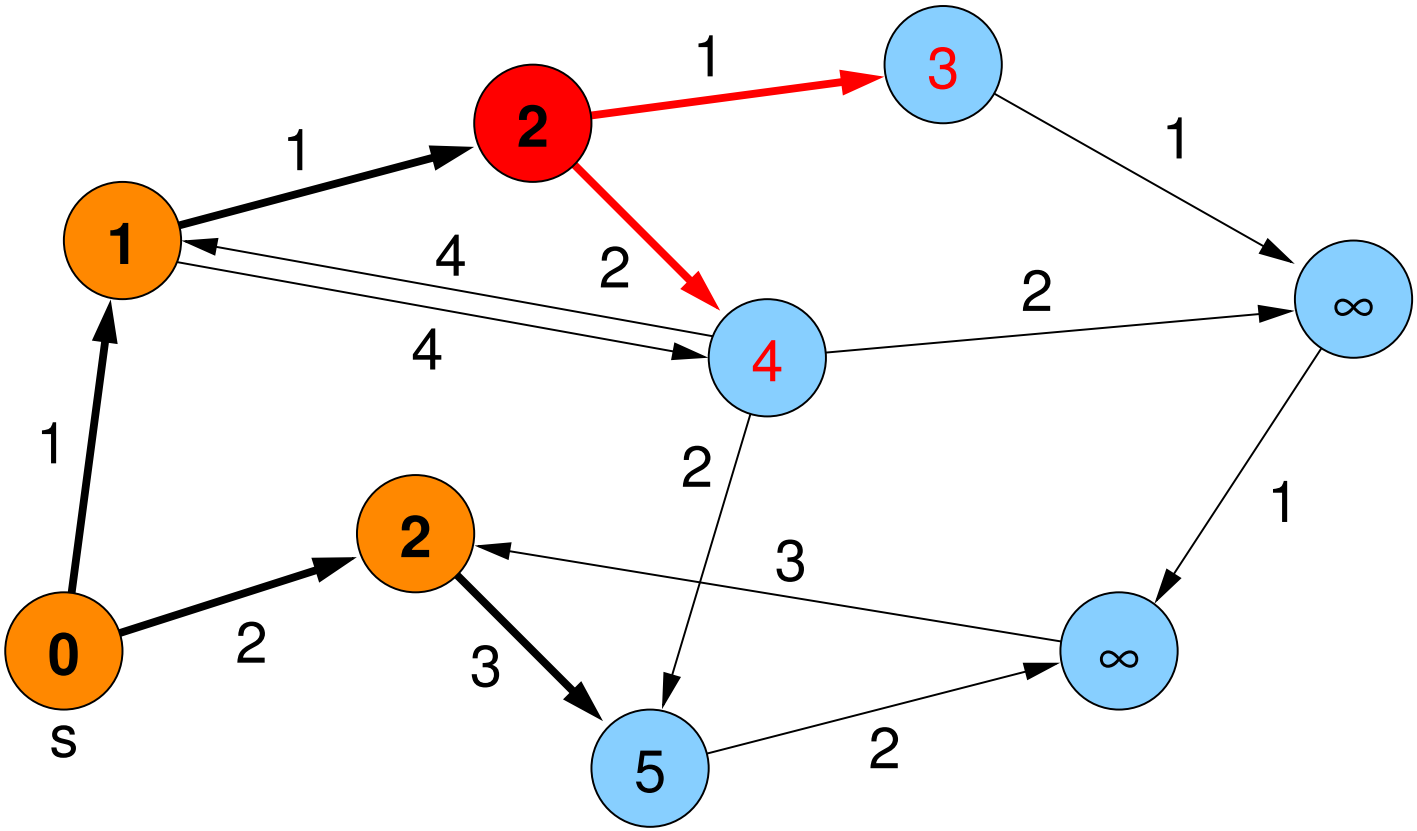
Dijkstra-Algorithmus

Beispiel:



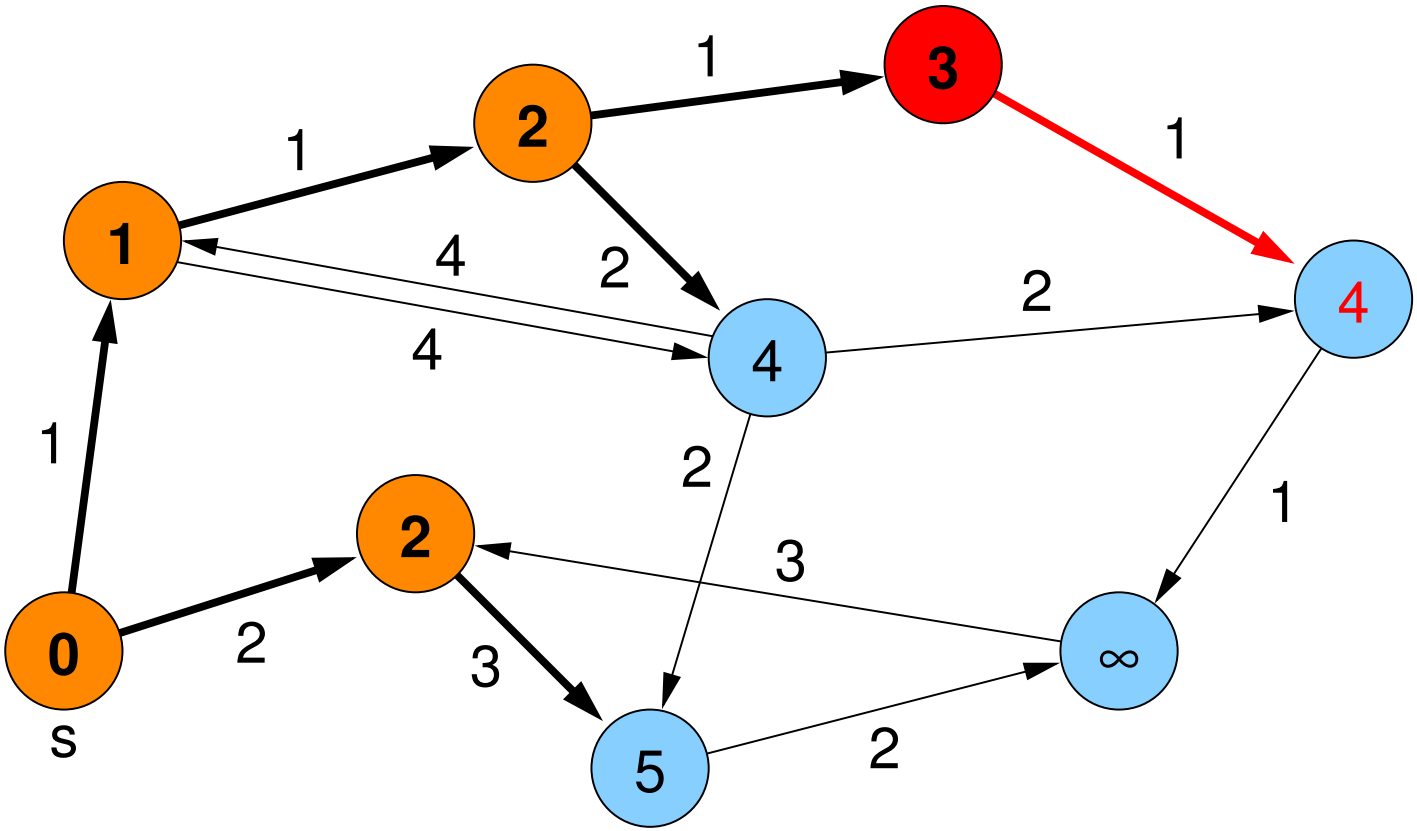
Dijkstra-Algorithmus

Beispiel:



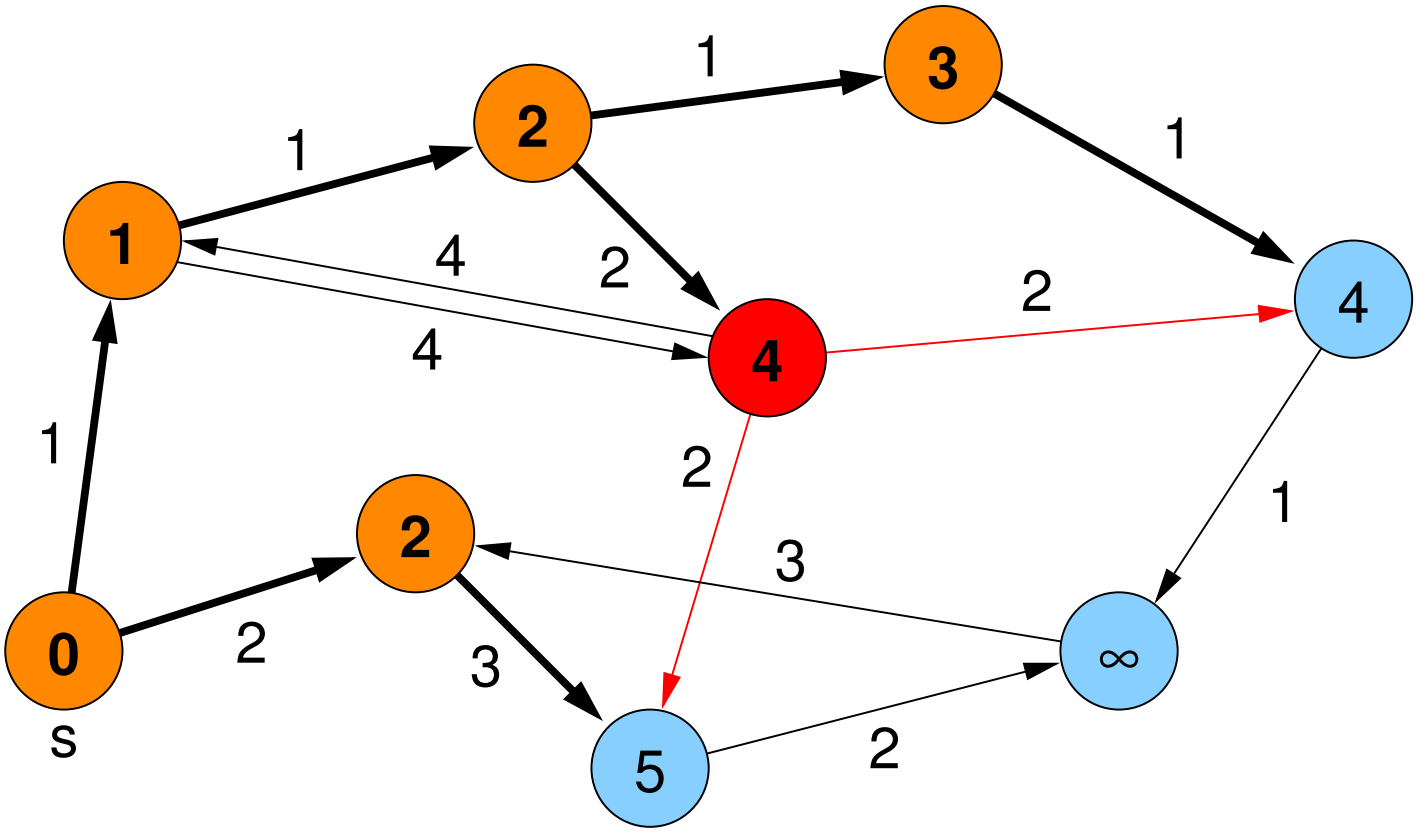
Dijkstra-Algorithmus

Beispiel:



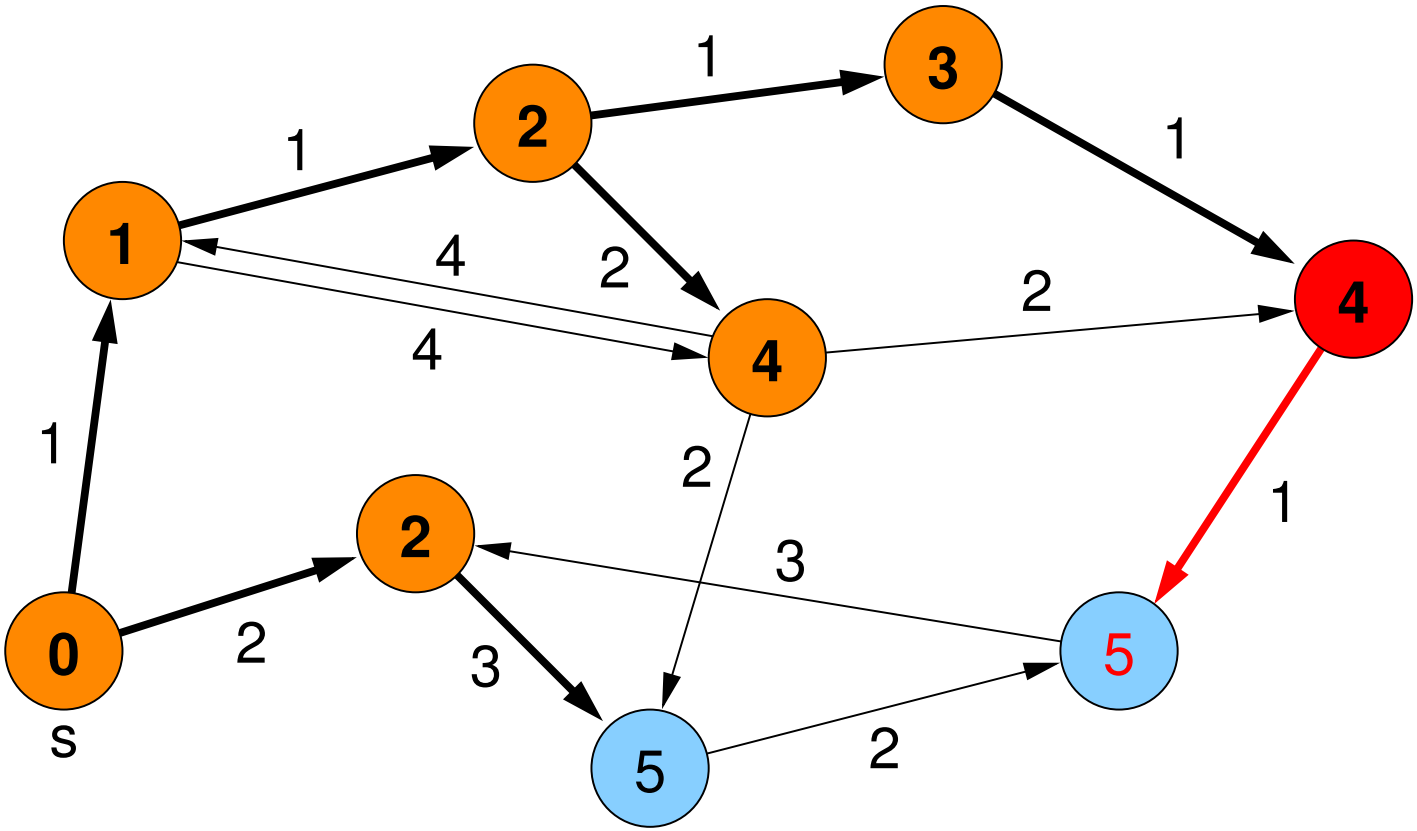
Dijkstra-Algorithmus

Beispiel:



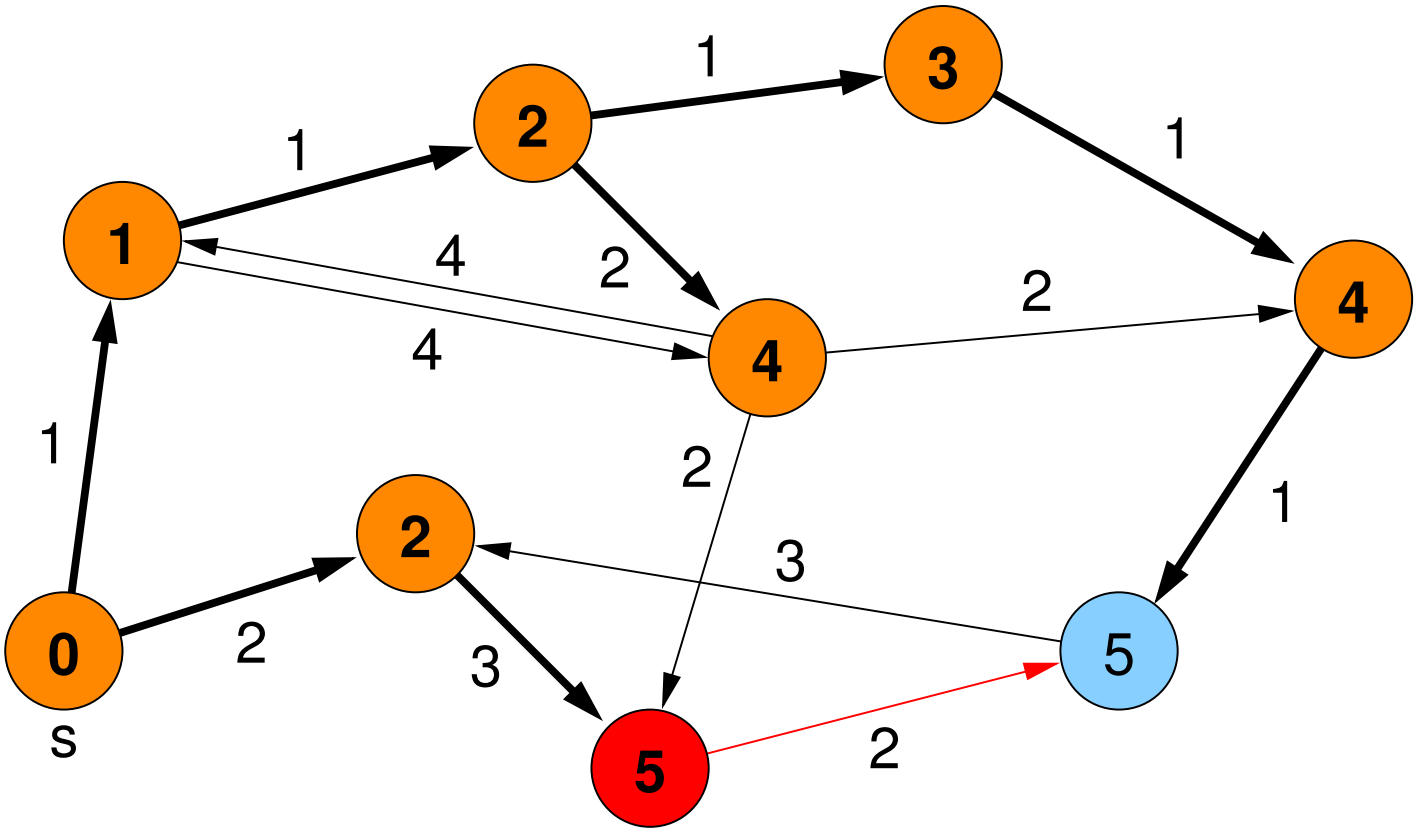
Dijkstra-Algorithmus

Beispiel:



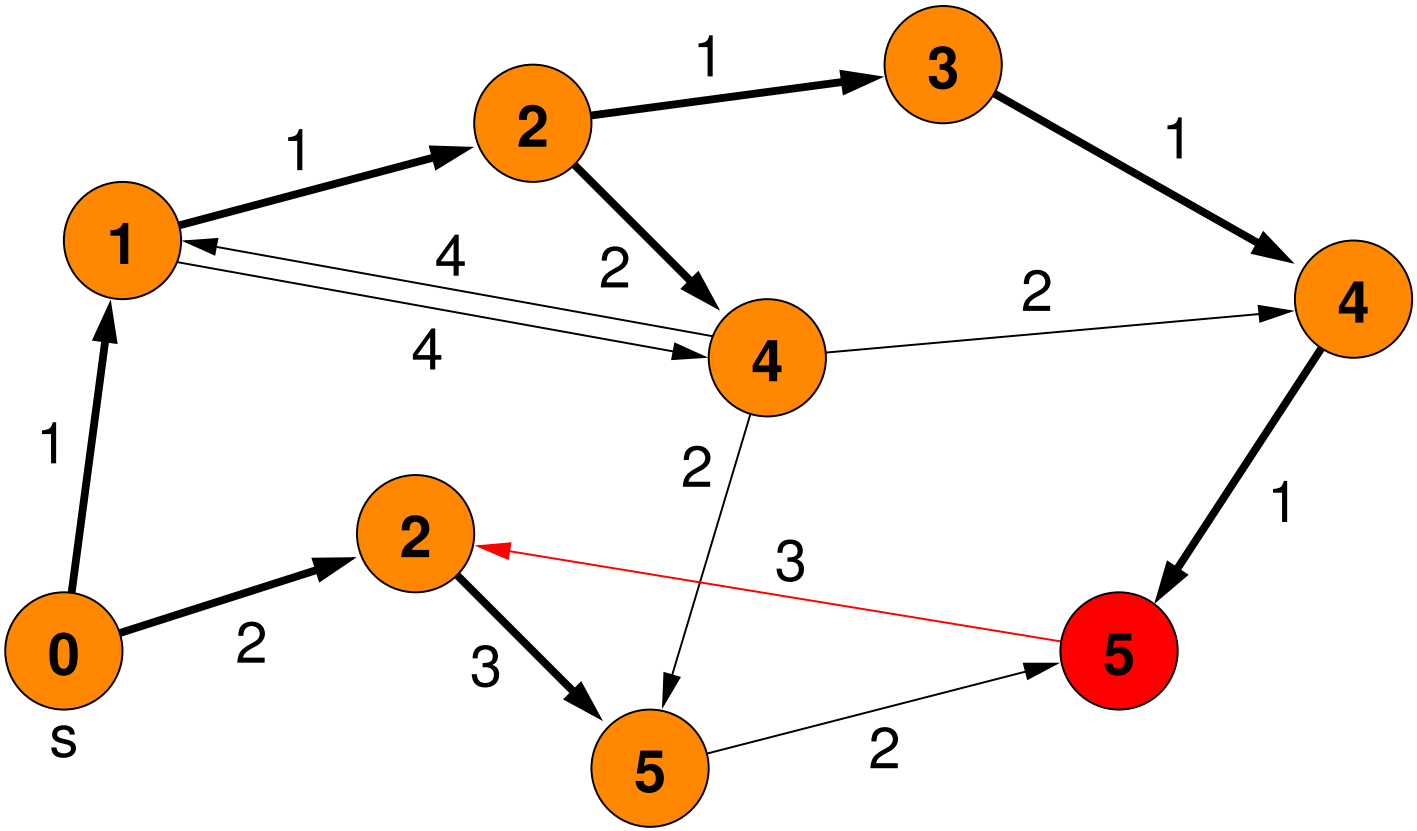
Dijkstra-Algorithmus

Beispiel:



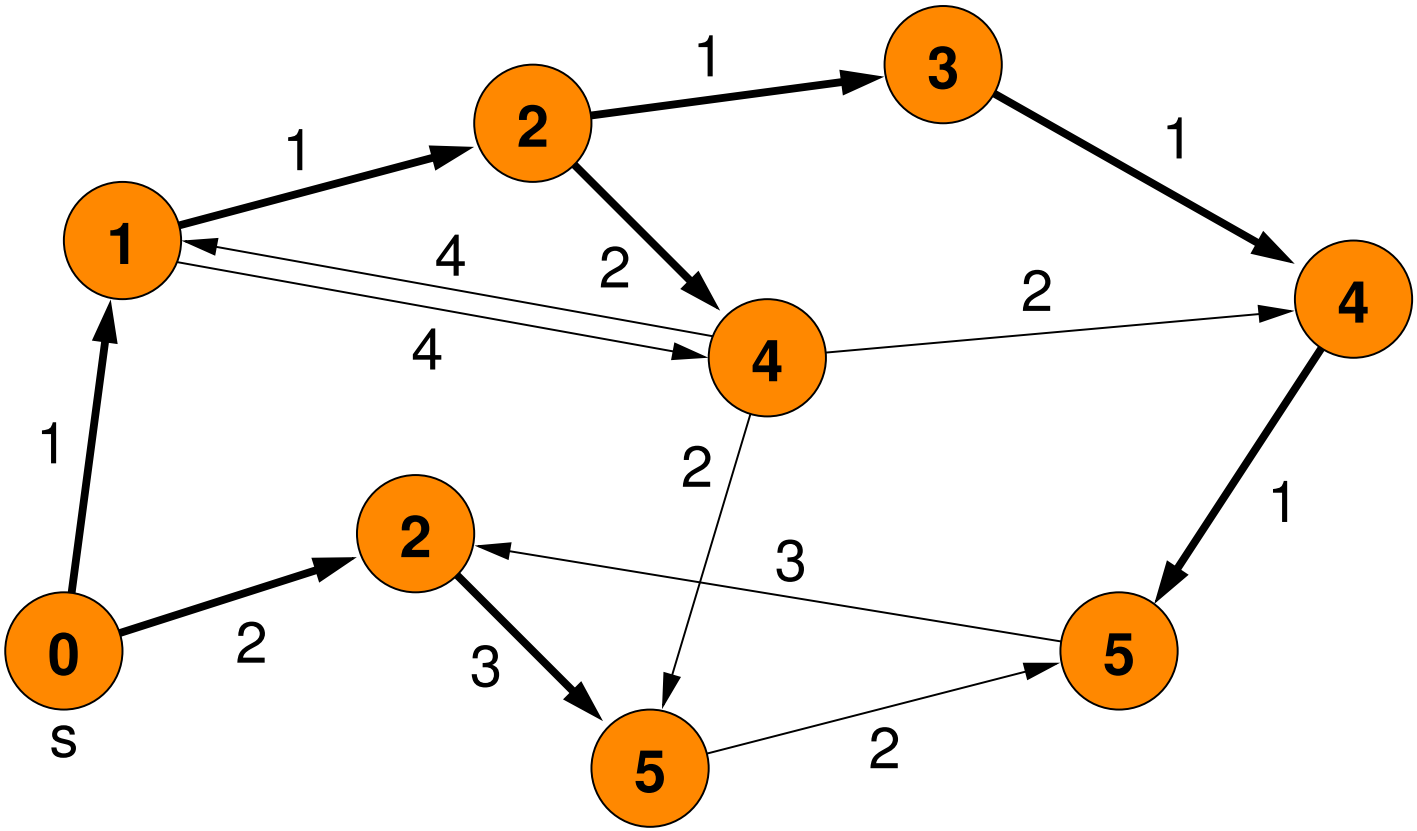
Dijkstra-Algorithmus

Beispiel:



Dijkstra-Algorithmus

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 zu klein 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 $\mathcal{O}(1)$ für `insert` und `decreaseKey`, $\mathcal{O}(\log n)$ `deleteMin`)

- Komplexität:
  - $\mathcal{O}(n)$ `insert`
  - $\mathcal{O}(n)$ `deleteMin`
  - $\mathcal{O}(m)$ `decreaseKey`

  $\Rightarrow$ $\mathcal{O}(m + n \log n)$

- 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

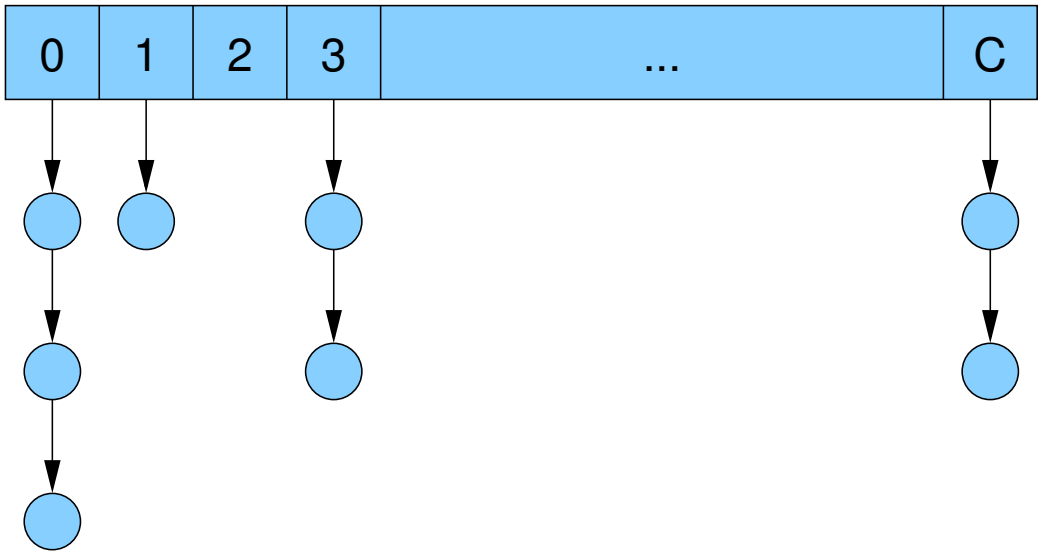
Annahme: alle Kantengewichte im Bereich $[0, C]$

Konsequenz für Dijkstra-Algorithmus:

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

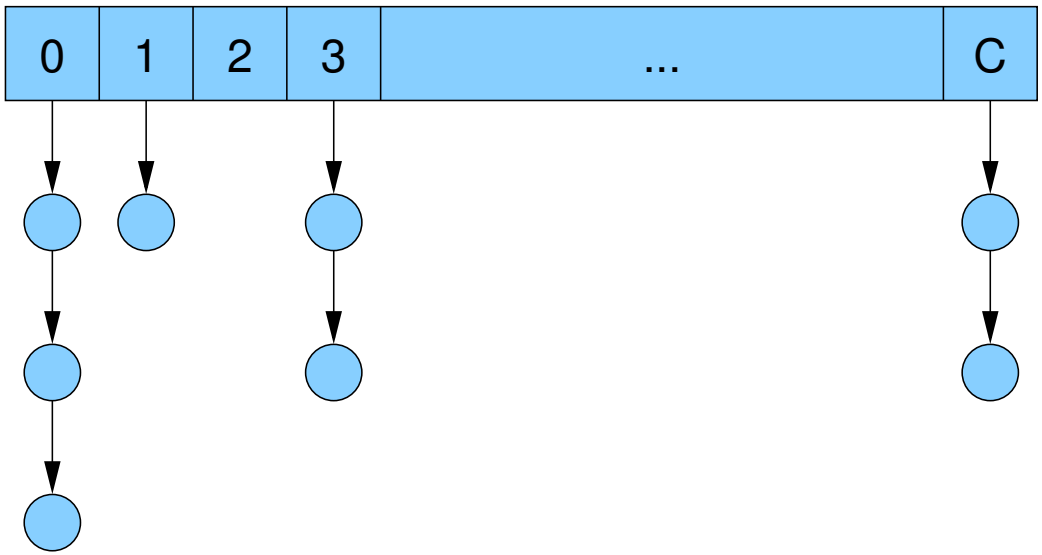
Bucket Queue

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



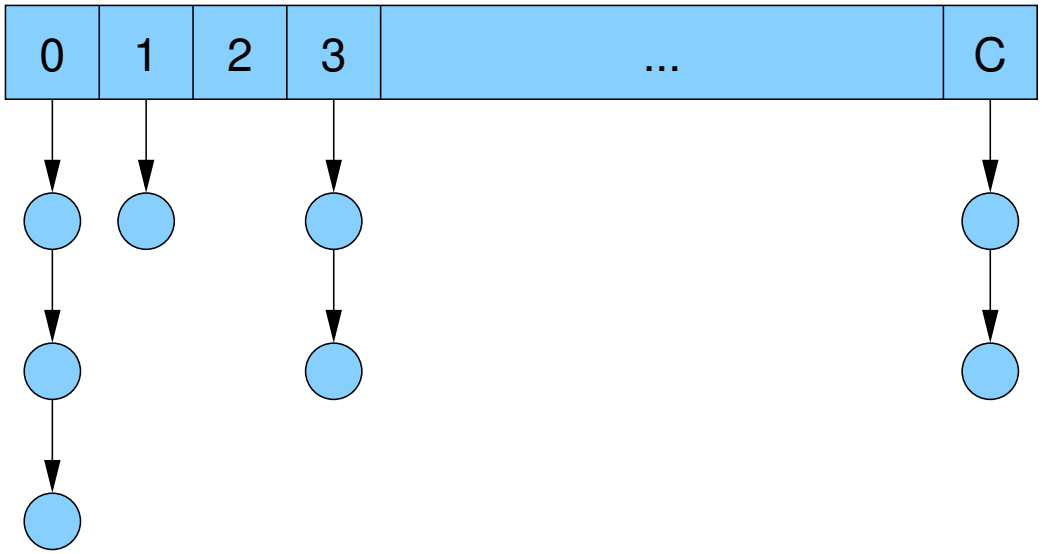
Bucket Queue

- 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 ($\mathcal{O}(1)$)
- decreaseKey(v): entfernt v aus momentaner Liste ($\mathcal{O}(1)$ falls Handle auf Listenelement in v gespeichert) und fügt v in Liste $B[d[v] \bmod (C+1)]$ ein ($\mathcal{O}(1)$)
- 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 ($\mathcal{O}(C)$)



Dijkstra mit Bucket Queue

- insert, decreaseKey: $\mathcal{O}(1)$
- deleteMin: $\mathcal{O}(C)$
- Dijkstra: $\mathcal{O}(m + C \cdot n)$

- lässt sich mit Radix Heaps noch verbessern
- verwendet exponentiell wachsende Bucket-Größen
- Details in der Vorlesung Effiziente Algorithmen und Datenstrukturen
- Laufzeit ist dann $\mathcal{O}(m + n \log C)$