

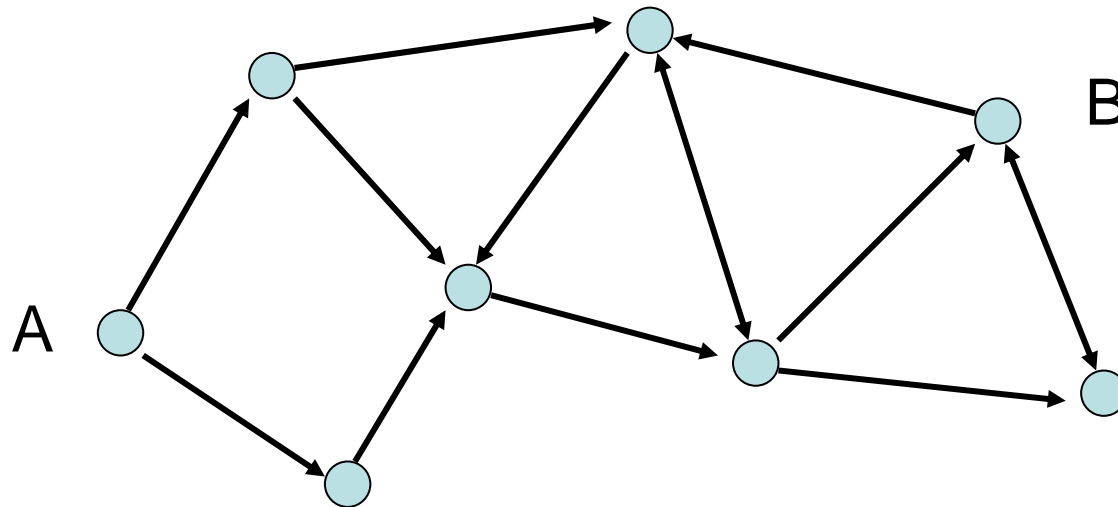
Effiziente Algorithmen und Datenstrukturen I

Kapitel 8: Kürzeste Wege

Christian Scheideler
WS 2008

Kürzeste Wege

Zentrale Frage: Wie komme ich am schnellsten von A nach B?



Kürzeste Wege

Zentrale Frage: Wie komme ich am schnellsten von A nach B?

Fälle:

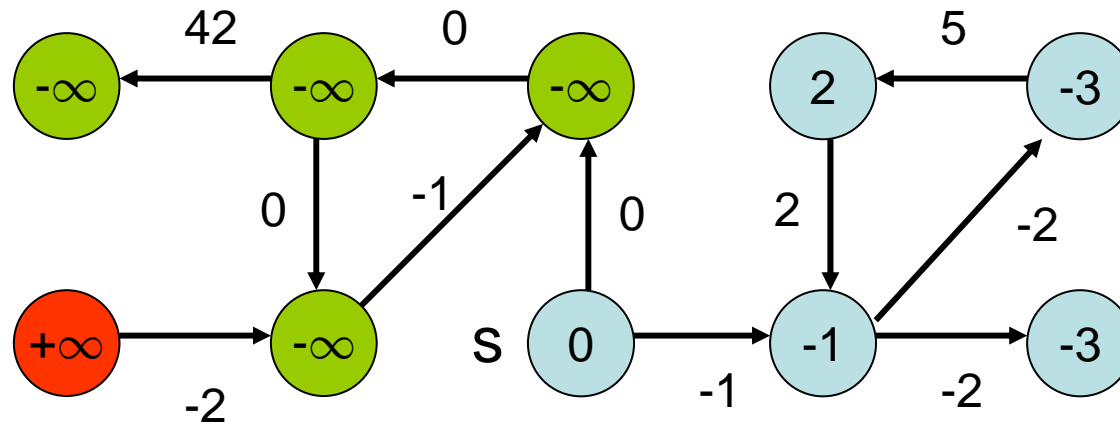
- Kantenkosten 1
- DAG, beliebige Kantenkosten
- Beliebiger Graph, positive Kantenkosten
- Beliebiger Graph, beliebige Kosten

Einführung

Kürzeste-Wege-Problem:

- gerichteter Graph $G=(V,E)$
- Kantenkosten $c:E\rightarrow\mathbb{R}$
- **SSSP** (single source shortest path):
Kürzeste Wege von einer Quelle zu allen anderen Knoten
- **APSP** (all pairs shortest path):
Kürzeste Wege zwischen allen Paaren

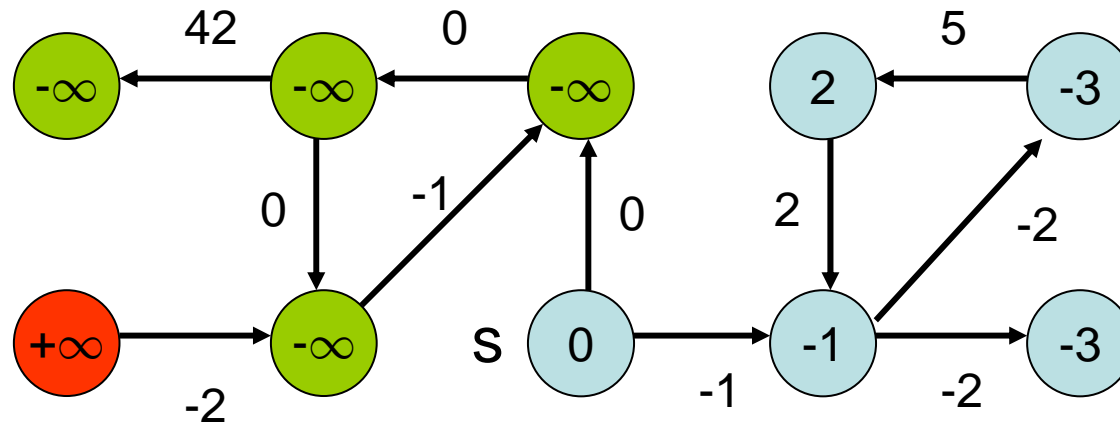
Einführung



$\mu(s,v)$: Distanz zwischen s und v

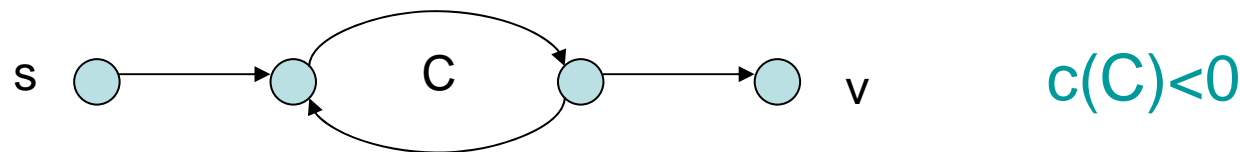
$$\mu(s,v) = \left\{ \begin{array}{l} \infty \quad \text{kein Weg von } s \text{ nach } v \\ -\infty \quad \text{Weg bel. kleiner Kosten von } s \text{ nach } v \\ \min\{ c(p) \mid p \text{ ist Weg von } s \text{ nach } v \} \end{array} \right\}$$

Einführung



Wann sind die Kosten $-\infty$?

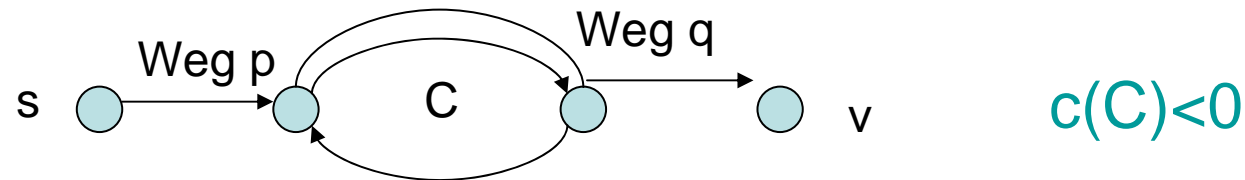
Wenn es einen negativen Kreis gibt:



Einführung

Negativer Kreis hinreichend und notwendig für Wegekosten $-\infty$.

Negativer Kreis hinreichend:



Kosten für i -fachen Durchlauf von C :

$$c(p) + i \cdot c(C) + c(q)$$

Für $i \rightarrow \infty$ geht Ausdruck gegen $-\infty$.

Einführung

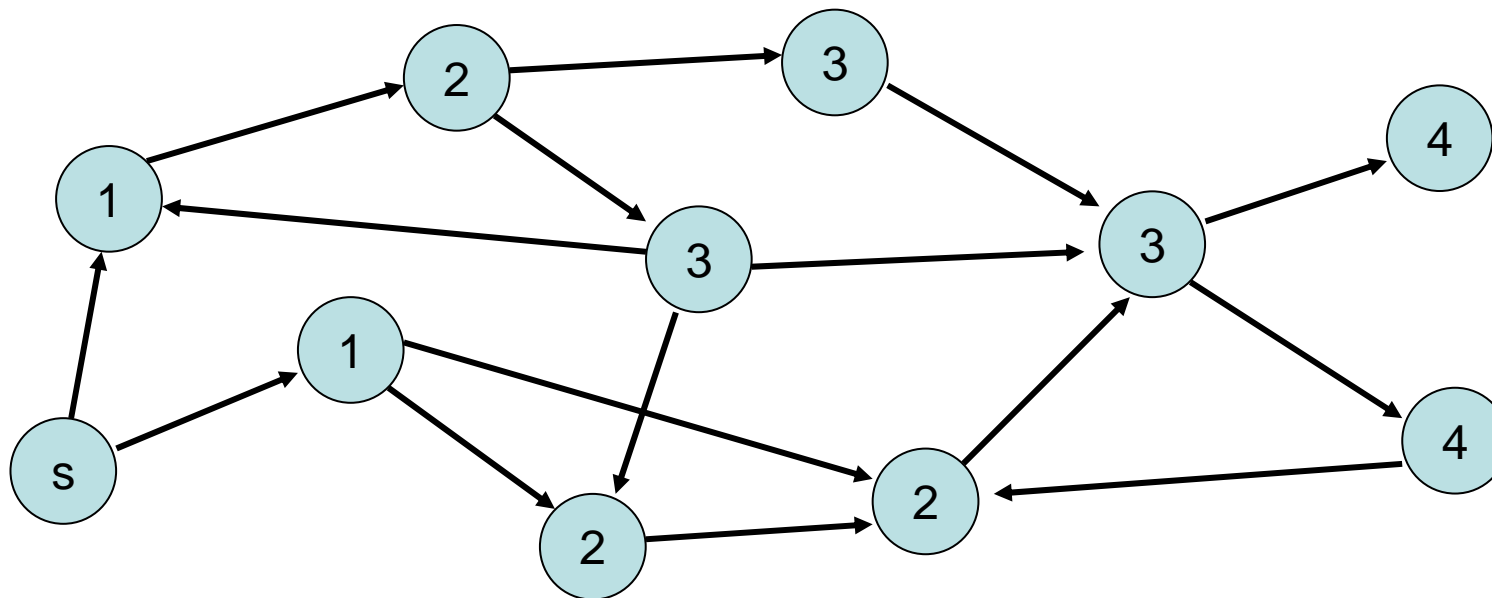
Negativer Kreis hinreichend und notwendig für Wegekosten $-\infty$.

Negativer Kreis notwendig:

- l : minimale Kosten eines **einfachen** Weges von s nach v
- es gibt **nichteinfachen** Weg r von s nach v mit Kosten $c(r) < l$
- r nicht einfach: Zerlegung in pCq , wobei C ein Kreis ist und pq ein einfacher Pfad
- da $c(r) < l \leq c(pq)$ ist, gilt $c(C) < 0$

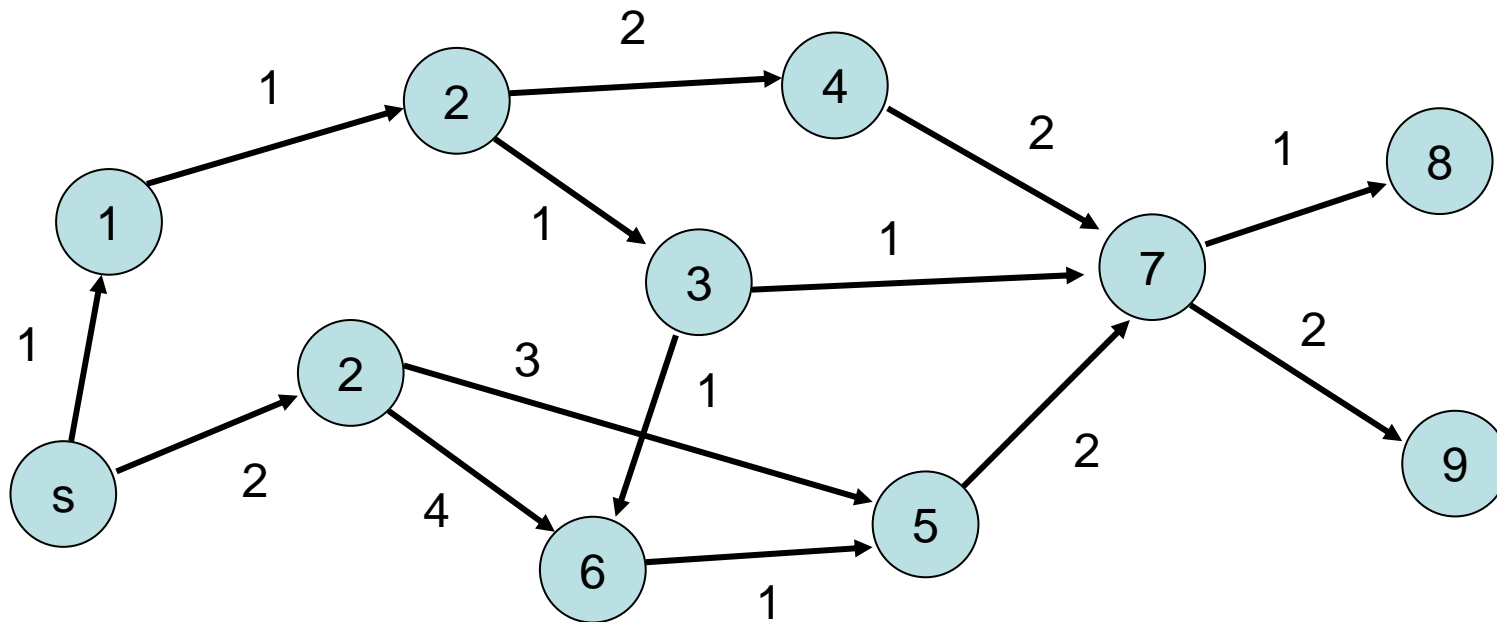
Kürzeste Wege

Graph mit Kantenkosten 1:
Führe Breitensuche durch.



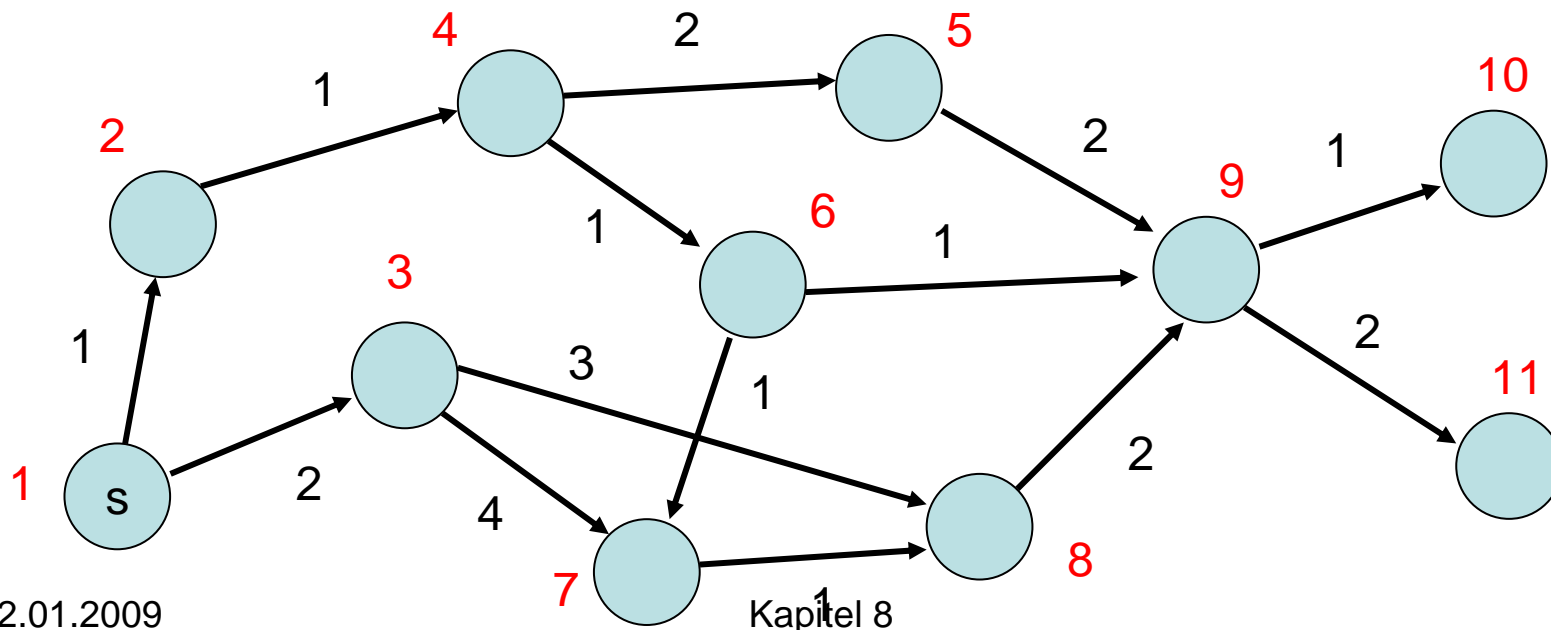
Kürzeste Wege in DAGs

Reine Breitensuche funktioniert nicht.



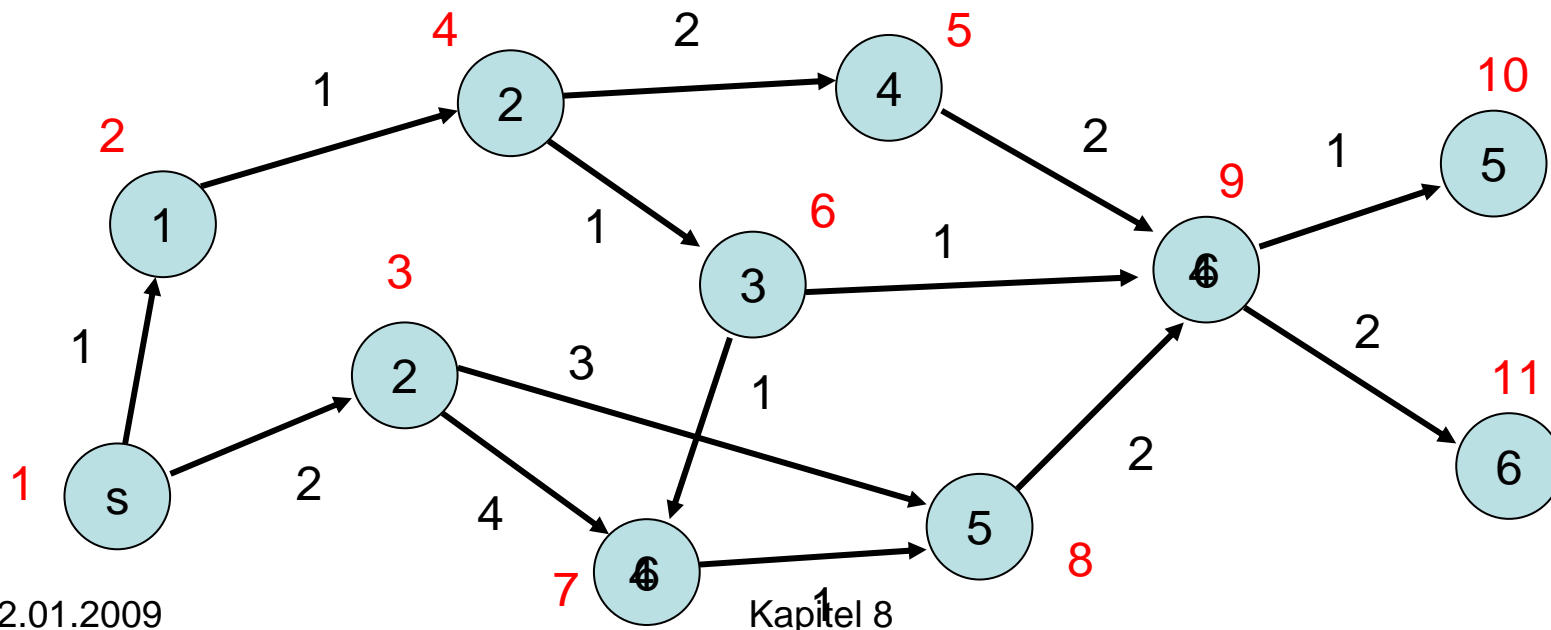
Kürzeste Wege in DAGs

Strategie: nutze aus, dass Knoten in DAGs topologisch sortiert werden können (alle Kanten $a \rightarrow b$ erfüllen $a < b$)



Kürzeste Wege in DAGs

Strategie: betrachte dann Knoten in der Reihenfolge ihrer topologischen Sortierung und aktualisiere Distanzen zu **s**



Kürzeste Wege in DAGs

Strategie:

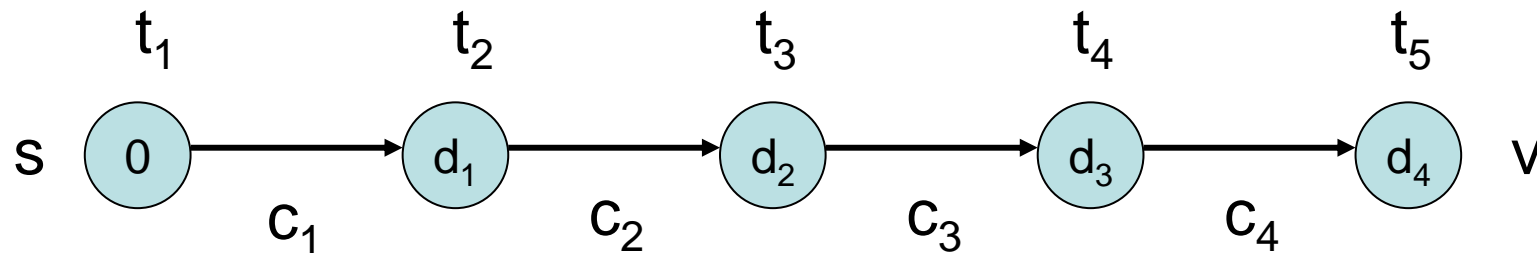
1. Topologische Sortierung der Knoten
2. Aktualisierung der Distanzen gemäß der topologischen Sortierung

Warum funktioniert das??

Kürzeste Wege in DAGs

Betrachte **kürzesten Weg** von **s** nach **v**.

Dieser hat topologische Sortierung $(t_i)_i$ mit $t_i < t_{i+1}$ for alle i .

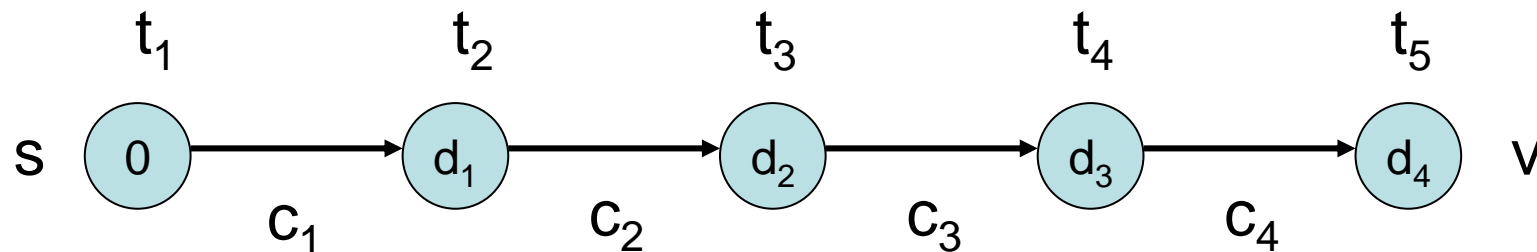


Besuch in topologischer Reihenfolge führt zu richtigen Distanzen ($d_i = \sum_{j \leq i} c_j$).

Kürzeste Wege in DAGs

Betrachte **kürzesten Weg** von s nach v .

Dieser hat topologische Sortierung $(t_i)_i$ mit $t_i < t_{i+1}$ für alle i .



Bemerkung: kein Knoten auf dem Weg zu v kann Distanz $< d_i$ zu s haben, da sonst kürzerer Weg zu v möglich wäre.

Kürzeste Wege in Graphen

Allgemeine Strategie:

- Am Anfang, setze $d(s) := 0$ und $d(v) := \infty$ für alle Knoten
- 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

Zurück zur Strategie:

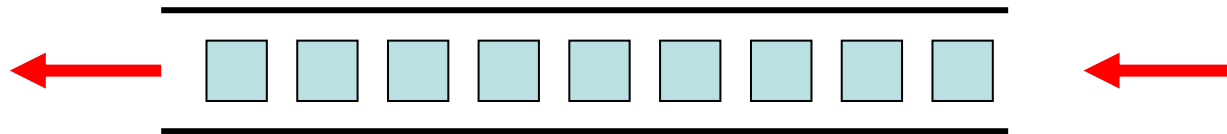
1. Topologische Sortierung der Knoten
2. Aktualisierung der Distanzen gemäß der topologischen Sortierung

Wie führe ich eine topologische Sortierung durch?

Kürzeste Wege in DAGs

Topologische Sortierung:

- Verwende eine FIFO Queue q

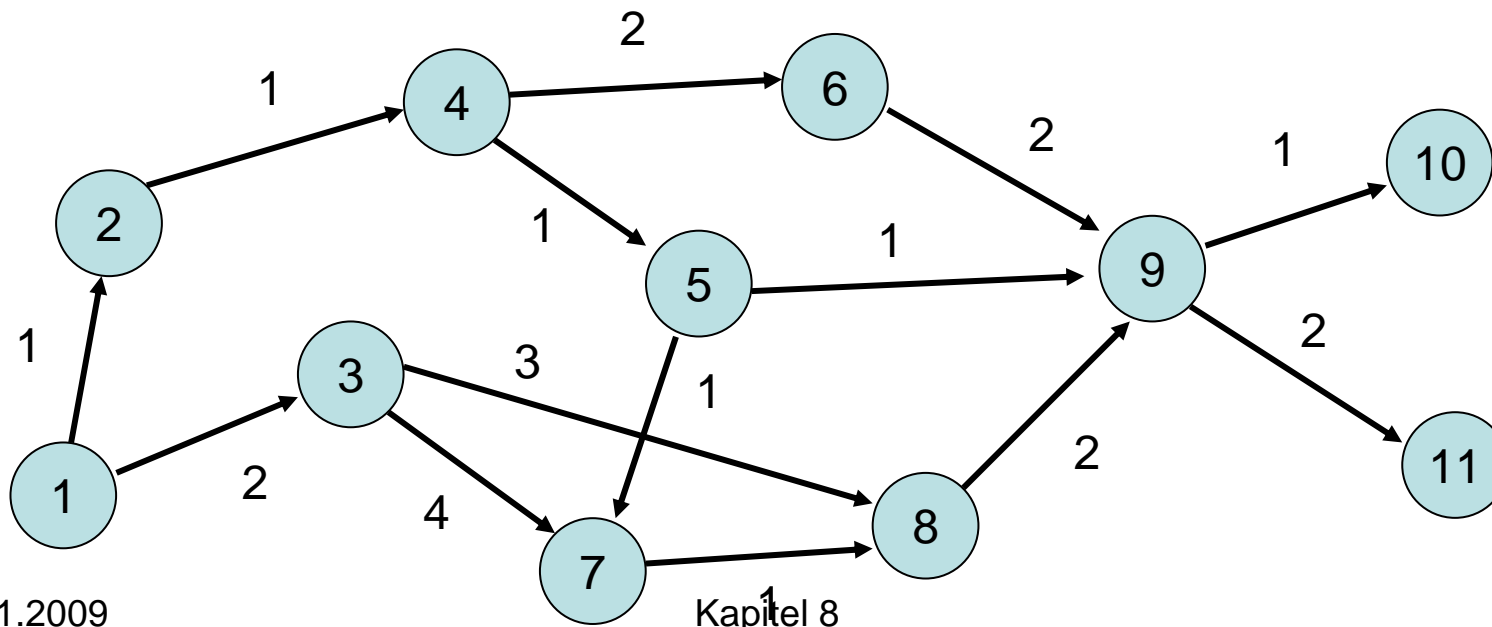


- Anfangs enthält q alle Knoten, die **keine** eingehende Kante haben (Quellen).
- Entnehme v aus q und markiere alle $(v,w) \in E$. Falls alle Kanten nach w markiert sind und w noch nicht in q war, füge w in q ein. Wiederhole das, bis q leer ist.

Kürzeste Wege in DAGs

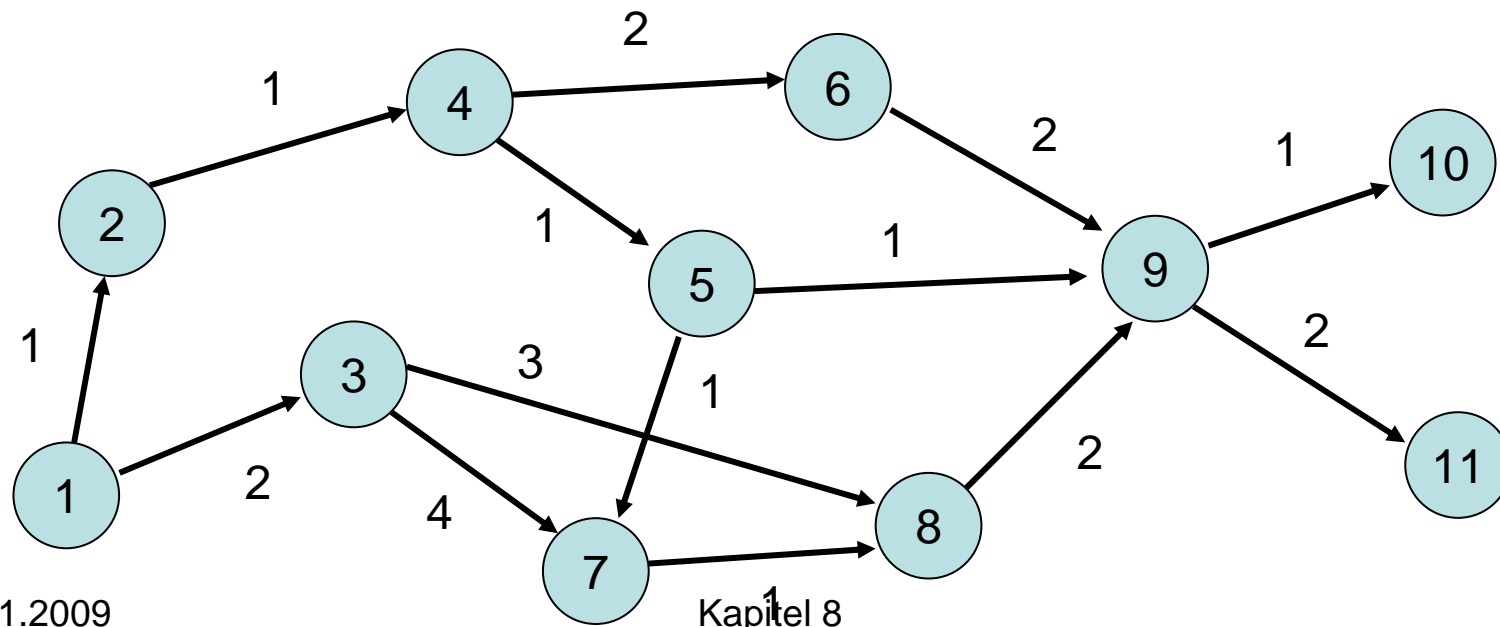
Beispiel:

- **●**: Knoten momentan in Queue q
- Nummerierung nach Einfügereihenfolge



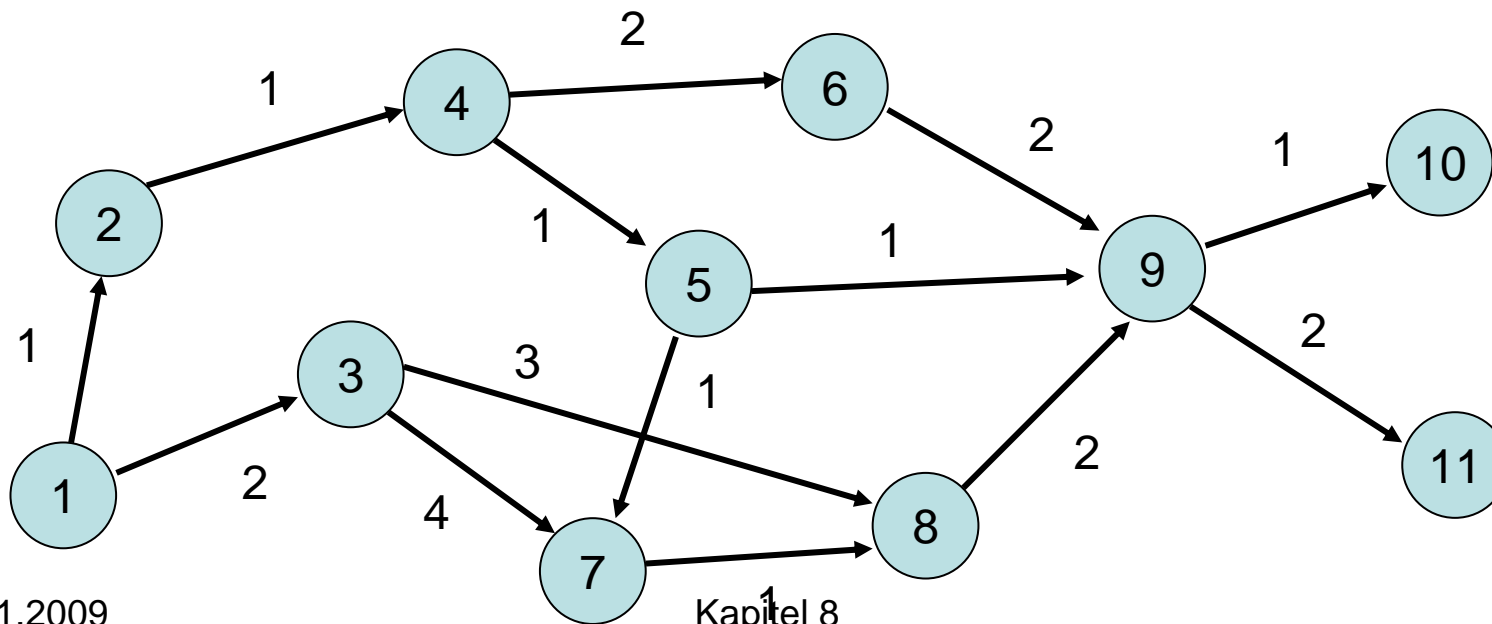
Kürzeste Wege in DAGs

Korrektheit der topologischen Nummerierung:
Knoten wird erst dann nummeriert, wenn alle
Vorgänger nummeriert sind.



Kürzeste Wege in DAGs

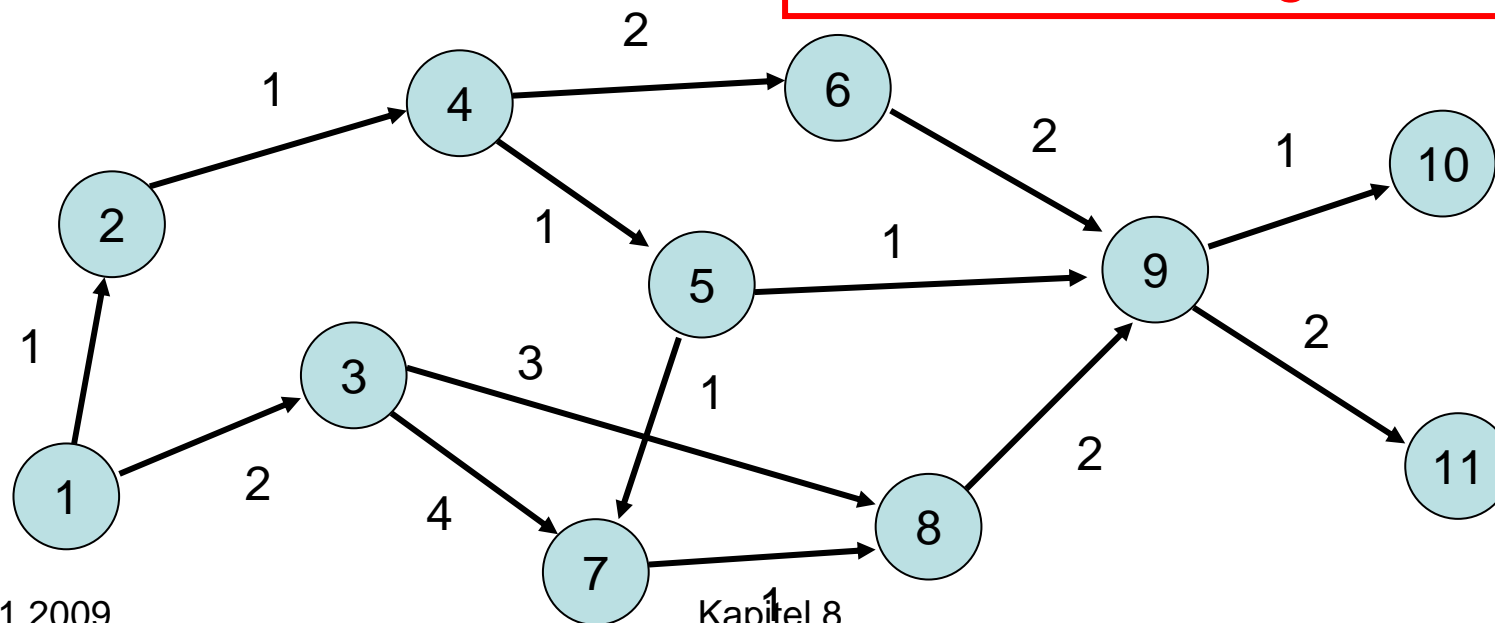
Laufzeit: Zur Bestimmung aller Knoten ohne eingehende Kante muss Graph einmal durchlaufen werden. Danach wird jeder Knoten und jede Kante genau einmal betrachtet, also Zeit $O(n+m)$.



Kürzeste Wege in DAGs

Bemerkung: topologische Sortierung kann nicht alle Knoten nummerieren genau dann, wenn Graph gerichteten Kreis enthält (kein DAG ist)

Test auf DAG-Eigenschaft



Kürzeste Wege in DAGs

DAG-Strategie:

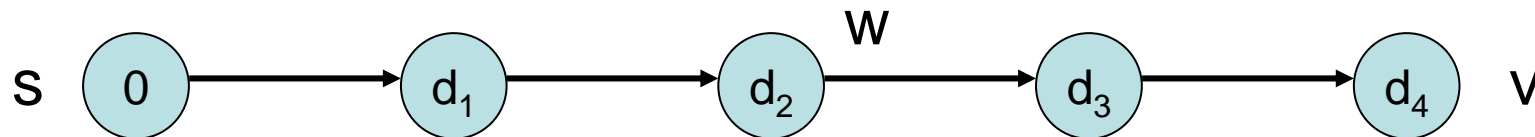
1. Topologische Sortierung der Knoten
Laufzeit $O(n+m)$
2. Aktualisierung der Distanzen gemäß der topologischen Sortierung
Laufzeit $O(n+m)$

Insgesamt Laufzeit $O(n+m)$.

Dijkstras Algorithmus

Nächster Schritt: Kürzeste Wege für beliebige Graphen mit positiven Kanten.

Problem: besuche Knoten eines kürzesten Weges in richtiger Reihenfolge



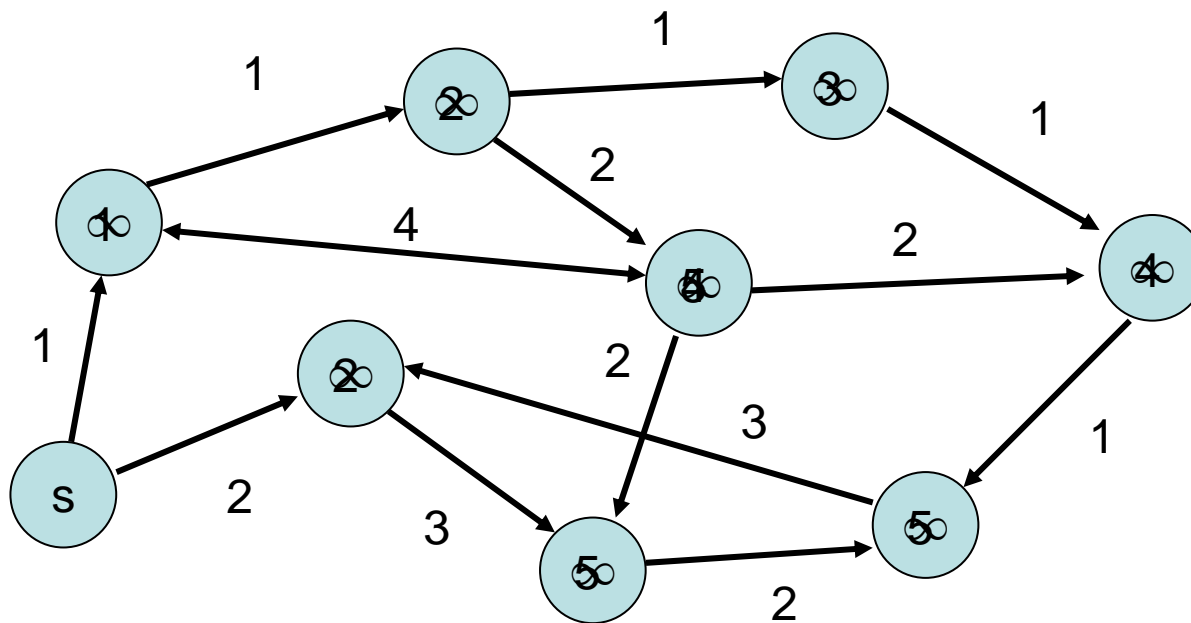
Lösung: besuche Knoten in der Reihenfolge der kürzesten Distanz zur Quelle s

Dijkstras Algorithmus

- Am Anfang, setze $d(s) := 0$ und $d(v) := \infty$ für alle Knoten. Füge s in Priority Queue q ein, wobei die Prioritäten in q gemäß der aktuellen Distanzen $d(v)$ definiert sind.
- Wiederhole, bis q leer ist:
Entferne aus q (deleteMin) den Knoten v mit niedrigstem $d(v)$. Für alle $(v, w) \in E$, setze $d(w) := \min\{d(w), d(v) + c(v, w)\}$. Falls w noch nicht in q war, füge w in q ein.

Dijkstras Algorithmus

Beispiel: (●: aktuell, ●: fertig)



Dijkstras Algorithmus

```
Procedure Dijkstra(s: NodeId)
  d=< $\infty$ ,..., $\infty$ >: NodeArray of  $\mathbb{R} \cup \{-\infty, \infty\}$ 
  parent=< $\perp$ ,..., $\perp$ >: NodeArray of NodeId
  d[s]:=0; parent[s]:=s
  q=<s>: NodePQ
  while q  $\neq \emptyset$  do
    u:=q.deleteMin() // u: min. Distanz zu s in q
    foreach e=(u,v)  $\in E$  do
      if d[v] > d[u]+c(e) then // aktualisiere d[v]
        if d[v]= $\infty$  then q.insert(v) // v schon in q?
        d[v]:=d[u]+c(e); parent[v]:=u
        q.decreaseKey(v) // nur PQ q reparieren
```

Dijkstras Algorithmus

Korrektheit:

- Angenommen, Algo sei inkorrekt.
- Sei v der Knoten mit kleinstem $\mu(s,v)$, der aus q entfernt wird mit $d[v] > \mu(s,v)$.
- Sei $p = (s = v_1, v_2, \dots, v_k = v)$ ein kürzester Weg von s nach v .
- Weil alle $c(e) > 0$ sind, steigt minimales $d[w]$ in q monoton an. Da $d[v_{k-1}] < d[v_k]$, muss v_{k-1} vor v_k aus q entfernt worden sein, und wegen der Annahme galt $d[v_{k-1}] = \mu(s, v_{k-1})$.
- Bei Entfernung von v_{k-1} hat Algo überprüft, ob $d[v_{k-1}] + c(v_{k-1}, v_k) < d[v_k]$. Das muss bei Annahme oben der Fall gewesen sein, aber dann hätte Algo $d[v]$ auf $d[v_{k-1}] + c(v_{k-1}, v) = \mu(s, v)$ gesetzt, ein Widerspruch.

Dijkstras Algorithmus

Laufzeit:

$$T_{\text{Dijkstra}} = O(n(T_{\text{DeleteMin}}(n) + T_{\text{Insert}}(n)) + m \cdot T_{\text{decreaseKey}}(n))$$

Binärer Heap: alle Operationen $O(\log n)$, also

$$T_{\text{Dijkstra}} = O((m+n)\log n)$$

Fibonacci Heap:

- $T_{\text{DeleteMin}}(n) = T_{\text{Insert}}(n) = O(\log n)$
- $T_{\text{decreaseKey}}(n) = O(1)$
- Damit $T_{\text{Dijkstra}} = O(n \log n + m)$

Monotone Priority Queues

Einsicht: Dijkstras Algorithmus braucht keine allgemeine Priority Queue, sondern nur eine monotone Priority Queue.

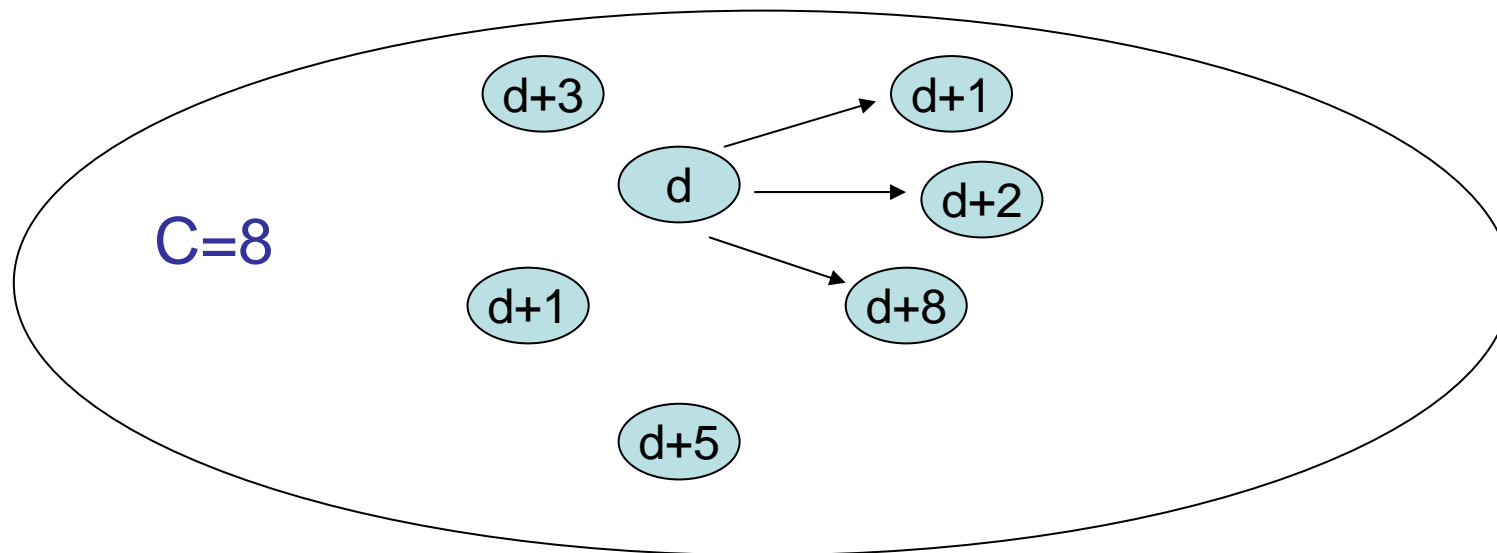
Monotone Priority Queue: Folge der gelöschten Elemente hat monoton steigende Werte.

Effiziente Implementierungen von monotonen Priority Queues möglich, falls Kantenkosten ganzzahlig.

Bucket Queue

Annahme: alle Kantenkosten im Bereich $[0, C]$.

Konsequenz: zu jedem Zeitpunkt enthält q Distanzen im Bereich $[d, d+C]$ für ein d .

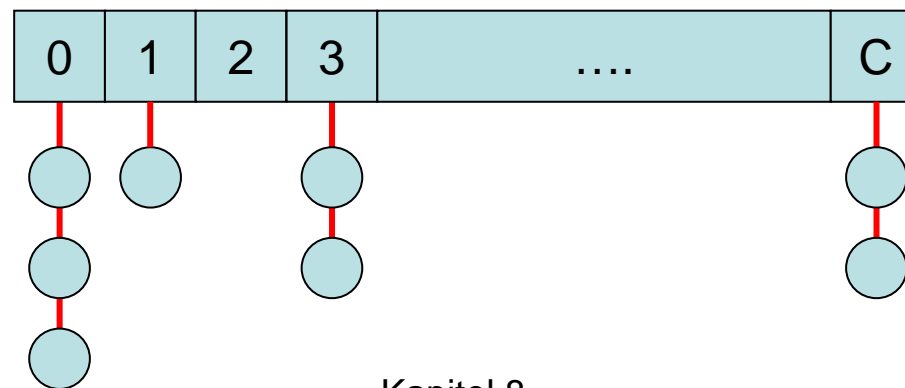


Bucket Queue

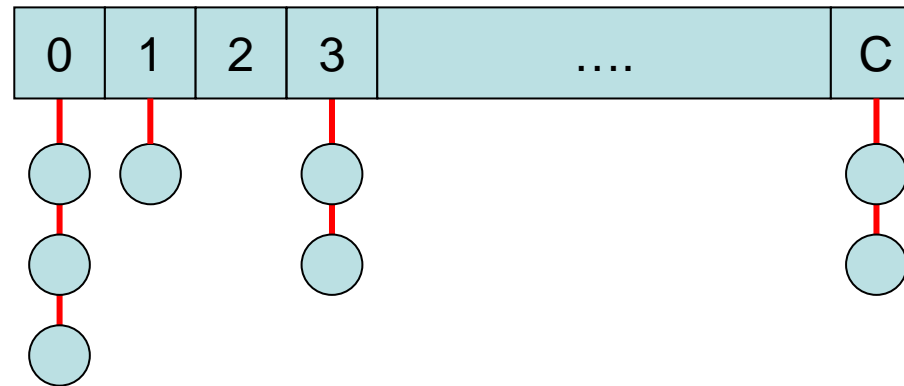
Annahme: alle Kantenkosten im Bereich $[0, C]$.

Konsequenz: zu jedem Zeitpunkt enthält q Distanzen im Bereich $[d, d+C]$ für ein d .

Bucket Queue: Array B aus $C+1$ Listen und Variable d_{\min} für aktuell minimale Distanz ($\text{mod } C+1$)

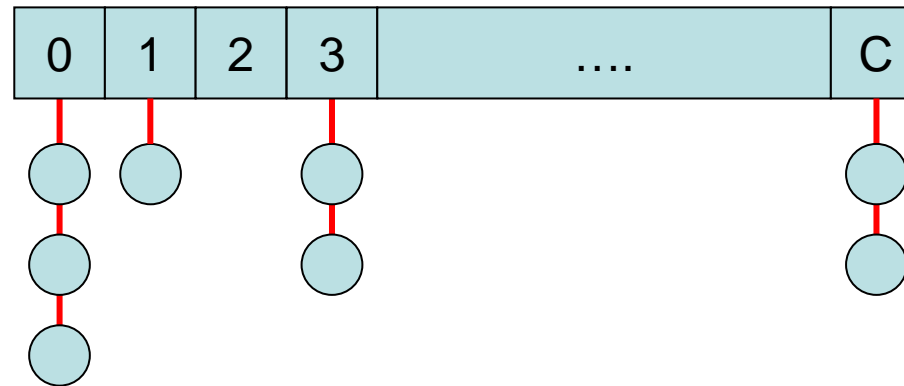


Bucket Queue



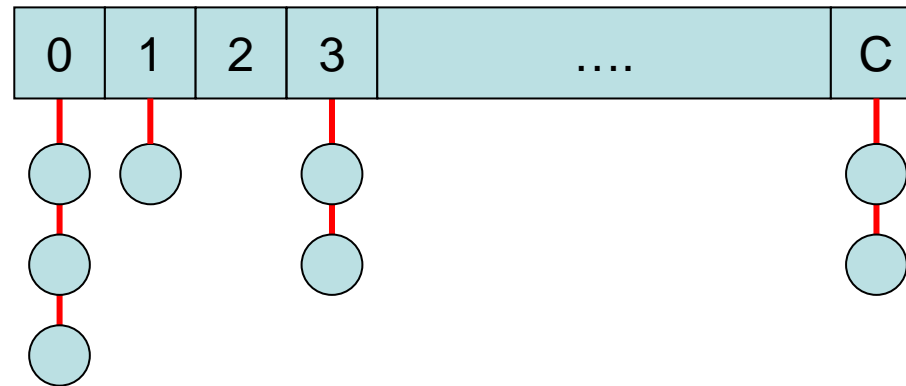
- Jeder Knoten v mit momentaner Distanz $d[v]$ wird in Liste $B[d[v] \bmod (C+1)]$ gespeichert
- Da momentane $d[v]$'s im Bereich $[d, d+C]$ für ein d sind, haben alle v in einem $B[d]$ dasselbe $d[v]$.

Bucket Queue



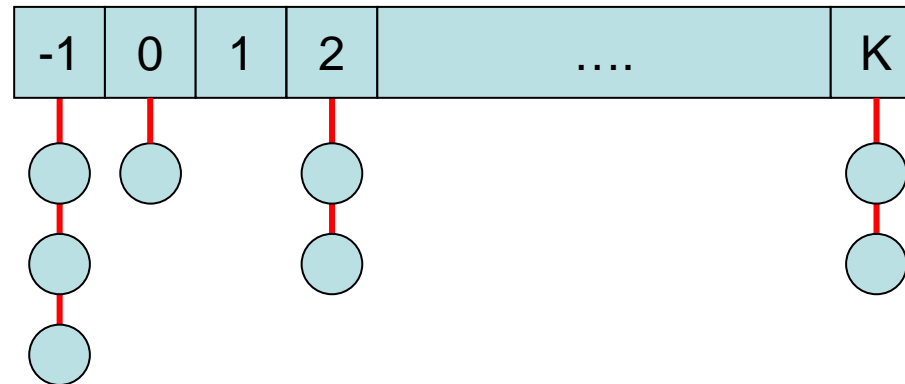
- **Insert(v)**: fügt v in $B[d[v] \bmod (C+1)]$ ein. Laufzeit $O(1)$.
- **decreaseKey(v)**: entfernt v aus momentaner Liste und fügt v in $B[d[v] \bmod (C+1)]$ ein. Falls in v Position in der Liste speichert, Laufzeit $O(1)$.
- **deleteMin()**: solange $B[d_{\min}] = \emptyset$, setze $d_{\min} := (d_{\min} + 1) \bmod (C+1)$. Nimm dann einen Knoten u aus $B[d_{\min}]$ heraus. Laufzeit $O(C)$.

Bucket Queue



- $\text{Insert}(v)$, $\text{decreaseKey}(v)$: Laufzeit $O(1)$.
- $\text{deleteMin}()$: Laufzeit $O(C)$.
- Laufzeit von Dijkstras Algo mit Bucket Queue: $O(m+n \cdot C)$

Radix Heap

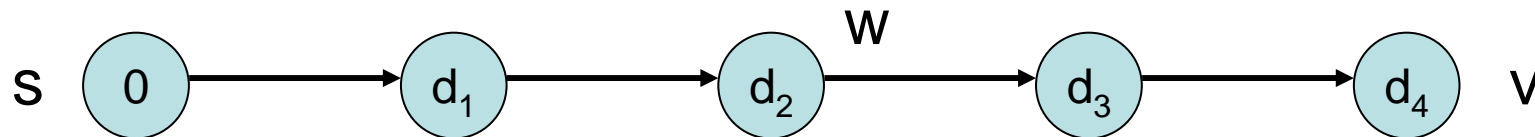


- $\text{Insert}(v)$, $\text{decreaseKey}(v)$: Laufzeit $O(1)$.
- $\text{deleteMin}()$: amortisierte Laufzeit $O(\log C)$.
- Laufzeit von Dijkstras Algo mit Radix Heap: $O(m + n \log C)$

Bellman-Ford Algorithmus

Nächster Schritt: Kürzeste Wege für beliebige Graphen mit beliebigen Kantenkosten.

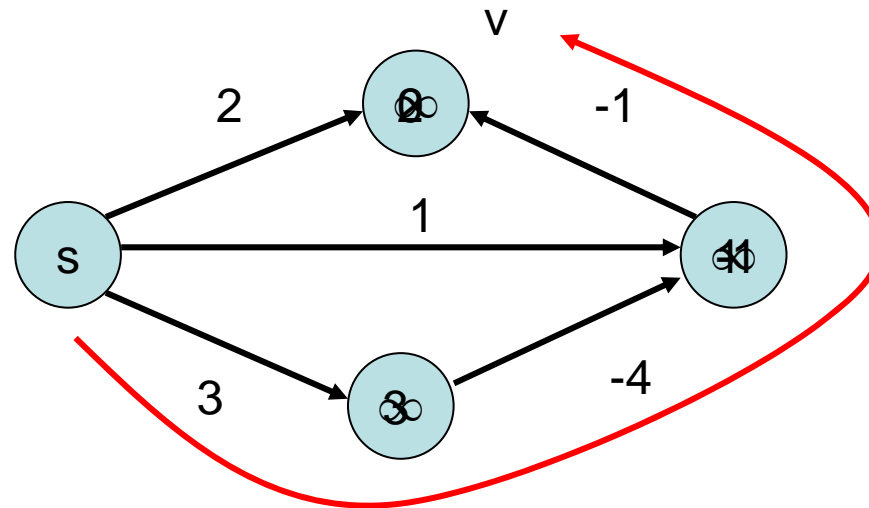
Problem: besuche Knoten eines kürzesten Weges in richtiger Reihenfolge



Dijkstra Algo kann nicht mehr verwendet werden, da er im Allgemeinen nicht mehr Knoten in der Reihenfolge ihrer Distanz zu s besucht.

Bellman-Ford Algorithmus

Beispiel für Problem mit Dijkstra Algo:



Knoten **v** hat falschen Distanzwert!

Bellman-Ford Algorithmus

Lemma 8.1: Für jeden Knoten v mit $\mu(s,v) > -\infty$ zu s gibt es **einfachen** Weg (ohne Kreis!) von s nach v der Länge $\mu(s,v)$.

Beweis:

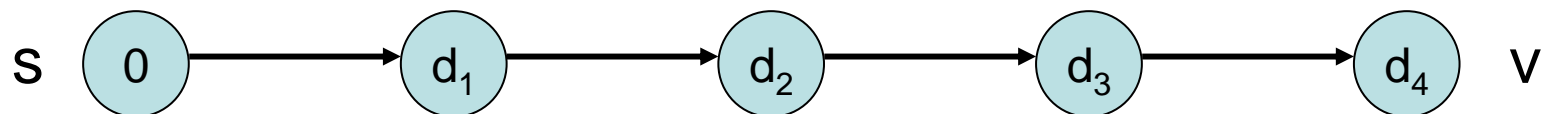
- Weg mit Kreis mit Kantenkosten ≥ 0 :
Kreisdistanz erhöht nicht die Kosten
- Weg mit Kreis mit Kantenkosten < 0 :
Distanz zu s ist $-\infty$!

Bellman-Ford Algorithmus

Folgerung: (Graph mit n Knoten)

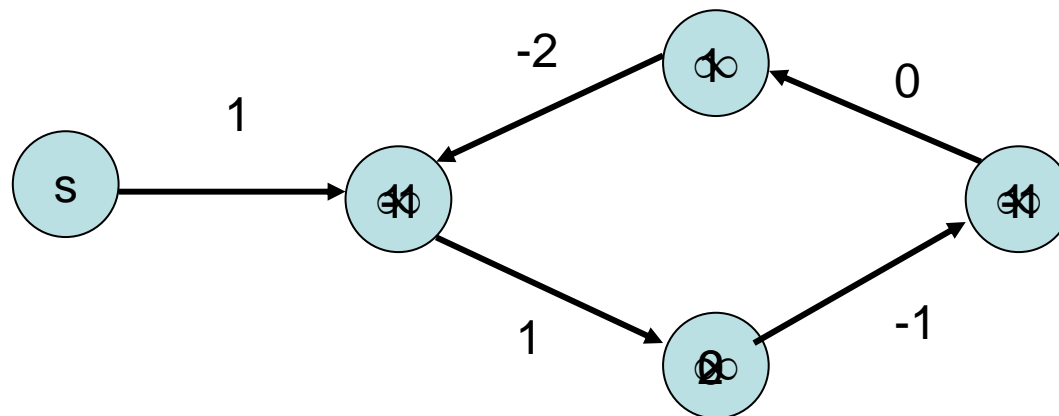
Für jeden Knoten v mit $\mu(s,v) > -\infty$ gibt es kürzesten Weg der Länge $< n$ zu v .

Strategie: Durchlaufe $(n-1)$ -mal **sämtliche Kanten** in Graph und aktualisiere Distanz. Dann alle kürzeste Wege berücksichtigt.



Bellman-Ford Algorithmus

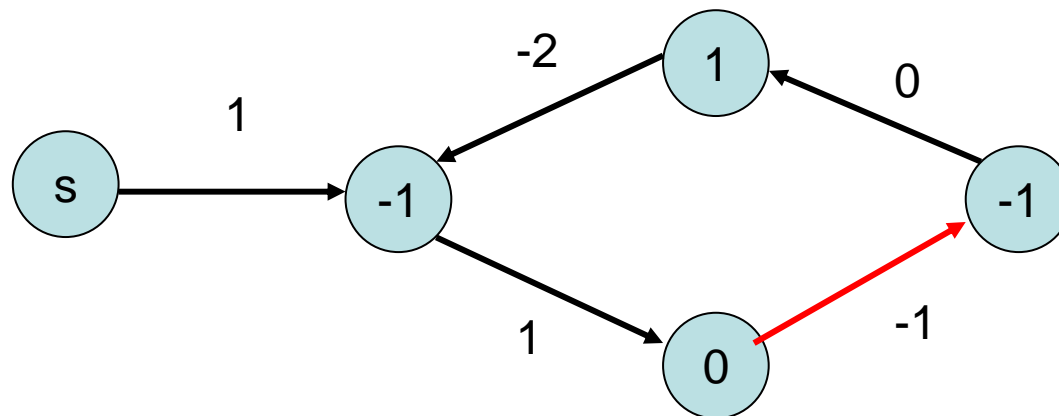
Problem: Erkennung negativer Kreise



Einsicht: in negativem Kreis **erniedrigt sich Distanz**
in jeder Runde bei mindestens einem Knoten

Bellman-Ford Algorithmus

Problem: Erkennung negativer Kreise



Zeitpunkt: kontinuierliche Distanzniedrigung
startet spätestens in n -ter Runde (dann Kreis
mindestens einmal durchlaufen)

Bellman-Ford Algorithmus

Keine Distanzniedrigung möglich:

- Angenommen wir erreichen Zeitpunkt mit $d[v] + c(v, w) \geq d[w]$ für alle Knoten w .
- Dann gilt (über Induktion) für jeden Weg p , dass $d[s] + c(p) \geq d[w]$ für alle Knoten w .
- Falls sichergestellt ist, dass für den kürzesten Weg p nach w , $d[w] \geq c(p)$ zu jedem Zeitpunkt ist, dann gilt am Ende $d[w] = \mu(s, w)$.

Bellman-Ford Algorithmus

Zusammenfassung:

- **Keine Distanzniedrigung** mehr möglich
($d[v]+c(v,w) \geq d[w]$ für alle w):
Fertig, $d[w]=\mu(s,w)$ für alle w
- **Distanzniedrigung möglich** selbst noch in n -ter
Runde ($d[v]+c(v,w) < d[w]$ für ein w):
Dann gibt es negative Kreise, also Knoten w mit
Distanz $\mu(s,w)=-\infty$. Ist das wahr für ein w , dann
für alle von w erreichbaren Knoten.

Bellman-Ford Algorithmus

```
Procedure BellmanFord(s: NodeId)
  d=< $\infty$ ,..., $\infty$ >: NodeArray of  $\mathbb{R} \cup \{-\infty, \infty\}$ 
  parent=< $\perp$ ,..., $\perp$ >: NodeArray of NodeId
  d[s]:=0; parent[s]:=s
  for i:=1 to n-1 do // aktualisiere Kosten für n-1 Runden
    forall e=(v,w)  $\in$  E do
      if d[w] > d[v]+c(e) then // bessere Distanz möglich?
        d[w]:=d[v]+c(e); parent[w]:=v
  forall e=(v,w)  $\in$  E do // in n-ter Runde noch besser?
    if d[w] > d[v]+c(e) then infect(w)
```

```
Procedure infect(v) // setze  $-\infty$ -Kosten von v aus
  if d[v]> $-\infty$  then
    d[v]:= $-\infty$ 
    forall (v,w)  $\in$  E do infect(w)
```

Bellman-Ford Algorithmus

Laufzeit: $O(n \cdot m)$

Verbesserungsmöglichkeiten:

- Überprüfe in jeder Aktualisierungsrunde, ob noch irgendwo $d[v] + c[v, w] < d[w]$ ist.
Nein: fertig!
- Besuche in jeder Runde nur die Knoten w , für die Test $d[v] + c[v, w] < d[w]$ sinnvoll (d.h. $d[v]$ hat sich in letzter Runde geändert).

All Pairs Shortest Paths

Annahme: Graph mit beliebigen Kantenkosten, aber keine negativen Kreise

Naive Strategie für Graph mit n Knoten: lass n -mal Bellman-Ford Algorithmus (einmal für jeden Knoten) laufen

Laufzeit: $O(n^2 m)$

All Pairs Shortest Paths

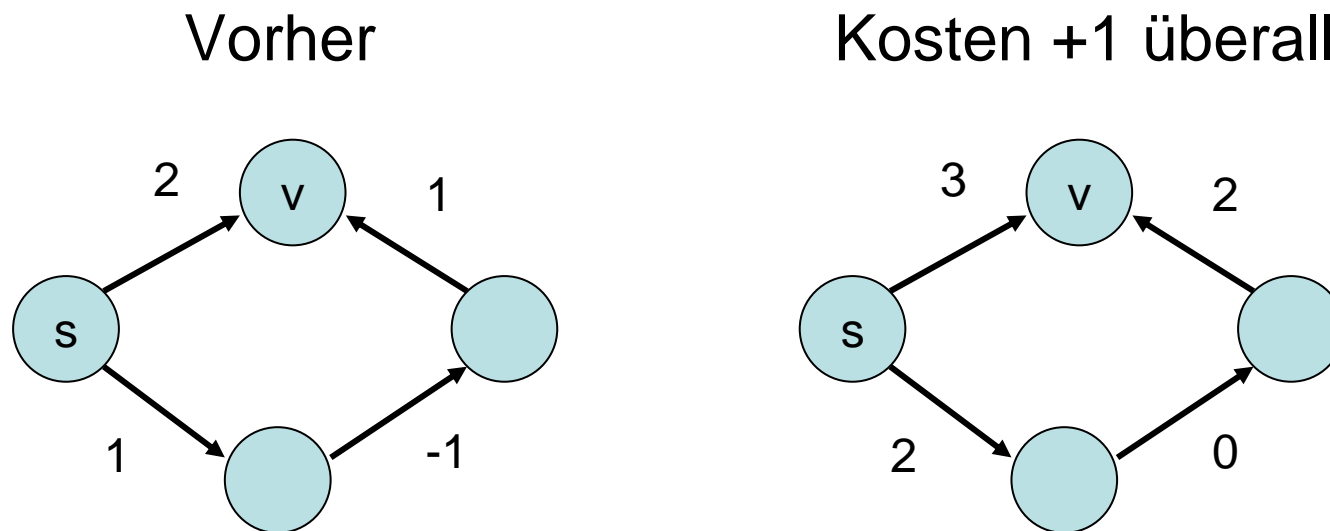
Bessere Strategie: Reduziere n Bellman-Ford Anwendungen auf n Dijkstra Anwendungen

Problem: wir brauchen dazu **nichtnegative** Kantenkosten

Lösung: Umwandlungsstrategie in nichtnegative Kantenkosten, ohne kürzeste Wege zu verfälschen (nicht so einfach!)

All Pairs Shortest Paths

Gegenbeispiel zur Erhöhung um Wert c :



— : kürzester Weg

All Pairs Shortest Paths

- Sei $\phi: V \rightarrow \mathbb{R}$ eine Funktion, die jedem Knoten ein **Potenzial** zuweist.
- Die **reduzierten Kosten** von $e=(v,w)$ sind:
$$r(e) := \phi(v) + c(e) - \phi(w)$$

Lemma 8.2: Seien p und q Wege in G . Dann gilt für jedes Potenzial ϕ : $r(p) < r(q)$ genau dann wenn $c(p) < c(q)$.

All Pairs Shortest Paths

Lemma 8.2: Seien p und q Wege in G . Dann gilt für jedes Potenzial ϕ : $r(p) < r(q)$ genau dann wenn $c(p) < c(q)$.

Beweis: Sei $p = (v_1, \dots, v_k)$ ein beliebiger Weg und $e_i = (v_i, v_{i+1})$ für alle i . Es gilt:

$$\begin{aligned} r(p) &= \sum_i r(e_i) \\ &= \sum_i (\phi(v_i) + c(e) - \phi(v_{i+1})) \\ &= \phi(v_1) + c(p) - \phi(v_k) \end{aligned}$$

All Pairs Shortest Paths

Lemma 8.3: Angenommen, G habe keine negativen Kreise und dass alle Knoten von s erreicht werden können. Sei $\phi(v) = \mu(s, v)$ für alle $v \in V$. Mit diesem ϕ ist $r(e) \geq 0$ für alle e .

Beweis:

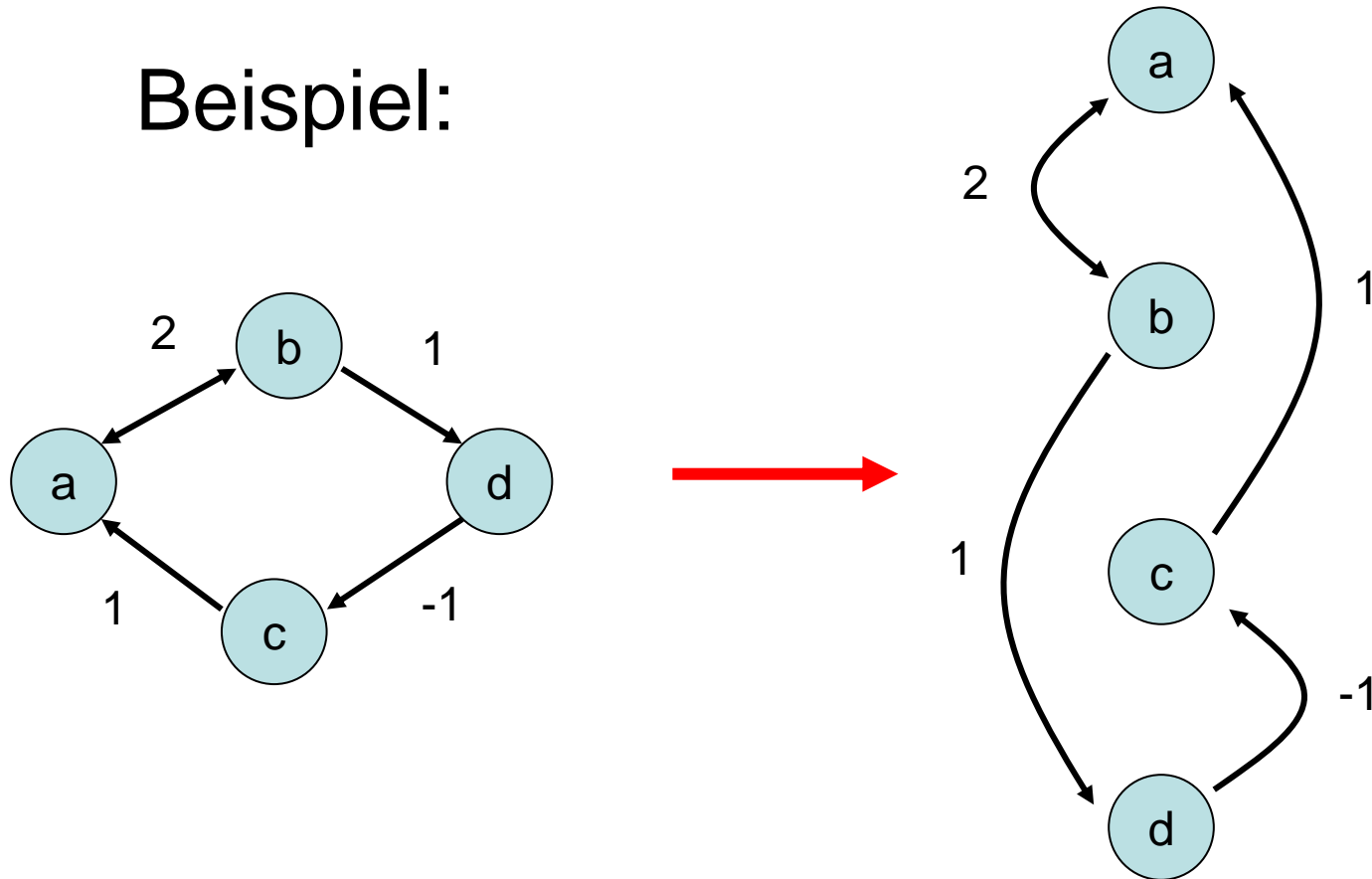
- nach Annahme ist $\mu(s, v) \in \mathbb{R}$ für alle v
- wir wissen: für jede Kante $e = (v, w)$ ist $\mu(s, v) + c(e) \geq \mu(s, w)$ (Abbruchbedingung!)
- also ist $r(e) = \mu(s, v) + c(e) - \mu(s, w) \geq 0$

All Pairs Shortest Paths

1. Füge **neuen** Knoten s und Kanten (s,v) für alle v hinzu mit $c(s,v)=0$ (**alle erreichbar!**)
2. Berechne $\mu(s,v)$ nach **Bellman-Ford** und setze $\phi(v):=\mu(s,v)$ für alle v
3. Berechne die reduzierten Kosten $r(e)$
4. Berechne für alle Knoten v die Distanzen $\bar{\mu}(v,w)$ mittels **Dijkstra Algo** mit reduzierten Kosten auf Graph **ohne Knoten s**
5. Berechne korrekte Distanzen $\mu(v,w)$ durch $\mu(v,w)=\bar{\mu}(v,w)+\phi(w)-\phi(v)$

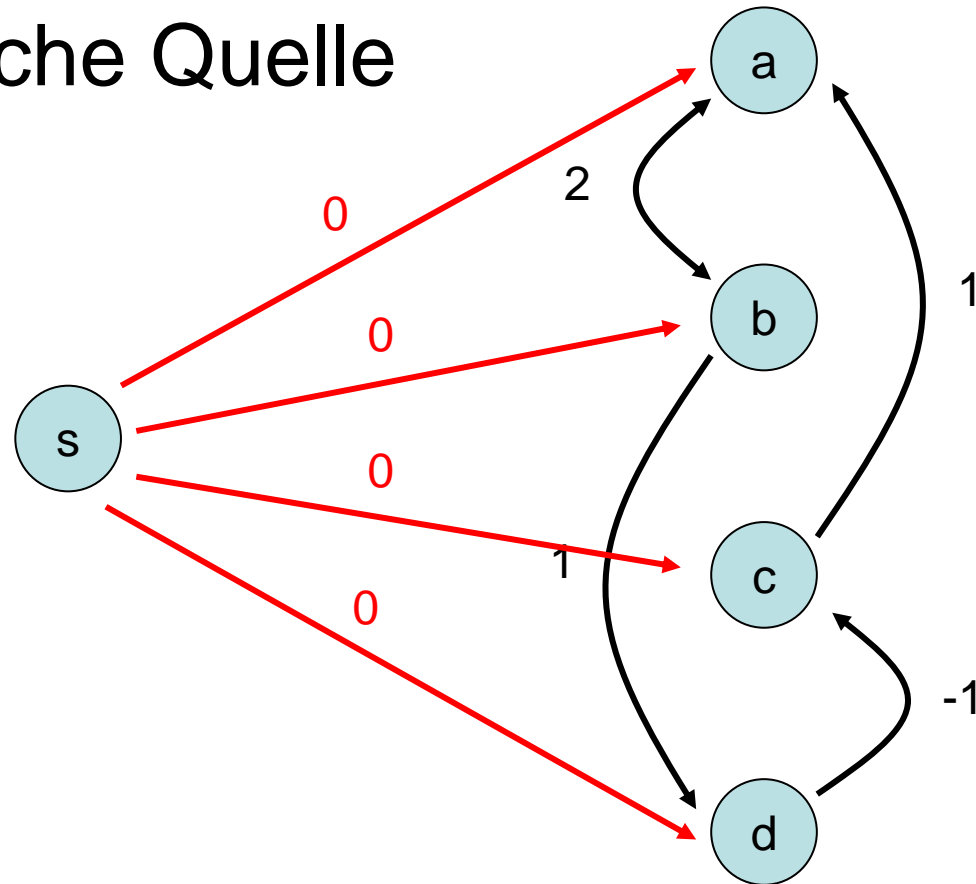
All Pairs Shortest Paths

Beispiel:



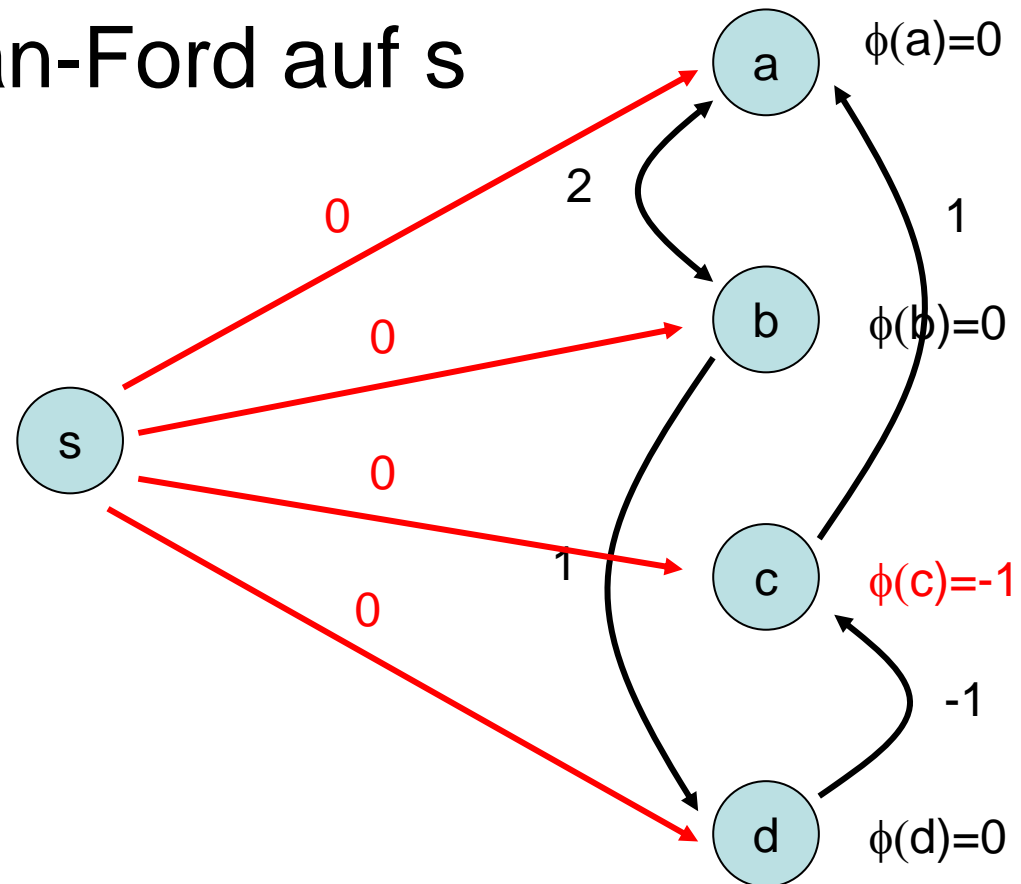
All Pairs Shortest Paths

Schritt 1: künstliche Quelle



All Pairs Shortest Paths

Schritt 2: Bellman-Ford auf s

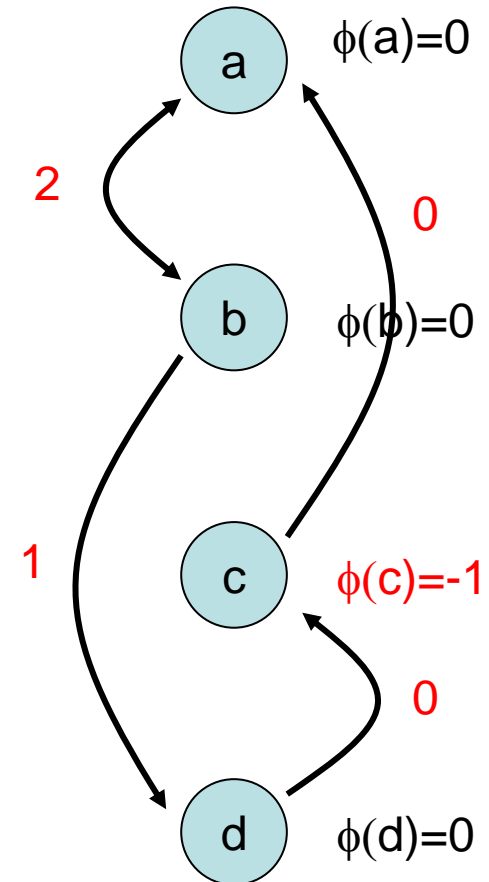


All Pairs Shortest Paths

Schritt 3: $r(e)$ -Werte berechnen

Die **reduzierten Kosten** von $e=(v,w)$ sind:

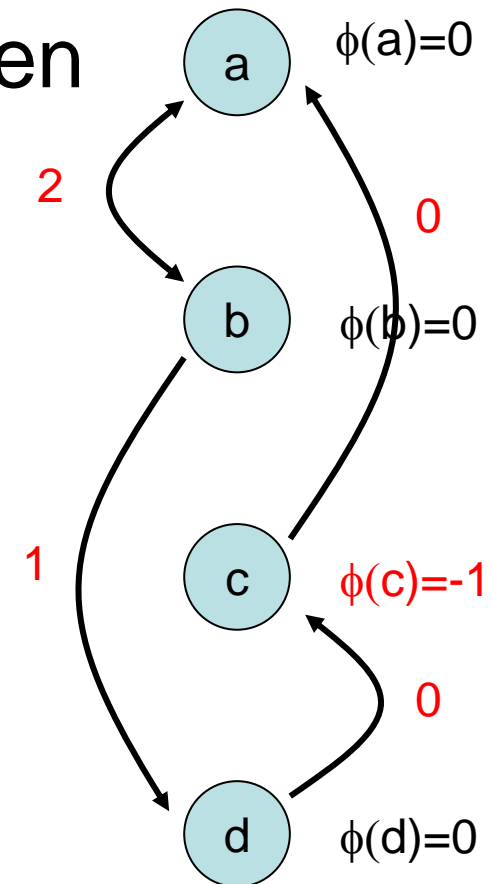
$$r(e) := \phi(v) + c(e) - \phi(w)$$



All Pairs Shortest Paths

Schritt 4: berechne alle Distanzen $\bar{\mu}(v,w)$ via Dijkstra

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

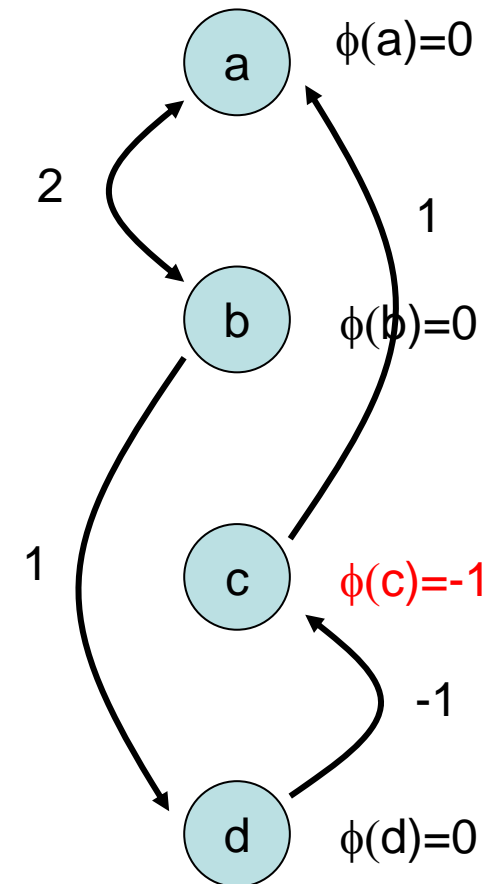


All Pairs Shortest Paths

Schritt 5: berechne korrekte Distanzen durch die Formel

$$\mu(v,w) = \bar{\mu}(v,w) + \phi(w) - \phi(v)$$

μ	a	b	c	d
a	0	2	2	3
b	1	0	0	1
c	1	3	0	4
d	0	2	-1	0



All Pairs Shortest Paths

Laufzeit des APSP-Algorithmus:

$$\begin{aligned} & O(T_{\text{Bellman-Ford}}(n,m) + n \cdot T_{\text{Dijkstra}}(n,m)) \\ & = O(n \cdot m + n(n \log n + m)) \\ & = O(n \cdot m + n^2 \log n) \end{aligned}$$

unter Verwendung von Fibonacci Heaps.

Transitive Hülle

Problem: Konstruiere für einen gerichteten Graphen $G=(V,E)$ eine Datenstruktur, die die folgende Operation (speicher- und zeit-)effizient unterstützt:

- **Reachable(v,w):** liefert **1**, falls es einen gerichteten Weg von **v** nach **w** in G gibt und sonst **0**

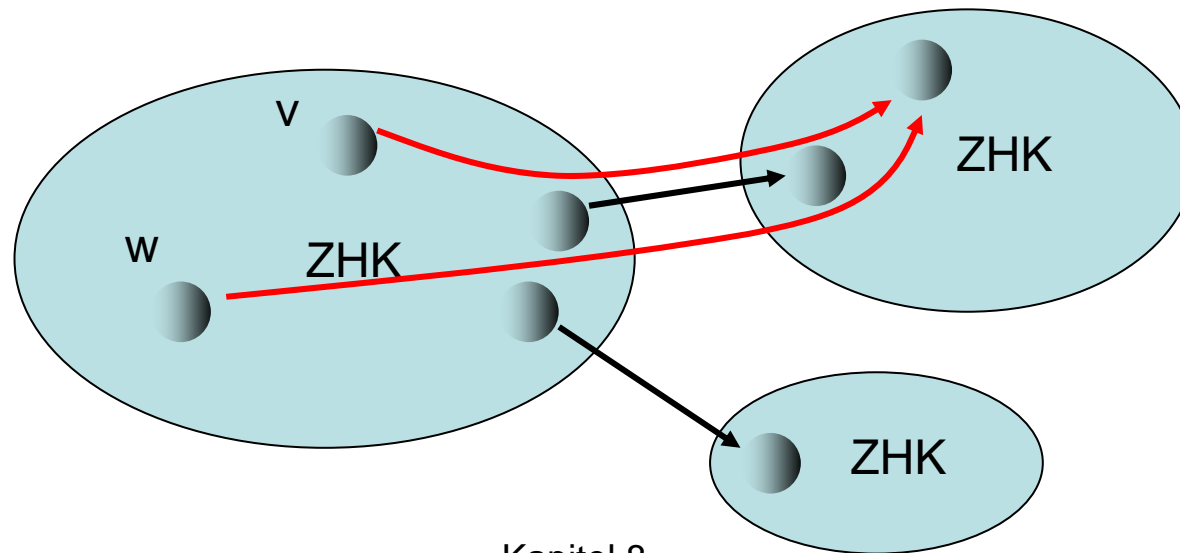
Transitive Hülle

Lösung 1: verwende APSP Algorithmus

- Laufzeit zur Erstellung der DS:
 $O(n \cdot m + n^2 \log n)$
- Speicheraufwand: $O(n^2)$
- Laufzeit von $\text{Reachable}(v,w)$: $O(1)$
(nachschaun in Tabelle, ob $\mu(v,w) < \infty$)

Transitive Hülle

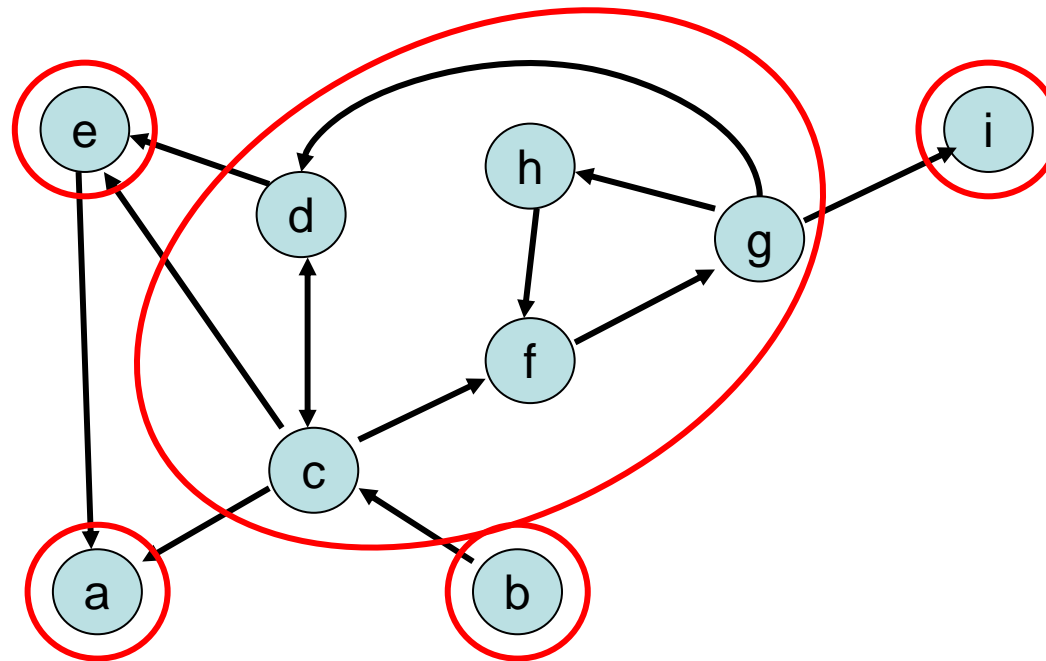
Einsicht: Alle Knoten in einer starken ZHK haben **dieselbe** Menge erreichbarer Knoten. Daher reicht es, sie durch Repräsentanten zu vertreten.



Transitive Hülle

Lösung 2: verwende ZHK-Algorithmus

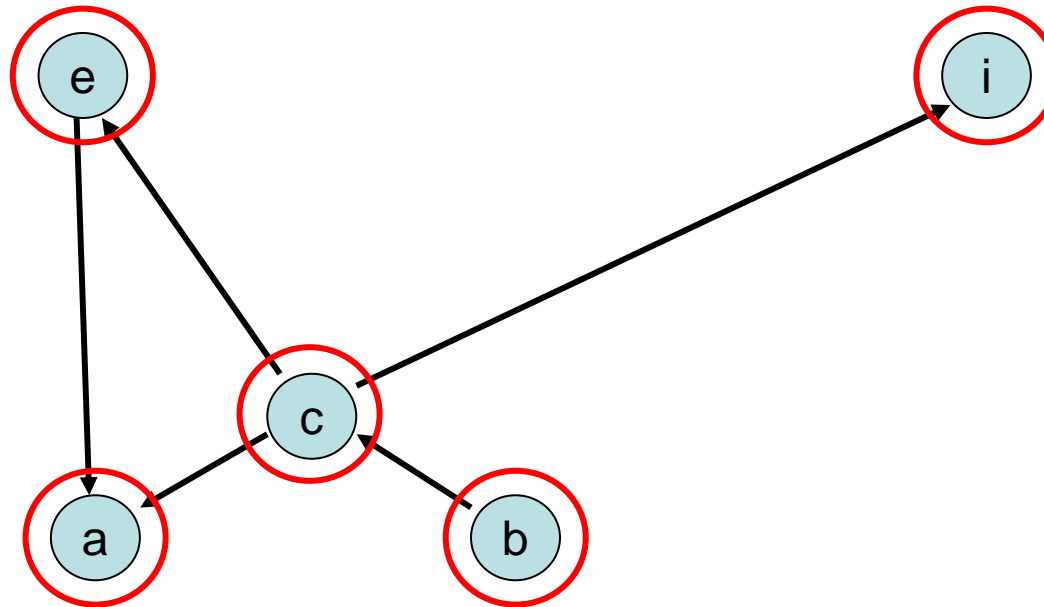
- bestimme starke ZHKs



Transitive Hülle

Lösung 2: verwende ZHK-Algorithmus

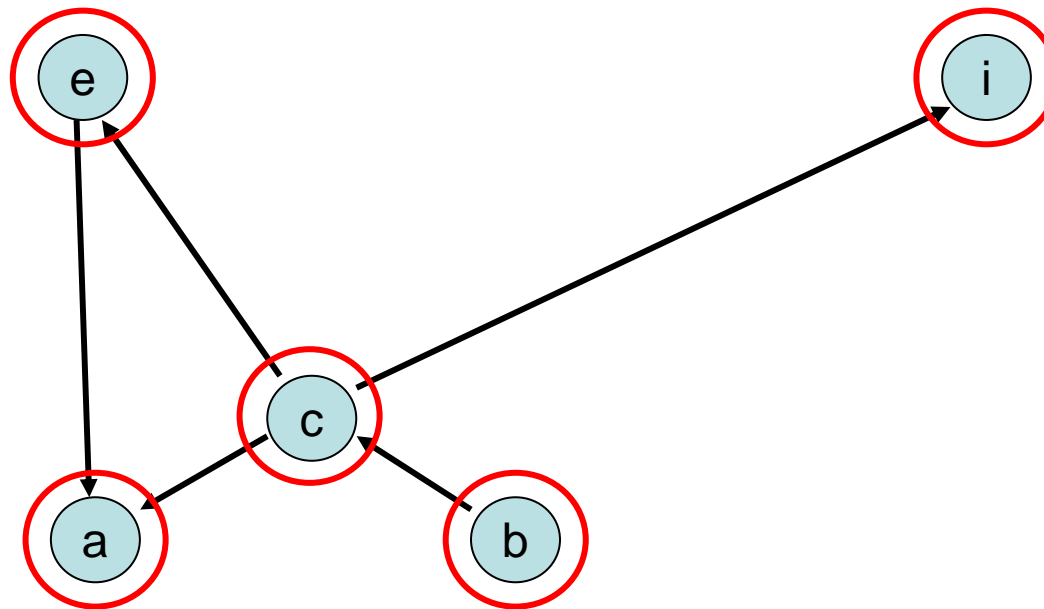
- bestimme ZHK-Graph (Repräsentanten)



Transitive Hülle

Lösung 2: verwende ZHK-Algorithmus

- Wende APSP-Algo auf ZHK-Graph an



Transitive Hülle

Reachable(v, w):

- Bestimme Repräsentanten r_v und r_w von v und w
- $r_v=r_w$: gib 1 aus
- sonst gib $\text{Reachable}(r_v, r_w)$ für ZHK-Graph zurück

Transitive Hülle

- Graph $G=(V,E)$: $n=|V|$, $m=|E|$
- ZHK-Graph $G'=(V',E')$: $n'=|V'|$, $m'=|E'|$

Datenstruktur:

- Berechnungszeit:
 $O(n + m + n' \cdot m' + (n')^2 \log n')$
- Speicher: $O(n + (n')^2)$

Reachable(v,w): Laufzeit $O(1)$

Transitive Hülle

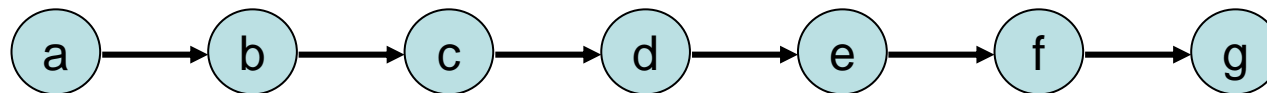
Ist es auch möglich, mit $\sim O(n+m)$ Speicher für die Datenstruktur die Operation $\text{Reachable}(v,w)$ effizient abzuarbeiten?

Einsicht: Wenn für eine topologische Sortierung $(t_v)_{v \in V}$ der Repräsentanten gilt $r_v > r_w$, dann gibt es keinen gerichteten Weg von r_v nach r_w

Was machen wir, falls $r_v < r_w$?

Transitive Hülle

Fall 1: Der ZHK-Graph ist eine gerichtete Liste



Reachable(v,w) ergibt $1 \Leftrightarrow t_v < t_w$

Transitive Hülle

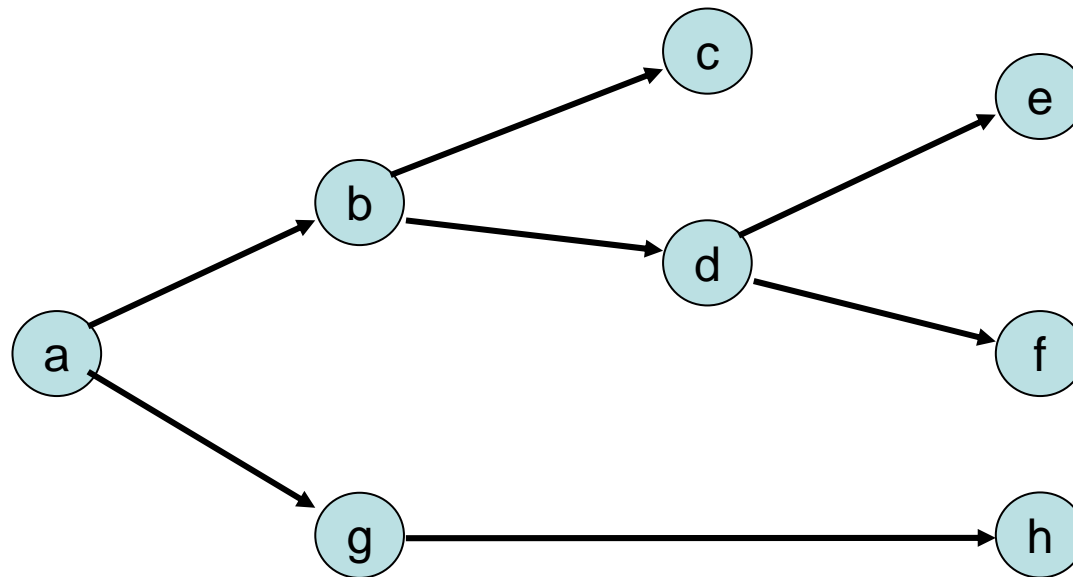
Fall 1: Der ZHK-Graph ist eine gerichtete Liste

Datenstruktur: $O(n+m)$ Zeit, $O(n)$ Speicher
(speichere Repräsentanten zu jedem Knoten und gib Repr. Ordnungsnummern)

Reachable(v,w): Laufzeit $O(1)$

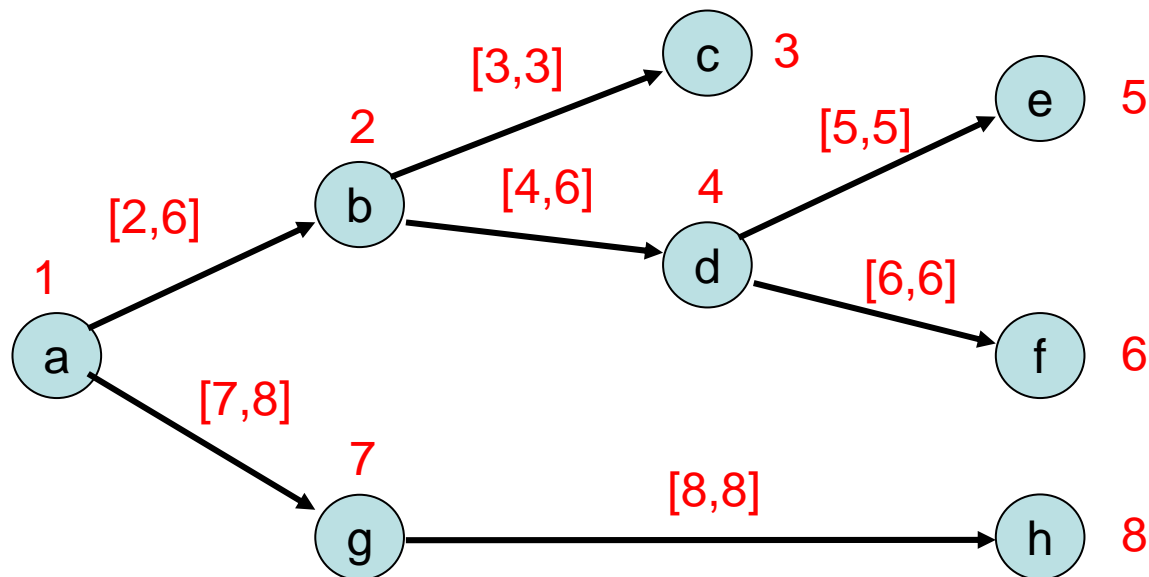
Transitive Hülle

Fall 2: Der ZHK-Graph ist ein gerichteter Baum



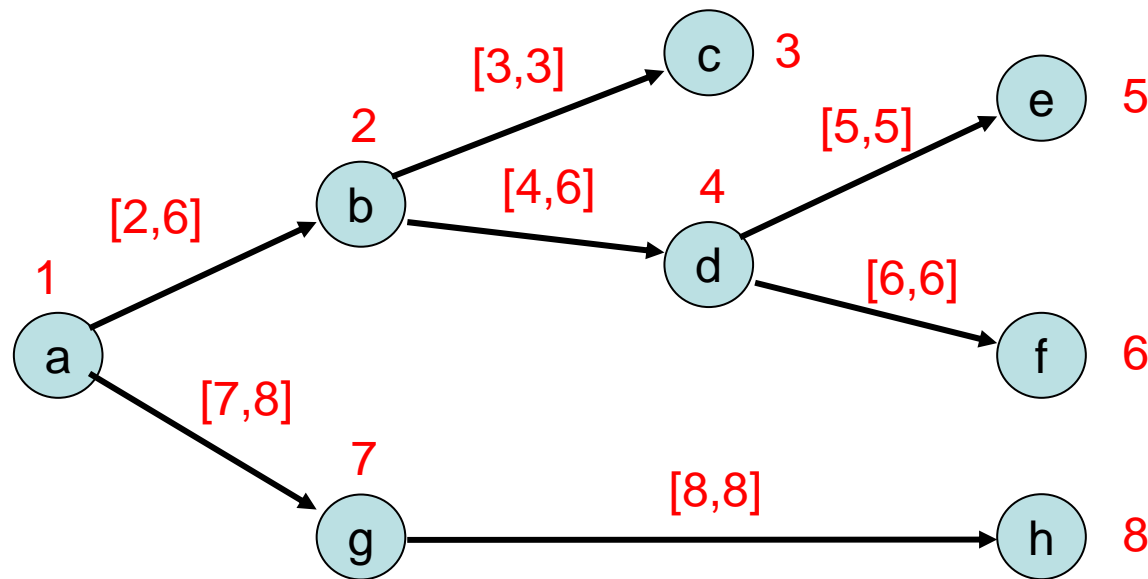
Transitive Hülle

Strategie: DFS-Durchlauf von Wurzel,
Kanten mit dfsnum-Bereichen markieren



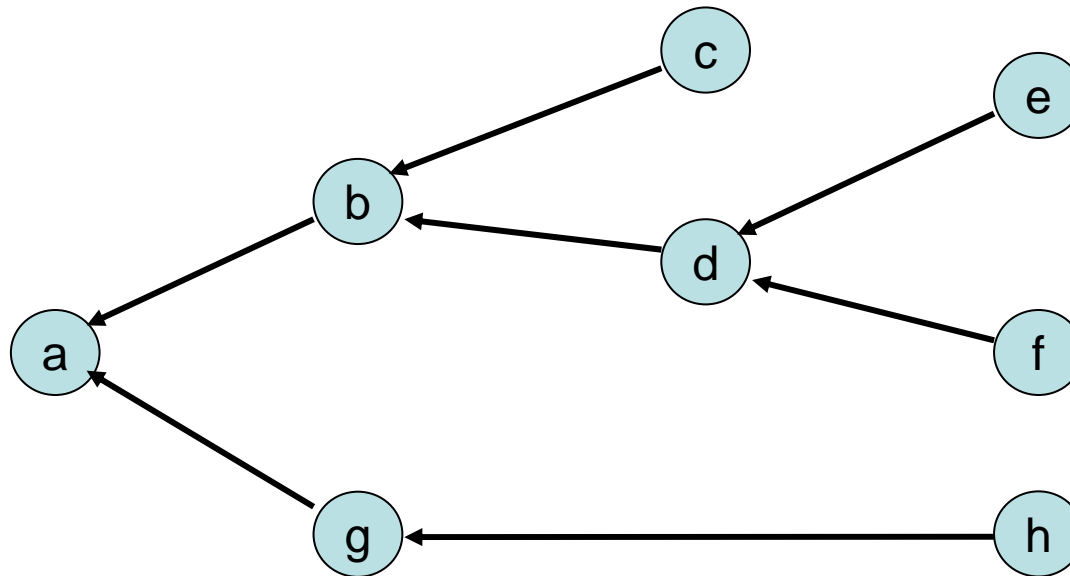
Transitive Hülle

Reachable(v,w): bestimme Repräsentanten r_v und r_w , teste ob r_w in Intervall von ausgehender Kante von r_v



Transitive Hülle

Kantenrichtungen zur Wurzel:



$\text{Reachable}(v,w)$ ist 1 \Leftrightarrow $\text{Reachable}(w,v)$ ist 1 für umgekehrte Richtungen

Transitive Hülle

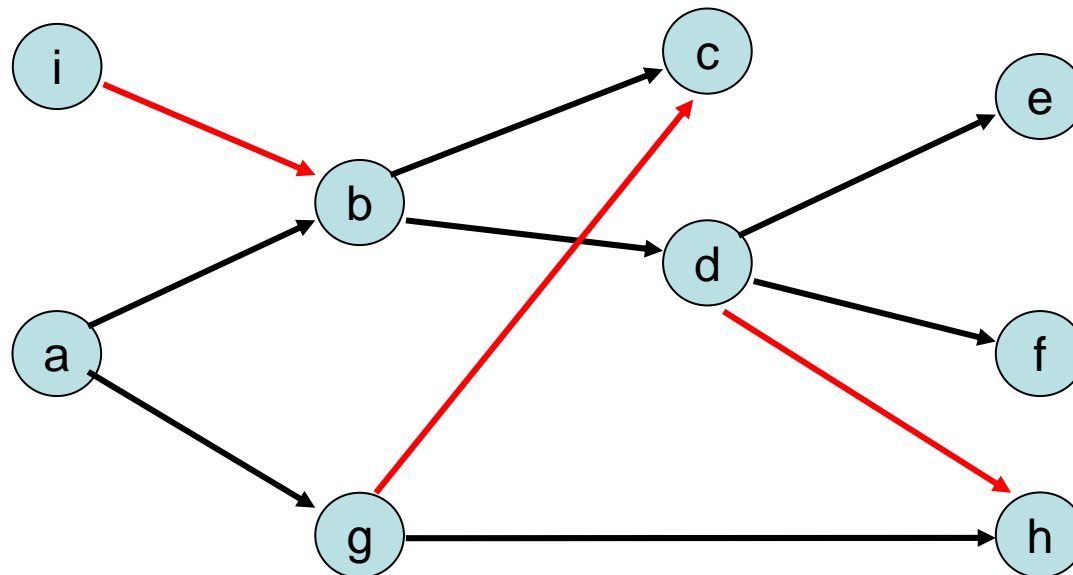
Fall 2: Der ZHK-Graph ist ein gerichteter Baum

Datenstruktur: $O(n+m)$ Zeit und Speicher (speichere Repräsentanten zu jedem Knoten Kantenintervalle zu jedem Repräsentanten)

Reachable(v,w): Laufzeit $O(\log d)$ (binäre Suche auf Intervallen), wobei d der maximale Grad im ZHK-Graph ist

Transitive Hülle

Fall 3: Der ZHK-Graph ist ein beliebiger DAG



Transitive Hülle

Auflösung der (roten) Kreuzkanten problematisch.

Strategie:

- Erzeuge künstliche Wurzel r , die Kanten zu allen Knoten mit Ingrad 0 hat
- Führe DFS-Durchlauf von r durch, wobei bereits besuchte Knoten nochmal besucht werden dürfen (und damit **mehrere** DFS-Nummern erhalten). Siehe dazu die nächste Folie.
- Bestimme Kantenintervalle beim ersten DFS-Durchlauf eines Knotens

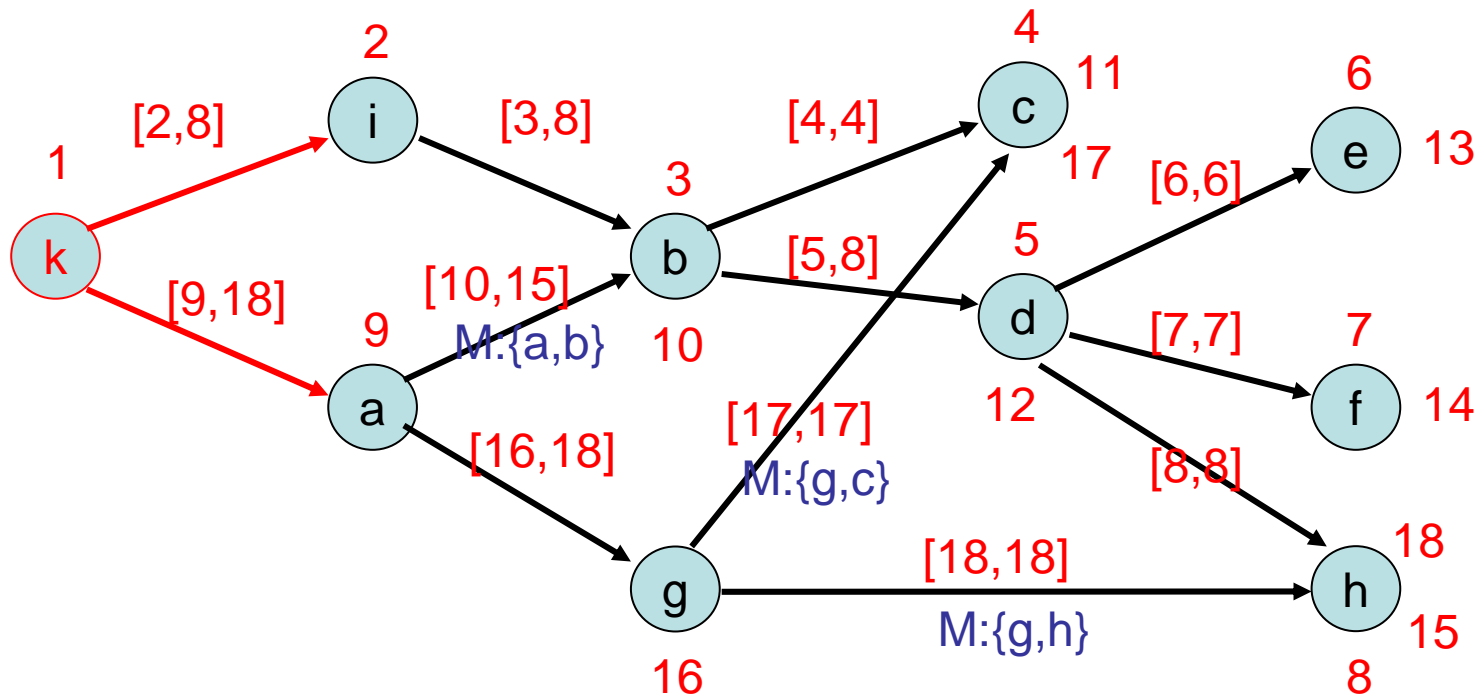
Transitive Hülle

DFS-Durchlauf:

- Mit dem DFS-Durchlauf wird ein Marker M weitergereicht. Jeder Knoten v setzt $M_v := \emptyset$.
- Anfangs ist $M = \emptyset$.
- Wird ein Knoten v beim DFS-Durchlauf besucht, dann unterscheidet v zwischen drei Fällen:
 - $M = \emptyset$: Ist v noch nicht besucht worden, dann setzt v den DFS-Durchlauf bei sich fort. Ist v schon besucht worden, dann setzt v M und M_v auf $\{u, v\}$ und führt den DFS-Durchlauf mit diesem M bei sich fort.
 - Ist $M \neq \emptyset$ und $M \neq M_v$, dann setzt v M_v auf $\{u, v\}$ und führt den DFS-Durchlauf mit M bei sich fort.
 - Sonst endet der DFS-Durchlauf bei v und wird zurückgereicht an den aufrufenden Knoten.

Transitive Hülle

Beispiel:



Transitive Hülle

Reachable(v,w): bestimme r_v und r_w und teste, ob es eine DFS-Nummer x von r_w gibt, so dass x in einem Intervall einer ausgehenden Kante von r_v ist.

Laufzeit: $O(d \log n)$ mit binärer Suche auf sortierter Folge der DFS-Nummern

Problem: Speicher für DS evtl. hoch ($O(n^2)$)

Ausblick

Weiter mit minimalen Spannäumen...