Grundlagen der Algorithmen und Datenstrukturen

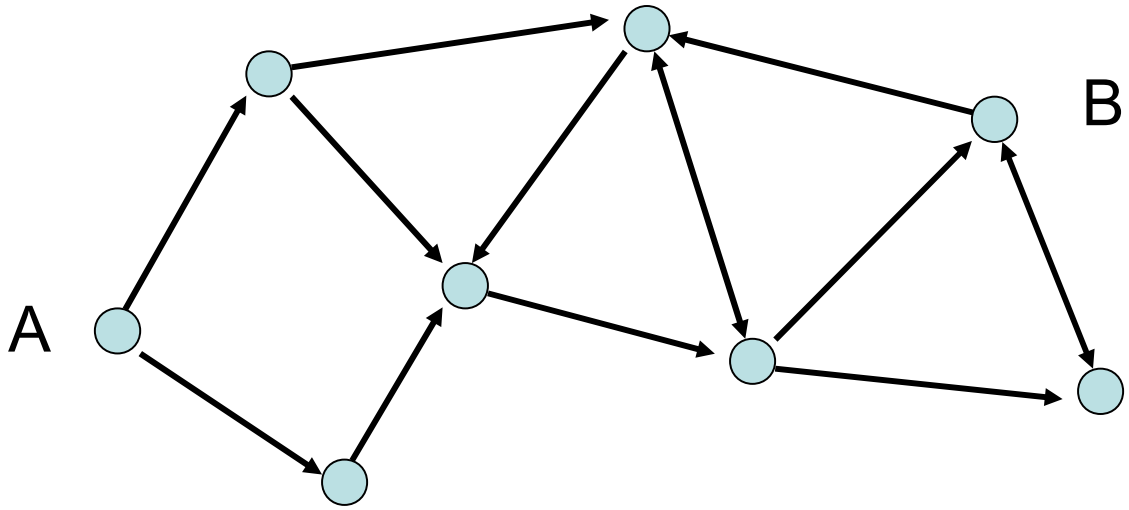
Kapitel 10

Christian Scheideler + Helmut Seidl

SS 2009

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

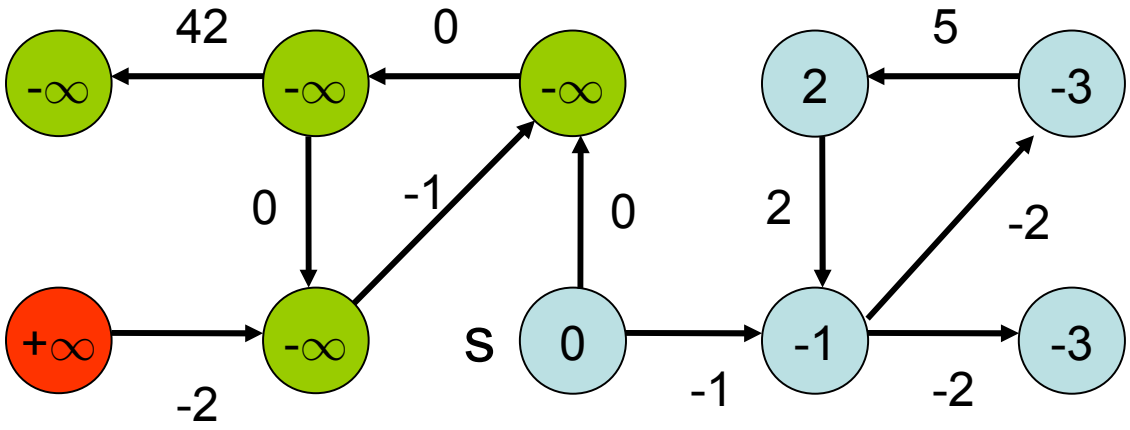
Einführung

Kürzeste-Wege-Problem:

- gerichteter Graph $G=(V,E)$
- Kantenkosten $c:E \rightarrow IR$

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

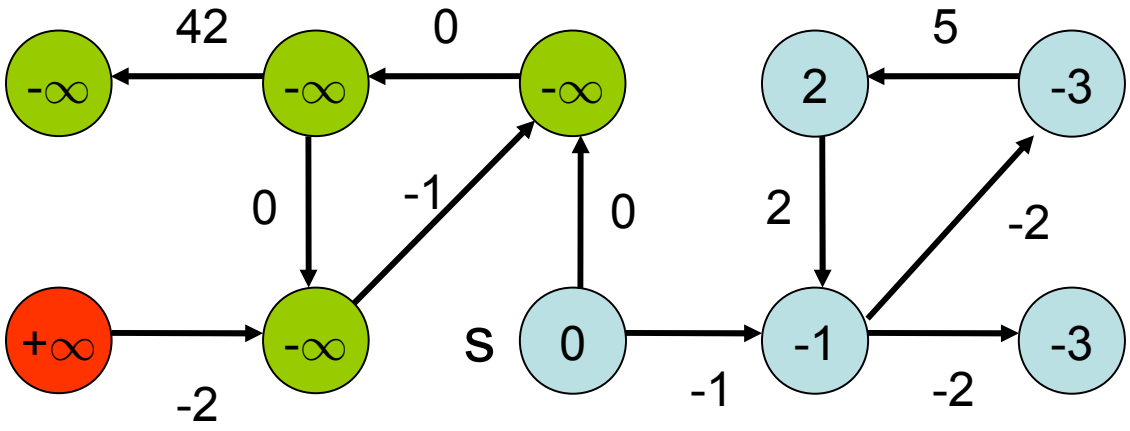
Einführung



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

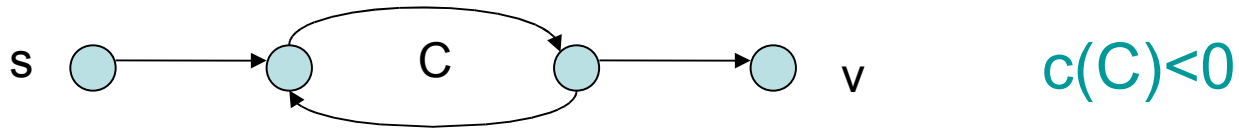
$$\mu(s,v) = \begin{cases} \infty & \text{kein Weg von s nach v} \\ -\infty & \text{Weg bel. kleiner Kosten von s nach v} \\ \min\{ c(p) \mid p \text{ ist Weg von s nach v}\} \end{cases}$$

Einführung



Wann sind die Kosten $-\infty$?

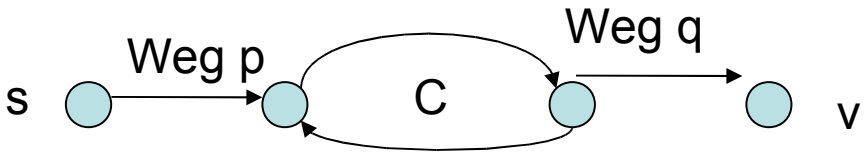
Wenn es einen negativen Kreis gibt:



Einführung

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

Negativer Kreis hinreichend:



$c(C)<0$

Kosten für i-fachen Durchlauf von C:
$c(p) + i \cdot c(C) + c(q)$
Für $i \to \infty$ 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 $p_0 C_1 p_1 ... p_{n-1} C_n p_n$,

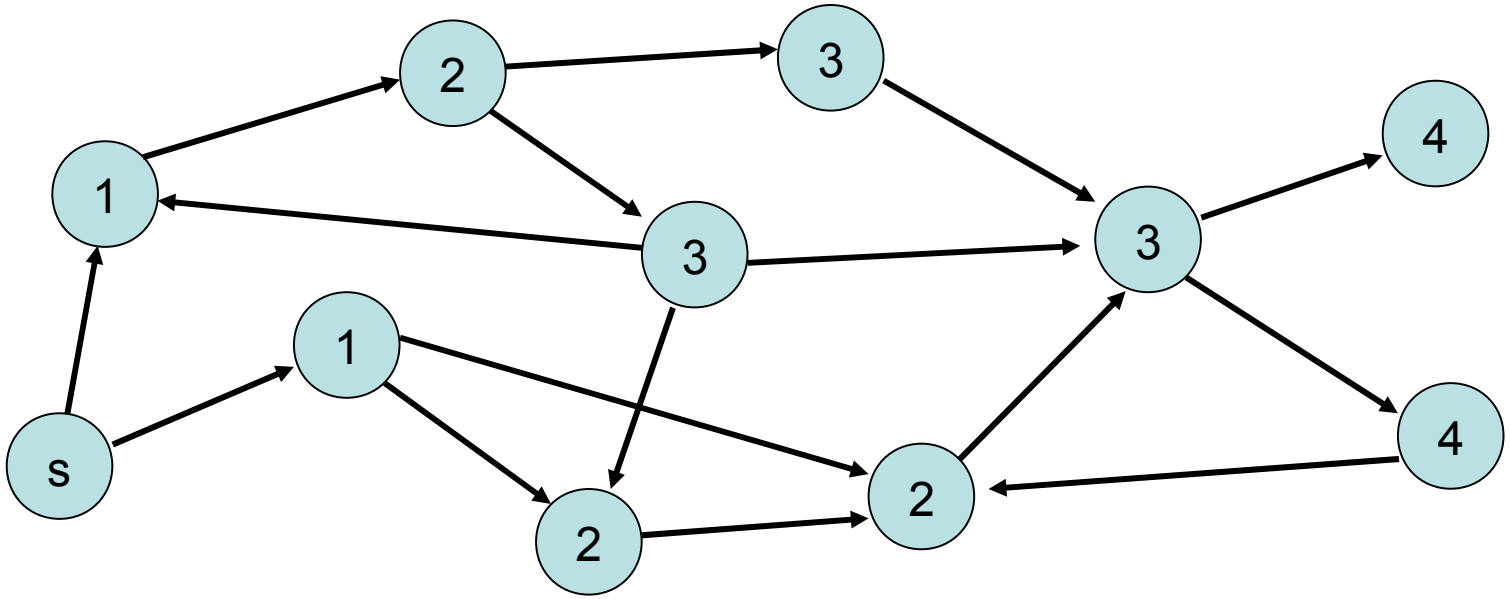
wobei C_i Kreise sind und $p_0 ... p_n$ ein einfacher Pfad

da $c(r) < l \leq c(p_0 ... p_n)$ ist, gilt $c(C_i) < 0$ für ein i

Kürzeste Wege

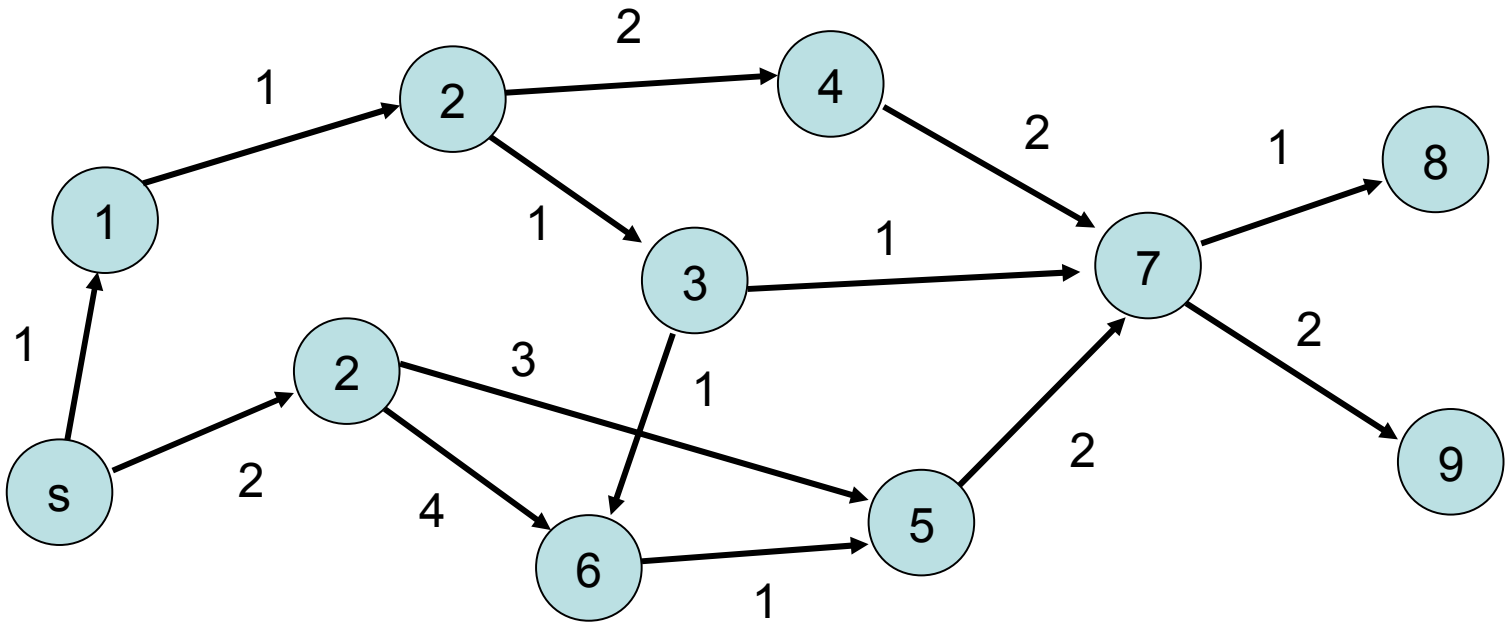
Graph mit Kantenkosten 1:

Führe Breitensuche durch.



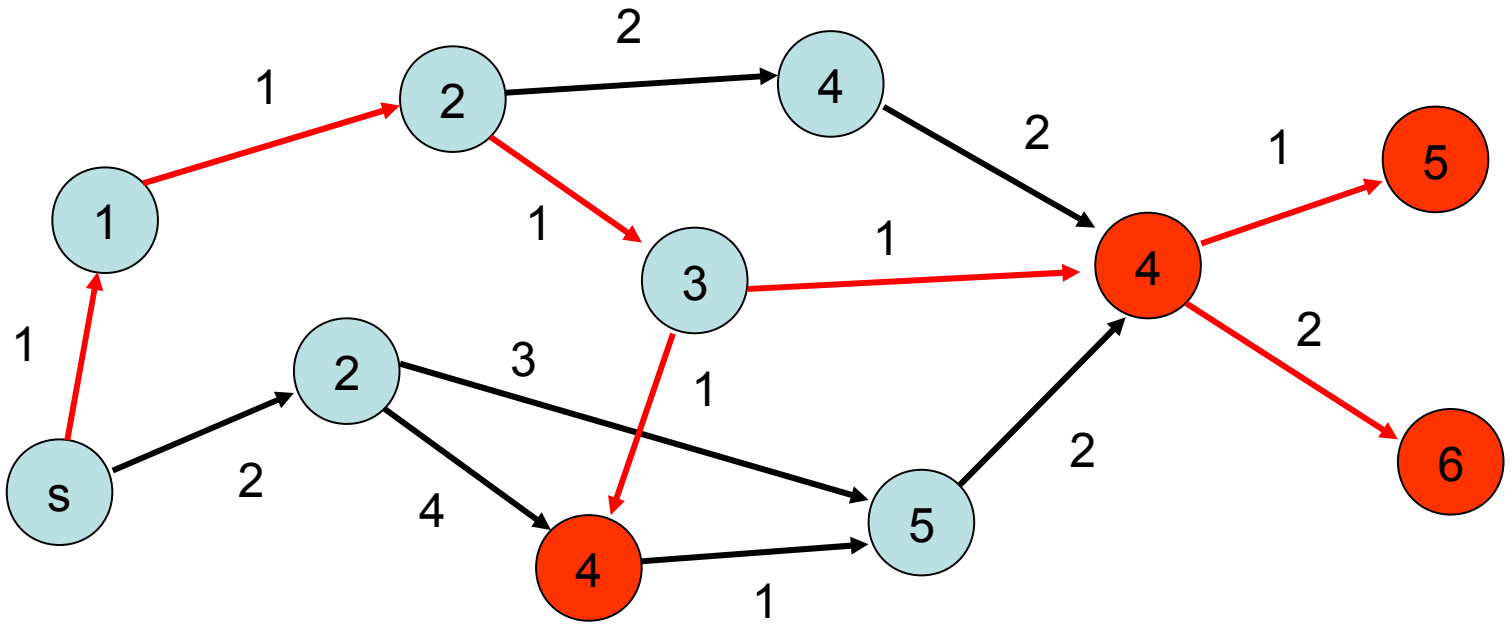
Kürzeste Wege in DAGs

Reine Breitensuche funktioniert nicht.

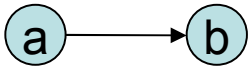


Kürzeste Wege in DAGs

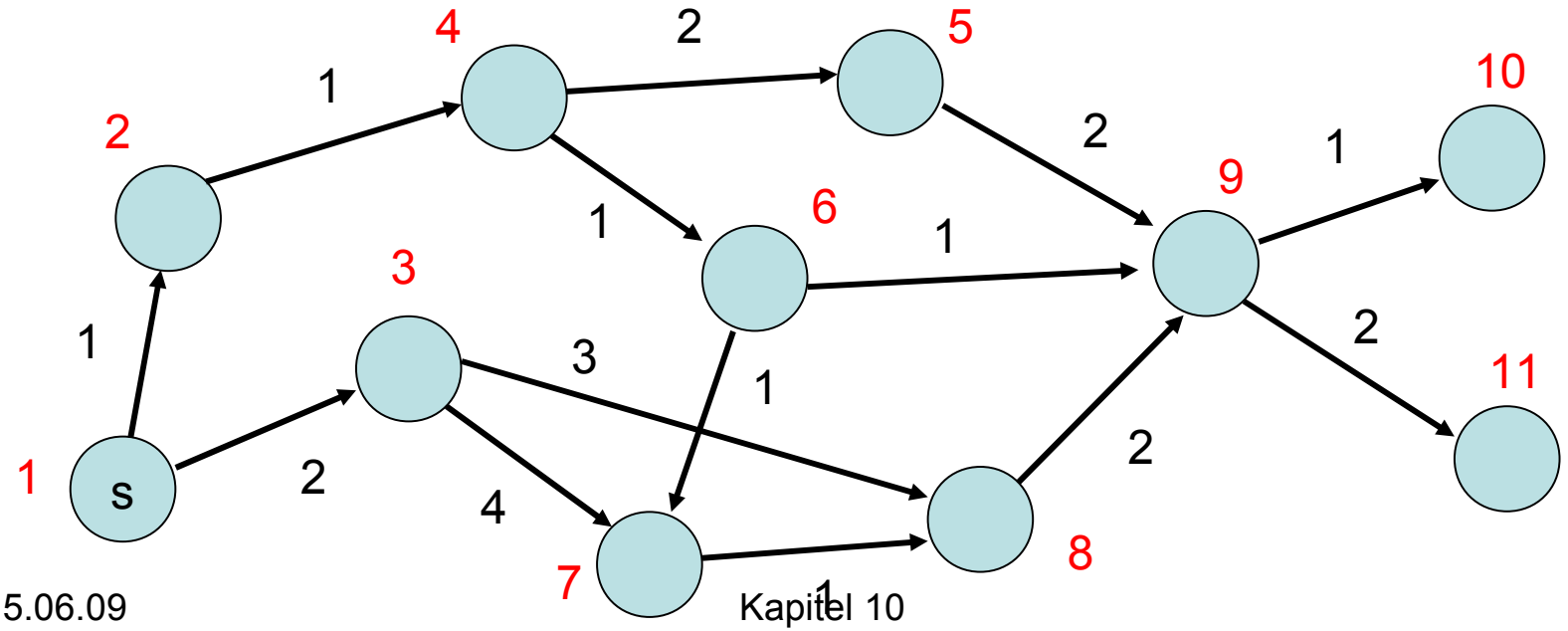
Korrekte Distanzen:



Kürzeste Wege in DAGs

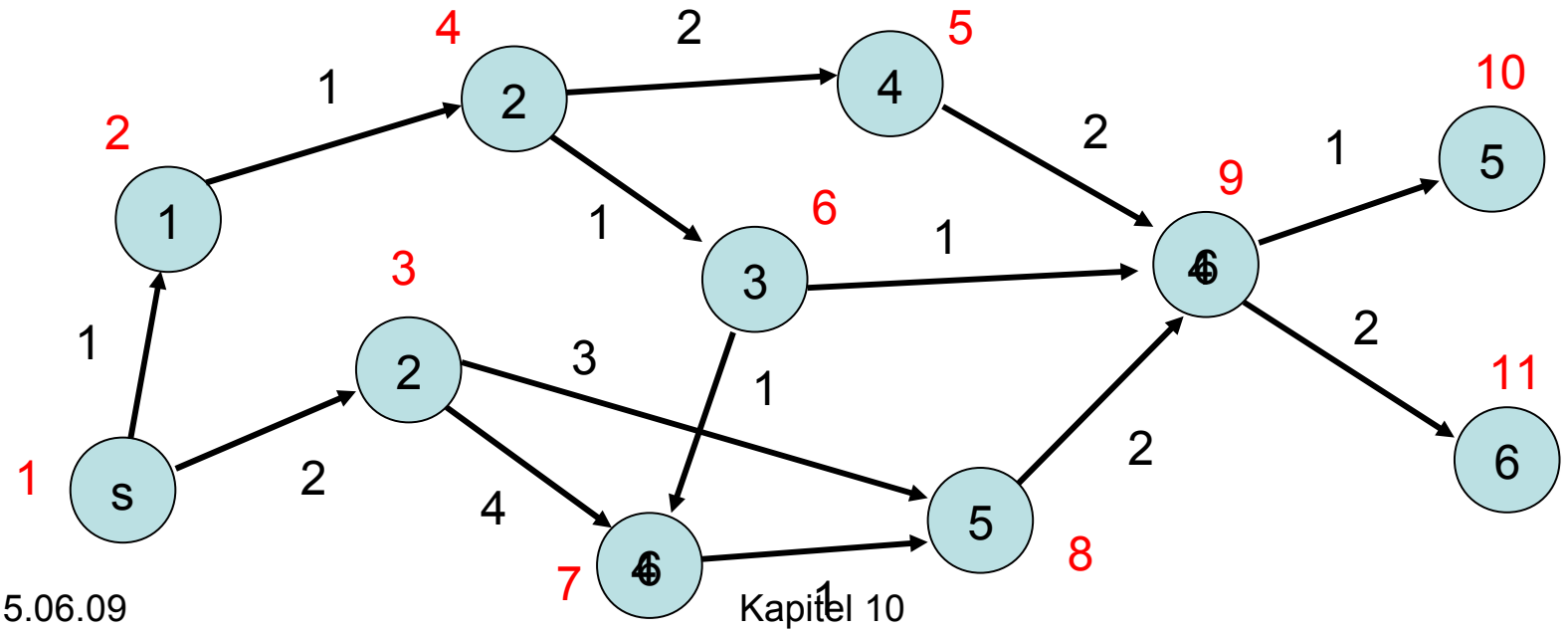
Strategie: nutze aus, dass Knoten in DAGs topologisch sortiert werden können, d.h.: alle Kanten 

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:

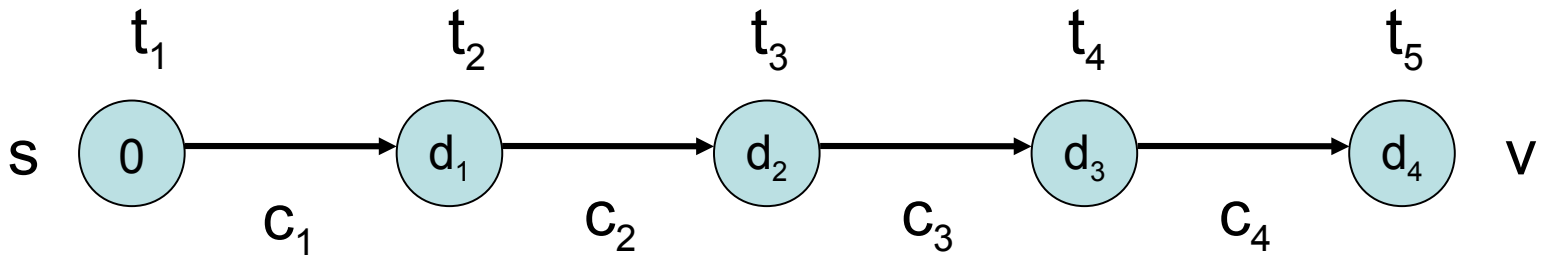
1. Topologische Sortierung der Knoten
2. Aktualisierung der Distanzen gemäß der topologischen Sortierung

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.

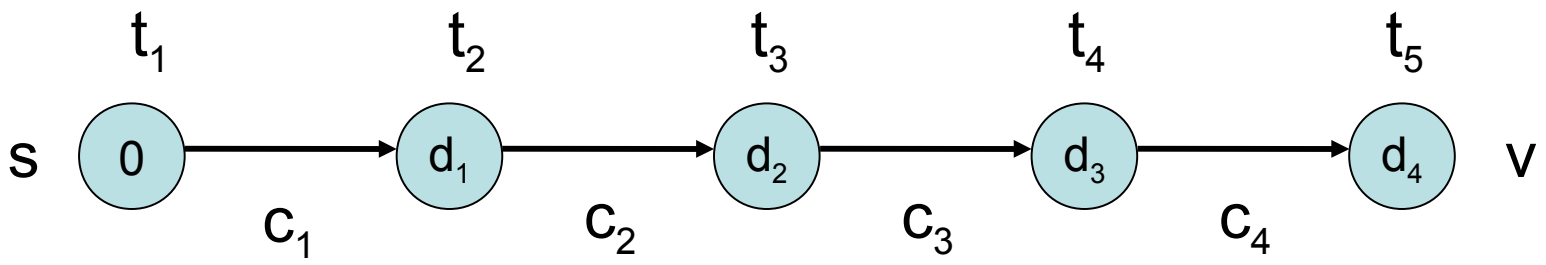


Besuch in topologischer Reihenfolge führt zu richtigen Distanzen ($d_i = \sum_{i<=i} c_i$).

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 anderen 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:

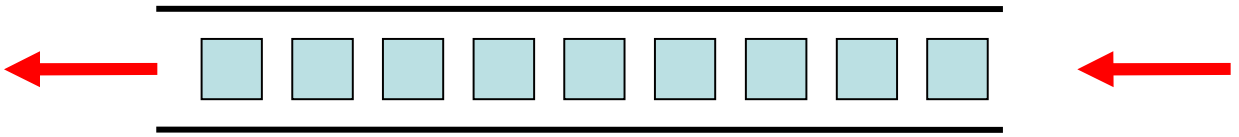
1. Topologische Sortierung der Knoten
2. Aktualisierung der Distanzen gemäß der topologischen Sortierung

Wie führe ich eine topologische Sortierung durch?

Kürzeste Wege in DAGs

Topologische Sortierung:

- Verwende eine FIFO Queue q

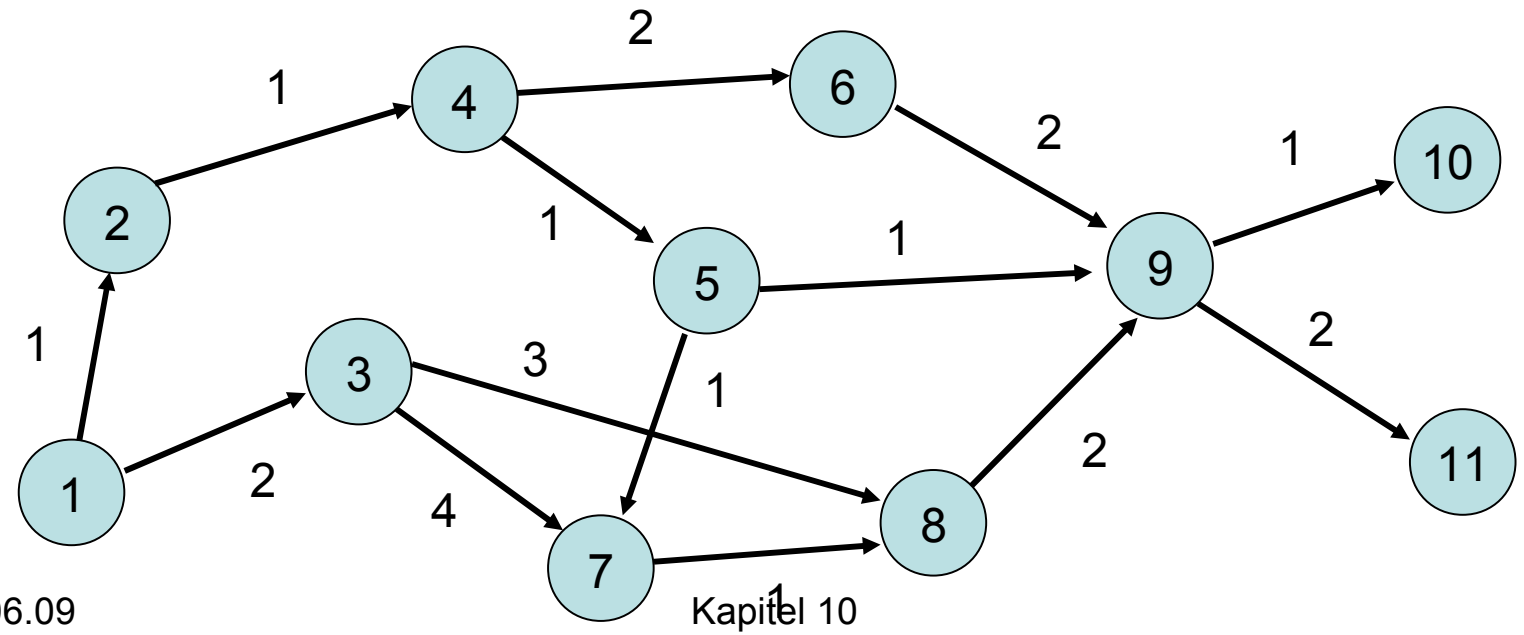


- Anfangs enthält q alle Knoten, die keine eingehende Kante haben (Quellen).
- Verwalte für jeden Knoten Zähler der noch nicht markierten eingehenden Kanten.
- Entnehme v aus q und markiere alle $(v,w) \in E$, d.h. dekrementiere den Zähler für w. Falls der Zähler von w null wird, füge w in q ein. Wiederhole das, bis q leer ist.

Kürzeste Wege in DAGs

Beispiel:

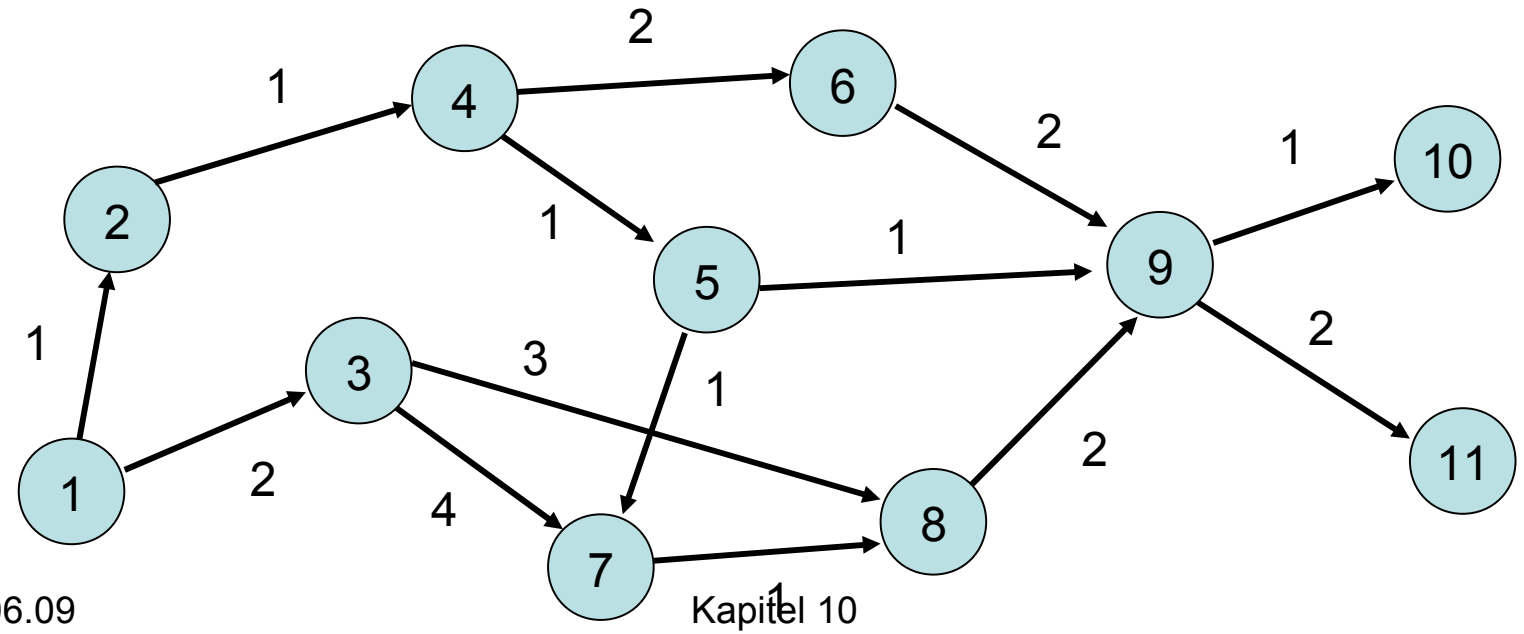
- ●: Knoten momentan in Queue q
- Nummerierung nach Einfügereihenfolge



Kürzeste Wege in DAGs

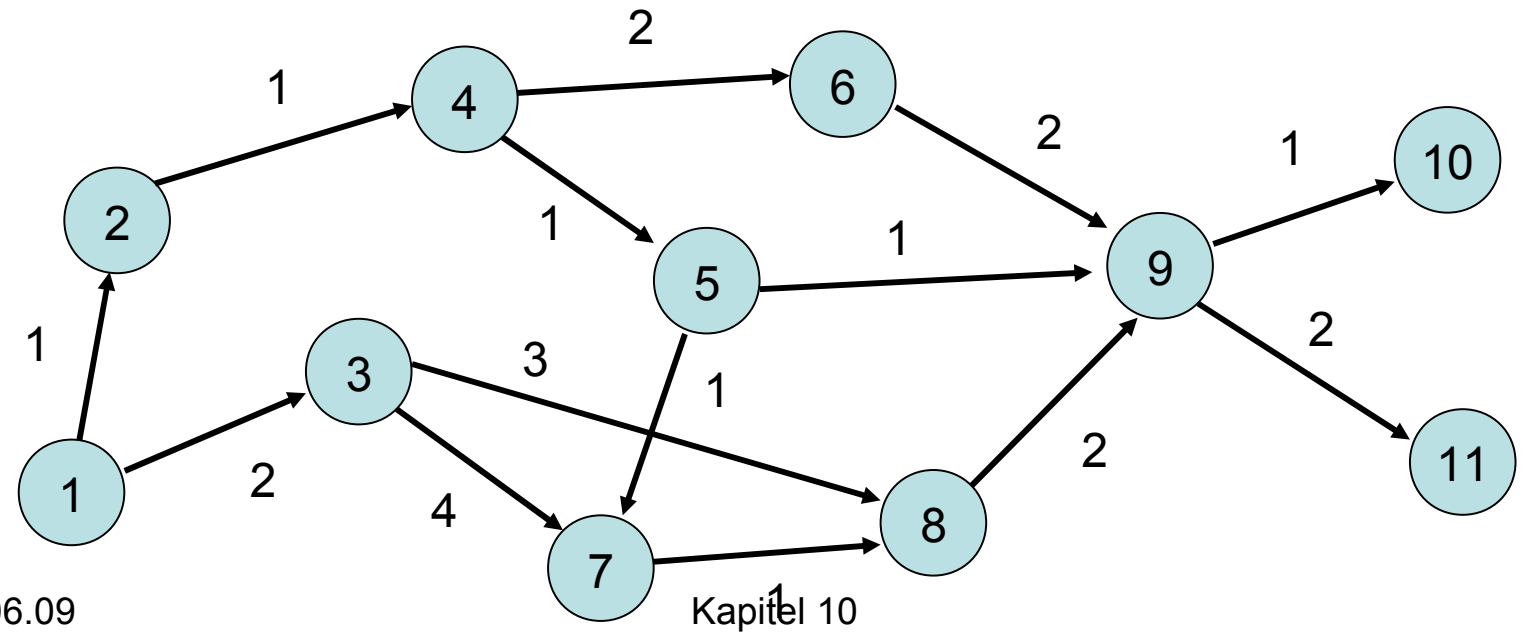
Korrektheit der topologischen Nummerierung:

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



Kürzeste Wege in DAGs

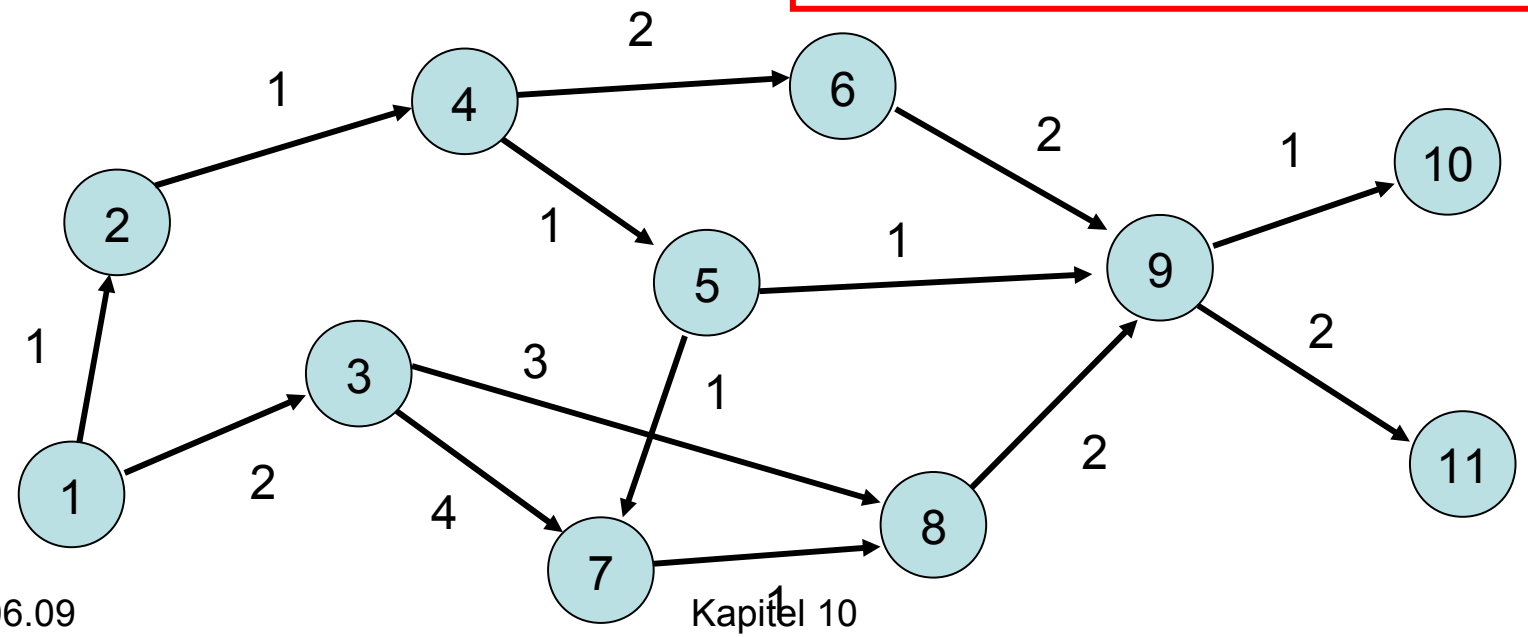
Laufzeit: Zur Bestimmung der Anzahlen der eingehenden Kanten aller Knoten muss Graph einmal durchlaufen werden. Danach wird jeder Knoten und jede Kante genau einmal betrachtet, also Zeit O(n+m).



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:

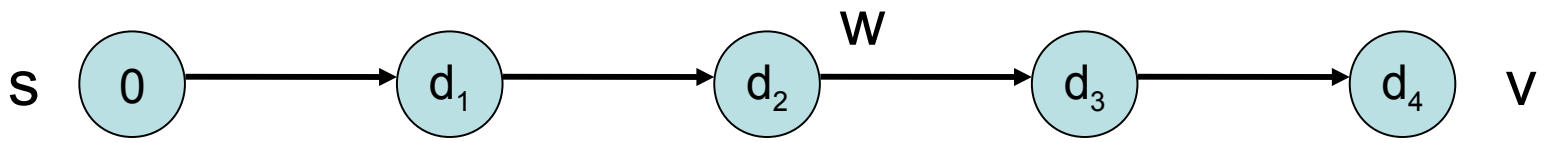
1. Topologische Sortierung der Knoten
   Laufzeit $O(n+m)$
2. Aktualisierung der Distanzen gemäß der topologischen Sortierung
   Laufzeit $O(n+m)$

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



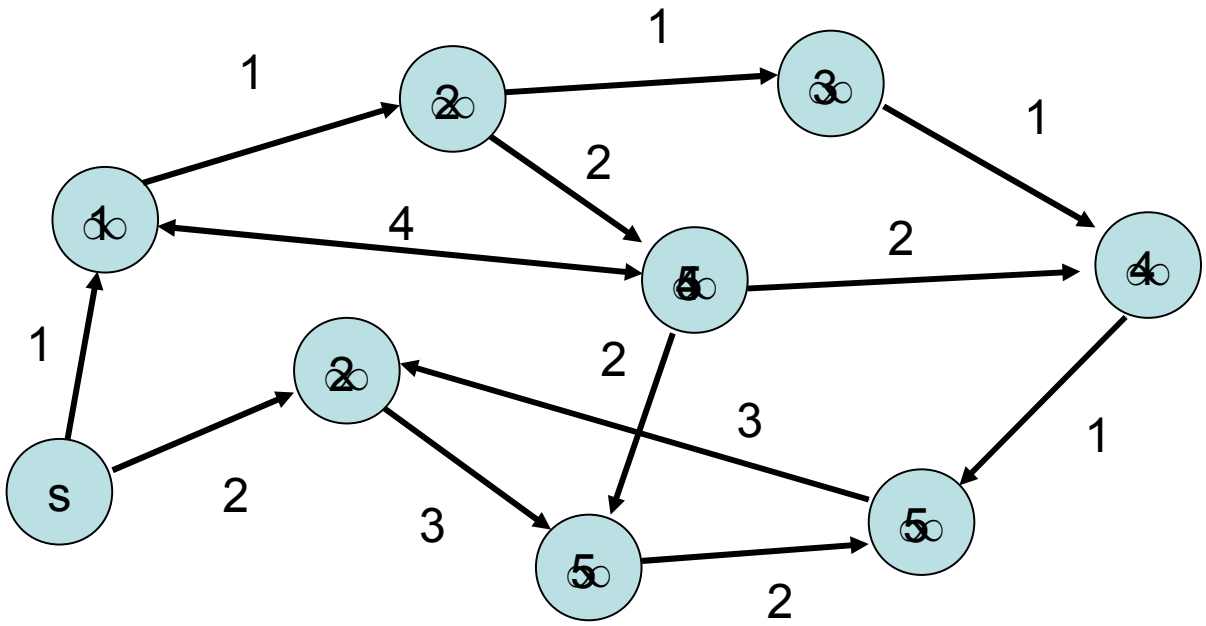
Lösung: besuche Knoten in der Reihenfolge der kürzesten Distanz zur Quelle s

Dijkstras Algorithmus

- Am Anfang, setze d[s] = 0; und d[v] = ∞; für alle andern 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

```
void Dijkstra(Node s) {
   d = new double[n]; Arrays.fill(d,∞);
   parent = new Node[n];
   d[s] = 0; parent[s] = s;
   PriorityQueue<Node> q = <s>;
   while (q != <>) {
      u = q.deleteMin();  // u: min. Distanz zu s in q
      foreach (e=(u,v) ∈ E)
         if (d[v] > d[u]+c(e)) { // aktualisiere d[v]
            if (d[v]==∞) 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 $d[v] > \mu(s,v)$ realisiert durch einen kanten-minimalen Pfad.
- Sei $p=(s=v_1,v_2,\ldots,v_k=v)$ ein solcher von s nach v. Weil alle $c(e) \geq 0$ sind, steigt minimales $d[w]$ in q monoton an. Da $d[v_{k-1}] \leq d[v_k]$, muss v_{k-1} vor v_k aus q entfernt worden sein oder gleiche Distanz haben. Entweder gilt auch $d[v_{k-1}] > \mu(s,v_{k-1})$, dann haben wir einen Widerspruch für einen kürzeren Pfad, oder $d[v_{k-1}] = \mu(s,v_{k-1})$
- 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:

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

Binärer Heap: alle Operationen $O(\log n)$, also $T_{Dijkstra} = O((m+n) \log n)$

Fibonacci Heap (nicht in Vorlesung behandelt):

- $T_{DeleteMin}(n)=T_{Insert}(n)=O(\log n)$
- $T_{decreaseKey}(n)=O(1)$

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

Monotone Priority Queues

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

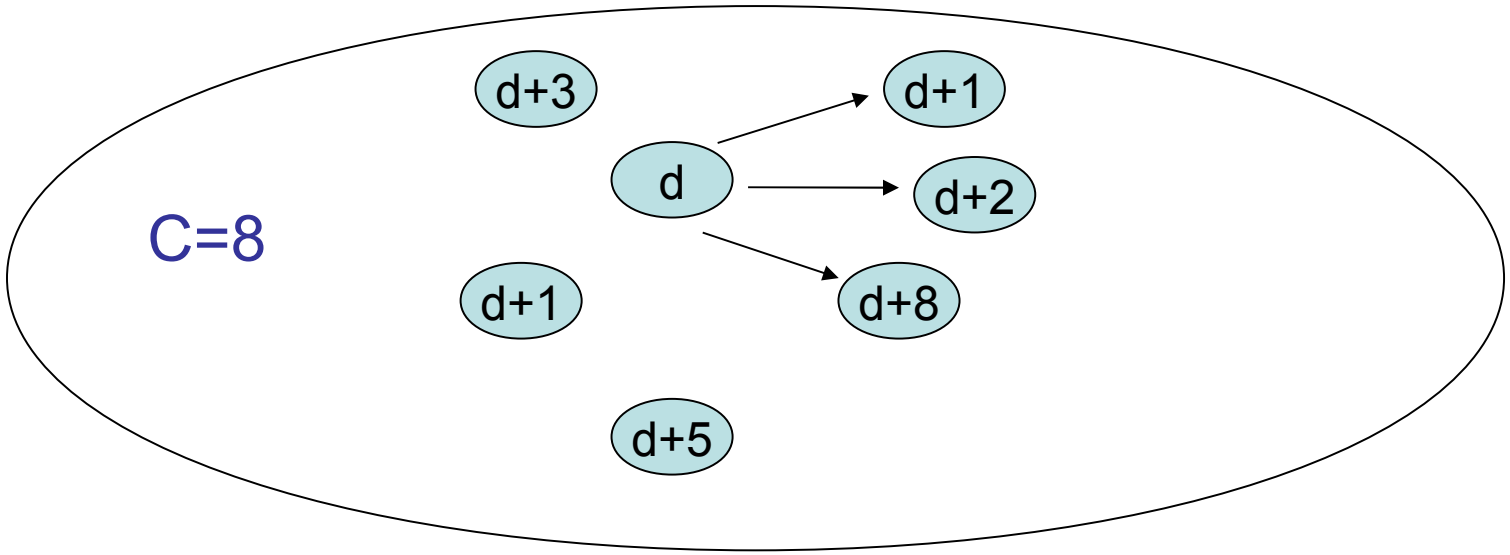
Monotone Priority Queue: Folge der gelöschten Elemente hat monoton steigende Werte.

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.

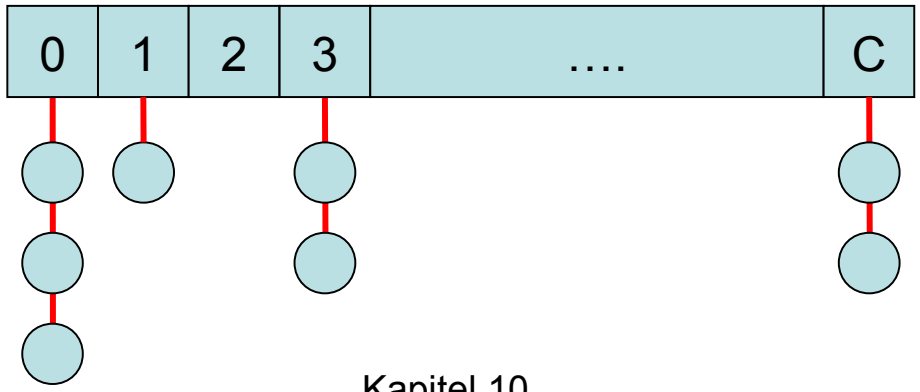


Bucket Queue

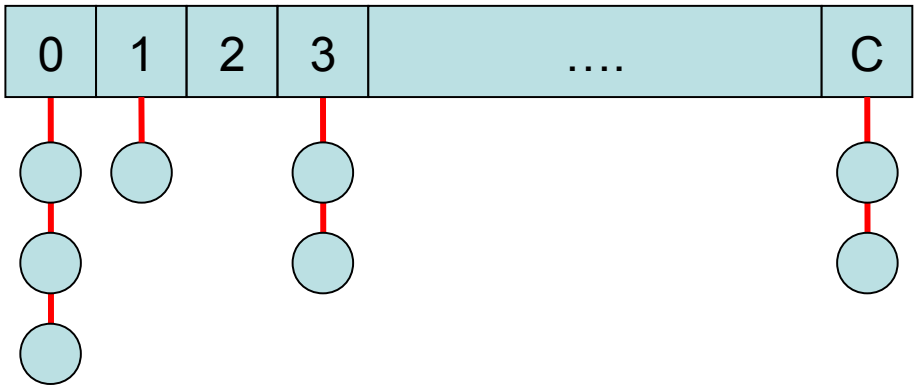
Annahme: alle Kantenkosten im Bereich [0,C].

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 (mod C+1)

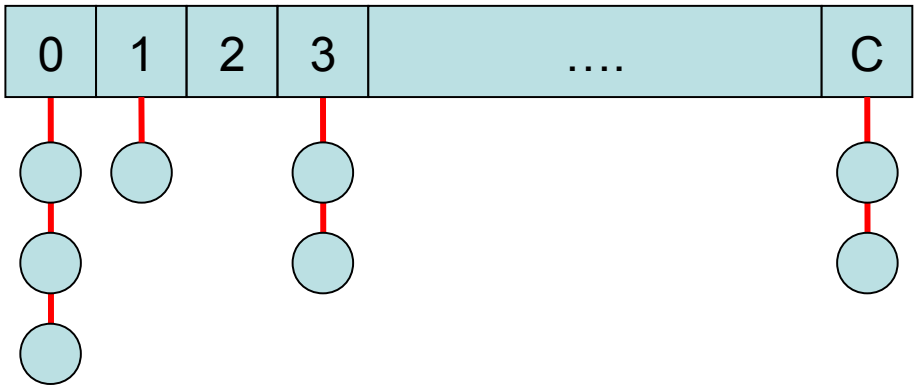


Bucket Queue



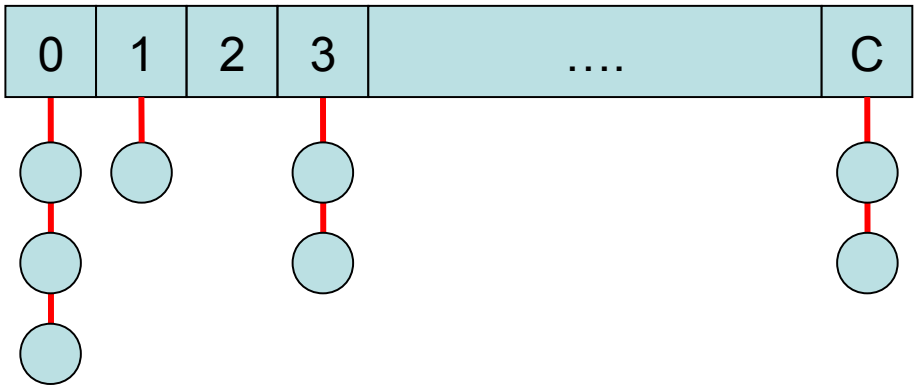
- Jeder Knoten v mit momentaner Distanz d[v] wird in Liste B[d[v] mod (C+1)] gespeichert
- Da momentane d[v]´s im Bereich [d,d+C] für ein d sind, haben alle v in einem B[d] dasselbe d[v].

Bucket Queue



- Insert(v): fügt v in B[d[v] mod (C+1)] ein. Laufzeit O(1).
- decreaseKey(v): entfernt v aus momentaner Liste und fügt v in B[d[v] mod (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



- Insert(v), decreaseKey(v): Laufzeit O(1).
- deleteMin(): Laufzeit O(C).
- Laufzeit von Dijkstras Algo mit Bucket Queue: O(m+n·C)

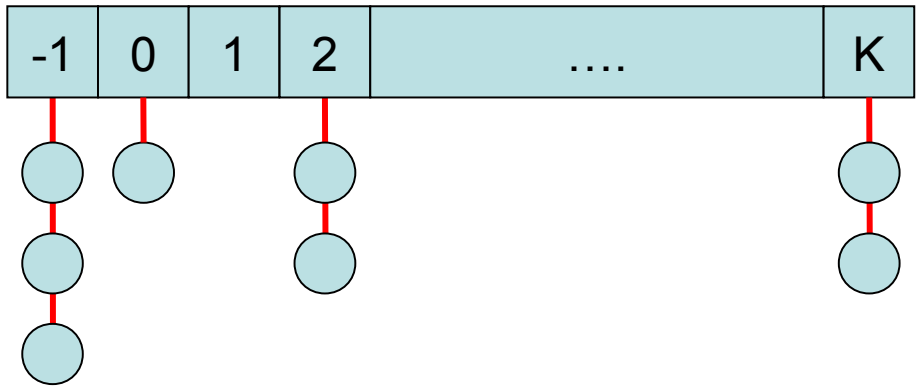
Radix Heap

Radix Heaps sind eine verbesserte Form der Bucket Queues.

Laufzeit von Dijkstras Algo mit Radix Heap:
$O(m + n \log C)$

Radix Heap

Array B mit Listen B[-1] bis B[K], $K=1+\lfloor \log C \rfloor$.



Regel: Knoten v in $B[\min(msd(d_{min},d[v]),K)]$

$msd(d_{min},d[v])$: höchstes Bit, in dem sich Binärdarstellungen von d_{min} und d[v] unterscheiden (-1: kein Unterschied)

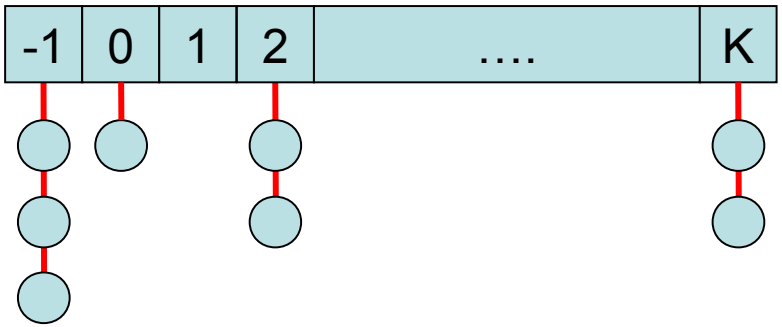
Radix Heap

Beispiel für $msd(d_{min},d)$:

- sei $d_{min}=17$ oder binär 10001
- $d=17$: $msd(d_{min},d)=-1$
- $d=18$: binär 10010, also $msd(d_{min},d)=1$
- $d=21$: binär 10101, also $msd(d_{min},d)=2$
- $d=52$: binär 110100, also $msd(d_{min},d)=5$

Radix Heap

- Berechnung von msd für a=b:
  $$msd(a,b)=\lfloor \log(a \oplus b) \rfloor$$
  Zeit O(1) (mit entspr. Maschinenbefehlen)



- insert, decreaseKey: wie vorher (mit neuer Regel), Laufzeit O(1)

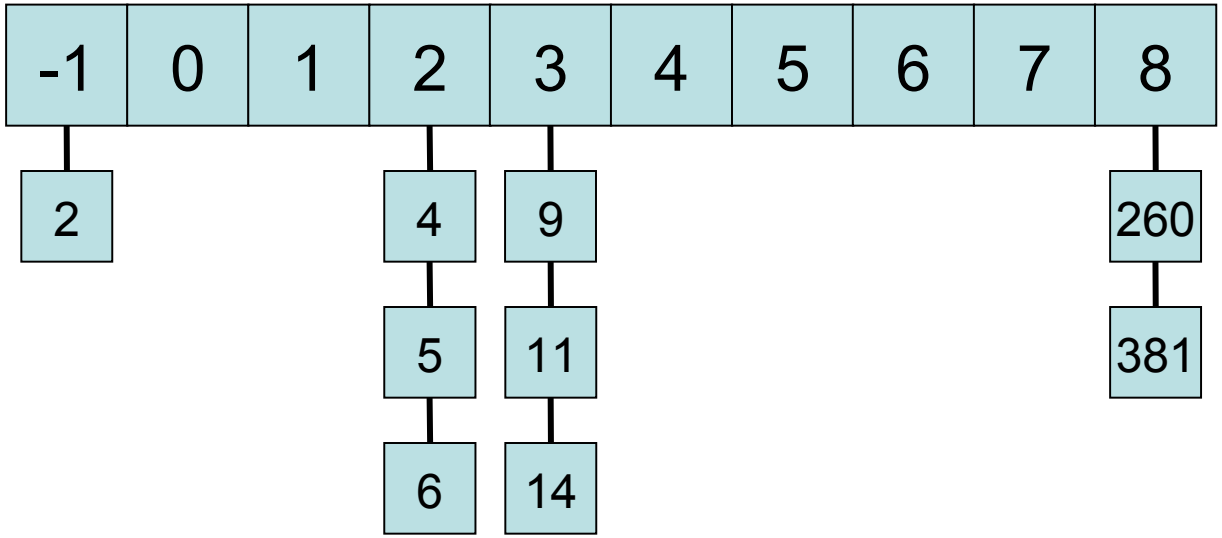
Radix Heap

deleteMin Operation:

- Falls B[-1] besetzt, entnehme ein v aus B[-1]
  (sonst kein Element mehr in Heap und fertig)
- finde minimales i, so dass B[i] ≠ ∅(falls kein solches i, dann fertig)
- bestimme d_{min} in B[i] (in O(1) Zeit möglich, falls z.B. d_{min}-Werte in extra Feld min[-1..K] gehalten)
- verteile Knoten in B[i] über B[-1],…,B[i-1] gemäß neuem d_{min}

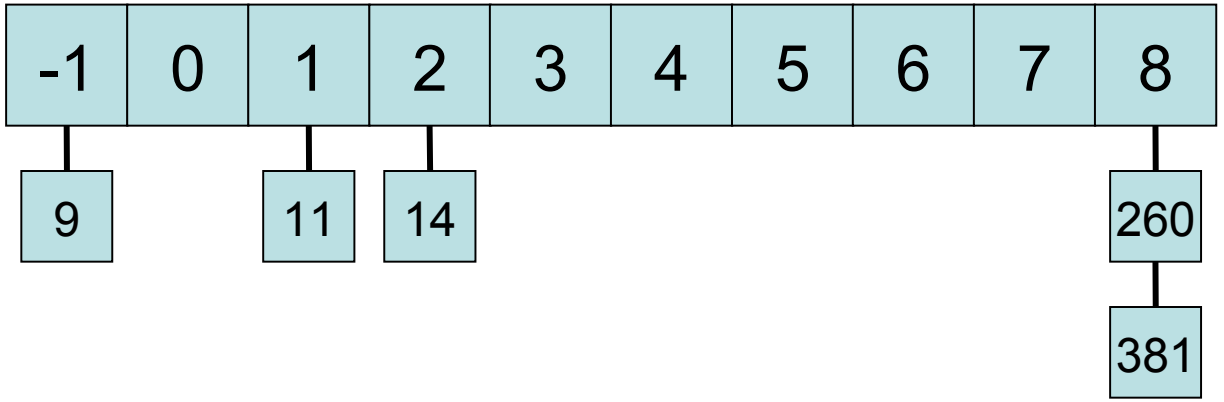
Entscheidend: für alle v in B[j], j>i, gilt nach wie vor $msd(d_{min},d[v])==j$, d.h. sie müssen nicht verschoben werden.

Radix Heap



Wir betrachten Sequenz von deleteMin Operationen

Radix Heap



Wir betrachten Sequenz von deleteMin Operationen

Radix Heap

Lemma 10.6: Sei B[i] die minimale nichtleere Liste, i>0. Sei d_{min} das kleinste Element in B[i]. Dann ist $msd(d_{min},x)<i$ für alle $x \in B[i]$.

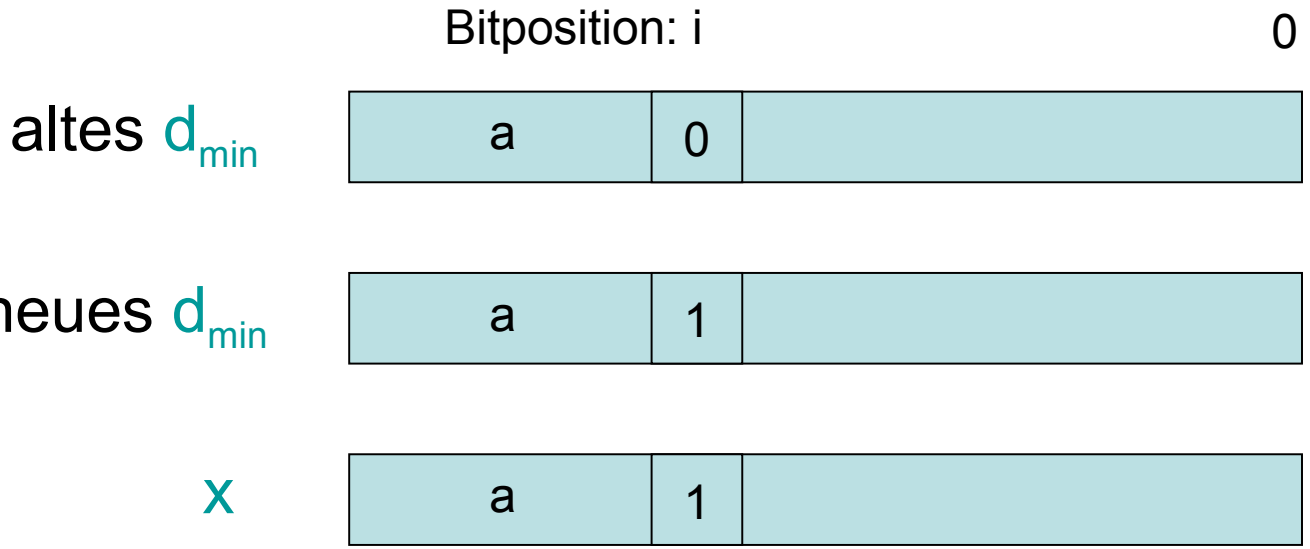
Alle Elemente in B[i] nach links

Beweis:

- $x \neq d_{min}$: klar
- $x = d_{min}$: wir unterscheiden zwei Fälle:
  1) $i<K$, 2) $i=K$

Radix Heap

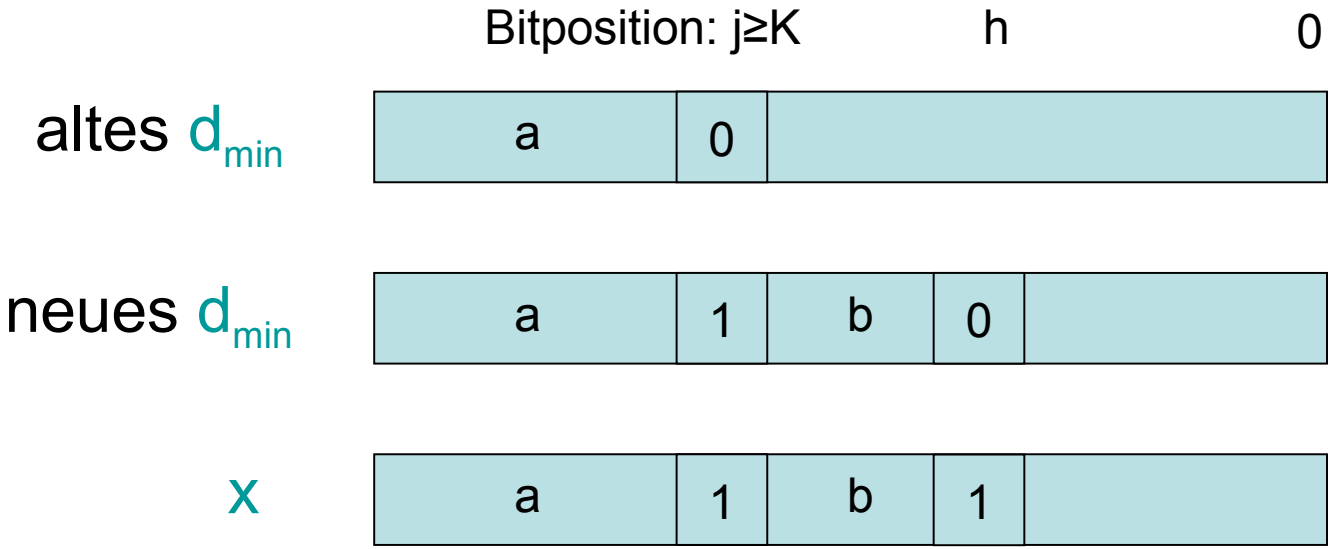
Fall i<K:



Also muss $msd(d_{min},x)<i$ sein.

Radix Heap

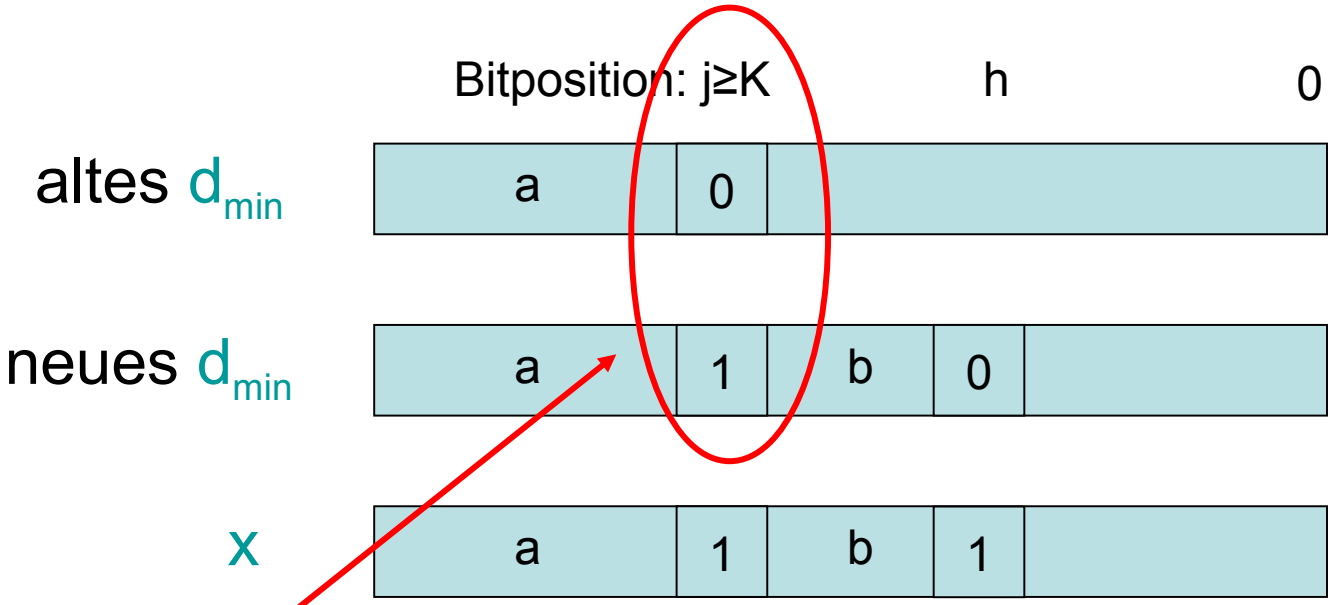
Fall i=K:



- Für alle x: $d_{min}(alt) < d_{min}(neu) < x < d_{min}(alt)+C$
- $j = msd(d_{min}(alt),d_{min}(neu))$, $h = msd(\overline{d}_{min}(\overline{neu}),x)$

Radix Heap

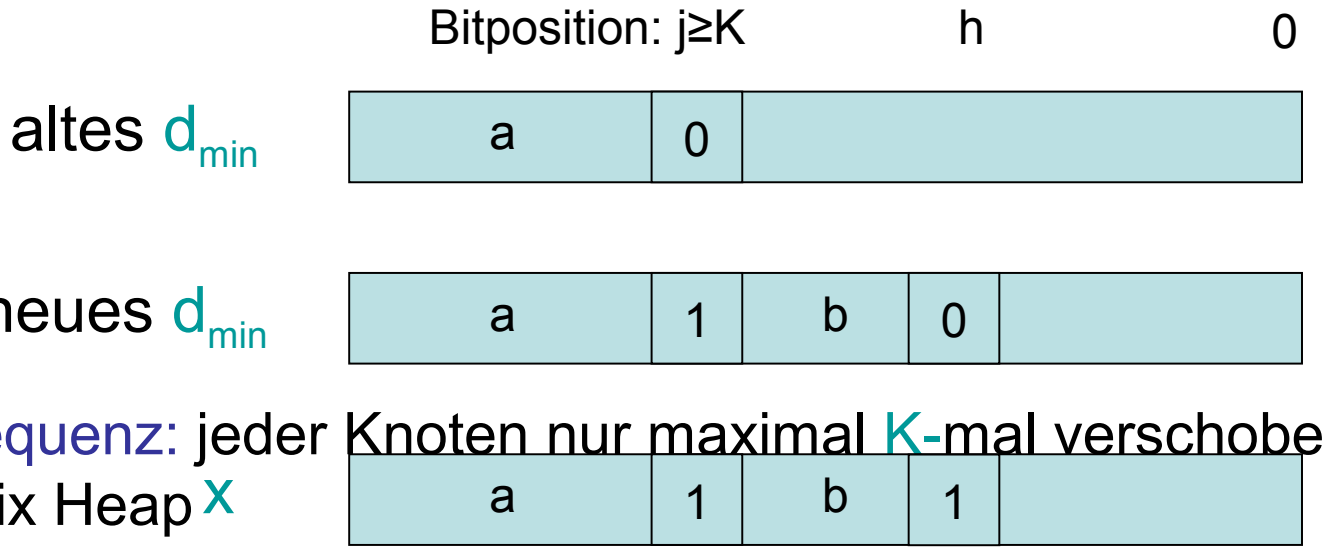
Fall i=K:



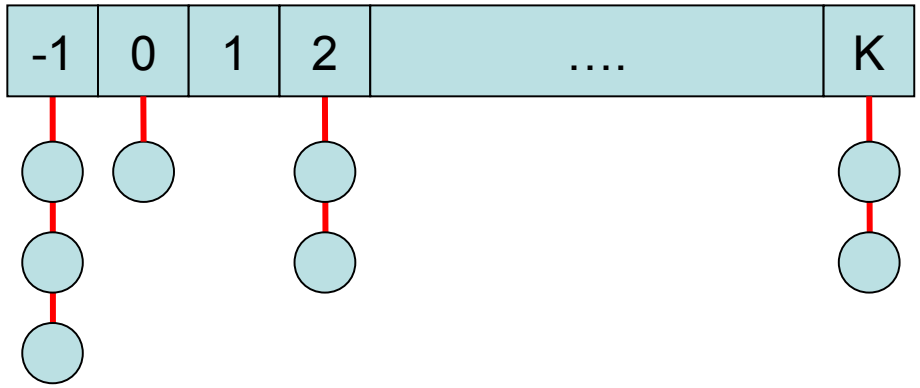
- Wegen j≥k hat Binärdarstellung von jedem übrig-bleibenden x dasselbe Präfix a1b bis runter zu Bitstelle j‘<K, d.h. h≤K

Radix Heap

Fall i=K:



Radix Heap

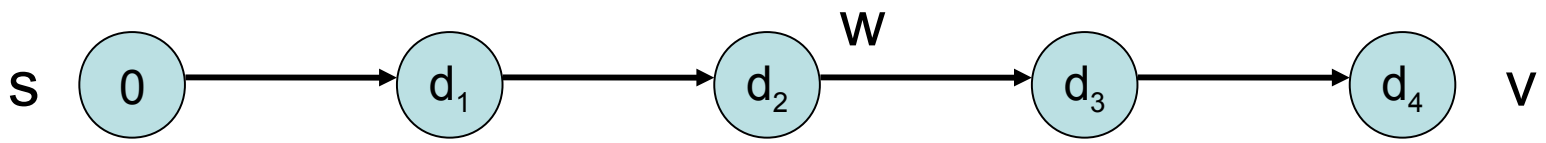


- Insert(v), decreaseKey(v): Laufzeit O(1).
- deleteMin(): amortisierte Laufzeit O(log C).
- Laufzeit von Dijkstras Algo mit Radix Heap: O(m + n log C)

Bellman-Ford Algorithmus

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

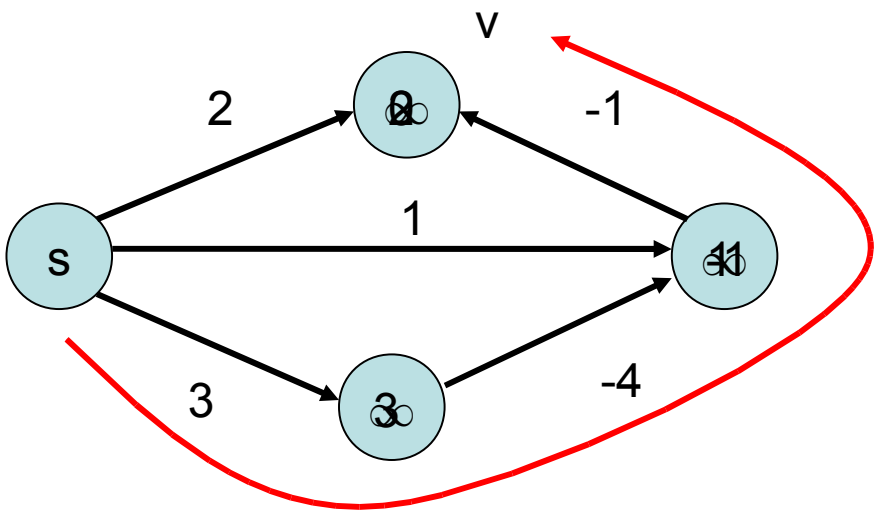
Problem: besuche Knoten eines kürzesten Weges in richtiger Reihenfolge



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

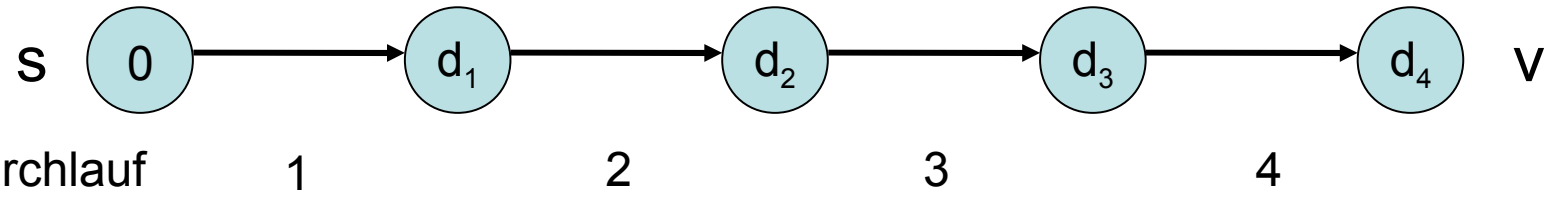
Beweis:

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

Bellman-Ford Algorithmus

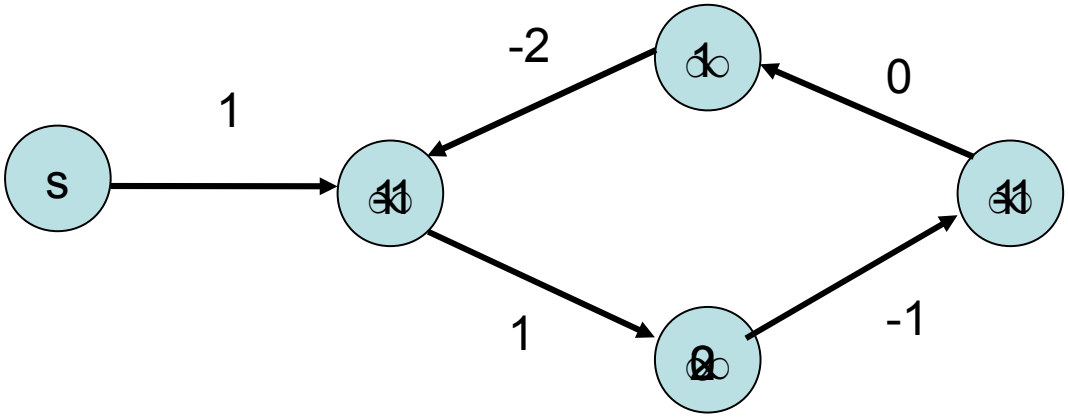
Folgerung: (Graph mit n Knoten)
Für jeden erreichbaren Knoten v mit $\mu(s,v) > -\infty$ gibt es kürzesten Weg der Länge $<n$ zu v.

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



Bellman-Ford Algorithmus

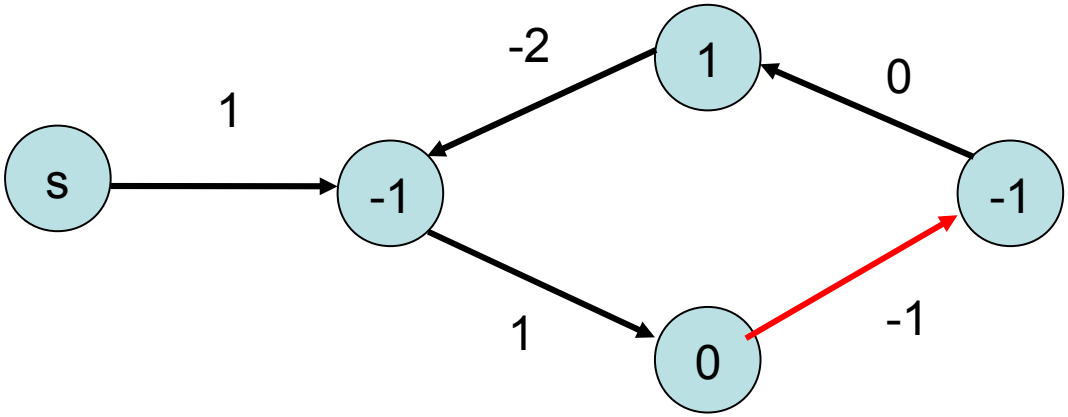
Problem: Erkennung negativer Kreise



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

Bellman-Ford Algorithmus

Problem: Erkennung negativer Kreise



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

Bellman-Ford Algorithmus

Keine Distanzerniedrigung 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 Distanzerniedrigung mehr möglich ($d[v] + c(v,w) \geq d[w]$ für alle w):
  Fertig, $d[w] == \mu(s,w)$ für alle w

- Distanzerniedrigung 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

```
void BellmanFord(Node s) {
    d = new double[n]; Arrays.fill(d,∞);
    parent = new Node [n];
    d[s] = 0; parent[s] = s;
    for(int i=0;i< n-1;i++) {  // aktualisiere Kosten für n-1 Runden
        foreach (e=(v,w) ∈ E)
            if (d[w] > d[v]+c(e)) {  // bessere Distanz möglich?
                d[w] =d[v]+c(e); parent[w] = v; }}
    foreach (e=(v,w) ∈ E)          // in n-ter Runde noch besser?
        if (d[w] > d[v]+c(e)) infect(w);
}
void infect(Node v) {  // setze -∞-Kosten von v aus
    if (d[v] > -∞) {
        d[v] = -∞;
        foreach ((v,w) ∈ E) 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

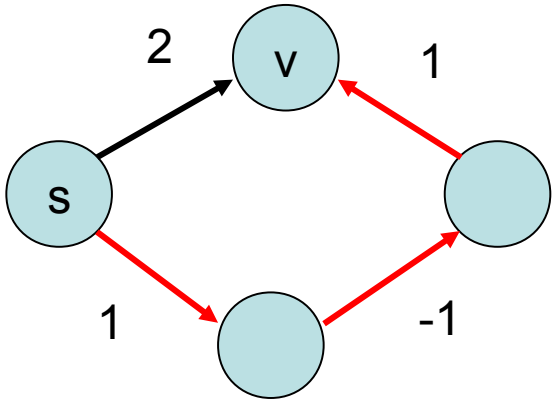
Problem: wir brauchen dazu nichtnegative Kantenkosten

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

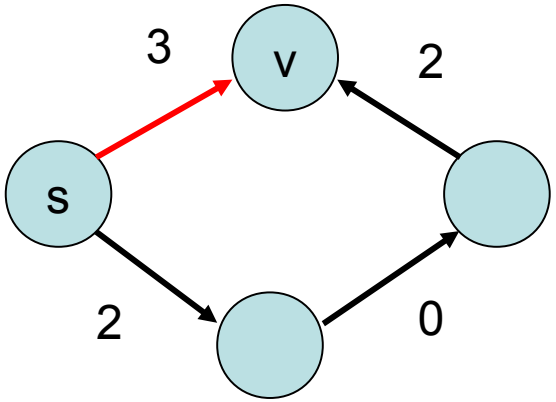
All Pairs Shortest Paths

Gegenbeispiel zur Erhöhung um Wert c:

Vorher



Kosten +1 überall



: 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 10.8: 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 10.8: 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 $p=(v_1,\ldots,v_k)$ ein beliebiger Weg und $e_i=(v_i,v_{i+1})$ für alle i. Es gilt:

$$r(p) = \sum_i r(e_i)$$
$$= \sum_i (\Phi(v_i) + c(e) - \Phi(v_{i+1}))$$
$$= \Phi(v_1) + c(p) - \Phi(v_k)$$

All Pairs Shortest Paths

Lemma 10.9: 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!)

   Berechne μ (s,v) nach Bellman-Ford und

   setze $\Phi(v) = \mu$ (s,v); für alle v

Berechne die reduzierten Kosten r(e)

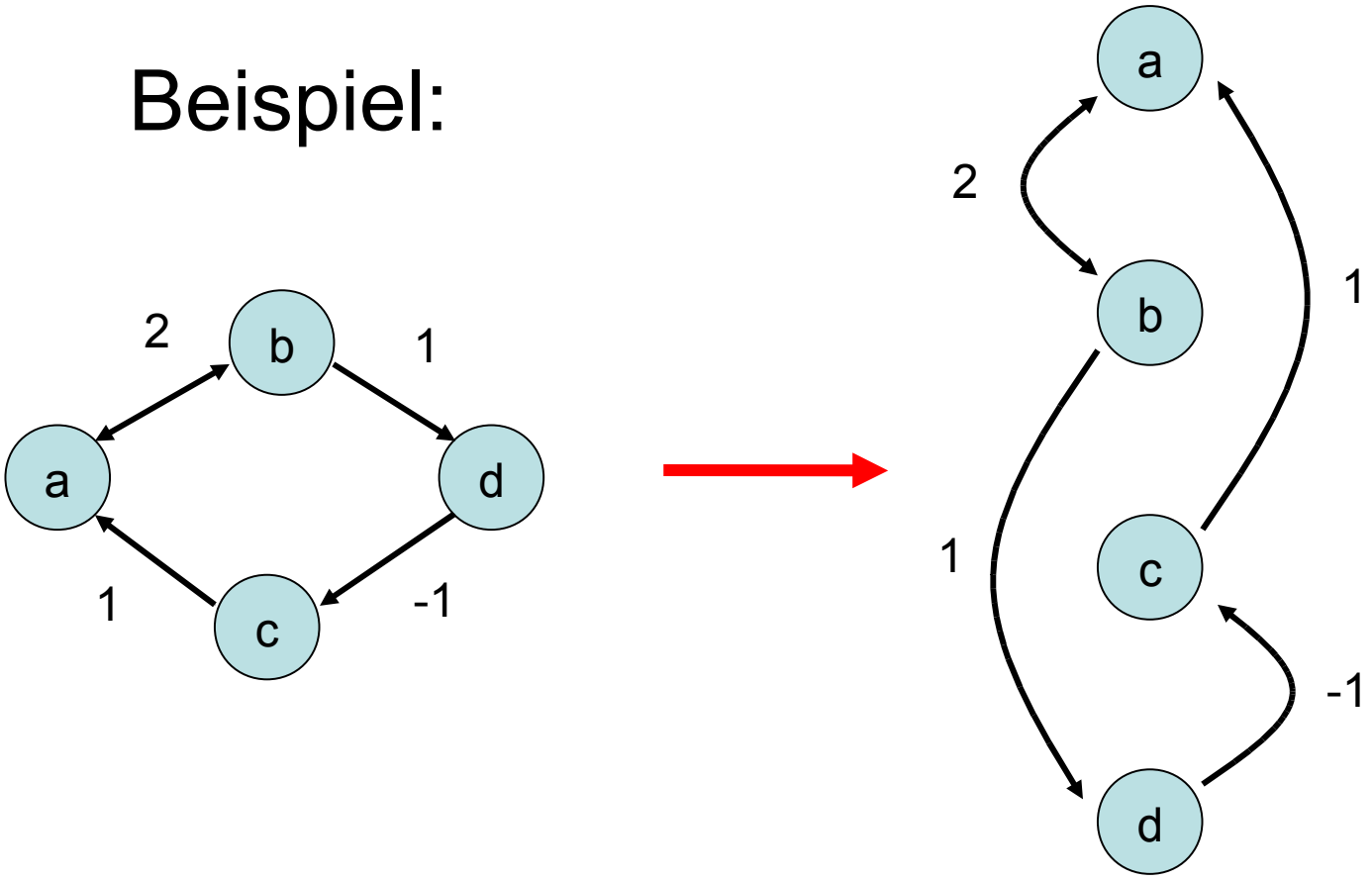
1. Berechne für alle Knoten v die Distanzen $\bar{\mu}$ (v,w) mittels Dijkstra Algo mit reduzierten Kosten auf Graph ohne Knoten s

   Berechne korrekte Distanzen μ (v,w) durch

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

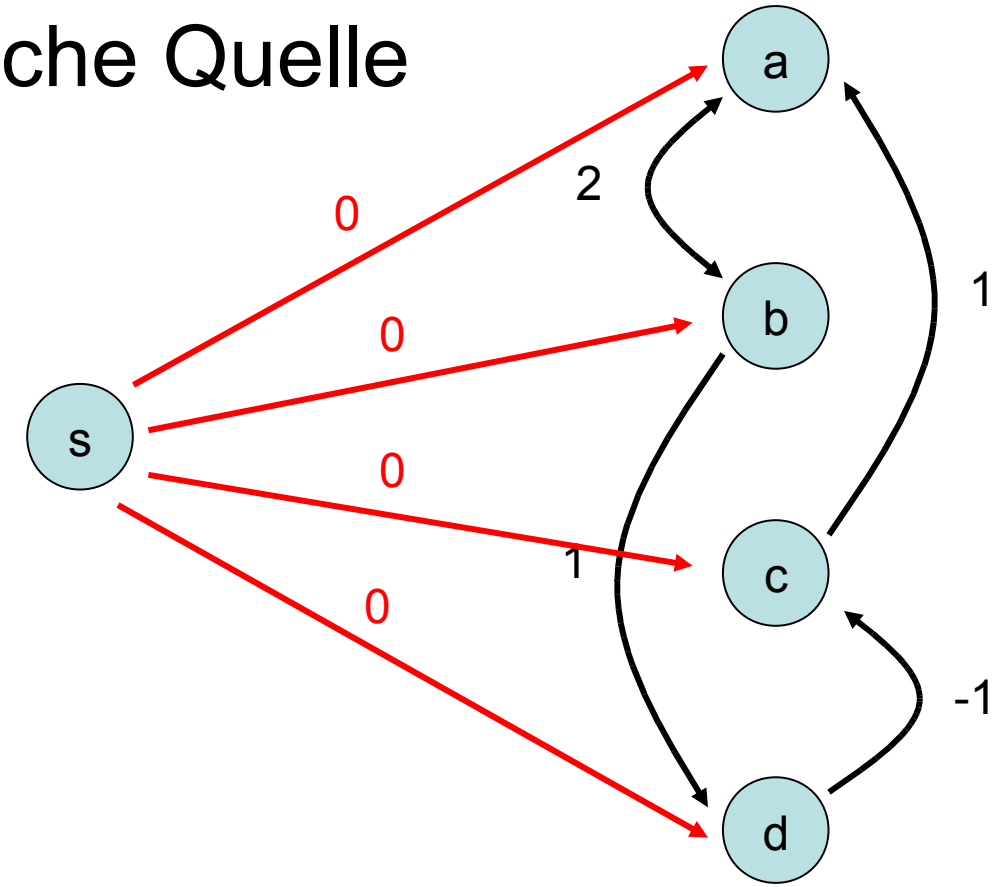
All Pairs Shortest Paths

Beispiel:



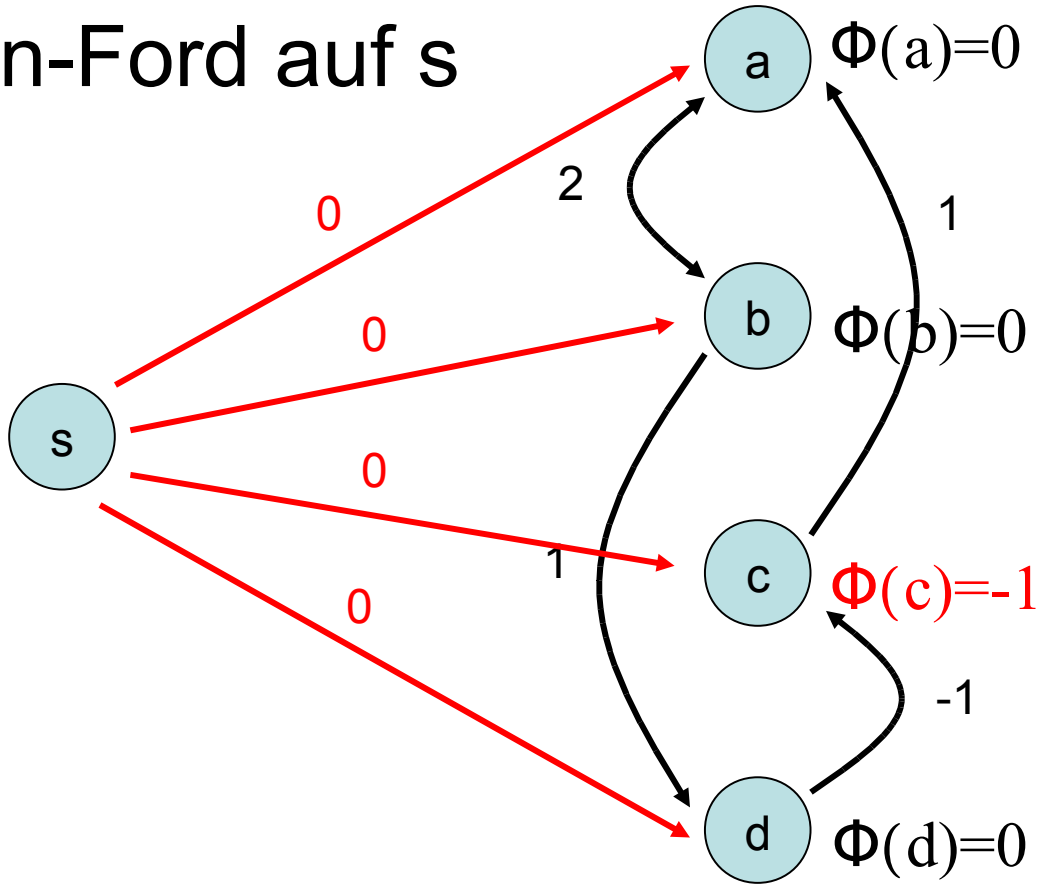
All Pairs Shortest Paths

Schritt 1: künstliche Quelle



All Pairs Shortest Paths

Schritt 2: Bellman-Ford auf s

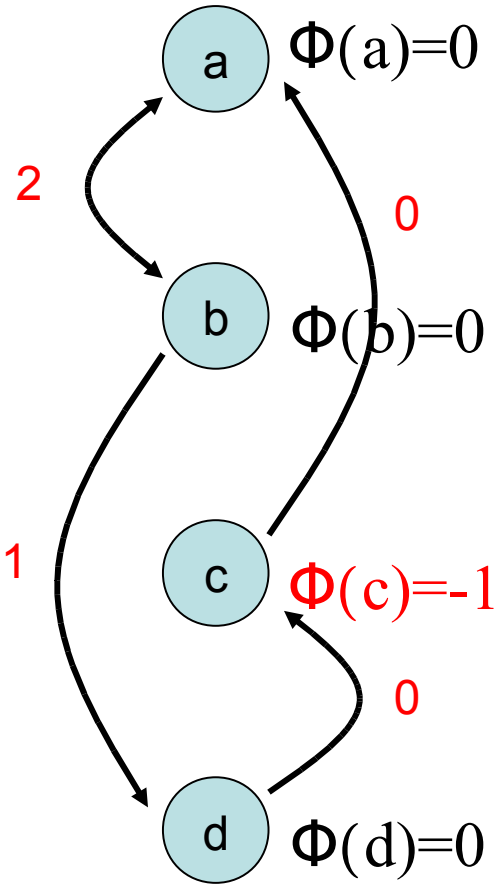


All Pairs Shortest Paths

Schritt 3: r(e)-Werte berechnen

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

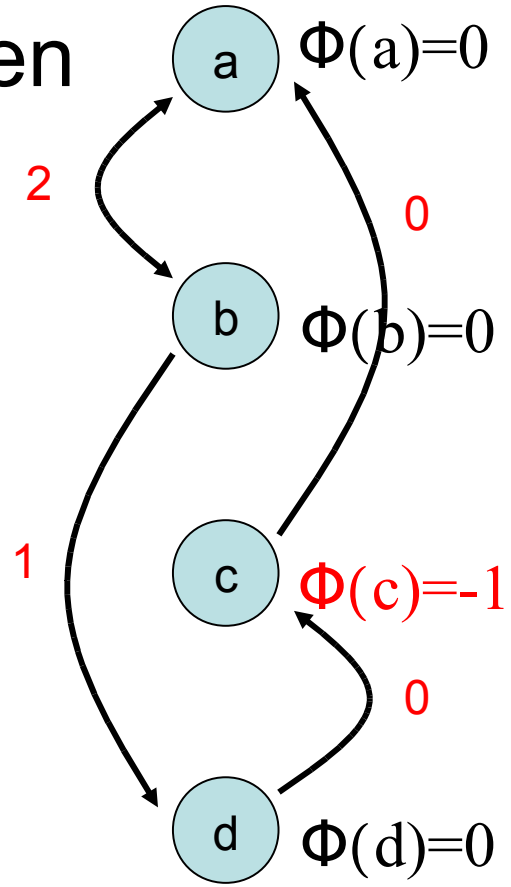
$$r(e) := \Phi(v) + c(e) - \Phi(w)$$



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

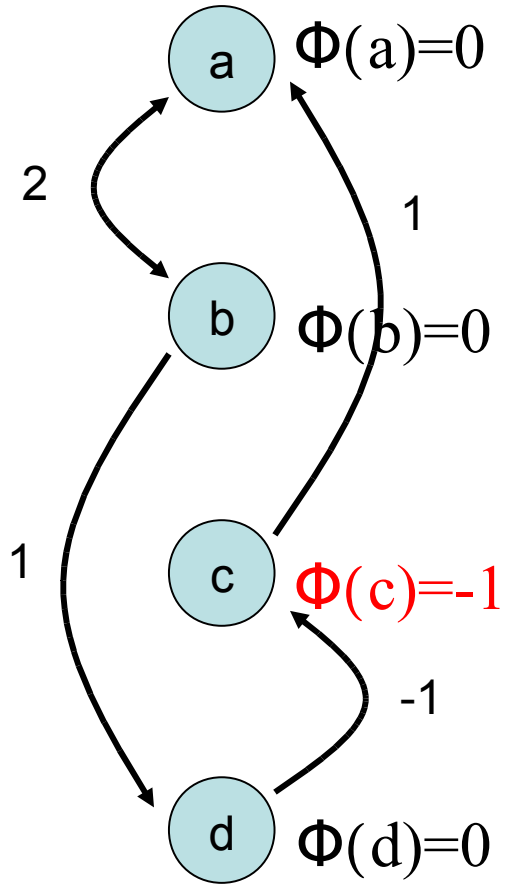


All Pairs Shortest Paths

Schritt 5: berechne korrekte Distanzen durch die Formel

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

$\bar{\mu}$	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:

$$O(T_{\text{Bellman-Ford}}(n,m) + n \cdot T_{\text{Dijk stra}}(n,m))$$
$$= O(n \cdot m + n(n \log n + m))$$
$$= O(n \cdot m + n^2 \log n)$$

unter Verwendung von Fibonacci Heaps.