
Algorithmen und Datenstrukturen

Graphen

Prof. Dr. Ralf Möller

Universität zu Lübeck

Institut für Informationssysteme

Magnus Bender (Übungen)

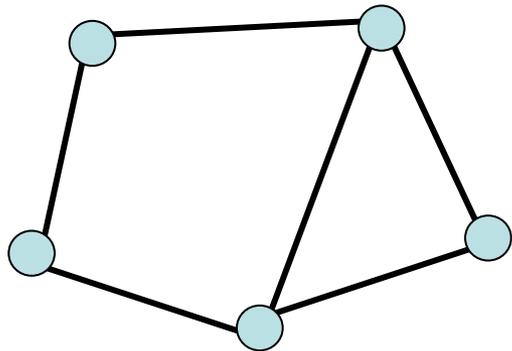
sowie viele Tutoren



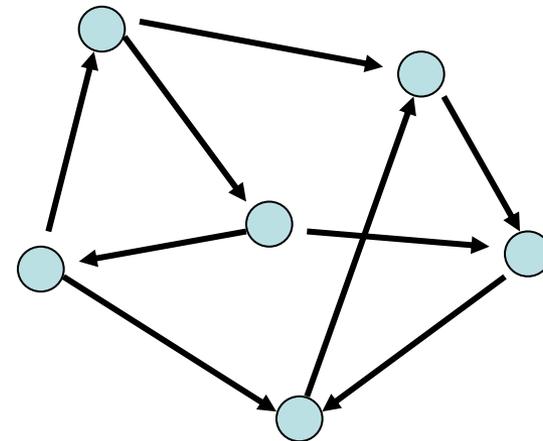
Graphen

Graph $G=(V, E)$ besteht aus

- Knotenmenge V
- Kantenmenge E



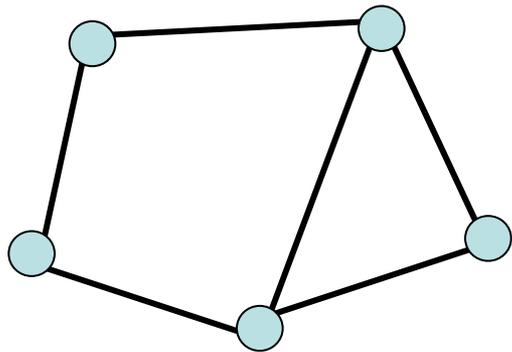
ungerichteter Graph



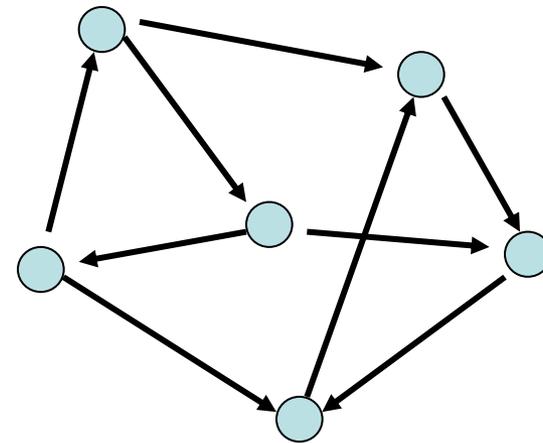
gerichteter Graph

Graphen

- **Ungerichteter Graph:** Kante repräsentiert durch Menge $\{v,w\}$ mit $v, w \in V$
- **Gerichteter Graph:** Kante repräsentiert durch Paar $(v,w) \in V \times V$ (bedeutet $v \rightarrow w$)



ungerichteter Graph



gerichteter Graph

Graphen

- **Ungerichtete Graphen: Symmetrische** Beziehungen jeglicher Art
 - z.B. $\{v,w\} \in E$ genau dann, wenn Distanz zwischen v und w maximal 1 km
- **Gerichtete Graphen: Asymmetrische** Beziehungen
 - z.B. $(v,w) \in E$ genau dann, wenn Person v einer Person w eine Nachricht sendet
- **Grad eines Knotens:** Anzahl der ausgehenden Kanten

Graphen

Im Folgenden: **nur gerichtete Graphen.**

Modellierung eines ungerichteten Graphen als gerichteter Graph:

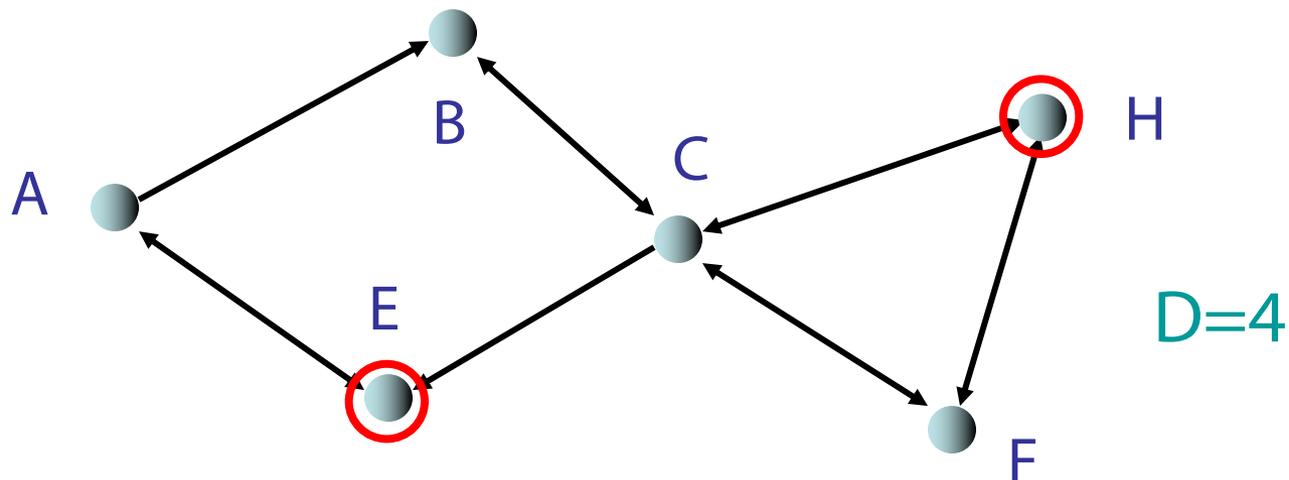


Ungerichtete Kante ersetzt durch zwei gerichtete Kanten.

- **n**: aktuelle Anzahl Knoten
- **m**: aktuelle Anzahl Kanten

Graphen

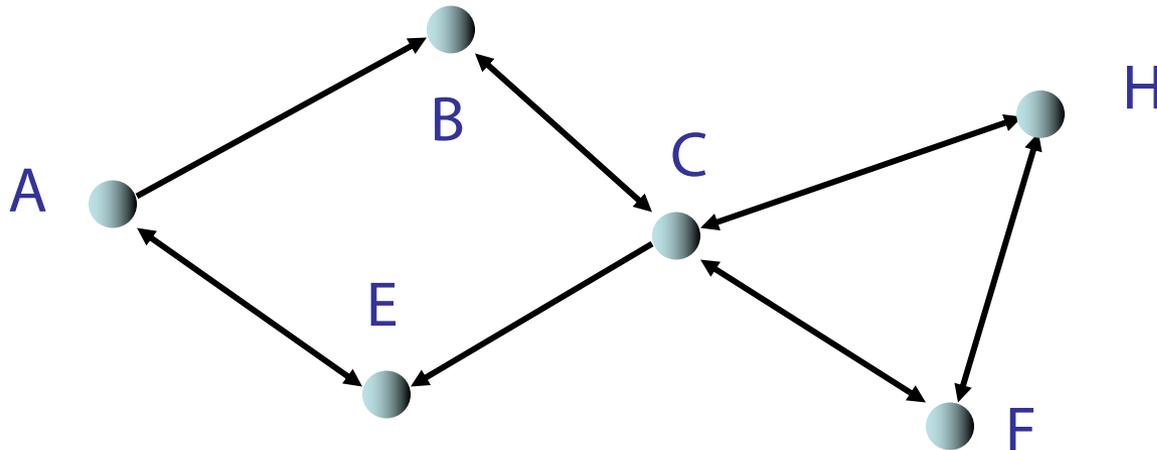
- $\delta(v,w)$: **Distanz**
Länge eines kürzesten gerichteten Weges von w zu v in G , ∞ wenn v von w nicht erreichbar
- $D = \max_{v,w} \delta(v,w)$: **Durchmesser** von G



Graphen

G heißt

- (schwach) zusammenhängend: Durchmesser D endlich, wenn alle Kanten als ungerichtet betrachtet werden
- stark zusammenhängend: wenn D endlich



Operationen auf Graphen

Sei $G=(V, E)$ ein Graph,
 $e \in E$ eine Kante und $v \in V$ ein Knoten

Operationen:

- **insert**(e, G): $E = E \cup \{e\}$
- **remove**(i, j, G): $E = E \setminus \{e\}$ für die Kante $e=(v,w)$
mit $\text{key}(v)=i$ und $\text{key}(w)=j$
- **insert**(v, G): $V = V \cup \{v\}$
- **remove**(i, G): Sei $v \in V$ der Knoten mit $\text{key}(v)=i$
 $V = V \setminus \{v\}, E = E \setminus \{(x,y) \mid x=v \vee y=v\}$
- **find**(i, G): gib Knoten v aus mit $\text{key}(v)=i$
- **find**(i, j, G): gib Kante (v,w) aus
mit $\text{key}(v)=i$ und $\text{key}(w)=j$

Operationen auf Graphen

Anzahl der Knoten oft **fest**. In diesem Fall:

- $V = \{1, \dots, n\}$ (Knoten hintereinander nummeriert, identifiziert durch ihren Schlüssel aus $\{1, \dots, n\}$)

Relevante Operationen:

- **insert**(e, G): $E = E \cup \{e\}$
- **remove**(i, j, G): $E = E \setminus \{e\}$ für die Kante $e = (i, j)$
- **find**(i, j, G): gib Kante $e = (i, j)$ aus

Operationen auf Graphen

Anzahl der Knoten **variabel**:

- **Hashing** kann verwendet werden, um Schlüssel von n Knoten in Bereich $\{1, \dots, O(n)\}$ zu hashen
- Damit kann variabler Fall auf den Fall einer statischen Knotenmenge reduziert werden. (Nur $O(1)$ -Vergrößerung gegenüber statischer Datenstruktur)

Operationen auf Graphen

Im Folgenden:

Konzentration auf statische Anzahl an Knoten.

Parameter für Laufzeitanalyse:

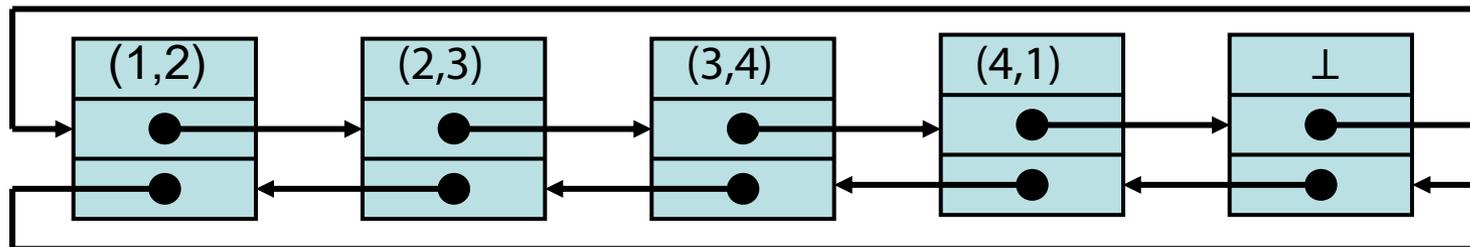
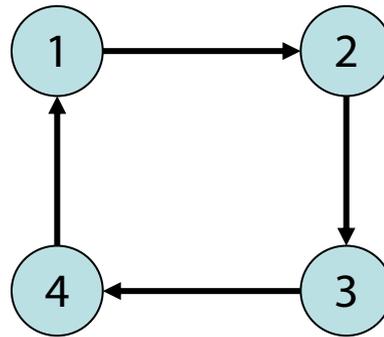
- n : Anzahl Knoten
- m : Anzahl Kanten
- d : maximaler Knotengrad (maximale Anzahl ausgehender Kanten von Knoten)

Graphrepräsentationen

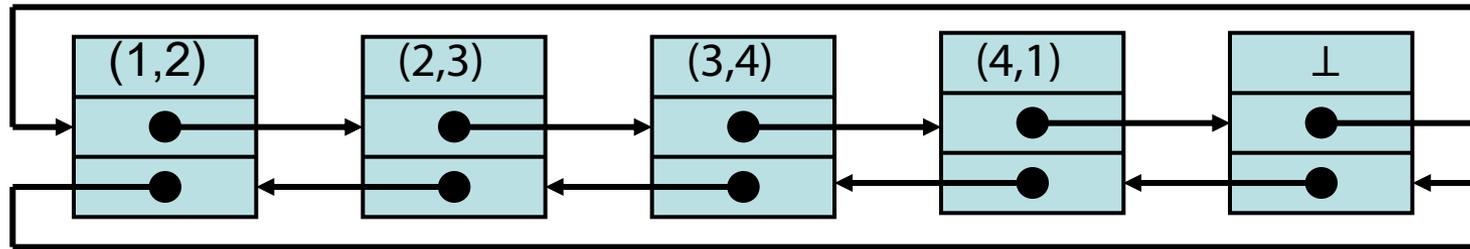
1. Sequenz von Kanten
2. Adjazenzfeld
3. Adjazenzliste
4. Adjazenzmatrix
5. Adjazenzliste + Hashtabelle
6. Implizite Repräsentationen

Graphrepräsentationen

1: Sequenz von Kanten



Sequenz von Kanten

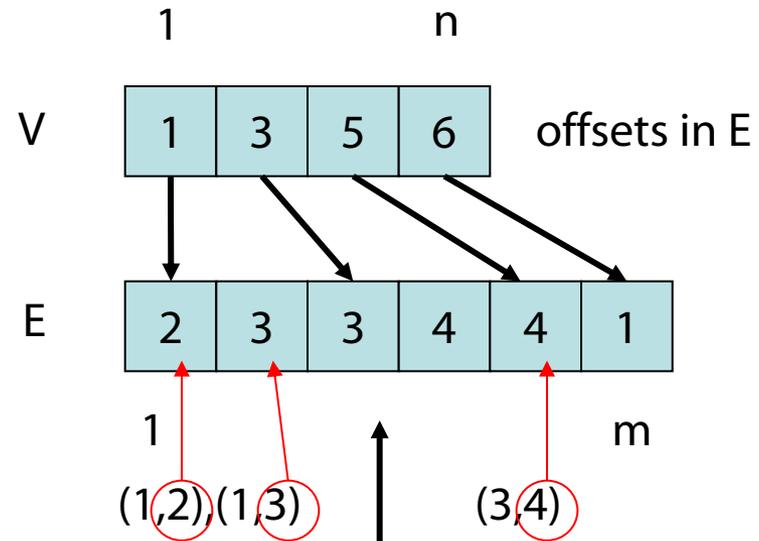
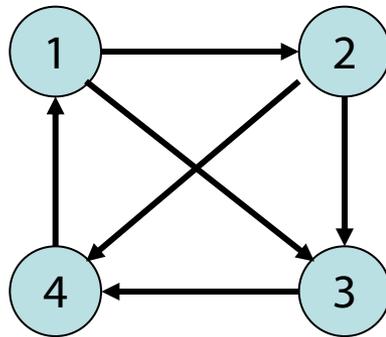


Zeitaufwand:

- **find**(i, j, G): $\Theta(m)$ im schlimmsten Fall
- **insert**(e, G): $O(1)$
- **remove**(i, j, G): $\Theta(m)$ im schlimmsten Fall

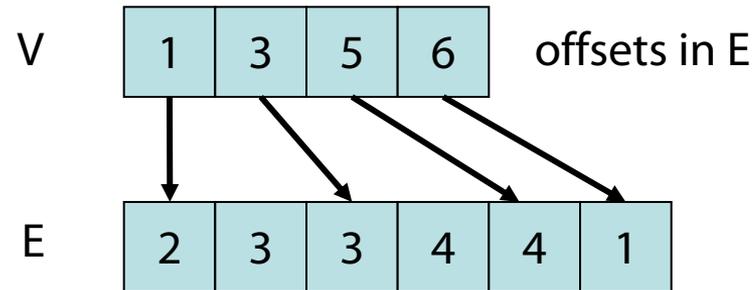
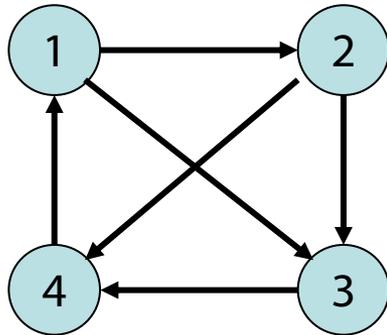
Graphrepräsentationen

2: Adjazenzfeld



Nur
Zielschlüssel
dargestellt

Adjazenzfeld

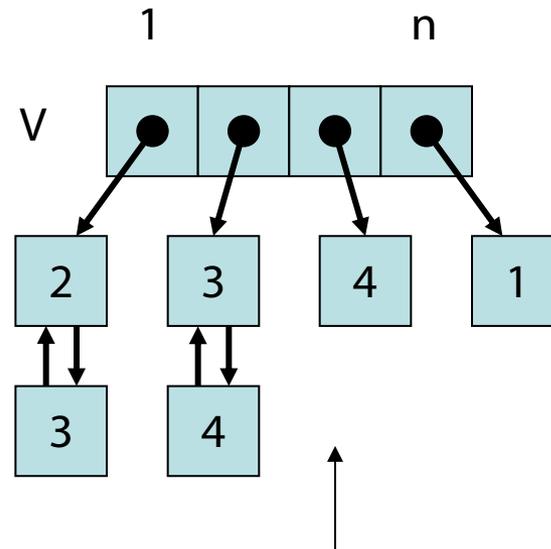
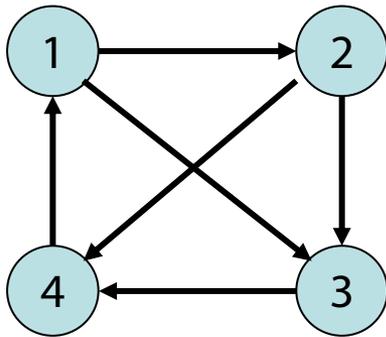


Zeitaufwand:

- **find**(i, j, G): Zeit $O(d)$
- **insert**(e, G): Zeit $O(m)$ (schlimmster Fall)
- **remove**(i, j, G): Zeit $O(m)$ (schlimmster Fall)

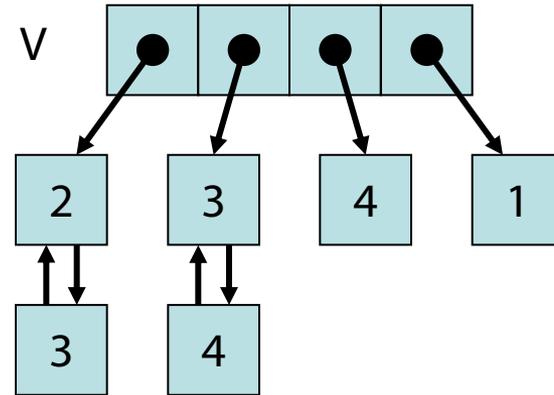
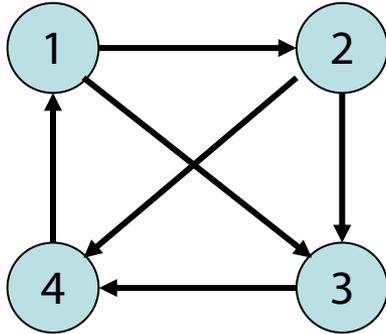
Graphrepräsentationen

3: Adjazenzliste



Nur
Zielschlüssel
dargestellt

Adjazenzliste



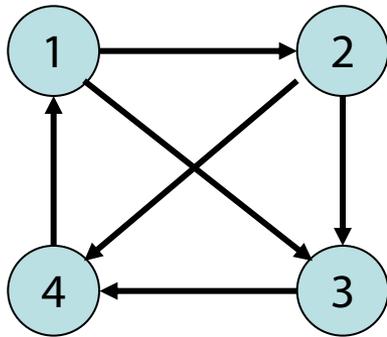
Zeitaufwand:

- **find**(i, j, G): Zeit $O(d)$
- **insert**(e, G): Zeit $O(d)$
- **remove**(i, j, G): Zeit $O(d)$

Problem: d kann auch groß sein!

Graphrepräsentationen

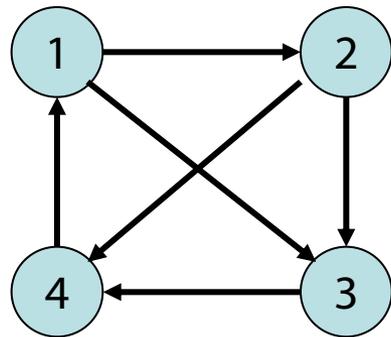
4: Adjazenzmatrix



$$A = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

- $A[i,j] \in \{0,1\}$ (bzw. Zeiger auf ein $e \in E$)
- $A[i,j] = 1$ genau dann, wenn $(i,j) \in E$

Adjazenzmatrix



$$A = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

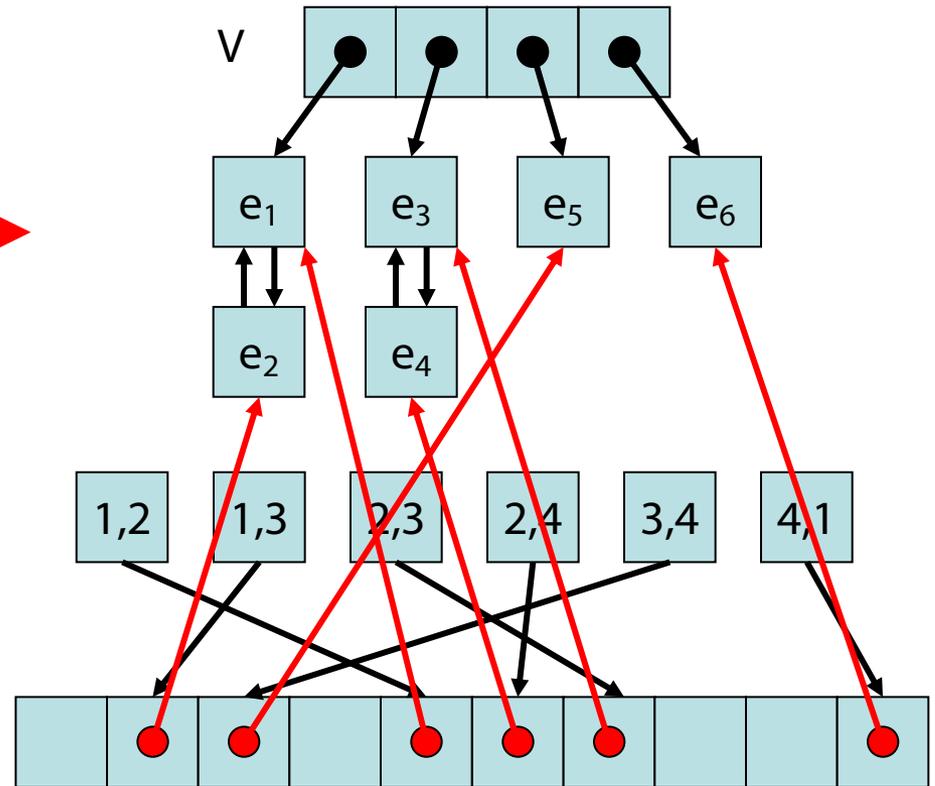
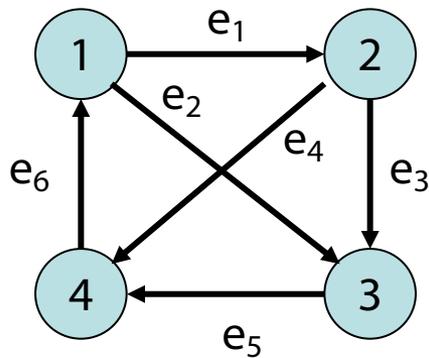
Zeitaufwand:

- **find**(i, j, G): Zeit $O(1)$
- **insert**(e, G): Zeit $O(1)$
- **remove**(i, j, G): Zeit $O(1)$

Aber: Speicher-
aufwand $O(n^2)$

Graphrepräsentationen

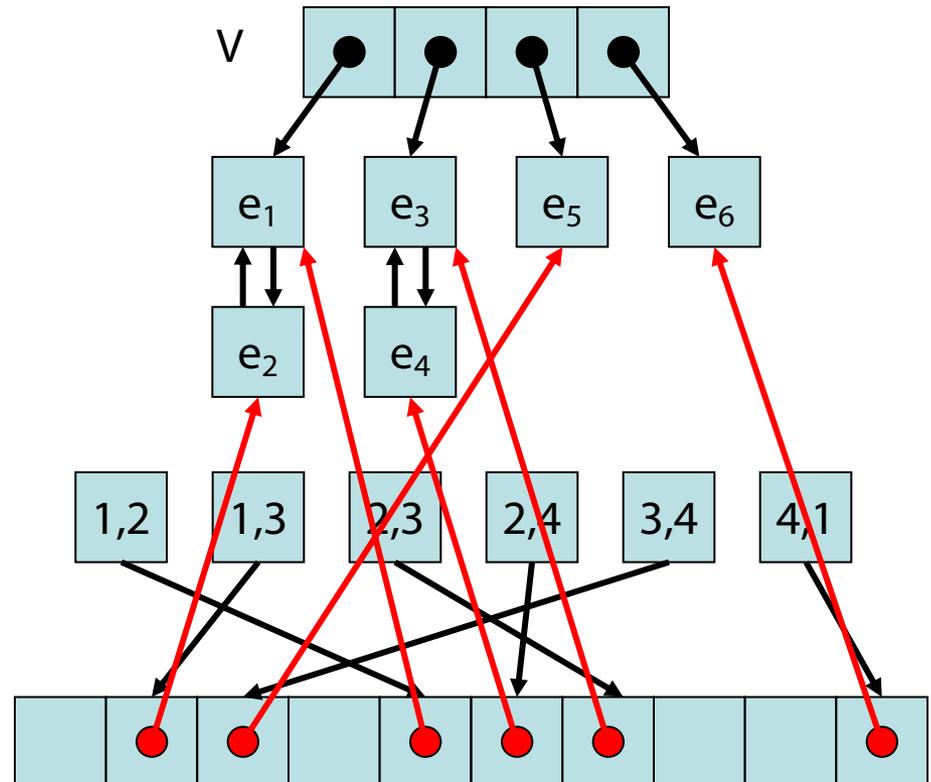
5: Adjazenzliste + Hashtabelle



Adjazenzliste+Hashtabelle

Zeitaufwand (grob):

- **find**(i, j, G):
 $O(1)$ (worst case)
- **insert**(e, G):
 $O(1)$ (amortisiert)
- **remove**(i, j, G):
 $O(1)$ (amortisiert)
- Speicher: $O(n+m)$



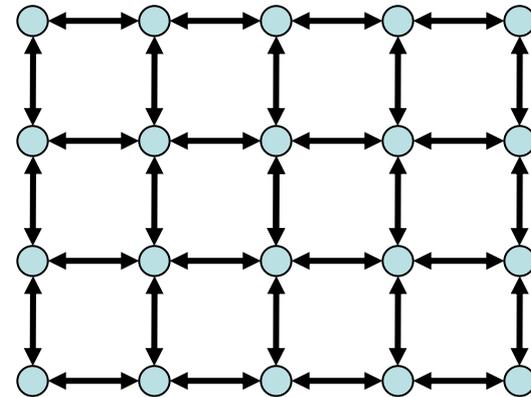
Graphrepräsentationen

6: Implizite Repräsentationen

(k,l) -Gitter $G=(V,E)$:

- $V=[k] \times [l]$ ($[a]=\{0,\dots,a-1\}$ für $a \in \mathbb{N}$)
- $E=\{((v,w),(x,y)) \mid (v=x \wedge |w-y|=1) \vee (w=y \wedge |v-x|=1)\}$

Beispiel: $(5,4)$ -Gitter



Graphrepräsentationen

6: Implizite Repräsentationen

(k,l) -Gitter $G=(V,E)$:

- $V=[k] \times [l]$ ($[a]=\{0,\dots,a-1\}$ für $a \in \mathbb{N}$)
- $E=\{((v,w),(x,y)) \mid (v=x \wedge |w-y|=1) \vee (w=y \wedge |v-x|=1)\}$
- Speicheraufwand: $O(\log k + \log l)$
(speichere Kantenregel sowie k und l)
- Find-Operation: $O(1)$ Zeit (reine Rechnung)

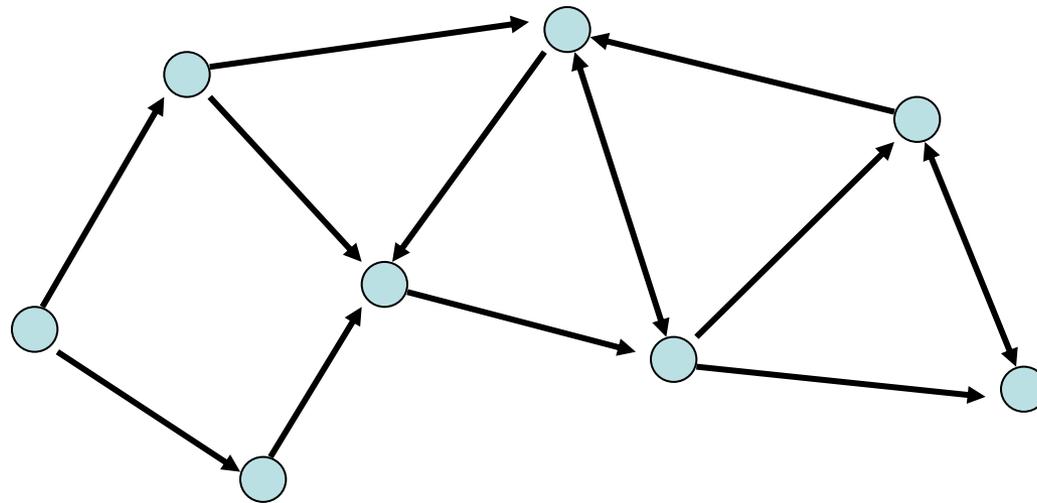
Graphen

Traversierung



Graphdurchlauf

Zentrale Frage: Wie können wir die Knoten eines Graphen durchlaufen, so dass jeder Knoten mindestens einmal besucht wird?



Graphdurchlauf

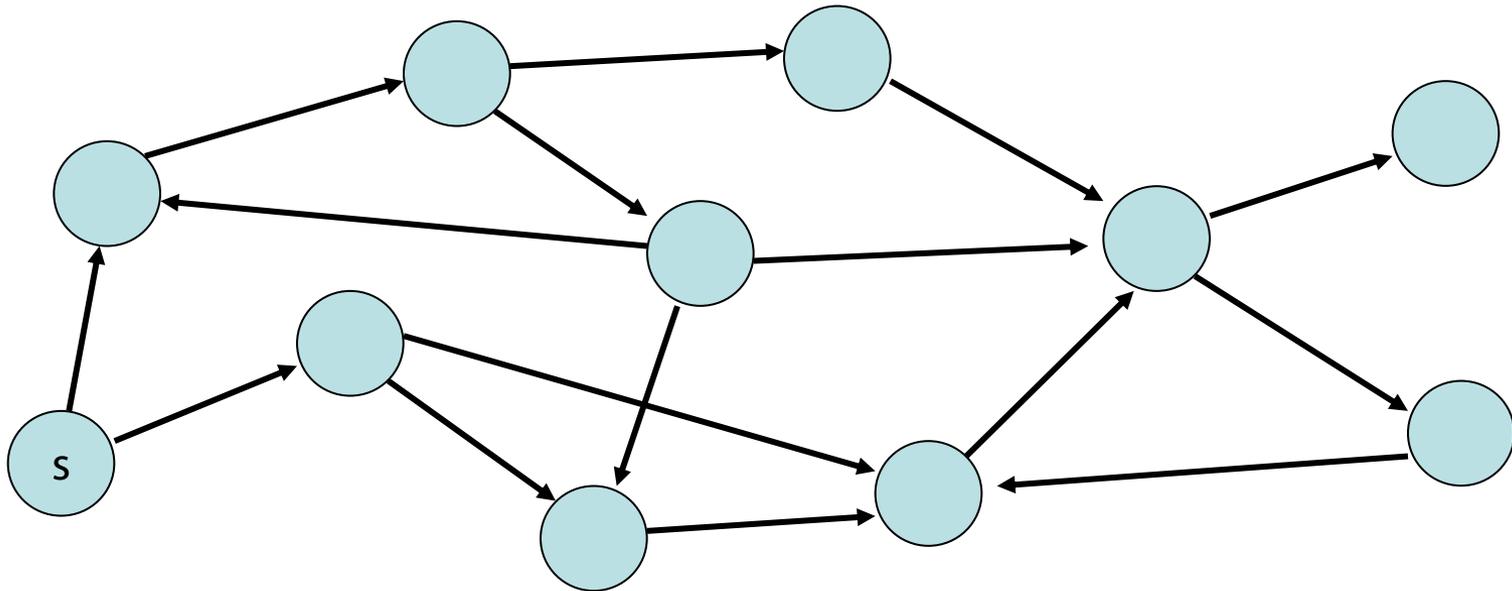
Zentrale Frage: Wie können wir die Knoten eines Graphen durchlaufen, so dass jeder Knoten mindestens einmal besucht wird?

Grundlegende Strategien:

- Breitensuche
- Tiefensuche

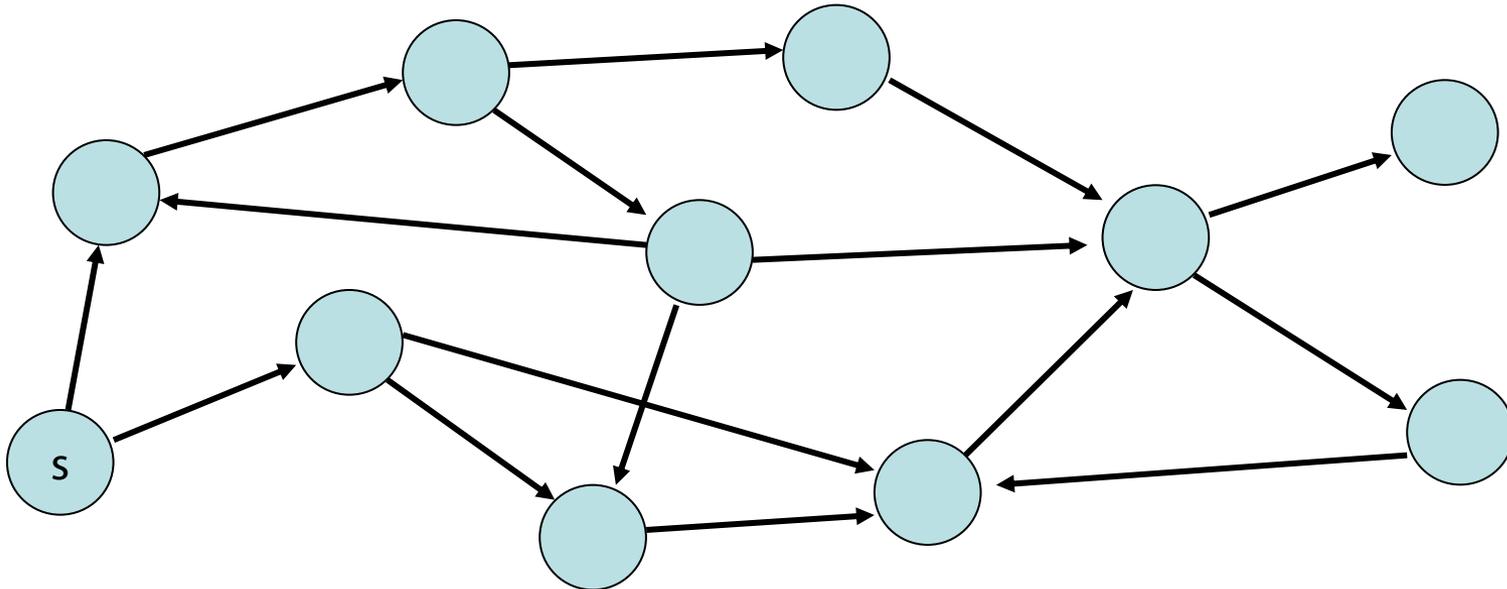
Breitensuche

- Starte von einem Knoten s
- Exploriere Graph Distanz für Distanz



Tiefensuche

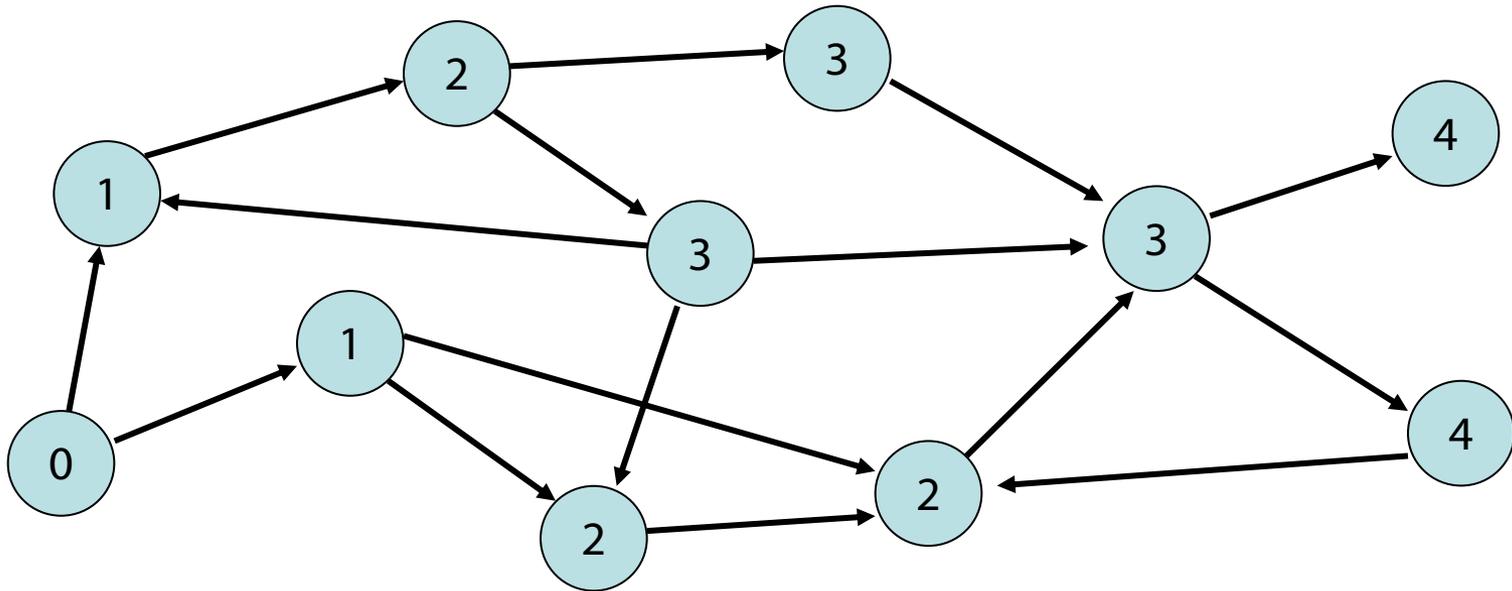
- Starte von einem Knoten s
- Exploriere Graph in die Tiefe
(●: aktuell, ●: noch aktiv, ●: fertig)



Breitensuche

- **v.d**: Distanz von Knoten **v** zu **s** ($s.d = 0$)
- **v.parent**: Knoten, von dem **v** besucht

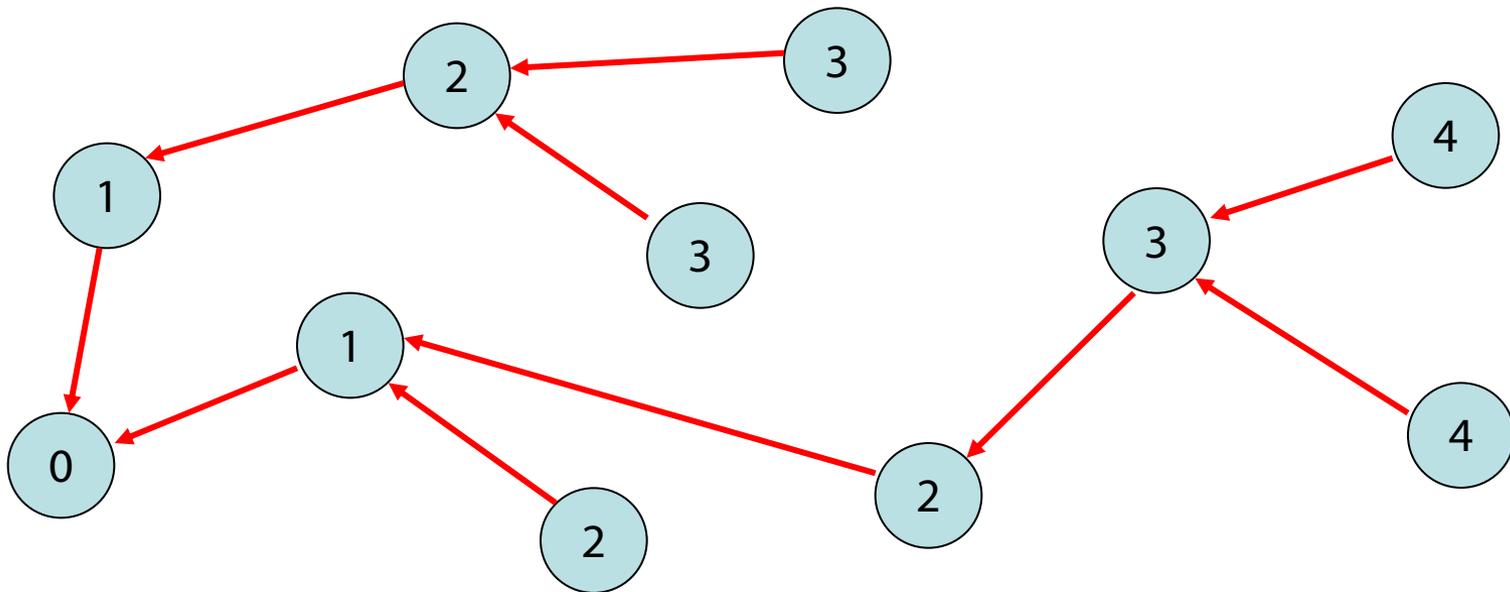
Distanzen:



Breitensuche

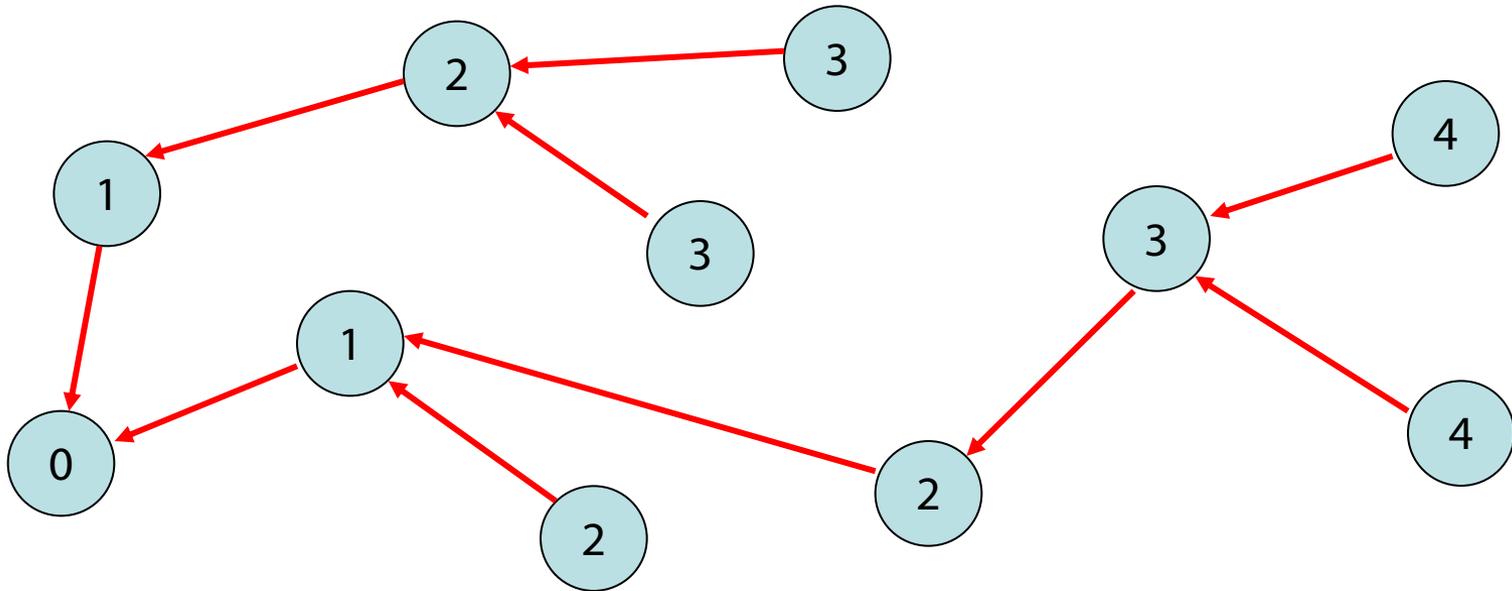
- $v.d$: Distanz von Knoten v zu s ($s.d = 0$)
- $v.parent$: Knoten, von dem v besucht

Mögliche Parent-Beziehungen in rot:



Breitensuche

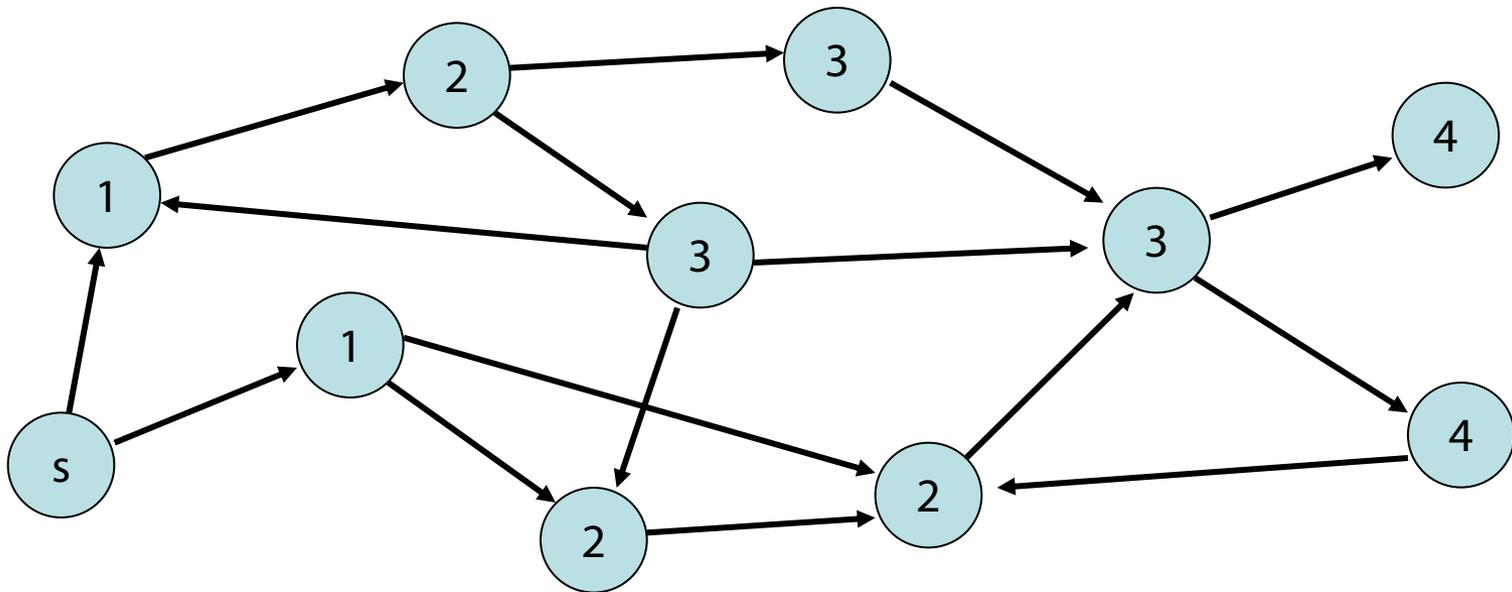
Parent-Beziehung eindeutig: wenn Knoten v zum ersten Mal besucht wird, wird $v.parent$ gesetzt und dadurch v markiert, so dass v nicht nochmal besucht wird



Breitensuche

Kantentypen:

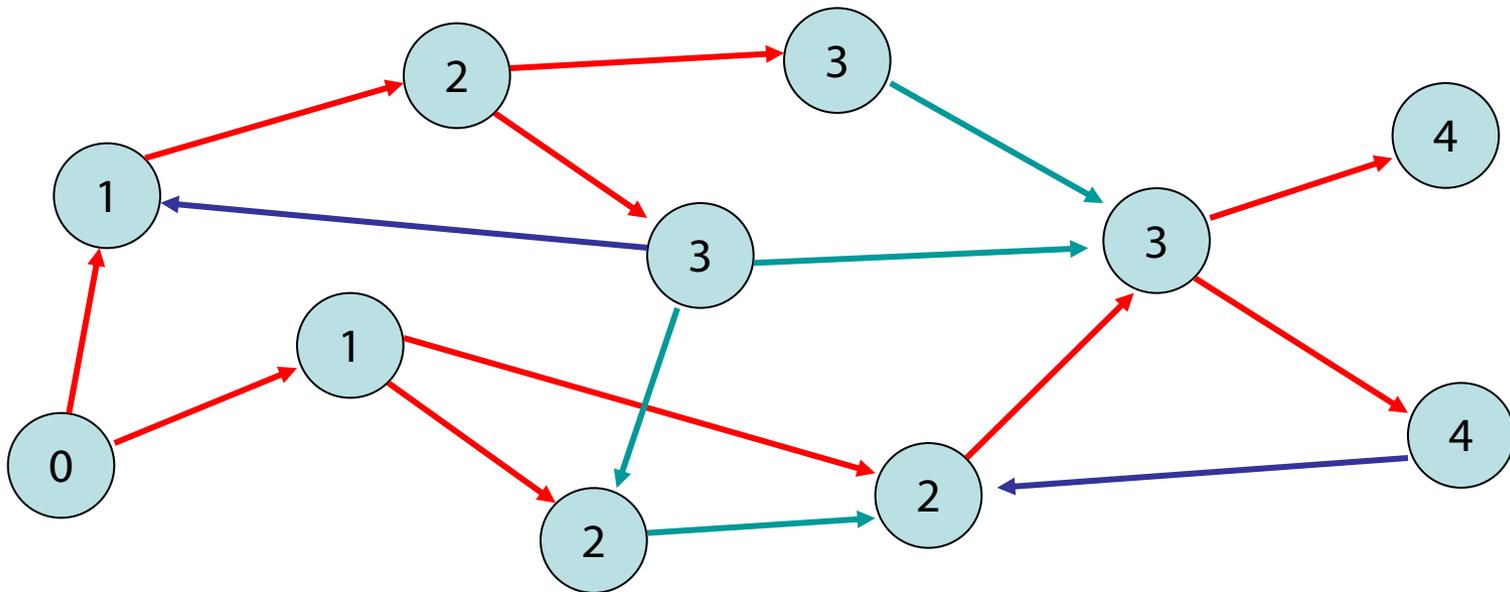
- **Baumkante:** zum Kind
- **Rückwärtskante:** zu einem Vorfahr
- **Kreuzkante:** alle sonstige Kanten



Breitensuche

Kantentypen:

- **Baumkante:** zum Kind
- **Rückwärtskante:** zu einem Vorfahr
- **Kreuzkante:** alle sonstige Kanten



Iteratoren

Sei $G=(V, E)$ ein Graph

- Iteration über Kanten:

- `for (u, v) in edges(G) ... end`

- Finde alle von **u** ausgehenden Kanten

- `for (_, v) in edges_out(u, G) ... end`

- Finde alle bei **v** eintreffenden Kanten

- `for (u, _) in edges_in(v, G) ... end`

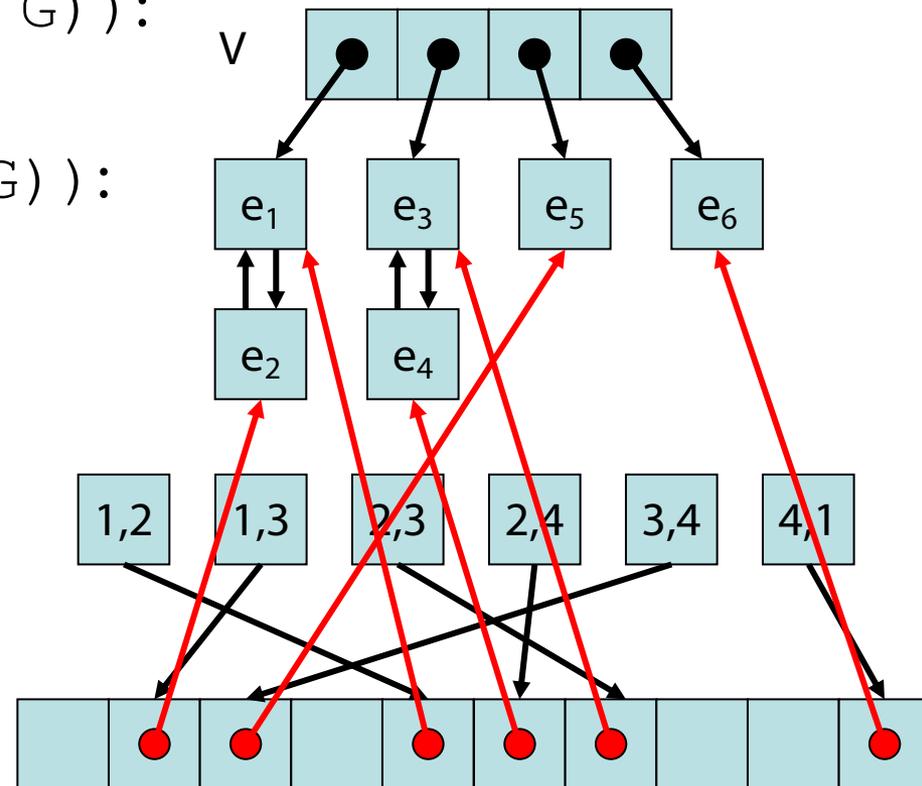
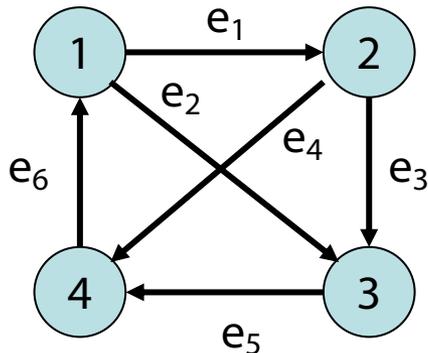
- Iteration über Knoten:

- `for v in nodes(G) ... end`

Iteratoren

Zeitaufwand (grob):

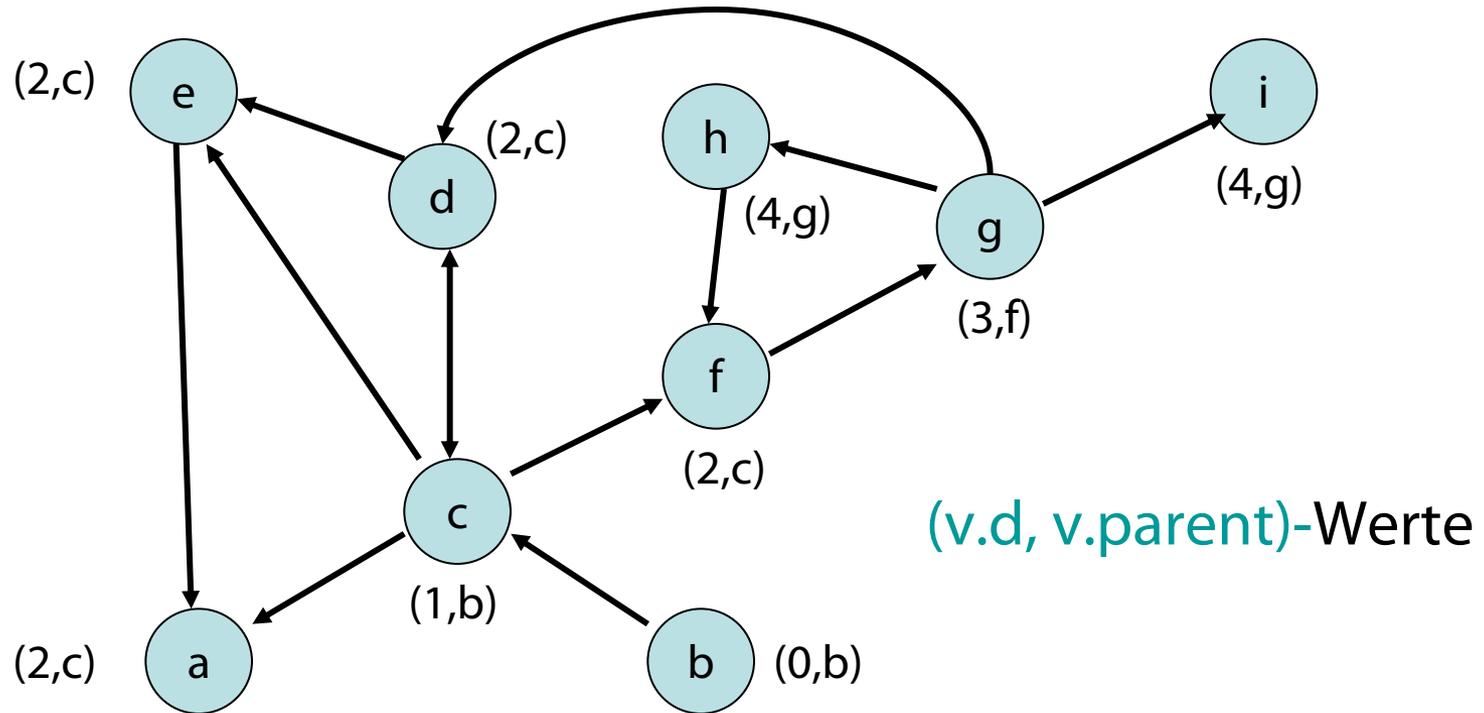
- `iterate(edges_out(v, G))`:
 $O(1)$ (worst case)
- `iterate(edges_in(v, G))`:
 $O(?)$ (worst case)



Breitensuche

```
function bfs(s, g) # Breitensuche ausgehend von s in g
    s.d = 0          # s hat Distanz 0 zu sich
    s.parent = s    # s ist sein eigener Vater
    q = make_queue([s]) # Queue zu besuchender Knoten
    while !empty_queue(q) # solange q nicht leer
        u = dequeue(q) # nimm Knoten nach FIFO Regel
        for (_, v) in edges_out(u, g)
            if isnothing(v.parent) # v schon besucht?
                enqueue(v, q) # nein, dann in q hinten einfüegen
                v.d = u.d + 1
                v.parent = u
            end
        end
    end
end
end
end
```

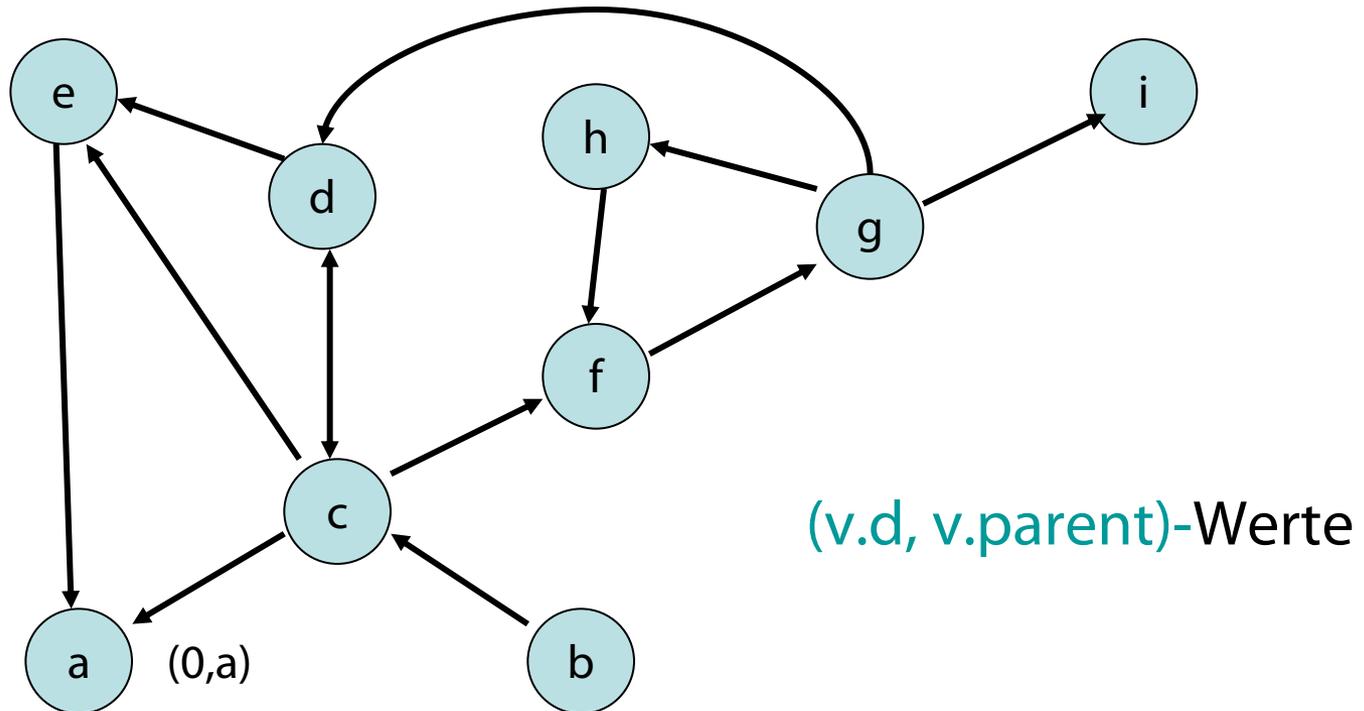
BFS(b)



● : besucht, noch in q

● : besucht, nicht mehr in q

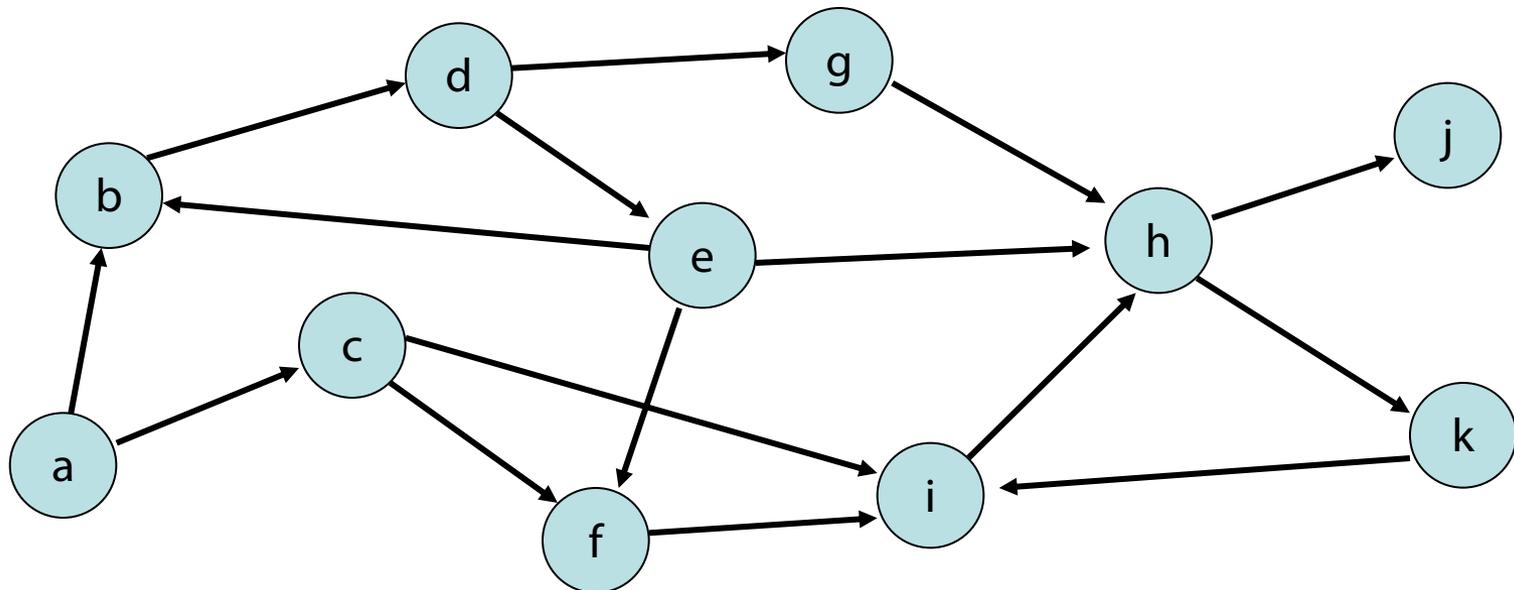
BFS(a)



Von **a** kein anderer Knoten erreichbar.

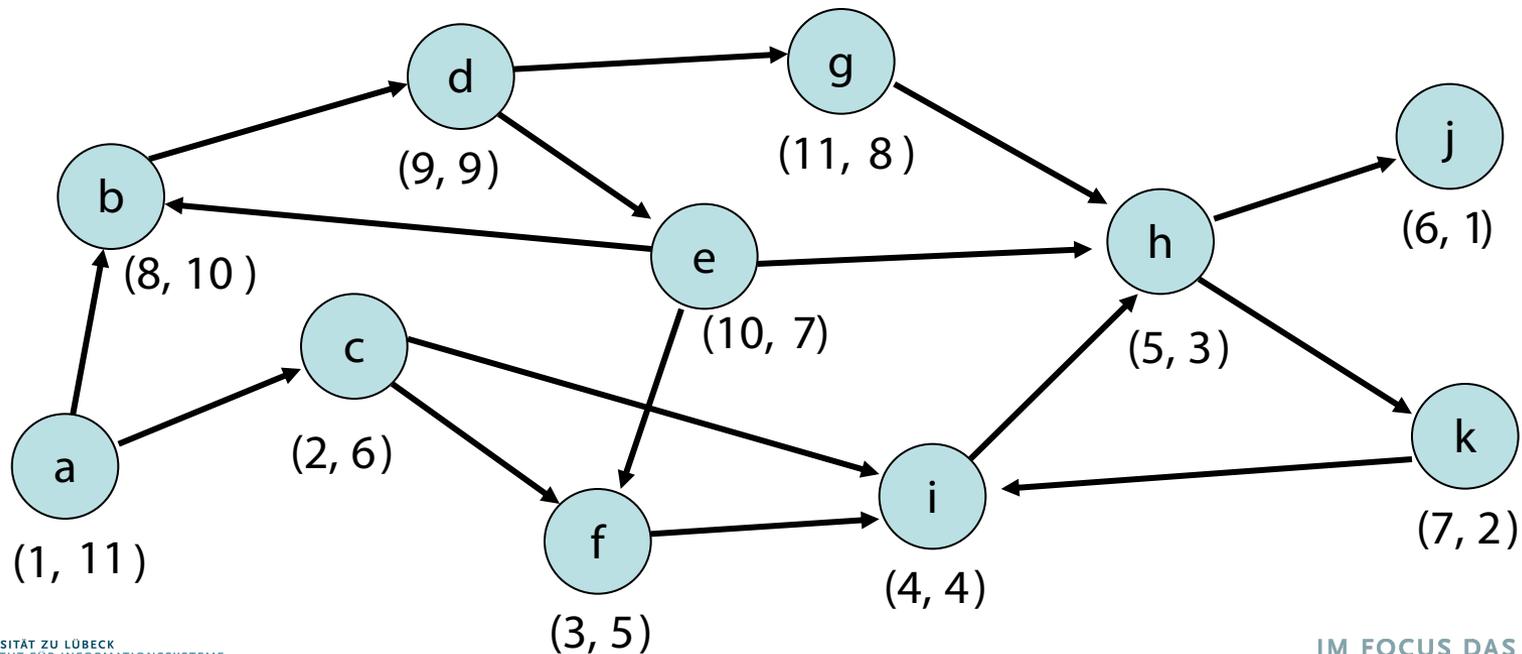
Tiefensuche

- Starte von einem Knoten s
- Exploriere Graph in die Tiefe
(●: aktuell, ●: noch aktiv, ●: fertig)



DFS-Nummerierung

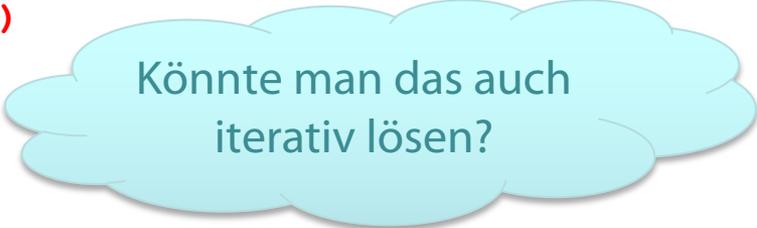
- Exploriere Graph in die Tiefe
(●: aktuell, ●: noch aktiv, ●: fertig)
- Paare (i,j) : i : dfsNum, j : finishTime



Tiefensuche – Design Pattern

```
function dfs(g)
  for v in nodes(g)    v.marked = false end # Knoten entmarkieren
  init()
  for s in nodes(g) # alle Knoten besuchen
    if !s.marked
      s.marked = true
      root(s)
      dfs(s, s, g) # s Startknoten
    end
  end
end
```

```
function dfs(u, v, g) # u ist Vater von v
  for (_, w) in edges_out(v, g)
    if w.marked    handle_non_tree_edge(v, w)
    else    traverse_tree_edge(v, w)
      w.marked = true
      dfs(v, w, g)
    end
  end
  backtrack(u, v)
end
```



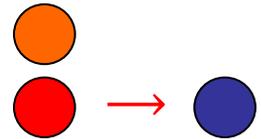
Könnte man das auch iterativ lösen?

Prozeduren in rot: noch zu spezifizieren

DFS-Nummerierung

Variablen:

- `v.dfsNum`: für jeden Knoten $v \in V$ // Zeitpunkt wenn Knoten
- `v.finishTime`: für jeden Knoten $v \in V$ // Zeitpunkt wenn Knoten
- `dfsPos, finishingTime`: \mathbb{N} // Zähler



Prozeduren:

- `init()`
`dfsPos = 1; finishingTime = 1`
- `root(s)`
`s.dfsNum = dfsPos; dfsPos = dfsPos + 1`
- `traverse_tree_edge(v, w)`
`w.dfsNum = dfsPos; dfsPos = dfsPos + 1`
- `handle_non_tree_edge(v, w)`
- `backtrack(u, v)`
`v.finishTime = finishingTime`
`finishingTime = finishingTime + 1`

DFS-Nummerierung

Ordnung $<$ auf den Knoten:

$u < v$ gdw. $u.\text{dfsNum} < v.\text{dfsNum}$

Lemma 1: Die Knoten im DFS-Rekursionsstack (alle  Knoten) sind sortiert bezüglich $<$.

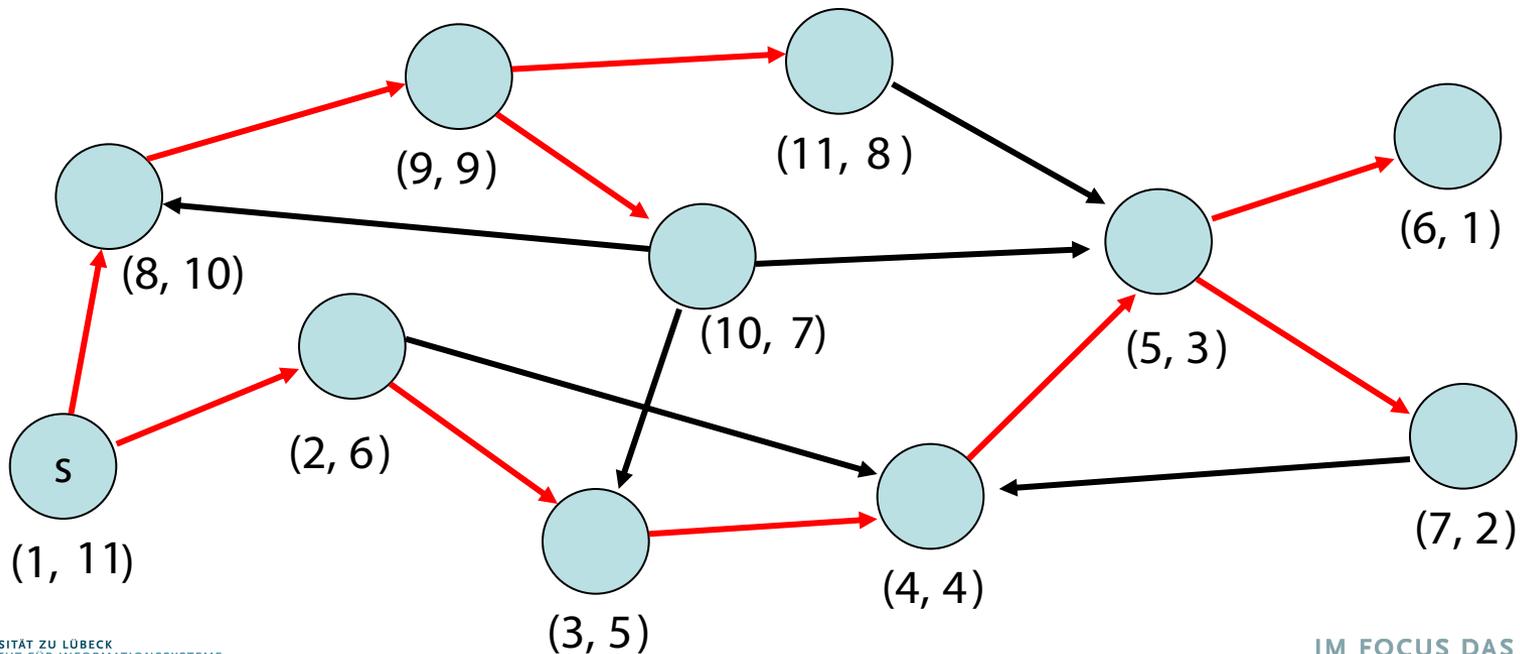
Beweis:

dfsPos wird nach jeder Zuweisung von dfsNum erhöht. Jeder neue aktive Knoten hat also immer die höchste dfsNum .

DFS-Nummerierung

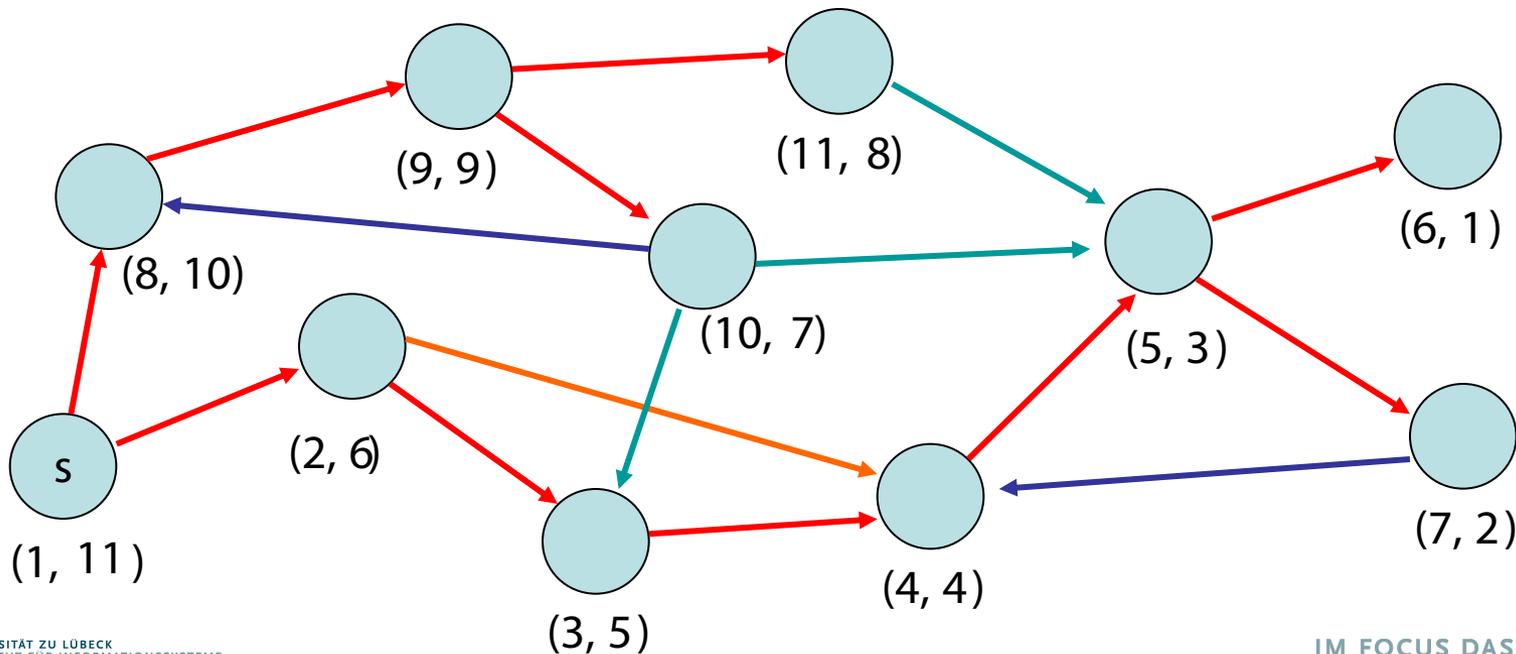
Überprüfung von Lemma 1:

- Rekursionsstack: roter Pfad von s
- Paare (i,j) : i : dfsNum, j : finishTime



DFS-Nummerierung

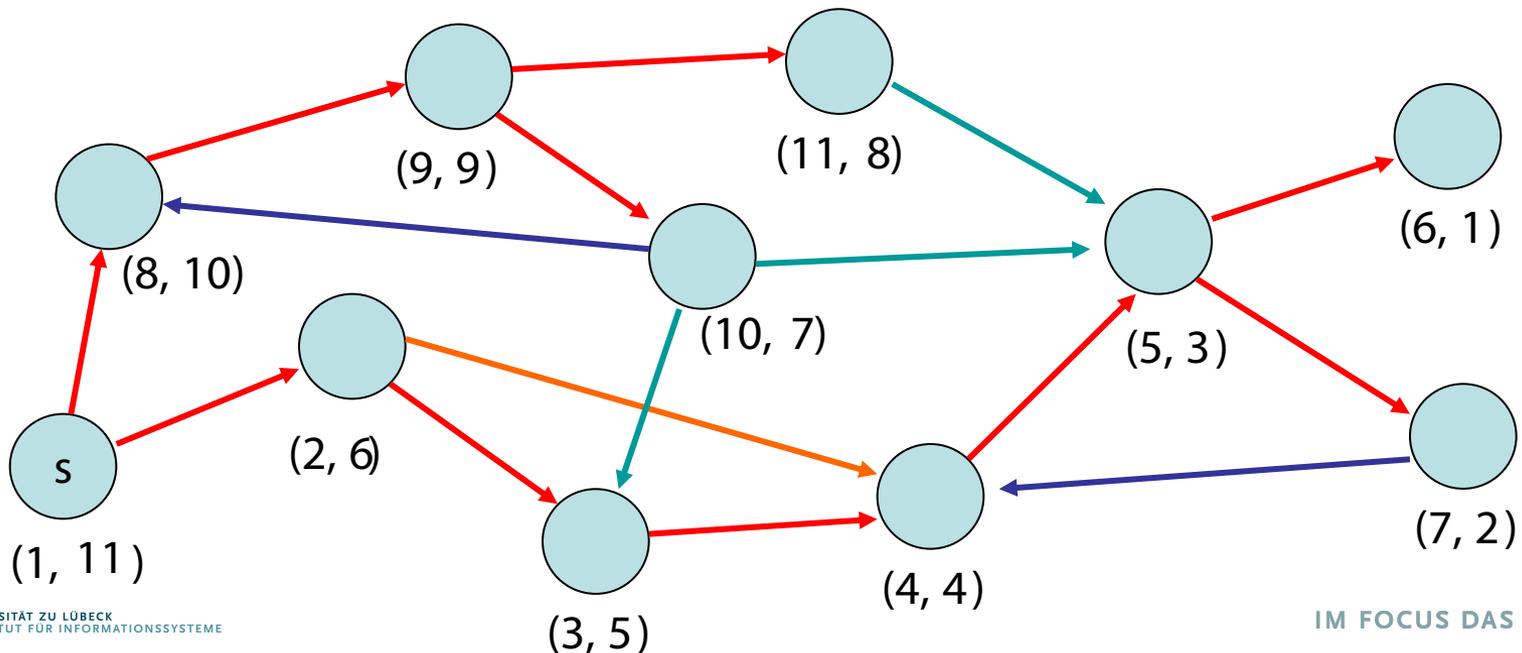
- **Baumkante:** zum Kind
- **Vorwärtskante:** zu einem Nachkommen
- **Rückwärtskante:** zu einem Vorfahr (siehe `handle_non_tree_edge`)
- **Kreuzkante:** alle sonstige Kanten (siehe `handle_non_tree_edge`)



DFS-Nummerierung

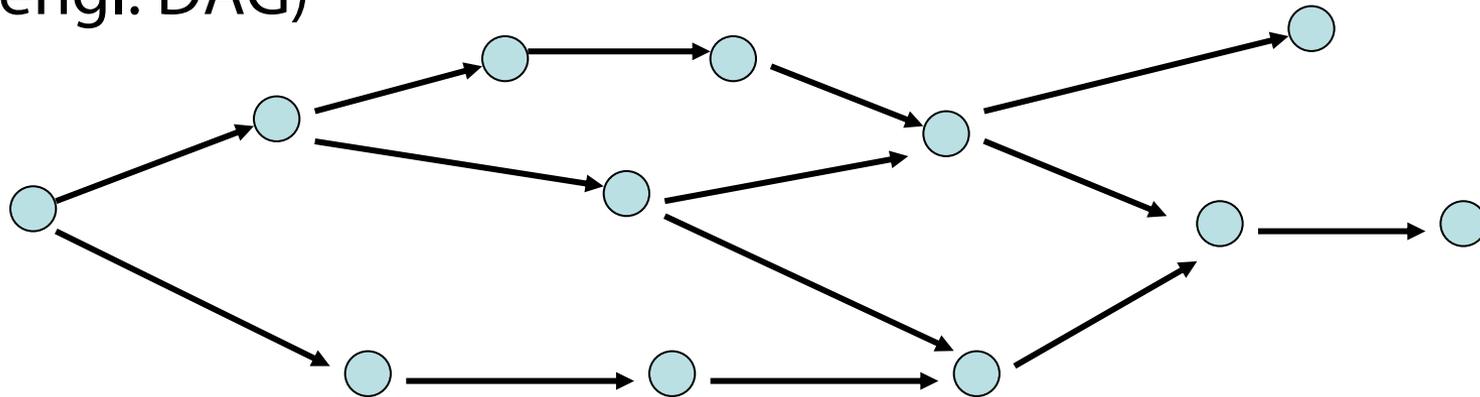
Beobachtung für
Kante (v, w) :

Kantentyp	$v.dfsNum < w.dfsNum$	$v.finishTime > w.finishTime$
Baum & Vorwärts	Ja	Ja
Rückwärts	Nein	Nein
Kreuz	Nein	Ja



DAGs

- Erkennung eines **azyklischen** gerichteten Graphen (engl. DAG)



Merkmal: keine gerichteten Kreise

Anwendung der DFS-Nummerierung:

In einem DAG gibt es keine Rückwärtskanten

DFS-Nummerierung

Behauptung: Folgende Aussagen sind äquivalent:

1. G ist ein DAG
2. DFS enthält keine Rückwärtskante
3. $\forall (v, w) \in E : v.\text{finishTime} > w.\text{finishTime}$

Beweis (2. \Leftrightarrow 3.):

2. \Leftrightarrow 3.: folgt aus Tabelle

Kantentyp	$v.\text{dfsNum} < w.\text{dfsNum}$	$v.\text{finishTime} > w.\text{finishTime}$
Baum & Vorwärts	Ja	Ja
Rückwärts	Nein	Nein
Kreuz	Nein	Ja

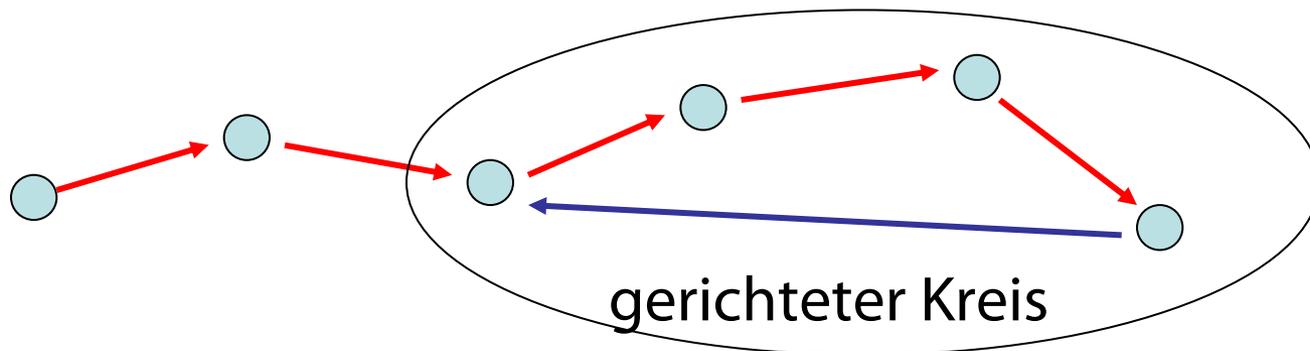
DFS-Nummerierung

Behauptung: Folgende Aussagen sind äquivalent :

1. G ist ein DAG
2. DFS enthält keine Rückwärtskante
3. $\forall (v, w) \in E : v.\text{finishTime} > w.\text{finishTime}$

Beweis (1. \Rightarrow 2.):

kontrapositiv $\neg 2. \Rightarrow \neg 1.$



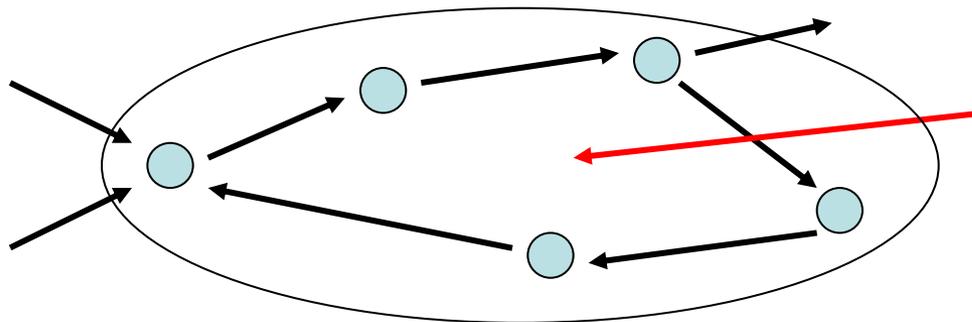
DFS-Nummerierung

Behauptung: Folgende Aussagen sind äquivalent :

1. G ist ein DAG
2. DFS enthält keine Rückwärtskante
3. $\forall (v, w) \in E : v.\text{finishTime} > w.\text{finishTime}$

Beweis (2. \Rightarrow 1.):

kontrapositiv $\neg 1. \Rightarrow \neg 2.$



Eine davon
Rückwärtskante

DFS-Nummerierung

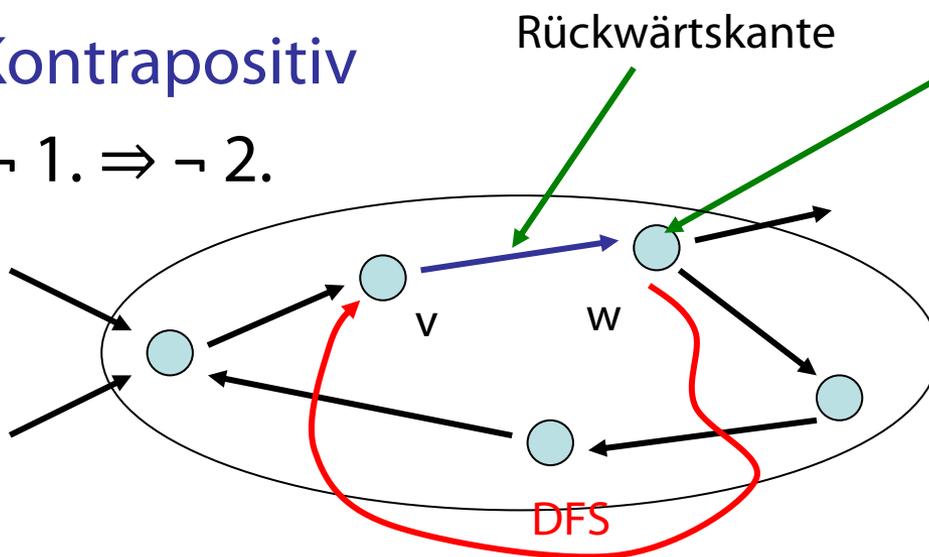
Behauptung: Folgende Aussagen sind äquivalent :

1. G ist ein DAG
2. DFS enthält keine Rückwärtskante
3. $\forall (v, w) \in E : v.finishTime > w.finishTime$

Beweis (2. \Rightarrow 1.):

Kontrapositiv

$\neg 1. \Rightarrow \neg 2.$



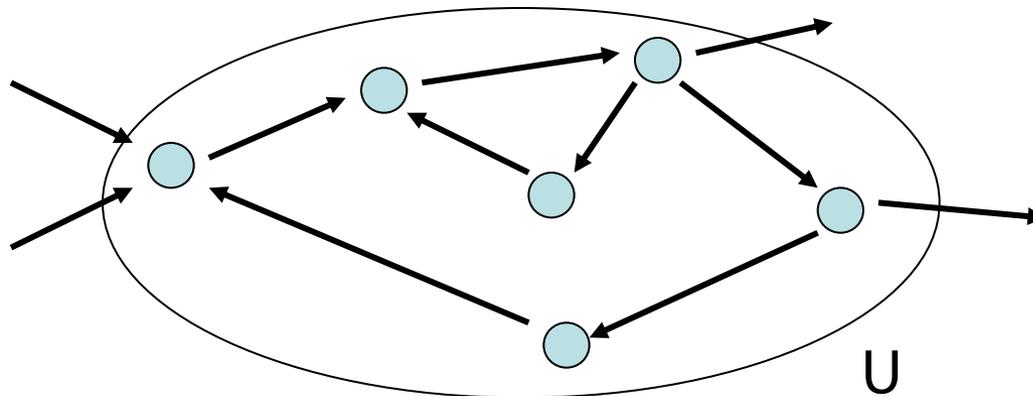
Annahme: Erster von DFS besuchter Knoten im Kreis

Kantentyp	$v.dfsNum < w.dfsNum$	$v.finishTime > w.finishTime$
Baum & Vorwärts	Ja	Ja
Rückwärts	Nein	Nein
Kreuz	Nein	Ja

Starke ZHKs

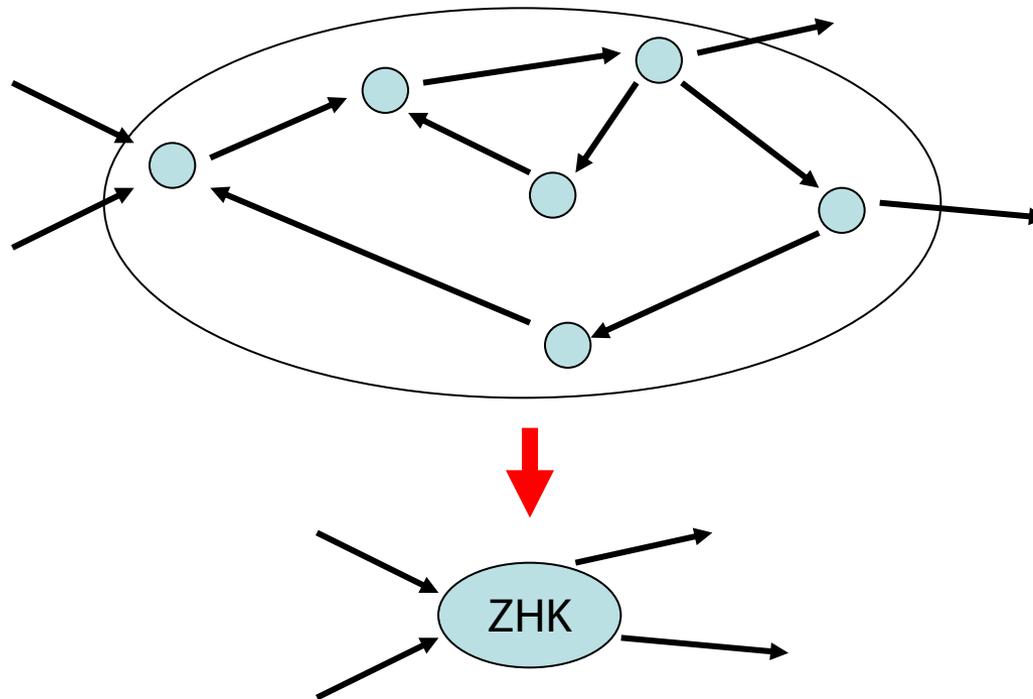
Definition: Sei $G=(V,E)$ ein gerichteter Graph.

$U \subseteq V$ ist eine **starke Zusammenhangskomponente** (ZHK) von V gdw. für alle $u, v \in U$ gibt es einen gerichteten Weg von u nach v in G und U maximal

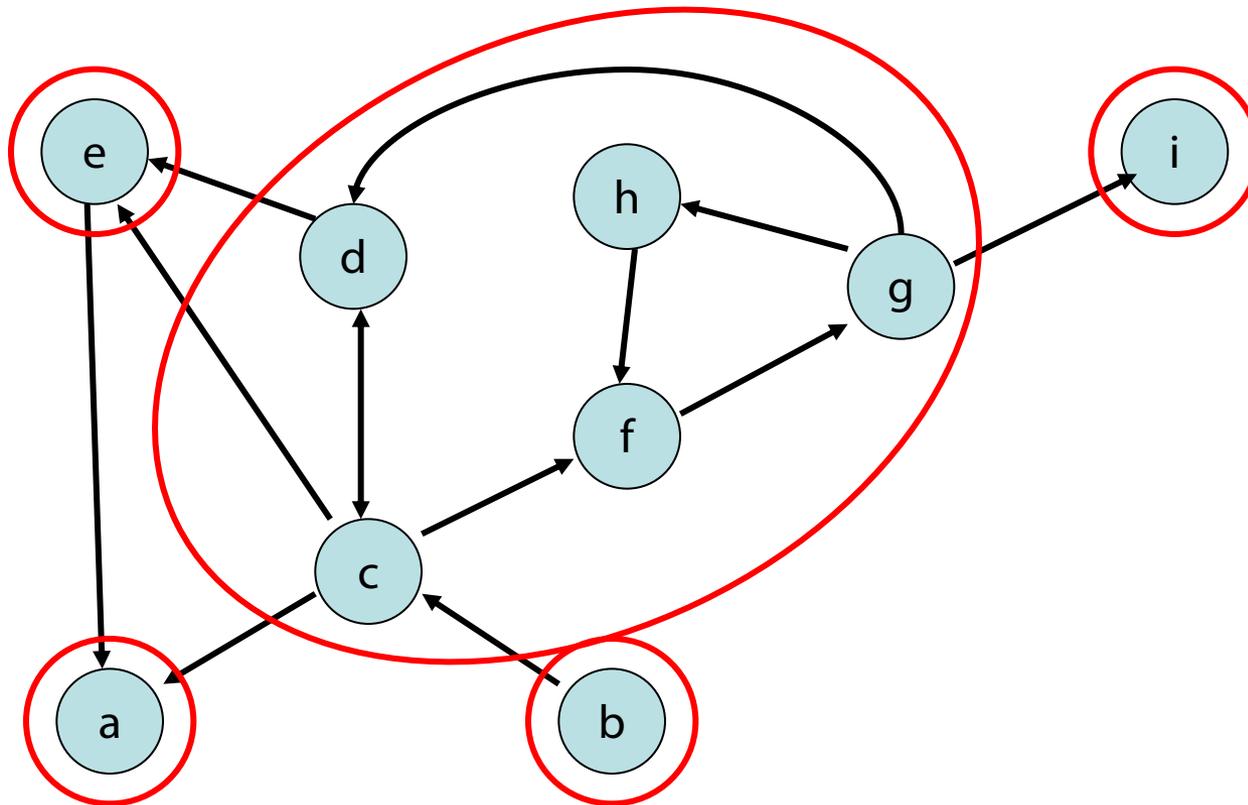


Starke ZHKs

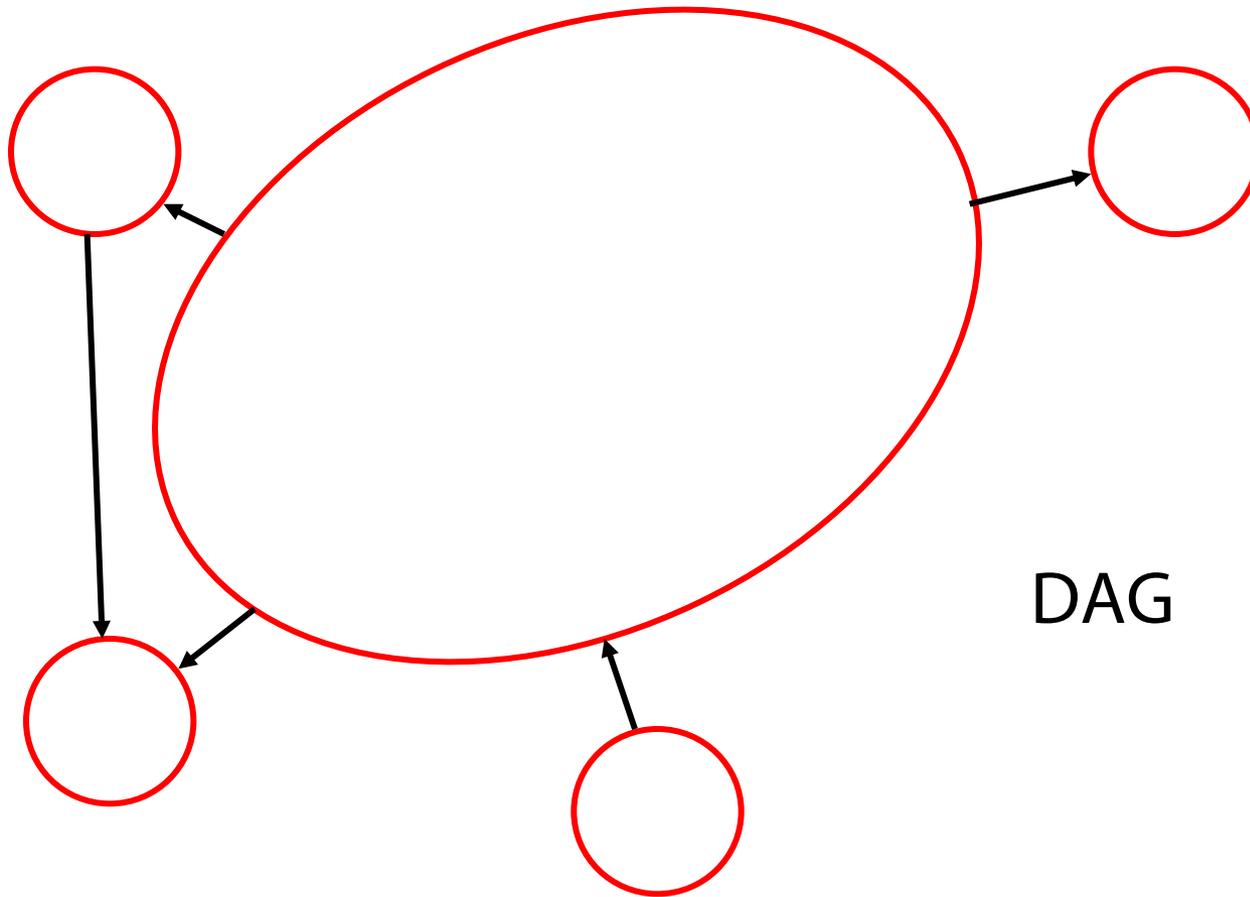
Beobachtung: Schrumpft man starke ZHKs zu einzelnen Knoten, dann ergibt sich DAG.



Starke ZHKs - Beispiel



Starke ZHKs - Beispiel



Starke ZHKs

Ziel: Finde alle starken ZHKs im Graphen in $O(n+m)$ Zeit
(n : #Knoten, m : #Kanten)

Strategie: Verwende DFS-Verfahren mit
component für jeden Knoten $v \in V$

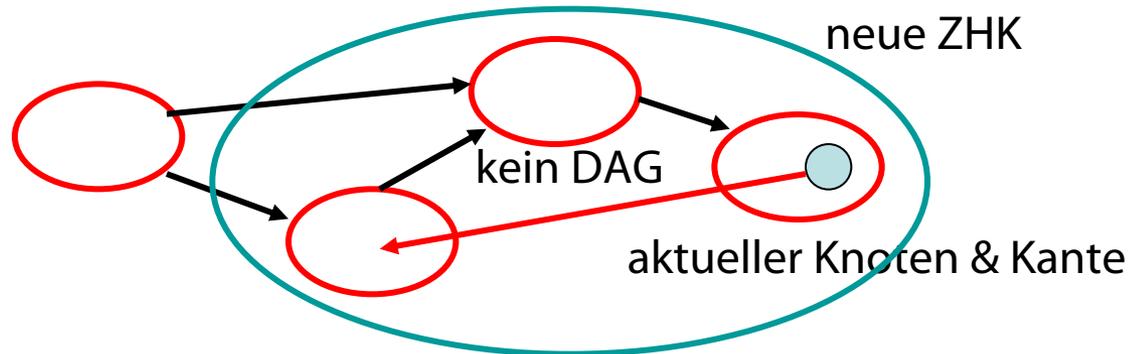
Am Ende: $v.component = w.component \Leftrightarrow$
 v und w sind in derselben starken ZHK

Starke ZHKs

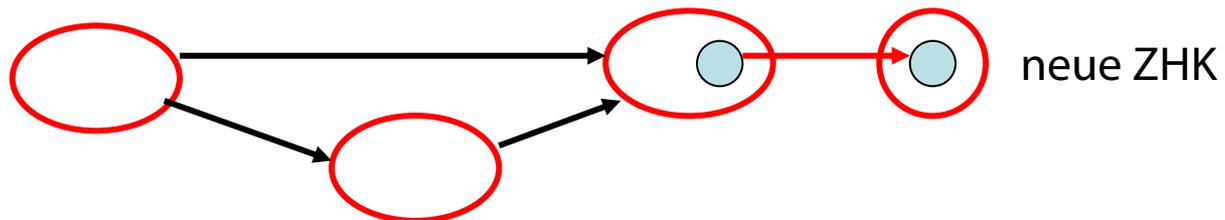
- Betrachte DFS auf $G=(V,E)$
- Sei $G_c=(V_c,E_c)$ bereits besuchter Teilgraph von G
- Ziel: bewahre starke ZHKs in G_c

- Idee:

- a)

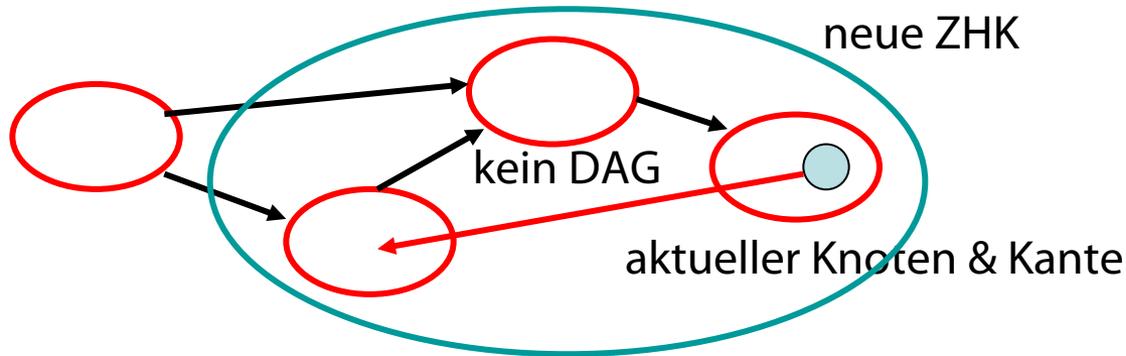


- b)

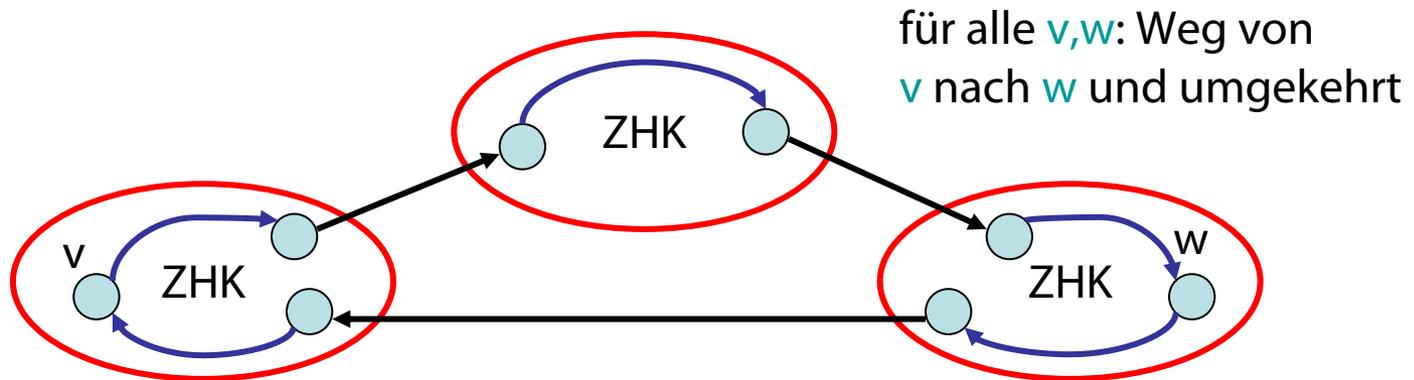


Starke ZHKs

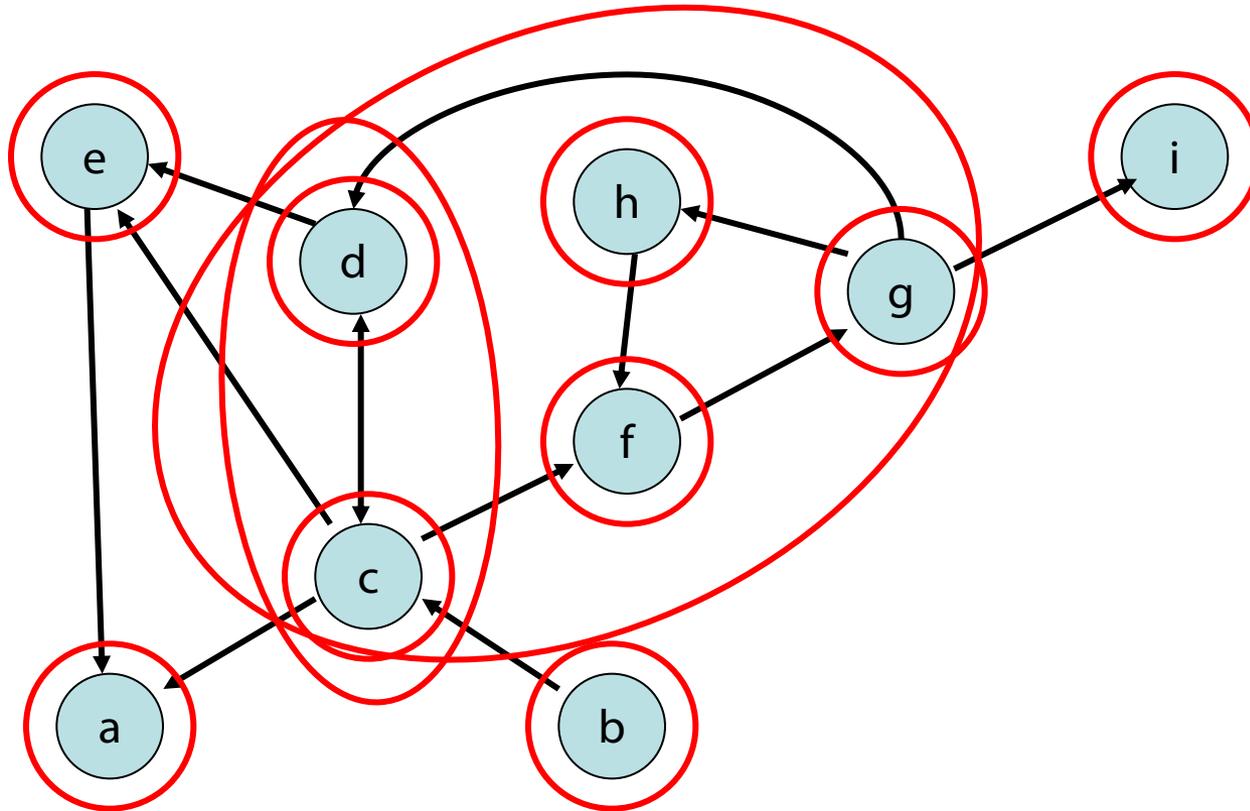
Warum ZHKs zu einer zusammenfassbar?



Grund:



Starke ZHKs - Beispiel



Problem: wie fasst man ZHKs effizient zusammen?

Starke ZHKs

Definition:

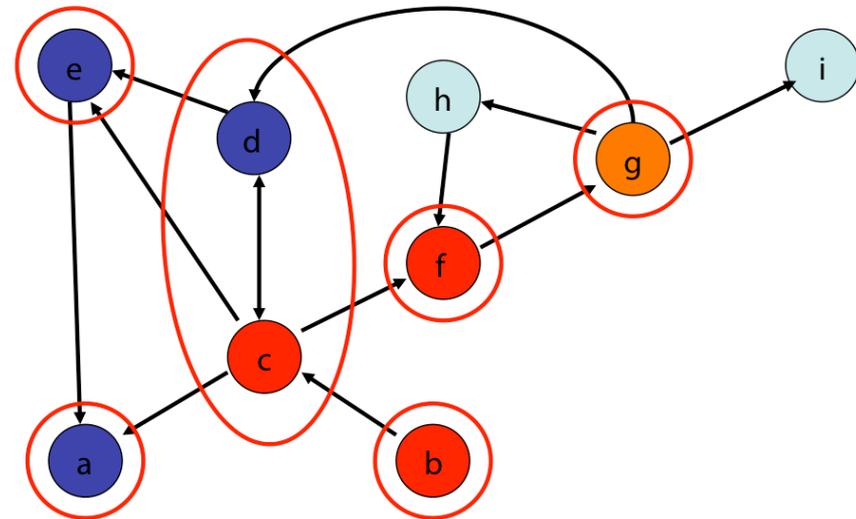
-  : unfertiger Knoten
- : fertiger Knoten
- Eine ZHK in G heißt **offen**, falls sie noch unfertige Knoten enthält. Sonst heißt sie (und ihre Knoten) **geschlossen**.
- **Repräsentant** einer ZHK: Knoten mit kleinster **dfsNum**.

Starke ZHKs

Beobachtungen (Invarianten):

1. Alle Kanten aus geschlossenen Knoten führen zu geschlossenen Knoten.
2. Der Pfad zum aktuellen Knoten enthält die Repräsentanten aller offenen ZHKs.
3. Betrachte die Knoten in offenen ZHKs sortiert nach DFS-Nummern:

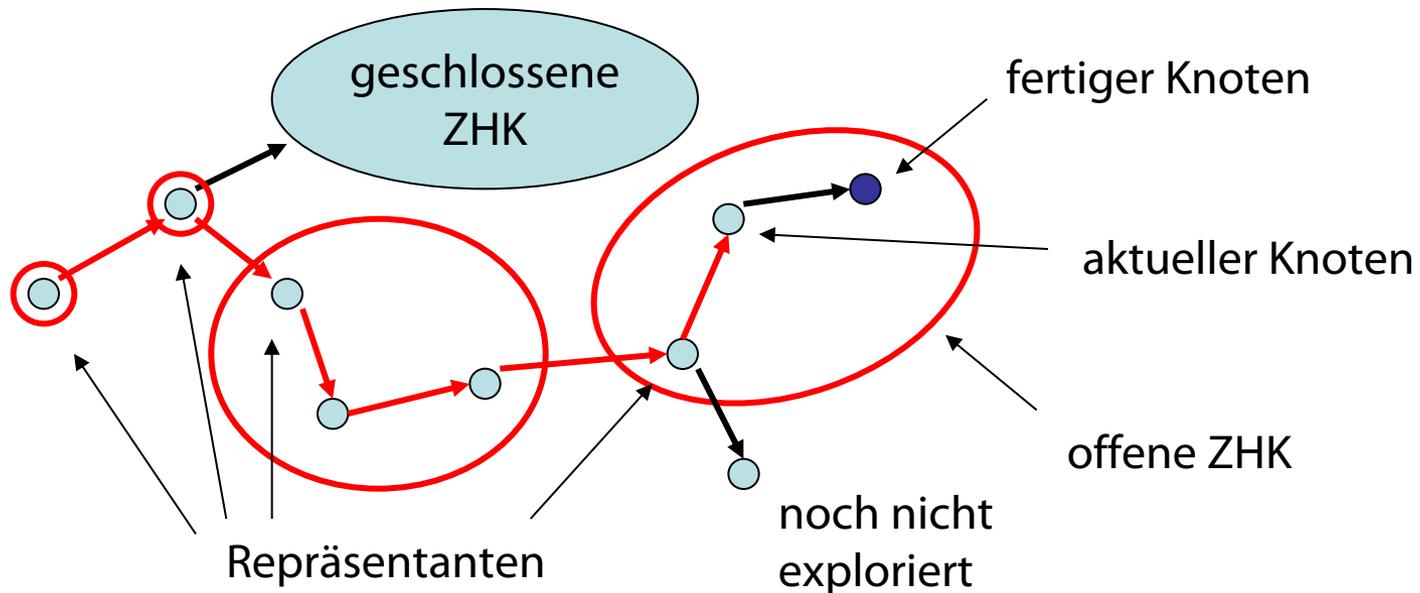
Die Repräsentanten partitionieren diese Folge in die offenen ZHKs.



Starke ZHKs

Beweis über vollständige Induktion.

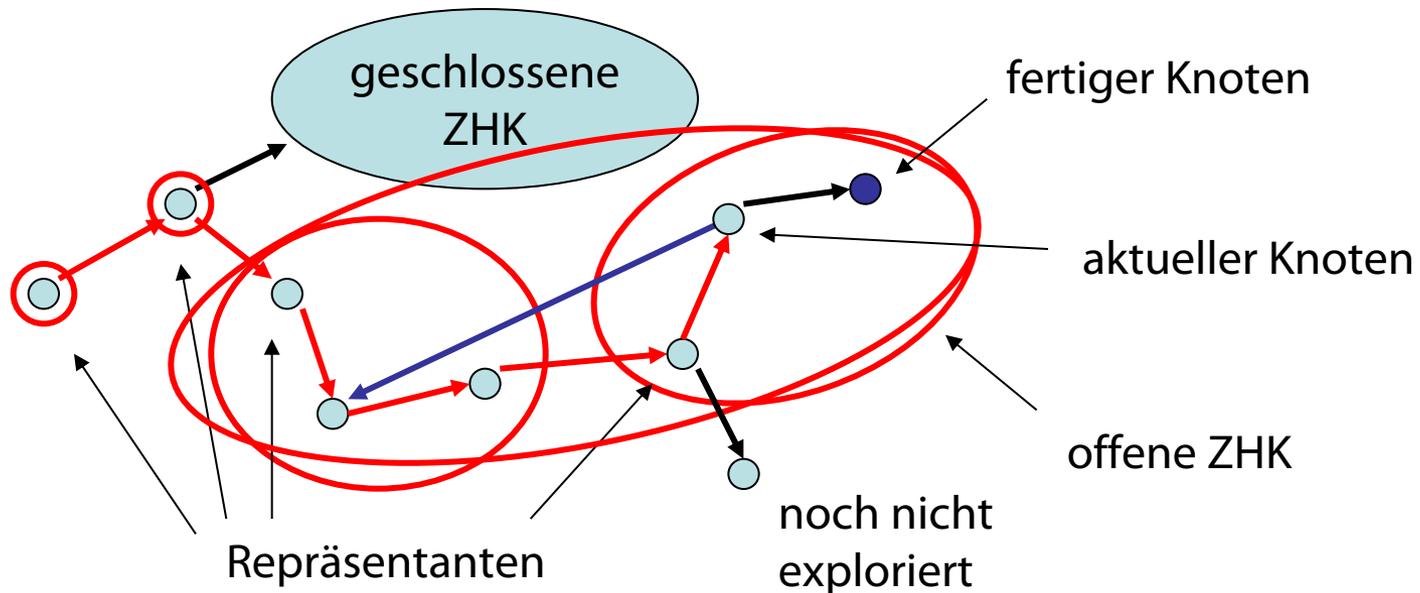
- Anfangs gelten alle Invarianten
- Wir betrachten verschiedene Fälle



Starke ZHKs

Beweis über vollständige Induktion.

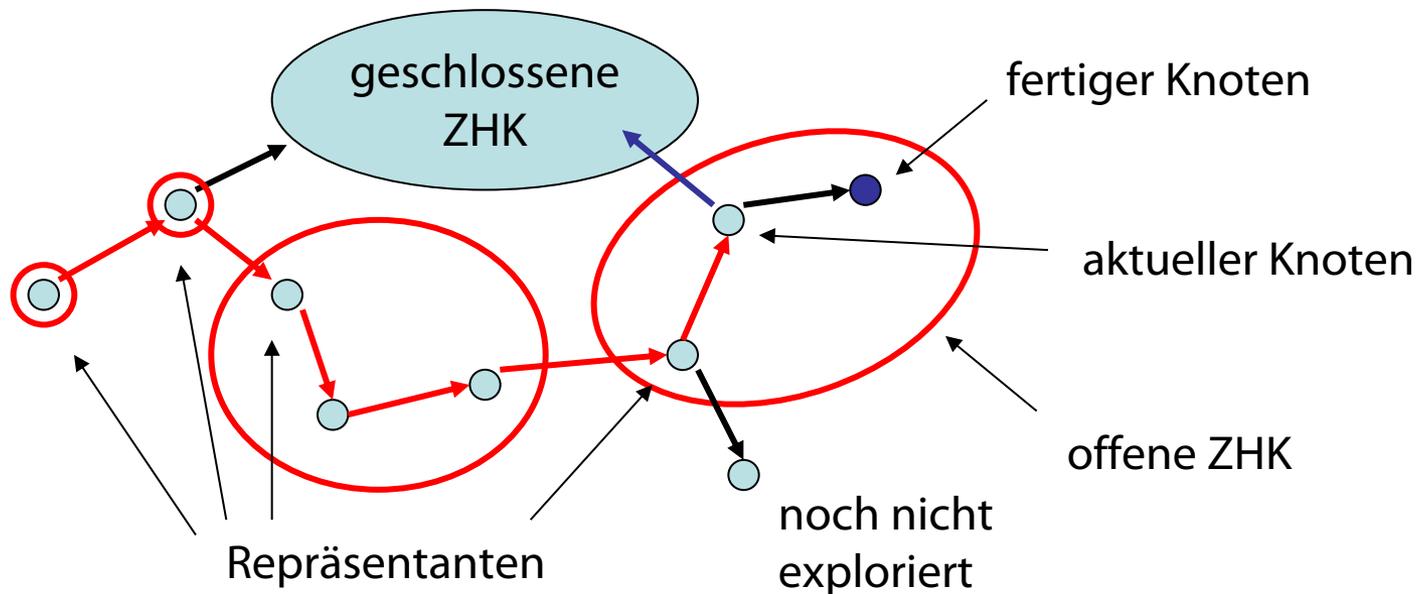
- Anfangs gelten alle Invarianten
- Fall 1: Kante zu unfertigem Knoten



Starke ZHKs

Beweis über vollständige Induktion.

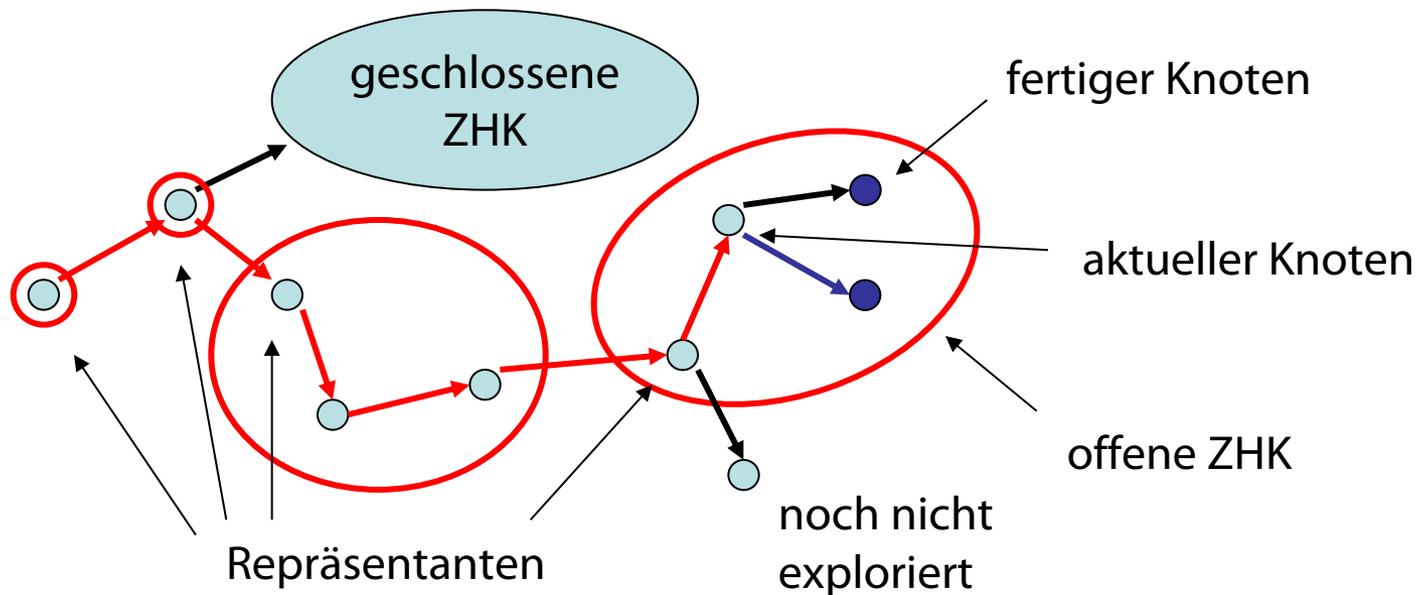
- Anfangs gelten alle Invarianten
- Fall 2: Kante zu geschlossenem Knoten



Starke ZHKs

Beweis über vollständige Induktion.

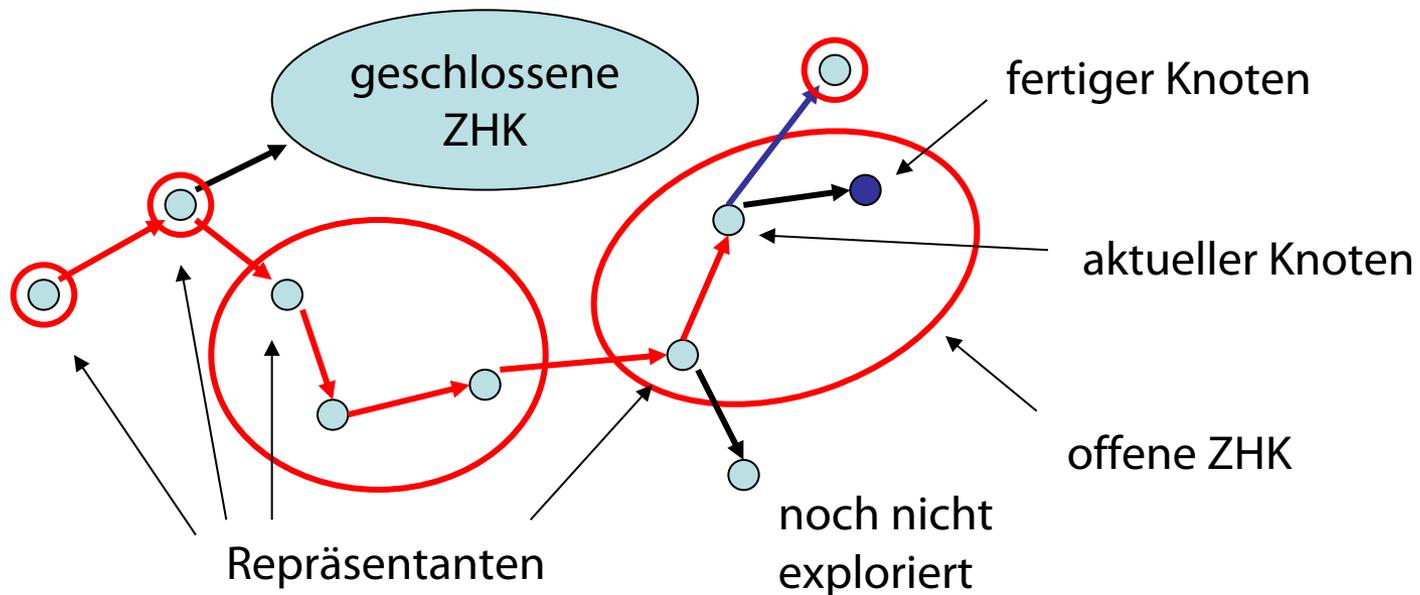
- Anfangs gelten alle Invarianten
- Fall 3: Kante zu fertigem Knoten



Starke ZHKs

Beweis über vollständige Induktion.

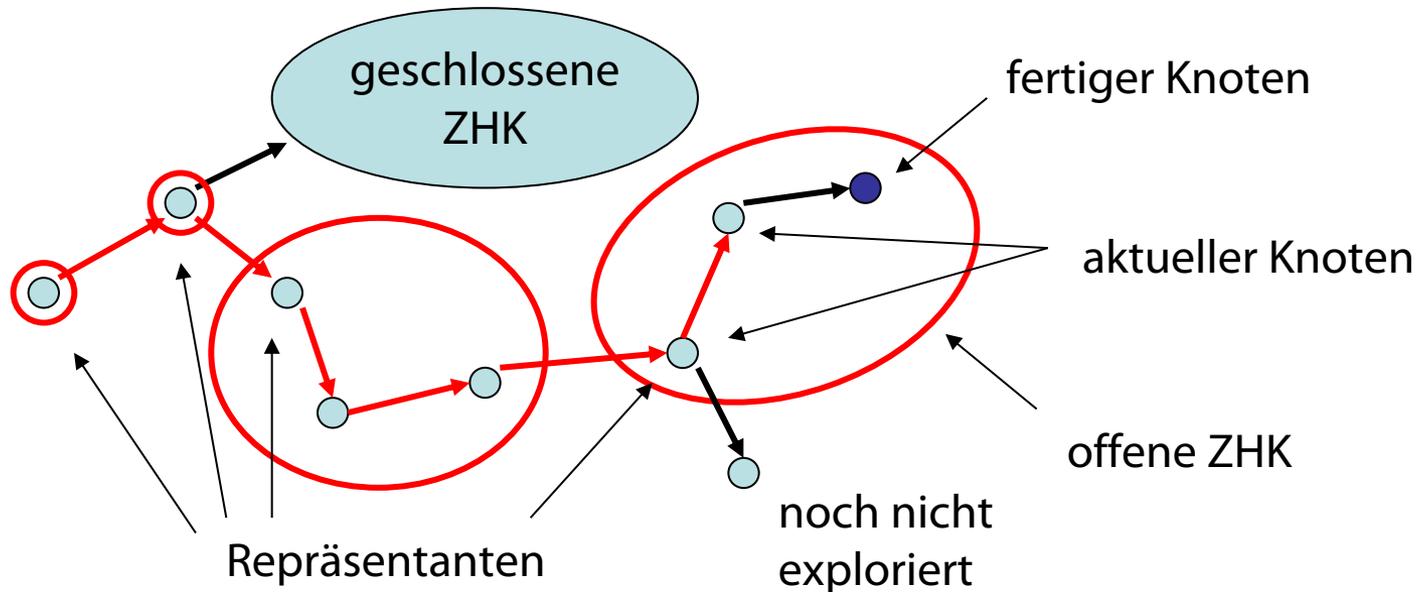
- Anfangs gelten alle Invarianten
- Fall 4: Kante zu nicht exploriertem Knoten



Starke ZHKs

Beweis über vollständige Induktion.

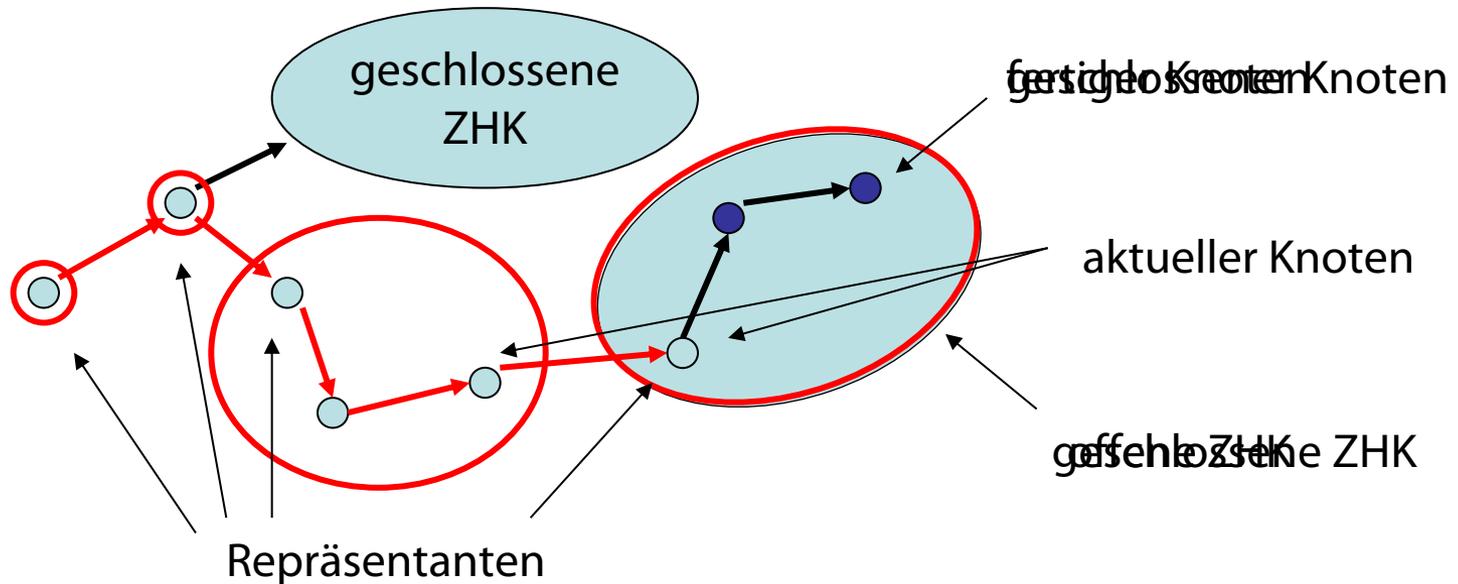
- Anfangs gelten alle Invarianten
- Fall 5: Knoten exploriert



Starke ZHKs

Beweis über vollständige Induktion.

- Anfangs gelten alle Invarianten
- Fall 5: Knoten exploriert



Starke ZHKs

Behauptung: Eine *geschlossene* ZHK G_c im besuchten Teilgraphen C von G ist eine ZHK in G (die geschlossene ZHK G_c ist maximal)

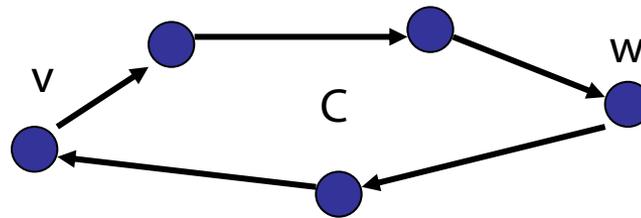
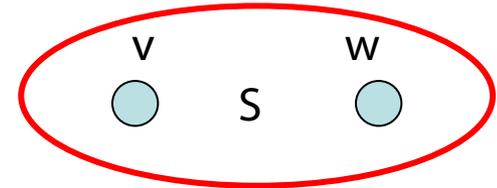
Überlegung:

- v : geschlossener Knoten
- S : ZHK in G , die v enthält
- S_c : ZHK in G_c , die v enthält
- Es gilt: $S_c \subseteq S$ (v erreicht jeden Knoten in der ZHK)
- Zu zeigen (Maximalität): $S \subseteq S_c$

Starke ZHKs

Begründung für $S \subseteq S_c$:

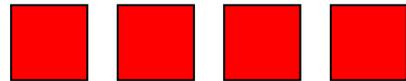
- w : beliebiger Knoten in S
- Es gibt gerichteten Kreis C durch v und w
- Nutze **Invariante 1**: alle Knoten in C geschlossen



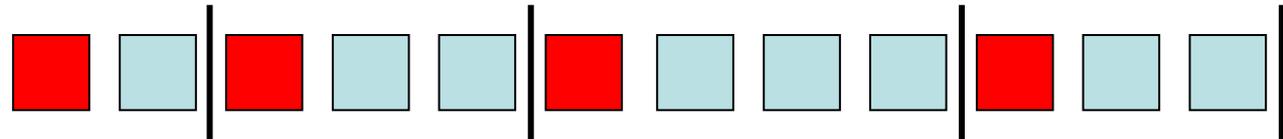
- Da alle Kanten geschlossener Knoten exploriert worden sind, ist C in G_c und daher $w \in S_c$

Invarianten 2 und 3

- Der Pfad zum aktuellen Knoten enthält die Repräsentanten aller offenen ZHKs (**oReps**)



- Betrachte die Knoten in offenen ZHKs sortiert nach DFS-Nummern. Die Repräsentanten partitionieren diese Folge in die offenen ZHKs (**oNodes**)



Stack ausreichend für beide Folgen
(**oNodes** sei allerdings ein Stack mit Element-Test)

Wiederholung: Tiefensuche-Schema

```
function dfs(g)
  for v in nodes(g)    v.marked = false end
  init()
  for s in nodes(g)
    if !s.marked
      s.marked = true
      root(s)
      dfs(s, s, g)
    end
  end
end
```

```
function dfs(u, v, g)
  for (_, w) in edges_out(v, g)
    if w.marked    handle_non_tree_edge(v, w)
    else    traverse_tree_edge(v, w)
      w.marked = true
      dfs(v, w, g)
    end
  end
  backtrack(u, v)
end
```

Prozeduren in rot: noch zu spezifizieren

Starke ZHKs

- **init()**

```
oReps = make_stack()  
oNodes = make_stack()  
dfsPos = 1
```

- **root(s)**

```
traverse_tree_edge(nothing, s)
```

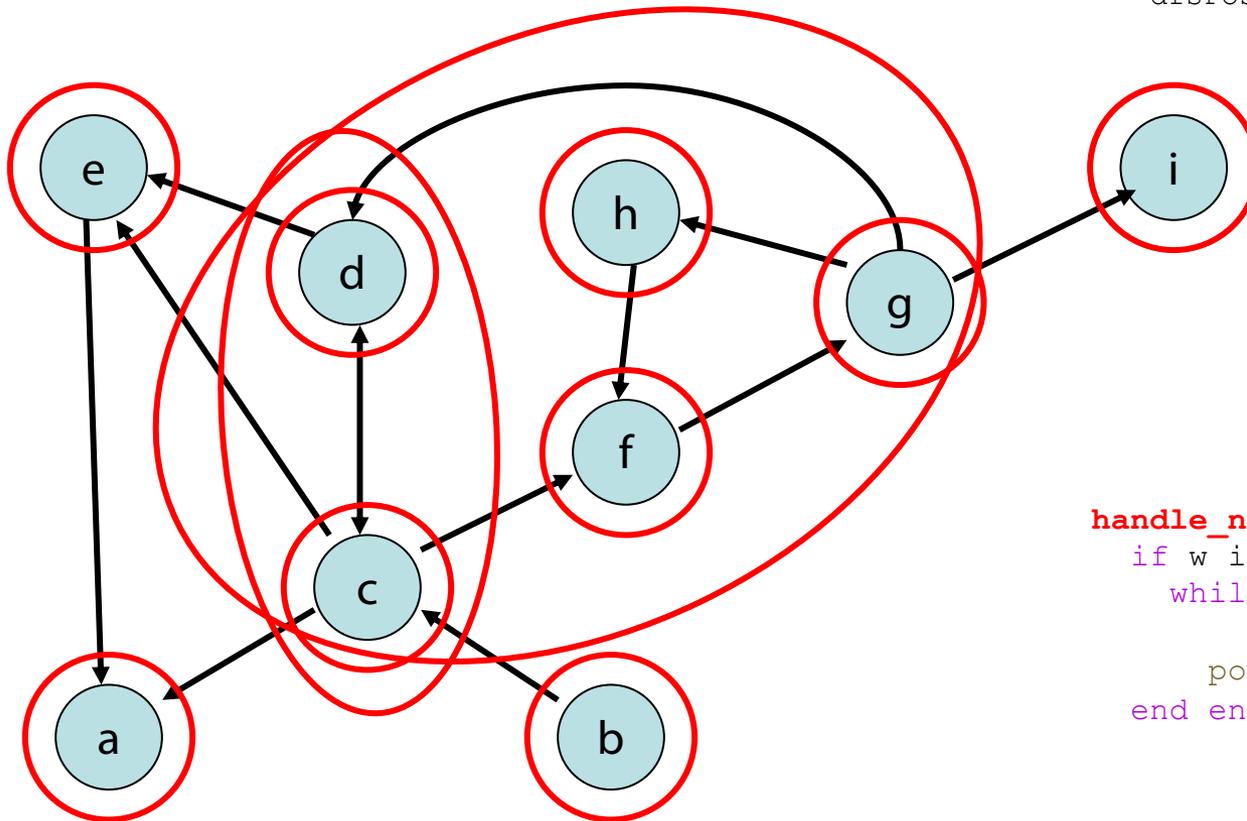
- **traverse_tree_edge(v, w)**

```
push(w, oReps) # neue ZHK  
push(w, oNodes) # neuer offener Knoten  
w.dfsNum = dfsPos  
dfsPos = dfsPos + 1
```

Starke ZHKs - Beispiel

```

traverse_tree_edge(v, w)
  push(w, oReps) # neue ZHK
  push(w, oNodes) # neuer offener Knoten
  w.dfsNum = dfsPos
  dfsPos = dfsPos + 1
  
```

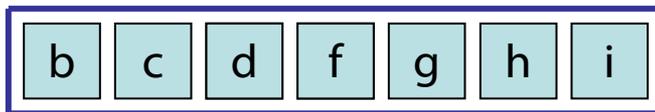


```

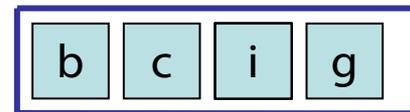
backtrack(u, v)
  if v == top(oReps)
    pop(oReps)
    while true
      w = pop(oNodes)
      w.component = v
      if w == v break end
    end end
  
```

```

handle_non_tree_edge(v, w)
  if w in oNodes
    while w.dfsNum <
      top(oReps).dfsNum
      pop(oReps)
    end end
  
```



oNodes



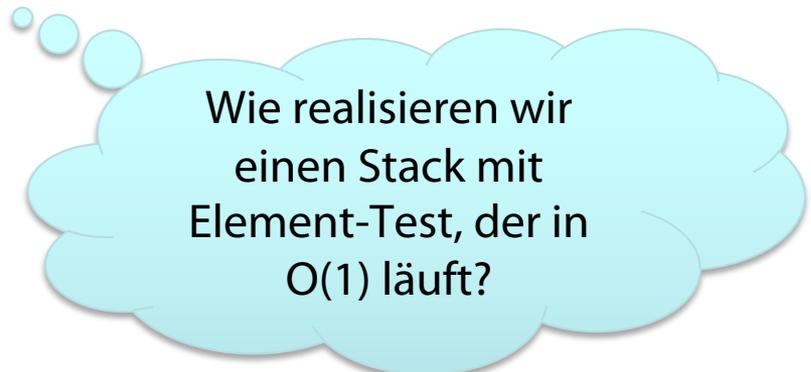
oReps

Starke ZHKs

Behauptung: Der DFS-basierte Algorithmus für starke ZHKs benötigt $O(n+m)$ Zeit.

Beweis:

- `init`, `root`, `traverse_tree_edge`: Zeit $O(1)$
- `backtrack`, `handle_non_tree_edge`: da jeder Knoten nur höchstens einmal in `oReps` und `oNodes` landet, insgesamt Zeit $O(n)$
- DFS-Gerüst: Zeit $O(n+m)$



Kürzeste Wege

Zentrale Frage:

Wie komme ich am schnellsten von A nach B in einem Graphen, in dem Kanten Kosten zugeordnet werden?

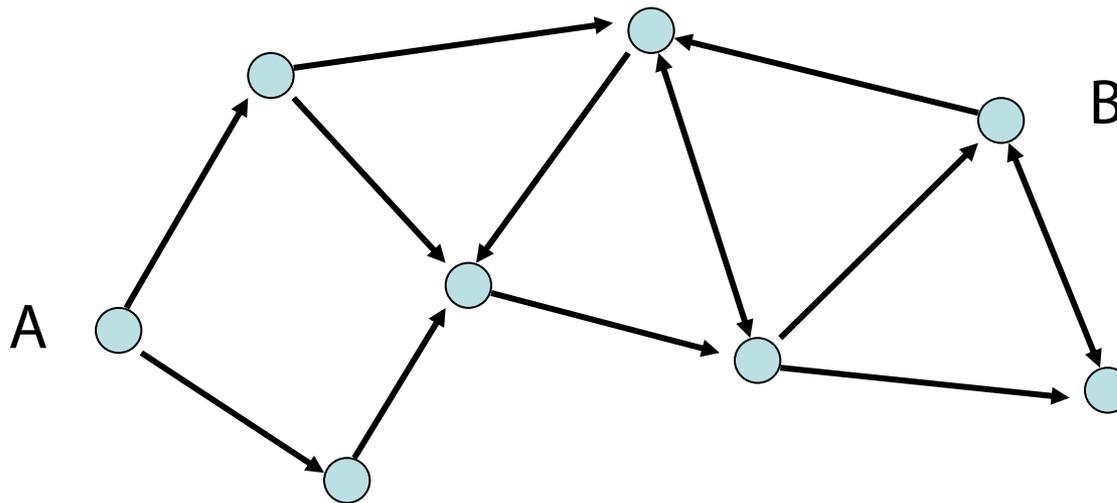
Fälle:

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

Kürzeste Wege

Zentrale Frage:

Wie komme ich am schnellsten von A nach B?

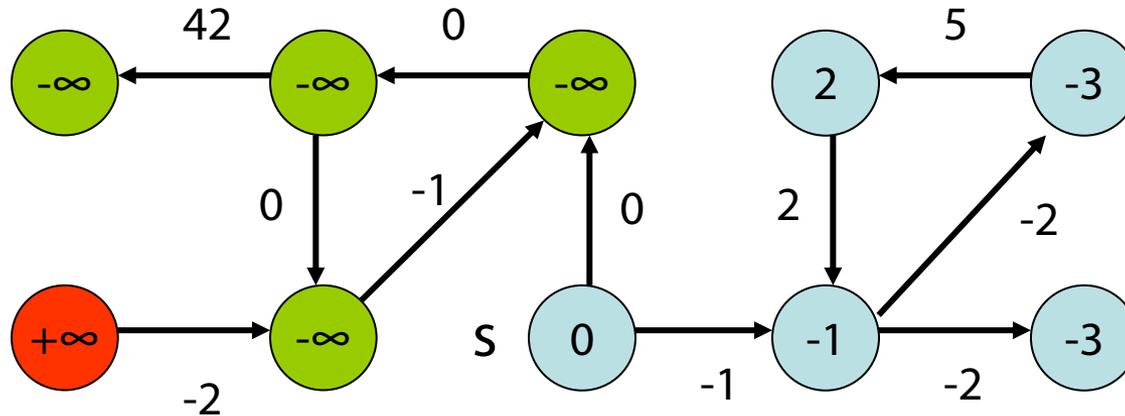


Kürzeste Wege

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

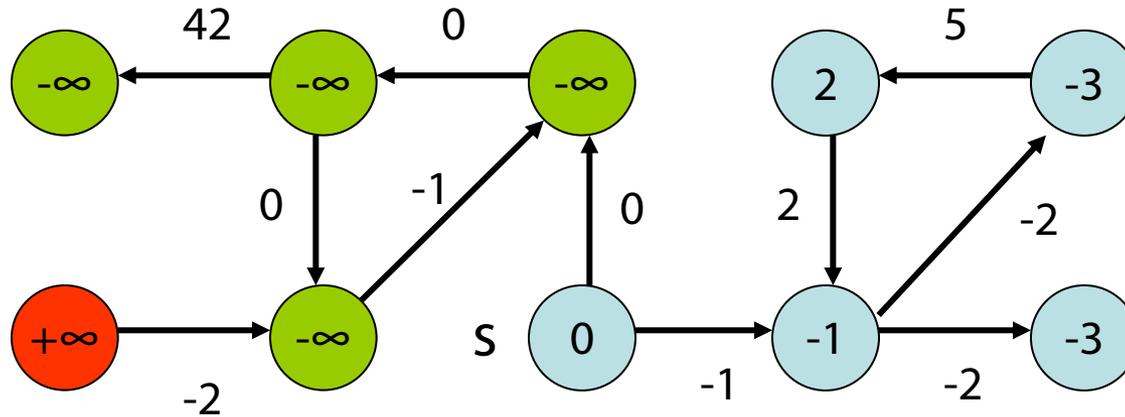
Kürzeste Wege



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

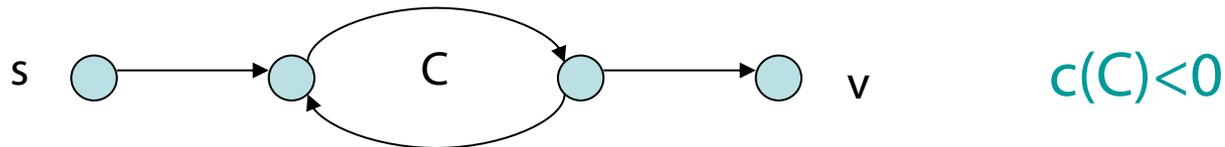
$$\mu(s,v) = \left\{ \begin{array}{l} \infty \quad \text{kein Weg von } s \text{ nach } v \\ -\infty \quad \text{Weg bel. kleiner Kosten von } s \text{ nach } v \\ \min\{ c(p) \mid p \text{ ist Weg von } s \text{ nach } v \} \end{array} \right.$$

Kürzeste Wege



Wann sind die Kosten $-\infty$?

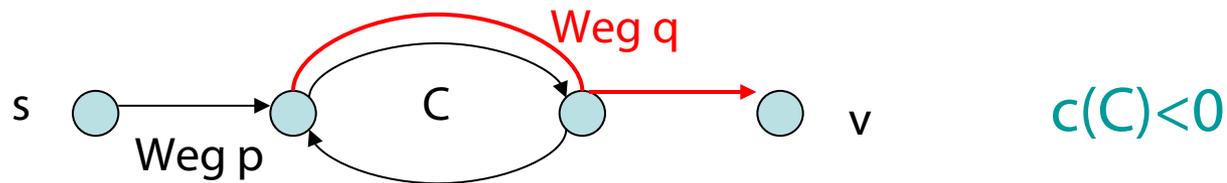
Wenn es einen negativen Kreis gibt:



Kürzeste Wege

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

Negativer Kreis hinreichend:



Kosten für i -fachen Durchlauf von C :

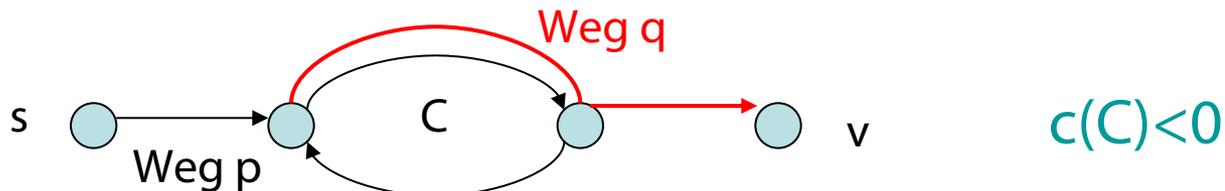
$$c(p) + i \cdot c(C) + c(q)$$

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

Kürzeste Wege

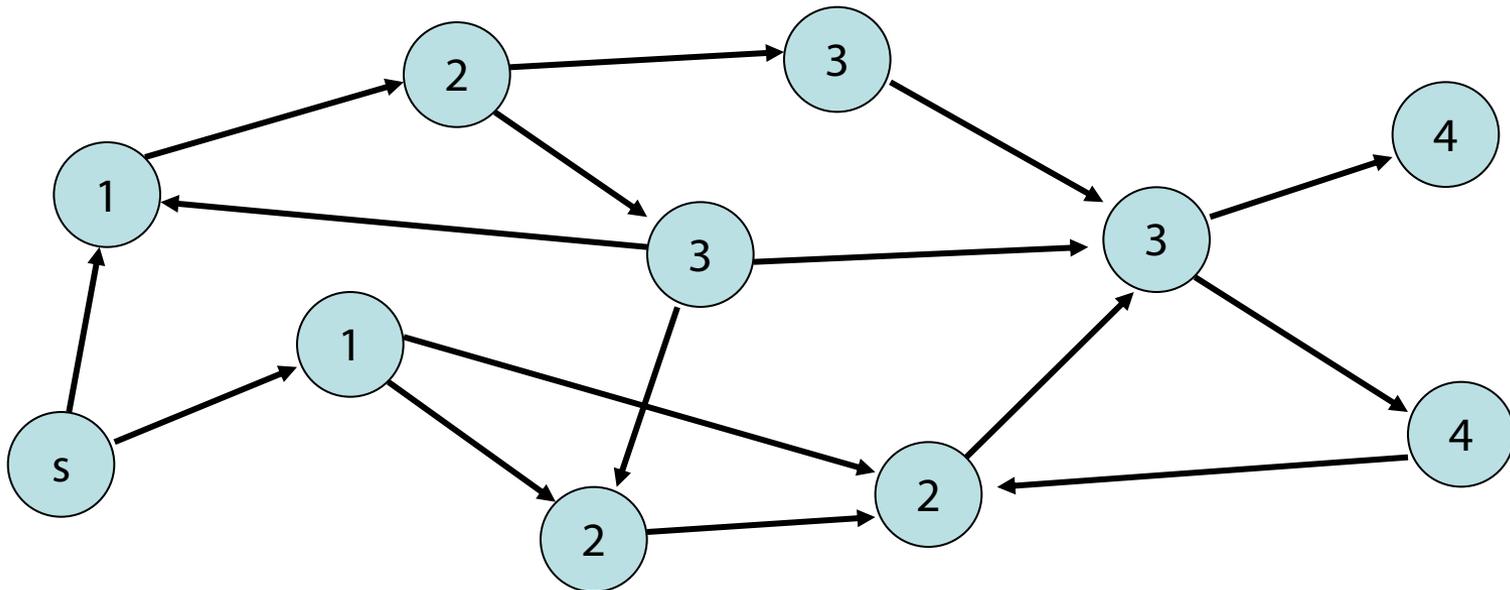
Negativer Kreis notwendig:

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



Kürzeste Wege in Graphen

Graph mit Kantenkosten 1:
Führe Breitensuche durch.

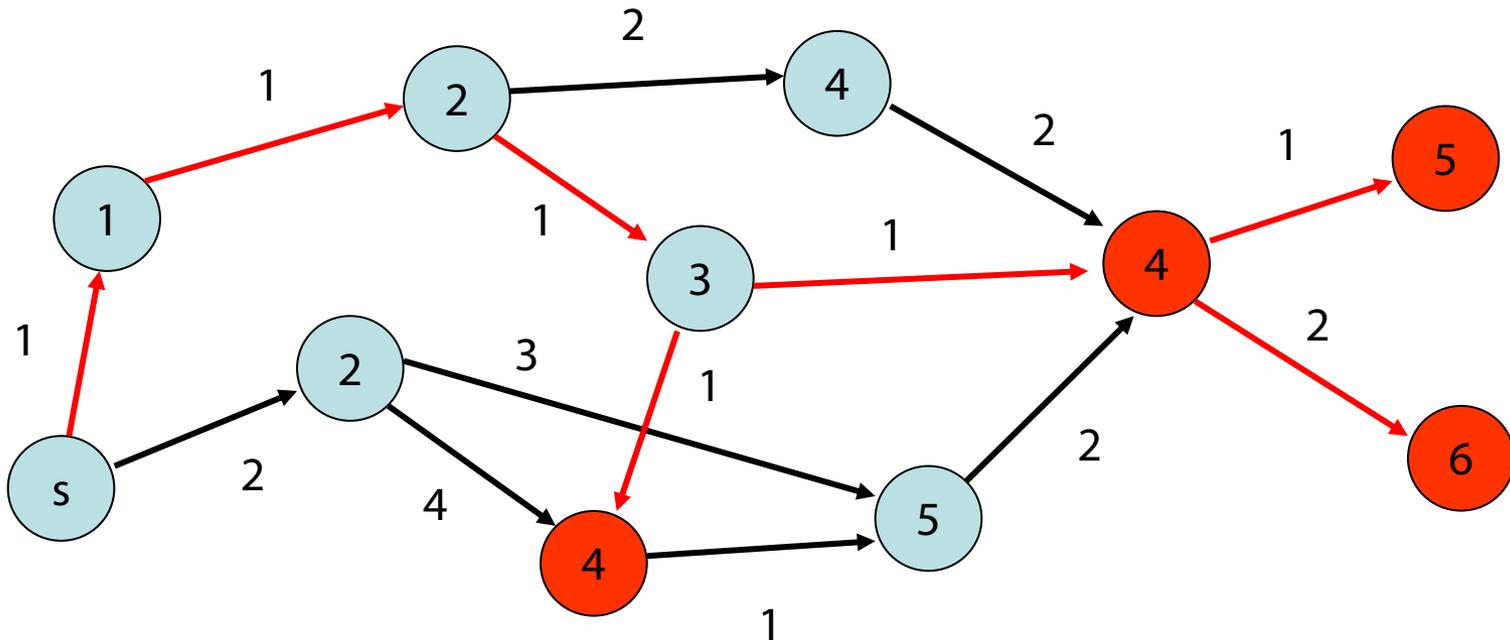


Kürzeste Wege in DAGs

- DAGs sind Graphen, also Breitensuche bei Kantenkosten 1
- Reine Breitensuche funktioniert schon bei DAGs nicht, wenn Kantenkosten nicht gleich 1.

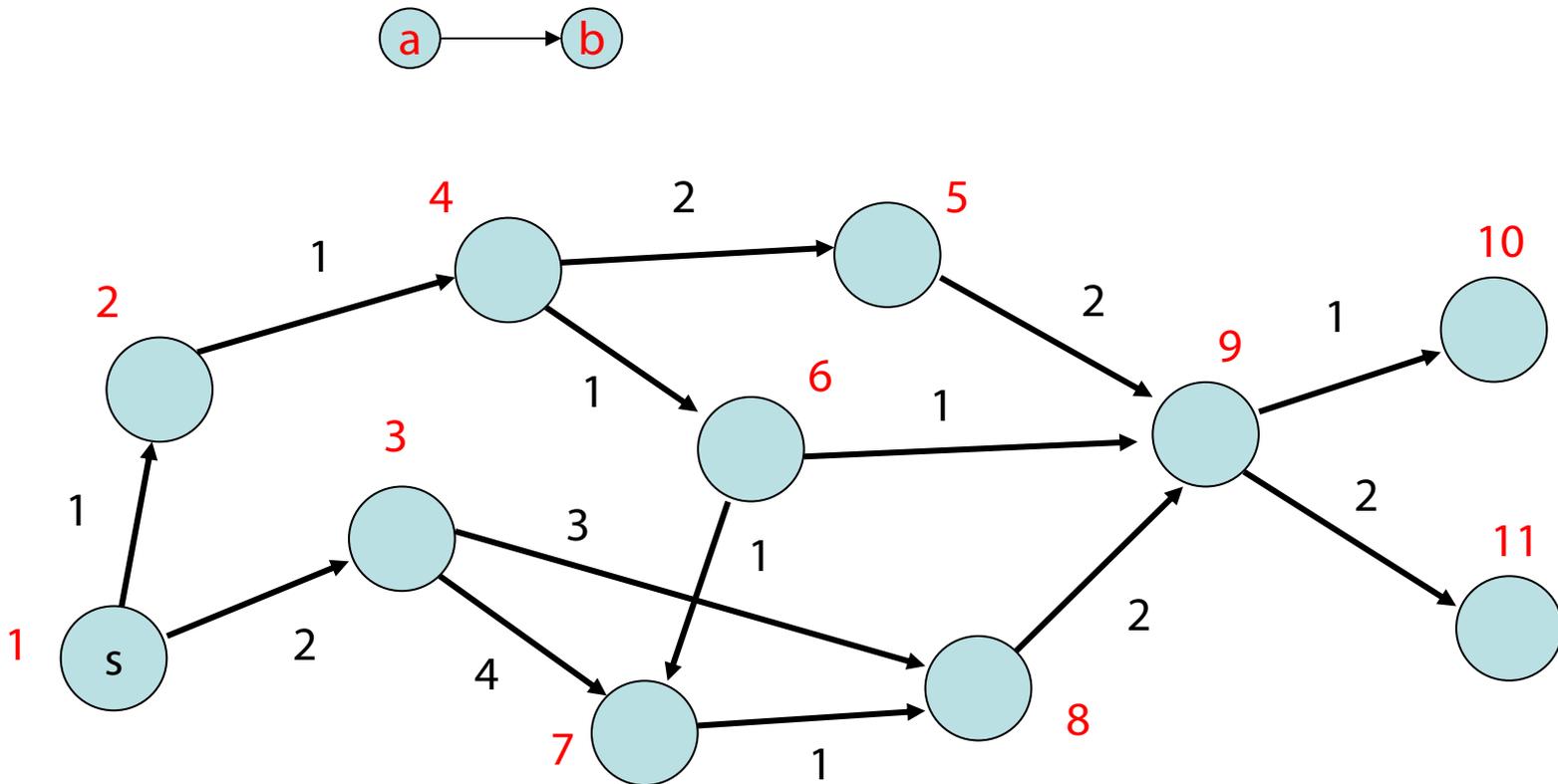
Kürzeste Wege in DAGs

Korrekte Distanzen:



Kürzeste Wege in DAGs

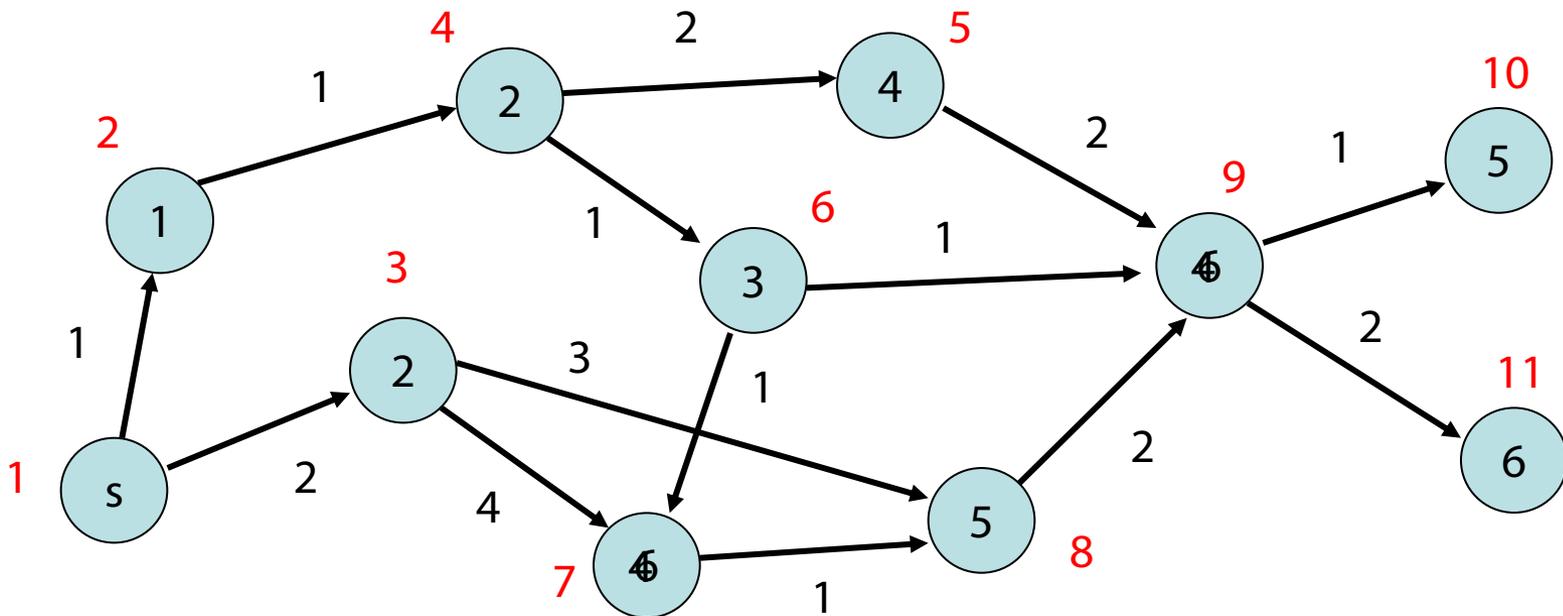
Strategie: nutze aus, dass Knoten in DAGs topologisch sortiert werden können (alle Kanten erfüllen $a < b$)



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

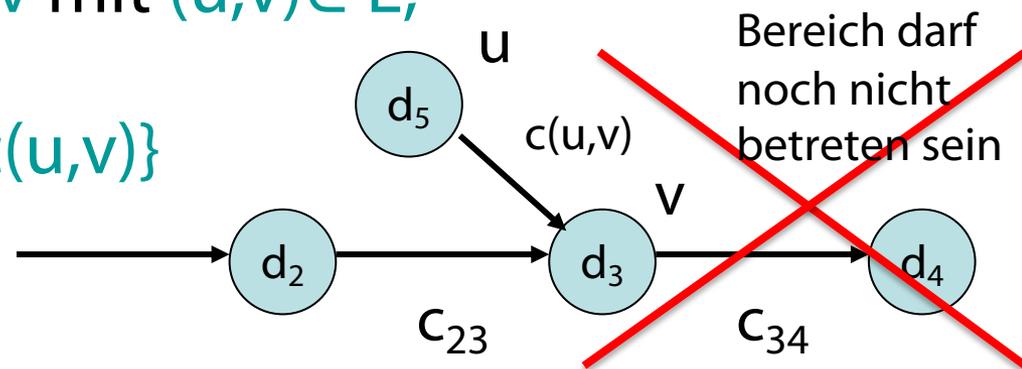
Warum funktioniert das??

Kürzeste Wege in Graphen

Allgemeine Strategie:

- Am Anfang, setze $d(s) = 0$ und $d(v) = \infty$ für alle Knoten $v \in V \setminus \{s\}$
- Für jeden besuchten Knoten u , aktualisiere die Distanzen der Knoten v mit $(u,v) \in E$,

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



- Besuche Knoten in einer Reihenfolge, die **sicherstellt**, dass **mindestens ein** kürzester Weg von s zu jedem v gefunden ist, bevor v expandiert wird

Kürzeste Wege in DAGs

Es gilt:

Expansion in topologischer Reihenfolge
führt zu richtigen Distanzen

Zurück zur Strategie:

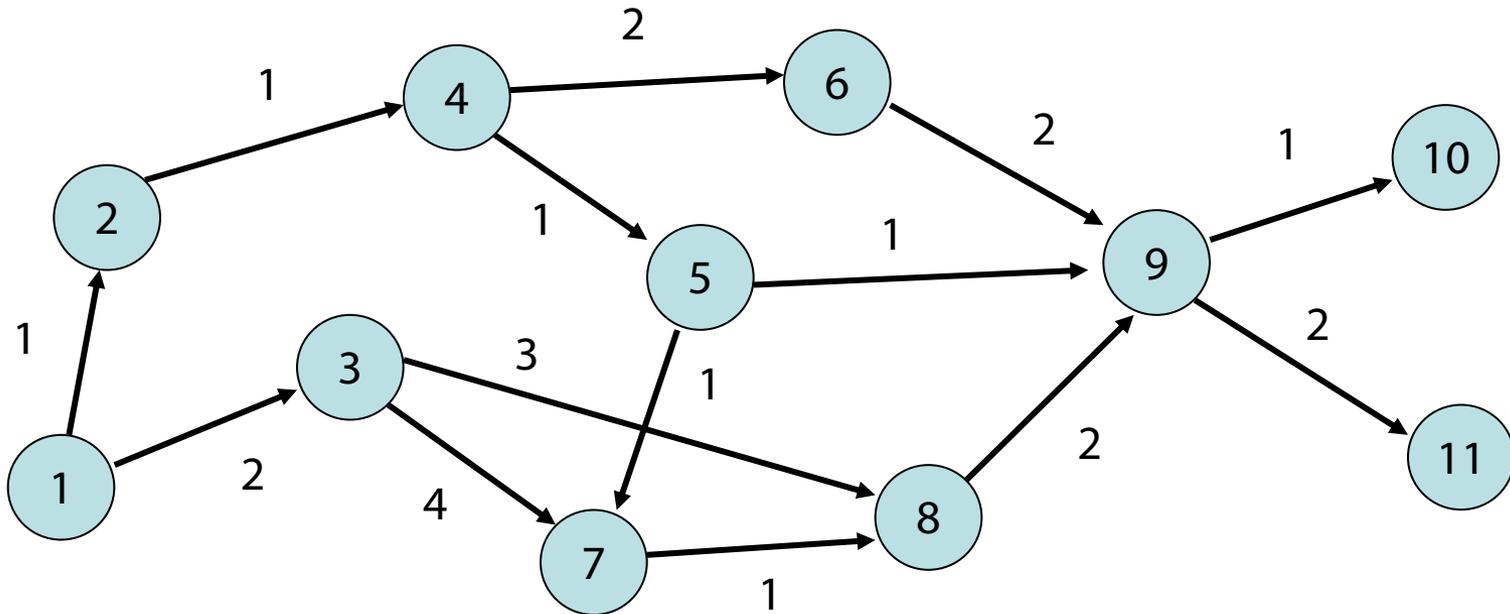
1. Topologische Sortierung der Knoten
2. Aktualisierung der Distanzen gemäß der topologischen Sortierung

Wie führe ich eine topologische
Sortierung durch?

Kürzeste Wege in DAGs

Beispiel:

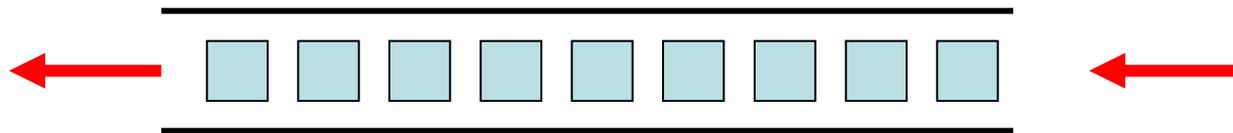
- : Knoten momentan in Queue q
- Nummerierung nach Einfügereihenfolge



Kürzeste Wege in DAGs

Topologische Sortierung:

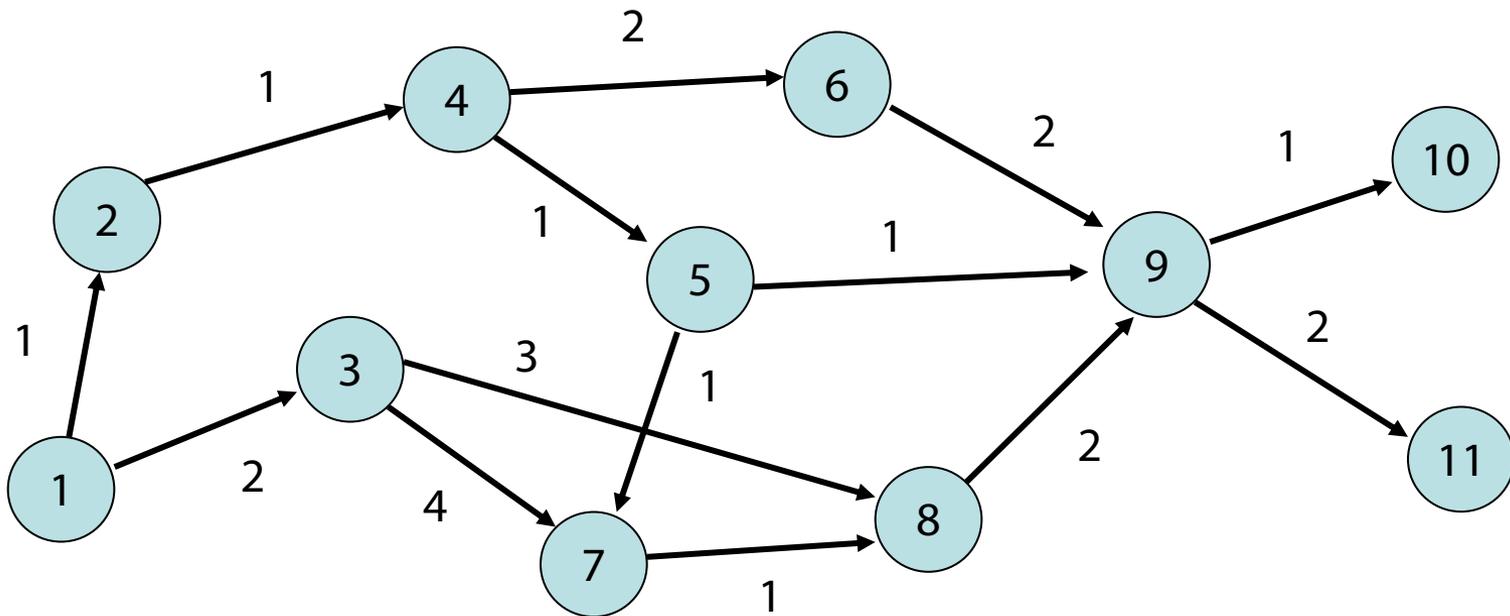
- Verwende eine FIFO Queue q und Zähler $n = 1$



- Bei Einfügen von v in q : $\text{num}(v) = n$; $n = n + 1$
- Anfangs enthält q alle Knoten, die **keine** eingehende Kante haben (Quellen).
- Solange q nicht leer
 - Entnehme u aus q und markiere alle $(u, v) \in E$.
Falls alle Kanten nach v markiert, füge v in q ein.
- Erfolg, falls alle Knoten nummeriert

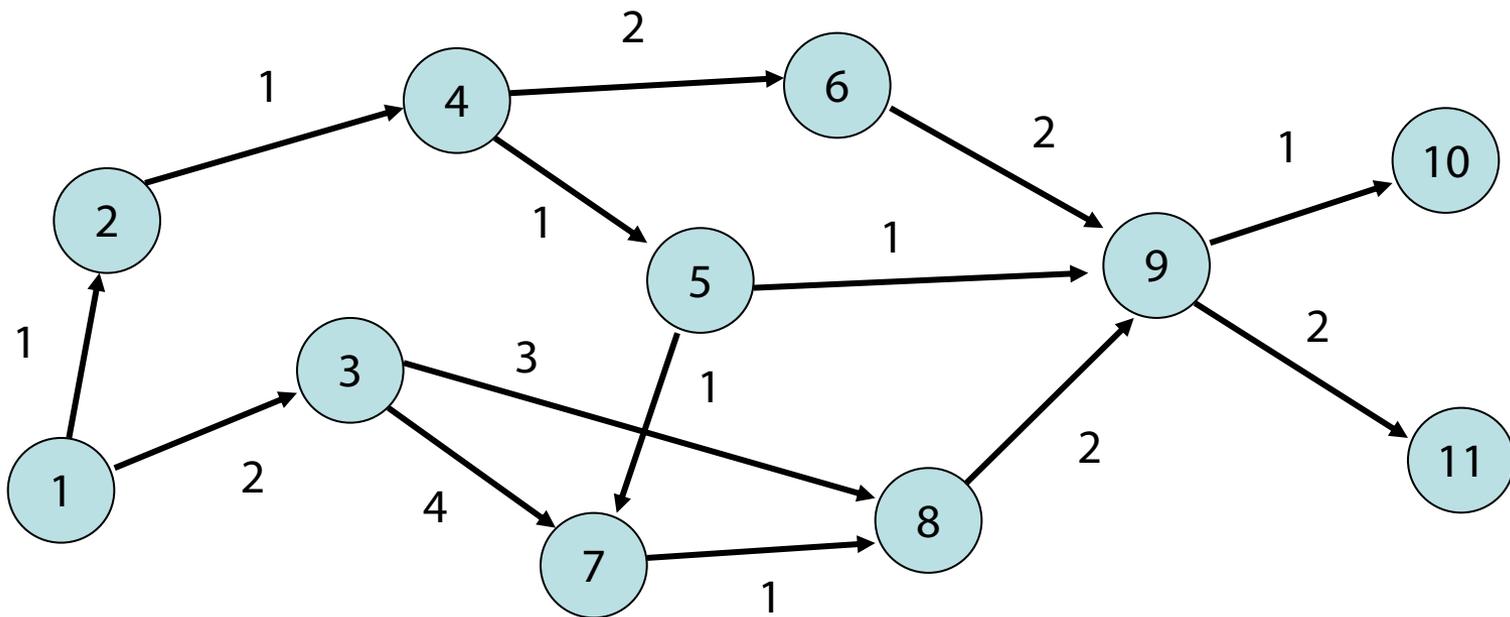
Kürzeste Wege in DAGs

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



Kürzeste Wege in DAGs

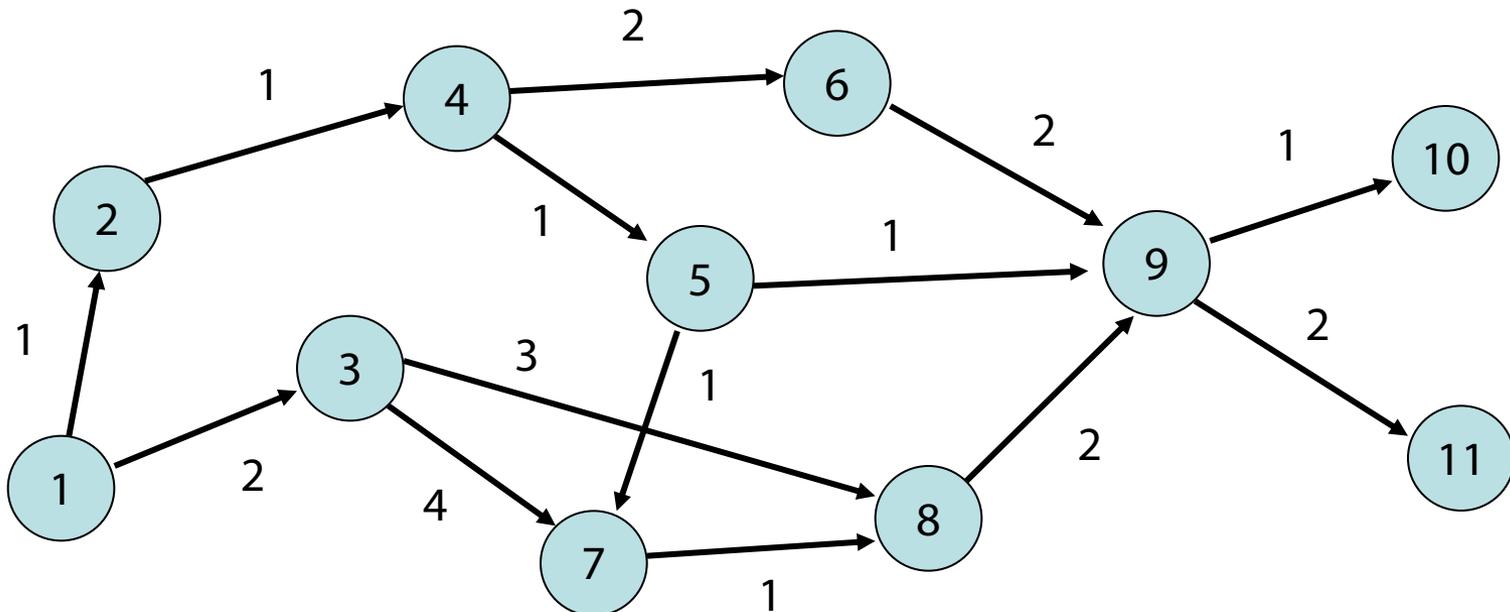
Laufzeit: Zur Bestimmung aller Knoten ohne eingehende Kante muss Graph einmal durchlaufen werden. Danach wird jeder Knoten und jede Kante genau einmal betrachtet, also Zeit $O(n+m)$.



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 $O(n+m)$

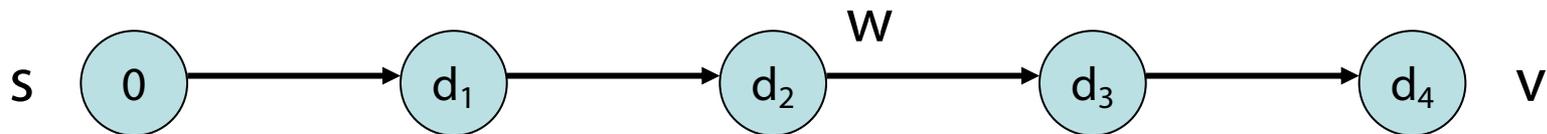
Insgesamt Laufzeit $O(n+m)$.

Beispielimplementierung
in Julia vorhanden.

Dijkstras Algorithmus

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

Problem: besuche Knoten eines kürzesten Weges in richtiger Reihenfolge



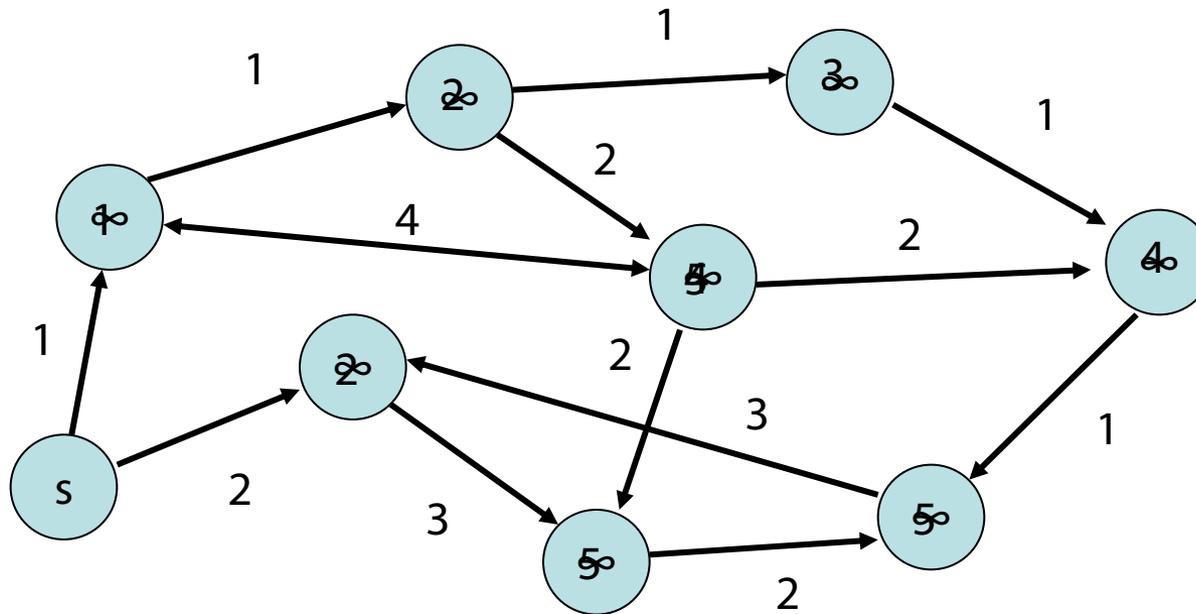
Lösung: besuche Knoten in der Reihenfolge der kürzesten Distanz zur Quelle **s**

Dijkstras Algorithmus

- Am Anfang, setze $s.d = 0$ und $v.d = \infty$ für alle Knoten. Füge s in **Prioritätswarteschlange** q ein, wobei die Prioritäten in q gemäß der aktuellen Distanzen $v.d$ von s zu v definiert sind.
- Wiederhole, bis q leer:
 - Entferne mittels **delete_min**(q) aus q den Knoten u mit niedrigstem $u.d$
 - Für alle $(u, v) \in E$, setze $v.d := \min\{v.d, u.d + c(u, v)\}$.
 - Falls v noch nicht in q war,
 - dann füge v in q ein,
 - sonst verringere Kosten von v mit **decrease_key**(v, q, Δ), sofern $\Delta < 0$

Dijkstras Algorithmus

Beispiel: (● : aktuell, ● : fertig)



Dijkstras Algorithmus

```
function dijkstra(s, g)
  s.d = 0; s.parent = s
  q = build_pq([s], (x) -> x.d, # Distanz ist key() in PQ,
    (x, new_k) -> x.d = new_k) # set_key() für die PQ
  while !empty_pq(q)
    u = delete_min(q) # u: min. Distanz zu s in q
    for (_, v, cost) in edges_out(u, g)
      dv = v.d
      if isinf(dv) # v schon in q?
        v.d = u.d + cost; v.parent = u; insert(v, q)
      elseif dv > u.d + cost # aktualisiere v.d
        v.parent = u; decrease_key(v, q, dv - (u.d + cost))
      end
    end
  end
end
end
end
```

Dijkstras Algorithmus

Laufzeit:

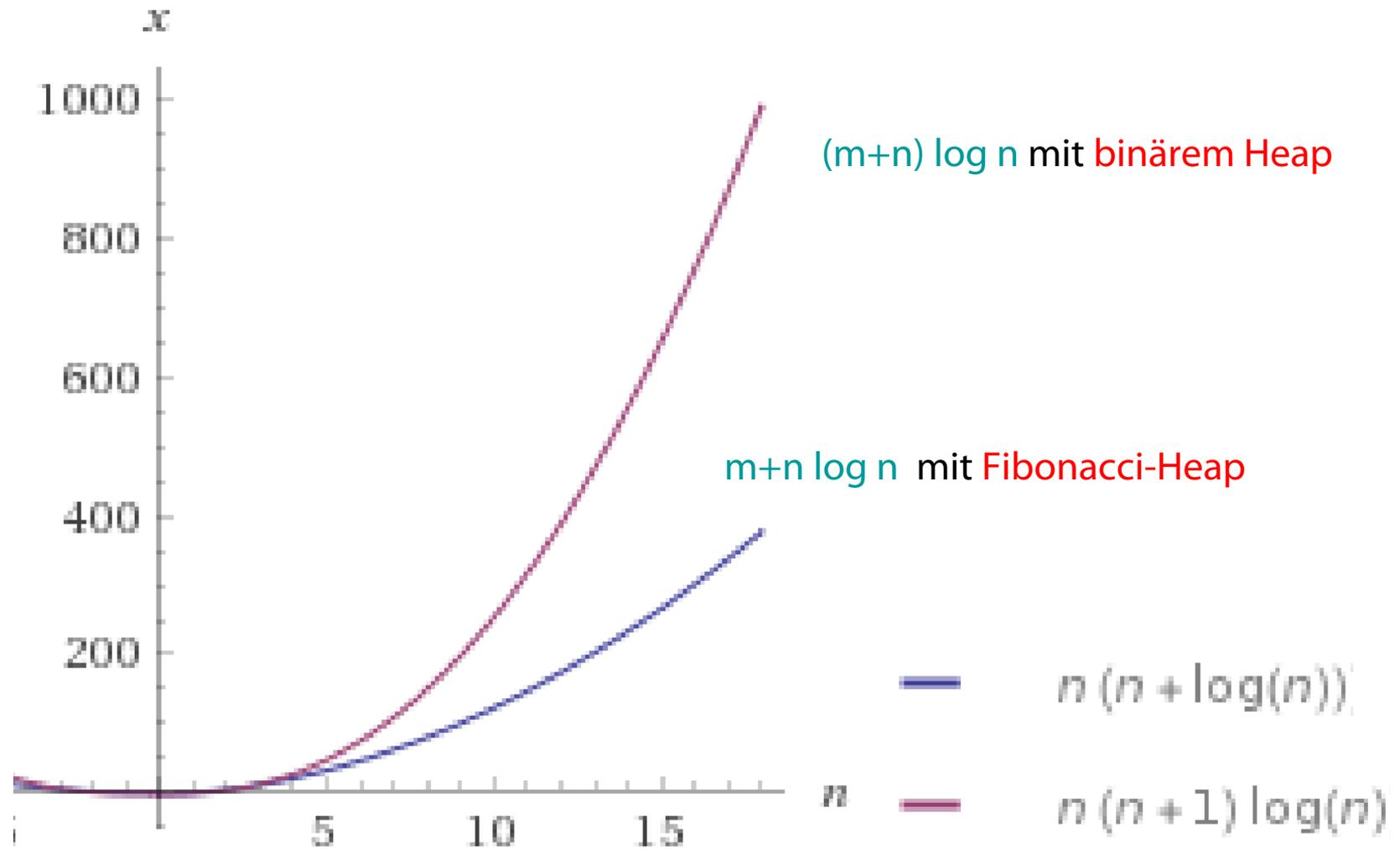
$$T_{\text{Dijkstra}} \in O(n(T_{\text{delete_min}}(n) + T_{\text{insert}}(n)) + m \cdot T_{\text{decrease_key}}(n))$$

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

Fibonacci Heap:

- $T_{\text{delete_min}}(n) = T_{\text{insert}}(n) \in O(\log n)$
- $T_{\text{decrease_key}}(n) \in O(1)$
- Damit $T_{\text{Dijkstra}} \in O(n \log n + m)$

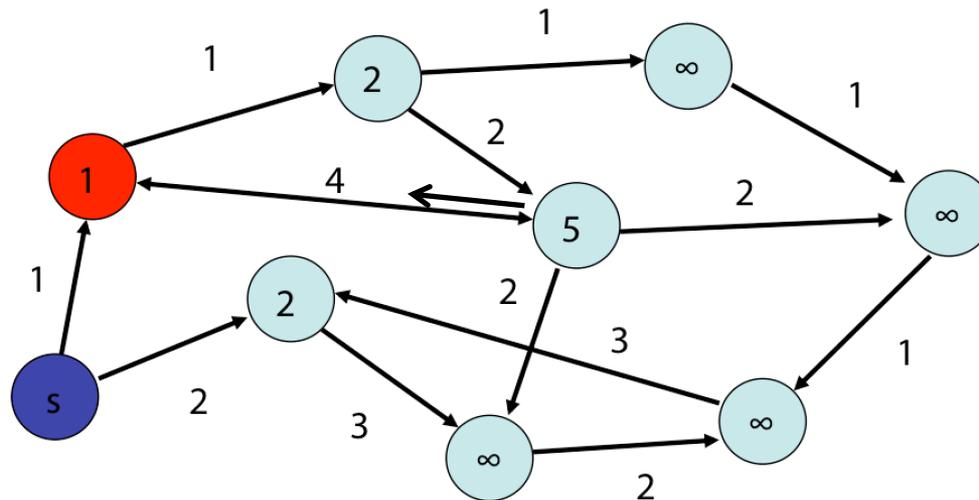
Dijkstra-Algorithmus: Vergleich mit $m = n^2$



Kürzeste Wege

- Nachteil der bisherigen Verfahren:
 - Nur Länge des kürzesten Weges bestimmt
 - Zur Wegebestimmung muss Rückzeiger verwaltet werden

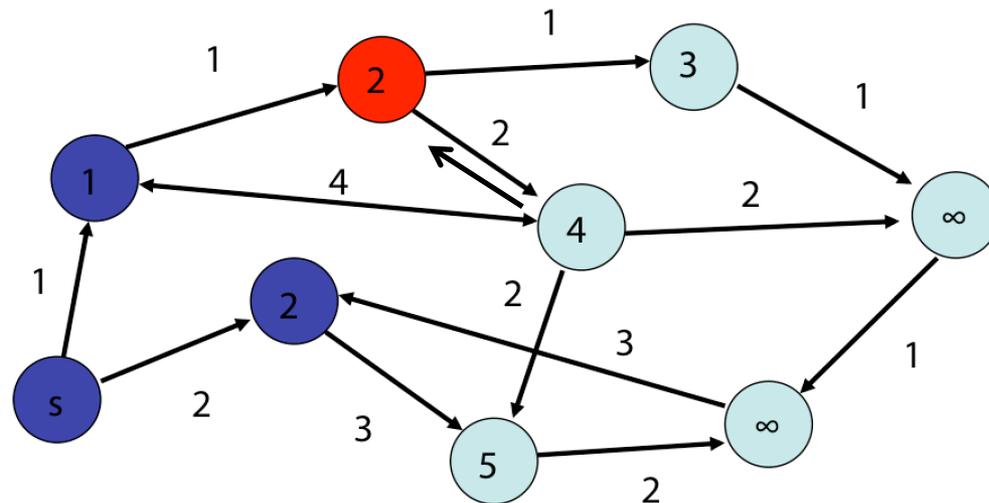
Beispiel: (● : aktuell, ● : fertig)



Kürzeste Wege

- Nachteil der bisherigen Verfahren:
 - Nur Länge des kürzesten Weges bestimmt
 - Zur Wegebestimmung muss Rückzeiger verwaltet werden

Beispiel: (● : aktuell, ● : fertig)



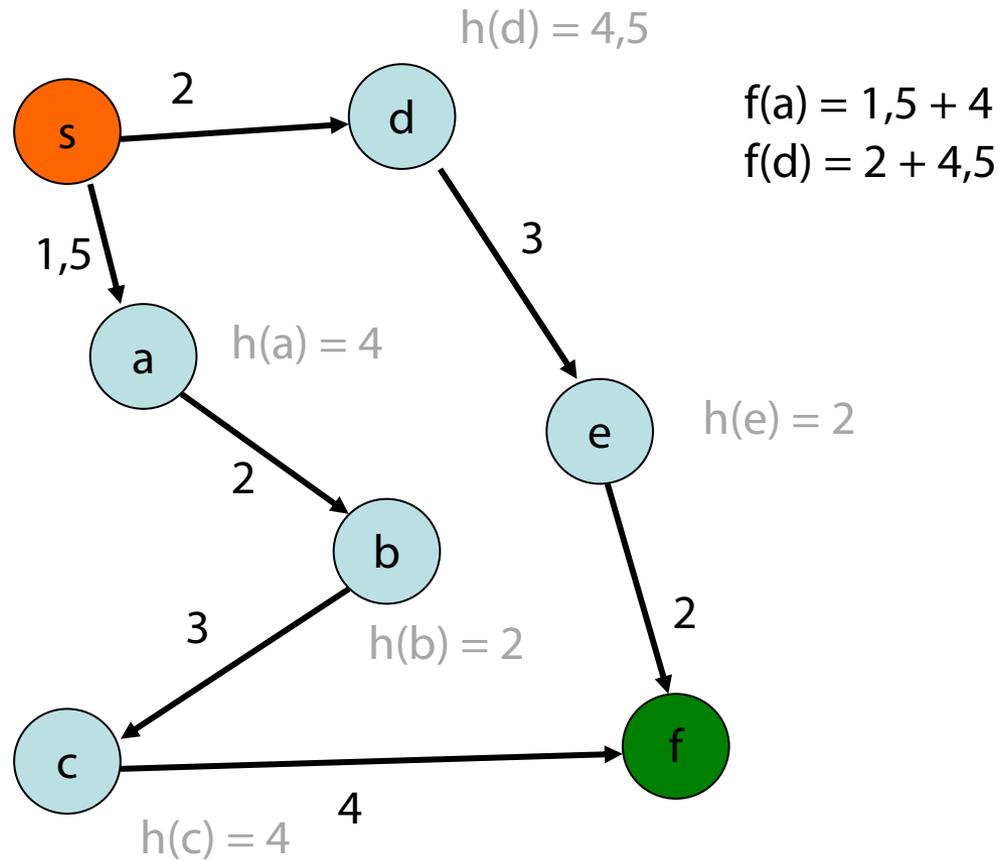
Kürzeste Wege: Heuristische Suche

- Was sollten wir machen, wenn nur der kürzeste Weg zu einem gegebenen Knoten gesucht ist?
 - Es werden im Dijkstra-Algorithmus zu viele Knoten betrachtet (d.h. "expandiert")
 - Weiterhin: Keine Abschätzung der Entfernung zum Ziel
- Abhilfe: A*-Algorithmus
 - Informierte Suche mit Zielschätzer (Heuristik)

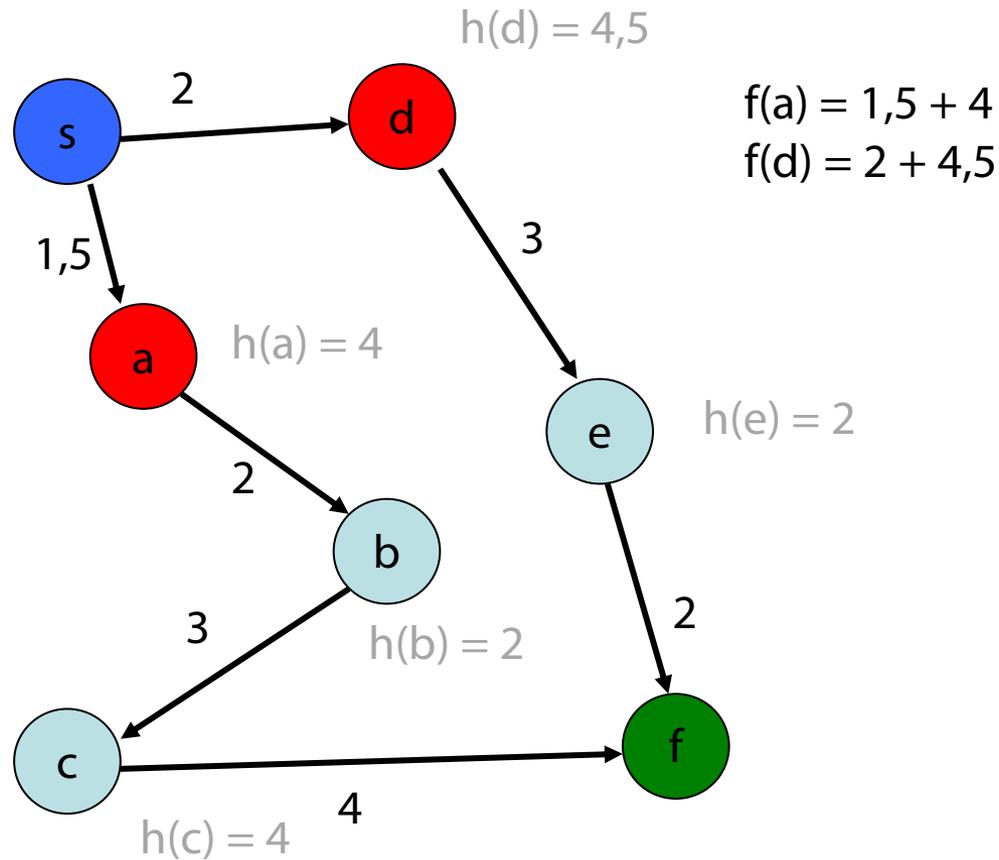
P. E. Hart, N. J. Nilsson, B. Raphael: A Formal Basis for the Heuristic Determination of Minimum Cost Paths, IEEE Transactions on Systems Science and Cybernetics SSC4 (2), pp. 100–107, **1968**

P. E. Hart, N. J. Nilsson, B. Raphael: Correction to „A Formal Basis for the Heuristic Determination of Minimum Cost Paths“, SIGART Newsletter, 37, pp. 28–29, **1972**

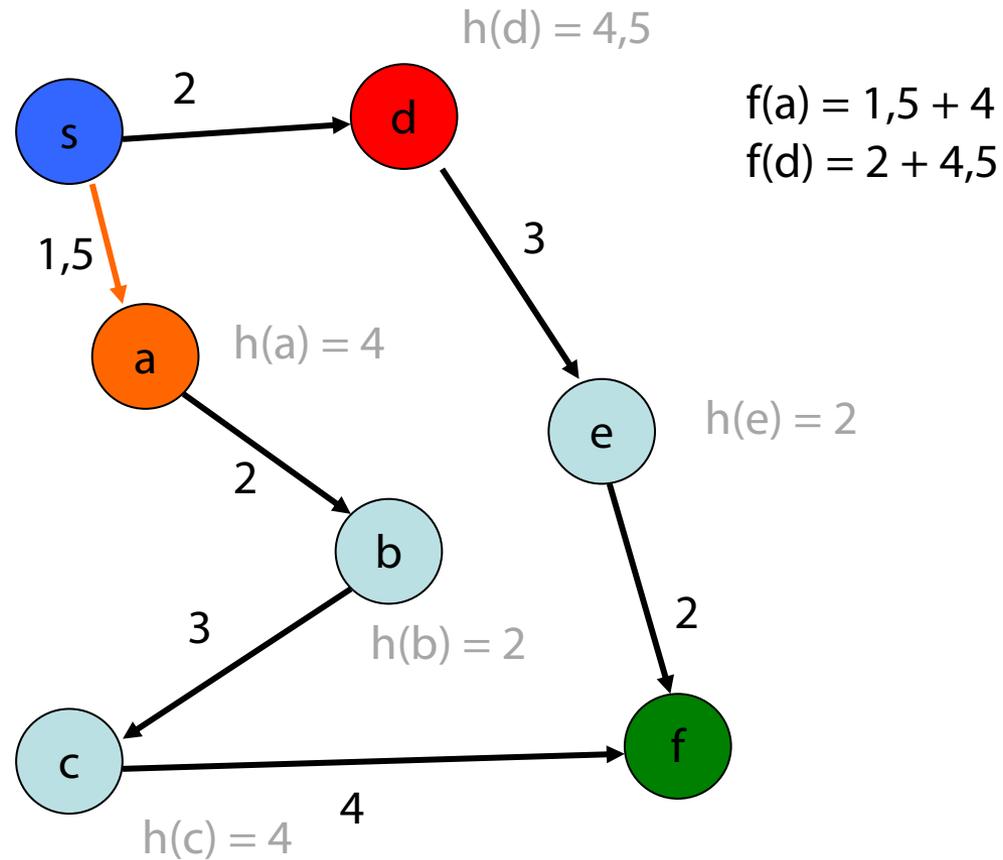
A* – Beispiel



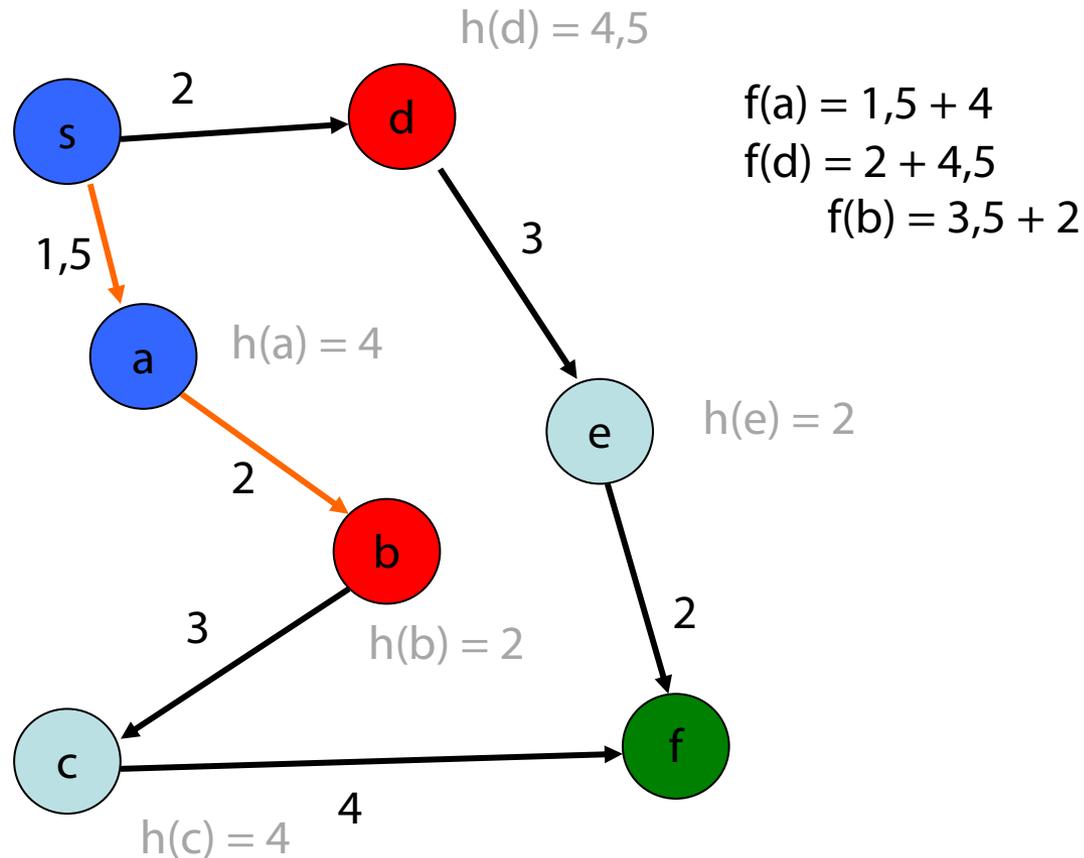
A* – Beispiel



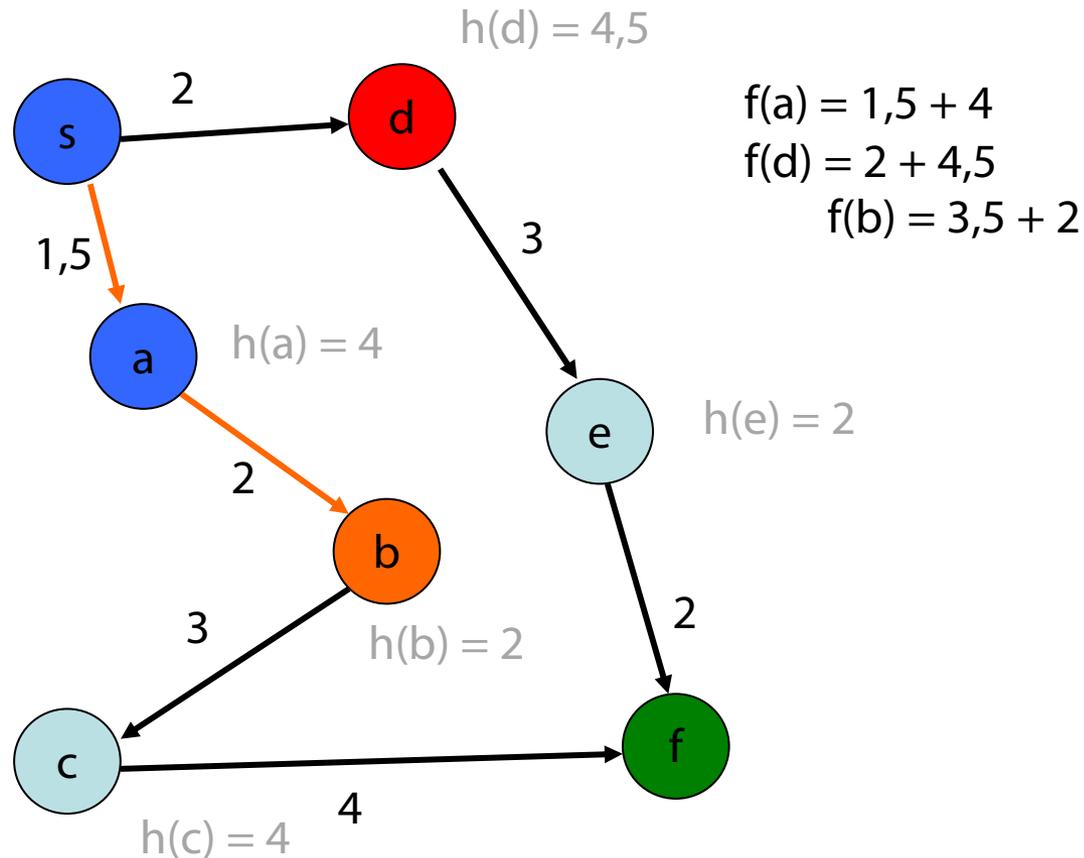
A* – Beispiel



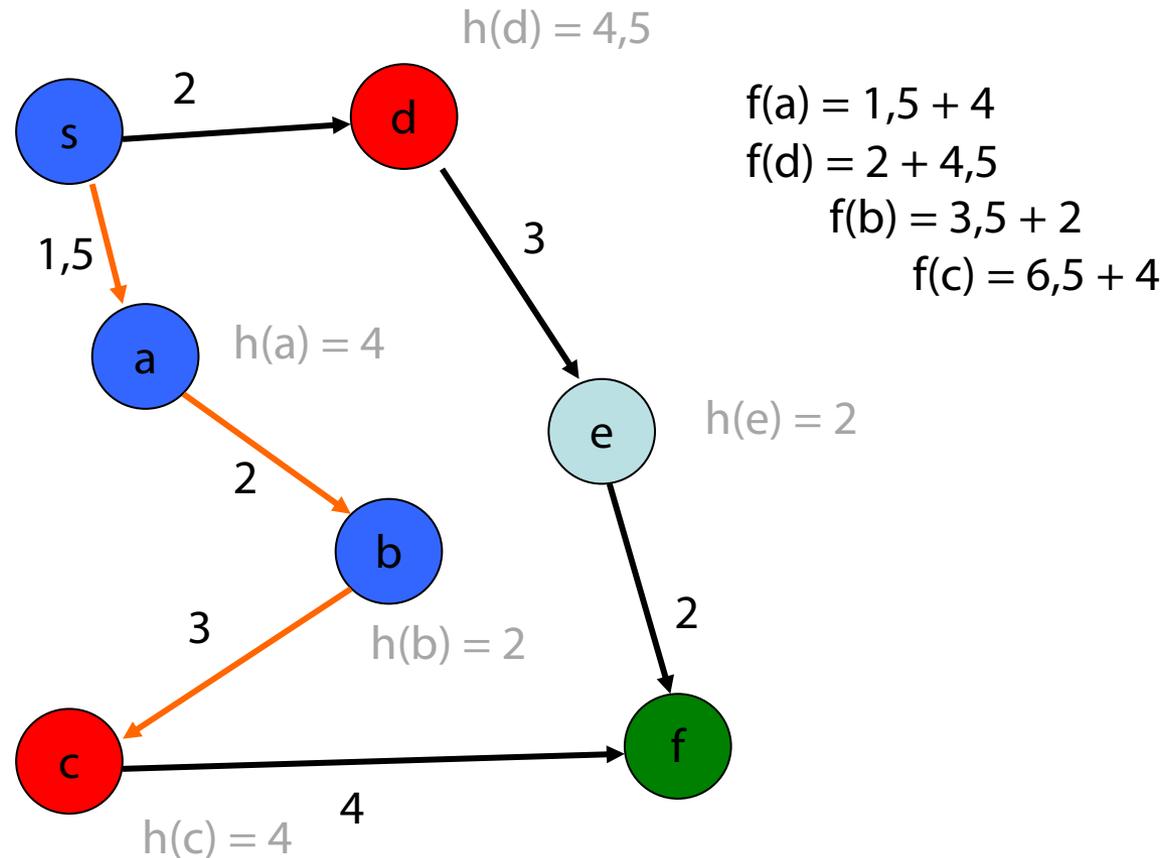
A* – Beispiel



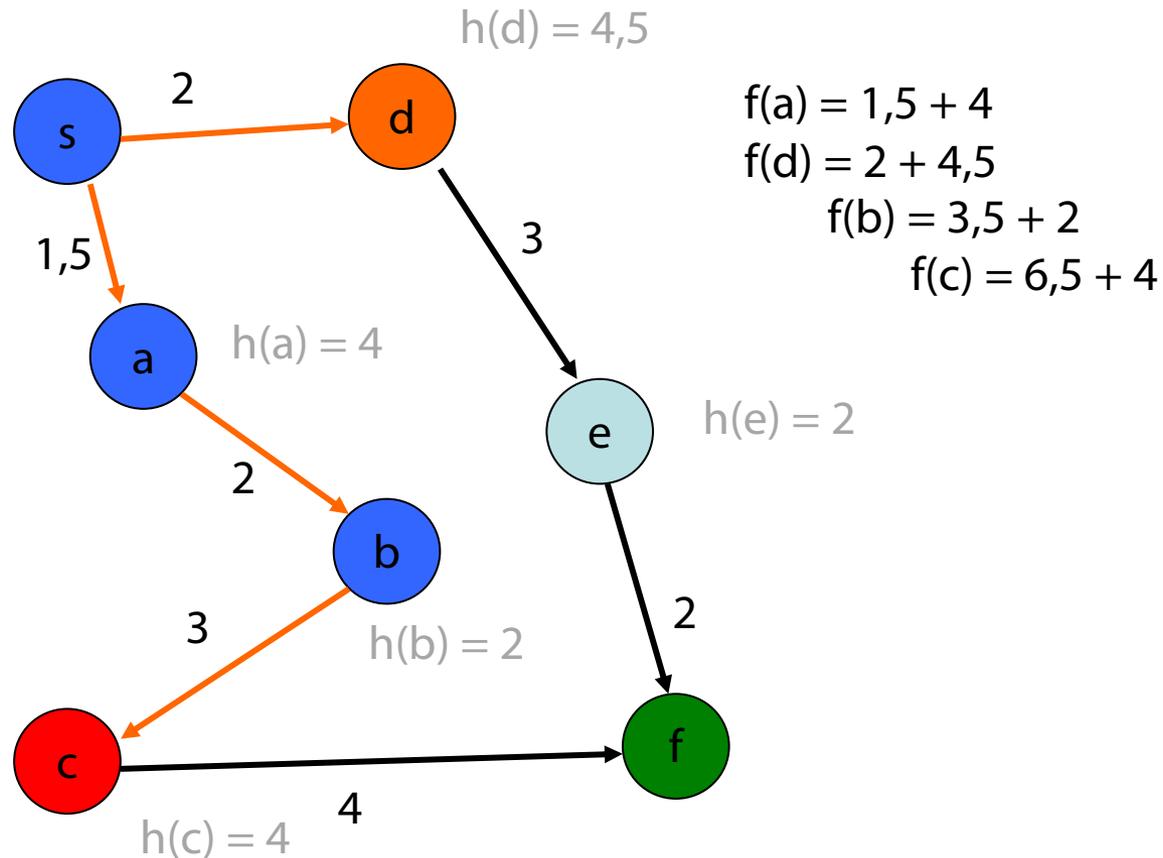
A* – Beispiel



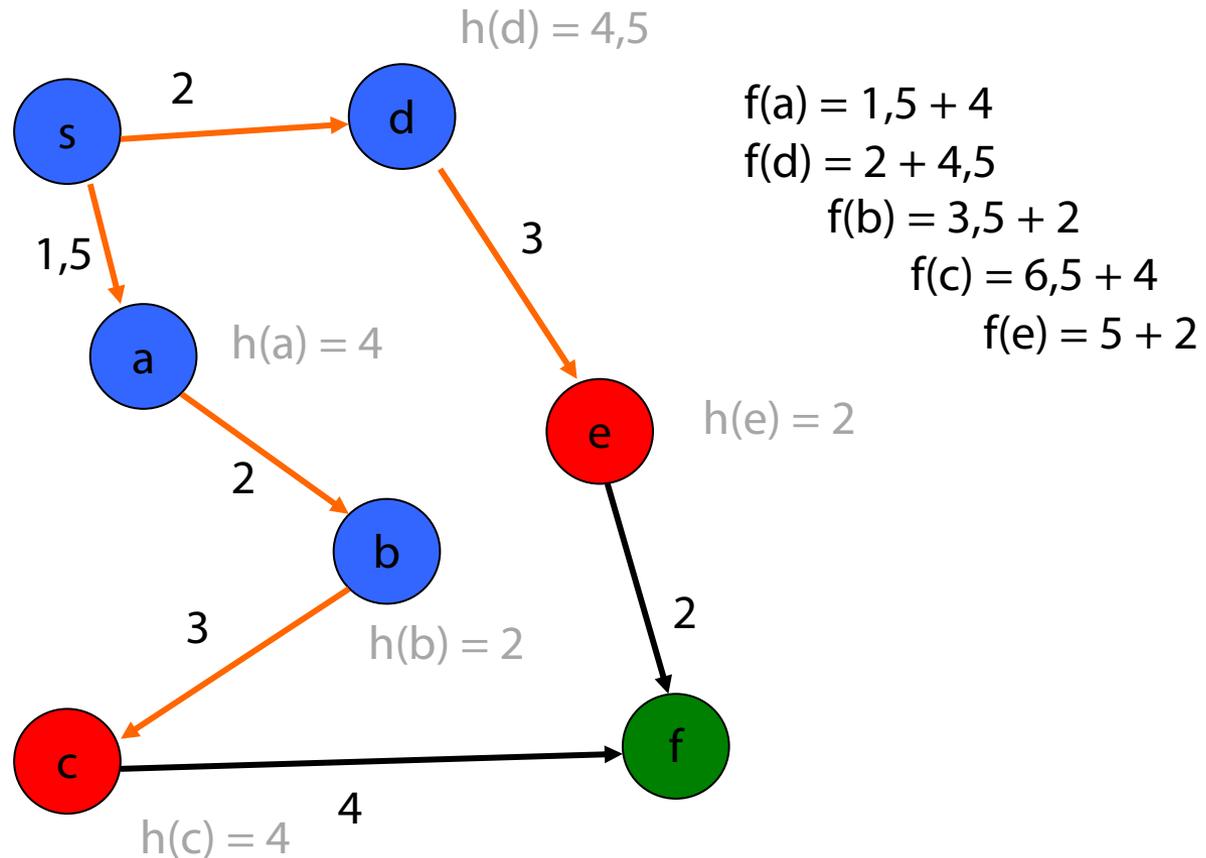
A* – Beispiel



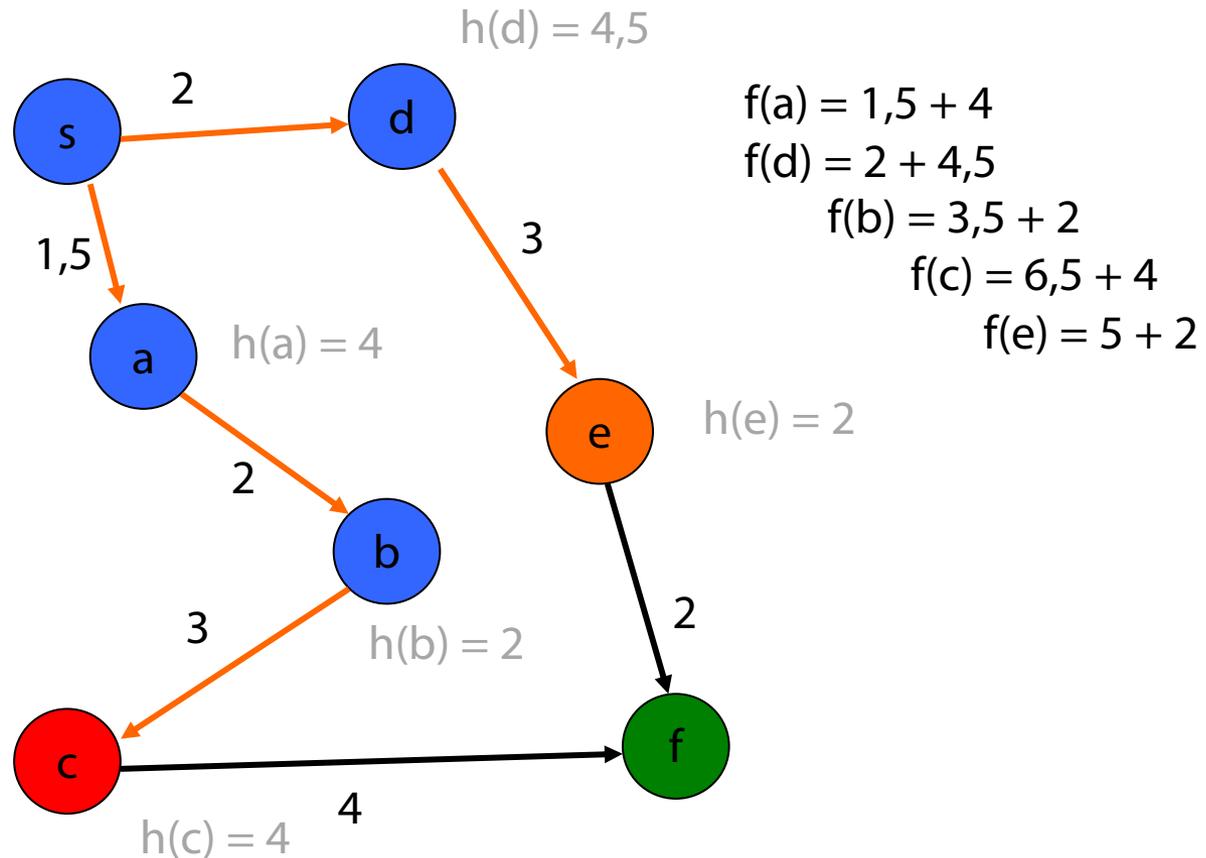
A* - Beispiel



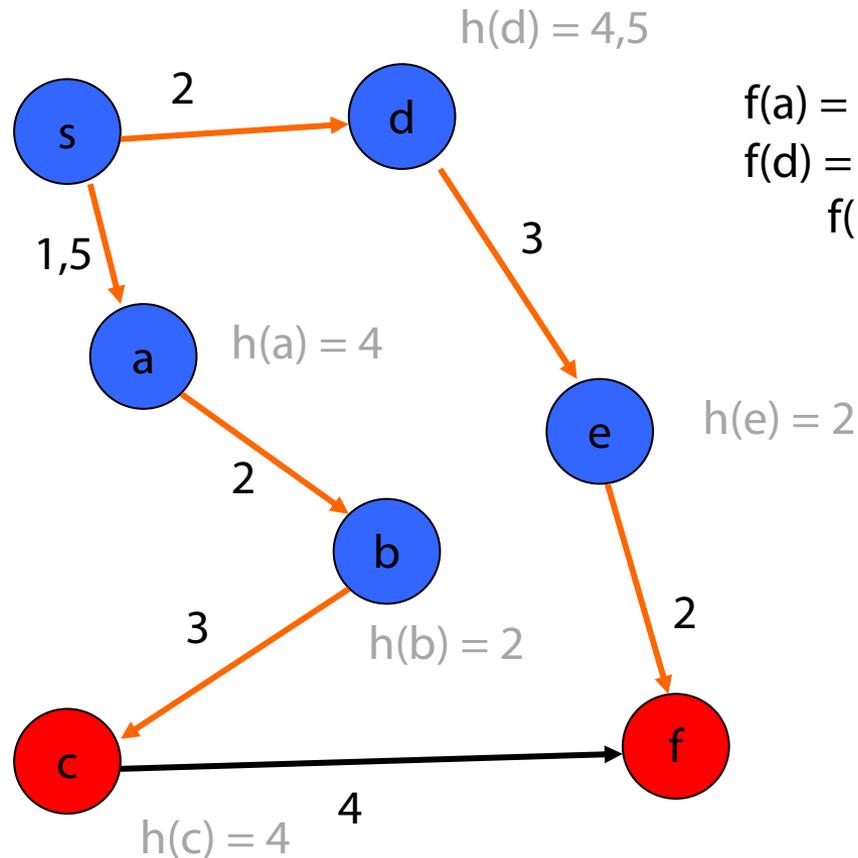
A* - Beispiel



A* – Beispiel



A* – Beispiel



$$f(a) = 1,5 + 4$$

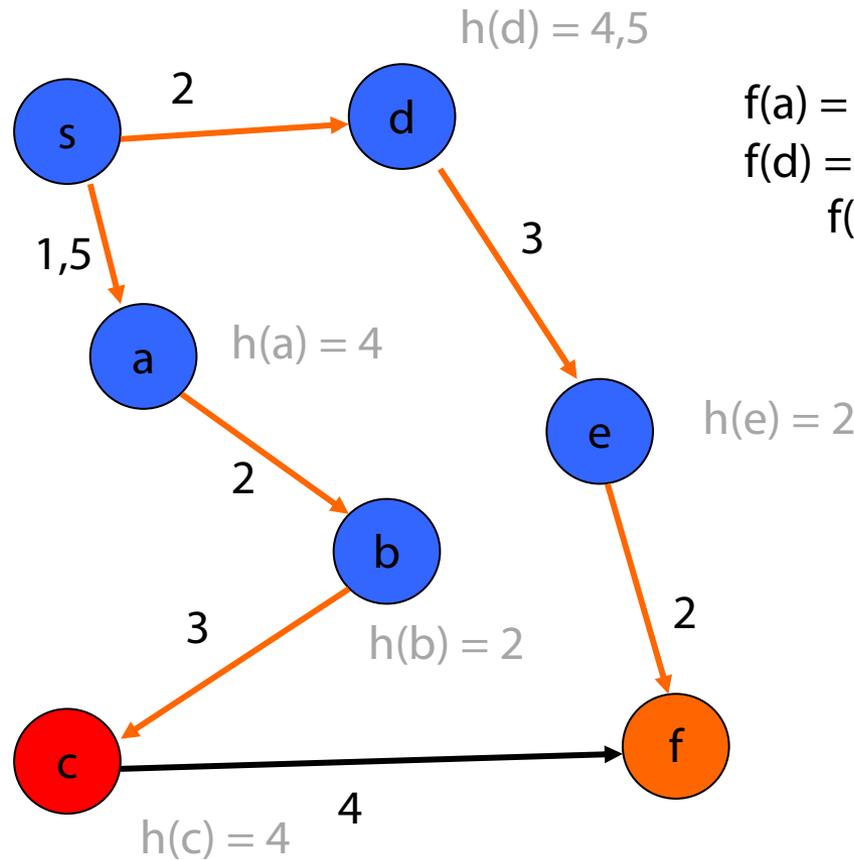
$$f(d) = 2 + 4,5$$

$$f(b) = 3,5 + 2$$

$$f(c) = 6,5 + 4$$

$$f(e) = 5 + 2$$

A* – Beispiel



$$f(a) = 1,5 + 4$$

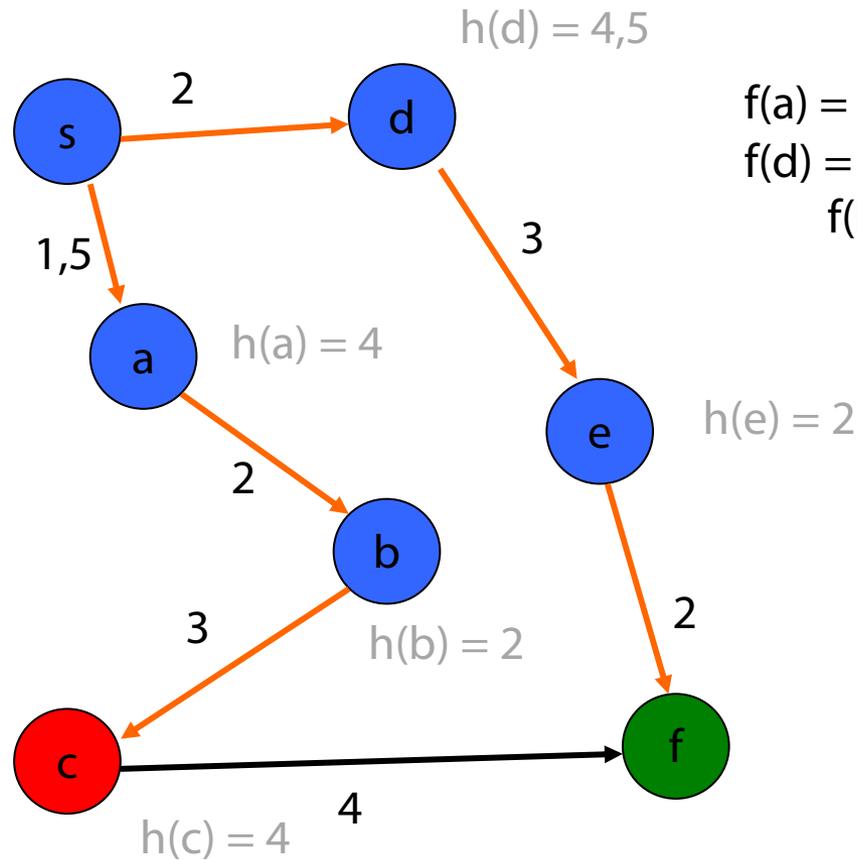
$$f(d) = 2 + 4,5$$

$$f(b) = 3,5 + 2$$

$$f(c) = 6,5 + 4$$

$$f(e) = 5 + 2$$

A* – Beispiel



$$f(a) = 1,5 + 4$$

$$f(d) = 2 + 4,5$$

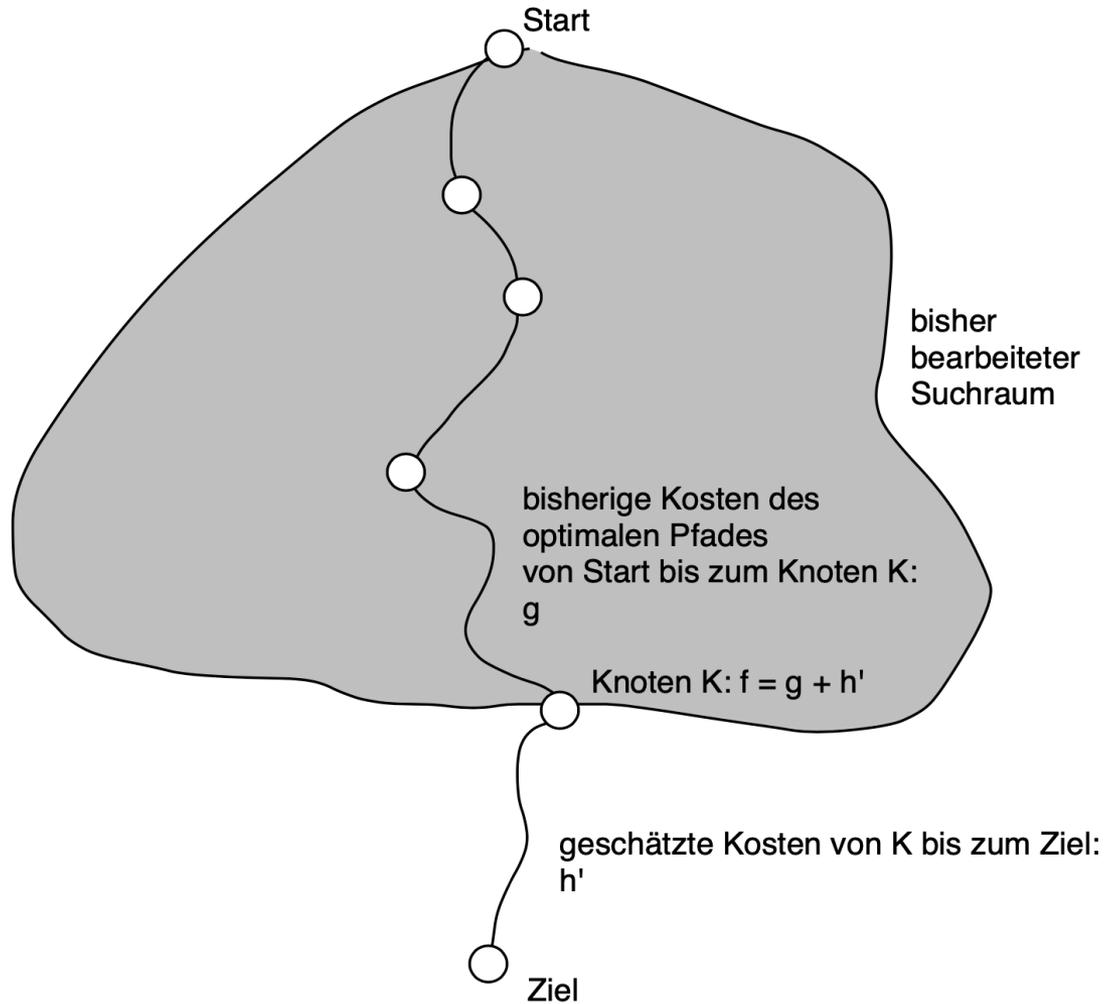
$$f(b) = 3,5 + 2$$

$$f(c) = 6,5 + 4$$

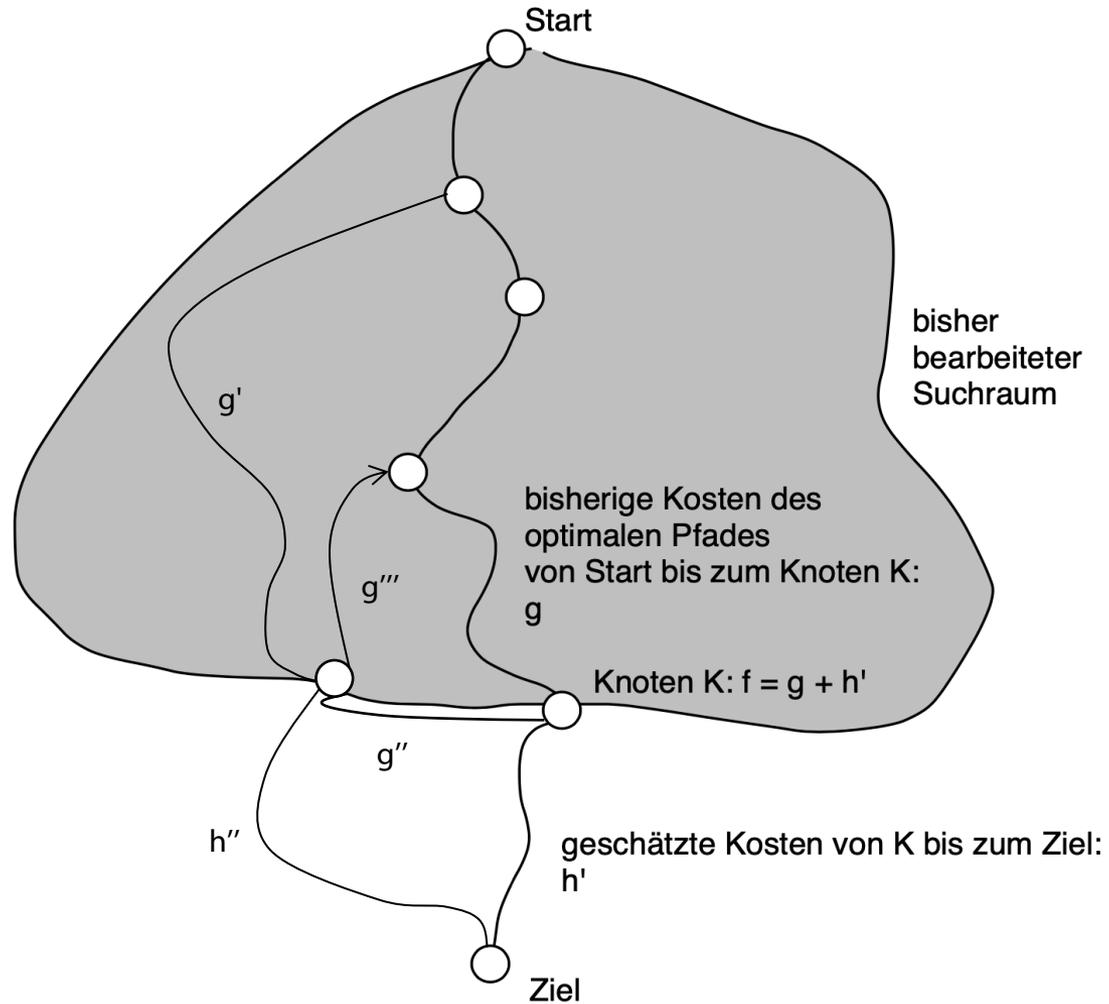
$$f(e) = 5 + 2$$

$$f(f) = 7 + 0$$

Exploration des (implizit definierten) Suchraums



Preis für den Zielschätzerereinsatz: Propagierung

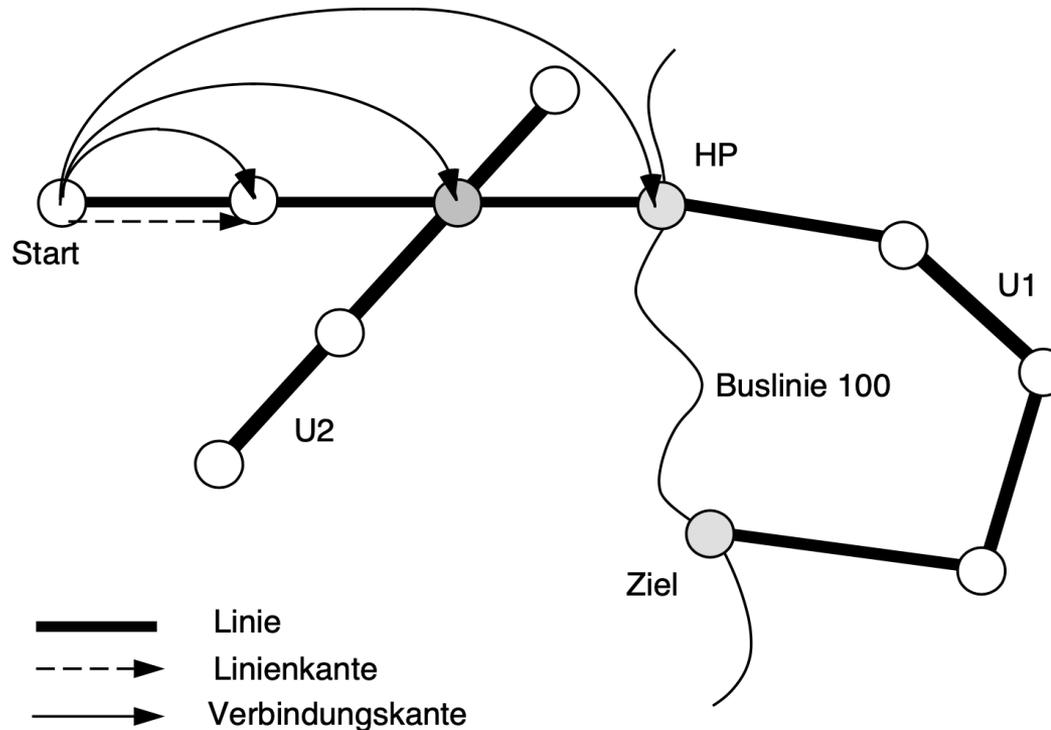


Anwendung: Fahrplaninformationssystem

- Vorwärtssuche
 - Gegeben: Start- und Zielhaltepunkt und frühestmögliche Abfahrtszeit am Start
- Rückwärtssuche
 - Gegeben: Start- und Zielhaltepunkt und spätestmögliche Ankunftszeit am Ziel
- Zu berücksichtigen
 - Umsteigezeiten zwischen Linien an einem Haltepunkt
 - Fußwegezeiten (vgl. z.B. Jungfernstieg)
 - Umsteigemodalitäten
 - Behindertengerechter Umstieg (1991 nicht selbstverständlich)
 - Erweiterung: Start- und Zielort (Fußwegezeiten zu den umliegenden Haltepunkte bestimmen)

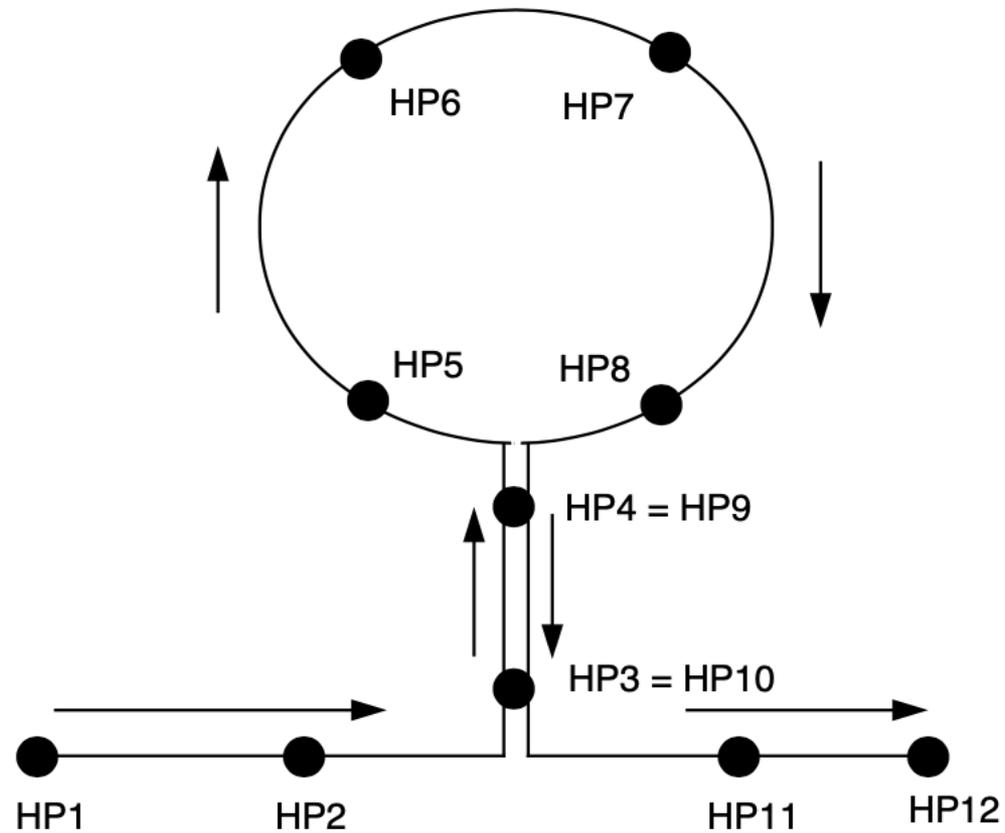
Anwendung: Fahrplaninformationssystem

- Nachverkehrssystem repräsentiert als Graph



- Realisierung mit A^* : Wie ist der Suchgraph definiert?

Besondere Linienformen in der Praxis

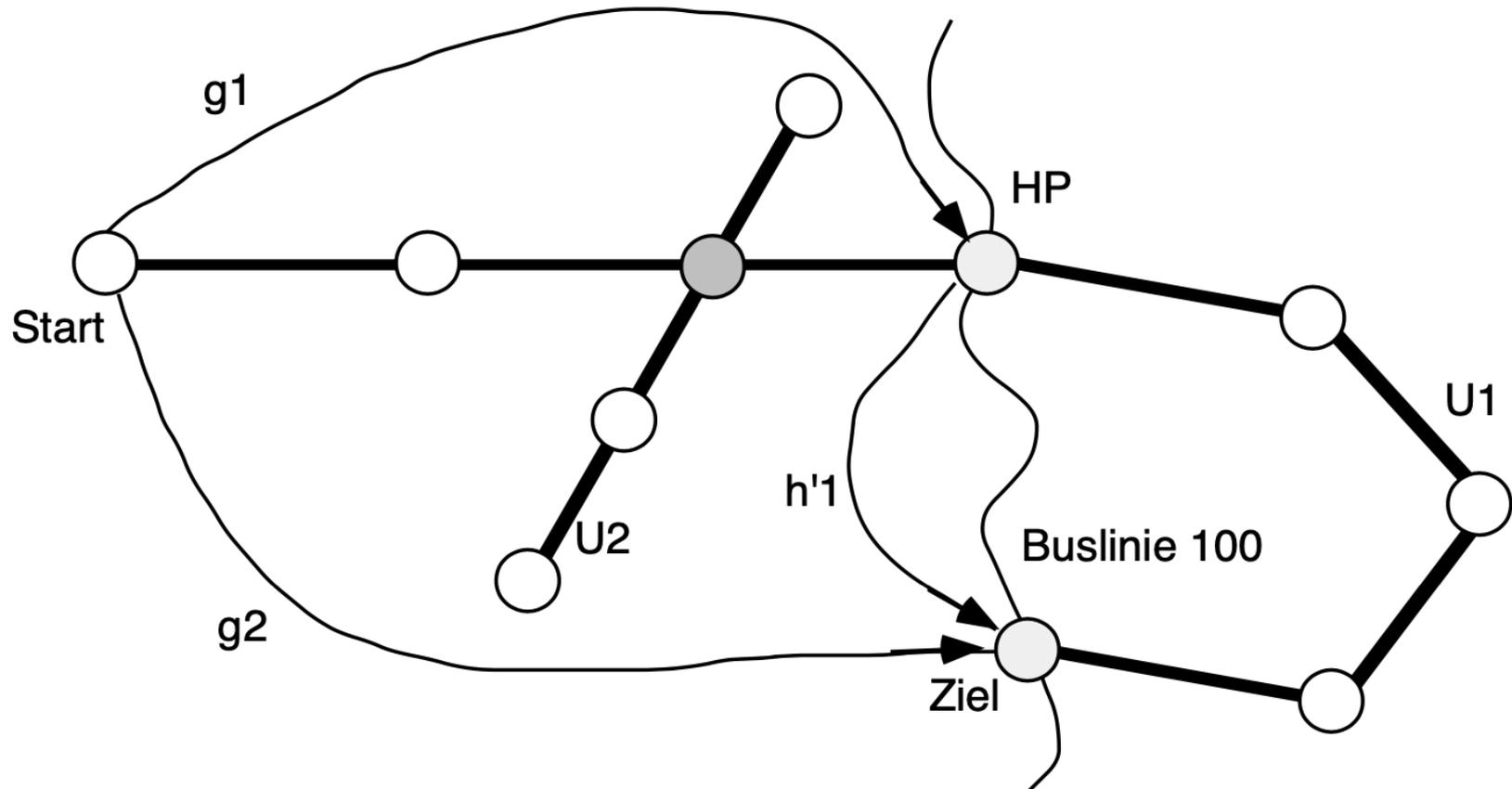


Kosten

- Fahrtzeit
- Anzahl der Umsteigevorgänge
- Fußwegezeit
- Behindertengerechtigkeit
- Verkehrsmitteltyp
- ...

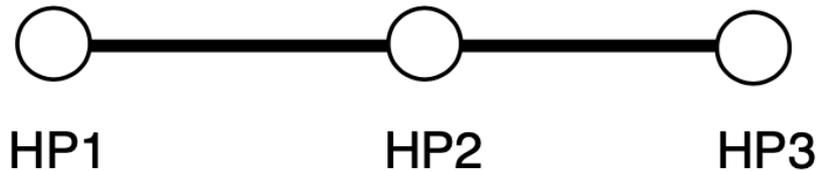
- Pro Kante: (reine Fahrzeit, abstrakte Fahrzeit)
 - **Reine Fahrzeit:** Bestimmung frühestmögliche Abfahrtzeit/Ankunftszeit am Ziel
 - **Abstrakte Fahrzeit:** Kosten

Zeit, Kosten, Zielschätzung



Vorwärts- und Rückwärtssuche

Zug 1	08:00	08:15	
Zug 2	08:10	08:25	
Zug 3		08:40	09:00



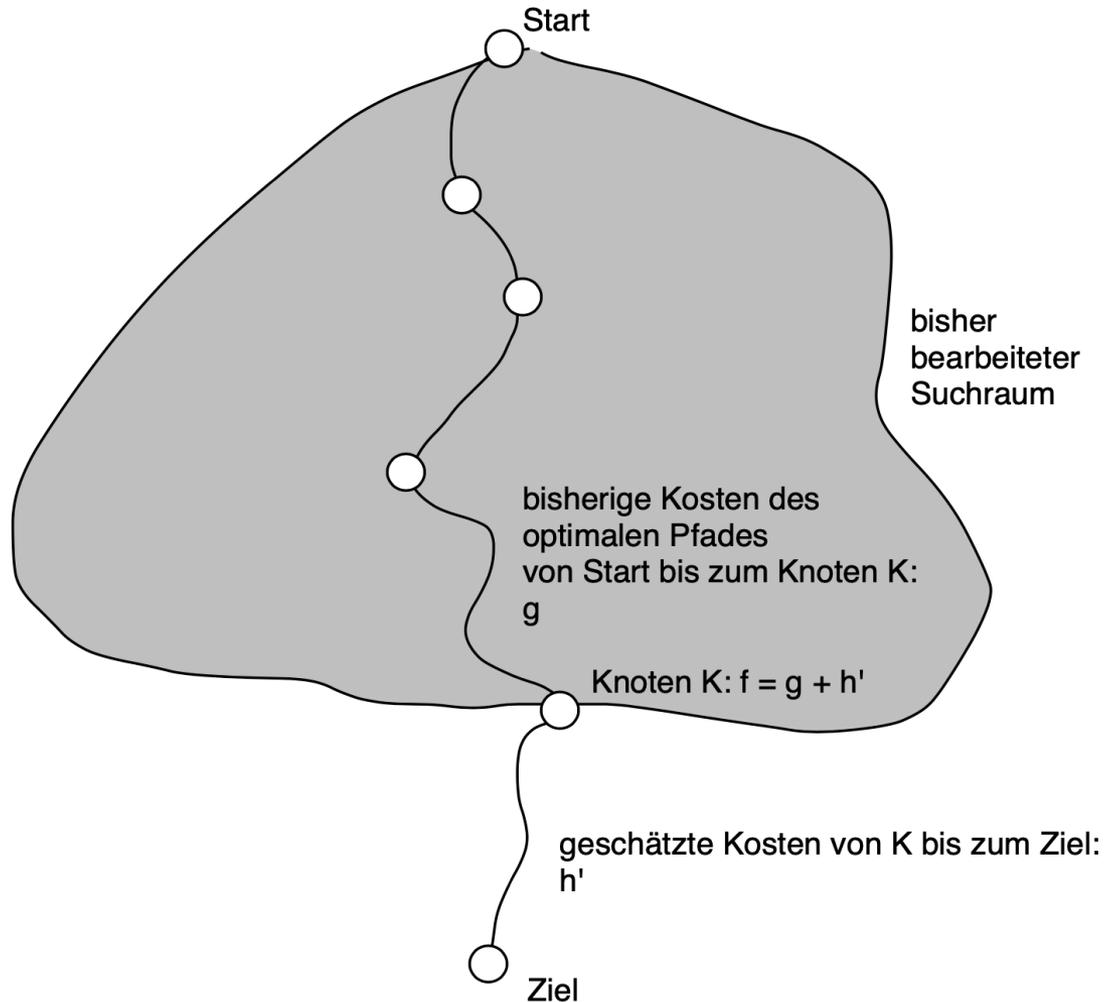
Abfahrtszeitpunkt 08:00 Ziel

- Auf Vorwärtssuche (frühestmögliche Ankunftszeit) folgt Rückwärtssuche (spätestmögliche Abfahrtszeit)

Schätzergenerierung: Offline-Berechnung von h

- Schätzergenerierung aus **geographischen Daten?**
 - Haben wir verworfen (Elbe-dazwischen-Problem)
- Schätzergenerierung aus **Fahrplan**
 - Schnellstmögliche direkte Verbindung zwischen jedem Paar von Haltepunkten unabhängig vom Zeitpunkt (ohne Umsteigen)
 - Es entsteht ein Graph mit direkten Verbindungen als Kanten, deren Kostenbeschriftung in Fahrtzeit angegeben wird
- Zu lösen: **All-Pairs Shortest Paths** Problem zur Bestimmung der schnellsten Verbindung mit Umsteigen
 - Wartezeiten auf Anschluss ignoriert
 - Fußwegezeiten ignoriert
- Schätzer soll Kosten unterschätzen

Exploration des (implizit definierten) Suchraums



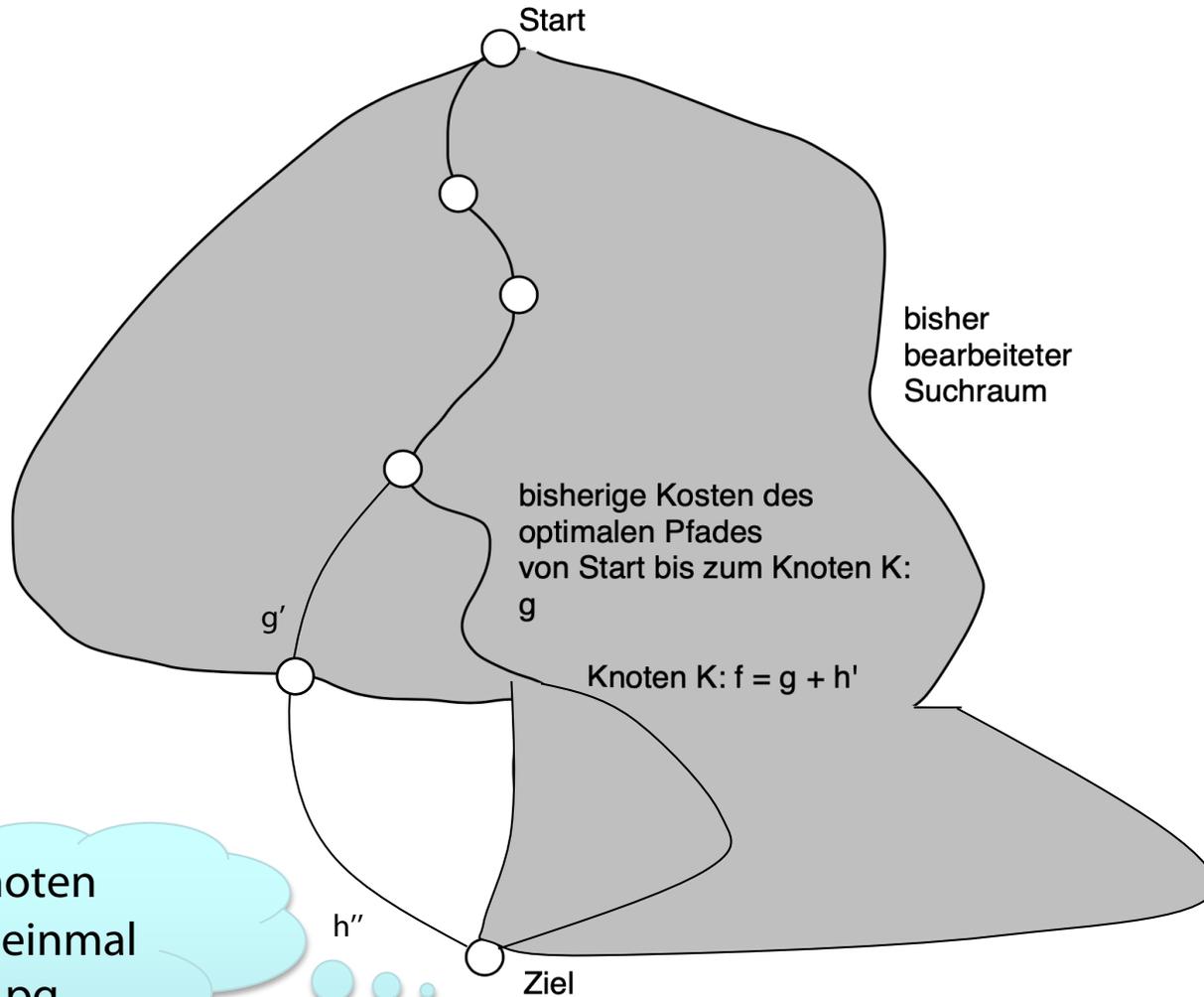
Analyse

Sei $G=(V, E)$ ein Graph mit positiven Kantenkosten und h^* eine Funktion, die die tatsächlichen Kosten von jedem Knoten $u \in V$ zum Ziel $v \in V$ bestimmt.

Definition: Ein Schätzer h heißt **zulässig**, wenn für alle $v \in V$ gilt, dass $h(v) \leq h^*(v)$, wobei $h^*(v)$ die optimale Schätzfunktion darstellt, die die tatsächlichen Kosten genau einschätzt.

Behauptung: A^* findet die optimale Lösung, wenn h zulässig ist

Zulässigkeit des Schätzers und optimale Lösung



Auch Zielknoten
kommen erst einmal
nur in die pq

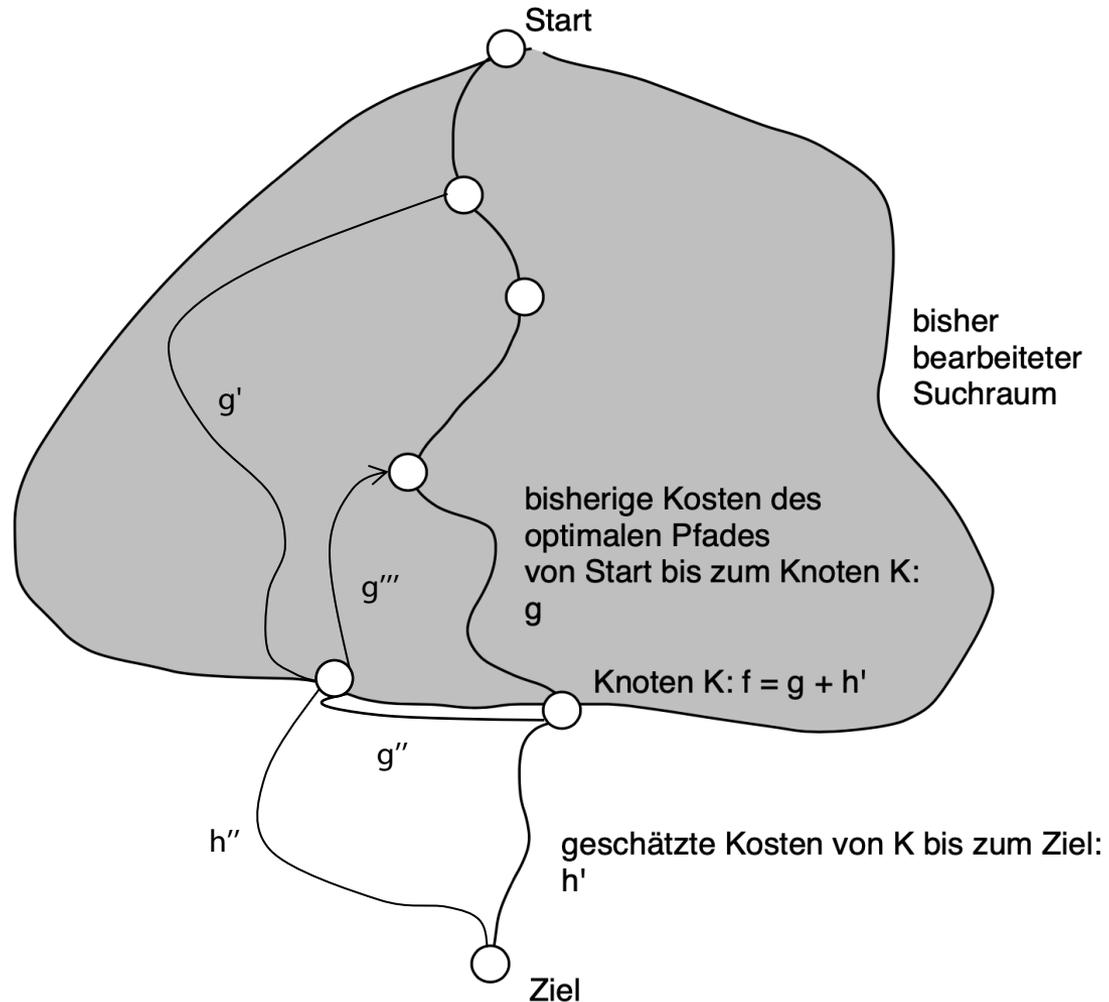
Funktion A*

```
function a_star(s, goal, h, g) # Startknoten, Zielknoten, Schaetzer, Graph
  s.g = 0; s.f = s.g + h(s); s.parent = s # Startknoten fuer PQ
  pq = build_pq([s], (u) -> u.f, (u, new_key) -> u.f = new_key)
  expanded = Set() # Menge mit expandierten Knoten
  while !empty_pq(pq)
    u = delete_min(pq)
    push!(expanded, u)
    if u == goal return path(u) end # Ausgabe: Loesungspfad (rueckwaerts)
    for (_, v, cost) in edges_out(u, g)
      if v ∉ expanded # Knoten noch nicht expandiert?
        if v ∉ pq # Knoten noch nicht gesehen?
          v.g = u.g + cost; v.f = v.g + h(v); v.parent = u
          insert(v, pq)
        else # Bekannter Knoten, neuer Weg dorthin
          new_g = u.g + cost
          if new_g < v.g # Neuer Weg kuerzer?
            decrease_key(v, pq, v.g - new_g )
            v.parent = u; v.g = new_g # Eltern und Weglaenge anpassen
          end end
        elseif u.g + cost < v.g # Kuerzterer Weg zu bereits expandiertem Knoten
          propagate(u, v, u.g + cost, expanded, pq, g) # Neue Kosten propagieren
        end end end end
end end end end
```

Hilfsfunktionen

```
function path(u)
  # Konstruiere Pfad p ausgehend vom Zielknoten u
  p = make_stack()
  while u.parent != u # Startknoten hat sich
                        selbst als Eltern
    push(u, p)
    u = u.parent
  end
  push(u, p) # Startknoten anfüegen
  return p
end
```

Preis für den Zielschätzerereinsatz: Propagierung



Hilfsfunktionen

```
function propagate(u, v, new_g, expanded, pq, g)
  # Propagiere neue Kosten new_g fuer Knoten v in
  # die Nachfolger von v

  if v ∉ expanded          # Noch nicht expandiert
    if new_g < v.g         # Kuerzterer Weg
      decrease_key(v, pq, v.g - new_g) # Neue Prio. in PQ
      v.parent = u; v.g = new_g
    end
  else                      # Bereits expandiert
    if new_g < v.g         # Kuerzterer Weg
      v.parent = u; v.g = new_g # Eltern, Distanz aktual.
      for (_, w, cost) in edges_out(v, g)
        propagate(v, w, new_g + cost, expanded, pq, g)
      end                  # Weiter propagieren
    end
  end end

end
```

Analyse von A*

- Kantenkosten müssen positiv sein
- Schlimmster Fall:
 - $h(n) = 0$ für alle Knoten n
 - Dann Verhalten wie beim Dijkstra-Algorithmus
- Aber: Je besser der Schätzer, desto besser das Verhalten
 - Bei optimalem Schätzer h^* Verhalten linear zur Länge des Lösungspfades (durch h^* ist der Name A* motiviert)
- Schätzer h ist nicht immer einfach zu bestimmen
 - Anwendungsspezifisches Wissen
 - Funktion h geht als Parameter in A* ein

SSSP Anwendung: Ereignisplanung

- Ereignis 2 mind. 2 Tage nach Ereignis 1 $x_2 - x_1 > 2$
- Ereignis 3 mind. 6 Tage nach Ereignis 1 $x_3 - x_1 > 6$
- Ereignis 4 nicht mehr als 5 Tage vor Ereignis 3
- Ereignis 2 nicht mehr als 4 Tage vor Ereignis 3
- Ereignis 4 nicht mehr als 1 Tag vor Ereignis 2
- Ereignis 1 nicht mehr als 1 Tag vor Ereignis 4

- Tag von Ereignis $i = x_i$

- Lösung des Ungleichungssystems ergibt Ereignisplanung

$$x_1 - x_2 \leq -2$$

$$x_1 - x_3 \leq -6$$

$$x_3 - x_4 \leq 5$$

$$x_3 - x_2 \leq 4$$

$$x_2 - x_4 \leq 1$$

$$x_4 - x_1 \leq 1$$

Umwandlung in gerichteten Graphen

$$x_1 - x_2 \leq -2$$

$$x_1 - x_3 \leq -6$$

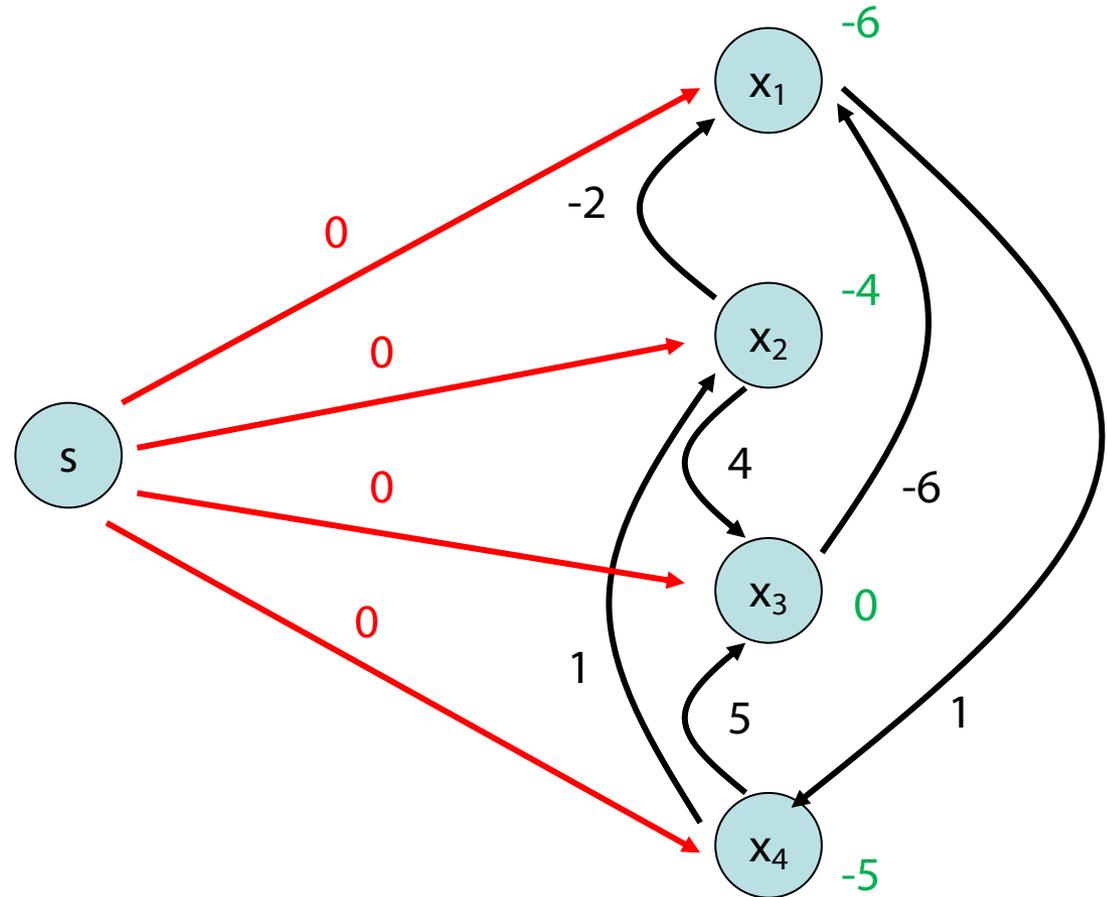
$$x_3 - x_4 \leq 5$$

$$x_3 - x_2 \leq 4$$

$$x_2 - x_4 \leq 1$$

$$x_4 - x_1 \leq 1$$

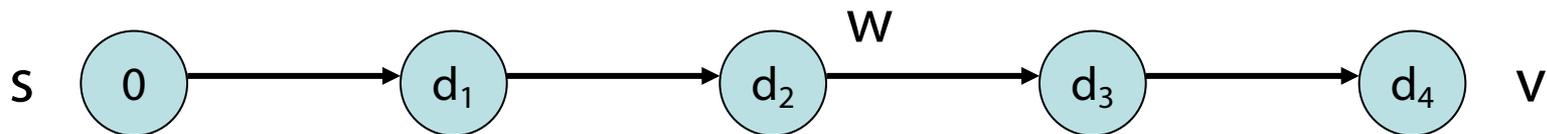
Lösung: $\mu(s, x_i)$



Bellman-Ford Algorithmus

Kürzeste Wege für beliebige Graphen mit **beliebigen** Kantenkosten (aber noch SSSP).

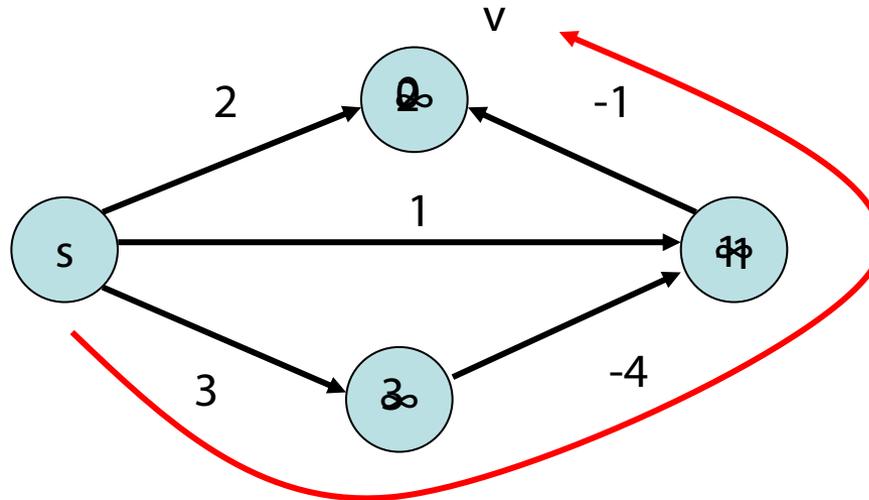
Problem: besuche Knoten eines kürzesten Weges in richtiger Reihenfolge



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

Bellman-Ford Algorithmus

Beispiel für Problem mit Dijkstra Algo:



Knoten v hat falschen Distanzwert!

Bellman-Ford Algorithmus

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

Beweis:

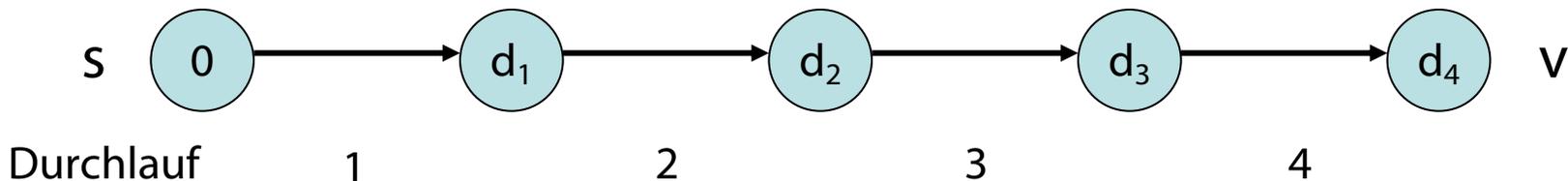
- Weg mit Kreis mit Kantenkosten ≥ 0 : Entfernen des Kreises erhöht nicht die Kosten
- Weg mit Kreis mit Kantenkosten < 0 : Distanz zu s ist $-\infty$!

Bellman-Ford Algorithmus

Folgerung: (für Graph mit n Knoten)

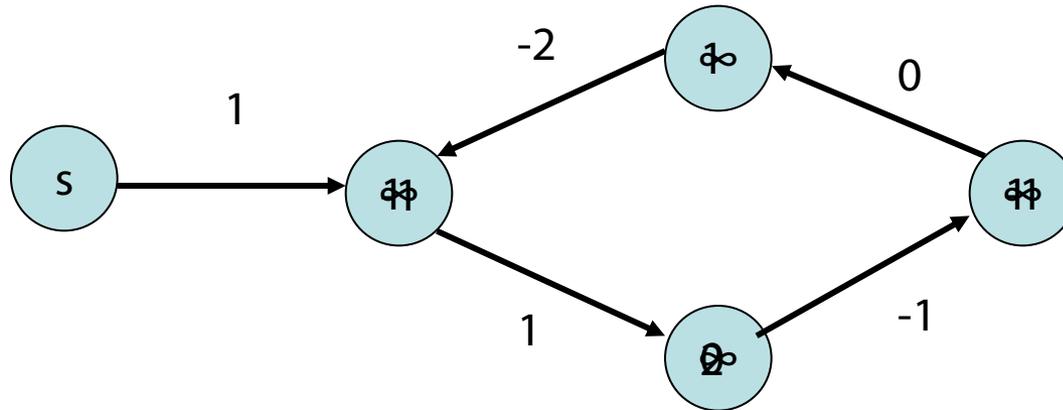
Für jeden Knoten v mit $\mu(s,v) > -\infty$ gibt es kürzesten Weg der Länge (Anzahl Kanten!) $< n$ zu v

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



Bellman-Ford Algorithmus

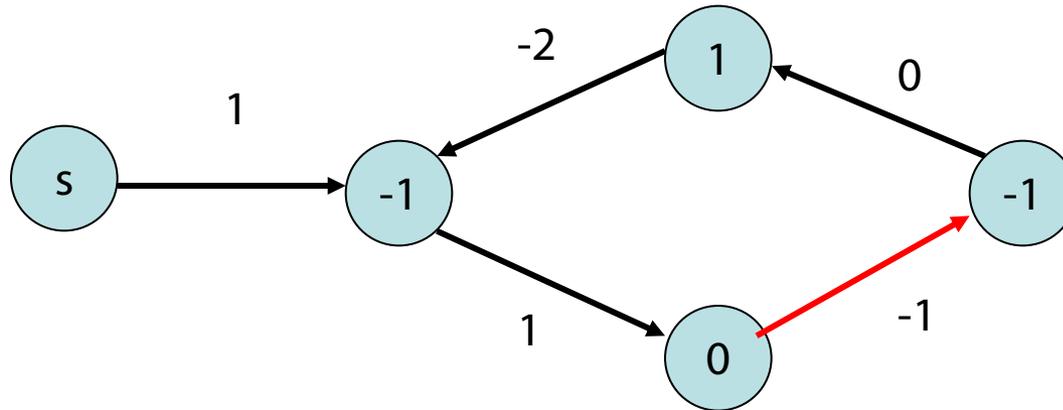
Problem: Erkennung negativer Kreise



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

Bellman-Ford Algorithmus

Problem: Erkennung negativer Kreise



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

Bellman-Ford Algorithmus

Keine Distanzniedrigung möglich:

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

Bellman-Ford Algorithmus

Zusammenfassung:

- **Keine Distanzniedrigung** mehr möglich
($v.d + c(v, w) \geq d[w]$ für alle w):
Fertig, $w.d = \mu(s, w)$ für alle w
- **Distanzniedrigung möglich** selbst noch in n -ter
Runde ($v.d + c(v, w) < w.d$ 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 (Infektion).

Bellman-Ford Algorithmus

```
function bellman_ford(s, g)
    n = num_nodes(g)
    s.d = 0; s.parent = s
    for i = 1:n-1    # aktualisiere Kosten fuer n-1 Runden
        for (v, w, cost) in edges(g)
            if w.d > v.d + cost    # bessere Distanz möglich?
                w.d = v.d + cost; w.parent = v
            end
        end
    end
    for (v, w, cost) in edges(g) # in n-ter Runde noch besser?
        if w.d > v.d + cost
            infect(w, g) # propagiere -Inf Kosten von w aus
        end
    end
end
```



Hilfsfunktion

```
function infect(v, g)
  # propagiere -Inf Kosten von v aus
  if v.d > -Inf # noch nicht markiert?
    v.d = -Inf
    for (v, w) in edges_out(v, g)
      infect(w, g)
    end
  end
end
```

Bellman-Ford Algorithmus

Laufzeit: $O(n \cdot m)$

Verbesserungsmöglichkeiten:

- Überprüfe in jeder Aktualisierungsrunde, ob noch irgendwo $v.d + c(v, w) < w.d$ ist.
Nein: fertig!
(Hauptschleife kann frühzeitig verlassen werden)
- Besuche in jeder Runde nur die Knoten w , für die Test $v.d + c(v, w) < w.d$ sinnvoll (d.h. $v.d$ 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: $O(n^2 \cdot m)$

All Pairs Shortest Paths

Bessere Strategie: Reduziere n Bellman-Ford Anwendungen auf n Dijkstra Anwendungen (auch Johnson-Dijkstra-Algorithmus genannt)

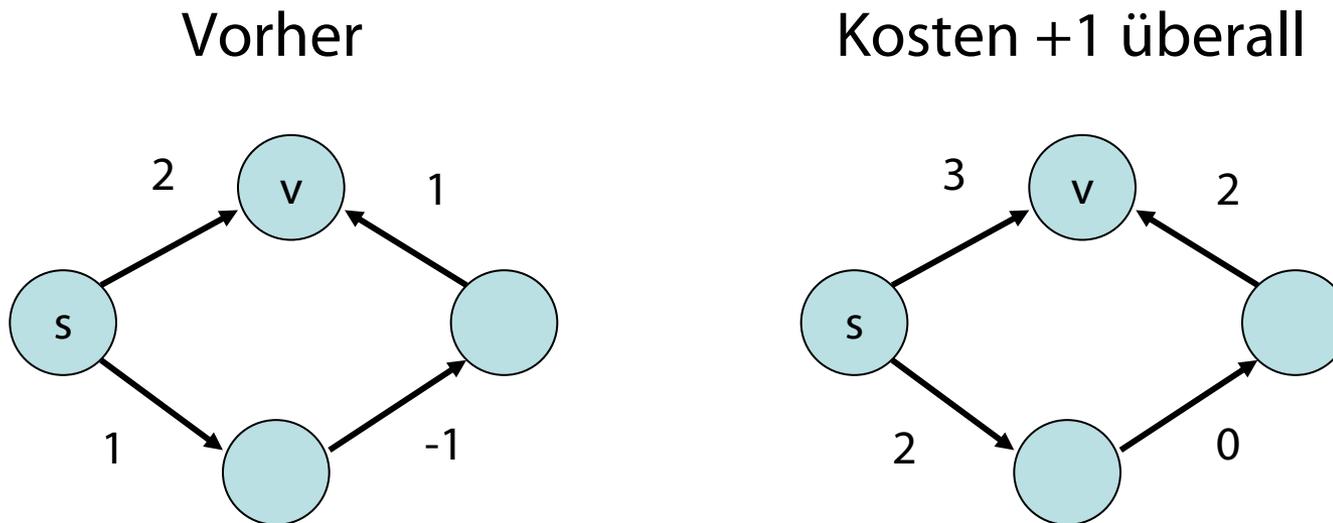
Problem: wir brauchen dazu **nichtnegative** Kantenkosten

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

Dijkstra erfordert, dass jeder expandierte Knoten über kürzesten Weg erreicht

Naive Erhöhung?

- Naive Umwandlung negativer Kantenkosten durch Addition von $c = -\min(\{0\} \cup \{c(e) \mid e \in E, c(e) < 0\})$ geht im Allgemeinen schief
- Gegenbeispiel zur Erhöhung um Wert c :



— : kürzester Weg

Neuer Ansatz

- Sei $\phi:V \rightarrow \mathbb{R}$ eine Funktion, die jedem Knoten ein **Potenzial** zuweist
- Die **reduzierten Kosten** von $e=(v,w)$ sind:
$$r(e) := \phi(v) + c(e) - \phi(w)$$
- ϕ müssen wir noch sinnvoll bestimmen
- Kantenkosten r und c in offensichtlicher Weise auf Wegekosten erweiterbar

Beh: Seien p und q Wege in G von v_1 bis v_k . Dann gilt für jedes Potenzial $\phi : r(p) < r(q)$ genau dann wenn $c(p) < c(q)$

All Pairs Shortest Paths

Beh : Seien p und q Wege in G von v_1 bis v_k . Dann gilt für jedes Potenzial $\phi : r(p) < r(q)$ 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:

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

All Pairs Shortest Paths

Beh: Angenommen, G habe keine negativen Kreise und alle Knoten können von s erreicht werden. Sei $\phi(v) = \mu(s, v)$ für alle $v \in V$.

Mit diesem ϕ ist $r(e) = \phi(v) + c(e) - \phi(w) \geq 0$ für alle $e = (v, w) \in E$.

Beweis:

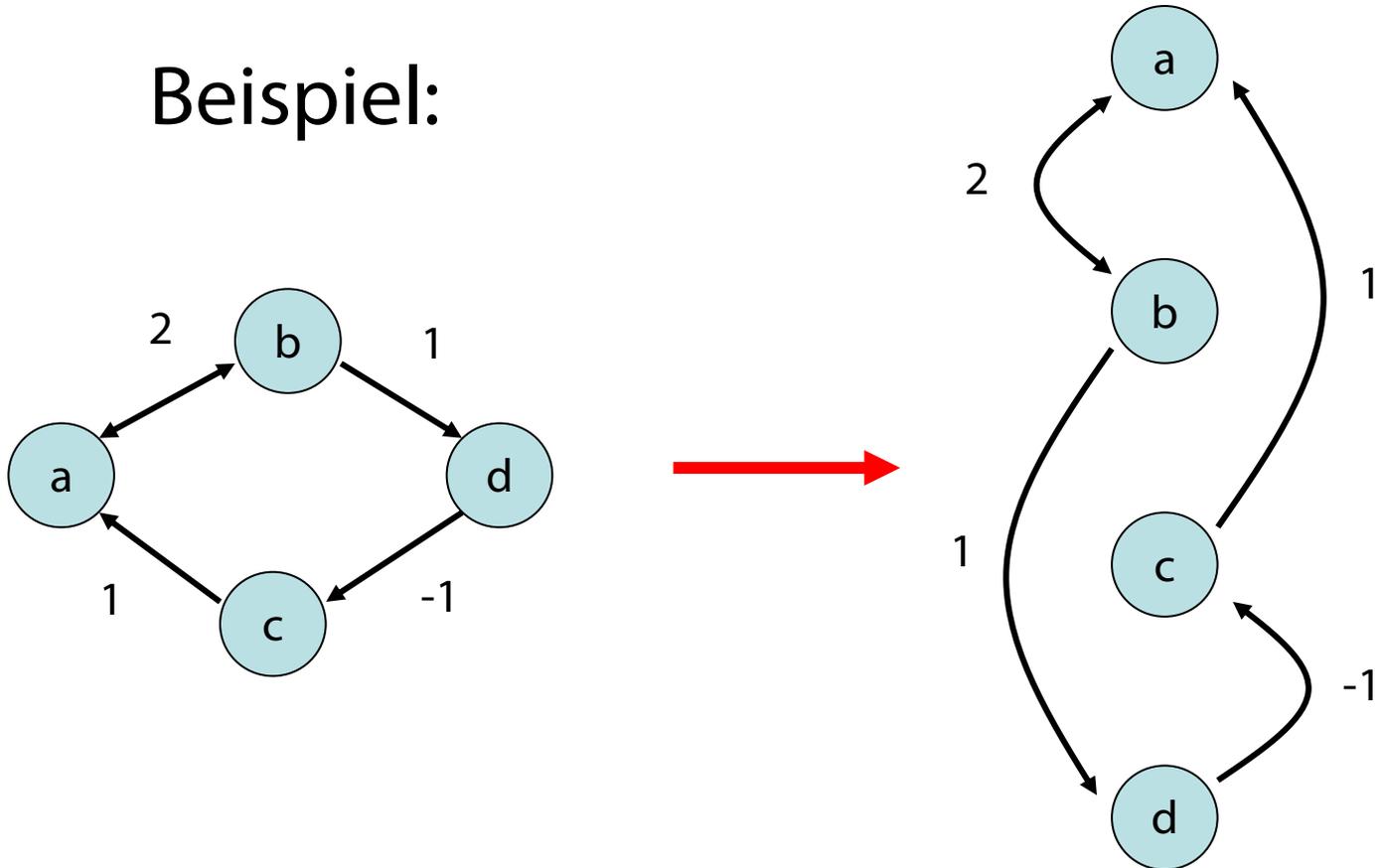
- Nach Annahme ist $\mu(s, v) \in \mathbb{R}$ für alle v
- Wir wissen: Für jede Kante $e = (v, w)$ ist $\mu(s, v) + c(e) \geq \mu(s, w)$ (wg. Minimalität von μ)
- Also ist $r(e) = \mu(s, v) + c(e) - \mu(s, w) \geq 0$

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 für $e = (v, w)$
 $r(e) := \phi(v) + c(e) - \phi(w)$
4. Berechne für alle Knoten v die Distanzen $\bar{\mu}(v,w)$ mittels **Dijkstra Algorithmus** mit reduzierten Kosten auf Graph **ohne Knoten s**
5. Berechne korrekte Distanzen $\mu(v,w)$ durch
 $\mu(v,w)=\bar{\mu}(v,w)+ \phi(w)- \phi(v)$

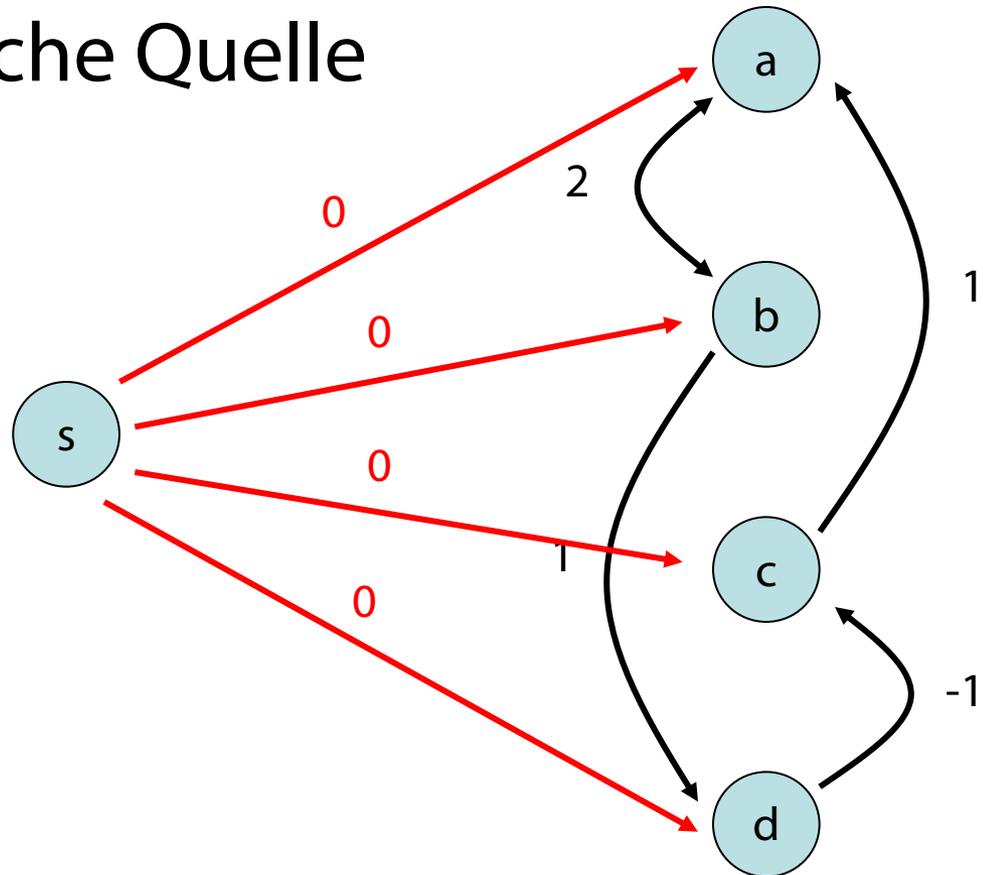
All Pairs Shortest Paths

Beispiel:



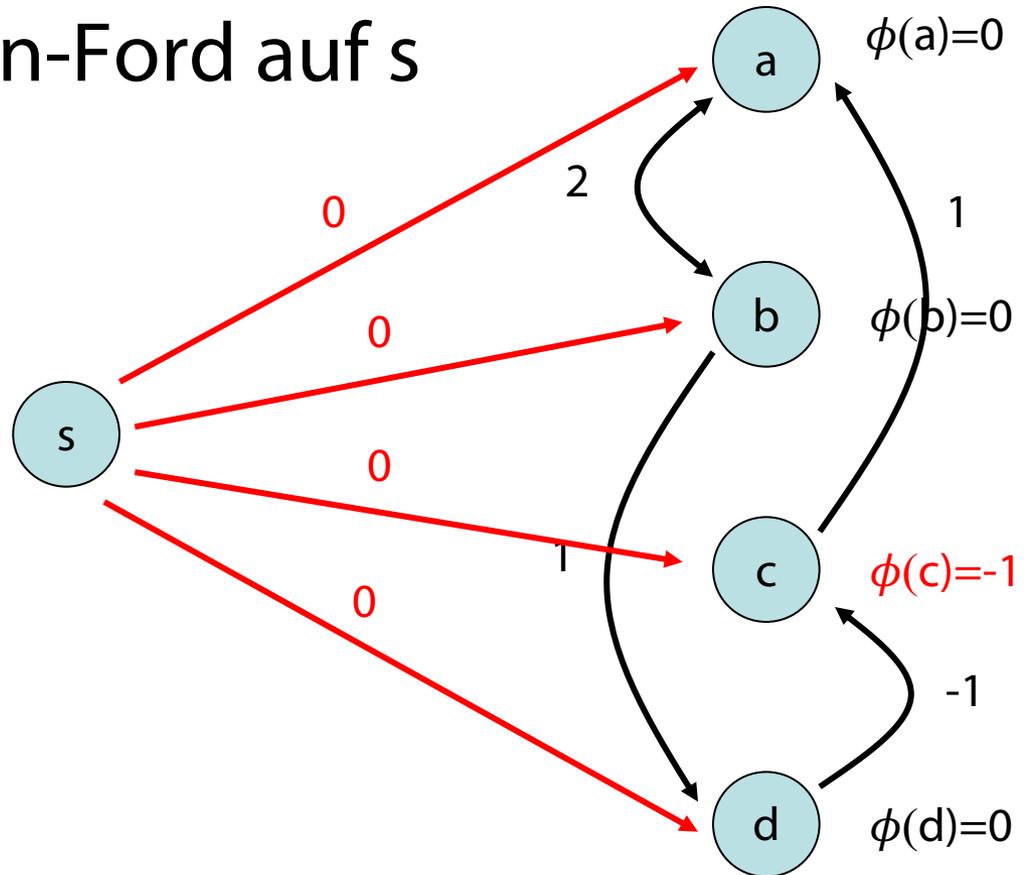
All Pairs Shortest Paths

Schritt 1: Künstliche Quelle



All Pairs Shortest Paths

Schritt 2: Bellman-Ford auf s

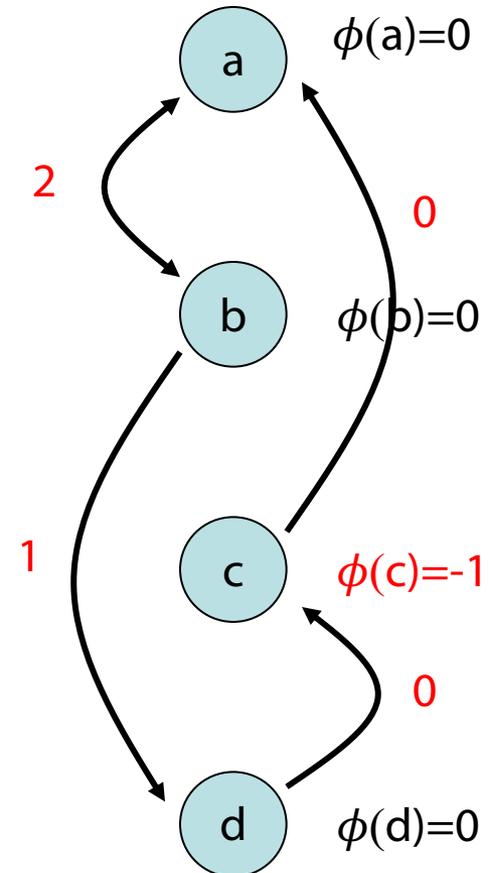


All Pairs Shortest Paths

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

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

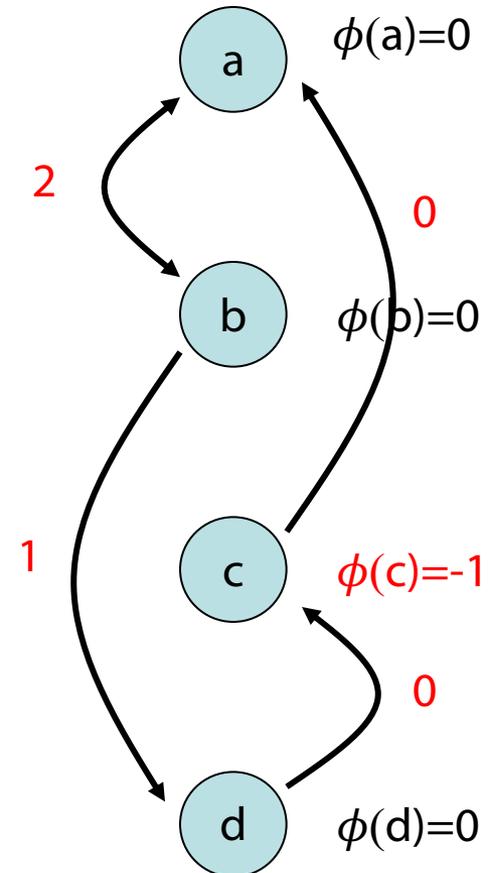
$$r(e) = \phi(v) + c(e) - \phi(w)$$



All Pairs Shortest Paths

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

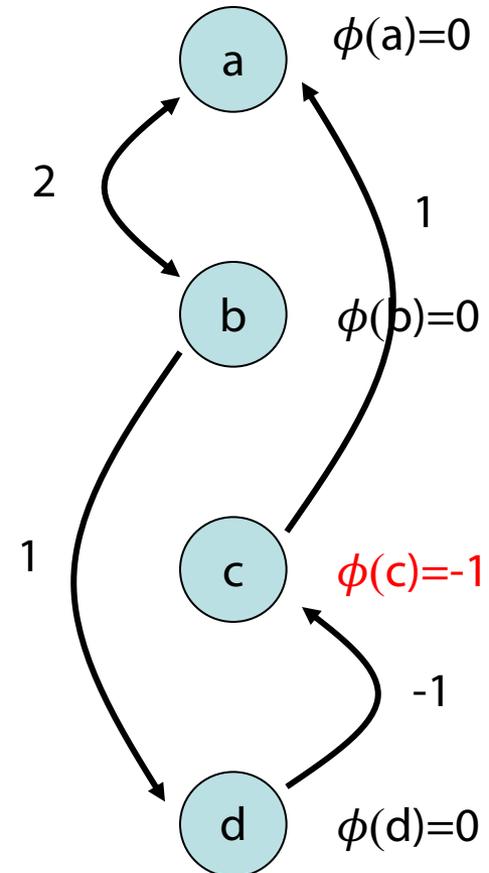
$\bar{\mu}$	a	b	c	d
a	0	2	3	3
b	1	0	1	1
c	0	2	0	3
d	0	2	0	0



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 (Johnson-Dijkstra):

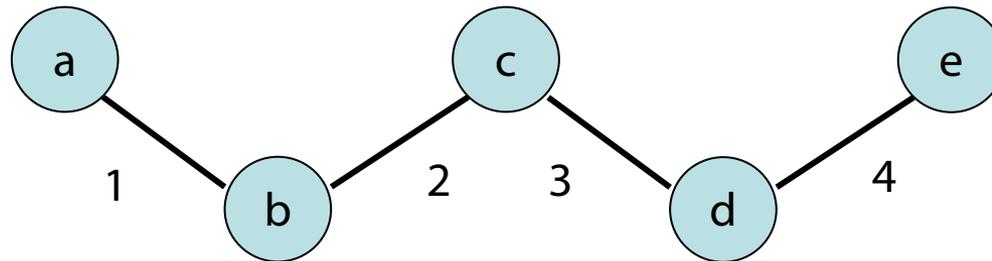
$$\begin{aligned} &O(n + T_{\text{Bellman-Ford}}(n, m) + m + n \cdot T_{\text{Dijkstra}}(n, m) + n^2) \\ &= O(n + n \cdot m + m + n(n \log n + m) + n^2) \\ &= O(n \cdot m + n^2 \log n) \end{aligned}$$

unter Verwendung von Fibonacci Heaps.

Da i. Allg. $m > n$, ist das sicher besser als
 $n \times \text{Bellman-Ford} \in O(n^2 \cdot m)$

Annahme: Ungerichteter Graph gegeben

Gegeben: Kantenkosten $w[i, j]$
Können wir **APSP** nicht besser hinkriegen?



```
for (i, j, cost) in edges(g)
  d[i, j] = cost
end
for k = 1:n
  for (i, j) in indices(d)
    d[i, j] = min( d[i, j], d[i, k] + d[k, j] )
  end
end
```

Annahme: Ungerichteter Graph gegeben

```
for (i, j, cost) in edges(g)
    d[i, j] = cost
end
for k = 1:n
    for (i, j) in indices(d)
        d[i, j] = min( d[i, j], d[i, k] + d[k, j] )
    end
end
```

Analyse: Zeit $O(n^3)$, Platz $O(n^2)$

- Wenn wir annehmen, dass $m > n$, ist das immer noch besser als $n \times$ Bellman-Ford $\in O(n^2 \cdot m)$
- Aber nicht besser als Johnson-Dijkstra: $O(n \cdot m + n^2 \log n)$

Robert W. Floyd: Algorithm 97 (SHORTEST PATH).

In: Communications of the ACM 5, 6, S. 345, 1962

und schon früher wurden ähnliche Verfahren veröffentlicht

Transitive Hülle / Erreichbarkeit

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

- $\text{Reachable}(v,w)$: liefert 1, falls es einen gerichteten Weg von v nach w in G gibt und sonst 0

Naives Verfahren für Erreichbarkeit

Algorithmus von Warshall

```
for k = 1:n
  for i = 1:n
    if d[i, k] == 1
      for j = 1:n
        if d[k, j] == 1
          d[i, j] = 1
        end
      end
    end
  end
end
```

Im Prinzip gleiche Idee wie APSP nach Floyd, daher auch Floyd-Warshall-Algorithmus genannt

Analyse: $O(n^3)$ **Das sollten wir doch besser hinkriegen?**

Transitive Hülle

Lösung 1: verwende **APSP** Algorithmus zur Erstellung einer Datenstruktur, aus der Ergebnis direkt ablesbar

- Laufzeit zur Erstellung der Datenstruktur:

$$O(n \cdot m + n^2 \log n)$$

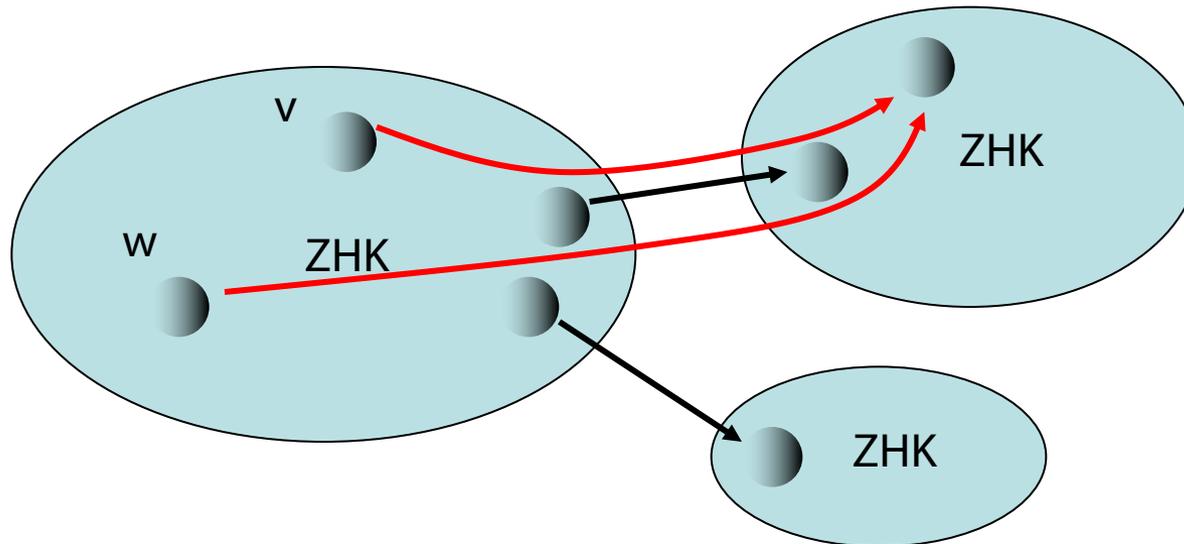
- Speicheraufwand: $O(n^2)$

- Laufzeit von $\text{Reachable}(v,w)$: $O(1)$

(Nachschauen in Tabelle, ob $\mu(v,w) < \infty$)

Transitive Hülle

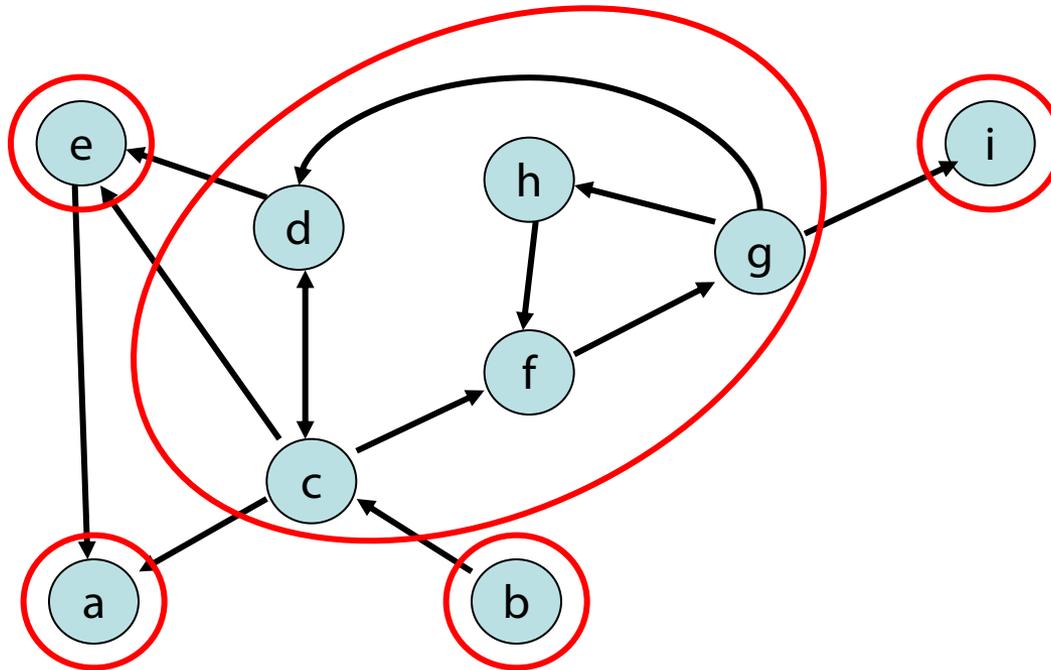
Einsicht: Alle Knoten in einer starken ZHK haben **dieselbe** Menge erreichbarer Knoten. Daher reicht es, sie durch Repräsentanten zu vertreten.



Transitive Hülle

Lösung 2: verwende **ZHK**-Algorithmus

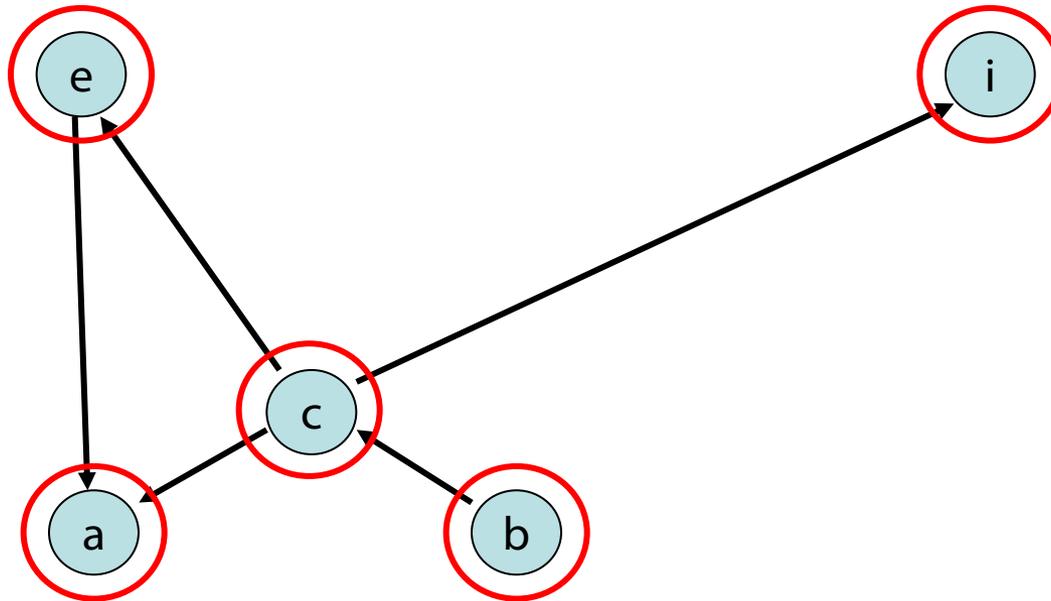
- Bestimme starke **ZHKs**



Transitive Hülle

Lösung 2: verwende **ZHK**-Algorithmus

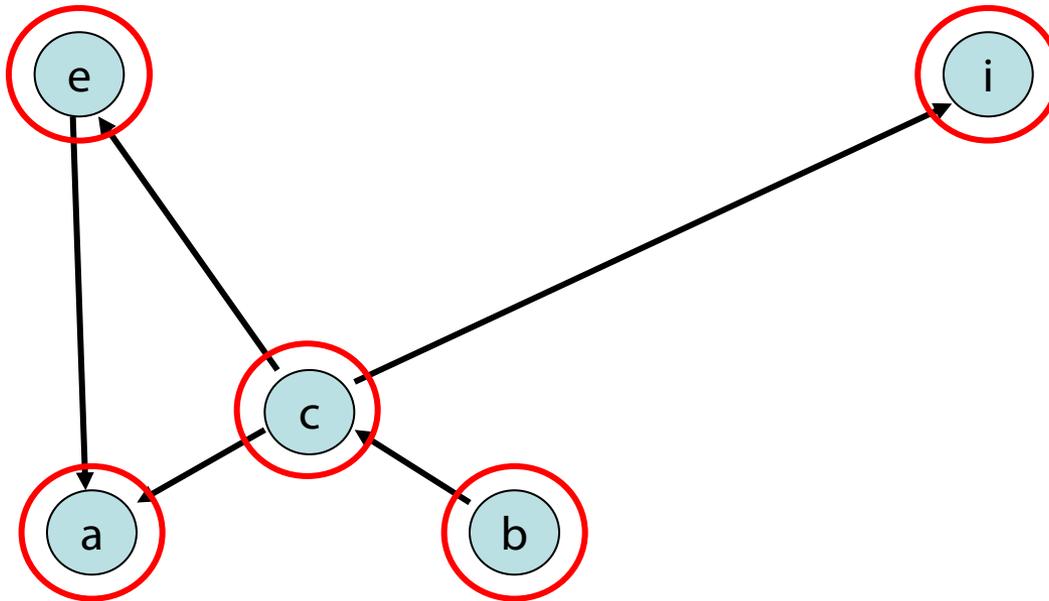
- Bestimme **ZHK**-Graph (Repräsentanten)



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

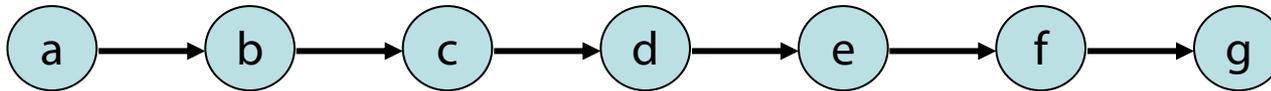
Ist es auch möglich, mit $\sim O(n+m)$ Speicher für die Datenstruktur die Operation $\text{Reachable}(v,w)$ effizient abzuarbeiten?

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

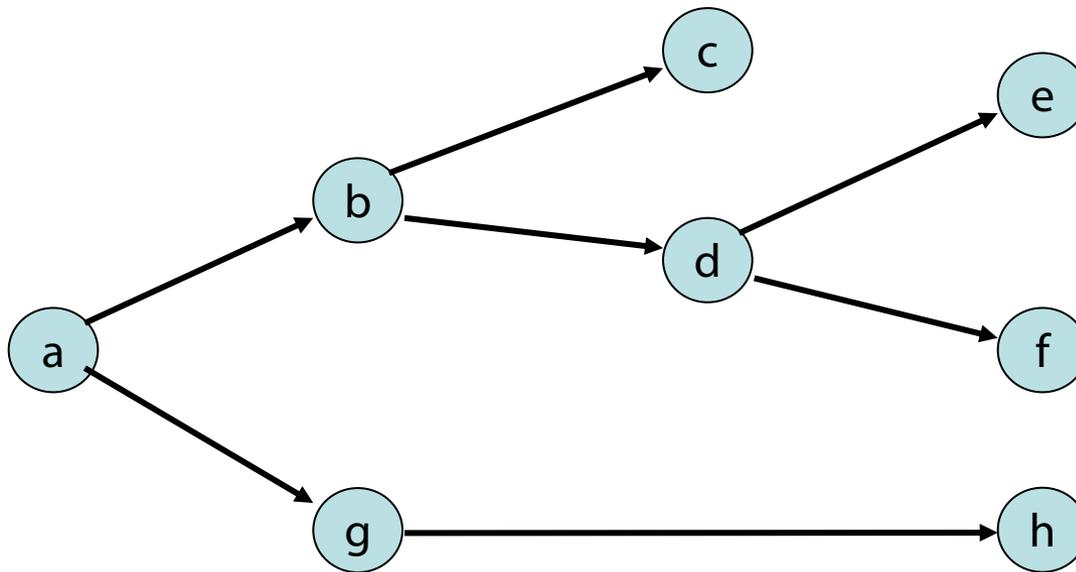
Datenstruktur: $O(n+m)$ Zeit, $O(n)$ Speicher

(speichere Repräsentanten zu jedem Knoten und gib
Repr. Ordnungsnummern)

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

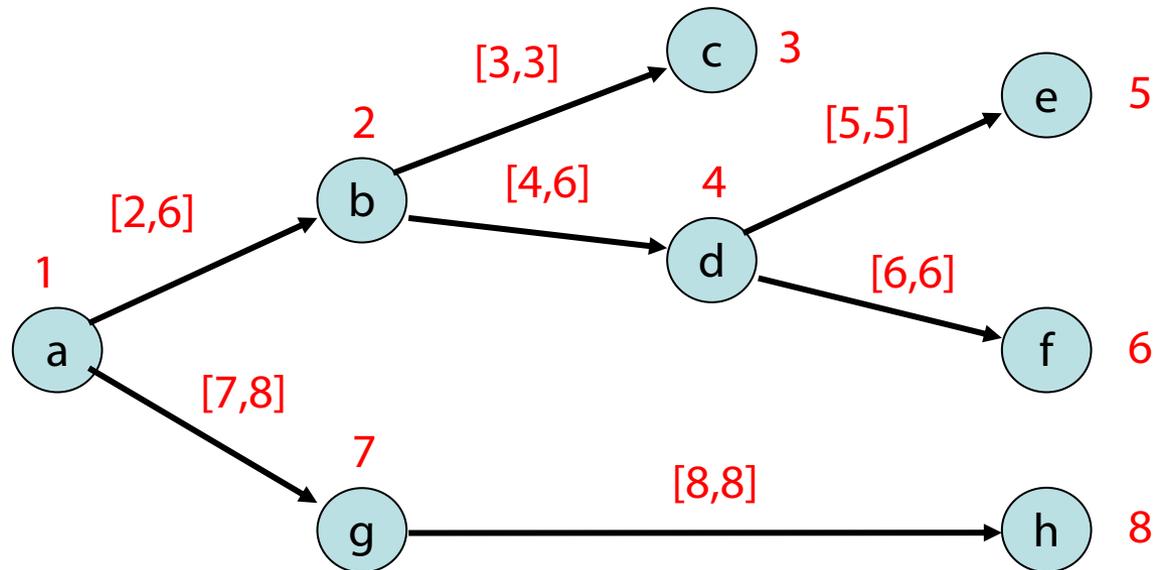
Transitive Hülle

Fall 2: Der ZHK-Graph ist ein gerichteter Baum



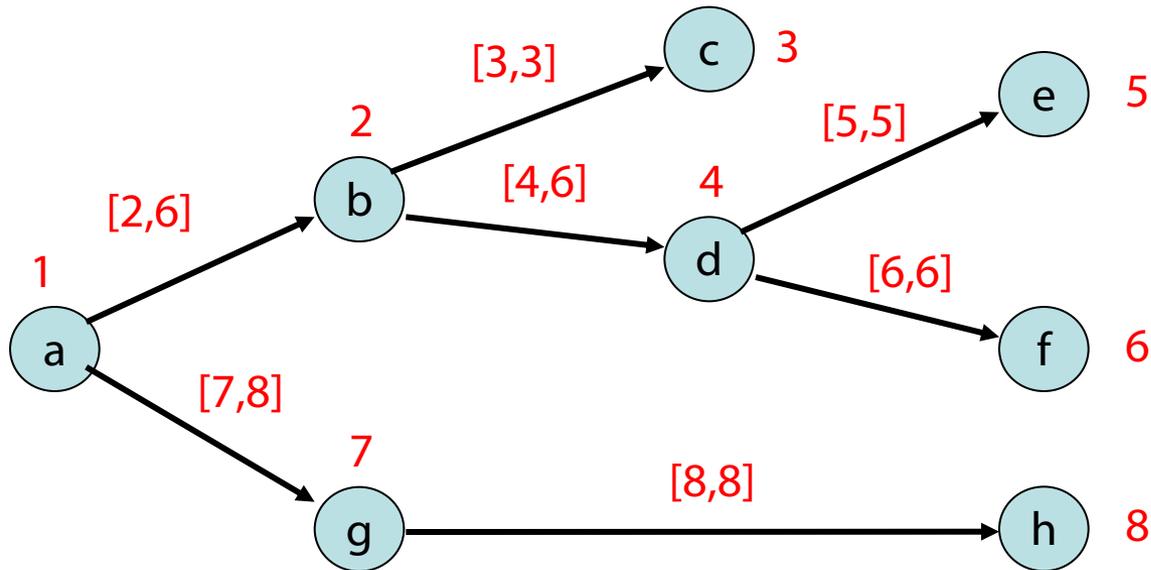
Transitive Hülle

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



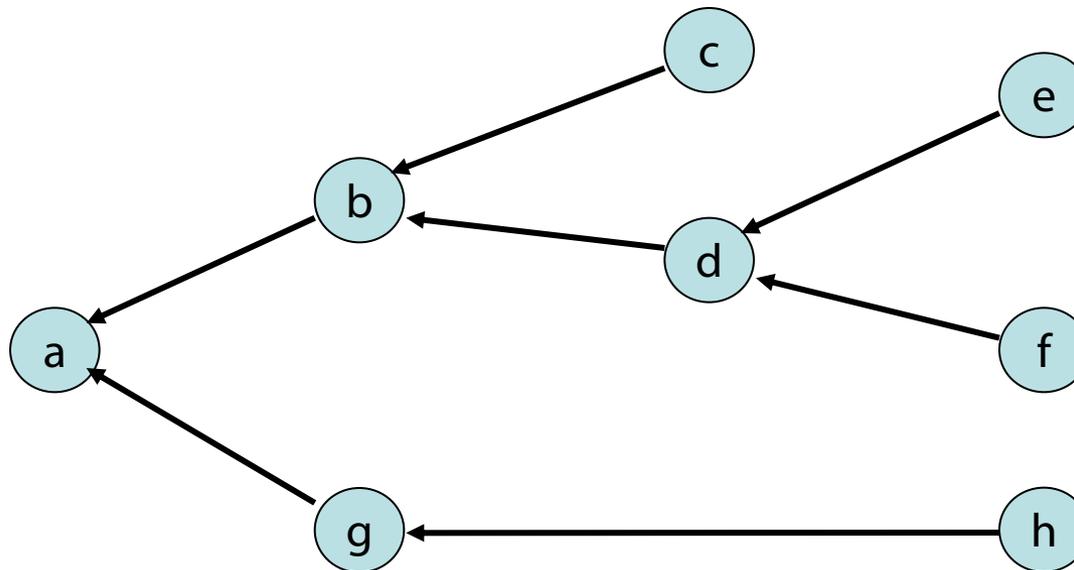
Transitive Hülle

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

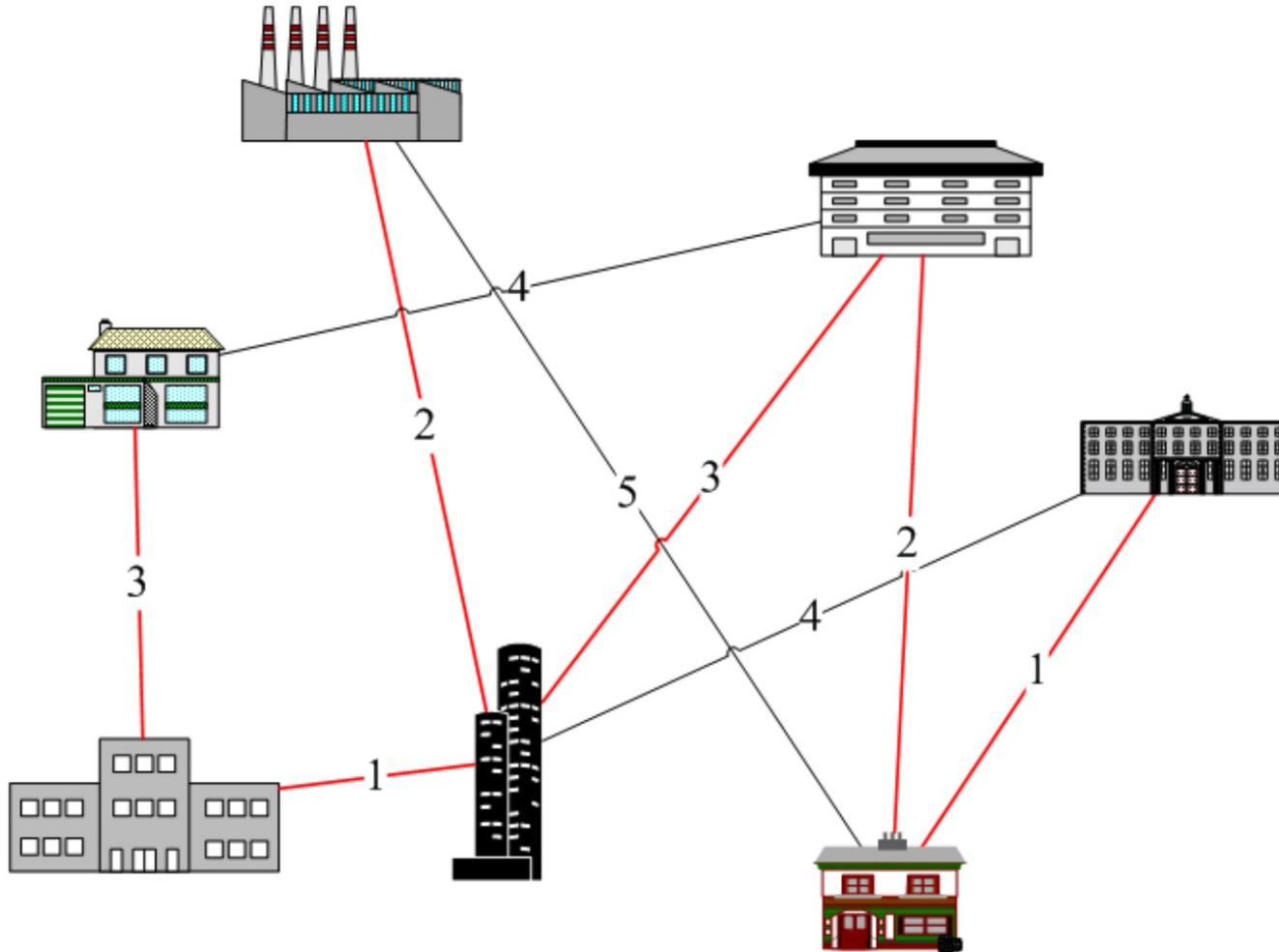
Datenstruktur: $O(n+m)$ Zeit und Speicher
(speichere Repräsentanten zu jedem Knoten
Kantenintervalle zu jedem Repräsentanten)

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

Fall 3: Der ZHK-Graph ist ein beliebiger DAG

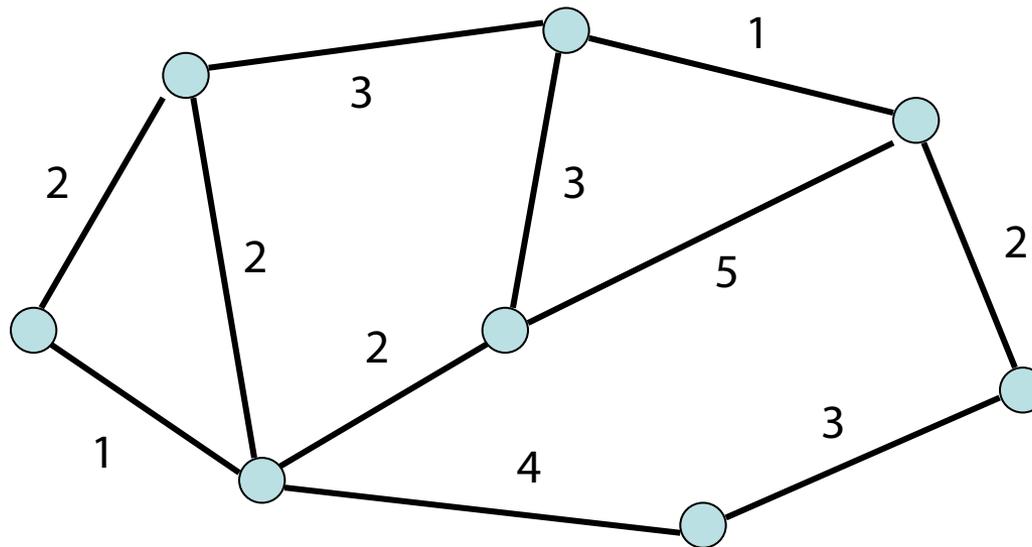
Geht in $O(d \log n)$ Zeit und $O(n^2)$ Speicher
(hier nicht vertieft)

Ein neues Anwendungsproblem



Minimaler Spannbaum

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



Anwendungen in der Praxis

- Erstellung von kostengünstigen zusammenhängenden Netzwerken
 - Beispielsweise Telefonnetze oder elektrische Netze
- Computernetzwerke mit redundanten Pfaden:
 - Spannbäume genutzt zum Routing und dabei zur Vermeidung von Paketverdopplungen

Minimaler Spannbaum

Eingabe:

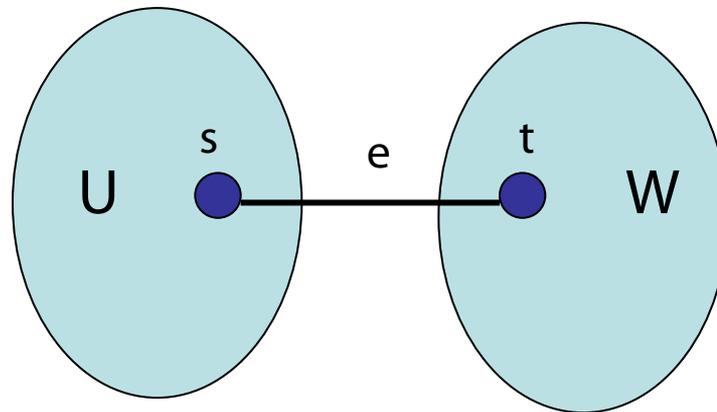
- ungerichteter Graph $G=(V,E)$
- Kantenkosten $c : E \rightarrow \mathbb{R}_+$

Ausgabe: $\operatorname{argmin}_{T \subseteq E \wedge (V,T) \text{ verbunden}} \sum_{e \in T} c(e)$

- Teilmenge $T \subseteq E$, so dass Graph (V,T) verbunden und $c(T)=\sum_{e \in T} c(e)$ minimal
- T formt **immer** einen Baum (wenn c positiv).
- Baum über alle Knoten in V mit minimalen Kosten: **minimaler Spannbaum (MSB)**

Minimaler Spannbaum

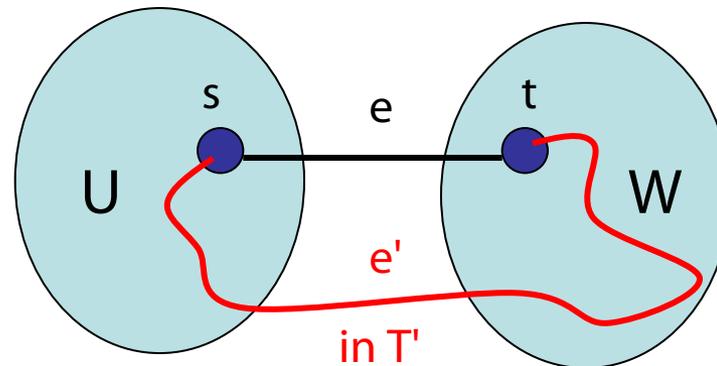
Beh 1: Sei (U,W) eine Partition von V (d.h. $U \cup W = V$ und $U \cap W = \emptyset$) und $e=\{s,t\}$ eine Kante mit minimalen Kosten mit $s \in U$ und $t \in W$. Dann gibt es einen minimalen Spannbaum (MSB) T , der e enthält.



Minimaler Spannbaum

Beweis von Beh 1:

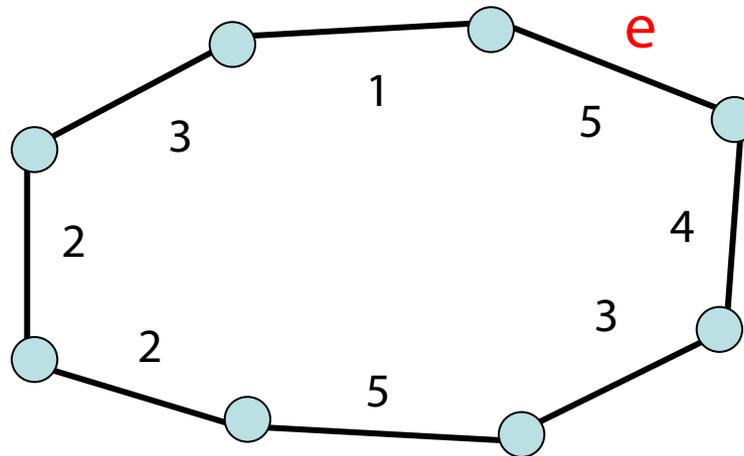
- Betrachte beliebigen MSB T'
- $e=\{s,t\}$: (U,W) -Kante minimaler Kosten



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

Minimaler Spannbaum

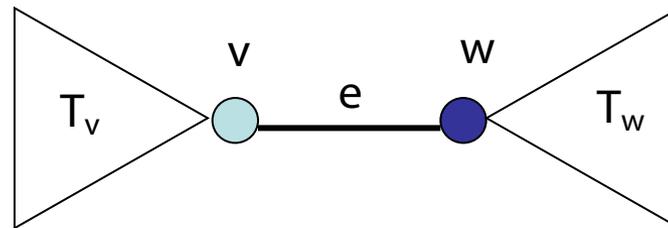
Beh 2: Betrachte beliebigen Kreis C in G und sei e Kante in C mit maximalen Kosten. Dann ist jeder MSB in G ohne e auch ein MSB in G .



Minimaler Spannbaum

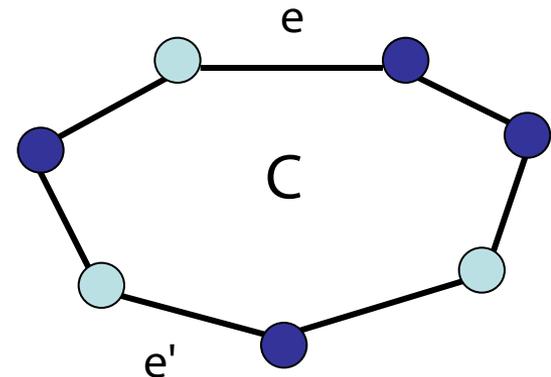
Beweis von Beh 2:

- Betrachte beliebigen MSB T in G
- Angenommen, T enthalte e



e maximal für C

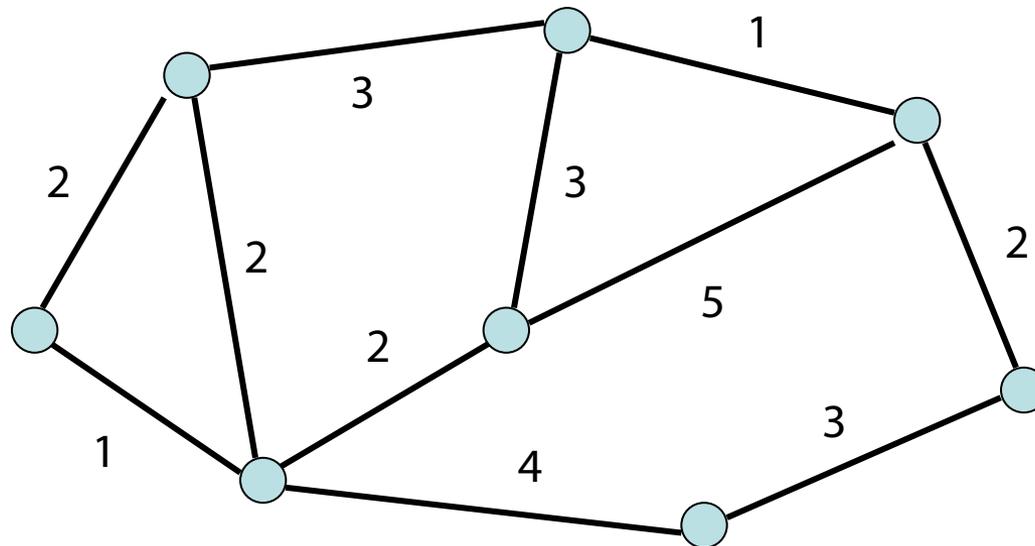
- \circ : zu T_v , \bullet : zu T_w
 - es gibt e' von T_v nach T_w
 - Ersetzung $e \rightarrow e'$ ergibt MSB T' ohne e



Minimaler Spannbaum

Regel aus Beh 2:

Lösche wiederholt Kante mit maximalen Kosten, die Zusammenhang nicht gefährdet, bis ein Baum übrig



Minimaler Spannbaum

Problem: Wie implementiert man die Regeln effizient?

Strategie aus Beh 1:

- Setze $T = \emptyset$ und sortiere die Kanten aufsteigend nach ihren Kosten
- Für jede Kante (u,v) in der sortierten Liste, teste, ob u und v bereits im selben Baum in T sind. Falls nicht, füge (u,v) zu T hinzu.

benötigt Union-Find DS

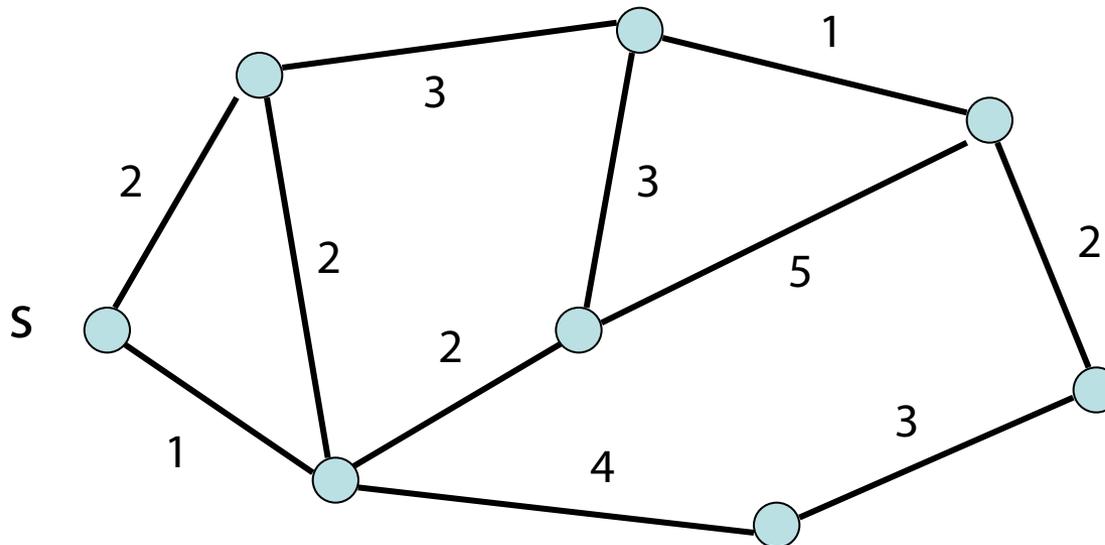
Erinnerung: Union-Find DS

Operationen:

- **Union(x_1, x_2):** vereinigt die Elemente in den Teilmengen T_1 und T_2 , zu denen die Elemente x_1 und x_2 gehören, zu $T = T_1 \cup T_2$
- **Find(x):** gibt (eindeutigen) Repräsentanten der Teilmenge aus, zu der Element x gehört

Minimaler Spannbaum

Beispiel: (—: Kanten im MSB)



Joseph Kruskal: On the shortest spanning subtree and the traveling salesman problem. In: Proceedings of the American Mathematical Society. 7, S. 48–50, 1956

Kruskal-Algorithmus

Laufzeit:

- Mergesort: $O(m \log m)$ Zeit
- $2m$ Find-Operationen und $n-1$ Union-Operationen:
 $O(m \cdot \log^* n)$ Zeit

Insgesamt Zeit $O(m \log m)$.

- Mit **Sortieren durch Verteilen** (Counting Sort, Bucket Sort...) weiter reduzierbar bei "kleinen" Graphen und Kantenkosten
- Dann dominiert $O(m \cdot \log^* n)$

Minimaler Spannbaum

Alternative Strategie (motiviert aus Beh 2):

- Starte bei beliebigem Knoten s , MSB T besteht anfangs nur aus s
- Ergänze T durch günstigste Kante zu äußerem Knoten w und füge w zu T hinzu bis T alle Knoten im Graphen umfasst

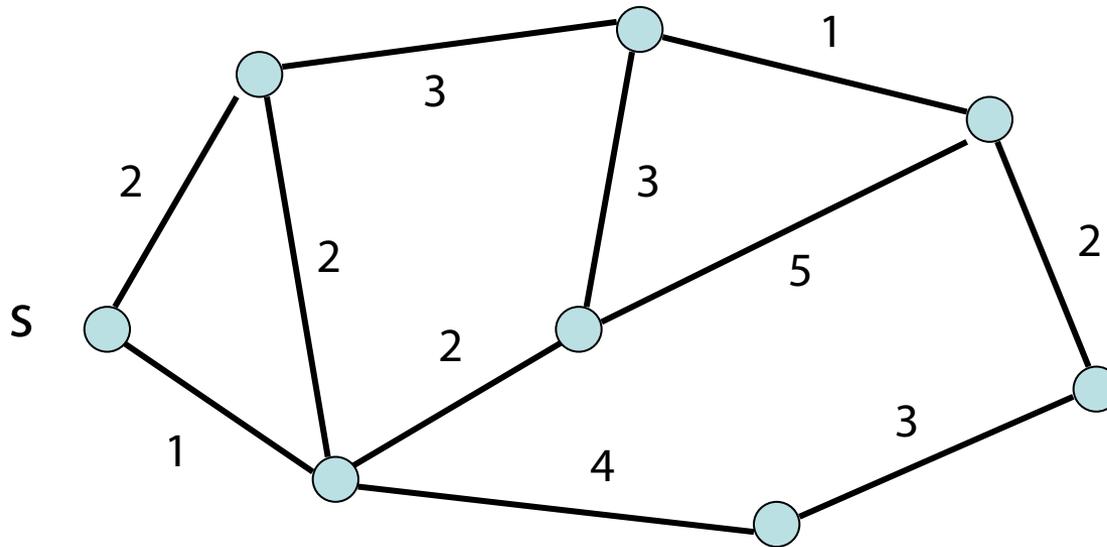
Jarník, V., "O jistém problému minimálním" [About a certain minimal problem], Práce Moravské Přírodovědecké Společnosti (in Czech) 6: S. 57–63, **1930**

Prim, R. C., "Shortest connection networks And some generalizations", Bell System Technical Journal 36 (6): S. 1389–1401, **1957**

Dijkstra, E. W., "A note on two problems in connexion with graphs", Numerische Mathematik 1: S. 269–271, **1959**

Minimaler Spannbaum

Beispiel:



Jarnik-Prim Algorithmus

Laufzeit:

$$T_{JP} = O(n(T_{\text{delete_min}}(n) + T_{\text{insert}}(n)) + m \cdot T_{\text{decrease_key}}(n))$$

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

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

Fibonacci Heap:

- $T_{\text{delete_min}}(n) = T_{\text{insert}}(n) = O(\log n)$
- $T_{\text{decrease_key}}(n) = O(1)$
- Damit $T_{JP} = O(n(\log n) + m)$

Vergleich: $O(m \log m)$ **bei Kruskal** ($m > n$)

Betrachtete Arten von Netzwerken

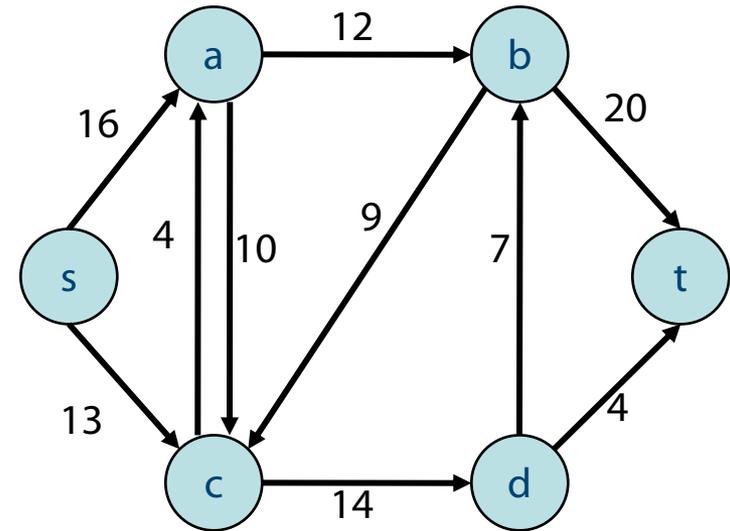
- Internet
- Telefonnetz
- Autobahnen/Eisenbahnnetz
- Elektrizitätsnetz
- Öl-/Gaspipelines
- Kanalisation
- ...

Netzwerke

- Gegeben: Gerichteter Graph $G=(V, E)$
 - Kanten repräsentieren Flüsse von Material/Energie/Daten/...
 - Jede Kante hat eine maximale Kapazität, dargestellt durch (totale) Funktion $c: E \rightarrow \mathbb{R}_+$
 - Knoten $s \in V$ als Quelle des Flusses
 - Knoten $t \in V$ als Senke des Flusses
- Ein Netzwerk ist ein Tupel (G, c, s, t) mit $s \in V$ und $t \in V$
- Die Funktion c macht G zum gewichteten Graphen
- Für jede Kante eines Netzwerks ist die Größe des Flusses steuerbar, dargestellt durch (totale) Funktion $f: E \rightarrow \mathbb{R}$

Problem des maximalen Flusses in Netzwerken

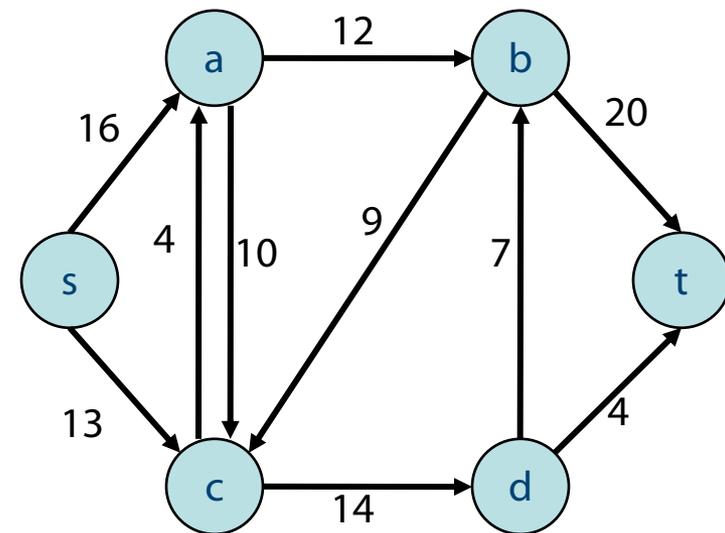
- Gegeben sei ein gerichteter gewichteter Graph
 - nicht-negative Gewichte
 - Gewichte repräsentieren Kapazität der Kanten (Funktion c)
- 2 ausgezeichnete Knoten s, t
 - s hat nur ausgehende Kanten
 - t hat nur eingehende Kanten
- Finde die **maximale Anzahl von Einheiten**, die von der Quelle zur Senke in diesem Graphen fließen kann
- Maximale Anzahl von Einheiten pro Einzelkante dargestellt durch Funktion $f_{\max}: E \rightarrow \mathbb{R}$



Jede Zahl steht für die Kapazität dieser Kante

Problem des maximalen Flusses in Netzwerken

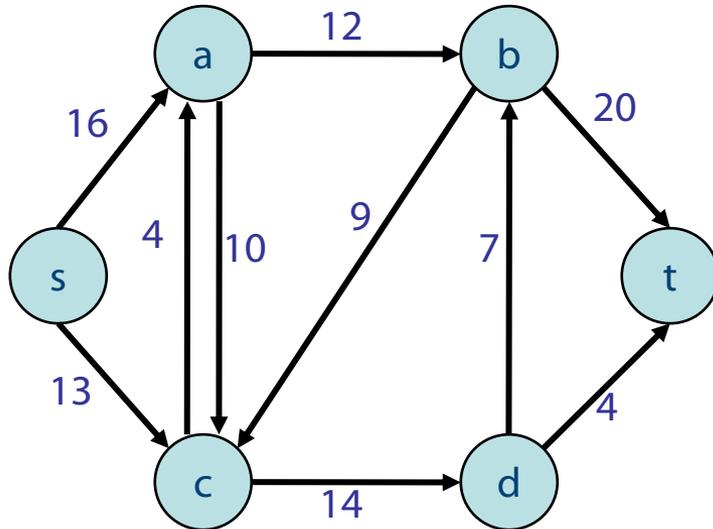
- Jede Kante könnte ein Wasserrohr darstellen
 - Von einer Quelle fließt Wasser zu einer Senke
 - Jedes Wasserrohr kann eine maximale Anzahl von Litern Wasser pro Sekunde transportieren



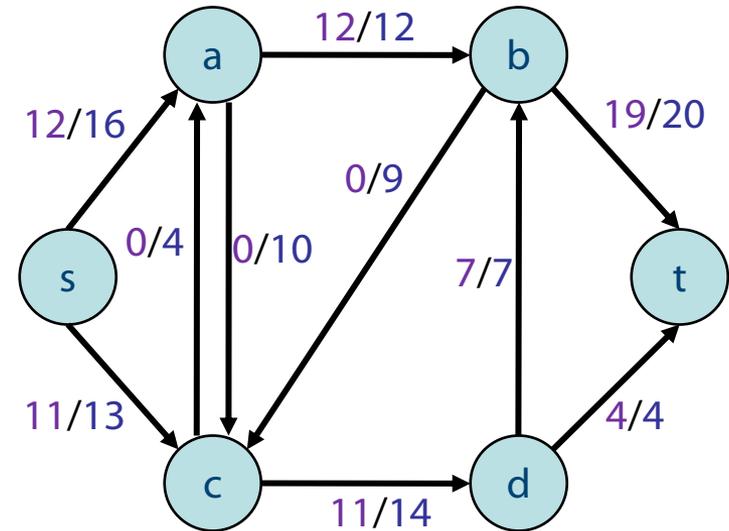
Jede Zahl steht für die Kapazität dieser Kante

- Wie viel Wasser pro Sekunde kann nun von **s** zu **t** maximal fließen?

Netzwerkfluss



Dieser Graph enthält die **Kapazitäten** jeder Kante im Graph (Beschriftung $c(e)$)

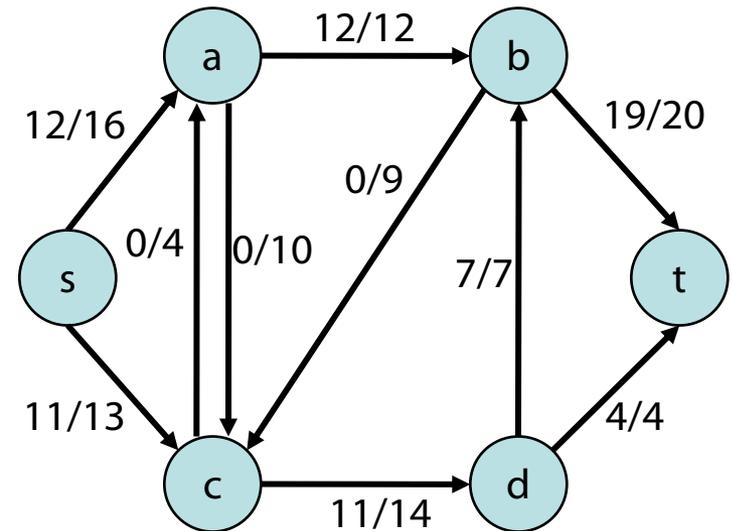


Dieser Graph enthält zusätzlich den **Fluss** im Graphen (Beschriftung $f(e)/c(e)$)

- Der Fluss des Netzwerkes ist definiert als der Fluss von der Quelle **s** (oder in die Senke **t**)
- Im Beispiel oben ist der Netzwerkfluss 23

Netzwerkfluss

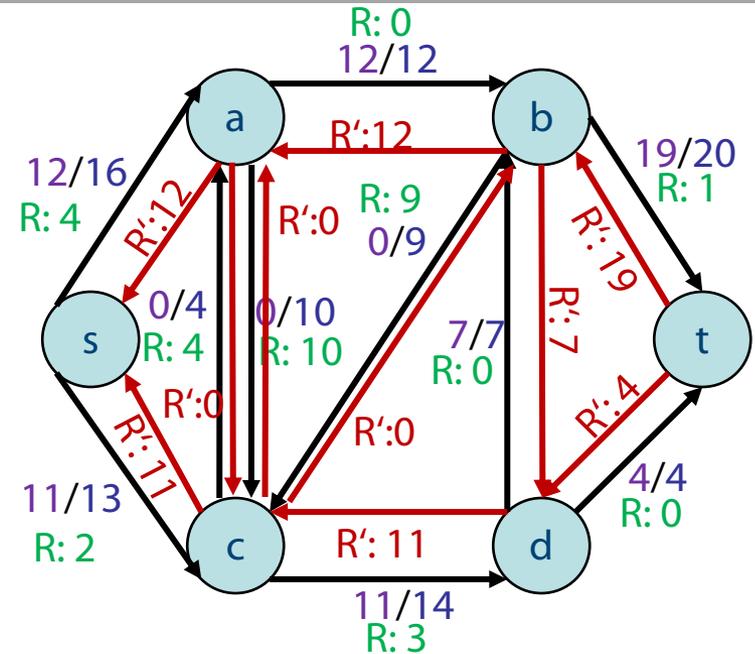
- Flusserhaltung:
 - Mit Ausnahmen der Quelle **s** und Senke **t** ist der Fluss, der in einen Knoten hineinfließt, genauso groß wie der Fluss, der aus diesem Knoten herausfließt
- Beachtung maximaler Kapazitäten:
 - Jeder Fluss in einer Kante muss kleiner oder gleich der Kapazität dieser Kante sein



Fluss / Kapazität im Graph

Netzwerkfluss

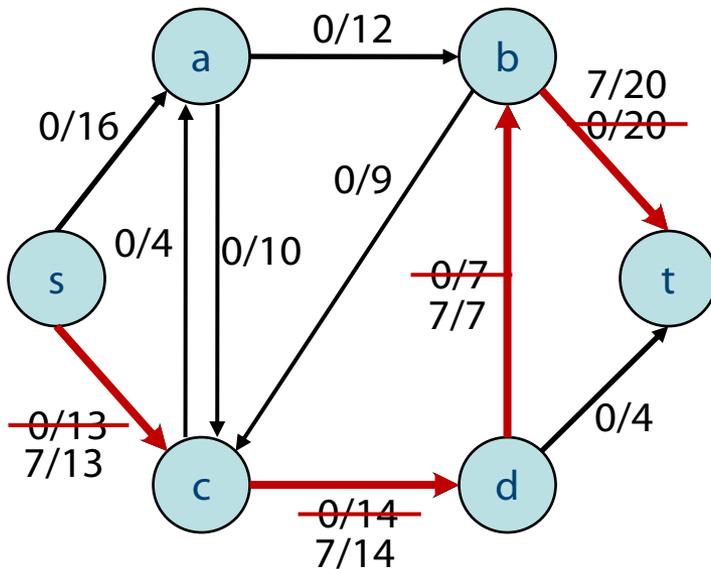
- Restkapazität einer Kante
 - Unbenutzte Kapazität jeder Kante
 - Zu Beginn ist der Fluss 0 und damit ist die Restkapazität genau so groß wie die Kapazität
 - Existiert ein Fluss, so kann der Fluss auch wieder reduziert werden, dies ist wie eine Restkapazität in die entgegengesetzte Richtung
- Restkapazität eines Pfades
 - Minimale Restkapazität aller Kanten entlang des Pfades
- Flusserhöhender Pfad
 - Pfad von der Quelle zur Senke mit Restkapazität größer als 0
 - Kann auch „Restkapazitäten in die entgegengesetzte Richtung“ beinhalten



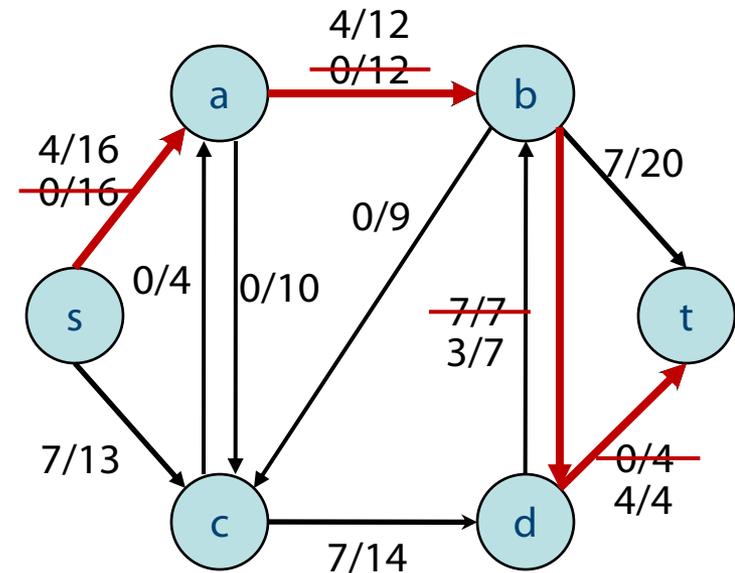
Fluss / Kapazität im Graph
 Restkapazität R: Kapazität – Fluss
 Restkapazität R' in die entgegengesetzte Richtung: Fluss

Beispiel für flusserhöhende Pfade

Flusserhöhender Pfad nur mit „normalen“ Restkapazitäten



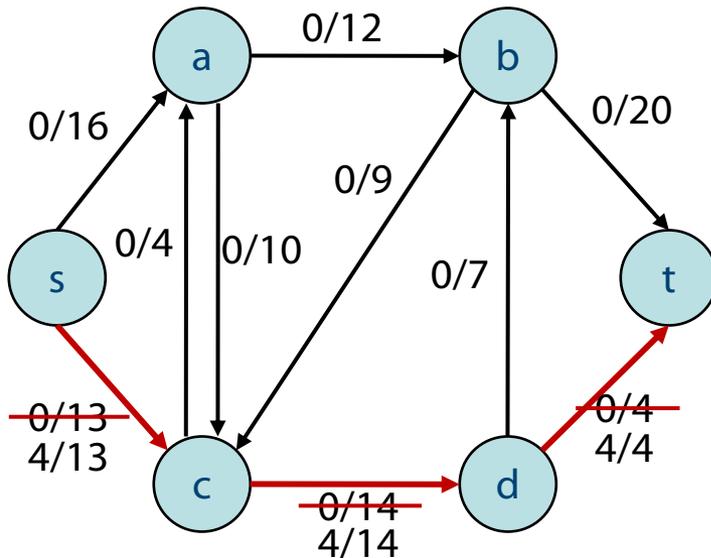
Flusserhöhender Pfad auch mit Restkapazitäten in die entgegengesetzte Richtung



Restkapazität ist auch für Pfade entsprechend definiert

Ford-Fulkerson Algo – Beispieldurchlauf

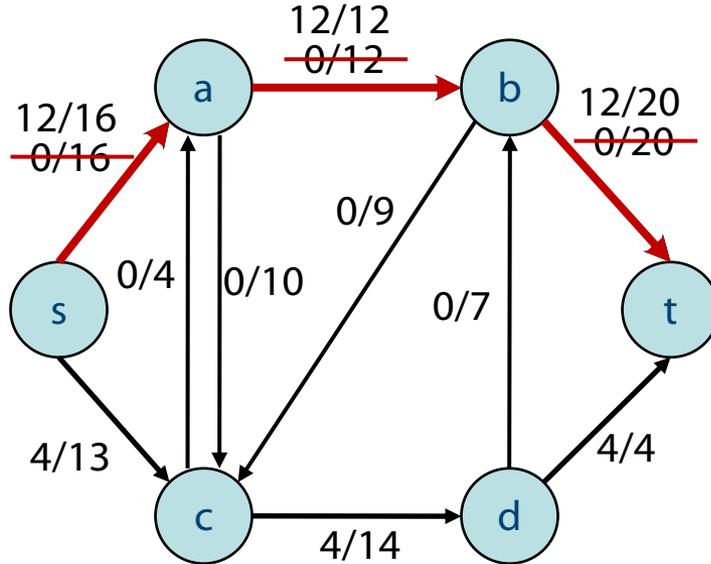
`while flow_augmenting_path(s, t, g, rest_caps) != nothing`
Erhöhe Fluss f von s nach t in p um Restkapazität von p



- Wähle flusserhöhenden Pfad, z.B. s, c, d, t
- Restkapazität dieses Pfades ist 4

Ford-Fulkerson Algo – Beispieldurchlauf

`while flow_augmenting_path(s, t, g, rest_caps) != nothing`
Erhöhe Fluss f von s nach t in p um Restkapazität von p



- Wähle anderen zunehmenden Pfad, z.B. s, a, b, t
- Restkapazität dieses Pfades ist 12

Pfadbestimmung z.B. mit Tiefensuche

Flusserhöhende Pfade

- Nutzung von Tiefensuche
- Nutzung von Breitensuche

Edmonds-Karp Algorithmus

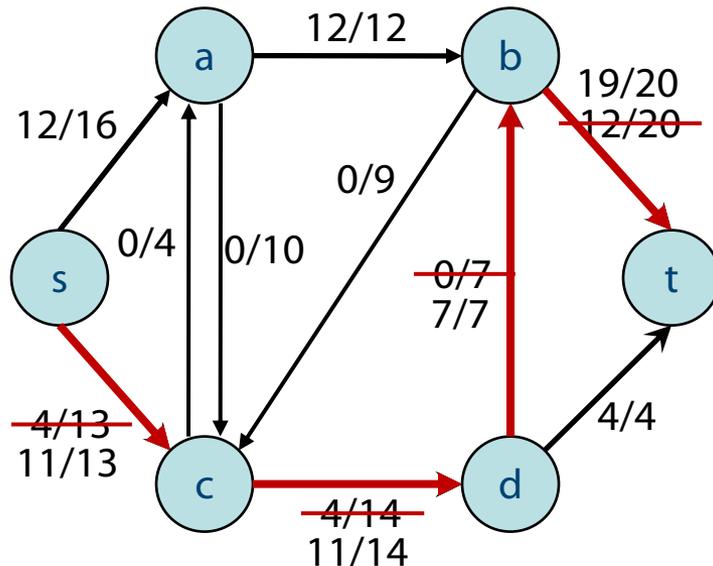
- Variation des Ford-Fulkerson Algorithmus durch Wählen von günstigen flusserhöhenden Pfaden
- Wähle als nächstes den flusserhöhenden Pfad mit einer minimalen Anzahl von Kanten

Jack Edmonds, Richard M. Karp: Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. In: J. ACM. 19, Nr. 2, S. 248-264, **1972**

Ford-Fulkerson Algo – Beispieldurchlauf

```
while flow_augmenting_path(s, t, g, rest_caps) != nothing
```

Erhöhe Fluss f von s nach t in p um Restkapazität von p

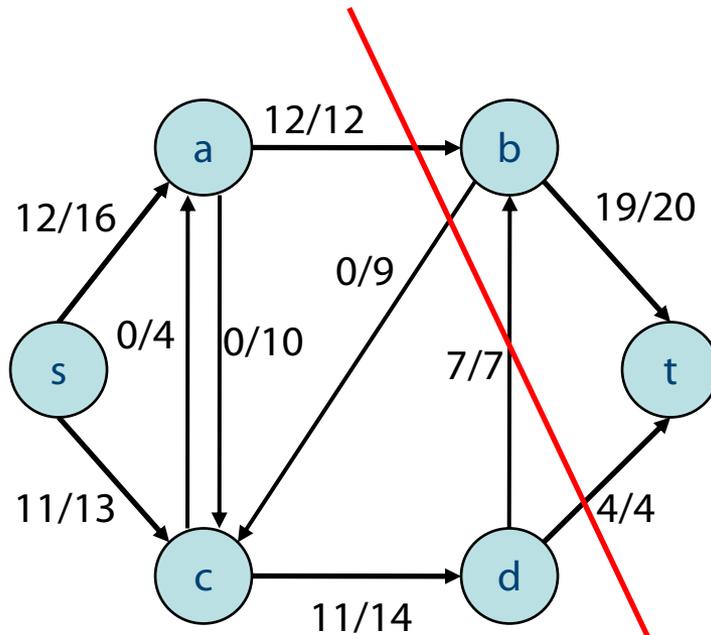


- Wähle anderen zunehmenden Pfad, z.B. s, c, d, b, t
- Restkapazität dieses Pfades ist 7

Ford-Fulkerson Algo – Beispieldurchlauf

```
while flow_augmenting_path(s, t, g, rest_caps) != nothing
```

Erhöhe Fluss f von s nach t in p um Restkapazität von p



- Gibt es weitere flusserhöhende Pfade?
Nein!
Fertig
- Maximaler Fluss:
 $19+4 = 23$ (bzw. $11+12$)

Minimaler Schnitt

Analyse des Algorithmus von Ford/Fulkerson

- Ein **Schnitt** in $N = ((V, E), c, s, t)$ ist eine disjunkte Zerlegung von V in Mengen $S \subseteq V$ und $T \subseteq V$ mit $s \in S, t \in T$.
- Die **Kapazität** des Schnittes ist $c(S, T) = \sum_{e \in E \cap (S \times T)} c(e)$
- Die **Kapazität** eines **minimalen Schnittes** ist
$$c_{\min} = \min_{(S, T) \text{ Schnitt in } N} c(S, T)$$
- Der **Fluss** eines Schnittes ist
$$f((S, T)) = \sum_{e \in E \cap (S \times T)} f(e) - \sum_{e \in E \cap (T \times S)} f(e)$$
- Mit f_{\max} bezeichnen wir den Wert eines **maximalen Flusses**

Max Flow/Min Cut-Theorem

- In jedem Netzwerk $N=(G, c, s, t)$ gilt: Der Wert eines jeden Flusses ist kleiner oder gleich der Kapazität eines jeden Schnittes. Insbesondere gilt: $f_{\max} \leq C_{\min}$.
- Sei f der vom F.F.-Algo für $N=(G, c, s, t)$ berechnete Fluss. Dann gibt es einen Schnitt (S,T) in N mit $f(G) = c(S,T)$.
- **Satz:** (Max Flow-Min Cut Theorem; Satz von Ford/Fulkerson)
Der Algorithmus von Ford/Fulkerson berechnet einen maximalen Fluss. In jedem Netzwerk gilt $f_{\max} = C_{\min}$.
(ohne formalen Beweis)

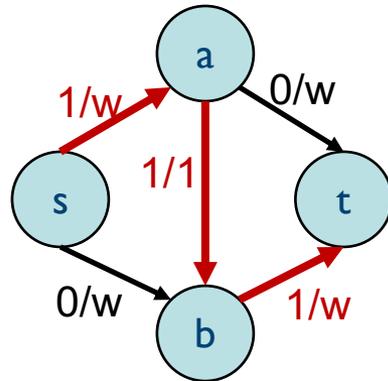
Der Wert eines maximalen Flusses ist gleich der Kapazität eines minimalen Schnittes.

Ford-Fulkerson Algorithmus – Analyse

- Finden eines flusserhöhenden Pfades z.B. mit einer Tiefensuche: $O(n + m)$
- Aber: Pfade können über Tiefensuche in einer ungünstigen Reihenfolge betrachtet werden

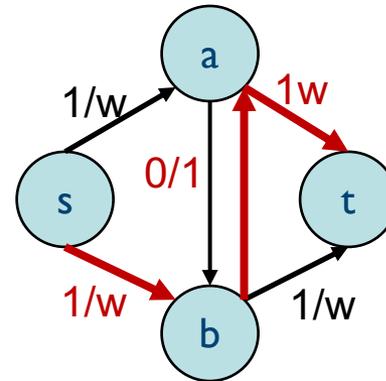
Schlechte Abfolge von zunehmenden Pfaden

1. flusserhöhender Pfad



2i. flusserh. Pfad

2. flusserhöhender Pfad



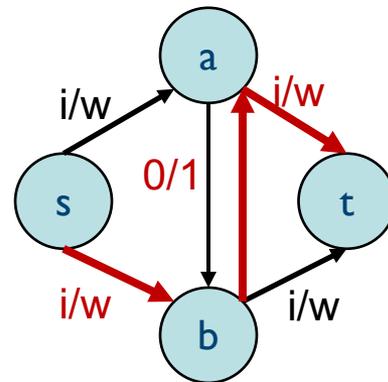
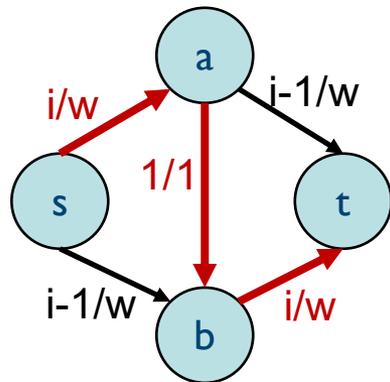
2w-1. flusserh. Pfad

...

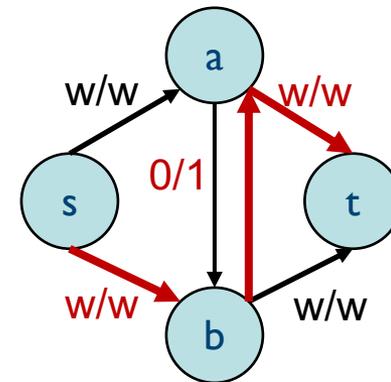
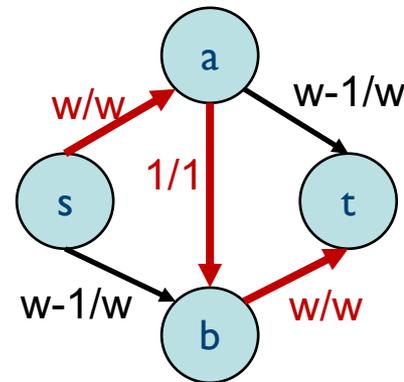
Fertig nach 2w flusserhöhenden Pfaden, obwohl der Algo auch schon mit 2 günstigen flusserhöhenden Pfaden fertig sein könnte!

2w. flusserh. Pfad

2i-1. flusserh. Pfad



...



Ford-Fulkerson Algorithmus – Analyse

Damit ergibt sich mit f_{\max} , dem maximalen Fluss von G , und der Verwendung von Tiefensuche als totale Laufzeit:

$$f_{\max}(G) \cdot O(n+m)$$

Da für die betrachteten G s gilt $m \gg n$ gilt, bekommen wir:

$$T_{\text{Ford-Fulkerson}}(G) \in f_{\max}(G) \cdot O(m)$$

Man beachte: Wenn wir die Zahl f_{\max} **binär codiert** als k -stelligen Bitvektor aus $\{0,1\}^k$ sehen, gibt es 2^k viele Erhöhungen von 0^k um 1, bis Wert f_{\max} erreicht

F.F. ist also in diesem Sinne **exponentiell** in der Länge k

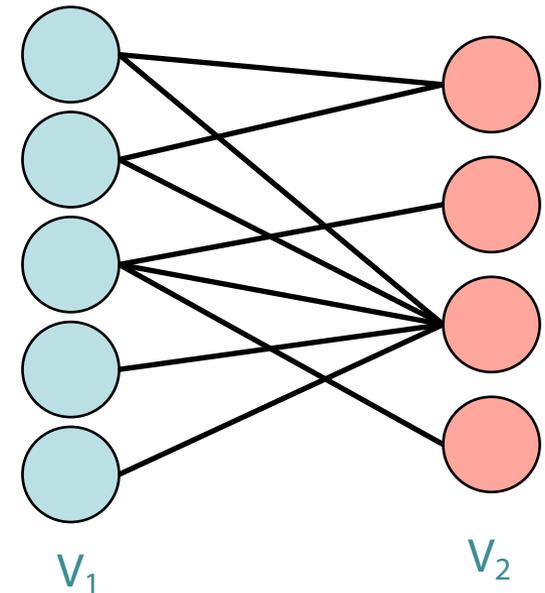
Edmonds-Karp Algorithmus – Analyse

- Maximale Anzahl von betrachteten flusserhöhenden Pfaden, und damit Schleifendurchläufen: $n \cdot m$
 - Ohne Beweis
- $T_{\text{Edmonds-Karp}}(n,m) \in O(n \cdot m^2)$
 - Berechnung des maximalen Flusses im Beispiel mit 2 flusserhöhenden Pfaden

Jack Edmonds, Richard M. Karp: Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. In: J. ACM. 19, Nr. 2, S. 248-264, 1972

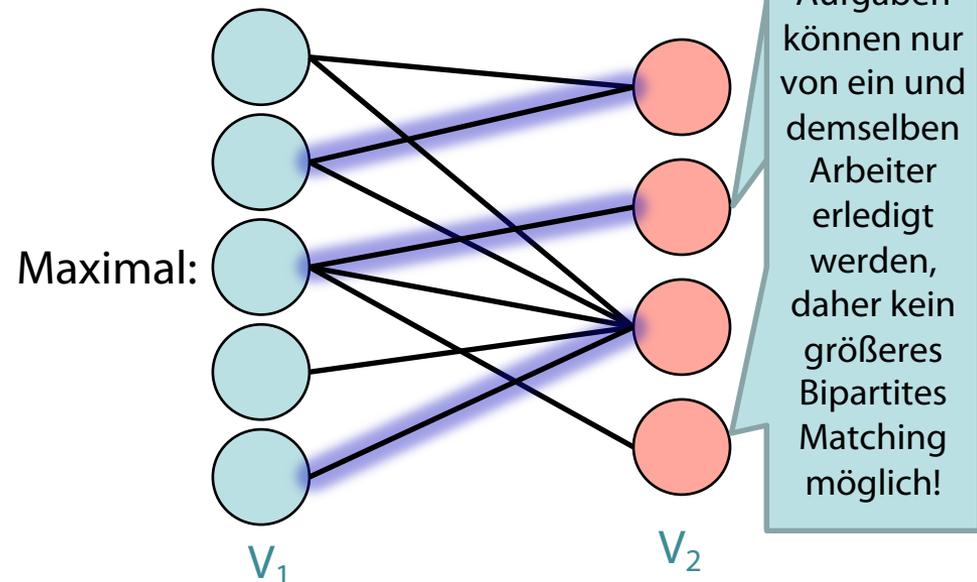
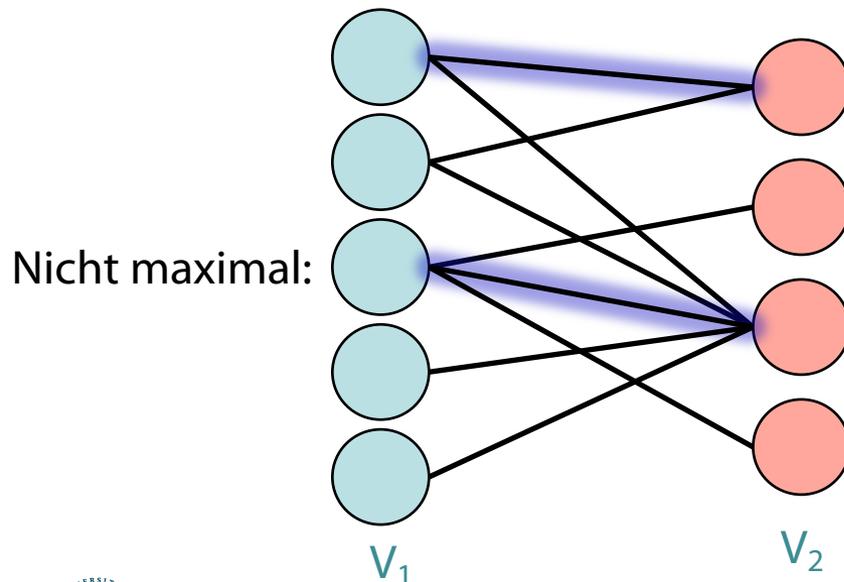
Anwendung: Maximale bipartite Matchings

- Bipartite Graphen sind Graphen $G=(V, E)$ in denen die Knotenmenge V in zwei disjunkte Knotenmengen V_1 und V_2 aufgeteilt werden können ($V = V_1 \cup V_2$), so dass $\forall (u, v) \in E: (u \in V_1 \wedge v \in V_2) \vee (u \in V_2 \wedge v \in V_1)$
- Beispiel eines bipartiten Graphen:
 - Knoten aus V_1 repräsentieren ausgebildete Arbeiter und
 - Knoten aus V_2 repräsentieren Aufgaben,
 - Kanten verbinden die Aufgaben mit den Arbeitern, die sie (bzgl. ihrer Ausbildung) ausführen können



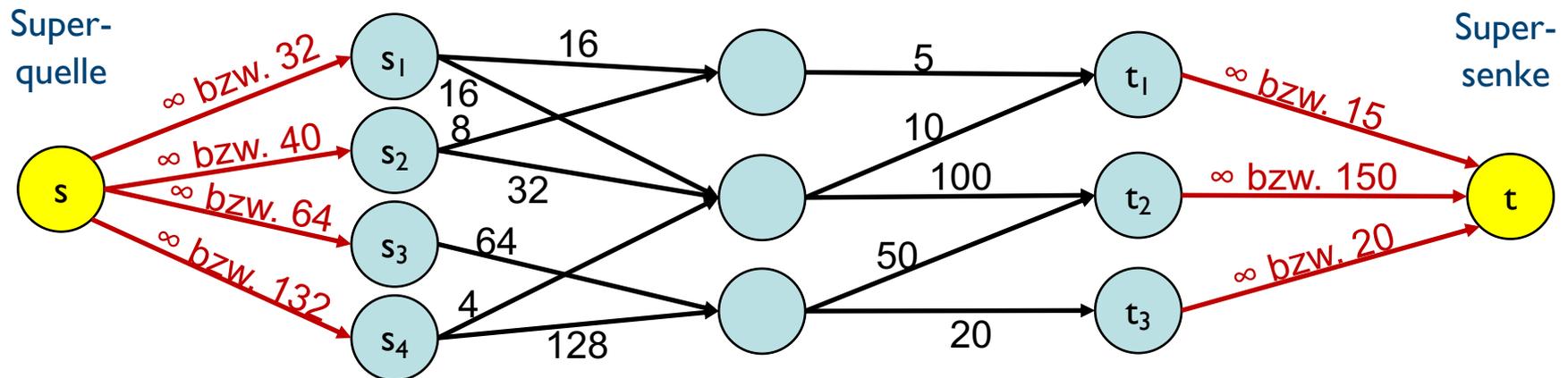
Bipartites Matching

- Finde $E' \subseteq E$, so dass $\forall v \in V: \text{degree}(v) \leq 1$ bezüglich E'
 - 1 Arbeiter kann zur selben Zeit nur 1 Aufgabe erledigen und 1 Aufgabe braucht nur max. von einem Arbeiter bearbeitet zu werden
- Maximales bipartites Matching: $|E'|$ maximal
 - maximale Aufteilung der Aufgaben
 - so wenig Aufgaben wie möglich bleiben liegen und
 - so wenig Arbeiter wie möglich sind unbeschäftigt



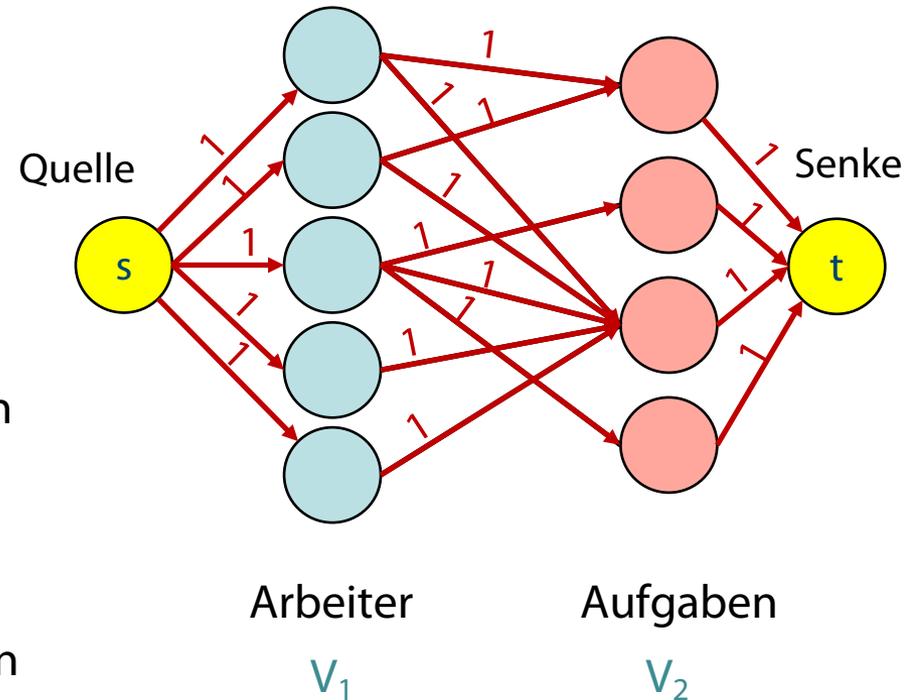
Mehrere Quellen und mehreren Senken

- Reduzierung auf maximalen Fluss in Netzwerk mit *einer* Quelle und *einer* Senke durch Einführung
 - einer Superquelle, die mit allen Quellen
 - einer Supersenke, die von allen Senkenmit einer Kante mit unbeschränkter Kapazität verbunden ist
 - Anstatt Kanten mit unbeschränkter Kapazität kann man auch Kanten mit der Kapazität der entsprechenden Quelle bzw. Senke verwenden

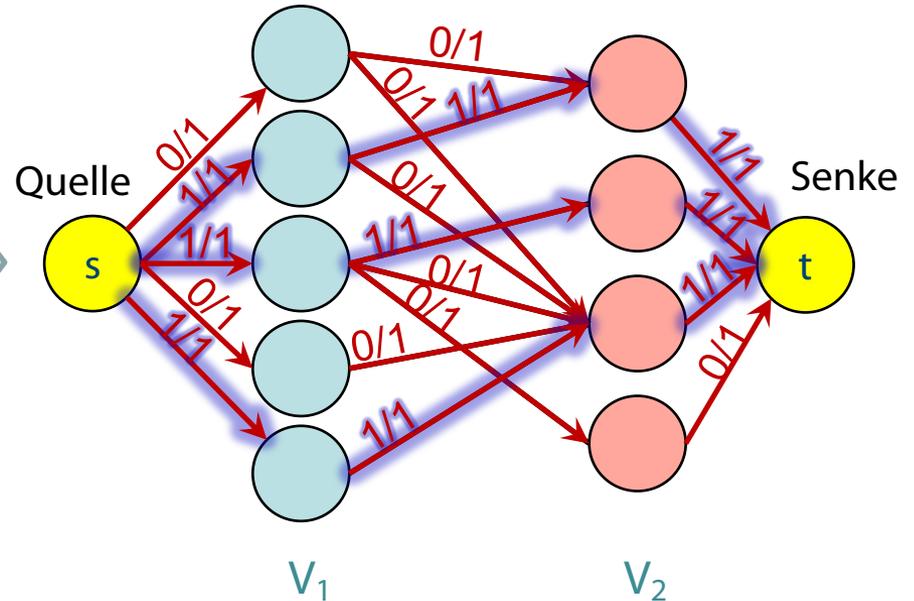
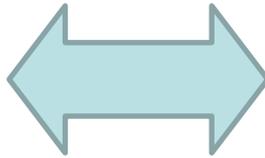
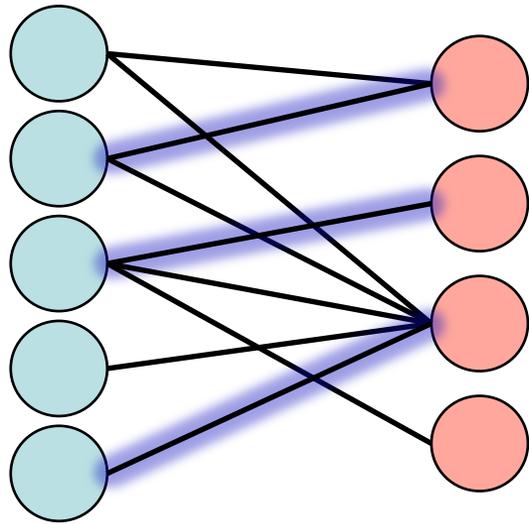


Lösung des maximalen Bipartiten Matchings

- Reduzierung auf das Problem des maximalen Flusses
 - Transformation des bipartiten Graphen auf einen Graphen für den Netzwerkfluss
 - Gerichtete Kanten von Knoten aus V_1 zu Knoten aus V_2 anstatt der ungerichteten Kanten des bipartiten Graphen
 - Einführung einer Quelle, die mit allen Knoten aus V_1 verbunden ist
 - Einführung einer Senke, die mit allen Knoten aus V_2 verbunden ist
 - Maximale Kapazität jeder Kante ist 1



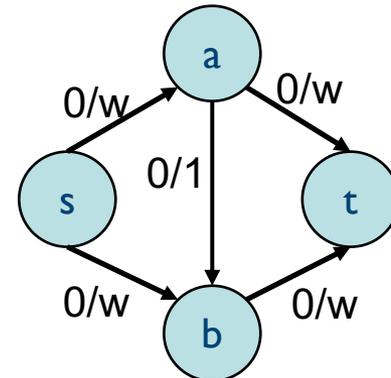
Maximaler Fluss \Leftrightarrow Maximales bipartites Matching



$$G=(V_1 \cup V_2, E)$$

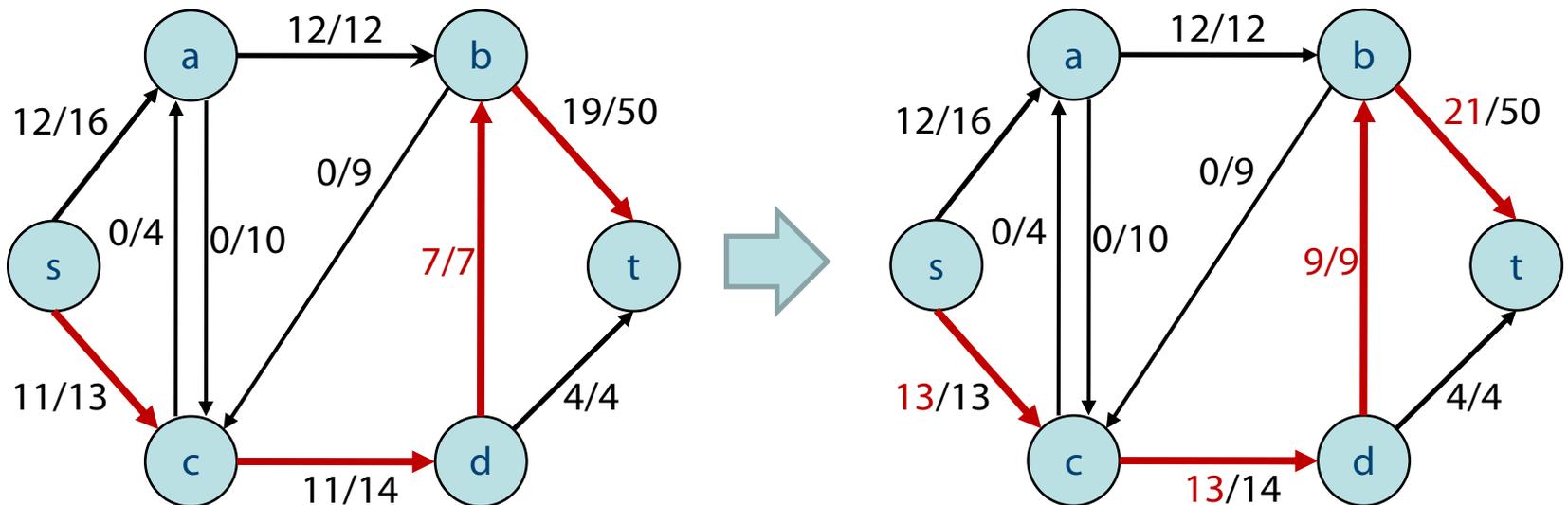
$T_{\text{bipartite-match}}(G) \in O(c \cdot (n+m))$

c bestimmt durch min der
Kantenkostensumme vom
Ausgang von s oder Eingang in t



Praktische Fragestellung

- Wie kann man durch Erhöhung der Kapazität an einer/wenigen Kanten den maximalen Fluss erhöhen?
 - Betrachte **Pfade** von der Quelle zu der Senke, deren Fluss die **volle Kapazität einer Kante ausnutzen**
 - Erhöhe die Kapazität der Kante(n), die die volle Kapazität ausnutzen, um das Minimum der Restkapazitäten der anderen Kanten des Pfades



Übersicht über Max-Flow-Algorithmen

($n=|V|$, $e=|E|$, $U=\max\{c(e)$ für alle $e \in E\}$)

Jahr	Autoren	Zeit gemessen in n, e, U	Zeit, wenn $e = \Omega(n^2)$
1969	Edmonds/Karp	$O(ne^2)$	$O(n^5)$
1970	Dinic	$O(n^2e)$	$O(n^4)$
1974	Karzanov	$O(n^3)$	$O(n^3)$
1977	Cherkasky	$O(n^2e^{1/2})$	$O(n^3)$
1978	Malhotra/Pramodh Kumar/ Maheshvari	$O(n^3)$	$O(n^3)$
1978	Galil	$O(n^{5/3}e^{2/3})$	$O(n^3)$
1978	Galil/Naamad sowie Shiloach	$O(ne \log^2 n)$	$O(n^3 \log^2 n)$
1980	Sleator/Tarjan	$O(ne \log n)$	$O(n^3 \log n)$
1982	Shiloach/Vishkin	$O(n^3)$	$O(n^3)$
1983	Gabow	$O(ne \log U)$	$O(n^3 \log U)$
1984	Tarjan	$O(n^3)$	$O(n^3)$
1985	Goldberg	$O(n^3)$	$O(n^3)$
1986	Goldberg/Tarjan	$O(ne \log(n^2/e))$	$O(n^3)$
1986	Ahuja/Orlin	$O(ne + n^2 \log U)$	$O(n^3 + n^2 \log U)$
1989	Ahuja/Orlin/Tarjan	$O(ne + n^2 \log U / \log \log U)$ $O(ne + n^2 \log^{1/2} U)$ $O(ne \log(\frac{n}{e} \log^{1/2} U + 2))$	
1989	Cheriyān/Hagerup (rand.) det. Version von Alon det. Version von Tarjan	$O(ne + n^2 \log^3 n)$ $O(\min(ne \log n, ne + n^{8/3} \log n))$ $O(\min(ne \log n, ne + n^2 \log^2 n))$	
1990	Cheriyān/Hagerup/Mehlhorn rand.	$O(n^3 / \log n)$ $O(\min(ne \log n, ne + n^2 \log^2 n, n^3 / \log n))$	

Zusammenfassung

- Probleme über Graphen zur **Modellierung von Anwendungsproblemen**
- Verständnis für Probleme über Graphen
 - Nicht immer muss man nach Wegen *suchen*...
 - Manchmal reicht die Betrachtung von polynomial berechenbaren Eigenschaften des Graphen, um ein Problem zu entscheiden