Algorithmen und Datenstrukturen

Prof. Dr. Ralf Möller

Universität zu Lübeck

Institut für Informationssysteme

Tanya Braun (Übungen)

sowie viele Tutoren

Danksagung

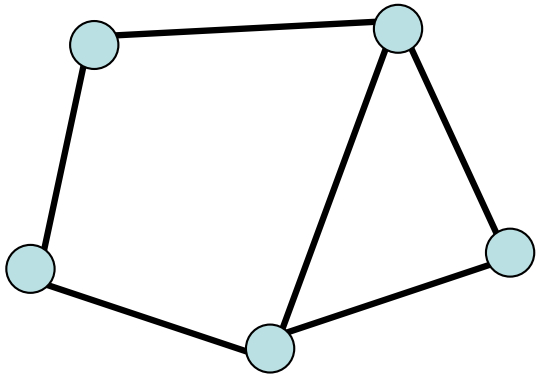
Die nachfolgenden Präsentationen wurden mit ausdrücklicher Erlaubnis des Autors übernommen aus:

- „Effiziente Algorithmen und Datenstrukturen“ (Kapitel 7,8,9) gehalten von Christian Scheideler an der TUM
  http://www14.in.tum.de/lehre/2008WS/ea/index.html.de

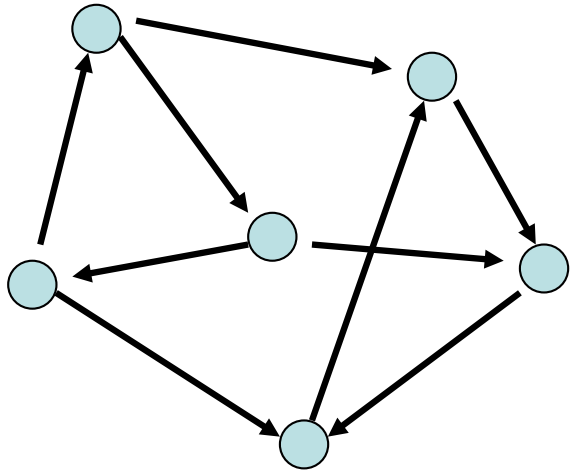
Graphen

Graph G=(V, E) besteht aus

- Knotenmenge V
- Kantenmenge E



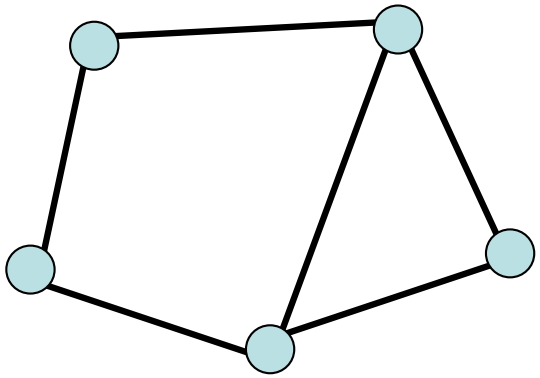
ungerichteter Graph



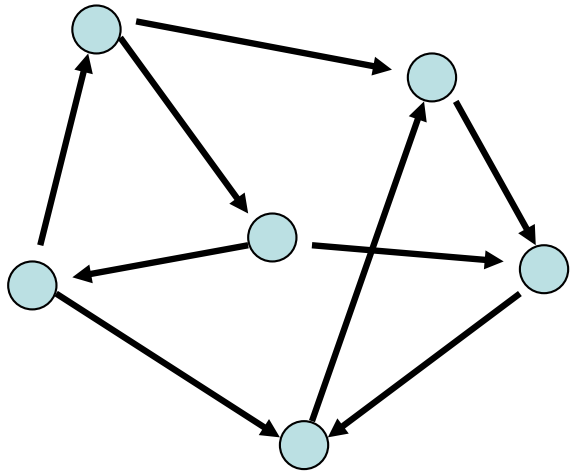
gerichteter Graph

Graphen

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



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.

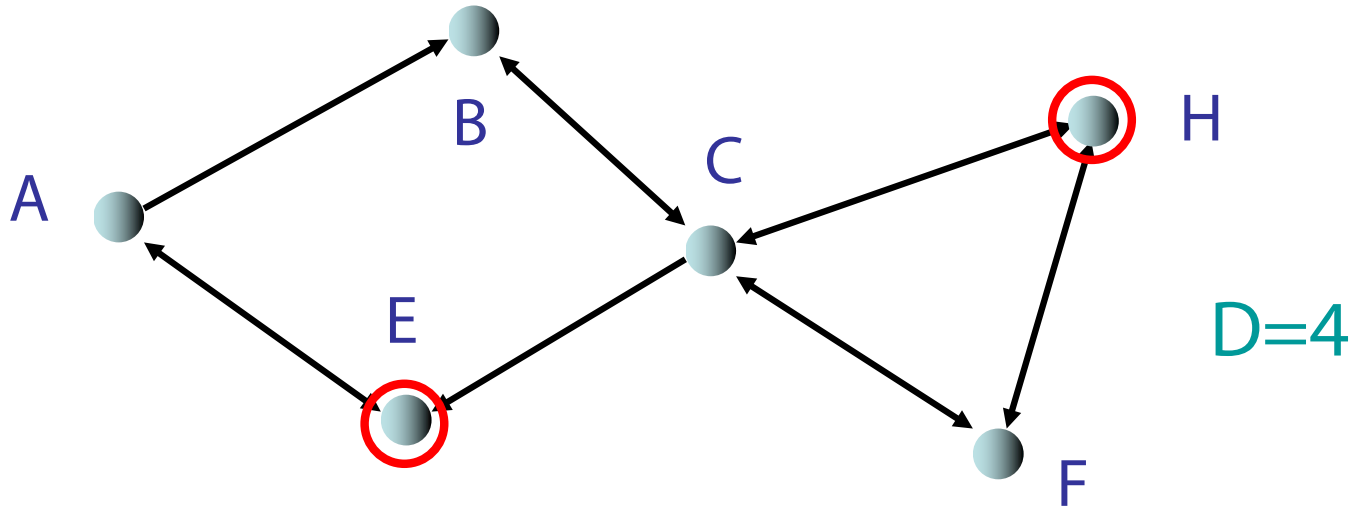
- n: aktuelle Anzahl Knoten
- m: aktuelle Anzahl Kanten

Graphen

- $\delta(v,w)$: Distanz

  Länge eines kürzesten gerichteten Weges
  von w zu v in G, ∞ wenn v von w nicht erreichbar

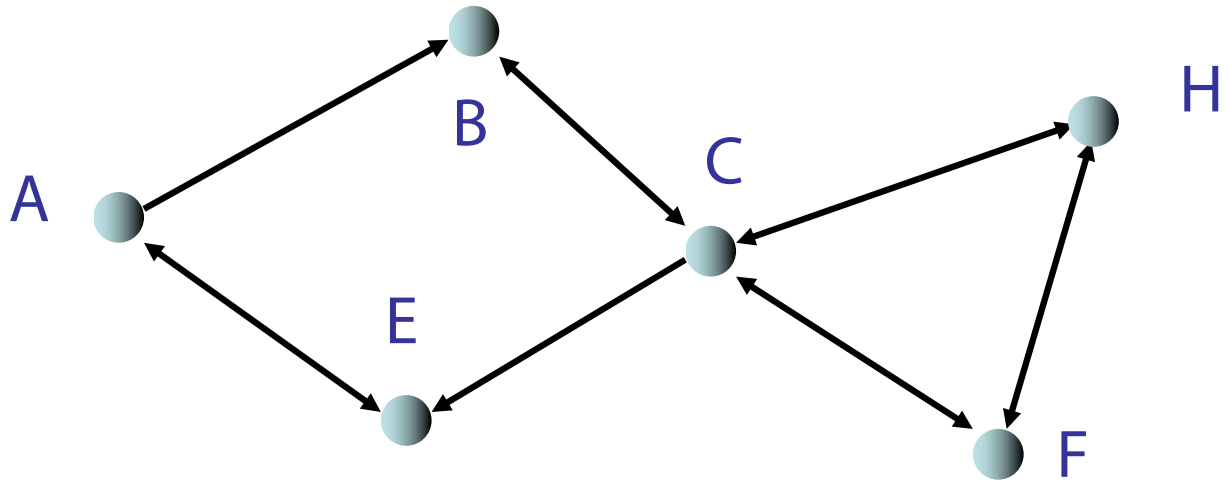
- $D=\max_{v,w} \delta(v,w)$: Durchmesser von G



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

Operationen auf Graphen

Anzahl der Knoten oft fest. In diesem Fall:

- V={1,…,n} (Knoten hintereinander nummeriert, identifiziert durch ihren Schlüssel aus {1,…,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 Keys von n Knoten in Bereich {1,…,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:

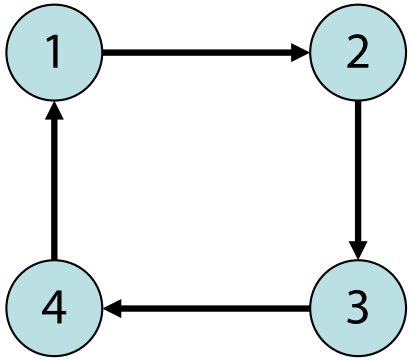
- n: Anzahl Knoten
- m: Anzahl Kanten
- d: maximaler Knotengrad (maximale Anzahl ausgehender Kanten von Knoten)

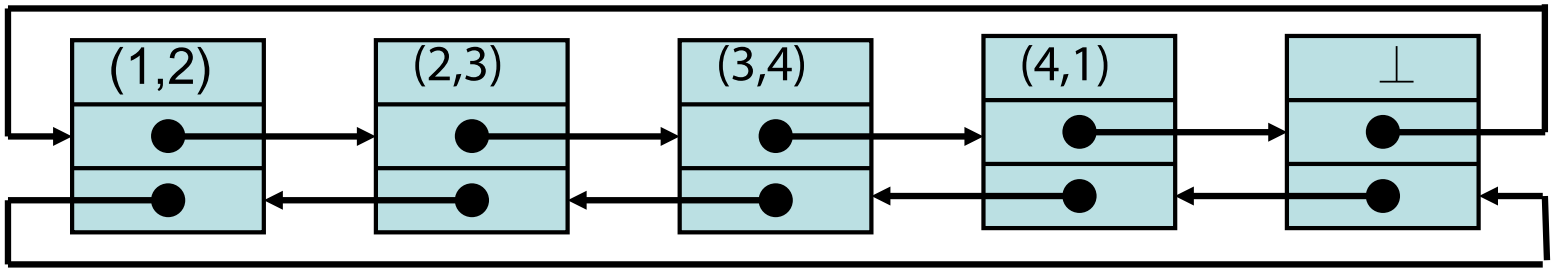
Graphrepräsentationen

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

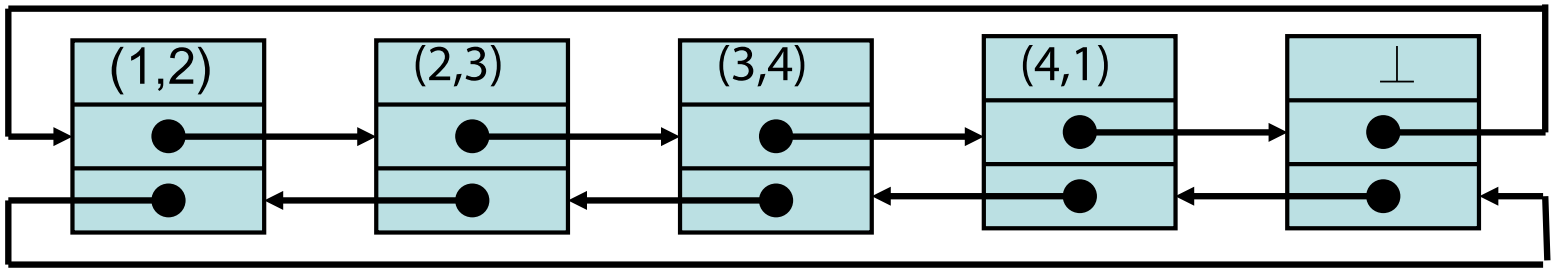
Graphrepräsentationen

1: Sequenz von Kanten





Sequenz von Kanten

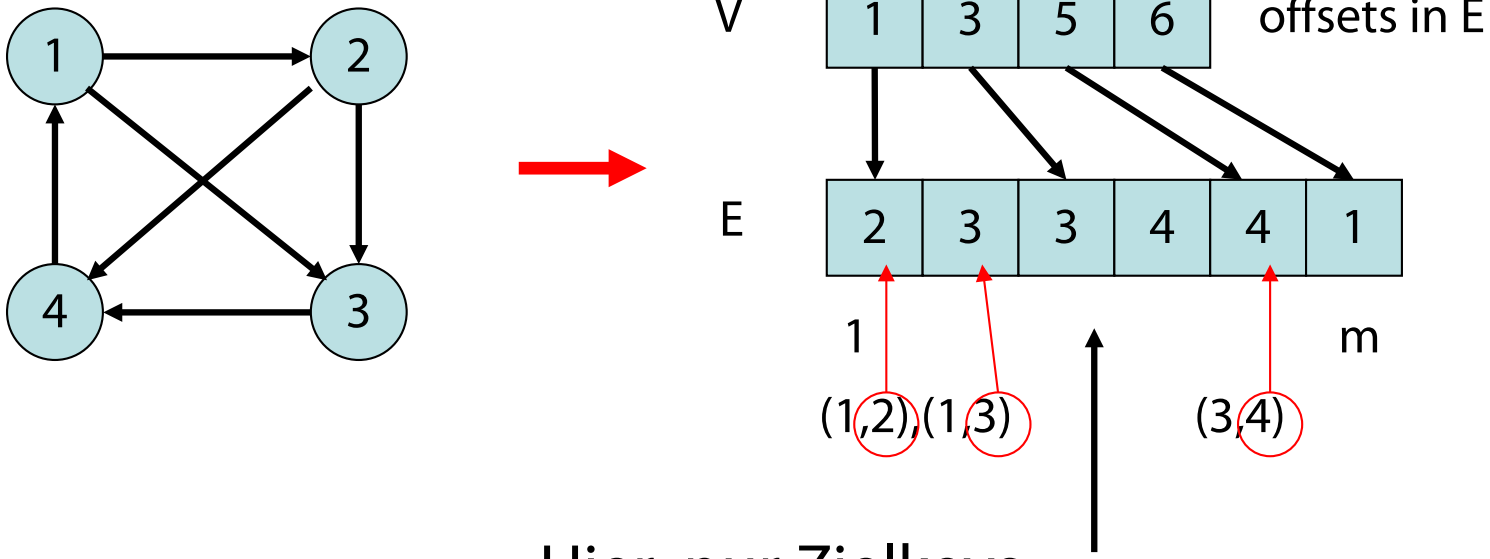


Zeitaufwand:

- find(i, j, G): Θ(m) im schlimmsten Fall
- insert(e, G): O(1)
- remove(i, j, G): Θ(m) im schlimmsten Fall

Graphrepräsentationen

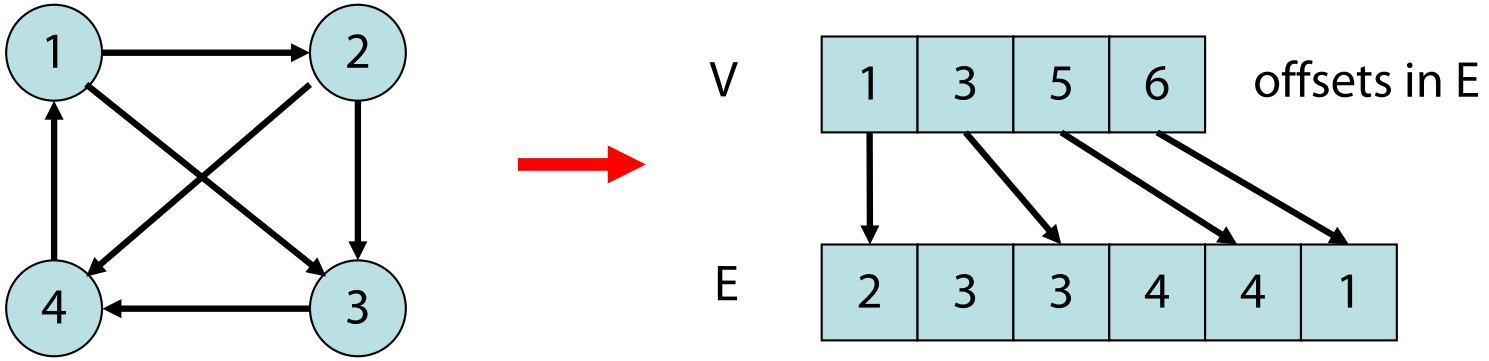
2: Adjazenzfeld



Hier: nur Zielkeys

In echter DS: E: Array [1..m] of Edge

Adjazenzfeld

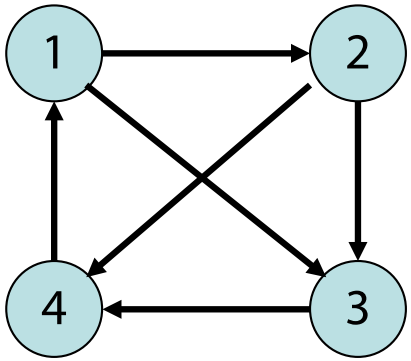


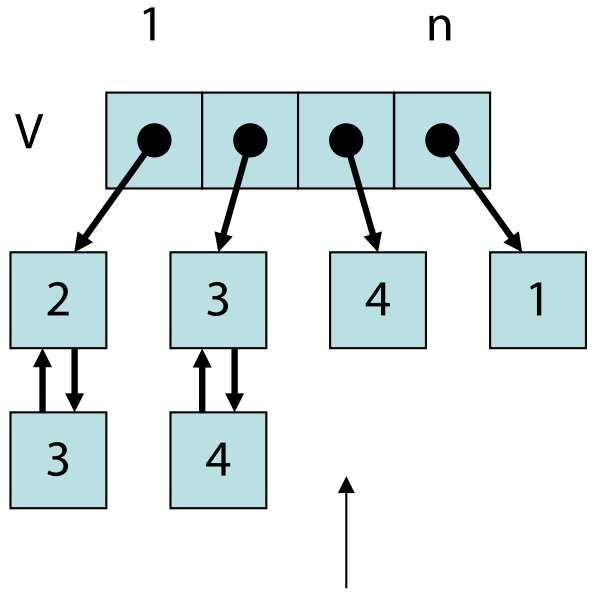
Zeitaufwand:

- 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

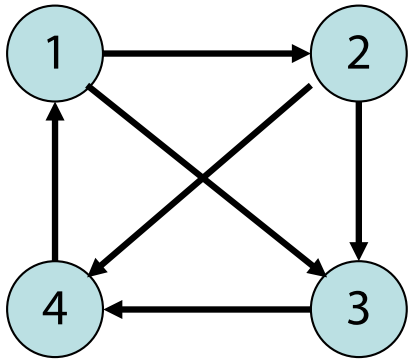


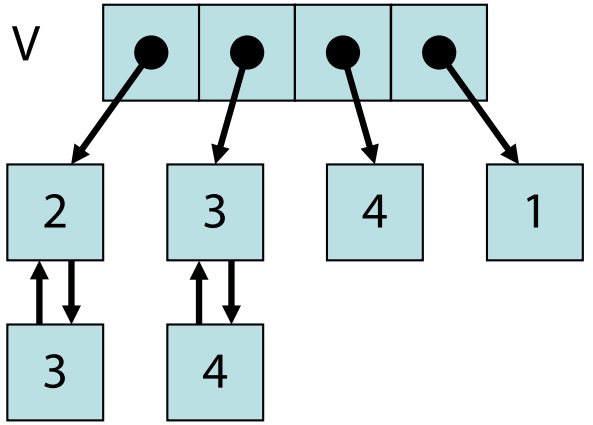


Hier: nur Zielkeys

In echter DS: V: Array [1..n] of List of Edge

Adjazenzliste





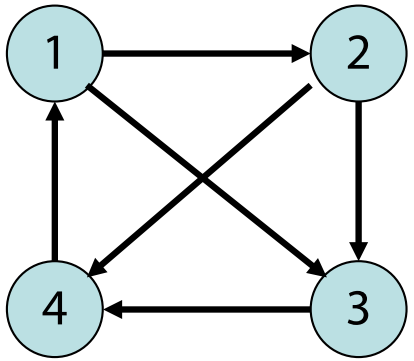
Zeitaufwand:

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

Problem: d kann groß sein!

Graphrepräsentationen

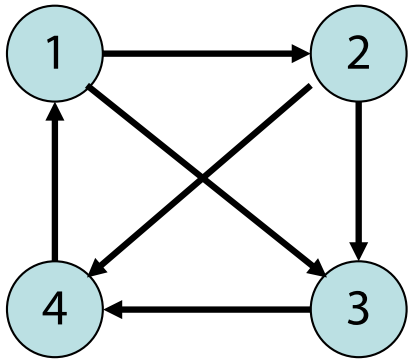
4: Adjazenzmatrix



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

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

Adjazenzmatrix



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

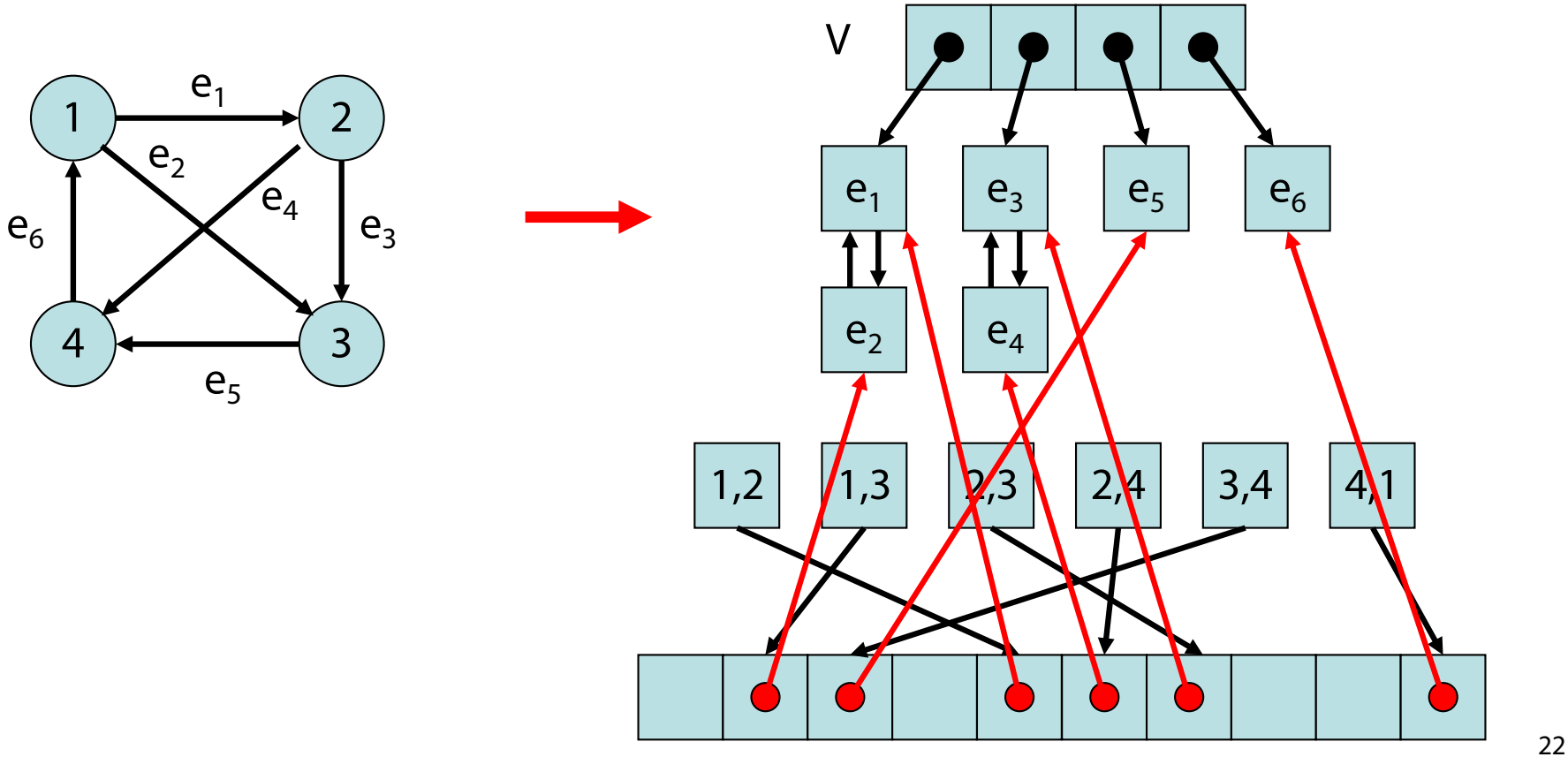
Zeitaufwand:

- 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

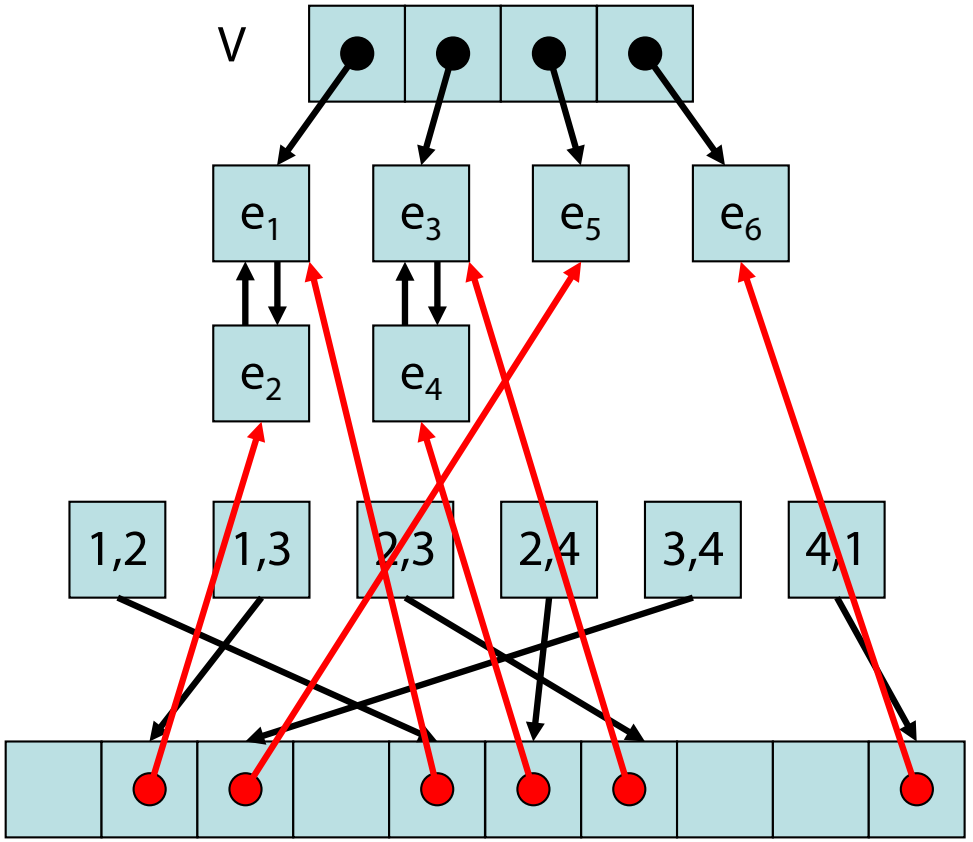
5: Adjazenzliste + Hashtabelle



Adjazenzliste+Hashtabelle

Zeitaufwand (grob):

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



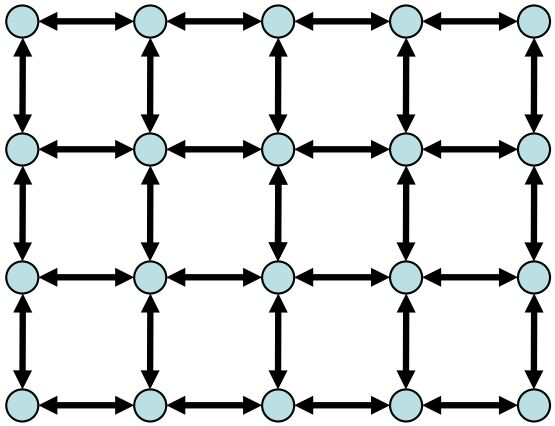
Graphrepräsentationen

6: Implizite Repräsentationen

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

- $V=[k] \times [l]$ ($[a]=\{0,\ldots,a-1\}$ 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

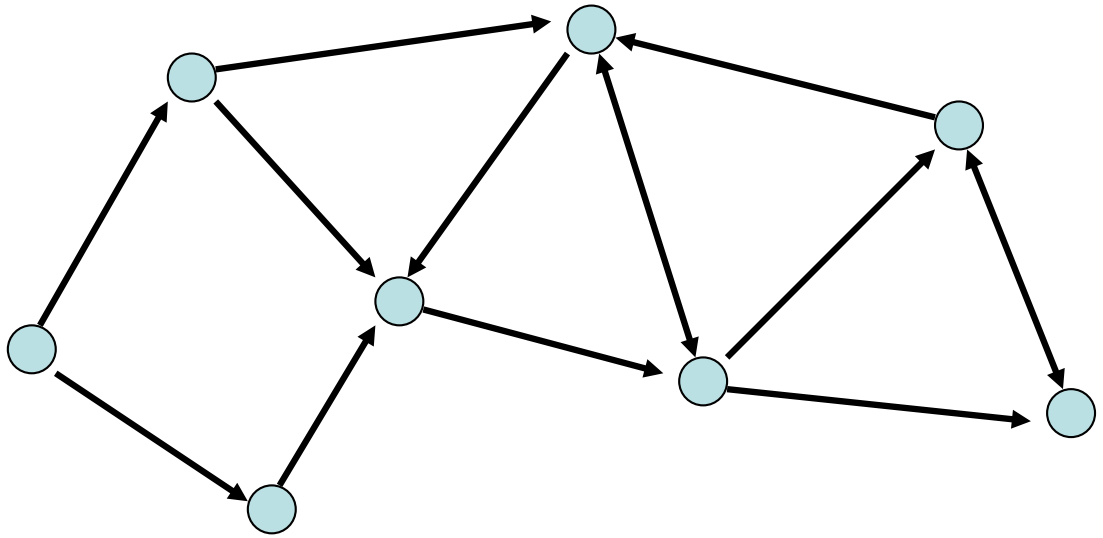
(k,l)-Gitter G=(V,E):

- $V=[k] \times [l]$ ($[a]=\{0,\ldots,a-1\}$ 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)

Graphdurchlauf

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



Graphdurchlauf

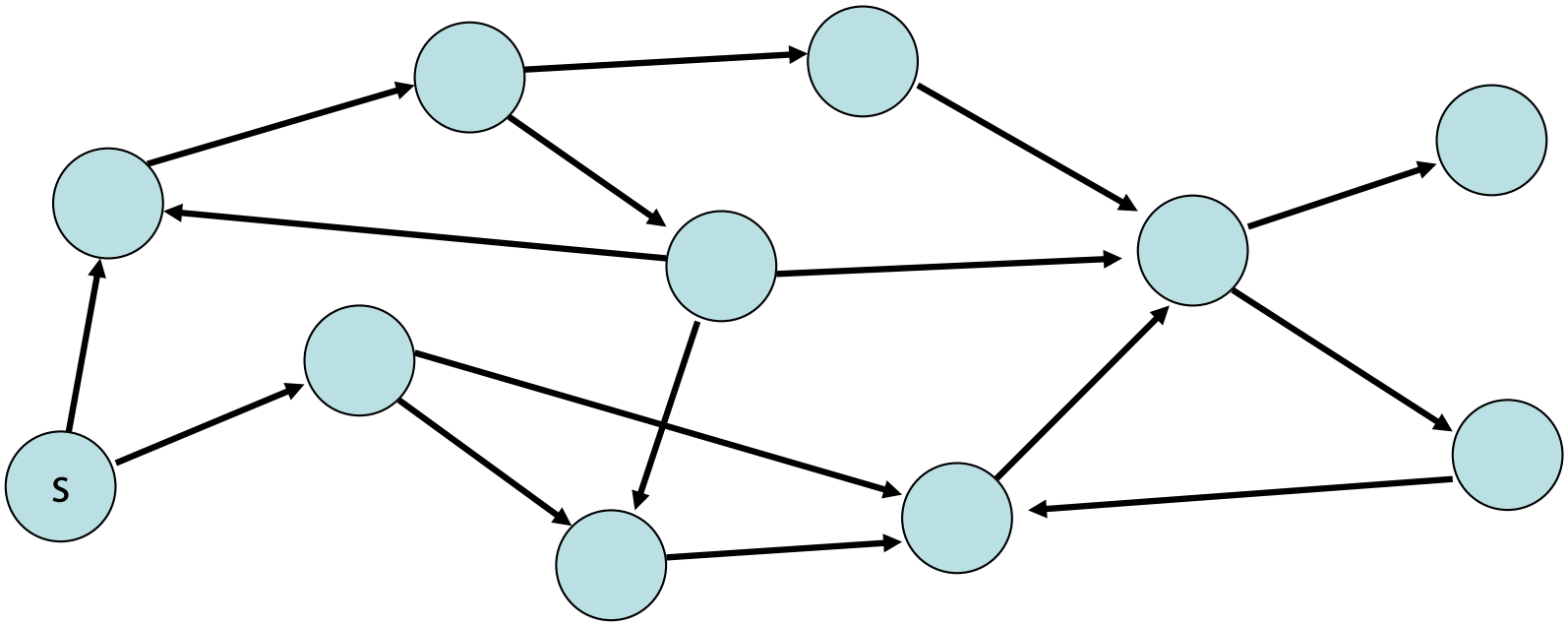
Zentrale Frage: Wie können wir die Knoten eines Graphen durchlaufen, so dass jeder Knoten mindestens einmal besucht wird?

Grundlegende Strategien:

- Breitensuche
- Tiefensuche

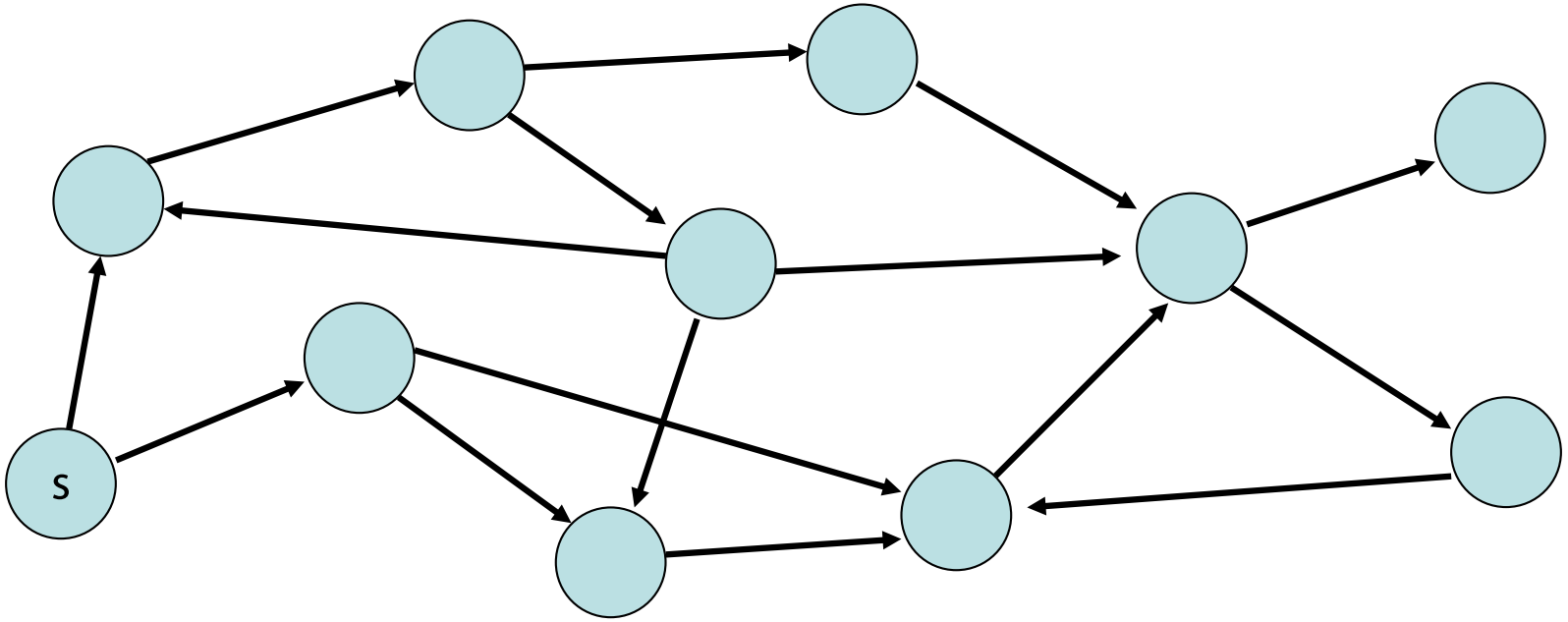
Breitensuche

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



Tiefensuche

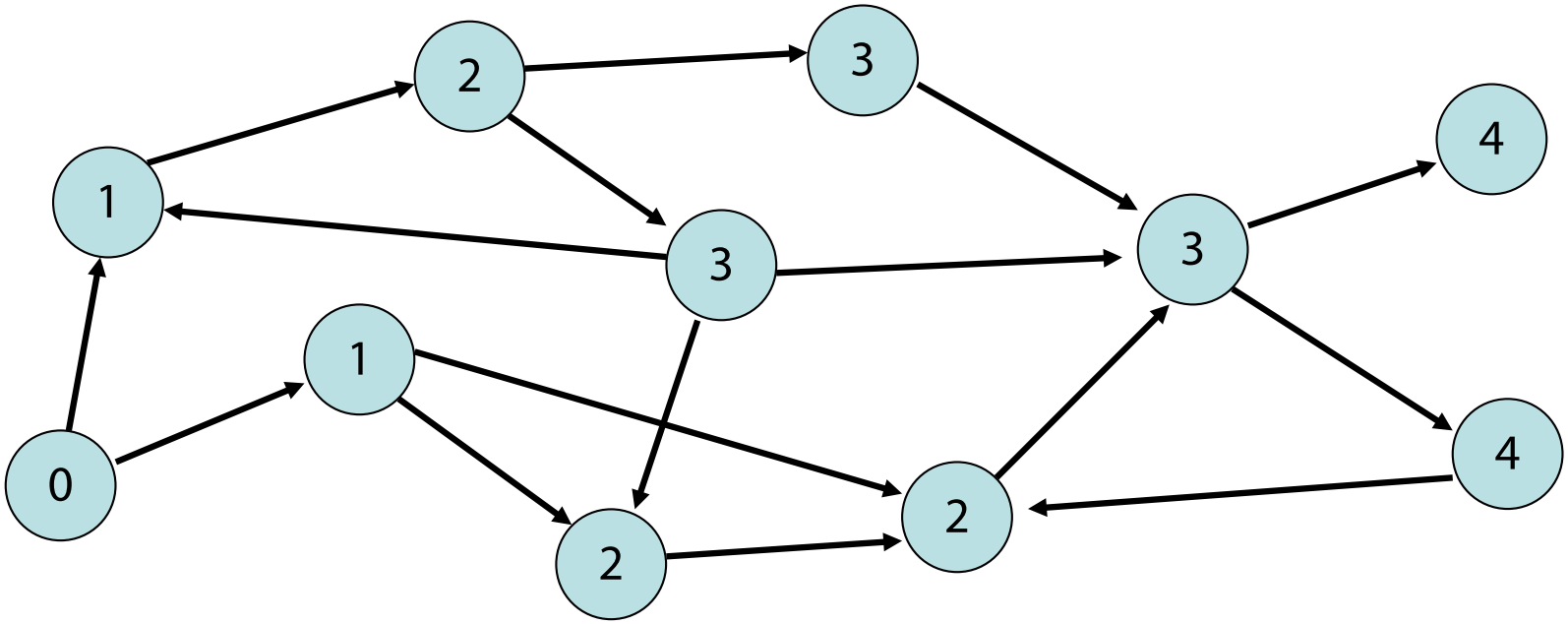
- Starte von einem Knoten s
- Exploriere Graph in die Tiefe
  (●: aktuell, ●: noch aktiv, ●: fertig)



Breitensuche

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

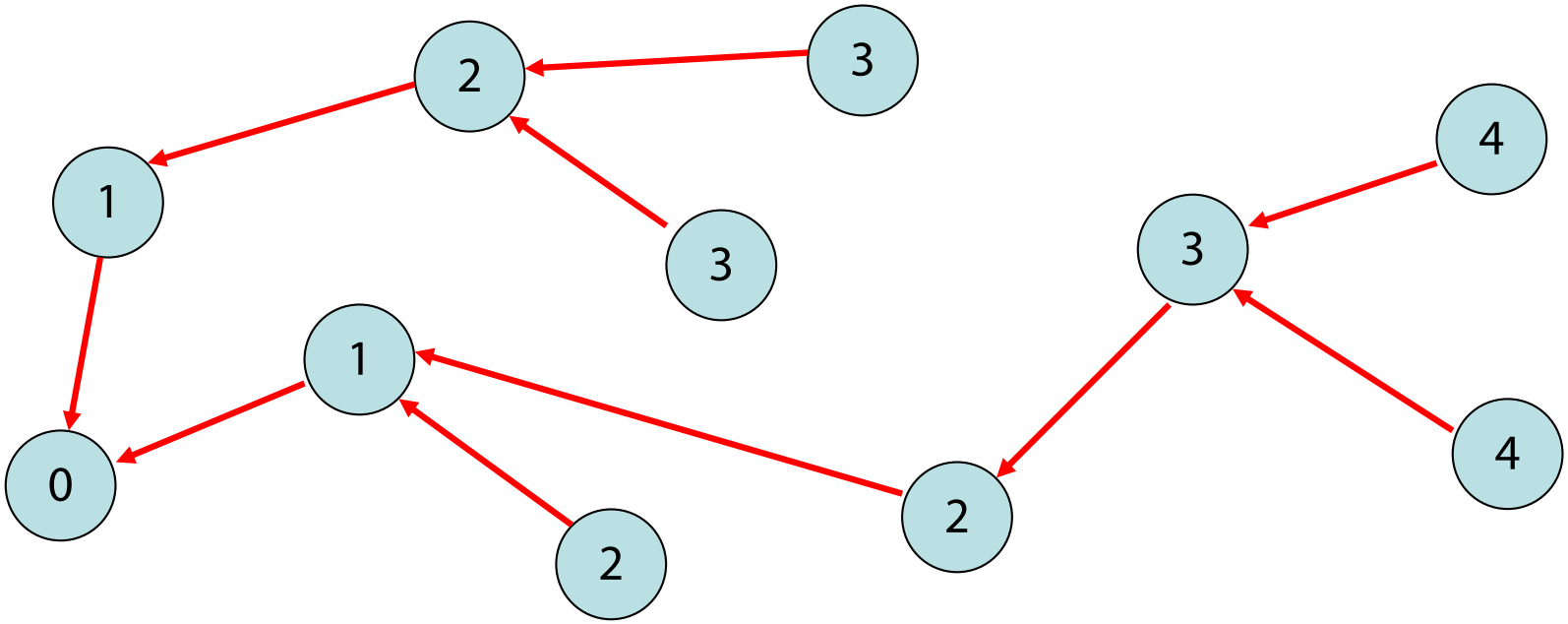
Distanzen:



Breitensuche

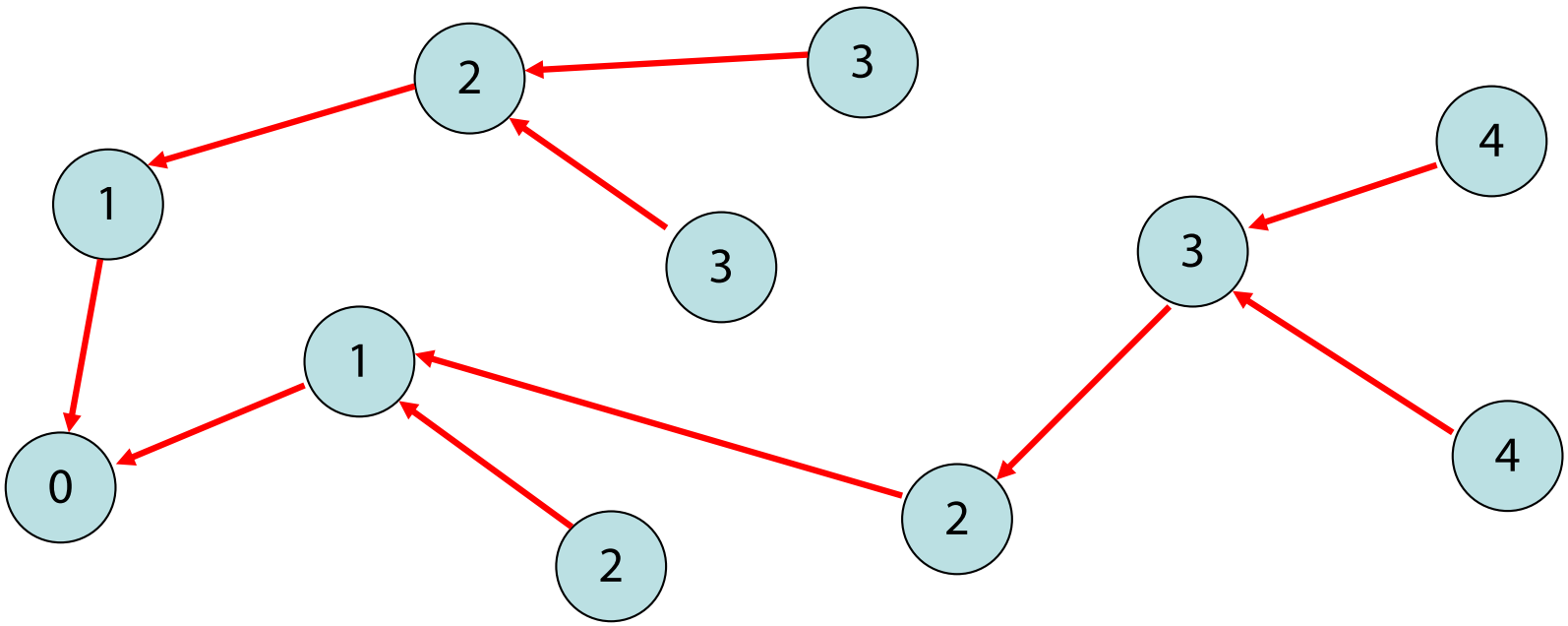
- d(v): Distanz von Knoten v zu s (d(s)=0)
- parent(v): Knoten, von dem v besucht

Mögliche Parent-Beziehungen in rot:



Breitensuche

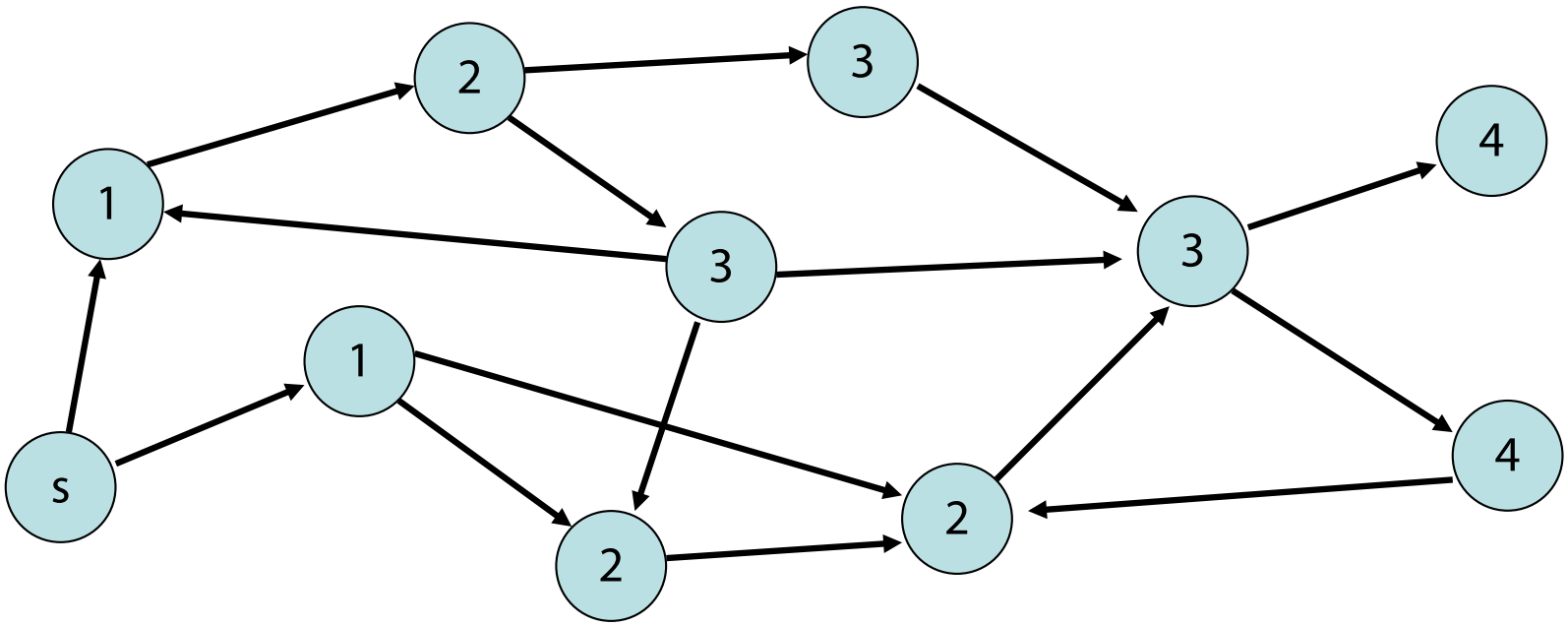
Parent-Beziehung eindeutig: wenn Knoten v zum erstenmal besucht wird, wird parent(v) gesetzt und dadurch v markiert, so dass v nicht nochmal besucht wird



Breitensuche

Kantentypen:

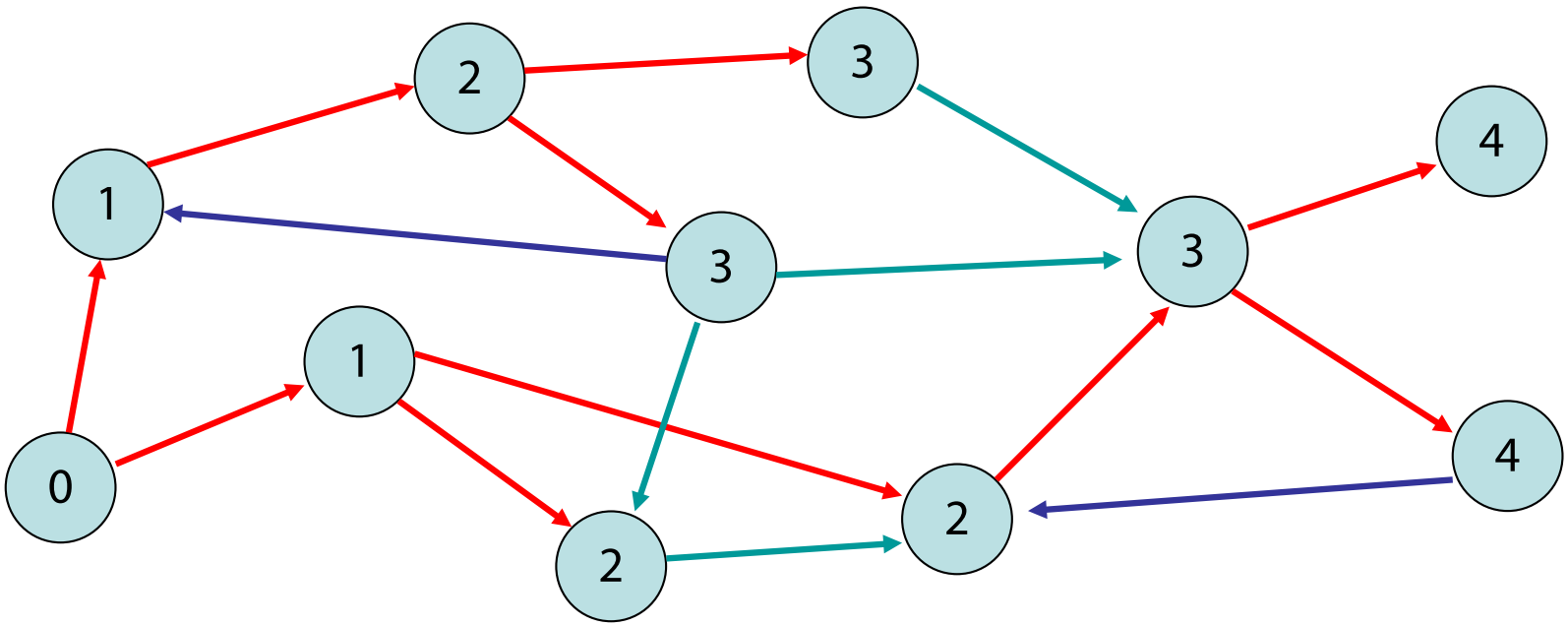
- Baumkante: zum Kind
- Rückwärtskante: zu einem Vorfahr
- Kreuzkante: alle sonstige Kanten



Breitensuche

Kantentypen:

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



Iteratoren

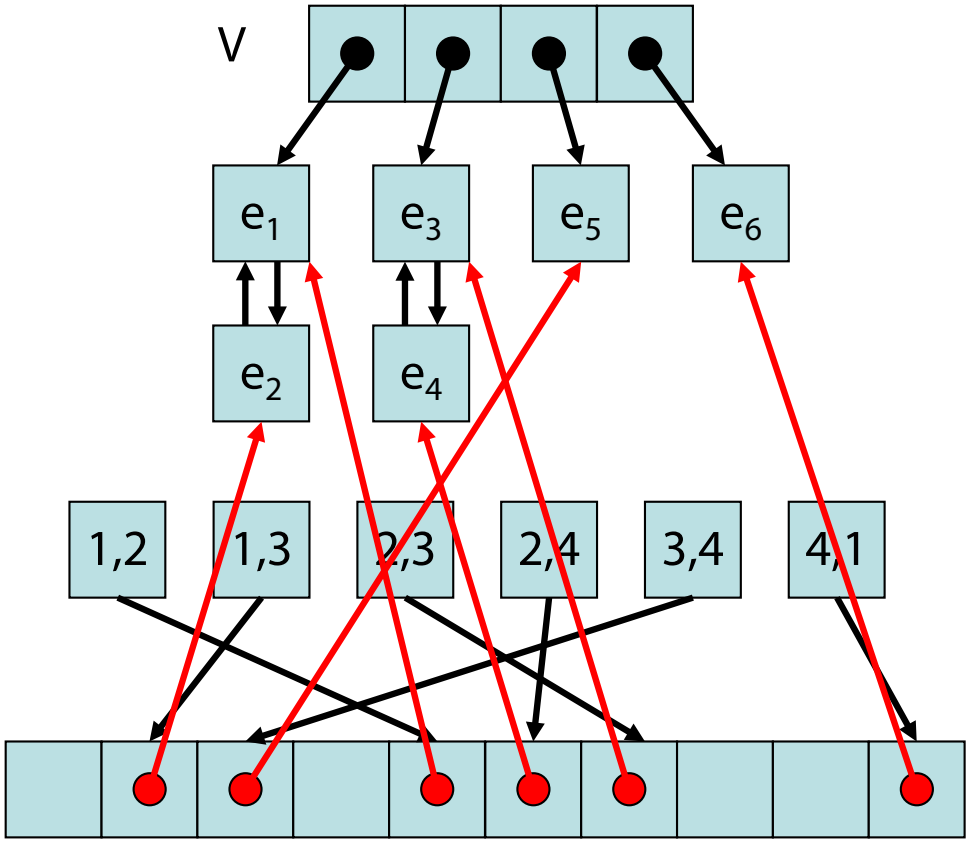
Sei $G=(V, E)$ ein Graph

- Iteration über Kanten (u und v nicht instantiiert):
  - **foreach** $(u,v) \in E$ **do**
- Iteration über Knoten:
  - **foreach** $v \in V$ **do**
  - **foreach** $(u,v) \in E$ **do**, wobei
    - u instantiiert
      - Finde alle von u ausgehenden Kanten
    - oder v instantiiert
      - Finde alle bei v eintreffenden Kanten

Iteratoren

Zeitaufwand (grob):

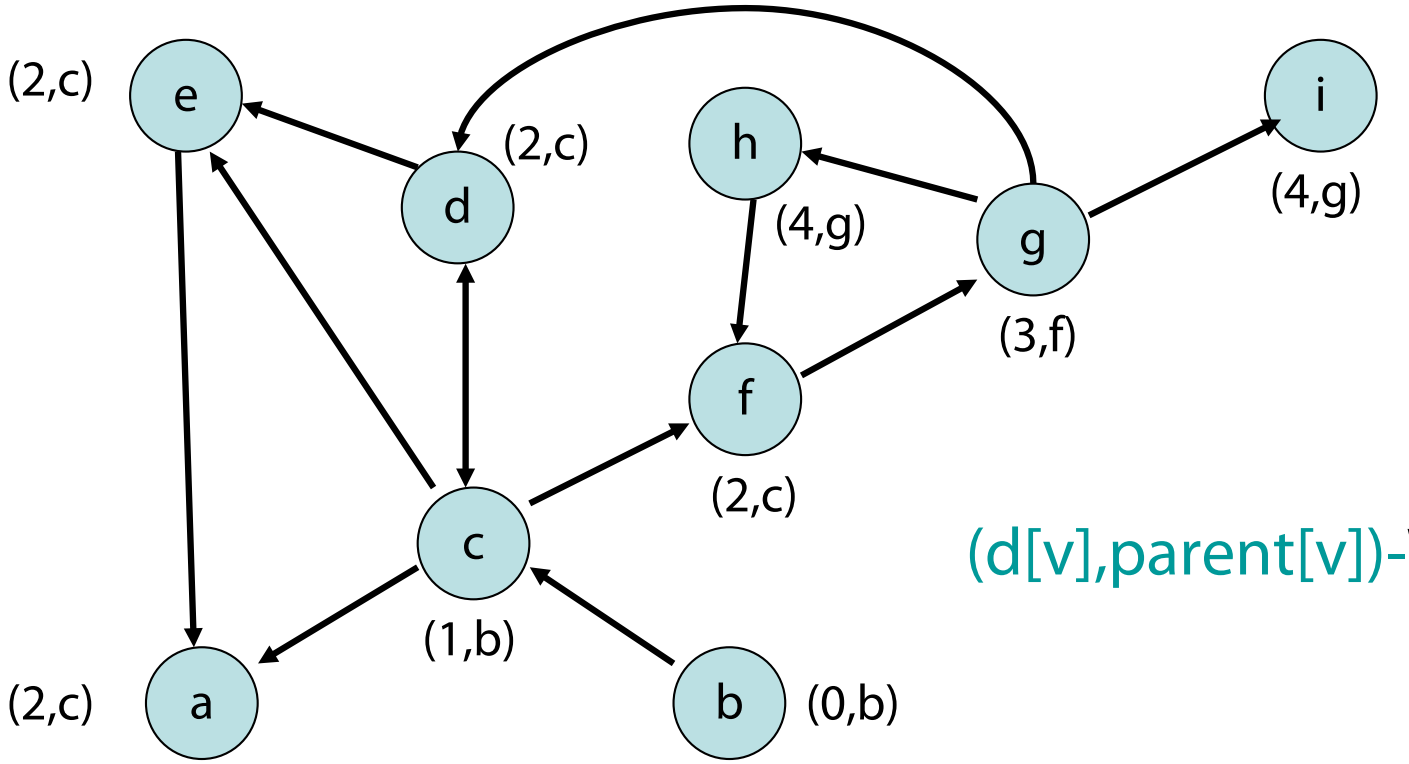
- getIteratorOut(v, G): O(1) (worst case)
- getIteratorIn(v, G): O(?) (worst case)



Breitensuche

```
Procedure BFS(s: Node)
    d = <∞,…,∞>: Array [1..n] of IN
    parent = <⊥,…, ⊥>: Array [1..n] of Node
    d[key(s)]:=0              // s hat Distanz 0 zu sich
    parent[Key(s)]:=s         // s ist sein eigener Vater
    q:=<s>: Queue             // q:Queue zu besuchender Knoten
    while q ≠ <> do           // solange q nicht leer
       u:= dequeue(q)         // nimm Knoten nach FIFO-Regel
       foreach (u,v) ∈ E do
          if parent(key(v))=⊥ then  // v schon besucht?
             enqueue(v, q)  // nein, dann in q hinten einfügen
             d[Key(v)]:=d[key(u)]+1
             parent[key(v)]:=u
```

BFS(b)

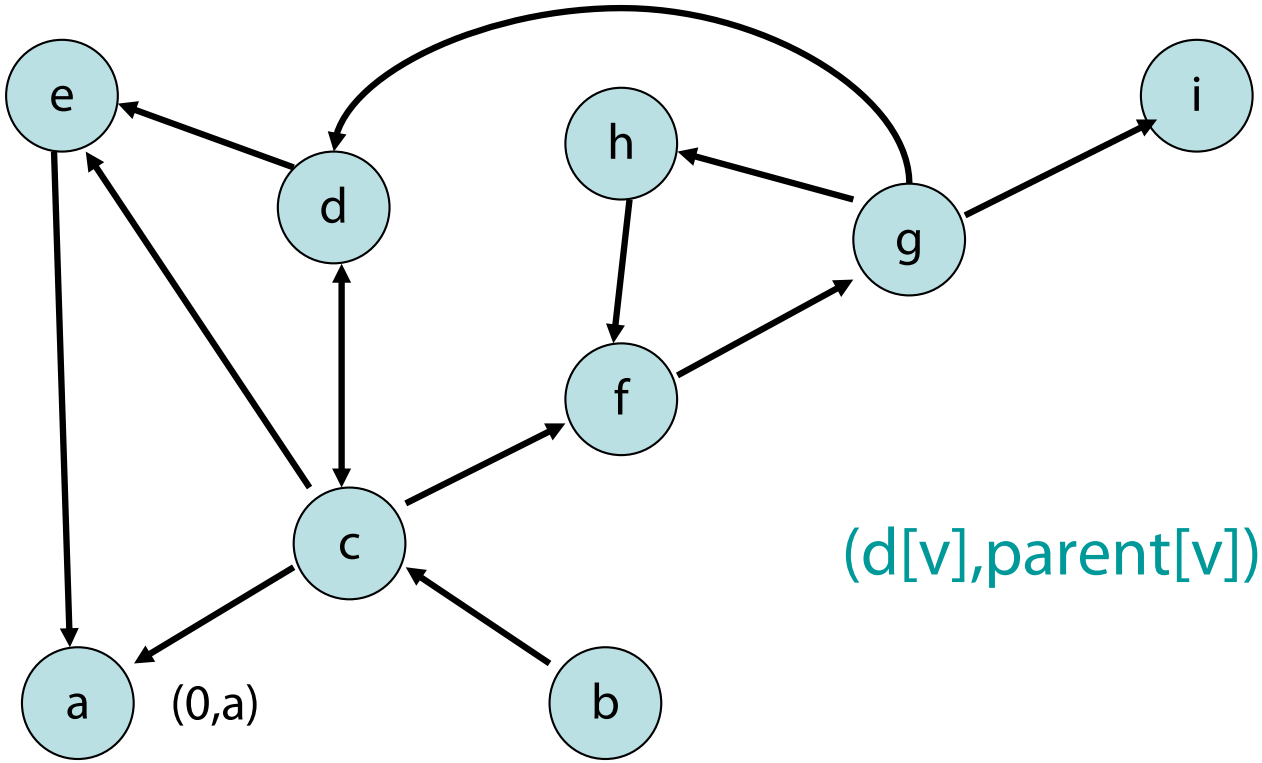


(d[v],parent[v])-Werte

: besucht, noch in q

: besucht, nicht mehr in q

BFS(a)

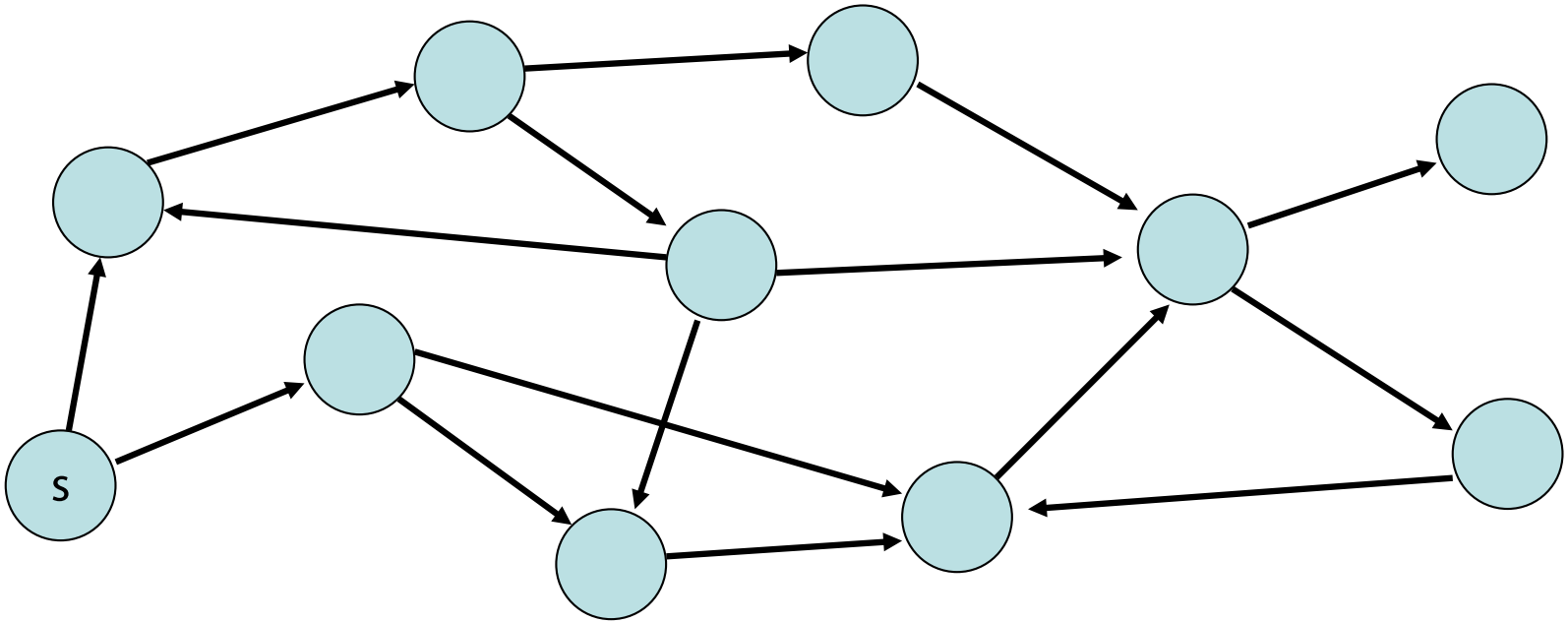


(d[v],parent[v])-Werte

Von a kein anderer Knoten erreichbar.

Tiefensuche

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



Tiefensuche - Schema

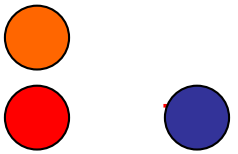
```
Übergeordnete Prozedur:
    unmark all nodes
    init()
    foreach s∈V do  // stelle sicher, dass alle Knoten besucht werden
      if s is not marked then
        mark s
        root(s)
        DFS(s,s)    // s: Startknoten

Procedure DFS(u,v: Node)  // u: Vater von v
    foreach (v,w)∈E do
      if w is marked then traverseNonTreeEdge(v,w)
      else traverseTreeEdge(v,w)
        mark w
        DFS(v,w)
    backtrack(u,v)
```

Prozeduren in rot: noch zu spezifizieren

DFS-Nummerierung

Variablen:

- dfsNum: Array [1..n] of IN // Zeitpunkt wenn Knoten
- finishTime: Array [1..n] of IN // Zeitpunkt wenn Knoten
- dfsPos, finishingTime: IN // Zähler

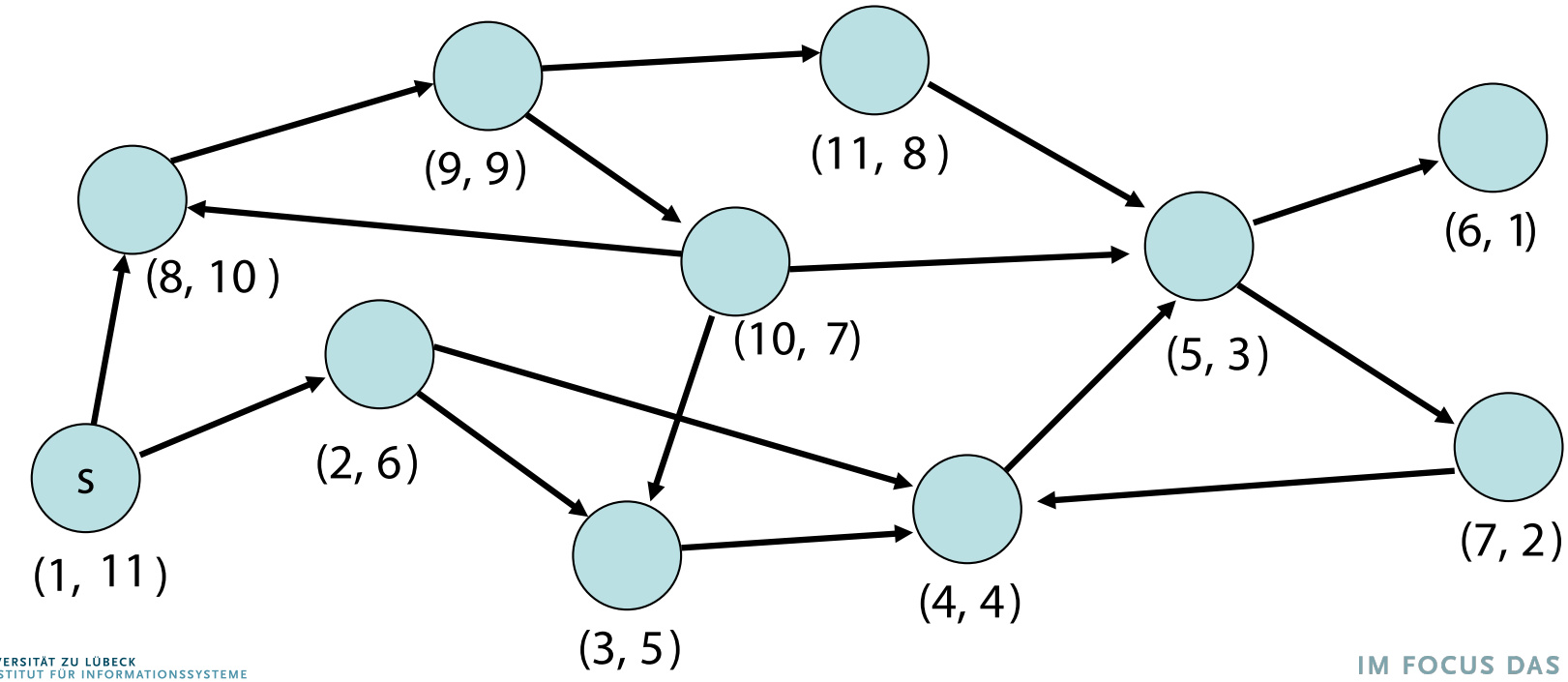


Prozeduren:

- init():
  dfsPos:=1; finishingTime:=1
- root(s):
  dfsNum[s]:=dfsPos; dfsPos:=dfsPos+1
- traverseTreeEdge(v,w):
  dfsNum[w]:=dfsPos; dfsPos:=dfsPos+1
- traverseNonTreeEdge(v,w):
  -
- backtrack(u,v):
  finishTime[v]:=finishingTime; finishingTime := finishingTime+1

DFS-Nummerierung

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



DFS-Nummerierung

Ordnung $<$ auf den Knoten:

$u<v$ gdw. $dfsNum[u]<dfsNum[v]$

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

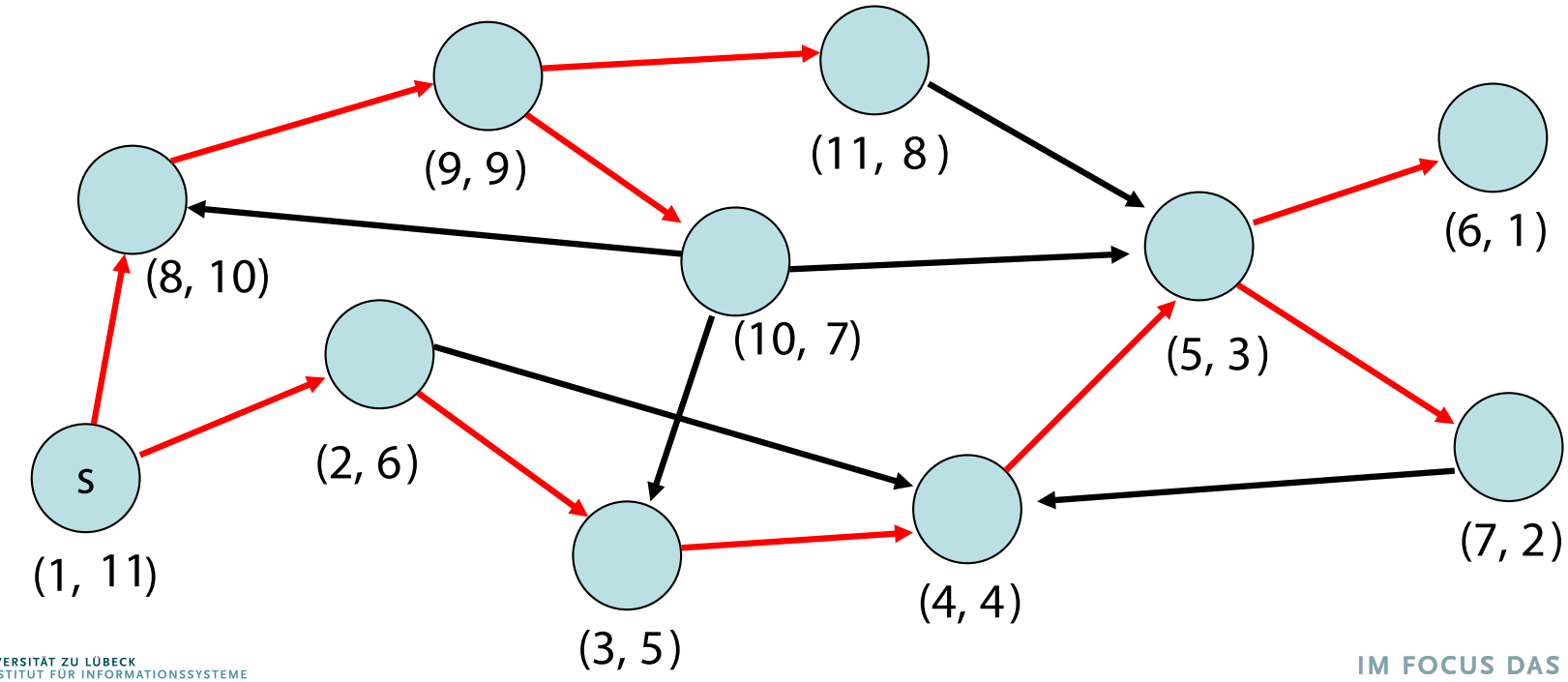
Beweis:

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

DFS-Nummerierung

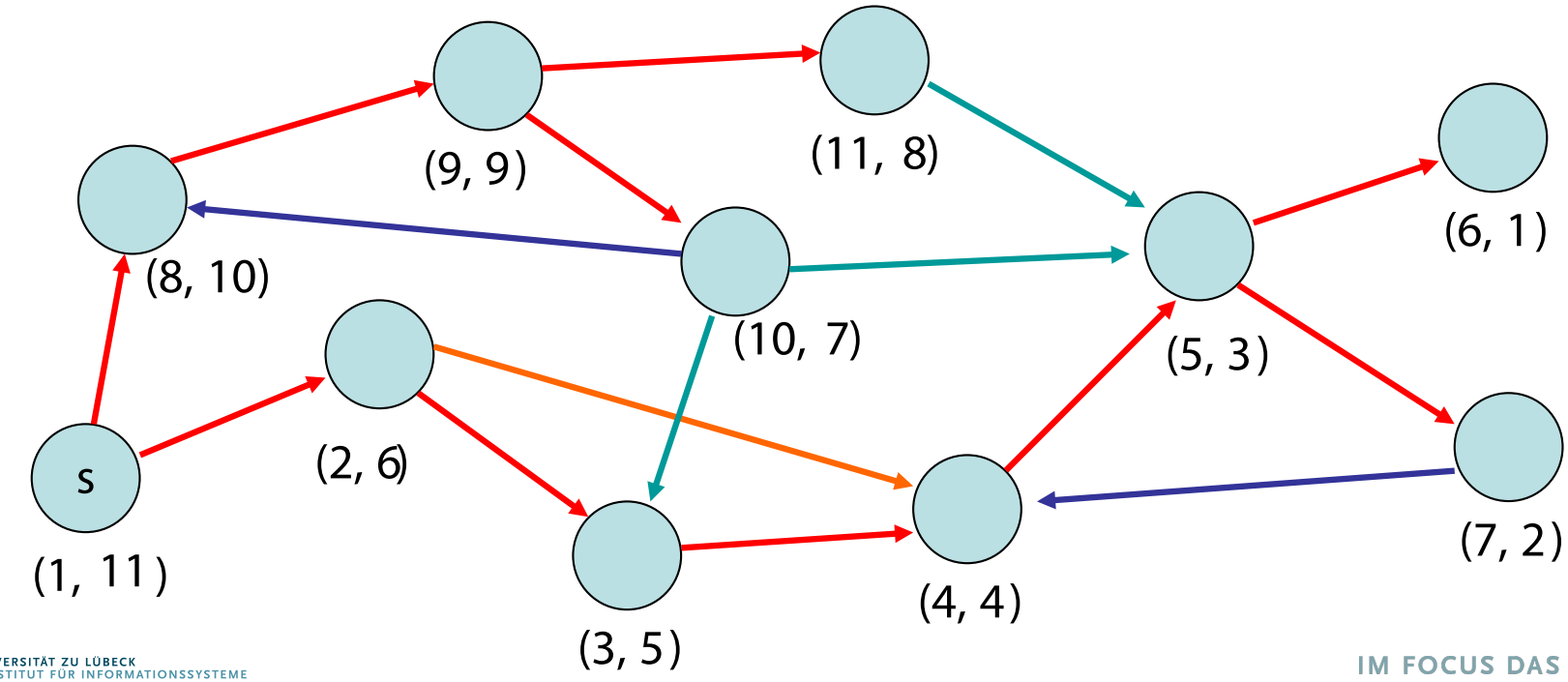
Überprüfung von Lemma 1:

- 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
- Kreuzkante: alle sonstige Kanten



DFS-Nummerierung

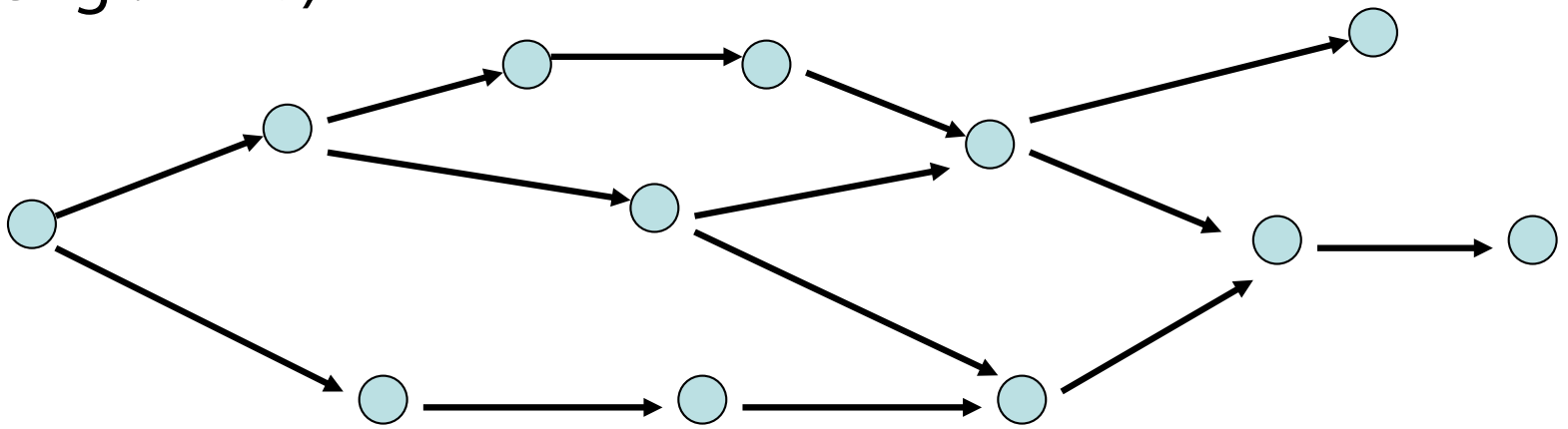
Beobachtung für Kante (v,w):

Kantentyp	dfsNum[v]< dfsNum[w]	finishTime[v]> finishTime[w]
Baum & Vorwärts	Ja	Ja
Rückwärts	Nein	Nein
Kreuz	Nein	Ja

DFS-Nummerierung

Anwendung:

- Erkennung eines azyklischen gerichteten Graphen (engl. DAG)



Merkmal: keine gerichteten Kreise

DFS-Nummerierung in O(n+m)

DFS-Nummerierung

Behauptung: Das Folgende ist äquivalent:

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

Beweis (2. $\Leftrightarrow$ 3.):

2. $\Leftrightarrow$ 3.: folgt aus Tabelle

Kantentyp	dfsNum[v]< dfsNum[w]	finishTime[v]> finishTime[w]
Baum & Vorwärts	Ja	Ja
Rückwärts	Nein	Nein
Kreuz	Nein	Ja

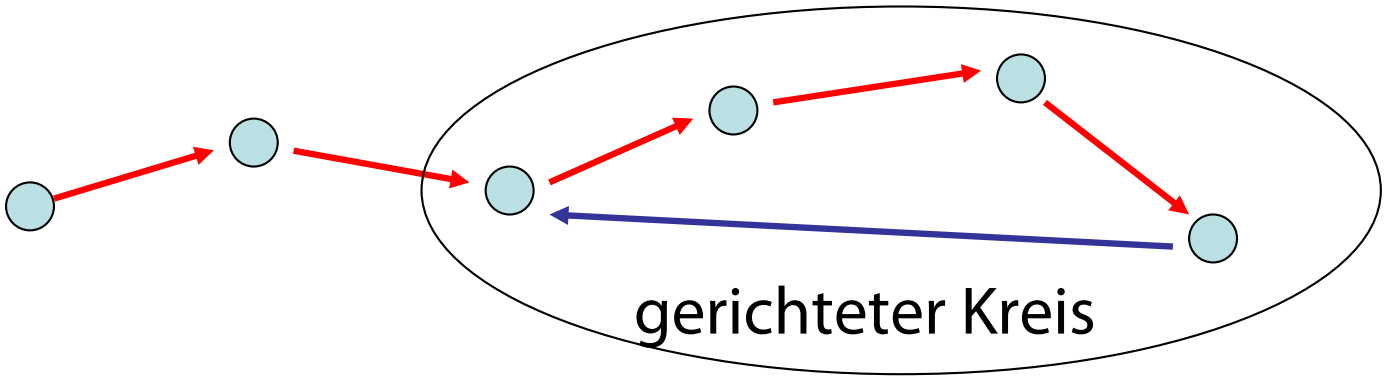
DFS-Nummerierung

Behauptung: Das Folgende ist äquivalent:

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

Beweis (1. $\Rightarrow$ 2.):

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



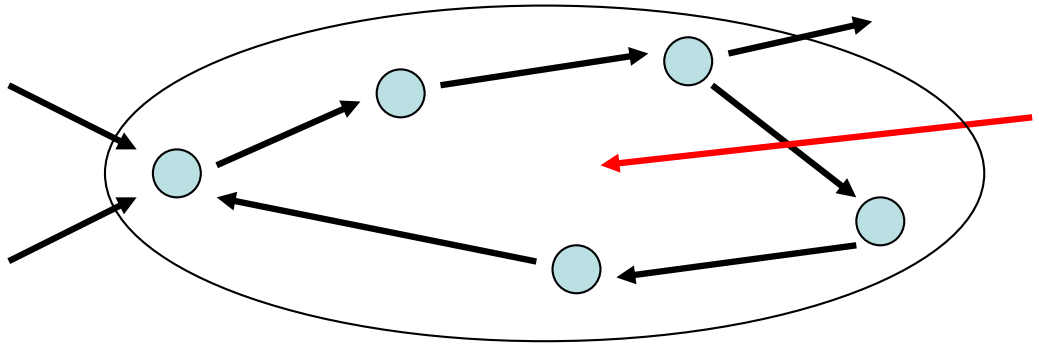
DFS-Nummerierung

Behauptung: Das Folgende ist äquivalent:

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

Beweis (2. $\Rightarrow$ 1.):

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



DFS-Nummerierung

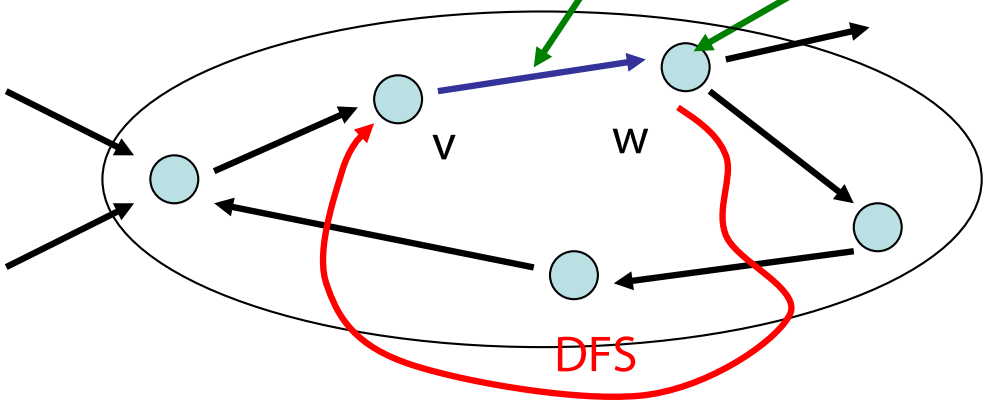
Behauptung: Das Folgende ist äquivalent:

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

Beweis (2. $\Rightarrow$ 1.):

Kontrapositiv

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



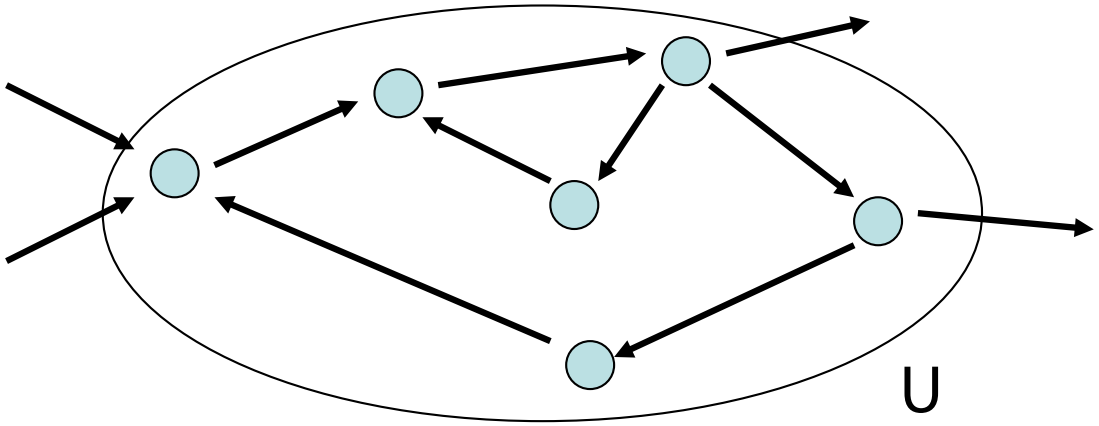
Annahme: Erster von DFS besuchter Knoten im Kreis

Kantentyp	dfsNum[v]< dfsNum[w]	finishTime[v]> finishTime[w]
Baum & Vorwärts	Ja	Ja
Rückwärts	Nein	Nein
Kreuz	Nein	Ja

Starke ZHKs

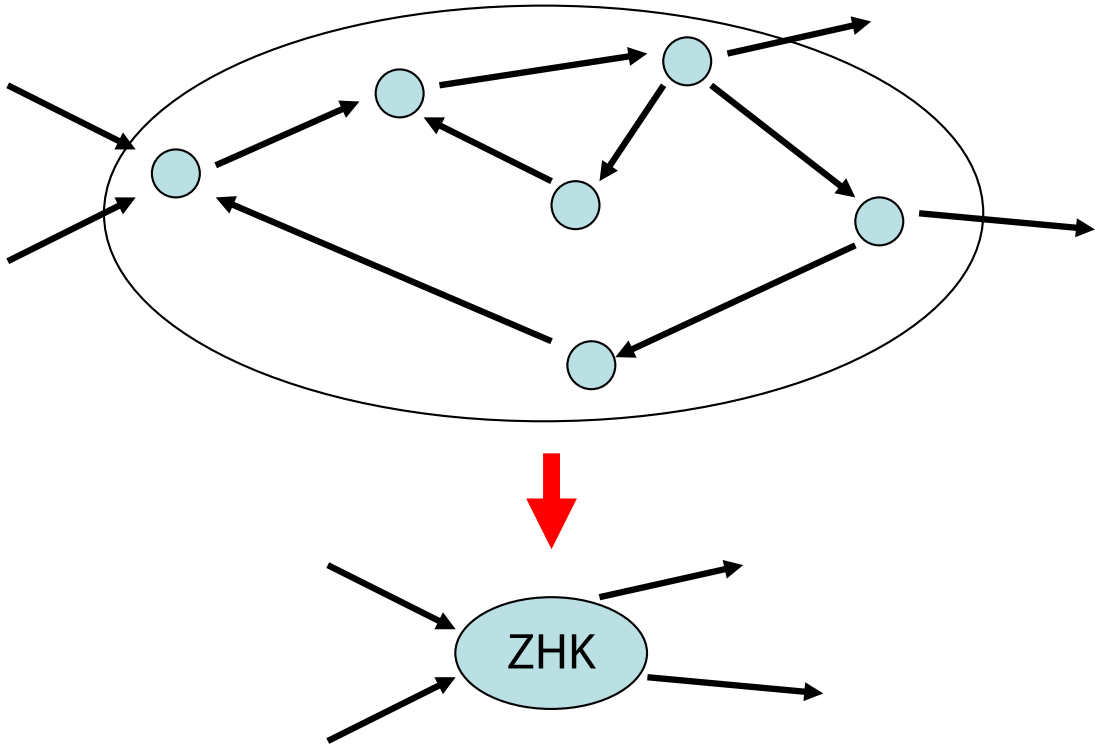
Definition: Sei G=(V,E) ein gerichteter Graph.

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

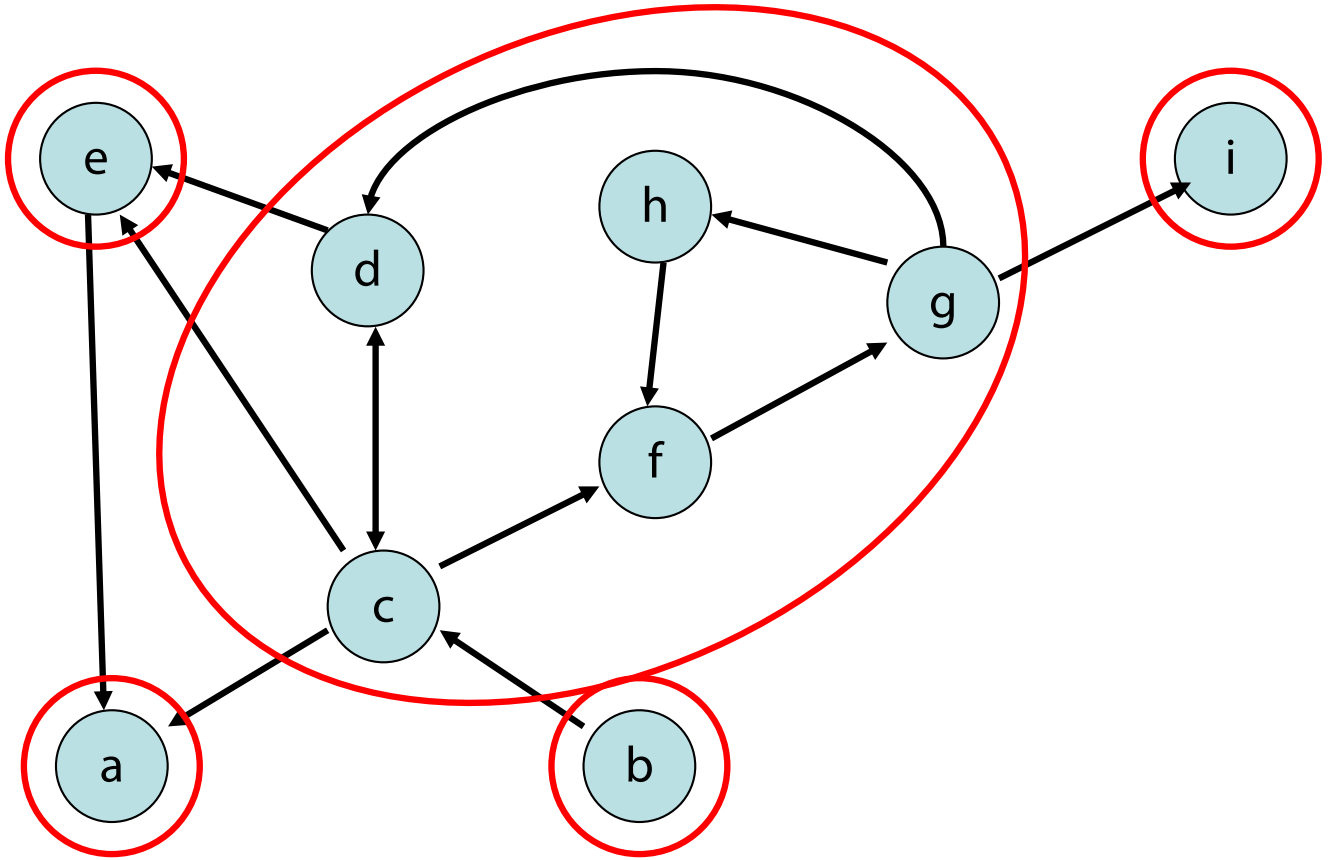


Starke ZHKs

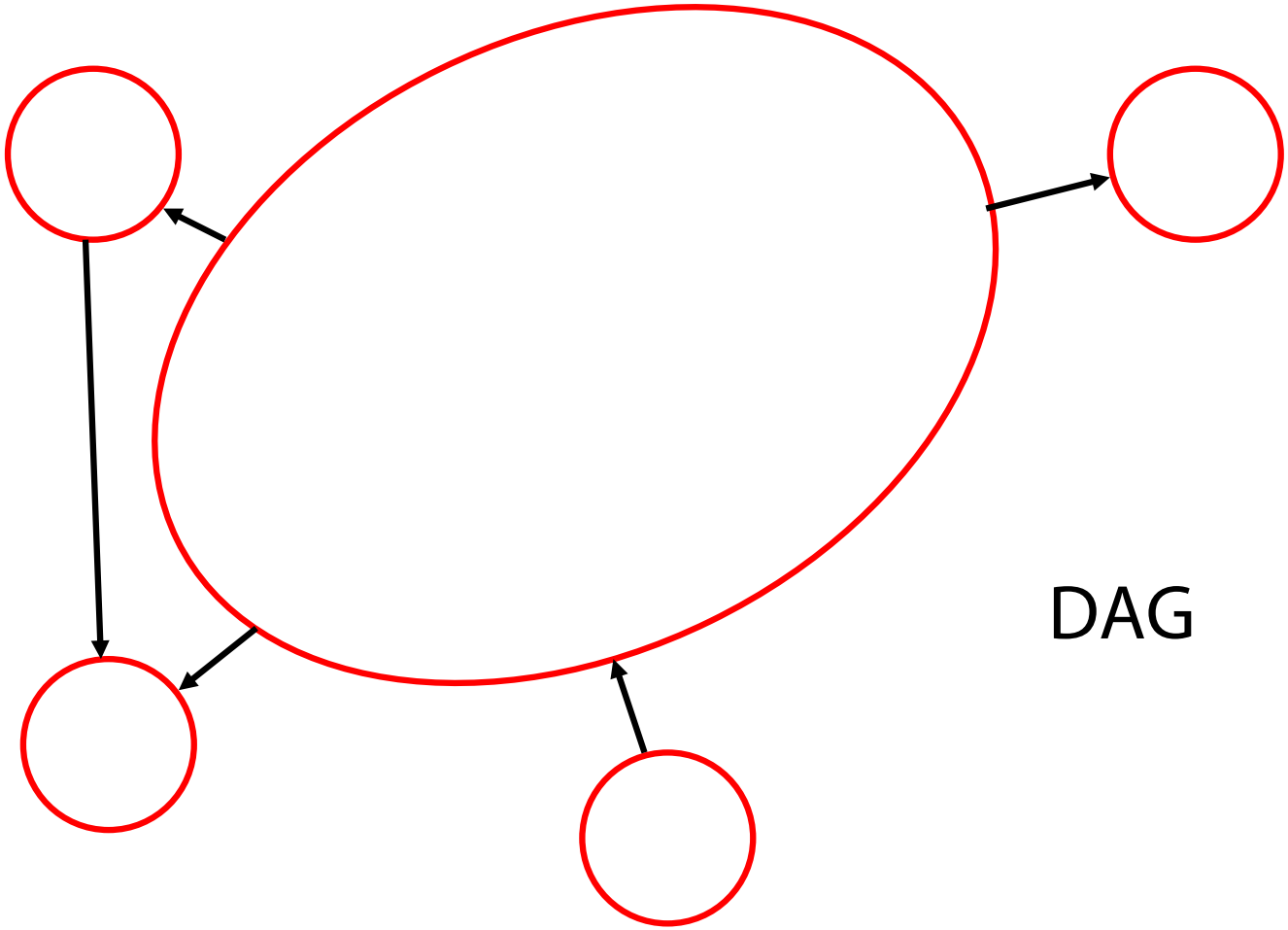
Beobachtung: Schrumpft man starke ZHKs zu einzelnen Knoten, dann ergibt sich DAG.



Starke ZHKs - Beispiel



Starke ZHKs - Beispiel



Starke ZHKs

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

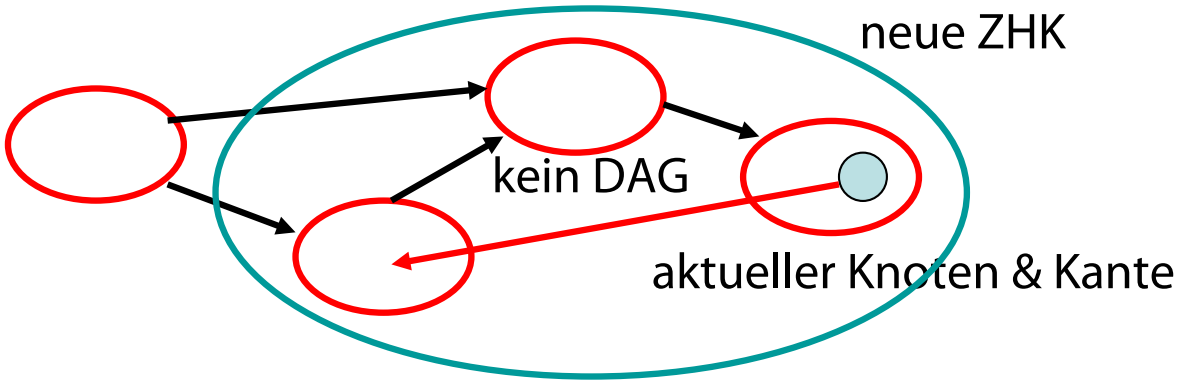
Strategie: Verwende DFS-Verfahren mit
component: Array [1..n] of 1..n

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

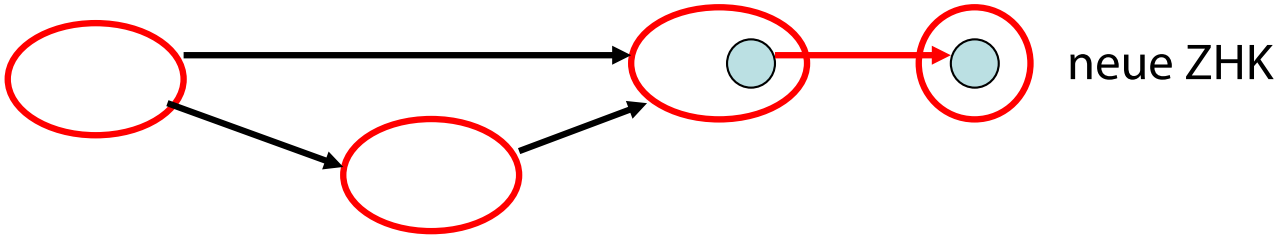
Robert Tarjan: *Depth-first search and linear graph algorithms*.
In: *SIAM Journal on Computing*. Bd. 1, Nr. 2, S. 146-160, **1972**

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)

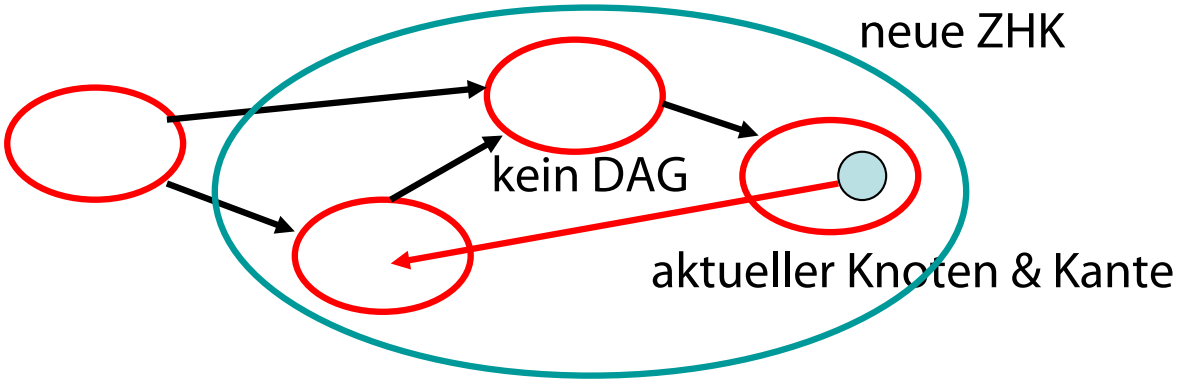


  - b)



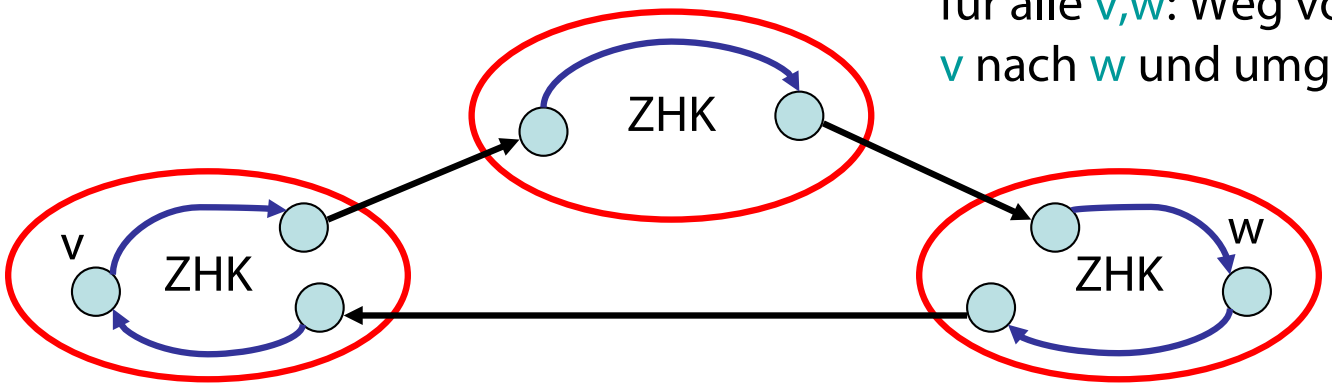
Starke ZHKs

Warum ZHKs zu einer zusammenfassbar?

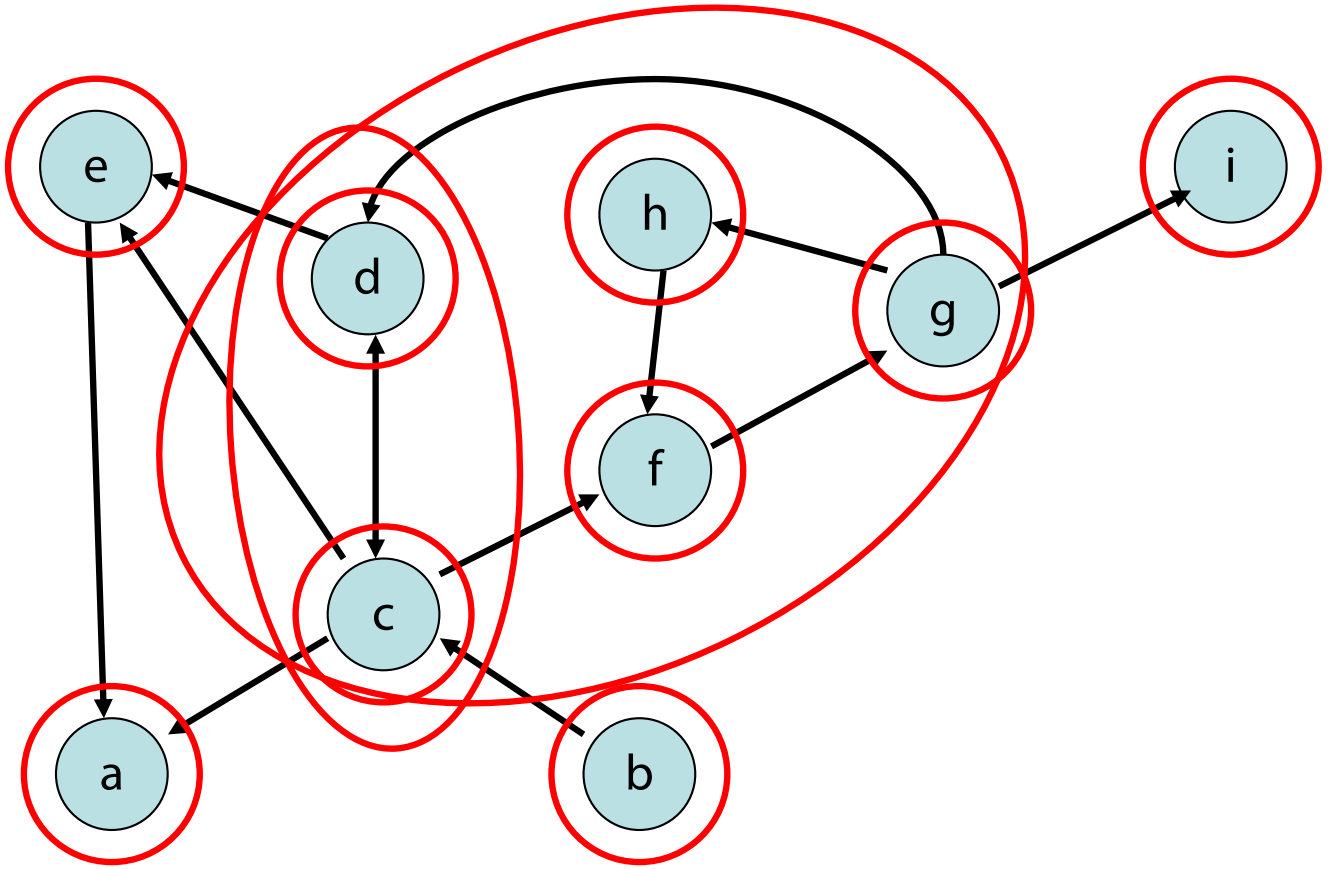


Grund:

für alle v,w: Weg von
v nach w und umgekehrt



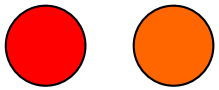
Starke ZHKs - Beispiel



Problem: wie fasst man ZHKs effizient zusammen?

Starke ZHKs

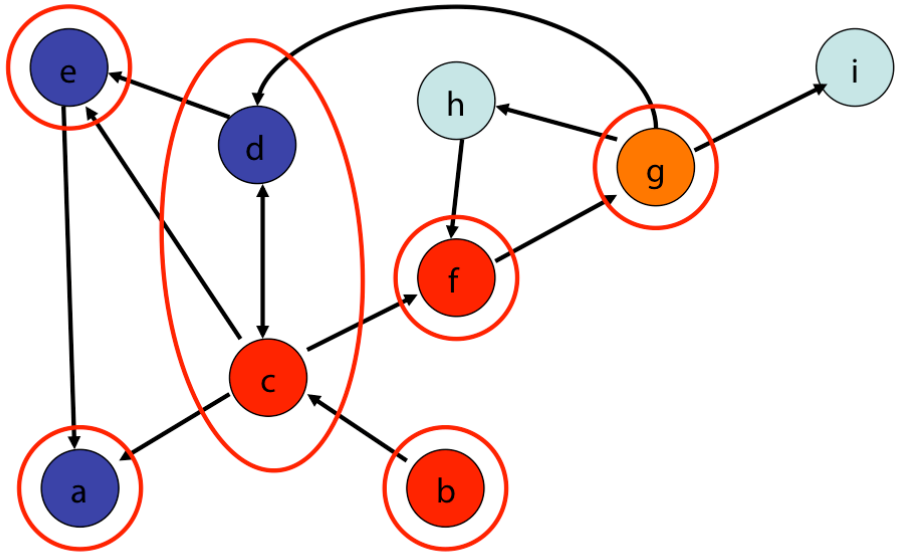
Definition:

-  : 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:

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

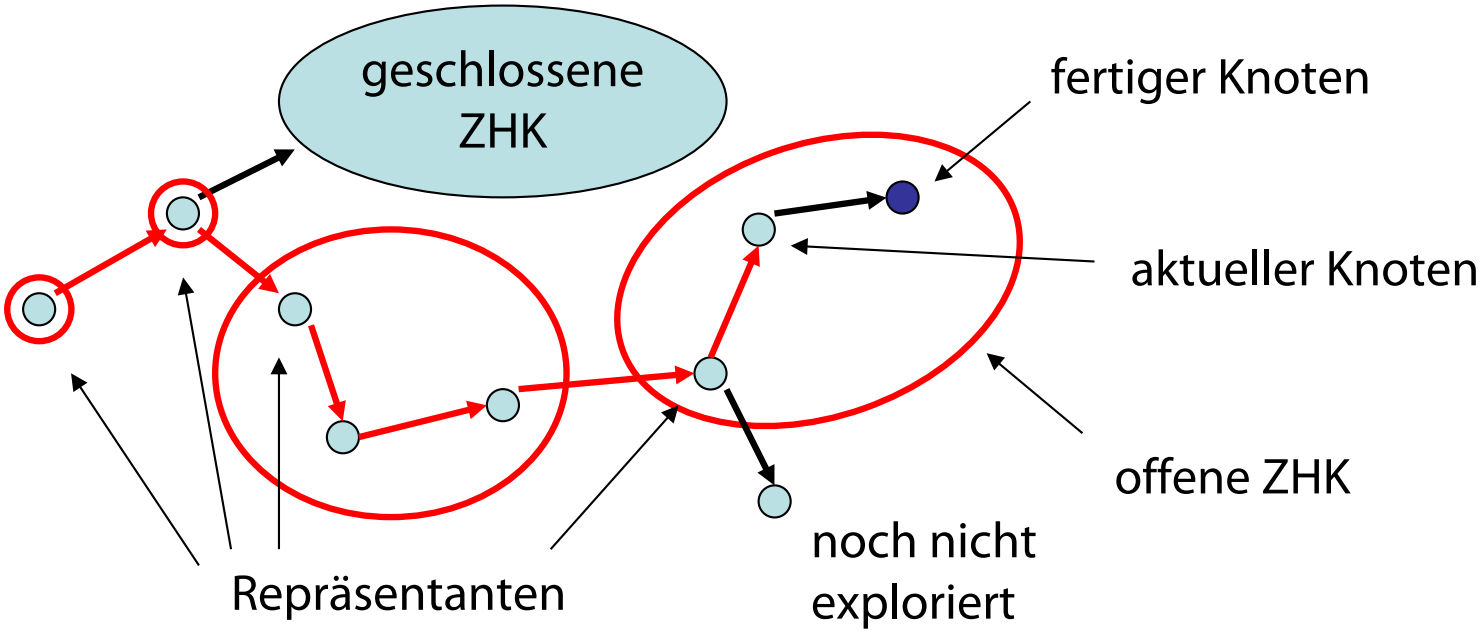
Beobachtungen sind Invarianten:

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.

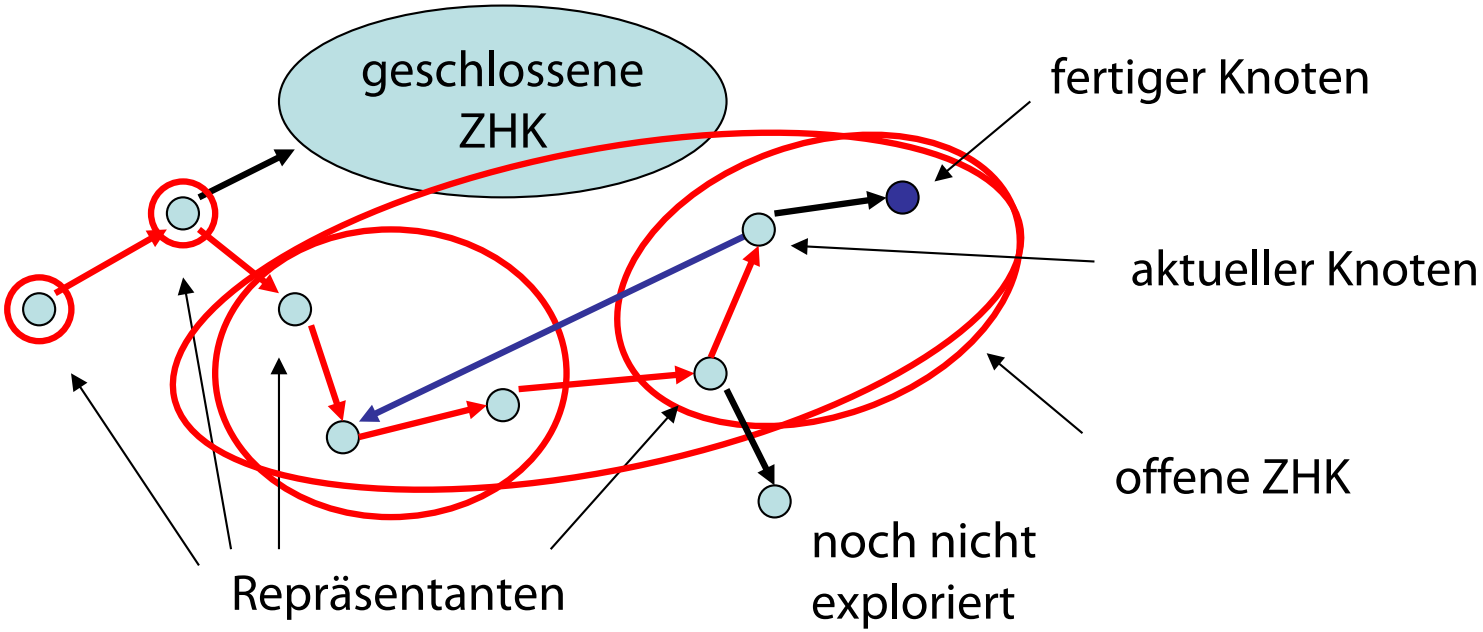
- Anfangs gelten alle Invarianten
- Wir betrachten verschiedene Fälle



Starke ZHKs

Beweis über vollständige Induktion.

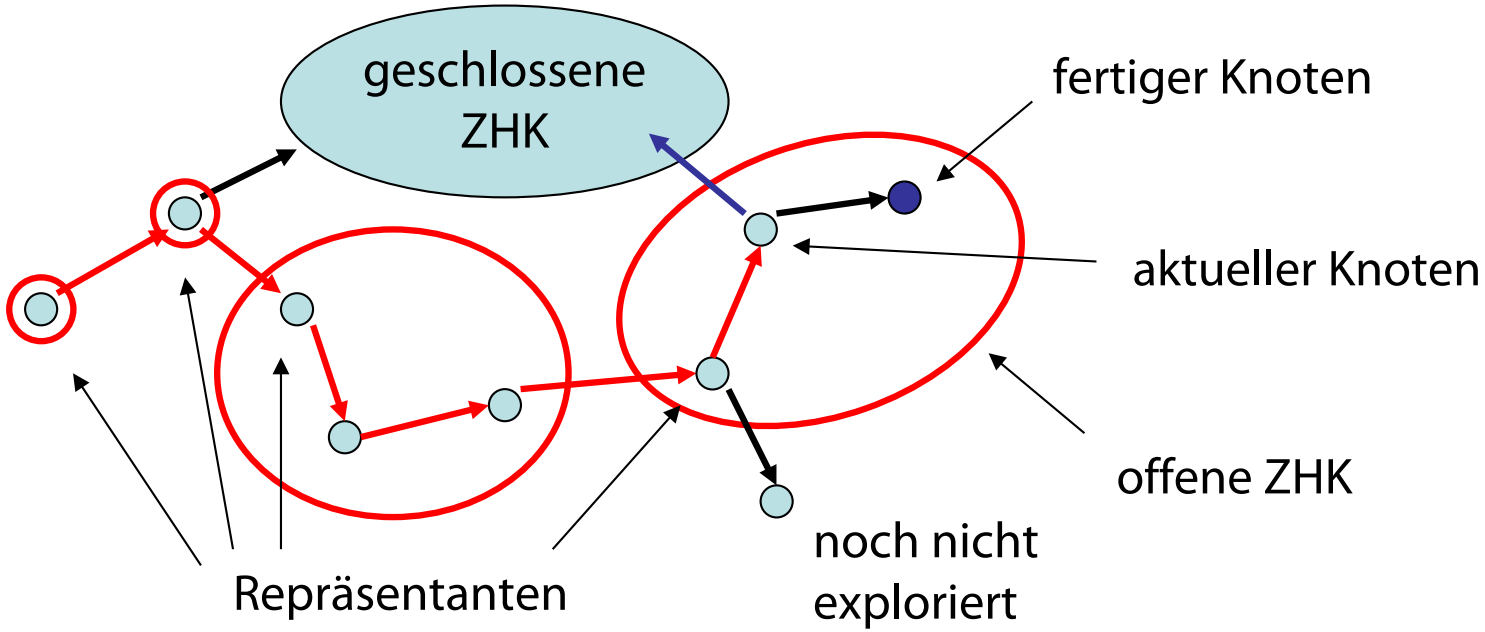
- Anfangs gelten alle Invarianten
- Fall 1: Kante zu unfertigem Knoten



Starke ZHKs

Beweis über vollständige Induktion.

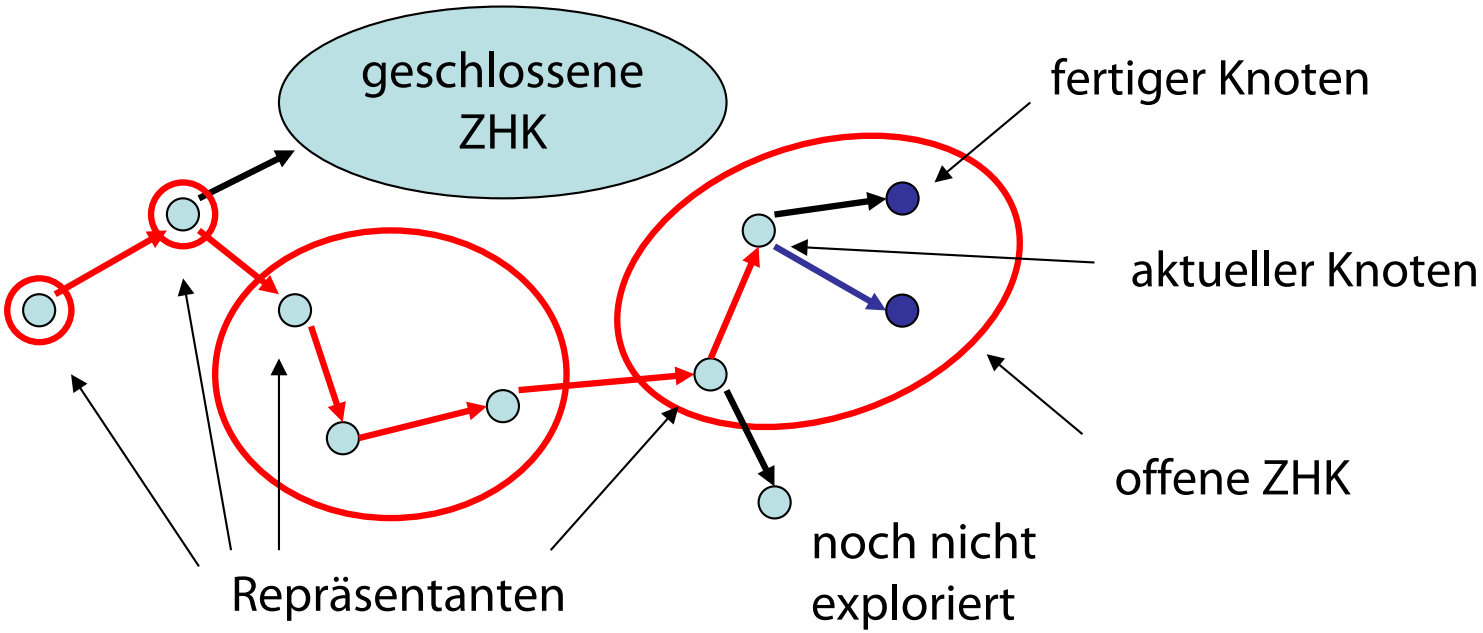
- Anfangs gelten alle Invarianten
- Fall 2: Kante zu geschlossenem Knoten



Starke ZHKs

Beweis über vollständige Induktion.

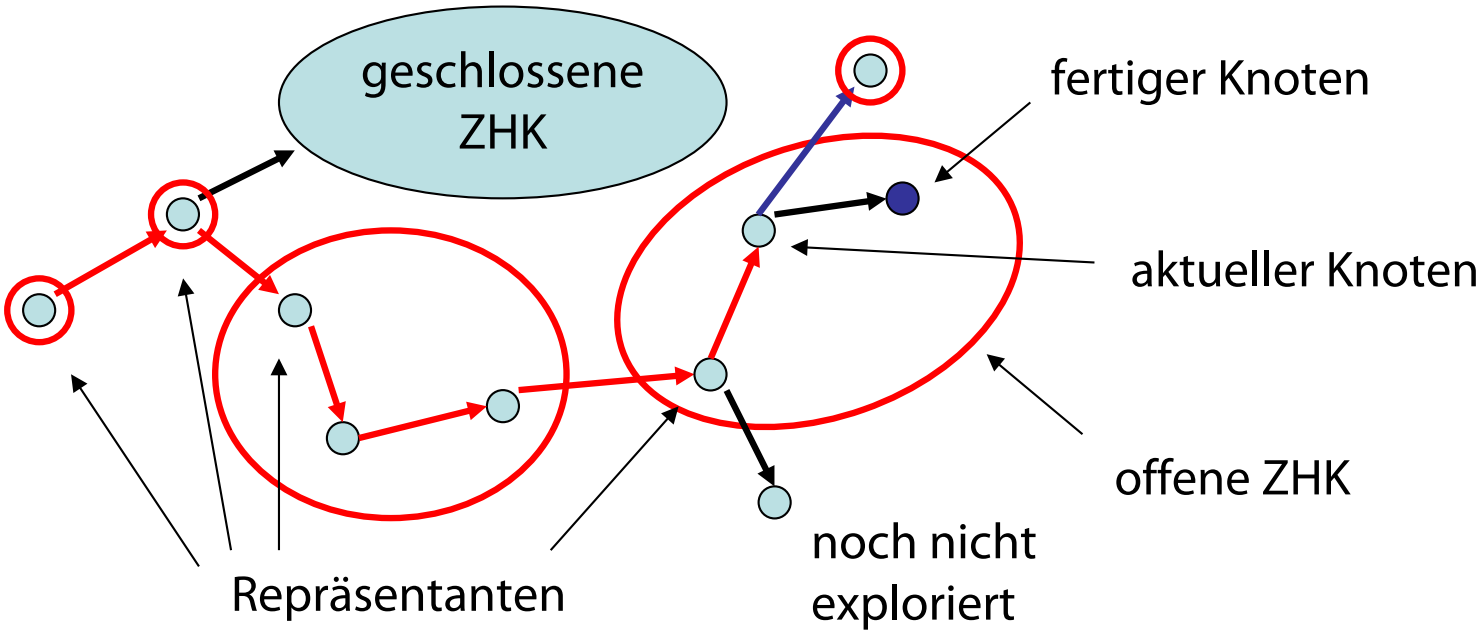
- Anfangs gelten alle Invarianten
- Fall 3: Kante zu fertigem Knoten



Starke ZHKs

Beweis über vollständige Induktion.

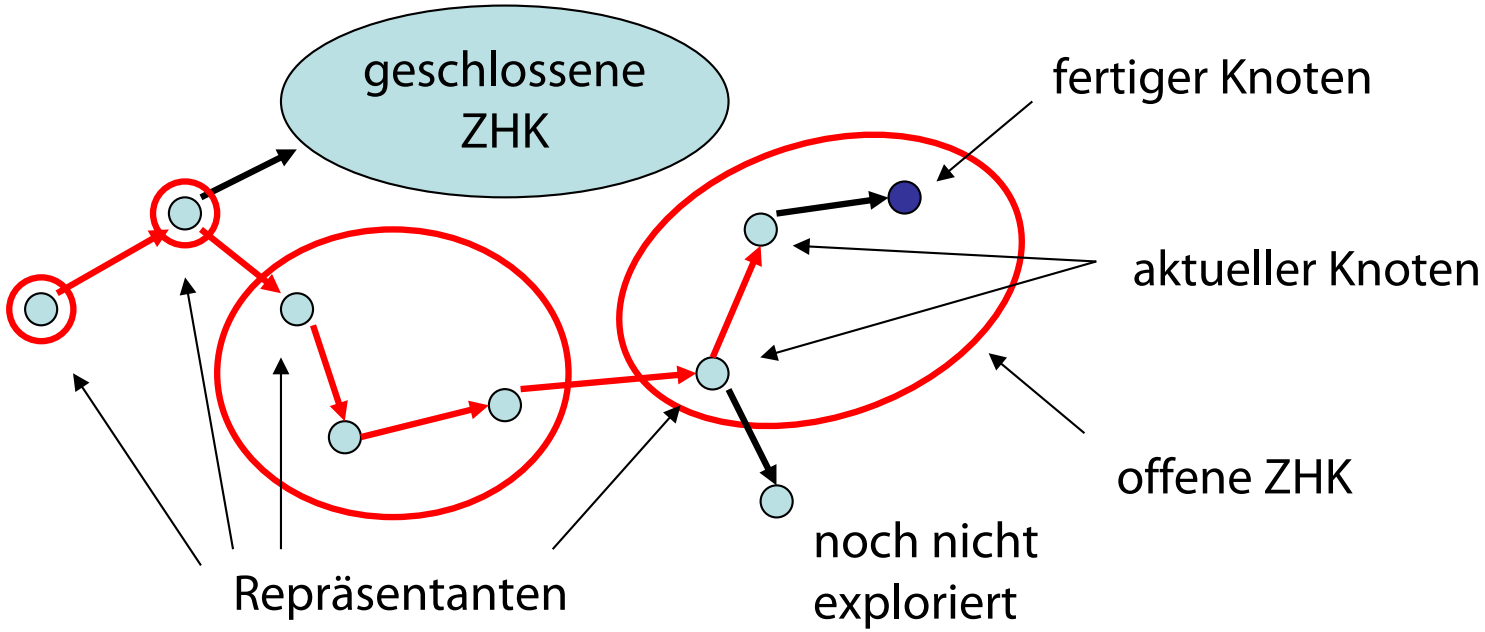
- Anfangs gelten alle Invarianten
- Fall 4: Kante zu nicht exploriertem Knoten



Starke ZHKs

Beweis über vollständige Induktion.

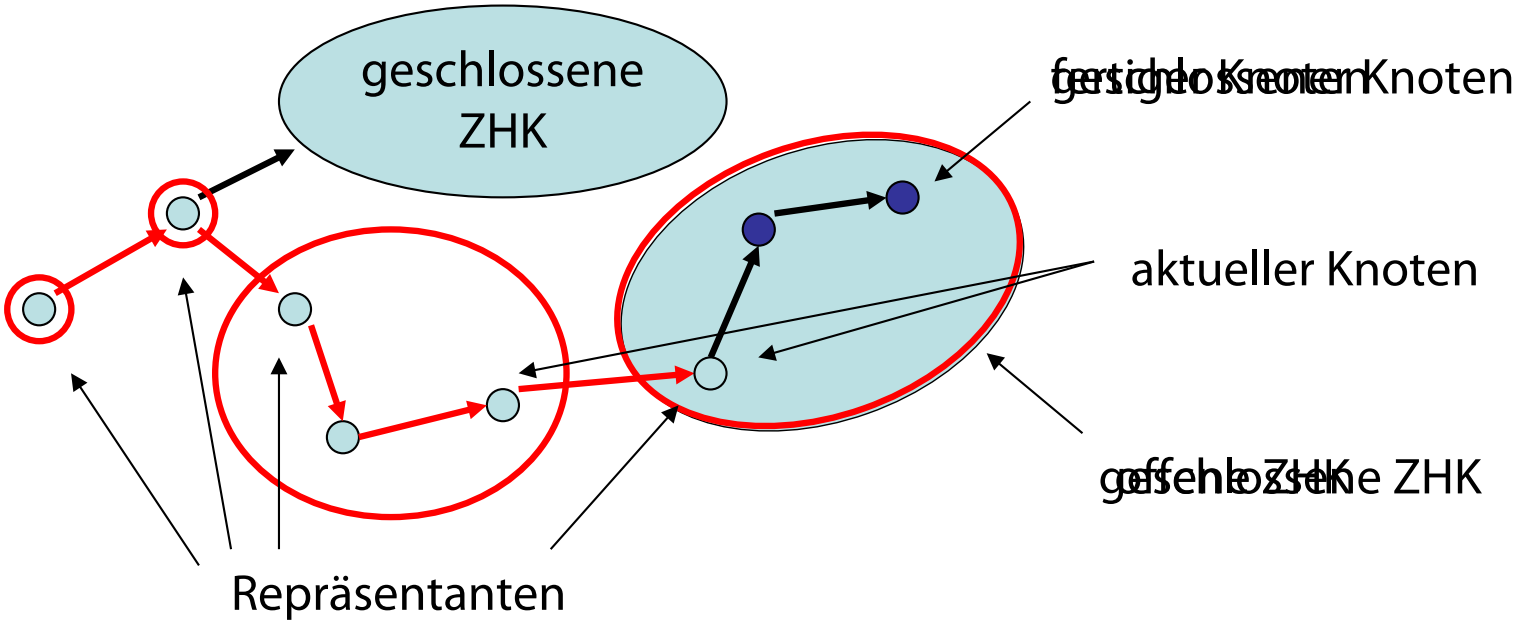
- Anfangs gelten alle Invarianten
- Fall 5: Knoten exploriert



Starke ZHKs

Beweis über vollständige Induktion.

- Anfangs gelten alle Invarianten
- Fall 5: Knoten exploriert



Starke ZHKs

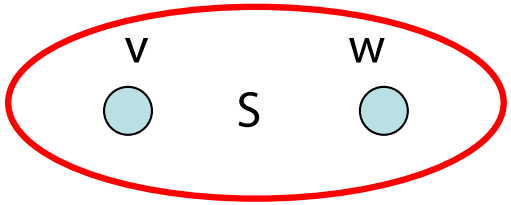
Behauptung: Eine geschlossene ZHK G_c im besuchten Teilgraphen C von G ist eine ZHK in G.

Beweis:

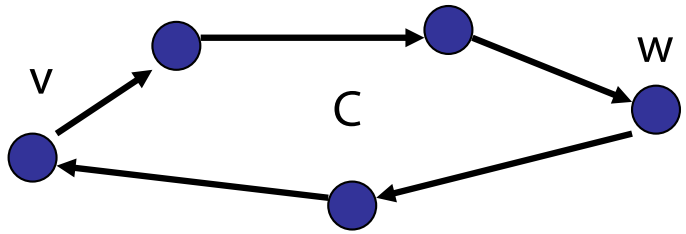
- v: geschlossener Knoten
- S: ZHK in G, die v enthält
- S_c: ZHK in G_c, die v enthält
- Es gilt: $S_c \subseteq S$
- Zu zeigen: $S \subseteq S_c$

Starke ZHKs

Beweis:



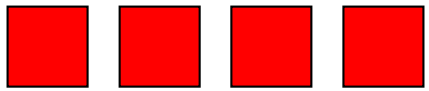
- w: beliebiger Knoten in S
- Es gibt gerichteten Kreis C durch v und w
- Nutze Invariante 1: alle Knoten in C geschlossen



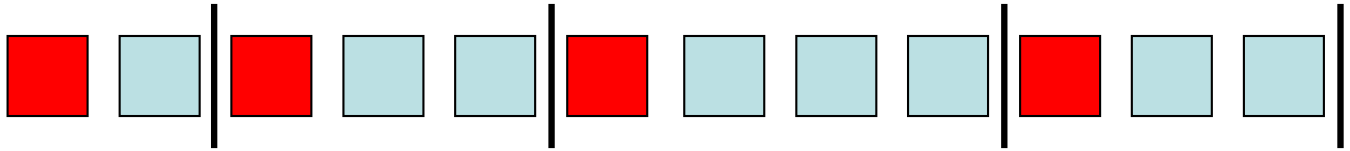
- Da alle Kanten geschlossener Knoten exploriert worden sind, ist C in G_c und daher $w \in S_c$

Wiederholung 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.

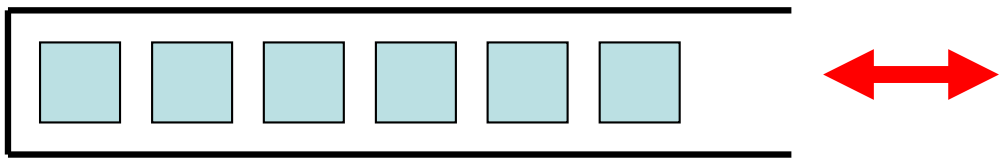


Starke ZHKs

Invarianten 2 und 3:

einfache Methode, um offene ZHKs in G_c zu repräsentieren:

- Wir verwalten Folge oNodes aller offenen (nicht geschl.) Knoten in steigender DFS-Nummer und eine Teilfolge oReps aller offenen ZHK-Repräsentanten
- Stack ausreichend für beide Folgen



Wiederholung: Tiefensuche-Schema

```
Übergeordnete Prozedur = Bestimme ZHKs
    unmark all nodes
    init()
    foreach s∈V do  // stelle sicher, dass alle Knoten besucht werden
      if s is not marked then
        mark s
        root(s)
        DFS(s,s)    // s: Startknoten

Procedure DFS(u,v: Node)  // u: Vater von v
    foreach (v,w)∈E do
      if w is marked then traverseNonTreeEdge(v,w)
      else traverseTreeEdge(v,w)
        mark w
        DFS(v,w)
    backtrack(u,v)
```

Prozeduren in rot: noch zu spezifizieren

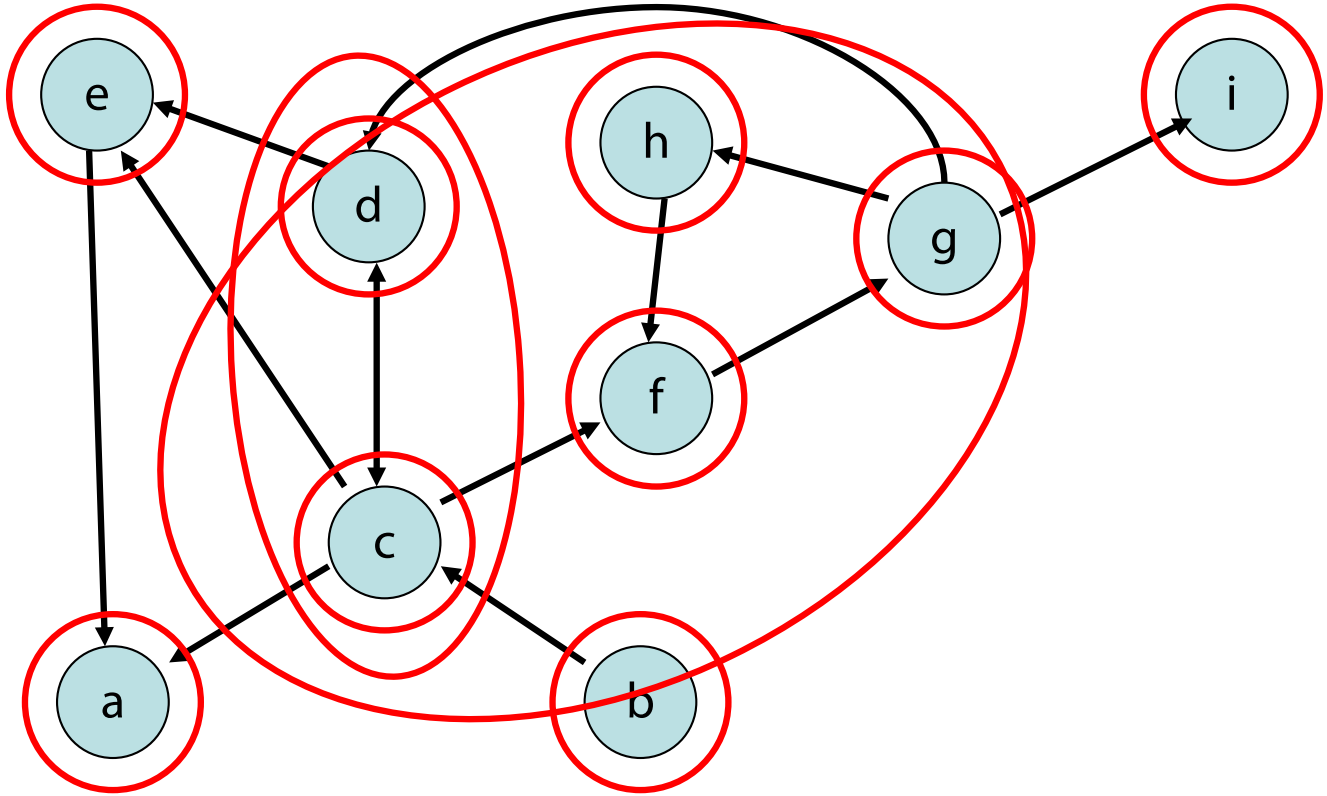
Starke ZHKs

```
init():
    component: Array [1..n] of NodeId
    oReps = <>: Stack of NodeId
    oNodes = <>: Stack of NodeId
    dfsPos:=1

root(w) oder traverseTreeEdge(v,w):
    push(w, oReps)     // neue ZHK
    push(w, oNodes)   // neuer offener Knoten
    dfsNum[w]:=dfsPos; dfsPos:=dfsPos+1
```

Starke ZHKs

```
traverseNonTreeEdge(v,w):
    if w ∈ oNodes then  // kombiniere ZHKs
       while dfsNum[w] < dfsNum[top(oReps)] do
          pop(oReps)


backtrack(u,v):
    if v = top(oReps) then  // v Repräsentant?
       pop(oReps)  // ja: entferne v
       repeat          // und offene Knoten bis v
          w:= pop(oNodes)
          component[w]:=v
       until w=v
```

Starke ZHKs - Beispiel



oNodes: | b | c | d | f | g | h | i |

oReps: | b | c | i | g |

Starke ZHKs

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

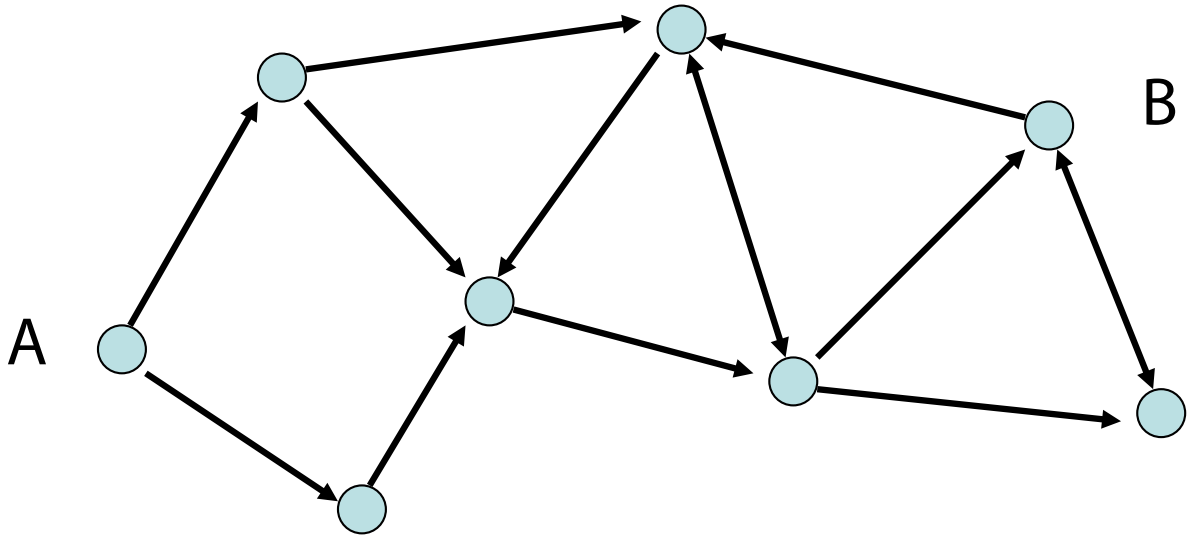
Beweis:

- init, root, traverseTreeEdge: Zeit O(1)
- Backtrack, traverseNonTreeEdge: 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?



Kürzeste Wege

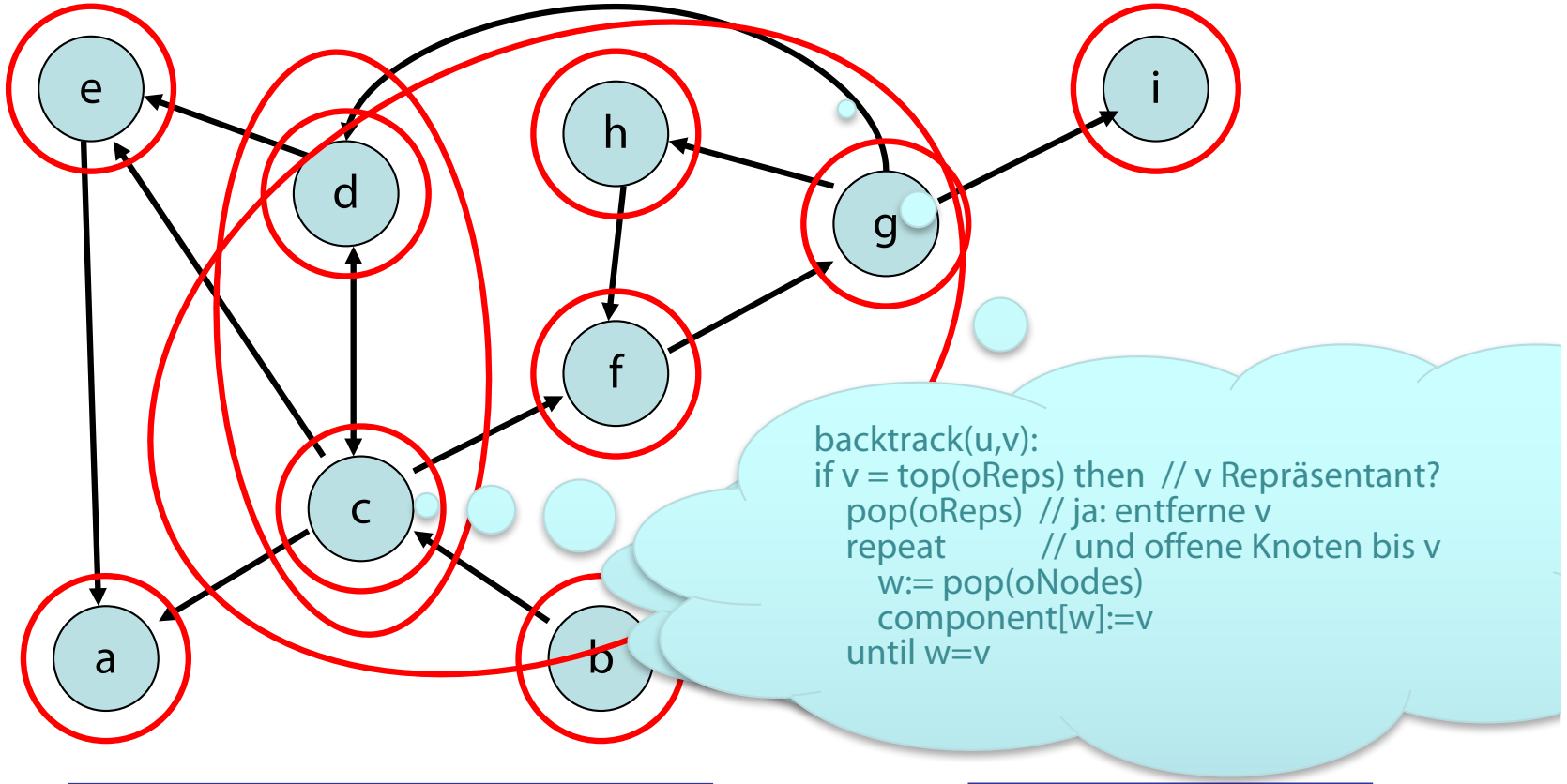
Zentrale Frage:

Wie komme ich am schnellsten von A nach B?

Fälle:

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

Starke ZHKs - Wiederholung



oNodes: | b | c | d | f | g | h | i |

oReps: | b | c | i | g |

Kürzeste Wege

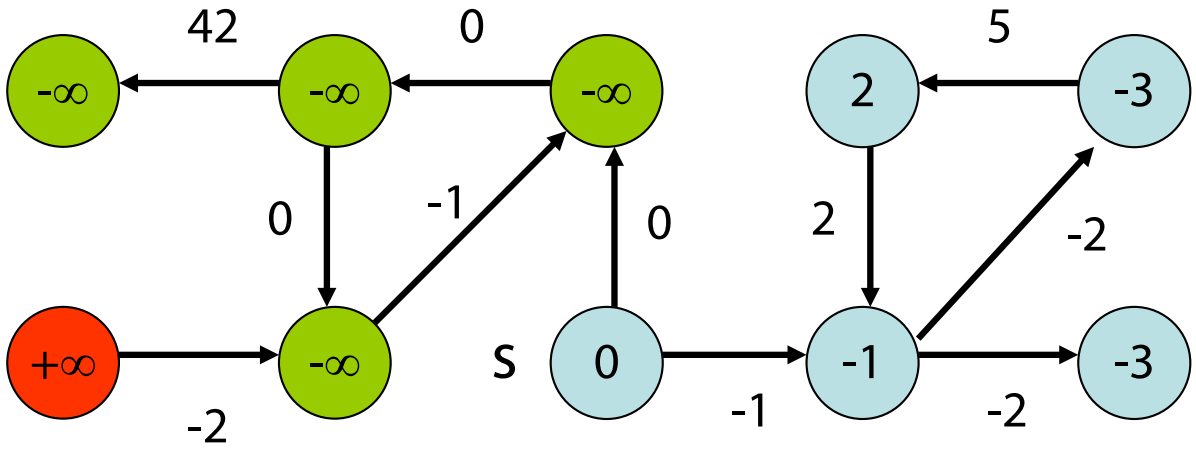
Kürzeste-Wege-Problem:

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

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

H. C. Joksch. *The shortest route problem with constraints.*
J. Math. Analysis and Applications. 14, Seite 191–197, **1966**

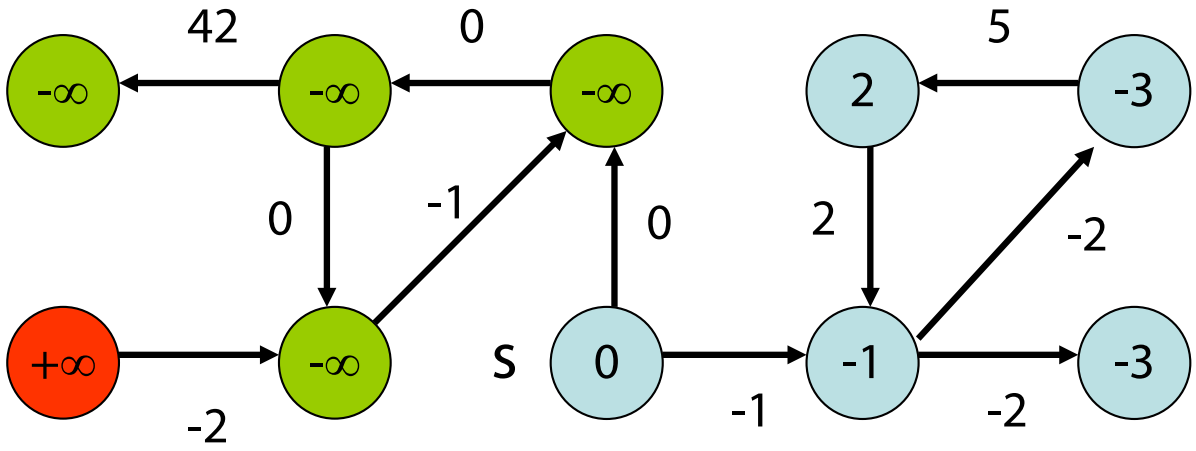
Kürzeste Wege



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

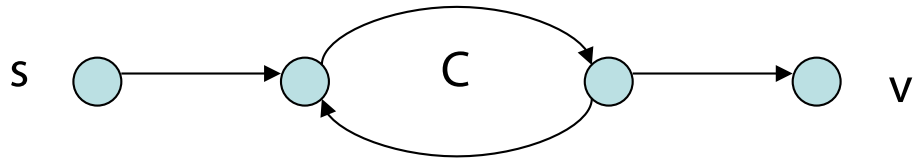
$$\mu(s,v) = \begin{cases} \infty & \text{kein Weg von s nach v} \\ -\infty & \text{Weg bel. kleiner Kosten von s nach v} \\ \min\{\, c(p) \mid p \text{ ist Weg von s nach v} \} \end{cases}$$

Kürzeste Wege



Wann sind die Kosten -∞?

Wenn es einen negativen Kreis gibt:

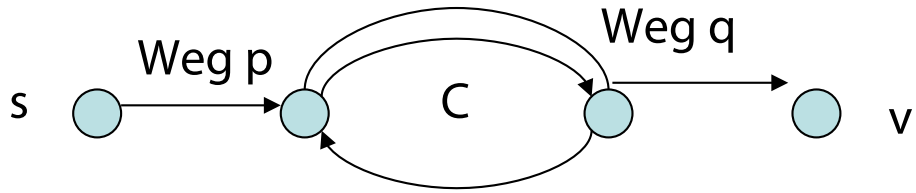


c(C)<0

Kürzeste Wege

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

Negativer Kreis hinreichend:



$c(C)<0$

Kosten für i-fachen Durchlauf von C:

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

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

Kürzeste Wege

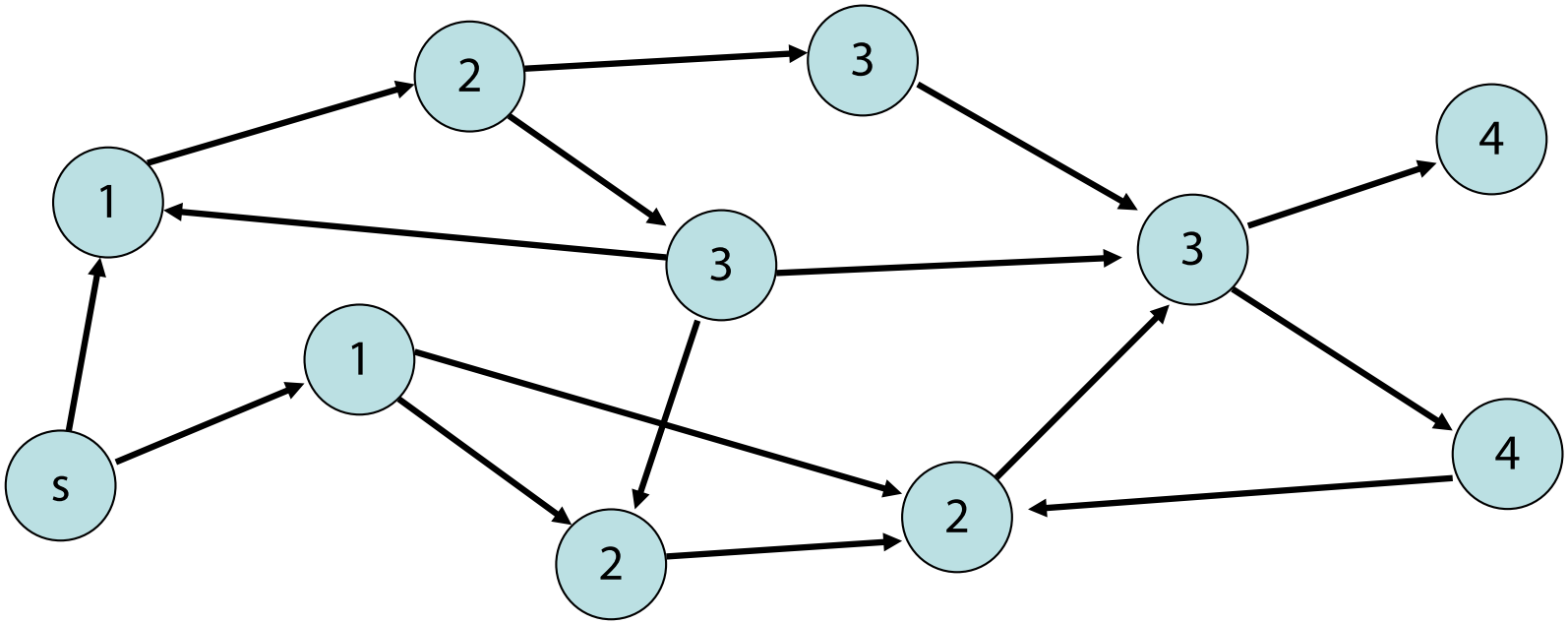
Negativer Kreis notwendig:

- 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

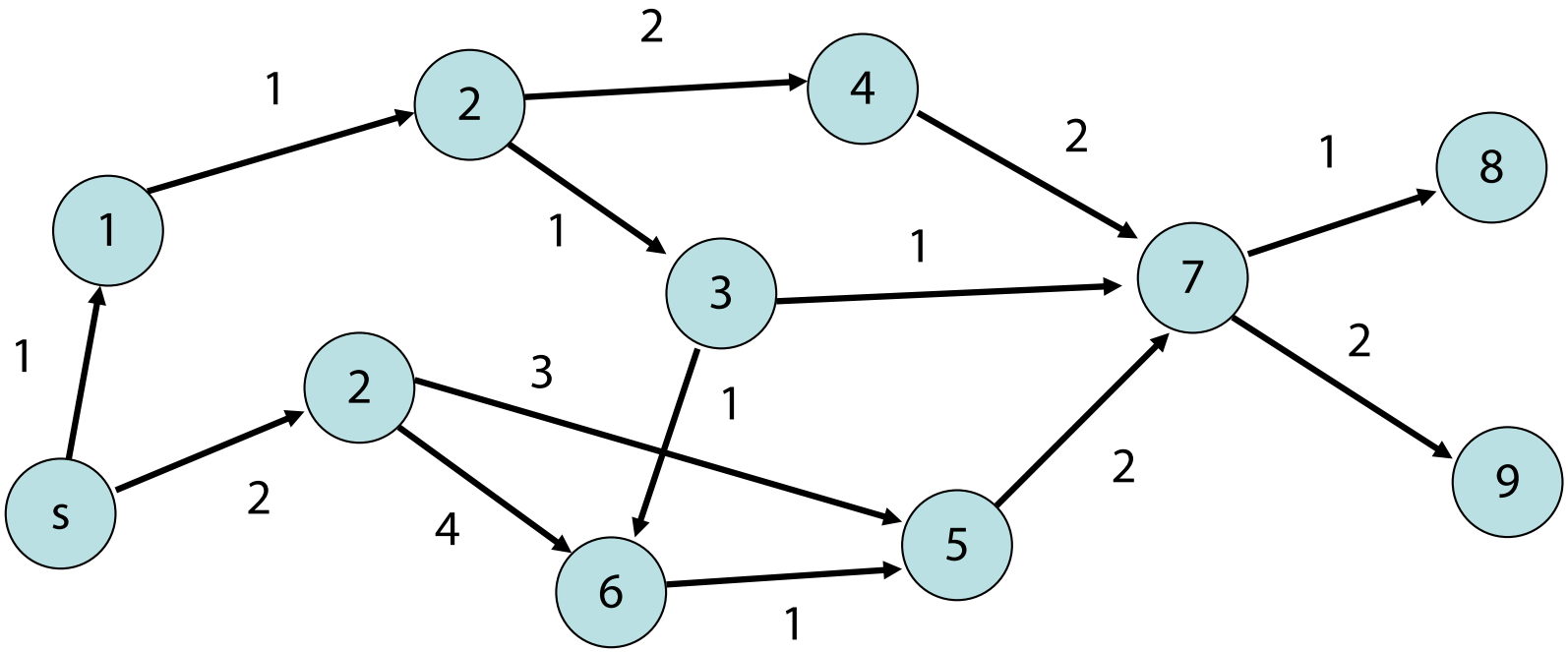
Graph mit Kantenkosten 1:

Führe Breitensuche durch.



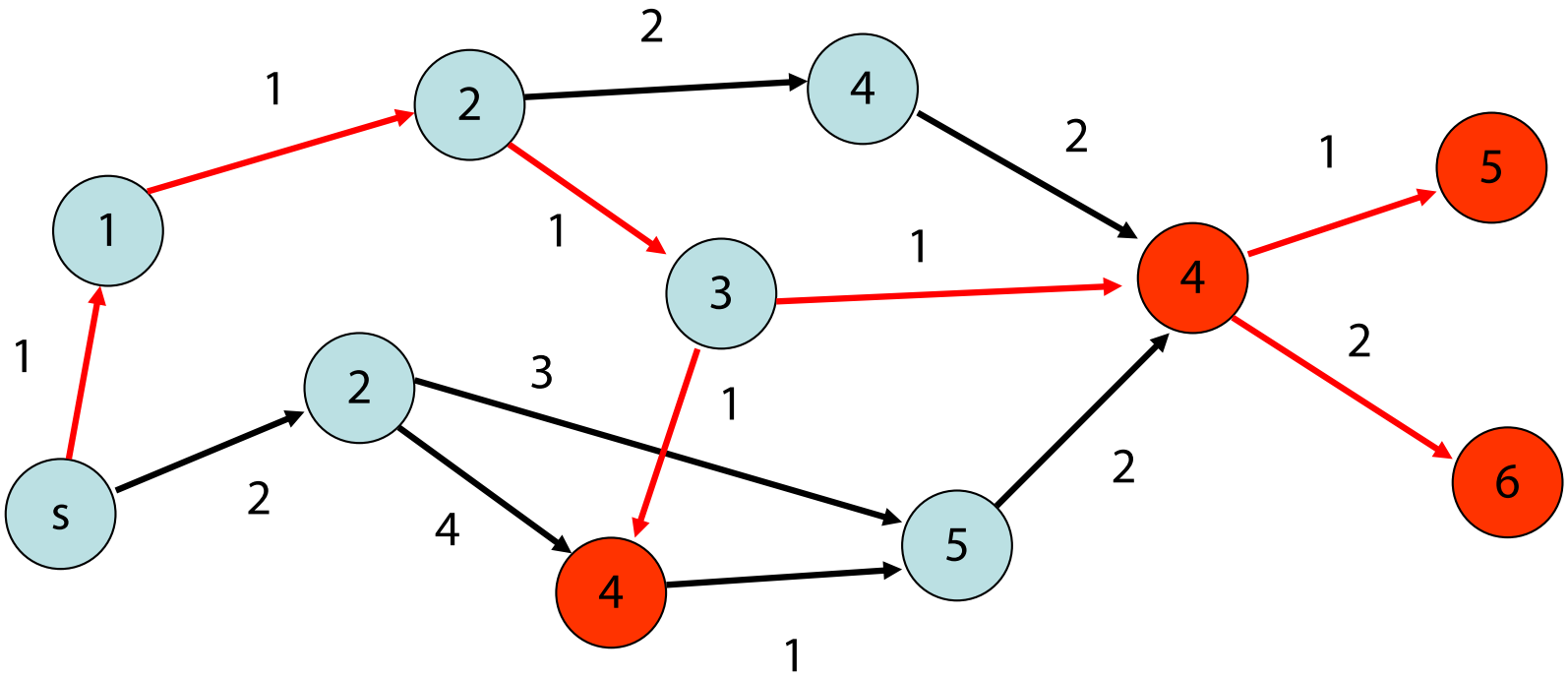
Kürzeste Wege in DAGs

- Reine Breitensuche funktioniert nicht, wenn Kantenkosten nicht gleich 1.



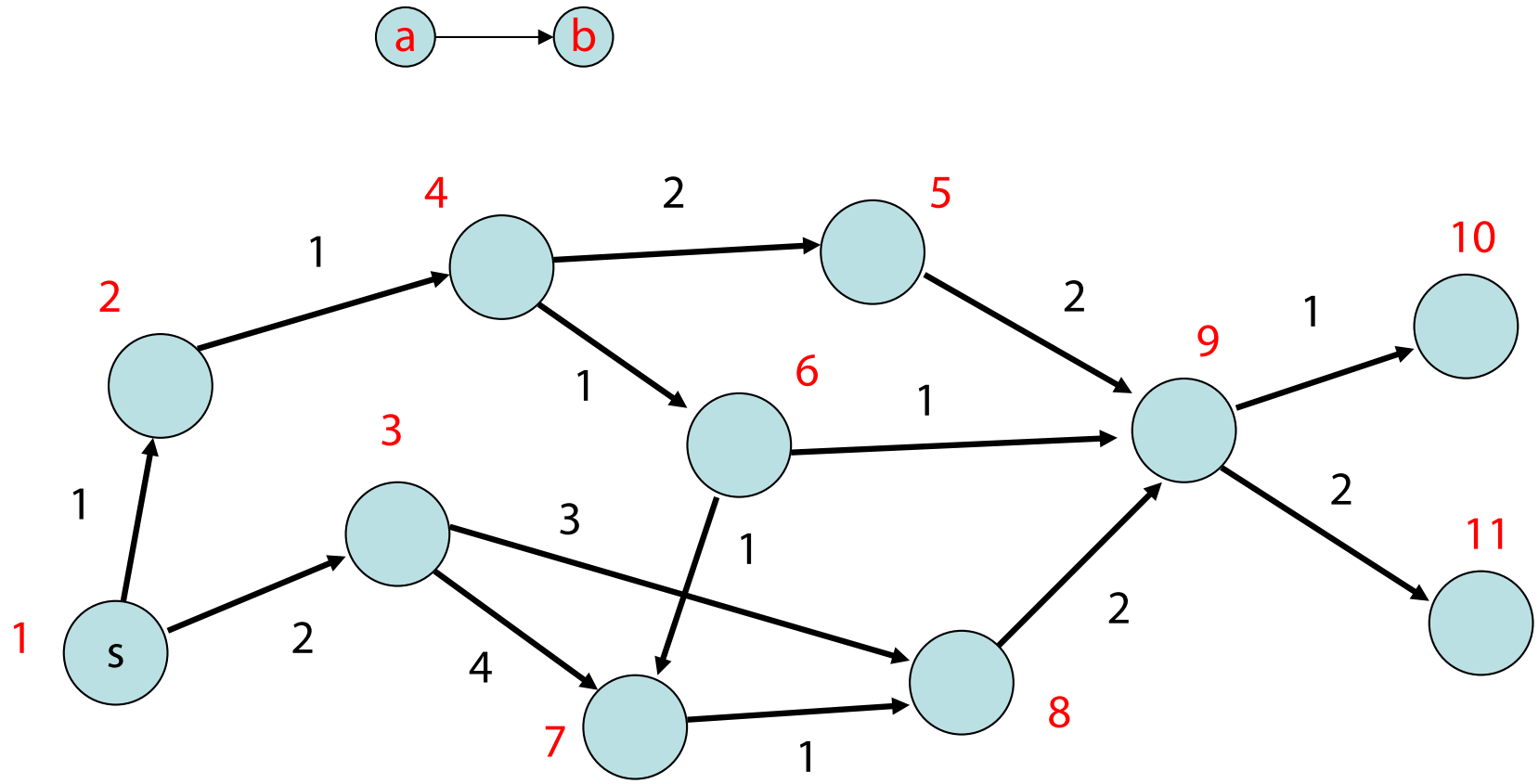
Kürzeste Wege in DAGs

Korrekte Distanzen:



Kürzeste Wege in DAGs

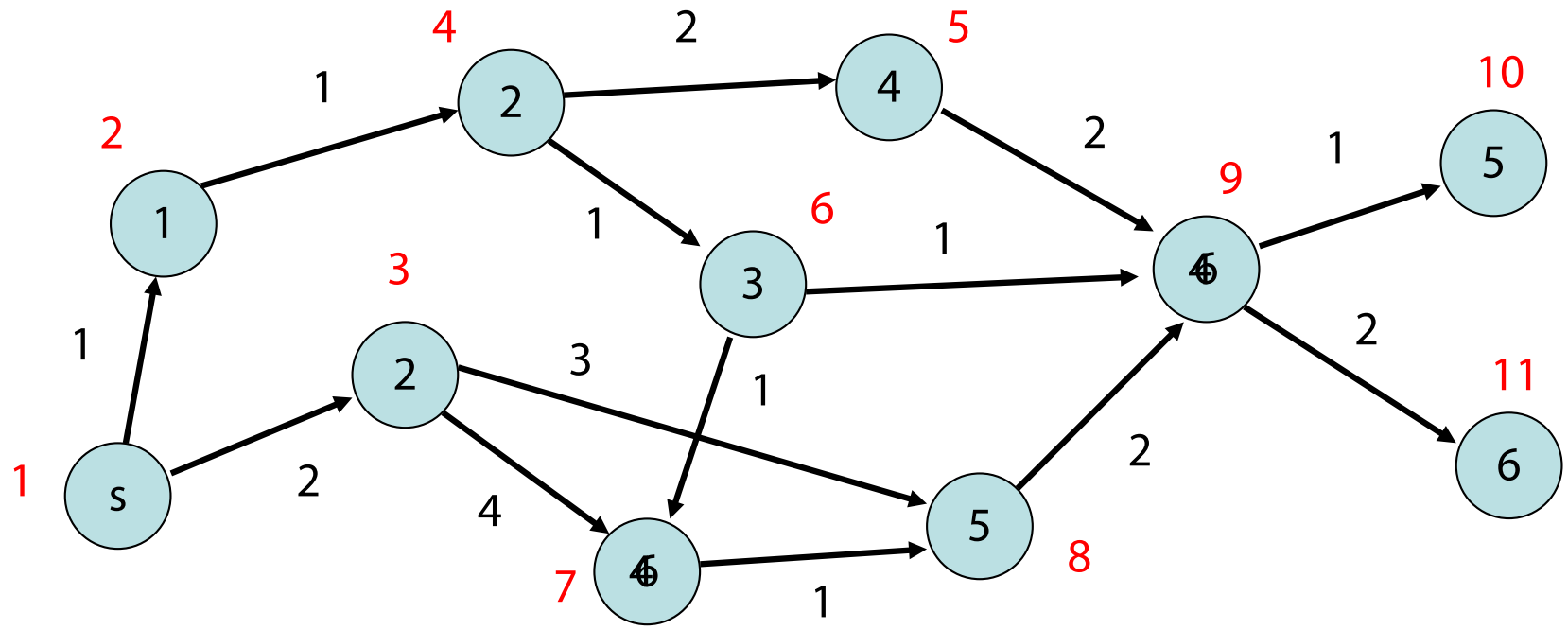
Strategie: nutze aus, dass Knoten in DAGs topologisch sortiert werden können (alle Kanten erfüllen a<b)



Kürzeste Wege in DAGs

Strategie:

Betrachte dann Knoten in der Reihenfolge ihrer topologischen Sortierung und aktualisiere Distanzen zu s



Kürzeste Wege in DAGs

Strategie:

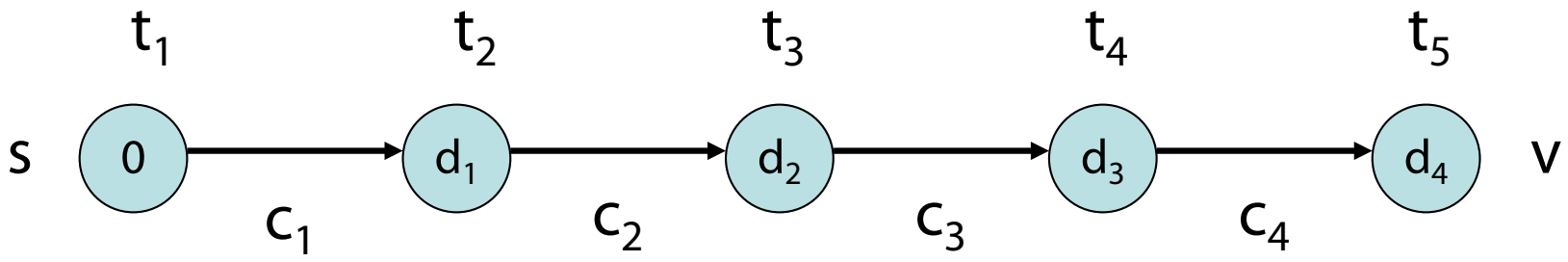
1. Topologische Sortierung der Knoten
2. Aktualisierung der Distanzen gemäß der topologischen Sortierung

Warum funktioniert das??

Kürzeste Wege in DAGs

Betrachte kürzesten Weg von s nach v.

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

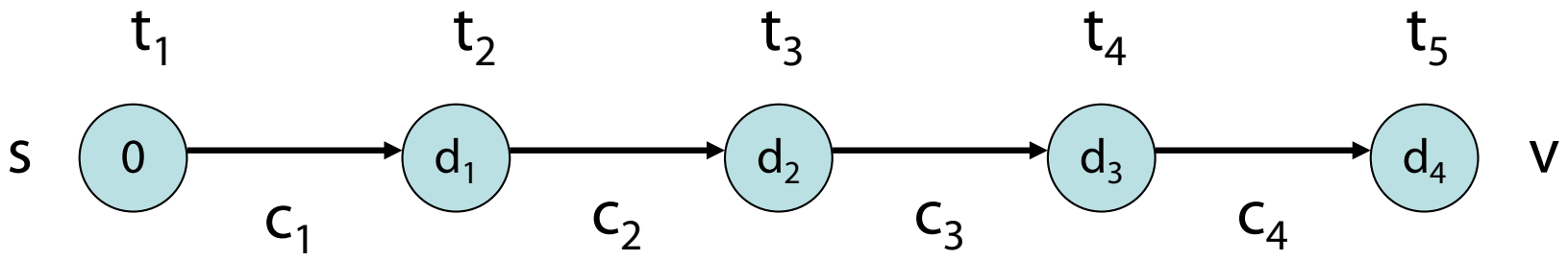


Besuch in topologischer Reihenfolge führt zu richtigen Distanzen ($d_i = \Sigma_{j\leq i}\, c_i$).

Kürzeste Wege in DAGs

Betrachte kürzesten Weg von s nach v.

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



Bemerkung: kein Knoten t_{i+1} auf dem Weg zu v
kann Distanz $< d_i$ zu s haben,
da sonst kürzerer Weg zu v bzw. t_{i+1} möglich wäre.

Kürzeste Wege in Graphen

Allgemeine Strategie:

- Am Anfang, setze $d(s):=0$ und $d(v):=\infty$ für alle Knoten $v \in V \setminus \{s\}$
- Besuche Knoten in einer Reihenfolge, die sicherstellt, dass mindestens ein kürzester Weg von s zu jedem v in der Reihenfolge seiner Knoten besucht wird
- Für jeden besuchten Knoten v, aktualisiere die Distanzen der Knoten w mit $(v,w) \in E$, d.h. setze $d(w) := \min\{d(w), d(v)+c(v,w)\}$

Kürzeste Wege in DAGs

Zurück zur Strategie:

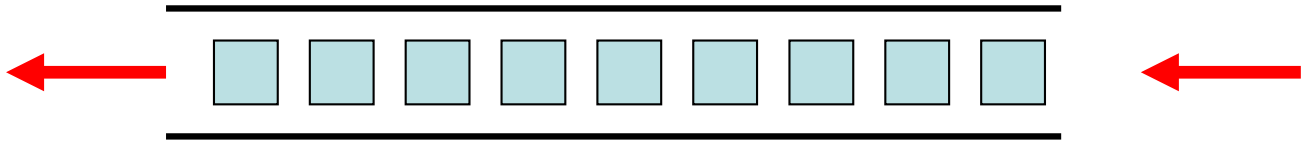
1. Topologische Sortierung der Knoten
2. Aktualisierung der Distanzen gemäß der topologischen Sortierung

Wie führe ich eine topologische Sortierung durch?

Kürzeste Wege in DAGs

Topologische Sortierung:

- Verwende eine FIFO Queue q

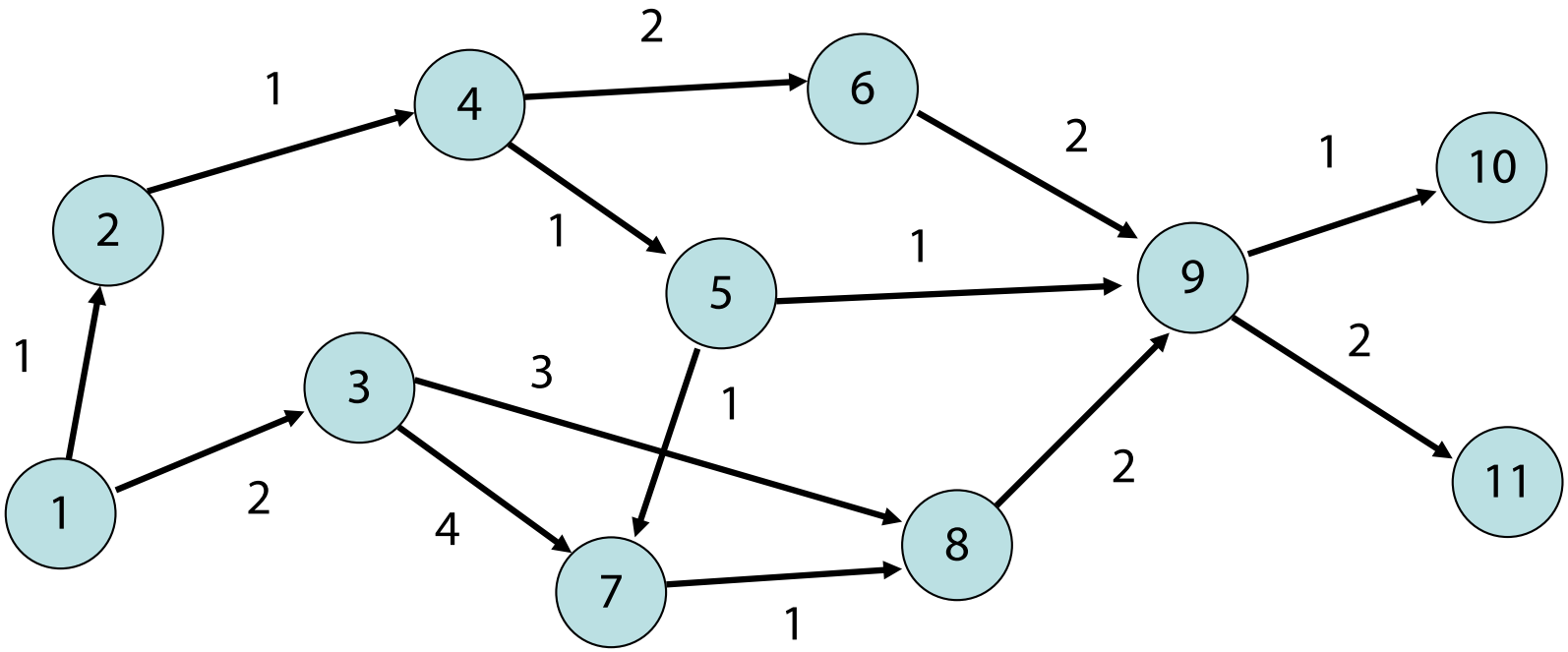


- Anfangs enthält q alle Knoten, die keine eingehende Kante haben (Quellen).
- Entnehme v aus q und markiere alle (v,w) $\in$ E. Falls alle Kanten nach w markiert sind und w noch nicht in q war, füge w in q ein. Wiederhole das, bis q leer ist.

Kürzeste Wege in DAGs

Beispiel:

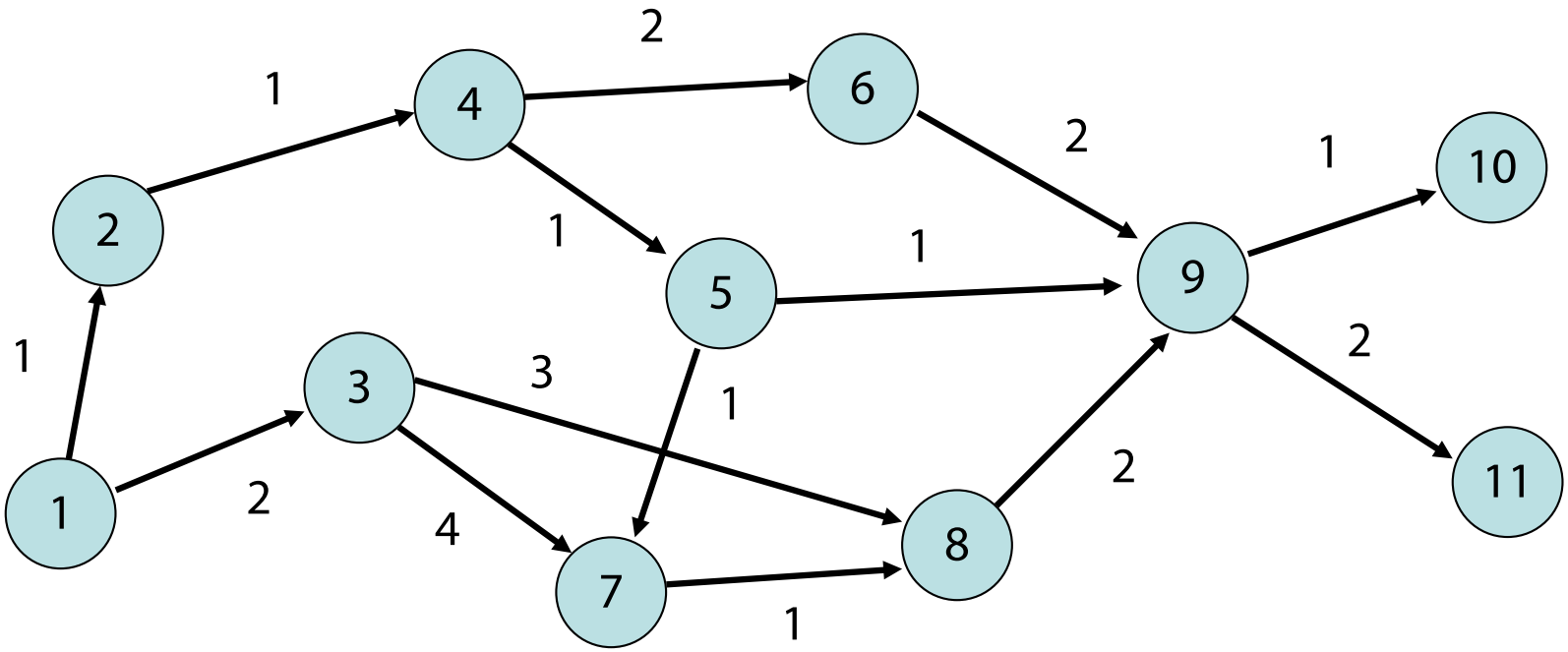
- : Knoten momentan in Queue q
- Nummerierung nach Einfügereihenfolge



Kürzeste Wege in DAGs

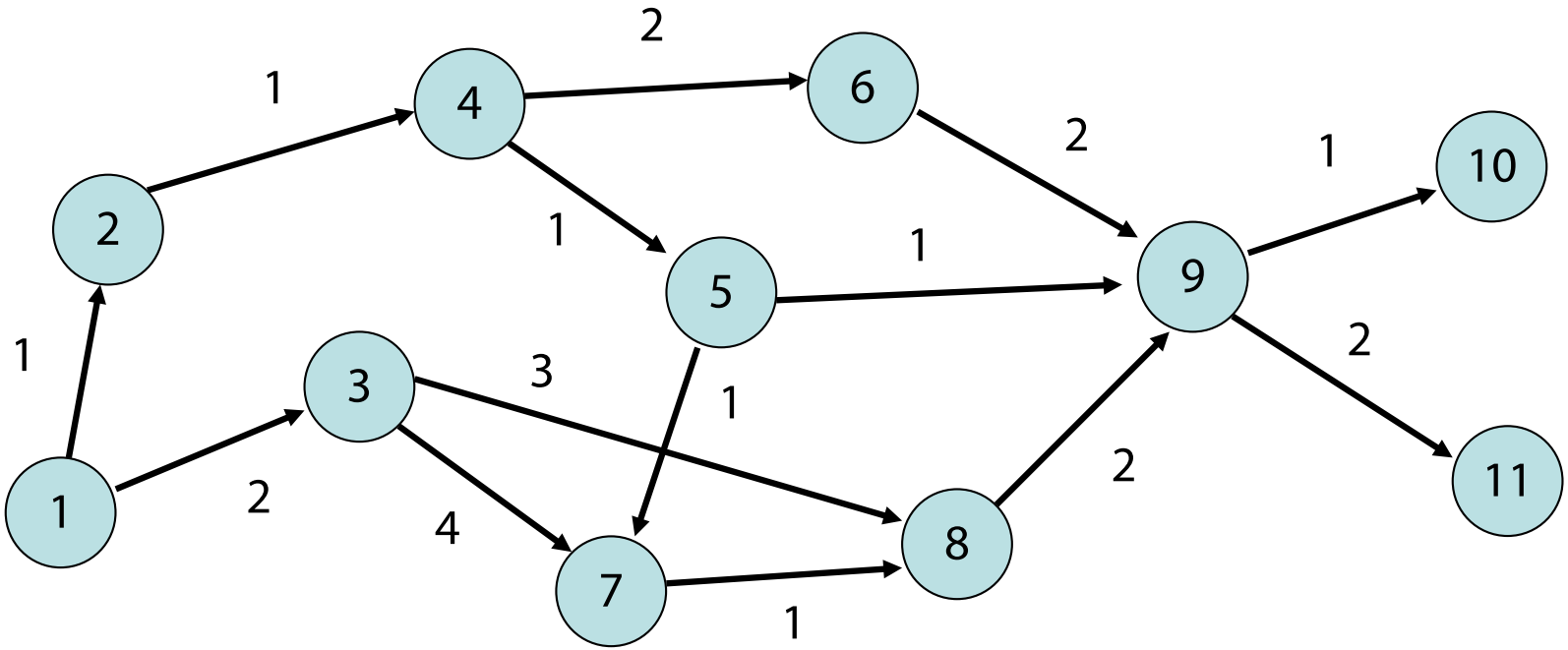
Korrektheit der topologischen Nummerierung:

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



Kürzeste Wege in DAGs

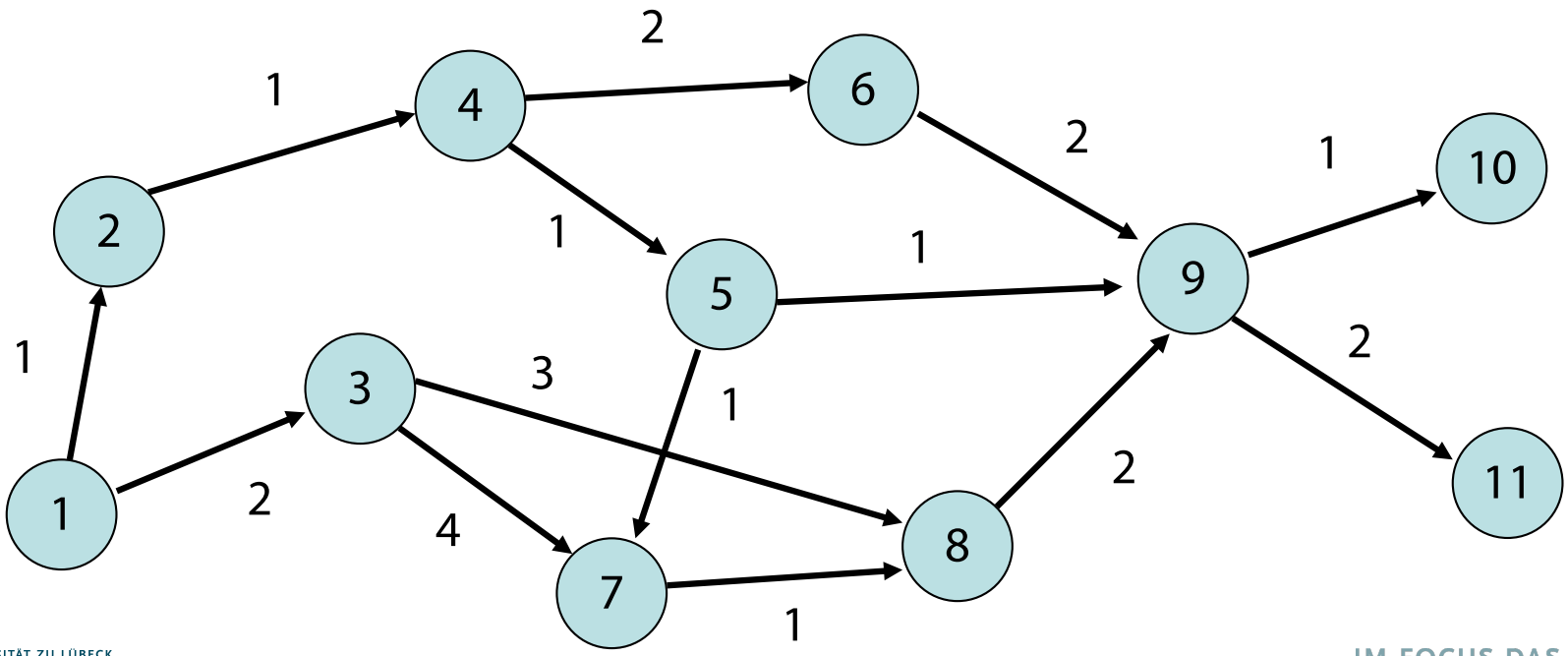
Laufzeit: Zur Bestimmung aller Knoten ohne eingehende Kante muss Graph einmal durchlaufen werden. Danach wird jeder Knoten und jede Kante genau einmal betrachtet, also Zeit O(n+m).



Kürzeste Wege in DAGs

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

Test auf DAG-Eigenschaft



Kürzeste Wege in DAGs

DAG-Strategie:

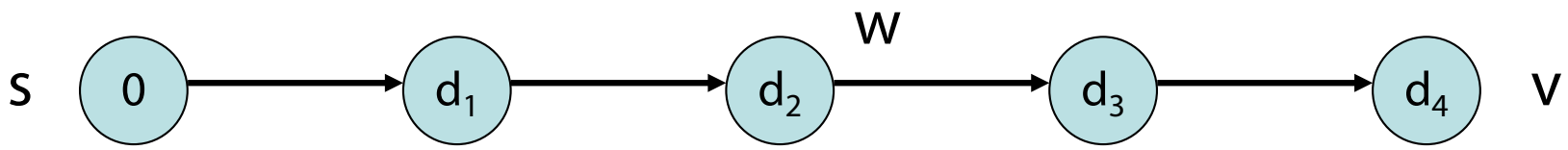
1. Topologische Sortierung der Knoten
   Laufzeit O(n+m)
2. Aktualisierung der Distanzen gemäß der topologischen Sortierung
   Laufzeit O(n+m)

Insgesamt Laufzeit O(n+m).

Dijkstras Algorithmus

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

Problem: besuche Knoten eines kürzesten Weges in richtiger Reihenfolge



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

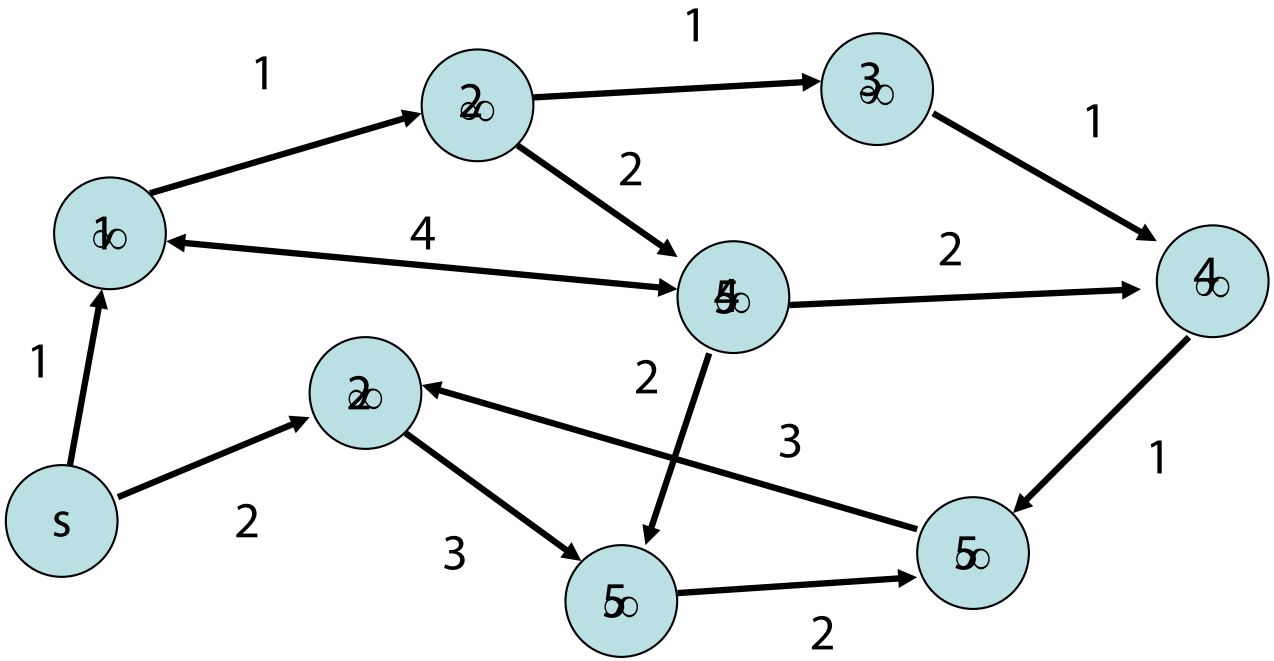
Edsger W. Dijkstra: A note on two problems in connexion with graphs. In: Numerische Mathematik. 1, S. 269–271, **1959**

Dijkstras Algorithmus

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

Dijkstras Algorithmus

Beispiel: (: aktuell, : fertig)



Dijkstras Algorithmus

```
Procedure Dijkstra(s: NodeId)
    d=<∞,…,∞>: NodeArray of ℝ ∪ {-∞,∞}
    d[s]:=0
    q=<s>: PQ mit Schlüssel d so dass d(x) = d[x]
    while q ≠ <> do
       u := deleteMin(q)  // u: min. Distanz zu s in q
       foreach e=(u,v) ∈ E do
          if d[v] > d[u]+c(e) then // aktualisiere d[v]
             dv' :=d[v]; d[v]:=d[u]+c(e)
             if dv'=∞ then insert(v, q) // v schon in q?
             else decreaseKey(v, q, dv'-d[v])
```

Dijkstras Algorithmus

Laufzeit:

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

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

also $T_{Dijkstra} \in O((m+n)\log n)$

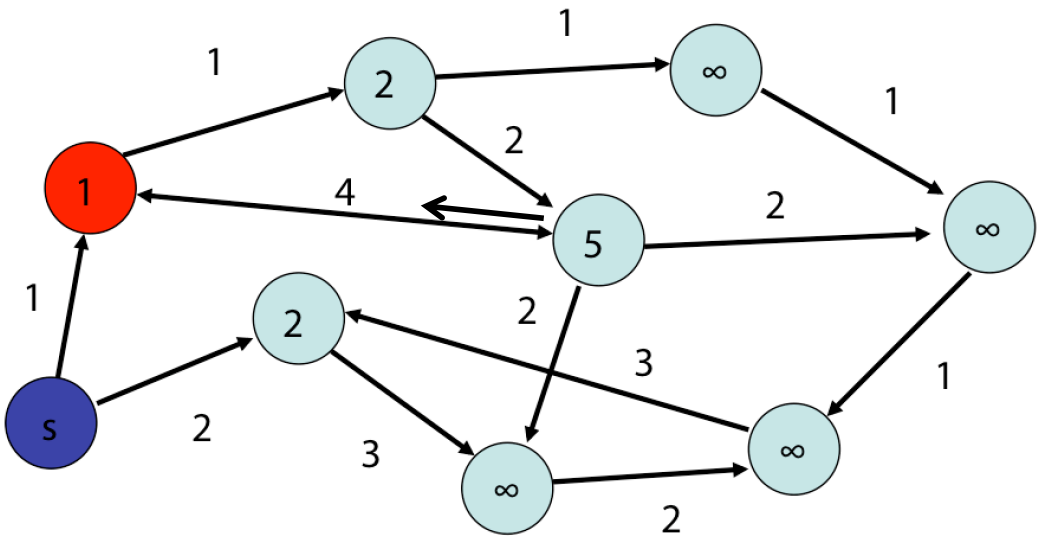
Fibonacci Heap:

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

Kürzeste Wege

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

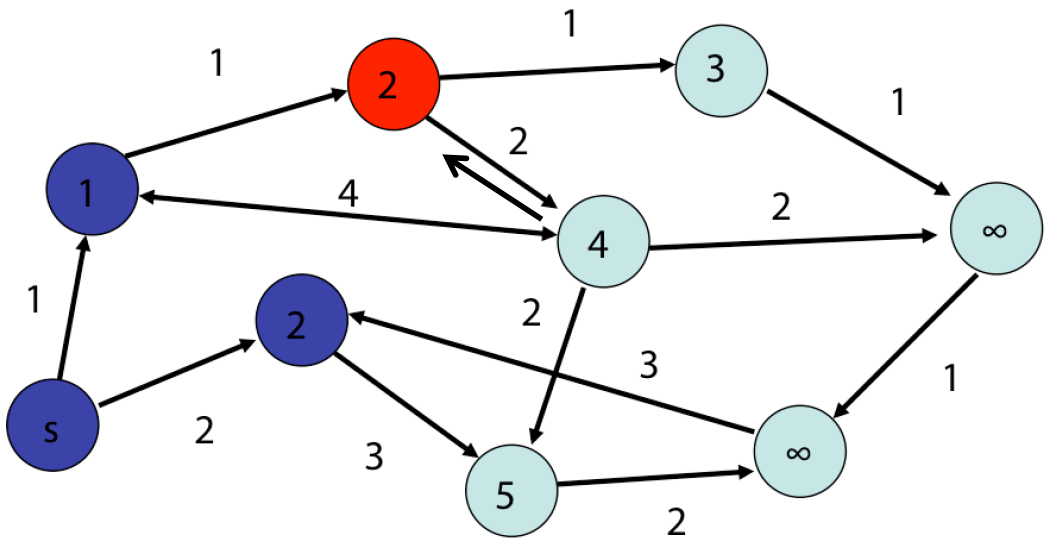
Beispiel: (: aktuell, : fertig)



Kürzeste Wege

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

Beispiel: (: aktuell, : fertig)

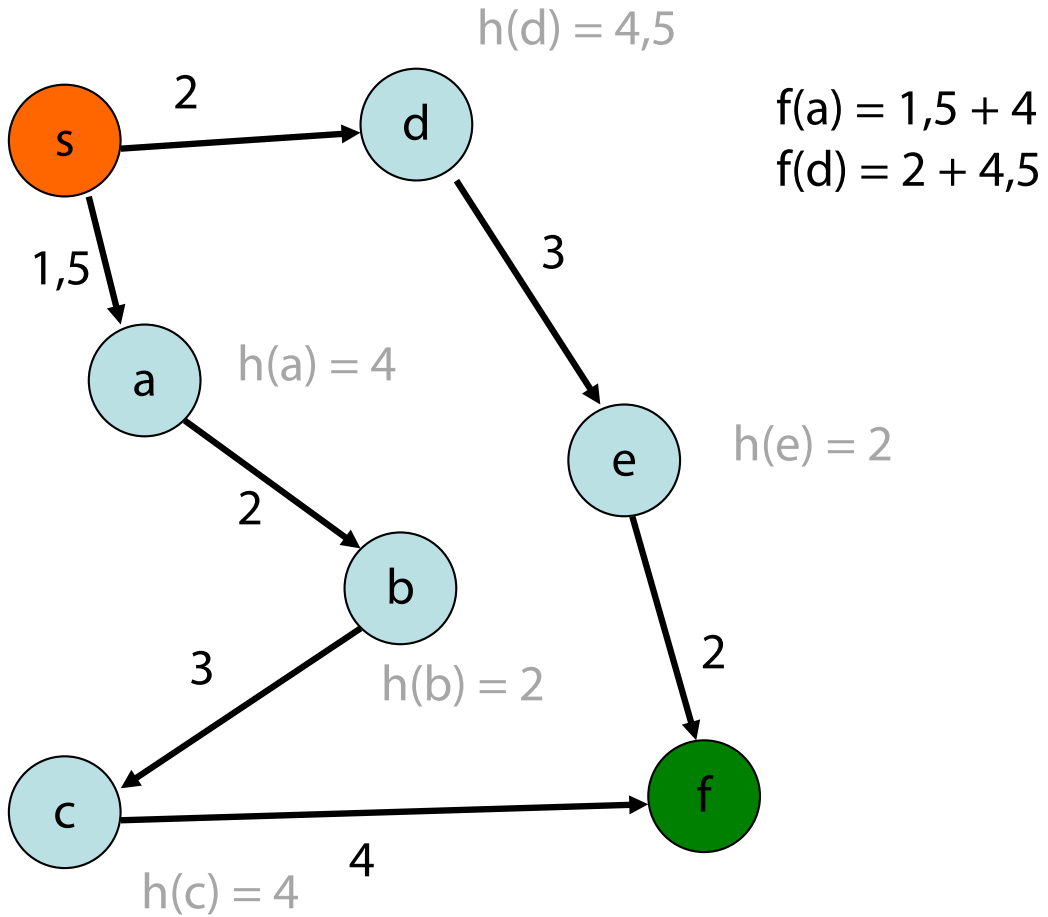


Kürzeste Wege: Heuristische Suche

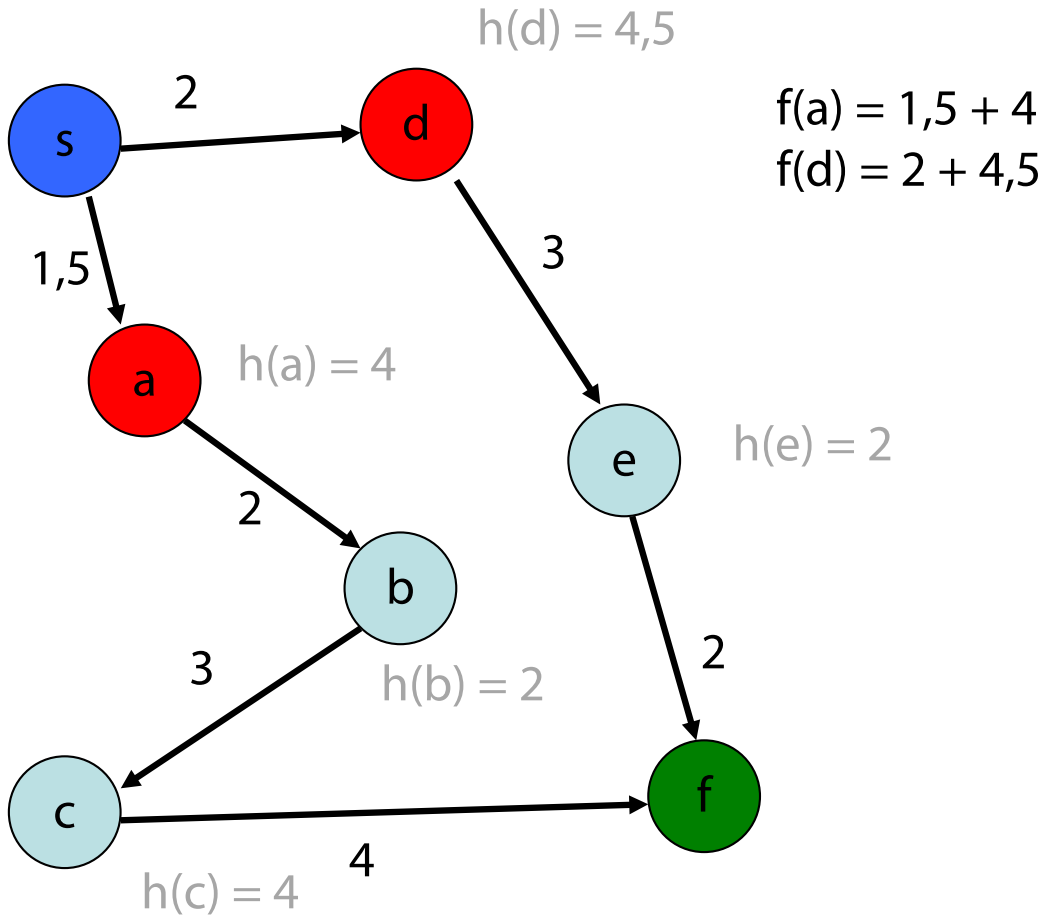
- Was sollten wir machen, wenn nur der kürzeste Weg zu einem gegebenen Knoten gesucht ist?
  - Es werden im Dijkstra-Algorithmus meist zu viele Knoten betrachtet (d.h. "expandiert")
- Keine Abschätzung der Entfernung zum Ziel
- Informierte Suche über Zielschätzer (Heuristik)
- A*-Algorithmus

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**

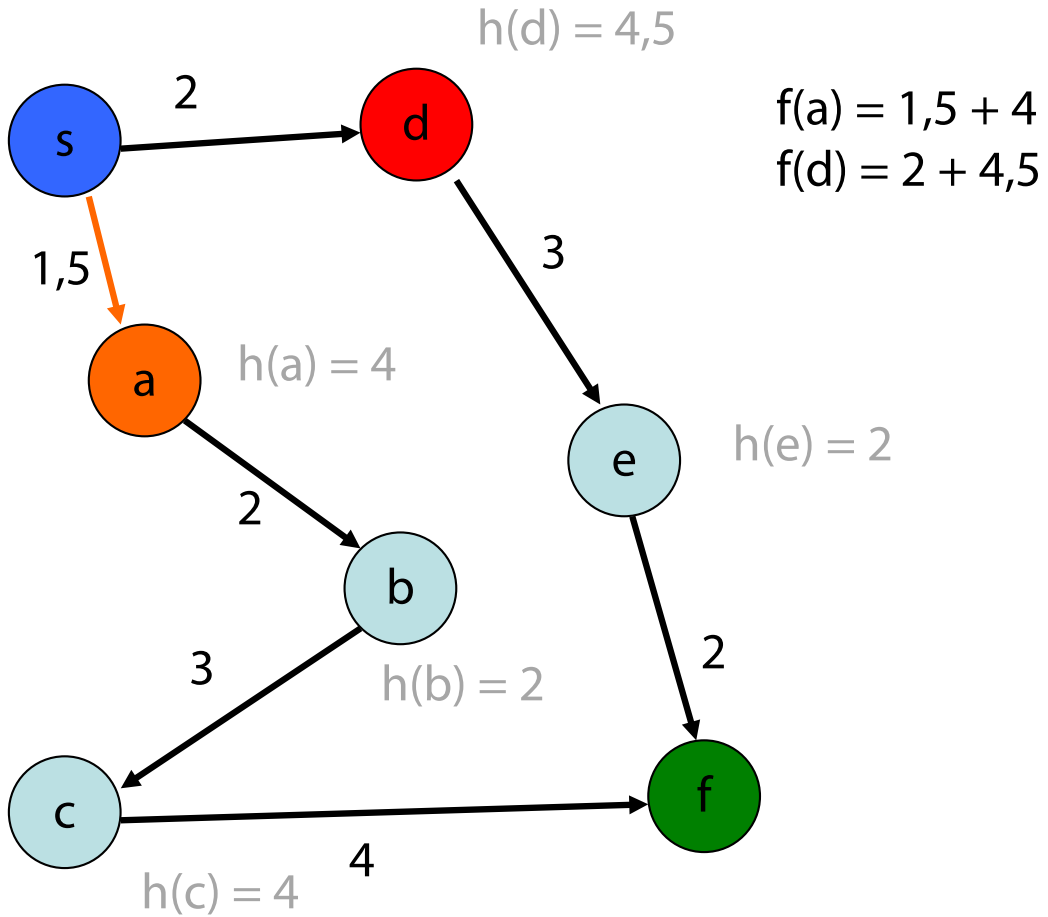
A* – Beispiel



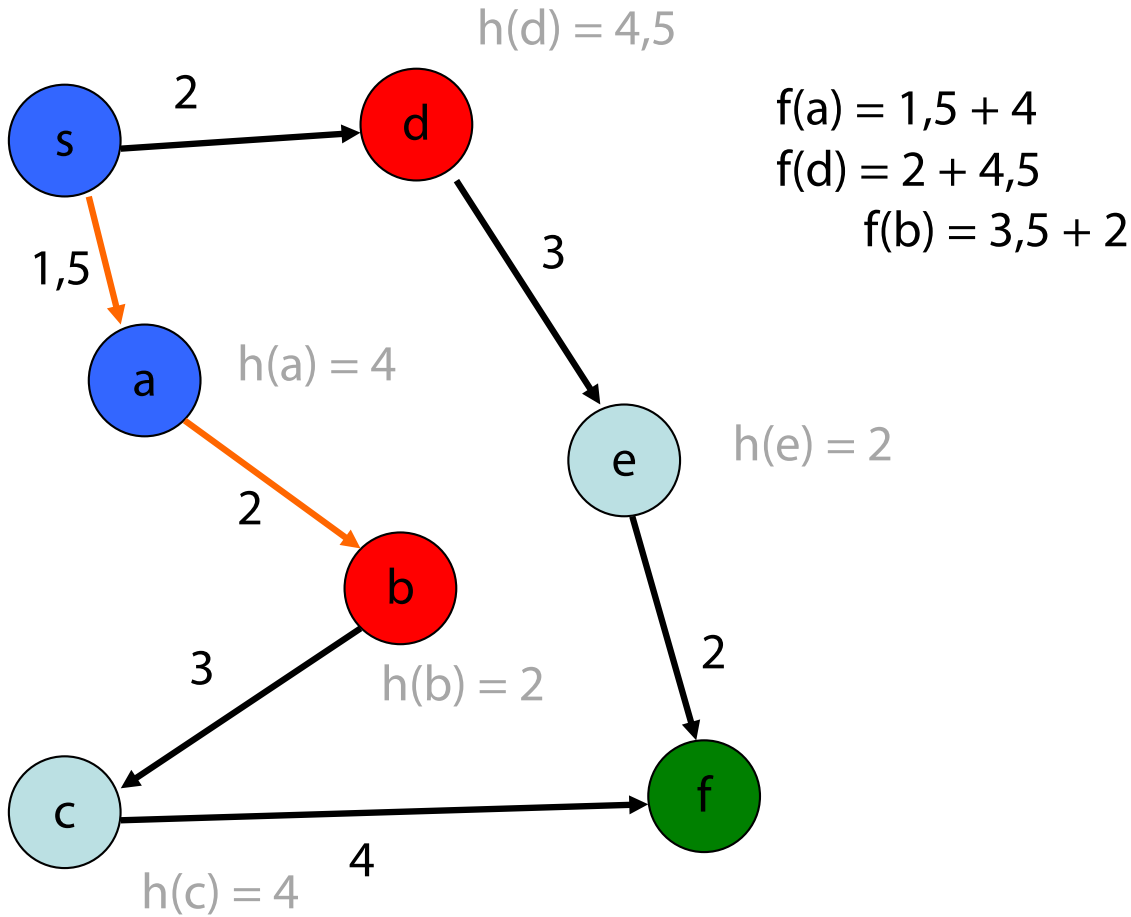
A* – Beispiel



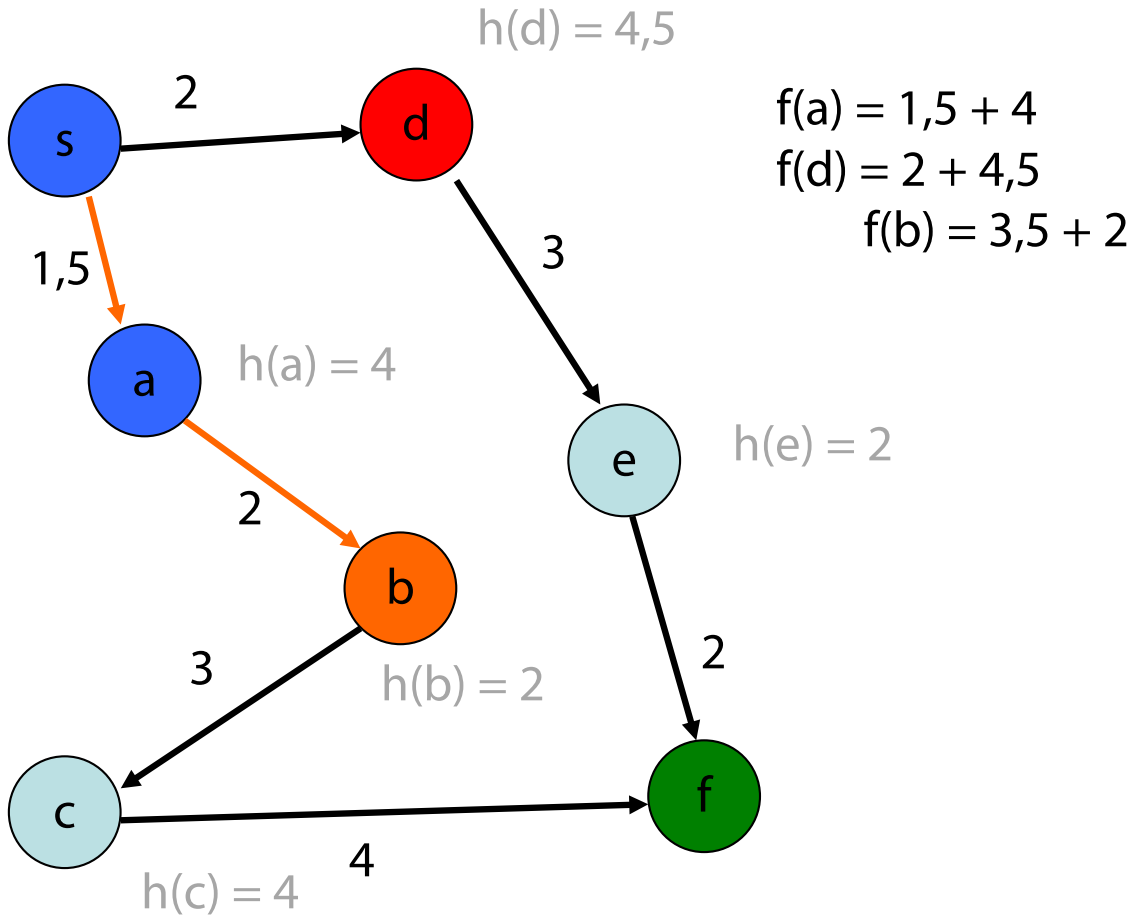
A* – Beispiel



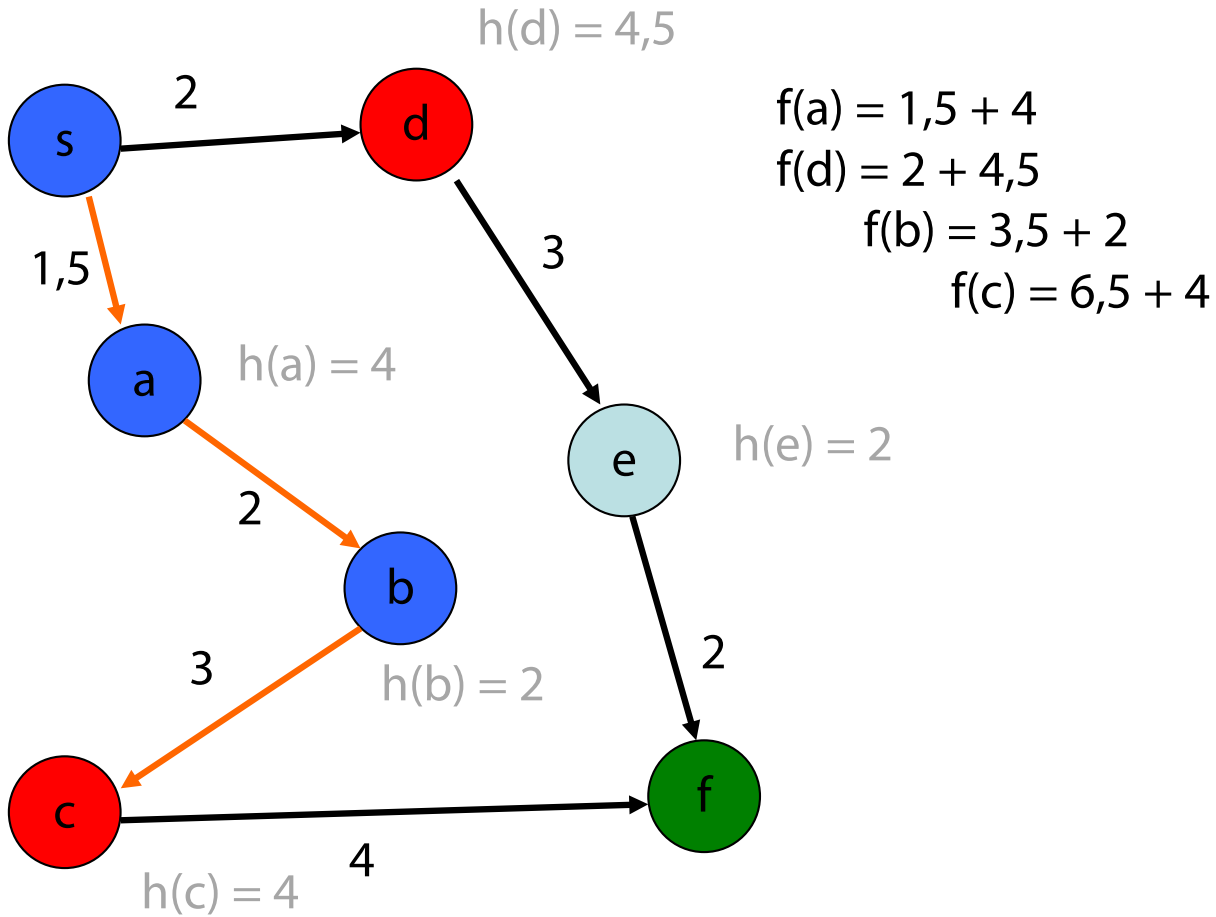
A* – Beispiel



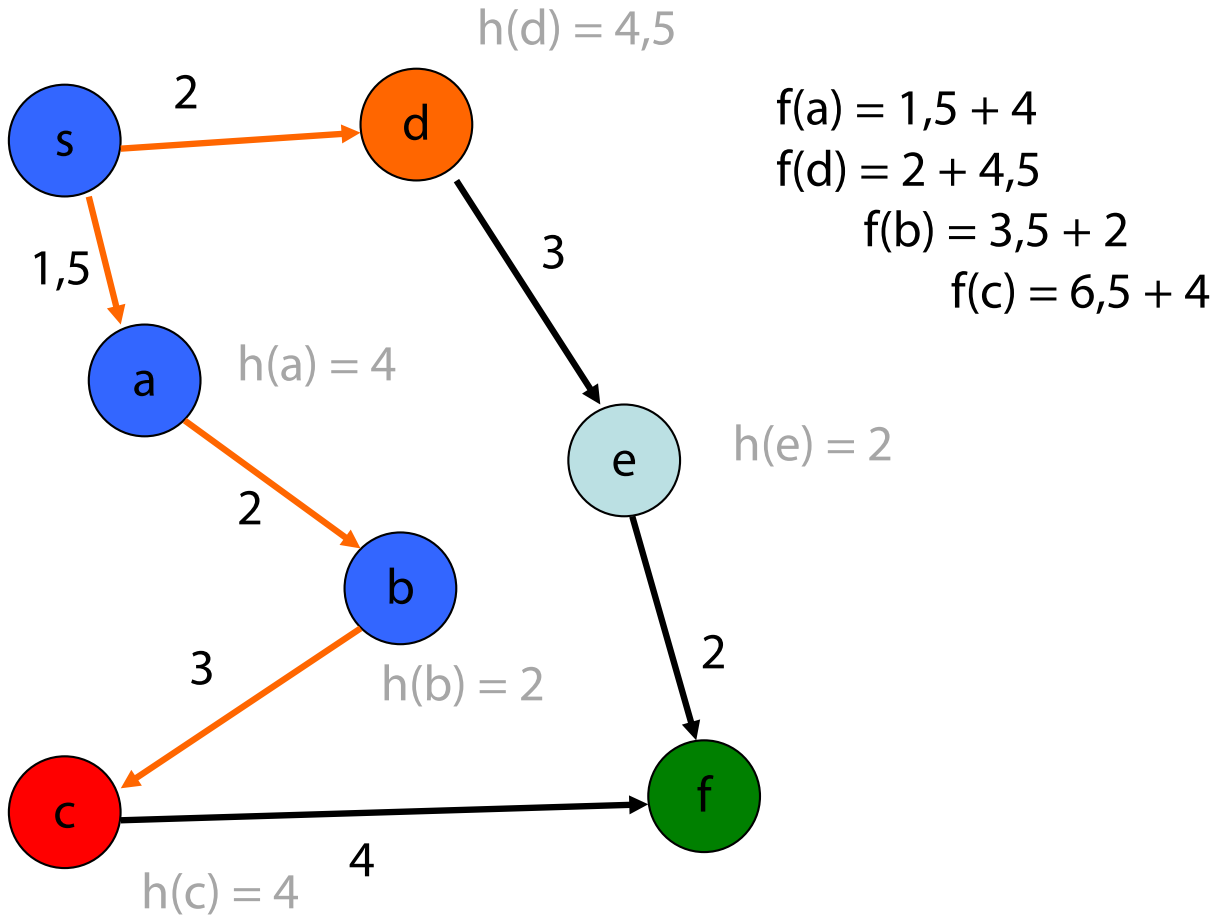
A* – Beispiel



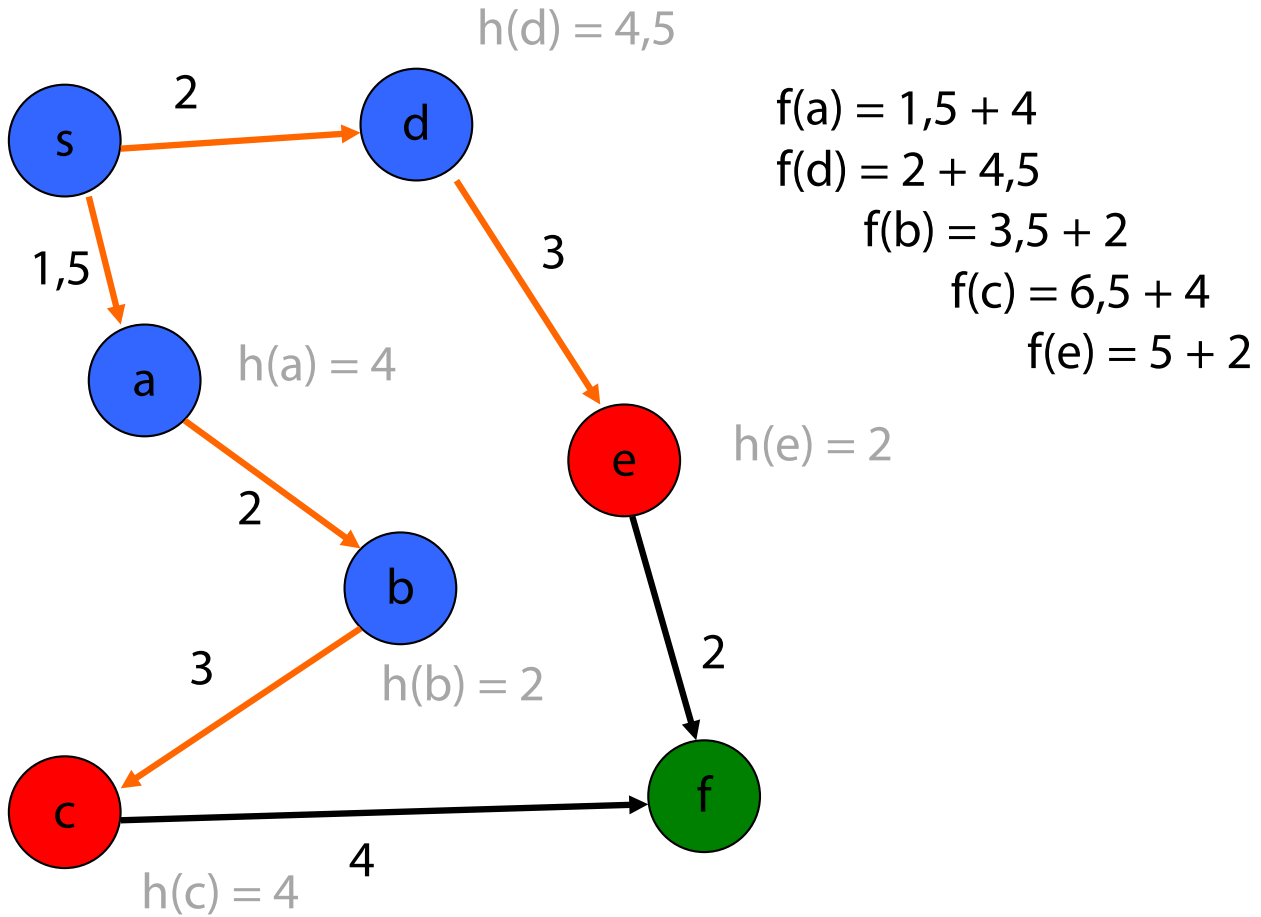
A* – Beispiel


h(d) = 4,5
2
s
d
1,5
3
a
h(a) = 4
e
h(e) = 2
2
b
3
h(b) = 2
2
c
f
4
h(c) = 4
f(a) = 1,5 + 4
f(d) = 2 + 4,5
f(b) = 3,5 + 2
f(c) = 6,5 + 4

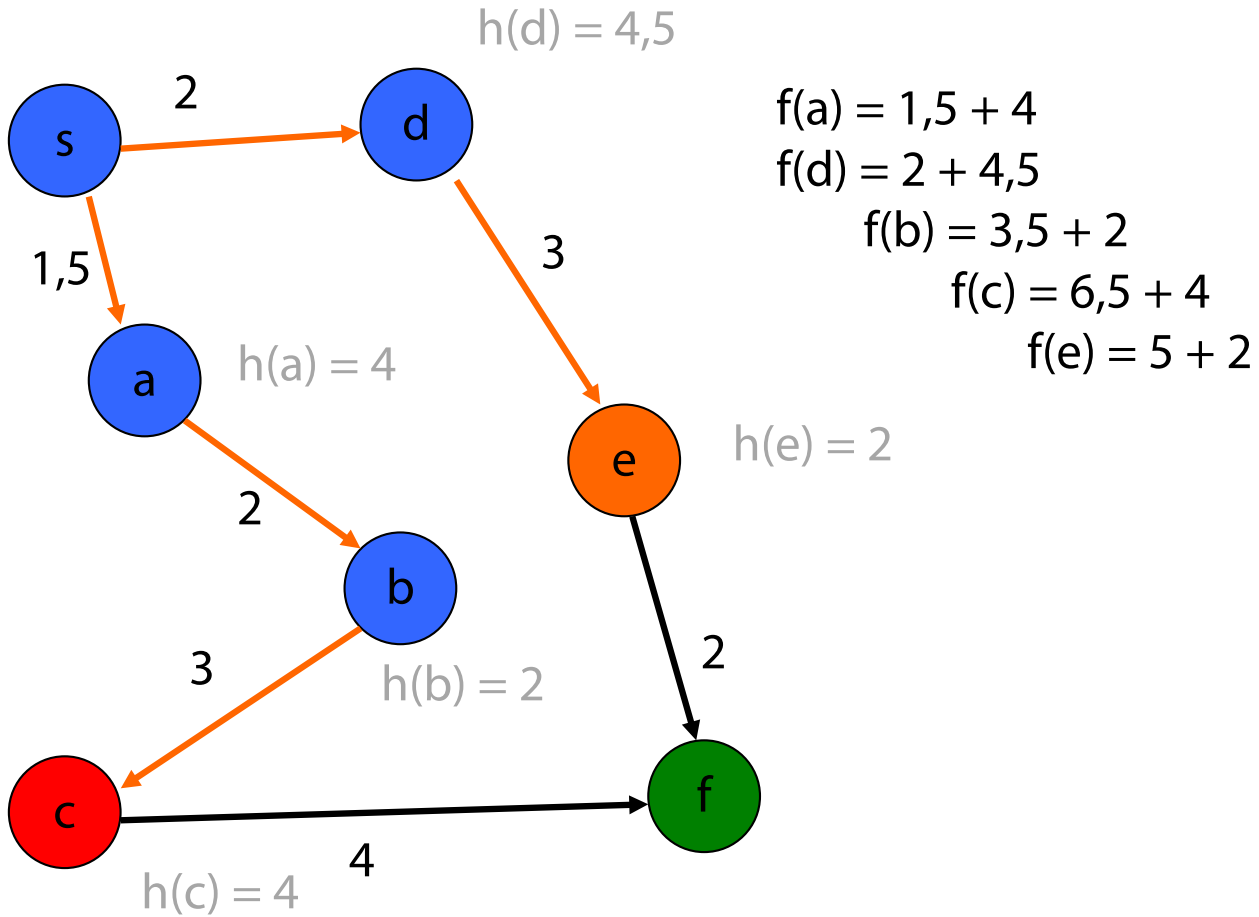
A* – Beispiel


h(d) = 4,5
2
s
d
1,5
3
a
h(a) = 4
e
h(e) = 2
2
b
3
h(b) = 2
2
c
f
4
h(c) = 4
f(a) = 1,5 + 4
f(d) = 2 + 4,5
f(b) = 3,5 + 2
f(c) = 6,5 + 4

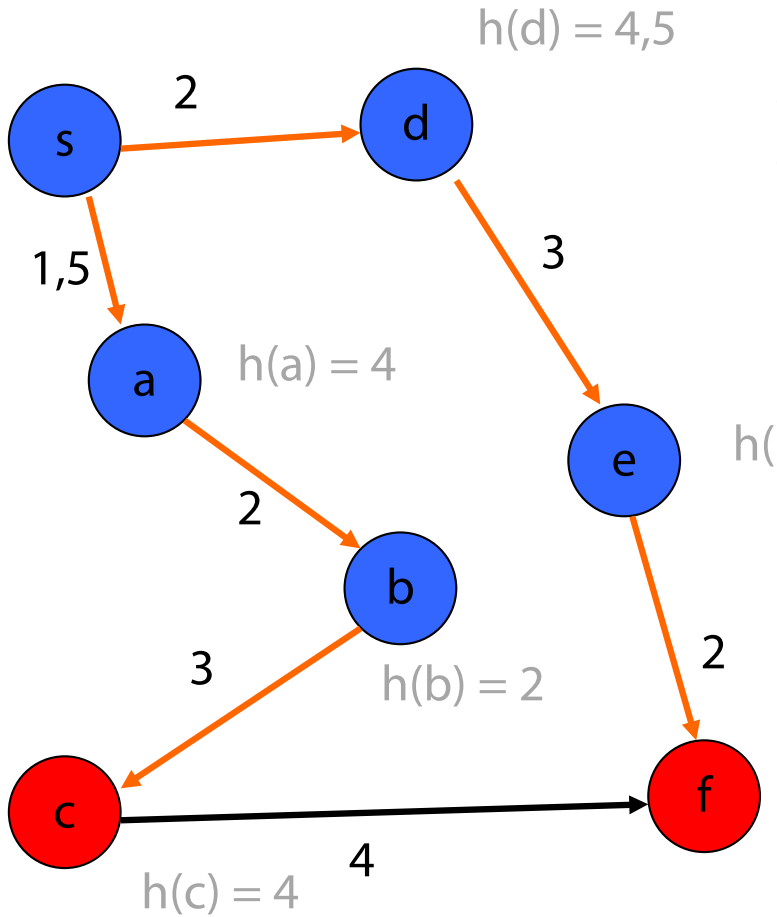
A* – Beispiel


h(d) = 4,5
2
s
d
1,5
3
a
h(a) = 4
e
h(e) = 2
2
b
3
h(b) = 2
2
c
f
4
h(c) = 4
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

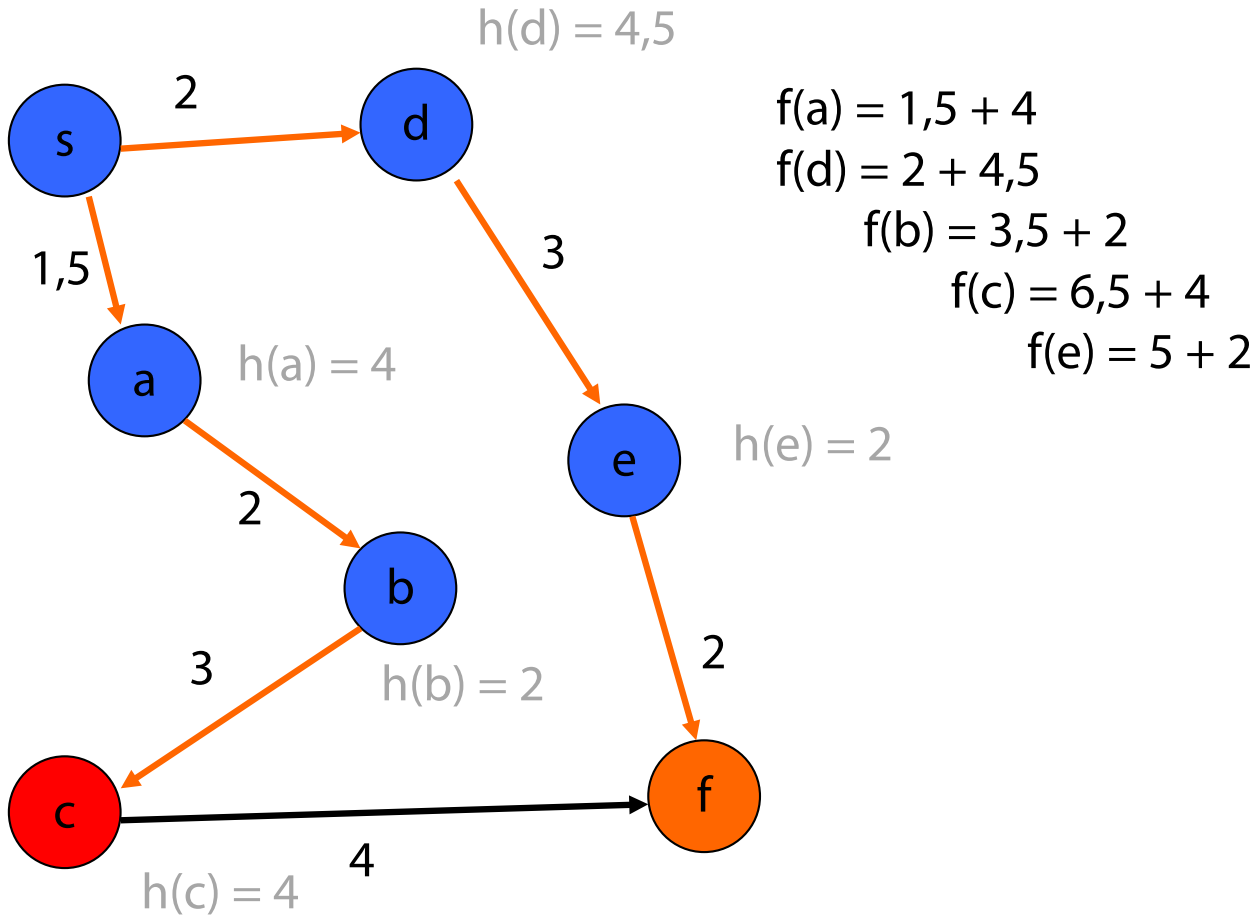

h(d) = 4,5
2
s
d
1,5
3
a
h(a) = 4
e
h(e) = 2
2
b
3
h(b) = 2
2
c
f
4
h(c) = 4
f(a) = 1,5 + 4
f(d) = 2 + 4,5
f(b) = 3,5 + 2
f(c) = 6,5 + 4
f(e) = 5 + 2

A* – Beispiel

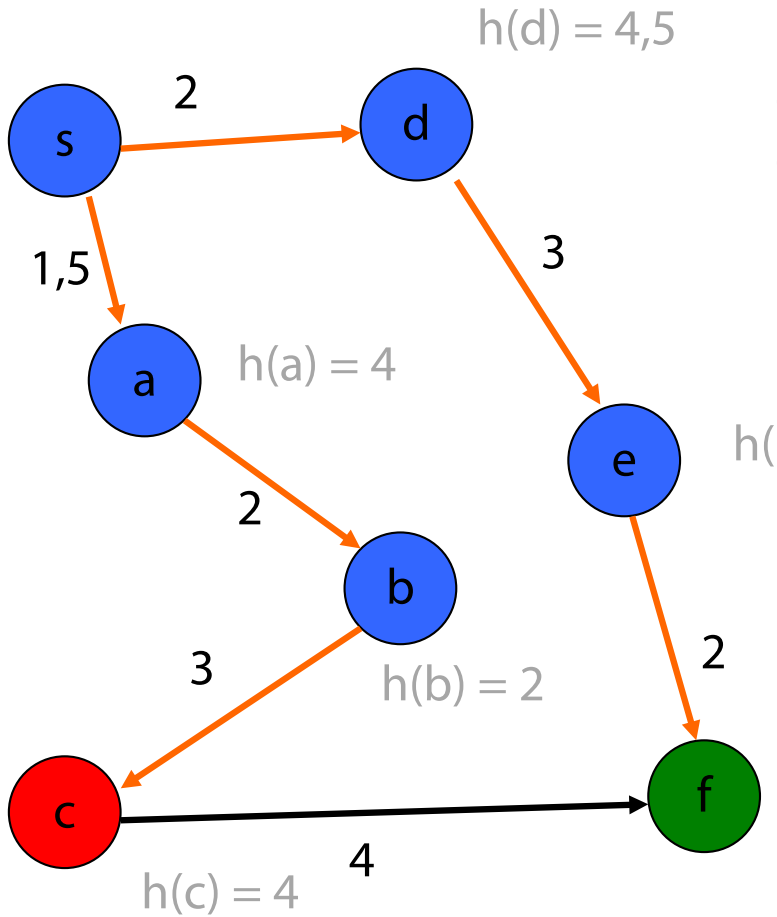


f(a) = 1,5 + 4
f(d) = 2 + 4,5
f(b) = 3,5 + 2
f(c) = 6,5 + 4
f(e) = 5 + 2

A* – Beispiel


h(d) = 4,5
2
s
d
1,5
3
a
h(a) = 4
e
h(e) = 2
2
b
3
h(b) = 2
2
c
f
4
h(c) = 4
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

Vereinbarung: Mehrfachwertzuweisung

Sei die Variable x an ein Tupel (1, 2, 3) gebunden, dann setzt der Ausdruck

$$(a, b, c) := x$$

die Variablen a, b, c entsprechend auf 1, 2, 3

Wenn ein Wert nicht interessant ist, schreiben wir z.B.:

$$(a, b, \_) := x$$

Dann wird nur a und b gesetzt (auf 1 bzw. 2).

Vereinbarung: Getter und Setter

Sei f eine Funktion, die aus Datum x etwas "extrahiert"

Dann verwenden wir

$$f(x) := c$$

um anzuzeigen, dass der nächste Aufruf
von f(x) den Wert c liefern soll.

Vereinbarung: Datentypen, Initwerte und Keys

Wir verwenden Datentypen

- Prioritätswarteschlange: PQ
- Menge: Set
- Kellerspeicher: Stack

Initialisierungschema:

var=init : Typ with key(x) = *Ausdruck*

Initialwert-Notation init:

- Prioritätswarteschlange oder Keller als Sequenz: <...>
- Menge als Menge: ∅ bzw. {...}

Funktion A*

```
Function A* (s, ziel?, cost, h, (V, E)):   // Eingabe: Start s ∈ V, Kantenkosten cost, Schätzer h und Graph (V, E)
  pq=<>:PQ with key((_,_, n_f,_)) = n_f;          // PQ mit Einträgen (Knoten, g-Kosten, f-Kosten, Vorgänger)
  expanded=∅:Set with key((v, _, _)) = v;  // Menge expandierter Knoten (Knoten, g-Kosten, Vorgänger)
  insert((s, 0, 0+h(s), ⊥), pq)
  while pq ≠ ∅ do
     (u, u_g , _, w) := deleteMin(pq)
     insert((u, u_g, w), expanded)
     if ziel?(u) then return path(u, expanded)  // Ausgabe: Lösungspfad rückwärts zum Start s
     foreach (u, v)∈ E do
        x := find(v, expanded)                  // Knoten schon gesehen und expandiert?
        if x = ⊥ then y := find(v, pq)          // Knoten schon gesehen aber noch nicht expandiert?
          v_g := u_g + cost(u, v)
          if  y = ⊥ then
              insert( (v, v_g, v_g + h(v), u) , pq)
          else (v, v_g-old, _, _) := y
               if v_g < v_g-old then              // Günstigerer Weg zu noch nicht expandiertem Knoten?
                  decreaseKey(y, pq, v_g-old - v_g)  // Expandiere früher und trage
                  parent(y) := u; g(y) := v_g        // neuen Vorgänger und neue g-Kosten ein
        else if u_g+ cost(u, v) < g(x) then       // Günstigerer Weg zu schon expandiertem Knoten
              propagate(u v, u_g+ cost(u, v), expanded, pq, (V, E))  // Propagiere neue Kosten für exp. Knoten
```

Hilfsfunktionen

```
Function path(v, expanded):
  // Konstruiere Pfad p aus Menge von expandierten Knoten
  // Einträge in expanded haben die Form (Knoten, g-Kosten, Vorgänger)
  p= <>:Stack (node)
  while true do
     x := find(v, expanded)
     if x = ⊥ then
        return p
     else (u, _, v) := x
           push(u, p)
```

Hilfsfunktionen

```
Function propagate(u, v, new-g, expanded, pq, (V, E)):
  // Propagieren neue Kosten g für Knoten v in die Nachfolger von v
   x := find(v, expanded)
   if x = ⊥ then
      x := find(v, pq)
      (v, old-g, _, _) := x                    // Noch nicht expandiert!
      if  new-g < old-g then                   // Besserer Weg und damit
         decreaseKey(v, pq, old-g – new-g) // neue Priorisierung
         g(x) := new-g; parent(x) := u         // neues g und neuer Vorgänger
  else (v, old-g, _) := x                      // Schon expandiert!
       if new-g < old-g then                   // Besserer Weg und damit
         g(x) := new-g; parent(x) := u         // neues g und neuer Vorgänger
         foreach (v, w) ∈ E do                 // weiter propagieren
             propagate(v, w, new-g + cost(v, w), expanded, pq, (V, E))
```

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 (also: $ziel?(v) = true$).

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

Beweis: (siehe Tafel)

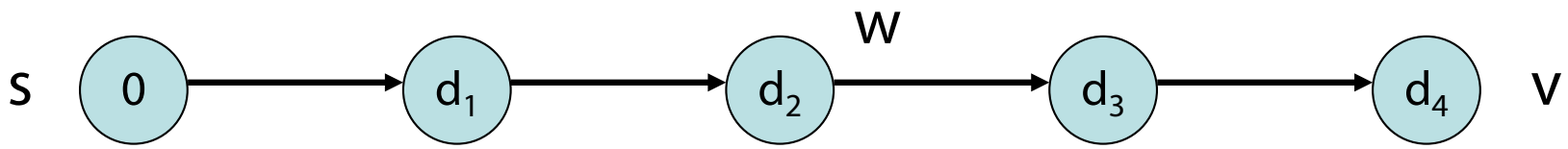
Analyse

- Im schlimmsten Fall wie Dijkstra-Algorithmus:
  - $h(n) = 0$ für alle Knoten n
  - Zielknoten hat höchste minimale Kosten
    (dann werden die minimalen Kosten g zu allen anderen Knoten ebenfalls ermittelt)
- 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
  - All-Pairs-Shortest-Paths (offline)
- Kantenkosten müssen positiv sein

Bellman-Ford Algorithmus

Nächster Schritt: Kürzeste Wege für beliebige Graphen mit beliebigen Kantenkosten (aber noch SSSP).

Problem: besuche Knoten eines kürzesten Weges in richtiger Reihenfolge

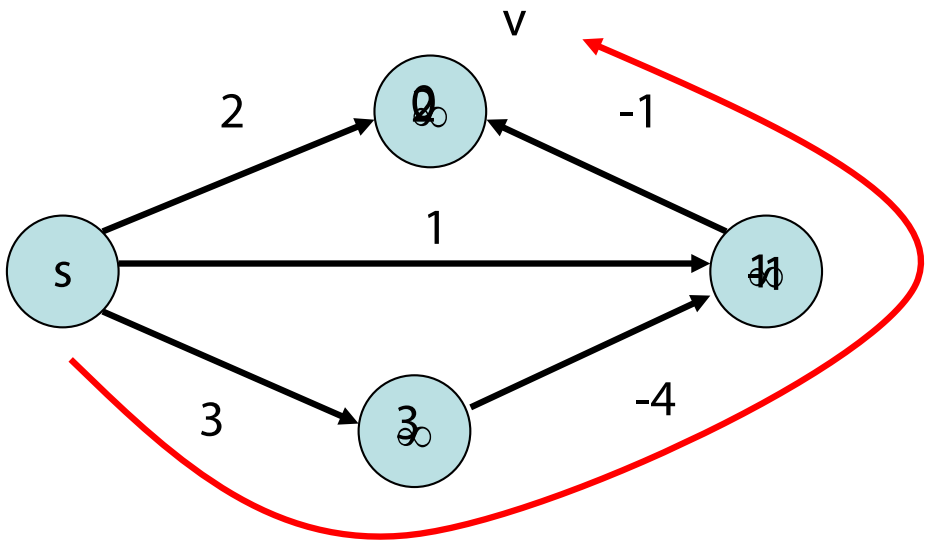


- Dijkstra Algo kann nicht verwendet werden, da im Allgemeinen nicht mehr die Knoten in der Reihenfolge ihrer Distanz zu s besucht werden

R. E. Bellman: On a Routing Problem. In: Quarterly of Applied Mathematics. 16(1). Brown University, S. 87-90, **1958**

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:

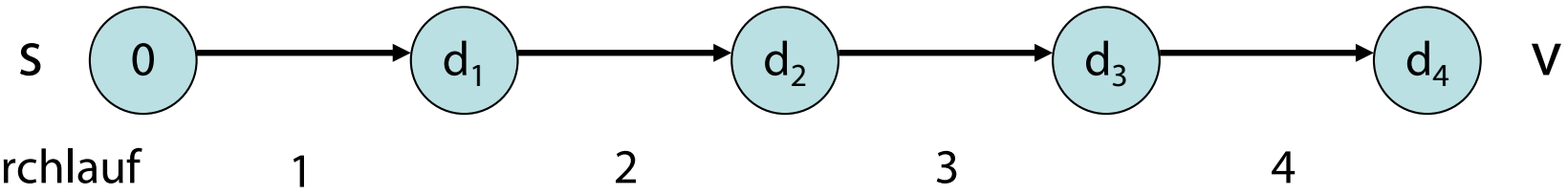
- Weg mit Kreis mit Kantenkosten ≥ 0: Kreisentfernung erhöht nicht die Kosten
- Weg mit Kreis mit Kantenkosten < 0: Distanz zu s ist $-\infty$!

Bellman-Ford Algorithmus

Folgerung: (für Graph mit n Knoten)

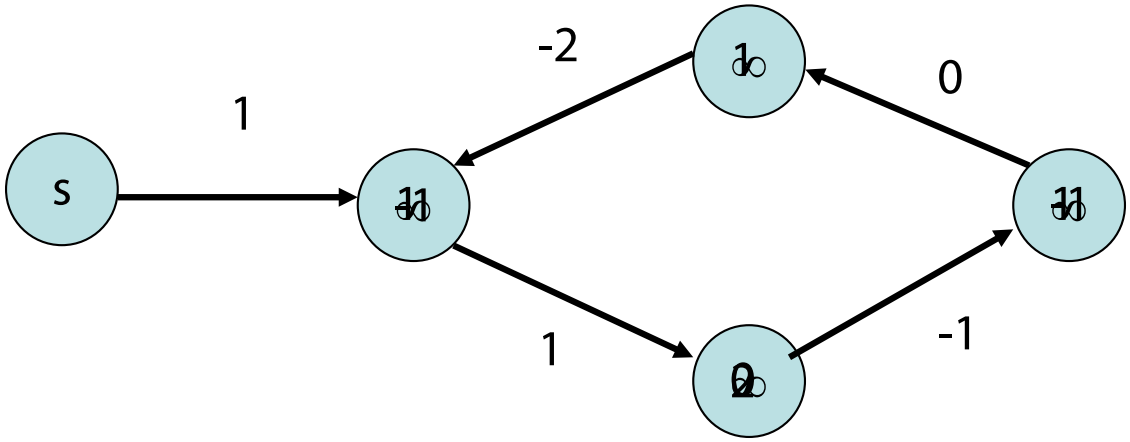
Für jeden Knoten v mit $\mu(s,v) > -\infty$ gibt es kürzesten Weg der Länge (Anzahl Kanten!) <n zu v

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



Bellman-Ford Algorithmus

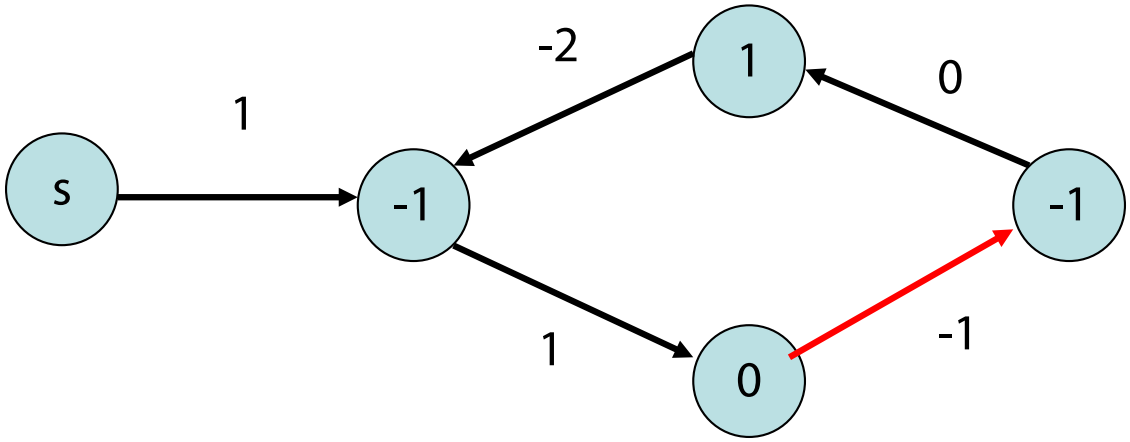
Problem: Erkennung negativer Kreise



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

Bellman-Ford Algorithmus

Problem: Erkennung negativer Kreise



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

Bellman-Ford Algorithmus

Zusammenfassung:

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

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

Bellman-Ford Algorithmus

```
Procedure BellmanFord(s: NodeId)
    d=<∞,…,∞>: Array of ℝ ∪ {-∞,∞}
    parent=<⊥,…,⊥>: Array of NodeId
    d[s]:=0; parent[s]:=s
    for i:=1 to n-1 do  // aktualisiere Kosten für n-1 Runden
       forall e=(v,w) ∈ E do
          if d[w] > d[v]+c(e) then  // bessere Distanz möglich?
             d[w]:=d[v]+c(e); parent[w]:=v
    forall e=(v,w) ∈ E do       // in n-ter Runde noch besser?
       if d[w] > d[v]+c(e) then infect(w)


Procedure infect(v)  // propagiere -∞-Kosten von v aus
    if d[v]>-∞ then     // noch nicht markiert?
       d[v]:=-∞
       forall (v,w) ∈ E do infect(w)
```

Bellman-Ford Algorithmus

Laufzeit: $O(n \cdot m)$

Verbesserungsmöglichkeiten:

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

All Pairs Shortest Paths

Annahme: Graph mit beliebigen Kantenkosten, aber keine negativen Kreise

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

Laufzeit: $O(n^2 \cdot m)$

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

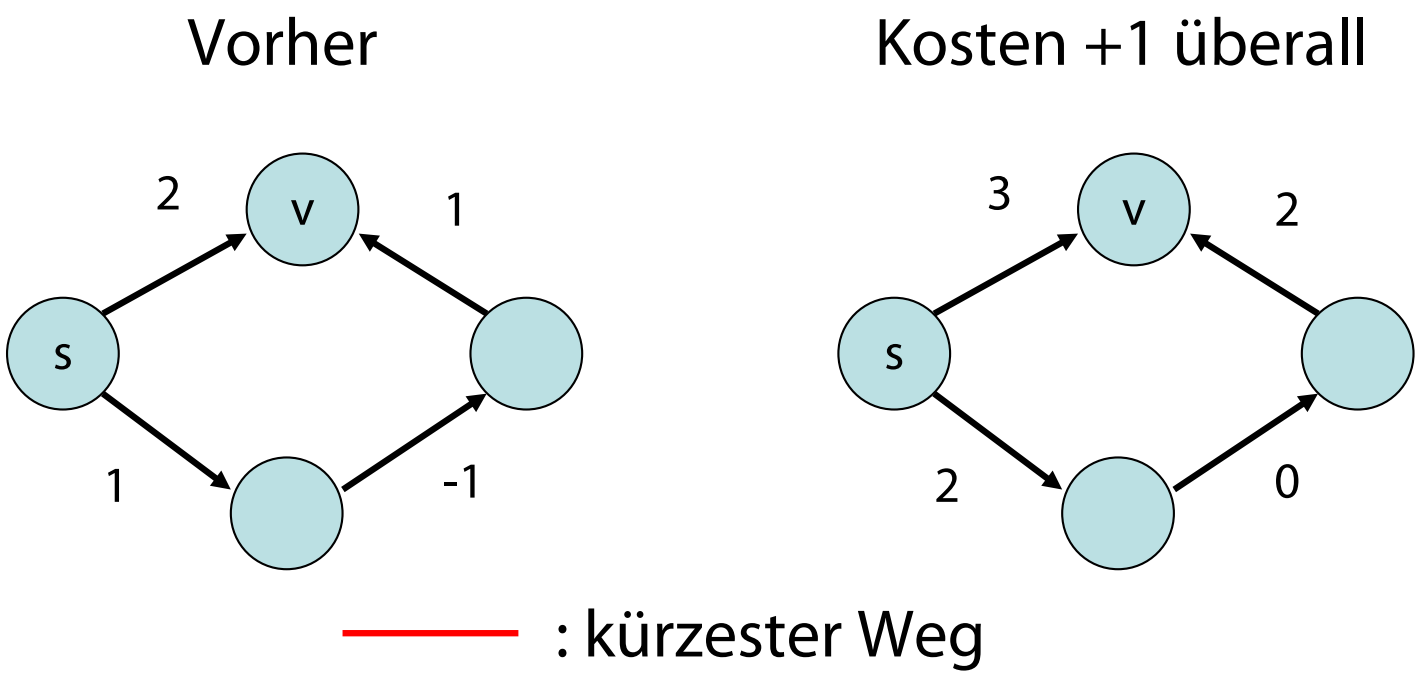
Lösung: Umwandlungsstrategie in nichtnegative Kantenkosten, ohne kürzeste Wege zu verfälschen (nicht so einfach!)

Dijkstra erfordert, dass jeder Knoten über kürzesten Weg erreichbar

Johnson, Donald B., Efficient algorithms for shortest paths in sparse networks, Journal of the ACM 24 (1): pp. 1–13, **1977**

Naive Erhöhung?

- Naive Umwandlung negativer Kantenkosten durch Addition von $c := - \min \{ 0 \} \cup \{ c(e) \mid e \in E, c(e) < 0 \}$ schlägt i.a. fehl
- Gegenbeispiel zur Erhöhung um Wert c:



Neuer Ansatz

- Sei ϕ:V$\longrightarrow \mathbb{R}$ eine Funktion, die jedem Knoten ein Potenzial zuweist.
- Die reduzierten Kosten von e=(v,w) sind:

  $r(e) := \phi(v) + c(e) - \phi(w)$
- 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 ϕ : 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 ϕ : $r(p)<r(q)$ genau dann wenn $c(p)<c(q)$.

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

$$r(p) = \Sigma_i\, r(e_i)$$
$$= \Sigma_i\, (\phi(v_i) + c(e_i) - \phi(v_{i+1}))$$
$$= \phi(v_1) + c(p) - \phi(v_k)$$

All Pairs Shortest Paths

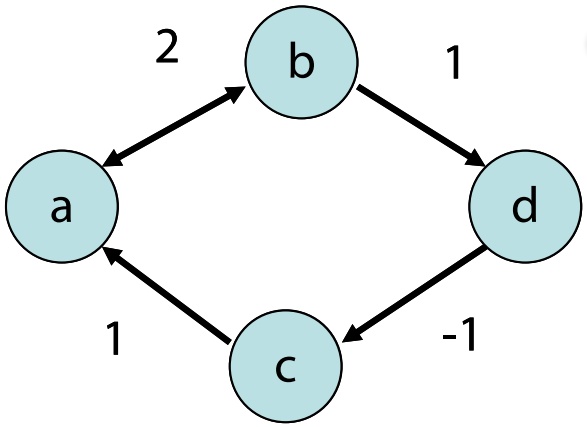
Beh: Angenommen, G habe keine negativen Kreise und alle Knoten können von s erreicht werden. Sei $\phi(v)=\mu(s,v)$ für alle $v \in V$.
Mit diesem ϕ ist $r(e)=\phi(v) + c(e) - \phi(w) \geq 0$
für alle $e = (v, w) \in E$.

Beweis:

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

All Pairs Shortest Paths

Beispiel:



Umwandlung der Kantenkosten in positive Werte, so dass kürzeste Wege erhalten bleiben

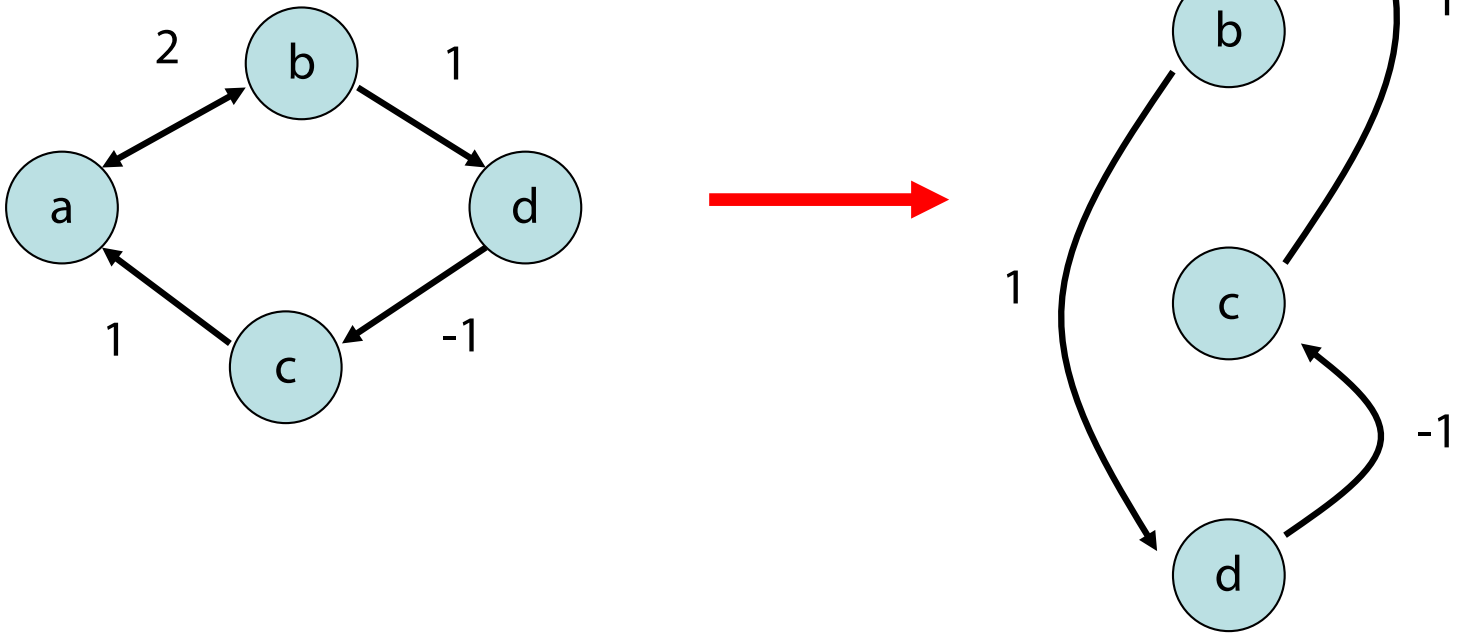
Nehme an, kürzester Weg von Zentralkonten zu Start- und Endknoten einer Kante sind bestimmt

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

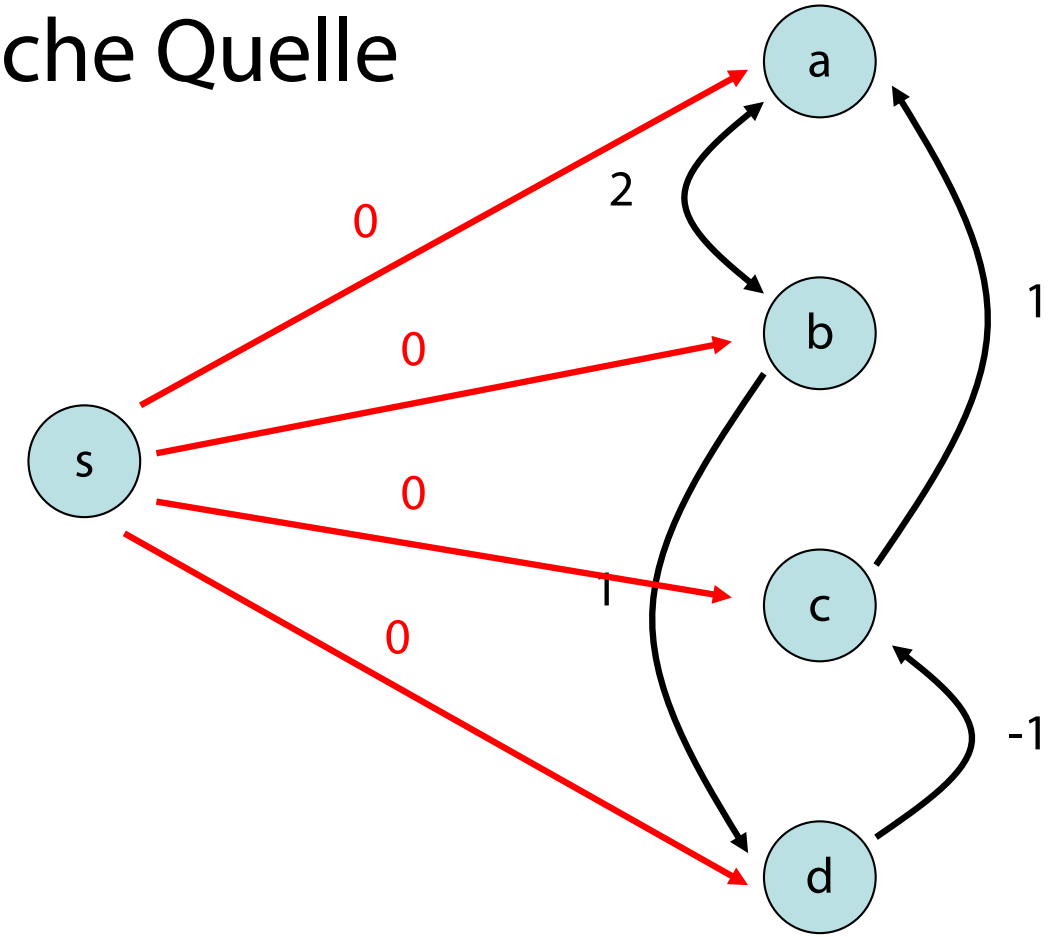
All Pairs Shortest Paths

Beispiel:



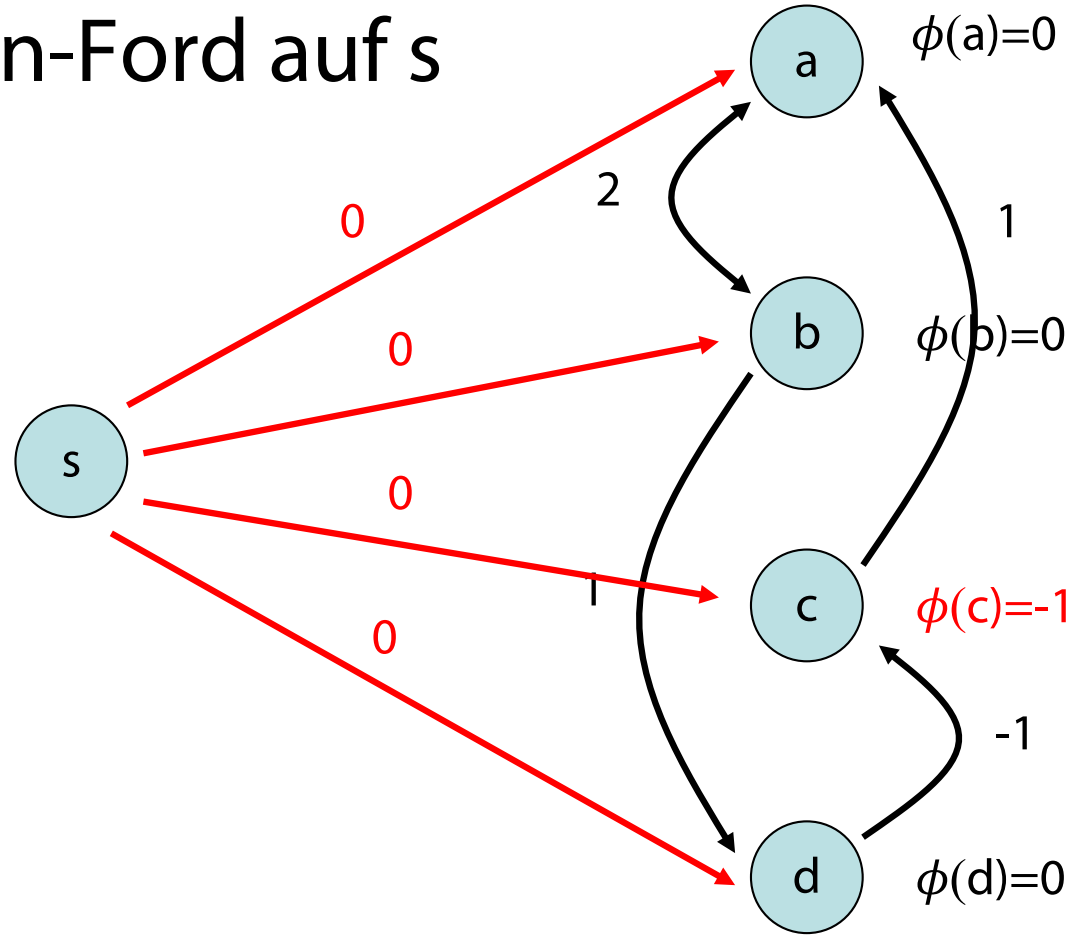
All Pairs Shortest Paths

Schritt 1: Künstliche Quelle



All Pairs Shortest Paths

Schritt 2: Bellman-Ford auf s

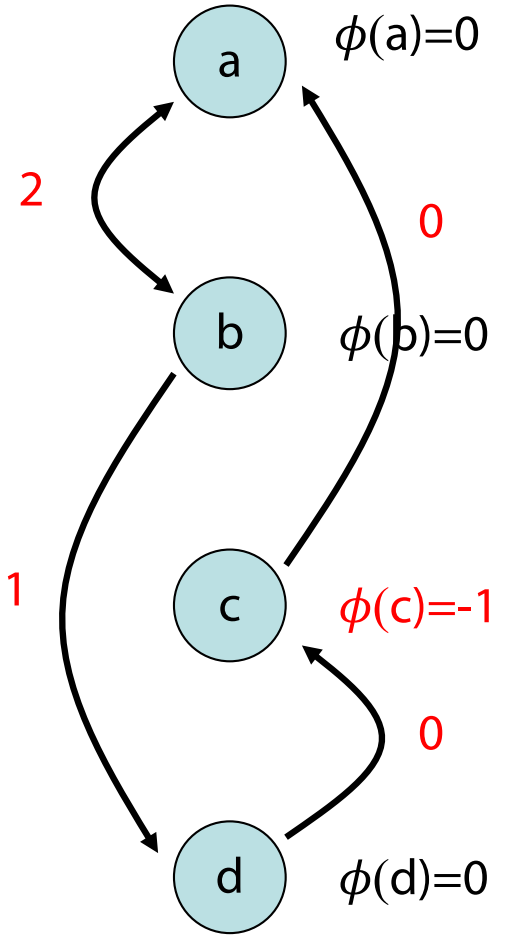


All Pairs Shortest Paths

Schritt 3: r(e)-Werte berechnen

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

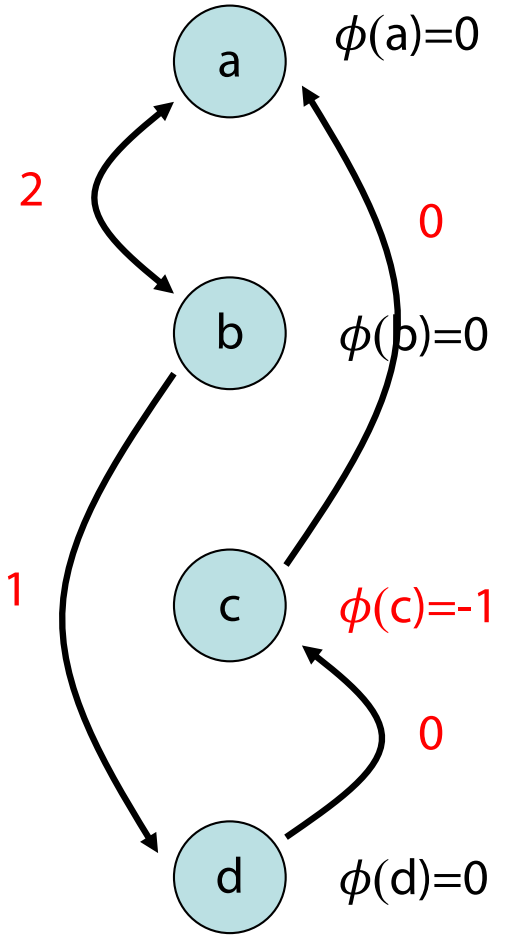
$$r(e) := \phi(v) + c(e) - \phi(w)$$



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

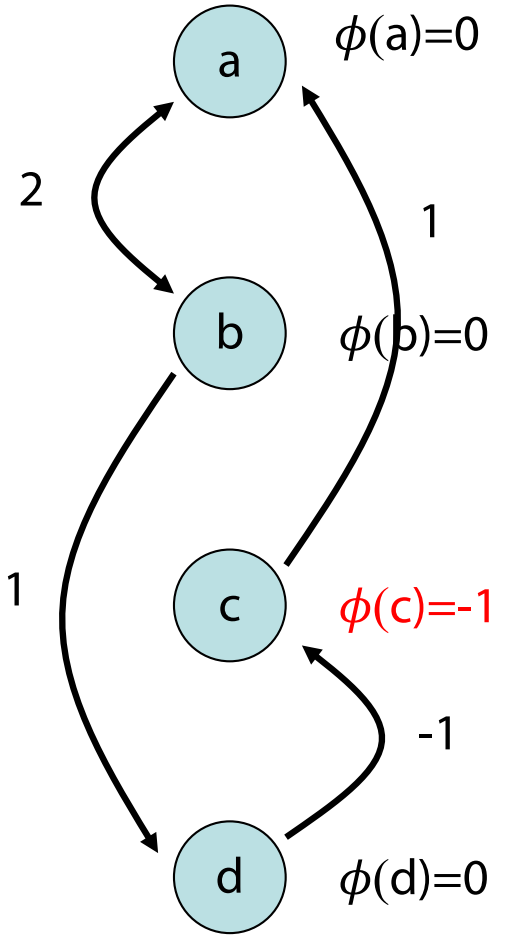


All Pairs Shortest Paths

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):

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

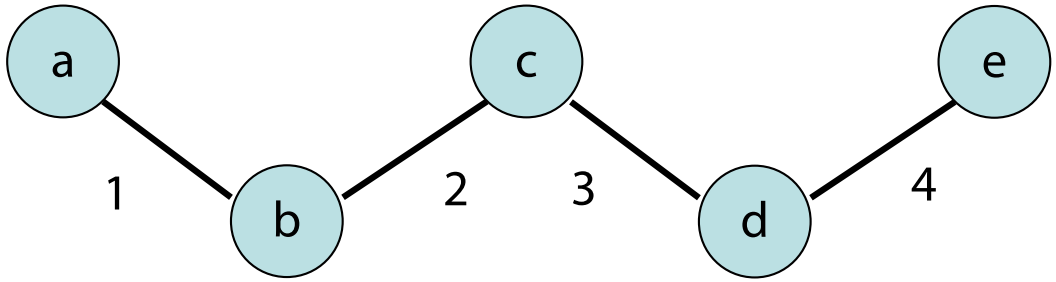
unter Verwendung von Fibonacci Heaps.

Da i.a. $m>n$, ist das sicher besser als

n mal Bellman-Ford $\in O(n^2 \cdot m)$

Annahme: Ungerichteter Graph gegeben

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




```
(1) Für alle i,j   : d[i,j] = w[i,j]
(2) Für k = 1 bis n
(3)    Für alle Paare i,j
(4)       d[i,j] = min (d[i,j],d[i,k] + d[k,j])
```

Annahme: Ungerichteter Graph gegeben

```
(1) Für alle i,j  : d[i,j] = w[i,j]
(2) Für k = 1 bis n
(3)   Für alle Paare i,j
(4)     d[i,j] = min (d[i,j],d[i,k] + d[k,j])
```

Analyse: $O(n^3)$

- Wenn wir annehmen, dass m>n, ist das immer noch besser als n mal 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 für DAGs

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

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

Naives Verfahren für Erreichbarkeit

Algorithmus von Warshall [Wikipedia]

```
(1) Für k = 1 bis n
(2)   Für i = 1 bis n
(3)     Falls d[i,k] = 1
(4)       Für j = 1 bis n
(5)         Falls d[k,j] = 1
(6)           d[i,j] := 1
```

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

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

Stephen Warshall: A Theorem on Boolean Matrices.
In: Journal of the ACM 9, 1, **1962**

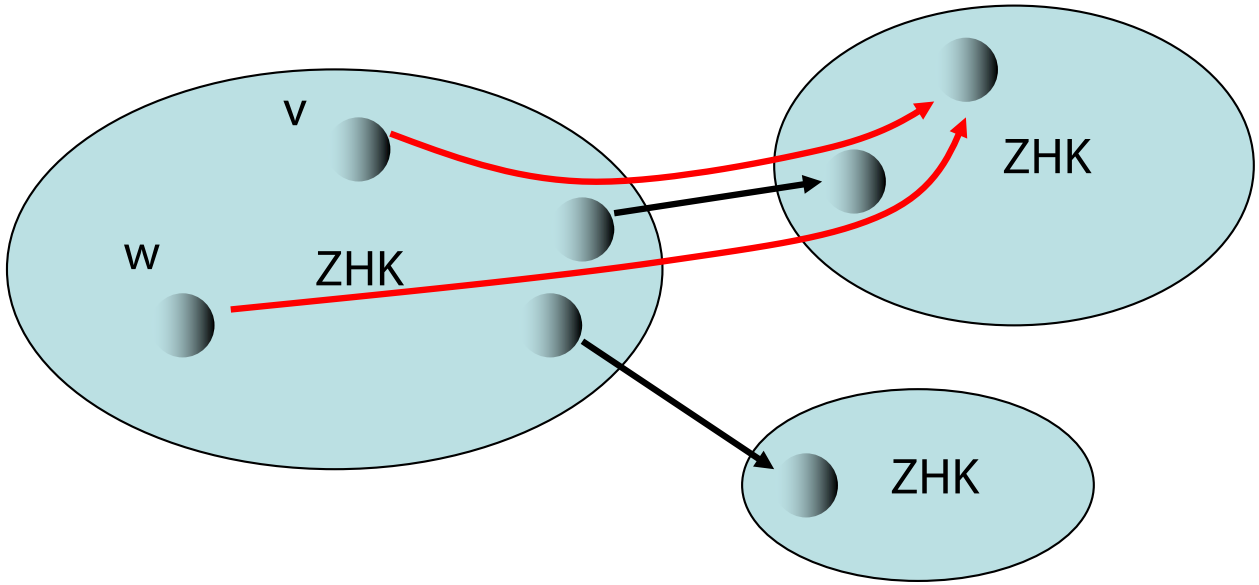
Transitive Hülle

Lösung 1: verwende APSP Algorithmus

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

Transitive Hülle

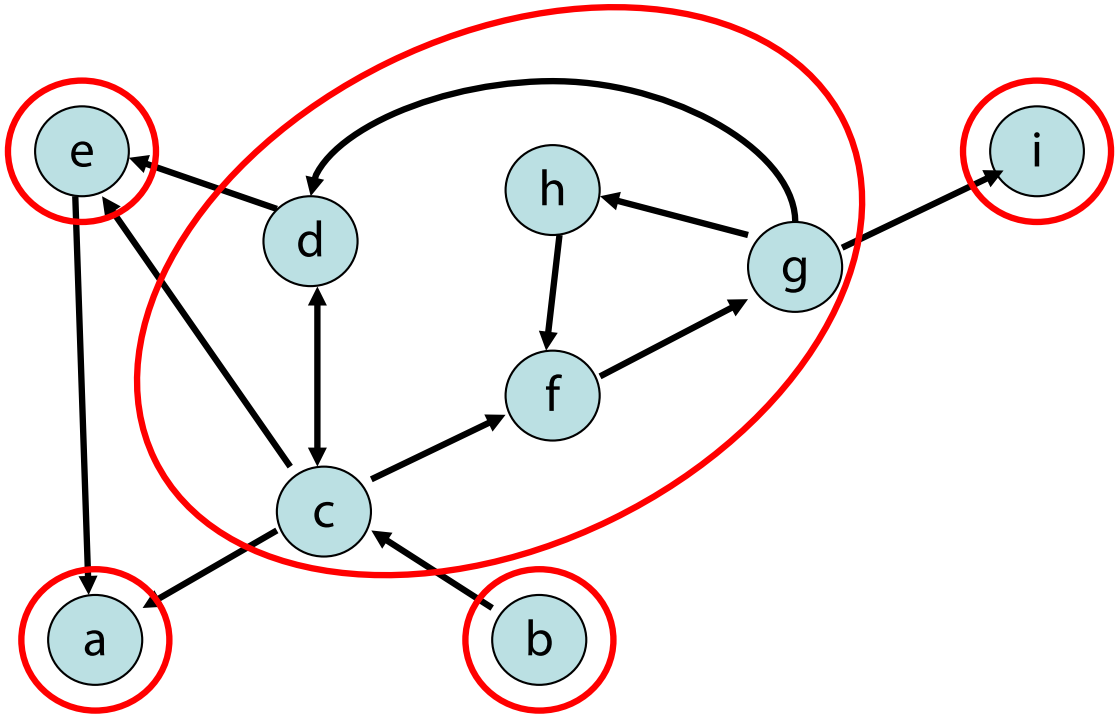
Einsicht: Alle Knoten in einer starken ZHK haben dieselbe Menge erreichbarer Knoten. Daher reicht es, sie durch Repräsentanten zu vertreten.



Transitive Hülle

Lösung 2: verwende ZHK-Algorithmus

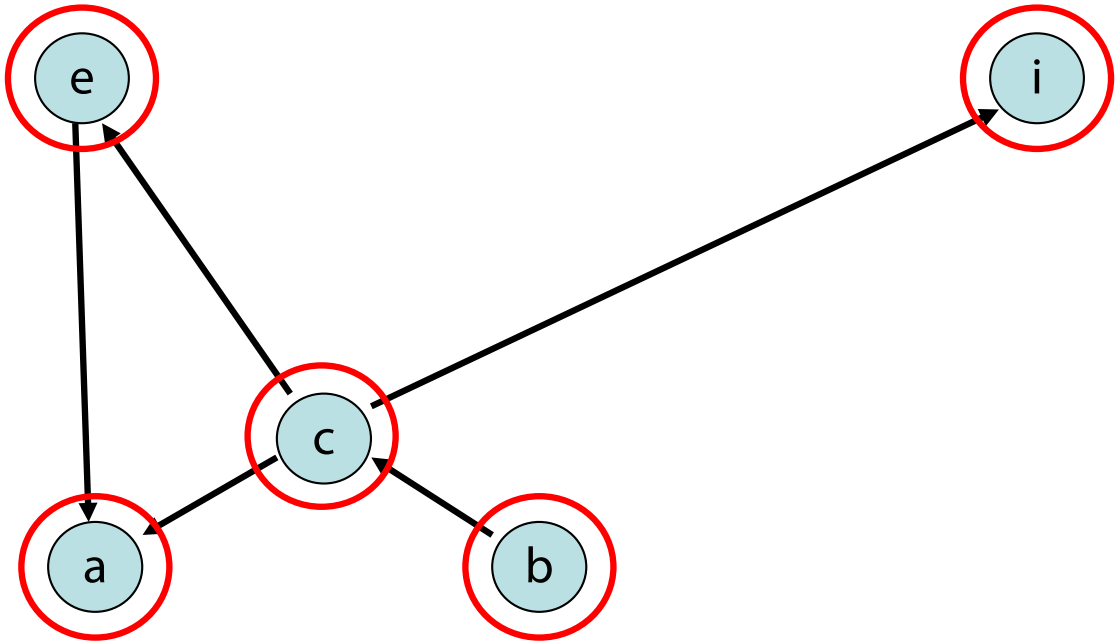
- Bestimme starke ZHKs



Transitive Hülle

Lösung 2: verwende ZHK-Algorithmus

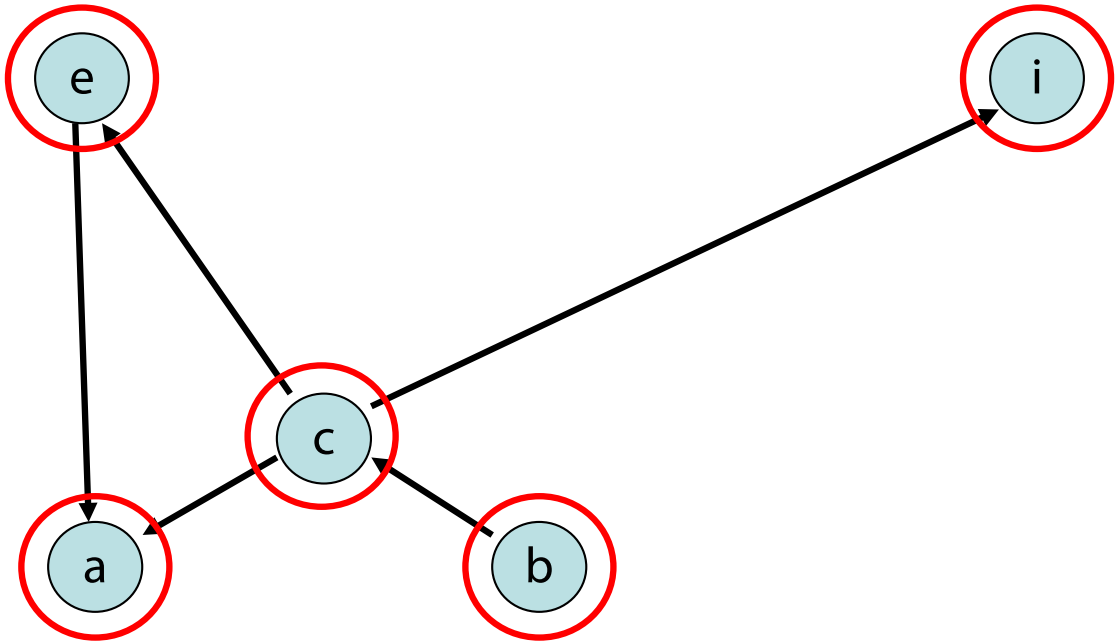
- Bestimme ZHK-Graph (Repräsentanten)



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 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

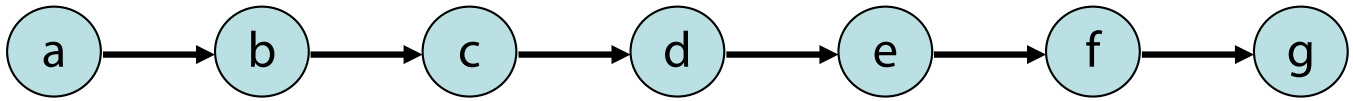
Ist es auch möglich, mit ~ O(n+m) Speicher für die Datenstruktur die Operation Reachable(v,w) effizient abzuarbeiten?

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

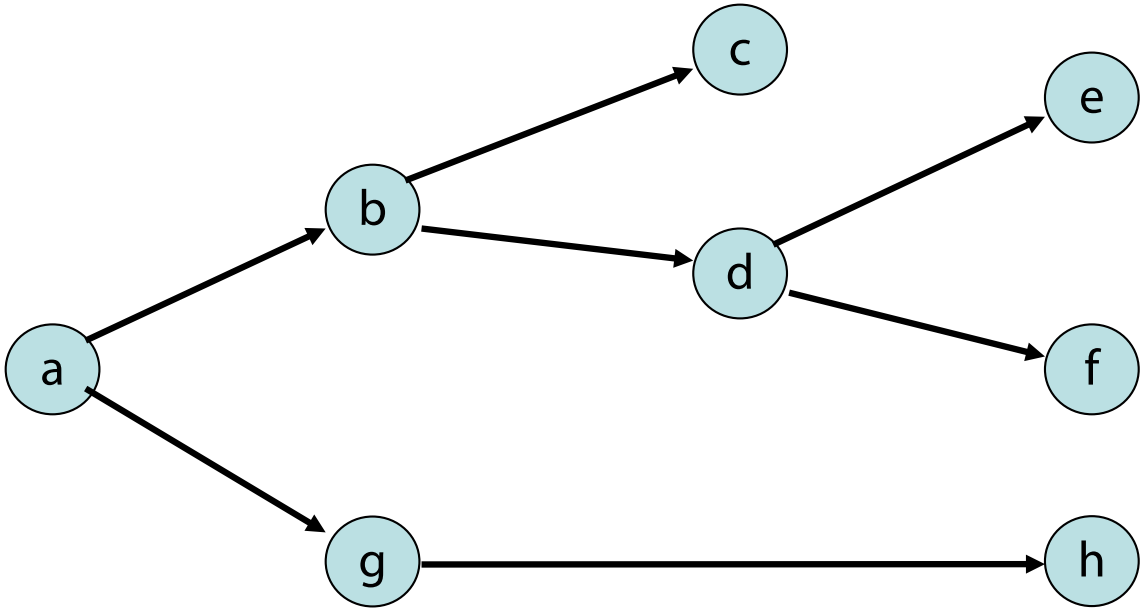
Fall 1: Der ZHK-Graph ist eine gerichtete Liste

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

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

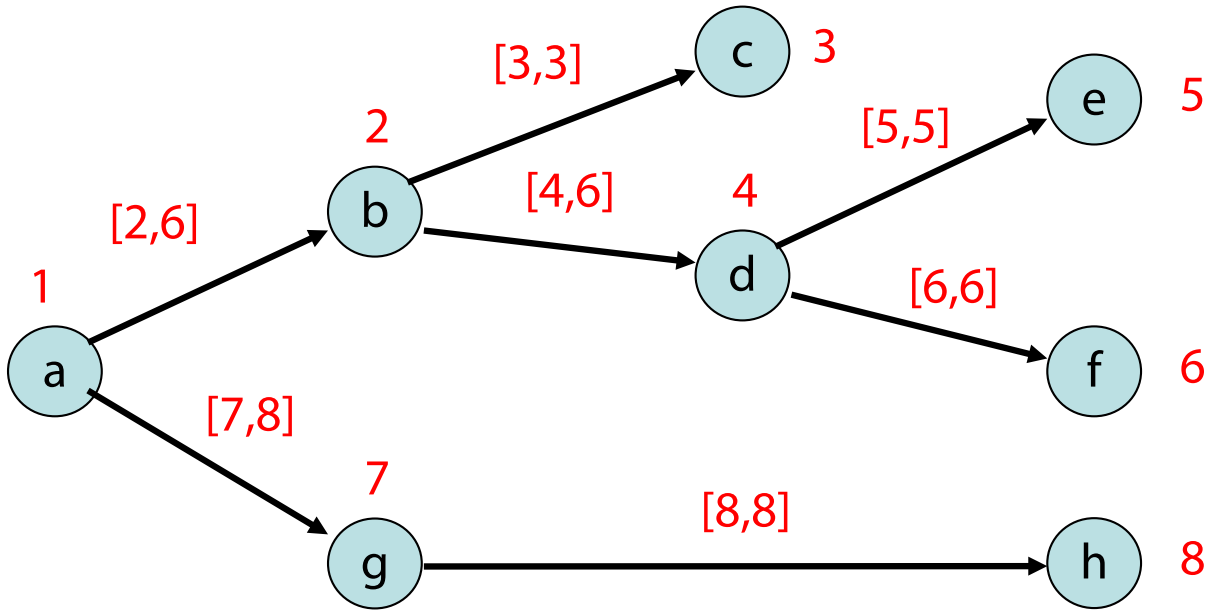
Transitive Hülle

Fall 2: Der ZHK-Graph ist ein gerichteter Baum



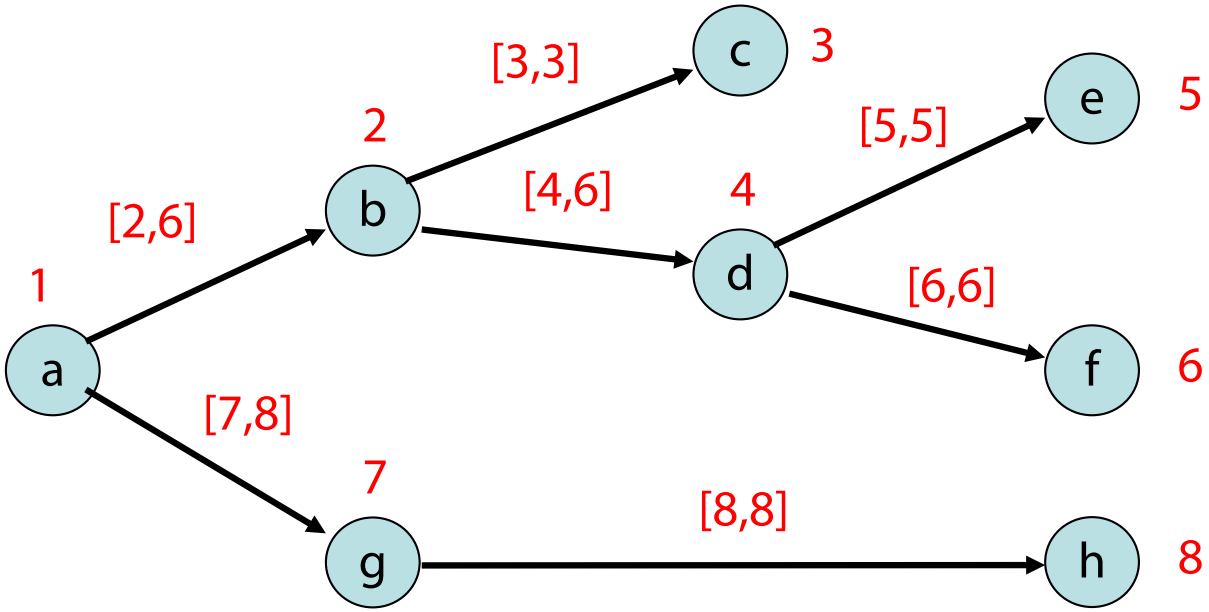
Transitive Hülle

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



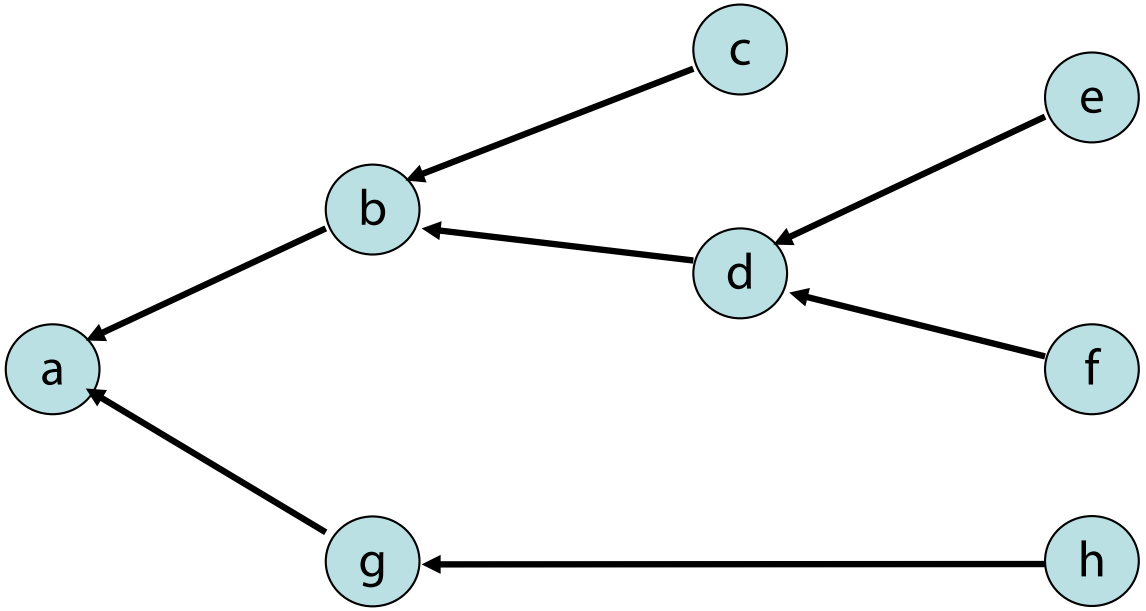
Transitive Hülle

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:



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

Transitive Hülle

Fall 2: Der ZHK-Graph ist ein gerichteter Baum

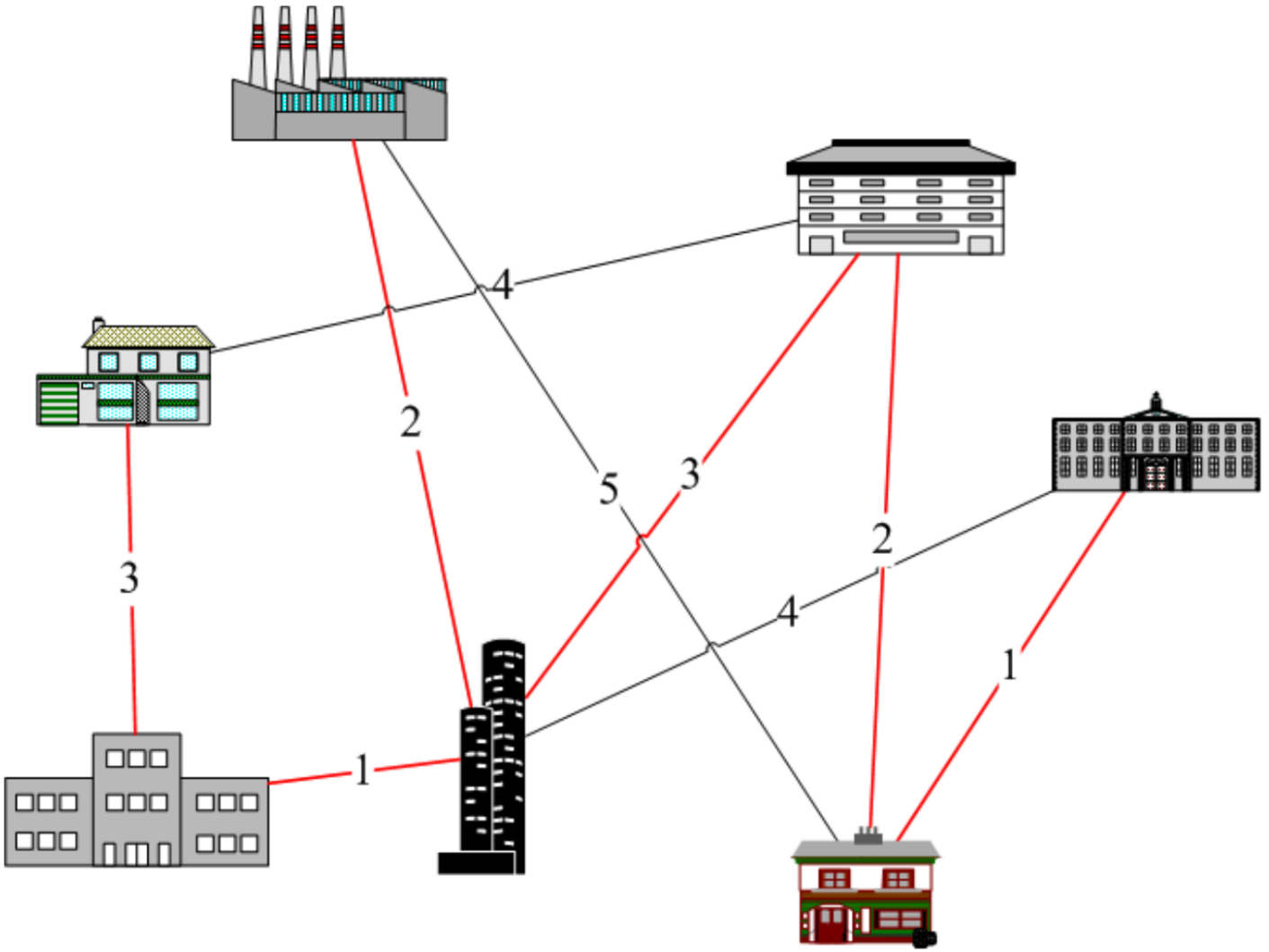
Datenstruktur: O(n+m) Zeit und Speicher
(speichere Repräsentanten zu jedem Knoten
Kantenintervalle zu jedem Repräsentanten)

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

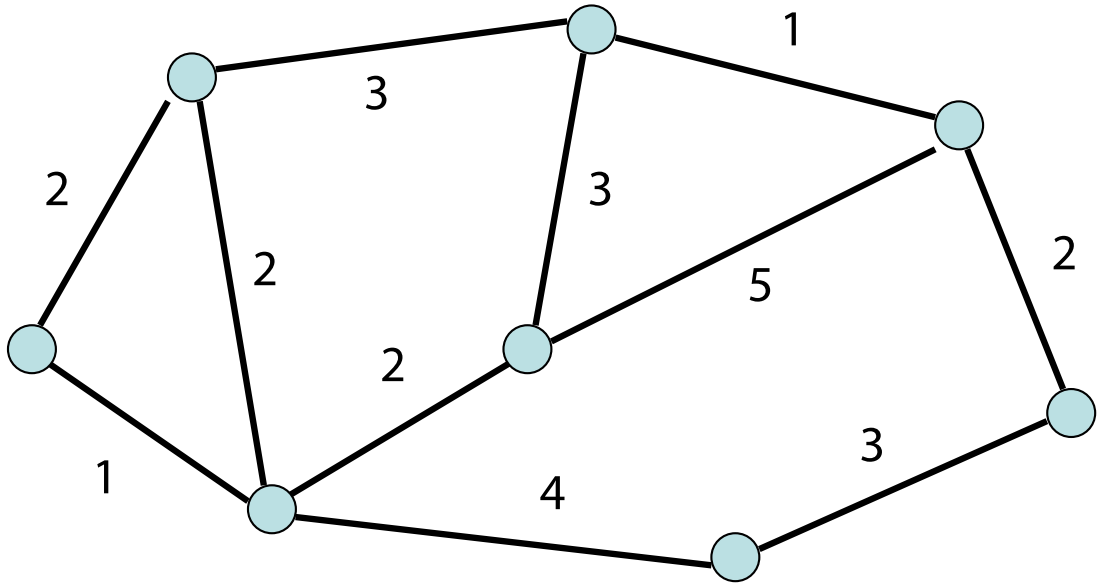
Geht auch noch (in O(d log n), hier nicht vertieft)

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 $c : E \longrightarrow \mathbb{R}_+$

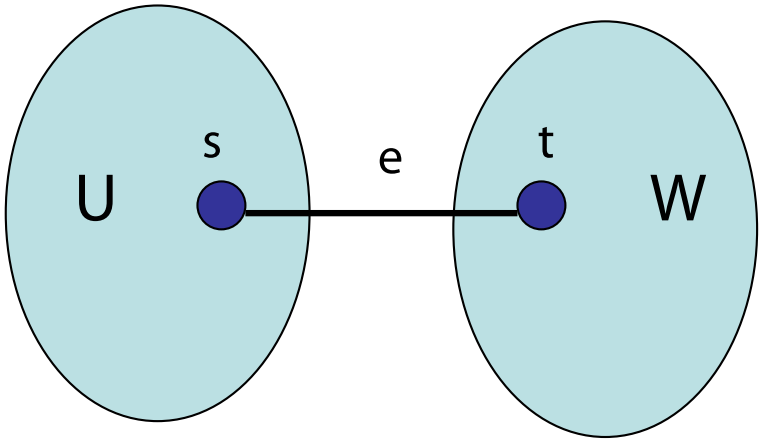
Ausgabe: $\text{argmin}_{T \subseteq E \wedge (V,T) \text{ verbunden}} \Sigma_{e \in T} c(e)$

- Teilmenge $T \subseteq E$, so dass Graph (V,T) verbunden und $c(T)=\Sigma_{e \in T} c(e)$ minimal

- T formt immer einen Baum (wenn c positiv).
- Baum über alle Knoten in V mit minimalen Kosten: minimaler Spannbaum (MSB)

Minimaler Spannbaum

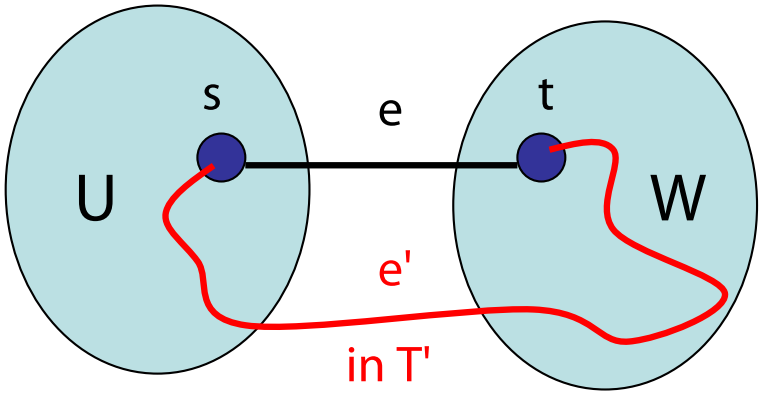
Beh 1: Sei (U,W) eine Partition von V (d.h. $U \cup W = V$ und $U \cap W = \varnothing$) und e={s,t} eine Kante mit minimalen Kosten mit $s \in U$ und $t \in W$. Dann gibt es einen minimalen Spannbaum (MSB) T, der e enthält.



Minimaler Spannbaum

Beweis von Beh 1:

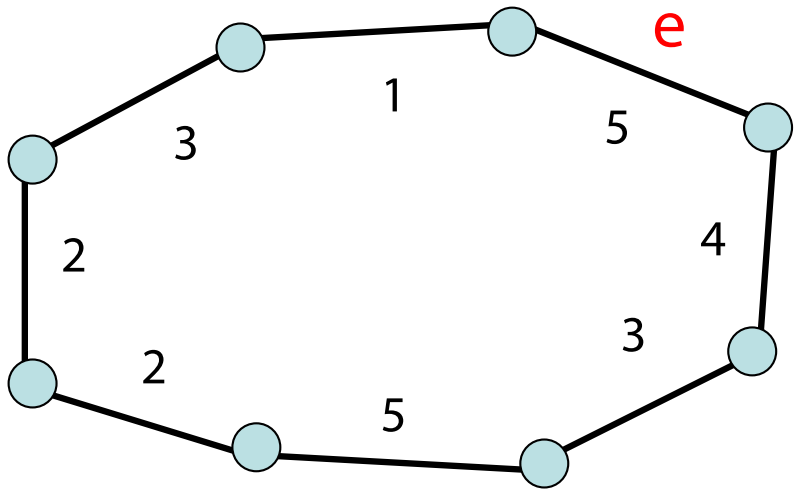
- Betrachte beliebigen MSB T'
- e={s,t}: (U,W)-Kante minimaler Kosten



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

Minimaler Spannbaum

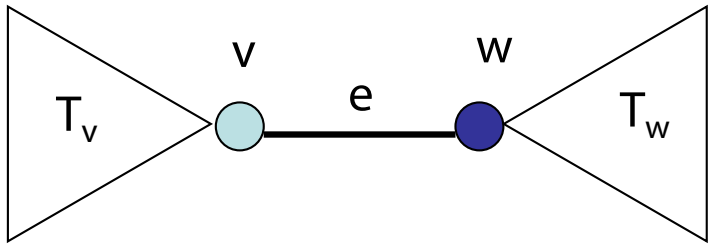
Beh 2: Betrachte beliebigen Kreis C in G und sei e Kante in C mit maximalen Kosten. Dann ist jeder MSB in G ohne e auch ein MSB in G.



Minimaler Spannbaum

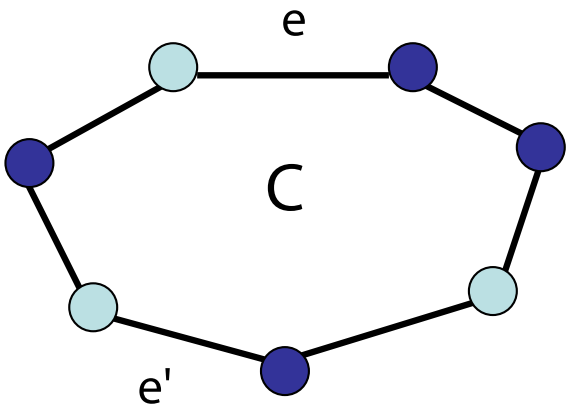
Beweis von Beh 2:

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



e maximal für C

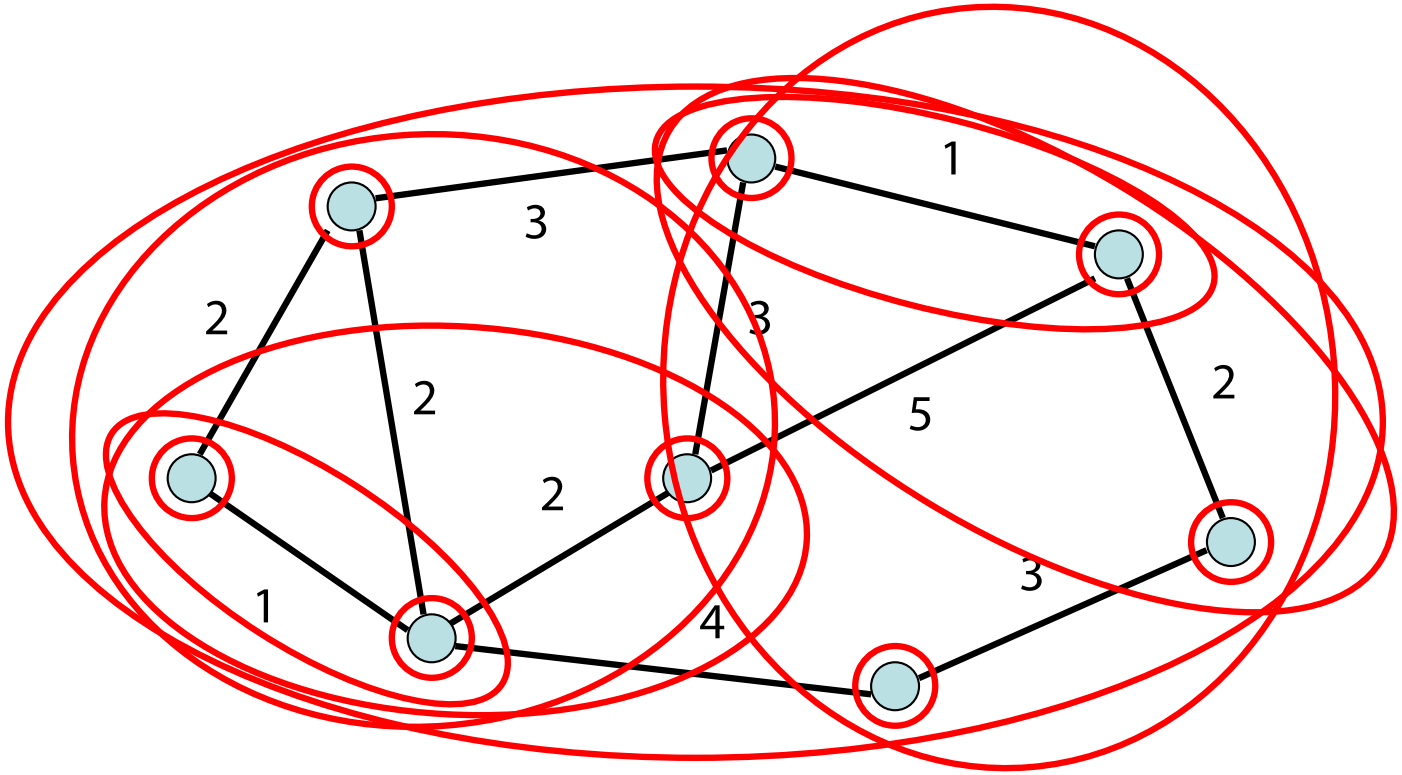
- ○ : zu T_v, ● : zu T_w
  - es gibt e' von T_v nach T_w
  - e ⟶ e' ergibt MSB T' ohne e



Minimaler Spannbaum

Regel aus Beh 1:

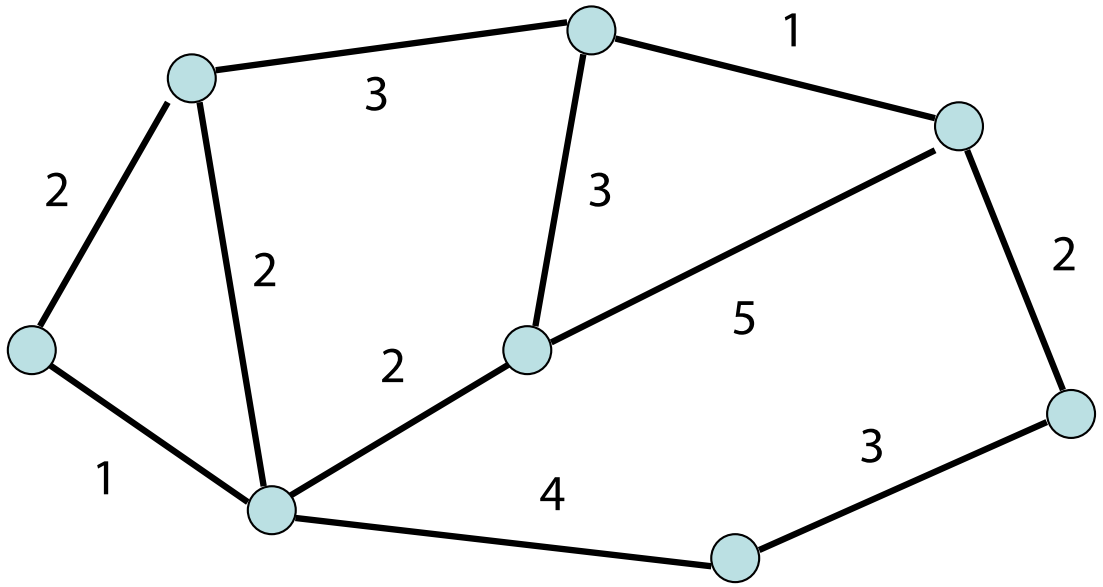
Wähle wiederholt Kante mit minimalen Kosten, die verschiedene ZHKs verbindet, bis eine ZHK übrig



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=∅ 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

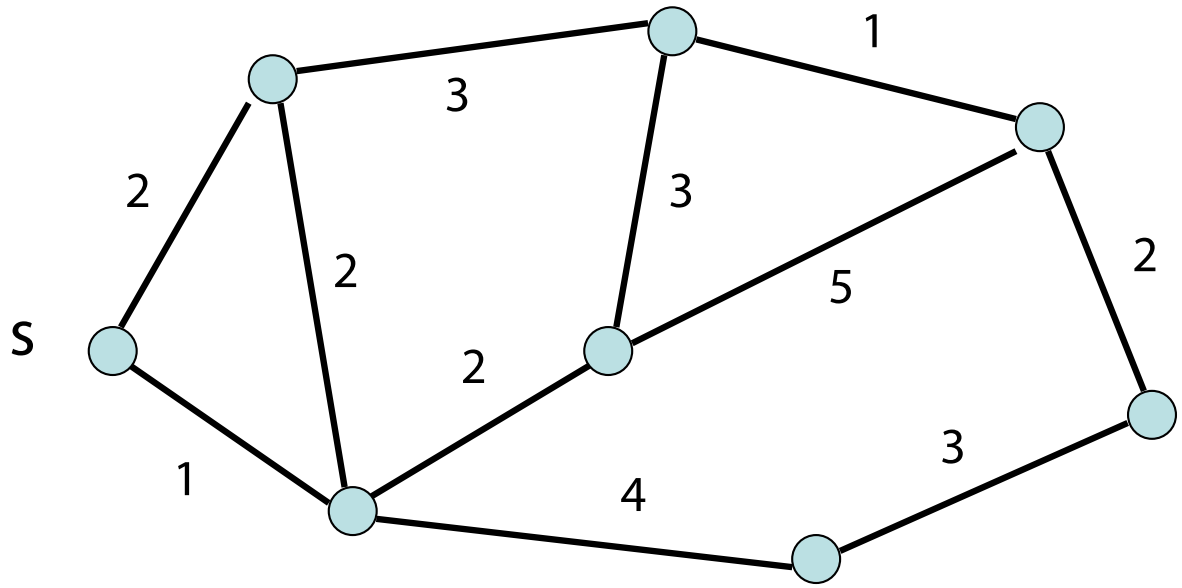
Erinnerung: Union-Find DS

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)



Kruskal-Algorithmus

```
Function MinSpanningTree((V, E), c):
    T:=⊘
    init(V) // initialisiere einelem. Mengen für V
    S:=mergesort(E)  // aufsteigend sortiert
    foreach {u,v}∈ S do
      if find(u)≠find(v) then // versch. Mengen
         T:=T ∪ { {u,v} }
         union(u, v)  // u und v in einer Menge
    return T
```

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

Minimaler Spannbaum

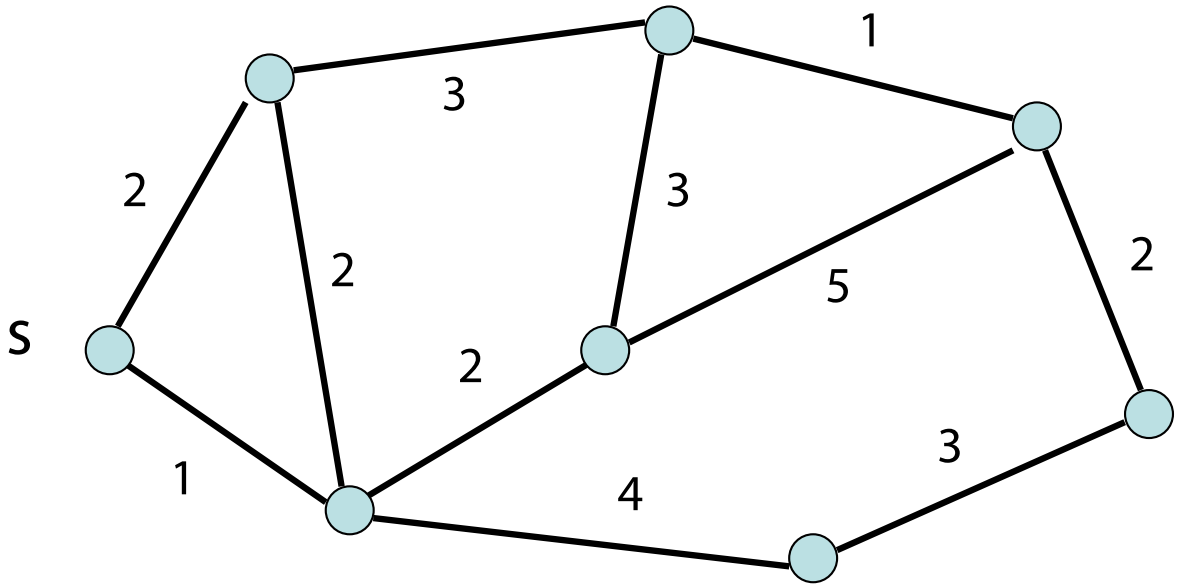
Alternative Strategie (motiviert aus Beh 2):

- 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

```
Procedure JarnikPrim(s: NodeId)
    d=<∞,…,∞>: Array of ℝ ∪ {-∞, ∞}
    parent=<⊥,…,⊥>: Array of NodeId
    d[s]:=0; parent[s]:=s // T anfangs nur aus s
    q=<s>: PQ with key(x) = d[x]
    while q ≠ ∅ do
       u:= deleteMin(q) // u: min. Distanz zu T in q
       foreach e={u,v} ∈ E ∧ parent[v] = ⊥ do
            if c(e) < d[v] then // aktualisiere d[v] zu T
                d := d[v]; d[v] := c(e); parent[v] := u
                if d=∞ // v noch nicht in q?
                    then insert(v, q)
                    else decreaseKey(v, d-d[v], q)
     return constructTree(s, parent)
```

Jarnik-Prim Algorithmus

Laufzeit:

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

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

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

Fibonacci Heap:

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

Vergleich: O(m log m) bei Kruskal (m > n)

Danksagung

Die nachfolgenden Präsentationen wurden mit einigen Änderungen übernommen aus:

- „Algorithmen und Datenstrukturen“
  gehalten von Sven Groppe an der UzL

Betrachtete Arten von Netzwerken

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

Netzwerke

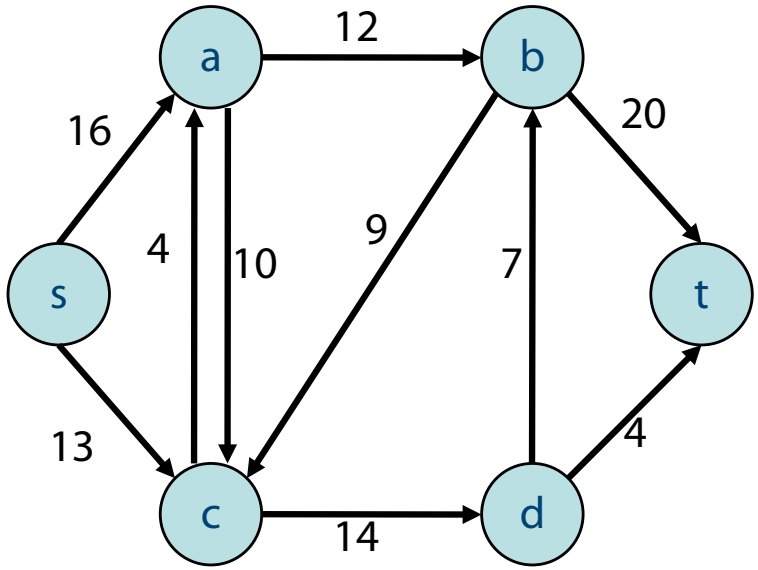
- Gegeben: Gerichteter Graph $G=(V, E)$
  - Kanten repräsentieren Flüsse von Material/Energie/Daten/…
    - Jede Kante hat eine maximale Kapazität, dargestellt durch (totale) Funktion $c: E \longrightarrow \mathbb{R}_+$
  - Knoten $s \in V$ als Quelle des Flusses
  - Knoten $t \in V$ als Senke des Flusses

- 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 $f: E \longrightarrow \mathbb{R}$

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

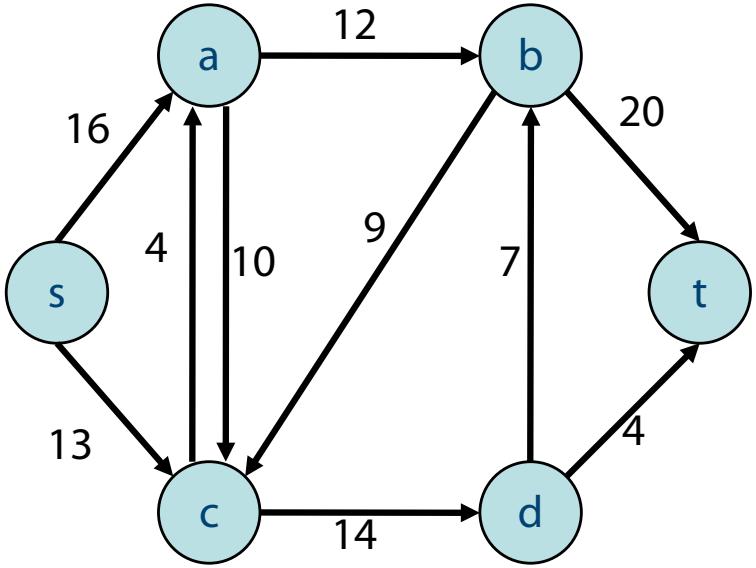


Jede Zahl steht für die Kapazität dieser Kante

- Finde die maximale Anzahl von Einheiten , die von der Quelle zu der Senke in diesem Graphen fließen kann (dargestellt durch Funktion $f_{max}: E \longrightarrow \mathbb{R}$)

Problem des maximalen Flusses in Netzwerken

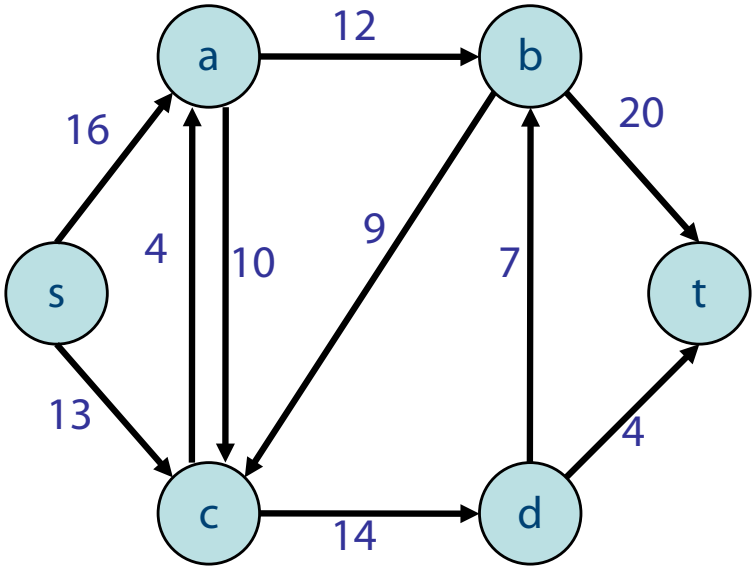
- Jede Kante könnte ein Wasserrohr darstellen
  - Von einer Quelle fließt Wasser zu einer Senke
  - Jedes Wasserrohr kann eine maximale Anzahl von Litern Wasser pro Sekunde transportieren



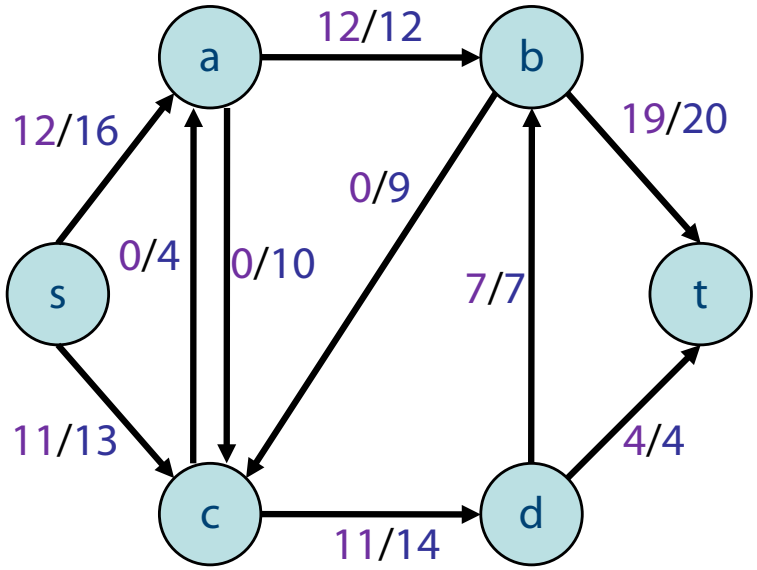
Jede Zahl steht für die Kapazität dieser Kante

- 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))

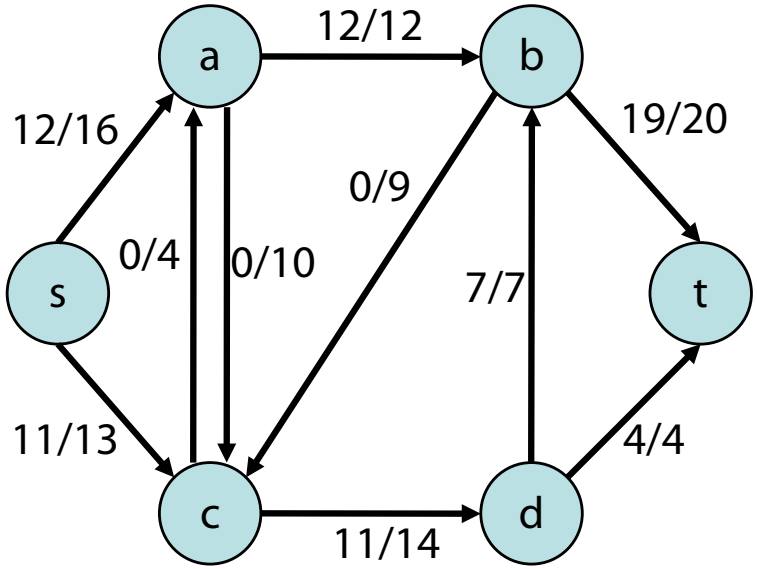


Dieser Graph enthält zusätzlich den Fluss im Graphen (Beschriftung f(e)/c(e)

- 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

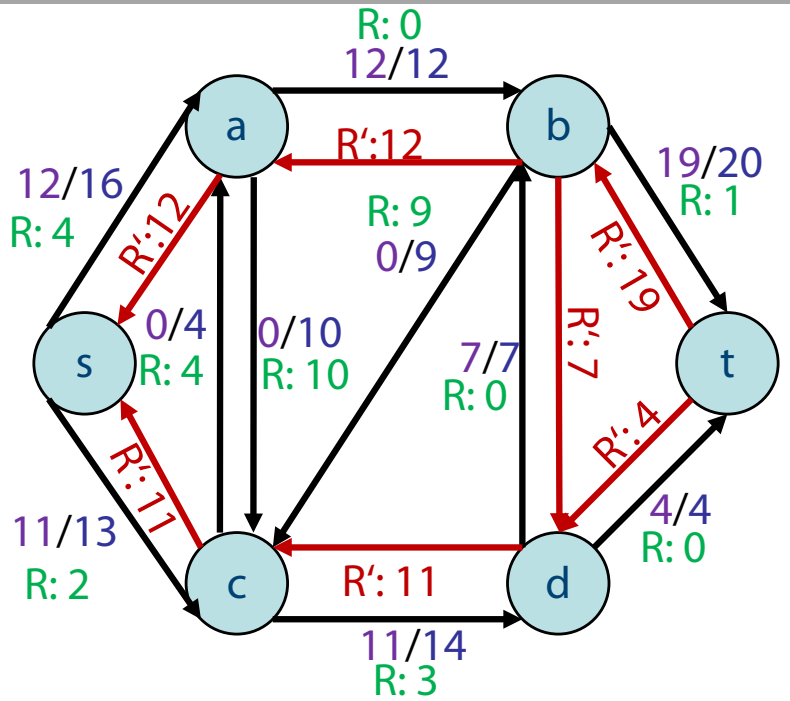
- Flusserhaltung:
  - Mit Ausnahmen der Quelle s und Senke t ist der Fluss, der in einen Knoten hineinfließt, genauso groß wie der Fluss, der aus diesem Knoten herausfließt
- Beachtung maximaler Kapazitäten:
  - Jeder Fluss in einer Kante muss kleiner oder gleich der Kapazität dieser Kante sein



Fluss / Kapazität im Graph

Netzwerkfluss

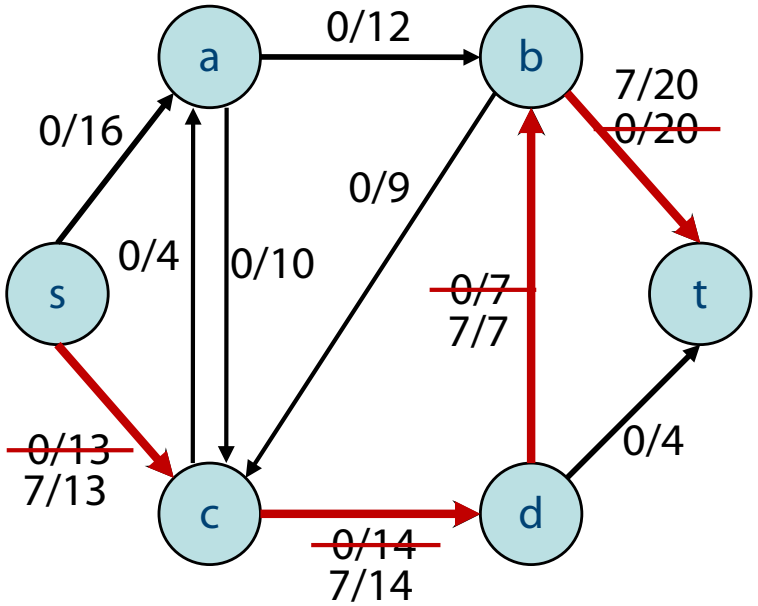
- Restkapazität einer Kante
  - Unbenutzte Kapazität jeder Kante
  - Zu Beginn ist der Fluss 0 und damit ist die Restkapazität genau so groß wie die Kapazität
  - Existiert ein Fluss, so kann der Fluss auch wieder reduziert werden, dies ist wie eine Restkapazität in die entgegengesetzte Richtung
- Restkapazität 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



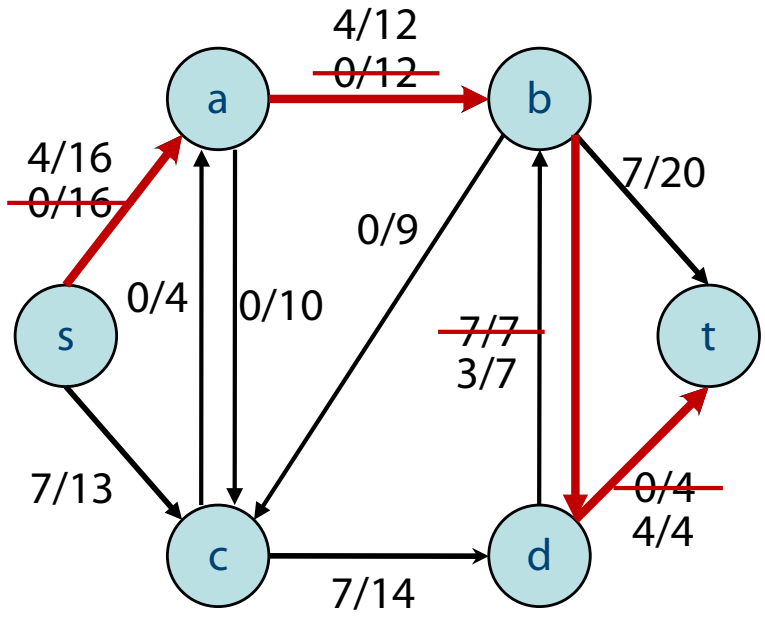
Fluss / Kapazität im Graph
Restkapazität R: Kapazität – Fluss
Restkapazität R‘ in die entgegen-gesetzte Richtung: Fluss

Beispiel für flusserhöhende Pfade

Flusserhöhender Pfad nur mit „normalen“ Restkapazitäten



Flusserhöhender Pfad auch mit Restkapazitäten in die entgegengesetzte Richtung



Ford-Fulkerson-Algorithmus

Procedure Ford-Fulkerson (G, f):

```
// Sei s Quelle und t Ziel in G=(V, E) mit s,t∈V
for (u, v) ∈ E do f(u,v) := 0
while ∃ p∈pfade(s, t, G): flusserhöhenderPfad (p) do
    for (u_i, v_i) auf p do
        f(u_i, v_i) := f(u_i, v_i) + Restkapazität von p
return f(s, t)
```

Pfadbestimmung mit Tiefensuche FF-DFS,
Flusserhöhung, wenn Restkapazität > 0

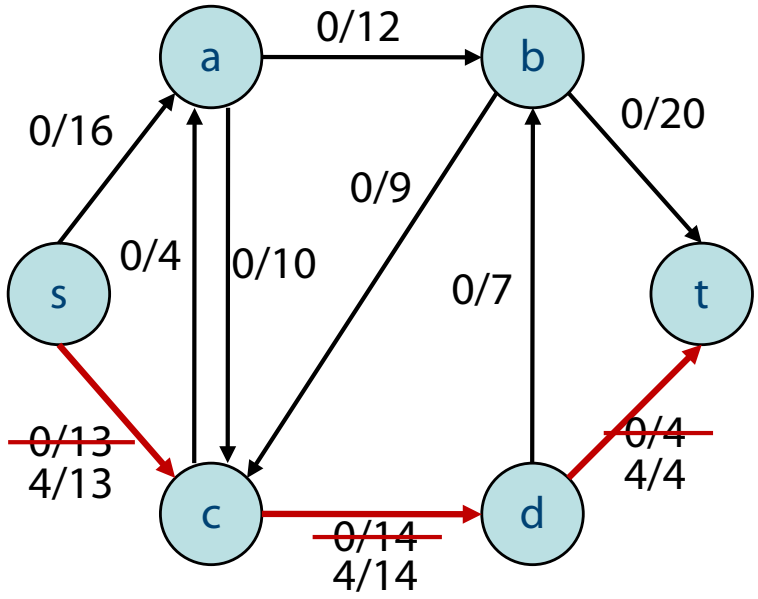
Ford, L. R.; Fulkerson, D. R. "Maximal flow through a network". Canadian Journal of Mathematics 8: 399. **1956**

Tiefensuche – Schema: FF-DFS

```
Function FF-DFS((V,E), s) :
    unmark all nodes
    DFS(((V,E), s,s)    // s: Sourceknoten
    return nil

Procedure DFS(((V,E), u,v: Node)  // u: Vater von v
    if not exists (v,w)∈E and flow-augmenting-path( path(v) ) then // v=t
        return-from FF-DFS path(v)
    foreach (v,w)∈E do
        if w is not marked then
            mark w with predecessor v
            DFS((V,E), v,w)
    backtrack(u,v)

backtrack(u,v):
    unmark(v)
```

Ford-Fulkerson Algo – Beispieldurchlauf

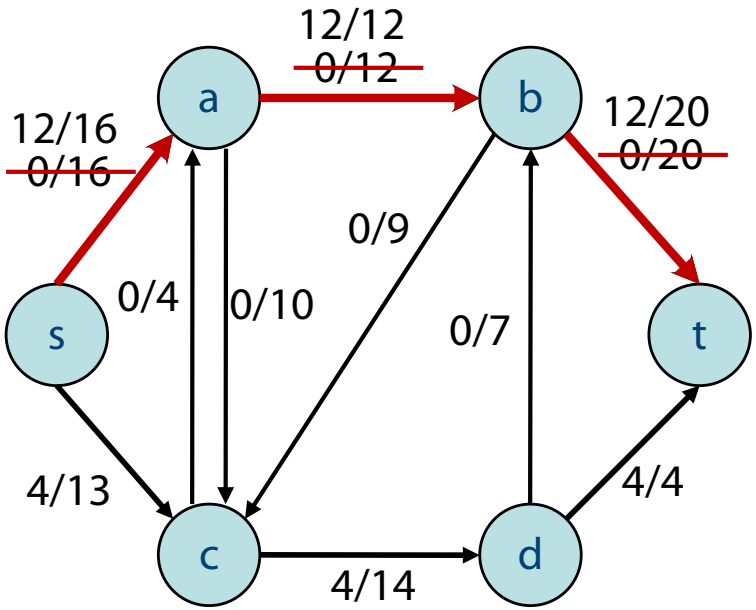
while $\exists p \in pfade(s, t, G)$: flusserhöhenderPfad (p) do
Erhöhe Fluss f von s nach t in p um Restkapazität von p



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

Ford-Fulkerson Algo – Beispieldurchlauf

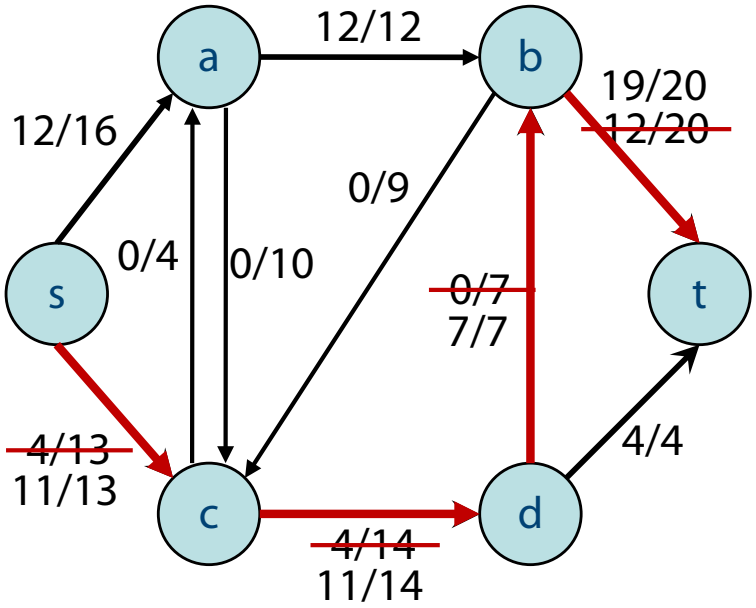
while $\exists p \in pfade(s, t, G)$: flusserhöhenderPfad (p) do
Erhöhe Fluss f von s nach t in p um Restkapazität von p



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

Ford-Fulkerson Algo – Beispieldurchlauf

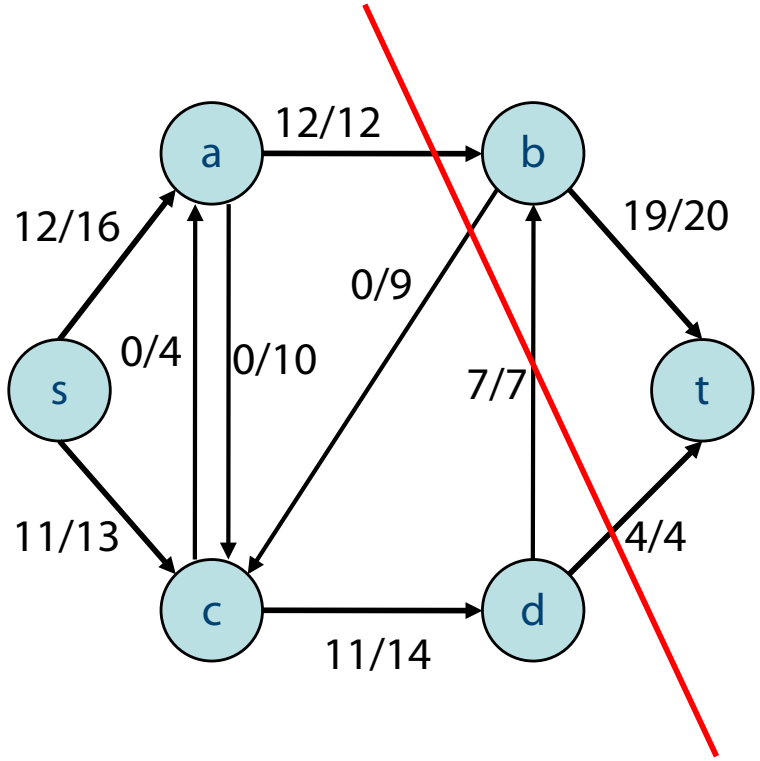
while $\exists p \in pfade(s, t, G)$: flusserhöhenderPfad (p) do
  Erhöhe Fluss f von s nach t in p um Restkapazität von p



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

Ford-Fulkerson Algo – Beispieldurchlauf

while $\exists p \in pfade(s, t, G)$: flusserhöhenderPfad (p) do
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)

Analyse des Algorithmus von Ford/Fulkerson

- Ein Schnitt in $N=((V, E), c, s, t)$ ist ein 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 minimalem Schnittes ist
  $c_{min} = \min_{(S, T) \text{ Schnitt in } N} c(S, T)$
- Der Flusswert 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} bezeichnem 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: $f_{max} \leq c_{min}$.
- Sei f der vom F.F.-Algo für N berechnete Fluss.
  Dann gibt es einen Schnitt (S,T) in N mit f(G) = c(S,T).

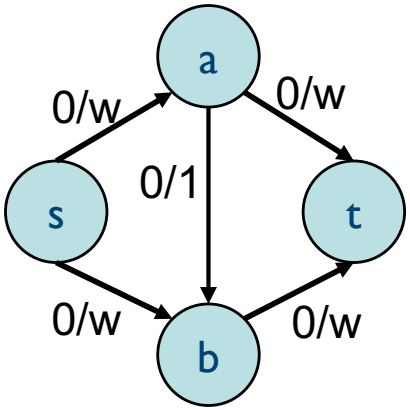
- Satz: (Max Flow-Min Cut Theorem; Satz von Ford/Fulkerson)
  Der Algorithmus von Ford/Fulkerson berechnet einen maximalen Fluss. In jedem Netzwerk gilt $f_{max} = c_{min}$.
  (ohne formalen Beweis)

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

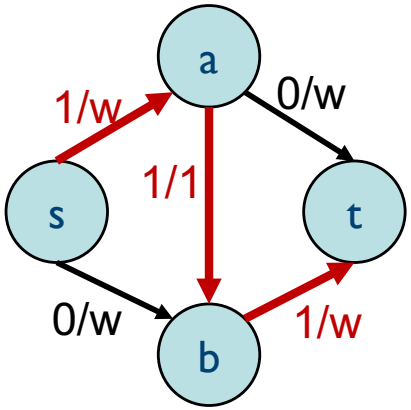
Ford-Fulkerson Algorithmus – Analyse

- Finden eines flusserhöhenden Pfades z.B. mit einer Variation der Tiefensuche: $O(n + m)$
- Restkapazität des flusserhöhenden Pfades kann jedes Mal nur 1 sein.

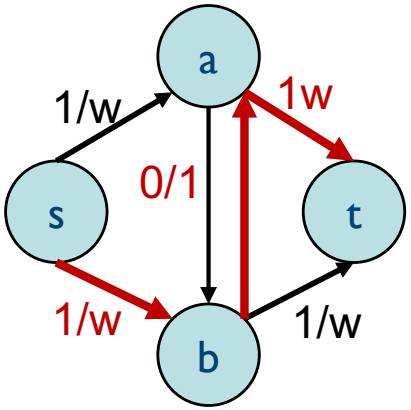
Schlechte Abfolge von zunehmenden Pfaden



1. flusserhöhender Pfad



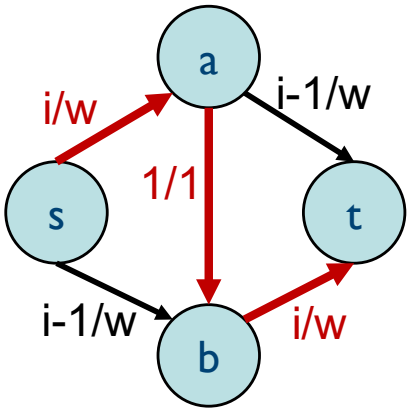
2. flusserhöhender Pfad



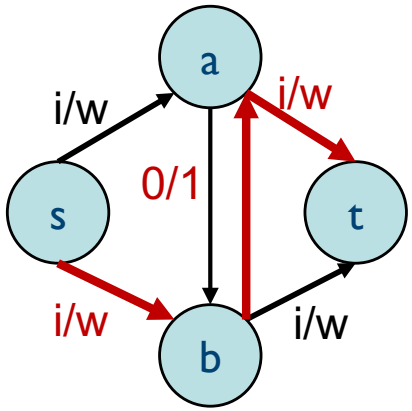
…

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

2i-1. flusserh. Pfad

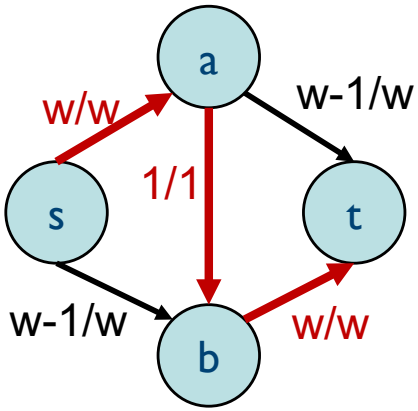


2i. flusserh. Pfad

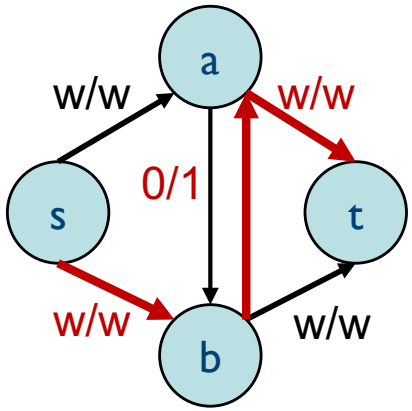


…

2w-1. flusserh. Pfad



2w. flusserh. Pfad



Ford-Fulkerson Algorithmus – Analyse

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

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

Da für die betrachteten Gs 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

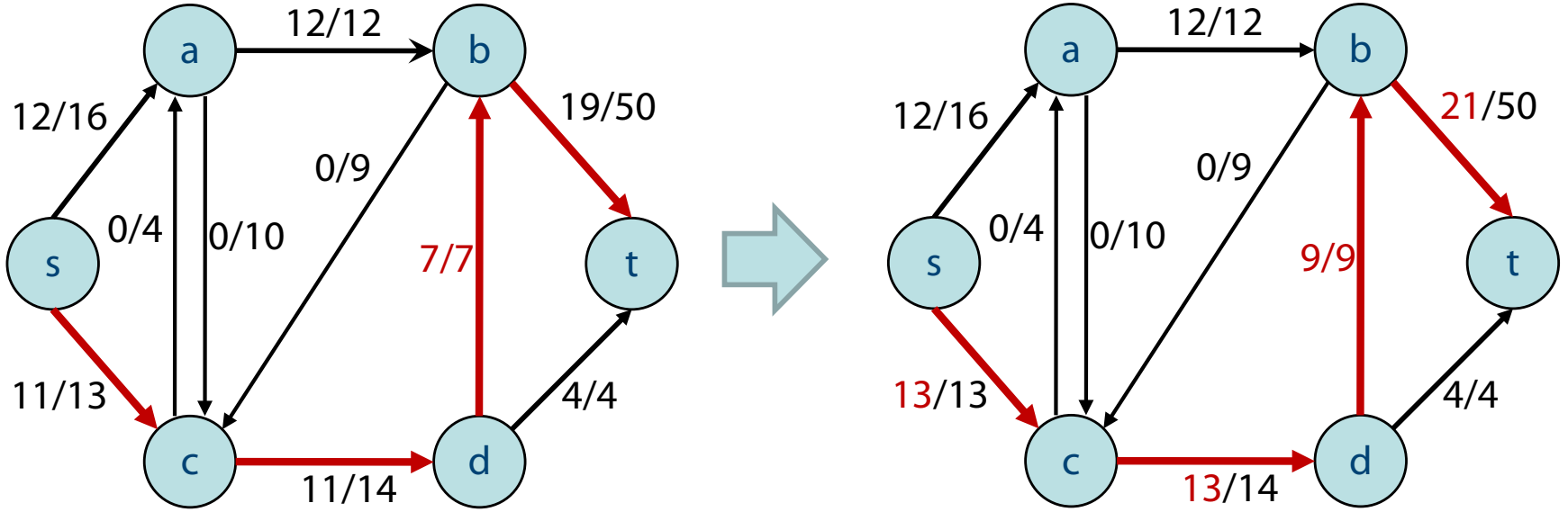
Edmonds-Karp Algorithmus

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

Praktische Fragestellung

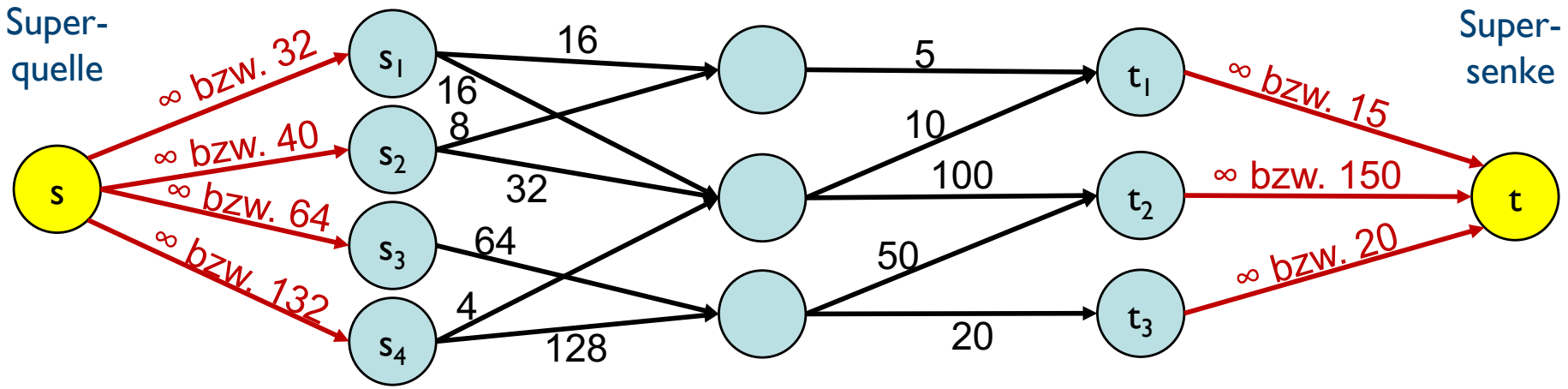
- Wie kann man durch Erhöhung der Kapazität an einer/wenigen Kanten den maximalen Fluss erhöhen?
  - Betrachte Pfade von der Quelle zu der Senke, deren Fluss die volle Kapazität einer Kante ausnutzen
  - Erhöhe die Kapazität der Kante(n), die die volle Kapazität ausnutzen, um das Minimum der Restkapazitäten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



Mehrere Quellen und mehreren Senken

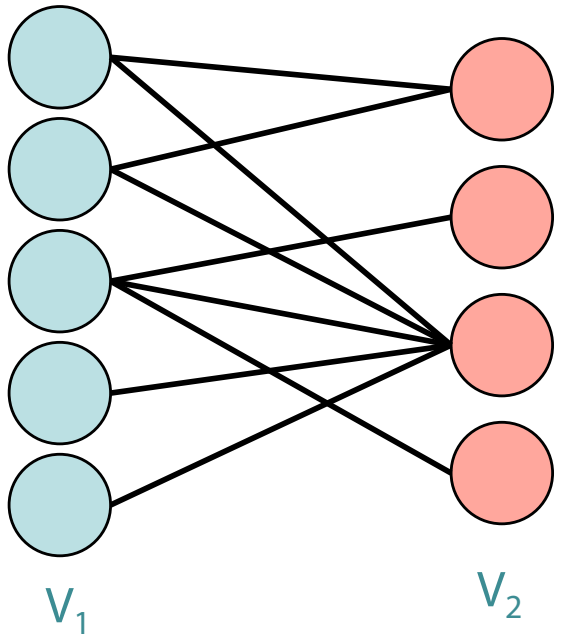
- Reduzierung auf maximalen Fluss in Netzwerk mit einer Quelle und einer Senke durch Einführung
  - einer Superquelle, die mit allen Quellen
  - einer Supersenke, die von allen Senken

  mit 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



Anwendung: Maximale bipartite Matchings

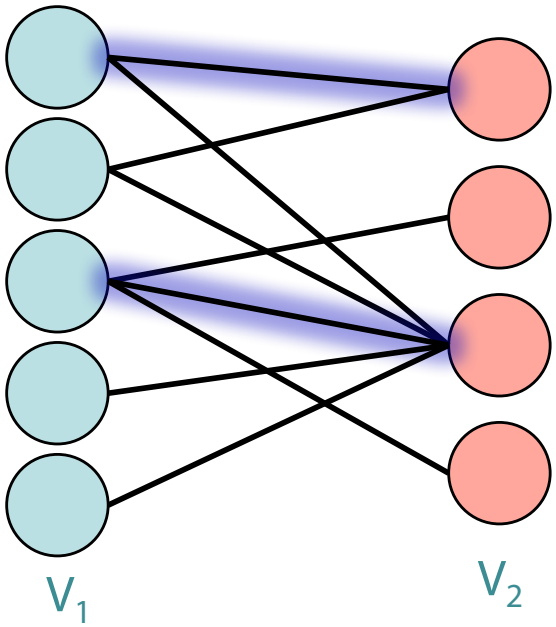
- Bipartite Graphen sind Graphen $G=(V, E)$ in denen die Knotenmenge V in zwei disjunkte Knotenmengen V_1 und V_2 aufgeteilt werden können $(V = V_1 \cup V_2)$, so dass $\forall\ (u, v) \in E: (u \in V_1 \wedge v \in V_2) \vee (u \in V_2 \wedge v \in V_1)$
- Beispiel eines bipartiten Graphen:
  - Knoten 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



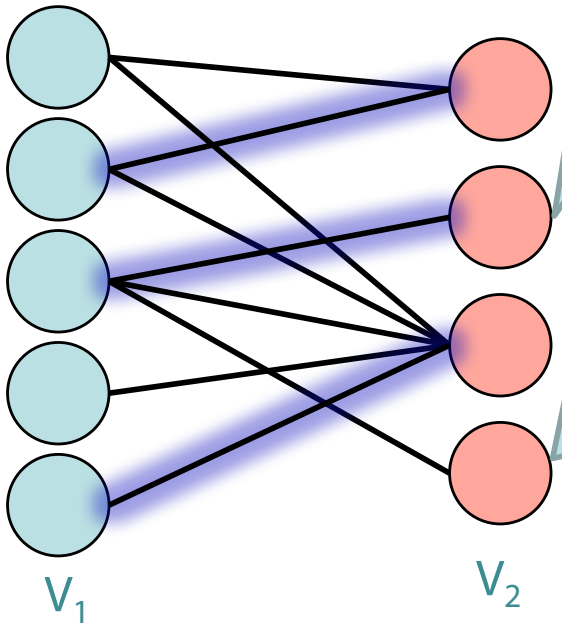
Bipartites Matching

- Finde $E' \subseteq E$, so dass $\forall v \in V$: degree(v)≤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

Nicht maximal:



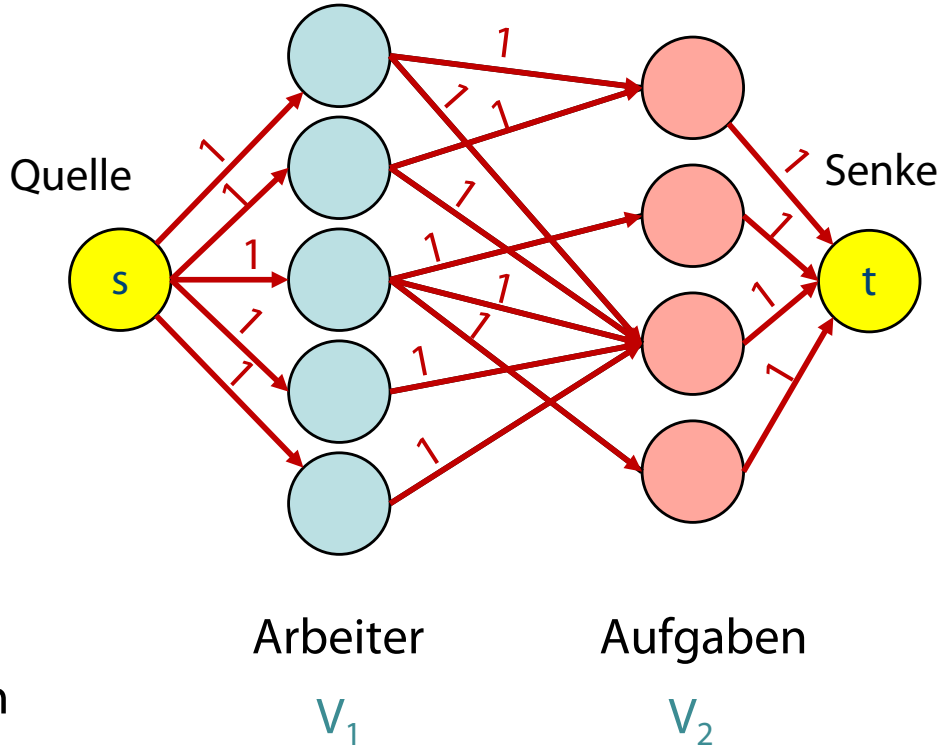
Maximal:



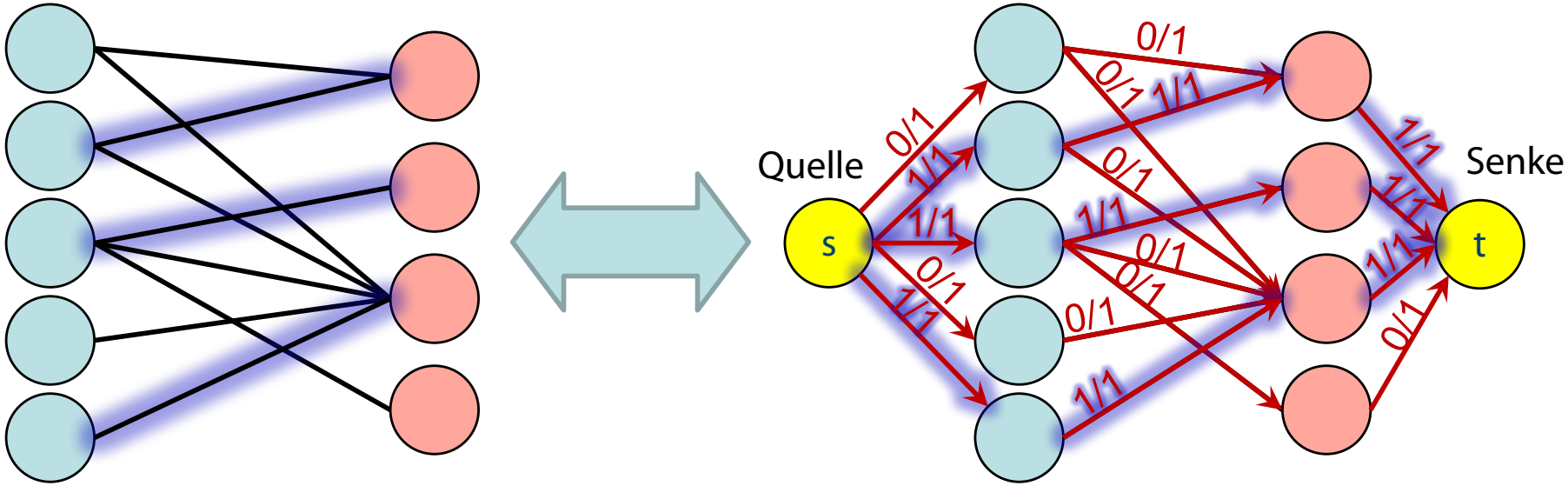
Diese Aufgaben können nur von ein und demselben Arbeiter erledigt werden, daher kein größeres Bipartites Matching möglich!

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



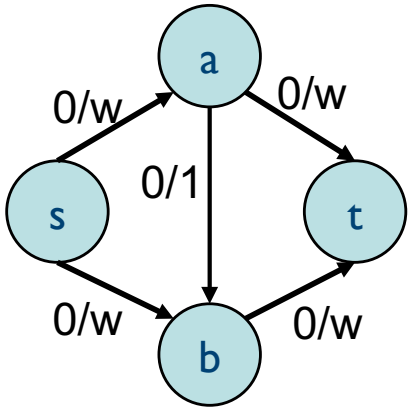
Maximaler Fluss ⇔ Maximales bipartites Matching



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

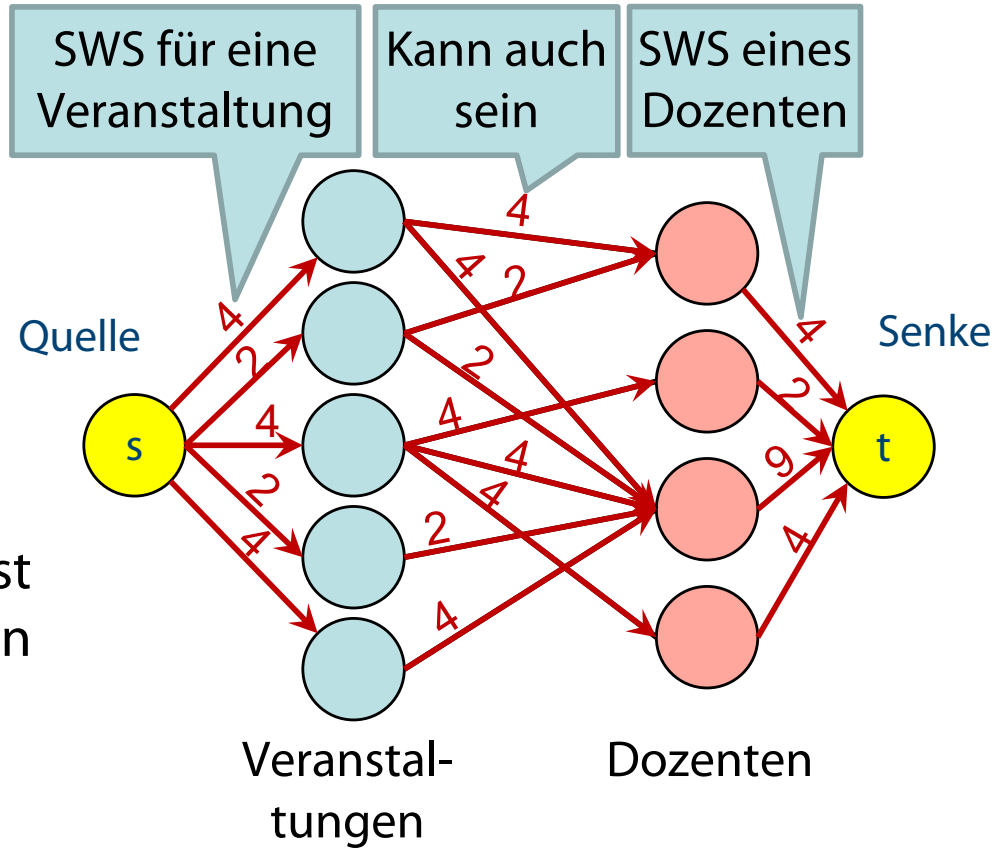
$T_{\text{bipartite-match}}(G) \in O(c \cdot (n+m))$

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



Weiteres Beispiel zur Lösung von kombinatorischen Problemen mit Hilfe des maximalen Flusses

- Zuordnung von Lehrveranstaltungen zu Dozenten
  - Jeder Dozent hat eine Verpflichtung zur Lehre einer gewissen Anzahl von Semesterwochenstunden
  - Jede Lehrveranstaltung hat einen Umfang von gegebenen Semesterwochenstunden
  - Jeder Dozent kann nur bestimmte Lehrveran-staltungen halten
  - Mehrere Dozenten können sich eine Lehrveranstaltung teilen
  - Finde eine Lösung, in der möglichst viele Lehrveranstaltungen gehalten werden



Übersicht über Max-Flow-Algorithmen

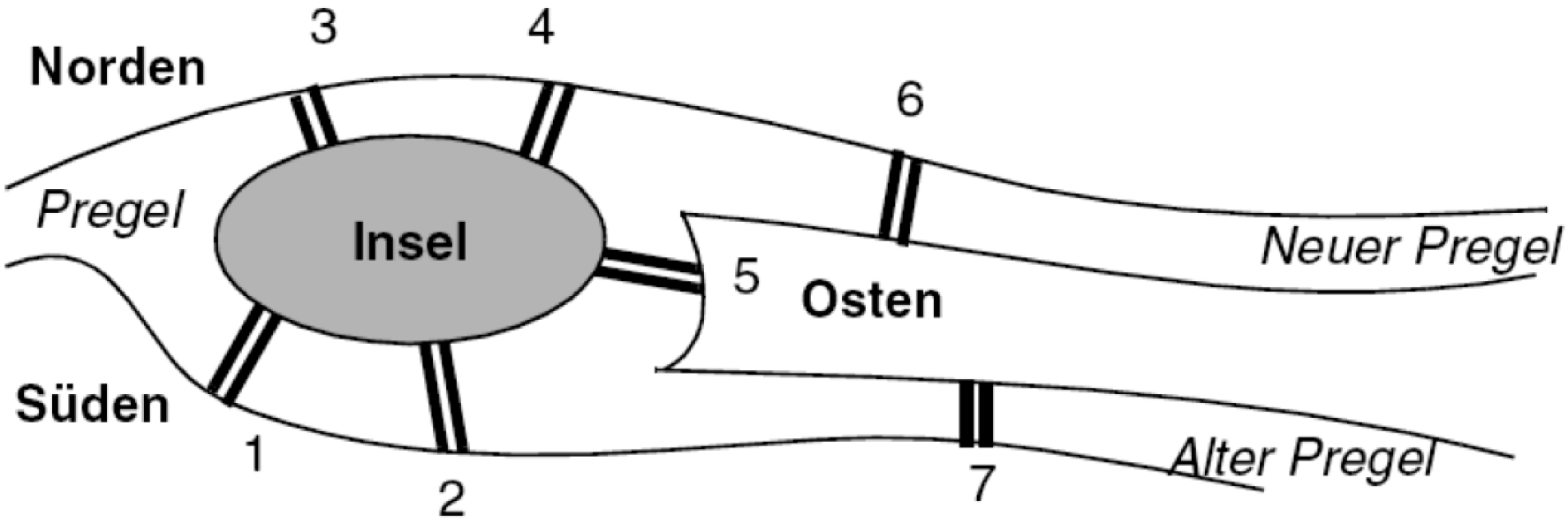
(n=|V|, e=|E|, U=max{c(e) für alle e∈E})

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	Cheriyan/Hagerup (rand.)	$O(ne + n^2\log^3 n)$	
	det. Version von Alon	$O(\min(ne\log n, ne + n^{8/3}\log n))$	
	det. Version von Tarjan	$O(\min(ne\log n, ne + n^2\log^2 n))$	
1990	Cheriyan/Hagerup/Mehlhorn rand.	$O(n^3/\log n)$ $O(\min(ne\log n, ne + n^2\log^2 n, n^3/\log n))$	

Königsberger Brückenproblem

Gibt es einen Weg über alle sieben Brücken von einem beliebigen Ausgangspunkt zurück zum Ausgangspunkt?

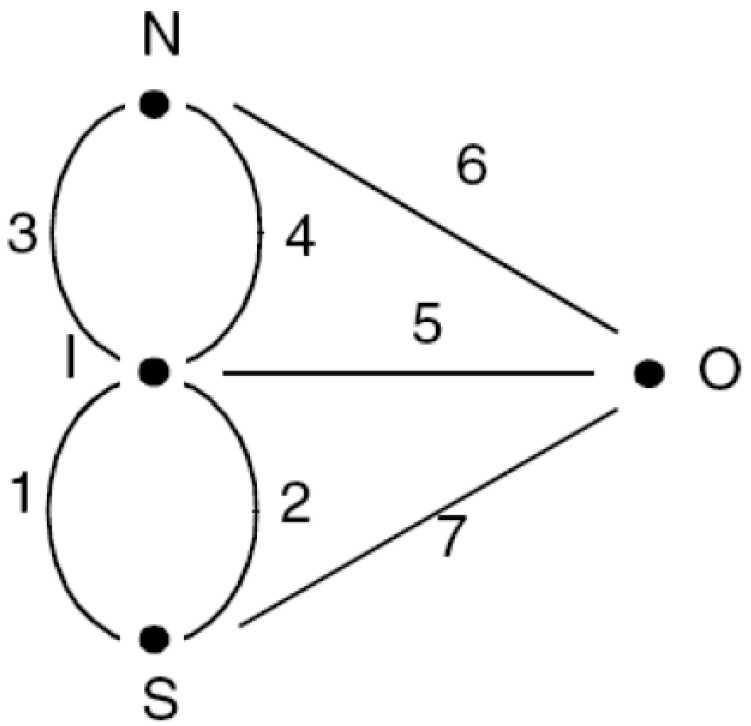
- Wobei jede Brücke nur einmal benutzt werden darf.



Transformation des Problems

Darstellung als Prüfung von Grapheigenschaften

- Insel, Landgebiet: Knoten
- Gebietsnamen
  - Knotenbeschriftung aus {S, I, N ,O}
- Brücken: Kanten
- Brückennamen:
  - Kantenbeschriftung



Definition des Problems

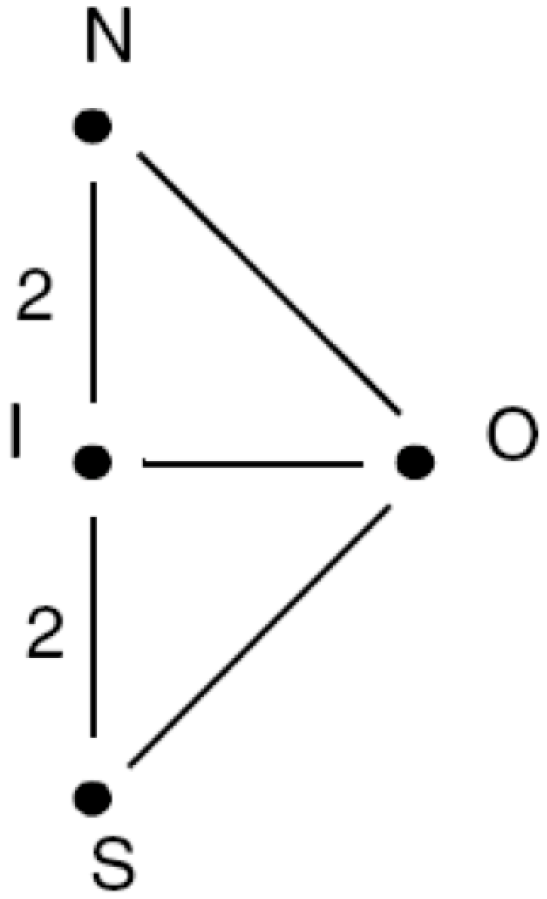
- Eingabe:
  - Welche Brücke führt von wo nach wo?
- Ausgabe:
  - Ja, es gibt einen geschlossenen Weg
    (alle Landstücke besucht und Anfang = Ende)
    oder nur einen offenen Weg
    (alle Landstücke besucht aber Anfang ≠ Ende) oder
  - Nein, es gibt keine Lösung

Entwurfsmuster Suchproblem?

- Idee für ein Verfahren:
  - Starte an einem Ort (z.B. Süden)
  - Durchlaufe alle möglichen Wege unter Einhaltungen der Bedingungen, so dass der Startort (Süden) wieder erreicht wird und jede Brücke nur einmal benutzt wird
  - Wenn Weg gefunden, Lösung ausgeben
    Und sonst ?
  - Frage: Müssen wir für einen offenen Weg sogar von jedem Ort $\in$ {N, S, O, I} aus unsere Suche beginnen?
- Es ergäbe sich ein Verfahren mit hohem Aufwand

Beobachtung

- Es reicht vielleicht, nur die Anzahl der Brücken zwischen zwei Knoten zu erfassen



Neuformulierung des Problems

- Gibt es einen Weg (zusammenhängende Folge von Kanten), der alle Kanten genau einmal enthält (Knoten beliebig oft) und möglichst geschlossen ist (Anfangsknoten = Endknoten)
  - Wir beschränken uns auf die Frage nach der Existenz eines solchen Wegs (und verzichten auf die Bestimmung eines solchen Weges, falls es ihn gibt)
- Besitzt der Graph einen Eulerweg bzw. Eulerkreis?

Trotz seines Namens ist der Eulerkreis kein Kreis, zumindest wenn man der häufigen Definition folgt, nach der sich in einem Kreis kein Knoten wiederholen darf.

Analyse auf Graphenebene

- Beim Passieren eines Knotens (hin- und wieder weg) werden zwei anliegende Kanten verwendet
- Ein Knoten u mit einer ungeraden Anzahl von anliegenden Kanten kann also nur ein Randknoten des gesuchten Weges sein
- Die Anzahl U solcher Knoten u kann nur 0 oder ganzzahlig sein
- Wenn
  - $U = 0$: dann existiert Eulerkreis (mit beliebigem Anfang)
  - $U = 2$: dann existiert Eulerweg
  - sonst existiert keine Lösung

Aus einer Arbeit von Leonhard Euler, **1736**
Wladimir Velminski: *Leonhard Euler. Die Geburt der Graphentheorie*. Kulturverlag Kadmos, Berlin **2008**

Algorithmus Euler (1)

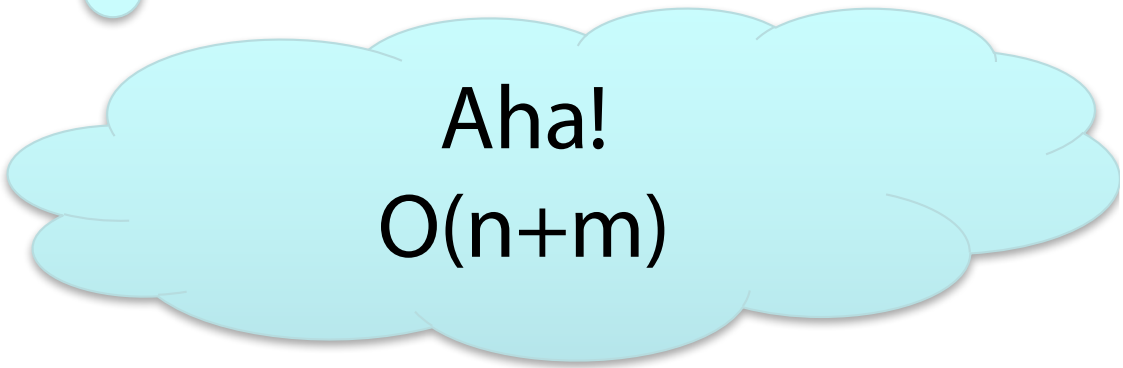
```
Function Euler( (V, E) )
    U := 0;
    foreach v∈ V do
        if odd?(degree(v, E)) then
            U := U+1
    if U=0 then
        return “Eulerkreis“
    else if U = 2 then
            return “Eulerweg“
        else return ⊥
```

Wieviele Schritte benötigt degree(v, E)?

O(m) mit m = |E|
Oh! Und das in einer Schleife!
Macht O(nm) insgesamt.

Algorithmus Euler (2)

```
Function Euler( (V, E) )
    d=<0, ... 0> Array [1..length(V)] of IN   // Grad (degree) für alle Knoten in V
    foreach (u, v) ∈ E do
        d[u] := d[u]+1
        d[v] := d[v]+1
    U := 0; i := 1
    while i ≤ length(V) do
        if odd?(d[i]) then U := U +1
        i := i +1
    if U=0 then
        return "Eulerkreis"
    else if U = 2 then
            return "Eulerweg"
         else return ⊥
```

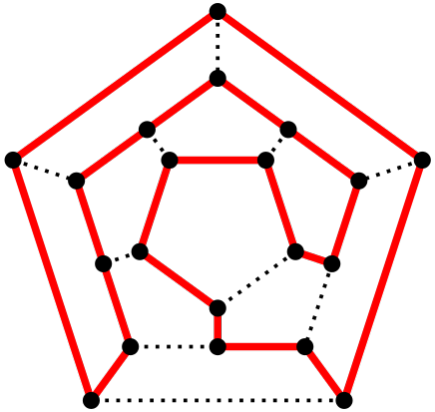


Entwurfsmuster: Nachdenken

- Nachdenken wandelt ein scheinbares Suchproblem in eine Berechnung mit linearem Aufwand O(n+m), wobei n die Anzahl der Knoten und m die Anzahl der Kanten im Graphen ist

Kleine Modifikation

- Bisher studiertes Problem: „Euler-Kreis/Weg vorhanden“
  Jede Kante einmal überschritten
- Neues Problem: „Hamilton-Kreis vorhanden“
  Jeder Knoten genau einmal berührt
- Ist erheblich schwieriger, es gibt weder
  eine einfache hinreichende Bedingung
  noch eine einfache notwendige Bedingung



[Wikipedia]

- Interessanterweise ist es einfach, eine vorgeschlagene Lösung zu verifizieren
  - NB: Einfache Verifikation ist nicht immer der Fall

Zusammenfassung Fluss in Netzwerken

- Ford-Fulkerson Algorithmus
  - zus. Variante von Edmonds und Karp zur Auswahl der flusserhöhenden Pfade betrachtet
- Anwendungen des maximalen Flusses
  - Maximale bipartite Matchings
- 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