

# Algorithmen und Datenstrukturen



TECHNISCHE  
UNIVERSITÄT  
DARMSTADT



**Cryptoplexity**

Cryptography & Complexity Theory  
Technische Universität Darmstadt  
[www.cryptoplexity.de](http://www.cryptoplexity.de)

Prof. Marc Fischlin, SS 2023

---

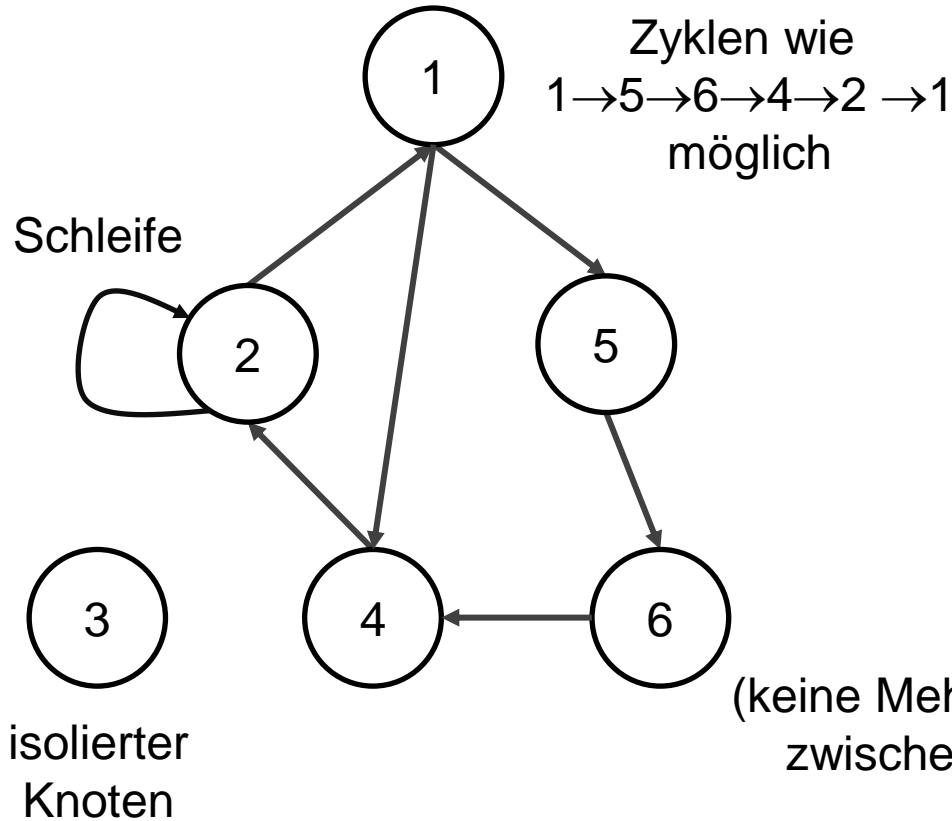
06

Graph Algorithms

---

# Graphen

# (Endliche) Gerichtete Graphen



$$V = \{1, 2, 3, 4, 5, 6\}$$

$$E = \{(1, 4), (1, 5), (2, 1), (2, 2), (4, 2), (5, 6), (6, 4)\}$$

Ein (endlicher) gerichteter Graph  $G = (V, E)$  besteht aus

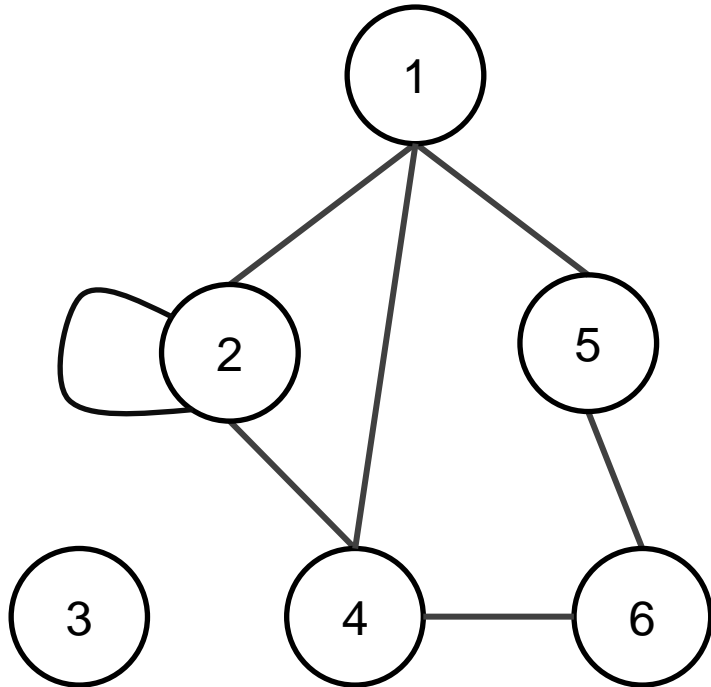
(1) einer (endlichen) Knotenmenge  $V$  („vertices“)

(2) einer (endlichen) Kantenmenge  $E \subseteq V \times V$  („edges“)

$(u, v) \in E$ : Kante von Knoten  $u$  zu  $v$

im Unterschied zu Bäumen  
Anordnung der Knoten  
in der Darstellung irrelevant

# Ungerichtete Graphen



$$V = \{1, 2, 3, 4, 5, 6\}$$

$$E = \{\{1, 4\}, \{1, 5\}, \{1, 2\}, \{2, 2\}, \{2, 4\}, \{4, 6\}, \{5, 6\}\}$$

Ein (endlicher)  
**ungerichteter Graph**  $G = (V, E)$  besteht aus

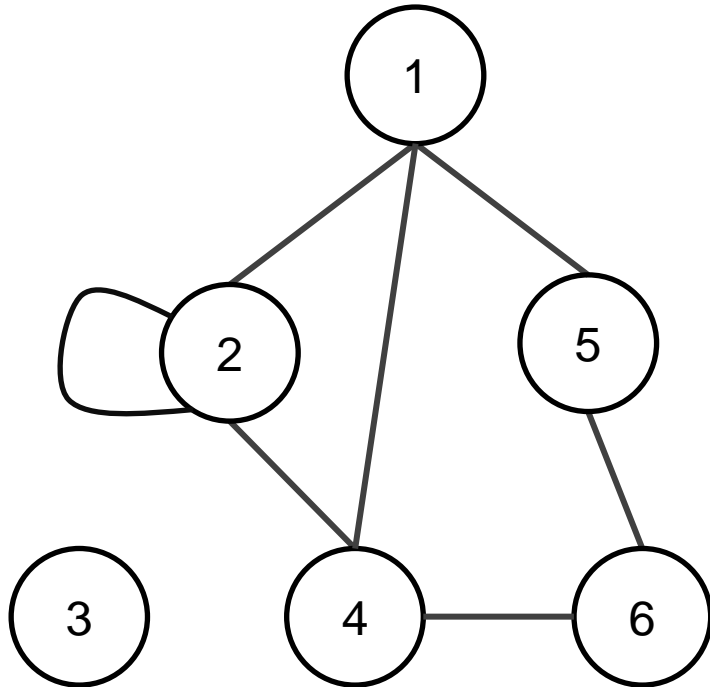
(1) einer (endlichen)  
Knotenmenge  $V$   
(„vertices“)

(2) einer (endlichen)  
Kantenmenge  $E \subseteq V \times V$   
(„edges“), **so dass**  
 $(u, v) \in E \Leftrightarrow (v, u) \in E$

Alternative Darstellung:

|                       |                  |
|-----------------------|------------------|
| —                     | $\{u, v\}$       |
| statt                 | statt            |
| $\longleftrightarrow$ | $(u, v), (v, u)$ |
| $\longrightarrow$     |                  |

# Pfadfinder



Knoten  $v$  ist von Knoten  $u$  im Graphen  $G = (V, E)$  erreichbar, wenn es Pfad  $(w_1, \dots, w_k) \in V^k$  gibt, so dass  $(w_i, w_{i+1}) \in E$  für  $i = 1, 2, \dots, k - 1$  und  $w_1 = u$  und  $w_k = v$ .

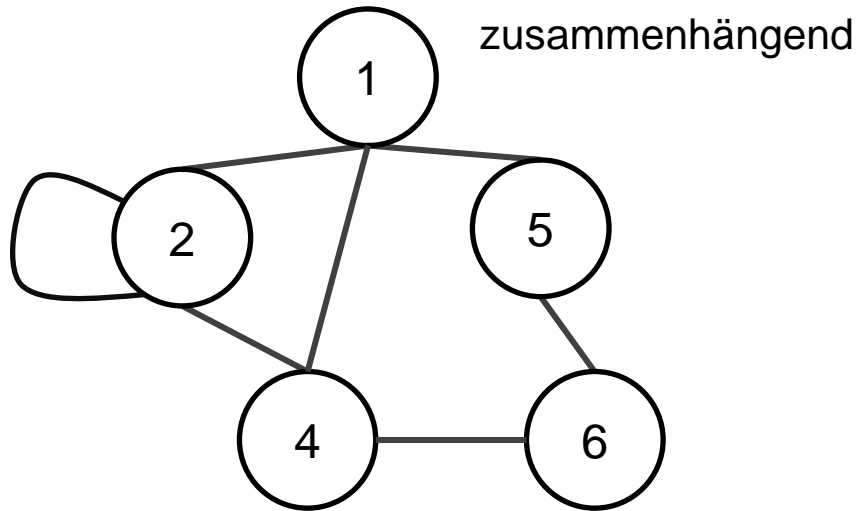
Insbesondere ist  $u$  immer von  $u$  per „leerem Pfad“ ( $k = 1$ ) erreichbar.

Länge des Pfades =  $k - 1$  = Anzahl Kanten

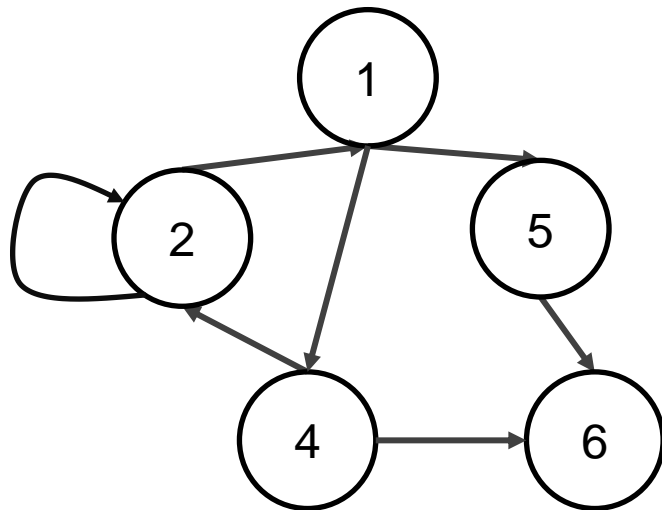
$(w_1, \dots, w_k)$  ist ein kürzester Pfad von  $u$  nach  $v$ , wenn es keinen kürzeren Pfad gibt.

$shortest(u, v)$  = Länge eines kürzesten Pfades von  $u$  nach  $v$

# Zusammenhänge



Ungerichteter Graph  
ist **zusammenhängend**,  
wenn jeder Knoten von jedem  
anderen Knoten aus erreichbar ist

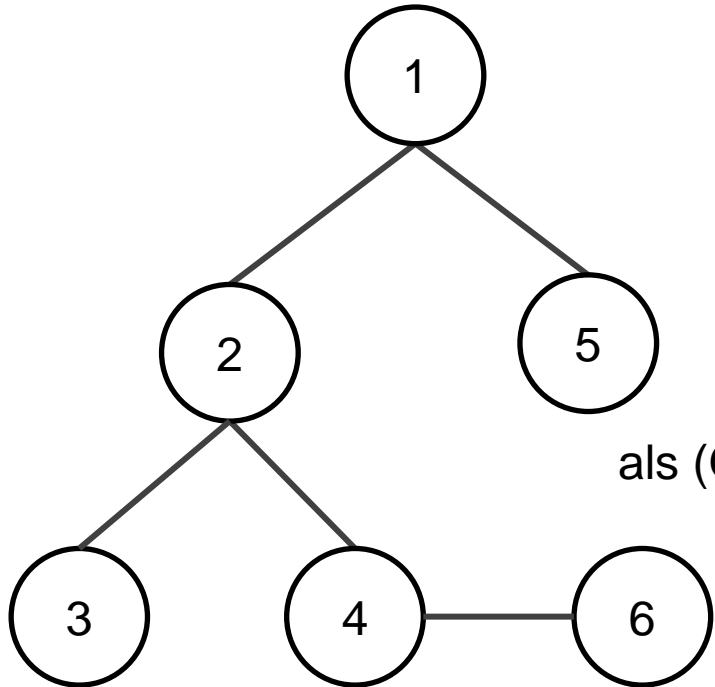


nicht stark  
zusammenhängend,  
da kein Pfad  $5 \rightarrow 1$

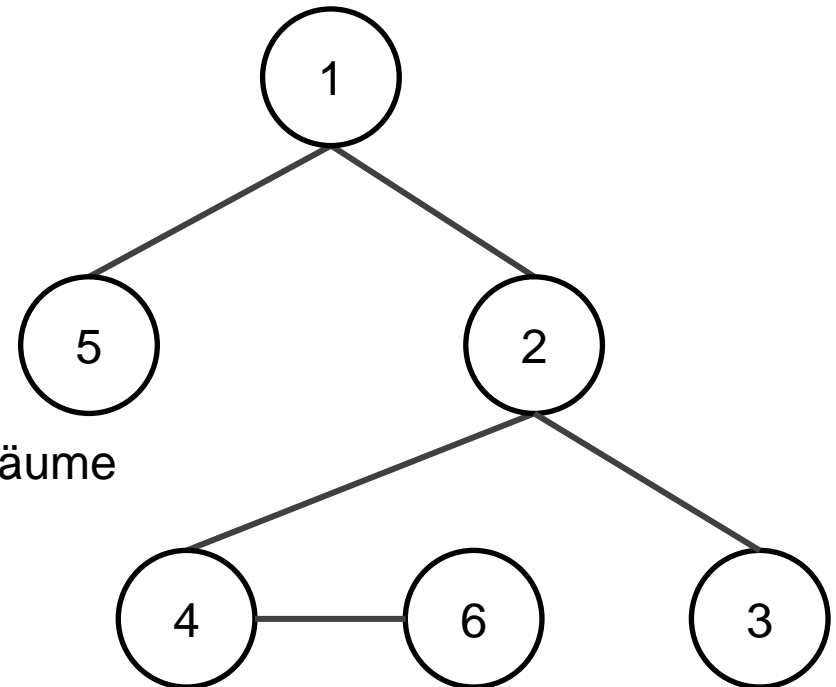
Gerichteter Graph  
ist **stark zusammenhängend**,  
wenn jeder Knoten von jedem  
anderen Knoten aus  
(gemäß Kantenrichtung)  
erreichbar ist

# Graphen und Bäume

Achtung: Diese (Graphen-)Bäume haben keine Ordnung auf den Kindern!



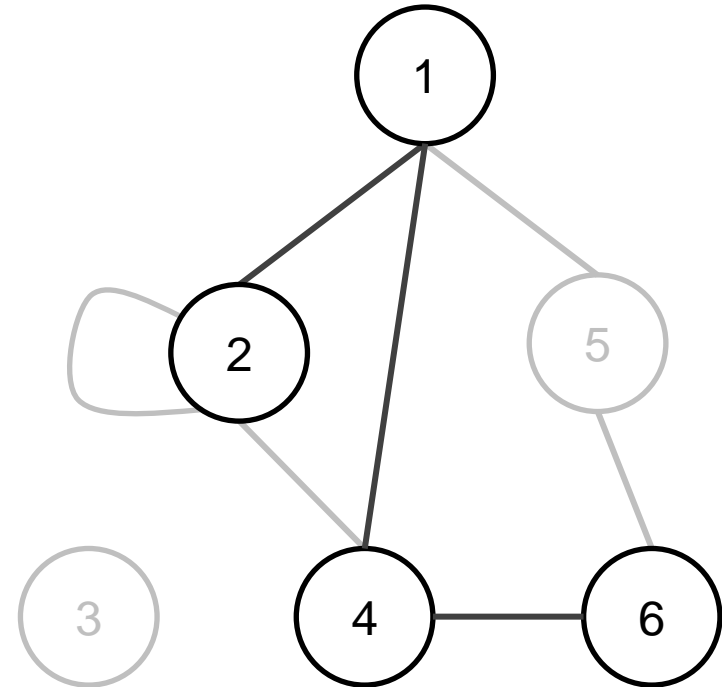
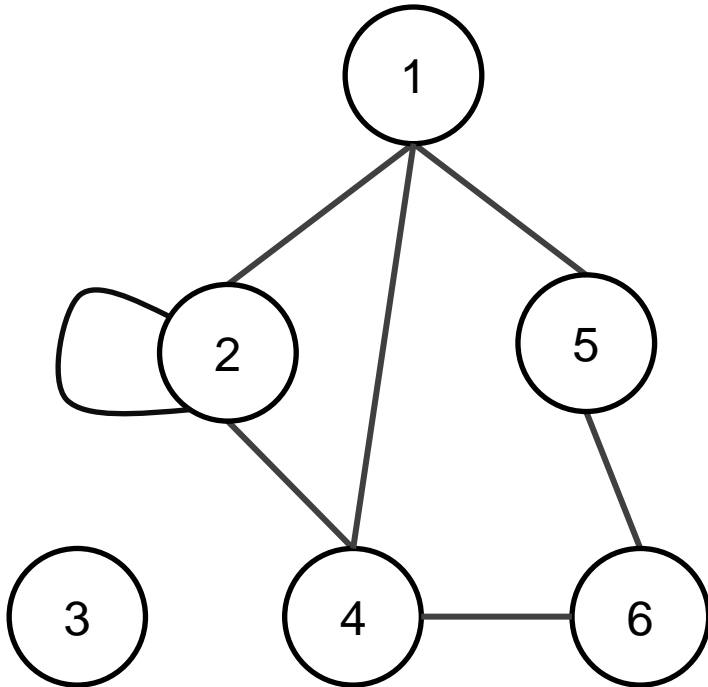
als (Graphen-)Bäume  
=



Graph  $G = (V, E)$  ist ein **Baum**, wenn  $V$  leer ist oder es einen Knoten  $\mathbf{x} \in V$  („Wurzel“) gibt, so dass jeder Knoten  $\mathbf{v}$  von der Wurzel aus per eindeutigem Pfad erreichbar ist.

# Subgraphen

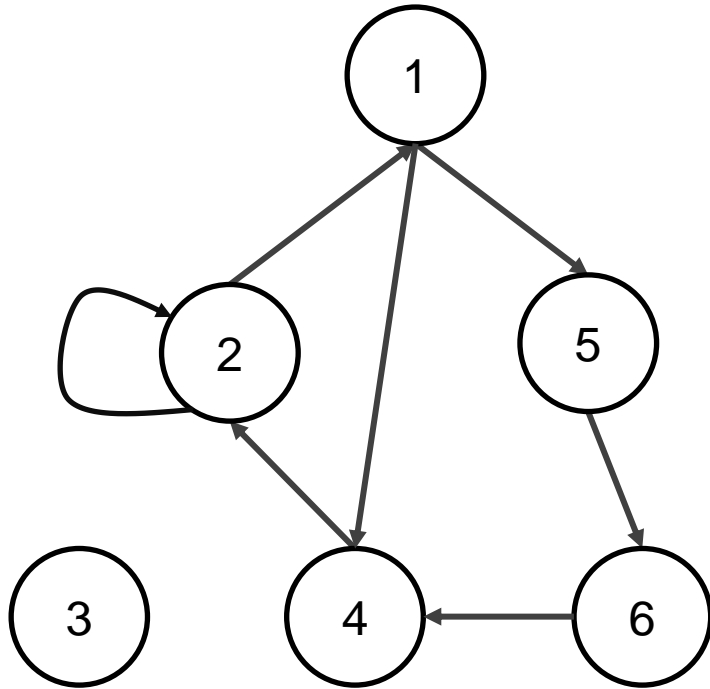
Beachte:  $G'$  muss selbst wieder ein Graph des gleichen Typs (gerichtet oder ungerichtet) sein!



(Gerichteter oder ungerichteter) Graph  $G' = (V', E')$  ist Subgraph (oder Untergraph oder Teilgraph) des (gerichteten oder ungerichteten) Graphen  $G = (V, E)$ , wenn  $V' \subseteq V$  und  $E' \subseteq E$ .



# Darstellung von Graphen (I)



$$V = \{1, 2, 3, 4, 5, 6\}$$

$$E = \{(1, 4), (1, 5), (2, 1), (2, 2), (4, 2), (5, 6), (6, 4)\}$$

Als Adjazenzmatrix:

$$A[i, j] = \begin{cases} 1 & \text{wenn Kante von } i \text{ zu } j \\ 0 & \text{wenn keine Kante} \end{cases}$$

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

bei ungerichteten Graphen ist  
Matrix (spiegel-)symmetrisch  
zur Hauptdiagonalen

$$\text{Speicherbedarf} = \Theta(|V|^2)$$

# Matrix → Eigenschaft (I)

Der Eintrag  $a_{i,j}^{(m)}$  in der  $i$ -ten Zeile und  $j$ -ten Spalte der  $m$ -ten Potenz  $A^m$  der Adjazenzmatrix  $A$  eines Graphen gibt die Anzahl der Wege an, die von Knoten  $i$  zu Knoten  $j$  entlang von genau  $m$  Kanten führen ( $m \geq 0$ ).

Per Induktion (Basisfall  $m = 0$ ):

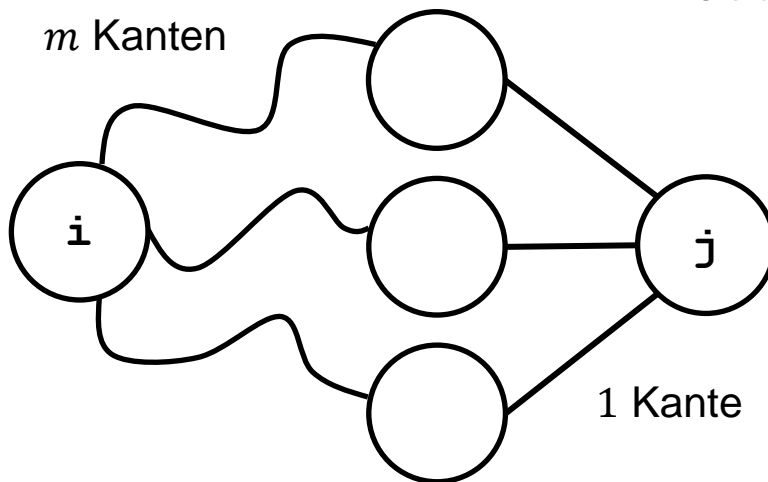
$$A^0 = I = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Nur für  $i = j$  gibt es „den“  
Weg von Knoten  $i$  zu Knoten  $j$   
mit genau 0 Kanten

## Matrix → Eigenschaft (II)

Der Eintrag  $a_{i,j}^{(m)}$  in der  $i$ -ten Zeile und  $j$ -ten Spalte der  $m$ -ten Potenz  $A^m$  der Adjazenzmatrix  $A$  eines Graphen gibt die Anzahl der Wege an, die von Knoten  $i$  zu Knoten  $j$  entlang von genau  $m$  Kanten führen ( $m \geq 0$ ).

Per Induktion (Schritt  $m \rightarrow m + 1$ ):



Jeder Weg mit  $m + 1$  Kanten von  $i$  zu  $j$  führt entlang  $m$  Kanten zu einem Knoten  $k$  und dann mit einer Kante weiter zu  $j$

Dies sind auch alle Wege der Länge  $m + 1$

## Matrix → Eigenschaft (III)

Der Eintrag  $a_{i,j}^{(m)}$  in der  $i$ -ten Zeile und  $j$ -ten Spalte der  $m$ -ten Potenz  $A^m$  der Adjazenzmatrix  $A$  eines Graphen gibt die Anzahl der Wege an, die von Knoten  $i$  zu Knoten  $j$  entlang von genau  $m$  Kanten führen ( $m \geq 0$ ).

Per Induktion (Schritt  $m \rightarrow m + 1$ ):

$$i\text{-te Zeile, } j\text{-te Spalte von } A^{m+1}: \quad a_{i,j}^{(m+1)} = \sum_{k=1}^n a_{i,k}^{(m)} \cdot a_{k,j}$$

$$\begin{aligned} A^{m+1} &= A^m \cdot A \\ &= \begin{pmatrix} a_{i,1}^{(m)} & a_{i,2}^{(m)} & \dots & a_{i,n}^{(m)} \end{pmatrix} \cdot \begin{pmatrix} a_{1,j} \\ a_{2,j} \\ \vdots \\ a_{n,j} \end{pmatrix} \end{aligned}$$

## Matrix → Eigenschaft (IV)

Der Eintrag  $a_{i,j}^{(m)}$  in der  $i$ -ten Zeile und  $j$ -ten Spalte der  $m$ -ten Potenz  $A^m$  der Adjazenzmatrix  $A$  eines Graphen gibt die Anzahl der Wege an, die von Knoten  $i$  zu Knoten  $j$  entlang von genau  $m$  Kanten führen ( $m \geq 0$ ).

Per Induktion (Schritt  $m \rightarrow m + 1$ ):

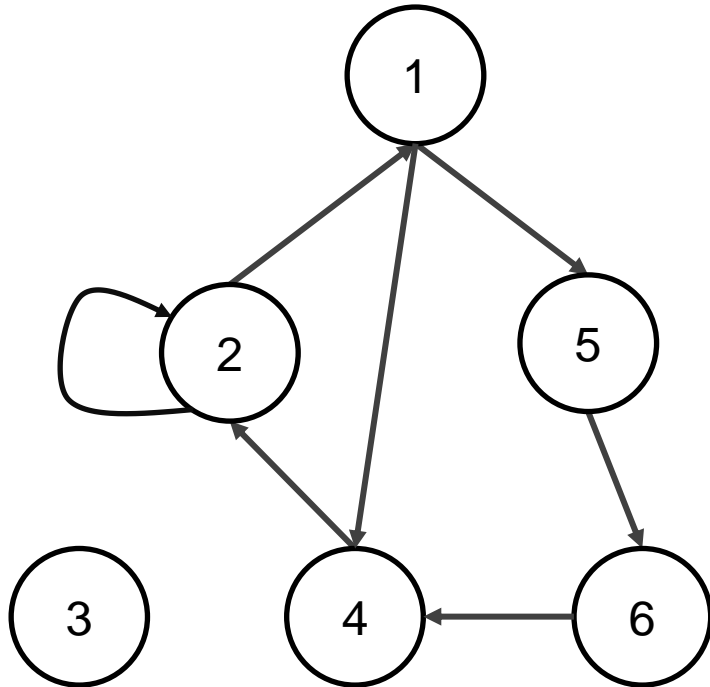
$$i\text{-te Zeile, } j\text{-te Spalte von } A^{m+1}: \quad a_{i,j}^{(m+1)} = \sum_{k=1}^n a_{i,k}^{(m)} \cdot a_{k,j}$$

Anzahl der Wege von  $i$  nach  $k$   
entlang  $m$  Kanten (Induktionsvoraussetzung)

1, wenn es eine Kante von  $k$  zu  $j$  gibt, 0 sonst

Eintrag beschreibt genau Anzahl der Wege  
von  $i$  über alle  $k$  mit jeweils  $m$  Kanten und einer Kante von  $k$  zu  $j$

# Darstellung von Graphen (II)

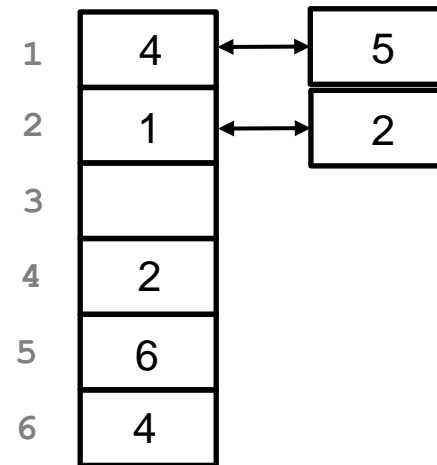


$$V = \{1,2,3,4,5,6\}$$

$$E = \{(1,4), (1,5), (2,1), (2,2), (4,2), (5,6), (6,4)\}$$

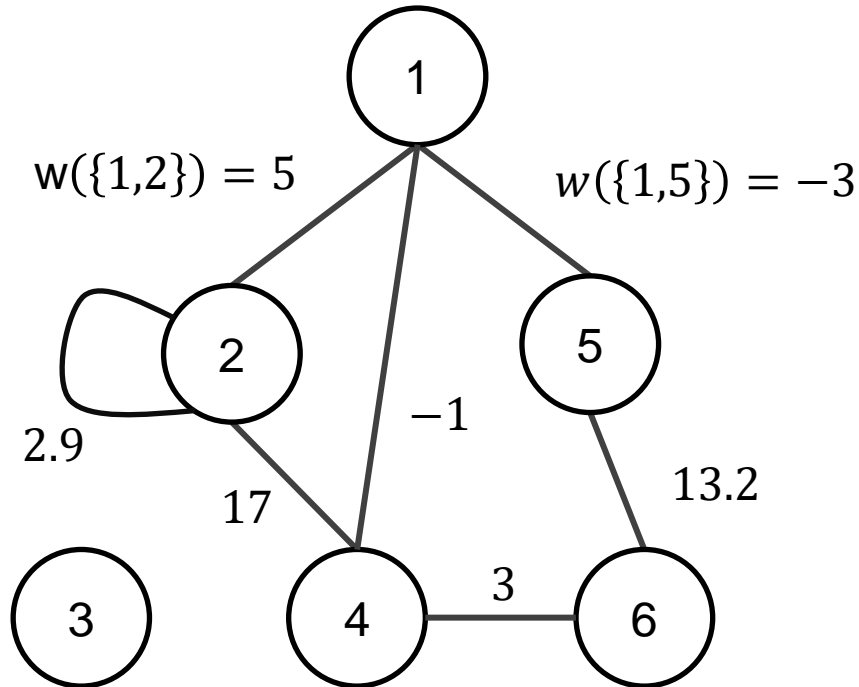
Adjazenzliste:

Als Array mit verketteten Listen  
(sortiert oder unsortiert)



$$\text{Speicherbedarf} = \Theta(|V| + |E|)$$

# Gewichtete Graphen



Beispiel:

Knoten = Städte

Kanten = Zugverbindungen

Gewicht = Entfernung

Ein **gewichteter gerichteter Graph**  $G = (V, E)$  besitzt zusätzlich Funktion

$$w: E \rightarrow \mathbb{R}$$

Bei gewichteten ungerichteten Graphen gilt zusätzlich  
 $w((u, v)) = w((v, u))$   
für alle  $(u, v) \in E$ .

Speicher zusätzlich zu Kante  
 $(u, v)$  auch Wert  $w((u, v))$



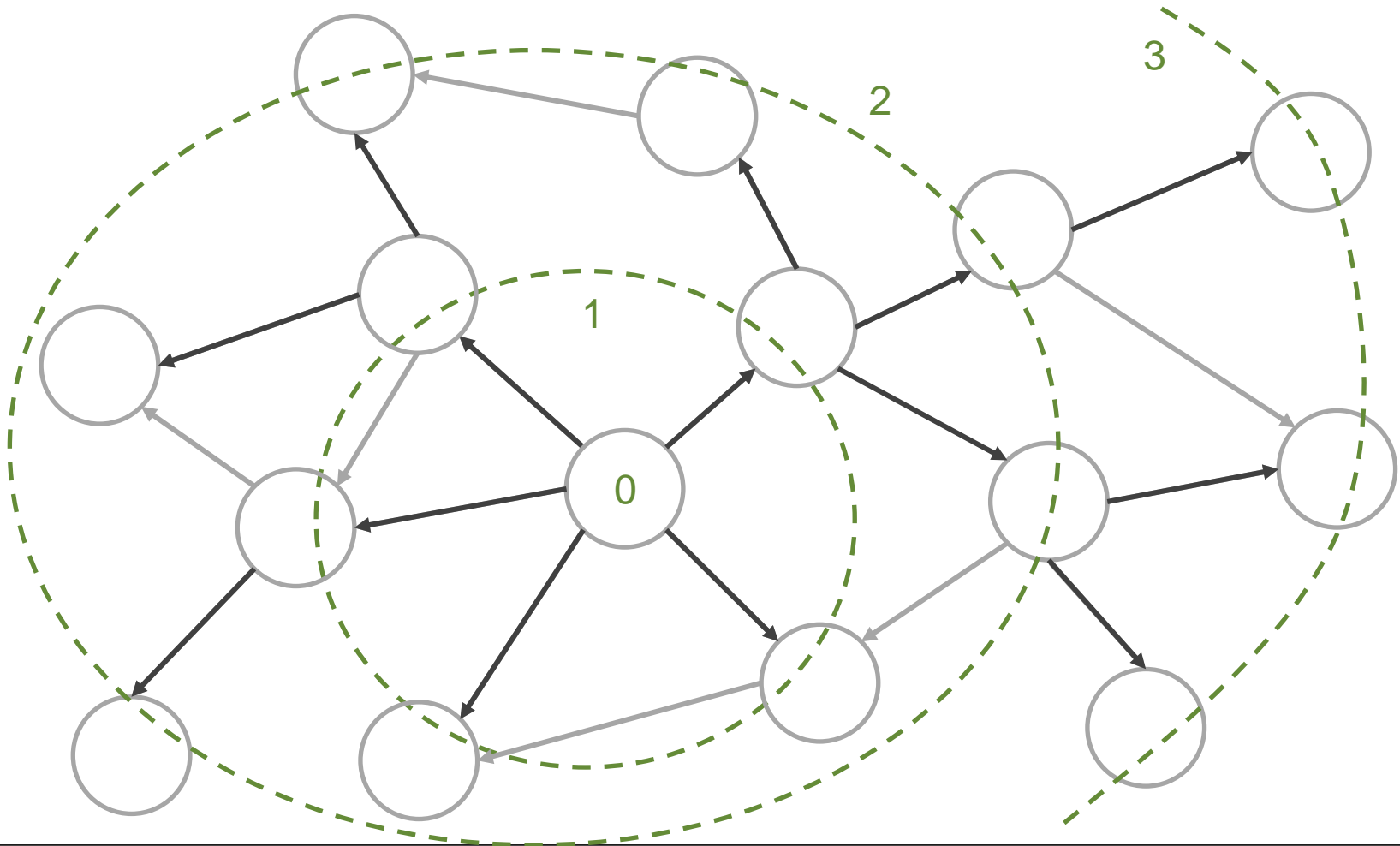
Wie können Sie feststellen, ob es in einem (gerichteten) Graphen einen Weg von einem Knoten  $u$  zu einem Knoten  $v$  gibt?



# Breadth-First Search (BFS)

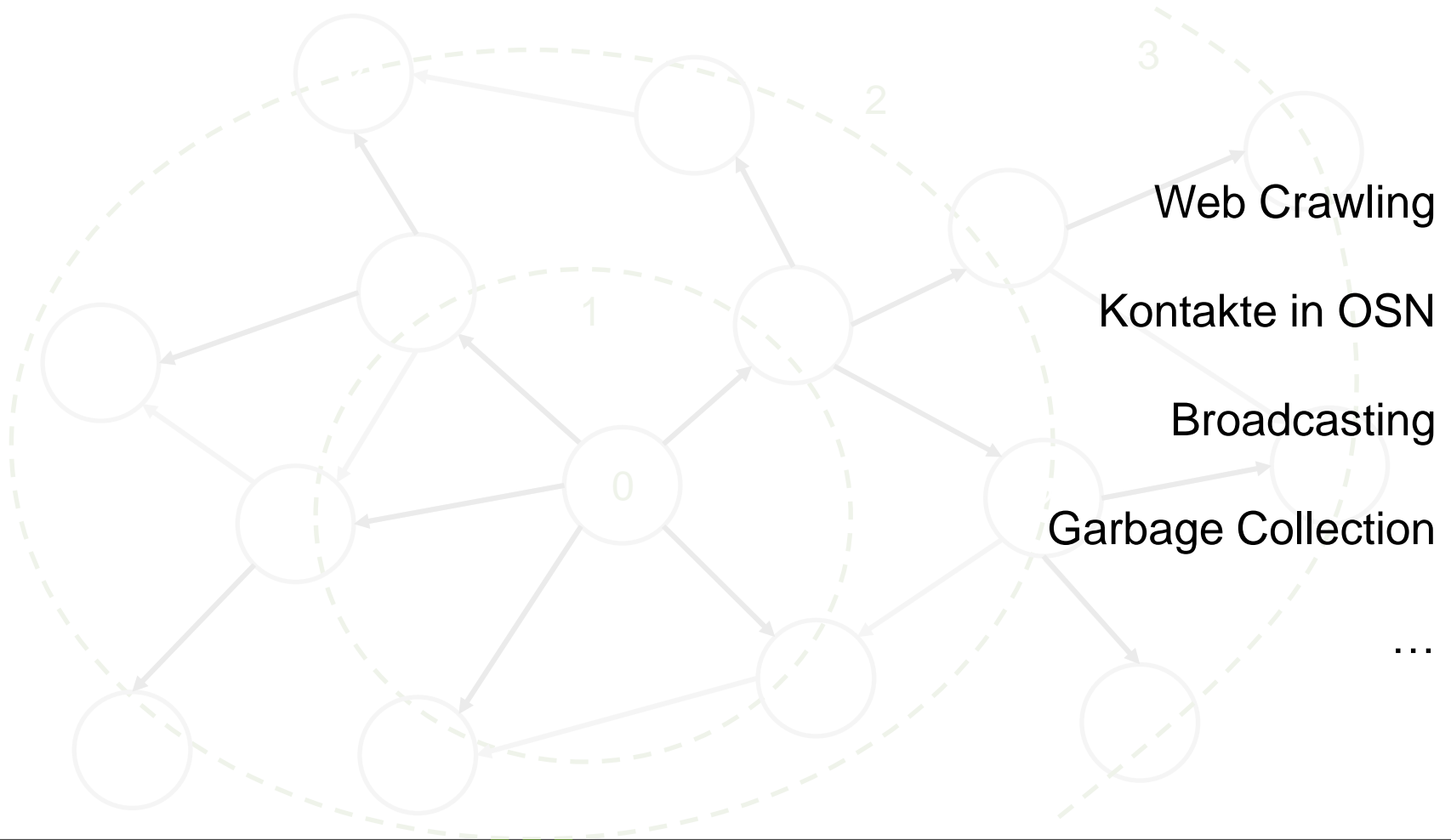
# Idee

Besuche zuerst alle unmittelbaren Nachbarn, dann deren Nachbarn usw.



# Anwendungen

Besuche zuerst alle unmittelbaren Nachbarn, dann deren Nachbarn usw.



# BFS: Algorithmus

**dist** =Distanz von s  
**pred** =Vorgängerknoten

**BFS** (**G**,**s**) // **G**=(**V**,**E**) , **s**=source node in **V**

```
1  FOREACH u in V-{s} DO
2      u.color=WHITE; u.dist=+∞; u.pred=NIL;
3  s.color=GRAY; s.dist=0; s.pred=NIL;
4  newQueue(Q);
5  enqueue(Q,s);
6  WHILE !isEmpty(Q) DO
7      u=dequeue(Q);
8      FOREACH v in adj(G,u) DO
9          IF v.color==WHITE THEN
10             v.color=GRAY; v.dist=u.dist+1; v.pred=u;
11             enqueue(Q,v);
12  u.color=BLACK;
```

**WHITE** =Knoten noch nicht besucht  
**GRAY**=in Queue für nächsten Schritt  
**BLACK** =fertig

**adj** (**G**,**u**) = Liste aller Knoten  $v \in V$  mit  $(u, v) \in E$   
(Reihenfolge irrelevant)

# BFS: Algorithmus (Beispiel I)

```
BFS(G,s) //G=(V,E), s=source node in V
```

```
...
```

```
6 WHILE !isEmpty(Q) DO
```

```
7   u=dequeue(Q);
```

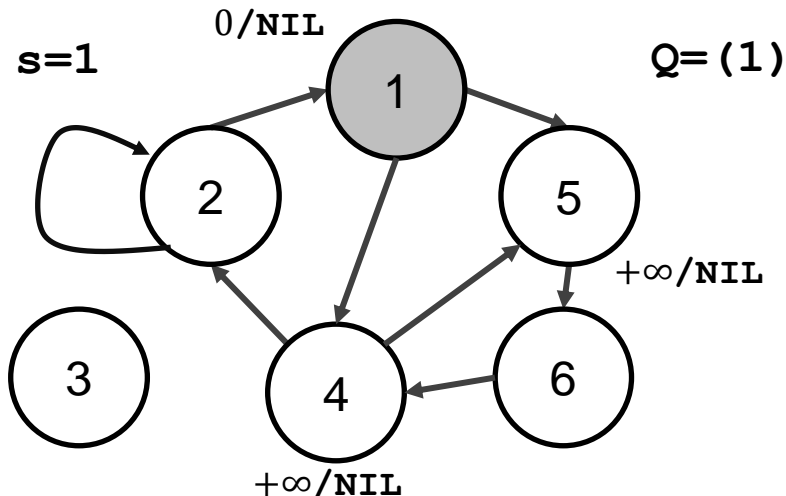
```
8   FOREACH v in adj(G,u) DO
```

```
9     IF v.color==WHITE THEN
```

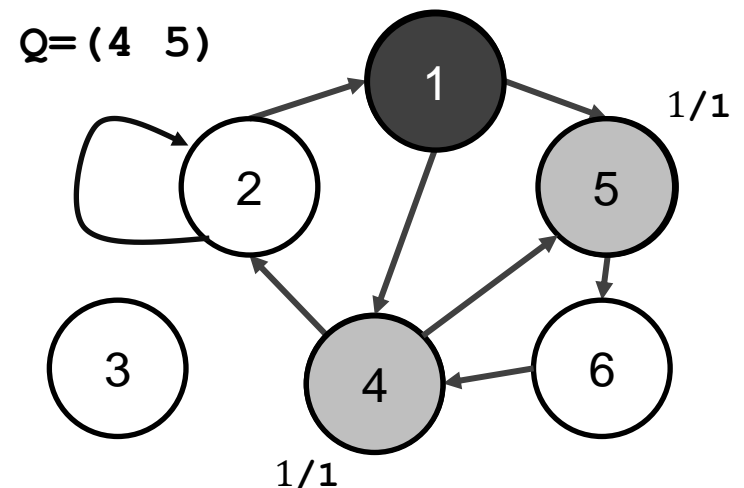
```
10      v.color=GRAY; v.dist=u.dist+1; v.pred=u;
```

```
11      enqueue(Q,v);
```

```
12   u.color=BLACK;
```



7-12



## BFS: Algorithmus (Beispiel II)

```
BFS(G,s) //G=(V,E), s=source node in V
```

```
...
```

```
6 WHILE !isEmpty(Q) DO
```

```
7   u=dequeue(Q);
```

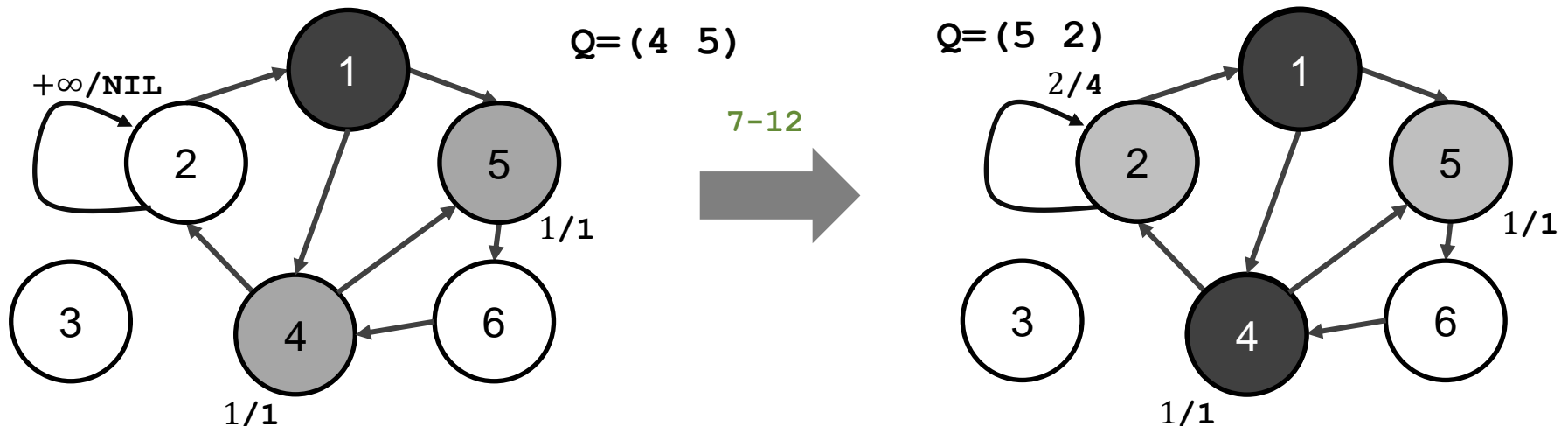
```
8   FOREACH v in adj(G,u) DO
```

```
9     IF v.color==WHITE THEN
```

```
10      v.color=GRAY; v.dist=u.dist+1; v.pred=u;
```

```
11      enqueue(Q,v);
```

```
12   u.color=BLACK;
```



# BFS: Algorithmus (Beispiel III)

```
BFS(G,s) //G=(V,E), s=source node in V
```

```
...
```

```
6 WHILE !isEmpty(Q) DO
```

```
7   u=dequeue(Q);
```

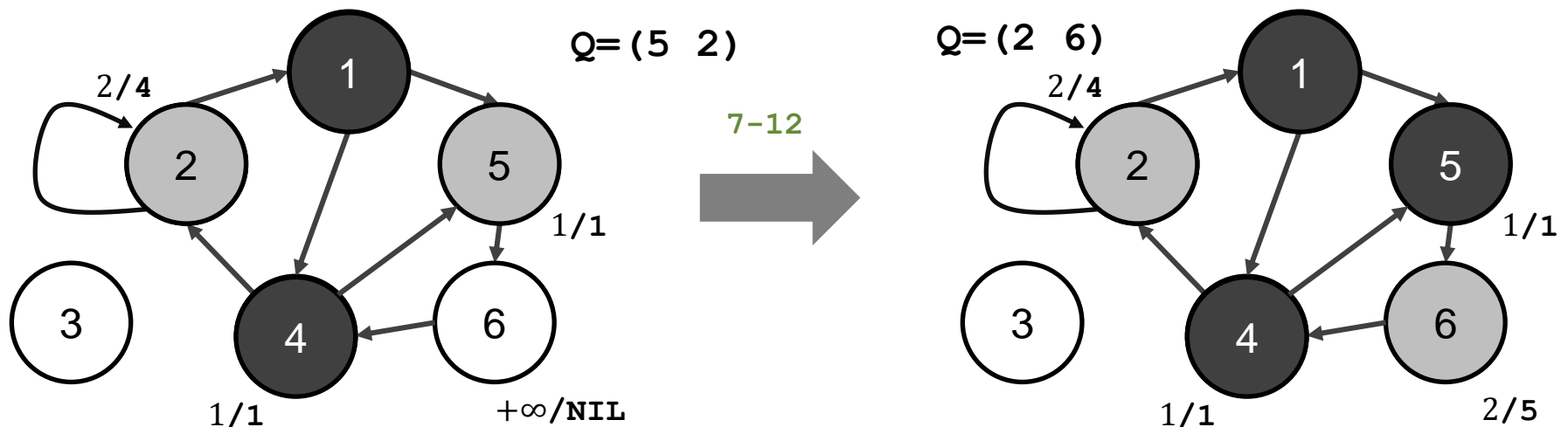
```
8   FOREACH v in adj(G,u) DO
```

```
9     IF v.color==WHITE THEN
```

```
10      v.color=GRAY; v.dist=u.dist+1; v.pred=u;
```

```
11      enqueue(Q,v);
```

```
12   u.color=BLACK;
```



# BFS: Algorithmus (Beispiel IV)

```
BFS(G,s) //G=(V,E), s=source node in V
```

```
...
```

```
6 WHILE !isEmpty(Q) DO
```

```
7   u=dequeue(Q);
```

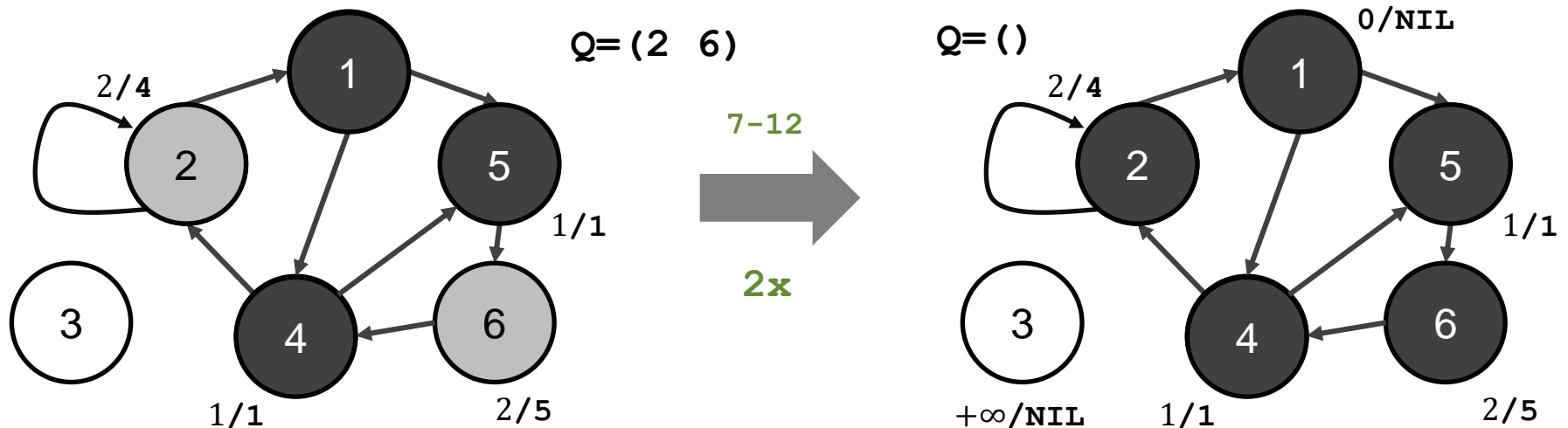
```
8   FOREACH v in adj(G,u) DO
```

```
9     IF v.color==WHITE THEN
```

```
10      v.color=GRAY; v.dist=u.dist+1; v.pred=u;
```

```
11      enqueue(Q,v);
```

```
12   u.color=BLACK;
```





# BFS: Algorithmus - Laufzeit

$$\text{Laufzeit} = O(|V| + |E|)$$

BFS( $G, s$ ) //  $G=(V, E)$ ,  $s$ =source node in  $V$

Schleife mit  $O(|V|)$  vielen Iterationen

```
1  FOREACH  $u$  in  $V - \{s\}$  DO
2       $u.color = \text{WHITE}$ ;  $u.dist = +\infty$ ;  $u.pred = \text{NIL}$ ;
3   $s.color = \text{GRAY}$ ;  $s.dist = 0$ ;  $s.pred = \text{NIL}$ ;
4  newQueue( $Q$ );
5  enqueue( $Q, s$ );
6  WHILE !isEmpty( $Q$ ) DO
7       $u = \text{dequeue}(Q)$ ;
8      FOREACH  $v$  in adj( $G, u$ ) DO
9          IF  $v.color == \text{WHITE}$  THEN
10              $v.color = \text{GRAY}$ ;  $v.dist = u.dist + 1$ ;  $v.pred = u$ ;
11             enqueue( $Q, v$ );
12      $u.color = \text{BLACK}$ ;
```

Jeder Knoten wird maximal einmal in  $Q$  aufgenommen ( $s$  oder **WHITE**)  
(bzw. gar nicht, wenn nicht erreichbar)

Insgesamt werden maximal  $\sum_{u \in V} |adj(G, u)| = O(|E|)$  Kanten betrachtet

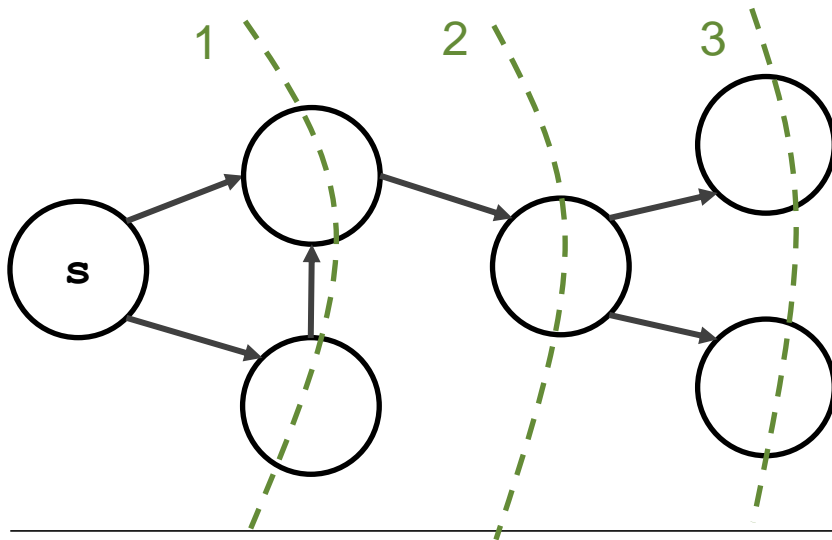
# Korrektheit (I)

Teil 1: **dist**=Länge des kürzesten Pfades

Teil 2: Pfad kann abgelesen werden

Sei  $G = (V, E)$  gerichteter oder ungerichteter Graph mit Knoten  $s \in V$ . Dann gilt nach Terminierung von **BFS** ( $G, s$ ) für jeden von  $s$  aus erreichbaren Knoten  $v$ , dass  $\text{shortest}(s, v) = v.\text{dist}$ .

Für  $v \neq s$  ist ein kürzester Pfad durch einen kürzesten Pfad von  $s$  nach  $v.\text{pred}$  und der Kante  $(v.\text{pred}, v)$  gegeben.



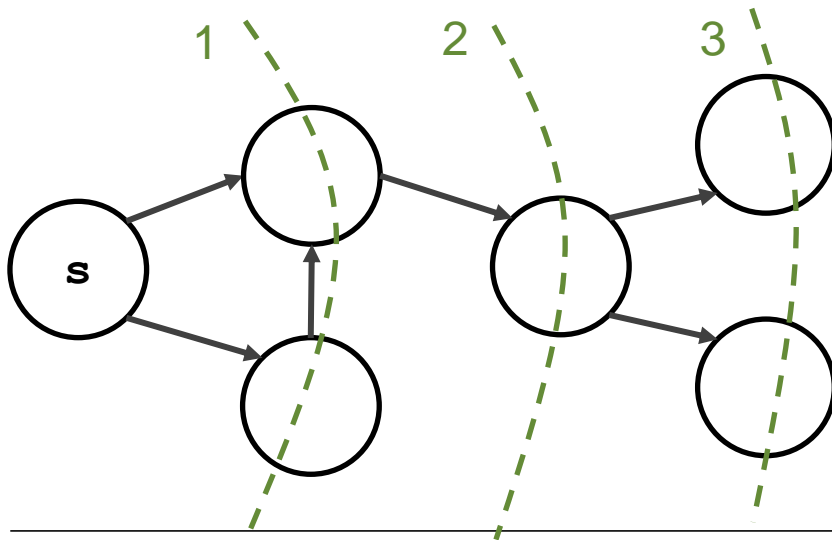
Intuition für Korrektheit **dist**:

Im ersten Schritt werden genau die Knoten besucht, die von  $s$  aus über eine Kante erreicht werden können; diese Knoten erhalten **dist=1**

## Korrektheit (II)

Sei  $G = (V, E)$  gerichteter oder ungerichteter Graph mit Knoten  $s \in V$ . Dann gilt nach Terminierung von **BFS** ( $G, s$ ) für jeden von  $s$  aus erreichbaren Knoten  $v$ , dass  $\text{shortest}(s, v) = v.\text{dist}$ .

Für  $v \neq s$  ist ein kürzester Pfad durch einen kürzesten Pfad von  $s$  nach  $v.\text{pred}$  und der Kante  $(v.\text{pred}, v)$  gegeben.



Intuition für Korrektheit **dist**:

Im zweiten Schritt werden nur die Knoten besucht die in zwei oder mehr Schritten von  $s$  aus erreichbar sind; diese erhalten **dist=2** usw.

## Korrektheit (III)

Sei  $G = (V, E)$  gerichteter oder ungerichteter Graph mit Knoten  $s \in V$ . Dann gilt nach Terminierung von **BFS** ( $G, s$ ) für jeden von  $s$  aus erreichbaren Knoten  $v$ , dass  $shortest(s, v) = v.dist$ .

Für  $v \neq s$  ist ein kürzester Pfad durch einen kürzesten Pfad von  $s$  nach  $v.pred$  und der Kante  $(v.pred, v)$  gegeben.

Korrektheit kürzester Pfad:

Für  $u = v.pred$  ist, da  $v$  von  $u$  aus per Kante  $(u, v) \in E$  besucht wurde, der Pfad von  $s$  nach  $u$  und dann zu  $v$  ein Pfad der Länge

$$1 + shortest(s, u) = 1 + u.dist = v.dist = shortest(s, v)$$

gemäß  
Teil 1

gemäß  
Algorithmus

gemäß  
Teil 1

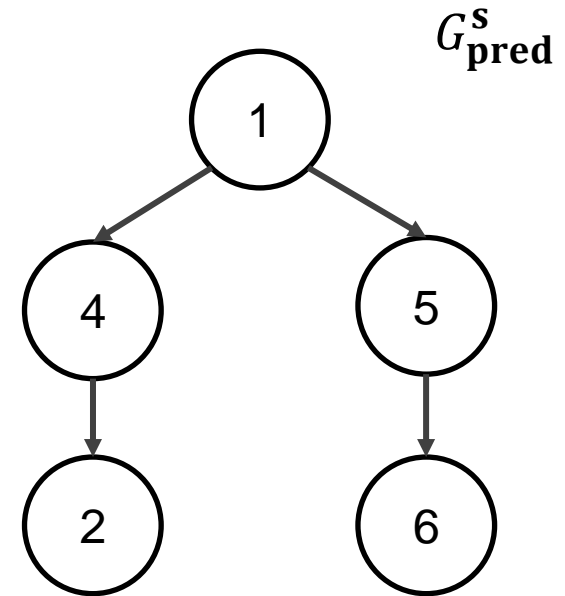
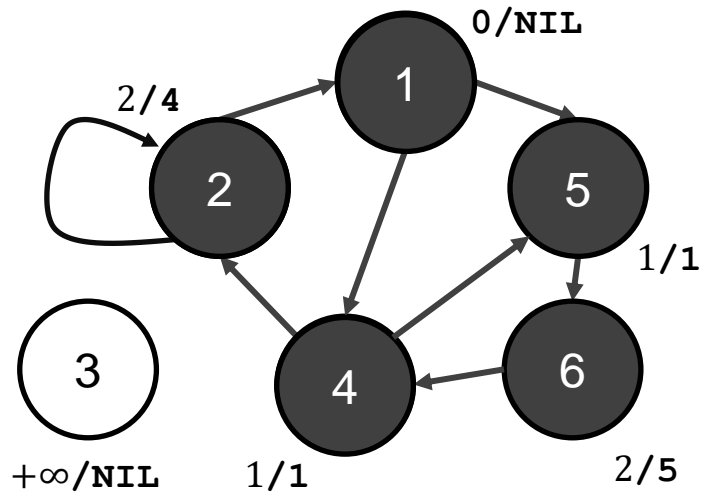
# Kürzeste Pfade ausgeben

Laufzeit\* =  $O(|V|)$   
\*ohne BFS

```
PRINT-PATH (G, s, v)
//assumes that BFS (G, s) has already been executed

1  IF v==s THEN
2      PRINT s
3  ELSE
4      IF v.pred==NIL THEN
5          PRINT 'no path from s to v'
6      ELSE
7          PRINT-PATH (G, s, v.pred) ;
8          PRINT v;
```

# Abgeleiteter BFS-Baum (I)



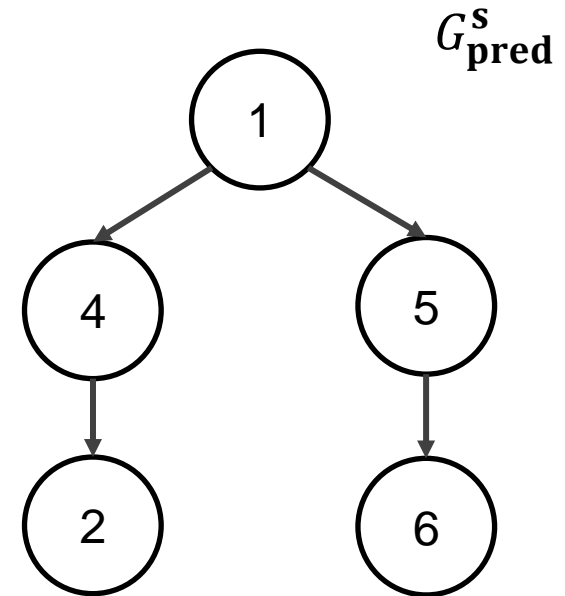
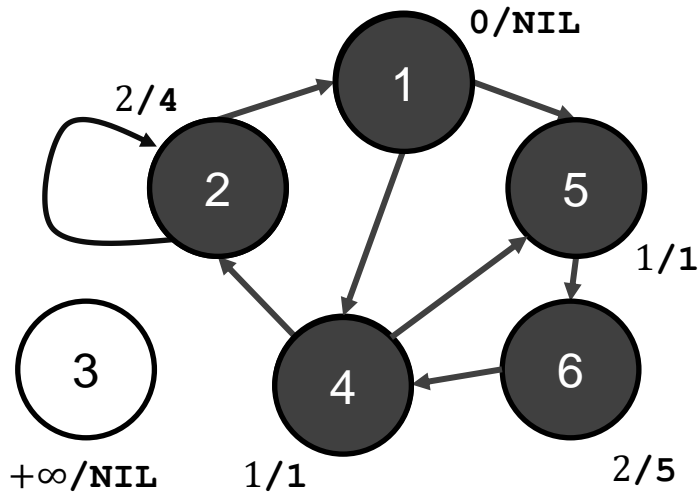
Definiere Subgraph  $G_{\text{pred}}^s = (V_{\text{pred}}^s, E_{\text{pred}}^s)$  von  $G$  durch:

$$V_{\text{pred}}^s = \{ \mathbf{v} \in V \mid \mathbf{v}.\text{pred} \neq \text{NIL} \} \cup \{ \mathbf{s} \}$$

$$E_{\text{pred}}^s = \{ (\mathbf{v}.\text{pred}, \mathbf{v}) \mid \mathbf{v} \in V_{\text{pred}}^s - \{ \mathbf{s} \} \}$$

(und  $(\mathbf{v}, \mathbf{v}.\text{pred})$  für ungerichtete Graphen)

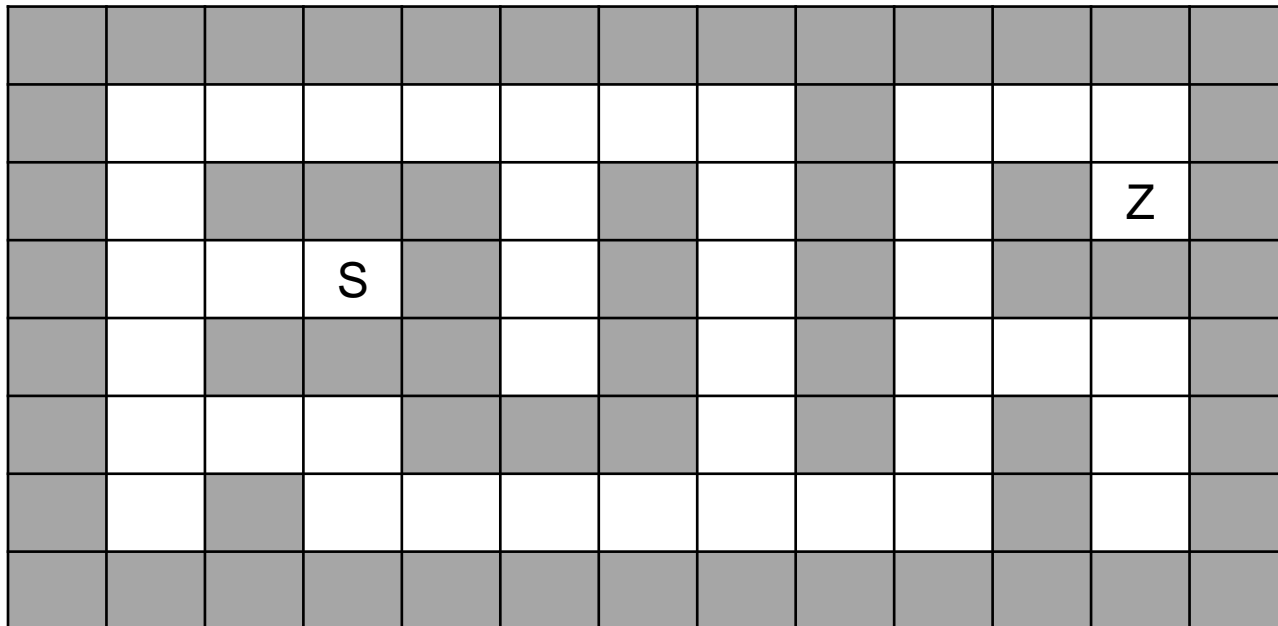
## Abgeleiteter BFS-Baum (II)



$G_{\text{pred}}^s$  ist BFS-Baum zu  $G$ ,  
d.h. enthält alle von  $s$  aus erreichbaren Knoten in  $G$  und  
für jeden Knoten  $v \in V_{\text{pred}}^s$  existiert genau ein Pfad von  $s$   
in  $G_{\text{pred}}^s$ , der auch ein kürzester Pfad von  $s$  zu  $v$  in  $G$  ist.



Überlegen Sie sich, wie Sie mittels BFS in einem Labyrinth der folgenden Form den kürzesten Weg von Start (S) zu Ziel (Z) finden können. Welche Laufzeit hat Ihr Verfahren?



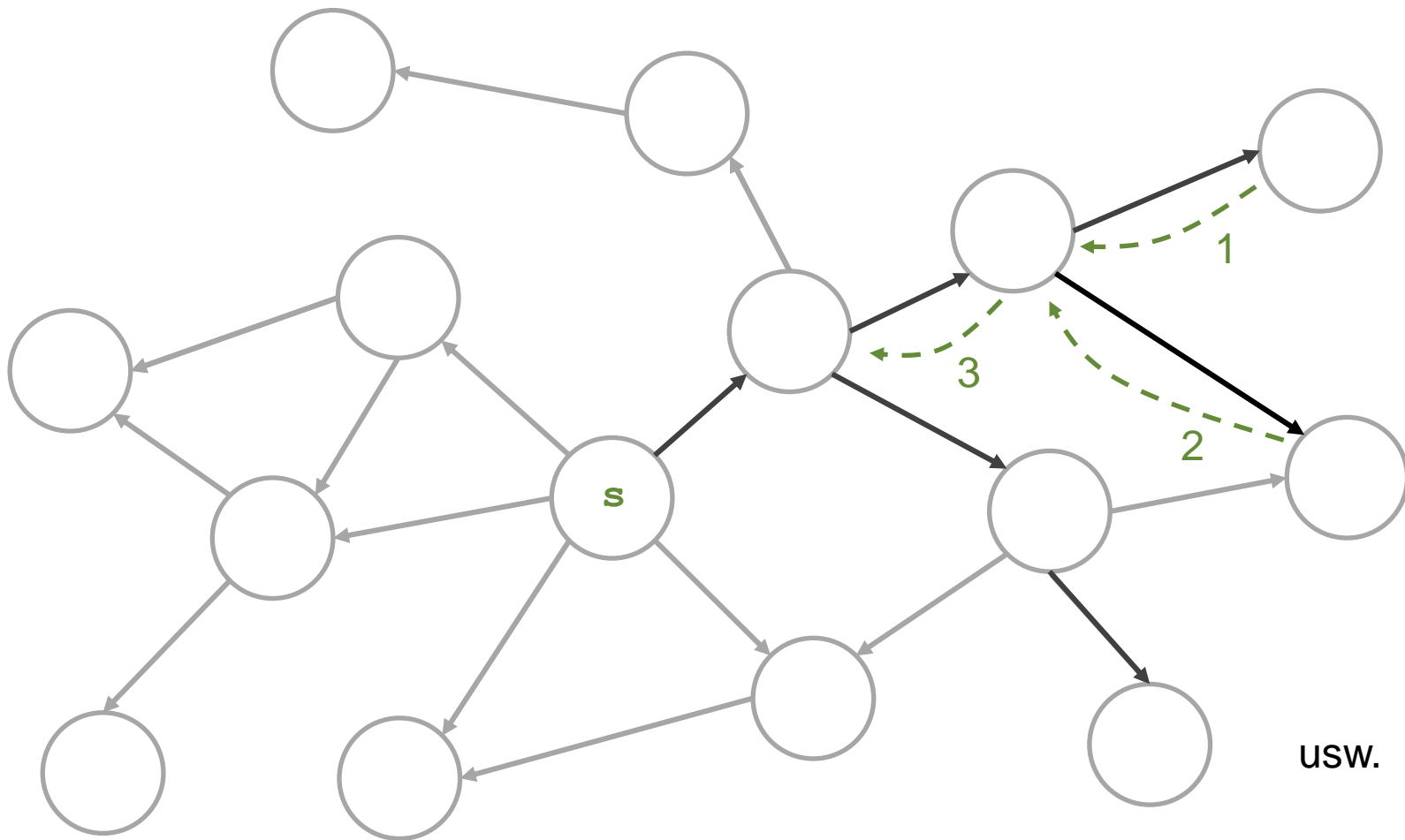
Hinweis: Sie sind selbst nicht im Labyrinth, sondern „schauen von oben darauf“



# Depth-First Search (DFS)

## Idee

Besuche zuerst alle noch nicht besuchten Nachfolgeknoten („Laufe so weit wie möglich weg von aktuellem Knoten“)



USW.

# DFS: Algorithmus

Laufzeit =  $O(|V| + |E|)$

time globale Variable

## DFS-VISIT (G, u)

```
1  time=time+1;
2  u.disc=time;
3  u.color=GRAY;
4  FOREACH v in adj(G,u) DO
5      IF v.color==WHITE THEN
6          v.pred=u;
7          DFS-VISIT (G, v) ;
8  u.color=BLACK;
9  time=time+1;
10 u.finish=time;
```

## DFS (G) //G=(V,E)

```
1  FOREACH u in V DO
2      u.color=WHITE;
3      u.pred=NIL;
4  time=0;
5  FOREACH u in V DO
6      IF u.color==WHITE THEN
7          DFS-VISIT (G, u)
```

disc = discovery time  
finish=finish time

# DFS: Beispiel (I)

Aufrufs-Stack: **DFS-VISIT**(G,1)

↓↑

DFS-VISIT(G,u)

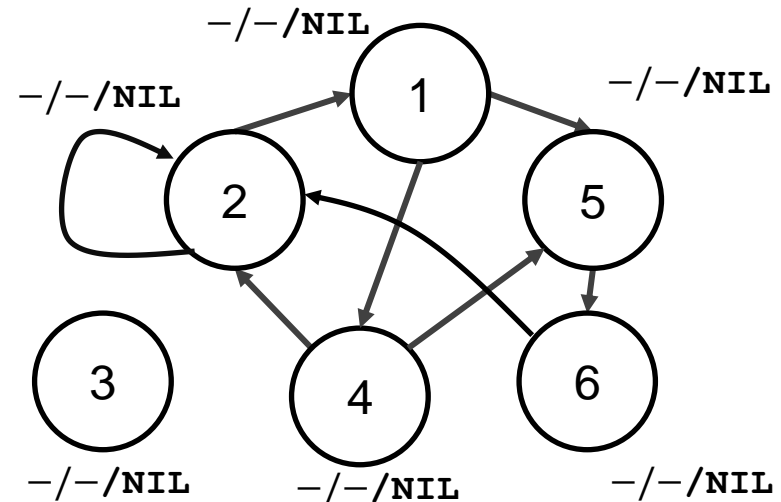
time=0

```
1  time=time+1;
2  u.disc=time;
3  u.color=GRAY;
4  FOREACH v in adj(G,u) DO
5      IF v.color==WHITE THEN
6          v.pred=u;
7          DFS-VISIT(G,v);
8  u.color=BLACK;
9  time=time+1;
10 u.finish=time;
```

Ordnung auf Knotenlisten gemäß Knotennummern

DFS(G) //G=(V,E)

```
1  FOREACH u in V DO
2      u.color=WHITE;
3      u.pred=NIL;
4  time=0;
5  FOREACH u in V DO
6      IF u.color==WHITE THEN
7          DFS-VISIT(G,u)
```



# DFS: Beispiel (II)

Aufrufs-Stack: **DFS-VISIT (G, 1)**  
                  ↓↑  
                  **DFS-VISIT (G, 4)**

DFS-VISIT (G, u)

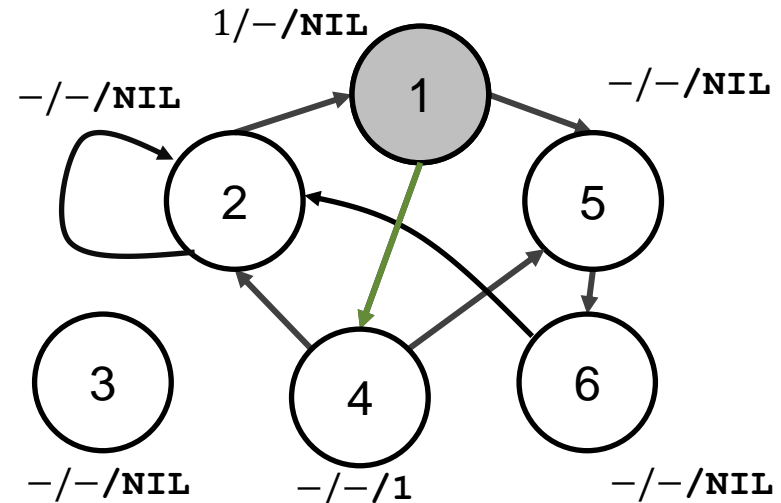
time=1

```
1  time=time+1;
2  u.disc=time;
3  u.color=GRAY;
4  FOREACH v in adj(G,u) DO
5      IF v.color==WHITE THEN
6          v.pred=u;
7          DFS-VISIT (G, v) ;
8  u.color=BLACK;
9  time=time+1;
10 u.finish=time;
```

Ordnung auf Knotenlisten gemäß Knotennummern

DFS (G) //G= (V, E)

```
1  FOREACH u in V DO
2      u.color=WHITE;
3      u.pred=NIL;
4  time=0;
5  FOREACH u in V DO
6      IF u.color==WHITE THEN
7          DFS-VISIT (G, u)
```



# DFS: Beispiel (II)

Aufrufs-Stack: **DFS-VISIT (G, 1)**  
                  ↓↑  
                  **DFS-VISIT (G, 4)**  
                  **DFS-VISIT (G, 2)**

**DFS-VISIT (G, u)**

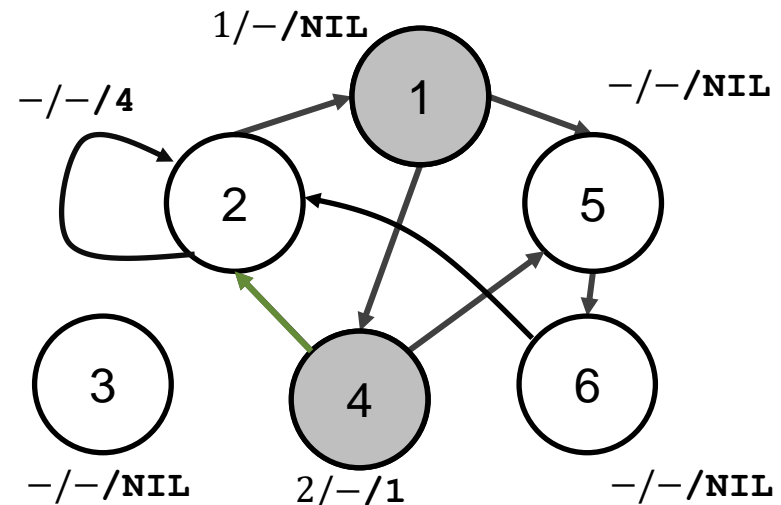
time=2

```
1  time=time+1;
2  u.disc=time;
3  u.color=GRAY;
4  FOREACH v in adj(G,u) DO
5      IF v.color==WHITE THEN
6          v.pred=u;
7          DFS-VISIT (G, v) ;
8  u.color=BLACK;
9  time=time+1;
10 u.finish=time;
```

Ordnung auf Knotenlisten gemäß Knotennummern

**DFS (G) //G=(V,E)**

```
1  FOREACH u in V DO
2      u.color=WHITE;
3      u.pred=NIL;
4  time=0;
5  FOREACH u in V DO
6      IF u.color==WHITE THEN
7          DFS-VISIT (G, u)
```



# DFS: Beispiel (III)

Aufrufs-Stack: **DFS-VISIT (G, 1)**  
                  ↓↑  
                  **DFS-VISIT (G, 4)**  
                  **DFS-VISIT (G, 2)**

DFS-VISIT (G, u)

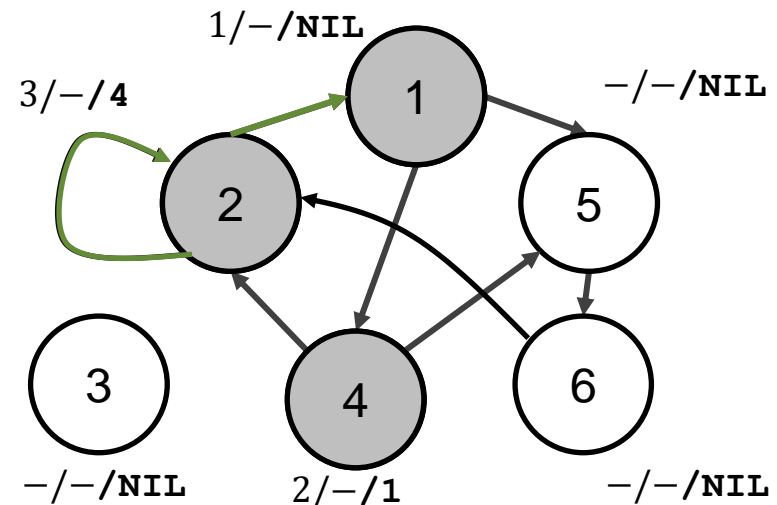
time=3

```
1  time=time+1;
2  u.disc=time;
3  u.color=GRAY;
4  FOREACH v in adj(G,u) DO
5      IF v.color==WHITE THEN
6          v.pred=u;
7          DFS-VISIT (G, v) ;
8  u.color=BLACK;
9  time=time+1;
10 u.finish=time;
```

Ordnung auf Knotenlisten gemäß Knotennummern

DFS (G) //G= (V, E)

```
1  FOREACH u in V DO
2      u.color=WHITE;
3      u.pred=NIL;
4  time=0;
5  FOREACH u in V DO
6      IF u.color==WHITE THEN
7          DFS-VISIT (G, u)
```



# DFS: Beispiel (IV)

Aufrufs-Stack: **DFS-VISIT (G, 1)**  
                  ↓↑  
                  **DFS-VISIT (G, 4)**

DFS-VISIT (G, u)

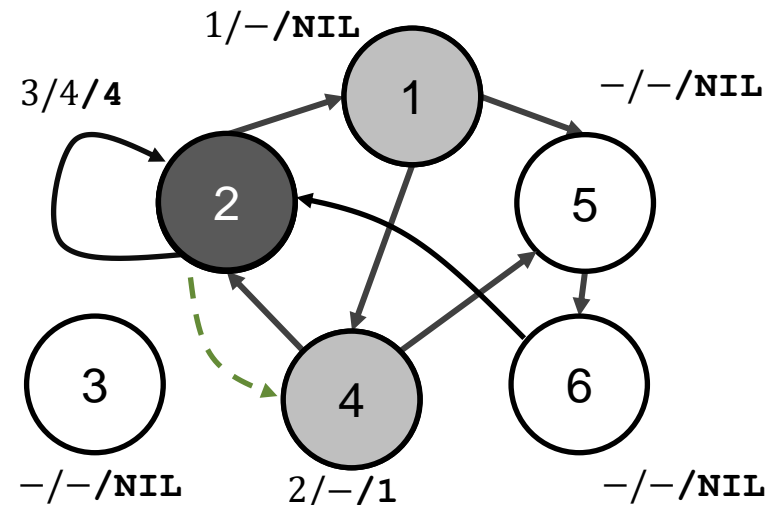
time=4

```
1  time=time+1;
2  u.disc=time;
3  u.color=GRAY;
4  FOREACH v in adj(G,u) DO
5      IF v.color==WHITE THEN
6          v.pred=u;
7          DFS-VISIT (G, v) ;
8  u.color=BLACK;
9  time=time+1;
10 u.finish=time;
```

Ordnung auf Knotenlisten gemäß Knotennummern

DFS (G) //G= (V, E)

```
1  FOREACH u in V DO
2      u.color=WHITE;
3      u.pred=NIL;
4  time=0;
5  FOREACH u in V DO
6      IF u.color==WHITE THEN
7          DFS-VISIT (G, u)
```





# DFS: Beispiel (V)

Aufrufs-Stack: **DFS-VISIT (G, 1)**  
                  ↓↑  
                  **DFS-VISIT (G, 4)**  
                  **DFS-VISIT (G, 5)**

**DFS-VISIT (G, u)**

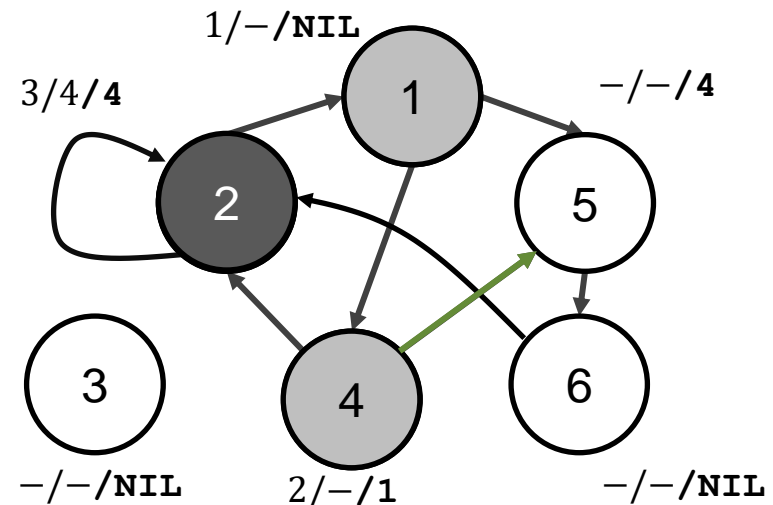
time=4

```
1  time=time+1;
2  u.disc=time;
3  u.color=GRAY;
4  FOREACH v in adj(G,u) DO
5      IF v.color==WHITE THEN
6          v.pred=u;
7          DFS-VISIT (G, v) ;
8  u.color=BLACK;
9  time=time+1;
10 u.finish=time;
```

Ordnung auf Knotenlisten gemäß Knotennummern

**DFS (G) //G= (V, E)**

```
1  FOREACH u in V DO
2      u.color=WHITE;
3      u.pred=NIL;
4  time=0;
5  FOREACH u in V DO
6      IF u.color==WHITE THEN
7          DFS-VISIT (G, u)
```



# DFS: Beispiel (VI)

Aufrufs-Stack: **DFS-VISIT (G, 1)**  
                  ↓↑  
                  **DFS-VISIT (G, 4)**  
                  **DFS-VISIT (G, 5)**  
                  **DFS-VISIT (G, 6)**

**DFS-VISIT (G, u)**

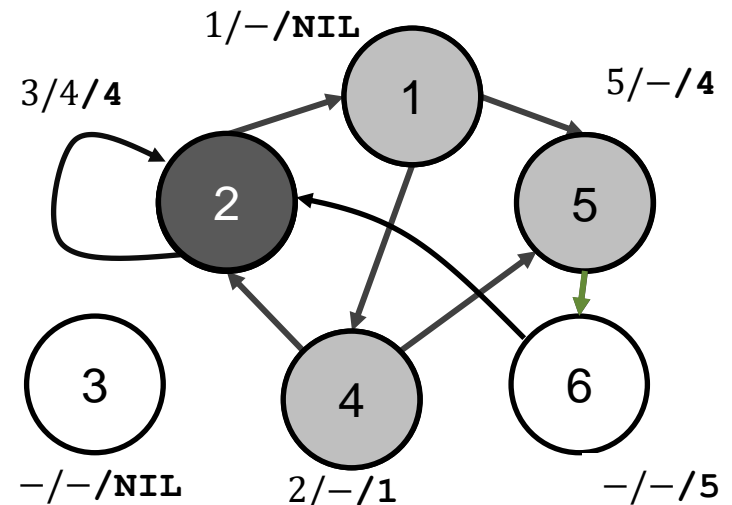
time=5

```
1  time=time+1;
2  u.disc=time;
3  u.color=GRAY;
4  FOREACH v in adj(G,u) DO
5      IF v.color==WHITE THEN
6          v.pred=u;
7          DFS-VISIT (G, v) ;
8  u.color=BLACK;
9  time=time+1;
10 u.finish=time;
```

Ordnung auf Knotenlisten gemäß Knotennummern

**DFS (G) //G=(V,E)**

```
1  FOREACH u in V DO
2      u.color=WHITE;
3      u.pred=NIL;
4  time=0;
5  FOREACH u in V DO
6      IF u.color==WHITE THEN
7          DFS-VISIT (G, u)
```



# DFS: Beispiel (VII)

Aufrufs-Stack: **DFS-VISIT (G, 1)**  
                  ↓↑  
                  **DFS-VISIT (G, 4)**  
                  **DFS-VISIT (G, 5)**  
                  **DFS-VISIT (G, 6)**

**DFS-VISIT (G, u)**

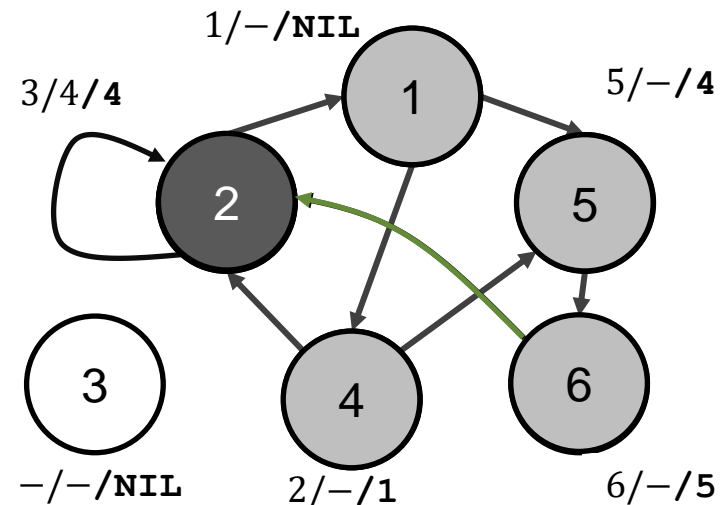
time=6

```
1  time=time+1;
2  u.disc=time;
3  u.color=GRAY;
4  FOREACH v in adj(G,u) DO
5      IF v.color==WHITE THEN
6          v.pred=u;
7          DFS-VISIT (G, v) ;
8  u.color=BLACK;
9  time=time+1;
10 u.finish=time;
```

Ordnung auf Knotenlisten gemäß Knotennummern

**DFS (G) //G=(V,E)**

```
1  FOREACH u in V DO
2      u.color=WHITE;
3      u.pred=NIL;
4  time=0;
5  FOREACH u in V DO
6      IF u.color==WHITE THEN
7          DFS-VISIT (G, u)
```



# DFS: Beispiel (VIII)

Aufrufs-Stack: **DFS-VISIT (G, 1)**  
                  ↓↑ **DFS-VISIT (G, 4)**  
                          **DFS-VISIT (G, 5)**

DFS-VISIT (G, u)

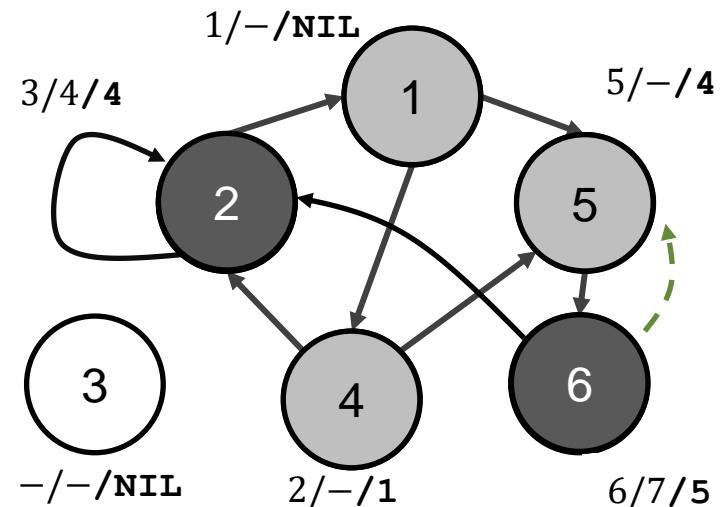
time=7

```
1  time=time+1;
2  u.disc=time;
3  u.color=GRAY;
4  FOREACH v in adj(G,u) DO
5      IF v.color==WHITE THEN
6          v.pred=u;
7          DFS-VISIT (G, v) ;
8  u.color=BLACK;
9  time=time+1;
10 u.finish=time;
```

Ordnung auf Knotenlisten gemäß Knotennummern

DFS (G) //G= (V, E)

```
1  FOREACH u in V DO
2      u.color=WHITE;
3      u.pred=NIL;
4  time=0;
5  FOREACH u in V DO
6      IF u.color==WHITE THEN
7          DFS-VISIT (G, u)
```



# DFS: Beispiel (IX)

Aufrufs-Stack: **DFS-VISIT (G, 1)**  
                  ↓↑  
                  **DFS-VISIT (G, 4)**

DFS-VISIT (G, u)

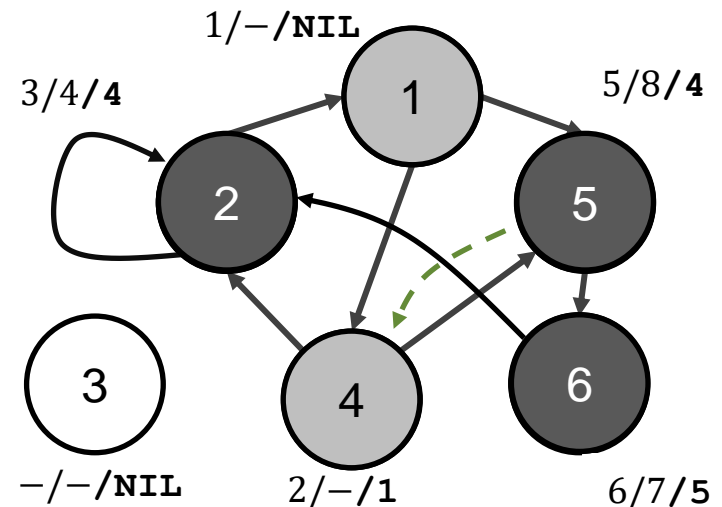
time=8

```
1  time=time+1;
2  u.disc=time;
3  u.color=GRAY;
4  FOREACH v in adj(G,u) DO
5      IF v.color==WHITE THEN
6          v.pred=u;
7          DFS-VISIT (G, v) ;
8  u.color=BLACK;
9  time=time+1;
10 u.finish=time;
```

Ordnung auf Knotenlisten gemäß Knotennummern

DFS (G) //G= (V, E)

```
1  FOREACH u in V DO
2      u.color=WHITE;
3      u.pred=NIL;
4  time=0;
5  FOREACH u in V DO
6      IF u.color==WHITE THEN
7          DFS-VISIT (G, u)
```



# DFS: Beispiel (X)

Aufrufs-Stack: **DFS-VISIT (G, 1)**

↓↑

DFS-VISIT (G, u)

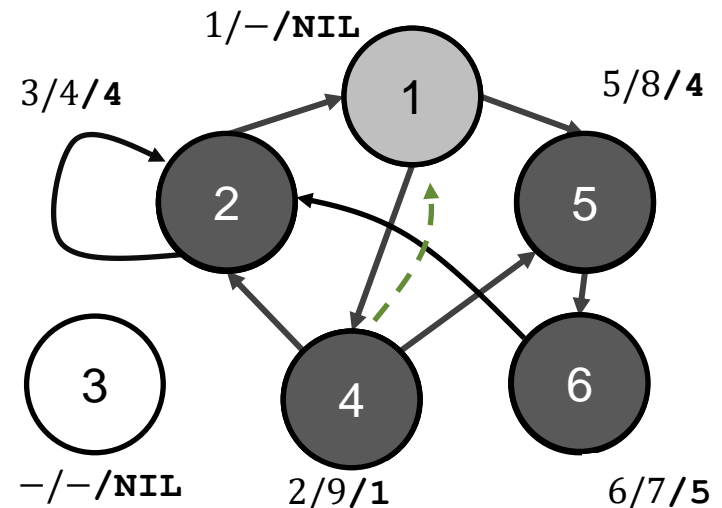
time=9

```
1  time=time+1;
2  u.disc=time;
3  u.color=GRAY;
4  FOREACH v in adj(G,u) DO
5      IF v.color==WHITE THEN
6          v.pred=u;
7          DFS-VISIT (G, v) ;
8  u.color=BLACK;
9  time=time+1;
10 u.finish=time;
```

Ordnung auf Knotenlisten gemäß Knotennummern

DFS (G) //G= (V, E)

```
1  FOREACH u in V DO
2      u.color=WHITE;
3      u.pred=NIL;
4  time=0;
5  FOREACH u in V DO
6      IF u.color==WHITE THEN
7          DFS-VISIT (G, u)
```



# DFS: Beispiel (X)

Aufrufs-Stack:

↓↑

DFS-VISIT (G, u)

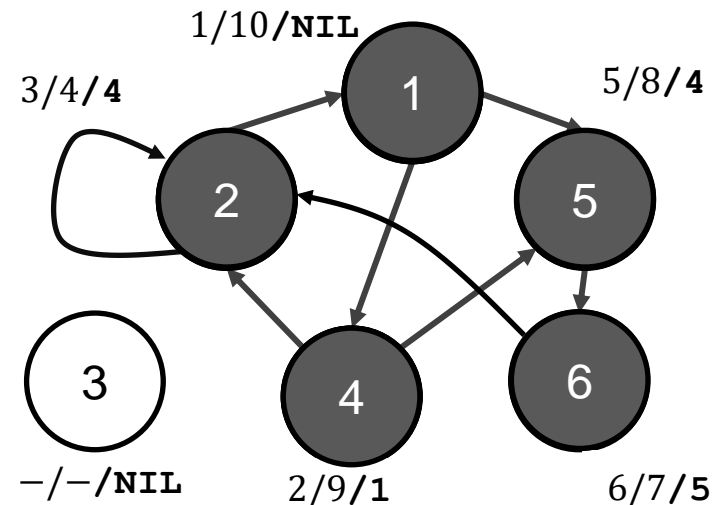
time=10

```
1  time=time+1;
2  u.disc=time;
3  u.color=GRAY;
4  FOREACH v in adj(G,u) DO
5      IF v.color==WHITE THEN
6          v.pred=u;
7          DFS-VISIT (G, v) ;
8  u.color=BLACK;
9  time=time+1;
10 u.finish=time;
```

Ordnung auf Knotenlisten gemäß Knotennummern

DFS (G) //G=(V,E)

```
1  FOREACH u in V DO
2      u.color=WHITE;
3      u.pred=NIL;
4  time=0;
5  FOREACH u in V DO
6      IF u.color==WHITE THEN
7          DFS-VISIT (G, u)
```



# DFS: Beispiel (XI)

Aufrufs-Stack: **DFS-VISIT (G, 3)**

↓↑

DFS-VISIT (G, u)

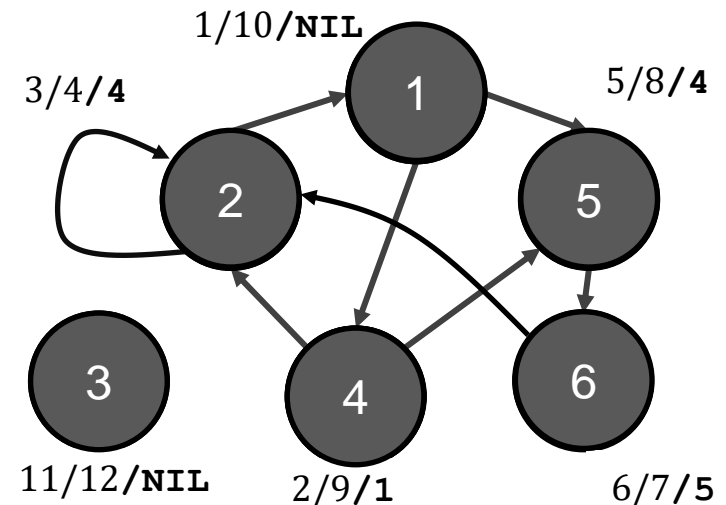
time=12

```
1  time=time+1;
2  u.disc=time;
3  u.color=GRAY;
4  FOREACH v in adj(G,u) DO
5      IF v.color==WHITE THEN
6          v.pred=u;
7          DFS-VISIT (G, v) ;
8  u.color=BLACK;
9  time=time+1;
10 u.finish=time;
```

Ordnung auf Knotenlisten gemäß Knotennummern

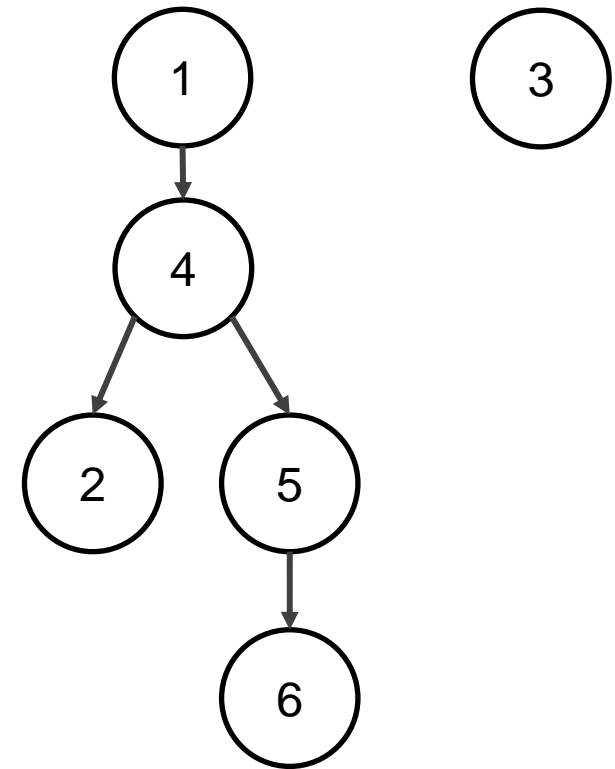
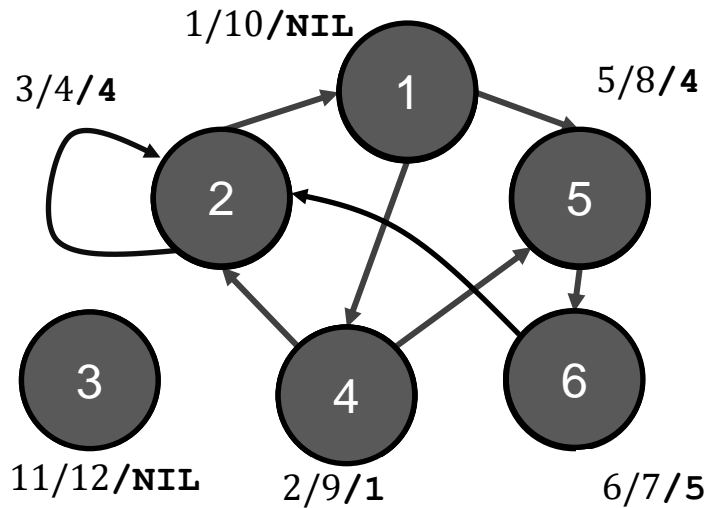
DFS (G) //G= (V, E)

```
1  FOREACH u in V DO
2      u.color=WHITE;
3      u.pred=NIL;
4  time=0;
5  FOREACH u in V DO
6      IF u.color==WHITE THEN
7          DFS-VISIT (G, u)
```





# DFS-Wald = Menge von DFS-Bäumen



Subgraph  $G_{\text{pred}} = (V, E_{\text{pred}})$  von  $G$ :

$$E_{\text{pred}} = \{ (\mathbf{v}.\text{pred}, \mathbf{v}) \mid \mathbf{v} \in V, \mathbf{v}.\text{pred} \neq \text{NIL} \}$$

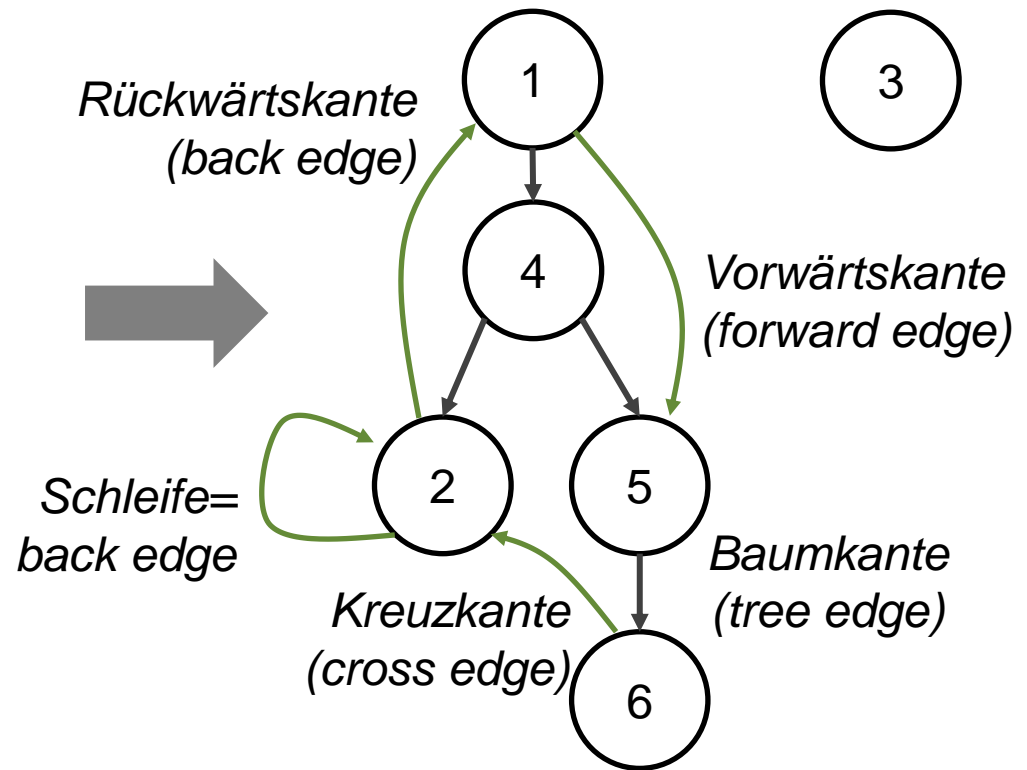
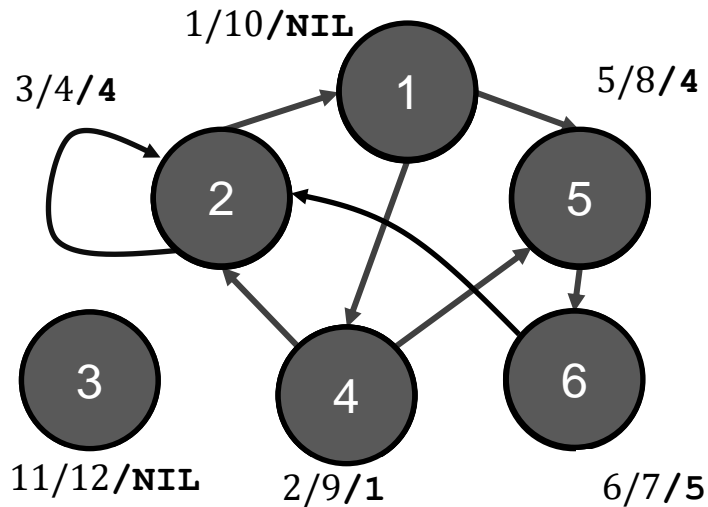
(und  $(\mathbf{v}, \mathbf{v}.\text{pred})$  für ungerichtete Graphen)

DFS-Baum gibt nicht  
unbedingt kürzesten  
Weg wieder:

$1 \rightarrow 5 \rightarrow 6$  vs.  $1 \rightarrow 4 \rightarrow 5 \rightarrow 6$

# Kante zeigen

Zeichne restlichen Kanten aus  $G$  auch in  $G_{\text{pred}}$  ein



Charakterisierung der Kanten in  $G$  mittels DFS

- Baumkanten: alle Kanten in  $G_{\text{pred}}$
- Vorwärtskanten: alle Kanten in  $G$  zu Nachkommen in  $G_{\text{pred}}$ , die nicht Baumkante
- Rückwärtskanten: alle Kanten in  $G$  zu Vorfahren in  $G_{\text{pred}}$ , die nicht Baumkante
- Kreuzkanten: alle anderen Kanten in  $G$  (inkl. Schleifen)

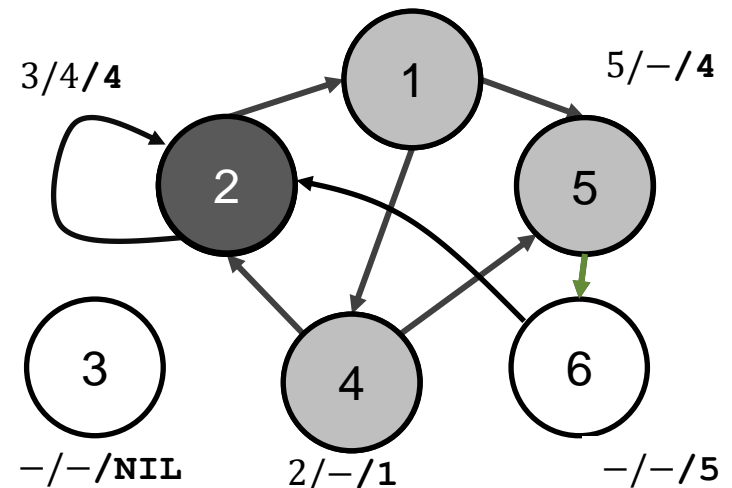
# Kantenart erkennen (I)

Sei  $(u, v)$  gerade betrachtete Kante im DFS-Algorithmus. Dann ist  $(u, v)$ ...

...eine Baumkante, wenn  $v.\text{color} == \text{WHITE}$

Beispiel:  $5 \rightarrow 6$

da  $v$  noch nicht besucht wurde  
und  $v.\text{pred} = u$  gesetzt wird  
und dann  $(v.\text{pred}, v) = (u, v)$   
als Kante in  $G_{\text{pred}}$  auftaucht



## Kantenart erkennen (II)

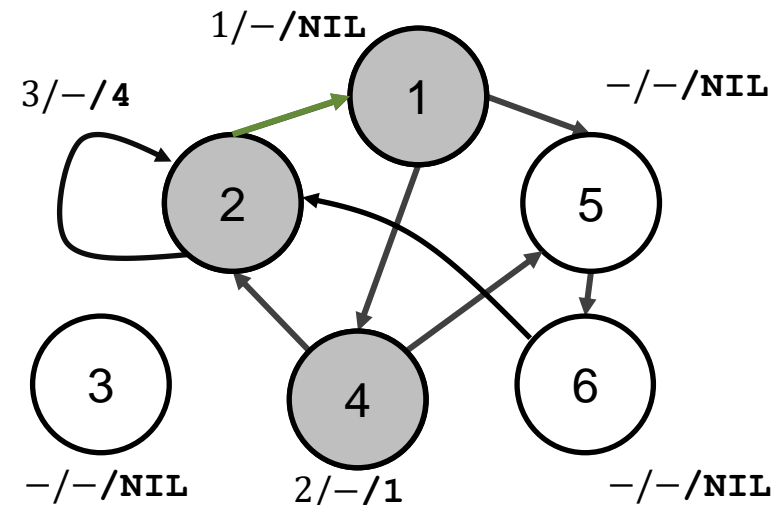
Sei  $(u, v)$  gerade betrachtete Kante im DFS-Algorithmus. Dann ist  $(u, v)$ ...

...eine Baumkante, wenn  $v.\text{color} == \text{WHITE}$

...eine Rückwärtskante, wenn  $v.\text{color} == \text{GRAY}$

Beispiel:  $2 \rightarrow 1$

da die Kette von  
grauen Knoten  
auch im DFS-Baum  
eine Kette bilden



## Kantenart erkennen (III)

Sei  $(u, v)$  gerade betrachtete Kante im DFS-Algorithmus. Dann ist  $(u, v)$ ...

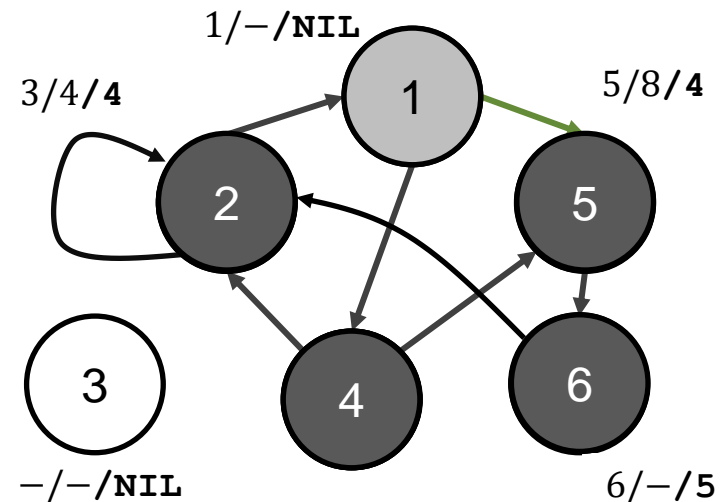
...eine Baumkante, wenn  $v.color == WHITE$

...eine Rückwärtskante, wenn  $v.color == GRAY$

...eine Vorwärtskante, wenn  $v.color == BLACK$  und  $u.disc < v.disc$

Beispiel:  $1 \rightarrow 5$

da  $u.disc < v.disc$  wurde  $v$  erst schwarz,  
als  $u$  schon grau war;  
aber  $u$  ist noch nicht abgeschlossen,  
also wurde  $v$  von einem  
echten Nachkomme von  $u$  besucht,  
da  $u$  vorher zu einem weißen Knoten überging



## Kantenart erkennen (IV)

Sei  $(u, v)$  gerade betrachtete Kante im DFS-Algorithmus. Dann ist  $(u, v)$ ...

...eine Baumkante, wenn  $v.\text{color} == \text{WHITE}$

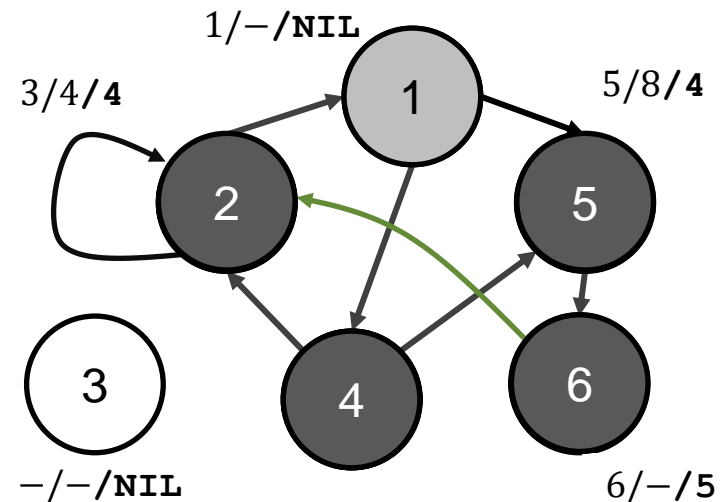
...eine Rückwärtskante, wenn  $v.\text{color} == \text{GRAY}$

...eine Vorwärtskante, wenn  $v.\text{color} == \text{BLACK}$  und  $u.\text{disc} < v.\text{disc}$

...eine Kreuzkante, wenn  $v.\text{color} == \text{BLACK}$  und  $v.\text{disc} < u.\text{disc}$

Beispiel:  $6 \rightarrow 2$

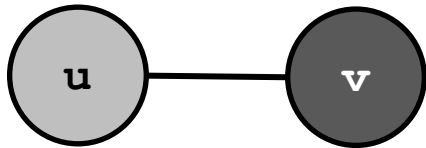
bilden per Definition  
die restlichen Kanten



# Kantenarten in ungerichteten Graphen (I)

In einem ungerichteten Graphen  $G$  entstehen durch DFS nur Baum- und Rückwärtskanten.

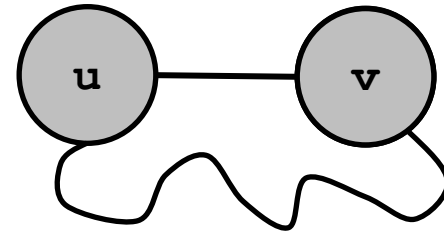
Zur Erinnerung: Vorwärts- und Kreuzkanten haben  $v.\text{color} == \text{BLACK}$



Sei  $u$  gerade aktiv (grau).  
Betrachtete neue Kante  $\{u, v\}$   
kann nur Vorwärts- oder  
Kreuzkante werden, wenn  
 $v$  schon abgeschlossen  
(schwarz).

1. Fall:  $u.\text{disc} < v.\text{disc}$

Hätte nur passieren können, als  $v$  von  $u$   
aus durch anderen Pfad bereits erreicht  
und  $v$  in dem Moment grau wurde:

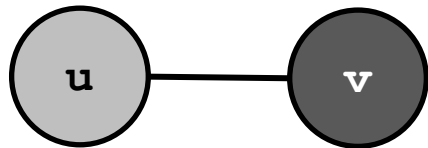


Dann wäre aber Kante  $\{v, u\}$  bereits  
bei  $v$  Rückwärtskante geworden.

# Kantenarten in ungerichteten Graphen (II)

In einem ungerichteten Graphen  $G$  entstehen durch DFS nur Baum- und Rückwärtskanten.

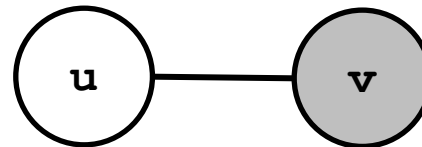
Zur Erinnerung: Vorwärts- und Kreuzkanten haben  $v.color == BLACK$



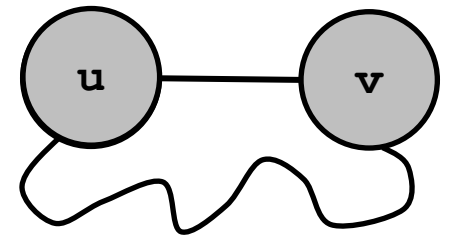
Sei  $u$  gerade aktiv (grau).  
Betrachtete neue Kante  $\{u, v\}$   
kann nur Vorwärts- oder  
Kreuzkante werden, wenn  
 $v$  schon abgeschlossen  
(schwarz).

2.Fall:  $u.disc > v.disc$

Hätte nur passieren können, wenn  $u$  grau  
geworden wäre, als  $v$  schon aktiv (grau) war:



direkt:  
 $u$  noch weiß und  $\{v, u\}$   
bereits Baumkante.

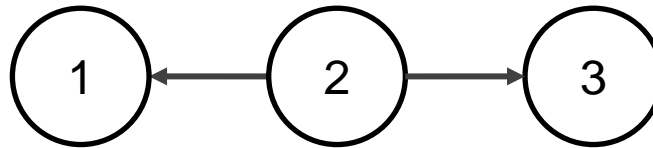


indirekt:  
 $u$  wird grau und  $\{u, v\}$   
bereits Rückwärtskante.





Geben Sie die Kreuzkanten in folgendem Graphen an, wenn die DFS die Knoten gemäß Ordnungsnummer besucht:



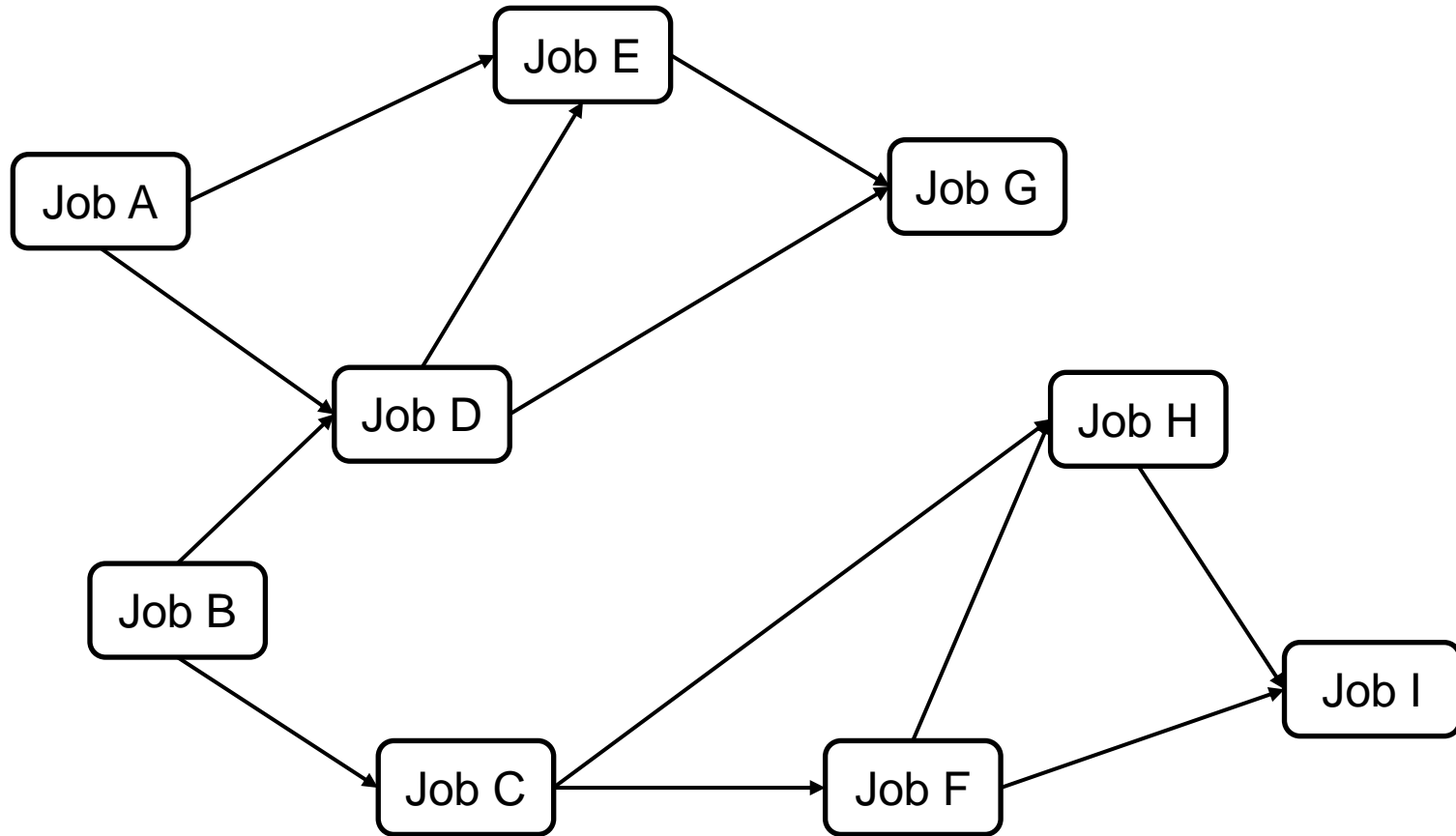
Geben Sie eine andere Ordnung für den Durchlauf des DFS an, bei der keine Kreuzkanten entstehen.

---

# **Anwendungen DFS: Topologisches Sortieren und Starke Zusammenhangskomponenten**

# Job Scheduling

Kante: Job X muss vor Job Y beendet sein



In welcher Reihenfolge sollen die Jobs bearbeitet werden?

# Anwendungen Job Scheduling

Quelle: [www.gnu.org](http://www.gnu.org)

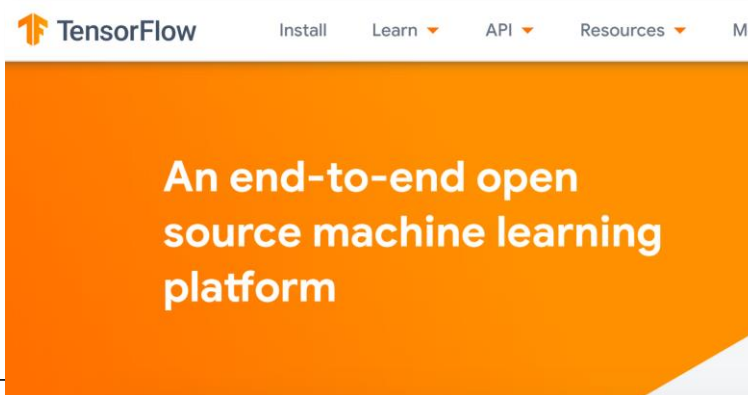
|   | A   | B          | C          | D    | E        | F    |
|---|-----|------------|------------|------|----------|------|
| 1 | 12  | 2          | 54         | 4    | 22322332 | 44   |
| 2 | 15  | 13.02.1900 | 454        | 3    | 12       | 232  |
| 3 | 14  | 232        | 54         | 455  | 2222     | 2    |
| 4 | 12  | 2          | 05.01.1900 | 22   | 2        | 12   |
| 5 | 3   | 12         | 232        | 4545 | 2222     | 2    |
| 6 | 455 | 2222       | 2          | 232  | 455      | 455  |
| 7 | 22  | 2          | 12         | 455  | 22       | 22   |
| 8 | 4   | 44         | 2222       | 4545 | 4545     | 4545 |
| 9 | 4   | 42         | 2          |      |          |      |

Spreadsheets (Formeln aktualisieren)

```
edit : main.o kbd.o command.o display.o \  
      insert.o search.o files.o utils.o  
      cc -o edit main.o kbd.o display.o \  
      insert.o search.o files.o utils.o
```

```
main.o : main.c defs.h  
      cc -c main.c  
kbd.o : kbd.c defs.h command.h  
      cc -c kbd.c  
...
```

**makefile** (Dateien compilieren)



Quelle:

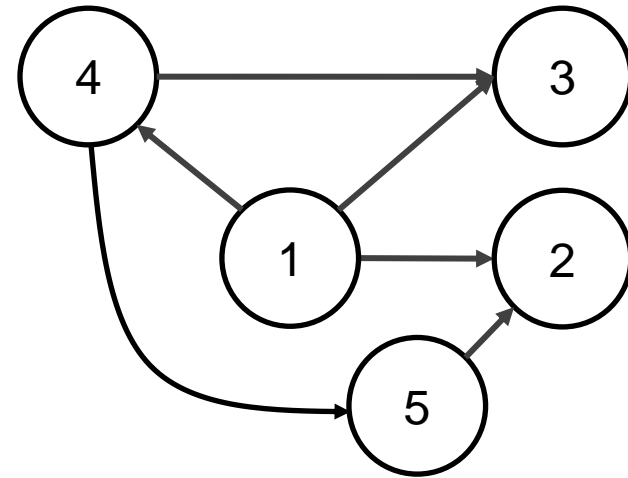
[www.tensorflow.org](http://www.tensorflow.org)

(Berechnung als  
Computation Graph)

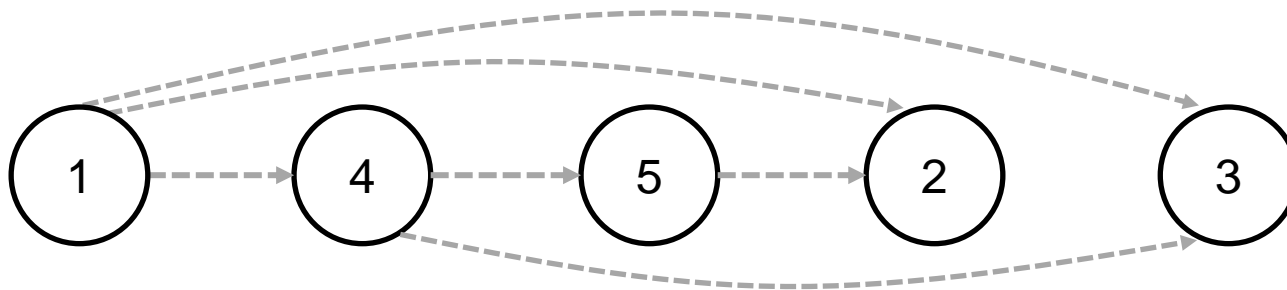
# Abstrakte Modellierung

Topologische Sortierung nur für  
„directed acyclic graph“ (dag):

gerichteter Graph ohne Zyklen



Topologische Sortierung eines dag  $G = (V, E)$ :  
Knoten in linearer Ordnung, so dass für alle Knoten  $u, v \in V$  gilt,  
dass  $u$  vor  $v$  in der Ordnung kommt, wenn  $(u, v) \in E$ .



„Kanten gehen immer nur nach rechts“

Sortierung nicht eindeutig,  
z.B. hier 2 und 3 vertauschbar

# Topologisches Sortieren mittels DFS

```
TOPOLOGICAL-SORT(G) // G=(V,E) dag
```

```
1  newLinkedList(L);  
2  run DFS(G) but, each time a node is finished,  
   insert in front of L  
3  return L.head
```

Laufzeit =  $O(|V| + |E|)$

da Einfügen  
in Liste vorne  
in Zeit  $O(1)$

# Topologisches Sortieren: Korrektheit (I)

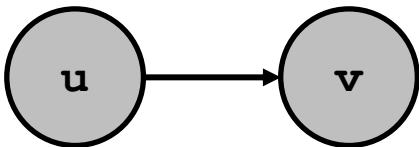
```
TOPOLOGICAL-SORT(G) //  $G=(V,E)$  dag
```

```
1  newLinkedList(L);  
2  run DFS(G) but, each time a node is finished,  
   insert in front of L  
3  return L.head
```

Korrektheit:

Es genügt zu zeigen, dass jede von DFS inspizierte Kante  $(u,v) \in E$  erfüllt:  
 $v.\text{finish} < u.\text{finish}$ , so dass  $u$  zeitlich nach  $v$  in Liste eingefügt wird  
und daher positionell vor  $v$  in Liste zu finden ist

1.Fall:  $v$  bereits grau



Würde Rückwärtskante erzeugen,  
d.h. der Graph hätte einen Zyklus (Widerspruch).

# Topologisches Sortieren: Korrektheit (II)

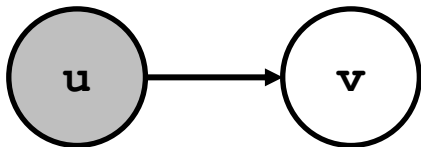
```
TOPOLOGICAL-SORT(G) //  $G=(V,E)$  dag
```

```
1  newLinkedList(L);  
2  run DFS(G) but, each time a node is finished,  
   insert in front of L  
3  return L.head
```

Korrektheit:

Es genügt zu zeigen, dass jede von DFS inspizierte Kante  $(u,v) \in E$  erfüllt:  
 $v.\text{finish} < u.\text{finish}$ , so dass  $u$  zeitlich nach  $v$  in Liste eingefügt wird  
und daher positionell vor  $v$  in Liste zu finden ist

2.Fall:  $v$  noch weiß



Erzeugt Baumkante, also wird  $v$  Nachfahre von  $u$   
und  $v.\text{finish} < u.\text{finish}$ , da Aufrufs-Stack  
nicht zu  $u$  zurückkehrt, bevor  $v$  abgeschlossen.



# Topologisches Sortieren: Korrektheit (III)

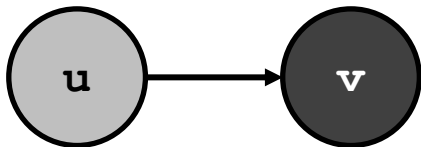
```
TOPOLOGICAL-SORT(G) //  $G=(V,E)$  dag
```

```
1  newLinkedList(L);  
2  run DFS(G) but, each time a node is finished,  
   insert in front of L  
3  return L.head
```

Korrektheit:

Es genügt zu zeigen, dass jede von DFS inspizierte Kante  $(u,v) \in E$  erfüllt:  
 $v.\text{finish} < u.\text{finish}$ , so dass  $u$  zeitlich nach  $v$  in Liste eingefügt wird  
und daher positionell vor  $v$  in Liste zu finden ist

3.Fall:  $v$  schwarz



Dann  $v.\text{finish}$  bereits gesetzt,  
während  $u.\text{finish}$  erst später gesetzt wird,  
also  $v.\text{finish} < u.\text{finish}$ .



Wie können Sie bei der topologischen Sortierung mittels DFS fast ohne zusätzlichen Aufwand erkennen, ob ihr Graph einen Zyklus hat?



Betrachten Sie folgendes Verfahren für topologisches Sortieren, wenn die Anzahl eingehender Kanten jeweils bekannt ist. Argumentieren Sie, dass das Verfahren korrekt ist:

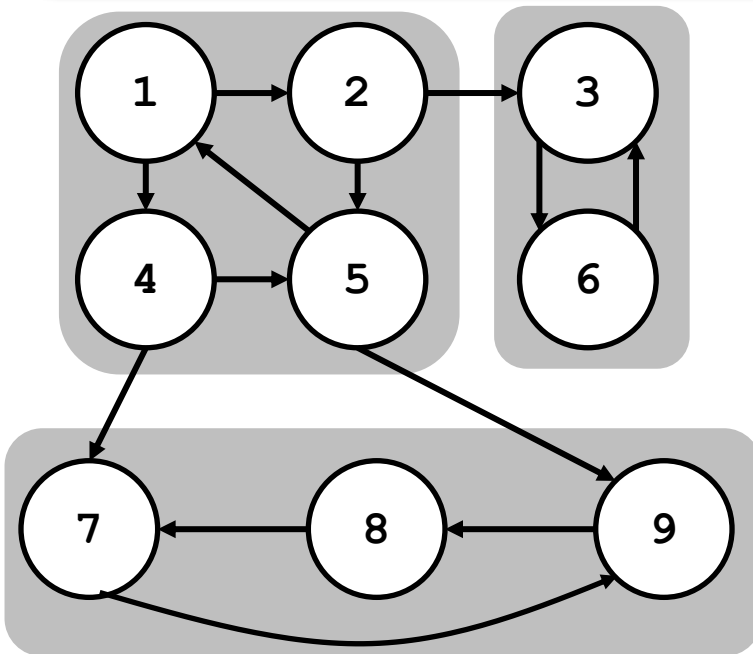
```
Kahn1962(G) //G=(V,E) dag, inbound[u]=#edges to u

1  WHILE !isEmpty(V) DO
2      pick vertex u with inbound[u]==0
3      add u at the end of a list L
4      inbound[v]=inbound[v]-1 for each v with (u,v) in E
5      remove u from V and all (u,*) from E
```

# Starke Zusammenhangskomponenten

Eine starke Zusammenhangskomponente eines gerichteten Graphen  $G = (V, E)$  ist eine Knotenmenge  $C \subseteq V$ , so dass

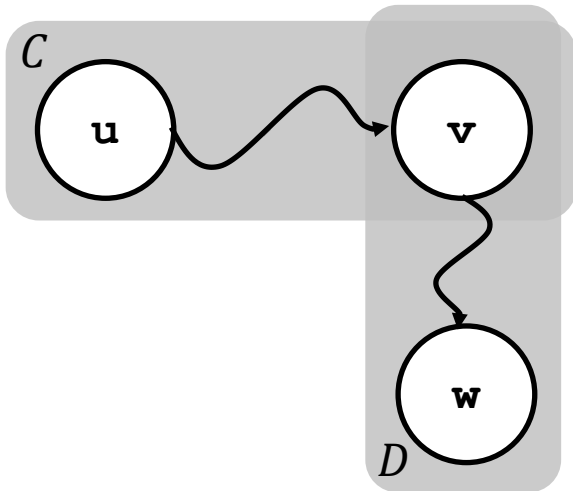
- (a) es zwischen je zwei Knoten  $u, v \in C$  einen Pfad von  $u$  nach  $v$  gibt, und
- (b) es keine Menge  $D \subseteq V$  mit  $C \subsetneq D$  gibt, für die (a) auch gilt  
( $C$  ist maximal).



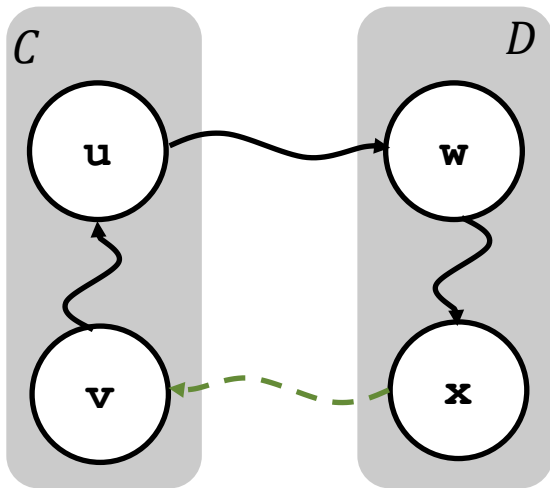
Anwendung: Model Checking für  
Korrektheitsnachweis von Systemen

Graph kann mehrere  
starke Zusammenhangskomponenten  
(strongly connected components, SCCs) haben

# Eigenschaften



Verschiedene SCCs  $C, D$  sind disjunkt, sonst gäbe es  $v \in C \cap D$  und für beliebige  $u \in C$  und  $w \in D$  auch einen Pfad von  $u$  nach  $w$  über  $v$  (und umgekehrt), somit wären  $C$  und  $D$  identisch.



Wenn es für verschiedene SCCs  $C, D$  mit  $u, v \in C$  und  $w, x \in D$  einen Pfad  $u \rightarrow w$  gibt, dann kann es keinen Pfad  $x \rightarrow v$  geben, sonst wären  $C$  und  $D$  identisch.

„Zwei SCCs sind nur in eine Richtung verbunden.“

# SCC Algorithmus: Ansatz

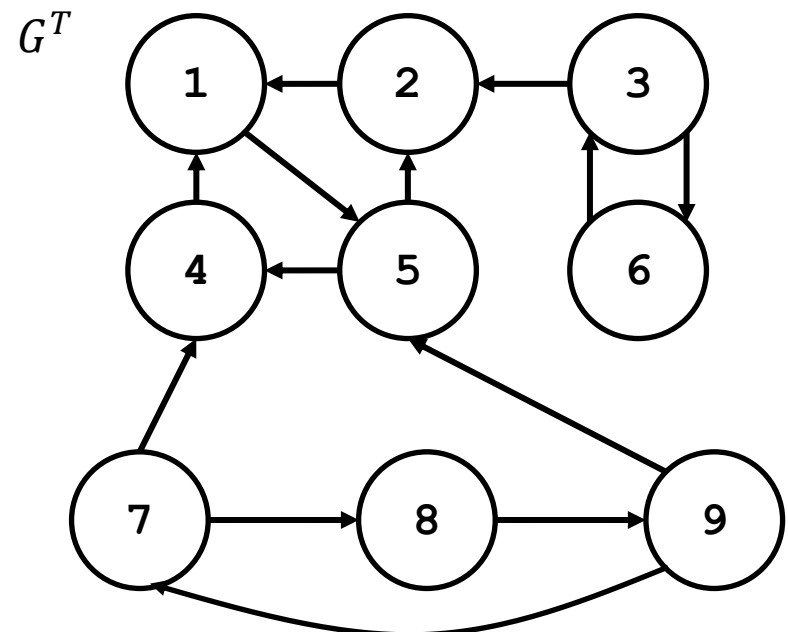
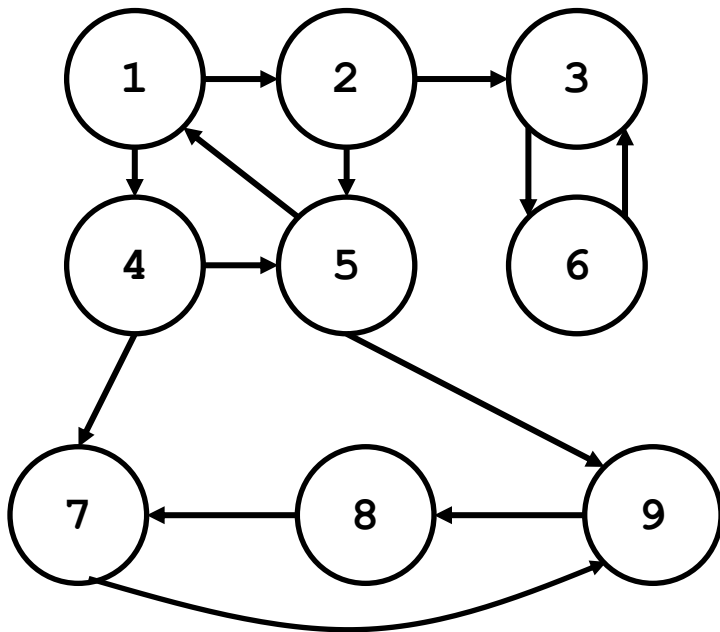
Algorithmus von Kosaraju

Lasse zweimal DFS laufen:

- einmal auf Graph  $G$
- einmal auf transponiertem Graphen  $G^T = (V, E^T)$ :

$$E^T = \{ (v, u) \mid (u, v) \in E \}$$

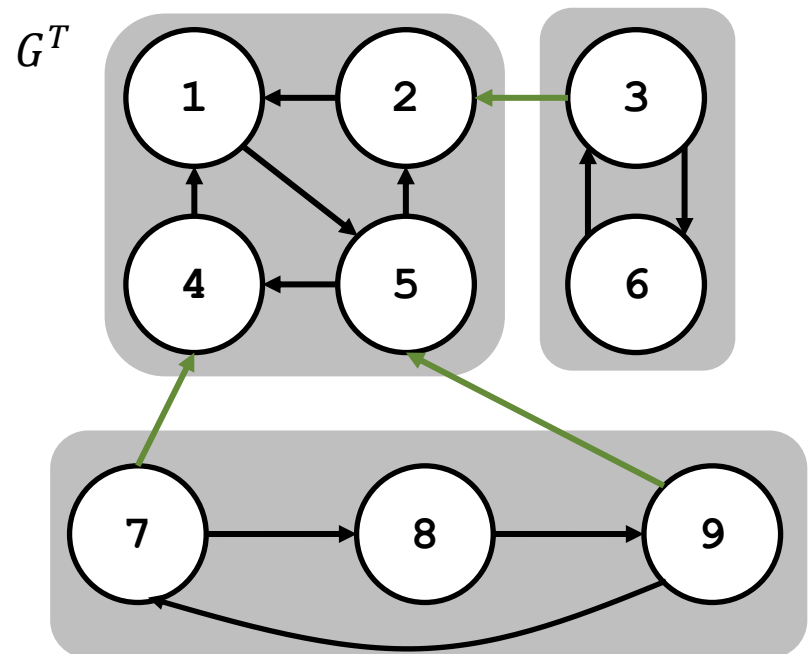
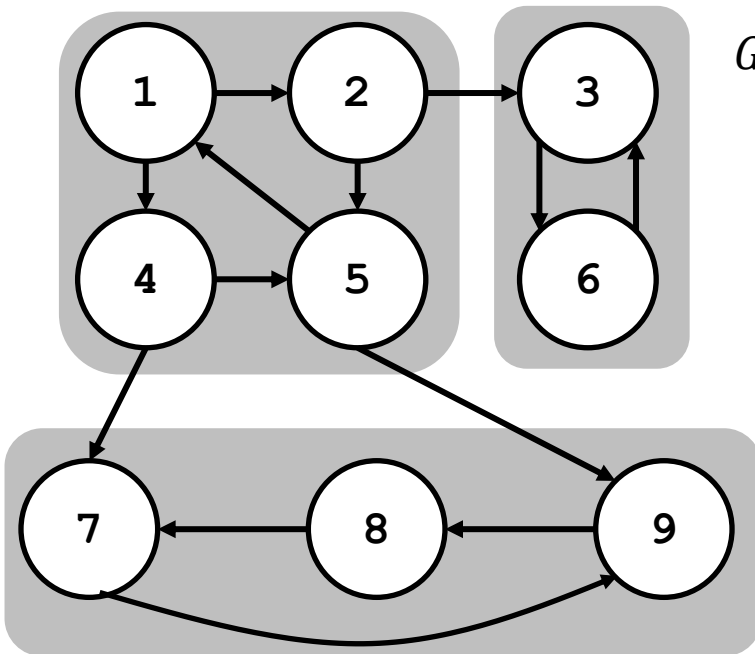
„drehe Kanten in  $G$  um“



# SCCs im transponierten Graphen

SCCs in  $G$  und  $G^T$  bleiben identisch  
(in beiden Fällen gibt es in jeder SCC  
einen Weg von jedem Knoten zum anderen Knoten)

nur Übergänge zwischen SCCs drehen sich um



# SCC Algorithmus

Laufzeit =  $O(|V| + |E|)$

SCC(G) //  $G=(V,E)$  directed graph

- 1 run DFS(G)
- 2 compute  $G^T$
- 3 run DFS( $G^T$ ) but visit vertices in main loop  
in descending finish time from 1
- 4 output each DFS tree in 3 as one SCC

DFS(G) //  $G=(V,E)$

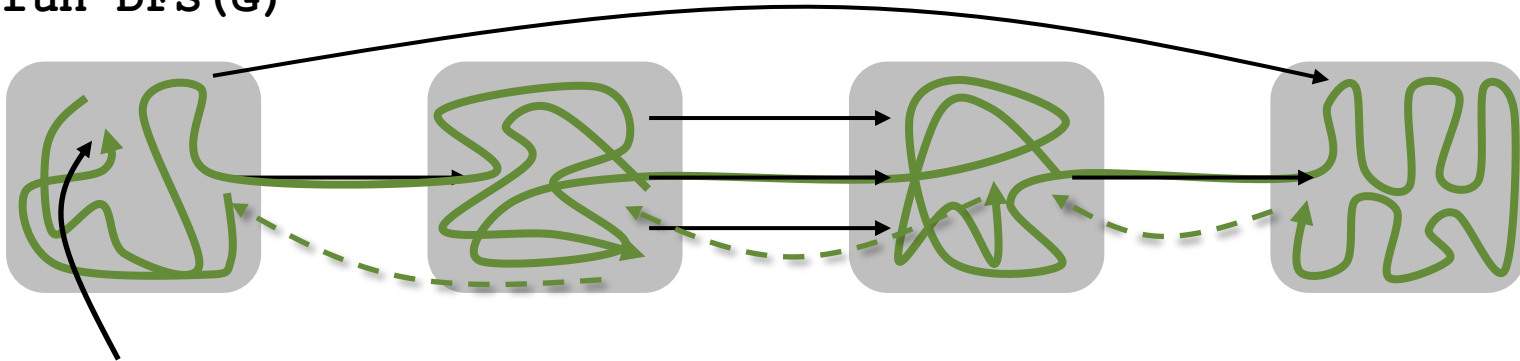
- 1 FOREACH  $u$  in  $V$  DO
- 2      $u.color=WHITE$ ;
- 3      $u.pred=NIL$ ;
- 4  $time=0$ ;
- 5 FOREACH  $u$  in  $V$  DO
- 6     IF  $u.color==WHITE$  THEN
- 7         DFS-VISIT( $G,u$ )

# SCC Algorithmus: Idee (I)

SCC (G) //  $G=(V,E)$  directed graph

- 1 run DFS (G)
- 2 compute  $G^T$
- 3 run DFS( $G^T$ ) but visit vertices in main loop in descending finish time from 1
- 4 output each DFS tree in 3 as one SCC

1 run DFS (G)



Knoten mit höchster  
**finish** time liegt  
in dieser SCC

Bild nimmt zusammenhängenden Graphen an,  
funktioniert aber auch allgemein, dann nacheinander  
für die zusammenhängenden Subgraphen

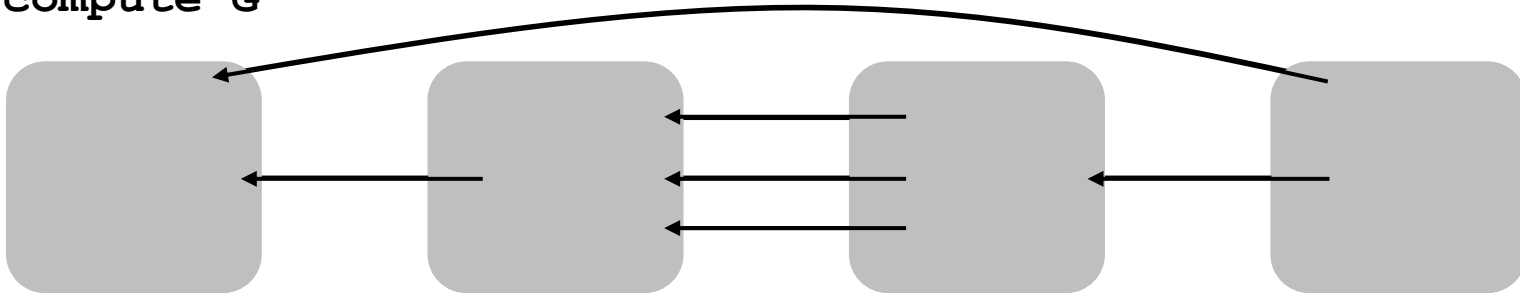


# SCC Algorithmus: Idee (II)

SCC(G) //  $G=(V,E)$  directed graph

- 1 run DFS(G)
- 2 compute  $G^T$
- 3 run DFS( $G^T$ ) but visit vertices in main loop in descending finish time from 1
- 4 output each DFS tree in 3 as one SCC

2 compute  $G^T$



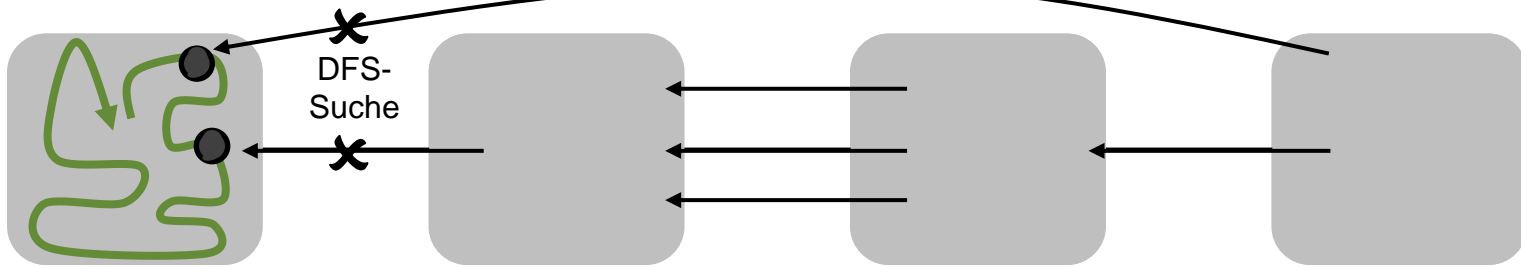
SCCs bleiben erhalten

# SCC Algorithmus: Idee (III)

SCC (G) //  $G=(V,E)$  directed graph

- 1 run DFS (G)
- 2 compute  $G^T$
- 3 run DFS( $G^T$ ) but visit vertices in main loop in descending finish time from 1
- 4 output each DFS tree in 3 as one SCC

3 run DFS ( $G^T$ ) ...



Besucht alle Knoten der SCC und kehrt dann zu Hauptschleife DFS zurück, da kein Übergang zum nächsten SCC

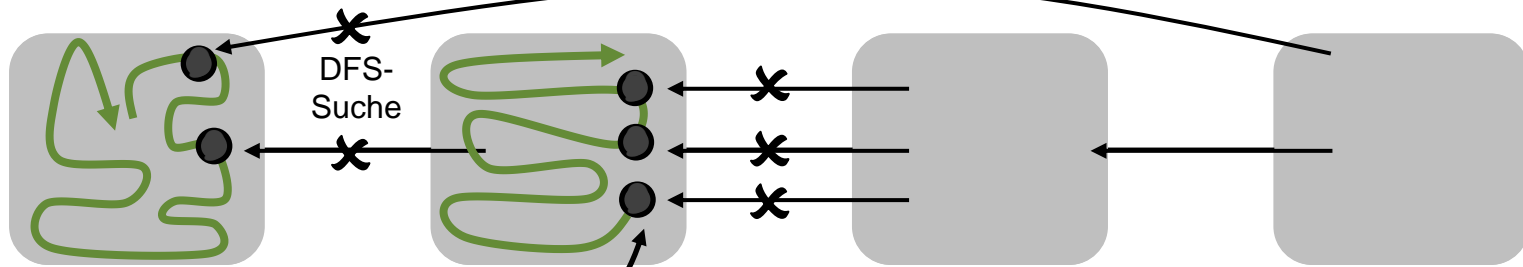
Diesen DFS-Baum (und damit die SCC) geben wir in Schritt 4 aus

# SCC Algorithmus: Idee (IV)

SCC (G) //  $G=(V,E)$  directed graph

- 1 run DFS (G)
- 2 compute  $G^T$
- 3 run DFS( $G^T$ ) but visit vertices in main loop in descending finish time from 1
- 4 output each DFS tree in 3 as one SCC

3 run DFS ( $G^T$ ) ...



verbleibender Knoten mit  
nächsthöheren  
**finish** time liegt  
in dieser SCC

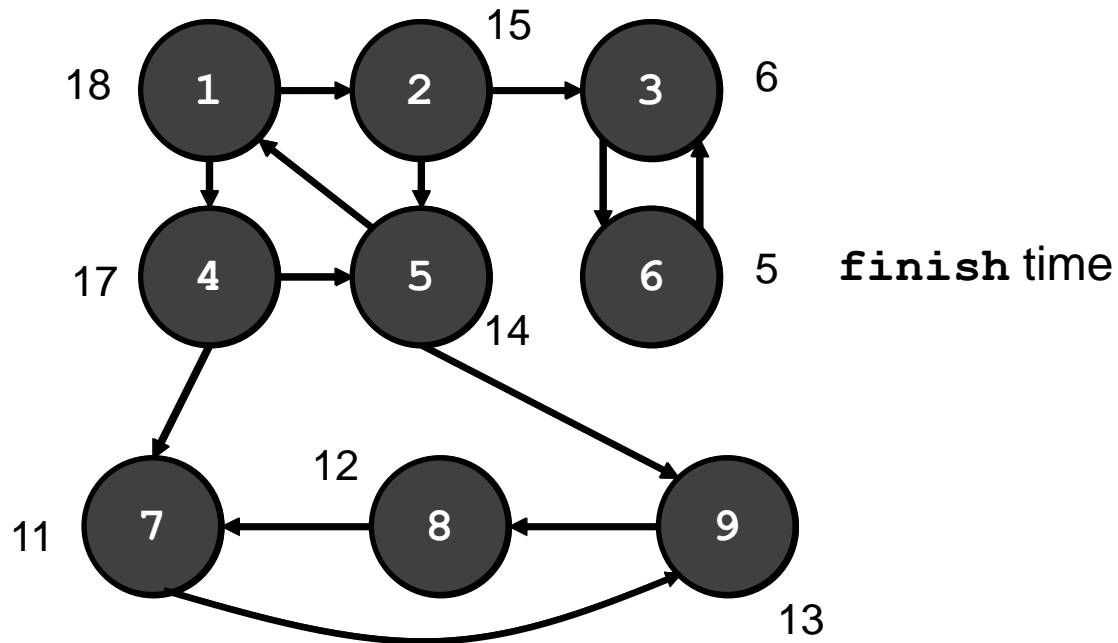
Besucht alle Knoten der SCC  
und kehrt dann zu Hauptschleife  
DFS zurück, da kein Übergang  
zum nächsten SCC

usw.

# SCC Algorithmus: Beispiel (I)

SCC(G) //  $G=(V,E)$  directed graph

- 1 run DFS(G)
- 2 compute  $G^T$
- 3 run DFS( $G^T$ ) but visit vertices in main loop in descending finish time from 1
- 4 output each DFS tree in 3 as one SCC

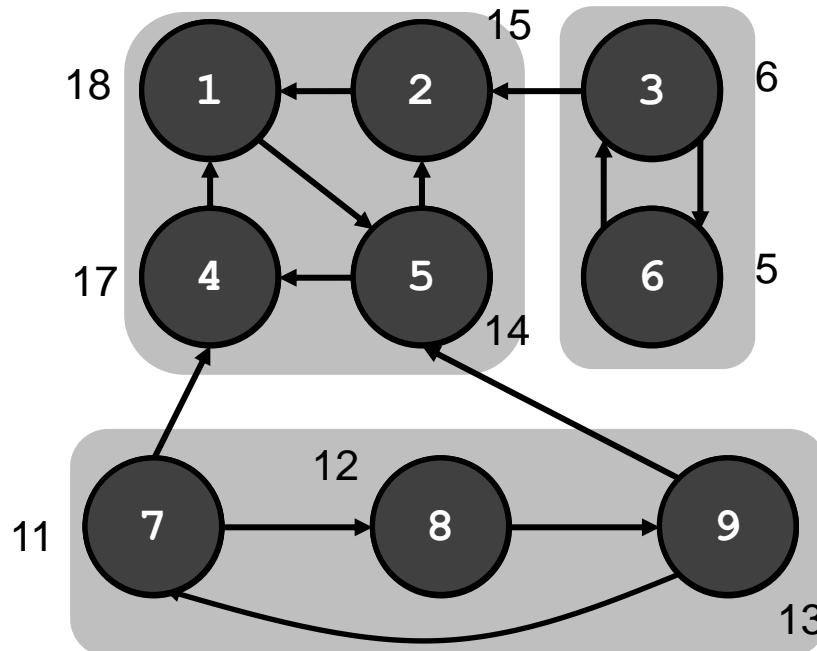


# SCC Algorithmus: Beispiel (II)

SCC(G) //  $G=(V,E)$  directed graph

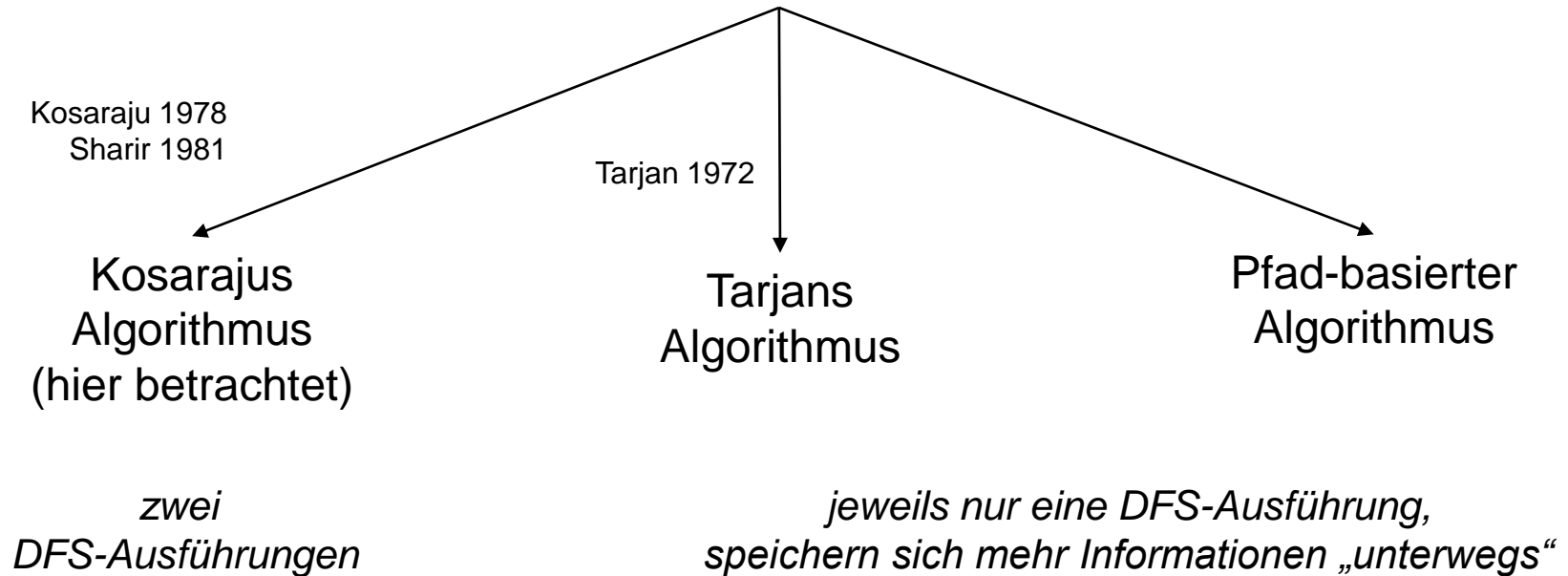
- 1 run DFS(G)
- 2 compute  $G^T$
- 3 run DFS( $G^T$ ) but visit vertices in main loop in descending finish time from 1
- 4 output each DFS tree in 3 as one SCC

umgedrehte Kanten



# Algorithmendesign

alle bekannten SCC-Algorithmen  
sind DFS-basiert



asymptotisch alle gleich schnell, aber Tarjans und pfad-basierter Algorithmus schneller in Praxis

# Minimale Spannbäume

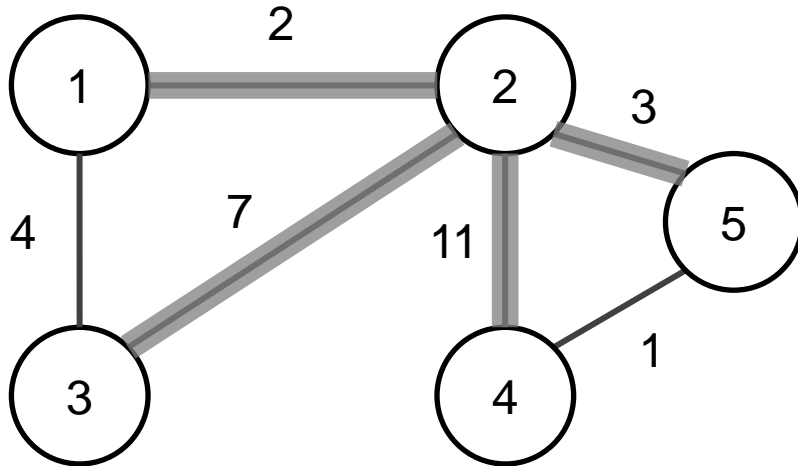
# Minimaler Spannbaum (MST)

Für einen zusammenhängenden, ungerichteten, gewichteten Graphen  $G = (V, E)$  mit Gewichten  $w$  ist der Subgraph  $T = (V, E_T)$  von  $G$  ein **Spannbaum** („spanning tree“), wenn  $T$  azyklisch ist und alle Knoten verbindet.

Der Spannbaum ist **minimal**, wenn

$$w(T) = \sum_{\{u,v\} \in E_T} w(\{u,v\})$$

minimal für alle Spannbäume von  $G$  ist.



Spannbaum des Graphen  
(durch breite  
Kanten gekennzeichnet)



# Gewichtsvergleich

...

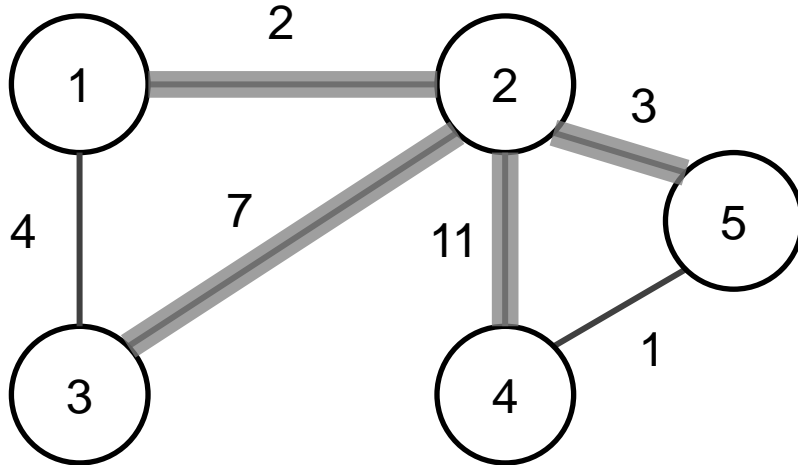
Der Spannbaum ist **minimal**, wenn

$$w(T) = \sum_{\{u,v\} \in E_T} w(\{u,v\})$$

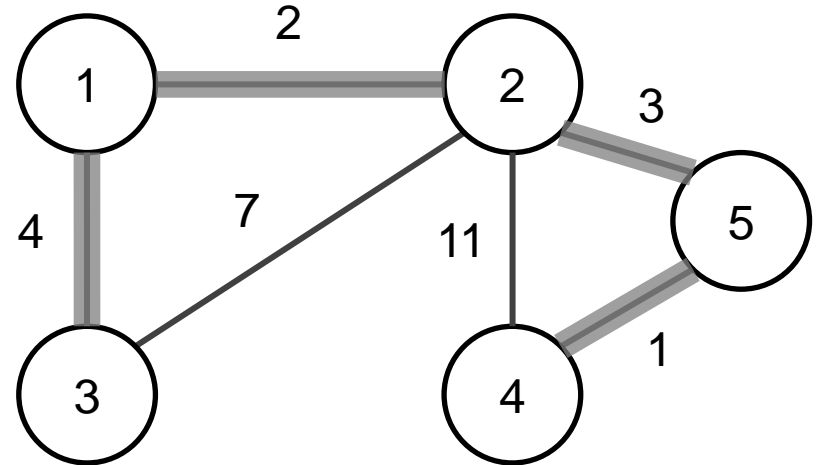
minimal für alle Spannäume von  $G$  ist.

rechter Spannbaum ist minimal, da jeder Spannbaum, der Gewicht 7 oder 11 enthält, ein größeres Gesamtgewicht hat.

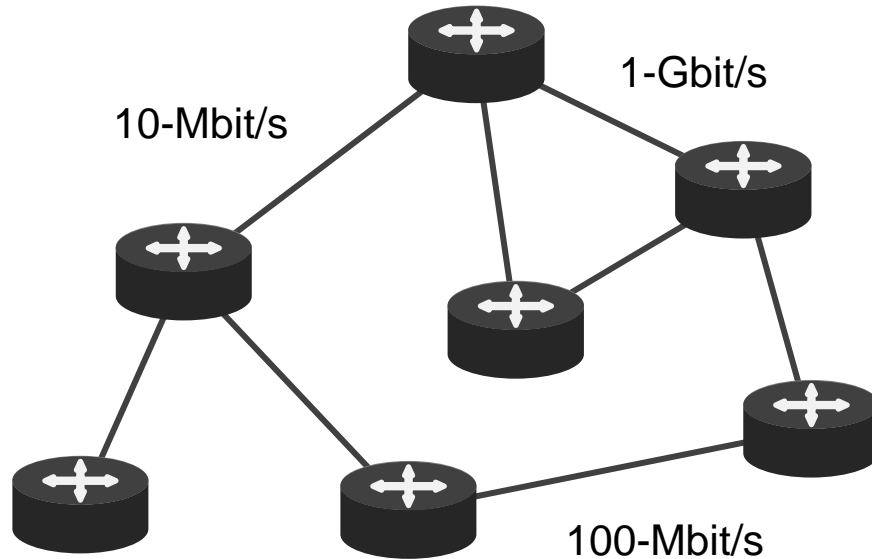
$$w(T) = 2 + 3 + 7 + 11 = 23$$



$$w(T') = 4 + 2 + 3 + 1 = 10$$



# Anwendung: Broadcast in Netzwerken



Broadcast:  
verteile Nachricht an alle Switches

verhindere „Broadcast Storm“:  
Nachricht würde stets  
zyklisch weiterverteilt

Spanning Tree Protocol (IEEE 802.1D):

Wähle „Root Bridge“ als Wurzel des Spannbaums

Gewicht abhängig von Geschwindigkeit und Entfernung von Root Bridge

# Allgemeiner MST-Algorithmus: Idee

```
genericMST(G,w) // G=(V,E) undirected, connected graph
                  w weight function

1  A=∅
2  WHILE A does not form a spanning tree for G DO
3      find safe edge {u,v} for A
4      A = A ∪ {{u,v}}
5  return A
```

**A** Teilmenge der Kanten eines MST

Kante  $\{u,v\}$  ist sicher („safe“) für **A**,  
wenn  $A \cup \{u,v\}$  noch Teilmenge eines MST ist

# Allgemeiner MST-Algorithmus

```
genericMST(G,w) // G=(V,E) undirected, connected graph
                  w weight function

1  A=∅
2  WHILE A does not form a spanning tree for G DO
3      find safe edge {u,v} for A
4      A = A ∪ {{u,v}}
5  return A
```

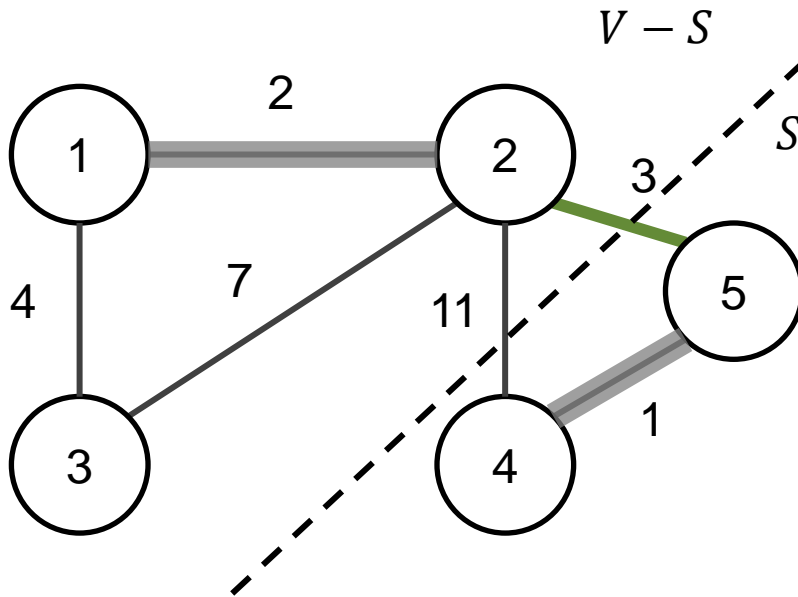
Terminierung:

Da wir zeigen werden, dass es in jeder Iteration eine sichere Kante für **A** gibt (sofern **A** noch kein Spannbaum), terminiert die Schleife nach maximal  $|E|$  Iterationen.

Korrektheit:

Da in jeder Iteration nur sichere Kanten hinzugefügt werden (für die  $A \cup \{ \{u,v\} \}$  noch Teilmenge eines MST ist), ist am Ende der **WHILE**-Schleife **A** ein MST.

# Terminologie



Kanten  $\{2, 5\}$ ,  $\{2, 4\}$   
überbrücken Schnitt

$\mathbf{A}$  (grau markierte Kanten)  
von Schnitt respektiert

Kante  $\{2, 5\}$  ist leicht  
für Schnitt

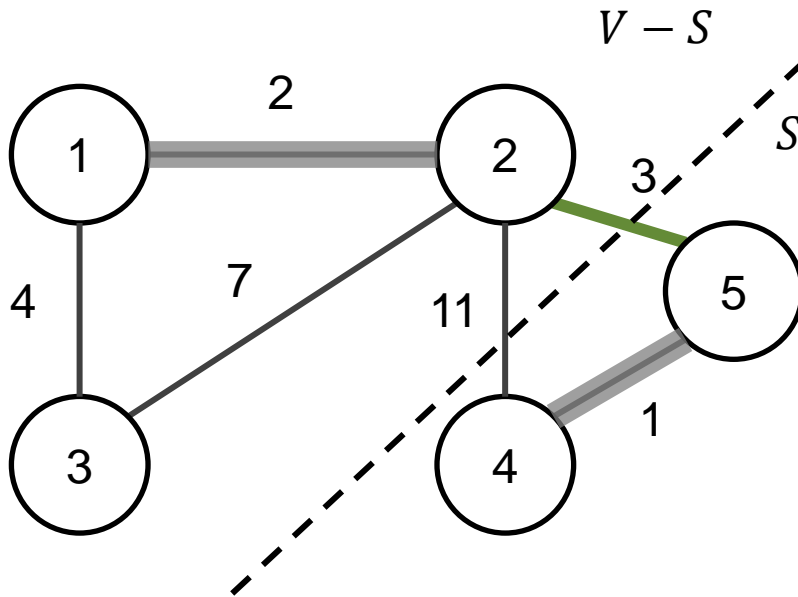
**Schnitt**  $(S, V - S)$  partitioniert  
Knoten des Graphen in zwei Mengen

$\{u, v\}$  **überbrückt Schnitt**  $(S, V - S)$ ,  
wenn  $u \in S$  und  $v \in V - S$

**Schnitt**  $(S, V - S)$  **respektiert**  $\mathbf{A} \subseteq E$ ,  
wenn keine Kante  $\{u, v\}$  aus  $\mathbf{A}$   
den Schnitt überbrückt

$\{u, v\}$  **leichte Kante für**  $(S, V - S)$ ,  
wenn  $w(\{u, v\})$  minimal für alle den  
Schnitt überbrückenden Kanten

## Leicht = sicher



Sei  $\mathbf{A}$  Teilmenge eines MST,  
 $(S, V-S)$  Schnitt, der  $\mathbf{A}$  respektiert,  
und  $\{u, v\}$  eine leichte Kante,  
die den Schnitt überbrückt.  
Dann ist  $\{u, v\}$  sicher für  $\mathbf{A}$ .

$\{u, v\}$  **sicher** für  $\mathbf{A}$ , wenn  
 $\mathbf{A} \cup \{\{u, v\}\}$  Teilmenge eines MST

**Schnitt**  $(S, V-S)$  partitioniert  
Knoten des Graphen in zwei Mengen

$\{u, v\}$  **überbrückt Schnitt**  $(S, V-S)$ ,  
wenn  $u \in S$  und  $v \in V-S$

**Schnitt**  $(S, V-S)$  **respektiert**  $\mathbf{A} \subseteq E$ ,  
wenn keine Kante  $\{u, v\}$  aus  $\mathbf{A}$   
den Schnitt überbrückt

$\{u, v\}$  **leichte Kante** für  $(S, V-S)$ ,  
wenn  $w(\{u, v\})$  minimal für alle den  
Schnitt überbrückenden Kanten

## Leicht = sicher: Beweis (I)

Sei  $T$  ein MST, der  $\mathbf{A}$  enthält.  
Wenn  $\{u, v\}$  in  $T$ , dann fertig.

Wenn  $\{u, v\}$  nicht in  $T$ , dann  
konstruieren wir MST  $U$ , der  
 $\mathbf{A} \cup \{\{u, v\}\}$  enthält, folglich  
ist  $\{u, v\}$  trotzdem sicher für  $\mathbf{A}$ .

Sei  $\mathbf{A}$  Teilmenge eines MST,  
 $(S, V-S)$  Schnitt, der  $\mathbf{A}$  respektiert,  
und  $\{u, v\}$  eine leichte Kante,  
die den Schnitt überbrückt.  
Dann ist  $\{u, v\}$  sicher für  $\mathbf{A}$ .

$\{u, v\}$  **sicher** für  $\mathbf{A}$ , wenn  
 $\mathbf{A} \cup \{\{u, v\}\}$  Teilmenge eines MST

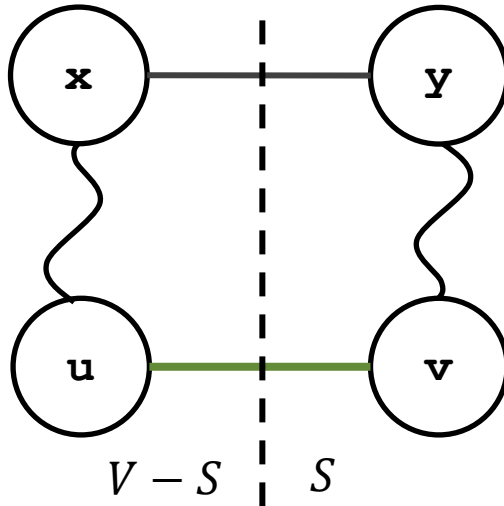
**Schnitt**  $(S, V - S)$  partitioniert  
Knoten des Graphen in zwei Mengen

$\{u, v\}$  **überbrückt Schnitt**  $(S, V - S)$ ,  
wenn  $u \in S$  und  $v \in V - S$

**Schnitt**  $(S, V - S)$  **respektiert**  $\mathbf{A} \subseteq E$ ,  
wenn keine Kante  $\{u, v\}$  aus  $\mathbf{A}$   
den Schnitt überbrückt

$\{u, v\}$  **leichte Kante** für  $(S, V - S)$ ,  
wenn  $w(\{u, v\})$  minimal für alle den  
Schnitt überbrückenden Kanten

## Leicht = sicher: Beweis (II)



MST  $T$  muss eine andere überbrückende Kante  $\{x, y\}$  für den Pfad von  $u$  nach  $v$  enthalten, damit alle Knoten erreichbar sind.

Da der Schnitt  $\mathbf{A}$  respektiert, ist diese Kante  $\{x, y\}$  nicht in  $\mathbf{A}$  enthalten.

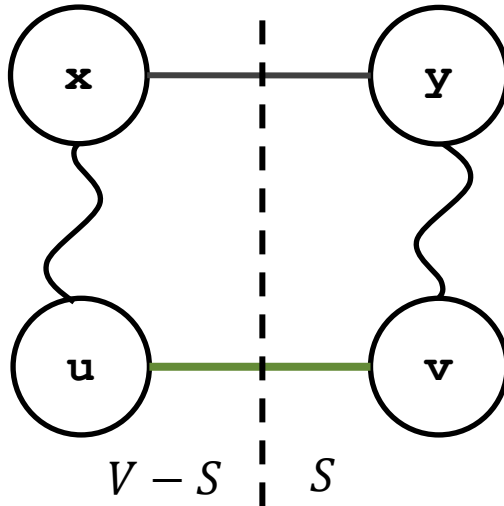
Setze  $U = (T - \{x, y\}) \cup \{u, v\}$

Sei  $\mathbf{A}$  Teilmenge eines MST,  $(S, V-S)$  Schnitt, der  $\mathbf{A}$  respektiert, und  $\{u, v\}$  eine leichte Kante, die den Schnitt überbrückt. Dann ist  $\{u, v\}$  sicher für  $\mathbf{A}$ .

$U$  ist Spannbaum, da jeder Knoten erreichbar ist:  
Nimm statt „Brücke“  $\{x, y\}$  den Pfad  $x$  nach  $u$ , dann  $\{u, v\}$ , dann  $v$  nach  $y$ .



## Leicht = sicher: Beweis (III)



MST  $T$  muss eine andere überbrückende Kante  $\{x, y\}$  für den Pfad von  $u$  nach  $v$  enthalten, damit alle Knoten erreichbar sind.

Da der Schnitt  $\mathbf{A}$  respektiert, ist diese Kante  $\{x, y\}$  nicht in  $\mathbf{A}$  enthalten.

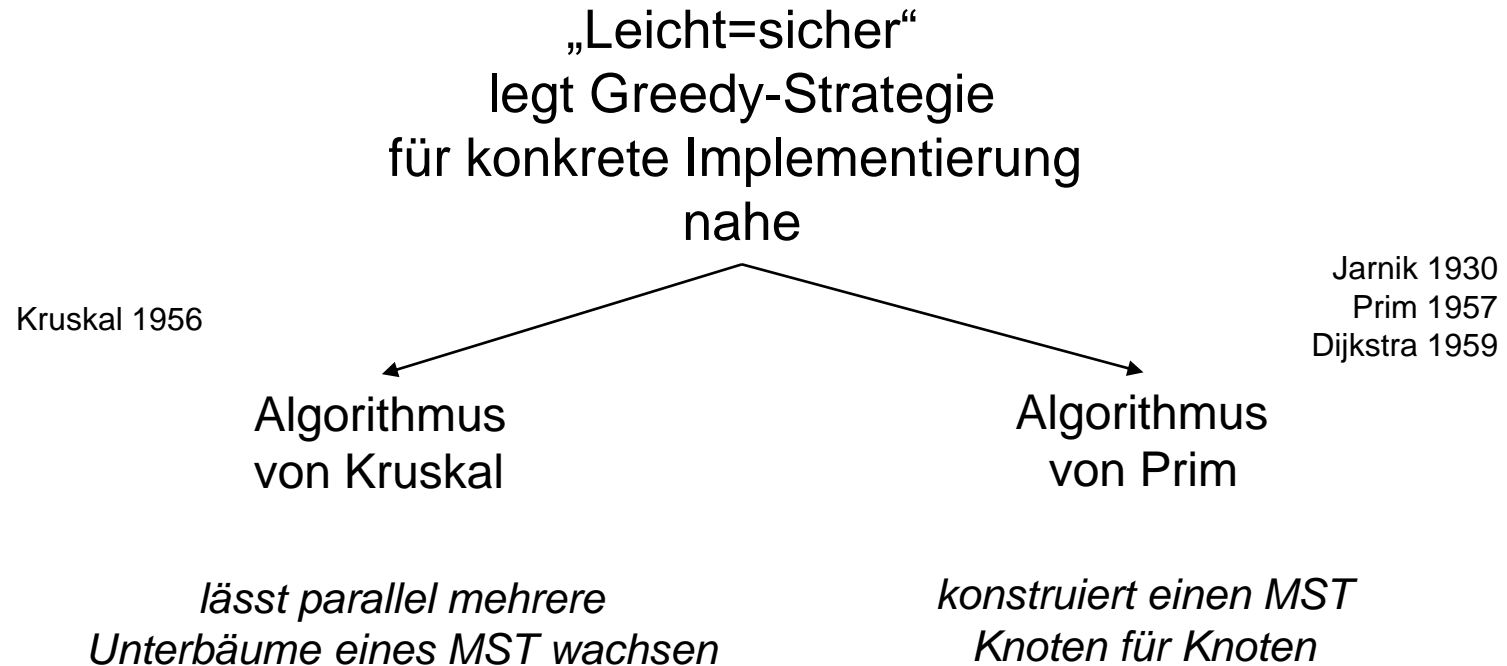
Setze  $U = (T - \{x, y\}) \cup \{u, v\}$

Sei  $\mathbf{A}$  Teilmenge eines MST,  $(S, V-S)$  Schnitt, der  $\mathbf{A}$  respektiert, und  $\{u, v\}$  eine leichte Kante, die den Schnitt überbrückt. Dann ist  $\{u, v\}$  sicher für  $\mathbf{A}$ .

$U$  ist minimal, da für leichte Kante  $\{u, v\}$  gilt:  $w(\{u, v\}) \leq w(\{x, y\})$  und

$$w(U) = w(T) - w(\{x, y\}) + w(\{u, v\}) \leq w(T)$$

# Algorithmendesign



Bemerkung: beide Algorithmen funktionieren auch für negative Kantengewichte

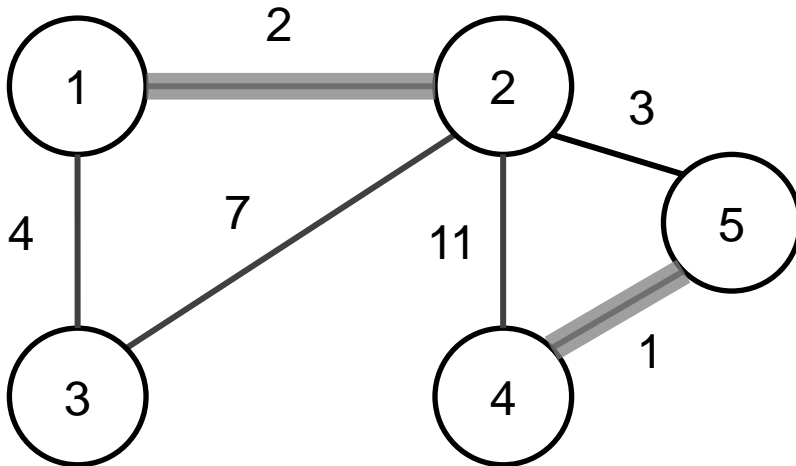
# Algorithmus von Kruskal

$\text{UNION}(G, u, v)$  setzt  $\text{set}(w) = \text{set}(u) \cup \text{set}(v)$   
für alle Knoten  $w \in \text{set}(u) \cup \text{set}(v)$

**MST-Kruskal** ( $G, w$ ) //  $G=(V, E)$  undirected, connected graph  
                                   $w$  weight function

```
1  A = ∅
2  FOREACH v in V DO set(v) = {v};
3  Sort edges according to weight in nondecreasing order
4  FOREACH {u,v} in E according to order DO
5      IF set(u) != set(v) THEN
6          A = A ∪ {{u,v}}
7          UNION(G, u, v);
8  return A
```

wenn  $\text{set}(u) == \text{set}(v)$ ,  
dann wären Knoten schon  
verbunden und  $\{u, v\}$  erzeugte Zyklus



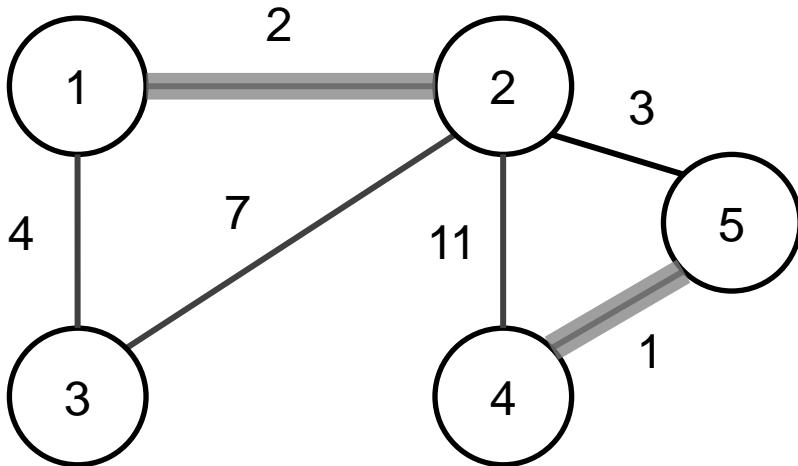
Zu jedem Knoten  $v$  sei  $\text{set}(v)$  Menge  
von mit  $v$  durch  $A$  verbundenen Knoten.  
Zu Beginn ist  $\text{set}(v) = \{v\}$ .

$\text{set}(u), \text{set}(v)$  sind disjunkt oder identisch

Im Beispiel  $\text{set}(1) = \{1, 2\}, \text{set}(4) = \{4, 5\}$ .

# Algorithmus von Kruskal: Korrektheit

```
MST-Kruskal( $G, w$ ) //  $G=(V, E)$  undirected, connected graph  
                       $w$  weight function  
1   $A = \emptyset$   
2  FOREACH  $v$  in  $V$  DO  $\text{set}(v) = \{v\};$   
3  Sort edges according to weight in nondecreasing order  
4  FOREACH  $\{u, v\}$  in  $E$  according to order DO  
5      IF  $\text{set}(u) \neq \text{set}(v)$  THEN  
6           $A = A \cup \{u, v\}$   
7          UNION( $G, u, v$ );  
8  return  $A$ 
```



Jede ausgewählte Kante  $\{u, v\}$  mit  $\text{set}(u) \neq \text{set}(v)$  ist leicht für Schnitt  $(\text{set}(u), V - \text{set}(u))$ .

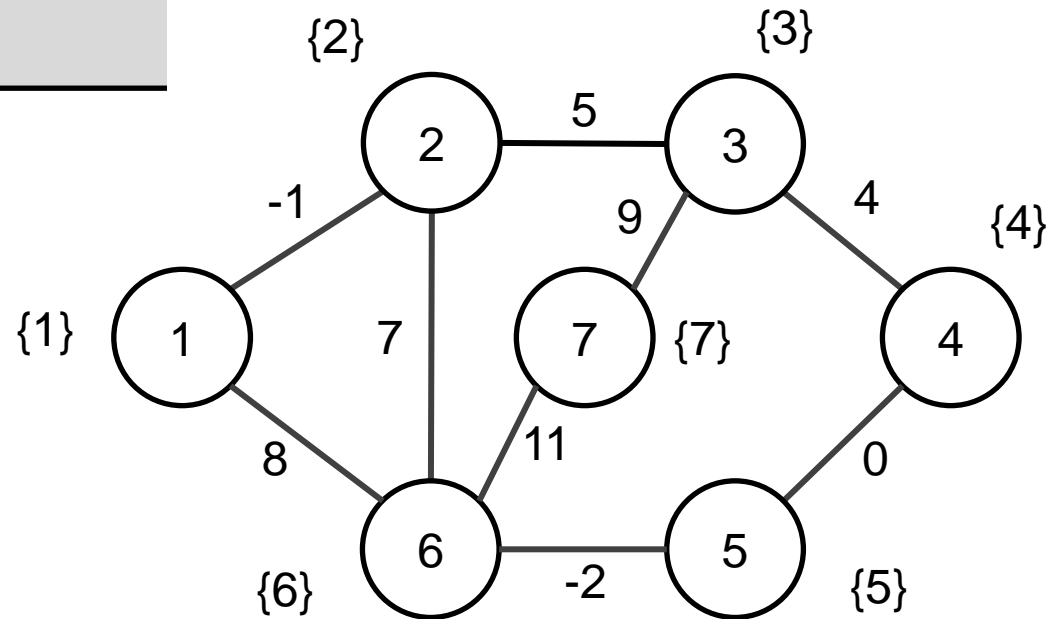
Schnitt respektiert **A**.

Somit Kante auch sicher für **A**.

# Algorithmus von Kruskal: Beispiel (I)

MST-Kruskal( $G, w$ ) //  $G=(V, E)$  undirected, connected graph  
                                   $w$  weight function

```
1  A = ∅
2  FOREACH v in V DO set(v) = {v};
3  Sort edges according to weight in nondecreasing order
4  FOREACH {u,v} in E according to order DO
5      IF set(u) ≠ set(v) THEN
6          A = A ∪ {{u,v}}
7          UNION(G, u, v);
8  return A
```



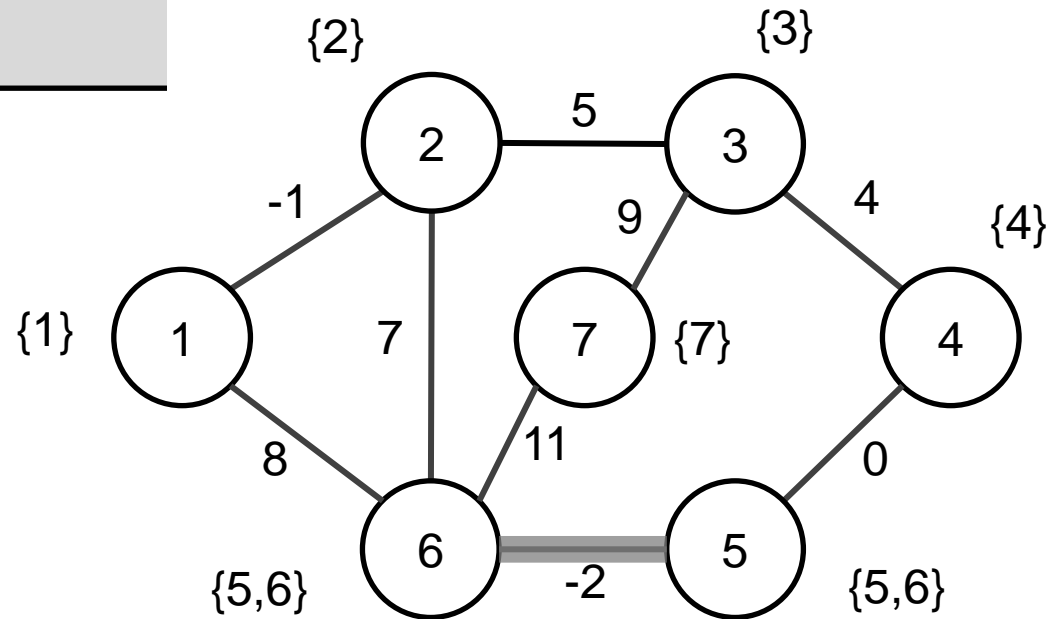
Initialisierung (1–3)

# Algorithmus von Kruskal: Beispiel (II)

MST-Kruskal( $G, w$ ) //  $G=(V, E)$  undirected, connected graph  
                                   $w$  weight function

```
1  A = ∅
2  FOREACH v in V DO set(v) = {v};
3  Sort edges according to weight in nondecreasing order
4  FOREACH {u,v} in E according to order DO
5      IF set(u) ≠ set(v) THEN
6          A = A ∪ {{u,v}}
7          UNION(G, u, v);
8  return A
```

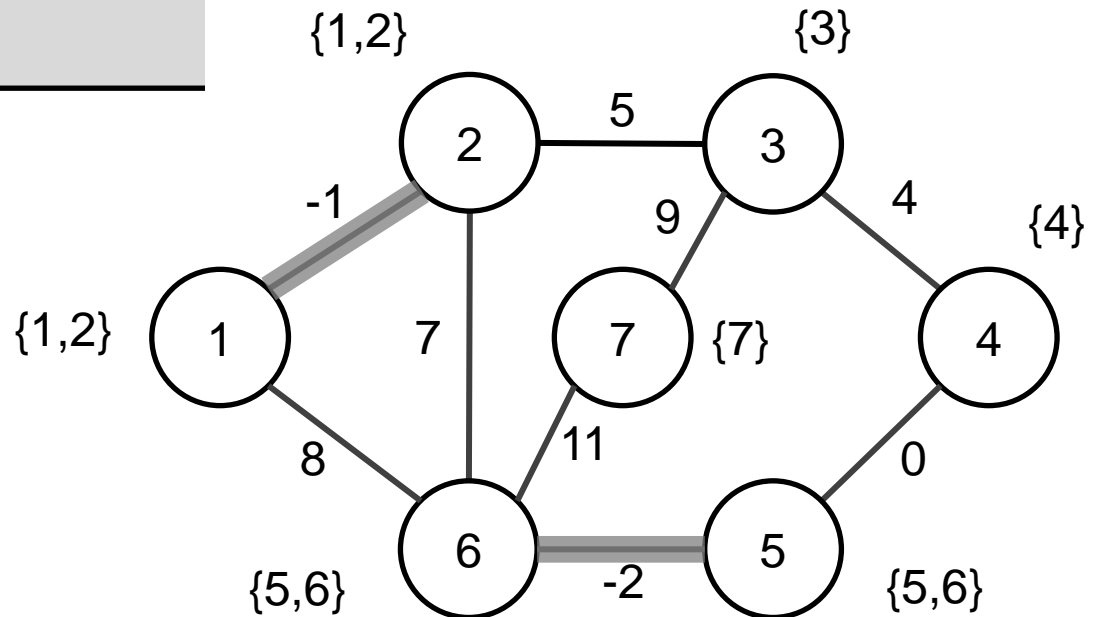
Schritte 4–8:  
Kante {5,6} aufnehmen



# Algorithmus von Kruskal: Beispiel (III)

MST-Kruskal( $G, w$ ) //  $G=(V, E)$  undirected, connected graph  
                                   $w$  weight function

```
1  A=∅
2  FOREACH v in V DO set(v)={v};
3  Sort edges according to weight in nondecreasing order
4  FOREACH {u,v} in E according to order DO
5      IF set(u)≠set(v) THEN
6          A = A ∪ {{u,v}}
7          UNION(G, u, v);
8  return A
```



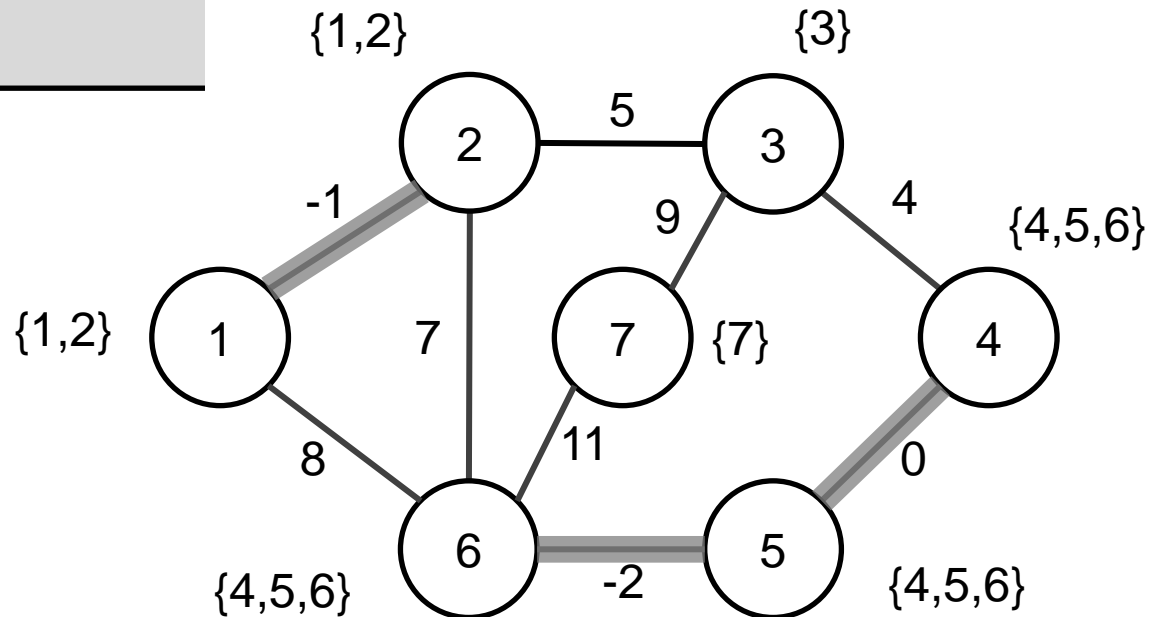
Schritte 4–8:  
Kante {1,2} aufnehmen

# Algorithmus von Kruskal: Beispiel (IV)

```
MST-Kruskal( $G, w$ ) //  $G=(V, E)$  undirected, connected graph  
                 $w$  weight function
```

```
1   $A = \emptyset$   
2  FOREACH  $v$  in  $V$  DO  $\text{set}(v) = \{v\}$ ;  
3  Sort edges according to weight in nondecreasing order  
4  FOREACH  $\{u, v\}$  in  $E$  according to order DO  
5      IF  $\text{set}(u) \neq \text{set}(v)$  THEN  
6           $A = A \cup \{\{u, v\}\}$   
7          UNION( $G, u, v$ ) ;  
8  return  $A$ 
```

Schritte 4–8:  
Kante  $\{4, 5\}$  aufnehmen



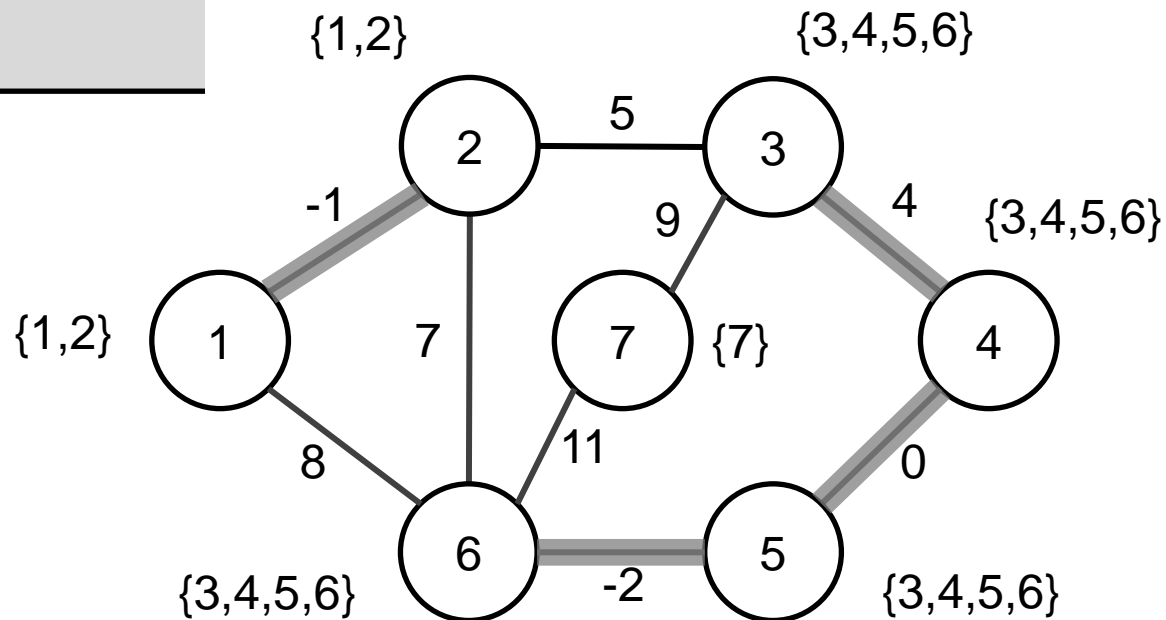


# Algorithmus von Kruskal: Beispiel (V)

```
MST-Kruskal( $G, w$ ) //  $G=(V, E)$  undirected, connected graph  
                     $w$  weight function
```

```
1   $A = \emptyset$   
2  FOREACH  $v$  in  $V$  DO  $\text{set}(v) = \{v\};$   
3  Sort edges according to weight in nondecreasing order  
4  FOREACH  $\{u, v\}$  in  $E$  according to order DO  
5      IF  $\text{set}(u) \neq \text{set}(v)$  THEN  
6           $A = A \cup \{\{u, v\}\}$   
7          UNION( $G, u, v$ );  
8  return  $A$ 
```

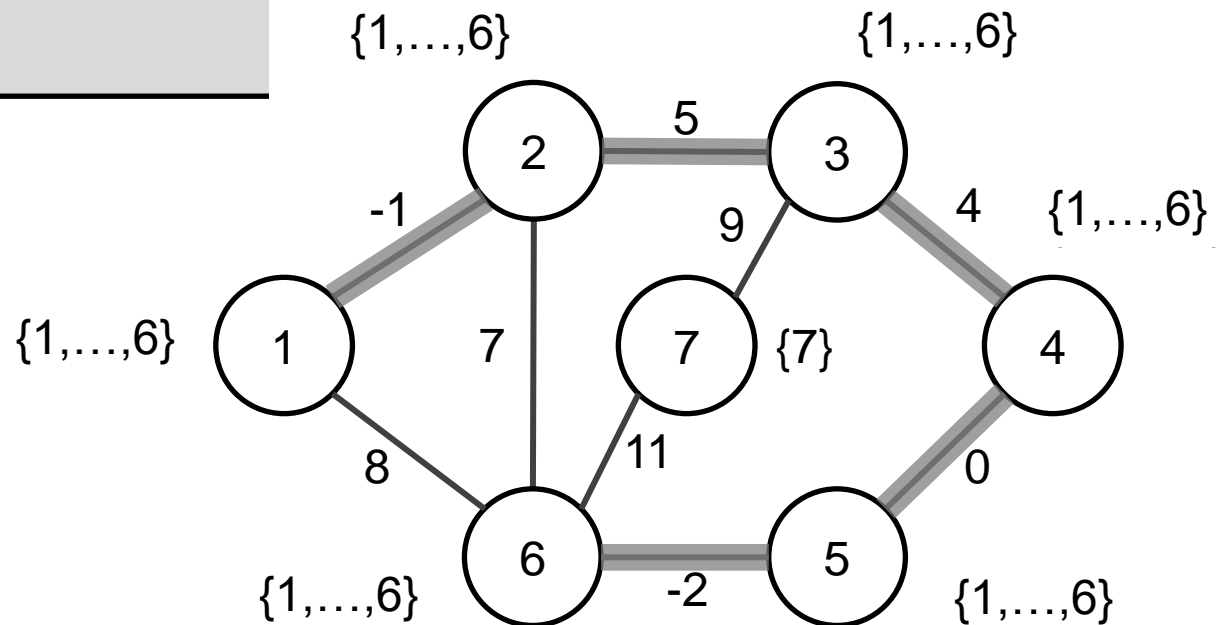
Schritte 4–8:  
Kante  $\{3, 4\}$  aufnehmen



# Algorithmus von Kruskal: Beispiel (VI)

`MST-Kruskal(G,w) // G=(V,E) undirected, connected graph  
w weight function`

```
1  A=∅
2  FOREACH v in V DO set(v)={v};
3  Sort edges according to weight in nondecreasing order
4  FOREACH {u,v} in E according to order DO
5      IF set(u)≠set(v) THEN
6          A = A ∪ {{u,v}}
7          UNION(G,u,v);
8  return A
```

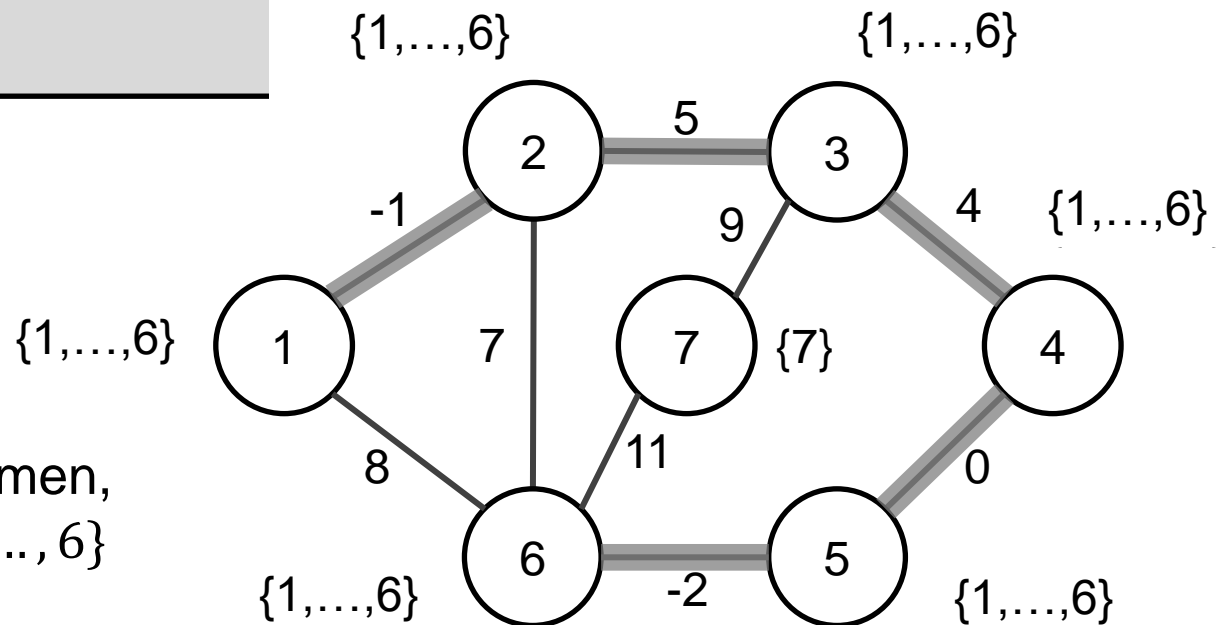


Schritte 4–8:  
Kante {2,3} aufnehmen

# Algorithmus von Kruskal: Beispiel (VII)

MST-Kruskal( $G, w$ ) //  $G=(V, E)$  undirected, connected graph  
                                   $w$  weight function

```
1  A = ∅
2  FOREACH v in V DO set(v) = {v};
3  Sort edges according to weight in nondecreasing order
4  FOREACH {u,v} in E according to order DO
5      IF set(u) ≠ set(v) THEN
6          A = A ∪ {{u,v}}
7          UNION(G, u, v);
8  return A
```



Schritte 4–8:

Kante  $\{2,6\}$  **nicht** aufnehmen,  
da  $set(2) = set(6) = \{1, \dots, 6\}$

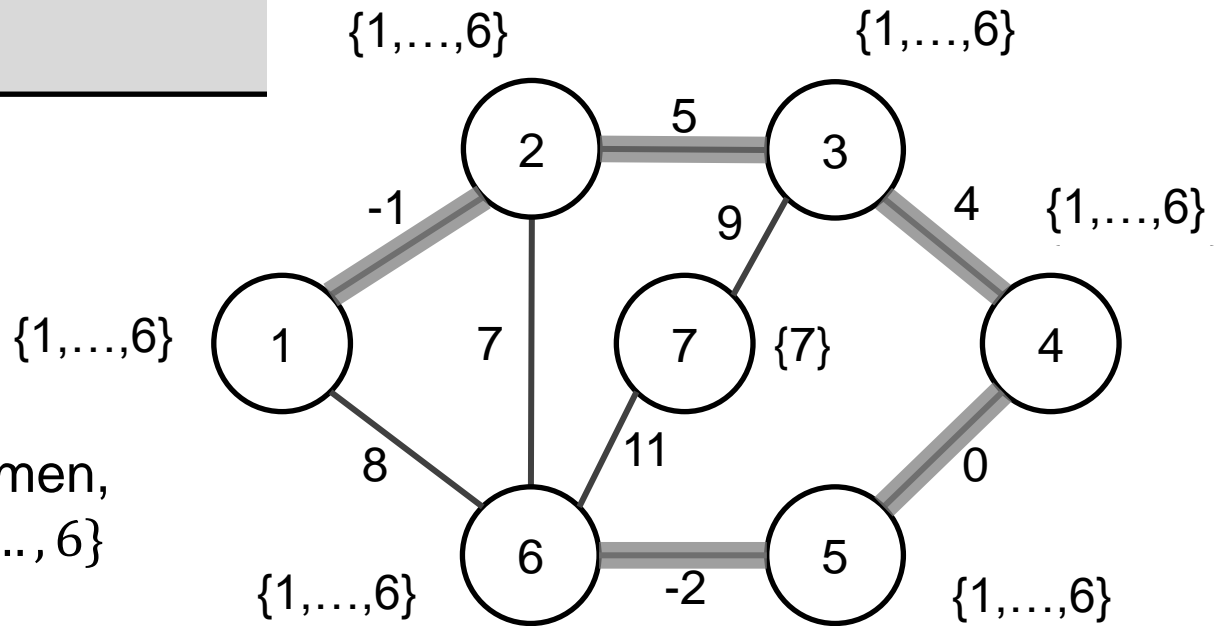
## Algorithmus von Kruskal: Beispiel (VIII)

```
MST-Kruskal(G,w) // G=(V,E) undirected, connected graph
                  w weight function
```

```

1  A = ∅
2  FOREACH v in V DO set(v) = {v};
3  Sort edges according to weight in nondecreasing order
4  FOREACH {u,v} in E according to order DO
5      IF set(u) ≠ set(v) THEN
6          A = A ∪ {{u,v}}
7          UNION(G, u, v);
8  return A

```



## Schritte 4–8:

Kante  $\{1,6\}$  **nicht** aufnehmen,  
da  $set(1) = set(6) = \{1, \dots, 6\}$

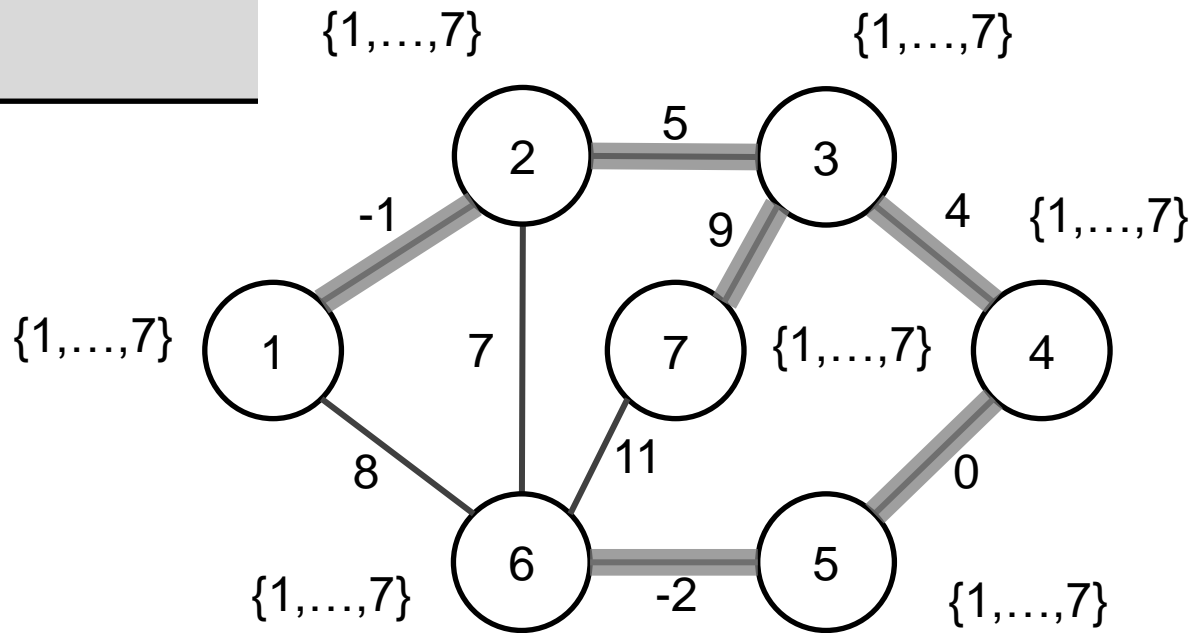
## Algorithmus von Kruskal: Beispiel (IX)

```
MST-Kruskal(G,w) // G=(V,E) undirected, connected graph
                  w weight function
```

```

1  A = ∅
2  FOREACH v in V DO set(v) = {v};
3  Sort edges according to weight in nondecreasing order
4  FOREACH {u,v} in E according to order DO
5      IF set(u) ≠ set(v) THEN
6          A = A ∪ {{u,v}}
7          UNION(G, u, v);
8  return A

```



Schritte 4–8:  
Kante {3,7} aufnehmen

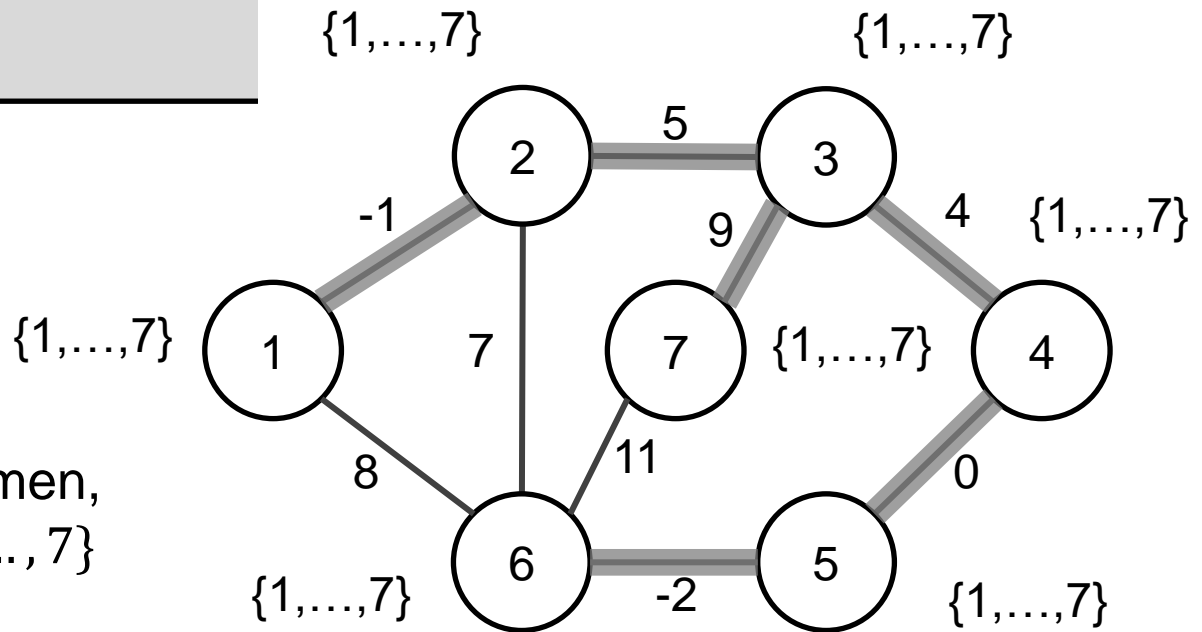
## Algorithmus von Kruskal: Beispiel (X)

**MST-Kruskal** ( $G, w$ ) //  $G=(V, E)$  undirected, connected graph  
 $w$  weight function

```

1  A = ∅
2  FOREACH v in V DO set(v) = {v};
3  Sort edges according to weight in nondecreasing order
4  FOREACH {u,v} in E according to order DO
5      IF set(u) ≠ set(v) THEN
6          A = A ∪ {{u,v}}
7          UNION(G, u, v);
8  return A

```



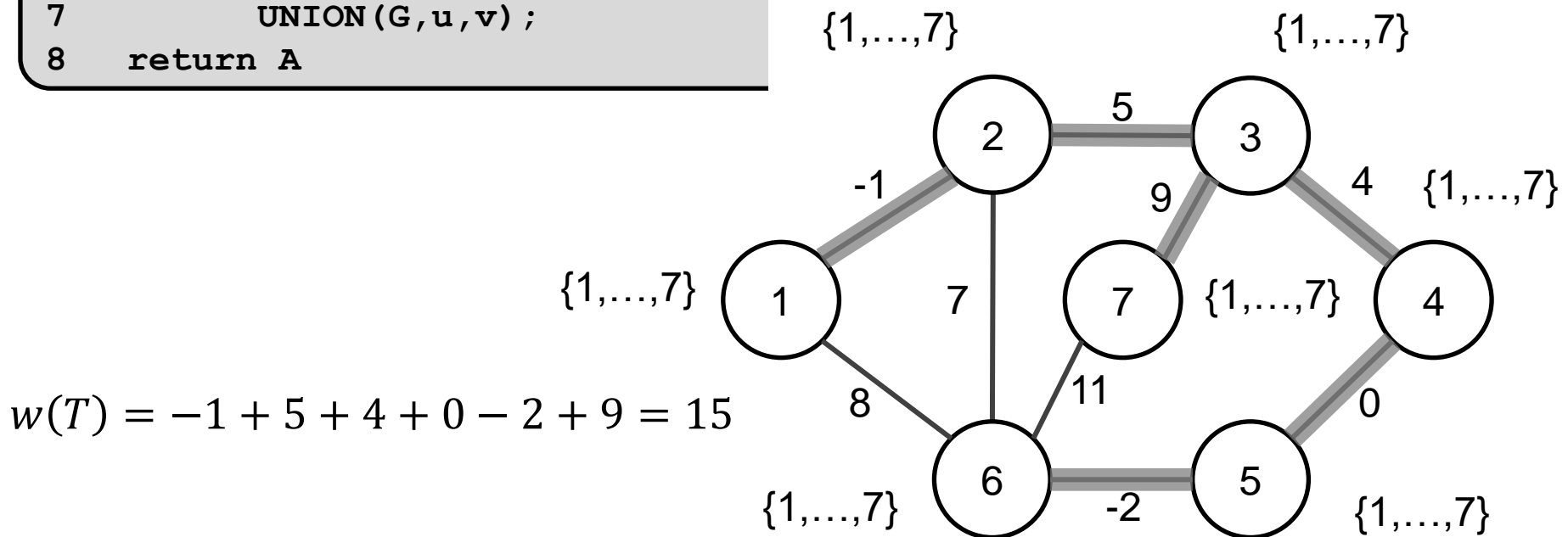
## Schritte 4–8:

Kante  $\{6,7\}$  **nicht** aufnehmen,  
da  $set(6) = set(7) = \{1, \dots, 7\}$

# Algorithmus von Kruskal: Beispiel (XI)

**MST-Kruskal**( $G, w$ ) //  $G=(V, E)$  undirected, connected graph  
                                   $w$  weight function

```
1  A = ∅
2  FOREACH v in V DO set(v) = {v};
3  Sort edges according to weight in nondecreasing order
4  FOREACH {u,v} in E according to order DO
5      IF set(u) ≠ set(v) THEN
6          A = A ∪ {{u,v}}
7          UNION(G, u, v);
8  return A
```



# Algorithmus von Kruskal: Laufzeit

```
MST-Kruskal(G,w) // G=(V,E) undirected, connected graph
                    w weight function

1  A=∅
2  FOREACH v in V DO set(v)={v};
3  Sort edges according to weight in nondecreasing order
4  FOREACH {u,v} in E according to order DO
5      IF set(u)≠set(v) THEN
6          A = A ∪ {{u,v}}
7          UNION(G,u,v);
8  return A
```

Laufzeit =  $O(|E| \cdot \log |E|)$

(mit vielen Optimierungen)

Laufzeit =  $O(|E| \cdot \log |V|)$

da  $|V| - 1 \leq |E| \leq |V|^2$  und  
somit  $\log |E| = \theta(\log |V|)$





Überlegen Sie sich anhand eines Beispielgraphen, dass ein MST zwar das Gesamtgewicht reduziert, aber nicht immer kurze Wege (#Kanten) garantiert.



Angenommen, Sie haben bereits einen MST berechnet, und reduzieren nachträglich das Gewicht einer Kante des MSTs. Wie ändert sich Ihr MST?

# Algorithmus von Prim

MST-Prim( $G, w, r$ ) //  $r$  root in  $V$ , MST given through  $v.pred$  values

```
1  FOREACH  $v$  in  $V$  DO { $v.key = \infty$ ;  $v.pred = NIL$ ;}
2   $r.key = -\infty$ ;  $Q = V$ ;
3  WHILE !isEmpty( $Q$ ) DO
4       $u = EXTRACT-MIN(Q)$ ; //smallest key value
5      FOREACH  $v$  in adj( $u$ ) DO
6          IF  $v \in Q$  and  $w(\{u, v\}) < v.key$  THEN
7               $v.key = w(\{u, v\})$ ;
8               $v.pred = u$ ;
```

Idee: Algorithmus fügt, beginnend mit Wurzelknoten, immer leichte Kante zu zusammenhängender Menge hinzu

Auswahl der nächsten Kante gemäß **key**-Wert, der stets aktualisiert wird

**A** implizit definiert durch

$$\mathbf{A} = \{ \{v, v.pred\} \mid v \in V - (\{r\} \cup Q) \}$$

# Algorithmus von Prim: Beispiel (I)

MST-Prim( $G, w, r$ ) //  $r$  root in  $V$ , MST given through  $v.\text{pred}$  values

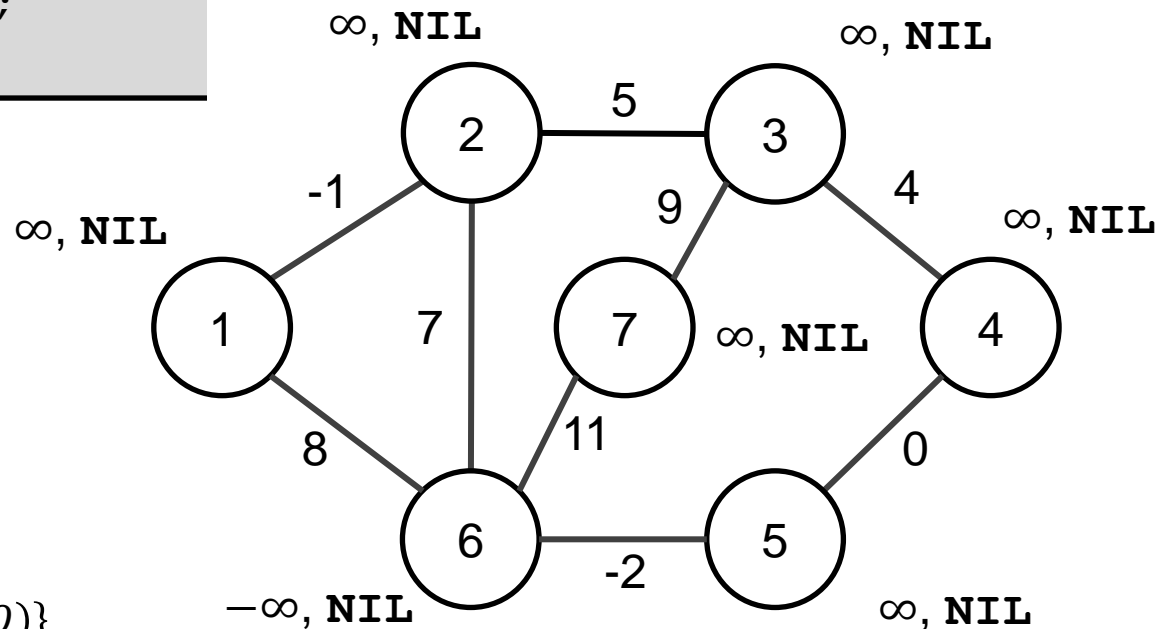
```
1  FOREACH  $v$  in  $V$  DO { $v.\text{key} = \infty$ ;  $v.\text{pred} = \text{NIL}$ ;}
2   $r.\text{key} = -\infty$ ;  $Q = V$ ;
3  WHILE !isEmpty( $Q$ ) DO
4     $u = \text{EXTRACT-MIN}(Q)$ ; //smallest key value
5    FOREACH  $v$  in adj( $u$ ) DO
6      IF  $v \in Q$  and  $w(\{u, v\}) < v.\text{key}$ 
7         $v.\text{key} = w(\{u, v\})$ ;
8         $v.\text{pred} = u$ ;
```

Initialisierung (1–2)

$r = 6$

$Q = \{1, 2, 3, 4, 5, 6, 7\}$

$A = \{ \{v, v.\text{pred}\} \mid v \in V - (\{r\} \cup Q) \}$



# Algorithmus von Prim: Beispiel (II)

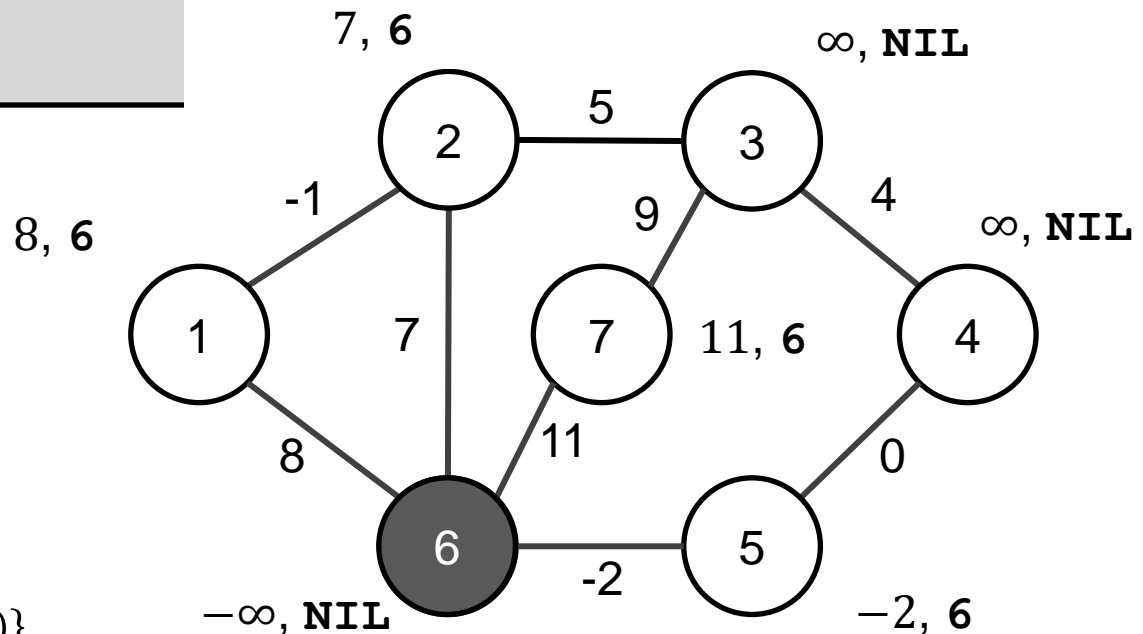
MST-Prim( $G, w, r$ ) //  $r$  root in  $V$ , MST given through  $v.\text{pred}$  values

```
1  FOREACH  $v$  in  $V$  DO { $v.\text{key} = \infty$ ;  $v.\text{pred} = \text{NIL}$ ;}
2   $r.\text{key} = -\infty$ ;  $Q = V$ ;
3  WHILE !isEmpty( $Q$ ) DO
4     $u = \text{EXTRACT-MIN}(Q)$ ; //smallest key value
5    FOREACH  $v$  in adj( $u$ ) DO
6      IF  $v \in Q$  and  $w(\{u, v\}) < v.\text{key}$ 
7         $v.\text{key} = w(\{u, v\})$ ;
8         $v.\text{pred} = u$ ;
```

Schritte **3–8**:  
 $u=r=6$  extrahieren

$Q = \{1, 2, 3, 4, 5, 7\}$

$A = \{ \{v, v.\text{pred}\} \mid v \in V - (\{r\} \cup Q) \}$



# Algorithmus von Prim: Beispiel (III)

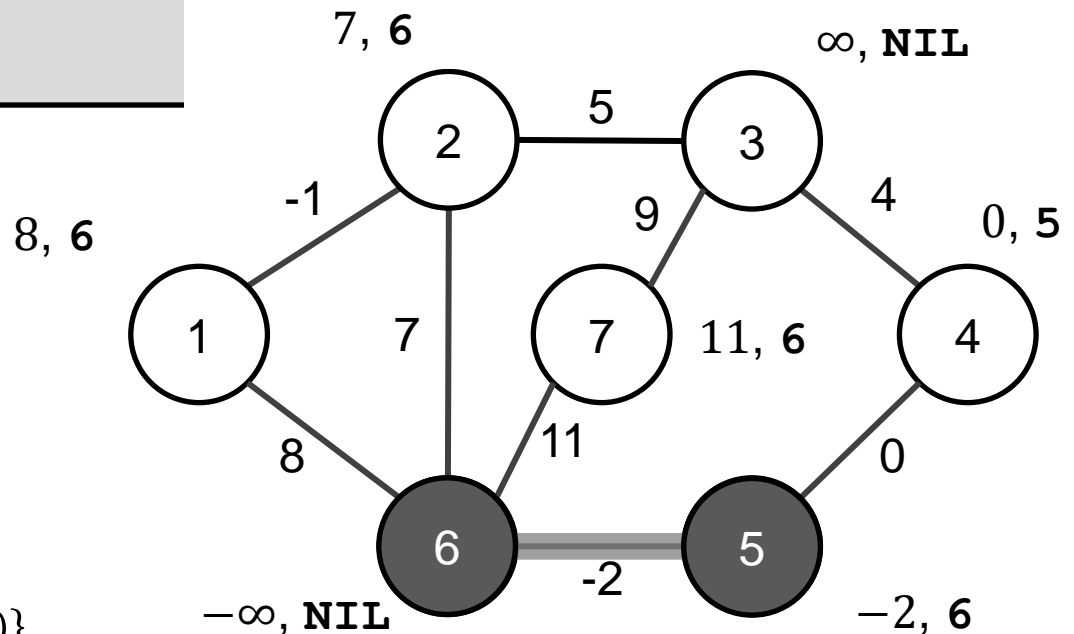
MST-Prim( $G, w, r$ ) //  $r$  root in  $V$ , MST given through  $v.\text{pred}$  values

```
1  FOREACH  $v$  in  $V$  DO { $v.\text{key} = \infty$ ;  $v.\text{pred} = \text{NIL}$ ;}
2   $r.\text{key} = -\infty$ ;  $Q = V$ ;
3  WHILE !isEmpty( $Q$ ) DO
4     $u = \text{EXTRACT-MIN}(Q)$ ; //smallest key value
5    FOREACH  $v$  in adj( $u$ ) DO
6      IF  $v \in Q$  and  $w(\{u, v\}) < v.\text{key}$ 
7         $v.\text{key} = w(\{u, v\})$ ;
8         $v.\text{pred} = u$ ;
```

Schritte **3–8**:  
 $u=5$  extrahieren

$Q = \{1, 2, 3, 4, 7\}$

$A = \{ \{v, v.\text{pred}\} \mid v \in V - (\{r\} \cup Q) \}$



# Algorithmus von Prim: Beispiel (IV)

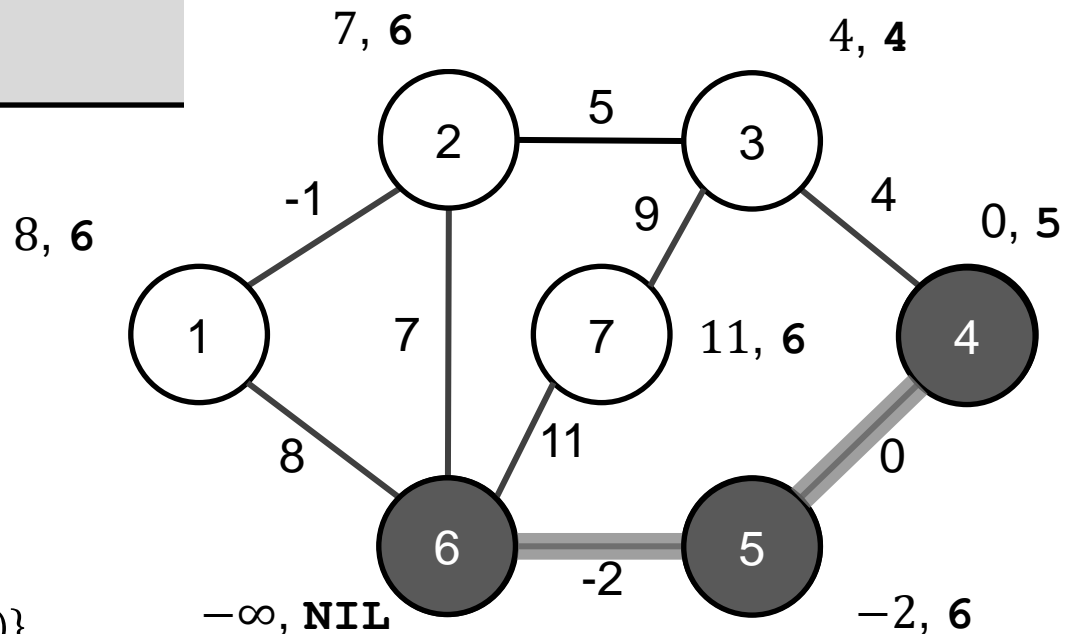
MST-Prim( $G, w, r$ ) //  $r$  root in  $V$ , MST given through  $v.\text{pred}$  values

```
1  FOREACH  $v$  in  $V$  DO { $v.\text{key} = \infty$ ;  $v.\text{pred} = \text{NIL}$ ;}
2   $r.\text{key} = -\infty$ ;  $Q = V$ ;
3  WHILE !isEmpty( $Q$ ) DO
4     $u = \text{EXTRACT-MIN}(Q)$ ; //smallest key value
5    FOREACH  $v$  in adj( $u$ ) DO
6      IF  $v \in Q$  and  $w(\{u, v\}) < v.\text{key}$ 
7         $v.\text{key} = w(\{u, v\})$ ;
8         $v.\text{pred} = u$ ;
```

Schritte **3–8**:  
 $u=4$  extrahieren

$Q = \{1, 2, 3, 7\}$

$A = \{ \{v, v.\text{pred}\} \mid v \in V - (\{r\} \cup Q) \}$



# Algorithmus von Prim: Beispiel (V)

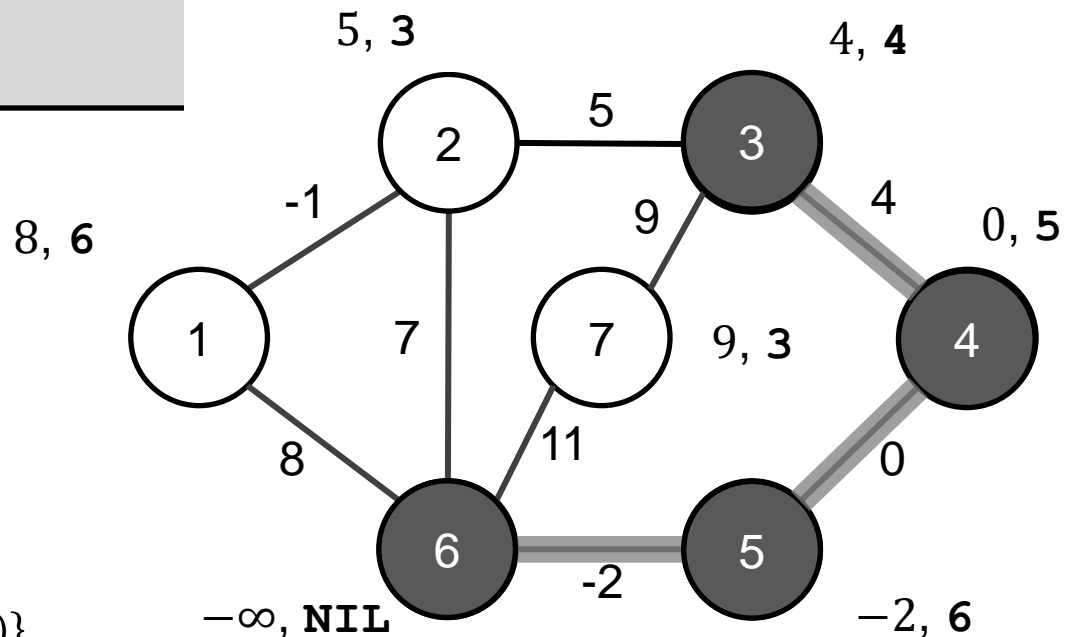
MST-Prim( $G, w, r$ ) //  $r$  root in  $V$ , MST given through  $v.\text{pred}$  values

```
1  FOREACH  $v$  in  $V$  DO { $v.\text{key} = \infty$ ;  $v.\text{pred} = \text{NIL}$ ;}
2   $r.\text{key} = -\infty$ ;  $Q = V$ ;
3  WHILE !isEmpty( $Q$ ) DO
4     $u = \text{EXTRACT-MIN}(Q)$ ; //smallest key value
5    FOREACH  $v$  in adj( $u$ ) DO
6      IF  $v \in Q$  and  $w(\{u, v\}) < v.\text{key}$ 
7         $v.\text{key} = w(\{u, v\})$ ;
8         $v.\text{pred} = u$ ;
```

Schritte **3–8**:  
 $u=3$  extrahieren

$Q = \{1, 2, 7\}$

$A = \{ \{v, v.\text{pred}\} \mid v \in V - (\{r\} \cup Q) \}$



# Algorithmus von Prim: Beispiel (VI)

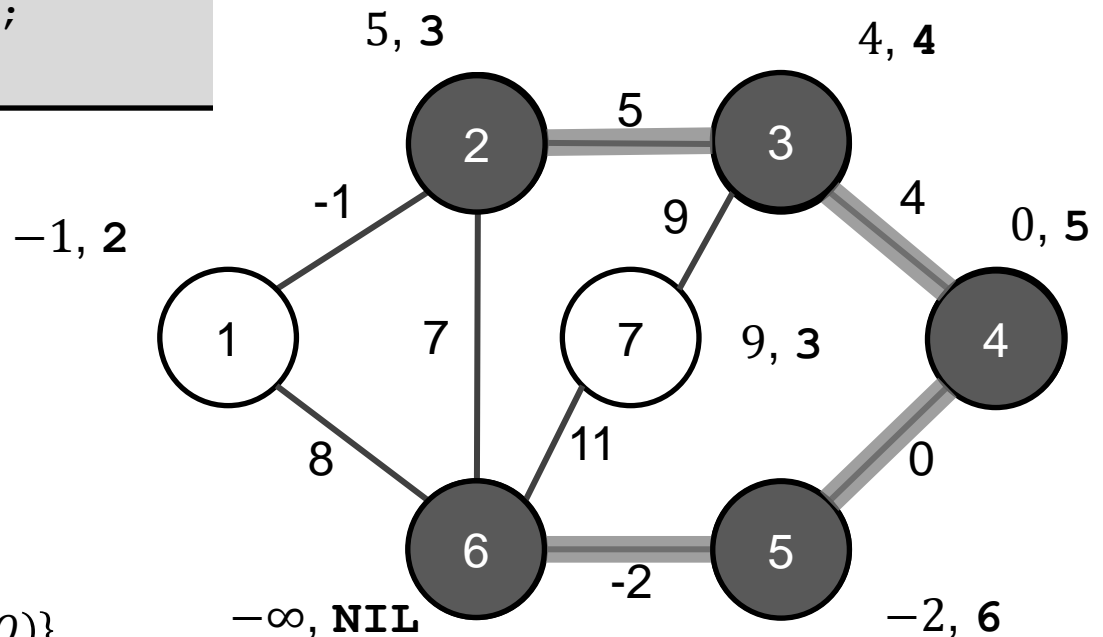
MST-Prim( $G, w, r$ ) //  $r$  root in  $V$ , MST given through  $v.\text{pred}$  values

```
1  FOREACH  $v$  in  $V$  DO { $v.\text{key} = \infty$ ;  $v.\text{pred} = \text{NIL}$ ;}
2   $r.\text{key} = -\infty$ ;  $Q = V$ ;
3  WHILE !isEmpty( $Q$ ) DO
4     $u = \text{EXTRACT-MIN}(Q)$ ; //smallest key value
5    FOREACH  $v$  in adj( $u$ ) DO
6      IF  $v \in Q$  and  $w(\{u, v\}) < v.\text{key}$ 
7         $v.\text{key} = w(\{u, v\})$ ;
8         $v.\text{pred} = u$ ;
```

Schritte **3–8**:  
 $u=2$  extrahieren

$Q = \{1, 7\}$

$A = \{ \{v, v.\text{pred}\} \mid v \in V - (\{r\} \cup Q) \}$





# Algorithmus von Prim: Beispiel (VII)

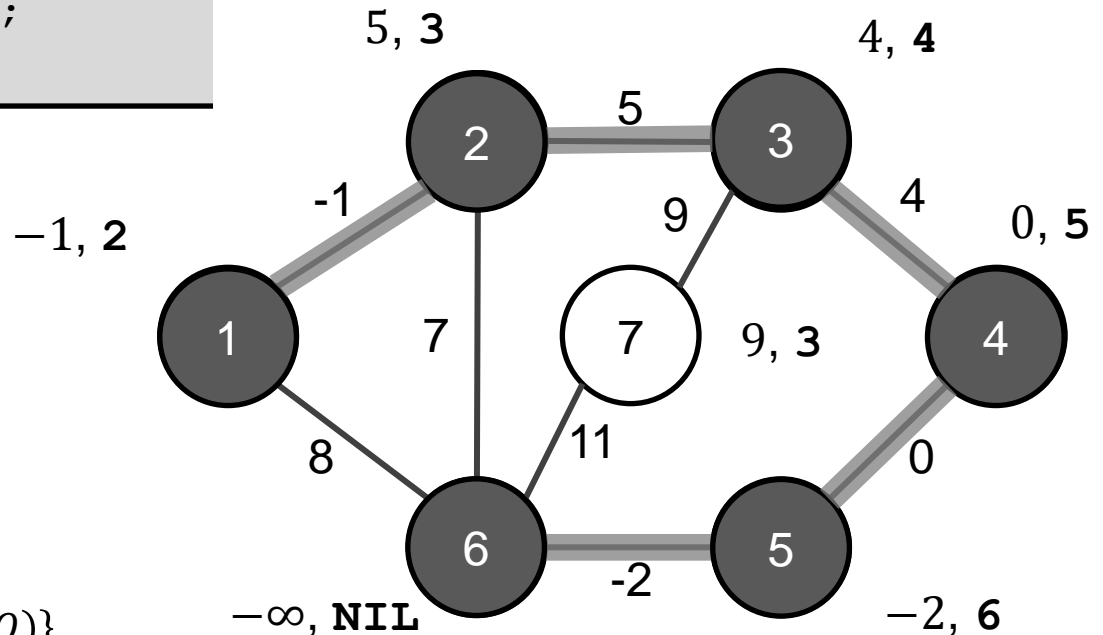
MST-Prim( $G, w, r$ ) //  $r$  root in  $V$ , MST given through  $v.\text{pred}$  values

```
1  FOREACH  $v$  in  $V$  DO { $v.\text{key} = \infty$ ;  $v.\text{pred} = \text{NIL}$ ;}
2   $r.\text{key} = -\infty$ ;  $Q = V$ ;
3  WHILE !isEmpty( $Q$ ) DO
4     $u = \text{EXTRACT-MIN}(Q)$ ; //smallest key value
5    FOREACH  $v$  in adj( $u$ ) DO
6      IF  $v \in Q$  and  $w(\{u, v\}) < v.\text{key}$ 
7         $v.\text{key} = w(\{u, v\})$ ;
8         $v.\text{pred} = u$ ;
```

Schritte **3–8**:  
 $u=1$  extrahieren

$Q = \{7\}$

$A = \{ \{v, v.\text{pred}\} \mid v \in V - (\{r\} \cup Q) \}$



# Algorithmus von Prim: Beispiel (VIII)

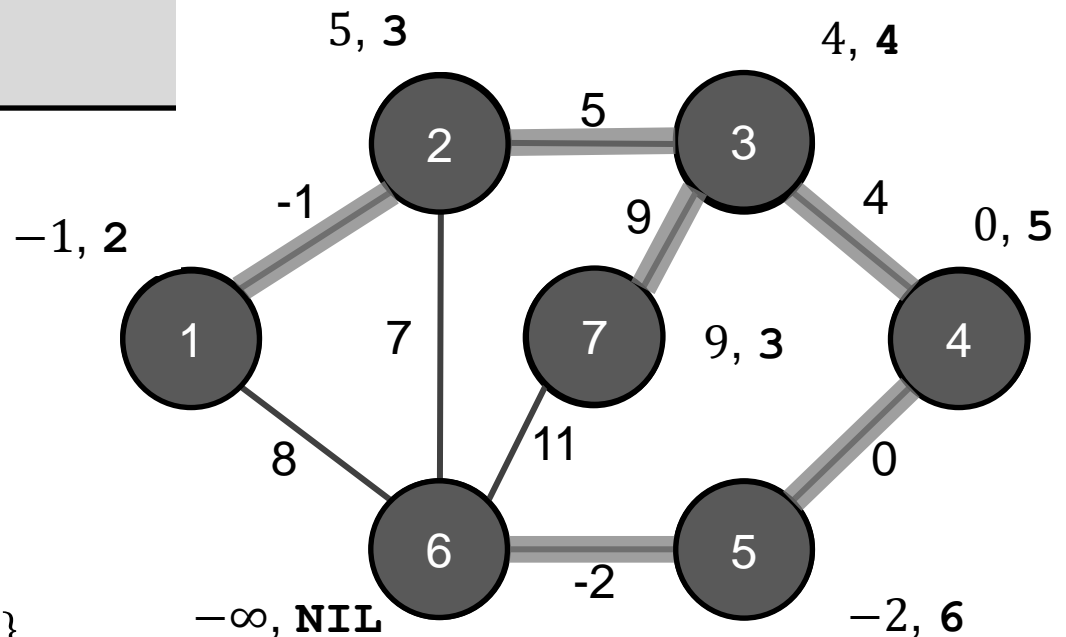
MST-Prim( $G, w, r$ ) //  $r$  root in  $V$ , MST given through  $v.\text{pred}$  values

```
1  FOREACH  $v$  in  $V$  DO { $v.\text{key} = \infty$ ;  $v.\text{pred} = \text{NIL}$ ;}
2   $r.\text{key} = -\infty$ ;  $Q = V$ ;
3  WHILE !isEmpty( $Q$ ) DO
4     $u = \text{EXTRACT-MIN}(Q)$ ; //smallest key value
5    FOREACH  $v$  in adj( $u$ ) DO
6      IF  $v \in Q$  and  $w(\{u, v\}) < v.\text{key}$ 
7         $v.\text{key} = w(\{u, v\})$ ;
8         $v.\text{pred} = u$ ;
```

Schritte **3–8**:  
 $u=7$  extrahieren

$Q = \{\}$

$A = \{ \{v, v.\text{pred}\} \mid v \in V - (\{r\} \cup Q) \}$



# Algorithmus von Prim: Korrektheit

`MST-Prim( $G, w, r$ ) //  $r$  root in  $V$ , MST given through  $v.\text{pred}$  values`

```
1  FOREACH  $v$  in  $V$  DO { $v.\text{key} = \infty$ ;  $v.\text{pred} = \text{NIL}$ ;}
2   $r.\text{key} = -\infty$ ;  $Q = V$ ;
3  WHILE !isEmpty( $Q$ ) DO
4     $u = \text{EXTRACT-MIN}(Q)$ ; //smallest key value
5    FOREACH  $v$  in adj( $u$ ) DO
6      IF  $v \in Q$  and  $w(\{u, v\}) < v.\text{key}$  THEN
7         $v.\text{key} = w(\{u, v\})$ ;
8         $v.\text{pred} = u$ ;
```

Kanten in  $A$  laufen nur  
zwischen den bereits  
aufgesammelten Knoten in  $V - Q$

Folglich respektiert der  
Schnitt  $(Q, V - Q)$  die Menge  $A$

Alle Knoten  $v \in Q$  enthalten als Wert  
 $v.\text{key}$  immer das kleinste Kantengewicht  
zu einem bereits aufgesammelten  
Knoten  $v.\text{pred}$  in  $V - Q$

Daher beschreibt der in Schritt 4  
ausgewählte Knoten  $u$  eine  
überbrückende, leichte Kante  $(u, u.\text{pred})$

# Algorithmus von Prim: Laufzeit

MST-Prim( $G, w, r$ ) //  $r$  root in  $V$ , MST given through  $v.pred$  values

```
1  FOREACH  $v$  in  $V$  DO { $v.key = \infty$ ;  $v.pred = NIL$ ;}
2   $r.key = -\infty$ ;  $Q = V$ ;
3  WHILE !isEmpty( $Q$ ) DO
4       $u = EXTRACT-MIN(Q)$ ; //smallest key value
5      FOREACH  $v$  in adj( $u$ ) DO
6          IF  $v \in Q$  and  $w(\{u, v\}) < v.key$  THEN
7               $v.key = w(\{u, v\})$ ;
8               $v.pred = u$ ;
```

Laufzeit =  $O(|E| + |V| \cdot \log |V|)$

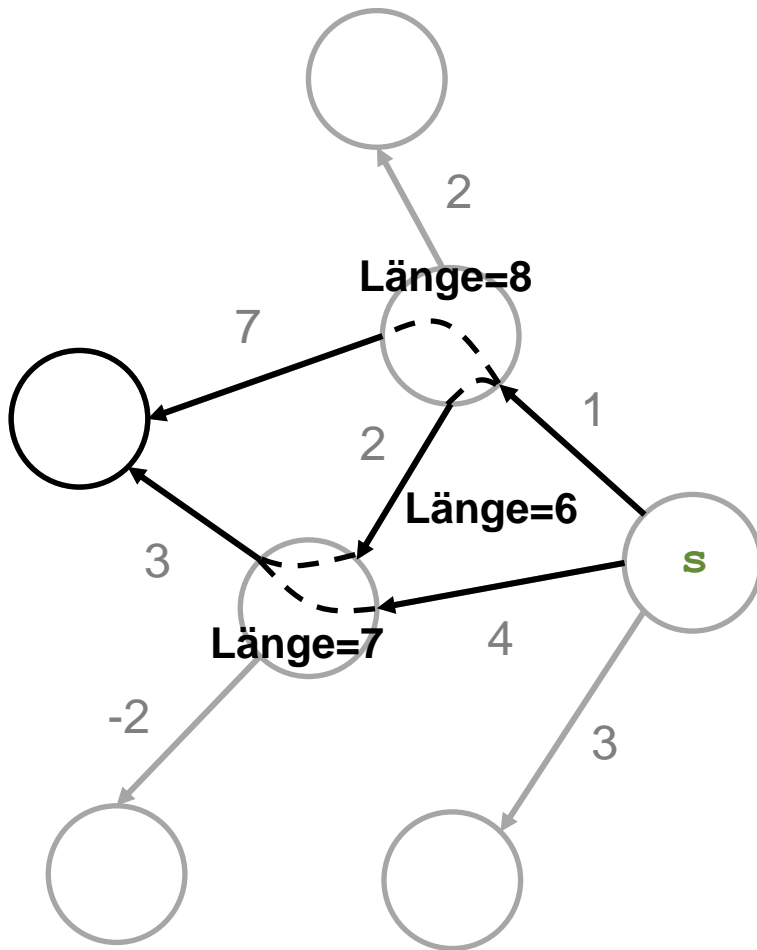
(mit vielen Optimierungen,  
speziell Fibonacci-Heaps)

zum Vergleich: Kruskal  
 $O(|E| \cdot \log |V|) = O(|E| \cdot \log |E|)$

# Kürzeste Wege in (gerichteten) Graphen

# Single-Source Shortest Path (SSSP)

Finde von Quelle  $s$  aus jeweils den (gemäß Kantengewichten) kürzesten Pfad zu allen anderen Knoten



Länge eines Pfades

$$p = (v_1, \dots, v_k) \in V^k$$

von  $u = v_1$  zu  $v = v_k$ :

$$w(p) = \sum_{i=1}^{k-1} w((v_i, v_{i+1}))$$

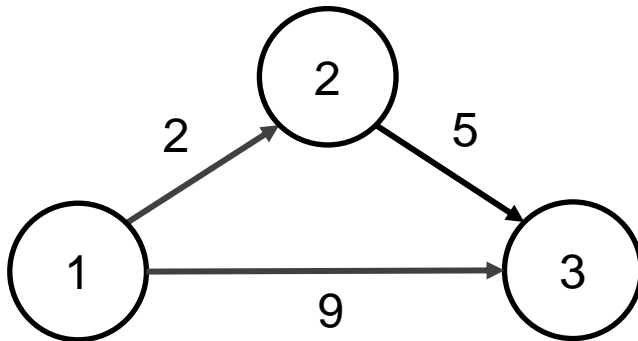
$$\text{shortest}(u, v) =$$

$$\begin{cases} \min\{w(p) : p \text{ Pfad von } u \text{ nach } v\} & \text{wenn } v \text{ erreichbar von } u \\ \infty & \text{sonst} \end{cases}$$

# SSSP vs. BFS, DFS, MST

BFS + DFS: keine Kantengewichte

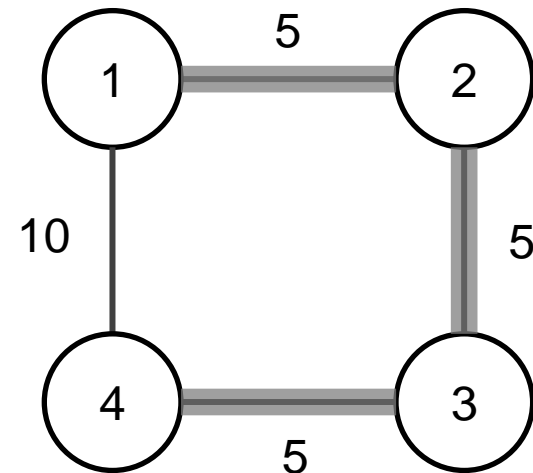
BFS findet kürzeste „Kantenwege“,  
aber nicht kürzeste „Gewichtsweg“:



Kürzester „Kantenweg“ 1 nach 3 =  $1 \rightarrow 3$   
Kürzester „Gewichtsweg“ =  $1 \rightarrow 2 \rightarrow 3$

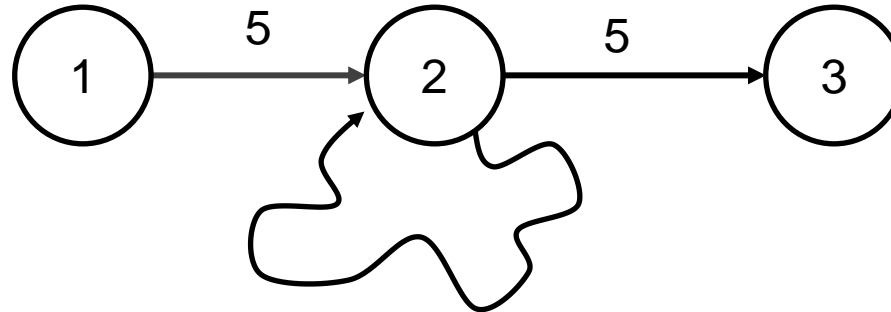
## MST

(für ungerichteten Graphen)  
minimiert *Gesamtgewicht*  
 $w(T) = \sum w(\{u, v\})$   
des Baumes



Kürzester Weg  $1 \rightarrow 4$  mit Gewicht 10 im MST nicht enthalten

# Negative Schwingungen



Zyklus mit Gesamtlänge =  $-5$

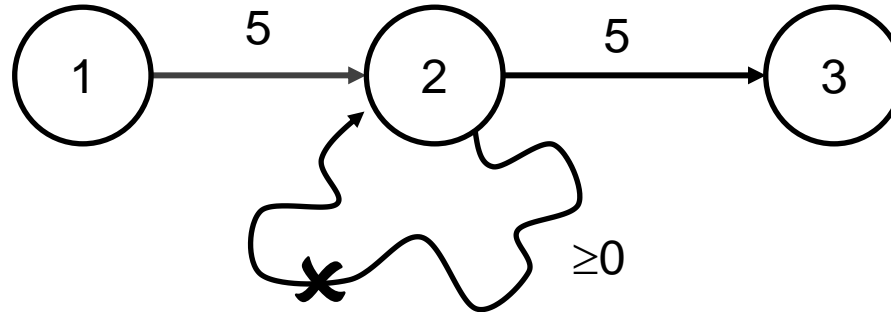
Wiederholtes Durchlaufen des Zyklus  
ergäbe beliebig kleine Gesamtlänge



**Negative Kantengewichte sind erlaubt,  
aber keine (erreichbaren) Zyklen mit negativem Gesamtgewicht!**



# Zyklen?



Kürzeste Pfade können keine Zyklen  
mit positivem Gesamtgewicht enthalten

(sonst ohne Zyklus kürzerer Pfad)

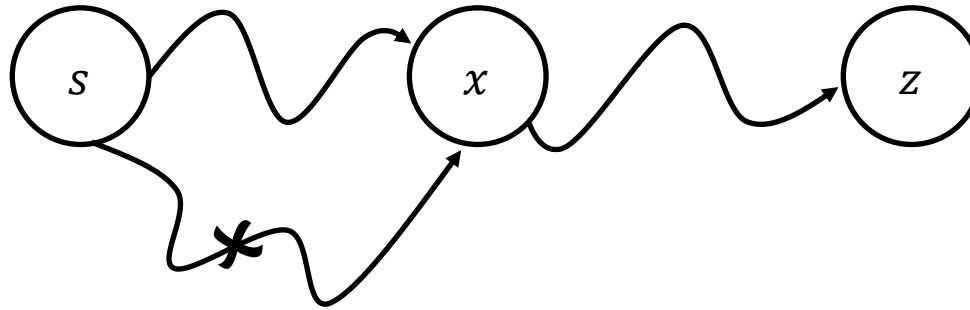


+Annahme über  
nicht-negative Zyklen

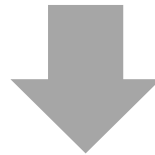
**Kürzeste Pfade enthalten höchstens (eliminierbare) Zyklen mit Gewicht 0**

**Es gibt stets einen kürzesten Pfad mit Kantenlänge  $\leq |V| - 1$**

# Kürzeste Teilpfade



Kürzester Pfad von  $s$  nach  $z$  durch Knoten  $x$



**Teilpfad  $s \rightarrow x$  eines kürzesten Pfades  $s \rightarrow x \rightarrow z$   
ist auch stets kürzester Pfad von  $s$  nach  $x$**

(sonst gäbe es kürzeren Pfad von  $s$  nach  $z$ )

# Algorithmen für SSSP

Gemeinsame Idee:  
„Lockerung“ / Relaxation

Bellmann 1958  
Ford 1962

Algorithmus  
von Bellman-Ford

*Laufzeit* =  $O(|V| \cdot |E|)$

*funktioniert  
allgemein*

*(auch ungerichtete Graphen)*

Lawler 1976

Algorithmus  
für dags

*Laufzeit* =  $O(|V| + |E|)$

*funktioniert nur für dags*

Dijkstra 1959

Algorithmus  
von Dijkstra

*Laufzeit*  
=  $O(|V| \cdot \log |V| + |E|)$

*funktioniert nur für  
nicht-negative  
Kantengewichte*

*(auch ungerichtete Graphen)*

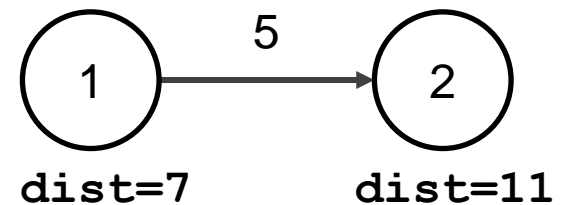
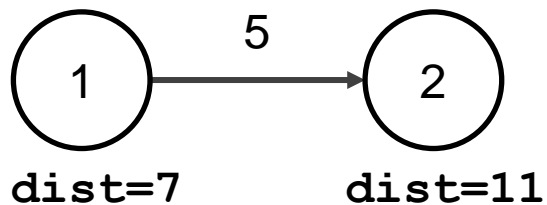
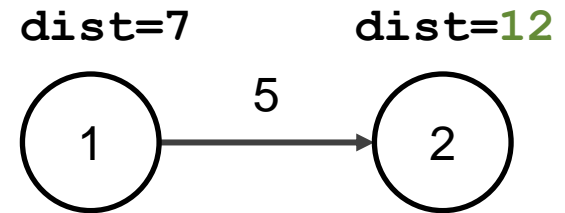
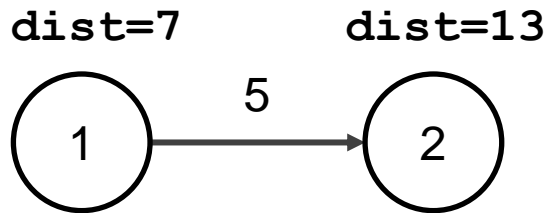
# Relax!

Idee: verringere aktuelle Distanz von Knoten  $v$ ,  
wenn durch Kante  $(u, v)$  kürzere Distanz erreichbar:

Zu Beginn  
Distanz =  $\infty$   
für alle Knoten  $\neq s$

```
relax(G, u, v, w)
```

```
1  IF v.dist > u.dist + w((u, v)) THEN
2      v.dist = u.dist + w((u, v));
3      v.pred = u;
```



# Bellman-Ford-Algorithmus

Bellman-Ford-SSSP( $G, s, w$ )

```
1  initSSSP( $G, s, w$ );
2  FOR  $i=1$  TO  $|V|-1$  DO
3      FOREACH  $(u, v)$  in  $E$  DO
4          relax( $G, u, v, w$ );
5  FOREACH  $(u, v)$  in  $E$  DO
6      IF  $v.dist > u.dist + w((u, v))$  THEN
7          return false;
8  return true;
```

initSSSP( $G, s, w$ )

```
1  FOREACH  $v$  in  $V$  DO
2       $v.dist = \infty$ ;
3       $v.pred = NIL$ ;
4   $s.dist = 0$ ;
```

prüft zusätzlich,  
ob „negativer Zyklus“  
erreichbar (=false)

Laufzeit =  $\theta(|E| \cdot |V|)$

wegen geschachtelter  
FOR-Schleifen in 2 und 3

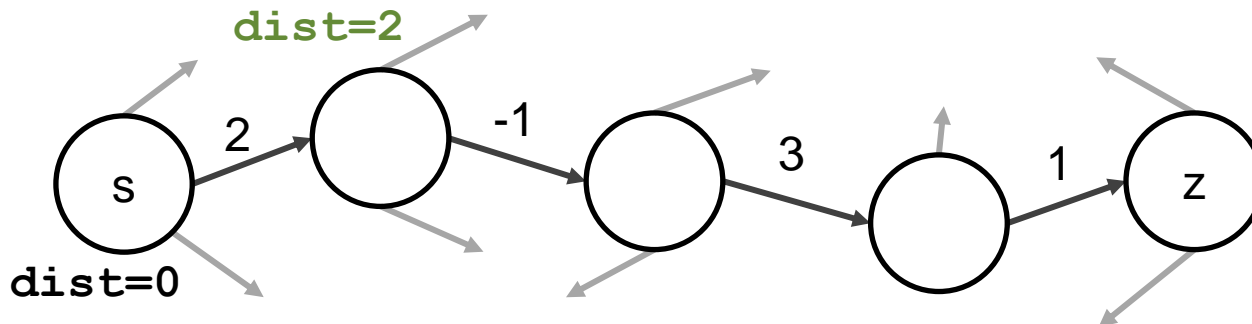
# Bellman-Ford: Idee / Korrektheit (I)

Bellman-Ford-SSSP ( $G, s, w$ )

```
1  initSSSP( $G, s, w$ ) ;
2  FOR  $i=1$  TO  $|V|-1$  DO
3      FOREACH ( $u, v$ ) in  $E$  DO
4          relax( $G, u, v, w$ ) ;
5  FOREACH ( $u, v$ ) in  $E$  DO
6      IF  $v.\text{dist} > u.\text{dist} + w((u, v))$  THEN
7          return false;
8  return true;
```

Betrachte Wirkung  
auf kürzesten Pfad  
von  $s$  nach  $z$

Erste Iteration der  
**FOR**-Schleife in 2  
erfasst (mindestens)  
den ersten Schritt  
von  $s$  zum  
nächsten Knoten



Wenn keine „negativen Zyklen“, gibt es  
kürzesten Pfad der Kantenlänge  $\leq |V| - 1$  ohne Schleifen

Andere Relaxation-  
Schritte (auch später)  
können dies  
nicht „zerstören“,  
da sonst  
kürzerer Pfad

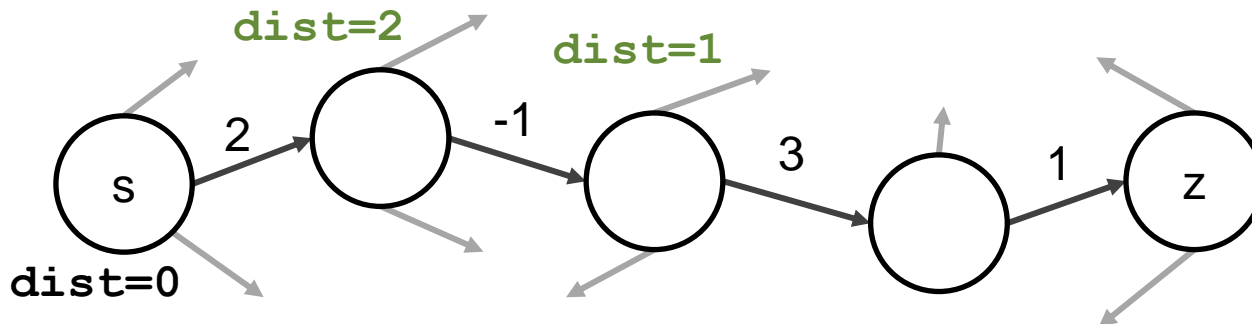
# Bellman-Ford: Idee / Korrektheit (II)

Bellman-Ford-SSSP ( $G, s, w$ )

```
1  initSSSP( $G, s, w$ ) ;
2  FOR  $i=1$  TO  $|V|-1$  DO
3      FOREACH ( $u, v$ ) in  $E$  DO
4          relax( $G, u, v, w$ ) ;
5  FOREACH ( $u, v$ ) in  $E$  DO
6      IF  $v.\text{dist} > u.\text{dist} + w((u, v))$  THEN
7          return false;
8  return true;
```

Betrachte Wirkung  
auf kürzesten Pfad  
von  $s$  nach  $z$

Zweite Iteration der  
**FOR**-Schleife in 2  
erfasst (mindestens)  
den zweiten Schritt  
von  $s$  zum  
zweiten Knoten



# Bellman-Ford: Idee / Korrektheit (III)

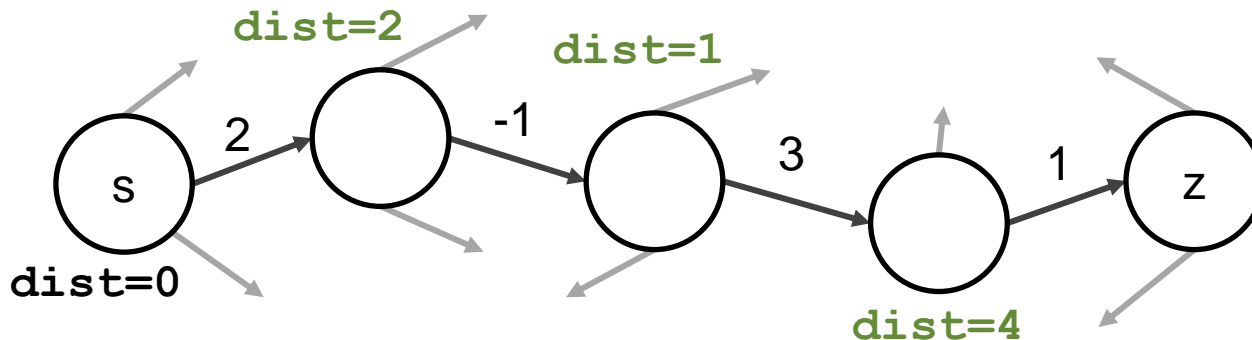
Bellman-Ford-SSSP ( $G, s, w$ )

```
1  initSSSP( $G, s, w$ ) ;
2  FOR  $i=1$  TO  $|V|-1$  DO
3      FOREACH ( $u, v$ ) in  $E$  DO
4          relax( $G, u, v, w$ ) ;
5  FOREACH ( $u, v$ ) in  $E$  DO
6      IF  $v.\text{dist} > u.\text{dist} + w((u, v))$  THEN
7          return false;
8  return true;
```

Betrachte Wirkung  
auf kürzesten Pfad  
von  $s$  nach  $z$

Dritte Iteration der  
**FOR**-Schleife in 2  
erfasst (mindestens)  
den dritten Schritt  
von  $s$  zum  
dritten Knoten

USW.





# Bellman-Ford: Idee / Korrektheit (IV)

Bellman-Ford-SSSP ( $G, s, w$ )

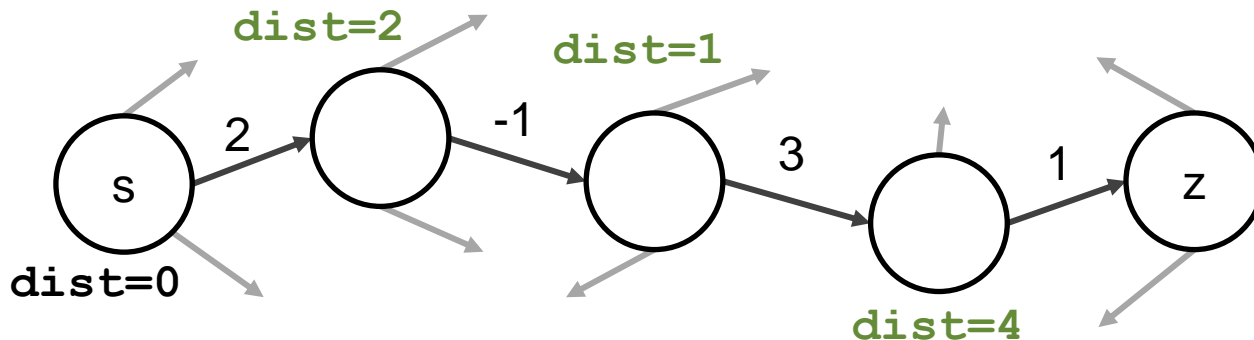
```
1  initSSSP( $G, s, w$ ) ;
2  FOR  $i=1$  TO  $|V|-1$  DO
3      FOREACH ( $u, v$ ) in  $E$  DO
4          relax( $G, u, v, w$ ) ;
5  FOREACH ( $u, v$ ) in  $E$  DO
6      IF  $v.\text{dist} > u.\text{dist} + w((u, v))$  THEN
7          return false;
8  return true;
```

Genau dann,  
wenn es keinen Pfad  
von  $s$  zu einem  
Knoten  $u$  gibt, dann  
bleibt  $u.\text{dist} = \infty$

Am Ende steht  
in jedem Knoten  
 $u.\text{dist} = \text{shortest}(s, u)$

und

$u.\text{pred}$  zeigt  
auf Vorgängerknoten  
in kürzestem Pfad



# Bellman-Ford: Idee / Korrektheit (V)

Bellman-Ford-SSSP ( $G, s, w$ )

```
1  initSSSP( $G, s, w$ ) ;
2  FOR  $i=1$  TO  $|V|-1$  DO
3      FOREACH ( $u, v$ ) in  $E$  DO
4          relax( $G, u, v, w$ ) ;
5  FOREACH ( $u, v$ ) in  $E$  DO
6      IF  $v.\text{dist} > u.\text{dist} + w((u, v))$  THEN
7          return false;
8  return true;
```

Prüfschritte 5–7

Fall 1:  
Wenn keine  
„negativen Zyklen“  
erreichbar, dann auch  
keine Rückgabe **false**

Wenn  $(u, v)$  Kante ist, dann ist

$$\begin{aligned} v.\text{dist} &= \text{shortest}(s, v) \\ &\leq \text{shortest}(s, u) + w((u, v)) \\ &\leq u.\text{dist} + w((u, v)) \end{aligned}$$

und folglich Bedingung in 6 nie erfüllt.

Gilt auch im Fall  
 $\text{shortest}(s, u) = \infty$ ;  
der Fall  $[\text{shortest}(s, v) = \infty$   
und  $\text{shortest}(s, u) < \infty]$   
ist nicht möglich.

# Bellman-Ford: Idee / Korrektheit (VI)

Bellman-Ford-SSSP ( $G, s, w$ )

```
1  initSSSP( $G, s, w$ ) ;
2  FOR  $i=1$  TO  $|V|-1$  DO
3      FOREACH ( $u, v$ ) in  $E$  DO
4          relax( $G, u, v, w$ ) ;
5  FOREACH ( $u, v$ ) in  $E$  DO
6      IF  $v.\text{dist} > u.\text{dist} + w((u, v))$  THEN
7          return false;
8  return true;
```

Prüfschritte 5–7

Fall 2:

Wenn „negative  
Zyklen“ erreichbar,  
dann Rückgabe  
**false**

$$v_0 = v_k$$

Sei  $c = (v_0, \dots, v_k) \in V^k$  „negativer Zyklus“ mit  $w(c) = \sum_{i=1}^k w((v_{i-1}, v_i)) < 0$

Verfahren gäbe nur **true**, wenn  $v_i.\text{dist} \leq v_{i-1}.\text{dist} + w((v_{i-1}, v_i))$  für  $i = 1, \dots, k$

Dann wäre wegen  $w(c) < 0$  und  $v_0 = v_k$  und  $v_i.\text{dist} < \infty$  für erreichbare Knoten:

$$\begin{aligned} \sum_{i=1}^k v_i.\text{dist} &\leq \sum_{i=1}^k v_{i-1}.\text{dist} + \sum_{i=1}^k w((v_{i-1}, v_i)) \\ &< \sum_{i=1}^k v_{i-1}.\text{dist} = \sum_{i=1}^k v_i.\text{dist} \end{aligned}$$

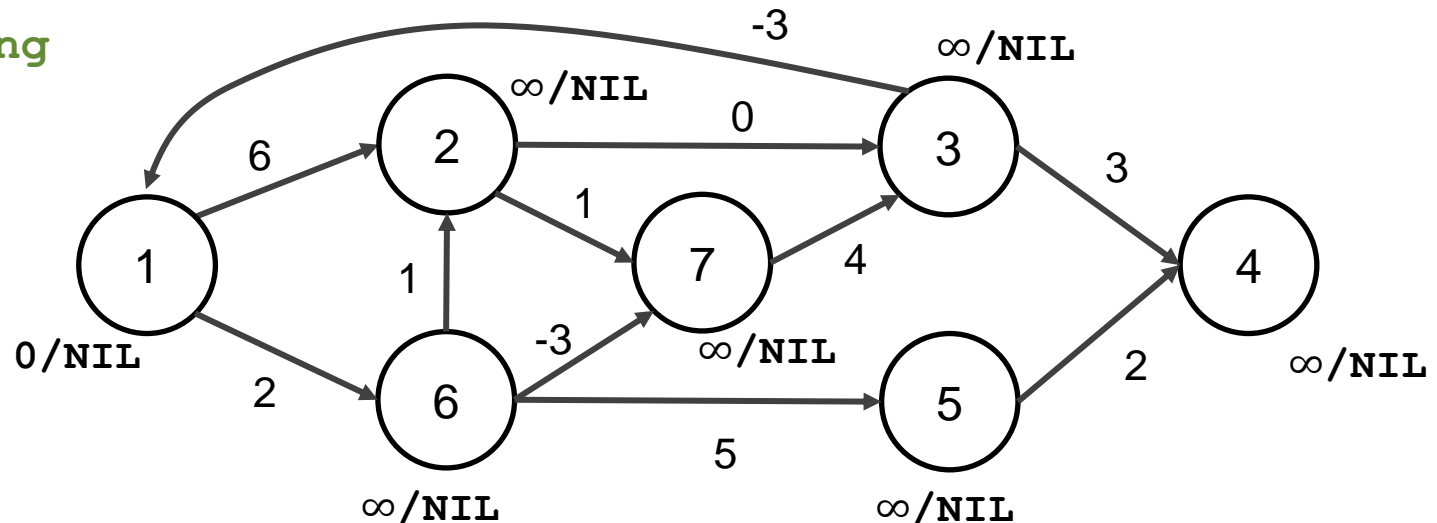
Widerspruch.

# Bellman-Ford: Beispiel (I)

Bellman-Ford-SSSP ( $G, s, w$ )

```
1  initSSSP( $G, s, w$ );
2  FOR  $i=1$  TO  $|V|-1$  DO
3      FOREACH ( $u, v$ ) in  $E$  DO
4          relax( $G, u, v, w$ );
5  FOREACH ( $u, v$ ) in  $E$  DO
6      IF  $v.\text{dist} > u.\text{dist} + w((u, v))$  THEN
7          return false;
8  return true;
```

Initialisierung  
für  $s=1$  in 1



# Bellman-Ford: Beispiel (II)

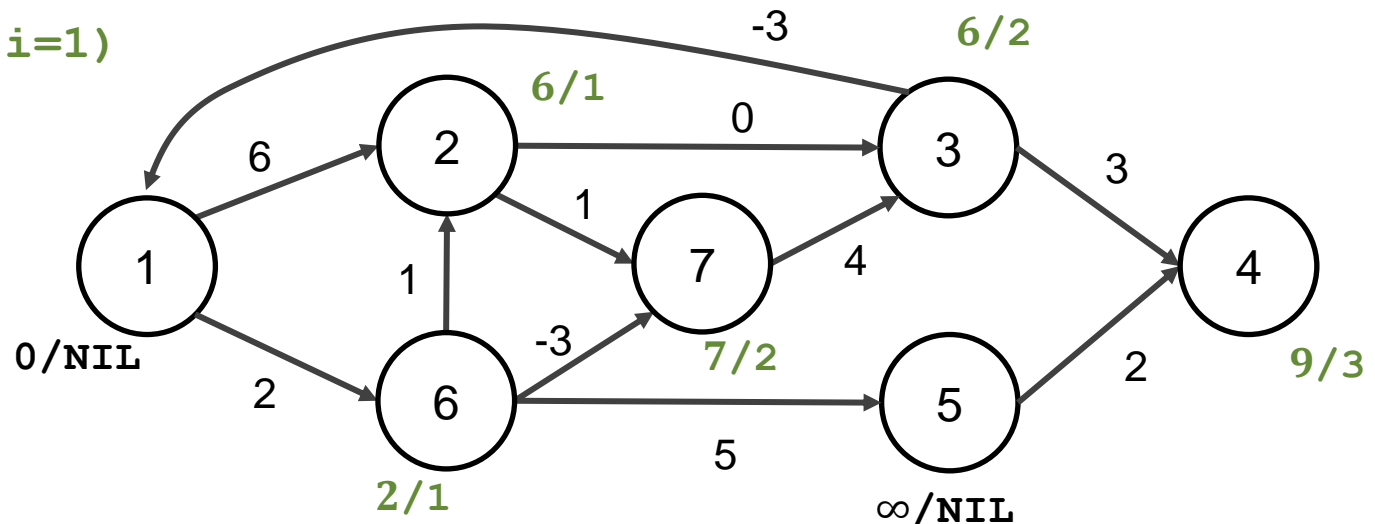
Bellman-Ford-SSSP ( $G, s, w$ )

```
1  initSSSP( $G, s, w$ );
2  FOR  $i=1$  TO  $|V|-1$  DO
3      FOREACH ( $u, v$ ) in  $E$  DO
4          relax( $G, u, v, w$ );
5  FOREACH ( $u, v$ ) in  $E$  DO
6      IF  $v.\text{dist} > u.\text{dist} + w((u, v))$  THEN
7          return false;
8  return true;
```

Kanten in **FOREACH** in 3  
gemäß lexikographischer  
Ordnung: (1,2), (1,6), (2,3), ...

FOR-Schleife 2 ( $i=1$ )

relax(1,2)  
relax(1,6)  
relax(2,3)  
relax(2,7)  
relax(3,1)  
relax(3,4)  
relax(5,4)



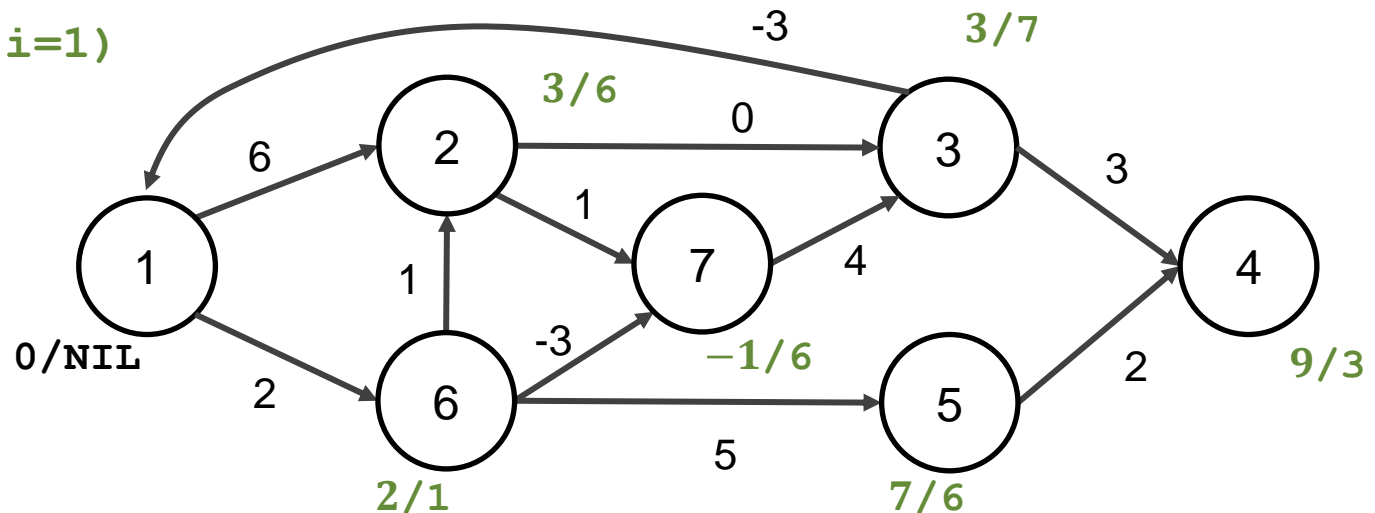
# Bellman-Ford: Beispiel (III)

Bellman-Ford-SSSP ( $G, s, w$ )

```
1  initSSSP( $G, s, w$ );
2  FOR  $i=1$  TO  $|V|-1$  DO
3      FOREACH ( $u, v$ ) in  $E$  DO
4          relax( $G, u, v, w$ );
5  FOREACH ( $u, v$ ) in  $E$  DO
6      IF  $v.\text{dist} > u.\text{dist} + w((u, v))$  THEN
7          return false;
8  return true;
```

FOR-Schleife 2 ( $i=1$ )

relax(6,2)  
relax(6,5)  
relax(6,7)  
relax(7,3)



# Bellman-Ford: Beispiel (IV)

Bellman-Ford-SSSP ( $G, s, w$ )

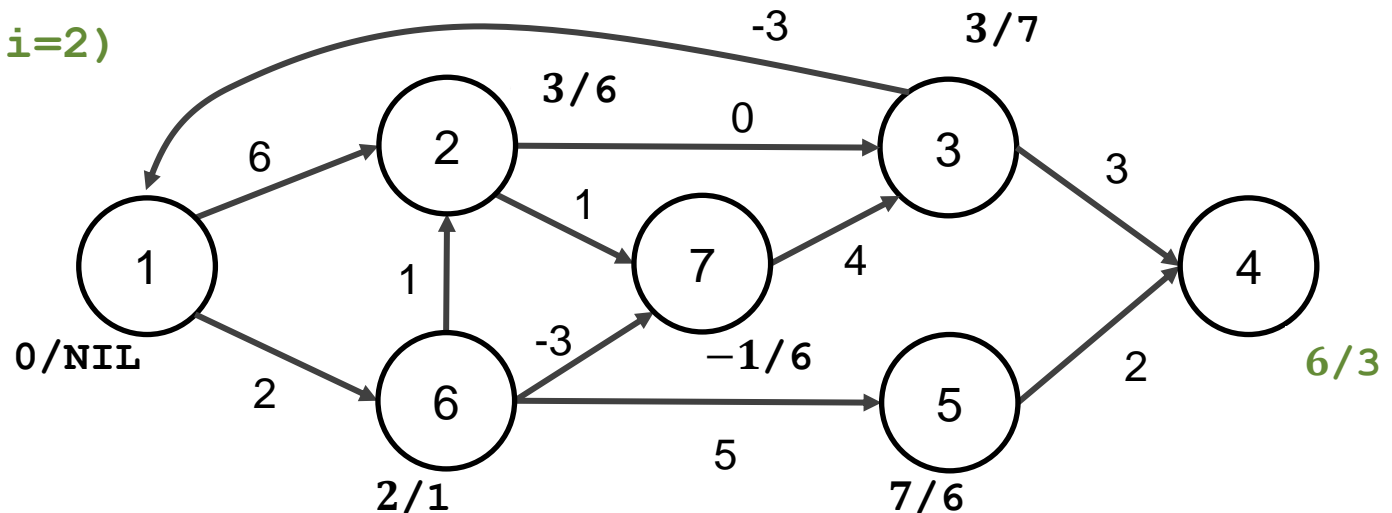
```
1  initSSSP( $G, s, w$ );
2  FOR  $i=1$  TO  $|V|-1$  DO
3      FOREACH ( $u, v$ ) in  $E$  DO
4          relax( $G, u, v, w$ );
5  FOREACH ( $u, v$ ) in  $E$  DO
6      IF  $v.\text{dist} > u.\text{dist} + w((u, v))$  THEN
7          return false;
8  return true;
```

Keine Änderungen  
mehr in den  
folgenden Iterationen

Algorithmus gibt  
**true** zurück

FOR-Schleife 2 ( $i=2$ )

```
relax(1,2)
relax(1,6)
relax(2,3)
relax(2,7)
relax(3,1)
relax(3,4)
relax(5,4)
```

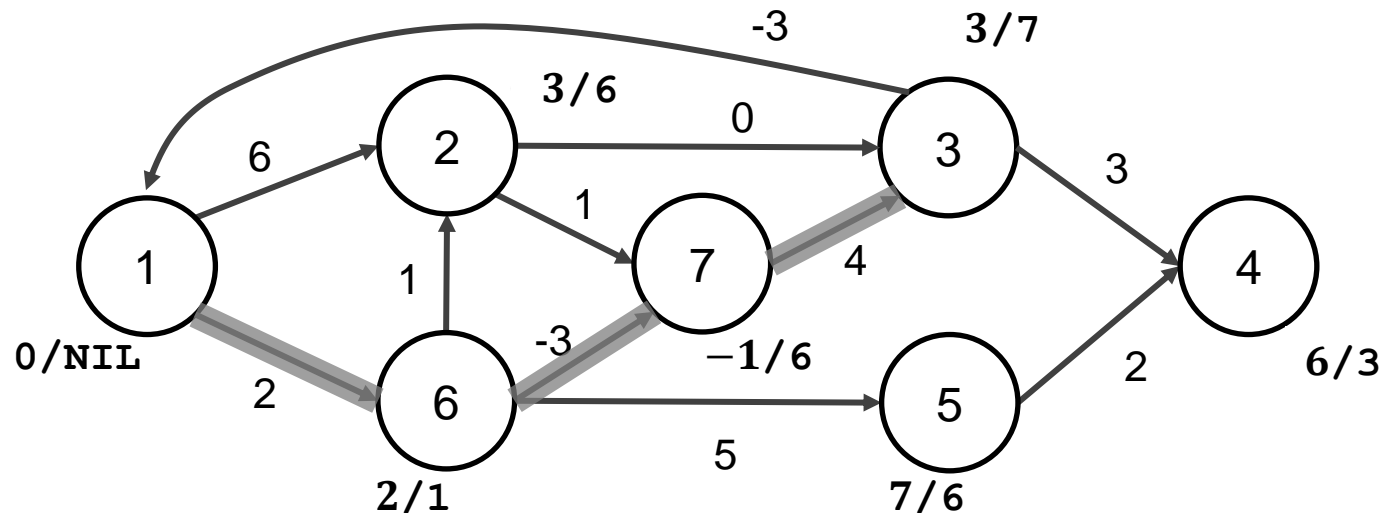


# Bellman-Ford: Beispiel (V)

Bellman-Ford-SSSP ( $G, s, w$ )

```
1  initSSSP( $G, s, w$ );
2  FOR  $i=1$  TO  $|V|-1$  DO
3      FOREACH ( $u, v$ ) in  $E$  DO
4          relax( $G, u, v, w$ );
5  FOREACH ( $u, v$ ) in  $E$  DO
6      IF  $v.\text{dist} > u.\text{dist} + w((u, v))$  THEN
7          return false;
8  return true;
```

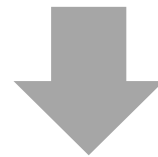
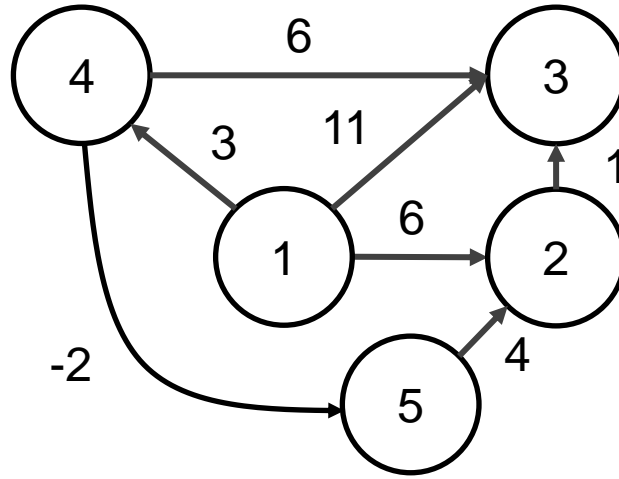
Kürzester Weg  
z.B. von 1 zu 3  
durch  
Vorgängerwerte  
gegeben



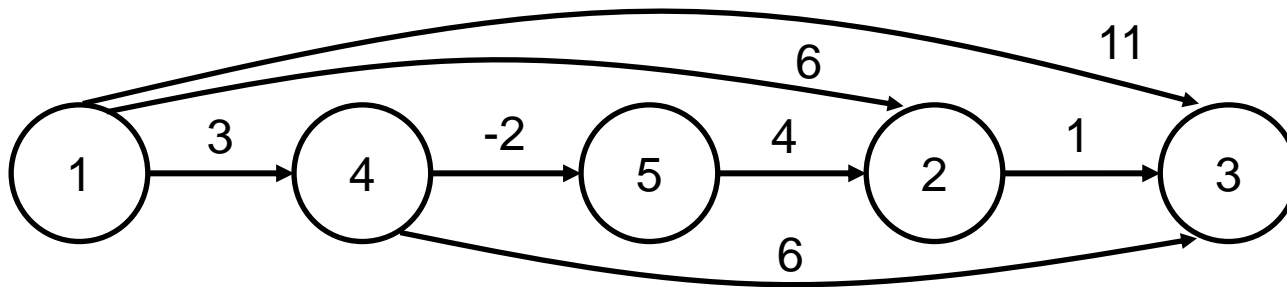


# SSSP mittels Topologischer Sortierung

dag  
mit Gewichten



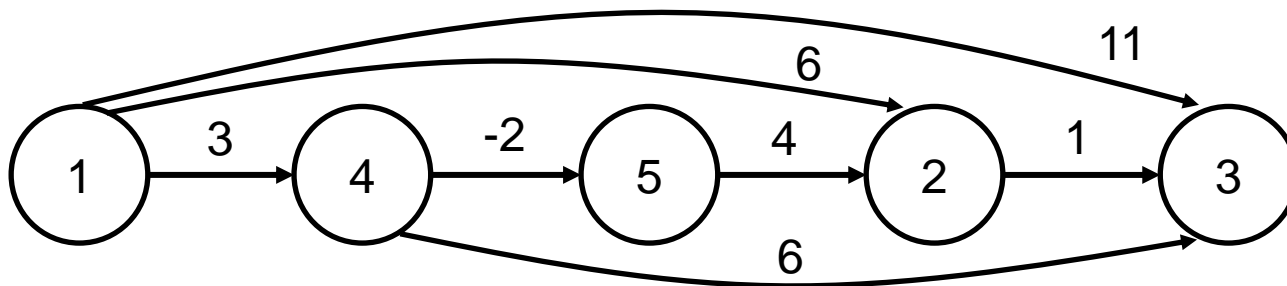
Topologisches Sortieren



# SSSP-Algorithmus für Dags

TopoSort-SSSP( $G, s, w$ ) //  $G$  dag

```
1  initSSSP( $G, s, w$ );  
2  execute topological sorting  
3  FOREACH  $u$  in  $V$  in topological order DO  
4      FOREACH  $v$  in  $\text{adj}(u)$  DO  
5          relax( $G, u, v, w$ );
```

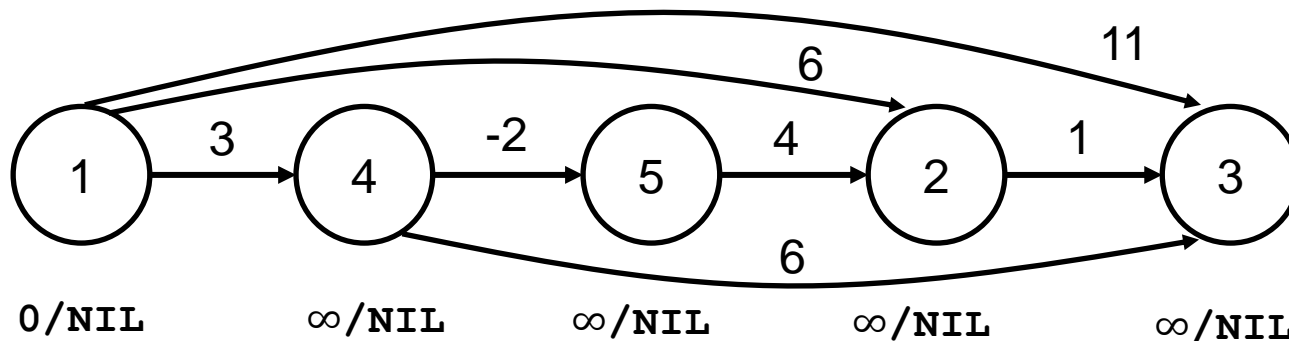


# SSSP-Algorithmus für Dags: Beispiel (I)

TopoSort-SSSP( $G, s, w$ ) //  $G$  dag

```
1  initSSSP( $G, s, w$ );  
2  execute topological sorting  
3  FOREACH  $u$  in  $V$  in topological order DO  
4      FOREACH  $v$  in  $\text{adj}(u)$  DO  
5          relax( $G, u, v, w$ );
```

Initialisierung für  $s=1$  in 1  
und Sortieren in 2

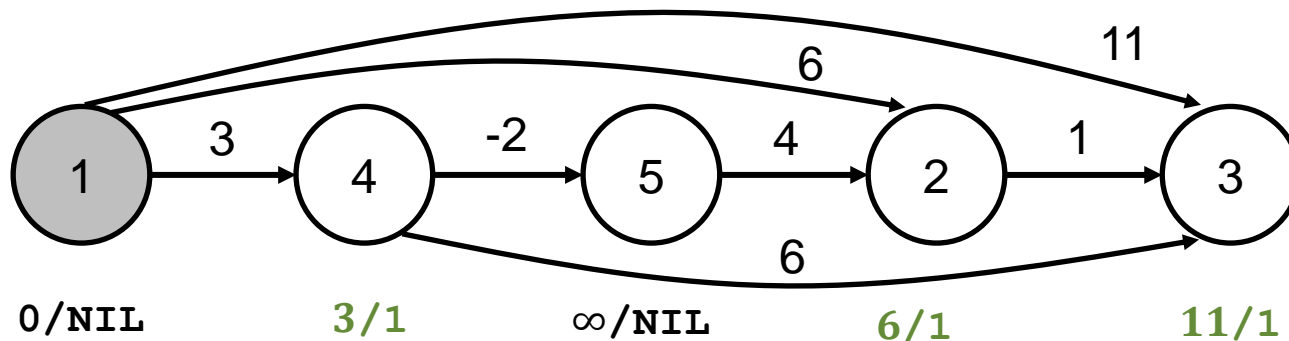


# SSSP-Algorithmus für Dags: Beispiel (II)

TopoSort-SSSP( $G, s, w$ ) //  $G$  dag

```
1  initSSSP( $G, s, w$ );  
2  execute topological sorting  
3  FOREACH  $u$  in  $V$  in topological order DO  
4      FOREACH  $v$  in  $\text{adj}(u)$  DO  
5          relax( $G, u, v, w$ );
```

3 FOREACH ( $u=1$ )

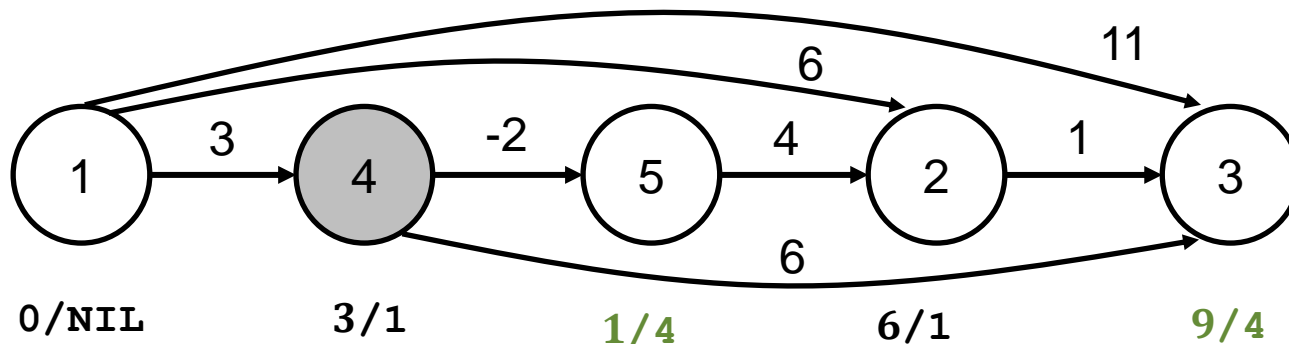


# SSSP-Algorithmus für Dags: Beispiel (III)

TopoSort-SSSP( $G, s, w$ ) //  $G$  dag

```
1  initSSSP( $G, s, w$ );  
2  execute topological sorting  
3  FOREACH  $u$  in  $V$  in topological order DO  
4      FOREACH  $v$  in  $\text{adj}(u)$  DO  
5          relax( $G, u, v, w$ );
```

3 FOREACH ( $u=4$ )

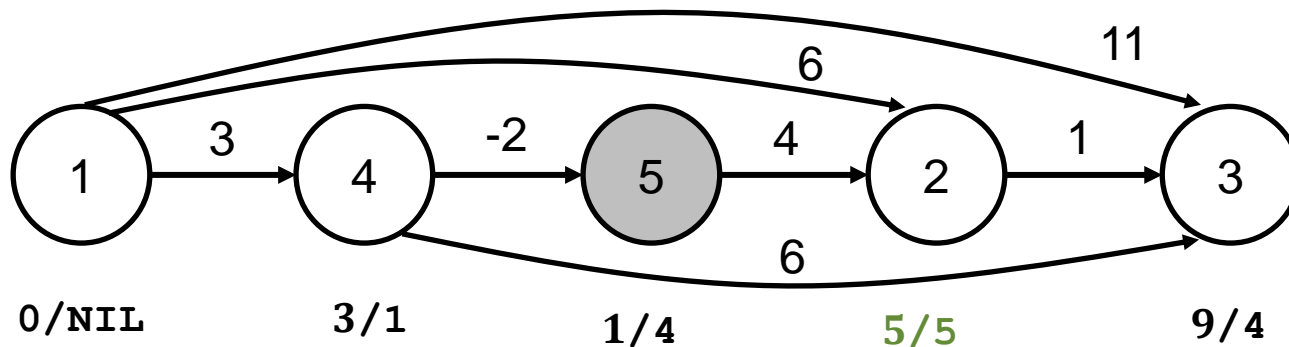


# SSSP-Algorithmus für Dags: Beispiel (IV)

TopoSort-SSSP( $G, s, w$ ) //  $G$  dag

```
1  initSSSP( $G, s, w$ );  
2  execute topological sorting  
3  FOREACH  $u$  in  $V$  in topological order DO  
4      FOREACH  $v$  in  $\text{adj}(u)$  DO  
5          relax( $G, u, v, w$ );
```

3 FOREACH ( $u=5$ )

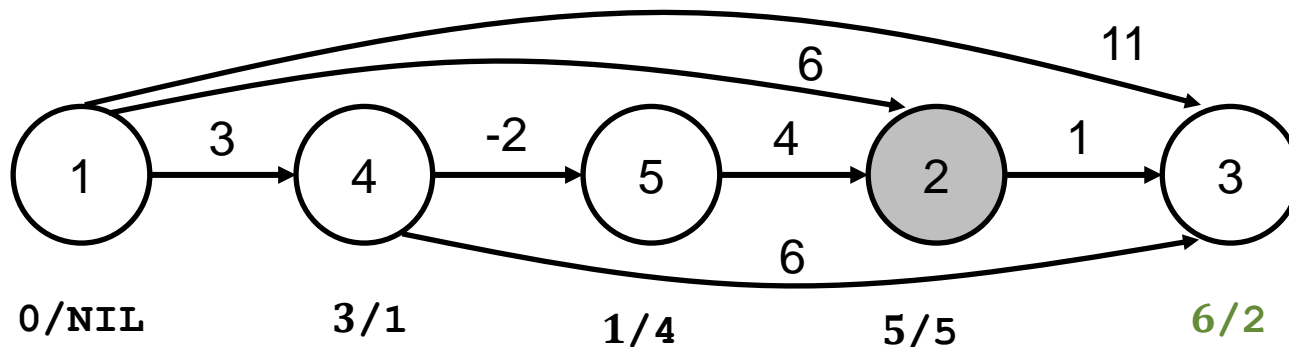


# SSSP-Algorithmus für Dags: Beispiel (V)

TopoSort-SSSP( $G, s, w$ ) //  $G$  dag

```
1  initSSSP( $G, s, w$ );  
2  execute topological sorting  
3  FOREACH  $u$  in  $V$  in topological order DO  
4      FOREACH  $v$  in  $\text{adj}(u)$  DO  
5          relax( $G, u, v, w$ );
```

3 FOREACH ( $u=2$ )



# SSSP-Algorithmus für Dags: Korrektheit+Laufzeit

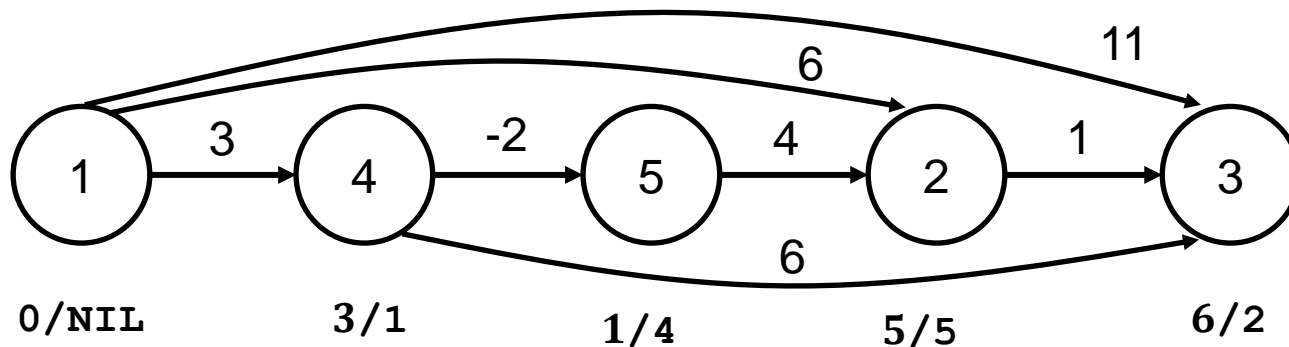
TopoSort-SSSP( $G, s, w$ ) //  $G$  dag

```
1  initSSSP( $G, s, w$ );  
2  execute topological sorting  
3  FOREACH  $u$  in  $V$  in topological order DO  
4      FOREACH  $v$  in  $\text{adj}(u)$  DO  
5          relax( $G, u, v, w$ );
```

Laufzeit =  $\theta(|E| + |V|)$

Korrektheit:  
Kanten auf einem  
kürzesten Pfad  
werden nacheinander  
„gelockert“

(vgl. Bellman-Ford)



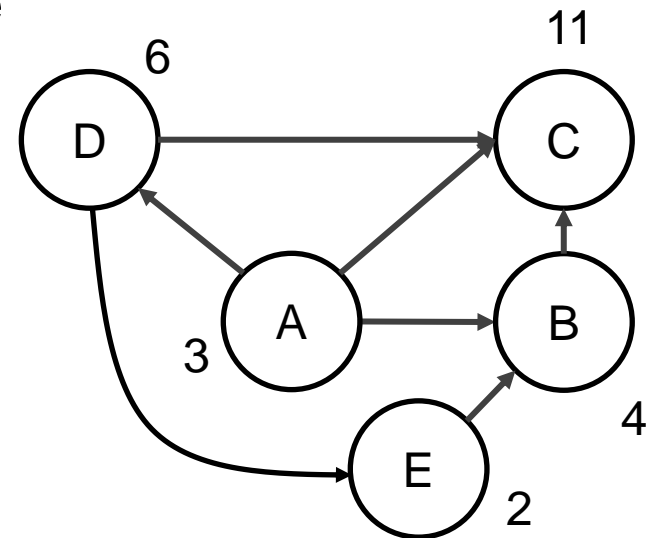




Überlegen Sie sich, wie Sie mit Hilfe des Bellman-Ford-Algorithmus alle Knoten identifizieren können, die auf einem „negativen Zyklus“ liegen.



Angenommen, Sie beschreiben ein Job-Scheduling-Problem per Dag, wobei jeder Job eine gewisse Zeit benötigt, bevor er beendet ist. Wie können Sie mit Hilfe des Dag-SSSP-Algorithmus bestimmen, wie lange ihr Projekt insgesamt braucht?



# Dijkstra-Algorithmus

Dijkstra-SSSP( $G, s, w$ )

```
1  initSSSP( $G, s, w$ );
2   $Q = V$ ; //let  $S = V - Q$ 
3  WHILE !isEmpty( $Q$ ) DO
4       $u = \text{EXTRACT-MIN}(Q)$ ; //wrt. dist
5      FOREACH  $v$  in adj( $u$ ) DO
6          relax( $G, u, v, w$ );
```

**Voraussetzung:**  
 $w((u, v)) \geq 0$   
für alle Kanten

(mittels Fibonacci-Heaps)

Laufzeit =  $\theta(|V| \cdot \log |V| + |E|)$

Ähnlichkeit zu  
Prims Algorithmus

MST-Prim( $G, w, r$ )

```
1  FOREACH  $v$  in  $V$  DO { $v.key = \infty$ ;  $v.pred = \text{NIL}$ ;}
2   $r.key = -\infty$ ;  $Q = V$ ;
3  WHILE !isEmpty( $Q$ ) DO
4       $u = \text{EXTRACT-MIN}(Q)$ ;
5      FOREACH  $v$  in adj( $u$ ) DO
6          IF  $v \in Q$  and  $w(\{u, v\}) < v.key$  THEN
7               $v.key = w(\{u, v\})$ ;
8               $v.pred = u$ ;
```

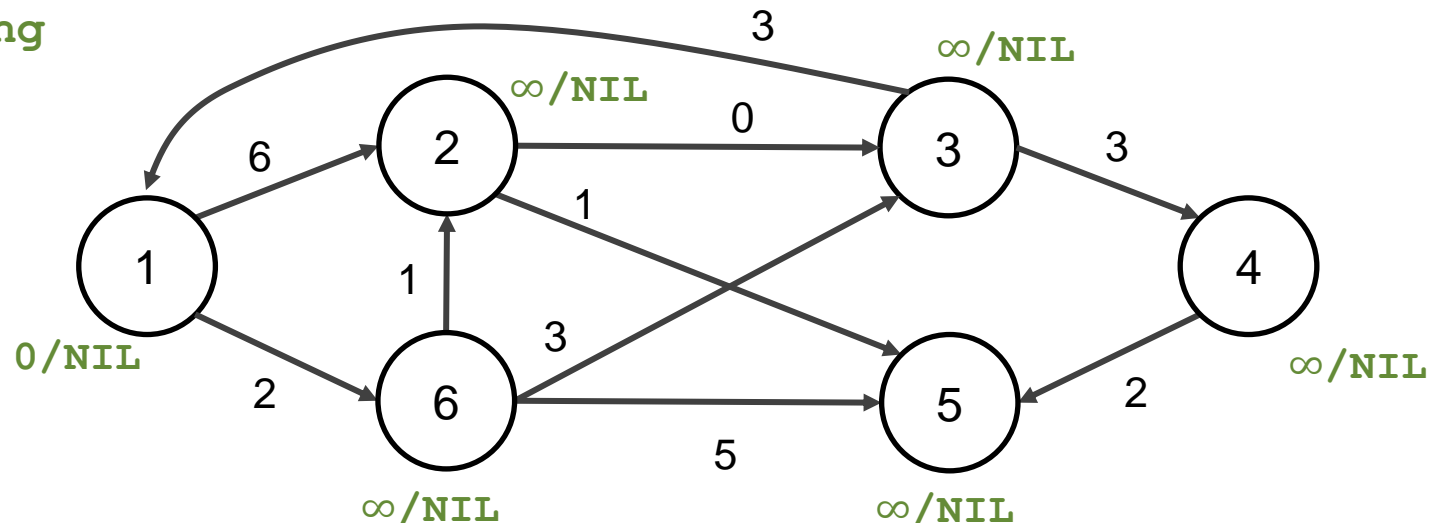
# Dijkstra-Algorithmus: Beispiel (I)

Dijkstra-SSSP ( $G, s, w$ )

```
1  initSSSP( $G, s, w$ );  
2   $Q = V$ ; //let  $S = V - Q$   
3  WHILE !isEmpty( $Q$ ) DO  
4       $u = \text{EXTRACT-MIN}(Q)$ ; //wrt. dist  
5      FOREACH  $v$  in adj( $u$ ) DO  
6          relax( $G, u, v, w$ );
```

Voraussetzung:  
 $w((u, v)) \geq 0$   
für alle Kanten

Initialisierung  
in Schritt 1



# Dijkstra-Algorithmus: Beispiel (II)

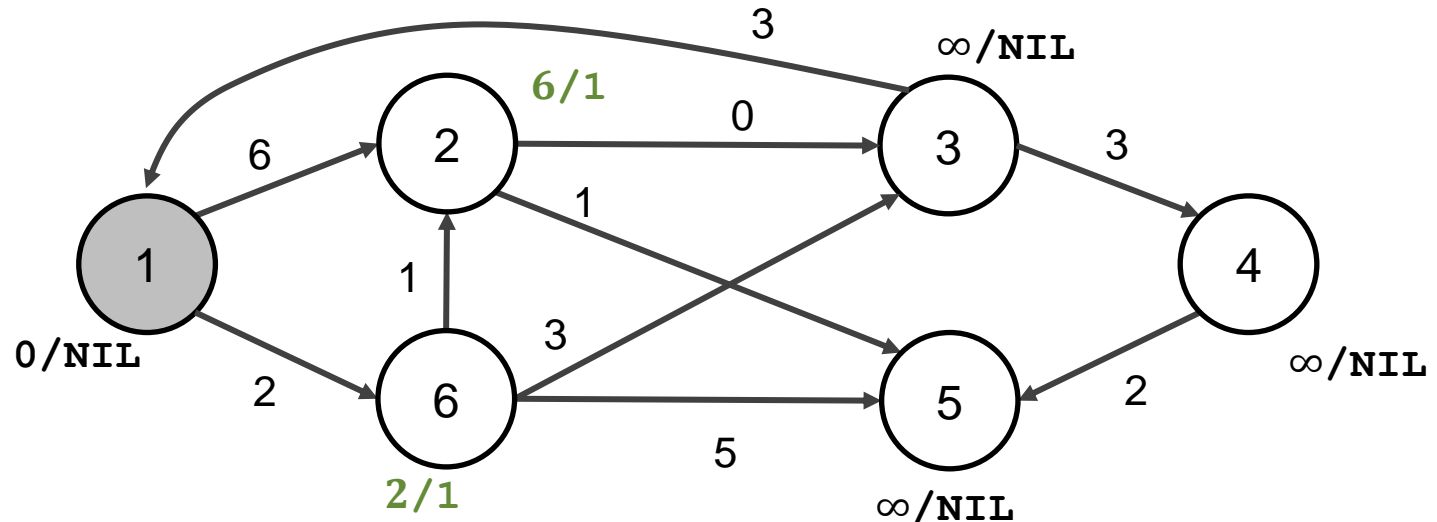
Dijkstra-SSSP ( $G, s, w$ )

```
1  initSSSP( $G, s, w$ );  
2   $Q = V$ ; //let  $S = V - Q$   
3  WHILE !isEmpty( $Q$ ) DO  
4       $u = \text{EXTRACT-MIN}(Q)$ ; //wrt. dist  
5      FOREACH  $v$  in adj( $u$ ) DO  
6          relax( $G, u, v, w$ );
```

**Voraussetzung:**  
 $w((u, v)) \geq 0$   
für alle Kanten

3 WHILE ( $u=1$ )

relax(1,2)  
relax(1,6)



# Dijkstra-Algorithmus: Beispiel (III)

Dijkstra-SSSP ( $G, s, w$ )

```
1  initSSSP( $G, s, w$ );  
2   $Q = V$ ; //let  $S = V - Q$   
3  WHILE !isEmpty( $Q$ ) DO  
4       $u = \text{EXTRACT-MIN}(Q)$ ; //wrt. dist  
5      FOREACH  $v$  in adj( $u$ ) DO  
6          relax( $G, u, v, w$ );
```

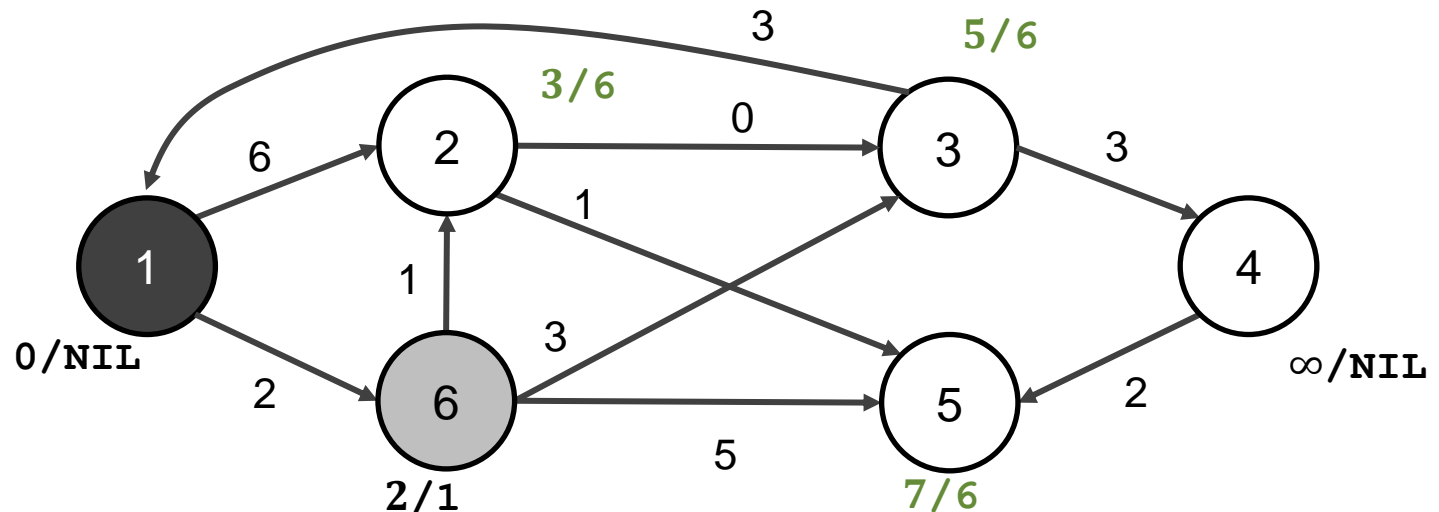
**Voraussetzung:**  
 $w((u, v)) \geq 0$   
für alle Kanten

3 WHILE ( $u=6$ )

relax( $6, 2$ )

relax( $6, 3$ )

relax( $6, 5$ )



# Dijkstra-Algorithmus: Beispiel (IV)

Dijkstra-SSSP ( $G, s, w$ )

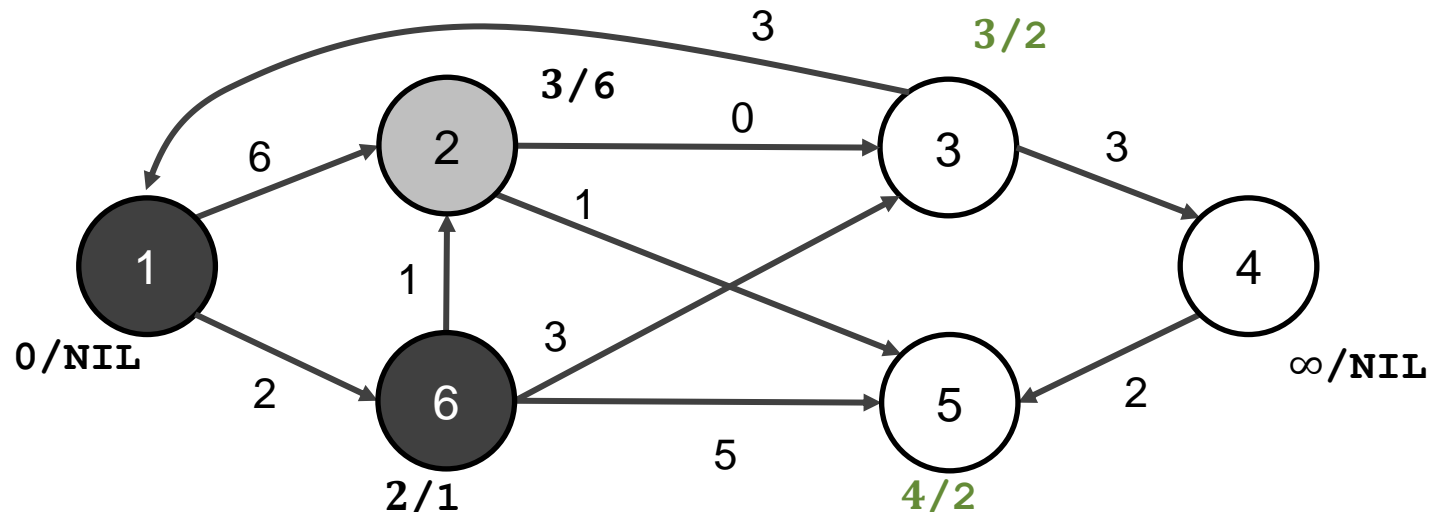
```
1  initSSSP( $G, s, w$ );  
2   $Q = V$ ; //let  $S = V - Q$   
3  WHILE !isEmpty( $Q$ ) DO  
4       $u = \text{EXTRACT-MIN}(Q)$ ; //wrt. dist  
5      FOREACH  $v$  in adj( $u$ ) DO  
6          relax( $G, u, v, w$ );
```

Voraussetzung:  
 $w((u, v)) \geq 0$   
für alle Kanten

3 WHILE ( $u=2$ )

relax( $2, 3$ )

relax( $2, 5$ )



# Dijkstra-Algorithmus: Beispiel (V)

Dijkstra-SSSP ( $G, s, w$ )

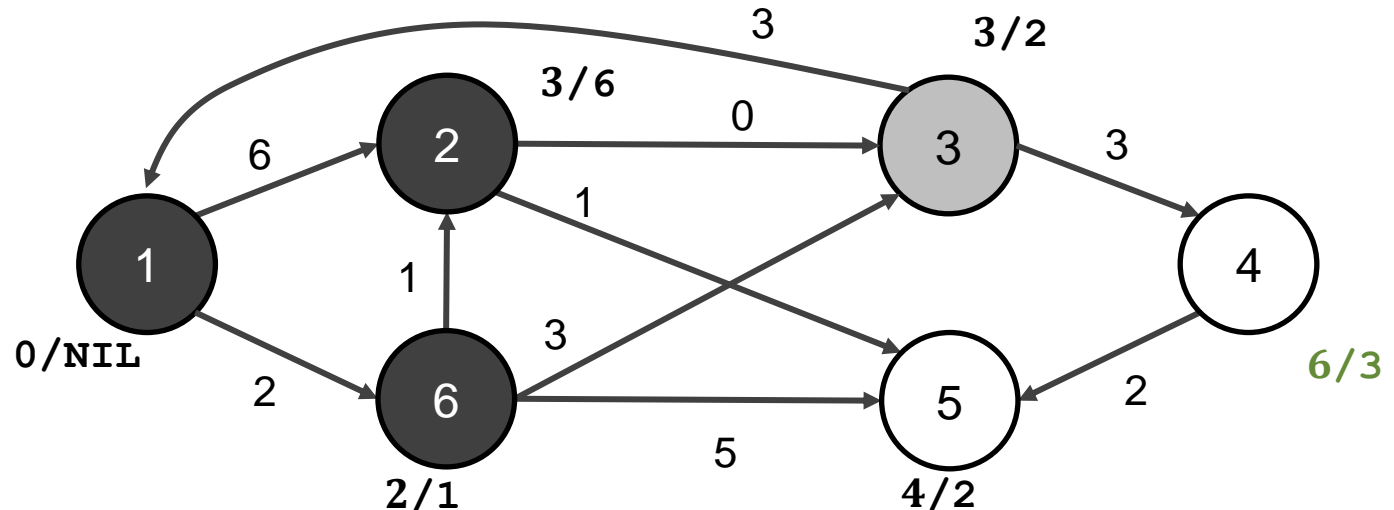
```
1  initSSSP( $G, s, w$ );  
2   $Q = V$ ; //let  $S = V - Q$   
3  WHILE !isEmpty( $Q$ ) DO  
4       $u = \text{EXTRACT-MIN}(Q)$ ; //wrt. dist  
5      FOREACH  $v$  in adj( $u$ ) DO  
6          relax( $G, u, v, w$ );
```

**Voraussetzung:**  
 $w((u, v)) \geq 0$   
für alle Kanten

3 WHILE ( $u=3$ )

relax( $3, 1$ )

relax( $3, 4$ )



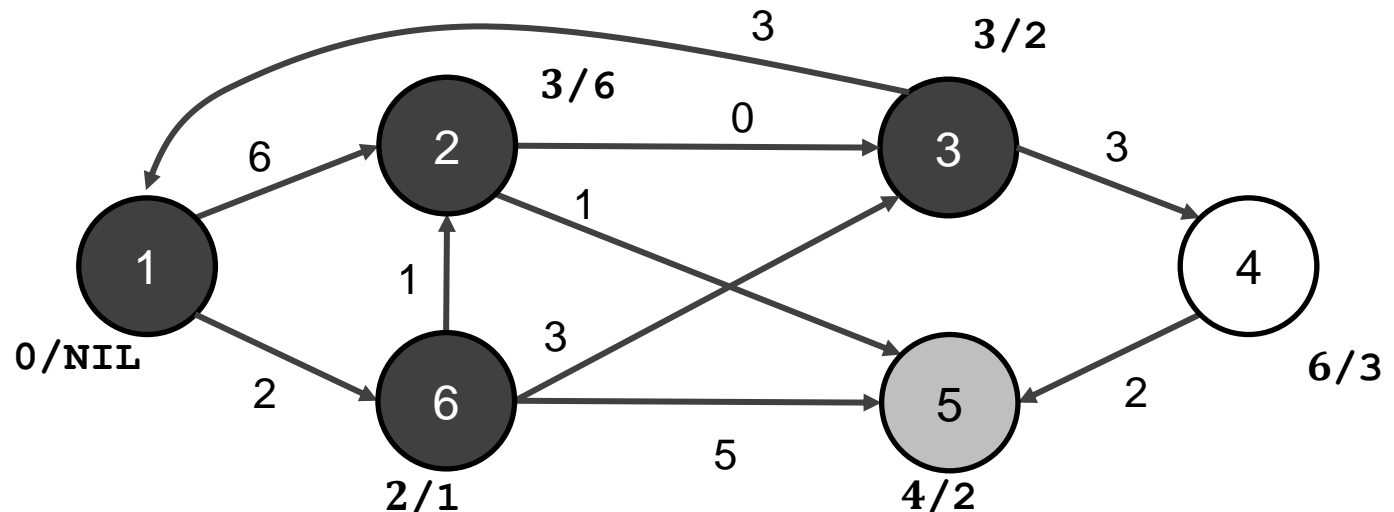
# Dijkstra-Algorithmus: Beispiel (VI)

Dijkstra-SSSP ( $G, s, w$ )

```
1  initSSSP( $G, s, w$ );  
2   $Q = V$ ; //let  $S = V - Q$   
3  WHILE !isEmpty( $Q$ ) DO  
4       $u = \text{EXTRACT-MIN}(Q)$ ; //wrt. dist  
5      FOREACH  $v$  in adj( $u$ ) DO  
6          relax( $G, u, v, w$ );
```

Voraussetzung:  
 $w((u, v)) \geq 0$   
für alle Kanten

3 WHILE ( $u=5$ )





# Dijkstra-Algorithmus: Beispiel (VII)

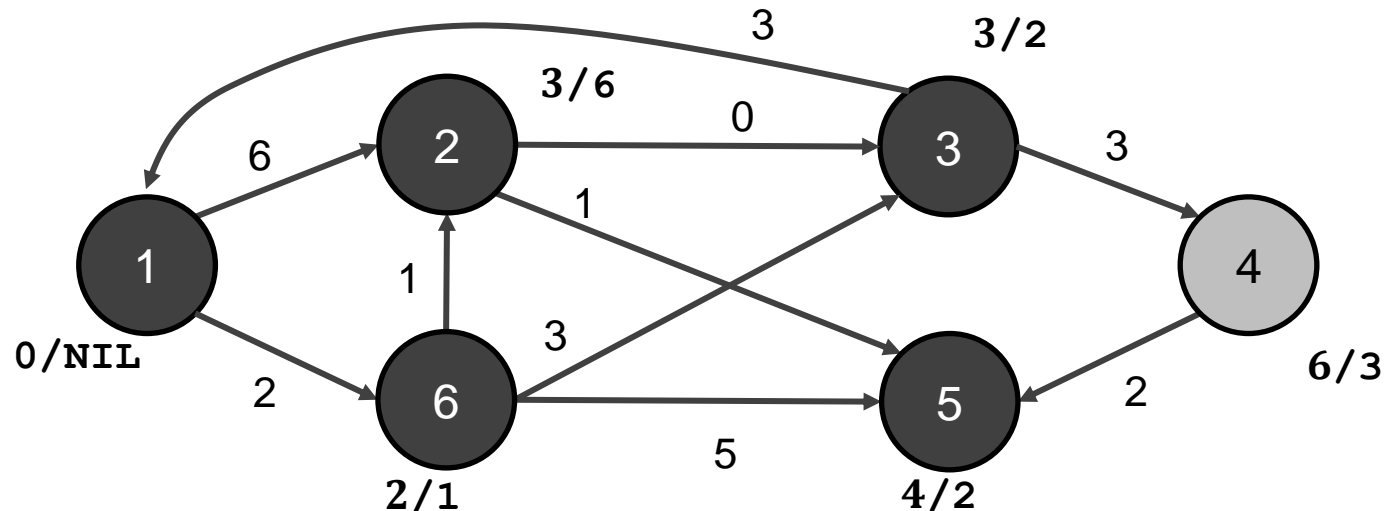
Dijkstra-SSSP ( $G, s, w$ )

```
1  initSSSP( $G, s, w$ );  
2   $Q = V$ ; //let  $S = V - Q$   
3  WHILE !isEmpty( $Q$ ) DO  
4       $u = \text{EXTRACT-MIN}(Q)$ ; //wrt. dist  
5      FOREACH  $v$  in adj( $u$ ) DO  
6          relax( $G, u, v, w$ );
```

**Voraussetzung:**  
 $w((u, v)) \geq 0$   
für alle Kanten

3 WHILE ( $u=4$ )

relax( $4, 5$ )

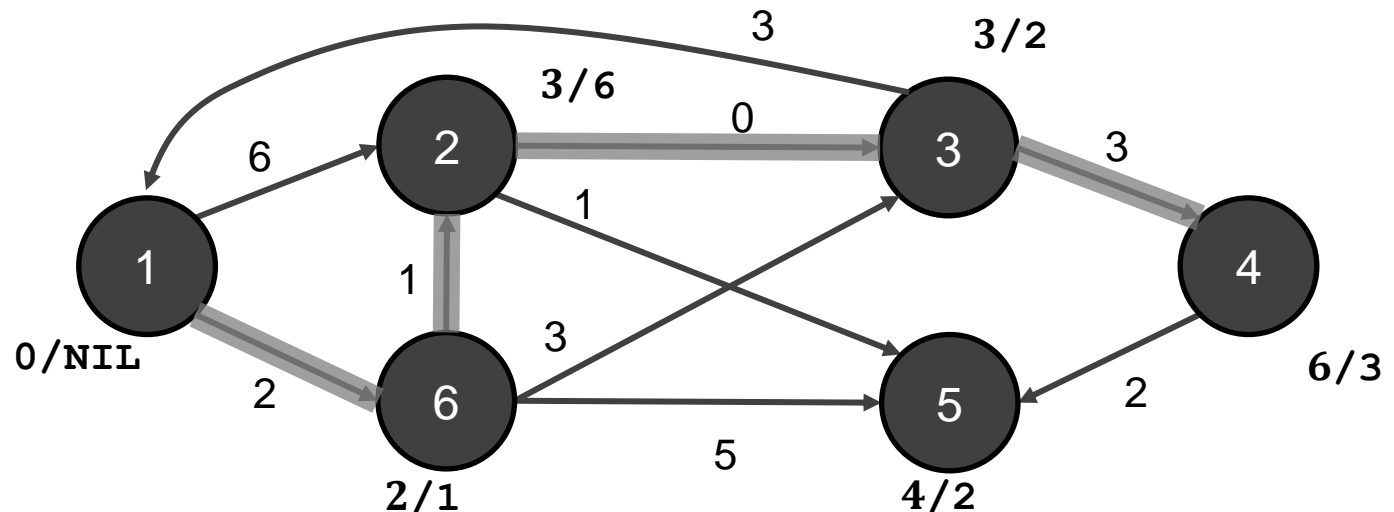


# Dijkstra-Algorithmus: Beispiel (VIII)

Dijkstra-SSSP ( $G, s, w$ )

```
1  initSSSP( $G, s, w$ );  
2   $Q = V$ ; //let  $S = V - Q$   
3  WHILE !isEmpty( $Q$ ) DO  
4       $u = \text{EXTRACT-MIN}(Q)$ ; //wrt. dist  
5      FOREACH  $v$  in adj( $u$ ) DO  
6          relax( $G, u, v, w$ );
```

**Voraussetzung:**  
 $w((u, v)) \geq 0$   
für alle Kanten

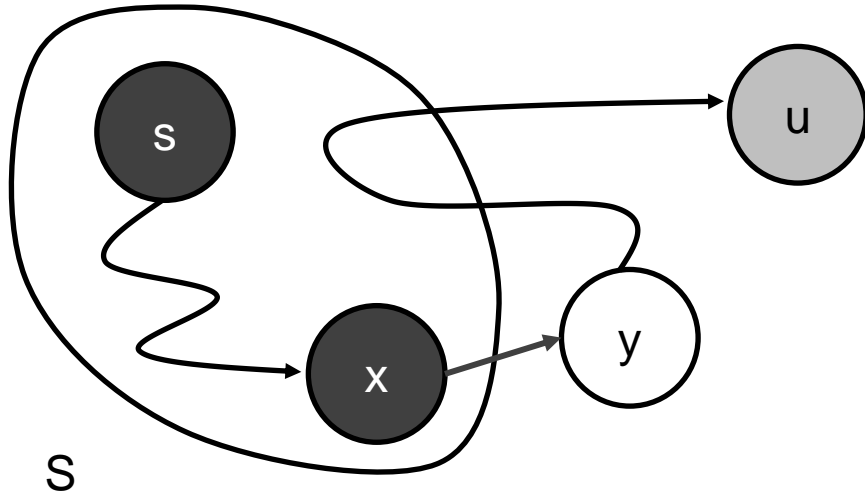


Beispiel:  
kürzester  
Weg  $1 \rightarrow 4$

# Dijkstra-Algorithmus: Korrektheit (I)

Dijkstra-SSSP ( $G, s, w$ )

```
1  initSSSP( $G, s, w$ ) ;  
2   $Q = V$ ; //let  $S = V - Q$   
3  WHILE !isEmpty( $Q$ ) DO  
4       $u = \text{EXTRACT-MIN}(Q)$ ; //wrt. dist  
5      FOREACH  $v$  in adj( $u$ ) DO  
6          relax( $G, u, v, w$ );
```



Für jeden betrachteten Knoten  $u$  in der WHILE-Schleife gilt:  
 $u.\text{dist} = \text{shortest}(s, u)$

Angenommen,  $u$  wäre erster Knoten, bei dem nicht der Fall

Insbesondere  $u \neq s$  und  $S \neq \emptyset$   
da  $s.\text{dist} = 0$  korrekt

Betrachte kürzesten Pfad  $s \rightarrow u$   
mit „erstem“ Knoten  $y$  nicht in  $S$ ,  
und Vorgängerknoten  $x$  in  $S$ .

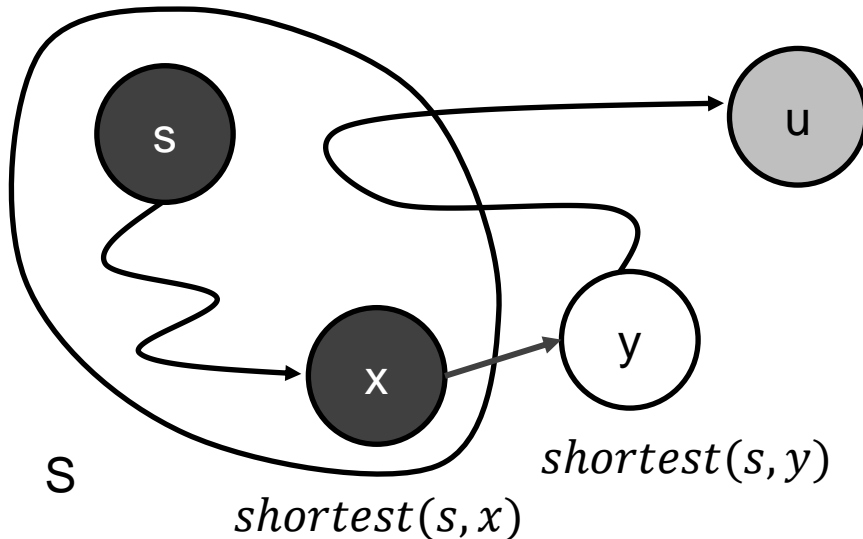
# Dijkstra-Algorithmus: Korrektheit (II)

Dijkstra-SSSP ( $G, s, w$ )

```
1  initSSSP( $G, s, w$ );  
2   $Q = V$ ; //let  $S = V - Q$   
3  WHILE !isEmpty( $Q$ ) DO  
4     $u = \text{EXTRACT-MIN}(Q)$ ; //wrt. dist  
5    FOREACH  $v$  in adj( $u$ ) DO  
6      relax( $G, u, v, w$ );
```

Für jeden betrachteten Knoten  $u$  in der WHILE-Schleife gilt:  
 $u.\text{dist} = \text{shortest}(s, u)$

Angenommen,  $u$  wäre erster Knoten, bei dem nicht der Fall



Es gilt  $x.\text{dist} = \text{shortest}(s, x)$ , da  $u$  der erste Knoten ist, bei dem diese Gleichung nicht gilt

Da Kante  $(x, y)$  in dem Moment, als  $x$  zu  $S$  hinzugefügt wurde, auch „relaxed“ wurde, gilt auch  $y.\text{dist} = \text{shortest}(s, y)$

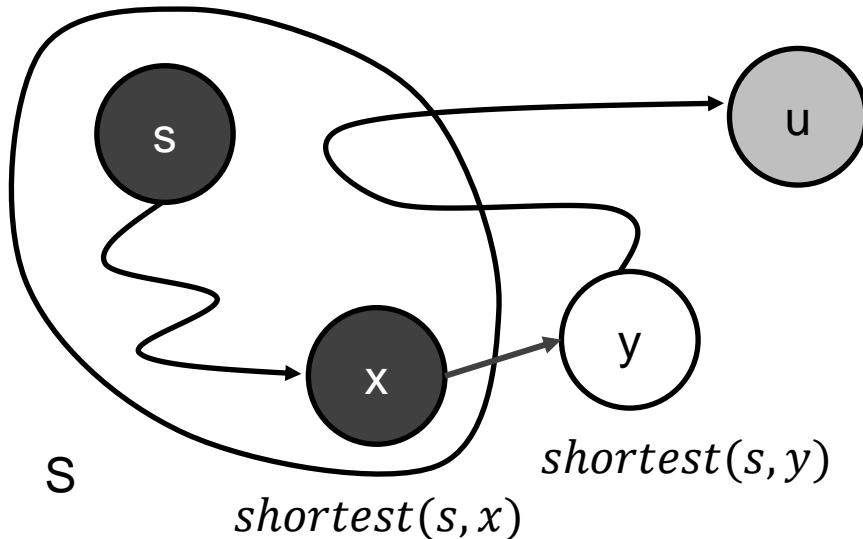
# Dijkstra-Algorithmus: Korrektheit (III)

Dijkstra-SSSP ( $G, s, w$ )

```
1  initSSSP( $G, s, w$ );  
2   $Q = V$ ; //let  $S = V - Q$   
3  WHILE !isEmpty( $Q$ ) DO  
4     $u = \text{EXTRACT-MIN}(Q)$ ; //wrt. dist  
5    FOREACH  $v$  in adj( $u$ ) DO  
6      relax( $G, u, v, w$ );
```

Für jeden betrachteten Knoten  $u$  in der WHILE-Schleife gilt:  
 $u.\text{dist} = \text{shortest}(s, u)$

Angenommen,  $u$  wäre erster Knoten, bei dem nicht der Fall



Da nur **nicht-negative Kantengewichte** gilt

$$\text{shortest}(s, y) \leq \text{shortest}(s, u),$$

und damit

$$y.\text{dist} = \text{shortest}(s, y) \leq \text{shortest}(s, u) \leq u.\text{dist}$$

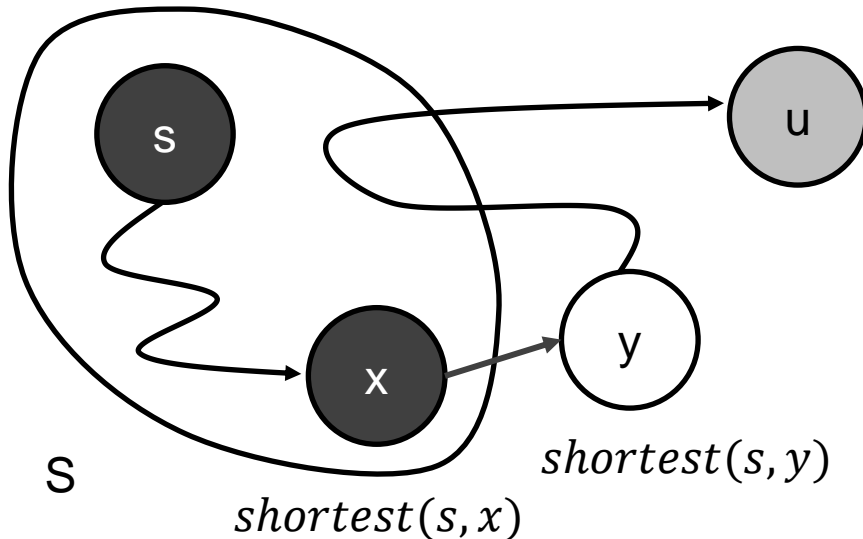
# Dijkstra-Algorithmus: Korrektheit (IV)

Dijkstra-SSSP ( $G, s, w$ )

```
1  initSSSP( $G, s, w$ ) ;  
2   $Q = V$ ; //let  $S = V - Q$   
3  WHILE !isEmpty( $Q$ ) DO  
4       $u = \text{EXTRACT-MIN}(Q)$ ; //wrt. dist  
5      FOREACH  $v$  in adj( $u$ ) DO  
6          relax( $G, u, v, w$ );
```

Für jeden betrachteten Knoten  $u$  in der WHILE-Schleife gilt:  
 $u.\text{dist} = \text{shortest}(s, u)$

Angenommen,  $u$  wäre erster Knoten, bei dem nicht der Fall



Andererseits wurde  $u$  vor  $y$  für  $S$  ausgewählt, also

$u.\text{dist} \leq y.\text{dist}$  und

$y.\text{dist} = \text{shortest}(s, y) \leq \text{shortest}(s, u) \leq u.\text{dist}$

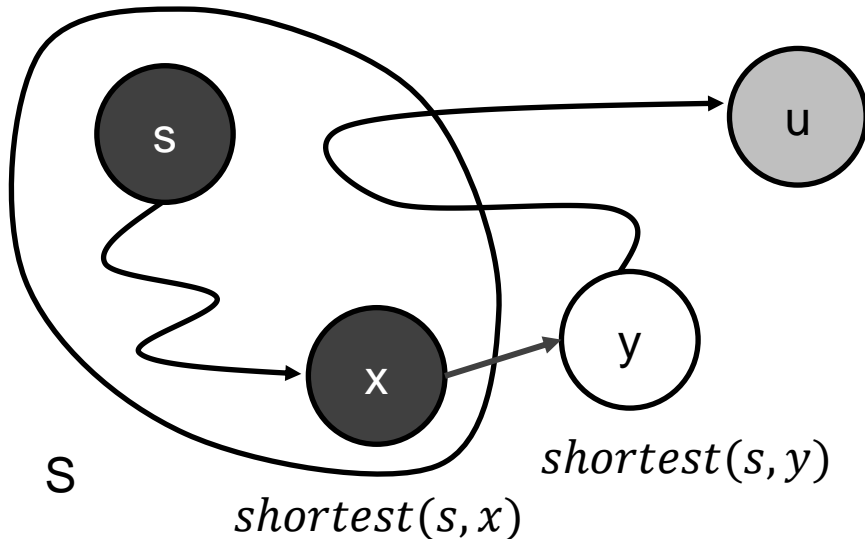
# Dijkstra-Algorithmus: Korrektheit (V)

Dijkstra-SSSP ( $G, s, w$ )

```
1  initSSSP( $G, s, w$ ) ;
2   $Q = V$ ; //let  $S = V - Q$ 
3  WHILE !isEmpty( $Q$ ) DO
4       $u = \text{EXTRACT-MIN}(Q)$ ; //wrt. dist
5      FOREACH  $v$  in adj( $u$ ) DO
6          relax( $G, u, v, w$ );
```

Für jeden betrachteten Knoten  $u$  in der WHILE-Schleife gilt:  
 $u.\text{dist} = \text{shortest}(s, u)$

Angenommen,  $u$  wäre erster Knoten, bei dem nicht der Fall



Folglich

$y.\text{dist} = \text{shortest}(s, y)$   
 $= \text{shortest}(s, u) = u.\text{dist}$ , da

$u.\text{dist} \leq y.\text{dist}$  und

$y.\text{dist} = \text{shortest}(s, y)$   
 $\leq \text{shortest}(s, u) \leq u.\text{dist}$

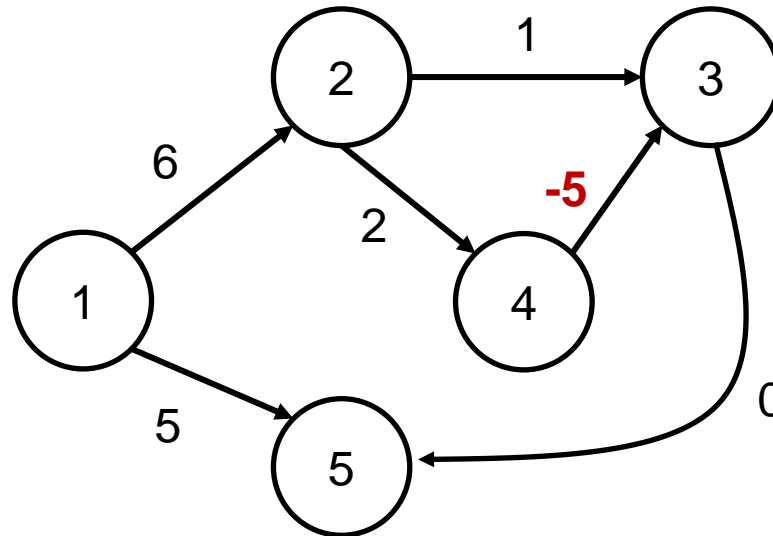
# Dijkstra-Algorithmus und negative Kantengewichte (I)

Dijkstra-SSSP( $G, s, w$ )

```
1  initSSSP( $G, s, w$ ) ;  
2   $Q = V$ ; //let  $S = V - Q$   
3  WHILE !isEmpty( $Q$ ) DO  
4       $u = \text{EXTRACT-MIN}(Q)$  ;  
5      FOREACH  $v$  in adj( $u$ ) DO  
6          relax( $G, u, v, w$ ) ;
```

kürzeste Weg  $1 \rightarrow 5$  via  
 $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 5$   
mit Gewicht  $6+2-5+0=3$

Dijkstra-Algorithmus für  $s=1$   
gibt aber  $1 \rightarrow 5$  an





# Dijkstra-Algorithmus und negative Kantengewichte (II)

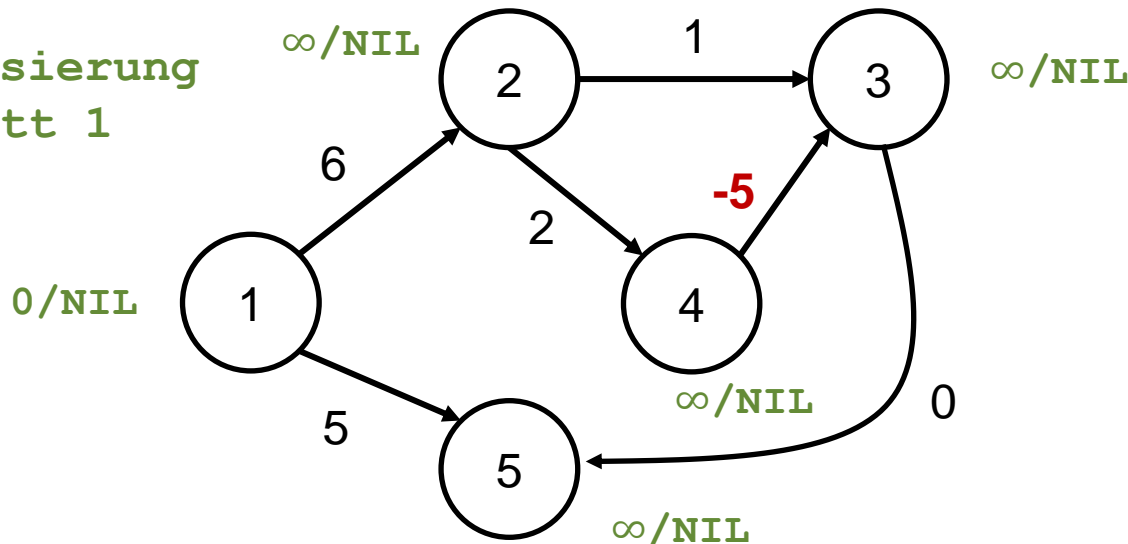
Dijkstra-SSSP ( $G, s, w$ )

```
1  initSSSP( $G, s, w$ ) ;  
2   $Q = V$ ; //let  $S = V - Q$   
3  WHILE !isEmpty( $Q$ ) DO  
4     $u = \text{EXTRACT-MIN}(Q)$  ;  
5    FOREACH  $v$  in adj( $u$ ) DO  
6      relax( $G, u, v, w$ ) ;
```

kürzeste Weg  $1 \rightarrow 5$  via  
 $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 5$   
mit Gewicht  $6 + 2 - 5 + 0 = 3$

Dijkstra-Algorithmus für  $s=1$   
gibt aber  $1 \rightarrow 5$  an

Initialisierung  
in Schritt 1



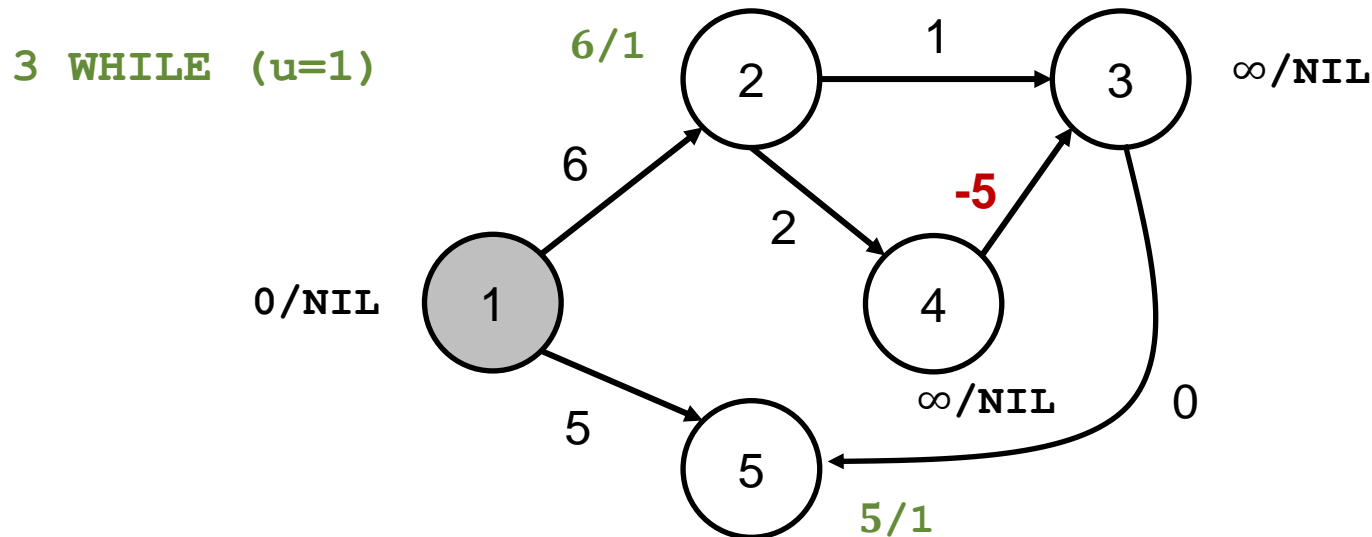
# Dijkstra-Algorithmus und negative Kantengewichte (III)

Dijkstra-SSSP ( $G, s, w$ )

```
1  initSSSP( $G, s, w$ ) ;  
2   $Q = V$ ; //let  $S = V - Q$   
3  WHILE !isEmpty( $Q$ ) DO  
4       $u = \text{EXTRACT-MIN}(Q)$  ;  
5      FOREACH  $v$  in  $\text{adj}(u)$  DO  
6          relax( $G, u, v, w$ ) ;
```

kürzeste Weg  $1 \rightarrow 5$  via  
 $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 5$   
mit Gewicht  $6+2-5+0=3$

Dijkstra-Algorithmus für  $s=1$   
gibt aber  $1 \rightarrow 5$  an



# Dijkstra-Algorithmus und negative Kantengewichte (IV)

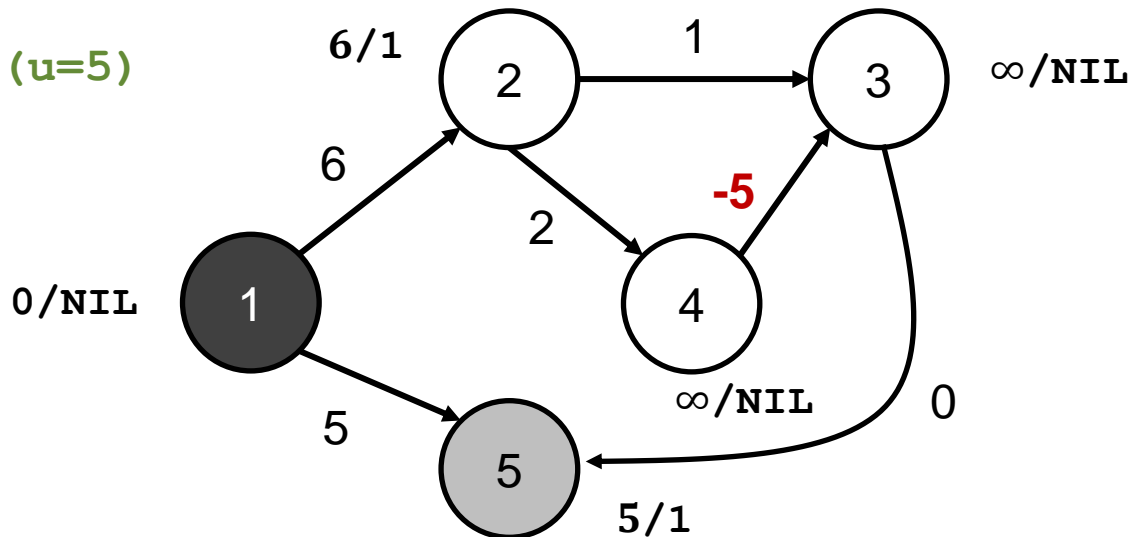
Dijkstra-SSSP ( $G, s, w$ )

```
1  initSSSP( $G, s, w$ ) ;
2   $Q = V$ ; //let  $S = V - Q$ 
3  WHILE !isEmpty( $Q$ ) DO
4       $u = \text{EXTRACT-MIN}(Q)$  ;
5      FOREACH  $v$  in adj( $u$ ) DO
6          relax( $G, u, v, w$ ) ;
```

kürzeste Weg  $1 \rightarrow 5$  via  
 $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 5$   
mit Gewicht  $6+2-5+0=3$

Dijkstra-Algorithmus für  $s=1$   
gibt aber  $1 \rightarrow 5$  an

3 WHILE ( $u=5$ )



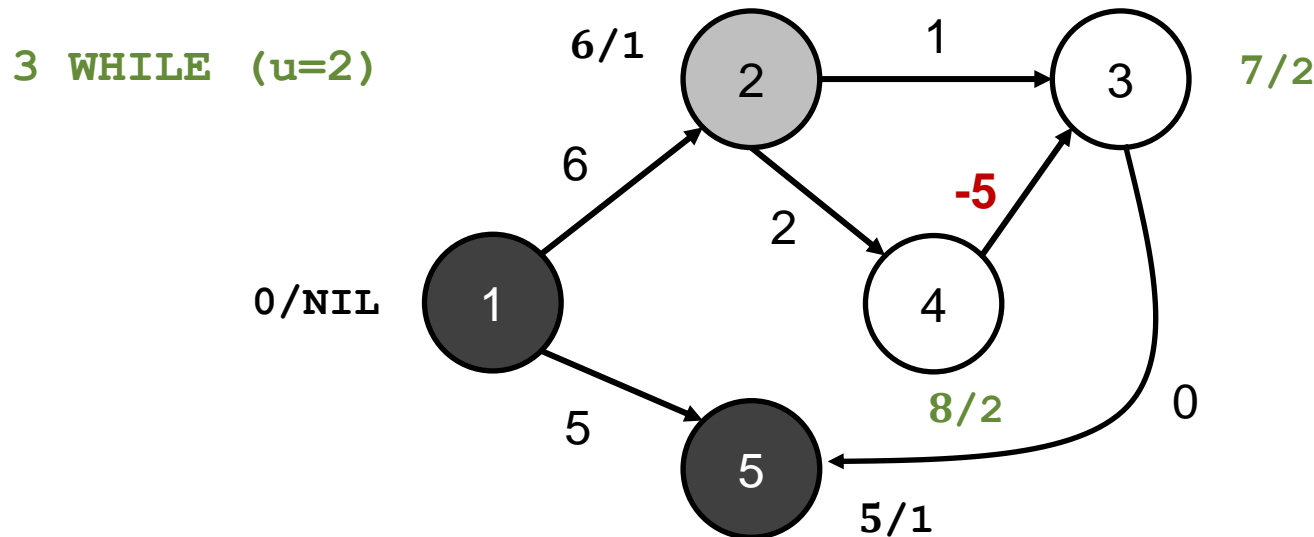
# Dijkstra-Algorithmus und negative Kantengewichte (V)

Dijkstra-SSSP ( $G, s, w$ )

```
1  initSSSP( $G, s, w$ ) ;
2   $Q = V$ ; //let  $S = V - Q$ 
3  WHILE !isEmpty( $Q$ ) DO
4       $u = \text{EXTRACT-MIN}(Q)$  ;
5      FOREACH  $v$  in adj( $u$ ) DO
6          relax( $G, u, v, w$ ) ;
```

kürzeste Weg  $1 \rightarrow 5$  via  
 $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 5$   
mit Gewicht  $6+2-5+0=3$

Dijkstra-Algorithmus für  $s=1$   
gibt aber  $1 \rightarrow 5$  an



# Dijkstra-Algorithmus und negative Kantengewichte (VI)

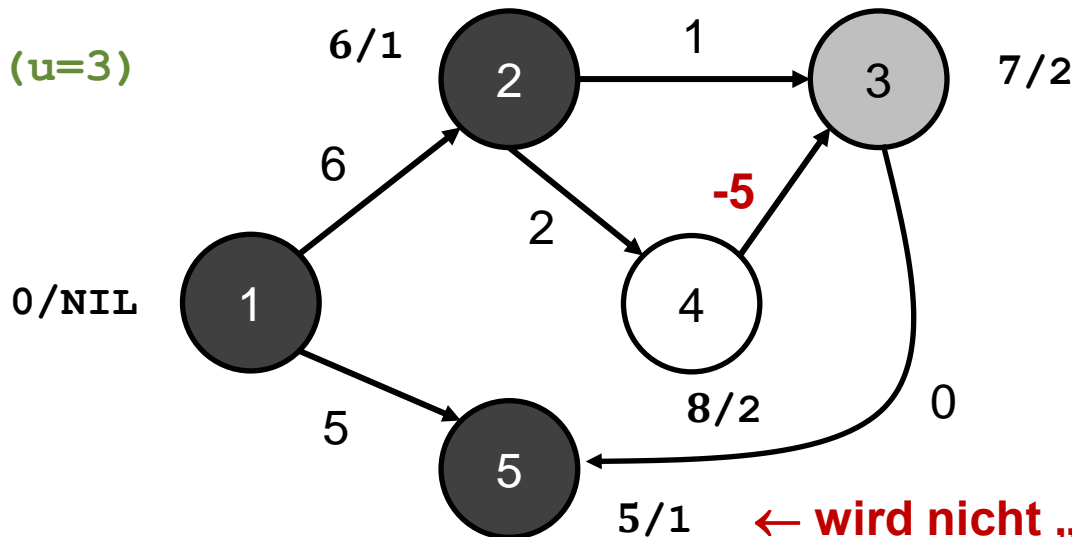
Dijkstra-SSSP ( $G, s, w$ )

```
1  initSSSP( $G, s, w$ ) ;
2   $Q = V$ ; //let  $S = V - Q$ 
3  WHILE !isEmpty( $Q$ ) DO
4       $u = \text{EXTRACT-MIN}(Q)$  ;
5      FOREACH  $v$  in  $\text{adj}(u)$  DO
6          relax( $G, u, v, w$ ) ;
```

kürzeste Weg  $1 \rightarrow 5$  via  
 $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 5$   
mit Gewicht  $6+2-5+0=3$

Dijkstra-Algorithmus für  $s=1$   
gibt aber  $1 \rightarrow 5$  an

3 WHILE ( $u=3$ )



*Der im Augenblick  
nicht erfasste Weg  
von  $1 \rightarrow 3$  über 4  
wird später zum  
kürzeren Weg*

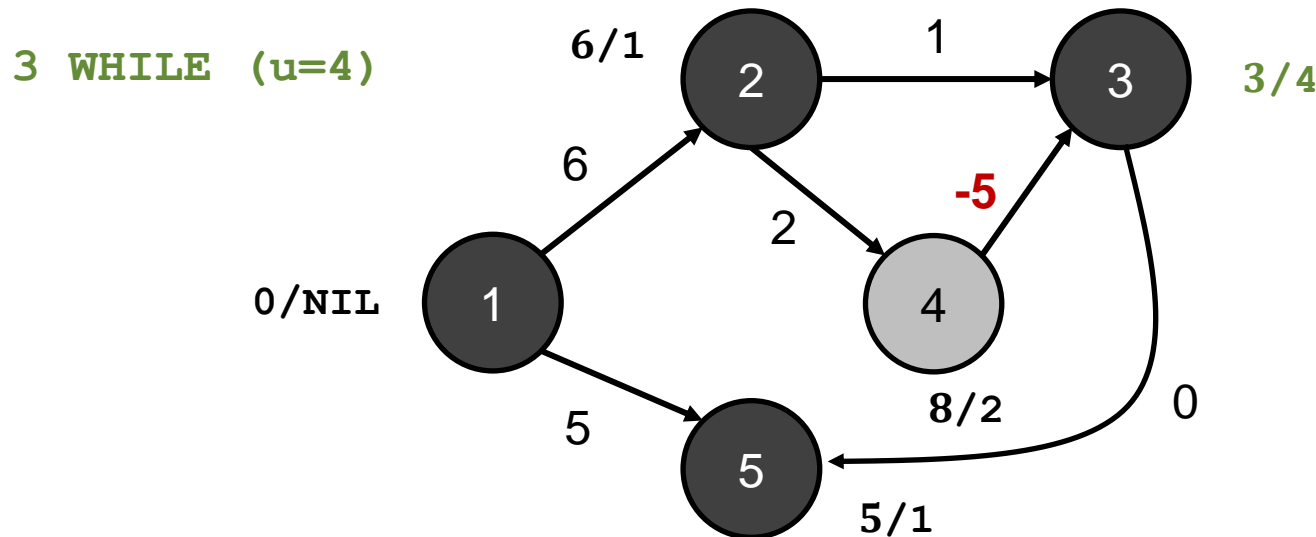
# Dijkstra-Algorithmus und negative Kantengewichte (VII)

Dijkstra-SSSP ( $G, s, w$ )

```
1  initSSSP( $G, s, w$ ) ;
2   $Q = V$ ; //let  $S = V - Q$ 
3  WHILE !isEmpty( $Q$ ) DO
4       $u = \text{EXTRACT-MIN}(Q)$  ;
5      FOREACH  $v$  in  $\text{adj}(u)$  DO
6          relax( $G, u, v, w$ ) ;
```

kürzeste Weg  $1 \rightarrow 5$  via  
 $1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 5$   
mit Gewicht  $6+2-5+0=3$

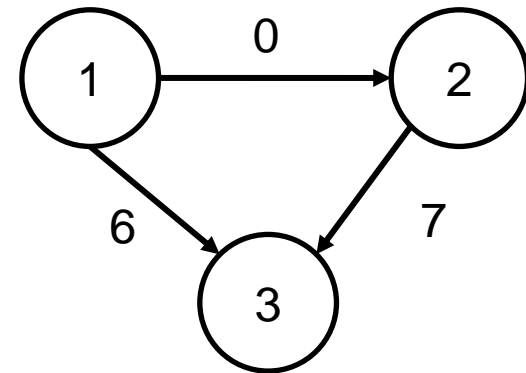
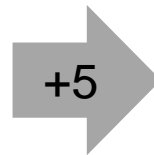
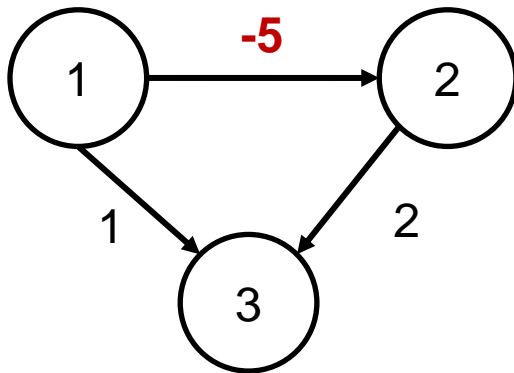
Dijkstra-Algorithmus für  $s=1$   
gibt aber  $1 \rightarrow 5$  an



Algorithmus terminiert.

# Kantengewichte nicht-negativ machen?

Versuch: Addiere absoluten Wert der kleinsten Kante zu allen Werten



kürzester Weg 1→3:  
über 2, mit Gewicht -3

kürzester Weg 1→3:  
direkt, mit Gewicht 6 (bzw.  $6-5=1$ )

Problem: man addiert den Wert so oft, wie #Kanten auf dem kürzesten Weg

# A\*-Algorithmus (I)

Hart, Nilsson, Raphael (1968)

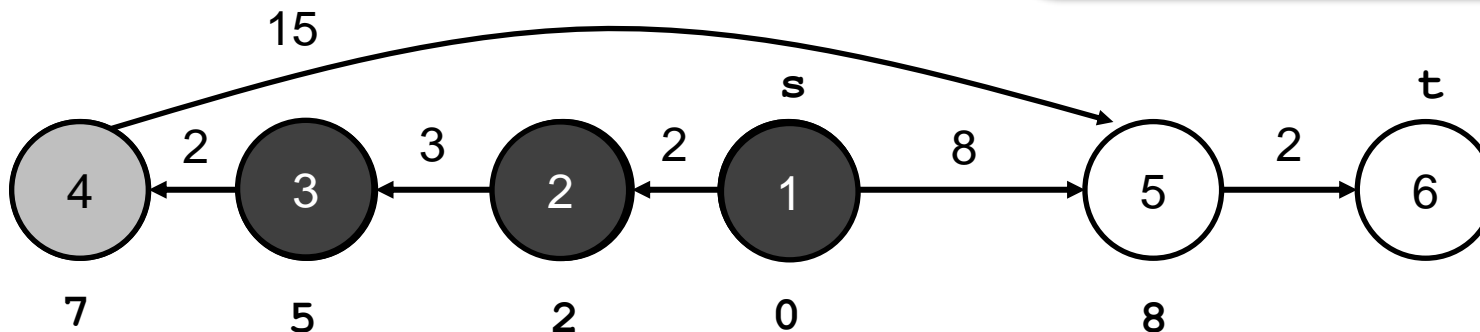


Spezialfall: suche kürzesten Weg von  $s$  zu einem Ziel  $t$

Dijkstra-SSSP ( $G, s, w$ )

```
1  initSSSP( $G, s, w$ ) ;
2   $Q = V$ ; //let  $S = V - Q$ 
3  WHILE !isEmpty( $Q$ ) DO
4       $u = \text{EXTRACT-MIN}(Q)$  ;
5      FOREACH  $v$  in adj( $u$ ) DO
6          relax( $G, u, v, w$ ) ;
```

Dijkstras Algorithmus sucht lokal vom gegenwärtigen Punkt aus günstigsten nächsten Schritt, ignoriert aber Zielrichtung



Algorithmus sucht erst in falscher Richtung



# A\*-Algorithmus (II)

(nicht-negative Kantengewichte)

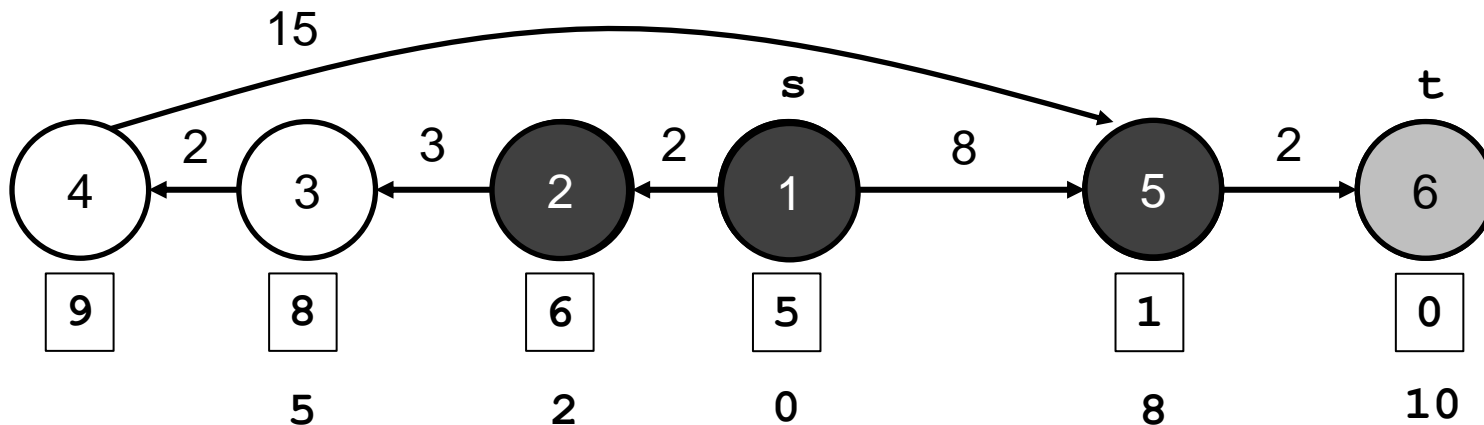


Idee: füge Heuristik hinzu, die „vom Ziel her denkt“

```
A*(G, s, t, w)
1  init(G, s, t, w);
2  Q=V; //let S=V-Q
3  WHILE !isEmpty(Q) DO
4      u=EXTRACT-MIN(Q);
5      IF u==t THEN break;
6      FOREACH v in adj(u) DO
7          relax(G, u, v, w);
```

jeder Knoten  $u$  bekommt  
zusätzlich Wert  $u.heur$  zugewiesen  
(Beispiel: Abstand Luftlinie vom Ziel)

Minimum über  
 $u.dist + u.heur$



# A\*-Algorithmus (III)

(nicht-negative Kantengewichte)



A\* findet optimale Lösung, wenn gilt:

1. Heuristik überschätzt nie tatsächliche Kosten:

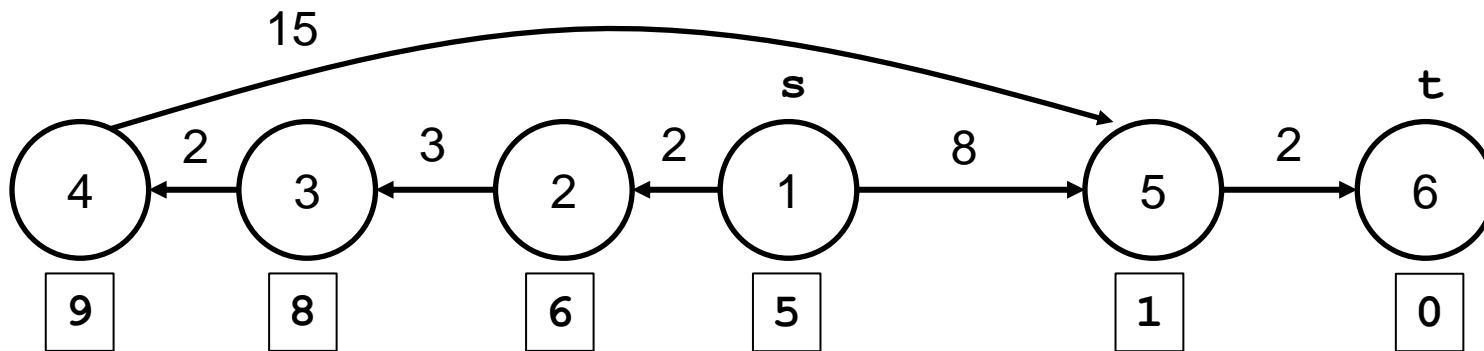
$$u.\text{heur} \leq \text{shortest}(u, t)$$

(insb.  $t.\text{heur} == 0$ )

und

2. Heuristik ist monoton, d.h. für alle  $(u, v) \in E$  gilt:

$$u.\text{heur} \leq w(u, v) + v.\text{heur}$$



# A\*-Algorithmus (IV)

(nicht-negative Kantengewichte)

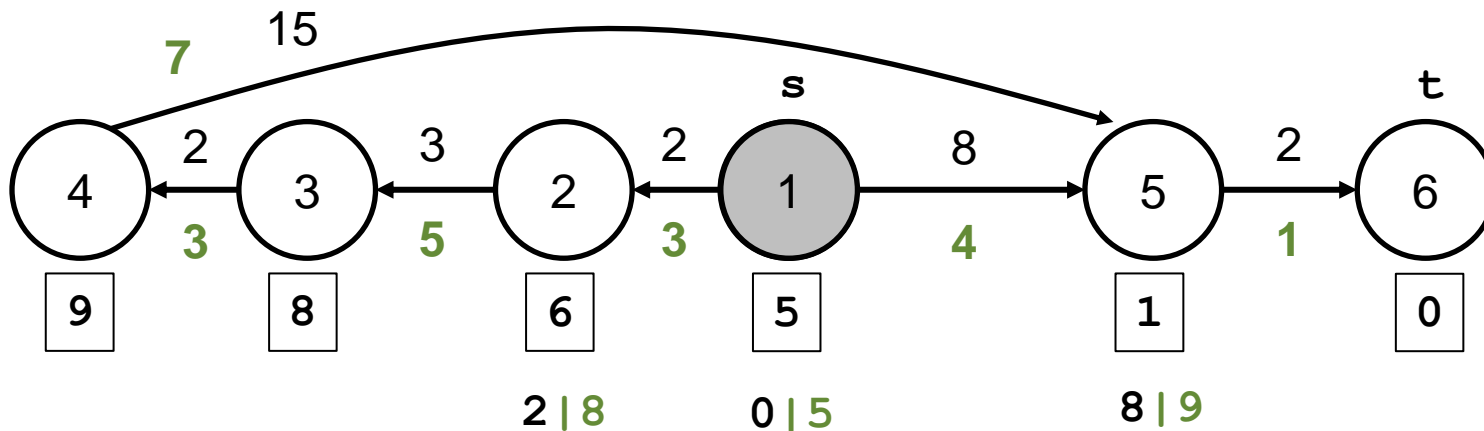


„Dijkstra ist A\* mit Heuristik 0“

„A\* mit monotoner Heuristik ist Dijkstra mit Kantengewichten  $w(u, v) + v.\text{heur} - u.\text{heur}$  und  $s.\text{dist} = s.\text{heur}$ “

Kantengewichte sind nicht-negativ wegen Monotonie

A\* und Dijkstra wählen dann jeweils gleichen Knoten, da  $u.\text{dist} + u.\text{heur} = u.\text{dist}$ ,  
und am Ende  $t.\text{dist} = t.\text{dist}$

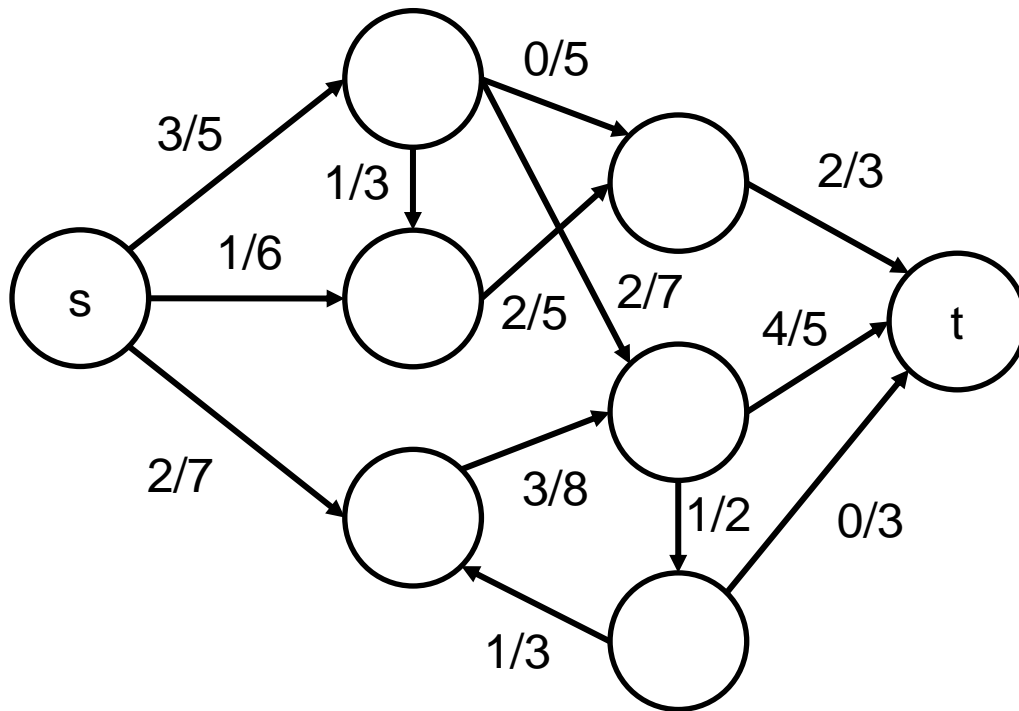


---

# Maximaler Fluss in Graphen

# Netzwerkflüsse: Idee

Kanten haben  
(aktuellen) Flusswert  
und (maximale) Kapazität



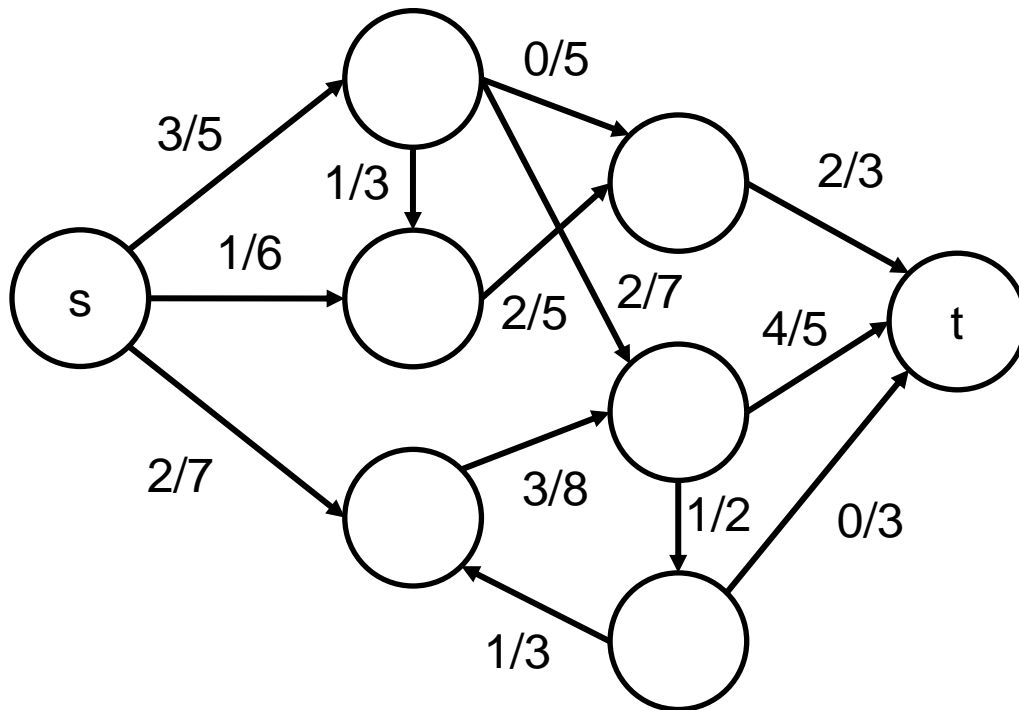
Jeder Knoten außer s und t  
hat gleichen  
eingehenden und  
ausgehenden Fluss

Ziel:  
Finde maximalen Fluss  
von s nach t

# Netzwerkflüsse

Insbesondere  
 $|E| \geq |V| - 1$

Ein **Flussnetzwerk** ist ein gewichteter, gerichteter Graph  $G = (V, E)$  mit Kapazität(-sgewicht)  $c$ , so dass  $c(u, v) \geq 0$  für  $(u, v) \in E$  und  $c(u, v) = 0$  für  $(u, v) \notin E$ , mit zwei Knoten  $s, t \in V$  (Quelle und Senke), so dass jeder Knoten von  $s$  aus erreichbar ist und  $t$  von jedem Knoten aus erreichbar ist.



Ein **Fluss**  $f: V \times V \rightarrow \mathbb{R}$  für ein Flussnetzwerk  $G = (V, E)$  mit Kapazität  $c$  und Quelle  $s$  und Senke  $t$  erfüllt  $0 \leq f(u, v) \leq c(u, v)$  für alle  $u, v \in V$ , sowie für alle  $u \in V - \{s, t\}$ :

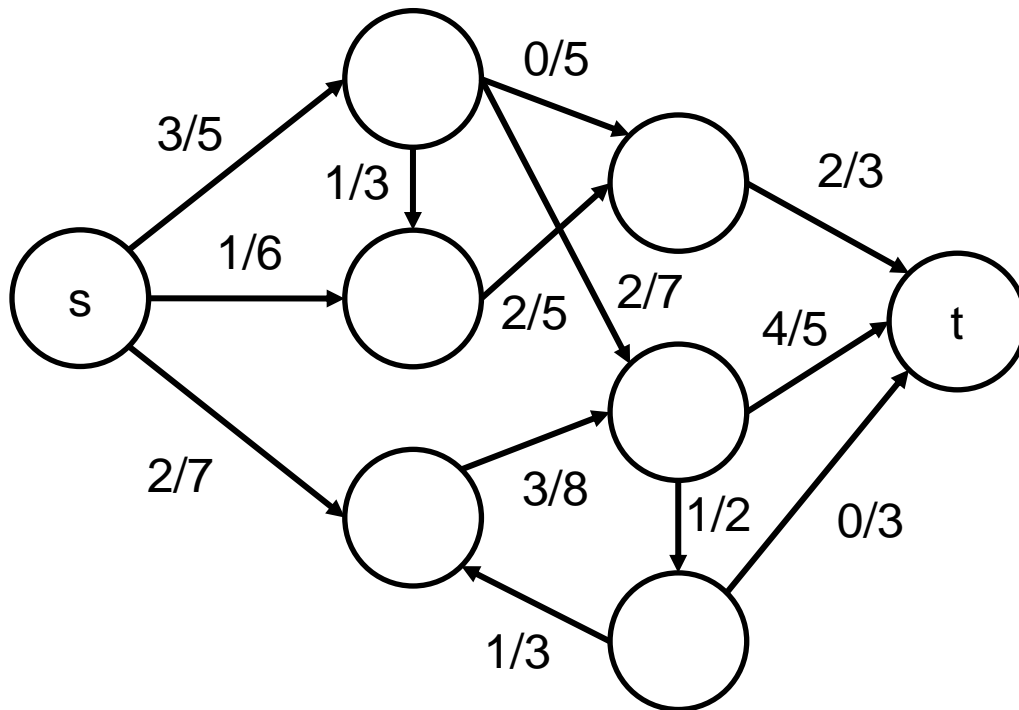
$$\sum_{v \in V} f(u, v) = \sum_{v \in V} f(v, u)$$

# Maximale Flüsse

Beispiel:  $|f| = 6$ ,  
aber nicht maximal,  
da z.B. noch +1  
über obere Kanten

Der **Wert**  $|f|$  eines **Flusses**  $f: V \times V \rightarrow \mathbb{R}$  für ein Flussnetzwerk  $G = (V, E)$  mit Kapazität  $c$  und Quelle  $s$  und Senke  $t$  ist

$$|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$$

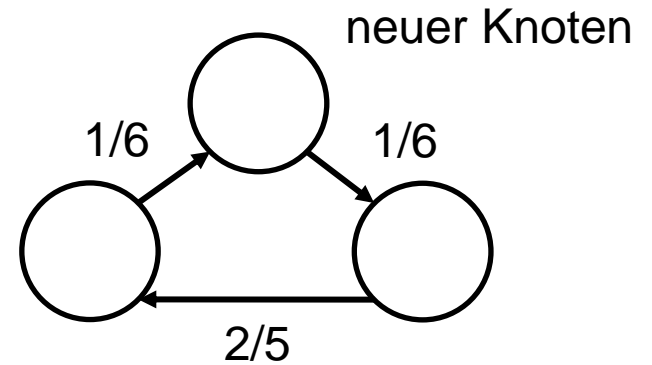
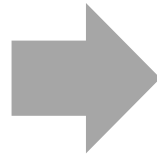
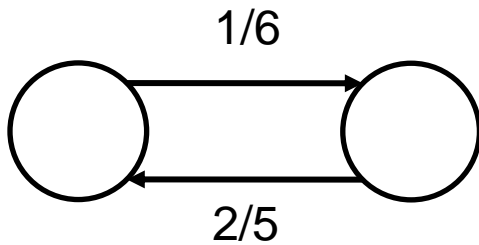


Ein **Fluss**  $f: V \times V \rightarrow \mathbb{R}$  für ein Flussnetzwerk  $G = (V, E)$  mit Kapazität  $c$  und Quelle  $s$  und Senke  $t$  erfüllt  $0 \leq f(u, v) \leq c(u, v)$  für alle  $u, v \in V$ , sowie für alle  $u \in V - \{s, t\}$ :

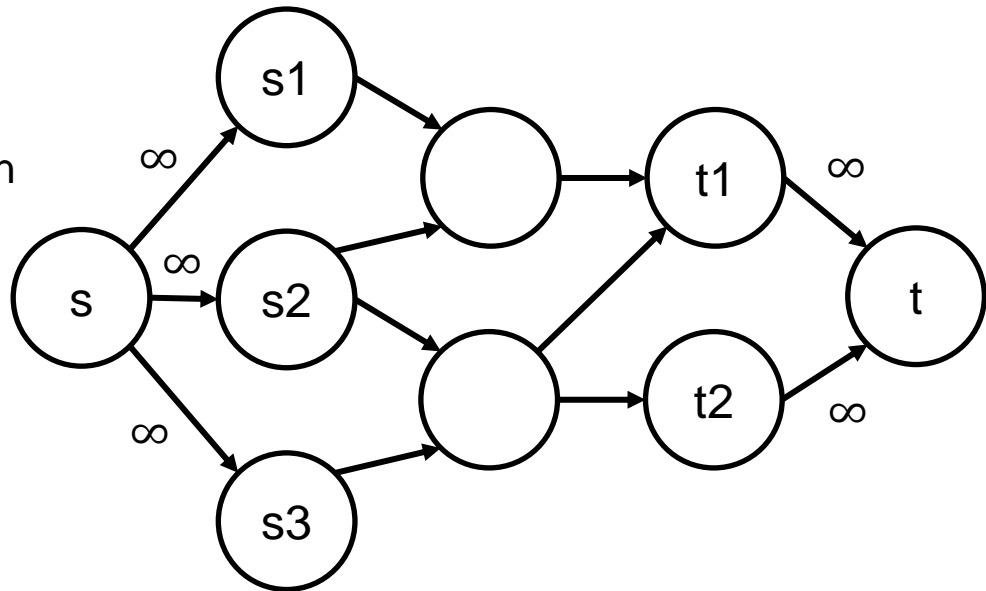
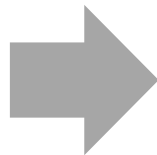
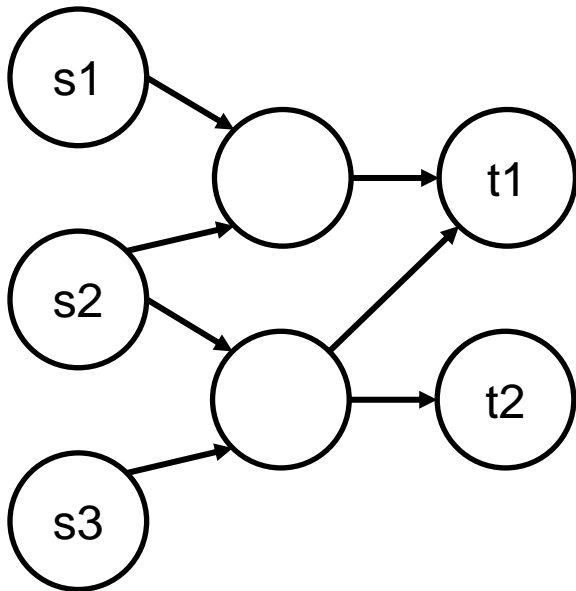
$$\sum_{v \in V} f(u, v) = \sum_{v \in V} f(v, u)$$

# Transformationen

Eliminiere  
antiparallele Kanten



Vereinige  
Quellen  
und Senken

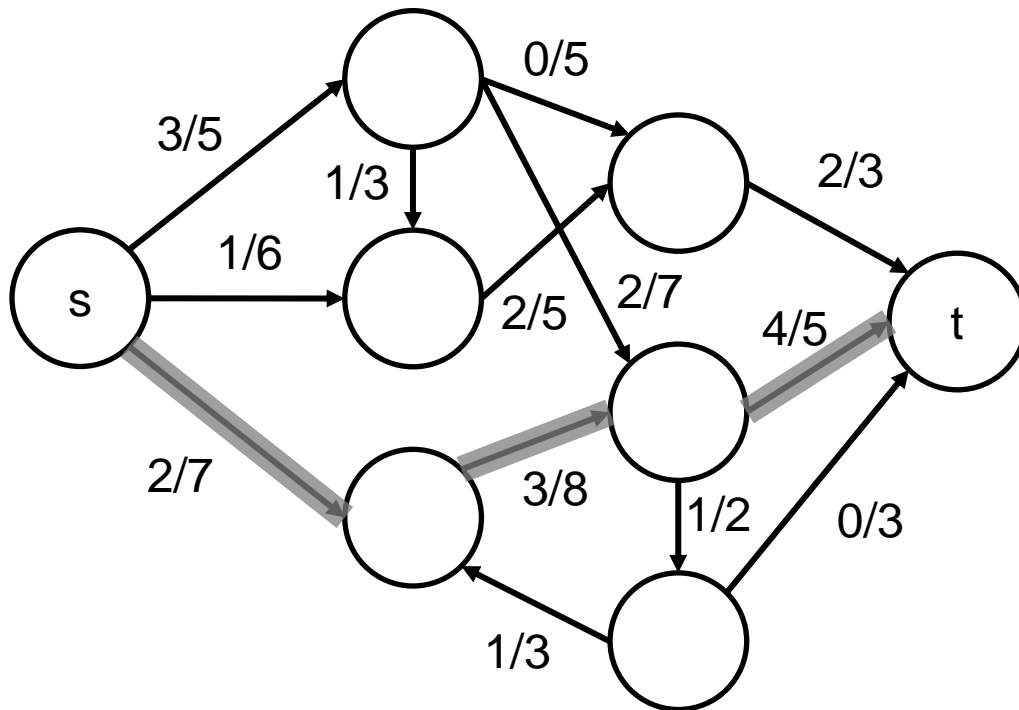




# Ford-Fulkerson-Methode

Idee: Suche Pfad von s nach t, der noch erweiterbar (bzgl. des Flusses) ist

Aber: Pfad suchen wir im „Restkapazitäts“-Graph  $G_f$ ,  
der die möglichen Zu- und Abflüsse beschreibt



# Reste

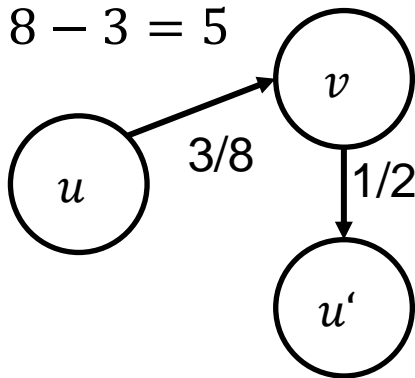
Restkapazität:

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{falls } (u, v) \in E \\ f(v, u) & \text{falls } (v, u) \in E \\ 0 & \text{sonst} \end{cases}$$

„Wieviel eingehenden Fluss über  $(u, v)$  könnte man noch zu  $v$  hinzufügen?“

„Wieviel abgehenden Fluss über  $(v, u)$  könnte man wegnehmen und damit quasi zu  $v$  hinzufügen?“

$$c_f(u, v) = 8 - 3 = 5$$



$$c_f(u', v) = 1$$

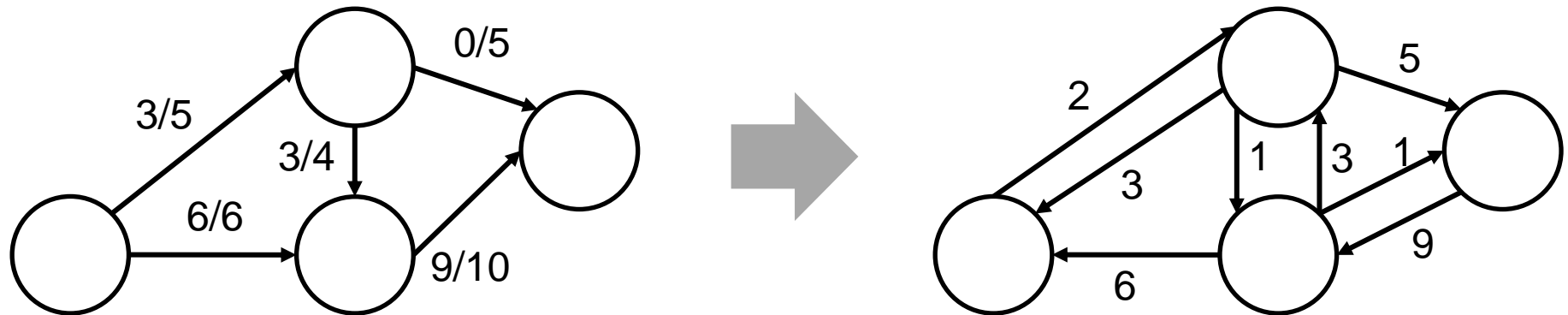
Bemerkung: nach Voraussetzung nicht beide Kanten  $(u, v)$ ,  $(v, u)$  im Netzwerk, daher wohldefiniert

# Restkapazitäts-Graph

$G_f = (V, E_f)$  mit  $E_f = \{ (u, v) \in V \times V \mid c_f(u, v) > 0 \}$

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{falls } (u, v) \in E \\ f(v, u) & \text{falls } (v, u) \in E \\ 0 & \text{sonst} \end{cases}$$

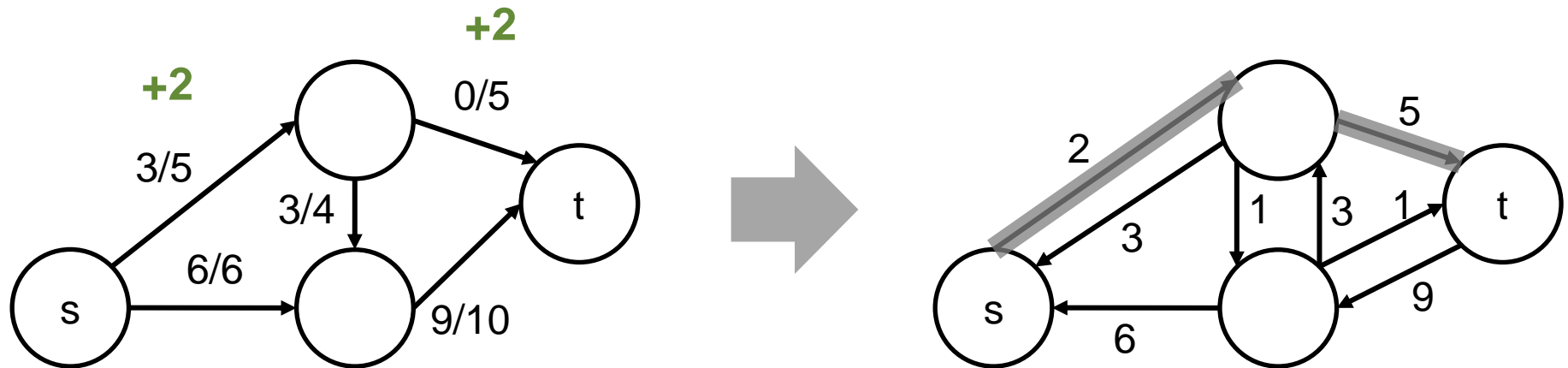
Bemerkung:  
Im Restkapazitäts-Graph  
sind antiparallele Kanten  
erlaubt



# Restkapazitäten ausnutzen

$G_f = (V, E_f)$  mit  $E_f = \{ (u, v) \in V \times V \mid c_f(u, v) > 0 \}$

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v) & \text{falls } (u, v) \in E \\ f(v, u) & \text{falls } (v, u) \in E \\ 0 & \text{sonst} \end{cases}$$



Finde Pfad von  $s$  zu  $t$  in  $G_f$  und erhöhe (für Kanten in  $G$ ) bzw. erniedrige (für Nicht-Kanten) um Minimum  $c_f(u, v)$  aller Werte auf dem Pfad in  $G$

# Ford-Fulkerson-Algorithmus

Ford-Fulkerson( $G, s, t, c$ )

```
1  FOREACH  $e$  in  $E$  DO  $e.\text{flow}=0$ ;  
2  WHILE there is path  $p$  from  $s$  to  $t$  in  $G_{\text{flow}}$  DO  
3       $c_{\text{flow}}(p) = \min \{c_{\text{flow}}(u, v) \mid (u, v) \text{ in } p\}$   
4      FOREACH  $e$  in  $p$  DO  
5          IF  $e$  in  $E$  THEN  
6               $e.\text{flow}=e.\text{flow}+ c_{\text{flow}}(p)$   
7          ELSE  
8               $e.\text{flow}=e.\text{flow}- c_{\text{flow}}(p)$ 
```

Z.B. wenn  
in jeder Iteration  
der Fluss nur  
um  $1/u = 0.1$   
erhöht wird

Laufzeit =  $O(|E| \cdot u \cdot |f^*|)$

Pfadsuche z.B. per BFS oder DFS

(wobei  $f^*$  maximaler Fluss  
und Fluss um bis zu  $1/u$  pro Iteration wächst)

Laufzeit =  $O(|V| \cdot |E|^2)$

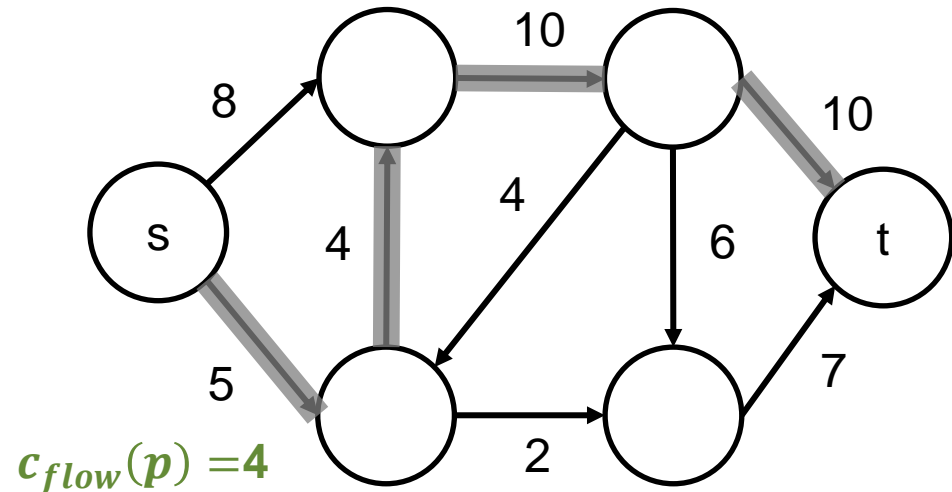
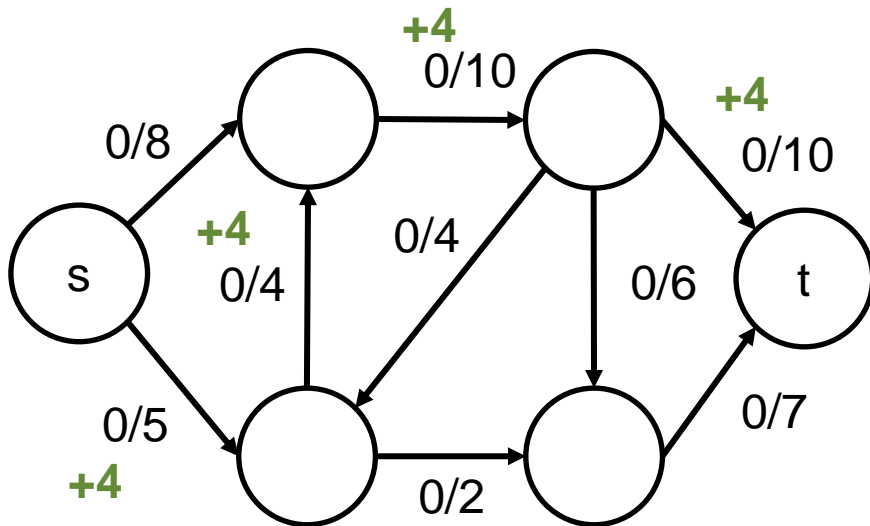
(mit Verbesserung  
nach Edmonds-Karp)

# Ford-Fulkerson-Algorithmus: Beispiel (I)

Ford-Fulkerson( $G, s, t, c$ )

```
1  FOREACH  $e$  in  $E$  DO  $e.\text{flow}=0$ ;  
2  WHILE there is path  $p$  from  $s$  to  $t$  in  $G_{\text{flow}}$  DO  
3     $c_{\text{flow}}(p) = \min \{c_{\text{flow}}(u, v) \mid (u, v) \text{ in } p\}$   
4    FOREACH  $e$  in  $p$  DO  
5      IF  $e$  in  $E$  THEN  
6         $e.\text{flow} = e.\text{flow} + c_{\text{flow}}(p)$   
7      ELSE  
8         $e.\text{flow} = e.\text{flow} - c_{\text{flow}}(p)$ 
```

Restkapazitäts-  
graph



# Ford-Fulkerson-Algorithmus: Beispiel (II)

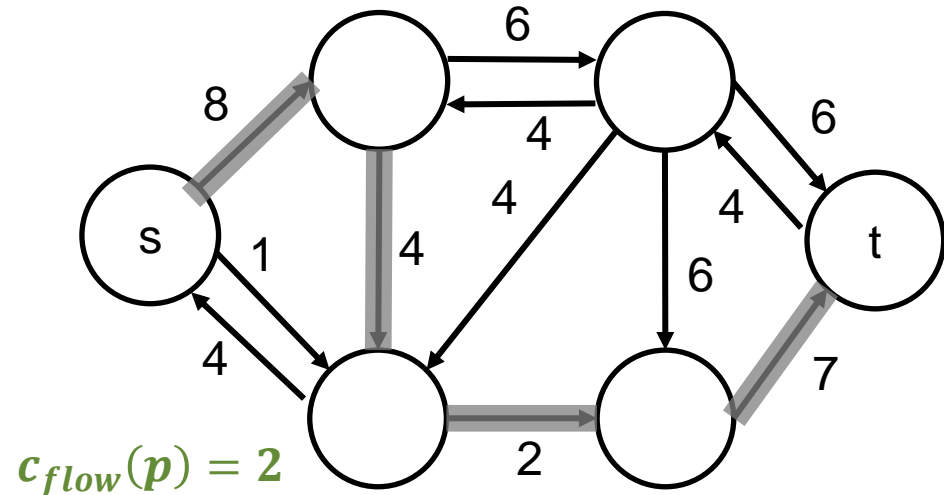
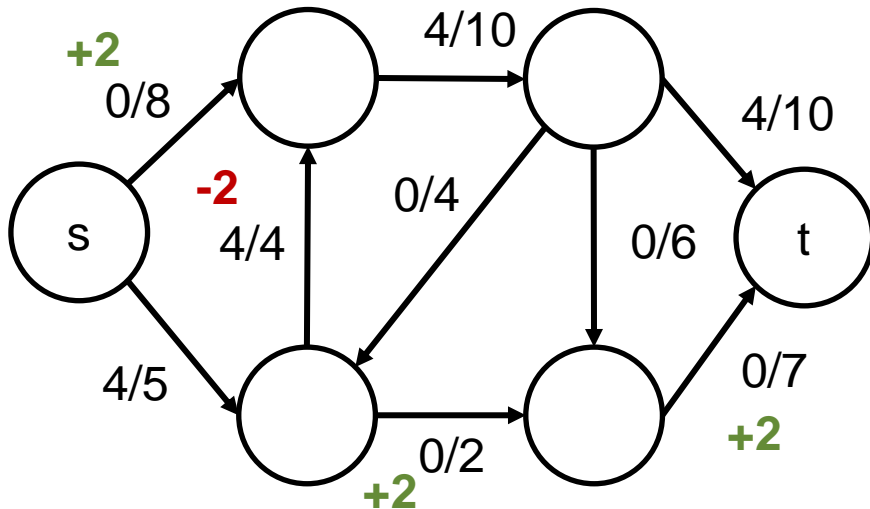
Ford-Fulkerson( $G, s, t, c$ )

```

1  FOREACH  $e$  in  $E$  DO  $e.\text{flow}=0$ ;
2  WHILE there is path  $p$  from  $s$  to  $t$  in  $G_{\text{flow}}$  DO
3       $c_{\text{flow}}(p) = \min \{c_{\text{flow}}(u, v) \mid (u, v) \text{ in } p\}$ 
4      FOREACH  $e$  in  $p$  DO
5          IF  $e$  in  $E$  THEN
6               $e.\text{flow}=e.\text{flow}+ c_{\text{flow}}(p)$ 
7          ELSE
8               $e.\text{flow}=e.\text{flow}- c_{\text{flow}}(p)$ 

```

Restkapazitäts-  
graph



# Ford-Fulkerson-Algorithmus: Beispiel (III)

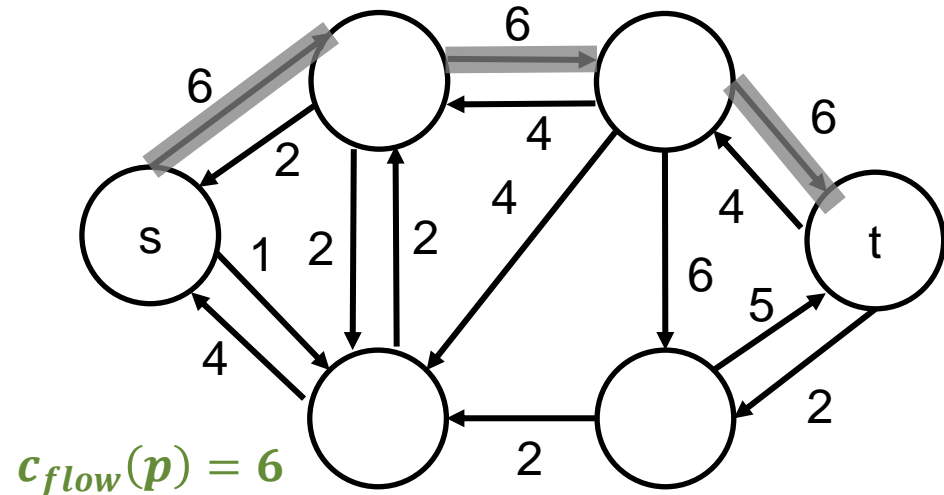
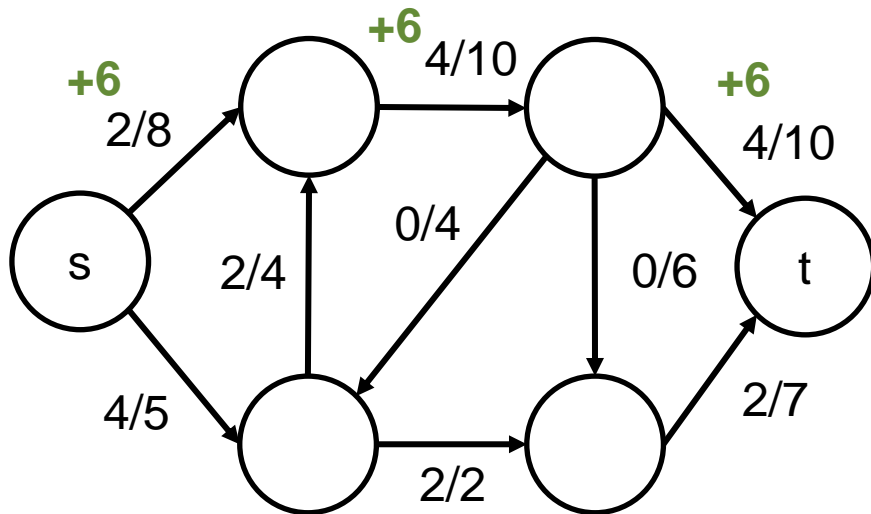
Ford-Fulkerson( $G, s, t, c$ )

```

1  FOREACH  $e$  in  $E$  DO  $e.\text{flow}=0$ ;
2  WHILE there is path  $p$  from  $s$  to  $t$  in  $G_{\text{flow}}$  DO
3       $c_{\text{flow}}(p) = \min \{c_{\text{flow}}(u, v) \mid (u, v) \text{ in } p\}$ 
4      FOREACH  $e$  in  $p$  DO
5          IF  $e$  in  $E$  THEN
6               $e.\text{flow}=e.\text{flow}+ c_{\text{flow}}(p)$ 
7          ELSE
8               $e.\text{flow}=e.\text{flow}- c_{\text{flow}}(p)$ 

```

Restkapazitäts-  
graph





# Ford-Fulkerson-Algorithmus: Beispiel (IV)

Ford-Fulkerson( $G, s, t, c$ )

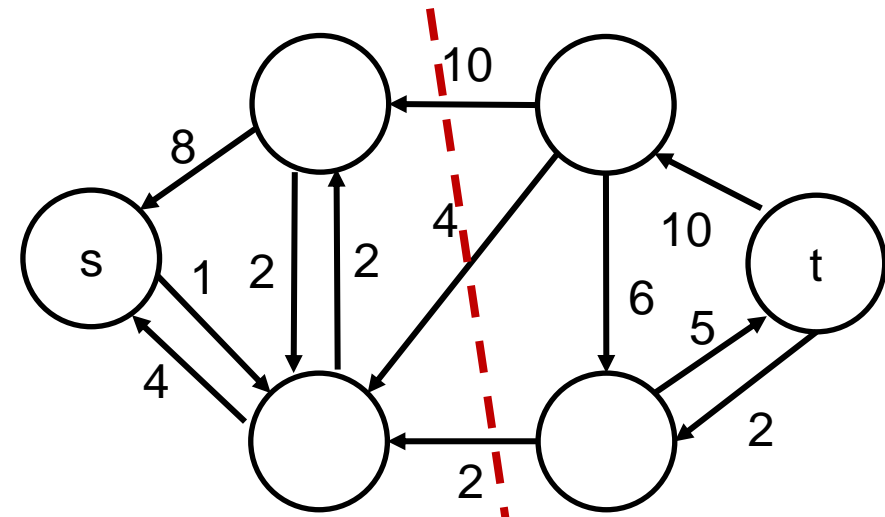
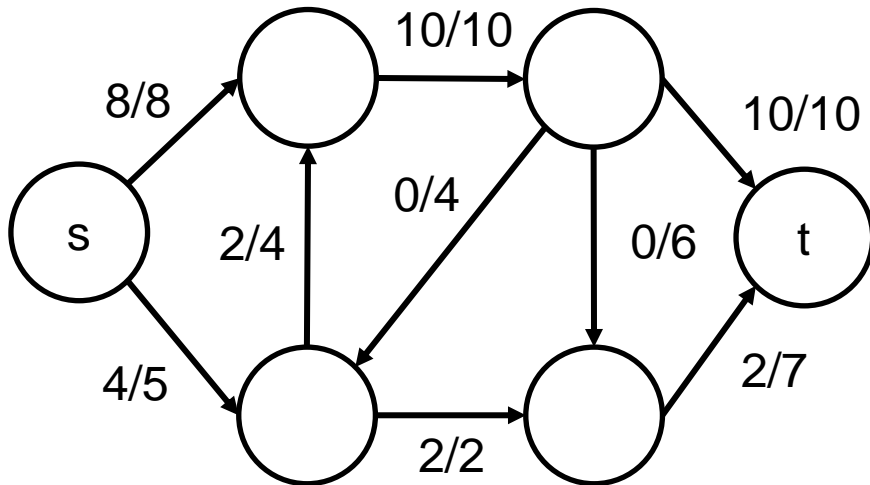
```

1  FOREACH  $e$  in  $E$  DO  $e.\text{flow}=0$ ;
2  WHILE there is path  $p$  from  $s$  to  $t$  in  $G_{\text{flow}}$  DO
3       $c_{\text{flow}}(p) = \min \{c_{\text{flow}}(u, v) \mid (u, v) \text{ in } p\}$ 
4      FOREACH  $e$  in  $p$  DO
5          IF  $e$  in  $E$  THEN
6               $e.\text{flow}=e.\text{flow}+ c_{\text{flow}}(p)$ 
7          ELSE
8               $e.\text{flow}=e.\text{flow}- c_{\text{flow}}(p)$ 

```

Kein Pfad  
mehr im  
Restkapazitäts-  
Graph

Restkapazitäts-  
graph

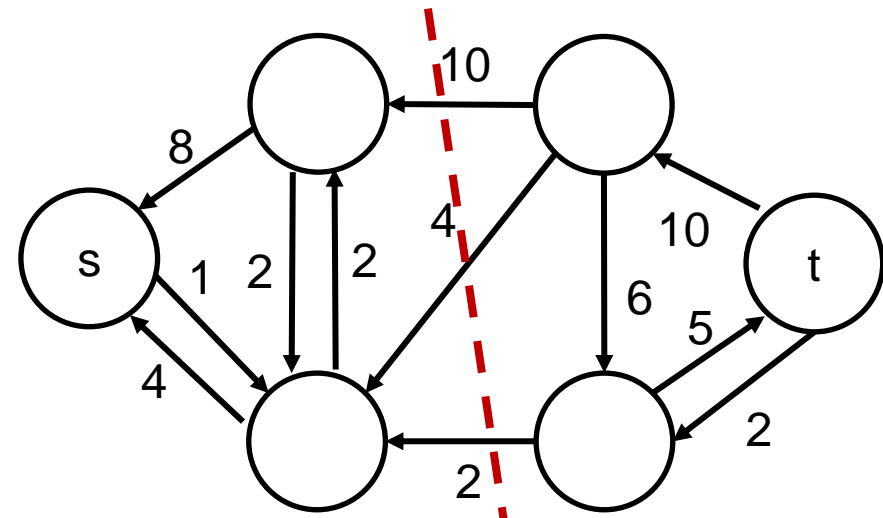
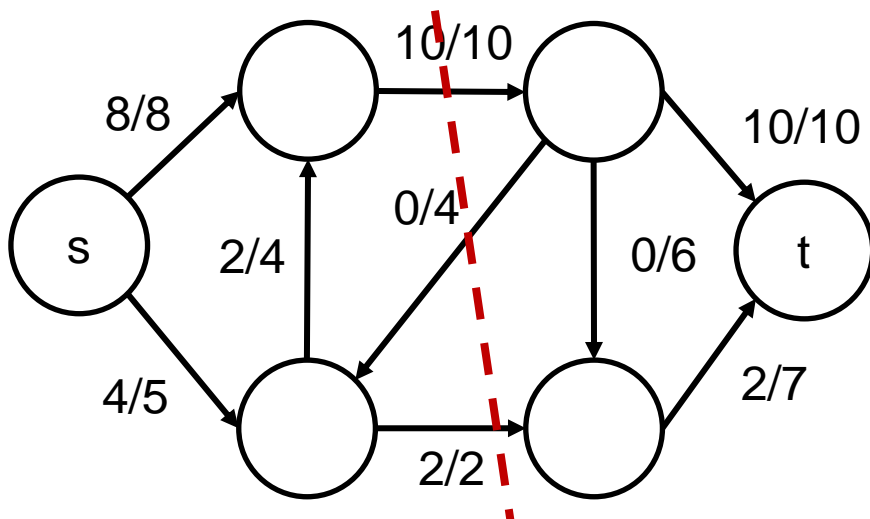


# Max-Flow Min-Cut Theorem (I)

Korrektheit des Ford-Fulkerson-Algorithmus

Sei  $f: V \times V \rightarrow \mathbb{R}$  Fluss für ein Flussnetzwerk  $G = (V, E)$  mit Kapazität  $c$  und Quelle  $s$  und Senke  $t$ . Dann sind äquivalent:

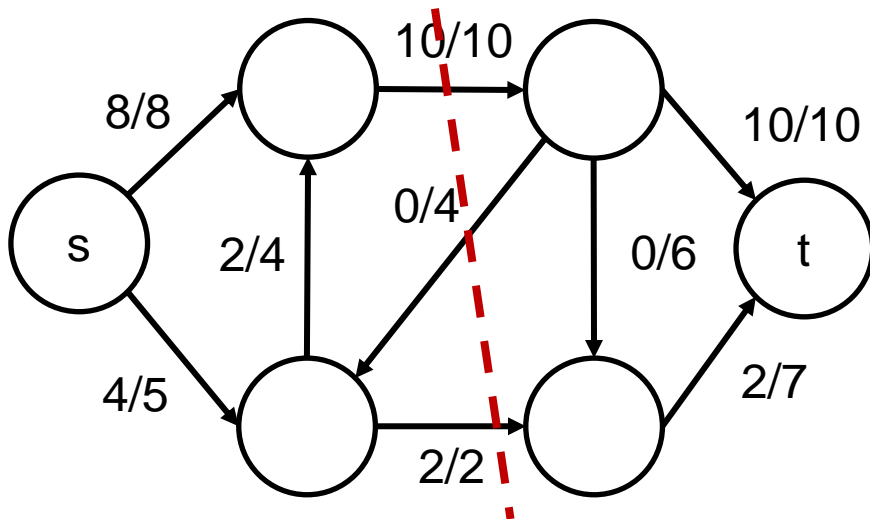
1.  $f$  ist ein maximaler Fluss für  $G$
2. Der Restkapazitätsgraph  $G_f$  enthält keinen erweiterbaren Pfad
- 3.



# Max-Flow Min-Cut Theorem (II)

Sei  $f: V \times V \rightarrow \mathbb{R}$  Fluss für ein Flussnetzwerk  $G = (V, E)$  mit Kapazität  $c$  und Quelle  $s$  und Senke  $t$ . Dann sind äquivalent:

1.  $f$  ist ein maximaler Fluss für  $G$
2. Der Restkapazitätsgraph  $G_f$  enthält keinen erweiterbaren Pfad
3.  $|f| = \min_S c(S, V - S)$  mit  $s \in S$  und  $t \in V - S$



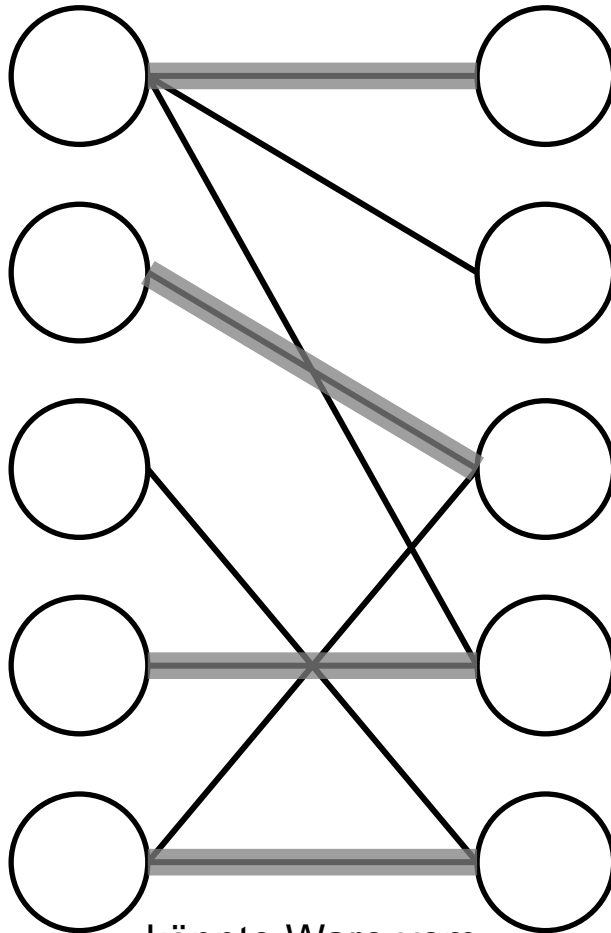
wobei

$$c(S, V - S) = \sum_{u \in S} \sum_{v \in V - S} c(u, v)$$

für  $s \in S$  und  $t \in V - S$  die  
**Kapazität eines Schnitts**  $(S, V - S)$   
ist.

# Beispielanwendung: Bipartites Matching (I)

Käufer



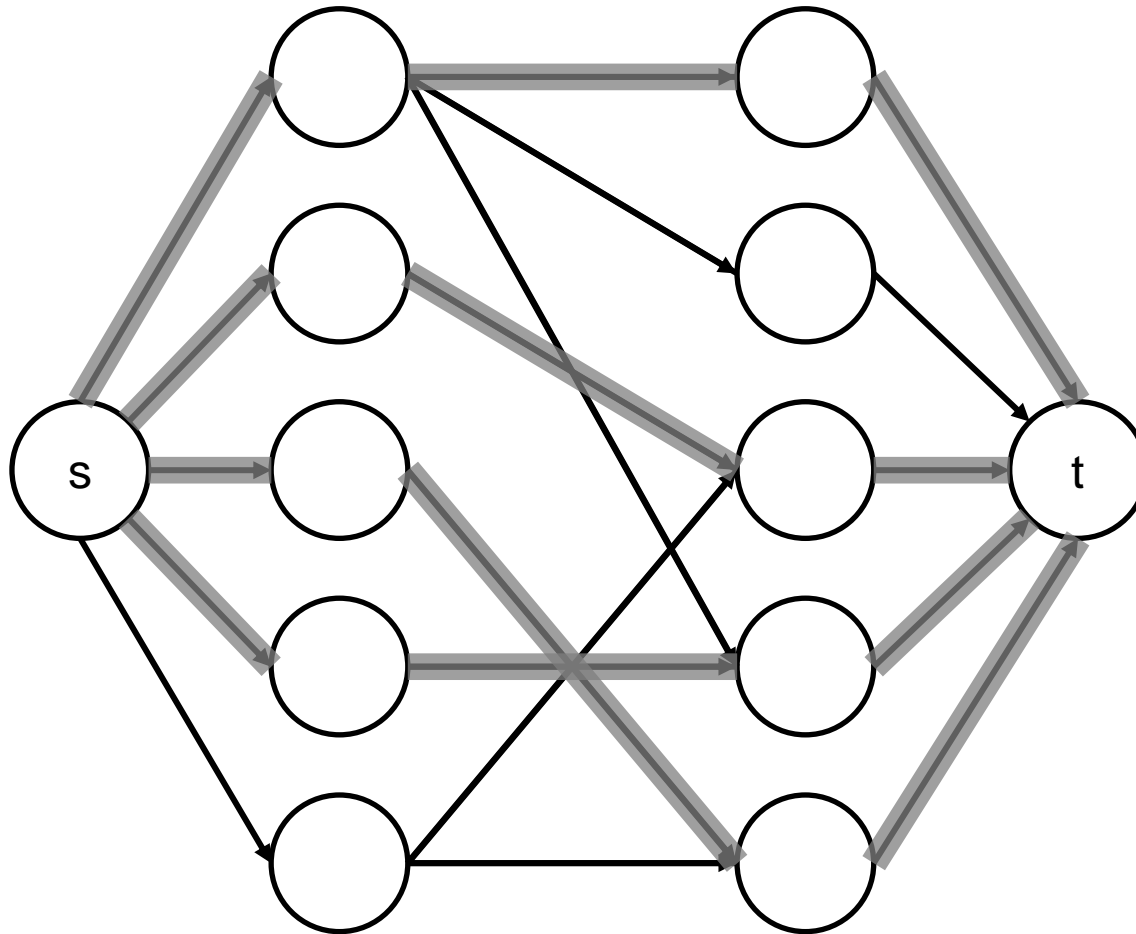
könnte Ware vom  
Verkäufer kaufen

Verkäufer

bipartiter Graph:  
Knotenmenge zerfällt in  
zwei disjunkte Mengen,  
so dass Kanten nur  
zwischen den Mengen

maximales bipartites Matching:  
Finde maximale Anzahl  
von Kanten, so dass  
jeder Käufer genau einem  
Verkäufer zugeordnet wird

# Beispielanwendung: Bipartites Matching (II)



Kapazität = 1  
für jede Kante

Fluss=1 sagt,  
Kante aktiv

Wert des Flusses  
gibt an, wie viele  
aktive Kanten aus  
s ausgehen bzw.  
wie viele in t ankommen



Zeigen Sie, dass der Wert  $|f|$  eines Flusses –definiert als der Netto-Ausfluss der Quelle– auch das eingehende Netto in der Senke beschreibt.



Angenommen, Sie beschreiben ein Job-Scheduling-Problem per Dag, wobei jeder Job eine gewisse Zeit benötigt, bevor er beendet ist. Wie können Sie mit Hilfe des Dag-SSSP-Algorithmus bestimmen, wie lange ihr Projekt insgesamt braucht?

