Algorithmen und Datenstrukturen

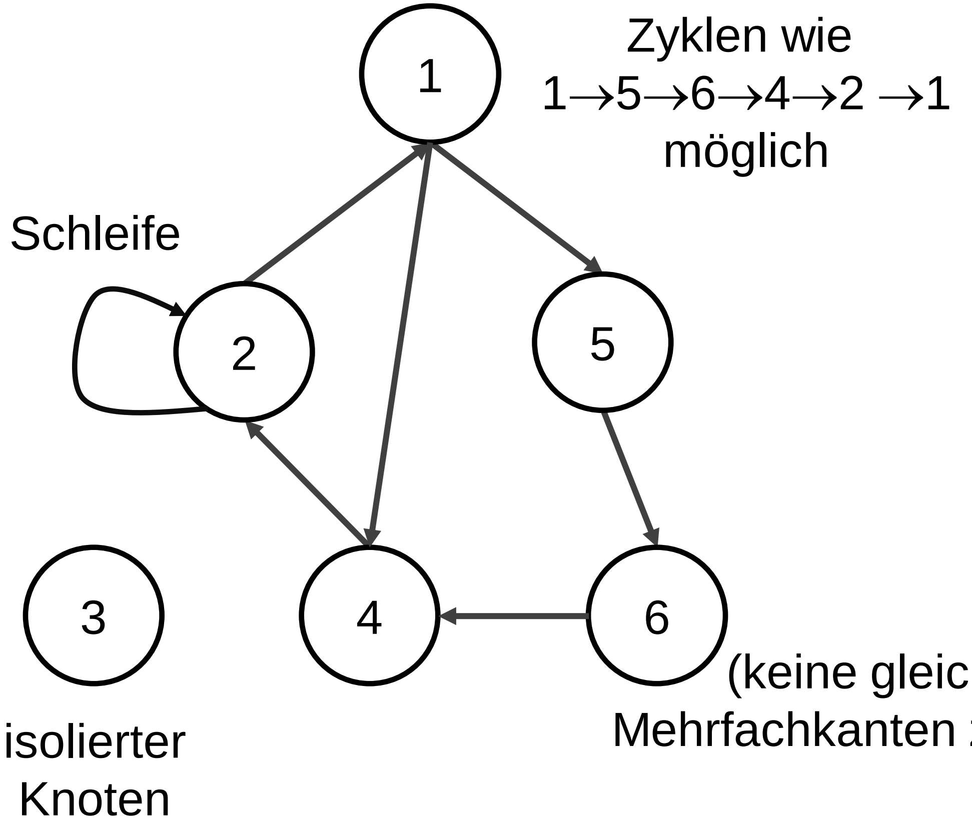
Prof. Marc Fischlin, SS 2024

06

Graphen-Algorithmen

Graphen

(Endliche) Gerichtete Graphen



(keine gleichgerichteten Mehrfachkanten zwischen Knoten)

Ein **(endlicher) gerichteter Graph $\boldsymbol{G = (V, E)}$** besteht aus

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

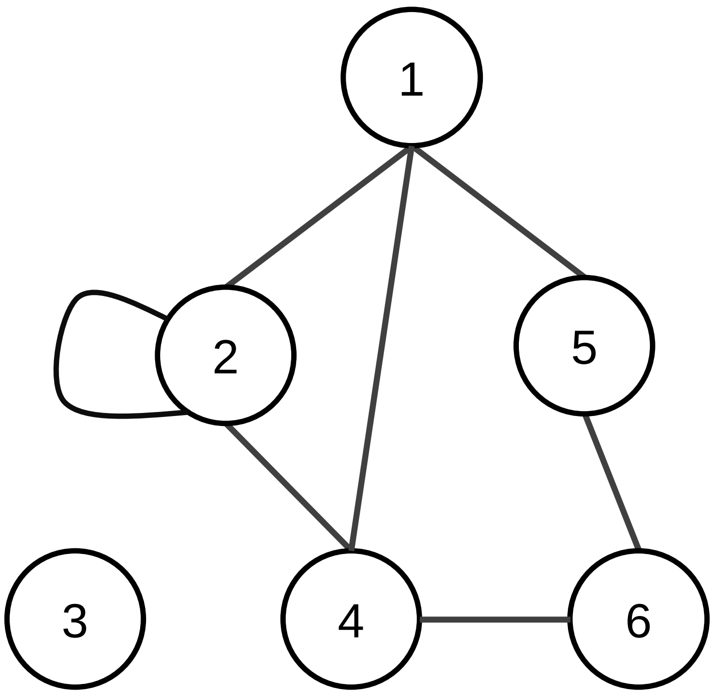
(2) einer (endlichen) Kantenmenge $E \subseteq V \times V$ („edges“)

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

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

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** $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ besteht aus

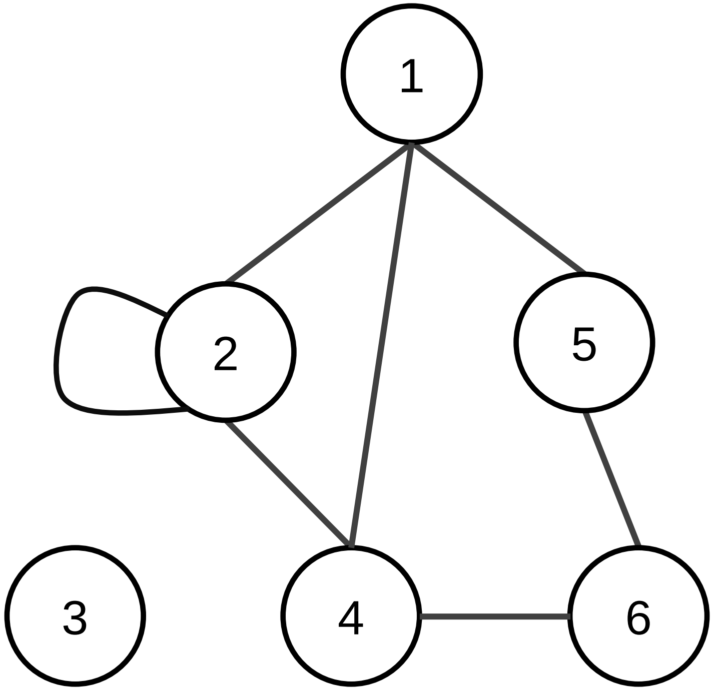
(1) einer (endlichen) Knotenmenge V („vertices“)

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

Alternative Darstellung:

$-\!-\!-$	$\{u,v\}$
statt	statt
$\leftrightarrows$	$(u,v),(v,u)$

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 $w_k = v$.

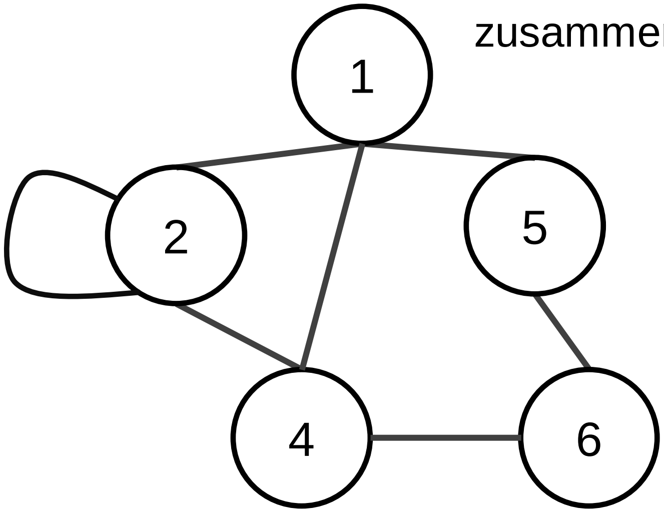
Insbesondere ist u immer von u
per „leerem Pfad" ($k = 1$) erreichbar.

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.

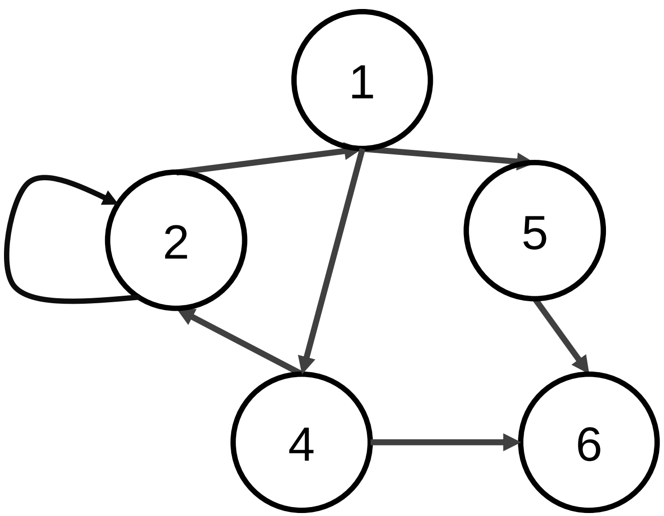
$shortest(u, v) =$ Länge eines kürzesten Pfades von u nach v

Zusammenhänge



zusammenhängend

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

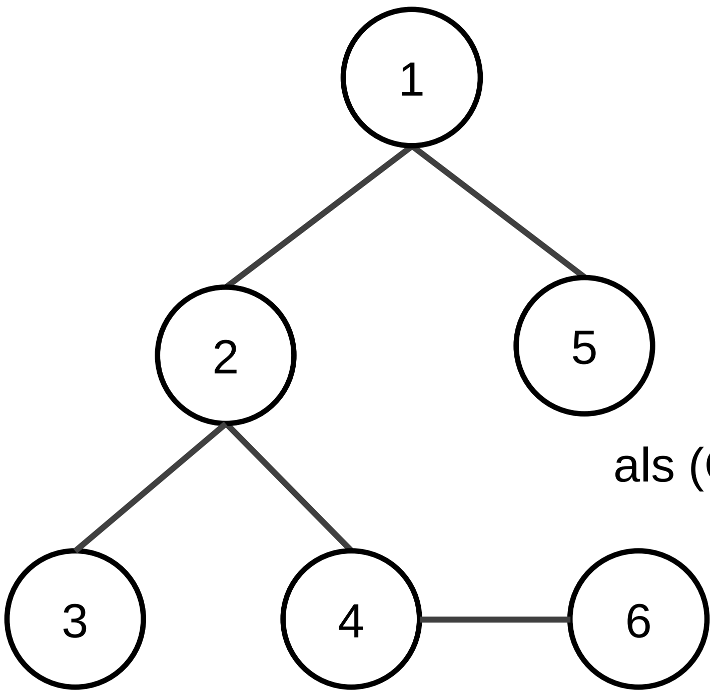


nicht stark
zusammenhängend,
da kein Pfad 5→1

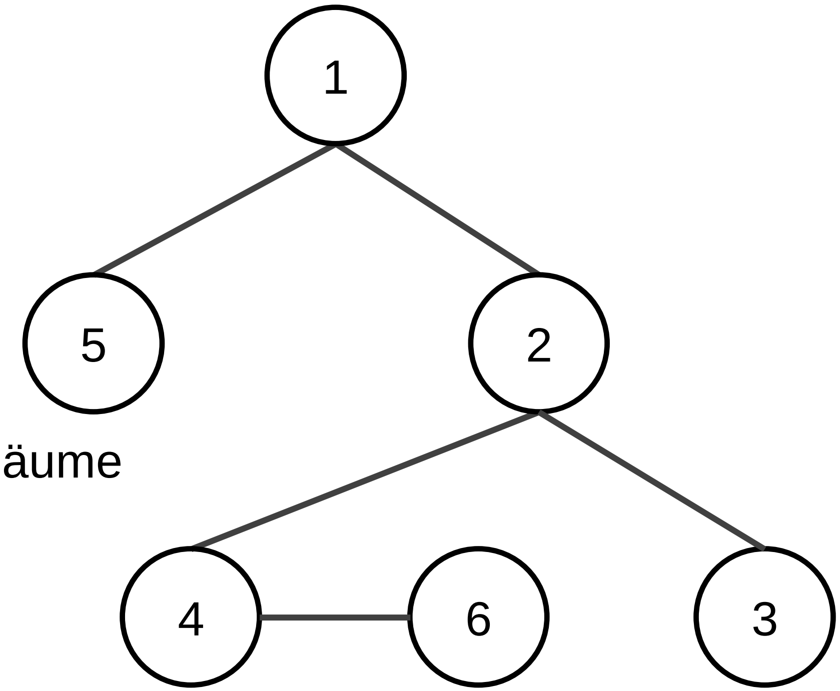
Gerichteter Graph
ist **stark zusammenhängend**,
wenn jeder Knoten von jedem
anderen Knoten aus
(gemäß Kantenrichtung)
erreichbar ist

Graphen und Bäume

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



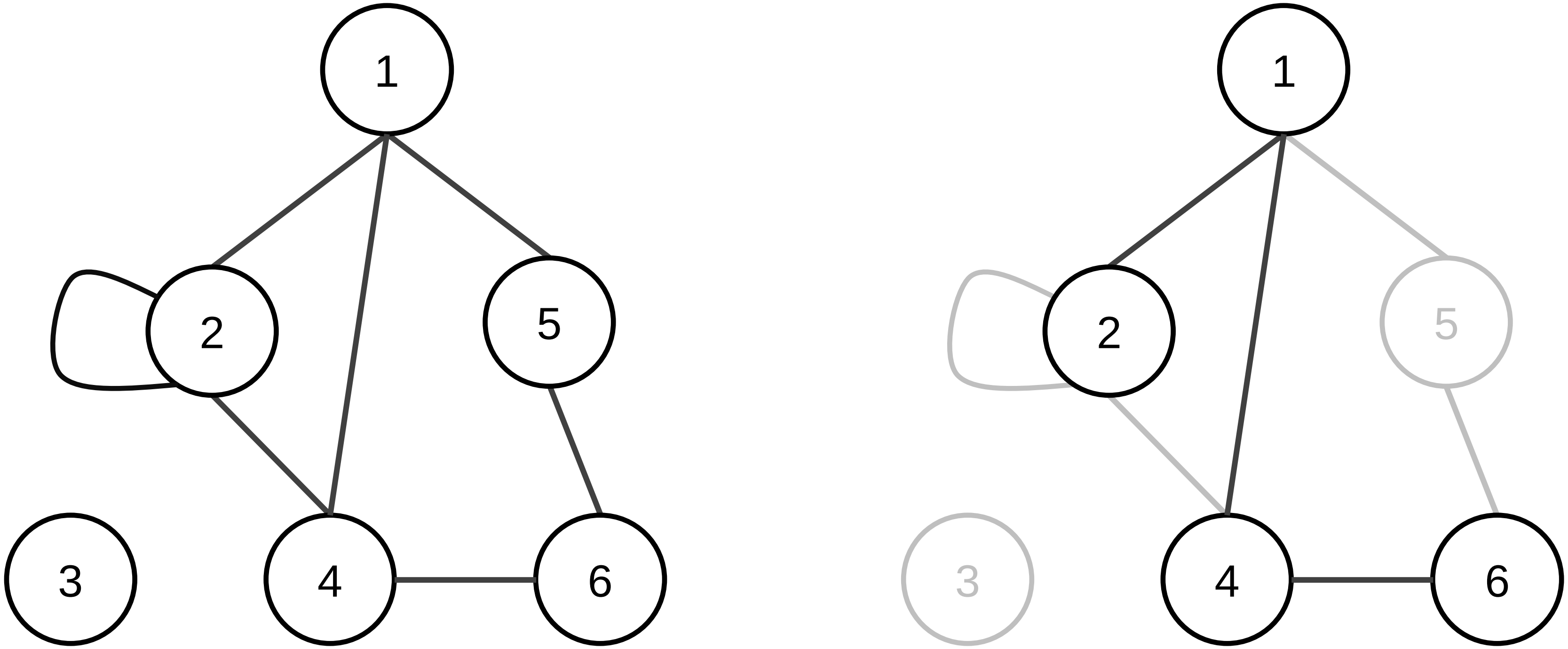
als (Graphen-)Bäume
=



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

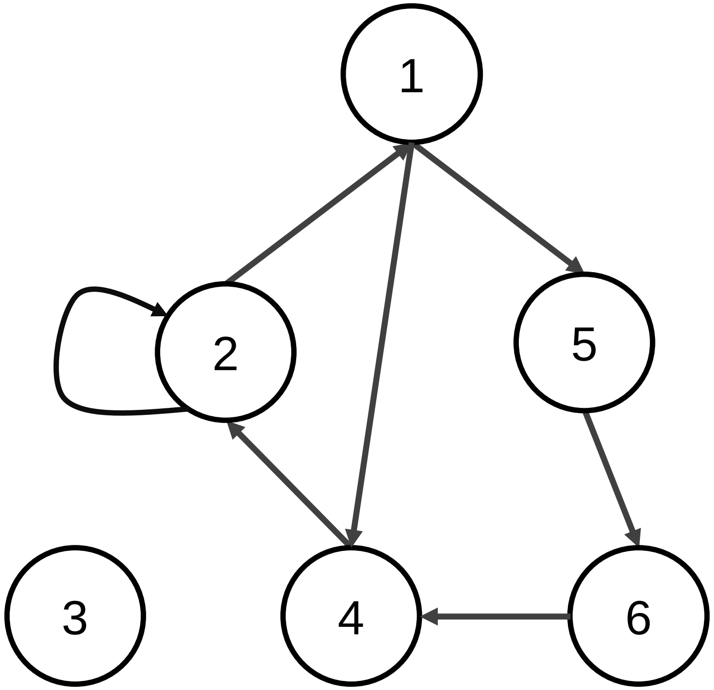
Subgraphen

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 $\mathrm{E}' \subseteq E$.

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:

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

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

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

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 $m = 0$):

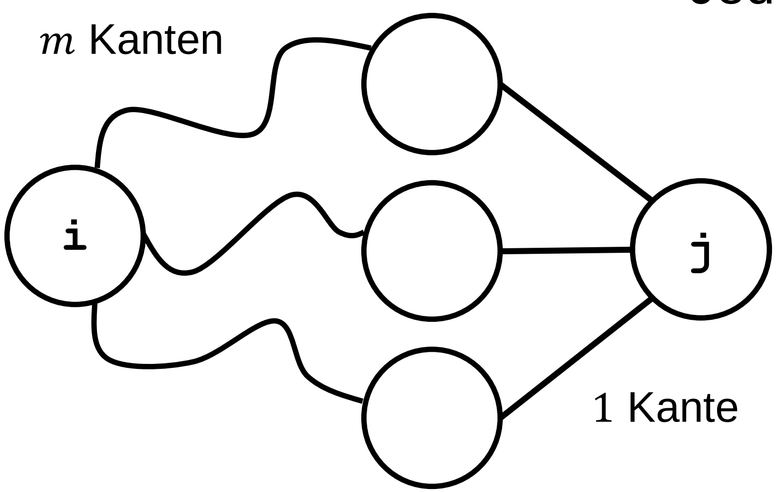
$$A^0 = I = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

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 $m \to m + 1$):



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 $m \to m+1$):

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

$$A^{m+1} = A^m \cdot A$$

$$= \begin{pmatrix} & & & \\ \boxed{a_{i,1}^{(m)} \quad a_{i,2}^{(m)} \quad \cdots \quad a_{i,n}^{(m)}} \\ & & & \end{pmatrix} \cdot \begin{pmatrix} & \boxed{\begin{matrix} a_{1,j} \\ a_{2,j} \\ \vdots \\ a_{n,j} \end{matrix}} & \end{pmatrix}$$

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 $m \to m+1$):

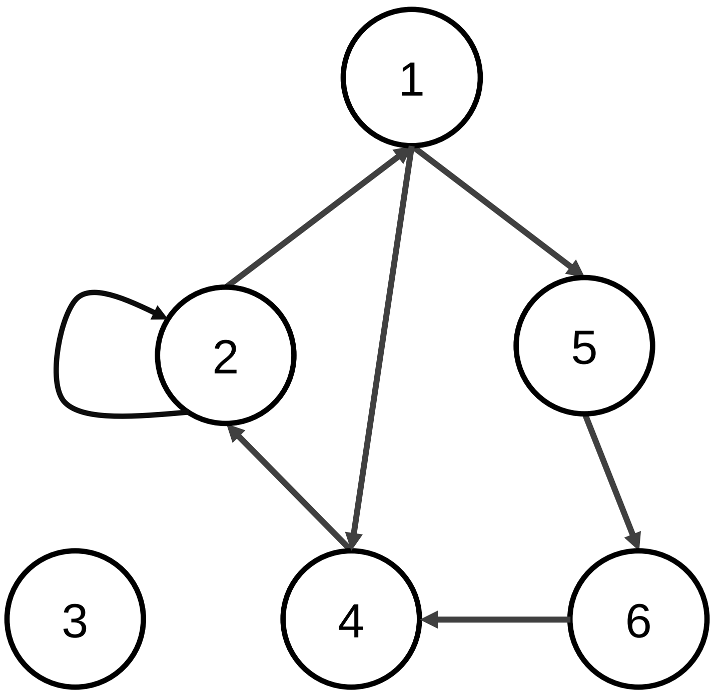
i-te Zeile, j-te Spalte von A^{m+1}: $$a_{i,j}^{(m+1)} = \sum_{k=1}^{n} a_{i,k}^{(m)} \cdot a_{k,j}$$

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

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

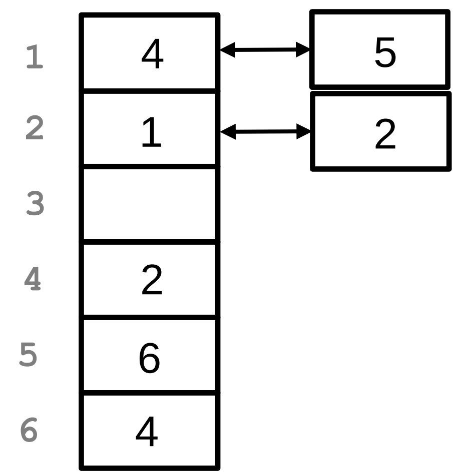
Eintrag beschreibt genau Anzahl der Wege von i über alle k mit jeweils m Kanten und einer Kante von k zu j

Darstellung von Graphen (II)



Adjazenzliste:

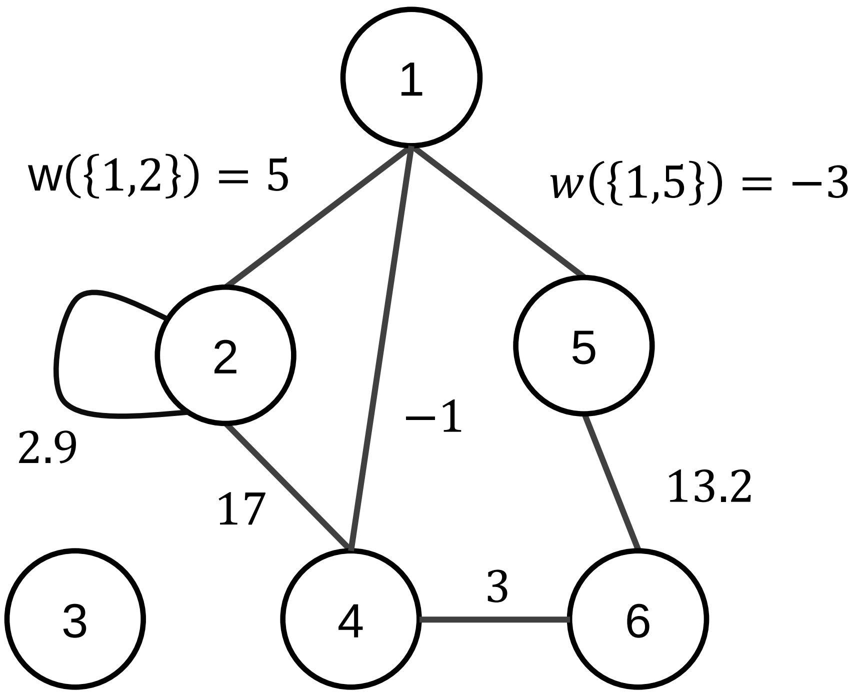
Als Array mit verketteten Listen (sortiert oder unsortiert)



Speicherbedarf $= \Theta(|V| + |E|)$

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

Gewichtete Graphen



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

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

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

Beispiel:
Knoten = Städte
Kanten = Zugverbindungen
Gewicht = Entfernung

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

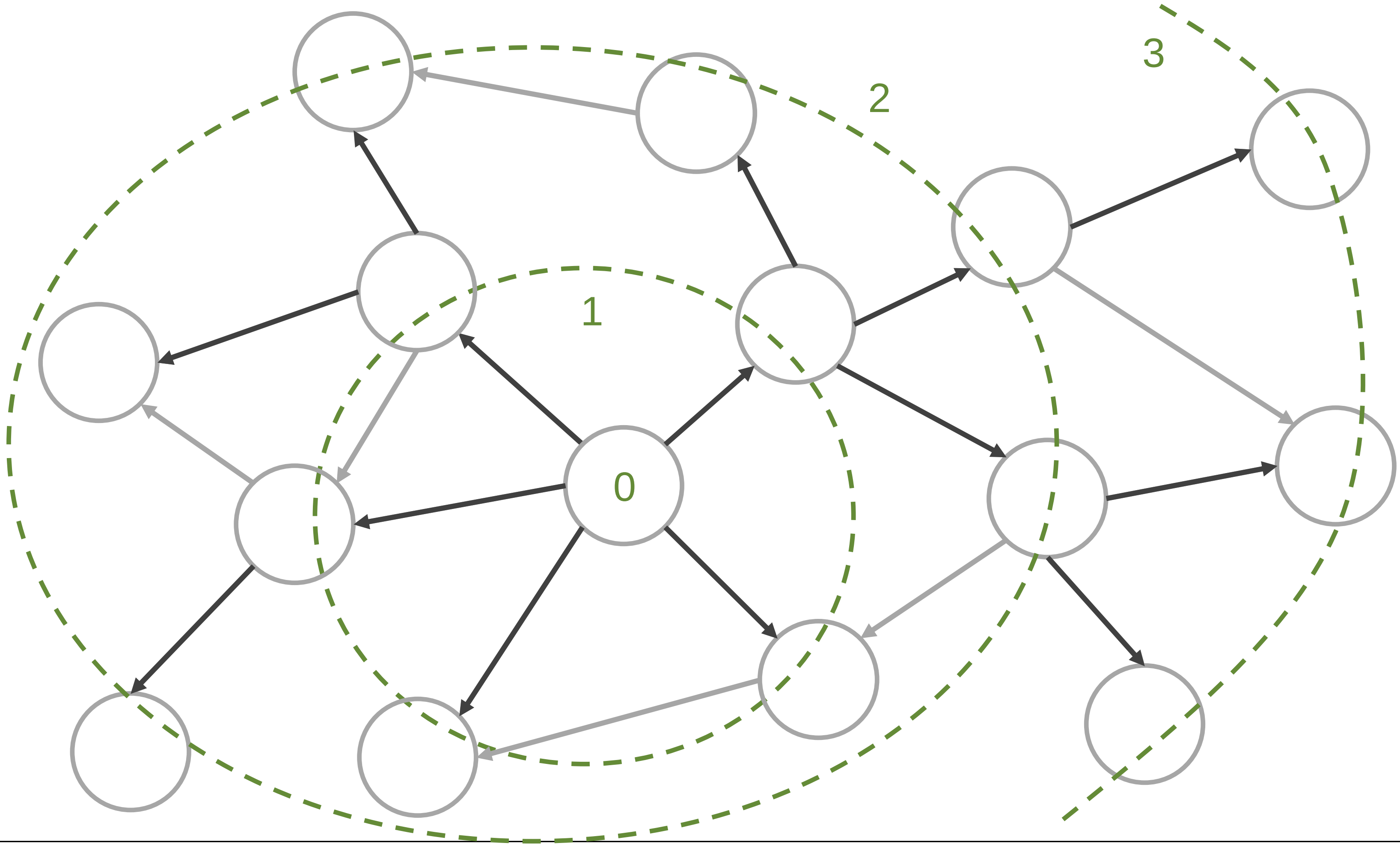
Überlegen Sie sich, wie man in einem gerichteten, gewichteten Graphen (in manchen Fällen) gleichgerichtete Mehrfachkanten „auflösen“ könnte.

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)

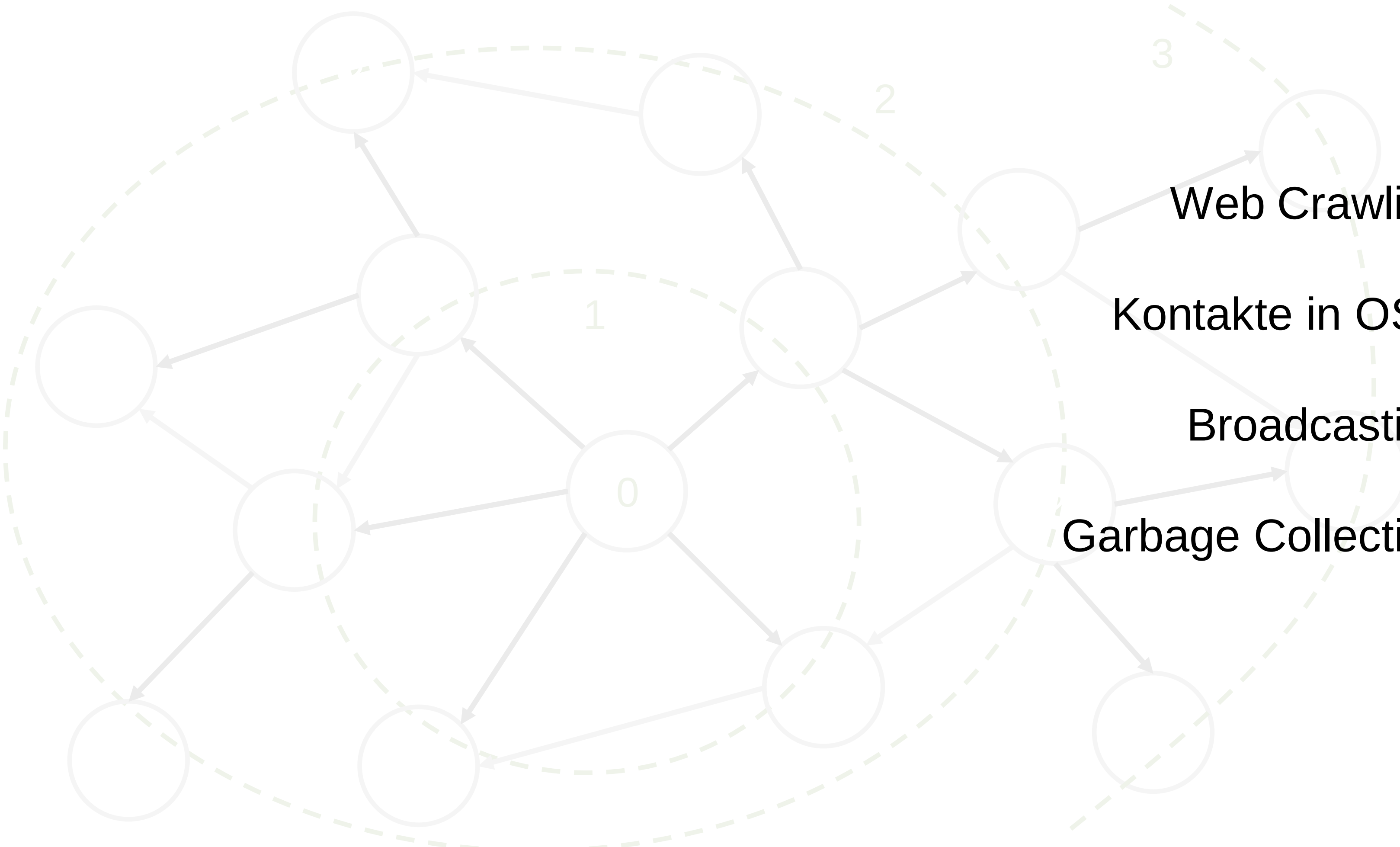
Idee

Besuche zuerst alle unmittelbaren Nachbarn, dann deren Nachbarn usw.



Anwendungen

Besuche zuerst alle unmittelbaren Nachbarn, dann deren Nachbarn usw.



Web Crawling

Kontakte in OSN

Broadcasting

Garbage Collection

…

BFS: Algorithmus

dist =Distanz von s
pred =Vorgängerknoten

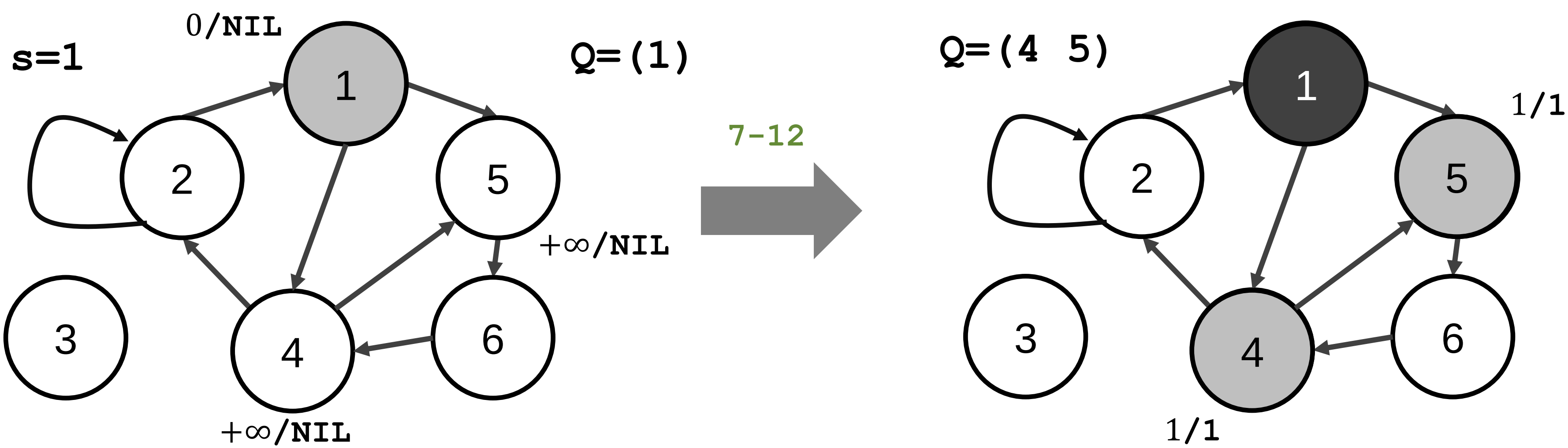
```
BFS(G,s) //G=(V,E), s=source node in V

FOREACH u in V-{s} DO
   u.color=WHITE; u.dist=+∞; u.pred=NIL;
s.color=GRAY; s.dist=0; s.pred=NIL;
newQueue(Q);
enqueue(Q,s);
WHILE !isEmpty(Q) DO
   u=dequeue(Q);
   FOREACH v in adj(G,u) DO
      IF v.color==WHITE THEN
         v.color=GRAY; v.dist=u.dist+1; v.pred=u;
         enqueue(Q,v);
   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)

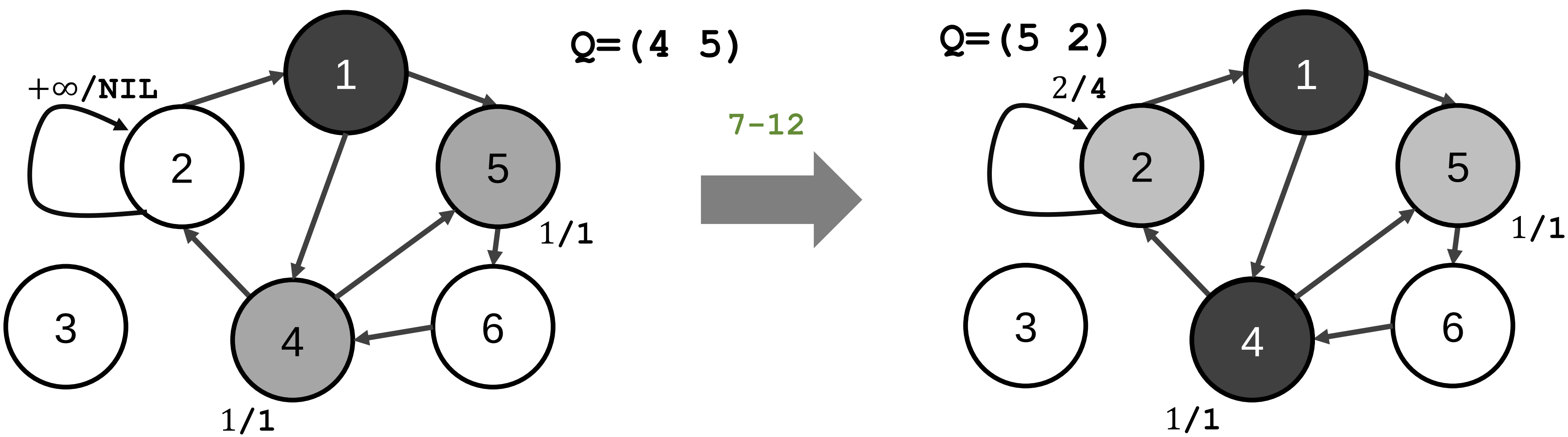
BFS: Algorithmus (Beispiel I)

```
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;
```



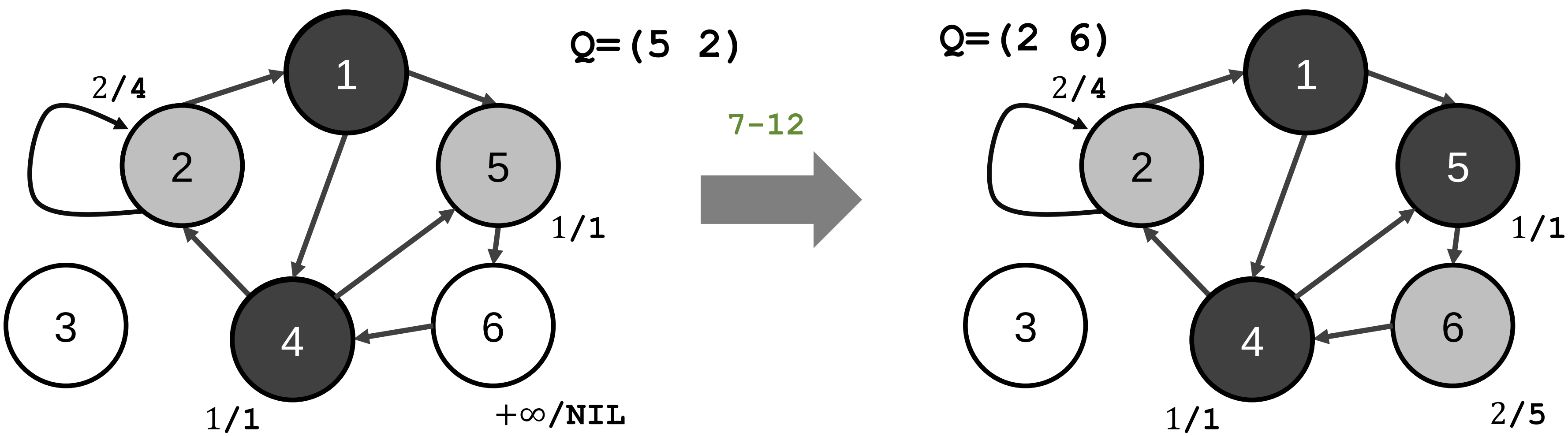
BFS: Algorithmus (Beispiel II)

```
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;
```



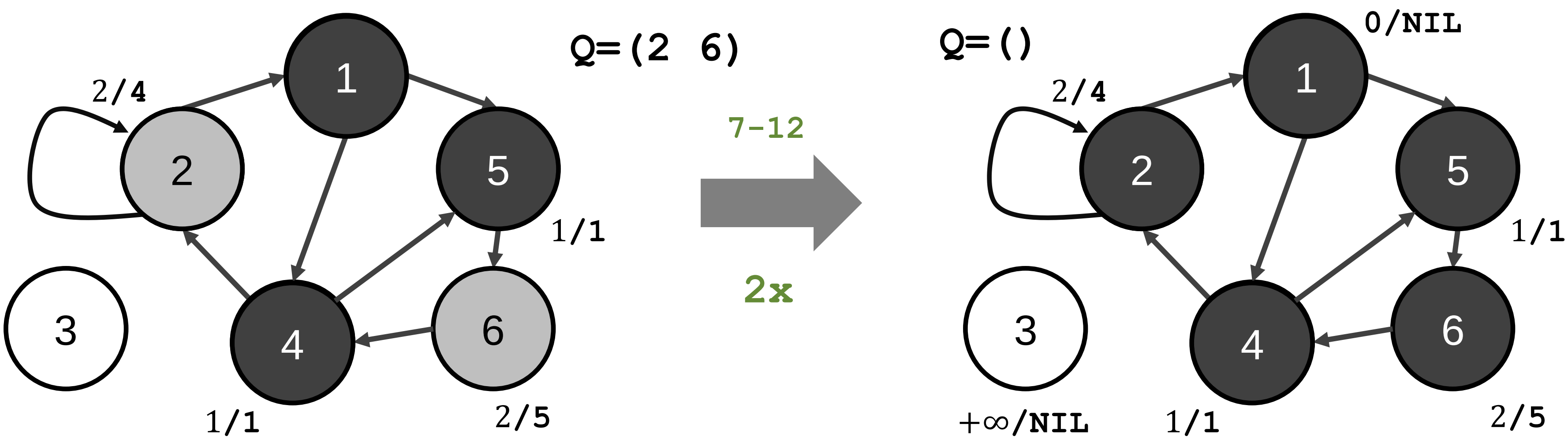
BFS: Algorithmus (Beispiel III)

```
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

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

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

FOREACH u in V-{s} DO
   u.color=WHITE; u.dist=+∞; u.pred=NIL;
s.color=GRAY; s.dist=0; s.pred=NIL;
newQueue(Q);
enqueue(Q,s);
WHILE !isEmpty(Q) DO
   u=dequeue(Q);
   FOREACH v in adj(G,u) DO
      IF v.color==WHITE THEN
         v.color=GRAY; v.dist=u.dist+1; v.pred=u;
         enqueue(Q,v);
   u.color=BLACK;
```

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

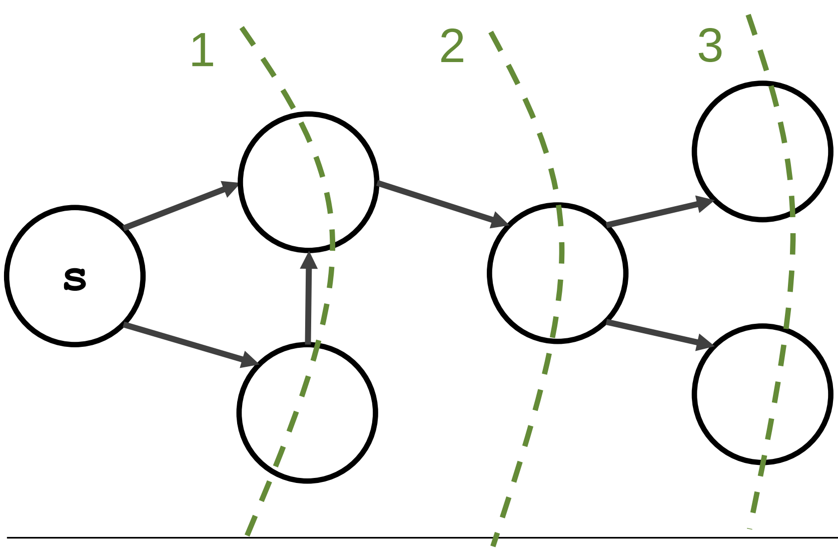
Jeder Knoten wird maximal einmal in **Q** aufgenommen (**s** oder **WHITE**)
(bzw. gar nicht, wenn nicht erreichbar)

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 $shortest(\mathbf{s},\mathbf{v}) =$**`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.

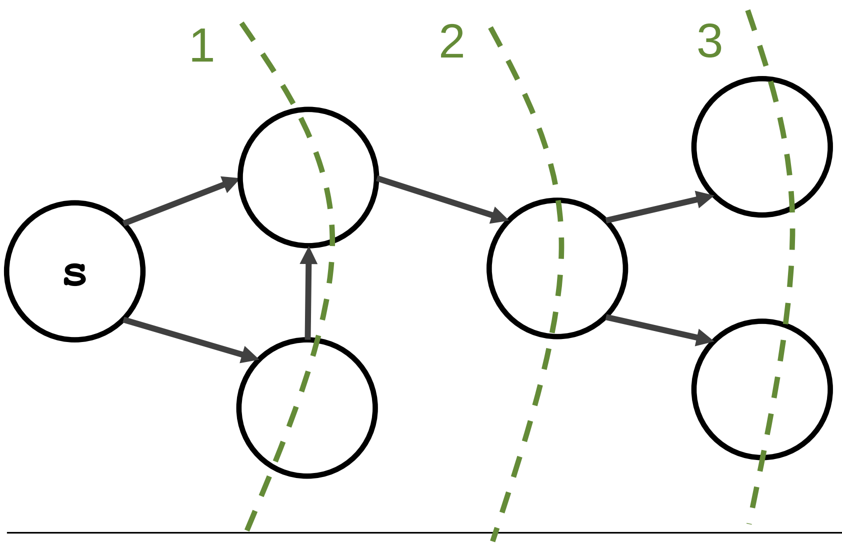


Intuition für Korrektheit **`dist`**:

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 $shortest($**`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.



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 $\mathtt{s} \in V$. Dann gilt nach Terminierung von `BFS(G,s)` für jeden von `s` aus erreichbaren Knoten `v`, dass $shortest(\mathtt{s,v}) = \mathtt{v.dist}$.
Für $\mathtt{v} \neq \mathtt{s}$ 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 $(\mathtt{u,v}) \in E$ besucht wurde, der Pfad von `s` nach `u` und dann zu `v` ein Pfad der Länge

$$1 + shortest(\mathtt{s,u}) \underset{\text{gemäß Teil 1}}{=} 1 + \mathtt{u.dist} \underset{\text{gemäß Algorithmus}}{=} \mathtt{v.dist} \underset{\text{gemäß Teil 1}}{=} shortest(\mathtt{s,v})$$

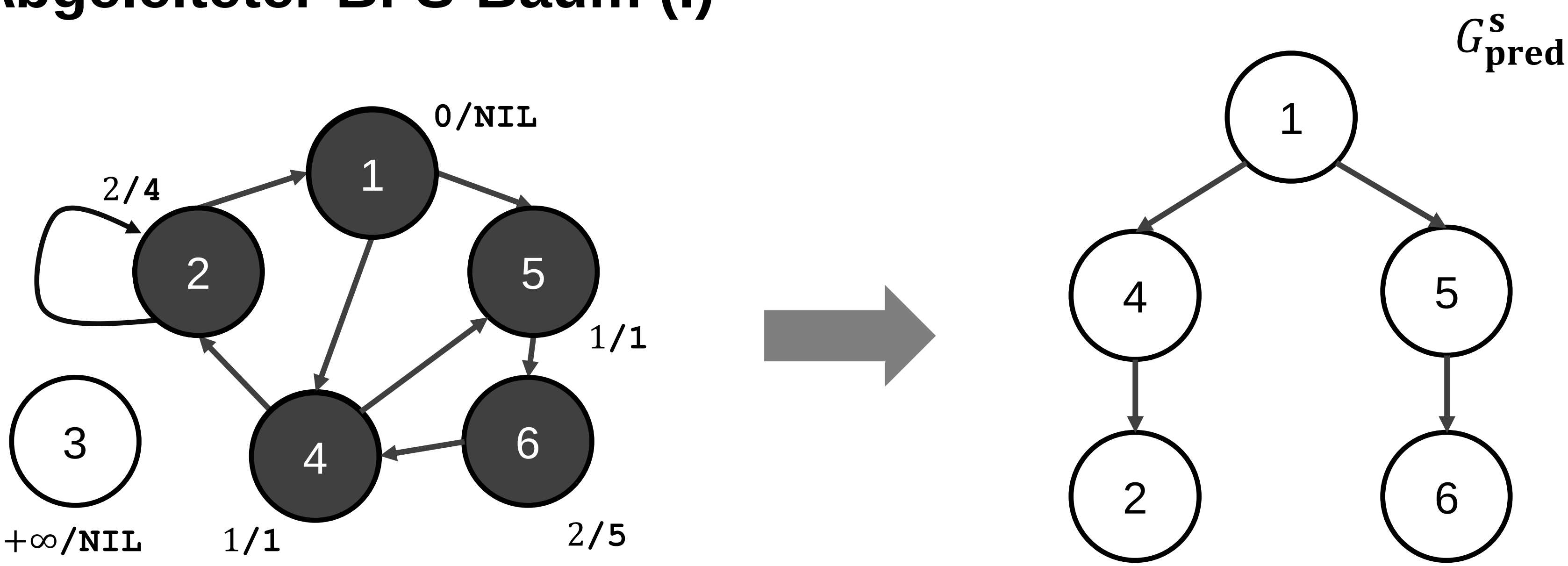
Kürzeste Pfade ausgeben

Laufzeit* = $O(|V|)$

*ohne BFS

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

IF v==s THEN
   PRINT s
ELSE
   IF v.pred==NIL THEN
      PRINT ‘no path from s to v‘
   ELSE
      PRINT-PATH(G,s,v.pred);
      PRINT v;
```

Abgeleiteter BFS-Baum (I)

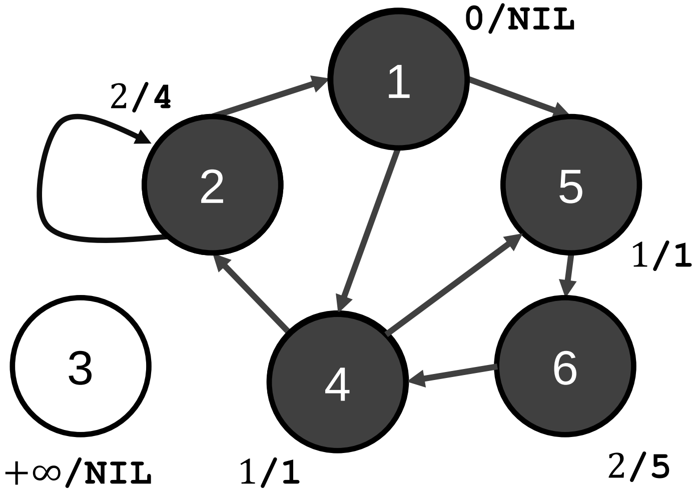


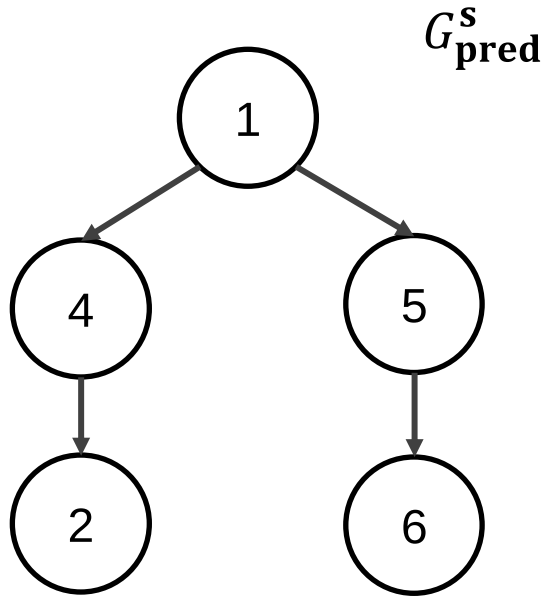
Definiere Subgraph $G^{\mathbf{s}}_{\mathbf{pred}} = (V^{\mathbf{s}}_{\mathbf{pred}}, E^{\mathbf{s}}_{\mathbf{pred}})$ von G durch:

$$V^{\mathbf{s}}_{\mathbf{pred}} = \{\, \mathtt{v} \in V \mid \mathtt{v.pred} \neq \mathtt{NIL} \} \cup \{\mathtt{s}\}$$
$$E^{\mathbf{s}}_{\mathbf{pred}} = \{\, (\mathtt{v.pred}, \mathtt{v}) \mid \mathtt{v} \in V^{\mathbf{s}}_{\mathbf{pred}} - \{\mathtt{s}\} \,\}$$

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

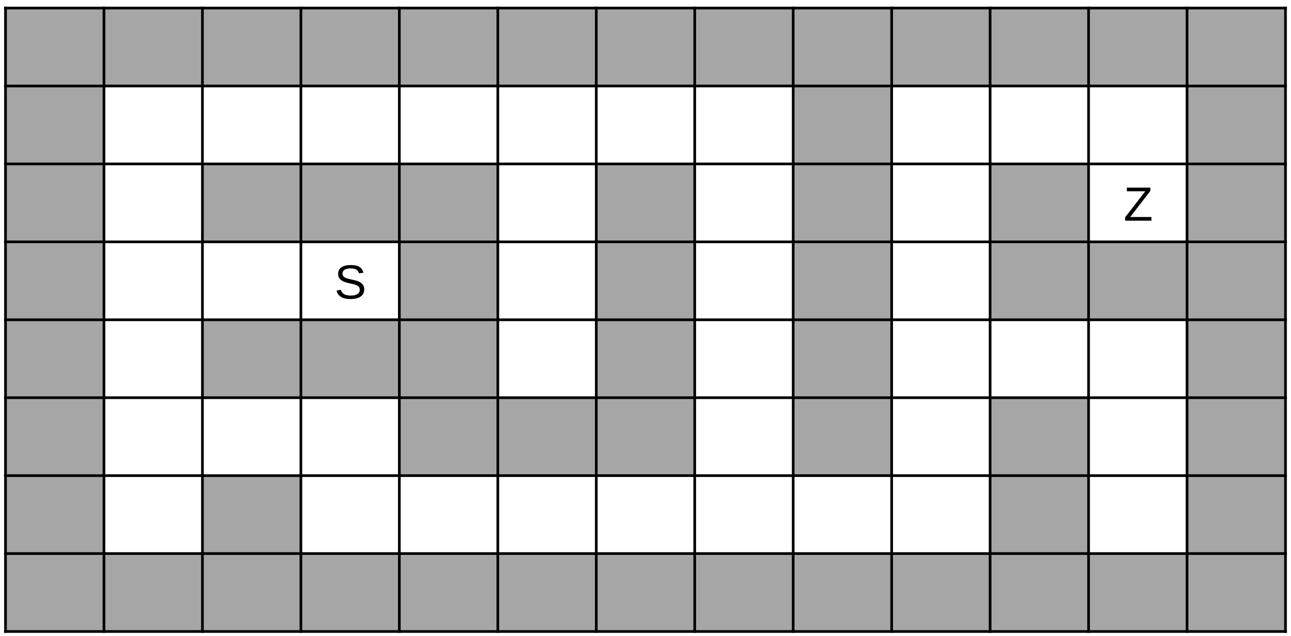
Abgeleiteter BFS-Baum (II)





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



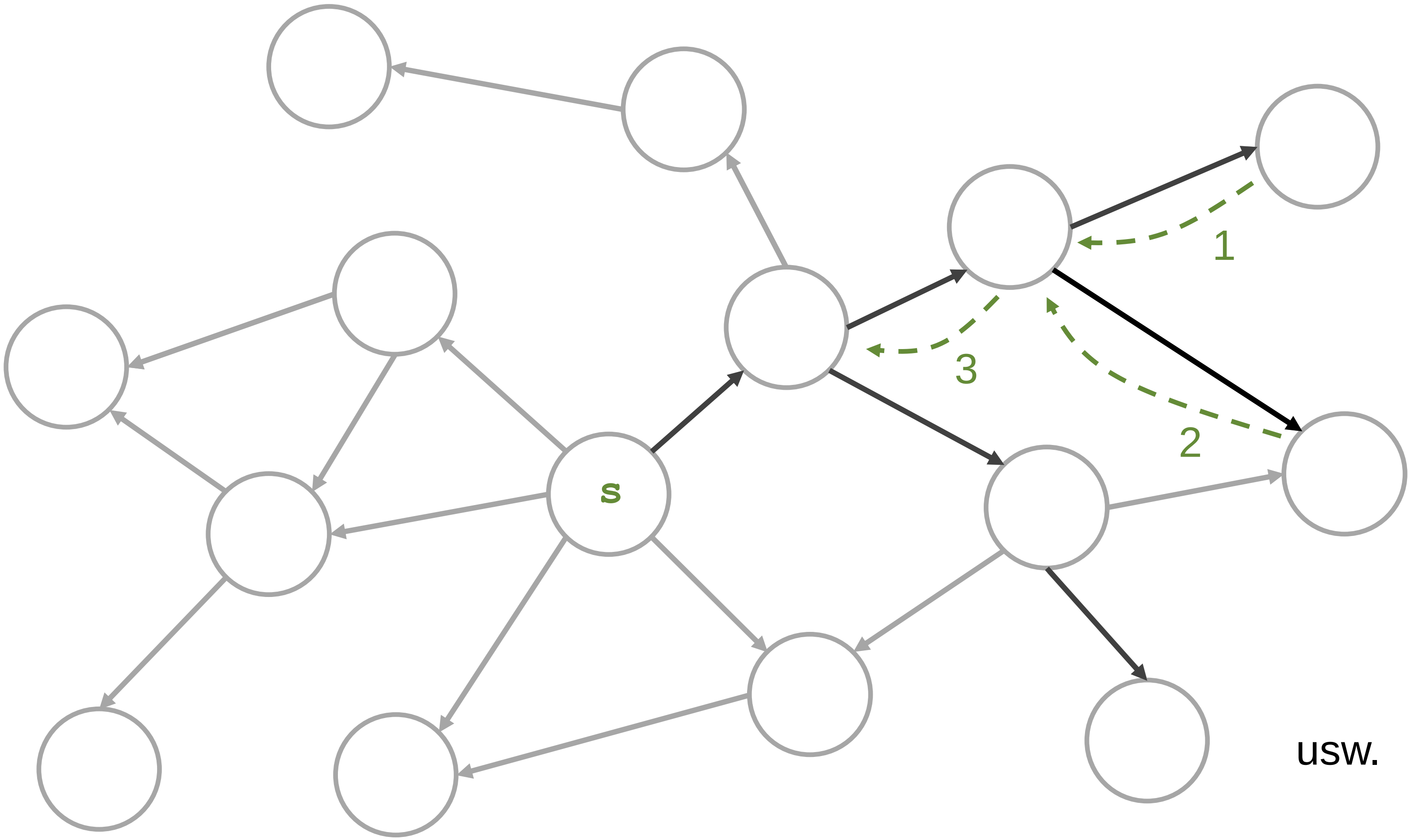
Hinweis: Sie sind selbst nicht im Labyrinth, sondern „schauen von oben darauf“

Depth-First Search (DFS)

TECHNISCHE UNIVERSITÄT DARMSTADT
Cryptoplexity
Cryptography & Complexity Theory
Technische Universität Darmstadt
www.cryptoplexity.de

Idee

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



usw.

DFS: Algorithmus

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

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

FOREACH u in V DO
    u.color=WHITE;
    u.pred=NIL;
time=0;
FOREACH u in V DO
    IF u.color==WHITE THEN
        DFS-VISIT(G,u)
```

`time` globale Variable

```
DFS-VISIT(G,u)

time=time+1;
u.disc=time;
u.color=GRAY;
FOREACH v in adj(G,u) DO
    IF v.color==WHITE THEN
        v.pred=u;
        DFS-VISIT(G,v);
u.color=BLACK;
time=time+1;
u.finish=time;
```

`disc` = discovery time
`finish`=finish time

DFS: Beispiel (I)

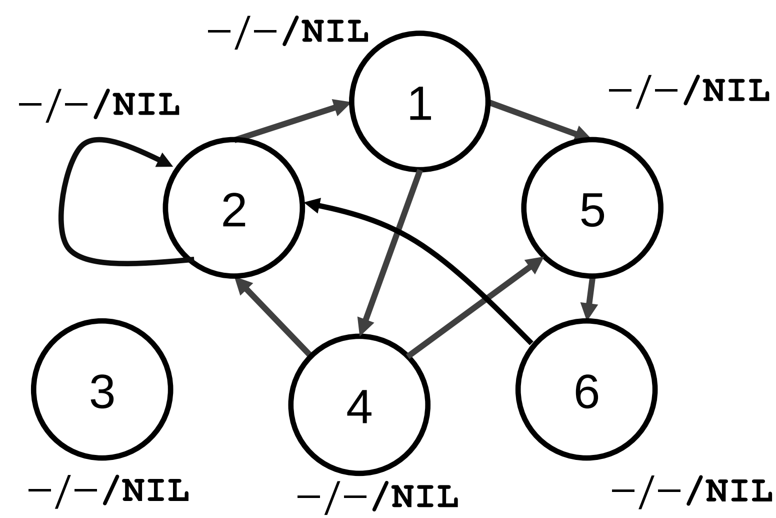
Aufrufs-Stack: DFS-VISIT(G,1)

Ordnung auf Knotenlisten gemäß Knotennummern

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

FOREACH u in V DO
    u.color=WHITE;
    u.pred=NIL;
time=0;
FOREACH u in V DO
    IF u.color==WHITE THEN
        DFS-VISIT(G,u)
```

time=0

```
DFS-VISIT(G,u)

time=time+1;
u.disc=time;
u.color=GRAY;
FOREACH v in adj(G,u) DO
    IF v.color==WHITE THEN
        v.pred=u;
        DFS-VISIT(G,v);
u.color=BLACK;
time=time+1;
u.finish=time;
```



DFS: Beispiel (II)

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

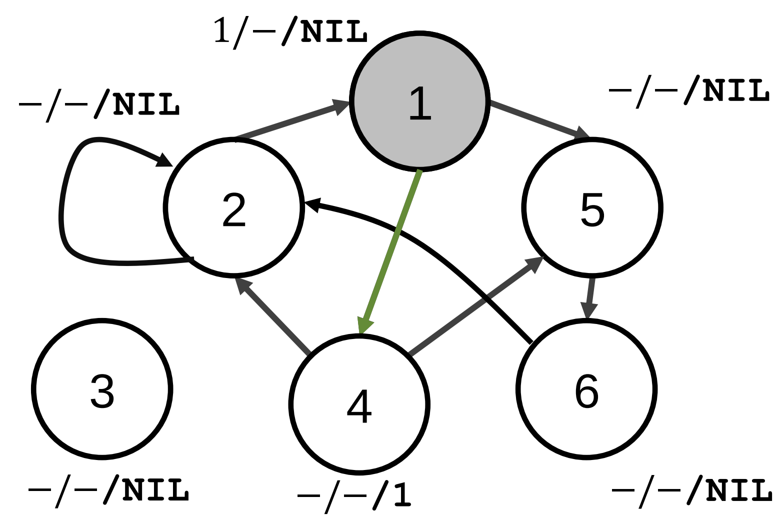
↓↑ DFS-VISIT(G,4)

Ordnung auf Knotenlisten gemäß Knotennummern

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

FOREACH u in V DO
    u.color=WHITE;
    u.pred=NIL;
time=0;
FOREACH u in V DO
    IF u.color==WHITE THEN
        DFS-VISIT(G,u)
```

time=1

```
DFS-VISIT(G,u)

time=time+1;
u.disc=time;
u.color=GRAY;
FOREACH v in adj(G,u) DO
    IF v.color==WHITE THEN
        v.pred=u;
        DFS-VISIT(G,v);
u.color=BLACK;
time=time+1;
u.finish=time;
```



DFS: Beispiel (II)

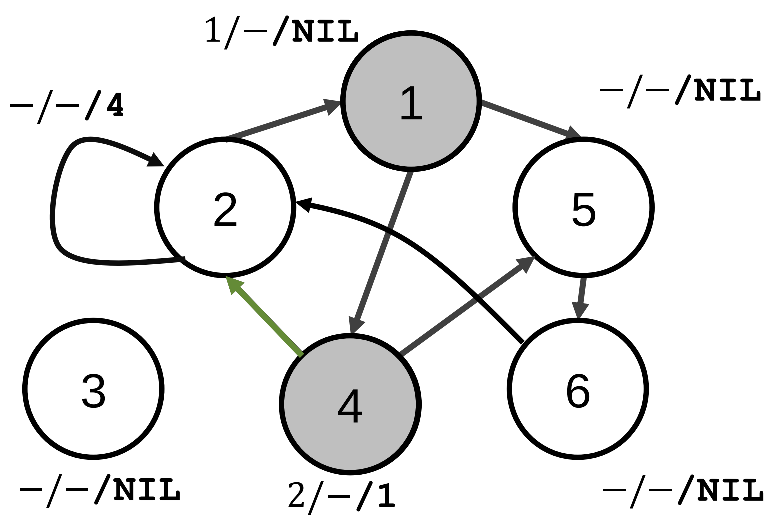
Aufrufs-Stack: DFS-VISIT(G,1)
↓↑ DFS-VISIT(G,4)
DFS-VISIT(G,2)

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)
```

time=2

```
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: Beispiel (III)

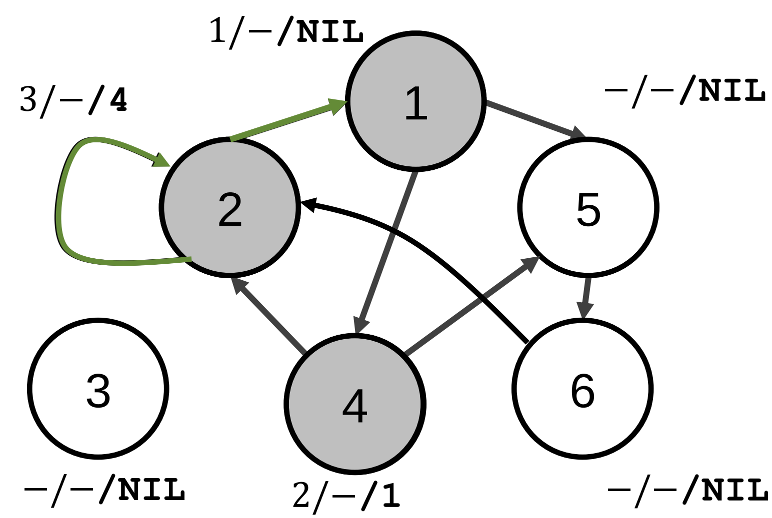
Aufrufs-Stack: **DFS-VISIT(G,1)**
↓↑ **DFS-VISIT(G,4)**
DFS-VISIT(G,2)

Ordnung auf Knotenlisten gemäß Knotennummern

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

FOREACH u in V DO
    u.color=WHITE;
    u.pred=NIL;
time=0;
FOREACH u in V DO
    IF u.color==WHITE THEN
        DFS-VISIT(G,u)
```

time=3

```
DFS-VISIT(G,u)

time=time+1;
u.disc=time;
u.color=GRAY;
FOREACH v in adj(G,u) DO
    IF v.color==WHITE THEN
        v.pred=u;
        DFS-VISIT(G,v);
u.color=BLACK;
time=time+1;
u.finish=time;
```



DFS: Beispiel (IV)

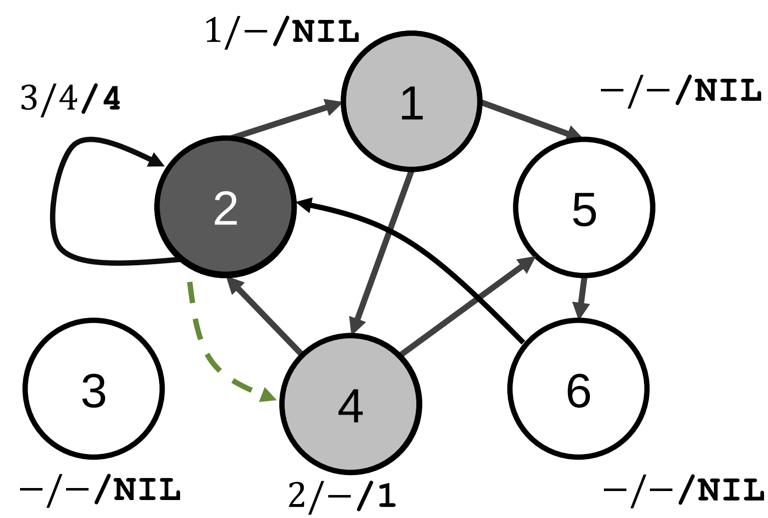
Ordnung auf Knotenlisten gemäß Knotennummern

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

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

FOREACH u in V DO
    u.color=WHITE;
    u.pred=NIL;
time=0;
FOREACH u in V DO
    IF u.color==WHITE THEN
        DFS-VISIT(G,u)
```

time=4

```
DFS-VISIT(G,u)

time=time+1;
u.disc=time;
u.color=GRAY;
FOREACH v in adj(G,u) DO
    IF v.color==WHITE THEN
        v.pred=u;
        DFS-VISIT(G,v);
u.color=BLACK;
time=time+1;
u.finish=time;
```



DFS: Beispiel (V)

Ordnung auf Knotenlisten gemäß Knotennummern

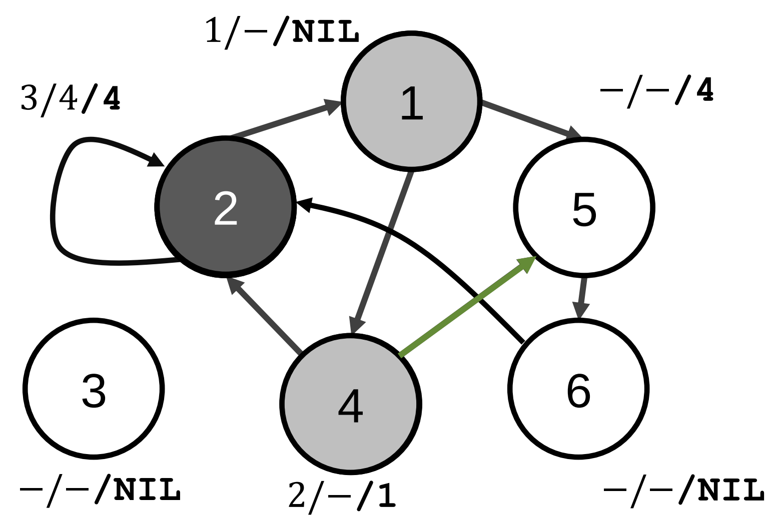
Aufrufs-Stack: ↓↑

```
DFS-VISIT(G,1)
DFS-VISIT(G,4)
DFS-VISIT(G,5)
```

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

FOREACH u in V DO
    u.color=WHITE;
    u.pred=NIL;
time=0;
FOREACH u in V DO
    IF u.color==WHITE THEN
        DFS-VISIT(G,u)
```

time=4

```
DFS-VISIT(G,u)

time=time+1;
u.disc=time;
u.color=GRAY;
FOREACH v in adj(G,u) DO
    IF v.color==WHITE THEN
        v.pred=u;
        DFS-VISIT(G,v);
u.color=BLACK;
time=time+1;
u.finish=time;
```



DFS: Beispiel (VI)

Ordnung auf Knotenlisten gemäß Knotennummern

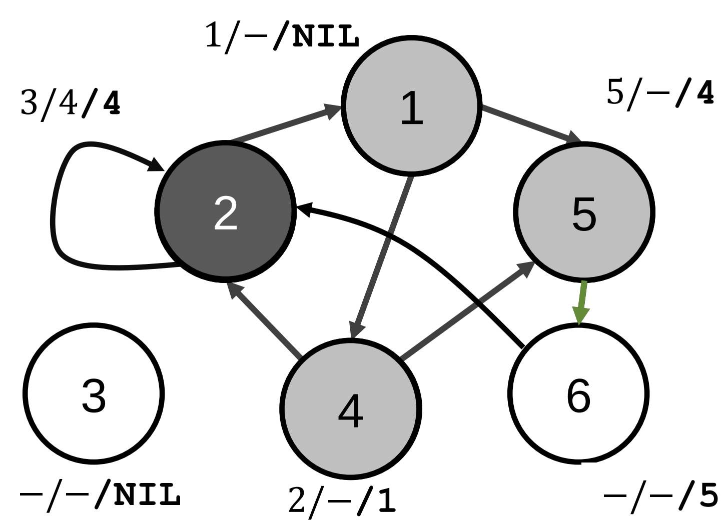
Aufrufs-Stack: ↓↑

```
DFS-VISIT(G,1)
DFS-VISIT(G,4)
DFS-VISIT(G,5)
DFS-VISIT(G,6)
```

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

FOREACH u in V DO
    u.color=WHITE;
    u.pred=NIL;
time=0;
FOREACH u in V DO
    IF u.color==WHITE THEN
        DFS-VISIT(G,u)
```

time=5

```
DFS-VISIT(G,u)

time=time+1;
u.disc=time;
u.color=GRAY;
FOREACH v in adj(G,u) DO
    IF v.color==WHITE THEN
        v.pred=u;
        DFS-VISIT(G,v);
u.color=BLACK;
time=time+1;
u.finish=time;
```



DFS: Beispiel (VII)

Aufrufs-Stack: ↓↑

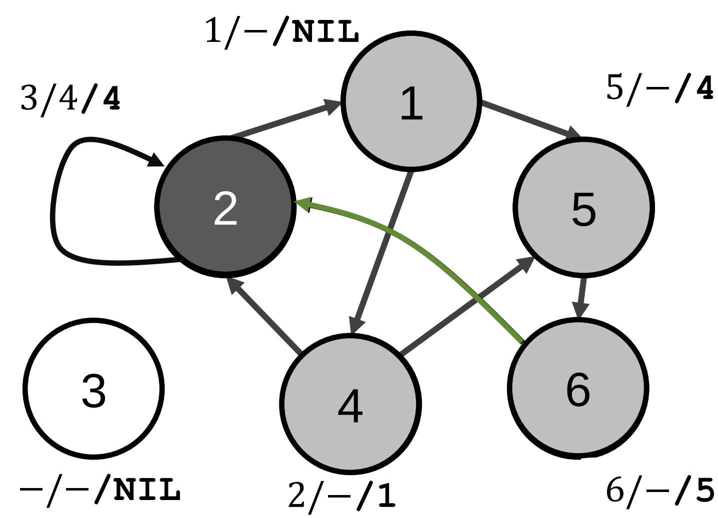
DFS-VISIT(G,1)
DFS-VISIT(G,4)
DFS-VISIT(G,5)
DFS-VISIT(G,6)

Ordnung auf Knotenlisten gemäß Knotennummern

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

FOREACH u in V DO
    u.color=WHITE;
    u.pred=NIL;
time=0;
FOREACH u in V DO
    IF u.color==WHITE THEN
        DFS-VISIT(G,u)
```

time=6

```
DFS-VISIT(G,u)

time=time+1;
u.disc=time;
u.color=GRAY;
FOREACH v in adj(G,u) DO
    IF v.color==WHITE THEN
        v.pred=u;
        DFS-VISIT(G,v);
u.color=BLACK;
time=time+1;
u.finish=time;
```



DFS: Beispiel (VIII)

Aufrufs-Stack: ↓↑

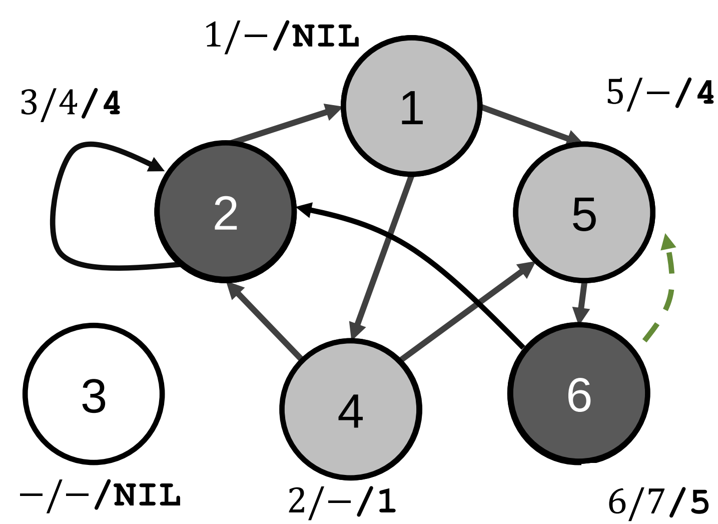
```
DFS-VISIT(G,1)
DFS-VISIT(G,4)
DFS-VISIT(G,5)
```

Ordnung auf Knotenlisten gemäß Knotennummern

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

FOREACH u in V DO
    u.color=WHITE;
    u.pred=NIL;
time=0;
FOREACH u in V DO
    IF u.color==WHITE THEN
        DFS-VISIT(G,u)
```

time=7

```
DFS-VISIT(G,u)

time=time+1;
u.disc=time;
u.color=GRAY;
FOREACH v in adj(G,u) DO
    IF v.color==WHITE THEN
        v.pred=u;
        DFS-VISIT(G,v);
u.color=BLACK;
time=time+1;
u.finish=time;
```



DFS: Beispiel (IX)

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

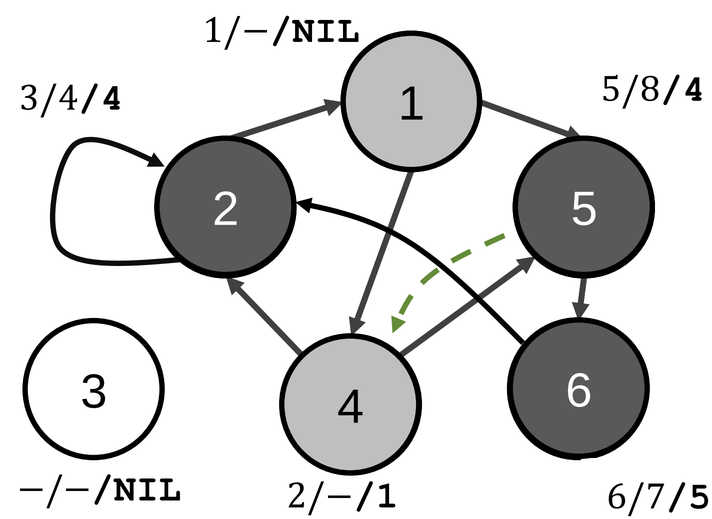
↓↑ DFS-VISIT(G,4)

Ordnung auf Knotenlisten gemäß Knotennummern

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

FOREACH u in V DO
    u.color=WHITE;
    u.pred=NIL;
time=0;
FOREACH u in V DO
    IF u.color==WHITE THEN
        DFS-VISIT(G,u)
```

time=8

```
DFS-VISIT(G,u)

time=time+1;
u.disc=time;
u.color=GRAY;
FOREACH v in adj(G,u) DO
    IF v.color==WHITE THEN
        v.pred=u;
        DFS-VISIT(G,v);
u.color=BLACK;
time=time+1;
u.finish=time;
```



DFS: Beispiel (X)

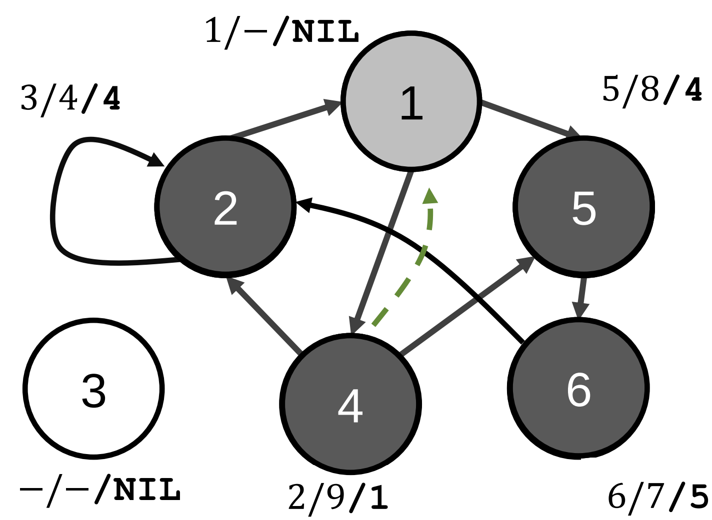
Ordnung auf Knotenlisten gemäß Knotennummern

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

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

FOREACH u in V DO
   u.color=WHITE;
   u.pred=NIL;
time=0;
FOREACH u in V DO
   IF u.color==WHITE THEN
      DFS-VISIT(G,u)
```

time=9

```
DFS-VISIT(G,u)

time=time+1;
u.disc=time;
u.color=GRAY;
FOREACH v in adj(G,u) DO
   IF v.color==WHITE THEN
      v.pred=u;
      DFS-VISIT(G,v);
u.color=BLACK;
time=time+1;
u.finish=time;
```



DFS: Beispiel (X)

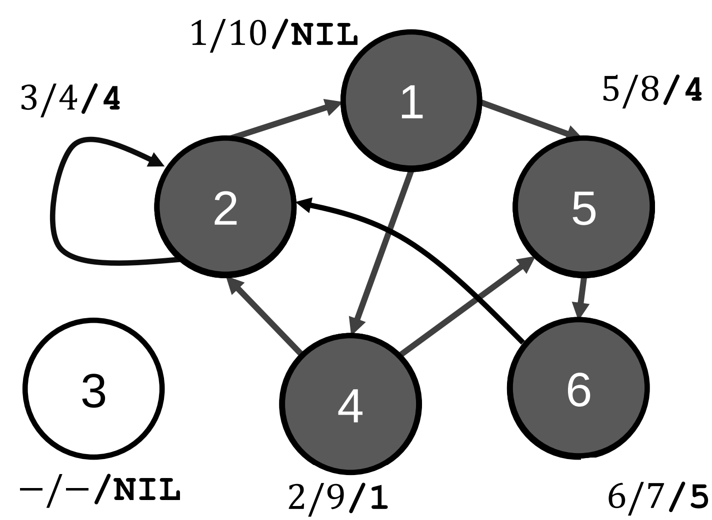
Ordnung auf Knotenlisten gemäß Knotennummern

Aufrufs-Stack:

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

FOREACH u in V DO
    u.color=WHITE;
    u.pred=NIL;
time=0;
FOREACH u in V DO
    IF u.color==WHITE THEN
        DFS-VISIT(G,u)
```

time=10

```
DFS-VISIT(G,u)

time=time+1;
u.disc=time;
u.color=GRAY;
FOREACH v in adj(G,u) DO
    IF v.color==WHITE THEN
        v.pred=u;
        DFS-VISIT(G,v);
u.color=BLACK;
time=time+1;
u.finish=time;
```



DFS: Beispiel (XI)

Ordnung auf Knotenlisten gemäß Knotennummern

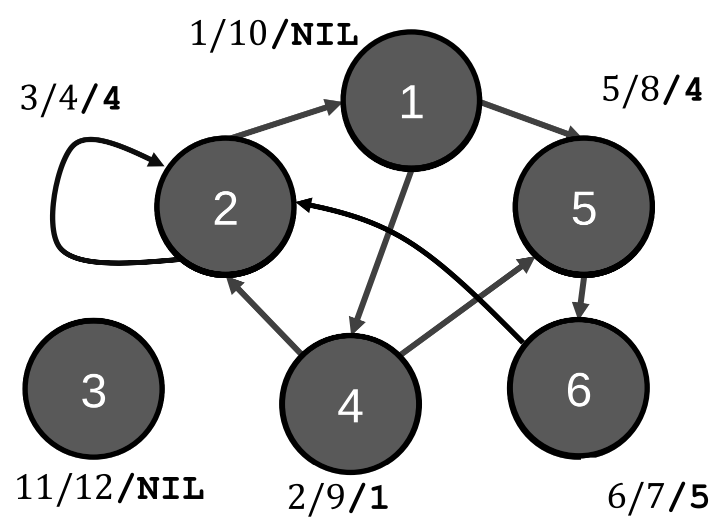
Aufrufs-Stack: **DFS-VISIT(G,3)**

↓↑

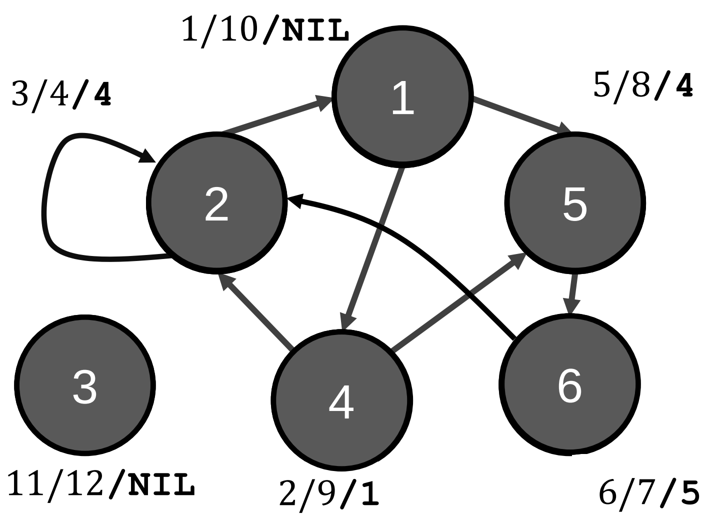
```
DFS-VISIT(G,u)

time=time+1;
u.disc=time;
u.color=GRAY;
FOREACH v in adj(G,u) DO
    IF v.color==WHITE THEN
        v.pred=u;
        DFS-VISIT(G,v);
u.color=BLACK;
time=time+1;
u.finish=time;
```

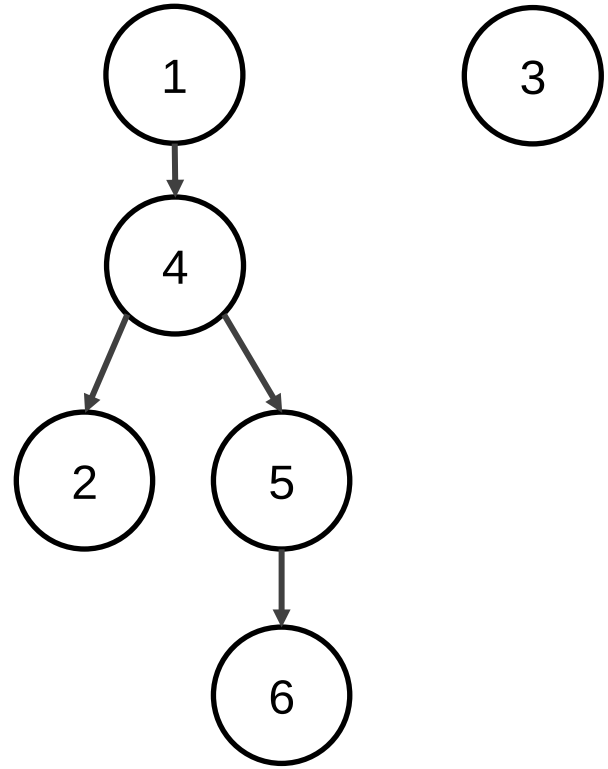
time=12

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

FOREACH u in V DO
    u.color=WHITE;
    u.pred=NIL;
time=0;
FOREACH u in V DO
    IF u.color==WHITE THEN
        DFS-VISIT(G,u)
```



DFS-Wald = Menge von DFS-Bäumen





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

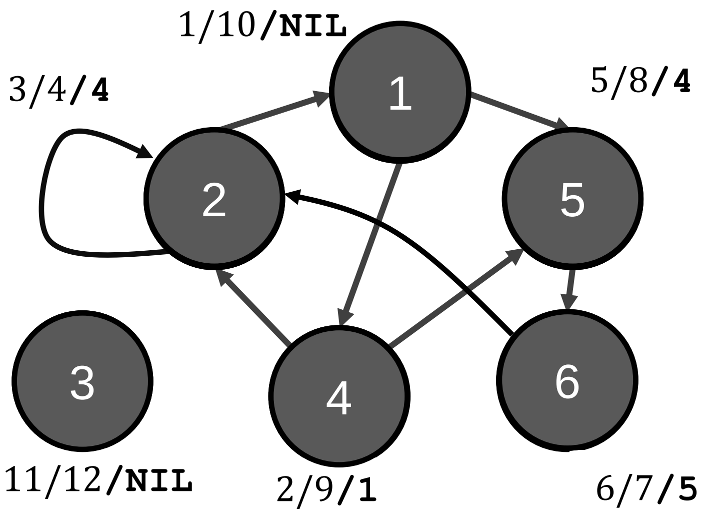
$$E_{\mathbf{pred}} = \{\, (\mathtt{v.pred}, \mathtt{v}) \mid \mathtt{v} \in V, \mathtt{v.pred} \neq \mathtt{NIL} \,\}$$

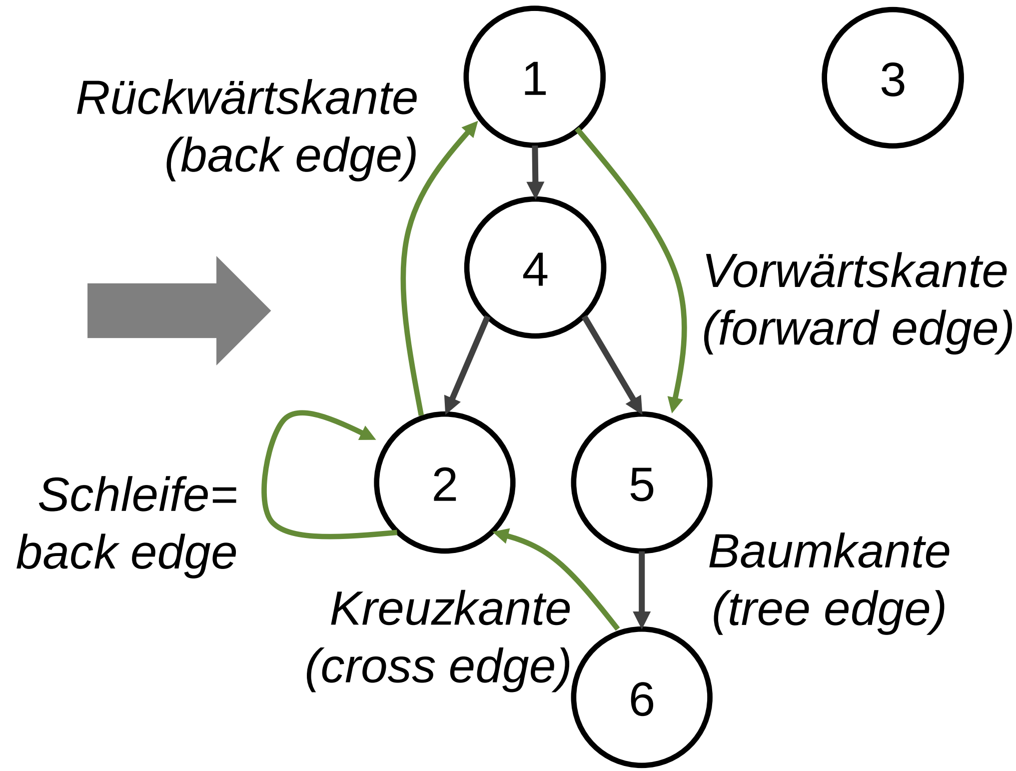
(und $(\mathtt{v}, \mathtt{v.pred})$ für ungerichtete Graphen)

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 $G_{\mathbf{pred}}$ ein





Charakterisierung der Kanten in G mittels DFS

Baumkanten: alle Kanten in $G_{\mathbf{pred}}$

Vorwärtskanten: alle Kanten in G zu Nachkommen in $G_{\mathbf{pred}}$, die nicht Baumkante

Rückwärtskanten: alle Kanten in G zu Vorfahren in $G_{\mathbf{pred}}$, die nicht Baumkante (inkl. Schleifen)

Kreuzkanten: alle anderen Kanten in G

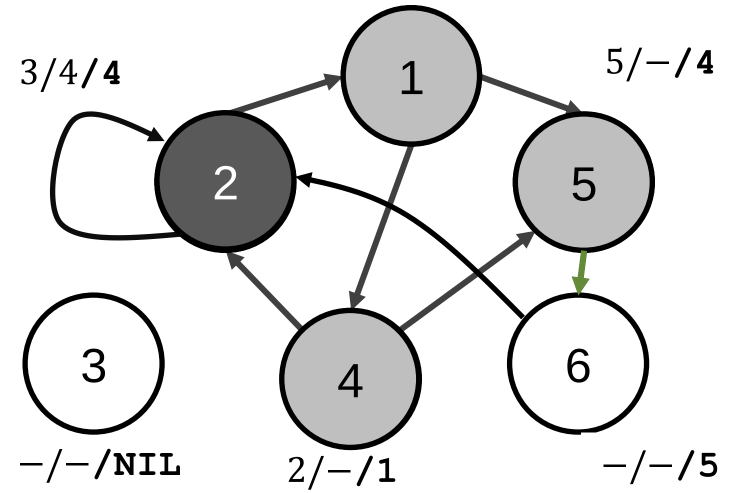
Kantenart erkennen (I)

Sei $(\mathbf{u}, \mathbf{v})$ gerade betrachte Kante im DFS-Algorithmus. Dann ist $(\mathbf{u}, \mathbf{v})$…

…eine Baumkante, wenn **`v.color==WHITE`**

Beispiel: 5→6

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



Kantenart erkennen (II)

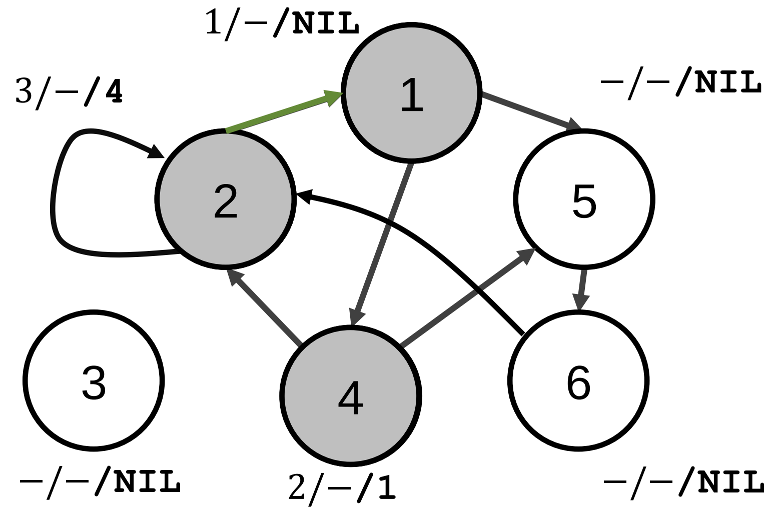
Sei $(\mathbf{u}, \mathbf{v})$ gerade betrachte Kante im DFS-Algorithmus. Dann ist $(\mathbf{u}, \mathbf{v})$…

…eine Baumkante, wenn **`v.color==WHITE`**

…eine Rückwärtskante, wenn **`v.color==GRAY`**

Beispiel: 2→1

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



Kantenart erkennen (III)

Sei $(\mathbf{u}, \mathbf{v})$ gerade betrachte Kante im DFS-Algorithmus. Dann ist $(\mathbf{u}, \mathbf{v})$…

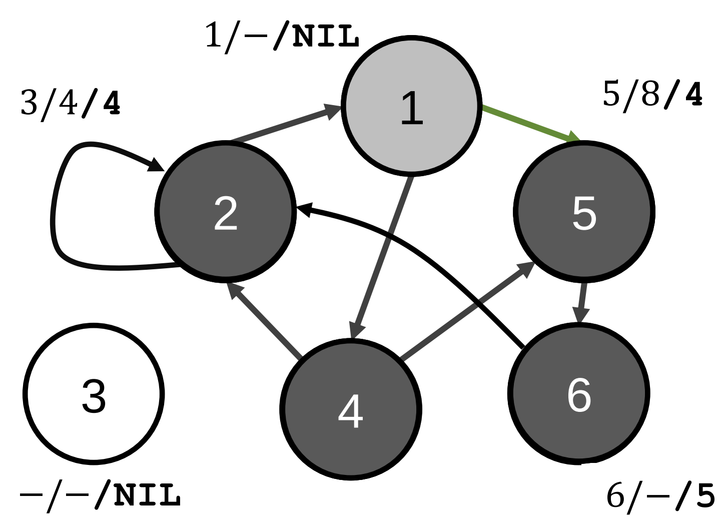
…eine Baumkante, wenn **`v.color==WHITE`**

…eine Rückwärtskante, wenn **`v.color==GRAY`**

…eine Vorwärtskante, wenn **`v.color==BLACK`** und **`u.disc<v.disc`**

Beispiel: 1→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 Nachkommen von **`u`** besucht,
da **`u`** vorher zu einem weißen Knoten überging



Kantenart erkennen (IV)

Sei $(\mathbf{u}, \mathbf{v})$ gerade betrachte Kante im DFS-Algorithmus. Dann ist $(\mathbf{u}, \mathbf{v})$…

…eine Baumkante, wenn `v.color==WHITE`

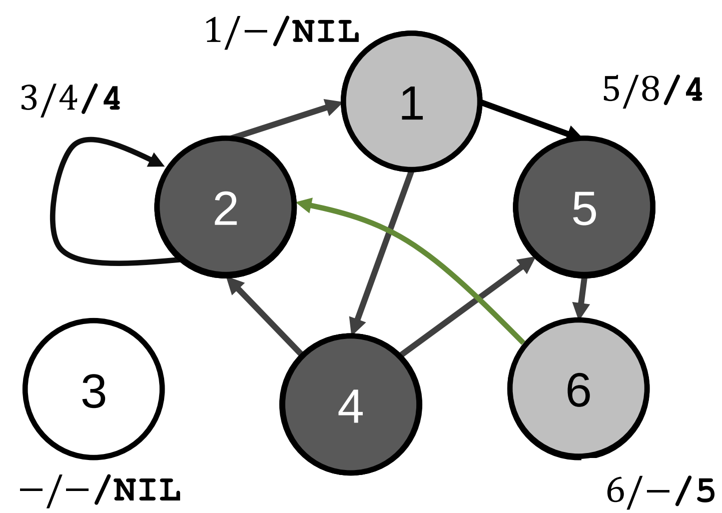
…eine Rückwärtskante, wenn `v.color==GRAY`

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

…eine Kreuzkante, wenn `v.color==BLACK` und `v.disc<u.disc`

Beispiel: 6→2

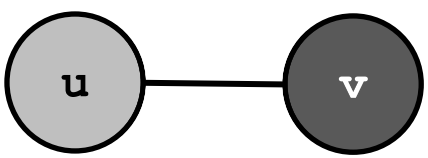
bilden per Definition
die restlichen Kanten



Kantenarten in ungerichteten Graphen (I)

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

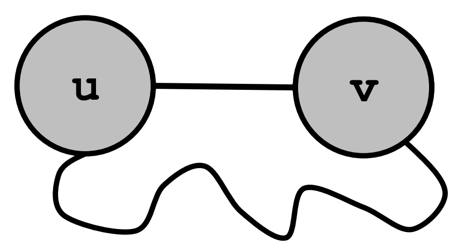
Zur Erinnerung: Vorwärts- und Kreuzkanten haben **`v.color==BLACK`**



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

1.Fall: **`u.disc<v.disc`**

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

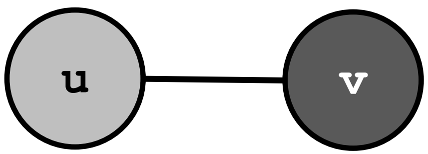


Dann wäre aber Kante {**`v,u`**} bereits bei **`v`** Rückwärtskante geworden.

Kantenarten in ungerichteten Graphen (II)

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

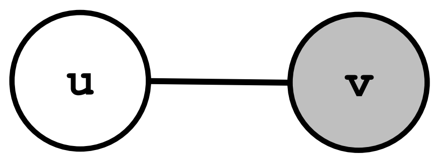
Zur Erinnerung: Vorwärts- und Kreuzkanten haben `v.color==BLACK`



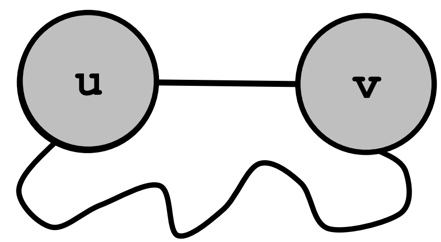
Sei `u` gerade aktiv (grau).
Betrachtete neue Kante `{u,v}` kann nur Vorwärts- oder Kreuzkante werden, wenn `v` schon abgeschlossen (schwarz).

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.

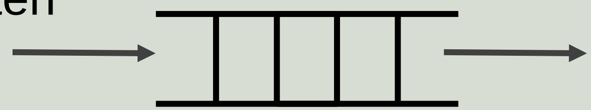
Datenstrukturen und Graphensuche

BFS

Queue-basiert

merke zu besuchende Knoten

verarbeite nächsten Knoten



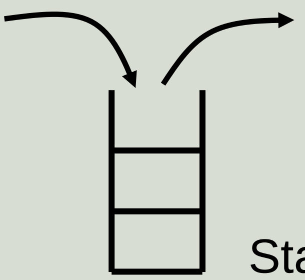
Queue

DFS

Stack-basiert

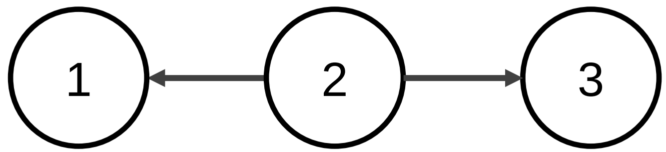
merke zu besuchende Knoten

verarbeite nächsten Knoten



Stack

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

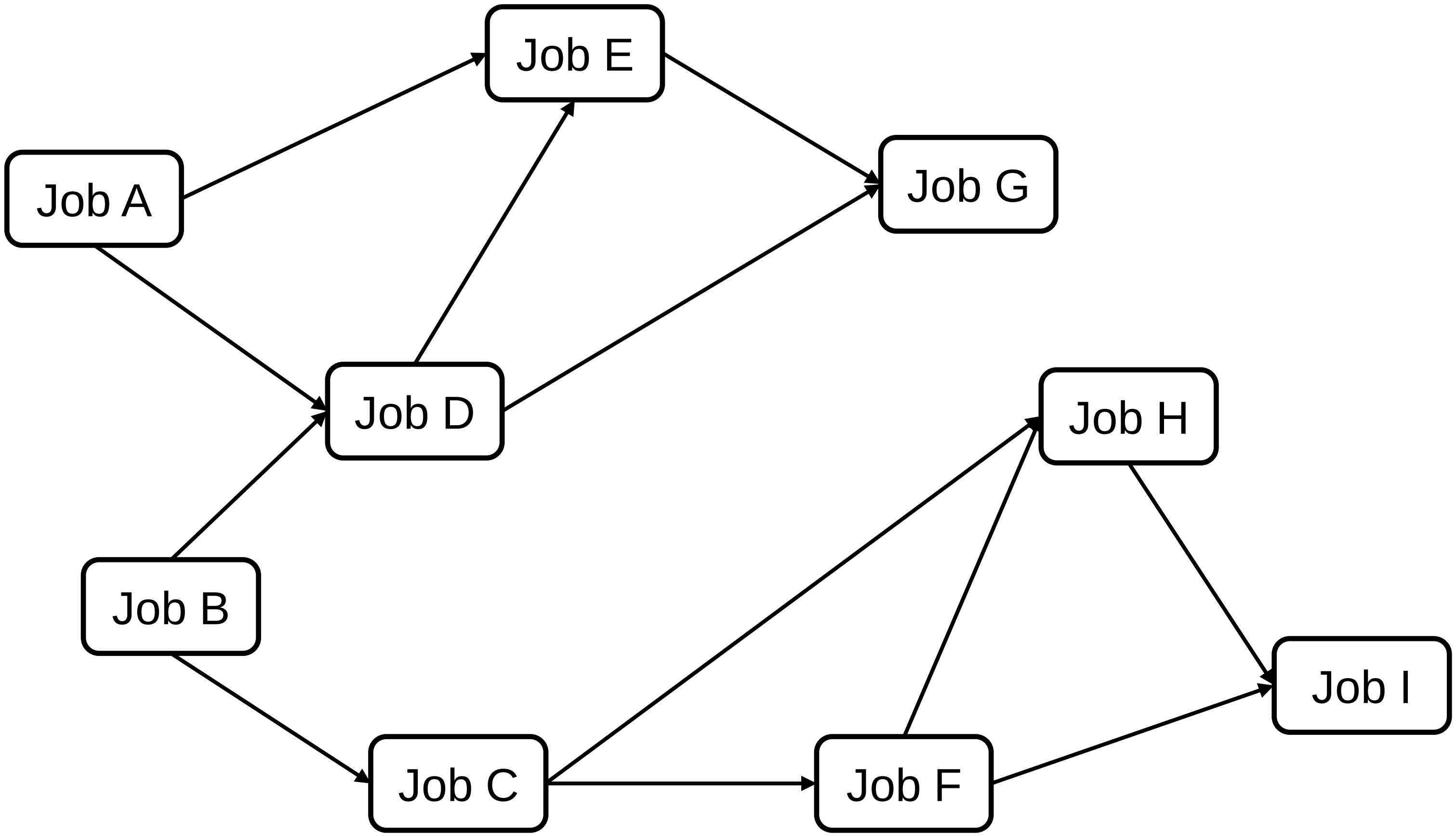


Geben Sie eine andere Ordnung für den Durchlauf des DFS an,
bei der keine Kreuzkanten entstehen.

Anwendungen DFS:
Topologisches Sortieren und
Starke Zusammenhangskomponenten

Job Scheduling

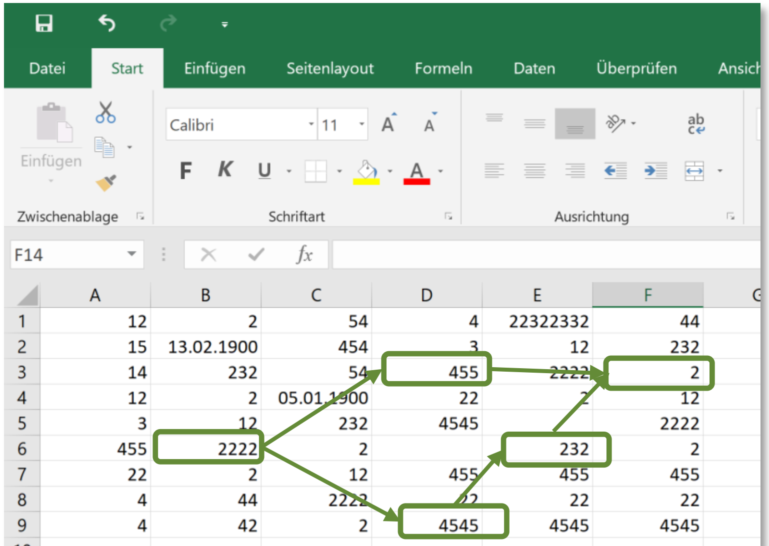
Kante: Job X muss vor Job Y beendet sein



In welcher Reihenfolge sollen die Jobs bearbeitet werden?

Anwendungen Job Scheduling

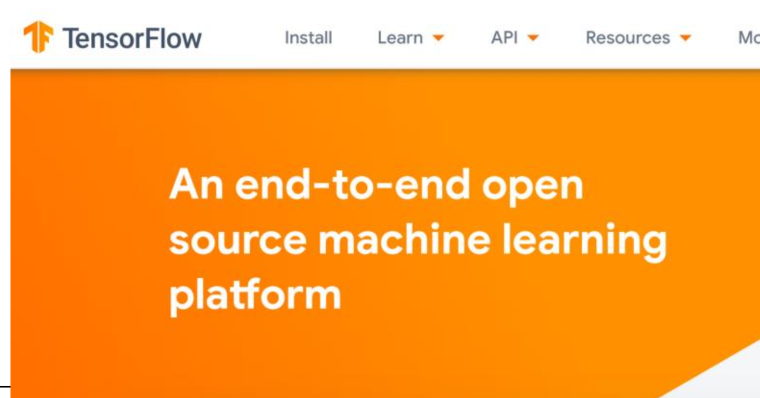
Quelle: **`www.gnu.org`**



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)



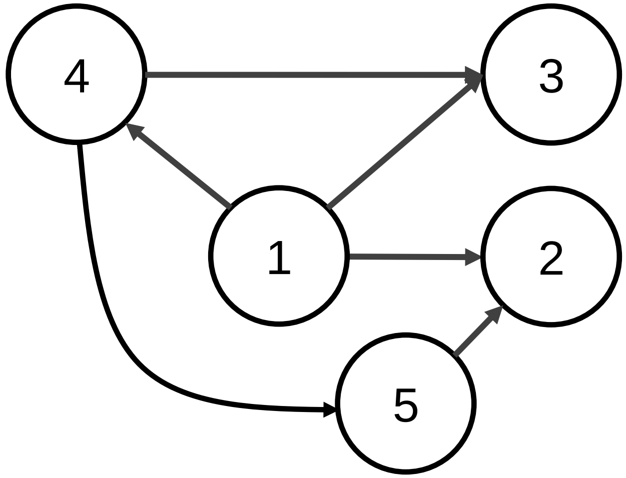
Quelle: **`www.tensorflow.org`**

(Berechnung als Computation Graph)

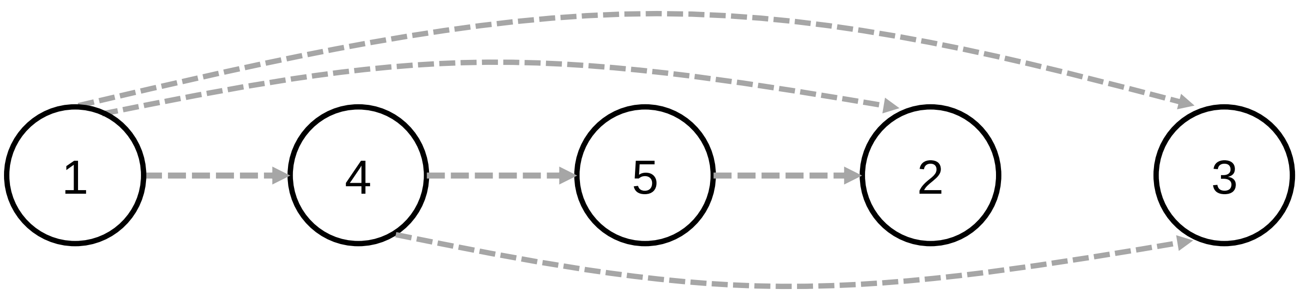
Abstrakte Modellierung

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

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

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

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

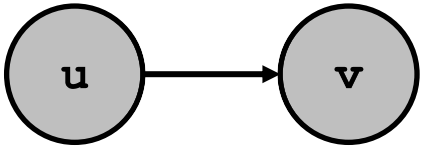
Topologisches Sortieren: Korrektheit (I)

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

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

Korrektheit:

Es genügt zu zeigen, dass jede von DFS inspizierte Kante $(\mathbf{u},\mathbf{v}) \in E$ erfüllt:
`v.finish<u.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).

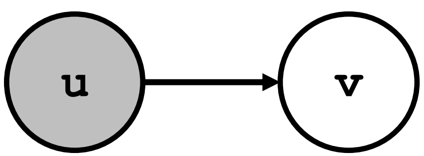
Topologisches Sortieren: Korrektheit (II)

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

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

Korrektheit:

Es genügt zu zeigen, dass jede von DFS inspizierte Kante $(\mathbf{u},\mathbf{v}) \in E$ erfüllt:
`v.finish<u.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.finish<u.finish`**, da Aufrufs-Stack nicht zu **`u`** zurückkehrt, bevor **`v`** abgeschlossen.

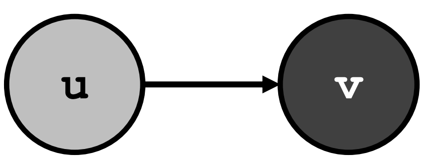
Topologisches Sortieren: Korrektheit (III)

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

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

Korrektheit:

Es genügt zu zeigen, dass jede von DFS inspizierte Kante $(\mathbf{u},\mathbf{v}) \in E$ erfüllt:
`v.finish<u.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.finish`** bereits gesetzt,
während **`u.finish`** erst später gesetzt wird,
also **`v.finish<u.finish`**.

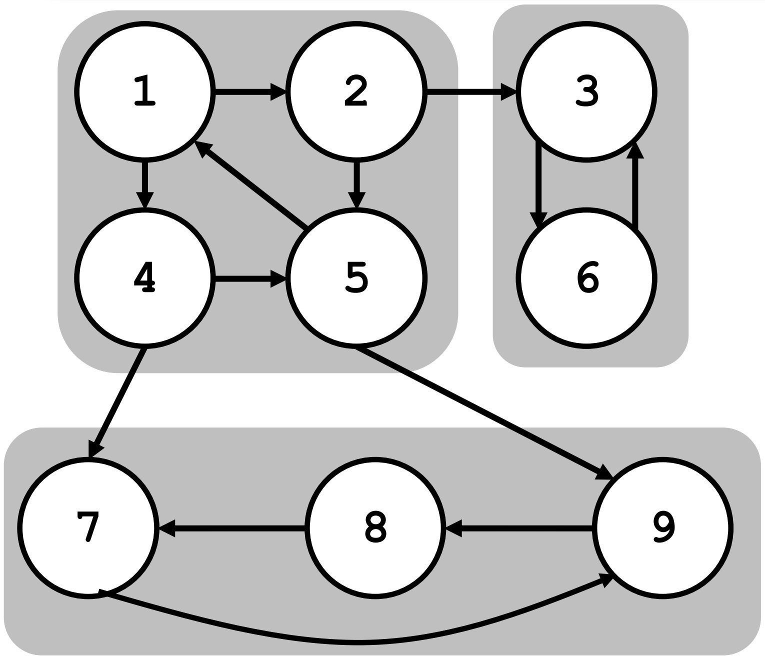
Wie können Sie bei der topologischen Sortierung mittels DFS fast ohne zusätzlichen Aufwand erkennen, ob ihr Graph einen Zyklus hat?

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

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

Starke Zusammenhangskomponenten

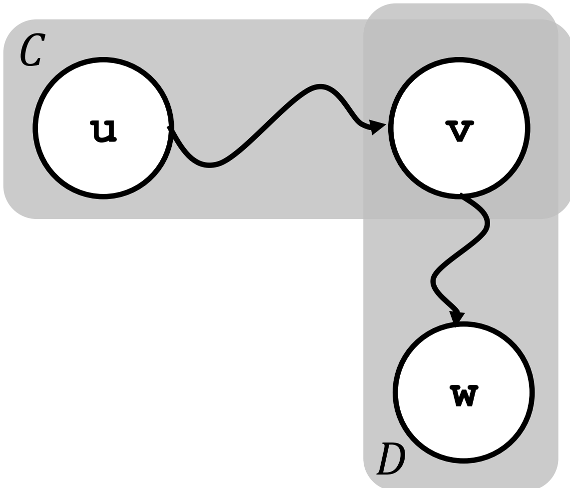
Eine starke Zusammenhangskomponente eines gerichteten Graphen $G = (V, E)$ ist eine Knotenmenge $C \subseteq V$, so dass
(a) es zwischen je zwei Knoten $u, v \in C$ einen Pfad von u nach v gibt, und
(b) es keine Menge $D \subseteq V$ mit $C \subsetneq D$ gibt, für die (a) auch gilt
(C ist maximal).



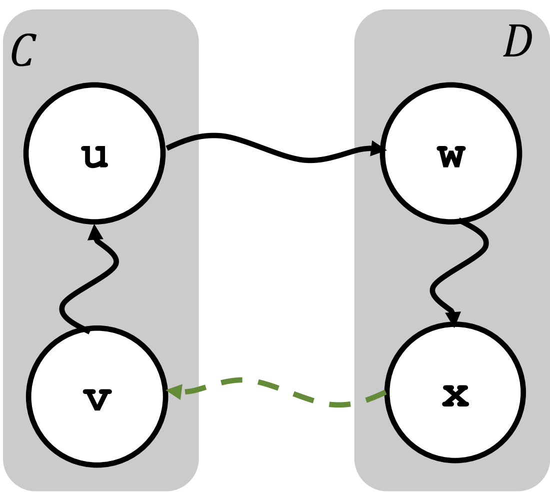
Anwendung: Model Checking für Korrektheitsnachweis von Systemen

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

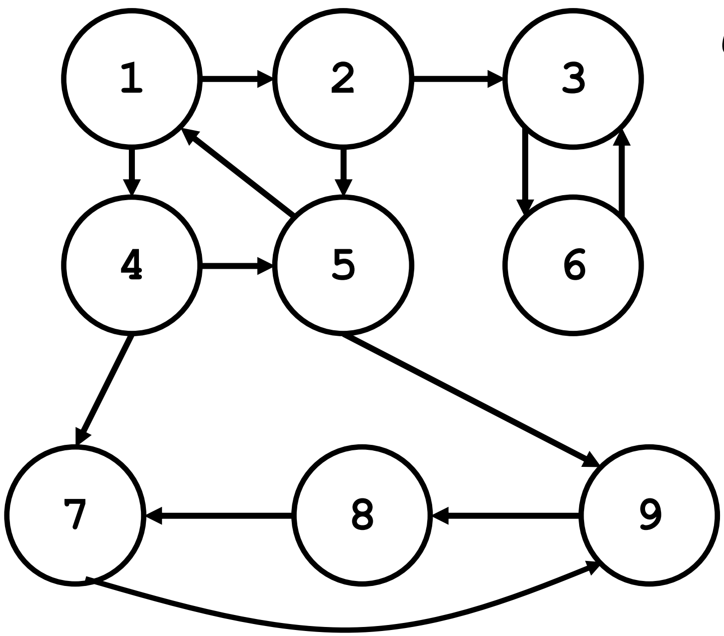
Algorithmus von Kosaraju

Lasse zweimal DFS laufen:

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

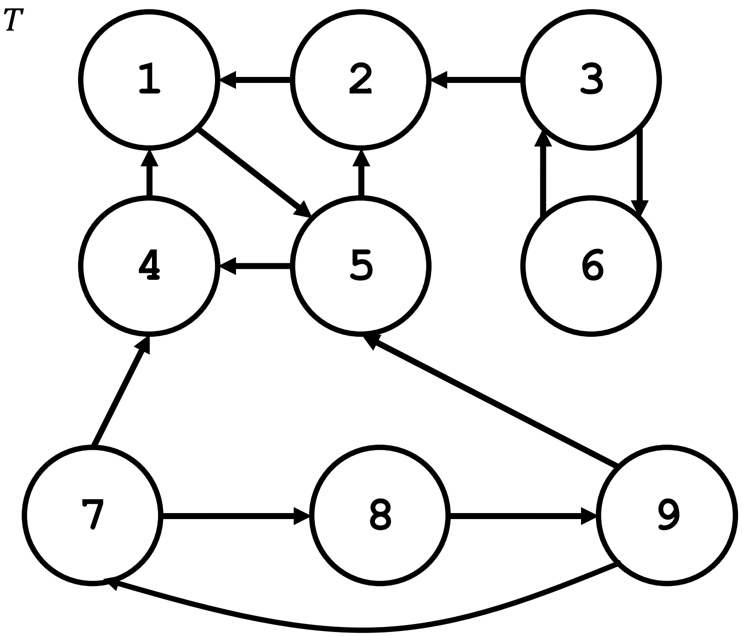
$$E^T = \{\ (v,u) \mid (u,v) \in E\ \}$$

„drehe Kanten in G um“



G

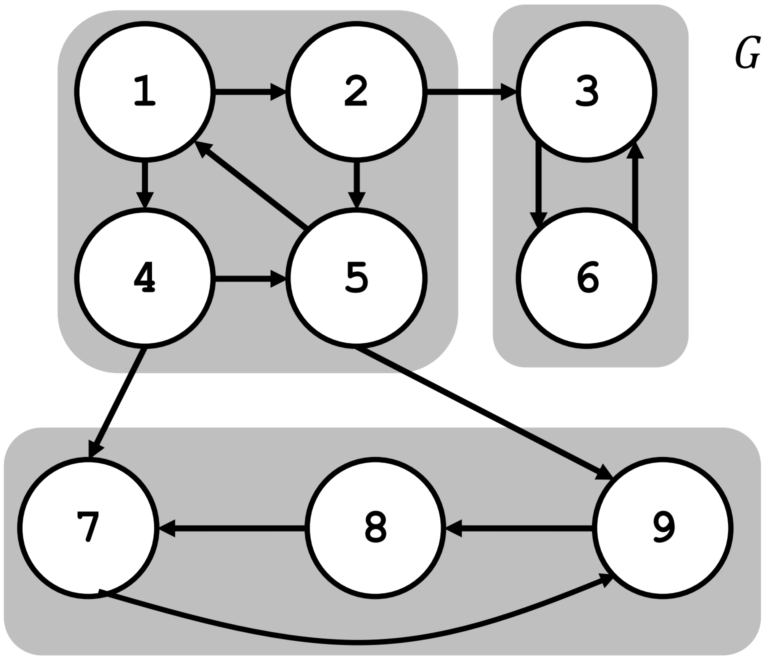
G^T

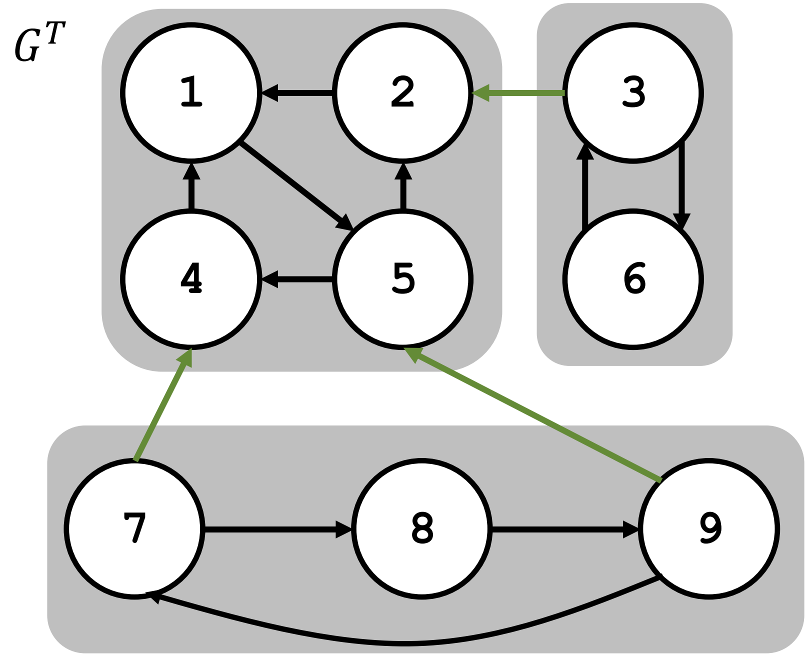


SCCs im transponierten Graphen

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

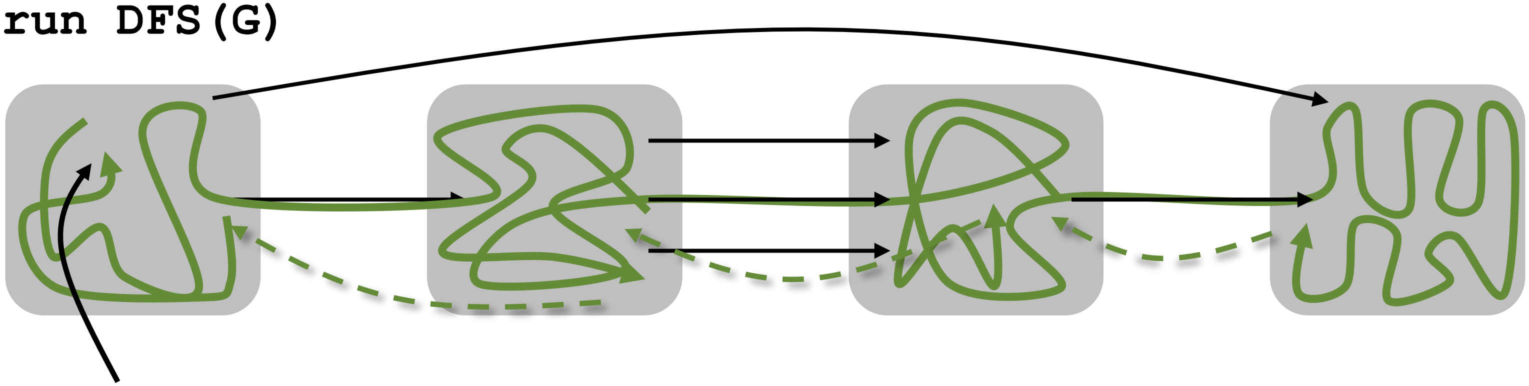
```
SCC(G) // G=(V,E) directed graph

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)

FOREACH u in V DO
   u.color=WHITE;
   u.pred=NIL;
time=0;
FOREACH u in V DO
   IF u.color==WHITE THEN
      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 step 1
4   output each DFS tree in 3 as one SCC
```

`1  run DFS(G)`



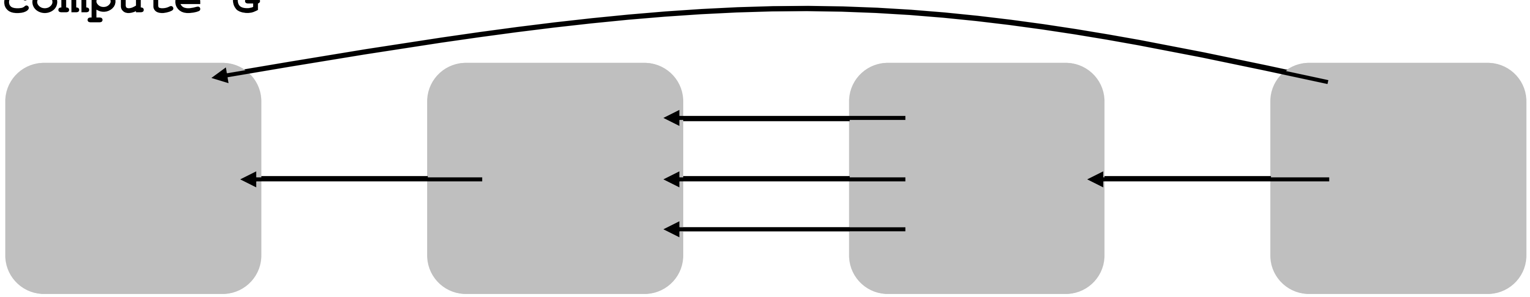
Knoten mit höchster **`finish`** time liegt in dieser SCC

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

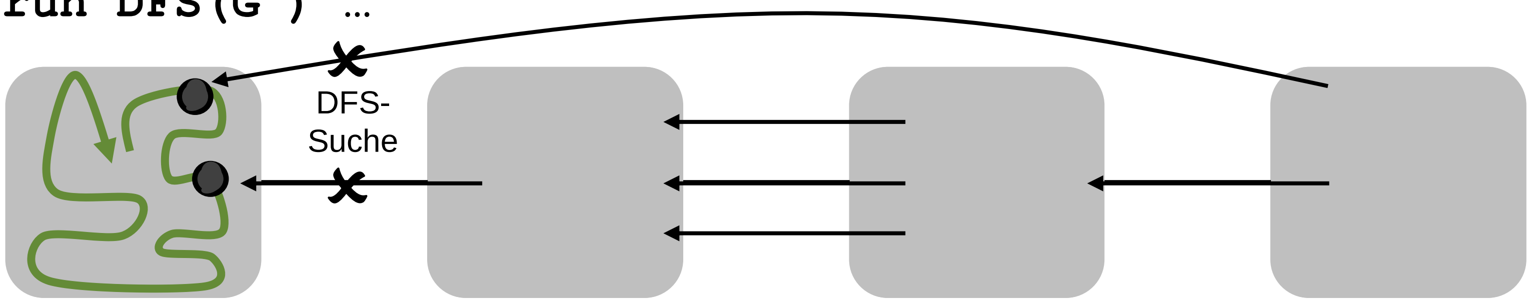


SCCs bleiben erhalten

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) …



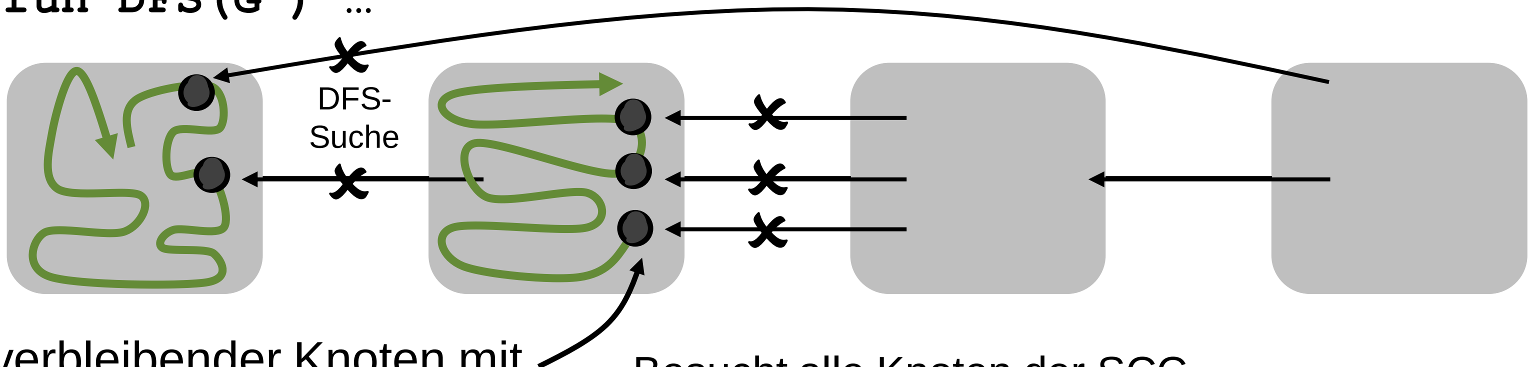
Besucht alle Knoten der SCC und kehrt dann zu Hauptschleife **DFS** zurück, da kein Übergang zum nächsten SCC

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) …



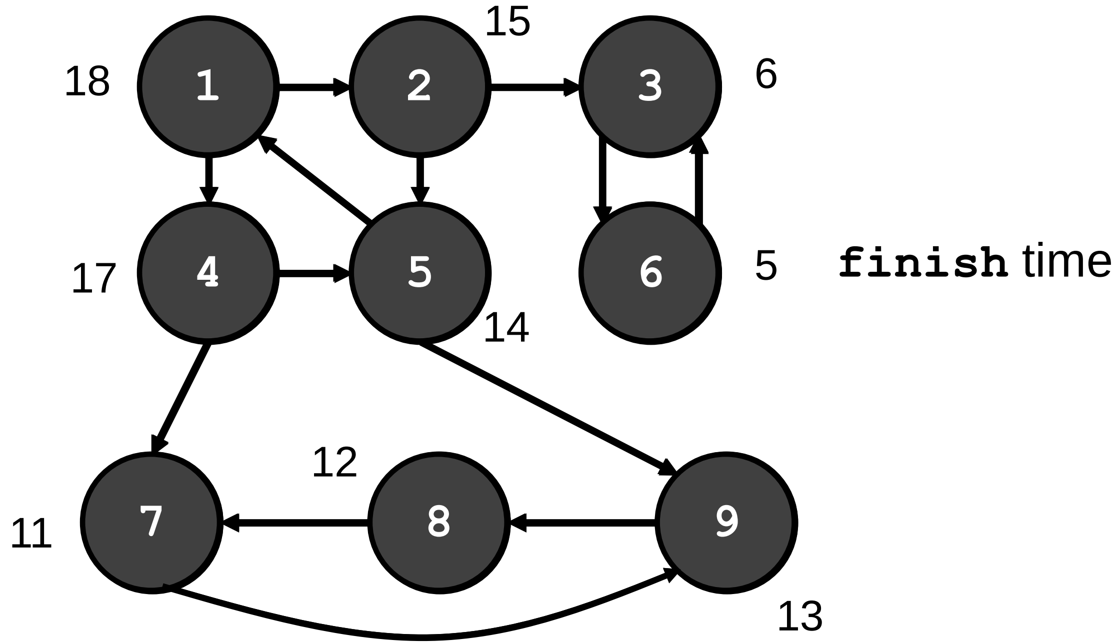
verbleibender Knoten mit nächsthöheren **finish** time liegt in dieser SCC

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

usw.

SCC Algorithmus: Beispiel (I)

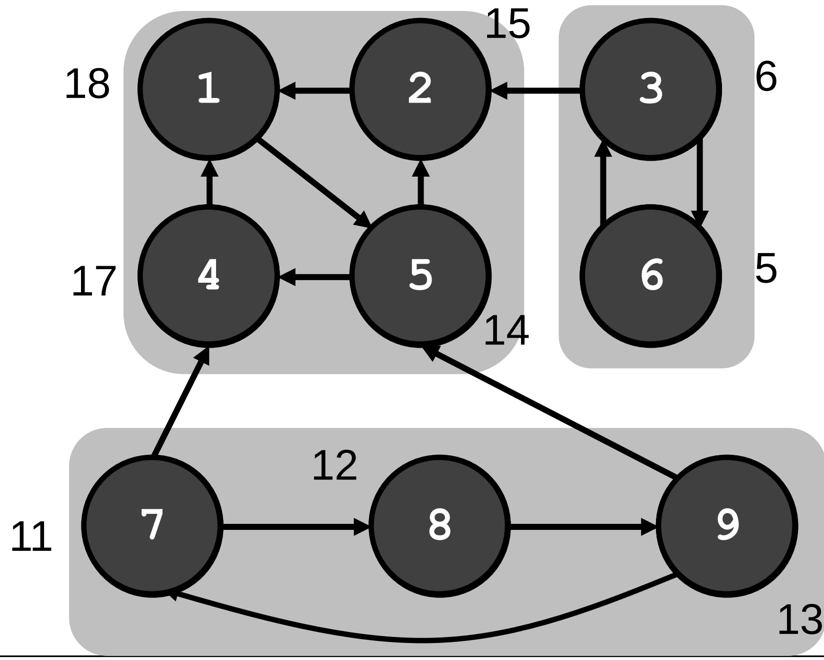
```
SCC(G) // G=(V,E) directed graph

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

Kosaraju 1978
Sharir 1981

Tarjan 1972

Kosarajus
Algorithmus
(hier betrachtet)

Tarjans
Algorithmus

Pfad-basierter
Algorithmus

zwei
DFS-Ausführungen

jeweils nur eine DFS-Ausführung,
speichern sich mehr Informationen „unterwegs“

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

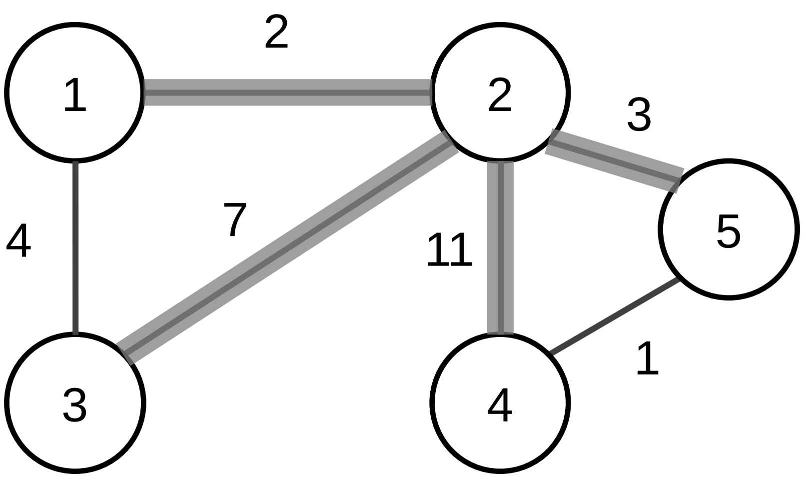
Minimale Spannbäume

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

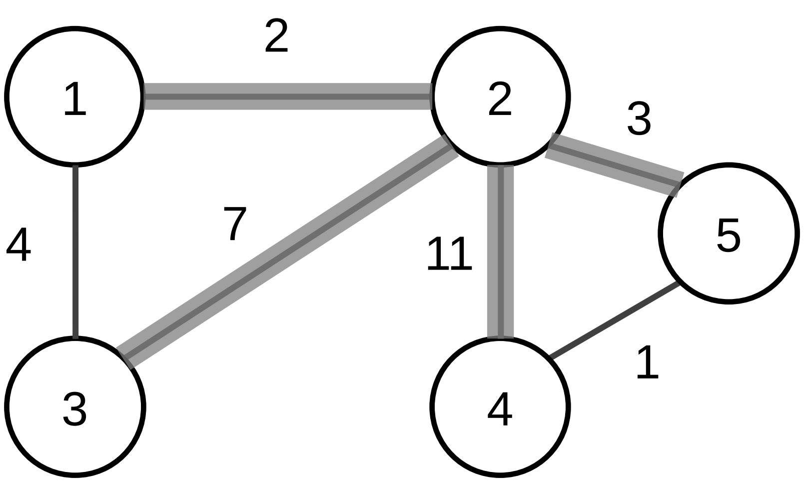
…
Der Spannbaum ist **minimal**, wenn

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

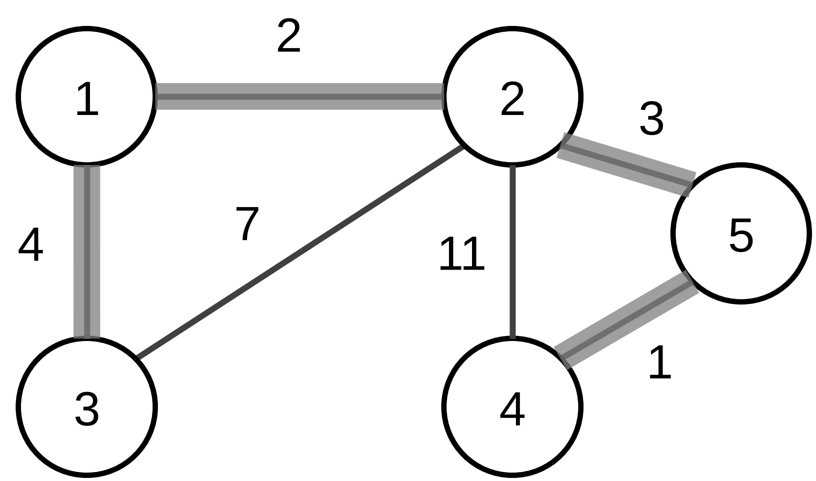
minimal für alle Spannbäume von G ist.

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

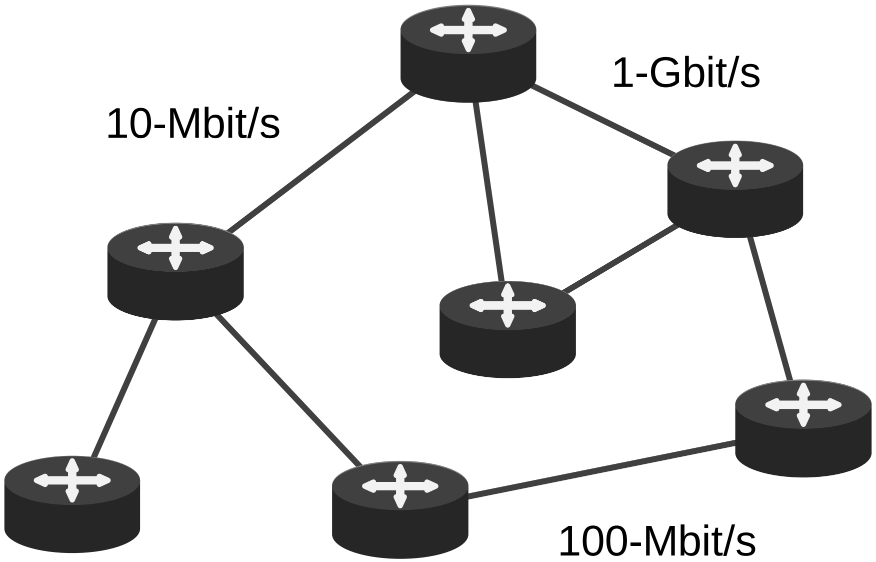
$$w(T) = 2 + 3 + 7 + 11 = 23$$



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



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∪{{u,v}}** noch Teilmenge eines MST ist

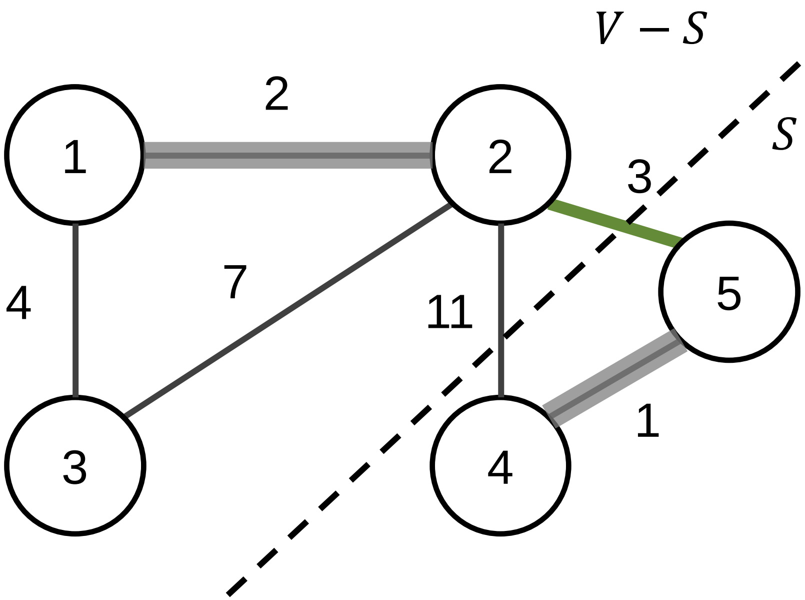
Allgemeiner MST-Algorithmus

```
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∪{{u,v}}** noch Teilmenge eines MST ist), ist am Ende der **WHILE**-Schleife **A** ein MST.

Terminologie



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** $\mathtt{A} \subseteq E$, wenn keine Kante $\{u, v\}$ aus $\mathtt{A}$ den Schnitt überbrückt

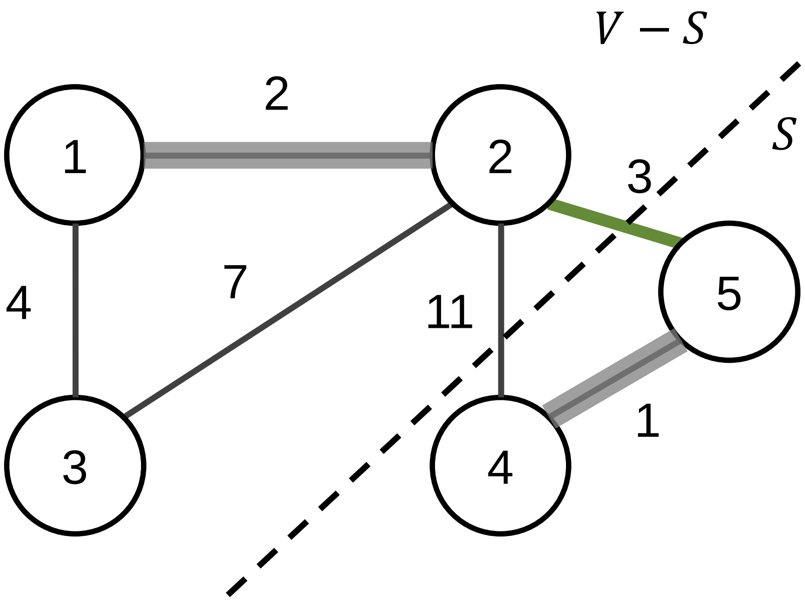
$\{u, v\}$ **leichte Kante für** $(S, V-S)$, wenn $\mathrm{w}(\{u, v\})$ minimal für alle den Schnitt überbrückenden Kanten

Kanten `{2,5}`, `{2,4}` überbrücken Schnitt

`A` (grau markierte Kanten) von Schnitt respektiert

Kante `{2,5}` ist leicht für Schnitt

Leicht = sicher



$\{\mathtt{u,v}\}$ **sicher für $\mathtt{A}$,** wenn $\mathtt{A} \cup \{\{\mathtt{u,v}\}\}$ 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** $\mathtt{A} \subseteq E$, wenn keine Kante $\{u,v\}$ aus $\mathtt{A}$ den Schnitt überbrückt

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

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

Leicht = sicher: Beweis (I)

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

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

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

`{u,v}` **sicher für `A`,** wenn
`A` ∪**`{{u,v}}`** Teilmenge eines MST

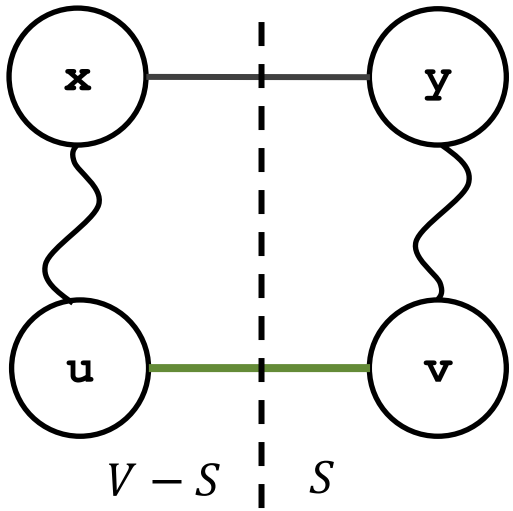
Schnitt $(\boldsymbol{S}, \boldsymbol{V} - \boldsymbol{S})$ partitioniert Knoten des Graphen in zwei Mengen

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

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

$\{\boldsymbol{u}, \boldsymbol{v}\}$ **leichte Kante für** $(\boldsymbol{S}, \boldsymbol{V} - \boldsymbol{S})$, wenn $\mathrm{w}(\{u, v\})$ 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.

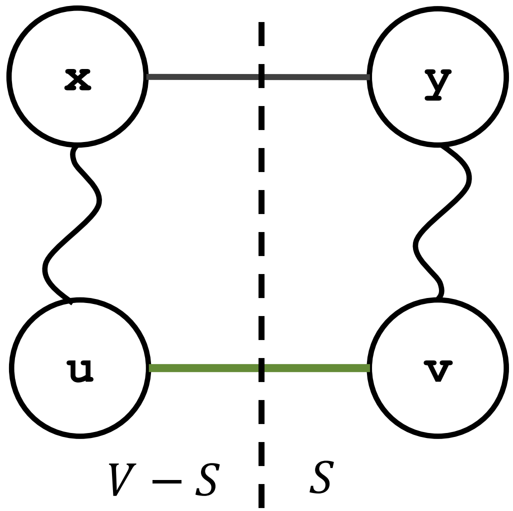
Da der Schnitt **A** respektiert, ist diese Kante $\{x,y\}$ nicht in **A** enthalten.

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

Sei **A** Teilmenge eines MST,
($\boldsymbol{S}$,$\boldsymbol{V-S}$) Schnitt, der **A** respektiert,
und $\{u,v\}$ eine leichte Kante,
die den Schnitt überbrückt.
Dann ist $\{u,v\}$ sicher für **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 **A** respektiert, ist diese Kante $\{x, y\}$ nicht in **A** enthalten.

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

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

„Leicht=sicher"
legt Greedy-Strategie
für konkrete Implementierung
nahe

Kruskal 1956

Jarnik 1930
Prim 1957
Dijkstra 1959

Algorithmus
von Kruskal

Algorithmus
von Prim

lässt parallel mehrere Unterbäume eines MST wachsen

konstruiert einen MST Knoten für Knoten

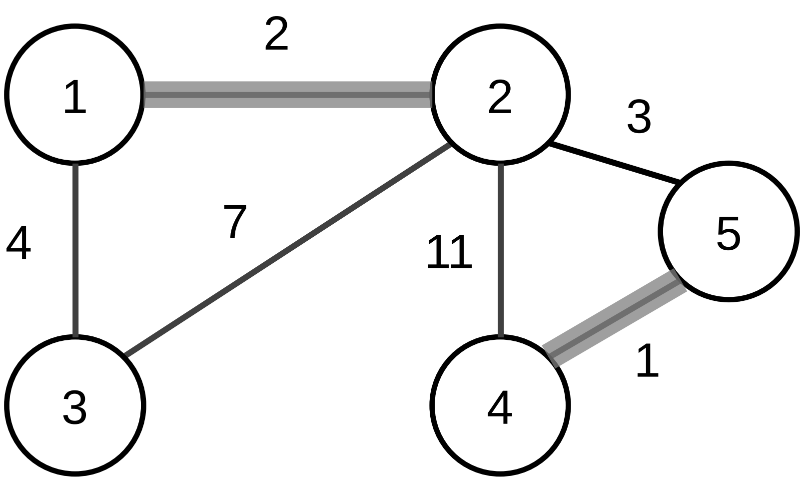
Bemerkung: beide Algorithmen funktionieren auch für negative Kantengewichte

Algorithmus von Kruskal

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

```
MST-Kruskal(G,w) // G=(V,E) undirected, connected graph
                    w weight function
    A=∅
    FOREACH v in V DO set(v)={v};
    Sort edges according to weight in nondecreasing order
    FOREACH {u,v} in E according to order DO
       IF set(u)!=set(v) THEN
          A = A∪{{u,v}}
          UNION(G,u,v);
    return A
```

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



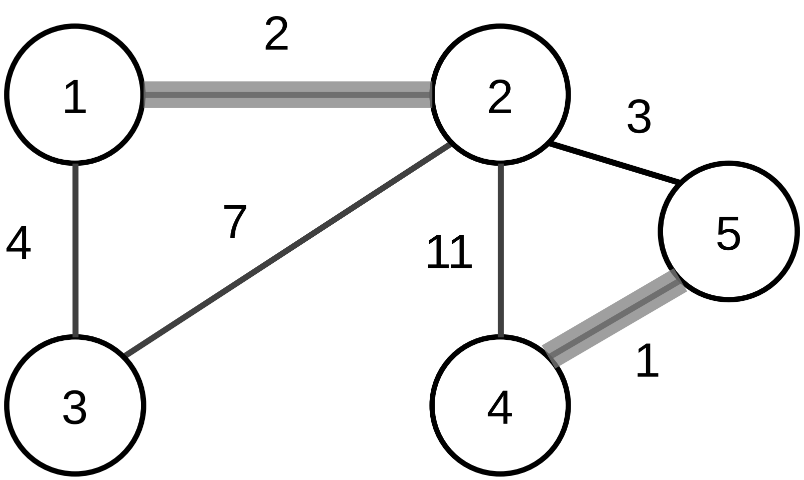
Zu jedem Knoten v sei $set(v)$ Menge von mit v durch **A** verbundenen Knoten. Zu Beginn ist $set(v) = \{v\}$.

$set(u), set(v)$ sind disjunkt oder identisch

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

Algorithmus von Kruskal: Korrektheit

```
MST-Kruskal(G,w) // G=(V,E) undirected, connected graph
                    w weight function
    A=∅
    FOREACH v in V DO set(v)={v};
    Sort edges according to weight in nondecreasing order
    FOREACH {u,v} in E according to order DO
       IF set(u)!=set(v) THEN
          A = A∪{{u,v}}
          UNION(G,u,v);
    return A
```



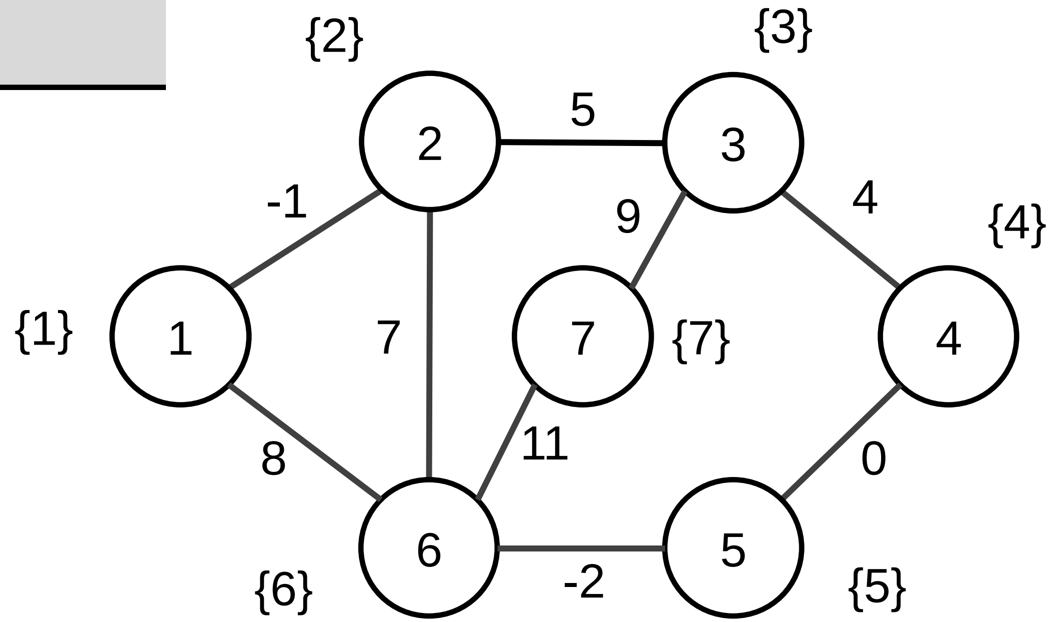
Jede ausgewählte Kante $\{u, v\}$ mit $set(u) \neq set(v)$ ist leicht für Schnitt $\big(set(u),\ V - set(u)\big)$.

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
```

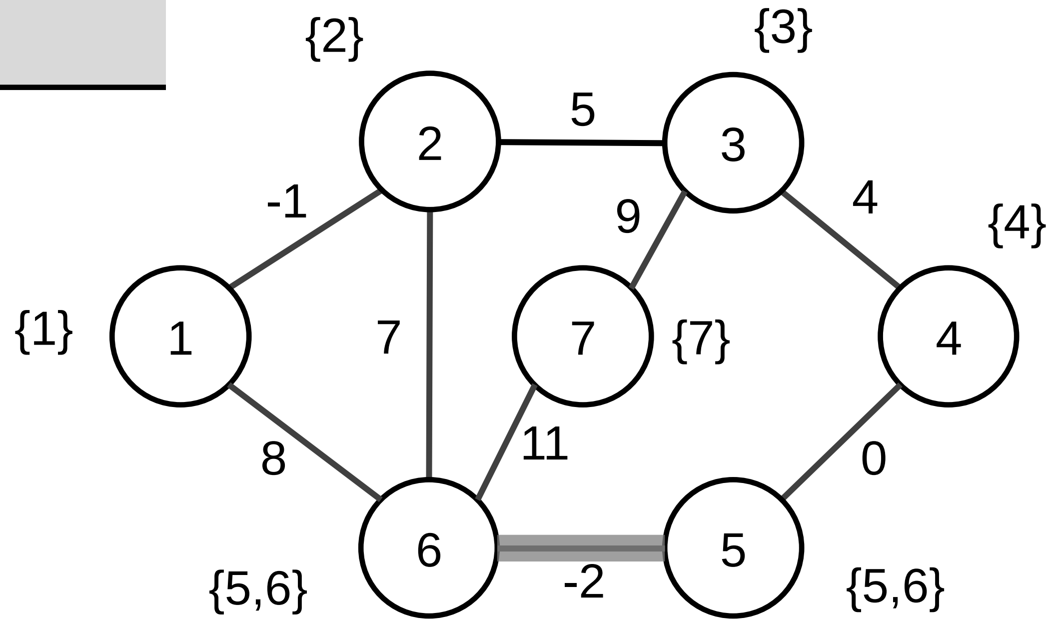


Initialisierung (1–3)

Algorithmus von Kruskal: Beispiel (II)

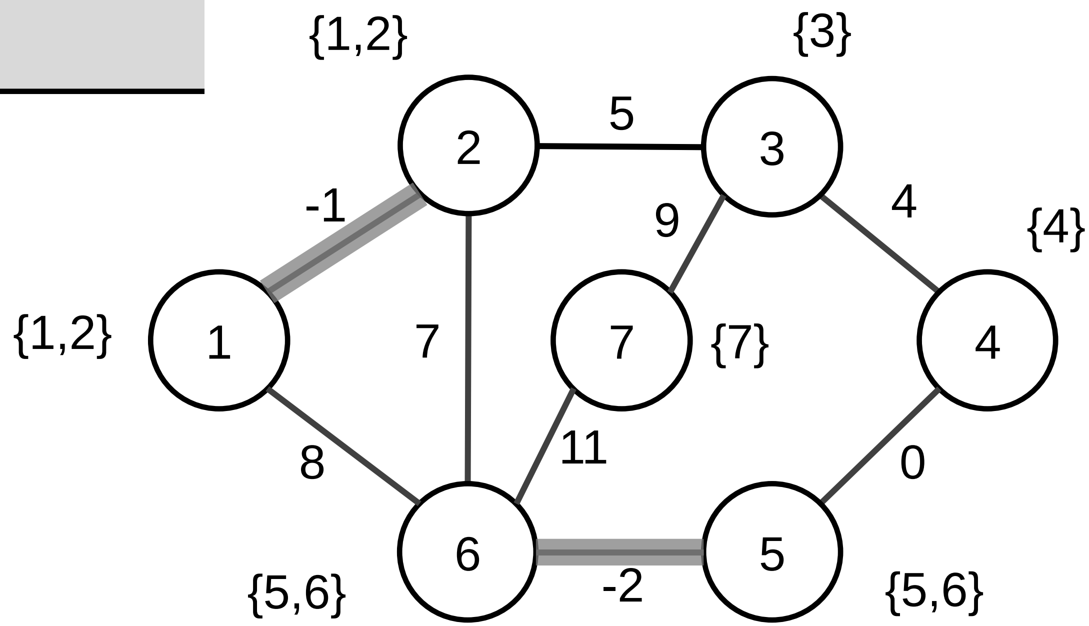
```
MST-Kruskal(G,w) // G=(V,E) undirected, connected graph
                    w weight function
1   A=∅
2   FOREACH v in V DO set(v)={v};
3   Sort edges according to weight in nondecreasing order
4   FOREACH {u,v} in E according to order DO
5      IF set(u)!=set(v) THEN
6         A = A∪{{u,v}}
7         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
```

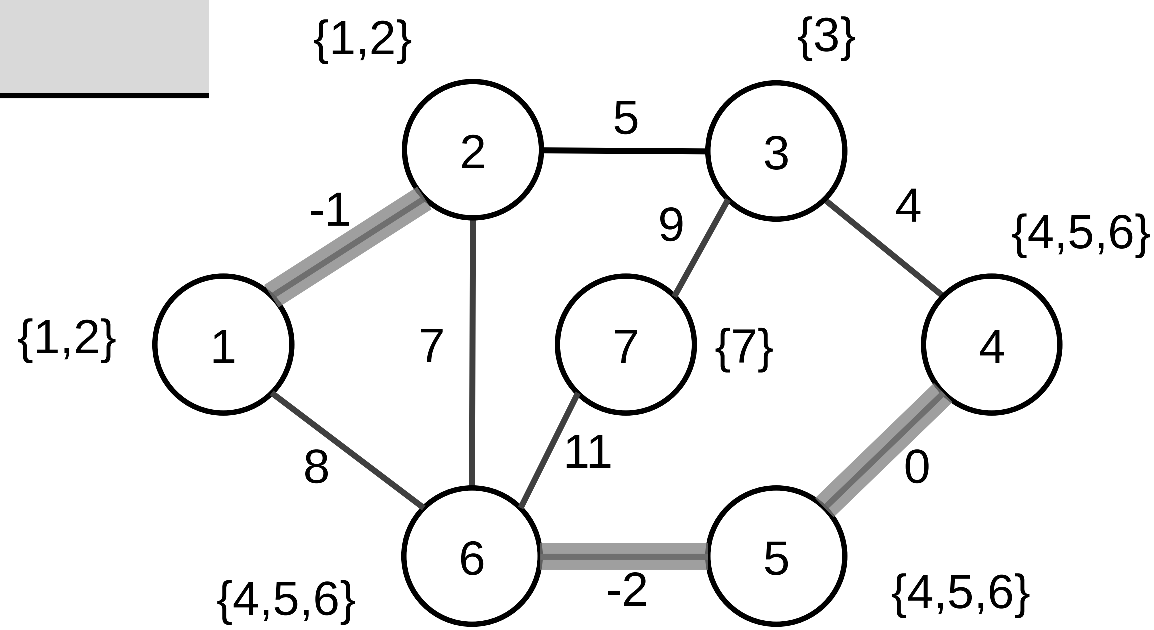


Schritte **4–8**:
Kante {1,2} aufnehmen

Algorithmus von Kruskal: Beispiel (IV)

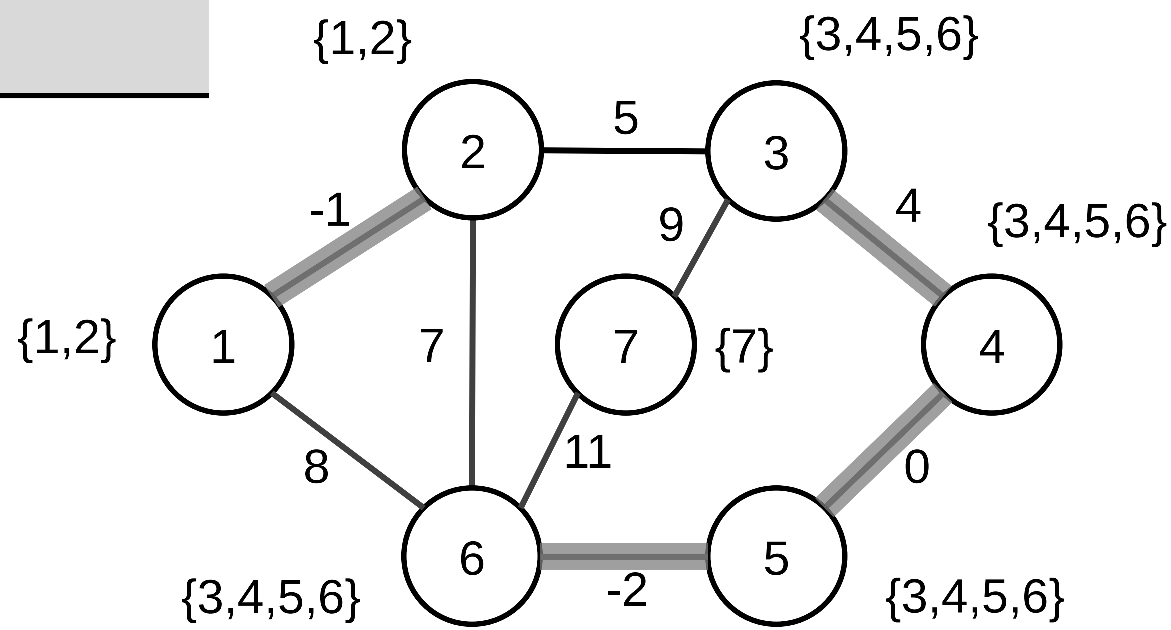
```
MST-Kruskal(G,w) // G=(V,E) undirected, connected graph
                    w weight function
1   A=∅
2   FOREACH v in V DO set(v)={v};
3   Sort edges according to weight in nondecreasing order
4   FOREACH {u,v} in E according to order DO
5      IF set(u)!=set(v) THEN
6         A = A∪{{u,v}}
7  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

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



Algorithmus von Kruskal: Beispiel (V)

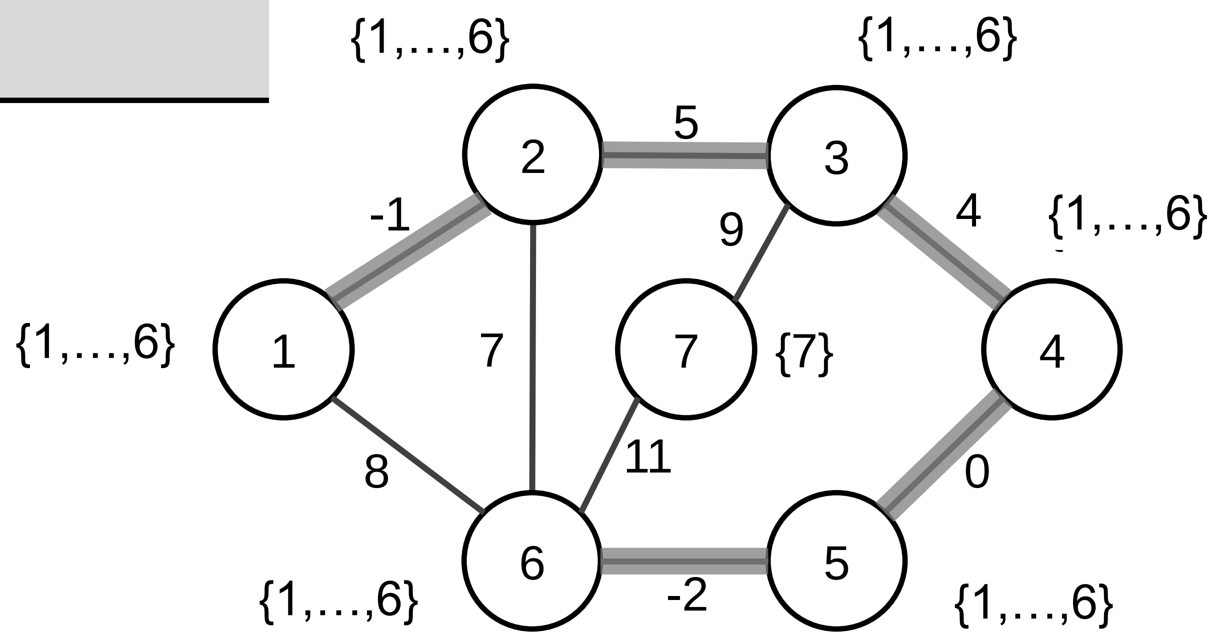
```
MST-Kruskal(G,w) // G=(V,E) undirected, connected graph
                    w weight function
1   A=∅
2   FOREACH v in V DO set(v)={v};
3   Sort edges according to weight in nondecreasing order
4   FOREACH {u,v} in E according to order DO
5      IF set(u)!=set(v) THEN
6         A = A∪{{u,v}}
7  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



Schritte **4–8**:
Kante {3,4} aufnehmen

Algorithmus von Kruskal: Beispiel (VI)

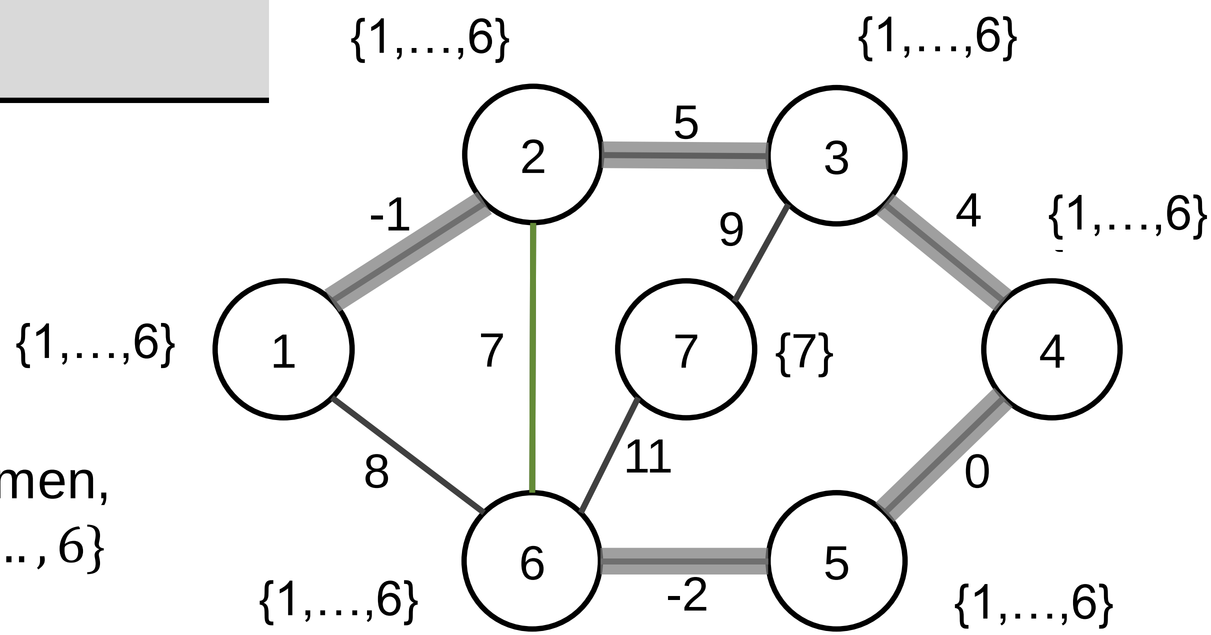
```
MST-Kruskal(G,w) // G=(V,E) undirected, connected graph
                    w weight function
1   A=∅
2   FOREACH v in V DO set(v)={v};
3   Sort edges according to weight in nondecreasing order
4   FOREACH {u,v} in E according to order DO
5      IF set(u)!=set(v) THEN
6         A = A∪{{u,v}}
7         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
```



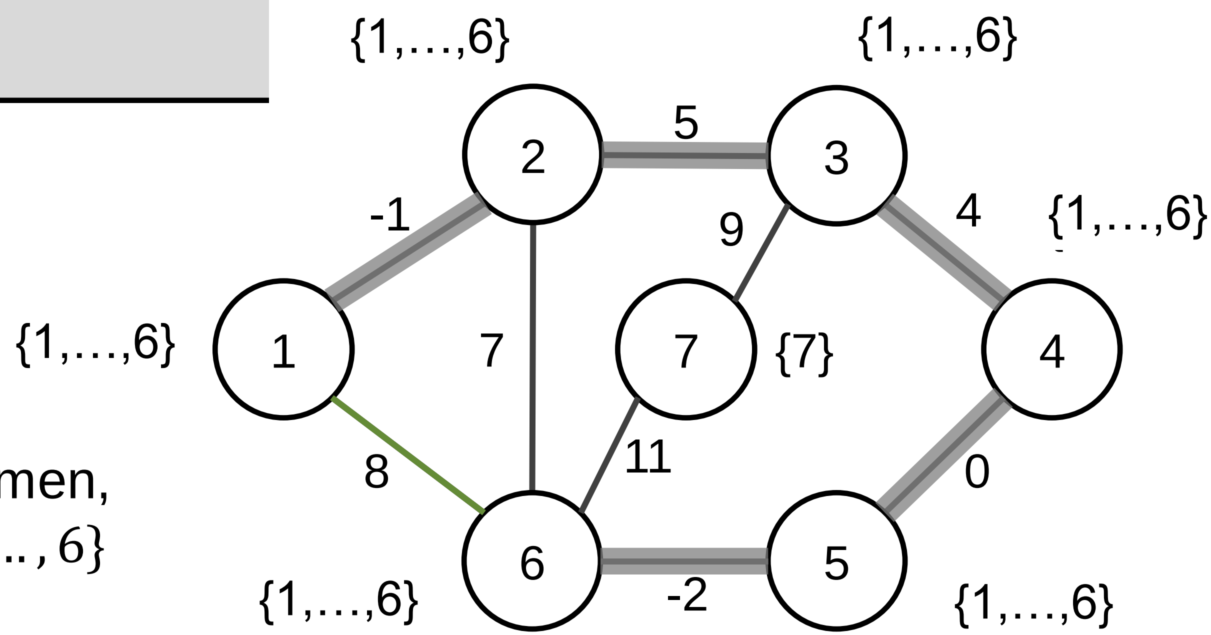
Schritte **4–8**:

Kante {2,6} **nicht** aufnehmen,

da $set(2) = set(6) = \{1, \ldots, 6\}$

Algorithmus von Kruskal: Beispiel (VIII)

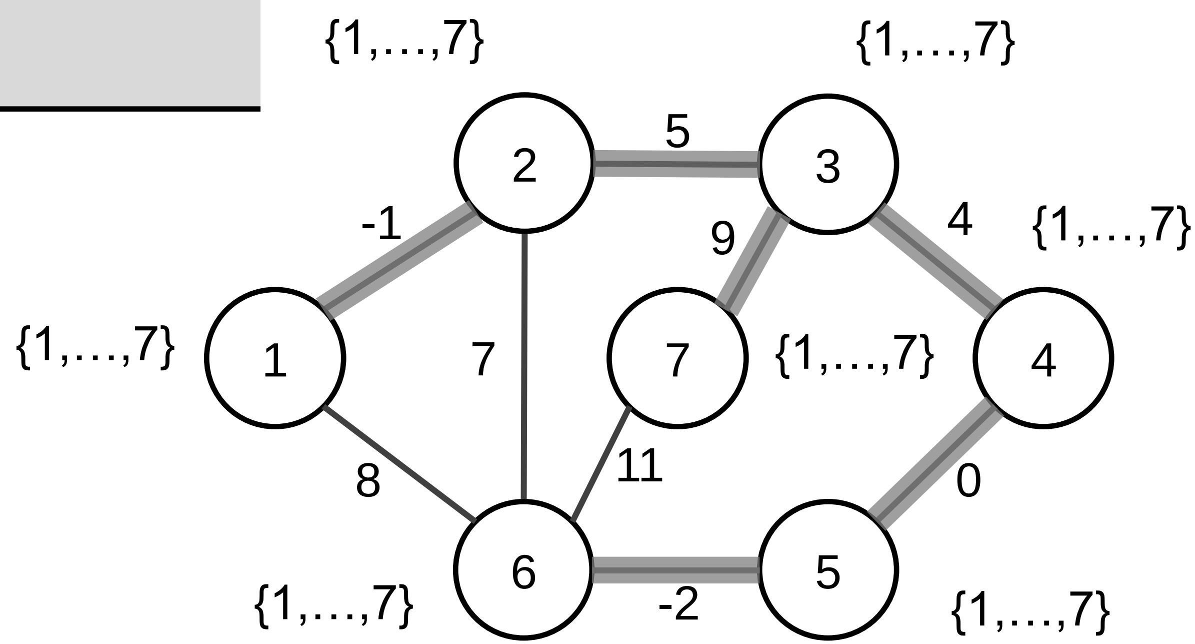
```
MST-Kruskal(G,w) // G=(V,E) undirected, connected graph
                    w weight function
1   A=∅
2   FOREACH v in V DO set(v)={v};
3   Sort edges according to weight in nondecreasing order
4   FOREACH {u,v} in E according to order DO
5      IF set(u)!=set(v) THEN
6         A = A∪{{u,v}}
7  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



Schritte **4–8**:
Kante {1,6} **nicht** aufnehmen,
da $set(1) = set(6) = \{1, \ldots, 6\}$

Algorithmus von Kruskal: Beispiel (IX)

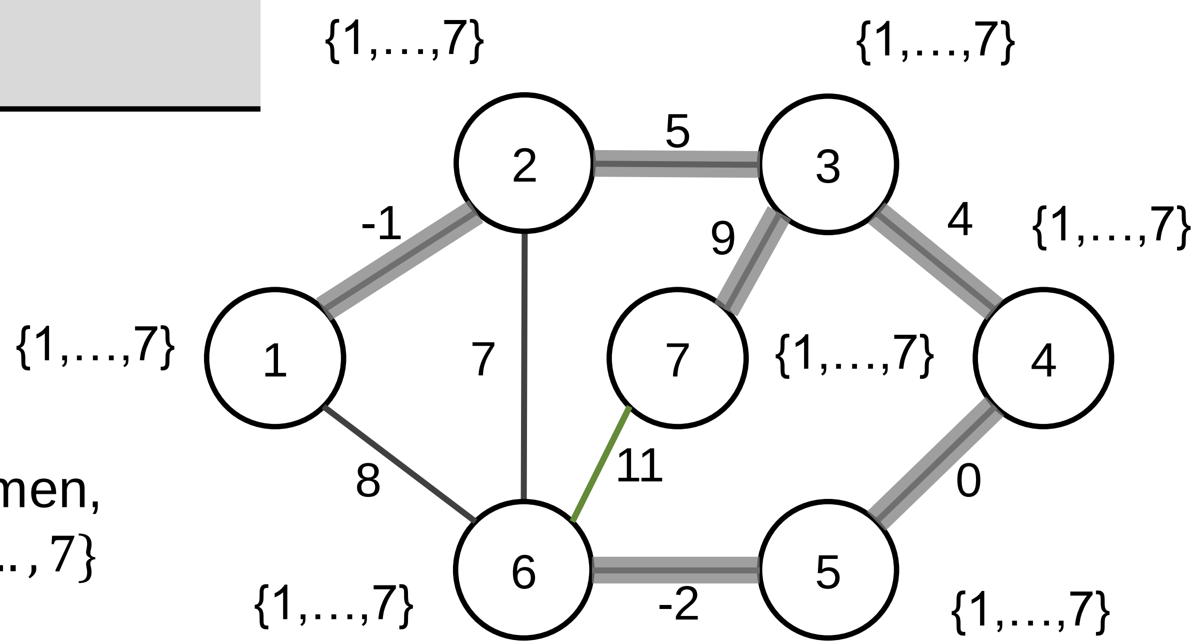
```
MST-Kruskal(G,w) // G=(V,E) undirected, connected graph
                    w weight function
1   A=∅
2   FOREACH v in V DO set(v)={v};
3   Sort edges according to weight in nondecreasing order
4   FOREACH {u,v} in E according to order DO
5      IF set(u)!=set(v) THEN
6         A = A∪{{u,v}}
7         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**:

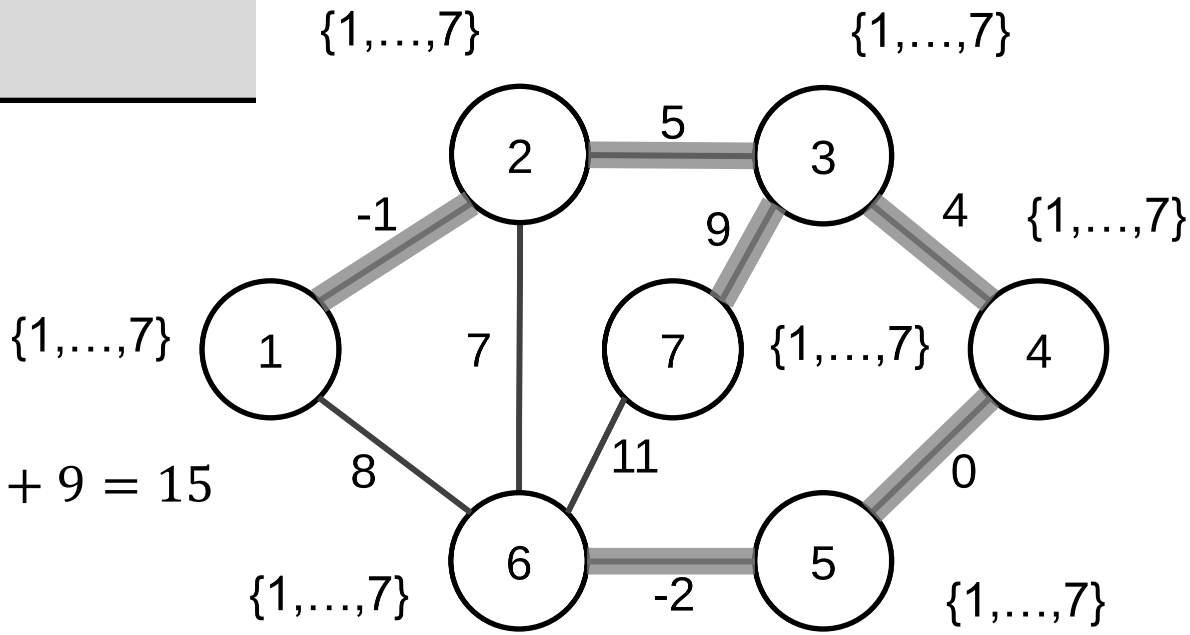
Kante {6,7} **nicht** aufnehmen,

da $set(6) = set(7) = \{1, \ldots, 7\}$

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
```

$$w(T) = -1 + 5 + 4 + 0 - 2 + 9 = 15$$



Algorithmus von Kruskal: Laufzeit

```
MST-Kruskal(G,w) // G=(V,E) undirected, connected graph
                    w weight function
A=∅
FOREACH v in V DO set(v)={v};
Sort edges according to weight in nondecreasing order
FOREACH {u,v} in E according to order DO
   IF set(u)!=set(v) THEN
      A = A∪{{u,v}}
      UNION(G,u,v);
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

(Wahl des Wurzelknoten beliebig)

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

FOREACH v in V DO {v.key=∞; v.pred=NIL;}
r.key=-∞; Q=V;
WHILE !isEmpty(Q) DO
  u=EXTRACT-MIN(Q); //smallest key value
  FOREACH v in adj(u) DO
    IF v∈Q and w({u,v})<v.key THEN
       v.key=w({u,v});
       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

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

Algorithmus von Prim: Beispiel (I)

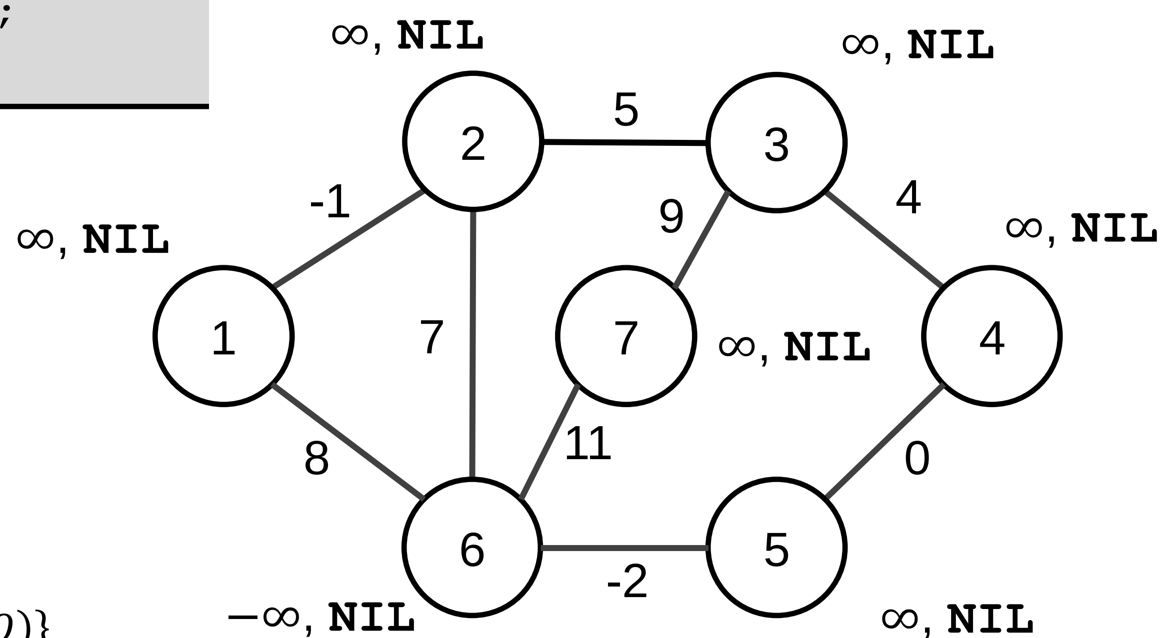
```
MST-Prim(G,w,r) // r root in V, MST given through v.pred values

1   FOREACH v in V DO {v.key=∞; v.pred=NIL;}
2   r.key=-∞; 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∈Q and w({u,v})<v.key
7          v.key=w({u,v});
8          v.pred=u;
```

Initialisierung (**1–2**)

r=6

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

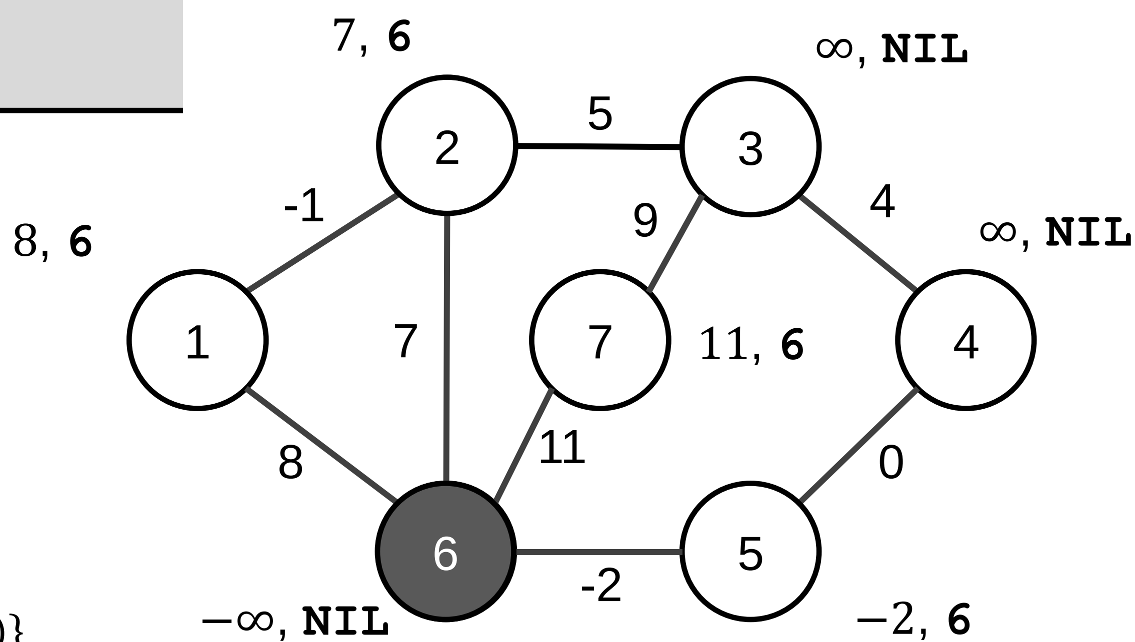
$\mathbf{A} = \{\,\{\mathbf{v,v.pred}\} \mid \mathbf{v} \in V - (\{r\} \cup Q)\}$



Algorithmus von Prim: Beispiel (II)

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

1   FOREACH v in V DO {v.key=∞; v.pred=NIL;}
2   r.key=-∞; 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∈Q and w({u,v})<v.key
7          v.key=w({u,v});
8          v.pred=u;
```

Schritte **3–8**:
u=r=6 extrahieren

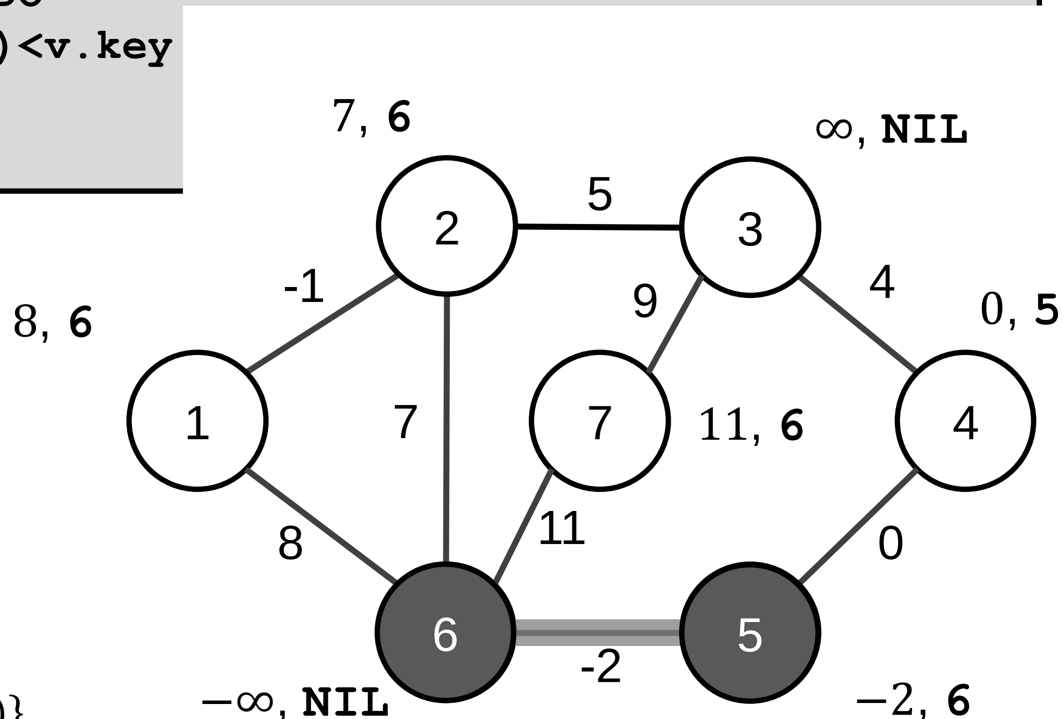


$Q = \{1,2,3,4,5,7\}$

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

Algorithmus von Prim: Beispiel (III)

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

1   FOREACH v in V DO {v.key=∞; v.pred=NIL;}
2   r.key=-∞; 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∈Q and w({u,v})<v.key
7          v.key=w({u,v});
8          v.pred=u;
```



Schritte **3–8**:
u=5 extrahieren

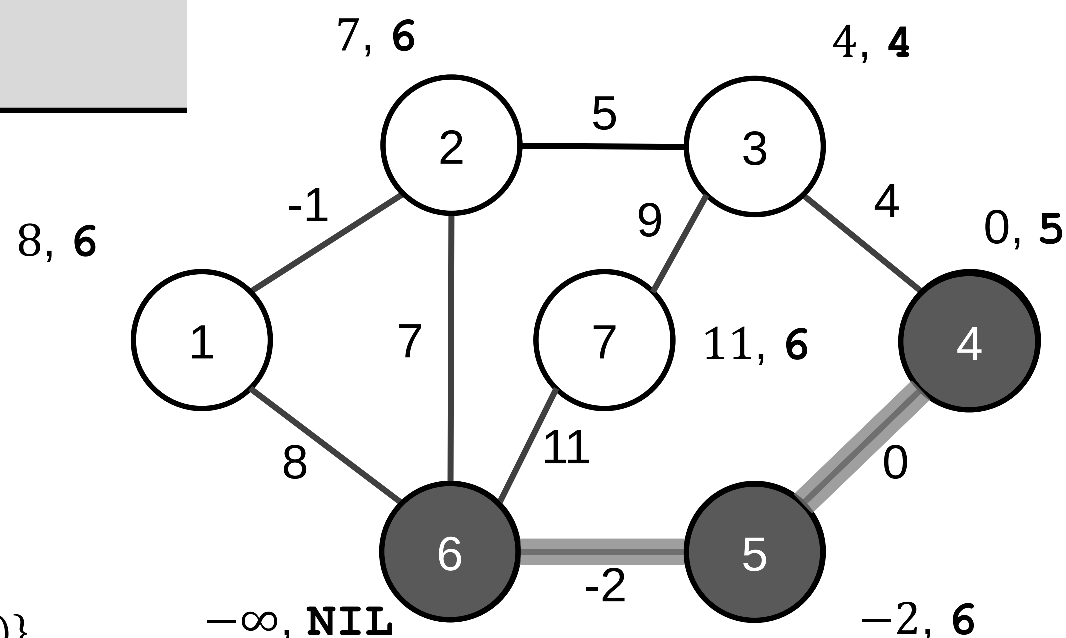
$Q = \{1,2,3,4,7\}$

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

Algorithmus von Prim: Beispiel (IV)

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

1   FOREACH v in V DO {v.key=∞; v.pred=NIL;}
2   r.key=-∞; 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∈Q and w({u,v})<v.key
7          v.key=w({u,v});
8          v.pred=u;
```

Schritte **3–8**:
`u=4` extrahieren



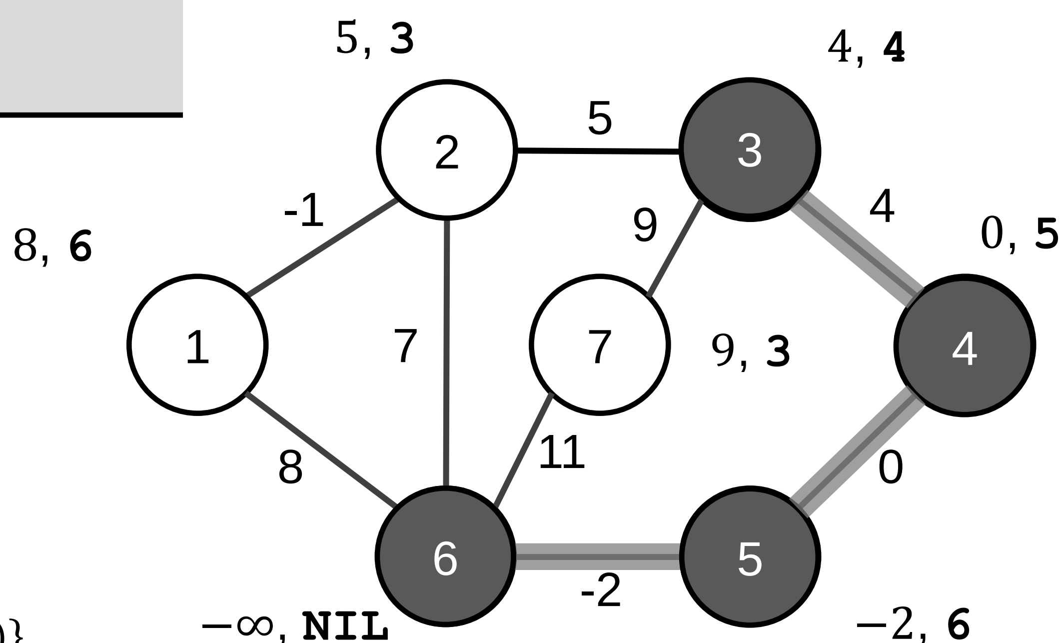
$Q = \{1,2,3,7\}$

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

Algorithmus von Prim: Beispiel (V)

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

1    FOREACH v in V DO {v.key=∞; v.pred=NIL;}
2    r.key=-∞; 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∈Q and w({u,v})<v.key
7           v.key=w({u,v});
8           v.pred=u;
```

Schritte **3–8**:
u=3 extrahieren



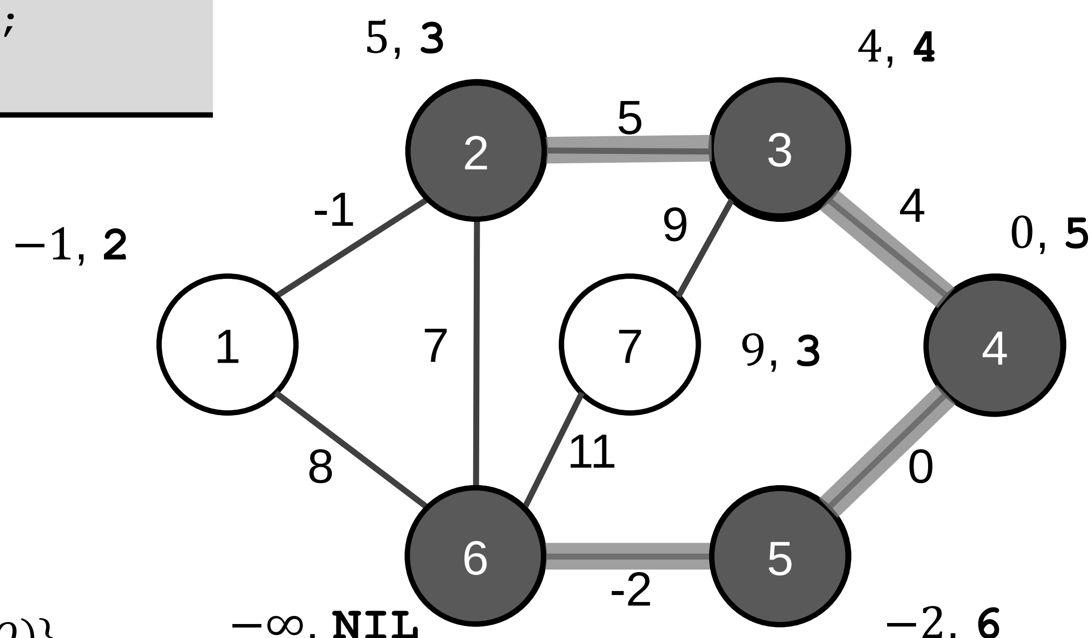
$Q = \{1,2,7\}$

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

Algorithmus von Prim: Beispiel (VI)

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

1    FOREACH v in V DO {v.key=∞; v.pred=NIL;}
2    r.key=-∞; 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∈Q and w({u,v})<v.key
7           v.key=w({u,v});
8           v.pred=u;
```

Schritte **3–8**:
u=2 extrahieren



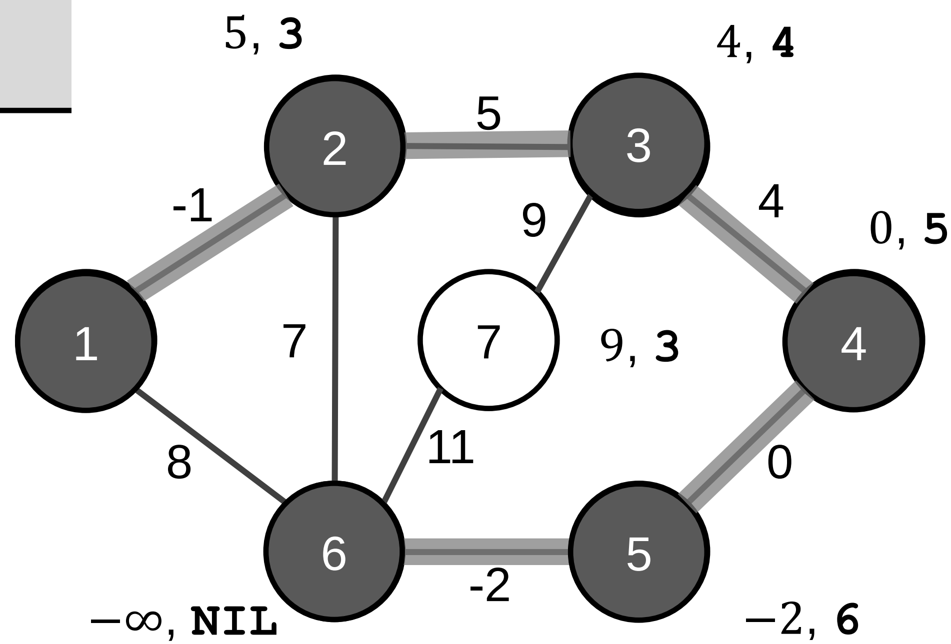
$$Q = \{1,7\}$$

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

Algorithmus von Prim: Beispiel (VII)

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

1   FOREACH v in V DO {v.key=∞; v.pred=NIL;}
2   r.key=-∞; 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∈Q and w({u,v})<v.key
7          v.key=w({u,v});
8          v.pred=u;
```

Schritte **3–8**:
u=1 extrahieren



$$Q = \{7\}$$

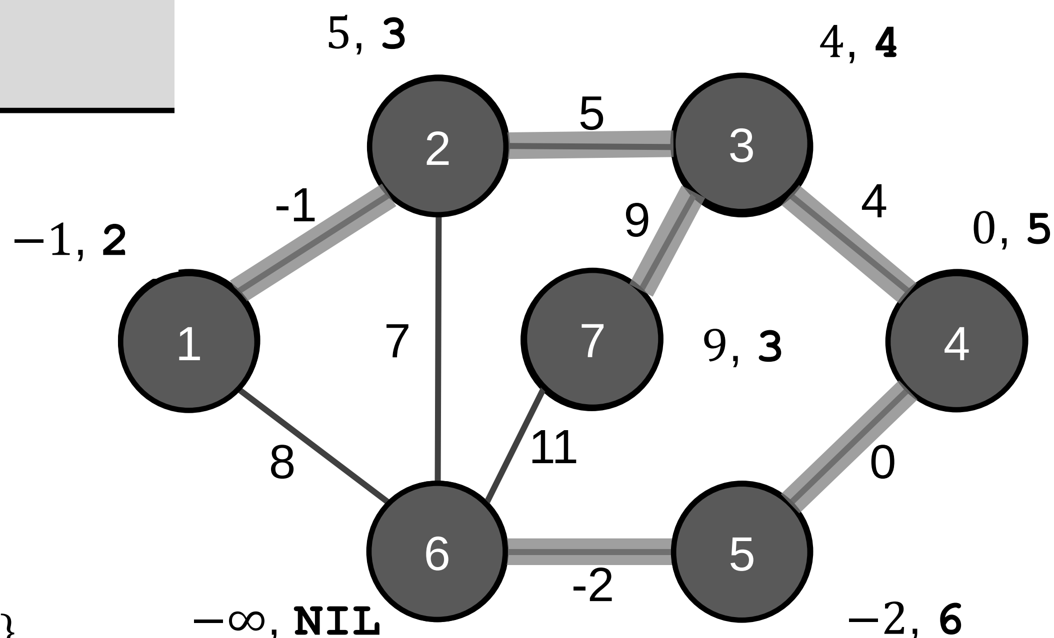
$\mathbf{A} = \{\, \{\mathbf{v,v.pred}\} \mid \mathbf{v} \in V - (\{r\} \cup Q)\}$

Algorithmus von Prim: Beispiel (VIII)

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

1    FOREACH v in V DO {v.key=∞; v.pred=NIL;}
2    r.key=-∞; 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∈Q and w({u,v})<v.key
7           v.key=w({u,v});
8           v.pred=u;
```

Schritte **3–8**:
u=7 extrahieren

$Q = \{\,\}$



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

Algorithmus von Prim: Korrektheit

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

1   FOREACH v in V DO {v.key=∞; v.pred=NIL;}
2   r.key=-∞; 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∈Q and w({u,v})<v.key THEN
7          v.key=w({u,v});
8          v.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.key` immer das kleinste Kantengewicht zu einem bereits aufgesammelten Knoten `v.pred` in $V-Q$

Daher beschreibt der in Schritt `4` ausgewählte Knoten `u` eine überbrückende, leichte Kante `(u,u.pred)`

Algorithmus von Prim: Laufzeit

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

  FOREACH v in V DO {v.key=∞; v.pred=NIL;}
  r.key=-∞; Q=V;
  WHILE !isEmpty(Q) DO
    u=EXTRACT-MIN(Q); //smallest key value
    FOREACH v in adj(u) DO
      IF v∈Q and w({u,v})<v.key THEN
         v.key=w({u,v});
         v.pred=u;
```

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

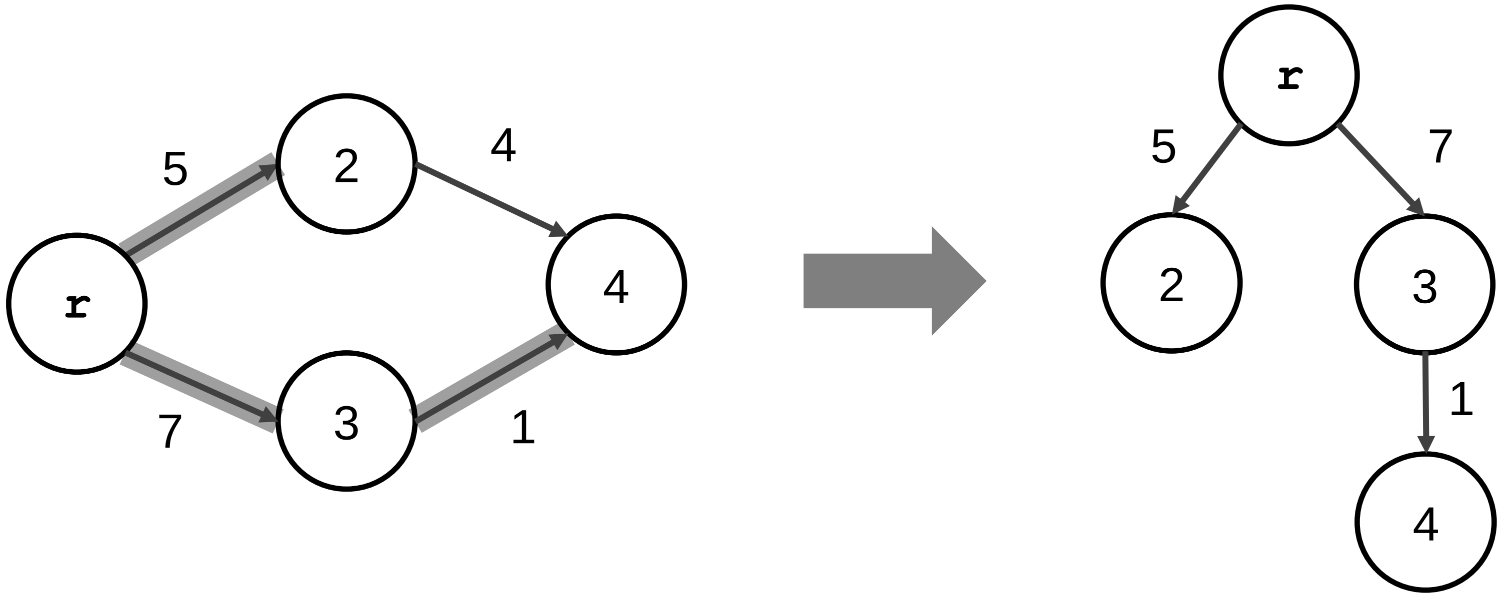
(mit vielen Optimierungen, speziell Fibonacci-Heaps)

zum Vergleich: Kruskal

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

Kruskal und Prim für gerichtete Graphen? (I)

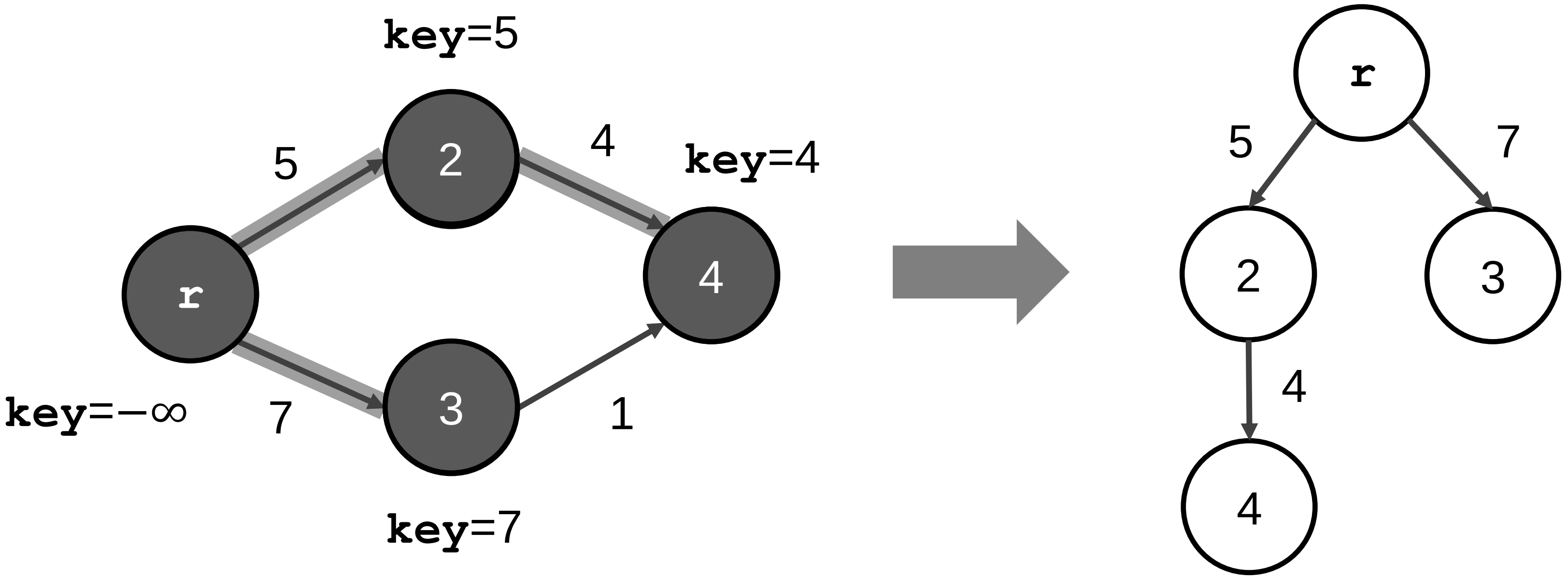
Directed MST (DMST) in Wurzelknoten **r** ist Spannbaum mit minimalem Gewicht (über alle Spannbäume mit Wurzel **r**)



$w(T) = 5 + 7 + 1 = 13$ minimal

Kruskal und Prim für gerichtete Graphen? (II)

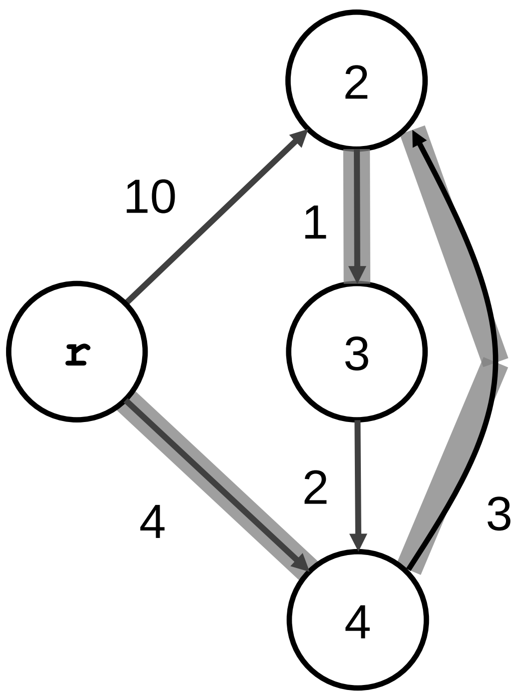
Prims Algorithmus findet Spannbaum mit Wurzel **`r`** (sofern es einen gibt), aber nicht immer minimalen Spannbaum:



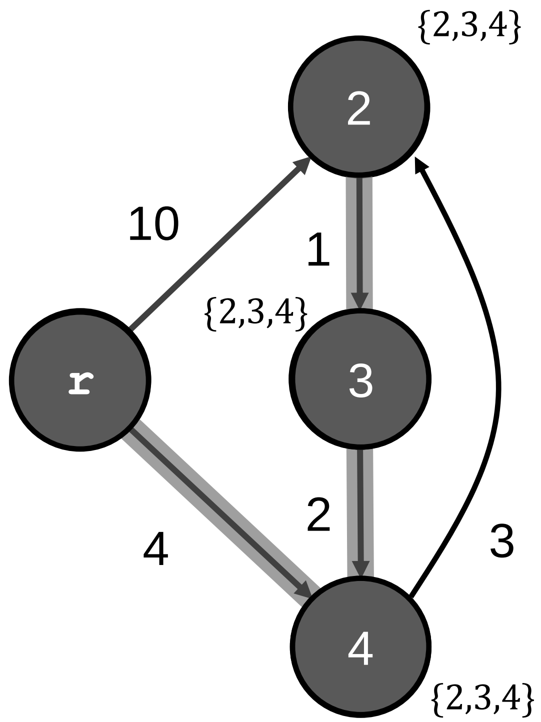
$w(T) = 5 + 7 + 4 = 16$ **nicht** minimal

Kruskal und Prim für gerichtete Graphen? (III)

Kruskals Algorithmus findet evtl. nicht Spannbaum mit Wurzel **`r`**:



Minimaler Spannbaum
$w(T) = 4 + 3 + 1 = 8$

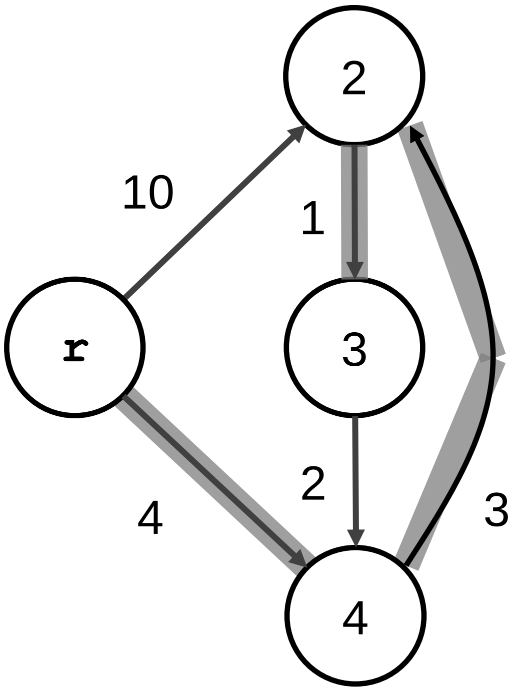


Kruskal wählt Kanten
(2,3) …(3,4) …(nicht (4,2)) …**(r,4)???**

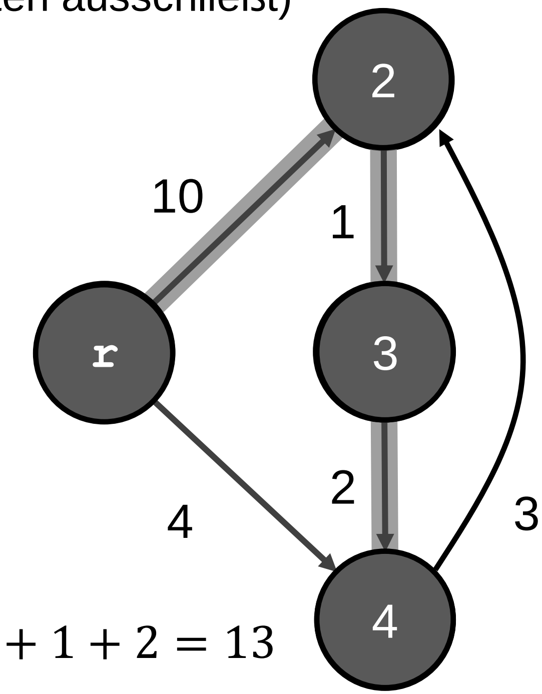
Kruskal und Prim für gerichtete Graphen? (IV)

Kruskals Algorithmus findet evtl.
nicht **minimalen** Spannbaum mit Wurzel `r`:

(selbst wenn man in Iteration bereits besuchte Knoten ausschließt)



Minimaler Spannbaum
$w(T) = 4 + 3 + 1 = 8$



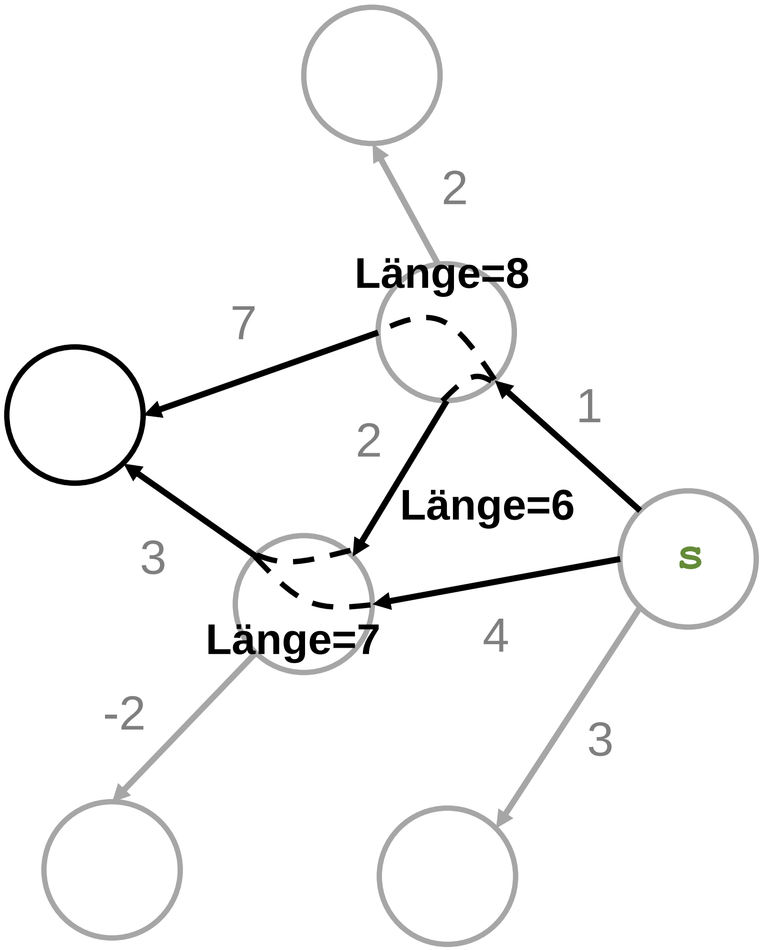
$w(T) = 10 + 1 + 2 = 13$

Kruskal wählt Kanten
(2,3) …(3,4) …(nicht (4,2)) …**(r,2)**

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
$p = (v_1, \ldots, v_k) \in V^k$
von $u = v_1$ zu $v = v_k$:

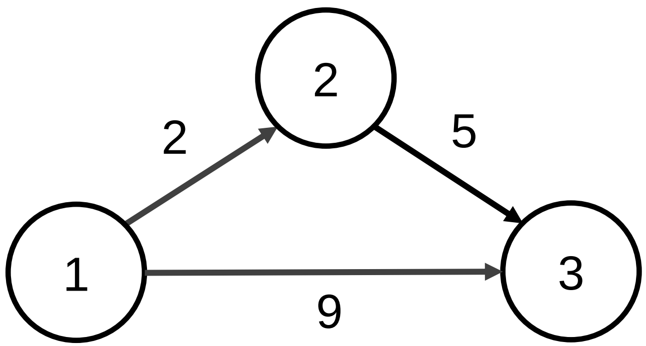
$$w(p) = \sum_{i=1}^{k-1} w((v_i, v_{i+1}))$$

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

SSSP vs. BFS, DFS, MST

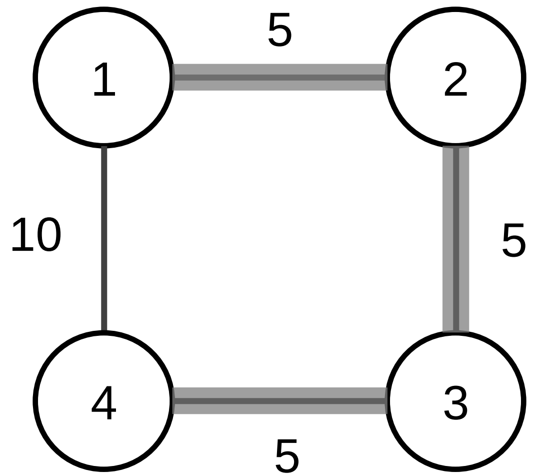
BFS + DFS: keine Kantengewichte

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



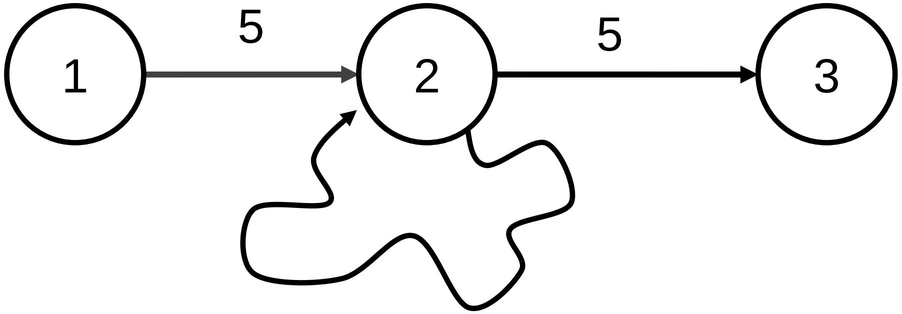
Kürzester „Kantenweg“ 1 nach 3 = 1→3
Kürzester „Gewichtsweg“ = 1→2→3

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



Kürzester Weg 1→4 mit Gewicht
10 im MST nicht enthalten

Negative Schwingungen

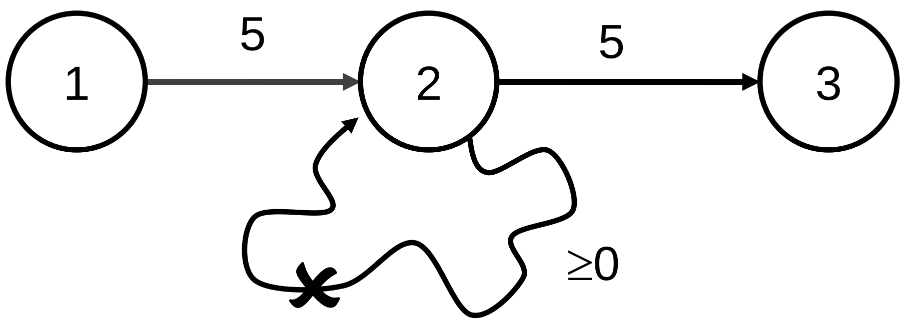


Zyklus mit Gesamtlänge = –5

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

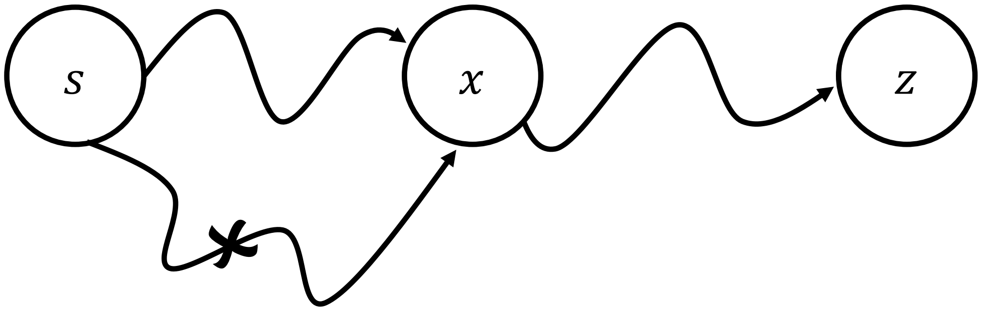
(sonst ohne Zyklus kürzerer Pfad)

+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

Lawler 1976

Dijkstra 1959

Algorithmus von Bellman-Ford	Algorithmus für dags	Algorithmus von Dijkstra
Laufzeit $= O(\lvert V\rvert \cdot \lvert E\rvert)$	*Laufzeit* $= O(\lvert V\rvert + \lvert E\rvert)$	*Laufzeit* $= O(\lvert V\rvert \cdot \log \lvert V\rvert + \lvert E\rvert)$
funktioniert allgemein	*funktioniert nur für dags*	*funktioniert nur für nicht-negative Kantengewichte*
(auch ungerichtete Graphen)		*(auch ungerichtete Graphen)*

Relax!

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

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

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

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

Bellman-Ford-Algorithmus

```
initSSSP(G,s,w)

   FOREACH v in V DO
      v.dist=∞;
      v.pred=NIL;
   s.dist=0;
```

```
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;
```

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

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

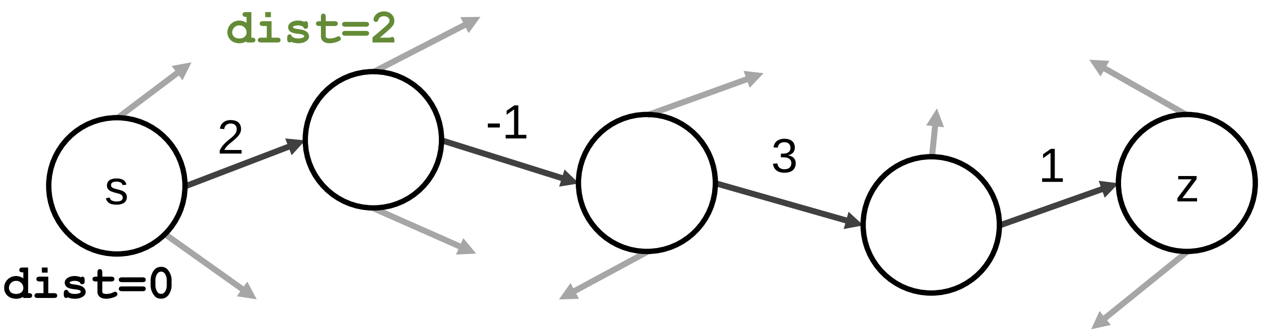
wegen geschachtelter `FOR`-Schleifen in 2 und 3

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.dist > u.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



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

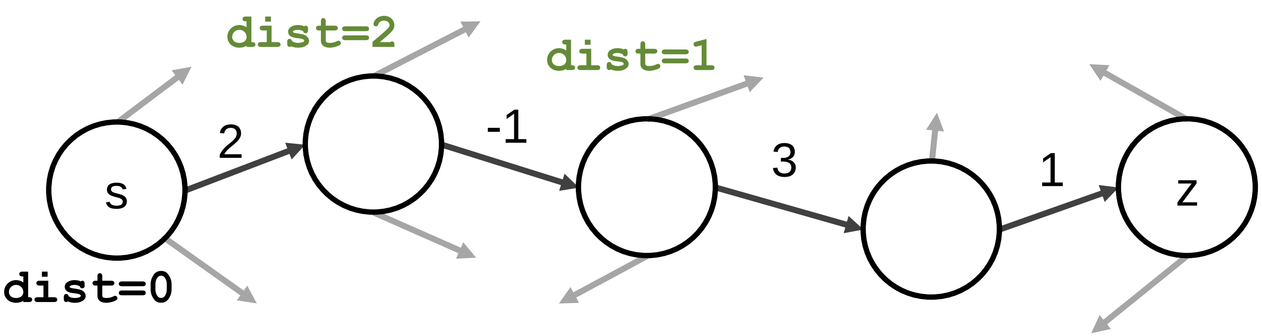
Wenn keine „negativen Zyklen“, gibt es kürzesten Pfad der Kantenlänge $\leq |V|-1$ ohne Schleifen

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.dist > u.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



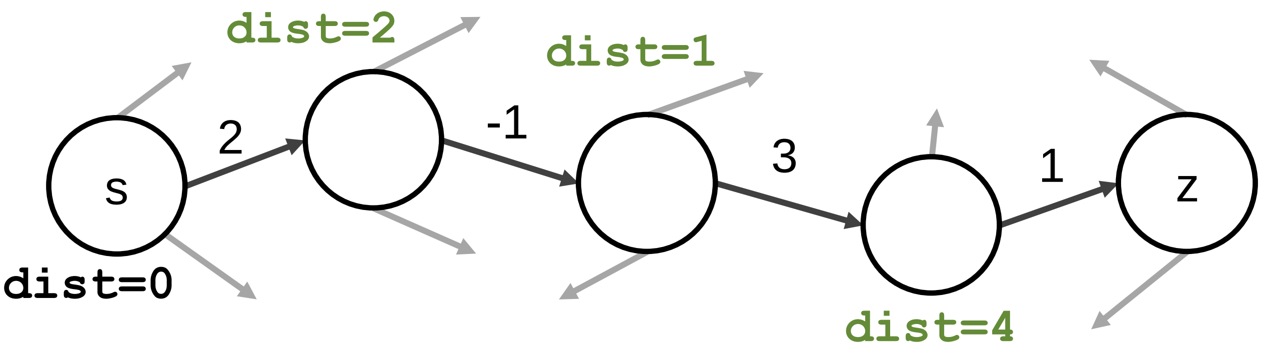
Bellman-Ford: Idee / Korrektheit (III)

```
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.dist > u.dist+w((u,v)) THEN
7         return false;
8   return true;
```

Betrachte Wirkung auf kürzesten Pfad von s nach z

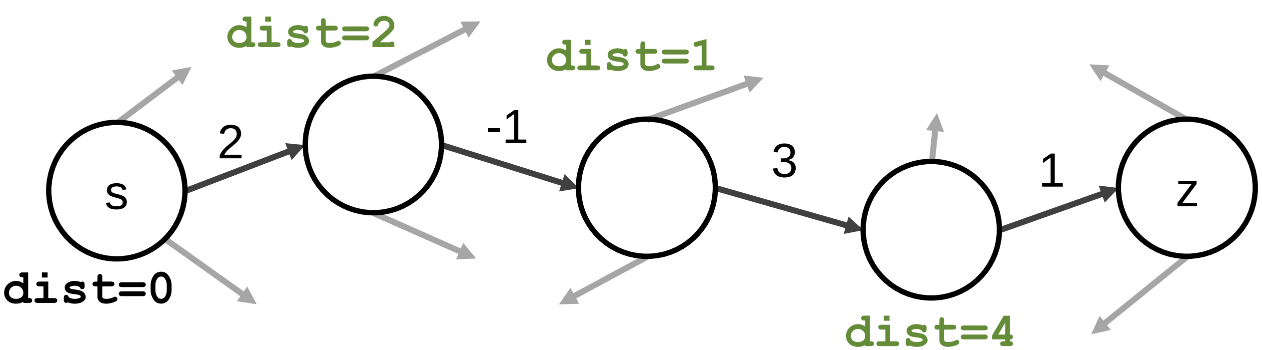
Dritte Iteration der **FOR**-Schleife in **2** erfasst (mindestens) den dritten Schritt von s zum dritten Knoten

usw.



Bellman-Ford: Idee / Korrektheit (IV)

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

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



Genau dann, wenn es keinen Pfad von **s** zu einem Knoten **u** gibt, dann bleibt **u.dist**=∞

Am Ende steht in jedem Knoten **u.dist**=$shortest(s,u)$

und

u.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.dist > u.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

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

und folglich Bedingung in **6** nie erfüllt.

Gilt auch im Fall $shortest(s,u) = \infty$;
der Fall $[shortest(s,v) = \infty$ und $shortest(s,u) < \infty]$ 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.dist > u.dist+w((u,v)) THEN
7         return false;
8   return true;
```

Prüfschritte **5-7**

Fall 2:
Wenn „negative Zyklen“ erreichbar, dann Rückgabe **false**

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

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

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

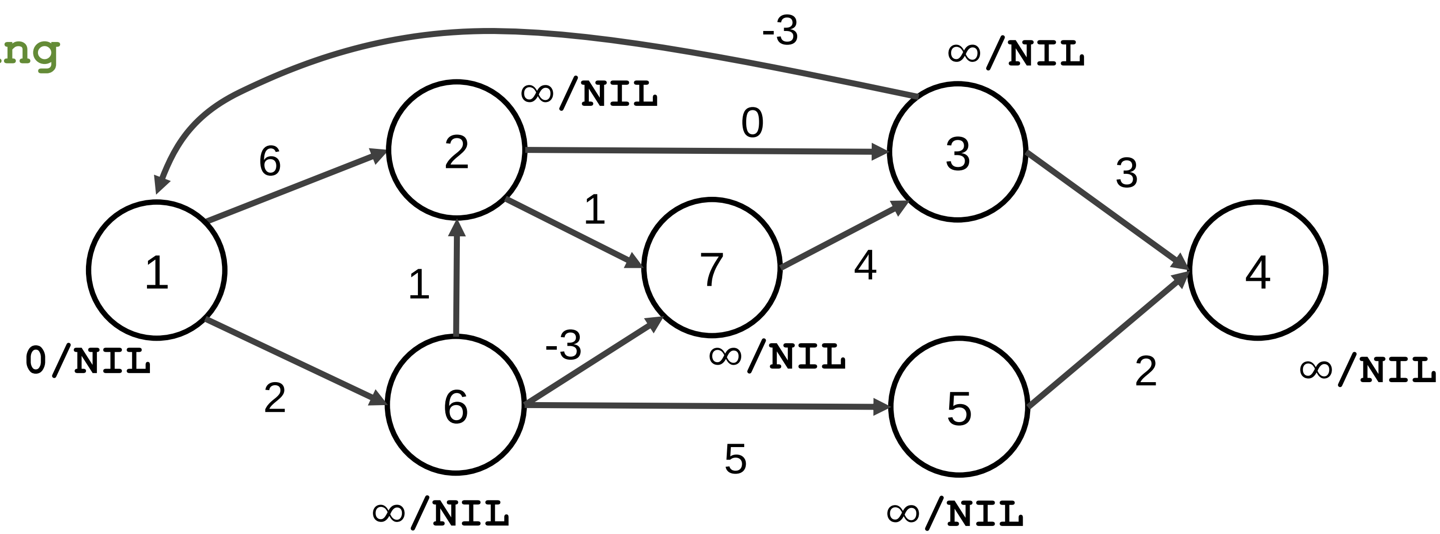
$$\sum_{i=1}^{k} v_i.\mathbf{dist} \leq \sum_{i=1}^{k} v_{i-1}.\mathbf{dist} + \sum_{i=1}^{k} w((v_{i-1}, v_i))$$
$$< \sum_{i=1}^{k} v_{i-1}.\mathbf{dist} = \sum_{i=1}^{k} v_i.\mathbf{dist}$$

Widerspruch.

Bellman-Ford: Beispiel (I)

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

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

Initialisierung für s=1 in 1



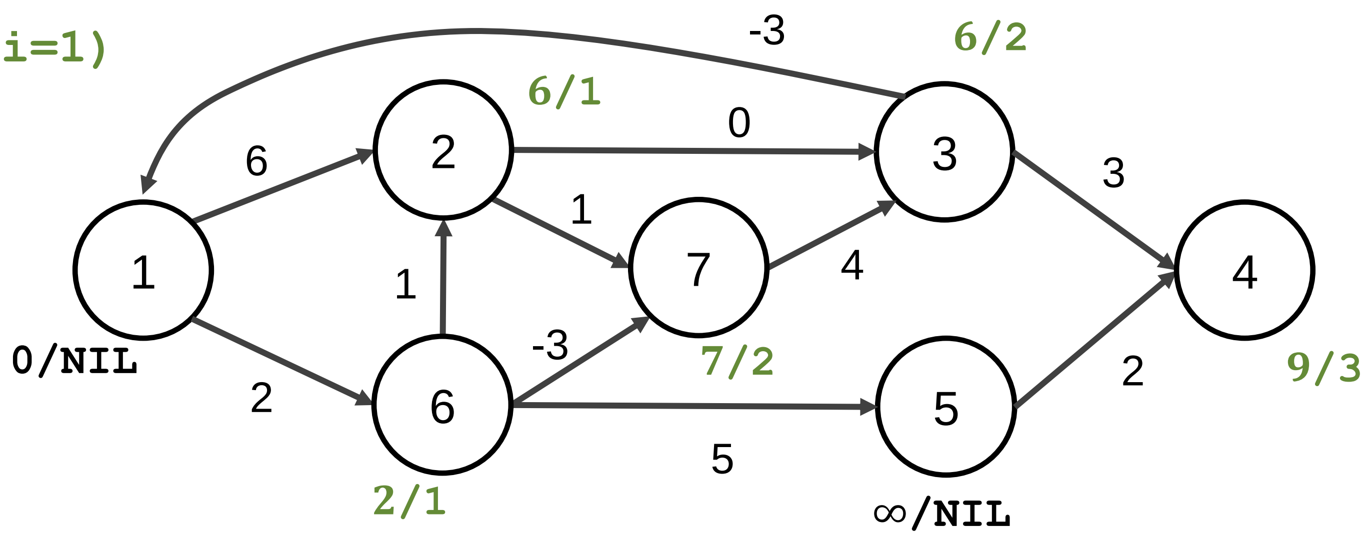
Bellman-Ford: Beispiel (II)

```
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.dist > u.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)

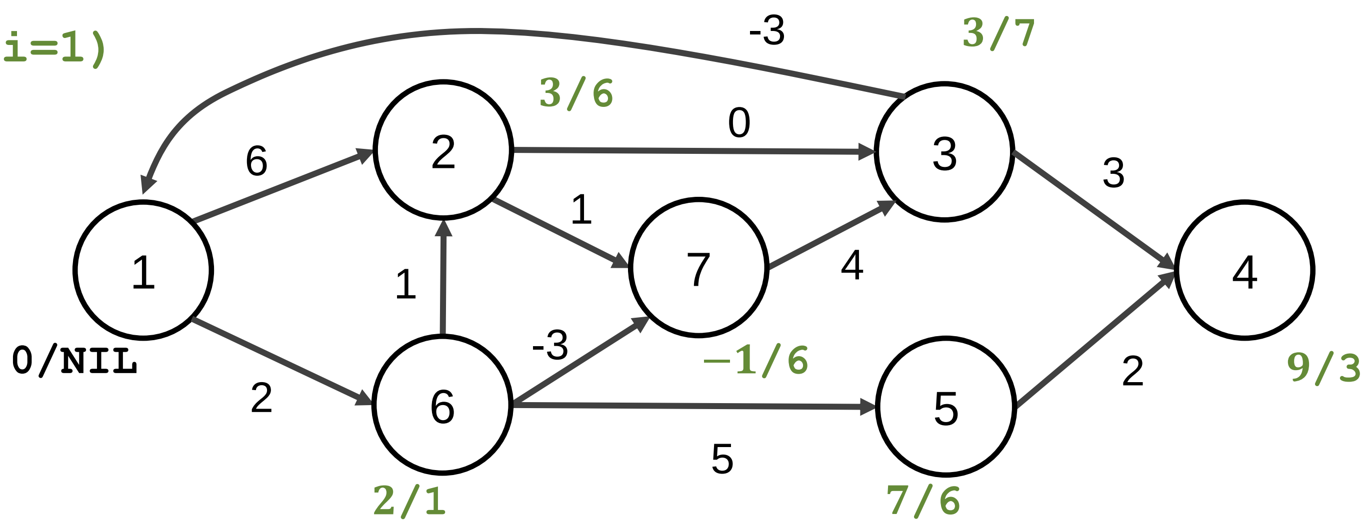


Bellman-Ford: Beispiel (III)

```
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.dist > u.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)

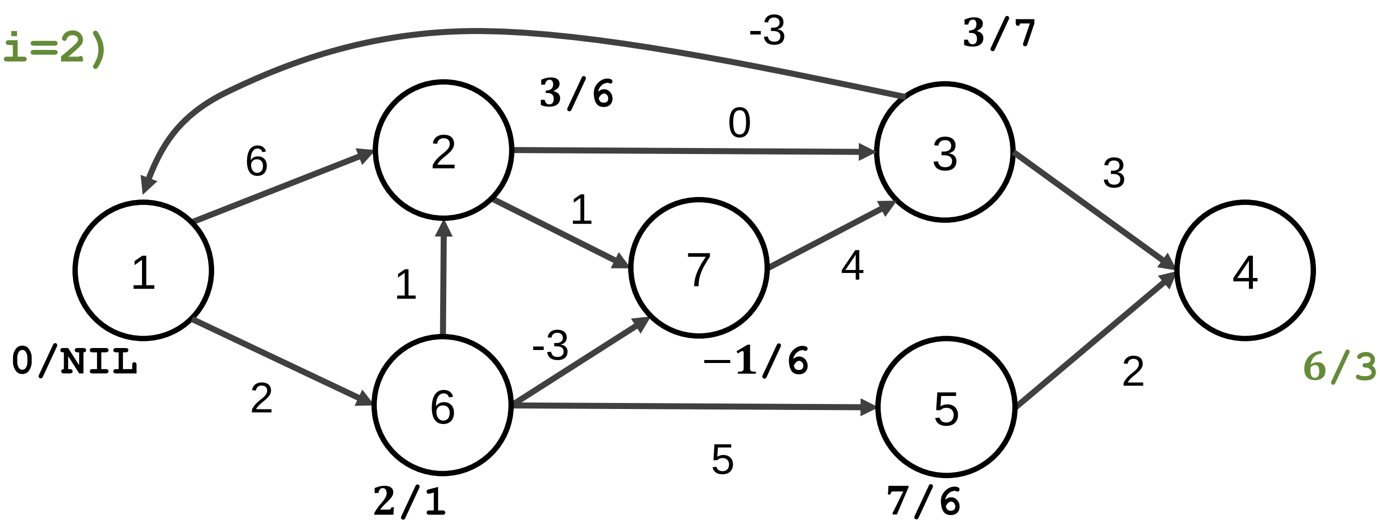
```
Bellman-Ford-SSSP(G,s,w)

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

Keine Änderungen mehr in den folgenden Iterationen

Algorithmus gibt **`true`** zurück

`FOR`–Schleife 2 **`(i=2)`**

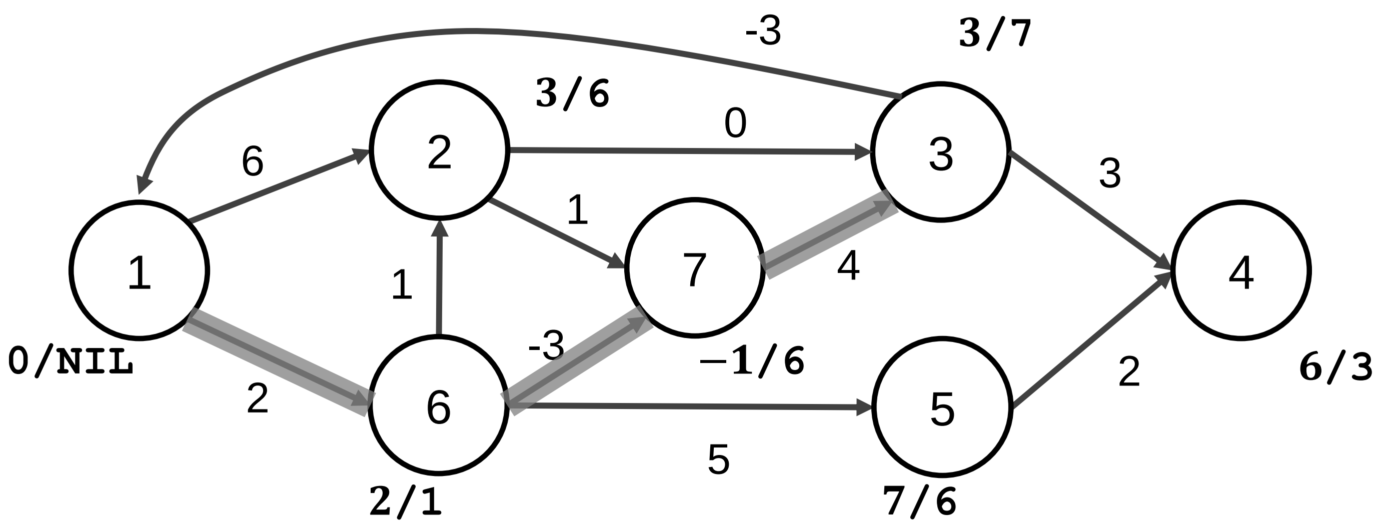
`relax(1,2)`
`relax(1,6)`
`relax(2,3)`
`relax(2,7)`
`relax(3,1)`
`relax(3,4)`
`relax(5,4)`



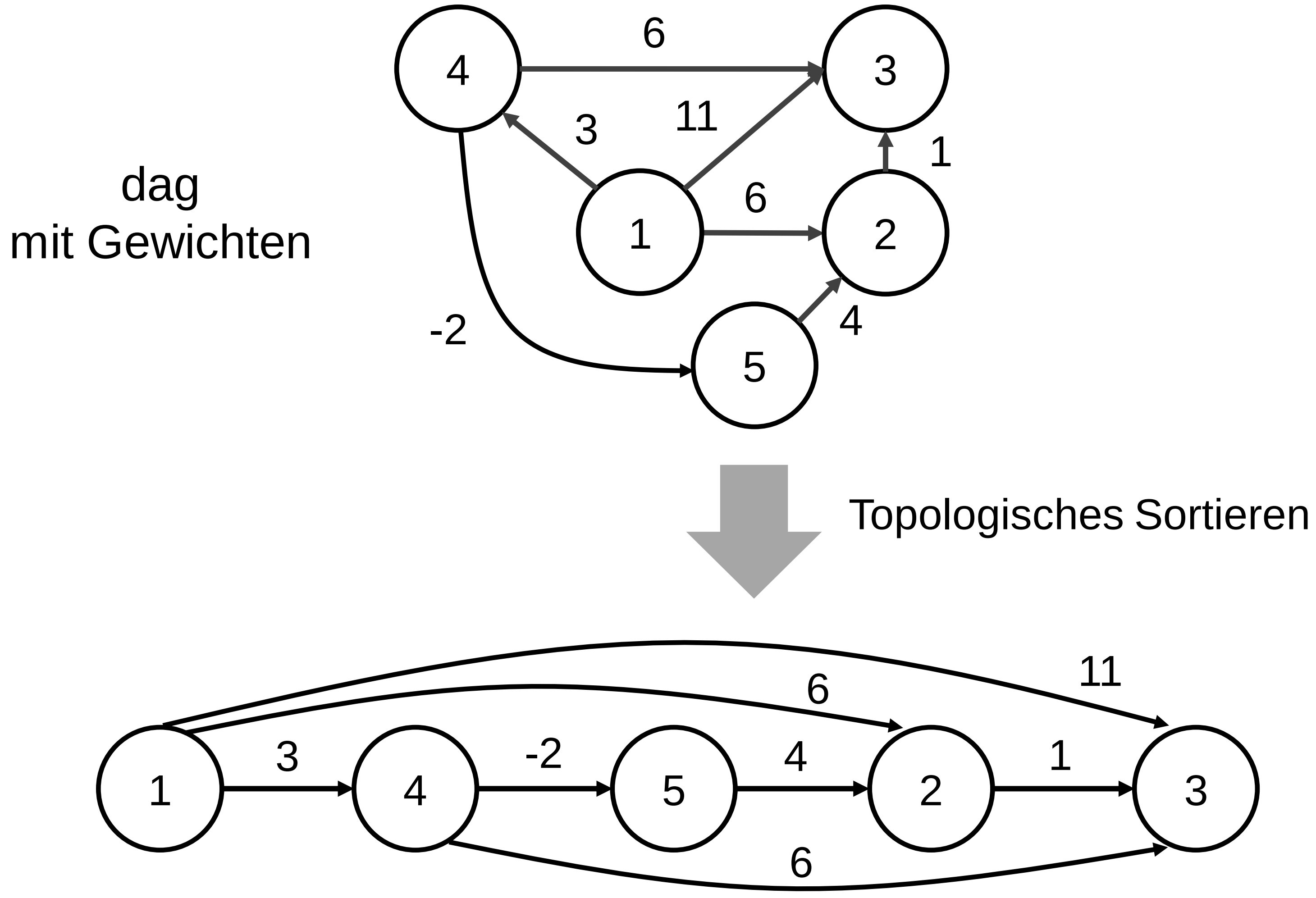
Bellman-Ford: Beispiel (V)

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

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

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

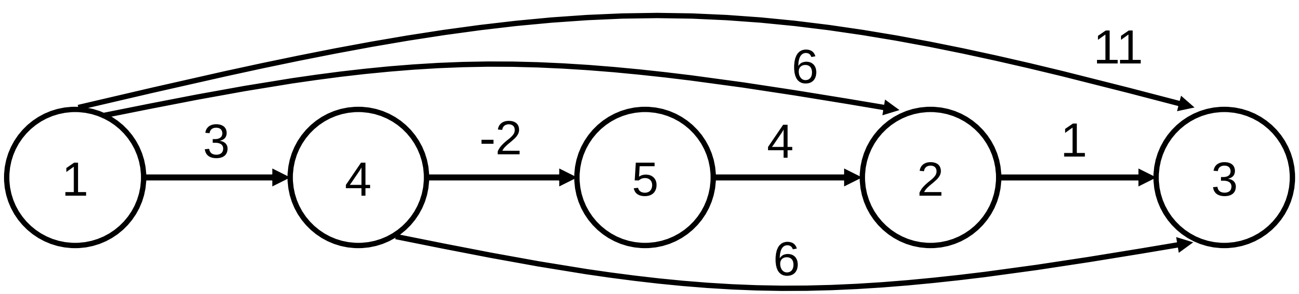


SSSP mittels Topologischer Sortierung



SSSP-Algorithmus für Dags

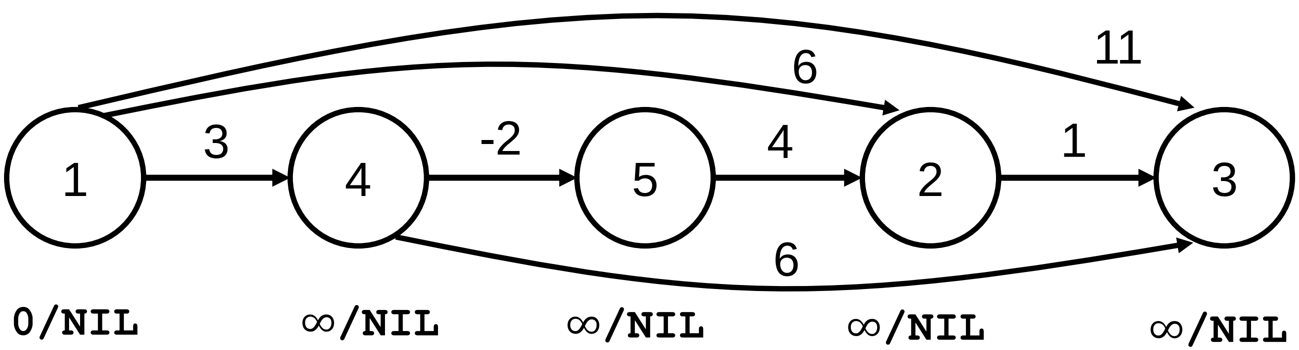
```
TopoSort-SSSP(G,s,w) // G dag

initSSSP(G,s,w);
execute topological sorting
FOREACH u in V in topological order DO
   FOREACH v in adj(u) DO
      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 adj(u) DO
5         relax(G,u,v,w);
```

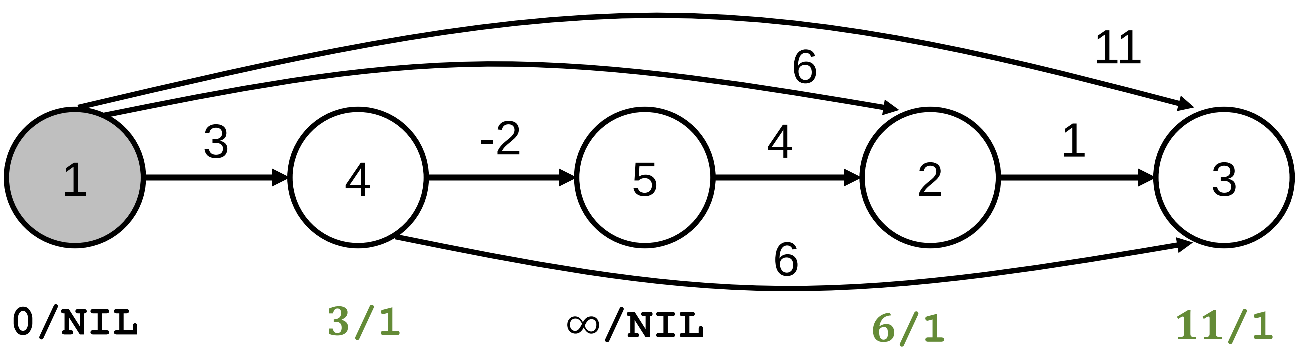
Initialisierung für s=1 in 1
und Sortieren in 2



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 adj(u) DO
5         relax(G,u,v,w);
```

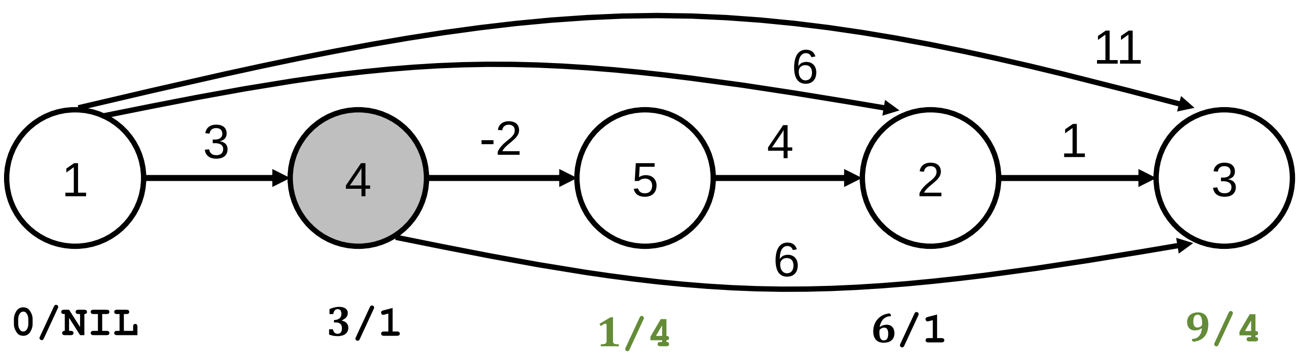
3 FOREACH (u=1)



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 adj(u) DO
5         relax(G,u,v,w);
```

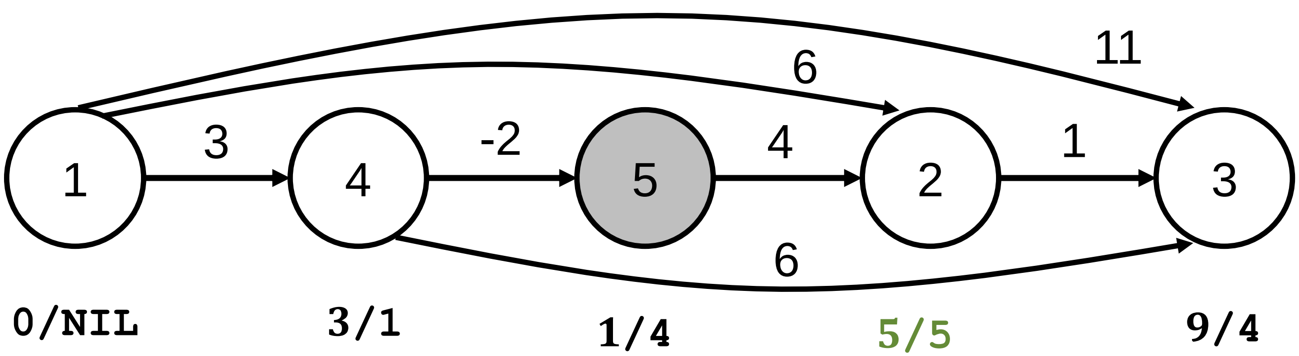
3 FOREACH (u=4)



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 adj(u) DO
5         relax(G,u,v,w);
```

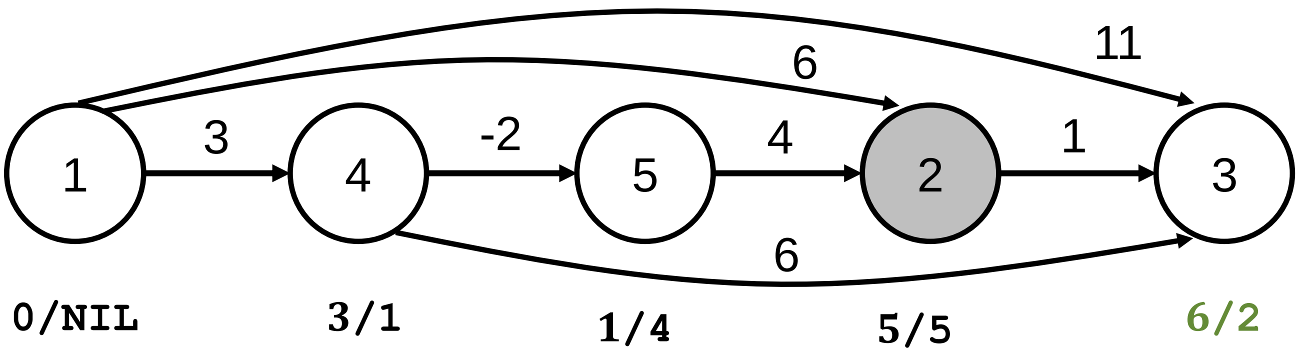
3 FOREACH (u=5)



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 adj(u) DO
5           relax(G,u,v,w);
```

3 FOREACH (u=2)



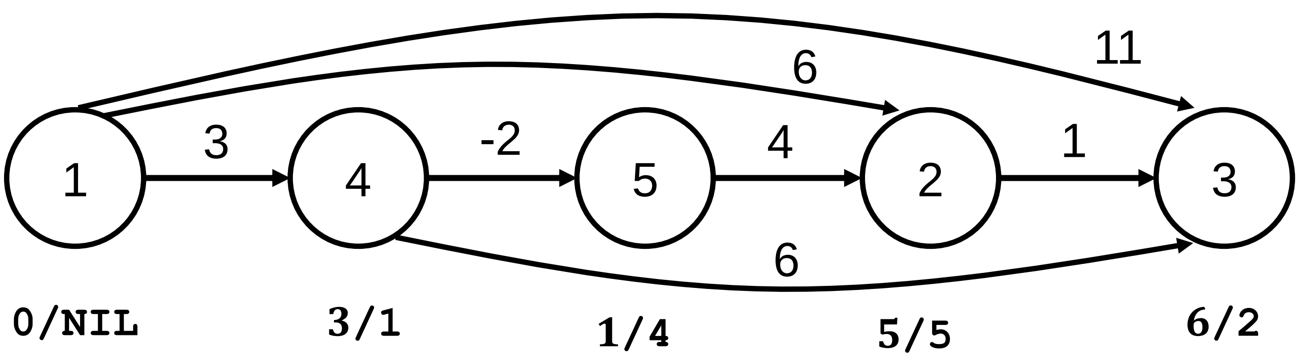
SSSP-Algorithmus für Dags: Korrektheit+Laufzeit

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

initSSSP(G,s,w);
execute topological sorting
FOREACH u in V in topological order DO
   FOREACH v in adj(u) DO
      relax(G,u,v,w);
```

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

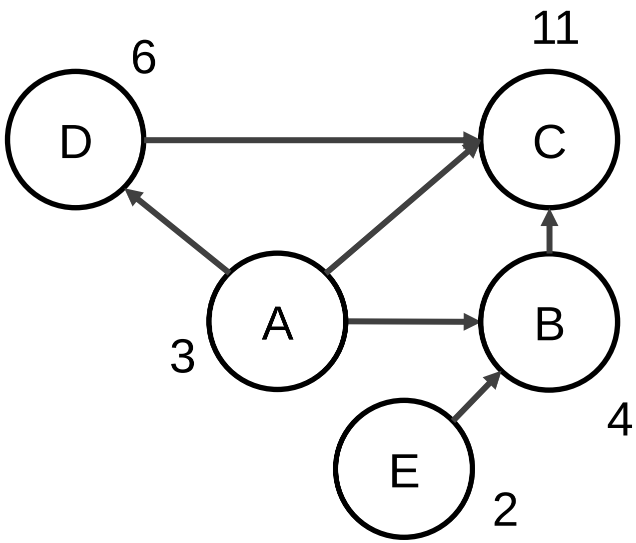
Korrektheit:
Kanten auf einem kürzesten Pfad werden nacheinander „gelockert"

(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 (positive) 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)

initSSSP(G,s,w);
Q=V; //let S=V-Q
WHILE !isEmpty(Q) DO
    u=EXTRACT-MIN(Q); //wrt. dist
    FOREACH v in adj(u) DO
        relax(G,u,v,w);
```

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

(mittels Fibonacci-Heaps)

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

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

FOREACH v in V DO {v.key=∞; v.pred=NIL;}
r.key=-∞; Q=V;
WHILE !isEmpty(Q) DO
  u=EXTRACT-MIN(Q);
  FOREACH v in adj(u) DO
    IF v∈Q and w({u,v})<v.key THEN
       v.key=w({u,v});
       v.pred=u;
```

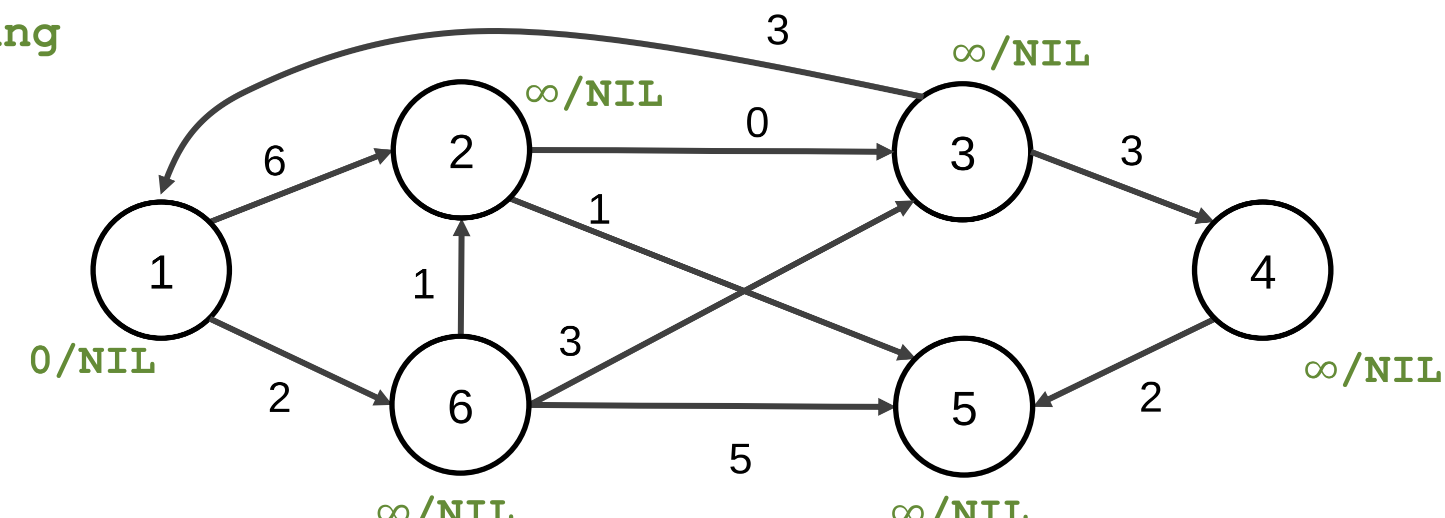
Ähnlichkeit zu Prims Algorithmus

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=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



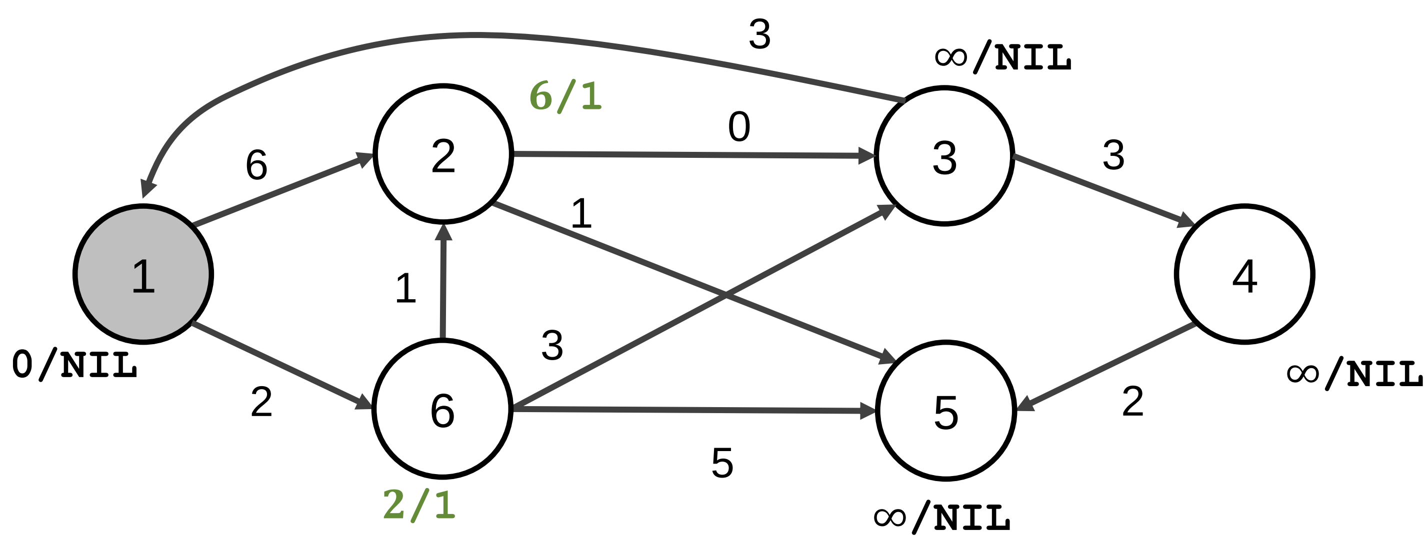
Dijkstra-Algorithmus: Beispiel (II)

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

1   initSSSP(G,s,w);
2   Q=V; //let S=V-Q
3   WHILE !isEmpty(Q) DO
4       u=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)



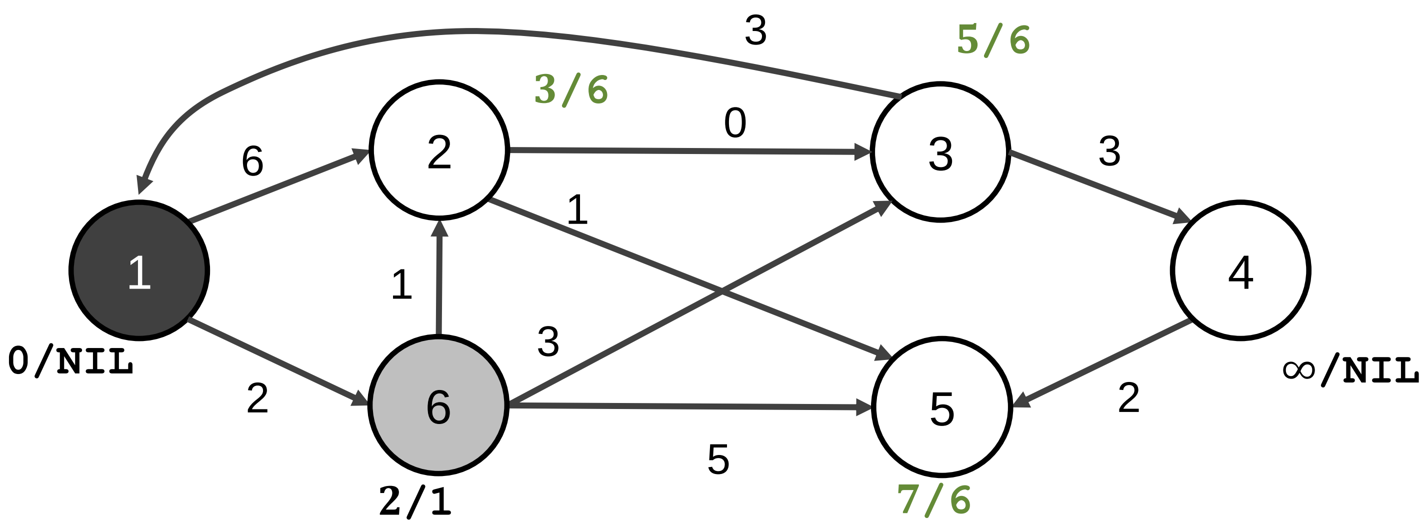
Dijkstra-Algorithmus: Beispiel (III)

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

1   initSSSP(G,s,w);
2   Q=V; //let S=V-Q
3   WHILE !isEmpty(Q) DO
4       u=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)`



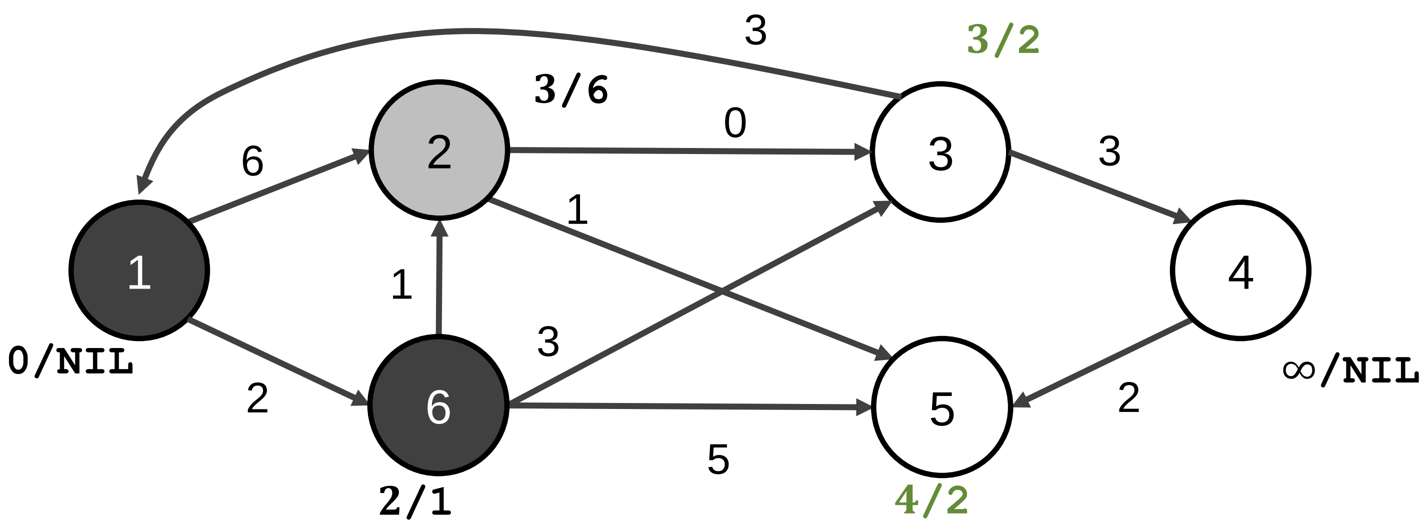
Dijkstra-Algorithmus: Beispiel (IV)

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

1   initSSSP(G,s,w);
2   Q=V; //let S=V-Q
3   WHILE !isEmpty(Q) DO
4       u=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)`



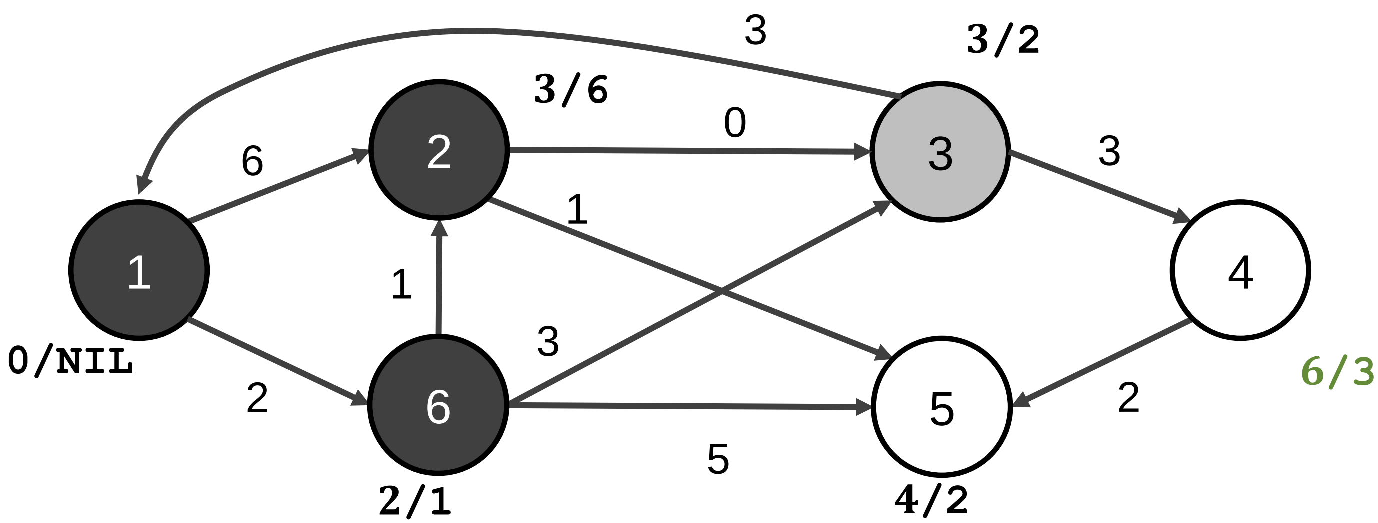
Dijkstra-Algorithmus: Beispiel (V)

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

1   initSSSP(G,s,w);
2   Q=V; //let S=V-Q
3   WHILE !isEmpty(Q) DO
4       u=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)`

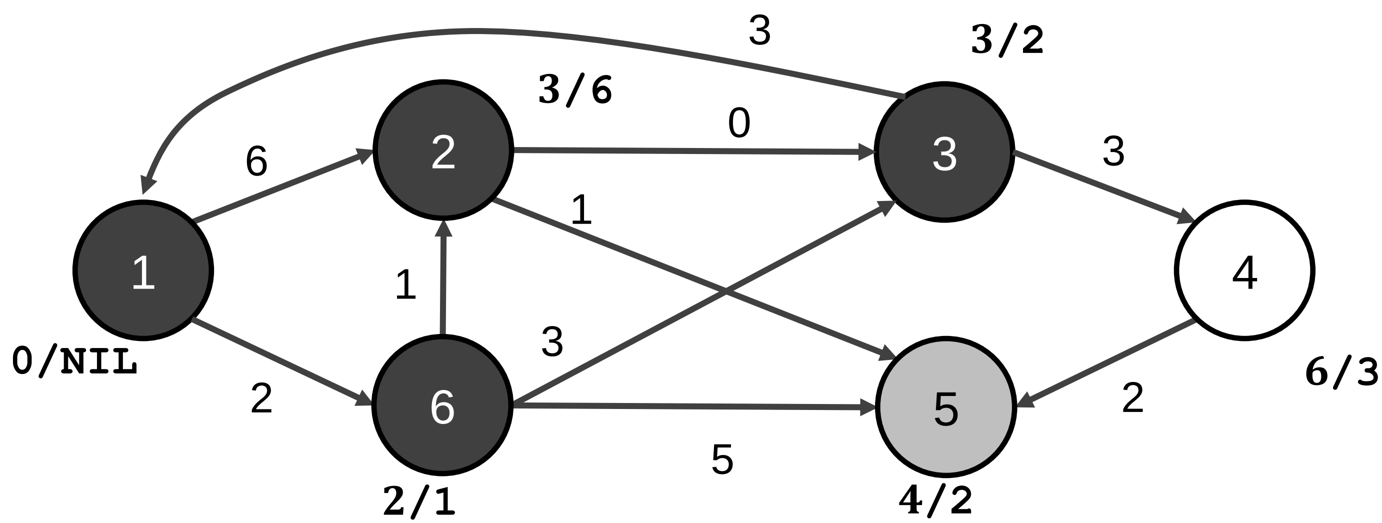


Dijkstra-Algorithmus: Beispiel (VI)

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

1   initSSSP(G,s,w);
2   Q=V; //let S=V-Q
3   WHILE !isEmpty(Q) DO
4       u=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)



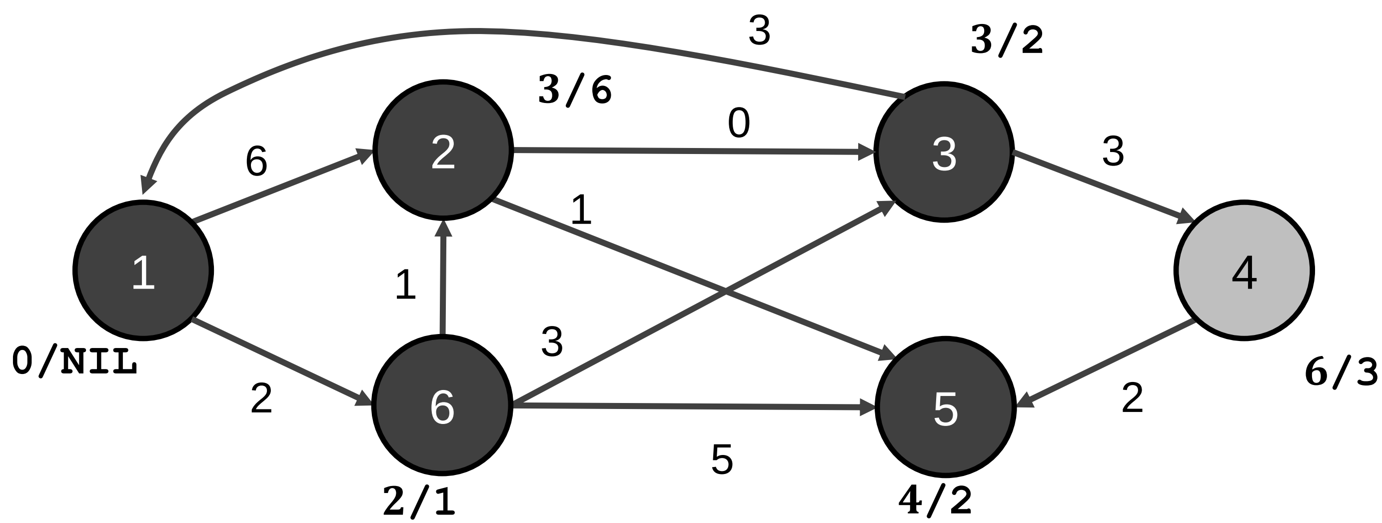
Dijkstra-Algorithmus: Beispiel (VII)

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

1   initSSSP(G,s,w);
2   Q=V; //let S=V-Q
3   WHILE !isEmpty(Q) DO
4       u=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)`

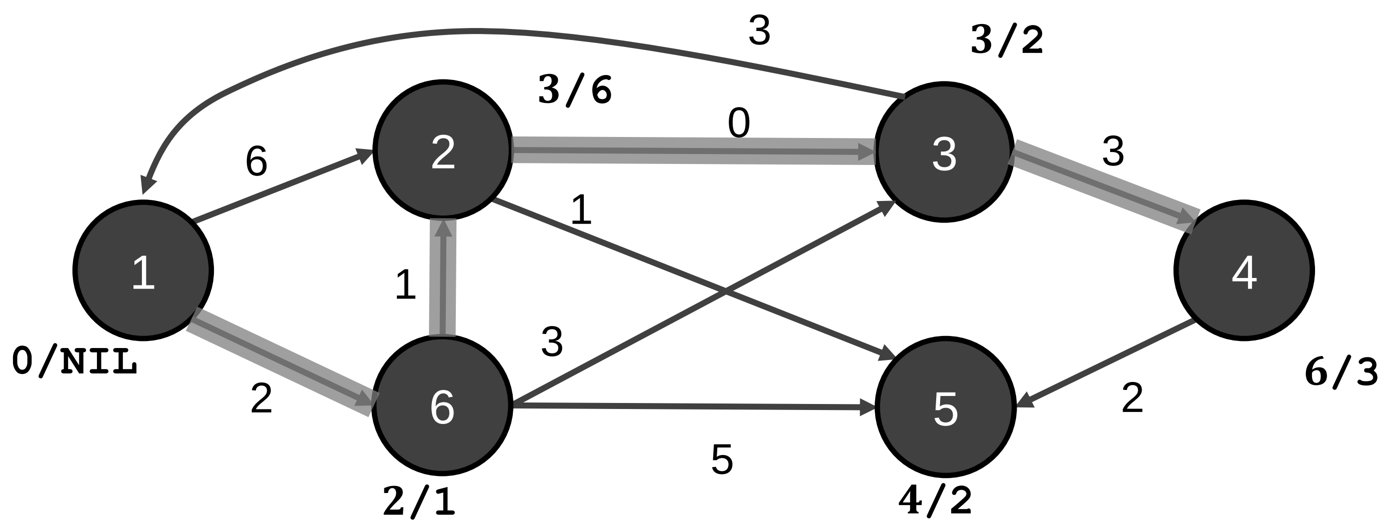


Dijkstra-Algorithmus: Beispiel (VIII)

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

initSSSP(G,s,w);
Q=V; //let S=V-Q
WHILE !isEmpty(Q) DO
    u=EXTRACT-MIN(Q); //wrt. dist
    FOREACH v in adj(u) DO
        relax(G,u,v,w);
```

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

Beispiel: kürzester Weg 1→4



Dijkstra-Algorithmus: Korrektheit (I)

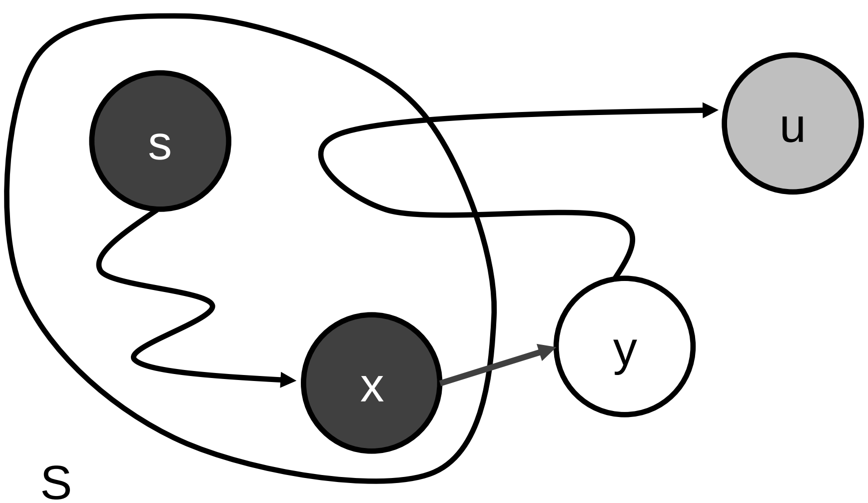
```
Dijkstra-SSSP(G,s,w)

initSSSP(G,s,w);
Q=V; //let S=V-Q
WHILE !isEmpty(Q) DO
    u=EXTRACT-MIN(Q); //wrt. dist
    FOREACH v in adj(u) DO
        relax(G,u,v,w);
```

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

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

Insbesondere **u**$\neq$**s** und $S \neq \emptyset$ da **s.dist=0** korrekt

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

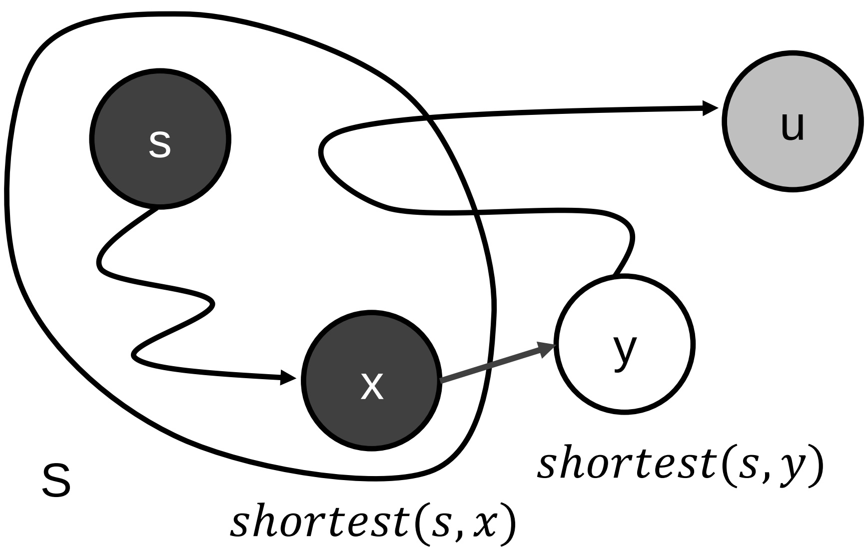


Dijkstra-Algorithmus: Korrektheit (II)

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

initSSSP(G,s,w);
Q=V; //let S=V-Q
WHILE !isEmpty(Q) DO
    u=EXTRACT-MIN(Q); //wrt. dist
    FOREACH v in adj(u) DO
        relax(G,u,v,w);
```

Für jeden betrachteten Knoten **`u`** in der **`WHILE`**-Schleife gilt:
`u.dist`$= shortest(s,u)$

Angenommen, **`u`** wäre erster Knoten, bei dem nicht der Fall



Es gilt **`x.dist`**$= shortest(s,x)$, 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.dist`**$= shortest(s,y)$

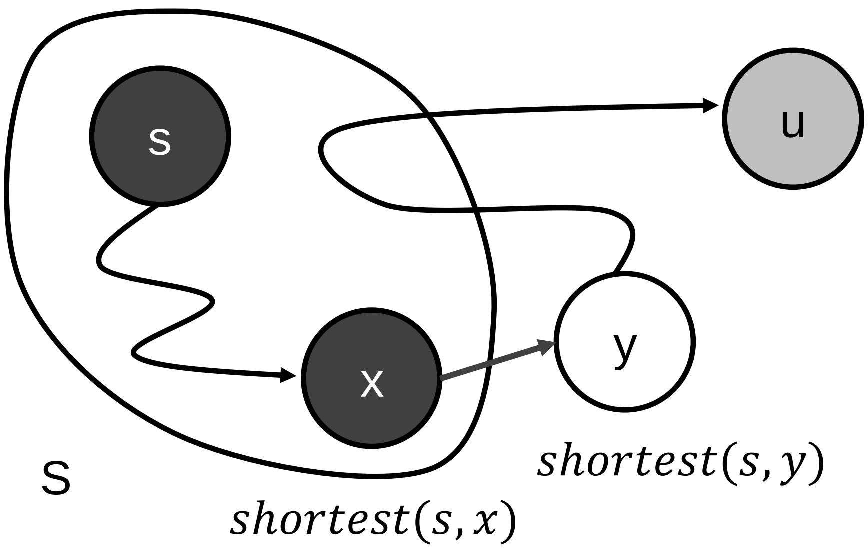
Dijkstra-Algorithmus: Korrektheit (III)

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

initSSSP(G,s,w);
Q=V; //let S=V-Q
WHILE !isEmpty(Q) DO
    u=EXTRACT-MIN(Q); //wrt. dist
    FOREACH v in adj(u) DO
        relax(G,u,v,w);
```

Für jeden betrachteten Knoten **`u`** in der **`WHILE`**-Schleife gilt:

$$\texttt{u.dist} = shortest(s,u)$$

Angenommen, **`u`** wäre erster Knoten, bei dem nicht der Fall



Da nur **nicht-negative Kantengewichte** gilt

$$shortest(s,y) \leq shortest(s,u),$$

und damit

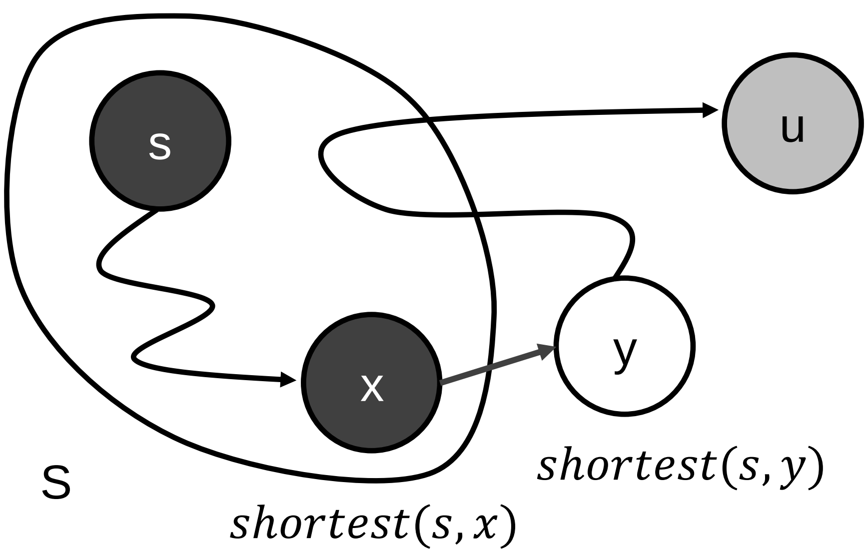
$$\texttt{y.dist} = shortest(s,y) \leq shortest(s,u) \leq \texttt{u.dist}$$

Dijkstra-Algorithmus: Korrektheit (IV)

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

   initSSSP(G,s,w);
   Q=V; //let S=V-Q
   WHILE !isEmpty(Q) DO
      u=EXTRACT-MIN(Q); //wrt. dist
      FOREACH v in adj(u) DO
         relax(G,u,v,w);
```

Für jeden betrachteten Knoten **`u`** in der **`WHILE`**-Schleife gilt:
`u.dist`$= shortest(s,u)$

Angenommen, **`u`** wäre erster Knoten, bei dem nicht der Fall



Andererseits wurde **`u`** vor **`y`** für S ausgewählt, also

`u.dist` $\leq$ **`y.dist`** und

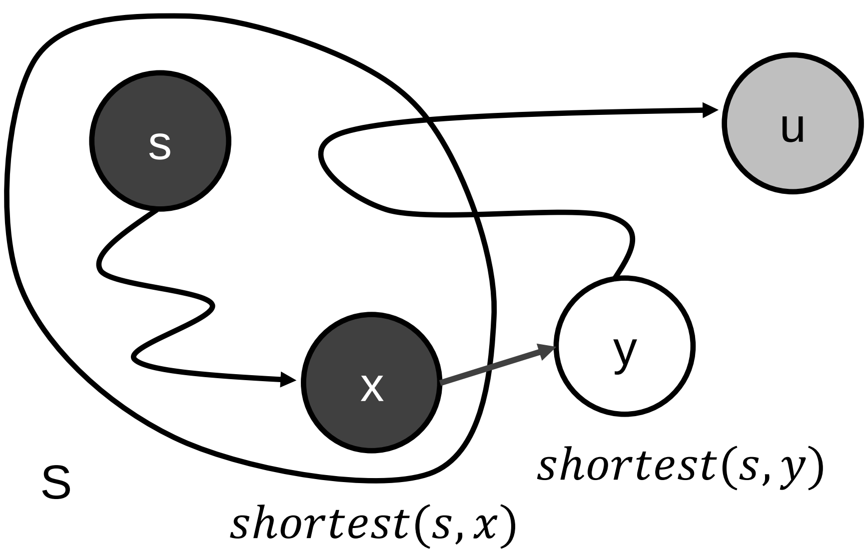
`y.dist` $= shortest(s,y)$
$\leq shortest(s,u) \leq$ **`u.dist`**

Dijkstra-Algorithmus: Korrektheit (V)

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

initSSSP(G,s,w);
Q=V; //let S=V-Q
WHILE !isEmpty(Q) DO
    u=EXTRACT-MIN(Q); //wrt. dist
    FOREACH v in adj(u) DO
        relax(G,u,v,w);
```

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

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



Folglich
y.dist $= shortest(s,y)$
$=\ shortest(s,u) =$ **u.dist**, da

u.dist $\leq$ **y.dist** und

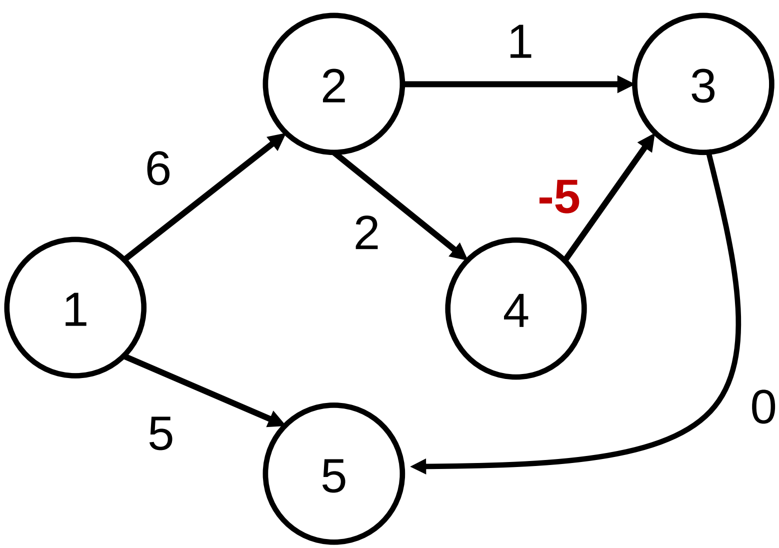
y.dist $= shortest(s,y)$
$\leq shortest(s,u)\ \leq$ **u.dist**

Dijkstra-Algorithmus und negative Kantengewichte (I)

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

initSSSP(G,s,w);
Q=V; //let S=V-Q
WHILE !isEmpty(Q) DO
    u=EXTRACT-MIN(Q);
    FOREACH v in adj(u) DO
        relax(G,u,v,w);
```

kürzeste Weg 1→5 via
1→ 2 → 4 → 3 → 5
mit Gewicht 6+2-5+0=3

Dijkstra-Algorithmus für s=1
gibt aber 1→5 an



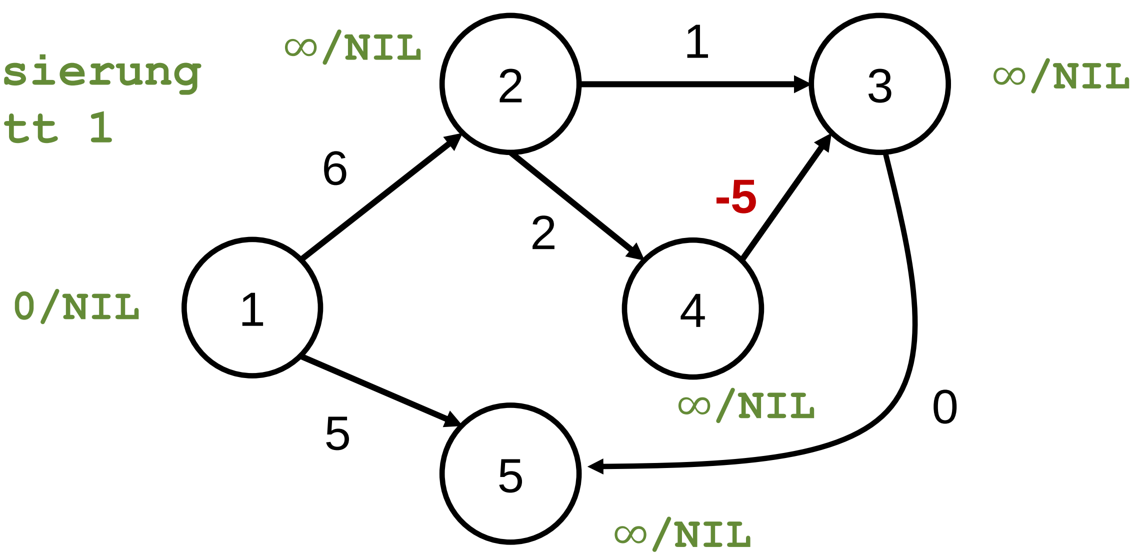
Dijkstra-Algorithmus und negative Kantengewichte (II)

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

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

kürzeste Weg 1→5 via
1→ 2 → 4 → 3 → 5
mit Gewicht 6+2-5+0=3

Dijkstra-Algorithmus für s=1
gibt aber 1→5 an

Initialisierung in Schritt 1



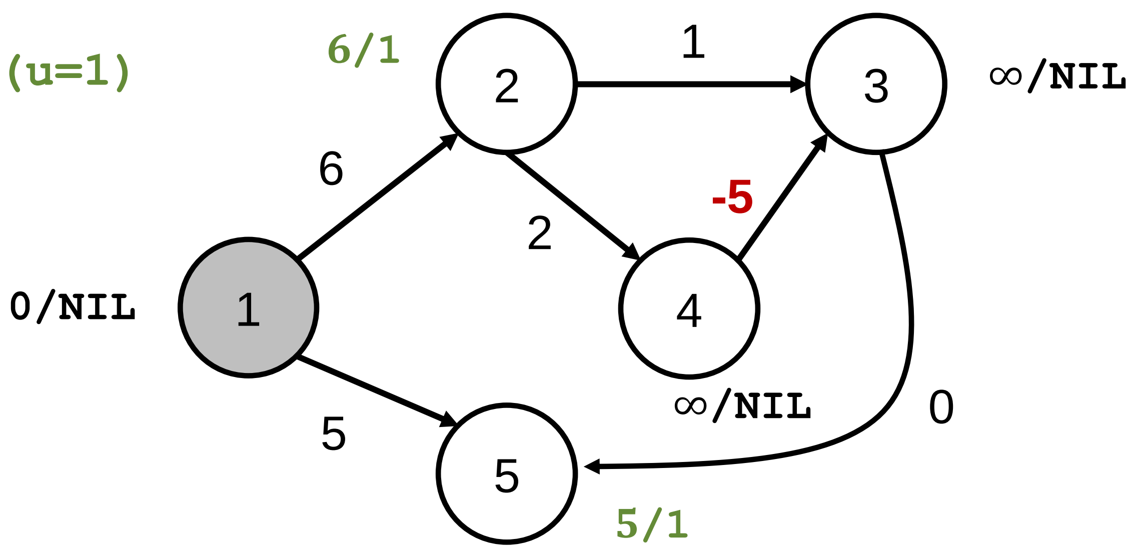
Dijkstra-Algorithmus und negative Kantengewichte (III)

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

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

kürzeste Weg 1→5 via
1→ 2 → 4 → 3 → 5
mit Gewicht 6+2-5+0=3

Dijkstra-Algorithmus für s=1
gibt aber 1→5 an

3 WHILE (u=1)



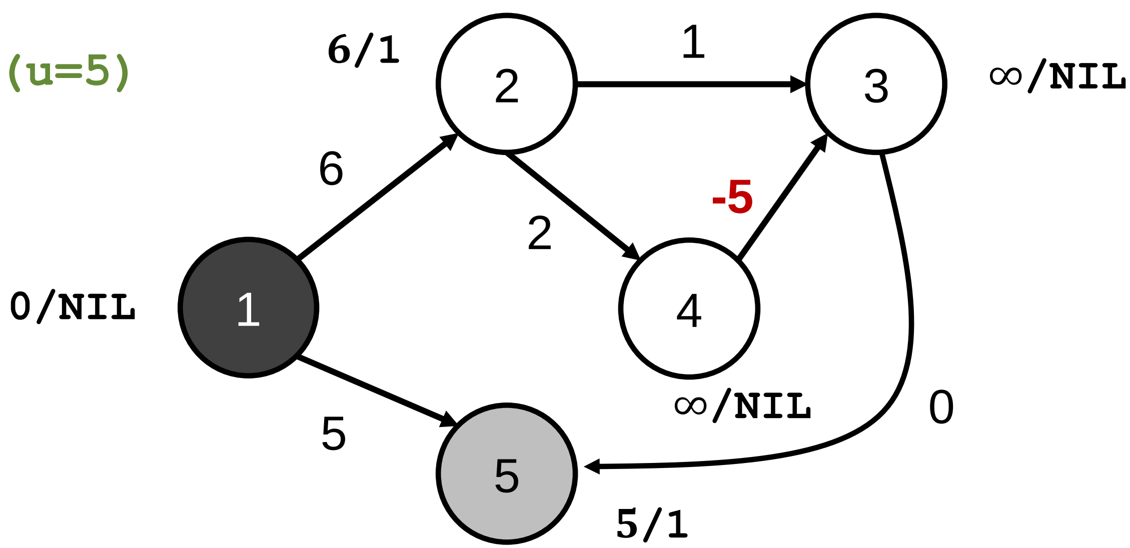
Dijkstra-Algorithmus und negative Kantengewichte (IV)

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

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

kürzeste Weg 1→5 via
1→ 2 → 4 → 3 → 5
mit Gewicht 6+2-5+0=3

Dijkstra-Algorithmus für s=1
gibt aber 1→5 an

3 WHILE (u=5)



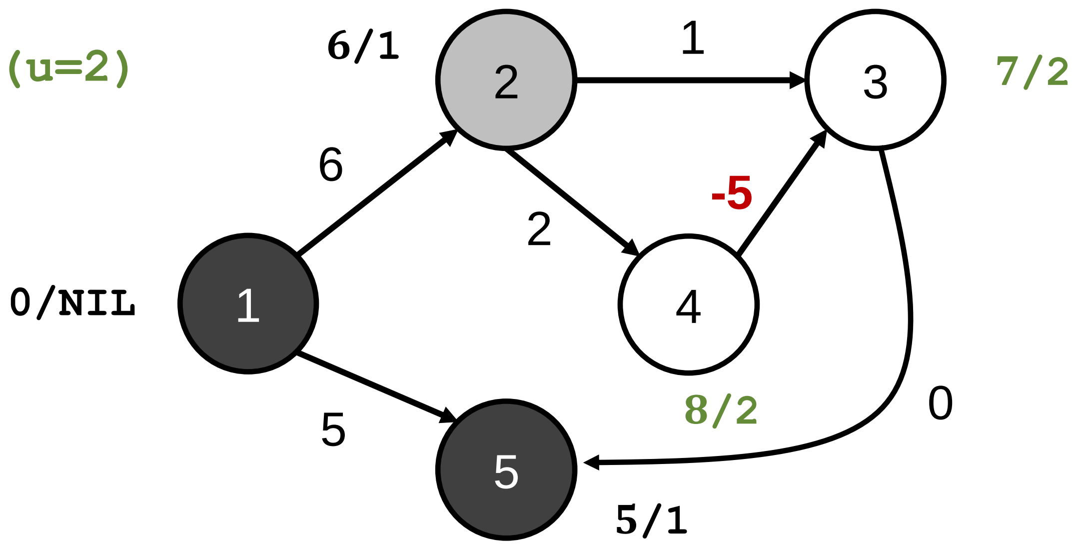
Dijkstra-Algorithmus und negative Kantengewichte (V)

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

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

kürzeste Weg 1→5 via
1→ 2 → 4 → 3 → 5
mit Gewicht 6+2-5+0=3

Dijkstra-Algorithmus für s=1
gibt aber 1→5 an

3 WHILE (u=2)



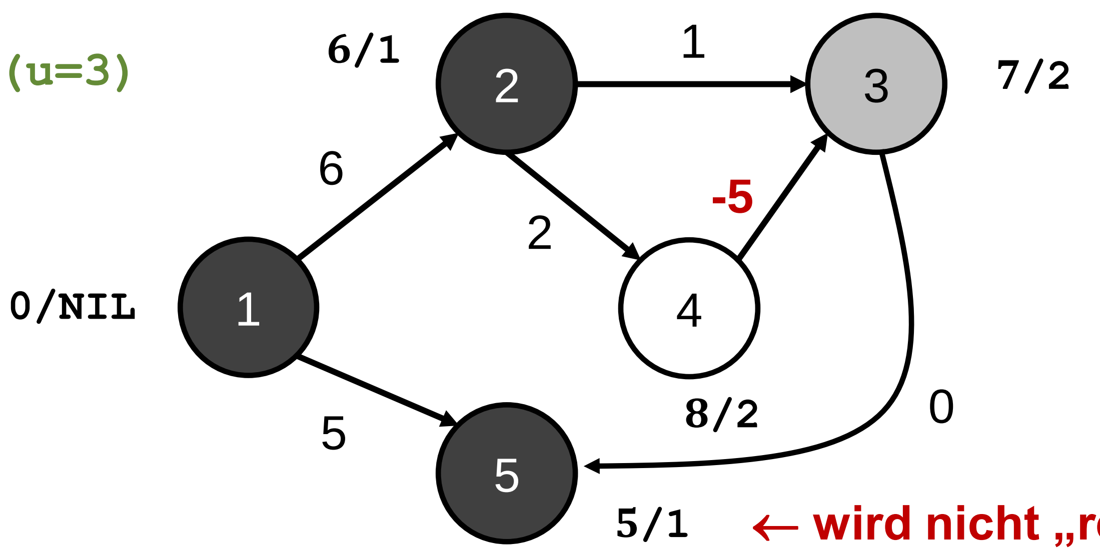
Dijkstra-Algorithmus und negative Kantengewichte (VI)

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

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

kürzeste Weg 1→5 via
1→ 2 → 4 → 3 → 5
mit Gewicht 6+2-5+0=3

Dijkstra-Algorithmus für s=1
gibt aber 1→5 an

`3 WHILE (u=3)`



← **wird nicht „relaxed“**

Der im Augenblick nicht erfasste Weg von 1→3 über 4 wird später zum kürzeren Weg

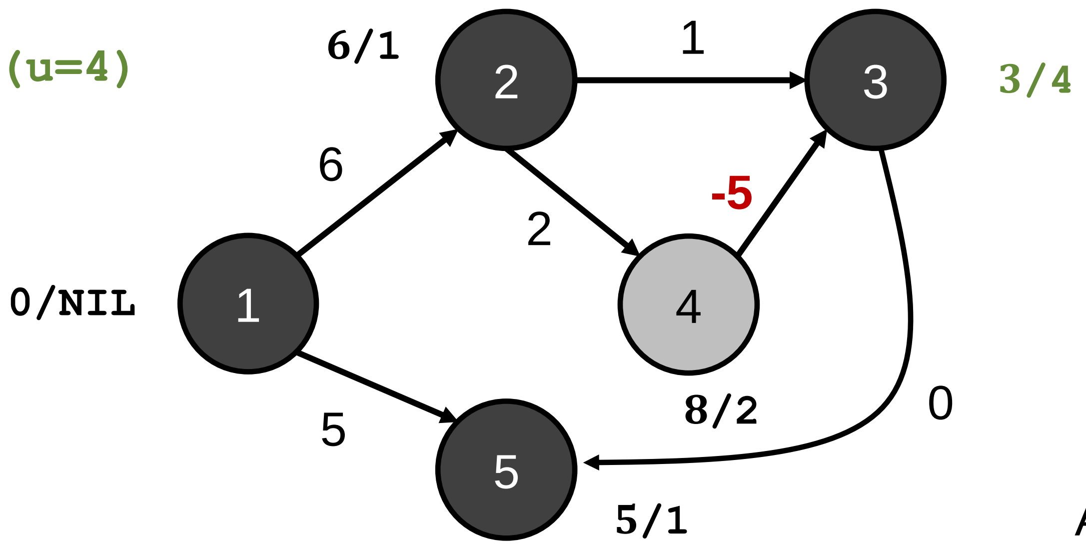
Dijkstra-Algorithmus und negative Kantengewichte (VII)

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

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

kürzeste Weg 1→5 via
1→ 2 → 4 → 3 → 5
mit Gewicht 6+2-5+0=3

Dijkstra-Algorithmus für s=1
gibt aber 1→5 an

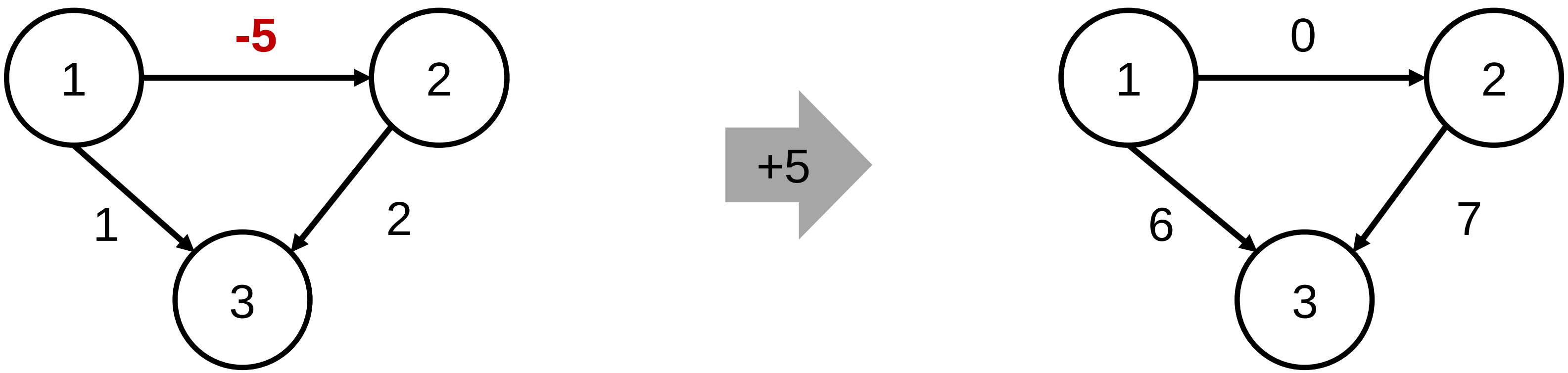
3 WHILE (u=4)



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

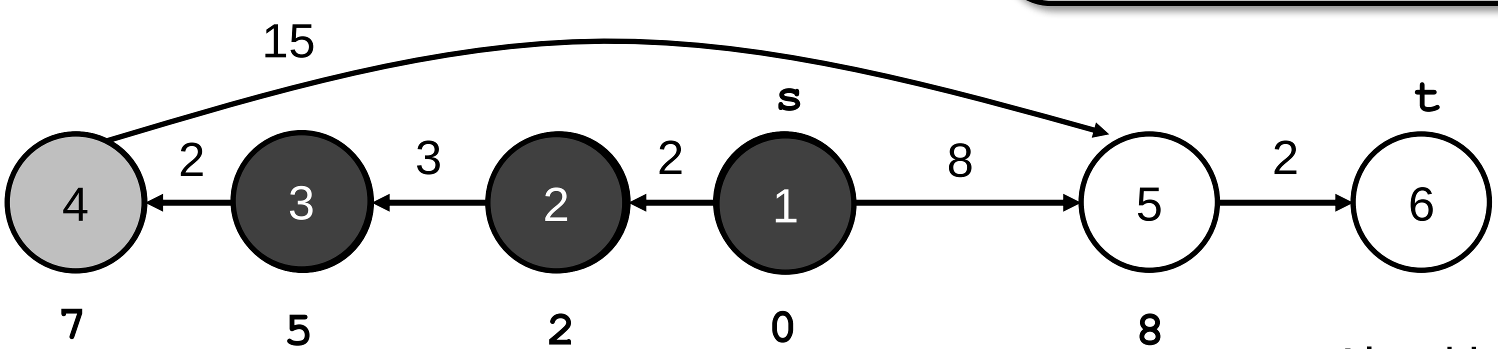
A*-Algorithmus (I)

Hart, Nilsson, Raphael (1968)

Spezialfall: suche kürzesten Weg von **`s`** zu einem Ziel **`t`**

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

initSSSP(G,s,w);
Q=V; //let S=V-Q
WHILE !isEmpty(Q) DO
    u=EXTRACT-MIN(Q);
    FOREACH v in adj(u) DO
        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)

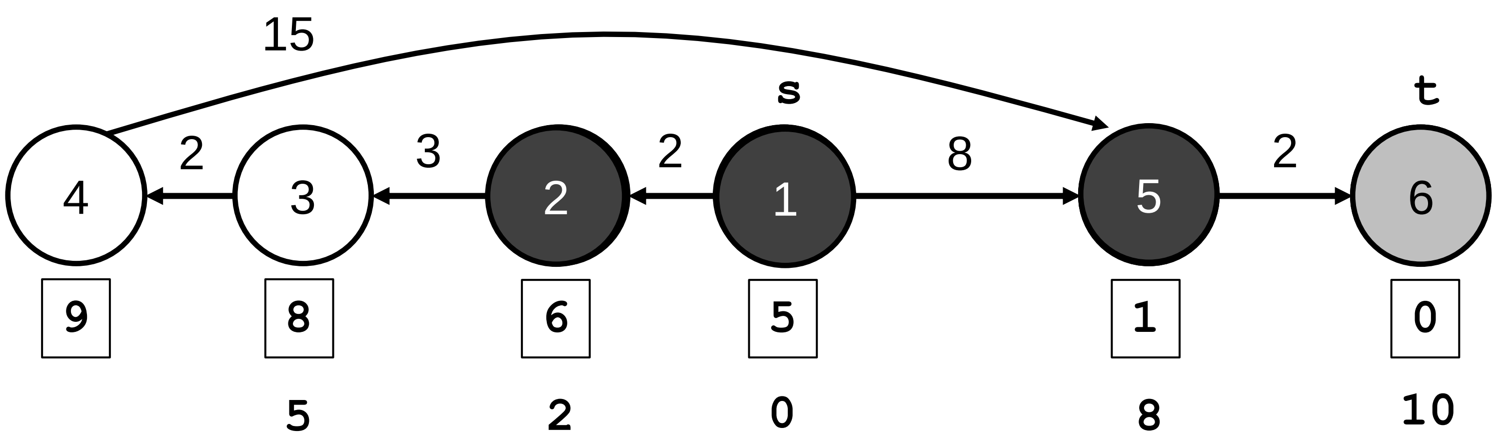
(nicht-negative Kantengewichte)

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

```
A*(G,s,t,w)
  init(G,s,t,w);
  Q=V; //let S=V-Q
  WHILE !isEmpty(Q) DO
     u=EXTRACT-MIN(Q);
     IF u==t THEN break;
     FOREACH v in adj(u) DO
        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:

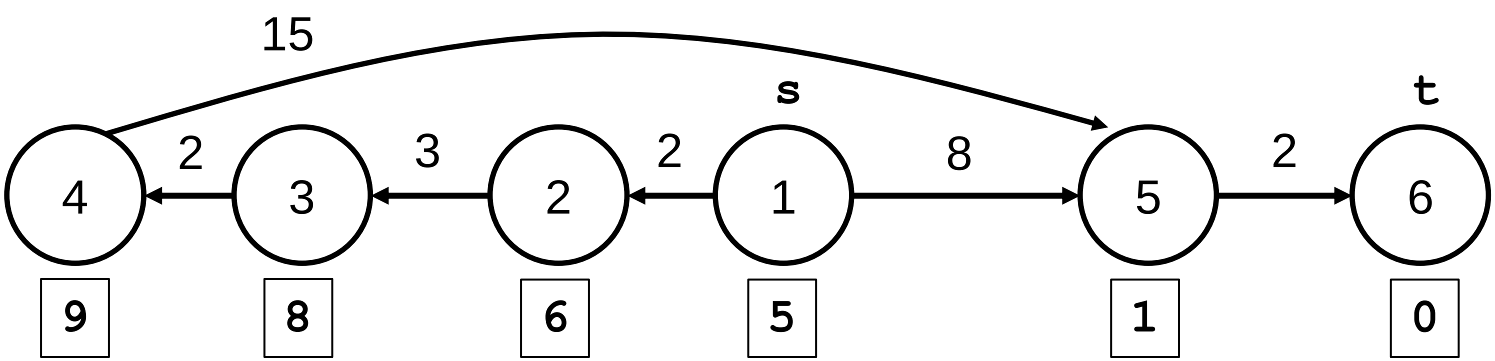
$$\mathbf{u.heur} \ \leq \ \mathit{shortest}(\mathbf{u},\mathbf{t})$$

(insb. `t.heur==0`)

und

2. Heuristik ist monoton, d.h. für alle $(\mathbf{u},\mathbf{v}) \epsilon E$ gilt:

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



A*-Algorithmus (IV)

(nicht-negative Kantengewichte)

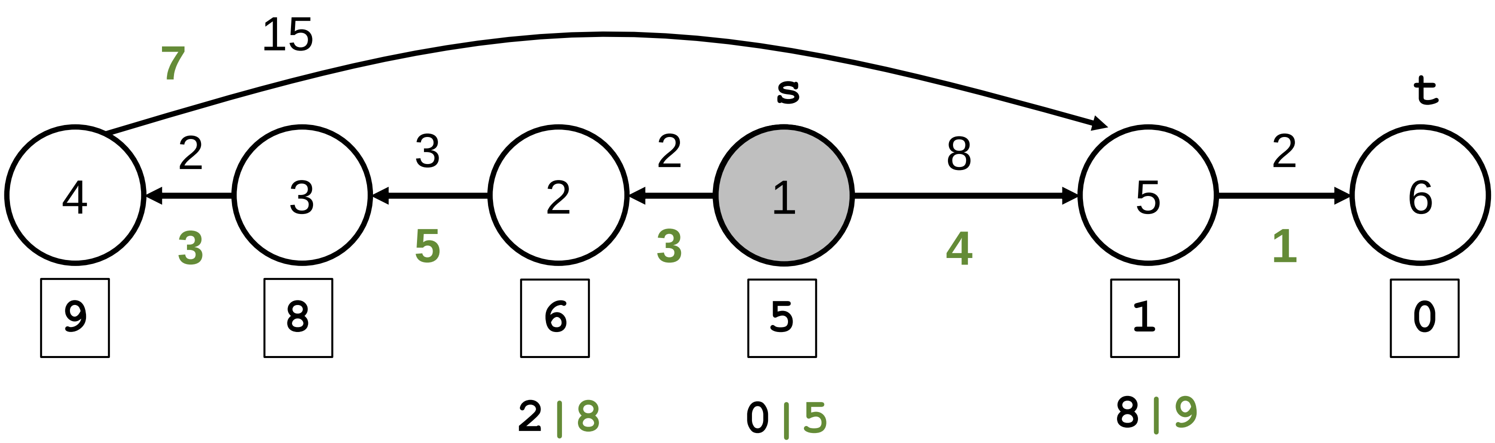
„Dijkstra ist A* mit Heuristik 0“

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

Kantengewichte sind nicht-negativ wegen Monotonie

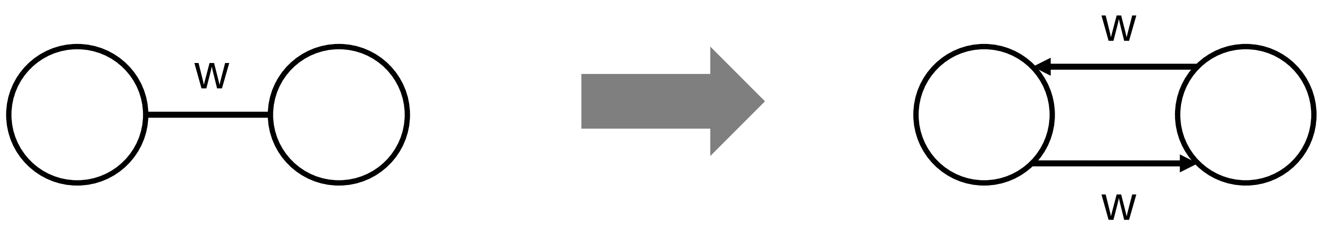
A* und Dijsktra wählen dann jeweils gleichen Knoten, da `u.dist+u.heur==u.dist`,

und am Ende `t.dist==t.dist`



Ungerichtete Graphen

Bellman-Ford und Dijkstra für ungerichtete Graphen:
Verwende implizite Darstellung als gerichteter Graph ($\{u, v\}$ entspricht (u, v) und (v, u))



Bellman-Ford:
Wenn $w < 0$, entstehen negative Zyklen

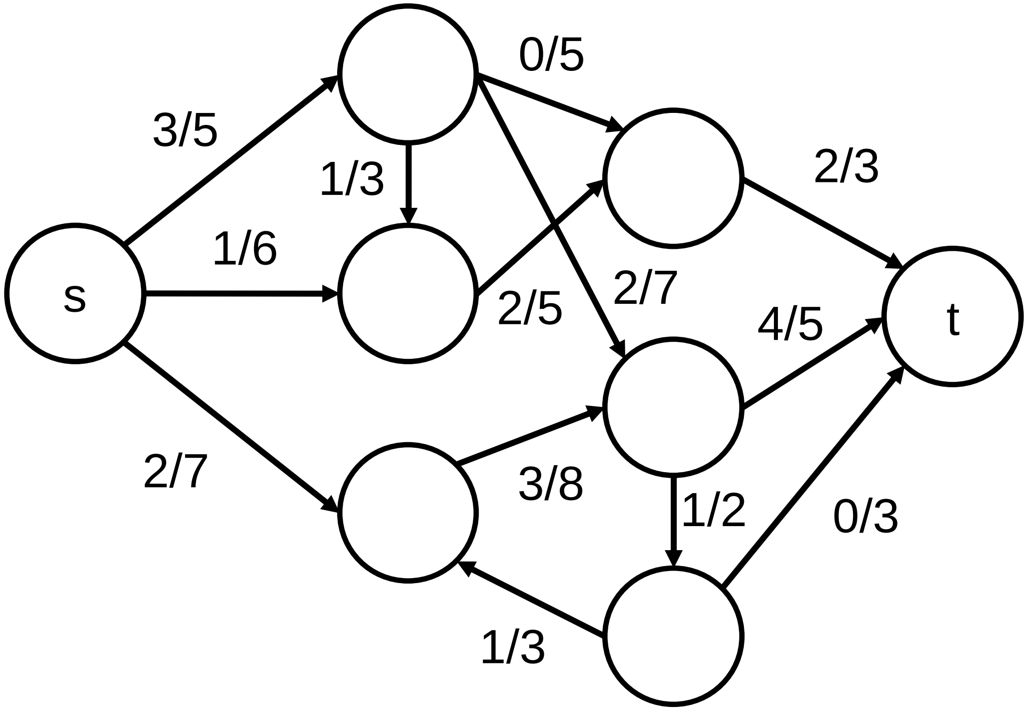
(Wenn Annahme, dass nur nicht-negative Gewichte, dann besser gleich Dijkstra verwenden)

Dijkstra:
alle Gewichte bleiben nicht-negativ

(kann mit dieser Darstellung also direkt ausgeführt werden)

Maximaler Fluss in Graphen

Netzwerkflüsse: Idee



Kanten haben (aktuellen) Flusswert und (maximale) Kapazität

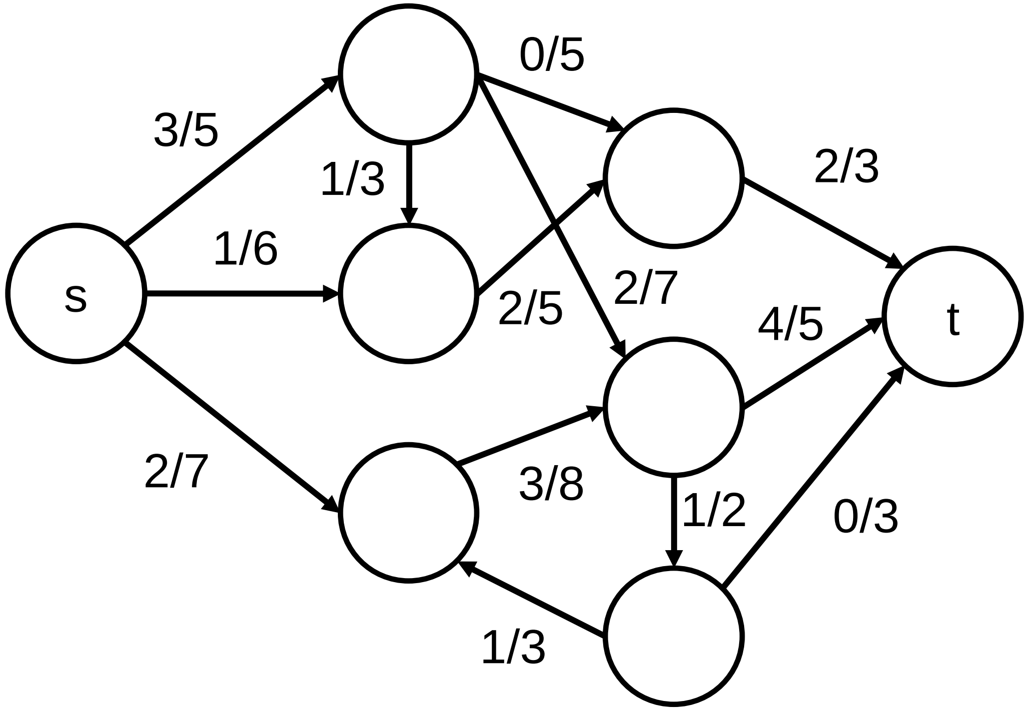
Jeder Knoten außer s und t hat gleichen eingehenden und ausgehenden Fluss

Ziel:
Finde maximalen Fluss von s nach t

Netzwerkflüsse

Ein **Flussnetzwerk** ist ein gewichteter, gerichteter Graph $G = (V, E)$ mit Kapazität(-sgewicht) c, so dass $\mathrm{c(u,v)} \geq 0$ für $\mathrm{(u,v)} \in E$ und $\mathrm{c(u,v)} = 0$ für $\mathrm{(u,v)} \notin E$, 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.

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



Ein **Fluss** $f: V \times V \to \mathbb{R}$ 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 $u \in V - \{s, t\}$:

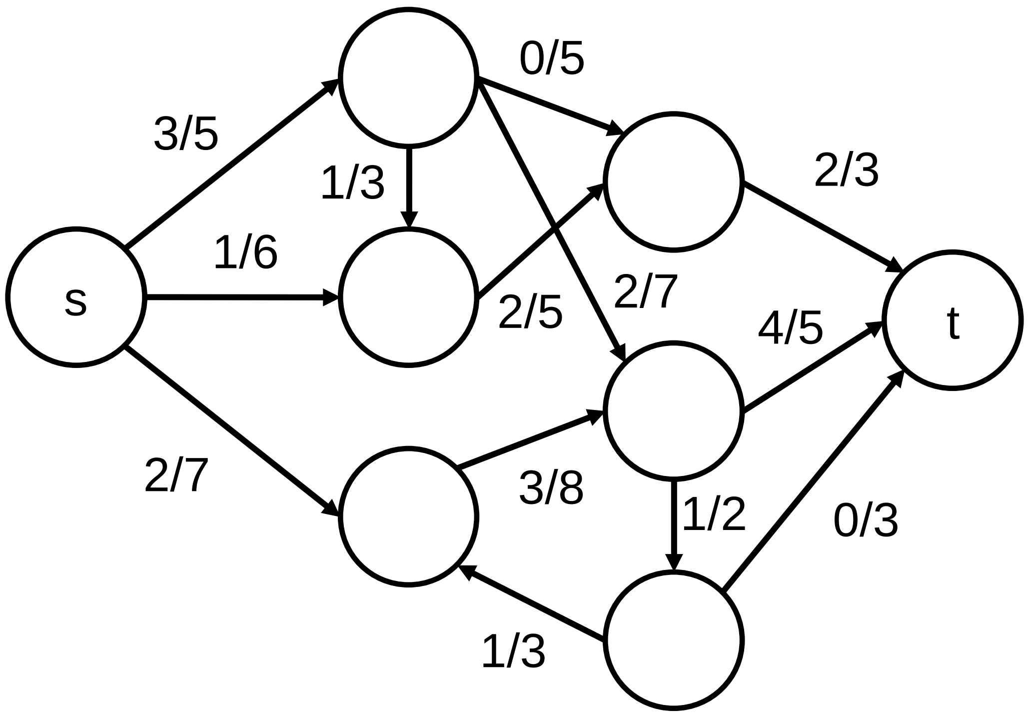
$$\sum_{v \in V} f(u,v) = \sum_{v \in V} f(v,u)$$

Maximale Flüsse

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

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

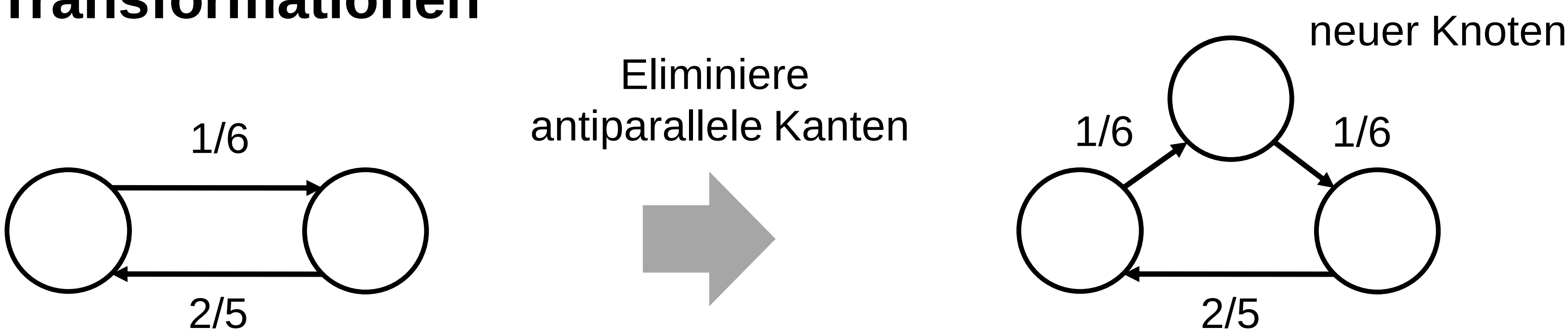
Beispiel: $|f| = 6$, aber nicht maximal, da z.B. noch $+1$ über obere Kanten

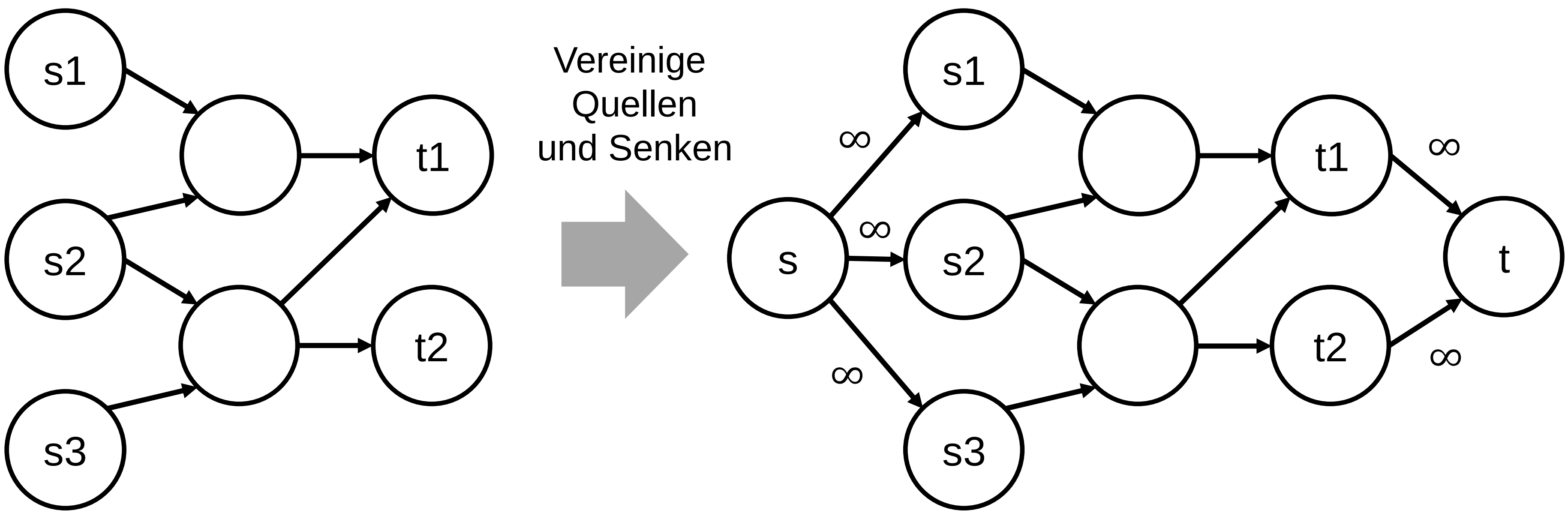


Ein **Fluss** $f: V \times V \to \mathbb{R}$ für ein Flussnetzwerk $G = (V, E)$ mit Kapazität c und Quelle s und Senke t erfüllt $0 \leq f(u, v) \leq c(u, v)$ für alle $u, v \in V$, sowie für alle $u \in V - \{s, t\}$:

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

Transformationen

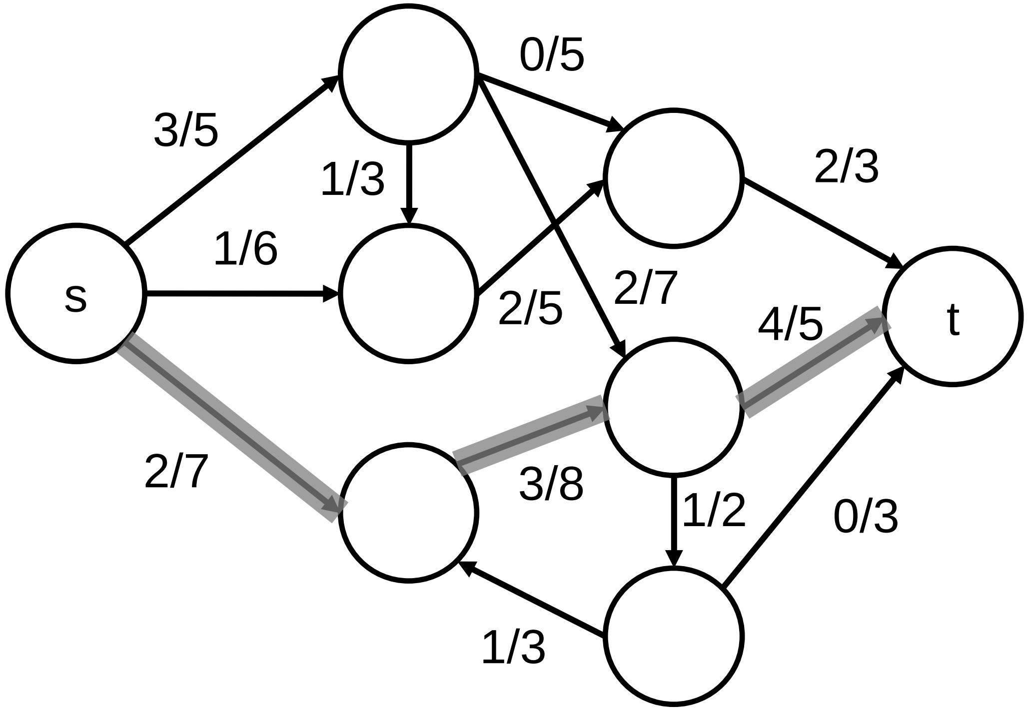




Ford-Fulkerson-Methode

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

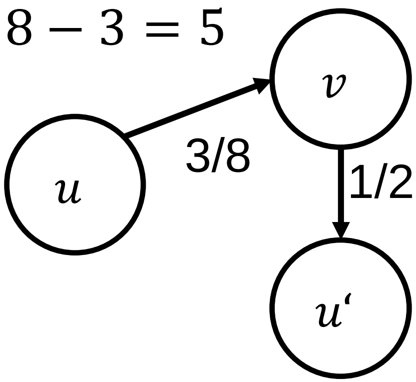
„Wieviel eingehenden Fluss über (u,v) könnte man noch zu v hinzufügen?“

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

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



$c_f(u',v) = 1$

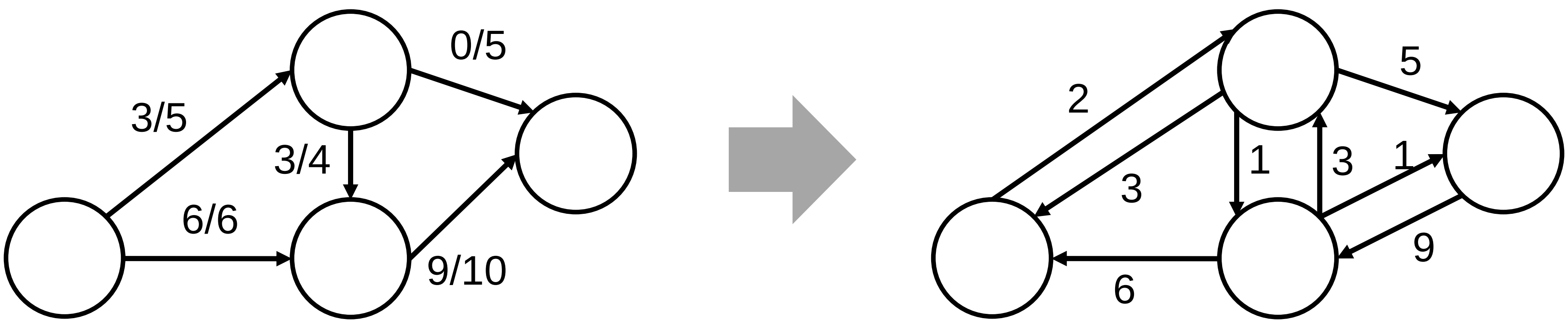
Bemerkung: nach Voraussetzung nicht beide Kanten $(u,v), (v,u)$ im Netzwerk, daher wohldefiniert

Restkapazitäts-Graph

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

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

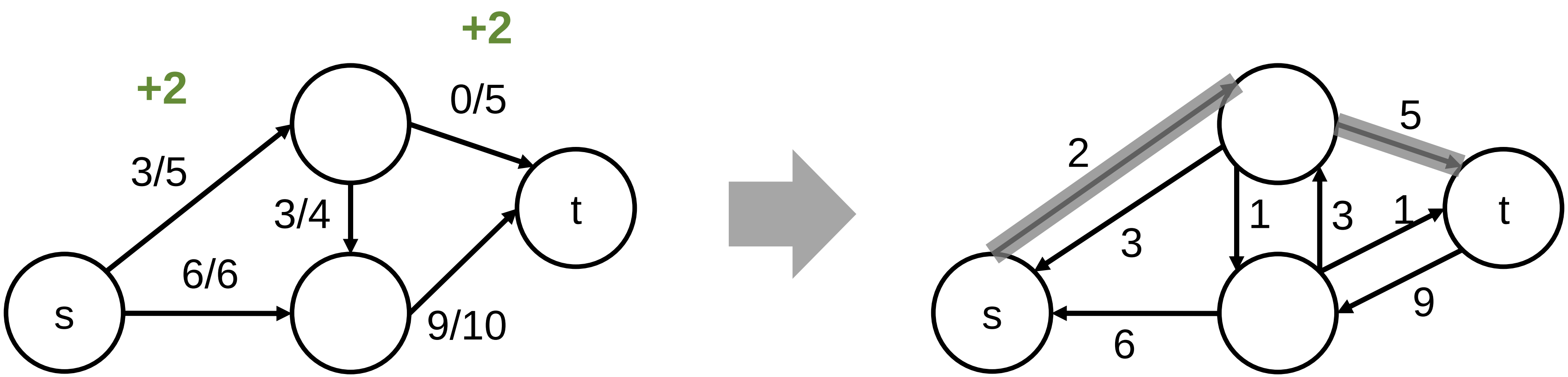
Bemerkung:
Im Restkapazitäts-Graph sind antiparallele Kanten erlaubt



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)`

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

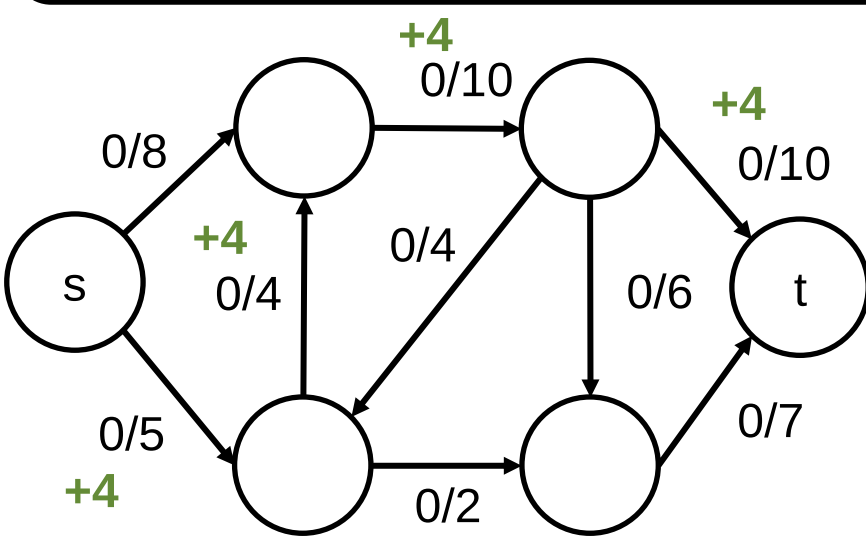
(wobei f^* maximaler Fluss und Fluss um bis zu $1/u$ pro Iteration wächst)

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

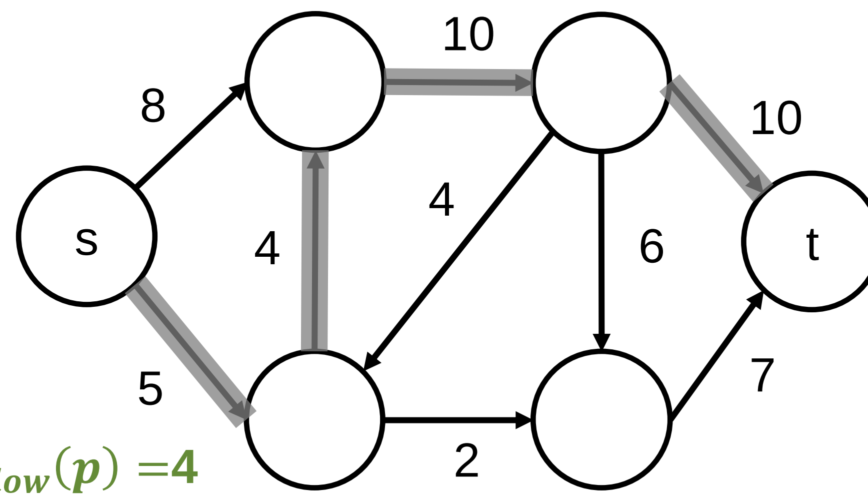
(mit Verbesserung nach Edmonds-Karp)

Ford-Fulkerson-Algorithmus: Beispiel (I)

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

FOREACH e in E DO e.flow=0;
WHILE there is path p from s to t in G_flow DO
    c_flow(p) = min {c_flow(u,v) | (u,v) in p }
    FOREACH e in p DO
        IF e in E THEN
            e.flow=e.flow+ c_flow(p)
        ELSE
            e.flow=e.flow- c_flow(p)
```



Restkapazitäts-
graph



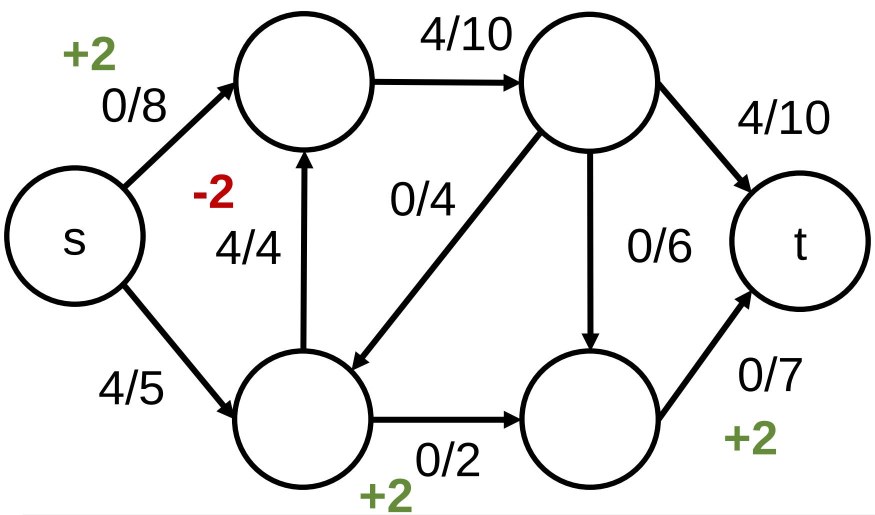
$c_{flow}(p) = 4$

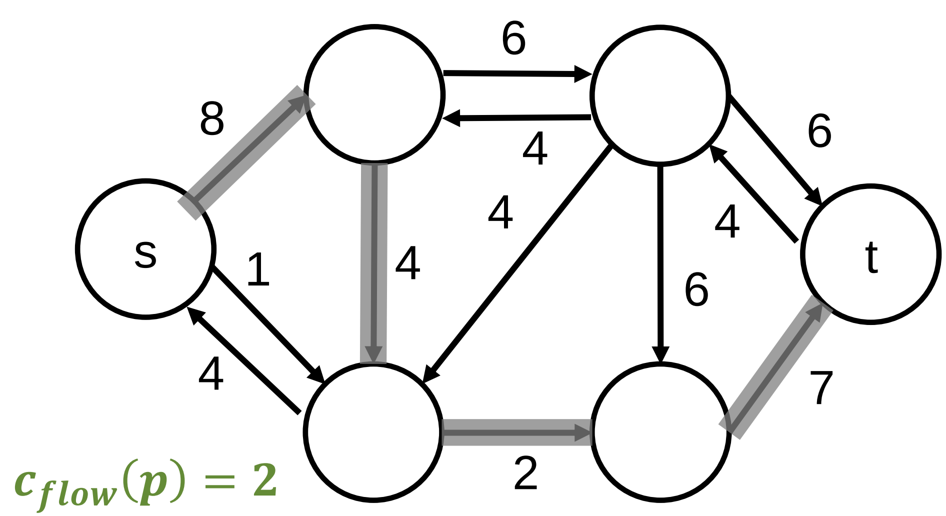
Ford-Fulkerson-Algorithmus: Beispiel (II)

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

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

Restkapazitäts-graph

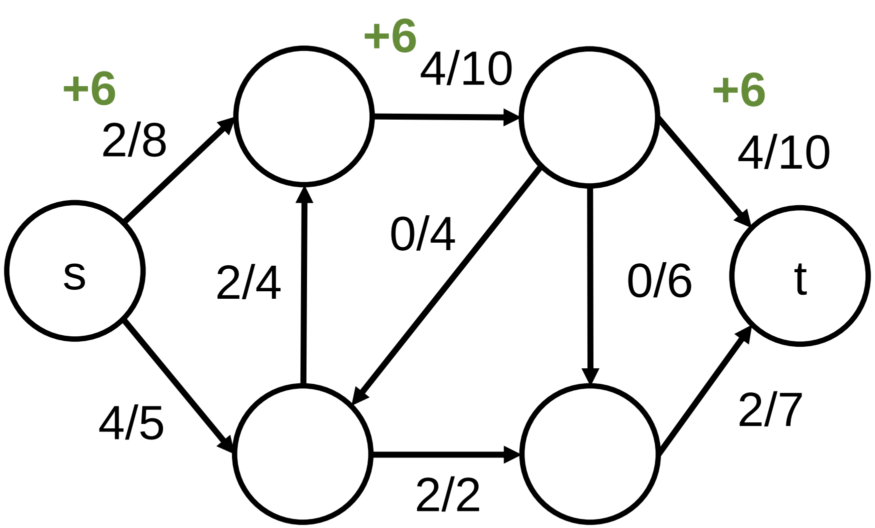




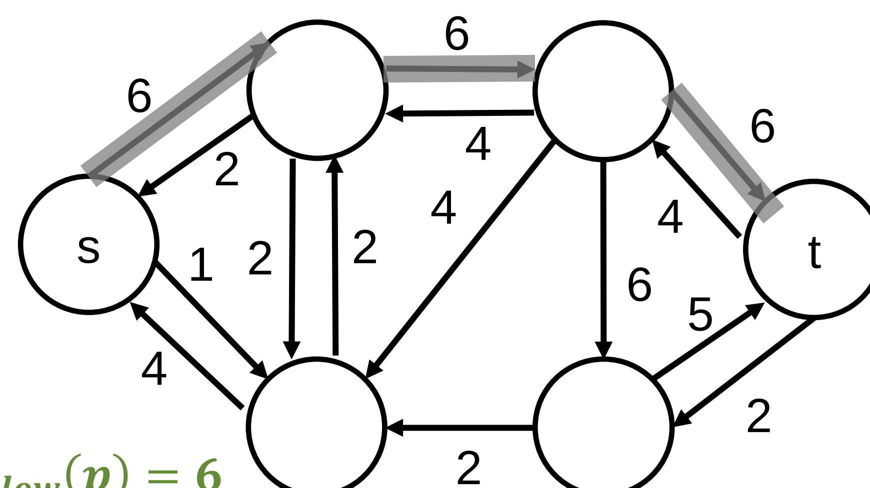
Ford-Fulkerson-Algorithmus: Beispiel (III)

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

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



Restkapazitäts-
graph



$c_{flow}(p) = 6$

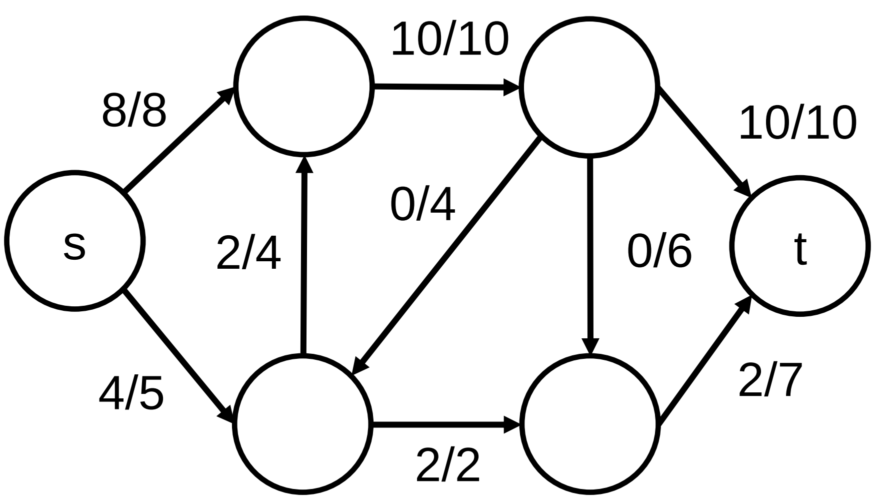
Ford-Fulkerson-Algorithmus: Beispiel (IV)

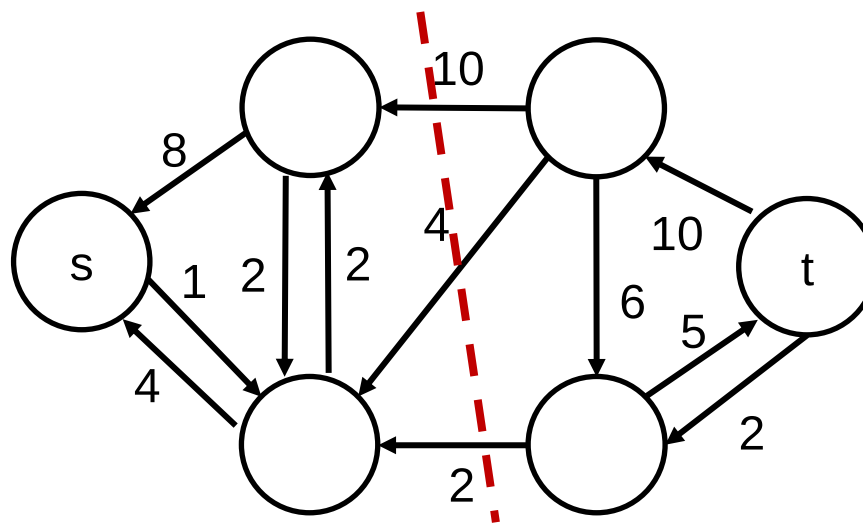
Ford-Fulkerson(G,s,t,c)

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

Kein Pfad mehr im Restkapazitäts-Graph

Restkapazitäts-graph



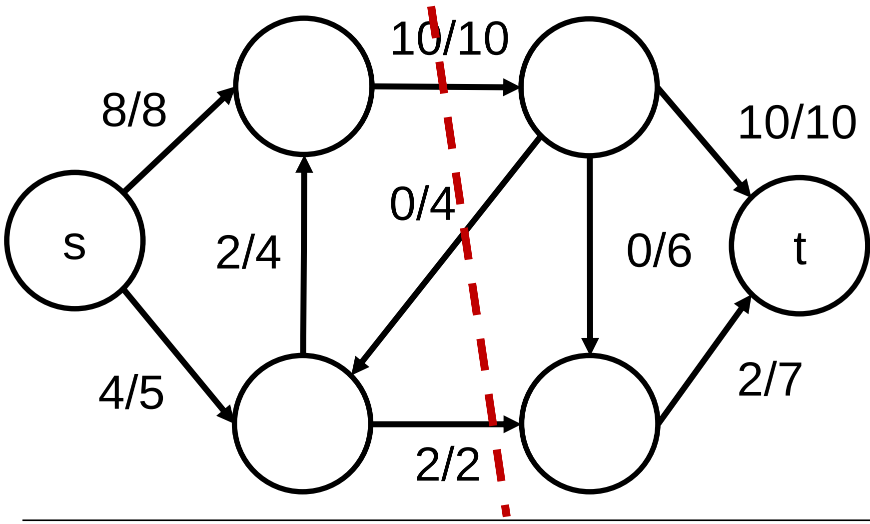


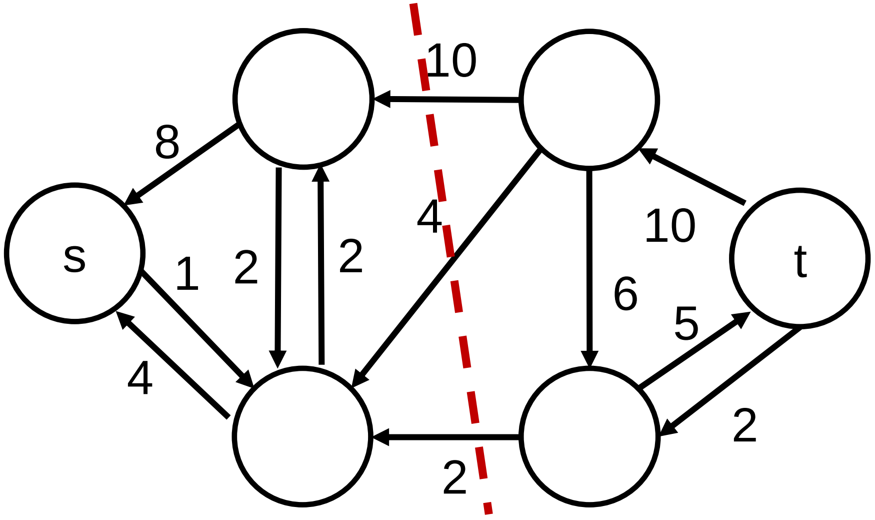
Max-Flow Min-Cut Theorem (I)

Korrektheit des Ford-Fulkerson-Algorithmus

Sei $f: V \times V \to \mathbb{R}$ 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.

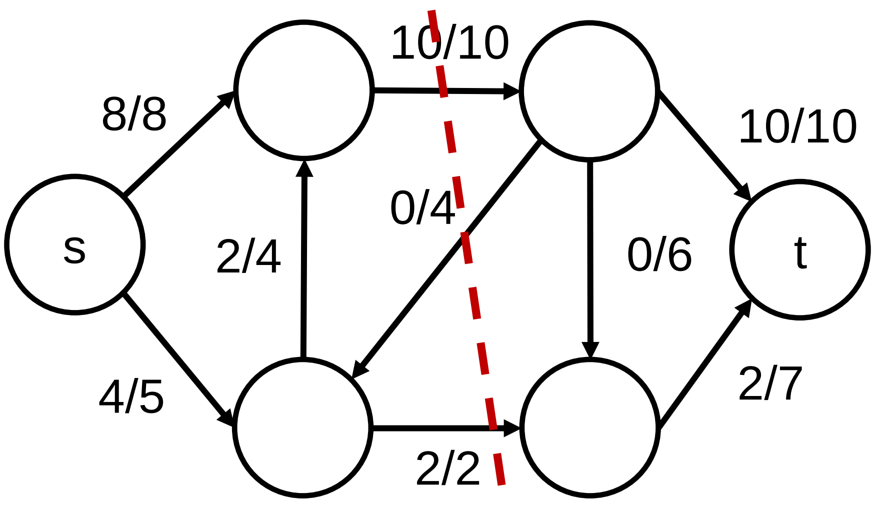




Max-Flow Min-Cut Theorem (II)

Sei $f : V \times V \to \mathbb{R}$ 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

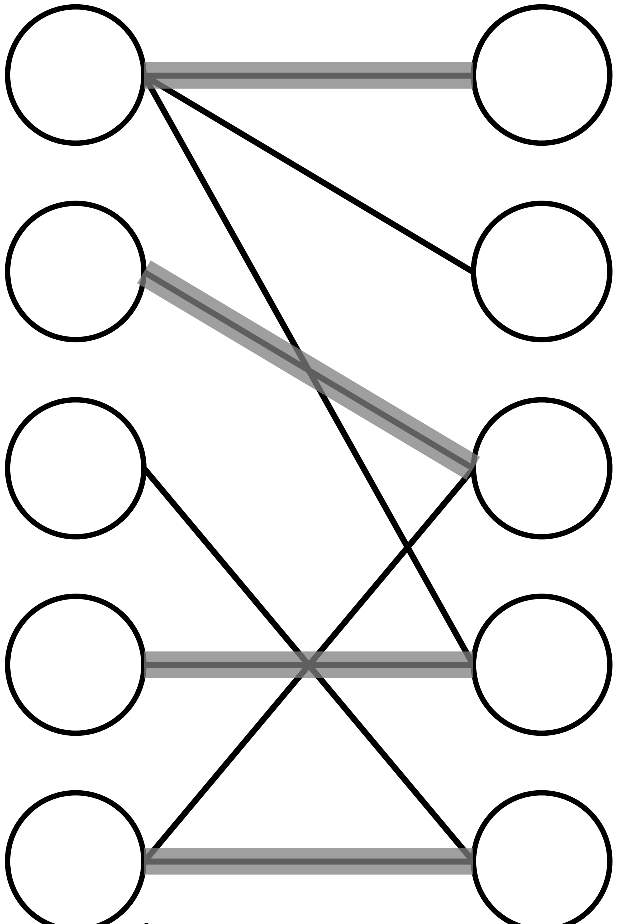
$c(S, V - S) = \sum_{u \in S} \sum_{v \in V-S} c(u, v)$

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

Beispielanwendung: Bipartites Matching (I)

Käufer

Verkäufer

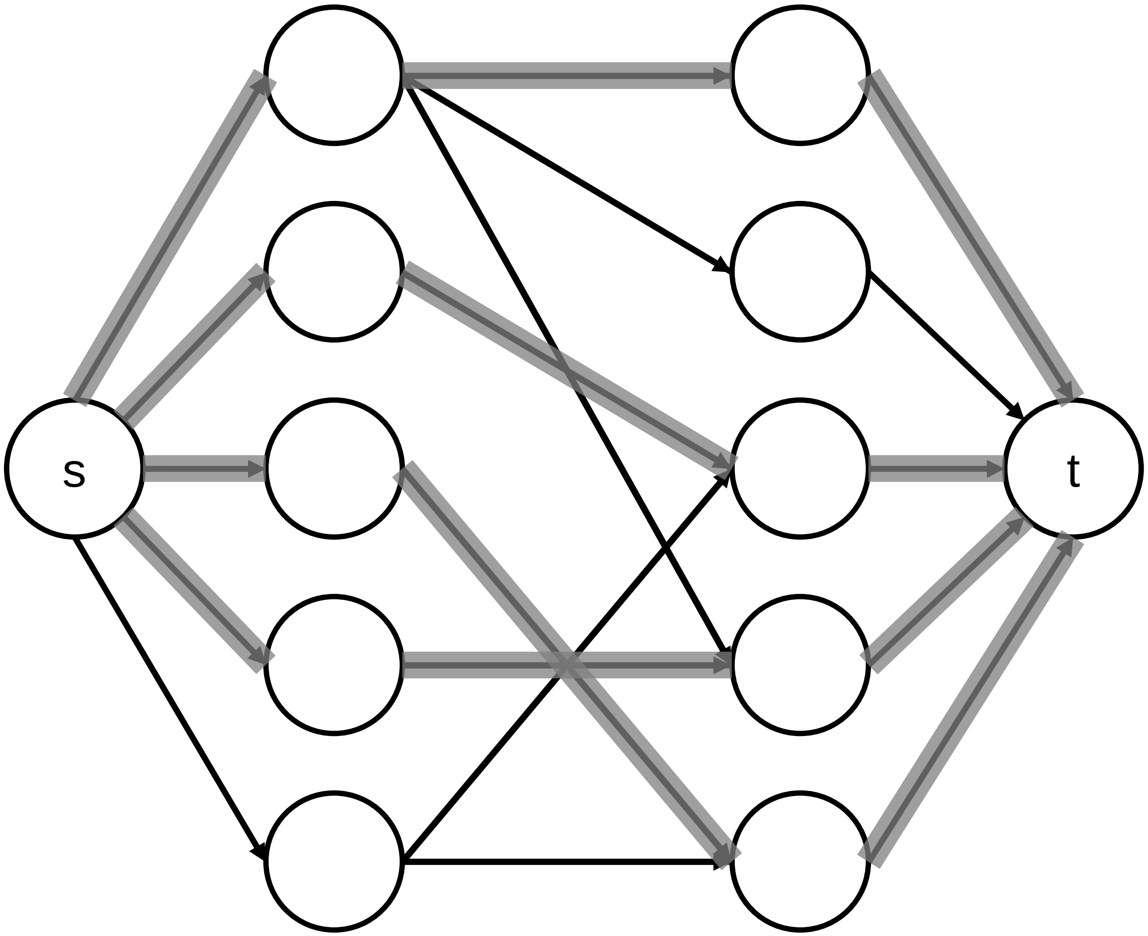


könnte Ware vom
Verkäufer kaufen

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)



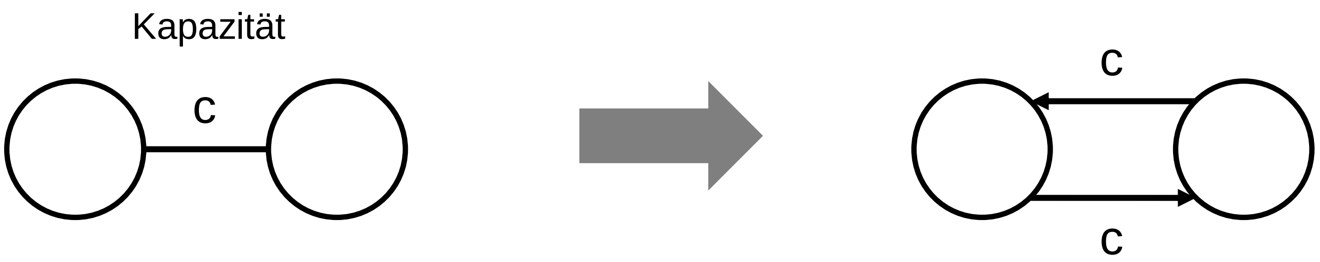
Kapazität = 1
für jede Kante

Fluss=1 sagt,
Kante aktiv

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

Ungerichtete Graphen

Ford-Fulkerson für ungerichtete Graphen:
Verwende implizite Darstellung als gerichteter Graphen



(und eliminiere dann alle antiparallelen Kanten durch Einführung neuer Knoten)

(später beim maximalen Fluss wieder „für ungerichtete Kante zurückrechnen“)

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