# Kürzeste Wege

**Wie kommt man am schnellsten von A nach B?**

# 9 Kürzeste Wege

Gegeben gewichteter, gerichteter graph $G = (V, E)$, $w : E \to \mathbb{R}$.

- SSSP (single source shortest paths):
  finde kürzesten Weg von Quelle $s$ zu allen anderen Knoten

- APSP (all pairs shortest paths):
  finde kürzesten Weg zwischen allen Knotenpaaren

Manchmal wird auch von SSSP/APSP geredet wenn man nur die Länge der entsprechenden Wege berechnen möchte.

# 9 Kürzeste Wege

Die Distanz $d(x,y)$ zwischen Knoten $x$ und $y$ ist die Länge eines kürzesten Weges.

Formal:

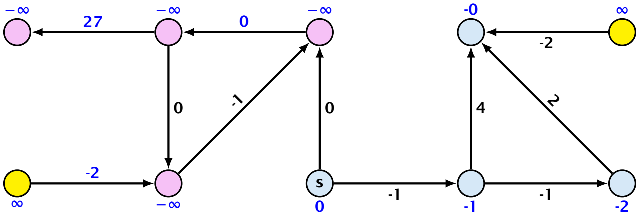
$$d(s,v) = \begin{cases} +\infty & \text{kein Pfad von } s \text{ nach } v \\ -\infty & \text{kein kürzester Pfad von } s \text{ nach } v \\ \min_{p \text{ path from } s \text{ to } v} w(p) & \text{sonst} \end{cases}$$

# 9 Kürzeste Wege

Die Distanz $d(x,y)$ zwischen Knoten $x$ und $y$ ist die Länge eines kürzesten Weges.

Formal:

$$d(s,v) = \begin{cases} +\infty & \text{kein Pfad von } s \text{ nach } v \\ -\infty & \text{kein kürzester Pfad von } s \text{ nach } v \\ \min_{p \text{ path from } s \text{ to } v} w(p) & \text{sonst} \end{cases}$$



# 9 Kürzeste Wege

Varianten des Problems:

- uniform Gewichte (alle 1)
  BFS, Laufzeit $\mathcal{O}(m+n)$
- beliebige Gewichte in einem DAG (**D**irected **A**cyclic **G**raph)
  Topologische Sortierung + Kantenrelaxierung, Laufzeit $\mathcal{O}(m+n)$
- beliebiger Graph mit nichtnegativen Gewichten
  Dijkstras Algorithmus,
  Laufzeit $\mathcal{O}((m+n)\log n)$ oder $\mathcal{O}(m+n\log n)$
- beliebiger Graph mit beliebigen Gewichten
  a) ohne negative Zyklen
  b) mit negativen Zyklen

# Kürzeste Wege

**Idee SSSP:**

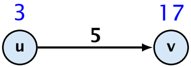
- Jeder Knoten $v$ hat Distanzlabel $\text{dist}(v)$.
- Anfangs ist $\text{dist}(s) = 0$ und $\text{dist}(v) = \infty$, d.h. $\text{dist}(x) \geq d(s, x)$ für alle Knoten $x$.
- Führe Kantenrelaxierungen durch...

# Kürzeste Wege

**Idee SSSP:**

- Jeder Knoten $v$ hat Distanzlabel $\text{dist}(v)$.
- Anfangs ist $\text{dist}(s) = 0$ und $\text{dist}(v) = \infty$, d.h. $\text{dist}(x) \geq d(s,x)$ für alle Knoten $x$.
- Führe Kantenrelaxierungen durch...

**Kantenrelaxierung:**



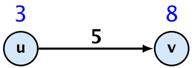

$$\text{dist}(v) := \min\{\text{dist}(v), \text{dist}(u) + w(u,v)\}$$

# Kürzeste Wege

**Idee SSSP:**

- Jeder Knoten $v$ hat Distanzlabel $\mathrm{dist}(v)$.
- Anfangs ist $\mathrm{dist}(s) = 0$ und $\mathrm{dist}(v) = \infty$, d.h. $\mathrm{dist}(x) \geq d(s,x)$ für alle Knoten $x$.
- Führe Kantenrelaxierungen durch...

**Kantenrelaxierung:**




$$\mathrm{dist}(v) := \min\{\mathrm{dist}(v), \mathrm{dist}(u) + w(u,v)\}$$

es gilt immer noch $\mathrm{dist}(v) \geq d(s,v)$...

# Kürzeste Wege

**Beobachtung:**

- Es gilt immer $\mathrm{dist}(v) \geq d(s, v)$ für alle Knoten.

**Hoffnung:**

Wenn wir genug Relaxierungen durchführen haben wir die richtigen Distanzen, da Distanzlabel nur kleiner werden.

Bei negativen Kreisen funktioniert das nicht.

Annahme: keine negativen Kreise.
$\Longrightarrow$ die kürzesten Pfade haben endlich viele „hops“.

# Kürzeste Wege

Kanten können in der Sequenz häufiger vorkommen, d.h., $e_5$ und $e_{10}$ könnten z.B. die gleiche Kante bezeichnen.

Sei $\mathcal{R} = R_1, R_2, R_3, \ldots$ eine Folge von Kantenrelaxierungen, wobei die $i$-te Relaxierung $R_i$ auf Kante $e_i$ operiert.

Sei $p = (s = v_0, v_1, \ldots, v_k = x)$ ein kürzester Pfad von $s$ nach $x$, und sei $a_i = (v_{i-1}, v_i)$ die $i$-te Kante dieses Pfades ($1 \le i \le k$).

**Beobachtung**

Falls ein $k$-elementige Teilfolge $\mathcal{S} = S_1, S_2, \ldots$ von $\mathcal{R}$ existiert bei der $S_i$ auf Kante $a_i$ operiert hat der Knoten $x$ nach Abarbeitung von $\mathcal{R}$ das richtige Distanzlabel.

Beweis: durch vollständige Induktion; nachdem $S_i$ durchgeführt ist hat $v_i$ das korrekte Distanzlabel;

# Kürzeste Wege

Kanten können in der Sequenz häufiger vorkommen, d.h., $e_5$ und $e_{10}$ könnten z.B. die gleiche Kante bezeichnen.

Sei $\mathcal{R} = R_1, R_2, R_3, \ldots$ eine Folge von Kantenrelaxierungen, wobei die $i$-te Relaxierung $R_i$ auf Kante $e_i$ operiert.

Sei $p = (s = v_0, v_1, \ldots, v_k = x)$ ein kürzester Pfad von $s$ nach $x$, und sei $a_i = (v_{i-1}, v_i)$ die $i$-te Kante dieses Pfades ($1 \leq i \leq k$).

**Beobachtung**

Falls ein $k$-elementige Teilfolge $\mathcal{S} = S_1, S_2, \ldots$ von $\mathcal{R}$ existiert bei der $S_i$ auf Kante $a_i$ operiert hat der Knoten $x$ nach Abarbeitung von $\mathcal{R}$ das richtige Distanzlabel.

Beweis: durch vollständige Induktion; nachdem $S_i$ durchgeführt ist hat $v_i$ das korrekte Distanzlabel;

# Kürzeste Wege

Kanten können in der Sequenz häufiger vorkommen, d.h., $e_5$ und $e_{10}$ könnten z.B. die gleiche Kante bezeichnen.

Sei $\mathcal{R} = R_1, R_2, R_3, \ldots$ eine Folge von Kantenrelaxierungen, wobei die $i$-te Relaxierung $R_i$ auf Kante $e_i$ operiert.

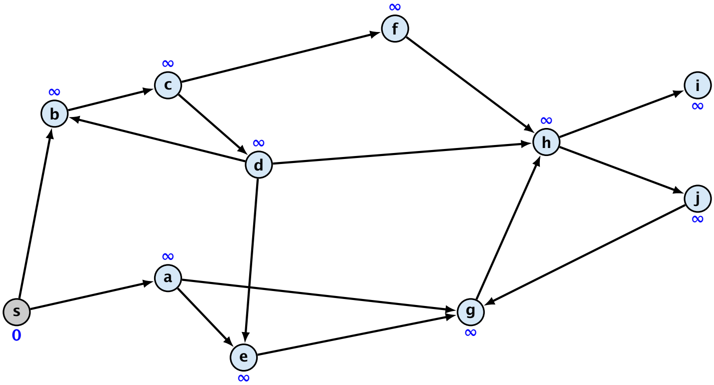
Sei $p = (s = v_0, v_1, \ldots, v_k = x)$ ein kürzester Pfad von $s$ nach $x$, und sei $a_i = (v_{i-1}, v_i)$ die $i$-te Kante dieses Pfades ($1 \le i \le k$).

**Beobachtung**

Falls ein $k$-elementige Teilfolge $\mathcal{S} = S_1, S_2, \ldots$ von $\mathcal{R}$ existiert bei der $S_i$ auf Kante $a_i$ operiert hat der Knoten $x$ nach Abarbeitung von $\mathcal{R}$ das richtige Distanzlabel.

Beweis: durch vollständige Induktion; nachdem $S_i$ durchgeführt ist hat $v_i$ das korrekte Distanzlabel;

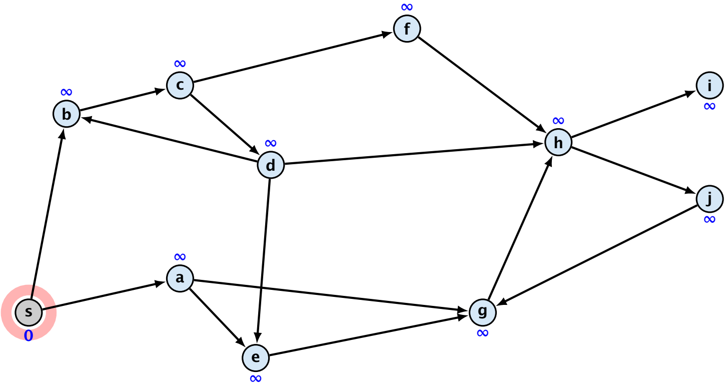
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

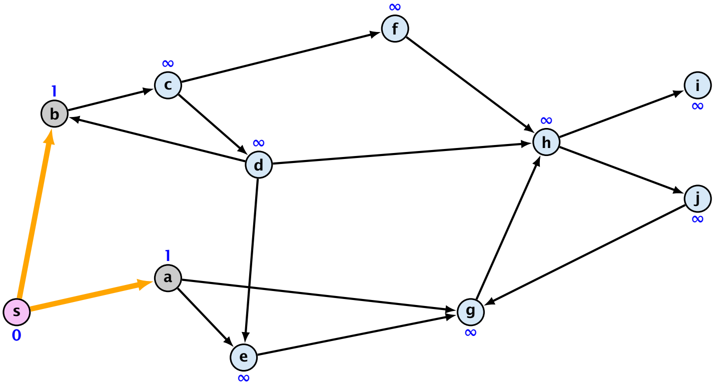
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

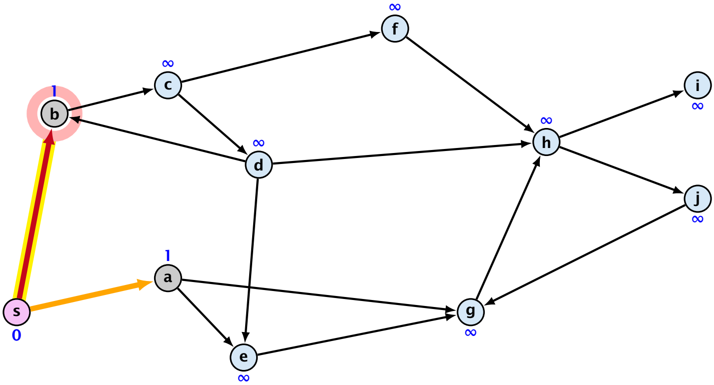
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

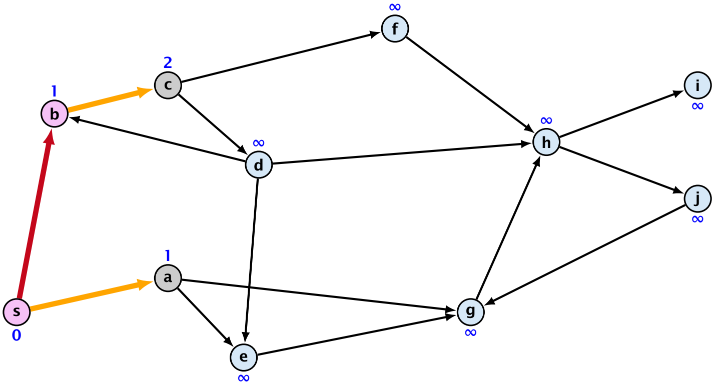
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

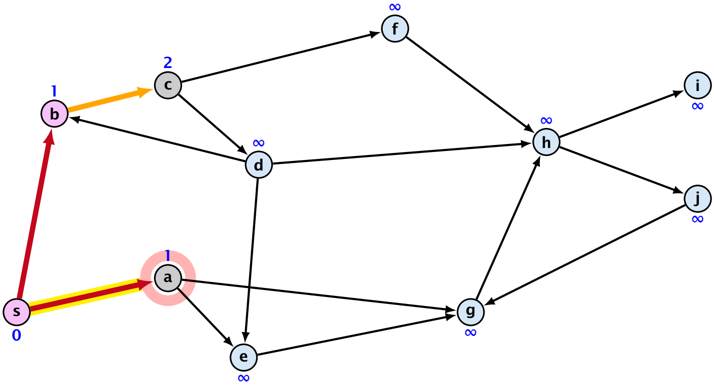
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

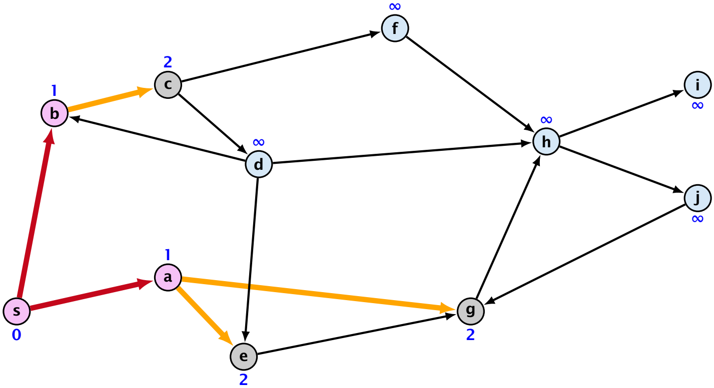
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

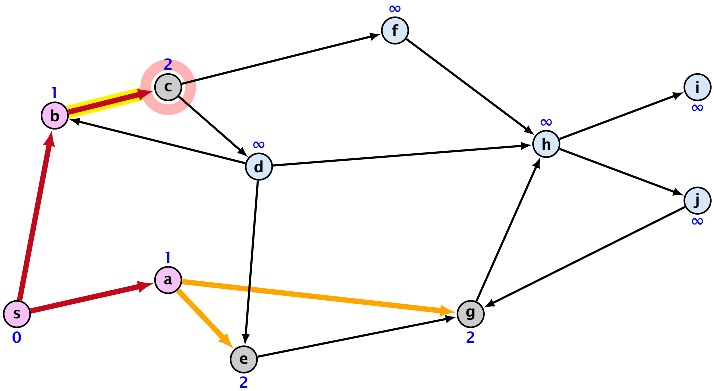
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

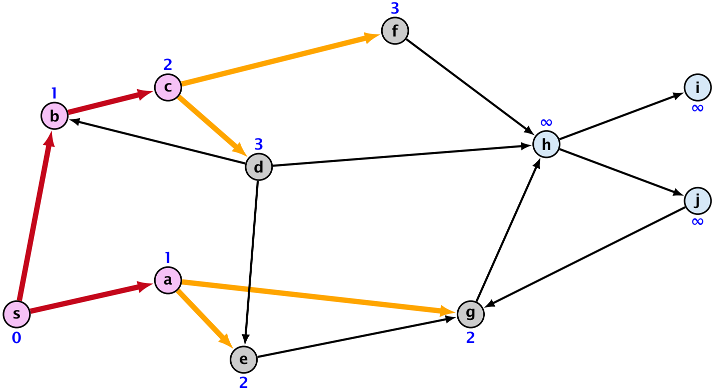
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

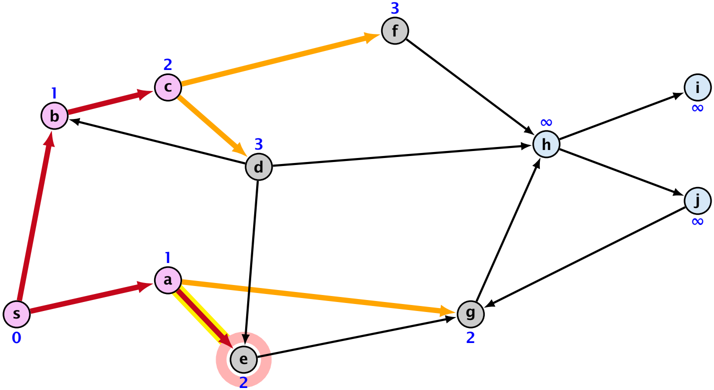
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

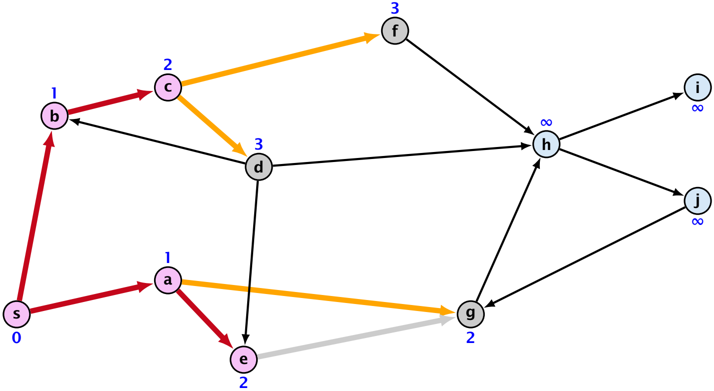
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

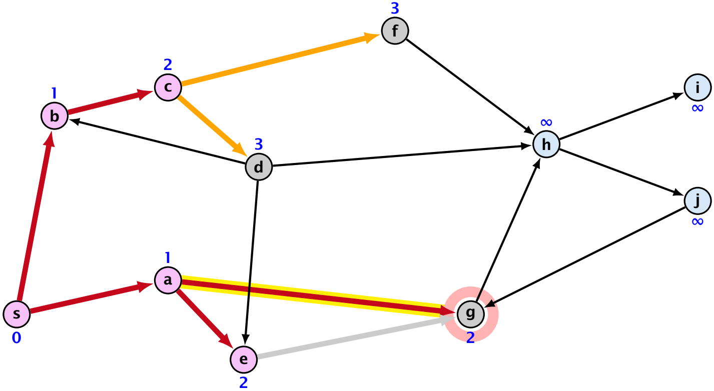
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

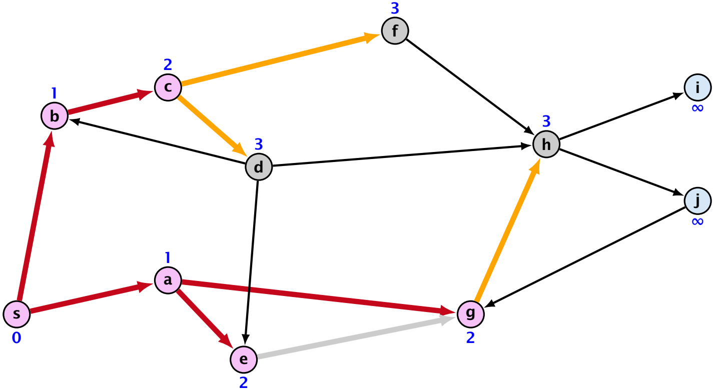
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

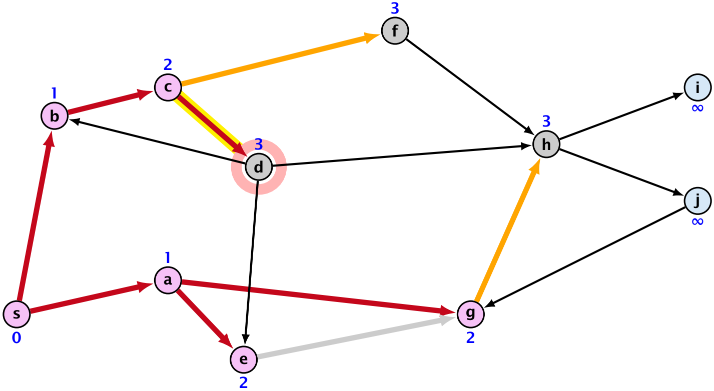
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

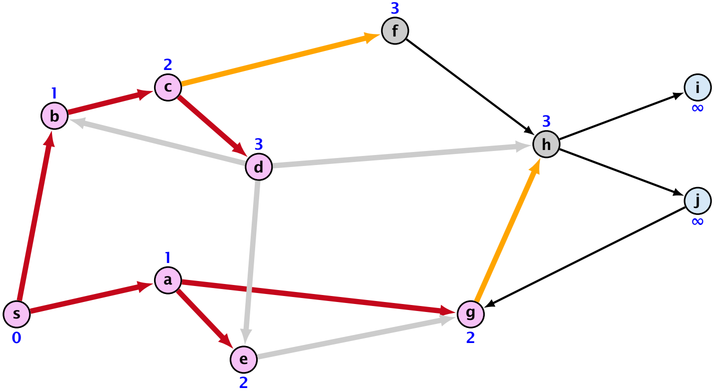
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

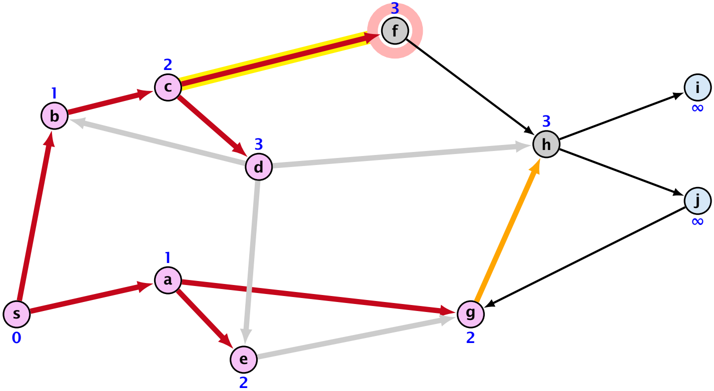
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

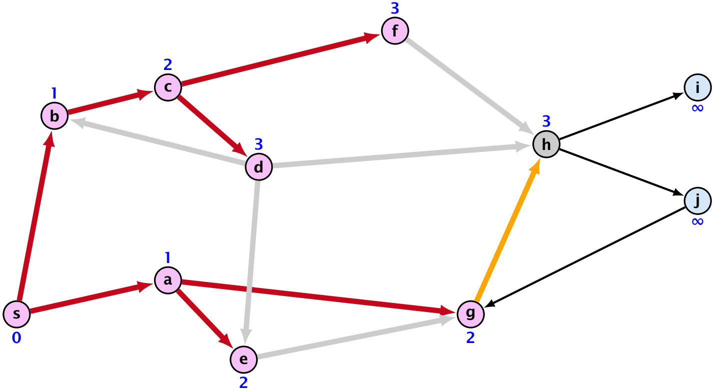
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

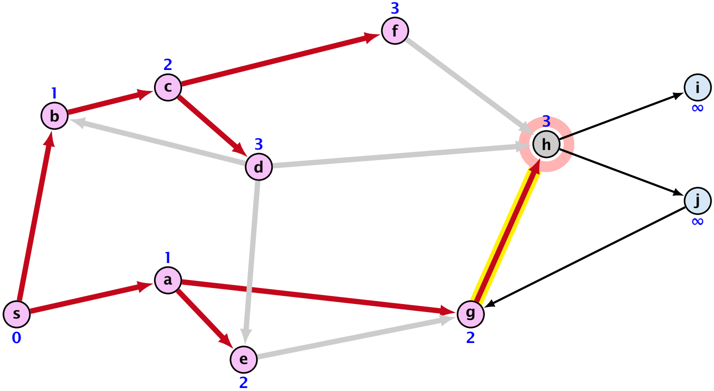
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

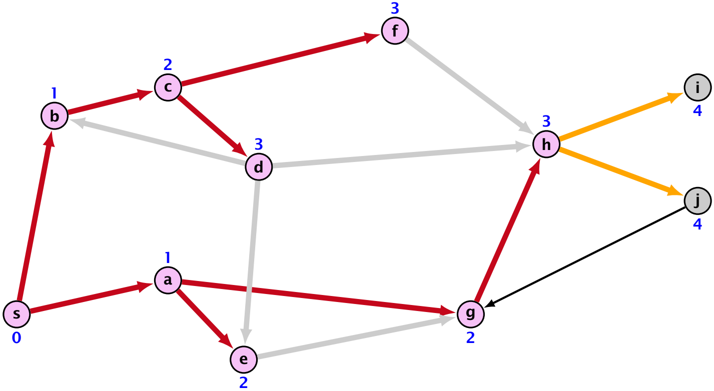
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

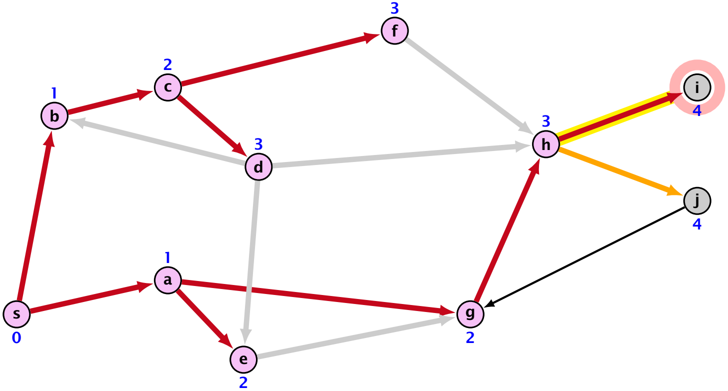
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

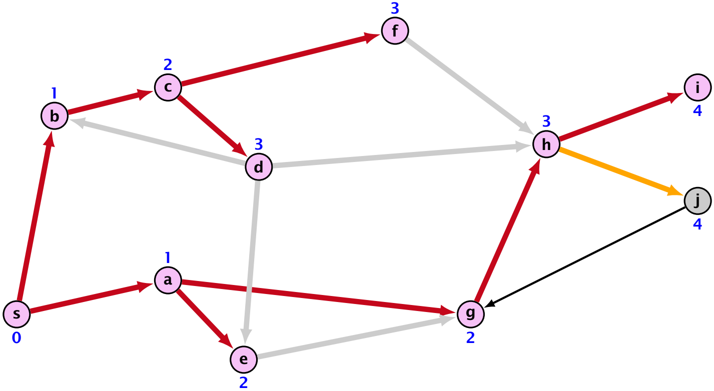
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

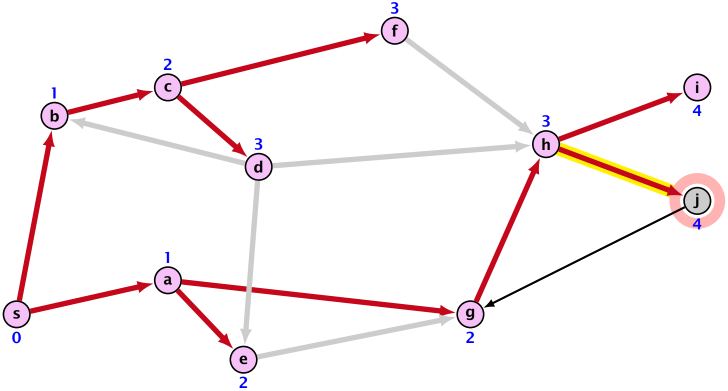
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

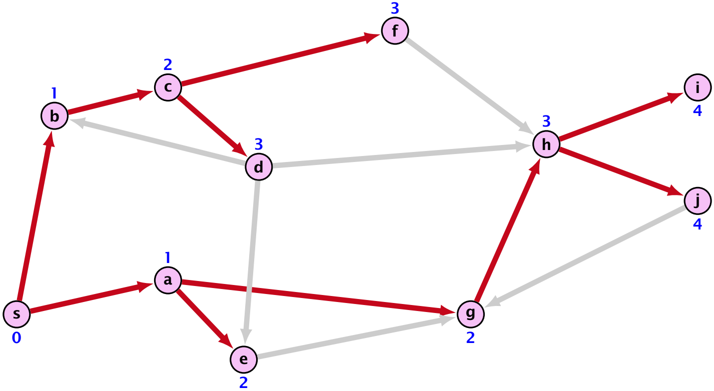
# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

# Uniforme Kantengewichte – BFS




visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

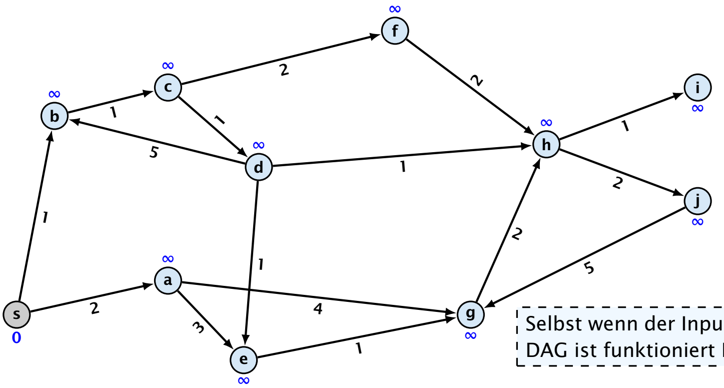
# Kürzeste Wege

Der Level ist die hop-Distanz von $s$ zu $v$; bei uniformen Kantengewichten entspricht dies der tatsächlichen Distanz.

- in der ersten Runde wird für jeden Level 1 Knoten eine eingehende Kante von Level 0 relaxiert
- in der zweiten Runde wird für jeden Level 2 Knoten eine eingehende Kante von Level 1 relaxiert
- ...

Bei einem kürzesten Pfad werden Relaxierungen in der Reihenfolge des Pfades durchgeführt.

# Beliebige Gewichte – BFS funktioniert nicht



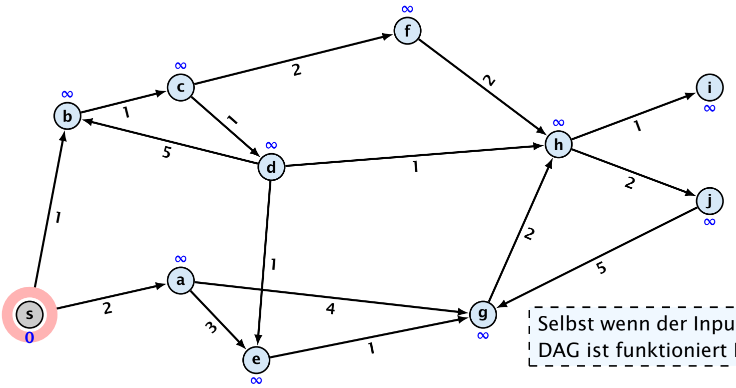

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreich relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



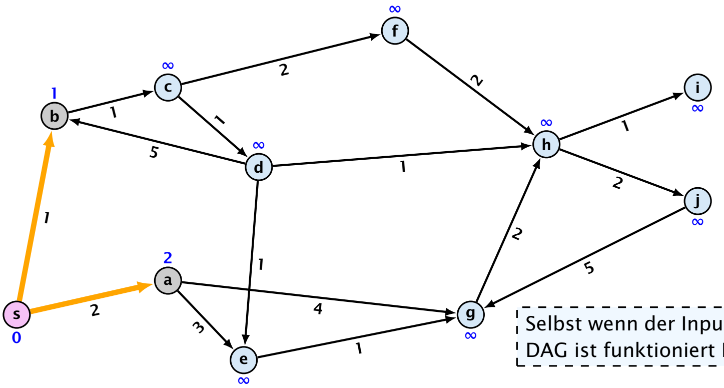

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



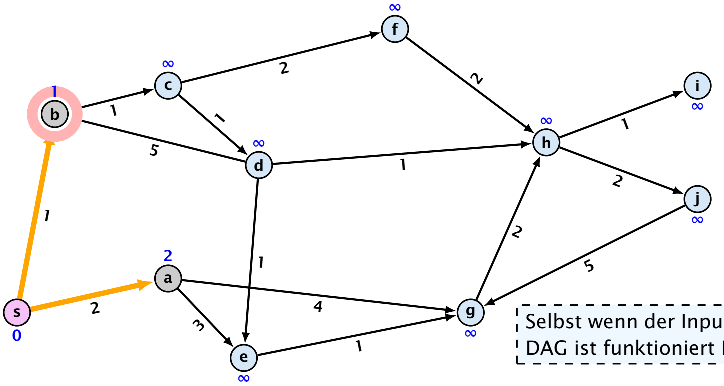

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreich relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



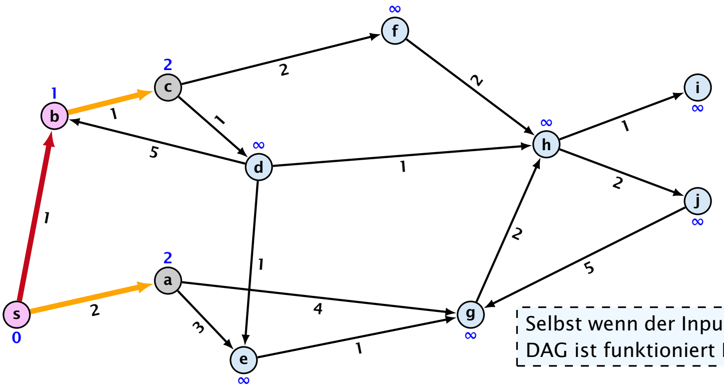

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreich relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



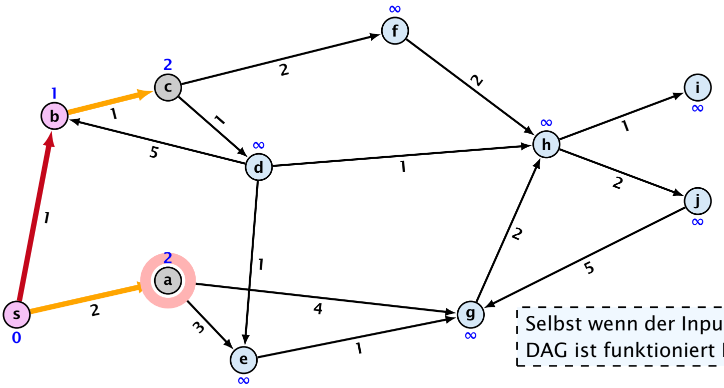

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



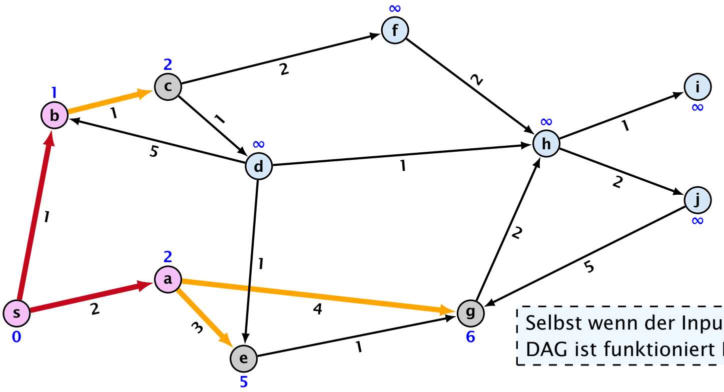

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



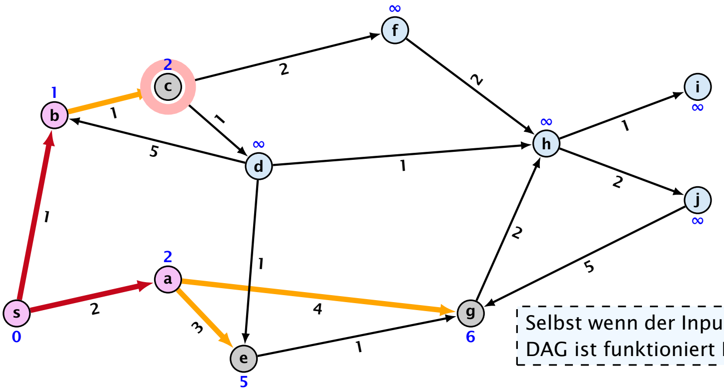

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



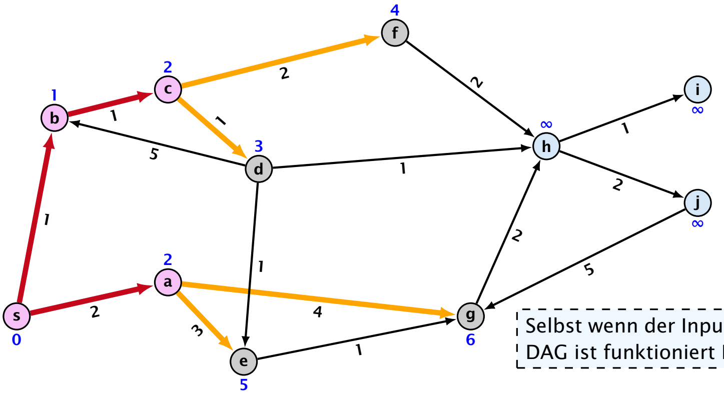

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



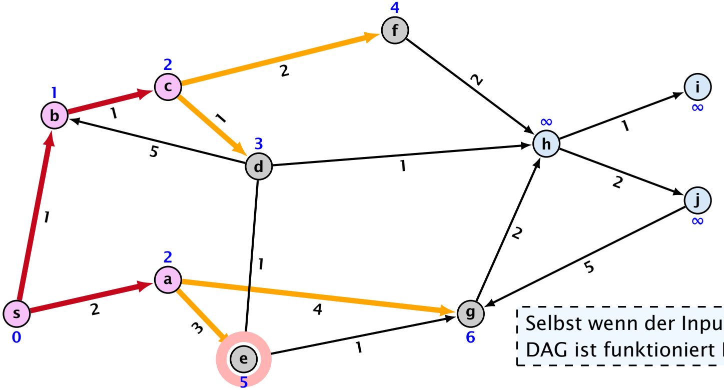

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreich relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



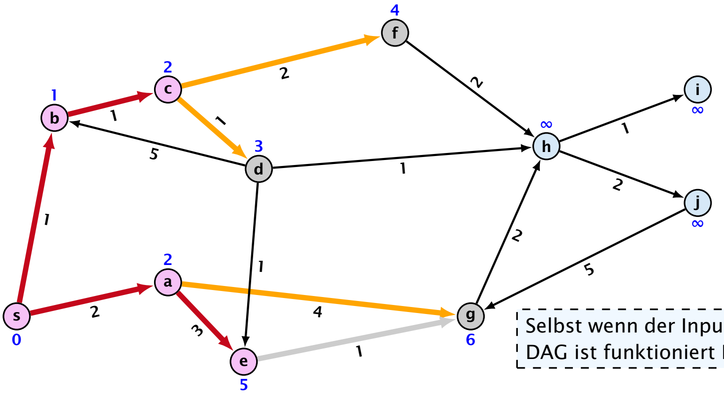

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreich relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



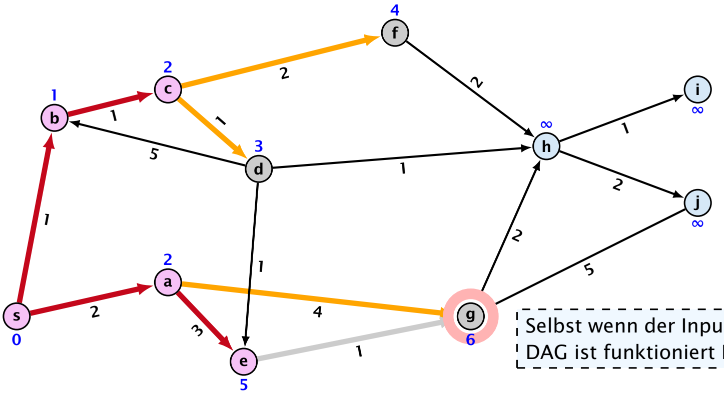

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



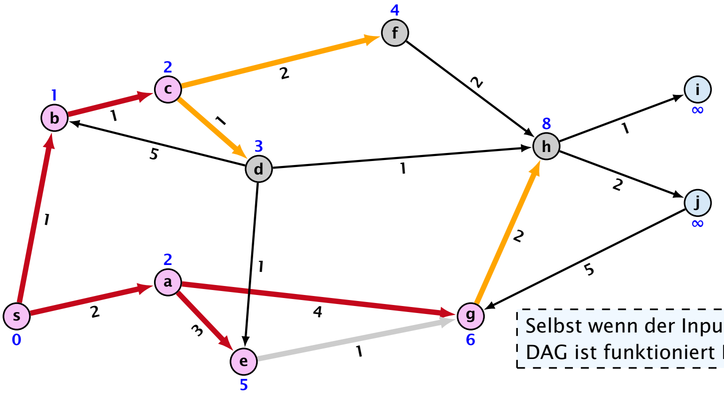

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



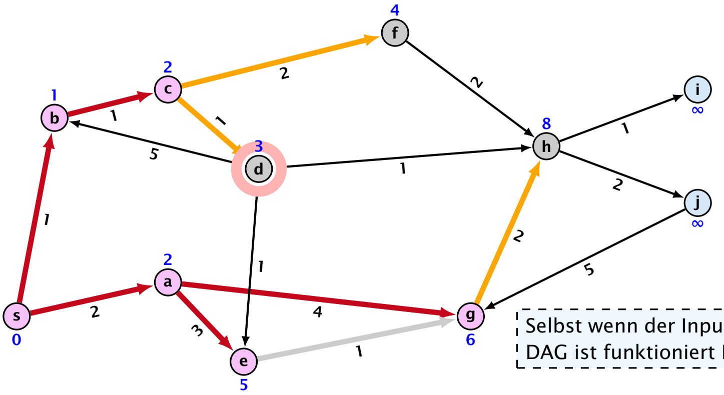

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



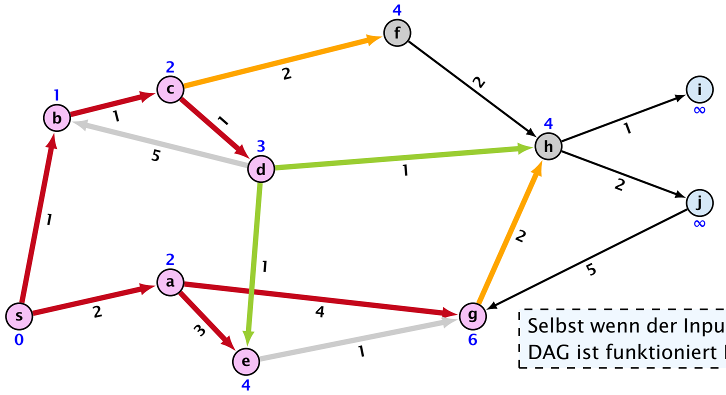

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreich relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



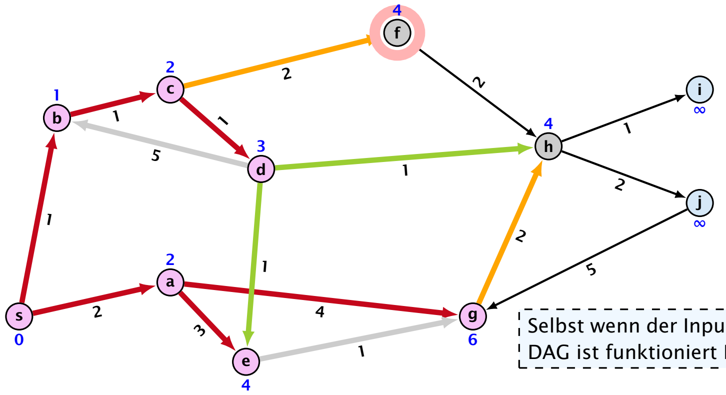

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



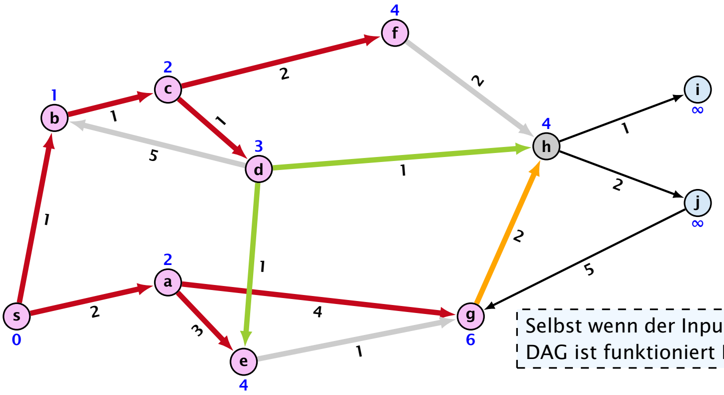

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreich relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



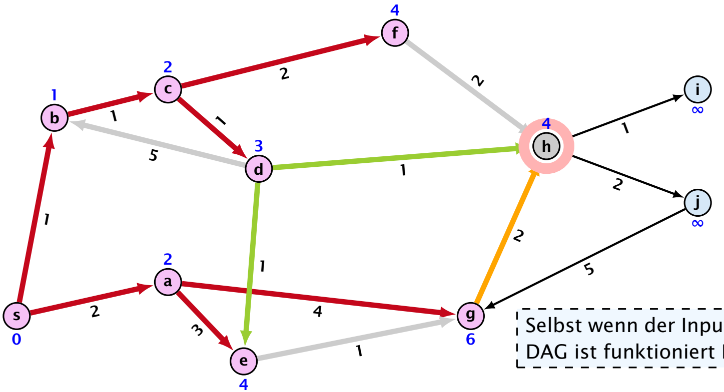

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



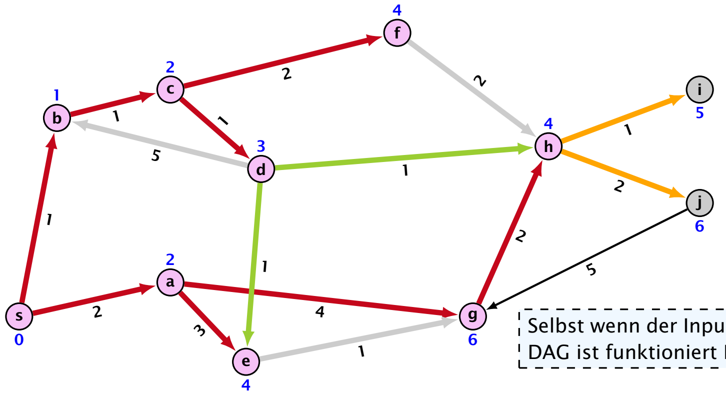

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreich relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



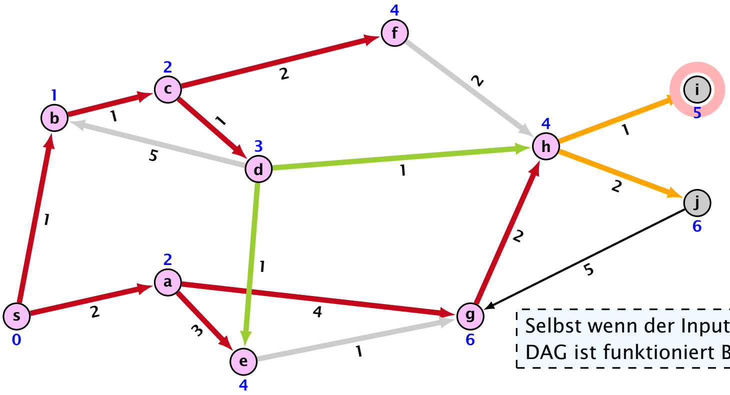

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreicht relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



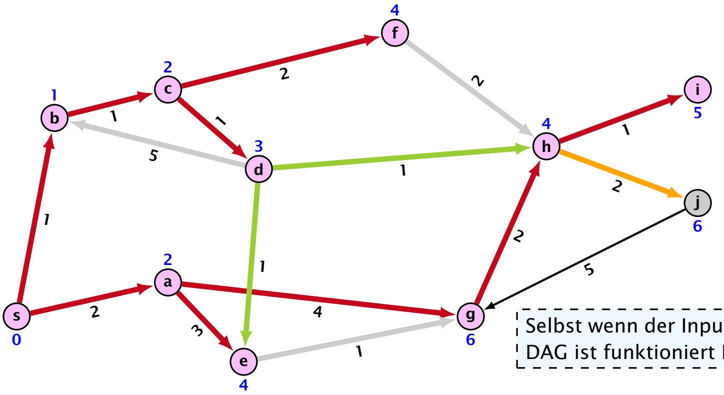

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreich relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



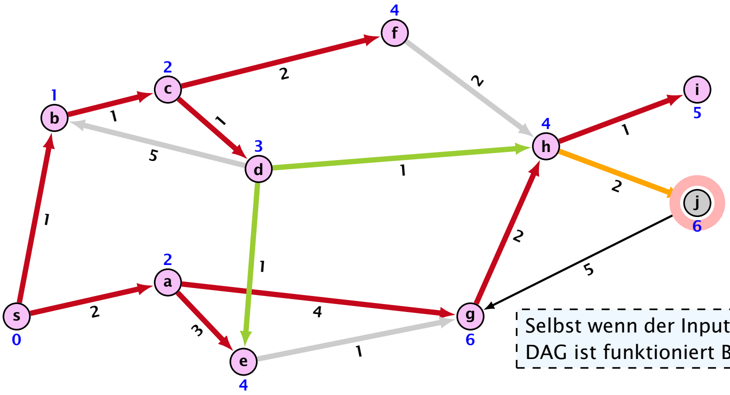

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreich relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht



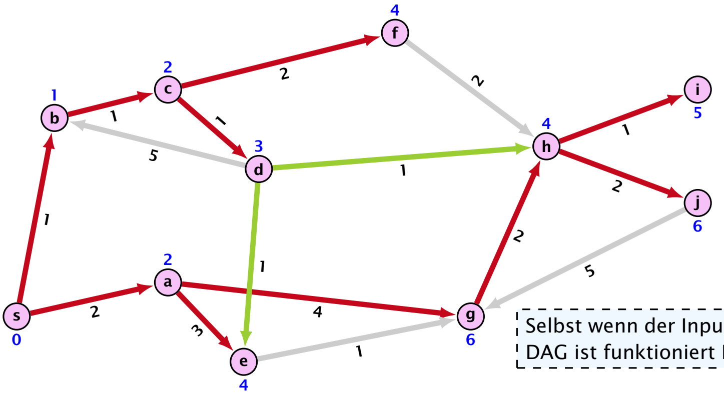

Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreich relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Beliebige Gewichte – BFS funktioniert nicht




Selbst wenn der Inputgraph ein DAG ist funktioniert BFS nicht.

visit-operation:

- relaxiere alle ausgehenden Kanten
- füge unbesuchte Nachbarn in Queue ein

Kante $(d, e)$ wird nach $(e, g)$ relaxiert. Deshalb hat Knoten $g$ am Ende falsche Distanz. Kante $(d, h)$ wird erfolgreich relaxiert; der Vorgänger von $h$ ist aber $g$ (nach BFS-Regel).

# Kürzeste Wege in DAGs

`finNum(v)` ist die bei `DFS` berechnete Reihenfolge, in der die rekursiven Aufrufe von `dfsVisit(v)` enden.

In DAGs existiert topologische Sortierung der Knoten:

- $\mathrm{top}: V \to \{1, \dots, n\}$, bijektiv
- für alle $(x, y) \in E$: $\mathrm{top}(x) < \mathrm{top}(y)$

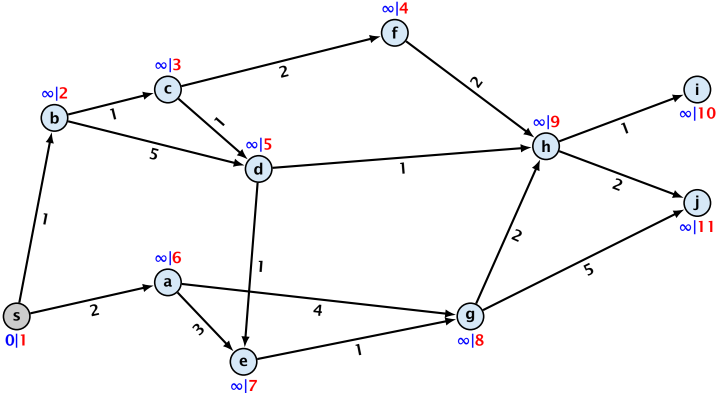
Zum Beispiel: $\mathrm{top}(v) = n - \mathrm{finNum}(v) + 1$.

```
Input: weighted DAG G=(V,E,w); start vertex s;
       array top mit top[i] i-ter Knoten in TopSort
Output: key-field of node contains distance from s;

ShortestPathDag(s,top)
    foreach v ∈ V
        v->key = ∞;
    s->key = 0;
    for i = 1 to n do
        v = top[i];
        foreach x ∈ N[v]
            x->key = min(v->key + w(v,x),x->key);
```

# Kürzeste Wege in DAGs

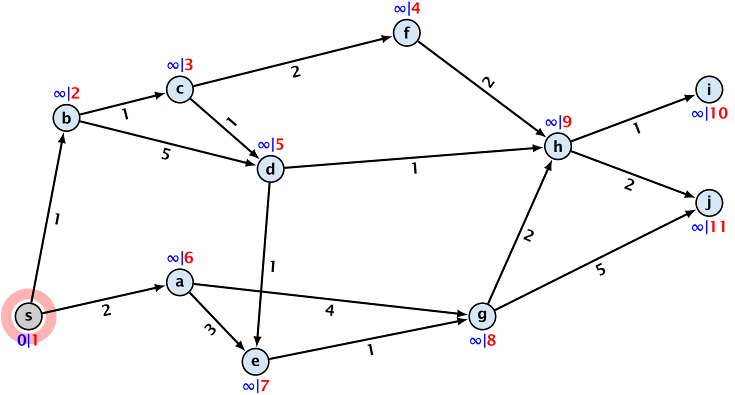
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\mathrm{dist}(y) := \min\{\mathrm{dist}(x) + w(x,y), \mathrm{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

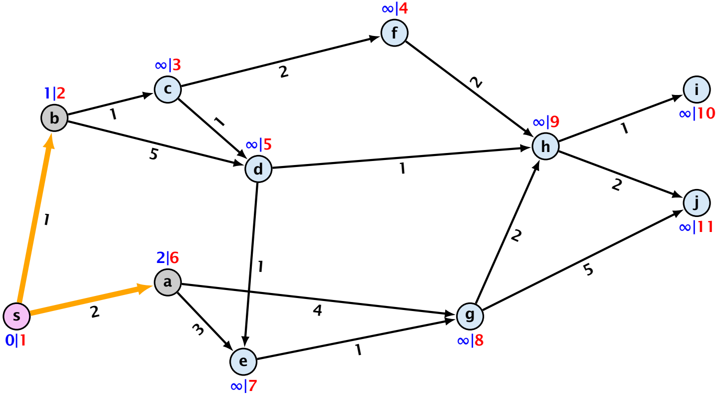
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\mathrm{dist}(y) := \min\{\mathrm{dist}(x) + w(x,y), \mathrm{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

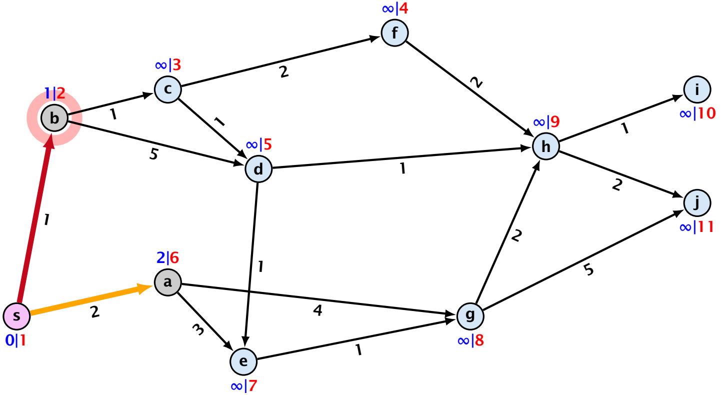
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x, y)$): $\mathrm{dist}(y) := \min\{\mathrm{dist}(x) + w(x, y), \mathrm{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

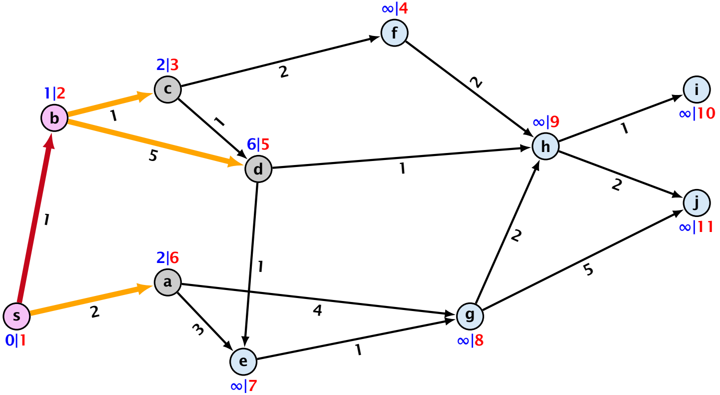
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x,y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

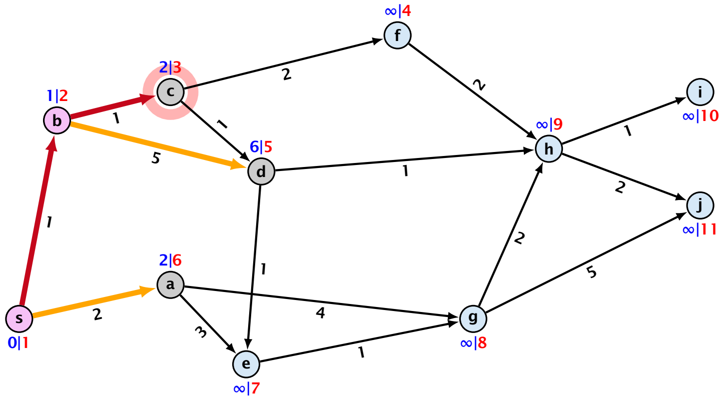
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x, y)$): $\mathrm{dist}(y) := \min\{\mathrm{dist}(x) + w(x, y), \mathrm{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

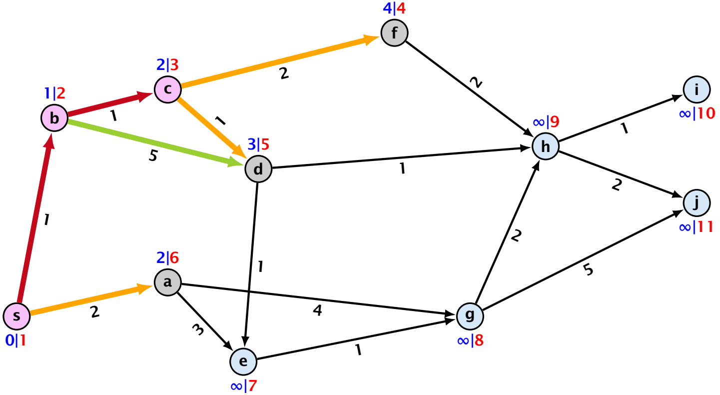
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x, y)$): $\mathrm{dist}(y) := \min\{\mathrm{dist}(x) + w(x, y), \mathrm{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

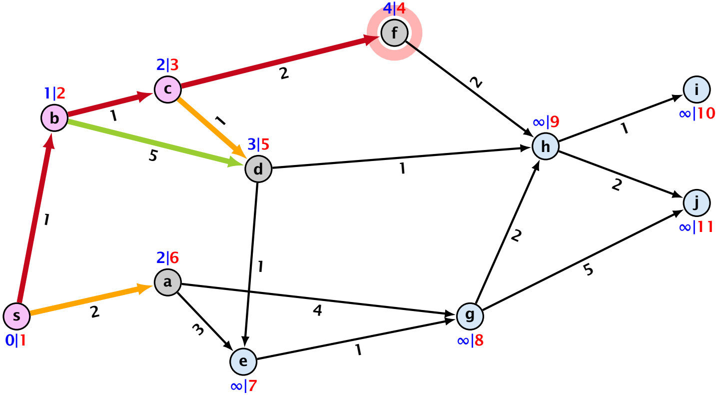
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x, y)$): $\mathrm{dist}(y) := \min\{\mathrm{dist}(x) + w(x, y), \mathrm{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

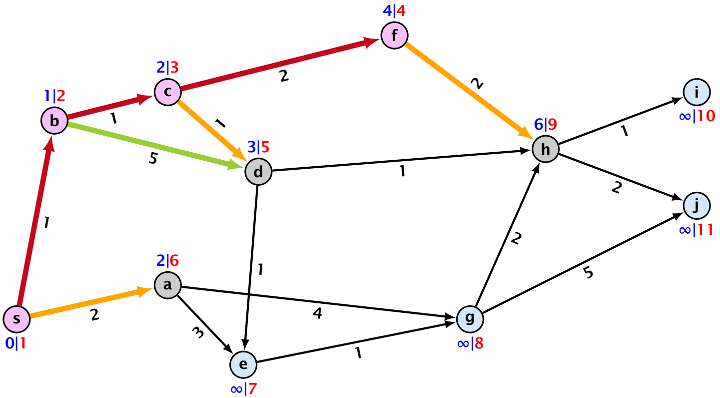
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\mathrm{dist}(y) := \min\{\mathrm{dist}(x) + w(x,y), \mathrm{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

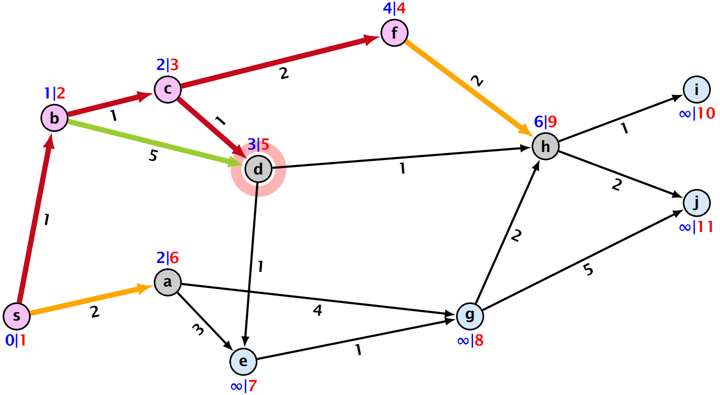
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x, y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x, y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

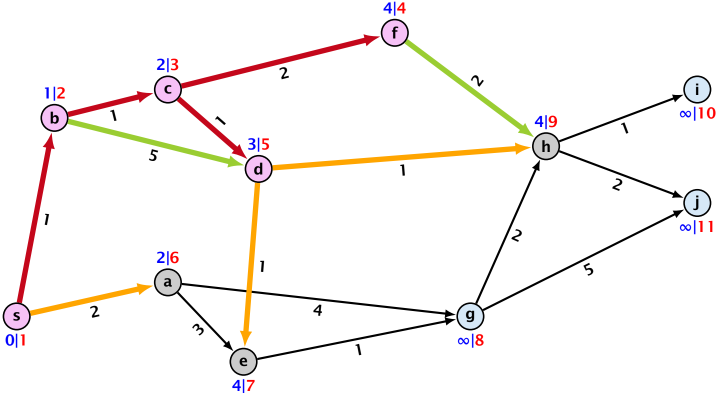
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x, y)$): $\mathrm{dist}(y) := \min\{\mathrm{dist}(x) + w(x, y), \mathrm{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

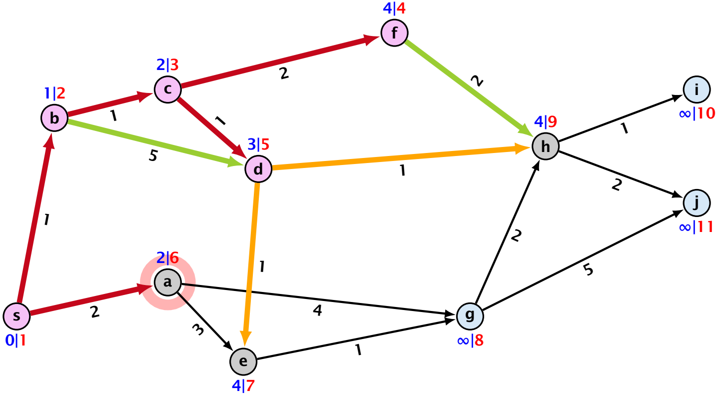
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x,y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

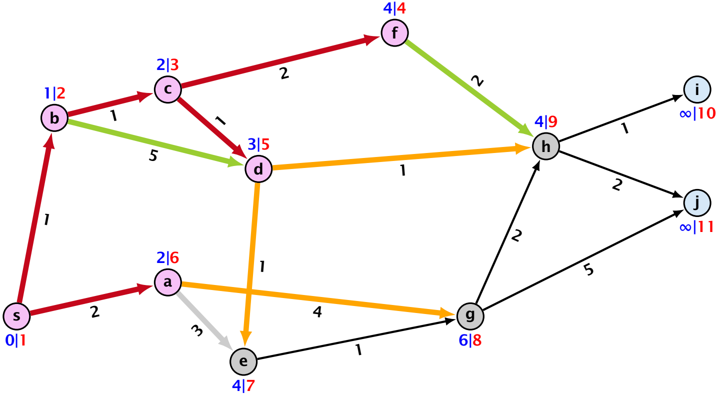
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x, y)$): $\mathrm{dist}(y) := \min\{\mathrm{dist}(x) + w(x, y), \mathrm{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

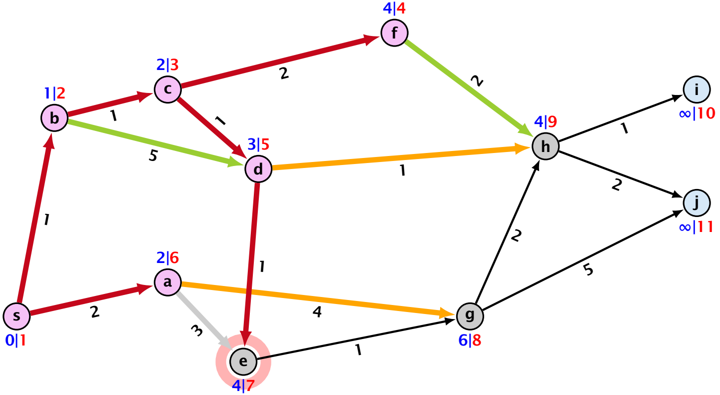
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x, y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x, y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

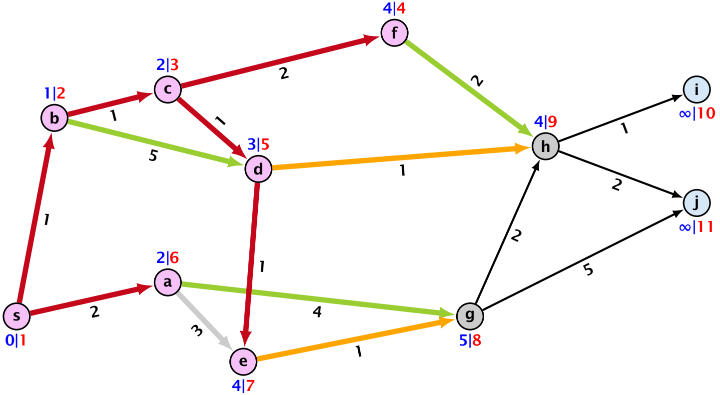
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x,y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

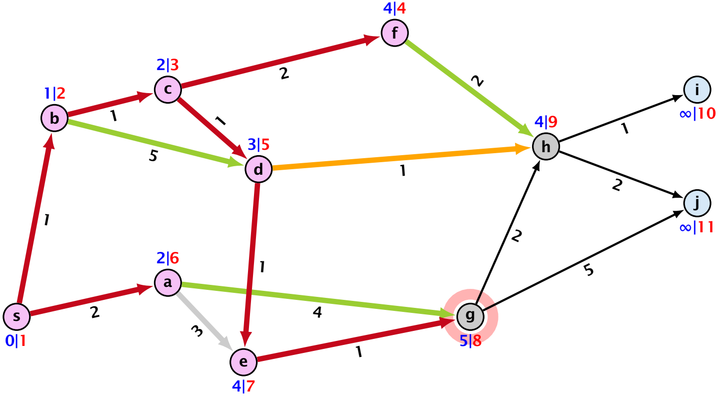
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x, y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x, y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

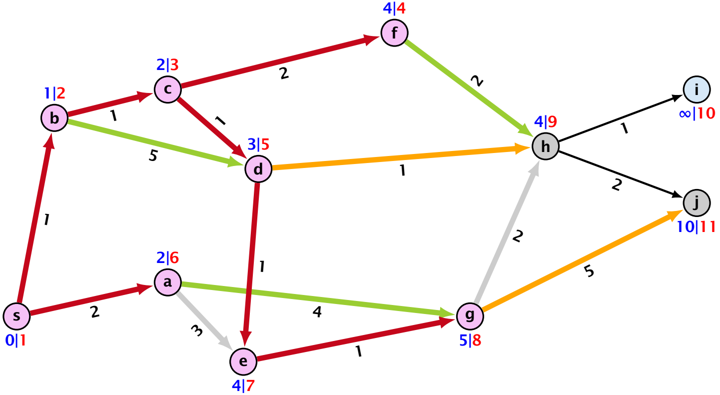
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\mathrm{dist}(y) := \min\{\mathrm{dist}(x) + w(x,y), \mathrm{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

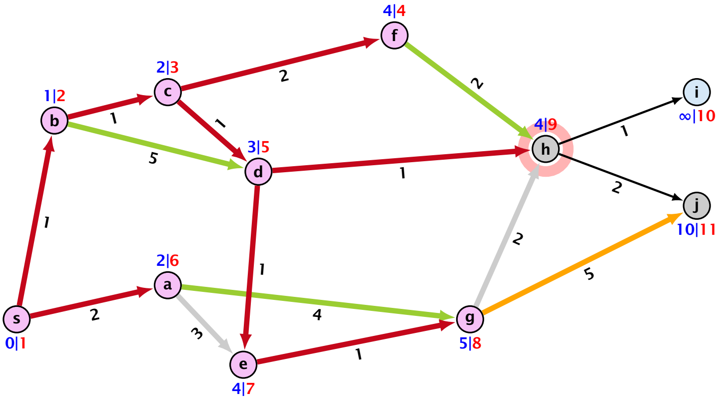
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x, y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x, y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

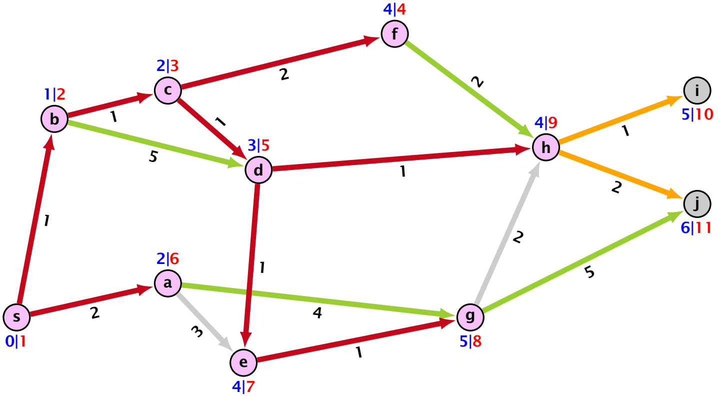
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\mathrm{dist}(y) := \min\{\mathrm{dist}(x) + w(x,y), \mathrm{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

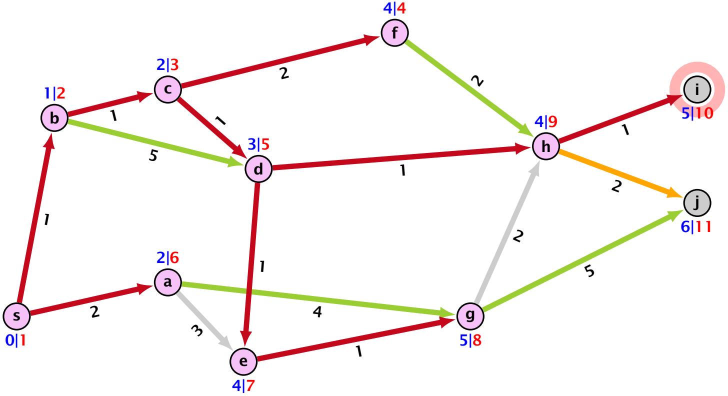
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\mathrm{dist}(y) := \min\{\mathrm{dist}(x) + w(x,y), \mathrm{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

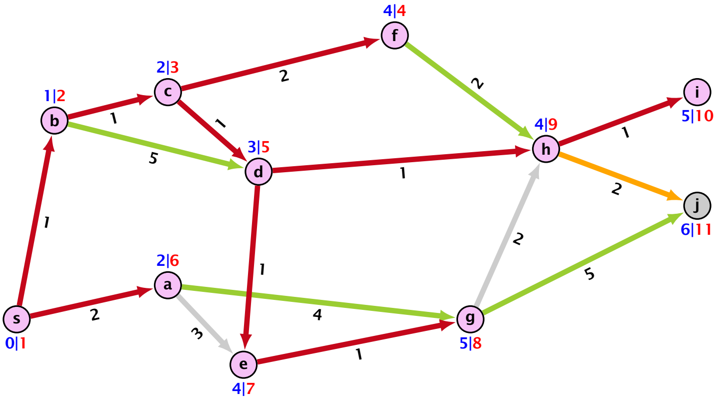
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\mathrm{dist}(y) := \min\{\mathrm{dist}(x) + w(x,y), \mathrm{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

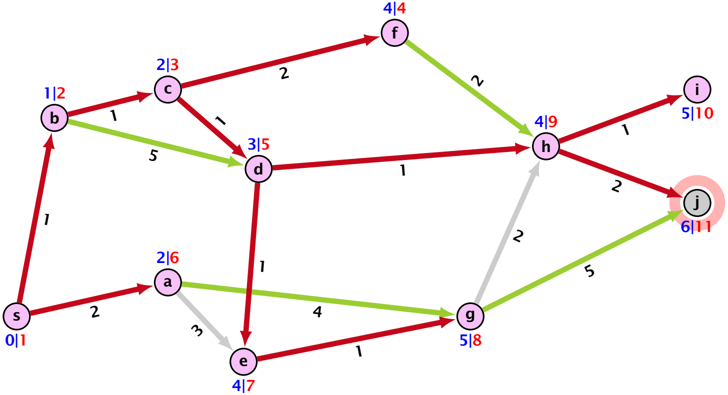
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x, y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x, y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

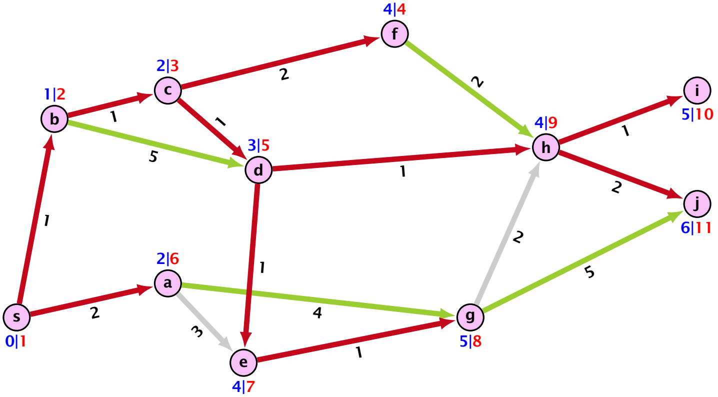
Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x, y)$): $\text{dist}(y) := \min\{\text{dist}(x) + w(x, y), \text{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

Besuche Knoten gemäß topologischer Sortierung. Wenn man Knoten $x$ besucht führe Relaxierung von ausgehenden Kanten durch (Kante $(x,y)$): $\mathrm{dist}(y) := \min\{\mathrm{dist}(x) + w(x,y), \mathrm{dist}(y)\}$.




Funktioniert für beliebige Kantengewichte.

# Kürzeste Wege in DAGs

**Korrektheit:**
Für **jeden** Pfad werden die Relaxierungen in der Reihenfolge des Pfades durchgeführt.

# Kürzeste Wege in DAGs

**Laufzeit:**

DFS für topologische Sortierung

- Laufzeit $\mathcal{O}(n+m)$

Danach besucht der Algorithmus jede Kante genau einmal.
Zusätzlich wird jeder Knoten besucht:

- Laufzeit $\mathcal{O}(n+m)$

Also, Gesamtlaufzeit $\mathcal{O}(n+m)$.

# Beliebige Graphen — nichtnegative Gewichte

Eingabe:

- beliebiger Graph (gerichtet oder ungerichtet)
- nichtnegative Gewichte

# Beliebige Graphen — nichtnegative Gewichte

Eingabe:

- beliebiger Graph (gerichtet oder ungerichtet)
- nichtnegative Gewichte

Idee:

- Distanzrelaxierungen entlang eines kürzesten $s$-$x$ für alle $x \in V$, $x \neq s$.

# Beliebige Graphen — nichtnegative Gewichte

Eingabe:

- beliebiger Graph (gerichtet oder ungerichtet)
- nichtnegative Gewichte

Idee:

- Distanzrelaxierungen entlang eines kürzesten $s$-$x$ für alle $x \in V$, $x \neq s$.

Problem:

- führe Relaxierungen in der richtigen Reihenfolge durch

# Beliebige Graphen — nichtnegative Gewichte

Eingabe:

- beliebiger Graph (gerichtet oder ungerichtet)
- nichtnegative Gewichte

Idee:

- Distanzrelaxierungen entlang eines kürzesten $s$-$x$ für alle $x \in V$, $x \neq s$.

Problem:

- führe Relaxierungen in der richtigen Reihenfolge durch

Lösung(?):

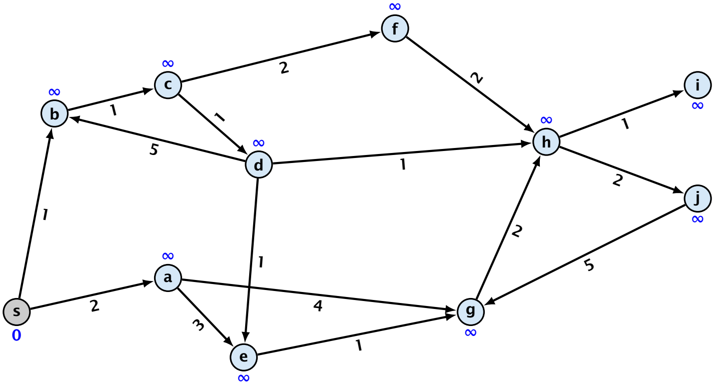
- führe Relaxierungen gemäß der Distanz von $s$ durch

# Dijkstra's Algorithmus

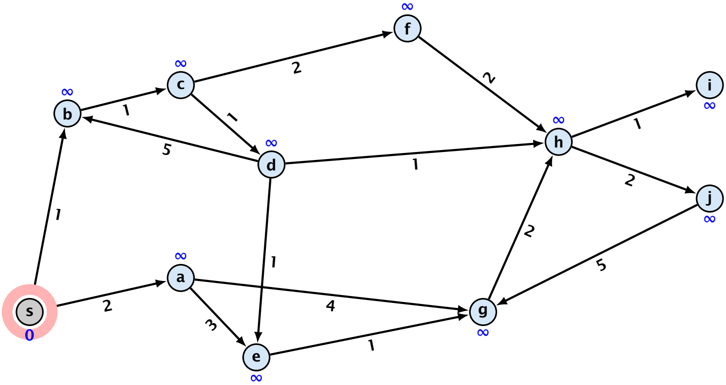
```
Input: weighted graph G=(V,E,w); start vertex s;
Output: key-field of node contains distance from s;

Dijkstra(s)
    S = new PriorityQueue()
    foreach v ∈ V
        v->key = ∞;
        v->par = NULL;
        v->han = S->insert(v);
    s->key = 0; S->decreaseKey(s->han,0);
    while (!S->empty()) {
        v = S->extractMin();
        foreach x ∈ N[v]
            if (x->key > v->key + w(v,x))
                S->decreaseKey(x->han,v->key+w(v,x));
                x->key = v->key + w(v,x);
                x->par = v;
    }
```

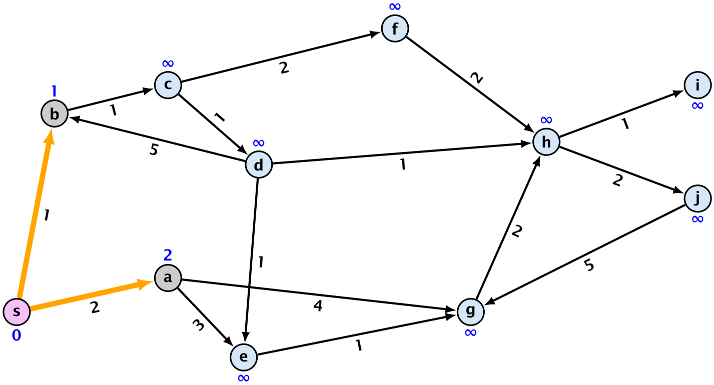
# Beliebige Graphen — nichtnegative Gewichte



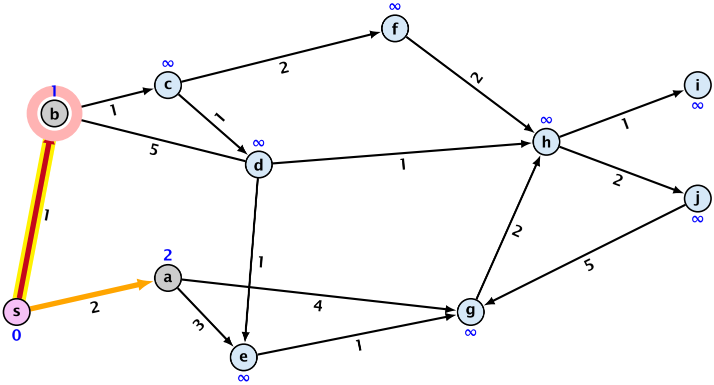
# Beliebige Graphen — nichtnegative Gewichte



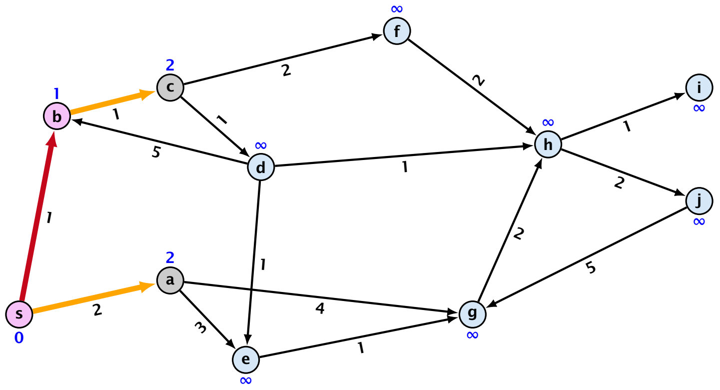
# Beliebige Graphen — nichtnegative Gewichte



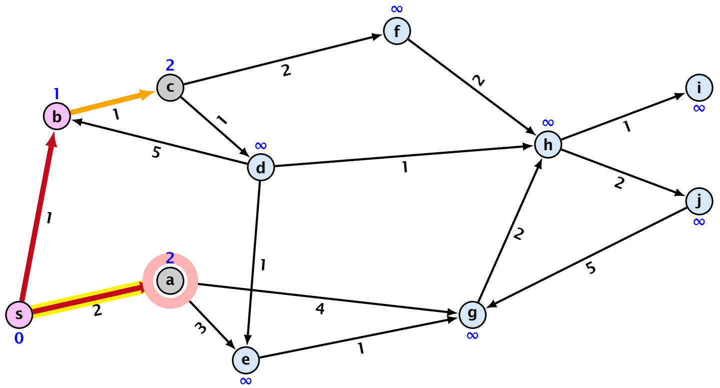
# Beliebige Graphen — nichtnegative Gewichte



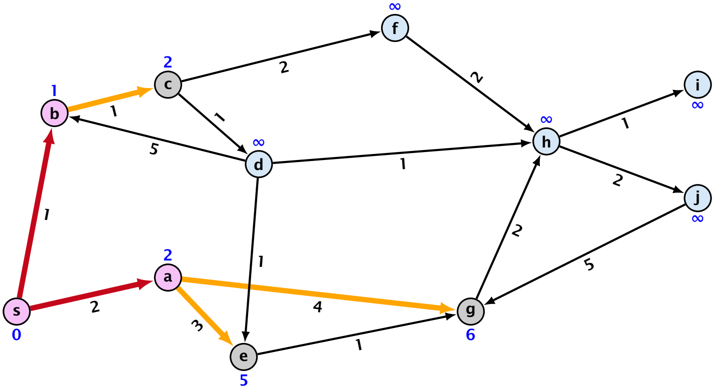
# Beliebige Graphen — nichtnegative Gewichte



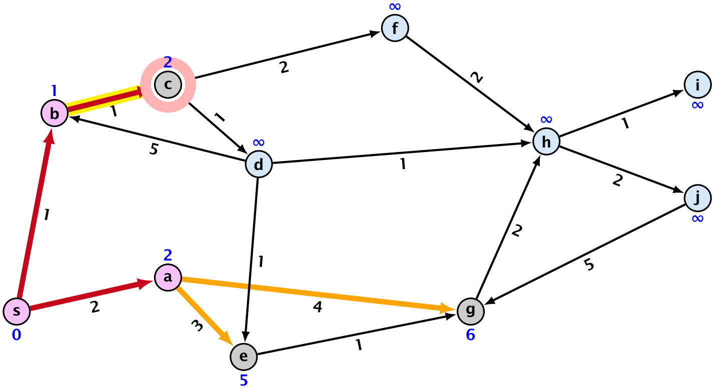
# Beliebige Graphen — nichtnegative Gewichte



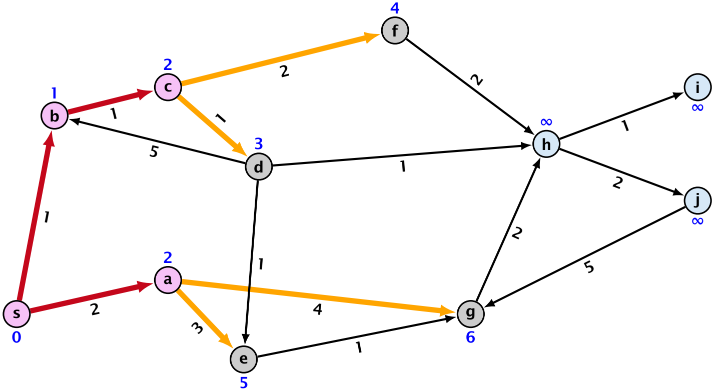
# Beliebige Graphen — nichtnegative Gewichte



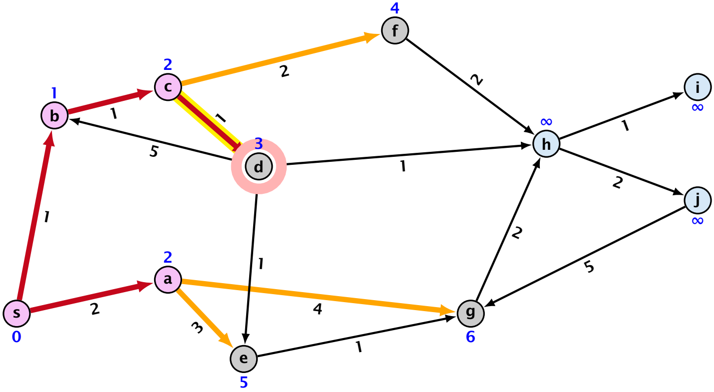
# Beliebige Graphen — nichtnegative Gewichte



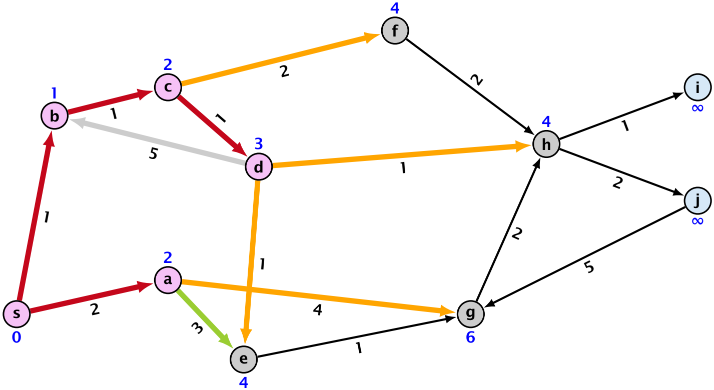
# Beliebige Graphen — nichtnegative Gewichte



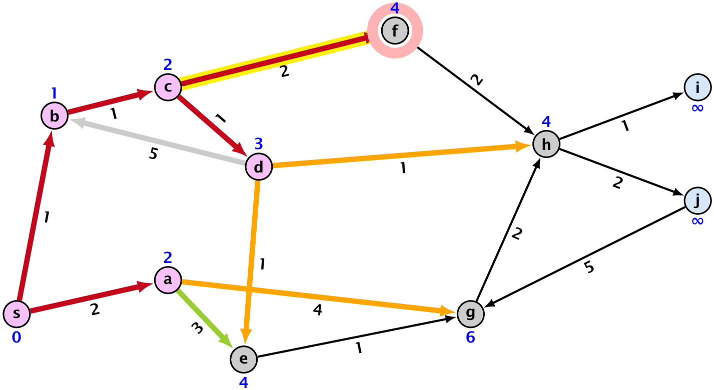
# Beliebige Graphen — nichtnegative Gewichte



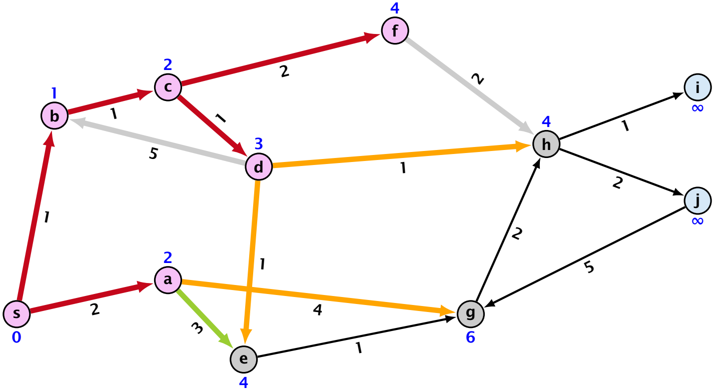
# Beliebige Graphen — nichtnegative Gewichte



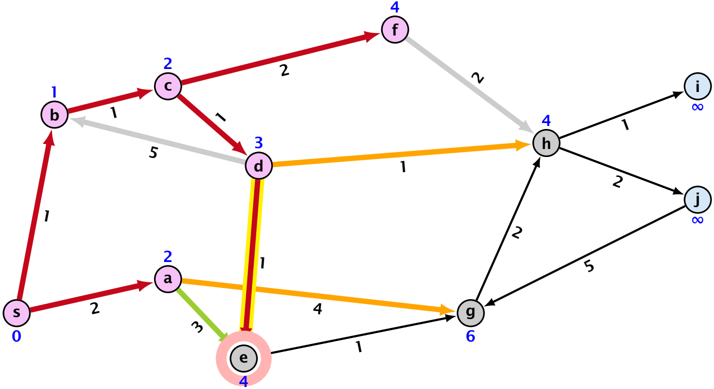
# Beliebige Graphen — nichtnegative Gewichte



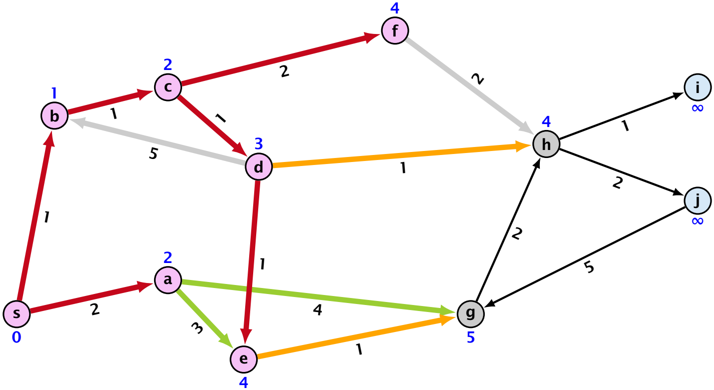
# Beliebige Graphen — nichtnegative Gewichte



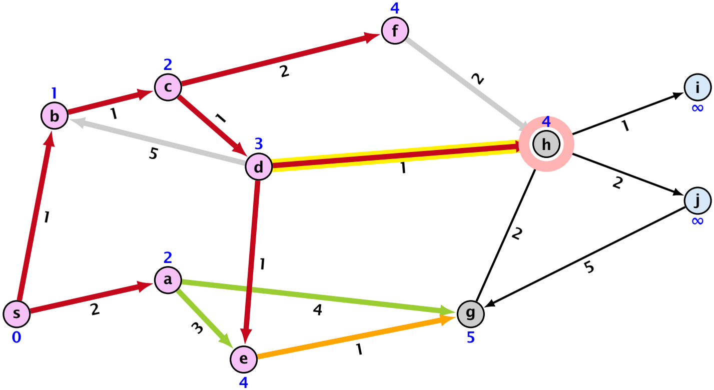
# Beliebige Graphen — nichtnegative Gewichte



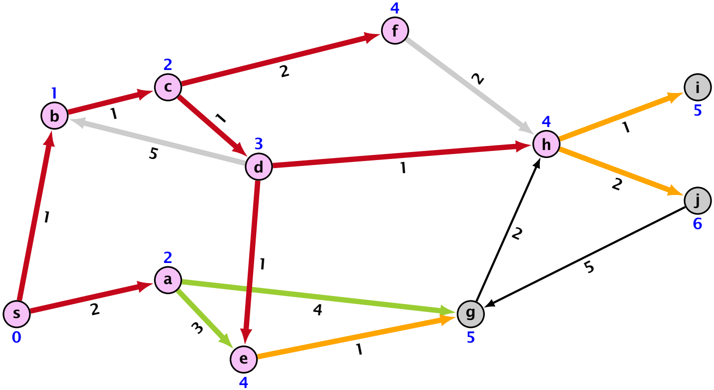
# Beliebige Graphen — nichtnegative Gewichte



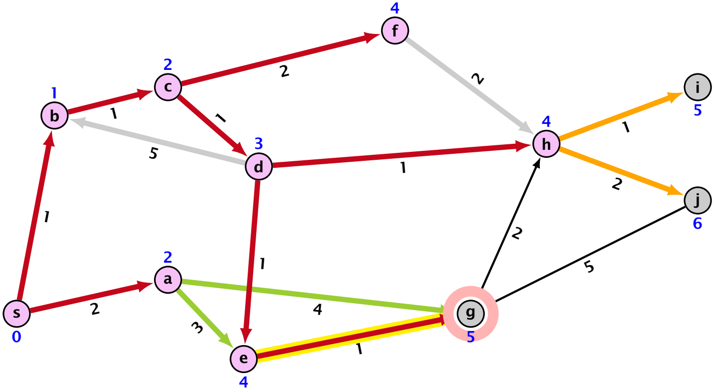
# Beliebige Graphen — nichtnegative Gewichte



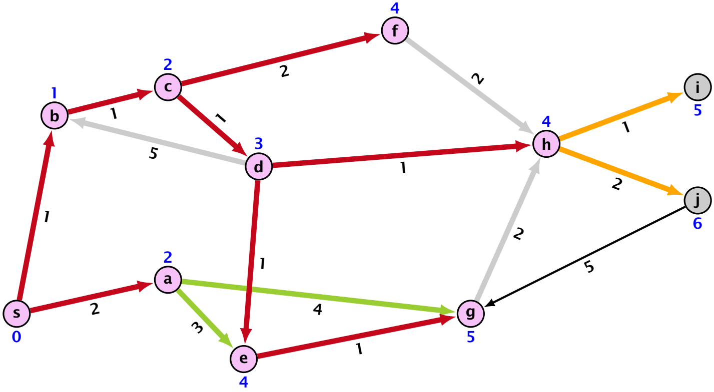
# Beliebige Graphen — nichtnegative Gewichte



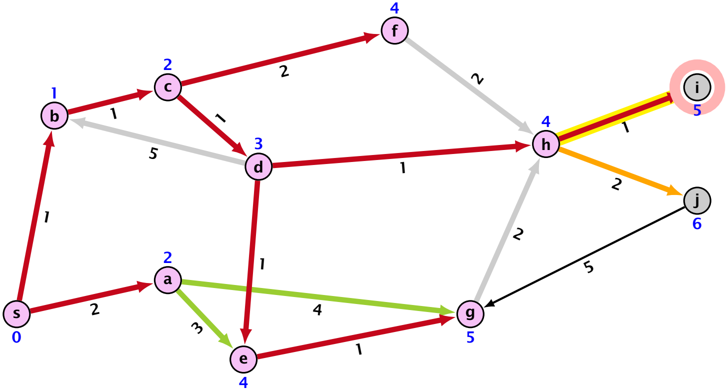
# Beliebige Graphen — nichtnegative Gewichte



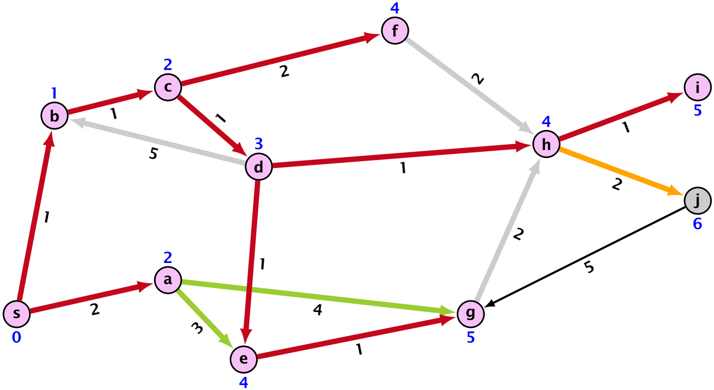
# Beliebige Graphen — nichtnegative Gewichte



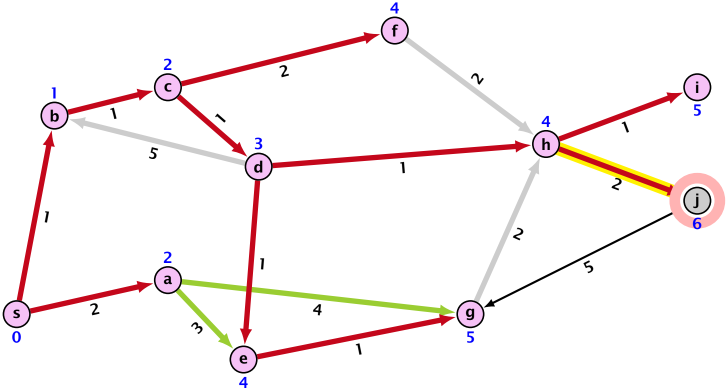
# Beliebige Graphen — nichtnegative Gewichte



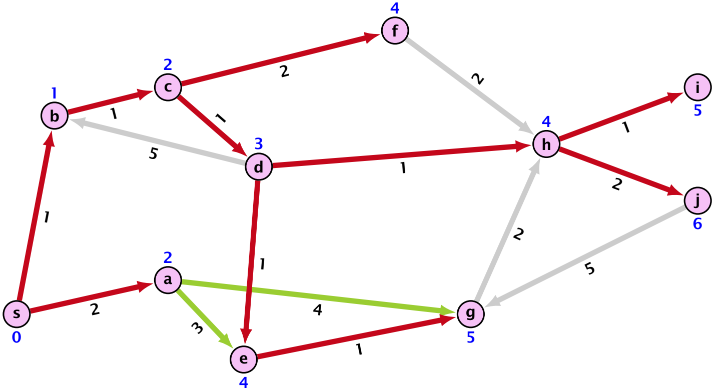
# Beliebige Graphen — nichtnegative Gewichte



# Beliebige Graphen — nichtnegative Gewichte



# Beliebige Graphen — nichtnegative Gewichte



# Warum funktioniert das?

**Invariante:**
Sei $A$ die Menge der Knoten, die schon aus der PQ entfernt wurden.

- **A** Das Distanzlabel $x.\text{key}$ erfüllt $x.\text{key} \leq w(P)$ für alle $s$-$x$ Pfade $P$ die **entweder** mit Kante $(a, x)$, $a \in A$ enden **oder** leer sind (nur aus Knoten $s$ bestehen).
- **B** Knoten in $A$ haben ihre korrekte Distanz.

Außerdem nutzen wir, dass $x.\text{key}$ eine obere Schranke an die Länge eines kürzesten $s$-$x$ Pfades ist (folgt da wir nur Relaxierungen durchführen).

**Initialisierung:**

**A** Initial ist $A$ leer; d.h. $s$ ist der einzig erlaubte Pfad. Deshalb ist Invariante erfüllt wenn man $s$ Distanz $0$ gibt and allen anderen Knoten Distanz $\infty$.

**B** Gilt, da $A$ leer ist.

**Initialisierung:**

**A** Initial ist $A$ leer; d.h. $s$ ist der einzig erlaubte Pfad. Deshalb ist Invariante erfüllt wenn man $s$ Distanz $0$ gibt and allen anderen Knoten Distanz $\infty$.

**B** Gilt, da $A$ leer ist.

**Beibehaltung der Invariante:**

**B** Wir nehmen den Knoten $v$ mit kleinstem key-Wert. Ein Pfad $P$ von $s$ nach $v$ muss Länge mindestens $v.\mathrm{key}$ haben.

Dies gilt, da der Pfad wenn er $A$ verläßt (zum Knoten $x$) schon Länge mindestens $x.\mathrm{key} \geq v.\mathrm{key}$ hat (durch Invariante A für $x.\mathrm{key}$). Er kann nicht kürzer werden da Kantengewichte nichtnegativ sind.

Deshalb, ist das Distanzlabel für $v$ korrekt (es ist eine obere Schranke und kein Pfad ist kürzer)

D.h. Invariante B gilt für $A' = A \cup \{v\}$.

**A** Da $v$ korrekte Distanz hat gilt $x.\mathrm{key} \le w(P)$ für alle Pfade die in Kante $(v, x)$ enden.

Da durch die Invariante gilt, dass $x.\mathrm{key} \le w(P)$ für Pfade, die in $(a, x)$, $a \in A$ enden (oder leer sind), gilt Invariante A für $A' = A \cup \{v\}$.

**Laufzeit:** $m$ decreaseKey Operationen, $n$ Einfügeoperationen, $n$ extractMin() Operationen

Laufzeit: $\mathcal{O}((m+n)\log n)$.

Eine Laufzeit von $\mathcal{O}(m + n\log n)$ kann man durch Implementierung der Prioritätswarteschlange mit Fibonacciheaps erreichen.

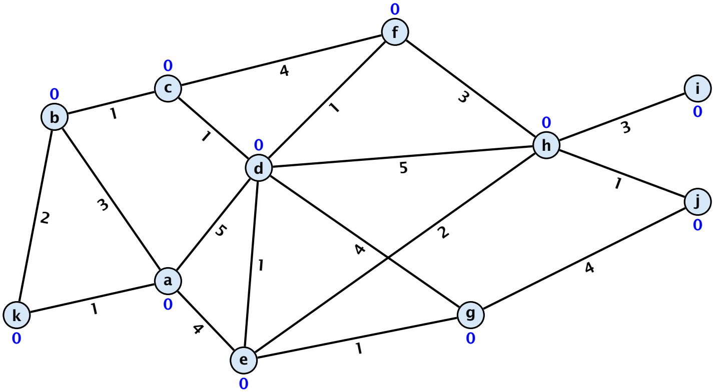
**Laufzeit:** $m$ decreaseKey Operationen, $n$ Einfügeoperationen, $n$ extractMin() Operationen

Laufzeit: $\mathcal{O}((m+n)\log n)$.

Eine Laufzeit von $\mathcal{O}(m + n\log n)$ kann man durch Implementierung der Prioritätswarteschlange mit Fibonacciheaps erreichen.

# Minimaler Spannbaum

Welche Kanten sollte man wählen um alle Knoten zu verbinden?




Minimiere Kosten!!!

# Minimaler Spannbaum

**Eingabe:**

- ungerichteter zusammenhängender graph $G = (V, E)$
- positive Kantengewichte (Kosten) $w : E \to \mathbb{R}^+$

**Ausgabe:**

- Teilmenge $T \subseteq E$ s.t. $G = (V, T)$ verbunden und $w(T) = \sum_{e \in T} w(e)$ minimal

**Beobachtung:**

- da Kantengewichte positiv ist $T$ ein Baum

# Minimaler Spannbaum

**Lemma 1**

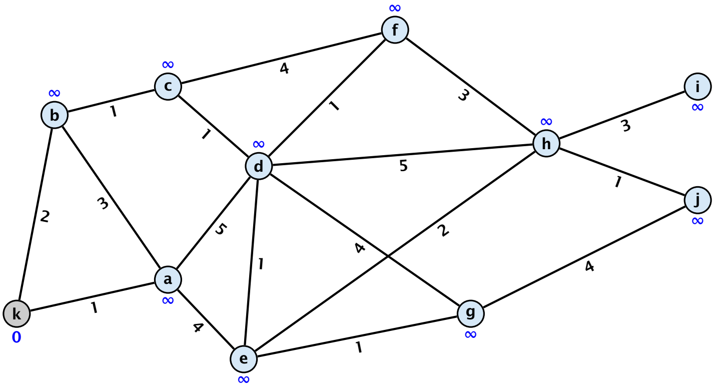
*Sei $X, Y$ eine Partionierung von $V$ (d.h., $X \cup Y = V$ and $X \cap V = \emptyset$), und sei $e = (x, y)$ eine billigste Kante zwischen $X$ und $Y$ Dann existiert ein Minimaler Spannbaum der $e$ enthält.*

# Prims MST-Algorithmus

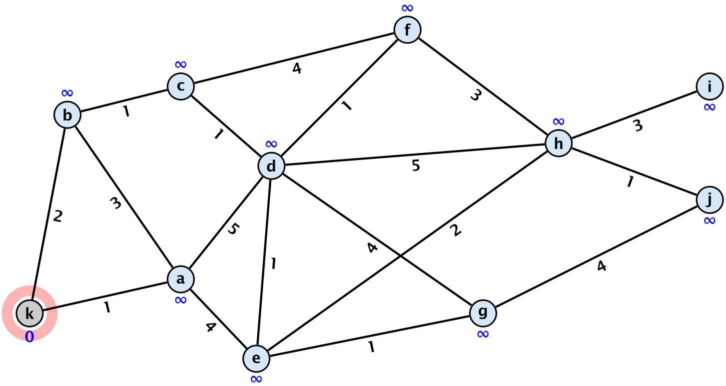
```
Input: weighted graph G=(V,E,w); start vertex s;
Output: parent-fields of nodes encode MST;

PrimMST(s)
    S = new PriorityQueue()
    foreach v ∈ V
        v->key = ∞;
        v->par = NULL;
        v->han = S->insert(v);
    s->key = 0; S->decreaseKey(s->han,0);
    while (!S->empty()) {
        v = S->extractMin();
        foreach x ∈ N[v]
            if (x->key > w(v,x))
                S->decreaseKey(x->han,w(v,x));
                x->key = w(v,x);
                x->par = v;
    }
```

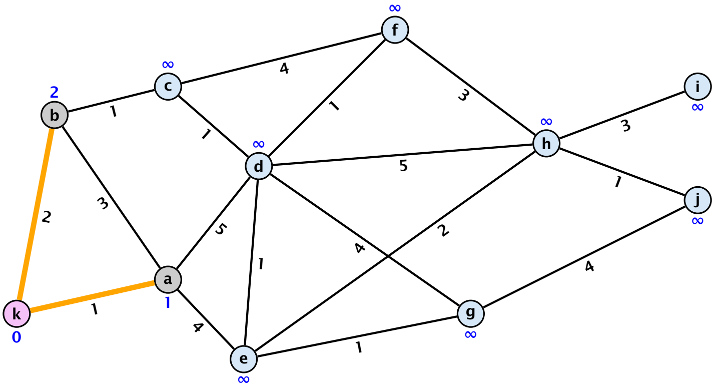
# Minimaler Spannbaum (Prim)



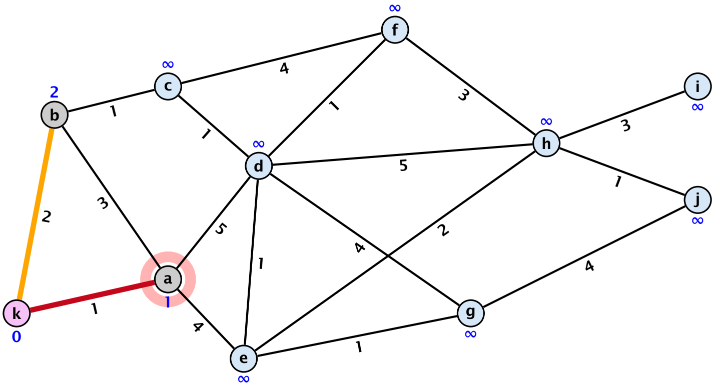
# Minimaler Spannbaum (Prim)



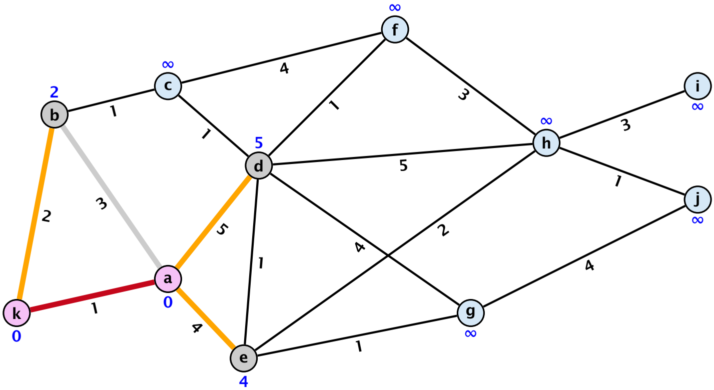
# Minimaler Spannbaum (Prim)



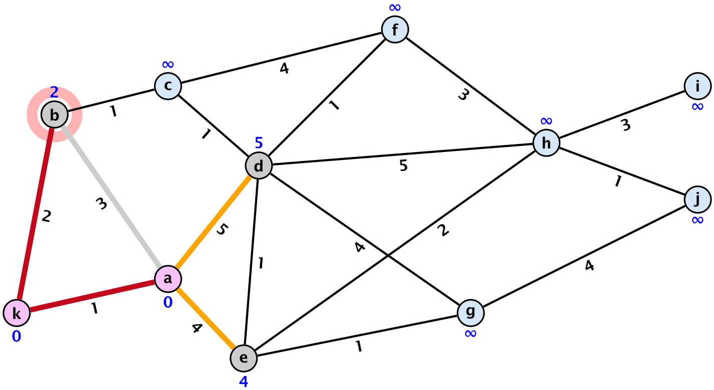
# Minimaler Spannbaum (Prim)



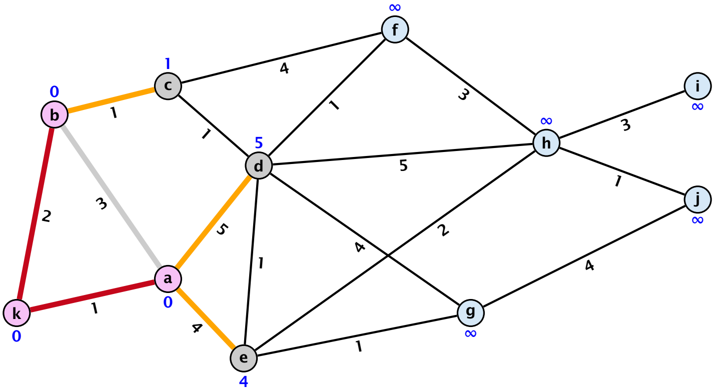
# Minimaler Spannbaum (Prim)



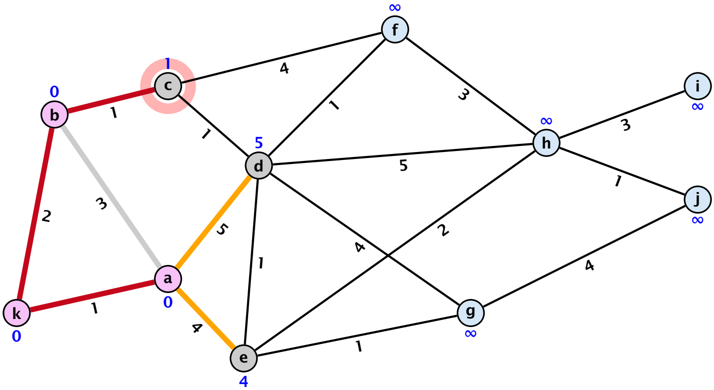
# Minimaler Spannbaum (Prim)



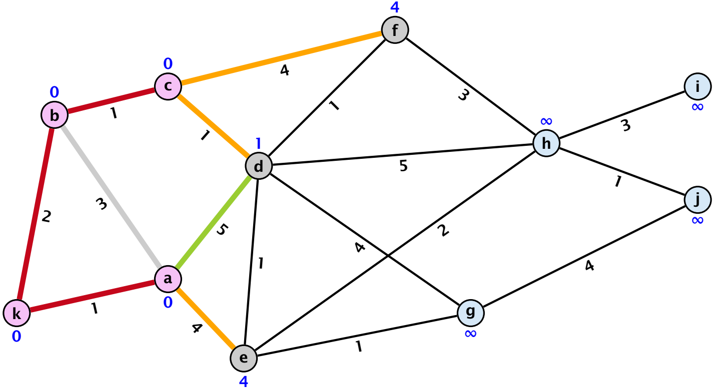
# Minimaler Spannbaum (Prim)



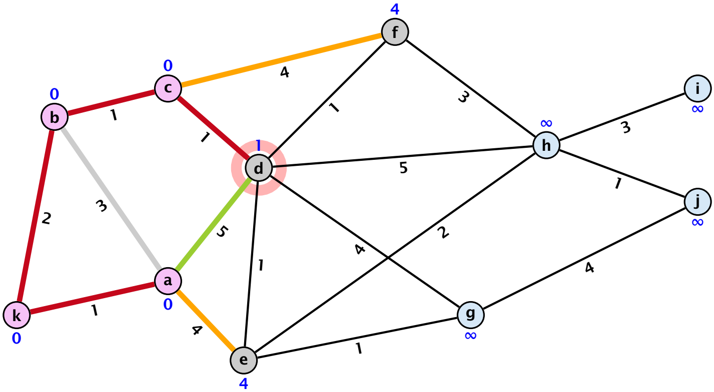
# Minimaler Spannbaum (Prim)



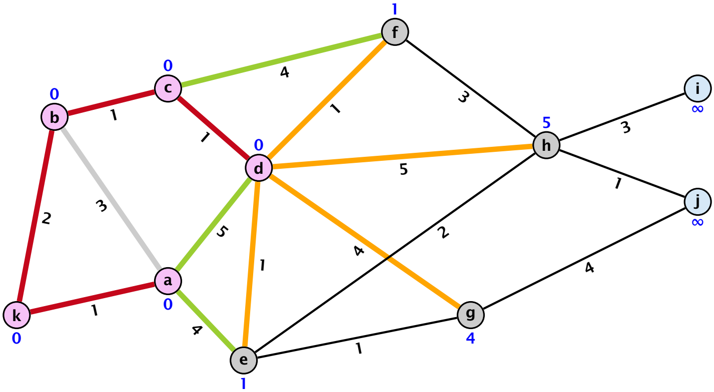
# Minimaler Spannbaum (Prim)



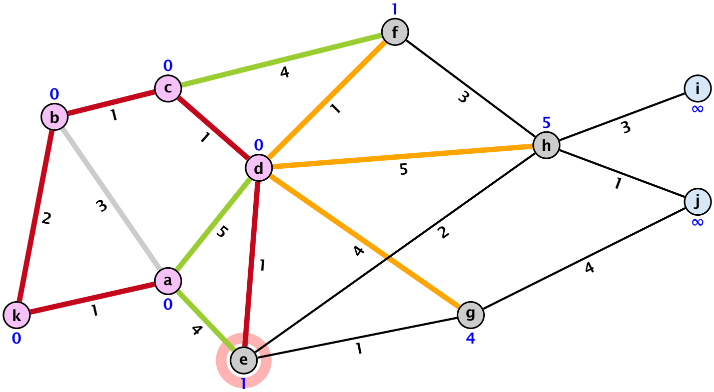
# Minimaler Spannbaum (Prim)



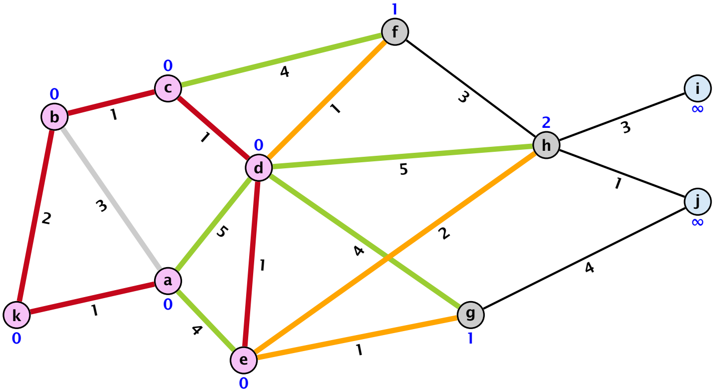
# Minimaler Spannbaum (Prim)



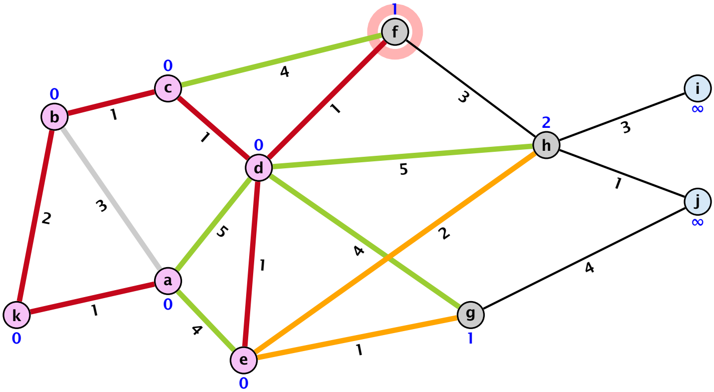
# Minimaler Spannbaum (Prim)



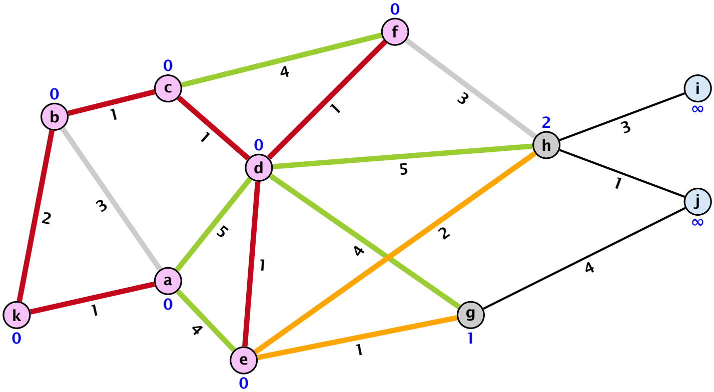
# Minimaler Spannbaum (Prim)



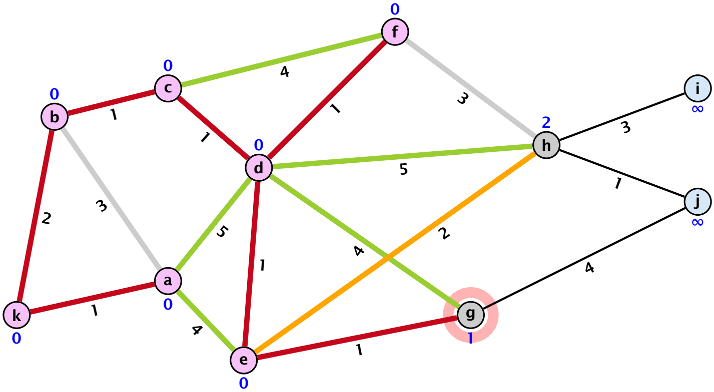
# Minimaler Spannbaum (Prim)



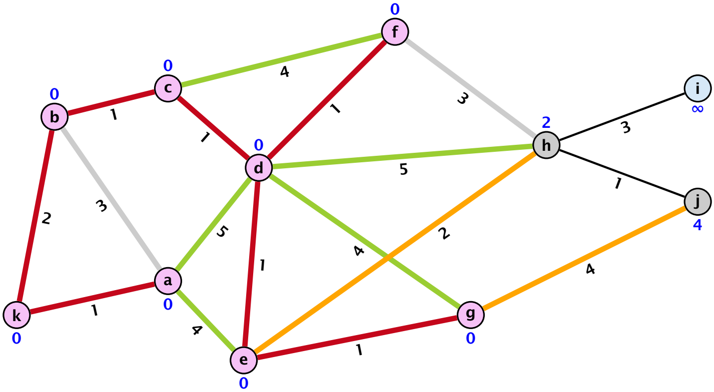
# Minimaler Spannbaum (Prim)



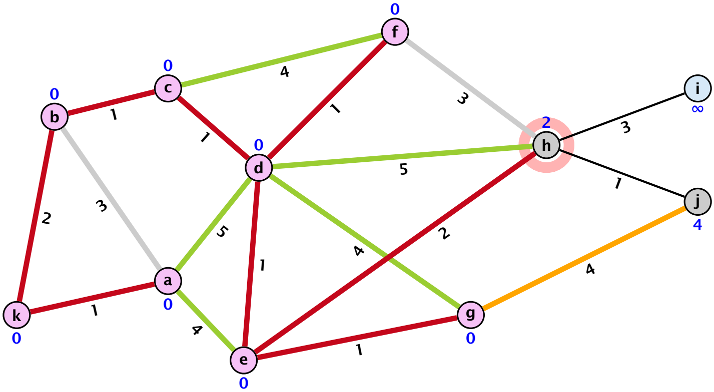
# Minimaler Spannbaum (Prim)



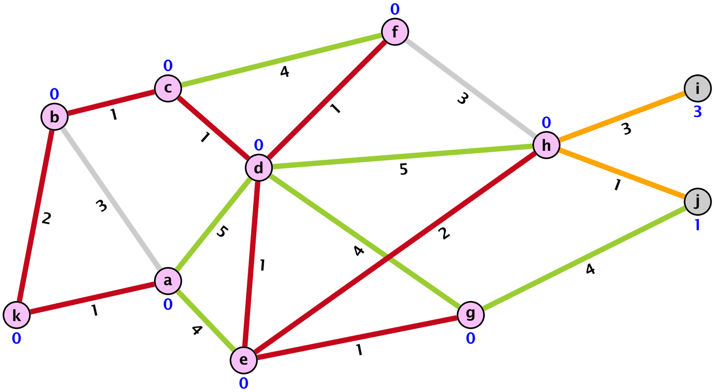
# Minimaler Spannbaum (Prim)



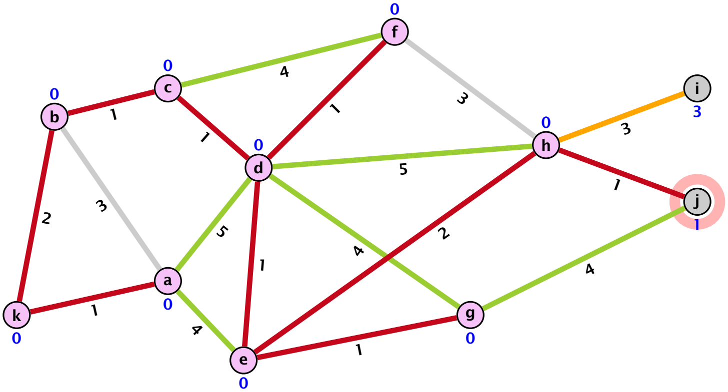
# Minimaler Spannbaum (Prim)



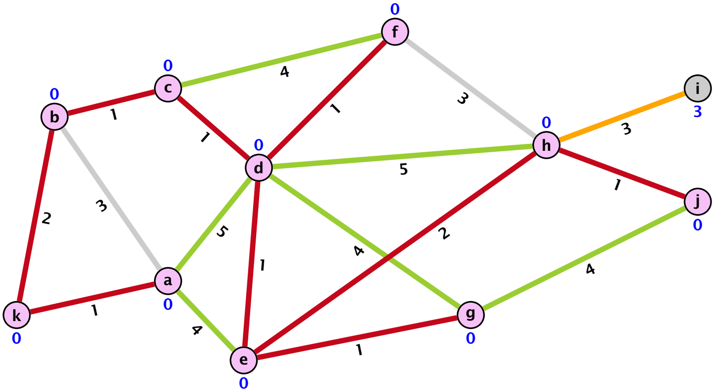
# Minimaler Spannbaum (Prim)



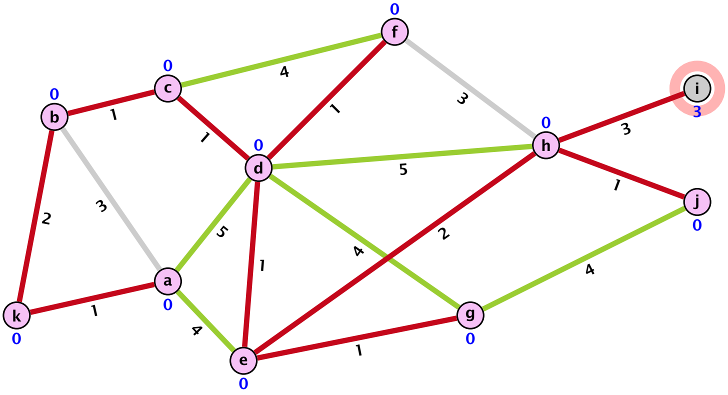
# Minimaler Spannbaum (Prim)



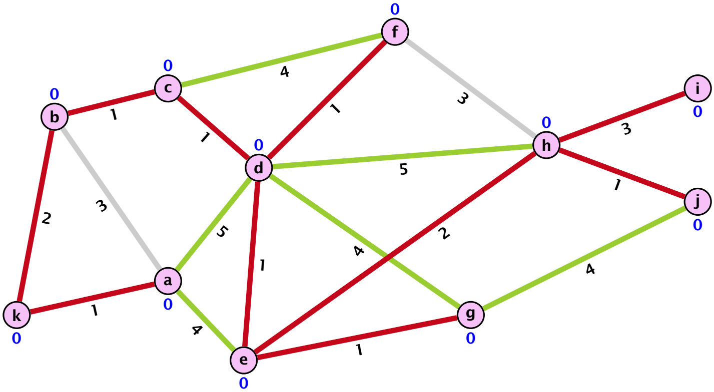
# Minimaler Spannbaum (Prim)



# Minimaler Spannbaum (Prim)

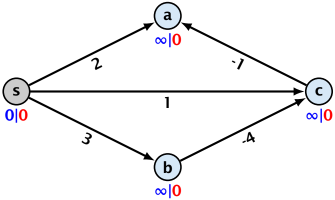


# Minimaler Spannbaum (Prim)



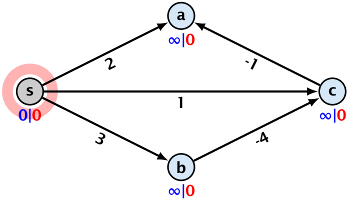
# Negative Kantengewichte

Dijkstra funktioniert nicht:



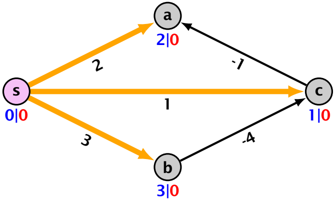
# Negative Kantengewichte

Dijkstra funktioniert nicht:



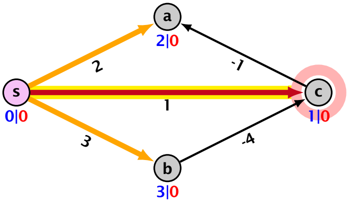
# Negative Kantengewichte

Dijkstra funktioniert nicht:



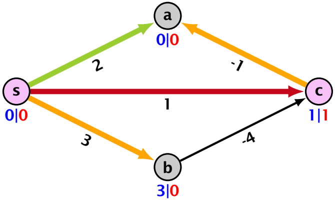
# Negative Kantengewichte

Dijkstra funktioniert nicht:



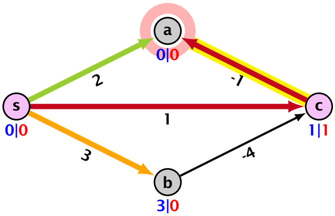
# Negative Kantengewichte

Dijkstra funktioniert nicht:



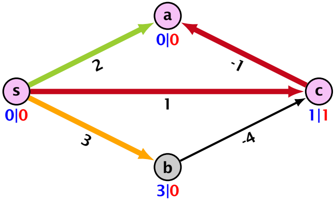
# Negative Kantengewichte

Dijkstra funktioniert nicht:



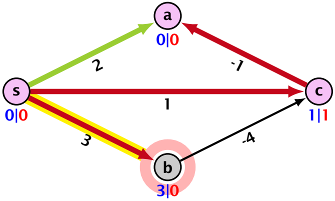
# Negative Kantengewichte

Dijkstra funktioniert nicht:



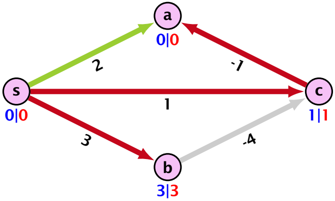
# Negative Kantengewichte

Dijkstra funktioniert nicht:



# Negative Kantengewichte

Dijkstra funktioniert nicht:



# Negative Kantengewichte

## Lemma 2

*Für jeden Knoten $x$ mit $d(s,x) \neq \pm\infty$, existiert ein kürzester Weg mit höchstens $n-1$ Kanten zwischen $s$ und $x$.*

**Beweis:**

- sei $x$ ein Knoten mit $d(s,x) \neq \pm\infty$; dann existiert kürzester Pfad; sei $P$ solch ein Pfad mit wenigsten Kanten
- angenommen, $P$ hat $\geq n$ Kanten (d.h. $\geq n+1$ Knoten)
- dann enthält $P$ einen gerichteten Zyklus $Z$
- wenn $Z$ negatives Gewicht hat gilt $d(s,x) = -\infty$ (↯)
- andernfalls entfernen wir $Z$ aus $P$ und erhalten ein Pfad der Länge höchstens $w(P)$ hat mit weniger hops (↯)

# Bellman-Ford

Für $n - 1$ Phasen relaxiert man alle Kanten.

```
Input: weighted graph G=(V,E,w); start vertex s;
Output: key-field of very node contains distance from s

NaiveBellmanFord(s)
    foreach v ∈ V
        v->key = ∞;
    s->key = 0;
    for i = 1 to n-1
        foreach (x,y) ∈ E
            y->key = min(y->key, x->key + w(x,y))
```

Für hop-minimalen kürzesten $s$-$x$ Pfad $P$ existiert Teilfolge von Relaxierungen, die $P$ von $s$ nach $x$ abarbeitet.

$\Longrightarrow$ alle Distanzen sind am Ende korrekt, falls man keine negativen Kreise hat

# Bellman-Ford

Wie finden wir Knoten, die von $s$ aus über einen Pfad erreichbar sind der einen Knoten eines negativen Kreises enthält? (Das sind genau die Knoten die Distanz $-\infty$ haben)

# Bellman-Ford

Wir fügen eine weitere Phase hinzu.

**Lemma 3**

*Jeder von $s$ erreichbare negative Zyklus enthält einen Knoten, der in Phase $n$ sein Distanzlabel ändert.*

**Beweis:**

- Sei $Z$ negativer Zyklus (erreichbar von $s$). Nach Phase $n-1$ haben alle Knoten des Zyklus endliches Label.
- Angenommen kein Knoten in $Z$ ändert sein Label in Phase $n$. Das bedeutet

$$v_{i+1} \to \mathrm{key} \le v_i \to \mathrm{key} + w(v_i, v_{i+1})$$

# Bellman-Ford

D.h. wir müssen nur all Knoten markieren die in Runde $n$ ihr Label ändern (und alle Knoten, die von diesen erreichbar sind).

Dies geschieht durch folgende Routine:

```
mark(x)
    if (x->key != −∞)
        x->key = −∞;
        for ((x,y) ∈ E) mark(y)
```

Falls wir $\text{mark}(x)$ für alle Knoten $x \in S \subseteq V$ (für beliebige Teilmenge $S$) aufrufen benötigt dies Zeit $\mathcal{O}(|S| + m)$, da jede Kante nur einmal geprüft wird.

# Bellman-Ford

```
BellmanFord(s)
    foreach v ∈ V
        v->key = ∞;
        v->par = NULL;
    s->key = 0;
    for i = 1 to n-1
        foreach (x,y) ∈ E
            if (y->key > x->key + w(x,y))
                y->key = x->key + w(x,y);
                y->par = x;
    foreach (x,y) ∈ E
        if (y->key > x->key + w(x,y))
            y->par = x;
            mark(y)
```

Laufzeit: $\mathcal{O}(mn)$

# Bellman-Ford

**Invariante:**

Nach der $\ell$-ten Runde zeigt parent-Zeiger von $x$ auf den Vorgänger von $x$ auf einem kürzesten $s$-$x$ Pfad mit höchstens $\ell$ Kanten.

$x.\text{parent} = \emptyset$ bedeutet **entweder**, dass es keinen $s$-$x$ Pfad mit höchstens $\ell$ Kanten gibt **oder** dass der kürzeste dieser Pfade leer ist ($s = x$).

**Nach Terminierung:**

Für Knoten $v$ mit $d(s,v) \neq \pm\infty$ führen die parent pointer von $x$ nach $s$ (rückwärts entlang kürzestem $s$-$v$ Pfad).

Wenn man den parent-Zeigern von Knoten $v$ mit $d(s,v) = -\infty$ folgt gelangt man zu einem negativen Kreis (dies kann man nutzen um einen negativen Kreis zu finden).

# Bellman-Ford

**Verbesserung**

- Benutze Queue, die Knoten speichert zu denen ein kürzerer Pfad gefunden wurde (und deren ausgehende Kanten deshalb relaxiert werden müssen).
- Wiederhole:
  Entferne Element $x$ aus der Queue und relaxiere ausgehende Kanten.

  Falls Relaxierung entlang Kante $(x, y)$ erfolgreich wird $y$ in die Queue aufgenommen (falls noch nicht enthalten).

  Runde/Phase endet wenn wenn die zu Anfang der Phase in der Queue enthaltenen Knoten bearbeitet sind.

# APSP – Floyd Warshall

gegeben:

- Graph mit beliebigen Kantengewichten; keine negativen Kreise

gesucht:

- Distanzen/kürzeste Weg zwischen allen Knotenpaaren.

Naive Strategie

- $n$-mal Bellman-Ford. Laufzeit $\mathcal{O}(m \cdot n^2)$.

# APSP – Floyd Warshall

**Beobachtung:**
geht der kürzeste $u$-$w$ Pfad über $v$ dann sind auch die Teile von $u$ nach $v$ und $v$ nach $w$ kürzeste Pfade.

**Dynamisches Programmieren:**

- Berechne kürzeste Wege $P_A(u,w)$ von $u$ nach $w$, die nur Zwischenknoten aus Menge $A$ benutzen (initial ist Menge $A$ leer).
- Wenn man alle Pfade $P_A(u,w)$ kennt, kann man einfach die Pfade $P_{A\cup\{v\}}(u,w)$ berechnen.
- Am Ende möchten wir $P_V(u,w)$ für alle Paare $u,w$ kennen.

# APSP – Floyd Warshall

```
Input: weighted graph G=(V,E,w);
Output: d[u,v] is distance from u to v; pred[u,v]
        is predecessor of v on shortest path tree from u

FloydWarshall(G)
    for u,v ∈ V
        d[u,v]    = ∞;
        pred[u,v] = NULL;
    for v ∈ V
        d[u,v]    = 0;
    for (u,v) ∈ E
        d[u,v]    = w(u,v);
        pred[u,v] = u;
    for v ∈ V
        for {u,w} ∈ V × V
            if (d[u,w] > d[u,v] + d[v,w])
                d(u,w)    = d(u,v) + d(v,w);
                pred(u,w) = pred(v,w);
```

# APSP – Floyd Warshall

- Komplexität $\mathcal{O}(n^3)$
- funktioniert auch bei Kanten mit negativem Gewicht
- Kreise negativer Länge verfälschen das Ergebnis; können aber durch negative Diagonaleinträge in der Ergebnismatrix erkannt werden.