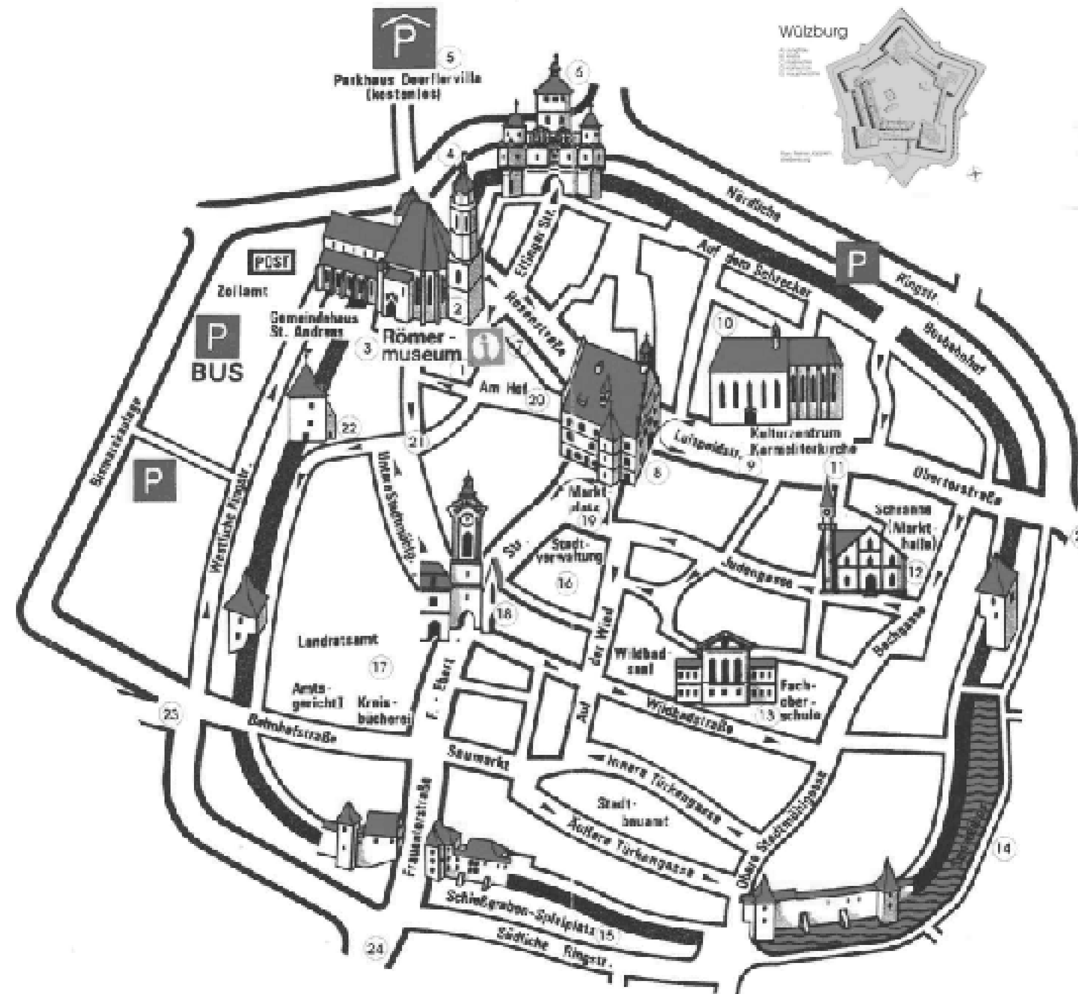
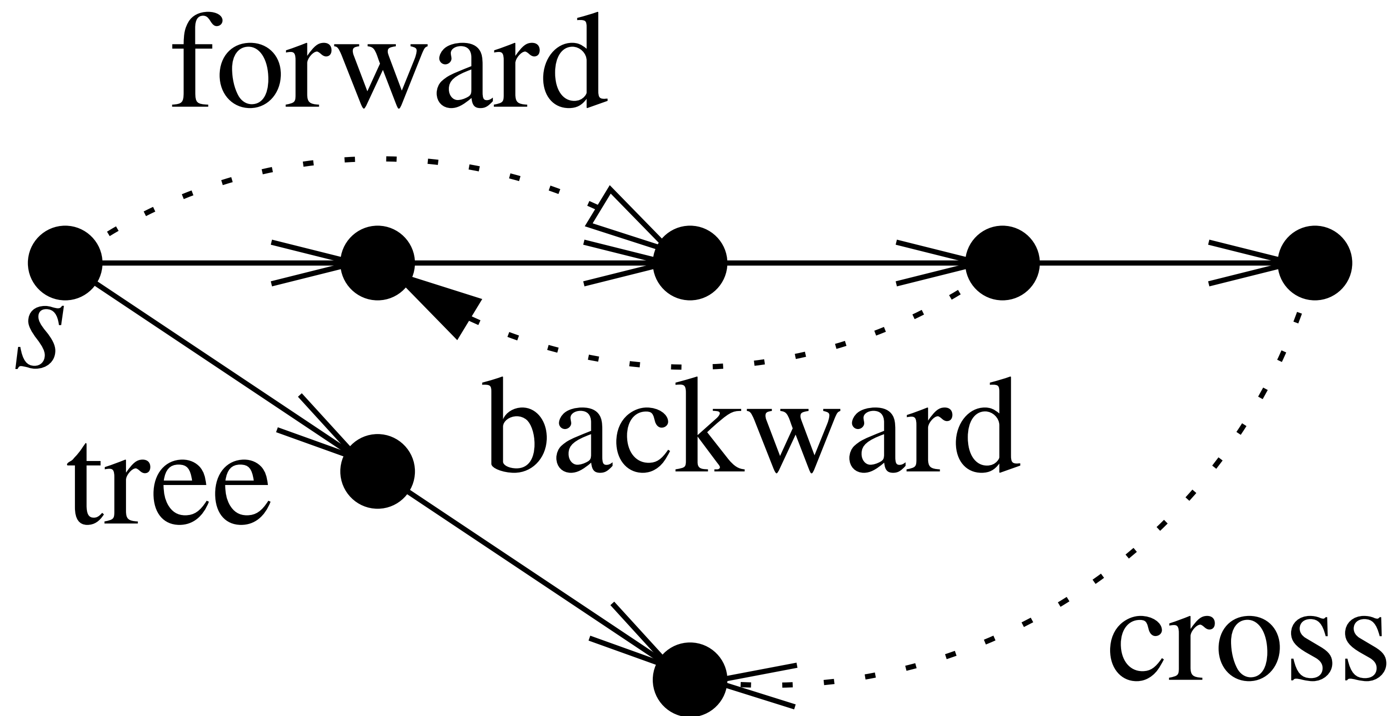
# Kap. 9: Graphtraversierung



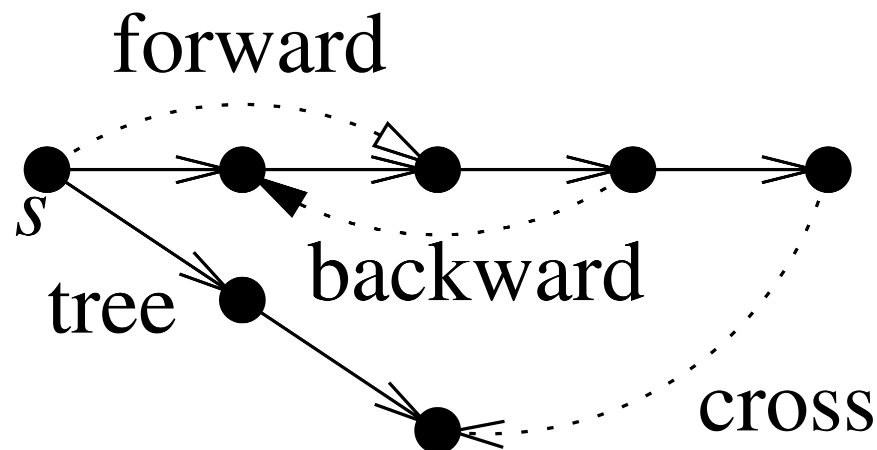

Ausgangspunkt oder Baustein fast jedes nichttrivialen Graphenalgorithmus

# Graphtraversierung als Kantenklassifizierung



# Graphtraversierung als Kantenklassifizierung

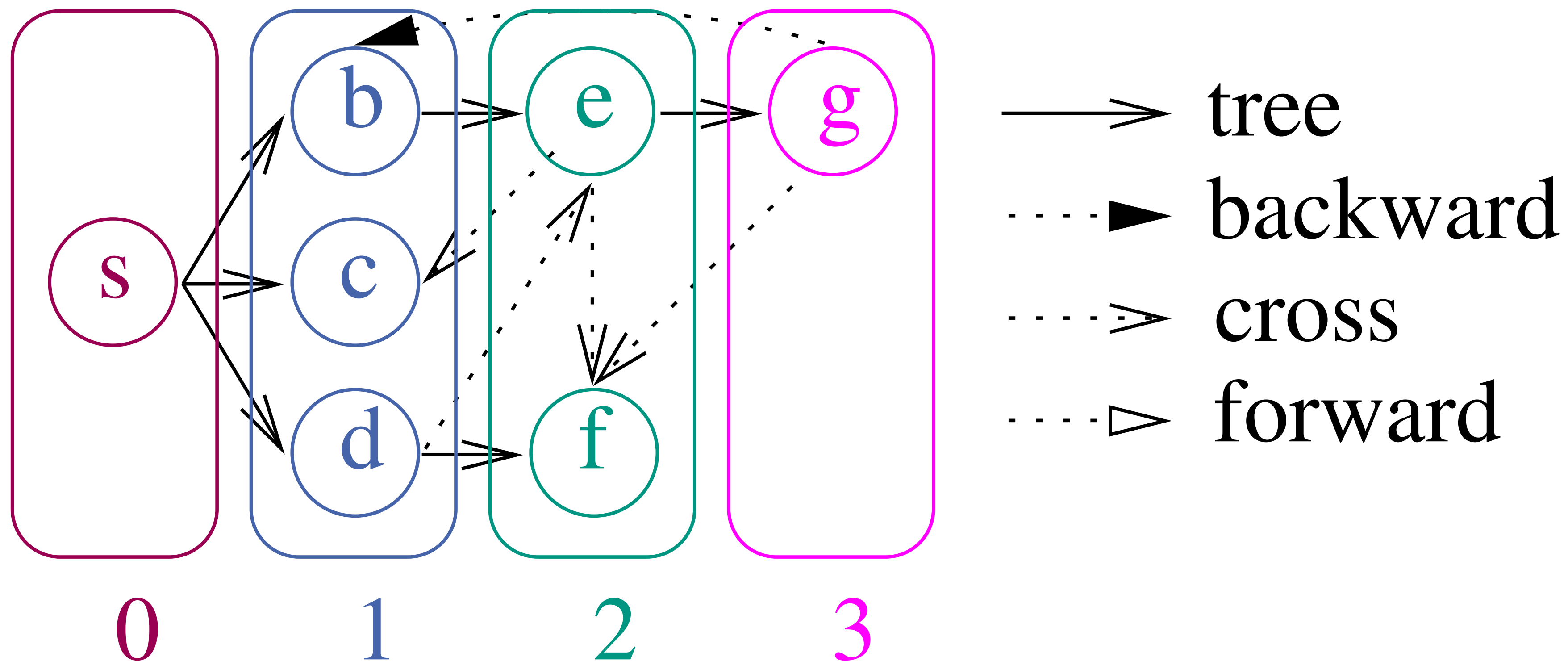
- Baumkanten: Elemente des Waldes, der bei der Suche gebaut wird
- Vorwärtskanten: verlaufen parallel zu Wegen aus Baumkanten
- Rückwärtskanten: verlaufen antiparallel zu Wegen aus Baumkanten
- Querkanten: alle übrigen



# Breitensuche

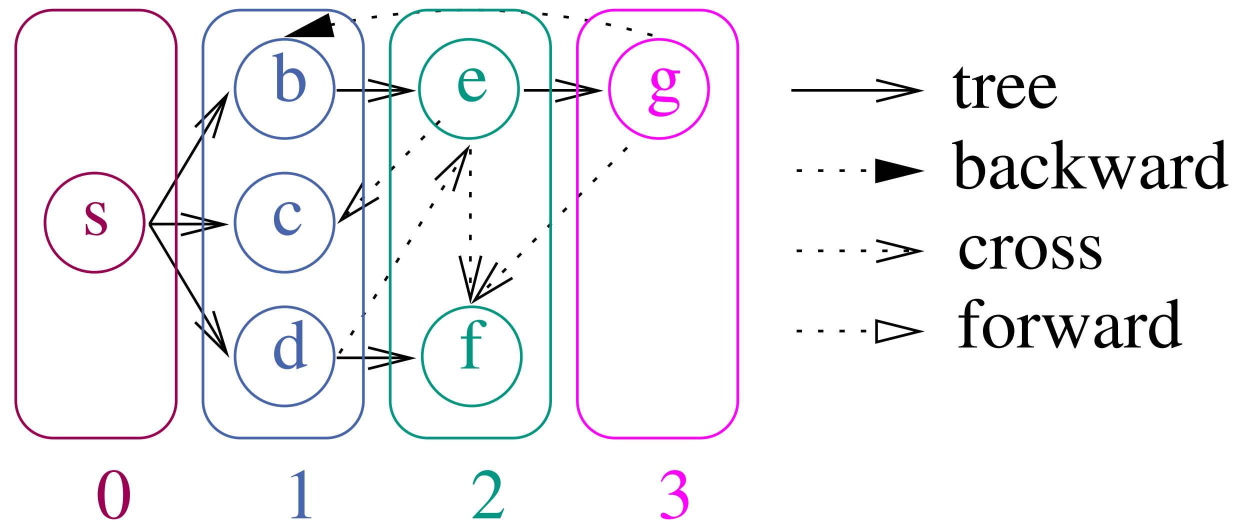
Baue Baum von Startknoten $s$,
der alle von $s$ erreichbaren Knoten
mit möglichst kurzen Pfaden erreicht.

Berechne Abstände:



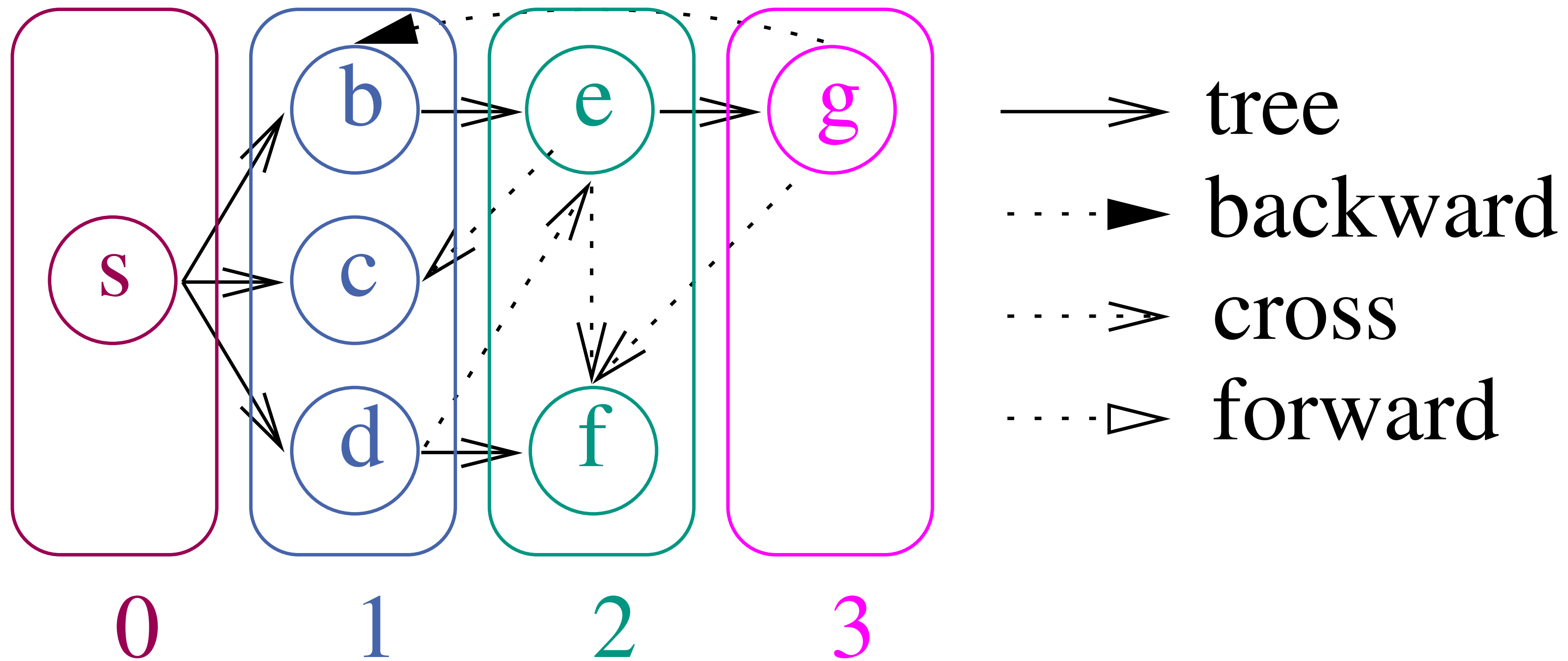
# Breitensuche

- Einfachste Form des Kürzeste-Wege-Problems
- Umgebung eines Knotens definieren
  (ggf. begrenzte Suchtiefe)
- Einfache, effiziente Graphtraversierung
  (auch wenn Reihenfolge egal)



# Breitensuche

Algorithmenidee: Baum Schicht für Schicht aufbauen



**Function** bfs($s$) :

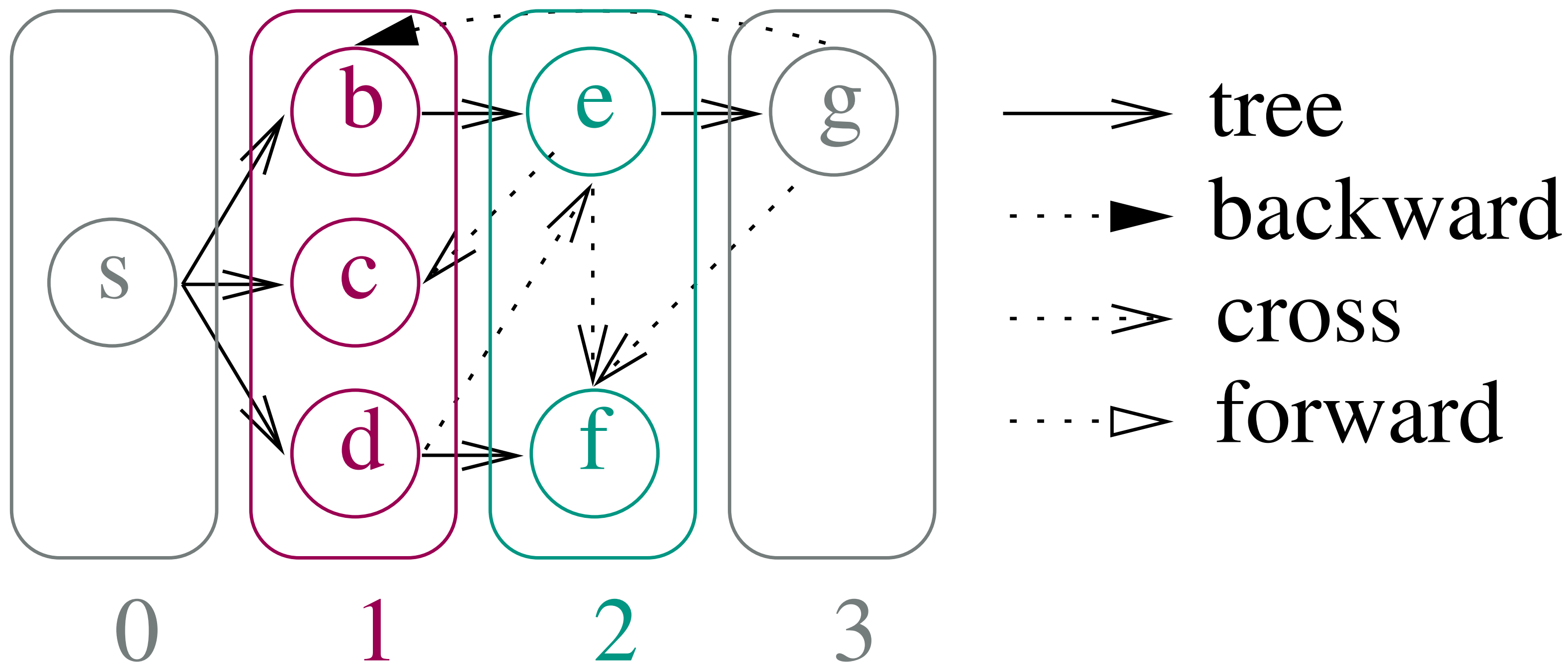
$Q := \langle s \rangle$ // aktuelle Schicht

**while** $Q \neq \langle\rangle$ **do**

exploriere Knoten in $Q$

merke dir Knoten der nächsten Schicht in $Q'$

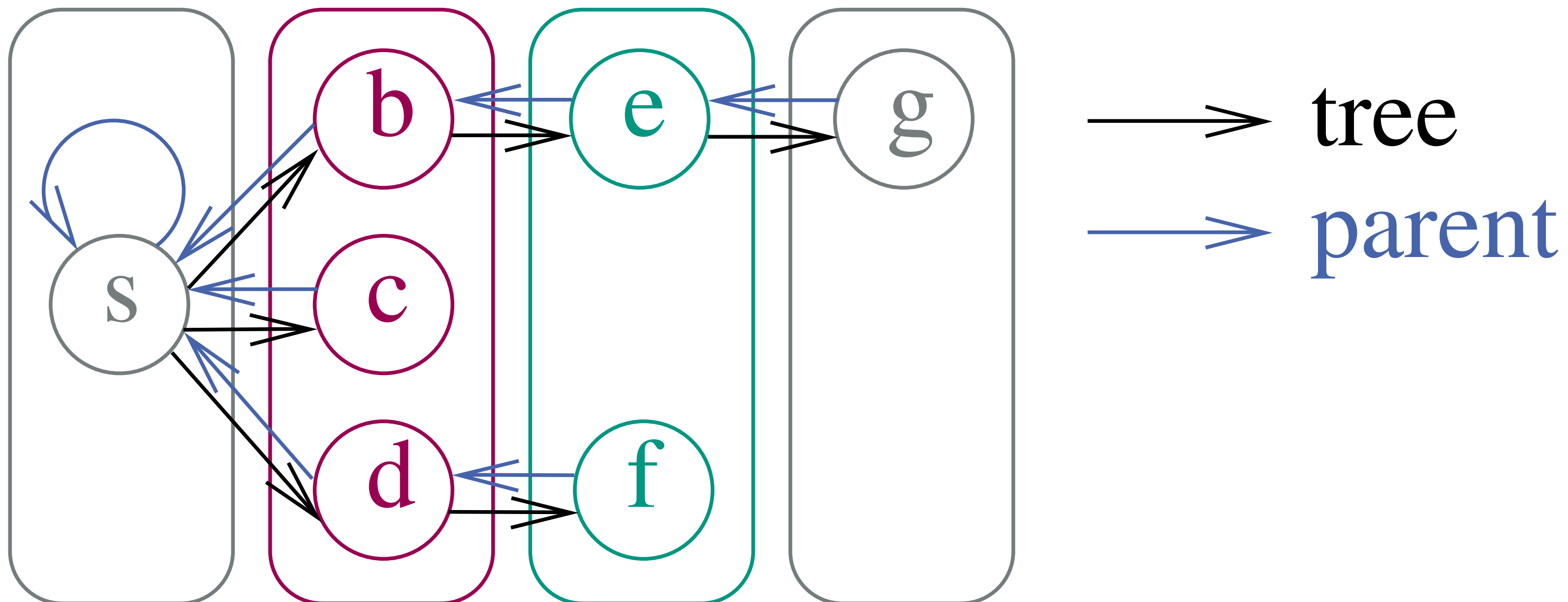
$Q := Q'$



# Repräsentation des Baums

Feld parent speichert Vorgänger.

- noch nicht erreicht: parent$[v] = \bot$
- Startknoten/Wurzel: parent$[s] = s$



**Function** bfs($s$ : NodeId) : (NodeArray **of** NodeId)
$\times$(NodeArray **of** $\mathbb{N}_0 \cup \{\infty\}$)

$d{=}\langle\infty,\ldots,\infty\rangle$ : NodeArray **of** $\mathbb{N}_0 \cup \{\infty\}$; $\quad d[s]{:=}\,0$
parent$=\langle\bot,\ldots,\bot\rangle$ : NodeArray **of** NodeId; $\quad$ parent$[s]{:=}\,s$
$Q = \langle s\rangle, Q' = \langle\rangle$ : Set **of** NodeId $\quad$ // current, next layer
**for** ($\ell{:=}\,0$; $Q \neq \langle\rangle$; $\ell{+}{+}$ )
  **invariant** $Q$ contains all nodes with distance $\ell$ from $s$
  **foreach** $u \in Q$ **do**
    **foreach** $(u,v) \in E$ **do** $\quad$ // scan $u$
      **if** parent$(v) = \bot$ **then** $\quad$ // unexplored
        $Q'{:=}\, Q' \cup \{v\}$
        $d[v]{:=}\,\ell + 1$;
        parent$(v){:=}\, u$
  $(Q,Q'){:=}\,(Q',\langle\rangle)$ $\quad$ // next layer
**return** (parent$, d$) $\quad$ // BFS $= \{(v,w) : w \in V, v = \text{parent}(w)\}$

# Repräsentation von $Q$ und $Q'$ mittels FIFO

$Q, Q' \longrightarrow$ einzelne FIFO-Queue

- Standardimplementierung in anderen Büchern

$+$ „Oberflächlich“ einfach

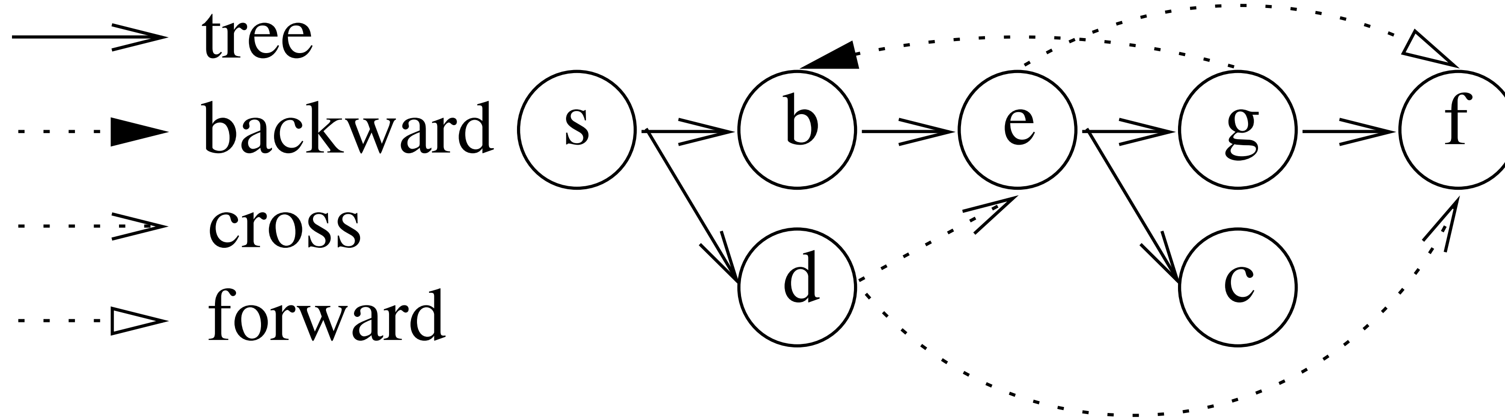
$-$ Korrektheit mglw. weniger evident — Übung!

$=$ Effizient (?) — Übung: ausprobieren!

# Alternative Repräsentation von $Q$ und $Q'$

- Zwei Stapel
- Schleife $1\times$ ausrollen
  **loop** $Q \longrightarrow Q'$; $Q' \longrightarrow Q$
- Beide Stapel in ein Feld der Größe $n$

# Tiefensuche



# Tiefensuchschema für $G = (V, E)$

```
unmark all nodes;
init
foreach s ∈ V do
    if s is not marked then
        mark s                                  // make s a root and grow
        root(s)                                 // a new DFS tree rooted at s
        DFS(s, s)

Procedure DFS(u, v : NodeId)                    // Explore v coming from u
    foreach (v, w) ∈ E do
        if w is marked then traverseNonTreeEdge(v, w)
        else     traverseTreeEdge(v, w)
                 mark w
                 DFS(v, w)
    backtrack(u, v)             // return from v along the incoming edge
```

# DFS-Baum

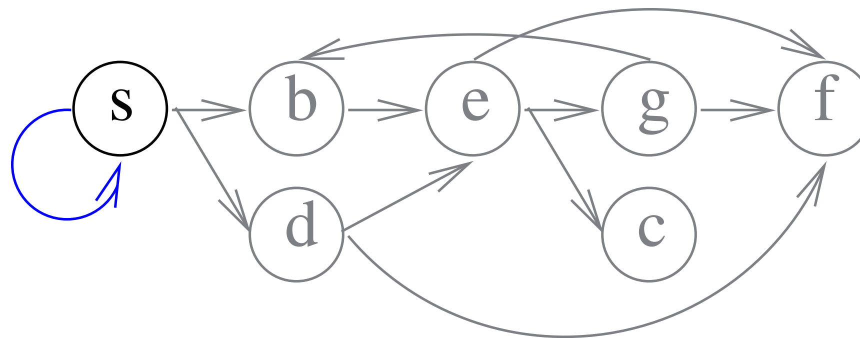
init: parent$=\langle\bot,\ldots,\bot\rangle$ : NodeArray **of** NodeId

root($s$): parent[$s$]:= $s$

traverseTreeEdge($v,w$): parent[$w$]:= $v$

→ tree

→ parent




mark s

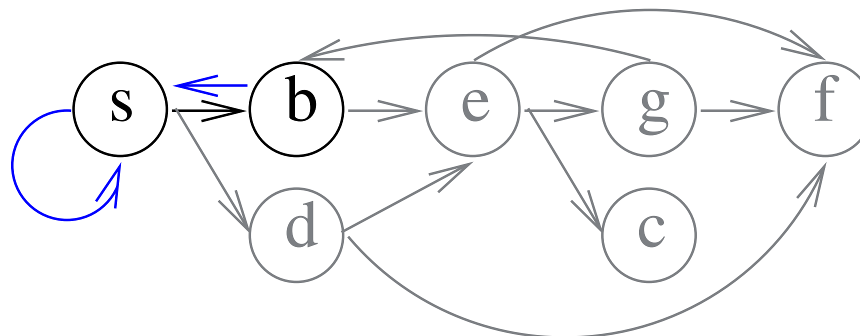
root(s)

dfs(s,s)

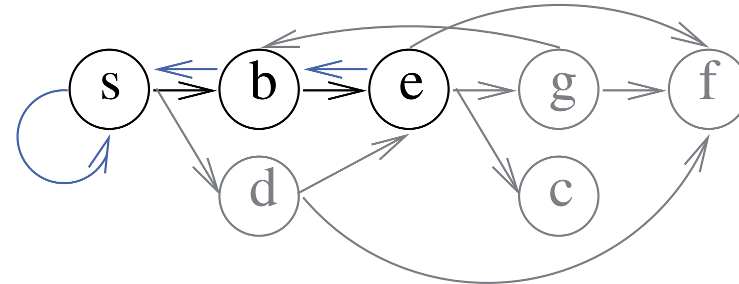
traverseTreeEdge(s,b)

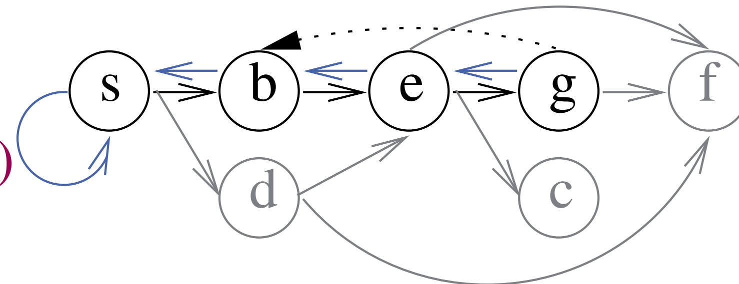
mark b

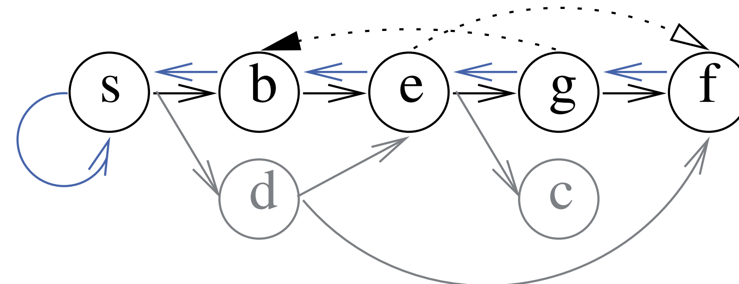
dfs(s,b)

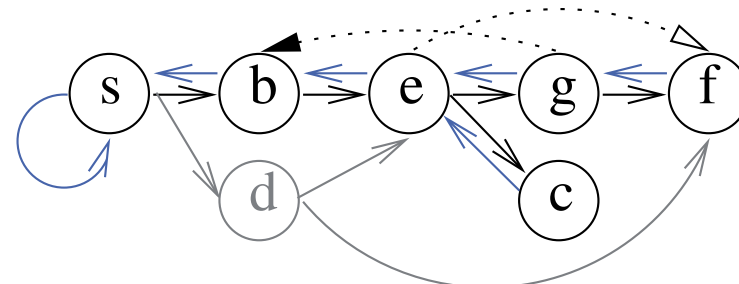


```
dfs(s,b)
  traverseTreeEdge(b,e)
  mark(e)
  dfs(b,e)
    traverseTreeEdge(e,g)
    mark(g)
    dfs(e,g)
      traverseNonTreeEdge(g,b)
      traverseTreeEdge(g,f)
      mark(f)
      dfs(g,f)
        backtrack(g,f)
      backtrack(e,g)
    traverseNonTreeEdge(e,f)
    traverseTreeEdge(e,c)
    mark(c)
    dfs(e,c)
      backtrack(e,c)
    backtrack(b,e)
  backtrack(s,b)
```

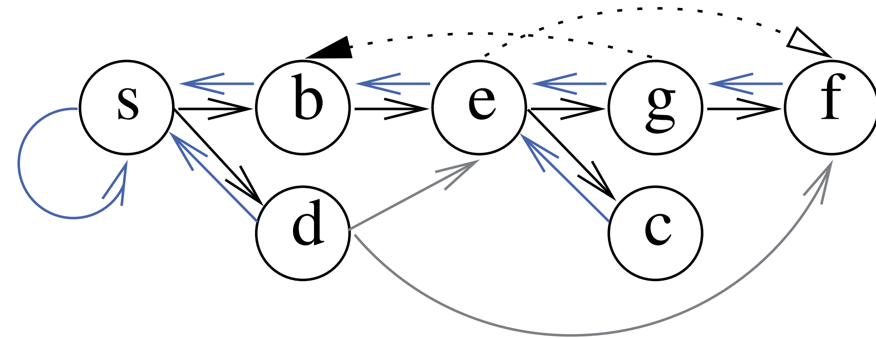


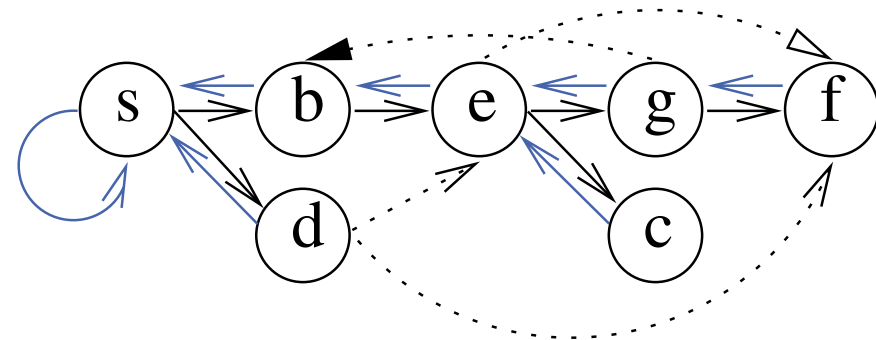



traverseTreeEdge(s,d)
mark(d)
dfs(s,d)
  traverseNonTreeEdge(d,e)
  traverseNonTreeEdge(d,f)
  backtrack(s,d)
backtrack(s,s)



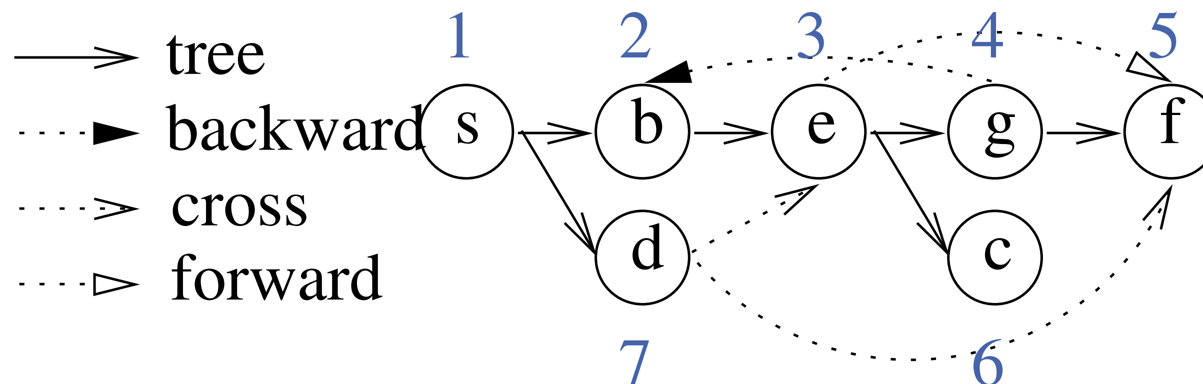
# DFS-Nummerierung

| | |
|---|---|
| init: | dfsPos=1 : $1..n$ |
| root($s$): | dfsNum[$s$]:= dfsPos++ |
| traverseTreeEdge($v, w$): | dfsNum[$w$]:= dfsPos++ |

$$u \prec v :\Leftrightarrow \text{dfsNum}[u] < \text{dfsNum}[v] \ .$$

**Beobachtung:**

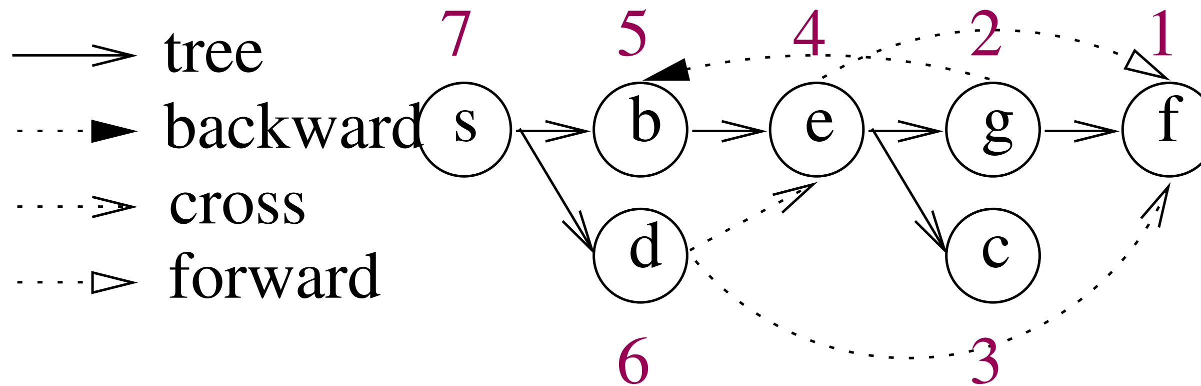
Knoten auf dem Rekursionsstapel sind bzgl. $\prec$ sortiert



# Fertigstellungszeit

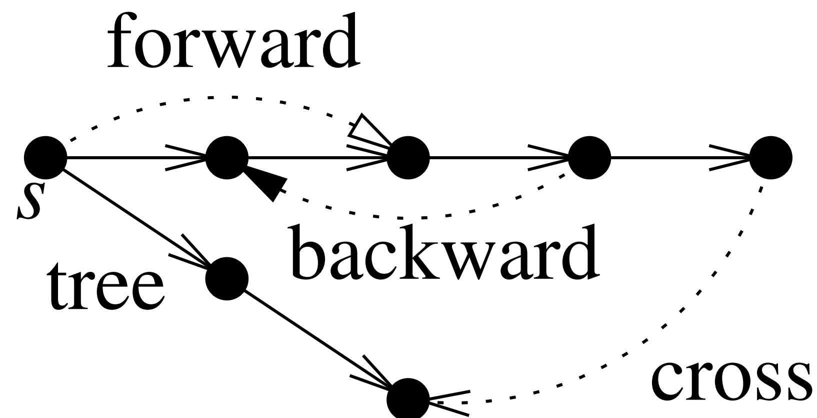
init: finishingTime=1 : 1..$n$

backtrack$(u, v)$: finishTime$[v]$:= finishingTime++



# Kantenklassifizierung bei DFS

| type $(v, w)$ | $\text{dfsNum}[v] < \text{dfsNum}[w]$ | $\text{finishTime}[w] < \text{finishTime}[v]$ | $w$ is marked |
|---|---|---|---|
| tree | yes | yes | no |
| forward | yes | yes | yes |
| backward | no | no | yes |
| cross | no | yes | yes |



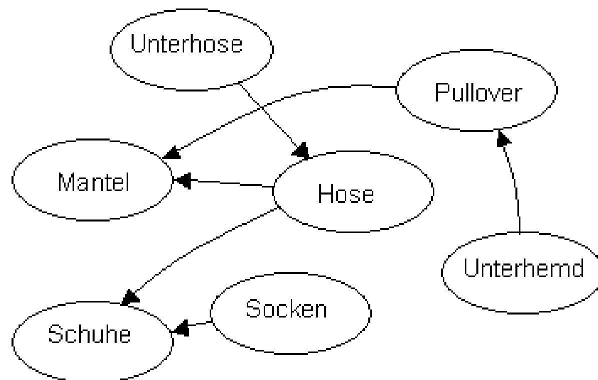
# Topologische Sortierung

## Definition 5

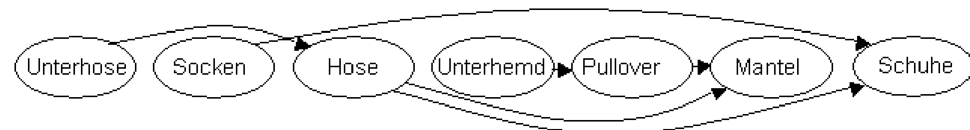
Eine lineare Anordnung $t$ der Knoten eines DAGs $G = (V, E)$, in der alle Kanten von "kleineren" zu "größeren" Knoten verlaufen, heißt topologische Sortierung, d. h.

$$\forall (u, v) \in E : t(u) < t(v).$$

**Beispiel:**




Kleidergraph, Quelle: Wikipedia




topologisch sortierter Kleidergraph, Quelle: Wikipedia

# Topologisches Sortieren mittels DFS

## Theorem 6

*G ist kreisfrei (DAG) $\Leftrightarrow$ DFS findet keine Rückwärtskante.*
*In diesem Fall liefert*

$$t(v) := n - \mathrm{finishTime}[v]$$

*eine topologische Sortierung.*

# Topologisches Sortieren mittels DFS

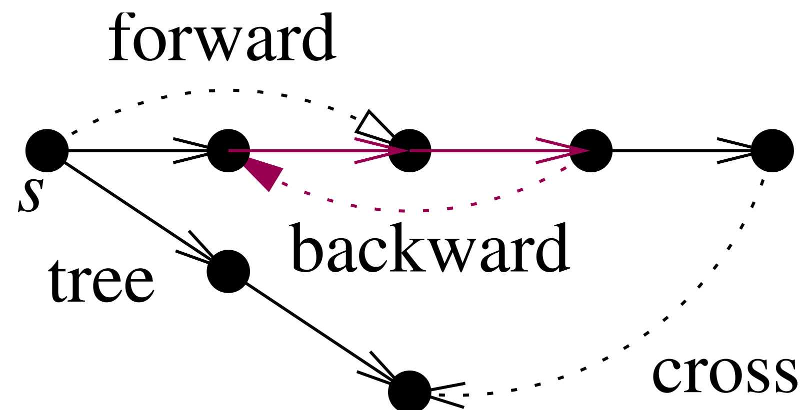
## Theorem 6

*G ist kreisfrei (DAG) ⇔ DFS findet keine Rückwärtskante.*
*In diesem Fall liefert*

$$t(v) := n - \mathrm{finishTime}[v]$$

*eine topologische Sortierung.*

**Beweis "⇒":** Annahme: ∃ Rückwärtskante.
Zusammen mit Baumkanten ergibt sich ein Kreis.
Widerspruch.



# Topologisches Sortieren mittels DFS

**Satz:** $G$ kreisfrei (DAG) $\Leftrightarrow$ DFS findet keine Rückwärtskante.
In diesem Fall liefert $t(v) := n - \text{finishTime}[v]$ eine topologische Sortierung, d. h. $\forall (u, v) \in E : t(u) < t(v)$.

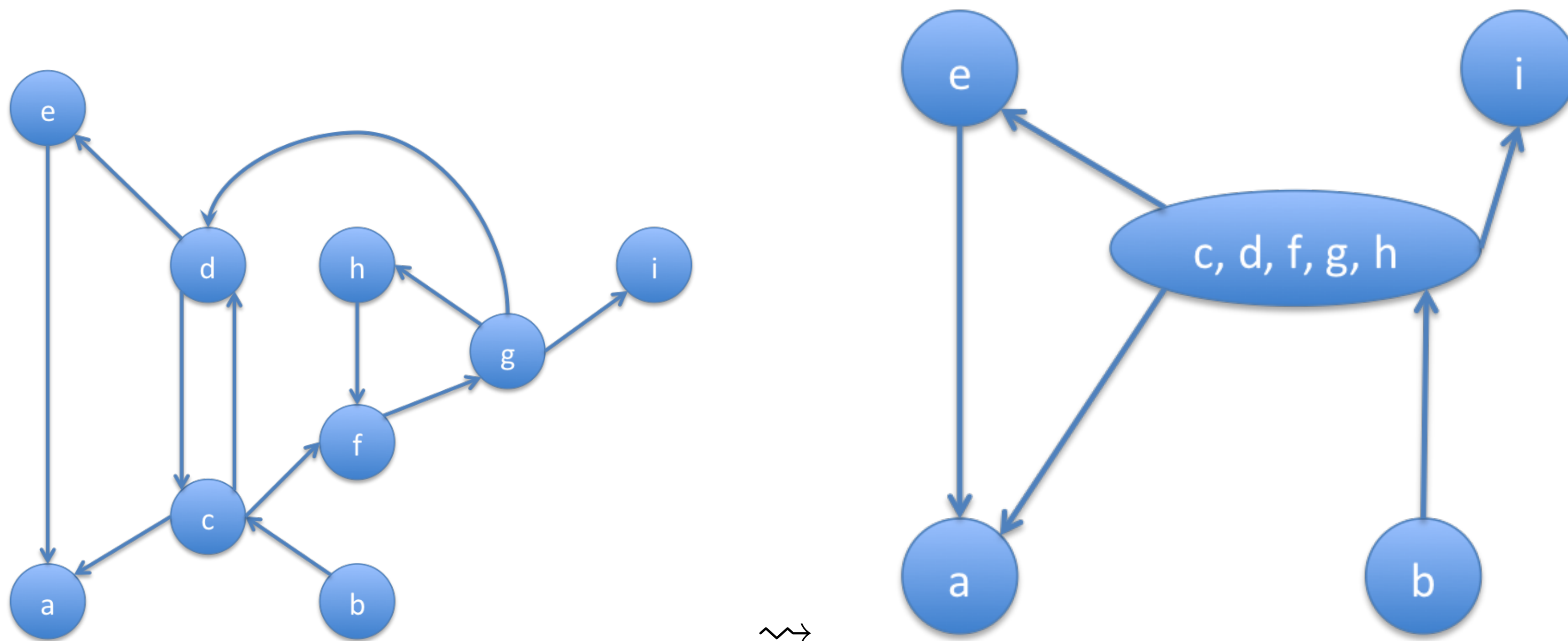
**Beweis "$\Leftarrow$":**
Keine Rückwärtskante

$$\overbrace{\Rightarrow}^{\text{Kantenklassifizierung}} \quad \forall (v, w) \in E : \text{finishTime}[v] > \text{finishTime}[w]$$

$\Rightarrow$ $G$ ist kreisfrei und
finishTime definiert umgekehrte topologische Sortierung.

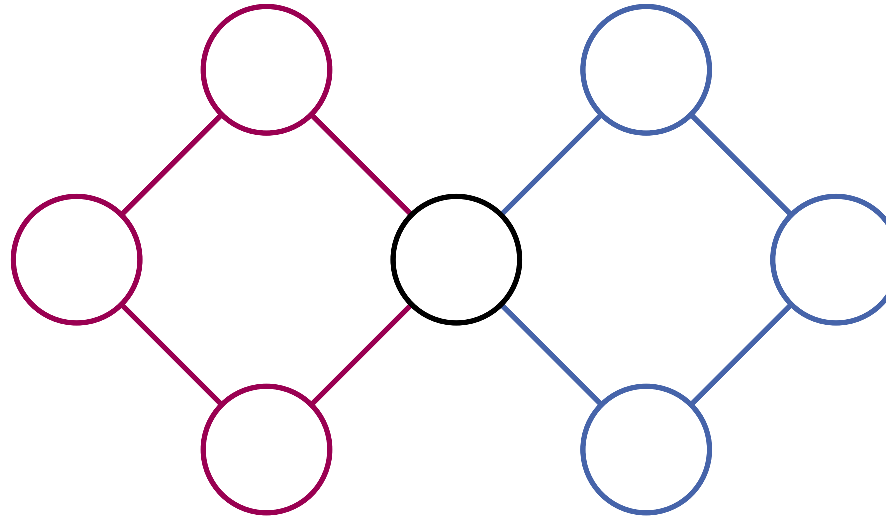
# Starke Zusammenhangskomponenten

Betrachte die Relation $\overset{*}{\leftrightarrow}$ mit
$u \overset{*}{\leftrightarrow} v$ falls $\exists$ Pfad $\langle u, \ldots, v\rangle$ und $\exists$ Pfad $\langle v, \ldots, u\rangle$.
**Beobachtung:** $\overset{*}{\leftrightarrow}$ ist Äquivalenzrelation Übung
Die Äquivalenzklassen von $\overset{*}{\leftrightarrow}$ bezeichnet man als starke Zusammenhangskomponenten.




DFS-basierter Linearzeitalgorithmus $\longrightarrow$ Algorithmen II

# Mehr DFS-basierte Linearzeitalgorithmen



- 2-zusammenhängende Komponenten: bei Entfernen eines Knotens aus einer Komponente bleibt diese zusammenhängend (ungerichtet)
- 3-zusammenhängende Komponenten
- Planaritätstest (lässt sich der Graph kreuzungsfrei zeichnen?)
- Einbettung planarer Graphen

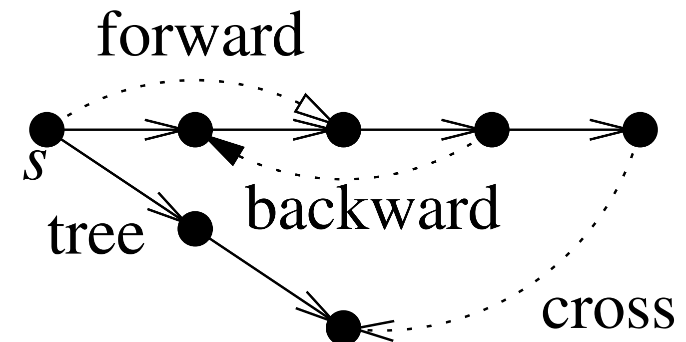
# BFS $\longleftrightarrow$ DFS

**pro BFS:**

- nichtrekursiv
- keine Vorwärtskanten
- kürzeste Wege, „Umgebung“

**pro DFS:**

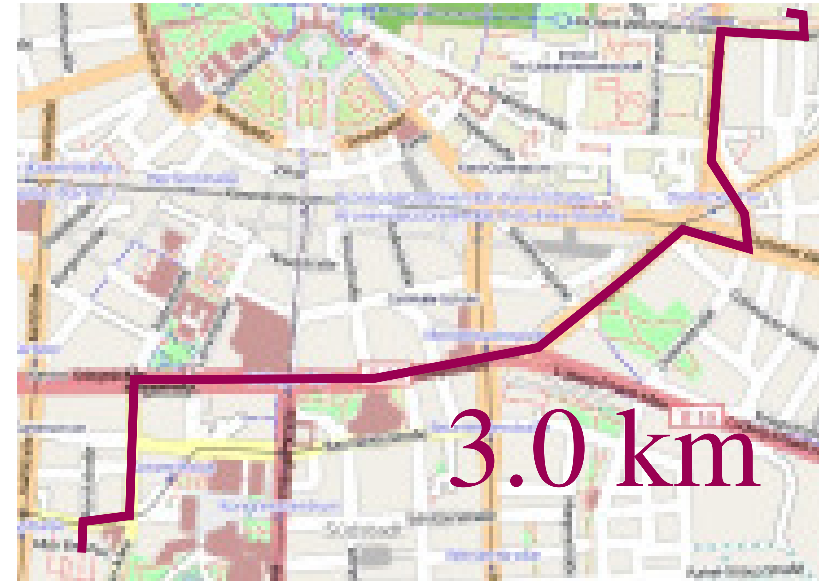
- keine explizite Datenstruktur (Rekursionsstapel) für ToDos, daher mglw. einfacher
- Grundlage vieler Algorithmen



# Kap. 10: Kürzeste Wege

**Eingabe:**

- Graph $G = (V, E)$ mit
- Kostenfunktion/Kantengewicht $c : E \to \mathbb{R}$ sowie
- Startknoten $s$.




**Ausgabe:** für alle $v \in V$:

- Länge $\mu(v)$ des kürzesten Pfades von $s$ nach $v$,
- $\mu(v) := \min\{c(p) : p \text{ ist Pfad von } s \text{ nach } v\}$ mit $c(\langle e_1, \ldots, e_k \rangle) := \sum_{i=1}^{k} c(e_i)$.

# Kap. 10: Kürzeste Wege

**Eingabe:**

- Graph $G = (V, E)$ mit
- Kostenfunktion/Kantengewicht $c : E \to \mathbb{R}$ sowie
- Startknoten $s$.

**Ausgabe:** für alle $v \in V$:

- Länge $\mu(v)$ des kürzesten Pfades von $s$ nach $v$,
- $\mu(v) := \min\{c(p) : p \text{ ist Pfad von } s \text{ nach } v\}$ mit $c(\langle e_1, \ldots, e_k \rangle) := \sum_{i=1}^{k} c(e_i)$.

Oft wollen wir auch „geeignete" Repräsentation der kürzesten Pfade.

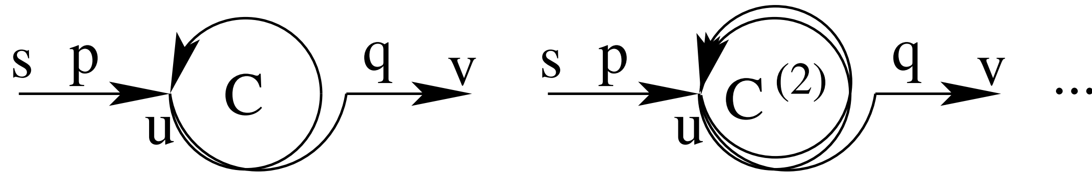
# Anwendungen

- Routenplanung
  - Straßennetze
  - Spiele
  - Kommunikationsnetze
- Unterprogramm
  - Flüsse in Netzwerken
  - ...
- Tippfehlerkorrektur
- Disk Scheduling
- ...

# Grundlagen

Gibt es immer einen kürzesten Pfad?

Es kann negative Kreise geben!



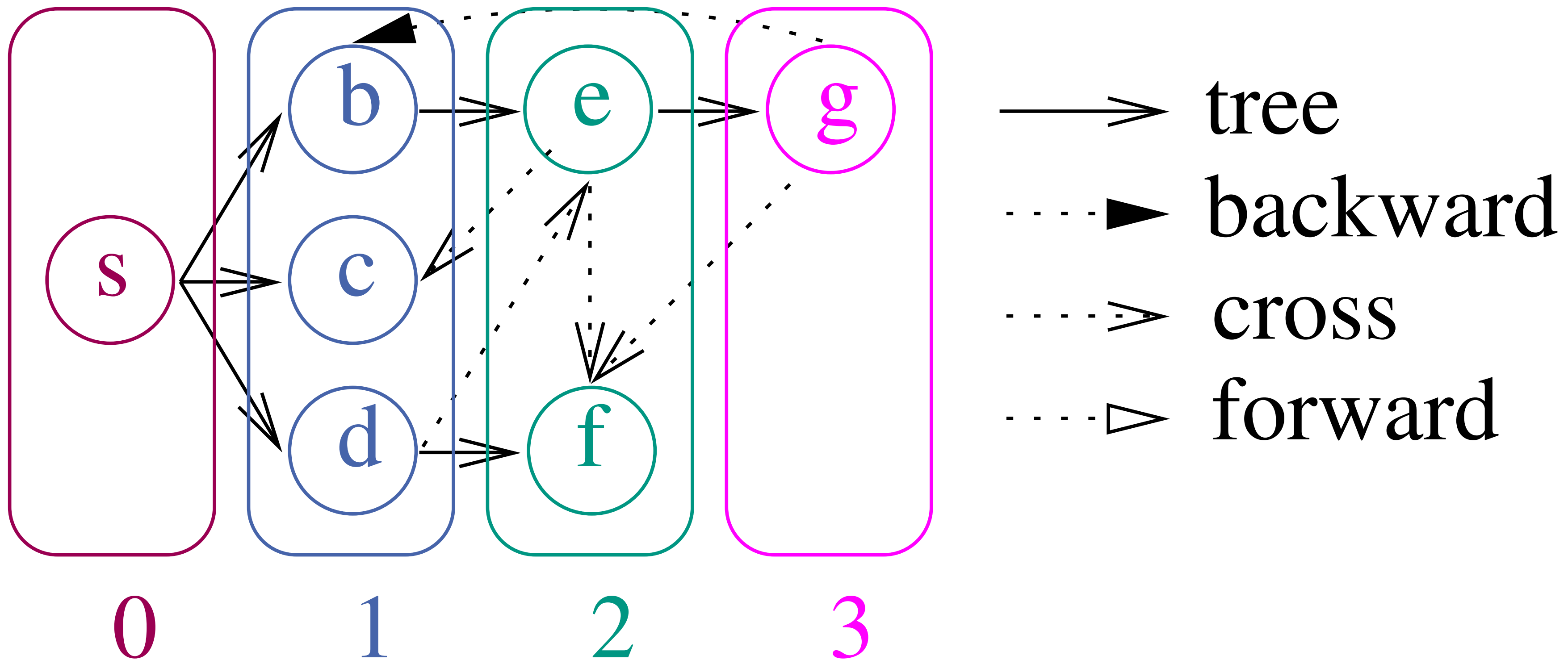

weitere Grundlagen just in time

# Azyklische Graphen

später

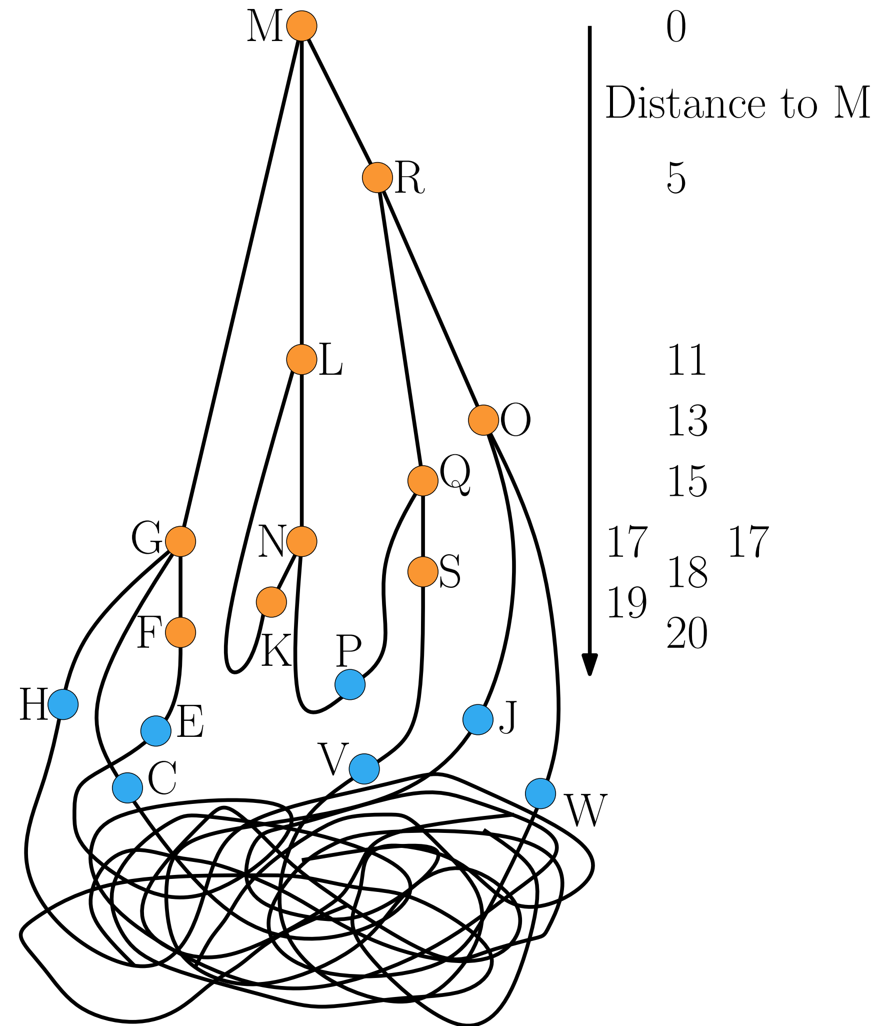
# Kantengewichte $\geq 0$

**Alle Gewichte gleich:** Breitensuche (BFS)!



# Dijkstras Algorithmus

**Nun:** Beliebige nichtnegative Kantengewichte




**Lösung ohne Rechner:**

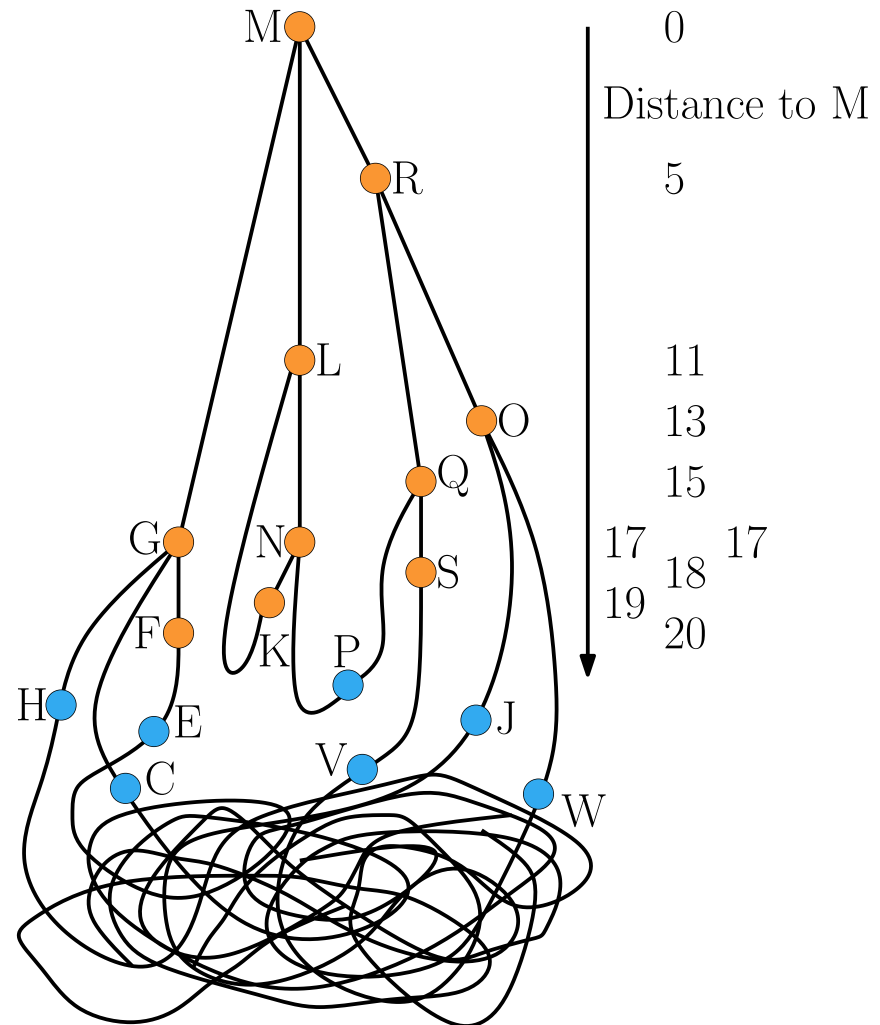
- Kanten $\to$ Fäden
- Kantengewicht $\to$ Fadenlänge
- Knoten $\to$ Knoten
- **Dann:** Am Startknoten anheben.

# Korrektheit der Bindfäden

Betrachte beliebigen Knoten $v$ mit Hängetiefe $d[v]$.

$\exists$ **Pfad mit Hängetiefe:**
verfolge straffe Fäden

$\neg\exists$ **kürzerer Pfad:**
falls es einen solchen Pfad gäbe, wäre einer seiner Fäden zerrissen!



# Edsger Wybe Dijkstra 1930–2002

Bildquelle: Wikipedia

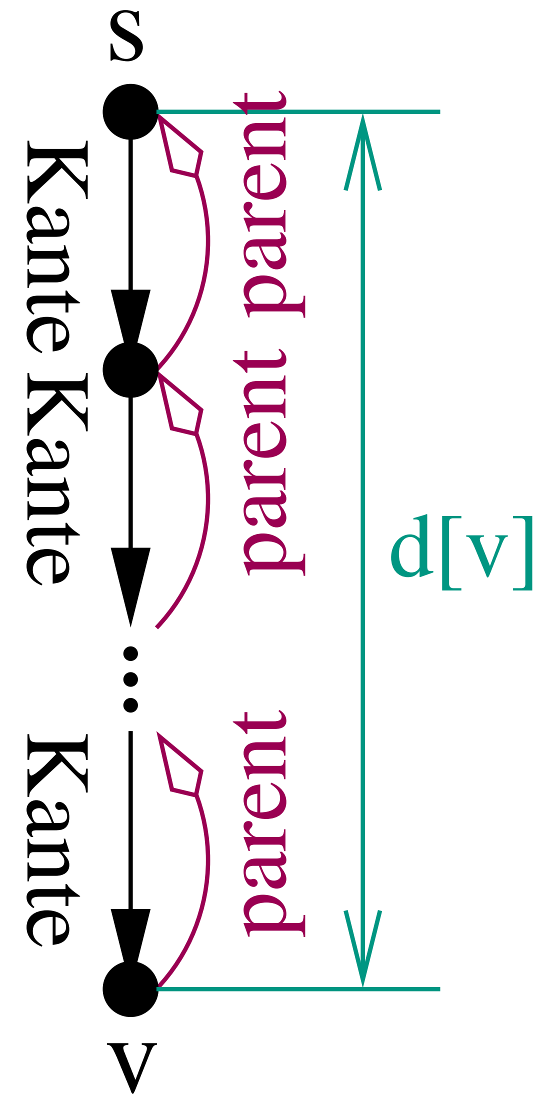
- 1972 ACM Turing Award
- THE: das erste Mulitasking-OS
- Semaphor
- Selbst-stabilisierende Systeme
- GOTO Statement Considered Harmful

# Allgemeine Definitionen

Wie bei BFS benutzen wir zwei Knotenarrays:

- $d[v]$ = aktuelle (vorläufige) Distanz von $s$ nach $v$
  **Invariante:** $d[v] \geq \mu(v)$
- parent$[v]$ = Vorgänger von $v$ auf dem (vorläufigen) kürzesten Pfad von $s$ nach $v$
  **Invariante:**
  dieser Pfad bezeugt $d[v]$

**Initialisierung:**
$d[s] = 0$, parent$[s] = s$
$d[v] = \infty$, parent$[v] = \perp$



# Kante $(u, v)$ relaxieren

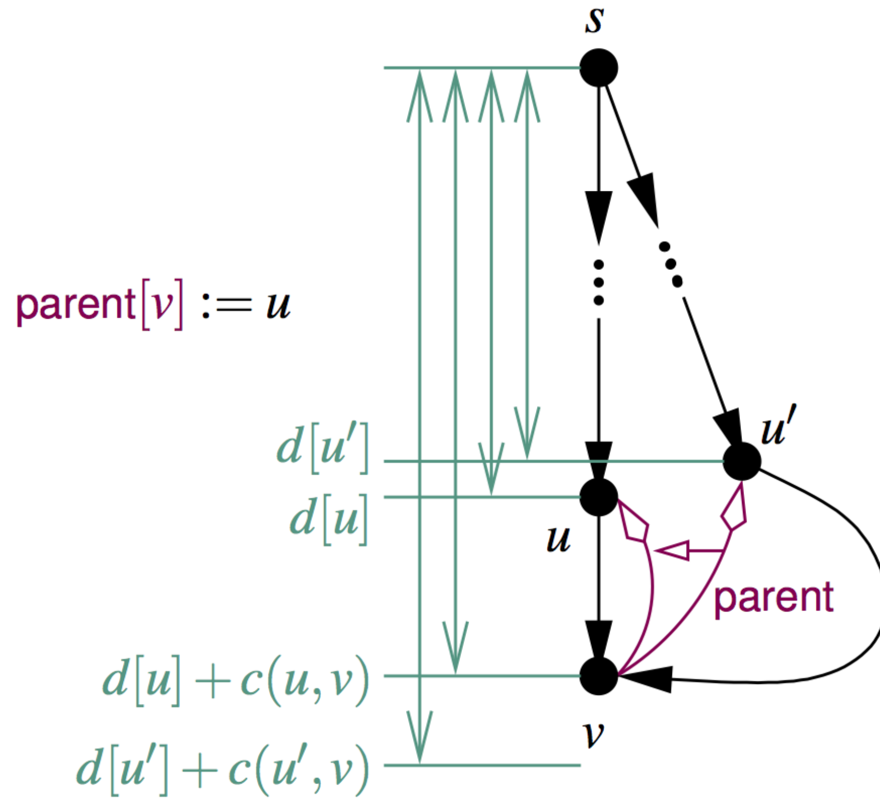
Falls $d[u] + c(u, v) < d[v]$
(vielleicht $d[v] = \infty$),

setze

- $d[v] := d[u] + c(u, v)$
  und
- $\text{parent}[v] := u$

Invarianten bleiben erhalten!

**Beobachtung:**
$d[v]$ kann sich mehrmals ändern!



# Dijkstras Algorithmus: Pseudocode

initialize $d$, parent
all nodes are non-scanned
**while** $\exists$ non-scanned node $u$ with $d[u] < \infty$
    $u :=$ non-scanned node $v$ with minimal $d[v]$
    relax all edges $(u, v)$ out of $u$
    $u$ is scanned now

**Behauptung:** Am Ende definiert $d$ die optimalen Entfernungen und parent die zugehörigen Wege

# Beispiel