Customization (Zuschneiden)

Anpassen der (Graph)Datenstruktur an die Anwendung.

- Ziel: schnell, kompakt.
- benutze Entwurfsprinzip: make the common case fast
- Listen vermeiden

Mögliches Problem: Software-Engineering-Alptraum

Möglicher Ausweg: Trennung von Algorithmus und Repräsentation

Beispiel: DAG-Erkennung

Vgl. Folie 73ff. (dort aber Test auf Ausgangsgrad)!

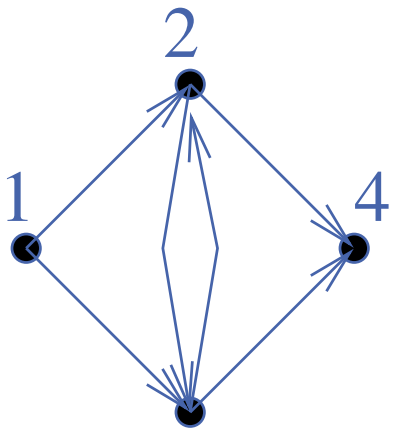
Function isDAG($G = (V, E)$) // Adjazenzarray!
  dropped:= 0
  compute array inDegree of indegrees of all nodes // Zeit $O(m)$!
  droppable=$\{v \in V : \text{inDegree}[v] = 0\}$: Stack
  while droppable $\neq \emptyset$ **do**
    invariant G is a DAG iff the input graph is a DAG
    v:= droppable.pop
    dropped++
    foreach edge $(v, w) \in E$ **do**
      inDegree[w]−−
      if inDegree[w] $= 0$ **then** droppable.push(w)
  return $|V| =$ dropped

Laufzeit: $O(m+n)$ (auch ohne dynamische Graphdatenstruktur!)

Adjazenz-Matrix

$A \in \{0,1\}^{n \times n}$ with $A(i,j) = [(i,j) \in E]$

- $+$ platzeffizient für sehr dichte Graphen
- $--$ platzineffizient sonst.
  Übung: was bedeutet "sehr dicht" hier?
- $+$ einfache Kantenanfragen
- $-$ langsame Navigation
- $++$ verbindet lineare Algebra und Graphentheorie



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

Beispiel: $\mathbf{C} = \mathbf{A}^k$.
$\mathbf{C}_{ij} =$# k-Kanten-Pfade von i nach j

Wichtige Beschleunigungstechniken:

- $O(\log k)$ Matrixmult. für Potenzberechnung
- Matrixmultiplikation in subkubischer Zeit, z. B., Strassens Algorithmus

Pfade zählen mittels LA

Adjanzenzmatrix:

$A \in \{0,1\}^{n \times n}$ mit $A(i,j) = [(i,j) \in E]$
Sei $C := A^k$.

Behauptung: $C_{ij} =$# k-Kanten-Pfade von i nach j.

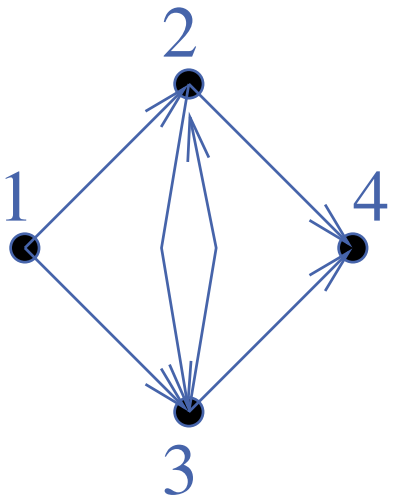
Beweis: IA $(k = 1)$ $C = A^1 = A$ stimmt nach Definition von A.

Schluss $k \rightsquigarrow k+1$: $C_{ij} = (A^k A)_{ij} = \sum_\ell A^k_{i\ell} A_{\ell j}$

$A^k_{i\ell} =$#k-Kanten-Pfade von i nach ℓ (nach IV).
$A^k_{i\ell} A_{\ell j} =$#$k+1$-Kanten-Pfade von i nach j mit (ℓ, j) als letzter Kante.
Jede mögliche letzte Kante wird genau einmal gezählt.

□

Übung: zähle Pfade der Länge $\leq k$



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

Beispiel, wo Graphentheorie bei LA hilft

Problemstellung: löse $\mathbf{Bx} = \mathbf{c}$
Sei $G = (1..n, E = \{\{i,j\} : B_{ij} \neq 0\})$
Nehmen wir an, G habe zwei Zusammenhangskomponenten
$\Rightarrow$ tausche Zeilen und Spalten derart, dass

$$\begin{pmatrix} \mathbf{B}_1 & \mathbf{0} \\ \mathbf{0} & \mathbf{B}_2 \end{pmatrix} \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix} = \begin{pmatrix} \mathbf{c}_1 \\ \mathbf{c}_2 \end{pmatrix}$$

zu lösen bleibt.

Übung: Was passiert, wenn $(1..n, E = \{(i,j) : B_{ij} \neq 0\})$ ein DAG ist?

Implizite Repräsentation

Kompakte Repräsentation möglicherweise sehr dichter Graphen
Implementiere Algorithmen direkt mittels dieser Repräsentation

Beispiel: Intervall-Graphen

Knoten: Intervalle $[a, b] \subseteq \mathbb{R}$
Kanten: zwischen überlappenden Intervallen

Zusammenhangstest für Intervallgraphen

$V = \{[a_1, b_1], \ldots, [a_n, b_n]\}$
$E = \{\{[a_i, b_i], [a_j, b_j]\} : [a_i, b_i] \cap [a_j, b_j] \neq \emptyset\}$

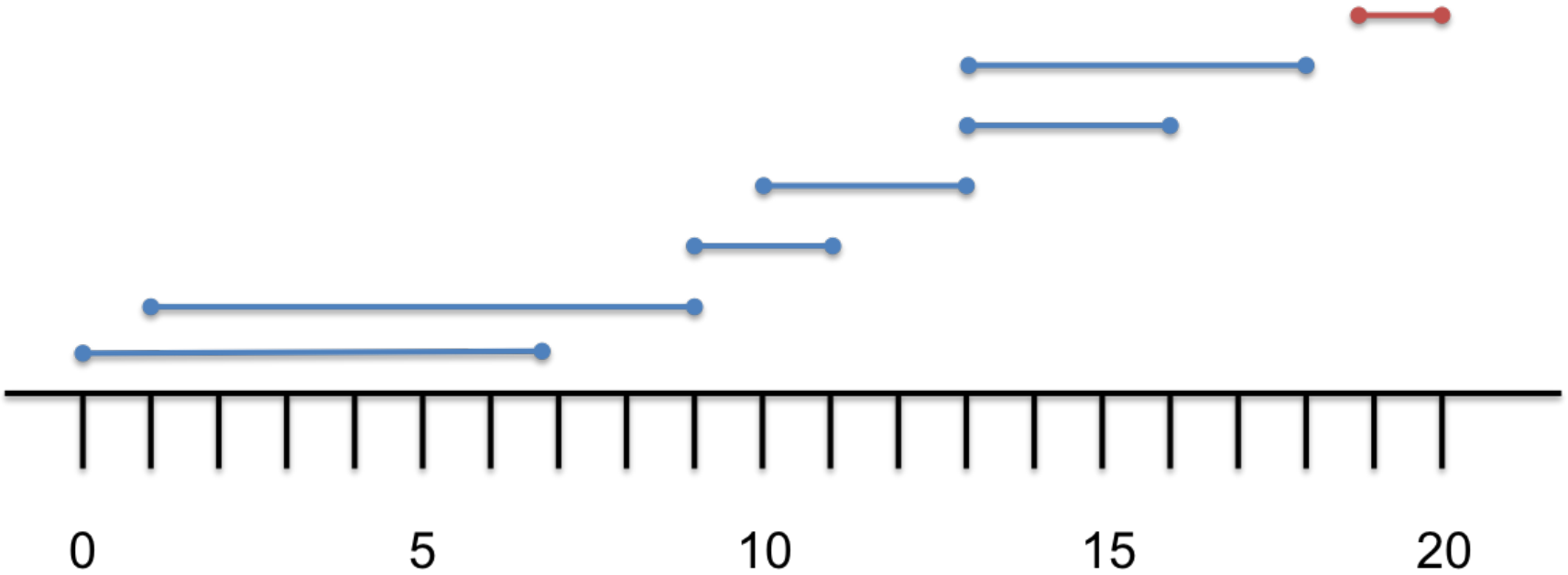
Idee: durchlaufe Intervalle von links nach rechts.
Die Anzahl überlappender Intervalle darf nie auf null sinken.
Annahme: Startpunkte in Sortierung vor Endpunkten!

Function isConnected(L : SortedListOfIntervalEndPoints) : $\{0, 1\}$
    remove first element of L; overlap := 1
    foreach $p \in L$ **do**
        if overlap$= 0$ **return** false
        if p is a start point **then** overlap++
        else overlap−− // end point
    return true

$O(n \log n)$ Algorithmus für bis zu $O(n^2)$ Kanten!
Übung: Zusammenhangskomponenten finden

Beispiel

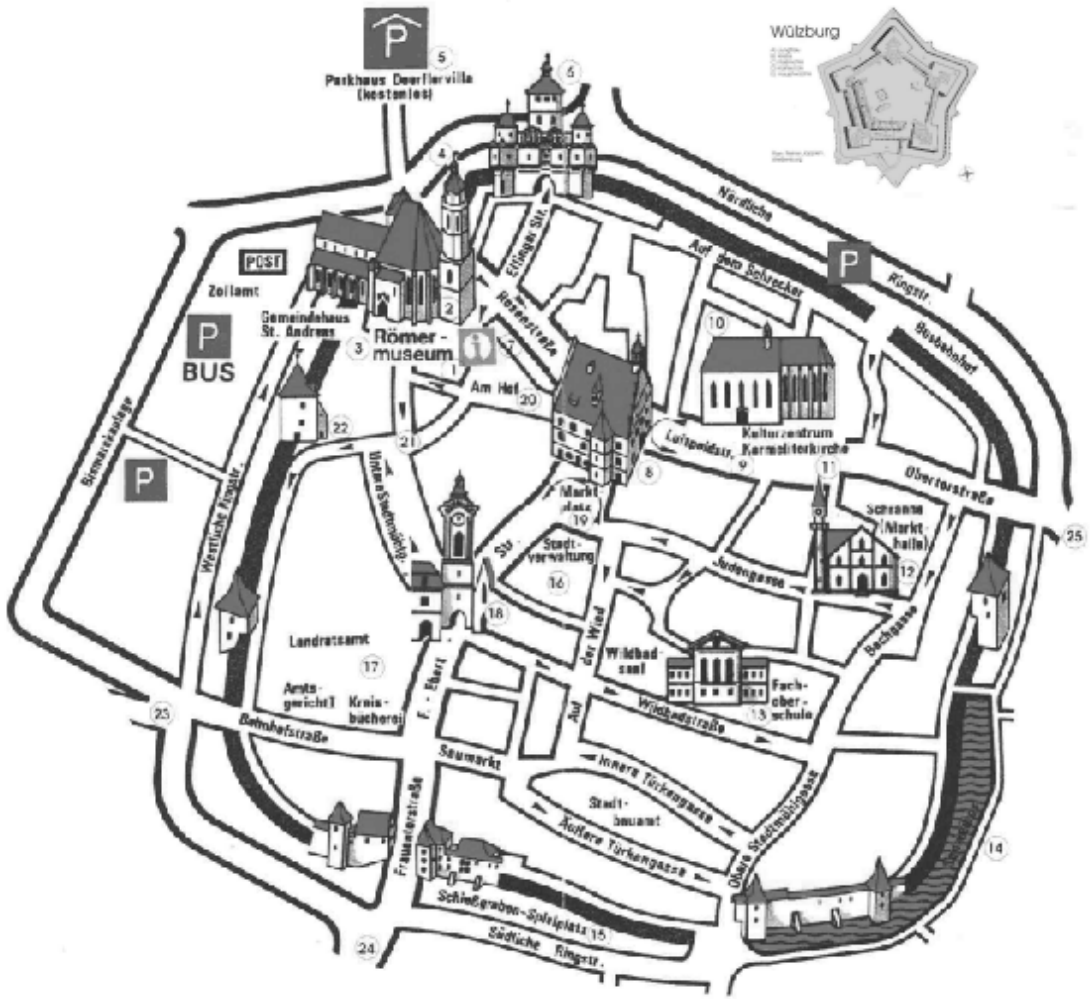
Function isConnected(L : SortedListOfIntervalEndPoints) : $\{0,1\}$
    remove first element of L; overlap := 1
    foreach $p \in L$ **do**
        if overlap$=0$ **return** false
        if p is a start point **then** overlap++
        else overlap−− // end point
    return true



Graphrepräsentation: Zusammenfassung

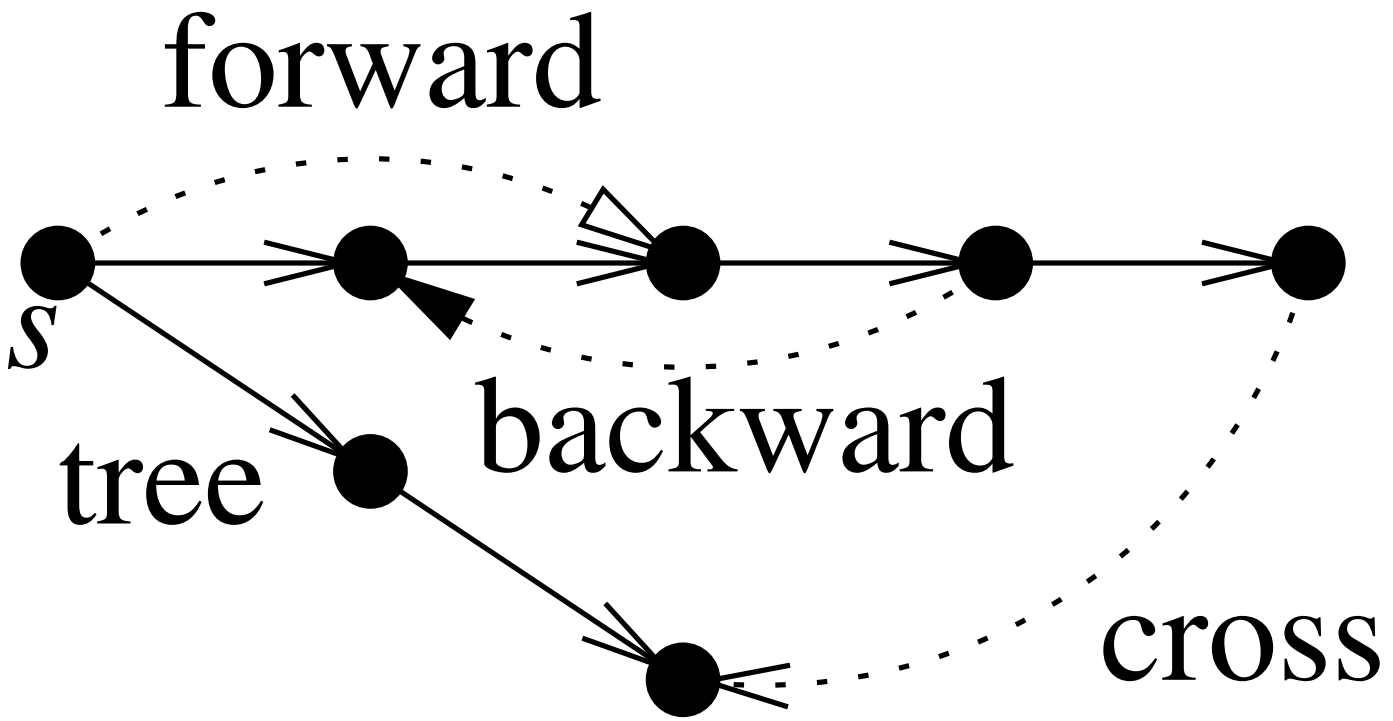
- Welche Operationen werden gebraucht?
- Wie oft?
- Adjazenzarrays gut für statische Graphen
- Pointer $\rightsquigarrow$ flexibler, aber auch teurer
- Matrizen eher konzeptionell interessant

Kap. 9: Graphtraversierung



Ausgangspunkt oder Baustein fast jedes nichttrivialen Graphenalgorithmus

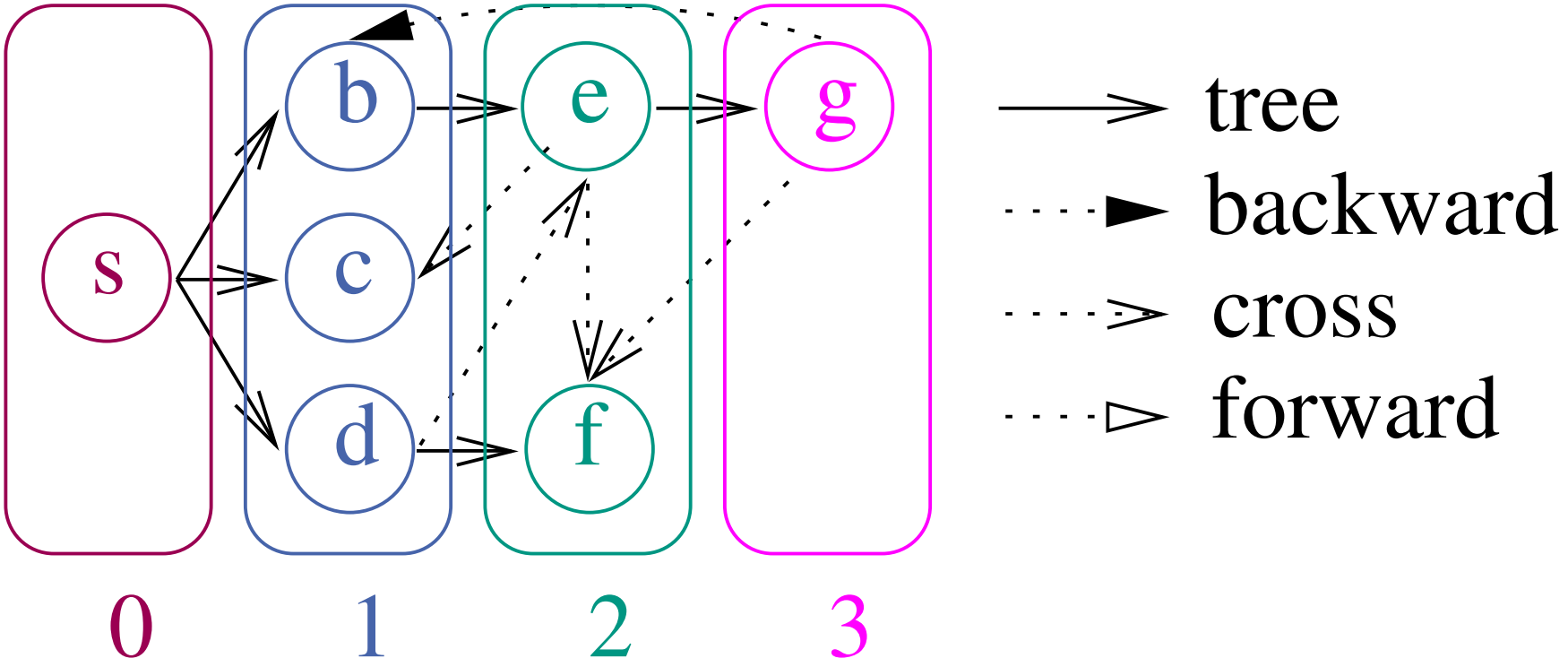
Graphtraversierung als Kantenklassifizierung



Breitensuche

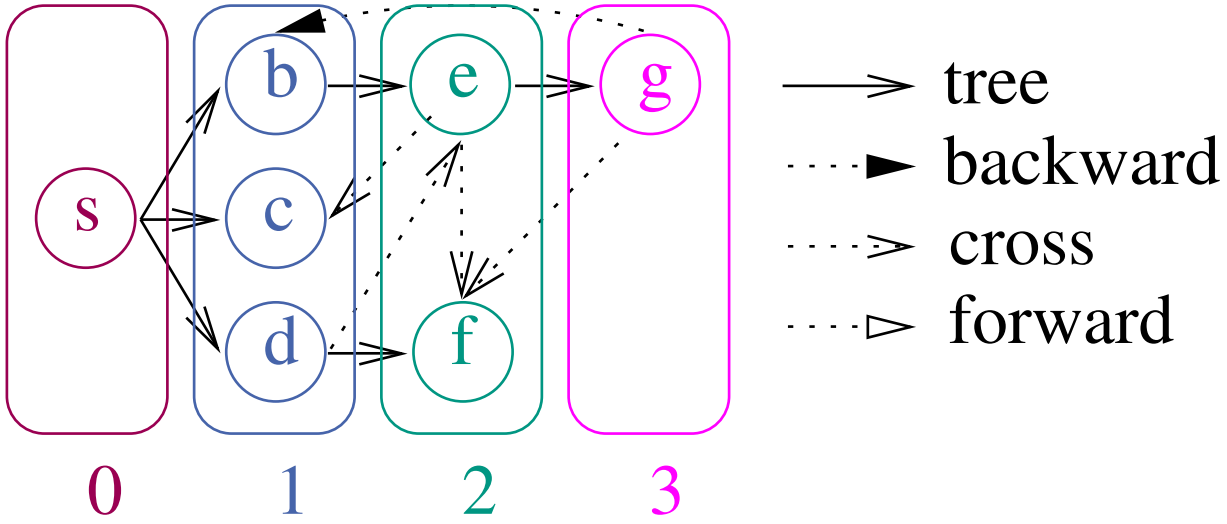
Baue Baum von Startknoten s,
der alle von s erreichbaren Knoten
mit möglichst kurzen Pfaden erreicht.

Berechne Abstände:



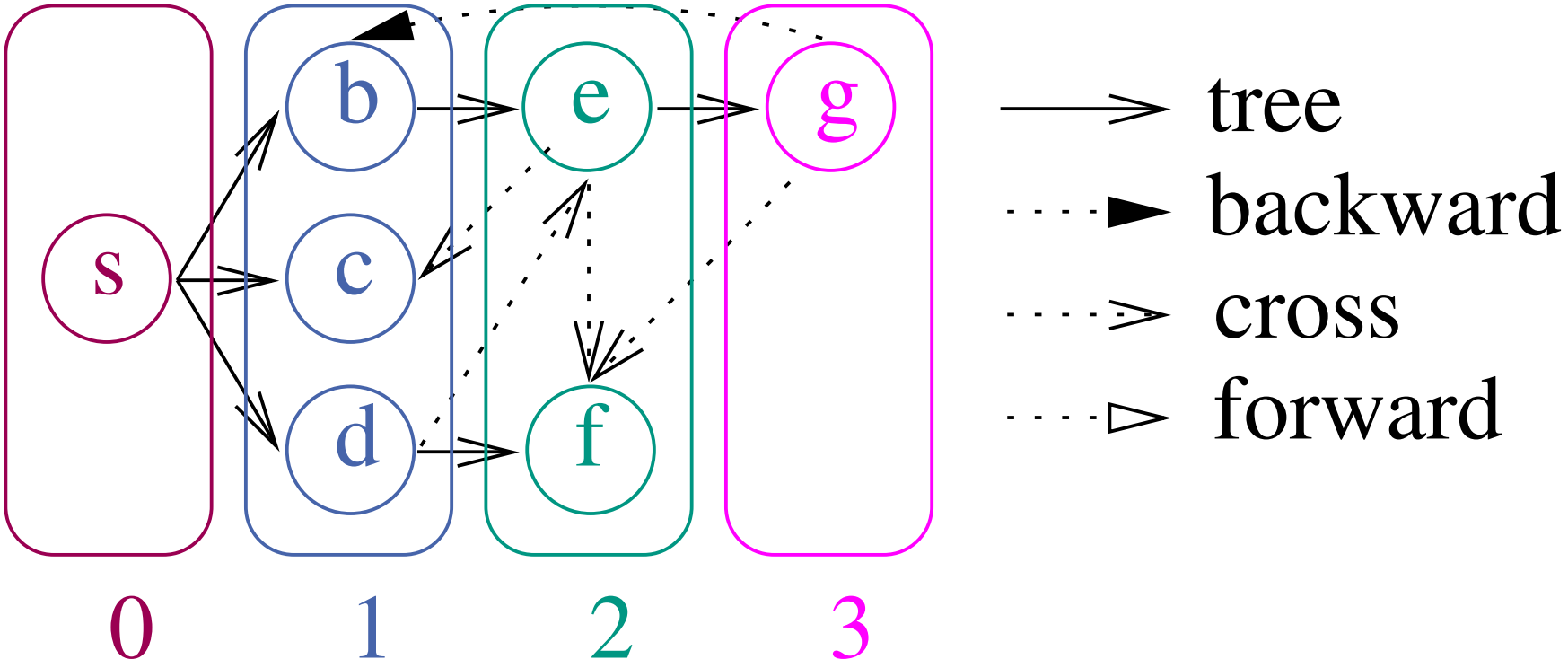
Breitensuche

- 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'

$Q := Q'$