

# Datenstrukturen und Algorithmen

## Vorlesung 14+15: Elementare Graphenalgorithmen (K22,K24.2)

Joost-Pieter Katoen

Lehrstuhl für Informatik 2  
Software Modeling and Verification Group

<http://moves.rwth-aachen.de/teaching/ss-15/dsa1/>

5.+11. Juni 2015



## Übersicht

- 1 Softwarewettbewerb
- 2 Graphen
  - Terminologie
  - Repräsentation von Graphen
- 3 Graphendurchlauf
  - Breitensuche
  - Tiefensuche
- 4 Anwendungen der Tiefensuche
  - Erreichbarkeitsanalyse
  - CCs in ungerichteten Graphen
  - SCCs in gerichteten Graphen
  - Topologische Sortierung in gerichteten azyklischen Graphen
  - Kritische-Pfad-Analyse in gewichteten gerichteten azyklischen Graphen

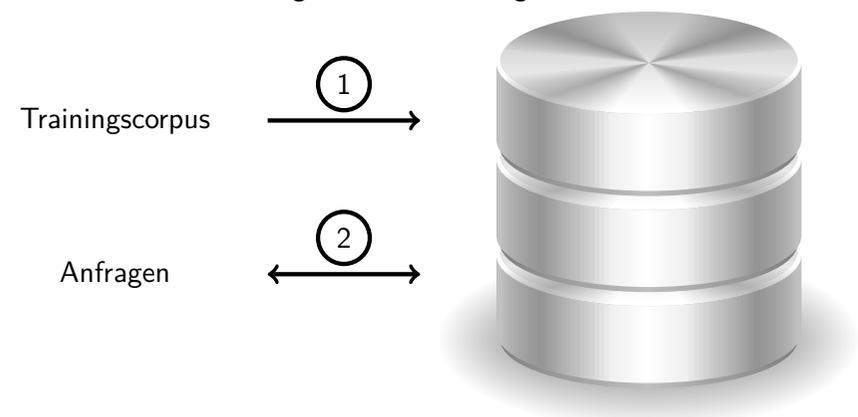
## Übersicht

- 1 Softwarewettbewerb
- 2 Graphen
  - Terminologie
  - Repräsentation von Graphen
- 3 Graphendurchlauf
  - Breitensuche
  - Tiefensuche
- 4 Anwendungen der Tiefensuche
  - Erreichbarkeitsanalyse
  - CCs in ungerichteten Graphen
  - SCCs in gerichteten Graphen
  - Topologische Sortierung in gerichteten azyklischen Graphen
  - Kritische-Pfad-Analyse in gewichteten gerichteten azyklischen Graphen

## Trigramme . . .

sind wichtig für automatische Verarbeitung natürlicher Sprache:

- ▶ Trainingscorpus liefert Trigramme mit Anzahl Vorkommen
- ▶ Datenstruktur speichert diese Informationen
- ▶ beantwortet Anfragen auf neuen Eingabedaten



## ADT TrigramStorage

### TrigramStorage

Ein `TrigramStorage` `s` speichert eine Ansammlung von **Trigrammen** und den dazugehörigen **Counts** und bietet folgende Operationen:

- ▶ `void insert(s, TrigramAndCount t)` fügt das Trigramm `t` in `s` ein.
- ▶ `QueryResult query(s, Trigram t1, Trigram t2)` gibt das **Maximum** und die **Summe** der Counts aller Trigramme `t` in `s`, die lexikographisch zwischen `t1` und `t2` liegen.
- ▶ `void build(s)` wird nach dem Einfügen aller Elemente aufgerufen und kann die interne Repräsentation der Daten ändern, um schneller auf Anfragen antworten zu können.

## Übersicht

- 1 Softwarewettbewerb
- 2 **Graphen**
  - Terminologie
  - Repräsentation von Graphen
- 3 Graphendurchlauf
  - Breitensuche
  - Tiefensuche
- 4 Anwendungen der Tiefensuche
  - Erreichbarkeitsanalyse
  - CCs in ungerichteten Graphen
  - SCCs in gerichteten Graphen
  - Topologische Sortierung in gerichteten azyklischen Graphen
  - Kritische-Pfad-Analyse in gewichteten gerichteten azyklischen Graphen

## Ihre Aufgabe

Schreiben Sie eine Implementierung des ADT `TrigramStorage` in Java, der Anfragen **so schnell wie möglich** beantwortet.

### Regeln

- ▶ bis zu (ca.)  $100 \cdot 10^6$  Trigramme und (ca.)  $100 \cdot 10^3$  Anfragen
- ▶ Zeit für `insert` und `build` wird nicht gezählt, nur die **Anfragezeiten**
- ▶ Java 7
- ▶ keine nicht-elementaren Bibliotheksfunktionen benutzen (kein `java.util.sort`, kein `java.util.hash`, etc.)
- ▶ kein Source Code aus anderen Quellen

Details (Framework, Beispiel-Implementierung):

<http://moves.rwth-aachen.de/teaching/ss-15/dsa1/>

## Die Bedeutung von Graphen

**Graphen** werden in vielen (Informatik-)Anwendungen verwendet:

### Beispiele

- ▶ (Computer-)Netzwerke
- ▶ Webseiten und ihre Vernetzung mittels Links
- ▶ Darstellung von topologischen Informationen (Karten, ...)
- ▶ Darstellung von elektronischen Schaltungen
- ▶ Vorranggraphen (precedence graph), Ablaufpläne, ...
- ▶ Semantische Netze (z. B. Entity-Relationship-Diagramme)

*Wir werden uns auf fundamentale Graphalgorithmen konzentrieren.*

## Gerichteter Graph

### Gerichteter Graph

Ein **gerichteter Graph** (auch: **digraph**)  $G$  ist ein Paar  $(V, E)$  mit

- ▶ einer Menge  $V$  von **Knoten** (**vertices**) und
- ▶ einer Menge  $E \subseteq \{(u, v) \mid u, v \in V\}$  von **Kanten** (**edges**).

### Ungerichteter Graph

Ein **ungerichteter Graph**  $G$  ist ein Paar  $(V, E)$  mit

- ▶ einer Menge  $V$  von **Knoten** (**vertices**) und
- ▶ einer Menge  $E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$  von **Kanten** (**edges**).

Auch für ungerichtete Kanten verwenden wir die Notation  $(u, v)$ .

### Adjazent

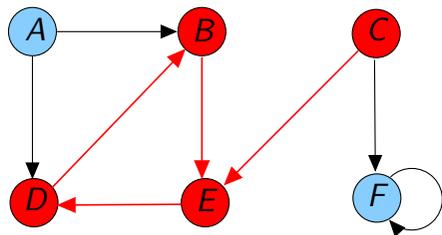
Knoten  $v \in V$  ist **adjazent** zu  $u \in V$ , wenn  $(u, v) \in E$ .

## Terminologie bei Graphen

### Teilgraph

Ein **Teilgraph** (subgraph) eines Graphen  $G = (V, E)$  ist ein Graph  $G' = (V', E')$  mit:

- ▶  $V' \subseteq V$  und  $E' \subseteq E$ .
- ▶ Außerdem ist  $E' \subseteq V' \times V'$  wegen der Grapheigenschaft von  $G'$ .
- ▶ Ist  $V' \subset V$  und  $E' \subset E$ , so heißt  $G'$  **echter** (proper) Teilgraph.

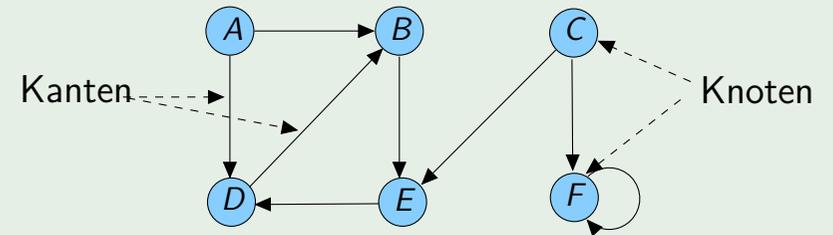


Beispiel: Rote Knoten und Kanten bilden einen (echten) Teilgraphen

## Gerichteter Graph

### Beispiel

- ▶  $V = \{A, \dots, F\}$
- ▶  $E = \{(A, B), (A, D), (B, E), (C, E), (C, F), (D, B), (E, D), (F, F)\}$



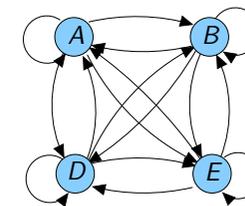
A und B sind adjazent, wie auch z.B. C und E, und F und F.

## Terminologie bei Graphen

### Symmetrischer Graph

Der Graph  $G$  heißt **symmetrisch**, wenn aus  $(u, v) \in E$  folgt  $(v, u) \in E$ .

- ▶ Zu jedem ungerichteten Graphen gibt es einen korrespondierenden symmetrischen Digraphen.



Beispiel: Symmetrischer Digraph

### Vollständiger Graph

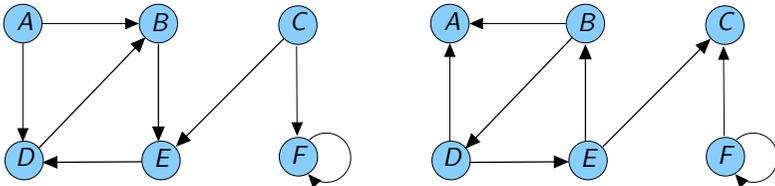
Ein ungerichteter Graph ist **vollständig**, wenn *jedes* Paar von Knoten durch eine Kante verbunden ist.

## Terminologie bei Graphen

### Transponieren

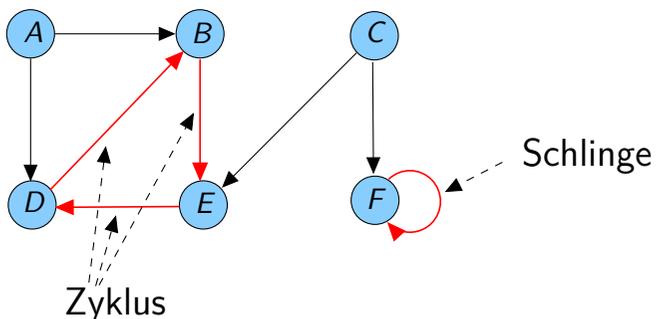
**Transponiert** man  $G = (V, E)$  (transpose graph), so erhält man  $G^T = (V, E')$  mit  $(v, u) \in E' \text{ gdw. } (u, v) \in E$ .

- ▶ In  $G^T$  ist die Richtung der Kanten von  $G$  umgedreht.



Beispiel: Ein Graph und sein transponierter Graph

## Pfade und Zyklen



$ABEDB$  und  $CF$  sind Pfade, aber keine einfache Pfade.

$EDB$  und  $CF$  sind einfache Pfade.

## Pfade und Zyklen

### Pfad (Weg)

Ein **Pfad** der Länge  $k$  vom Knoten  $u \in V$  zum Knoten  $w \in V$  in  $G = (V, E)$  ist eine Folge  $v_0 v_1 v_2 \dots v_{k-1} v_k$  von Knoten  $v_i \in V$  mit  $(v_i, v_{i+1}) \in E$ ,  $v_0 = u$  und  $v_k = w$ .

- ▶ Die Länge eines Pfades ist die Anzahl der durchlaufenen Kanten.
- ▶ Einen Pfad der Länge 0 nennen wir **leer**.
- ▶ Ein Pfad mit  $v_i \neq v_j$  für alle  $i \neq j$  heißt **einfach**.
- ▶ Knoten  $w$  ist **erreichbar** von  $u$ , wenn es einen Pfad von  $u$  nach  $w$  gibt.

### Zyklen in gerichteten Graphen

Ein **Zyklus** in einem *gerichteten* Graph ist ein nicht-leerer Pfad bei dem der Startknoten auch der Endknoten ist.

- ▶ Ein Zyklus  $vv$  der Länge 1 heißt **Schlinge** (self-loop).
- ▶ Ein Zyklus  $v_0 \dots v_k$  ist **einfach** wenn  $v_1 \dots v_k$  paarweise verschieden sind.

- ▶ Ein gerichteter Graph ist **azyklisch**, wenn er keine Zyklen hat.

## Zusammenhangskomponenten in ungerichteten Graphen

### Zusammenhangskomponenten in ungerichteten Graphen

Sei  $G$  ein **ungerichteter** Graph.

- ▶  $G$  heißt **zusammenhängend** (**connected**), wenn jeder Knoten von jedem anderen Knoten aus erreichbar ist.
- ▶ Eine **Zusammenhangskomponente** (**connected component, CC**) von  $G$  ist ein maximaler (d.h. nicht erweiterbarer) zusammenhängender Teilgraph von  $G$ .

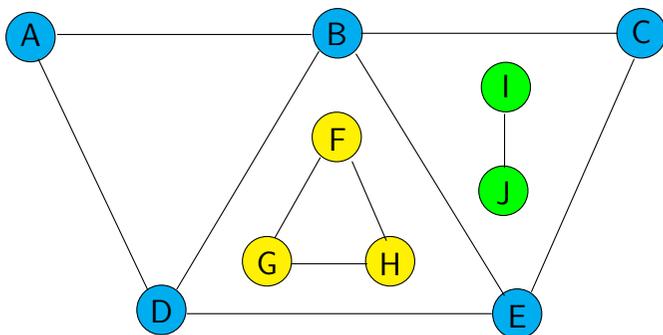
## Zusammenhangskomponenten in gerichteten Graphen

### Zusammenhangskomponenten in gerichteten Graphen

Sei  $G$  ein **gerichteter** Graph.

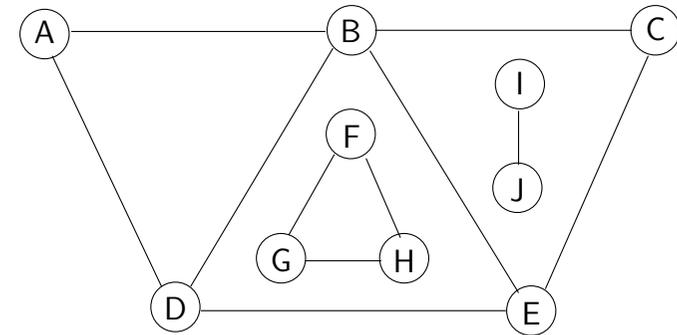
- ▶  $G$  heißt **stark zusammenhängend** (*strongly connected*), wenn jeder Knoten von jedem anderen Knoten aus erreichbar ist.
  - ▶  $G$  heißt **schwach zusammenhängend** (*weakly connected*), wenn der zugehörige ungerichtete Graph (wenn man alle Kanten ungerichtet macht) zusammenhängend ist.
  - ▶ Eine **starke Zusammenhangskomponente** (*strongly connected component, SCC*) von  $G$  ist ein maximaler stark zusammenhängender Teilgraph von  $G$ .
- ▶ Die Zerlegung eines Graphen in seine Zusammenhangskomponente ist **eindeutig**.

## Ungerichtete, zusammenhängende Graphen



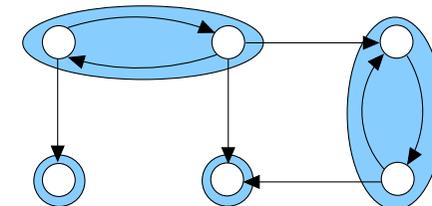
Die Zusammenhangskomponenten.

## Ungerichtete, zusammenhängende Graphen



Ein ungerichteter Graph; Was sind die Zusammenhangskomponenten?

## Starke Zusammenhangskomponenten



Ein Digraph und seine SCCs.

## Zusammenhang

### Bemerkungen

- ▶ Ein Baum (zusammenhängender azyklischer Graph) mit  $n$  Knoten hat  $n - 1$  Kanten.
- ▶ Ein ungerichteter Graph mit  $n$  Knoten und weniger als  $n - 1$  Kanten kann nicht zusammenhängend sein.
- ▶ Ein ungerichteter Graph mit  $n$  Knoten und mindestens  $n$  Kanten muss einen Zyklus enthalten.

## Repräsentation von Graphen: Adjazenzmatrix

Sei  $G = (V, E)$  mit  $|V| = n$ ,  $|E| = m$  und  $V = \{v_1, \dots, v_n\}$ .

### Adjazenzmatrix

Die **Adjazenzmatrix**-Darstellung eines Graphen ist durch eine  $n \times n$  boolesche Matrix  $A$  gegeben, wobei  $A(i, j) = 1$ , wenn  $(v_i, v_j) \in E$ , sonst 0.

- ▶ Wenn  $G$  ungerichtet ist, ergibt sich symmetrisches  $A$  (d. h.  $A = A^T$ ). Dann muss nur die Hälfte gespeichert werden.
- ⇒ Platzbedarf:  $\Theta(n^2)$ .

## Übersicht

- 1 Softwarewettbewerb
- 2 **Graphen**
  - Terminologie
  - Repräsentation von Graphen
- 3 Graphendurchlauf
  - Breitensuche
  - Tiefensuche
- 4 Anwendungen der Tiefensuche
  - Erreichbarkeitsanalyse
  - CCs in ungerichteten Graphen
  - SCCs in gerichteten Graphen
  - Topologische Sortierung in gerichteten azyklischen Graphen
  - Kritische-Pfad-Analyse in gewichteten gerichteten azyklischen Graphen

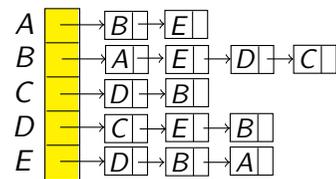
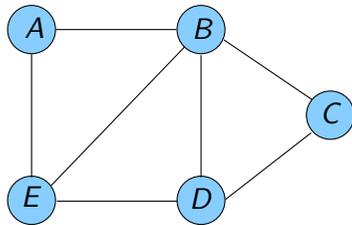
## Repräsentation von Graphen: Adjazenzliste

### Adjazenzliste

Bei der Darstellung als **Array von Adjazenzlisten** gibt es ein durch die Nummer des Knoten indiziertes Array, das jeweils verkettete Listen (Adjazenzlisten) enthält.

- ▶ Der  $i$ -te Arrayeintrag enthält alle Kanten von  $G$ , die von  $v_i$  „ausgehen“.
  - ▶ Ist  $G$  ungerichtet, dann werden Kanten doppelt gespeichert.
  - ▶ Kanten, die in  $G$  nicht vorkommen, benötigen keinen Speicherplatz.
- ⇒ Platzbedarf:  $\Theta(n + m)$ .

## Darstellung eines ungerichteten Graphen

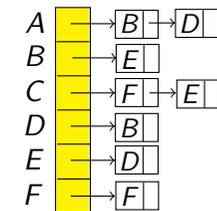
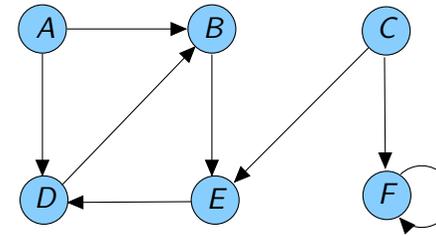


Adjazenzliste

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

Adjazenzmatrix

## Darstellung eines gerichteten Graphen



Adjazenzliste

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

Adjazenzmatrix

## Übersicht

- 1 Softwarewettbewerb
- 2 Graphen
  - Terminologie
  - Repräsentation von Graphen
- 3 Graphendurchlauf
  - Breitensuche
  - Tiefensuche
- 4 Anwendungen der Tiefensuche
  - Erreichbarkeitsanalyse
  - CCs in ungerichteten Graphen
  - SCCs in gerichteten Graphen
  - Topologische Sortierung in gerichteten azyklischen Graphen
  - Kritische-Pfad-Analyse in gewichteten gerichteten azyklischen Graphen

## Graphendurchlauf

Viele Algorithmen, die wir später kennenlernen werden, untersuchen jeden Knoten (und jede Kante).

Es gibt verschiedene **Graphendurchlaufstrategien** (traversal strategies), die jeden Knoten (oder jede Kante) besuchen:

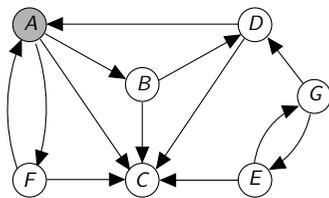
- ▶ **Tiefensuche**
- ▶ **Breitensuche**

- ▶ Dies sind Verallgemeinerungen von Strategien zur Baumtraversierung.
- ▶ Da Graphen zyklisch sein können, müssen wir uns aber alle bereits gefundenen Knoten merken.
- ▶ Im Folgenden nehmen wir die **Adjazenzlisten**-Darstellung an.
- ▶ Dann ist der Zeitaufwand von Algorithmen in  $O(|V| + |E|)$ .

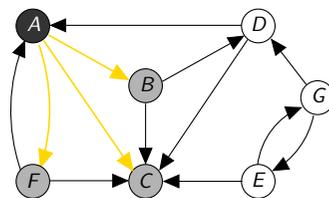
## Übersicht

- 1 Softwarewettbewerb
- 2 Graphen
  - Terminologie
  - Repräsentation von Graphen
- 3 Graphendurchlauf
  - Breitensuche
  - Tiefensuche
- 4 Anwendungen der Tiefensuche
  - Erreichbarkeitsanalyse
  - CCs in ungerichteten Graphen
  - SCCs in gerichteten Graphen
  - Topologische Sortierung in gerichteten azyklischen Graphen
  - Kritische-Pfad-Analyse in gewichteten gerichteten azyklischen Graphen

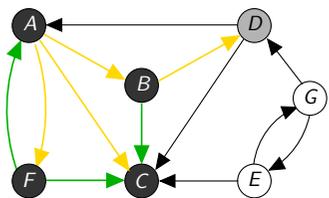
## Breitensuche: Beispiel



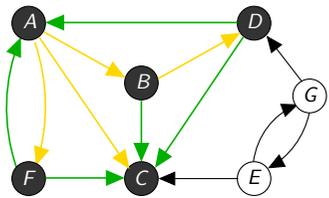
Beginn der Breitensuche



Erforsche alle folgenden nicht-gefundenen Knoten



Erforsche alle folgenden nicht-gefundenen Knoten



Fertig!

## Breitensuche

### Breitensuche (Breadth-First Search, BFS)

Am Anfang seien alle Knoten als "nicht-gefunden" (WHITE) markiert. Die zugrundeliegende Strategie ist:

- ▶ Markiere den aktuellen Knoten  $v$  als "gefunden" (GRAY).
  - ▶ Suche „gleichzeitig“ aus allen "gefundenen" Knoten weiter:
    - ▶ Markiere alle ihrer noch "nicht-gefundenen" Nachfolger als "gefunden" und
    - ▶ markiere die Knoten selbst als "abgeschlossen" (BLACK).
- ▶ Man erhält die Menge aller Knoten, die vom Startknoten aus **erreichbar** sind.

## Breitensuche: Implementierung

```

1 void BFS(List adj[n], int start, int &color[n]) {
2   Queue wait; // zu verarbeitende Knoten
3   color[start] = GRAY; // Knoten start ist noch zu verarbeiten
4   wait.enqueue(start);
5   while (!wait.isEmpty()) { // es gibt noch unverarbeitete Knoten
6     int v = wait.dequeue(); // nächster unverarbeiteter Knoten
7     foreach (w in adj[v]) {
8       if (color[w] == WHITE) { // neuer ("ungefundener") Knoten
9         color[w] = GRAY; // w ist noch zu verarbeiten
10        wait.enqueue(w);
11      }
12    }
13    color[v] = BLACK; // v ist abgeschlossen
14  }
15 }

17 void completeBFS(List adj[n], int n) {
18   int color[n] = WHITE; // noch kein Knoten ist gefunden worden
19   for (int i = 0; i < n; i++)
20     if (color[i] == WHITE) BFS(adj, n, i, color);
21 }

```

## Eigenschaften der Breitensuche

- ▶ Knoten werden in der Reihenfolge mit **zunehmendem kürzesten Abstand (Kantendistanz)** vom Startknoten aus besucht.
  - ▶ Nachdem alle Knoten mit Abstand  $d$  verarbeitet wurden, werden die mit  $d + 1$  angegangen.
  - ▶ Die Suche terminiert, wenn in Abstand  $d$  keine neuen Knoten auftreten.
- ▶ Die Tiefe des Knotens  $v$  im Breitensuchbaum ist seine **kürzeste Kantendistanz** zum Startknoten.
- ▶ Die zu verarbeitenden Knoten werden als **FIFO-Queue** (first-in first-out) organisiert.

### Theorem (Komplexität der Breitensuche)

Die Zeitkomplexität ist  $\mathcal{O}(|V| + |E|)$ , der Platzbedarf  $\Theta(|V|)$ .

## Tiefensuche

### Tiefensuche (Depth-First Search, DFS)

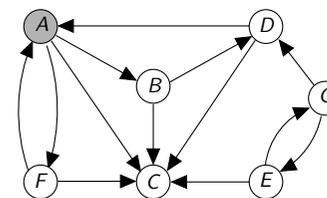
Am Anfang seien alle Knoten als „nicht-gefunden“ (WHITE) markiert. Die zugrundeliegende Strategie ist:

- ▶ Markiere den aktuellen Knoten  $v$  als „gefunden“ (GRAY).
  - ▶ Solange es noch eine Kante  $(v, u)$  mit „nicht-gefundenem“ Nachfolger  $u$  gibt:
    - ▶ Suche rekursiv von  $u$  aus, d. h.:
    - ▶ Erforsche Kante  $(u, w)$ , besuche  $w$ , forsche von dort aus, bis es nicht mehr weiter geht.
    - ▶ Markiere  $u$  als „abgeschlossen“ (BLACK).
    - ▶ Backtracke von  $u$  nach  $v$ .
  - ▶ Markiere  $v$  als „abgeschlossen“ (BLACK).
- ▶ Man erhält wieder die Menge aller Knoten, die vom Startknoten aus **erreichbar** sind.

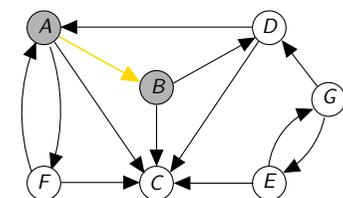
## Übersicht

- 1 Softwarewettbewerb
- 2 Graphen
  - Terminologie
  - Repräsentation von Graphen
- 3 Graphendurchlauf
  - Breitensuche
  - Tiefensuche
- 4 Anwendungen der Tiefensuche
  - Erreichbarkeitsanalyse
  - CCs in ungerichteten Graphen
  - SCCs in gerichteten Graphen
  - Topologische Sortierung in gerichteten azyklischen Graphen
  - Kritische-Pfad-Analyse in gewichteten gerichteten azyklischen Graphen

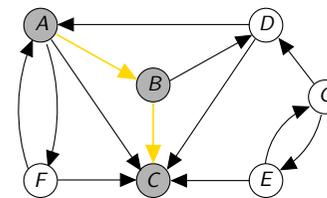
## Tiefensuche: Beispiel



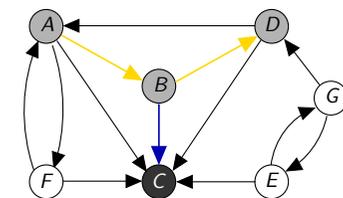
Beginn der Tiefensuche



Erforsche einen Knoten



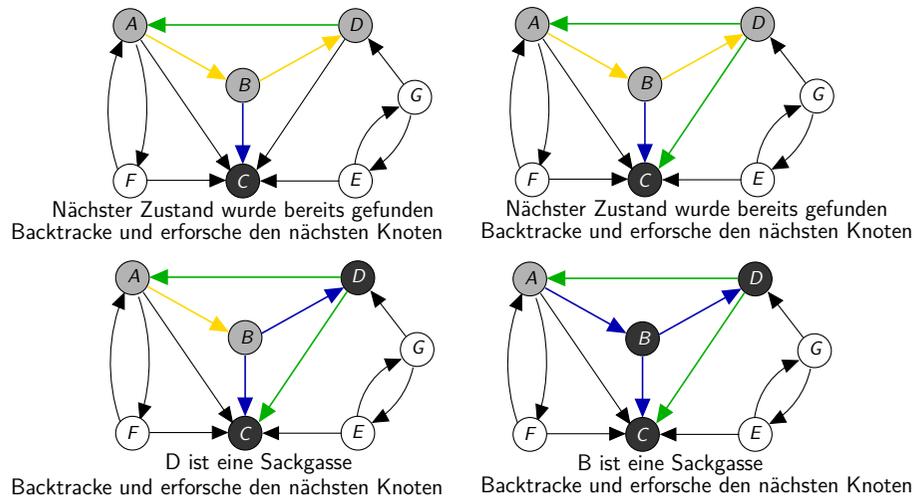
Erforsche einen Knoten



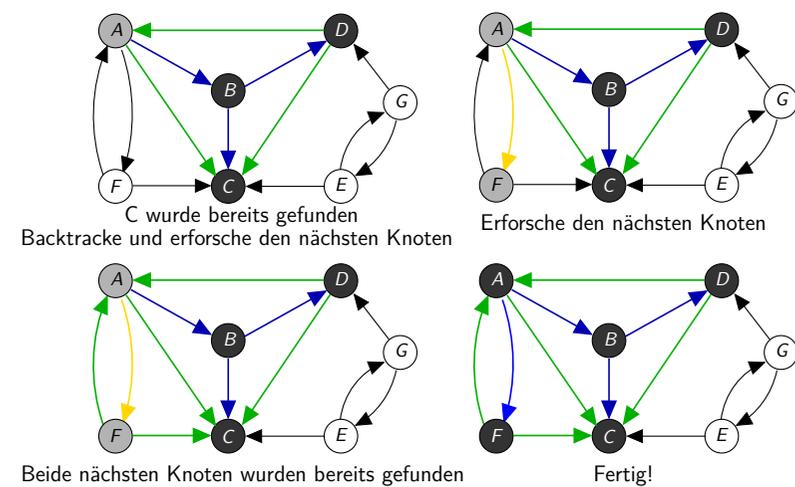
Sackgasse!

Backtracke und erforsche den nächsten Knoten

## Tiefensuche: Beispiel



## Tiefensuche: Beispiel



## Tiefensuche: Implementierung

```

1 void DFS(List adj[n], int start, int &color[n]) {
2   color[start] = GRAY; // start ist noch zu verarbeiten
3   foreach (w in adj[start]) {
4     if (color[w] == WHITE) { // neuer ("ungefundener") Knoten
5       DFS(adj, w, color);
6     }
7   }
8   color[start] = BLACK; // start ist abgeschlossen
9 }

11 void completeDFS(List adj[n], int n, int start) {
12   int color[n] = WHITE; // noch kein Knoten ist gefunden worden
13   for (int i = 0; i < n; i++)
14     if (color[i] == WHITE) DFS(adj, i, color);
15 }

```

## Eigenschaften der Tiefensuche

- ▶ Erforsche einen Pfad **so weit wie möglich** bevor backtracking.
  - ▶ Die zu verarbeitenden Knoten werden in LIFO-Reihenfolge (last-in first-out) geprüft.
  - ▶ Es gibt **zwei** „Verarbeitungsmöglichkeiten“ für einen Knoten:
    1. Wenn der Knoten entdeckt wird.
    2. Wenn der Knoten als „abgearbeitet“ markiert wird (und alle seine Nachfolger entdeckt werden).
- ⇒ Diese letztgenannte Möglichkeit macht Tiefensuche beliebt.

### Theorem (Komplexität der Tiefensuche)

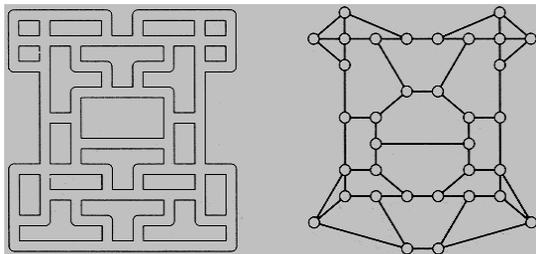
Zeitkomplexität:  $\mathcal{O}(|V| + |E|)$ , Platzkomplexität:  $\Theta(|V|)$ .

## Übersicht

- 1 Softwarewettbewerb
- 2 Graphen
  - Terminologie
  - Repräsentation von Graphen
- 3 Graphendurchlauf
  - Breitensuche
  - Tiefensuche
- 4 Anwendungen der Tiefensuche
  - Erreichbarkeitsanalyse
  - CCs in ungerichteten Graphen
  - SCCs in gerichteten Graphen
  - Topologische Sortierung in gerichteten azyklischen Graphen
  - Kritische-Pfad-Analyse in gewichteten gerichteten azyklischen Graphen

## Geschichte

- ▶ **Tiefensuche** wurde bereits vor Jahrhunderten formal beschrieben als ein Verfahren zum Durchqueren von **Labyrinthen**.
- ▶ Labyrinth als Graph:
  - ▶ **Knoten**: Stellen, an denen mehr als ein Weg gewählt werden kann
  - ▶ **Kanten**: Wege zwischen Knoten
  - ▶ **Eingang** als spezieller Knoten start
  - ▶ **Ziel** als spezieller Knoten target



## Irrgärten

“Das eben geschieht den Menschen, die in einem Irrgarten hastig werden:  
Eben die Eile führt immer tiefer in die Irre.”

[Lucius Annaeus Seneca (4 n. Chr. – 65 n. Chr.)]



## Erreichbarkeitsanalyse: DFS Implementierung

```

1 bool DFS(List adj[n], int start, int &color[n],
2         int target, List path) {
3     if (start == target){
4         path.addAtFront(start); return true;
5     }
6     color[start] = GRAY;
7     foreach (w in adj[start]) {
8         if (color[w] == WHITE) {
9             if (DFS(adj, w, color, target, path)) {
10                path.addAtFront(start); return true;
11            }
12        }
13    }
14    color[start] = BLACK;
15    return false;
16 }
17 bool reach(List adj[n], int n, int start, int target,
18           List &path) { // path ist leer
19     int color[n] = WHITE;
20     return DFS(adj, start, color, target, path);
21 }

```

## Übersicht

- 1 Softwarewettbewerb
- 2 Graphen
  - Terminologie
  - Repräsentation von Graphen
- 3 Graphendurchlauf
  - Breitensuche
  - Tiefensuche
- 4 Anwendungen der Tiefensuche
  - Erreichbarkeitsanalyse
  - CCs in ungerichteten Graphen
  - SCCs in gerichteten Graphen
  - Topologische Sortierung in gerichteten azyklischen Graphen
  - Kritische-Pfad-Analyse in gewichteten gerichteten azyklischen Graphen

## CCs in ungerichteten Graphen

### Problem

Finde die Zusammenhangskomponenten (CCs) eines *ungerichteten* Graphen  $G$ .

### Lösung

- ▶ Finden des CCs eines Knotens  $v$ :  
Verwende Tiefen- oder Breitensuche, um alle aus  $v$  erreichbaren Knoten zu bestimmen.
- ▶ Die Zeitkomplexität ist  $\mathcal{O}(|V| + |E|)$ .

## CCs in ungerichteten Graphen

### Zusammenhangskomponenten in ungerichteten Graphen

Sei  $G$  ein *ungerichteter* Graph.

- ▶  $G$  heißt *zusammenhängend* (*connected*), wenn jeder Knoten von jedem anderen Knoten aus erreichbar ist.
- ▶ Eine *Zusammenhangskomponente* (*connected component, CC*) von  $G$  ist ein maximaler (d.h. nicht erweiterbarer) zusammenhängender Teilgraph von  $G$ .

## CCs in ungerichteten Graphen

```

1 void DFS(List adj[n], int start, int &color[n],
2           int leader, int &cc[n]) {
3   color[start] = GRAY;
4   foreach (next in adj[start])
5     if (color[next] == WHITE)
6       DFS(adj, next, color, leader, cc);
7   color[start] = BLACK;
8   cc[start] = leader; // speichere das CC von start
9 }

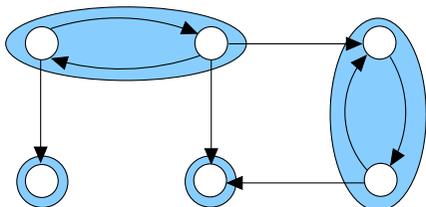
11 void connComponents(List adj[n], int n, int &cc[n]) {
12   int color[n] = WHITE;
13   for (int v = 0; v < n; v++)
14     if (color[v] == WHITE) // weitere Komponente
15       DFS(adj, v, color, v, cc);
16 } // Ausgabe in cc: cc[v] = Komponente von Knoten v

```

## Übersicht

- 1 Softwarewettbewerb
- 2 Graphen
  - Terminologie
  - Repräsentation von Graphen
- 3 Graphendurchlauf
  - Breitensuche
  - Tiefensuche
- 4 Anwendungen der Tiefensuche
  - Erreichbarkeitsanalyse
  - CCs in ungerichteten Graphen
  - SCCs in gerichteten Graphen
  - Topologische Sortierung in gerichteten azyklischen Graphen
  - Kritische-Pfad-Analyse in gewichteten gerichteten azyklischen Graphen

## Starke Zusammenhangskomponenten (SCCs)



Ein Digraph und seine SCCs.

## SCCs in gerichteten Graphen

### Zusammenhangskomponenten

Sei  $G$  ein gerichteter Graph.

- ▶  $G$  heißt **stark zusammenhängend** (**strongly connected**), wenn jeder Knoten von jedem anderen aus erreichbar ist.
- ▶ Eine **starke Zusammenhangskomponente** (**strongly connected component, SCC**) von  $G$  ist ein maximaler (d.h., nicht erweiterbarer) stark zusammenhängender Teilgraph von  $G$ .

## Kondensationsgraph

Die starken Zusammenhangskomponenten von  $G$  induzieren den **Kondensationsgraph**.

### Kondensationsgraph

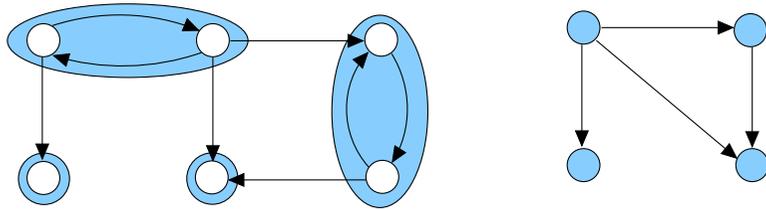
Sei  $G = (V, E)$  ein gerichteter Graph mit  $k$  SCCs  $S_i = (V_i, E_i)$  für  $0 < i \leq k$ .

Der **Kondensationsgraph**  $G_{\downarrow} = (V', E')$  ist definiert als:

- ▶  $V' = \{V_1, \dots, V_k\}$ .
- ▶  $(V_i, V_j) \in E'$  gdw.  $i \neq j$  und es gibt  $(v, w) \in E$  mit  $v \in V_i$  und  $w \in V_j$ .

Der Kondensationsgraph  $G_{\downarrow}$  ist **azyklisch**.

## Kondensationsgraph: Beispiel



Ein Digraph und seine Kondensation.

## SCCs und Transponierung

### Transponieren

Der **transponierte** Graph von  $G = (V, E)$  ist  $G^T = (V, E')$  mit  $(v, w) \in E'$  gdw.  $(w, v) \in E$ .

In  $G^T$  ist die Richtung der Kanten von  $G$  gerade umgedreht.

### Lemma: Beziehung zwischen $G$ und $G^T$

1. Die SCCs von  $G$  und  $G^T$  sind **die selben**.
2. Die Kondensation und die Transposition **kommutieren**, d. h.:

$$(G \downarrow)^T = (G^T) \downarrow.$$

## Algorithmus zum Finden von SCCs

**Kosaraju-Sharir Algorithmus** findet SCCs in **zwei** Phasen:

1. Führe ein DFS auf  $G$  durch, wobei alle Knoten beim Abschließen (d. h. wenn der Knoten BLACK gefärbt wird) auf einem Stack  $S$  gespeichert werden.
2. Färbe alle Knoten wieder WHITE.
3. Wiederhole solange  $S$  noch weiße Knoten enthält:
  - ▶ Wähle den obersten noch weißen (Leiter-)Knoten  $v$  vom  $S$ .
  - ▶ Führe DFS mit Startknoten  $v$  auf dem transponierten Graphen  $G^T$  aus und speichere für jeden besuchten Knoten den **Leiterknoten**  $v$  als Repräsentanten seines SCCs.

### Leiter einer SCC

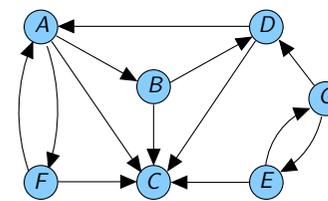
Ein Knoten  $v$  heißt **Leiter** (leader), wenn er als **letzter** Knoten aus seinem SCC bei einer DFS BLACK gefärbt wird.

```

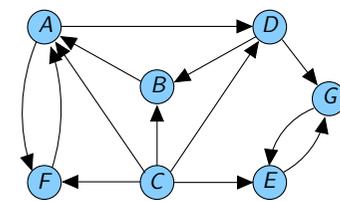
1 void DFS1(List adj[n], int start, int &color[n], Stack &S) {
2   color[start] = GRAY;
3   foreach (w in adj[start])
4     if (color[w] == WHITE) DFS1(adj, w, color, S);
5   color[start] = BLACK; S.push(start);
6 }
7 void DFS2(List adj[n], int start, int &color[n], int leader,
8           int &scc[n]) {
9   color[start] = GRAY;
10  foreach (w in adj[start])
11    if (color[w] == WHITE) DFS2(adj, w, color, leader, scc);
12  color[start] = BLACK; scc[start] = leader;
13 }
14 void kosaraju-sharir(List adj[n], int n) {
15   int color[n] = WHITE; Stack S; int scc[n];
16   for (int i = 0; i < n; i++) // Phase 1
17     if (color[i] == WHITE) DFS1(adj, i, color, S);
18   List adj_T = adj^T; // Transponiere
19   for (int i = 0; i < n; i++) color[i] = WHITE; // Phase 2
20   while (!S.empty()) { // Phase 3
21     int v = S.pop();
22     if (color[v] == WHITE) DFS2(adj_T, v, color, v, scc);
23   }
24 }

```

## Kosaraju-Sharir Algorithmus: Beispiel



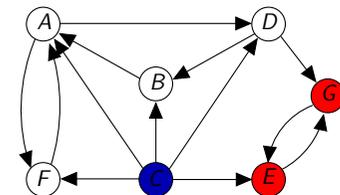
Ursprünglicher Digraph



Transponierter Graph



Stack am Ende der Phase 1



In Phase 2 gefundene starke Komponenten

# Korrektheit

# Übersicht

- 1 Softwarewettbewerb
- 2 Graphen
  - Terminologie
  - Repräsentation von Graphen
- 3 Graphendurchlauf
  - Breitensuche
  - Tiefensuche
- 4 **Anwendungen der Tiefensuche**
  - Erreichbarkeitsanalyse
  - CCs in ungerichteten Graphen
  - SCCs in gerichteten Graphen
  - Topologische Sortierung in gerichteten azyklischen Graphen
  - Kritische-Pfad-Analyse in gewichteten gerichteten azyklischen Graphen

# Komplexität

## Zeitkomplexität

Die Worst-Case Zeitkomplexität vom Kosaraju-Sharir Algorithmus zum Finden von SCCs in einem gerichteten Graphen ist in  $\Theta(|V| + |E|)$ . Seine Speicherkomplexität ist in  $\Theta(|V|)$ .

## Beweis

- ▶ Die DFS über  $G$  und  $G^T$  benötigen jeweils  $\Theta(|V| + |E|)$ .
- ▶ Der transponierte Graph  $G^T$  kann in  $\Theta(|V| + |E|)$  gebildet werden.
- ▶ Der Stack benötigt  $\Theta(|V|)$  Speicher.

# Gerichtete azyklische Graphen

## Gerichtete azyklische Graphen

**Gerichtete azyklische Graphen** (directed acyclic graph, DAG) sind eine wichtige Klasse von Graphen:

- ▶ Viele Probleme lassen sich naturgemäß mit Hilfe von DAGs formulieren.
  - ▶ Scheduling: Vorranggraphen beschreiben, welche Aufgaben erledigt sein müssen, bevor ein nachfolgender Schritt beginnen kann.
  - ▶ Ein Zyklus in solch einem Vorranggraphen wäre ein Deadlock.
- ▶ Viele Probleme haben auf DAGs eine niedrigere Komplexität als auf Digraphen.
- ▶ Ein DAG entspricht einer **partiellen Ordnung**  $<$  auf den Knoten:
  - ▶ Eine Kante  $(v, w)$  besagt:  $v > w$ .
  - ⇒ Da eine partielle Ordnung anti-symmetrisch ist, kann sie keinen Zyklus enthalten.
- ▶ Wir betrachten: **Topologische Sortierung** und **Kritische-Pfad-Analyse**.

## Topologische Ordnung

### Topologische Ordnung

Sei  $G = (V, E)$  ein gerichteter Graph mit  $n$  Knoten. Eine **topologische Ordnung** von  $G$  ist eine Zuordnung  $topo : V \rightarrow \{1, \dots, n\}$ , so dass:

für jede Kante  $(v, w) \in E$  gilt:  $topo(v) > topo(w)$ .

$topo(v)$  heißt der **topologische Zahl** von  $v$ .

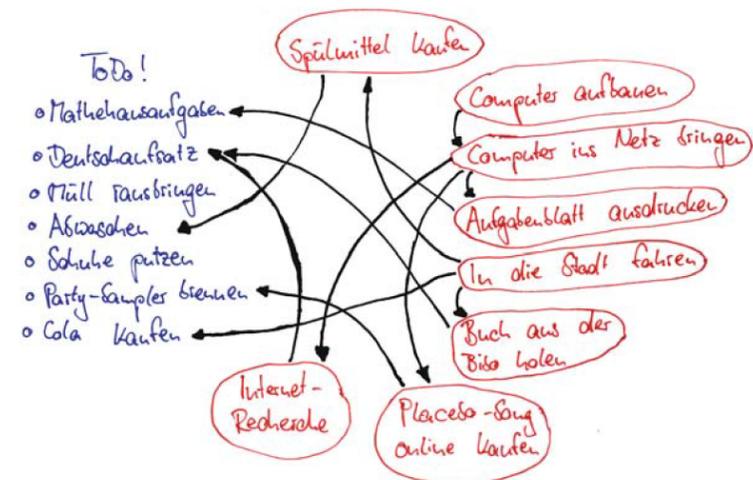
### Lemma

1. Für einen Digraph  $G$  mit einem Zyklus existiert **keine** topologische Ordnung.
2. Jeder DAG  $G$  dagegen hat mindestens eine topologische Ordnung.

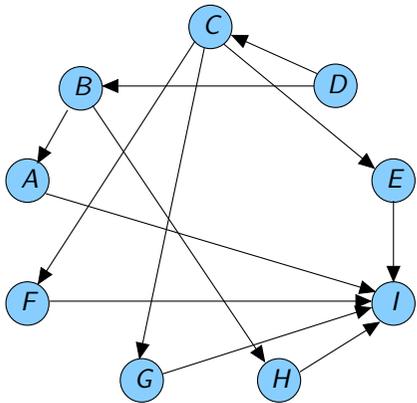
## Topologische Sortierung: Beispiel 1

- ToDo!
- Mathehausaufgaben
  - Deutschsaufsatz
  - Müll rausbringen
  - Abwaschen
  - Schuhe putzen
  - Party-Sampler brennen
  - Cola kaufen

## Topologische Sortierung: Beispiel 1



## Topologische Sortierung: Beispiel 2

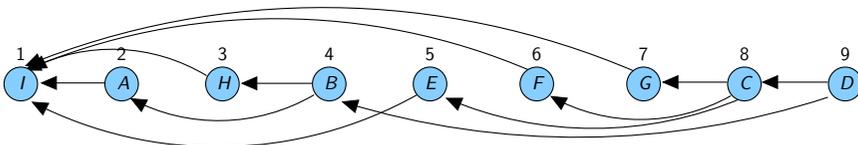


Abhängigkeitsgraph

Nr	Aufgabe	Hängt ab von
A	choose clothes	I
B	dress	A, H
C	eat breakfast	E, F, G
D	leave	B, C
E	make coffee	I
F	make toast	I
G	pour juice	I
H	shower	I
I	wake up	-

► Gibt es einen **Schedule** für dieses Problem? D. h. kann man eine Reihenfolge finden, um alle Aufgaben ausführen zu können?

## Topologische Sortierung: Ergebnis Beispiel 2



I	A	H	B	E	F	G	C	D
wake up	choose clothes	shower	dress	make coffee	make toast	pour juice	eat breakfast	leave

Abhängigkeitsgraph, der topologischen Ordnung entsprechend gezeichnet.

## Topologische Sortierung: Implementierung

```

1 void DFS(List adj[n], int start, int &color[n],
2           int &topoNum, int &topo[n]) {
3     color[start] = GRAY;
4     foreach (next in adj[start]) {
5         // if (color[next] == GRAY) throw "Graph ist zyklisch";
6         if (color[next] == WHITE) {
7             DFS(adj, next, color, topoNum, topo);
8         }
9     }
10    topo[start] = ++topoNum;
11    color[start] = BLACK;
12 }

14 // Ausgabe der topologischen Zahl von Knoten v in topo[v]
15 void topoSort(List adj[n], int n, int &topo[n]) {
16     int color[n] = WHITE, topoNum = 0;
17     for (int v = 0; v < n; v++)
18         if (color[v] == WHITE)
19             DFS(adj, v, color, topoNum, topo);
20 }
    
```

## Korrektheit und Komplexität

### Theorem

Der Algorithmus terminiert, und wenn er terminiert enthält das Array topo eine topologische Ordnung von Digraph G.

### Beweis:

- Die DFS besucht jeden Knoten, daher sind die Zahlen in dem Array topo alle verschieden im Bereich 1 bis  $|V|$ .
- Sei  $(v, w) \in E$ .
  - w ist kein Vorgänger von v im DFS-Baum, sonst wäre G zyklisch.
  - Daher ist w BLACK, wenn topo[v] ein Wert zugewiesen wird.
  - Also wurde topo[w] schon vorher ein Wert zugewiesen.
  - Da topoNum immer größer wird, folgt topo[v] > topo[w].

### Zeitkomplexität

Eine topologische Ordnung kann in  $O(|V| + |E|)$  bestimmt werden.

## Übersicht

- 1 Softwarewettbewerb
- 2 Graphen
  - Terminologie
  - Repräsentation von Graphen
- 3 Graphendurchlauf
  - Breitensuche
  - Tiefensuche
- 4 Anwendungen der Tiefensuche
  - Erreichbarkeitsanalyse
  - CCs in ungerichteten Graphen
  - SCCs in gerichteten Graphen
  - Topologische Sortierung in gerichteten azyklischen Graphen
  - Kritische-Pfad-Analyse in gewichteten gerichteten azyklischen Graphen

## Gewichtete Graphen: Darstellung

Gewichtete Graphen werden ebenso als Adjazenzlisten oder Adjazenzmatrix dargestellt:

- ▶ Bei **knotengewichteten** Graphen wird die Zusatzinformation zu den Knoten üblicherweise in einem weiteren Array gespeichert – vgl. `int color[n]`; bei BFS oder DFS.
- ▶ **Kantengewichte** können bei der Adjazenzmatrixdarstellung direkt in der Matrix gespeichert werden. Ein besonderer Wert, etwa  $\infty$  besagt, dass keine Kante existiert.

## Gewichtete Graphen

### Knotengewichteter Graph

Ein **knotengewichteter** Graph  $G$  ist ein Tripel  $(V, E, W)$ , wobei:

- ▶  $(V, E)$  ein – gerichteter oder ungerichteter – Graph ist, und
- ▶  $W : V \rightarrow \mathbb{R}$  **Gewichtsfunktion**.  $W(v)$  ist das Gewicht des Knotens  $v$ .

### Kantengewichteter Graph

Ein **(kanten-)gewichteter** Graph  $G$  ist ein Tripel  $(V, E, W)$ , wobei:

- ▶  $(V, E)$  ein – gerichteter oder ungerichteter – Graph ist, und
- ▶  $W : E \rightarrow \mathbb{R}$  **Gewichtsfunktion**.  $W(e)$  ist das Gewicht der Kante  $e$ .
- ▶ Ein knotengewichteter Graph  $(V, E, W)$  lässt sich in einen kantengewichteten Graphen  $(V, E, W')$  überführen, indem *alle* von einem Knoten  $v$  ausgehenden Kanten  $e = (v, \cdot) \in E$  das Gewicht  $W'(e) = W(v)$  erhalten.

## Das Kritische-Pfad-Problem: Einführung

Das Gewicht eines Pfades ist die Summe der Kantengewichten der besuchten Kanten, oder die Summe der Knotengewichte der besuchten Knoten.

### Kritischer-Pfad-Problem

Finde den **längsten** Pfad (bezogen auf das Gesamtgewicht) in einem (kanten- oder knoten-)gewichteten **DAG**.

- ▶ Wir betrachten hier *nur* **knotengewichtete** DAGs.

### Beispiel (Anwendung)

Wie lange benötigt man für die Ausführung der bereits vorgestellten Aufgaben mindestens, wenn für jede Aufgabe eine Dauer gegeben ist und unabhängige Aufgaben gleichzeitig erledigt werden können?

## Das Kritische-Pfad-Problem: Anwendung

### Kritische Pfad

Der **kritische Pfad** ist eine Folge von Aufgaben  $v_0, \dots, v_k$ , so dass

- ▶  $v_0$  keine Abhängigkeiten hat.
- ▶  $v_i$  abhängig von  $v_{i-1}$  ist, wobei  $est(v_i) = eft(v_{i-1})$ .
- ▶  $eft(v_k)$  das Maximum über alle Aufgaben ergibt.

Dann gibt es eine **kritische** Abhängigkeit zwischen  $v_{i-1}$  und  $v_i$ , d.h. eine Verzögerung in  $v_{i-1}$  führt zu einer Verzögerung in  $v_i$ .

## Das Kritische-Pfad-Problem: Anwendung

### Früheste Startzeit- und Endzeitpunkt

Finde den **frühestmöglichen Beendigungszeitpunkt** (earliest finish time, eft) für eine Menge voneinander abhängiger Aufgaben.

- ▶ Jede Aufgabe hat eine (nicht-negative) **Dauer**.
- ▶ Der **früheste Startzeitpunkt** (earliest start time, est) für Aufgabe  $v$  ( $est(v)$ ) ist 0 wenn  $v$  keine Abhängigkeiten hat; andernfalls:
- ▶  $est(v)$  ist das Maximum der frühesten Endzeitpunkte seiner Abhängigkeiten.

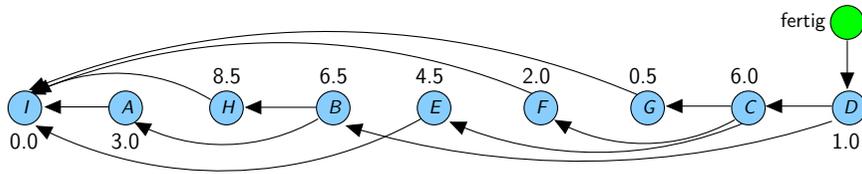
Der **früheste Endzeitpunkt** (earliest finish time) für Aufgabe  $v$  ( $eft(v)$ ) ist gleich  $est(v)$  plus der Dauer von  $v$ .

## Kritische-Pfad-Analyse: Implementierung

```

1 // Knotengewichte in duration.
2 // Ausgabe: eft, kritischer Pfad kodiert in critDep
3 void DFS(List adj[n], int start, int &color[n],
4           int duration[n], int &critDep[n], int &eft[n]) {
5   color[start] = GRAY; critDep[start] = -1; int est = 0;
6   foreach (next in adj[start]) {
7     if (color[next] == WHITE)
8       DFS(adj, next, color, duration, critDep, eft);
9     if (eft[next] >= est) {
10      est = eft[next]; critDep[start] = next;
11    }
12  }
13  eft[start] = est + duration[start];
14  color[start] = BLACK;
15 }
16 void critPath(List adj[n], int n,
17               int duration[n], int &critDep[n], int &eft[n]){
18   int color[n] = WHITE;
19   for (int i = 0; i < n; i++)
20     if (color[i] == WHITE)
21       DFS(adj, i, color, duration, critDep, eft);
22 }
```

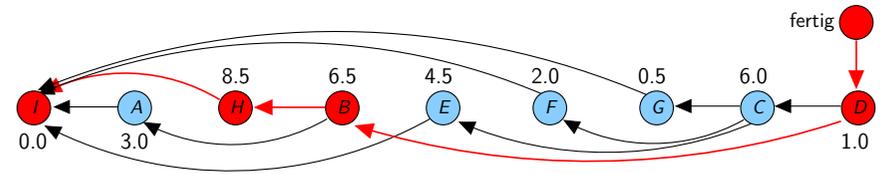
### Kritische-Pfad-Analyse: Beispiel



I	A	H	B	E	F	G	C	D
wake up	choose clothes	shower	dress	make coffee	make toast	pour juice	eat breakf	leave
0.0	3.0	8.5	6.5	4.5	2.0	0.5	6.0	1.0

Dauer

### Kritische-Pfad-Analyse: Beispiel



I	A	H	B	E	F	G	C	D
wake up	choose clothes	shower	dress	make coffee	make toast	pour juice	eat breakf	leave
0.0	3.0	8.5	6.5	4.5	2.0	0.5	6.0	1.0

Dauer

►  $eft = 1 + 6.5 + 8.5 + 0 = 16.$