

Graphen

Algorithmen und Datenstrukturen 2

- Grüne Farbe: Bitte im Script nachtragen

Graphentheorie (Repetition)

- Graphentheorie (wird als bekannt vorausgesetzt)
- Zusätzliches Dokument auf dem Netzlaufwerk:

08-1 Graphen - einige Definitionen

Anwendungsfälle Graphen

- Navigation
- Soziale Netzwerkanalyse
- Projektpläne
- Garbage Collection in Java

Speichern von Graphen (Kap. 7.4)

- Graph-Eigenschaften
 - Anzahl Knoten / Kanten
 - Verhältnis Knoten / Kanten
 - Parallele Kanten / Verschiedene Kanten-Eigenschaften
- Algorithmen => Graph-Operationen

Speichern von Graphen – Adjazenzmatrix (1)

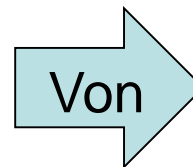
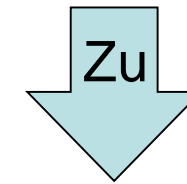
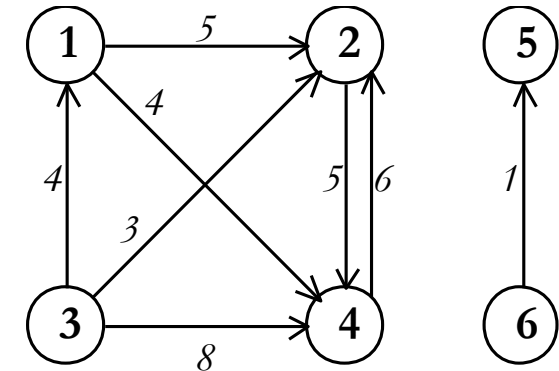
- $n \times n$ -Matrix (n = Anzahl Knoten) (speicherhungrig)

- Gewichtet: Integer-Matrix

```
int[][] g = new int[N][N];
```

- Ungewichtet: Boolean-Matrix

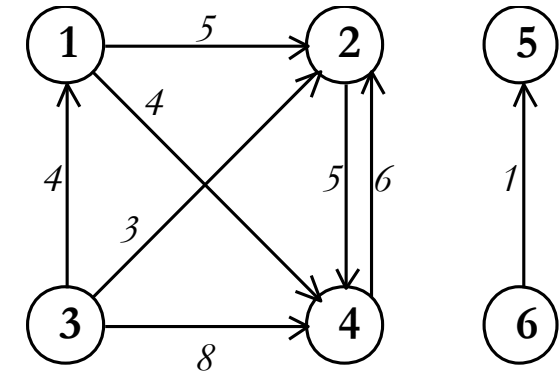
```
boolean[][] g = new boolean[N][N];
```



	1	2	3	4	5	6
1	-	5	-	4	-	-
2	-	-				
3	4	3				
4						
5						
6						

Speichern von Graphen – Adjazenzmatrix (2)

- **Abbildung: Keine Kante**
- Abbildung Abhängig von Interpretation
 - Distanz: Integer.MAX_VALUE
 - Kapazität: 0
- => Keine Fallunterscheidung bei Algorithmen nötig
- Ungerichtete Graphen: $A = A^T$

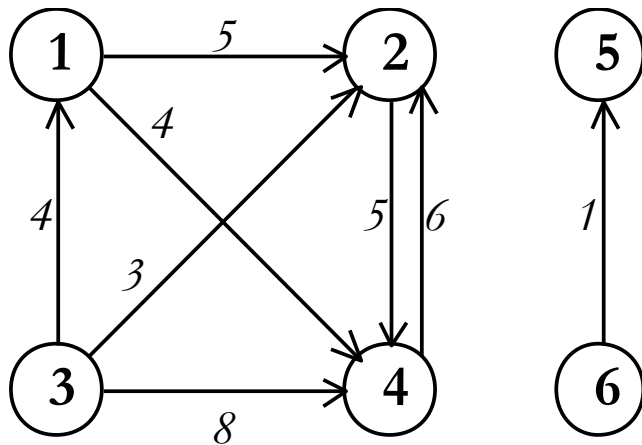


	1	2	3	4	5	6
1	-	5	-	4	-	-
2	-	-				
3	4	3				
4						
5						
6						

Speichern von Graphen - Kantentabelle

- Tabelle oder Liste mit Edge-Einträgen:
 - Edge: from, to, weight

```
class Edge {
    int from, to, weight;
}
```

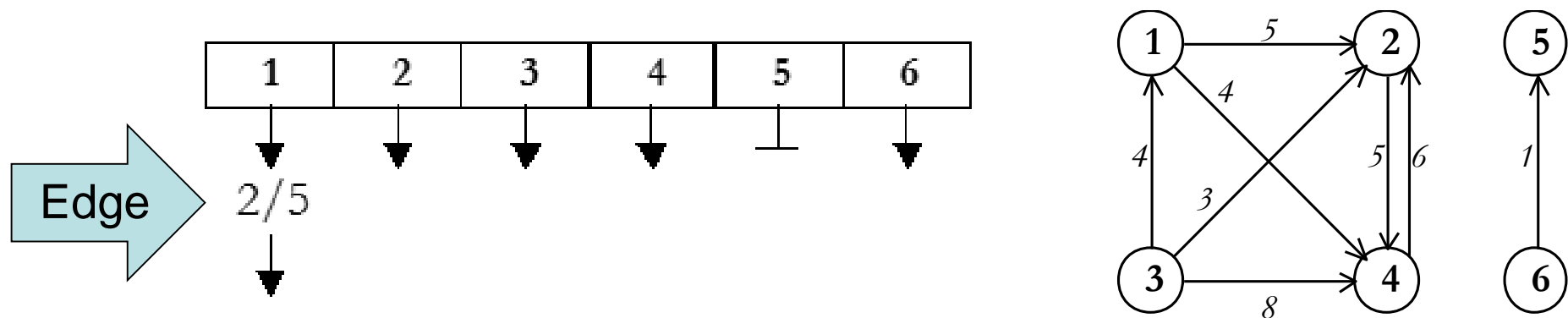


from	1							
to	2							
weight	5							

- Ungerichtete Graphen: 2x eintragen oder from / to gleich behandeln

Speichern von Graphen – Adjazenzlisten (Verbindungslisten) (1)

- Array und pro Knoten eine Liste mit ausgehenden Kanten



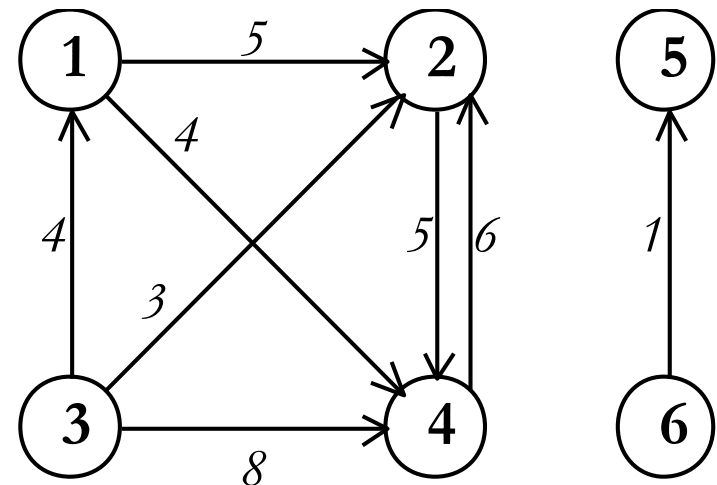
- Ungerichtete Graphen: Einträge bei beiden Knoten (2 Kanten)

Aufgabe: Speichern von Graphen

Ergänzen Sie die Einträge für die Graph-Repräsentationen im Script:

- Adjazenzmatrix (Kap. 7.4.1)
- Kantentabelle (Kap. 7.4.2)
- Adjazenzliste (Kap. 7.4.3)

für den im Script abgebildeten Beispielgraphen:



Aufgaben - Lösungen

Adjazenzliste

1	2	3	4	5	6
↓	↓	↓	↓	↓	↓
2/5	4/5	1/4	2/6		5/1
↓		↓			
4/4		2/3			
		↓			
		4/8			

Adjazenzmatrix

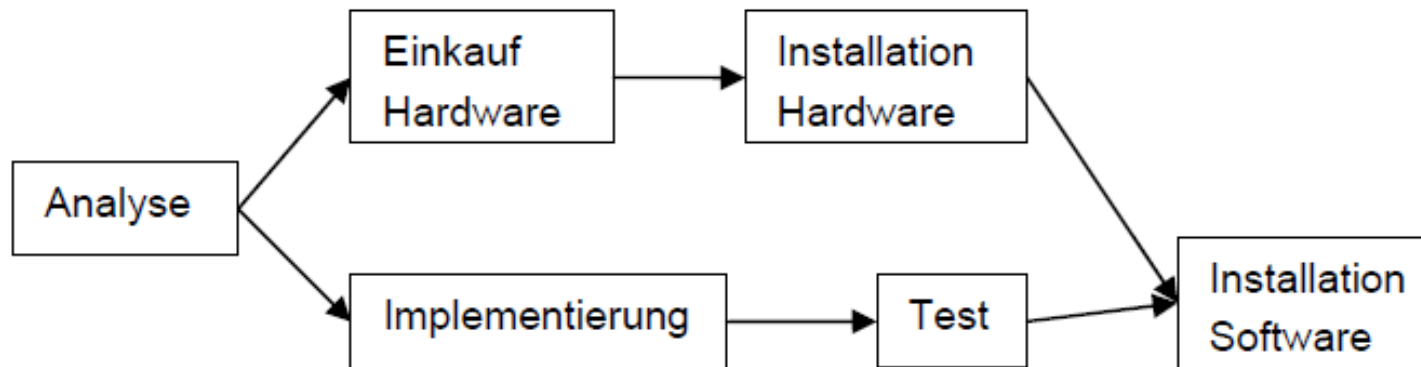
	1	2	3	4	5	6
1	-	5	-	4	-	-
2	-	-	-	5	-	-
3	4	3	-	8	-	-
4	-	6	-	-	-	-
5	-	-	-	-	-	-
6	-	-	-	-	1	-

Kantenliste

fr	1	1	2	3	3	3	4	6
to	2	4	4	1	2	4	2	5
weight	5	4	5	4	3	8	6	1

Topologisches Sortieren (Kap. 7.5) - Idee

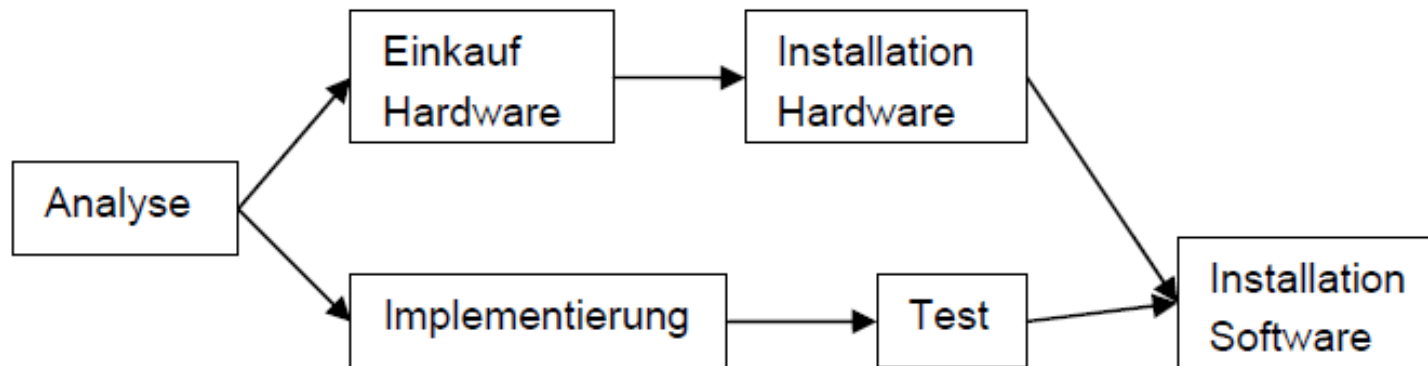
- Reihenfolge ohne Abhängigkeits-Konflikte
- Kanten-Bedeutung: «ist Vorbedingung für»



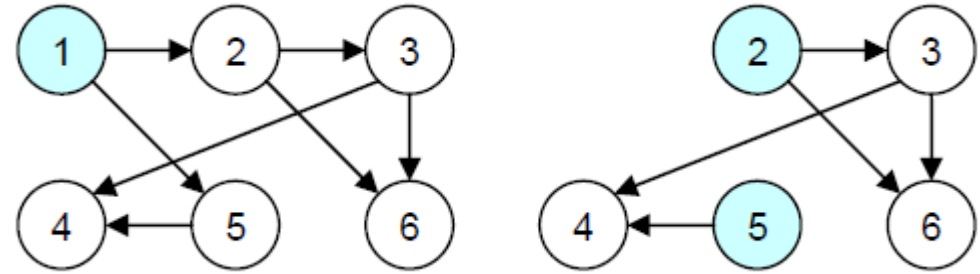
- Verschiedene «Topologische Sortierungen» möglich

Topologisches Sortieren - Vorbedingungen

- Gerichteter Graph
 - Zyklensfrei
- } Directed Acyclic Graph (DAG)



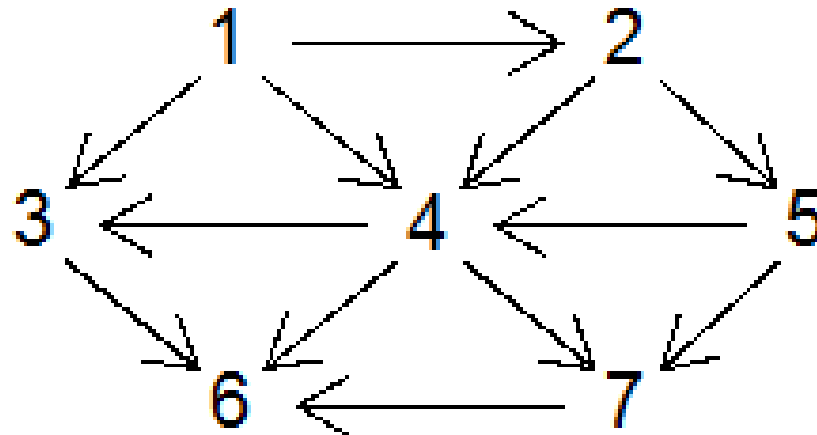
Topologisches Sortieren - Algorithmus



1. Bestimme für alle Knoten den Indegree
2. Sammle alle Knoten mit Indegree == 0 in einem Set Z
3. Solange Elemente in Z
 1. Nehme Knoten v aus Z
 2. Füge v in topologisch sortierte Reihenfolge ein
 3. Für alle von v ausgehenden Kanten (v,w), reduziere den Indegree von w um 1.
 4. Entstehen durch 3 Knoten mit $\text{inDeg}(v) == 0$, füge diese in Z ein
4. Falls alle Knoten verarbeitet: Zyklensfrei (topologisch sortiert), sonst G hat Zyklen

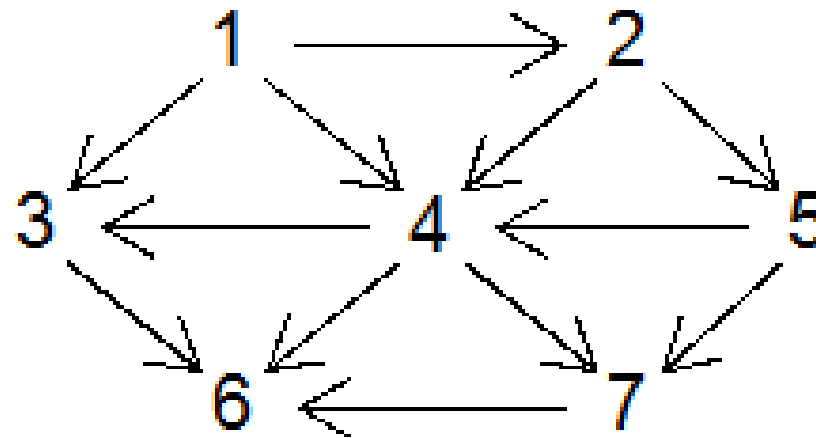
Topologisches Sortieren - Aufgabe

- Finden Sie zwei topologische Sortierungen für den Graphen im Script auf Seite 3 oben:



Topologisches Sortieren - Aufgabe

- Finden Sie zwei topologische Sortierungen für den Graphen im Script auf Seite 3:



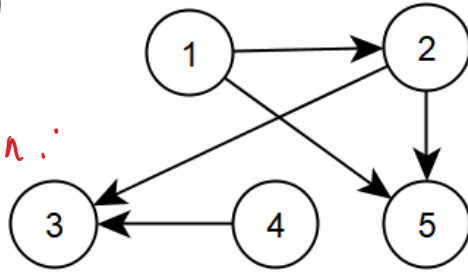
Topologische Sortierungen:

1 2 5 4 3 7 6

1 2 5 4 7 3 6

n|w

Gegeben:



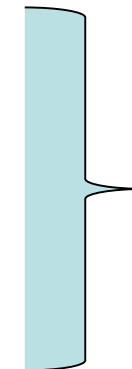
Adjazenz-Liste:

<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>
↓	↓		↓	
2	3		3	
↓	↓			
5	5			



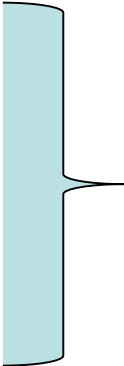

1.) Eingangsgrade (Map):
1:
2:
3:
...

Worst-Case Komplexitätsanalyse: Topologisches Sortieren mit einer Adjazenzliste. Verwenden Sie: n = Anz. Knoten, m = Anz. Kanten

1. Eingangsgrad für alle Knoten bestimmen
2. Alle Knoten mit Indegree 0 in Set Z sammeln
3. Solange Z nicht leer
 1. Knoten v aus Z entnehmen
 2. Indegree adjazenter Knoten um 1 reduzieren
 3. Falls ein Knoten einen Indegree 0 bekommt, füge diesen in Z ein
4. Falls alle Knoten bearbeiten liegt eine top. Sortierung vor, ansonsten enthält der Graph Zyklen.



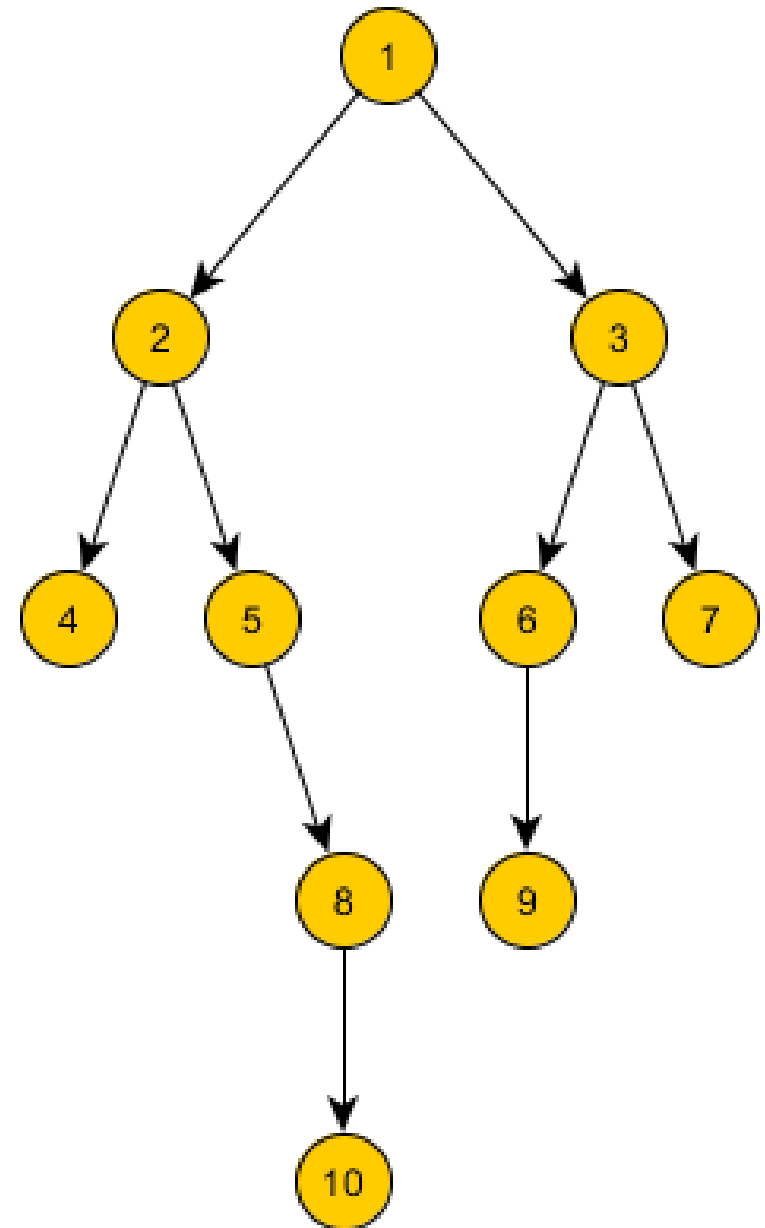
Worst-Case Komplexitätsanalyse: Topologisches Sortieren (Adjazenzliste)

1. Eingangsgrad für alle Knoten bestimmen  $O(n+m)$
2. Alle Knoten mit Indegree 0 in Set sammeln  $O(n)$
3. Solange Z (Queue) nicht leer
 1. Knoten v aus Z entnehmen
 2. Indegree adjazenter Knoten um 1 reduzieren
 3. Falls ein Knoten einen Indegree 0 bekommt, füge diesen in Z ein $O(n+m)$
4. Falls alle Knoten bearbeiten liegt eine top. Sortierung vor, ansonsten enthält der Graph Zyklen.  $O(1)$

=> $O(n+m)$ für topologische Sortierung / Prüfen von Graphen auf Zyklen

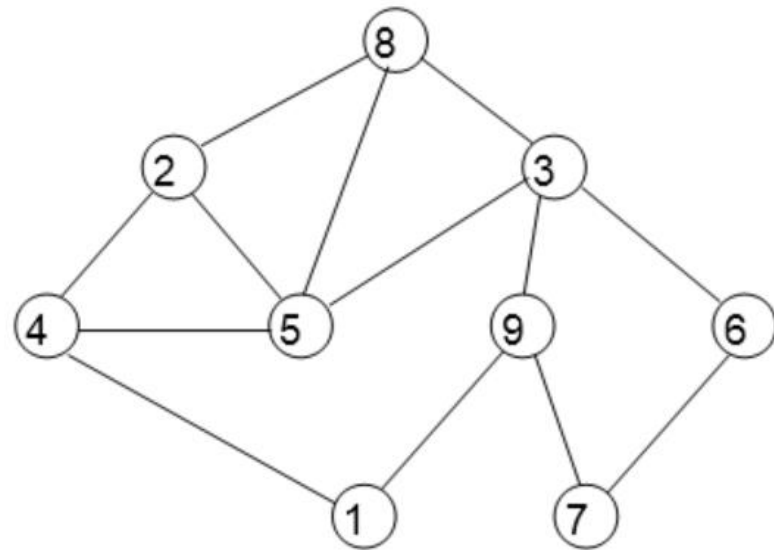
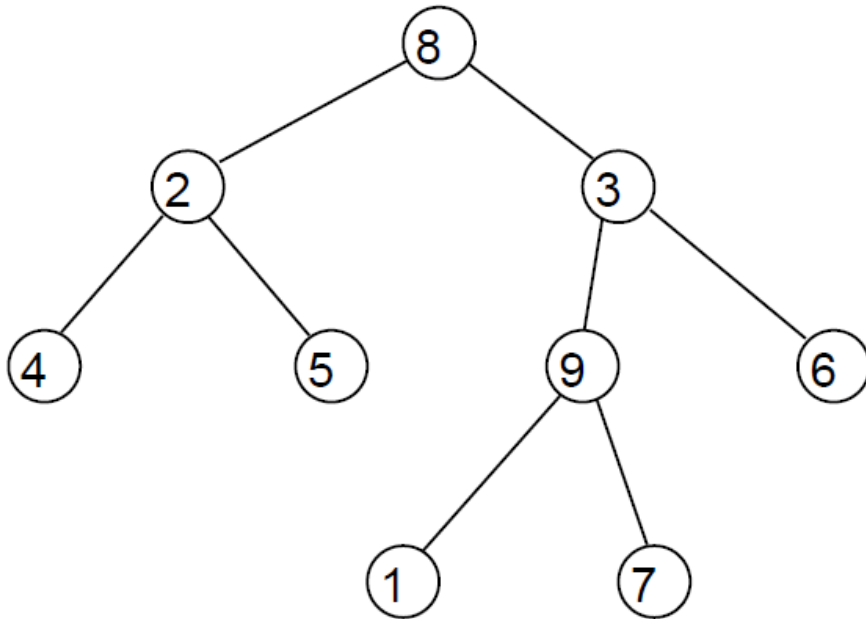
Graph Traversierungen

- Alle Knoten durchlaufen
- **Tiefensuche** (Pre-Order Traversierung)
 - Weite Knoten möglichst rasch erreichen
- **Breitensuche**
 - Naheliegende Knoten möglichst rasch erreichen



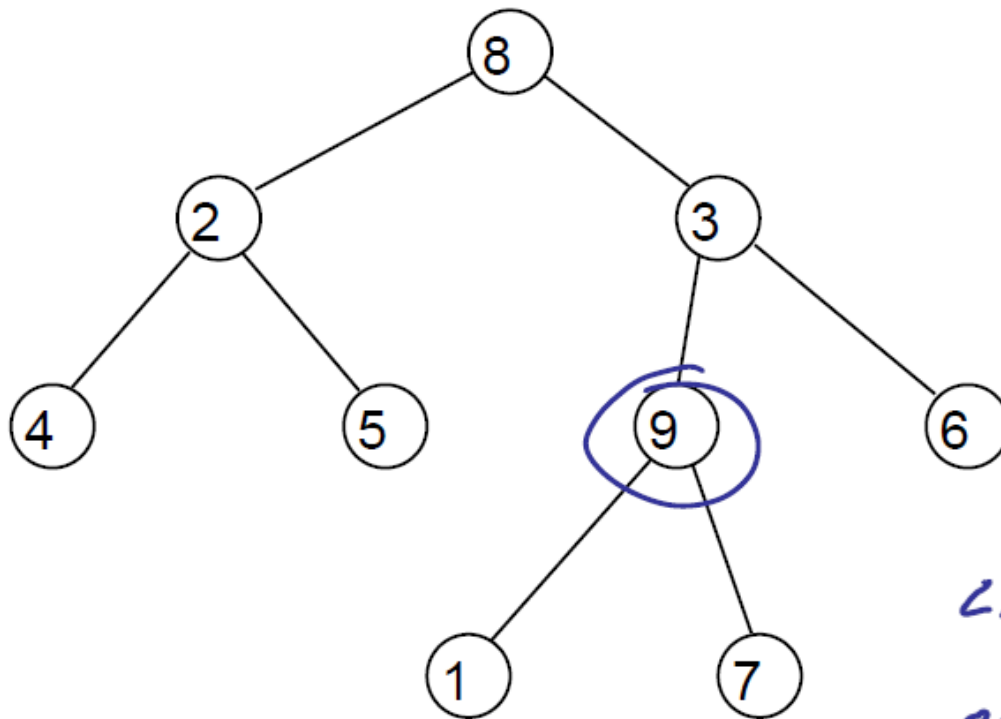
Depth-First-Search (Tiefensuche) (Kap. 7.6)

- Arbeitsblatt DFS
 - Erkenntnisse der Aufgabe 2 im Abschnitt 7.6.1 anfügen



Depth-First-Search (Tiefensuche) (Kap. 7.6)

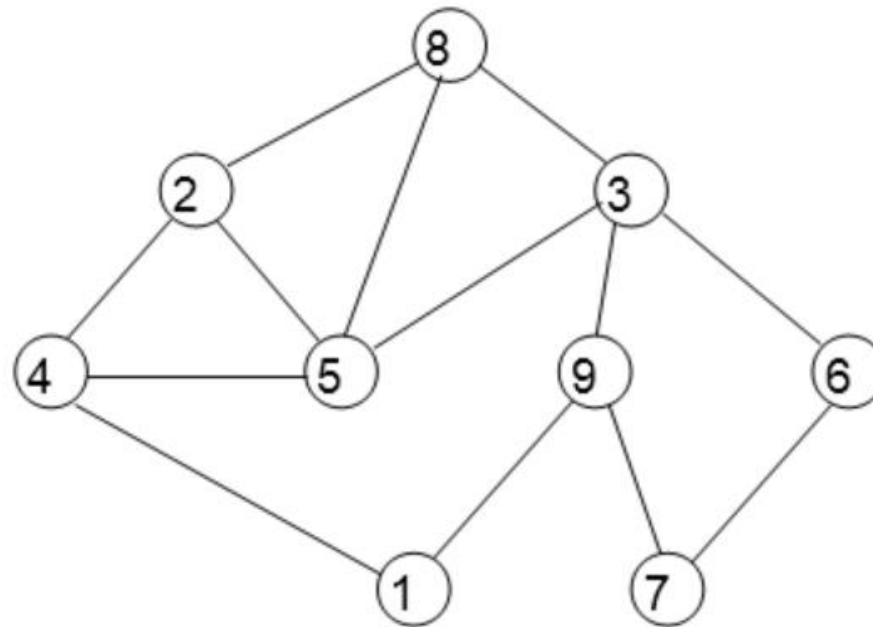
- Lösung: Arbeitsblatt DFS - Aufgabe 1



z.B.: 9, 1, 7, 3, 8, 2, 4, 5, 6
od.: 9, 3, 8, 2, 4, 5, 6, 1, 7

Depth-First-Search (Tiefensuche) (Kap. 7.6)

- Lösung: Arbeitsblatt DFS - Aufgabe 2



Depth-First-Search (Tiefensuche) : Arbeitsblatt Erkenntnisse

- Im Gegensatz zu Bäumen muss bei allgemeinen Graphen mit folgender Situation umgegangen werden:
 - Man kann auf mehreren Wegen zum selben Knoten gelangen
 - Bei Zyklen will man nicht ewig im Kreis laufen
- Dazu drängt sich die folgende Lösung auf:
 - Besuchte Knoten «maskieren»: `v.visited=true;`
 - Markierte Knoten nicht nochmals besuchen `if (!v.visited) {...}`

Depth-First-Search (Tiefensuche) : Algorithmus

Initialisierung:

1. Alle Knoten als nicht besucht markieren
2. *dfs(startKnoten)* aufrufen

(Unterstrichene Teile = Neu gegenüber Baum-Traversierung)

dfs (Vertex v):

falls v als nicht besucht markiert ist

1. v «besuchen» (verarbeiten, ausgeben usw.)
2. v als besucht markieren

3. Für alle Knoten w, welche adjazent zu v sind: *dfs(w)* aufrufen

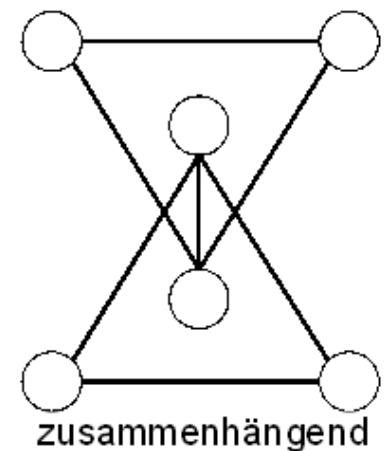
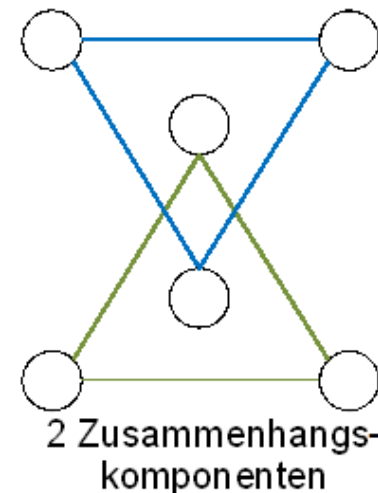
```
if (!v.visited) {  
    print(v);  
    v.visited = true;  
    for (Vertex w : v.adjList)  
        dfs(w);  
}
```

DFS-Anwendung: Graph auf Zusammenhang prüfen

Definition: Ein ungerichteter Graph ist zusammenhängend, falls es für jedes Paar von verschiedenen Knoten einen verbindenden Pfad gibt.

Algorithmus:

1. Alle Knoten als bisher **nicht** besucht markieren
2. Dfs(v) für beliebigen Knoten aufrufen
3. Prüfen, ob alle Knoten markiert wurden



DFS-Anwendung: Spannbaum

- Baum, der vom Startknoten alle erreichbaren Knoten enthält
- Jeder Baum mit n Knoten enthält genau $(n-1)$ Kanten

Algorithmus, um während DFS den Spannbaum zu speichern:

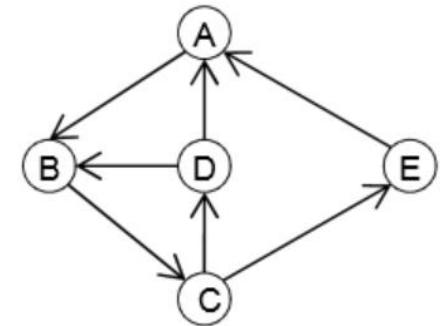
dfs (Vertex v):

falls v als nicht besucht markiert ist:

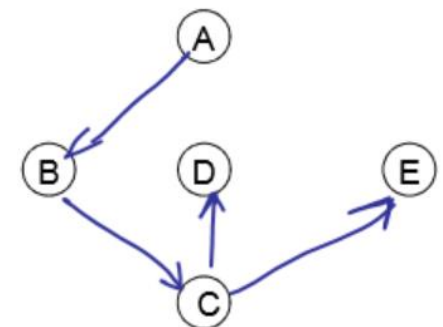
1. v als *besucht* markieren
2. Für alle Knoten w adjazent zu v

Falls w als *nicht besucht* markiert ist:

- a) Kante $\langle v, w \rangle$ dem Graphen *tree* hinzufügen
- b) dfs(w) aufrufen



dfs('A') führt zu folgendem Spannbaum:



Kürzeste Wege: Definition

- **Ungewichtete Graphen:** Anzahl Kanten von Start zu Ziel
- **Gewichtete Graphen:** Summe der Kantengewichte aller Kanten von Start zu Ziel

Kürzeste Wege in Ungewichteten Graphen: Breadth-First-Search (BFS)

- BFS: Man geht zuerst in die Breite und markiert Knoten mit Distanz 1, Distanz 2 usw.

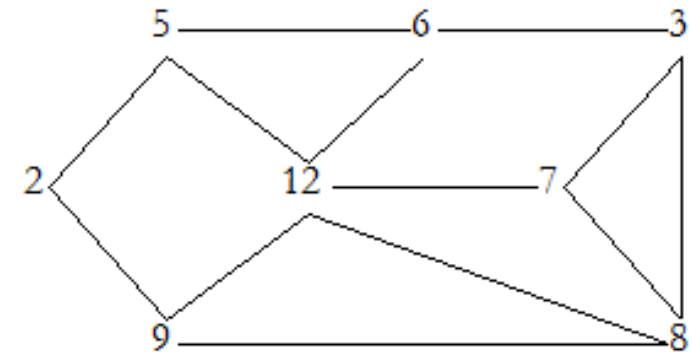
Algorithmus:

1. Bei allen Knoten Distanzangabe auf ∞ setzen
2. Beim Startknoten 2 die Distanzangabe auf 0 setzen
3. Startknoten in Warteschlange *erreichbar* anfügen
4. Solange Warteschlange *erreichbar* nicht leer ist:
 - a. Vordersten Knoten v aus der Warteschlange entnehmen
 - b. Für alle zu v adjazenten Knoten w :

Falls Distanzangabe von $w = \infty$ (Bisher unbesucht)

Distanzangabe von $w = 1 + \text{Distanzangabe von } v$

Knoten w hinten an Warteschlange *erreichbar* anfügen



Kürzeste Wege in gewichteten Graphen: Algorithmus nach Dijkstra

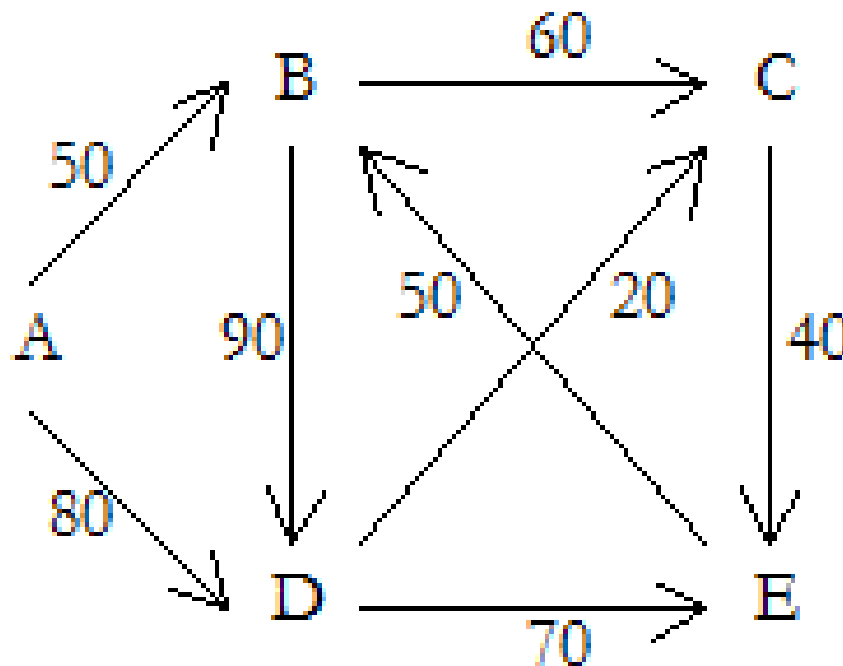
- Findet kürzeste Wege ausgehend von einem Knoten zu allen anderen
- Kein schnellerer Algorithmus für dieses Problem bekannt

- Annahmen
 - Keine Doppelten Kanten
 - Keine Schlingen
 - Nur positive Kantengewichte

Kürzeste Wege in gewichteten Graphen: Algorithmus nach Dijkstra

- Aufgabe
 - Dijkstra-Algorithmus Arbeitsblatt & Script
 - Aufwandsanalyse
 - Selbststudium

**Kürzester Weg
Von A nach E:**



Dijkstra Aufwandsanalyse

Schritt 1: Tabelle für alle Knoten initialisieren: $O(n)$

Schritt 2a: n Knoten verarbeiten, jeweils n Knoten durchsuchen: $O(n^2)$

Schritt 2c: Allen Kanten genau einmal folgen: $O(m)$

Insgesamt also: $O(n^2 + m)$

Verbesserung durch geschickte Wahl der Hilfsstruktur für 2a:

$O((m + n) \log n)$

Selbststudium

IKEA TopSort Aufgabe d.)

Siehe Netzlaufwerk