

# Informatik II - SS 2016

## (Algorithmen & Datenstrukturen)

Vorlesung 14 (10.6.2016)

### Graphenalgorithmen II



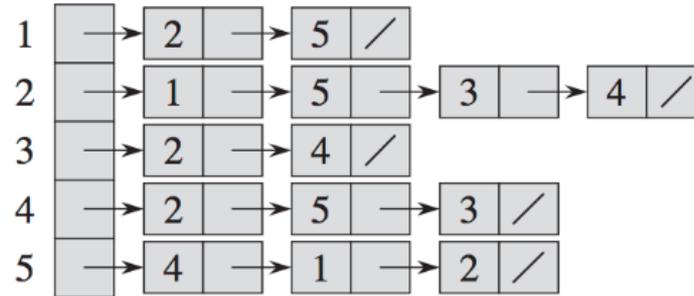
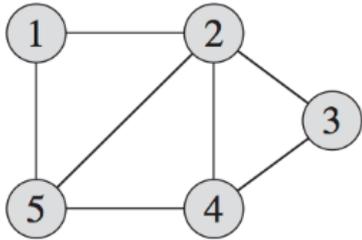
**UNI  
FREIBURG**

Fabian Kuhn

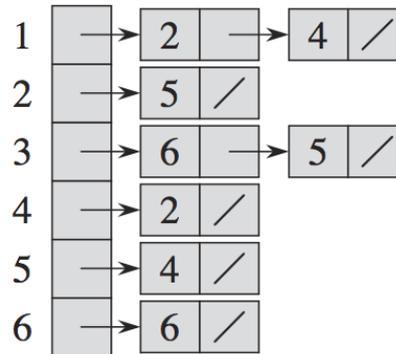
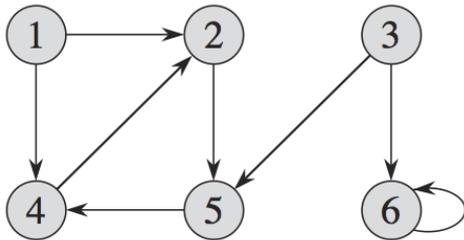
Algorithmen und Komplexität

# Repräsentation von Graphen

Beispiele aus [CLRS]:



|   | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 1 |
| 2 | 1 | 0 | 1 | 1 | 1 |
| 3 | 0 | 1 | 0 | 1 | 0 |
| 4 | 0 | 1 | 1 | 0 | 1 |
| 5 | 1 | 1 | 0 | 1 | 0 |



|   | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 0 | 1 | 1 |
| 4 | 0 | 1 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 1 |

## Graph-Traversierung (Graph-Exploration) informell

- Gegeben ein Graph  $G = (V, E)$  und ein Knoten  $s \in V$ , besuche “systematisch” alle Knoten, welche von  $s$  aus erreichbar sind.
- Das haben wir bereits bei den Binärbäumen gesehen
- Wie bei den Bäumen gibt es zwei grundsätzliche Verfahren
- **Breitensuche (BFS = breadth first search)**
  - zuerst “in die Breite” (nähere Knoten zu  $s$  zuerst)
- **Tiefensuche (DFS = depth first search)**
  - zuerst “in die Tiefe” (besuche alles, was an einem Knoten  $u$  “dranhängt”, bevor der nächste Nachbar von  $u$  besucht wird)
- Graph-Traversierung ist wichtig, da es oft als Subroutine auftaucht
  - z.B., um die Zusammenhangskomponenten eines Graphen zu finden
  - Wir werden einige Beispiele sehen...

- Wir merken uns zusätzlich die Distanz zu  $s$  im Baum

BFS-Tree( $s$ ):

```
Q = new Queue();
for all u in V: u.marked = false;
s.marked = true;
s.parent = NULL;
s.d = 0
Q.enqueue(s)
while not Q.empty() do
    u = Q.dequeue()
    visit(u)
    for v in u.neighbors do
        if not v.marked then
            v.marked = true;
            v.parent = u;
            v.d = u.d + 1;
            Q.enqueue(v)
```

In der Folge benennen wir die Knoten folgendermaßen

- weiße Knoten: Knoten, welche der Alg. noch nicht gesehen hat
- graue Knoten: markierte Knoten
  - Knoten werden grau, wenn sie in die Warteschlange eingefügt werden
  - Knoten sind grau, solange sie in der Warteschlange sind
- schwarze Knoten: besuchte Knoten
  - Knoten werden schwarz, wenn sie aus der Warteschlange genommen werden

Im BFS-Baum eines ungewichteten Graphen ist die Distanz von jedem Knoten  $u$  zur Wurzel  $s$  gleich  $d_G(s, u)$ .

- Baumdistanz zur Wurzel:  $d_T(s, u) = u.d$
- Wir müssen also zeigen, dass  $u.d = d_G(s, u)$
- Wir zeigen zuerst, dass  $u.d \geq d_G(s, u)$

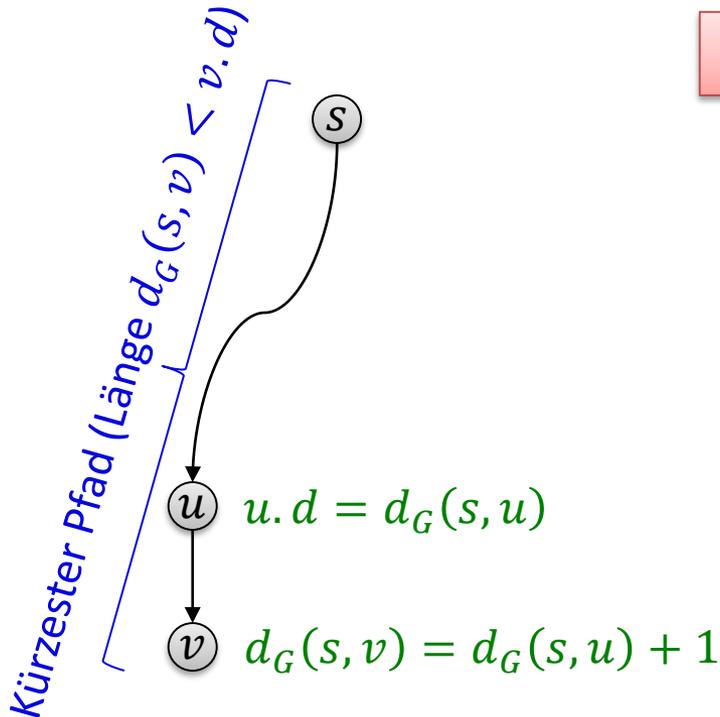
**Lemma:** Annahme: Während BFS-Traversal ist Zustand der Queue

$$Q = \langle v_1, v_2, \dots, v_r \rangle \quad (v_1: \text{head}, v_r: \text{tail})$$

Dann gilt  $v_r.d \leq v_1.d + 1$  und  $v_i.d \leq v_{i+1}.d$  (für  $i = 1, \dots, r - 1$ )

Im BFS-Baum eines ungewichteten Graphen ist die Distanz von jedem Knoten  $u$  zur Wurzel  $s$  gleich  $d_G(s, u)$ .

- Widerspruchsbeweis:
  - Annahme:  $v$  ist Knoten mit kleinstem  $d_G(s, v)$ , für welchen  $v.d > d_G(s, v)$



$$v.d > d_G(s, v) = d_G(s, u) + 1 = u.d + 1$$

**Betrachte dequeue von  $u$ :**

- $v$  wird als ein Nachbar von  $u$  betrachtet
- $v$  ist weiss  $\Rightarrow v.d = u.d + 1$
- $v$  ist schwarz  $\Rightarrow v.d \leq u.d$
- $v$  is grau  $\Rightarrow v$  ist in der Warteschlange  
Lemma:  $v.d \leq u.d + 1$

## Grundidee Tiefensuche in $G$ (Start bei Knoten $s \in V$ )

- **Markiere Knoten  $v$**  (am Anfang ist  $v = s$ )
- Besuche die Nachbarn von  $v$  der Reihe nach *rekursiv*
- Nachdem alle Nachbarn besucht sind, **besuche  $s$**
- **rekursiv:** Beim Besuchen der Nachbarn werden deren Nachbarn besucht, und dabei deren Nachbarn, etc.
- **Zyklen in  $G$ :** Besuche jeweils nur Knoten, welche noch nicht markiert sind
- entspricht der Postorder-Traversierung in Bäumen
- Fall man gleich beim Markieren den Knoten besucht, entspricht es der Preorder-Traversierung

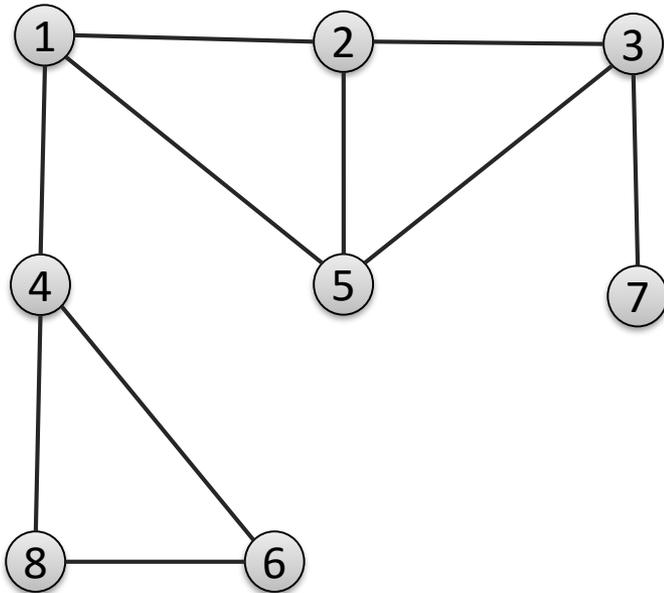
DFS-Traversal(s):

```
for all u in V: u.color = white;  
DFS-visit(s, NULL)
```

DFS-visit(u, p):

```
u.color = gray;  
u.parent = p;  
for all v in u.neighbors do  
    if v.color = white  
        DFS-visit(v, u)  
visit node u;  
u.color = black;
```

# Tiefensuche: Beispiel



In der gleichen Art wie bei der Breitensuche, kann man auch bei der Tiefensuche einen Spannbaum konstruieren

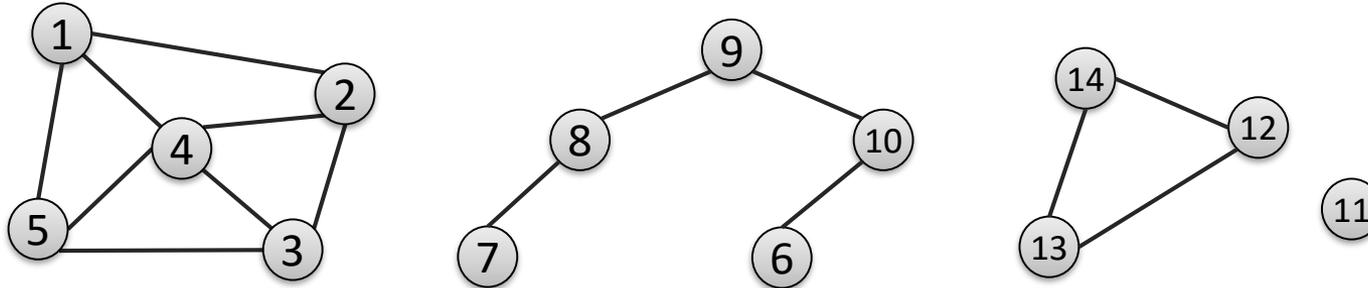
- Die Eigenschaften dieses DFS-Baums werden wir noch anschauen

**Die Laufzeit der Tiefensuche (DFS-Traversierung) ist  $O(n + m)$ .**

- Wir färben die Knoten weiß, grau und schwarz wie vorher
  - nicht markiert = weiß, markiert = grau, besucht = schwarz

# Zusammenhangskomponenten

- Die Zusammenhangskomponenten (oder einfach Komponenten) eines Graphen sind seine zusammenhängenden Teile.



Ziel: Finde alle Komponenten eines Graphen.

```
for u in V do
```

```
    if not u.marked then
```

```
        start new component
```

```
        explore with DFS/BFS starting at u
```

- Die Zusammenhangskomponenten eines Graphen können in  $O(n + m)$  Zeit identifiziert werden. (mit Hilfe von DFS oder BFS)

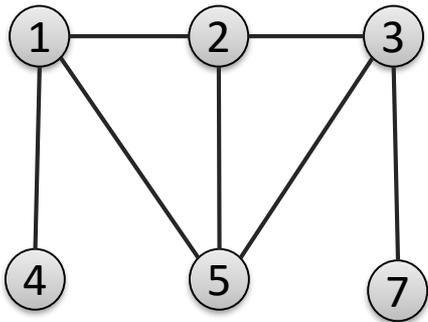
# DFS- “Klammer”-Theorem

Wir definieren für jeden Knoten  $v$  die folgenden zwei Zeitpunkte

- $t_{v,1}$ : Zeitpunkt, wenn  $v$  in der DFS-Suche grau gefärbt wird
- $t_{v,2}$ : Zeitpunkt, wenn  $v$  in der DFS-Suche schwarz gefärbt wird

**Theorem:** Im DFS-Baum ist ein Knoten  $v$  genau dann im Teilbaum eines Knoten  $u$ , falls das Intervall  $[t_{v,1}, t_{v,2}]$  vollständig im Intervall  $[t_{u,1}, t_{u,2}]$  enthalten ist. Zudem sind zwei Intervalle entweder disjunkt, oder das eine ist komplett im anderen enthalten.

**Beispiel:**



# DFS- “Klammer”-Theorem

**Theorem:** Im DFS-Baum ist ein Knoten  $v$  ein Teilbaum eines Knoten  $u$ , falls das Intervall  $[t_{v,1}, t_{v,2}]$  vollständig im Intervall  $[t_{u,1}, t_{u,2}]$  enthalten ist. Zudem sind zwei Intervalle entweder disjunkt, oder das eine ist komplett im anderen enthalten.

**Beweis:**

# DFS- “Klammer”-Theorem

**Theorem:** Im DFS-Baum ist ein Knoten  $v$  genau dann im Teilbaum eines Knoten  $u$ , falls das Intervall  $[t_{v,1}, t_{v,2}]$  vollständig im Intervall  $[t_{u,1}, t_{u,2}]$  enthalten ist. Zudem sind zwei Intervalle entweder disjunkt, oder das eine ist komplett im anderen enthalten.

**Beweis:**

**Theorem:** Im DFS-Baum ist ein Knoten  $v$  genau dann im Teilbaum eines Knoten  $u$ , falls das Intervall  $[t_{v,1}, t_{v,2}]$  vollständig im Intervall  $[t_{u,1}, t_{u,2}]$  enthalten ist.

## Implikationen

- Zwei Intervalle sind entweder disjunkt, oder das eine ist komplett im anderen enthalten.
- Ein weisser Knoten  $v$ , welcher in der rekursiven Suche von  $u$  entdeckt wird, wird schwarz, bevor die Rekursion zu  $u$  zurückkehrt.
- Wieso “Klammer”-Theorem:  
Wenn man bei jedem  $t_{v,1}$  eine öffnende Klammer und bei jedem  $t_{v,2}$  eine schließende Klammer hinschreibt, bekommt man ein Klammersausdruck, welcher korrekt geschachtelt ist.

**Theorem:** In einem DFS-Baum ist ein Knoten  $v$  genau dann im Teilbaum eines Knoten  $u$ , falls unmittelbar vor dem Markieren von  $u$ , ein komplett weißer Pfad von  $u$  nach  $v$  besteht.

**Beweis:**

# Klassifizierung der Kanten (bei DFS-Suche)

## Baumkanten:

- $(u, v)$  ist eine Baumkante, falls  $v$  von  $u$  aus entdeckt wird

## Rückwärtskanten:

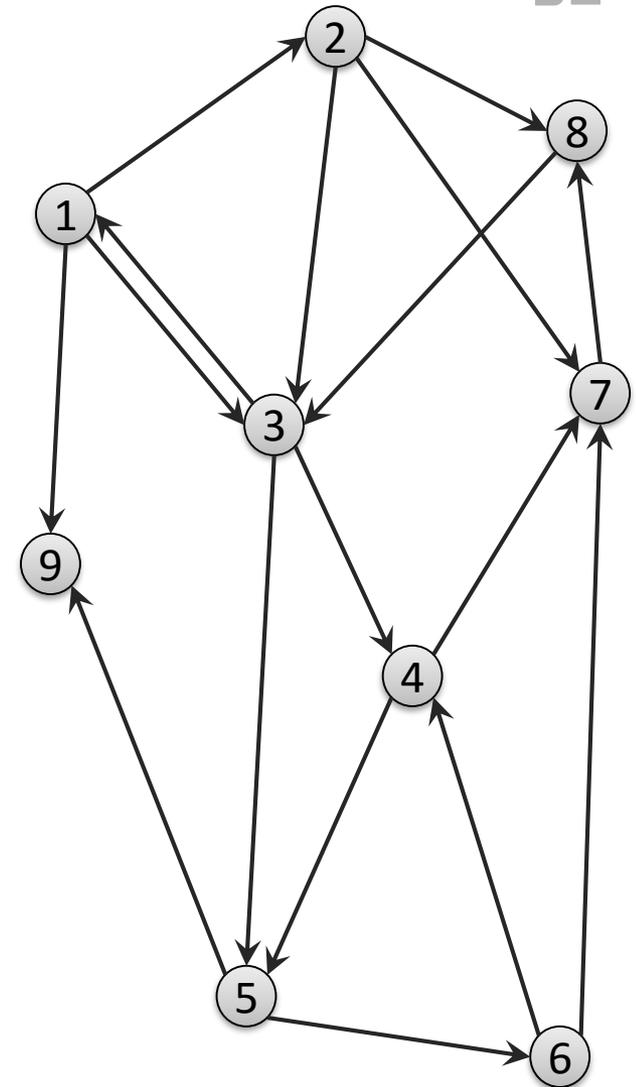
- $(u, v)$  ist eine Rückwärtskante, falls  $v$  ein Vorgängerknoten von  $u$  ist

## Vorwärtskanten:

- $(u, v)$  ist eine Vorwärtskante, falls  $v$  ein Nachfolgerknoten von  $u$  ist

## Queranten:

- Alle übrigen Kanten



**Baumkante  $(u, v)$ :**

**Rückwärtskante  $(u, v)$ :**

**Vorwärtskante  $(u, v)$ :**

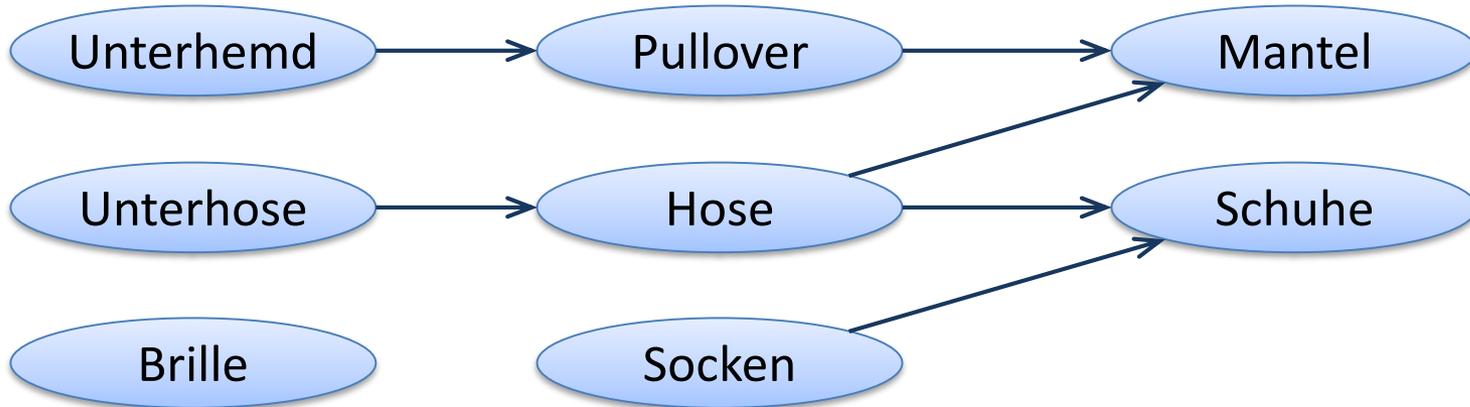
**Querkante  $(u, v)$ :**

**Theorem:** Bei einer DFS-Suche in ungerichteten Graphen ist jede Kante entweder eine Baumkante oder eine Rückwärtskante.

**Theorem:** Ein gerichteter Graph hat genau dann keine Zyklen, falls es bei der DFS-Suche keine Rückwärtskanten gibt.

## Zyklenfreie, gerichtete Graphen:

- **DAG**: directed acyclic graph
- Modellieren z.B. zeitliche Abhängigkeiten von Aufgaben
- Beispiel: Anziehen von Kleidungsstücken



## Topologische Sortierung:

- Sortiere die Knoten eines DAGs so, dass  $u$  vor  $v$  erscheint, falls ein gerichteter Pfad von  $u$  nach  $v$  existiert
- Im Beispiel: Finde eine mögliche Anziehreihenfolge

## Zyklenfreie, gerichtete Graphen:

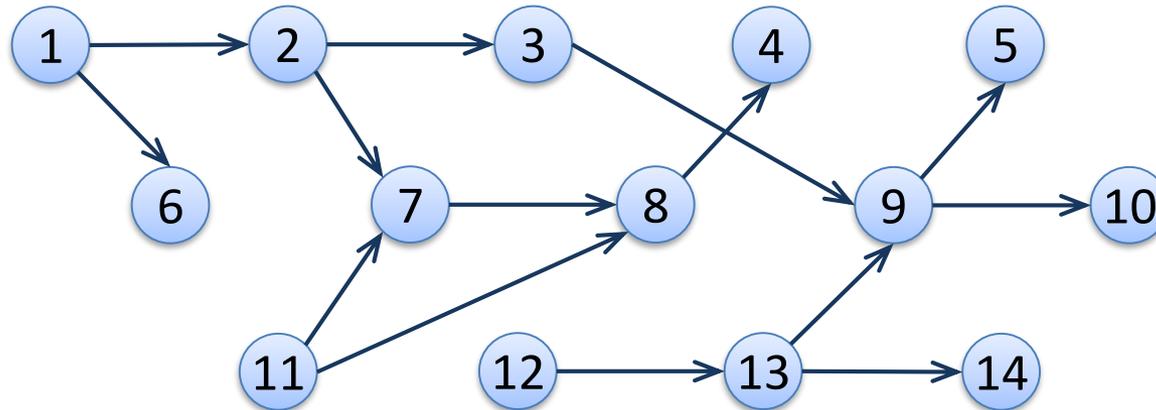
- repräsentieren partielle Ordnungsrelationen
  - asymmetrisch:  $a < b \Rightarrow \neg(b < a)$
  - transitiv:  $a < b \wedge b < c \Rightarrow a < c$
  - partielle Ordnung: nicht alle Paare müssen vergleichbar sein
- Beispiel: Teilmengenrelation bei Mengen

## Topologische Sortierung:

- Sortiere die Knoten eines DAGs so, dass  $u$  vor  $v$  erscheint, falls ein gerichteter Pfad von  $u$  nach  $v$  existiert
- Erweitere eine partielle Ordnung zu einer totalen Ordnung

# Topologische Sortierung: Algorithmus

Führe DFS aus...



## Beobachtung:

- Knoten ohne Nachfolger werden als erstes besucht (schwarz gef.)
- Besuchreihenfolge ist umgekehrte topologische Sortierung

# Topologische Sortierung: Algorithmus

---

**Theorem:** Umgekehrte “Visit”-Reihenfolge (schwarz färben) der Knoten bei DFS-Traversierung ergibt topologische Sortierung