

Informatik II - SS 2018

(Algorithmen & Datenstrukturen)

Vorlesung 13 (6.6.2018)

Graphenalgorithmen II



**UNI
FREIBURG**

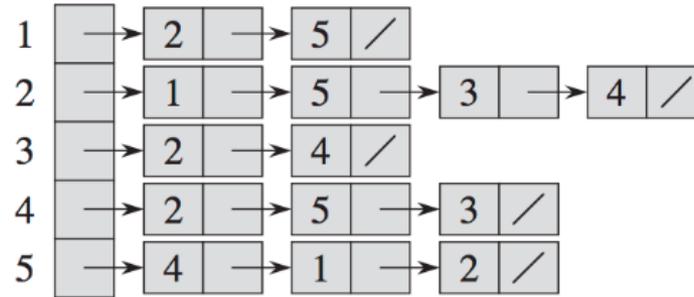
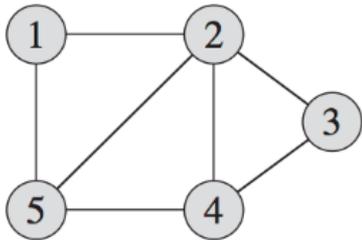
Yannic Maus

Algorithmen und Komplexität

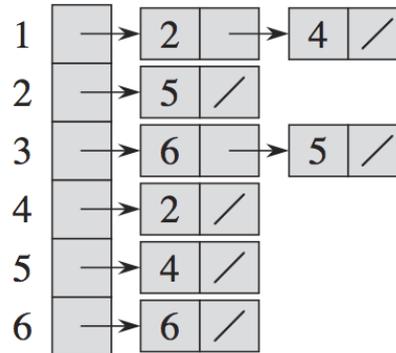
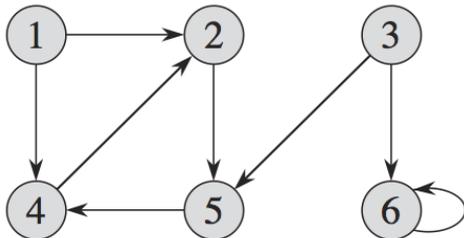
Repräsentation von Graphen

Zwei klassische Arten, einen Graphen im Rechner zu repräsentieren

- **Adjazenzmatrix:** Platzverbrauch $O(|V|^2) = O(n^2)$
- **Adjazenzlisten:** Platzverbrauch $O(|V| + |E|) = O(n + m)$



	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

Hauptunterschied Binärbaum $T \Leftrightarrow$ allg. Graph G

- Graph G kann Zyklen haben

Breitensuche in Graph G (Start bei Knoten $s \in V$)

- **Zyklen:** markiere Knoten, welche man schon gesehen hat
- **Markiere** Knoten s , hänge s in die Queue
- Wie bisher, nehme immer den ersten Knoten u aus der Queue:
 - **besuche Knoten u**
 - Gehe durch die Nachbarn v von u
Falls v nicht markiert, markiere v und hänge v in Queue
Falls v markiert ist, muss nichts getan werden

- 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)
```

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

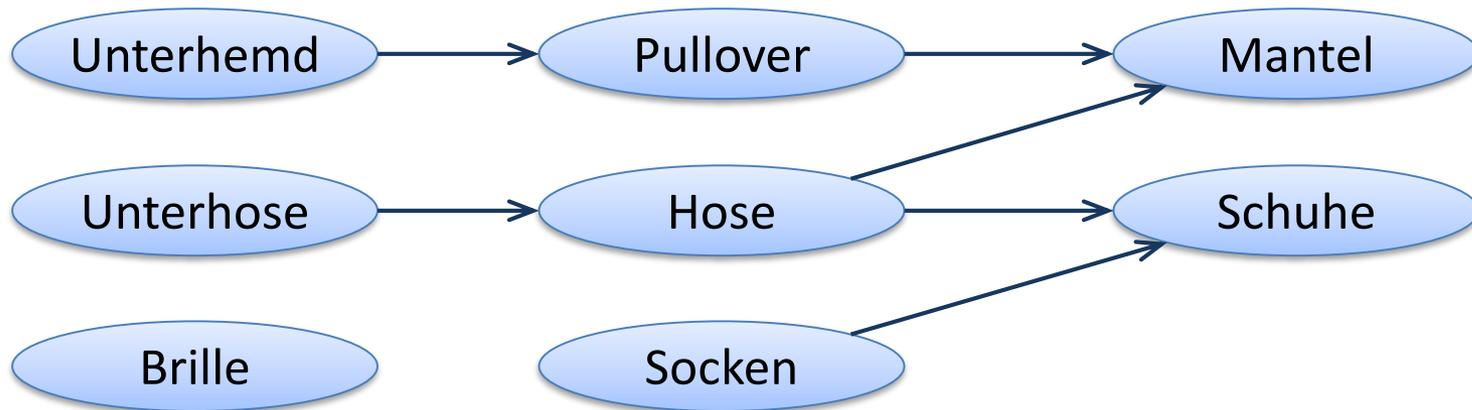
- Die ungewichteten Abstände zwischen allen Knotenpaaren lassen sich in Zeit $O(|V| \cdot (|V| + |E|))$ berechnen
- Jeder Graph lässt sich in Zeit $O(|V| + |E|)$ auf Zusammenhang testen
- Die Anzahl der Zusammenhangskomponenten lässt sich in Zeit $O(|V| + |E|)$ berechnen
- Ein ungerichteter Graph lässt sich in Zeit $O(|V| + |E|)$ auf Kreisfreiheit testen (dies geht auch in $O(|V|)$, Übung?)

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

Warum schauen wir DFS so intensiv an?

- Wird sehr häufig als Subroutine verwendet
- Impliziert einen (einfachen) Algorithmus um **Zykel (Kreise)** in **gerichteten** Graphen zu finden
- Ermöglicht einen Algorithmus zum **topologischen Sortieren**.



Ausgabe: Möglich Anziehreihenfolge, wichtig beim Kompilieren

- Viele weitere Anwendungen: Zusammenhangskomponenten finden, Artikulationsknoten finden, Bidirected Components finden, usw.

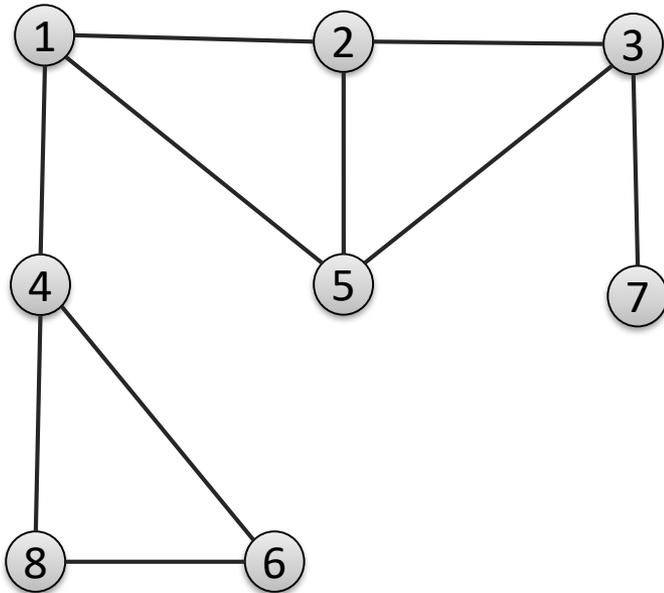
DFS-Traversal(s):

```
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

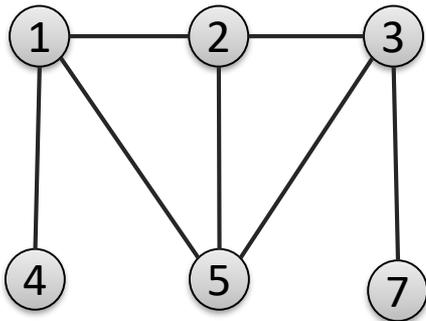
DFS- “Klammer”-Theorem

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.

Beispiel:



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.

Implikationen

- Zwei Intervalle sind entweder disjunkt, oder das eine ist komplett im anderen enthalten.
- 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 Klammersymbol, welches korrekt geschachtelt ist.
- Ein weisser Knoten v , welcher in der rekursiven Suche von u entdeckt wird, wird schwarz, bevor die Rekursion zu u zurückkehrt.

DFS- “Klammer”-Theorem: Warum?

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.

Wieso ist dies nützlich?

- Erhöht das Verständnis für den resultierenden DFS-Baum
- Wir benötigen das Theorem z.B. bei der Korrektheit des Algorithmus' zur topologischen Sortierung

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.

- Insbesondere sind zwei Intervalle entweder disjunkt, oder das eine ist komplett im anderen enthalten.

Beweis:

- graue Knoten bilden immer einen Pfad (G zusammenhängend)

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.

- Insbesondere sind zwei Intervalle entweder disjunkt, oder das eine ist komplett im anderen enthalten.

Beweis:

- graue Knoten bilden immer einen Pfad (G zusammenhängend)

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:

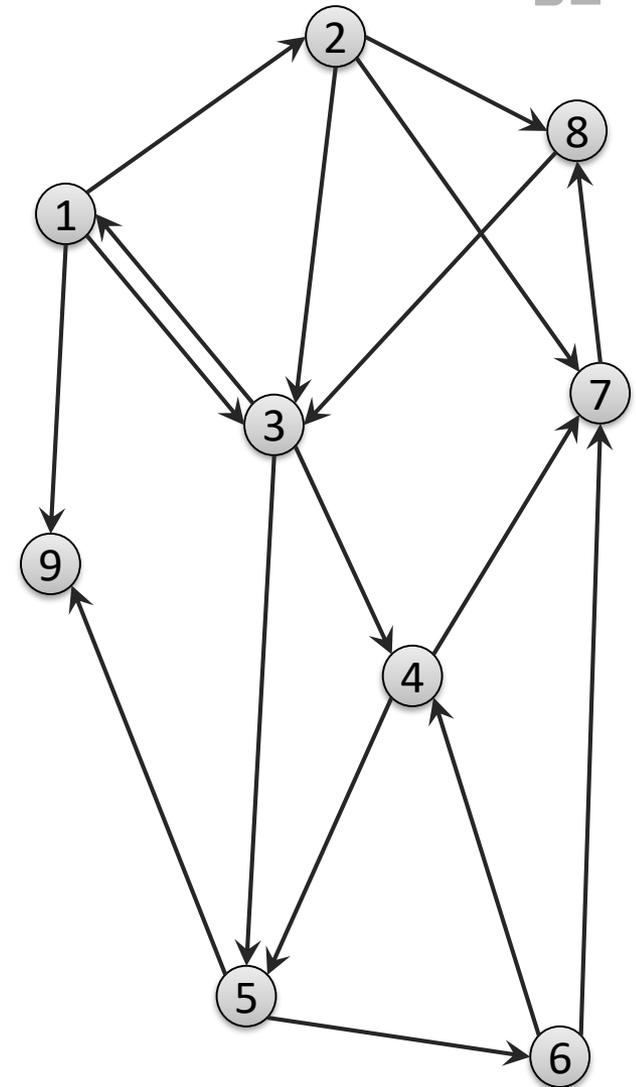
- (u, v) ist eine Rückwärtskante, falls v ein Vorgängerknoten von u ist

Vorwärtskanten:

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

Queranten:

- Alle übrigen Kanten



Baumkante (u, v) :

Rückwärtskante (u, v) :

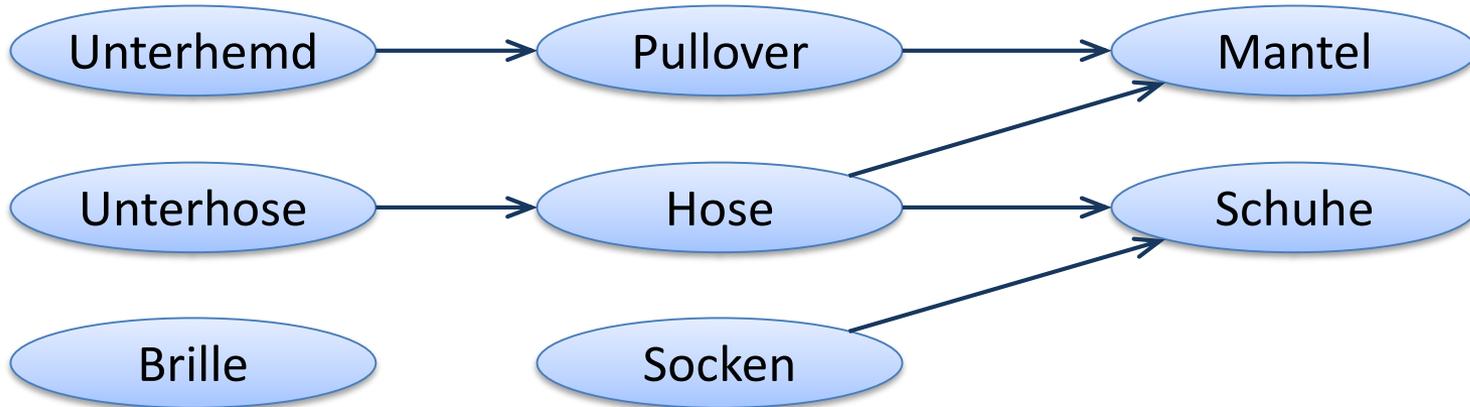
Vorwärtskante (u, v) :

Querkante (u, v) :

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

Zyklusfreie, 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:

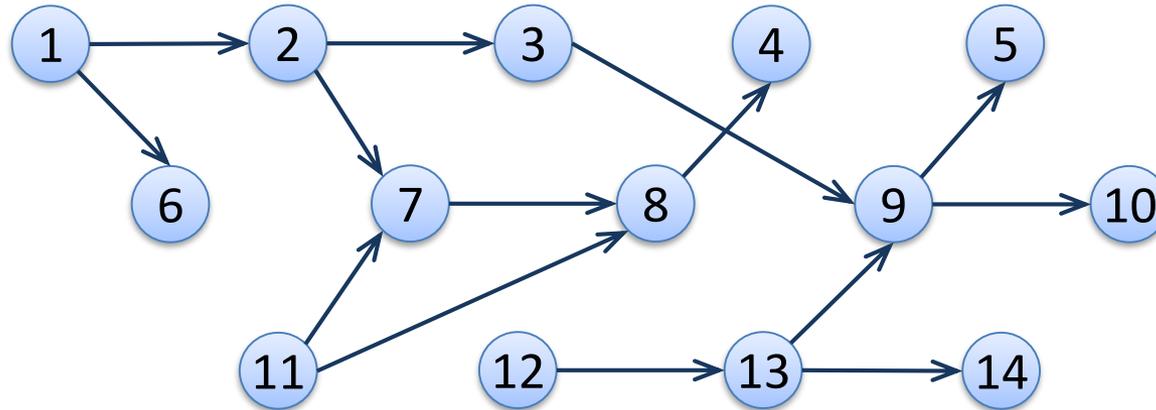
- repräsentieren partielle Ordnungsrelationen
 - asymmetrisch: $a < b \Rightarrow \neg(b < a)$
 - transitiv: $a < b \wedge b < c \Rightarrow a < c$
 - partielle Ordnung: nicht alle Paare müssen vergleichbar sein
- Beispiel: Teilmengenrelation bei Mengen

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

Topologische Sortierung: Algorithmus

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

Stark zusammenhängende Komponenten

- Stark zus.-hängende Komponente eines gerichteten Graphen:
“Maximale Knoten-Teilmenge, so dass jeder jeden erreichen kann”

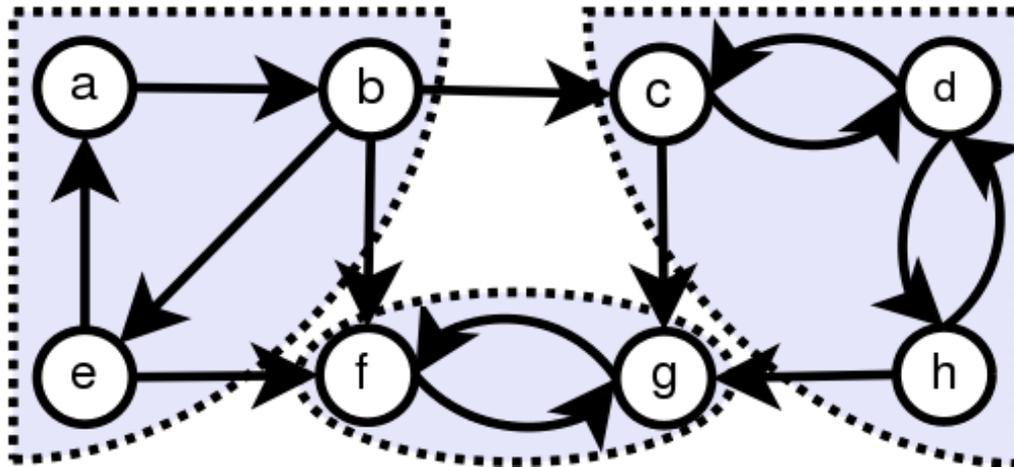


Bild: Wikipedia

- Benötigt 2 DFS-Traversierungen (Zeit $\in O(m + n)$)
 - auf G und auf G^T (alle Kanten umgedreht)
 - G und G^T haben die gleichen stark zus.-hängenden Komponenten
- Details z.B. in [CLRS]

Artikulationsknoten, Brücken, Biconnected Components

- Annahme: ungerichteter Graph
- **Artikulationsknoten v :**
 v entfernen vergrößert die Anzahl Komponenten

- **Brücke e :**
Kante e entfernen vergrößert die Anzahl Komponenten

Biconnected Components

- Komponenten, welche übrig bleiben, wenn man alle Brücken entfernt

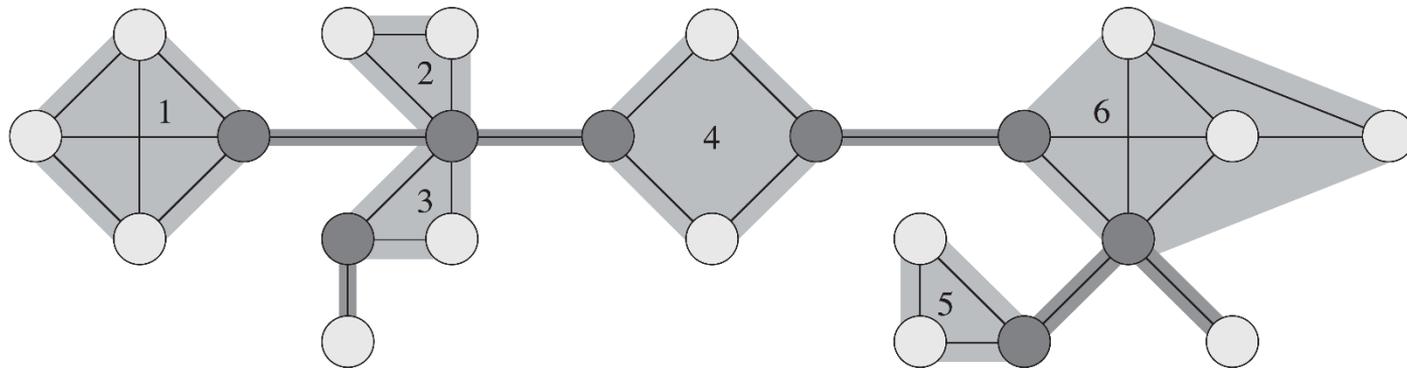


Bild: [CLRS]

- Artikulationsknoten und Brücken können mit einer DFS-Traversierung in $O(m + n)$ Zeit gefunden werden
 - Algorithmus von Hopcroft, Tarjan (1973)
- Zerlegung in Biconnected Components daher in der gleichen Zeit

- Als ungerichtete Graphen (mit n Knoten) betrachtet...

Baum:

- Zusammenhängender ungerichteter Graph, ohne Zyklen
 - Ein nicht zus.-hängender zyklener (unger.) Graph heisst Wald
 - Anzahl Kanten: $n - 1$ (jede Kante reduziert die #Komponenten um 1)

Äquivalente Definitionen:

- Minimaler zusammenhängender Graph
- Maximaler zyklener Graph
- Eindeutiger Pfad zwischen jedem Knotenpaar
- Zusammenhängender Graph mit $n - 1$ Kanten