

Algorithmen und Datenstrukturen

Vorlesung 8

Graphenalgorithmen I: BFS- und DFS Traversierung



**UNI
FREIBURG**

Fabian Kuhn

Algorithmen und Komplexität

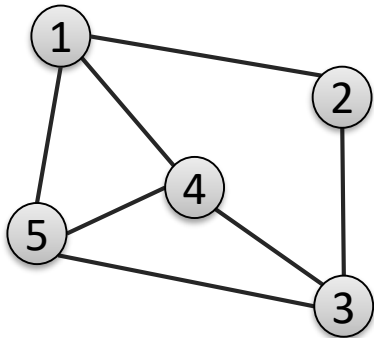
Graphen

Knotenmenge V , typischerweise $n := |V|$ (English: **vertex** or **node**)

Kantenmenge E , typischerweise $m := |E|$ (English: **edge**)

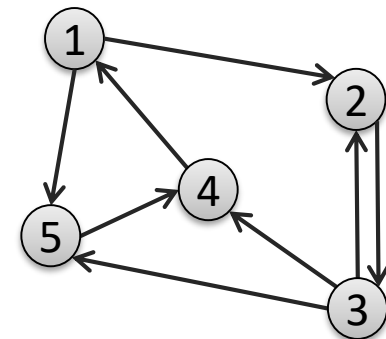
- ungerichteter Graph: $E \subseteq \{\{u, v\} : u, v \in V\}$
- gerichteter Graph: $E \subseteq V \times V$

Beispiele:



$$V = \{1, 2, 3, 4, 5\}$$

$$E = \{(1,2), (1,5), (2,3), (3,4), (3,4), (3,5), (4,1), (5,4)\}$$



$$V = \{1, 2, 3, 4, 5\}$$

$$E = \{\{1,2\}, \{1,4\}, \{1,5\}, \{2,3\}, \{3,4\}, \{3,5\}, \{4,5\}\}$$

Graph $G = (V, E)$ ungerichtet:

- Grad eines Knoten $u \in V$: Anzahl Kanten (Nachbarn) von u
$$\deg(u) := |\{u, v\} : \{u, v\} \in E|$$

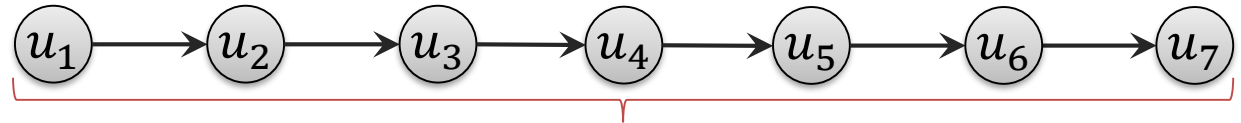
Graph $G = (V, E)$ gerichtet:

- Eingangsgrad eines Knoten $u \in V$: Anzahl eingehende Kanten
$$\deg_{in}(u) := |(v, u) : (v, u) \in E|$$
- Ausgangsgrad eines Knoten $u \in V$: Anzahl ausgehende Kanten
$$\deg_{out}(u) := |(u, v) : (u, v) \in E|$$

Pfade in einem Graph $G = (V, E)$

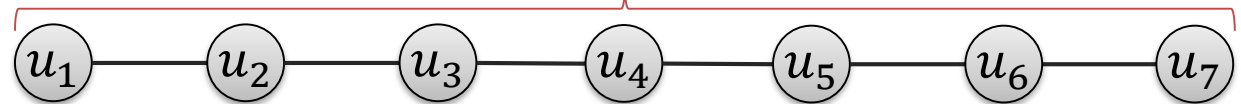
- Pfad in G : eine Folge $u_1, u_2, \dots, u_k \in V$ mit $u_i \neq u_j$ (falls $i \neq j$)
 - gerichteter Graph: $(u_i, u_{i+1}) \in E$ für alle $i \in \{1, \dots, k-1\}$
 - ungerichteter Graph: $\{u_i, u_{i+1}\} \in E$ für alle $i \in \{1, \dots, k-1\}$

Pfad, gerichtet:



Länge des Pfades: 6

Pfad, ungerichtet:



Länge eines Pfades

- Anzahl der Kanten des Pfades
- mit Kantengewichten: Summe der Kantengewichte

Pfade in einem Graph $G = (V, E)$

- Pfad in G : eine Folge $u_1, u_2, \dots, u_k \in V$ mit $u_i \neq u_j$ (falls $i \neq j$)
 - gerichteter Graph: $(u_i, u_{i+1}) \in E$ für alle $i \in \{1, \dots, k-1\}$
 - ungerichteter Graph: $\{u_i, u_{i+1}\} \in E$ für alle $i \in \{1, \dots, k-1\}$

Länge eines Pfades

- ohne Kantengewichte: Anzahl der Kanten
- mit Kantengewichten: Summe der Kantengewichte

Kürzester Pfad (shortest path) zwischen Knoten u und v

- Pfad u, \dots, v mit kleinster Länge
- **Distanz** $d_G(u, v)$: Länge eines kürzesten Pfades zwischen u und v

Durchmesser $D := \max_{u, v \in V} d_G(u, v)$

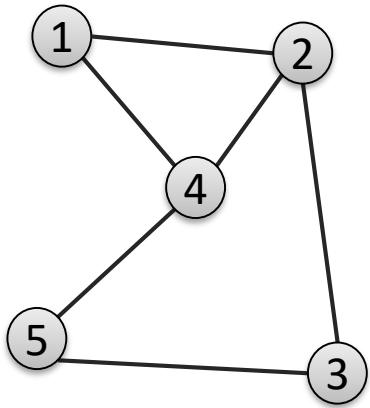
- Länge des längsten kürzesten Pfades

Repräsentation von Graphen

Zwei klassische Arten, einen Graphen im Rechner zu repräsentieren

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

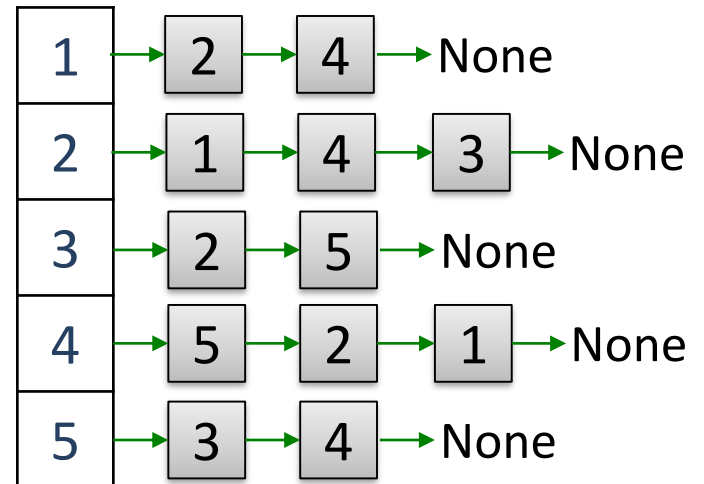
Beispiel:



Adjazenzmatrix

	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	1	0
3	0	1	0	0	1
4	1	1	0	0	1
5	0	0	1	1	0

Adjazenzlisten



Details:

- Bei **Kantengewichten**, trägt man die Gewichte (statt 0/1) in die Matrix ein (implizit: Gewicht 0 = Kante existiert nicht)
- **Gerichtete Graphen**: Ein Eintrag pro gerichtete Kante
 - Kante von i nach j : Eintrag in Zeile i und Spalte j
- **Ungerichtete Graphen**: Zwei Einträge pro Kante
 - Matrix ist in diesem Fall symmetrisch

Eigenschaften Adjazenzmatrix:

- Speichereffizient, falls $|E| = m \in \Theta(|V|^2) = \Theta(n^2)$
 - speziell für ungewichtete Graphen: nur ein Bit pro Matrixeintrag
- Nicht speichereffizient bei dünn besetzten Graphen ($m \in o(n^2)$)
- Für gewisse Algorithmen, die “richtige” Datenstruktur
- “Kante zwischen u und v ” kann in $O(1)$ Zeit beantwortet werden

Struktur

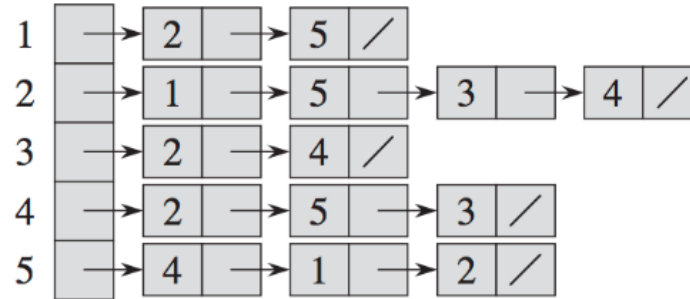
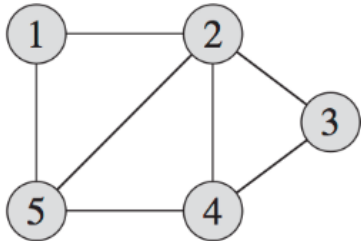
- Ein Array mit allen Knoten
- Einträge in diesem Knotenarray (oder der Liste):
 - Verkettete Listen (Arrays) mit allen Kanten des Knoten

Eigenschaften

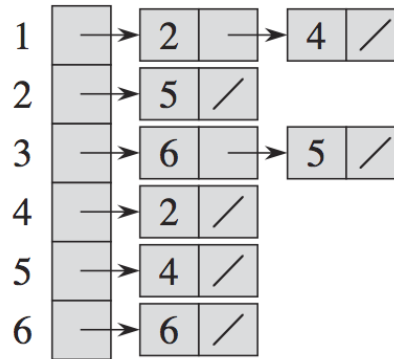
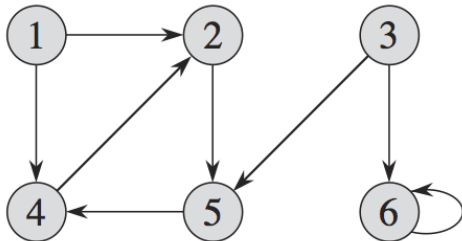
- Speichereffizient bei dünn besetzten Graphen
- Speichereffizienz immer (fast) asymptotisch optimal
 - aber bei dicht besetzten Graphen trotzdem deutlich schlechter
 - Wenn man's streng sieht, braucht man wohl $O(\log n)$ Bits pro Knoten
- Abfragen nach bestimmten Kanten nicht besonders schnell
 - Falls nötig, eine zusätzliche Datenstruktur (z.B. Hashtabelle) anlegen
- Für viele Algorithmen, die "richtige" Datenstruktur
- Z.B. für Tiefensuche (und Breitensuche)

Beispiele

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

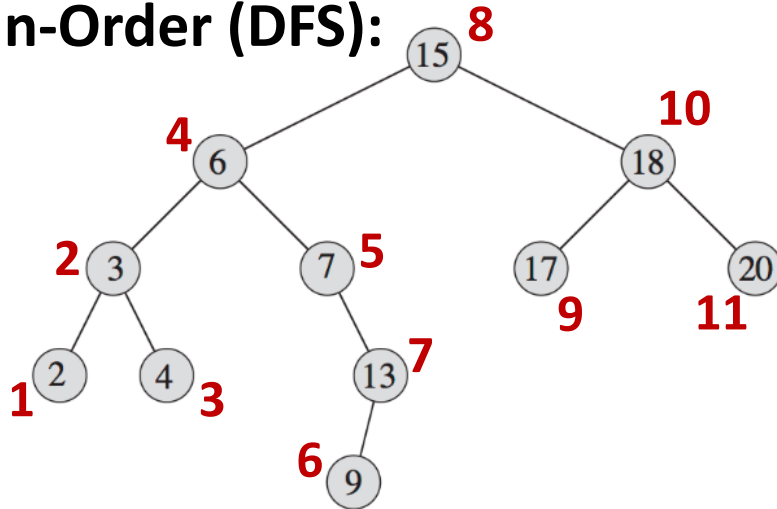
Graph-Traversierung (Graph-Exploration) informell

- 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...

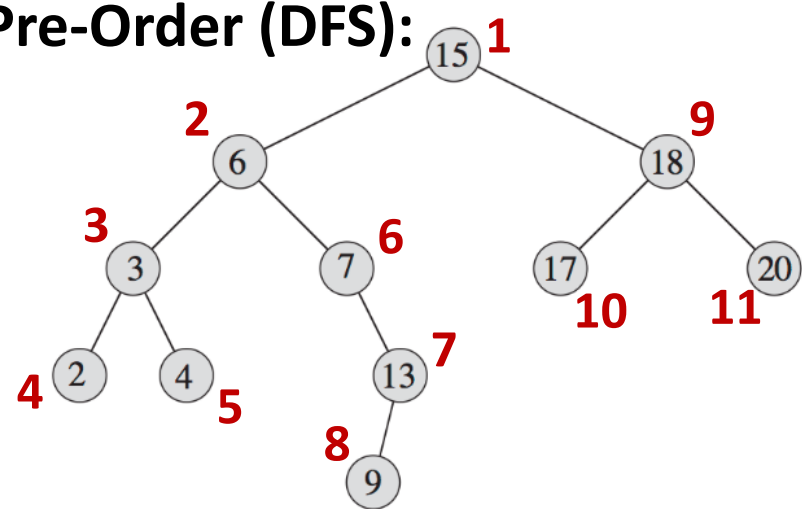
Traversieren eines binären Suchbaums

Ziel: Besuche alle Knoten eines binären Suchbaums einmal

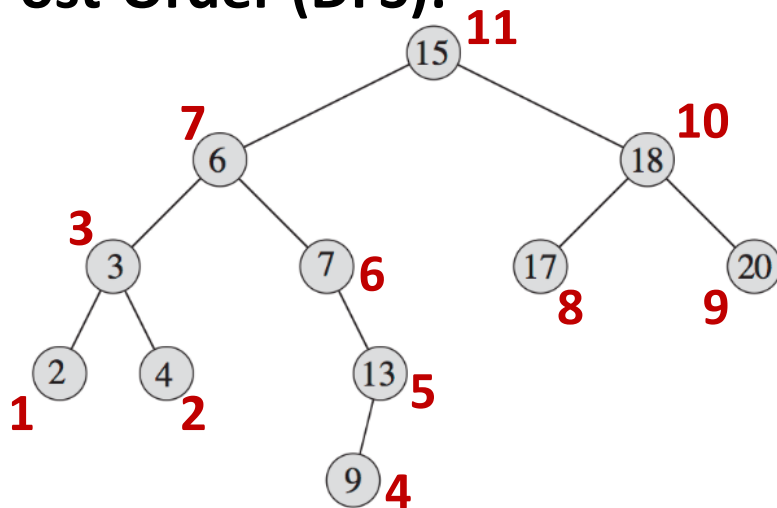
In-Order (DFS):



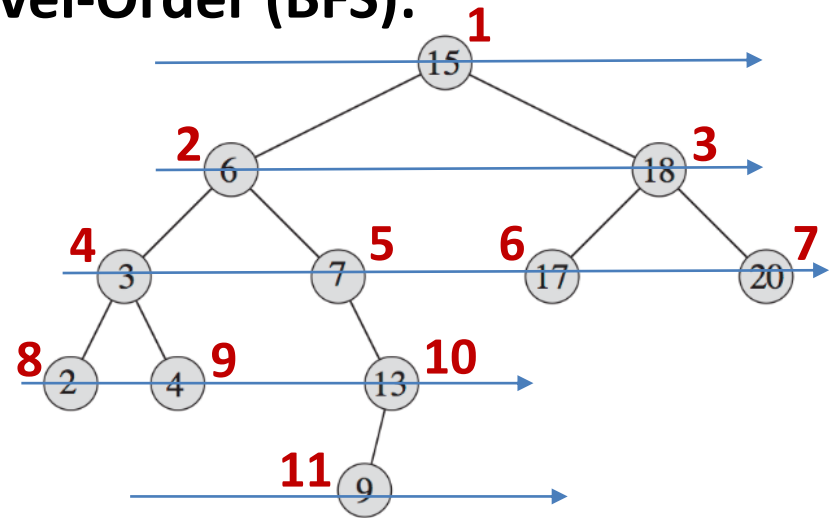
Pre-Order (DFS):



Post-Order (DFS):



Level-Order (BFS):



Breitensuche (BFS Traversierung)

- Lösung mit einer Warteschlange:
 - Wenn ein Knoten besucht wird, werden seine Kinder in die Queue eingereiht

BFS-Traversal:

```
Q = new Queue()
Q.enqueue(root)
while not Q.empty() do
    node = Q.dequeue()
    visit(node)
    if node.left != null
        Q.enqueue(node.left)
    if node.right != null
        Q.enqueue(node.right)
```

Unterschiede Binärbaum $T \Leftrightarrow$ allg. Graph G

- Graph G kann Zyklen haben
- In T haben wir eine Wurzel und kennen von jedem Knoten die Richtung zur Wurzel
 - etwas allgemeiner bezeichnen wir solche Bäume auch als gewurzelte Bäume

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

Breitensuche (BFS Traversierung)

- Am Anfang ist v .marked bei allen Knoten v auf **false** gesetzt

BFS-Traversal(s):

```
for all u in V: u.marked = false;
```

```
Q = new Queue()
```

```
s.marked = true
```

```
Q.enqueue(s)
```

```
while not Q.empty() do
```

```
    u = Q.dequeue()
```

```
    visit(u)
```

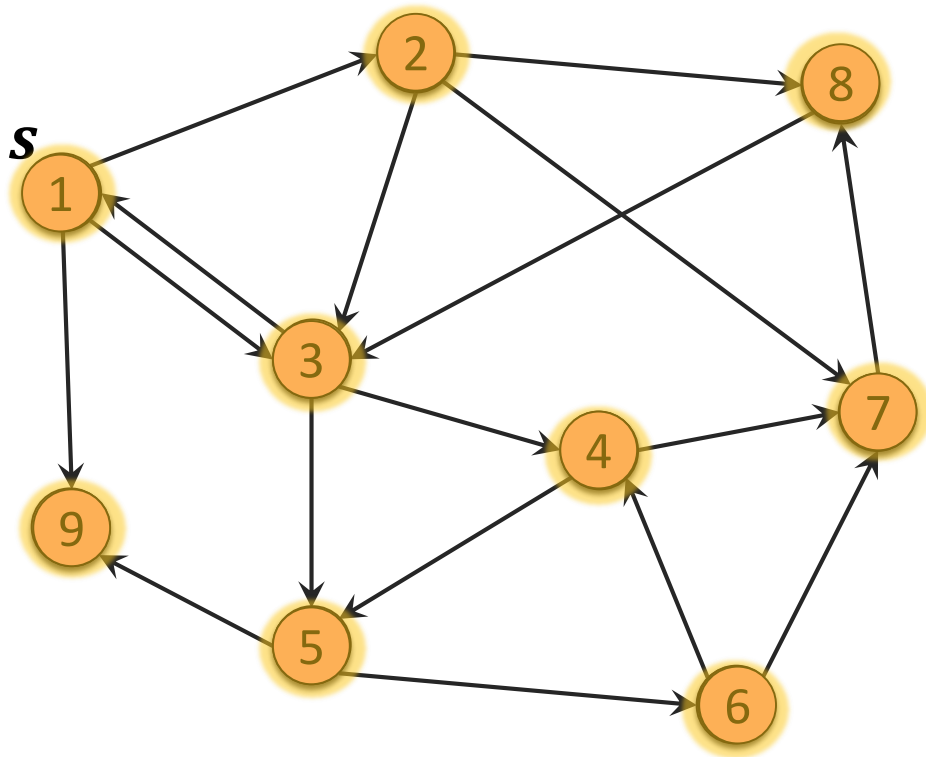
```
    for v in u.neighbors do
```

```
        if not v.marked then
```

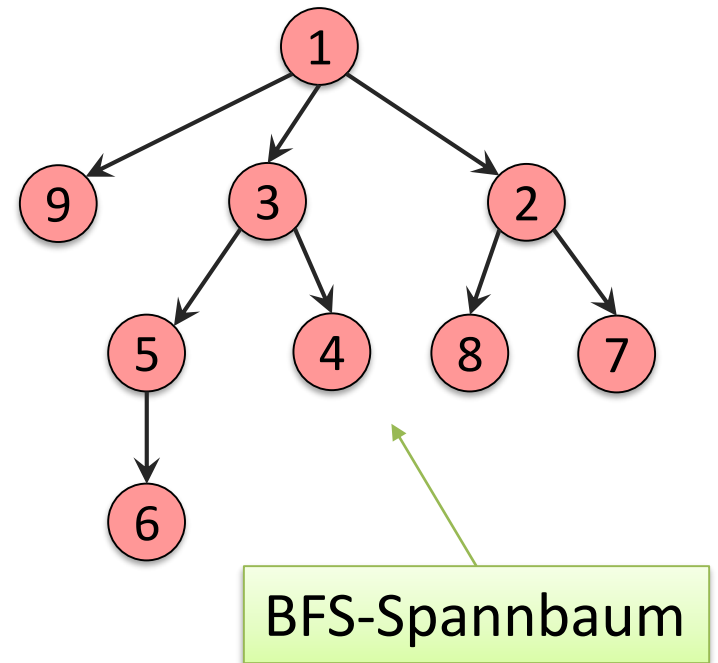
```
            v.marked = true;
```

```
            Q.enqueue(v)
```

Breitensuche Beispiel



Besuchsreihenfolge:



FIFO Queue:



In der Folge benennen wir die Knoten folgendermaßen

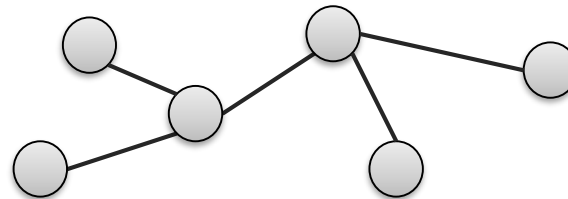
- weiße Knoten: Knoten, welche der Alg. noch nicht gesehen hat
- graue (vorher: blaue) Knoten: markierte Knoten
 - Knoten werden grau, wenn sie in die Warteschlange eingefügt werden
 - Knoten sind grau, solange sie in der Warteschlange sind
- schwarze (rote) Knoten: besuchte Knoten
 - Knoten werden schwarz,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

Die Laufzeit der BFS-Traversierung ist $O(n + m)$.

- **Annahme:** Graph durch Adjazenzlisten gegeben
 - Falls der Graph durch die Adjazenzmatrix gegeben ist, ist die Laufzeit $\Theta(n^2)$.
 - weiße Knoten: Knoten, welche der Algorithmus noch nicht gesehen hat
 - graue (vorher: blaue) Knoten: markierte Knoten
 - schwarze (vorher: rote) Knoten: besuchte Knoten
-
- Jeder Knoten kommt höchstens einmal in die Warteschlange.
 - Insgesamt gibt das $O(n)$ Warteschlangenoperationen.
 - Wenn der Knoten v aus der Warteschlange entfernt wird, werden alle seine Nachbarn angeschaut.
 - Jede gerichtete Kante wird so genau ein Mal betrachtet .
 - Adjazenzlisten: Kosten pro Knoten = $O(\#\text{Nachbarn})$
 - Man muss einmal durch die Liste der Nachbarn gehen.
 - Adjazenzmatrix: Kosten pro Knoten = $O(n)$
 - Man muss die ganze Zeile oder Spalte der Matrix durchgehen.

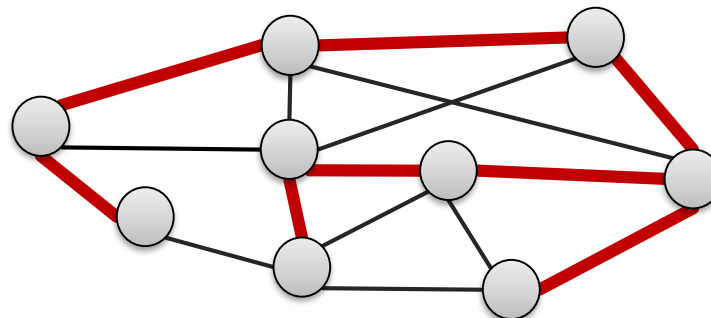
Baum:

- Ein zusammenhängender, ungerichteter Graph ohne Zyklen
 - bzw. auch ein gerichteter Graph, aber dann darf es auch ohne Berücksichtigung der Richtungen keine Zyklen haben



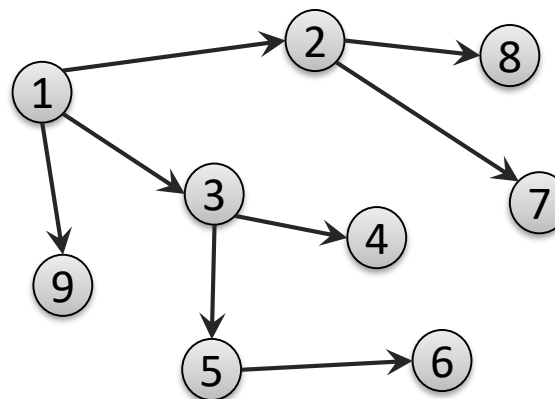
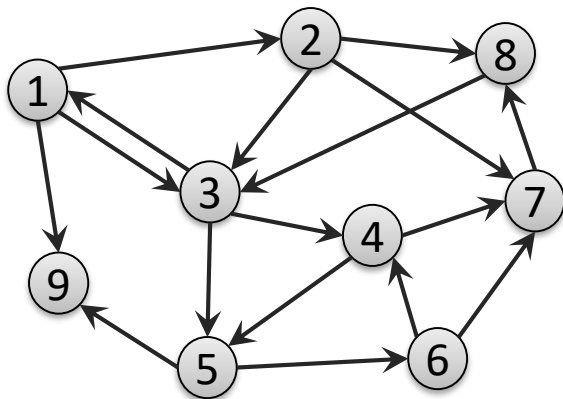
Spannbaum eines Graphen G :

- Ein Teilgraph T , so dass T ein Baum ist, welcher alle Knoten von G enthält
 - Teilgraph: Teilmenge der Knoten und Kanten, welche einen Graph ergeben



Mit der BFS-Traversierung können wir folgendermaßen einen Spannbaum generieren (falls G zusammenhängend ist):

- Jeder Knoten u merkt sich, von welchem Knoten v aus er markiert wurde
- Der Knoten v ist dann der Parent-Knoten von u
 - Da jeder Knoten genau einmal markiert wird, ist der Parent von jedem Knoten eindeutig definiert, s ist die Wurzel und hat keinen Parent.

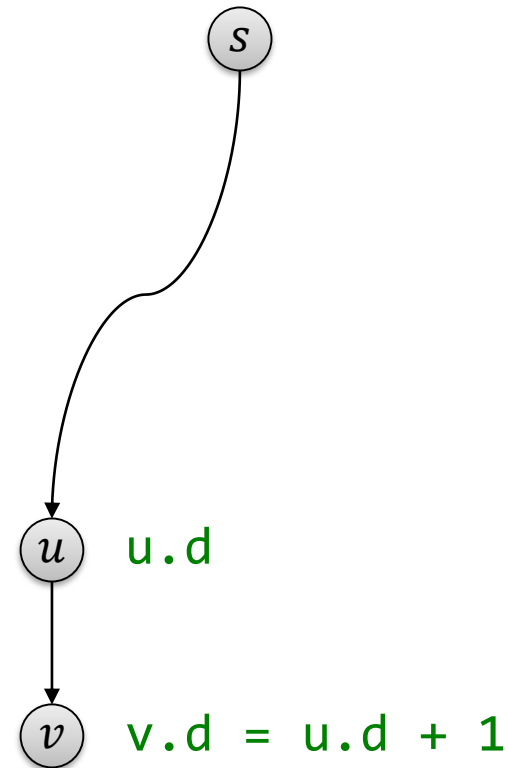


BFS Baum: Pseudocode

- Wir merken uns zusätzlich die Distanz von 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 der Wurzel s zu jedem Knoten u gleich $d_G(s, u)$.

- Baumdistanz von der Wurzel: $d_T(s, u) = u.d$
- Wir müssen also zeigen, dass $u.d = d_G(s, u)$
- Es gilt sicher, dass $u.d \geq d_G(s, u)$
 - Weil $u.d = d_T(s, u)$, ist das äquivalent zu $d_T(s, u) \geq d_G(s, u)$
 - Das gilt natürlich, weil ja der Pfad in T auch ein Pfad in G ist, die Distanz in T kann also sicher nicht kleiner sein als die Distanz in G .

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 $i = 1, \dots, r - 1$)

Beweis: Per Induktion über die Warteschlangen-Operationen

- **Verankerung:** Am Anfang ist nur s mit $d.s = 0$ in der Queue.

- **Induktionsschritt:**

Induktionsvoraussetzung

- dequeue-Operation: $Q = \langle v_1, v_2, \dots, v_r \rangle$, $v_r.d \leq v_1.d + 1 \leq v_2.d + 1$

- enqueue-Operation: $u, \langle v_1, v_2, \dots, v_r \rangle, v$

zuletzt entfernter
Knoten

Neuer Knoten in
der Queue



Wenn v in die Queue eingefügt wird, wird gerade der zuletzt entfernte Knoten u bearbeitet (v ist Nachbar von u).

$$\Rightarrow v.d = u.d + 1$$

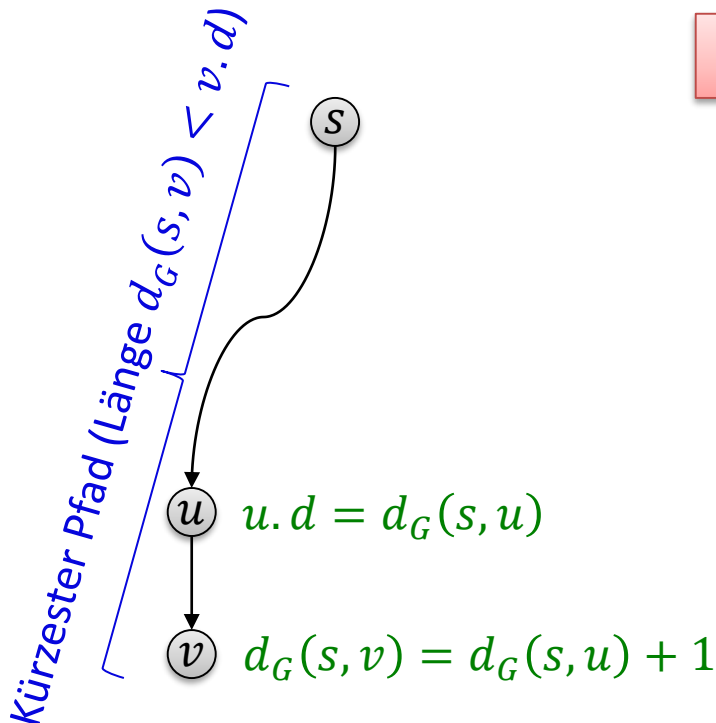
- **Aus Induktionsvoraussetzung:**

$$v.d \leq v_1.d + 1: \quad v.d = u.d + 1 \leq v_1.d + 1$$

$$v_r.d \leq v.d: \quad v_r.d \leq u.d + 1 = v.d$$

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 ist 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 v**
- **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

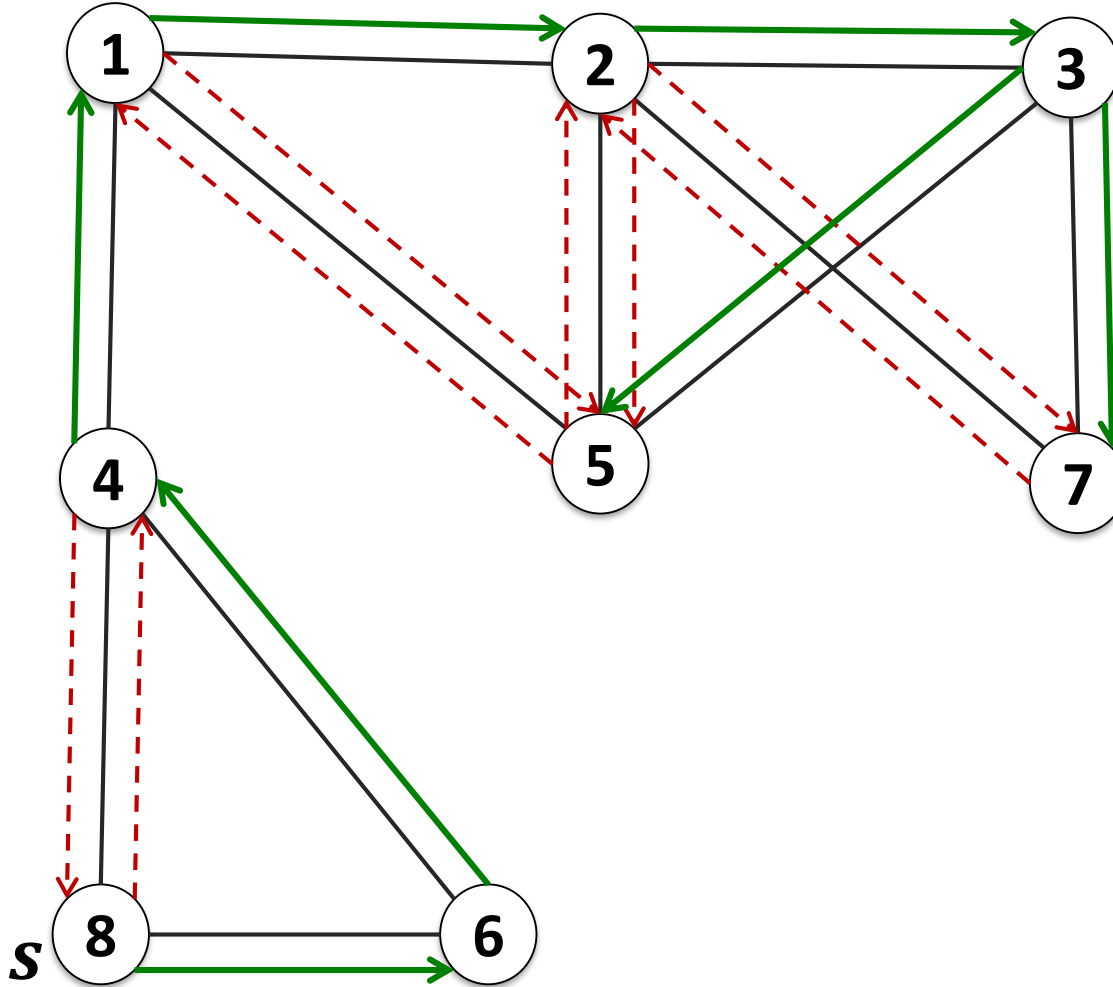
DFS-Traversal(s):

```
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



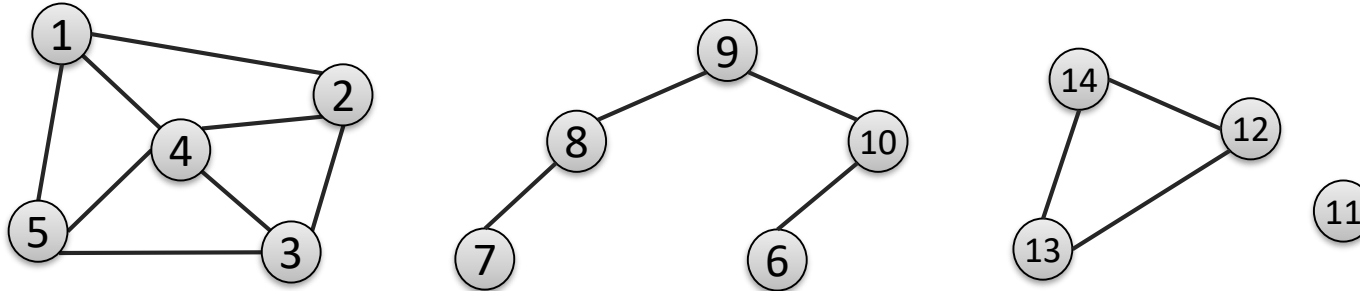
In der gleichen Art wie bei der Breitensuche, kann man auch bei der Tiefensuche einen Spannbaum konstruieren

Die 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
- Wir führen die rekursive Funktion zur Traversierung für jeden Knoten höchstens einmal aus.
- Die Zeit, um Knoten v zu verarbeiten ist proportional zur Anzahl der (ausgehenden) Kanten von v

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)

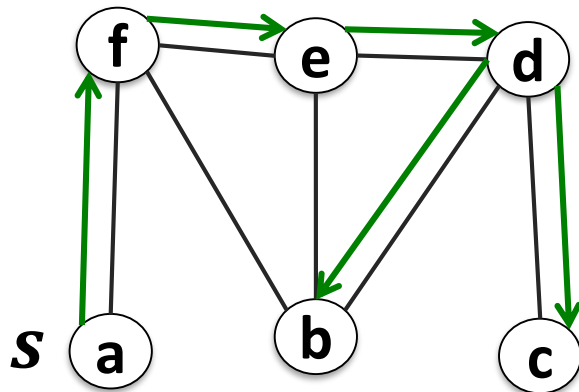
DFS- “Klammer”-Theorem

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.

Beispiel:



$t_{a,1}$ $t_{f,1}$ $t_{e,1}$ $t_{d,1}$ $t_{c,1}$ $t_{c,2}$ $t_{b,1}$ $t_{b,2}$ $t_{d,2}$ $t_{e,2}$ $t_{f,2}$ $t_{a,2}$

[[[[[] []]]]]

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.

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.

Beweis:

- Graue Knoten bilden immer einen Pfad, der bei s startet.
 - Pfad beginnt bei s , aktueller Knoten am Ende des Pfads
 - Neuer Knoten w wird grau $\Rightarrow w$ Nachbar von aktivem Knoten
 - Knoten wird schwarz \Rightarrow aktiver Knoten beendet Rekursion



- Knoten, v ist genau dann im Teilbaum von u wenn u Teil des Pfads ist, wenn v grau wird und damit gdw. $t_{u,1} < t_{v,1} < t_{u,2}$.
- Knoten v ist dann weiter rechts im grauen Pfad als u und muss dann auch vor u wieder Schwarz werden, also $t_{v,2} < t_{u,2}$

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

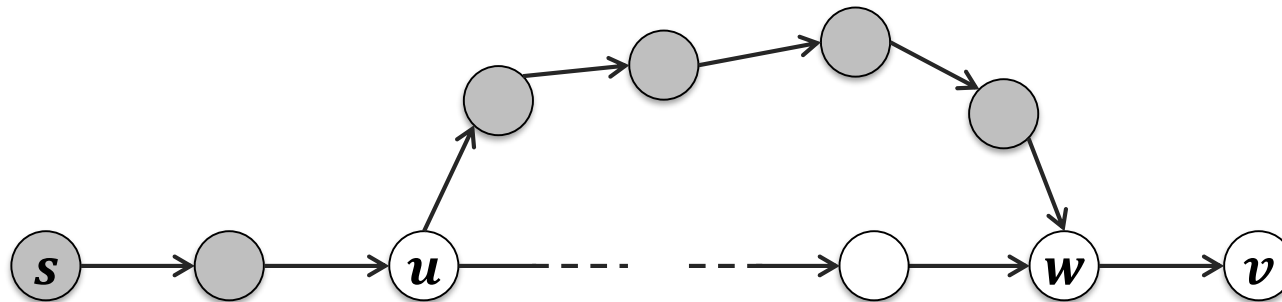
- Zwei Intervalle sind entweder disjunkt, oder das eine ist komplett im anderen enthalten.
- Wieso “Klammer”-Theorem:
Wenn man bei jedem $t_{v,1}$ eine öffnende Klammer und bei jedem $t_{v,2}$ eine schließende Klammer hinschreibt, bekommt man ein Klammersausdruck, welcher korrekt geschachtelt ist.
-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

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

zur Zeit $t_{u,1}$

Beweis:

- **Widerspruchsbeweis:** Nehme an, dass es einen Knoten v gibt, zu dem es einen weißen Pfad gibt, der aber nicht im Teilbaum ist
- Nehme an, dass v der Knoten mit dieser Eigenschaft ist, der unmittelbar vor dem Einfügen von u den kürzesten komplett weißen Pfad von u hat.



Klassifizierung der Kanten (bei DFS-Suche)

Baumkanten:

- (u, v) ist eine Baumkante, falls v von u aus entdeckt wird
 - Bei Betrachten von (u, v) ist v weiß

Rückwärtskanten:

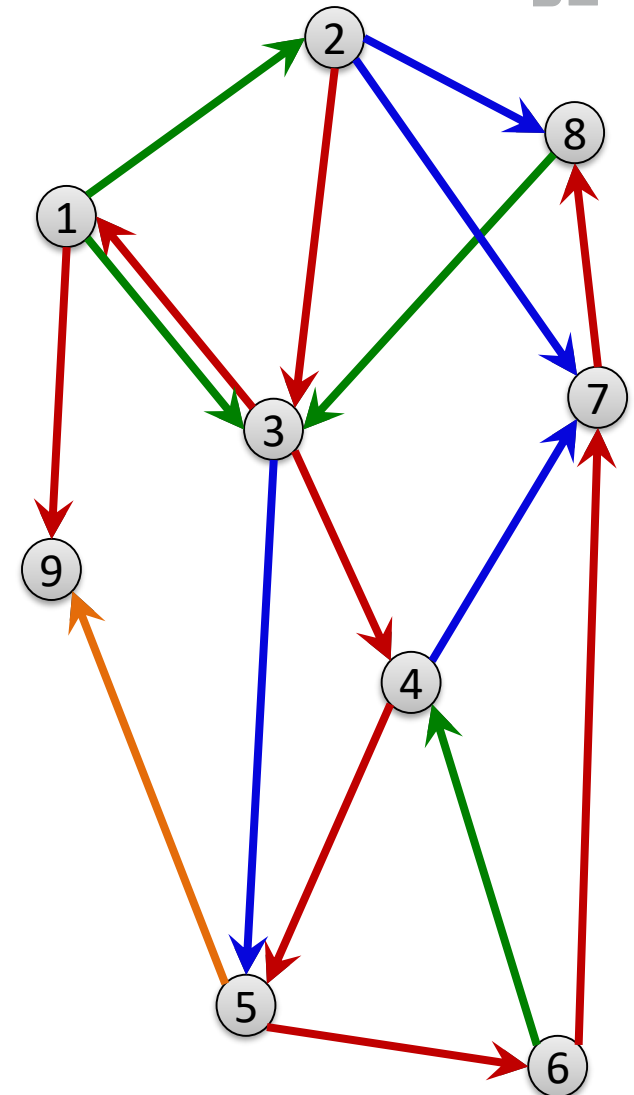
- (u, v) ist eine Rückwärtskante, falls v ein Vorgängerknoten von u ist
 - Bei Betrachten von (u, v) ist v grau

Vorwärtskanten:


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

Queranten:


- Alle übrigen Kanten
 - Bei Betrachten von (u, v) ist v schwarz




Klassifizierung der Kanten (bei DFS-Suche)

Baumkante (u, v) : 

- Knoten v wird als weisser Nachbar von u aus besucht
 - Falls bei Betrachten von (u, v) v **weiß** ist $\Rightarrow (u, v)$ **Baumkante**


Rückwärtskante (u, v) : 

- Teilbaum von u wird fertig besucht, bevor u schwarz wird
 - Falls bei Betrachten von (u, v) v **grau** ist $\Rightarrow (u, v)$ **Rückwärtskante**

Vorwärtskante (u, v) : 

$$t_{u,1} < t_{v,1} < t_{v,2} < t_{u,2}$$

- v ist in einem Teilbaum von u , der schon fertig besucht ist
 - Da v im Teilbaum von u ist, muss v **schwarz** und $t_{v,1} > t_{u,1}$ sein

Querkante (u, v) : 

$$t_{v,1} < t_{v,2} < t_{u,1} < t_{u,2}$$

- Solange u grau ist, sind alle neu besuchten Knoten im Teilbaum von u , v wird also vor u besucht: v **schwarz** und $t_{v,1} < t_{u,1}$.

- Bei ungerichteten Graphen wird jede Kante $\{u, v\}$ zwei Mal betrachtet (einmal von u und einmal von v)
- Wir klassifizieren die Kante gemäss der ersten Betrachtung

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

Beweis:

- Wir nehmen o.B.d.A. an, dass u vor v grau wird.
- Aus dem Theorem über weisse Pfade wissen wir, dass v besucht wird, solange u noch grau ist.
- Falls die Kante $\{u, v\}$ zuerst von u aus betrachtet wird, ist der Knoten v noch weiss $\implies \{u, v\}$ ist eine Baumkante
- Falls die Kante $\{u, v\}$ zuerst von v aus betrachtet wird, ist der Knoten u noch grau $\implies \{u, v\}$ ist eine Rückwärtskante

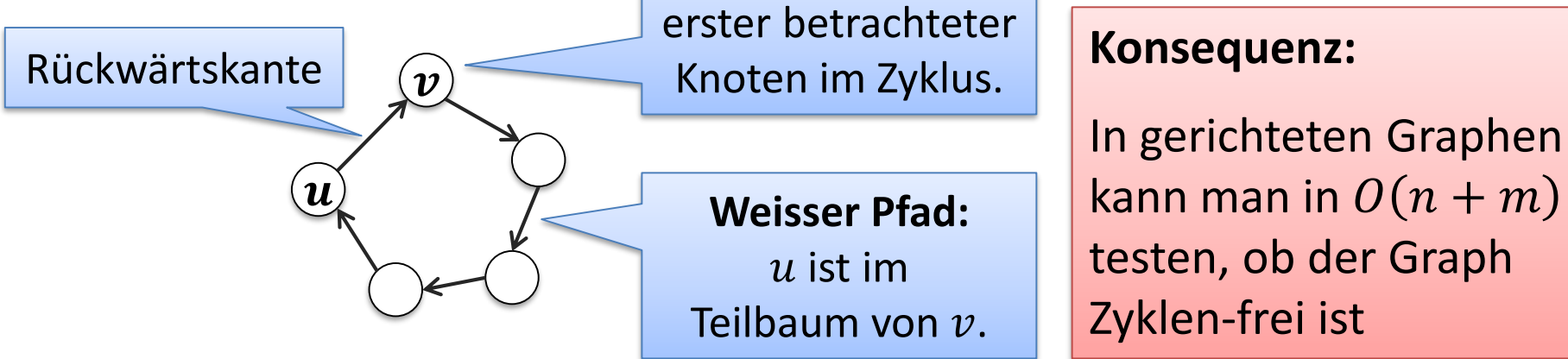
DFS – Gerichtete Graphen

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

Rückwärtskante \Rightarrow Zyklus:

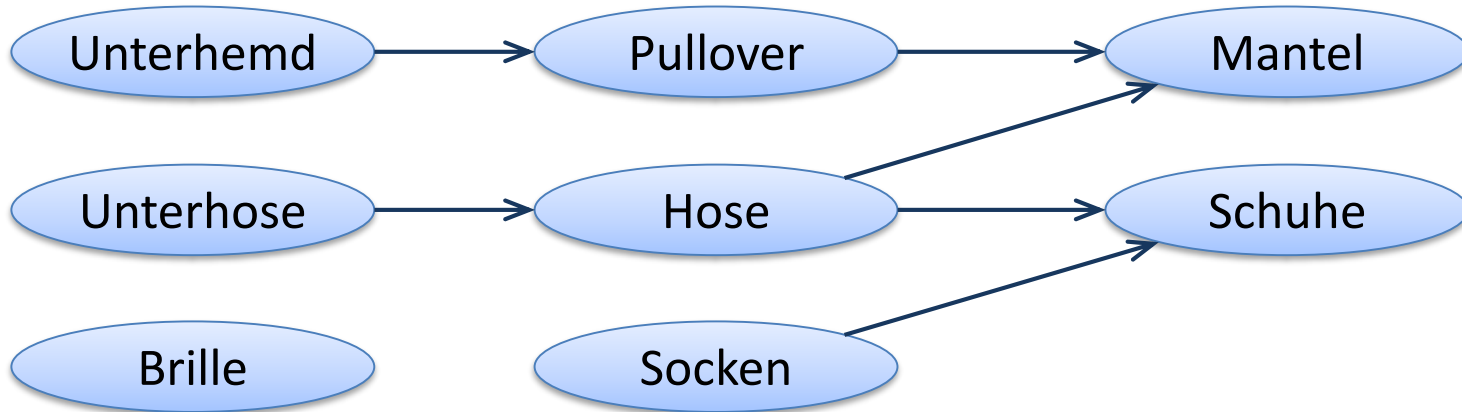


Zyklus \Rightarrow Rückwärtskante:



Zyklusfreie, gerichtete Graphen:

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

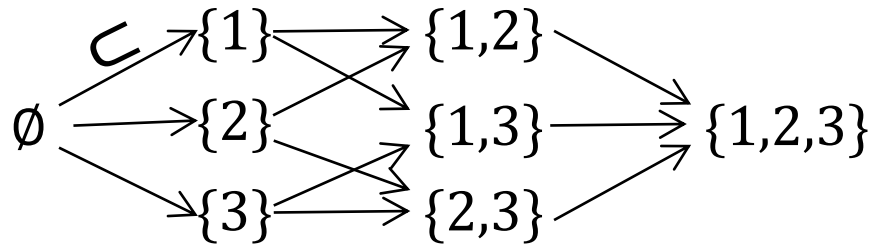


Topologische Sortierung:

- 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



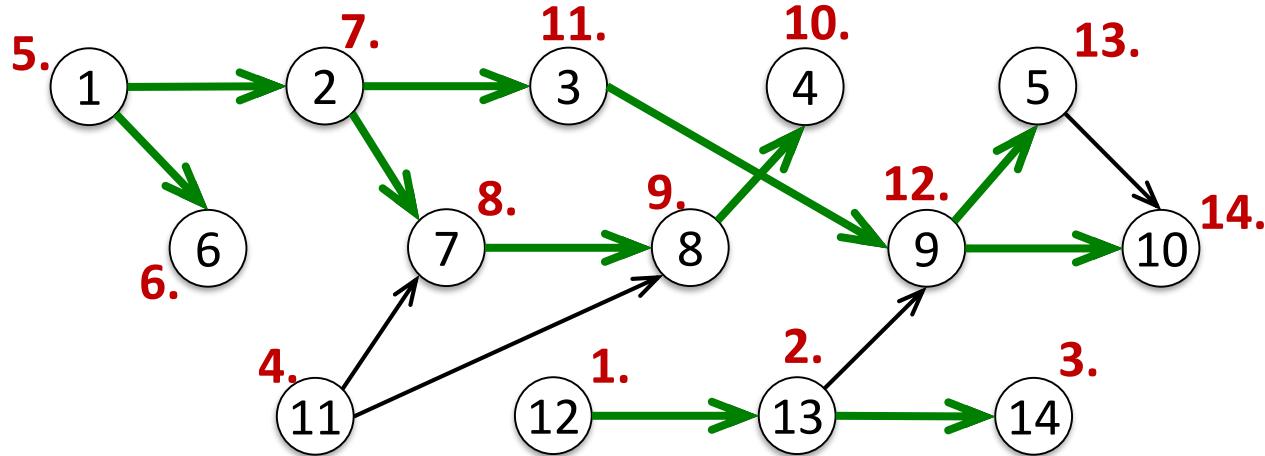
Topologische Sortierung:

- 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

$\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}$

Topologische Sortierung: Algorithmus

Führe DFS aus...



$t_{1,1}$ $t_{2,1}$ $t_{3,1}$ $t_{9,1}$ $t_{10,1}$ $t_{10,2}$ $t_{5,1}$ $t_{5,2}$ $t_{9,2}$ $t_{3,2}$ $t_{7,1}$ $t_{8,1}$ $t_{4,1}$ $t_{4,2}$ $t_{8,2}$
 $t_{7,2}$ $t_{2,2}$ $t_{6,1}$ $t_{6,2}$ $t_{1,2}$ $t_{11,1}$ $t_{11,2}$ $t_{12,1}$ $t_{13,1}$ $t_{14,1}$ $t_{14,2}$ $t_{13,2}$ $t_{12,2}$

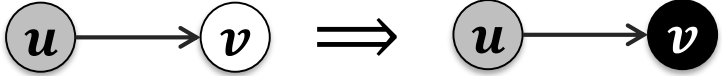
Beobachtung:

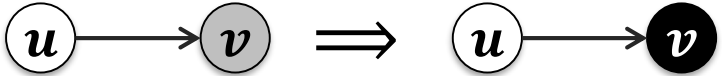
- Knoten ohne Nachfolger werden als erstes besucht (schwarz gef.)
- Besuchreihenfolge ist umgekehrte topologische Sortierung

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

Beweis:

- Wir müssen zeigen, dass für jede Kante (u, v) , Knoten v vor Knoten u schwarz gefärbt wird.

- **Fall 1: u wird vor v grau:**  \implies
 - Dann ist v im Teilbaum von u und es gilt deshalb $t_{u,1} < t_{v,1} < t_{u,2}$.
Wegen dem Klammertheorem gilt dann auch $t_{v,2} < t_{u,2}$.

- **Fall 2: v wird vor u grau:**  \implies
 - u kann nur grau werden, bevor v schwarz wird, wenn u im Teilbaum von v ist. Dann hätte es aber einen gerichteten Pfad von v nach $u \implies$ Zyklus!

Stark zusammenhängende Komponenten

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

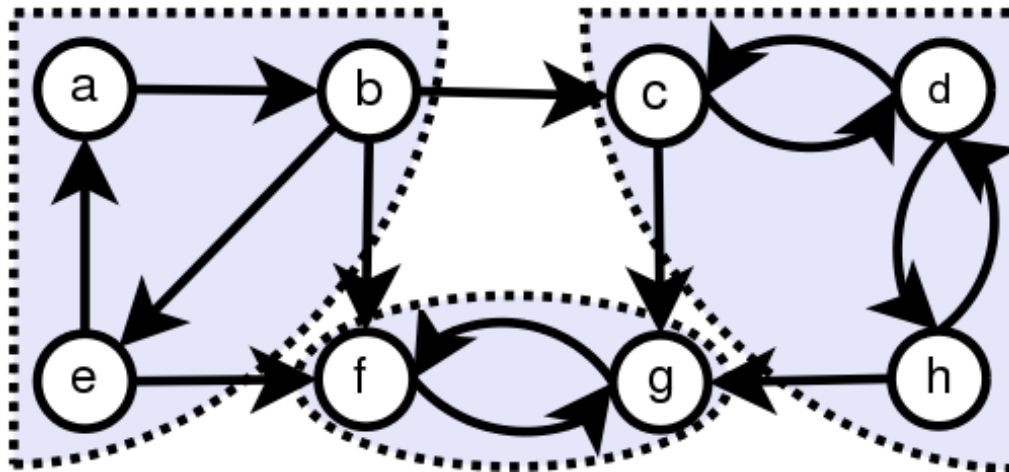


Bild: Wikipedia

- 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]