Informatik II - SS 2016

(Algorithmen & Datenstrukturen)

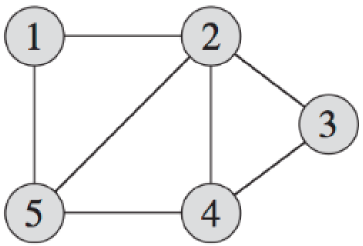
Vorlesung 14 (10.6.2016)

Graphenalgorithmen II

Fabian Kuhn
Algorithmen und Komplexität

Repräsentation von Graphen

Beispiele aus [CLRS]:

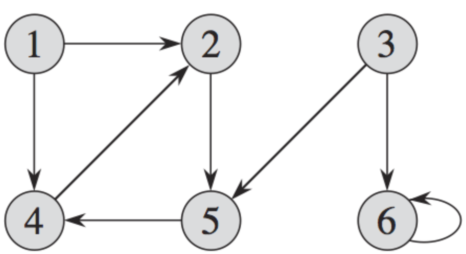


Adjazenzlisten

1	→ 2	→ 5 /		
2	→ 1	→ 5	→ 3	→ 4 /
3	→ 2	→ 4 /		
4	→ 2	→ 5	→ 3 /	
5	→ 4	→ 1	→ 2 /	

Adj.-Matrix

	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	→ 4 /
2	→ 5 /	
3	→ 6	→ 5 /
4	→ 2 /	
5	→ 4 /	
6	→ 6 /	

	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-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…

BFS Traversierung / BFS Baum: Pseudocode

- 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;    grau
    s.parent = NULL;
    s.d = 0
    Q.enqueue(s)
    while not Q.empty() do
        u = Q.dequeue()
        visit(u)  schwarz
        for v in u.neighbors do
            if not v.marked then
                v.marked = true;    grau
                v.parent = u;
                v.d = u.d + 1;
                Q.enqueue(v)
```

Analyse Breitensuche

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

Analyse Breitensuche

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

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

schwierig: $u.d \leq d_G(s,u)$

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

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

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

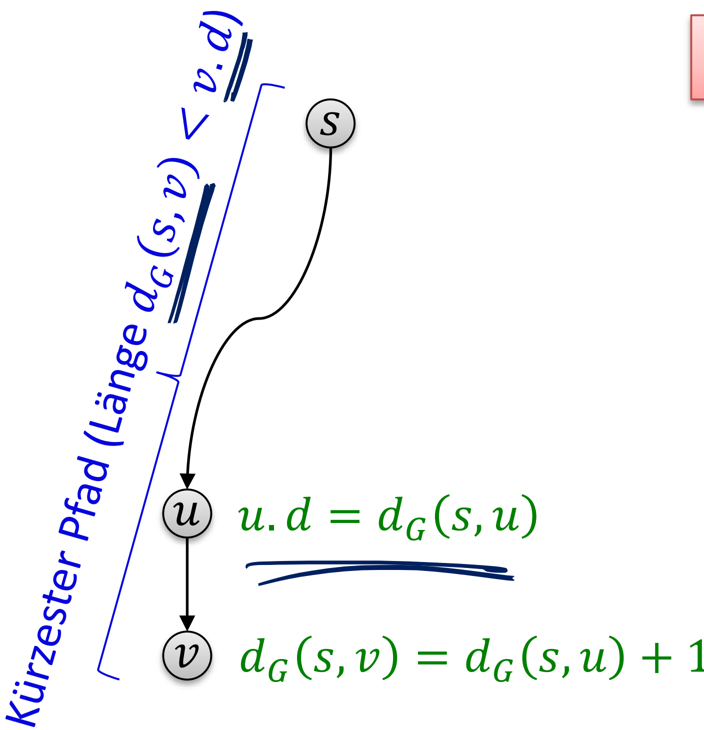
$v_1, v_2, v_3, \ldots, v_r$

3 3 3 .. 3 4 4

Analyse Breitensuche

Im BFS-Baum eines ungewichteten Graphen ist die Distanz von jedem Knoten $\boldsymbol{u}$ zur Wurzel $\boldsymbol{s}$ gleich $\boldsymbol{d_G(s,u)}$. $v.d \geq d_G(s,v)$

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



$$\boldsymbol{v.d} > d_G(s,v) = d_G(s,u) + 1 = \boldsymbol{u.d + 1}$$

Betrachte dequeue von $\boldsymbol{u}$:

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

  Lemma: $v.d \leq u.d + 1$

u —— $u.d+1$

Tiefensuche in allgemeinen Graphen

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

Knoten alle weiss

- **Markiere Knoten v** (am Anfang ist $v = s$) grau
- Besuche die Nachbarn von v der Reihe nach *rekursiv*
- Nachdem alle Nachbarn besucht sind, **besuche s** post order

  schwarz
- **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

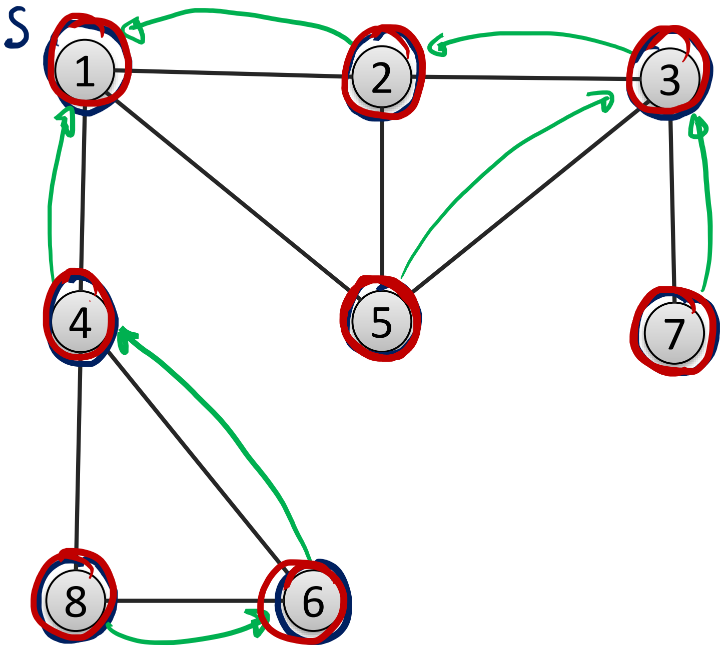
- entspricht der Postorder-Traversierung in Bäumen
- Fall man gleich beim Markieren den Knoten besucht, entspricht es der Preorder-Traversierung

Tiefensuche: Pseusocode

```
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



Tiefensuche: Analyse

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

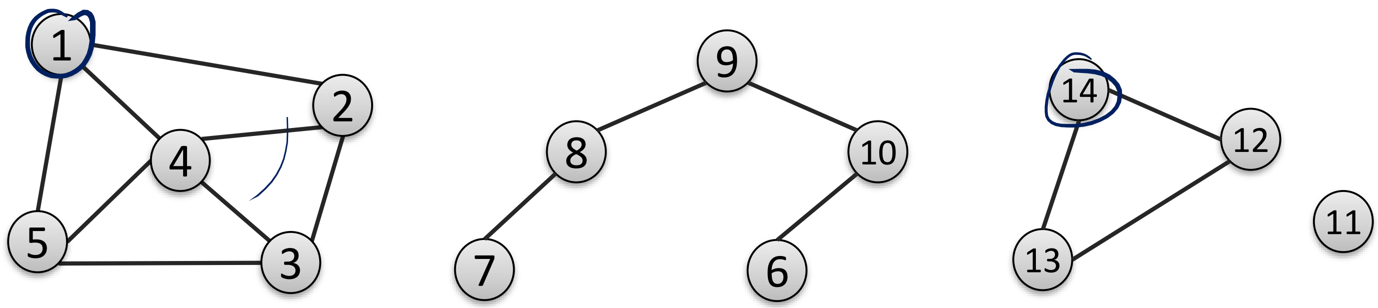
- Die Eigenschaften dieses DFS-Baums werden wir noch anschauen

Die Laufzeit der Tiefensuche (DFS-Traversierung) ist $\boldsymbol{O(n+m)}$.

- 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
```

$O(m+n)$

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

DFS- “Klammer”-Theorem

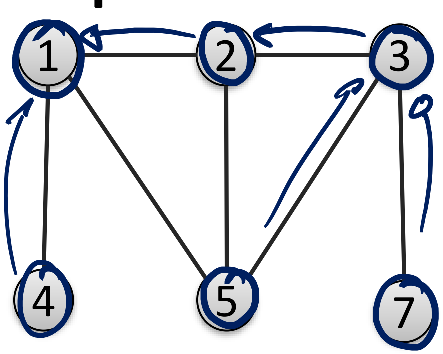
Wir definieren für jeden Knoten v die folgenden zwei Zeitpunkte

- $t_{v,1}$: 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:

$t_{v,1}$ $t_{u,1}$ $t_{u,2}$ $t_{v,2}$ [[ ]] [ ]



$t_{1,1}$ $t_{2,1}$ $t_{3,1}$ $t_{5,1}$ $t_{5,2}$ $t_{7,1}$ $t_{7,2}$ $t_{3,2}$ $t_{2,2}$ $t_{4,1}$ $t_{4,2}$ $t_{1,2}$

DFS- “Klammer”-Theorem

Theorem: Im DFS-Baum ist ein Knoten v ist 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:

graue Knoten (G zus.-hängend)

DFS- “Klammer”-Theorem

Theorem: Im DFS-Baum ist ein Knoten v ist 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.

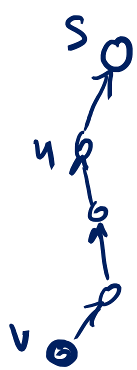
Beweis:

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

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

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

zu zeigen $t_{v,2} < t_{u,2}$ ✓



✓

v im Teilbaum von u

Fall 2: $t_{v,1} > t_{u,2}$

v nicht im Teilbaum von u

DFS- “Klammer”-Theorem

Theorem: Im DFS-Baum ist ein Knoten v ist 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.

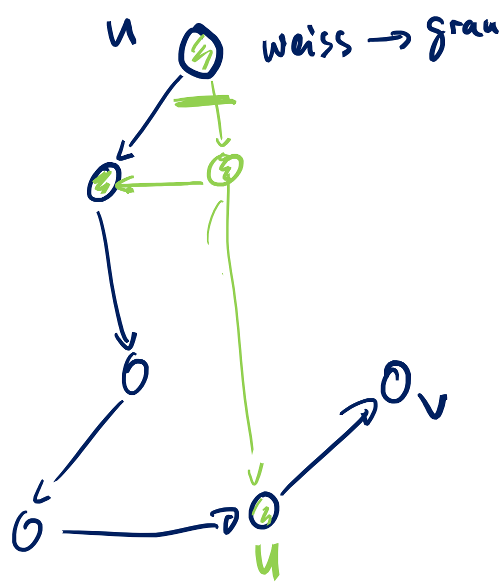
Implikationen

- 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 Klammerausdruck, welcher korrekt geschachtelt ist.

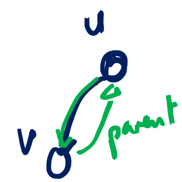
Weiße Pfade

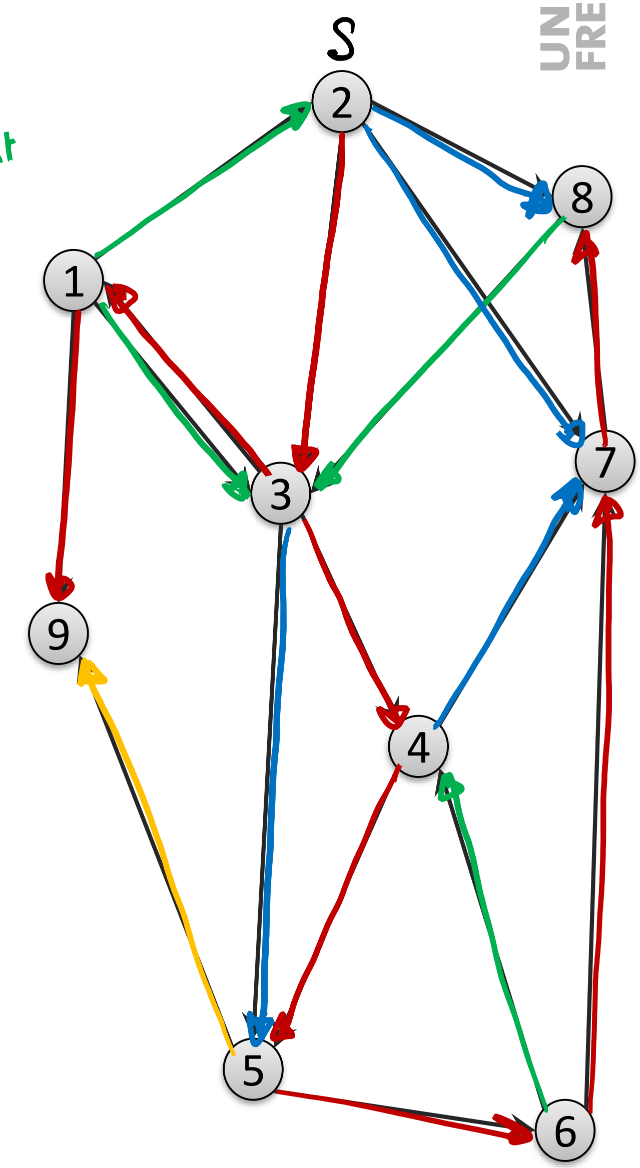
Theorem: In einem DFS-Baum ist ein Knoten v genau dann im Teilbaum eines Knoten u, falls unmittelbar vor dem Markieren von u, ein komplett weißer Pfad von u nach v besteht. zur Zeit $t_{u,1}$

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

Querkanten:

- Alle übrigen Kanten

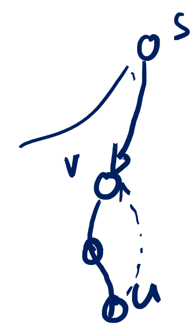
Klassifizierung der Kanten (bei DFS-Suche)

Baumkante (u, v):

bei Betrachtung von (u,v)

falls v weiss $\rightarrow$ (u,v) Baumkante

Rückwärtskante (u, v):



bei Betr. von (u,v)

falls v grau $\rightarrow$ (u,v) Rückwärtskante

Vorwärtskante (u, v):

Querkante (u, v):

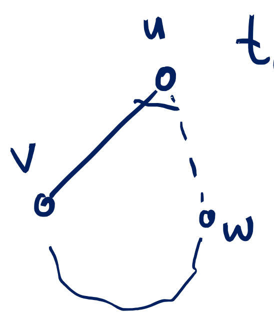
v schwarz

Vorwärtskante: $t_{v,2}, t_{v,1} > t_{u,1}$

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

DFS – Ungerichtete Graphen

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



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

$\{u,v\}$ zuerst von u aus betrachtet

v weiss → Baumkante

$\{u,v\}$ zuerst von v aus betr.

u grau → Rückwärtskante

DFS-visit(u):

u wird grau

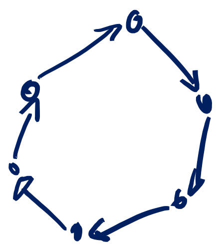
{ [gehe durch Nachbarn]

u wird schwarz

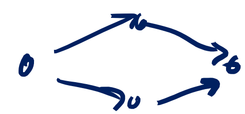
DFS – Gerichtete Graphen

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

Zyklus



Rückw.-kante (u,v) $\rightarrow$ Zyklus

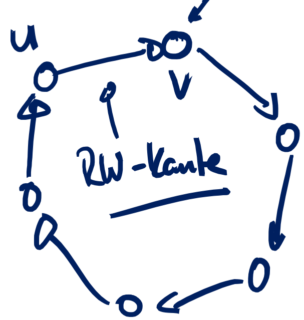


Implikation:

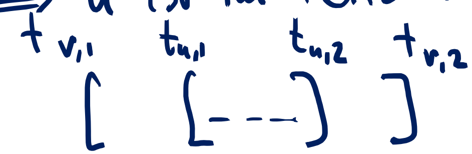
Man kann in $O(m+n)$ Zeit erkennen, ob ein gegebener ger. Graph zyklenfrei ist.

Zyklus $\longrightarrow$ Rückwärtskante

erster besuchter Knoten



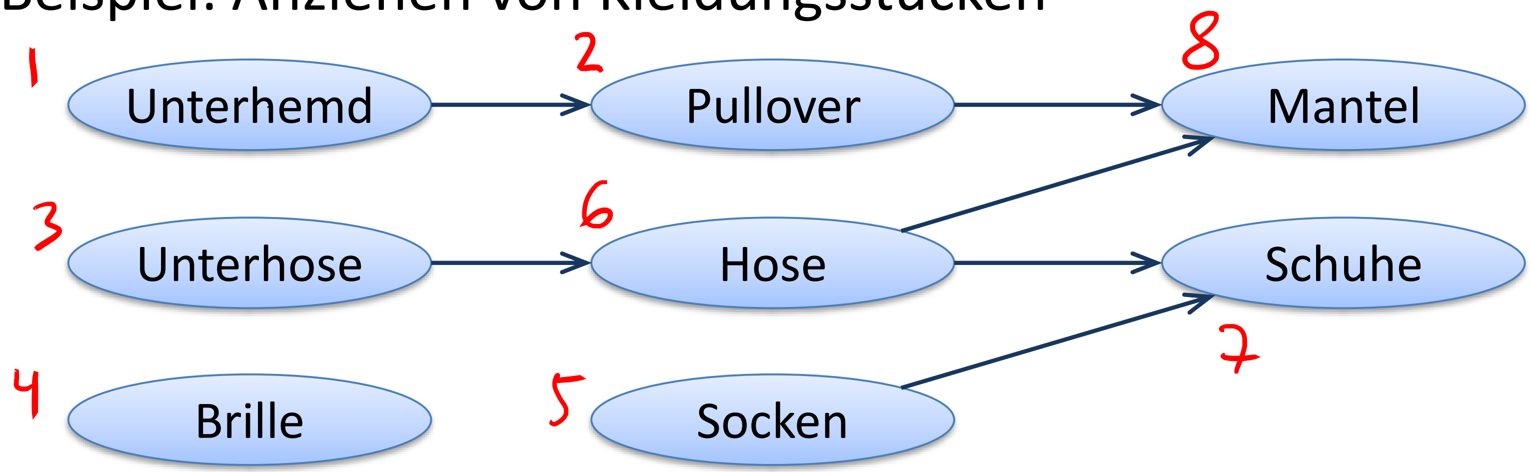
$\Longrightarrow$ u ist im Teilbaum von v



Anwendung: Topologische Sortierung

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

Topologische Sortierung: Etwas formaler…

Zyklenfreie, gerichtete Graphen:

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

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

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

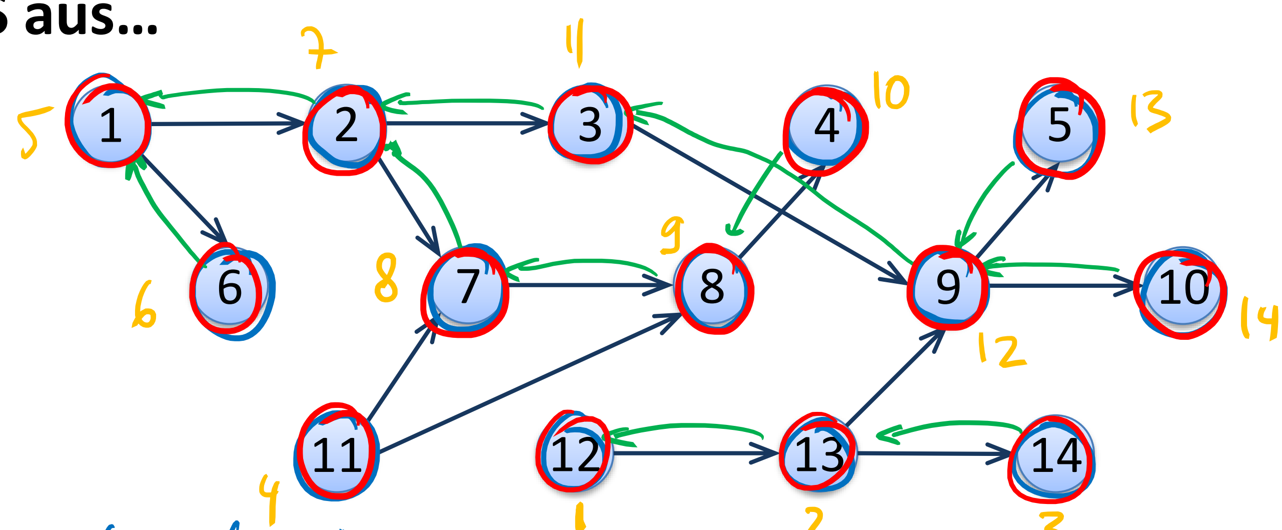
Topologische Sortierung:

- 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

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

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

Topologische Sortierung: Algorithmus

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