ADS 2: Algorithmen und Datenstrukturen

Teil 5

Prof. Dr. Gerhard Heyer

Institut für Informatik
Abteilung Automatische Sprachverarbeitung
Universität Leipzig

09. Mai 2018

[Letzte Aktualisierung: 06/07/2018, 08:42]

Gerichtete azyklische Graphen (DAGs)

Im weiteren steht eg(x) bzw. ag(y) für den Eingangsgrad von x bzw. Ausgangsgrad von y.
Sei $G = (V, E)$ ein gerichteter Graph. G heißt *azyklisch* wenn für jede Kantenfolge k in G gilt: k ist kein Zyklus.

Beobachtung:
Wenn G azyklisch ist, gibt es Knoten $x, y \in V$ mit $\mathrm{eg}(x) = \mathrm{ag}(y) = 0$.

Wir finden solche Knoten wie folgt: Starte mit beliebigem Knoten w und gehe entlang eingehender Kanten "rückwärts". Da der Graph nach Annahme azyklisch und endlich ist, wird kein Knoten auf diesem "Rückweg" zweimal besucht und dieser Prozess terminiert mit einem Knoten v mit $eg(v) = 0$. Analog für ag.

Topologische Sortierung

Eine *topologische Sortierung* eines Digraphen $G = (V, E)$ ist eine bijektive Abbildung

$$s : V \to \{1, \ldots, |V|\}$$

so dass für alle $(u, v) \in E$ gilt:

$$s(u) < s(v) \; .$$

Satz: Ein Digraph G besitzt eine topologische Sortierung, genau dann wenn G azyklisch ist.

Beweis: "$\Rightarrow$"
Sei G zyklisch. Dann ist $(v_0, v_1, ..., v_k)$, $k \geqslant 1$ ein Kreis, also
$s(v_0) < s(v_1) < ... < s(v_k) = s(v_0)$. Widerspruch!

Topologische Sortierung

Beweis: "$\Leftarrow$" durch Induktion über $|V|$.

Anfang: $|V| = 1$, keine Kante, bereits topologisch sortiert.
Schritt: $|V| = n$. Da G azyklisch ist, gibt es ein $v \in V$ mit $\mathrm{eg}(v) = 0$. Der Graph $G' = G - \{v\}$ ist ebenfalls azyklisch und hat $n - 1$ Knoten. Nach Induktionsannahme hat G' eine topologische Sortierung $s : V \setminus \{v\} \to \{1, \ldots, n-1\}$, die wir mit $s(v) = n$ zu einer topologischen Sortierung für G erweitern.

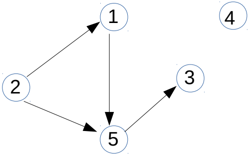
Algorithmus für topologische Sortierung

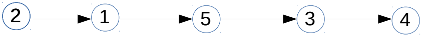
Gegeben ein Graph, $G = (V, E)$, $i = |V|$;
bestimme für jeden Knoten seinen Eingangsgrad und seine Vorgänger;

```
while G hat einen Knoten v mit eg(v) = 0 do
    wähle Knoten ohne Vorgänger v ∈ V (Quelle) und füge
     ihn in die Liste gewählter Quellen an
     (Ergebnisliste);
    entferne v und alle davon ausgehenden Kanten aus G;
    wiederhole diesen Schritt bis alle Knoten aus G
     entfernt sind;
    if keine Quelle mehr vorhanden und Knotenmenge leer then
        Ausgabe "G ist zyklenfrei", topologische Sortierung = Liste der
         gewählten Quellen
    end
    Ausgabe "G ist Zyklus"
end
```

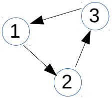
Topologische Sortierung - Beispiel

Schritt	mögl. Quelle	gew. Quelle
1	2, 4	2
2	1, 4	1
3	5, 4	5
4	3, 4	3
5	4	4





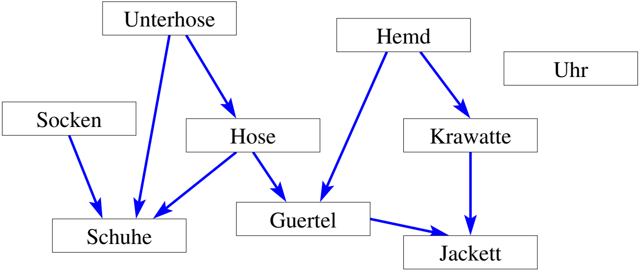
Sortierung 1 2 3 4 5



Keine topologische Sortierung möglich!

... und mal ein ganz praktisches Anwendungsbeispiel

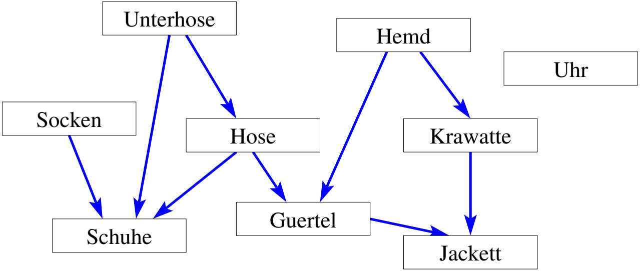
Task scheduling beim Anziehen am Morgen:



Kante (u, v) gibt zeitliche Abfolge vor: u vor v

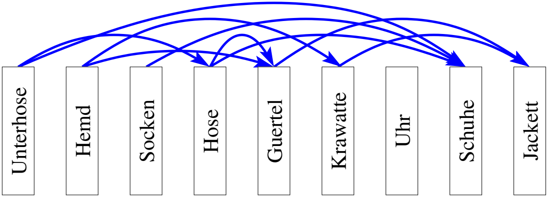
... und mal ein ganz praktisches Anwendungsbeispiel

Task scheduling beim Anziehen am Morgen:



Kante (u, v) gibt zeitliche Abfolge vor: u vor v

Eine topologische Sortierung:



Transitive Hülle

Erreichbarkeit von Knoten

- welche Knoten sind von einem gegebenen Knoten aus erreichbar?
- gibt es Knoten, von denen aus alle anderen erreicht werden können?
- Bestimmung der transitiven Hülle ermöglicht Beantwortung solcher Fragen

Ein Digraph $G^* = (V, E^*)$ heißt *transitive Hülle* eines Digraphen $G = (V, E)$, wenn für alle $u, v \in V$ gilt:

$$(u, v) \in E^* \Leftrightarrow \text{Es gibt einen Weg von } u \text{ nach } v \text{ in } G$$

Eine *reflexive transitive Hülle* ist eine transitive Hülle für die weiter gilt:

$$(u, u) \in E^* \quad \forall u \in V^*$$

Der Algorithmus von Warshall - Idee

Der naive Ansatz würde von jedem Startknoten $u \in V$ eine *Breitensuche* durchführen (mit Komplexität $O(n^3)$ für jeden Knoten).

Der Algorithmus von Warshall basiert auf der Idee, den Graph G^* aus G zu entwickeln, indem schrittweise neue Kanten hinzugenommen werden (und dabei bereits abgeleitete Kantenverbindungen weiter berücksichtigt werden, $\rightarrow$ *dynamische Programmierung*):

Eingabe: Graph G = (V, E) mit V = $0, ..., n-1$
Ausgabe: Graph $G^* = (V, E^*)$

- E^* := E
- für Knoten k = 0, ... , n-1
  - für alle Paare von Knoten (i,j)
    wenn (i,k) und (k,j) Kanten in E^* sind, dann erzeuge neue Kante (i,j) in E^*

Reflexive Transitive Hülle: Warshall-Algorithmus

Direkte Berechnung in $O(n^3)$ für alle Knoten

```
/*Reflexivität*/
boolean[][] A= {...};       //Adjazenzmatrix
for (int i=1; i<=A.length; i++)
    A[i][i]=true;

/*Transitivität*/
for (int k=1; k<=A.length; k++)
    for (int i=1; i<=A.length; i++)
        if (A[i][k])
            for (int j=1; j<=A.length; j++)
                if (A[k][j]) A[i][j]=true;
```

Korrektheit des Warshall-Algorithmus

- Induktionshypothese $P(k)$: Gibt es zu beliebigen Knoten i und j einen Pfad $(i, v_1, v_2, v_3, \ldots, v_{l-1}, j)$ mit inneren Knoten $v_1, v_2, \ldots, v_{l-1} \in \{1, ..., k\}$, so ist nach dem Durchlauf k der äußeren Schleife $A[i][j] = \text{true}$.
- Induktionsanfang, $k = 1$: Falls $A[i][1]$ und $A[1][j]$ gilt, wird in der Schleife mit $k = 1$ auch $A[i][j] = \text{true}$ gesetzt
- Induktionsschluss: Wir nehmen an, daß $P(k-1)$ bereits gezeigt ist, und folgern daraus $P(k)$. Existiere ein Pfad $(i, v_1, \ldots, v_{l-1}, j)$ mit inneren Knoten $v_1, v_2, \ldots, v_{l-1} \in \{1, ..., k\}$. Wenn diese inneren Knoten nicht k enthalten, so folgt mit $P(k-1)$ bereits $A[i][j] = \text{true}$. Anderenfalls gibt es genau einen Index r mit $v_r = k$. Daher sind $(i, v_1, \ldots, v_r)$ und $(v_r, v_{r+1}, \ldots, j)$ Pfade mit inneren Knoten in $\{1, \ldots, k-1\}$. Wegen $P(k-1)$ sind daher $A[i][k] = \text{true}$ und $A[k][j] = \text{true}$ nach Durchlauf der Schleife mit $k-1$. Im Durchlauf k wird daher $A[i][j] = \text{true}$ gesetzt.

Traversierung

Durchlaufen eines Graphen, bei dem jeder vom gewählten Startknoten erreichbare Knoten (bzw. jede Kante) genau 1-mal aufgesucht wird. Jeweils nächster besuchter Knoten hat mindestens einen Nachbarn in der zuvor besuchten Knotenmenge.

Generische Lösungsmöglichkeit für Graphen $G = (V, E)$:

```
FOREACH v in V DO {markiere v als unbearbeitet};
B={s}; // Menge besuchter Knoten, anfangs = Startknoten s
markiere s als bearbeitet;
WHILE es gibt unbearbeiteten Knoten v'
  mit (v,v') in E und v in B
  { B = B + {v'}; markiere v';}
```

Realisierungen unterscheiden sich bezüglich Verwaltung der noch abzuarbeitenden Knotenmenge und Auswahl der jeweils nächsten Kante.

Breiten- und Tiefendurchlauf

Breitendurchlauf (Breadth First Search, BFS)

- ausgehend von Startknoten werden zunächst alle direkt erreichbaren Knoten bearbeitet,
- danach die über mindestens zwei Kanten vom Startknoten erreichbaren Knoten, dann die über drei Kanten usw.
- es werden also erst die Nachbarn besucht, bevor zu den Kindern gegangen wird.
- kann mit FIFO-Datenstruktur für noch zu bearbeitende Knoten realisiert werden.

Tiefendurchlauf (Depth First Search, DFS)

- ausgehend von Startknoten werden zunächst rekursiv alle Kinder (Nachfolger) bearbeitet; erst dann wird zu den (anderen) Nachbarn gegangen.
- kann mit Stack-Datenstruktur für noch zu bearbeitende Knoten realisiert werden.
- Verallgemeinerung der Traversierung von Bäumen.

Breitensuche (BFS)

Bearbeite einen Knoten, der in n Schritten von u erreichbar ist, erst, wenn alle Knoten abgearbeitet wurden, die in $n-1$ Schritten erreichbar sind.

- gerichteter Graph $G = (V, E)$; Startknoten s; Q sei FIFO-Warteschlange.
- zu jedem Knoten u werden der aktuelle Farbwert und der Vorgänger $p[u]$, von dem aus u erreicht wurde, gespeichert.

Breitensuche: Algorithmus

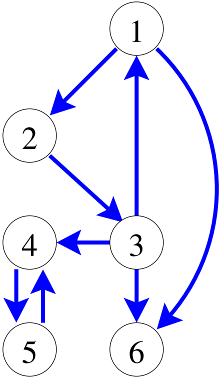
```
BFS(G,s)
  FOREACH v in V DO {farbe(v)=weiss; p[v]=null; }
  farbe[s]=grau; INIT(Q); Q=ENQUEUE(Q,s);
  WHILE NOT (EMPTY(Q)) DO
  {
    v=FRONT(Q);
    FOREACH u in succ(v) DO
    {
      If farbe[u]=weiss THEN
      { farbe[u]=grau; p[u]=v; Q=ENQUEUE(Q,u);}
    }
    DEQUEUE(Q); farbe[v]=schwarz;
  }
```

Farben:

`weiss`=unbearbeitet, `grau`=in Bearbeitung, `schwarz`=bearbeitet

Breitensuche: Beispiel

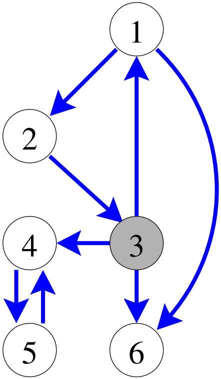
Startknoten $s = 3$



$Q = []$

Breitensuche: Beispiel

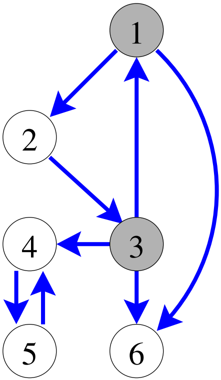
Startknoten $s = 3$



$Q = [3]$

Breitensuche: Beispiel

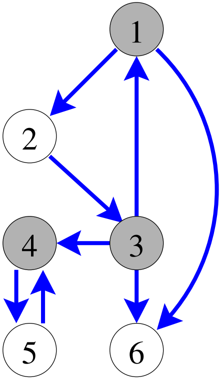
Startknoten $s = 3$



$Q = [3, 1]$

Breitensuche: Beispiel

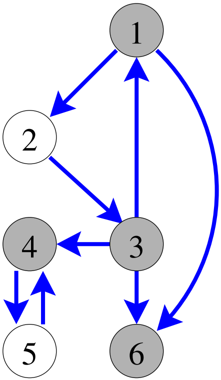
Startknoten $s = 3$



$Q = [3, 1, 4]$

Breitensuche: Beispiel

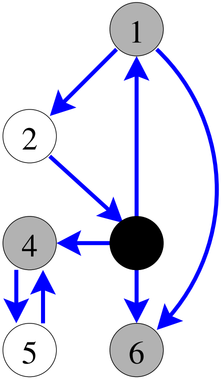
Startknoten $s = 3$



$Q = [3, 1, 4, 6]$

Breitensuche: Beispiel

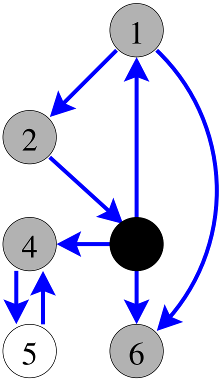
Startknoten $s = 3$



$Q = [1, 4, 6]$

Breitensuche: Beispiel

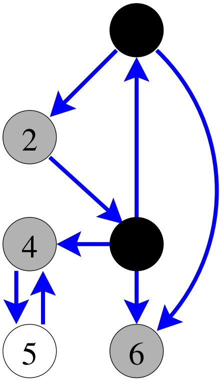
Startknoten $s = 3$



$Q = [1, 4, 6, 2]$

Breitensuche: Beispiel

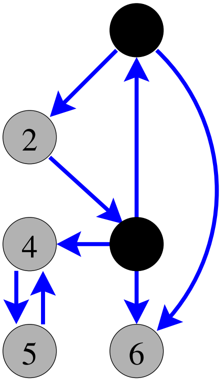
Startknoten $s = 3$



$Q = [4, 6, 2]$

Breitensuche: Beispiel

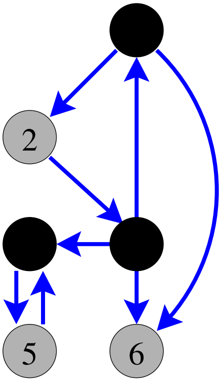
Startknoten $s = 3$



$Q = [4, 6, 2, 5]$

Breitensuche: Beispiel

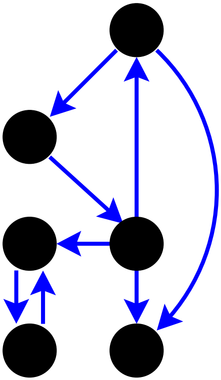
Startknoten $s = 3$



$$Q = [6, 2, 5]$$

Breitensuche: Beispiel

Startknoten $s = 3$



$Q = []$

Tiefensuche (DFS)

- Bearbeite einen Knoten v erst dann, wenn alle seine Kinder bearbeitet sind
- gerichteter Graph $G = (V, E)$;
- zu jedem Knoten v werden gespeichert: der aktuelle Farbwert `farbe[v]`, die Zeitpunkte `in[v]` und `out[v]`, zu denen der Knoten im Rahmen der Tiefensuche erreicht bzw. verlassen wurde und der Vorgänger `p[v]`, von dem aus v erreicht wurde
- die `in`- bzw. `out`-Zeitpunkte ergeben eine Reihenfolge der Knoten analog zur Vor- bzw. Nachordnung bei Bäumen.

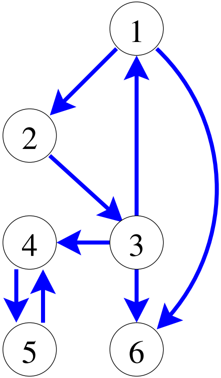
Tiefensuche: Algorithmus

```
DFS(G){
  FOR EACH v in V do { farbe[v]=weiss; p[v]=null; }
  zeit=0
  FOR EACH v in V do { IF farbe[v]=weiss THEN DFS-visit[v] }
}

DFS-visit(G,v){ // rekursiver Teil der Tiefensuche
  farbe[v]=grau; in[v]=zeit; zeit=zeit+1;
  FOR EACH u in succ(v) DO
  { IF farbe[u]=weiss THEN { p[u]=v; DFS-visit[u]; } }
  farbe[v]=schwarz; zeit=zeit+1; out[v]=zeit;
}
```

Tiefensuche: Beispiel

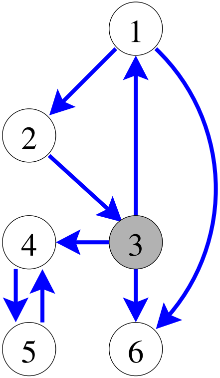
Startknoten $s = 3$



v	in[v]	out[v]
1		
2		
3		
4		
5		
6		

Tiefensuche: Beispiel

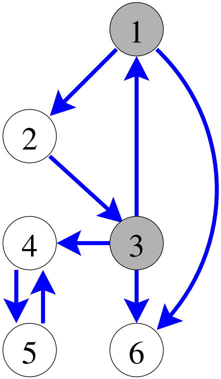
Startknoten $s = 3$



v	in[v]	out[v]
1		
2		
3	1	
4		
5		
6		

Tiefensuche: Beispiel

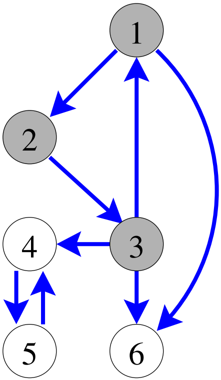
Startknoten $s = 3$



v	in[v]	out[v]
1	2	
2		
3	1	
4		
5		
6		

Tiefensuche: Beispiel

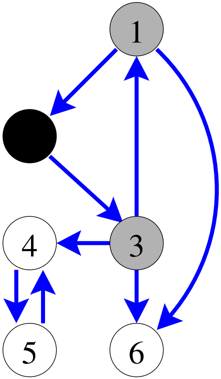
Startknoten $s = 3$



v	in[v]	out[v]
1	2	
2	3	
3	1	
4		
5		
6		

Tiefensuche: Beispiel

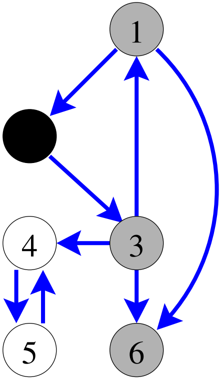
Startknoten $s = 3$



v	in[v]	out[v]
1	2	
2	3	4
3	1	
4		
5		
6		

Tiefensuche: Beispiel

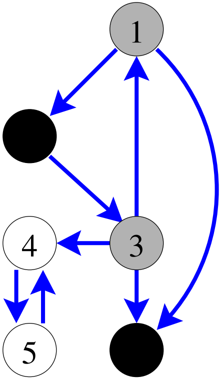
Startknoten $s = 3$



v	in[v]	out[v]
1	2	
2	3	4
3	1	
4		
5		
6	5	

Tiefensuche: Beispiel

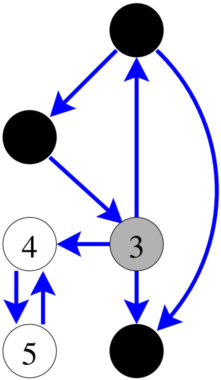
Startknoten $s = 3$



v	in[v]	out[v]
1	2	
2	3	4
3	1	
4		
5		
6	5	6

Tiefensuche: Beispiel

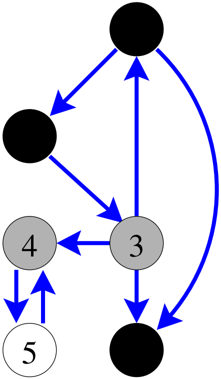
Startknoten $s = 3$



v	in[v]	out[v]
1	2	7
2	3	4
3	1	
4		
5		
6	5	6

Tiefensuche: Beispiel

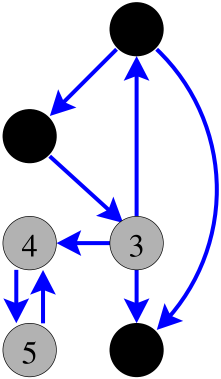
Startknoten $s = 3$



v	in[v]	out[v]
1	2	7
2	3	4
3	1	
4	8	
5		
6	5	6

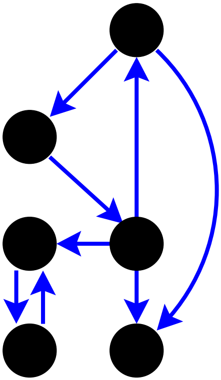
Tiefensuche: Beispiel

Startknoten $s = 3$



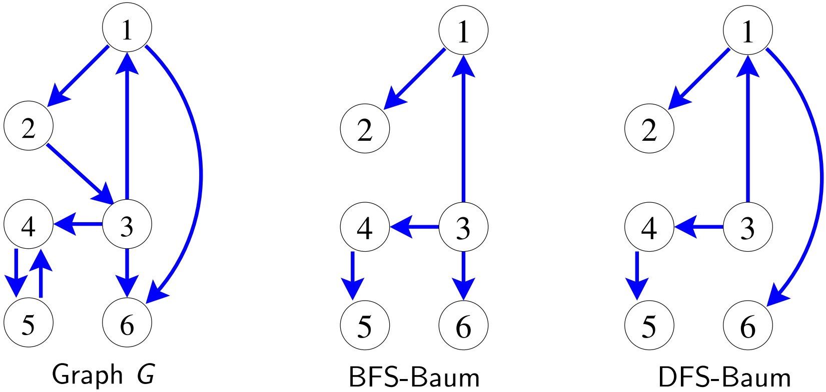
v	in[v]	out[v]
1	2	7
2	3	4
3	1	
4	8	
5	9	
6	5	6

Startknoten $s = 3$



v	in[v]	out[v]
1	2	7
2	3	4
3	1	12
4	8	11
5	9	10
6	5	6

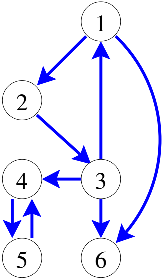
Bäume aus Breiten- und Tiefensuche



Starke Zusammenhangskomponenten

Ein gerichteter Graph $G = (V, E)$ heißt *stark zusammenhängend*, wenn für **alle** $u, v \in V$ gilt: Es gibt einen Pfad von u nach v.

Eine *starke Zusammenhangskomponente* von G ist ein maximaler stark zusammenhängender Teilgraph von G.

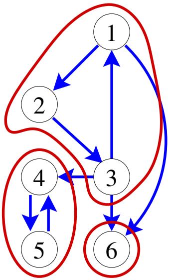


Jeder Knoten eines Graphen ist in genau einer starken Zusammenhangskomponente enthalten.

Starke Zusammenhangskomponenten

Ein gerichteter Graph $G = (V, E)$ heißt *stark zusammenhängend*, wenn für **alle** $u, v \in V$ gilt: Es gibt einen Pfad von u nach v.

Eine *starke Zusammenhangskomponente* von G ist ein maximaler stark zusammenhängender Teilgraph von G.



Jeder Knoten eines Graphen ist in genau einer starken Zusammenhangskomponente enthalten.

gegenseitige Erreichbarkeit = äquivalenzrelation

Sei $G = (V, E)$ ein gerichteter Graph. Sind Knoten $u, v \in V$ gegenseitig erreichbar, schreiben wir $u \sim v$. Die so definierte Relation $\sim$ auf V ist eine äquivalenzrelation also

- symmetrisch
- transitiv und
- reflexiv

Die Knotenmengen der starken Zusammenhangskomponenten von G sind die äquivalenzklassen von $\sim$.

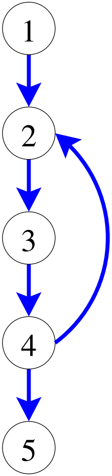
Starke Zus.hang-Komponenten durch Tiefensuche

- Führe Tiefensuche (DFS) auf G aus.
- Für jeden Knoten `v` berechne dabei `l[v]` $=$ Index des "ersten" Knotens, der von v erreichbar ist in der durch `in[]` gegebenen Reihenfolge.
- Wenn alle Kindsknoten von `v` abgearbeitet sind und `l[v]=in[v]`, ist `v` die "Wurzel" einer starken Zusammenhangskomponente. Deren Knoten werden dann gleich ausgegeben und nicht mehr weiter betrachtet (denn jeder Knoten ist in nur einer Komponente).

Algorithmus von Tarjan (1972)

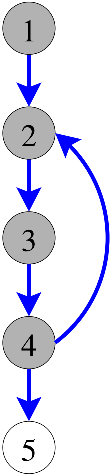
```
DFS-visit(G){
  FOR EACH v in V do {farbe[v]=weiss; p[v]=null;} zeit=0
  FOR EACH v in V do {IF farbe[v]=weiss THEN Tarjan-visit[v]}}
Tarjan-visit(G,v){
  farbe[v]=grau; zeit=zeit+1; in[v]=zeit; l[v]=zeit;
  PUSH(S,v);
  FOR EACH u in succ(v) DO {
    IF farbe[u]=weiss THEN {
      Tarjan-visit(G,u);
      l[v]=min(l[v],l[u]);}
    ELSEIF u in S THEN { l[v]=min(l[v],in[u]); }}
  IF (l[v]=in[v]) {
    Ausgabe("starke ZshK":)
    DO  { u=TOP(S); Ausgabe(u); POP(S); }
    UNTIL u=v;}
  farbe[v]=schwarz; zeit=zeit+1;}
```

Tarjan: Beispiel



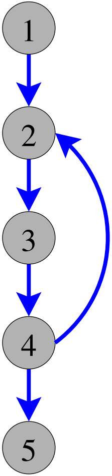
v	in[v]	l[v]
1	1	1
2	2	2
3	3	3
4	4	2
5	5	

Tarjan: Beispiel



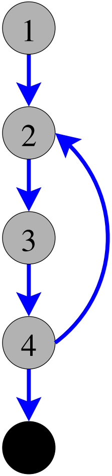
v	in[v]	l[v]
1	1	1
2	2	2
3	3	3
4	4	2
5	5	

Tarjan: Beispiel



v	in[v]	l[v]
1	1	1
2	2	2
3	3	3
4	4	2
5	5	5

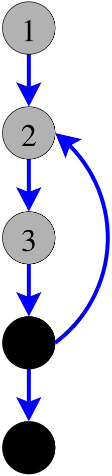
Tarjan: Beispiel



v	in[v]	l[v]
1	1	1
2	2	2
3	3	3
4	4	2
5	5	5

Zshk: 5

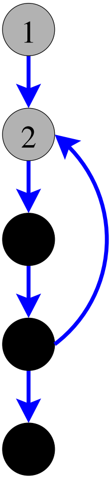
Tarjan: Beispiel



v	in[v]	l[v]
1	1	1
2	2	2
3	3	3
4	4	2
5	5	5

Zshk: 5

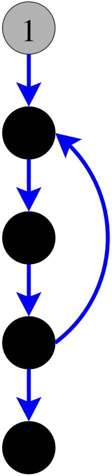
Tarjan: Beispiel



v	in[v]	l[v]
1	1	1
2	2	2
3	3	2
4	4	2
5	5	5

Zshk: 5

Tarjan: Beispiel

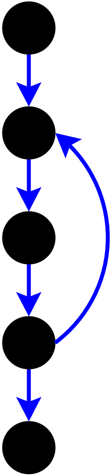


v	in[v]	l[v]
1	1	1
2	2	2
3	3	2
4	4	2
5	5	5

Zshk: 5

Zshk: 2,3,4

Tarjan: Beispiel



v	in[v]	l[v]
1	1	1
2	2	2
3	3	2
4	4	2
5	5	5

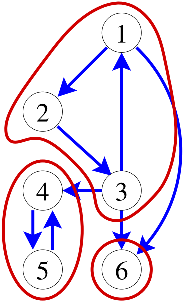
Zshk: 5

Zshk: 2,3,4

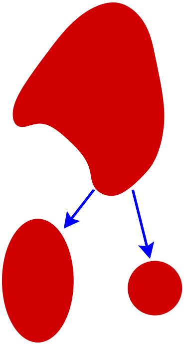
Zshk: 1

Komponentengraph G^*

Fasse alle Knoten jeweils einer starken Zusammenhangskomponente zu einem einzigen Knoten zusammen. Kante von Komponente A nach Komponente B, wenn es $u \in A$ und $v \in B$ gibt, so daß (u, v) eine Kante in G ist.



Graph G


Komponentengraph G^*

Erlaubt z.B. schnellere Berechnung der transitiven Hülle von G.

Zentralität

Gegeben ein Graph mit $V = (v_1, \ldots, v_n)$ Knoten. Wie können wir verschiedene Knoten nach ihrer „Zentralität" unterscheiden?

1. Grad: Zentrale Knoten haben mehr Nachbarn als weniger zentrale. Algorithmus: einfaches Zählen der Nachbarn der Knoten
2. Exzentrizität: Derjenige Knoten mit dem geringsten Abstand zu dem am weitesten entfernten Knoten
   Algorithmus: Zählen des Abstands und Bildung des Kehrwerts
3. Distanzsumme: Der Knoten, für den die Summe der Distanzen minimal ist, hat minimalen Durchschnittsabstand zu allen Knoten
   Algorithmus: Aufsummieren der Distanzen und Bildung des Kehrwerts

Zentralität

1. Shortest Path Betweenness: Derjenige Knoten, der an den meisten Kommunikationswegen zwischen Paaren von Knoten beteiligt ist.
   Algorithmus: Berechne für jeden Knoten seinen Anteil in der Menge der kürzesten Pfade für alle Knotenpaare
   (Für jeden Knoten $v \in V$ lassen sich alle Distanzen d(s, t) im Graphen V sowie die Anzahlen der kürzesten Wege zwischen allen Paaren von Knoten mit Hilfe der Breitensuche berechnen. Durch Aufsummieren erhält man die Shortest Path Betweenness für einen Knoten v bzw. für alle Knoten)
2. PageRank: Derjenige Knoten mit der höchsten Anzahl von „wichtigen" Knoten als Indegree
   Algorithmus: Berechne für jeden Knoten ein Gewicht aus der Menge der auf ihn zeigenden Knoten und bestimme daraus (rekursiv) den PageRank für jeden Knoten.