ADS 2: Algorithmen und Datenstrukturen

Teil 3

Prof. Dr. Gerhard Heyer

Institut für Informatik
Abteilung Automatische Sprachverarbeitung
Universität Leipzig

17. April 2019

[Letzte Aktualisierung: 23/05/2019, 09:51]

Gerichtete azyklische Graphen (DAGs)

Wie bisher steht eg(x) bzw. ag(y) für den Eingangsgrad von x bzw. Ausgangsgrad von y.
Sei $G = (V, E)$ ein gerichteter Graph (im folgenden kurz *Digraph*). 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

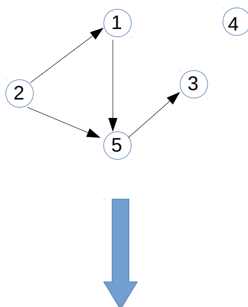
$$s : V \to \{1, \dots, |V|\}$$

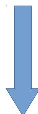
so dass für alle $(u, v) \in E$ gilt:

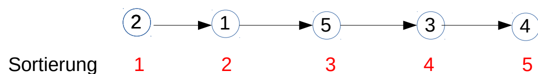
$$s(u) < s(v) \; .$$

Topologische Sortierung - Beispiel

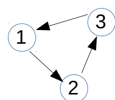
Schritt	mögl. Quelle	gew. Quelle
1	2, 4	2
2	1, 4	1
3	5, 4	5
4	3, 4	3
5	4	4







Sortierung 1 2 3 4 5



Keine topologische Sortierung möglich!

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 $v \in V$ (*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

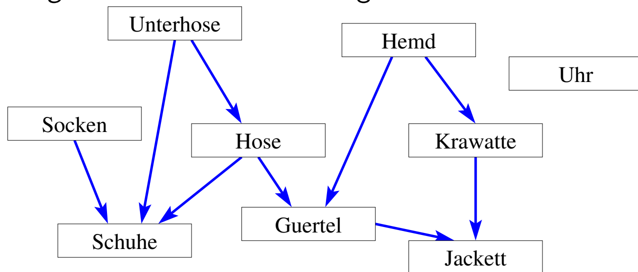
Satz: Ein Digraph G besitzt eine topologische Sortierung, genau dann wenn G azyklisch ist.

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!

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 $s : V \setminus \{v\} \rightarrow \{1, \dots, n-1\}$, die wir mit $s(v) = n$ zu einer topologischen Sortierung für G erweitern.

... und mal ein ganz praktisches Anwendungsbeispiel

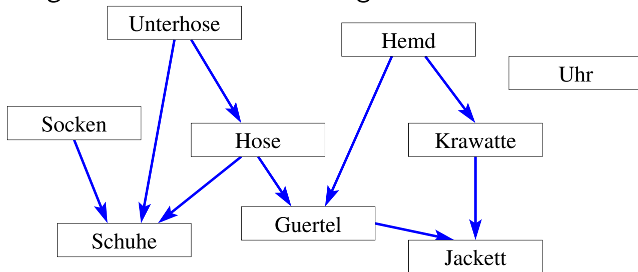
Task scheduling beim Anziehen am Morgen:



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

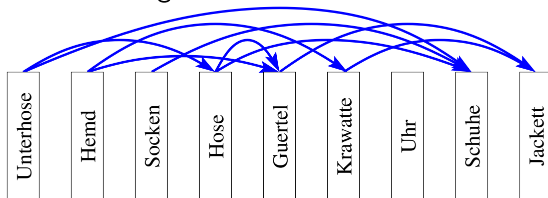
... und mal ein ganz praktisches Anwendungsbeispiel

Task scheduling beim Anziehen am Morgen:



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

Eine topologische Sortierung:



Transitive Hülle

<u>Erreichbarkeit von Knoten</u>

- 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

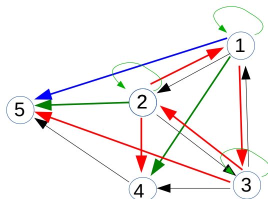
Ein Digraph $G^* = (V, E^*)$ heißt *transitive Hülle* eines Digraphen $G = (V, E)$, wenn für alle $u, v \in V$ gilt:

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

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

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

Transitive Hülle - Beispiel



Hülle mit Pfad der Länge 2

Hülle mit Pfad der Länge 3

Hülle mit Pfad der Länge 4

j (columns), **i** (rows)

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

j (columns), **i** (rows)

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

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. 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 v; Q sei FIFO-Warteschlange.
- zu jedem Knoten u werden der aktuelle Farbwert und der Vorgänger $p[u]$, von dem aus u erreicht wurde, gespeichert.

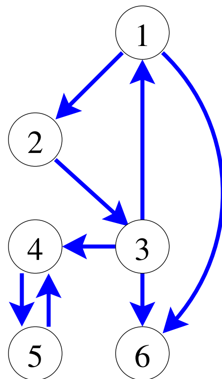
Breitensuche: Algorithmus

```
BFS(G){
  FOR EACH v in V do { farbe[v]=weiss; p[v]=null; }
  FOR EACH v in V do { IF farbe[v]=weiss THEN BFS-visit(G,v) }
}
BFS-visit(G,v){
  farbe[v]=grau; INIT(Q); Q=ENQUEUE(Q,v);
  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**

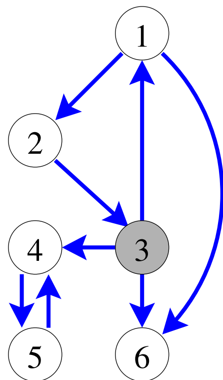
Breitensuche: Beispiel

Startknoten $v = 3$



$Q = []$

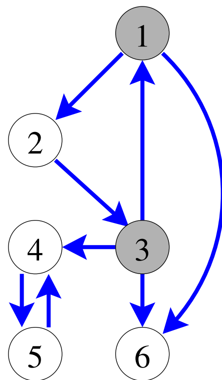
Startknoten $v = 3$



$Q = [3]$

Breitensuche: Beispiel

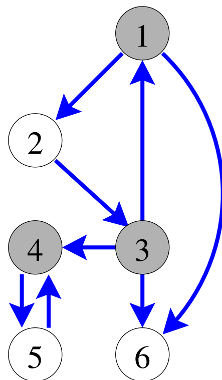
Startknoten $v = 3$



$Q = [3, 1]$

Breitensuche: Beispiel

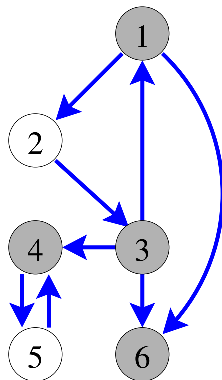
Startknoten $v = 3$



$Q = [3, 1, 4]$

Breitensuche: Beispiel

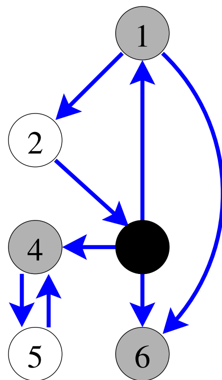
Startknoten $v = 3$



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

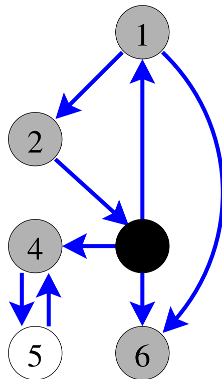
Breitensuche: Beispiel

Startknoten $v = 3$



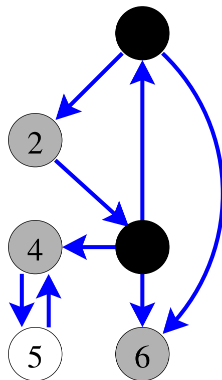
$Q = [1, 4, 6]$

Startknoten $v = 3$



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

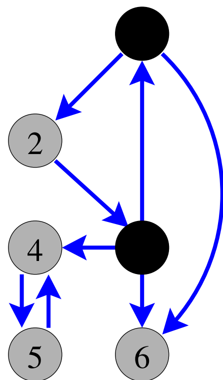
Startknoten $v = 3$



$Q = [4, 6, 2]$

Breitensuche: Beispiel

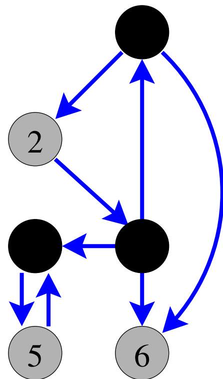
Startknoten $v = 3$



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

Breitensuche: Beispiel

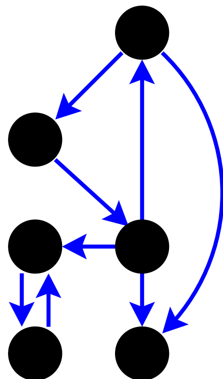
Startknoten $v = 3$



$Q = [6, 2, 5]$

Breitensuche: Beispiel

Startknoten $v = 3$



$Q = []$

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.

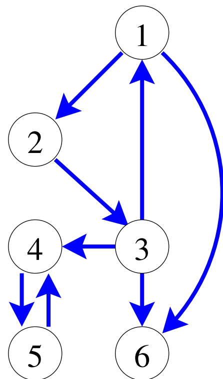
Tiefensuche: Algorithmus

```
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(G,v) }
}

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

Tiefensuche: Beispiel

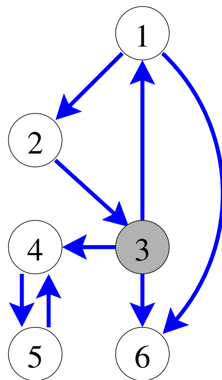
Startknoten $s = 3$



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

Tiefensuche: Beispiel

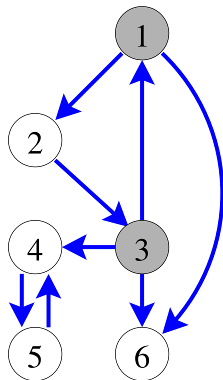
Startknoten $s = 3$



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

Tiefensuche: Beispiel

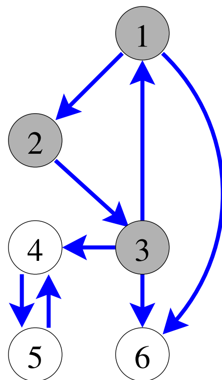
Startknoten $s = 3$



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

Tiefensuche: Beispiel

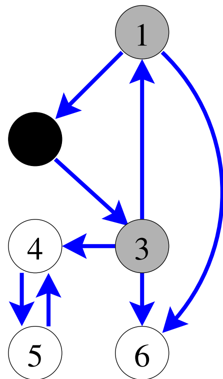
Startknoten $s = 3$



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

Tiefensuche: Beispiel

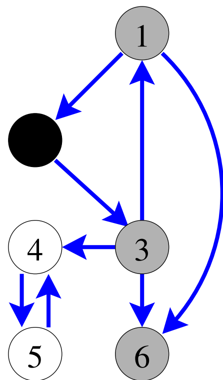
Startknoten $s = 3$



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

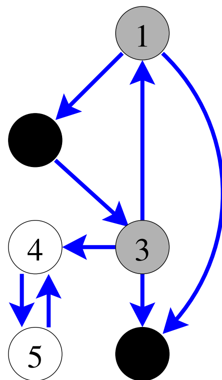
Tiefensuche: Beispiel

Startknoten $s = 3$



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

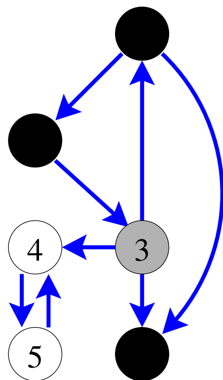
Startknoten $s = 3$



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

Tiefensuche: Beispiel

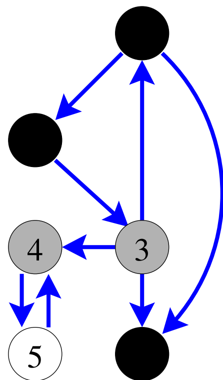
Startknoten $s = 3$



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

Tiefensuche: Beispiel

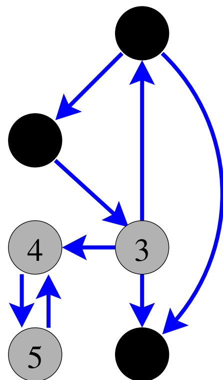
Startknoten $s = 3$



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

Tiefensuche: Beispiel

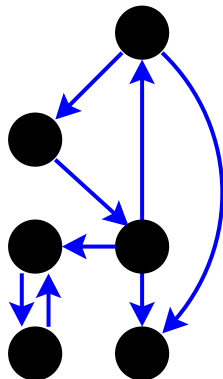
Startknoten $s = 3$



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

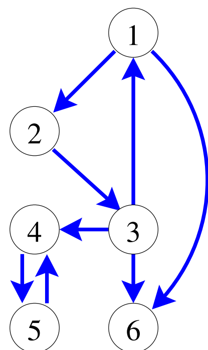
Tiefensuche: Beispiel

Startknoten $s = 3$

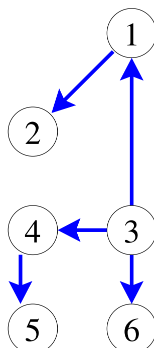


v	in[v]	out[v]
1	2	7
2	3	4
3	1	12
4	8	11
5	9	10
6	5	6

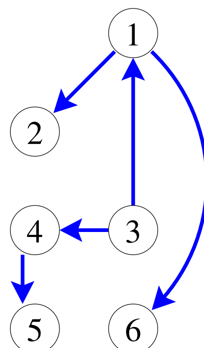
Bäume aus Breiten- und Tiefensuche



Graph G



BFS-Baum

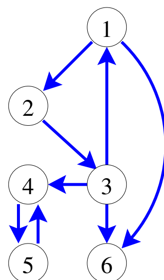


DFS-Baum

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.

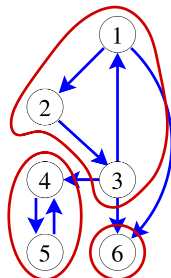


Jeder Knoten eines Graphen ist in genau einer starken Zusammenhangskomponente enthalten.

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

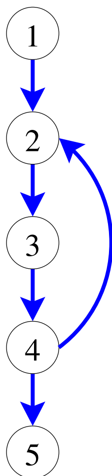
- symmetrisch
- transitiv und
- reflexiv

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

Starke Zus.hang-Komponenten durch Tiefensuche

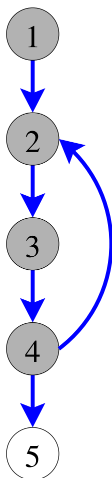
- Führe Tiefensuche (DFS) auf G aus.
- Für jeden Knoten `v` berechne dabei `l[v]` = Index des "ersten" Knotens, der von v erreichbar ist in der durch `in[]` gegebenen Reihenfolge.
- Wenn alle Kindsknoten von `v` abgearbeitet sind und `l[v]=in[v]`, ist `v` die "Wurzel" einer starken Zusammenhangskomponente.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

Beispiel (Algorithmus von Tarjan)



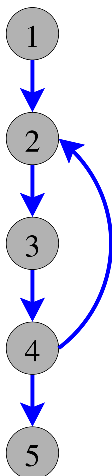
v	in[v]	l[v]
1	1	1
2	2	2
3	3	3
4	4	2
5	5	

Beispiel (Algorithmus von Tarjan)



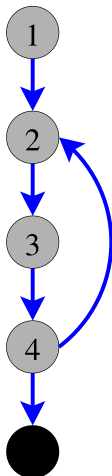
v	in[v]	l[v]
1	1	1
2	2	2
3	3	3
4	4	2
5	5	

Beispiel (Algorithmus von Tarjan)



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

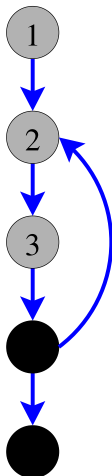
Beispiel (Algorithmus von Tarjan)



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

Zshk: 5

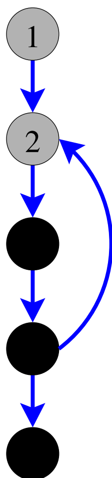
Beispiel (Algorithmus von Tarjan)



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

Zshk: 5

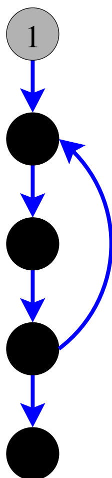
Beispiel (Algorithmus von Tarjan)



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

Zshk: 5

Beispiel (Algorithmus von Tarjan)

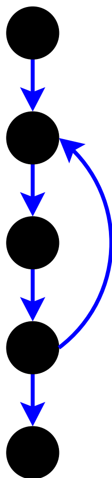


v	in[v]	l[v]
1	1	1
2	2	2
3	3	2
4	4	2
5	5	5

Zshk: 5

Zshk: 2,3,4

Beispiel (Algorithmus von Tarjan)

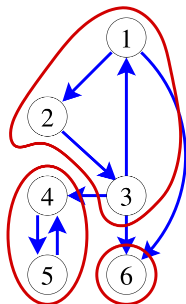


v	in[v]	l[v]
1	1	1
2	2	2
3	3	2
4	4	2
5	5	5

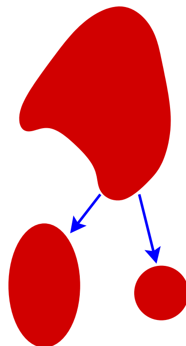
Zshk: 5
Zshk: 2,3,4
Zshk: 1

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

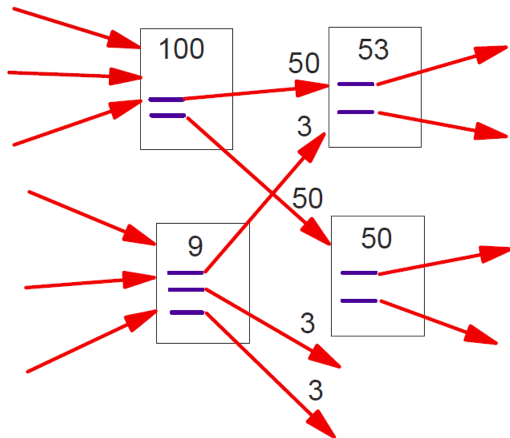
Zentralität

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.

In einem gerichteten Graphen $G = (V,E)$ sind die Knoten miteinander *verlinkt*. Wir nutzen diese Linkstruktur, um die Zentralität von Knoten in dem Graphen zu berechnen.

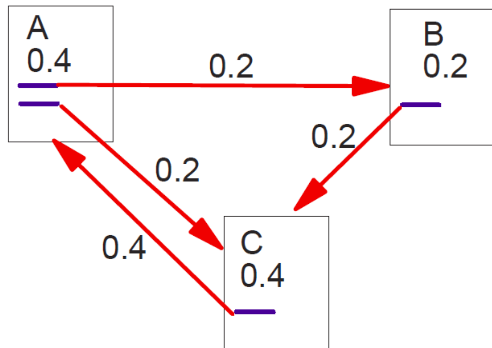
Idee:

- Jeder Knoten hat einen PageRank und verweist auf eine bestimmte Anzahl anderer Knoten.
- Jeder der verlinkten Knoten hat selbst wiederum einen Pagerank.
- Die Links haben ein Gewicht in Abhängigkeit von dem PageRank des Knotens: Je höher der PageRank eines Knotens ist, desto höher ist das Gewicht seiner ausgehenden Kanten.
- Der PageRank eines Knotens verteilt sich gleichmäßig auf alle ausgehenden Kanten.



Quelle: Lawrence Page, Sergey Brin, Rajeev Motwani, Terry Winograd, The PageRank Citation Ranking: Bringing Order to the Web", Technical Report, January 29th, 1998. Stanford InfoLab.

PageRank Beispiel: Verteilung von Gewichten



Quelle: Lawrence Page, Sergey Brin, Rajeev Motwani, Terry Winograd, The PageRank Citation Ranking: Bringing Order to the Web", Technical Report, January 29th, 1998. Stanford InfoLab.

PageRank: Vereinfachte Berechnung

Es sei G = (V, E) ein gerichteter und gewichteter Graph von verlinkten Webseiten mit

- $u, v \in V$
- F_v der Menge der Seiten auf die v zeigt
- B_u der Menge der Seiten, die auf u zeigen
- c ein Normalisierungsfaktor (bezogen auf die Anzahl der betrachteten Webseiten)
- $N_v = |F_v|$

$$R(u) = c \sum_{v \in B_u} \frac{R(v)}{N_v}$$

PageRank: Kommentar

- Pagerank ist gleichmäßig auf die vorwärts verlinken Seiten verteilt
- die Berechnung erfolgt rekursiv
- Berechnung des Pagerank kann bei beliebigen Knoten begonnen werden, bis der Wert konvergiert

Aber:

- Wenn eine Seite auf zwei Seiten zeigt, die nur auf sich gegenseitig verlinken, entsteht eine Senke (weil nicht auf weitere Seiten verlinkt wird)

Deshalb angepasste Formel mit einem *Dämpfungsfaktor d* entsprechend dem *Random-Surfer-Modell*

$$R(u) = (1 - d) + d \sum_{v \in B_u} \frac{R(v)}{N_v}$$

PageRank: Dämpfungsfaktor d

- Dem Random-Surfer-Modell liegt die Annahme zugrunde, dass ein Surfer, der sich durchs Netz klickt, nicht in der Schleife verharrt, sondern auf eine andere Seite geht.
- Der Dämpfungsfaktor d modelliert das Nutzerverhalten, mit welcher Wahrscheinlichkeit ein Nutzer von einer Seite zu anderen Seiten wechselt. Dieser Wert ist empirisch zu bestimmen, in Brin und Page: *The Anatomy of a Large-Scale Hypertextual Web Search Engine* (1998) wird er mit 0,85 angegeben.
- Der Ausdruck 1 - d repräsentiert die Wahrscheinlichkeit, bei einem Random Walk im Webgraphen ausgehend von einer beliebigen Seite auf dem Knoten u zu landen
- Der Ausdruck $d \sum_{v \in B_u} \frac{R(v)}{N_v}$ repräsentiert die Wahrscheinlichkeit, ausgehend vom Knoten u, beim Knoten v zu landen. (d gibt also den Anteil von Page-Rank an, den eine Seite immer zu den von ihr verlinkten Seiten abgibt).

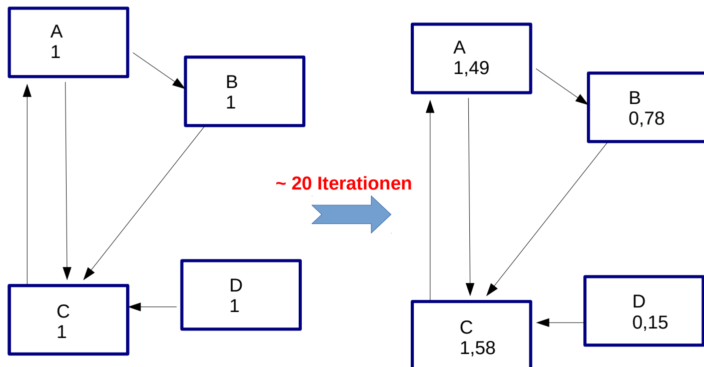
PageRank: Berechnung

Jede Webseite weist jeder von ihr verlinkten Seite einen bestimmten Anteil ihres eigenen Page-Ranks zu. Sei $|S_i|$ diese Anzahl der ausgehenden Links für eine Seite S_i.
Der Page-Rank einer Seite ist dann durch folgendes Gleichungssystem definiert und kann iterativ berechnet werden (für n von 1 bis i):
R(S_i) := (1-d) + d (R(S_1) / $|S_1|$ + ... + R(S_n) / $|S_n|$)) (mit $|S_i| > 0$)

Verfahren:

- initialisiere jede Seite mit dem initialen Wert (1 - d)
- wähle eine beliebige Seite
- iteriere bis $|R(S_i)|_{j+1} - |R(S_i)|_j$ minimal
  - berechne (mit Breitensuche) alle Seiten, auf die S_i verlinkt
  - berechne für jede verlinkte Seite ihren Page-Rank R

PageRank: Beispiel



PageRank: Kurze Historie

Versionen und Weiterentwicklungen:

- 1998: Veröffentlichung des PageRank-Algorithmus von Brin und Page
- 2010: Rational Surfer Modell
- 2013: neuer Algorithmus - Hummingbird