UNIVERSITÄT
LEIPZIG

Algorithmen und Datenstrukturen II

Vorlesung 2wee

Leipzig, 09.04.2024
Peter F. Stadler & Thomas Gatter & Ronny Lorenz

DAGS (DIRECTED ACYCLIC GRAPHS)

Gerichtete azyklische Graphen (DAGs) I

Definition

Sei $G = (V, E)$ ein gerichteter Graph. G heißt **azyklisch** wenn für jede Kantenfolge k in G gilt: k ist kein Zyklus.

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

Gerichtete azyklische Graphen (DAGs) II

Beweis

Annahme: $\text{ag}(y) > 0$ für alle $y \in V$.

Dann gibt es zu jedem $\ell \in \mathbb{N}$ eine Kantenfolge der Länge ℓ, denn jede Kantenfolge der Länge $\ell - 1$ kann durch Hinzunahme eines Nachfolgers des letzten Knotens auf Länge ℓ gebracht werden. In einer Kantenfolge der Länge $\ell > |V|$ muss mindestens ein Knoten mehrfach auftreten, so dass die Kantenfolge einen Zyklus enthält. Widerspruch, denn G ist azyklisch.
(Beweis für Eingangsgrad analog) □

Gerichtete azyklische Graphen (DAGs) III

siehe auch im englischen Wikipedia:

- Directed acyclic graph *(Wikipedia)*
  Der Artikel ist sehr tiefgehend und betrifft die gesamte Vorlesung 2.

Topologische Sortierung I

Definition

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:

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

Topologische Sortierung II

Theorem

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

Topologische Sortierung III

Beweis

"$\Rightarrow$" klar.

"$\Leftarrow$" *durch Induktion über* $|V|$.
Induktionsanfang: $|V| = 1$, keine Kante, bereits topologisch sortiert.
Induktionsschritt: $|V| = n$. Da G azyklisch ist, $v \in V$ mit $\text{ag}(v) = 0$. 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. $\square$

Algorithmus für topologische Sortierung

Algorithmus (topologische Sortierung)

input : Graph $G = (V, E)$
output : topologischen Sortierung *s* von *G*

```
i = |V|;

while (G hat einen Knoten v mit eg(v)==0) do
    s[v] = i ;
    G = G - v ;
    i = i - 1 ;

if (i == 0) then
    Ausgabe(G ist zyklenfrei) ;

else
    Ausgabe(G hat Zyklus) ;
```

Algorithmus für topologische Sortierung

Der Algorithmus testet auf Zyklenfreiheit und damit auf die Existenz einer topologischen Sortierung.

Wie bestimmt eine topologischen Sortierung s für $G = (V, E)$ mit Hilfe des Algorithmus?

- **Wichtig:** (Neu-)Bestimmung des Ausgangsgrades aufwendig
- *Effizienter:* aktuellen Ausgangsgrad zu jedem Knoten speichern

Animationen: Topologische Sortierung

Animation topologisches Sortieren - Indegree (www.cs.usfca.edu) *(Link)*

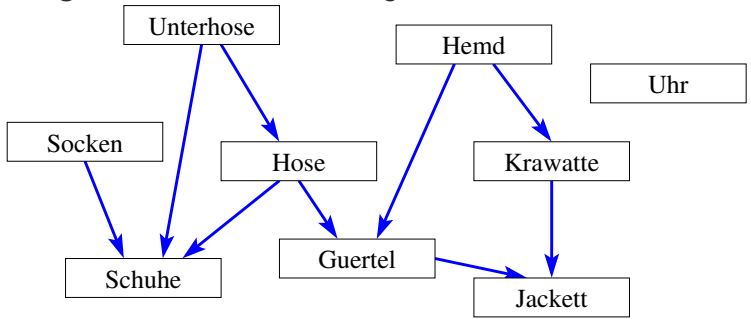
- die Schritte sind expliziter, indem ein stack vorgehalten wird

Animation topologisches Sortieren - DFS (www.cs.usfca.edu) *(Link)*

- man kann auch Tiefensuche (später in dieser VL-Einheit) nutzen

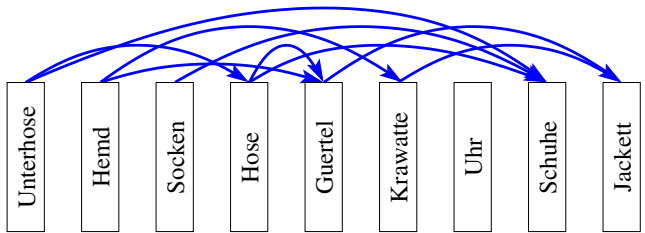
Anwendungsbeispiel: Topologische Sortierung

Task scheduling beim Anziehen am Morgen:



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

Anwendungsbeispiel: Topologische Sortierung



HÜLLEN

Transitive Hülle I

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

Definition

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

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

Transitive Hülle II

Reflexivität: die Hülle ist *reflexiv*, wenn wir zulassen das $u = v$ in $u, v \in V$.
Das bedeutet das eine Schleife vom Knoten u zu sich selbst (weil $u = v$) vorhanden ist. Die Hülle ist *nicht reflexiv*, wenn wir fordern, dass Kanten (u, v) nur existieren für $u \neq v$.

Animation: Floyd-Warshall Algorithmus

Animation Floyd-Warshall Algorithmus (www.cs.usfca.edu) *(Link)*

Dieser liefert für jedes Knotenpaar (u, v) den günstigsten Weg (summiert über Kantengewichte), oder ∞ falls v nicht von u erreichbar ist.

Transitive Hülle: naiver Ansatz

Naiver Algorithmus zur Berechnung von Kanten der reflexiven transitiven Hülle

Algorithmus (naiver Ansatz)

```
boolean[][] A = {...} ;     // Graph als Adjazenzmatrix, 1-basiert
for (int i=1; i<=A.length; i++) do                                  // Teil 1
    A[i][i] = 1 ;     // Selbsterreichbarkeit bedeutet Schleifen (Reflexivität)
for (int i=1; i<=A.length; i++) do                                  // Teil 2
    for (int j=1; j<=A.length; j++) do
        if (A[i][j] == 1) then
            for (int k=1; k<=A.length; k++) do
                if (A[j][k] == 1) then
                    A[i][k] = 1 ;
```

Transitive Hülle: naiver Ansatz

- Es werden in *Teil 2* allgemein nur Pfade der Länge 2 bestimmt.
- Deshalb ergibt sich als **naiver Ansatz** zu Berechnung der vollständigen transitiven Hülle:

Algorithmus (naiver Ansatz) - Fortsetzung

Iteriere *Teil 2* solange, bis sich A nicht mehr ändert.

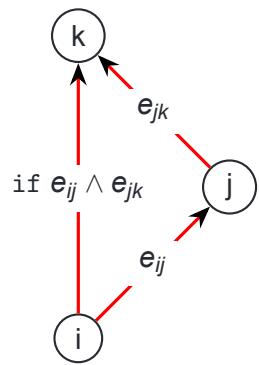
- Komplexität jeder Iteration: $O(n^3)$

Transitive Hülle: Warshall-Algorithmus

Eine einfache Modifikation erlaubt die direkte Berechnung in $O(n^3)$, ohne Iteration:

Algorithmus (Modifikation des naiven Ansatzes)

```
boolean[][] A = {...} ;      // Adjazenzmatrix, 1-basiert
for (int i=1; i<=A.length; i++) do
  | A[i][i] = 1 ;
// Man beachte die veränderten Indize
for (int j=1; j<=A.length; j++) do
  | for (int i=1; i<=A.length; i++) do
  |   | if (A[i][j] == 1) then
  |   |   | for (int k=1; k<=A.length; k++) do
  |   |   |   | if (A[j][k] == 1) then
  |   |   |   |   | A[i][k] = 1 ;
```



Transitive Hülle: Warshall-Algorithmus

Algorithmus (Modifikation) - Erklärung

Falls e_{ij} existiert (`A[i][j] == TRUE`),
dann teste alle k:
falls e_{jk} existiert (`A[j][k] == 1`),
dann gibt es einen Pfad $i \rightsquigarrow k$ (zB über j)
und wir setzen e_{ik} (`A[i][k] = 1`).

e_{ij}, e_{jk}, e_{ik} sind Kanten (edges).

Korrektheit des Warshall-Algorithmus

Beweis (1/2)

- **Induktionshypothese** $P(j)$**:** gibt es zu beliebigen Knoten i und k einen Pfad $(i, v_1, v_2, v_3, \ldots, v_{l-1}, k)$ mit inneren Knoten $v_1, v_2, \ldots, v_{l-1} \in \{1, ...j\}$, so ist nach dem Durchlauf j der äußeren Schleife $A[i][k] = 1$.
- **Induktionsanfang,** $j = 1$**:** Falls $A[i][1]$ und $A[1][k]$ gilt, wird in der Schleife mit $j = 1$ auch $A[i][k] = 1$ gesetzt.

Korrektheit des Warshall-Algorithmus

Beweis (2/2)

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

TRAVERSIERUNG

Traversierung I

Definition

Traversierung ist das Durchlaufen eines Graphen, bei dem jeder vom gewählten Startknoten erreichbare Knoten (bzw. jede Kante) genau 1-mal aufgesucht wird. Der jeweils nächste besuchte Knoten hat mindestens einen Nachbarn in der zuvor besuchten Knotenmenge.

Traversierung II

Algorithmus (Generische Lösung)

input : Graph $G = (V, E)$
output : Reihenfolge der besuchten Knoten B

---

for *(jeden Knoten v aus V)* **do**
  markiere v als unbearbeitet

$B = \{s\}$; `// Menge besuchter Knoten, anfangs nur Startknoten s`
markiere s als bearbeitet

while *es gibt unbearbeitete Knoten* v' **do**

  if *(*$(v, v') \in E$*) and (*$v \in B$*)* **then**

    $B = B + \{v'\}$;
    markiere v' als bearbeitet

Traversierung III

Realisierungen unterscheiden sich bezüglich Verwaltung der noch abzuarbeitenden Knotenmenge (siehe: *es gibt unbearbeitete Knoten* v' im Algorithmus) und Auswahl der jeweils nächsten Kante (siehe: $(v, v') \in E$ im Algorithmus).

$\rightarrow$Breitensuche
$\rightarrow$Tiefensuche

Breiten- und Tiefendurchlauf I

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.

Breiten- und Tiefendurchlauf II

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.
- p-Werte liefern nach Abarbeitung für zusammenhängende Graphen einen Spannbaum.

Der Algorithmus auf den kommenden Slides nutzt explizit eine Farbe "grau" um "in Bearbeitung" zu kennzeichnen. Das ist nicht unbedingt nötig:
Eine Visuallisierung von www.cs.usfca.edu: (Link)

Algorithmus BFS

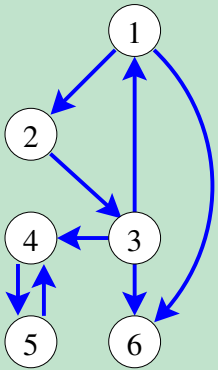
input : Graph $G = (V, E)$, Startknoten s
output : Reihenfolge der besuchten Knoten, Vorgänger p

```
for (jeden Knoten v ∈ 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) ;
    for (u ∈ 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

Beispiel

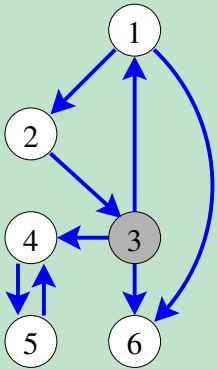


Startknoten $s = 3$

$Q = []$

Breitensuche: Beispiel

Beispiel

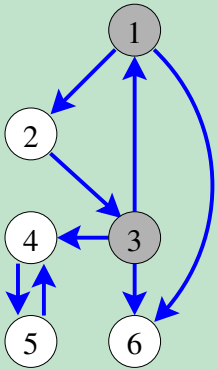


Startknoten $s = 3$

$Q = [3]$

Breitensuche: Beispiel

Beispiel

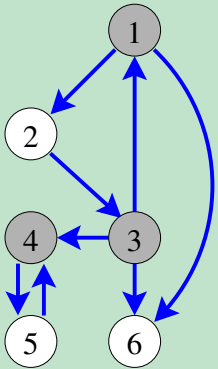


Startknoten $s = 3$

$Q = [3, 1]$

Breitensuche: Beispiel

Beispiel

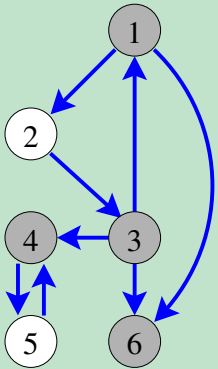


Startknoten $s = 3$

$Q = [3, 1, 4]$

Breitensuche: Beispiel

Beispiel

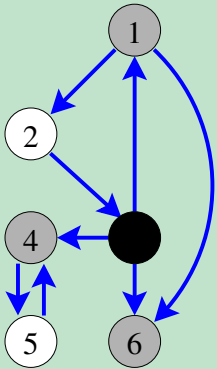


Startknoten $s = 3$

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

Breitensuche: Beispiel

Beispiel

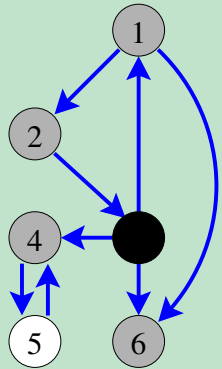


Startknoten $s = 3$

$Q = [1, 4, 6]$

Breitensuche: Beispiel

Beispiel

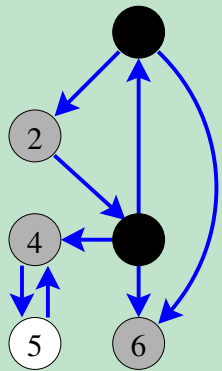


Startknoten $s = 3$

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

Breitensuche: Beispiel

Beispiel

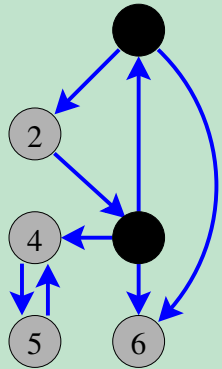


Startknoten $s = 3$

$Q = [4, 6, 2]$

Breitensuche: Beispiel

Beispiel

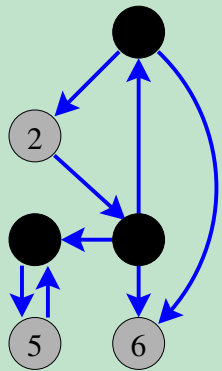


Startknoten $s = 3$

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

Breitensuche: Beispiel

Beispiel

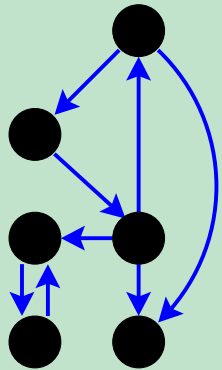


Startknoten $s = 3$

$Q = [6, 2, 5]$

Breitensuche: Beispiel

Beispiel



Startknoten $s = 3$

$Q = []$

Tiefensuche (DFS) I

- Bearbeite einen Knoten v erst dann, wenn alle seine Kinder bearbeitet sind (außer wenn ein Kind auf dem Weg zu v liegt)
- 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 (DFS) II

Wir werden wieder **grau** als *in Bearbeitung* nutzen. Dazu merken wir uns außerdem wann wir einen Knoten zuerst *(in)* und zuletzt besuchen *(out)*.
Es geht aber auch mit weniger Information: *(Link)*

Algorithmus DFS

input : Graph $G = (V, E)$, Startknoten s
output : Reihenfolge der besuchten Knoten, Vorgänger p

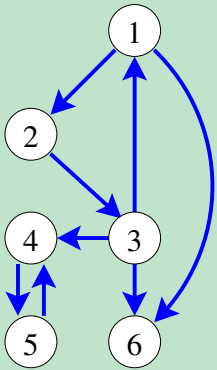
```
for (jeden Knoten v ∈ V) do
    farbe[v] = weiß ;
    p[v] = null ;

zeit = 0;
DFS-visit(G, s);
```

```
DFS-visit(G, v):
    farbe[v] = grau ;
    zeit = zeit + 1 ;
    in[v] = zeit ;
    for (u ∈ succ(v)) do
        if (farbe[u] == weiß) then
            p[u] = v ;
            DFS-visit(G, u); ;
    farbe[v] = schwarz ;
    zeit = zeit + 1 ;
    out[v] = zeit ;
```

weiß=*unbearbeitet*, grau=*in Bearbeitung*, schwarz=*bearbeitet*

Tiefensuche: Beispiel

Beispiel

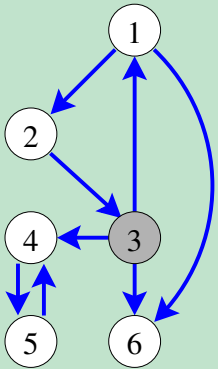


Startknoten *s* = *3*

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

Tiefensuche: Beispiel

Beispiel

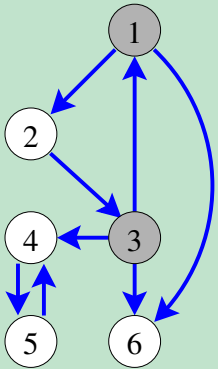


Startknoten *s* = *3*

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

Tiefensuche: Beispiel

Beispiel

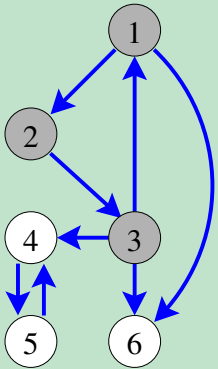


Startknoten *s* = *3*

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

Tiefensuche: Beispiel

Beispiel

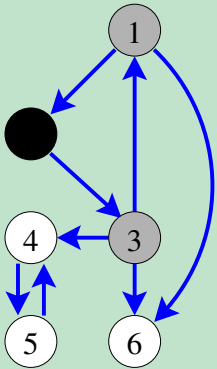


Startknoten *s* = *3*

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

Tiefensuche: Beispiel

Beispiel

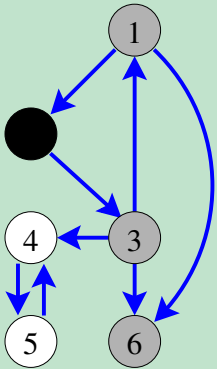


Startknoten *s* = *3*

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

Tiefensuche: Beispiel

Beispiel

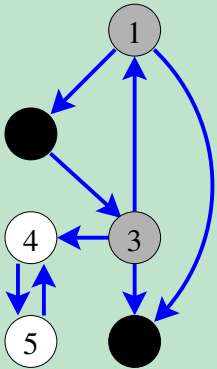


Startknoten *s* = *3*

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

Tiefensuche: Beispiel

Beispiel

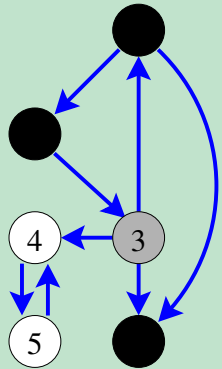


Startknoten *s* = *3*

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

Tiefensuche: Beispiel

Beispiel

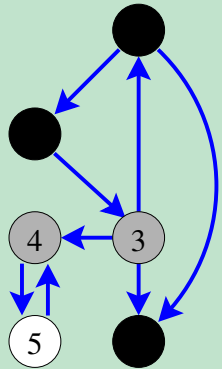


Startknoten *s* = *3*

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

Tiefensuche: Beispiel

Beispiel

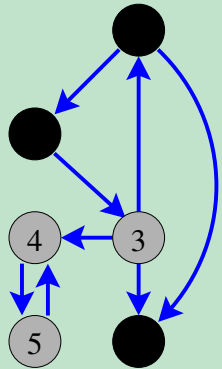


Startknoten *s* = *3*

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

Tiefensuche: Beispiel

Beispiel

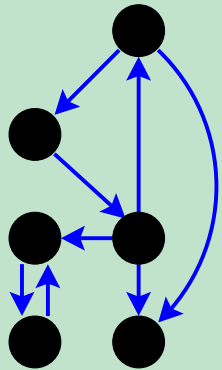


Startknoten *s* = *3*

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

Tiefensuche: Beispiel

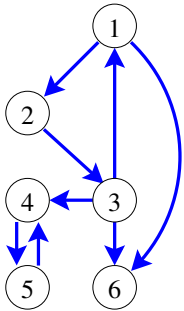
Beispiel



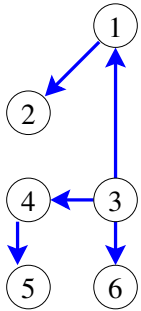
Startknoten *s* = *3*

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

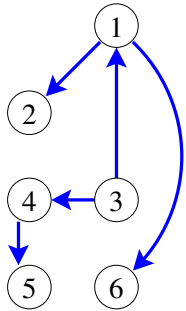
Bäume aus Breiten- und Tiefensuche



Graph *G*



BFS-Baum



DFS-Baum

Gibt es keinen festen Startknoten, wählen wir solange weiße Knoten und suchen von dort aus bis jeder Knoten besucht wurde.

Suche ohne Startknoten

input : Graph $G = (V, E)$,
output : Reihenfolge der besuchten Knoten, Vorgänger p

---

for *(jeden Knoten $v \in V$)* **do**
    farbe[v] = weiß ;
    p[v] = null ;

for *(jeden Knoten $v \in V$)* **do**
    if *(farbe[v] == weiß)* **then**
        Visit(G, v);

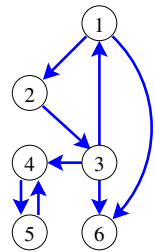
KOMPONENTEN

Starke Zusammenhangskomponenten

Definition

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.

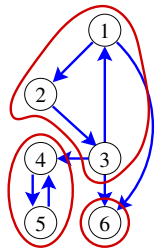


Starke Zusammenhangskomponenten

Definition

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.



Starke Zusammenhangskomponenten

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

gegenseitige Erreichbarkeit is eine 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 Zusammenhangskomponenten durch Tiefensuche

- Führe Tiefensuche (DFS) auf *G* aus.
- Für jeden Knoten `v` berechne dabei `l[v]` = “frühester” Knoten, 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).

***wiki**: Algorithmus von Tarjan zur Bestimmung starker Zusammenhangskomponenten*

Algorithmus von Tarjan (1972)

input : Graph $G = (V, E)$, Knoten v
output : Starke Zusammenhangskomponenten

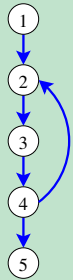
```
Tarjan-visit(G,v):
    farbe[v] = grau ; zeit = zeit + 1 ; in[v] = zeit ; l[v] = zeit ;
    PUSH(S, v) ;
    for (jeden Knoten u ∈ succ(v)) do
        if (farbe[u] == weiß) then
            Tarjan-visit(G, u) ;
            l[v] = min(l[v], l[u]) ;
        else
            if u ∈ S then
                l[v] = min(l[v], in[u]) ;
    if l[v] == in[v] then    // v ist "least" für Komponente
        Ausgabe(”starke Zusammenhangskomponte”) ;
        repeat
            u = TOP(S) ; Ausgabe(u) ; POP(S) ;
        until u==v;
    farbe[v] = schwarz;
```

weiß=*unbearbeitet*, grau=*in Bearbeitung*, schwarz=*bearbeitet*

Tarjan

Beispiel

Wir tabellieren die Knoten in der Reihenfolge, in der sie besucht werden.



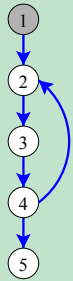
v	in[v]	l[v]

zeit markiert Knoten eindeutig

Tarjan

Beispiel

Wir tabellieren die Knoten in der Reihenfolge, in der sie besucht werden.



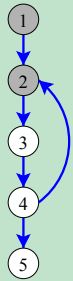
v	in[v]	l[v]
1	1	1

in und *l*east werden mit *zeit* initialisiert wenn ein Knoten das erste mal besucht wird

Tarjan

Beispiel

Wir tabellieren die Knoten in der Reihenfolge, in der sie besucht werden.



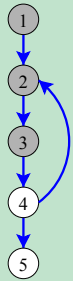
v	in[v]	l[v]
1	1	1
2	2	2

in und *l*east werden mit *zeit* initialisiert wenn ein Knoten das erste mal besucht wird

Tarjan

Beispiel

Wir tabellieren die Knoten in der Reihenfolge, in der sie besucht werden.



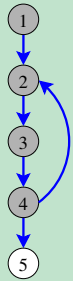
v	in[v]	l[v]
1	1	1
2	2	2
3	3	3

in und *l*east werden mit *zeit* initialisiert wenn ein Knoten das erste mal besucht wird

Tarjan

Beispiel

Wir tabellieren die Knoten in der Reihenfolge, in der sie besucht werden.



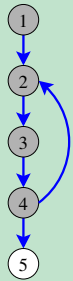
v	in[v]	l[v]
1	1	1
2	2	2
3	3	3
4	4	4

in und *l*east werden mit *zeit* initialisiert wenn ein Knoten das erste mal besucht wird

Tarjan

Beispiel

Wir tabellieren die Knoten in der Reihenfolge, in der sie besucht werden.



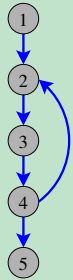
v	in[v]	l[v]
1	1	1
2	2	2
3	3	3
4	4	2

Knoten 1-3 haben jeweils einen Nachfolger, *in* und *l*east sind einfach; Knoten 4 hat $\{2, 5\}$ als Nachfolger, *l*east von 4, ist damit 2

Tarjan

Beispiel

Wir tabellieren die Knoten in der Reihenfolge, in der sie besucht werden.



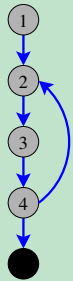
v	in[v]	l[v]
1	1	1
2	2	2
3	3	3
4	4	2
5	5	5

Bearbeite Knoten 5

Tarjan

Beispiel

Wir tabellieren die Knoten in der Reihenfolge, in der sie besucht werden.



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

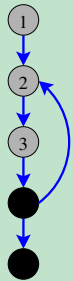
Zshk: 5

Knoten 5 hat keine Nachfolger. Damit ist *l*east==*in* und 5 wird als Zusammenhangskomponente ausgegeben

Tarjan

Beispiel

Wir tabellieren die Knoten in der Reihenfolge, in der sie besucht werden.



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

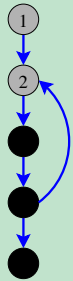
Zshk: 5

Beginne aus den rekursiven Aufrufen für 4,…
zurückzukehren und markiere als erledigt

Tarjan

Beispiel

Wir tabellieren die Knoten in der Reihenfolge, in der sie besucht werden.



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

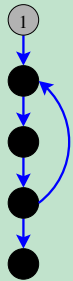
Zshk: 5

3 erledigt, setze *l*east auf Minimum aller Kinder und sich selbst!

Tarjan

Beispiel

Wir tabellieren die Knoten in der Reihenfolge, in der sie besucht werden.



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

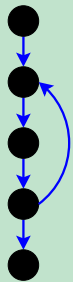
Zshk: 5
Zshk: 2,3,4

2 hat *in*==*l*east und damit haben wir wieder eine Komponente

Tarjan

Beispiel

Wir tabellieren die Knoten in der Reihenfolge, in der sie besucht werden.



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

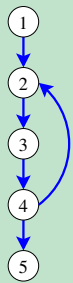
Zshk: 5
Zshk: 2,3,4
Zshk: 1

1 ist die letzte Komponente. Kein weitere Knoten hat ’1’ als *succ*/Nachfolger.

Tarjan

Beispiel

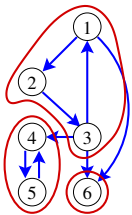
Meistens notieren wir Knoten in der Reihenfolge, in der sie vollständig bearbeitet (schwarz gefärbt) wurden, da sich so die Tabelle nicht mehr verändert. Das gleiche Beispiel ergibt sich so zu:



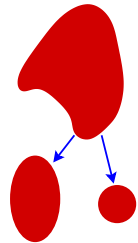
v	in[v]	l[v]
5	5	5
4	4	2
3	3	2
2	2	2
1	1	1

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.

BFS, DFS und Komponenten in ungerichteten Graphen

- Ungerichtete Graphen können als *symmetrische gerichtete Graphen* betrachtet werden, d.h., $\{u, v\} \in E$ entspricht $(u, v) \in E$ *und* $(v, u) \in E$.
- BFS und DFS können daher aus (ohne Änderungen) auf ungerichtet Kanten angewendet werden.
- *Gegenseitige Erreichbarkeit in symmetrischen Graphen* entspricht der Existenz eines Weges im ungerichteten Graphen.
  Zusammenhangskomponenten in umgerichteten Graph sind daher äquivalent zu den starken Zusammenhangskomponenten in symmetrischen Digraphen.
- In einem symmetrischen Digraphen ist x von y genau dann erreichbar, wenn x und y in der selben (starken) Zusammenhangskomponente liegen.
  Erreichbarkeit reduziert sich daher auf Zusammenhang

Zusammenhangskomponenten in ungerichteten Graphen

In ungerichteten (symmetrischen) Graphen können Zusammenhangskomponenten daher viel einfacher bestimmt werden:

- *x* is von einem gegeben Startpunkt *r* aus erreichbar, wenn *x* durch Breitensuche oder Tiefensuche erreichbar ist.
- *x* und *y* liegen in den selben Zusammehangskomponente, wenn sie durch Breitensuche oder Tiefensuche von selben Startpunkt *r* aus erreichbar sind.

Zusammenhangskomponenten in ungerichteten Graphen

Wir erhalten daraus den folgende Algorithmus zur Berechnung der Zusammenhangskomponenten in einem ungerichteten Graphen $G = (V, E)$:

Algorithmus ungerichtete Zusammenhangskomponente

```
Zshg(G)
    WHILE (V≠∅) {
        wähle r∈V beliebig
        Zshgs Komponente V(r) = Menge der Knoten, die von r aus
            durch BFS/DFS erreichbar sind
        V←V\V(r)
    }
```

Die Union-Find Datenstruktur

Oftmals ist es von Interesse rasch überprüfen zu können ob zwei Knoten in der selben Zusammenhangskomponente eines Graphen liegen.

- **Grundidee:** jede Zusammenhangskomponente wird durch einen eindeutigen Knoten repräsentiert. (Z.B. der Wurzel des BFS Baums).
- $\texttt{Find}(x)$ Abfrage: bestimmt zu jedem Knoten x den Repräsentanten $\texttt{Find}(x)$.
  Praktische Implementation: Baum.
  $\texttt{Find}(x)$ sucht den Weg von x zur Wurzel, die Wurzel enthält den Repräsentanten.
- Zwei Knoten sind genau dann in der gleichen Zusammenhangskomponente wenn $\texttt{Find}(x) = \texttt{Find}(y)$.
  Kann in $O(\log |V|)$ Operationen überprüft werden, wenn die Tiefe des Baums auf $O(\log |V|)$ beschränkt ist.

Die Union-Find Datenstruktur

Im Kruskal Algorithmus für Spannbäume werden Zusammenhangskomponente durch Einfügen einer Kante vereinigt.

- `Union`(x, y) Operation. Vereinigt die Zusammenhangskomponenten von x und y.
- **Implementation:** Wenn $r_x := \texttt{Find}(x) \neq \texttt{Find}(y) =: r_y$ dann hänge die Wurzel des Baums mit der kleineren Tiefe als Kind an die Wurzel des Baumes mit grösseren Tiefe. (Bei Gleichheit der Tiefe behandle den zweiten Baum als den mit kleinerer Tiefe.)
- Sei r_y die Wurzel des Baums mit kleinerer Tiefe. Dann ist r_x die Wurzel des vereinigten Baums und damit der (eindeutig bestimmte) Repräsentant der vereinigten Zusammenhangskomponenten.
- Die Vereinigung der beiden Bäume benötigt nur Konstante Zeit (Einfügen einer Kante). Der Aufwand für die `Union`-Operation ist daher durch den Aufwand für die beiden `Find`-Operationen gegeben.

Die Union-Find Datenstruktur

Die Pfadkompression garantiert, dass der Aufwand für `Find`-Operationen auf $O(\log |V|)$ beschränkt bleibt.

Pfadkompression: Nach jedem Aufruf vom `Find`(x) werden alle knoten entlang des Weges von x zu Wurzel direkt mit der Wurzel verlink.