

Datenstrukturen und Algorithmen

Vorlesung 15: Elementare Graphenalgorithmen II

Joost-Pieter Katoen

Lehrstuhl für Informatik 2
Software Modeling and Verification Group

<http://www-i2.rwth-aachen.de/i2/dsa110/>

22. Juni 2010



Übersicht

1 Starke Zusammenhangskomponenten

- Kondensationsgraph
- Sharir's Algorithmus

2 Gerichtete zyklfreie Graphen

- Topologische Sortierung
- Kritische-Pfad-Analyse

Übersicht

1 Starke Zusammenhangskomponenten

- Kondensationsgraph
- Sharir's Algorithmus

2 Gerichtete zyklfreie Graphen

- Topologische Sortierung
- Kritische-Pfad-Analyse

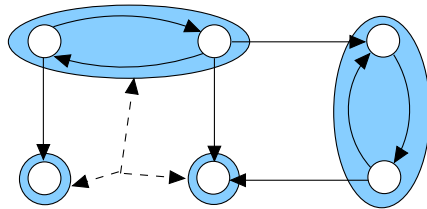
Zusammenhängende gerichtete Graphen

Sei G ein gerichteter Graph.

Zusammenhang

- ▶ G heißt **stark zusammenhängend** (strongly connected), wenn jeder Knoten von jedem anderen aus erreichbar ist.
- ▶ G heißt **schwach zusammenhängend**, wenn der zugehörige ungerichtete Graph (wenn man alle Kanten ungerichtet macht) zusammenhängend ist.
- ▶ Eine **starke Zusammenhangskomponente** (strongly connected component) von G ist ein maximaler stark zusammenhängender Teilgraph von G .
- ▶ Ein nicht-verbundener Graph kann **eindeutig** in verschiedene Zusammenhangskomponenten **aufgeteilt** werden.

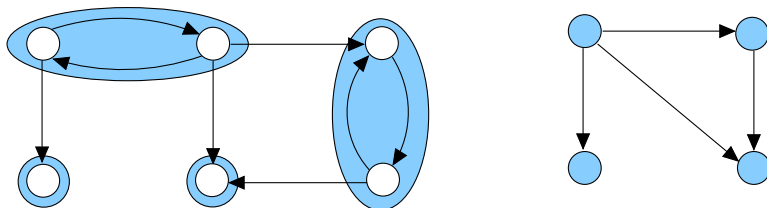
Starke Zusammenhangskomponenten



Zusammenhangskomponenten

Ein nicht-zusammenhängender Digraph, aufgeteilt in seine maximalen zusammenhängenden Teilgraphen.

Kondensationsgraph – Beispiel



starke Zusammenhangskomponenten

Ein nicht-zusammenhängender Digraph und seine Kondensation.

Kondensationsgraph

Die starken Komponenten von G induzieren den **Kondensationsgraph**.

Kondensationsgraph

Sei $G = (V, E)$ ein gerichteter Graph mit k starken Komponenten $S_i = (V_i, E_i)$ für $0 < i \leq k$.

Der **Kondensationsgraph** $G \downarrow = (V', E')$ ist definiert als:

- ▶ $V' = \{ V_1, \dots, V_k \}$.
- ▶ $(V_i, V_j) \in E'$ gdw. $i \neq j$ und es gibt $(v, w) \in E$ mit $v \in V_i$ und $w \in V_j$.

Der Kondensationsgraph $G \downarrow$ ist **azyklisch**.

Starke Komponenten und Transponierung

Transponieren

Der **transponierte** Graph von $G = (V, E)$ ist $G^T = (V, E')$ mit $(v, w) \in E'$ gdw. $(w, v) \in E$.

In G^T ist die Richtung der Kanten von G gerade umgedreht.

Lemma: Beziehung zwischen G und G^T

1. Die starken Komponenten von G und G^T sind **die selben**.
2. Die Kondensation und die Transposition **kommutieren**, d. h.:

$$(G \downarrow)^T = (G^T) \downarrow.$$

Beweis: Übungsaufgabe.

Erinnerung: Tiefensuche

Tiefensuche (Depth-First Search, DFS)

Am Anfang seien alle Knoten als „nicht-gefunden“ (WHITE) markiert.

Die zugrundeliegende Strategie ist:

- ▶ Markiere den aktuellen Knoten v als „gefunden“ (GRAY).
 - ▶ Für jede Kante (v, w) im Graph G mit „nicht-gefundenem“ Nachfolger w :
 - ▶ Suche rekursiv von w aus, d. h.:
 - ▶ Erforsche Kante (v, w) , besuche w , forsche von dort aus, bis es nicht mehr weiter geht.
 - ▶ Dann backtracke von w nach v .
 - ▶ Für jede Kante (v, w) in G mit gefundenem Nachfolger w :
 - ▶ „Überprüfe“ die Kante, ohne aber w zu besuchen.
 - ▶ Markiere Knoten v als „abgeschlossen“ (BLACK).
- ▶ Man erhält wieder die Menge aller Knoten, die vom Startknoten aus **erreichbar** sind.

DFS auf einen Graphen

```

1 void dfsGraphSearch(List adjLst[n], int n) {
2   int color[n];
3   for (int v = 0; v < n; v++) { color[v] = WHITE; }
4   for (int v = 0; v < n; v++) {
5     if (color[v] == WHITE) { dfsSearch(adjLst, n, v, color); }
6   }
7 }

9 void dfsSearch(List adjL[n], int n, int start, int &color[n]) {
10  color[start] = GRAY;
11  foreach (next in adjL[start]) {
12    if (color[next] == WHITE) {
13      dfsSearch(adjL, n, next, color);
14    }
15  }
16  color[start] = BLACK; // Schliesse ab
17 }

```

Algorithmus zum Finden starker Komponenten

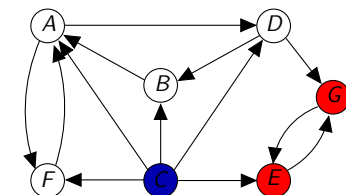
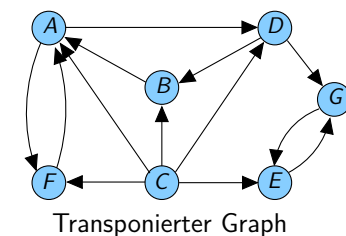
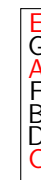
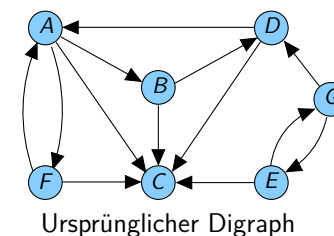
Sharir's Algorithmus findet starke Komponenten in **zwei** Phasen:

1. Führe eine DFS auf G durch,
 - ▶ wobei alle Knoten beim Abschließen (d. h. wenn der Knoten BLACK gefärbt wird) auf einem Stack gespeichert werden.
2. Führe eine DFS auf dem transponierten Graphen G^T durch. Dazu
 - ▶ färbe alle Knoten WHITE (wie üblich);
 - ▶ beginne jeweils bei noch weißen Knoten vom (in Phase 1 erzeugten) Stack, d. h. Knoten auf dem Stapel, die grau oder schwarz sind, werden ignoriert, und
 - ▶ speichere den **Leiter** der zu Knoten v gehörenden starken Komponente.

Leiter einer starken Komponente

Ein Knoten v in einer starken Komponente S_i heißt **Leiter** (leader), wenn er als **erster** Knoten bei einer DFS von S_i entdeckt wird (d. h. GRAY gefärbt wird).

Sharir's Algorithmus – Beispiel



Korrektheit (I)

Lemma

Sei v Leiter der starken Komponente S_i , w eine Knoten in S_j , $i \neq j$, und es existiert ein Pfad von v nach w . Es gilt: Wenn v bei einer DFS entdeckt wird (d.h. GRAY gefärbt wird), dann:

1. w ist BLACK, oder
2. es existiert ein weißer Pfad $\underbrace{w \ u_0 \ \dots \ u_n \ v}_{\text{weiße Knoten}}$ von w nach v .

Beweis:

In der Vorlesung. Beruht auf folgender Eigenschaft der Tiefensuche: Es existiert ein weißer Pfad von w nach v . Dann gilt: v ist Nachfolger von w in dem DFS-Baum. Dies kann man durch Induktion über die Länge des weißen Pfades von w nach v beweisen.

Komplexität

Zeitkomplexität

Die Worst-Case Zeitkomplexität von Sharir's Algorithmus zum Finden starker Komponenten in einen gerichteten Graph ist $\Theta(|V| + |E|)$. Seine Speicherkomplexität ist $\Theta(|V|)$.

Beweis

- ▶ Die DFS über G und G^T benötigen jeweils $\Theta(|V| + |E|)$.
- ▶ Der transponierte Graph G^T kann in $\Theta(|V| + |E|)$ gebildet werden.
- ▶ Der Stack benötigt $\Theta(|V|)$ Speicher.

Korrektheit (II)

Lemma

Jeder weißer Knoten, der in der 2. Phase vom Stapel genommen wird, ist Leiter einer starken Komponente.

Korrektheit

1. Jeder in Phase 2 erzeugte DFS-Baum ist gerade eine starke Komponente.
2. Alle starken Komponenten von G^T (und deswegen auch von G) werden in der 2. Phase bestimmt.

Übersicht

- 1 Starke Zusammenhangskomponenten
 - Kondensationsgraph
 - Sharir's Algorithmus
- 2 Gerichtete zyklfreie Graphen
 - Topologische Sortierung
 - Kritische-Pfad-Analyse

Gerichtete zyklfreie Graphen

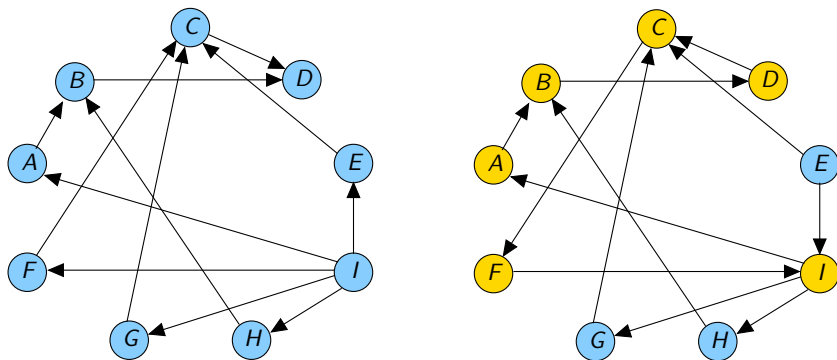
Gerichtete zyklfreie Graphen

Gerichtete zyklfreie Graphen (directed acyclic graph, DAG) sind eine wichtige Klasse von Graphen:

- ▶ Viele Probleme lassen sich naturgemäß mit Hilfe von DAGs formulieren.
 - ▶ Scheduling: Vorranggraphen beschreiben, welche Aufgaben erledigt sein müssen, bevor ein nachfolgender Schritt beginnen kann.
 - ▶ Ein Zyklus in solch einem Vorranggraphen wäre ein Deadlock.
- ▶ Viele Probleme haben auf DAGs eine niedrigere Komplexität als auf Digraphen.
- ▶ Ein DAG entspricht einer **partiellen Ordnung** $<$ auf den Knoten:
 - ▶ Eine Kante (v, w) besagt: $v < w$.
 - ⇒ Da eine partielle Ordnung anti-symmetrisch ist, kann sie keine Zyklen enthalten.

- ▶ Wir betrachten: **Topologische Sortierung** und **Kritische-Pfad-Analyse**.

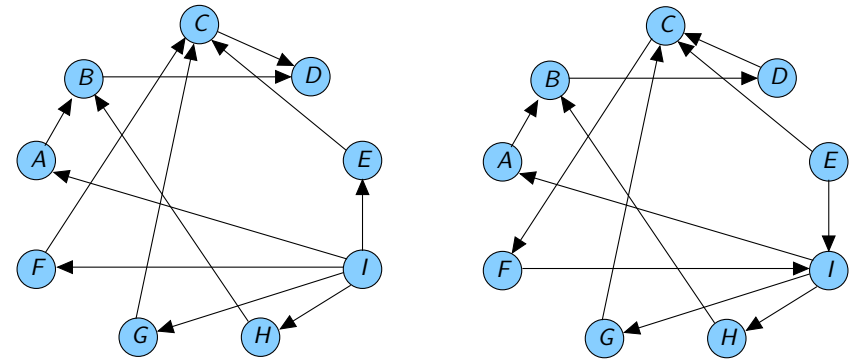
Topologische Sortierung – Motivation (II)



Der rechte enthält einen Zyklus!

Der linke enthält keinen Zyklus.

Topologische Sortierung – Motivation (I)



Welcher der gerichteten Graphen ist azyklisch?

Topologische Ordnung

Topologische Ordnung

Sei $G = (V, E)$ ein gerichteter Graph mit n Knoten. Eine **topologische Ordnung** von G ist eine Zuordnung $topo : V \rightarrow \{1, \dots, n\}$, so dass:

für jede Kante $(v, w) \in E$ gilt: $topo(v) < topo(w)$.

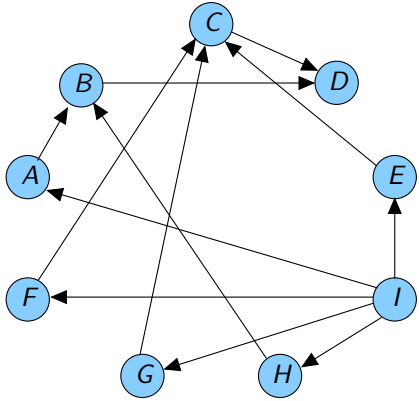
$topo(v)$ heißt der **topologische Zahl** von v .

Eine topologische Ordnung ist die Einbettung einer partiellen Ordnung in eine totale (d. h. lineare Ordnung).

Lemma

1. Für einen Digraph G mit einem Zyklus existiert **keine** topologische Ordnung.
2. Jeder DAG G dagegen hat mindestens eine topologische Ordnung.

Topologische Sortierung – Beispiel



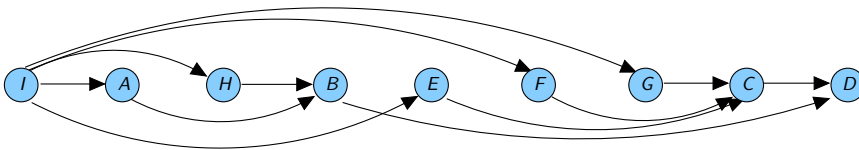
Nr	Aufgabe	Hängt ab von
A	choose clothes	I
B	dress	A, H
C	eat breakfast	E, F, G
D	leave	B, C
E	make coffee	I
F	make toast	I
G	pour juice	I
H	shower	I
I	wake up	–

Vorranggraph (precedence graph)

Transponierter Graph von „Hängt ab von“

- Gibt es einen **Schedule** für dieses Problem? D. h. kann man eine Reihenfolge finden, um alle Aufgaben ausführen zu können?

Topologische Sortierung – Ergebnis



Abhängigkeitsgraph, der topologischen Ordnung entsprechend gezeichnet.

Topologische Sortierung – Implementierung

```

1 void dfsSearch(List adjLst[n], int n, int start, int &color[n],
2                 int &topoNum, int &topo[n]) {
3     color[start] = GRAY;
4     foreach (next in adjLst[start]) {
5         // if (color[next] == GRAY) throw "Graph ist zyklisch";
6         if (color[next] == WHITE) {
7             dfsSearch(adjLst, n, next, color, topoNum, topo);
8         }
9     }
10    topo[start] = ++topoNum;
11    color[start] = BLACK;
12 }

14 // Ausgabe der topologischen Zahl von Knoten v in topo[v]
15 void topoSort(List adjLst[n], int n, int &topo[n]) {
16     int color[n] = WHITE, topoNum = 0; // vgl. connComponents()
17     for (int v = 0; v < n; v++)
18         if (color[v] == WHITE)
19             dfsSearch(adjLst, n, v, color, topoNum, topo);
20 }

```

Korrektheit

Theorem

Der Algorithmus terminiert, und wenn er terminiert enthält das Array topo eine topologische Ordnung von G.

Beweis:

1. Die DFS besucht jeden Knoten, daher sind die Zahlen in dem Array topo alle verschieden im Bereich 1 bis N.
2. Sei $(v, w) \in E$. w ist kein Vorgänger im DFS-Baum, sonst wäre G nicht azyklisch. Damit folgt, dass w BLACK ist, wenn topo[v] ein Wert zugewiesen wird. Also wurde topo[w] schon vorher ein Wert zugewiesen. Da topoNum immer größer wird, folgt topo[v] > topo[w].

Komplexität

Zeitkomplexität

Eine topologische Ordnung kann in $\Theta(|V| + |E|)$ bestimmt werden.

Gewichtete Graphen – Darstellung (I)

Gewichtete Graphen werden ebenso als Adjazenzlisten oder Adjazenzmatrix dargestellt:

- ▶ Bei **knotengewichteten** Graphen wird die Zusatzinformation zu den Knoten üblicherweise in einem weiteren Array gespeichert – vgl. `int color[n]`; bei BFS oder DFS.
- ▶ **Kantengewichte** können bei der Adjazenzmatrixdarstellung direkt in der Matrix gespeichert werden.
Ein besonderer Wert, etwa 0 oder ∞ besagt, dass keine Kante existiert.

Gewichtete Graphen

Knotengewichteter Graph

Ein **knotengewichteter** Graph G ist ein Tripel (V, E, W) , wobei:

- ▶ (V, E) ein – gerichteter oder ungerichteter – Graph ist, und
- ▶ $W : V \rightarrow \mathbb{R}$ die **Gewichtsfunktion**.
 $W(v)$ ist das Gewicht des Knotens v .

Kantengewichteter Graph

Ein **(kanten-)gewichteter** Graph G ist ein Tripel (V, E, W) , wobei:

- ▶ (V, E) ein – gerichteter oder ungerichteter – Graph ist, und
- ▶ $W : E \rightarrow \mathbb{R}$ Gewichtsfunktion. $W(e)$ ist das Gewicht der Kante e .
- ▶ Ein knotengewichteter Graph (V, E, W) lässt sich in einen kantengewichteten Graphen (V, E, W') überführen, indem *alle* von einem Knoten v ausgehenden Kanten $e = (v, \cdot) \in E$ das Gewicht $W'(e) = W(v)$ erhalten.

Gewichtete Graphen – Darstellung (II)

- ▶ Bei der Adjazenzlistendarstellung von kantengewichteten Graphen wird das Gewicht jeweils mit in der Liste gespeichert:

Beispiel (Kantengewichteter Graph als Adjazenzlisten)

```
1 // bisher (ohne Gewichte):
2 List adjList[n]; // wobei List im Grunde "List<int>" war, also:
3 List<int> adjList[n];

5 // neu (mit Gewichten):
6 struct Edge {
7     int w;
8     float weight;
9 }
10 List<Edge> adjList[n];
11 // wir verwenden aber weiterhin die Kurzschreibweise:
12 List adjList[n]; // ggf. mit Gewichten
```

Das Kritische-Pfad-Problem – Einführung

Kritischer-Pfad-Problem

Finde den **längsten** Pfad (bezogen auf das Gesamtgewicht) in einem (kanten- oder knoten-)gewichteten DAG.

- Wir betrachten hier *nur* **knotengewichtete** DAGs.

Beispiel (Anwendung)

Wie lange benötigt man für die Ausführung der bereits vorgestellten Aufgaben mindestens, wenn für jede Aufgabe eine Dauer gegeben ist und unabhängige Aufgaben gleichzeitig erledigt werden können?

Das Kritische-Pfad-Problem – Anwendung

Kritische Pfad

Der **kritische Pfad** ist eine Folge von Aufgaben $v_0, \dots, v_k$, so dass

- v_0 keine Abhängigkeiten hat.
- v_i abhängig von v_{i-1} ist, wobei $est(v_i) = eft(v_{i-1})$ (kein Schlupf).
- $eft(v_k)$ das Maximum über alle Aufgaben ergibt.

Es gibt eine **kritische** Abhängigkeit zwischen v_{i-1} und v_i , d.h. eine Verzögerung in v_{i-1} führt zu einer Verzögerung in v_i .

Das Kritische-Pfad-Problem – Anwendung

Früheste Startzeit- und Endzeitpunkt

Finde den **frühestmöglichen Beendigungszeitpunkt** (earliest finish time) für eine Menge voneinander abhängiger Aufgaben.

- Jede Aufgabe hat eine (nicht-negative) **Dauer**.
- Der **früheste Startzeitpunkt** (earliest start time) für Aufgabe v ($est(v)$) ist 0 wenn v keine Abhängigkeiten hat; andernfalls:
- $est(v)$ ist das Maximum der frühesten Endzeitpunkte seiner Abhängigkeiten.

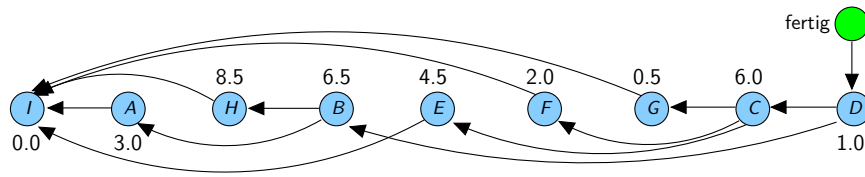
Der **früheste Endzeitpunkt** (earliest finish time) für Aufgabe v ($eft(v)$) ist gleich $est(v)$ plus der Dauer von v .

Kritische-Pfad-Analyse (I)

```

1 // Knotengewichte in duration. Aufruf analog zu
  connComponents()
2 // Ausgabe: eft, kritischer Pfad als Vorgängerliste in critDep
3 void dfsSearch(List adjL[n], int n, int start, int &color[n],
4               int duration[n], int &critDep[n], int &eft[n]) {
5     int est = 0;
6     color[start] = GRAY;
7     critDep[start] = -1;
8     foreach (next in adjL[start]) {
9         if (color[next] == WHITE) {
10             dfsSearch(adjL, n, next, color, duration, critDep, eft);
11         }
12         if (eft[next] >= est) {
13             est = eft[next];
14             critDep[start] = next;
15         }
16     }
17     eft[start] = est + duration[start];
18     color[start] = BLACK;
19 }
```

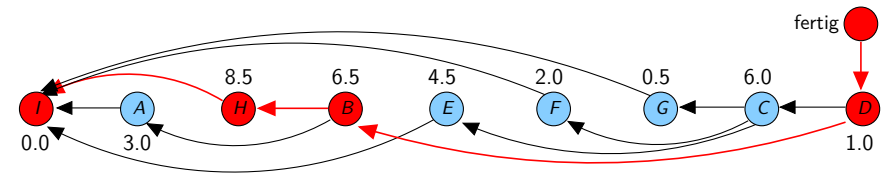
Kritische-Pfad-Analyse (II)



I	A	H	B	E	F	G	C	D
wake up	choose clothes	shower	dress	make coffee	make toast	pour juice	eat breakf	leave
0.0	3.0	8.5	6.5	4.5	2.0	0.5	6.0	1.0

Dauer

Kritische-Pfad-Analyse (II)



I	A	H	B	E	F	G	C	D
wake up	choose clothes	shower	dress	make coffee	make toast	pour juice	eat breakf	leave
0.0	3.0	8.5	6.5	4.5	2.0	0.5	6.0	1.0

Dauer

► $eft = 1 + 6.5 + 8.5 + 0 = 16.$