

ALGORITHMEN UND DATENSTRUKTUREN

ÜBUNG 10: TOPOLOGISCHES SORTIEREN & GRAPHENSUCHE

Eric Kunze

`eric.kunze@tu-dresden.de`

Topologisches Sortieren

Implementierung

TOPOLOGISCHES SORTIEREN

Gegeben sei ein gerichteter, azyklischer Graph $G = (V, E)$. Eine **topologische Sortierung** von G ist eine *bijektive* Abbildung $\text{ord}: V \rightarrow \{1, \dots, |V|\}$, sodass für alle $v, v' \in V$ mit $(v, v') \in E$ die Relation $\text{ord}(v) < \text{ord}(v')$ gilt.

Anschaung: $\text{ord}(v) = n \rightsquigarrow$ Knoten v wird als n -tes Element gewählt (erhält Sortierungsnummer n)

Algorithmus:

while (Elemente übrig)

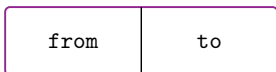
- ▶ wähle Element v ohne Vorgänger
- ▶ dekrementiere Anzahl der Vorgänger in den Nachfolgern von v
- ▶ füge v der Ausgabeliste hinzu
- ▶ lösche v aus G

KODIERUNGEN

Kanten:

```
struct Edge { int from, to; };
```

struct Edge



Bsp.: Kante $e = (3, 5)$

```
struct Edge e = {3,5}
```

```
e.from == 3
```

```
e.to == 5
```

Abbildung ord: Array int ord[]

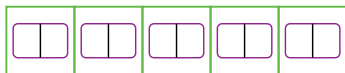
mit $\text{ord}(v) = j \Leftrightarrow \text{ord}[v] = j$

► initialisiert mit -1

Graph: Liste von Kanten

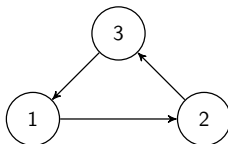
```
struct Edge edges[];
```

struct Edge edges[]



Bsp.: Graph mit Kantenmenge

$E = \{(1, 2), (2, 3), (3, 1)\}$



```
struct Edge edges[] =
```

```
{ {1,2}, {2,3}, {3,1} };
```

EIN ALTERNATIVER ALGORITHMUS

while (Elemente übrig)

- ▶ wähle Element v ohne Vorgänger
- ▶ dekrementiere Anzahl der Vorgänger in den Nachfolgern von v
- ▶ füge v der Ausgabeliste hinzu
- ▶ lösche v aus G

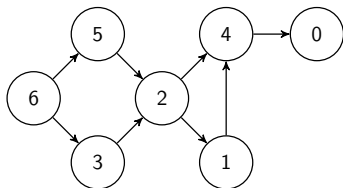
for jede Sortierungsnummer j

- ▶ for jeden Knoten v
 - ▷ teste ob v gewählt werden kann, d.h. noch nicht platziert ist und eingehende Kanten bereits platziert
- ▶ falls v gewählt werden darf, setze $\text{ord}[v] = j$

AUFGABE 1

```
1 void topsort(int n, int e, struct Edge edges[], int ord[]) {
2     int j = 1, node, edge, ok;
3     while (j <= n) {
4         for (node = 0; node < n; ++node) {
5             if (ord[node] == -1) {
6                 ok = 1;
7                 for (edge = 0; edge < e; ++edge) {
8                     if (edges[edge].to == node &&
9                         ord[edges[edge].from] == -1)
10                         ok = 0;
11                 }
12                 if (ok) {
13                     ord[node] = j;
14                     j++;
15                     break;
16                 }
17             }
18         }
19     }
20 }
```

EIN BEISPIEL



Knoten 6 bekommt Nummer 1.
Knoten 3 bekommt Nummer 2.
Knoten 5 bekommt Nummer 3.
Knoten 2 bekommt Nummer 4.
Knoten 1 bekommt Nummer 5.
Knoten 4 bekommt Nummer 6.
Knoten 0 bekommt Nummer 7.

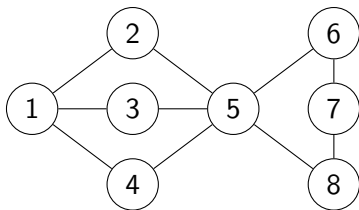
$v =$	0	1	2	3	4	5	6
$\text{ord}(v) =$	7	5	4	2	6	3	1

$\text{ord} = [7, 5, 4, 2, 6, 3, 1]$

Breiten- und Tiefensuche

SUCHVERFAHREN

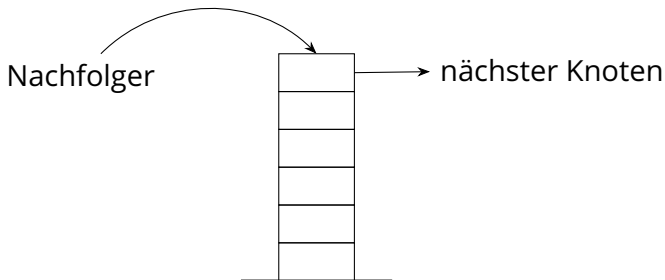
- ▶ Ziel: Finden eines Knotens mit bestimmter Beschriftung in einem Graphen
- ▶ hier: uninformierte Suche mit Tiefen- und Breitensuche



TIEFENSUCHE

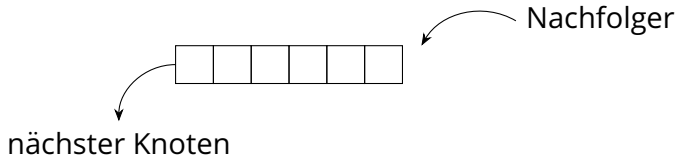
- ▶ gehe in die Tiefe:
„entdecke erst Kinder, dann Geschwister“
- ▶ Datenstruktur: Keller
- ▶ Nachfolger werden *oben* auf den Keller gelegt
- ▶ nächster Knoten wird *oben* vom Keller genommen

Keller:



- ▶ gehe in die Breite:
„entdecke erst Geschwister, dann Kinder“
- ▶ Datenstruktur: Warteschlange
- ▶ Nachfolger stellen sich *hinten* an
- ▶ nächster Knoten wird von *vorn* genommen

Warteschlange:



VERALLGEMEINERTE GRAPHENSUCHE

Beobachtung: Suche läuft ähnlich ab

- ▶ Operation 1: Lesen des nächsten Knotens READ
- ▶ Operation 2: Löschen des gewählten Knotens REMOVE
- ▶ Operation 3: Hinzufügen eines Nachfolgerknotens INSERT
- ▶ Operation 4: Leerheit der Datenstruktur EMPTY

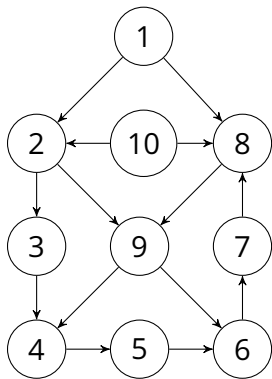
	STORAGE	READ	REMOVE	INSERT	EMPTY
Tiefensuche	Keller	top	pop	push	empty
Breitensuche	Queue	head	dequeue	enqueue	nil

weitere Möglichkeit für STORAGE: **Prioritätswarteschlange**

(vgl. Übung 11, Dijkstra-Algorithmus für kürzeste Wege in gewichteten Graphen)

- ▶ Wahl des nächsten Elementes anhand eines Prioritätswertes
- ▶ Vorstellung: *geordnete* Warteschlange

AUFGABE 2



(a) **Tiefensuche:**

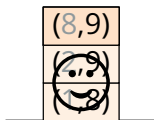
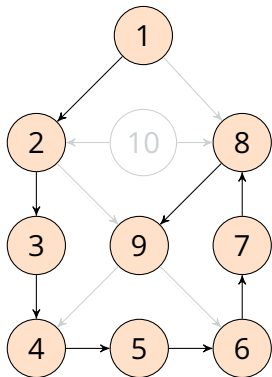
3 verschiedene
depth-first trees

(b) **Breitensuche:**

3 verschiedene
breadth-first trees

AUFGABE 2 — TEIL (A)

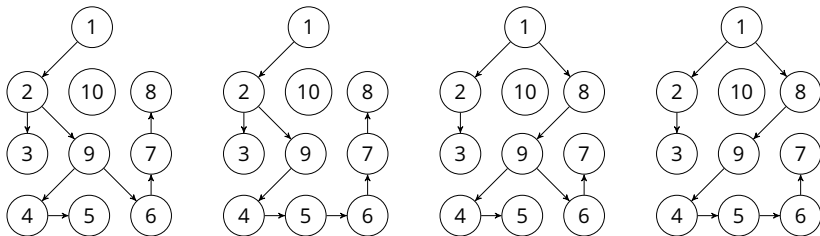
Tiefensuche



AUFGABE 2 — TEIL (A)

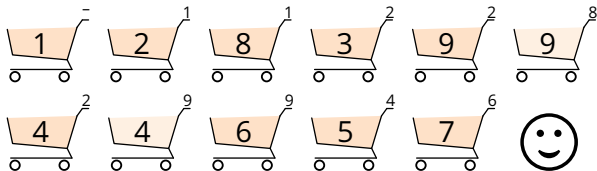
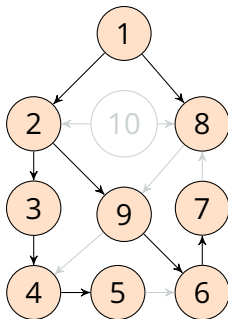
Tiefensuche

Es gibt 5 verschiedene depth-first-trees, die weiteren DFTs sind:



AUFGABE 2 — TEIL (B)

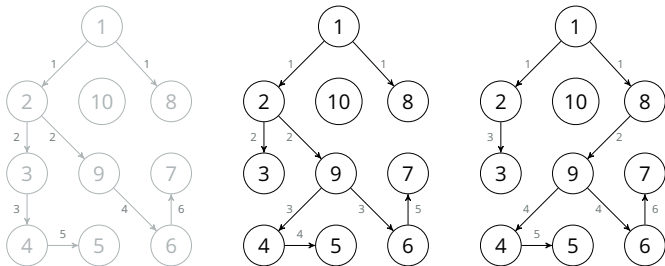
Breitensuche



AUFGABE 2 — TEIL (B)

Breitensuche

Es gibt 5 verschiedene breadth-first-trees, zwei weitere sind z.B.:



... die zwei weiteren Varianten einer möglichen Warteschlange liefern Bäume, die wir schon gefunden haben. Effektiv gibt es also nur diese drei BFTs.