Datenstrukturen und Algorithmen

Übung 10

FS 2020

Keine Übungsgruppe

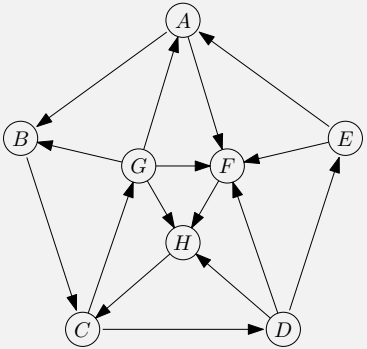
Auch wenn keine Übungen stattfinden diese Woche wollten wir Ihnen das Folienmaterial nicht vorenthalten

Programm von heute

1. Feedback letzte Übungen
2. Wiederholung Vorlesung
   - Algorithmus von Jarnik, Prim, Dijkstra
3. Programmieraufgabe

1. Feedback letzte Übungen

Tiefen- und Breitensuche

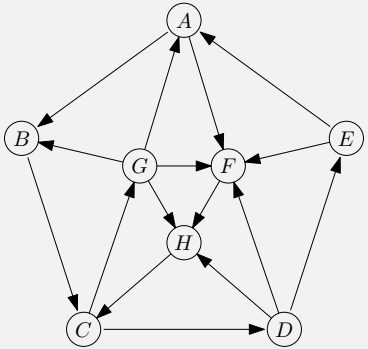


Start bei A

DFS: A, B, C, D, E, F, H, G

BFS: A, B, F, C, H, D, G, E

Tiefen- und Breitensuche



Start bei A

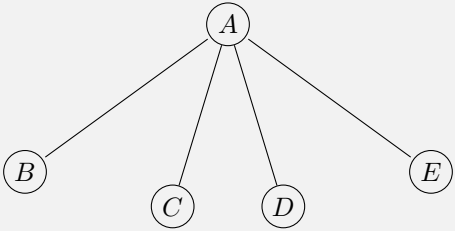
DFS: A, B, C, D, E, F, H, G

BFS: A, B, F, C, H, D, G, E

Es gibt keinen Startknoten, sodass die DFS-Ordnung der BFS-Ordnung entspricht.

Tiefen- und Breitensuche

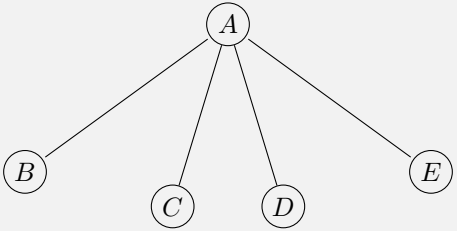
Stern: DFS-Ordnung entspricht BFS-Ordnung



Start bei A
DFS: A, B, C, D, E
BFS: A, B, C, D, E

Tiefen- und Breitensuche

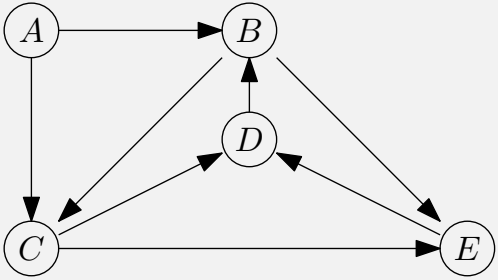
Stern: DFS-Ordnung entspricht BFS-Ordnung



Start bei A
DFS: A, B, C, D, E
BFS: A, B, C, D, E

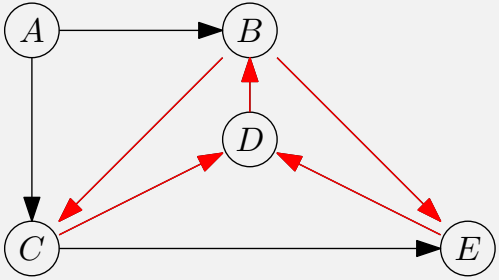
Start bei C
DFS: C, A, B, D, E
BFS: C, A, B, D, E

Topologische Sortierung



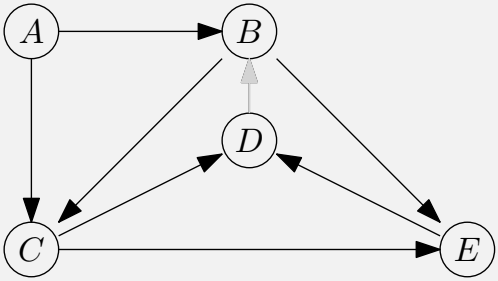
- der Graph hat Kreise

Topologische Sortierung



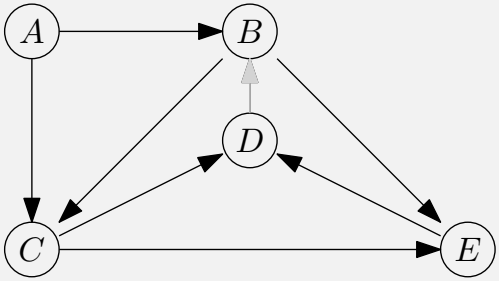
- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante

Topologische Sortierung



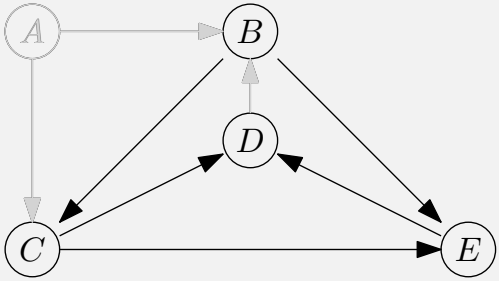
- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante
- entferne die Kante
  $\Longrightarrow$ kreisfrei

Topologische Sortierung



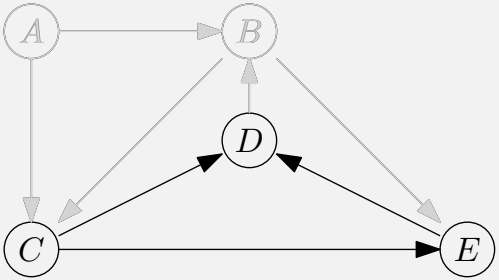
- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante
- entferne die Kante
  $\Longrightarrow$ kreisfrei
- topologische Sortierung durch „Entfernen“ von Knoten mit Eingangsgrad 0

Topologische Sortierung



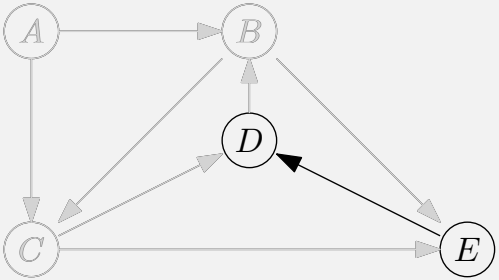
- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante
- entferne die Kante
  $\Longrightarrow$ kreisfrei
- topologische Sortierung durch „Entfernen“ von Knoten mit Eingangsgrad 0

Topologische Sortierung



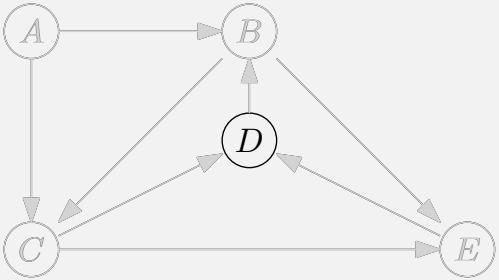
- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante
- entferne die Kante
  $\Longrightarrow$ kreisfrei
- topologische Sortierung durch „Entfernen“ von Knoten mit Eingangsgrad 0

Topologische Sortierung



- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante
- entferne die Kante
  $\Longrightarrow$ kreisfrei
- topologische Sortierung durch „Entfernen“ von Knoten mit Eingangsgrad 0

Topologische Sortierung



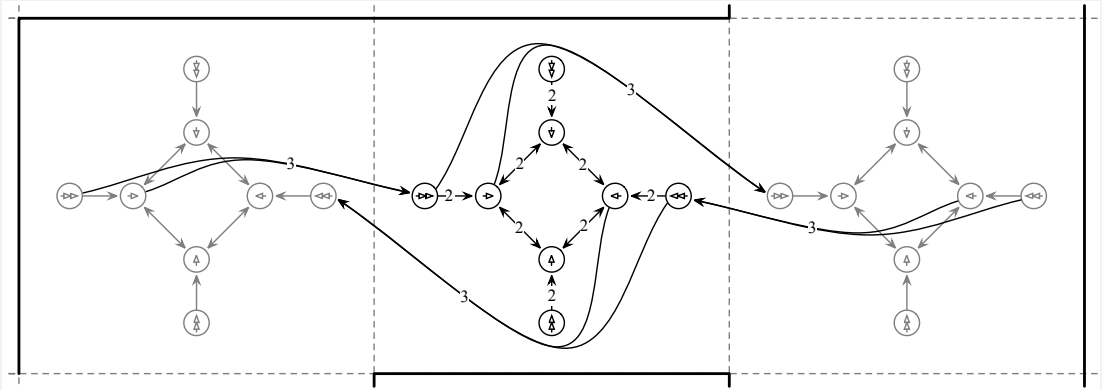
- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante
- entferne die Kante
  $\Longrightarrow$ kreisfrei
- topologische Sortierung durch „Entfernen“ von Knoten mit Eingangsgrad 0

Aufgabe : Labyrinth

- Roboter muss anhalten um Richtung zu ändern
- Als shortest-path Problem auffassen

Aufgabe : Labyrinth

- Position $\times$ Richtung $\times$ Geschwindigkeit



- Laufzeit?

Aufgabe Labyrinth

- Sei n Anzahl Quadrate. Graph hat $|V| = 8n$ Knoten
- Graph hat $|E| \leq 20n$ Kanten
- Daher hat Dijkstra $\mathcal{O}(|E| + |V| \log |V|)$ Laufzeit $\mathcal{O}(n \log n)$

2. Wiederholung Vorlesung

A*-Algorithmus($G, s, t, \hat{h}$)

Input: Positiv gewichteter Graph $G = (V, E, c)$, Startpunkt $s \in V$, Endpunkt $t \in V$, Schätzung $\widehat{h}(v) \leq \delta(v, t)$

Output: Existenz und Wert eines kürzesten Pfades von s nach t

foreach $u \in V$ **do**
    $d[u] \leftarrow \infty$; $\widehat{f}[u] \leftarrow \infty$; $\pi[u] \leftarrow$ null

$d[s] \leftarrow 0$; $\widehat{f}[s] \leftarrow \widehat{h}(s)$; $R \leftarrow \{s\}$; $M \leftarrow \{\}$
while $R \neq \emptyset$ **do**
    $u \leftarrow \text{ExtractMin}_{\widehat{f}}(R)$; $M \leftarrow M \cup \{u\}$
    if $u = t$ **then return** success
    foreach $v \in N^+(u)$ with $d[v] > d[u] + c(u, v)$ **do**
        $d[v] \leftarrow d[u] + c(u, v)$; $\widehat{f}[v] \leftarrow d[v] + \widehat{h}(v)$; $\pi[v] \leftarrow u$
        $R \leftarrow R \cup \{v\}$; $M \leftarrow M - \{v\}$

return failure

DP-Algorithmus Floyd-Warshall(G)

Input: Azyklischer Graph $G = (V, E, c)$
Output: Minimale Gewichte aller Pfade d

$d^0 \leftarrow c$
for $k \leftarrow 1$ **to** $|V|$ **do**
  for $i \leftarrow 1$ **to** $|V|$ **do**
    for $j \leftarrow 1$ **to** $|V|$ **do**
      $d^k(v_i, v_j) = \min\{d^{k-1}(v_i, v_j), d^{k-1}(v_i, v_k) + d^{k-1}(v_k, v_j)\}$

Laufzeit: $\Theta(|V|^3)$

Bemerkung: Der Algorithmus kann auf einer einzigen Matrix d (in place) ausgeführt werden.

Algorithmus Johnson(G)

Input: Gewichteter Graph $G = (V, E, c)$
Output: Minimale Gewichte aller Pfade D.

Neuer Knoten s. Berechne $G' = (V', E', c')$
if BellmanFord(G', s) = false **then** return "graph has negative cycles"
foreach $v \in V'$ **do**
    $h(v) \leftarrow d(s, v)$ // d aus BellmanFord Algorithmus
foreach $(u, v) \in E'$ **do**
    $\tilde{c}(u, v) \leftarrow c(u, v) + h(u) - h(v)$
foreach $u \in V$ **do**
    $\tilde{d}(u, \cdot) \leftarrow$ Dijkstra$(\tilde{G}', u)$
    foreach $v \in V$ **do**
        $D(u, v) \leftarrow \tilde{d}(u, v) + h(v) - h(u)$

Vergleich der Verfahren

Algorithmus			Laufzeit
Dijkstra (Heap)	$c_v \geq 0$	1:n	$\mathcal{O}(\lvert E\rvert \log \lvert V\rvert)$
Dijkstra (Fibonacci-Heap)	$c_v \geq 0$	1:n	$\mathcal{O}(\lvert E\rvert + \lvert V\rvert \log \lvert V\rvert)$ *
Bellman-Ford		1:n	$\mathcal{O}(\lvert E\rvert \cdot \lvert V\rvert)$
Floyd-Warshall		n:n	$\Theta(\lvert V\rvert^3)$
Johnson		n:n	$\mathcal{O}(\lvert V\rvert \cdot \lvert E\rvert \cdot \log \lvert V\rvert)$
Johnson (Fibonacci-Heap)		n:n	$\mathcal{O}(\lvert V\rvert^2 \log \lvert V\rvert + \lvert V\rvert \cdot \lvert E\rvert)$ *

* amortisiert

Johnson ist besser als Floyd-Warshall für dünn besetzte Graphen ($|E| \approx \Theta(|V|)$).

Union-Find Algorithmus MST-Kruskal(G)

Input: Gewichteter Graph $G = (V, E, c)$
Output: Minimaler Spannbaum mit Kanten A.

Sortiere Kanten nach Gewicht $c(e_1) \leq ... \leq c(e_m)$
$A \leftarrow \emptyset$
for $k = 1$ **to** m **do**
    MakeSet(k)
for $k = 1$ **to** m **do**
    $(u, v) \leftarrow e_k$
    if Find(u) $\neq$ Find(v) **then**
        Union(Find(u), Find(v)) // konzeptuell: $A \leftarrow A \cup e_k$
    else // konzeptuell: $R \leftarrow R \cup e_k$
return (V, A, c)

Implementation Union-Find

Index	1	2	3	4	5	6	7	8	9	10
Parent	**1**	1	1	6	**5**	**6**	5	5	3	**10**

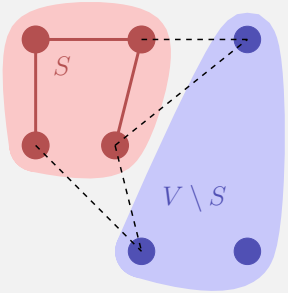
Operationen:

- Make-Set(i): $p[i] \leftarrow i$; **return** i
- Find(i): **while** $(p[i] \neq i)$ **do** $i \leftarrow p[i]$; **return** i
- Union(i, j): $p[j] \leftarrow i$; **return** i

MST Algorithmus von Jarnik, Prim, Dijkstra

Idee: Starte mit einem $v \in V$ und lasse von dort unter Verwendung der Auswahlregel einen Spannbaum wachsen:

$S \leftarrow \{v_0\}$
for $i \leftarrow 1$ **to** $|V|$ **do**
    Wähle billigste (u, v) mit $u \in S$, $v \notin S$
    // konzeptuell $A \leftarrow A \cup \{(u, v)\}$
    $S \leftarrow S \cup \{v\}$ // (Färbung)



Anmerkung: man benötigt keine Union-Find Datenstruktur. Es genügt, Knoten zu färben, sobald sie zu S hinzugenommen werden.

Laufzeit

Trivial $\mathcal{O}(|V| \cdot |E|)$.

Verbesserungen (wie bei Dijskstras ShortestPath):

- Billigeste Kante nach S merken: für jedes $v \in V \setminus S$. Jeweils $\deg^+(v)$ viele Updates für jedes neue $v \in S$. Kosten: $|V|$ viele Minima + Updates: $\mathcal{O}(|V|^2 + \sum_{v \in V} \deg^+(v)) = \mathcal{O}(|V|^2 + |E|)$
- Mit Minheap, Kosten: $|V|$ viele Minima $= \mathcal{O}(|V| \log |V|)$, $|E|$ Updates: $\mathcal{O}(|E| \log |V|)$, Initialisierung $\mathcal{O}(|V|)$: $\mathcal{O}(|E| \cdot \log |V|.)$
- Mit Fibonacci-Heap: $\mathcal{O}(|E| + |V| \cdot \log |V|)$.

3. Programmieraufgabe

Closeness Centrality

- Gegeben: Eine Adjazenzmatrix für einen *ungerichteten* Graphen auf n Knoten.
- Aufgabe: für jeden Knoten v die *Closeness Centrality* $C(v)$ von v.

$$C(v) = \sum_{u \in V \setminus \{v\}} d(v, u)$$

Closeness Centrality

- Gegeben: Eine Adjazenzmatrix für einen *ungerichteten* Graphen auf n Knoten.
- Aufgabe: für jeden Knoten v die *Closeness Centrality* $C(v)$ von v.

$$C(v) = \sum_{u \in V \setminus \{v\}} d(v, u)$$

- Intuitiv: Wenn viele verbundene Knoten nahe bei v liegen, dann ist $C(v)$ klein.
- „Wie zentral ist ein Knoten in seiner Zusammenhangskomponente?“

Alle kürzesten Pfade

- Wir brauchen $d(u, v)$ für alle Knotenpaare (u, v).
- $\Longrightarrow$ berechne alle kürzesten Pfade mit Floyd-Warshall.

```
template<typename Matrix>
void allPairsShortestPaths(unsigned n, Matrix& m)
{
  // your code here
```

- Das Feld `m` soll mit den Distanzen überschrieben werden.
- Achtung: anfangs bedeutet `0` „keine Kante“.
- Ungerichteter Graph: `m[i][j] == m[j][i]`

Closeness Centrality

```
vector<vector<unsigned> > adjacencies(n,
                  vector<unsigned>(n, 0));
vector<string> names(n);
// ...
allPairsShortestPaths(n, adjacencies);
for(unsigned i = 0; i < n; ++i) {
  cout << names[i] << ": ";
  unsigned centrality = 0;
  // your code here
  cout << centrality << endl;
}
```

Closeness Centrality: Eingabedaten

- Der Eingabegraph beschreibt die Zusammenarbeit von gewissen Autoren an wissenschaftlichen Publikationen.
- Die Knoten des Graphen stehen für die Co-Autoren des Mathematikers Paul Erdős.
- Wenn sie zusammen eine Arbeit veröffentlicht haben, sind sie durch eine Kante verbunden.
- Quelle: `https://oakland.edu/enp/thedata/`

Closeness Centrality: Output

```
vertices: 511
ABBOTT, HARVEY LESLIE                : 1625
ACZEL, JANOS D.                      : 1681
AGOH, TAKASHI                        : 2132
AHARONI, RON                         : 1578
AIGNER, MARTIN S.                    : 1589
AJTAI, MIKLOS                        : 1492
ALAOGLU, LEONIDAS*                   : 0
ALAVI, YOUSEF                        : 1561
...
```

Wo kommt die 0 her?

Aufgabe Union-Find

- Input: *union*-Operationen, gefolgt von Anfragen, ob sich zwei Elemente im selben Set befinden.
- Output: Für jede Anfrage, beantworte ob sich die Elemente im selben Set befinden.
- Stelle sicher, dass der Code für die nächste Aufgabe wiederverwendet werden kann.

Aufgabe Kruskals MST-Algorithmus

- Kanten müssen sortiert werden.

Aufgabe Kruskals MST-Algorithmus

- Kanten müssen sortiert werden.
- Erstelle eine *Edge*-Klasse, die Vergleichsoperator implementiert.
- Dann verwende *std::sort*.

Fragen?