Datenstrukturen und Algorithmen

Übung 9

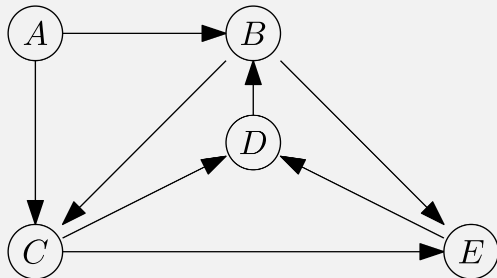
FS 2018

Programm von heute

1. Feedback letzte Übung
2. Wiederholung Theorie
3. Programmieraufgabe

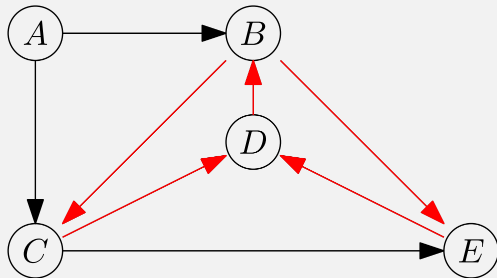
1. Feedback letzte Übung

Topologische Sortierung



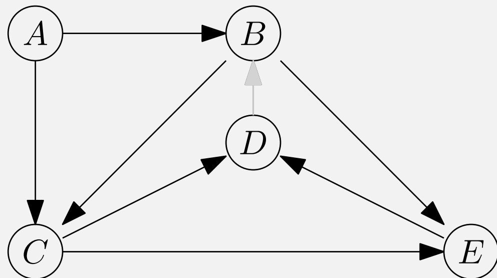
- der Graph hat Kreise

Topologische Sortierung



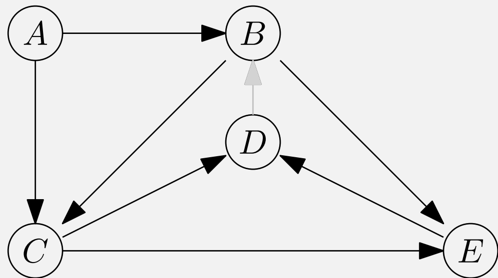
- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante

Topologische Sortierung



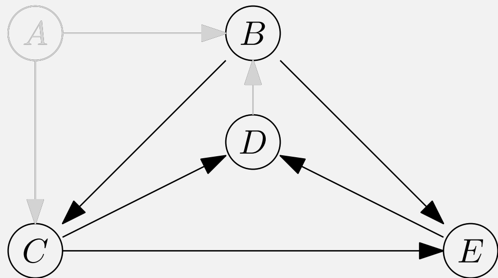
- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante
- entferne die Kante
  $\Longrightarrow$ kreisfrei

Topologische Sortierung



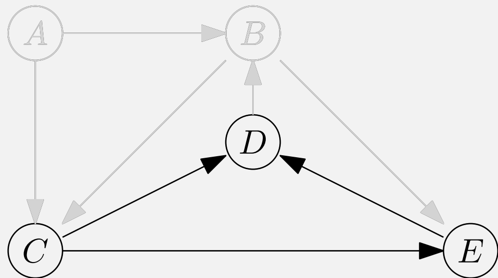
- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante
- entferne die Kante $\implies$ kreisfrei
- topologische Sortierung durch „Entfernen“ von Knoten mit Eingangsgrad 0

Topologische Sortierung



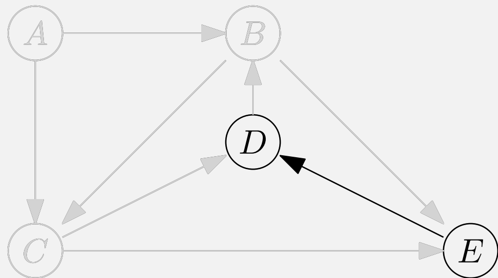
- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante
- entferne die Kante $\Longrightarrow$ kreisfrei
- topologische Sortierung durch „Entfernen“ von Knoten mit Eingangsgrad 0

Topologische Sortierung



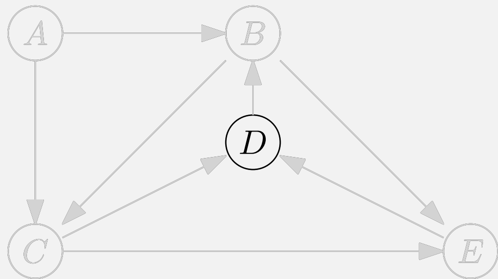
- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante
- entferne die Kante $\implies$ kreisfrei
- topologische Sortierung durch „Entfernen“ von Knoten mit Eingangsgrad 0

Topologische Sortierung



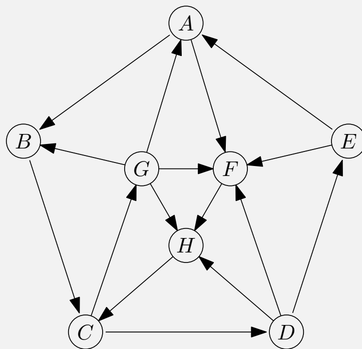
- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante
- entferne die Kante $\Longrightarrow$ kreisfrei
- topologische Sortierung durch „Entfernen“ von Knoten mit Eingangsgrad 0

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

Tiefen- und Breitensuche

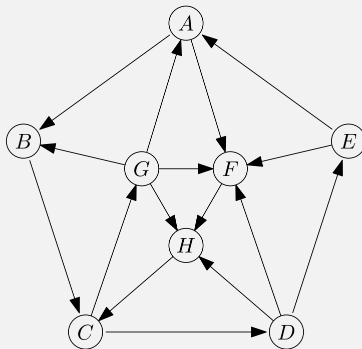


Start bei A

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

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

Tiefen- und Breitensuche



Start bei A

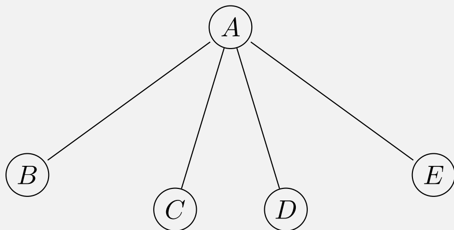
DFS: A, B, C, D, E, F, H, G

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

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

Tiefen- und Breitensuche

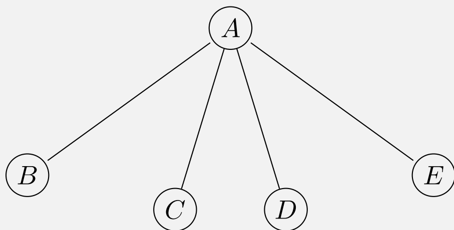
Stern: DFS-Ordnung entspricht BFS-Ordnung



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

Huffman Node

```
using SharedNode=std::shared_ptr<Node>;
struct Node{
    char value;
    int frequency;
    SharedNode left;
    SharedNode right;

    // constructor for leafs
    Node(char v, int f): value{v}, frequency{f},
        left{nullptr}, right{nullptr} {}
    // constructor for inner nodes
    Node(SharedNode l, SharedNode r): value{0},
        frequency{l->frequency + r->frequency}, left{l}, right{r} {};
};
```

Huffman Code- Frequencies

```
std::map<char, int> m;
char x; int n = 0;
while (in.get(x)){
    ++m[x]; ++n;
}
std::cout << "n = " << n << " characters" << std::endl;
```

Huffman Code - Heap

```
struct comparator {
 bool operator()(const SharedNode a, const SharedNode b) const {
       return a->frequency > b->frequency;
 }
};
       ...

    // build heap
    std::priority_queue<SharedNode, std::vector<SharedNode>, comparato
    for (auto y: m){
        q.push(std::make_shared<Node>(y.first, y.second));
    }
```

Huffman Code – Tree

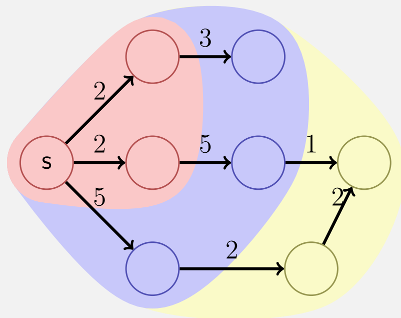
```
// build code tree
SharedNode left;
while (!q.empty()){
    left = q.top();q.pop();
    if (!q.empty()){
        auto right = q.top();q.pop();
        q.push(std::make_shared<Node>(left, right));
    }
}
```

2. Wiederholung Theorie

Dijkstra ShortestPath Grundidee

Menge V aller Knoten wird unterteilt in

- die Menge M von Knoten, für die schon ein kürzester Weg von s bekannt ist
- die Menge $R = \bigcup_{v \in M} N^{+}(v) \setminus M$ von Knoten, für die kein kürzester Weg bekannt ist, die jedoch von M direkt erreichbar sind.
- die Menge $U = V \setminus (M \cup R)$ von Knoten, die noch nicht berücksichtigt wurden.



Algorithmus Dijkstra

Initial: $\mathrm{PL}(n) \leftarrow \infty$ für alle Knoten.

- Setze $\mathrm{PL}(s) \leftarrow 0$
- Starte mit $M = \{s\}$. Setze $k \leftarrow s$.
- Solange ein neuer Knoten k hinzukommt und dieser nicht der Zielknoten ist
  1. Für jeden Nachbarknoten n von k:
     - Berechne Pfadlänge x nach n über k
     - Wenn $\mathrm{PL}(n) = \infty$, so nimm n zu R hinzu
     - Ist $x < \mathrm{PL}(n) < \infty$, so setze $\mathrm{PL}(n) \leftarrow x$ und passe R an.
  2. Wähle als neuen Knoten k den mit kleinster Pfadlänge in R.

Allgemeine Bewertete Graphen

Verbesserungsschritt wie bei Dijkstra:

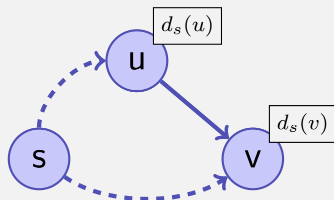
Relax(u, v) $(u, v \in V, (u, v) \in E)$

if $d_s(v) > d_s(u) + c(u, v)$ **then**

  $d_s(v) \leftarrow d_s(u) + c(u, v)$

  return true

return false



Problem: Zyklen mit negativen Gewichten können Weg verkürzen: es muss keinen kürzesten Weg mehr geben

Dynamic-Programming-Ansatz (Bellman)

Induktion über Anzahl Kanten. $d_s[i,v]$: Kürzeste Weglänge von s nach v über maximal i Kanten.

$$d_s[i,v] = \min\{d_s[i-1,v], \min_{(u,v)\in E}(d_s[i-1,u] + c(u,v))$$

$$d_s[0,s] = 0, d_s[0,v] = \infty \; \forall v \neq s.$$

DP Induktion für alle kürzesten Pfade

$d^k(u,v) =$ Minimales Gewicht eines Pfades $u \rightsquigarrow v$ mit Zwischenknoten aus V^k

Induktion

$$d^k(u,v) = \min\{d^{k-1}(u,v), d^{k-1}(u,k) + d^{k-1}(k,v)\}(k \geq 1)$$
$$d^0(u,v) = c(u,v)$$

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

* amortisiert

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**
    if $(V, A \cup \{e_k\})$ kreisfrei **then**
        $A \leftarrow E' \cup \{e_k\}$

return (V, A, c)

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).
- $\implies$ 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?

Fragen?