Datenstrukturen und Algorithmen

Übungen 9-10

FS 2019

Programm von heute

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

1. Feedback letzte Übungen

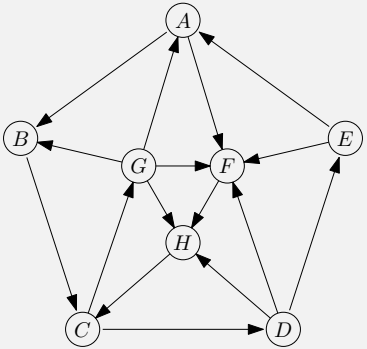
Levenshtein Distance

```
// D[n,m] = distance between x and y
// D[i,j] = distance between strings x[1..i] and y[1..j]
vector<vector<unsigned>> D(n+1,vector<unsigned>(m+1,0));
for (unsigned j = 0; j <=m; ++j)
  D[0][j] = j;
for (unsigned i = 1; i <= n; ++i){
  D[i][0] = i;
  for (unsigned j = 1; j <=m; ++j){
    unsigned q = D[i-1][j-1] + (x[i-1]!=y[j-1]);
    q = std::min(q,D[i][j-1]+1);
    q = std::min(q,D[i-1][j]+1);
    D[i][j] = q;
  }
}
return D[n][m];
```

Traveling Salesman

siehe Musterlösung mit detaillierten Kommentaren

Tiefen- und Breitensuche

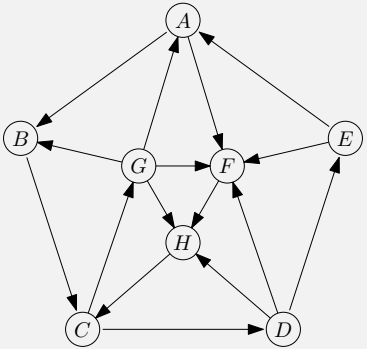


Start bei A

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

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

Tiefen- und Breitensuche



Start bei A

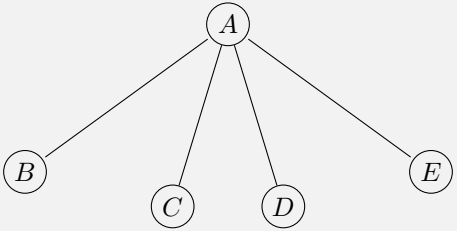
DFS: A, B, C, D, E, F, H, G

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

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

Tiefen- und Breitensuche

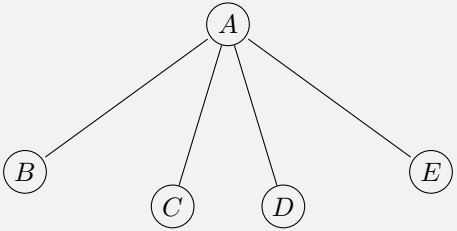
Stern: DFS-Ordnung entspricht BFS-Ordnung



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

Tiefen- und Breitensuche

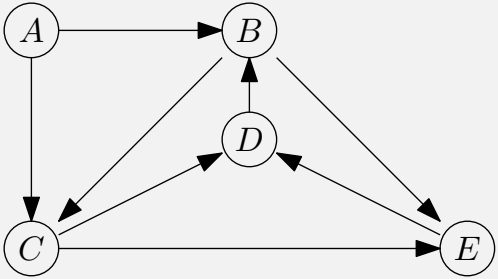
Stern: DFS-Ordnung entspricht BFS-Ordnung



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

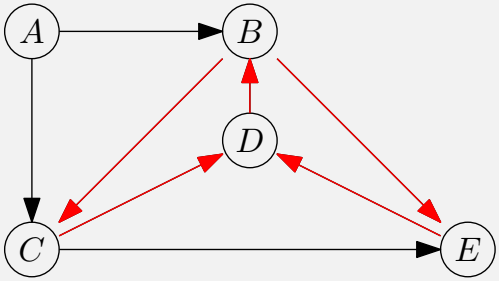
Start bei C
DFS: C, A, B, D, E
BFS: C, A, B, D, E

Topologische Sortierung



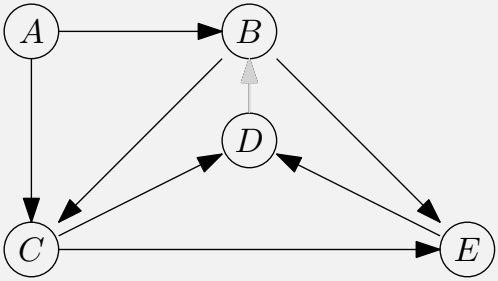
- der Graph hat Kreise

Topologische Sortierung



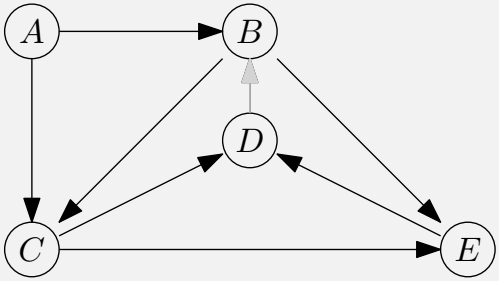
- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante

Topologische Sortierung



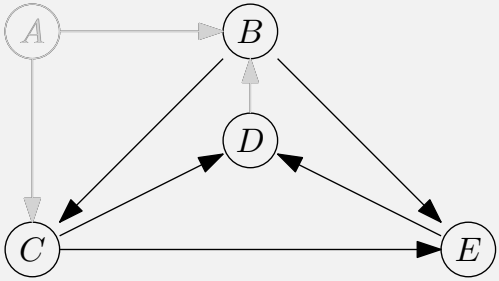
- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante
- entferne die Kante
  $\Longrightarrow$ kreisfrei

Topologische Sortierung



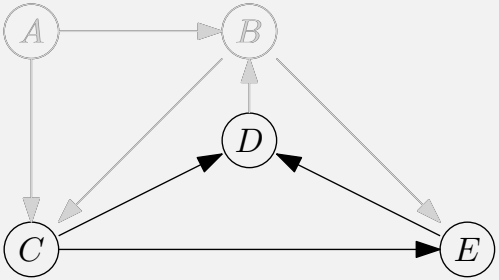
- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante
- entferne die Kante
  $\Longrightarrow$ kreisfrei
- topologische Sortierung durch „Entfernen“ von Knoten mit Eingangsgrad 0

Topologische Sortierung



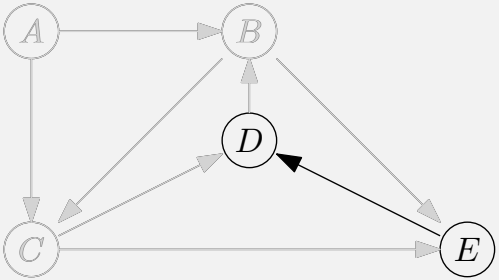
- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante
- entferne die Kante
  $\Longrightarrow$ kreisfrei
- topologische Sortierung durch „Entfernen“ von Knoten mit Eingangsgrad 0

Topologische Sortierung



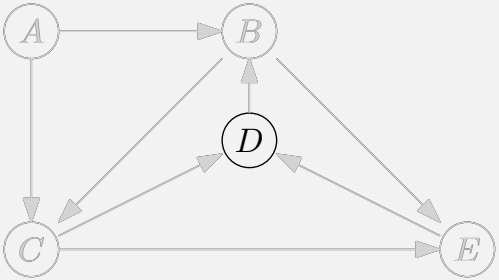
- der Graph hat Kreise
- zwei minimale Kreise teilen sich eine Kante
- entferne die Kante
  $\Longrightarrow$ kreisfrei
- topologische Sortierung durch „Entfernen“ von Knoten mit Eingangsgrad 0

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

Huffman Code- Frequencies: Hashmap!

```
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 - Nodes: SharedPointers on a Heap

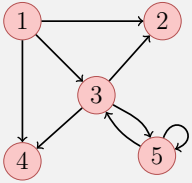
```
struct comparator {
 bool operator()(const SharedNode a, const SharedNode b) const {
       return a->frequency > b->frequency;
 }
};
...

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

Huffman Code – Tree: SharedPointers in 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

Adjazenzmatrizen multipliziert



$$B := A_G^2 = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}^2 = \begin{pmatrix} 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 2 \end{pmatrix}$$

Interpretation

Theorem

Sei $G = (V, E)$ ein Graph und $k \in \mathbb{N}$. Dann gibt das Element $a_{i,j}^{(k)}$ der Matrix $(a_{i,j}^{(k)})_{1 \leq i,j \leq n} = A_G^k$ die Anzahl der Wege mit Länge k von v_i nach v_j an.

Graphen und Relationen

Graph $G = (V, E)$ mit Adjazenzen $A_G \mathrel{\widehat{=}}$ Relation $E \subseteq V \times V$ auf V

- *reflexiv* $\Leftrightarrow a_{i,i} = 1$ für alle $i = 1, \ldots, n$.
- *symmetrisch* $\Leftrightarrow a_{i,j} = a_{j,i}$ für alle $i, j = 1, \ldots, n$ (ungerichtet)
- *transitiv* $\Leftrightarrow$ $(u, v) \in E$, $(v, w) \in E \Rightarrow (u, w) \in E$.

Äquivalenzrelation $\Leftrightarrow$ Kollektion vollständiger, ungerichteter Graphen, für den jedes Element eine Schleife hat.

Reflexive transitive Hülle von G $\Leftrightarrow$ *Erreichbarkeitsrelation* E^*:
$(v, w) \in E^*$ gdw. $\exists$ Weg von Knoten v zu w.

Algorithmus ReflexiveTransitiveClosure(A_G)

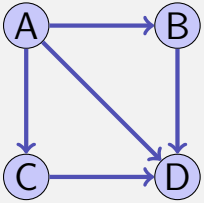
Input: Adjazenzmatrix $A_G = (a_{ij})_{i,j=1}^{n}$
Output: Reflexive Transitive Hülle $B = (b_{ij})_{i,j=1}^{n}$ von G

$B \leftarrow A_G$
for $k \leftarrow 1$ **to** n **do**
    $a_{kk} \leftarrow 1$ // Reflexivität
    for $i \leftarrow 1$ **to** n **do**
        for $j \leftarrow 1$ **to** n **do**
            $b_{ij} \leftarrow \max\{b_{ij}, b_{ik} \cdot b_{kj}\}$ // Alle Wege über v_k
return B

= *Algorithmus von Warshall*. Vergleiche Algorithmus Floyd-Warshall: Kürzeste Pfade für alle Knotenpaare

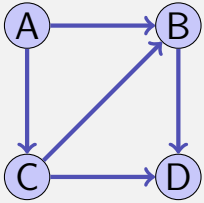
Quiz: Topologisch Sortieren

Auf wie viele Arten können die folgenden gerichteten Graphen jeweils topologisch sortiert werden?



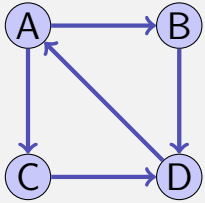
Anzahl Sortierungen

?



Anzahl Sortierungen

?

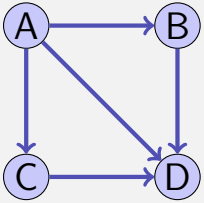


Anzahl Sortierungen

?

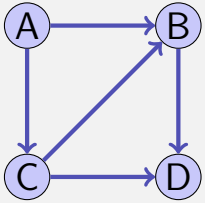
Quiz: Topologisch Sortieren

Auf wie viele Arten können die folgenden gerichteten Graphen jeweils topologisch sortiert werden?



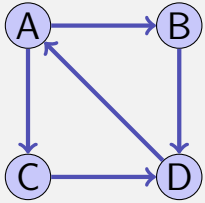
Anzahl Sortierungen

2



Anzahl Sortierungen

1



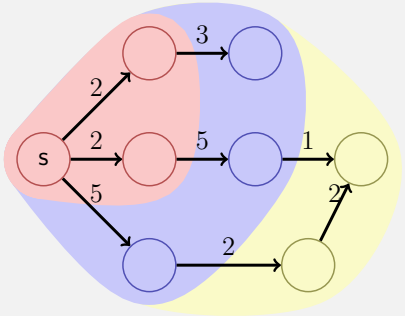
Anzahl Sortierungen

0

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 $R = \cup_{v \in M} N^+(v) \setminus M$ 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

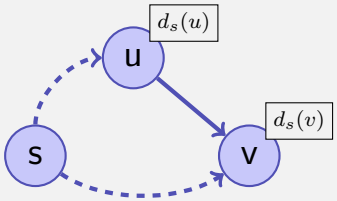
Initial: $\mathrm{PL}(n) \leftarrow \infty$ für alle Knoten.

- 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	$\mathcal{O}(\lvert E\rvert \log \lvert V\rvert)$
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 ($|E| \approx \Theta(|V|)$).

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?

Fragen?