Datenstrukturen und Algorithmen

Exercises 9-10

FS 2019

Program of today

1. Feedback of last exercises
2. Recap Theory
3. Programming Task

1. Feedback of last exercises

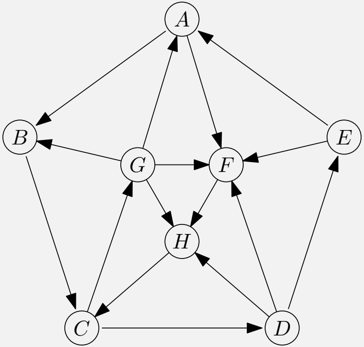
Levenshtein Distance

```
// 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

see master solution with detailed comments

Depth-first-search and Breadth-first-search

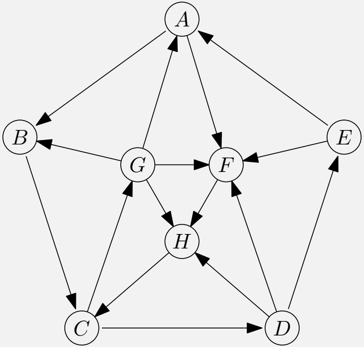


Starting at A

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

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

Depth-first-search and Breadth-first-search



Starting at A

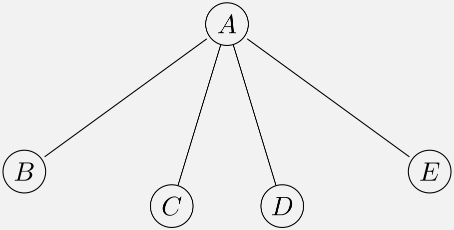
DFS: A, B, C, D, E, F, H, G

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

There is no starting vertex where the DFS ordering equals the BFS ordering.

Depth-first-search and Breadth-first-search

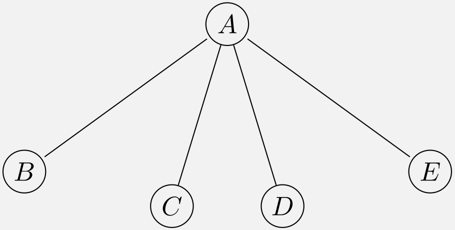
Star: DFS ordering equals BFS ordering



Starting at A
DFS: A, B, C, D, E
BFS: A, B, C, D, E

Depth-first-search and Breadth-first-search

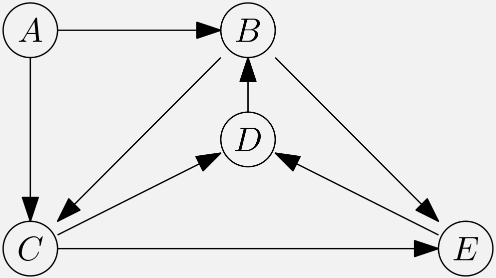
Star: DFS ordering equals BFS ordering



Starting at A
DFS: A, B, C, D, E
BFS: A, B, C, D, E

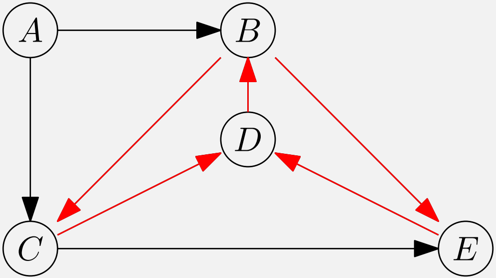
Starting at C
DFS: C, A, B, D, E
BFS: C, A, B, D, E

Topological Sorting



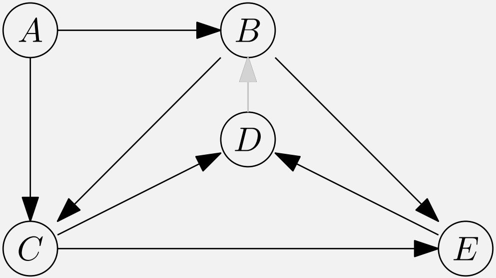
- Graph with cycles

Topological Sorting



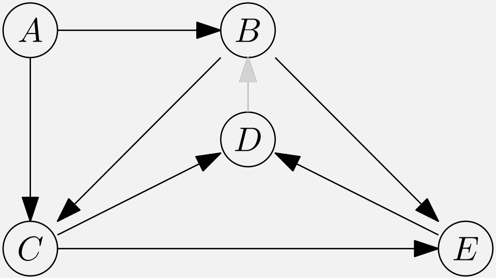
- Graph with cycles
- Two minimal cycles sharing an edge

Topological Sorting



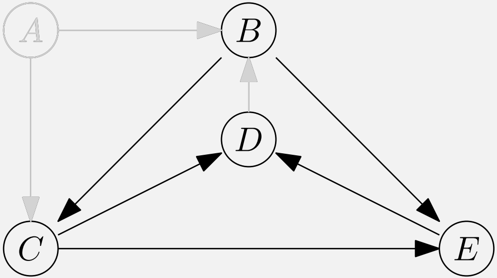
- Graph with cycles
- Two minimal cycles sharing an edge
- Remove edge $\implies$ cycle-free

Topological Sorting



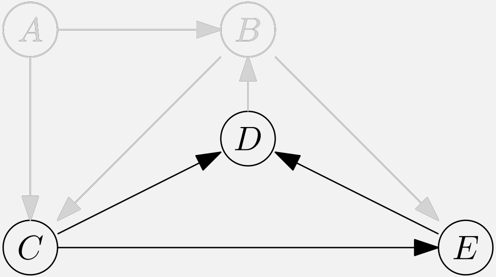
- Graph with cycles
- Two minimal cycles sharing an edge
- Remove edge $\implies$ cycle-free
- Topological Sorting by "removing" elements with in-degree 0

Topological Sorting



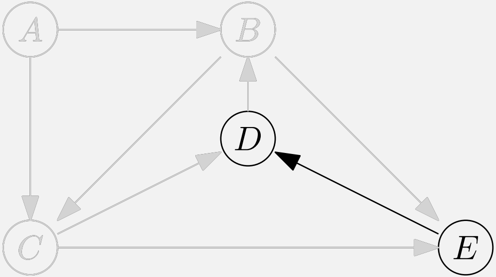
- Graph with cycles
- Two minimal cycles sharing an edge
- Remove edge $\Longrightarrow$ cycle-free
- Topological Sorting by "removing" elements with in-degree 0

Topological Sorting



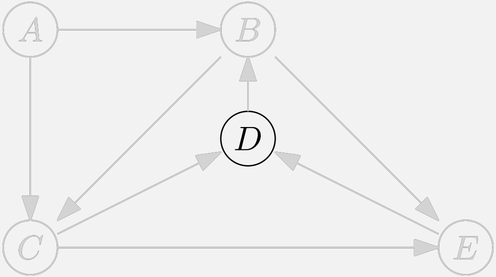
- Graph with cycles
- Two minimal cycles sharing an edge
- Remove edge $\implies$ cycle-free
- Topological Sorting by “removing” elements with in-degree 0

Topological Sorting



- Graph with cycles
- Two minimal cycles sharing an edge
- Remove edge $\implies$ cycle-free
- Topological Sorting by "removing" elements with in-degree 0

Topological Sorting



- Graph with cycles
- Two minimal cycles sharing an edge
- Remove edge $\implies$ cycle-free
- Topological Sorting by "removing" elements with in-degree 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

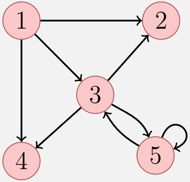
```
struct comparator {
 bool operator()(const SharedNode a, const SharedNode b) const {
        return a->frequency > b->frequency;
 }
};
...

// 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. Recap Theory

Adjacency Matrix Product



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

Interpretation

Theorem

Let $G = (V, E)$ *be a graph and* $k \in \mathbb{N}$*. Then the element* $a_{i,j}^{(k)}$ *of the matrix* $(a_{i,j}^{(k)})_{1 \leq i,j \leq n} = A_G^k$ *provides the number of paths with length* k *from* v_i *to* v_j *.*

Graphs and Relations

Graph $G = (V, E)$ with adjacencies $A_G \mathrel{\widehat{=}}$ Relation $E \subseteq V \times V$ over V

- *reflexive* $\Leftrightarrow a_{i,i} = 1$ for all $i = 1, \ldots, n$.
- *symmetric* $\Leftrightarrow a_{i,j} = a_{j,i}$ for all $i, j = 1, \ldots, n$ (undirected)
- *transitive* $\Leftrightarrow (u, v) \in E$, $(v, w) \in E \Rightarrow (u, w) \in E$.

Equivalence relation $\Leftrightarrow$ collection of complete, undirected graphs where each element has a loop.

Reflexive transitive closure of $G \Leftrightarrow$ *Reachability relation* E^*:
$(v, w) \in E^*$ iff $\exists$ path from node v to w.

Algorithm ReflexiveTransitiveClosure(A_G)

Input: Adjacency matrix $A_G = (a_{ij})_{i,j=1}^n$
Output: Reflexive transitive closure $B = (b_{ij})_{i,j=1}^n$ of G

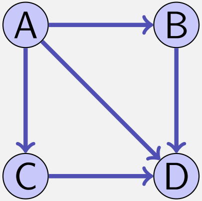
$B \leftarrow A_G$
for $k \leftarrow 1$ **to** n **do**
    $a_{kk} \leftarrow 1$ // Reflexivity
    for $i \leftarrow 1$ **to** n **do**
        for $j \leftarrow 1$ **to** n **do**
            $b_{ij} \leftarrow \max\{b_{ij}, b_{ik} \cdot b_{kj}\}$ // All paths via v_k

return B

= Warshall algorithm. Cf algorithm Floyd-Warshall: shortest paths for all point pairs

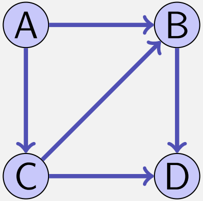
Quiz: Topological Sorting

In how many ways can the following directed graphs be topologically sorted each?



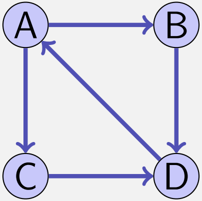
number sortings

?



number sortings

?

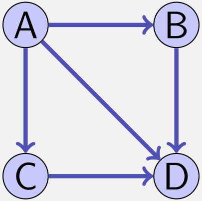


number sortings

?

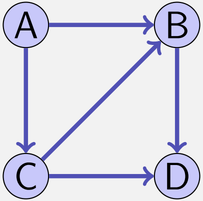
Quiz: Topological Sorting

In how many ways can the following directed graphs be topologically sorted each?



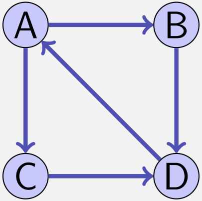
number sortings

2



number sortings

1



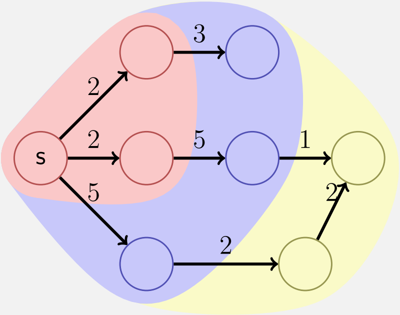
number sortings

0

Dijkstra ShortestPath Basic Idea

Set V of nodes is partitioned into

- the set M of nodes for which a shortest path from s is already known,
- the set $R = \bigcup_{v \in M} N^+(v) \setminus M$ of nodes where a shortest path is not yet known but that are accessible directly from M,
- the set $U = V \setminus (M \cup R)$ of nodes that have not yet been considered.



Algorithm Dijkstra

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

- Set $\mathrm{PL}(s) \leftarrow 0$
- Start with $M = \{s\}$. Set $k \leftarrow s$.
- While a new node k is added and this is not the target node
  1. For each neighbour node n of k:
     - compute path length x to n via k
     - If $\mathrm{PL}(n) = \infty$, than add n to R
     - If $x < \mathrm{PL}(n) < \infty$, then set $\mathrm{PL}(n) \leftarrow x$ and adapt R .
  2. Choose as new node k the node with smallest path length in R.

General Weighted Graphs

Relaxing Step as with Dijkstra:

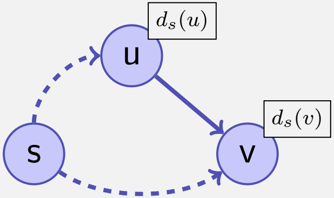
Relax(u, v) $(u, v \in V, (u, v) \in E)$

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: cycles with negative weights can shorten the path, a shortest path is not guaranteed to exist.

Dynamic Programming Approach (Bellman)

Induction over number of edges $d_s[i,v]$: Shortest path from s to v via maximally i edges.

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

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

DP Induction for all shortest paths

$d^k(u, v) =$ Minimal weight of a path $u \rightsquigarrow v$ with intermediate nodes in 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 Algorithm Floyd-Warshall(G)

Input: Acyclic Graph $G = (V, E, c)$
Output: Minimal weights of all paths 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**
            $d^k(v_i, v_j) = \min\{d^{k-1}(v_i, v_j), d^{k-1}(v_i, v_k) + d^{k-1}(v_k, v_j)\}$

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

Remark: Algorithm can be executed with a single matrix d (in place).

Algorithm Johnson(G)

Input: Weighted Graph $G = (V, E, c)$
Output: Minimal weights of all paths D.

New node s. Compute $G' = (V', E', c')$
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**
        $D(u, v) \leftarrow \tilde{d}(u, v) + h(v) - h(u)$

Comparison of the approaches

Algorithm			Runtime
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	$\mathcal{O}(\lvert V\rvert^2 \log \lvert V\rvert + \lvert V\rvert \cdot \lvert E\rvert)$ *

* amortized

Johnson is better than Floyd-Warshall for sparse graphs ($|E| \approx \Theta(|V|)$).

3. Programming Task

Closeness Centrality

- Given: an adjacency matrix for an *undirected* graph on n vertices.
- Output: the *closeness centrality* $C(v)$ of every vertex v.

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

Closeness Centrality

- Given: an adjacency matrix for an *undirected* graph on n vertices.
- Output: the *closeness centrality* $C(v)$ of every vertex v.

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

- Intuition: If many connected vertices are close to v, then $C(v)$ is small.
- "How central is the vertex in its connected component?"

All Pairs Shortest Paths

- We require $d(u, v)$ for all vertex pairs (u, v).
- $\implies$ compute all shortest paths using Floyd-Warshall.

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

- Simply overwrite `m` with the distance values.
- Attention: initially `0` means "no edge".
- Undirected 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: Input Data

- A graph that stems from collaborations on scientific papers.
- The vertices of the graph are the co-authors of the mathematician Paul Erdős.
- There is an edge between them if the authors have jointly published a paper.
- Source: `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
...
```

Where does the 0 come from?

Questions?