ETH zürich

Exercise Session 11

Data Structures and Algorithms, D-MATH, ETH Zurich

Program Today

Feedback of last exercise

Repetition of Lecture
- All Pairs Shortest Paths
- Kruskal

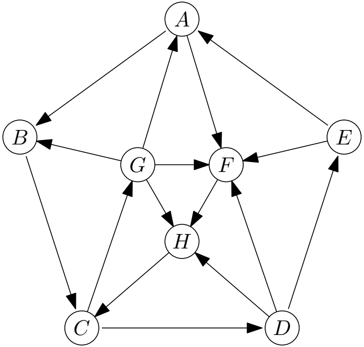
Hints for current tasks
- Closeness Centrality
- TSP

In-Class-Exercise practical

In-Class-Exercise (theoretical)

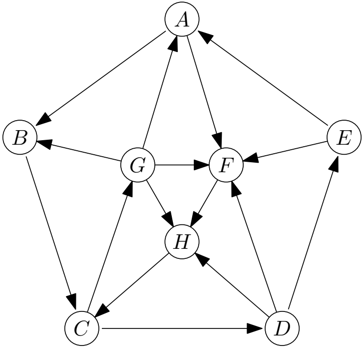
1. Feedback of last exercise

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

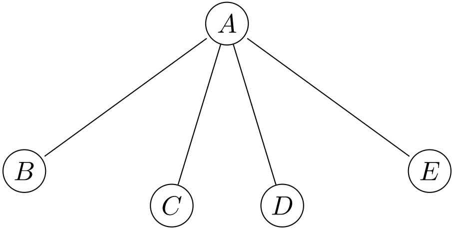
DFS: A, B, C, D, E, F, H, G

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

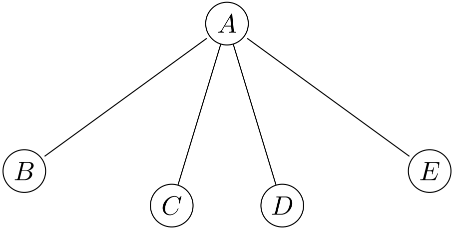
Star: DFS ordering equals BFS ordering



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

Depth-first-search and Breadth-first-search

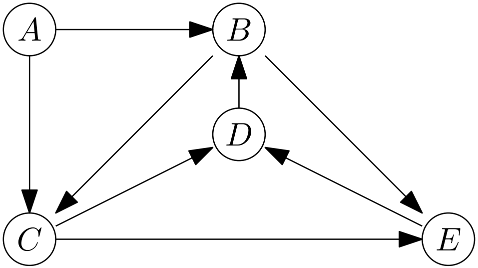
Star: DFS ordering equals BFS ordering



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

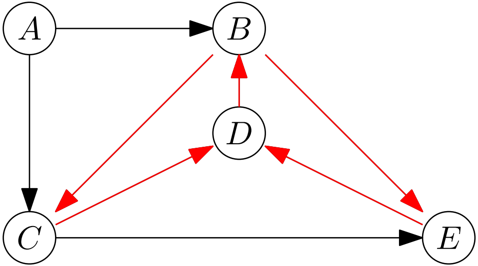
Starting at C
DFS: C, A, B, D, E
BFS: C, A, B, D, E

Topological Sorting



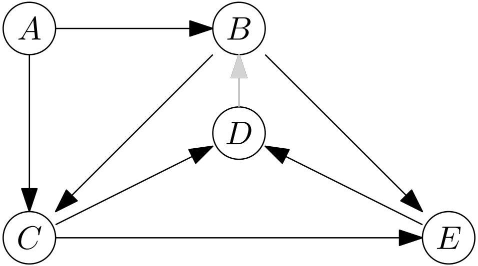
- Graph with cycles

Topological Sorting



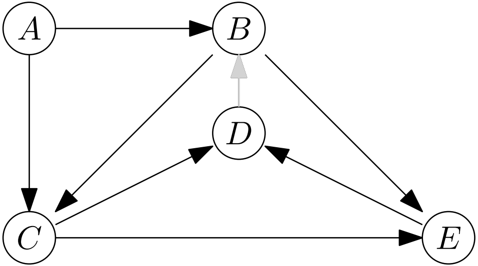
- Graph with cycles
- Two minimal cycles sharing an edge

Topological Sorting



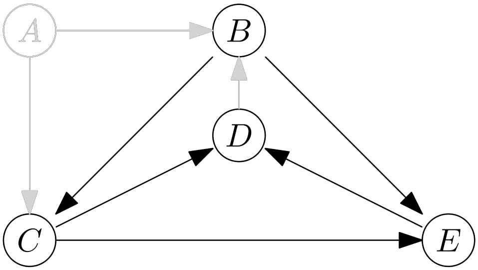
- Graph with cycles
- Two minimal cycles sharing an edge
- Remove edge $\implies$ cycle-free

Topological Sorting



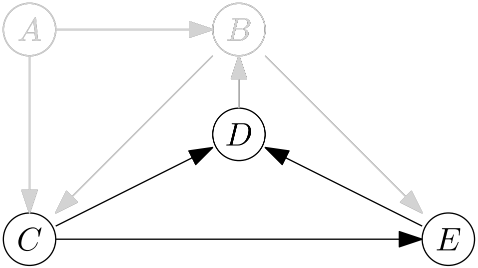
- Graph with cycles
- Two minimal cycles sharing an edge
- Remove edge $\implies$ cycle-free
- Topological Sorting by "removing" elements with in-degree 0

Topological Sorting



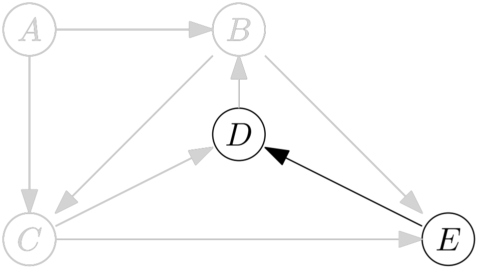
- Graph with cycles
- Two minimal cycles sharing an edge
- Remove edge $\Longrightarrow$ cycle-free
- Topological Sorting by "removing" elements with in-degree 0

Topological Sorting



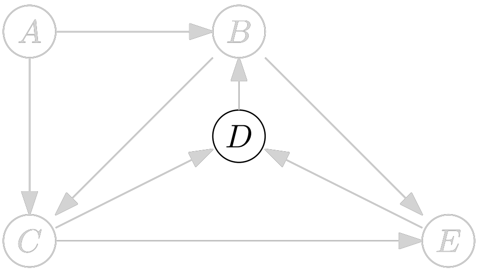
- Graph with cycles
- Two minimal cycles sharing an edge
- Remove edge $\implies$ cycle-free
- Topological Sorting by "removing" elements with in-degree 0

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

2. Repetition of Lecture

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).

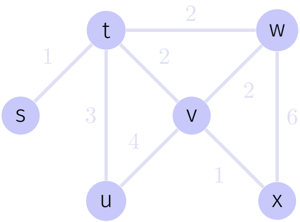
Comparison of the approaches

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

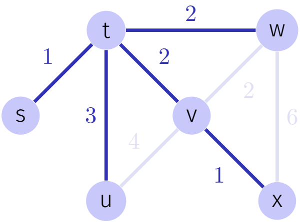
* amortized

Johnson (not explained this year) is better than Floyd-Warshall only for sparse graphs ($\lvert E\rvert \approx \Theta(\lvert V\rvert)$).

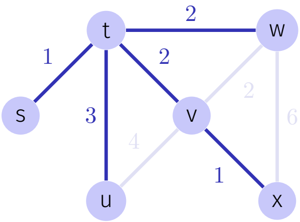
Minimum Spanning Trees

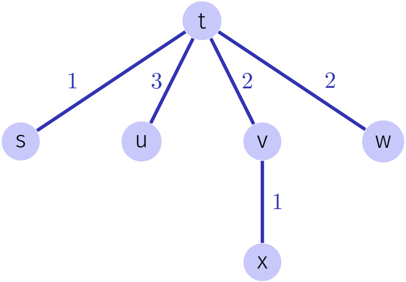


Minimum Spanning Trees

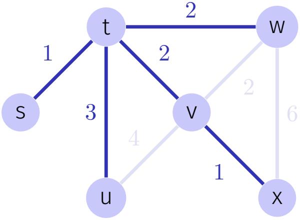


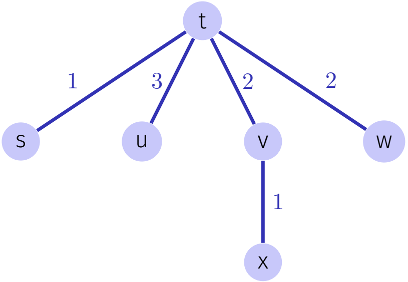
Minimum Spanning Trees





Minimum Spanning Trees





(Solution is not unique.)

MakeSet, Union, and Find

- Make-Set(i): create a new set represented by i.
- Find(e): name of the set i that contains e.
- Union(i, j): union of the sets with names i and j.

MakeSet, Union, and Find

- Make-Set(i): create a new set represented by i.
- Find(e): name of the set i that contains e .
- Union(i, j): union of the sets with names i and j.

In MST-Kruskal:

- Make-Set(i): New tree with i as root.
- Find(e): Find root of e
- Union(i, j): Union of the trees i and j.

Algorithm MST-Kruskal(G)

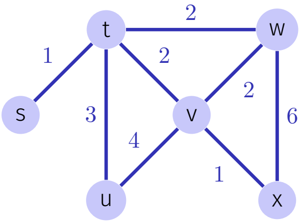
Input: Weighted Graph $G = (V, E, c)$
Output: Minimum spanning tree with edges A.

Sort edges by weight $c(e_1) \leq \ldots \leq c(e_m)$
$A \leftarrow \emptyset$
for $k = 1$ **to** $|V|$ **do**
    MakeSet(k)

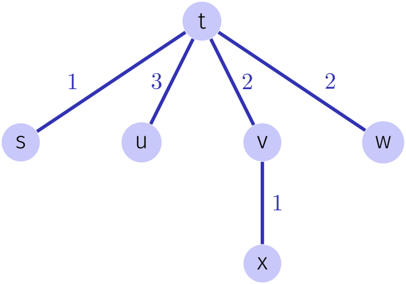
for $k = 1$ **to** m **do**
    $(u, v) \leftarrow e_k$
    if Find(u) $\neq$ Find(v) **then**
        Union(Find(u), Find(v))
        $A \leftarrow A \cup e_k$

return (V, A, c)

Representation as array



Index	s	t	w	v	u	x



Index	s	t	u	v	w	x
Parent	t	t	t	t	t	v

Different kind of improvement

Link all nodes to the root when Find is called.

Find(i):

$j \leftarrow i$

while $(p[i] \neq i)$ **do** $i \leftarrow p[i]$

while $(j \neq i)$ **do**

  $t \leftarrow j$

  $j \leftarrow p[j]$

  $p[t] \leftarrow i$

return i

Cost: amortised *nearly* constant (inverse of the Ackermann-function).[1]

[1] We do not go into details here.

Running time of Kruskal's Algorithm

- Sorting of the edges: $\Theta(|E|\log|E|) = \Theta(|E|\log|V|)$. [2]
- Initialisation of the Union-Find data structure $\Theta(|V|)$
- $|E|\times$ Union(Find(x),Find(y)): $\mathcal{O}(|E|\log|E|) = \mathcal{O}(|E|\log|V|)$.

Overal $\Theta(|E|\log|V|)$.

[2] because G is connected: $|V| \leq |E| \leq |V|^2$

3. Hints for current tasks

Closeness Centrality, TSP

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. (`APSH.h`)

```
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

Centrality.h

```
void printCentrality(unsigned n, vector<vector<unsigned>>
   adjacencies, vector<string> names)
{
  for(unsigned i = 0; i < n; ++i)
  {
    cout << names[i] << ": ";
    unsigned centrality = 0;
    // TODO: compute centrality of vertex i 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?

Travelling Salesperson Problem

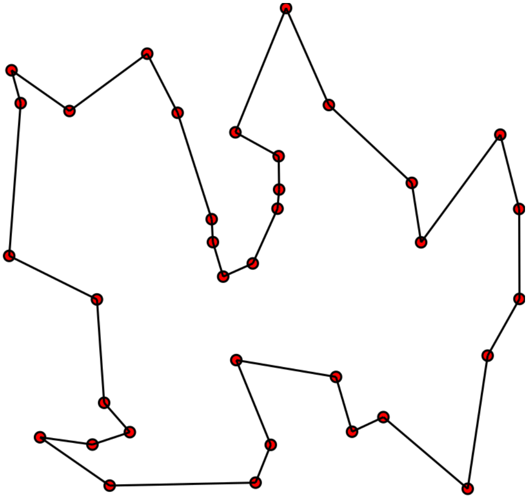
Problem

Given a map and list of cities, what is the shortest possible route that visits each city once and returns at the original city?

Mathematical model

On an undirected, weighted graph G, which cycle containing all of G's vertices has the lowest weight sum?

Travelling Salesperson Problem



Travelling Salesperson Problem

- The problem has no known polynomial-time solution.
- Many heuristic algorithms exists. They do not always return the optimal solution.

Travelling Salesperson Problem

- The heuristic algorithm that you are asked to implement on CodeExpert (*The Travelling Student*) on CodeExpert uses an MST:
  1. Compute the minimum spanning tree M
  2. Make a depth first search on M
- The algorithm is 2-approximate, meaning that the solution it generates has at most twice the cost of the optimal solution.
- The algorithm assumes a complete graph $G = (V, E, c)$that satisfies the triangle inequality: $c(v, w) \leq c(v, x) + c(x, w) \; \forall \; v, w, x \in V$

4. In-Class-Exercise practical

Union-Find Experiments (Code-Expert)

5. In-Class-Exercise (theoretical)

In-Class-Exercises: Longest Path in DAGs

Finding a shortest path is easy (BFS, Dijkstra, Bellman-Ford). Finding a long path is incredibly hard! For directed graphs, nobody knows how to even efficiently find paths of length $\gg \log^2 n$.

In-Class-Exercises: Longest Path in DAGs

Finding a shortest path is easy (BFS, Dijkstra, Bellman-Ford). Finding a long path is incredibly hard! For directed graphs, nobody knows how to even efficiently find paths of length $\gg \log^2 n$.

Exercise:

You are given a directed, **acyclic** graph (DAG) $G = (V, E)$.

Design an $\mathcal{O}(|V| + |E|)$-time algorithm to find the longest path.

In-Class-Exercises: Longest Path in DAGs

Finding a shortest path is easy (BFS, Dijkstra, Bellman-Ford). Finding a long path is incredibly hard! For directed graphs, nobody knows how to even efficiently find paths of length $\gg \log^2 n$.

Exercise:

You are given a directed, **acyclic** graph (DAG) $G = (V, E)$.

Design an $\mathcal{O}(|V| + |E|)$-time algorithm to find the longest path.

Hint: G is acyclic, meaning you can topologically sort G.

In-Class-Exercises: Longest Path in DAGs

Solution:

1. Topological Sorting. Running time: $\mathcal{O}(|V| + |E|)$.

In-Class-Exercises: Longest Path in DAGs

Solution:

1. Topological Sorting. Running time: $\mathcal{O}(|V| + |E|)$.
2. Compute for each node all incoming edges: $\mathcal{O}(|V| + |E|)$.

In-Class-Exercises: Longest Path in DAGs

Solution:

1. Topological Sorting. Running time: $\mathcal{O}(|V| + |E|)$.
2. Compute for each node all incoming edges: $\mathcal{O}(|V| + |E|)$.
3. Visit each node v in topological order and consider all incoming edges: $\mathcal{O}(|V| + |E|)$.

In-Class-Exercises: Longest Path in DAGs

Solution:

1. Topological Sorting. Running time: $\mathcal{O}(|V| + |E|)$.
2. Compute for each node all incoming edges: $\mathcal{O}(|V| + |E|)$.
3. Visit each node v in topological order and consider all incoming edges: $\mathcal{O}(|V| + |E|)$.

$$\texttt{dist}[v] = \begin{cases} 0 & \text{no incoming edges,} \\ \max\limits_{(u,v)\in E} \{\texttt{dist}[u] + c(u,v)\} & \text{otherwise.} \end{cases}$$

In-Class-Exercises: Longest Path in DAGs

Solution:

1. Topological Sorting. Running time: $\mathcal{O}(|V| + |E|)$.
2. Compute for each node all incoming edges: $\mathcal{O}(|V| + |E|)$.
3. Visit each node v in topological order and consider all incoming edges: $\mathcal{O}(|V| + |E|)$.

   $$\texttt{dist}[v] = \begin{cases} 0 & \text{no incoming edges,} \\ \max\limits_{(u,v)\in E} \{\texttt{dist}[u] + c(u,v)\} & \text{otherwise.} \end{cases}$$

   Store predecessor!