Datenstrukturen und Algorithmen

Exercise 10

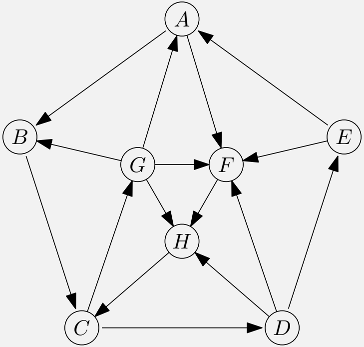
FS 2021

Program of today

1. Feedback of last exercises
2. Shortest Paths
   - Dijkstra
   - Heaps, DecreaseKey and Lazy Deletion
   - Running Time of the Algorithms
   - Dijkstra and Negative Edge Weights?
3. Programming Task
4. In-Class-Exercise (theoretical)

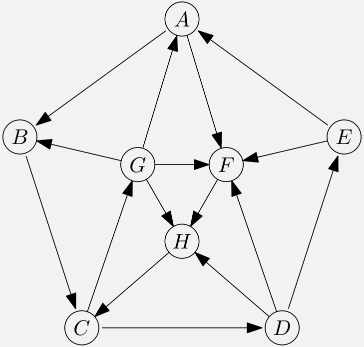
1. Feedback of last exercises

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

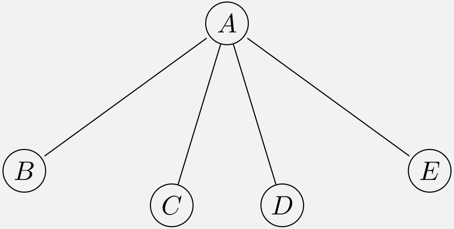
Depth-first-search and Breadth-first-search



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

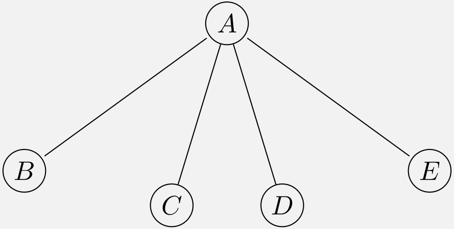
Star: DFS ordering equals BFS ordering



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

Depth-first-search and Breadth-first-search

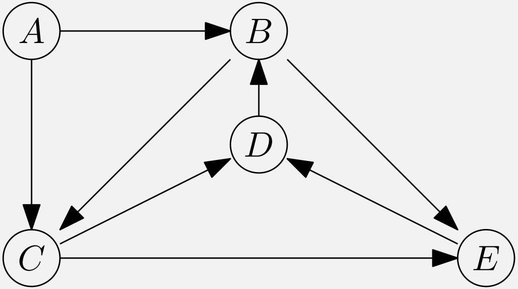
Star: DFS ordering equals BFS ordering



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

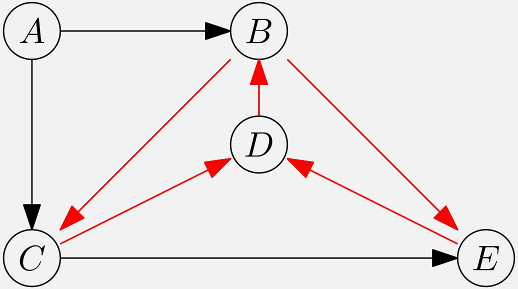
Starting at C
DFS: C, A, B, D, E
BFS: C, A, B, D, E

Topological Sorting



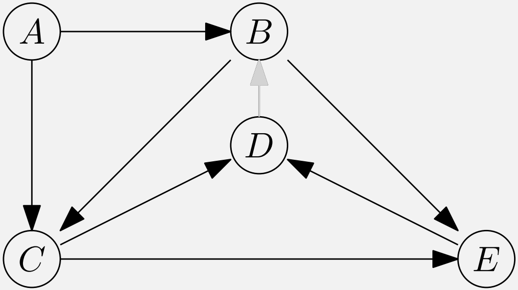
- Graph with cycles

Topological Sorting



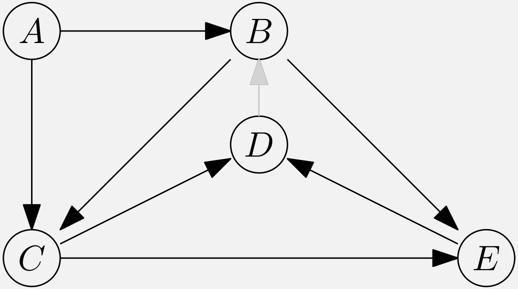
- Graph with cycles
- Two minimal cycles sharing an edge

Topological Sorting



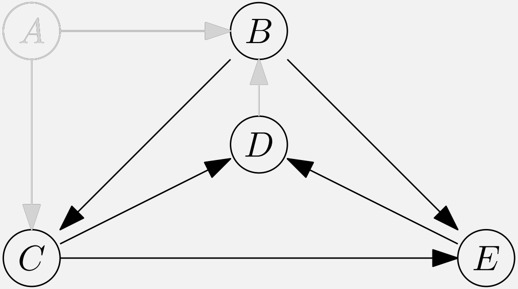
- Graph with cycles
- Two minimal cycles sharing an edge
- Remove edge $\Longrightarrow$ cycle-free

Topological Sorting



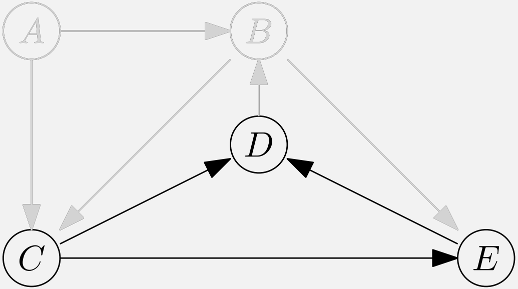
- Graph with cycles
- Two minimal cycles sharing an edge
- Remove edge $\Longrightarrow$ cycle-free
- Topological Sorting by “removing” elements with in-degree 0

Topological Sorting



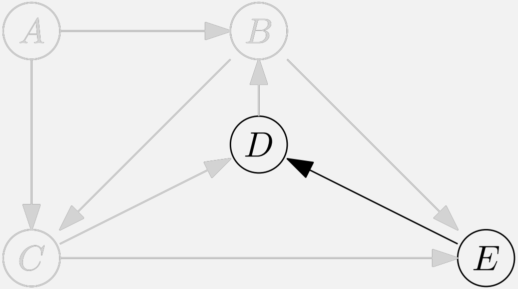
- Graph with cycles
- Two minimal cycles sharing an edge
- Remove edge $\implies$ cycle-free
- Topological Sorting by “removing” elements with in-degree 0

Topological Sorting



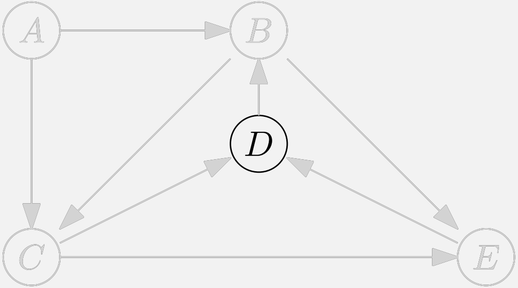
- Graph with cycles
- Two minimal cycles sharing an edge
- Remove edge $\implies$ cycle-free
- Topological Sorting by “removing” elements with in-degree 0

Topological Sorting



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

Topological Sorting



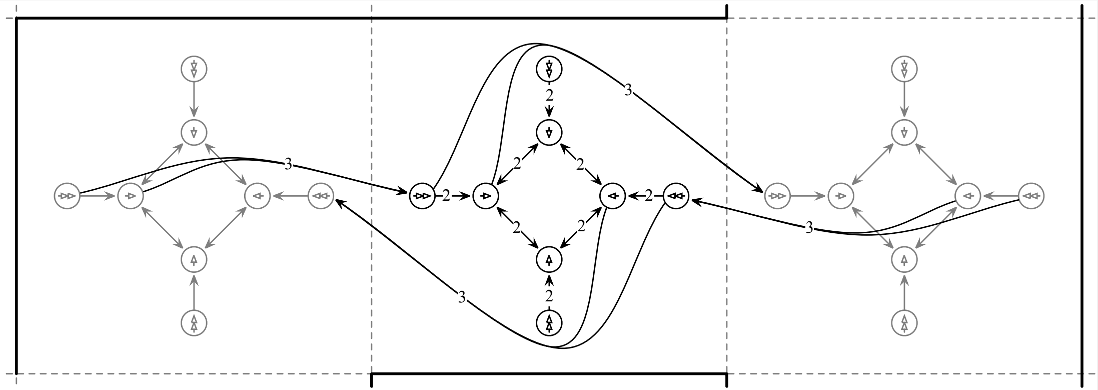
- Graph with cycles
- Two minimal cycles sharing an edge
- Remove edge $\implies$ cycle-free
- Topological Sorting by “removing” elements with in-degree 0

Exercise : Labyrinth

- Robot has to stop to change direction
- Interpret as shortest path problem

Exercise : Labyrinth

- position × direction × speed



- Runtime?

Exercise Labyrinth

- Let n be the number of squares. Graph has $|V| = 8n$ nodes
- Graph has at $|E| \leq 20n$ edges
- Therefore, Dijkstra $\mathcal{O}(|E| + |V| \log |V|)$ has runtime $\mathcal{O}(n \log n)$

2. Shortest Paths

General Algorithm

1. Initialise d_s and π_s: $d_s[v] = \infty$, $\pi_s[v] =$ null for each $v \in V$
2. Set $d_s[s] \leftarrow 0$
3. Choose an edge $(u, v) \in E$

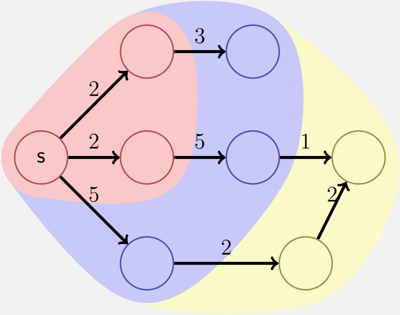
```
Relaxiere $(u, v)$:
        if $d_s[v] > d[u] + c(u, v)$ then
                $d_s[v] \leftarrow d_s[u] + c(u, v)$
                $\pi_s[v] \leftarrow u$
```

4. Repeat 3 until nothing can be relaxed any more.
   (until $d_s[v] \leq d_s[u] + c(u, v) \quad \forall (u, v) \in E$)

Dijkstra (positive egde weights)

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(G, s)

Input: Positively weighted Graph $G = (V, E, c)$, starting point $s \in V$,

Output: Minimal weights d of the shortest paths and corresponding predecessor node for each node.

```
foreach $u \in V$ do
    $d_s[u] \leftarrow \infty$; $\pi_s[u] \leftarrow null$
$d_s[s] \leftarrow 0$; $R \leftarrow \{s\}$
while $R \neq \emptyset$ do
    $u \leftarrow$ ExtractMin($R$)
    foreach $v \in N^+(u)$ do
        if $d_s[u] + c(u, v) < d_s[v]$ then
            $d_s[v] \leftarrow d_s[u] + c(u, v)$
            $\pi_s[v] \leftarrow u$
            $R \leftarrow R \cup \{v\}$
```

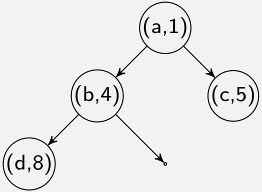
Implementation: Data Structure for R?

Relax for Dijkstra:

```
if $d_s[u] + c(u,v) < d_s[v]$ then
    $d_s[v] \leftarrow d_s[u] + c(u,v)$
    $\pi_s[v] \leftarrow u$
    if $v \notin R$ then
        Add$(R, v)$                    // Update of $(v, d(v))$ in the heap of $R$
    else
        DecreaseKey$(R, v)$            // Update of a $(v, d(v))$ in the heap of $R$
```

DecreaseKey ?

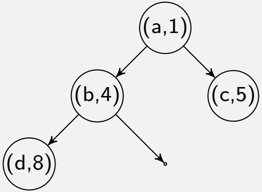
Heap ($(a,1),(b,4),(c,5),(d,8)$) $=$



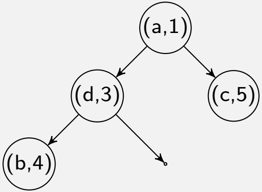
after DecreaseKey($d,3$):

DecreaseKey ?

Heap ($(a,1),(b,4),(c,5),(d,8)$) =



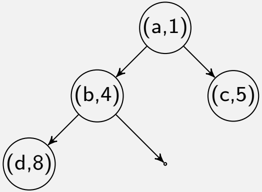
after DecreaseKey($d,3$):



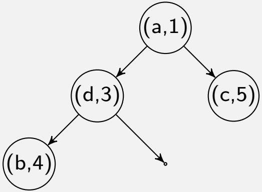
2 Probleme:

DecreaseKey ?

Heap ($(a,1),(b,4),(c,5),(d,8)$) $=$



after DecreaseKey($d,3$):

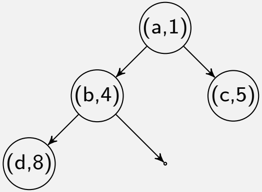


2 Probleme:

- Position of d unknown at first. Seach: $\Theta(n)$
- Positions of the nodes can change during DecreaseKey

Lazy Deletion !

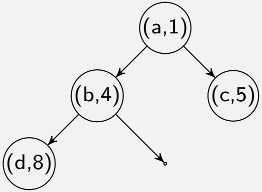
Heap ($(a,1),(b,4),(c,5),(d,8)$) =



Insert$(d,3)$:

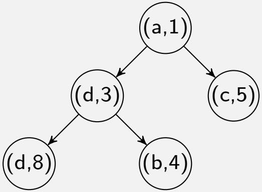
Lazy Deletion !

Heap ($(a,1),(b,4),(c,5),(d,8)$) =



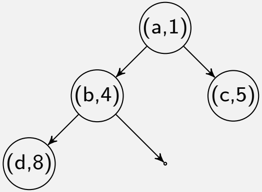
ExtractMin()

Insert($d,3$):

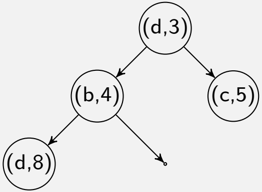


Lazy Deletion !

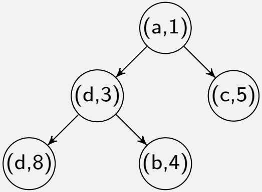
Heap ($(a,1),(b,4),(c,5),(d,8)$) =



ExtractMin() $\rightarrow (a,1)$



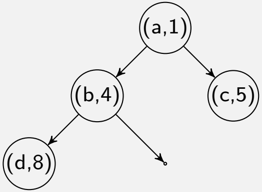
Insert($d,3$):



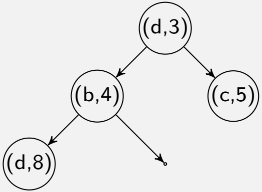
ExtractMin()

Lazy Deletion !

Heap ($(a, 1), (b, 4), (c, 5), (d, 8)$) =

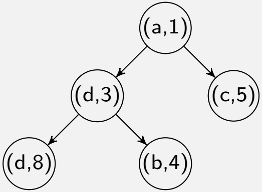


ExtractMin() → $(a, 1)$

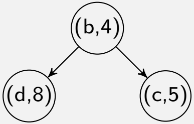


Later ExtractMin() → $(d, 8)$ must be ignored

Insert$(d, 3)$:



ExtractMin() → $(d, 3)$



Runtime Dijkstra

$n := |V|$, $m := |E|$

- $n\times$ ExtractMin: $\mathcal{O}(n \log n)$
- $m\times$ Insert or DecreaseKey: $\mathcal{O}(m \log |V|)$
- $1\times$ Init: $\mathcal{O}(n)$
- Overal: $\mathcal{O}((n+m)\log n)$. for connected graphs: $\mathcal{O}(m \log n)$

Quiz: Single Source Shortest Paths

$n := |V|, m := |E|$

problem	method	runtime	dense $m \in \mathcal{O}(n^2)$	sparse $m \in \mathcal{O}(n)$
$c \equiv 1$	BFS			

Quiz: Single Source Shortest Paths

$n := |V|, m := |E|$

problem	method	runtime	dense $m \in \mathcal{O}(n^2)$	sparse $m \in \mathcal{O}(n)$
$c \equiv 1$	BFS	$\mathcal{O}(m + n)$		

Quiz: Single Source Shortest Paths

$n := |V|, m := |E|$

problem	method	runtime	dense $m \in \mathcal{O}(n^2)$	sparse $m \in \mathcal{O}(n)$
$c \equiv 1$	BFS	$\mathcal{O}(m + n)$	$\mathcal{O}(n^2)$	

Quiz: Single Source Shortest Paths

$n := |V|, m := |E|$

problem	method	runtime	dense $m \in \mathcal{O}(n^2)$	sparse $m \in \mathcal{O}(n)$
$c \equiv 1$	BFS	$\mathcal{O}(m + n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
DAG	Top-Sort			

Quiz: Single Source Shortest Paths

$n := |V|, m := |E|$

problem	method	runtime	dense $m \in \mathcal{O}(n^2)$	sparse $m \in \mathcal{O}(n)$
$c \equiv 1$	BFS	$\mathcal{O}(m + n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
DAG	Top-Sort	$\mathcal{O}(m + n)$		

Quiz: Single Source Shortest Paths

$n := |V|, m := |E|$

problem	method	runtime	dense $m \in \mathcal{O}(n^2)$	sparse $m \in \mathcal{O}(n)$
$c \equiv 1$	BFS	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
DAG	Top-Sort	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	

Quiz: Single Source Shortest Paths

$n := |V|, m := |E|$

problem	method	runtime	dense $m \in \mathcal{O}(n^2)$	sparse $m \in \mathcal{O}(n)$
$c \equiv 1$	BFS	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
DAG	Top-Sort	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
$c \geq 0$	Dijkstra			

Quiz: Single Source Shortest Paths

$n := |V|, m := |E|$

problem	method	runtime	dense $m \in \mathcal{O}(n^2)$	sparse $m \in \mathcal{O}(n)$
$c \equiv 1$	BFS	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
DAG	Top-Sort	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
$c \geq 0$	Dijkstra	$\mathcal{O}((m+n)\log n)$		

Quiz: Single Source Shortest Paths

$n := |V|, m := |E|$

problem	method	runtime	dense $m \in \mathcal{O}(n^2)$	sparse $m \in \mathcal{O}(n)$
$c \equiv 1$	BFS	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
DAG	Top-Sort	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
$c \geq 0$	Dijkstra	$\mathcal{O}((m+n)\log n)$	$\mathcal{O}(n^2 \log n)$	

Quiz: Single Source Shortest Paths

$n := |V|, m := |E|$

problem	method	runtime	dense $m \in \mathcal{O}(n^2)$	sparse $m \in \mathcal{O}(n)$
$c \equiv 1$	BFS	$\mathcal{O}(m + n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
DAG	Top-Sort	$\mathcal{O}(m + n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
$c \geq 0$	Dijkstra	$\mathcal{O}((m + n) \log n)$	$\mathcal{O}(n^2 \log n)$	$\mathcal{O}(n \log n)$
general	Bellman-Ford	$\mathcal{O}(m \cdot n)$		

Quiz: Single Source Shortest Paths

$n := |V|, m := |E|$

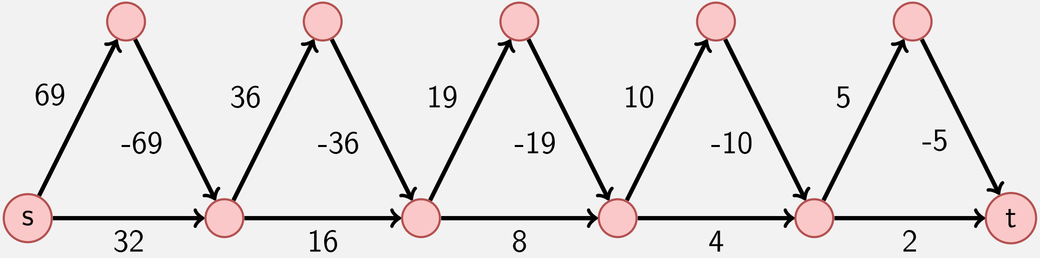
problem	method	runtime	dense $m \in \mathcal{O}(n^2)$	sparse $m \in \mathcal{O}(n)$
$c \equiv 1$	BFS	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
DAG	Top-Sort	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
$c \geq 0$	Dijkstra	$\mathcal{O}((m+n)\log n)$	$\mathcal{O}(n^2 \log n)$	$\mathcal{O}(n \log n)$
general	Bellman-Ford	$\mathcal{O}(m \cdot n)$	$\mathcal{O}(n^3)$	

Quiz: Single Source Shortest Paths

$n := |V|, m := |E|$

problem	method	runtime	dense $m \in \mathcal{O}(n^2)$	sparse $m \in \mathcal{O}(n)$
$c \equiv 1$	BFS	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
DAG	Top-Sort	$\mathcal{O}(m+n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
$c \geq 0$	Dijkstra	$\mathcal{O}((m+n)\log n)$	$\mathcal{O}(n^2 \log n)$	$\mathcal{O}(n \log n)$
general	Bellman-Ford	$\mathcal{O}(m \cdot n)$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$

An Interesting Graph



Does Dijkstra work?

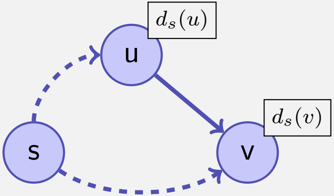
Answer

Answer

Dijkstra (as we have presented it) works also for graphs with negative edge weights, if no negative weight cycles are present. But Dijkstra may then exhibit exponential running time!

General Weighted Graphs

```
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.

$$d_s[i,v] = \min\{d_s[i-1,v], \min_{(u,v)\in E}(d_s[i-1,u] + c(u,v))$$
$$d_s[0,s] = 0, d_s[0,v] = \infty \ \forall v \neq s.$$

Algorithm Bellman-Ford(G, s)

Input: Graph $G = (V, E, c)$, starting point $s \in V$

Output: If return value true, minimal weights d for all shortest paths from s, otherwise no shortest path.

```
foreach $u \in V$ do
    $d_s[u] \leftarrow \infty$; $\pi_s[u] \leftarrow$ null
$d_s[s] \leftarrow 0$;
for $i \leftarrow 1$ to $|V|$ do
    $f \leftarrow$ false
    foreach $(u, v) \in E$ do
        $f \leftarrow f \vee \mathrm{Relax}(u, v)$
    if $f =$ false then return true
return false;
```

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

Input: Positively weighted Graph $G = (V, E, c)$, starting point $s \in V$, end point $t \in V$, estimate $\widehat{h}(v) \leq \delta(v, t)$

Output: Existence and value of a shortest path from s to 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
    $u \leftarrow$ 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 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. (`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?

Dijkstra and A*

Edge data structure

- Stores length and the destination node
- Start node is denoted by `get_adj(src)` or `std::map<NodeP,Edge> path`

Dijkstra and A*

Edge data structure

- Stores length and the destination node
- Start node is denoted by `get_adj(src)` or `std::map<NodeP,Edge> path`

Graph data structure

- `NodeP`: Pointer to a node
- `std::vector<Edge> get_adj(NodeP src)`: Returns a vector of `Edge` starting from `src`

Dijkstra and A*

Edge data structure

- Stores length and the destination node
- Start node is denoted by `get_adj(src)` or `std::map<NodeP,Edge> path`

Graph data structure

- `NodeP`: Pointer to a node
- `std::vector<Edge> get_adj(NodeP src)`: Returns a vector of `Edge` starting from `src`

`std::map<NodeP,Edge>`

- Maps a `NodeP` to an `Edge`
- `m[u]` Returns an edge

Dijkstra and A*

Node Struct

- Stores x and y coordinates

Dijkstra and A*

Node Struct

- Stores x and y coordinates

Manhattan Distance:

- $d = |\Delta x| + |\Delta y|$

Dijkstra and A*

Node Struct

- Stores x and y coordinates

Manhattan Distance:

- $d = |\Delta x| + |\Delta y|$

`std::pair`

- Access with `p.first` und `p.second`

4. 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!

Questions?