Informatik II

Übung 11

FS 2019

Program Today

1. Last Week: BFS with Lazy Deletion
2. Adjacency List in Java, continued
3. Repetition of Lecture
   - Dijkstra’s Algorithm
   - Bellman-Ford Algorithm
4. In-Class-Exercise (theoretical)
5. In-Class-Exercise (practical)

BFS with Lazy Deletion

```
public void BFS2(int s) {
        boolean visited[] = new boolean[V];
        LinkedList<Integer> queue = new LinkedList<Integer>();
        queue.add(s);
        while (!queue.isEmpty()) {
                int u = queue.poll();
                if (!visited[u]) {
                        visited[u] = true;
                        System.out.print(u + " ");
                        for (int v : adj.get(u))
                                queue.add(v);
                }
        }
}
```

BFS with Lazy Deletion

```
public void BFS2(int s) {
        boolean visited[] = new boolean[V];
        LinkedList<Integer> queue = new LinkedList<Integer>();
        queue.add(s);
        while (!queue.isEmpty()) {
                int u = queue.poll();
                if (!visited[u]) {
                        visited[u] = true;
                        System.out.print(u + " ");
                        for (int v : adj.get(u))
                                queue.add(v);
                }
        }
}
```

A node is pushed on Queue once for each in-coming edge.

BFS with Lazy Deletion

```
public void BFS2(int s) {
        boolean visited[] = new boolean[V];
        LinkedList<Integer> queue = new LinkedList<Integer>();
        queue.add(s);
        while (!queue.isEmpty()) {
                int u = queue.poll();
                if (!visited[u]) {
                        visited[u] = true;
                        System.out.print(u + " ");
                        for (int v : adj.get(u))
                                queue.add(v);
                }
        }
}
```

Node marked as visited, but its copies are not immediately removed from Queue. ("Lazy Deletion")

A node is pushed on Queue once for each in-coming edge.

Adjacency List Unweighted Graph

```
class Graph { // G = (V,E) as adjacency list
        private int V; // number of vertices
        private ArrayList<LinkedList<Integer>> adj; // adj. list
        // Constructor
        public Graph(int n) {
                V = n;
                adj = new ArrayList<LinkedList<Integer>>(V);
                for (int i=0; i<V; ++i)
                        adj.add(i,new LinkedList<Integer>());
        }
        // Edge adder method
        public void addEdge(int u, int v) {
                adj.get(u).add(v);
        }
}
```

Adjacency List weighted Graph

```
class Graph { // G = (V,E) as adjacency list
        private int V; // number of vertices
        private ArrayList<LinkedList<Pair>> adj; // adj. list
        // Constructor
        public Graph(int n) {
                V = n;
                adj = new ArrayList<LinkedList<Pair>>(V);
                for (int i=0; i<V; ++i)
                        adj.add(i,new LinkedList<Pair>());
        }
        // Edge adder method, (u,v) has weight w
        public void addEdge(int u, int v, int w) {
                adj.get(u).add(new Pair(v,w));
        }
}
```

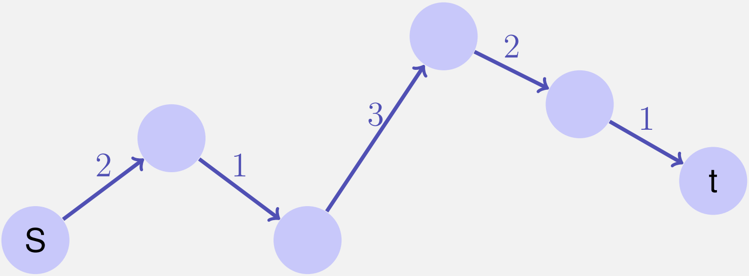
Adjacency List weighted Graph

```
public class Pair implements Comparable<Pair> {
        public int key;
        public int value;
        // Constructor
        public Pair(int key, int value) {
                this.key = key;
                this.value = value;
        }
        @Override // we need this later...
        public int compareTo(Pair other) {
                return this.value-other.value;
        }
        // for general usage of pairs we would also need
        // to provide equals(), hashCode(), ...
}
```

3. Repetition of Lecture

Weighted Graphs

Given: $G = (V, E, c)$, $c : E \to \mathbb{R}$, $s, t \in V$.
Wanted: Length (weight) of a shortest path from s to t.
Path: $p = \langle s = v_0, v_1, \ldots, v_k = t \rangle$, $(v_i, v_{i+1}) \in E$ $(0 \leq i < k)$
Weight: $c(p) := \sum_{i=0}^{k-1} c((v_i, v_{i+1}))$.



Path with weight 9

Shortest Paths

Weight of a shortest path from u to v:

$$\delta(u,v) = \begin{cases} \infty & \text{no path from } u \text{ to } v \\ \min\{c(p) : u \overset{p}{\rightsquigarrow} v\} & \text{sonst} \end{cases}$$

Ingredients of an Algorithm

Wanted: shortest paths from a starting node s.

- Weight of the shortest path found so far

  $$d_s : V \to \mathbb{R}$$

  At the beginning: $d_s[v] = \infty$ for all $v \in V$.
  Goal: $d_s[v] = \delta(s, v)$ for all $v \in V$.

- Predecessor of a node

  $$\pi_s : V \to V$$

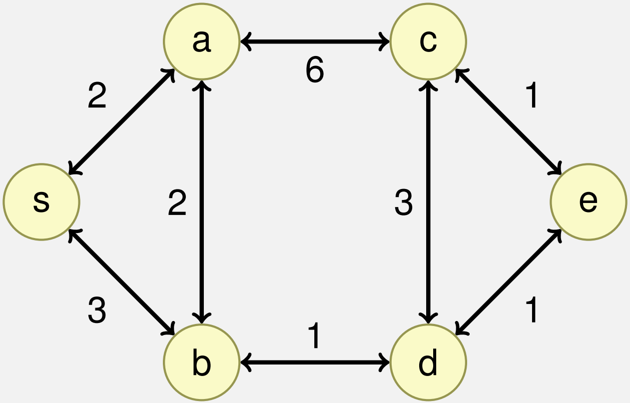
  Initially $\pi_s[v]$ undefined for each node $v \in V$

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 $d_s[v] \leq d_s[u] + c(u, v) \quad \forall (u, v) \in E$)

Assumption

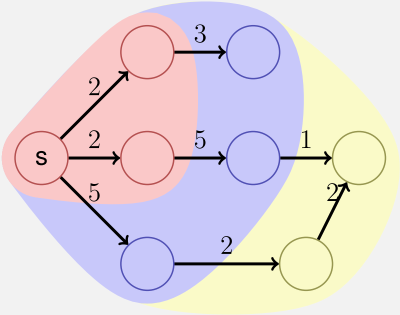


All weights of G are *positive*.

Basic Idea

Set V of nodes is partitioned into

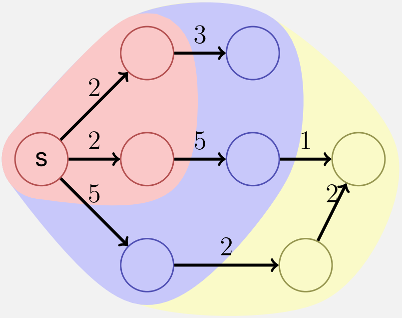
- the set M of nodes for which a shortest path from s is already known,
- the set $R = \bigcup_{v \in M} N^+(v) \setminus M$ of nodes where a shortest path is not yet known but that are accessible directly from M,
- the set $U = V \setminus (M \cup R)$ of nodes that have not yet been considered.



Induction

Induction over $|M|$: choose nodes from R with smallest upper bound. Add r to M and update R and U accordingly.

Correctness: if within the “wavefront” a node with minimal weight w has been found then no path over later nodes (providing weight $\geq d$) can provide any improvement.



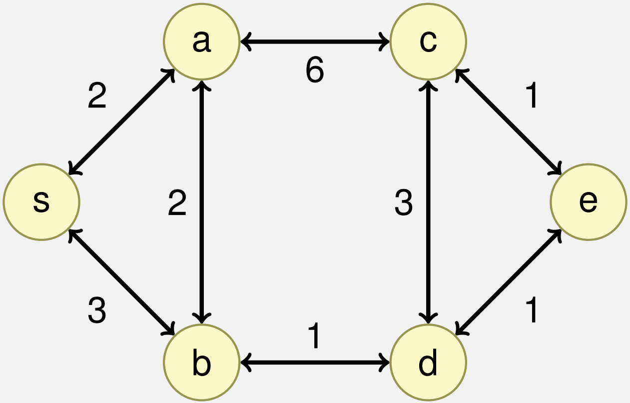
Algorithm Dijkstra(G, s)

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.

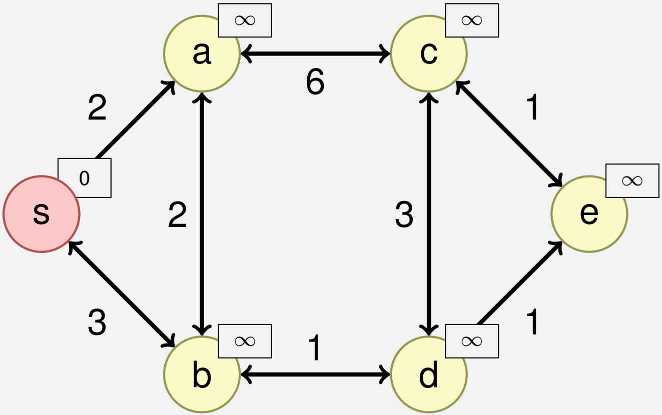
foreach $u \in V$ **do**
  $d_s[u] \leftarrow \infty$; $\pi_s[u] \leftarrow$ null

$d_s[s] \leftarrow 0$; $R \leftarrow \{s\}$
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**
      $d_s[v] \leftarrow d_s[u] + c(u, v)$
      $\pi_s[v] \leftarrow u$
      $R \leftarrow R \cup \{v\}$

Example



Example

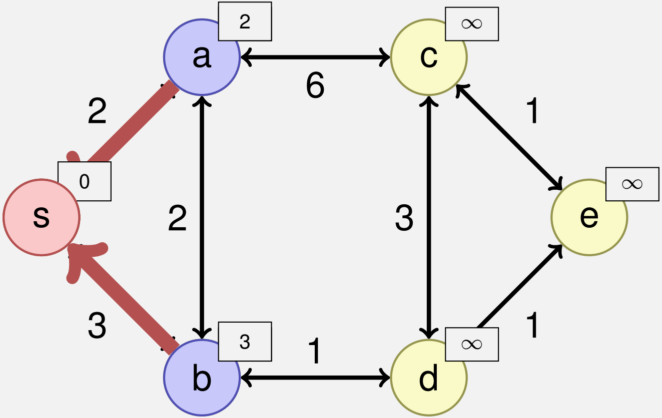


$$M = \{s\}$$

$$R = \{\}$$

$$U = \{a, b, c, d, e\}$$

Example

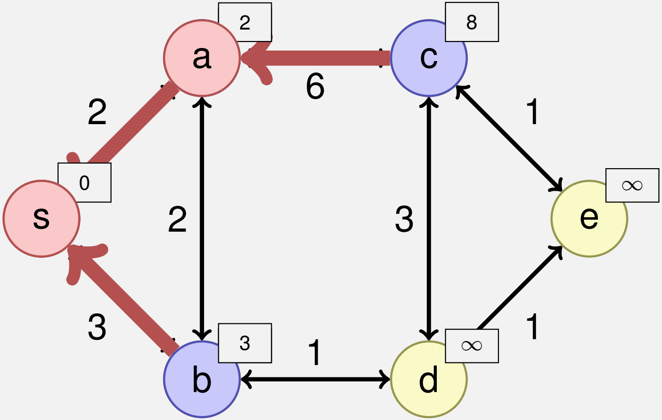


$$M = \{s\}$$

$$R = \{a, b\}$$

$$U = \{c, d, e\}$$

Example

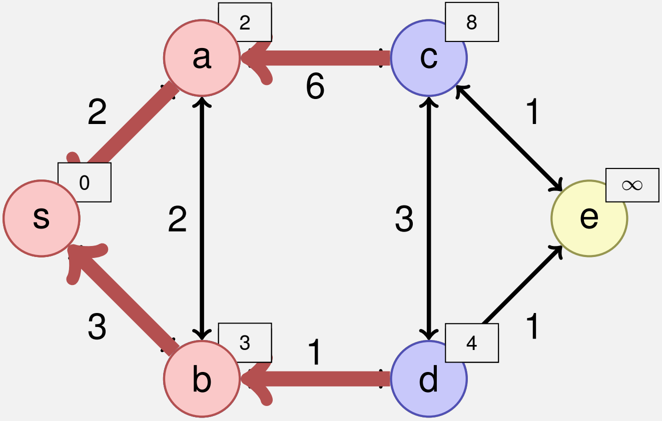


$$M = \{s, a\}$$

$$R = \{b, c\}$$

$$U = \{d, e\}$$

Example

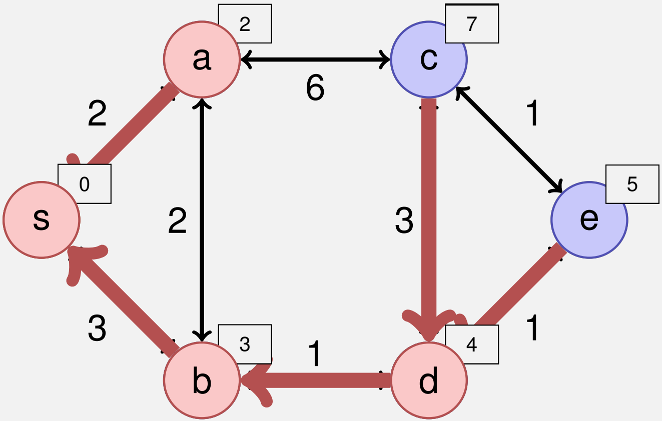


$$M = \{s, a, b\}$$

$$R = \{c, d\}$$

$$U = \{e\}$$

Example

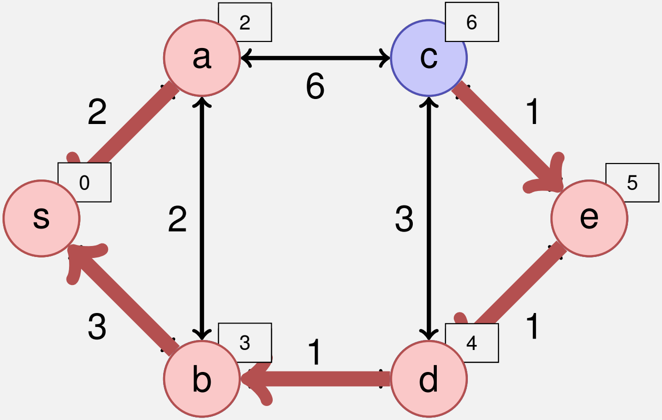


$$M = \{s, a, b, d\}$$

$$R = \{c, e\}$$

$$U = \{\}$$

Example

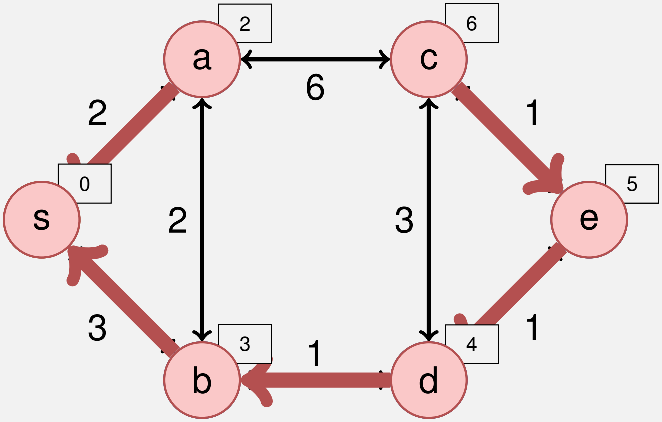


$$M = \{s, a, b, d, e\}$$

$$R = \{c\}$$

$$U = \{\}$$

Example



$$M = \{s, a, b, d, e, c\}$$

$$R = \{\}$$

$$U = \{\}$$

Implementation: Data Structure for R?

Required operations:

- Insert (add to R)
- ExtractMin (over R) and DecreaseKey (Update in R)

foreach $v \in N^+(u)$ **do**
    if $d_s[u] + c(u,v) < d_s[v]$ **then**
        $d_s[v] \leftarrow d_s[u] + c(u,v)$
        $\pi_s[v] \leftarrow u$
        if $v \in R$ **then**
            DecreaseKey(R, v) // Update of a $d(v)$ in the heap of R
        else
            $R \leftarrow R \cup \{v\}$ // Update of $d(v)$ in the heap of R

MinHeap!

DecreaseKey

- DecreaseKey: climbing in MinHeap in $\mathcal{O}(\log |V|)$
- Position in the heap (i.e. array index of element in the heap)?

DecreaseKey

- DecreaseKey: climbing in MinHeap in $\mathcal{O}(\log |V|)$
- Position in the heap (i.e. array index of element in the heap)?
  - alternative (a): Store position at the nodes

DecreaseKey

- DecreaseKey: climbing in MinHeap in $\mathcal{O}(\log |V|)$
- Position in the heap (i.e. array index of element in the heap)?
  - alternative (a): Store position at the nodes
  - alternative (b): Hashtable of the nodes

DecreaseKey

- DecreaseKey: climbing in MinHeap in $\mathcal{O}(\log |V|)$
- Position in the heap (i.e. array index of element in the heap)?
  - alternative (a): Store position at the nodes
  - alternative (b): Hashtable of the nodes
  - alternative (c): re-insert node each time after update-operation and mark it as visited ("deleted") once extracted (Lazy Deletion)

Runtime

- $|V| \times$ ExtractMin: $\mathcal{O}(|V| \log |V|)$
- $|E| \times$ Insert or DecreaseKey: $\mathcal{O}(|E| \log |V|)$
- $1 \times$ Init: $\mathcal{O}(|V|)$
- Overal: $\mathcal{O}(|E| \log |V|)$.

General Weighted Graphs

Relaxing Step as with Dijkstra:

Relax(u, v)

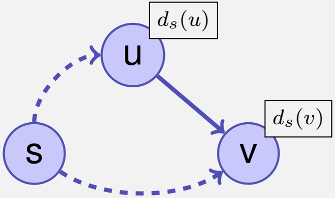
if $d_s(v) > d_s(u) + c(u, v)$ **then**

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

$\quad \pi_s(v) \leftarrow u$

$\quad$ **return** true

return false



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

Observations

- *Observation 1:* Sub-paths of shortest paths are shortest paths. Let $p = \langle v_0, \ldots, v_k \rangle$ be a shortest path from v_0 to v_k. Then each of the sub-paths $p_{ij} = \langle v_i, \ldots, v_j \rangle$ $(0 \leq i < j \leq k)$ is a shortest path from v_i to v_j.
  Proof: if not, then one of the sub-paths could be shortened which immediately leads to a contradiction.
- *Observation:* If there is a shortest path then it is simple, thus does not provide a node more than once.
  Immediate Consequence of observation 1.

Dynamic Programming Approach (Bellman)

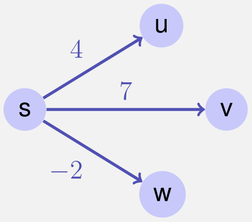
Induction over number of edges $d_s[i,v]$: Shortest path from s to v via maximally i edges.

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

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

Dynamic Programming Approach (Bellman)

	s	$\cdots$	v	$\cdots$	w
0	0	∞	∞	∞	∞
1	0	∞	7	∞	-2
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$n-1$	0	$\cdots$	$\cdots$	$\cdots$	$\cdots$



Algorithm: Iterate over last row until the relaxation steps do not provide any further changes, maximally $n-1$ iterations. If still changes, then there is no shortest path.

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.

$d(v) \leftarrow \infty \; \forall v \in V$; $d(s) \leftarrow 0$
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;   // Negative Cycle!

Runtime $\mathcal{O}(|E| \cdot |V|)$.

Conclusion

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

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	$\mathcal{O}(m \cdot n)$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$

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: $\mathcal{O}(|V| + |E|)$.

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

In-Class-Exercises: Longest Path in DAGs

Solution:

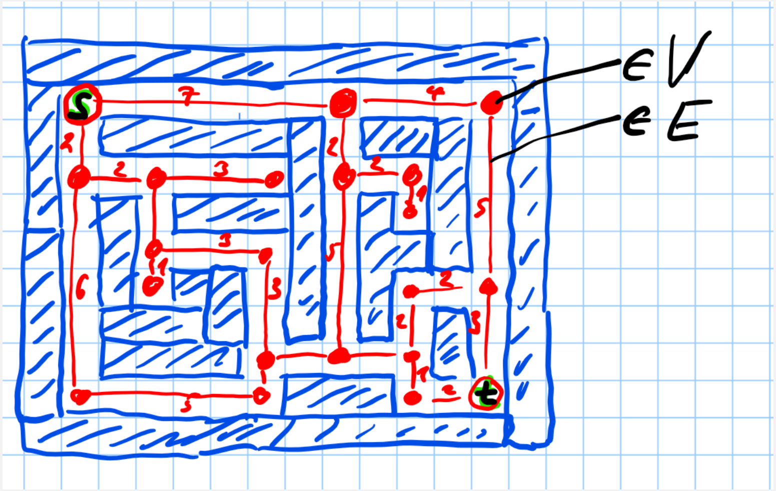
1. Topological Sorting. Running time: $\mathcal{O}(|V| + |E|)$.
2. Compute for each node all incoming edges: $\mathcal{O}(|V| + |E|)$.
3. Visit each node v in topological order and consider all incoming edges: $\mathcal{O}(|V| + |E|)$.

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

   Store predecessor!

5. In-Class-Exercise (practical)

Shortest Path in a Maze



Questions / Suggestions?