Informatik II

Übung 9

Andreas Bärtschi, Andreea Ciuprina, Felix Friedrich, Patrick Gruntz,
Hermann Lehner, Max Rossmannek, Chris Wendler

FS 2018

Program Today

1. Last Week: BFS with Lazy Deletion
2. Adjacency List in Java, continued
3. Repetition of Lecture: Dijkstra's Algorithm
4. In-Class-Exercise

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 to Queue once for each incoming 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 to Queue once for each incoming 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(), ...
}
```

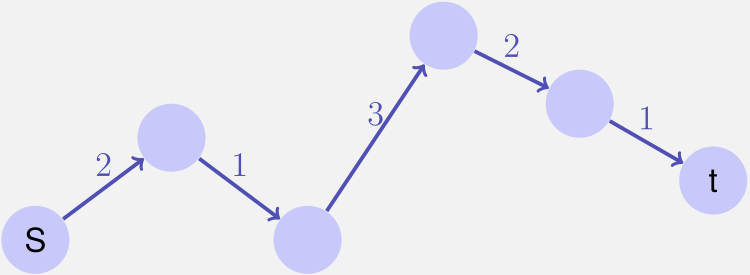
Weighted Graphs

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.

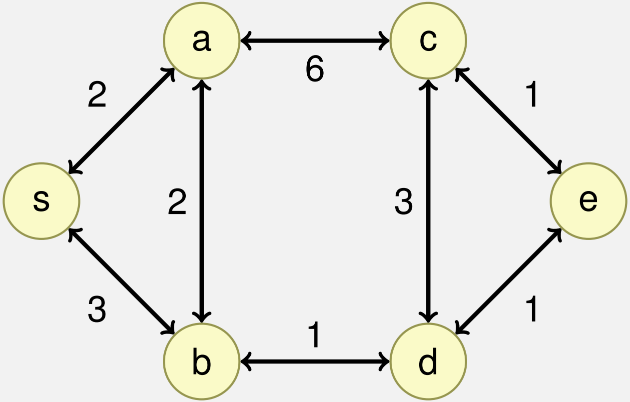
Path: $p = \langle s = v_0, v_1, \ldots, v_k = t \rangle$, $(v_i, v_{i+1}) \in E$ $(0 \leq i < k)$

Weight: $c(p) := \sum_{i=0}^{k-1} c((v_i, v_{i+1}))$.



Path with weight 9

Assumption

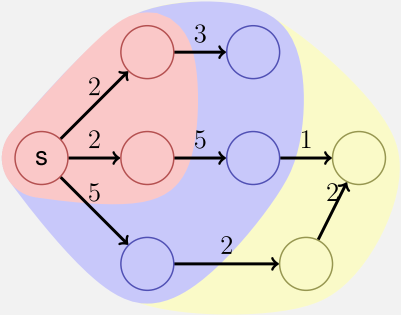


All weights of G are *positive*.

Basic Idea

Set V of nodes is partitioned into

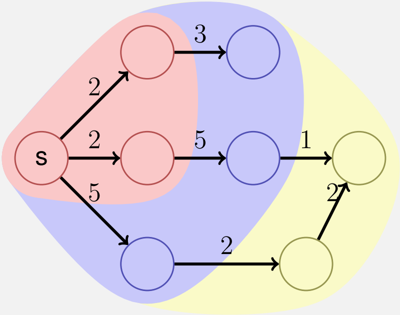
- the set M of nodes for which a shortest path from s is already known,
- the set $R = \bigcup_{v \in M} N^+(v) \setminus M$ of nodes where a shortest path is not yet known but that are accessible directly from M,
- the set $U = V \setminus (M \cup R)$ of nodes that have not yet been considered.



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 has been found then no path with greater weight over different nodes can provide any improvement.



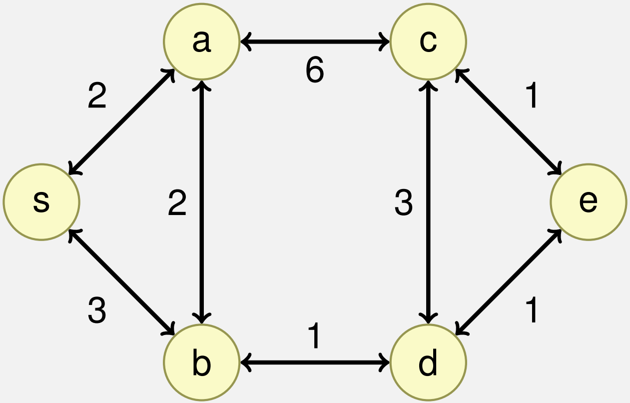
Algorithmus Dijkstra

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

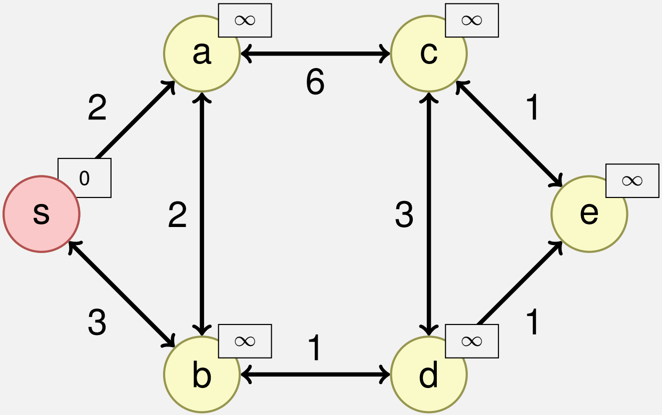
- Set $\mathrm{PL}(s) \leftarrow 0$
- Start with $M = \{s\}$. Set $k \leftarrow s$.
- While a new node k is added and this is not the target node

  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.

Example



Example

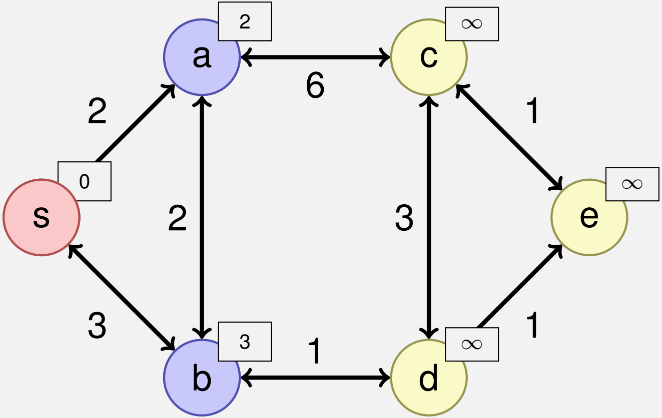


$$M = \{s\}$$

$$R = \{\}$$

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

Example

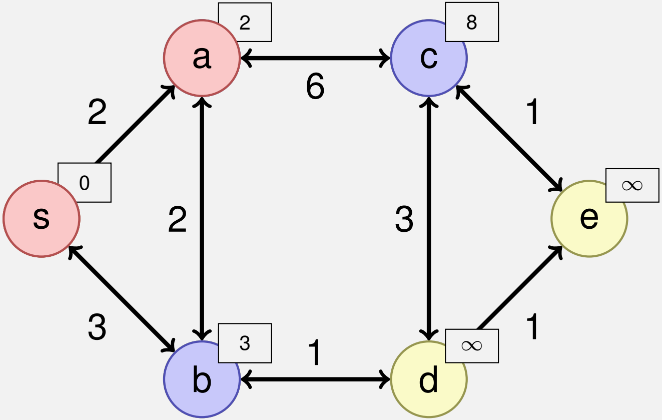


$$M = \{s\}$$

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

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

Example

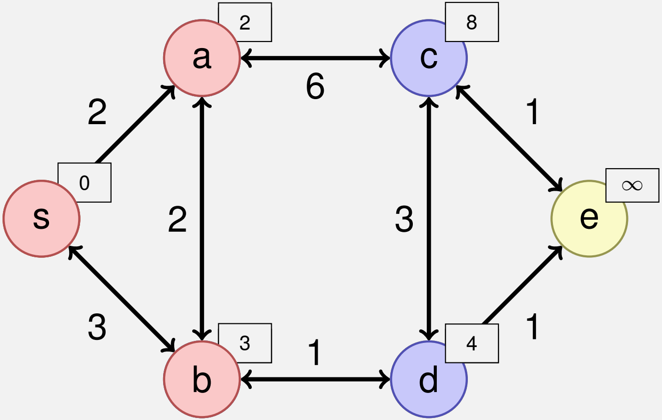


$$M = \{s, a\}$$

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

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

Example

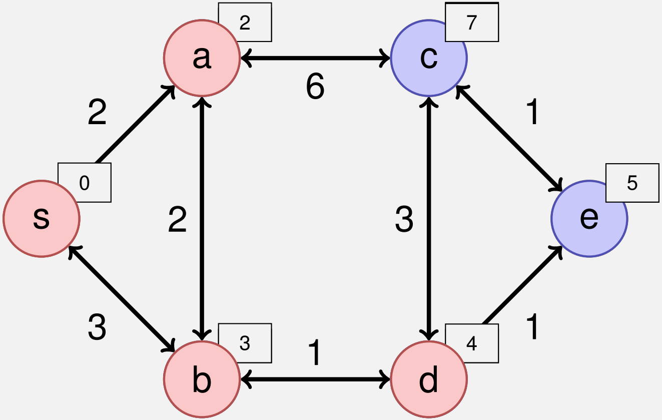


$$M = \{s, a, b\}$$

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

$$U = \{e\}$$

Example

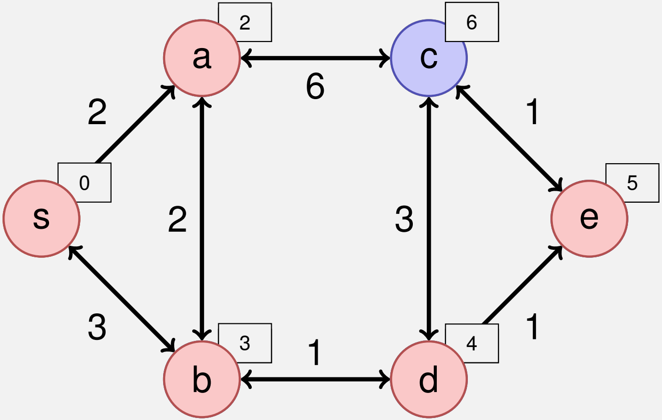


$$M = \{s, a, b, d\}$$

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

$$U = \{\}$$

Example

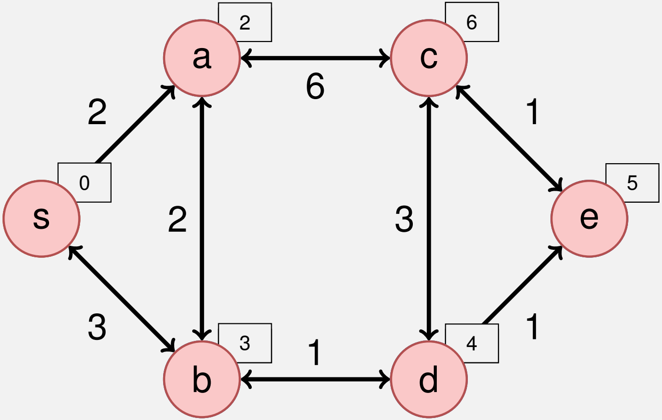


$$M = \{s, a, b, d, e\}$$

$$R = \{c\}$$

$$U = \{\}$$

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 $v \in N^+(m)$ **do**
  if $d(m) + c(m,v) < d(v)$ **then**
    $d(v) \leftarrow d(m) + c(m,v)$
    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

Implementation: Data Structure for R?

Required operations:

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

foreach $v \in N^+(m)$ **do**
  if $d(m) + c(m, v) < d(v)$ **then**
    $d(v) \leftarrow d(m) + c(m, v)$
    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

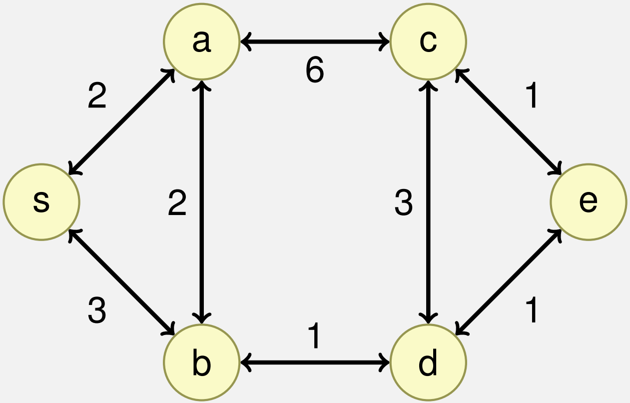
- $|V| \times$ ExtractMin: $\mathcal{O}(|V| \log |V|)$
- $|E| \times$ Insert or DecreaseKey: $\mathcal{O}(|E| \log |V|)$
- $1 \times$ Init: $\mathcal{O}(|V|)$
- Overal: $\mathcal{O}(|E| \log |V|)$.

Can be improved when a data structure optimized for ExtractMin and DecreaseKey ist used (Fibonacci Heap), then runtime $\mathcal{O}(|E| + |V| \log |V|)$.

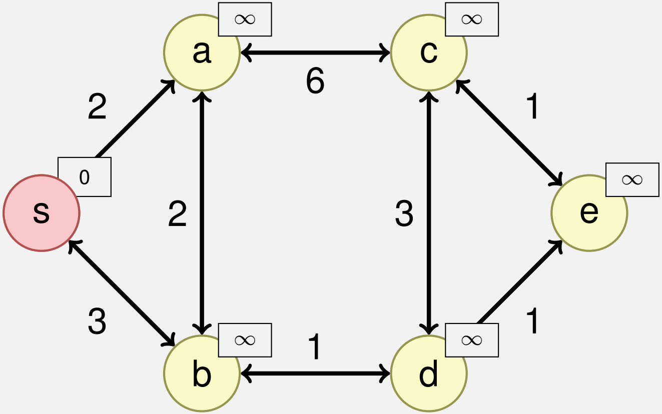
Reconstruct shortest Path

- Memorize best predecessor during the update step in the algorithm above. Store it with the node or in a separate data structure.
- Reconstruct best path by traversing backwards via best predecessor

Example



Example

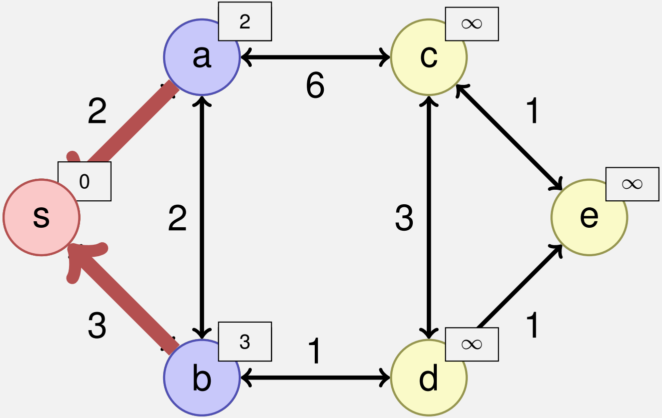


$$M = \{s\}$$

$$R = \{\}$$

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

Example

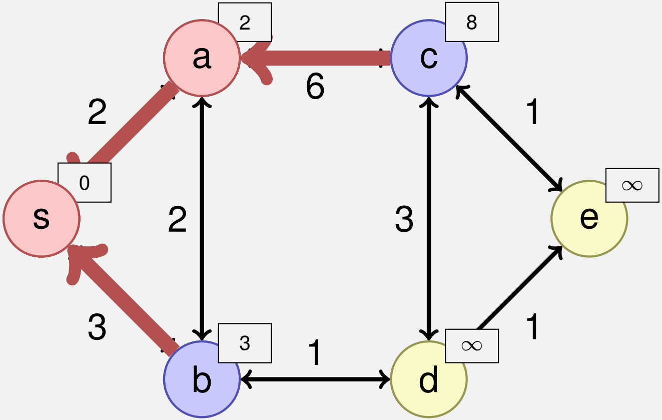


$$M = \{s\}$$

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

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

Example

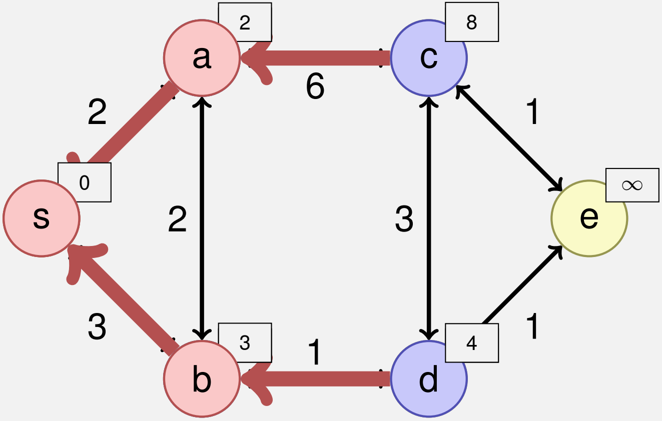


$$M = \{s, a\}$$

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

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

Example

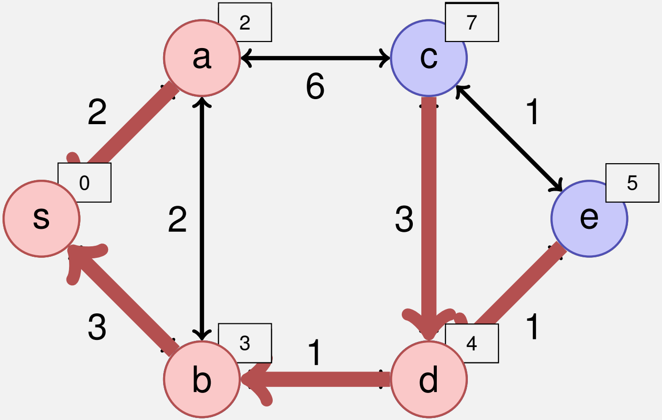


$$M = \{s, a, b\}$$

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

$$U = \{e\}$$

Example

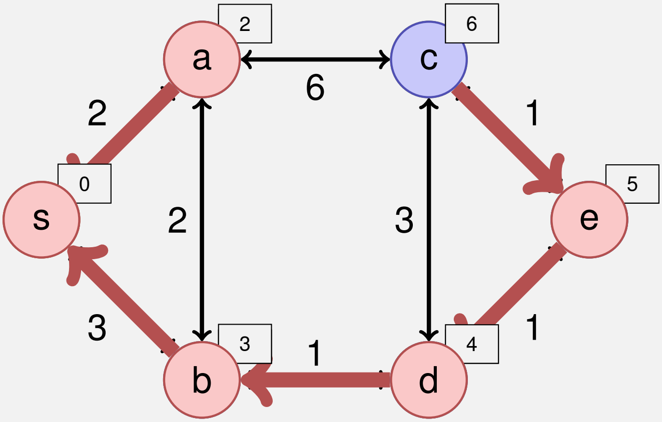


$$M = \{s, a, b, d\}$$

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

$$U = \{\}$$

Example

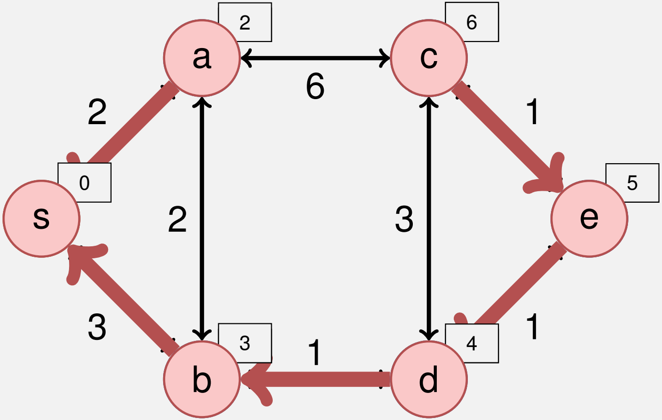


$$M = \{s, a, b, d, e\}$$

$$R = \{c\}$$

$$U = \{\}$$

Example



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

$$R = \{\}$$

$$U = \{\}$$

In-Class-Exercises: Longest Path in DAGs

Finding a shortest path is easy (BFS, Dijkstra). 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). 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). 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: $\mathcal{O}(|V| + |E|)$.

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

Store predecessor!

Questions / Suggestions?