Informatik II

Übung 9

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

FS 2018

Heutiges Programm

1. Letzte Woche: BFS mit Lazy Deletion
2. Adjazenzlisten in Java, Fortsetzung
3. Wiederholung Vorlesung: Algorithmus von Dijkstra
4. In-Class-Exercise

BFS mit 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 mit 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);
                }
        }
}
```

Knoten kommt für jede Eingangskante genau einmal auf die Queue.

BFS mit 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);
                }
        }
}
```

Knoten wird als besucht markiert, weitere Einträge in Queue nicht sofort gelöscht. (“Lazy Deletion”)

Knoten kommt für jede Eingangskante genau einmal auf die Queue.

Adjazenzliste ungewichteter 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);
        }
}
```

Adjazenzliste gewichteter Graph

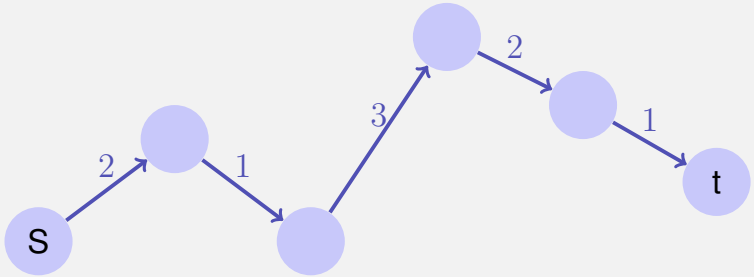
```
class Graph { // G = (V,E) as adjacency list
        private int V; // number of vertices
        private ArrayList<LinkedList<Pair>> adj; // adj. list
        // Constructor
        public Graph(int n) {
                V = n;
                adj = new ArrayList<LinkedList<Pair>>(V);
                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));
        }
}
```

Adjazenzliste gewichteter 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(), ...
}
```

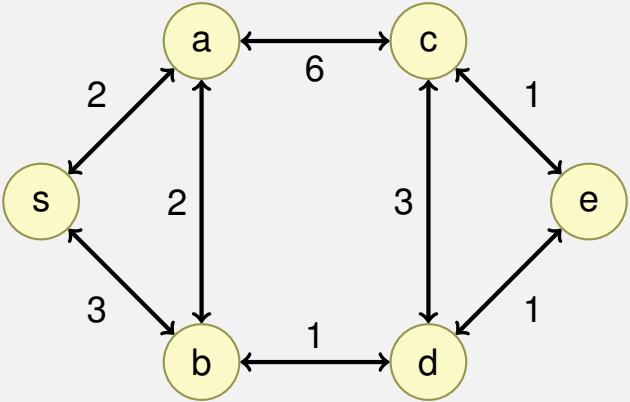
Gewichtete Graphen

Gegeben: $G = (V, E, c)$, $c : E \to \mathbb{R}$, $s, t \in V$.
Gesucht: Länge (Gewicht) eines kürzesten Weges von s nach t.
Weg: $p = \langle s = v_0, v_1, \ldots, v_k = t \rangle$, $(v_i, v_{i+1}) \in E$ $(0 \leq i < k)$
Gewicht: $c(p) := \sum_{i=0}^{k-1} c((v_i, v_{i+1}))$.



Weg mit Gewicht 9

Annahme

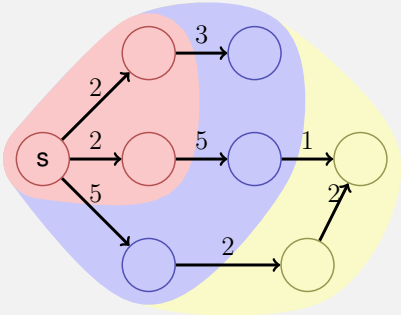


Alle Gewichte von G sind *positiv*.

Grundidee

Menge V aller Knoten wird unterteilt in

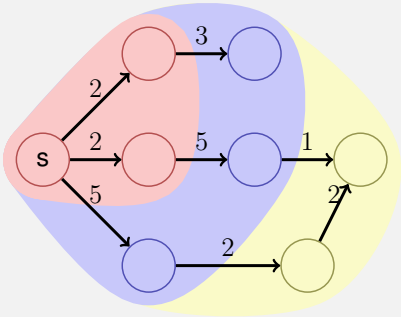
- die Menge M von Knoten, für die schon ein kürzester Weg von s bekannt ist
- die Menge $R = \bigcup_{v \in M} N^+(v) \setminus M$ von Knoten, für die kein kürzester Weg bekannt ist, die jedoch von M direkt erreichbar sind.
- die Menge $U = V \setminus (M \cup R)$ von Knoten, die noch nicht berücksichtigt wurden.



Induktion

Induktion über $|M|$: Wähle Knoten aus R mit kleinster oberer Schranke. Nimm r zu M hinzu, und update R und U.

Korrektheit: Ist innerhalb einer “Wellenfront” einmal ein Knoten mit minimalem Pfadgewicht gefunden, kann kein Pfad grösseren Gewichts über andere Knoten zu einer Verbesserung führen.

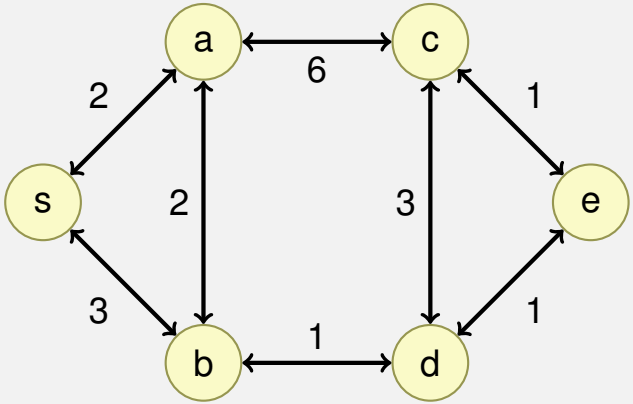


Algorithmus Dijkstra

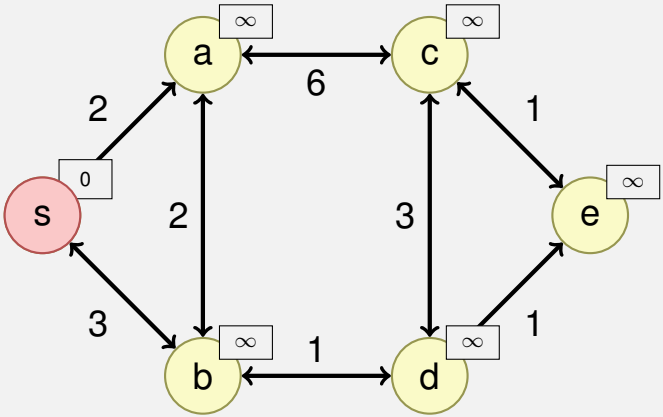
Initial: $\mathrm{PL}(n) \leftarrow \infty$ für alle Knoten.

- Setze $\mathrm{PL}(s) \leftarrow 0$
- Starte mit $M = \{s\}$. Setze $k \leftarrow s$.
- Solange ein neuer Knoten k hinzukommt und dieser nicht der Zielknoten ist
  1. Für jeden Nachbarknoten n von k:
     - Berechne Pfadlänge x nach n über k
     - Wenn $\mathrm{PL}(n) = \infty$, so nimm n zu R hinzu
     - Ist $x < \mathrm{PL}(n) < \infty$, so setze $\mathrm{PL}(n) \leftarrow x$ und passe R an.
  2. Wähle als neuen Knoten k den mit kleinster Pfadlänge in R.

Beispiel



Beispiel

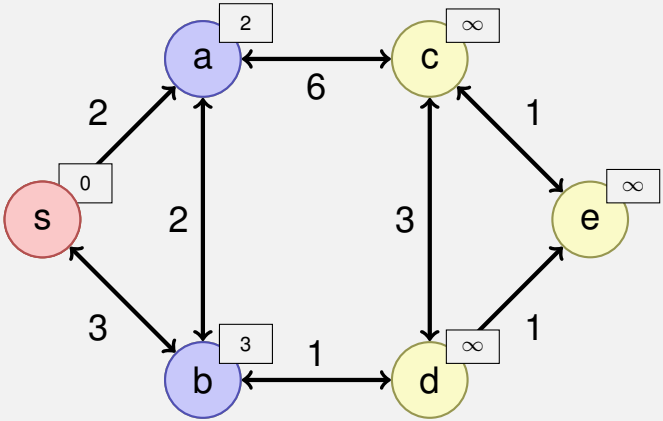


$$M = \{s\}$$

$$R = \{\}$$

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

Beispiel

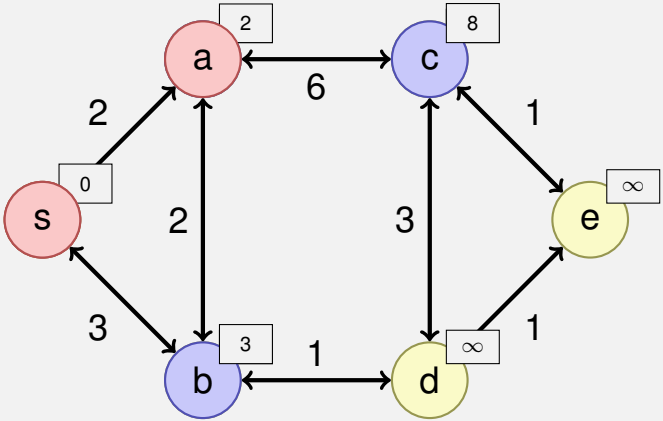


$$M = \{s\}$$

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

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

Beispiel

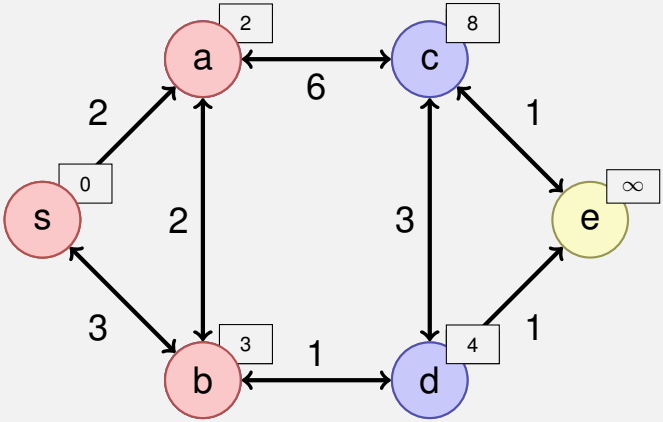


$$M = \{s, a\}$$

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

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

Beispiel

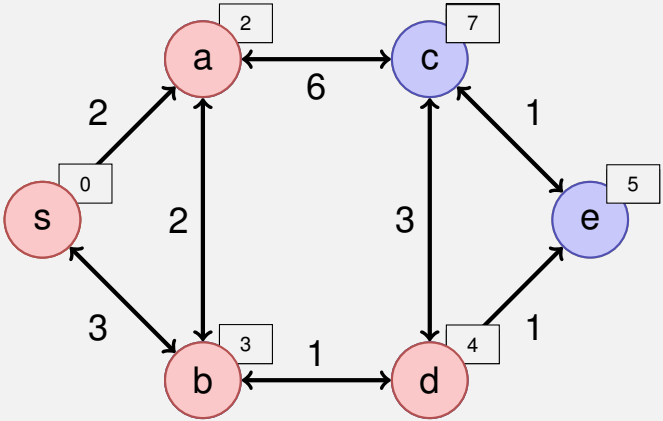


$$M = \{s, a, b\}$$

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

$$U = \{e\}$$

Beispiel

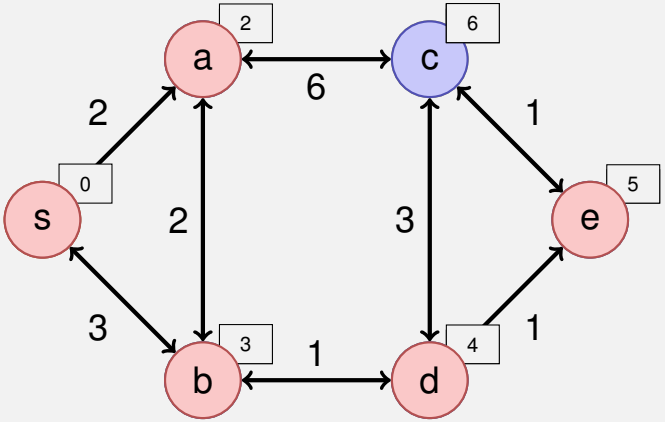


$$M = \{s, a, b, d\}$$

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

$$U = \{\}$$

Beispiel

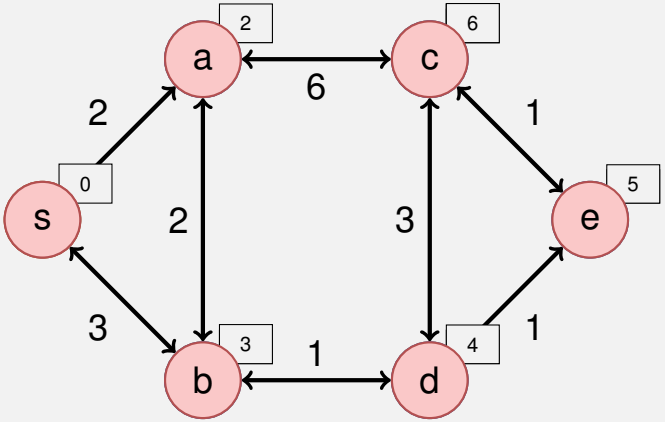


$$M = \{s, a, b, d, e\}$$

$$R = \{c\}$$

$$U = \{\}$$

Beispiel



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

$$R = \{\}$$

$$U = \{\}$$

Zur Implementation: Datenstruktur für R?

Benötigte Operationen:

- Insert (Hinzunehmen zu R)
- ExtractMin (über R) und 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 eines $d(v)$ im Heap zu R
    else
      $R \leftarrow R \cup \{v\}$ // Einfügen eines neuen $d(v)$ im Heap zu R

Zur Implementation: Datenstruktur für R?

Benötigte Operationen:

- Insert (Hinzunehmen zu R)
- ExtractMin (über R) und DecreaseKey (Update in R)

```
foreach v ∈ N⁺(m) do
    if d(m) + c(m, v) < d(v) then
        d(v) ← d(m) + c(m, v)
        if v ∈ R then
            DecreaseKey(R, v)          // Update eines d(v) im Heap zu R
        else
            R ← R ∪ {v}                // Einfügen eines neuen d(v) im Heap zu R
```

MinHeap!

DecreaseKey

- DecreaseKey: Aufsteigen im MinHeap in $\mathcal{O}(\log |V|)$
- Position im Heap (das heisst Arrayindex des Elements im Heap)?

DecreaseKey

- DecreaseKey: Aufsteigen im MinHeap in $\mathcal{O}(\log |V|)$
- Position im Heap (das heisst Arrayindex des Elements im Heap)?
  - Möglichkeit (a): Speichern am Knoten

DecreaseKey

- DecreaseKey: Aufsteigen im MinHeap in $\mathcal{O}(\log |V|)$
- Position im Heap (das heisst Arrayindex des Elements im Heap)?
  - Möglichkeit (a): Speichern am Knoten
  - Möglichkeit (b): Hashtabelle über Knoten

DecreaseKey

- DecreaseKey: Aufsteigen im MinHeap in $\mathcal{O}(\log |V|)$
- Position im Heap (das heisst Arrayindex des Elements im Heap)?
  - Möglichkeit (a): Speichern am Knoten
  - Möglichkeit (b): Hashtabelle über Knoten
  - Möglichkeit (c): Knoten nach Update-Operation immer wieder erneut einfügen. Knoten beim Entnehmen als visited ("deleted") kennzeichnen (Lazy Deletion)

Laufzeit

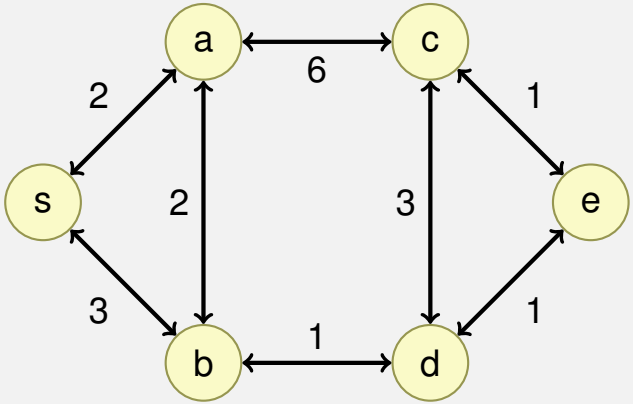
- $|V|\times$ ExtractMin: $\mathcal{O}(|V| \log |V|)$
- $|E|\times$ Insert oder DecreaseKey: $\mathcal{O}(|E| \log |V|)$
- $1\times$ Init: $\mathcal{O}(|V|)$
- Insgesamt: $\mathcal{O}(|E| \log |V|)$.

Kann verbessert werden unter Verwendung einer für ExtractMin und DecreaseKey optimierten Datenstruktur (Fibonacci Heap), dann Laufzeit $\mathcal{O}(|E| + |V| \log |V|)$.

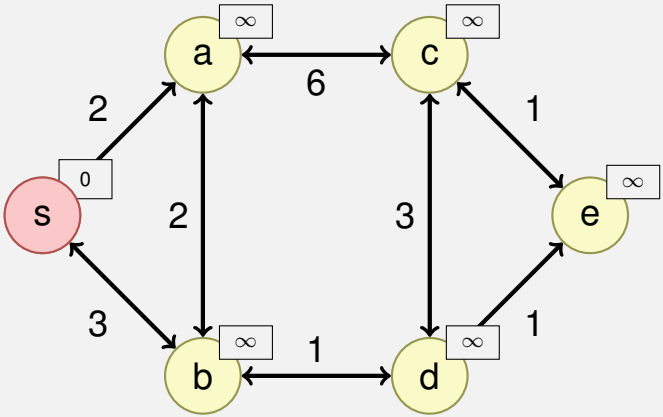
Kürzesten Weg Rekonstruieren

- Beim Updateschritt im obigen Algorithmus jeweils besten Vorgänger merken, an Knoten oder in separater Datenstruktur.
- Besten Pfad rekonstruieren durch Rückwärtslaufen der besten Kanten

Beispiel



Beispiel

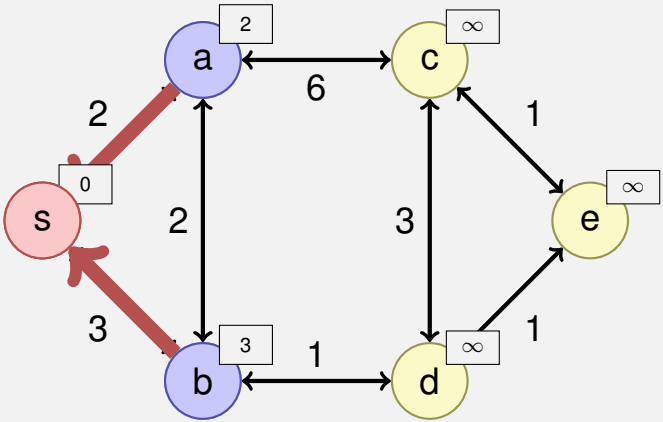


$$M = \{s\}$$

$$R = \{\}$$

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

Beispiel

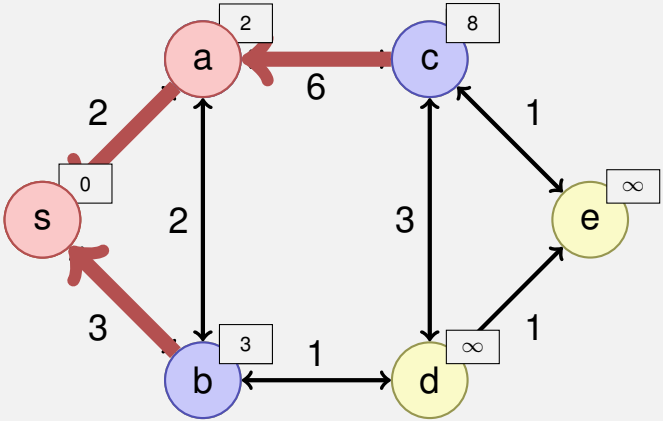


$$M = \{s\}$$

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

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

Beispiel

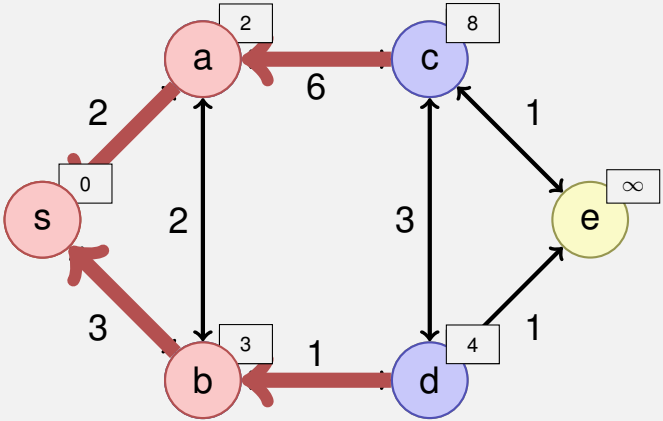


$$M = \{s, a\}$$

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

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

Beispiel

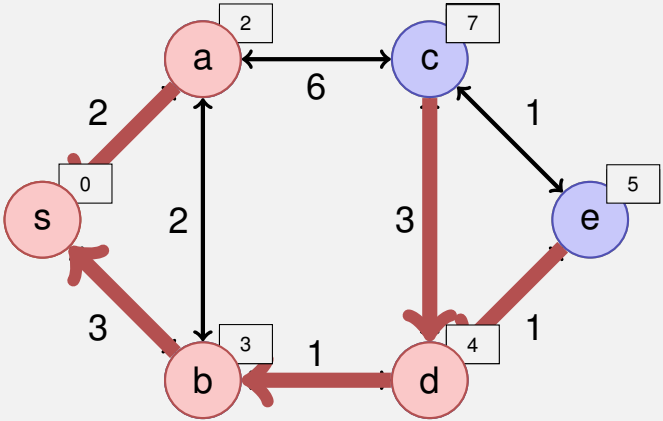


$$M = \{s, a, b\}$$

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

$$U = \{e\}$$

Beispiel

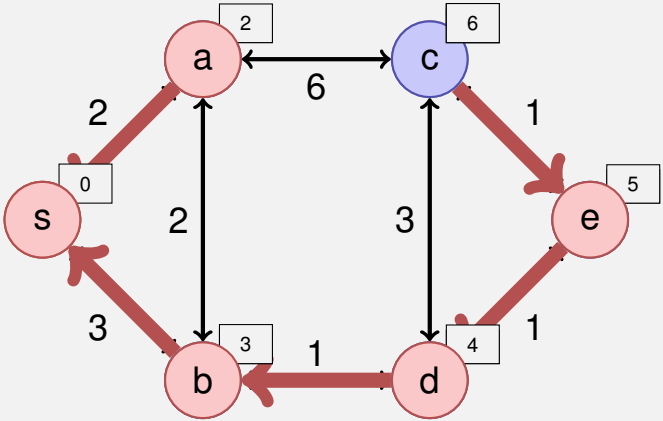


$$M = \{s, a, b, d\}$$

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

$$U = \{\}$$

Beispiel

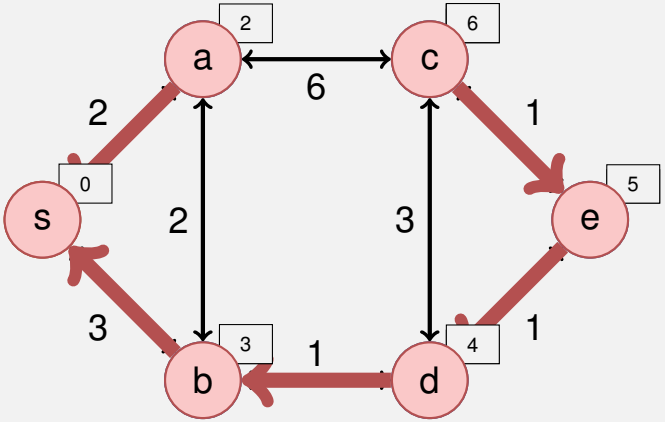


$$M = \{s, a, b, d, e\}$$

$$R = \{c\}$$

$$U = \{\}$$

Beispiel



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

$$R = \{\}$$

$$U = \{\}$$

In-Class-Exercises: Längster Pfad in DAGs

Das Kürzeste-Pfad-Problem hat einfache Lösungen (BFS, Dijkstra). Das Längste-Pfad-Problem hingegen ist sehr schwierig! Für gerichtete Graphen gibt es vermutlich keinen schnellen Algorithmus, um Pfade der Länge $\gg \log^2 n$ zu finden.

In-Class-Exercises: Längster Pfad in DAGs

Das Kürzeste-Pfad-Problem hat einfache Lösungen (BFS, Dijkstra). Das Längste-Pfad-Problem hingegen ist sehr schwierig! Für gerichtete Graphen gibt es vermutlich keinen schnellen Algorithmus, um Pfade der Länge $\gg \log^2 n$ zu finden.

Aufgabe:

Gegeben sei ein gerichteter, **kreisfreier** Graph (DAG) $G = (V, E)$.

Entwerfen Sie einen $\mathcal{O}(|V| + |E|)$-Laufzeit Algorithmus, um den *längsten Pfad* zu finden.

In-Class-Exercises: Längster Pfad in DAGs

Das Kürzeste-Pfad-Problem hat einfache Lösungen (BFS, Dijkstra). Das Längste-Pfad-Problem hingegen ist sehr schwierig! Für gerichtete Graphen gibt es vermutlich keinen schnellen Algorithmus, um Pfade der Länge $\gg \log^2 n$ zu finden.

Aufgabe:

Gegeben sei ein gerichteter, **kreisfreier** Graph (DAG) $G = (V, E)$.

Entwerfen Sie einen $\mathcal{O}(|V| + |E|)$-Laufzeit Algorithmus, um den *längsten Pfad* zu finden.

Tipp: G ist kreisfrei, Sie können also zuerst topologisch sortieren.

In-Class-Exercises: Längster Pfad in DAGs

Lösung:

1. Topologisch Sortieren. Laufzeit: $\mathcal{O}(|V| + |E|)$.

In-Class-Exercises: Längster Pfad in DAGs

Lösung:

1. Topologisch Sortieren. Laufzeit: $\mathcal{O}(|V| + |E|)$.
2. Berechne für jeden Knoten alle eingehenden Kanten: $\mathcal{O}(|V| + |E|)$.

In-Class-Exercises: Längster Pfad in DAGs

Lösung:

1. Topologisch Sortieren. Laufzeit: $\mathcal{O}(|V| + |E|)$.
2. Berechne für jeden Knoten alle eingehenden Kanten: $\mathcal{O}(|V| + |E|)$.
3. Besuche jeden Knoten v in Reihenfolge der topologischen Sortierung und betrachte die Eingangs-Kanten: $\mathcal{O}(|V| + |E|)$.

In-Class-Exercises: Längster Pfad in DAGs

Lösung:

1. Topologisch Sortieren. Laufzeit: $\mathcal{O}(|V| + |E|)$.
2. Berechne für jeden Knoten alle eingehenden Kanten: $\mathcal{O}(|V| + |E|)$.
3. Besuche jeden Knoten v in Reihenfolge der topologischen Sortierung und betrachte die Eingangs-Kanten: $\mathcal{O}(|V| + |E|)$.

$$\texttt{dist}[v] = \begin{cases} 0 & \text{keine Kanten,} \\ \max\limits_{(u,v)\in E} \{\texttt{dist}[u] + c(u,v)\} & \text{sonst.} \end{cases}$$

In-Class-Exercises: Längster Pfad in DAGs

Lösung:

1. Topologisch Sortieren. Laufzeit: $\mathcal{O}(|V| + |E|)$.
2. Berechne für jeden Knoten alle eingehenden Kanten: $\mathcal{O}(|V| + |E|)$.
3. Besuche jeden Knoten v in Reihenfolge der topologischen Sortierung und betrachte die Eingangs-Kanten: $\mathcal{O}(|V| + |E|)$.

$$\texttt{dist}[v] = \begin{cases} 0 & \text{keine Kanten,} \\ \max\limits_{(u,v)\in E} \{\texttt{dist}[u] + c(u,v)\} & \text{sonst.} \end{cases}$$

Vorgänger merken!

Fragen oder Anregungen?