Informatik II

Übung 11

FS 2019

Heutiges Programm

1. Letzte Woche: BFS mit Lazy Deletion
2. Adjazenzlisten in Java, Fortsetzung
3. Wiederholung Vorlesung
   - Algorithmus von Dijkstra
   - Algorithmus von Bellman-Ford
4. In-Class-Exercise (theoretisch)
5. In-Class-Exercise (praktisch)

BFS mit Lazy Deletion

```java
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

```java
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(), ...
}
```

3. Wiederholung Vorlesung

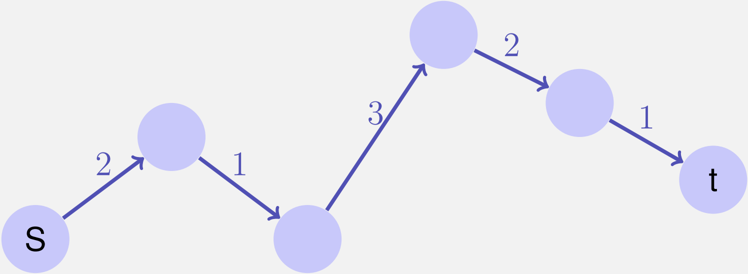
Gewichtete Graphen

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

Kürzeste Wege

Gewicht eines kürzesten Weges von u nach v:

$$\delta(u,v) = \begin{cases} \infty & \text{kein Weg von } u \text{ nach } v \\ \min\{c(p) : u \overset{p}{\rightsquigarrow} v\} & \text{sonst} \end{cases}$$

Zutaten für einen Algorithmus

Gesucht: Kürzeste Wege von einem Startknoten s aus.

- Gewicht des kürzesten bisher gefundenen Pfades

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

  Zu Beginn: $d_s[v] = \infty$ für alle Knoten $v \in V$.
  Ziel: $d_s[v] = \delta(s, v)$ für alle $v \in V$.
- Vorgänger eines Knotens

  $$\pi_s : V \to V$$

  Zu Beginn $\pi_s[v]$ undefiniert für jeden Knoten $v \in V$

Allgemeiner Algorithmus

1. Initialisiere d_s und π_s: $d_s[v] = \infty$, $\pi_s[v] =$ null für alle $v \in V$
2. Setze $d_s[s] \leftarrow 0$
3. Wähle eine Kante $(u, v) \in E$

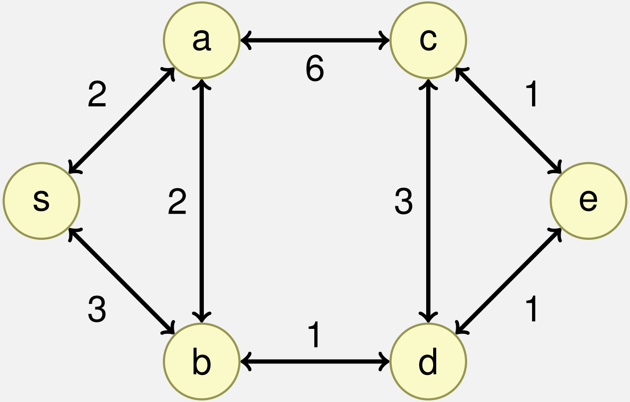
   Relaxiere (u, v):

   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. Wiederhole 3 bis nichts mehr relaxiert werden kann.
   (bis $d_s[v] \leq d_s[u] + c(u, v) \quad \forall (u, v) \in E$)

Annahme

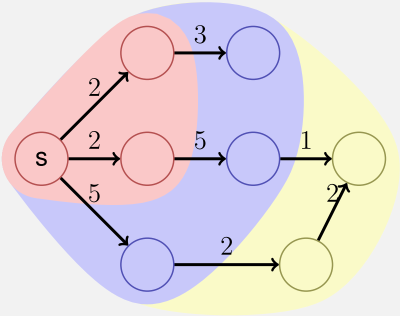


Alle Gewichte von G sind *positiv*.

Grundidee

Menge V aller Knoten wird unterteilt in

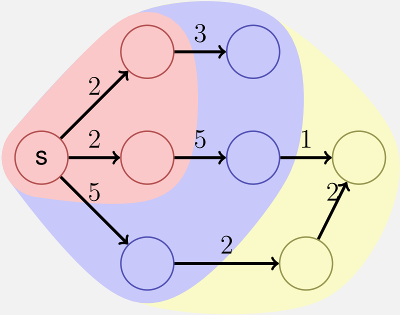
- die Menge M von Knoten, für die schon ein kürzester Weg von s bekannt ist
- die Menge $R = \bigcup_{v \in M} N^+(v) \setminus M$ von Knoten, für die kein kürzester Weg bekannt ist, die jedoch von M direkt erreichbar sind.
- die Menge $U = V \setminus (M \cup R)$ von Knoten, die noch nicht berücksichtigt 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 w gefunden, kann kein Pfad über später gefundene Knoten (mit Gewicht $\geq w$) zu einer Verbesserung führen.



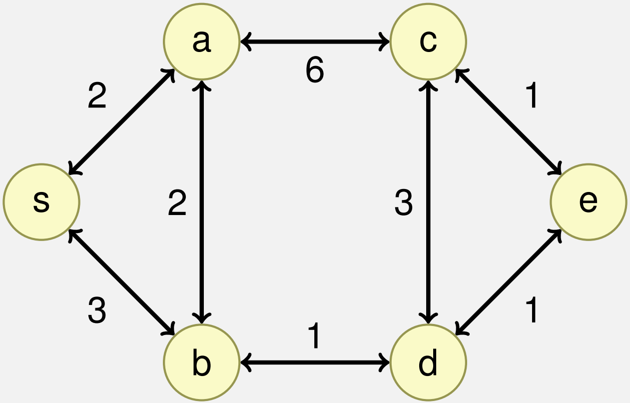
Algorithmus Dijkstra(G, s)

Input: Positiv gewichteter Graph $G = (V, E, c)$, Startpunkt $s \in V$
Output: Minimale Gewichte d der kürzesten Pfade und Vorgängerknoten für jeden Knoten.

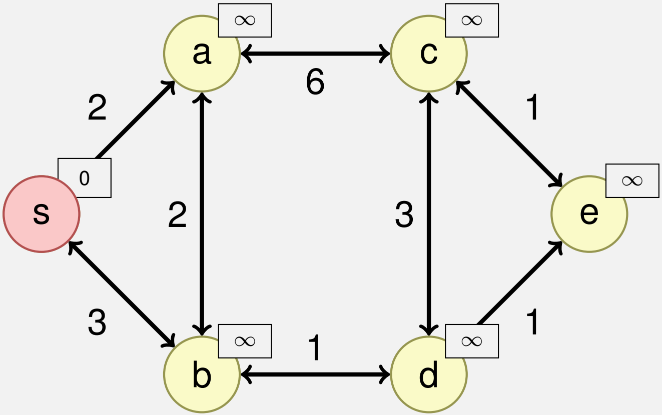
foreach $u \in V$ **do**
  $d_s[u] \leftarrow \infty$; $\pi_s[u] \leftarrow$ null

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

Beispiel



Beispiel

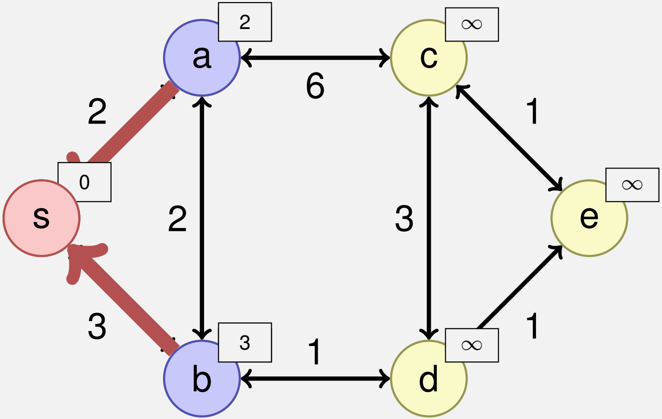


$$M = \{s\}$$

$$R = \{\}$$

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

Beispiel

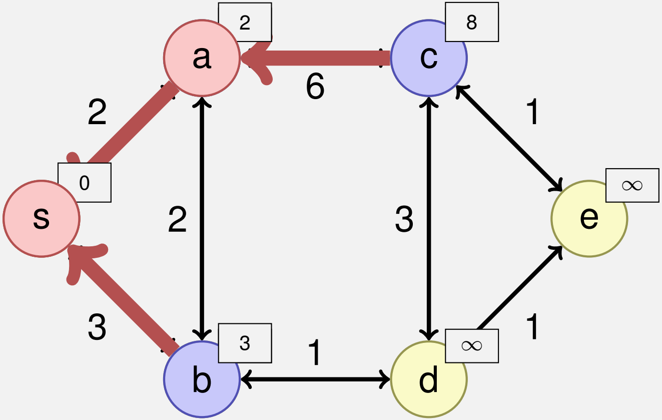


$$M = \{s\}$$

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

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

Beispiel

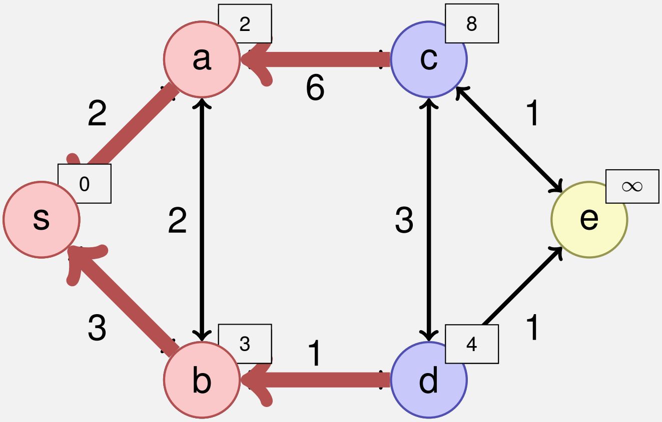


$$M = \{s, a\}$$

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

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

Beispiel

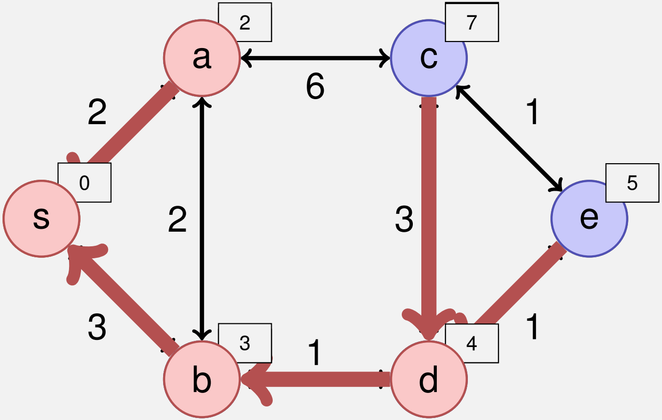


$$M = \{s, a, b\}$$

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

$$U = \{e\}$$

Beispiel

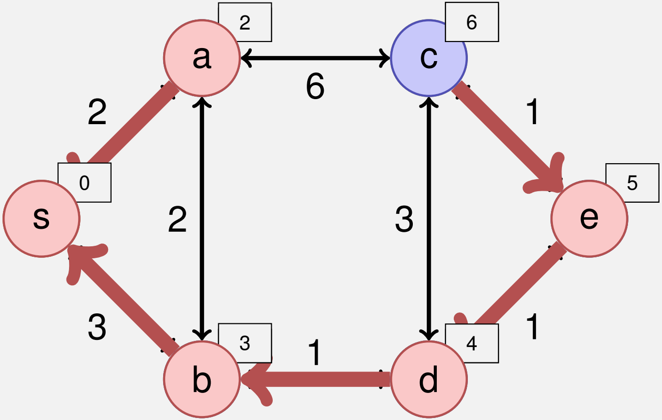


$$M = \{s, a, b, d\}$$

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

$$U = \{\}$$

Beispiel

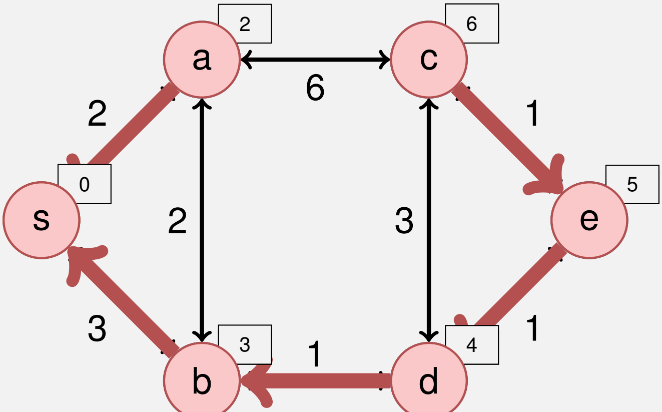


$$M = \{s, a, b, d, e\}$$

$$R = \{c\}$$

$$U = \{\}$$

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 ∈ N⁺(u) do
    if d_s[u] + c(u,v) < d_s[v] then
        d_s[v] ← d_s[u] + c(u,v)
        π_s[v] ← u
        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: $\mathcal{O}(|E|\log|V|)$.

Allgemeine Bewertete Graphen

Verbesserungsschritt wie bei Dijkstra:

Relax(u, v)

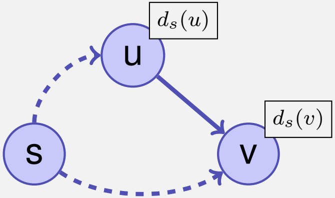
if $d_s(v) > d_s(u) + c(u, v)$ **then**

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

  $\pi_s(v) \leftarrow u$

  return true

return false



Problem: Zyklen mit negativen Gewichten können Weg verkürzen: es muss keinen kürzesten Weg mehr geben

Beobachtungen

- *Beobachtung 1:* Teilpfade von kürzesten Pfaden sind kürzeste Pfade: Sei $p = \langle v_0, \ldots, v_k \rangle$ ein kürzester Pfad von v_0 nach v_k. Dann ist jeder der Teilpfade $p_{ij} = \langle v_i, \ldots, v_j \rangle$ $(0 \leq i < j \leq k)$ ein kürzester Pfad von v_i nach v_j.
  Beweis: wäre das nicht so, könnte man einen der Teilpfade kürzen, Widerspruch zur Voraussetzung.
- *Beobachtung 2:* Wenn es einen kürzesten Weg gibt, dann ist dieser einfach, hat also keine doppelten Knoten.
  Folgt direkt aus Beobachtung 1.

Dynamic Programming Ansatz (Bellman)

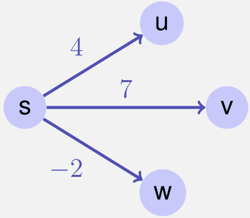
Induktion über Anzahl Kanten. $d_s[i, v]$: Kürzeste Weglänge von s nach v über maximal i Kanten.

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

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

Dynamic Programming Ansatz (Bellman)

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



Algorithmus: Iteriere über letzte Zeile bis die Relaxationsschritte keine Änderung mehr ergeben, maximal aber $n-1$ mal. Wenn dann noch Änderungen, dann gibt es keinen kürzesten Pfad.

Algorithmus Bellman-Ford(G, s)

Input: Graph $G = (V, E, c)$, Startpunkt $s \in V$
Output: Wenn Rückgabe true, Minimale Gewichte d der kürzesten Pfade zu jedem Knoten, sonst kein kürzester Pfad.

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

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

Zusammenfassung

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

Problem	Methode	Laufzeit	dicht $m \in \mathcal{O}(n^2)$	dünn $m \in \mathcal{O}(n)$
$c \equiv 1$	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	$\mathcal{O}((m+n)\log n)$	$\mathcal{O}(n^2 \log n)$	$\mathcal{O}(n \log n)$
allgemein	Bellman-Ford	$\mathcal{O}(m \cdot n)$	$\mathcal{O}(n^3)$	$\mathcal{O}(n^2)$

4. In-Class-Exercise (theoretisch)

In-Class-Exercises: Längster Pfad in DAGs

Das Kürzeste-Pfad-Problem hat einfache Lösungen (BFS, Dijkstra, Bellman-Ford). 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, Bellman-Ford). 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, Bellman-Ford). 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: $\mathcal{O}(|V| + |E|)$.

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

In-Class-Exercises: Längster Pfad in DAGs

Lösung:

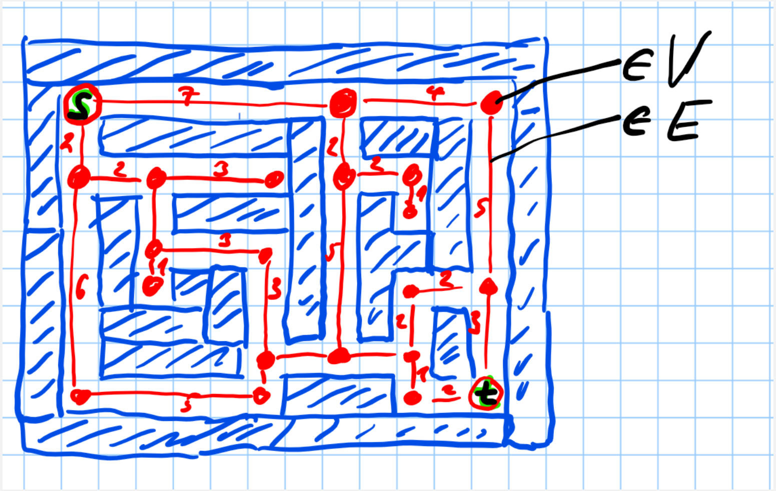
1. Topologisch Sortieren. Laufzeit: $\mathcal{O}(|V| + |E|)$.
2. Berechne für jeden Knoten alle eingehenden Kanten: $\mathcal{O}(|V| + |E|)$.
3. Besuche jeden Knoten v in Reihenfolge der topologischen Sortierung und betrachte die Eingangs-Kanten: $\mathcal{O}(|V| + |E|)$.

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

Vorgänger merken!

5. In-Class-Exercise (praktisch)

Wege in einem Labyrinth



Fragen oder Anregungen?