Informatik II

Übung 9

FS 2020

Heutiges Programm

1. Wiederholung BFS
2. Wiederholung Vorlesung
3. In-Class-Exercise (praktisch)

Python: Datenstrukturen

- Lists, Stacks, Queues:
  https://docs.python.org/3/tutorial/datastructures.html
- Heap: https://docs.python.org/3/library/heapq.html
- (Synchronized) Queue, PriorityQueue (=Heap):
  https://docs.python.org/3/library/queue.html

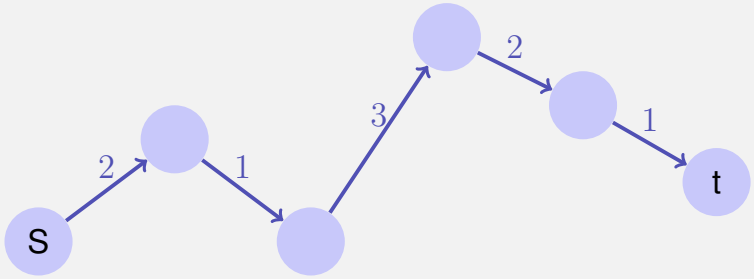
BFS

```
def BFS(v):
  color = {v: Grey} # white if not contained
  queue = Queue();
  queue.put(v);
  while not queue.empty():
    u = queue.get()
    for e in u.edges:
      w = e.target
      if w not in color:
        color[w] = Grey
        print(w)
        queue.put(w)
    color[u] = Black
```

2. 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:

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

Allgemeiner Algorithmus

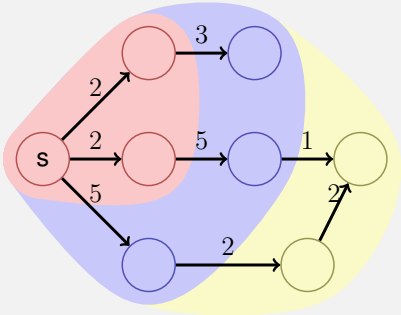
1. Initialisiere d_s und π_s: $d_s[v] = \infty$, $\pi_s[v] =$ null für alle $v \in V$
2. Setze $d_s[s] \leftarrow 0$
3. Wähle eine Kante $(u, v) \in E$

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

Dijkstra (positive Kantengewichte)

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.

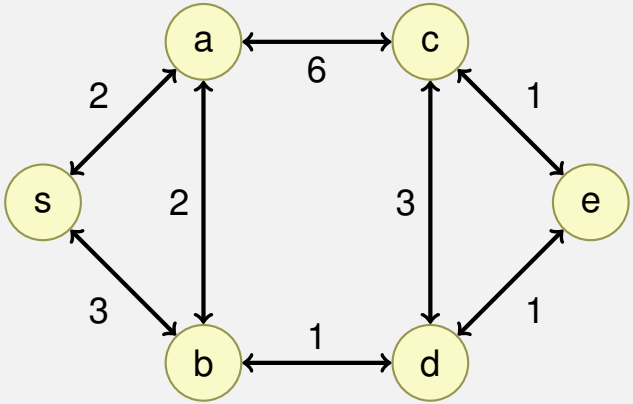


Algorithmus Dijkstra(G, s)

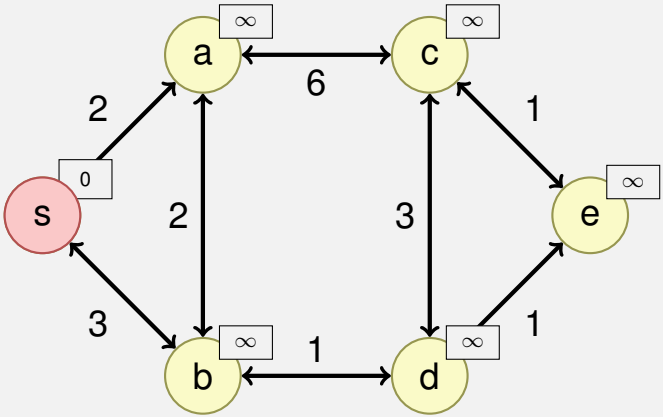
Input: Positiv gewichteter Graph $G = (V, E, c)$, Startpunkt $s \in V$
Output: Minimale Gewichte d der kürzesten Pfade und Vorgängerknoten für jeden Knoten.

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

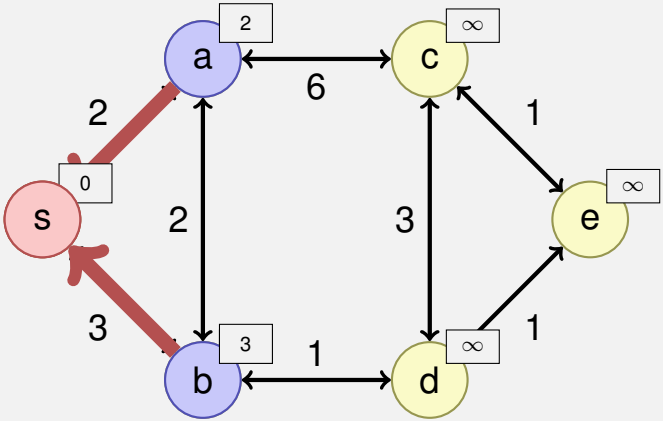


$$M = \{s\}$$

$$R = \{\}$$

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

Beispiel

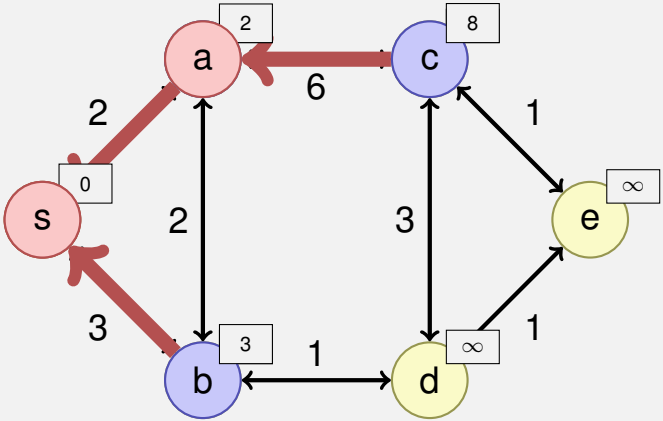


$$M = \{s\}$$

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

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

Beispiel

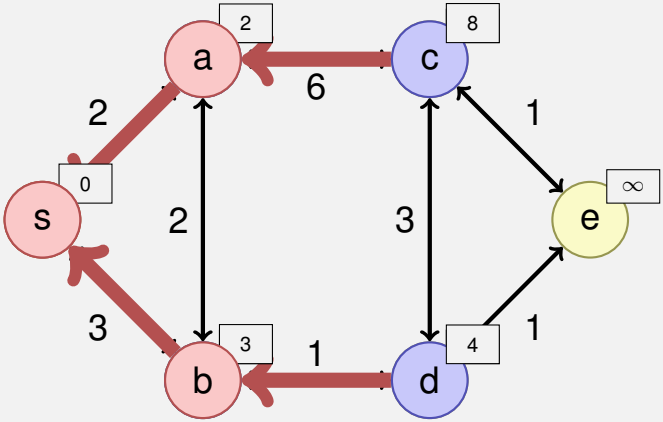


$$M = \{s, a\}$$

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

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

Beispiel

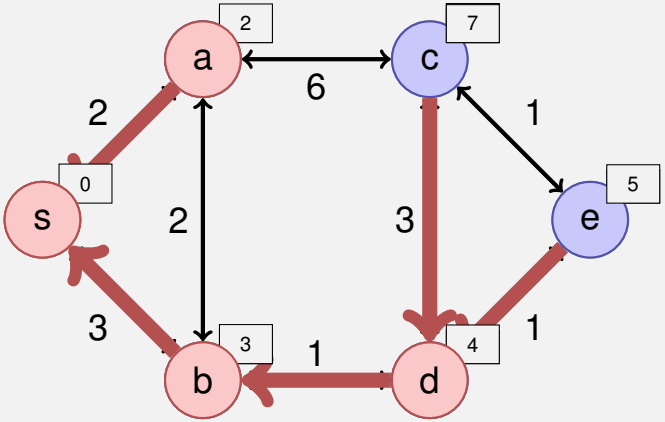


$$M = \{s, a, b\}$$

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

$$U = \{e\}$$

Beispiel

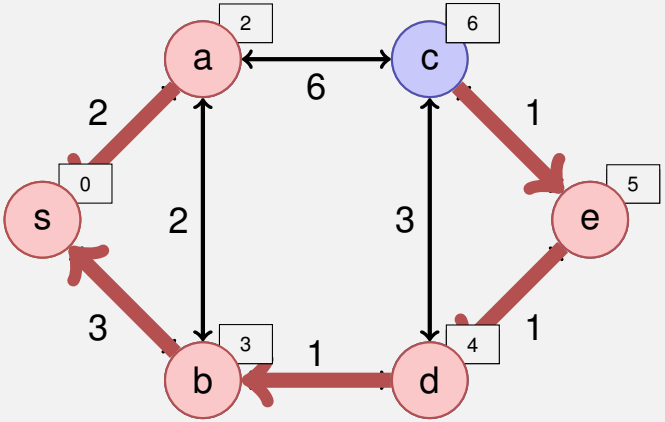


$$M = \{s, a, b, d\}$$

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

$$U = \{\}$$

Beispiel

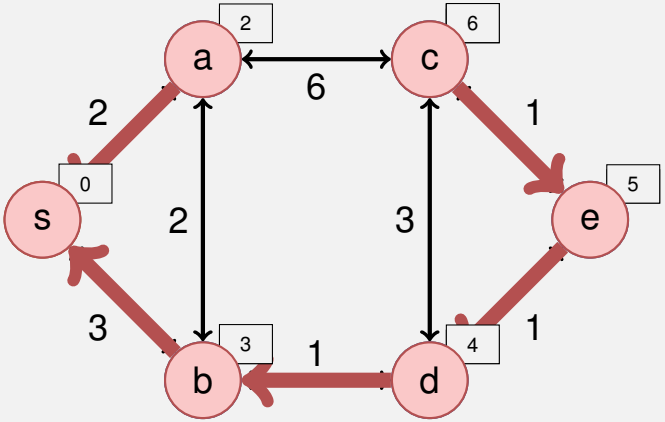


$$M = \{s, a, b, d, e\}$$

$$R = \{c\}$$

$$U = \{\}$$

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 $v \in N^+(u)$ **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 eines $d(v)$ im Heap zu R
    else
      $R \leftarrow R \cup \{v\}$ // 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 Dijkstra

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

- $n\times$ ExtractMin: $\mathcal{O}(n \log n)$
- $m\times$ Insert oder DecreaseKey: $\mathcal{O}(m \log |V|)$
- $1\times$ Init: $\mathcal{O}(n)$
- Insgesamt: $\mathcal{O}((n+m) \log n)$. Für zusammenhängende Graphen: $\mathcal{O}(m \log n)$

Zusammenfassung

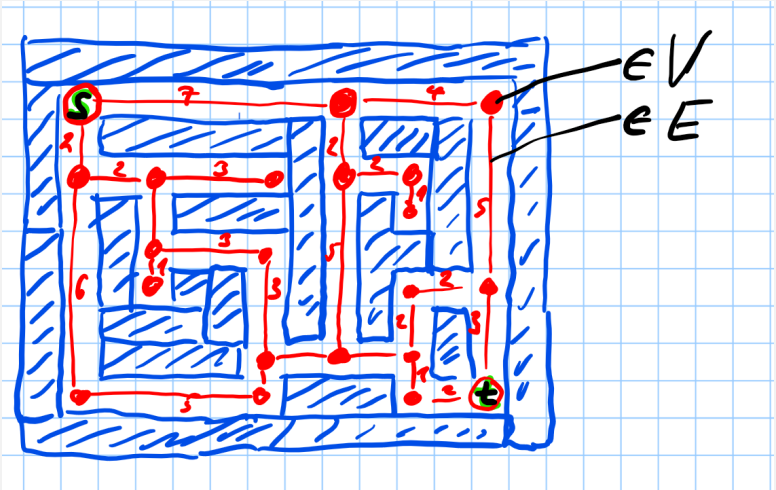
$n := |V|, m := |E|$

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

[1] kommt später in der Vorlesung (Dynamische Programmierung)

3. In-Class-Exercise (praktisch)

Wege in einem Labyrinth



BFS

```
color = {s: Grey}
dist = {s:0}
predecessor = {s: None}
queue = Queue();
queue.put(s);
while not queue.empty():
  u = queue.get()
  for e in u.edges:
    w = e.target
    if w not in color: # color is white
      color[w] = Grey
      predecessor[w] = u
      dist[w] = dist[u] + e.weight
      queue.put(w)
  color[u] = Black
```

Lösung Dijkstra

```
dist = {s:0}
predecessor = {s: None}
R = PriorityQueue()
R.put(PriorityEntry(0,s))
while not R.empty():
  p = R.get()
  u = p.data
  if p.priority == dist[u]: # lazy deletion
    for e in u.edges:
      v = e.target
      w = dist[u] + e.weight
      if v not in dist or w < dist[v]:
        dist[v] = w
        predecessor[v] = u
        R.put(PriorityEntry(w,v))
```

Fragen oder Anregungen?