technische universität dortmund

SoSe 2018

M. Buchin
A. Krivošija, D. Mezlaf
L. Pfahler, A. Rey

DAP2 – Heimübung 12

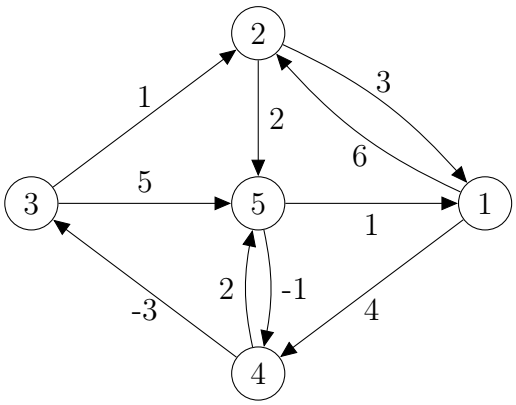
Ausgabedatum: 22.06.2018 — Abgabedatum: Mo. 02.07.2018 bis 12 Uhr

Abgabe:

Schreiben Sie unbedingt immer Ihren vollständigen Namen, Ihre Matrikelnummer und Ihre Gruppennummer auf Ihre Abgaben! Beweise sind nur dort notwendig, wo explizit danach gefragt wird. Eine Begründung der Antwort wird allerdings *immer* verlangt.

Aufgabe 12.1 (5 Punkte): (Bellman-Ford Algorithmus)

Gegeben sei der folgende gewichtete, gerichtete Graph (G, w):



a) Das Array $d[1..5]$ speichert an Position i, $i \in \{1, \ldots, 5\}$, die aktuelle Distanz von Startknoten $s = 1$ zu Knoten i. Wenden Sie den Algorithmus von Bellman und Ford auf den gegebenen Graphen an und geben Sie den Inhalt des Arrays d nach der Initialisierung sowie nach jedem Durchlauf der äußeren **for**-Schleife an. Führen Sie so viele Iterationen dieser **for**-Schleife durch, wie es nötig ist.

b) Erläutern Sie in einem kurzen Satz, ob der gegebene Graph negative Zyklen enthält oder nicht. Argumentieren Sie dabei anhand ihrer zuvor berechneten Lösung.

Lösung:

a) Die Werte, die im Array $d[1..5]$ gespeichert sind, sind wie folgt:

Nach der Initialisierung ist $d = [0, \infty, \infty, \infty, \infty]$

Nach Durchlauf $i = 1$ ist $d = [0, 6, \infty, 4, \infty]$

Nach Durchlauf $i = 2$ ist $d = [0, 6, 1, 4, 6]$

Nach Durchlauf $i = 3$ ist $d = [0, 2, 1, 4, 6]$

Nach Durchlauf $i = 4$ ist $d = [0, 2, 1, 4, 4]$

b) Wir führen noch eine Iteration der **for**-Schleife im Bellman-Ford Algorithmus durch, und erhalten das Array:

Nach Durchlauf $i = 5$ ist $d = [0, 2, 1, 3, 4]$

Da sich mindestens einer der Einträge im Array d in der fünften Iteration geändert hat, enthält der Graph G nach Lemma 55 (Vorlesung 18, Folie 83) einen negativen Zyklus: $(2) \to (5) \to (4) \to (3)$.

Aufgabe 12.2 (5 Punkte): (Dijkstra Algorithmus)

Gegeben sei der gewichtete, gerichtete Graph (G, w) ohne negative Kantengewichte.

a) Geben Sie einen Algorithmus an, der für einen Startknoten s und einen Zielknoten z einen kürzesten Weg von s nach z berechnet, wobei der Pfad mit ausgegeben werden soll. Sie dürfen den Dijkstra-Algorithmus als Unterfunktion aufrufen, diesen aber nicht verändern. Geben Sie die Worst-Case-Laufzeit ihres Algorithmus in der O-Notation an und begründen Sie sie.

b) Wenn man, wie in der Teilaufgabe a), einen kürzesten Weg von einem Startknoten s zu einem Zielknoten z berechnen möchte (und nicht zu allen anderen Knoten im Graphen), kann man den Dijkstra-Algorithmus früher abbrechen. Geben Sie an, wann und warum man frühestens abbrechen kann, wenn man vor dem Aufruf des Dijkstra-Algorithmus ein Preprocessing mit Laufzeit $O(n+m)$ erlaubt ($n = |V|$ ist die Anzahl der Knoten, $m = |E|$ ist die Anzahl der Kanten, wobei $G = (V, E)$).

Hinweis: Eine Verbesserung der Worst-Case-Laufzeit in O-Notation ist durch den früheren Abbruch nicht zu erwarten.

Lösung:

a) Vorab: Die Lösung, sich während des Dijkstras die Vorgänger zu merken, ist hier explizit ausgeschlossen, da der Dijkstra-Algorithmus nicht verändert werden darf. Stattdessen überlegen wir uns, wie man eine optimale Lösung zurückrechnen kann, ganz ähnlich dazu, wie beim Rucksackproblem eine optimale Lösung berechnet wurde.

Allerdings müssen wir dazu Kanten "rückwärts" benutzen. Damit wir das können, berechnen wir uns in einem Preprocessingschritt den Graphen, in dem alle Kanten umgedreht sind.

Input: Ein gerichteter Graph $\vec{G}= (V, \vec{E})$, wobei $\vec{E}$ implizit durch $\text{Adj}_{\vec{E}}$ gegeben ist

Output: Der gerichtete Graph $\overleftarrow{G}= (V, \overleftarrow{E})$ mit $\overleftarrow{E}= \{(v,u)|(u,v) \in \vec{E}\}$, wobei $\overleftarrow{E}$ implizit durch $\text{Adj}_{\overleftarrow{E}}$ gegeben ist

```
1  GraphDrehen(G→)
2      foreach u ∈ V do
3          Adj←E[u] ← ∅                          Alle Adjazenzlisten initialisieren
4      foreach u ∈ V do
5          foreach v ∈ Adj→G[u] do
6              Einfügen(Adj←E[v],u)              Siehe Vorlesung 11, Folie 39
7      return G←= (V, E←)
```

Diese Funktion hat Laufzeit $O(n + m)$, da einmal über alle Knoten und einmal über alle Kanten iteriert wird, und jeder Schritt die Laufzeit $O(1)$ benötigt. Mit Hilfe von $\overleftarrow{G}$ können wir jetzt für jeden Knoten die Knoten abrufen, von denen aus er erreicht werden kann. Damit können wir rekursiv einen kürzesten Weg von Startknoten s zu Zielknoten z berechnen, wenn wir alle d-Werte kennen: Dazu suchen wir uns immer einen Knoten v, von dem aus wir den aktuellen Knoten u erreichen können (am Anfang ist $u = z$), und für den $d[v] + w(v, u) = d[u]$ gilt. Einen solchen Knoten muss es geben, da $d[v]$ die Länge eines kürzesten Weges angibt, und die Gleichung daher für jeden Vorgänger von u auf einem kürzesten Weg von s nach u gilt. Außerdem bilden ein kürzester Weg von s zu v zusammen mit der Kante von v nach u einen kürzesten Weg zu u. Daher merken wir uns v. Anschließend setzen wir $u \leftarrow v$ und laufen auf diese Weise durch den Graphen, bis wir s erreichen.

Input: Ein gewichteter, gerichteter Graph $\vec{G}$, dessen Kantengewichte durch w gegeben sind, ein Startknoten s und ein Zielknoten z

Output: Ein kürzester Pfad von s nach z als verkettete Liste

```
1  KürzesterPfad(G→, w, s, z)
2      new array d[1...n]
3      d ← Dijkstra(G→,w,s)              Speichert die kürzeste-Wege-Distanzen in d
4      G←← GraphDrehen(G→)
5      P ← leere Liste
6      u ← z
7      while u ≠ s do
8          Einfügen(P,u)                          Siehe Vorlesung 11, Folie 39
9                  Beachte: Funktion setzt neues Element an den Anfang der Liste
10         foreach v ∈ Adj←E[u] do          Alle möglichen Vorgänger durchgehen
11             if d[v] + w(v,u) = d[u] then      Kann v Vorgänger von u sein?
12                 t ← v                          Dann v zwischenspeichern!
13         u ← t                  Wir gehen zum letzten gefundenen Knoten weiter.
14     Einfügen(P,s)                                              s fehlt noch
15     return P
```

Der Aufruf des Dijkstra-Algorithmus in Zeile 3 hat Laufzeit $O((n+m) \log n)$. Zeile 4 hat Laufzeit $O(n + m)$. Zeile 5 und 6 haben konstante Laufzeit. Die While-Schleife in Zeile 7-13 wird so oft ausgeführt wie der kürzeste Pfad lang ist, und da kürzeste Pfade keinen

Knoten doppelt enthalten, also maximal n mal. Zeile 8 hat konstante Laufzeit, trägt also zur Gesamtlaufzeit $O(n)$ bei, ebenso Zeile 13. Die Zeilen 11 und 12 werden einmal für jede Kante ausgeführt, die von einem Knoten auf dem kürzesten Pfad ausgeht. Das können schlimmstenfalls alle Kanten des Graphen sein, so dass Zeile 11 und 12 zur Gesamtlaufzeit im Worst-Case $O(m)$ beitragen. Zeile 14 und 15 haben konstante Laufzeit. Damit ist die Laufzeit insgesamt $O((n+m)\log n)$ (da $O(n+m)$ durch $O((n+m)\log n)$ dominiert wird).

b) Für das Preprocessing verwenden wir eine Breitensuche, um die Anzahl der eingehenden Kanten für jeden Knoten zu bestimmen. Wir wandeln dazu den Pseudocode aus Vorlesung 16, Folie 19 ab, um den Eingangsgrad für alle Knoten zu bestimmen. In der folgenden Funktion haben wir Zeile 6 ergänzt. Nach Zeile 8 haben wir zwei Zeilen zur Berechnung der Distanz und des Vorgängers gestrichen, die wir für unsere Anwendung hier nicht benötigen. Der Pseudocode ist ursprünglich für ungerichtete Graphen eingeführt worden, funktioniert aber auch für gerichtete. Breitensuche hat Worst-Case-Laufzeit $O(n+m)$, was durch die zusätzliche Zeile nicht verändert wird, also ist unser Preprocessing erlaubt. Die Anzahl der Kanten, die auf einen Knoten v zeigen, speichern wir in $deg[v]$. Das Array deg wird für alle Knoten mit 0 initialisiert.

```
   Input: Ein Graph G, ein Startknoten s
1  BFS(G,s)
2      initialisiere BFS
3      while Q ≠ ∅ do
4          u ← head[Q]
5          foreach v ∈ Adj[u] do
6              deg[v] ← deg[v] + 1
7              if color[v]=weiß then
8                  color[v] ← grau
9                  enqueue(Q,v)
10             dequeue(Q)
11             color[u] ← schwarz
```

Während des Dijkstra-Durchlaufs können wir abbrechen, sobald der Dijkstra alle in unseren Zielknoten z eingehenden Kanten gefunden hat. Zu diesem Zeitpunkt kann sich nämlich der d-Wert von z nicht mehr ändern, somit haben wir bereits einen kürzesten Weg gefunden.

Um die Abbruchbedingung zu realisieren, zählen wir mit, wie oft wir z bereits gefunden haben.

Input: Ein gewichteter, gerichteter Graph G, dessen Kantengewichte durch w gegeben sind, die Anzahl eingehender Kanten für alle Knoten in G, ein Startknoten s und ein Zielknoten z

Output: Die Länge der kürzesten Wege von s zu z

```
1  DIJKSTRA(G, w, deg, s, z)
2      Initialisiere SSSP
3      Q ← V[G]
4      counter ← 0
5      while Q ≠ ∅ do
6          u ← EXTRACTMIN(Q)
7          if color[u]=weiß then
8              color[u] ← schwarz
9              foreach v ∈ Adj[u] do
10                 if d[u] + w(u, v) < d[v] then
11                     d[v] ← d[u] + w(u, v)
12                     DECREASEKEY(v, d[v])
13                 if v = z then
14                     counter ← counter + 1
15                     if counter = deg[z] then
16                         return d[z]
```

Es ist wichtig, zu bemerken, dass man *nicht* abbrechen kann, sobald man z zum ersten Mal sieht, da sich der d-Wert später noch ändern kann. Es ist natürlich möglich, einfach zu warten, bis z aus Q extrahiert wird, was aber deutlich länger dauern kann (z.B. wenn die letzte Kante auf dem kürzesten Weg zu z sehr früh gefunden wird, aber einen sehr hohen Wert hat). Einen Vorteil in der Asymptote der Worst-Case-Laufzeit bringen diese Verbesserungen nicht, da es immer sein kann, dass die letzte eingehende Kante zu z die letzte ist, die vom Algorithmus betrachtet wird.