DAP2 – Präsenzübung 12

Besprechung: 12.07.2017 — 14.07.2017

Präsenzaufgabe 12.1: Floyd–Warshall-Algorithmus

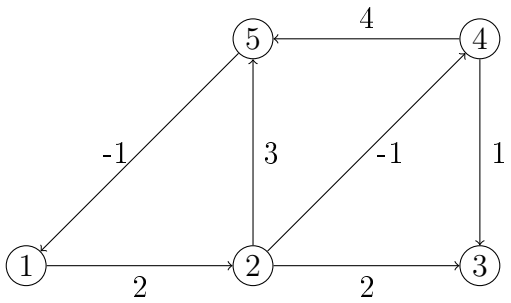
In dieser Aufgabe widmen wir uns dem Floyd–Warshall-Algorithmus.

Input: Ein gewichteter, gerichteter Graph G ohne negativen Zyklen, dessen Kantengewichte durch W gegeben sind und die Anzahl der Knoten n

Output: Für jedes Paar von Knoten $u, v \in V[G]$ die Distanz von u nach v sowie einen kürzesten Weg.

```
1 FloydWarshall(W, n)
2     D^(0) ← W
3     for k ← 1 to n do
4         for i ← 1 to n do
5             for j ← 1 to n do
6                 d_ij^(k) ← min{d_ij^(k−1), d_ik^(k−1) + d_kj^(k−1)}
7     return D^(n)
```

a) Führen Sie den Floyd–Warshall-Algorithmus für den folgenden Graphen $G = (V, E)$ aus. Geben Sie dabei nach der Initialisierung, also für $k = 0$, und nach jeder Iteration k der äußersten **for**-Schleife die Distanzmatrix $D^{(k)}$ an.



b) Im Floyd–Warshall-Algorithmus benutzen wir die Distanzmatrizen $D^{(k)}$, $0 \leq k \leq n$. Zeigen Sie, dass man mit genau einer Distanzmatrix D auskommen kann. Geben Sie dazu die entsprechende Modifikation am Pseudocode an und begründen Sie die Korrektheit dieser Modifikation.

Lösung:

a) Wir betrachten folgende Tabellen.

$D^{(0)}$	1	2	3	4	5
1	0	2	∞	∞	∞
2	∞	0	2	-1	3
3	∞	∞	0	∞	∞
4	∞	∞	1	0	4
5	-1	∞	∞	∞	0

$D^{(1)}$	1	2	3	4	5
1	0	2	∞	∞	∞
2	∞	0	2	-1	3
3	∞	∞	0	∞	∞
4	∞	∞	1	0	4
5	-1	1	∞	∞	0

$D^{(2)}$	1	2	3	4	5
1	0	2	4	1	5
2	∞	0	2	-1	3
3	∞	∞	0	∞	∞
4	∞	∞	1	0	4
5	-1	1	3	0	0

$D^{(3)} = D^{(2)}$, da von 3 keine Kanten ausgehen.

$D^{(4)}$	1	2	3	4	5
1	0	2	2	1	5
2	∞	0	0	-1	3
3	∞	∞	0	∞	∞
4	∞	∞	1	0	4
5	-1	1	1	0	0

$D^{(5)}$	1	2	3	4	5
1	0	2	2	1	5
2	2	0	0	-1	3
3	∞	∞	0	∞	∞
4	3	5	1	0	4
5	-1	1	1	0	0

b) Zur Berechnung der Distanzmatrix $D^{(k)}$ benötigt man lediglich die Distanzmatrix $D^{(k-1)}$. Daher ist es höchstens nötig, zwei Distanzmatrizen zur selben Zeit im Speicher zu halten. Es ergibt sich bei diesem Vorgehen daher ein Speicheraufwand von $\Theta(|V|^2)$.

Tatsächlich kommen wir sogar mit nur einer einzigen Distanzmatrix aus, falls wir nur an dem Endergebnis interessiert sind. Dazu modifizieren wir Zeile 5 des Floyd–Warshall-Algorithmus:

Floyd–Warshall2(W, n):

```
1 D ← W
2 for k ← 1 to n do
3     for i ← 1 to n do
4         for j ← 1 to n do
5             d_ij ← min(d_ij, d_ik + d_kj)
6 return D
```

Der Floyd–Warshall-Algorithmus aus der Vorlesung berechnet

$$d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}).$$

Unser modifizierter Algorithmus führt in Zeile 5 folglich möglicherweise eine der folgenden drei alternativen Berechnungen durch, falls mindestens einer der verwendeten Werte bereits einem Update unterzogen wurde, bevor diese Berechnung geschieht:

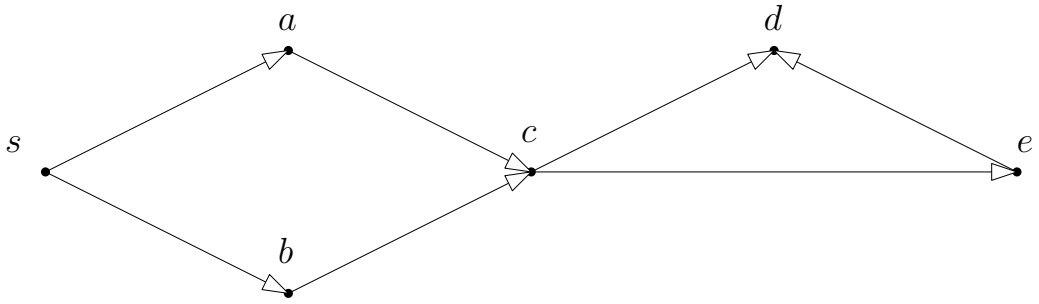
- $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k)} + d_{kj}^{(k-1)})$,
- $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k)})$ oder
- $d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k)} + d_{kj}^{(k)})$.

In jedem dieser Fälle wird die Länge eines kürzesten i-j-Pfades berechnet, wobei nur Knoten aus der Menge $\{1, 2, \ldots, k\}$ verwendet werden dürfen Falls nun ein Wert $d_{ik}^{(k)}$ anstatt $d_{ik}^{(k-1)}$ für die Berechnung verwendet wird, verwenden wir also einen Pfad von i nach k und dürfen dabei nur Knoten aus $\{1, 2, \ldots, k\}$ verwenden. Knoten k selbst kann aber nicht als Zwischenknoten auf so einem kürzesten Pfad liegen, da wir sonst einen Zyklus hätten, der nach Voraussetzung nicht negativ sein kann. Diesen könnte man also weglassen und hätte einen kürzeren i-k-Pfad. Daher gilt $d_{ik}^{(k)} = d_{ik}^{(k-1)}$. Analog kann man begründen, dass auch $d_{kj}^{(k)} = d_{kj}^{(k-1)}$ gilt. Daher bleibt die Korrektheit durch unsere Modifikation unberührt.

Präsenzaufgabe 12.2: (Graphenalgorithmen)

Gegeben sei ein gerichteter azyklischer Graph $G = (V, E)$ und der Startknoten $s \in V$. Uns interessiert die Anzahl der unterschiedlichen Pfade von s zu allen anderen Knoten aus V. Sei $P(u)$ die Anzahl der Pfade zum Knoten $u \in V$.

Im unten abgebildeten Graph ist z. B. $P(c) = 2$ und $P(d) = 4$.



a) Für alle Knoten $u \in V$ mit $(u, v) \in E$ sei $P(u)$ berechnet. Wie kann man $P(v)$ berechnen?

b) Entwerfen Sie einen Algorithmus `AnzahlPfade`(V, E, s), der die Anzahl unterschiedlicher Pfade $P(u)$ vom Startknoten s zu jedem Knoten $v \in V$ berechnet. Beschreiben Sie den Algorithmus zunächst mit eigenen Worten. Setzen Sie den Algorithmus dann in Pseudocode um. Für die volle Punktzahl wird ein Algorithmus erwartet, dessen Laufzeit durch $\mathcal{O}(|V| + |E|)$ beschränkt ist.

c) Analysieren Sie die Laufzeit Ihres Algorithmus.

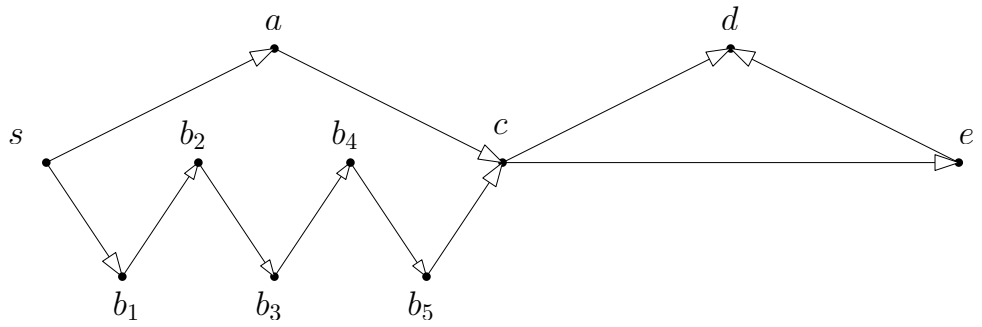
d) Beweisen Sie die Korrektheit Ihres Algorithmus.

Lösung:

a) Wenn für alle Knoten $u \in V$, sodass $(u, v) \in E$ ist, der Wert $P(u)$ bekannt ist, kann man $P(v)$ als $P(v) = \sum_{u \in V:(u,v) \in E} P(u)$ berechnen. Dabei ist $P(s) = 1$ (es gibt einen (leeren) Pfad von s nach s).

b) Um die Formel aus dem Aufgabenteil a) umzusetzen, muss man vorerst absichern, dass die $P(u)$-Werte aller Vorgängerknoten schon berechnet sind, bevor man den Wert des Knotens $v \in V$ berechnet.

Die beste verlangte Laufzeit weist auf eine angepasste Breitensuche hin. Leider reicht sie allein nicht. Ein Beispiel dazu ist auf dem Bild unten gegeben. Wenn man den Wert $P(c)$ mit der Breitensuche berechnen möchte, brauchte man $P(b_5)$ dafür. Die Breitensuche besucht aber erst die Knoten d und e, bevor sie zu b_5 kommt. Daher werden die nötige Werte im Moment des Rechnens von $P(c)$ nicht bekannt. Einen ähnlichen Beispiel kann man auch für direkte Anpassung der Tiefensuche ausdenken.



Wenn man aber zuerst durch die Tiefensuche (und entsprechende $f(v)$-Werte) die topologische Sortierung der Knoten berechnen würde, hätte man abgesichert, dass alle $P(u)$-Werte der Vorgängerknoten bekannt sind. Damit erfolgt folgender Algorithmus:

```
AnzahlPfade (V, E, s):
1.  P ← new Array [1..|V|]
2.  G ← (V, E)
3.  G ← Topologische Sortierung(G)
4.  for v ∈ V do
5.      Vor[v] ← nullptr
6.  for v ∈ V do
7.      for u ∈ Adj[v] do
8.          Vor[u] ← Vor[u] ∪ {v}                    /* Vorgängerliste erstellen */
9.  P[s] ← 1
10. for v ∈ V \ {s} do
11.     P[v] ← 0                                     /* die Formel aus a) umsetzen */
12.     for u ∈ Vor[v] do
13.         P[v] ← P[v] + P[u]
14. return P
```

c) Die Laufzeit der topologischen Sortierung ist die Zeit einer Tiefensuche, d. h. $\mathcal{O}(|V|+|E|)$. Gleiche Zeit benötigen wir für die Erstellung der Vorgängerliste in den Zeilen 6–8 sowie für die Umsetzung der Formel durch die verschachtelten Schleifen über allen Knoten und allen Kanten in den Zeilen 10–13. Für die Initialisierung des Arrays in der Zeile 1 sowie für die Initialisierung der Vorgängerliste (Zeilen 4–5) benötigen wir $\mathcal{O}(|V|)$ Zeit. Die Anweisungen in den Zeilen 2 und 9 sowie die Rückgabe in Zeile 14 benötigt konstante Zeit. Insgesamt ist die Laufzeit des Algorithmus $\mathcal{O}(|V| + |E|)$.

d) Wir zeigen die Korrektheit des Algorithmus mithilfe der Induktion nach (topologisch sortierten) Knoten aus V. Wir dürfen dies tun, da eine Ordnung der Knoten des Graphens vorhanden ist (was aber nicht immer der Fall ist). Die Behauptung lautet: Algorithmus `AnzahlPfade`(V, E, s) berechnet korrekt die Anzahl der unterschiedlichen Pfade von s zu den allen anderen Knoten aus $v \in V$.

I.A. $v = s$: Da der Startknoten s der erste Knoten in der topologischen Sortierung ist, ist die einzige Möglichkeit ein leerer Pfad von s nach s. Damit ist $P[s] = 1$ korrekt.

I.V. Sei $P[u]$ korrekt berechnet für alle u die in topologischer Sortierung vor v sind (d. h. für alle u mit $f(u) > f(v)$).

I.S. Wir berechnen $P[v] = \sum_{u \in Vor[v]} P[u]$, wobei $Vor[v]$ die Liste aller direkten Vorgängerknoten von v ist. Alle unterschiedliche Pfade von s nach v führen über die Kanten $(u, v) \in E$, d. h. nur über den Knoten aus $Vor[v]$. D. h. die Anzahl unterschiedlichen Pfade nach v ist die Summe der Anzahlen der Pfade nach Knoten aus $Vor[v]$. Da nach der topologischen Sortierung alle Knoten aus $Vor[v]$ bevor v sind, sind laut Induktionsvoraussetzung alle $P[u]$ korrekt berechnet. Somit wird auch $P[v]$ korrekt berechnet, was wir beweisen wollten.