10 Kürzeste Pfade

In diesem Kapitel setzen wir uns mit der Berechnung von kürzesten Pfaden in einem Graphen auseinander.

Definition 10.1 (Pfadgewichte).

i) Das *Gewicht eines Pfades* p = (v_0, v_1, . . . , v_k) ist die Summe der Kantengewichte, die zu den aufeinander folgenden Knotenpaaren gehören.

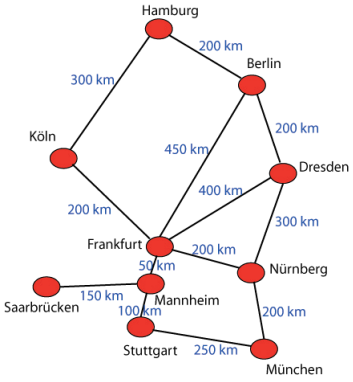
$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

ii) Das *Gewicht des kürzesten Pfades* von s nach v ist definiert als

$$\delta(s,v) = \begin{cases} min\{w(p) | s \rightarrow^p v\} & \text{falls ein Pfad existiert} \\ \infty & \text{sonst} \end{cases}$$

iii) Der *kürzeste Pfad* p von s nach v ist ein Pfad mit minimalem Gewicht: w(p) = δ(s,v)

Beispiel 10.1.



Beispiel eines gewichteten Graphen anhand eines Städtenetz, dessen Kanten durch die Entfernung gewichtet sind. Das Gewicht des kürzesten Pfades von Hamburg nach Nürnberg ist z.B.: δ(Hamburg,Nürnberg) = 700 km.

10.1 SSSP-Problem

Gegeben ein ungerichteter Graph G = (V,E) und eine Gewichtsfunktion w

$$w : E \rightarrow \mathbb{R}$$

Man berechne für einen vorgegebenen Knoten s die kürzesten Pfade zu allen anderen Knoten.

Single-Destination Shortest Path (SDSP)

Gegeben ein ungerichteter Graph G = (V,E) und eine Gewichtsfunktion w

$$w : E \to \mathbb{R}$$

Man berechne für einen vorgegebenen Knoten s die kürzesten Pfade von allen anderen Knoten zu v.

Lösung. Durch Umkehrung der Kantenrichtung führt man das SDSP-Problem auf ein SSSP-Problem zurück. ■

Single-Pair Shortest Path (SPSP)

Gegeben ein ungerichteter Graph G = (V,E) und eine Gewichtsfunktion w

$$w : E \to \mathbb{R}$$

Man berechne für eine vorgegebenes Knotenpaar (s,v) den kürzesten Pfad von s nach v.

Lösung. Durch Anwendung des SSSP-Problems auf s erhält man alle kürzesten Pfade, also auch den kürzesten Pfad nach v. ■

All-Pair Shortest Path (APSP)

Gegeben ein ungerichteter Graph G = (V,E) und eine Gewichtsfunktion w

$$w : E \to \mathbb{R}$$

Man berechne für jedes Knotenpaar (s,v) den kürzesten Pfad von s nach v.

Lösung. Durch Anwendung des SPSP-Problems auf alle Knotenpaare erhält man die Lösung dieses Problems. Es gibt jedoch effizientere Verfahren, die nicht auf SPSP basieren. ■

Lemma 10.1 (Teilpfade von kürzesten Pfaden sind kürzeste Pfade). *Gegeben ein gewichteter Graph G = (V,E) mit Gewichtsfunktion $w : E \to \mathbb{R}$.*
Sei p = (v_1, v_2, ... , v_k) ein kürzester Pfad von Knoten v_1 nach v_k und sei für jedes Paar i,j mit $1 \leq i \leq j \leq k$ p_{ij} = (v_i, v_{i+1}, ... , v_j) der Teilpfad von Knoten v_i nach Knoten v_j.
Dann ist p_{ij} ein kürzester Pfad von v_i nach v_j.

Beweis.

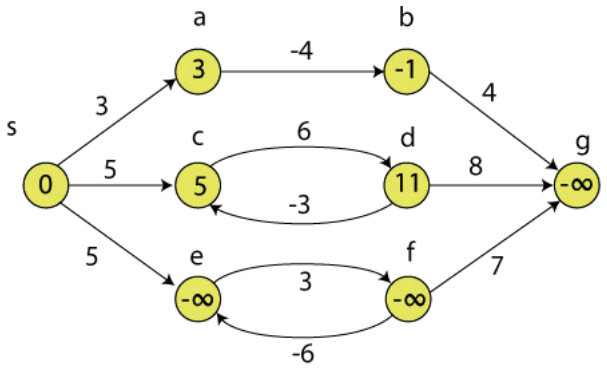
$$w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$$

Annahme: p_{ij} ist nicht der kürzeste Pfad von v_i nach v_j.
Damit folgt, daß es einen Pfad p'_{ij} gibt, mit $w(p'_{ij}) < w(p_{ij})$

$$\Rightarrow \qquad w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk}) > w(p_{1i}) + w(p'_{ij}) + w(p_{jk})$$

Damit haben wir einen kürzeren Pfad gefunden als p. ↯

Anmerkung 10.1. Die Gewichtsfunktion $w : E \to \mathbb{R}$ kann Kanten auch negative Gewichte zuordnen. Unter diesen Umständen spielen Kreise bzw. Rundgänge mit negativem Gesamtgewicht eine besondere Rolle.



Falls es einen negativen Zyklus auf dem Pfad gibt, dann ist der kürzeste Pfad nicht wohldefiniert, da man durch Hinzufügen weiterer Zyklusrunden das Pfadgewicht beliebig klein machen kann (in unserem Beispiel wäre dies der Pfad (s,e,f,e,f,e,f,. . .)). In diesem Fall setzt man das Gewicht des kürzesten Pfades $\delta(s, g) = -\infty$.

Lemma 10.2. *Gegeben ein gewichteter Graph G = (V,E) mit $|V| = n$ und $|E| = m$. Findet man zwei kürzeste Pfade mit gleichem Gewicht aber unterschiedlicher Anzahl an Kanten, betrachten wir nur den kürzesten Pfad mit der kleineren Anzahl an Kanten. Dann gilt:*
Die gesuchten kürzesten Pfade enthalten höchstens n Knoten und n-1 Kanten.

Beweis.
Wir zeigen: Es kommen keine Zyklen in den gesuchten kürzesten Pfaden vor.

i) Zyklen mit negativem Gewicht: Wie in Anm. 10.1 gezeigt, existiert kein wohldefinierter kürzester Pfad.

ii) Zyklen mit Gewicht größer 0: Findet man einen Pfad mit einem Zyklus, so verkleinert man das Gesamtgewicht durch Entfernen dieses Zyklus aus dem Pfad.

iii) Zyklen mit Gewicht 0: Durch Streichen des Zyklus aus dem Pfad verändert sich nicht das Gesamtgewicht, aber die Anzahl der Kanten des Pfades verkleinert sich. Nach Voraussetzung betrachten wir damit nur den Pfad mit der kleineren Anzahl der Kanten.

Also gilt, daß keine Zyklen im gesuchten kürzesten Pfad vorkommen, der kürzeste Pfad jeden Knoten nur einmal enthalten kann. Somit folgt die Behauptung. ■

10.2 Bellman-Ford Algorithmus

In diesem Abschnitt wird der Bellman-Ford Algorithmus zur Lösung des SSSP-Problems vorgestellt. Dieser Algorithmus verwendet eine Relaxationstechnik, die an dieser Stelle eingeführt werden soll.

Definition 10.2 (Relaxation). Eine Kante (u,v) zu *entspannen (relaxieren)*, bedeutet, daß man einen Test durchführt, ob man einen neuen kürzesten Pfad von s nach v erhält, wenn man den aktuell kürzesten Pfad von s nach u betrachtet und die Kante (u,v) hinzufügt.

Zur Berechnung benötigen wir spezielle Datenstrukturen:

- Die Gewichte der Kanten werden in einem Kantenfeld gespeichert.

  ```
  Kantenfeld<float> gewicht
  ```

- Die Gewichte der aktuell kürzesten Pfade zu den Knoten werden in einem Knotenfeld gespeichert.

  ```
  Knotenfeld<float> länge
  ```

- Die Vorgängerknoten auf den aktuell kürzesten Pfaden zu den Knoten werden in einem Knotenfeld gespeichert.

  ```
  Knotenfeld<Knoten> vorgänger
  ```

  Mit Hilfe dieses Feldes kann man den kürzesten Pfad zu einem Knoten v bestimmen.

Solange man den kürzesten Pfad von s zu einem Knoten v noch nicht gefunden hat, speichert man in `länge[v]` eine obere Schranke für die Länge des kürzesten Pfades von s nach v.
Die Knoten werden am Start wie folgt initialisiert (Initialisiere(G,s)):

```
Für alle Knoten v in E\{s}:     länge[v] = infinity
Für alle Knoten v in E:     vorgänger[v] = nil
                                länge[s] = 0
```

Relax(e,gewicht,länge,vorgänger):

```
(1) hilfe = länge[quelle(e)] + gewicht[e]
(2) Falls länge[ziel(e)] > hilfe
(3)      länge[ziel(e)] = hilfe
(4)      vorgänger[ziel(e)] = quelle(e)
```

Lemma 10.3 (Eigenschaften der Pfad Relaxation). *Falls p = (v_0, v_1, . . . , v_k) ein kürzester Pfad von s = v_0 nach v_k ist und die Kanten von p in der Reihenfolge (v_0,v_1), (v_1,v_2), . . . (v_{k-1},v_k) relaxiert werden, dann gilt:*

$$länge[v_k] = \delta(s, v_k)$$

Diese Eigenschaft gilt unabhängig von anderen Relaxationsschritten, die eventuell vorher, nachher oder auch zwischendurch ausgeführt werden.

Beweis. [mittels vollständiger Induktion]

Induktionsannahme: Nach der Relaxation von (v_{i-1},v_i) gilt:

$$länge[v_i] = \delta(s, v_i)$$

Induktionsstart: i=0: länge[s] = 0 = δ(s,s)

Induktionsschluß: Die Induktionsannahme gelte $\forall$ k $<$ i. Damit folgt:

$$\begin{aligned}
\delta(s, v_i) &= \sum_{k=1}^{i} w(v_{k-1}, v_k) \\
&= \sum_{k=1}^{i-1} w(v_{k-1}, v_k) + w(v_{i-1}, v_i) \\
&= \delta(s, v_{i-1}) + w(v_{i-1}, v_i) \\
&= länge[v_{i-1}] + w(v_{i-1}, v_i) \\
&= länge[v_i]
\end{aligned}$$

■

Mit diesem Wissen können wir nun den Bellman-Ford Algorithmus im Pseudocode vorstellen:

Bellman-Ford(G,w,s):

```
(1) Initialisiere(G,s)
(2) Für alle i = 1, ... ,n-1:
(3)      Für alle Kanten e = (u,v)∈E: Relax(e,w,länge,vorgänger)
(4) Für alle Kanten e = (u,v):
(5)      Falls länge[v] > länge[u] + w[e]
(6)           Gib FALSE zurück
(7) Gib TRUE zurück
```

Der Bellman-Ford Algorithmus löst den allgemeinen Fall des SSSP-Problems, d.h. negative Kantengewichte sind erlaubt. Der Rückgabewert ist eine boolsche Variable, die angibt ob erfolgreich alle kürzesten Pfade zu den von s aus erreichbaren Knoten bestimmt wurden oder ein negativer Zyklus gefunden wurde.

Laufzeit des Algorithmus

Im Folgenden wollen wir uns die Laufzeit des Bellman-Ford-Algorithmus näher ansehen.

Satz 10.1 (Laufzeit des Bellman-Ford-Algorithmus).
Der Bellman-Ford-Algorithmus hat eine Laufzeit $\mathcal{O}(nm)$, wobei $|V| = n$ und $|E| = m$.

Beweis. Die Initialisierung läuft in Θ(V) ab. In Zeile 4 wird für alle |V|-1 Durchläufe $\mathcal{O}$(E) Zeit benötigt. Schließlich wird die Schleife von Zeile 4-6 in $\mathcal{O}$(E) Zeit durchlaufen. Damit folgt die Behauptung. ■

Lemma 10.4. *Sei G = (V,E) ein gewichteter und gerichteter Graph mit Quelle s und Gewichtsfunktion $w : E \to \mathbb{R}$. Enthält der Graph G keine negativen Zyklen, die von s aus erreicht werden können, dann berechnet der Algorithmus die Gewichte (Längen) der kürzesten Pfade korrekt, d.h., für alle erreichbaren Knoten gilt:*

länge[v] = δ(s,v)

Beweis. Sei v ∈ V erreichbar von s und sei p = (v_0, ... ,v_k) mit s = v_0 und v = v_k ein azyklischer, kürzester Pfad von s nach v. Der Pfad p hat höchstens n-1 Kanten. Daher ist k<n. In Zeile 2 und 3 wird jede Kante in jeder der n-1 Iterationen relaxiert. Unter den Kanten, die in der i-ten Iteration entspannt werden, ist auch (v_{i-1},v_i). Aus Lemma 10.3 folgt daher:

länge[v] = länge[v_k] = δ(s,v_k) = δ(s,v)

■

Beweis der Korrektheit des Bellman-Ford-Algorithmus.
Angenommen der Graph G enthält keine negativ gewichteten Zyklen, die von s aus erreichbar sind. Nach der Terminierung gilt ∀v ∈ V: länge[v] = δ(s,v). Nach Lemma 10.4 gilt diese Aussage, wenn v von s erreichbar ist. Falls v nicht von s erreichbar ist, folgt die Behauptung aus Lemma 10.3. Der Vorgängergraph G_p = (V_p, E_p) zum Graphen G = (V,E) mit

$$\begin{aligned} V_p &= \{v \in V | vorgänger[v] \neq nil\} \cup \{s\} \\ E_p &= \{(vorgänger[v], v) \in E | v \in V_p\} \end{aligned}$$

ist ein kürzester-Pfad Baum. Der Beweis wird als leichte Fingerübung dem Leser überlassen. Bei der Terminierung des Algorithmus gilt ∀(u,v) ∈ E:

$$\begin{aligned} länge[v] &= \delta(s,v) \\ &\leq \delta(s,u) + w(u,v) \\ &= länge[u] + w(u,v) \end{aligned}$$

Damit folgt, dass in Zeile 5 keiner der Tests dazu führt, daß FALSE zurückgegeben wird, also liefert er Algorithmus TRUE zurück. Umgekehrt, nehmen wir daß G einen negativ gewichteten Zyklus z = (v_0,v_1 ... ,v_k) mit v_0 = v_k enthält, der von s aus erreichbar ist. Damit gilt:

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0 \tag{10.1}$$

Annahme: Der Bellman-Ford Algorithmus liefert TRUE zurück.
Damit folgt: $\delta[v_i] \leq länge[v_{i-1}] + w(v_{i-1}, v_i) für i = 1, 2, \ldots, k$

Summieren wir nun diese Ungleichungen entlang des Zyklus z, erhalten wir

$$\sum_{i=1}^{k} länge[v_i] \leq \sum_{i=1}^{k} länge[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Da jeder Knoten in z exakt einmal in jedem der ersten beiden Summen vorkommt, gilt:

$$\sum_{i=1}^{k} länge[v_i] = \sum_{i=1}^{k} länge[v_{i-1}]$$

Da der Algorithmus nur terminiert, wenn $länge[v_i] < \infty \; \forall i = 1, 2, \ldots, k$ gilt:

$$0 \leq \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

was der Ungleichung 10.1 widerspricht.
Damit folgt, daß der Bellman-Ford Algorithmus TRUE zurückliefert, wenn der Graph G keine negativ gewichteten Zyklen enthält, die von der Quelle erreichbar sind, sonst FALSE. ■

10.3 Der Algorithmus von Dijkstra

In diesem Abschnitt wird der Dijkstra-Algorithmus zur Lösung des SSSP-Problems für gerichtete Graphen mit Gewichtsfunktionen, die den Kanten nur positive Kantengewichten zuordnen vorgestellt.

Dijkstra(G,w,s):

```
(1) Initialisiere(G,s)
(2) S = ∅
(3) Q = V
(4) Solange Q ≠ V
(5)      u = Extrakt-Minimum(Q)
(6)      S = S ∪ {u}
(7)      Für jede Kante e = (u,v)∈E, die u verlässt:
(8)           Relax(e,w,länge,vorgänger)
```

Der Algorithmus verwaltet eine Knotenmenge S, für die bereits ein kürzester Pfad von der Quelle s aus identifiziert wurde.
Aus der restlichen Knotenmenge V\S wählt man in jeder Iteration den Knoten u mit dem kleinsten gespeicherten Wert für den aktuell kürzesten Pfad, addiert u zu S und entspannt alle Kanten, die u verlassen.

Satz 10.2 (Korrektheit des Dijkstra-Algorithmus). *Der Algorithmus von Dijkstra berechnet für einen gewichteten und gerichteten Graphen G = (V,E), eine Gewichtsfunktion $w : E \to \mathbb{R}$ mit nicht negativen Gewichten und eine Quelle s die kürzesten Pfade von s zu allen erreichbaren Knoten v ∈V korrekt. D.h.*

$$länge[v] = \delta(s, v)$$

Beweis. Wir beweisen die folgende Invariante für die Iteration: *Am Start jeder Iteration in der „Solange“-Schleife gilt für alle Knoten v ∈ S: länge[v] = δ(s,v)*
Es genügt dies für alle zu S hinzugefügten Knoten zu zeigen.

Initialisierung: S = ∅. Daher ist die Invariante erfüllt.

Aufrechterhaltung: Annahme: $\exists u \in S$ mit $länge[u] \neq \delta(s, u)$

Wir betrachten den Zeitpunkt zu dem u in S eingefügt wurde.

Es gilt: u ≠ s und S ≠ ∅

Es muss ein Pfad, also auch einen kürzesten Pfad von s nach u geben, da sonst aufgrund der Initialisierung $länge[u] = \delta(s, u) = \infty$.

Bevor u zu S addiert wird, gilt:

$$s \in S \text{ und } u \in V \backslash S$$

Sei y der erste Knoten auf dem kürzesten Pfad p von s nach u, der zu diesem Zeitpunkt nicht zu S gehört und x sein Vorgänger in S. Damit zerlegen wir p in folgende Komponenten:

$$s \xrightarrow{p_1} x \to y \xrightarrow{p_2} u$$

Da x ∈ S ist und vor u in S eingefügt wurde, gilt für x: $länge[x] = \delta(s, x)$

Als x zu S addiert wurde, wurde (x,y) entspannt.

$$\Rightarrow \qquad länge[y] = länge[x] + w(x, y) = \delta(s, x) + w(x, y) = \delta(s, y)$$

Da y auf dem kürzesten Pfad von s nach u auftaucht und $w(u, v) \geq 0 \quad \forall (u, v) \in E$, folgt:

$$\delta(s, y) \leq \delta(s, u)$$

$$länge[y] = \delta(s, y) \leq \delta(s, u) \leq länge[u]$$

Da aber beide Knoten u und y in Q = V\S waren, als u als Knoten mit minimalem Gewicht bestimmt wurde, gilt:

$$länge[y] = \delta(s, y) = \delta(s, u) = länge[u]$$

Da $\delta(s, u) = länge[u]$ erhalten wir einen Widerspruch zur Wahl von u.

Ende des Algorithmus: Am Ende ist Q = ∅ und S = V, so dass ∀v∈V

$$länge[v] = \delta(s, v)$$

Im Folgenden wollen wir die Laufzeit des Dijkstra-Algorithmus betrachten.

Satz 10.3 (Laufzeit des Dijkstra-Algorithmus).
Die amortisierte Laufzeit des Dijkstra-Algorithmus für einen gerichteten Graphen G = (V,E) mit |V| = n und |E| = m ist $\mathcal{O}(n \log(n) + m)$.

Beweis. Implementiert man die Extrakt-Minimum(Q) mittels eines Fibonacci-Heap (siehe Kap. **??**), so kann man in Zeit $\mathcal{O}(\log(n))$ das Minimum „extrahieren". Damit läuft die Zeile 5 in $\mathcal{O}(n \log(n))$. Entspannt man eine Kante, so ändert sich eventuell der aktuell kürzeste Pfad und man muss im Fibonacci-Heap eine Decrease-Key-Operation ausführen, die amortisierte Zeit $\mathcal{O}(1)$ kostet. Also benötigen Zeile 6 $\mathcal{O}(n)$ und Zeile 8 $\mathcal{O}(m)$ Zeit, womit die behauptete Laufzeit folgt. ■