1 Kürzeste Pfade in Graphen

Es sei ein gerichteter Graph $G = (V, E)$ mit $|V| = n$ Knoten, $|E| = m$ Kanten und Kantengewichten $c : E \to \mathbb{R}$ gegeben. Ein *Pfad* in G von $u \in V$ nach $v \in V$ ist eine Folge (x_0, x_1), (x_1, x_2), (x_2, x_3), …, (x_{r-1}, x_r) von Kanten aus E mit $x_0 = u$ und $x_r = v$. Der Pfad heißt *einfach*, wenn kein Knoten mehrfach vorkommt. Unter der *Länge* eines Pfades verstehen wir die Summe der Gewichte seiner Kanten, also $\sum_{i=1}^{r} c(x_{i-1}, x_i)$. Die Zahl der Kanten auf dem Pfad ist seine *Kantenlänge*.

In vielen Anwendungen ist man daran interessiert, *kürzeste* Pfade in einem Graphen zu berechnen, z.B. beim Routing von Verbindungen in Kommunikationsnetzwerken oder bei der Auswahl von Wegen in Navigationssystemen. Man unterscheidet dabei die folgenden Problemstellungen:

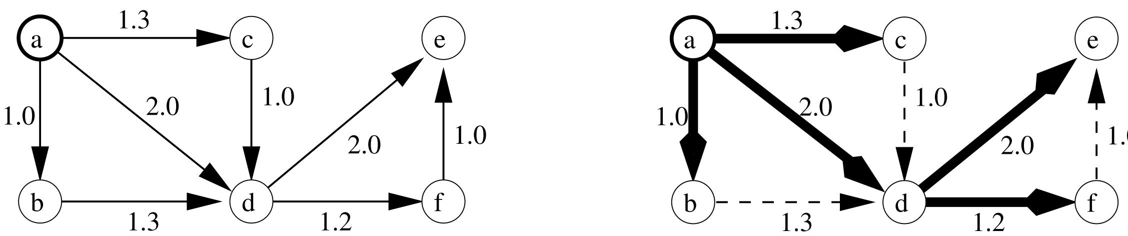
- *single-pair*: berechne für gegebene u und v einen kürzesten Pfad von u nach v
- *single-source*: berechne für einen gegebenen Startknoten v die kürzesten Pfade zu allen anderen Knoten
- *single-sink*: berechne für einen gegebenen Zielknoten v die kürzesten Pfade von allen anderen Knoten zu v
- *all-pairs*: berechne kürzeste Pfade für alle Knotenpaare

Bis heute ist für das single-pair-Problem kein substantiell besserer Algorithmus bekannt als für das single-source-Problem. Deshalb konzentrieren wir uns im folgenden auf das single-source-Problem. Das single-sink-Problem lässt sich einfach auf das single-source-Problem zurückführen, indem man alle Kanten umkehrt. Auch das all-pairs-Problem lässt sich unter anderem durch das Lösen von n single-source-Problemen lösen.

Sei v also der Startknoten, für den wir das single-source-Problem lösen wollen. Wir nehmen an, dass alle anderen Knoten von v aus erreichbar sind (zu Knoten, die nicht über einen Pfad erreichbar sind, kann es auch keinen kürzesten Pfad geben) und dass es in G keinen Kreis negativer Länge gibt (sonst könnte man durch wiederholtes Herumlaufen um den Kreis einen Pfad beliebig kurz machen; fordert man beim Vorhandensein von Kreisen negativer Länge *einfache* kürzeste Pfade, so ergibt sich ein $\mathcal{NP}$-hartes Problem).

Wenn G keinen Kreis negativer Länge enthält, gibt es auf jeden Fall *einfache* kürzeste Pfade; jeder nicht einfache Pfad enthält nämlich einen Kreis, den man weglassen kann, ohne den Pfad länger zu machen. Die Länge eines kürzesten Pfades vom Startknoten zu einem Knoten w nennt man auch den *Abstand* von w.

Weiterhin bezeichne $\texttt{from}[w]$ für alle Knoten $w \neq v$ den letzten Knoten vor w auf einem kürzesten Pfad von v zu w. Damit ergeben die kürzesten Pfade von v zu allen anderen Knoten zusammen einen *Kürzeste-Pfade-Baum*, dessen Wurzel v ist und in dem jeder Knoten $w \neq v$ ein Kind von $\texttt{from}[w]$ ist.



(a) Eingabegraph (b) Kürzeste-Pfade-Baum

2 Algorithmus von Dijkstra

Der Algorithmus von Dijkstra löst das single-source-Problem in gerichteten Graphen mit *positiven* Kantengewichten. Die Hauptidee dabei ist, dass in diesem Fall die kürzesten Pfade in Reihenfolge aufsteigender Länge berechnet werden können. Der Algorithmus berechnet dabei für jeden Knoten w einen Wert $\texttt{dist}[w]$, der die Länge eines kürzesten Pfades vom Startknoten v zu w angibt, und einen Knoten $\texttt{from}[w]$ (außer für $w = v$), der den letzten Knoten vor w auf dem kürzesten Pfad angibt. (Für die Implementierung ist es auch sinnvoll, sich die Kante $(\texttt{from}[w], w)$ zu merken.) Aus den `from`-Werten lassen sich dann die kürzesten Pfade leicht konstruieren.

Der Algorithmus verwaltet (implizit) eine Teilmenge V' von V mit *erledigten* Knoten, zu denen bereits kürzeste Pfade bestimmt wurden, d.h. für die die `dist`- und `from`-Werte bereits korrekt sind und nicht mehr verändert werden. Zur Initialisierung wird $V' = \{v\}$ und $\texttt{dist}[v] = 0$ gesetzt. Dann wird in jedem Schritt ein zusätzlicher Knoten w erledigt und in V' eingefügt. Der Knoten w ist dabei derjenige unter allen noch nicht erledigten Knoten, der den Wert

$$\min_{u \in V', (u,w) \in E} \{\texttt{dist}[u] + c(u, w)\}$$

minimiert, also am günstigsten (bzgl. Pfadlänge) von V' aus über eine Kante (u, w) erreicht werden kann. Dann wird $\texttt{dist}[w] = \texttt{dist}[u] + c(u, w)$ und $\texttt{from}[w] = u$ gesetzt und Knoten w ist erledigt. Nach $n - 1$ Schritten ist der Algorithmus beendet.

Die Korrektheit des Algorithmus lässt sich leicht durch Induktion zeigen, indem man beweist, dass nach $k < n$ Schritten des Algorithmus V' aus k Knoten mit kleinsten Abständen von v besteht.

Zur effizienten Implementierung des Algorithmus von Dijkstra ist zu überlegen, wie in jedem Schritt der nächste zu erledigende Knoten w bestimmt werden kann. Dazu verwenden wir eine Priority-Queue, in der alle Knoten aus $V \setminus V'$ enthalten sind, die über eine Kante von V' aus erreichbar sind. Die Priorität eines Knoten w in der Priority-Queue soll dabei gleich $\min_{u \in V', (u,w) \in E} \{\texttt{dist}[u] + c(u, w)\}$ sein. Dann können wir durch eine DELETEMIN-Operation den nächsten zu erledigenden Knoten w bestimmen. Zusätzlich merken wir uns für jeden Knoten w in der Priority-Queue denjenigen Knoten u in V', der auf dem bisher kürzesten gefundenen Pfad zu w als letzter Knoten vor w besucht wird. Dieser Knoten liefert nämlich den korrekten Wert für $\texttt{from}[w]$, wenn w das Minimum der Priority-Queue wird.

Am Anfang, wenn $V' = \{v\}$ gilt, initialisieren wir die Priority-Queue, indem wir alle Nachbarn w von v (d.h. Knoten am anderen Ende von Kanten, die aus v herauslaufen) mit Priorität $c(v, w)$ in die Priority-Queue einfügen. Sobald wir einen Knoten w durch eine DELETEMIN-Operation aus der Priority-Queue herausnehmen und neu in V' einfügen, müssen eventuell Nachbarn von w in die Priority-Queue eingefügt werden oder ihre Priorität verringert werden. War ein Nachbar u von w vorher überhaupt nicht in der Priority-Queue, so muss er mit Priorität $\texttt{dist}[w] + c(w, u)$ eingefügt werden; war ein Nachbar u von w mit einer Priorität größer als $\texttt{dist}[w] + c(w, u)$ in der Priority-Queue, so muss die Priorität auf diesen Wert erniedrigt werden (DECREASEPRIORITY-Operation). In beiden Fällen merkt man sich in $\texttt{from}[u]$, dass der vorläufig kürzeste Pfad zu u als letzten Knoten vor u den Knoten w besucht.

Insgesamt ist diese Implementierung des Algorithmus von Dijkstra sehr ähnlich dem Algorithmus von Prim zur Berechnung minimaler Spannbäume. Der Hauptunterschied ist lediglich die Zuweisung von Prioritäten für die Knoten in der Priority-Queue. Auch die Laufzeit ist etwa dieselbe: es werden $n-1$ Schritte gemacht, in denen jeweils eine DELETEMIN-Operation und für alle Nachbarn des aktuellen Knotens evtl. eine INSERT- oder DECREASE-PRIORITY-Operation ausgeführt werden. Bei der Implementierung von Priority-Queues mittels Fibonacci-Heaps ergibt sich wieder eine Worst-Case-Laufzeit von $O(|V| \log |V| + |E|)$.

3 Kürzeste Pfade in allgemeinen Graphen

Es sei ein gerichteter Graph $G = (V, E)$ mit $|V| = n$, $|E| = m$ und Kantengewichten $c : E \to \mathbb{R}$ gegeben. Wir nehmen an, dass ein Startknoten $s \in V$ ausgewählt wurde und dass alle anderen Knoten von s aus erreichbar sind. (Daraus folgt dann auch, dass G mindestens $n-1$ Kanten enthält, also $n = O(m)$ gilt.) Falls $c(e) > 0$ für alle $e \in E$, so löst der Algorithmus von Dijkstra das single-source-Problem, d.h. die Berechnung der kürzesten Pfade von s zu allen anderen Knoten des Graphen, in Zeit $O(m + n \log n)$. Wenn auch negative Kantengewichte auftreten, liefert der Algorithmus von Dijkstra aber in der Regel keine korrekten Ergebnisse mehr (siehe Abbildung 1), und ein modifizierter Ansatz ist nötig. Im folgenden beschreiben wir den Algorithmus von Bellman und Ford.

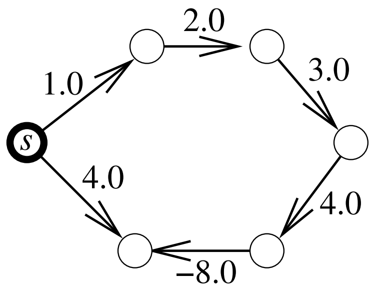


Abbildung 1: Gegenbeispiel für Algorithmus von Dijkstra

Statt die kürzesten Pfade in Reihenfolge aufsteigender Länge zu berechnen, betrachten wir die Pfade in Reihenfolge aufsteigender Kantenanzahl. Der resultierende Algorithmus wird in Phasen vorgehen und spätestens nach der i-ten Phase alle kürzesten Pfade mit $\leq i$ Kanten berechnet haben. Wenn G keinen Kreis negativer Länge enthält, gibt es einfache kürzeste Pfade und der Algorithmus terminiert nach spätestens n Phasen.

Wir speichern für jeden Knoten v wieder zwei Werte $\texttt{dist}[v]$ (den vorläufigen Abstand des Knotens vom Startknoten) und $\texttt{from}[v]$ (den letzten Knoten vor v auf dem vorläufig kürzesten Pfad von s nach v). Am Ende der Berechnung repräsentieren diese Werte die kürzesten Pfade von s zu allen anderen Knoten. Außerdem verwenden wir noch eine Queue Q von Knoten, zu denen wir einen kürzeren Pfad gefunden haben und deren Nachbarn (Knoten am anderen Ende von ausgehenden Kanten) deshalb noch einmal dahingehend geprüft werden müssen, ob sich auch zu ihnen ein kürzerer Pfad findet.

Anfangs setzen wir $\texttt{from}[v] = \texttt{NULL}$ für alle $v \in V$, $\texttt{dist}[s] = 0$ und $\texttt{dist}[v] = \infty$ für alle $v \neq s$. Außerdem fügen wir den Startknoten s in die Queue ein. Dann wiederholen wir die folgenden Schritte:

1. hole den ersten Knoten v aus der Queue
2. für jede ausgehende Kante $e = (v, w)$ von v, prüfe ob $\texttt{dist}[v] + c(v, w) < \texttt{dist}[w]$ (d.h. ob über v gerade ein kürzerer Pfad zu w gefunden wurde); falls ja, so setzte $\texttt{dist}[w] = \texttt{dist}[v] + c(v, w)$ und $\texttt{from}[w] = v$ und füge w in die Queue ein, falls es nicht schon darin enthalten ist

Diese Schritte gruppieren wir folgendermaßen in Phasen: Seien am Ende der vorherigen Phase (bzw. am Anfang des Algorithmus) die Knoten $v_1, \ldots, v_k$ in der Queue enthalten. Dann besteht die nächste Phase des Algorithmus aus den k folgenden Schritten, in denen jeweils einer dieser Knoten bearbeitet wird. Die Phase endet, wenn Knoten v_k fertig bearbeitet ist. Der erste Knoten, der nach v_k in die Queue eingefügt wurde, wird dann als erster in der darauffolgenden Phase bearbeitet. Eine Phase entspricht also der Bearbeitung aller Knoten, die zu Anfang der Phase in der Queue waren.

Die Laufzeit pro Phase ist höchstens $O(n + m)$, da in einer Phase jeder Knoten höchstens einmal bearbeitet wird und für jeden Knoten nur seine ausgehenden Kanten betrachtet werden. Da $n = O(m)$ gilt, kann der Zeitbedarf pro Phase auch als $O(m)$ angegeben werden.

Falls nach n Phasen die Queue noch Knoten enthält, so bricht der Algorithmus die Berechnung ab und meldet, dass G einen Zyklus negativer Länge enthält. Die Gesamtlaufzeit des Algorithmus ist also $O(nm)$.

Satz 1 *Falls G keinen Kreis negativer Länge enthält, berechnet der Algorithmus von Bellman und Ford korrekt die kürzesten Pfade von s zu allen anderen Knoten.*

Beweis: Es ist klar, dass die `dist`-Werte, die der Algorithmus berechnet, nie kleiner als die tatsächlichen Abstände werden können. Daher ist nur zu zeigen, dass am Ende alle `dist`-Werte mit den tatsächlich Abständen übereinstimmen. Dazu beweisen wir per Induktion nach i die folgende Behauptung:

$(*)$ Am Ende von Phase i hat der Algorithmus kürzeste Pfade mindestens zu denjenigen Knoten korrekt berechnet, zu denen es einen kürzesten Pfad mit $\leq i$ Kanten gibt.

Für $i = 1$ ist die Richtigkeit von $(*)$ offensichtlich. Sei $(*)$ nun für $i = 1, 2, \ldots, k - 1$ richtig. Sei w ein Knoten aus G, zu dem es einen kürzesten Pfad mit k Kanten, aber keinen mit $< k$ Kanten gibt. Sei w' der letzte Knoten vor w auf einem solchen Pfad. Dann gibt es einen kürzesten Pfad zu w' mit $k - 1$ Kanten. Also ist für w' am Anfang von Phase k der korrekte Abstand bereits berechnet. Zu dem Zeitpunkt, als der korrekte Abstand berechnet wurde, war w' entweder schon in der Queue oder wurde in sie eingefügt. Auf jeden Fall wurde bzw. wird w' (und damit seine ausgehende Kante (w', w) zu w) nach diesem Zeitpunkt noch einmal bearbeitet, spätestens in Phase k, und damit auch für w der korrekte Abstand und ein entsprechender kürzester Pfad berechnet. □

Falls G keinen negativen Zyklus enthält, so sind spätestens nach Phase $n-1$ alle kürzesten Pfade bestimmt und in Phase n werden keine Knoten mehr in die Queue eingefügt. Falls also nach n Phasen noch Knoten in der Queue sind, so muss G einen Zyklus negativer Länge enthalten.

Historische Bemerkung: Die Idee, mit vorläufigen `dist`-Werten zu arbeiten und bei Betrachtung einer Kante (v, w) mit $\texttt{dist}[v] + c(v, w) < \texttt{dist}[w]$ den Wert $\texttt{dist}[w]$ zu aktualisieren, stammt von L.R. Ford aus den 50'er Jahren. Die Idee, die noch zu bearbeitenden Knoten in einer Queue zu verwalten und für den aktuellen Knoten alle ausgehenden Kanten zu betrachten, hatten kurz darauf unabhängig voneinander R.E. Bellman und E.F. Moore. Der sich ergebende Algorithmus wird meist als Algorithmus von Bellman und Ford zitiert, gelegentlich auch als Algorithmus von Moore und Ford.

4 Erkennung von Kreisen negativer Länge

Der Algorithmus von Bellman und Ford erkennt das Vorhandensein eines negativen Zyklus daran, dass sich nach n Phasen noch Knoten in der Queue befinden. In diesem Fall würde man auch gerne einen solchen Zyklus negativer Länge wirklich ausgeben.

4.1 Erkennung nach n Phasen

Folgender Satz hilft uns, nach n Phasen ohne großen Aufwand einen negativen Zyklus zu bestimmen.

Satz 2 *Für $k = 1, \dots, n$ gilt, dass für einen Knoten w, der nach k Phasen in der Queue ist, die Kette $w, \texttt{from}[w], \texttt{from}[\texttt{from}[w]] = \texttt{from}^{(2)}[w], \dots$ mindestens k-mal zurückverfolgt werden kann, ohne auf einen `NULL`-Wert zu stoßen. Es gilt also $\texttt{from}^{(k)}[w] \neq \texttt{NULL}$.*

Beweis: Alle Knoten, die in einer der n Phasen in die Queue eingefügt wurden, haben einen definierten `from`-Wert ungleich `NULL`. Wir zeigen die Behauptung des Satzes per Induktion nach k. Für $k = 1$ ist sie offensichtlich richtig. Sei w einer der Knoten, die nach k Phasen noch in der Queue sind. Also muss es einen Knoten w' geben, der nach $k - 1$ Phasen in der Queue war und dessen ausgehende Kante (w', w) dazu geführt hat, dass w wieder in die Queue eingefügt wurde. Entweder gilt am Ende von Phase k immer noch $\texttt{from}[w] = w'$, dann wissen wir (Induktionsannahme), dass die `from`-Kette ab w' $(k-1)$-mal zurückverfolgt werden kann, also ab w k-mal. Oder es wurde anschließend ein noch kürzerer Pfad zu w' gefunden, dann gilt $\texttt{from}[w] = w''$ für einen Knoten w'', der nach Phase $k - 1$ in der Queue war. Auch dann folgt, dass die `from`-Kette ab w k-mal zurückverfolgt werden kann. □

Sei nun w ein Knoten, der nach n Phasen in der Queue ist. Da die `from`-Kette von w aus n-mal zurückverfolgt werden kann, aber die $n + 1$ Werte w, $\texttt{from}[w]$, $\texttt{from}^{(2)}[w]$, ..., $\texttt{from}^{(n)}[w]$ nicht alle verschieden sein können, muss diese Kette einen Zyklus enthalten. Man kann sich leicht überlegen, dass dieser Zyklus negative Länge haben muss. Ansonsten hätte der Algorithmus die Kante, die den Kreis geschlossen hat, nämlich nicht eingefügt.

Wenn nach n Phasen noch Knoten in der Queue sind, kann also ein negativer Zyklus einfach dadurch gefunden werden, dass man einen beliebigen Knoten w aus der Queue holt und von ihm aus die `from`-Kette zurückverfolgt, bis ein Knoten w' das zweite Mal auftritt. Die Kanten vom ersten zum zweiten Auftreten von w' ergeben einen Zyklus negativer Länge.

4.2 Frühzeitige Erkennung

Wenn der Eingabegraph einen Zyklus negativer Länge enthält, so tritt dieser Zyklus unter Umständen bereits nach wenigen Phasen in der `from`-Kette eines Knotens auf. In diesem Fall würde man gerne die Berechnung sofort abbrechen und nicht die vollen n Phasen abarbeiten. Zu diesem Zweck kann man jedesmal, wenn man über eine Kante (v, w) einen kürzeren Pfad zu w findet und $\texttt{dist}[w] = \texttt{dist}[v] + c(v, w)$ setzen will, überprüfen, ob w in der `from`-Kette von v auftaucht. Falls ja, so ergeben die Kanten der `from`-Kette von w bis v, gefolgt von der Kante (v, w), einen negativen Zyklus. Da die `from`-Kette aber bis zu n Knoten enthalten kann, dauert dieses Durchsuchen im schlimmsten Fall jedesmal $O(n)$ Zeit und die Worst-Case-Laufzeit dieses modifizierten Algorithmus von Bellman und Ford ergibt sich zu $O(n^2m)$.

Eine weitere Methode, negative Zyklen frühzeitig zu erkennen, überprüft nicht, ob der Knoten w in der `from`-Kette von v enthalten ist, sondern ob v unter den Nachfolgern von w im bisherigen Kürzeste-Pfade-Baum vorkommt. Diese Methode erfordert zusätzliche Datenstrukturen, um von einem Knoten aus effizient alle Nachfolger im Kürzeste-Pfade-Baum durchsuchen zu können, erhält aber bei geschickter Implementierung die Worst-Case-Laufzeit $O(nm)$.

Literatur

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1st edition, 1990.