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 $u \in V$ zu $v \in V$ ist eine Folge (x_0, x_1), (x_1, x_2), (x_2, x_3), $\ldots$, (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:

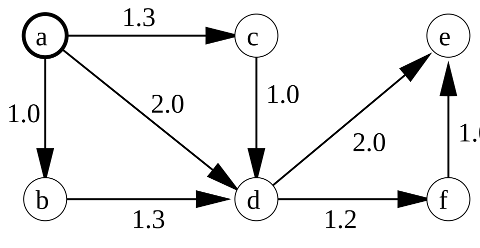
- *single-pair*: berechne für gegebene u und v einen kürzesten Pfad von u nach v
- *single-source*: berechne für einen gegebenen Startknoten v die kürzesten Pfade zu allen anderen Knoten
- *single-sink*: berechne für einen gegebenen Zielknoten v die kürzesten Pfade von allen anderen Knoten zu v
- *all-pairs*: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

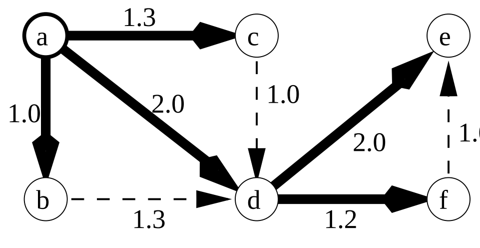
Sei v also der Startknoten, für den wir das single-source-Problem lösen wollen. Wir nehmen an, daß alle anderen Knoten von v aus erreichbar sind (zu Knoten, die nicht über einen Pfad erreichbar sind, kann es auch keinen kürzesten Pfad geben) und daß es in G keinen Kreis negativer Länge gibt (sonst könnte 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 DecreasePriority-Operation ausgeführt werden. Bei der Implementierung von Priority-Queues mittels Fibonacci-Heaps ergibt sich wieder eine Worst-Case-Laufzeit von $O(|V|\log|V|+|E|)$.

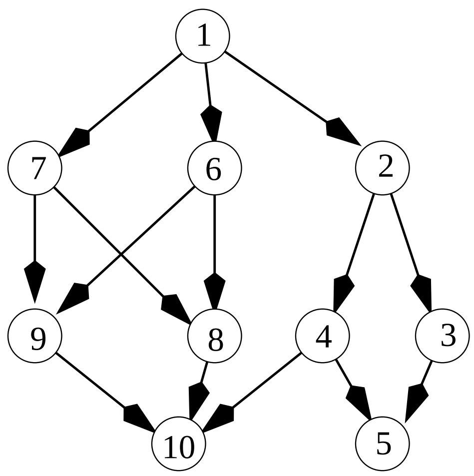
3 Pfade in azyklischen Graphen

Sei wieder ein gerichteter Graph mit Kantengewichten gegeben, der diesmal aber keine Kreise enthält, also azyklisch ist. Für solche Graphen lassen sich kürzeste Pfade einfacher berechnen als in beliebigen gerichteten Graphen. Wir wollen das single-source-Problem für diesen Spezialfall effizient lösen.

Wir nehmen an, daß der Startknoten v der einzige Knoten in G mit Eingangsgrad 0 (d.h. ohne eingehende Kante) ist und daß alle anderen Knoten von v aus erreichbar sind. (Sonst könnten wir einfach den Teilgraph aller von v aus erreichbaren Knoten betrachten.)

Eine hilfreiche Tatsache über azyklische gerichtete Graphen G ist, daß man eine Numerierung bzw. Reihenfolge $v_1, v_2, \ldots, v_n$ der Knoten von G finden kann, so daß alle Kanten des Graphen von einem Knoten v_i zu einem Knoten v_j mit $j > i$ laufen. Eine solche Reihenfolge nennt man auch *topologische Sortierung*. Sie läßt sich mit verschiedenen Methoden effizient in linearer Zeit bestimmen, etwa mit einer modifizierten Tiefensuche, bei der die Nummern in absteigender Reihenfolge am Ende der *visit*-Funktion vergeben werden. In einer topologischen Sortierung unter den obigen Annahmen muß der Startknoten v die Nummer 1 erhalten, da er der einzige Knoten ohne eingehende Kanten ist.

Die folgende Abbildung zeigt ein Beispiel eines azyklischen Graphen, bei dem die Nummern in den Knoten eine gültige topologische Sortierung darstellen:



Um die kürzesten Pfade in G zu berechnen, betrachten wir die Knoten von G in der Reihenfolge der topologischen Sortierung. Sei der aktuell bearbeitete Knoten v_i und seien die Abstände und kürzesten Pfade für $v_1, \ldots, v_{i-1}$ bereits berechnet. Da alle in v_i hineinlaufenden Kanten an einem der bereits erledigten Knoten beginnen, erfüllt der Abstand des aktuellen Knotens v_i von v die folgende Gleichung:

$$\texttt{dist}[v_i] = \min_{j=1,\ldots,i-1,(v_j,v_i)\in E} \{\texttt{dist}[v_j] + c(v_j, v_i)\}$$

Wenn der aktuelle Knoten v_i also d eingehende Kanten hat, so kann unter Zuhilfenahme von d bereits berechneten Werten der Wert $\texttt{dist}[v_i]$ bestimmt werden. Analog lassen sich nebenbei auch die `from`-Werte berechnen.

Mit diesen Hinweisen sollte es nicht schwer sein, ein Programm zu implementieren, das in Zeit $O(|V| + |E|)$ eine topologische Sortierung bestimmt und anschließend in Zeit $O(|V| + |E|)$ die kürzesten Pfade von v zu allen anderen Knoten.

Ebenso ist nicht schwer zu sehen, daß auf dieselbe Art und Weise auch die *längsten* Pfade von v zu allen anderen Knoten berechnet werden können, wenn man in der obigen Formel das Maximum statt des Minimums bildet. Die Berechnung längster Pfade kann also in azyklischen Graphen in linearer Zeit erfolgen, obwohl dieses Problem in beliebigen Graphen $\mathcal{NP}$-hart ist.

Literatur: Volker Turau. Algorithmische Graphentheorie. Addison-Wesley, Bonn, 1996. S. 250–254.