Dijkstra-Algorithmus

Letzte Woche haben wir zwei grundlegende Techniken kennengelernt, um Graphen zu traversieren (zu durchlaufen). In dieser Serie wollen wir einen praktisch anwendbaren Algorithmus kennenlernen, der von einem Knoten aus, die kürzesten Wege zu allen anderen Knoten ermittelt.

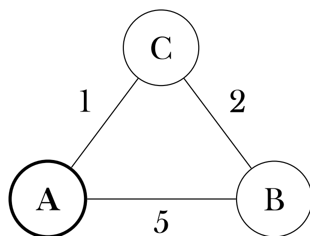
Gewichtete Graphen

Wir kennen normale Graphen (V, E) mit Knotenmenge V und Kantenmenge E. Wollen wir durch einen Graphen eine reale Situation modellieren, so sind die Wege zwischen zwei Knoten nicht immer genauso lang wie alle anderen Kanten. Aus diesem Grund fügen wir eine Funktion $d : E \to \mathbb{R}$ hinzu, welche jeder Kante ein „Gewicht" hinzufügt. Ist $e \in E$ eine Kante, so ist $d(e)$ intuitiv die „Länge" der Kante oder auch die Zeit, die gebraucht wird, um diese Kante zu durchlaufen. Das Tripel (V, E, d) nennen wir *gewichteter Graph*.

Die Idee

Gegeben sei ein gewichteter Graph (V, E, d) bei dem alle Kantengewichte nichtnegativ sind, also $d(e) \geq 0$ für alle $e \in E$. Für einen Knoten $v \in V$ kennen wir sofort die kürzesten Strecken zu v selbst.

Aufgabe 1 Betrachte den folgenden Graphen.



Knoten	Länge des kürzesten Weges	Kürzester Weg
A	0	A
B		
C		

Ausgehend vom Knoten A, fülle den Rest der Tabelle aus.

Ziel ist es nun zu demjenigen Knoten zu wechseln, dessen Distanz zum aktuell erforschten Teil des Graphen (in diesem Fall erstmal nur der Startknoten) am kleinsten ist. Dieser Knoten wird dann als besucht markiert. Für dessen Nachbarn aktualisieren wir die bisher bekannten kürzesten Wege. Falls ein Nachbar neu ist, dann ist das einfach die Summe aus dem Kantengewicht zu diesem hin, vom neusten Knoten aus plus die Distanz vom neusten Knoten zum Startknoten, die wir gespeichert haben. Falls ein Nachbar bereits eine eingetragene Distanz besitzt, dann kann es nun passieren, dass wir einen neuen kürzeren Weg entdeckt haben. Falls das der Fall ist, überschreiben wir die Informationen, falls nicht, dann nicht. Und das machen wir eben, bis wir alle Knoten im Graphen besucht haben. Beziehungsweise, bis wir keine weiteren Knoten besuchen können. Falls es noch weitere Zusammenhangskomponenten gibt, so erhalten diese automatisch die Distanz ∞.

Der Algorithmus

Sei $v^* \in V$ der Startknoten. Wir geben jedem Knoten $v \in V$ zwei Variablen, Distanz d_v und Weg w_v. Am Ende des Algorithmus soll d_v die Gesamtlänge des kürzesten Weges von v zu v^* angeben und w_v diesen kürzesten Weg.

Initial setzen wir $d_{v^*} = 0$ und $d_v = \infty$ für alle $v \neq v^*$. Durchlaufe nun alle Nachbarn u des aktuellen Pivotknoten $p = v^*$ und setze für diese $d_u = d(\{p, u\})$ und $w_u = w_p u$. Wähle nun einen der Nachbarn u mit d_u minimal und setze $p = u$. Der Rest ist Wiederholung, durchlaufe alle Nachbarn von p und aktualisiere gegebenenfalls die Distanzen und Wege, falls ein kürzerer Weg gefunden wurde. Der Algorithmus in Pseudocode sieht wie folgt aus.

Gegeben : Gerichteter Graph (V, E, d) und ein Startknoten $v^* \in V$
Initialisiere $Q = \{v^*\}$.
Setze $d_{v^*} = 0$ und $w_{v^*} =$ _ (leerer Weg).
for $v \in V \setminus \{v^*\}$ **do**
| Setze $d_v = \infty$ und $w_v =$ _.
end
while $Q \neq \emptyset$ **do**
| Wähle $p \in Q$ mit d_p minimal.
| **foreach** $u \in V$ *mit* $\{p, u\} \in E$ **do**
| | **if** u *nicht besucht und* $u \notin Q$ **then**
| | | Füge u der Menge Q hinzu.
| | **end**
| | **if** $d_p + d(\{p, u\}) < d_u$ **then**
| | | Überschreibe $d_u = d_p + d(\{p, u\})$ und $w_u = w_p u$ (Wege verknüpfen).
| | **end**
| **end**
end

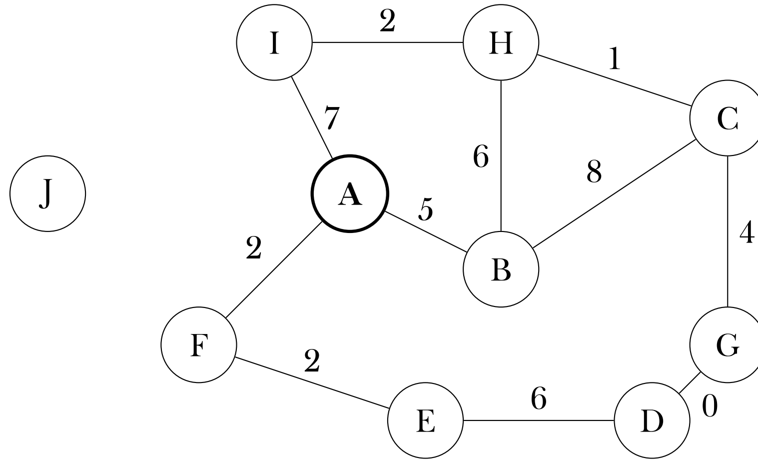
Algorithmus 1 : Dijkstra-Algorithmus

Einschub zur Implementierung

Um den Algorithmus effizient zu implementieren, muss man sich überlegen, wie man bei jedem Durchlauf das Minimum findet. Es ist offenbar ineffizient die Menge Q sortiert zu halten, aber geht es wirklich besser? Das Minimum dafür jedes Mal zu suchen ist auch keine Lösung. Tatsächlich gibt es eine Datenstruktur, bei der das vorderste Element immer das Minimum ist, und die diese Eigenschaft effizient aufrecht erhalten kann, wenn man Element hinzufügt oder wegnimmt. Sie nennt sich *Priority-Queue* und kann über sogenannte *Heaps* implementiert werden. Für mehr Informationen fragen Sie Ihren Informatiker des Vertrauens.

Aufgabe 2 Begründe die Korrektheit des Dijkstra-Algorithmus.

Aufgabe 3 Führe den Dijkstra-Algorithmus für den gewichteten Graphen auf der nächsten Seite an. Der Startknoten ist A.



Knoten v	besucht?	Distanz d_v	Weg w_v
A	○		
B	○		
C	○		
D	○		
E	○		
F	○		
G	○		
H	○		
I	○		
J	○		

Aufgabe 4 Warum fordern wir für den Dijkstra-Algorithmus, dass alle Kantengewichte nichtnegativ sind? Gebe Beispiel an, um deine Begründung zu unterstützen.