Fakultät IV - Elektrotechnik und Informatik
Algorithmen und Datenstrukturen SoSe 2016
Brock / Deimel / Foucard / Mabrouk / Sieverling

Aufgabenblatt 6

letzte Aktualisierung: 06. June, 11:33 Uhr

Ausgabe: 05.06.2016
Abgabe: 15.06.2016 23:59

Thema: *Kürzeste Wege mit positiven Kantengewichten (Dijkstra) und negativen Kantengewichten (Bellmann-Ford)*

Abgabe

Die folgenden Datei muss für eine erfolgreiche Abgabe im svn Ordner `Tutorien/txx/Studierende/deinname@TU-BERLIN.DE/Abgaben/` eingecheckt sein:

Geforderte Datei:	
`Blatt06/src/DiGraph.java`	Aufgaben 2.2-2.3, 3.4-3.5

Wichtige Ankündigungen

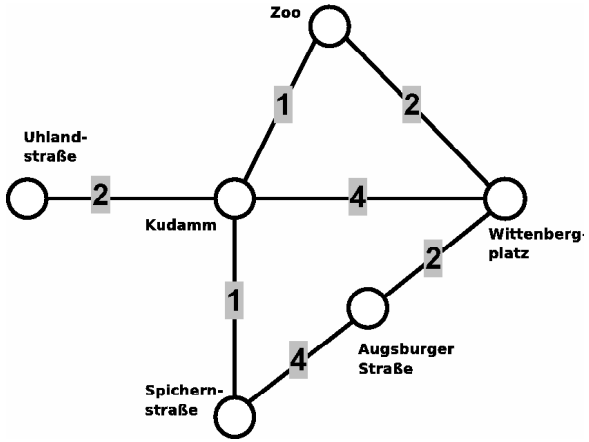
- Unit tests dürfen geteilt werden. Lösungen dürfen auf keinen Fall geteilt werden! Wir testen auf Plagiate.

1. Aufgabe: Kürzeste Wege mit Dijkstra

Mit dem 1959 von Dijkstra entwickelten Algorithmus lassen sich in einem Graphen die kürzesten Wege von einem bestimmten Startknoten zu allen anderen Knoten (1:n) effizient bestimmen. Im Folgenden soll der *Dijkstra*-Algorithmus implementiert werden. https://de.wikipedia.org/wiki/Dijkstra-Algorithmus

1.1. (Tut) Kürzeste Wege (Dijkstra) - Handsimulation

Auf dem letzten Aufgabenblatt haben wir diesen Graph kennengelernt:



Ein Navigationssystem soll die optimalen Wege vom Startpunkt Augsburger Straße zu allen anderen Punkten finden. Zeigt durch Handsimulation, wie der Dijkstra-Algorithmus dabei vorgeht.
Die linke Tabelle zeigt wie zuvor den aktuellen Knoten und die aktuelle Belegung der Queue, wobei die bis dahin kürzeste Distanz zum Startknoten mit notiert wird. Die rechte Tabelle beschreibt den Zustand der Knoten in jedem Schritt, fertig abgearbeitete Knoten (schwarz gefärbte) werden unterstrichen.

Schritt	akt. Knoten	Priority-Queue
0		Au^0
1		
2		
3		
4		
5		
6		

	Knoten					
Schritt	Au	Ku	Sp	Uh	Wi	Zoo
0	0	∞	∞	∞	∞	∞
1						
2						
3						
4						
5						
6						

1.2. Implementierung Kürzeste Wege mit Dijkstra (40 Punkte)

Implementiert in der Klasse `DiGraph` die Methode `populateDijkstraFrom(Node startNode)`, die vom gegebenen Startknoten die kürzeste Distanz zu jedem anderen Knoten berechnet.

Hinweis: Die Klasse `Node` wurde im Vergleich zum letzten Übungsblatt dahingehend angepasst, dass sie nun die Attribute `distance` und `predecessor` besitzt. Außerdem implementiert sie das Interface `Comparable`, mit der Knoten anhand ihrer Distanz verglichen werden können. In diesem Algorithmus muss eine so genannte *Priority Queue* genutzt werden. Java stellt eine PriorityQueue Klasse zur Verfügung, die ihr auch nutzen dürft.

Ihr findet in der Klasse `DijkstraTest.java` Testaufrufe, die eure Implementierung visualisieren. Um die einzelnen Schritte des Algorithmus sichtbar zu machen, könnt ihr die Methode `private void stopExecutionUntilSignal()` nutzen. Diese Methode sorgt dafür, dass das Programm bis zu einem Tastendruck nicht weiter läuft. Wenn ihr Knoten farblich markieren wollt, so könnt ihr das Feld `status` setzen (z.B.

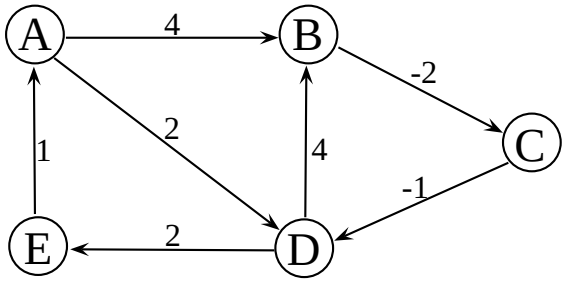
```
myNode.status = Node.GRAY;)
```

1.3. Den kürzesten Weg auslesen (10 Punkte)

Um den kürzesten Weg zwischen zwei Knoten auszugeben, könnt ihr jetzt auf die vorher geschriebene Methode zurückgreifen. Implementiert die Methode `getShortestPathDijkstra(Node startNode, Node targetNode)`, die zuerst `populateDijkstraFrom` aufruft und anschließend den kürzesten Pfad von Start- zu Endknoten ausliest und zurück gibt.

2. Aufgabe: Kürzeste Wege mit Bellman-Ford

Ein Student aus Aachen (A) möchte seine Eltern in Erfurt (E) besuchen. Es ist ihm wichtig, dass er für die Autofahrt so wenig wie möglich ausgibt. Wie alle Studenten kennt auch er Mitfahrgelegenheiten. So findet er im Internet zwei Leute, die von Berlin (B) nach Cottbus (C) und von Cottbus nach Darmstadt (D) mitfahren möchten. Die Benzinpreise abgezogen, würde er dabei sogar noch 2€ (B $\rightarrow$ C) bzw. 1€ (C $\rightarrow$ D) dazu verdienen. Findet für ihn die Fahrt mit den geringsten Kosten heraus.



2.1. (Tut) Wendet *Dijkstra* ausgehend von Knoten A in einer kurzen Handsimulation an. Wo und warum schlägt Dijkstra hier fehl?

2.2. (Tut) Kürzeste Wege (Bellman-Ford) - Handsimulation Wie löst *Bellmann-Ford* das Problem? Führt den *Bellmann-Ford*-Algorithmus in einer schrittweisen Handsimulation durch und füllt dabei die untenstehende Tabelle aus.

	Knoten				
Schritt	A	B	C	D	E
(Ausgangssituation)	0	∞	∞	∞	∞

2.3. (Tut) Bellman-Ford - Grenzen und Aufwand Verändert ein Kantengewicht so, dass *Bellman-Ford* nicht mehr angewendet werden kann. Gebt die Kante und deren neues Gewicht an. Warum schlägt *Bellman-Ford* dann fehl?

Begründet zusätzlich kurz, welcher der beiden vorgestellten Algorithmen den besseren Aufwand hat, um von einem Startknoten die Distanz zu allen anderen Knoten zu errechnen.

2.4. Implementierung Kürzeste Wege mit Bellman-Ford (40 Punkte)

Implementiert in der Klasse `DiGraph` die Methode `populateBellmanFordFrom(Node startNode)`, die den ersten Teil des Bellman-Ford-Algorithmus ausführt, analog zu `populateDijkstraFrom(Node startNode)`. Dabei soll vom gegebenen Startknoten die kürzeste Distanz zu jeden anderen Knoten berechnet werden.

2.5. Den kürzesten Weg auslesen (10 Punkte)

Um den kürzesten Weg zwischen zwei Knoten auszugeben, könnt ihr jetzt auf die vorher geschriebene Methode zurückgreifen. Implementiert dazu die Methode `getShortestPathBellmanFord(Node startNode, Node targetNode)`, die zuerst die Distanzen vom Startknoten zu jeden anderen Knoten setzt, anschließend den kürzesten Pfad von Start- zu Endknoten ausliest und zurück gibt.

3. Aufgabe: Topologische Sortierung

3.1. (Tut) Simulation an einem Beispiel Bei großen Softwareprojekten müssen oft viele Module kompiliert werden. Dabei sind einzelne Module voneinander abhängig und müssen somit in einer bestimmten Reihenfolge kompiliert werden. Diese Abhängigkeiten können in einem Graph dargestellt werden, wobei die Knoten die Module darstellen und die Abhängigkeiten als Kanten veranschaulicht werden.

Erstellt einen Graphen für folgende Module und Abhängigkeiten und findet dann eine geeignete Reihenfolge heraus, damit das Projekt problemlos kompilieren kann.

Module: 1; 2; 3; 4; 5

Abhängigkeiten: (2 braucht 1, 5); (3 braucht 5); (4 braucht 1, 2, 3)