

## Programmieren, Algorithmen, Datenstrukturen

### 5. Programmieraufgabe

Abnahme bis spätestens 22.11.2013

#### a) Vorbereitung

Ladet euch die Testdatei `PA05_example.zip` aus dem Lernraum herunter. Ihr findet dort eine Klasse `Example.java`, die allerdings erst kompiliert werden kann, wenn die anderen Aufgabenteile testbereit sind. Zusammen mit der Beispielausgabe könnt ihr die Ausgabe des fertigen Programms dann überprüfen.

#### b) Die Klasse Graph

Schreibt eine Klasse `Graph`, die einen gerichteten Graphen  $G = (V, E)$  mit Knotenmenge  $V = \{0, \dots, n-1\}$  und Kantenmenge  $E \subseteq \{(v, w) : v, w \in V, v \neq w\}$  verwalten kann. (Die Definition von  $E$  impliziert, dass es immer höchstens eine Kante von  $v$  nach  $w$  und keine "Schleifen"  $(v, v)$  gibt.) Zu dem Graphen gehört auch eine Gewichtungsfunktion  $d : E \rightarrow \mathbb{R}$ , die ihr (wie in der Vorlesung) als  $n \times n$ -Bewertungsmatrix speichern sollt. Verwendet den Typ `double` dafür. Nicht vorhandene Kanten  $(v, w)$  mit  $v \neq w$  könnt ihr mit der Konstante `Double.POSITIVE_INFINITY` darstellen, die sich zudem beim Vergleichen, Addieren und Minimum Bilden wie  $+\infty$  verhält. Der Konstruktor nimmt als Parameter  $n$  entgegen und erzeugt einen Graphen mit  $n$  Knoten und keinen Kanten.

Das Array (und damit der Graph) soll nicht von außerhalb der Klasse direkt veränderbar sein. Schreibt stattdessen die folgenden öffentlichen Instanzmethoden und gebt ihnen sinnvolle Parameter und Rückgabetypen: `containsEdge`, `addEdge`, `removeEdge`, `getWeight`, `setWeight`. Orientiert euch dabei an

```
public void addEdge (int v, int w, double weight),
```

welche für das Ausführen der Beispiele vorhanden sein muss. Fehler sollen mit einer sinnvollen `IllegalArgumentException` abgefangen werden, z.B. wenn ein Knoten einen ungültigen Index hat, das Gewicht für eine Kante (neu) gesetzt wird, die nicht im Graphen vorhanden ist, oder eine Kante hinzugefügt wird, die es schon gibt oder eine Schleife wäre. Ferner sollte auch sichergestellt werden, dass dem Graphen keine Kanten mit Gewicht `Double.POSITIVE_INFINITY` oder `Double.NEGATIVE_INFINITY` hinzugefügt werden.

Außerdem werden Methoden `getNumberOfNodes` und `getNumberOfEdges` benötigt. Statt bei jedem Aufruf die Anzahl der Kanten in der Matrix abzuzählen, sollt ihr jede Veränderung der Kantenzahl mitzählen.

Schließlich benötigt ihr noch eine Methode `toString`, welche die Parameter des Graphen und die Matrix ausgibt.

c) **Die Klasse BellmanFord**

Diese Klasse nimmt im Konstruktor einen Graphen  $G = (V, E)$  entgegen und berechnet dafür mit dem Bellman-Ford-Algorithmus die Matrix  $U^{(n)}$  und die zugehörige **tree**-Matrix. Hier bezeichnet, wie üblich,  $n$  die Anzahl der Knoten des Graphen  $G$ .

Implementiert die folgenden drei Instanzmethoden. Sie sollen eine Exception werfen, falls der Graph negative Zyklen enthält.

- `getDistance(int v, int w)` und `getShortestPath(int v, int w)`: geben die Länge eines kürzesten Weges von  $v$  nach  $w$  bzw. die Knoten entlang eines kürzesten Weges von  $v$  nach  $w$  (inklusive  $v$  und  $w$ ) zurück. Falls es keinen Weg von  $v$  nach  $w$  gibt, geben die Methoden  $\infty$  bzw. `null` zurück.
- `public Graph getShortestPathTree(int v)`: konstruiert aus der **tree**-Matrix einen Graphen, der genau die Kanten enthält, die für die kürzesten Wege von  $v$  zu allen anderen Knoten benötigt werden. Die Gewichte dieser Kanten sollen denen aus dem ursprünglichen Graphen entsprechen.

Der Graph, der für die **BellmanFord**-Klasse benötigt wird, kann zwischen zwei Methodenaufrufen der Klasse **BellmanFord** verändert werden. Die Klassen **Graph** und **BellmanFord** sollen dabei so miteinander interagieren, dass die Methoden der Klasse **BellmanFord** auch nach Manipulation/Veränderung des Graphen, die für den veränderten Graphen korrekten Ergebnisse liefern (Hinweis: Arbeiten mit Referenzen anstatt von neuen Objekten). Aus Effizienzgründen sollte der Bellman-Ford-Algorithmus aber nur dann erneut ausgeführt werden, wenn der Graph zwischen zwei Methodenaufrufen auch tatsächlich verändert wurde (z.B. könnt ihr euch die Anzahl der Änderungen am Graphen in einem Datenfeld der Klasse **Graph** abspeichern, dieses ebenfalls als Datenfeld der Klasse **BellmanFord** einrichten und entsprechend vergleichen).

d) **Die Main-Methode**

Reaktiviert die gesamte **Example.java**. Schreibt zusätzlich eine eigene Main-Methode (in einer anderen Klasse), die die weiteren Fähigkeiten eures Codes (Löschen von Kanten, Exceptions, etc) an kleinen Beispielen demonstriert.

Hinweise:

- Vergleicht eure Ergebnisse für die Beispielgraphen mit denen aus der Datei **Programmausgabe.txt**.
- Die Länge eines kürzesten Weges ist zwar eindeutig (wenn es keine negativen Kreise gibt), kürzeste Wege sind dies im Allgemeinen allerdings nicht. Überprüft Abweichungen also sorgfältig!

**Viel Spaß und Erfolg!**