technische universität dortmund

Fakultät für Informatik
Lehrstühle 2, 11, 13

Prof. Dr. C. Sohler
I. Kamehkhosh, N. Kriege, D. Kurz, M. Ludewig

Sommersemester 2017
11. Juli

DAP2 Praktikum – Blatt 13

Abgabe: 17.–21. Juli

Wichtig: Der Quellcode ist natürlich mit sinnvollen Kommentaren zu versehen. Überlegen Sie außerdem, in welchen Bereichen Invarianten gelten müssen, und überprüfen Sie diese ggf. an sinnvollen Stellen mit *Assertions* (siehe Hinweis auf Blatt 2).

Kurzaufgabe 13.1: Bellman-Ford-Algorithmus (4 Punkte)

Implementieren Sie den aus der Vorlesung bekannten Bellman-Ford-Algorithmus für das *Single-Source-Shortest-Path*-Problem (SSSP). Gehen Sie dabei wie folgt vor:

- Laden Sie die vorgegebenen Klassen `Graph`, `Node`, `Edge`, `Pair` und `GraphTest` sowie die Beispielgraphen herunter und kopieren Sie diese in ihr Projektverzeichnis.
  Achtung: Verwenden Sie nicht Ihre Klassen von Blatt 12, da diese hier um Methoden erweitert wurden. Dies gilt ebenfalls für die `.graph`-Dateien.
- Lesen Sie die Hinweise zur Klasse `Graph` und machen Sie sich mit den Schnittstellen für `Graph`, `Node` und `Edge` vertraut.
- Die Klasse `GraphTest` enthält bereits ein ausführbares Hauptprogramm. Wenn es mit zwei Parametern aufgerufen wird, werden diese folgendermaßen interpretiert: Das erste Argument gibt den Pfad zu einer `.graph`-Datei an und das zweite Argument den Index eines `source`-Knotens. Der Aufruf lautet also `java GraphTest FILE source`.
- Aus der angegebenen Datei wird bereits ein Graph geladen und anschließend eine Methode `sssp(Graph g, int source)` aufgerufen. Diese ist im Rahmen dieser Aufgabe von Ihnen mit Hilfe des Algorithmus von Bellman-Ford zu implementieren. Sie soll ein Array mit Weglängen zurückgeben, so dass das i-te Element die Länge eines kürzesten Weges von dem Knoten `source` zu dem Knoten i angibt.
- Auf der Grundlage des zurückgegebenen Arrays gibt das Hauptprogramm einen Knoten mit maximaler Entfernung sowie die Länge eines zugehörigen kürzesten Weges aus. Bei weniger als 20 Knoten im Graphen wird außerdem die Längen der kürzesten Wege zu den jeweiligen Knoten ausgegeben. Prüfen Sie anhand der Ausgabe, ob Ihre Implementierung plausible Ergebnisse liefert.

Kurzaufgabe 13.2: Floyd-Warshall-Algorithmus (4 Punkte)

Implementieren Sie den aus der Vorlesung bekannten Floyd-Warshall-Algorithmus für das *All-Pairs-Shortest-Path*-Problem (APSP).

- Wird das Hauptprogramm der Klasse `GraphTest` ohne `source`-Parameter aufgerufen, so wird die Methode `apsp(Graph g)` verwendet, die das APSP-Problem lösen soll. Diese ist im Rahmen der Aufgabe von Ihnen mit Hilfe des Algorithmus von Floyd-Warshall zu implementieren. Sie soll die Länge der kürzesten Wege zwischen allen Paaren von Knoten berechnen und eine Distanzmatrix zurückgeben, in der das Element (i, j) die Länge eines kürzesten Weges von dem Knoten i zu dem Knoten j angibt.
- Auf der Grundlage der zurückgegebenen Distanzmatrix gibt das Hauptprogramm ein Paar von Knoten mit maximaler Entfernung sowie die Länge eines zugehörigen kürzesten Weges aus. Bei weniger als 10 Knoten wird die vollständige Distanzmatrix ausgegeben. Prüfen Sie anhand der Ausgabe, ob Ihre Implementierung plausible Ergebnisse liefert.
- Die Klasse `GraphTest` enthält bereits die vorgegebene Methode `apspBellmanFord(Graph g)`, welche das APSP-Problem durch wiederholte Aufrufe der `sssp()`-Methode löst. Ergänzen Sie das Hauptprogramm um eine Laufzeitmessung und vergleichen Sie die Laufzeiten der Methoden `apsp()` und `apspBellmanFord()`. Dokumentieren Sie Ihre Resultate.

Beachten Sie die Hinweise und Tipps auf den folgenden Seiten.

Hinweise und Tipps

Die Klasse `Graph`

Die Klasse `Graph` stellt eine Datenstruktur für einen Graphen in Java zur Verfügung. Man kann sowohl gerichtete als auch ungerichtete Graphen erzeugen. Die wichtigsten Methoden sind:

- `addNode` zum Einfügen eines Knotens in den Graphen
- `addEdge` zum Einfügen einer Kante in den Graphen
- `getNodes` gibt eine Liste der Knoten zurück
- `getEdges` gibt eine Liste der Kanten zurück
- `getCost` zum Abfragen der Kosten einer Kante zwischen zwei Knoten
- `(static) fromFile` liest einen Graphen aus einer Datei ein

Der Graph benutzt zudem als Hilfsklassen `Node` und `Edge`, die jeweils einen Knoten bzw. eine Kante verwalten. Neben den Get-Methoden für die Knoten-ID, den Start- und Endknoten und die Kosten einer Kante gibt es noch folgende wichtige Methoden:

- `Node.addEdge` zum Einfügen einer neuen Kante
- `Node.getIncidenceList` gibt eine Liste der inzidenten Kanten zurück

Auf Knoten und Kanten gibt es eine natürliche Ordnung (`compareTo`), Knoten werden anhand ihrer eindeutigen ID und Kanten anhand ihrer Kosten verglichen. Im Gegensatz dazu liefert die `equals`-Methode von Kanten nicht true, wenn die Gewichte gleich sind, sondern wenn der Start- und Endknoten jeweils gleich ist. Knoten können außerdem mit einem `int` verglichen werden, der als ID interpretiert wird. Kanten und Knoten sind auch vergleichbar, dabei ergibt `equals` true, wenn der Knoten gleich dem Endknoten der Kante ist.

Eigene Graphen

Die vorgegebenen `.graph`-Dateien sind so aufgebaut, dass in der ersten Zeile die Anzahl der Knoten als `int`-Wert steht und `true` bzw. `false`, wenn der Graph gerichtet bzw. ungerichtet ist. Alle nachfolgenden Zeilen werden als Kanten (Startknoten, Endknoten, Kosten) interpretiert. Dabei werden Knoten durch ihre ID identifiziert und es wird davon ausgegangen, dass ein Graph mit n Knoten die Knoten IDs $\{0, \ldots, n-1\}$ verwendet. Es ist auch möglich, in Java erzeugte Graphen mittels `toFile(String filename)` wieder als Dateien zu speichern. Außerdem können Graphen direkt in einem Texteditor "erzeugt" und mit `fromFile(String filename)` als Java-Graph geladen werden. Achten Sie dabei insbesondere darauf, dass die Kantengewichte immer existieren und echt größer 0 sein müssen.