technische universität dortmund

Fakultät für Informatik
Lehrstuhl 11

Erik Thordsen
Lena Krieg
Sebastian Konietzny

Sommersemester 2023
07. Juni

DAP2 Praktikum – Blatt 9

Abgabe: 12. Juni

Studienleistung

- Zum Bestehen des Praktikums muss jeder Teilnehmer die folgenden Leistungen erbringen:
  - Es müssen mindestens 50 Prozent der Punkte in den Kurzaufgaben erreicht werden.
  - Es müssen mindestens 50 Prozent der Punkte in den Langaufgaben erreicht werden.
- Im Krankheitsfall kann ein Testat bei Vorlage eines Attests in der folgenden Woche nachgeholt werden.
- Wenn ein Praktikumstermin auf einen Feiertag fällt, müssen Sie sich an einem beliebigen anderen Praktikumstermin in der gleichen Woche testieren lassen.
- Ansonsten kann die Testierung **nur in der zugeteilten Gruppe** garantiert werden.
- Bitte bereiten Sie den Tester sowie in den Aufgaben angegebene Beispieltests vor, bevor Sie sich testieren lassen!

Wichtige Information (im Moodle verfügbar)

- Beachten Sie die Erklärung des **Ablaufs (Blatt A)**.
- Beachten Sie die **Regeln und Hinweise (Blatt R)** in der aktuellsten Version!
- Beachten Sie die **Hilfestellungen (Blatt H)** in der aktuellsten Version!

Kommandozeile Tester

Sie finden im Moodle eine Datei `Test.jar`. Fuer das Testen Ihrer Lösung laden Sie diese herunter und geben den unten stehenden Kommandozeilen Befehl ein. **Es ist möglich, dass die Tests verzögert zur Verfügung gestellt oder vervollständigt werden!**

```
java -jar <path-to-moodle-jar>/Test.jar -s <path-to-solution> 9 -e
```

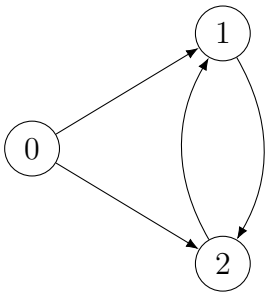
Kurzaufgabe 9.1: Pfadsuche auf Graphen (16 Punkte)

In dieser Aufgabe sollen Sie Breiten- und Tiefensuche implementieren. Dafür benötigen Sie zunächst eine Repräsentation von Graphen. Implementieren Sie eine Klasse `DirectedGraph`, die gerichtete und ungewichtete Graphen modelliert. Erstellen Sie dafür zuerst die Hilfsklassen `Edge` und `Node` mit den angegebenen Feldern und einem entsprechenden Konstruktor. Die `id`'s der `Node`'s soll dabei dem Index der `Node` im Array `DirectedGraph.nodes` entsprechen. Anschließend implementieren Sie die folgenden Methoden entsprechend Ihrer Beschreibung:

- `Node.addEdge` soll eine neue Kante als ausgehend abspeichern.
- `Node.hasEdge` soll `true` zurückgeben, falls eine ausgehende Kante zur `Node to` existiert und sonst `false`.
- `DirectedGraph:` der Konstruktor von `DirectedGraph` soll einen Graph mit n Knoten anlegen. Diese Knoten werden bereits im Konstruktor erzeugt un in dem Array `nodes` gespeichert.
- `DirectedGraph.getNode` soll die `Node` mit der angegebenen `id` zurück geben, falls diese existiert und sonst `null`.
- `DirectedGraph.addEdge` soll eine Kante zwischen den beiden angegebenen Knoten einfügen. Falls die angegebene Kante bereits existiert oder die Knoten zu den angegebenen `id`'s nicht existieren, soll eine `IllegalArgumentException` geworfen werden.
- `DirectedGraph.hasEdge` soll `true` zurückgeben, falls die angegebene Kante zwischen den beiden angegebenen `id`'s existiert und sonst `false`. Falls die Knoten zu den angegebenen `id`'s nicht existieren, soll eine `IllegalArgumentException` geworfen werden.
- `DirectedGraph.bfs` soll mittels Breitensuche die kürzeste Entfernung vom Knoten `iStart` zum Knoten `iEnd` zurückgeben, falls ein Pfad zwischen beiden Knoten existiert und sonst `null`. Falls die Knoten zu den angegebenen `id`'s nicht existieren, soll eine `IllegalArgumentException` geworfen werden. Sie sollen ausschließlich effiziente Strukturen verwenden. Zum „färben“ der Knoten bietet sich bspw. ein Array an.
- `DirectedGraph.dfs` soll mittels Tiefensuche die Länge *eines beliebigen* Pfads zwischen den beiden Knoten zurückgeben. Analog zu `DirectedGraph.bfs` soll `null` zurückgegeben werden, falls kein Pfad existiert und Fehlermeldungen geworfen werden, falls fehlerhafte Eingaben vorliegen. Auch hier sollen ausschließlich effiziente Strukturen verwendet werden.
- `DirectedGraph.readFile` soll die Datei unter dem angegebenen Pfad einlesen. Dabei steht in jeder Zeile zunächst ein Kontrollsymbol, dass angibt, was die Daten dahinter repräsentieren gefolgt von einem oder mehreren Werten. Beginnt eine Zeile mit `#` bedeutet dies, dass diese Zeile ignoriert werden kann, da es sich um einen Kommentar handelt. Beginnt eine Zeile mit `d`, soll ein gerichteter Graph erstellt werden. Der nachfolgende Wert gibt die Anzahl Knoten in dem Graphen an. Beginnt eine Zeile mit `e`, folgen darauf zwei Ganzzahlen, die den (0-basierten) Start und Zielknoten angeben. Sie dürfen davon ausgehen, dass es nur genau eine Zeile mit `d` beginnend gibt und dass diese vor den Kanteninformationen steht. Falls Ihr Code einen Fehler beim Einlesen der Datei feststellt, soll `null` zurückgegeben werden. Eine entsprechende Datei mit zugehörigem Graph könnte beispielsweise so aussehen:

```
# This defines a graph with 3 nodes.
d 3
# This defines the edges between graphs.
e 0 1
e 0 2
# Empty lines are allowed and
# should be skipped.

e 1 2
e 2 1
```



Geforderte Klassen und Methoden

```
public class Node {
  public int id;
  public ArrayList<Edge> outgoingEdges;
  public Node(int id) {...}
  public void addEdge(Node to) {...}
  public boolean hasEdge(Node to) {...}
}
public class Edge {
  public Node start;
  public Node end;
  public Edge(Node start, Node end) {...}
}
public class DirectedGraph {
  public DirectedGraph(int n) {...}
  public Node[] nodes;
  public Node getNode(int i) {...}
  public void addEdge(int i, int j) throws IllegalArgumentException {...}
  public boolean hasEdge(int i, int j) throws IllegalArgumentException
      {...}
  public Integer bfs(int iStart, int iEnd) throws IllegalArgumentException
      {...}
  public Integer dfs(int iStart, int iEnd) throws IllegalArgumentException
      {...}
  public static DirectedGraph readFile(String file) {...}
}
```