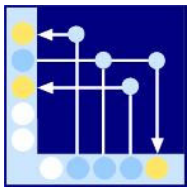

Algorithmen und Datenstrukturen 2

Vorlesung im Sommersemester 2017
Prof. Dr. habil. Christian Heinlein

3. Praktikumsaufgabe (12. Juni – 3. Juli 2017)

Aufgabe 3: Graphalgorithmen

Implementieren Sie folgende Graphalgorithmen in Java:

- Breitensuche
- Tiefensuche einschließlich topologischer Sortierung
- Bestimmung starker Zusammenhangskomponenten
- Bestimmung minimaler Spannbäume nach Prim
- Bestimmung kürzester Wege nach Bellman-Ford und Dijkstra

Erstellen Sie hierfür zu den Schnittstellen `Graph`, `WeightedGraph`, `BFS`, `DFS`, `SCC`, `MST` und `SP`, die in der Datei `graph.java` auf der Vorlesungswebseite vorgegeben sind, jeweils eine zugehörige Implementierungsklasse mit dem Suffix `Impl`, d. h. wenn eine Schnittstelle `XYZ` heißt, muss die zugehörige Implementierungsklasse `XYZImpl` heißen.

Die öffentlichen Konstruktoren von `GraphImpl` und `WeightedGraphImpl` erhalten als Parameter jeweils die Adjazenzlistendarstellung des Graphen als zweidimensionales Array von `int`-Werten (siehe Beispiel unten). Der Konstruktor von `WeightedGraphImpl` erhält als zweiten Parameter die zugehörigen Kantengewichte als zweidimensionales Array von `double`-Werten. Die übrigen Implementierungsklassen besitzen keinen expliziten Konstruktor.

Die Schnittstellen `Graph` und `WeightedGraph` definieren Methoden wie `size`, `deg` und `succ` zur Abfrage aller für die Algorithmen relevanten Eigenschaften eines Graphen. Dementsprechend darf ein Algorithmus nur die in der jeweiligen Schnittstelle definierten Methoden verwenden und keinerlei weitere Annahmen über den tatsächlichen Typ des übergebenen Graphobjekts machen.

Die Schnittstellen der einzelnen Algorithmen (wie z. B. `DFS`) definieren einerseits eine oder eventuell mehrere Methoden, um den jeweiligen Algorithmus auf einem Graphen auszuführen (z. B. `search` und `sort`), und andererseits Methoden, um anschließend die vom Algorithmus ermittelte Information abfragen zu können (z. B. `det` und `fin`). Eine typische Verwendung sieht daher wie folgt aus:

```
// Graph g erzeugen.
Graph g = new GraphImpl(new int [] [] {
    { 1, 2 },   // Knoten 0 hat als Nachfolger Knoten 1 und 2.
    { },        // Knoten 1 hat keine Nachfolger.
    { 2 }       // Knoten 2 hat als Nachfolger sich selbst.
});

// Tiefensuche auf g ausführen.
DFS d = new DFSImpl();
d.search(g);

// Die Knoten v des Graphen nach aufsteigenden Abschlusszeiten
// durchlaufen und für jeden Knoten seine Entdeckungs- und
// Abschlusszeit ausgeben.
for (int i = 0; i < g.size(); i++) {
    int v = d.sort(i);
    System.out.println(v + ": " + d.det(v) + " " + d.fin(v));
}
```

Für Algorithmen, die eine Vorrangwarteschlange benötigen, verwenden Sie die auf der Vorlesungswebseite bereitgestellte Implementierung `binheap.zip` gemäß Aufgabe 2!

Testen Sie Ihre Implementierung mit unterschiedlichen Graphen und ggf. unterschiedlichen Startknoten sorgfältig und ausführlich!

Auf der Vorlesungswebseite steht hierfür ein Testprogramm `graphtest.java` zur Verfügung, das abhängig von den übergebenen Kommandozeilenargumenten einen bestimmten Algorithmus auf einem bestimmten Graphen ggf. mit einem bestimmten Startknoten ausführt und die vom Algorithmus ermittelte Information auf der Standardausgabe ausgibt. Die Liste der Testgraphen kann nach Belieben erweitert werden.

Beim Algorithmus MST ist darauf zu achten, dass der jeweilige Testgraph ungerichtet sein muss, d. h. er muss zu jeder Kante auch die entgegengesetzte Kante mit dem gleichen Gewicht enthalten.

Abzuliefern ist entweder eine einzige Java-Datei mit allen Implementierungsklassen oder eine Zip-Datei mit beliebig vielen Java-Dateien auf oberster Ebene (keine Unterverzeichnisse, keine Pakete). Die vorgegebenen Schnittstellen – die nicht verändert werden dürfen! –, das Testprogramm sowie die Implementierung von Vorrangwarteschlangen müssen nicht mit abgeliefert werden.