Dokumentation

SKIZZE ZUR AUFGABE 2 AUS DER VORLESUNGSREIHE GKA

TEAM: ANTON, & MESUT

Aufgabenaufteilung

Grundsätzlich wurden alle Entwicklungsvorgänge und Aufgaben zusammen erarbeitet. Trotz dessen haben wir uns entschieden, die Aufgaben aufzuteilen, sodass wir schneller vorankommen.

- Mesut: Run.java, GraphReader.java, GraphSaver.java & GraphBuilder.java
- Anton: BreadFirstSearch.java und alle zugehörigen Tests

Quellenangaben

GraphStream Dokumentation: https://github.com/graphstream

Verschiedene Lösungswege von stackoverflow.com

Lehrveranstaltung

Bearbeitungszeitraum

Datum	**Dauer in Stunden**
Anton	20
Mesut	20
Gesamt	**40**

Aktueller Stand

Unsere Software funktioniert vollständig und die Tests durchlaufen alle eine positive Instanz.

Funktionen unserer Software

Folgende Funktionen haben wir aus dem Aufgabenblatt herausgelesen, sodass wir Sie ebenfalls auch implementiert haben:

- Es können .gka-Dateien eingelesen werden
  - Es können ebenfalls .gka-Dateien gespeichert werden
- Es kann mithilfe des Dijkstra und Floyd Algorithmen den kürzesten Weg von einem Vertex zu einem anderen angegebenen Vertex finden.
- Vollständige Tests
- Eine vollständige Javadoc.

Algorithmus

Dijkstra:

Unsere Umsetzung des Dijkstra Algorithmus lehnt sich stark an die Folien aus der Vorlesung an.
Wir suchen uns zunächst einen Startpunkt, also unseren SourceNode. Den setzen wir auf True, weil unser SourceNode als Vorgänger sich selbst hat und die Entfernung 0 beträgt, also der kürzeste Weg.
Anschließend suchen wir uns Nachbarknoten von Source und merken uns die Entfernung, die zurückgelegt werden muss, sowie den Vorgänger (in diesem Fall unser SourceNode). Dabei sind die Nachbarknoten von Source in dem Moment alle auf dem Status false, da für diese Knoten noch keine Berechnung für die optimale Distanz stattgefunden hat.

Jetzt arbeiten wir uns systematisch Knoten für Knoten durch, die alle den Status false haben. Dabei müssen wir stets nach der kürzesten Distanz suchen. Falls es für einen Knoten, der bereits eine Distanz besitzt eine neue kürzere Strecke gibt, wird diese ersetzt und der Vorgänger geändert. Wenn wir den kürzesten Weg gefunden haben setzen wir den Knoten auf true und fahren mit dem nächsten Knoten fort (der noch den Status false hat).

Sobald wir das TargetNode gefunden haben, gehen wir rückwärts durch den Graphen und suchen nach den Vorgängern von TargetNode, diese werden in eine Liste gespeichert. Sollten keine Vorgänger existieren, so erzeugen wir die Meldung kein Pfad gefunden. Am SourceNode wieder angelangt, drehen wir unsere Liste für die Ausgabe.

Floyd:

Der Floyd Algorithmus sucht den optimalen Weg von allen Knoten zu den anderen.
Alle Knoten, die nicht mit dem aktuellen Knoten durch direkte Kanten verbunden sind, erhalten am Anfang den Wert „Unendlich“.
Ziel vom Algorithmus, ist es den kürzesten den Weg zu finden für alle möglichen Start und Endknoten.
Also arbeiten wir uns Zeile und Spalte durch, da wir mit einer Matrix arbeiten. Die Anzahl der jeweiligen Schritte bestimmt unser K (Anzahl der Knoten im Graph).
Sobald wir eine Distanz gefunden haben, die kleiner ist, als unsere aktuelle Distanz wird diese in der Matrix ersetzt und somit unser Weg optimiert.

Anhand eines Beispiels möchten wir illustrieren, wie der Algorithmus arbeitet:

K=0

	1	2	3	4	5
1	0	6	4	∞	∞
2	∞	0	7	5	∞
3	1	8	0	3	2
4	1	∞	∞	0	5
5	∞	∞	4	∞	0

Wir haben jetzt alle Gewichtungen aus dem Graphen in die Matrix Tabelle übertragen und können jetzt mit Hilfe des Algorithmus nach dem optimalen Weg suchen. Als nächstes erhöhen wir unser K um 1 und führen den Algorithmus aus.

K=1

	1	2	3	4	5
1	0	6	4	∞	∞
2	∞	0	7	5	∞
3	1	7	0	3	2
4	1	7	5	0	5
5	∞	∞	4	∞	0

In diesem Schritt schauen wir uns die Spalte und die Zeile 1 an. Wir schauen uns unser j und i an, addieren diese und falls der Wert kleiner ist als der, der eingetragen ist, wird dieser ersetzt. Die ersetzten Werte werden farblich hinterlegt.
In diesem Fall finden wir drei kürzere Wege, sodass wir diese in die Tabelle eintragen können.

Den Floyd Algorithmus führen wir nun für alle K's durch, bis wir in allen Fällen die optimalen Wege gefunden haben.

JUnit-Testfälle

Dijkstra:

Wir haben uns entschieden Stichprobenmäßig Testfälle in die Dokumentation einzufügen.
Anbei sind drei Einlesetests aus der graph03.gka – Datei. Die Beispiele beinhalten zwei positive sowie einen negativen Test. Weitere Tests können sie aus unserem scr-code entnehmen.

```
    @Test
    public void getShortestPathTwo() throws Exception {
        DijkstraAlgorithm expO = new DijkstraAlgorithm();
        DijkstraAlgorithm resO = new DijkstraAlgorithm();
Graph graph03 = GraphReader.openFile(new
File("graph/subwerkzeuge/bspGraphen/graph03.gka"));
        expO.init(graph03);
        resO.init(graph03);
        List<Node> exp = expO.getShortestPath(graph03.getNode("Münster"),
graph03.getNode("Hamburg"));
        List<Node> res = resO.getShortestPath(graph03.getNode("Münster"),
graph03.getNode("Hamburg"));
        assertEquals(exp, res);
        System.out.println("getShortestPathTwo() ok");
    }
```

```
@Test
    public void testOwnDijk() throws Exception {
        DijkstraAlgorithm exp = new DijkstraAlgorithm();
        Graph owng = new SingleGraph("owng");
        owng.addNode("a");
        owng.addNode("b");
        owng.addNode("c");
        owng.addNode("d");
        owng.addNode("e");
        owng.addNode("f");
        owng.addNode("g");

        owng.addEdge("ab", "a", "b").addAttribute("weight", "2");
        owng.addEdge("ac", "a", "c").addAttribute("weight", "10");
        owng.addEdge("bc", "b", "c").addAttribute("weight", "6");
        owng.addEdge("bd", "b", "d").addAttribute("weight", "15");
        owng.addEdge("cd", "c", "d").addAttribute("weight", "2");
        owng.addEdge("de", "d", "e").addAttribute("weight", "1");
        owng.addEdge("df", "d", "f").addAttribute("weight", "5");
        owng.addEdge("dg", "d", "g").addAttribute("weight", "20");
        owng.addEdge("ef", "e", "f").addAttribute("weight", "4");
        owng.addEdge("fg", "f", "g").addAttribute("weight", "3");
        exp.init(owng);
        List<Node> listeExpected = exp.getShortestPath(owng.getNode("a"),
owng.getNode("g"));
        List<Node> result = new ArrayList<>();
        result.add(owng.getNode("a"));
        result.add(owng.getNode("b"));
        result.add(owng.getNode("c"));
        result.add(owng.getNode("d"));
        result.add(owng.getNode("f"));
        result.add(owng.getNode("g"));
```

```
        assertEquals(result, listeExpected);
        System.out.println("ownDijk() is ok");
    }

@Test (expected = NullPointerException.class)
    public void negGetPath() throws Exception {
            graph03 = GraphReader.openFile(new
File("src/graph/subwerkzeuge/graph03.gka"));
            init();
            expected.getPath(graph03.getNode("Exist"), graph03.getNode("Exist"));
        result.getPath(graph03.getNode("Exist"), graph03.getNode("Exist"));
        assertEquals(expected.toString(), result.toString());

        System.out.println("negGetPath() is ok");
    }
```

Floyd:

```
    @Test
    public void getShortestPathTwo() throws Exception {
        Graph graph03 = GraphReader.openFile(new
File("graph/subwerkzeuge/bspGraphen/graph03.gka"));
        List<Node> exp = FloydWarshall.getShortestPath(graph03,
graph03.getNode("Münster"), graph03.getNode("Hamburg"));
        List<Node> res = FloydWarshall.getShortestPath(graph03,
graph03.getNode("Münster"), graph03.getNode("Hamburg"));
        assertEquals(exp, res);
        System.out.println("getShortestPathTwo() ok");
    }
@Test
    public void testOwnFlyod() throws Exception {
        Graph owng = new SingleGraph("owng");
        owng.addNode("a");
        owng.addNode("b");
        owng.addNode("c");
        owng.addNode("d");
        owng.addNode("e");
        owng.addNode("f");
        owng.addNode("g");

        owng.addEdge("ab", "a", "b").addAttribute("weight", "2");
        owng.addEdge("ac", "a", "c").addAttribute("weight", "10");
        owng.addEdge("bc", "b", "c").addAttribute("weight", "6");
        owng.addEdge("bd", "b", "d").addAttribute("weight", "15");
        owng.addEdge("cd", "c", "d").addAttribute("weight", "2");
        owng.addEdge("de", "d", "e").addAttribute("weight", "1");
        owng.addEdge("df", "d", "f").addAttribute("weight", "5");
        owng.addEdge("dg", "d", "g").addAttribute("weight", "20");
        owng.addEdge("ef", "e", "f").addAttribute("weight", "4");
        owng.addEdge("fg", "f", "g").addAttribute("weight", "3");

        List<Node> listeExpected = FloydWarshall.getShortestPath(owng,
        owng.getNode("a"), owng.getNode("g"));
        assertEquals("[a, b, c, d, f, g]", listeExpected.toString());
        System.out.println("ownDijk() is ok");
    }
```

```
    @Test(expected = NullPointerException.class)
    public void negGetPath() throws Exception {
        Graph graph03 = GraphReader.openFile(new
File("graph/subwerkzeuge/bspGraphen/graph03.gka"));
        FloydWarshall.getShortestPath(graph03, graph03.getNode("ExistiertNicht"),
graph03.getNode("ExistiertNicht"));
        FloydWarshall.getShortestPath(graph03, graph03.getNode("ExistiertNicht"),
graph03.getNode("ExistiertNicht"));
        System.out.println("negGetPath() ok");
    }
```

Fragebogen

1. **Bekommen Sie für einen Graphen immer den gleichen kürzesten Weg? Warum?**
   Im worst case gibt für einen Pfad mehrere kürzeste Wege in einem Graphen. Es kann passieren, dass unsere beiden Algorithmen unterschiedliche Ergebnisse für einen Weg liefern, bzw. es kann nicht garantiert werden, dass beide Algorithmen denselben Weg liefern.
2. **Was passiert, wenn der Eingabegraph negative Kantengewichte hat?**
   Die negativen Kantengewichte fangen wir ab und werfen eine Exception Meldung. -> ungültiger Graph. Bei Floyd können wir alles verarbeiten, solange kein negativer Kreis entsteht.
3. **Wie allgemein ist Ihre Konstruktion von BIG, kann jeder beliebige, gerichtete Graph erzeugt werden?**
   Die Anzahl der Knoten und Kanten können wir selbst bestimmen. Die erste Kante setzen wir fest, da wir nicht nach „Random“ Kanten suchen können, weil wir die Kanten danach Random erzeugen. Die Gewichtung der Kanten ist ebenfalls zufällig. Alle Kanten sind gerichtet.
4. **Wie testen Sie für BIG, ob Ihre Implementierung den kürzesten Weg gefunden hat?**
   Wir prüfen bei Floyd vs. Dijkstra auf die gleiche Distanz, die zurückgelegt wurde. Der Weg kann sich, jedoch unterscheiden.
5. **Wie mussten Sie Ihre Lösung erweitern, um die Menge der kürzesten Wege zu bekommen?**
   Bei dem Floyd Algorithmus haben wir bereits eine HashMap, die uns alle möglichen Kombinationen an Wegen für jeden Knoten anzeigt.
   Bei Dijkstra müssten wir diese zusätzlich implementieren.
6. **Wie mussten Sie Ihre Lösung erweitern, damit die Suche nicht-deterministisch ist?**
   Bei Floyd könnten wir den Ablauf der Knoten ändern. Zum Beispiel für j=[1,2,3,4] j=[2,1,4,3]. Das Resultat wäre das gleiche, nur unser Weg wäre anders.
   Bei Dijkstra könnten wir die Abfrage auf die minimalste Distanz weglassen, sodass es dazu führen würde, dass unsere Nachbarknoten per Zufall abgearbeitet werden könnten. Somit hätten wir immer unterschiedliche Wege, aber am Ende stets den kürzesten Weg.