

Génie Logiciel partie tests fonctionnels et structurels

Groupe TP41 : Floréal RISSO et Téo TINARRAGE

Question 0

- a) Assurez-vous que vous disposez d'une implémentation en Java de l'algorithme de Dijkstra, que vous avez bien identifié les entrées et les sorties du programme.

L'implémentation utilisée est celle proposée dans le [sujet \(https://steemit.com/graphs/@drifter1/programming-java-graph-shortest-path-algorithm-dijkstra\)](https://steemit.com/graphs/@drifter1/programming-java-graph-shortest-path-algorithm-dijkstra).

- b) Mettez au propre la spécification de ce programme.

L'implémentation utilisée est celle proposée en exemple, étendue par nous-même. Elle applique l'algorithme et réalise les tests sur ses résultats (au lieu d'avoir un script de test qui appelle un programme implémentant l'algorithme, tout est réalisé par un seul programme).

Les fichiers de l'implémentation d'origine sont présents tels quels et nos ajouts se situent dans d'autres classes qui exploitent les précédentes.

Partie Algo de Dijkstra

Entrées : liste non-vide d'arcs (*orientés ou non*) pondérés (c'est à dire numéro de sommet de départ, numéro de sommet d'arrivée, poids(float)) décrivant un graphe (l'existence d'un sommet est impliquée par l'existence d'un arc l'ayant comme extrémité), ainsi que le *numéro du sommet de départ* pour le calcul des distances, au format JSON*

Sortie : *Distance* entre chaque sommet du graphe et le sommet de départ choisi, ("Distance" désignant la somme des poids des arcs parcourus par le chemin minimisant cette somme)

*structure attendue du JSON :

Pour les données d'entrée

```
{
  "graphs" : [
    {
      "departDijkstra" : <numéro du sommet de départ de l'algo>,
      "estOriente" : <booléen>
      "chemins" : [
        {
          "depart" : <numéro sommet de départ>,
          "arrivée" : <numéro sommet d'arrivée>,
          "poids" : <poids de l'arc (flottant)>
        },
        ...
      ]
    }
  ]
}
```

Partie Test

Entrée : Résultats attendus (oracles) pour l'application de l'algorithme aux données d'entrées spécifiées

Sortie : résultat des tests.

Pour les résultats attendus :

```
{  
    "results": [  
        "distances" : [...] <liste des distances attendues>  
        "parents" : [...] <liste des parents de chaque sommets>  
    ]  
}
```

Question 1

Le programme de test est [test.java](#) (`./dijkstra/src/test/Test.java`).

Son fonctionnement est simple, dans un premier temps il lit les jeux de tests et les oracles.

```
private static final String testFile = "./tests.json";  
private static final String resultFile = "./results.json";  
  
// La fonction qui lit les jeux de tests écrit dans le fichier "./tests.json".  
private static Tester getTests();  
  
// La classe qui contient le contenu du fichier de test.  
public static class Tester {  
    private Graph[] graphs; // les graphes  
    private int[] startDijkstra; // le sommet de départ  
}  
  
// La fonction qui lit les oracles  
private static Result[] getResults();  
  
// La classe qui contient le résultat attendu d'un jeu de test.  
public class Result {  
    private float[] distances;  
    private int[] parents;  
}
```

Ensuite, on appelle la fonction qui calcule la distance de Dijkstra sur l'ensemble des jeux de tests et on compare le résultat attendu du résultat obtenu.

```
// Parcourt des jeux de tests  
for(Graph graph : graphs) {  
    // Calcul du résultat  
    Result result = TestGraphs.Dijkstra(graph,tester.startDijkstra[i]);  
    // Comparaison du résultat au résultat attendu  
    TestResult testResult = expectedResults[i].equals(result);  
}
```

La fonction de comparaison donne de nombreuses informations sur les différences afin d'aider le débogage et à la fin du programme il est indiqué le nombre de test réussi sur le nombre de test tenté.

```
// La fonction qui vérifie l'égalité des résultats attendus et calculés.
public TestResult equals(Result r) {
    // Des branchements if / else
    // ...
    return new TestResult("Un message expliquant la différence...");
}

// La classe renvoyée par equals :
public class TestResult {
    // Les résultats sont les mêmes
    private boolean areEquals;
    // Le message d'erreur
    private String error;
}

// Enfin le print final qui compare les nombres de tests réussis et échoués.
System.out.println("Résultat : ["+passed+"/"+i+"]");
```

Question 2

Voici les captures d'écrans montrant les lignes de code exécutées avec le jeu donné dans l'énoncé.

Le jeu de test :

```
{
  "graphs": [
    {
      "estOriente": false,
      "departDijkstra": 0,
      "sommets": 6,
      "chemins": [
        {
          "depart": 0,
          "arrivee": 1,
          "poids": 2.0
        },
        {
          "depart": 0,
          "arrivee": 4,
          "poids": 3.0
        },
        {
          "depart": 1,
          "arrivee": 4,
          "poids": 4.0
        },
        {
          "depart": 1,
```

```

    "arrivee": 2,
    "poids": 5.0
},
{
    "depart": 1,
    "arrivee": 5,
    "poids": 2.0
},
{
    "depart": 2,
    "arrivee": 3,
    "poids": 2.0
},
{
    "depart": 2,
    "arrivee": 5,
    "poids": 4.0
},
{
    "depart": 3,
    "arrivee": 4,
    "poids": 2.0
},
{
    "depart": 3,
    "arrivee": 5,
    "poids": 5.0
},
{
    "depart": 4,
    "arrivee": 5,
    "poids": 2.0
}
]
}
]
}

```

Le résultat attendu :

```
{
    "results": [
        {
            "distances" :[0.0, 2.0, 7.0,3.4028235E38,3.0,4.0],
            "parents" :[-1,0,1,-1,0,1]
        }
    ]
}
```

La couverture :

GitHub - dijkstra/src/test/Test.java - Eclipse IDE

```
1 package test;
2
3 import java.io.FileReader;
4
5 public class Test {
6
7     public static class Tester {
8         private Graph[] graphs;
9         private int[] startDijkstra;
10    public Tester(Graph[] graphs, int[] startDijkstra) {
11        this.graphs = graphs;
12        this.startDijkstra = startDijkstra;
13    }
14}
15
16 private static final String testFile = "./t1.json";
17 private static final String resultfile = "./result.json";
18
19 private static void addPath(Graph graph, JSONArray chemins, Boolean isOriented) {
20     for(Object oChemin : chemins) {
21         JSONObject jChemin = (JSONObject) oChemin;
22         Long depart = (Long) jChemin.get("depart");
23         Long arrive = (Long) jChemin.get("arrive");
24         Double poids = (Double) jChemin.get("poids");
25         graph.addEdge(depart.intValue(), arrive.intValue(), poids.floatValue());
26         if(!isOriented) {
27             graph.addEdge(arrive.intValue(), depart.intValue(), poids.floatValue());
28         }
29     }
30 }
31
32 /**
33 * Makes a array of the graphs from the .JSON
34 * @return the array of graphs
35 */
36 private static Tester getTests() {
37     Graph[] testGraphs = null;
38     int[] startDijkstra = null;
39     JSONParser parser = new JSONParser();
40     try {
41         Object obj = parser.parse(new FileReader(testFile));
42         JSONObject jsonObject = (JSONObject) obj;
43         JSONArray graphs = (JSONArray) jsonObject.get("graphs");
44
45         int size = graphs.size();
46         int i = 0;
47         testGraphs = new Graph[size];
48         startDijkstra = new int[size];
49         /**
50          * for graphs in the .JSON
51          */
52         for(Object oGraph : graphs) {
53             Graph graph = new Graph();
54             JSONObject jGraph = (JSONObject) oGraph;
55             JSONArray jParents = (JSONArray) jGraph.get("parents");
56             double[] distances = Arrays.stream(jParents.toArray()).mapToDouble(o -> (Double)o).toArray();
57             long[] parents = Arrays.stream(jParents.toArray()).mapToInt(o -> (Long)o).toArray();
58             testGraphs[i] = graph;
59             startDijkstra[i] = (int) jGraph.get("start");
60             i++;
61         }
62     } catch (Exception e) {
63         e.printStackTrace();
64     }
65     return new Tester(testGraphs, startDijkstra);
66 }
```

Writable Smart Insert | 115 : 1 : 3414

GitHub - dijkstra/src/test/Test.java - Eclipse IDE

```
95     JSONArray results = (JSONArray) jsonObject.get("results");
96     int size = results.size();
97     int i = 0;
98     testResults = new Result[size];
99
100    for(Object oResult : results) {
101        JSONObject jResult = (JSONObject) oResult;
102        JSONArray jDistances = (JSONArray) jResult.get("distances");
103        JSONArray jParents = (JSONArray) jResult.get("parents");
104        double[] distances = Arrays.stream(jDistances.toArray()).map.ToDouble(o -> (Double)o).toArray();
105        long[] parents = Arrays.stream(jParents.toArray()).mapToInt(o -> (Long)o).toArray();
106        testResults[i++] = new Result(convertor.doubleToFloat(distances), convertor.longToInt(parents));
107    }
108 } catch (Exception e) {
109     e.printStackTrace();
110 }
111 return testResults;
112 }
```

```
114    public static void main(String[] args) {
115        Tester tester = Test.getTests();
116        Graph[] graphs = tester.graphs;
117        Result[] expectedResults = Test.getResults();
118
119        // Some verifications on the read data...
120        if(graphs == null) {
121            System.err.println("Erreur il n'y aucun graph, ou la lecture n'a pas fonctionnée...");
122            return;
123        }
124        if(expectedResults == null) {
125            System.err.println("Erreur il n'y aucun resultat, ou la lecture n'a pas fonctionnée...");
126            return;
127        }
128        if(graphs.length!=expectedResults.length) {
129            System.err.println("Erreur il n'y pas autant de graphs(\""+graphs.length+"\") que de resultat(\""+expectedResults.length+"\")...");
130            return;
131        }
132
133        // Commence the result for all graphs and compare to the expected results
134        int i=0;
135        int passed = 0;
136        for(Graph graph : graphs) {
137            Result result = TestGraphs.Dijkstra(graph,tester.startDijkstra[i]);
138            //System.out.println(result);
139            TestResult testResult = expectedResults[i].equals(result);
140            passed += testResult ? 1 : 0;
141            testResult.print();
142            i++;
143        }
144        System.out.println("Resultat : ["+passed+"/"+i+"]");
145
146    }
147 }
148 }
```

Writable Smart Insert | 115 : 1 : 3414

GitHub - dijkstra/src/test/Test.java - Eclipse IDE

```
95     JSONArray results = (JSONArray) jsonObject.get("results");
96     int size = results.size();
97     int i = 0;
98     testResults = new Result[size];
99
100    for(Object oResult : results) {
101        JSONObject jResult = (JSONObject) oResult;
102        JSONArray jDistances = (JSONArray) jResult.get("distances");
103        JSONArray jParents = (JSONArray) jResult.get("parents");
104        double[] distances = Arrays.stream(jDistances.toArray()).map.ToDouble(o -> (Double)o).toArray();
105        long[] parents = Arrays.stream(jParents.toArray()).mapToInt(o -> (Long)o).toArray();
106        testResults[i++] = new Result(convertor.doubleToFloat(distances), convertor.longToInt(parents));
107    }
108 } catch (Exception e) {
109     e.printStackTrace();
110 }
111 return testResults;
112 }
```

Writable Smart Insert | 115 : 1 : 3414

```
114● public static void main(String[] args) {
115     Teste tester = new Tests();
116     Graph[] graphs = tester.getGraphs();
117     Result[] expectedResults = test.getResults();
118
119     // Some verifications on the read data...
120     if(graphs == null) {
121         System.err.println("Erreur il n'y aucun graph, ou la lecture n'a pas fonctionnée...");
122         return;
123     }
124     if(expectedResults==null) {
125         System.err.println("Erreur il n'y aucun résultat, ou la lecture n'a pas fonctionnée...");
126         return;
127     }
128     if(graphs.length!=expectedResults.length) {
129         System.err.println("Erreur il n'y pas autant de graphs("+graphs.length+") que de résultat("+expectedResults.length+ ")");
130         return;
131     }
132
133     // Computes the result for all graphs and compare to the expected results
134     int i = 0;
135     int passed = 0;
136     for(Graph graph : graphs) {
137         Result result = TestGraphs.Dijkstra(graph,tester.startDijkstra[i]);
138         //System.out.println(result);
139         if(result.equals(expectedResults[i])) {
140             passed += testResult.areEquals();
141             testResult.print();
142         }
143         i++;
144     }
145     System.out.println("Résultat : ["+passed+"/"+i+"]");
146
147 }
148 }
```

Writable

Smart Insert

115 : 1 : 3414



```
GitHub - dijkstra/src/dijkstra/TestGraphs.java - Eclipse IDE
1 package dijkstra;
2
3 import java.util.Iterator;
4
5 public class TestGraphs {
6
7     public static void main(String[] args) {
8         Graph g = new Graph();
9
10        System.out.println("Graph:");
11        // add Edges
12        g.addEdge(0, 1, 5.2f);
13        g.addEdge(0, 3, 12.5f);
14        g.addEdge(1, 2, 3.1f);
15        g.addEdge(1, 3, 5.9f);
16        g.addEdge(1, 4, 15.2f);
17        g.addEdge(2, 1, 1.5f);
18        g.addEdge(2, 3, 2.3f);
19        g.addEdge(3, 4, 8.5f);
20        g.addEdge(4, 2, 2.7f);
21
22        // print Graph
23        g.printGraph();
24
25        // Dijkstra Shortest Path Algorithm
26        System.out.println("Dijkstra Shortest Path:");
27        Dijkstra(g, 0);
28    }
29
30    public static Result Dijkstra(Graph g, int startVertex) {
31        // for storing distances after removing vertex from Queue
32        float[] distances = new float[g.getCount()];
33        // for storing father id's after removing vertex from Queue
34        int[] parents = new int[g.getCount()];
35        for (int i = 0; i < g.getCount(); i++) {
36            for (int j = 0; j < g.getCount(); j++) {
37                parents[i][j] = -1;
38            }
39        }
40        // set up vertex queue
41        PriorityQueue<Vertex> Q = new PriorityQueue<Vertex>();
42        for (int i = 0; i < g.getCount(); i++) {
43            if (i != startVertex) {
44                Q.add(new Vertex(i));
45            }
46        }
47        // add startVertex
48        Vertex node = new Vertex(startVertex);
49        node.setDistance(0);
50        Q.add(node);
51
52        // loop through all vertices
53        while (!Q.isEmpty()) {
54            // get vertex with shortest distance
55            Vertex u = Q.remove();
56
57            // iterate through all neighbours
58            Iterator<Edge> it = u.getNeighbours(u.getId()).iterator();
59            while (it.hasNext()) {
60                Edge e = it.next();
61                Iterator<Vertex> it2 = e.iterator();
62                while (it2.hasNext()) {
63                    Vertex v = it2.next();
64                    // check if vertex was visited already
65                    if (e.getEndPoint() != v.getId()) {
66                        continue;
67                    }
68                    // check distance
69                    if (v.getDistance() > u.getDistance() + e.getWeight()) {
70                        v.setDistance(u.getDistance() + e.getWeight());
71                        v.setParent(u);
72                        parents[v.getId()] = v.getParent().getId();
73                    }
74                }
75            }
76        }
77
78    }
79
80    // print final shortest paths
81    /*
82     * System.out.println("Vertex\tDistance\tParent Vertex");
83     * for (int i = 0; i < g.getCount(); i++) {
84     *     System.out.println(i + "\t" + distances[i] + "\t" + parents[i]);
85     * }
86     */
87
88    return new Result(distances, parents);
89
90}
91 }
```

```
GitHub - dijkstra/src/dijkstra/TestGraphs.java - Eclipse IDE
1 package dijkstra;
2
3 import java.util.PriorityQueue;
4
5 public class TestGraphs {
6
7     public static void main(String[] args) {
8         Graph g = new Graph();
9
10        System.out.println("Graph:");
11        // add Edges
12        g.addEdge(0, 1, 5.2f);
13        g.addEdge(0, 3, 12.5f);
14        g.addEdge(1, 2, 3.1f);
15        g.addEdge(1, 3, 5.9f);
16        g.addEdge(1, 4, 15.2f);
17        g.addEdge(2, 1, 1.5f);
18        g.addEdge(2, 3, 2.3f);
19        g.addEdge(3, 4, 8.5f);
20        g.addEdge(4, 2, 2.7f);
21
22        // set up vertex queue
23        PriorityQueue<Vertex> Q = new PriorityQueue<Vertex>();
24        for (int i = 0; i < g.getCount(); i++) {
25            if (i != startVertex) {
26                Q.add(new Vertex(i));
27            }
28        }
29
30        // add startVertex
31        Vertex node = new Vertex(startVertex);
32        node.setDistance(0);
33        Q.add(node);
34
35        // loop through all vertices
36        while (!Q.isEmpty()) {
37            // get vertex with shortest distance
38            Vertex u = Q.remove();
39            distances[u.getId()] = u.getDistance();
40
41            // iterate through all neighbours
42            Iterator<Edge> it = u.getNeighbours(u.getId()).iterator();
43            while (it.hasNext()) {
44                Edge e = it.next();
45                Iterator<Vertex> it2 = e.iterator();
46                while (it2.hasNext()) {
47                    Vertex v = it2.next();
48                    // check if vertex was visited already
49                    if (e.getEndPoint() != v.getId()) {
50                        continue;
51                    }
52                    // check distance
53                    if (v.getDistance() > u.getDistance() + e.getWeight()) {
54                        v.setDistance(u.getDistance() + e.getWeight());
55                        v.setParent(u);
56                        parents[v.getId()] = v.getParent().getId();
57                    }
58                }
59            }
60        }
61
62        // print final shortest paths
63        /*
64         * System.out.println("Vertex\tDistance\tParent Vertex");
65         * for (int i = 0; i < g.getCount(); i++) {
66         *     System.out.println(i + "\t" + distances[i] + "\t" + parents[i]);
67         * }
68         */
69
70    return new Result(distances, parents);
71
72}
73 }
```

GitHub - dijkstra/src/dijkstra/Vertex.java - Eclipse IDE

```
1 package dijkstra;
2
3 public class Vertex implements Comparable<Vertex>{
4     private int id;
5     private float distance;
6     private Vertex parent;
7
8     public Vertex(){
9         distance = Float.MAX_VALUE; // "infinity"
10    parent = null;
11 }
12
13     public Vertex(int id){
14         this.id = id;
15         distance = Float.MAX_VALUE; // "infinity"
16         parent = null;
17     }
18
19     public int getId() {
20         return id;
21     }
22
23     public void setId(int id) {
24         this.id = id;
25     }
26
27     public float getDistance() {
28         return distance;
29     }
30
31     public void setDistance(float distance) {
32         this.distance = distance;
33     }
34
35     public Vertex getParent() {
36         return parent;
37     }
38
39     public void setParent(Vertex parent){
40         this.parent = parent;
41     }
42
43     public int compareTo(Vertex other) {
44         return Double.compare(this.distance, other.distance);
45     }
46 }
```

GitHub - dijkstra/src/dijkstra/Edge.java - Eclipse IDE

```
1 package dijkstra;
2
3 public class Edge implements Comparable<Edge>{
4     private int startPoint;
5     private int endPoint;
6     private float weight;
7
8     public Edge(int startPoint, int endPoint, float weight) {
9         this.startPoint = startPoint;
10        this.endPoint = endPoint;
11        this.weight = weight;
12    }
13
14    public int getStartPoint() {
15        return startPoint;
16    }
17
18    public int getEndPoint() {
19        return endPoint;
20    }
21
22    public float getWeight() {
23        return weight;
24    }
25
26    public boolean equals(Edge other) {
27        if (this.startPoint == other.startPoint) {
28            if (this.endPoint == other.endPoint) {
29                return true;
30            }
31        }
32        return false;
33    }
34
35    public int compareTo(Edge other) {
36        return Double.compare(this.weight, other.weight);
37    }
38
39    public String toString() {
40        return startPoint + "-" + endPoint + "(" + weight + ")";
41    }
42
43 }
```

The screenshot shows the Eclipse IDE interface with the Graph.java file open in the central editor window. The code implements a graph structure using PriorityQueues for edges. It includes methods for adding edges, removing edges, checking if an edge exists, getting neighbors, printing the graph, and converting it to a string. The code uses annotations like `@SuppressWarnings("unchecked")` and `@Override`.

```
1 package dijkstra;
2
3 import java.util.Iterator;
4
5 public class Graph {
6     private int vCount;
7     private PriorityQueue<Edge>[] adj;
8
9     public int getCount() {
10         return vCount;
11     }
12
13     @SuppressWarnings("unchecked")
14     public Graph(int vCount) {
15         this.vCount = vCount;
16         // initialize adj
17         adj = new PriorityQueue[vCount];
18         for (int i = 0; i < vCount; i++) {
19             adj[i] = new PriorityQueue<Edge>();
20         }
21     }
22
23     public void addEdge(int i, int j, float weight) {
24         adj[i].add(new Edge(i, j, weight));
25     }
26
27     public void addEdge(Edge e) {
28         adj[e.getStartPoint()].add(e);
29     }
30
31     public void removeEdge(int i, int j) {
32         Iterator<Edge> it = adj[i].iterator();
33         Edge other = new Edge(i, j, 0);
34         while (it.hasNext()) {
35             if (it.next().equals(other)) {
36                 it.remove();
37             }
38         }
39     }
40
41     public boolean hasEdge(Edge e) {
42         return adj[e.getStartPoint()].contains(e);
43     }
44
45     public PriorityQueue<Edge> neighbours(int vertex) {
46         return adj[vertex];
47     }
48
49     public void printGraph() {
50         for (int i = 0; i < vCount; i++) {
51             PriorityQueue<Edge> edges = neighbours(i);
52             Iterator<Edge> it = edges.iterator();
53             System.out.print(i + ": ");
54             for (int j = 0; j < edges.size(); j++) {
55                 System.out.print(it.next() + " ");
56             }
57             System.out.println();
58         }
59     }
60
61     public String toString() {
62         StringBuilder builder = new StringBuilder();
63         for (int i = 0; i < vCount; i++) {
64             PriorityQueue<Edge> edges = neighbours(i);
65             Iterator<Edge> it = edges.iterator();
66             builder.append(i + ": ");
67             for (int j = 0; j < edges.size(); j++) {
68                 builder.append(it.next() + " ");
69             }
70             builder.append("\n");
71         }
72         return builder.toString();
73     }
74 }
75
76 }
```

This screenshot shows the same Eclipse IDE session with the Graph.java code, but it includes additional print statements within the `printGraph()` and `toString()` methods to output the graph structure. The code is identical to the one in the first screenshot, except for the added logging logic.

```
1 package dijkstra;
2
3 import java.util.PriorityQueue;
4 import java.util.Iterator;
5 import java.util.List;
6 import java.util.ArrayList;
7 import java.util.LinkedList;
8
9 public class Graph {
10     private int vCount;
11     private PriorityQueue<Edge>[] adj;
12
13     public void addEdge(int i, int j, float weight) {
14         adj[i].add(new Edge(i, j, weight));
15     }
16
17     public void addEdge(Edge e) {
18         adj[e.getStartPoint()].add(e);
19     }
20
21     public void removeEdge(int i, int j) {
22         Iterator<Edge> it = adj[i].iterator();
23         Edge other = new Edge(i, j, 0);
24         while (it.hasNext()) {
25             if (it.next().equals(other)) {
26                 it.remove();
27             }
28         }
29     }
30
31     public boolean hasEdge(Edge e) {
32         return adj[e.getStartPoint()].contains(e);
33     }
34
35     public PriorityQueue<Edge> neighbours(int vertex) {
36         return adj[vertex];
37     }
38
39     public void printGraph() {
40         for (int i = 0; i < vCount; i++) {
41             PriorityQueue<Edge> edges = neighbours(i);
42             Iterator<Edge> it = edges.iterator();
43             System.out.print(i + ": ");
44             for (int j = 0; j < edges.size(); j++) {
45                 System.out.print(it.next() + " ");
46             }
47             System.out.println();
48         }
49     }
50
51     public String toString() {
52         StringBuilder builder = new StringBuilder();
53         for (int i = 0; i < vCount; i++) {
54             PriorityQueue<Edge> edges = neighbours(i);
55             Iterator<Edge> it = edges.iterator();
56             builder.append(i + ": ");
57             for (int j = 0; j < edges.size(); j++) {
58                 builder.append(it.next() + " ");
59             }
60             builder.append("\n");
61         }
62         return builder.toString();
63     }
64 }
65
66 }
```

The screenshot shows the Eclipse IDE interface with the title "GitHub - dijkstra/src/dijkstra/Result.java - Eclipse IDE". The code editor displays the `Result.java` class. The code is annotated with several red error markers, indicating syntax or compilation errors. The class defines a constructor, methods for getting distances and parents arrays, and an equals method. The equals method compares the lengths of the distances and parents arrays, and then iterates through each element to check if they are equal.

```
1 package dijkstra;
2
3 import utilities.TestResult;
4
5 public class Result {
6     private float[] distances;
7     private int[] parents;
8
9     public Result(float[] distances, int[] parents) {
10        super();
11        this.distances = distances;
12        this.parents = parents;
13    }
14
15    public float[] getDistances() {
16        return distances;
17    }
18
19    public int[]getParents() {
20        return parents;
21    }
22
23    public String toString() {
24        StringBuilder builder = new StringBuilder("Vertex\tDistance\tParent Vertex\n");
25        for (int i = 0; i < parents.length; i++) {
26            builder.append(i + "\t" + distances[i] + "\t" + parents[i] + "\n");
27        }
28        return builder.toString();
29    }
30
31    public TestResult equals(Result r) {
32        // Compare the arrays sizes
33        if(r.distances.length!=distances.length) {
34            return new TestResult("Difference taille des tableaux distances");
35        }
36        if(r.parents.length!=parents.length) {
37            return new TestResult("Difference taille des tableaux parents");
38        }
39        if(r.parents.length!=distances.length) {
40            return new TestResult("Difference entre les tailles des tableaux distances et parents");
41        }
42        // Compare the arrays
43        for(int i=0;i<r.parents.length;i++) {
44            if(r.distances[i] != distances[i]) {
45                return new TestResult("Difference à l'indice : "+i+" result.distances="+r.distances[i]+" et expectedResult.distances="+distances[i]);
46            }
47            if(r.parents[i]!=parents[i]) {
48                return new TestResult("Difference à l'indice : "+i+" result.parents="+r.parents[i]+" et expectedResult.parents="+parents[i]);
49            }
50        }
51        return new TestResult();
52    }
53
54 }
55 }
```

The screenshot shows the Eclipse IDE interface with the title "GitHub - dijkstra/src/utilities/Convertor.java - Eclipse IDE". The code editor displays the `Convertor.java` class. The class contains two static methods: `longToInt` and `doubleToFloat`. The `longToInt` method converts a `long[]` parents array into an `int[]` array. The `doubleToFloat` method converts a `double[]` distances array into a `float[]` array. Both methods iterate through the input arrays and store the values in the output arrays.

```
1 package utilities;
2
3 public class Convertor {
4
5     /**
6      * Convert a Long[] to a Int[]
7      * @param parents
8      * @return
9     */
10    public static int[] longToInt(long[] parents) {
11        int size = parents.length;
12        int i = 0;
13        int[] iTab = new int[size];
14        for(Long l : parents) {
15            iTab[i++] = l.intValue();
16        }
17        return iTab;
18    }
19
20
21    /**
22     * Convert a double[] to a float[]
23     * @param fTab
24     * @return
25     */
26    public static float[] doubleToFloat(double[] distances) {
27        int size = distances.length;
28        int i = 0;
29        float[] fTab = new float[size];
30        for(Double d : distances) {
31            fTab[i++] = d.floatValue();
32        }
33        return fTab;
34    }
35
36 }
```

Question 3

Dans la version originale de l'implémentation, qui nous est donnée dans le sujet, n'importe quelle suite de test passait par toutes les instructions, à l'exception de certains fonctions utilitaires qui ne sont jamais appelées.

Cependant, dans notre version (qui ajoute un système de lecture d'entrée sous forme de fichier JSON, et la comparaison de la sortie avec la sortie attendue, ce qui permet l'automatisation des tests, là où avec l'implémentatation initiale les données d'entrée devaient se trouver dans le code directement), Certaines instructions ne pourront pas être exécutées, notamment celles liées au contrôle d'erreur : en effet, on trouve beaucoup de conditions de la forme "si erreur, afficher quelque chose, et terminer le programme" : ainsi, s'il n'y a pas d'erreur les instructions d'affichage ne sont pas couvertes, s'il y a erreur le reste du programme n'est pas exécuté.

Autre exemple du même type, toutes les séquences de lecture/écriture de fichier impliquent l'utilisation d'un bloc try/catch :

- si une exception est levée avant la fin du bloc try, celui-ci est interrompu, les dernières instructions ne sont donc pas couvertes
 - si aucune exception n'est levée, le bloc catch n'est jamais exécuté
- Par conséquent, tout bloc try/catch implique qu'une certaine portion de code ne sera pas exécutée.

Globalement il est impossible de passer par toutes les instructions du programme, cependant les instructions servant à réaliser l'algorithme en lui-même peuvent être toutes traversées par plus ou moins n'importe quelle suite de tests.

Question 4

Préambule

Java restreint fortement les possibilités dans les tests, car les types sont static et fort, ainsi il n'est pas possible de donner un mauvais type ou une valeur trop grande.

Alors nous ne pourront pas tester les classes invalides, ni un nombre de sommet inférieure ou égale à zéro. Mais nous allons testé les poids (négatifs, nul, positif) et si un graphe est orienté ou non.

Partition des domaines des entrées

Variable	poid<0.0	poid=0	poid>0.0	poid in [-inf,inf]
Orienté	Test 2	Test 1	Test4	
Non orienté	Test 3		Test 5	

(cf. voir tests.json et results.json)

Question 5

Préambule

Il est difficile de trouver des mutants très pertinent vu la simplicité du code et le faible nombre d'instructions.

Premier mutant :

```
// TestGraph.java ligne 71
// ">" devient ">="
if (v.getDistance() >= u.getDistance() + e.getWeight()) {
    v.setDistance(u.getDistance() + e.getWeight());
    v.setParent(u);
    parents[v.getId()] = v.getParent().getId();
}
```

Ce mutant est choisi, car il est souvent compliqué de choisir entre "supérieure >" et "supérieure ou égale >=".

```
[V] - Test : 0
[X] - Test : 1
|---> Difference à l'indice : 2 result.parents=1 et expectedResult.parents=0
[X] - Test : 2
|---> Difference à l'indice : 3 result.parents=1 et expectedResult.parents=2
[X] - Test : 3
|---> Difference à l'indice : 6 result.parents=7 et expectedResult.parents=5
[X] - Test : 4
|---> Difference à l'indice : 5 result.parents=4 et expectedResult.parents=-1
Resultat : [1/5]
```

Seul le test zéro passe, c'est normal, car tous les sommets ont des valeurs différentes et aucune somme ne donne le même résultat.

Deuxième mutant :

```
// TestGraph.java ligne 67
// "!=" devient "=="
```



```
// check if vertex was visited already
if (e.getEndPoint() == v.getId()) {
    continue;
}
```

Ce mutant est choisi, car il est fréquent de se tromper dans les opérations logiques.

```
[X] - Test : 0
|---> Difference à l'indice : 1 result.distances=7.5 et expectedResult.distances=5.2
[X] - Test : 1
|---> Difference à l'indice : 1 result.distances=-4.3 et expectedResult.distances=-2.1
[X] - Test : 2
|---> Difference à l'indice : 3 result.parents=0 et expectedResult.parents=2
[X] - Test : 3
|---> Difference à l'indice : 1 result.distances=-8.0 et expectedResult.distances=2.0
[X] - Test : 4
|---> Difference à l'indice : 1 result.distances=-3.0 et expectedResult.distances=1.0
Resultat : [0/5]
```

Aucun test ne passe, ce qui est aussi normal. Ici on ne calcule plus le plus court chemin, car on peut passer dans des boucles.

Troisième mutant :

```
// TestGraph.java ligne 72
// le "+" devient "-"
v.setDistance(u.getDistance() - e.getWeight());

if (v.getDistance() > u.getDistance() + e.getWeight()) {
    v.setDistance(u.getDistance() - e.getWeight());
    v.setParent(u);
    parents[v.getId()] = v.getParent().getId();
}
```

Ce mutant est choisi, car il est intéressant de vérifier s'il n'y a pas une erreur dans l'implémentation.

```
[X] - Test : 0
|---> Difference à l'indice : 1 result.distances=-11.8 et expectedResult.distances=5.2
[X] - Test : 1
|---> Difference à l'indice : 1 result.distances=2.1 et expectedResult.distances=-2.1
[V] - Test : 2
[X] - Test : 3
|---> Difference à l'indice : 1 result.distances=-2.0 et expectedResult.distances=2.0
[X] - Test : 4
|---> Difference à l'indice : 1 result.distances=-1.0 et expectedResult.distances=1.0
Resultat : [1/5]
```

Seul le test deux est validé, car il n'y a que des zéros et zéro est le neutre de l'addition.

```
if (v.getDistance() > u.getDistance() + e.getWeight()) {
    v.setDistance(u.getDistance() + e.getWeight());
    v.setParent(v);
    parents[v.getId()] = v.getParent().getId();
}
```

Ce mutant est choisi, car les noms des variables sont mal choisies "u" et "v" sont deux lettres très ressemblante, il est facile de les confondre.

```
[X] - Test : 0
|---> Difference à l'indice : 1 result.parents=1 et expectedResult.parents=0
[X] - Test : 1
|---> Difference à l'indice : 1 result.parents=1 et expectedResult.parents=0
[X] - Test : 2
|---> Difference à l'indice : 1 result.parents=1 et expectedResult.parents=0
[X] - Test : 3
|---> Difference à l'indice : 1 result.parents=1 et expectedResult.parents=0
[X] - Test : 4
|---> Difference à l'indice : 1 result.parents=1 et expectedResult.parents=0
Resultat : [0/5]
```

Le mutant démontre qu'il n'y a pas eu d'erreur.