PORTFOLIO III

Implementation of Dijkstra Algorithm

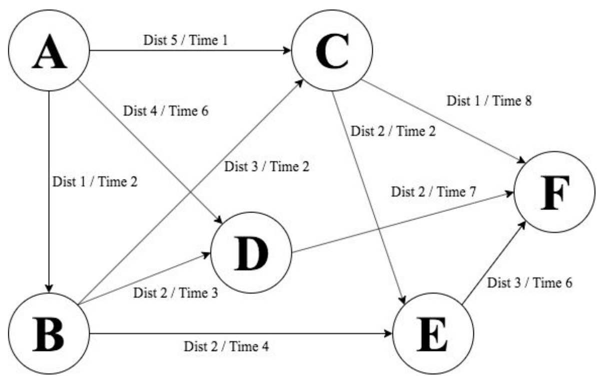
Essential Computing II
Autumn 2019

Teachers
Line Reinhardt
Henrik Bullskov

Group Members
Abdul Halim bin Abdul Rahman (63870)
Athanasios Kandylas (62658)
Dylan Rayner (62638)
Kristian Andreasen (63771)

Github code
https://github.com/tetrahydra/dijkstra

1. Graph visualisation, minimum distance calculations and flow diagram

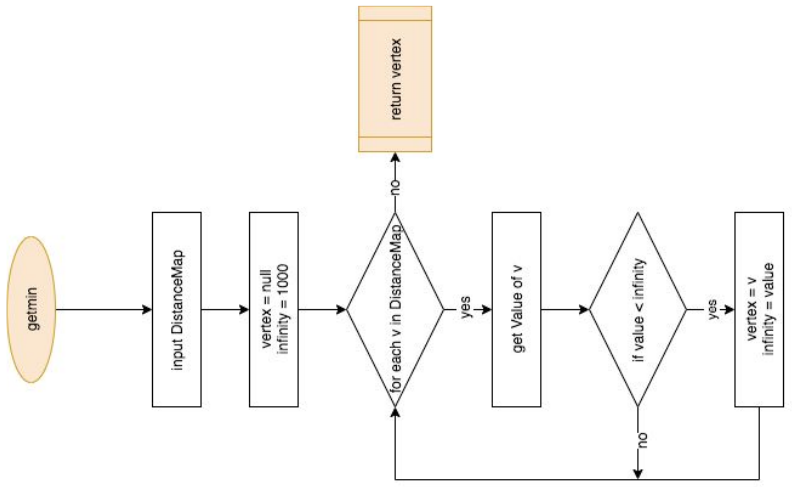


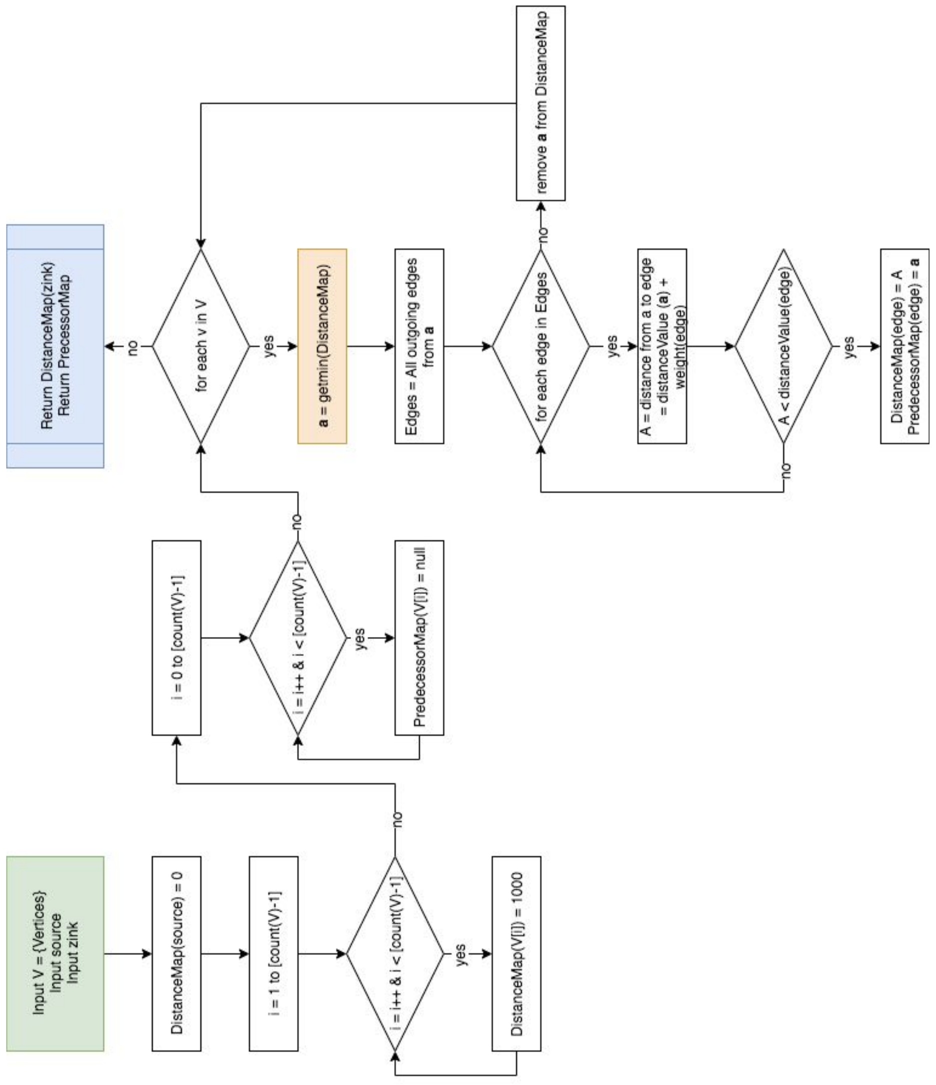
Travel possibilities for A to F

$A \rightarrow B \rightarrow E \rightarrow F$
$A \rightarrow B \rightarrow D \rightarrow F$
$A \rightarrow B \rightarrow C \rightarrow F$
$A \rightarrow B \rightarrow C \rightarrow E \rightarrow F$
$A \rightarrow C \rightarrow E \rightarrow F$
$A \rightarrow C \rightarrow F$
$A \rightarrow D \rightarrow F$

Iteration	Unvisited	Visited	Evaluation Vertex	A	B	C	D	E	F
1	A		A	0	A-1	A-5	A-4	X-∞	X-∞
2	B, C, D	A	B	0	A-1	B-4	B-3	B-3	X-∞
3	C, D, E	A, B	C	0	A-1	B-4	B-3	C-7	C-6
4	D, E, F	A, B, C	D	0	A-1	B-4	B-3	C-7	D-6
5	E, F	A, B, C, D	E	0	A-1	B-4	B-3	C-7	E-6
6	F	A,B,C,D,E	F	0	A-1	B-4	B-3	C-7	E-6
Final	-	ALL	NONE	0	A-1	B-4	B-3	C-7	E-6

This table is a representation of the Dijkstra algorithm. The vertex with the smallest value is chosen first (starting with the source $A = 0$) and the cost of its edges is calculated, and then the vertex value updated $(B = 1, C = 5, D = 4)$, since the number is smaller than ∞. Then, the smallest of all of the vertices not yet picked is selected and the distance cost of the edges to the connected vertices is once again updated if smaller than the previous value. The process is repeated until all vertices have been selected. The green value represents the shortest distance from the source A each vertex present in the graph. The notation $B-3$ means that vertex B is the immediate predecessor, with a total distance of 3 from vertex A. (*source: baeldung.com/java-dijkstra*).





2. Dijkstra implementation : distance and time test

```java
public Pair<Integer, Map<Vertex, Vertex>> ShortestDistance(Vertex source, Vertex zink) {
   Map<Vertex, Vertex> PredecessorMap = new HashMap<>();
   Map<Vertex, Integer> DistanceMap = new HashMap<>();
   Vertex a;

   // initialize arrays
   for (Vertex v : Vertices) {
       DistanceMap.put(v, 1000);
       PredecessorMap.put(v, null);
   }

   DistanceMap.put(source, 0);

   while (!DistanceMap.isEmpty()) {
       a = getmin(DistanceMap);

       if (a == null) {
           return (new Pair<Integer, Map<Vertex, Vertex>>(DistanceMap.get(zink),
                                                           PredecessorMap));
       }

       ArrayList<Edge> edges = a.getOutEdges();

       for (Edge e : edges) {
           if (DistanceMap.containsKey(e.getTovertex())) {
               Integer distance_edge = DistanceMap.get(a) + e.distance;
               if (distance_edge < DistanceMap.get(e.getTovertex())) {
                   DistanceMap.put(e.getTovertex(), distance_edge);
                   PredecessorMap.put(e.getTovertex(), a);
                   System.out.println(" Predecessor pairing : " +
                                      e.getTovertex().Name + ", " + a.Name);
               }
           }
       }
       DistanceMap.remove(a);
   }
   return (new Pair<Integer, Map<Vertex, Vertex>>(DistanceMap.get(zink), PredecessorMap));
}

public Vertex getmin(Map<Vertex, Integer> qmap) {
   Vertex vertex = null;
   int infinity_value = 1000;

   for (Map.Entry<Vertex, Integer> entry : qmap.entrySet()) {
       Vertex current_vertex_iteration = entry.getKey();
       int current_vertex_value = entry.getValue();
       if (current_vertex_value < infinity_value) {
           vertex = current_vertex_iteration;
           infinity_value = current_vertex_value;
       }
   }
   return vertex;
}
```

Output from terminal for the first given graph, from node A to node F.

```
Path (distance and time) analysis for smaller graph.

Predecessor pairing : B, A
Predecessor pairing : C, A
Predecessor pairing : D, A
Predecessor pairing : C, B
Predecessor pairing : D, B
Predecessor pairing : E, B
Predecessor pairing : F, D
Edges path (distance) : A →  B →  D →  F

Predecessor pairing : B, A
Predecessor pairing : C, A
Predecessor pairing : D, A
Predecessor pairing : F, C
Predecessor pairing : E, C
Predecessor pairing : D, B
Edges path (time)   : A →  C →  F
```

3. Discussion for Big O

For the `ShortestDistance` method, the code has a `for` loop function, iterating V, to assign values to `DistanceMap` and `PrecessorMap` and run only a single time, which produces $O(V)$.

Then it is followed by a `while` loop, looping through V only one time as well, which produces another $O(V)$. Then in the body of `while` loop, it has a call to a `getmin` method which has a `for` loop function iterating the V which produces another $O(V)$. Then followed by a `for` loop for check all the edges which produces $O(E)$. Since the inner $O(V)$ and $O(E)$ are independent from each other, then it can be written as $O(V+E)$. This notation resides in a `while` loop, then it can be written as $O(V(V+E))$.

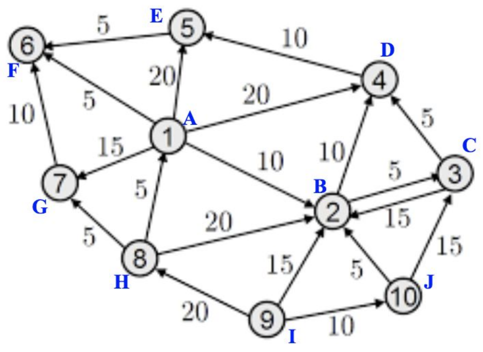
Since the first $O(V)$ is meant to assign values and `while` loop are independent from each $other, then they can both be written as $O(V+V(V+E))$.

$\rightarrow O(V+V(V+E))$
$\rightarrow O(V+V^2+VE)$

Then it can be simplified to, $O(V^2+VE)$.

Based on other references, the best worst-case scenario is $O(E+V\log V)$. But our result shows very poor performance.

4. Test on bigger graph



Output from the terminal for node **10** (denoted as **J**) to node **6** (denoted as **F**)

```
 Path (distance and time) analysis for bigger graph.

  Predecessor pairing : B, J
  Predecessor pairing : C, J
  Predecessor pairing : C, B
  Predecessor pairing : D, B
  Predecessor pairing : E, D
  Predecessor pairing : F, E
  Edges path (distance) : J →  B →  D →  E →  F
```

The possibilities from J to F are

J → B → D → E → F
= 0 → (0+5=5) → (5+10=15) → (15+10=25) → (25+5=30) = 30

J → C → D → E → F
= 0 → (0+15=15) → (15+5=20) → (20+10=30) → (30+5=35) = 35

The by-hand calculations can be compared to the output from the program. It can be seen that the computed value is equal to the first calculated value by hand.

5. Discussion for negative edge weights

In the Dijkstra algorithm, Vertex is selected in succession. If we take for instance the following example:

Vertices, $V = \{A, B, C, D\}$ with edges E$= \{(A, B, 5), (A, C, 10), (C, D, 1), (D, B, -10)\}$.

Except for source, A which has the value set to 0, all other vertices have their respective values set to ∞.

The algorithm will first select the node A with value 0, being the source vertex and should be lower than the others. Next it will look for the connected outgoing edges, $A \rightarrow B = 5$ and $A \rightarrow C = 10$. It chooses the edge with the lowest cost which is 5.

From vertex A to vertex B has a cost of 5. So the algorithm changed the vertex B value from ∞ to 5 and also the vertex C value from ∞ to 10. Then close the vertex B has the vertex A as its predecessor.

The process is repeated with outgoing edges from B. With no connection to another vertex, B will not update any other vertex value and the node will be closed.

Next, the vertex C is selected. From vertex C to vertex D has a cost of $10 + 1$, 10 is the cost previously associated to C. The vertex D is updated from ∞ to 11.

Finally, the vertex D will be selected and the only direction it will be able to go is to vertex B with a cost of -10. The value of vertex B should then be updated to 1, on the formulation $(11 - 10)$. BUT, this cannot happen because according to the Dijkstra algorithm, as the vertex B as already being visited and selected, its value cannot be updated.

One of the drawbacks of the Dijkstra algorithm is that it has the chances of not work with negative edge weights, but it will surely not work having negative weight cycles giving an incorrect result.

6. Extra discussion

It can be that from A to F, the path iteration can either take place as $A \rightarrow B \rightarrow E \rightarrow F$ or $A \rightarrow B \rightarrow D \rightarrow F$. With the first three vertices are producing similar values, $A(1) \rightarrow C(0+5=1) \rightarrow F(5+1=6)$ $A(1) \rightarrow B(0+1=1) \rightarrow E(1+2=3)$ and $A(1) \rightarrow B(0+1=1) \rightarrow D(1+2=3)$. How do we choose which one is the smallest now that D and E have similar values?

Similar situation can be seen for $A(1) \rightarrow B(0+1=1) \rightarrow D(1+2=3) \rightarrow F(3+3=6)$ and also . Both paths are the shortest but both are having similar values.

Travelling from A to F	*Distance*	*Time*
A → B → E → F	6	12
A → B → D → F	5	12
A → B → C → F	5	12
A → B → C → E → F	9	12
A → C → E → F	10	9
A → C → F	6	9
A → D → F	6	13

The table above summarizes each possible path combination. In red the best results for the shortest path show that the algorithm chooses different solution to the problem, in relation to distance and time from A to F. By assuming that the distance and time are equivalent to a specific cost, the best possible path would be one taking less time and less distance. In this case A → C → F would be the ideal choice, with the other options taking either too much time or being too long.