21/04/2017

Report Tutorial Course 4 (Part B)

Algorithmics and Advanced Programming

WU Zheng

Report Tutorial Course 4 (Part B)

Algorithmics and Advanced Programming

Dijkstra algorithm for weighted digraphs

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:             // Initialization
        dist[v] ← INFINITY                  // Unknown distance from source to v
        prev[v] ← UNDEFINED                 // Previous node in optimal path from source
        add v to Q                          // All nodes initially in Q (unvisited nodes)

    dist[source] ← 0                        // Distance from source to source

    while Q is not empty:
        u ← vertex in Q with min dist[u]    // Node with the least distance will be selected first
        remove u from Q

        for each neighbor v of u:           // where v is still in Q.
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:               // A shorter path to v has been found
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

Create a class called DijkstraSP. This class will implement the Dijkstra algorithm for detecting shortest paths in weighted-digraphs. This class will contain the following functions:

1. 3 arrays: **boolean**[] marked, **int**[] previous and **int**[] distance as for the unweighted graphs.
2. A function called verifyNonNegative(*WDGraph* G) which takes as input a weidhted-directed graph and verifies than all weights in the graph are non negative.
3. Create a function called DijkstraSP(*WDgraph* G, **int** s)which implements the Dijkstra algorithm for shortest paths studied in lecture 4. The input arguments are a weighted-digraph and a root vertex s.
4. As for the previous section, create the functions hasPathTo(**int** v), distTo(**int** v) and printSP(**int** v).

```
import java.util.*;

public class DijkstraSP {

    private int sourceNode;
    private boolean[] marked;
    private int[] previous;
    private double[] distance;

    private boolean verifyNonNegative(WDGraph G) {
        for (LinkedList<WDGraph.DirectedEdge> edges: G.adj) {
            for (WDGraph.DirectedEdge n : edges) {
                if (n.weight() <= 0) {
                    return false;
                }
            }
        }
        return true;
    }

    public ArrayList<Integer> DijkstraSP(WDGraph G, int s) {
        // To ensure all weights of edges are positive.
        if (!verifyNonNegative(G)) {
            return null;
        }
        sourceNode = s;
        int v = G.V + 1;
        marked = new boolean[v];
        previous = new int[v];
        distance = new double[v];
        // Use an array to store unvisited nodes
        HashSet<Integer> openedNodes = new HashSet<>();
        // Open all nodes
        for (int i = 0; i < v; i++) {
            previous[i] = -1; // UNDEFINED
            distance[i] = Double.MAX_VALUE; // +INFINITY
            openedNodes.add(i);
        }
        // Use an array list to record visit orders.
        ArrayList<Integer> visitOrder = new ArrayList<>();
        // Distance from source to source
        distance[s] = 0; // distance
        marked[s] = true; // mark
        visitOrder.add(s); // visit
        while (!openedNodes.isEmpty()) {
            // Choose the smallest distance.
            double smallestDistance = Double.MAX_VALUE;
            int smallestNode = -1;
            for (Integer thisNode : openedNodes) {
                if (distance[thisNode] < smallestDistance) {
                    smallestDistance = distance[thisNode];
```

```
                    smallestNode = thisNode;
                }
            }
            // Go to the smallest one.
            openedNodes.remove(smallestNode);
            visitOrder.add(smallestNode);
            // If remained nodes are not available, it is not a connected graph,
terminate the progress.
            if (smallestNode == -1) {
                break;
            }
            // Check all neighbours and update distances
            for (WDGraph.DirectedEdge directedEdge : G.adj[smallestNode]) {
                int childNode = directedEdge.to();
                double alt = distance[smallestNode] + directedEdge.weight();
                if (alt < distance[childNode]) {
                    marked[childNode] = true;
                    previous[childNode] = smallestNode;
                    distance[childNode] = alt;
                }
            }
        }
        return visitOrder;
    }

    public boolean hasPathTo(int v) {
        return marked[v];
    }

    public double distTo(int v) {
        return distance[v];
    }

    public void printSP(int v) {
        ArrayList<Integer> shortestPath = new ArrayList<>();
        int thisNode = v;
        while (thisNode > -1) {
            shortestPath.add(thisNode);
            thisNode = previous[thisNode];
            if (thisNode == sourceNode) {
                shortestPath.add(sourceNode);
                break;
            }
        }
        Collections.reverse(shortestPath);
        System.out.println(shortestPath);
    }

}
```

Test the previous functions with the graph graph-WDG.txt. The result of this test is:

```
0:
1: (2, 9.0), (6, 14.0), (7, 15.0),
2: (3, 24.0),
3: (5, 2.0), (8, 19.0),
4: (3, 6.0), (8, 6.0),
5: (4, 11.0), (8, 16.0),
6: (3, 18.0), (5, 30.0), (7, 5.0),
7: (5, 20.0), (8, 44.0),
```

```
8:
true
50.0
[1, 6, 3, 5, 8]
```