ISEP
Algorithmics and Advanced programming
March 2019

# Tutorial course 3:
# Algorithmics for traversing a graph and shortest paths

**Instructions:**

- Prepare a ZIP archive containing your source code in a "src" folder and written report in a pdffile and upload it to moodle into the folder corresponding to your group (tuesday orwednesday)

In this TP, you have 3 choices. You can select to do Parts A-E or E+G or E+F or H.

## Objectives

- Getting familiar with BFS and DFS algorithms.
- Getting familiar with weighted and directed graphs.
- Use BFS algorithm for calculating shortest paths in unweighted graphs.
- Use Dijkstra algorithm for calculating shortest paths in weighted digraphs.

## The Graph Class

Intstructions and source code of the Graph Class are available on Moodle in preparation of this tutorial class. You should have read it !

## A. The Depth Search First algorithm (DFS)

Depth-first search (DFS) is a classic method for systematically examining each of the vertices and edges in a graph. The aim of DFS algorithm is to traverse by going through the depth of the graph from the starting node.

1. Given a graph, create a function called `List<V> dfs(Graph G)` that performs the deep first search (DFS) algorithm for visiting the graph G. This function must return the list of vertices in the order of their first encounter. In case of choice, the vertex with the smallest identifier will be chosen.
2. One important application of the depth first search algorithm is to find the **connected components** of a graph. Write a function *int cc(Graph G)* that takes as input a simple graph and determines the number of connected components. The function *int cc(Graph G)* must use the DFS algorithm. Write a function called `isConnected`() that returns true if the graph is connected, false otherwise.

3. Test the `dfs(.)` and *cc(.)* functions with the graph is the *graph-DFS-BFS.txt* file. Consider as starting node the node 5. What is the order of the first encounter of the nodes? How many components does the graph have? Is it connected?

## B. Breadth Search First algorithm (BFS)

Breadth-first search (BFS) is a classic method for systematically examining each of the vertices and edges in a graph. Giving the source node, unlike DFS, BFS explores the neighbor nodes first, before moving to the next level neighbors.

1. Given a graph, create a function called *`bfs(Graph G)`* that performs the breadth first search (BFS) algorithm for visiting the graph G. This function must return the list of vertices in the order of their first encounter. In case of choice, the vertex with the smallest identifier will be chosen.
2. Test the *`bfs(.)`* function with the graph is the *graph-DFS-BFS.txt* file. Consider as starting node the node 5. What is the order of the first encounter of the nodes? How many components does the graph have? Is it connected?

## C. Breadth Search First (BFS) for shortest paths in unweighted (di)graphs

BFS allows to find shortest paths from a single source vertex *v* to all other vertices in an unweighted (di)graph. BFS considers:

1. First all direct neighbors of v
2. Second neighbors of neighbors of v
3. Etc.

Create a class called *BFSShortestPaths*. This class will implement the BFS algorithm for shortest paths from a given vertex `s` (seen in lecture 3). This class will contain:

1. 3 vertex-indexed arrays: `boolean[] marked`, `int[] previous` and `int[] distance`. The entry `marked[v]` is set to true if `v` has been visited (false otherwise). `previous[v]` indicates the vertex preceding `v` on the shortest path. `distance[v]` represents the distance (in number of edges) from the source vertex `s` to vertex `v`.
2. Modify the function `bfs(graph G)` called `bfs(Digraph G, int s)` which executes the BFS algorithm to calculate all the shortest paths from the root vertex `s`. This function will update the `marked`, `previous` and `distance` arrays. It will be of `void` type.
3. Add a boolean function `hasPathTo(int v)` which returns true if there is a path from `s` to `v`.
4. Add the function `distTo(int v)` which returns the length of the shortest path from `s` to `v`.
5. Add the function `printSP(int v)` which prints the shortest path from `s` to `v`.
6. Test the previous functions with the graph *graph-BFS-SP.txt* . Find all the shortest paths and deduce the excentricity of each vertex, the diameter and the radius of the graph.

## D. Manipulating Weighted digraphs

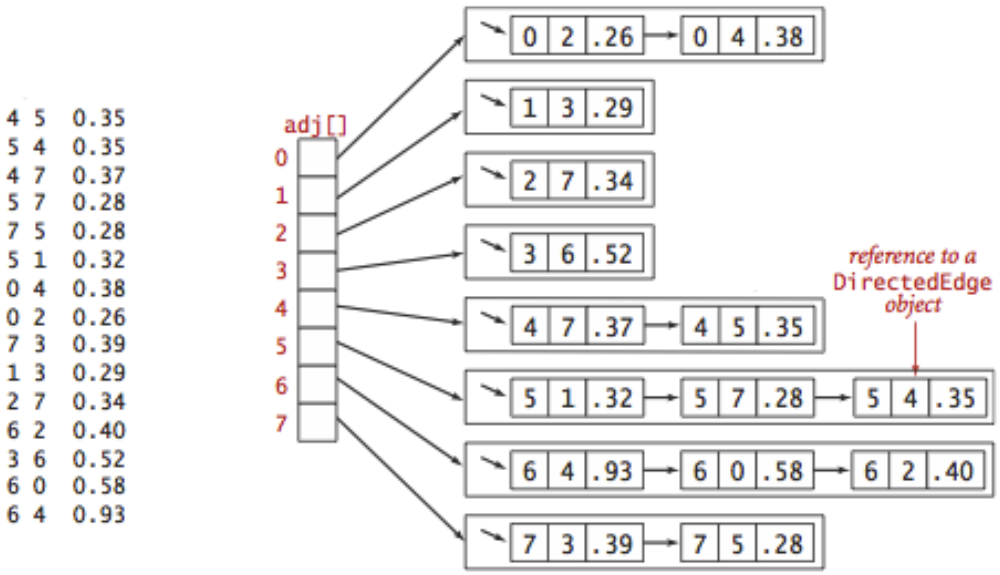
We will consider the adjacency list representation for unweighted and weighted digraphs.

Create a class called `DirectedEdge` containing three attributes as follows:

```
public class DirectedEdge {
    private final int v;
    private final int w;
    private final double weight;

…

public int from() {
        return v;
    }

public int to() {
        return w;
    }

public double weight() {
        return weight;
    }

}
```

where v represents the source vertex, w the destination vertex and weight the edge-weight. The functions **`from()`**, **`to()`** and **`weight()`** constitute the getter functions for the source node, the destination node and the edge-weight respectively.

Create a class called *`WDgraph`* to represent weighted-digraphs. This class will use an adjacency list representation of the graph, where the entry corresponding to vertex v will contain the list of all the outgoing arcs and the associated weights. In other words, that will be a list of objects of type `DirectedEdge`. As in the following example (taken from http://algs4.cs.princeton.edu/44sp/ ):




Adjacency list representation of a weighted digraph

Test the functions of this class on the graph *graph-WDG.txt.* The input file lines:

```
1 2 9
1 6 14
1 7 15
```

must be interpreted as follows:

- The first line indicates that there is an arc from vertex 1 to vertex 2 and its weight is 9
- The second line implies that there is an arc from vertex 1 to vertex 6 and its weight is 14
- … and so on.

## E. Dijkstra algorithm for weighted digraphs

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 3. 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)`.

Test the previous functions with the graph *graph-WDG.txt*.

## F. Faster Implementation of Dijkestra

Propose a faster implementation for Dijkestra and compare it with the original suggested in E.

## G. Dijkstra for longest path tree from single source

How we can use Dijkstra for finding the longest path tree from single source

## H. Added value for Google Map or SNCF App

If you have an idea that SNCF or Google Map have not considered in providing the shortest paths, you can start implementing it (Make sure that it has not been considered with other apps in the demain).

## I. Junit test for your Dijkstra