PRACTICAL WORK NO. 2

# Ciama Andreea CiamaGroup: 911, year 1

# Practical work no. 2

This project represents the implementation of a *directed graph* for which we can:

* parse its vertices
* add and remove vertices and their incident edges
* add and remove edges
* get and set costs of edges
* get in- and out-degrees of vertices
* get a clone/deepcopy of the graph
* we can read and write the file to a graph

Specification

We define a class named Graph representing a directed graph. The class Graph will provide the following methods:

* init (self, n=0, edges=[], costs=[])
  + Constructs a graph with n vertices, with the given edges and costs
* parse\_Vertices(self)
  + Returns the copy of the list of vertices of the graph
* Nr\_of\_vertices(self)
  + Returns the nr of vertices of the graph
* Nr\_of\_edges(self)
  + Returns the nr of edges of the graph
* is\_Edge(self, s, d)
  + Returns True if there is an edge between s and d and False otherwise
* Parse\_nin (self, x)
  + Returns a list of the vertices by parsing inside the graph from vert X
* Parse\_nout(self, x)
  + Returns a list of the vertices by parsing outside the graph from vert X
* Modify\_Value(self, x,y,val)
  + Modifies the value of the edge (x,y) and puts the value VAL inside the dic
* Retrieve\_Value(self, x,y,val)
  + Retrieves the value of the edge (x,y) and puts the value VAL inside the dic
* Copy\_Graph(self,)
  + Makes a copy of the graph and returns it
* Add\_Vertex(self, x)
  + Adds vertex x to the end of the graph.
* remove\_Vertex(self, x)
  + Removes vertex x from the graph and from the dictionary of values
* Add\_Edge(self, v1,v2)
  + Adds the edge (v1, v2) with a cost of 0 to the graph
  + :param v1: integer, source of the edge
  + :param v2: integer, destination (direction) of the edge
* Delete\_Edge(self, v1, v2)
  + Removes edge (v1, v2) from the graph
  + :param v1: integer, source of the edge
  + :param v2: integer, destination (direction) of the edge
* getCopy(self)
  + Returns a deepcopy of the graph of type DirGraph
* From\_File(self, filename)
  + Reads a graph from the given file.
  + :param filename: name of the file which contains the graph

Text file format:

On the first line, the number n of vertices and the number m of edges;

On each of the following m lines, three numbers, x, y and c, describing an edge: the origin, the target and the cost of that edge.

* Into\_File(self, filename)
  + Writes a graph to the given file.
  + :param filename: name of the file which contains the graph
* def shortest\_path(self, start\_vertex, end\_vertex):
  + finds a lowest length path between the 2 vertices using a breadth-first search from the starting vertex.
  + param: start\_vertex, end\_vertex: the 2 vertices
  + The algorith below visits all the vertices that are accessible from the start vertex. They are visited in the order of increasing distances from the starting vertex. A parent vector or map is computed, allowing us to compute the minimum length path from the starting vertex to any choosen accessible vertex.

We also define a function *randomGraph(n, m)* which generates a random graph of type DirGraph with n vertices and m edges

The class UI controls the user-interface part and provides the menu of the application.

## Implementation

I used a class DirGraph to model the directed graph. Its fields memorize the in- and out- neighbours of each vertex and also the cost of each edge in dictionaries.

* Each vertex will have an ID (from 0 to n-1) and will have a corresponding list of inbound/outbound neighbours in both in\_neighbours and out\_neighbours.
* Costs of edges will be kept in costs, in *(src, dest):cost* format.
* The graph admits edges from self to self and also admits negative costs.

A range of public methods (fully specified in the actual implementation) model the behaviour of a directed graph.

The class uses no sub- or helper-classes.