Specification

We shall define a class named **Graph** representing a *directed graph*.

We need tree auxiliary classes:

- **Console**, a class that communicate with the user and handle all the commands
- **RandomGraph** ->generates a random graph with a given number of vertices and edges and it prints it
- **MyException** -> raises exceptions

The class **Graph** will provide the following methods:

def parseX(self)

Returns *a copy of all the vertex keys*

def parse_iterable_in(self, x)

Returns a list of in neighbours of x

def parse_iterable_out(self, x)

Returns a list of all out neighbours of x

def get_number_of_vertices(self)
Returns the number of vertices of the graph

def get_number_of_edges(self)

Returns the number of edges

def is_edge(self, x, y)
Returns true if there is an edge from x to y, false otherwise

def add_edge(self, x, y, cost)
Adds an edge (x, y) having the cost, 'cost' to the graph
precondition: the edge must not exist in the graph and the vertices must be valid; in case we already have that edge in the graph, or the vertices are not valid the error is handled and the user is informed

def in_degree(self, vertex)
Returns the in degree of a given vertex
precondition: x needs to be a valid vertex in the graph, in case it isn't, the error is handled and the user is informed

def out_degree(self, vertex)
Returns the out degree of a given vertex
precondition: x needs to be a valid vertex in the graph, in case it isn't, the error is handled and the user is informed

def change_cost(self, x, y, value)

modify the cost of a given edge

precondition: the edge must exist, otherwise errors are handled

def retrieve_cost(self, x, y)

Returns the cost of the edge (x, y)

precondition: (x, y) must exist, if it doesn't errors are handled and the user is informed

def remove_edge(self, x, y)

remove the edge (x,y)

precondition: (x,y) needs to be a valid edge in the graph, if it isn't, the error is handled and the user is informed

def add_vertex(self)

add a new vertex to the graph

def remove_vertex(self, vertex)

remove a given vertex

precondition: the vertex must exist in the graph, it it doesn't, the error is handled and the user is informed

def isolated_vertices(self)

Returns a list with isolates vertices

def copy_graph(self)

Returns a copy of the graph

The class **Console** reads the inputs from the user, communicate with the class Graph, solve the user commands and handles all the input errors, providing a message.

The class **RandomGraph** provides the following methods:

def random_graph(self, x, y)

Generates a random graph having x vertices and y edges; if it is impossible an error is raised.

def store(self)

Stores the graph into a given file, passed as a parameter to the class; if the file cannot be opened an error is raised

def print_graph(self)

Prints the graph

This class has 3 parameters:

- **x** – the number of vertices
- **y** - the number of edges
- **fileName** – the name of the file

Implementation

The implementation uses 3 dictionaries

- dictin – has as keys the vertices, and as values the list of predecessors
- dictout - has as keys the vertices, and as values the list of successors
- costs – has as keys the edges, and as values the costs

Each vertex belongs to 2 dictionaries dictin and dictout as keys and each edge belongs to 1 dictionary, costs, as a key

Class Graph reads from a file, which is a parameter, a given graph and if the given file is modified it will be saved in another file, the second parameter of the class.