**Practical Work No. 1**

**Specification**

We shall define three classes. One called Graph which will be represented by the list of vertices and the number of edges, one called DirectedGraph which will represent the directed graph itself and one called Console, which will represent the user interface console.

There is also implemented the class called GraphException which is used exclusively for throwing the errors specified to the graph and catching them.

There also exists an auxiliary class called Validation which is used for validating the the vertices

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

**def initialize\_vertices(self, number\_of\_vertices):**

With this function we initialize the vertices with the corresponding ones.

**def initialize\_edges(self, number\_of\_edges):**

With this function we set the number of edges with the corresponding value.

**def add\_vertex(self, new\_vertex):**

With this function we add a vertex. **Precondition:** the vertex must not already exit.

**def remove\_vertex(self, vertex):**

With this function we remove a vertex. **Precondition**: the vertex must already exist.

**def find\_vertex(self, vertex):**

Returns true if the given vertex exists or false otherwise

**@property  
def edges(self):**

This function represents a getter and returns the number of edges.

**@edges.setter  
def edges(self, new\_value):**

This function represents a setter and with it we can change the number of edges.

**@property  
def vertices(self):**

This function represents a getter and returns the vertices.

**def parse\_vertices(self):**

Returns the vertices as a generator.

**def check\_existence\_of\_vertex(self, check\_vertex):**

This function returns true if the given vertex already exists and false otherwise.

**def \_\_len\_\_(self):**

Returns the number of vertices

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

**@property  
def inbounds(self):**

This function represents a getter and returns the inbounds.

**@inbounds.setter  
def inbounds(self, key, value):**

This function represents a setter and with it we can change the list of vertices attached to a specific vertex.

**@property**

**def outbounds(self):**

This function represents a getter and returns the outbounds.

**@outbounds.setter**

**def outbounds(self, key, value):**

This function represents a setter and with it we can change the list of vertices attached to a specific vertex.

**@property**

**def cost(self):**

This function represents a getter and returns the costs of the graph.

**@cost.setter**

**def cost(self, key, value):**

This function represents a setter and with it we can change the cost of an edge.

**def copy\_graph(self):**This function returns a copy of the graph.

**def initialize\_graph(self, number\_of\_vertices):**

With this function we initialize the graph.

**def initialize\_costs(self):**  
With this function we initialize the costs of the graph.

**def initialize\_vertices\_in(self):**  
With this function we initialize the inbounds of the graph.

**def initialize\_vertices\_out(self):**  
With this function we initialize the outbounds of the graph.

**def get\_inbound\_vertices(self, vertex):**  
This function returns the list of inbound vertices for a specified vertex.  
  
**def get\_outbound\_vertices(self, vertex):**  
This function returns the list of outbound vertices for a specified vertex.

**def add\_vertex(self, new\_vertex):**With this function we add a vertex to the graph.  
  
**def remove\_vertex(self, vertex):**

With this function we remove a vertex from the graph.

**def add\_edge(self, first\_vertex, second\_vertex, cost):**With this function we add an edge to the graph. **Preconditions:** first \_vertex and second\_vertex must already exist. And there should not exist an edge from the first\_vertex to the second\_vertex.

**def delete\_edge(self, first\_vertex, second\_vertex):**With this function we delete and edge from the graph. **Preconditions:** first\_vertex and second\_vertex must already exist. And there should already exist an edge from the first\_vertex to the second\_vertex.

**def add\_cost(self, first\_edge, second\_edge, cost):**With this function we set the cost for a specified edge.  
  
**def add\_vertex\_to\_inbounds(self, first\_vertex, second\_vertex):**  
With this function we add the first vertex as being an inbound vertex for the second one.

**def add\_vertex\_to\_outbounds(self, first\_vertex, second\_vertex):**With this function we add the second vertexas being an outbound vertex for the second one.

**def parse\_outbound\_vertices(self, vertex):**

Returns the outbounds as a generator.

**def parse\_inbound\_vertices(self, vertex):**Returns the inbounds as a generator.  
  
**def get\_in\_degree(self, vertex):**Returns the indegree of a vertex.

**def get\_cost(self):**

Returns the costs as a generator.

**def get\_out\_degree(self, vertex):**Returns the outdegree of a vertex.

**def check\_existence\_edge(self, first\_edge, second\_edge):**Returns true if the edge already exists and false otherwise.

**def change\_cost(self, key, cost):**With this function we change the cost of an edge.

**def change\_cost\_edge(self, first\_vertex, second\_vertex, cost):**With this function we change the cost of an edge.

**def get\_cost\_for\_edge(self, first\_vertex, second\_vertex):**

With this function we get the cost for a specific edge

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

**def display\_number\_of\_vertices(self):**

With this function we display the number of vertices.

**def display\_the\_vertices(self):**This function display all the vertices.

**def display\_inbound\_vertices(self):**This function display the inbound vertices of a specified vertex.

**def change\_cost\_edge(self):**This function change the cost of an edge.

**def display\_outbound\_vertices(self):**This function display the outbound vertices of a specified vertex.

**def get\_in\_degree(self):**This function display the indegree of a vertex.

**def get\_out\_degree(self):**This function display the outdegree of a vertex.

**def add\_vertex(self):**This function adds a vertex to the graph.

**def remove\_vertex(self):**This function removes an vertex.

**def add\_an\_edge(self):**This function adds a vertex.

**def delete\_an\_edge(self):**

This function removes a vertex.  
**def copy\_graph(self):**This function creates a copy of the graph.

**def read\_graph(self):**This function reads the graph from a file.

**def display\_graph(self):**This frunction display the graph into a file.

**def generate\_a\_random\_graph(self):**This function generate a random graph.

**def display\_menu(self):**This function displays the available commands.

**def get\_cost(self):**

This function gets the cost for a specified edge.

**def start(self):**

This function starts the execution of the program.

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

**def exist\_vertex(self, first\_vertex, second\_vertex**):

This function raise a GraphException if one of the vertices doesn’t exist.

**There are also 5 external functions.**

**def read\_graph\_from\_file(file\_name, graph, service):**

This function reads the graph from a file.

**def write\_graph\_to\_file(file\_name, graph, service):**

This function writes the graph to a file without marking the isolated points

**def generate\_random\_graph(graph, directed\_graph, number\_of\_vertices, number\_of\_edges):**

This function generates a random graph.

**def create\_text\_file(file\_name):**

This function creates a text file.

**def write\_graph\_to\_file2(file\_name, graph, service):**

This function writes the graph to a file marking the isolated points.

**Implementation**

Like I said before, the graph is implemented using 2 principal classes, whose functions work directly on the graph, one class which represents the user interface console and one class for validating the data.

***The Class Graph will have the following data members:***

**self.\_vertices = []**

Represents the vertices

**self.\_edges = 0**

Represents the number of edges

***The Class DirectedGraph will have the following data members:***

**self.\_graph = graph**

Which represents the object of type Graph

**self.\_inbounds = {}**  
In order to store for each vertex the inbound ones we used a dictionary

(For example: 1<-2, 1<-3, for the vertex 1 we will have the list of vertices 2, 3)

The vertex will represent the key and the value will be represented by the corresponding vertices

**self.\_outbounds = {}**

In order to store for each vertex the outbound ones we used a dictionary

(For example: 2->1, 2->3, for the vertex 2 we will have the list of vertices 1, 3)

The vertex will represent the key and the value will be represented by the corresponding vertices

**self.\_cost = {}**

In order to store the cost for each edge we used a dictionary, where the key is represented by the edge and the value represents the cost

***The Class Console will have the following data members:***

**self.\_graph = graph**

Which represents the graph which contains the list of vertices and the number of edges

**self.\_service = service**

Represents the DirectedGraph which contains the inbounds, outbounds and the costs

**self.\_validation = validation**

Represents the object for the class validation

***The Class Validation will have the following data members:***

**self.\_graph = graph**

Which represents the graph which contains the list of vertices and the number of edges

**self.\_service = service**

Represents the DirectedGraph which contains the inbounds, outbounds and the costs