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