Specification

-Python Project – Laboratory Work 1-

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

It's **__init__** procedure depends on the number of arguments received, number_of_vertices and number_of_edges:

- number_of_vertices = number_of_edges = 0: an empty graph is generated
- number_of_vertices > 0 and number_of_edges = 0: a graph with number_of_vertices vertices and no edges is generated
- number_of_vertices > 0 and number_of_edges > 0: a graph with number_of_vertices vertices and number_of_edges edges is generated

self._vertices = a set containing the graph's vertices

self._outbound_neighbours = a dictionary in which the key-values pairs represent the vertices and its outbounded neighbouring vertices

self._inbound_neighbours = a dictionary in which the key-values pairs represent the vertices and its inbounded neighbouring vertices

self._cost = a dictionary in which the key[0]-key[1]-values triples represent the delimiting vertices of an edge and its cost

The class Graph will provide the following methods:

is_edge(self : Graph, first_vertex: integer, second_vertex: integer) = returns True if the vertices are the endpoints of an edge and False otherwise

inbound_degree_of_given_vertex(self, vertex) = returns the number of inbound vertices of a given vertex by returning the length of the _inbound_neighbours dictionary

outbound_degree_of_given_vertex(self, vertex) = returns the number of outbound vertices of a given vertex by returning the length of the _outbound_neighbours dictionary

get_cost_of_edge(self, first_vertex, second_vertex) = returns the cost of an edge given by its end-point vertices by accessing the dictionary element with key[0] and key[1] corresponding to the vertices

set_cost_of_edge(self, first_vertex, second_vertex, new_cost) = sets the cost of an edge given by its end-point vertices by accessing the dictionary element with key[0] and key[1] corresponding to the vertices

count_number_of_vertices(self) = returns the number of vertices of the graph by computing the length of the _vertices set

count_number_of_edges(self) = returns the number of edges of the graph by computing the length of the _cost dictionary (it contains the edge vertices and the cost of each edge)

add_vertex(self, new_vertex) = adds in the graph a valid vertex (no overlapping)

add_vertex_isolate(self, new_vertex) = adds in the graph a valid vertex (no overlapping), not having a predefined vertex list already as in the method **add_vertex**, but checks during the reading process if it exists or not. Used only when reading from a file a graph with isolated vertices.

add_edge(self, first_vertex, second_vertex, new_edge_cost = 0) = adds in the graph a new edge (its vertices have to exist

add_edge_isolate(self, first_vertex, second_vertex, new_edge_cost = 0) = adds in the graph a new edge (its vertices don't have to exist but are added during the reading process from the file). It allows isolated vertices to exist.

remove_edge(self, first_vertex, second_vertex) = removes an existing edge from the graph, and the cost of that edge

remove_vertex(self, vertex_to_delete) = removes an existing vertex from the graph, all its outbound vertices from the _outbound_neighbours dictionary, all its appearances in the _inbound_neighbours dictionary, and the costs of the edges in which it was an end-point vertex

vertices_iteraror(self) = returns an iterable strucuture containing all the vertices in the _vertices set

outbound_vertices_iterator(self, vertex) = returns an iterable structure containing all the vertices in the _outbound_neighbours dictionary

inbound_vertices_iteraor(self, vertex) = returns an iterable structure containing all the vertices in the _inbound_neighbours dictionary

edges_iterator(self) = returns an iterable structure containing all the edges adn their costs in the _cost dictionary

make_copy_of_current_graph(self) = creates a Graph object which is the copy of the current graph, being separate from the graph that the program is currently editing

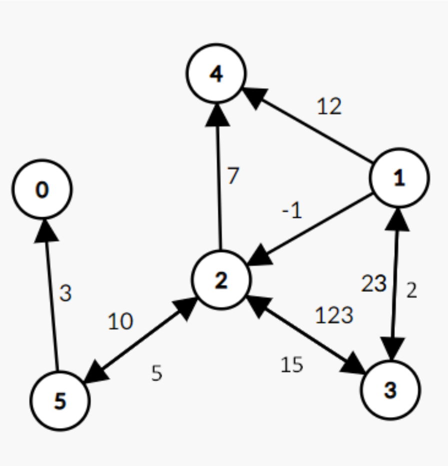
The following extern methods have been implemented for file work:

read_from_file(file_path, reading_type) = opens an existing file with the name "file_path" and interprets its data to generate a Graph object

write_in_file(file_path, graph_to_be_saved_in_file, writing_type) = creates a file with the name „file_path" and appends the contents of the current graph the program is using

The method used in the **__init__** process in the Graph class is also implemented externally in the function **generate_random_graph(number_of_vertices, number_of_edges)**, behaving in the same manner as the aforementioned one.

Example Graph nr. 1



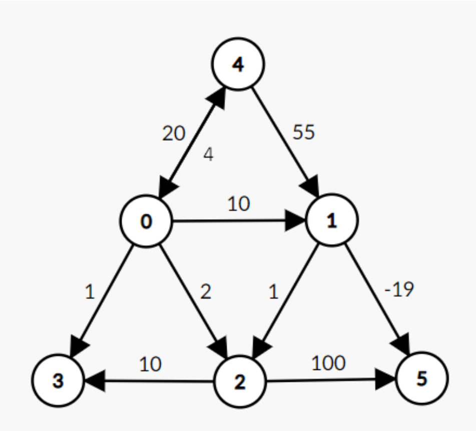
self._vertices = {0, 1, 2, 3, 4, 5}

self._outbound_neighbours = { 0: {}, 1: {2, 3, 4}, 2: {3, 4, 5}, 3: {1, 2}, 4: {}, 5:{0, 2} }

self._inbound_neighbours = { 0:{5}, 1: {3}, 2: {1, 3, 5}, 3: {1, 2}, 4: {1, 2}, 5: {2} }

self._cost = { {1,2} : -1, {1,3} : 23, {1,4} : 12, {2,3} : 15, {2,4}: 7, {2,5}: 10, {3,1}: 2, {3,2}: 123, {5,0} : 3, {5,2}: 5}

Example Graph nr. 2



self._vertices = {0, 1, 2, 3, 4, 5}

self._outbound_neighbours = {0: {1, 2, 3, 4}, 1: {2, 5}, 2: {3, 5}, 3: {}, 4: {0, 1}, 5: {} }

self._inbound_neighbours = {0: {4}, 1: {0, 4}, 2: {0, 1}, 3: {0, 2}, 4: {0}, 5: {1, 2} }

self._cost = { {0,1}: 10, {0,2}: 2, {0,3}: 1, {0,4}: 20, {1, 2}: 1, {1,5}: -19, {2,3}: 10, {2,5}: 100, {4,0}: 4, {4,1}: 55}

In the UI section the fucntions only deal with printing on console/writing in file, or reading from console/from file. Only handles input and output, and calls the Graph procedures for data operations.