The implementation for this practical project uses the python programming language. In the solving of the statement a Graph class was defined which represents a directed graph, class names UI which contains the part of the program designed to interact with the user and three external functions(read_from_file, write_to_file, generate_random_graph). Moreover, an edge is identified by two end-points.

The Class Graph has the following interface:

initialize_dictionaries(self): this function is responsible of making sure that each vertex has an in dictionary(where the neighbours that have a given index as an endpoint) and an out dictionary(where the neighbours that have a given index as an start point)

check_if_vertex_exists(self, vertex_to_check): this function is responsible of checking whether a certain vertex exists in the graph or not

number_of_vertices(self): is a property and is responsible of retrieving the number of vertices in the graph

get_number_of_edges(self): is a getter function and is responsible of retrieving the number of edges in the graph

get_edge_info(self,start_vertex,end_vertex): this function is responsible of retrieving the cost associated to a certain edge(the edge being specified by the start_vertex->start point and by the end_vertex->end_point)
returns the cost if the edge exists, None if the edge does not exist

set_edge_info(self, start_vertex, end_vertex, new_cost): this function is responsible of updating the cost associated to a certain edge(the edge being specified by the start_vertex->start point and by the end_vertex->end_point)
returns True if the edge exists and the cost was updated, False if the edge does not exist

parse_vertices(self): Iterator for the vertices of the graph

parse_inbound_of_vertex: Iterator for the inbound edges of a specified vertex

parse_outbound_of_vertex: Iterator for the outbound edges for a specified vertex

is_there_edge(self, start_vertex, destination_vertex): this function is responsible of checking whether a certain edge(the edge being specified by the start_vertex->start point and by the end_vertex->end_point) exists or not
returns True if the edge exists in the graph, False otherwise

get_in_degree_and_out_degree_of_vertex(self, given_vertex): this function is responsible of computing the in-degree and out-degree of a specified vertex
If the vertex does not exist it returns False, a tuple containing the out_degree and in_degree otherwise

add_vertex(self, vertex_to_add): this function is responsible of adding a new vertex to the graph
returns True if the vertex was added, False otherwise(vertex already existed)

add_to_in_dict(self, source_vertex, destination_vertex): this function is responsible of adding to the in dictionary of the source vertex the destination vertex(this means that we have an edge between them)

add_to_out_dict(self, source_vertex, destination_vertex): this function is responsible of adding to the out dictionary of the source vertex the destination vertex(this means that we have an edge between them)

add_to_cost(self, source_vertex, destination_vertex, cost): this function is responsible of adding a given edge to the edges dictionary

add_edge(self, start_vertex, end_vertex, cost): this function is responsible of adding a new edge to the graph
returns None if one of the given vertices does not belong to the graph, False if the edge already exists, True if the edge was added

remove_edge(self, start_vertex, end_vertex): this function is responsible of removing an edge from the graph
return False if the given edge does not exist in the graph, True if the edge existed and it was removed

__remove_vertex_from_neighbours(target_vertex): this function is responsible of removing an edge from all its neighbours(as the target_vertex is supposed to get deleted)

remove_vertex(target_vertex): this function is responsible of removing a certain vertex from the graph
returns False if the given vertex does not exist in the graph, True if it existed and it got removed

get_all_edges_plus_costs(self): this function is responsible of returning a copy of the cost dictionary(the one containing all the edges and costs)

The class Graph is initialized with the following data:
self.__in_dict - dictionary that holds the inbounds for every vertex
self.__out_dict - dictionary that holds the outbounds for every vertex
self.__cost_dict - dictionary that holds the edges and costs
self.__number_of_vertices - field that holds the number of vertices in the graph
self.__number_of_edges - filed that holds the number of edges in the graph

External Functions:

read_graph_from_file(file_name) - receives the name of the file containing the input data and then it uses it to create a graph containing that data… returns the created graph

write_graph_to_file(file_name, graph_instance) - receives the name of the file where the graph should be written to and a graph_instance that contains the data that should be written to the file given

generate_random_graph(number_of_vertices, number_of_edges) - receives the number of vertices and the number of edges that the generated graph should contain
Precondition: returns None if the number_of_edges is greater than the number of vertices squared
returns the randomly generated graph at the end

The Class UI is initialized with the following data:
self.__list_of_graph_copies[]- a list that will contain the copies of the graph
self.__active_graph - contains the currently focused graph

The Class UI interface is:

__print_menu(): this function is responsible of printing to the console the available commands

run_console(): this function is responsible for displaying the whole console along with the menu.
Also contains all the handlers for the different commands(inputs of the user)