Practical Work 1 – documentation

The implementation uses python language. For solving the specified requirements we shall use the Graph class, which represents a directed graph. The Console class is created to let the user use the application.

class Graph:

Fields:

- no_of_veritces # integer number
- no_of_edges # integer number
- dictionary_in # dictionary holding the inbound neighbours for every vertex
- dictionary_out # dictionary holding the outbound neighbours for every vertex
- dictionary_cost # dictionary holding the cost for every edge, an edge having the form [x, y]

Methods:

- initialise_in_and_out_for_random(self)
  # when generating a random graph, the indexes will be from 0 to no_of_vertices – 1,
  # and we'll create an empty list for inbound and outbound, having as keys every vertex

- no_of_vertices(self)
  # return the number of vertices of the graph

- no_of_edges(self)
  # return the number of edges of the graph

- dictionary_in(self)
  # return the dictionary_in field

- dictionary_out(self)
  # return the dictionary_out field

- valid_vertex(self, x)
  # check if a given index is part of the graph

- check_edge(self, x, y)
  # check if a given edge is part of the graph
  # return the cost if the edge exists or None otherwise

- get_in_degree(self, x)
  # precondition: x to be a valid vertex
  # return the in degree of x

- get_out_degree(self, x)
  # precondition: x to be a valid vertex
  # return the out degree

- parse_vertices(self)
  # iterate the vertices of the graph

- parse_inbound_edges(self, x)
  # precondition: x to be a valid vertex
  # iterate through the inbound neighbours of x

- parse_outbound_edges(self, x)
  # precondition: x to be a valid vertex
  # iterate through the outbound neighbours of x

- get_cost(self, x, y)
  # precondition: (x, y) to be a valid edge
  # return the cost associated to the given edge

- change_cost(self, x, y, new_cost)
  # precondition: (x, y) to be a valid edge
  # change the cost associated to edge (x, y)

- add_vertex(self, x)
  # precondition: the vertex x must not already exist
  # initialise the inbound and outbound neighbours of x as an empty list and increase the
  # number of vertices

- add_vertex_from_file(self, x)
  # just initialise the corresponding dictionaries

- remove_vertex(self, x)
  # precondition: x must be part of the graph
  # remove x as neighbour for every other vertex
  # remove any edge containing x
  # delete x from the dictionary_in and dictionary_out
  # decrease the number of vertices

- add_edge(self, x, y, cost)
  # precondition: x and y must be valid vertices, the edge must not already exist
  # x will be an inbound neighbour for y, y will be an outbound neighbour for x
  # add the edge to the dictionary_cost

- add_edge_from_file(self, x, y, cost)
  # just add the edge without preconditions (they will be resolved automatically)

- remove_edge(self, x, y)
  # precondition: x and y must be valid vertices and the edge must exist
  # solve the neighbourhood relationship
  # delete the edge from dictionary_cost
  # decrease the number of edges

- make_copy(self)
  # return a deep copy of the current graph

Auxiliary functions for reading and writing the graph to a file:

- read_from_file(path_to_file)
  # precondition: the file must exist
  # on the first line we'll have the number of vertices and the number of edges
  # on the following lines we have 2 possibilities:
  # if we have one number, it represents an isolated vertex
  # if we have 3 numbers, they represent an edge with a cost

- write_to_file(graph, path_to_file)
  # precondition: the graph must not be empty
  # write the isolated vertices
  # write the edges of the graph

class Console:

Fields:

- graph # the current graph, which will be initialised as an empty graph

Methods:

- generate_random_graph(self)
  # precondition: number of edges must be smaller than or equal to the number of vertices squared
  # update the graph

- get_graph_from_file(self)
  # use the read_from_file function to read the graph and update the graph field

- push_graph_to_file(self)
  # use the write_graph_to_file function to save the graph to a file

- get_number_of_vertices(self)
  # print the number of vertices using the getter method from Graph class

- get_number_of_edges(self)
  # print the number of edges the getter method from Graph class

- get_the_vertices(self)
  # print the vertices list using parse_vertices method from Graph class

- read_edge()
  # this is a static method
  # read and return x, y representing an edge

- test_edge(self)
  # read an edge and check if it has an associated cost in the graph using check_edge
  # method from Graph class

- get_in_degree_of_vertex(self)
  # read an vertex and use get_in_degree method from Graph class to print the in degree
  # of x

- get_out_degree_of_vertex(self)
  # read an vertex and use get_out_degree method from Graph class to print the out degree
  # of x

- get_inbound_edges(self)
  # using the parse_inbound_edges method from Graph class print the inbound
  # neighbours of a certain vertex

- get_outbound_edges(self)
  # using the parse_outbound_edges method from Graph class print the outbound
  # neighbours of a certain vertex

- modify_edge_cost(self)
  # use the change_cost method from Graph class to modify a certain edge

- add_vertex_to_graph(self)
  # use add_vertex method from Graph class to add a certain vertex

- remove_vertex_from_graph(self)
  # use remove_vertex method from Graph class to remove a certain vertex

- add_edge_to_graph(self)
  # use the add_edge method from the Graph class to add an edge

- remove_edge_from_graph(self)
  # use the remove_edge method from the Graph class to remove an edge

- get_copy(self)
  # use the make_copy method from Graph class to make a deep copy of the current graph

- print_menu()
  # this is a static method
  # print the menu options

- read_command()
  # this is a static method
  # return the command number from the user

- start(self)
  # this is where our function runs
  # we have a dictionary holding pointer to the functions presented above that will be
  # called depending on the command number

REMARKS!

The random_graph1.txt and random_graph2.txt are stored in src/data folder.