Graph Algorithm
Carp Nicoleta
Gr. 911

Practical Work nr1

Implementation

I defined a class named *Graph*, which is the representation of a directed graph.
The class has as properties:

self.__nr_vert = 0 *// number of vertices*
self.__nr_edge = 0 *// number of edges*

self.__in_bound = {} *// dictionary where the key is a vertex and the value assigned to it is a list of all vertices which go to the key and create an edge*

self.__out_bound = {} *// dictionary where the key is a vertex and the value assigned to it is a list of all vertices which go from the key and create an edge*

self.__edges = {} *// dictionary where the key is a tuple with the edge coordinates and the value assigned to it is the cost*

Besides the graph class and its assigned operations the program contains two other classes *MainMenu* and *ModifyGraph* that deal with user interface, error management and calling the needed Graph functions as the user wishes.
Both of them have the following properties:

self._graph = g *// the graph we are currently working with*
self.__in_bound = {…} *// a dictionary in which the keys are command lines the user writes in the console to call a specific function according to the printed menu, and the assigned values are the functions*

Specifications

The *Graph* class provides the following operations:

def is_vertex(self, x) -> bool
parameter: x – vertex
searches for x in the self.__in_bound dictionary's keys
if found returns True, else False

def is_edge(self, x, y) -> bool
parameter: x, y – vertices
creates a tuple (x, y) and searches if it exists in the self.__edges' keys
if found returns True, else False

def add_vertex(self, x) -> bool

parameter: x – vertex
number of vertices is incremented by 1 and x is added as a new key to self.__in_bound and self.__out_bound, its assigned values being an empty list
if added with success return True, if vertex already exists return False

def add_edge(self, x, y, c) -> bool

parameter: x, y – vertices, c – cost of edge
number of edges is incremented by 1, y is appended to the list of out bound of x and x is appended to the list of in bound of y, then c is assigned in the edges dictionary to the (x, y) key
if added with success return True, if edge already exists return False

def del_edge(self, x, y) -> bool

parameter: x, y – vertices
number of edges is decremented by 1, y is removed from the list of out bound of x and x is removed from the list of in bound of y, and the (x, y) element from the edges dictionary is deleted together with the key
if removed with success return True, if edge doesn't exist return False

def del_vertex(self, x) -> bool

parameter: x – vertex
by going through the in bound and out bound dictionaries, we look for all pairs of vertices that contain x and remove the edge from the graph (from all dictionaries) then we also remove the keys from the in bound and out bound dictionaries
if removed with success return True, if vertex doesn't exist return False

def load_file(self, file_name)

parameter: file_name – the name of the file to be accessed
it opens a .txt file, reads all the lines and adds the graph's data to the entity we created which called the function

def save_file(self, file_name)

parameter: file_name – the name of the file to be accessed or created
it opens or creates a .txt file, and writes in it all the data of the graph that called the function

def __str__(self)

creates the string version of the graph that's to be printed on the console for the user

The *ModifyGraph* class provides the following operations:

def operations_menu(self)

the menu that is printed on the console with all the operations accessible to the user

def check_vertex_exists(self)

calls the is_vertex operation of the Graph and prints a message according to the result

def check_edge_exists(self)

calls the is_edge operation of the Graph and prints a message according to the result

def vertex_info(self)

prints how many vertices there are in the graph and a list with all of them

def in_bound_vertex(self)

reads a vertex, and checks if exists
prints its in degree, if it's bigger than 0 than also prints all its in bound edges

def out_bound_vertex(self)

reads a vertex, and checks if exists
prints its out degree, if it's bigger than 0 than also prints all its out bound edges

def get_cost_edge(self)

reads coordinates of an edge, and checks if exists
if yes, prints its cost

def vertex_add(self)

reads a vertex, calls the add_vertex function of the Graph class and prints an according message

def vertex_remove(self)

reads a vertex, calls the del_vertex function of the Graph class and prints an according message

def edge_add(self)

reads coordinates of an edge, and checks if those vertices exists
if yes, reads the cost, calls the add_edge function of the Graph class and prints an according message

def edge_remove(self)

reads coordinates of an edge, calls the del_edge function of the Graph class and prints an according message

def modify_cost(self)

reads coordinates of an edge, and checks if it exists
if yes, reads the new cost and changes it

def print_dict(self)

prints all dictionaries of the graph we are currently working with

def start(self)

reads user's command and calls the according function

The *MainMenu* class provides the following operations:

def print_menu(self)

the menu that is printed on the console with all the operations accessible to the user

def modify_menu(self)

creates a ModifyGraph entity and calls its start, moving the user to the other operations and menu

def make_copy(self)

creates a copy and saves it in the *copy.txt* file

def read_graph_file(self)

reads a file name and calls the load_file function of the graph, replacing the current data with the once we read

def save_graph_file(self)

reads a file name and calls the save_file function of the graph, saving the current data in the mentioned .txt file

def create_random_graph(self)

reads the number of vertices and edges we need and calls the create_random_graph function, creating a new graph which will replace our current one
in case number of edges is bigger than possible prints an error

def print_graph(self)

prints the graph

def start(self)

reads user's command and calls the according function

Additionally, there are two more functions outside the classes in the main file:

def create_random_graph(n, m)

parameter: n – number of vertices, m – number of edges
it creates a random graph
it chooses randomly from a list with all possible combinations of n coordinates for an edge and adds the new edge with a randomly chosen cost to the graph
return: the newly created graph

def copy_graph(g: Graph)

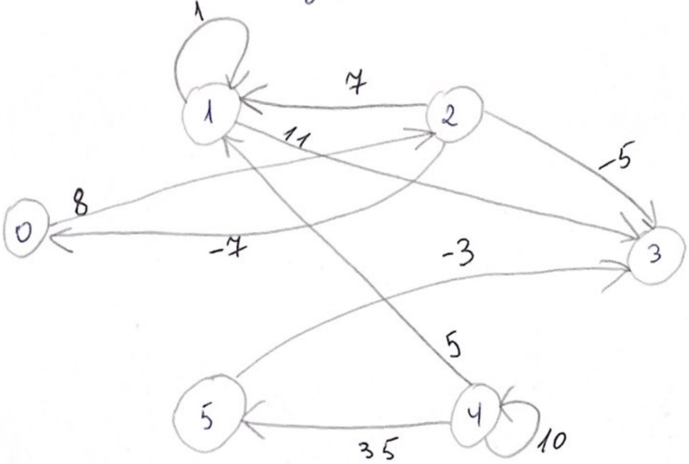
parameter: g – the graph we want to copy
it creates a new graph *copy* and copies all the data from *g* to *copy*
return: the *copy*

Carp Nicolita
gr. 911

Practical work Nr.1

input_file.txt graph

6 Vertices 10 edges



```
6 10
0 2 8
1 1 1
1 3 11
2 0 -7
2 1 7
2 3 -5
4 1 5
4 4 10
4 5 35
5 3 -3
```

in_bound dict.

0<-[2]
1<-[1, 2, 4]
2<-[0]
3<-[1, 2, 5]
4<-[4]
5<-[4]

out_bound dict.

0->[2]
1->[1, 3]
2->[0, 1, 3]
3->[ ]
4->[1, 4, 5]
5->[3]

edges dict.

(0, 2) -> 8
(1,1) -> 1
(1, 3) -> 11
(2, 0) -> -7
(2,1) -> 7
(2, 3) -> -5
(4, 1) -> 5
(4, 4) -> 10
(4,5) -> 35
(5, 3) -> -3