**Moldovan Alexandru-Vasile**

**Group 915**

**Practical work nr. 2**

Assigned problem: 3. Write a program that finds the connected components of an undirected graph using a depth-first traversal of the graph.

The data in the text file looks like:

The actual text file:

n\_vertices m\_edges 6 7

edgeFrom1 edgeTo1 0 0

…….. 0 1

edgeFromN edgeToN 1 3

2 1

1 2

2 3

3 4

The input data it’s read from a text file. The function that makes that looks like:

def readGraphFile(ctor):

"""

Reads the graph from the file Edges.txt

ctor: The class of the graph

"""

try:

edgesFile = open("EdgesInput.txt", "r") //The file from where we read

line = edgesFile.readline().strip()

lx = line.split(' ')

n\_vertex = int(lx[0]) //How many vertices

m\_edges = int(lx[1]) //How many edges we have

g = DoubleDictGraph(n\_vertex) //Where we save the read data

line = edgesFile.readline().strip()

while line != "":

lx = line.split(' ')

g.addEdge(int(lx[0]), int(lx[1]), int(lx[2]))

line = edgesFile.readline().strip()

edgesFile.close()

return [g, int(n\_vertex), int(m\_edges)] //Returns the graph with n

//vertices and m edges

except IOError: pass

The Graph class it made from:

class DictGraph:

"""

An undirected graph, represented as a map from each vertex to

the set of outbound neighbours

"""

def \_\_init\_\_(self, n):

"""

Creates a graph with n vertices (numbered from 0 to n-1)

and no edges

"""

self.\_dict = {} //Creates a null dictionary of vertices

for i in range(n):

self.\_dict[i] = [] //For every key, we add a null list

The program allows to make some operations on edges and vertices like: adding, deleting, updating.

Adding an edge:

def addEdge(self, x, y):

"""

Adds an edge from x to y.

"""

//The grapf it’s undirected, so we have an edge from x->y and y->x

self.\_dict[x].append(y) //We add y to the key x

self.\_dict[y].append(x) //We add x to the key y

Removing a vertex:

def removeVertex(self, x, m\_edges):

'''

:param x: The vertex

:return: the graph without that vertex

'''

ok = True

while ok == True: //We do this, until we don’t have any

//more edges

ok = False

for i in self.parseN(int(x)): //For all the neighbours of x

if self.isEdge(int(i), int(x)): //If there is an edge

self.removeEdge(int(i), int(x)) //Then we will remove the edge

m\_edges = m\_edges – 1 //We decrease the number of edges

ok = True

self.\_dict.pop(x) //We remove the key from the dictionary

return int(m\_edges) //We return the number of edges

Every input data is validated. For example when a vertex needs to be deleted, we check if that vertex does exist. This is done by:

def isVertex(self,x):

'''

Checks if the vertex does exists

:param x: the vertex we check for

'''

//We check if the given vertex does exist in the dictionary

return x in self.\_dict.keys()

Also we have a validator for the existance of an edge:

def isEdge(self, x, y):

"""

Returns True if there is an edge from x to y (respectivly y to x), False otherwise

"""

//If y does exist in the list of the vertex x, and x exist in the list of vertex y then we //have an edge

//between x and y

return y in self.\_dict[x] and x in self.\_dict[y]

In this program, the depth-first traversal it’s implemented. The algorithms looks like:

def DFSRec(self, v, visited):

visited[v] = True //If we reached this point, means that

//this vertex wasn’t parsed.

//It is marked as “True”

print("\t\t",v) //We print the vertex

for i in self.\_dict[v]: //We parse the list of the current key

if visited[i] == False: //We check if the current vertex is not parsed

self.DFSRec(i, visited) //If not, then this function it’s called again

def DFS(self):

visited = [False] \* (len(self.\_dict)) //A list it’s made. In this list is reserved

//how many vertices we have.

//On every position we place “Fasle”

//”False” means that we weren’t on that

self.DFSRec(v, visited) //v is the vertex from where we start

After the program it’s closed by introducing the command “0”, we got an output file that contains the graph after some changes were applied

def writeToGraphFile(graph,n,m):

'''

Writes the current graph in a file

'''

edgesFile = open("EdgesOutput.txt", "w") //Open the output file

try:

s = str(n) + " " + str(m) + "\n" //We write the number o

edgesFile.write(s) //vertices and edges on the first line

for i in graph.parseX(): //We go through all the vertices

for j in graph.parseNout(i):

//We write the edge from I to J

s = str(i) + ' ' + str(j) + ' ' + str(graph.getCost(int(i),int(j))) + '\n'

edgesFile.write(s) //Write in the file

edgesFile.close() //We close the file.

except Exception as e: print('\t \n', e)