Ungureanu Ionut-Vladimir Group 917

Graph Algorithm

Laboratory Practical Work

Storing mechanism:

- dictionaryIn = the dictionary which will store the inbound edges of a vertex. Every key of the dictionary will represent a vertex and will have as a value a list of inbound edges
- dictionaryIn = the dictionary which will store the outbound edges of a vertex. Every key of the dictionary will represent a vertex and will have as a value a list of outbound edges
- dictionaryEdges = the dictionary which will store the edges of the graph. Every key of the dictionary will be formed of a tuple representing the origin and destination of the edge. The value corresponding to each key will be the cost of the edge

```
class Graph:
    def __init__(self):
        # the dictionary that will define all the edges going IN and OUT of an element
        self.__dictionaryIn = dict()
        self.__dictionaryOut = dict()
        self.__dictionaryEdges = dict()

    def addVertex(self,index):
        '''
        Function will add a vertex with the given index in the dictionaries
representing the graph
        :param index: the index of the vertex to be added
        :post condition: the vertex will be added in the dictionaries
        :throws - Value error if index already exists
        '''
        if self.existsVertex(index):
            raise ValueError("Index already exists")
        else:
            self.__dictionaryIn[index] = dict()
            self.__dictionaryOut[index] = dict()


    def removeVertex(self,index):
        '''
        Function will remove a vertex from the graph by removing it from the
dictionaries and the edges
        :param index: the vertex to be removed
        :post condition: the vertex will be removed from the dictionaries and all
edges cut off
        :throws - Value error if index already exists
        '''

        if self.existsVertex(index) is not True:
            raise Exception("There is no such vertex in the graph")
```

```
        #we create new dictionaries not containing anything regarding the vertex to be
removed

        newDictionary = dict()
        for vertexStart,vertexDestination in self.__dictionaryOut.items(): # cut all
edges
            if index != vertexDestination and index !=vertexStart:
                newDictionary[vertexStart] = vertexDestination
        self.__dictionaryOut = newDictionary

        newDictionary = dict()
        for edge in self.__dictionaryEdges: #remove edges from edges dictionary
            if edge[1] != index and edge[0] != index:
                newDictionary[edge] = self.__dictionaryEdges[edge]
        self.__dictionaryEdges = newDictionary

        newDictionary = dict()
        for vertexDestination,vertexStart in self.__dictionaryIn.items(): # cut all
edges
            if index != vertexDestination and index !=vertexStart:
                newDictionary[vertexDestination] = vertexStart

        self.__dictionaryIn = newDictionary

    def removeEdge(self,firstVertex,secondVertex):
        '''
        Function will remove an edge from the edge dictionary
        :param firstVertex = the origin of the edge
        :param secondVertex = the destination of the edge
        :post condition = the edge dictionary won't contain the edge anymore and the
dictionaries of ins and outs won't either
        :throws Exception if there is no edge (which in turn will throw exception is
one of the vertices is missing)

        '''
        if self.isEdge(firstVertex,secondVertex) is not True:
            raise Exception("No edge here. Sorry")
        self.__dictionaryIn[secondVertex].pop(firstVertex)
        self.__dictionaryOut[firstVertex].pop(secondVertex)
        self.__dictionaryEdges.pop((firstVertex,secondVertex))

    def addEdge(self,firstVertex,secondVertex,cost):
        '''
        Function will add an edge with a given cost
        :param firstVertex = the origin of the edge
        :param secondVertex = the destination of the edge
        :param cost = the cost of the new edge
        :post condition = the edge dictionary will contain the new edge and the
dictionaries of ins and outs will as well
        :throw Exception if one of the vertices does no exist
        '''
        if (self.existsVertex(firstVertex) and self.existsVertex(secondVertex)) is not
True:
            raise Exception("One of the vertices does not exist. Sorry")
        if self.isEdge(firstVertex,secondVertex) is True:
            raise Exception("There already is an edge there. Don't be edgy")

        #This means the firstVertex key dictionary has a dictionary with key to
secondVertex and the cost is
        #the value between the 2 of them
        self.__dictionaryOut[firstVertex][secondVertex] = cost
        self.__dictionaryIn[secondVertex][firstVertex] = cost
```

```
        self.__dictionaryEdges[(firstVertex,secondVertex)] = cost

    def modifyEdge(self,firstVertex,secondVertex,newCost):
        '''
        Function will add an edge with a given cost
        :param firstVertex = the origin of the edge
        :param secondVertex = the destination of the edge
        :param cost = the cost of the new edge
        :post condition = the edge dictionary will modify the edge cost
        :throw Exception is edge does not exist (which in turn will raise exception if
vertices do not exist)
        '''
        if self.isEdge(firstVertex,secondVertex) is not True:
            raise Exception("There is no edge")
        self.__dictionaryEdges[(firstVertex,secondVertex)] = newCost


    def existsVertex(self,candidate):
        '''
        Function will return true if a certain vertex exists
        :param candidate: the vertex to be checked
        :return: true if the vertex with the given index is there and false otherwise
        '''
        if candidate in self.__dictionaryIn:
            return True
        return False

    def numberOfVertices(self):
        '''
        :return: the number of vertices in the graph
        '''
        return len(self.__dictionaryIn.keys())

    def numberOfEdges(self):
        '''
        Function will return the number of edges in the graph (by accessing the
dictionary edges keys)
        :return: the number of edges in the graph
        '''
        return len(self.__dictionaryEdges.keys())

    def getEdges(self):
        '''
        Function will return the edges of the graph.
        :return: the dictionary edges of the graph
        '''
        return self.__dictionaryEdges

    def getCost(self,firstVertex,secondVertex):
        '''
        Function will get the cost of an edge between 2 vertices
        :param firstVertex: the origin of the edge
        :param secondVertex: the destination of the edge
        :throw exception if there is no edge( which in turn will raise exception if
vertices do not exist)
        :return: the cost of the given edge in the graph
        '''
        if self.isEdge(firstVertex,secondVertex) is not True:
            raise Exception("No edge when getting cost")

        return self.__dictionaryEdges[(firstVertex,secondVertex)]
```

```
    def isEdge(self,firstVertex,secondVertex):
        '''
        Function will check if there exists an edge between 2 vertexes
        :param firstVertex: one of the vertices
        :param secondVertex: the other vertix
        :return:True if it does and False otherwise
        '''
        if self.existsVertex(firstVertex) is not True or
self.existsVertex(secondVertex) is not True:
            raise Exception("One of the vertices does not exist")
        if (firstVertex,secondVertex) in self.__dictionaryEdges:
            return True
        return False

    def inDegree(self,vertex):
        '''
        Function will return the in degree (the number of edges that go "in" the
graph)
        :param vertex: the vertex to compute de degree of
        :throw = function will raise exception if there is no vertex
        :return: the degree of the "in" graph
        '''
        if self.existsVertex(vertex) is not True:
            raise Exception("No such vertex")

        return len(self.__dictionaryIn[vertex])

    def outDegree(self,vertex):
        '''
        Function will return the out degree (the number of edges that go "out" of the
graph)
        :param vertex: the vertex to compute de degree of
        :throw = function will raise exception if there is no vertex
        :return: the degree of the "out" graph
        '''
        if self.existsVertex(vertex) is not True:
            raise Exception("No such vertex")

        return len(self.__dictionaryOut[vertex])

    def getInboundEdges(self,vertex):
        '''
        Function will return a list of inbound edges of the given vertex
        :param vertex: the vertex whose inbound edges should be returned
        :return: a python list of inbound edges
        :throw: value error if there is no such vertex
        '''
        if self.existsVertex(vertex) is not True:
            raise ValueError("There is no such vertex")
        return self.__dictionaryIn[vertex]

    def getOutboundEdges(self,vertex):
        '''
        Function will return a list of outbound edges of the given vertex
        :param vertex: the vertex whose outbound edges should be returned
        :return: a python list of outbound edges
        :throw: value error if there is no such vertex
        '''
        if self.existsVertex(vertex) is not True:
            raise ValueError("There is no such vertex")
        return self.__dictionaryOut[vertex]
```

```
    def readFromFile(self,fileName):
        '''
        Function will read from the given file the graph infomration and will store it
in the instance of this class
        :param fileName: the file with the information which should be read
        :return: True if the graph has been read sucessful or False oterwise (Also
will print the exception)
        '''
        try:
            i=1
            file = open(fileName, "r")
            firstLine = file.readline().strip().split(" ")

            # firstLine is an array. first element will be number of vertices and
second the number of edges
            for i in range(int(firstLine[0])):  # add vertices
                self.addVertex(i)

            for i in range(int(firstLine[1])):
                line = file.readline().strip().split(" ")
                print("Line number", i)
                i+=1
                # line is a list where the first 2 elements are the vertices and the
last is the cost of the edge
                self.addEdge(int(line[0]), int(line[1]), int(line[2]))

            file.close()
        except IOError as e:
            print(e)
            return False

        return True

    def writeToFile(self,fileName):
        '''
        Function will write to the given file the graph information for future use
        :param fileName: the file where the information should be written
        :param fileName: the file where the graph should be read
        '''
        file = open(fileName,"w")
        nrVertices = self.numberOfVertices()
        nrEdges = self.numberOfEdges()
        file.write(str(self.numberOfVertices()) + " " + str(self.numberOfEdges()) +
"\n")
        for key,elem in self.__dictionaryEdges.items():
            file.write(str(key[0]) + " " + str(key[1]) + " " + str(elem) + "\n")
        file.close()
```