**DOCUMENTATION FOR DIRECTED GRAPH IMPLEMENTATION**

**Specifications**

We shall define a class named **Graph** representing a *directed graph*, and an auxiliary class called **GraphException** for raising exceptions whenever the conditions implied are not met.

The class Graph will provide the following methods:

**\_\_init\_\_** : The constructor that initializes a graph with n vertices. The attributes of the class Graph consist of:

* \_dout: Dictionary where each key is a vertix and each value is a list where each element is an outbound vertix for the key vertix
* \_din: Dictionary where each key is a vertix and each value is a list where each element is an inbound vertix for the key vertix.
* \_costs: Dictionary where each key is a tuple representing an edge, and each value represents the cost of that edge.

**is\_vertex**: Checks if a vertex is already part of the graph, If the condition is satisfied, returns True, otherwise returns False.

**is\_edge**: Checks if two nodes determine an edge existent in the graph. Returns True if the condition is satisfied, False otherwise.

**add\_vertex**: Adds a new vertex to the dictionaries of outbounds and inbounds.

def add\_vertex(self, x):  
 *"""  
 Adds a vertex to the dictionary of inbounds and dictionary of outbounds* ***:param*** *x: vertex to add  
 """* if self.\_is\_vertex(x):  
 raise GraphException("The vertex is already among us!")  
 else:  
 self.\_dout[x] = []  
 self.\_din[x] = []

**add\_edge**: Adds a new edge to the dictionary costs and to the dictionaries of outbounds and inbounds by apepnding the corresponding origin/destination vertix.

def add\_edge(self,x,y,c):  
 *"""  
 Adds edge to dictionary of inbounds, outbounds and dictionary of costs* ***:param*** *x: origin* ***:param*** *y: destination* ***:param*** *c: cost  
 """* if self.is\_edge(x,y):  
 raise GraphException("The edge is already among us!")  
 else:  
 self.\_dout[x].append(y)  
 self.\_din[y].append(x)  
 self.\_costs[(x,y)]=c

**edge\_info**: Returns a string containing origin, destination and cost of an edge. If there’s an isolated vertix, the cost will be -1.

**get\_costs**: Returns the dictionary of costs.

**remove vertex**: Deletes the specified vertex and its respective edges from both dictionary of inbounds and dictionary of outbounds.

def remove\_vertex(self,x):  
 *"""  
 Deletes vertex from all the dictionaries* ***:param*** *x: vertex to remove  
 """* if self.\_is\_vertex(x):  
 for destination in self.\_dout[x]:  
 self.\_din[destination].remove(x)  
 del self.\_costs[(x, destination)]  
 for origin in self.\_din[x]:  
 self.\_dout[origin].remove(x)  
 del self.\_costs[(origin,x)]  
 del self.\_din[x]  
 del self.\_dout[x]

**remove\_edge**: Deletes the specified edge from the dictionary of costs and from the dictionary of inbounds, outbounds by removing the destination from the lists in dout, and removing the origin from the lists in din.

def remove\_edge(self,x,y):  
 *"""  
 Deletes edge from all the dictionaries* ***:param*** *x: origin* ***:param*** *y: destination  
 """* if self.is\_edge(x,y):  
 del self.\_costs[(x,y)]  
 self.\_dout[x].remove(y)  
 self.\_din[y].remove(x)

**change\_cos**t: Modifies the value of the cost of the specified edge.

**degree\_inbounds**: Returns integer representing inbound degree of specified vertex.

**degree\_outbounds**: Returns integer representing outbound degree of specified vertex.

**number\_of\_vertices**: Return integer representing the total number of vertices.

**parse\_vertices**: Returns iterator of the vertices.

**parse\_outbounds**: Returns iterator of outbound vertices of the specified vertex.

**parse\_inbounds**: Returns iterator of outbound vertices of the specified vertex.

def degree\_inbounds(self,x):  
 *"""* ***:param*** *x: vertex used for finding in degree* ***:return****: inbounds of vertex x  
 """* if self.\_is\_vertex(x):  
 return len(self.\_din[x])

**make\_copy**: Creates deep copy of the initial Graph.

**print\_dict**: Return all three dictionaries.

**UI FUNCTIONS**

**write\_to\_file**: writes to file the graph information, according to the format:

for isolated nodes:

<node> - 1

otherwise:

<origin> <destination> <cost>

def write\_to\_file(self,file\_name):  
 with open(file\_name, "w") as file:  
 for element in self.\_graph.get\_costs.keys():  
 file.write(self.\_graph.edge\_info(element)+ "\n")  
 file.close()

**read\_from\_file**:identifies the type of the file we are reading, and after that reads the information about the graph

1st type format: on the first line the number of vertices and edges, on the next lines<origin> <destination><cost>

2nd type format: the one we write to file

def read\_from\_file(self,file\_name):  
 with open(file\_name, 'r') as file:  
 prop = file.readline().split()  
 print(prop)  
 if len(prop)==2 and prop[1]!='-1':  
 n=int(prop[0])  
 m=int(prop[1])  
 self.\_graph = Graph(n)  
 coordinates= [line for line in file.read().strip().split("\n")]  
for coord in coordinates:  
 coord=coord.split()  
  
 x=int(coord[0])  
 y=int(coord[1])  
 c=int(coord[2])  
 try:  
 self.\_graph.add\_vertex(x)  
 except:  
 pass  
  
 try:  
 self.\_graph.add\_vertex(y)  
 except:  
 pass  
  
 self.\_graph.add\_edge(x,y,c)  
  
 self.\_graph.print\_dict()  
  
 else:  
 n=len(file.readlines())  
 self.\_graph=Graph(n)  
 if len(prop)==1:  
 try:  
 self.\_graph.add\_vertex(int(prop[0]))  
 except:  
 pass  
 elif len(prop)==3:  
 for i in range(2):  
 self.\_graph.add\_vertex(int(prop[i]))  
   
self.\_graph.add\_edge(int(prop[0]),int(prop[1]),int(prop[2]))  
 coordinates = [line for line in file.read().strip().split("\n")]  
 for coord in coordinates:  
 coord = coord.split()  
 if len(coord)>2:  
 x = int(coord[0])  
 y = int(coord[1])  
 c = int(coord[2])  
 else:  
 x = int(coord[0])  
 y = x  
 c = -1  
  
 try:  
 self.\_graph.add\_vertex(x)  
 except:  
 pass  
  
 try:  
 self.\_graph.add\_vertex(y)  
 except:  
 pass  
  
 self.\_graph.add\_edge(x, y, c)  
copy\_graph=self.\_graph.make\_copy()  
copy\_graph.print\_dict()

**create\_random**: Creates random graph considering the number of vertices, prints its information and returns the graph.

def create\_random(self,creator=Graph):

*"""****:param*** *creator: Class OBject Graph for generating a random graph****:return****: Randomly generated graph  
"""*

n=random.randint(1,10)  
 m=random.randint(0,n\*n)  
 graph=creator(n)  
 added\_edges=0  
 failed\_edges=0  
 while added\_edges<m:  
 x=random.randrange(n)  
 y=random.randrange(m)  
 c=random.randint(1,10)  
 try:  
 graph.add\_vertex(x) try:  
 graph.add\_vertex(x)  
 except:  
 pass  
 try:  
 graph.add\_vertex(y)  
 except:  
 pass  
 try:  
 graph.add\_edge(x,y,c)  
 added\_edges+=1  
 except:  
 failed\_edges+=1  
 print(f"Vertices: {n}, Edges: {m}, "  
 f"Added edges: {added\_edges}, "  
 f"Failed attempts: {failed\_edges}")  
 print("\n")  
 graph.print\_dict()  
 return graph

except:  
 pass  
 try:  
 graph.add\_vertex(y)  
 except:  
 pass  
 try:  
 graph.add\_edge(x,y,c)  
 added\_edges+=1  
 except:  
 failed\_edges+=1  
 print(f"Vertices: {n}, Edges: {m}, "  
 f"Added edges: {added\_edges}, "  
 f"Failed attempts: {failed\_edges}")  
 print("\n")  
 graph.print\_dict()  
 return graph