Directed Graph

I have created a Python class representing a Directed Graph. Furthermore, there can be no multiple edges between the same vertices and there cannot be an edge having as endpoins the same vertex.

These are the specifications of the methods:

class DirectedGraph:

def __init__(self, no_vertices = 0):

'''
Constructor Method

Preconditions:
-No_Vertices is an integer
-No_Vertices >= 0

Creates a Directed Graph with No_Vertices vertices.
Vertices have values between 0 and (No_Vertices - 1).
'''

def getNoVertices(self):

'''
Returns the number of vertices.
'''

def getNoEdges(self):

'''
Returns the number of edges.
'''

def addEdge(self, start, end, cost = 0):

'''
Preconditions:
-start, end, cost integers
-start != end
-start, end is a vertex from the graph
-there is no edge between start and end
Throws:
-ValueError if start or end are not vertices in the graph
-ValueError if there is already an edge between start and end
-ValueError is start == end

Adds an edge from vertex START to END with the cost COST.(COST is implicitly 0)
'''

def isEdge(self, start, end):

'''
Preconditions:
-start, end integers
-start, end is a vertex from the graph
Throws:
-ValueError if start or end are not vertices in the graph

Returns True if there is an edge from start to end, and False otherwise.
'''
def getInDegree(self, vertex):
'''
Preconditions:
-vertex integer
-vertex is a vertex in the graph
Throws:
-ValueError if there is no vertex VERTEX in the graph

Returns the in-bound degree of vertex.
'''
def getOutDegree(self, vertex):
'''
Preconditions:
-vertex integer
-vertex is a vertex in the graph
Throws:
-ValueError if there is no vertex VERTEX in the graph

Returns the out-bound degree of vertex.
'''
def getOutboundNeighbours(self, vertex):
'''
Preconditions:
-vertex integer
-vertex is a vertex in the graph
Throws:
-ValueError if there is no vertex VERTEX in the graph

Returns an iterator containing all out-bound neighbours of vertex VERTEX.
'''
def getInboundNeighbours(self, vertex):
'''
Preconditions:
-vertex integer
-vertex is a vertex in the graph
Throws:
-ValueError if there is no vertex VERTEX in the graph

Returns an iterator containing all in-bound neighbours of vertex VERTEX.
'''
def modifyCost(self, start, end, cost):
'''
Preconditions:
-start, end integers
-start, end is a vertex from the graph
-start end is an edge of the graph
Throws:
-ValueError if start or end are not vertices in the graph
-ValueError if start end is not an edge of the graph

Modifies the cost of the edge START END to COST```

def getEdgeCost(self, start, end):

'''

Preconditions:
-start, end integers
-start, end is a vertex from the graph
-start end is an edge of the graph
Throws:
-ValueError if start or end are not vertices in the graph
-ValueError if start end is not an edge of the graph

Returns the cost of the edge START END.
'''

def addVertex(self, vertex):

'''

Preconditions:
-vertex integer
-vertex is not already a vertex in the graph
Throws:
-ValueError if vertex is already a vertex in the graph

Adds vertex VERTEX to the graph.
'''

def removeVertex(self, vertex):

'''

Preconditions:
-vertex integer
-vertex is a vertex in the graph
Throws:
-ValueError if vertex is not a vertex in the graph

Removes vertex VERTEX from the graph.
'''

def removeEdge(self, start, end):

'''

Preconditions:
-start, end integers
-start, end is a vertex from the graph
-start end is an edge of the graph
Throws:
-ValueError if start or end are not vertices in the graph
-ValueError if start end is not an edge of the graph

Removes the edge START END from the graph.
'''

def readFromFile(self, filename):

'''

Preconditions:
-filename should be a valid, readable file
Throws:
-IOError if filename is not a valid, readable file
-Unexpected Errors if the file does not have the below specified format

Reads a Directed Graph from the FILE filename.

```
        The file must have the following format:
        no_vertcies no_edges
        start_vertex(1) end_vertex(1) cost(1)
        ...
        start_vertex(n-1) end_vertex(n-1) cost(n-1)
        '''
def makeCopy(self):
        '''
        Returns a (deep) copy of the Directed Graph.
        '''
def createRandomGraph(n, m):
        '''
        Returns a newly created Directed Graph with N nodes and M vertices, randomly generated.
        '''
```

These are the implementations of the functions:

```
def __init__(self, no_vertices = 0):
        self.__no_vertices = no_vertices
        self.__no_edges = 0
        self.__inMap = {}
        self.__outMap = {}
        self.__vertices = []
        for index in range(no_vertices):
            self.__inMap[index] = []
            self.__outMap[index] = []
            self.__vertices.append(index)
def getNoVertices(self):
            return self.__no_vertices
def getNoEdges(self):
            return self.__no_edges
def getVertices(self):
            for vertex in self.__vertices:
                        yield vertex
def addEdge(self, start, end, cost = 0):
         if self.isEdge(start, end):
            raise ValueError("The Edge does already exist!")
        if (start == end):
            raise ValueError("We cannot have an edge with endpoints being the same vertex!")
        if (start in self.__vertices and end in self.__vertices):
            self.__inMap[end].append([start, cost])
            self.__outMap[start].append([end, cost])
            self.__no_edges += 1
        else:
            raise ValueError("The Start Vertex or/and End Vertex does not exist!")
def isEdge(self, start, end):
      if (start in self.__vertices and end in self.__vertices):
            for edge in self.__outMap[start]:
                if (edge[0] == end):
                    return True
            return False
        else:
```

```
            raise ValueError("The Start Vertex or/and End Vertex does not exist!")
    def getInDegree(self, vertex):
        if vertex in self.__vertices:
            return len(self.__inMap[vertex])
        else:
            raise ValueError("The Vertex does not exist!")
    def getOutDegree(self, vertex):
        if vertex in self.__vertices:
            return len(self.__outMap[vertex])
        else:
            raise ValueError("The Vertex does not exist!")
    def getOutboundNeighbours(self, vertex):
        if vertex in self.__vertices:
            for edge in self.__outMap[vertex]:
                yield edge[0]
        else:
            raise ValueError("The Vertex does not exist!")
    def getInboundNeighbours(self, vertex):
        if vertex in self.__vertices:
            for edge in self.__inMap[vertex]:
                yield edge[0]
        else:
            raise ValueError("The Vertex does not exist!")
    def modifyCost(self, start, end, cost):
        if not self.isEdge(start, end):
            raise ValueError("The Edge does not exist!")
        for edge in self.__outMap[start]:
            if edge[0] == end:
                edge[1] = cost
                break
        for edge in self.__inMap[end]:
            if edge[0] == start:
                edge[1] = cost
                break
    def getEdgeCost(self, start, end):
        if not self.isEdge(start, end):
            raise ValueError("The Edge does not exist!")
        for edge in self.__outMap[start]:
            if edge[0] == end:
                return edge[1]
    def addVertex(self, vertex):
        if vertex in self.__vertices:
            raise ValueError("The Vertex does already exist!")
        self.__no_vertices += 1
        self.__vertices.append(vertex)
        self.__inMap[vertex] = []
        self.__outMap[vertex] = []
    def removeVertex(self, vertex):
        if not vertex in self.__vertices:
            raise ValueError("The Vertex does not exist!")

        #self.__inMap is modified by the method removeEdge, so we have to save it prior to
        #iterating through the map
```

```
        inMap = copy.deepcopy(self.__inMap[vertex])
        for edge in inMap:
            in_neighbour = edge[0]
            self.removeEdge(in_neighbour, vertex)

        outMap = copy.deepcopy(self.__outMap[vertex])
        for edge in outMap:
            out_neighbour = edge[0]
            self.removeEdge(vertex, out_neighbour)

        self.__inMap.pop(vertex)
        self.__outMap.pop(vertex)
        self.__no_vertices -= 1
        self.__vertices.remove(vertex)
    def removeEdge(self, start, end):
         if not self.isEdge(start, end):
            raise ValueError("The Edge does not exist!")
        for index in range(len(self.__inMap[end])):
            edge = self.__inMap[end][index]
            if edge[0] == start:
                self.__inMap[end].pop(index)
                break;
        for index in range(len(self.__outMap[start])):
            edge = self.__outMap[start][index]
            if edge[0] == end:
                self.__outMap[start].pop(index)
                break;
        self.__no_edges -= 1
    def readFromFile(self, filename):
        file = open(filename, "r")
        file_string = file.readline()
        file_string = file_string.split(" ")
        self.__no_vertices = int(file_string[0])
        self.__no_edges = 0
        #no_edges will be incremented in every addEdge
        aux_edges = int(file_string[1])
        self.__inMap.clear()
        self.__outMap.clear()
        self.__vertices.clear()
        for index in range(self.__no_vertices):
            self.__inMap[index] = []
            self.__outMap[index] = []
            self.__vertices.append(index)
        for _ in range(aux_edges):
            file_string = file.readline()
            file_string = file_string.split(" ")
            start = int(file_string[0])
            end = int(file_string[1])
            cost = int(file_string[2])
            self.addEdge(start, end, cost)
        file.close()
```

```
 def makeCopy(self):
        new_graph = DirectedGraph(0)
        new_graph.__no_vertices = self.__no_vertices
        new_graph.__no_edges = self.__no_edges
        new_graph.__inMap = copy.deepcopy(self.__inMap)
        new_graph.__inMap = copy.deepcopy(self.__outMap)
        new_graph.__vertices = copy.deepcopy(self.__vertices)
        return new_graph

@staticmethod
def createRandomGraph(n, m):
        if ((n - 1) * (n - 1) <= m):
            raise ValueError("The number of edges is too big!")
        new_graph = DirectedGraph(0)
        while n > 0:
            vertex = random.randrange(1000)
            if vertex not in new_graph.getVertices():
                new_graph.addVertex(vertex)
                n -= 1;
        n = new_graph.getNoVertices()
        possible_pairs = []
        for i in range(n):
            for j in range(n):
                if i != j:
                    possible_pairs.append([i, j])
        random.shuffle(possible_pairs)
        while m > 0:
            m -= 1
            cost = random.randrange(1000)
            pair = possible_pairs.pop()
            start = pair[0]
            end = pair[1]
            new_graph.addEdge(new_graph.__vertices[start], new_graph.__vertices[end], cost)
        return new_graph
```