**SPECIFICATION**

We will define a class “Graph” representing a direct graph and an auxiliary class called “Edge” which will model an edge corresponding to a directed graph.

**Class “Graph”** provides the following methods:

* \_\_init\_\_(self, n=0)
  + Constructs an empty graph in which the vertices can have as specifiers inters in range [0, n-1].
* readFromFile(self, filename)
  + Reads a graph from a text file “fileName” which has on the first line the number of vertices (and the number of edges) and then n lines containing 3 integers representing the start vertex, the end vertex and the cost of an edge.
  + Throws: IOError, if “fileName” cannot be opened.
* getNumberOfVertices(self)
  + Returns the number of vertices of the graph.
* parseVertices(self)
  + Returns a list of the vertices of the graph.
* isEdge(self, start, end)
  + Checks if the edge having as start vertex “start” and end vertex “end” exists in the graph.
  + Preconditions: start, end – integers
* getInDegree(self, vertex)
  + Returns the in degree of a vertex.
  + Precondition: vertex – integer
  + Throws: ValueError, if “vertex” is not a vertex of the graph.
* getOutDegree(self, vertex)
  + Returns the out degree of a vertex.
  + Precondition: vertex – integer
  + Throws: ValueError, if “vertex” is not a vertex of the graph.
* parseOutboundEdges(self, vertex)
  + Returns a list of the end vertices of the outbound edges of a vertex.
  + Precondition: vertex – integer
  + Throws: ValueError, if “vertex” is not a vertex of the graph.
* parseInboundEdges(self, vertex)
  + Returns a list of the start vertices of the inbound edges of a vertex.
  + Precondition: vertex – integer
  + Throws: ValueError, if “vertex” is not a vertex of the graph.
* getCost(self, start, end)
  + Returns the cost of the edge with the start vertex “start” and the end vertex “end”
  + Preconditions: start, end – integers
  + Throws: KeyError, if there is no such edge.
* setCost(self, start, end, value)
  + Sets the cost of the edge with the start vertex “start” and the end vertex “end” to “value”.
  + Preconditions: start, end – integers
  + Throws: KeyError, if there is no such edge
* addEdge(self, start, end, cost)
  + Adds an edge with the start vertex “start” and the end vertex “end” with cost “cost”.
  + Preconditions: start, end, cost – integers
  + Throws: ValueError, if at least one endpoint is not in the graph.
* removeEdge(self, start, end)
  + Removes an edge with the start vertex “start” and the end vertex “end”.
  + Preconditions: start, end – integers
  + Throws: KeyError, if there is no such edge.
* addVertex(self, vertex):
  + Adds a vertex “vertex” in the graph.
  + Precondition: vertex – integer
  + Throws: KeyError, if vertex already exists or is not in range [0,n-1].
* removeVertex(self, vertex):
  + Removes the vertex “vertex” from the graph.
  + Precondition: vertex – integer
  + Throws: ValueError, if there is no such vertex

**Class “Edge”** provides the following methods:

* \_\_init\_\_(self, start, end, cost=0)
  + Constructs an edge with start vertex “start”, end vertex “end” and cost “cost”.
  + Precondition: strart, end, cost – integers
* start(self), start(self,value) / end(self), end(self,value) / cost(self), cost(self,value)
  + getter / setter for start vertex / end vertex / cost

**IMPLEMENTATION**

class Edge():

def \_\_init\_\_(self, start, end, cost = 0):

self.\_\_start = start

self.\_\_end = end

self.\_\_cost = cost

@property

def start(self):

return self.\_\_start

@start.setter

def start(self, value):

self.\_\_start = value

@property

def end(self):

return self.\_\_end

@end.setter

def end(self, value):

self.\_\_end = value

@property

def cost(self):

return self.\_\_cost

@cost.setter

def cost(self, value):

self.\_\_cost = value

class DirectedGraph():

def \_\_init\_\_(self, n=1):

self.\_\_numberOfVertices = n;

self.\_\_vertices = []

self.\_\_edges = {}

def readFromFile(self, fileName):

try:

with open(fileName, "r") as file:

vertices = set()

data = file.read().split('\n')

self.\_\_numberOfVertices = int(data[0].split(' ')[0])

for edge in data[1:]:

numbers = edge.split(' ')

edgeID = numbers[0] + "to" + numbers[1]

start = int(numbers[0])

end = int(numbers[1])

cost = int(numbers[2])

self.\_\_edges[edgeID] = Edge(start, end, cost)

vertices.add(start)

vertices.add(end)

self.\_\_vertices = list(vertices)

except IOError:

raise IOError("Could not open file: " + str(fileName))

def getNumberOfVertices(self):

return len(self.\_\_vertices)

def parseVertices(self):

return self.\_\_vertices[:]

def isEdge(self, start, end):

for edge in self.\_\_edges.values():

if edge.start == start and edge.end == end:

return True

return False

def getInDegree(self, vertex):

if vertex not in self.\_\_vertices:

raise ValueError("There is no vertex with specifier: " + str(vertex))

inDegree = 0

for edge in self.\_\_edges.values():

if edge.end == vertex:

inDegree += 1

return inDegree

def getOutDegree(self, vertex):

if vertex not in self.\_\_vertices:

raise ValueError("There is no vertex with specifier: " + str(vertex))

outDegree = 0

for edge in self.\_\_edges.values():

if edge.start == vertex:

outDegree += 1

return outDegree

def parseOutboundEdges(self, vertex):

if vertex not in self.\_\_vertices:

raise ValueError("There is no vertex with specifier: " + str(vertex))

endVertices = []

for edge in self.\_\_edges.values():

if edge.start == vertex:

endVertices.append(edge.end)

return endVertices

def parseInboundEdges(self, vertex):

if vertex not in self.\_\_vertices:

raise ValueError("There is no vertex with specifier: " + str(vertex))

startVertices = []

for edge in self.\_\_edges.values():

if edge.end == vertex:

startVertices.append(edge.start)

return startVertices

def getCost(self, start, end):

edgeID = str(start) + "to" + str(end)

try:

return self.\_\_edges[edgeID].cost

except KeyError:

raise KeyError("There is no edge: " + str(start) + " -> " + str(end))

def setCost(self, start, end, value):

edgeID = str(start) + "to" + str(end)

try:

self.\_\_edges[edgeID].cost = value

except KeyError:

raise KeyError("There is no edge: " + str(start) + " -> " + str(end))

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

if start not in self.\_\_vertices or end not in self.\_\_vertices:

raise ValueError("At least one endpoint is not in the graph.")

edgeID = str(start) + "to" + str(end)

edge = Edge(start, end, cost)

self.\_\_edges[edgeID] = edge

return edgeID

def removeEdge(self, start, end):

edgeID = str(start) + "to" + str(end)

try:

del self.\_\_edges[edgeID]

except KeyError:

raise KeyError("There is no edge: " + str(start) + " -> " + str(end))

def addVertex(self, vertex):

if vertex in self.\_\_vertices:

raise ValueError("Vertex already exists: " + str(vertex))

if not(vertex >= 0 and vertex < self.\_\_numberOfVertices):

raise ValueError("Invalid specifier for vertex (must be in range [0," + str(self.\_\_numberOfVertices) + ") ): " + str(vertex))

self.\_\_vertices.append(vertex)

def removeVertex(self, vertex):

try:

self.\_\_vertices.remove(vertex)

except ValueError:

raise ValueError("There is no vertex with specifier: " + str(vertex))

toDelete = []

for edgeID, edge in self.\_\_edges.items():

if vertex == edge.start or vertex == edge.end:

toDelete.append(edgeID)

for edgeID in toDelete:

del self.\_\_edges[edgeID]