Directed Graph

Documentation 1 – Programming Language Python

Representation:

The Abstract Data Type Directed Graph is represented using three dictionaries. For each vertex, we kept a collection of its neighbours (inbound and outbound) and another dictionary which contains pairs of edges and costs.

- dict_in: it will store for every vertex a list of all the vertices which are the starting point of a edge, where the ending point is the given vertex
- dict_out: it will store for every vertex a list of all the vertices which are the starting point of a edge, where the starting point is the given vertex
- dict_cost: for every edge (vertex1, vertex2) stored as a tuple it will store the cost associated to that edge

Interface

The interface of the ADT is consisted of the following functions:

- init(numberOfVertices):
  - constructor functions
  - creates an directed graph with numberOfVertices vertices: from 0 to numberOfVertices – 1 and 0 edges

- ```
  def __init__(self, numberOFVertices = 0):
      """
      initialize the directed graph
      :param numberOFVertices: integer, the number of vertices
      """
      self._dict_in = {}
      self._dict_out = {}
      self._dict_cost = {}
      for i in range(numberOFVertices):
          self._dict_in[i] = []
          self._dict_out[i] = []
  ```

- isEdge(startVertex, endVertex)
  ```

- check if the edge (startVertex, endVertex) is an edge into the graph
- Preconditions:
  - startVertex, endVertex – integers

-
```
def isEdge(self, startVertex, endVertex):
    """
    check if the edge startVertex->endVertex is an edge
    :param startVertex: integer, the first vertex
    :param endVertex: integer, the second vertex
    :return: True if exists the edge, False otherwise
    """
    edge = (startVertex, endVertex)

    if edge in self._dict_cost.keys():
        return True
    return False
```

- isVertex(vertex)
  - check if the vertex belongs to the graph
  - Preconditions:
    - Vertex – integer

-
```
def isVertex(self, vertex):
    """
    check if the vertex exists
    :param vertex: integer
    :return: True if vertex is a vertex in graph, False otherwise
    """
    if vertex in self._dict_in.keys():
        return True
    return False
```

- addEdge(startVertex, endVertex, costOfTheEdge):
  - add the edge (startVertex, endVertex) with cost costOfTheEdge to the graph
  - Preconditons:
    - startVertex, endVertex – integer
    - the edge doesn't aready exist into the graph

-
```
def addEdge(self, startVertex, endVertex, costOfTheEdge):
    """
    add a new edge to the directed graph
    precondition: the edge doesn't exist
    :param startVertex: integer, the source vertex
```

```
        :param endVertex: integer, the destination vertex
        :param costOfTheEdge: integer, cost of the edge
        :return: -
        """
        edge = (startVertex, endVertex)
        if self.isEdge(startVertex, endVertex):
            return

        if self.isVertex(startVertex):
            self.addVertex(startVertex)
        if self.isVertex(endVertex):
            self.addVertex(endVertex)

        self._dict_out[startVertex].append(endVertex)
        self._dict_in[endVertex].append(startVertex)
        self._dict_cost[edge] = costOfTheEdge
```

- addVertex(vertex)
  - add a new vertex to the graph
  - Preconditions:
    - vertex – integer
    - the vertex doesn't already exist in the graph

- ```
  def addVertex(self, vertex):
      """
      Add a vertex to the graph
      Precondition: the vertex is not already added to the graph
      :param vertex: integer, vertex to be added
      :return: -
      """
      if self.isVertex(vertex) == True:
          return
      self._dict_in[vertex] = []
      self._dict_out[vertex] = []
  ```

- removeEdge(startVertex, endVertex)
  - remove the edge (startVertex, endVertex) from the graph
  - Preconditions:
    - startVertex, endVertex – integers
    - the edge has to exist into the graph

- ```
  def removeEdge(self, startVertex, endVertex):
      """
  ```

```
        remove an edge from the directed graph
        :param startVertex: integer, the source vertex of the edge to
    be removed
        :param endVertex: integer, the destination vertex of the edge
    to be removed
        :return: -
        """
        if self.isEdge(startVertex, endVertex):
            edge = (startVertex, endVertex)
            del self._dict_cost[edge]
        else:
            return

        self._dict_out[startVertex].remove(endVertex)
        self._dict_in[endVertex].remove(startVertex)
```

- removeVertex(vertex)
  - remove the vertex from the graph
  - Preconditions:
    - vertex – integer
    - the vertex has to exist into the graph

```
def removeVertex(self, vertex):
    """
    remove a vertex from the directed graph
    :param vertex:
    :return:
    """
    if self.isVertex(vertex):
        Neigbours = self._dict_out[vertex]
        for secondVertex in Neigbours:
            if self.isEdge(vertex, secondVertex):
                self.removeEdge(vertex, secondVertex)

        Neigbours = self._dict_in[vertex]
        for firstVertex in Neigbours:
            if self.isEdge(firstVertex, vertex):
                self.removeEdge(firstVertex, vertex)

        del self._dict_in[vertex]
        del self._dict_out[vertex]
```

- getNumberOfVertices()
  - returns the total number of vertices of the graph

- ```
  def getNumberOfVertices(self):
      """
      get the number of all the vertices
      :return: integer, the number of vertices
      """
      return len(self.getAllVertices())
  ```

- getAllVertices()
  - returns a list containing all the vertices of the graph

- ```
  def getAllVertices(self):
      """
      return a list of all the vertices
      :return: list - containing all the vertices
      """
      return self._dict_out.keys()
  ```

- getAllEdges()
  - returns a list containing tuples, which represent all the edges from the graph

- ```
  def getAllEdges(self):
      """
      return the list of all the edges
      :return: tuple of 3 elements consisting of (startVertex,
  endVertex, costOfEdge)
      """
      edges = self._dict_cost.keys()
      edgesWithCost = []
      for edge in edges:
          edgeWithCost = (edge[0], edge[1], self._dict_cost[edge])
          edgesWithCost.append(edgeWithCost)

      return edgesWithCost
  ```

- getOutboundEdges(vertex)
  - returns a list containing all the outbound edges of a given vertex
  - Precodition:
    - vertex – integer
    - vertex has to exist into the graph

- ```
  def getOutboundEdges(self, vertex):
      """
      return a list of all the outbound neighbours of a vertex
      """
  ```

```
        outboundEdges = []

        for edge in self._dict_cost.keys():
            if edge[0] == vertex:
                costOfEdge = self._dict_cost[edge]
                edgeWithCost = (edge[0], edge[1], costOfEdge)
                outboundEdges.append(edgeWithCost)

        return outboundEdges
```

- getInboundEdges(vertex)
    - returns a list containing all the inbound edges of a given vertex
    - Preconditions:
        - vertex – integer
        - vertex has to exist into the graph

- ```
  def getInboundEdges(self, vertex):
      """
      return a list of all the inbound neighbours of a vertex
      :param vertex: integer
      :return: list
      """
      inboundEdges = []

      for edge in self._dict_cost.keys():
          if edge[1] == vertex:
              costOfEdge = self._dict_cost[edge]
              edgeWithCost = (edge[0], edge[1], costOfEdge)
              inboundEdges.append(edgeWithCost)
      return inboundEdges
  ```

- getIndegreeOfAVertex(vertex)
    - returns an integer representing the number of inbound vertices of a given vertex
    - Preconditions:
        - vertex – integer
        - vertex has to exist into the graph

- ```
  def getInDegreeOfAVertex(self, vertex):
      """
      return the indegree of a vertex
      :param vertex: integer
      :return: integer
      """
      if self.isVertex(vertex):
  ```

```
            return len(self.getInboundEdges(vertex))
        return 0
```

- getOutDegreeOfAVertex(vertex)
    - returns an integer containing the number of outbound vertices of a given vertex in the graph
    - Preconditions:
        - vertex – integer
        - the vertex has to exist into the graph

```
def getOutDegreeOfAVertex(self, vertex):
    """
    return the  outdegree of a vertex
    :param vertex: integer
    :return: integer
    """
    if self.isVertex(vertex):
        return len(self.getOutboundEdges(vertex))
    return 0
```

- modifyEdge(startVertex, endVertex, newCostOfEdge)
    - change the cost of the edge (startVertex, endVertex) with the newCostOfEdge
    - Preconditions:
        - startVertex, endVertex, newCostOfEdge – integers
        - the edge (startVertex, endVertex) has to exist into the graph

```
def modifyEdge(self, startVertex, endVertex, newCostOfEdge):
    """
    modifiy the cost of the edge (startVertex, endVertex)
    :param startVertex: integer, the start of the edge
    :param endVertex: integer, the end of the edge
    :param newCostOfEdge: integer, the new cost attached to the
edge
    :return: -
    """
    if self.isEdge(startVertex, endVertex):
        edge = (startVertex, endVertex)
        self._dict_cost[edge] = newCostOfEdge
    else:
        self.addEdge(startVertex, endVertex, newCostOfEdge)
```

- copyGraph()
    - returns a new graph which represent the copy of the initial graph

- ```
  def copyGraph(self):
      """
      function which creates a copy of a graph
      :return: a copy of a directed graph given as parameter
      """
      directed_graph = DirectedGraph(self.getNumberOfVertices())
      directed_graph._dict_in = copy.deepcopy(self._dict_in)
      directed_graph._dict_out = copy.deepcopy(self._dict_out)
      directed_graph._dict_cost = copy.deepcopy(self._dict_cost)
      return directed_graph
  ```

External functions

The external functions implemented are:

- readGraphFromFile(filename)
    - read vertices and edges from the file and construct a graph according to the information read
    - return: a new graph

- ```
  def readGraphFromFile(filename):
      fin = open(filename, "r")
      line = fin.readline().strip()
      line = line.split(" ")
      numberOfVertices = int(line[0])
      numberOfEdges = int(line[1])

      graph = DirectedGraph(numberOfVertices)

      for i in range(numberOfEdges):
          line = fin.readline().strip()
          line = line.split(' ')

          startVertex = int(line[0])
          endVertex = int(line[1])
          costOfEdge = int(line[2])

          graph.addEdge(startVertex, endVertex, costOfEdge)

      return graph
  ```

- writeGraphToFile(graph, filename)

- write the vertices and edges of the graph in a file

- ```
  def writeGraphToFile(graph, filename):
      fout = open(filename, "w")

      edges = graph.getAllEdges()

      for edge in edges:
          fout.write(str(edge[0]) + " " + str(edge[1]) + " " +
  str(edge[2]) + "\n")

      vertices = graph.getAllVertices()

      for vertex in vertices:
          if graph.getOutDegreeOfAVertex(vertex) == 0 or
  graph.getInDegreeOfAVertex(vertex) == 0:
              fout.write(str(vertex) + " -1" + "\n")
  ```

- getRandomGraph(int numberOfVertices, int numberOfEdges)
  - creates a new graph with random vetices and random edges
  - return: a new graph
  - Precondition: the numberOfVertices <= numberOfVertices^2

- ```
  def getRandomGraph(numberOfVertices, numberOfEdges):
      if numberOfEdges > numberOfVertices * numberOfVertices:
          return

      graph = DirectedGraph(numberOfVertices)
      i = 0

      while i < numberOfEdges:
          startVertex = randint(0, numberOfVertices - 1)
          endVertex = randint(0, numberOfVertices - 1)
          costOfEdge = randint(0, 1000)

          if graph.isEdge(startVertex, endVertex) == False:
              graph.addEdge(startVertex, endVertex, costOfEdge)
              i += 1

      return graph
  ```