Practical Work No.1- Documentation

The class called Graph represents a directed graph and contains the following methods:

- The Constructor(__init__(self)):

The graph is represented using 3 dictionaries, named as it follows:

- dIn: dictionary in
- dOut: dictionary out
- dCost: dictionary of costs of edges
- noVertices: the current number of vertices in the graph
- noEdges: the current number of edges in the graph
- vertexList: the list of vertices in the graph

Complexity: O(1)

- loadFromFileInit(self, file_name):

“””

Loads the data of a directed graph from a text file (initial read so the isolated vertices are not taken into account)

:param: file_name -the name of the file we want to read the data from

“””””

Complexity: O(number of edges) -linear complexity for reading a list of edges

- loadFromFile(self, file_name):

“””

Loads the data of a directed graph from a text file (normal read so the isolated vertices are taken into account)- since, at this point, the graph may not contain all its vertices, we also list at the beginning the list of current vertices, beginning on the 2nd row of the file. (see Page 6 for example)

:param: file_name -the name of the file we want to read the data from

“””””

Complexity: O(number of edges + number of isolated vertices) -linear complexity for reading a list of edges and the isolated vertices, if any

- writeToFile(self, file_name):

“””

Save a graph's data to a file with a given name
:param: file_name -the name of the file we save the data to

Complexity: O(edge number + isolated vertex number)

- addVertex(self, vertex):
  """
  Add a new vertex to the graph by creating its corresponding successors and predecessors lists, incrementing the vertex number and adding it to the vertices list
  Precondition: the vertex must not exist in the graph beforehand
  :param vertex: the given vertex that we must add to the graph
  :return: 0 if the vertex already exists and cannot be added, else 1

  Complexity: O(1)

- removeVertex(self, vertex):
  """
  Remove a vertex from the graph (and the edges containing it)
  Precondition: the vertex must exist in the graph
  :param vertex: the given vertex that we must remove from the graph
  :return: Raise a ValueError if the vertex does not exist in the graph

  Complexity: O(number_of_edges) - we parse through the list of edges to see which ones contain the vertex we have just removed

- addEdge(self, StartVertex, EndVertex, edgeCost):
  """
  Adds an edge (StartVertex, EndVertex) of cost edgeCost
  Preconditions: Both the source and the target must exist
  The edge must not exist in the graph beforehand
  :param StartVertex: the source vertex of the edge
  :param EndVertex: the target vertex of the edge
  :param edgeCost: the edge's cost
  :return: Raises ValueErrors if the preconditions are not fulfilled

  Complexity: O(1)

- removeEdge(self, SourceVertex, TargetVertex):
  """
  Remove an edge identified by its target and source vertices
  Preconditions: both vertices must exist in the graph

The edge must exist in the graph
:param SourceVertex: the source vertex of the edge
:param TargetVertex: the target vertex of the edge
:return: Raises ValueErrors if the preconditions are not fulfilled
Complexity: O(1)

- EmptyGraphInit(self, id):
  “””
  Initialises an empty graph with the given number of vertices/ given list of vertices
  :param id: 0 when loading the graph from the initial file, 1 when loading the graph with isolated vertices
  :return:
  Complexity: O(number_of_vertices)

- checkifEdgeExists(self, StartVertex, EndVertex):
  “””
  Checks if an edge already exists in the graph
  :param StartVertex: the source vertex of the edge
  :param EndVertex: the target vertex of the edge
  :return: 1 if the edge already exists, else 0
  “””
  Complexity: O(n), where n is length(_dIn.keys)+length(_dOut.keys)+length(_dIn[EndVertex])

- checkifVertexExists(self, vertex):
  “””
  Checks if a given vertex exists in the graph
  :param vertex: the given vertex
  :return: 1 if the vertex exists, else 0, or a ValueError if the precondition is not fulfilled

  “””
  Complexity:
  O(n), where n is the length of the vertex list

- numberOfVertices(self):
  “””
  Returns the number of vertices in the graph
  “””

- inDegree(self, vertex):
  “””

Gets the in degree of a given vertex
Precondition: the vertex must exist in the graph beforehand
:param vertex: given vertex
:return: the in-degree of the vertex, or a ValueError if the precondition is not fulfilled

Complexity: O(1)

- outDegree(self, vertex):
  “””
  Gets the out degree of a given vertex
  Precondition: the vertex must exist in the graph beforehand
  :param vertex: given vertex
  :return: the out-degree of the vertex
  Complexity: O(1)

- retrieveCost(self, edge):
  “””
  Retrieve the information(cost) associated to an edge
  Precondition: the edge must exist in the graph beforehand
  :param edge: the edge we want to retrieve the cost of
  :return: the cost of the edge, or a ValueError if the precondition is not fulfilled
  Complexity: O(1)

- modifyCost(self, edge, value):
  “””
  Modify the information(cost) associated to an edge
  Precondition: the edge must exist in the graph beforehand
  :param edge: the edge
  :param value: the new cost of the edge
  :return: raise a ValueError if the condition is not fulfilled
  Complexity: O(1)

- checkIsolatedVertex(self, vertex):
  “””
  Checks if a vertex is an isolated one
  Precondition: the vertex must exist in the graph beforehand
  :param vertex: the vertex
  :return: 1 if the vertex is isolated (both its in-degree and out-degreeare 0), else 0. Raises a ValueError if the precondition is not fulfilled
  Complexity: O(1)

- parseVertices(self):
  “””
  Parse through the set of vertices in the graph
  Return: an iterator for the set of vertices
  “””
  Complexity: O(n) , where n is the length of VertexList

- parseInbound(self, vertex):
  “””
  Parse the set of inbound edges of a specified vertex and provide the source vertex for each edge
  Precondition: the given vertex must exist in the graph
  :param vertex: the given vertex
  :return: an iterator for the set of source vertices
  Complexity: O(n), where n is the in-degree of the given vertex

- parseOutbound(self, vertex):
  “””
  Parse the set of qoutbound edges of a specified vertex and provide the target vertex for each edge
  Precondition: the given vertex must exist in the graph
  :param vertex: the given vertex
  :return: an iterator for the set of target vertices
  Complexity: O(n), where n is the out-degree of the given vertex

- Copy(self):
  “””
  Returns a copy of the graph
  “””
  Complexity: O(1)

The UI class acts as a link between the directed graph implementation and the user interface (it is, in fact, the user interface), and, besides the command menu, it also contains a random graph generator functions, defined as follows:

- randomGraphGenerator(vertexNumber, edgeNumber, fileName):
  “””
  Generate a random graph with a specified number of vertices and edges and write it to a file whose name is given from the user input

Precondition: number of edges can be maximum the square of the number of vertices

:param vertexNumber: the number of vertices

:param edgeNumber: the number of edges

:param fileName: the name of the file the graph will be saved to

:return: -

“””

Complexity: O(n) , where n is the number of edges

----- Saving graph to file----

Initial file:

```
5 6
0 0 1
0 1 7
1 2 2
2 1 -1
1 3 8
2 3 5
```

Operation: remove vertex 2

Save graph to file

Output file:

```
4 3
0
1
3
4
0 0 1
0 1 7
1 3 8
4
```

4 3- 4 vertices and 3 edges

0, 1,3,4 -the vertices contained in the graph

(0,0,1), (0,1,7), (1,3,8) -edges contained in the graph

4 -isolated vertex