Practical Work No. 1

Documentation

This implementation of this project is done using Python language. To solve the problem statement, we need to implement a "*DirectedGraph*" class which represents a directed graph and an "*Ui*" class that represents the user interfaced menu that operates the methods of the Graph.

Implementation:

"*DirectedGraph*" class has three dictionaries representing the inbound and outbound neighbours of each vertex and one for the cost of each edge. The dictionary, self._dictOut, keeps a set of every neighbour of a given vertex x. For example, self._dictOut[x] is a set containing all the vertices that form and edge from x to that vertex. For the dictionary self._dictIn is the same explanation as above but it keeps track the inbound neighbours of a given vertex x. The dictionary self._dictCost, keeps track of the costs of every edge that it contained by the graph. The key of an item is a tuple (vertex, vertex) that represents and edge and self._dictCost [(v), v2)] is an integer representing the cost of that edge.

"Ui" class has 2 variables in the constructor: self.graph is used to pass the graph to the menu driven console, first it is initialized with an empty graph, and self.copy to remember the last copy (made by the user) of the graph.

"*DirectedGraph*" class has the following methods:

- Constructor -> creates the initial graph. It is passed a parameter "n" which is the number of nodes so it can prepare the dictionaries defined above for the following operations.
- parseX(self) -> List[int]: Returns a list of all the vertices in the graph.
- parseNout(self, x: int) -> List[int]: Returns a list of all the neighbors of vertex x.
- parseNin(self, x: int) -> List[int]: Returns a list of all the predecessors of vertex x.
- isVertex(self, x: int) -> bool: Returns True if vertex x is in the graph, False otherwise.
- addVertex(self, x: int) -> Adds vertex x to the graph, if vertex x is not already a vertex of the graph .
- removeVertex(self, x: int) -> Removes vertex x and all edges incident to it from the graph.

- isEdge(self, x: int, y: int) -> bool: Returns True if there is an edge from vertex x to vertex y, False otherwise.
- addEdge(self, x: int, y: int, c: int) ->Adds an edge from vertex x to vertex y with cost c to the graph, if edge is not already an edge of the graph.
- removeEdge(self, x: int, y: int) -> Removes the edge from vertex x to vertex y from the graph.
- setEdgeInfo(self, x: int, y: int, c: int) -> Updates the cost of the edge from vertex x to vertex y to c.
- getEdgeInfo(self, x: int, y: int) -> int: Returns the cost of the edge from vertex x to vertex y.
- getNumVertices(self) -> int: Returns the number of vertices in the graph.
- getNumEdges(self) -> int: Returns the number of edges in the graph.
- getInDegree(self, x: int) -> int: Returns the in-degree of vertex x.
- getOutDegree(self, x: int) -> int: Returns the out-degree of vertex x.
- copy(self) -> "DirectedGraph": Returns a deep copy of the graph.
- generateRandom(self, n: int, m: int) -> Generates a random directed graph with n vertices and m edges, and adds it to the graph. The graph is generated by randomly choosing two distinct vertices and adding an edge between them with a random cost between 1 and 100. If m is greater than the maximum number of edges in a directed graph with n vertices, raises a ValueError with an appropriate message.

“Ui” class provides testing and using all the methods of the graph:

- emptyGraph(self) -> Creates an empty directed graph.
- randomGraph(self) -> Creates a directed graph with random edges and costs.
- addNextVertex(self) -> Adds a vertex to the graph.
- addVertexUi(self) - >Adds a vertex to the graph based on user input.
- addEdgeUi(self) - >Adds an edge to the graph based on user input.
- removeVertexUi(self) -> Removes a vertex from the graph based on user input.
- removeEdgeUi(self) - >Removes an edge from the graph based on user input.
- updateEdgeCost(self) -> Updates the cost of an edge based on user input.
- printCost(self) - >Prints the cost of an edge based on user input.
- printInDegree(self) -> Prints the in-degree of a vertex based on user input.
- printOutDegree(self) -> Prints the out-degree of a vertex based on user input.
- printNumVertices(self) -> Prints the number of vertices in the graph.
- printNumEdges(self) -> Prints the number of edges in the graph.
- checkVertex(self) -> Checks if a vertex is in the graph based on user input. It returns a Boolean value True if there is an edge, otherwise False.

- checkEdge(self) -> Checks if there is an edge between two vertices based on user input. It returns a Boolean value True if there is an edge, otherwise False.
- printVertices(self)-> Prints all the vertices of the graph.
- printNin(self)-> Takes a vertex as input and prints the number of incoming edges to that vertex.
- printNout(self)-> Takes a vertex as input and prints the number of outgoing edges from that vertex.
- printEdges(self)-> Prints all the edges of the graph.
- printGraph(self)-> Prints the inbound neighbours and the outbound neighbours representation of each vertex of the graph.
- copyGraph(self)-> Creates a copy of the graph in self.copy.
- readGraph(self)-> Reads a graph from a predefined file with consecutive vertices.
- readGraph2(self)-> Reads a graph from a written file that can contain any vertices.
- writeGraph(self)-> Writes the graph to a file in the adjacency matrix representation.
- revertCopy(self)-> Reverts the graph to the copy of the graph that was made earlier.

In the "*Ui*" class we also have a function to print the menu, for the user to know its options and a starts the program. At the beginning the program initializes the current graph with an empty graph and gives the user various options to modify the elements. The graph can be stored in an output file and used at another run of the application.