Background:

This next file will explain the purpose of this assignment.

Issued by Ariel University, OOP course 2020.

1. *node\_info* implemented by *NodeInfo*:

The interface *node\_info* is designed to have all the qualities that are needed per one node in a weighted graph, if you’re wondering what they are:

*int id* – the node’s unique name.

*String info* – the node’s meta data.

*double tag* – the node’s temporal data which can be used when working with graphs (aka color: e, g, white, gray, black, etc.).

*Static int defultId* – makes sure each node gets a unique id and ,make sure we avoid overriding data in the HashMap.

1. **NodeInfo():** this method is made to initialize the node once creating it.
2. **NodeInfo(int id):** this method creates a new node with a specific id.
3. **getKey():** each node gets a unique key that will be her “name” – that is the most basic way to find a specific node in our collection, neighbors, and graph in general. This method allows us to know what the node “name” is.
4. **getInfo():** this method returns the node’s meta data that associated with her.
5. **getTag():** this method returns the node’s temporal data which can be used in different graphs.
6. **setTag(int t):** this method allows setting the temporal marking of the node.
7. **setInfo(String s):** this method changes the meta data associated with the node to **s**.

/\*

1. **getNi():** each node has a Collection (sort of a list) of the nodes connected to her. This method returns that Collection by the values contained in the node’s Neighbors HashMap.
2. **hasNi(int key):** this method is check if the node has a neighbor with the “name”: key.
3. **addNi(node\_data t):** this method adds the node a new neighbor – node t.
4. **removeNode(node\_data node):** this method removes an edge between the node and node\_data node. This method is implements by the fact the “having an edge” between nodes means that each node has the other node in its neighbors list.

\*/

2. *weighted\_graph* implement by *WGraph\_DS*:

The interface *weighted\_graph* is designed to help the user control his graph by these methods, if you’re wondering what they are:

*HashMap graphNodes* – a list containing all our graph’s nodes.

*HashMap adjacencyLists* – maps from each node, to a map of node to edge weight (node to his neighbors with weight).

*Int nodeCount* – counts the number of nodes in our graph.

*Int edgeCount* – counts the number of edges in our graph.

*Int modeCount* – counts any change in the inner state of the graph such as removing or adding an edge, connecting two nodes, adding a node to the graph etc.

1. **WGraph\_DS():** this method initiates all the parameters above, there for initiating a weighted graph.
2. **getNode(int key):** this method searching our graph for a node by the “name” key. By using the data structure HashMap we guaranty that searching for this node will take O(1).
3. **hasEdge(int node1, int node2):** this method checks whether node1 and node2 are neighbors or not. By using the data structure HashMap we guaranty that searching this node will take O(1).
4. **getEdge(int node1, int node2):** this method checks whether node1 and node2 are neighbors or not. If the nodes are connected – they are neighbors and has an edge between them, this method will return the weight of the edge. By using the data structure HashMap we guaranty that searching this node will take O(1).
5. **addNode(int key):** this method adds a new node to our graph. By using the data structure HashMap we guaranty that adding this node will take O(1).
6. **Connect(int node1, int node2, double w):** this method connects two nodes together, and that means of course adding node1 to node2 neighbors list and vice versa. The nodes “names” are node1 and node2. The edge’s weight will be **w**. By using HashMap at the Neighbors list in NodeData we guaranty that adding these nodes will take O(1).
7. **getV():** this method returns a shallow copy of graphNodes which represents all the graph nodes. By using the data structure HashMap we guaranty that a shallow copy of this collection will take O(1) (because the method will use a HashMap default method.
8. **getV(int node\_id):** this method returns a collection containing all of node\_id neighbors. By using the data structure HashMap we guaranty that getting this collection will take O(k) while k being the node\_id dgree.
9. **removeNode(int key):** this method removes a node by the “name” of key from our graph completely. The method goes in the node we wish to remove and finds its neighbors, afterwards it disconnects the edges between the node and its neighbors and finally removing the node from the graphNodes HashMap. By using the data structure HashMap we guaranty that removing this node from all of its neighbors will take O(n) while |V| = n.
10. **removeEdge(int node1, int node2):** this method deletes the edge connecting the two nodes by the “names” node1 and node2 and therefore disconnecting the nodes. By using HashMap at the Neighbors list in NodeData we guaranty that finding and deleting these nodes will take O(1).
11. **nodeSize():** this method returns the number of nodes and so returns: the nodeCount defined in the beginning of the class. The fact that we are returning an int from the class is in fact O(1).
12. **edgeSize():** this method returns the number of edges and so returns: the edgeCount defined in the beginning of the class. The fact that we are returning an int from the class is in fact O(1).
13. **getMC():** this method returns the number of changes that happened in the graph.

3. *weighted\_graph\_algorithms* implement by *WGraph\_Algo*:

The interface *weighted\_graph\_algorithms* is designed to help the user making more complicated actions such as initiating a weighted graph, copying a weighted graph, making sure that all the graph nodes are connected, etc., if you’re wondering what they are:

*WGraph\_DS MyGraph* – representing our graph.

1. **Graph\_Algo():** this method is made to initialize the graph once creating it.
2. **Init(graph g):** this method initiates the graph and set the algorithms operates on.
3. **Copy():** this method computes a deep copy of MyGraph.
4. **isConnected():** this method returns true only if there is a valid path from every node to any node in the graph.
5. **shortestPathDist(int src, int dest):** this method runs from node src to all its neighbors and find if src and dest are connected and if so, it returns the value of the shortest path between them by weight of edges of course. If src and dest are not connected – this method returns the value -1.
6. **shortestPath(int src, int dest):** this method returns the nodes in the shortest path between src and dest by weight of edges.
7. **Save(String file):** this method saves this weighted graph to the given file name. this method will return true if and only if the file was successfully saved.
8. **Load(String file):** this method will load a graph to *WGraph\_Algo* if the file was uploaded then MyGraph will be changed to the graph in the file and if so, the method will return **true**.