**LAB1 Documentation**

In this project I have implemented a graph class and a directed graph class which has the graph class as it’s base. In addition I have implemented some additional functions for reading a graph from a file, writing a graph to a file, creating a file and generating a random graph. Also I have implemented a GraphException class for printing error messages in case of invalid inputs.

**Graph Class Methods:**

* **init(self)**: initializes the Graph object with empty lists for vertices and edges
* **initialize\_vertices(self, vertices\_count):** initializes the vertices list with vertices\_count number of vertices
* **initialize\_edges(self, edges\_count):** sets the number of edges in the graph
* **add\_vertex(self, vertex):** adds a vertex to the graph if it doesn't already exist
* **remove\_vertex(self, vertex)**: removes a vertex from the graph if it exists
* **edges:** a property that returns the number of edges in the graph
* **vertices**: a property that returns the list of vertices in the graph
* **set\_edges(self, edge)**: a setter method for the edges property
* **parse\_vertices(self):** a generator method that yields each vertex in the graph
* **check\_vertex\_existence(self, check\_vertex)**: checks if a vertex exists in the graph
* **find\_vertex(self, vertex)**: returns True if the vertex exists in the graph
* **len(self)**: returns the number of vertices in the graph

**Directed Graph Class Methods:**

* **\_\_init\_\_(self, graph):** Initializes a new DirectedGraph object with the provided graph. Also initializes the \_inbounds, \_outbounds, and \_cost dictionaries to empty.
* **copy\_graph(self):** Returns a deep copy of the graph and the DirectedGraph object itself.
* **initialize\_graph(self, vertices\_count):** Clears the \_inbounds, \_outbounds, and \_cost dictionaries and initializes the underlying graph vertices. Calls the initialize\_cost(), initialize\_vertices\_in(), and initialize\_vertices\_out() methods.
* **initialize\_cost(self):** Initializes the \_inbounds dictionary to have empty lists for each vertex.
* **initialize\_vertices\_in(self):** Initializes the \_inbounds dictionary to have empty lists for each vertex.
* **initialize\_vertices\_out(self):** Initializes the \_outbounds dictionary to have empty lists for each vertex.
* **get\_inbound\_vertices(self, vertex):** Returns a list of inbound vertices for the given vertex.
* **get\_outbound\_vertices(self, vertex):** Returns a list of outbound vertices for the given vertex.
* **add\_vertex(self, vertex):** Adds a new vertex to the graph, and initializes empty lists in the \_inbounds and \_outbounds dictionaries for this vertex.
* **remove\_vertex(self, vertex):** Removes the given vertex from the graph, and removes all inbound and outbound connections to this vertex from other vertices.
* **add\_edge(self, startV, endV, cost):** Adds a new edge to the graph between the given startV and endV vertices, with the given cost. Also updates the \_inbounds, \_outbounds, and \_cost dictionaries.
* **delete\_edge(self, startV, endV):** Deletes the edge between the given startV and endV vertices from the graph. Also updates the \_inbounds, \_outbounds, and \_cost dictionaries.
* **add\_cost(self, edge\_one, edge\_two, cost):** Adds a new cost value to the edge between the given edge\_one and edge\_two vertices.
* **add\_vertex\_to\_inbounds(self, startV, endV):** Adds the given endV vertex to the inbound vertices list of the given startV vertex.
* **add\_vertex\_to\_outbounds(self, startV, endV):** Adds the given endV vertex to the outbound vertices list of the given startV vertex.
* **parse\_outbound\_vertices(self, vertex):** Yields all the outbound vertices of the given vertex.
* **parse\_inbound\_vertices(self, vertex):** Yields all the inbound vertices of the given vertex.
* **get\_in\_degree(self, vertex):** Returns the number of inbound vertices for the given vertex.
* **get\_out\_degree(self, vertex):** Returns the number of outbound vertices for the given vertex.
* **get\_cost(self):** Yields all the edge costs in the \_cost dictionary.
* **check\_edge\_existence(self, edge\_one, edge\_two):** Checks if there is an edge between the given edge\_one and edge\_two vertices.
* **change\_cost(self,key,cost):** Changes the cost associated with a vertex,returns none
* **change\_edge\_cost(self,startV,endV,cost):** Changes the cost associated with an edge,returns none
* **get\_edge\_cost(self,startV,endV):** Returns the cost of the given edge