**Group Members:**
**Abror Khaytbaev, Collin Chimbwanda, Mihong Lee**
**Date 11/30/2020**

**Project Title:** Dijkstra’s algorithm

**Summary of the Project:** Implement Dijkstra’s algorithm to find the shortest path between two given vertices of an undirected weighted graph.
The task is to complete 5 abstract class methods given for the Graph class that add and remove vertices, add and remove edges, calculate the shortest path, and print the shortest path.

**Implementation:**
1) We decided to use the Greedy method using priority queues with an adjacency matrix. The graph we are looking at is taken as undirected and simple. To denote the vertex pairs of the graph - (u, v) in our case, as well as the weights (eW) corresponding to the edge, we defined an “EV” class that has three public member variables, and constructed to have u and v as empty string and edgeWeight to be 0.
2) The next step is to make sure that Graph class, which inherits the 5 methods from the given GraphBase class, can access the EV object and its variables that we created in the previous step. To do this, we include it in our graph as a private member variable but as a friend class. We also included a recursive helper function to print the shortest path. We also need 2 vectors: one to store the vertices and the other for edges.
3) The first method is for adding a vertex:
The method accepts the string “label” for the vertex as an argument and should create and add a vertex to the graph with a label. No two vertices should have the same label.
So we first iterate through the vertices vector and check for that condition. If the label already exists, then don’t add it to the vertice vector. Otherwise, it does not already exist, then we can push it to the vector.
4) The second is for removing a vertex:
The method accepts the string “label” for the vertex as an argument and should remove the vertex with the “label” argument from the graph. If the “label” exists, then remove it from the vector. It also has to remove the edges between that vertex and the other vertices of the graph. If the edge is connected to u or v, remove it.
5) The third method would be to add an edge:
The method accepts three arguments string “label1”, string “label2” and long integer “weight”. It has to add an edge of value “weight” to the graph between the vertex with “label1” and the vertex with “label2”.

a) A vertex with “label1” and a vertex with “label2” must both exist. To check that we iterate through the “vertices” vector and flag those labels with boolean variables if those are noticed:

b) There must not already be an edge between those vertices, and a vertex cannot have an edge to itself. After verifying that the edges exist, we can push the edge to the "components" vector we created to store the Edge objects.

6) The fourth method would be to remove an edge:
The method accepts two arguments string "label1", string "label2". It has to remove the edge from the graph between the vertex with "label1" and the vertex with "label2". A vertex with "label1" and a vertex with "label2" must both exist and there must be an edge between those vertices.
While looping through the "components" vector that stores EV objects, we find the ones that match the label arguments, and if the match is found we remove it from the vector.

7) The last method would be to find the shortest path:
The method calculates the shortest path between the vertex called startLabel and the vertex called endLabel using Dijkstra's algorithm. We, first initialize, three variables:

a) Map that maps each label with weights - all will be initialized as infinity (LONG_MAX)
b) Map that maps each label to previous label
c) Priority queue for of all the labels in the graph, where priority is defined by the weight of the path

We iterate through the priority queue we created until its empty and in each iteration:
Using a for loop, check with all its adjacent neighbors and their weights. Inside the for loop, edge relaxation updates the shortest path when the current weight of the path is larger than through other vertex + weight of the current vertex. In the code, it updates the current reaching cost to the current vertex (weightMap[current]) to the lower reaching cost (weightMap[topMost] + edgeweight of adjacent vertex). The reason why it updates the cost is that the path through the 'topMost' can be shorter because the reaching cost of through vertex u will be lower than the cost of the current path. Using a for loop, the algorithms for the shortest paths problem, solve the problem by repeatedly using the edge relaxation.
8) Time complexity:
As we implemented the standard template library priority with adjacency matrix(vector), and removing the least in queue will be called n times. So, for each operation we would spend O(n) and for each removing operation we would spend O(n^2). Apart from that updating "verticesMap" and "weightsMap" would be performed once for each edge and if we take edge as e, it would take us O(e) time. So, knowing for a fact that edges would be less than the number for pairs of vertices, we can deduce that O(n^2) will be the running time. The algorithm is not the most efficient one, compared to using adjacency list, however.

Sources used:

"Dijkstra's Algorithm." *Cracking the Coding Interview: 189 Programming Questions and Solutions*, by Gayle Laakmann McDowell, CareerCup, LLC, 2020, pp. 633–635.