GIT Department of Computer Engineering
CSE 222/505 - Spring 2022
Homework 8 Report

Student Name: Enes Abdülhalik
Student Number: 1901042803

Question 1:

1- System Requirements:

- **MyGraph Class:**
  a. A graph that provides various operations like adding, removing, filtering, and printing vertices and edges.
  b. The system provides vertex and edge classes as an output if ever needed.
  c. The graph can be either directed or not.
  d. The graph is always weighted but providing no weights will set them to default values.
  e. The IDs of the vertices are only assigned using the newVertex method.

- **Vertex Class:**
  a. A class that represents a vertex in a graph.
  b. Has a weight, ID, label, and a collection of user-provided custom properties.
  c. The weight is a double.
  d. The ID is an integer and unique. Using the newVertex method from MyGraph class guarantees a unique ID, and is the only way to have an ID.
  e. The label is a String.
  f. The properties are stored in a map and represented by two strings for each property, a key, and a value.

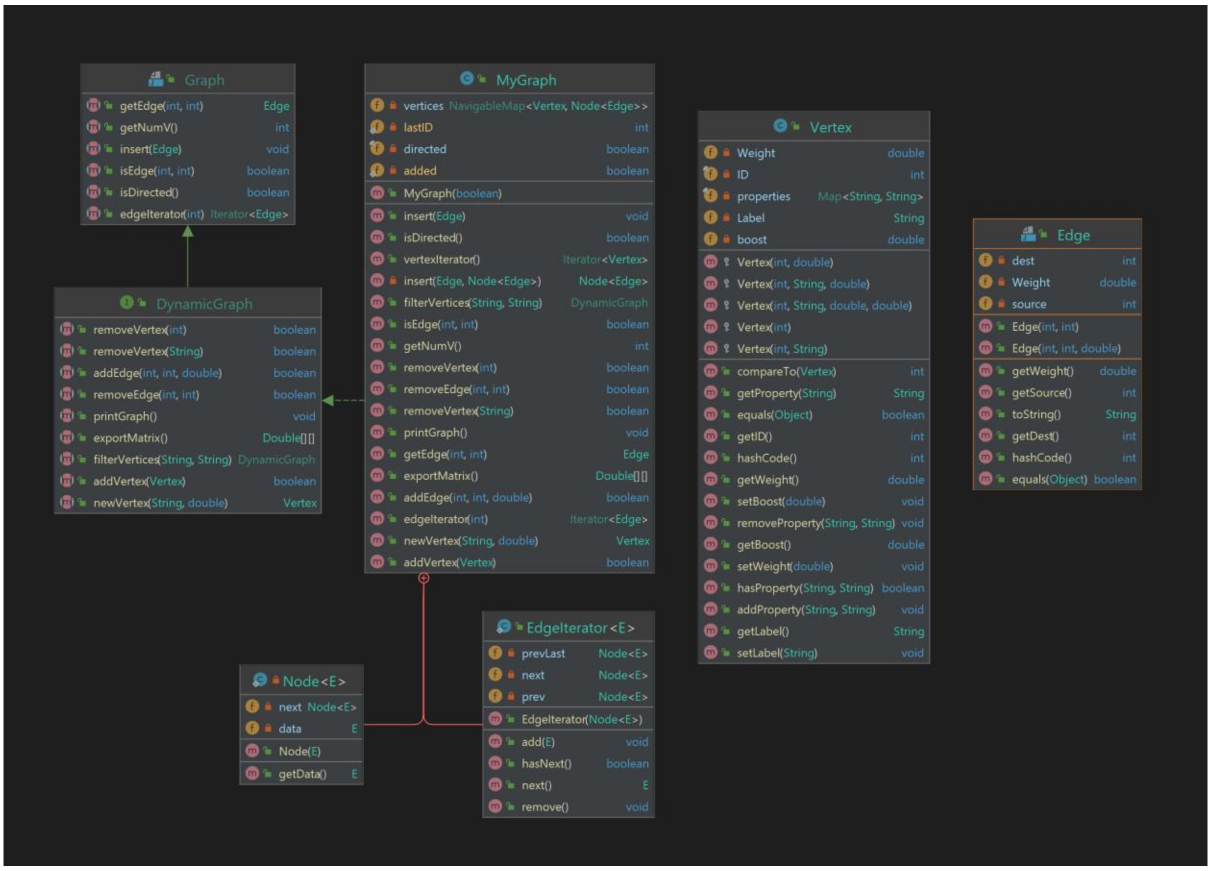
- **Edge Class:**
  a. A class that represents an edge in a graph.
  b. Has a weight, source, and destination IDs.
  c. The weight is a double.
  d. Source and destination IDs are integers.

2- Problem-Solution Approach:

The graph class stores the vertices and edges in a TreeMap, the keys of the map are vertices, and the values are edge adjacency lists. TreeMap was chosen to provide an ordered view of the vertices in the graph, and for easy delete and insert operations.

The edge adjacency lists are made of linked lists of custom nodes, so the head of a list is the head node only and could be traversed using the provided EdgeIterator class.

3- Class Diagram:



4- Test Cases:

Four vertices are added to the graph and then three edges are assigned between them.
The vertices are:

- ID = 0, label = one, weight = 500.0, properties = "farm", "big".
- ID = 1, label = two, weight = 5000.0, properties = "farm", "big".
- ID = 2, label = three, weight = 50.0, no properties.
- ID = 3, label = four, weight = 5.0, no properties.
- ID = 4, label = five, weight = 5000.0, properties = "farm", "big".

The edges are:

- Source = 0, Destination = 1, weight = 10
- Source = 1, Destination = 3, weight = 10
- Source = 2, Destination = 0, weight = 10
- Source = 4, Destination = 1, weight = 10
- Source = 3, Destination = 4, weight = 10
- Source = 4, Destination = 2, weight = 10
- Source = 0, Destination = 4, weight = 9
-

After that, the vertex with the label "two" is removed, then the matrix form is printed to the terminal, then the adjacency list form is printed, and then a filtered graph with the vertices that have the property ("farm", "big") is printed in the adjacency list form.

5- Running and Results:

The driver code is designed to be launched by a makefile in a Linux system.

```
[Infinity, 10.0, Infinity, Infinity, 9.0]
[Infinity, Infinity, Infinity, 10.0, Infinity]
[10.0, Infinity, Infinity, Infinity, Infinity]
[Infinity, Infinity, Infinity, Infinity, 10.0]
[Infinity, 10.0, 10.0, Infinity, Infinity]

list
0 1 10.000000
0 4 9.000000
1 3 10.000000
2 0 10.000000
3 4 10.000000
4 1 10.000000
4 2 10.000000

list
0 1 10.000000
0 4 9.000000
4 1 10.000000
```

6- Time Complexity: **n = number of vertices, e = number of edges from a vertex**

```
public MyGraph(boolean directed) {
    this.directed = directed;
    vertices = new TreeMap<>();
    added = false;
}
            T(n) = Θ(1)

public void insert(Edge edge) {
    added = false;
    if (!vertices.containsKey(new Vertex(edge.getSource())) ||          O(logn)
        !vertices.containsKey(new Vertex(edge.getDest())))              O(logn)
        return;

    Node<Edge> target = vertices.get(new Vertex(edge.getSource()));         O(logn)
    vertices.put(new Vertex(edge.getSource()), insert(edge, target));       O(logn) + O(e)
    if (!directed) {
        Edge reverse = new Edge(edge.getDest(), edge.getSource(), edge.getWeight());        Θ(1)
        target = vertices.get(new Vertex(reverse.getSource()));                    O(logn)
        vertices.put(new Vertex(reverse.getSource()), insert(reverse, target));            O(logn) + O(e)
    }
}
                                    T(n, e) = O(logn) + O(e) = O(e)

private Node<Edge> insert (Edge edge, Node<Edge> node) {
    if (node == null) {         Θ(1)
        added = true;
        return new Node<>(edge);         Θ(1)
    } else if (node.getData().getWeight() > edge.getWeight()) {          Θ(1)
        added = true;
        Node<Edge> newNode = new Node<>(edge);          Θ(1)
        newNode.next = node;          Θ(1)
        return newNode;         Θ(1)
    } else if (node.getData().equals(edge)) {    Θ(1)
        return node;
    } else {
        node.next = insert(edge, node.next);        T(e - 1)
        return node;
    }
}
                                T(e) = Θ(1) + T(e - 1) = O(e)

public boolean isEdge(int source, int dest) {
    // Search the edge list of the source for the given edge.
    Iterator<Edge> iter = edgeIterator(source);    O(logn)
    while (iter.hasNext())  O(e)
        if (iter.next().getDest() == dest)
            return true;

    return false;
}
                        T(e, n) = O(logn) + O(e) = O(e)

public Edge getEdge(int source, int dest) {
    // Search the edge list of the source for the given edge.
    Iterator<Edge> iter = edgeIterator(source);        O(logn)
    while (iter.hasNext()) {              O(e)
        Edge target = iter.next();
        if (target.getDest() == dest)
            // The edge was found, return it.
            return target;
    }
    return null;
}
                        T(e, n) = O(logn) + O(e) = O(e)

public Iterator<Edge> edgeIterator(int source) {
    return new EdgeIterator<>(vertices.get(new Vertex(source)));        O(logn)
}
                            T(e, n) = O(logn)

public Vertex newVertex(String label, double weight) {
    ++lastID;
    return new Vertex(lastID, label, weight);
}
                            T(n) = Θ(1)

public boolean addVertex(Vertex new_vertex) {
    // Do not add if it already exists.
    if (vertices.containsKey(new_vertex))       O(logn)
        return false;

    vertices.put(new_vertex, null);       O(logn)
    return true;
}
                        T(n) = O(logn)

public boolean addEdge(int vertexID1, int vertexID2, double weight) {
    // Do not add if it already exists.
    if (isEdge(vertexID1, vertexID2))      O(e)
        return false;

    insert(new Edge(vertexID1, vertexID2, weight));           O(e)
    boolean added1 = added;
    // Insert again to the destination if the graph is directed.
    if (!directed)
        insert(new Edge(vertexID2, vertexID1, weight));      O(e)

    return added || added1;
}
                                T(e) = O(e)

public boolean removeEdge(int source, int dest) {
    Iterator<Edge> iter = edgeIterator(source);        O(logn)
    while (iter.hasNext())       O(e)
        if (iter.next().getDest() == dest) {
            iter.remove();
            return true;
        }

    if (!directed) {
        // The graph is not directed, remove from both sides.
        iter = edgeIterator(dest);       O(logn)
        while (iter.hasNext())       O(e)
            if (iter.next().getDest() == source) {
                iter.remove();
                return true;
            }
    }
    return false;
}
                        T(e, n) = O(logn) + O(e) = O(e)

public boolean removeVertex(int vertexID) {
    if(vertices.remove(new Vertex(vertexID)) != null) {            O(logn)
        for (Map.Entry<Vertex, Node<Edge>> temp2 : vertices.entrySet())        O(n)
            if (temp2.getValue() != null) {
                if (temp2.getValue().data.getDest() == vertexID)
                        // An edge pointing to the deleted vertex is the head.
                        temp2.setValue(temp2.getValue().next);
                else {
                        Iterator<Edge> iter = new EdgeIterator<>(temp2.getValue());      O(logn)
                        while (iter.hasNext())            O(e)
                        if (iter.next().getDest() == vertexID)
                            iter.remove();
                }
            }
        return true;
    }
    return false;
}
                                T(e, n) = O(e*n)

public boolean removeVertex(String label) {
    // Find the target vertex.
    for (Vertex vertex: vertices.keySet())              O(n)
        if (vertex.getLabel().equals(label)) {
            // The target vertex was found.
            // Remove it.
            vertices.remove(vertex);            O(logn)
            // And remove all the edges pointing to it.
            for (Map.Entry<Vertex, Node<Edge>> temp2 : vertices.entrySet())   O(n)
                if (temp2.getValue() != null) {
                    if (temp2.getValue().data.getDest() == vertex.getID())
                        // An edge pointing to the deleted vertex is the head.
                        temp2.setValue(temp2.getValue().next);

                    else {
                        Iterator<Edge> iter = new EdgeIterator<>(temp2.getValue());      O(logn)
                        while (iter.hasNext())            O(e)
                            if (iter.next().getDest() == vertex.getID())
                                iter.remove();
                    }
                }
            return true;
        }
    return false;
}
                                        T(e, n) = O(e*n)

public DynamicGraph filterVertices(String key, String filter) {
    DynamicGraph result = new MyGraph(directed);          Θ(1)
    // Add the matching vertices.
    for (Vertex temp : vertices.keySet())          O(n)
        if (temp.hasProperty(key, filter))     Θ(1)
            result.addVertex(temp);       O(logn)
    // Add all the edges that are within the chosen vertices.
    for (Map.Entry<Vertex, Node<Edge>> temp : vertices.entrySet())     O(n)
        if (temp.getKey().hasProperty(key, filter)) {    Θ(1)
            Iterator<Edge> iter = edgeIterator(temp.getKey().getID());      O(logn)
            while (iter.hasNext())       O(e)
                result.insert(iter.next());       O(e)
        }
    return result;
}
                                        T(e, n) = O(n*e^2)

public Double[][] exportMatrix() {
    Double[][] result = new Double[vertices.size()][vertices.size()];         Θ(1)
    for (int i = 0; i < vertices.size(); ++i)       O(n)
        for (int j = 0; j < vertices.size(); ++j)       O(n)
            result[i][j] = Double.POSITIVE_INFINITY;         Θ(1)

    // Insert every edge found to the matrix according to it's connections.
    int i = 0, j;
    for (Map.Entry<Vertex, Node<Edge>> temp : vertices.entrySet()) {        O(n)
        // Iterate through an edge list.
        Iterator<Edge> iter = new EdgeIterator<>(temp.getValue());   Θ(1)
        while (iter.hasNext()) {          O(e)
            Edge target = iter.next();
            j = 0;
            for (Vertex temp2 : vertices.keySet()) {      O(n)
                if (temp2.getID() == target.getDest())
                    break;
                ++j;
            }
            result[i][j] = target.getWeight();
        }
        ++i;
    }
    return result;
}
                                        T(e, n) = O(e*n^2)

public void printGraph() {
    System.out.print("list\n");
    for (Map.Entry<Vertex, Node<Edge>> temp : vertices.entrySet()) {        Θ(n)
        Iterator<Edge> iter = new EdgeIterator<>(temp.getValue());       Θ(1)
        while (iter.hasNext()) {          Θ(e)
            Edge target = iter.next();
            System.out.printf("%d %d %f\n", target.getSource(), target.getDest(), target.getWeight());
        }
    }
}
                                                            T(n) = Θ(e*n)
```

Question 2:

1- System Requirements:

a. The system takes a graph and does BFS and DFS traversals until all vertices connected to the vertex with the smallest ID are visited, then calculates the total distance to all the vertices from the starting vertex of each traversal and returns the difference between them.
b. If the result was positive, then DFS method found the least total distances, if it was negative, then BFS had the least total distances.

2- Problem-Solution Approach:

The system uses a recursive algorithm for each of BFS and DFS traversing methods, and the total distance from the starting vertex is given as a parameter for every recursive call. Each traversing method has two functions, one recursive helper and another wrapper function.

The system has a total of five functions:

- public static double BFS_DFS_Dif(MyGraph InputGraph);

This is the main function in the system. It checks the validity of the input, calls the BFS method and stores the total distance from it, then calls DFS method and subtracts its total distance from the total of BFS and returns the result.

- public static double calc_BFS (int start);

This function Returns the total distance from a starting point to all the vertices in the graph when using BFS traversal.

- private static void calc_BFS(double distance, int current, int level);

This function keeps traversing the vertices in Breadth-First method until all the vertices in the current component are visited. The states of the visited vertices are updated as they are visited.

- public static double calc_DFS (int start);

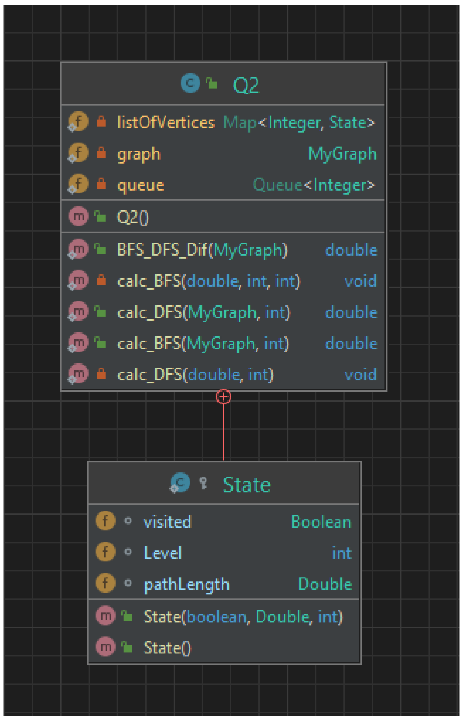
This function Returns the total distance from a starting point to all the vertices in the graph when using DFS traversal.

- private static void calc_DFS(double distance, int current);

This function keeps traversing the vertices in Depth-First method until all the vertices in the current component are visited. The states of the visited vertices are updated as they are visited.

An inner class (state) is used to store the information about a vertex. The states of the vertices are stored in a hashmap that has the key as the ID of a vertex, and the value is its state.

3- Class Diagram:


Q2
listOfVertices Map<Integer, State>
graph MyGraph
queue Queue<Integer>
Q2()
BFS_DFS_Dif(MyGraph) double
calc_BFS(double, int, int) void
calc_DFS(MyGraph, int) double
calc_BFS(MyGraph, int) double
calc_DFS(double, int) void
State
visited Boolean
Level int
pathLength Double
State(boolean, Double, int)
State()

4- Test Cases:

The method is applied on the same final graph from the first question.

5- Running and Results:

The driver code is designed to be launched by a makefile in a Linux system.

Output: -18.0

BDF method found the shortest total distance.

6- Time Complexity: **n = number of vertices, E = the total number of edges**

```
public static double BFS_DFS_Dif(MyGraph InputGraph) {
    if (InputGraph == null)
        return 0;
    graph = InputGraph;
    int start = graph.vertexIterator().next().getID();
    return calc_BFS(graph, start) - calc_DFS(graph, start);      O(n*(E + n))
}
                        T(E, n) = O(n*(E + n))
```

```
private static double calc_BFS (int start) {
    listOfVertices = new HashMap<>();           Θ(1)
    Iterator<Vertex> vIter = graph.vertexIterator();        Θ(1)
    while (vIter.hasNext())          O(n)
        listOfVertices.put(vIter.next().getID(), new State());      O(n)

    queue = new LinkedList<>();      Θ(1)
    listOfVertices.put(start, new State(true, 0.0, 0));      O(n)
    calc_BFS(0.0, start, 0);      O(n*(E + n))
    double result = 0;
    for (Map.Entry<Integer, State> temp : listOfVertices.entrySet()) {     O(n)
        if (temp.getValue().visited)         Θ(1)
            result += temp.getValue().pathLength;
    }

    return result;          Θ(1)
}
                                T(E, n) = O(n*(E + n)) = Ω(E*n)

private static void calc_BFS(double distance, int current, int level) {
    Iterator<Edge> iter = graph.edgeIterator(current);        O(logn)
    while (iter.hasNext()) {     Θ(E)
        Edge edge = iter.next();    Θ(1)
        State vertex = listOfVertices.get(edge.getDest());          O(n)

        if (vertex.visited) {       Θ(1)
            if (vertex.Level == level + 1)        Θ(1)
                vertex.pathLength = Math.min(vertex.pathLength, edge.getWeight() + distance);   Θ(1)
        } else {
            queue.add(edge.getDest());       Θ(1)
            vertex.pathLength = edge.getWeight() + distance;         Θ(1)
            vertex.Level = level + 1;         Θ(1)
            vertex.visited = true;        Θ(1)
        }
    }
    if (!queue.isEmpty()) {      Θ(1)
        State nextTarget = listOfVertices.get(queue.peek());          O(n)
        calc_BFS(nextTarget.pathLength, queue.remove(), nextTarget.Level);        C(n-1)
    }
}
                                T(E, n) = O(E*n) + C(n) = O(n*(E + n)) = Ω(E + n)
            (Recursive part) C(n) = O(n) + C(n - 1) = O(n^2) = Ω(n)
```

```
private static double calc_DFS () {
    /* initialize the list of vertices with the default states. */
    listOfVertices = new HashMap<>();            Θ(1)
    Iterator<Vertex> vIter = graph.vertexIterator();        Θ(1)
    while (vIter.hasNext())          O(n)
        listOfVertices.put(vIter.next().getID(), new State());       O(n)

    listOfVertices.put(start, new State(true, (double) 0, 0));
    calc_DFS((double) 0, start);        O(n*E) = Ω(E)
    double result = 0;
    for (Map.Entry<Integer, State> temp : listOfVertices.entrySet()) {      O(n)
        if (temp.getValue().visited)
            result += temp.getValue().pathLength;
    }
    return result;
}
                                        T(E, n) = O(n*(E + n))

private static void calc_DFS (double distance, int current) {
    Queue<Edge> queue = new LinkedList<>();          Θ(1)
    Iterator<Edge> iter = graph.edgeIterator(current);        O(logn)
    while (iter.hasNext())                 Θ(E)
        queue.add(iter.next());         Θ(1)

    while (!queue.isEmpty()) {          Θ(E)
        Edge edge = queue.remove();          Θ(1)
        State vertex = listOfVertices.get(edge.getDest());     O(n)

        if (!vertex.visited) {          Θ(1)
            vertex.pathLength = distance + edge.getWeight();              Θ(1)
            vertex.visited = true;                      Θ(1)
            calc_DFS(distance + edge.getWeight(), edge.getDest());          T(E, n-1)
        }
    }
}
                        T(n, E) = O(n*E) + C(n) = O(n*E) + O(nlogn) = O(n*E) = Ω(E)
    (Recursive part) C(n) = O(logn) + C(n - 1) = O(nlogn)
```

Question 3:

1- System Requirements:

a. The system takes a graph and an ID to the start vertex, then calculates the total distance from the start towards all the vertices in the graph using a modified version of Dijkstra's algorithm.
b. The class vertex has a boost value already.
c. A boost value should not be larger than the weight of an edge in a vertex. This will result in a negative distance.

2- Problem-Solution Approach:

The difference between this algorithm and the original one is that this algorithm uses a user defined double value assigned to a vertex called boost value. This value is subtracted from the total distance of a path when passing through the vertex holding it.

This system uses a similar approach as the BFS method described above and uses the same state class. The difference is that the map listOfVertices instead of being a map with the key as vertex and the value as state, it has the key as the ID of the vertex, and the value as an entry that stores references to the vertex and its state.

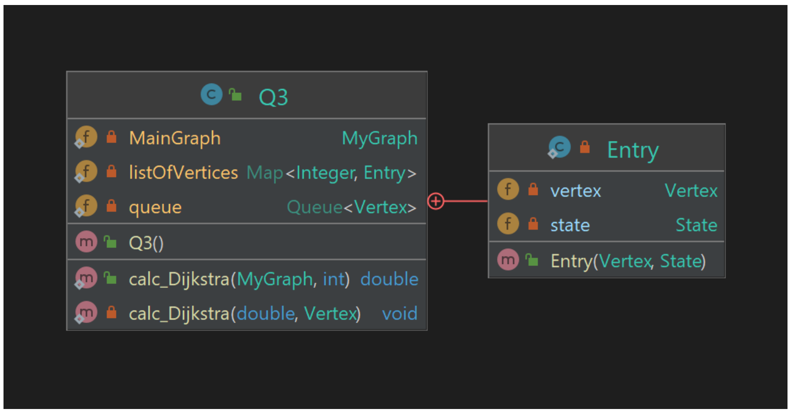
The system has two functions:

- public static double calc_Dijkstra (MyGraph graph, int targetV);
  This is the main method in the system. It checks the validity of the input, calls the helper method, and calculates the total distance from the listOfVertices map and returns it.

- private static void calc_Dijkstra(double distance, Vertex current);
  This recursive method traverses the graph received by the wrapper method and updates the state of every vertex with its new state.

An inner class (Entry) is used to store the references to a vertex and its state. The entries are stored in a hashmap that has the key as the ID of a vertex, and the value is an entry.

3- Class Diagram:


Q3
MainGraph MyGraph
listOfVertices Map<Integer, Entry>
queue Queue<Vertex>
Q3()
calc_Dijkstra(MyGraph, int) double
calc_Dijkstra(double, Vertex) void
Entry
vertex Vertex
state State
Entry(Vertex, State)

4- Test Cases:

Randomly generated 20 vertices with boost values are added to the previous graph, and 100 randomly generated edges are added too. Then the result from Breadth-First traversal is printed, then the result from this system is printed. The second result should be smaller than the first one.

5- Running and Results:

The driver code is designed to be launched by a makefile in a Linux system.

Output:

```
2178.0
1253.5262901550516
```

The modified Dijkstra's method found the shorter total distance.

6- Time Complexity: **n = number of vertices, E = the total number of edges**

```
public static double calc_Dijkstra (MyGraph graph, int targetV) {
    MainGraph = graph;
    /* initialize the list of vertices with the default states. */
    listOfVertices = new HashMap<>();
    Iterator<Vertex> vIter = MainGraph.vertexIterator();     Θ(1)
    while (vIter.hasNext()) {   O(n)
        Vertex vertex = vIter.next();       Θ(1)
        listOfVertices.put(vertex.getID(), new Entry(vertex, new State()));  O(n)
    }

    queue = new LinkedList<>();                    Θ(1)
    Entry entry = listOfVertices.get(targetV);  O(n)
    entry.state.visited = true;              Θ(1)
    entry.state.pathLength = 0.0;            Θ(1)
    calc_Dijkstra(0.0, entry.vertex);        O(n*(E + n)) = Ω(E + n)

    double result = 0.0;          Θ(1)
    for (Entry temp : listOfVertices.values())        O(n)
        if (temp.state.visited)          Θ(1)
            result += temp.state.pathLength;        Θ(1)

    return result;
}
```

$$T(E, n) = O(n*(E + n)) = \Omega(E*n)$$

```
private static void calc_Dijkstra(double distance, Vertex current) {
    Iterator<Edge> iter = MainGraph.edgeIterator(current.getID());        O(logn)
    while (iter.hasNext()) {      Θ(E)
        Edge edge = iter.next();          Θ(1)
        Entry entry = listOfVertices.get(edge.getDest());       O(n)

        if (entry.state.visited) {
            entry.state.pathLength =
                        Math.min(entry.state.pathLength, edge.getWeight() +
                                        distance - current.getBoost());              Θ(1)
        } else {
            queue.add(entry.vertex);                  Θ(1)
            entry.state.pathLength = edge.getWeight() + distance - current.getBoost();           Θ(1)
            entry.state.visited = true;            Θ(1)
        }
    }

    if (!queue.isEmpty()) {
        Entry nextTarget = listOfVertices.get(queue.peek().getID());        O(n)
        calc_Dijkstra(nextTarget.state.pathLength, queue.remove());      C(n - 1)
    }
}
```

T(E, n) = O(E*n) + C(n) = O(n*(E + n)) = Ω(E + n)

(Recursive part) C(n) = O(n) + C(n - 1) = O(n^2) = Ω(n)