CSE222 HW8

System Requirements

The DynamicGraph class is implanted in such way to provide dynamic number of vertices. In normal Graph implementation the vertices are reached by their index values but in our case, this is impractical since if they were stored in an ordered collection their indexes would change after each insertion or removal of vertices so indexes couldn't correlate with uniqueness. To solve this issue an ID field is declared in Vertex class and it's hashing, and equality are tied to that value of it.

To be able to add, get, and remove collection of edges of a vertex LinkedHashMap is used to store them instead of HashMap since its iteration is $\theta(n)$ while HashMap's is $\Omega(n)$ where n is number of vertices. This is because it is said in implementation comments that LinkedHashMap iterates linearly to its size, but HashMap iterates linearly to its capacity. Adding and removing are constant time in both while HashMap being a little faster.

The edge collection type that is used to store them correlated to vertex ids in LinkedHashMap is TreeMap which implements Red-Black Tree. This is done to be able to add, get, and remove individual edges fast and iteration is still linear.

Vertices are stored in a HashMap corresponding to their IDs. They could be stored in LinkedHashMap as keys but in that case anytime connected edges of that vertex were tried to be reached by user a new vertex object had to be created since user was giving an ID not a vertex object. So instead LinkedHashMap key type is Integer and vertices are stored in another HashMap with their IDs being keys.

The ID of a vertex is chosen not by the graph but the user. This is because other way around is just too impractical. If it was for the system to tell what ID is to be chosen it had to track what it can and can't assign, since IDs should be unique. Even if unique ID creation was guaranteed if user accesses vertices with their ID, they must know their ID. If that were to be done by telling the user the pattern that IDs are generated, that is both unsecure and will result in inefficiency in means of number of available IDs. The reason why newVertex method was not implemented is also this. And if user tries to add a vertex with an ID such that there exists a vertex in graph that has same ID, the new one is just put in the place of it and will keep the edges.

Q1

1. Time Complexity Analysis

a. addVertex

```
public void addVertex(Vertex vertex) {
    TreeMap<Integer, Edge> priorEdges = adjacencyList.get(vertex.getId());
    adjacencyList.put(vertex.getId(), priorEdges == null ? new TreeMap<>() : priorEdges);
    vertices.put(vertex.getId(), vertex);
}
```

Both adjacencyList and vertices are hash maps, therefore addVertex takes constant time.

b. addEdge

```
public void addEdge(int vertexID1, int vertexID2, double weight) {
    adjacencyList.get(vertexID1).put(vertexID2, new Edge(vertexID1, vertexID2, weight));
    if (!directed) {
      adjacencyList.get(vertexID2).put(vertexID1, new Edge(vertexID2, vertexID1, weight));
    }
}
```

Contains and get operations of adjacencyList are executed in constant time but edge is put in a Red-Black Tree map, therefore its time complexity is $O(log\ n)$.

c. *removeEdge*

```
public void removeEdge(int vertexID1, int vertexID2) {
    adjacencyList.get(vertexID1).remove(vertexID2);
    if (!directed) {
        adjacencyList.get(vertexID2).remove(vertexID1);
    }
}
```

Get function of adjacencyList is constant time since it's a hash map, but the removal of edge (id-edge pair) is $O(log\ n)$ since maximum number of edges is linearly proportional to the number of vertices and id-edge pair is stored in Red-Black Tree.

d. *removeVertex(int vertexID)*

```
public void removeVertex(int vertexID) {
    if (directed) {
        adjacencyList.remove(vertexID);
        adjacencyList.forEach((vertex, edges) -> edges.remove(vertexID));
    }
    else {
        for (Integer destID : adjacencyList.remove(vertexID).keySet()) {
            adjacencyList.get(destID).remove(vertexID);
        }
    }
    vertices.remove(vertexID);
}
```

The removal of the given vertex and its edges are constant, but here is the catch, the other vertices may have had edges towards it. So, to also update them adjacencyList must be iterated and every edge whose destination is the removed vertex also must be deleted. Iteration of adjacencyList is $\theta(n)$ but for every vertex an edge deletion operation is executed, and since it is Red-Black Tree and max edge count is n, it takes $O(log\ n)$ each time. So, it takes $O(n\ log\ n)$ to remove a vertex *from a directed graph.*

You see if graph is not directed then we can get every vertex that we must update from the edges of removed vertex. The maximum number of edges it can have is linearly proportional to number of vertices therefore for-loop by itself is $O(n)$, the removal of an edge still takes $O(log\ n)$ due to reasons mentioned above. So, for a non-directed graph it is $O(n\ log\ n)$, it may seem the same but the loop that encapsulate the edge removal is guaranteed to be executed less than the previously mentioned case. Therefore, it is faster.

The best case for a directed graph is $\theta(n)$ where there are no edges between any vertices. But the best case for a non-directed graph is $\theta(1)$ where the removed vertex has no edge.

e. *removeVertex(string label)*

```
public void removeVertex(String label) {
    BiConsumer<Integer, TreeMap<Integer, Edge>> remover;

    if (directed) {
        remover =
                (vertex, edges) -> {
                    if (vertices.get(vertex).getLabel().equals(label)) {
                        adjacencyList.remove(vertex);
                    }
                    else {
                        edges.forEach((dest, edge) -> {
                            if (vertices.get(dest).getLabel().equals(label)) {
                                edges.remove(dest);
                            }
                        });
                    }
                };
    }
    else {
        remover =
                (vertex, edges) -> {
                    if (vertices.get(vertex).getLabel().equals(label)) {
                        for (Integer dest : adjacencyList.remove(vertex).keySet()) {
                            if (vertices.get(dest).getLabel().equals(label)) {
                                adjacencyList.remove(dest);
                            }
                            else {
                                adjacencyList.get(dest).remove(vertex);
                            }
                        }
                    }
                };
    }

    adjacencyList.forEach(remover);
}
```

For this function too there are two different processes for directed and non-directed graphs. The time complexity for directed graph is:

$$\theta(n)\Big[O(m) + \Big[1 \oplus O(n)\big[O(m) + [O(\log n) \oplus 1]\big]\Big]\Big]$$
$$= O(mn) + \Big[\theta(n) \oplus \big[O(mn^2) + [O(n^2 \log n) \oplus O(n^2)]\big]\Big]$$

Where $a \oplus b$ indicates that *either* **a** *or* **b**. And m is length of label. Since equal method of String is also $\Omega(1)$ for directed graph $T_b(n) = \theta(n)$ and $T_w(n, m) = O(mn^2) + O(n^2 \log n)$.

Time complexity of non-directed graph is:

$$\theta(n)\Big[O(m) + \Big[1 \oplus O(n)\big[O(m) + [1 \oplus O(\log n)]\big]\Big]\Big]$$

Which seems about the same. And indeed, the best case time complexities are the same, but at worst case it is faster for non-directed graph because we can directly access to destination vertices of a deleted vertex and delete the edge between them resulting in one decrease of edges. So, if there were n-1 edges of deleted vertex then they would also be deleted from the edges of destination vertices, which would decrease the total number of edges by n-1 and then n-2 and then n-3… decreasing de decrement each iteration and that would make $\frac{n^2-n}{2} \log n$ instead of $n^2 \log n$ which is worst case for directed graph (apart from String.equal).

f. filterVertices

```
public MyGraph filterVertices(String key, String filter) {
    MyGraph filtered = new MyGraph(directed);

    vertices.forEach((integer, vertex) -> {
        String value = vertex.getValue(key);
        if (value != null && value.equals(filter)) {
            filtered.addVertex(vertex);
        }
    });

    for (Integer vertexID : filtered.vertices.keySet()) {
        Iterator<Edge> edgeIterator = edgeIterator(vertexID);

        while (edgeIterator.hasNext()) {
            Edge edge = edgeIterator.next();
            if (filtered.vertices.get(edge.getDest()) != null) {
                filtered.addEdge(vertexID, edge.getDest(), edge.getWeight());
            }
        }
    }

    return filtered;
}
```

The time complexity is

$$\theta(1) + \theta(n)O(m) + \theta(n)\left[O(n)[O(log\, n) \oplus 1]\right]$$

Which would make best case $\theta(n)$ with no edges, and worst case is actually $O(n\, log(n-1)!)$ since $log(n-1)! = \sum_{i=1}^{n-1} log(n-i)$ and not $O(n^2\, log\, n)$.

g. exportMatrix

```
public double[][] exportMatrix() {
    int vertexCount = getNumV();
    double[][] adjacencyMatrix = new double[vertexCount + 1][vertexCount + 1];

    for (int i = 1; i < vertexCount + 1; i++) {
        for (int j = 1; j < vertexCount + 1; j++) {
            adjacencyMatrix[i][j] = Double.POSITIVE_INFINITY;
        }
    }

    HashMap<Integer, Integer> idToIndex = new HashMap<>(vertexCount);
    LinkedList<Collection<Edge>> edgesList = new LinkedList<>(vertexCount;

    final int[] i = {0};
    BiConsumer<Integer, TreeMap<Integer, Edge>> vertexMatrixer = (vertex, edges) -> {
        idToIndex.put(vertex, i[0]);
        adjacencyMatrix[i[0] + 1][0] = vertex;
        adjacencyMatrix[0][i[0] + 1] = vertex;
        edgesList.add(edges.values());
        i[0]++;
    };

    adjacencyList.forEach(vertexMatrixer);

    for (Collection<Edge> edges : edgesList) {
        for (Edge edge : edges) {
            int source = edge.getSource();
            int dest = edge.getDest();

            int sourceIndex = idToIndex.get(source);
            int destIndex = idToIndex.get(dest);

            adjacencyMatrix[sourceIndex + 1][destIndex + 1] = edge.getWeight();
        }
    }

    return adjacencyMatrix;
}
```

This method takes quadratic time due to initialization of double matrix. vertexMatrixer is linear time and nested for-loops at the end are $O(n^2)$.

h. printGraph

Q3

The Time Complexity of Modified Dijkstra

To retain the minimum distance from given vertex to all other vertices and their leading vertices, it is implemented as a class and they are fields of it. Both minimum distances and leading vertices are kept in hash maps due to dynamic ID's. They are only created upon object construction but can accessed anytime.

Since method is long enough to not look good here a pseudo code is written to explain instead.

```
void execute(MyGraph graph, Vertex source) {
        create empty set to store traversed vertices
        create empty map for minimum distances
        create empty map for leading vertices
        create tree set for vertices yet to be traversed including source

        while there are vertices to traverse
                get vertex v with minimum distance
                add v to traversed vertices

                for every edge v has
                        get destination vertex u of edge
                        if u is traversed get to next edge
                        get current minimum distance d to u

                        if d is zero
                                set minimum distance of u to weight of edge
                                set leading vertex of u to v
                                add u to next vertices to be traversed
                        else
                                calculate potential new distance p also considering boosting

                                if d > p
                                        update min distance leading to u
                                        update leading vertex to u
                                        rearrange u in next vertices with its updated values
}
```

The time complexity is with n being number of vertices

$$O(n)\left[O(log\, n) + O(n)[1 \oplus O(log\, n)]\right]$$

Which would make best case constant time with source having no edges. And worst case would be $O(n^2\, log\, n)$ with every vertex having edges to every other vertex.