GIT Department of Computer Engineering

CSE 222/505 - Spring 2022

Homework 8 Report

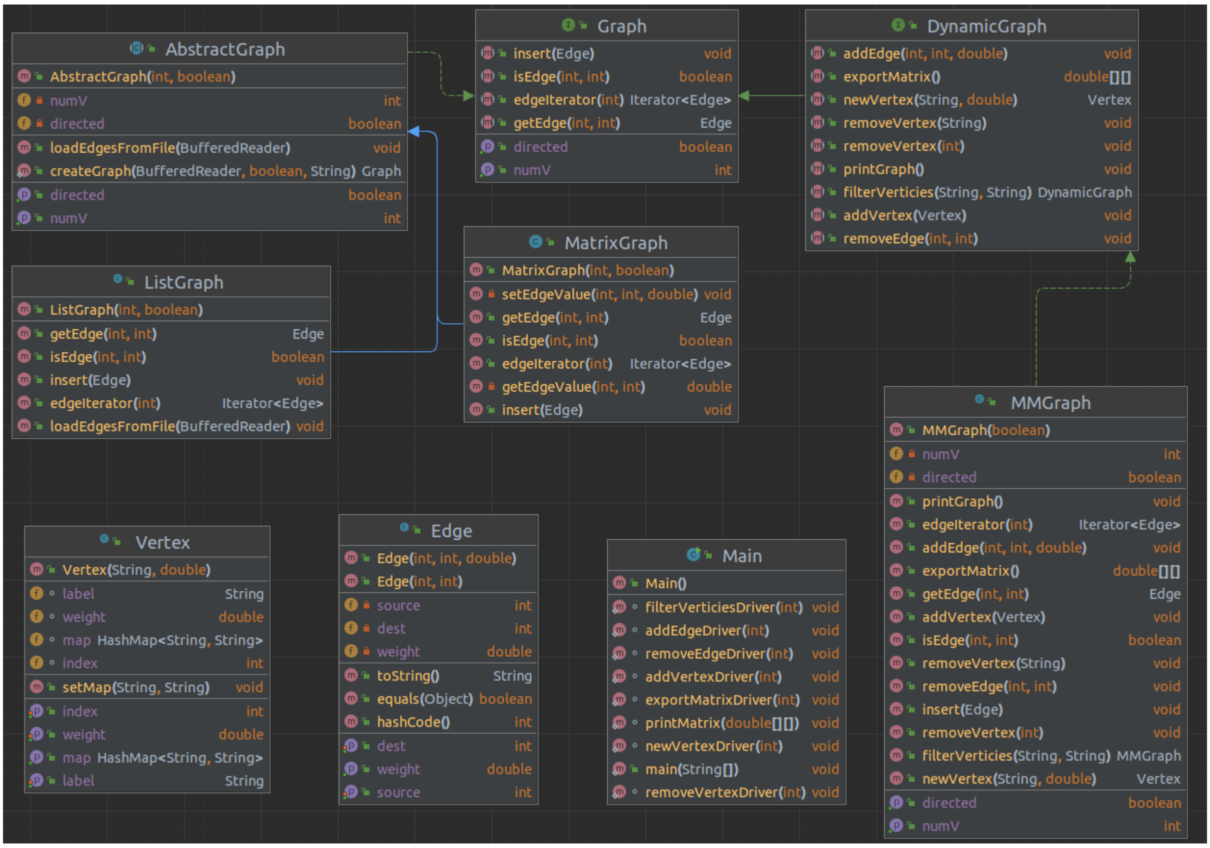
Mustafa MERT

200104004006

1. SYSTEM REQUIREMENTS

To use this program you need an operating system with jdk17 and jre1. We need memory to store strings, binary trees and arrays. Also integers for test cases. Also while this program works it uses CPU, RAM and disk from the pc, if they are better they can give faster results to you.

2. CLASS DIAGRAMS



3. PROBLEM SOLUTION APPROACH

In the first question I created methods and implemented them as specified in the homework pdf.

- newVertex method just creates new a vertex
- addVertex adds the vertex to graph.
- addEdge adds an edge to graph.
- removeEdge searches for the given edge and delete it.
- removeVertex(index) firstly searches for related edges with this vertex then deletes the vertex.
- removeVertex(label) uses removeVertex(index) and deletes all the vertices with given label.
- filterVertices duplicates the graph then deletes all vertices other than given key filter.
- exportMatrix returns two dimensional array for the graph
- printGraph prints the graph

4. TEST CASES

Test Case #	Test Case Description	Test Data	Expected Result	Actual Result	Pass/Fail
newVertex	Measuring time consumption for differnt number of elements	10000 vs 1000000 element	expected relation is close time consumptions because of $\Theta(1)$ complexity	image 5.1	Pass
addVertex	Measuring time consumption for differnt number of elements	100 vs 10000 element	expected relation is close time consumptions because of $\Theta(1)$ complexity	image 5.2	Pass
addEdge	Measuring time consumption for differnt number of elements	100 vs 10000 element	expected relation is close time consumptions because of $\Theta(1)$ complexity	image 5.3	Pass
removeEdge	Measuring time consumption for differnt number of elements	100 vs 10000 element	expected relation is quadratic time consumptions because of $O(n^2)$ complexity	image 5.4	Pass
removeVertex	Measuring time consumption for differnt number of elements	100 vs 10000 element	expected relation is quadratic time consumptions because of $O(n^2)$ complexity	image 5.5	Pass
filterVertices	Measuring time consumption for differnt number of elements	100 vs 10000 element	expected relation is quadratic time consumptions because of $O(n^2)$ complexity	image 5.6	Pass
exportMatrix	Measuring time consumption for differnt number of elements	100 vs 10000 element	expected relation is quadratic time consumptions because of $O(n^2)$ complexity	image 5.7	Pass

5. RUNNING AND RESULTS

Q1)

newVertex(String label, double weight)

```
public Vertex newVertex(String label, double weight) {
    return  new Vertex(label, weight);
}
```

time complexity of this function is theoretically $\Theta(1)$

practical result (image 5.1)

```
10000       number of elemnts   =>   0.79032ms
10000       number of elemnts   =>   0.75225ms
10000       number of elemnts   =>   0.74584ms
10000       number of elemnts   =>   0.76669ms
10000       number of elemnts   =>   0.95515ms
10000000        number of elemnts   =>   0.7732ms
10000000        number of elemnts   =>   0.7725ms
10000000        number of elemnts   =>   0.79925ms
10000000        number of elemnts   =>   0.80776ms
10000000        number of elemnts   =>   0.85295ms
```

addVertex(Vertex new_vertex)

```
public void addVertex(Vertex new_vertex) {
    new_vertex.index = numV++;
    verticies.add(new_vertex);
    edges.add(new LinkedList<>());
}
```

time complexity of this function is theoretically $\Theta(1)$

practical result (image 5.2)

```
100       number of elemnts   =>   0.06781ms
100       number of elemnts   =>   0.05479ms
100       number of elemnts   =>   0.09625ms
100       number of elemnts   =>   0.1355ms
100       number of elemnts   =>   0.04798ms
10000       number of elemnts   =>   1.3909ms
10000       number of elemnts   =>   1.1956ms
10000       number of elemnts   =>   1.46842ms
10000       number of elemnts   =>   1.10437ms
10000       number of elemnts   =>   0.98709ms
```

addEdge(int vertexID1, int vertexID2, double weight)

```
public void addEdge(int vertexID1, int vertexID2, double weight) {

    edges.get(vertexID1).add(new Edge(vertexID1,vertexID2,weight));
    if(!this.directed){
        edges.get(vertexID2).add(new Edge(vertexID2,vertexID1,weight));
    }
}
```

time complexity of this function is theoretically $\Theta(1)$

practical result (image 5.3)

```
100          elemnts   =>   0.07204ms
100          elemnts   =>   0.1031ms
100          elemnts   =>   0.1604ms
100          elemnts   =>   0.12794ms
100          elemnts   =>   0.12183ms
10000        elemnts   =>   12.59367ms
10000        elemnts   =>   8.23848ms
10000        elemnts   =>   6.17766ms
10000        elemnts   =>   7.46786ms
10000        elemnts   =>   7.29553ms
```

removeEdge(int vertexID1, int vertexID2)

```
public void removeEdge(int vertexID1, int vertexID2) {

    Edge target = new Edge(vertexID1, vertexID2, Double.POSITIVE_INFINITY);
    for (Edge edge : edges.get(vertexID1)) {
        if (edge.equals(target))
            edges.remove(edge);
    }
    if (!this.directed){
        Edge target2 = new Edge(vertexID2, vertexID1, Double.POSITIVE_INFINITY);
        for (Edge edge : edges.get(vertexID2)) {
            if (edge.equals(target))
                edges.remove(edge);
        }
    }
}
```

time complexity of this function is theoretically $O(n^2)$

practical result (image 5.4)

```
100          elemnts   =>   31.27018ms
100          elemnts   =>   20.43522ms
100          elemnts   =>   5.26746ms
100          elemnts   =>   3.7963ms
100          elemnts   =>   3.44304ms
10000        elemnts   =>   11049.733ms
10000        elemnts   =>   9876.543ms
10000        elemnts   =>   9624.447ms
10000        elemnts   =>   9707.418ms
10000        elemnts   =>   9713.561ms
```

removeVertex(int vertexID)

```java
public void removeVertex(int vertexID) {
    verticies.remove(vertexID);
    edges.remove(vertexID);
    numV--;
    for (int i = 0 ; i<numV; i++){
        for (Edge edge : edges.get(i)){
            if (edge.getSource() == vertexID || edge.getDest() == vertexID)
                edges.get(i).remove(edge);
            else{
                if (edge.getSource() > vertexID)
                    edge.setSource(edge.getSource()-1);
                if (edge.getDest() > vertexID)
                    edge.setDest(edge.getDest()-1);
            }
        }
    }
    for (int i = 0; i<numV ; i++){
        verticies.get(i).index = i;
    }
}
```

time complexity of this function is theoretically $O(n^2)$

practical result (image 5.5)

```
100        elemnts   =>   12.08441ms
100        elemnts   =>   11.89446ms
100        elemnts   =>   10.09605ms
100        elemnts   =>   3.69137ms
100        elemnts   =>   2.87112ms
10000      elemnts   =>   1517.1816ms
10000      elemnts   =>   1236.7426ms
10000      elemnts   =>   1235.7864ms
10000      elemnts   =>   1217.8083ms
10000      elemnts   =>   1236.3444ms
```

removeVertex(String label)

```java
public void removeVertex(String label) {
    for(int i= numV-1; i >= 0; i--){
        if(verticies.get(i).label.equals(label))
            removeVertex(i);
    }
}
```

uses removeVertex(int vertexID) method time complexity is same

filterVertices(String key, String filter)

```
public MMGraph filterVerticies(String key, String filter) {
    MMGraph newGraph = new MMGraph( directed: false);
    newGraph = this;

    for (int i = numV-1 ; i >=0; i--) {
        if (!verticies.get(i).map.containsKey(key) && !verticies.get(i).map.containsValue(filter))
            newGraph.removeVertex(i);
    }

    return newGraph;
}
```

time complexity of this function is theoretically $O(n^2)$

practical result (image 5.6)

exportMatrix()

```
public double[][] exportMatrix() {
    double nmatrix[][] = new double[numV][numV];
    for (int i=0; i<numV ; i++)
        for (int j=0; j<numV; j++)
            nmatrix[i][j] = -1;

    for (int source = 0; source < numV; source++)
        for(Edge edge: edges.get(source)){
            int dest = edge.getDest();
            nmatrix[source][dest] = edge.getWeight();
        }
    return nmatrix;
}
```

time complexity of this function is theoretically $O(n^2)$

practical result (image 5.7)

```
100          elemnts   =>   5.67988ms
100          elemnts   =>   4.88908ms
100          elemnts   =>   6.86203ms
100          elemnts   =>   4.7922ms
100          elemnts   =>   2.66105ms
10000        elemnts   =>   4681.434ms
10000        elemnts   =>   4153.6157ms
10000        elemnts   =>   4698.6826ms
10000        elemnts   =>   4181.189ms
10000        elemnts   =>   3236.688ms
```