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

Ahmet Kadir Aksu
200104004114

TIME COMPLEXITY

Q1)

1. newVertex (string label, double weight): ***Constant time O(1)***

2. addVertex (Vertex new_vertex): ***Constant time O(1)***

3. addEdge (int vertexID1, int vertexID2, double weight): ***Constant time O(1)***

4. removeEdge (int vertexID1, int vertexID2): ***Constant time O(1)***

5. removeVertex (int vertexID): ***Constant time O(1)***

6. removeVertex (string label): ***O(n)***

   Searching the vertices with the given label.

7. filterVertices (string key, string filter):

   For n vertices and k edges O(n*k)

8. exportMatrix():

   For n vertices and k edges O(n*k)

9. printGraph(): Print the graph in adjacency list format

   For n vertices and k edges O(n*k)

1. SYSTEM REQUIREMENTS

Q1)

MyGraph class builds a graph which uses *hashmap* to keep its vertices and edges. Edges inside the hashmap, will be stored in a *linked list.* Also, we need *iterator* to traverse the edges. We will get this data structures from java.util.

MyGraph class implements **DynamicGraph** interface which extends the **Graph** interface from our course book.

Also, we need an **Edge** class for our edges which is used from our course book.

Q2)

To create and test a graph we need the class MyGraph from Q1.

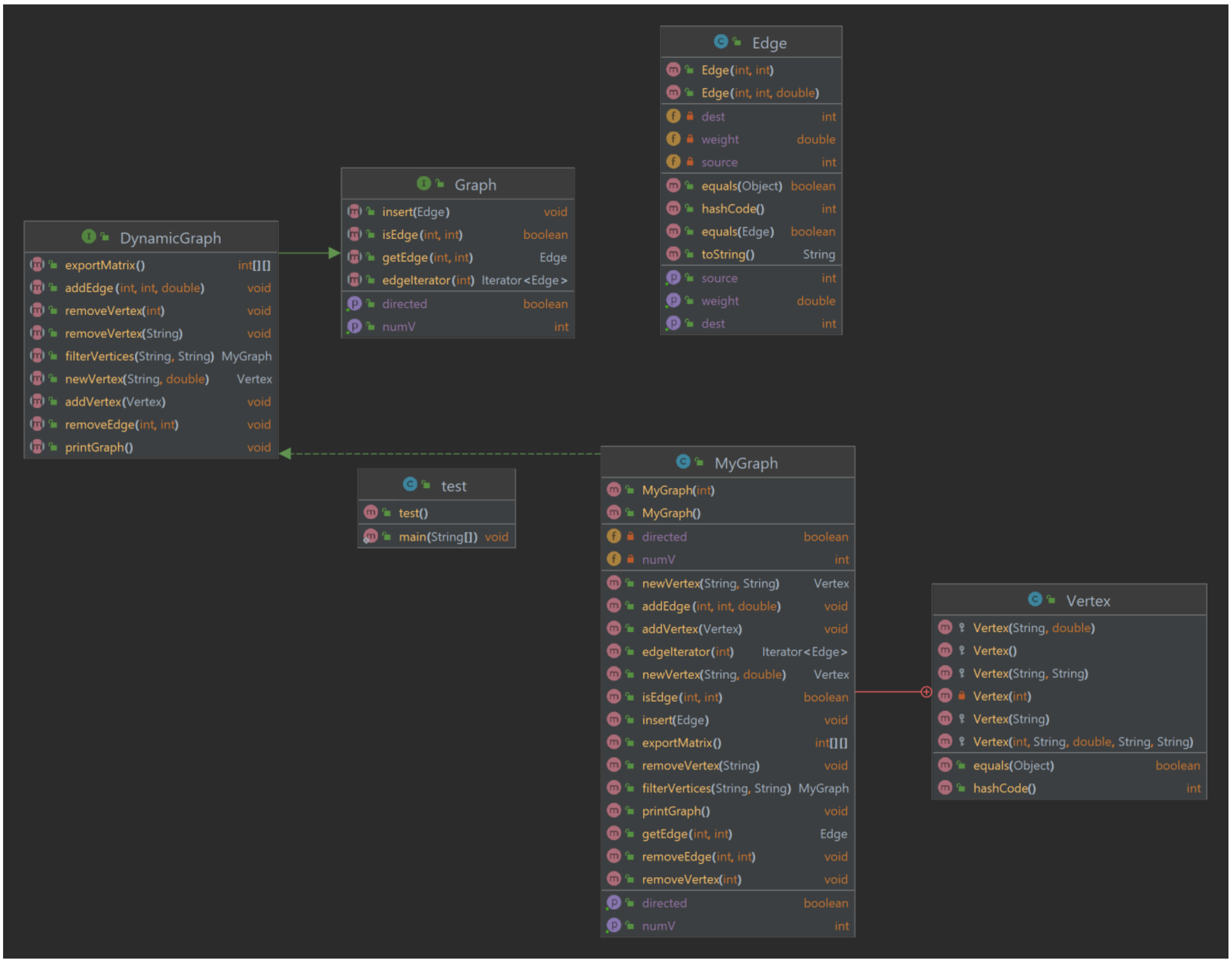
findDifference method calculates the difference of bfs and dfs traversals. This method is inside the **DifferenceOfTraversals** class. In this method, we calculate bfs distance by using **BreadthFirstSearch** class and dfs distance by using **DepthFirstSearch** class.

Q3)

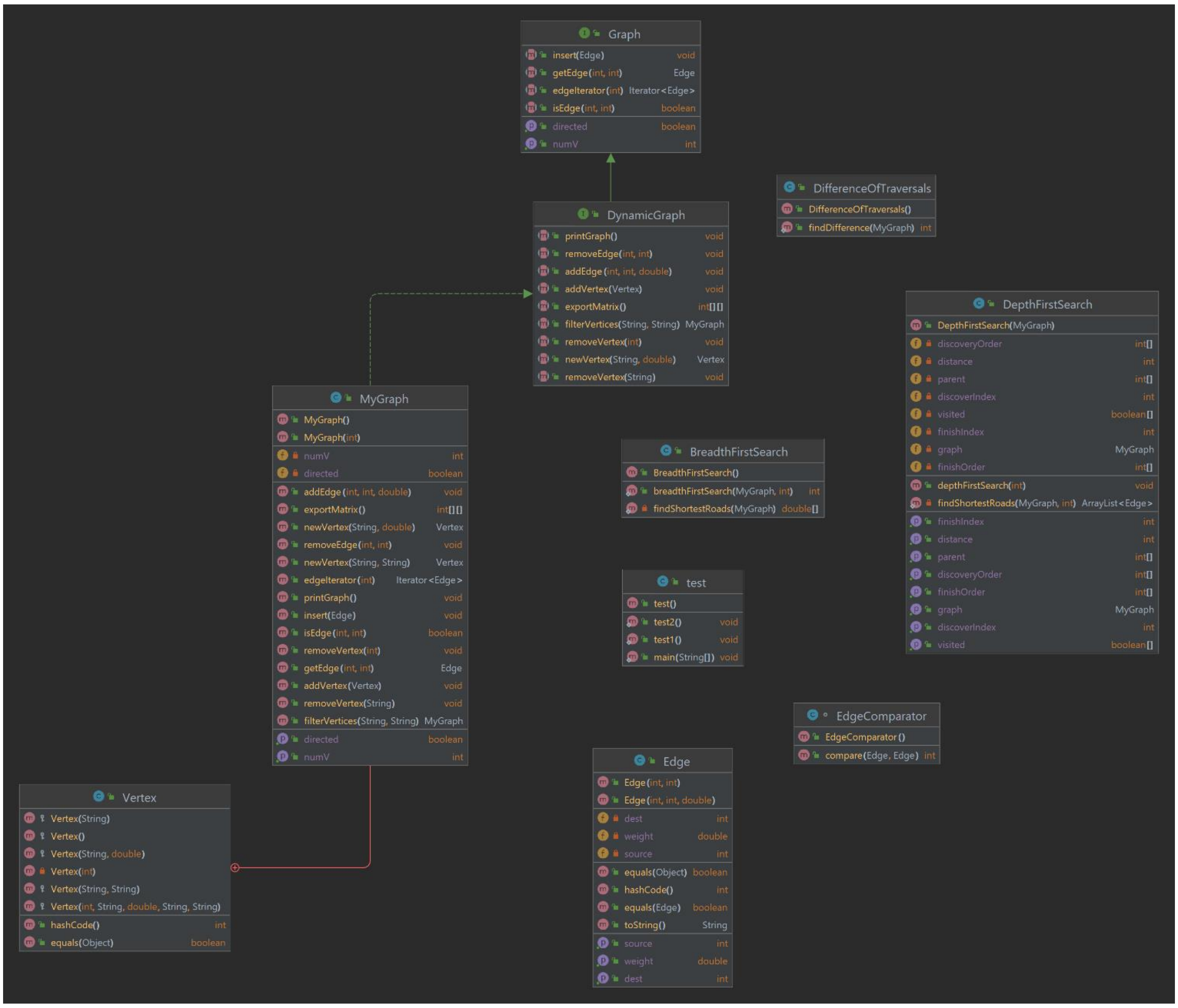
To create and test a graph we need the class MyGraph from Q1.
DijkstraWithBoost method uses the dijkstrasAlgorithm method from **Dijkstra** class. DijkstraWithBoost method is inside the **BoostedDijkstra** class.

2. **CLASS DIAGRAMS**

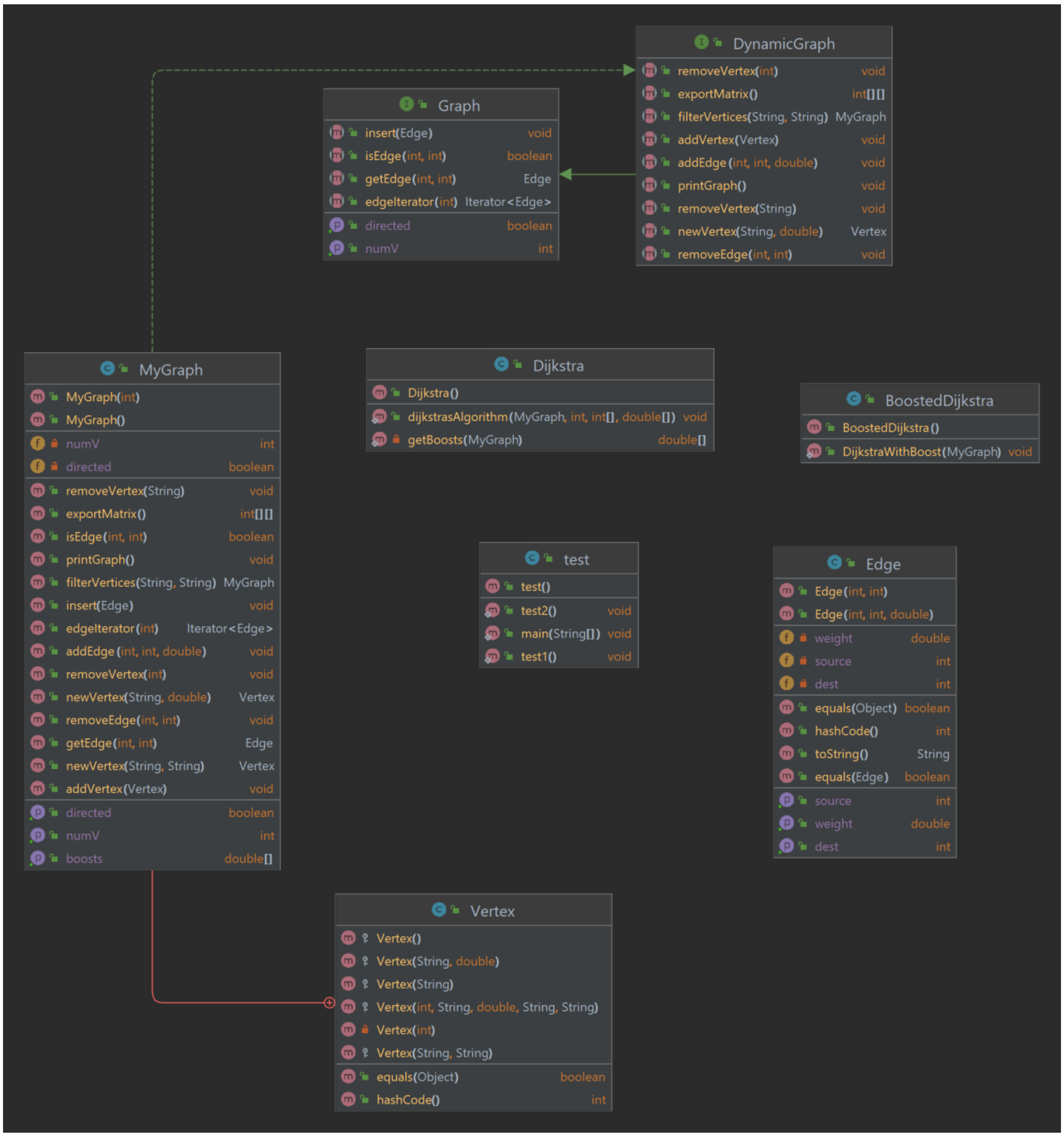
Q1



Q2

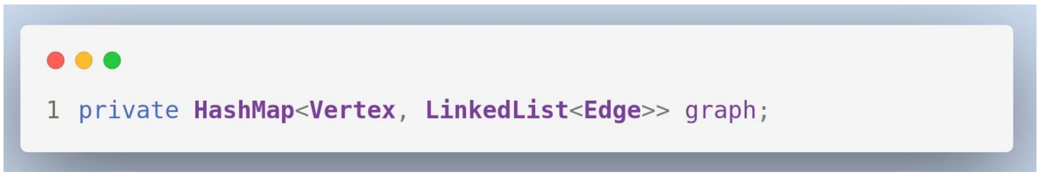

Graph
insert(Edge) void
getEdge(int, int) Edge
edgeIterator(int) Iterator<Edge>
isEdge(int, int) boolean
directed boolean
numV int
DifferenceOfTraversals
DifferenceOfTraversals()
findDifference(MyGraph) int
DynamicGraph
printGraph() void
removeEdge(int, int) void
addEdge(int, int, double) void
addVertex(Vertex) void
exportMatrix() int[][]
filterVertices(String, String) MyGraph
removeVertex(int) void
newVertex(String, double) Vertex
removeVertex(String) void
DepthFirstSearch
DepthFirstSearch(MyGraph)
discoveryOrder int[]
distance int
parent int[]
discoverIndex int
visited boolean[]
finishIndex int
graph MyGraph
finishOrder int[]
depthFirstSearch(int) void
findShortestRoads(MyGraph, int) ArrayList<Edge>
finishIndex int
distance int
parent int[]
discoveryOrder int[]
finishOrder int[]
graph MyGraph
discoverIndex int
visited boolean[]
MyGraph
MyGraph()
MyGraph(int)
numV int
directed boolean
addEdge(int, int, double) void
exportMatrix() int[][]
newVertex(String, double) Vertex
removeEdge(int, int) void
newVertex(String, String) Vertex
edgeIterator(int) Iterator<Edge>
printGraph() void
insert(Edge) void
isEdge(int, int) boolean
removeVertex(int) void
getEdge(int, int) Edge
addVertex(Vertex) void
removeVertex(String) void
filterVertices(String, String) MyGraph
directed boolean
numV int
BreadthFirstSearch
BreadthFirstSearch()
breadthFirstSearch(MyGraph, int) int
findShortestRoads(MyGraph) double[]
test
test()
test2() void
test1() void
main(String[]) void
EdgeComparator
EdgeComparator()
compare(Edge, Edge) int
Vertex
Vertex(String)
Vertex()
Vertex(String, double)
Vertex(int)
Vertex(String, String)
Vertex(int, String, double, String, String)
hashCode() int
equals(Object) boolean
Edge
Edge(int, int)
Edge(int, int, double)
dest int
weight double
source int
equals(Object) boolean
hashCode() int
equals(Edge) boolean
toString() String
source int
weight double
dest int

Q3


DynamicGraph
removeVertex(int) void
exportMatrix() int[][]
filterVertices(String, String) MyGraph
addVertex(Vertex) void
addEdge(int, int, double) void
printGraph() void
removeVertex(String) void
newVertex(String, double) Vertex
removeEdge(int, int) void
Graph
insert(Edge) void
isEdge(int, int) boolean
getEdge(int, int) Edge
edgeIterator(int) Iterator<Edge>
directed boolean
numV int
MyGraph
MyGraph(int)
MyGraph()
numV int
directed boolean
removeVertex(String) void
exportMatrix() int[][]
isEdge(int, int) boolean
printGraph() void
filterVertices(String, String) MyGraph
insert(Edge) void
edgeIterator(int) Iterator<Edge>
addEdge(int, int, double) void
removeVertex(int) void
newVertex(String, double) Vertex
removeEdge(int, int) void
getEdge(int, int) Edge
newVertex(String, String) Vertex
addVertex(Vertex) void
directed boolean
numV int
boosts double[]
Dijkstra
Dijkstra()
dijkstrasAlgorithm(MyGraph, int, int[], double[]) void
getBoosts(MyGraph) double[]
BoostedDijkstra
BoostedDijkstra()
DijkstraWithBoost(MyGraph) void
test
test()
test2() void
main(String[]) void
test1() void
Edge
Edge(int, int)
Edge(int, int, double)
weight double
source int
dest int
equals(Object) boolean
hashCode() int
toString() String
equals(Edge) boolean
source int
weight double
dest int
Vertex
Vertex()
Vertex(String, double)
Vertex(String)
Vertex(int, String, double, String, String)
Vertex(int)
Vertex(String, String)
equals(Object) boolean
hashCode() int

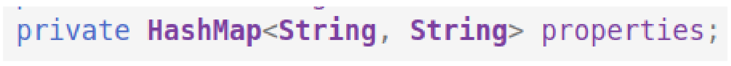
3. PROBLEM SOLUTION APPROACH

<u>**Q1**</u>- Created an inner class **Vertex**, then to keep the vertex and edges of the graph, used a hashmap which keeps vertices as key and edges(Linked List) as value.



```java
private HashMap<Vertex, LinkedList<Edge>> graph;
```

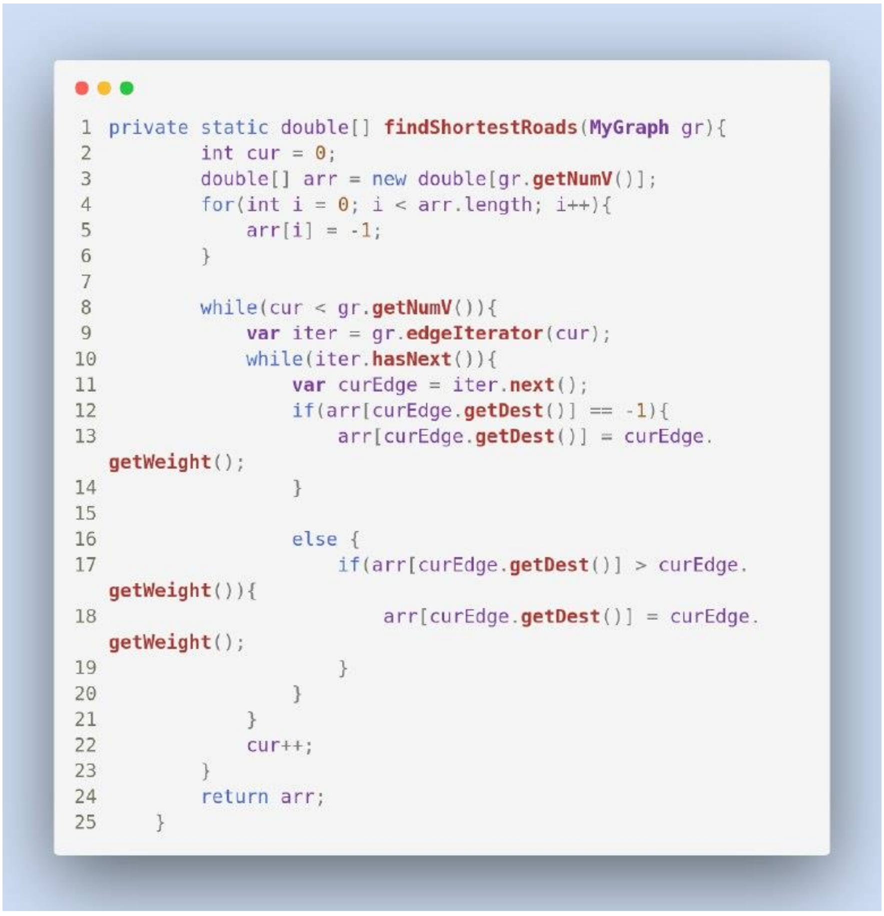
To keep properties of vertices used hashmap again.



```java
private HashMap<String, String> properties;
```

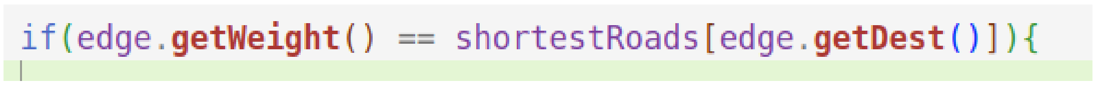
For the <u>filterVertices</u> method, created a new **MyGraph** object and iterate the main object, if a **vertex** has a matching property, just add to the new **MyGraph** object and finally return to it.

<u>**Q2**</u>- <u>findDifference</u> method, firstly passes the graph into <u>breadFirstSearch</u> method and gets bfs distance, then passes the graph into <u>DepthFirstSearch</u> method and get dfs distance. Then calculate the difference. If the result is negative, converts to positive.
"If there are more than one alternative to access a vertex at a specific level
during the BFS, the shortest alternative should be considered". To provide this I



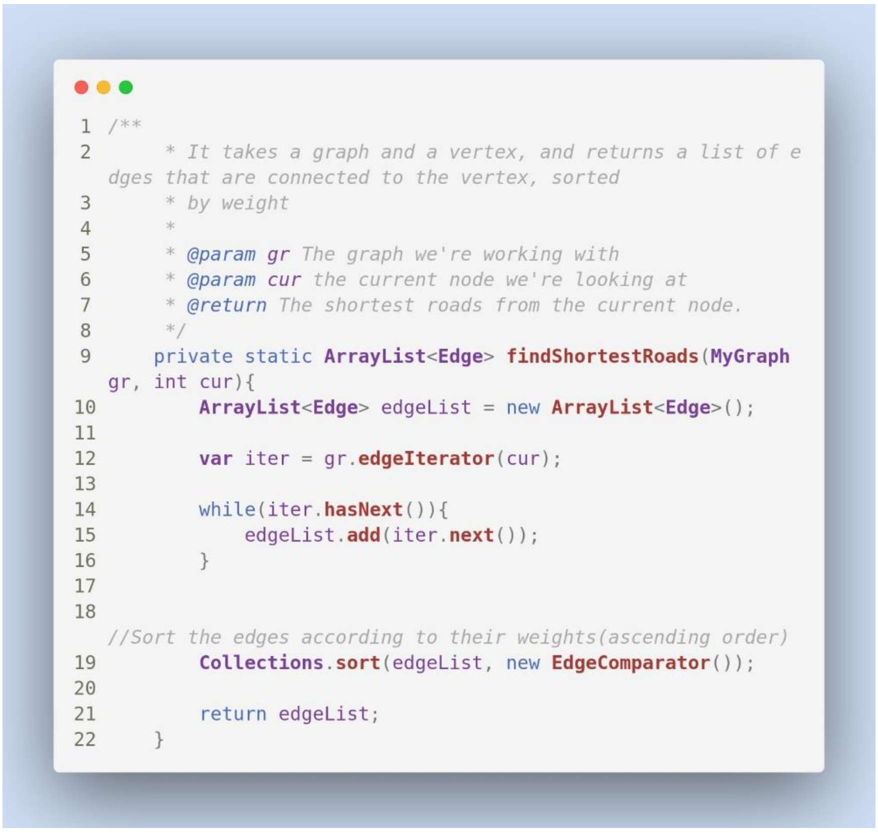
```java
private static double[] findShortestRoads(MyGraph gr){
        int cur = 0;
        double[] arr = new double[gr.getNumV()];
        for(int i = 0; i < arr.length; i++){
            arr[i] = -1;
        }

        while(cur < gr.getNumV()){
            var iter = gr.edgeIterator(cur);
            while(iter.hasNext()){
                var curEdge = iter.next();
                if(arr[curEdge.getDest()] == -1){
                    arr[curEdge.getDest()] = curEdge.getWeight();
                }

                else {
                    if(arr[curEdge.getDest()] > curEdge.getWeight()){
                        arr[curEdge.getDest()] = curEdge.getWeight();
                    }
                }
            }
            cur++;
        }
        return arr;
    }
```

implemented the findShortestRoads method which returns an array of shortest road for each vertex. In the bfs, if the next edges weight equals to the shortest road for that vertex, it goes for it, otherwise it keeps looking for other edges.



```java
if(edge.getWeight() == shortestRoads[edge.getDest()]){
```

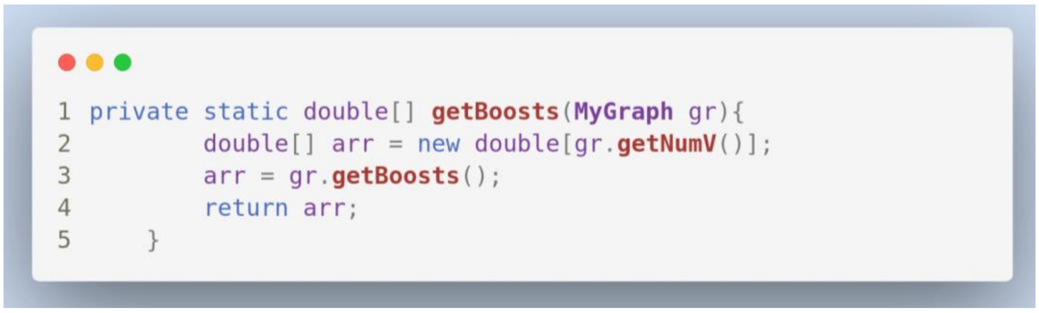
For dfs, I implemented a findShortestRoads method , but this returns an arrayList of Edges. In the method, the edges are sorted by their weights by shortest to longest for the specific vertex. So, when it traversals in dfs it chooses the shortest path first.



```java
/**
     * It takes a graph and a vertex, and returns a list of edges that are connected to the vertex, sorted
     * by weight
     *
     * @param gr The graph we're working with
     * @param cur the current node we're looking at
     * @return The shortest roads from the current node.
     */
    private static ArrayList<Edge> findShortestRoads(MyGraph gr, int cur){
        ArrayList<Edge> edgeList = new ArrayList<Edge>();

        var iter = gr.edgeIterator(cur);

        while(iter.hasNext()){
            edgeList.add(iter.next());
        }


//Sort the edges according to their weights(ascending order)
        Collections.sort(edgeList, new EdgeComparator());

        return edgeList;
    }
```

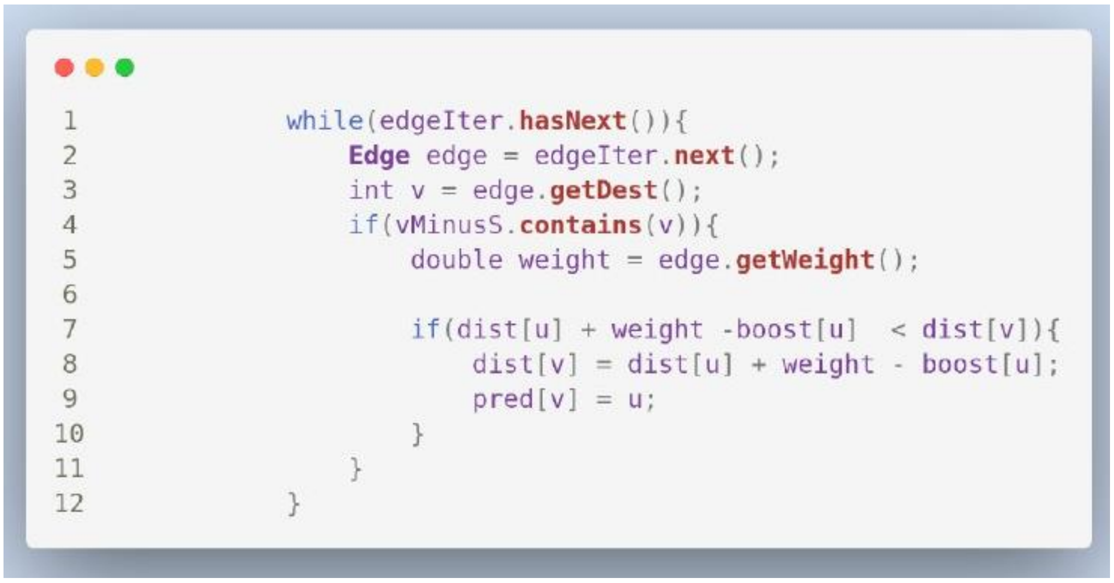
Q3-

I implemented a getBoosts method which returns an array of double, keeps the boost values of vertices.



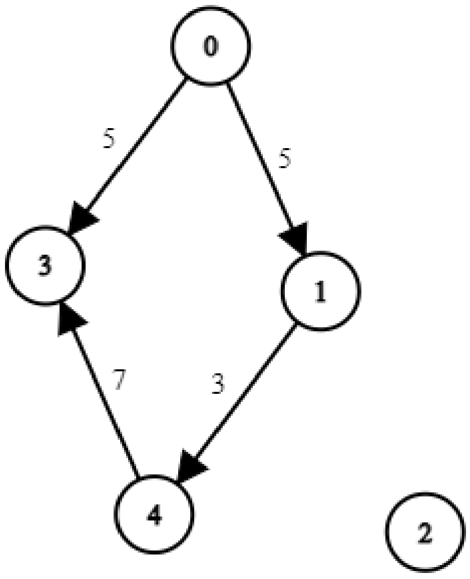
```java
private static double[] getBoosts(MyGraph gr){
        double[] arr = new double[gr.getNumV()];
        arr = gr.getBoosts();
        return arr;
    }
```

Then used these values in the Dijkstra algorithm which I got from our course book.


```
while(edgeIter.hasNext()){
    Edge edge = edgeIter.next();
    int v = edge.getDest();
    if(vMinusS.contains(v)){
        double weight = edge.getWeight();

        if(dist[u] + weight -boost[u]  < dist[v]){
            dist[v] = dist[u] + weight - boost[u];
            pred[v] = u;
        }
    }
}
```

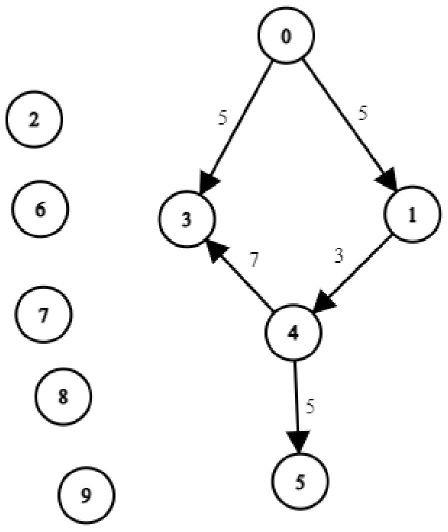
4. TEST CASES

Q1- Created this graph.

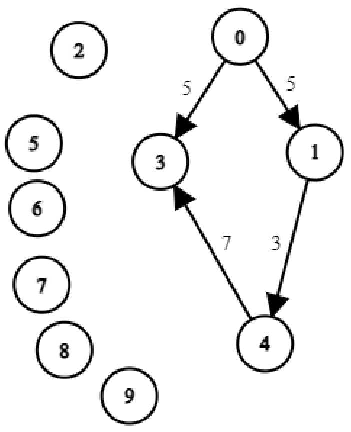


-Added new vertex with different labels or properties.

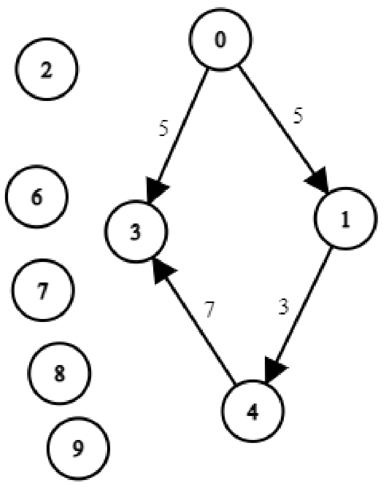
-Added new edge



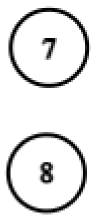
-Removed edge between 4 and 5



-Remove vertex 5



-filter Color = Blue

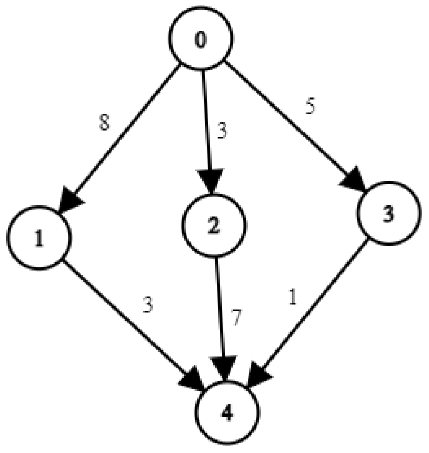


-Then export the matrix of the full graph.

<u>Q2-</u>

Test1-

The graph will be traversed



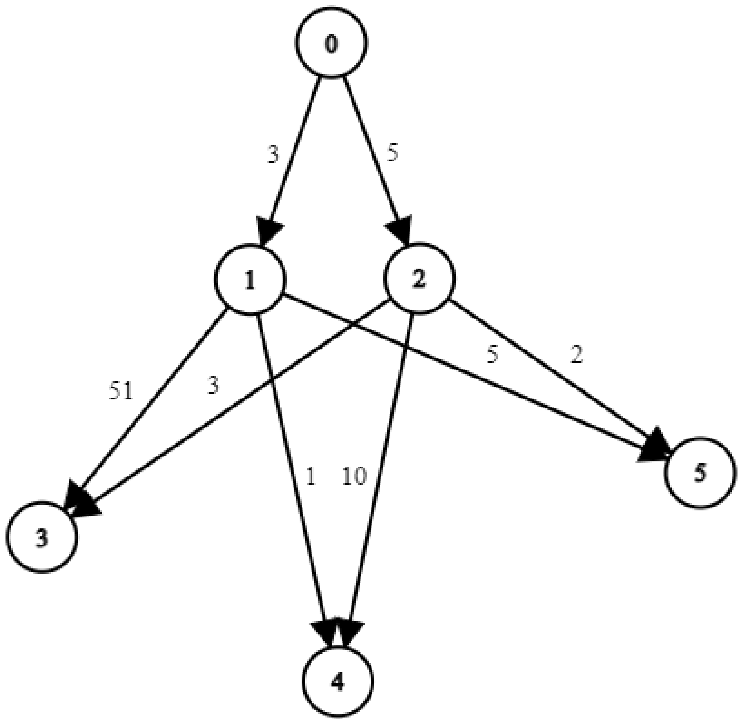
For bfs shortest road to 4 is 0-3-1, thus this way will be selected. The total distance should be = 8 + 3 + 5 + 1 = 17

For dfs after 0 it will select the shortest path so it will go first 2 then 4. The total distance should be 8 + 3 + 5 + 7 = 23

The difference = 6

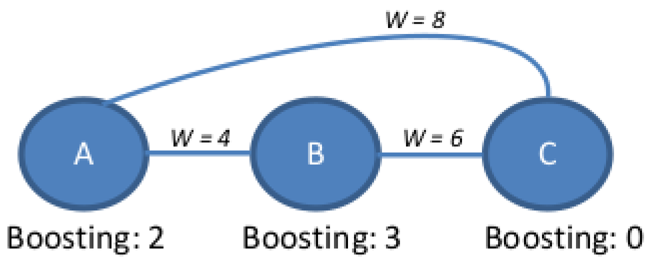
Test2-

The graph will be traversed

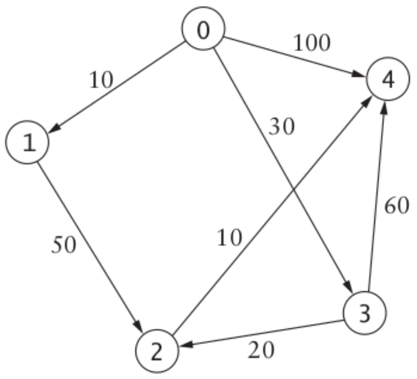


Q3-

Test1:

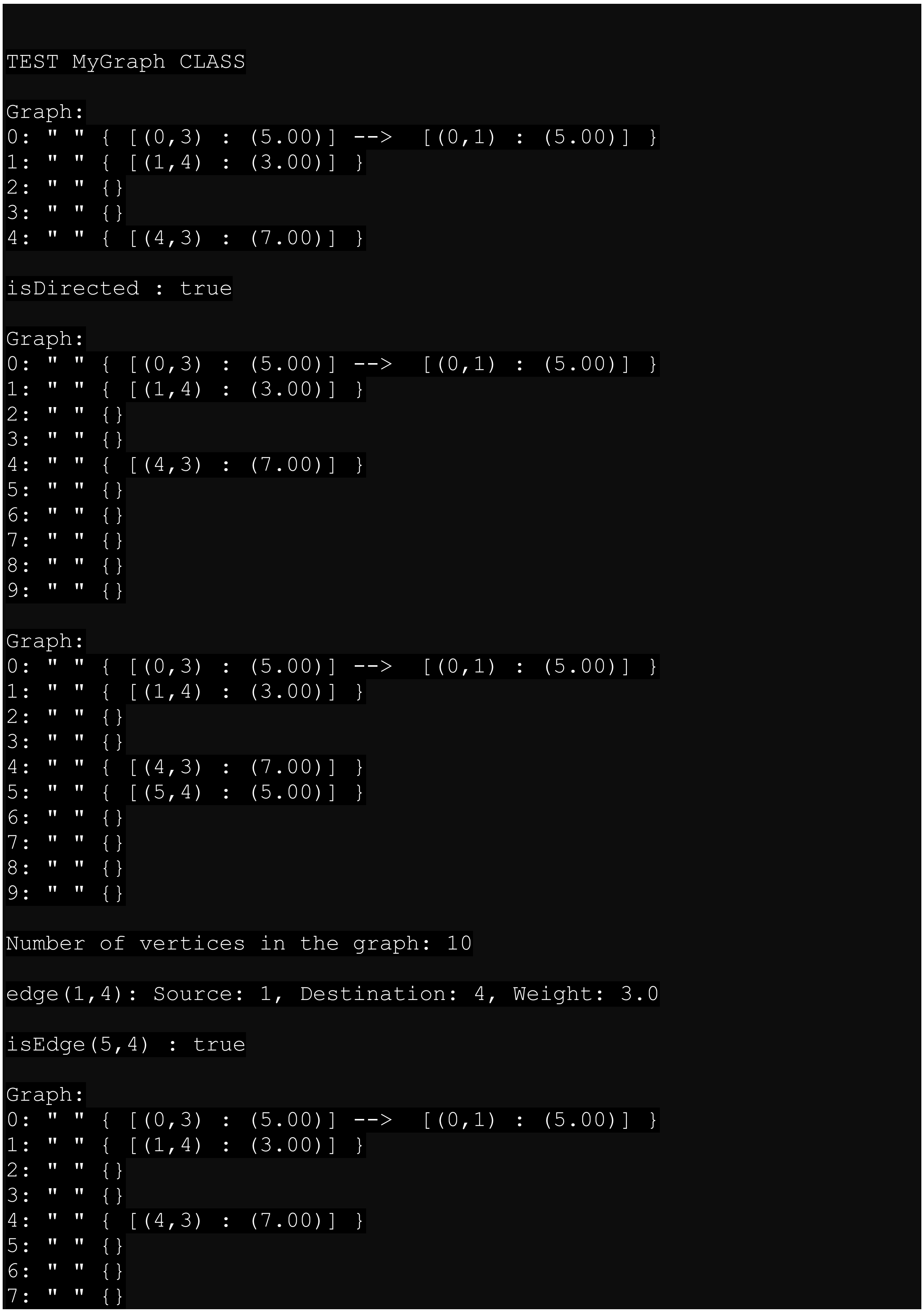


Test2: Vertex 1 is boosted by 20.

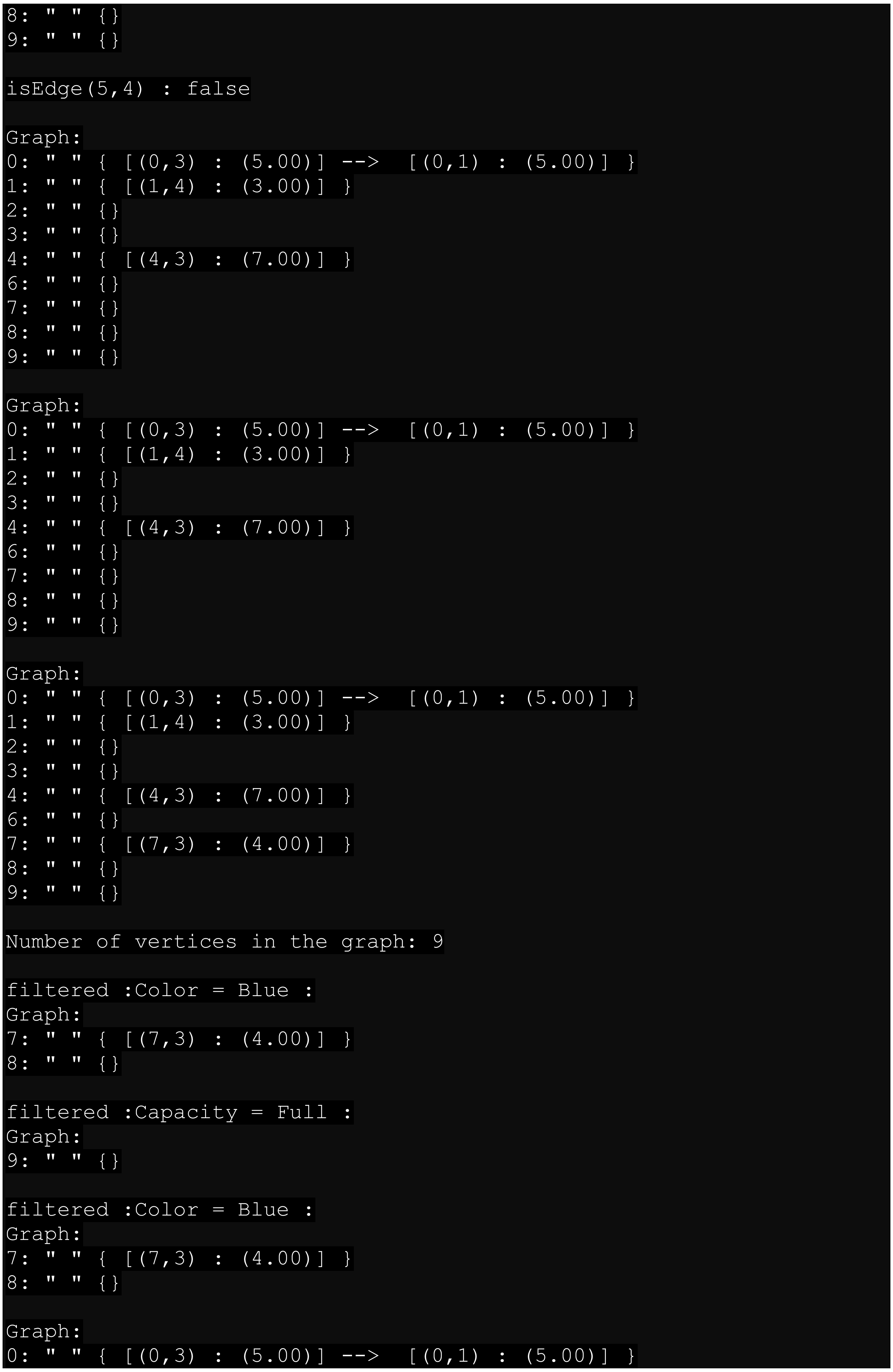


5- RUNNING AND RESULTS

Q1 result:



```
TEST MyGraph CLASS

Graph:
0: " " { [(0,3) : (5.00)] -->  [(0,1) : (5.00)] }
1: " " { [(1,4) : (3.00)] }
2: " " {}
3: " " {}
4: " " { [(4,3) : (7.00)] }

isDirected : true

Graph:
0: " " { [(0,3) : (5.00)] -->  [(0,1) : (5.00)] }
1: " " { [(1,4) : (3.00)] }
2: " " {}
3: " " {}
4: " " { [(4,3) : (7.00)] }
5: " " {}
6: " " {}
7: " " {}
8: " " {}
9: " " {}

Graph:
0: " " { [(0,3) : (5.00)] -->  [(0,1) : (5.00)] }
1: " " { [(1,4) : (3.00)] }
2: " " {}
3: " " {}
4: " " { [(4,3) : (7.00)] }
5: " " { [(5,4) : (5.00)] }
6: " " {}
7: " " {}
8: " " {}
9: " " {}

Number of vertices in the graph: 10

edge(1,4): Source: 1, Destination: 4, Weight: 3.0

isEdge(5,4) : true

Graph:
0: " " { [(0,3) : (5.00)] -->  [(0,1) : (5.00)] }
1: " " { [(1,4) : (3.00)] }
2: " " {}
3: " " {}
4: " " { [(4,3) : (7.00)] }
5: " " {}
6: " " {}
7: " " {}
```



```
8: " " {}
9: " " {}

isEdge(5,4) : false

Graph:
0: " " { [(0,3) : (5.00)] -->  [(0,1) : (5.00)] }
1: " " { [(1,4) : (3.00)] }
2: " " {}
3: " " {}
4: " " { [(4,3) : (7.00)] }
6: " " {}
7: " " {}
8: " " {}
9: " " {}

Graph:
0: " " { [(0,3) : (5.00)] -->  [(0,1) : (5.00)] }
1: " " { [(1,4) : (3.00)] }
2: " " {}
3: " " {}
4: " " { [(4,3) : (7.00)] }
6: " " {}
7: " " {}
8: " " {}
9: " " {}

Graph:
0: " " { [(0,3) : (5.00)] -->  [(0,1) : (5.00)] }
1: " " { [(1,4) : (3.00)] }
2: " " {}
3: " " {}
4: " " { [(4,3) : (7.00)] }
6: " " {}
7: " " { [(7,3) : (4.00)] }
8: " " {}
9: " " {}

Number of vertices in the graph: 9

filtered :Color = Blue :
Graph:
7: " " { [(7,3) : (4.00)] }
8: " " {}

filtered :Capacity = Full :
Graph:
9: " " {}

filtered :Color = Blue :
Graph:
7: " " { [(7,3) : (4.00)] }
8: " " {}

Graph:
0: " " { [(0,3) : (5.00)] -->  [(0,1) : (5.00)] }
```

```
1: " " { [(1,4) : (3.00)] }
2: " " {}
3: " " {}
4: " " { [(4,3) : (7.00)] }
6: " " {}
7: " " { [(7,3) : (4.00)] }
8: " " {}
9: " " {}

[0][1]
[0][3]
[1][4]
[4][3]
[7][3]
```

Q2 result:

```
TESTING findDifference Method // 1

Bfs distance: 17

Dfs distance: 23

Difference of the traversals: 6

TESTING findDifference Method // 2

Bfs distance: 14

Dfs distance: 65

Difference of the traversals: 51
```

Q3 Result:

```
TESTING Dijkstra With Boost
TESTING example 1:
Node, Predecessor, and Distance:
1:      0           4.0
2:      1           7.0
TESTING example 2:
Node, Predecessor, and Distance:
1:      0           10.0
2:      1           40.0
3:      0           30.0
4:      2           50.0
```