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

Emre YILMAZ
1901042606

1. SYSTEM REQUIREMENTS

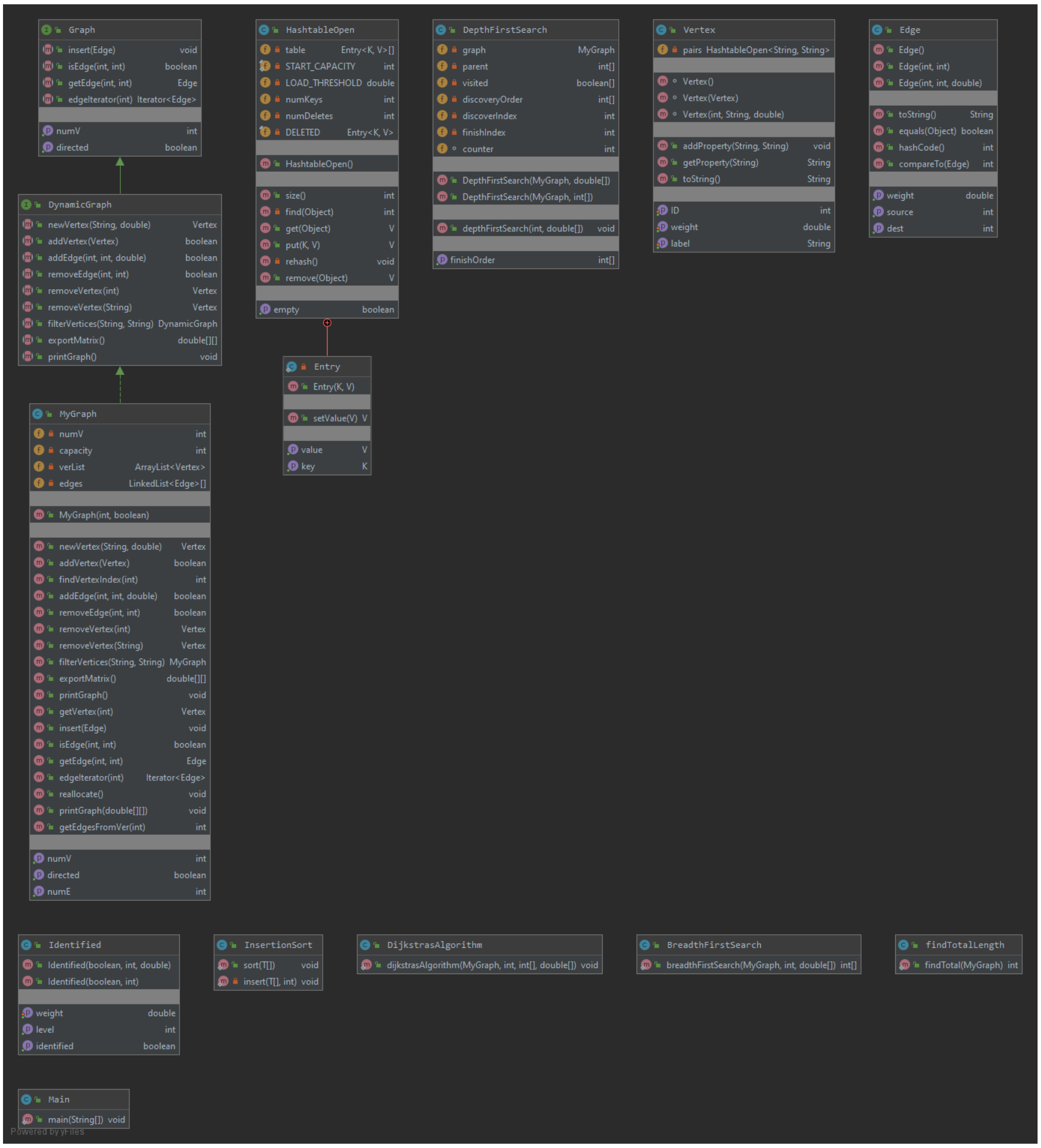
In the first part, we must design a graph interface extending the Graph class in the textbook. We will use adjacency list implementation. In addition to the graph implementation in the book, the vertices will be represented as data in the implementation. We must be able to create and display vertices with their information. Also, we must add a dynamic structure to the Graph class. We must be able to add/remove edges, add/remove vertices. The vertices may have additional properties such as length, color, boosting etc. We must handle this. Excluding the add/remove edge and vertex methods, we have a filter method. This filter method creates a subgraph with given property. For instance, if parameters of the filter method are (“color”, “red”), a graph will be generated that consists of having red color property. Also, we will have exportMatrix and printMatrix methods. exportMatrix method generate and returns the adjacency matrix. printMatrix method prints the graph and vertices with their information.

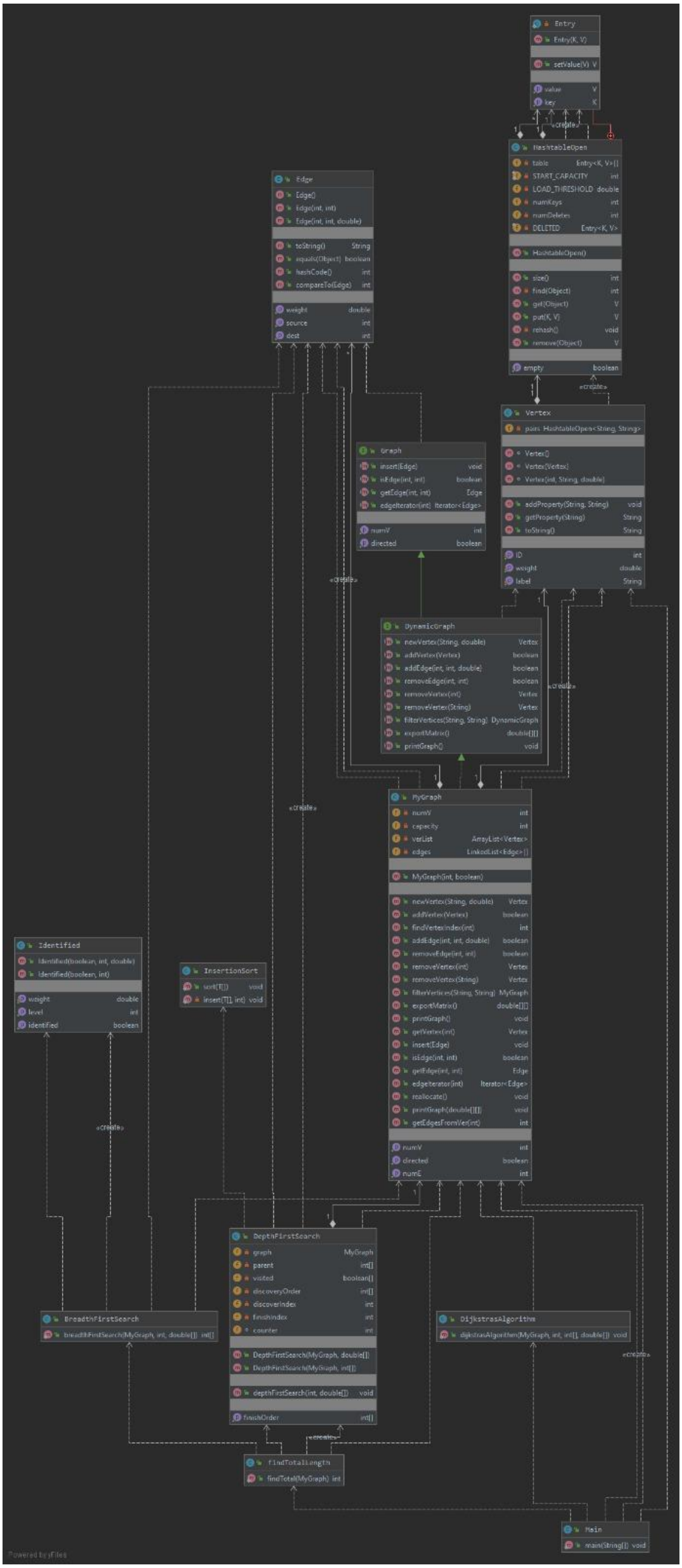
In the second part, we will traverse the graph using Breadth First Search and Depth First Search algorithms. We will calculate total distance while traversing with these two algorithms. There are some additional properties to classic implementations of these algorithms. In the BFS, if there are more than one alternative to access a vertex at a specific level, the shortest alternative should be considered. In the DFS, The vertices should be considered in distance order, so, from a vertex v, DFS should continue with a vertex w which has the smallest edge from v, among all adjacent vertices of v.

In the third part, we will write a method that takes a MyGraph object and a vertex as a parameter to perform a modified version of Dijkstra’s Algorithm for calculating the shortest paths from the given vertex to all other vertices in the graph. In this modified version, the algorithm considers boosting value of the vertices in addition to the edge weights. The boosting property is a user-defined property that takes double values. The boosting values are subtracted from the total length of paths that they are contained in.

2. CLASS DIAGRAMS

Without Composition (without has-a-relationship)





Including composition (with has-a-relationship)

3. OTHER DIAGRAMS

Not necessary in this homework.

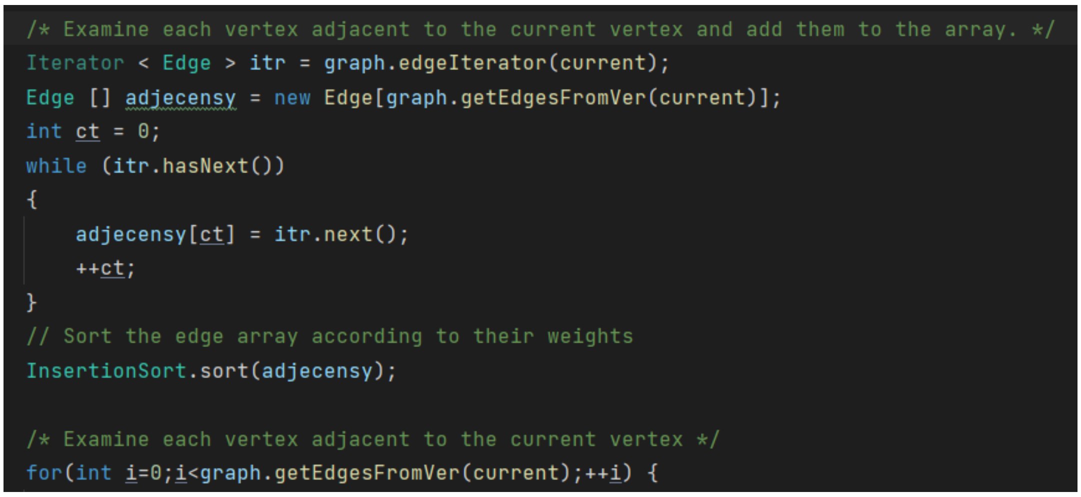
4. PROBLEM SOLUTION APPROACH

Part-1.)

- An ArrayList object is created to store vertices. The reason behind choosing ArrayList for data structure is direct access and its Θ(1) time complexity. Moreover, if there is no removed vertices in the Vertex List, the ID of the vertex will be the same as the its index in the ArrayList. So, getting a Vertex from ArrayList will be Θ(1) time complexity. Because of this, while trying to access an ID, I tried first access the VertexList[ID] element. If there is no ID match, then I tried a linear search to find vertex with given ID. I try to access a vertex is Θ (1) time. Also, I considered the small size data. If we consider huge data, Binary Search Tree could be implemented for store vertices. But in this homework, there is no need to use BST to make things easier.
- I use newVertex method as a creator. It only creates a Vertex and returns it. Then, addVertex method adds it to the Vertex ArrayList.
- One of the points that we need to be careful is removing a vertex. When we remove a vertex, we must remove all the edges that contains this vertex. I was careful about that while implementing. If an edge has the target vertex as source or destination, it is removed.
- I was careful about directed/undirected graph. While adding/removing edges and Removing vertices processes, directed property of the graph is considered.
- printGraph method prints the adjacency list and vertices with their weight and ID properties.
- Some methods that in the ListGraph class in the textbook is taken as they are such as isEdge, getEdge, getIterator, isDirected etc. Insert method in the textbook is used as a helper method for addEdge method.
- Vertex class has ID, weight, and property with getter and setter methods. Property data is set as HashTableOpen type object since there is need to supply a key-value pair structure.

Part-2.)

- There is a small change in the DepthFirstSearch implementation in the textbook to provide desired features in the PDF. In the implementation in the book, the algorithm, visits the neighbors of a vertex in the order of edge existence in the linked list. But now, we must visit them in order of edge's weights in the linked list. So, instead visiting vertices using iterator, I firstly create an Edge array. All the edges from the source vertex is added to this array. Then, this array is sent to a sort algorithm (Insertion Sort is selected since the data is small.) according to weights. After the sorting process, the vertices is visited in order of Edge array that is sorted according to weights using a simple for loop. In this way, we can always go to the shortest edge while traversing Depth First Search. Of course, comparing the edges is need something to be changed. Edge class is made Comparable and implements Comparable interface. Compare method in the Edge class comparing the weight variable of the edges. The change in the implementation is below.



```
/* Examine each vertex adjacent to the current vertex and add them to the array. */
Iterator < Edge > itr = graph.edgeIterator(current);
Edge [] adjecensy = new Edge[graph.getEdgesFromVer(current)];
int ct = 0;
while (itr.hasNext())
{
    adjecensy[ct] = itr.next();
    ++ct;
}
// Sort the edge array according to their weights
InsertionSort.sort(adjecensy);

/* Examine each vertex adjacent to the current vertex */
for(int i=0;i<graph.getEdgesFromVer(current);++i) {
```

- There is a small change in the BreadthFirstSearch implementation in the textbook to provide desired features in the PDF. In the implementation in the textbook, if there are more than one alternative to access a vertex at a specific level, the algorithm uses the edge that is added the Queue firstly and in the order of the queue is only according to the indexes of the vertices. In the new algorithm, instead of using Integer array to mark the vertices as identified, I use an object type. I write a class named Identified to be able to store the information of the all visiting. This class has Boolean identified, int level and double weight variables. For instance, if the vertex-5 at the level 3 and weight 10 is marked as identified, then,
    - IdentifiedObject[5].identified=true,
    - IdentifiedObject[5].level = 3,
    - IdentifiedObject[5].weight = 10

  assignments will be done.

  After this change everything is easier. If we are a vertex and one of destination vertex is visited before, we must consider something more. If source vertex level is only 1 less than destination vertex level that is although visited, we will look its weight. If the weight of the previous visiting is greater than the new one, we update the total distance. We will update the new one. This weight comparing process is done by Identified type object array. The calculating the total distance while traversing is done by help of the Identified array (with help of weight data field in it), too. It is so useful…

NOTE: In this part, added or changed lines in the BFS DFS algorithms are indicated with using comment lines.

Part-3.)

This part was easy to implement. There is small change in the Dijkstra's Algorithm in the textbook. While checking the weight from vertex *v* to vertex *u*, we will not only consider the weight of edge(v,u). We also check that "Boosting" property of the vertex v. If there is a "Boosting" property of the vertex v, we will convert it to double and use it to decrease the weight. MyGraph class is written pretty good, so, looking for "Boosting" property is so easy.

5. TEST CASES

Part-1.)

- A directed graph is created.
- Some vertices are created.
- Some properties such as "color" and "stress" is added.
- Created vertices is added to the graph
- Some edges are added to the graph.
- Print the graph.
- Use the filterVertices method and create a subgraph.
- Print the subgraph
- Use the exportMatrix method and create an adjacency Matrix
- Print the adjacency Matrix
- Remove some vertices.
- Print the graph.
- Remove some edges.
- Print the graph.

- An undirected graph is created.
- Some vertices are created that is different from the directed graph above.
- The other steps is the same as directed graph.

- All these cases run successfully. If you want to, you can change some properties of the vertices or edges. 2 different graph is created to prove the algorithm runs correctly. Also, in the 2nd and 3rd parts, there are some graphs are created. They are another proof.

Part-2.)

- A small, directed graph is created to test especially BFS. It is a test case we will observe whether BFS runs correctly. Nevertheless, it is also tested all the PART-2
- The graph in the textbook in the page 524 is created. It is a 5 vertex graph.
- The graphs in this part are deliberately kept small. In order to see that it is working correctly, quickly and practically.
- Some vertices are created
- Created vertices is added to the graph.
- Some edges are added to the graph.
- Print the graph.
- Run the part-2 and calculate the total distance.
- Create a graph too, do the same steps and run part-2 calculating the total distance for traversing algorithms
- *IMPORTANT*: The created graphs in this part are also drawn on paper. It will be showed in the Running and Results part. So, we will be able to see the result of the Part-2 and the real result. We will be able to compare the results.

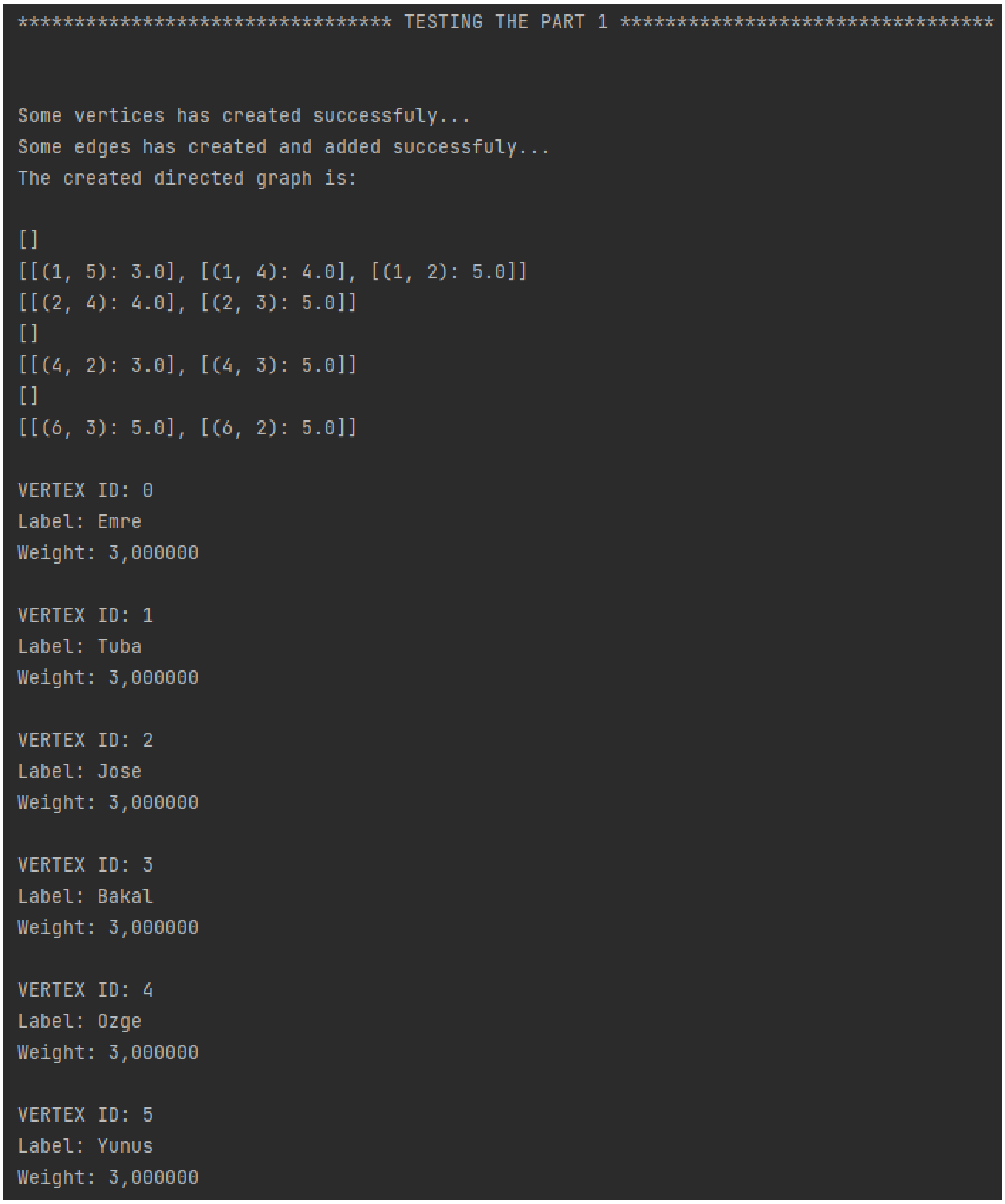
Part-3.)

- Run the Part-3 with graphs that is created previous part.
- The graphs that is crated in this part are also drawn on paper to prove the algorithm runs successfully.

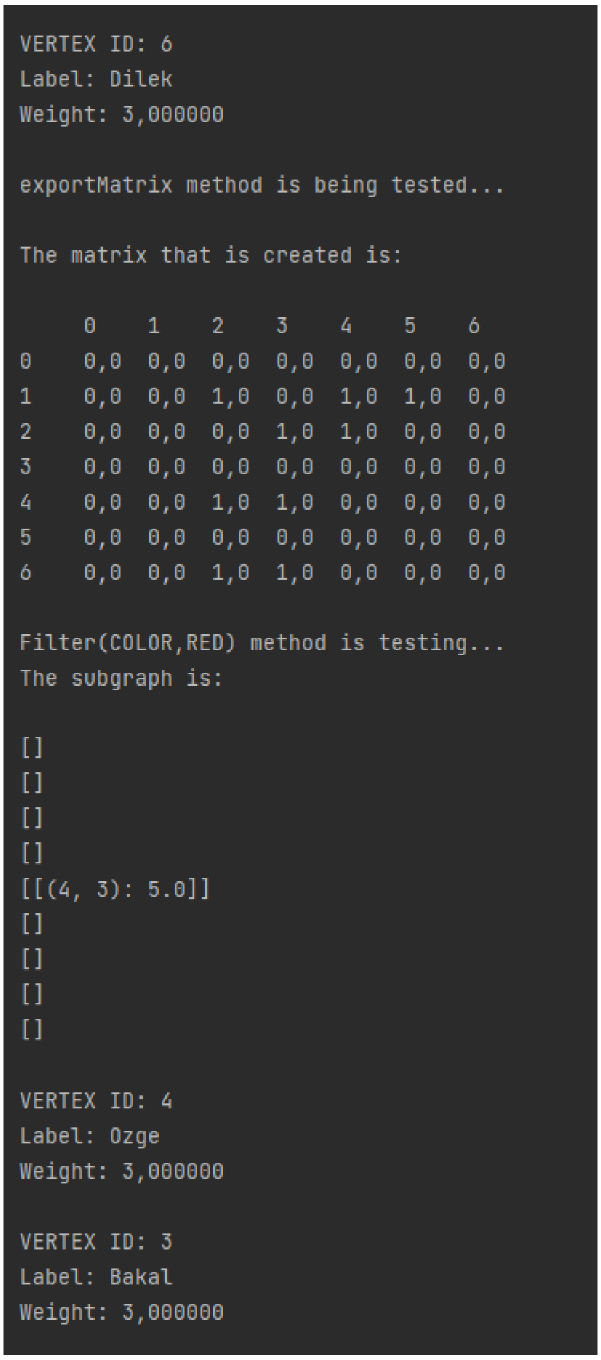
6. RUNNING AND RESULTS

TESTING PART-1

Directed Graph-1

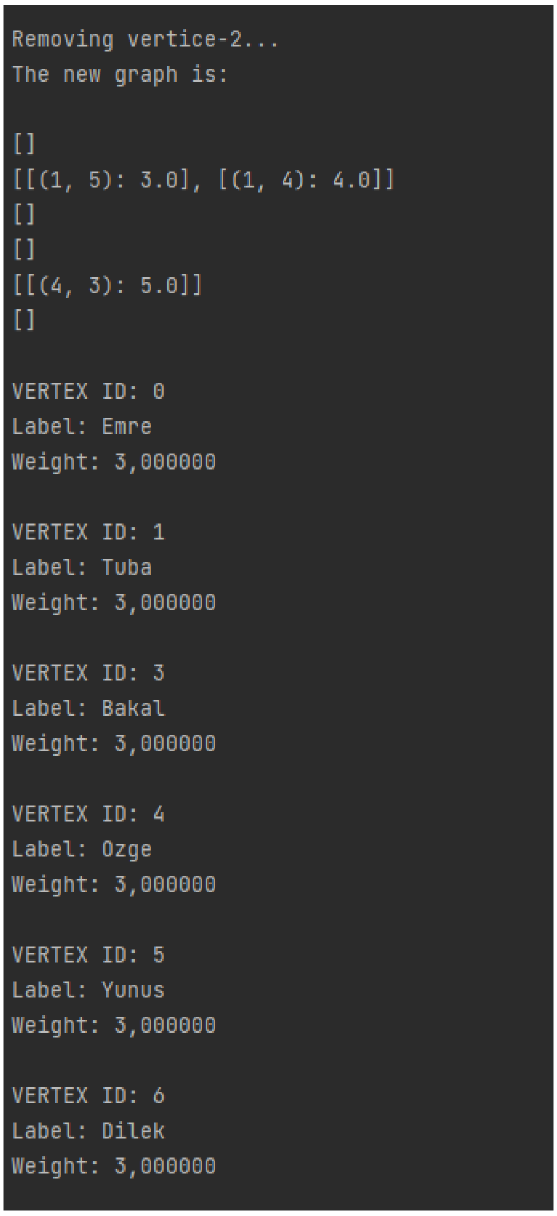


```
********************************* TESTING THE PART 1 *********************************


Some vertices has created successfuly...
Some edges has created and added successfuly...
The created directed graph is:

[]
[[(1, 5): 3.0], [(1, 4): 4.0], [(1, 2): 5.0]]
[[(2, 4): 4.0], [(2, 3): 5.0]]
[]
[[(4, 2): 3.0], [(4, 3): 5.0]]
[]
[[(6, 3): 5.0], [(6, 2): 5.0]]

VERTEX ID: 0
Label: Emre
Weight: 3,000000

VERTEX ID: 1
Label: Tuba
Weight: 3,000000

VERTEX ID: 2
Label: Jose
Weight: 3,000000

VERTEX ID: 3
Label: Bakal
Weight: 3,000000

VERTEX ID: 4
Label: Ozge
Weight: 3,000000

VERTEX ID: 5
Label: Yunus
Weight: 3,000000
```

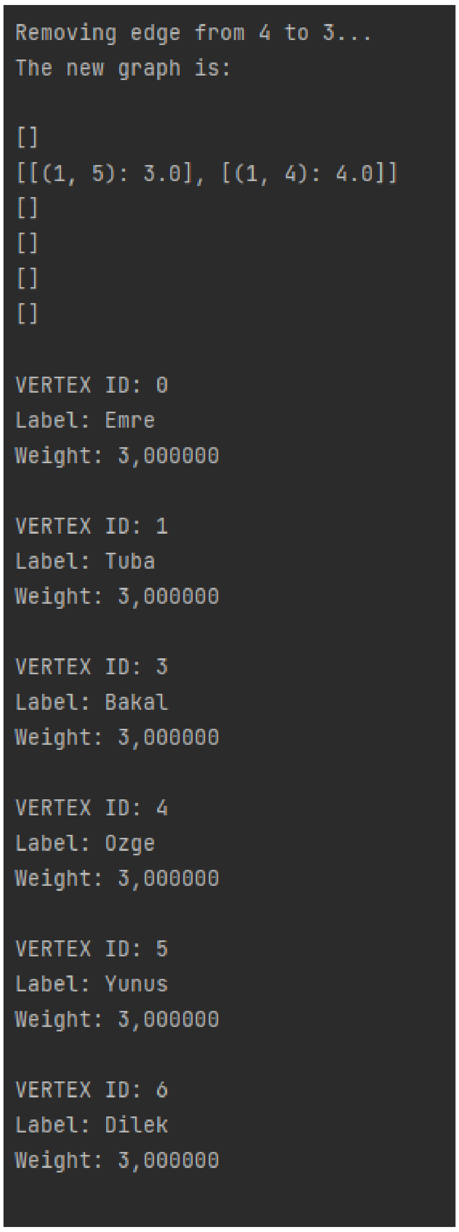


```
VERTEX ID: 6
Label: Dilek
Weight: 3,000000

exportMatrix method is being tested...

The matrix that is created is:

     0    1    2    3    4    5    6
0    0,0  0,0  0,0  0,0  0,0  0,0  0,0
1    0,0  0,0  1,0  0,0  1,0  1,0  0,0
2    0,0  0,0  0,0  1,0  1,0  0,0  0,0
3    0,0  0,0  0,0  0,0  0,0  0,0  0,0
4    0,0  0,0  1,0  1,0  0,0  0,0  0,0
5    0,0  0,0  0,0  0,0  0,0  0,0  0,0
6    0,0  0,0  1,0  1,0  0,0  0,0  0,0

Filter(COLOR,RED) method is testing...
The subgraph is:

[]
[]
[]
[]
[[(4, 3): 5.0]]
[]
[]
[]
[]

VERTEX ID: 4
Label: Ozge
Weight: 3,000000

VERTEX ID: 3
Label: Bakal
Weight: 3,000000
```

When filtering is done, it can be clearly observed above that the correct nodes and edges remain.

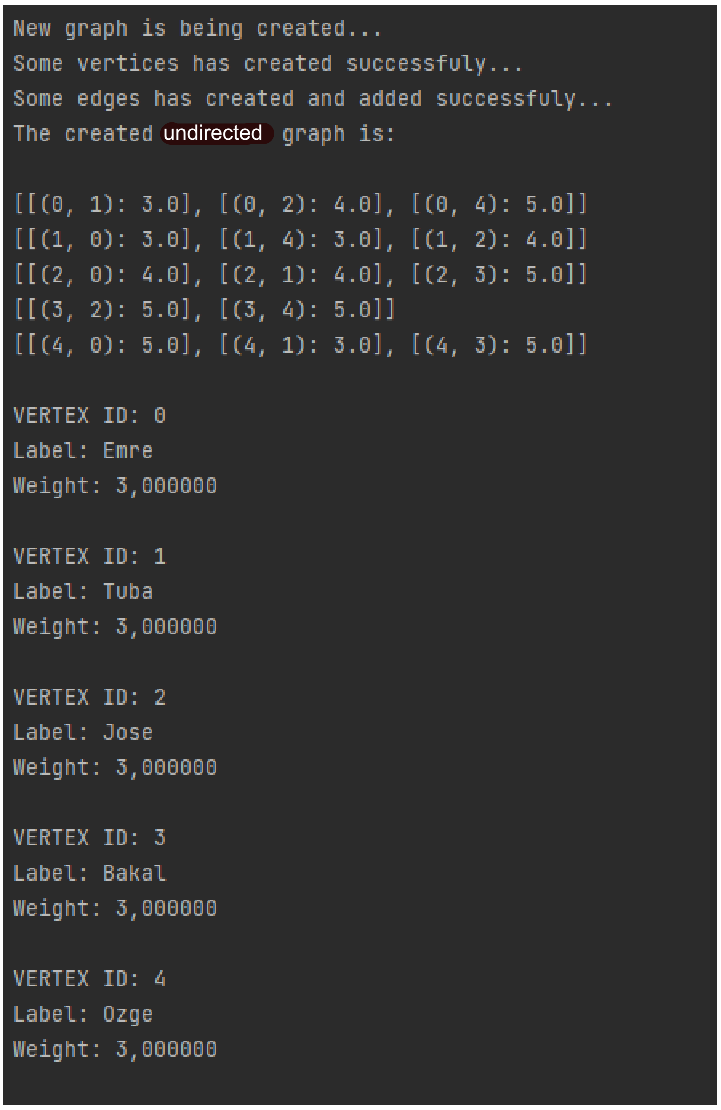

```
Removing vertice-2...
The new graph is:

[]
[[(1, 5): 3.0], [(1, 4): 4.0]]
[]
[]
[[(4, 3): 5.0]]
[]

VERTEX ID: 0
Label: Emre
Weight: 3,000000

VERTEX ID: 1
Label: Tuba
Weight: 3,000000

VERTEX ID: 3
Label: Bakal
Weight: 3,000000

VERTEX ID: 4
Label: Ozge
Weight: 3,000000

VERTEX ID: 5
Label: Yunus
Weight: 3,000000

VERTEX ID: 6
Label: Dilek
Weight: 3,000000
```

When we remove Vertex-2, it can be clearly observed above that the edges related to Vertex-2 is removed correctly. Also Vertex-2 is deleted from vertex list.



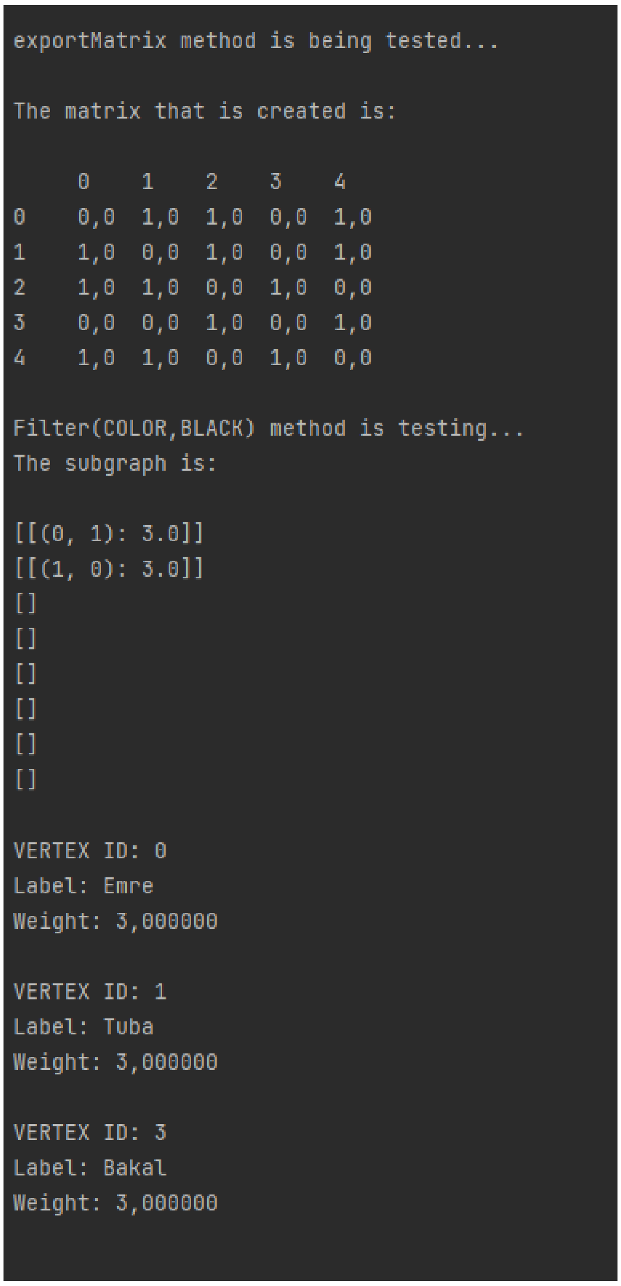
```
Removing edge from 4 to 3...
The new graph is:

[]
[[(1, 5): 3.0], [(1, 4): 4.0]]
[]
[]
[]
[]

VERTEX ID: 0
Label: Emre
Weight: 3,000000

VERTEX ID: 1
Label: Tuba
Weight: 3,000000

VERTEX ID: 3
Label: Bakal
Weight: 3,000000

VERTEX ID: 4
Label: Ozge
Weight: 3,000000

VERTEX ID: 5
Label: Yunus
Weight: 3,000000

VERTEX ID: 6
Label: Dilek
Weight: 3,000000
```

The edge from 4 to 3 is removed successfully.



```
New graph is being created...
Some vertices has created successfuly...
Some edges has created and added successfuly...
The created undirected graph is:

[[(0, 1): 3.0], [(0, 2): 4.0], [(0, 4): 5.0]]
[[(1, 0): 3.0], [(1, 4): 3.0], [(1, 2): 4.0]]
[[(2, 0): 4.0], [(2, 1): 4.0], [(2, 3): 5.0]]
[[(3, 2): 5.0], [(3, 4): 5.0]]
[[(4, 0): 5.0], [(4, 1): 3.0], [(4, 3): 5.0]]

VERTEX ID: 0
Label: Emre
Weight: 3,000000

VERTEX ID: 1
Label: Tuba
Weight: 3,000000

VERTEX ID: 2
Label: Jose
Weight: 3,000000

VERTEX ID: 3
Label: Bakal
Weight: 3,000000

VERTEX ID: 4
Label: Ozge
Weight: 3,000000
```

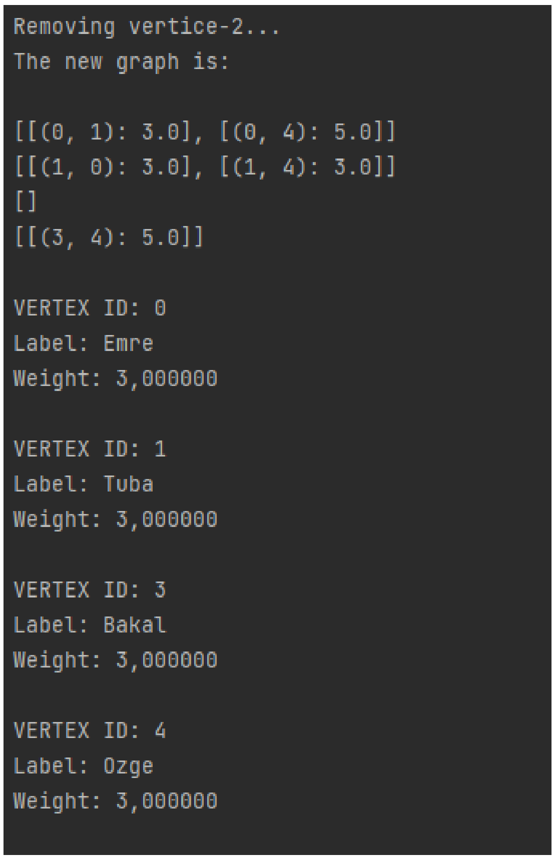
The undirected Graph-2 is created.



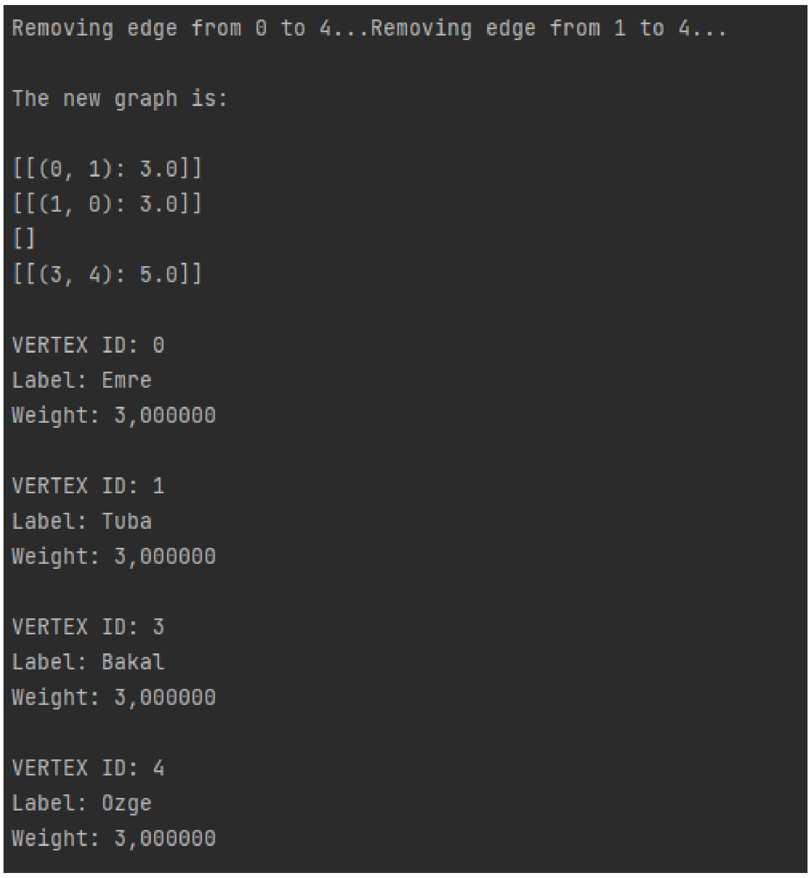
```
exportMatrix method is being tested...

The matrix that is created is:

     0    1    2    3    4
0    0,0  1,0  1,0  0,0  1,0
1    1,0  0,0  1,0  0,0  1,0
2    1,0  1,0  0,0  1,0  0,0
3    0,0  0,0  1,0  0,0  1,0
4    1,0  1,0  0,0  1,0  0,0

Filter(COLOR,BLACK) method is testing...
The subgraph is:

[[(0, 1): 3.0]]
[[(1, 0): 3.0]]
[]
[]
[]
[]
[]
[]

VERTEX ID: 0
Label: Emre
Weight: 3,000000

VERTEX ID: 1
Label: Tuba
Weight: 3,000000

VERTEX ID: 3
Label: Bakal
Weight: 3,000000
```

Export matrix is tested.

Color=>Black filter is run correctly. Only vertex 0-1 and 3 is black. Moreover, this vertex has only edge(0,1)


```
Removing vertice-2...
The new graph is:

[[(0, 1): 3.0], [(0, 4): 5.0]]
[[(1, 0): 3.0], [(1, 4): 3.0]]
[]
[[(3, 4): 5.0]]

VERTEX ID: 0
Label: Emre
Weight: 3,000000

VERTEX ID: 1
Label: Tuba
Weight: 3,000000

VERTEX ID: 3
Label: Bakal
Weight: 3,000000

VERTEX ID: 4
Label: Ozge
Weight: 3,000000
```

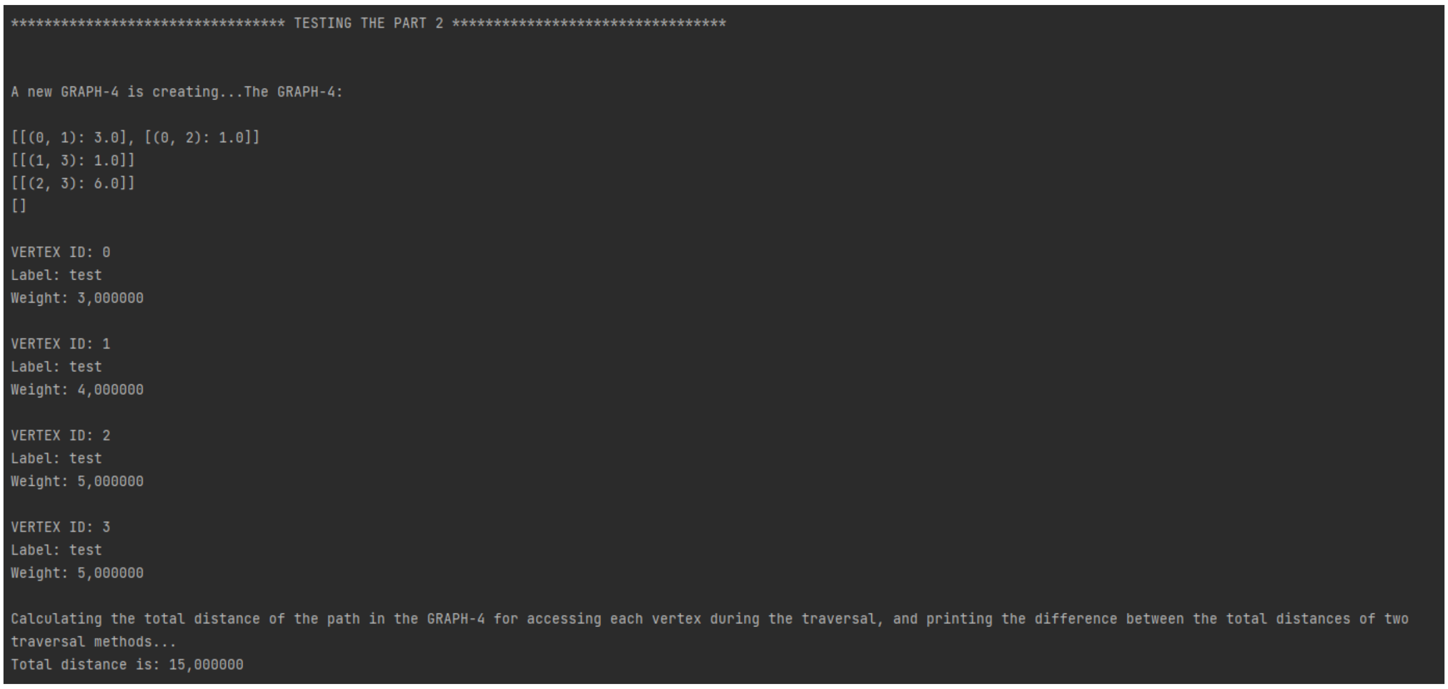
When we remove Vertex-2, it can be clearly observed above that the edges related to Vertex-2 is removed correctly. Also Vertex-2 is deleted from vertex list.



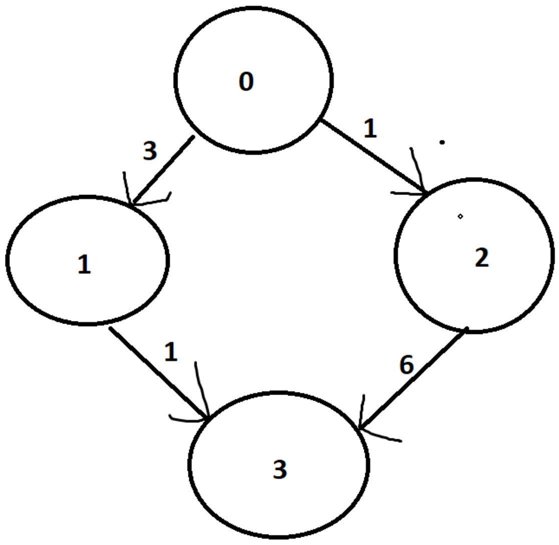
```
Removing edge from 0 to 4...Removing edge from 1 to 4...

The new graph is:

[[(0, 1): 3.0]]
[[(1, 0): 3.0]]
[]
[[(3, 4): 5.0]]

VERTEX ID: 0
Label: Emre
Weight: 3,000000

VERTEX ID: 1
Label: Tuba
Weight: 3,000000

VERTEX ID: 3
Label: Bakal
Weight: 3,000000

VERTEX ID: 4
Label: Ozge
Weight: 3,000000
```

The edge from 0 to 4 and the edge from 1 to 4 are removed successfully.

TESTING PART-2



```
******************************** TESTING THE PART 2 ********************************


A new GRAPH-4 is creating...The GRAPH-4:

[[(0, 1): 3.0], [(0, 2): 1.0]]
[[(1, 3): 1.0]]
[[(2, 3): 6.0]]
[]

VERTEX ID: 0
Label: test
Weight: 3,000000

VERTEX ID: 1
Label: test
Weight: 4,000000

VERTEX ID: 2
Label: test
Weight: 5,000000

VERTEX ID: 3
Label: test
Weight: 5,000000

Calculating the total distance of the path in the GRAPH-4 for accessing each vertex during the traversal, and printing the difference between the total distances of two
traversal methods...
Total distance is: 15,000000
```



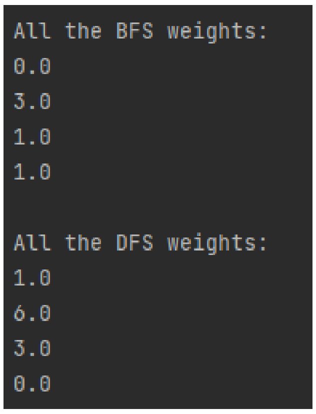
The created graph is on the left.

As you can see, BFS must be 5 according to algorithm in the PDF. The node 3 is can be reached by 2 vertex and we must pick the shorter one. So total BFS distance is 3+1+1 = 5

Also, DFS must be 1+6+3 = 10 since we must choose firsly (0,2) edge. Its weight less than other one. The algorithm in the PDF says that.

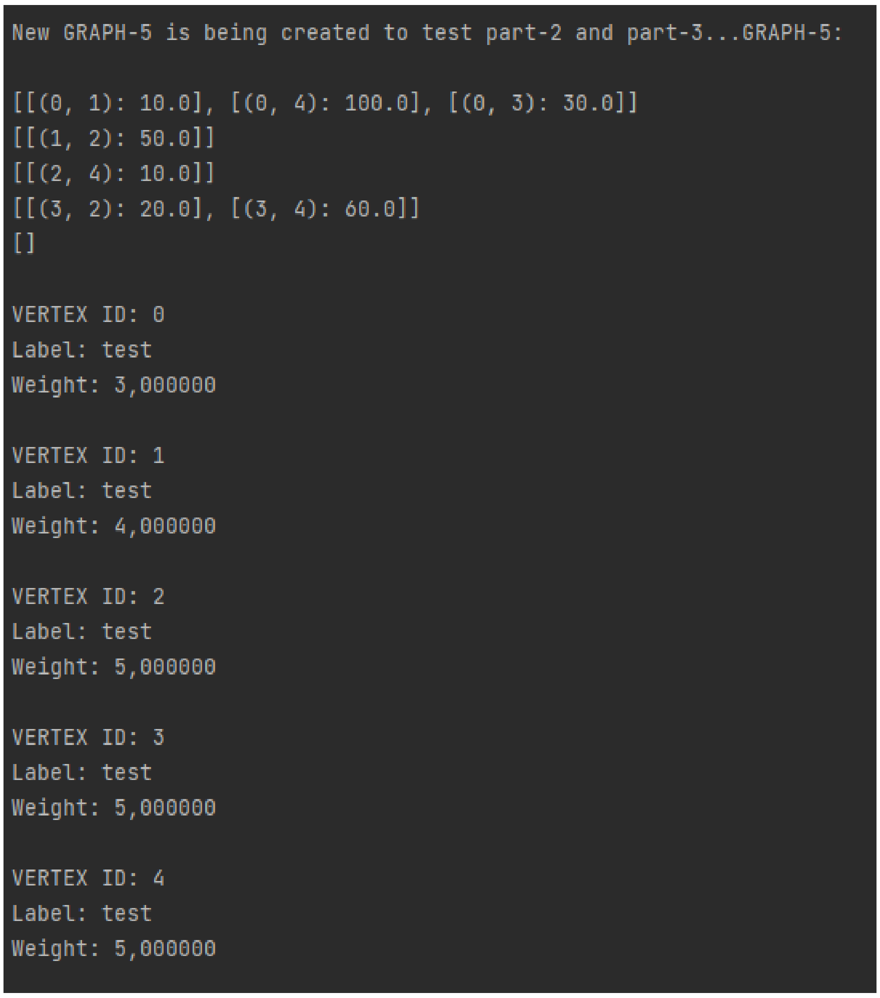
Total distance must be 15, and the test result shows us 15. My algorithm can run correctly!

Also, there are a few lines in the findTotalLength class. You can see all the distances removing the comment lines. I remove for you for once:

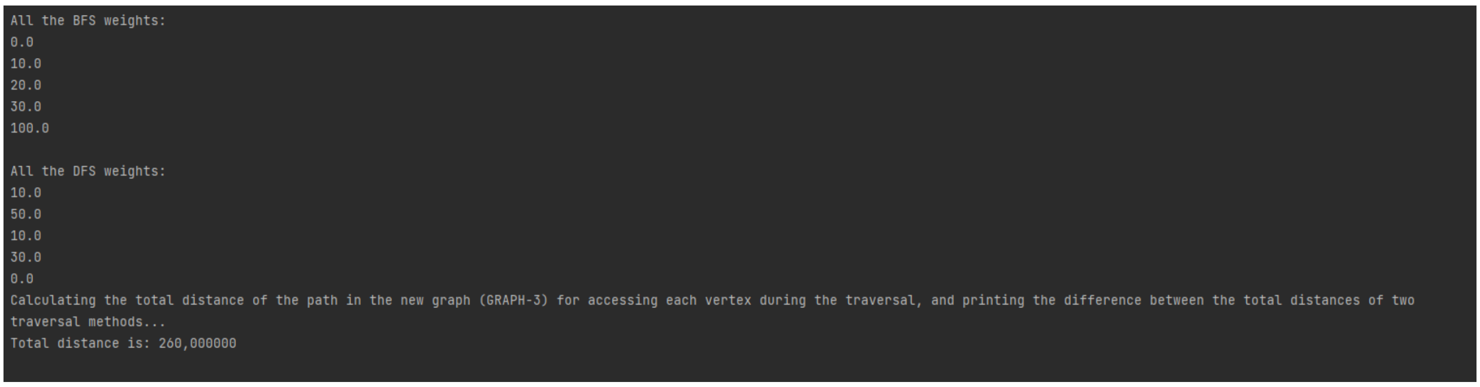


```
All the BFS weights:
0.0
3.0
1.0
1.0

All the DFS weights:
1.0
6.0
3.0
0.0
```

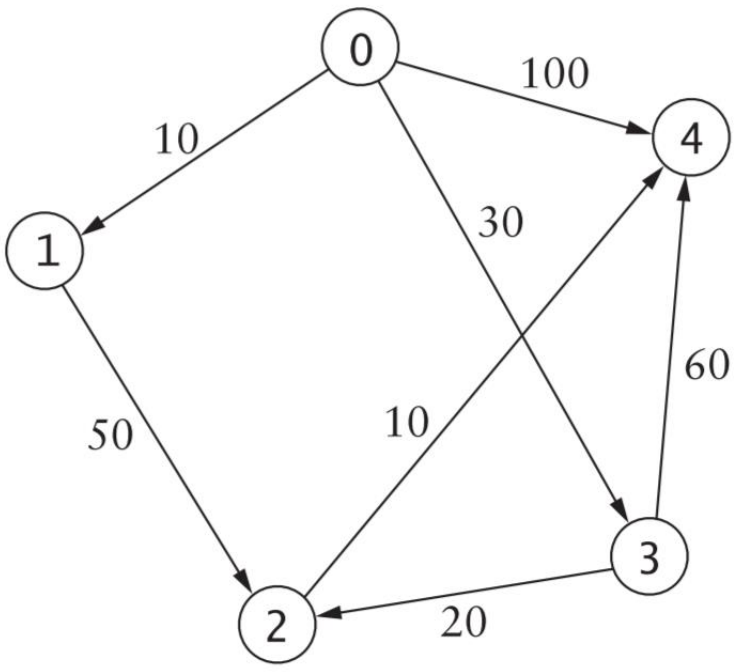
Continue to test part-2 with another example



```
New GRAPH-5 is being created to test part-2 and part-3...GRAPH-5:

[[(0, 1): 10.0], [(0, 4): 100.0], [(0, 3): 30.0]]
[[(1, 2): 50.0]]
[[(2, 4): 10.0]]
[[(3, 2): 20.0], [(3, 4): 60.0]]
[]

VERTEX ID: 0
Label: test
Weight: 3,000000

VERTEX ID: 1
Label: test
Weight: 4,000000

VERTEX ID: 2
Label: test
Weight: 5,000000

VERTEX ID: 3
Label: test
Weight: 5,000000

VERTEX ID: 4
Label: test
Weight: 5,000000
```



```
All the BFS weights:
0.0
10.0
20.0
30.0
100.0

All the DFS weights:
10.0
50.0
10.0
30.0
0.0
Calculating the total distance of the path in the new graph (GRAPH-3) for accessing each vertex during the traversal, and printing the difference between the total distances of two
traversal methods...
Total distance is: 260,000000
```



The created graph is in the previous page is above.

BFS distance must be

V-0 = 0

V-1 = 10

V-2 = 20

V-3 = 30

V-4 = 100

Total = 160

DFS distance must be

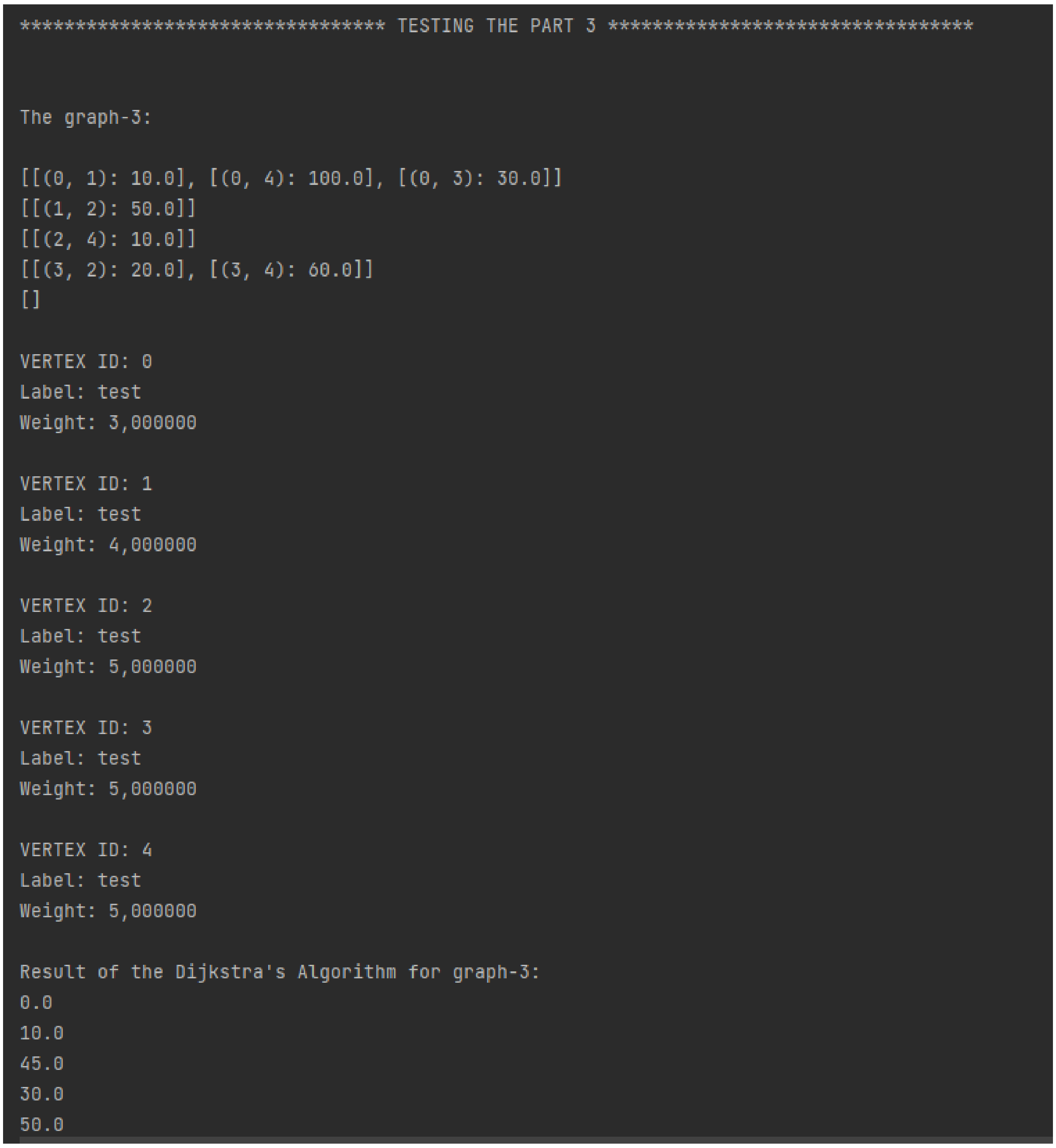
10+50+10+30 =100

Total = 260

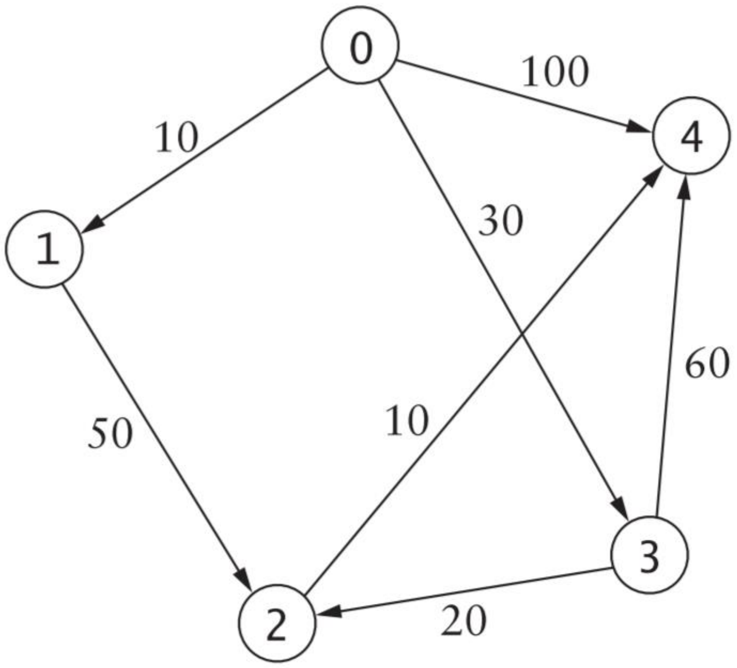
Total distance must be 160+100 = 260, and the test result shows us 260. My algorithm can run correctly!

PART-3 TEST

Graph-3 is used for this test.



```
********************************** TESTING THE PART 3 **********************************



The graph-3:

[[(0, 1): 10.0], [(0, 4): 100.0], [(0, 3): 30.0]]
[[(1, 2): 50.0]]
[[(2, 4): 10.0]]
[[(3, 2): 20.0], [(3, 4): 60.0]]
[]

VERTEX ID: 0
Label: test
Weight: 3,000000

VERTEX ID: 1
Label: test
Weight: 4,000000

VERTEX ID: 2
Label: test
Weight: 5,000000

VERTEX ID: 3
Label: test
Weight: 5,000000

VERTEX ID: 4
Label: test
Weight: 5,000000

Result of the Dijkstra's Algorithm for graph-3:
0.0
10.0
45.0
30.0
50.0
```



The created graph is above. Boosting values is indicated with red B.

V-0 = 0

V-1 = 10

V-2 = 30+20-5 = 45

V-3 = 30

V-4 = 30+20-5 + 10 -5 = 50

Output is int the previous page is true!

METHOD ANALYSES

- **newVertex = θ(1)**
- **addVertex = θ(1)**
- **addEdge = Calls insert method of graph, it is also θ(1). Also calls contains method θ(n). Total is θ(n)**
- **removeEdge = uses list iterator. θ(n)**
- **removeVertex = uses list iterator. θ(n)**
- **findVertexIndex (helper private method) =**
  Best case = θ(1)
  Worst case = θ(n)
- **filterVertices = It has a loop, and there is an inner loop with list iterator. Hashmap runs constant time in it. Also in the inner loop findVertexIndex is called. So,**
  Worst case = $\theta(n)^3$
  Best case = $\theta(n)^2$
- **exportMatrix = It has a for loop and list iterator loop. So, T(n) = $\theta(n)^2$**
- **printGraph = θ (n)**