GTU Department of Computer Engineering

CSE 222 / 505 – Spring 2022

Homework 8 - Report

Ahmet USLUOGLU

1801042602

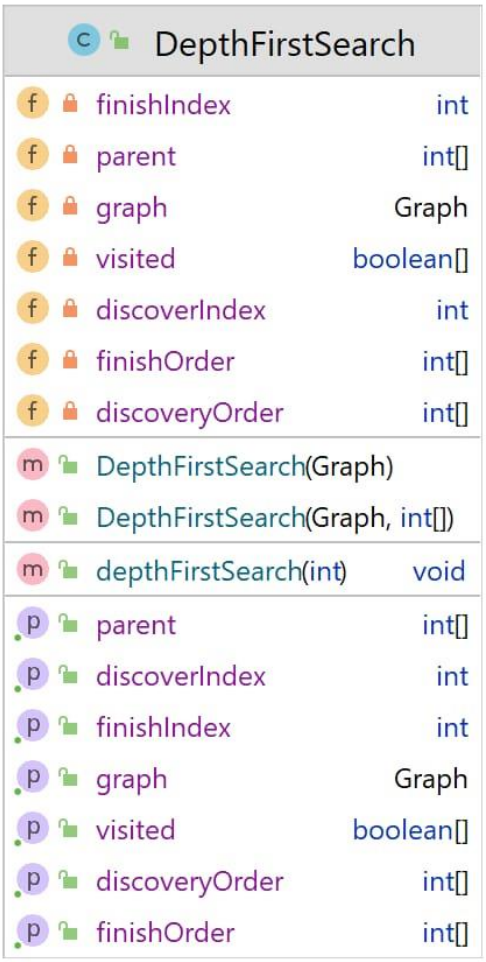
1 – System Requirement

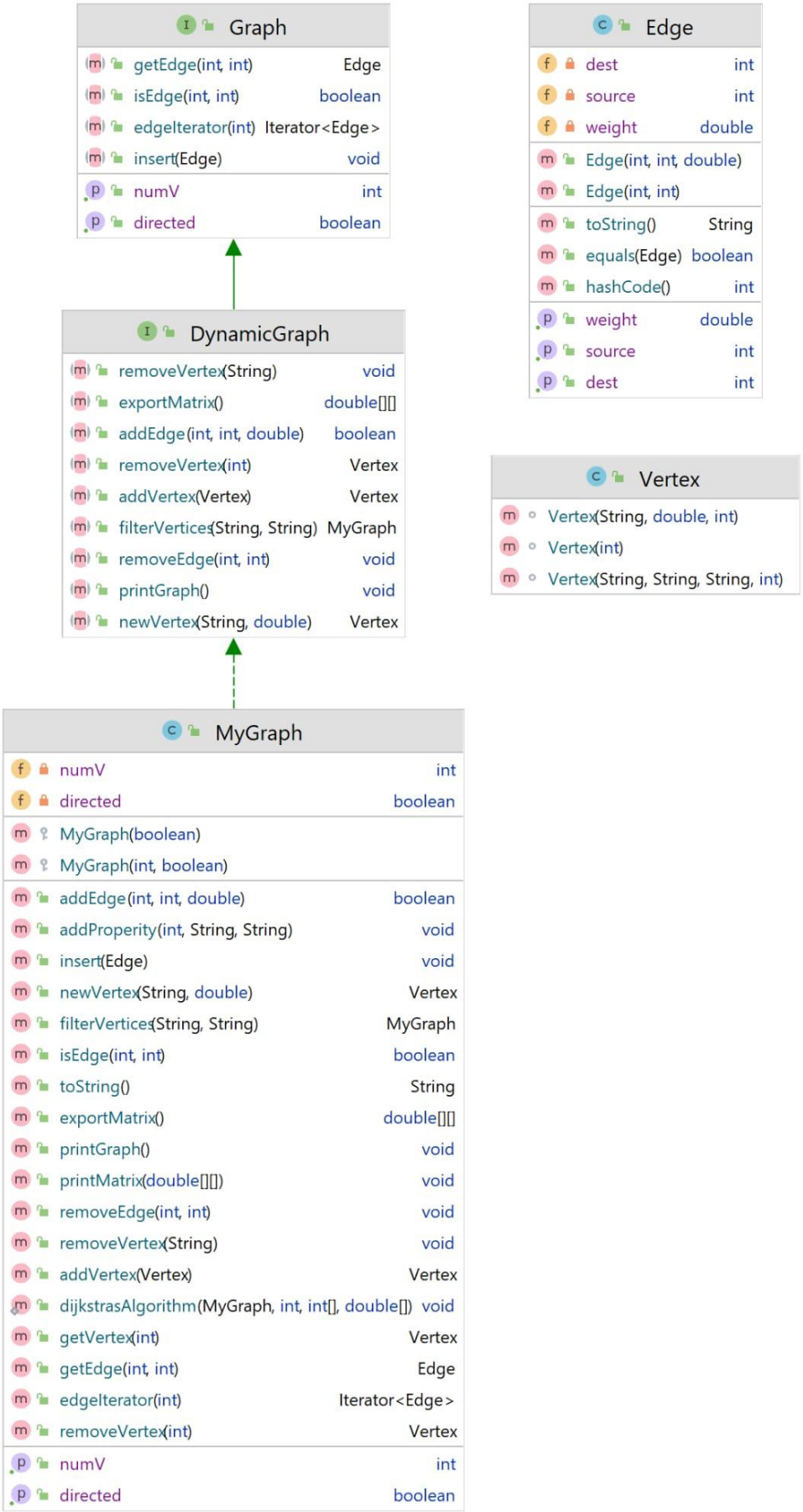
Operating System must have JDK (Java Development Kit) 11 and JRE (Java RuntimeEnvironment) 11 or higher.

There should be enough space for storing data's.

2 – Class Diagrams

*Higher Resolution Version of the Class Diagram is in the files.




Graph
getEdge(int, int) Edge
isEdge(int, int) boolean
edgeIterator(int) Iterator<Edge>
insert(Edge) void
numV int
directed boolean
Edge
dest int
source int
weight double
Edge(int, int, double)
Edge(int, int)
toString() String
equals(Edge) boolean
hashCode() int
weight double
source int
dest int
DynamicGraph
removeVertex(String) void
exportMatrix() double[][]
addEdge(int, int, double) boolean
removeVertex(int) Vertex
addVertex(Vertex) Vertex
filterVertices(String, String) MyGraph
removeEdge(int, int) void
printGraph() void
newVertex(String, double) Vertex
Vertex
Vertex(String, double, int)
Vertex(int)
Vertex(String, String, String, int)
MyGraph
numV int
directed boolean
MyGraph(boolean)
MyGraph(int, boolean)
addEdge(int, int, double) boolean
addProperity(int, String, String) void
insert(Edge) void
newVertex(String, double) Vertex
filterVertices(String, String) MyGraph
isEdge(int, int) boolean
toString() String
exportMatrix() double[][]
printGraph() void
printMatrix(double[][]) void
removeEdge(int, int) void
removeVertex(String) void
addVertex(Vertex) Vertex
dijkstrasAlgorithm(MyGraph, int, int[], double[]) void
getVertex(int) Vertex
getEdge(int, int) Edge
edgeIterator(int) Iterator<Edge>
removeVertex(int) Vertex
numV int
directed boolean

3. Problem - Solution Approach

Problem:

1 -) Define a DynamicGraph interface by extending the Graph interface in the book for the following definition of graph data structure. Write a MyGraph class for the implementation of DynamicGraph interface. In your implementation, define a Vertex class for representing the vertices in the graph. A vertex must have an index (ID), a label, and a weight. The vertices may have user-defined additional properties (Vertex class should be generic), so you have to handle this requirement. Use adjacency list representation to handle the edges between vertices in the graph data structure.

2-) Write a method that takes a MyGraph object as a parameter and performs BFS and DFS traversals. The method calculates the total distance of the path for accessing each vertex during the traversal, and it returns the difference between the total distances of two traversal methods. If there are more than one alternative to access a vertex at a specific level during the BFS, the shortest alternative should be considered. The vertices should be considered in distance order during DFS traversal, 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.

3-)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

Solutions:

1-) I have used HashMap to represent Adjacency List in MyGraph class which implements DynamicGraph Interface. This way most of the implementations are Constant Time.

I have used ID as key and Vertex as Value of the HashMap.

Every Vertex keeps Edge List which the source of each edge is The Vertex itself.

2-) I have used BFS and DFS examples in the book and modified the source code to our needs.

Each iteration in the edge list follows the least weighted edge. After taking the array of visited vertexes in each search.

I have added the total distance between those edges.

I have subtracted the total distances that I get from DFS and BFS to calculate difference.

3-) I have modified the Dijkstra's Algorithm by Subtracting Boosting Values of vertexes (if there is any) from the edge's weights that we iterated through except the start and final destination vertexes.

Calculated the least weighted path by substracting boosting value.

4 – Test Cases

a-) Testing the part1

1 – Creating an empty graph with n vertices.

2 – Adding new 2 vertices and 1 edge.

3 –Creating an empty graph and inserting edges.

4- Adding a new vertex.

5- Adding a new edge between empty vertexes.

6- Deleting an edge.

7- Deleting a vertex.

8- Exporting matrix version and print.

9- Testing isEdge function.

10- Adding Filter to Vertexes and Creating a subgraph.

b-) Testing the part2

1 -) Testing the Depth first search distance

2 -) Testing the Breadth first search distance

3- Testing the Difference between DFS and BFS.

c-) Testing the part3

1 -) Testing the Dijkstra's Algorithm

5 – Running Program and Results

a-) Testing the part1

1 – Creating an empty graph with n vertices.

```
Graph:
0=>
1=>
2=>
3=>
4=>
```

2 – Adding new 2 vertices and 1 edge.

```
Graph:
0=>
1=>
2=>
3=>
4=>
5=> [ (5 - 6)  W: 4.0 ]
6=> [ (6 - 5)  W: 4.0 ]
```

3 –Creating an empty graph and inserting edges.

```
Graph:
0=>
1=>
2=>
3=>
4=>
Graph:
0=> [ (0 - 1)  W: 1.0 ]  [ (0 - 2)  W: 2.0 ]
1=> [ (1 - 0)  W: 1.0 ]  [ (1 - 2)  W: 3.0 ]  [ (1 - 3)  W: 4.0 ]
2=> [ (2 - 0)  W: 2.0 ]  [ (2 - 1)  W: 3.0 ]
3=> [ (3 - 1)  W: 4.0 ]
4=>
```

4- Adding a new vertex.

```
Graph:
0=> [ (0 - 1)  W: 1.0 ]  [ (0 - 2)  W: 2.0 ]
1=> [ (1 - 0)  W: 1.0 ]  [ (1 - 2)  W: 3.0 ]  [ (1 - 3)  W: 4.0 ]
2=> [ (2 - 0)  W: 2.0 ]  [ (2 - 1)  W: 3.0 ]
3=> [ (3 - 1)  W: 4.0 ]
4=>
5=>
```

5- Adding a new edge between empty vertexes.

```
Graph:
0=> [ (0 - 1)  W: 1.0 ]  [ (0 - 2)  W: 2.0 ]
1=> [ (1 - 0)  W: 1.0 ]  [ (1 - 2)  W: 3.0 ]  [ (1 - 3)  W: 4.0 ]
2=> [ (2 - 0)  W: 2.0 ]  [ (2 - 1)  W: 3.0 ]
3=> [ (3 - 1)  W: 4.0 ]
4=> [ (4 - 5)  W: 5.0 ]
5=> [ (5 - 4)  W: 5.0 ]
```

6- Deleting an edge.

```
Graph:
0=> [ (0 - 1)  W: 1.0 ]  [ (0 - 2)  W: 2.0 ]
1=> [ (1 - 0)  W: 1.0 ]  [ (1 - 2)  W: 3.0 ]  [ (1 - 3)  W: 4.0 ]
2=> [ (2 - 0)  W: 2.0 ]  [ (2 - 1)  W: 3.0 ]
3=> [ (3 - 1)  W: 4.0 ]
4=>
5=>
```

7- Deleting a vertex.

```
Graph:
0=> [ (0 - 1)  W: 1.0 ]  [ (0 - 2)  W: 2.0 ]
1=> [ (1 - 0)  W: 1.0 ]  [ (1 - 2)  W: 3.0 ]  [ (1 - 3)  W: 4.0 ]
2=> [ (2 - 0)  W: 2.0 ]  [ (2 - 1)  W: 3.0 ]
3=> [ (3 - 1)  W: 4.0 ]
4=>
```

8- Exporting matrix version and print.

```
     0    1    2    3    4

0    -   1.0  2.0   -    -

1   1.0   -   3.0  4.0   -

2   2.0  3.0   -    -    -

3    -   4.0   -    -    -

4    -    -    -    -    -
```

9- Testing isEdge function.

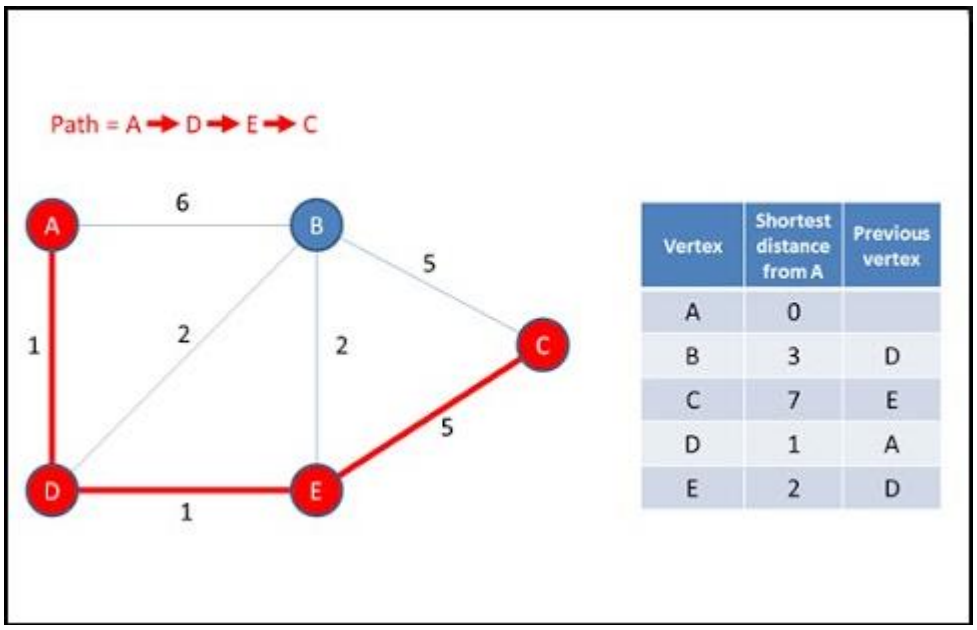
```
Is there a edge between 5 to 2?
g.isEdge(5, 2) = false
```

10- Adding Filter to Vertexes and Creating a subgraph.

```
Adding Key and Value properities to Vertexes 0 and 4

none  1.0  0  [Source: 0, Destination: 1Weight: 1.0, Source: 0, Destination: 2Weight: 2.0]  []
none  1.0  0  [Source: 0, Destination: 1Weight: 1.0, Source: 0, Destination: 2Weight: 2.0]  [value]
none  1.0  4  [Source: 4, Destination: 5Weight: 5.0]  []
none  1.0  4  [Source: 4, Destination: 5Weight: 5.0]  [value]
Printing The Subgraph with Key and Value properities.

Graph:
0=> [ (0 - 1)  W: 1.0 ]  [ (0 - 2)  W: 2.0 ]
4=> [ (4 - 5)  W: 5.0 ]
```

b-) Testing the part2

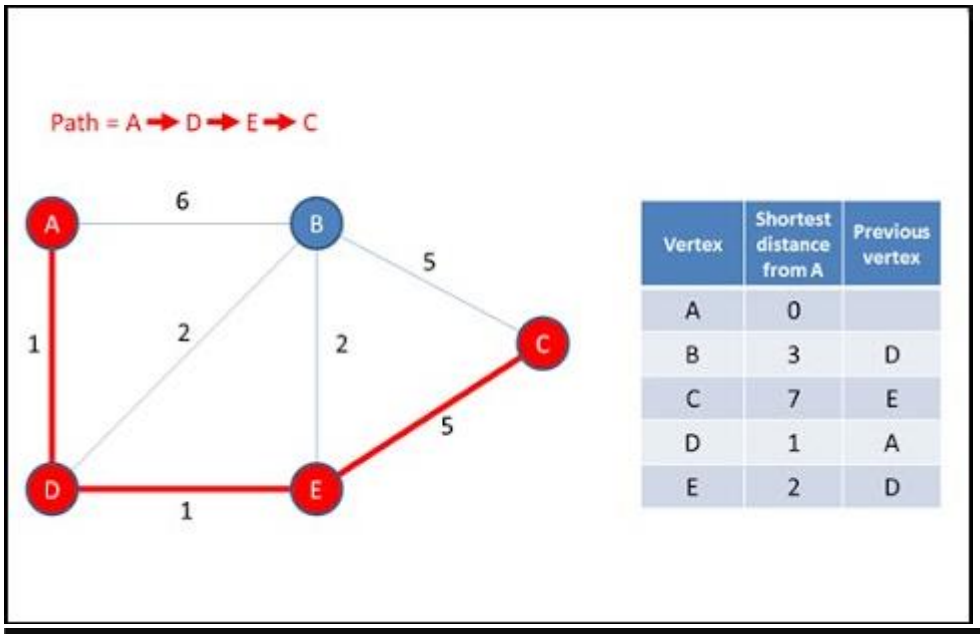
1 -) Testing the Depth first search distance



Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	D
C	7	E
D	1	A
E	2	D

```
Graph:
0=> [ (0 - 1)  W: 6.0 ]  [ (0 - 3)  W: 1.0 ]
1=> [ (1 - 0)  W: 6.0 ]  [ (1 - 2)  W: 5.0 ]  [ (1 - 3)  W: 2.0 ]  [ (1 - 4)  W: 2.0 ]
2=> [ (2 - 1)  W: 5.0 ]  [ (2 - 4)  W: 5.0 ]
3=> [ (3 - 0)  W: 1.0 ]  [ (3 - 1)  W: 2.0 ]  [ (3 - 4)  W: 1.0 ]
4=> [ (4 - 1)  W: 2.0 ]  [ (4 - 2)  W: 5.0 ]  [ (4 - 3)  W: 1.0 ]


     0    1    2    3    4

0    -   6.0   -   1.0   -

1   6.0   -   5.0  2.0  2.0

2    -   5.0   -    -   5.0

3   1.0  2.0   -    -   1.0

4    -   2.0  5.0  1.0   -

Discovery and finish order:
0 2
3 1
4 4
1 3
2 0
DFS Distance:9
```

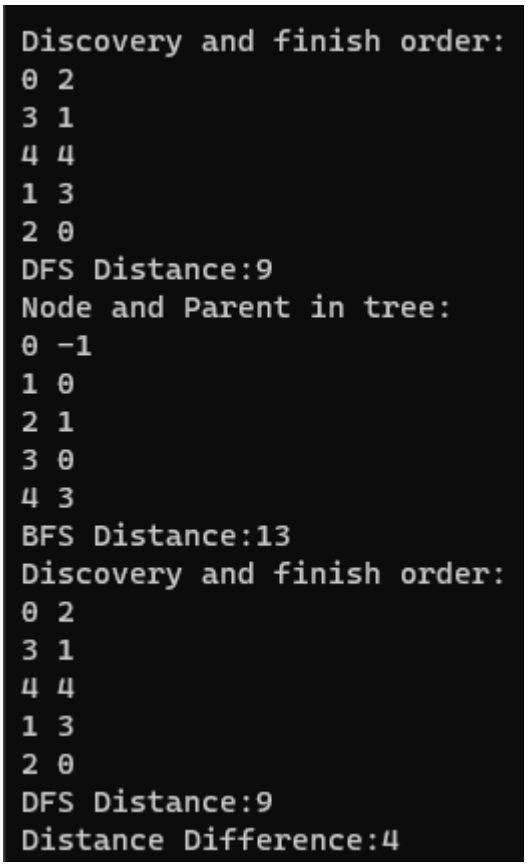
2 -) Testing the Breadth first search distance



Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	D
C	7	E
D	1	A
E	2	D

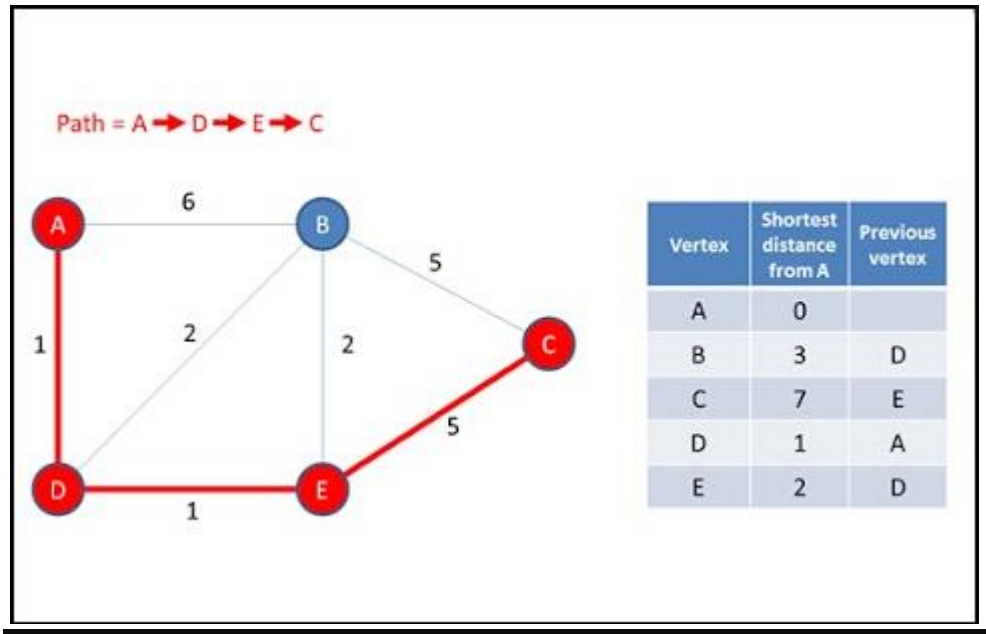
```
Graph:
0=> [ (0 - 1)  W: 6.0 ]  [ (0 - 3)  W: 1.0 ]
1=> [ (1 - 0)  W: 6.0 ]  [ (1 - 2)  W: 5.0 ]  [ (1 - 3)  W: 2.0 ]  [ (1 - 4)  W: 2.0 ]
2=> [ (2 - 1)  W: 5.0 ]  [ (2 - 4)  W: 5.0 ]
3=> [ (3 - 0)  W: 1.0 ]  [ (3 - 1)  W: 2.0 ]  [ (3 - 4)  W: 1.0 ]
4=> [ (4 - 1)  W: 2.0 ]  [ (4 - 2)  W: 5.0 ]  [ (4 - 3)  W: 1.0 ]


     0    1    2    3    4

0    -   6.0   -   1.0   -

1   6.0   -   5.0  2.0  2.0

2    -   5.0   -    -   5.0

3   1.0  2.0   -    -   1.0

4    -   2.0  5.0  1.0   -

Node and Parent in tree:
0 -1
1 0
2 1
3 0
4 3
BFS Distance:13
```

3- Testing the Difference between DFS and BFS.



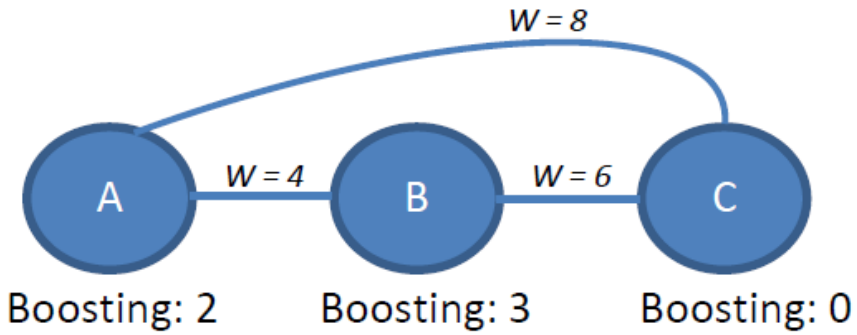
c-) Testing the part3

1 -) Testing the Dijkstra's Algorithm



Vertex	Shortest distance from A	Previous vertex
A	0	
B	3	D
C	7	E
D	1	A
E	2	D

```
Graph:
0=> [ (0 - 1)  W: 6.0 ]  [ (0 - 3)  W: 1.0 ]
1=> [ (1 - 0)  W: 6.0 ]  [ (1 - 2)  W: 5.0 ]  [ (1 - 3)  W: 2.0 ]  [ (1 - 4)  W: 2.0 ]
2=> [ (2 - 1)  W: 5.0 ]  [ (2 - 4)  W: 5.0 ]
3=> [ (3 - 0)  W: 1.0 ]  [ (3 - 1)  W: 2.0 ]  [ (3 - 4)  W: 1.0 ]
4=> [ (4 - 1)  W: 2.0 ]  [ (4 - 2)  W: 5.0 ]  [ (4 - 3)  W: 1.0 ]


     0    1    2    3    4

0    -   6.0   -   1.0   -

1   6.0   -   5.0  2.0  2.0

2    -   5.0   -    -   5.0

3   1.0  2.0   -    -   1.0

4    -   2.0  5.0  1.0   -

Node, Predecessor, and Distance:
A:        A             0.0
B:        D             3.0
C:        B             6.0
D:        A             1.0
E:        D             2.0
```




```
Node, Predecessor, and Distance:
0:          0               0.0
1:          0               4.0
2:          1               7.0
```

6 – Calculate Time complexity

1-) Vertex newVertex (String label, double weight);
Complexity O(1).

```java
/**
     * > The function `newVertex` creates a new vertex with the given label and weight, and returns it
     *
     * @param label The label of the vertex.
     * @param weight The weight of the vertex.
     * @return
A new Vertex object with the label, weight, and vertexCount.
     */
    @Override
    public Vertex newVertex (String label, double weight)
    {
        return new Vertex(label, weight, ++vertexCount);
    }
```

2-) Vertex addVertex (Vertex new_vertex);
Complexity O(1).

```java
/**
     * * The function takes in a vertex and adds it to the graph
     *
     * @param new_vertex The vertex to be added to the graph.
     * @return The new vertex that was added to the graph.
     */
    @Override
    public Vertex addVertex (Vertex new_vertex)
    {
        numV++;
        vertices.put(new_vertex.ID, new_vertex);
        return new_vertex;
    }
```

3-) boolean addEdge (int vertexID1, int vertexID2, double weight);
Complexity O(1).

```java
/**
     * This function adds an edge between two vertices
     *
     * @param vertexID1 The ID of the first vertex.
     * @param vertexID2 The ID of the vertex that the edge is going to.
     * @param weight The weight of the edge.
     * @return A boolean value.
     */
    @Override
    public boolean addEdge (int vertexID1, int vertexID2, double weight)
    {
        vertices.get(vertexID1).edges.add(new Edge(vertexID1, vertexID2, weight));

        if(!directed)
                vertices.get(vertexID2).edges.add(new Edge(vertexID2, vertexID1, weight));
        return true;
    }
```

4 -) void removeEdge (int vertexID1, int vertexID2);
Complexity O(1).

```java
/**
     * > Removes the edge between the two vertices with the given IDs
     *
     * @param vertexID1 The ID of the first vertex.
     * @param vertexID2 The ID of the second vertex.
     */
    @Override
    public void removeEdge (int vertexID1, int vertexID2)
    {
        vertices.get(vertexID1).edges.remove(getEdge(vertexID1, vertexID2));
        if(!directed) vertices.get(vertexID2).edges.remove(getEdge(vertexID2, vertexID1));
    }
```

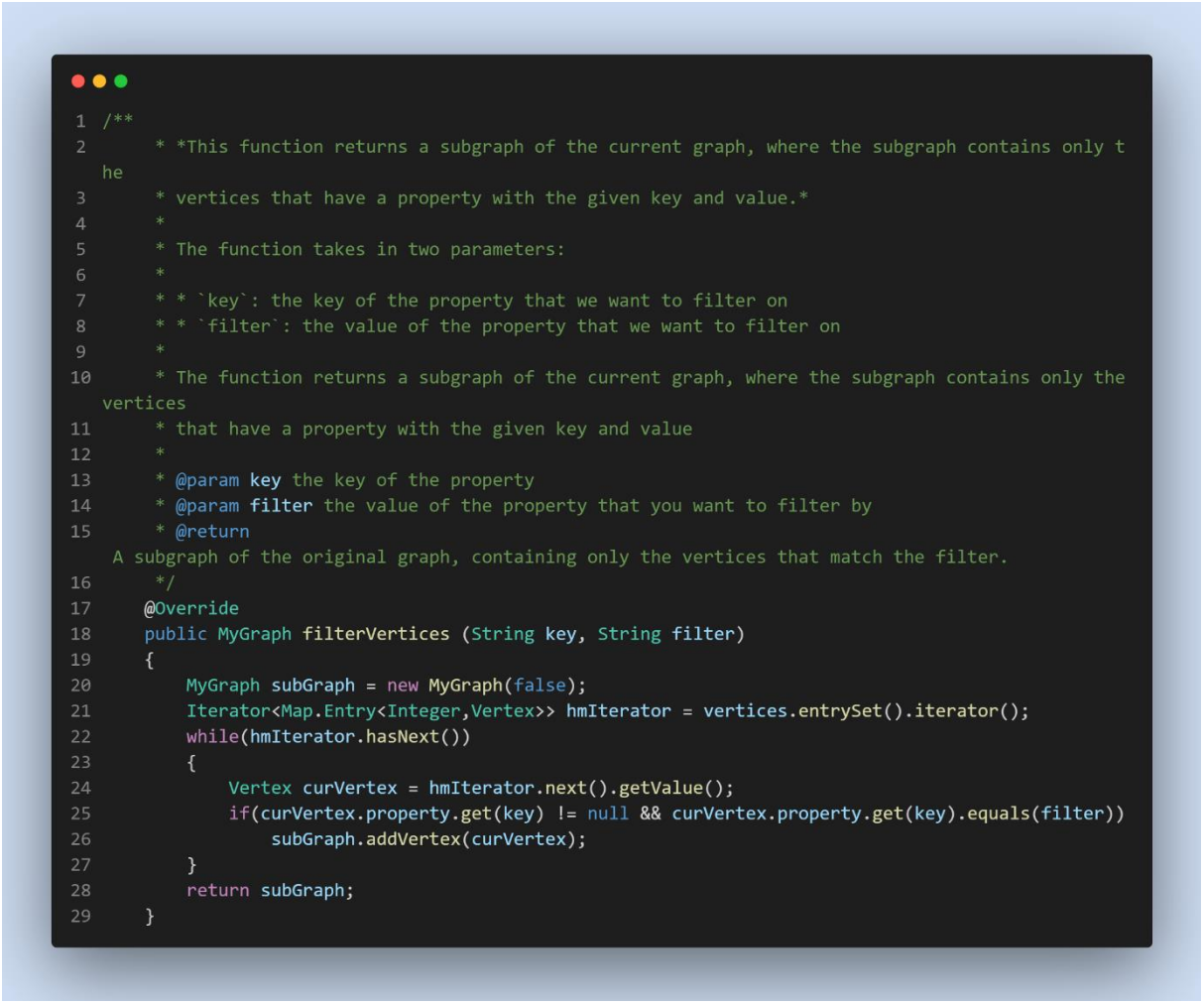
5-) Vertex removeVertex (int vertexID);
Complexity O(n*m).

```java
/**
     * > Remove the vertex with the given ID from the graph, and remove all edges that con
nect to it
     *
     * @param vertexID The ID of the vertex to be removed.
     * @return The vertex that was removed.
     */
    @Override
    public Vertex removeVertex (int vertexID)
    {
        Iterator<Map.Entry<Integer,Vertex>> hmIterator = vertices.entrySet().iterator();
        while(hmIterator.hasNext())
        {
            Vertex curVertex = hmIterator.next().getValue();
            curVertex.edges.remove(getEdge(curVertex.ID, vertexID));
        }
        return vertices.remove(vertexID);
    }
```

6-) void removeVertex (String label);
Complexity O(n*m * n) = O(n^2*m).

```java
/**
     * Iterate through the vertices, and if the label matches, remove the vertex.
     *
     * @param label The label of the vertex to be removed.
     */

    @Override
    public void removeVertex (String label)
    {
        Iterator<Map.Entry<Integer,Vertex>> hmIterator = vertices.entrySet().iterator();
        while(hmIterator.hasNext())
        {
            Vertex curVertex = hmIterator.next().getValue();
            if(curVertex.label.equals(label)) removeVertex(curVertex.ID);
        }
    }
```

7 -) MyGraph filterVertices (String key, String filter);
Complexity O(n).



```java
/**
     * *This function returns a subgraph of the current graph, where the subgraph contains only t
he
     * vertices that have a property with the given key and value.*
     *
     * The function takes in two parameters:
     *
     * * `key`: the key of the property that we want to filter on
     * * `filter`: the value of the property that we want to filter on
     *
     * The function returns a subgraph of the current graph, where the subgraph contains only the
vertices
     * that have a property with the given key and value
     *
     * @param key the key of the property
     * @param filter the value of the property that you want to filter by
     * @return
 A subgraph of the original graph, containing only the vertices that match the filter.
     */
    @Override
    public MyGraph filterVertices (String key, String filter)
    {
        MyGraph subGraph = new MyGraph(false);
        Iterator<Map.Entry<Integer,Vertex>> hmIterator = vertices.entrySet().iterator();
        while(hmIterator.hasNext())
        {
            Vertex curVertex = hmIterator.next().getValue();
            if(curVertex.property.get(key) != null && curVertex.property.get(key).equals(filter))
                subGraph.addVertex(curVertex);
        }
        return subGraph;
    }
```

8-) double[][] exportMatrix();
Complexity O(n^2).

```java
/**
     * We iterate through the vertices, and for each vertex, we iterate through its edges, and for each
     * edge, we set the corresponding element in the matrix to the weight of the edge
     *
     * @return A 2D array of doubles.
     */
    @Override
    public double[][] exportMatrix()
    {
        double[][] matrix = new double[numV][numV];
        Iterator<Map.Entry<Integer,Vertex>> hmIterator = vertices.entrySet().iterator();
        while(hmIterator.hasNext())
        {
            Vertex curVertex = hmIterator.next().getValue();
            for (var element : curVertex.edges) {
                matrix[element.getSource()][element.getDest()] = element.getWeight();
            }
        }
        return matrix;
    }
```

9-) void printGraph();
Complexity O(n^2).

```java
/**
     * It iterates through the vertices HashMap, and for each vertex, it iterates through its edges and
     * prints them
     */
    @Override
    public void printGraph()
    {
        System.out.print("Graph:\n");
        Iterator<Map.Entry<Integer,Vertex>> hmIterator = vertices.entrySet().iterator();
        while(hmIterator.hasNext())
        {
            Vertex curVertex = hmIterator.next().getValue();
            System.out.print(curVertex.ID + "=> ");
            for (var element : curVertex.edges)
            {
                System.out.print("[ ("+ element.getSource()+" - "+element.getDest()+")  W: " + element.getWeight() +" ]  ");
            }
            System.out.println();
        }
    }
```