

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

Yusuf Arslan  
200104004112

# 1. System Requirement

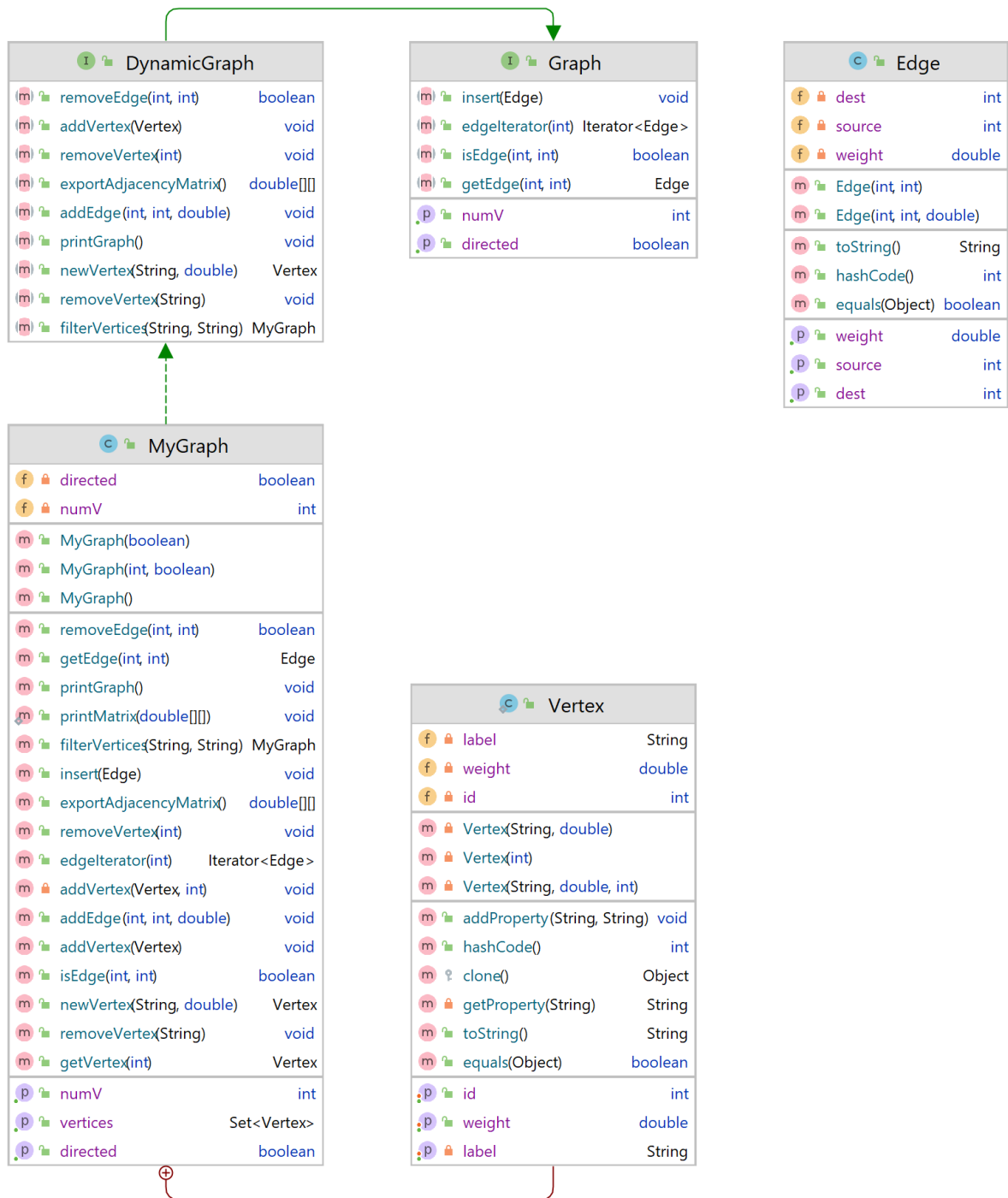
First of all the operating system should have Java virtual machine (JVM), Java development kit (JDK) and 'at least' Java runtime environment (JRE) – 11 in a linux distribution.

This application not interactive. User should compile and run program to see result of test of MyGraph application and other requested methods.

But, user is free to change tests which are in the App class.

## 2. Class Diagrams

### 1. MyGraph Diagrams



## 2. BFSDifferenceDFS Diagram

BFSDifferenceDFS		
f	<i>finishIndex</i>	int
f	<i>finishOrder</i>	Vertex[]
f	<i>totalDfsDistance</i>	int
f	<i>theVisited</i>	HashMap<Vertex, Boolean>
f	<i>discoveryIndex</i>	int
f	<i>theParent</i>	HashMap<Vertex, Vertex>
f	<i>discoveryOrder</i>	Vertex[]
f	<i>theGraph</i>	MyGraph
m	BFSDifferenceDFS()	
m	bfsDifferenceDfs(MyGraph, int)	double
m	findBfsTotalDistance(MyGraph, int)	double
m	findDfsTotalDistance(MyGraph, int)	double
m	depthFirstSearch(Vertex)	void

## 3. Dijkstras Diagram

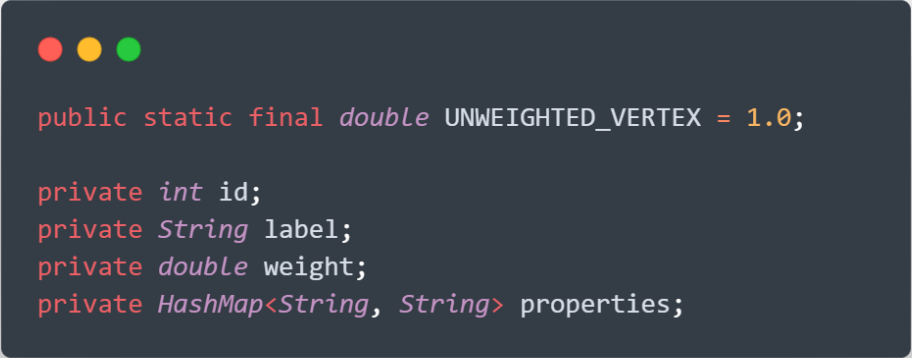
Dijkstras		
m	Dijkstras()	
m	dijkstras(MyGraph, Vertex, Map<Vertex, Vertex>, Map<Vertex, Double>)	void

### 3. Problem Solution Approaches and Code Analysis

#### 1. Analysing Vertex class

This class is a static public inner class of MyGraph class, user can construct a vertex object with weight, label and add properties to it but not the ID. The ID will be generated by MyGraph class since it is supposed to be unique.

Every method that will be used for Vertex class takes constant time and these methods will be wrapped by MyGraph methods.



```
public static final double UNWEIGHTED_VERTEX = 1.0;

private int id;
private String label;
private double weight;
private HashMap<String, String> properties;
```


Data fields of Vertex class.

To understand the program let's look at the MyGraph class.

#### 2. Analysing MyGraph class

MyGraph implements DynamicGraph which extends the Graph interface.

This class has NEXT\_ID (which will track the next ID that will be added to the graph), numV (keeps the number of vertices in the graph), directed (boolean value determines if the graph is directed or not), and adjList (which is a HashMap and its key is Vertex and the value is List of edges).



```
private int NEXT_ID = 0;
private int numV;
private boolean directed;
private HashMap<Vertex, List<Edge>> adjList;
```

Data fields of MyGraph class.

Let's look at the implementation of *Graph* interface methods then *DynamicGraph* interface methods.

a. `public void insert(Edge edge)`

-*Graph*-

Takes an edge and inserts it to the edge list of source vertex.



```
public void insert(Edge edge) {
    // Do not allow duplicate edges.
    if (isEdge(edge.getSource(), edge.getDest()))
        return;

    List<Edge> sourceEdges = adjList.get(new Vertex(edge.getSource()));
    sourceEdges.add(edge);
    if (!isDirected() && edge.getSource() != edge.getDest()) {
        List<Edge> destEdges = adjList.get(new Vertex(edge.getDest()));
        destEdges.add(new Edge(edge.getDest(), edge.getSource(), edge.getWeight()));
    }
}
```

Check if there is such a edge is  $O(n)$ . To add the edge to the vertex, first get the list of edges from the adjList, since it is a HashMap the get operation will be constant. Addition to the list also constant.

If the graph is undirected, then the edge will be add to the destination vertex aswell. So, all the operations will be  $O(n)$  where  $n$  is number of adjacent vertices to the source vertex.

## b. `public boolean isEdge(int source, int dest)` -Graph-

Takes a source and destination value, then determines if there is an edge between both vertices.

```
public boolean isEdge(int source, int dest) {  
    // Get the edge list of source vertex and check if edge exists.  
    return adjList.get(new Vertex(source)).  
        contains(new Edge(source, dest));  
}
```

First, it gets the edge list from the adjList, which is  $O(1)$  and then invokes contains method of List class which is  $O(n)$  where  $n$  is the number of adjacent vertices to the source vertex.

(In a dense graph,  $n$  will be number of vertices)

## c. `public Edge getEdge(int source, int dest)` -Graph-

Takes the source and destination value of the seeking edge and then returns the edge between the source and destination vertices if it exists, null otherwise.

```
public Edge getEdge(int source, int dest) {  
    // Get the edge list of source vertex and return the edge if it exists, or null.  
    List<Edge> edges = adjList.get(new Vertex(source));  
    if (edges == null || !edges.contains(new Edge(source, dest)))  
        return null;  
  
    return edges.get(edges.indexOf(new Edge(source, dest)));  
}
```

Gets the list of edges from source vertex which is  $O(1)$  and then checks if it contains in the edges list and finds the index of the edge, both take linear time. Therefore, it is

$O(n)$  where  $n$  is the number of adjacent vertices to the source vertex.

(In a dense graph,  $n$  will be number of vertices)

d. `public Iterator<Edge> edgeIterator(int id)`

-*Graph*-

Gets the edge list in the map then returns the iterator of ArrayList which is constant time  $O(1)$ .

```
public Iterator<Edge> edgeIterator(int id) {  
    return adjList.get(new Vertex(id)).iterator();  
}
```

e. `public Vertex newVertex(String label, double weight)`

-*DynamicGraph*-

Creates a new vertex and returns the reference of it. It is  $O(1)$ .

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



f. `public void addVertex(Vertex vertex)`

*-DynamicGraph-*

Adds a new vertex to the adjList, sets an NEXT\_ID to the vertex and increments the NEXT\_ID. Since the adjList is a HashMap the time complexity will be  $O(1)$ .

```
@Override
public void addVertex(Vertex vertex) {
    vertex.setId(NEXT_ID++);
    adjList.put(vertex, new ArrayList<Edge>());
    numV++;
}
```

g. `public void removeVertex(int id)`

*-DynamicGraph-*

It has to condition. One is graph undirected, other is directed.

```
public void removeVertex(int id) {
    // Remove all edges in the destination vertices.
    if (!isDirected()) {
        // If graph is not directed, find all edges in destination vertices and remove them.
        Iterator<Edge> edgeIterator = edgeIterator(id);
        while (edgeIterator.hasNext()) {
            Edge edge = edgeIterator.next();
            edgeIterator.remove();
            removeEdge(edge.getDest(), edge.getSource());
        }
    }
    else {
        // If graph is directed, remove all edges that connected to the vertex.
        Iterator<Vertex> vertices = adjList.keySet().iterator();
        while (vertices.hasNext()) {
            Vertex current = vertices.next();
            removeEdge(current.getId(), id);
        }
    }
    // Then set corresponding vertex to null.
    adjList.remove(new Vertex(id));
    // Add the id to the queue of removed vertices.
    numV--;
}
```

If the graph is undirected, it removes all edges which connected vertices in reverse order.

In a dense graph the time complexity of this operation will be  $O(n^2)$  where  $n$  is the number of vertices.

But in a sparse graph it will take  $O(1)$ .

If the graph is undirected, first, it will find the set of all vertices and its iterator which its cost is  $O(n)$ . Then it iterates all vertices to remove edge that was connected the removing vertex. Every removal operation is  $O(m)$  where  $m$  is number of adjacent vertex to the seeking vertex.

In a dense graph it is  $O(n^2)$  and in a sparse graph it is  $O(n)$  since we need to traverse all vertices.

## h. `public void removeVertex(String label)`

*-DynamicGraph-*

It finds the ID of the key from the label, so this operation is  $O(n)$  where  $n$  is the number of vertices. Then invokes the `removeVertex(int id)` method.

Best case time complexity will be  $O(n)$  for the undirected graph which was  $O(1)$ .

```
public void removeVertex(String label) {  
    // Find the vertex with the given label and remove it.  
    Iterator<Vertex> vertices = adjList.keySet().iterator();  
  
    while (vertices.hasNext())  
    {  
        Vertex current = vertices.next();  
        if (current.getLabel().equals(label))  
        {  
            removeVertex(current.getId());  
            return;  
        }  
    }  
}
```

i. `public void addEdge(int s, int d, double w)`  
-*DynamicGraph*-

Create a new edge and then invoke the insert method which its time complexity was  $O(n)$ .

```
public void addEdge(int source, int dest, double weight) {  
    // Create edge with IDs of source and destination and weight.  
    Edge edge = new Edge(source, dest, weight);  
  
    insert(edge);  
}
```

j. `public boolean removeEdge(int s, int d)`  
-*DynamicGraph*-

Finds the edge list of the source vertex then remove the seeking edge from the list if it is exist.

This operation is  $O(n)$  where  $n$  is the number of adjacent vertices to the source vertex.

If the graph is not directed, remove edge in destination vertex aswell. It is also  $O(n)$ .

```
public boolean removeEdge(int source, int dest) {  
    // Remove the edge from the vertices.  
  
    // Get list of edges from source vertex.  
    List<Edge> sourceEdges = adjList.get(new Vertex(source));  
  
    boolean removed = sourceEdges.remove(new Edge(source, dest));  
  
    if (removed && !isDirected() && source != dest)  
        adjList.get(new Vertex(dest)).remove(new Edge(dest, source));  
  
    return removed;  
}
```

The time complexity is  $O(n)$ .

## k. `public double[][] exportAdjacencyMatrix()` *-DynamicGraph-*

To export a matrix, create a 2D double array then set all elements `Double.POSITIVE_INFINITY`. This operation is  $O(n^2)$ . Then, to set ID's from 0 to  $n-1$ , first I mapped the previous ID with their first ID. Getting iterator is  $O(n)$  and also iteration will be  $O(n)$ .

Finally, iterate all edges then set all weights to the matrix. This operation is also  $O(n^2)$ .

```
public double[][] exportAdjacencyMatrix() {  
  
    // To create the adjacency matrix, create a new array with the number of vertices.  
    // Then, change ID's to 0-based index.  
    // To compress the matrix, we will ignore null  
    // indexes and store weights from 0 to number of vertices.  
  
    // Create the adjacency matrix.  
    double adjacencyMatix[][] = new double[numV][numV];  
    // Fill the array with Double.POSITIVE_INFINITY.  
    for (int i = 0; i < numV; i++) {  
        for (int j = 0; j < numV; j++) {  
            adjacencyMatix[i][j] = Double.POSITIVE_INFINITY;  
        }  
    }  
  
    // Iterate through the vertices and rearrange the previous IDs with the matrix graph IDs from 0 to n - 1.  
    Iterator<Vertex> vertices = adjList.keySet().iterator();  
    HashMap<Integer, Integer> idMap = new HashMap<Integer, Integer>();  
    for (int i = 0; i < numV; i++)  
    {  
        Vertex current = vertices.next();  
        idMap.put(current.getId(), i);  
    }  
  
    vertices = adjList.keySet().iterator();  
    while (vertices.hasNext()) {  
        Vertex current = vertices.next();  
        // Get the list of edges from the current vertex.  
        List<Edge> edges = adjList.get(current);  
        // Iterate through the edges.  
        for (Edge edge : edges) {  
            // Set the weight of the edge in the adjacency matrix.  
            adjacencyMatix[idMap.get(current.getId())][idMap.get(edge.getDest())] = edge.getWeight();  
        }  
    }  
  
    return adjacencyMatix;  
}
```

The total time complexity is  $O(n^2)$ .

## 1. public MyGraph filterVertices(String key, String filter)

-*DynamicGraph*-

First set all vertices which satisfy the filter to the filteredGraph,  $O(n)$ . Then, get this vertices edge lists from the original list to check if there is any edge between the filtered vertices. If there is, then edge them in the filtered graph aswell. Iterating vertices is  $O(n)$ , iterating through the edges is  $O(n)$  and adding edge to the vertices also  $O(n)$ .

```
public MyGraph filterVertices(String key, String filter) {  
  
    MyGraph filteredGraph = new MyGraph(isDirected());  
  
    // Iterate through the vertices and add the vertices that match the filter.  
    Iterator<Vertex> vertices = adjList.keySet().iterator();  
  
    while (vertices.hasNext()) {  
        Vertex current = vertices.next();  
        if (current.getProperty(key).equals(filter))  
        {  
            filteredGraph.addVertex(((Vertex)current.clone()), current.getId());  
        }  
    }  
  
    Iterator<Vertex> filteredVertices = filteredGraph.adjList.keySet().iterator();  
    while (filteredVertices.hasNext()) {  
        Vertex current = filteredVertices.next();  
        // Get the list of edges from the current vertex.  
        List<Edge> edges = adjList.get(current);  
        for (Edge edge : edges) {  
            if (filteredGraph.getVertex(edge.getDest()) != null)  
            {  
                filteredGraph.addEdge(edge.getSource(), edge.getDest(), edge.getWeight());  
            }  
        }  
    }  
    return filteredGraph;  
}
```

So, in a dense graph filteringVertices is  $O(n^3)$  and in a sparse graph it is  $O(n)$  since the iteration on edges will be constant and adding edge will be constant aswell.

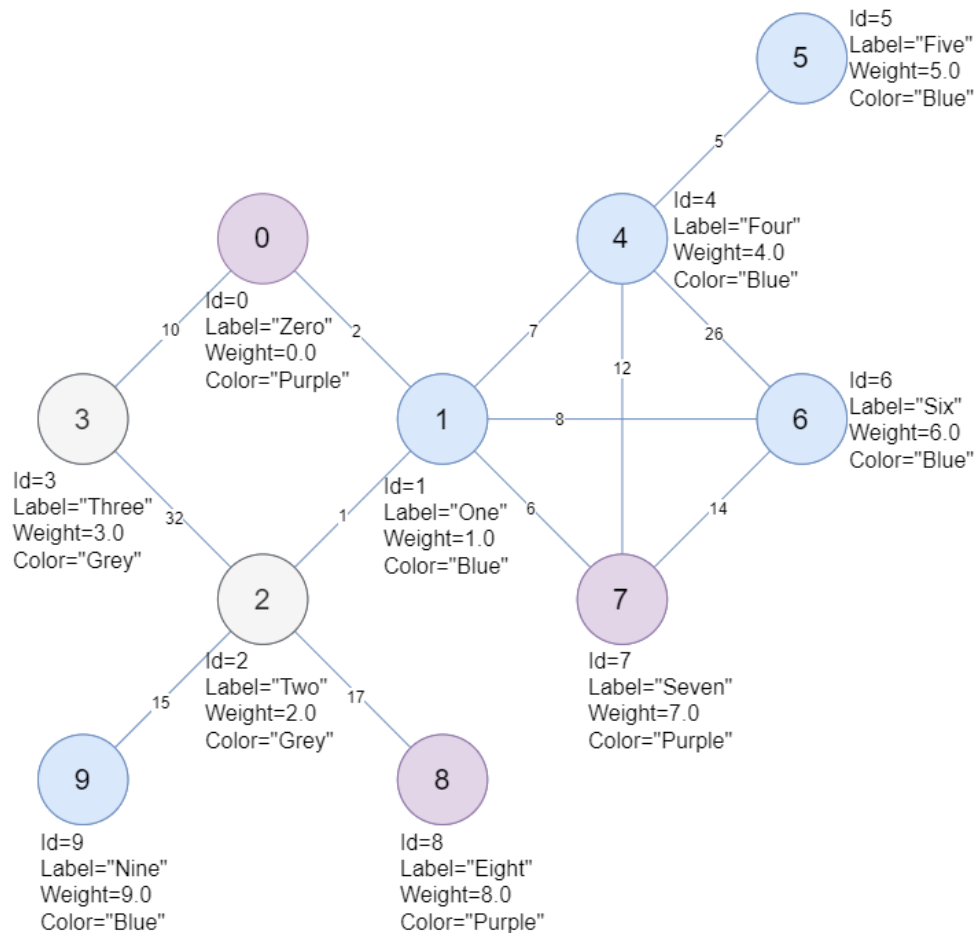
## m. public void printGraph()

-*DynamicGraph*-

Printing graph is just iterating all edges. Therefore, in a dense graph it is  $O(n^2)$  and  $O(n)$  in a sparse graph.

## 4. Test Cases

Let's construct a graph and makes operation one by one.



1. Case: Create vertices with given properties and add them to the graph.
2. Case: Add edges to the vertices.
3. Case: Remove edges from the graph.
4. Case: Remove vertex with given id.
5. Case: Remove vertex with given label.
6. Case: Filter vertices according to key-filter pair.
7. Case: Export the adjacent matrix.
8. Case: Print the graph in each step.

## 5. Running and Results

```
--- MyGraph Implementation and Running Tests ---

--Echo[1] An undirected graph has been created.

vertex 0: [(-1 : "Zero") : 0.0]
vertex 1: [(-1 : "One") : 1.0]
vertex 2: [(-1 : "Two") : 2.0]
vertex 3: [(-1 : "Three") : 3.0]
vertex 4: [(-1 : "Four") : 4.0]
vertex 5: [(-1 : "Five") : 5.0]
vertex 6: [(-1 : "Six") : 6.0]
vertex 7: [(-1 : "Seven") : 7.0]
vertex 8: [(-1 : "Eight") : 8.0]
vertex 9: [(-1 : "Nine") : 9.0]

--Echo[2] Vertices have been created.

--Echo[3] Vertices have been added to the graph.
```

```
The Graph
[(0 : "Zero") : 0.0]
[(1 : "One") : 1.0]
[(2 : "Two") : 2.0]
[(3 : "Three") : 3.0]
[(4 : "Four") : 4.0]
[(5 : "Five") : 5.0]
[(6 : "Six") : 6.0]
[(7 : "Seven") : 7.0]
[(8 : "Eight") : 8.0]
[(9 : "Nine") : 9.0]
```

```
--Echo[4] Edges have been added to the graph.

The Graph
[(0 : "Zero") : 0.0] → [(0,1) : 2.0] → [(0,3) : 10.0]
[(1 : "One") : 1.0] → [(1,0) : 2.0] → [(1,4) : 7.0] → [(1,6) : 8.0] → [(1,7) : 6.0] → [(1,2) : 1.0]
[(2 : "Two") : 2.0] → [(2,1) : 1.0] → [(2,3) : 32.0] → [(2,8) : 17.0] → [(2,9) : 15.0]
[(3 : "Three") : 3.0] → [(3,0) : 10.0] → [(3,2) : 32.0]
[(4 : "Four") : 4.0] → [(4,1) : 7.0] → [(4,5) : 5.0] → [(4,6) : 26.0] → [(4,7) : 12.0]
[(5 : "Five") : 5.0] → [(5,4) : 5.0]
[(6 : "Six") : 6.0] → [(6,1) : 8.0] → [(6,4) : 26.0] → [(6,7) : 14.0]
[(7 : "Seven") : 7.0] → [(7,1) : 6.0] → [(7,4) : 12.0] → [(7,6) : 14.0]
[(8 : "Eight") : 8.0] → [(8,2) : 17.0]
[(9 : "Nine") : 9.0] → [(9,2) : 15.0]
```

```
--Echo[5] Adjacency Matrix exported.
```

```
The Matrix
```

	0	1	2	3	4	5	6	7	8	9
0	inf	2.0	inf	10.0	inf	inf	inf	inf	inf	inf
1	2.0	inf	1.0	inf	7.0	inf	8.0	6.0	inf	inf
2	inf	1.0	inf	32.0	inf	inf	inf	inf	17.0	15.0
3	10.0	inf	32.0	inf	inf	inf	inf	inf	inf	inf
4	inf	7.0	inf	inf	inf	5.0	26.0	12.0	inf	inf
5	inf	inf	inf	inf	5.0	inf	inf	inf	inf	inf
6	inf	8.0	inf	inf	26.0	inf	inf	14.0	inf	inf
7	inf	6.0	inf	inf	12.0	inf	14.0	inf	inf	inf
8	inf	inf	17.0	inf	inf	inf	inf	inf	inf	inf
9	inf	inf	15.0	inf	inf	inf	inf	inf	inf	inf



```
--Echo[6] Edge (0,1) has been removed.
```

```
The Graph
```

```
[(0 : "Zero") : 0.0] -> [(0,3) : 10.0]
[(1 : "One") : 1.0] -> [(1,4) : 7.0] -> [(1,6) : 8.0] -> [(1,7) : 6.0] -> [(1,2) : 1.0]
[(2 : "Two") : 2.0] -> [(2,1) : 1.0] -> [(2,3) : 32.0] -> [(2,8) : 17.0] -> [(2,9) : 15.0]
[(3 : "Three") : 3.0] -> [(3,0) : 10.0] -> [(3,2) : 32.0]
[(4 : "Four") : 4.0] -> [(4,1) : 7.0] -> [(4,5) : 5.0] -> [(4,6) : 26.0] -> [(4,7) : 12.0]
[(5 : "Five") : 5.0] -> [(5,4) : 5.0]
[(6 : "Six") : 6.0] -> [(6,1) : 8.0] -> [(6,4) : 26.0] -> [(6,7) : 14.0]
[(7 : "Seven") : 7.0] -> [(7,1) : 6.0] -> [(7,4) : 12.0] -> [(7,6) : 14.0]
[(8 : "Eight") : 8.0] -> [(8,2) : 17.0]
[(9 : "Nine") : 9.0] -> [(9,2) : 15.0]
```

```
--Echo[7] Edge (2,3) has been removed.
```

```
The Graph
```

```
[(0 : "Zero") : 0.0] -> [(0,3) : 10.0]
[(1 : "One") : 1.0] -> [(1,4) : 7.0] -> [(1,6) : 8.0] -> [(1,7) : 6.0] -> [(1,2) : 1.0]
[(2 : "Two") : 2.0] -> [(2,1) : 1.0] -> [(2,8) : 17.0] -> [(2,9) : 15.0]
[(3 : "Three") : 3.0] -> [(3,0) : 10.0]
[(4 : "Four") : 4.0] -> [(4,1) : 7.0] -> [(4,5) : 5.0] -> [(4,6) : 26.0] -> [(4,7) : 12.0]
[(5 : "Five") : 5.0] -> [(5,4) : 5.0]
[(6 : "Six") : 6.0] -> [(6,1) : 8.0] -> [(6,4) : 26.0] -> [(6,7) : 14.0]
[(7 : "Seven") : 7.0] -> [(7,1) : 6.0] -> [(7,4) : 12.0] -> [(7,6) : 14.0]
[(8 : "Eight") : 8.0] -> [(8,2) : 17.0]
[(9 : "Nine") : 9.0] -> [(9,2) : 15.0]
```



```
--Echo[8] Vertex 1 has been removed.
```

```
The Graph
```

```
[(0 : "Zero") : 0.0] -> [(0,3) : 10.0]
[(2 : "Two") : 2.0] -> [(2,8) : 17.0] -> [(2,9) : 15.0]
[(3 : "Three") : 3.0] -> [(3,0) : 10.0]
[(4 : "Four") : 4.0] -> [(4,5) : 5.0] -> [(4,6) : 26.0] -> [(4,7) : 12.0]
[(5 : "Five") : 5.0] -> [(5,4) : 5.0]
[(6 : "Six") : 6.0] -> [(6,4) : 26.0] -> [(6,7) : 14.0]
[(7 : "Seven") : 7.0] -> [(7,4) : 12.0] -> [(7,6) : 14.0]
[(8 : "Eight") : 8.0] -> [(8,2) : 17.0]
[(9 : "Nine") : 9.0] -> [(9,2) : 15.0]
```

```
--Echo[9] Adjacency Matrix exported.
```

```
The Matrix
```

	0	1	2	3	4	5	6	7	8
0	inf	inf	10.0	inf	inf	inf	inf	inf	inf
1	inf	inf	inf	inf	inf	inf	inf	inf	15.0
2	10.0	inf	inf	inf	inf	inf	inf	inf	inf
3	inf	inf	inf	inf	5.0	26.0	12.0	inf	inf
4	inf	inf	inf	5.0	inf	inf	inf	inf	inf
5	inf	inf	inf	26.0	inf	inf	14.0	inf	inf
6	inf	inf	inf	12.0	inf	14.0	inf	inf	inf
7	inf	17.0	inf	inf	inf	inf	inf	inf	inf
8	inf	15.0	inf	inf	inf	inf	inf	inf	inf





```
--Echo[10] Graph filtered to only show vertices with the property "Color" = "Blue".
```

The Filtered Graph

```
[(4 : "Four") : 4.0] -> [(4,5) : 5.0] -> [(4,6) : 26.0]
```

```
[(5 : "Five") : 5.0] -> [(5,4) : 5.0]
```

```
[(6 : "Six") : 6.0] -> [(6,4) : 26.0]
```

```
[(9 : "Nine") : 9.0]
```

```
--Echo[11] Adjacency Matrix of filtered graph exported.
```

The Matrix

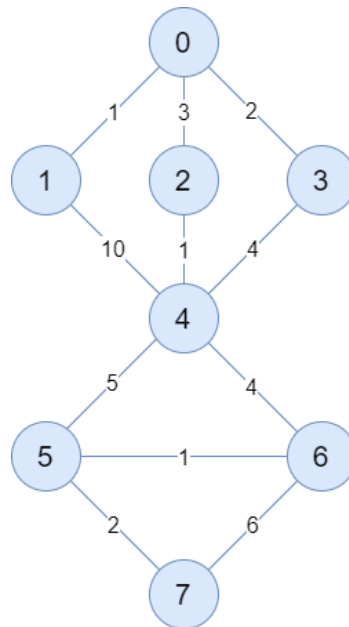
	0	1	2	3
0	inf	5.0	26.0	inf
1	5.0	inf	inf	inf
2	26.0	inf	inf	inf
3	inf	inf	inf	inf

## a. Question 2 (Distance difference of costum BFS and DFS algorithm)

In BFS algorithm every vertices are exploring level by level. To find the shortest path to a vertex in a specific level, I kept the levels of every vertex and during the traversing if there is a smallest path to vertex, I changed the parent of that vertex.

In DFS algorithm, I sorted the edges according to the their weight and provide a traversing through the smallest edge as requested in the homework.

To test methods construct the graph below to test the methods.



In BFS algorithm -Starts from 0-:

Parents = [null, 0, 0, 0, 2, 4, 4, 5] in order,

Distances = [0, 1, 3, 2, 1, 5, 4, 2] in order, Total distance = 18.

In DFS algorithm -Starts from 0-:

Finish order = [2, 3, 7, 5, 6, 4, 1, 0] in reverse order,

Total Distance =  $1 + 10 + 4 + 1 + 2 + 3 + 2 = 23$ .

Difference is 5.

Let's see the code then result of tests.

```

private static double findBfsTotalDistance(MyGraph graph, int initial)
{
    // Determine the level since we are looking shortest path from initial to certain level.
    int level = 0;

    // If a vertex is identified as visited, it means that the level has been changed.
    boolean isIdentified = true;

    // Get initial vertex.
    Vertex start = graph.getVertex(initial);

    // Create a queue for BFS.
    Queue<Vertex> theQueue = new LinkedList<Vertex>();

    // Create a map for keep the shortest distance.
    HashMap<Vertex, Double> distances = new HashMap<Vertex, Double>();

    // Set distance of all vertices to 0.0
    Set<Vertex> vertices = graph.getVertices();
    for (Vertex v : vertices)
    {
        distances.put(v, 0.0);
    }

    // Parent of vertices.
    HashMap<Vertex, Vertex> parent = new HashMap<Vertex, Vertex>();

    // Create a map to keep if a vertex is visited.
    HashMap<Vertex, Boolean> identified = new HashMap<Vertex, Boolean>();
    for (Vertex v : vertices)
    {
        identified.put(v, false);
    }

    // Map to keep the level of each vertex.
    HashMap<Vertex, Integer> levelMap = new HashMap<>();

    levelMap.put(start, 0);
    identified.put(start, true);
    theQueue.offer(start);

    // BFS algorithm to find the total distance from initial to certain level.
    while (!theQueue.isEmpty())
    {
        Vertex current = theQueue.poll();
        Iterator<Edge> itr = graph.edgeIterator(current.getId());
        if (isIdentified)
            level++;
        isIdentified = false;

        while (itr.hasNext())
        {
            Edge edge = itr.next();
            Vertex neighbor = graph.getVertex(edge.getDest());
            // If the neighbor is not visited, add it to the queue.
            if (!identified.get(neighbor))
            {
                identified.put(neighbor, true);
                theQueue.offer(neighbor);
                parent.put(neighbor, current);
                distances.put(neighbor, distances.get(current) + edge.getWeight());
                levelMap.put(neighbor, level);
                isIdentified = true;
            }
            else
            {
                // Check if in level - 1, there is a shortest path to the neighbor, set the short distance.
                if (levelMap.get(neighbor) - 1 == levelMap.get(current) &&
                    distances.get(neighbor) > distances.get(current) + edge.getWeight())
                {
                    distances.put(neighbor, distances.get(current) + edge.getWeight());
                    parent.put(neighbor, current);
                }
            }
        }
    }

    // System.out.println("levels" + levelMap);
    int bfsDistance = 0;
    for (var d : distances.keySet())
    {
        // System.out.println("Vertex " + d.getId() + ": " + distances.get(d));
        bfsDistance += (distances.get(d) - (parent.get(d) == null ? 0 : distances.get(parent.get(d))));
    }

    // System.out.println("parent: " + parent);
    return bfsDistance;
}

```

```

private static double findDfsTotalDistance(MyGraph graph, int current)
{
    theGraph = graph;
    int n = theGraph.getNumV();
    theParent = new HashMap<Vertex, Vertex>(n);
    theVisited = new HashMap<Vertex, Boolean>(n);
    discoveryOrder = new Vertex[n];
    finishOrder = new Vertex[n];
    discoveryIndex = 0;
    finishIndex = 0;
    // Get vertices set
    Set<Vertex> vertices = theGraph.getVertices();
    // Iterate through vertices
    for (Vertex v : vertices) {
        // If the vertex has not been visited
        theParent.put(v, null);
    }

    vertices = theGraph.getVertices();

    for (Vertex v : vertices) {
        theVisited.put(v, false);
    }

    totalDfsDistance = 0;

    vertices = theGraph.getVertices();
    for (Vertex v : vertices) {
        if (!theVisited.get(v)) {
            depthFirstSearch(v);
        }
    }
    return totalDfsDistance;
}

```

```

private static void depthFirstSearch(Vertex v)
{
    theVisited.put(v, true);
    discoveryOrder[discoveryIndex++] = v;
    Iterator<Edge> itr = theGraph.edgeIterator(v.getId());

    // Add edges to the arraylist and sort them by weight
    ArrayList<Edge> edges = new ArrayList<Edge>();
    while (itr.hasNext()) {
        edges.add(itr.next());
    }
    edges.sort(new Comparator<Edge>() {
        @Override
        public int compare(Edge o1, Edge o2) {
            return (o1.getWeight() - o2.getWeight()) >= 0 ? 1 : -1;
        }
    });
    itr = edges.iterator();

    while (itr.hasNext()) {
        Edge e = itr.next();
        Vertex w = theGraph.getVertex(e.getDest());
        if (!theVisited.get(w)) {
            theParent.put(w, v);
            totalDfsDistance += e.getWeight();
            depthFirstSearch(w);
        }
    }
    finishOrder[finishIndex++] = v;
}

```

Result of tests:

Vertex = [(id, "label" ) : weight]

Edge = [(source, destination) : weight]

```
----- BFS and DFS Difference Test -----

--Echo[1] A graph with size 8 created.

--Echo[2] Edges have been added to the graph.

The Graph
[(0 : "") : 1.0] → [(0,1) : 1.0] → [(0,2) : 3.0] → [(0,3) : 2.0]
[(1 : "") : 1.0] → [(1,0) : 1.0] → [(1,4) : 10.0]
[(2 : "") : 1.0] → [(2,0) : 3.0] → [(2,4) : 1.0]
[(3 : "") : 1.0] → [(3,0) : 2.0] → [(3,4) : 4.0]
[(4 : "") : 1.0] → [(4,1) : 10.0] → [(4,2) : 1.0] → [(4,3) : 4.0] → [(4,5) : 5.0] → [(4,6) : 4.0]
[(5 : "") : 1.0] → [(5,4) : 5.0] → [(5,6) : 1.0] → [(5,7) : 2.0]
[(6 : "") : 1.0] → [(6,4) : 4.0] → [(6,5) : 1.0] → [(6,7) : 6.0]
[(7 : "") : 1.0] → [(7,5) : 2.0] → [(7,6) : 6.0]

--Echo[3] Print the difference of distances between BFS and DFS algorithm if the smallest path chosen.

BFS distance: 38.0
DFS distance: 23.0
Difference: 15.0
```

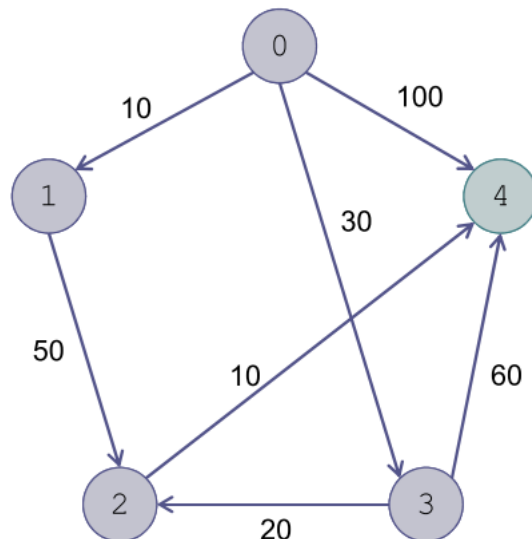
## b. Question 3 (Dijkstra algorithm with boosting values)

Dijkstra algorithms finds the shortest path to each vertex in a weighted graph.

During the scope of the homework we added the boosting values as a property to the vertices and changes the Dijkstra algorithm to be applicable for the MyGraph object.

Let's construct the graph in the lecture book and and see the results of Dijkstra algorithm with boosting Vertex 1.

v	d[v]	p[v]
1	10	0
2	50	3
3	30	0
4	60	2



Boost 1 with 20.

Results will be:

v	d[v]	p[v]
1	10	0
2	40	1
3	30	0
4	50	2

Let's see the code and command results.

```

public static void dijkstras(MyGraph graph, Vertex start, Map<Vertex,Vertex> pred, Map<Vertex, Double> dist)
{
    int numV = graph.getNumV();
    //Initialize V - S
    HashSet<Vertex> vMinusS = new HashSet<Vertex>(numV);
    Iterator<Vertex> itr = graph.getVertices().iterator();
    while (itr.hasNext())
    {
        Vertex v = itr.next();
        if (!v.equals(start))
        {
            vMinusS.add(v);
        }
    }
    for (Vertex v : vMinusS)
    {
        pred.put(v, start);
        Edge edge = graph.getEdge(start.getId(), v.getId());

        if (edge != null)
            dist.put(v, edge.getWeight());
        else
            dist.put(v, Double.POSITIVE_INFINITY);
    }

    //Main loop
    while (!vMinusS.isEmpty())
    {
        //Find the value u in V - S with the smallest dist of u
        double minDist = Double.POSITIVE_INFINITY;
        Vertex u = null;
        for (Vertex v : vMinusS)
        {
            if (dist.get(v) < minDist)
            {
                minDist = dist.get(v);
                u = v;
            }
        }

        // Remove u from vMinusS
        vMinusS.remove(u);

        //Update the distances
        Iterator<Edge> edgeIter = graph.edgeIterator(u.getId());
        while (edgeIter.hasNext())
        {
            Edge edge = edgeIter.next();
            Vertex v = graph.getVertex(edge.getDest());
            if (vMinusS.contains(v))
            {
                // Get the boosting value, if the new distance is smaller then the previous
                // value, change the distance.
                double weight = edge.getWeight();
                String boosting = u.getProperty("boosting");
                double boostValue = (boosting == null ? 0 : Double.parseDouble(boosting));
                if (dist.get(u) + weight - boostValue < dist.get(v))
                {
                    dist.put(v, dist.get(u) + weight - boostValue);
                    pred.put(v, u);
                }
            }
        }
    }
}

```

Result of tests:

Vertex = [(Id, "label" ) : weight]

Edge = [(source, destination) : weight]

```
----- Dijkstras Test -----

--Echo[1] A graph with size 5 created.

--Echo[2] Edges have been added to the graph.

The Graph
[(0 : "") : 1.0] -> [(0,1) : 10.0] -> [(0,3) : 30.0] -> [(0,4) : 100.0]
[(1 : "") : 1.0] -> [(1,2) : 50.0]
[(2 : "") : 1.0] -> [(2,4) : 10.0]
[(3 : "") : 1.0] -> [(3,2) : 20.0] -> [(3,4) : 60.0]
[(4 : "") : 1.0]

--Echo[3] Dijkstras algorithm has been run.

Results before boosting

The Predecessor Map
{[(0 : "") : 1.0]=null, [(1 : "") : 1.0]=[(0 : "") : 1.0], [(2 : "") : 1.0]=[(3 : "") : 1.0],
 [(3 : "") : 1.0]=[(0 : "") : 1.0], [(4 : "") : 1.0]=[(2 : "") : 1.0]}

The Distance Map
{[(0 : "") : 1.0]=Infinity, [(1 : "") : 1.0]=10.0, [(2 : "") : 1.0]=50.0,
 [(3 : "") : 1.0]=30.0, [(4 : "") : 1.0]=60.0}

--Echo[4] Boost the 1st vertex with 20.

Results after boosting

The Predecessor Map
{[(0 : "") : 1.0]=null, [(1 : "") : 1.0]=[(0 : "") : 1.0], [(2 : "") : 1.0]=[(1 : "") : 1.0],
 [(3 : "") : 1.0]=[(0 : "") : 1.0], [(4 : "") : 1.0]=[(2 : "") : 1.0]}

The Distance Map
{[(0 : "") : 1.0]=Infinity, [(1 : "") : 1.0]=10.0, [(2 : "") : 1.0]=40.0,
 [(3 : "") : 1.0]=30.0, [(4 : "") : 1.0]=50.0}
```