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

Ertuğrul Kaşıkçı
200104004097

Detailed System Requirements:

Theese classes written in Java 11 so it can be run in anyplace that java can run. Program starts and tests all the functionalty of the program and shows their results. First tests all the functions in MyGraph class then does breadth first search and depth first search with the graph contstructed in question 1 and finds differences of the distances of them. Lastly tests Dijkstra's Algorithm and shows its result.

MyGraph class is a implementation of the graph data structure by using adjacency list.

Question 1:

MyGraph uses adjacency list to implement graphs. Adjacency list in this class is a two dimensional ArrayList keeps edge class. Indexes of the first dimension of the adjacency list represent the vertices and every ArrayList attached to these vertices holds edges which makes bound with those vertices.

When removing a vertices, all the indexes of the vertices comes after that vertex are changed because of the ArrayList behavior.

Filter vertices method filters the vertices which has the same value for the given key.

Export Matrix method converts adjacency list into adjacency matrix and returns this double dimensional double array.

Algorithm Analysis of the MyGraph Functions:

n -> number of vertices   m -> number of edges

int getNumV() -> Θ(1)

boolean isDirected() -> Θ(1)

void insert(Edge edge)) -> Θ(1) (amortized constant time)

boolean isEdge(int source, int dest) -> O(m)

Edge getEdge(int source, int dest) -> O(m)

Iterator<Edge> edgeIterator(int source) -> Θ(1)

Vertex newVertex (String label, double weight) -> Θ(1)

boolean addVertex (Vertex new_vertex) -> Θ(1) (amortized constant time)

boolean addEdge (int vertexID1, int vertexID2, double weight) -> Θ(1) (amortized constant time)

Edge removeEdge (int vertexID1, int vertexID2) -> O(n)

Vertex removeVertex (int vertexID) -> O(n)

Vertex removeVertex (String label)) -> O(n)

DynamicGraph filterVertices (String key, String filter) -> Θ(n + m)

double[][] exportMatrix() -> Θ(n + m)

public void printGraph() -> Θ(n + m)

Question 2:

BFSTraversel method does breadth first search. Uses a queue structure to hande this. Starts from a vertex. Starts from a vertex, inserts all adjacent vertices into the queue and marks them as identified. Firstly inserts the vertices which has lower weight. Than marks the current vertex as visited, pulls a vertex from the queue than repeats the same process until all of the vertecies are marked as visited.

The complexity of the BFSTraversel function: normally it would be O (n+m) but because of the selection of the shortest edge in every time it increases to O (m*(n+m))

DFSTraversel function does depth first search. Uses helper DFSRecursive method which is a recursive method implementing depth first search. In implementation an array is used to keep track of the visited vertices.

The complexity of the DFSTraversel function: normally it would be Θ (n+m) but because of the selection of the shortest edge in every time it increases to O (m*(n+m))

Question 3:

DijkstrasAlgorithm method takes a graph and a vertex ID as parameters and finds the shortest paths to all vertives from the given vertex. As a distinction from the general concept, boosting values are used in this method. If the vertices which is in a pathway has a boosting value, then these values are subtructed from the distance of that path and when the shortes paths are selected this implementation is considered. The method also finds the predecessor vertices in the shortest paths so all the shortest paths starting from given vertex to all vertices can be found by this method.

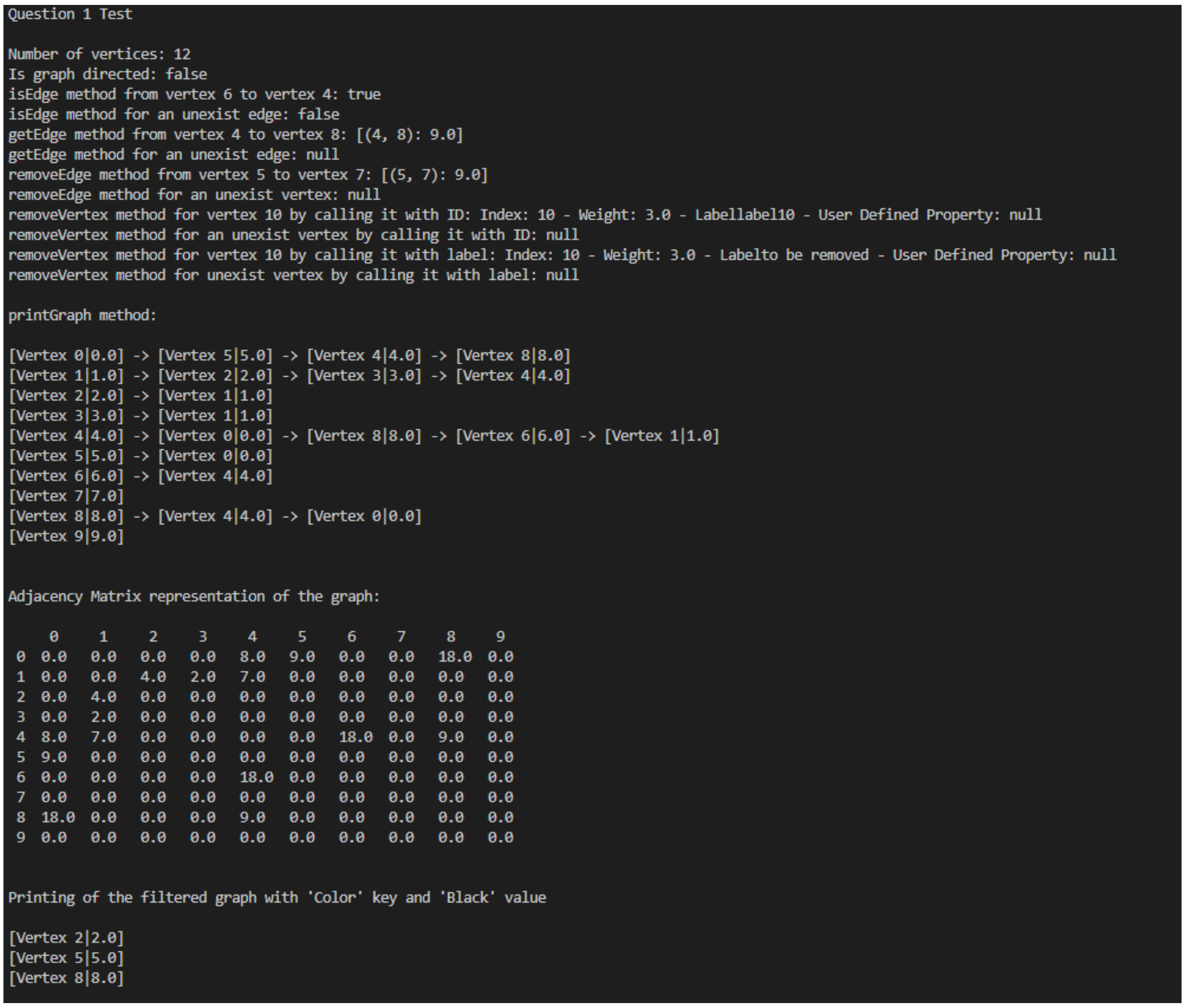
The complexity of the DijkstrasAlgorithm function: Θ (n^2)

Test Cases:

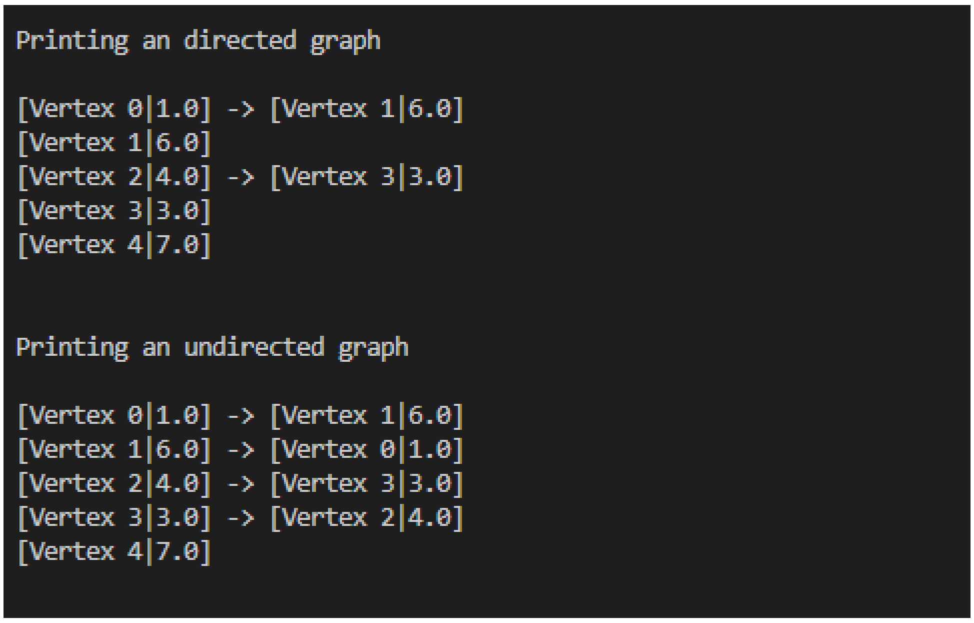
For the first question all of the MyGraph methods are tested. To show their functionality and correctness, valid and invalid conditions were implemented and their result were printed. Then to graph is constructed. The only difference between theese graphs is directed property. Theese graphs are printed to show the difference.

For the second question to implement breadth first search and depth first search, the graph which is constructed in question 1 test is used. Searching methods are implemented and vertices which are passed during traversel are printed in order to show the difference and also the total distance difference is printed to show the difference between the searching methods.

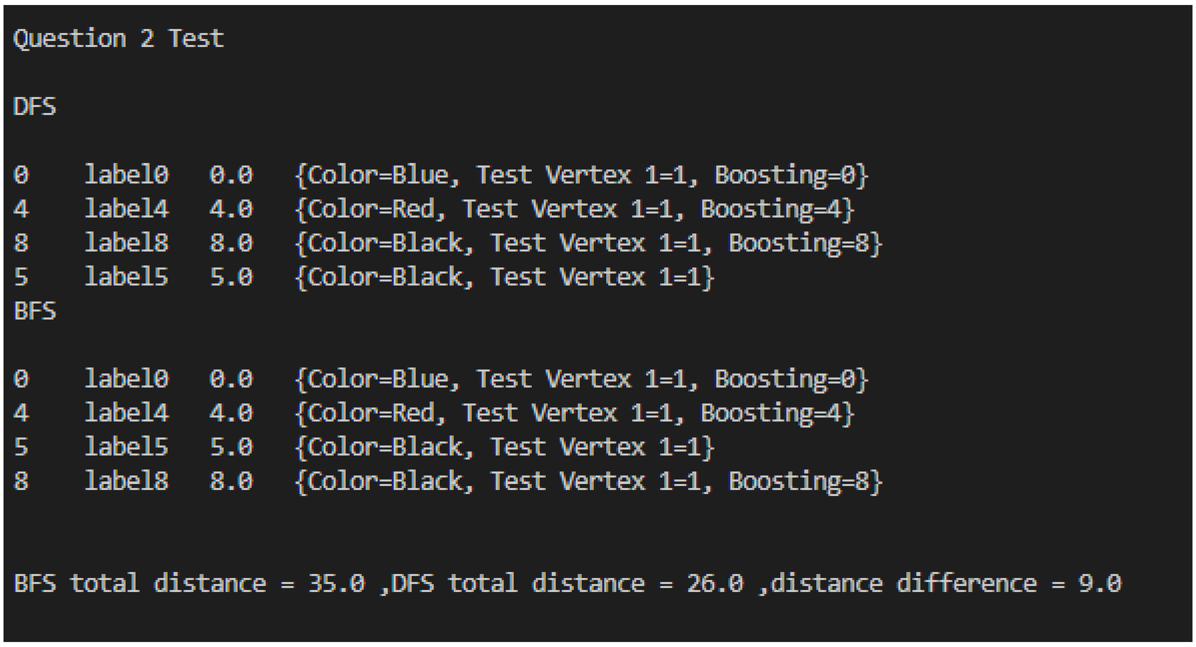
For the third question the same graph and its first vertex is used to show functionality and correctness of the DijkstrasAlgorithm method. Method prints all the shortest distances and predecessors on theese paths. The effect of the boosting value can be seen in the output of the program. The dictance for vertex 8 normally would be 17 but the boosting value in the vertex 4 decreasing it to 13.



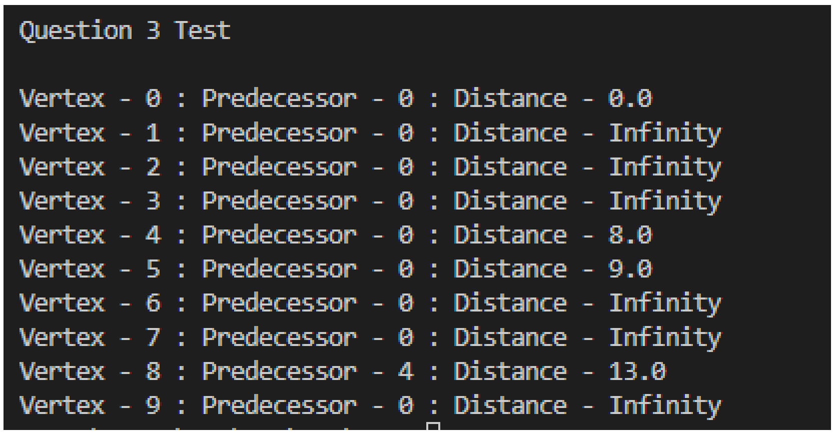
```
Question 1 Test

Number of vertices: 12
Is graph directed: false
isEdge method from vertex 6 to vertex 4: true
isEdge method for an unexist edge: false
getEdge method from vertex 4 to vertex 8: [(4, 8): 9.0]
getEdge method for an unexist edge: null
removeEdge method from vertex 5 to vertex 7: [(5, 7): 9.0]
removeEdge method for an unexist vertex: null
removeVertex method for vertex 10 by calling it with ID: Index: 10 - Weight: 3.0 - Labellabel10 - User Defined Property: null
removeVertex method for an unexist vertex by calling it with ID: null
removeVertex method for vertex 10 by calling it with label: Index: 10 - Weight: 3.0 - Labelto be removed - User Defined Property: null
removeVertex method for unexist vertex by calling it with label: null

printGraph method:

[Vertex 0|0.0] -> [Vertex 5|5.0] -> [Vertex 4|4.0] -> [Vertex 8|8.0]
[Vertex 1|1.0] -> [Vertex 2|2.0] -> [Vertex 3|3.0] -> [Vertex 4|4.0]
[Vertex 2|2.0] -> [Vertex 1|1.0]
[Vertex 3|3.0] -> [Vertex 1|1.0]
[Vertex 4|4.0] -> [Vertex 0|0.0] -> [Vertex 8|8.0] -> [Vertex 6|6.0] -> [Vertex 1|1.0]
[Vertex 5|5.0] -> [Vertex 0|0.0]
[Vertex 6|6.0] -> [Vertex 4|4.0]
[Vertex 7|7.0]
[Vertex 8|8.0] -> [Vertex 4|4.0] -> [Vertex 0|0.0]
[Vertex 9|9.0]


Adjacency Matrix representation of the graph:

    0    1    2    3    4    5    6    7    8    9
 0  0.0  0.0  0.0  0.0  8.0  9.0  0.0  0.0  18.0 0.0
 1  0.0  0.0  4.0  2.0  7.0  0.0  0.0  0.0  0.0  0.0
 2  0.0  4.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 3  0.0  2.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 4  8.0  7.0  0.0  0.0  0.0  0.0  18.0 0.0  9.0  0.0
 5  9.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 6  0.0  0.0  0.0  0.0  18.0 0.0  0.0  0.0  0.0  0.0
 7  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0
 8  18.0 0.0  0.0  0.0  9.0  0.0  0.0  0.0  0.0  0.0
 9  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0  0.0


Printing of the filtered graph with 'Color' key and 'Black' value

[Vertex 2|2.0]
[Vertex 5|5.0]
[Vertex 8|8.0]
```



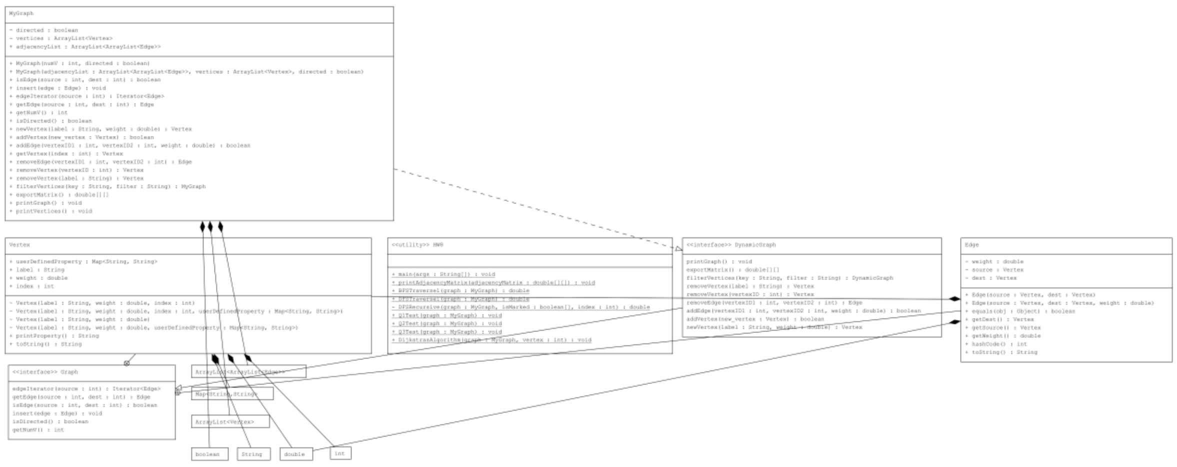
```
Printing an directed graph

[Vertex 0|1.0] -> [Vertex 1|6.0]
[Vertex 1|6.0]
[Vertex 2|4.0] -> [Vertex 3|3.0]
[Vertex 3|3.0]
[Vertex 4|7.0]


Printing an undirected graph

[Vertex 0|1.0] -> [Vertex 1|6.0]
[Vertex 1|6.0] -> [Vertex 0|1.0]
[Vertex 2|4.0] -> [Vertex 3|3.0]
[Vertex 3|3.0] -> [Vertex 2|4.0]
[Vertex 4|7.0]
```



```
Question 2 Test

DFS

0    label0   0.0   {Color=Blue, Test Vertex 1=1, Boosting=0}
4    label4   4.0   {Color=Red, Test Vertex 1=1, Boosting=4}
8    label8   8.0   {Color=Black, Test Vertex 1=1, Boosting=8}
5    label5   5.0   {Color=Black, Test Vertex 1=1}
BFS

0    label0   0.0   {Color=Blue, Test Vertex 1=1, Boosting=0}
4    label4   4.0   {Color=Red, Test Vertex 1=1, Boosting=4}
5    label5   5.0   {Color=Black, Test Vertex 1=1}
8    label8   8.0   {Color=Black, Test Vertex 1=1, Boosting=8}


BFS total distance = 35.0 ,DFS total distance = 26.0 ,distance difference = 9.0
```



```
Question 3 Test

Vertex - 0 : Predecessor - 0 : Distance - 0.0
Vertex - 1 : Predecessor - 0 : Distance - Infinity
Vertex - 2 : Predecessor - 0 : Distance - Infinity
Vertex - 3 : Predecessor - 0 : Distance - Infinity
Vertex - 4 : Predecessor - 0 : Distance - 8.0
Vertex - 5 : Predecessor - 0 : Distance - 9.0
Vertex - 6 : Predecessor - 0 : Distance - Infinity
Vertex - 7 : Predecessor - 0 : Distance - Infinity
Vertex - 8 : Predecessor - 4 : Distance - 13.0
Vertex - 9 : Predecessor - 0 : Distance - Infinity
```



MyGraph
- directed : boolean
- vertices : ArrayList<Vertex>
+ adjacencyList : ArrayList<ArrayList<Edge>>
+ MyGraph(numV : int, directed : boolean)
+ MyGraph(adjacencyList : ArrayList<ArrayList<Edge>>, vertices : ArrayList<Vertex>, directed : boolean)
+ isEdge(source : int, dest : int) : boolean
+ insert(edge : Edge) : void
+ edgeIterator(source : int) : Iterator<Edge>
+ getEdge(source : int, dest : int) : Edge
+ getNumV() : int
+ isDirected() : boolean
+ newVertex(label : String, weight : double) : Vertex
+ addVertex(new_vertex : Vertex) : boolean
+ addEdge(vertexID1 : int, vertexID2 : int, weight : double) : boolean
+ getVertex(index : int) : Vertex
+ removeEdge(vertexID1 : int, vertexID2 : int) : Edge
+ removeVertex(vertexID : int) : Vertex
+ removeVertex(label : String) : Vertex
+ filterVertices(key : String, filter : String) : MyGraph
+ exportMatrix() : double[][]
+ printGraph() : void
+ printVertices() : void
Vertex
+ userDefinedProperty : Map<String, String>
+ label : String
+ weight : double
+ index : int
~ Vertex(label : String, weight : double, index : int)
- Vertex(label : String, weight : double, index : int, userDefinedProperty : Map<String, String>)
~ Vertex(label : String, weight : double)
- Vertex(label : String, weight : double, userDefinedProperty : Map<String, String>)
+ printProperty() : String
+ toString() : String
<<utility>> HW8
+ main(args : String[]) : void
+ printAdjacencyMatrix(adjacencyMatrix : double[][]) : void
+ BFSTraversel(graph : MyGraph) : double
+ DFSTraversel(graph : MyGraph) : double
- DFSRecursive(graph : MyGraph, isMarked : boolean[], index : int) : double
+ Q1Test(graph : MyGraph) : void
+ Q2Test(graph : MyGraph) : void
+ Q3Test(graph : MyGraph) : void
+ DijkstrasAlgorithm(graph : MyGraph, vertex : int) : void
<<interface>> DynamicGraph
printGraph() : void
exportMatrix() : double[][]
filterVertices(key : String, filter : String) : DynamicGraph
removeVertex(label : String) : Vertex
removeVertex(vertexID : int) : Vertex
removeEdge(vertexID1 : int, vertexID2 : int) : Edge
addEdge(vertexID1 : int, vertexID2 : int, weight : double) : boolean
addVertex(new_vertex : Vertex) : boolean
newVertex(label : String, weight : double) : Vertex
Edge
- weight : double
- source : Vertex
- dest : Vertex
+ Edge(source : Vertex, dest : Vertex)
+ Edge(source : Vertex, dest : Vertex, weight : double)
+ equals(obj : Object) : boolean
+ getDest() : Vertex
+ getSource() : Vertex
+ getWeight() : double
+ hashCode() : int
+ toString() : String
<<interface>> Graph
edgeIterator(source : int) : Iterator<Edge>
getEdge(source : int, dest : int) : Edge
isEdge(source : int, dest : int) : boolean
insert(edge : Edge) : void
isDirected() : boolean
getNumV() : int
ArrayList<ArrayList<Edge>>
Map<String,String>
ArrayList<Vertex>
boolean
String
double
int