GIT

DEPARTMENT OF COMPUTER ENGINEERING

CSE 222/505 – Spring 2021

REPORT FOR HOMEWORK 8

SÜLEYMAN GÖLBOL

1801042656

1. SYSTEM REQUIREMENTS

FUNCTIONAL REQUIREMENTS

This application can be run by makefile file. In a bash or make entegreted PowerShell/cmd screen, “make” command will run the all tests.

Another functional requirement is having Java in computer. At least should *have Java 7 because* Java Comparable methods are used.

NON-FUNCTIONAL REQUIREMENTS

Because of the application is going to be used by a tester, a test method was prepared.

The program must be able to access all the graph classes’ methods. So importing right packages is important.

The program should be portable because it just making test.

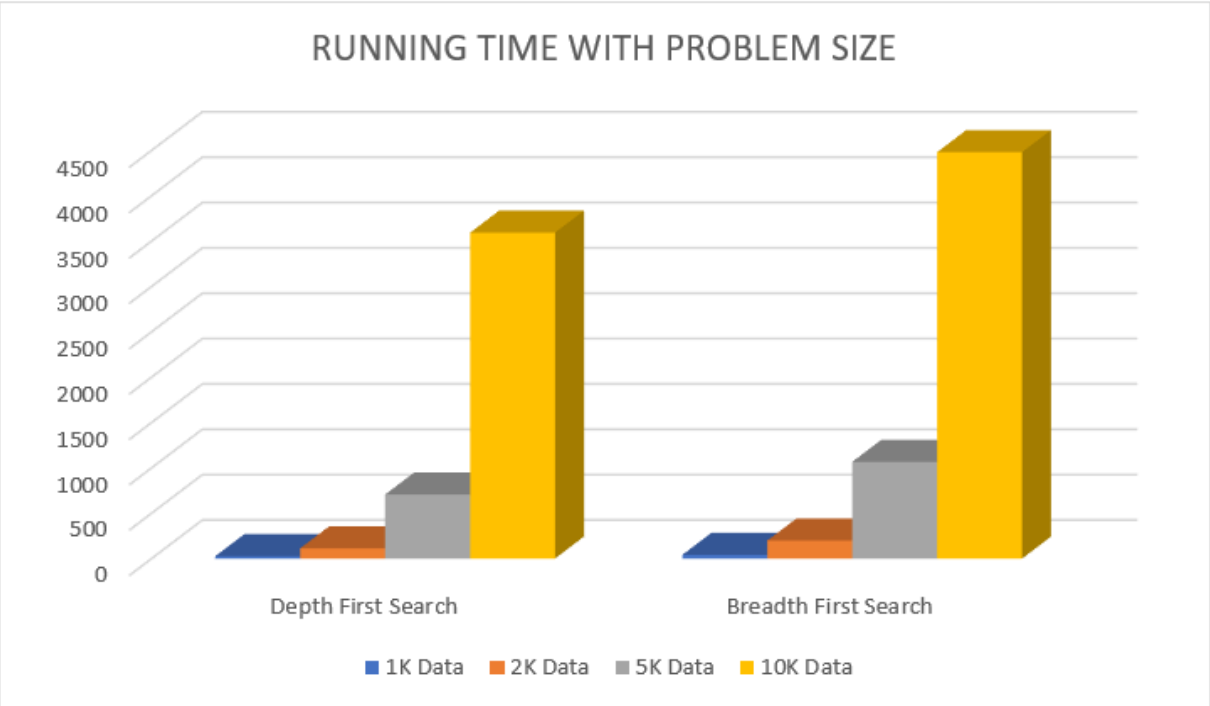
The program uses ram to store values in the memory.

So the computer must have a little bit ram to hold the data.

The program doesn’t create temporary text files to store values inside of text.

Also because of part2 method uses breadth first search and depth first search for big input values(like 10k) and for 10 times, running program can be slow(like close to 1 minute)ç

2. PART2 RESULT IN A GRAPHIC



For adding 1k,2k,5k and 10k data; as you can see breadth first search takes more time when data gets bigger. But when data is small the difference is hard to understand.

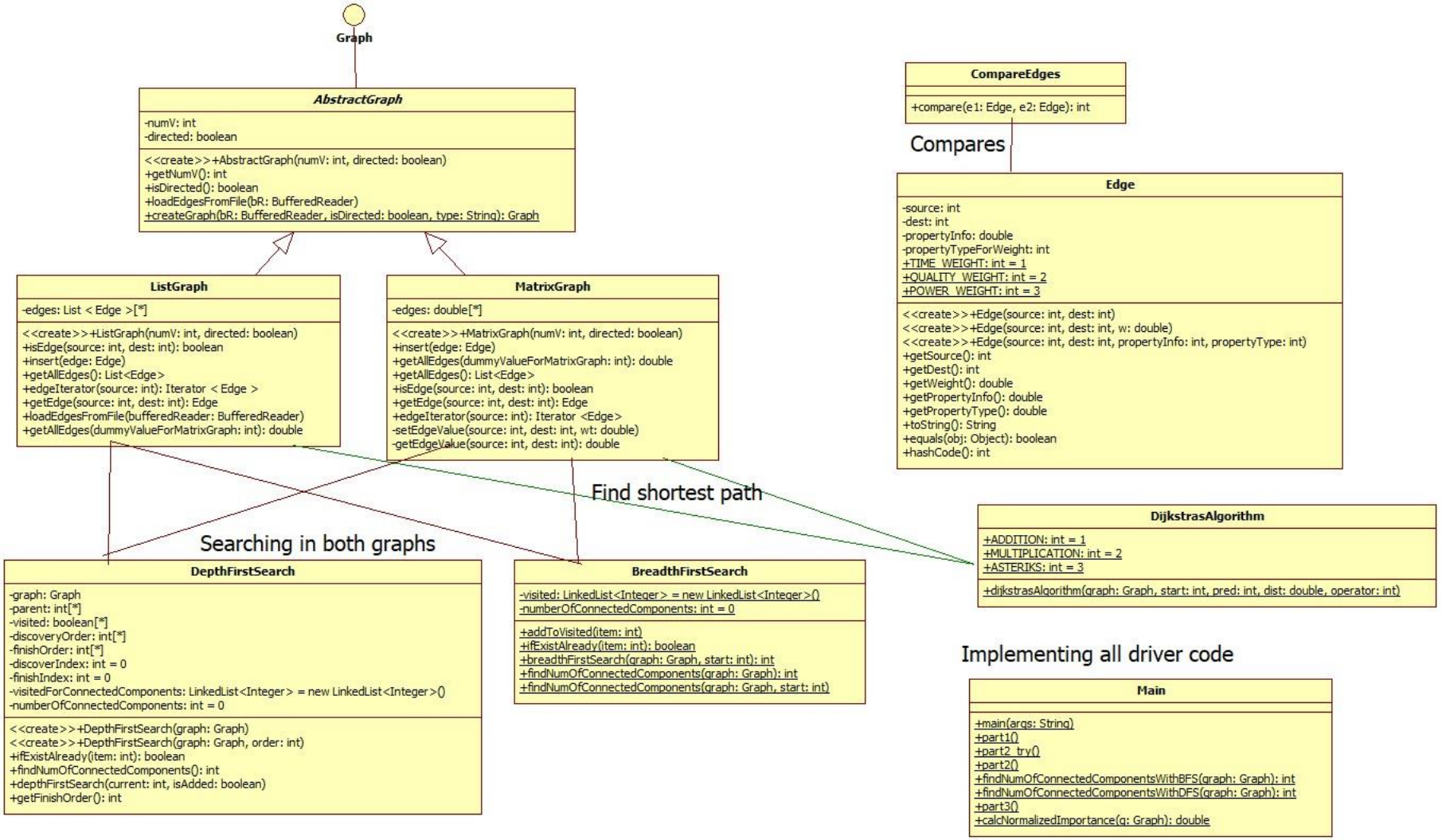
Also there is chance to it too, because depth first search is a little bit brave search, it might take long if there is more connected to it.

The table for datas:

	1K Data	2K Data	5K Data	10K Data
Depth First Search	26,6	113,5	706,3	3592,5
Breadth First Search	41	198,2	1066,1	4480,3

.

3. CLASS DIAGRAMS



4. PROBLEM SOLUTION APPROACH

My solution approach for Part1 was to create static final values which are constant values. I used them to check ADDITION, MULTIPLICATION and ASTERIKS(*). I set some integer datas to it to check if they are true or not.

For part2, I created some methods and data fields in some classes. For example in BreadthFirstSearch class, I created a LinkedList called visited. Also I created a variable to count number of connected components. Also I changed breadthFirstSearch method and I called it findNumOfConnectedComponents() and in this method I checked for every value if this value is seen before (if visited). If value was visited I didn't increment number of connected components, because this value means the component it belongs is already seen so when I found a value that if doesn't seen already I added it to visited linkedlist. Then I incremented the number of connected components. For depth first search, I hold a visitedForConnectedComponents LinkedList for visited values. Then I hold number of connected components as a counter. Because of this method was recursive, I didn't add a data field to it, I added a parameter called isAdded so I checked this current value if was seen before or not, if it seen I understand that we are in same connected component so I didn't increment the counter. But if it wasn't seen in hole array I incremented the number of connected components.

For part3 I checked every vertex(u vertex) And for every vertex found shortest path with Breadth first search. Then put these shortest path values to predecessor.
Then from first value of predecessor(u vertex) until last value of predecessor(w vertex) I made sure these are not same and then find middle vertex(v vertex) that different from u and w and increase numerator and denumerator values, and if couldn't find just increase denumerator value. Then add nominator/denominator value to summation and in for loop did this for all vertices gives as all summation value.

5. TEST CASES

5.1) Creating edges and catching exception.

```
System.out.println( "\n---CREATING EDGES---" );
Edge edge  = new Edge(0, 1, 10, Edge.TIME_WEIGHT);
Edge edge2 = new Edge(0, 4, 100, Edge.TIME_WEIGHT);
Edge edge3 = new Edge(0, 3, 30, Edge.TIME_WEIGHT);
Edge edge4 = new Edge(2, 4, 10, Edge.TIME_WEIGHT);
Edge edge5 = new Edge(3, 2, 20, Edge.TIME_WEIGHT);
Edge edge6 = new Edge(3, 4, 60, Edge.TIME_WEIGHT);
Edge edge7 = new Edge(1, 2, 50, Edge.TIME_WEIGHT);
System.out.println("---Catching exception that throwed when creating edge---"); int NEW_WEIGHT = 4;
try{
    Edge edge8 = new Edge(1, 3, 56, NEW_WEIGHT);
}catch(IllegalStateException exception){
    System.out.println("Exception catched.");
}
```

5.2) Creating list graph and inserting elements.

```
System.out.println( "\n---LIST GRAPH---" );
System.out.println( "---Creating list graph---" );
ListGraph listGraph = new ListGraph(5, true);

System.out.println( "---Inserting edges to list graph---" );
listGraph.insert(edge);
listGraph.insert(edge2);
listGraph.insert(edge3);
listGraph.insert(edge4);
listGraph.insert(edge5);
listGraph.insert(edge6);
listGraph.insert(edge7);
```

5.3) Creating pred and dist arrays to hold values of Dijkstras Algorithm.

```
System.out.println( "\n---Creating pred and dist arrays.---" );
int pred[] = new int[5];
double dist[] = new double[5];
for(int i=0; i<pred.length; i++) pred[i] = 0;
for(int i=0; i<dist.length; i++) dist[i] = listGraph.getEdge(0, i).getWeight();
```

5.4) Testing Dijkstras Algorithm for addidition, for multiplication and for asterisk (*) symbol.

```
System.out.println( "---LIST GRAPH Dijkstra's Algorithm TESTING---" );
System.out.println( "---Using Dijkstra's Algorithm for Addition---" );
DijkstrasAlgorithm.dijkstrasAlgorithm(listGraph, 0, pred, dist, DijkstrasAlgorithm.ADDITION);
System.out.println( "---Using Dijkstra's Algorithm for Multiplication---" );
DijkstrasAlgorithm.dijkstrasAlgorithm(listGraph, 0, pred, dist, DijkstrasAlgorithm.MULTIPLICATION);
System.out.println( "---Using Dijkstra's Algorithm for Asteriks(*)---" );
DijkstrasAlgorithm.dijkstrasAlgorithm(listGraph, 0, pred, dist, DijkstrasAlgorithm.ASTERIKS);
```

5.5) Creating matrix graph and inserting elements.

```
System.out.println( "\n---MATRIX GRAPH---" );
System.out.println( "---Creating matrix graph---" );
MatrixGraph matrixGraph = new MatrixGraph(5, false);

System.out.println("---Inserting edges to matrix graph---");
matrixGraph.insert(edge);
matrixGraph.insert(edge2);
matrixGraph.insert(edge3);
matrixGraph.insert(edge4);
matrixGraph.insert(edge5);
matrixGraph.insert(edge6);
matrixGraph.insert(edge7);
```

5.6) Creating pred and dist arrays to hold values of Dijkstras Algorithm predecessors and distance values.

```
System.out.println( "---Creating pred and dist arrays.---" );
int pred2[] = new int[5];
double dist2[] = new double[5];
for(int i=0; i<pred2.length; i++) pred2[i] = 0;
for(int i=0; i<dist2.length; i++) dist2[i] = matrixGraph.getEdge(0, i).getWeight();
```

5.7) Testing Dijkstras Algorithm for addidition, for multiplication and for asterisk (*) symbol.

```
System.out.println( "---MATRIX GRAPH Dijkstra's Algorithm TESTING---" );
System.out.println( "---Using Dijkstra's Algorithm for Addition---" );
DijkstrasAlgorithm.dijkstrasAlgorithm(matrixGraph, 0, pred2, dist2, DijkstrasAlgorithm.ADDITION);
System.out.println( "---Using Dijkstra's Algorithm for Multiplication---" );
DijkstrasAlgorithm.dijkstrasAlgorithm(matrixGraph, 0, pred2, dist2, DijkstrasAlgorithm.MULTIPLICATION);
System.out.println( "---Using Dijkstra's Algorithm for Asteriks(*)---" );
DijkstrasAlgorithm.dijkstrasAlgorithm(matrixGraph, 0, pred2, dist2, DijkstrasAlgorithm.ASTERIKS);
```

5.8) Part2: Creating average values to print at last.

```java
/** Part2 driver code */
public static void part2(){
    Random random = new Random();
    long nano_start = System.currentTimeMillis();
    long nano_end = System.currentTimeMillis();

    //Create average values for all. It will be printed at last.
    float runningTime_1000_avg_dfs  = 0;    float runningTime_1000_avg_bfs  = 0;
    float runningTime_2000_avg_dfs  = 0;    float runningTime_2000_avg_bfs  = 0;
    float runningTime_5000_avg_dfs  = 0;    float runningTime_5000_avg_bfs  = 0;
    float runningTime_10000_avg_dfs = 0;    float runningTime_10000_avg_bfs = 0;
```

5.9) Repeating for 20k, 40k and 80k data.
Creating for 1000 vertex list graph and matrix graph and adding RANDOM edges.

```java
for(int count=0; count<10; count++){
    System.out.println("\n" + (count+1) + ". Time");

    int MAX_VERTICES = 1000;
    ListGraph graph1_list = new ListGraph(MAX_VERTICES, false);
    MatrixGraph graph1_matrix = new MatrixGraph(MAX_VERTICES, false);

    int size = random.nextInt( MAX_VERTICES );
    for(int i=3; i < size; i++){
        int u = random.nextInt(i+1);
        int v = i-3;
        graph1_list.insert( new Edge(u, v, i+5, Edge.QUALITY_WEIGHT) );
        graph1_matrix.insert( new Edge(u, v, i+5, Edge.QUALITY_WEIGHT) );
    }
```

5.10) Calculating time values when finding number of connected components for list graph and matrix graph. Then adding thse bfs(breadth first) and dfs(depth first) values to average value to print at last.

```java
nano_start = System.currentTimeMillis();
System.out.println("For 1000 elements + List   Graph + DFS | Num of connect components: " + findNumOfConnectedComponentsWithDFS(graph1_list));
System.out.println("For 1000 elements + Matrix Graph + DFS | Num of connect components: " + findNumOfConnectedComponentsWithDFS(graph1_matrix));
nano_end = System.currentTimeMillis();
runningTime_1000_avg_dfs += (nano_end-nano_start);

nano_start = System.currentTimeMillis();
System.out.println("For 1000 elements + List   Graph + BFS | Num of connect components: " + findNumOfConnectedComponentsWithBFS(graph1_list));
System.out.println("For 1000 elements + Matrix Graph + BFS | Num of connect components: " + findNumOfConnectedComponentsWithBFS(graph1_matrix));
nano_end = System.currentTimeMillis();
runningTime_1000_avg_bfs += (nano_end-nano_start);
```

5.11) Creating for 2000 vertex list graph and matrix graph and adding RANDOM elements. Then, Calculating time values when finding number of connected components for list graph and matrix graph. Then adding these bfs(breadth first) and dfs(depth first) values to average value to print at last.

```java
MAX_VERTICES = 2000;
ListGraph graph2_list = new ListGraph(MAX_VERTICES, false);
MatrixGraph graph2_matrix = new MatrixGraph(MAX_VERTICES, false);

size = random.nextInt( MAX_VERTICES );
for(int i=1; i < size; i++){
    int u = random.nextInt(i+1);
    int v = i+3;
    graph2_list.insert( new Edge(u, v, i+5, Edge.TIME_WEIGHT) );
    graph2_matrix.insert( new Edge(u, v, i+5, Edge.TIME_WEIGHT) );
}
nano_start = System.currentTimeMillis();
System.out.println("For 2000 elements + List   Graph + DFS | Num of connect components: " + findNumOfConnectedComponentsWithDFS(graph2_list));
System.out.println("For 2000 elements + Matrix Graph + DFS | Num of connect components: " + findNumOfConnectedComponentsWithDFS(graph2_matrix));
nano_end = System.currentTimeMillis();
runningTime_2000_avg_dfs += (nano_end-nano_start);

nano_start = System.currentTimeMillis();
System.out.println("For 2000 elements + List   Graph + BFS | Num of connect components: " + findNumOfConnectedComponentsWithBFS(graph2_list));
System.out.println("For 2000 elements + Matrix Graph + BFS | Num of connect components: " + findNumOfConnectedComponentsWithBFS(graph2_matrix) +
nano_end = System.currentTimeMillis();
runningTime_2000_avg_bfs += (nano_end-nano_start);
```

5.12) Creating for 5000 vertex list graph and matrix graph and adding RANDOM elements. Then, Calculating time values when finding number of connected components for list graph and matrix graph. Then adding these bfs(breadth first) and dfs(depth first) values to average value to print at last.

```
MAX_VERTICES = 5000;
ListGraph graph3_list = new ListGraph(MAX_VERTICES, false);
MatrixGraph graph3_matrix = new MatrixGraph(MAX_VERTICES, false);

size = random.nextInt( MAX_VERTICES );
for(int i=0; i < size; i++){
    int u = random.nextInt(i+1);
    int v = i + 3;
    graph3_list.insert( new Edge(u, v, i+5, Edge.QUALITY_WEIGHT) );
    graph3_matrix.insert( new Edge(u, v, i+5, Edge.QUALITY_WEIGHT) );
}
nano_start = System.currentTimeMillis();
System.out.println("For 5000 elements + List   Graph + DFS | Num of connect components: " + findNumOfConnectedComponentsWithDFS(graph3_list));
System.out.println("For 5000 elements + Matrix Graph + DFS | Num of connect components: " + findNumOfConnectedComponentsWithDFS(graph3_matrix));
nano_end = System.currentTimeMillis();
runningTime_5000_avg_dfs += (nano_end-nano_start);

nano_start = System.currentTimeMillis();
System.out.println("For 5000 elements + List   Graph + BFS | Num of connect components: " + findNumOfConnectedComponentsWithBFS(graph3_list));
System.out.println("For 5000 elements + Matrix Graph + BFS | Num of connect components: " + findNumOfConnectedComponentsWithBFS(graph3_matrix) + "
nano_end = System.currentTimeMillis();
runningTime_5000_avg_bfs += (nano_end-nano_start);
```

5.13) Creating for 10000 vertex list graph and matrix graph and adding RANDOM elements. Then, Calculating time values when finding number of connected components for list graph and matrix graph. Then adding these bfs(breadth first) and dfs(depth first) values to average value to print at last.

```
nano_start = System.currentTimeMillis();
size = random.nextInt( MAX_VERTICES );
for(int i=0; i < size; i++){
    int u = random.nextInt(i+1);
    int v = i+1;
    graph4_list.insert( new Edge(u, v, i+5, Edge.TIME_WEIGHT) );
    graph4_matrix.insert( new Edge(u, v, i+5, Edge.TIME_WEIGHT) );
}
nano_start = System.currentTimeMillis();
System.out.println("For 10000 elements + List   Graph + DFS | Num of connect components: " + findNumOfConnectedComponentsWithDFS(graph4_list));
System.out.println("For 10000 elements + Matrix Graph + DFS | Num of connect components: " + findNumOfConnectedComponentsWithDFS(graph4_matrix));
nano_end = System.currentTimeMillis();
runningTime_10000_avg_dfs += (nano_end-nano_start);

nano_start = System.currentTimeMillis();
System.out.println("For 10000 elements + List   Graph + BFS | Num of connect components: " + findNumOfConnectedComponentsWithBFS(graph4_list));
System.out.println("For 10000 elements + Matrix Graph + BFS | Num of connect components: " + findNumOfConnectedComponentsWithBFS(graph4_matrix) +
nano_end = System.currentTimeMillis();
runningTime_10000_avg_bfs += (nano_end-nano_start);
```

5.13) Getting average values of all results.

```
//Getting average values of all results.
runningTime_1000_avg_dfs  /= 10.0;      runningTime_1000_avg_bfs  /= 10.0;
runningTime_2000_avg_dfs  /= 10.0;      runningTime_2000_avg_bfs  /= 10.0;
runningTime_5000_avg_dfs  /= 10.0;      runningTime_5000_avg_bfs  /= 10.0;
runningTime_10000_avg_dfs /= 10.0;      runningTime_10000_avg_bfs /= 10.0;
```

5.14) Printing results for 1000, 2000, 5000 and 1000 elements (Both breadth first search and depth first search)

```
System.out.println("Running time average for 1000 elements  DFS " + runningTime_1000_avg_dfs);
System.out.println("Running time average for 1000 elements  BFS " + runningTime_1000_avg_bfs);
System.out.println("Running time average for 2000 elements  DFS " + runningTime_2000_avg_dfs);
System.out.println("Running time average for 2000 elements  BFS " + runningTime_2000_avg_bfs);
System.out.println("Running time average for 5000 elements  DFS " + runningTime_5000_avg_dfs);
System.out.println("Running time average for 5000 elements  BFS " + runningTime_5000_avg_bfs);
System.out.println("Running time average for 10000 elements DFS " + runningTime_10000_avg_dfs);
System.out.println("Running time average for 10000 elements BFS " + runningTime_10000_avg_bfs);
```

5.15) This method is used to find # of components in graph with using breadth first search

```
public static int findNumOfConnectedComponentsWithBFS(Graph graph){
    return BreadthFirstSearch.findNumOfConnectedComponents(graph);
}
```

5.16) This method is used to find # of components in graph with using depth first search

```
public static int findNumOfConnectedComponentsWithDFS(Graph graph){
    DepthFirstSearch graphDfs = new DepthFirstSearch(graph);
    return graphDfs.findNumOfConnectedComponents();
}
```

5.17) Part3: Creating new list graph and creating edges. Then inserting edges to graph. Then calling calculating normalized importance method.

```
/** Driver code for Part3 */
public static void part3(){
    // Creating a graph to calculata normalized importance
    System.out.println("Testing part3 for list graph: ");
    Graph g = new ListGraph(5, false);
    Edge edge1 = new Edge(0, 4); g.insert( edge1 );
    Edge edge2 = new Edge(1, 3); g.insert( edge2 );
    Edge edge3 = new Edge(2, 4); g.insert( edge3 );
    Edge edge4 = new Edge(3, 2); g.insert( edge4 );
    Edge edge5 = new Edge(3, 4); g.insert( edge5 );
    System.out.println( "Normalized importance is : " + calcNormalizedImportance(g) );
```

5.18) Creating new matrix graph Then inserting edges to graph. Then calling calculating normalized importance method.

```
    System.out.println("Testing part3 for matrix graph: ");
    Graph g2 = new MatrixGraph(5, false);
    g2.insert( edge1 );
    g2.insert( edge2 );
    g2.insert( edge3 );
    g2.insert( edge4 );
    g2.insert( edge5 );
    System.out.println( "Normalized importance is : " + calcNormalizedImportance(g2) );
}
```

5.19) This method is used to find # of components in graph with using depth first search Firstly creating nominator denominator and summation values.

```
public static double calcNormalizedImportance(Graph g){
    int summatation_nominator = 0;
    int summatation_denominator = 0;
    double summatation = 0.0;
```

5.19) In for loop, checking every vertex.(u vertex) And for every vertex finding shortest path with Breadth first search. Then putting these shortest path values to predecessor. Then from first value of predecessor(u vertex) until last value of predecessor(w vertex) making sure these are not same and then find middle vertex(v vertex) that different from u and w and increase numerator and denumerator values, and if couldn't find just

increase denumerator value. Then add nominator/denominator value to summation and In for loop doing this for all vertices gives as all summation value.

```
for(int i=0; i<size; i++){ //Go all vertices
    summatation_nominator = 0;
    summatation_denominator = 0;
        int pred[] = new int[size]; //Create predecesoor to hold path.
        for(int m=0; m<pred.length; m++)
            pred[m] = 0;
        try{
            pred = BreadthFirstSearch.breadthFirstSearch(g, i); //Find shortest path with breadth first(no need dijkstra algorithm because unweighte
        }catch( IndexOutOfBoundsException e){ e.printStackTrace(); }
        //Make sure that u and w is not equal.
        if(pred[0] != pred[size-1]){
            // System.out.println( Arrays.toString(pred) );
            for(int j=1; j<size-1; j++){ //Finding v value or values between u and w.
                if(pred[j] != pred[0] && pred[j] != pred[size-1]){ //If found and number is different than u and w add nom and denom.
                    summatation_denominator++;
                    summatation_nominator++;
                }else  //If not found just increase denominator.
                    summatation_denominator++;
            }
        }
    //Add to summation value.
    summatation += (summatation_nominator*1.0)/summatation_denominator;
    // System.out.println(summatation);
}
```

5.19) To find fair importance divide summation to square of number of vertices and return this value.

```
//Find fair importance value by dividing the vertice's square value.
double fairImportance = summatation / (g.getNumV()*g.getNumV()) ;
// System.out.println(fairImportance);
return fairImportance;
```

6) RUNNING COMMAND AND RESULTS

Running make command to run the program.

```
sglbl@SglblPC:/mnt/c/Araclar/GTU/2.Sınıf/2.Dönem/Cse222/hw8$ make
javac -d classfiles *.java
```

```
java -cp classfiles Main
```

Creating edges and catching exceptions

```
---CREATING EDGES---
---Catching exception that throwed when creating edge---
Exception catched.
```

Creating list graph and adding edges to list graph.

```
---LIST GRAPH---
---Creating list graph---
---Inserting edges to list graph---
```

Creating predecessor and distance array and printing predecassor to find shortest path(For addition, multiplication and asteriks*)

```
---Creating pred and dist arrays.---
---LIST GRAPH Dijkstra's Algorithm TESTING---
---Using Dijkstra's Algorithm for Addition---
[0, 0, 3, 0, 2]
---Using Dijkstra's Algorithm for Multiplication---
[0, 0, 1, 0, 0]
---Using Dijkstra's Algorithm for Asteriks(*)---
[0, 0, 1, 0, 3]
```

Creating matrix graph and inserting edges to matrix graph.

```
---MATRIX GRAPH---
---Creating matrix graph---
---Inserting edges to matrix graph---
```

Creating predecessor and distance array and printing predecassor to find shortest path(For addition, multiplication and asteriks*)

```
---MATRIX GRAPH Dijkstra's Algorithm TESTING---
---Using Dijkstra's Algorithm for Addition---
[0, 0, 3, 0, 2]
---Using Dijkstra's Algorithm for Multiplication--
[0, 0, 1, 0, 0]
---Using Dijkstra's Algorithm for Asteriks(*)---
[0, 0, 1, 0, 3]
```

Testing part2 methods

For the first time(it will continue 10 times);

For 1000 elements create both list graph and matrix graph and find number of connected components (both depth first and breadth first search.)

```
1. Time
For 1000 elements + List   Graph + DFS | Num of connect components: 842
For 1000 elements + Matrix Graph + DFS | Num of connect components: 842
For 1000 elements + List   Graph + BFS | Num of connect components: 842
For 1000 elements + Matrix Graph + BFS | Num of connect components: 842
```

For 2000 elements create both list graph and matrix graph and find number of connected components (both depth first and breadth first search.)

```
For 2000 elements + List   Graph + DFS | Num of connect components: 725
For 2000 elements + Matrix Graph + DFS | Num of connect components: 725
For 2000 elements + List   Graph + BFS | Num of connect components: 725
For 2000 elements + Matrix Graph + BFS | Num of connect components: 725
```

For 5000 elements create both list graph and matrix graph and find number of connected components (both depth first and breadth first search.)

```
For 5000 elements + List   Graph + DFS | Num of connect components: 3030
For 5000 elements + Matrix Graph + DFS | Num of connect components: 3030
For 5000 elements + List   Graph + BFS | Num of connect components: 3030
For 5000 elements + Matrix Graph + BFS | Num of connect components: 3030
```

For 10000 elements create both list graph and matrix graph and find number of connected components (both depth first and breadth first search.)

```
For 10000 elements + List   Graph + DFS | Num of connect components: 5305
For 10000 elements + Matrix Graph + DFS | Num of connect components: 5305
For 10000 elements + List   Graph + BFS | Num of connect components: 5305
For 10000 elements + Matrix Graph + BFS | Num of connect components: 5305
```

…..

REPEATING THIS FOR 10 TIMES

…..

```
10. Time
For 1000 elements + List   Graph + DFS | Num of connect components: 694
For 1000 elements + Matrix Graph + DFS | Num of connect components: 694
For 1000 elements + List   Graph + BFS | Num of connect components: 694
For 1000 elements + Matrix Graph + BFS | Num of connect components: 694

For 2000 elements + List   Graph + DFS | Num of connect components: 1141
For 2000 elements + Matrix Graph + DFS | Num of connect components: 1141
For 2000 elements + List   Graph + BFS | Num of connect components: 1141
For 2000 elements + Matrix Graph + BFS | Num of connect components: 1141

For 5000 elements + List   Graph + DFS | Num of connect components: 2398
For 5000 elements + Matrix Graph + DFS | Num of connect components: 2398
For 5000 elements + List   Graph + BFS | Num of connect components: 2398
For 5000 elements + Matrix Graph + BFS | Num of connect components: 2398

For 10000 elements + List   Graph + DFS | Num of connect components: 2857
For 10000 elements + Matrix Graph + DFS | Num of connect components: 2857
For 10000 elements + List   Graph + BFS | Num of connect components: 2857
For 10000 elements + Matrix Graph + BFS | Num of connect components: 2857
```

Printing part2 time values for 1k, 2k,5k and 10K data. (Both depth first and breadth first search.)

```
Running time average for 1000 elements  DFS 26.6
Running time average for 1000 elements  BFS 41.0
Running time average for 2000 elements  DFS 113.5
Running time average for 2000 elements  BFS 198.2
Running time average for 5000 elements  DFS 706.3
Running time average for 5000 elements  BFS 1066.1
Running time average for 10000 elements DFS 3592.5
Running time average for 10000 elements BFS 4480.3
```

Testing part3 (normalized importance) for both list graph and matrix graph.