PROJECT REPORT

CSCE 629-602

ANALYSIS OF ALGORITHMS

BY:

Neeraj Aduri

UIN: 128000940

Introduction:

Network Optimization is one of the very important areas in the current research in computer science and computer engineering. This project is a network routing application which I have implemented in Java using a randomly generated Graph. This report I analyses the performance of these algorithms when run on different types of networks viz., Sparse and Dense generated with the given conditions as given.

The algorithms analyzed below are based on Dijkstra's algorithm and Kruskal's algorithm which are implemented using Lists and Max Heaps.

Implementation:

(All the programs were written in Java and appropriate classes are created)

Along with the randomly generated graphs, a MaxHeap is also developed as given in Problem 3. I had to design 2 algorithms one for Dijkstras and one for Krusklas vize., MaxHeap.java and MaxHeapKruskals.java. These heaps have the necessary methods like delete, inert and getting maximum weighted edge etc., which can solve the Dijkstras and Kruskals algorithms.

Note that my MaxHeap and MaxHeapKruskals are not zero-based indices, the index implementation starts from 1.

The following algorithms are all programmed in Java:

1. To find maximum bandwidth path using Dijkstra's algorithm without heap
2. To find maximum bandwidth path using Dijkstra's algorithm with heap.
3. To find maximum bandwidth path using Kruskal's algorithm.

Each of these programs have been written in their own classes. However, implementations have a separate methods which are public available to other classes.

The Kruskals algorithm has other functions like Make Union Find, but I have integrated them all in the same java class as I would not be using them anywhere else in this project. However, I still made them public in case I would need to access any of these methods from outside my class.

The following classes in my program serve in providing solutions to each of the listed routing algorithms,

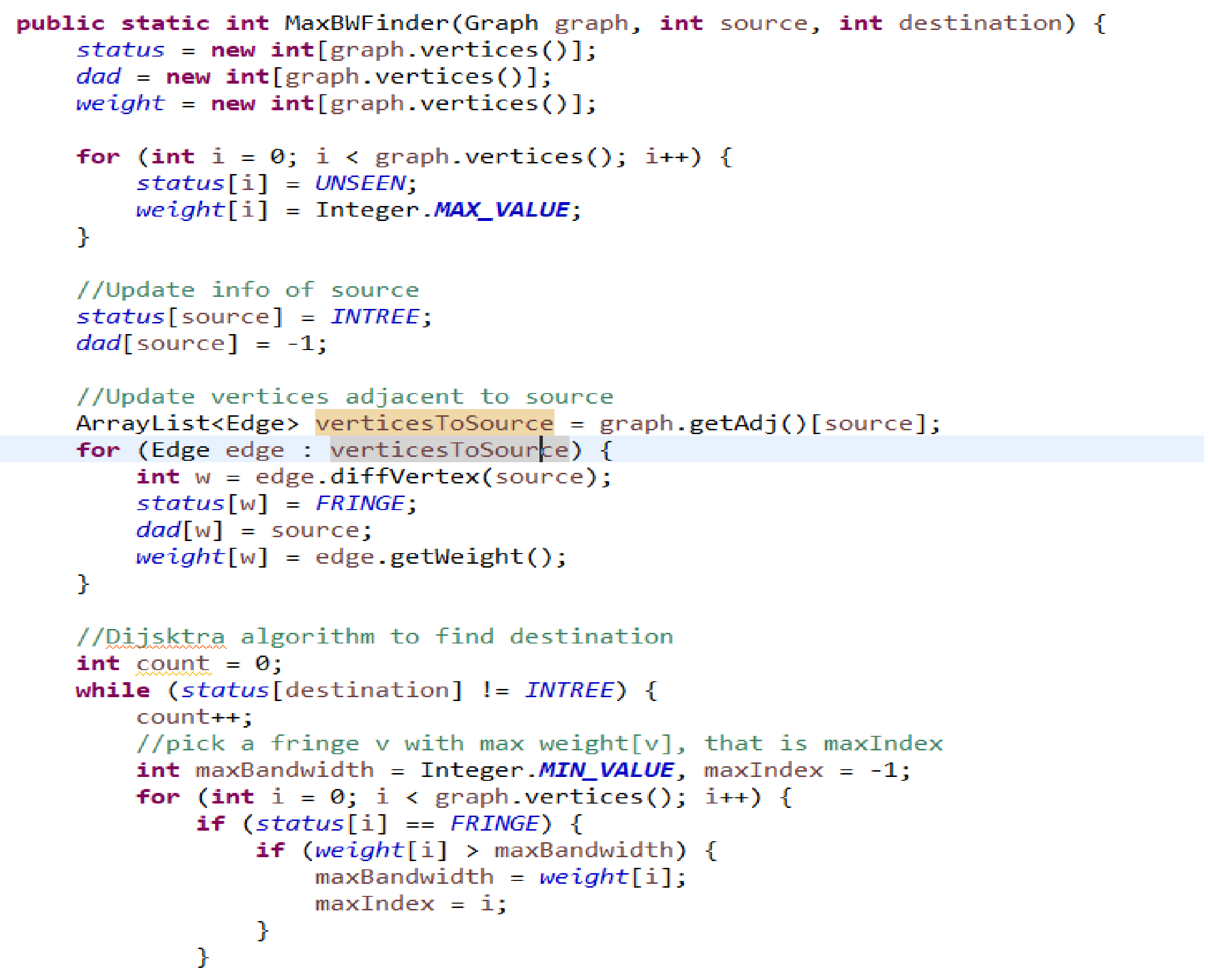
- MaxBWDijkstras.java is the implementation of Algorithm 1
- MaxBWDijkstrasHeapImplementation.java is the implementation of Algorithm 2
- MaxBWKruskals.java is the implementation of Algorithm 3

Each of these classes have their own main methods with both Sparse and Dense graphs so that individual algorithm can be tested.

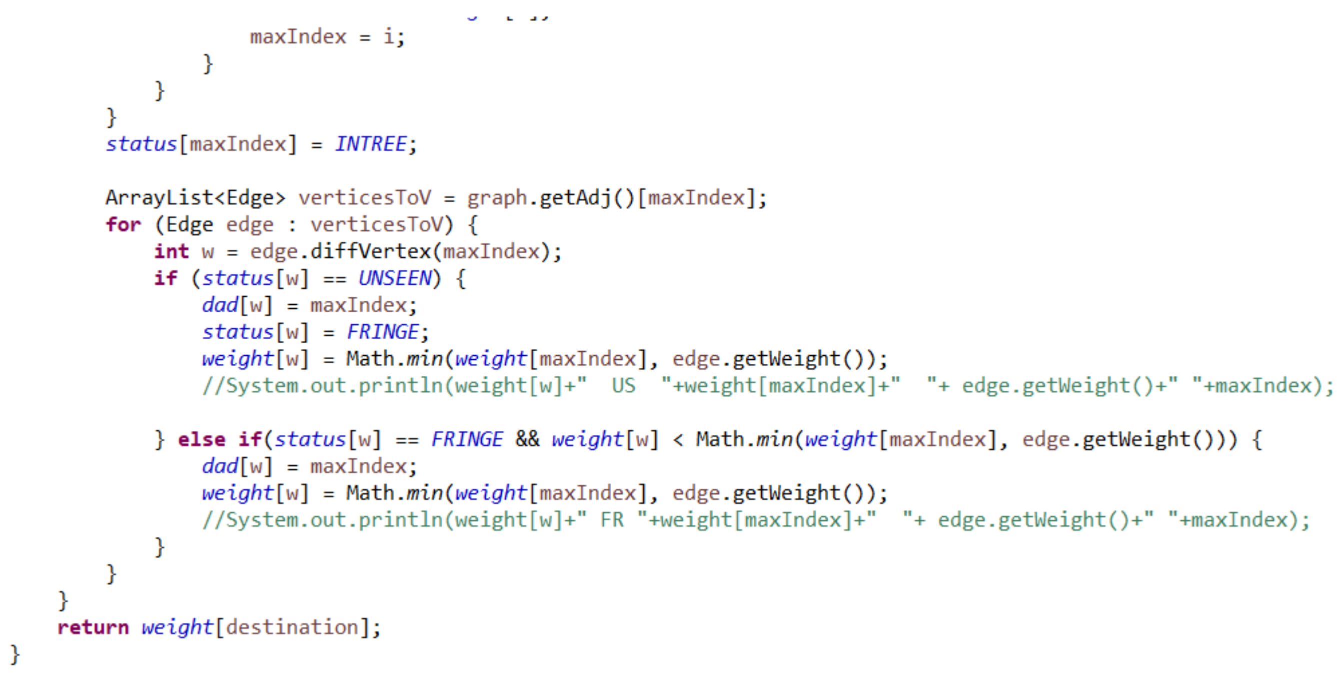
I am using `System.currentTimeMillis();` before and after each algorithm is executed and graph is generated. I am using these values to compute the time taken to execute each of these algorithms.

All these programs have been implemented using algorithms discussed in class by Dr. Chen. I have used the same terminology of Dad, Ranks, Intree, Fringes, Seen etc. as taken in the class notes.

- **Implementation of Algorithm 1 – Dijkstras without heap (Using array)**



```java
public static int MaxBWFinder(Graph graph, int source, int destination) {
    status = new int[graph.vertices()];
    dad = new int[graph.vertices()];
    weight = new int[graph.vertices()];

    for (int i = 0; i < graph.vertices(); i++) {
        status[i] = UNSEEN;
        weight[i] = Integer.MAX_VALUE;
    }

    //Update info of source
    status[source] = INTREE;
    dad[source] = -1;

    //Update vertices adjacent to source
    ArrayList<Edge> verticesToSource = graph.getAdj()[source];
    for (Edge edge : verticesToSource) {
        int w = edge.diffVertex(source);
        status[w] = FRINGE;
        dad[w] = source;
        weight[w] = edge.getWeight();
    }

    //Dijsktra algorithm to find destination
    int count = 0;
    while (status[destination] != INTREE) {
        count++;
        //pick a fringe v with max weight[v], that is maxIndex
        int maxBandwidth = Integer.MIN_VALUE, maxIndex = -1;
        for (int i = 0; i < graph.vertices(); i++) {
            if (status[i] == FRINGE) {
                if (weight[i] > maxBandwidth) {
                    maxBandwidth = weight[i];
                    maxIndex = i;
                }
            }
```



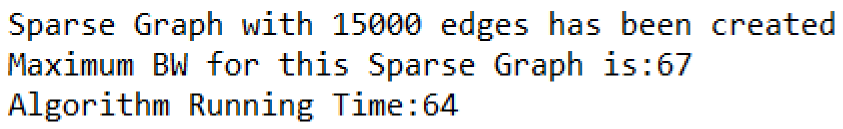
```
                    maxIndex = i;
                }
            }
        }
        status[maxIndex] = INTREE;

        ArrayList<Edge> verticesToV = graph.getAdj()[maxIndex];
        for (Edge edge : verticesToV) {
            int w = edge.diffVertex(maxIndex);
            if (status[w] == UNSEEN) {
                dad[w] = maxIndex;
                status[w] = FRINGE;
                weight[w] = Math.min(weight[maxIndex], edge.getWeight());
                //System.out.println(weight[w]+"  US  "+weight[maxIndex]+"  "+ edge.getWeight()+" "+maxIndex);

            } else if(status[w] == FRINGE && weight[w] < Math.min(weight[maxIndex], edge.getWeight())) {
                dad[w] = maxIndex;
                weight[w] = Math.min(weight[maxIndex], edge.getWeight());
                //System.out.println(weight[w]+" FR "+weight[maxIndex]+"  "+ edge.getWeight()+" "+maxIndex);
            }
        }
    }
    return weight[destination];
}
```

I implemented this algorithm based on the steps taught in class by the professor. It takes a max vertex from the fringe of vertices.

Each time the first for loop in while gets the best band width from the list of fringes available.

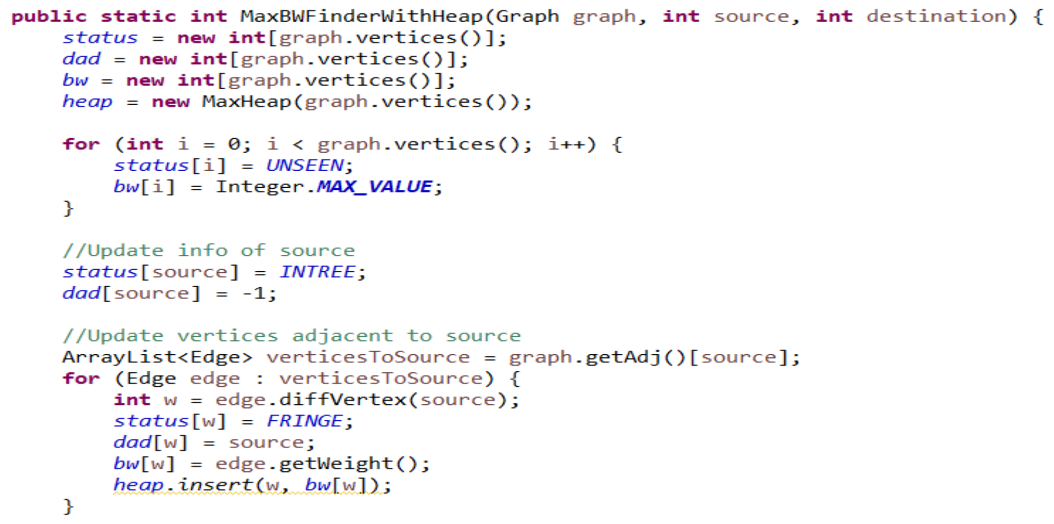
The second for loop iterates lover the adjacent vertices of the node, if the vertex is unseen the first reach it has is the best one and if it is seen it compares with the new path possible weight and keeps on updating the weight[w].

Run Time:

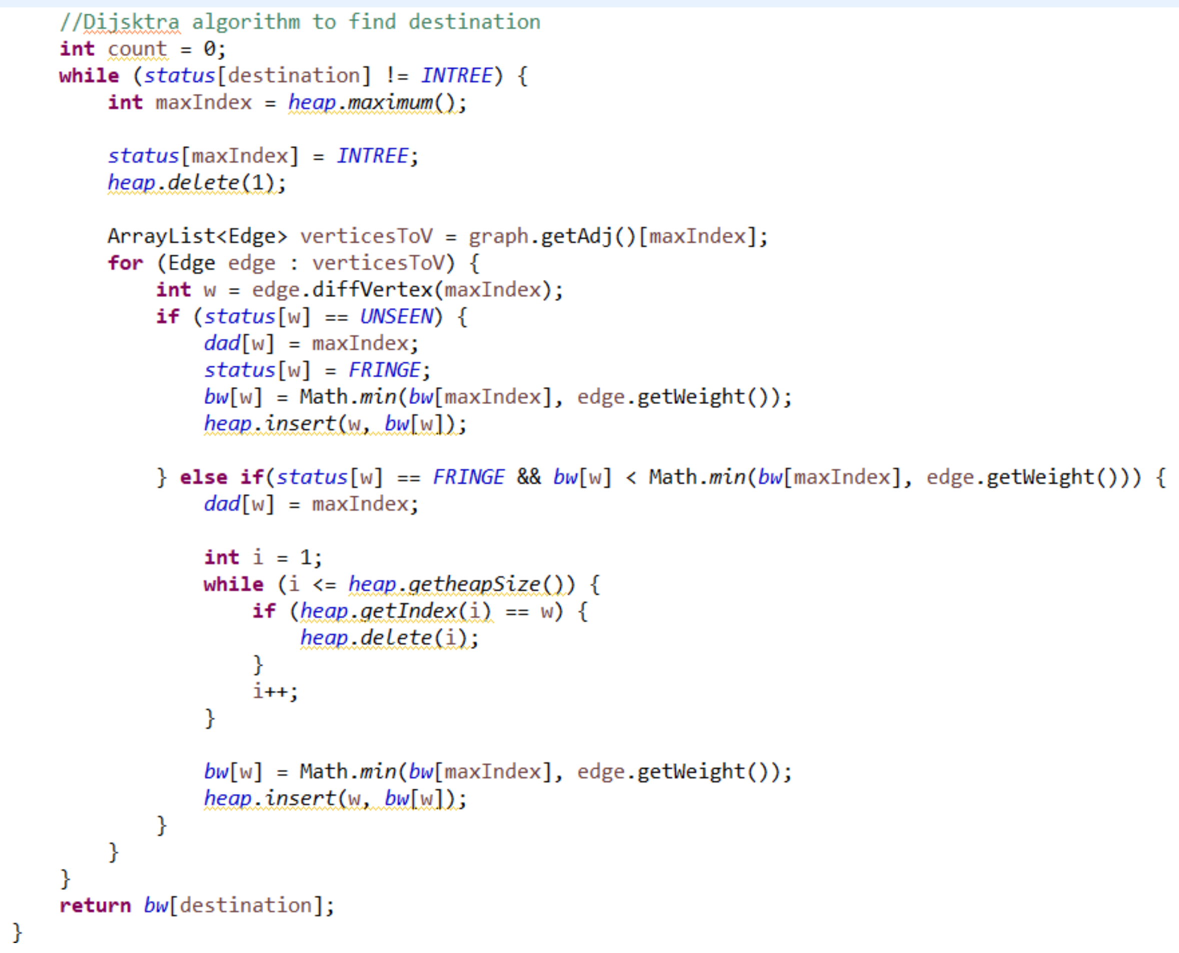


```
Sparse Graph with 15000 edges has been created
Maximum BW for this Sparse Graph is:67
Algorithm Running Time:64
```

- **Implementation of Algorithm 2 – Dijkstras using heap**



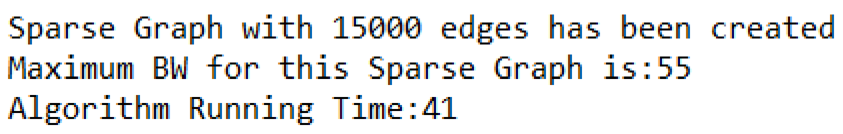
```
public static int MaxBWFinderWithHeap(Graph graph, int source, int destination) {
    status = new int[graph.vertices()];
    dad = new int[graph.vertices()];
    bw = new int[graph.vertices()];
    heap = new MaxHeap(graph.vertices());

    for (int i = 0; i < graph.vertices(); i++) {
        status[i] = UNSEEN;
        bw[i] = Integer.MAX_VALUE;
    }

    //Update info of source
    status[source] = INTREE;
    dad[source] = -1;

    //Update vertices adjacent to source
    ArrayList<Edge> verticesToSource = graph.getAdj()[source];
    for (Edge edge : verticesToSource) {
        int w = edge.diffVertex(source);
        status[w] = FRINGE;
        dad[w] = source;
        bw[w] = edge.getWeight();
        heap.insert(w, bw[w]);
    }
```



```
    //Dijsktra algorithm to find destination
    int count = 0;
    while (status[destination] != INTREE) {
        int maxIndex = heap.maximum();

        status[maxIndex] = INTREE;
        heap.delete(1);

        ArrayList<Edge> verticesToV = graph.getAdj()[maxIndex];
        for (Edge edge : verticesToV) {
            int w = edge.diffVertex(maxIndex);
            if (status[w] == UNSEEN) {
                dad[w] = maxIndex;
                status[w] = FRINGE;
                bw[w] = Math.min(bw[maxIndex], edge.getWeight());
                heap.insert(w, bw[w]);

            } else if(status[w] == FRINGE && bw[w] < Math.min(bw[maxIndex], edge.getWeight())) {
                dad[w] = maxIndex;

                int i = 1;
                while (i <= heap.getheapSize()) {
                    if (heap.getIndex(i) == w) {
                        heap.delete(i);
                    }
                    i++;
                }

                bw[w] = Math.min(bw[maxIndex], edge.getWeight());
                heap.insert(w, bw[w]);
            }
        }
    }
    return bw[destination];
}
```

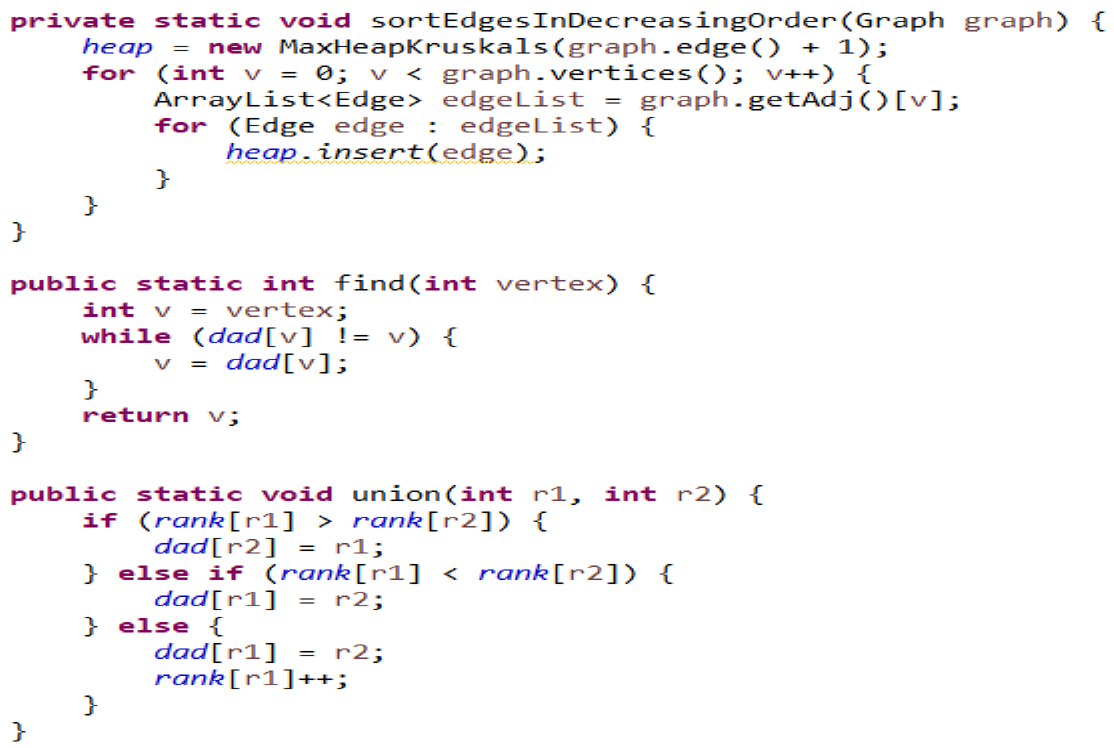
The approach is same as the one where I used an array, here I would not be having a for loop each time to get the best fringe as my MaxHeap always stores the best value at index 1. So, I can use that property of heap to avoid a loop, however each insert and delete statements still take O(logN) time to run.

Run Time:

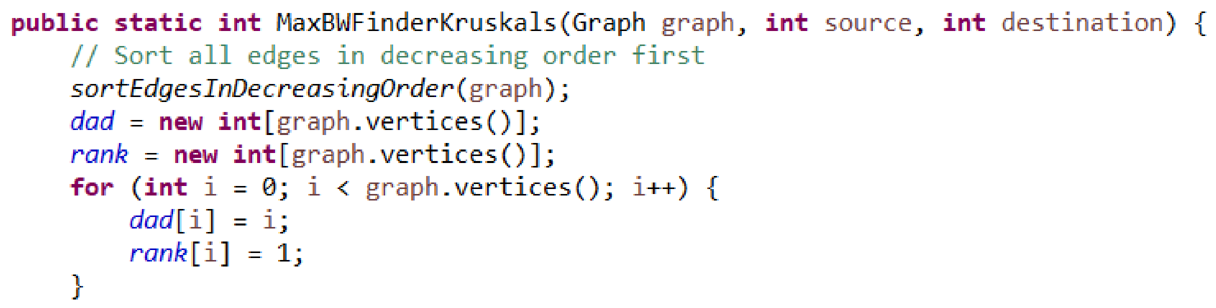


```
Sparse Graph with 15000 edges has been created
Maximum BW for this Sparse Graph is:55
Algorithm Running Time:41
```

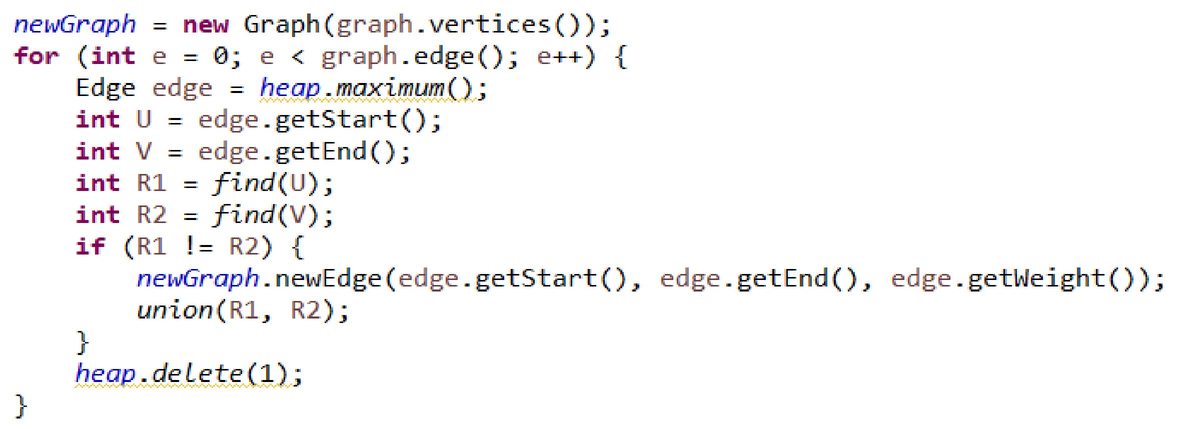
- **Implementation of Algorithm 3 – Kruskals approach using heap**



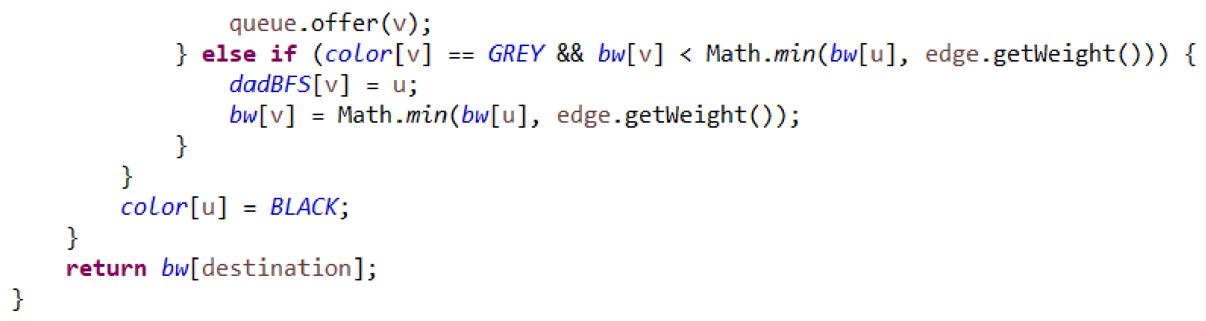
```
private static void sortEdgesInDecreasingOrder(Graph graph) {
    heap = new MaxHeapKruskals(graph.edge() + 1);
    for (int v = 0; v < graph.vertices(); v++) {
        ArrayList<Edge> edgeList = graph.getAdj()[v];
        for (Edge edge : edgeList) {
            heap.insert(edge);
        }
    }
}

public static int find(int vertex) {
    int v = vertex;
    while (dad[v] != v) {
        v = dad[v];
    }
    return v;
}

public static void union(int r1, int r2) {
    if (rank[r1] > rank[r2]) {
        dad[r2] = r1;
    } else if (rank[r1] < rank[r2]) {
        dad[r1] = r2;
    } else {
        dad[r1] = r2;
        rank[r1]++;
    }
}
```



```
public static int MaxBWFinderKruskals(Graph graph, int source, int destination) {
    // Sort all edges in decreasing order first
    sortEdgesInDecreasingOrder(graph);
    dad = new int[graph.vertices()];
    rank = new int[graph.vertices()];
    for (int i = 0; i < graph.vertices(); i++) {
        dad[i] = i;
        rank[i] = 1;
    }
```



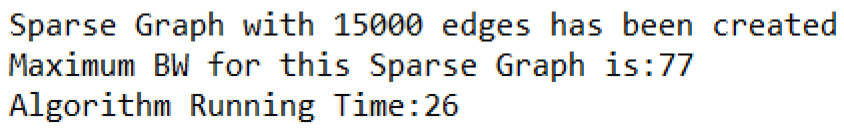
```
newGraph = new Graph(graph.vertices());
for (int e = 0; e < graph.edge(); e++) {
    Edge edge = heap.maximum();
    int U = edge.getStart();
    int V = edge.getEnd();
    int R1 = find(U);
    int R2 = find(V);
    if (R1 != R2) {
        newGraph.newEdge(edge.getStart(), edge.getEnd(), edge.getWeight());
        union(R1, R2);
    }
    heap.delete(1);
}

color = new int[newGraph.vertices()];
dadBFS = new int[newGraph.vertices()];
bw = new int[newGraph.vertices()];
for (int v = 0; v < newGraph.vertices(); v++) {
    color[v] = WHITE;
    dadBFS[v] = -1;
    bw[v] = Integer.MAX_VALUE;
}
color[source] = GREY;
dad[source] = -1;
Queue<Integer> queue = new LinkedList<>();
queue.offer(source);

while (color[destination] != BLACK && !queue.isEmpty()) {
    int u = queue.poll();
    ArrayList<Edge> uEdgeList = newGraph.getAdj()[u];
    for (Edge edge : uEdgeList) {
        int v = edge.diffVertex(u);
        if (color[v] == WHITE) {
            color[v] = GREY;
            bw[v] = Math.min(bw[u], edge.getWeight());
            dadBFS[v] = u;
```



```
                queue.offer(v);
            } else if (color[v] == GREY && bw[v] < Math.min(bw[u], edge.getWeight())) {
                dadBFS[v] = u;
                bw[v] = Math.min(bw[u], edge.getWeight());
            }
        }
        color[u] = BLACK;
    }
    return bw[destination];
}
```

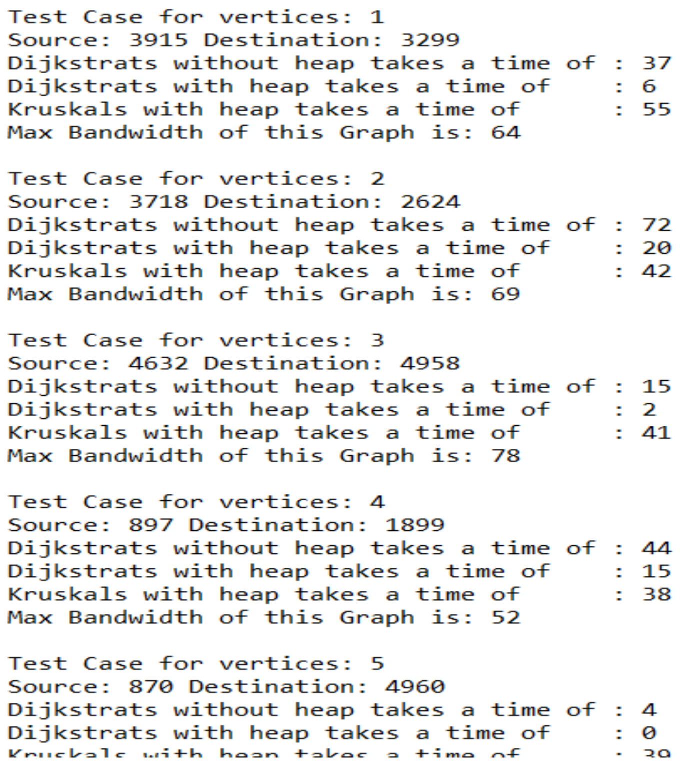
- The approach and implementation for kruskals is different from Dijkstras. Here we would be having 3 additional methods Make Union Find. In this approach we use Kruskal's algorithm to construct a maximum spanning tree, after which we take the path from S, T in G which will give us the maximum bandwidth graph. The approach followed is greedy.

Run Time:



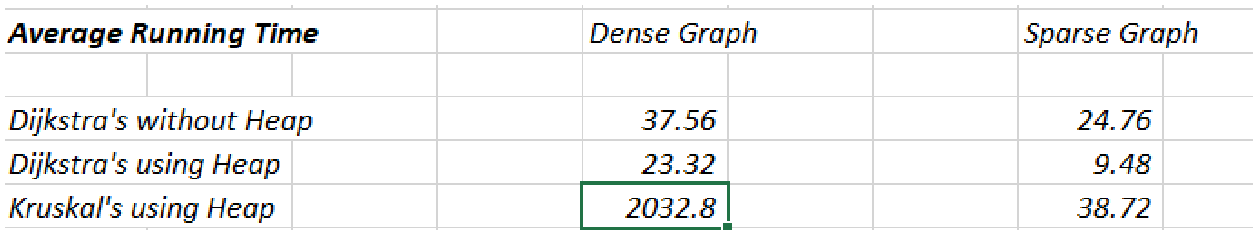
```
Sparse Graph with 15000 edges has been created
Maximum BW for this Sparse Graph is:77
Algorithm Running Time:26
```

➔ The TestingPart4.java class has all these algorithms implemented and the run time of an algorithm was compared against others by taking the same graph and a common set of source and destination for each of the algorithms. As given in the problem, 5 Graphs and 5 sets of source and destinations are run on both Sparse and Dense graphs. This has given out of 50 result size in total to compare the performance of each of these algorithms.



```
Test Case for vertices: 1
Source: 3915 Destination: 3299
Dijkstrats without heap takes a time of : 37
Dijkstrats with heap takes a time of    : 6
Kruskals with heap takes a time of      : 55
Max Bandwidth of this Graph is: 64

Test Case for vertices: 2
Source: 3718 Destination: 2624
Dijkstrats without heap takes a time of : 72
Dijkstrats with heap takes a time of    : 20
Kruskals with heap takes a time of      : 42
Max Bandwidth of this Graph is: 69

Test Case for vertices: 3
Source: 4632 Destination: 4958
Dijkstrats without heap takes a time of : 15
Dijkstrats with heap takes a time of    : 2
Kruskals with heap takes a time of      : 41
Max Bandwidth of this Graph is: 78

Test Case for vertices: 4
Source: 897 Destination: 1899
Dijkstrats without heap takes a time of : 44
Dijkstrats with heap takes a time of    : 15
Kruskals with heap takes a time of      : 38
Max Bandwidth of this Graph is: 52

Test Case for vertices: 5
Source: 870 Destination: 4960
Dijkstrats without heap takes a time of : 4
Dijkstrats with heap takes a time of    : 0
Kruskals with heap takes a time of      : 39
```

I have consolidated these results in an excel and found the average running time:



Average Running Time	Dense Graph	Sparse Graph
Dijkstra's without Heap	37.56	24.76
Dijkstra's using Heap	23.32	9.48
Kruskal's using Heap	2032.8	38.72

Performance: Dijkstra with Heap > Dijkstra without Heap > Kruskal

We see that Dijkstra's when using Heap is performing better on average than Dijkstra's without heap. As Array is being used and to get the maximum next BandWidth we were traversing the entire array. However, in case of Kruskal's it is taking longer than the other two algorithms. Also, these results are widely varying with sources and destinations.

This might be because Kruskal's uses edges on Heap, so for a Dense which typically has thousands of edges in whole, the sorting method here would be taking quite long. Also Insert and Delete operations even though O(logN) might be more expensive than what it happens on Dijkstra's because of the large constant values.

Due to the above reasons, we can justify why our results behave the way they do!

Future Improvements:

➔ If space is not a constraint and performance is very crucial than I might have used a 2-dimensional array instead of an adjacent list for my graph which would speed up my process eliminating the need of traversing the ArrayList multiple times.

➔ Also, I am not updating rank of remaining elements in the path of my rank array during Find operations, this approach was discussed by Dr.Chen in the class, However I have not implemented that and that would have made my Kruskal's faster than it is now.

These are the 2 areas I believe could have been used to increase the overall performance.