COMPUTER SCIENCE
& ENGINEERING
TEXAS A&M UNIVERSITY

NETWORK OPTIMIZATION

PROJECT REPORT

ANALYSIS OF ALGORITHMS

CSCE 629

FALL 2022

RISHABH BASSI

UIN: 532008692

1 Problem Statement

Network optimization has been an important area in the current research in computer science and computer engineering. In this course project, you will implement a network routing protocol using the data structures and algorithms we have studied in class. This provides you with an opportunity to translate your theoretical understanding into a real-world practical computer program. Translating algorithmic ideas at a "higher level" of abstraction into real implementations in a particular programming language is not at all always trivial. The implementations often force you to work on more details of the algorithms, which sometimes may lead to much better understanding.

2 Introduction

A Maximum Bandwidth Problem is basically finding a path from vertex s to vertex t in a weighted graph G such that the bandwidth of the path is the highest among all paths from vertex s to vertex t. There are several uses for the Max-BW issue in computer networks, including network dependability, network communication, and network flow analysis. This project is implemented in Java and has the main following functionalities:

1. Random Graph Generation: Returns connected undirected graph depending upon input of number of nodes and degree of each vertex.
2. Heap Structure: Implements functionalities of Heap (Insert, delete etc.) using three arrays as mentioned in project description.
3. Maximum Spanning Tree Generation: Implements two main functionalities of Make Union Find Algorithms and Depth First Search to find max bandwidth path from source to destination in max spanning tree.

4. Three Routing Algorithms:
    a. Dijkstra's without Heap
    b. Dijkstra's with Heap
    c. Kruskal's Algorithm

The report is divided into the following sections: Implementation Details in Section 3, Testing and Analysis of the algorithms in Section 4, Further Improvements in Section 5, and Conclusion for this project report in Section 6.

3 Implementation

3.1 Random Graph Generation

The Graphs are stored in Adjacency List. Because it is preferable for sparse graph storage, adjacency list was chosen in the implementation over adjacency matrix. Adjacency matrices have an O(n2) time complexity, but adjacency lists have an O(m+n) time complexity (where m n, m = number of edges, and n = number of vertices). Another justification for choosing a list over a matrix is that the project's Dijkstra and Kruskal algorithm implementations need edge traversals on vertices, which work better in adjacency lists. The algorithms for calculating the maximum bandwidth path are implemented on sparse, dense, undirected, and weighted graphs. The graph is undirected, therefore if there is an edge between vertex 1 and vertex 2, there should also be an edge between vertex 2 and vertex 1. Each edge's weight is chosen at random and has a value between 1 and 5000.

3.1.1 Graph 1

This sparse graph has an average degree of six for each vertex. Before any random edges are added to the graph, a cycle is built between each of the 5000 vertices to keep it always linked. For graph generation 1, the time complexity is O(m), where m is the number of edges. Below is the approach used to implement this (Figure. 1).

```
public static void sparseGraph(Class_Graph classGraph, int wtLimit, int deg, int numberNodes) {

    Class_Vertex[] vertices = classGraph.getVertices();
    List<Class_Edge> classEdges = classGraph.getEdges();
    int avSum_edge = numberNodes * deg / 2;
    int curSum_edge = numberNodes;

    int src, dest;
    while (curSum_edge < avSum_edge) {
        src = random.nextInt(numberNodes);
        dest = random.nextInt(numberNodes);

        if (!adjList(vertices[src], dest) && src != dest) {
            // random edge weight generator variable
            int weight = random.nextInt(wtLimit) + 1;
            classEdges.add(new Class_Edge(src, dest, weight));
            vertices[src] = new Class_Vertex(dest, weight,  deg: vertices[src].getDeg() + 1, vertices[src]);
            vertices[dest] = new Class_Vertex(src, weight,  deg: vertices[dest].getDeg() + 1, vertices[dest]);
            curSum_edge++;
        }
    }
```

3.1.2 Graph 2

Each vertex must be adjacent to around 20% of the other vertices in the second graph G2. A vertices is picked, then a set of random edges are added to the vertex. This shows that the degree of each vertex is roughly 1000. To do this, pick any vertex and keep adding edges to it until the vertex's degree is equal to 20% of 5000, or 1000. A highly connected graph is created after doing this for each vertices. For graph generation 2, the time complexity is O(m), where m is the number of edges. Below is the approach used to implement this (Figure. 2)

```
private static void denseGraph(Class_Graph classGraph, int wtLimit, int degree, int numberNodes) {

    List<Class_Edge> classEdges = classGraph.getEdges();
    int wt, dest;
    int currDeg;

    for (int i = 0; i < numberNodes; i++) {
        currDeg = classGraph.getVertices()[i].getDeg();
        while (currDeg < degree) {
            dest = random.nextInt(numberNodes);
            if (!adjList(classGraph.getVertices()[i], dest)
                    && i != dest
            ) {
                wt = random.nextInt(wtLimit) + 1;
                classEdges.add(new Class_Edge(i, dest, wt));
                classGraph.getVertices()[i] = new Class_Vertex(dest, wt,
                        deg: classGraph.getVertices()[i].getDeg() + 1, classGraph.getVertices()[i]);
                classGraph.getVertices()[dest] = new Class_Vertex(i, wt,
                        deg: classGraph.getVertices()[dest].getDeg() + 1, classGraph.getVertices()[dest]);
                currDeg++;
            }
        }
    }
}
```

3.2 Heap

A Max-Heap requires that the root node's root key rank highest among all of its children's root keys and this should hold true for every subtree in binary tree. In a modified version of Dijkstra's approach, the functions Insert (), Delete (), and extractMaximum () from the heap are supported by the Max Heap structure. In the modified Dijkstra's algorithm, these three operations are carried out for fringes using this data structure. Three arrays—heap, vertex, and position—were used to implement the maximum heap. The heap array, which is basically the D array as mentioned in the project statement, keeps track of the bandwidth associated with that vertex. In order to meet the heap property, we run the heapify method after adding the element to the heap's last index and setting the bandwidth, vertices, and position in the three arrays appropriately. Time required for the insertion operation is O

(logn). When we do the swap operation, we make sure that the values in all three arrays have been switched for the particular pair of vertices. This improves our Delete action since we can avoid inefficiently searching for a specific vertex in the heap because we know the precise location of the vertex in the position/ array. Time required for the deletion operation is O (logn) and to extract max element from heap takes O (1).

3.3 Dijkstra Algorithm without using Heap

The shortest path Dijkstra algorithm can be tweaked to obtain the path's greatest bandwidth. The status of the vertices is kept in a status array. Unseen is 0, fringe is 1, and intrigue is 2. The maximum bandwidth for that vertex from the source vertex is stored in a bandwidth array. The path from source to destination is obtained using a dad array. The loop continues as long as there are fringes in the array used to store all of the fringes. The outer loop so requires time O (n). The largest fringe is removed from each loop. This will require O(n) time because fringes are stored in a straightforward array. The inner loop inserts the unseen vertices as fringes and, if necessary, modifies the bandwidth value of the fringes. Since there can only be n nodes and m edges, the inner loop will take O(n+m) time to complete. As a result, this algorithm's temporal complexity is O(n(n+m)).

3.4 Dijkstra Algorithm using Heap

Comparable to the prior implementation is this one. Fringes are nonetheless kept in a heap structure as opposed to an array. It makes advantage of the maximum heap structure described in the section above. Because the greatest value will always be present at the tree's root, the time required to obtain the maximum fringe is decreased from O(n) to O (1). The maximum height of a binary tree means that deleting and adding a fringe from it will take O (log n) time. As a result, because there are n vertices, the outer loop will take time O (m log n). Given that there are a maximum of m edges and n vertices, the inner loop will take time O (m log n). As a result, this algorithm has an O((n+m) log n) time complexity.

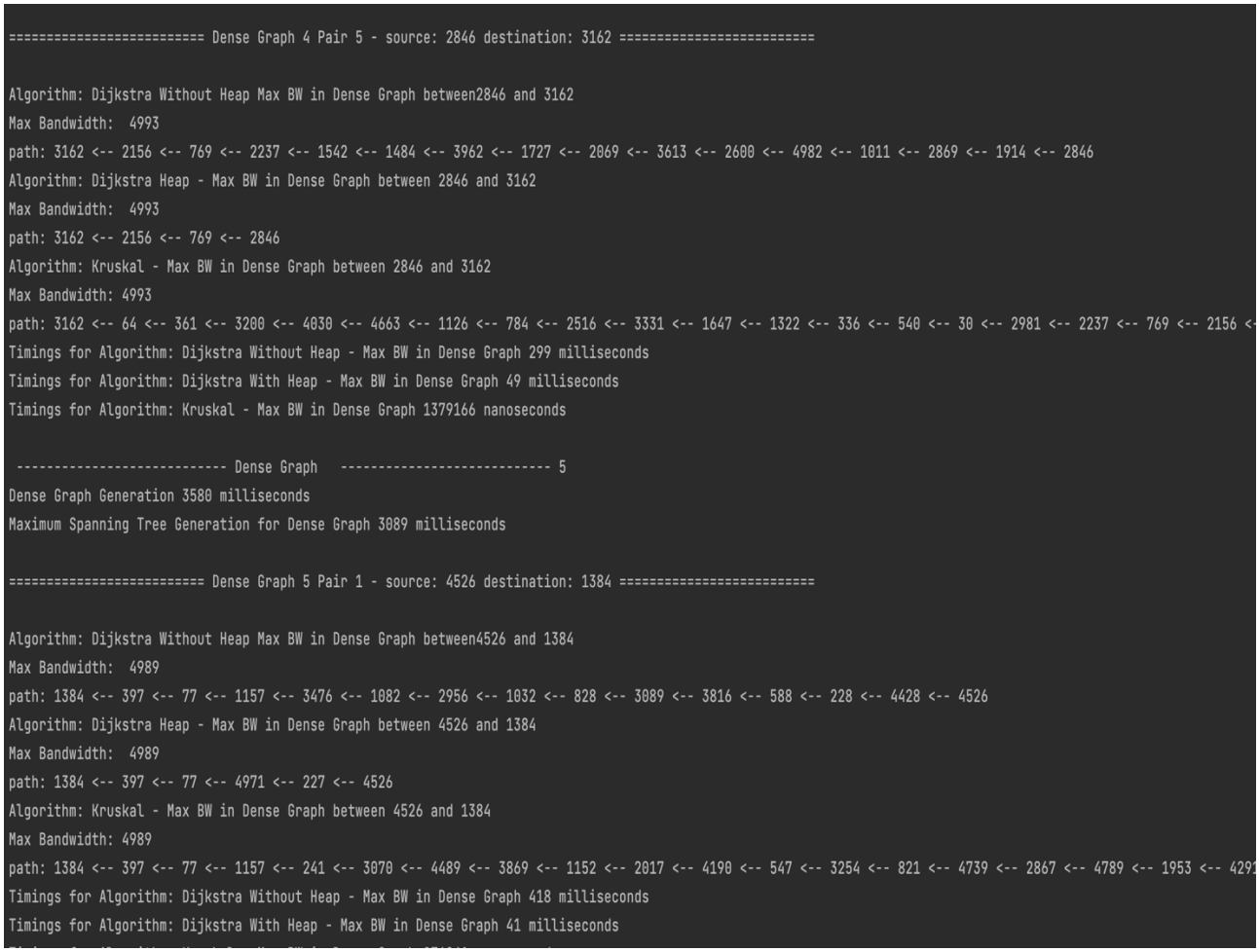
3.5 Kruskal Algorithm

A maximum spanning tree can be built using Kruskal's algorithm. The maximum bandwidth path between the source and the destination can be obtained using this tree. The edges are arranged in decreasing order of their weights for the purpose of building the largest spanning tree. Since there are m edges, this sorting uses a heap algorithm and requires time O (m log n). The MST is built by performing union and find operations for each edge. The union operation takes O (1) time and the find operation takes time O (log n) and it is performed on m edges, so it takes time O (m log n). As described in the class, find operation is further optimized so that further find operations take less time (Referred to as Path Compression). DFS is used to get the path between source and destination from the MST which will take time O(m+n) in the worst case when the number of edges in MST is almost same as in the graph. So, the total time complexity for this algorithm is O (m (log m + log n)).

4 Testing and Analysis

4.1 Sample Output

This is an example output that includes the maximum bandwidth and the Path from source to destination for all 3 algorithms. On five randomly chosen sparse and dense graphs, the three algorithms were put to the test. Five pairs of source and destination vertices were randomly selected for every graph. We can see that although the maximum bandwidth pass varies for each algorithm, the maximum bandwidth returned by all three techniques is the same. The running time of each algorithm is also displayed in the output.



```
========================= Dense Graph 4 Pair 5 - source: 2846 destination: 3162 =========================

Algorithm: Dijkstra Without Heap Max BW in Dense Graph between2846 and 3162
Max Bandwidth:  4993
path: 3162 <-- 2156 <-- 769 <-- 2237 <-- 1542 <-- 1484 <-- 3962 <-- 1727 <-- 2069 <-- 3613 <-- 2600 <-- 4982 <-- 1011 <-- 2869 <-- 1914 <-- 2846
Algorithm: Dijkstra Heap - Max BW in Dense Graph between 2846 and 3162
Max Bandwidth:  4993
path: 3162 <-- 2156 <-- 769 <-- 2846
Algorithm: Kruskal - Max BW in Dense Graph between 2846 and 3162
Max Bandwidth: 4993
path: 3162 <-- 64 <-- 361 <-- 3200 <-- 4030 <-- 4663 <-- 1126 <-- 784 <-- 2516 <-- 3331 <-- 1647 <-- 1322 <-- 336 <-- 540 <-- 30 <-- 2981 <-- 2237 <-- 769 <-- 2156 <-
Timings for Algorithm: Dijkstra Without Heap - Max BW in Dense Graph 299 milliseconds
Timings for Algorithm: Dijkstra With Heap - Max BW in Dense Graph 49 milliseconds
Timings for Algorithm: Kruskal - Max BW in Dense Graph 1379166 nanoseconds

 --------------------------- Dense Graph   --------------------------- 5
Dense Graph Generation 3580 milliseconds
Maximum Spanning Tree Generation for Dense Graph 3089 milliseconds

========================= Dense Graph 5 Pair 1 - source: 4526 destination: 1384 =========================

Algorithm: Dijkstra Without Heap Max BW in Dense Graph between4526 and 1384
Max Bandwidth:  4989
path: 1384 <-- 397 <-- 77 <-- 1157 <-- 3476 <-- 1082 <-- 2956 <-- 1032 <-- 828 <-- 3089 <-- 3816 <-- 588 <-- 228 <-- 4428 <-- 4526
Algorithm: Dijkstra Heap - Max BW in Dense Graph between 4526 and 1384
Max Bandwidth:  4989
path: 1384 <-- 397 <-- 77 <-- 4971 <-- 227 <-- 4526
Algorithm: Kruskal - Max BW in Dense Graph between 4526 and 1384
Max Bandwidth: 4989
path: 1384 <-- 397 <-- 77 <-- 1157 <-- 241 <-- 3070 <-- 4489 <-- 3869 <-- 1152 <-- 2017 <-- 4190 <-- 547 <-- 3254 <-- 821 <-- 4739 <-- 2867 <-- 4789 <-- 1953 <-- 429
Timings for Algorithm: Dijkstra Without Heap - Max BW in Dense Graph 418 milliseconds
Timings for Algorithm: Dijkstra With Heap - Max BW in Dense Graph 41 milliseconds
```

4.2 Timing Analysis for Sparse Graph

All timings below are in seconds.

4.2.1 Graph Generation

Graph	Graph1	Graph2	Graph3	Graph4	Graph5
Duration	0.016	0.01	0.005	0.006	0.004

4.2.2 Dijkstra without Heap

Duration	Pair1	Pair2	Pair3	Pair4	Pair5
Graph1	0.245	0.109	0.109	0.12	0.116
Graph2	0.105	0.091	0.094	0.08	0.093
Graph3	0.092	0.081	0.087	0.075	0.075
Graph4	0.083	0.086	0.099	0.082	0.07
Graph5	0.07	0.08	0.068	0.065	0.076

4.2.3 Dijkstra With Heap

Duration	Pair1	Pair2	Pair3	Pair4	Pair5
Graph1	0.023	0.009	0.012	0.021	0.018
Graph2	0.009	0.004	0.002	0.002	0.004
Graph3	0.004	0.002	0.002	0.001	0.003
Graph4	0.002	0.002	0.002	0.002	0.002
Graph5	0.002	0.002	0.004	0.001	0.004

4.2.4 Kruskal Algorithm

Time to find Kruskal is divided into two steps: First being Time taken for Max Spanning Tree Generations which is in milliseconds. Since MST Generation is common for each graph, we don't want to build this tree for each pair every time. Hence, we build it out of inner for loop as part of Code Optimization. Second part of timing is DFS to find path from source to destination.

Graph	Graph1	Graph2	Graph3	Graph4	Graph5
Duration	0.061	0.02	0.017	0.015	0.009

MST Build Time

Duration	Pair1	Pair2	Pair3	Pair4	Pair5
Graph1	0.024	0.005	0.011	0.016	0.003
Graph2	0.006	0.004	0.001	0.002	0.002
Graph3	0.002	0.006	0.0004	0.002	0.002
Graph4	0.002	0.0005	0.002	0.0005	0.001
Graph5	0.0009	0.0009	0.001	0.001	0.001

DFS Path Time

Duration	Pair1	Pair2	Pair3	Pair4	Pair5
Graph1	0.085	0.066	0.072	0.077	0.064
Graph2	0.026	0.024	0.021	0.022	0.022
Graph3	0.019	0.023	0.0174	0.019	0.019
Graph4	0.017	0.0155	0.017	0.0155	0.016
Graph5	0.0099	0.0099	0.01	0.01	0.01

Total Time

4.3 Timing Analysis for Dense Graph

All timings below are in seconds.

4.3.1 Graph Generation

Graph	Graph1	Graph2	Graph3	Graph4	Graph5
Duration	4.715	3.877	3.536	3.948	3.580

4.3.2 Dijkstra Without Heap

Duration	Pair1	Pair2	Pair3	Pair4	Pair5
Graph1	0.482	0.314	0.381	0.305	0.305
Graph2	0.480	0.295	0.283	0.309	0.282
Graph3	0.423	0.305	0.302	0.293	0.384
Graph4	0.489	0.317	0.299	0.288	0.299
Graph5	0.418	0.270	0.271	0.269	0.292

4.3.3 Dijkstra With Heap

Duration	Pair1	Pair2	Pair3	Pair4	Pair5
Graph1	0.067	0.046	0.046	0.049	0.046
Graph2	0.038	0.037	0.038	0.037	0.037
Graph3	0.044	0.044	0.046	0.046	0.045
Graph4	0.057	0.048	0.055	0.048	0.049
Graph5	0.041	0.041	0.040	0.039	0.042

4.3.4 Kruskal Algorithm

Time to find Kruskal is divided into two steps: First being Time taken for Max Spanning Tree Generations which is in milliseconds. Since MST Generation is common for each graph, we don't want to build this tree for each pair every time. Hence, we build it out of inner for loop as part of Code Optimization. Second part of timing is DFS to find path from source to destination.

Graph	Graph1	Graph2	Graph3	Graph4	Graph5
Duration	3.163	3.077	3.117	3.116	3.089

MST Build Time

Duration	Pair1	Pair2	Pair3	Pair4	Pair5
Graph1	0.005	0.0006	0.002	0.002	0.002
Graph2	0.004	0.0004	0.0005	0.0007	0.0009
Graph3	0.002	0.0006	0.0005	0.001	0.001
Graph4	0.003	0.0009	0.002	0.0005	0.001
Graph5	0.001	0.0005	0.0005	0.0004	0.0007

DFS Path Time

Duration	Pair1	Pair2	Pair3	Pair4	Pair5
Graph1	3.168	3.1636	3.165	3.165	3.165
Graph2	3.081	3.0774	3.0775	3.0777	3.0779
Graph3	3.119	3.1176	3.1175	3.118	3.118
Graph4	3.119	3.1169	3.118	3.1165	3.117
Graph5	3.090	3.0895	3.0895	3.0894	3.0897

Total Time

4.4 Discussion on the Outputs

4.4.1 Theoretical Time Complexity

- **Dijkstra without Heap: $O(n^2)$**
- **Dijkstra with Heap: $O((n + m) \log n)$**
- **Maximum Spanning Tree Build: $O(m \log n)$**
- **DFS to find path time: $O(m)$**

4.4.2 Analysis of Graphs

Timings Best to Worst:

For sparse graph -> Dijkstra's with heap > Kruskal's > Dijkstra's without heap.
For dense graph -> Dijkstra's with heap > Dijkstra's without heap > Kruskal's.

The theoretical Complexities show that the Dijkstra Implementation with Heap is quicker than the one without Heap, and the output table confirms this. According to the aforementioned findings, the implementation of Dijkstra's without a heap takes longer than the implementation of Dijkstra's with a heap. This performance matches what was anticipated. This indicates that using a heap to store the fringes speed up the algorithm's execution time. Theoretically, Kruskal's solution should be superior to Dijkstra's. But for dense graph, this is not the conduct that has been seen. The assumption that certain operations take a constant amount of time but that this time is large enough to have an impact on the algorithm's overall performance could be the cause.

It can be seen that given a dense graph, MST's construction takes longer than Dijkstra's, both with and without a heap. More time is required to construct MST than to construct either Dijkstra's in dense graph. This may be due to the fact that a dense graph has a significantly higher number of edges than a sparse graph. All of the graphs also show that utilizing DFS to discover the path between the source and the destination takes less time than using both Dijkstra's.

For Kruskal, all 5 random pairings of vertices can be represented by the greatest spanning tree built for one graph. Thus, Kruskal's is the best option if several paths need to be located in a single graph. However, Dijkstra's with heap is a better option if other graphs are considered.

5 Further Improvements

The number of edges in a graph appears to have an impact on Kruskal's performance. As was previously said, the more edges in the graph, the longer it takes to sort in Kruskal's. A more effective sorting method could help Kruskal's perform better.

6 Conclusion

Although the theoretical time complexity is the same, the outputs show that the running times for each algorithm vary depending on the instance. This is due to the fact that each graph prevents the algorithm from reaching its worst-case scenario, which is symbolized by the theoretical complexity, and that graphs are random in nature. When we need to find a single path connecting two vertices, Dijkstra performs better. We benefit from the Kruskal algorithm's ability to generate a given MST just once and determine the path based on that MST in linear time.