NETWORK OPTIMIZATION USING NETWORK ROUTING PROTOCOLS

A Report submitted

in fulfilment of the coursework program CSCE 629 Analysis of Algorithms

By

UNNATI ERAMANGALATH

UIN: 930001393

Pursued in

Texas A&M University

Fall 2019

Under the guidance of

Prof. Dr. Jianer Chen

College Station, Texas

26 November 2019

ABSTRACT

Network optimization is a technology used for improving network performance for a given environment. It is considered an important component of effective information systems management. Network optimization plays an important role as information technology is growing at exponential rates with business users producing large volumes of data and thus consuming larger network bandwidths. A routing protocol specifies how routers communicate with each other, distributing information that enables them to select routes between any two nodes on a computer network.

In this course project, I have implemented routing algorithms to address Maximum Bandwidth Path problem where I need to find a path of maximum bandwidth between two vertices (source and destination vertex) in each weighted undirected randomly generated graph G.

Firstly, I generated two kinds of "random" undirected graph of 5000 vertices each. First kind is a sparse graph (A graph in which the number of edges is much less than the possible number of edges) with the average vertex degree as 6. Second kind of generated graph is a dense graph (a graph in which the number of edges is close to the maximal number of edges) where each vertex is adjacent to about 20% of the other vertices that are randomly chosen.

I have implemented three different approaches to find the maximum Bandwidth Path in the randomly generated graph:

- An algorithm based on a modification of Dijkstra's algorithm without using a heap structure.
- An algorithm based on a modification of Dijkstra's algorithm using a heap structure for fringes.
- An algorithm based on a modification of Kruskal's algorithm, in which the edges are sorted by Heapsort.

Algorithms have been implemented in JAVA programming language on Eclipse IDE.

Finally, I have tested the implemented algorithms on 5 pairs of randomly generated graphs for selected source-destination vertices and presented the analysis based on their running times.

1. INTRODUCTION

Maximum Bandwidth/Widest Path Problem:

The widest path problem is the problem of finding a path between two designated vertices in a weighted graph G having source node s and destination node t, maximizing the weight w of the minimum-weight edge in the path.

Dijkstra's Algorithm

Dijkstra's Algorithm is an algorithm for finding the shortest paths between nodes in a graph. Here I have modified the algorithm to find the maximum bandwidth path by implementing the following algorithm. This algorithm uses max Heap data structure to find the best vertex having maximum value.

```
Algorithm: Dijkstra's algorithm with Heap (G = (V, E), s, w)
 1: for each v ∈ V do
 2:    status[v] = unseen;
 3: end for
 4: status[s] = in-tree;
 5: for each edge [s, w] ∈ E do
 6:    status[w] = fringe; dad[v] = s;
 7:    wt[w] = weight (s, w); Insert (Heap, w);
 8: end for
 9: while Heap ≠ ∅, do
10:    v = Max (Heap); status[v] = in-tree; Delete (Heap; v);
11:    for each edge (v, w) ∈ E do
12:       if status[w]= unseen then
13:          status[w] = fringe; dad[w] = v; wt[w] = min{wt[v], weight (v, w)};
14:          Insert (Heap, w);
15:       else if status[w]==fringe && wt[w] < min{wt[v], weight (v, w)}
16:          Delete (Heap, w); dad[w] = v;
17:          wt[w] = min{wt[v], weight (v, w)}; Insert (Heap, w);
18:       end if
19:    end for
20: end while
21: return dad []; wt []
```

Figure 1. Dijkstra's Algorithm – Heap Implementation

The following modification of Dijkstra's algorithm has been implemented in the second approach where array list collection of Java language is used instead of Heap data structure to find the maximum bandwidth path.

```
Algorithm: Dijkstra's algorithm using ArrayList (without Heap) (G = (V, E), s, w)
 1: for each v ∈ V do
 2:    status[v] = unseen;
 3: end for
 4: status[s] = in-tree;
 5: for each edge [s, w] ∈ E do
 6:    status[w] = fringe; dad[v] = s;
 7:    wt[w] = weight (s, w); Add (ArrayList, w);
 8: end for
 9: while ArrayList ≠ ∅, do
10:    v = Max (ArrayList); status[v] = in-tree; Remove (ArrayList; v);
11:    for each edge (v, w) ∈ E do
12:       if status[w]= unseen then
13:          status[w] = fringe; dad[w] = v; wt[w] = min{wt[v], weight (v, w)};
14:          Add (ArrayList, w);
15:       else if status[w]==fringe && wt[w] < min{wt[v], weight (v, w)}
16:          Remove (ArrayList, w); dad[w] = v;
17:          wt[w] = min{wt[v], weight (v, w)}; Add (ArrayList, w);
18:       end if
19:    end for
20: end while
21: return dad []; wt [];
```

Figure 2. Dijkstra's Algorithm – Without Heap Implementation

Kruskal's Algorithm

Kruskal's algorithm is a minimum-spanning-tree algorithm which finds an edge of the least possible weight that connects any two trees in the forest. Modified version of Kruskal's algorithm is implemented to find the maximum spanning tree for a connected weighted graph that adds cost arcs at each step in decreasing manner. Performing Depth First Search (DFS) on edges from Maximum spanning tree gives us the maximum bandwidth path.

```
Algorithm: KRUSKAL's MAXIMUM SPANNING TREE (G (V, E))
1:    sort edge weights of G in non-increasing order using Heapsort
2:    T = ∅;
3:    for each vertex v in G do
4:       MAKE SET(v).
5:    for i = 1 to m where m ∈ E do
6:       let ei = [ui, vi];
7:       r1 = FIND (ui);
8:       r2 =  FIND (vi);
9:       if r1 ≠ r2 then
10:         UNION(r1,r2);
11:         T = T + ei;
```

Figure 3. Kruskal's Algorithm for Maximum Spanning Tree

```
Algorithm: Kruskal's Maximum Bandwidth Path
1:    T = MAXIMUM SPANNING TREE (G)
2:        Add T to adjacency List L; Bandwidth[source] = ∞
3:    DFS (L, source)
```

Figure 4. Kruskal's Algorithm for Maximum Bandwidth Path

2. RANDOM GRAPH GENERATION

In order to generate two kinds of random undirected graphs, I have implemented an Edge1 class and GraphGenerator1 class in graph Generator package. The Edge class defines 3 variables for source vertex, destination vertex and edge weight values between two selected vertices. GraphGenerator class is the main class which generates 2 kinds of random undirected graph:

a) Generate Sparse Graph having an average vertex degree of 6:

Average degree = Number of total edges *2 / Total number of vertices
Total number of vertices for our implementation = 5000
Average degree = 6
Hence total number of edges = 6*5000/2 = 15000

To ensure that the graph is connected, a function is implemented to connect all vertices in a cycle. Since the number of vertices is 5000 for our implementation, we are left with 15000-5000 = 10000 edges which we need to randomly add to our connected graph in a way we avoid putting multiple edges for a set of vertices. To do so, we can iterate through the list of edges and check if it has already been added to the graph and add an edge only it it's not present. But this will consume more time in terms of time complexity. Hence, we can use unique key value property of hash set to allocate a unique value to any edge value been added to the graph. And add an edge only if its unique value does not match that of an edge that's already present in the graph, indicating multiple edges. Both ways of finding multiple edges has been implemented in the graph.

b) Generate Dense Graph such that each vertex is adjacent to about 20% of the other vertices that are randomly chosen:

Total number of vertices = 5000

To make a connected graph, we will call the function to connect all 5000 vertices in a cycle. To ensure that each vertex is adjacent to about 20% of the other vertices, we will call a probability function which randomly picks up a number from 1 to 100 and will add an edge between a given set of vertices if the following two conditions are followed:

1)The selected number value is less than 20
2)The given edge is not already present in the connected graph by checking its unique value from hash set or checking its presence in edge List.

Graphs are represented in adjacency list format in the above-mentioned implementations. Since my graph is undirected, whenever I will add an edge between i and j vertices, i will add the same between j and i in the adjacency list.

3. HEAP STRUCTURE

I have implemented a max heap data structure in which the value in each internal node is greater than or equal to the values in the children of that node.

I) **Max heap data structure** is used to find the best fringe having maximum value in modified Dijkstra's algorithm implementation to find the maximum bandwidth path in the generated graph. Thus, as mentioned in the project description, the Maxheap class implements subroutines for MAXIMUM, INSERT and DELETE of nodes in a heap.
Also,

- The vertices of a graph are named by integers 0, 1, ..., 4999;
- The heap is given by an array H [5000], where each element H[i] gives the name of a vertex in the graph;
- The vertex "values" are given in another array D [5000]. Thus, to find the value of a vertex H[i] in the heap, we can use D[H[i]].

```
Algorithm:  Maximum (H [1: : : max]; n) of Max-Heap
1: return (H [1]);
```

Figure 5. MAXIMUM MAXHEAP

```
Algorithm 2 Insert (H [1: : max]; n; a)
1: n = n + 1; H[n] = a; i = n;
2: while H[⌊i/2⌋] < H[i] and i >= 2 do
3:    swap H[⌊i/2⌋] with H[i];
4:    i = ⌊i/2⌋;
5: end while
```

Figure 6. INSERT MAXHEAP (1)

```
Algorithm: INSERT (u, weight) Max-Heap
1:    H. heap_size = H.heap_size + 1
3:    u = H [heap size – 1]
4:    D[u] = weight;
5:    Max Heapfy (H. heapsize – 1, weight).
```

Figure 7. INSERT MAXHEAP (2)

```
Algorithm: Max Heapfy (u, weight)
1: While (u > 0) do
2:     if parent (u) >= weight
3:         return;
6:     else
7:       swap H[u] with H[parent(u)]
8:     u = parent(u);
```

Figure 8. MAXHEAPFY MAXHEAP

```
Algorithm: Delete (H[1 : : : max]; n; i) Max Heap
 1: H[i] = H[n];     n = n - 1;
 2: if H[i] > H[⌊i/2⌋]  then
 3:    while H[⌊i/2⌋] < H[i] do
 4:       swap H[i] with H[⌊i/2⌋];
 5:       i = ⌊i/2⌋;
 6:    end while
 7: else
 8:    while (H[i] < H[2i] or H[i] < H [2i + 1]) do
 9:       if (2i = n) or (H[2i] > H [2i + 1]) then
10:          swap H[i] and H[2i]; i = 2i;
11:       else
12:          swap H[i] and H [2i + 1]; i = 2i + 1;
13:       end if
14:    end while
15: end if
```

Figure 9. DELETE MAXHEAP

II) **Heap Sort:** Heaps can be used in sorting an array. In max-heaps, maximum element will always be at the root. Heap Sort uses this property of heap to sort the array. Heap Sort is required while implementing Kruskal's algorithm for finding the maximum spanning tree in our generated graph. Root element in the heap will contain the maximum value element. Heap sort works by swapping this element with the last element in the heap and heapifying the max heap excluding the last element which is already in its correct position and then decreasing the length of heap by one. Recursively repeating this process until all elements are in their correct position gives us a list of sorted edges in decreasing order of their edge weight values.

```
Algorithm: HEAP SORT (H)
 1: While (i = size/2 and i>=0) do
 2:          Swap H[1] with H[i].
 5:          H.size = H.size - 1;
 6:          Max Heapfy(H,i).
```

Figure 10. HEAP SORT

4. ROUTING ALGORITHMS

A) Modified Dijkstra's Algorithm

Dijkstra's algorithm for shortest path is a greedy algorithm where for a given graph and a source vertex in the graph, the algorithm finds the shortest paths from source to all vertices in the given graph. This algorithm is modified to find maximum bandwidth path for our implementation where instead of updating each vertex with the shortest distance from source, we find the minimum of maximum weights between 2 vertices which gives the maximum bandwidth. Modified Dijkstra's Algorithm has been implemented using two approaches:

1)Dijkstra without using heap data structure

This implementation uses Array list to store fringe vertices and to find the best fringe value having maximum value Collections.max () of array list is called. Rest of the implementation is executed according the Algorithm as discussed in class.

2)Dijkstra using heap data structure

A maximum Heap data structure is used to store and retrieve fringe values. Since the root of the heap gives the vertex with the maximum edge weight value, H [1] (first element of heap) will give the best fringe value.

B) Modified Kruskal's Algorithm

Kruskal's minimum-spanning tree algorithm is modified to find maximum spanning tree, DFS on which gives us the maximum bandwidth path of the connected undirected graph. For maximum spanning tree we first sort the edge weights in non-increasing order using Heapsort implemented in Maxheap class. This helps in removing an edge with maximum weight from edge set in linear time. We then use a disjoint-set data structure to keep track of which vertices are in which components. The disjoint data structure consists of Make Set, Find and Union operation to keep track of elements partitioned into number of disjoint subsets.

MakeSet

The Make Set operation makes a new set by creating a new element with a unique id, a rank of 0, and a parent pointer to itself. The parent pointer to itself indicates that the element is the representative member of its own set.

The Make Set operation has O (1) time complexity, so initializing n sets has O(n) time complexity.

```
Algorithm: MAKE SET (v)
1:    Dad[v] = v;
2:    Rank[v] = 0;
```

Figure 11. MAKESET

Find

Find(x) follows the chain of parent pointers from *x* up the tree until it reaches a root element, whose parent is itself. This root element is the representative member of the set to which *x* belongs and may be *x* itself. *Path compression* flattens the structure of the tree by making every node point to the root whenever *Find* is used on it. This is valid, since each element visited on the way to a root is part of the same set. The resulting flatter tree speeds up future operations not only on these elements, but also on those referencing them.

```
Algorithm: FIND (v)
1:    int w = v.
2:    let q be a queue.
3:    while (Dad[w] ≠ w) do
4:        q.add(w)
5:        w = Dad[w].
6:    while (q ≠ Ø) do
7:        y = q. remove ();
8:        Dad[y] = w.
9:        Return w
```

Figure 12. FIND

Union

Union (x, y) uses *Find* to determine the roots of the trees *x* and *y* belong to. If the roots are distinct, the trees are combined by attaching the root of one to the root of the other. To implement *union by rank*, each element is associated with a rank. Initially a set has one element and a rank of zero. If two sets are unioned and have the same rank, the resulting set's rank is one larger; otherwise, if two sets are unioned and have different ranks, the resulting set's rank is the larger of the two.

```
Algorithm: UNION (r1, r2)
1: R1, R2 are parent Nodes:
2:     if Rank[r1] > Rank[r2] then
3:         Dad[r2] = r1;
4:     else if Rank[r1] < Rank[r2] then
5:         Dad[r1] = r2;
6:     else
7:         Dad[r1] = r2;
8:         Rank[r2] + +;
```

Figure 12. UNION

5. RESULTS

The 3 routing algorithms have been tested on 5 pairs of randomly generated graphs where source and destination vertex have also been randomly chosen. Following are the run time analysis and maximum bandwidth path and value found by the 3 algorithms for pairs of sparse and dense graphs.

SPARSE GRAPHS

1. Graph 1

Sr. No	Source Vertex	Destination Vertex	Modified Dijkstra's Algorithm without Heap	Modified Dijkstra's Algorithm with Heap	Modified Kruskal's Algorithm
1	979	990	Max BW: 1130	Max BW: 1130	Max BW: 1130
			Time: 0.182 secs	Time: 0.016 secs	Time: 0.034 secs
2	2987	4460	Max BW: 971	Max BW: 971	Max BW: 971
			Time: 0.194 secs	Time: 0.005 secs	Time: 0.018 secs
3	390	2750	Max BW: 1088	Max BW: 1088	Max BW: 1088
			Time: 0.013 secs	Time: 0.008 secs	Time: 0.133 secs
4	1479	672	Max BW: 551	Max BW: 551	Max BW: 551
			Time: 0.049 secs	Time: 0.003 secs	Time:0.391secs
5	4735	2629	Max BW: 1229	Max BW: 1229	Max BW: 1229

			Time: 0.01 secs	Time: 0.01 secs	Time: 0.01secs

2. Graph 2

Sr. No	Source Vertex	Destination Vertex	Modified Dijkstra's Algorithm without Heap	Modified Dijkstra's Algorithm with Heap	Modified Kruskal's Algorithm
1	114	4540	Max BW: 1015	Max BW: 1015	Max BW: 1015
			Time: 0.002 secs	Time: 0.017 secs	Time: 0.019 secs
2	2853	4609	Max BW: 931	Max BW: 931	Max BW: 931
			Time: 0.118 secs	Time: 0.185 secs	Time: 0.009 secs
3	2262	590	Max BW: 1148	Max BW: 1148	Max BW: 1148
			Time: 0.002 secs	Time: 0.001 secs	Time: 0.008 secs
4	2634	3348	Max BW: 903	Max BW: 903	Max BW: 903
			Time: 0.007 secs	Time: 0.07 secs	Time: 0.017 secs
5	3258	4705	Max BW: 997	Max BW: 997	Max BW: 997
			Time: 0.031 secs	Time: 0.002 secs	Time:0.008 sec

3. Graph 3

Sr. No	Source Vertex	Destination Vertex	Modified Dijkstra's Algorithm without Heap	Modified Dijkstra's Algorithm with Heap	Modified Kruskal's Algorithm
1	1477	4617	Max BW: 1231	Max BW: 1231	Max BW: 1231
			Time: 0.033 secs	Time: 0.001 secs	Time: 0.009 sec
2	4626	3032	Max BW: 1018	Max BW: 1018	Max BW: 1018
			Time: 0.07 secs	Time:0.001 secs	Time:0.425 sec
3	2069	2315	Max BW: 1156	Max BW: 1156	Max BW: 1156
			Time: 0.032 secs	Time: 0.004 secs	Time: 0.014 sec
4	1783	1298	Max BW: 1155	Max BW: 1155	Max BW: 1155
			Time: 0.031 secs	Time: 0.002 secs	Time: 1.538 sec
5	2297	2329	Max BW: 1171	Max BW: 1171	Max BW: 1171
			Time: 0.074 secs	Time: 0.002 secs	Time: 0.007 sec

4. Graph 4

Sr. No	Source Vertex	Destination Vertex	Modified Dijkstra's Algorithm without Heap	Modified Dijkstra's Algorithm with Heap	Modified Kruskal's Algorithm
1	16	1295	Max BW: 1143	Max BW: 1143	Max BW: 1143
			Time: 0.023 secs	Time: 0.001 secs	Time: 0.006sec
2	4646	4128	Max BW: 1079	Max BW: 1079	Max BW: 1079
			Time: 0.051 secs	Time: 0.002 secs	Time: 0.007sec
3	2261	1598	Max BW: 1164	Max BW: 1164	Max BW: 1164
			Time: 0.019 secs	Time: 0.002 secs	Time: 0.227sec
4	918	1464	Max BW: 1248	Max BW: 1248	Max BW: 1248
			Time: 0.002 secs	Time: 0.001 secs	Time: 0.312sec
5	2762	1179	Max BW: 1221	Max BW: 1221	Max BW: 1221
			Time: 0.044 secs	Time: 0.002 secs	Time: 0.011sec

5. Graph 5

Sr. No	Source Vertex	Destination Vertex	Modified Dijkstra's Algorithm without Heap	Modified Dijkstra's Algorithm with Heap	Modified Kruskal's Algorithm
1	1830	2546	Max BW: 1088	Max BW: 1088	Max BW: 1088
			Time: 0.032 secs	Time: 0.002 secs	Time: 0.006 sec
2	3233	4747	Max BW: 1131	Max BW: 1131	Max BW: 1131
			Time: 0.008 secs	Time:0.002 secs	Time: 0.023 sec
3	3799	763	Max BW: 1250	Max BW: 1250	Max BW: 1250
			Time: 0.002secs	Time: 0.001 secs	Time: 0.005 sec
4	3230	3674	Max BW: 1093	Max BW: 1093	Max BW: 1093
			Time: 0.035 secs	Time: 0.001 secs	Time:0.055 secs
5	1482	887	Max BW: 1137	Max BW: 1137	Max BW: 1137
			Time: 0.01 secs	Time: 0.318 secs	Time: 0.011 sec

DENSE GRAPHS

1. Graph 1

Sr. No	Source Vertex	Destination Vertex	Modified Dijkstra's Algorithm without Heap	Modified Dijkstra's Algorithm with Heap	Modified Kruskal's Algorithm
1	576	3863	Max BW: 1531	Max BW: 1531	Max BW: 1531
			Time: 0.301 secs	Time: 0.135 secs	Time: 4.149 sec
2	1337	3556	Max BW: 1533	Max BW: 1533	Max BW: 1533
			Time: 0.142 secs	Time: 0.043 secs	Time: 4.913 secs
3	4382	329	Max BW: 1532	Max BW: 1532	Max BW: 1532
			Time: 0.201 secs	Time: 0.054 secs	Time: 3.884 secs
4	1527	1783	Max BW: 1539	Max BW: 1539	Max BW: 1539
			Time: 0.198 secs	Time: 0.048 secs	Time: 4.811 secs
5	2054	4798	Max BW: 1535	Max BW: 1535	Max BW: 1535'
			Time: 0.107 secs	Time: 0.007 secs	Time: 3.968 secs

2 Graph 2

Sr. No	Source Vertex	Destination Vertex	Modified Dijkstra's Algorithm without Heap	Modified Dijkstra's Algorithm with Heap	Modified Kruskal's Algorithm
1	1732	280	Max BW: 1522	Max BW: 1522	Max BW: 1522
			Time: 0.137 secs	Time: 0.033 secs	Time: 4.147 secs
2	4752	3528	Max BW: 1524	Max BW: 1524	Max BW: 1524
			Time: 0.023 secs	Time: 0.037 secs	Time: 3.931 secs
3	4656	611	Max BW: 1524	Max BW: 1524	Max BW: 1524
			Time: 0.166 secs	Time: 0.034 secs	Time: 3.838 secs
4	3134	3959	Max BW: 1525	Max BW: 1525	Max BW: 1525
			Time: 0.141 secs	Time: 0.023 secs	Time: 3.908 secs
5	2334	3033	Max BW: 1523	Max BW: 1523	Max BW: 1523
			Time: 0.176 secs	Time: 0.031 secs	Time: 3.871 secs

2. Graph 3

Sr. No	Source Vertex	Destination Vertex	Modified Dijkstra's Algorithm without Heap	Modified Dijkstra's Algorithm with Heap	Modified Kruskal's Algorithm
1	2300	3068	Max BW: 1520	Max BW: 1520	Max BW: 1520
			Time: 0.045 secs	Time: 0.023 secs	Time: 4.155 secs
2	2271	4277	Max BW: 1516	Max BW: 1516	Max BW: 1516
			Time: 0.159 secs	Time:0.017 secs	Time:3.854 secs
3	3647	2561	Max BW: 1519	Max BW: 1519	Max BW: 1519
			Time: 0.046 secs	Time: 0.007 secs	Time: 3.955 secs
4	749	4603	Max BW: 1515	Max BW: 1515	Max BW: 1515
			Time: 0.128 secs	Time: 0.014 secs	Time: 3.943 secs
5	3460	4304	Max BW: 1519	Max BW: 1519	Max BW: 1519
			Time: 0.12 secs	Time: 0.008 secs	Time: 3.789 secs

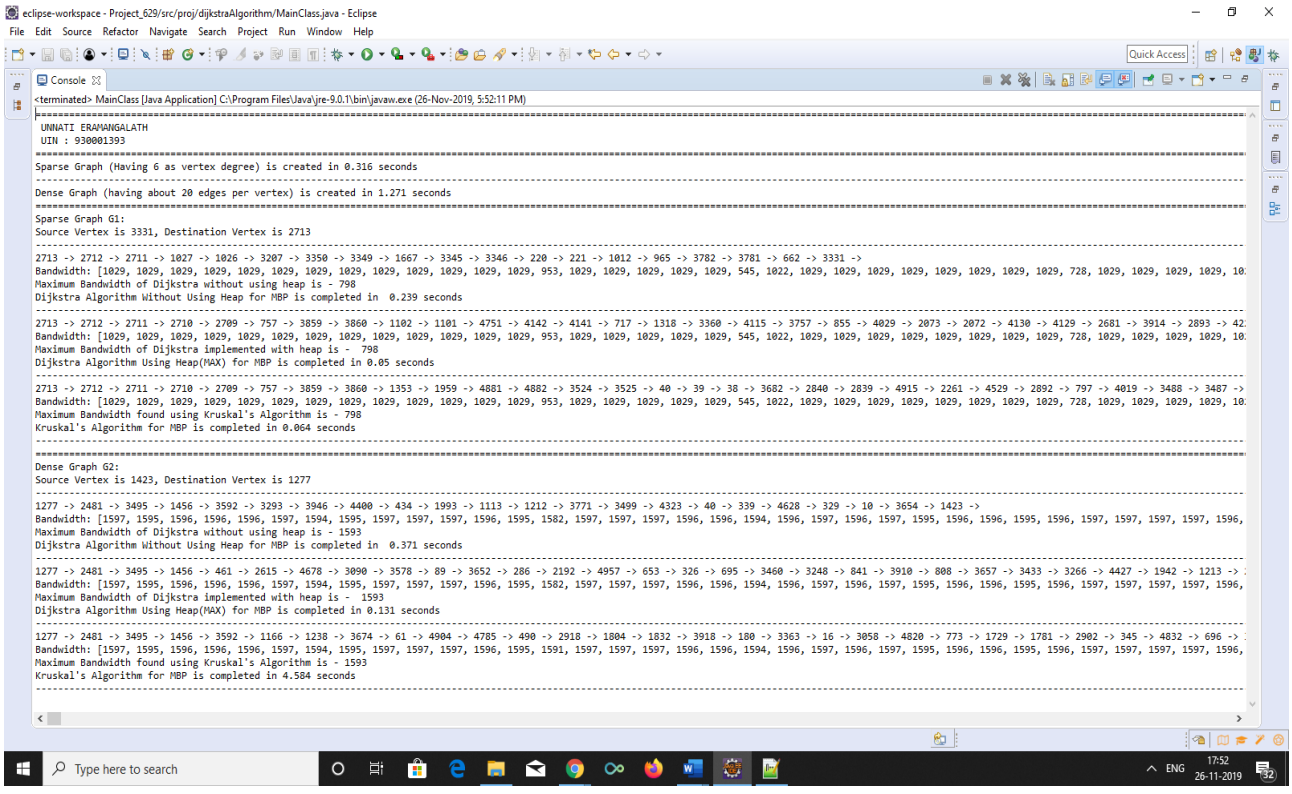
4. Graph 4

Sr. No	Source Vertex	Destination Vertex	Modified Dijkstra's Algorithm without Heap	Modified Dijkstra's Algorithm with Heap	Modified Kruskal's Algorithm
1	4739	3735	Max BW: 1505	Max BW: 1505	Max BW: 1505
			Time: 0.352 secs	Time: 0.012 secs	Time: 4.188 secs
2	3935	4128	Max BW: 1503	Max BW: 1503	Max BW: 1503
			Time: 0.166 secs	Time: 0.013 secs	Time: 4.069 secs
3	3731	4156	Max BW: 1502	Max BW: 1502	Max BW: 1502
			Time: 0.191 secs	Time: 0.075 secs	Time: 4.088 secs
4	4754	3110	Max BW: 1504	Max BW: 1504	Max BW: 1504
			Time: 0.217 secs	Time: 0.031 secs	Time: 4.127 secs
5	2601	1790	Max BW: 1505	Max BW: 1505	Max BW: 1505
			Time: 0.142 secs	Time: 0.011 secs	Time: 3.897 secs

5. Graph 5

Sr. No	Source Vertex	Destination Vertex	Modified Dijkstra's Algorithm without Heap	Modified Dijkstra's Algorithm with Heap	Modified Kruskal's Algorithm
1	3895	1597	Max BW: 1582	Max BW: 1582	Max BW: 1582
			Time: 0.186 secs	Time: 0.007 secs	Time: 4.105 secs
2	2701	4500	Max BW: 1581	Max BW: 1581	Max BW: 1581
			Time: 0.234 secs	Time:0.031 secs	Time: 4.137 secs
3	2742	4770	Max BW: 1579	Max BW: 1579	Max BW: 1579
			Time: 0.201 secs	Time: 0.053 secs	Time: 3.85 secs
4	748	4815	Max BW: 1581	Max BW: 1581	Max BW: 1581
			Time: 0.166 secs	Time: 0.014 secs	Time:3.857 secs
5	988	3257	Max BW: 1580	Max BW: 1580	Max BW: 1580
			Time: 0.212 secs	Time: 0.043 secs	Time: 3.908 secs

SAMPLE CONSOLE OUTPUT FOR A SINGLE PAIR OF VERTICES:



6. PERFORMANCE ANALYSIS

ALGORITHM	TIME COMPLEXITY
Dijkstra Without Heap	O(n^2)
Dijkstra With Heap	O((n+m) log n)
Kruskal's Algorithm	O (m log m)

Table. Time Complexity Analysis

Analysis of the performance output obtained by running the three implemented routing algorithms to find the maximum bandwidth path for a set of 5 randomly generated undirected sparse and dense graphs , we observe that in both types of graphs, performance of Dijkstra's with heap is better than the performance of Dijkstra's without heap implementation. This is because in Dijkstra's algorithm without heap implementation, finding the fringe vertex with maximum bandwidth among all fringe vertices takes time O(n), resulting in an overall quadratic time complexity of O(n^2). Since insertion and deletion operation in heap takes O(logn) time, overall time complexity of algorithm using heap comes to O((m+n) log n) which is better than that of without heap implementation.

From the time complexity analysis table, we observe that the running time for all algorithms except that of Dijkstra's without heap implementation, are proportional to the number of edges m. For Sparse graphs, since the total number of edges will be proportional to the number of vertices in the graph(m = 3*n), running time complexity for all three algorithms will be better in comparison to the performance of algorithms for a dense graph. This is because in a dense graph, total number of edges m amounts to O(n^2), significantly larger than sparse graph thus costing time for examination of a greater number of edges.

Sparse Graph Analysis:

1. Performance of Dijkstra's algorithm with heap is better than and much faster that Dijkstra's without heap implementation.
2. Performance of Kruskal's Algorithm is comparable to that of Dijkstra's algorithm with heap and better than that of Dijkstra's without heap implementation
3. These observations can be verified from the output table given in the above section.

Dense Graph Analysis:

1. Almost similar performance of Dijkstra's algorithm with heap and Dijkstra's algorithm without heap implementation with the one with heap performing slightly better.
2. Kruskal's has the highest running time making its performance the worst among all three.
3. These results can be verified from the data findings presented in the above section.

In our current implementation of Kruskal's Algorithm, we are creating a maximum spanning tree every time we want to find the maximum bandwidth path. Our current run time findings account for this time period as well. But in practical applications, we must create the maximum spanning tree only once per graph and find bandwidth in linear time using DFS/BFS. This will significantly improve the performance of Kruskal's algorithm making it comparable to that of Dijkstra's with heap implementation and better than without heap implementation.

7. CONCLUSION AND FUTURE SCOPE

The above findings showed that Kruskal's performance can be comparable to Dijkstra's with heap implantation. But they were derived following theory where we consider the worst case whereas in real-life its rarely the case.

Dijkstra's algorithm is preferred in a complex graph if we want to find the maximum bandwidth path between two specific vertices only once. Kruskal's Maximum Spanning tree algorithm is a better choice wherein we have to find the path for different pairs of vertices as once the spanning tree is constructed, bandwidth paths can be found in linear time using DFS.

The existing implementation can be further improved by reducing the data structures used to store information and trying to compute outcome dynamically. We may replace linear time DFS in Kruskal's algorithm to perform in constant time by storing additional information while constructing maximum spanning tree and fetching those to determine bandwidth path. We may also explore better algorithms for network optimization and try implementing them in future to further obtain a better performance in terms of rum time and efficiency.