INDEX

1. INTRODUCTION

Through this project, I have tried to implement a network routing protocol using data structures and algorithms. Working on the problem of the Maximum-Bandwidth-Path Problem and having implemented three algorithms, namely:

1. Dijkstra's Algorithm
2. Dijkstra's Algorithm using a heap structure for fringes
3. Modified Kruskal's Algorithm

The report discusses the implementation and the results of the implementation of these three algorithms on two types of graphs:

1. Sparse Graph – Graph with 5000 vertices. The average vertex degree is 8. The graph edges have been assigned random positive weights.
2. Dense Graph – Graph with 5000 vertices. Each vertex is adjacent to about 20% of the other vertices, which are randomly chosen, which means that each vertex is connected to an approximate 1000 other vertices. The edges have been assigned random positive weights.

In the process of the implementation, I created various other data structures, such as:

1. Max – Heap – needed for modified Dijkstra's Algorithm.
2. Set (Union – Find) – needed for modified Kruskal's Algorithm.

Next, I would like to discuss the implementation details and the results.

2. IMPLEMENTATION DETAILS

I have implemented the project in the Python Programming language. The graph is stored in the Dictionary Data Structure, with a list of lists as the entries. A list is very similar to an array in other programming languages.

Making sure that the graph is connected

The problem with generating a random graph is that we might get a case where the graph is not connected and then generating random 's' and the 't', i.e, the source and the target vertices, might be in different components of the graph and the algorithms will not work. Hence, in order to avoid that, I first made a cycle connecting all the 5000 vertices with random positive edge weights and then added the edges according to the type of the graph – Sparse or Dense.

1. Dijkstra's Algorithm

The Dijkstra's Algorithm is modified to solve the Max-Bandwidth-Path Problem and not the shortest path problem that it normally does.

Dijkstra's Algorithm is a greedy algorithm, in which, at every step it selects the path with the maximum Bandwidth. The selection of the maximum edge is done in Linear Time - O(n) time in a simple Dijkstra's Algorithm. This is then done for each Vertex in the loop, until the target vertex is found and then the lop terminates.

The class Graph(object) is the main class that defines the graph. The constructor of the class defines and stores a Graph for every object of the class.

The Class Function addEdge( ) takes in the arguments of the starting and the ending vertex for an edge and the weight of the edge and makes an entry in the dictionary corresponding to the edge. **A point to note is here that doing so results in the representation of the graph as an Adjacency List.**

The function Dijsktra( ) is the main function which has the modified Dijkstra's algorithm. The algorithm initializes the status of each vertex as 'unseen', the weight of each vertex is initialized with 'negative- infinity' and the dad (parent) of each vertex is initialized with 'None'.
The status of the starting vertex – 's' is initialized to 'in-tree', meaning that the vertex has been processed. Now going through the adjacency list of the vertex – 's', each of its adjacent vertices are given the status of 'fringes'. Now, a 'While loop' envelopes all this and goes till the ending vertex – 't' is processed, that is, it attains the status of 'in-tree'. The Max-Bandwidth-Path Problem is solved by checking whether the vertex is a 'fringe' or 'unseen' and then **relaxing** the edge accordingly, which essentially means that the Bandwidth of end of the edge is compared with the minimum of the Bandwidth of the stating edge and the edge weight and on finding if the minimum of the two is smaller, it is set as the Bandwidth of the end vertex of the edge.

The results can be seen in the lists – Dad[ ] – which holds the parent of each vertex and the Max-Bandwidth-Path can be traced back from the Dad[ ] array for each vertex.
The status of each vertex after the termination of the algorithm can be seen by the list – 'status'.

The results can be seen in the corresponding graphs and the table given at the end of the report. They present a very interesting find. The algorithm is then run on two types of graphs:

- Sparse Graph – with an average vertex degree of 8.
- Dense Graph – with each vertex being adjacent to about 20% of the other vertices.

Both the graphs have 5000 vertices.

2. Dijkstra's Algorithm with a Max-Heap

In this algorithm, the Dijkstra's Algorithm is modified to include a Max-Heap Data Structure. It is essentially done so as to satisfy the 'greedy' nature of the algorithm.
As we know, Dijkstra's Algorithm is a greedy algorithm, wherein, at each step, it selects the vertex with the maximum Bandwidth and keeps on doing that until the target vertex is processed.
Now, in a simple Dijkstra's Algorithm, the maximum Bandwidth edge is found in a Linear Time, that is O(n). However, if we use the Max-Heap Data Structure, the time can be reduced to a logarithmic time – O(logn). This reduces the overall time complexity and the algorithm can become quicker.

In the algorithm, once again, like in the normal Dijkstra's Algorithm, the class Graph(object) stores the graph as a Dictionary Data Structure. Doing this, the graph is represented as an Adjacency List. The function dijkstra( ) is the function that implements the algorithm. In each iteration, it finds the max Bandwidth path with the help of a Max-heap. Max-Heap is a data structure, in which the maximum element is found at a constant time – O(1). However, to heapify it, the time taken is of the order of the Logarithmic – O(logn). Heapifying the heap, means to maintain the property of a Heap – which is maintaining the parent has to be bigger than both of its children. The maximum Bandwith 'fringe' is extracted and taken out of the Max-heap, and the in the process of heapifying it, the next largest element comes at the top of the heap and the process is repeated. The process of heapifying a Max-heap takes Logarithmic time – O(logn). The rest of the algorithm is same as the normal Dijkstra – explained above. The list - dad[ ] stores the parent of the vertex, that is the path from where the vertex has come from. Hence, to find the path 's-t' path, the path can be traced back with the help of the dad[ ].
The Max-heap has been implemented with the help of Lists, wherein, the children of the parent at the i^{th} position can be found at $2i + 1^{st}$ and $2i + 2^{th}$ position. The function maxHeapify( ) maintain the Heap property of the Max-heap after every deletion, or addition. An element is

deleted from the Max-Heap in the extractMax( ) function, which gives the maximum element from the Heap – which is the top most element and then maintains the heap structure by calling the maxHeapify( ) function.

Again the algorithm is run on two types of graphs – Dense and Sparse.

3. **Kruskal's Algorithm**

Kruskal's Algorithm is entirely different from the Dijkstra's Algorithm, in which a Maximum Spanning Tree is constructed. As shown in the class, any path on the Maximum Spanning Tree is the Maximum Bandwidth Path and hence, a simple Depth First Search (DFS) is done to find the Maximum Bandwidth Path from 's – t'. The Kruskal's Algorithm also helps us to find a negative weight cycle. However, since I took only positive edge weights, such a scenario will not arise in the random graphs that I generate. The majority of the time of the algorithm goes in sorting the edges in a non-increasing order. Next, the edge with the highest edge weight is taken and put in the Maximum Spanning Tree, if connecting it doesn't make a cycle. Cycle is detected with the help of another Data Structure called Union-Find, wherein, an edge is rejected if the parents of both of its edges are in the same set, that is, is the same. The edge is kept in the Maximum Spanning Tree, if the putting it does not form a cycle. The Maximum Spanning Tree is stored in the treee[ ] list. Now, after forming the tree, to find the 's-t' path, a simple DFS is done starting from 's', till 't' is found.

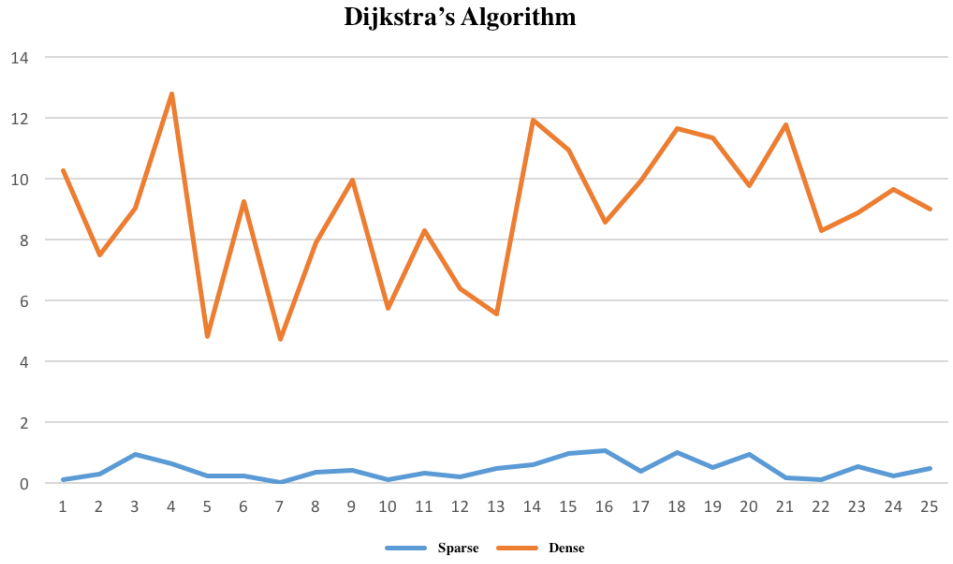
Once again, the algorithm is run on two graphs – Sparse and Dense, each having 5000 vertices.

3. PERFORMANCE OF VARIOUS ALGORITHMS

Having run all the three algorithms on both the types of graphs – Sparse and Dense, I plotted diagrams to understand the performance better.

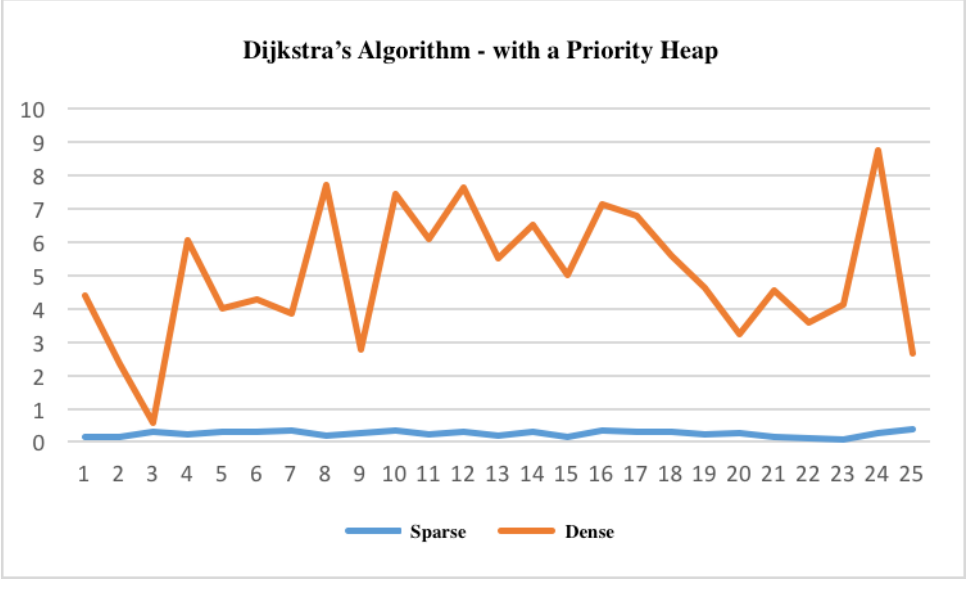
1. **Dijkstra's Algorithm**

Following is a graph comparing the running times of the Dijkstra's Algorithm on both the dense and the sparse graph. The lower line represents the Sparse Graph and the upper line corresponds to the Dense Graph. The algorithm was run on 5 randomly generated pair of graphs, with atleast 5 pairs of randomly generated source-destination vertices. As expected, the time taken for the dense graph is significantly larger than the sparse graph. **One constraint that I can conclude from this analysis is not computational but that of the Programming Language used. Python gets significantly slower for dense graphs.**



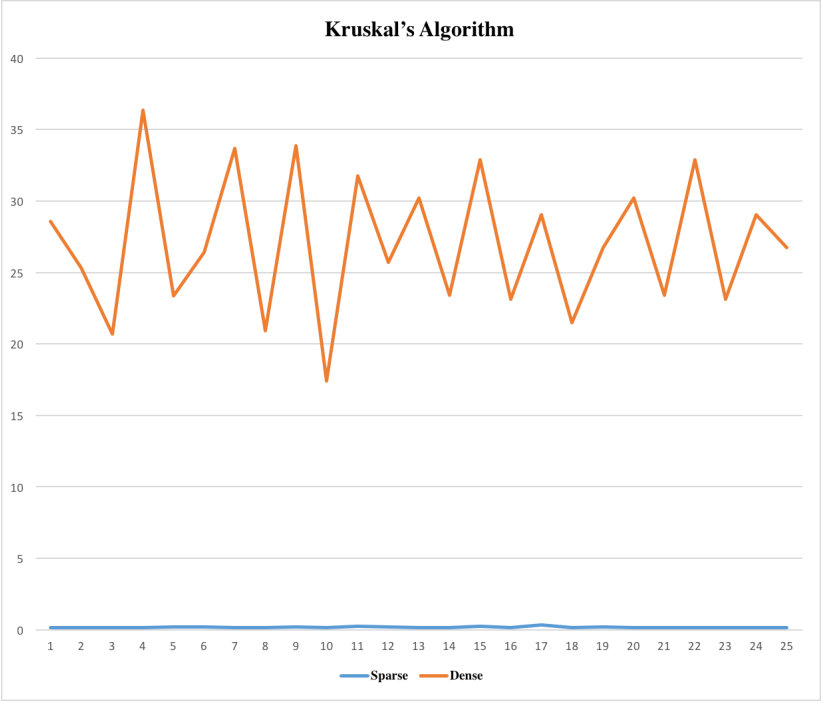
2. Dijkstra's Algorithm with a Max-Heap

Following is the plot for the dense and the sparse graphs. In this also, as expected the time for the dense graph is higher than the sparse one. In the graph – the Priority Heap is the Max Heap – I have written the name Priority Heap because the Priority Queue is implemented with the help of a Max-Heap! Now, what can be seen is that the difference between the times for the dense and the sparse graphs is lower than that between the dense and the sparse of the normal Dijkstra's Algorithm. This is because of the O(logn) factor which reduces the difference between the dense and the sparse graphs.



3. Kruskal's Algorithm

The graph given below is that for the Kruskal's Algorithm with the Union-Find Data Structure.

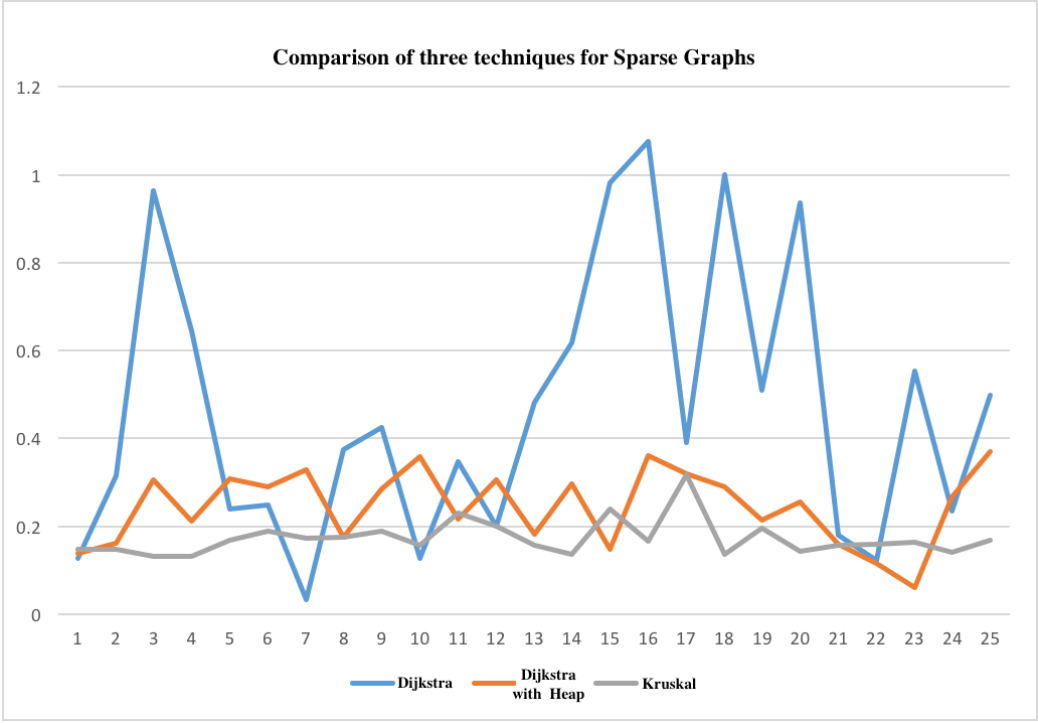


As can be seen, the difference between the Sparse and the Dense graphs on the Kruskal's Algorithm in the highest amongst the three algorithms! This is primarily because the time taken to sort the edges increases significantly with a dense graph, since the number of edges to sort is approximately 100 times more! This shows that for dense graphs, Kruskal's Algorithm given the maximum time.

Comparison of Performance of the three Algorithms for Sparse graphs

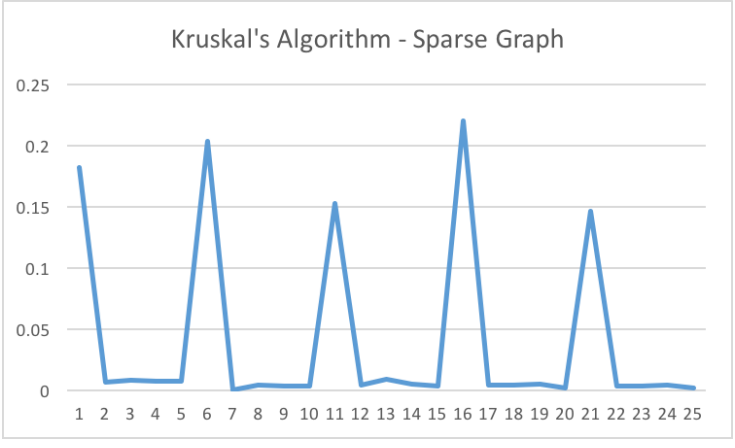
Following is the plot wherein, the time for all three algorithms is shown on the same axes. Suprisingly, Kruskal's algorithm wins the race with the least time amongst the three. What is more surprising is that, the time taken for Kruskal's Algorithm is lower than even the modified Dijkstra's – that is Dijkstra's Algorithm with a Max-Heap Structure. Other than that, the results are as expected, the normal Dijkstra's time is way higher than that of the Modified Dijkstra's Algorithm, because the normal Dijkstra's time is expected to be $O(n^2)$, whereas the time for the Modified Dijkstra's is O((n+m)logn). Now, since, in this scenario, m << n; the time for the Modified Dijkstra's is better than the normal Dijkstra's algorithm. Now, comparing the performances of the normal Dijkstra's algorithm and the Kruskal's Algorithm, both are of the same time complexity in worst case scenarios – O(mlogn), when simplified. However, Kruskal getting a better time means that the constant (c_1) of the Kruskal's Algorithm is better than the constant (c_2) of the Dijsktra's Algorithm and hence the Kruskal's Algorithm is faster.

Another advantage of the Kruskal's Algorithm is that once the Maximum Spanning tree is made, finding any 's-t' path is just a simple DFS and hence, takes much much lower time than any other algorithm (as shown later in the analysis).



Special Mention for Kruskal's Algorithm

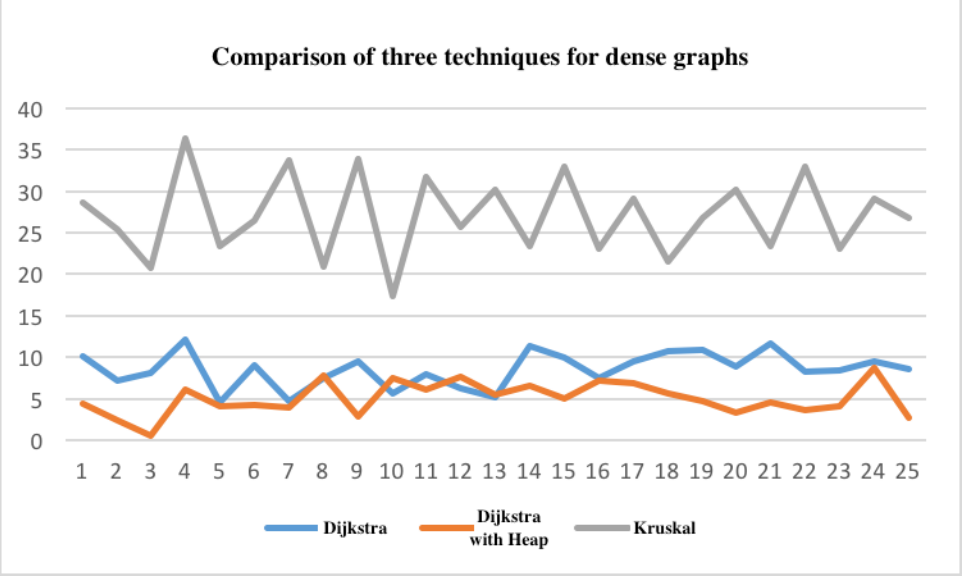
As mentioned earlier, the Kruskal's Algorithm is much much faster when the tree is already constructed. Hence, in a case when the graph is constant, finding the Max-Bandwidth-Path for a random set of 's-t' pairs is significantly higher by the Kruskal's Algorithm, if we store the Maximum Spanning Tree generated by the first iteration. The following plot shows the running times for the Sparse Graph. The peaks are the iterations where a new graph was taken – 5 times and the following 5 iterations show the time for the corresponding sparse graph, in which the Maximum Spanning Tree was already generated**. The amortized running time is way lower than the other algorithms.**



Comparison of Performance of the three Algorithms for Dense graphs

As can be seen in the following plot, which represents all the three algorithm's running time on a single plot. Surprisingly, Kruskal's Algorithm is way behind the other two. The modified Dijkstra's Algorithm – that is with a Max-heap is the clear winner. One observation that can be made is that the difference between the Modified Dijkstra and the normal Dijkstra has reduced! This was expected as 'm' has now increased and the time – O((m+n)logn) for the Modified Dijkstra has become larger; still it has not reached $O(n^2)$ required for the normal Dijkstra.

The second observation is that the Kruskal's Algorithm is far behind the Modified Dijkstra's Algorithm. Having the same time constant – O((m+n)logn), it still performs way slower than the modified Dijkstra's Algorithm. This shows the importance of the Constant factors in the running times! A point to be noted is that, in the plot I ensured that the Kruskal's Algorithm had to generate the Maximum Spanning Tree each time.
However, the beauty of the Kruskal's Algorithm is where the graph is constant, as shown in the next section.



Special Mention for Kruskal's Algorithm

As can be seen from the plot below, the running time for the first iteration is much slower than the other algorithms – which is represented by the peaks for the 3rd curve. However, following it, the time becomes faster significantly, only because the Maximum Spanning Tree has been generated in the first iteration, and the corresponding iterations are using the Spanning Tree already generated and saving significant running time. **Hence, similar to the Sparse Graphs, if the graph to be used is the same, then using Kruskal's Algorithm is much beneficial as the amortized time for all the runs will be much lower than the other two algorithms.** The time is so low that it comes as almost zero – a straight line in the graph.

However, the values of the running time can be seen in the attached table (Table – 1) at the end of the report in the last column of the table. It is truly eye opening how wide the difference is and how much the network can be optimized if Kruskal's Algorithm is used in case where the network is relatively non-changing.

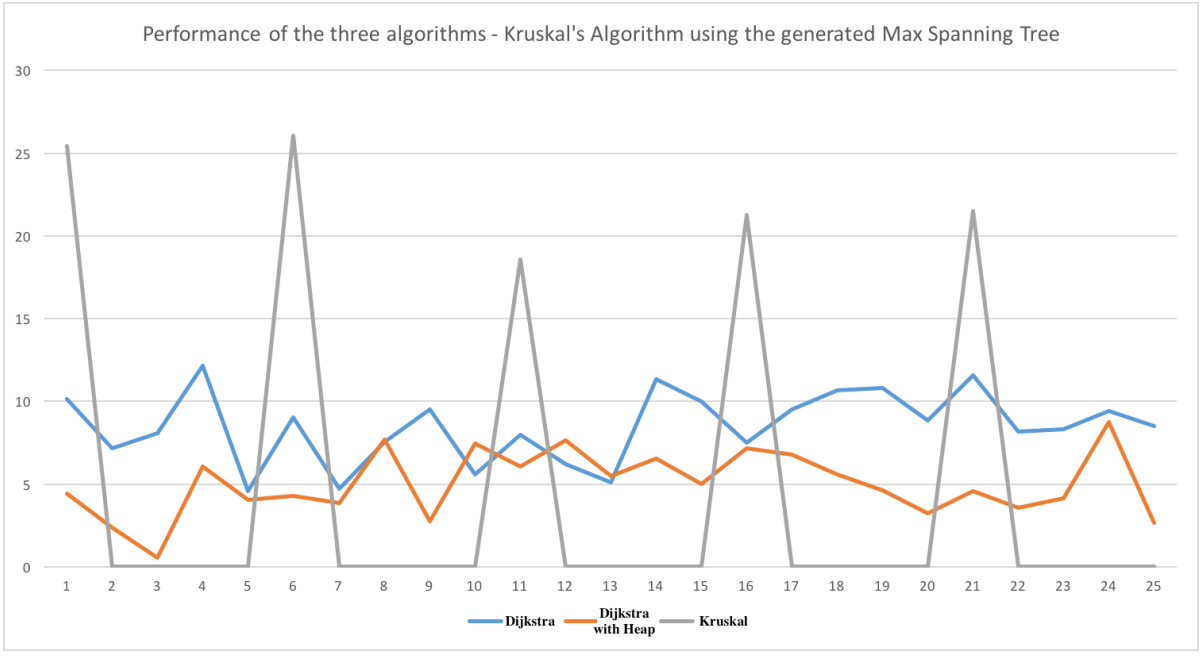


TABLE 1
(The bolded entries represent the First iterations of a graph, where the Maximum Spanning Tree was generated by the algorithm)

Kruskal's Algorithm – Sparse Graph (using the Spanning Tree generated in First Iteration)	Kruskal's Algorithm – Dense Graph (using the Spanning Tree generated in First Iteration)
0.181782	**25.435056**
0.006377	0.001699
0.00831	0.000438
0.007349	0.000714
0.007453	0.005739
0.203029	**26.04341**
0.000399	0.002334
0.003674	0.00248
0.003106	0.002877
0.003586	0.003507
0.152294	**18.582473**
0.004117	0.003723
0.008629	0.00364
0.004576	0.003664
0.003172	0.007392
0.219742	**21.270537**
0.004333	0.00532
0.003625	0.001452
0.004729	0.007513
0.001351	0.005508
0.146448	**21.513094**
0.003403	0.000579
0.003403	0.003292
0.004264	0.003233
0.001627	0.003364

TABLE 2
(1 – Sparse Graph, 2 – Dense Graph)
(Kruskal's Algorithm generating a Maximum Spanning Tree in every run in the random graph with random 's-t' in every iteration)

Graph Type	Dijkstra	Dijkstra-with-Heap	Kruskal
1	0.126711	0.137874	0.148327
2	10.128474	4.413414	28.581443
1	0.3149	0.160413	0.147102
2	7.190446	2.355545	25.344326
1	0.964097	0.305168	0.131429
2	8.081383	2.56958	20.698245
1	0.645153	0.21097	0.131788
2	12.153441	6.070027	36.360254
1	0.238929	0.3069	0.168192
2	4.581934	4.027825	23.349451
1	0.248535	0.288968	0.188269
2	9.016674	4.275293	26.394127
1	0.033487	0.328653	0.173541
2	4.698974	3.86511	33.69783
1	0.373937	0.175652	0.174851
2	7.526553	7.71403	20.935806
1	0.42568	0.285649	0.188051
2	9.522287	2.763204	33.875502
1	0.126847	0.357773	0.156903
2	5.606248	7.43764	17.410889
1	0.347152	0.21714	0.230981
2	7.96197	6.08343	31.746961
1	0.200756	0.305385	0.200238
2	6.186382	7.626309	25.712576
1	0.482942	0.182986	0.1558
2	5.088158	5.50133	30.220731
1	0.617245	0.295698	0.135395
2	11.326372	6.532296	23.396408
1	0.981107	0.14643	0.240173
2	9.975483	5.019146	32.889171
1	1.074954	0.360925	0.166301
2	7.516252	7.155485	23.115789
1	0.391368	0.31884	0.317447
2	9.527764	6.799172	29.018586
1	0.999639	0.288598	0.135858
2	10.643255	5.601032	21.501827
1	0.509809	0.212975	0.195422
2	10.821185	4.624906	26.735073
1	0.936457	0.254686	0.143515
2	8.833793	3.238124	30.220731
1	0.179996	0.158059	0.15588
2	11.582677	4.555545	23.396408
1	0.12277	0.115302	0.158161
2	8.191774	3.592672	32.889171
1	0.553813	0.060333	0.162593
2	8.327535	4.129657	23.115789
1	0.235462	0.266667	0.139557
2	9.410999	8.76216	29.018586
1	0.497875	0.368953	0.168939
2	8.524926	2.660611	26.735073