Minimum Spanning Tree

D Jayme Green

May 13, 2016

CSCI 261
Analysis of Algorithms
Section: 2

Contents

1 Overview

The Minimum Spanning Tree project, written in Java 1.7, analyses the different ways to implement a minimum spanning tree on a random, connected, acyclic graph. The graph is represented as an adjacency list and adjacency matrix with both Prim's algorithm and Kruskal's algorithm (using insertion, count, and quick sort). The algorithms are tested with different types of graphs with varying nodes and density of edges.

All tests are done on the below computer with no other processes being done at the same time (besides Windows 10 and other background processes):

Processor: Intel Core i7 3770k @ 3.50GHz

Motherboard: Maximus V Extreme

Memory: DDR3 16GB

While the times for all processes will be affected by this computer's specifications, the differences between each algorithm will still be evident and will be prevalent on all other computers.

Tests were designed to be varied with large dense graphs and small sparse graphs to see which algorithm with which graph representation performs the best. For large dense graphs and large sparse graphs, it was discovered that Kruskal's algorithm using count sort and an adjacency matrix was best. For medium-sized half-edges graphs, Prim's algorithm using an adjacency list becomes the fastest. For small sparse graphs and for small dense graphs, any algorithm and any implementation becomes ideal as modern computers have made all of these algorithms virtually done in 0 milliseconds.

This report will further explain how the tests were formulated and analysis of the results.

1.1 Definitions

Minimum Spanning Tree (MST)
: A sub-graph in the graph given which has the minimum amount of weights on edges that connects all the nodes together. Can be resolved by using one of two algorithms: Kruskal's algorithm and Prim's algorithm.

Kruskal's Algorithm
: One of the algorithms that creates a minimum spanning tree. It involves sorting all edges from least to greatest by weight and then selecting the smallest edge that does not create a cycle until every node is reached.

Prim's Algorithm
: One of the algorithms that creates a minimum spanning tree. It involves selecting an arbitrary node and putting it into the minimum spanning tree. Then, select the lowest edge from the minimum spanning tree to a node outside of the MST and add that node. This process continues until no more nodes are outside the MST.

Adjacency Matrix

A 2-D matrix with the rows and column numbers representing the node number and the meeting point of the row, column being the edge weight between the two nodes. If the weight is represented as 0 in the matrix, there is no edge between the two nodes. This is one way to represent a graph's connections.

Adjacency List

A list within a list which can represent a graph's connections. The first list's indexes correspond with each node in the graph. The list within each index represents the connections that the node has with the index node. The weights of these connections are further described inside the edges which can be obtained using the two nodes.

2 Experimental Design and Input

Two variations in data will influence the graph's performance: amount of nodes and the percentage of edges. With these variations, nine categories emerge of the different types of graphs: small-sparse graph, small-medium graph, small-dense graph, medium-sparse graph, medium-medium graph, medium-dense graph, large-sparse graph, large-medium graph, and large-dense graph.

All these nine categories are tested with three different random seeds which are the same throughout the categories. These seeds are 100,000, 10,000, and 987,654. While the seeds themselves are arbitrary, the use of three separate seeds prevents the random number generator from impacting the results with "unlucky" number generation. These seeds are also used by all categories making the random number generation the same throughout the categories resulting in the algorithms being the true difference in times.

All of the definitions are also standard. A small graph is always 7 nodes. A medium graph is 100 nodes while a large graph is 1,000 nodes (while trying to test higher quantity of nodes, there was not enough heap space on the testing computer).

The percentage of edges is also defined and standardized. All graphs are guaranteed to be connected, but the probability of edges influences additional edges between nodes. Sparse graphs have a probability of edges of 0.2 percent except for the small case which has 0.4 percent due to the time it took to produce a connected graph. Medium graphs have a probability of 0.5 and dense graphs have 0.8 percent.

The tests are shown below:

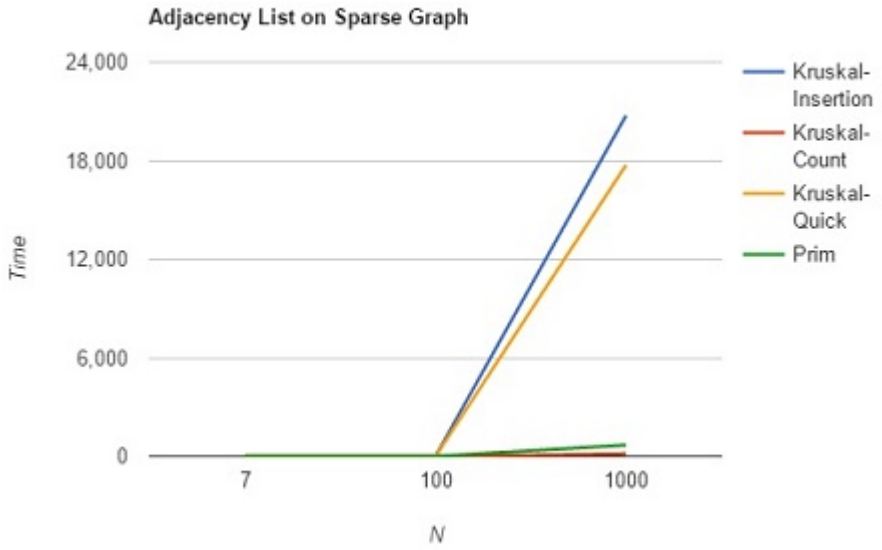
	Small	Medium	Large
Sparse	n=7 s=100,000 p=.4 n=7 s=10,000 p=.2 n=7 s=987,654 p=.2	n=100 s=100,000 p=.2 n=100 s=10,000 p=.2 n=100 s=987,654 p=.2	n=1000 s=100,000 p=.2 n=1000 s=10,000 p=.2 n=1000 s=987,654 p=.2
Medium	n=7 s=100,000 p=.5 n=7 s=10,000 p=.5 n=7 s=987,654 p=.5	n=100 s=100,000 p=.5 n=100 s=10,000 p=.5 n=100 s=987,654 p=.5	n=1000 s=100,000 p=.5 n=1000 s=10,000 p=.5 n=1000 s=987,654 p=.5
Dense	n=7 s=100,000 p=.8 n=7 s=10,000 p=.8 n=7 s=987,654 p=.8	n=100 s=100,000 p=.8 n=100 s=10,000 p=.8 n=100 s=987,654 p=.8	n=1000 s=100,000 p=.8 n=1000 s=10,000 p=.8 n=1000 s=987,654 p=.8

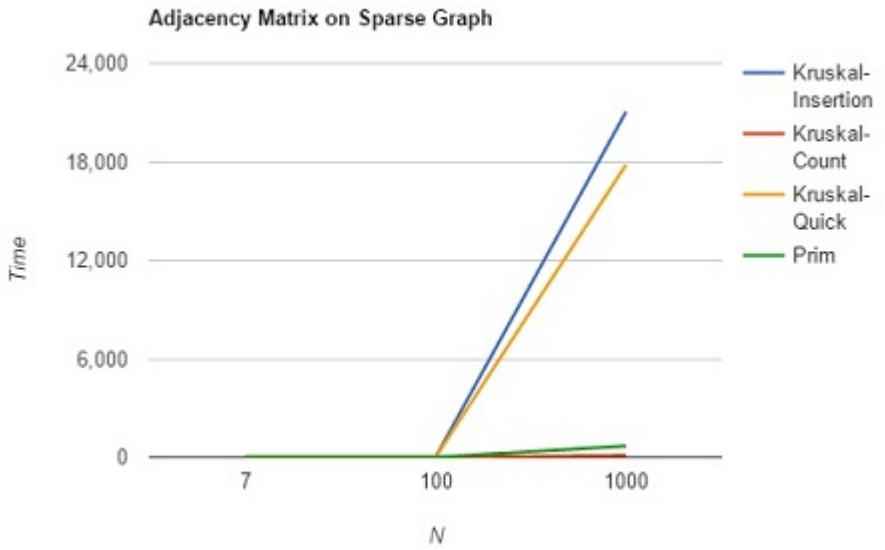
3 Results

After running the analysis program through the cases stated in the above section, the results were:

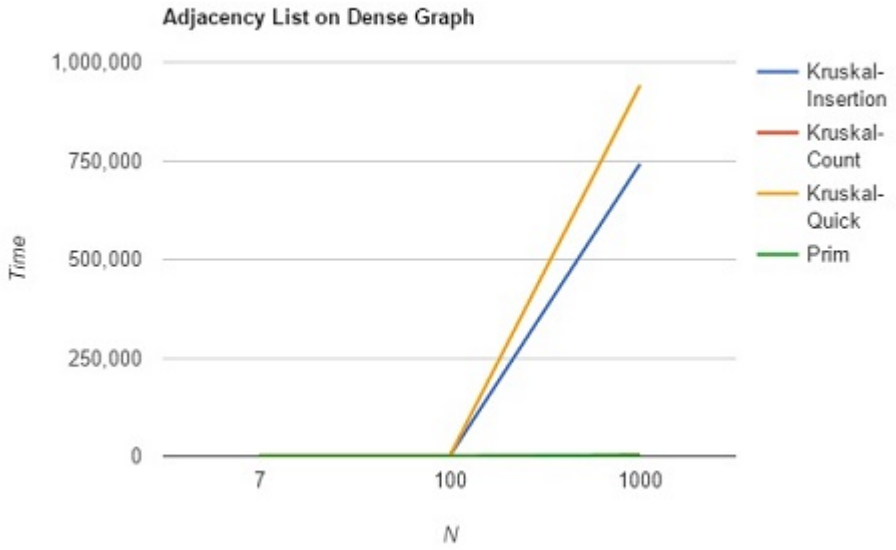
	Time took to (milliseconds):	Small	Medium	Large
Sparse	Generate the Graph	0	4.67	54
	Kruskal with Matrix using Insertion	0	5.33	21,047
	Kruskal with Matrix using Count	0	1.67	117.67
	Kruskal with Matrix using Quick	0	2.67	17,821.67
	Kruskal with List using Insertion	0	3	20,756.67
	Kruskal with List using Count	0	1	161.67
	Kruskal with List using Quick	0	2	17,742.67
	Prim with Matrix	0.67	3.33	695
	Prim with List	0	0.67	691.67
Medium	Generate the Graph	1.33	6	91.67
	Kruskal with Matrix using Insertion	0	15.67	154,298.67
	Kruskal with Matrix using Count	0	4	654
	Kruskal with Matrix using Quick	0	12.33	188,390.33
	Kruskal with List using Insertion	0	13.67	158,052
	Kruskal with List using Count	0	3.67	984.67
	Kruskal with List using Quick	0	11.33	196,837
	Prim with Matrix	0.67	5.33	1,938.33
	Prim with List	0	2.33	2,002
Dense	Generate the Graph	1	6.33	146.33
	Kruskal with Matrix using Insertion	0	38.67	741,338.67
	Kruskal with Matrix using Count	0	5.67	1627.67
	Kruskal with Matrix using Quick	0	32.67	915,342.67
	Kruskal with List using Insertion	0	35.33	742,352.33
	Kruskal with List using Count	0	5.67	2,826.67
	Kruskal with List using Quick	0	30.33	941,378.67
	Prim with Matrix	0.33	8	3,407.33
	Prim with List	0	5	3,388.67

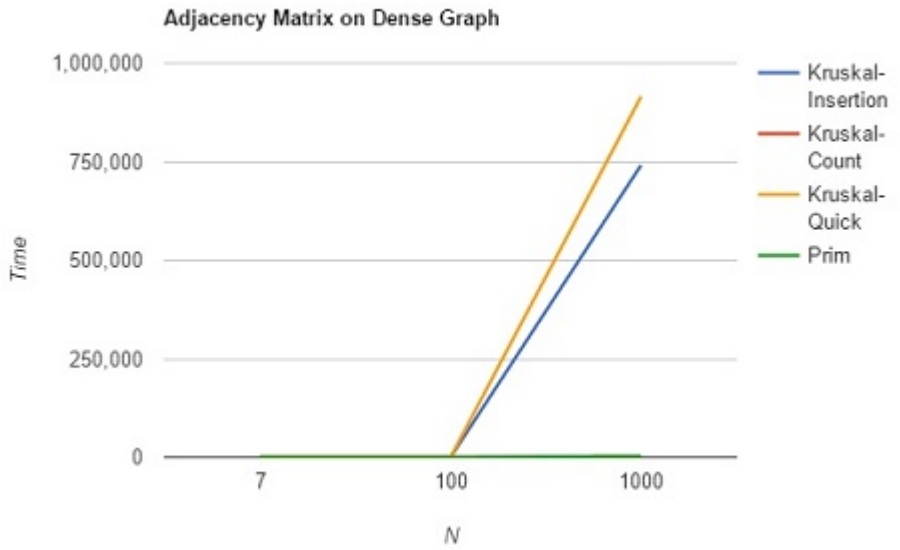
With a very small number of nodes, the algorithm choice becomes less relevant as all implementations take 0 (or close to 0) time to run. The difference starts to show in medium-sized graphs with Kruskal-Count's algorithm as well as Prim's algorithm running significantly faster than the other algorithms. Both of the adjacency matrix and adjacency list show this trend:





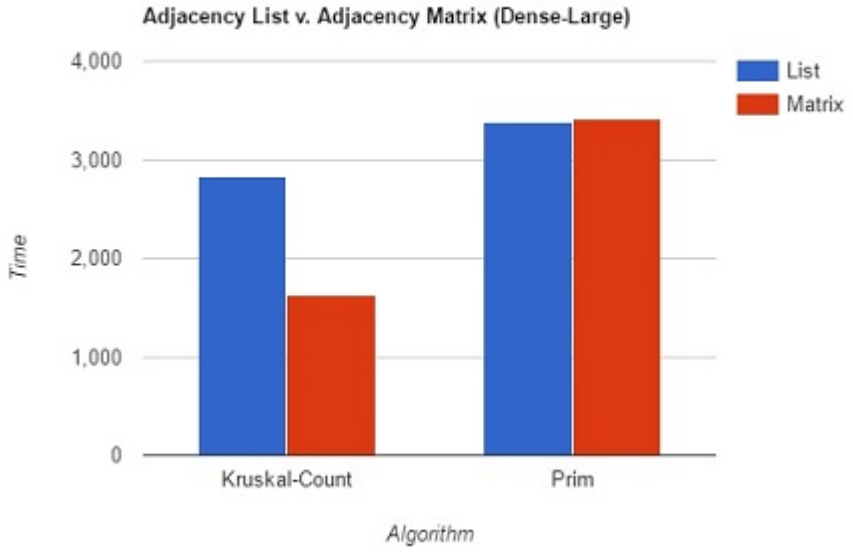
The trend also shows on dense graphs as well:

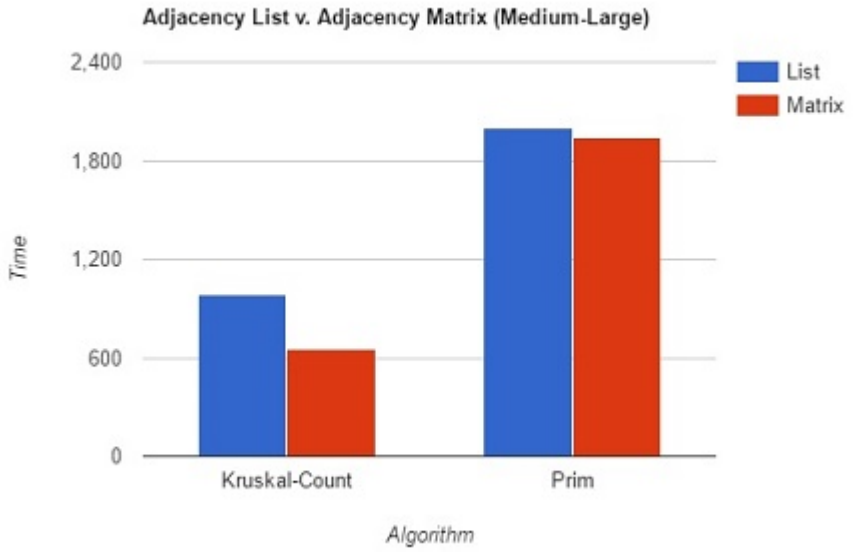


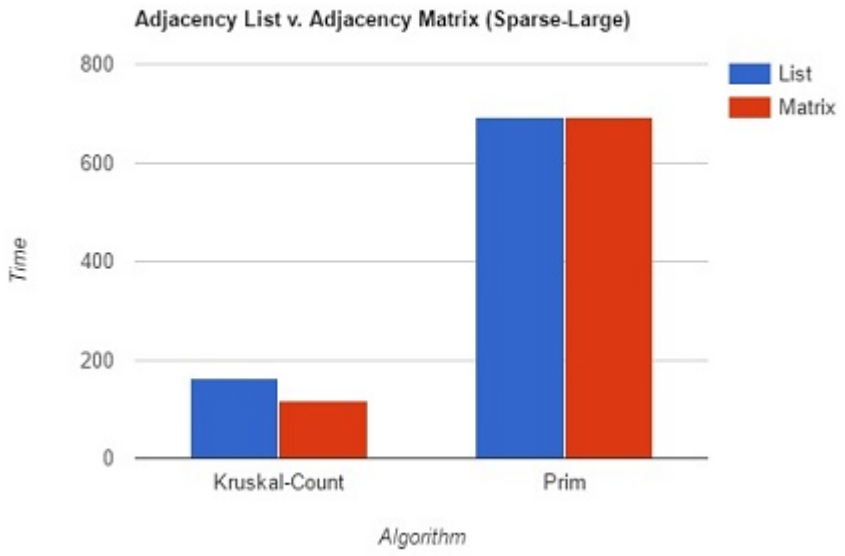


For large amount of nodes, the above graphs seems to imply that Kruskal-Count and Prim are identical. Comparing to Kruskal-Insertion and Kruskal-Count they are both significantly faster, but to each other they are vastly different. Also, both algorithms depend greatly on the graph's node amount,

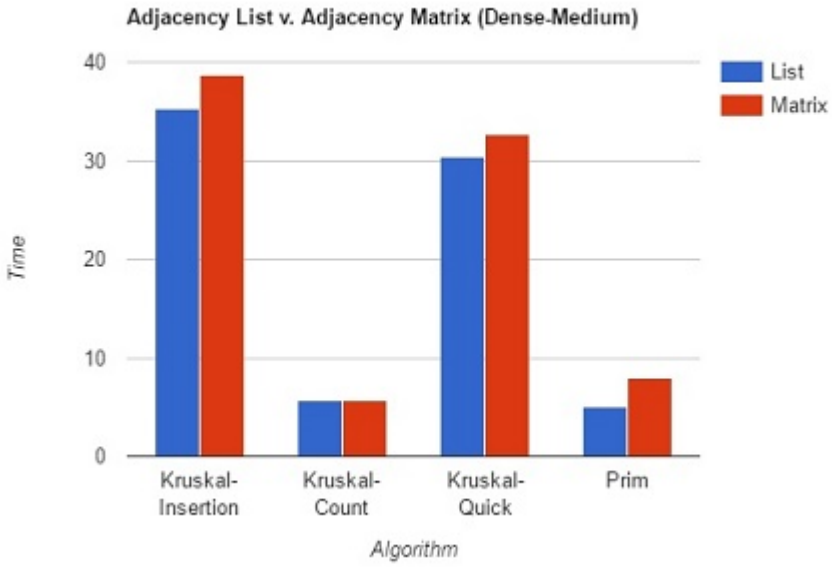
density of edges, and representation of the graph. Depending on those factors, each algorithm is superior over the other. Below are comparisons between the two algorithms with varying factors above. Small graphs were not included since the run-time was too fast to adequately compare the two algorithms.

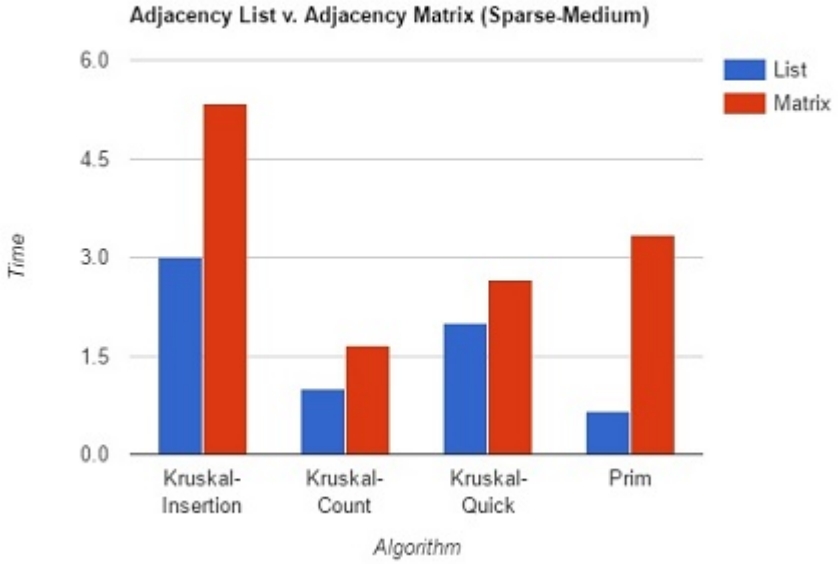






For sparse, medium, and dense graphs for large amount of nodes, Kruskal-Count's algorithm outperforms Prim's algorithm. While Kruskal-Count's matrix implementation always performs better than the list's implementation, it becomes much more pronounced the denser the graph gets. This juxtaposes the Prim's implementation which remains relatively the same throughout. While Kruskal-Count with Matrix looks like it is the superior algorithm, it does not do as well as Prim's when it comes to medium-sized graph cases. Also, the difference between all of the algorithms using both matrix and list is easily seen for the medium cases.





Again, Kruskal-Count's algorithm and Prim's algorithm both show to be better than the rest (though Kruskal-Quick's algorithm for sparse-medium graphs is respectable). For the larger graphs, the matrix implementation was always faster, yet for medium-sized, the list is -sometimes significantly- faster than the matrix implementation. Using an adjacency list, Prim's algorithm becomes

faster than either matrix or list of Kruskal-Count's algorithm. These results emphasize that there is not a simple answer to the best minimum spanning tree algorithm, but depend greatly on the size, representation, and density of the graph.

4 Analysis

4.1 Calculating the Constant

The time it takes these algorithms to run, as seen in results, depends greatly on the computer used as well as the implementation. A well-written algorithm will run slow on a bad computer. A poor-written algorithm will run fast on a super computer.

In perfect situations, the theoretical runtime of these algorithms should match the actual runtime. But there are outside impacts such as the computer used, other processes running, as well as "luckiness" of the random number generator. The formula of calculating this constant is:

Actual runtime =Constant * Theoretical runtime

Due to the random number generator in the program, the amount of edges in the graph varies from each iteration. Though, the average amount of edges can be achieved by multiplying the maximum amount of edges (the amount of edges if the probability was 100%) with the decimal percentage of the amount of edges wanted. This formula results into:

p * (n(n-1))

Theoretically, each of the algorithms have distinct time complexities. Kruskal's algorithms are dominated by their sorts and are reflected of them. Insertion, Count, and Quick sort time complexities are O(E^2), O(Eα(E,V)), and O(E log E) respectfully. Prim's algorithm is dominated by the heap implementation of a priority queue resulting in the algorithm to be O(E log V).

To get to these algorithms, some preparations were made that varied between the adjacency list and adjacency matrix implementation. For all the sorts in Kruskal's algorithm, all of the edges had to be put into an arraylist to be sorted. Representing the data as an adjacency matrix required an additional O(V * $\frac{V}{2}$) in order to access half of the 2D adjacency matrix which had the edges. The list implementation had to access the edges by looking into the adjacency list and finding the edge that connected the two vertices. The lookup of the edge was O(1), but the traversing of the adjacency list resulted in O(V *pV).

Prim does not have many differences in the matrix and list implementation. Both need to result in an arraylist full of all the vertices. This is easily achieved by both the matrix and the list by just traversing one-dimension of each, acquiring that vertex, and inserting into the new arraylist. For both of these setups, O(V) is the additional amount of time.

The theoretical time complexities of each algorithm is stated below:

Algorithm	Time Complexity
Kruskal-Insertion Matrix	O(V * $\frac{V}{2}$ + E^2)
Kruskal-Insertion List	O(V * pV + E^2)
Kruskal-Count Matrix	O(V * $\frac{V}{2}$ + Eα(E,V))
Kruskal-Count List	O(V * pV + Eα(E,V))
Kruskal-Quick Matrix	O((V * $\frac{V}{2}$ + E log E)
Kruskal-Quick List	O(V * pV + E log E)
Prim Matrix	O(V + E log V)
Prim List	O(V + E log V)

Using these formulas, the theoretical runtime can be found. For each of the nine categories, the theoretical times are below split into three different tables:

	Theoretical Time took to (milliseconds):	Small	Medium	Large
Sparse	Kruskal with Matrix using Insertion	94.56	3,935,400	3.92 E10
	Kruskal with Matrix using Count	32.4	6980	698000
	Kruskal with Matrix using Quick	49.8	26,683	3,983,837
	Kruskal with List using Insertion	80.36	3922400	3.92 E10
	Kruskal with List using Count	18.2	3980	398,000
	Kruskal with List using Quick	35.59	23,683	3,683,837
	Prim with Matrix	32.8	21,783	3,484,837
	Prim with List	32.8	21,783	3,484,837
Medium	Kruskal with Matrix using Insertion	465	2.45 E7	2.45 E11
	Kruskal with Matrix using Count	45	9950	995000
	Kruskal with Matrix using Quick	116	65,752	9,863,949
	Kruskal with List using Insertion	465	2.45 E7	2.45 E11
	Kruskal with List using Count	45	9,950	995,000
	Kruskal with List using Quick	116.7	65,752	9,863,949
	Prim with Matrix	99.24	60,852	9,364,949
	Prim with List	99.24	60,852	9,364,949
Dense	Kruskal with Matrix using Insertion	1152	6.27 E7	6.27 E11
	Kruskal with Matrix using Count	57.6	12,920	1,292,000
	Kruskal with Matrix using Quick	194	107,574	1.6 E7
	Kruskal with List using Insertion	1,168	6.27 E7	6.27 E11
	Kruskal with List using Count	72.8	15,920	1,592,000
	Kruskal with List using Quick	209.57	110,574	1.55 E7
	Prim with Matrix	177.37	102,674	1.55 E7
	Prim with List	177.37	102,674	1.55 E7

Using the previously stated formula to calculate the constant representing the "unexplained" differences between the actual and theoretical, the constants are shown below. These constants are extremely small mainly due to how fast

the CPU is in the testing computer. The only exception of these small numbers is quick sort. This result most likely comes from the programmer, myself, being unfamiliar with writing the quick sort algorithm and, most likely, made it less efficient than it could be.

Algorithm	Constant
Kruskal-Insertion Matrix	6.3 E-7
Kruskal-Count Matrix	6.54 E-4
Kruskal-Quick Matrix	0.019
Kruskal-Insertion List	6.45 E-7
Kruskal-Count List	9.9 E-4
Kruskal-Quick List	0.0066
Prim Matrix	2.06 E-4
Prim List	2.13 E-4

4.2 Profiling

The constants found above not only are influenced by how powerful the computer running it is, but how well the programmer programed it. As mentioned above, two outliers emerged from the constants: Kruskal-Quick Matrix and Kruskal-Quick List. While both performed better than the theoretical time, they ran significantly slower than the others proportionately.

Running hprof, a coding profiling tool, on both a large and small graph, some of the bottlenecks in the program coincide with the constant calculated. The four main bottlenecks are Edge.lessThan, Graph.partition, Graph.quickSort, and Graph.insertionSort. The insertion sort bottleneck was unavoidable since insertion sort naturally is O(N^2). Insertion also uses Edge.lessThan which is the comparison algorithm used for sorting. It compares the weights, then left node, and finally the right node. This priority system is the order of sorting. It is used be all the sorts (indirectly by count sort which calls insertion sort at nearly sorted making insertion just check for ties and be O(N)). The current implementation has three separate arraylist lookups and some minor looping. Seeing as how many functions rely on the function and how much it slows the program, another data structure or approach would greatly enhance the speed of the entire program.

The other two bottlenecks, Graph.quickSort and Graph.partition, were identified by the profiler but also the constant associated with them. These algorithms were the 2 and 4 slowest parts of the program with calls to Edge.lessThan (the slowest part). Being uncomfortable writing the algorithm due to unfamiliarity and modest understanding, there are probably something added to the code which makes it take more time than expected but also be more than O(N log N). With this coding mistake, many of the graphs in this report including the Kruskal-Quick's algorithm becomes false. Luckily, while the algorithm was skewed, it remained relative to where it was theoretically suppose to be. With count sort being O(N) and Prim's algorithm being O(E log V), Kruskal-Quick's algorithm should have been slower than both but faster than Kruskal-Insertion

which is O(N^2). With quick sort optimized correctly, the Kruskal-Quick's algorithm would be faster, but still remain in the same order of performance relative to the same algorithms.

5 Conclusion

There is not one perfect algorithm for a minimum spanning tree. Depending on the properties of the graph- including amount of nodes and density of edges- different algorithms become superior over others. For very small graphs, almost any algorithm would work simply because modern computing will make every algorithm run in 0 seconds. For bigger cases this changes.

For medium-sized graphs (around 100 nodes), Prim's algorithm using an adjacency list performs the best. Though if the program needs the graph to be represented as an adjacency matrix, Kruskal's algorithm using count sort emerges faster than Prim's algorithm using a matrix.

For the larger graphs (nodes around and greater than 1,000), Kruskal's algorithm using count sort using an adjacency matrix reigns supreme. It is significantly faster than Prim's algorithm -about 3 times faster for both sparse and medium-dense graphs. As the graph gets denser, Kruskal's algorithm becomes more comparable to Prim's algorithm yet still superior. Kruskal's algorithm using count sort and an adjacency matrix will save multiple minutes (at least) on large graphs.

But as shown in the constants and profiling, the algorithm must be well-written to get the desired, fast results.