A Comparison of Minimum Spanning Tree Algorithms
Prim's and Kruskal's Algorithms
December 04, 2014

Keith Cheng kchang@umbc.edu
Patrick Ritchie ritc1@umbc.edu
Michael Tang tang3@umbc.edu

Department of Computer Science and Electrical Engineering
University of Maryland, Baltimore County

Abstract

We found that Kruskal's algorithm is a better algorithm to use on setting up power lines in a new city compared to Prim's algorithm. These algorithms are used to solve the minimum spanning tree problem. Kruskal's algorithm sorts all the edges based on cost and creates trees while checking and avoiding cycles until all of the trees are merged. Prim's algorithm builds a subtree by picking one vertex and continuously adding a vertex with the cheapest edge connected to that tree until all vertices are reached. We conducted this research to find efficient ways of creating a power grid to connect all of the major points in a city, ensuring all points are connected. Multigraphs lets us take into account multiple variables on power lines that include cost, risk, distance, and other variables that should be taken into consideration based on the situation. Input graphs are randomly generated given number of vertices and a density, defined as the average number of edges leaving every vertex. The results of our investigation shows that Kruskal takes more time for more vertices and sparse graphs while Prims takes longer for more dense graphs. Some future considerations include determining how large of a difference the runtime will be if duplicates are eliminated or if there are directed edges.

Keywords:

Prim's Algorithm, Kruskal's Algorithm, Minimum Spanning Trees, Multigraphs

Acknowledgements:

Professor Alan Sherman - Ensuring complete research on time and identifying mistakes.
UMBC - Providing lab computers to run tests.

1. Introduction

We are exploring different situations where two algorithms create minimum spanning trees for a given graph based on variable number of vertices and densities. Prim's algorithm builds a subtree by picking one vertex and continuously adding a vertex with the cheapest edge connected to that tree until all vertices are reached. Kruskal's algorithm sorts all the edges based on cost and creates trees while checking and avoiding cycles until all of the trees are merged. We will determine the time and space requirements of each algorithm using a java program using several different undirected multigraphs which are manipulated by changing the number of edges and density.

The main problem is determining the conditions when Kruskal's algorithm or Prim's algorithm is more efficient in terms of time and space for finding the minimum spanning trees. Depending on the vertices and density on the spanning tree, the runtime and space allocation will change and there will be differences in runtime.

One example of a real world application of Prim's and Kruskal's algorithms would be creating new power grids in future cities. Power grids are loosely based on the minimum spanning tree problem as every point needs to be connected in order to transfer power. The edges can represent the cost of power lines and the vertices represent the consumers who needs to be connected to the electrical grid. Prim's and Kruskal's algorithms can help map new power grids by identifying which path is the best to take for making new power grids and determining whether to use underground or above ground lines to help reduce cost, as the physical distance between points is not the only factor of costs. Furthermore, both algorithms have their advantages and disadvantages when calculating the minimum spanning tree.

This report will contain a series of graphs showing the time and space usage, and a powerpoint presentation will be used to assist in displaying this information. We will also create a program to randomly create test graphs, with variable number of vertices and edges to gather more information than only using predefined graphs.

2. Background

Both Kruskal and Prim are greedy algorithms meaning they take the best possible choice in a locality to create the minimum spanning tree. Kruskal sorts the edges and then take the smallest edges and add them to tree as long as the edges do not create a cycle until the tree forms a minimum spanning tree. Prims takes a vertex and repeatedly adds a neighboring vertex with the lowest cost edge to find the minimum spanning tree. Both algorithm takes different time depending on how many vertices and edges there are in the minimum spanning tree.

Pseudocode for Prims:

```
E - array of edges of a graph
Prims(E)
Initialize an empty Priority queue
Initialize “edges” as an array of edges to be the MST
Initialize “reached” as a boolean array to keep track of vertices currently in the MST
For each vertex
        If the vertex has not been reached:
                Add the vertex to reached and add its neighbor edges that are not reached to the queue
                While Q not empty
                        Poll the minimum edge by weight
                        Add it to edges if at least one vertex is not reached
                        Add a vertex to reached and add its neighbors that are not reached to the
                        queue
                end while
        end if
end for
return edges
```

The act of polling picks the minimum edge regardless of multiple edges between two vertices handling multigraphs.

Pseudocode for Kruskal :

```
E - array of edges given to the program
Kruskal(E)
Sort E to from lowest to greatest
Initialize Tree to be empty
For each Edge in E
        If neither vertices in E is found in tree
                Create a new empty List
                Add the edge to the empty List
                Add the List to Tree
        If vertices of Edge is in one list of Tree
                Don’t add the E into Node
        If both vertices of E are in Two different ArrayList of Node
                Merge the Two List together and add the new edge
        If one of the vertices of E is in the ArrayList of Node
                Add the E into that ArrayList of Node
        End if

end for
return the edges with the minimum spanning tree
```

3. Previous Work

Two algorithms called Kruskal's algorithm developed by Joseph B. Kruskal and Prim's algorithm developed by Robert C. Prim to solve the minimum spanning tree problem. There was no available information at what ratio of vertices and densities does Kruskal become more efficient than Prims while the runtime of both algorithms were already well known.

4. Methodology

We decided to write our code in Java because it is a higher level language that has several built in libraries that we can utilize. Java had built in libraries of HashSet, HashMap, and PriorityQueue that we utilized because writing our own code has the risk of introducing errors and causing runtimes to be longer than necessary. HashSet and HashMap has O(1) amortized runtime for insert, remove, lookup, or delete because they obtain a hash value and looks up a hash table with that value. Priority queue has O(log(n)) runtime because of looking at each node and moving the nodes up or down depending on priority for add and poll.

There was also the choice of whether we wanted to import our graphs into algorithm as Adjacency List, Adjacency Matrix, or a list of edges. We decided to import our edges into our algorithm as a list of edges because both algorithm utilize edges differently. Kruskal would sort the edges first and create lists to hold edges with similar weights while Prims creates an Adjacency List to choose the next node to add to a priority queue.

We will create randomly generated graphs that control the number of vertices and density for testing. This procedure returns a list of objects called edges, by first creating a minimum spanning tree and then adding additional edges until the desired density is met, see attached code files for details. Every edge in the edge list will have two vertices and an edge cost and each algorithm will have its own approach in solving the Minimum Spanning Tree.

The results are gathered by randomly creating graphs with a given number of vertices and constant edges, and running each algorithm on each graph. This process is repeated three times and the average time and space usage is taken for reliable results. Space usage is measured in bytes, and time usage is measured in nanoseconds based by the systems internal clock. We then export our results into Excel and find the amount of edges where both algorithms solve the graph in the same amount of time. Then we repeat this entire process, this time holding the number of vertices constant and changing the density (the number of edges per vertex). Results are shown below.

5. Results

All below graphs can studies of 27-30 samples, each sample being the average of 3 test cases. This is to avoid Java's garbage collection interfering with the simulations. When garbage collection is called during runtime, there no guarantee that garbage collection actually occurs. This causes some test cases to appear to take more space as the calculations include space usage from prior tests.

A study comparing time differences solely increasing the number of vertices (1-1000) on an input graph. The density is one, meaning every vertex has one edge connected to it, so each input graph is a minimum spanning tree:

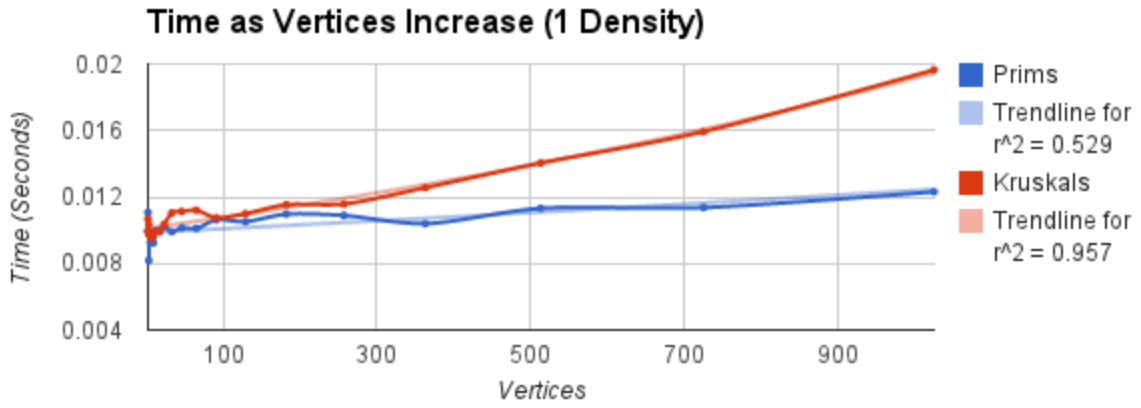

Figure 1 - Time, Small Number of Edges - Prims appears to run quicker on sparse input graphs. Since the density of these graphs is 1, the input graphs are minimum spanning trees and have (vertices - 1) edges.

The running times from this graph appear almost equal, but when you extend it to input graphs with a larger amount of vertices (1-8193), kruskal's running time is shown to of a larger order:

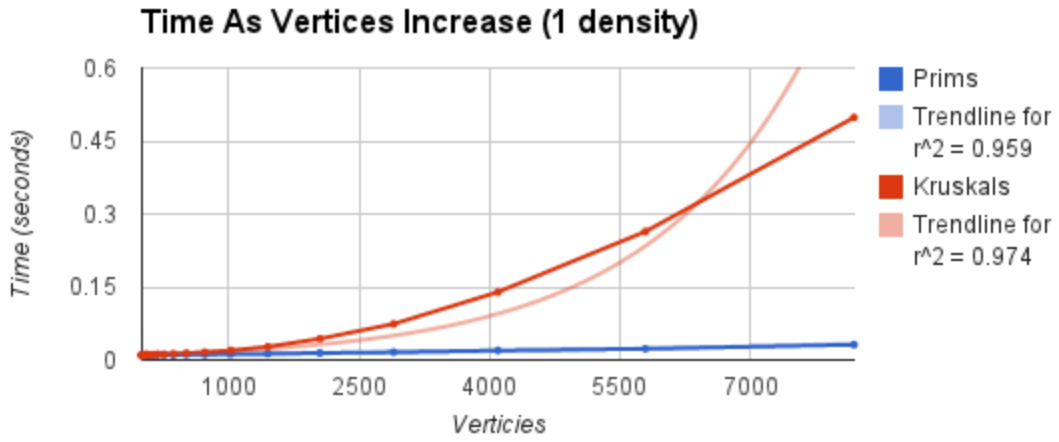

Figure 2 - Time, Large Number of Edges - Time with varying edge counts and constant density

Since both algorithms performed had asymptotically equal running times on small (under 1000) random graphs, we used a value of 100 vertices as the default for the next series of tests, which modify the density of the graph. Below shows the running time of each algorithm given a 100 vertex graph with a densities varying from 5 to 200. Note that, the density of a graph is the average number of edges leaving each vertex.

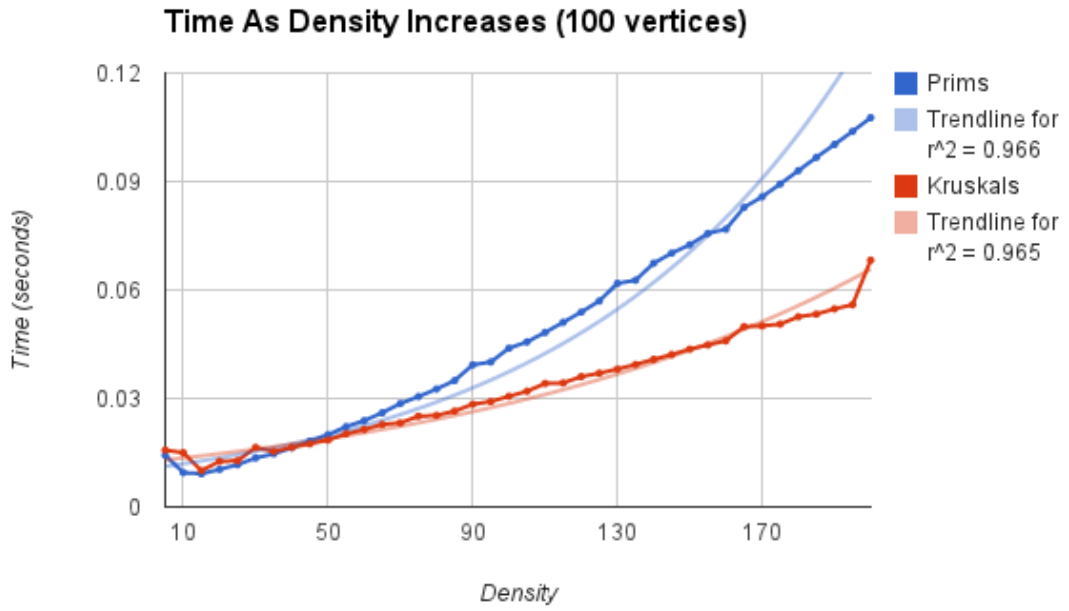

Figure 3 - Time, Varying Densities - Time usage with constant vertices and varying edge. Kruskal's algorithm runs at a faster rate after the crossover point 40 density

From above graph, when increasing the density of the graph, while holding the number of vertices constant, Prims algorithm runs asymptotic slower. A time usage graph is below with both increasing density and vertices.
This further increases evidence of Prims working better on lower densities, where the density just passes 50 and the number of vertices is around 40 Prims starts to take longer than Kruskal's to run.

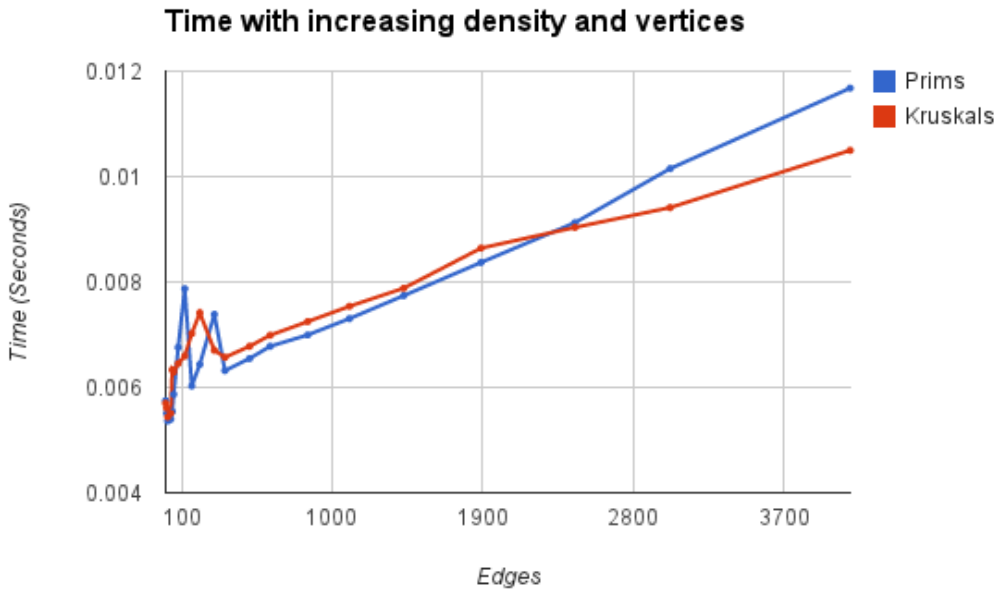

Figure 4 - Time with varying density and vertices - Time comparisons during tests with increasing densities and vertices. Prims runs faster on graphs with many edges.

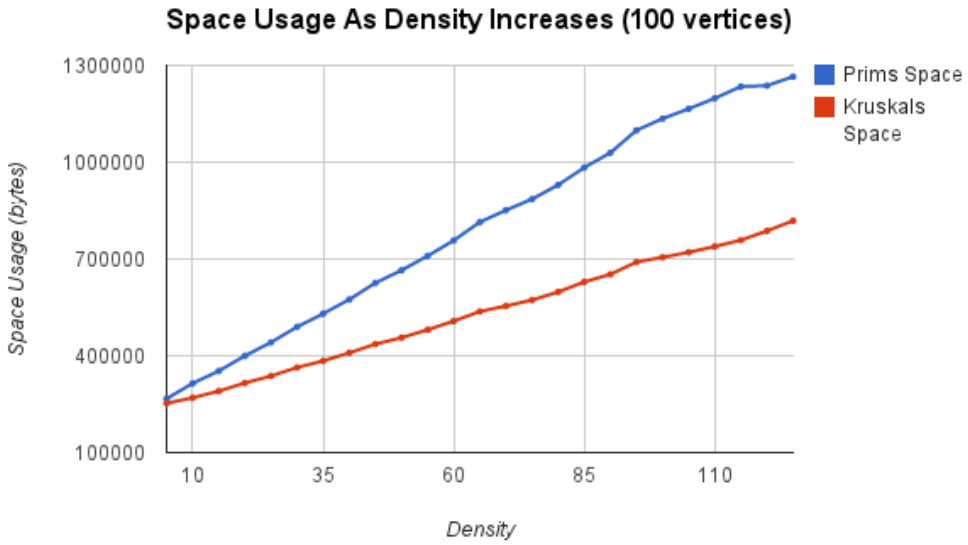

Figure 5 - Space usage - Space usage with input graphs with varying number of edges and constant vertices. Prims algorithm saves more data per edge than kruskal's, hence higher space usage in tests.

Note that since the implementation of the algorithms uses a list of edges, the space usage only depends on the number of total edges and not how those edges are positioned in the graph. This would only be a concern the algorithms could work on the data sets without first having to load in each one, which takes O(|E|) space. Therefore, the space usage is independent of the density of the graph.

6. Regression Analysis

Prims Regression Analysis on time as vertices increase:

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.896871092
R Square	0.804377755
Adjusted R Square	0.796552866
Standard Error	0.740197805
Observations	27

ANOVA

	df	*SS*	*MS*	*F*	*Significance F*
Regression	1	56.32191448	56.32191	102.797327	2.42545E-10
Residual	25	13.69731978	0.547893		
Total	26	70.01923425			

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*	*Lower 95.0%*	*Upper 95.0%*
Intercept	-0.267862996	0.160892254	-1.66486	0.10842575	-0.599226792	0.0635008	-0.5992268	0.063501
X Variable 1	0.000731523	7.21502E-05	10.1389	2.4254E-10	0.000582927	0.00088012	0.00058293	0.00088

Figure 6 - Prims Regressions Analysis - Z-statistics on Prim's algorithm as input size increases. Very high anova F values and low p-values such that input size has an effect on the time and space usage of the algorithm.

Kruskals Regression Analysis on time as vertices increase:

SUMMARY OUTPUT

Regression Statistics	
Multiple R	0.986787103
R Square	0.973748787
Adjusted R Square	0.972698739
Standard Error	0.001385286
Observations	27

ANOVA

	df	*SS*	*MS*	*F*	*Significance F*
Regression	1	0.001779575	0.001779575	927.3369465	2.77549E-21
Residual	25	4.79754E-05	1.91902E-06		
Total	26	0.00182755			

	Coefficients	*Standard Error*	*t Stat*	*P-value*	*Lower 95%*	*Upper 95%*	*Lower 95.0%*	*Upper 95.0%*
Intercept	0.006350752	0.000301111	21.09106351	1.92557E-17	0.005730602	0.006970902	0.005730602	0.0069709
X Variable 1	4.11195E-06	1.3503E-07	30.45220758	2.77549E-21	3.83385E-06	4.39005E-06	3.83385E-06	4.39E-06

Figure 7 - Kruskal's Regressions Analysis - Z-statistics on Kruskal's algorithm as input size increases, with 27 observations. Very high anova F values and low p-values such that input size has an effect on the time and space usage of the algorithm.

As the motivation for this project involves planning of future cities, the most realistic test case is shown above. A minimal amount of vertices, that could represent destinations, and a large possible densities, as there are often many different ways to travel from one point to another. The regression lines above show that Kruskal's runs faster, as Prims has a large exponent. This is the exact reverse of test cases with low density but a high number of vertices.

Along with these results, the option of removing duplicate edges was also tested. Only considering the cheapest edge between any two vertices changes the problem from a multigraph problem to a simple graph problem, however the operation of doing so takes O(|E|^2) time, as each edge needs to be compared to every other edge. Thus, in sparse graphs, this isn't useful because it adds additional running time while not simplifying the problem. In dense graphs, this is a useful technique, however if implemented from within both algorithms, the benefit is equal and the relative runtime / space usage is the same.

Since Kruskal's algorithm and Prim's algorithm can generate different multigraphs depending on how they break ties, there is no way to ensure both algorithms generate the same tree. This makes testing the correctness of each difficult, as you'd need to generate all possible solutions and compare each to the output of each function. So we instead compare the total weight and edge counts of each algorithm for equality. Setting the random graph generator to use a wide range of values (1-500) lowers the probability of those checks producing a false positive, as the chance of multiple paths having the same weight is lessened.

7. Discussion

Prims algorithm runs in O(|E|+|V| log |V|) using a binary heap priority queue and an adjacency list. Kruskal's runs in order O(|E| log |E|). Since multigraphs do not limit the amount of edges to |V|^2, Kruskal's algorithm running time cannot reduce further to O(|E| log |V|). Using different data structures would influence Prim's algorithm running time, such as storing the graph as an adjacency matrix (O(|V|^2)), or using a fibonacci heap (O(|E| + |V| log |V|)). In cases with dense graphs, Kruskal's runs faster than prims. In test cases where both density and vertices increase our implementation shows Kruskal's mostly running faster and using less space than Prims. Results shows that Prims only runs faster than Kruskal's with densities under 10 for graphs with 100 vertices, or generally with graphs where the average number of nodes leaving each vertex is less than the square root of the total number of vertices.

Prim's and Kruskal's are not the only algorithms that can be used to find the minimum spanning tree. Other greedy algorithms that can be used to find minimum spanning trees are Boruvka's algorithm and Reverse-delete algorithms. However, it is not known how the ratio of vertices and edges for runtimes will compare without implementing and testing the algorithms.

8. Conclusion

In an situation of city planning where the number of the vertices, or points to connect is large, but the possible paths between any two points is small, under sqrt(|V|) such as in the context of power lines, Prims algorithm will run faster. However, the space usage is higher than Kruskal's, as Kruskal's algorithm requires at most 60% of the space. In situations where the number of points to connect is low, but there are many different ways to connect them, Kruskal's algorithm runs faster, however still require additional space. In the context of city planning, Kruskal's algorithm is preferred for connecting water lines, since water lines can be placed anywhere underground.

9. Future Questions / Research

1.) Can time be decreased if we don't need a perfectly correct solution? Can graphs with a total cost close to a minimum spanning tree be computed faster?
2.) Can we preprocess the data (removing self loops, for example) to further reduce time or space?
3.) Would directed graphs change the complexity of the algorithm?
4.) How will Prims and Kruskal compare to the Boruvka's and Reverse-delete algorithm for finding the minimum spanning tree?
5.) Would results change if simulations were run using c/c++ or another language with stricter memory management?

10. References

Cormen, T., Leiserson, C., Rivest, R., & Stein, C. (2009). 23 - Minimum Spanning Trees. In Introduction To Algorithms (3rd ed.). Cambridge, Massachusetts: The MIT Press.

Graham, R.L, Hell, Perol, "On the history of the minimum spanning tree problem", Annals of the History of Computing, Vol 7, January, 1985, 10/9/2014, <http://www.ms.unimelb.edu.au/~woodd/NetOpt/GrahamHell-HistoryMST.pdf>

45th Southeastern International Conference on Combinatorics, Graph Theory, and Computing. (n.d.). Retrieved November 14, 2014, from http://math.fau.edu/cgtc/cgtc45/Abstracts.html

Malkevitch, J. (n.d.). Feature Column from the AMS. Retrieved December 5, 2014, from http://www.ams.org/samplings/feature-column/fcarc-trees

11. Appendix

General: (Used for graph generation, and testing)

- GraphGenerator.java
- TimeSpace.java
- Driver.java
- Edge.java

Kruskal's Algorithm:

- Kruskals.java

Prim's Algorithm:

- PrimsNew.java
- AdjacencyList.java.

Code files attached.