A comparative analysis of Prim’s, Kruskal’s, and Boruvka’s Algorithms for Minimum Spanning Trees

Ellie Sceeles

CS 584- Final Project

Spring, 2019

1. ABSTRACT

Though there are numerous methods for the discovery of minimum spanning trees (MSTs), Prim's, Kruskal's, and Boruvka's are among the most popular. In this paper, we analyze and compare these three algorithms on graphs of a variety of sizes and densities. Results show that despite their shared O(E log V) time complexity, Kruskal's algorithm outperformed the others in almost every case.

2. BACKGROUND

A spanning tree is a subset of a graph's edges that connect each node while remaining acyclic. Often, many spanning trees can be generated for a given graph and each can be described by their cost (the sum of the weights of its edges). A minimum spanning tree has the lowest possible cost for the graph. A graph can also have more than one MST, but there will only ever be one correct tree cost [7]. MST algorithms can be used to solve many classic computer science problems, like the traveling salesman, but are also used in a variety of applied fields. They are commonly used to design networks, perform cluster analysis, and aid in handwriting recognition [12].

The three algorithms I'll be analyzing are similar in that they share an algorithmic complexity class of O(E log V). In this paper, we will use E to represent the number of edges of a graph and V to represent the number of vertices. The graph classes constructed in these implementations store edge information as an adjacency list. This is generally considered to be the fastest structure for MST algorithms, though it would be possible to implement them using an adjacency matrix as well [2]. Also, all three algorithms are in the greedy class of algorithms, making the best local choice (smallest weighted edge) at each step. For pseudocode of all algorithms discussed, see Appendix A.

3. ALGORITHMS

Kruskal's algorithm was created in 1956 by Joseph Kruskal [10]. It starts by treating each node in the graph as an individual tree in a forest. The encompassing MST is built by first sorting the edges in increasing order and then iterating through each edge, adding it to the tree if it does not create a cycle. In other words, as each edge is considered, a check is performed to assure that both nodes are not already a part of the same tree. If they are not, that edge is added to the MST.

A popular implementation of this algorithm utilizes a Union-Find data structure that tracks and merges the desired groups or trees. Because of the simplicity of its data structures, Kruskal's is considered an optimal algorithm for dealing with sparse graphs [16].

The time complexity for Kruskal's is driven by the sorting of the edges, which takes O(E log E) time. As we iterate through each edge, connecting trees in the forest, we perform at most two 'Find-Set' operations and potentially one Union-Set operation, taking at most O(E log V) time. Since the number of edges in a simple graph cannot be larger than V^2, O(E Log E) and O(E log V) are equivalent, leaving us with a total worst-case complexity of O(E log V) [17].

Boruvka's algorithm was created by Otakar Boruvka in 1926 as an efficient method for planning electrical networks [4]. As with Kruskal's, it considers each node as its own tree at the outset of the problem but continues until all nodes have been incorporated into one group. At each iteration, it identifies the smallest weighted edge that connects any two separate groups. If these edges are not already included, they are added to the final tree. The algorithm continues merging until one final group remains.

Unlike the other algorithms considered in this paper, Boruvka's algorithm can add multiple edges to the final tree simultaneously without disrupting the results. Due to this, the algorithm lends itself nicely to parallel computing. When working with multiple CPU's, it's possible to allow for this merging of groups and for the identification of minimum weight edges to happen in parallel. Without parallelization, it still runs in O(E log V) time.

The selected implementation of Boruvka's for this paper utilizes the same Union-Find data structure as Kruskal's to quickly identify and merge groups. Its outer loop operates at most log V times, lending to the overall O(E log V) complexity. In certain graphs, such as planar graphs, it can be implemented to run in linear time [18].

Prim's algorithm was created by mathematician Vojtech Jarnik in 1930 [14]. It varies from the previous two algorithms in that it cannot be used on disconnected graphs. This is because Prim's considers only one tree, the current MST, and considers each node only in its relation to it. The algorithm utilizes several data structures that allow it to track which nodes are reachable (have edges) from our current tree and the weight of the minimum weighted edge connecting them. It begins by adding a randomly selected node to the tree. At each step, it selects the smallest weighted reachable edge and adds it to the tree. It then updates the data structures to include this newly added node's neighbors and updates all minimum weighted edges, if applicable.

Prim's complexity is dominated by the data structure chosen to store reachable, but unvisited, nodes. A standard implementation is to use a minimum binary heap ordered by the weight of each edge. Using this, the inner loop is executed at most V+E times and the actions within its O(log V) times. This provides for an overall complexity of O(E log V). Prim's can be optimized to run even faster with the use of a Fibonacci heap or d-ary heap. Utilizing these data structures for appropriately dense graphs can potentially bring the complexity down to linear time [1]. For the purposes of this paper, we will implement a standard binary heap.

4. PROCEDURES

For this paper, I tested graphs of a variety of sizes and densities against these three algorithms to analyze their runtimes. The graphs used were generated locally on my computer using a random, connected graph generator [9]. Edges and their weights were selected at random. Sizes and densities were tracked carefully for analysis and multigraphs were allowed due to the complicated and space sensitive nature of generating large simple graphs. All graph objects were initialized prior to the timing of the functions.

All source code was written in the Python language, using the JetBrain's PyCharm IDE. The Python interpreter does perform automatic garbage collection. To remove this interference, I utilized the garbage collector interface to disable collection for the duration of my program. Function timing was performed using the Timeit module from the Python Standard Library. Statistic test functions were imported from the SciPy and Statistics libraries.

To account for any variance in CPU performance, I designated a single, specific CPU core to be solely responsible for the execution of my code. I assured that the program was given the highest priority on my operating system and that no other processes were active while tests were being performed. The processor used has a base frequency of 1.8 GHz.

5. LIMITATIONS

I had originally planned to implement a parallel version of Boruvka's algorithm to optimize its performance. In the early stages of this project, I wrote a version of Boruvka's to utilize threading, but its runtime was significantly slower than the version without it, often running about 5X slower than the original version. After doing research into this project, I learned about Python's Global Interpreter Lock, which only allows one thread to run in the interpreter at once [3]. Due to the GIL and the massive slowdown in runtime that threading created, I have removed this algorithm from the analysis.

Additionally, though I would have liked to run tests on larger and denser graphs, I ran into memory limitations on my computer. Any graph with over 4,000,000 edges (for example, a 9,500-node graph at .1 density) would lead to memory errors and the program crashing. Due to this, I had to restrict the size of my input to smaller or sparser graphs. Future research should be done to compare extremely dense graphs ($D > .8$) on Prim's and Kruskal's algorithms to test how this effects runtime.

6. TESTS AND RESULTS

For the first test, I created three major graph classes based on the number of their nodes [Table 1]. For the Small and Medium sizes, I timed the algorithms on both sparse and dense graphs to see how they compared. Due to RAM limitations, I was unable to test dense graphs for the Large class. To account for any additional variance in performance, I timed each algorithm multiple times (10) for the same size and used the means for comparison.

Graph Class	Nodes	Edges
Small Sparse	1,000	4,995
Small Dense	1,000	399,600
Medium Sparse	10,000	49,995
Medium Dense	10,000	3,999,600
Large Sparse	30,000	449,985

Table 1

As can be seen in Figures 1 and 2, Boruvka's algorithm operated the slowest in every case, dwarfing all other results on large, sparse graphs by a factor of almost five and on medium, dense graphs by a factor of ten. Kruskal's operated the fastest on graphs of all densities and sizes, running consistently 5X faster than Prim's [Table 2].

Average Runtimes	Kruskal	Prim	Boruvka
Small Sparse	0.01833972700	0.23997965580	0.13373088189
Small Dense	0.01833972700	1.00799050290	6.46566033690
Medium Sparse	0.25054429370	1.38441561089	2.56537433479
Medium Dense	2.89363291370	11.4850195454	123.317127581
Large Sparse	1.08299643300	5.43939676109	25.4644365025

Table 2

One-way ANOVA tests performed on the data show that all tests performed at significantly different rates on every graph other than the small, sparse graph. Prim's and Kruskal's were more similar than Boruvka's so an additional T-Test was performed between those two groups [Table 3]. As with the ANOVA, the only similar results were from the small, sparse graphs. Both tests showed groups growing significantly more different as more edges were added.

P-Value Chart	ANOVA on all algorithms	T-Test on Prim and Kruskal
Small Sparse	0.1715416576720745	0.1303633902959629
Small Dense	1.368772877623775e-34	3.084957193519093e-17
Medium Sparse	7.836416355213673e-16	5.5174543300429124e-18
Medium Dense	2.1496130328342898e-58	3.622495830892177e-30
Large Sparse	8.792873886724299e-57	4.347286955076948e-29

Table 3

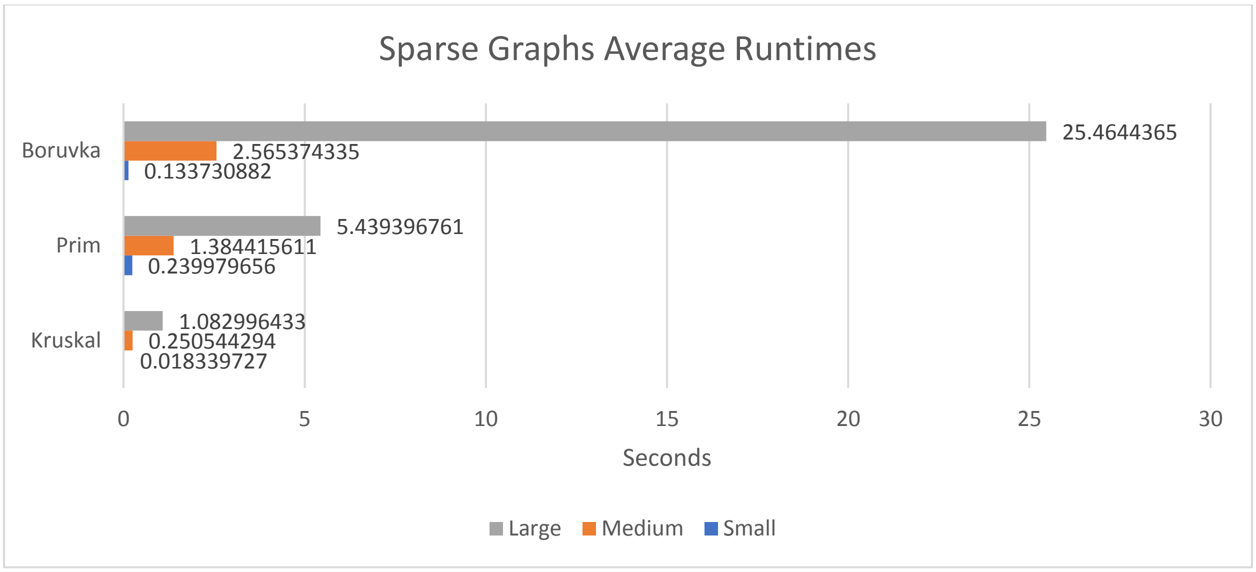

Figure 1

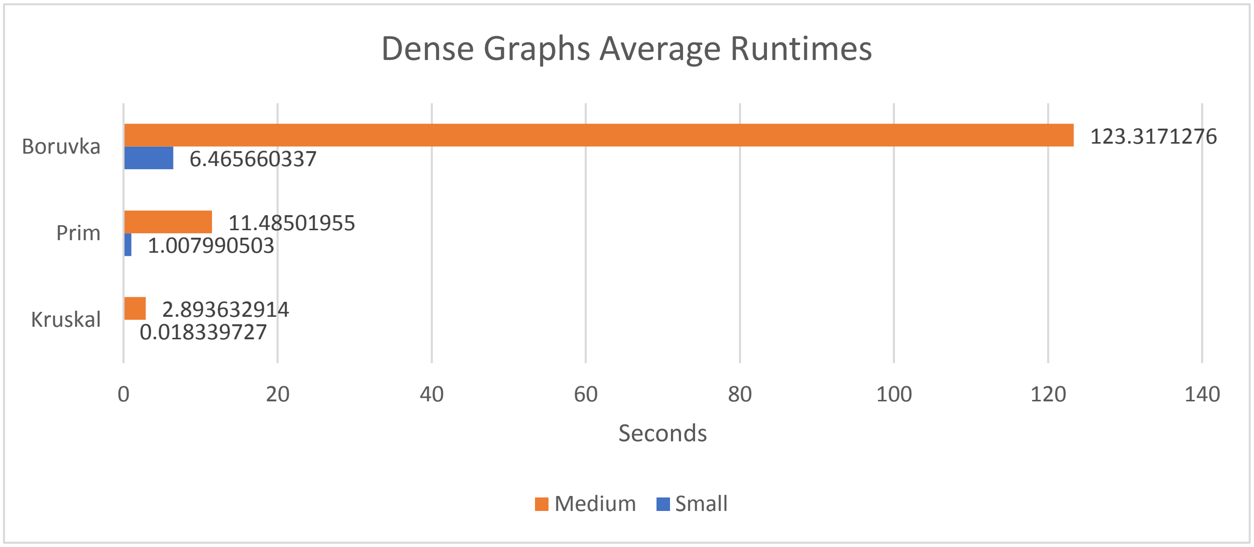

Figure 2

My second program compared growth in runtime for each algorithm as nodes increased. For these tests, I started with a small graph (V = 1000) and gradually incremented the number of nodes (max: V = 9000), timing the function at each change in size. The graph density was sparse (0.01) so that functions could be run on the max number of nodes my computer could store. After collecting data, I calculated the growth rates of each algorithm over the regular increment in node count. I compared these growth rates in an ANOVA test between the three algorithms. I also performed a two-sided t-test between the growth rates of each function and the theoretical model to see how accurate the theoretical model was.

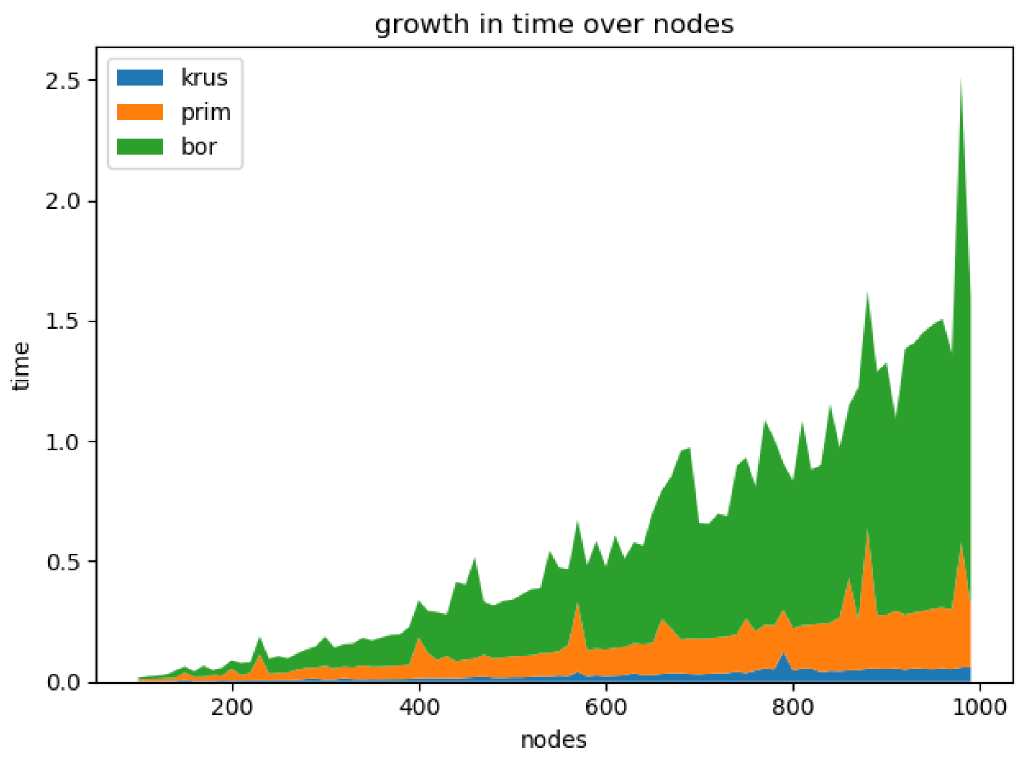

Figure 1

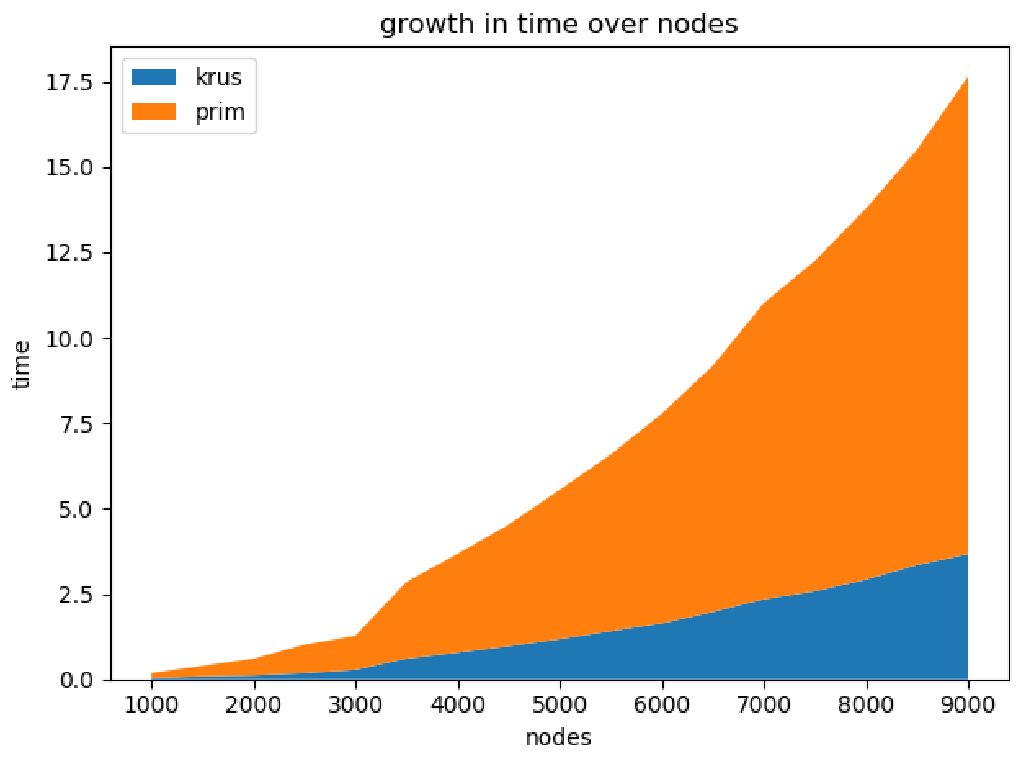

Figure 2

As seen in Figure 5, for all algorithms on smaller graphs, as nodes increased, so did run time. As with the prior test, Boruvka's performed the worst and Kruskal's the best at every interval. Figure 6 shows Prim's and Kruskal's runtimes on larger graphs. Figure 7 demonstrates the growth rate [13] at each interval for the three functions plotted against the E Log V growth rate to evaluate whether the change in time was expected. All three algorithms followed the basic trend of the theoretical growth rate. T-tests between all three algorithms (Table 3) and the E log V curve show that there were no significant differences, meaning that these algorithms did indeed grow in O(E Log V) time.

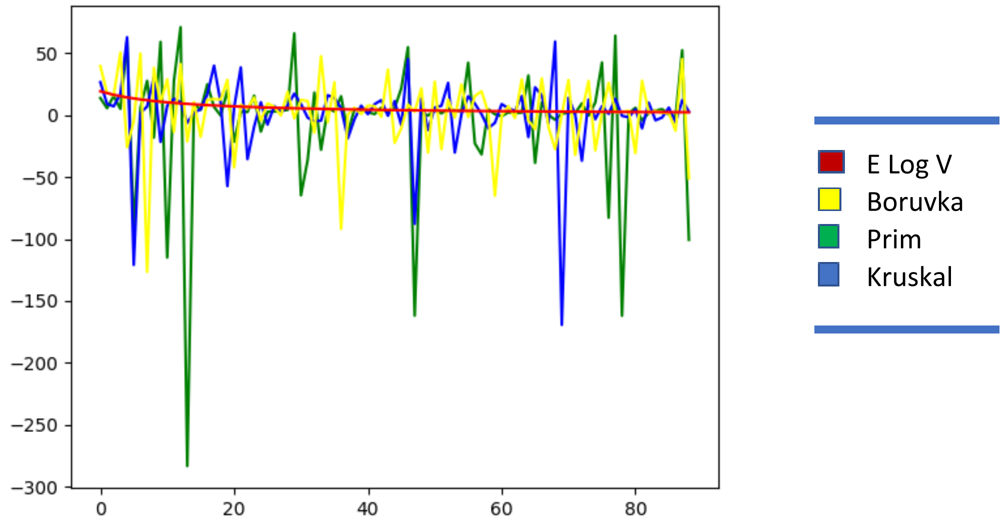

Figure 3 – growth rates

Growth Rate T-Test	P Value
Prim's and E Log V	0.8452312869790687
Kruskal's and E Log V	0.8998993046743137
Boruvka's and E Log V	0.25452861397179083

Table 3

7. OBSERVATIONS

While I had expected Prim's and Kruskal's algorithms to run more comparably, Kruskal's outperformed Prim's in every case. On graphs 10,000 nodes or higher, Kruskal's ran at least 5X faster regardless of the graph's density. I believe this is due to Prim's utilization of a heap and the overhead that maintaining it requires [8]. The Standard Python Library's sorting function (used in the other algorithms) is a variation of quicksort which, when implemented correctly, can run in the real-world much faster than using a heap.

I believe this may also account for its performance on small, sparse graphs. Despite Boruvka's overall lack of success, this was the one case where Prim's performed the worst. On these graphs, Prim's took almost twice as long as Boruvka's to identify an MST; whereas, in every other case, it ran 2-10X faster. Boruvka's, despite always coming in last, was seen to lag closer behind on sparser graphs.

8. CONCLUSION

In conclusion, testing Prim's Kruskal's and Boruvka's demonstrated clear and significant differences in performance. Kruskal's was consistently the fastest to produce a MST, followed by Prim's, with Boruvka's doing the worst every time. This fits in with the development timeline of these algorithms. Boruvka's may be the slowest, but it was the first and arguably the easiest to implement and understand. At the creation of each algorithm, it would have been the fastest and most efficient for its time. However, as new data structures and revelations in complexity occurred, newer algorithms would be created to set the standard and improve efficiency in MST generation. The algorithms are still used today in the development of networks for computers, electrical and water supply, transportation, communications, and so much more.

APPENDIX A: PSEUDOCODE

MakeSet(u) makes a set out of vertex, u
FindSet(u) returns the set that vertex, u, is a part of
Union(u, v) unions the sets that u and v belong to

MST-Kruskal(G)

```
result = emptySet()
for each v in G.V:
    MakeSet(v)
edges = sortNonDecreasingByWeight(G.E)
for each edge (u,v) in edges:
    if FindSet(u) $\neq$ FindSet(v):
        result = result $\cup$ (u,v)
        Union(u,v)
return result
```

GetSmallestEdge(T, F) finds the smallest weight edge connecting T and F

MST-Boruvka(G)

```
F = emptyForest()
while size(F) < |V| -1:
    for each tree T in F:
        e = GetSmallestEdge(T, F)
        if e not in F:
            F.append(e)
return F
```

MST-Prim(G)

```
remaining = Heap()
MST = list()
for each v in G.V:
    v.key = $\infty$
    v.parent = NULL
    remaining.push(v)
start = popRandomNode(remaining)
while remaining.notempty():
    cnode, edge = removeMinEdge(remaining)
    MST.append(edge)
    for each v in G.Adj[cnode]:
        if v in remaining and weight(cnode, v) < v.key:
            v.parent = cnode
            v.key = weight(cnode,v)
return MST
```

APPENDIX B: WORK CITED

[1] D. Mehta and A. Barnwal, “Prim's MST for Adjacency List Representation | Greedy Algo-6,” GeeksforGeeks, 27-Sep-2018. [Online]. Available: https://www.geeksforgeeks.org/prims-mst-for-adjacency-list-representation-greedy-algo-6/. [Accessed: 31-May-2019].

[2] A. Barnwal, “Graph and its representations,” GeeksforGeeks, 04-Oct-2018. [Online]. Available: https://www.geeksforgeeks.org/graph-and-its-representations/. [Accessed: 31-May-2019].

[3] D. Beazely, “Understanding the Python GIL.” [Online]. Available: http://www.dabeaz.com/python/UnderstandingGIL.pdf. [Accessed: 30-May-2019].

[4] “Borůvka's algorithm,” Wikipedia, 07-Feb-2019. [Online]. Available: https://en.wikipedia.org/wiki/Borůvka's_algorithm. [Accessed: 31-May-2019].

[5] S. Y. Cheung, “Prim's Algorithm for finding Minimum cost Spanning Tree,” Department of Math/CS - Home. [Online]. Available: http://www.mathcs.emory.edu/~cheung/Courses/171/Syllabus/11-Graph/prim2.html. [Accessed: 31-May-2019].

[6] M. Choquette, “Parallel Minimum Spanning Tree,” CMU 15-418/618 (Spring 2013) Final Project. [Online]. Available: http://www.andrew.cmu.edu/user/mchoquet/finalreport.html. [Accessed: 31-May-2019].

[7] D. Eppstein, “ICS 161: Design and Analysis of Algorithms Lecture notes for February 6, 1996,” Minimum spanning trees. [Online]. Available: https://www.ics.uci.edu/~eppstein/161/960206.html. [Accessed: 31-May-2019].

[8] “Heapsort,” Wikipedia, 16-Apr-2019. [Online]. Available: https://en.wikipedia.org/wiki/Heapsort. [Accessed: 05-Jun-2019].

[9] dtlafever, “How to generate a random connected graph?,” Algorithms Q&A, 14-Nov-2017. [Online]. Available: https://www.notexponential.com/482/how-to-generate-a-random-connected-graph?show=487. [Accessed: 31-May-2019].

[10] “Kruskal's algorithm,” Wikipedia, 15-May-2019. [Online]. Available: https://en.wikipedia.org/wiki/Kruskal's_algorithm. [Accessed: 31-May-2019].

[11] D. Lawrie, “Kruskal's and Boruvka's Algorithms.” [Online]. Available: http://www.cs.loyola.edu/~lawrie/CS302/S12/lecture/302-23.pdf. [Accessed: 30-May-2019].

[12] “Minimum Spanning Tree Tutorials & Notes | Algorithms,” HackerEarth. [Online]. Available: https://www.hackerearth.com/practice/algorithms/graphs/minimum-spanning-tree/tutorial/. [Accessed: 31-May-2019].

[13] B. Parker, “Planning Analysis: Calculating Growth Rates,” 30-Sep-2002. [Online]. Available: https://pages.uoregon.edu/rgp/PPPM613/class8a.htm. [Accessed: 03-Jun-2019].

[14] “Prim's algorithm,” Wikipedia, 27-May-2019. [Online]. Available: https://en.wikipedia.org/wiki/Prim's_algorithm. [Accessed: 31-May-2019].

[15] “Python.doc Runtime Services.” Python Software Foundation.

[16] Snicolas, “When should I use Kruskal as opposed to Prim (and vice versa)?,” Stack Overflow, 16-Jan-2014. [Online]. Available: https://stackoverflow.com/questions/1195872/when-should-i-use-kruskal-as-opposed-to-prim-and-vice-versa. [Accessed: 31-May-2019].

[17] N. Yadav, “Kruskal's Minimum Spanning Tree Algorithm | Greedy Algo-2,” GeeksforGeeks, 27-Sep-2018. [Online]. Available: https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/. [Accessed: 31-May-2019].

[18] N. Yadav, “Boruvka's algorithm | Greedy Algo-9,” GeeksforGeeks, 14-Aug-2018. [Online]. Available: https://www.geeksforgeeks.org/boruvkas-algorithm-greedy-algo-9/. [Accessed: 31-May-2019].

[19] S. Yadav, “Minimum Spanning Tree,” "ASPIRANTS", 15-Nov-2016. [Online]. Available: https://sameer9247.wordpress.com/2016/11/15/minimum-spanning-tree/. [Accessed: 31-May-2019].