Abstract

(Write when finished with paper.)

Introduction

Prim’s Minimum Spanning Tree algorithm is an algorithm that is used to determine the shortest path between the all vertices of a weighted undirected graph. A minimum spanning tree is a subset of a graph’s edges that connects all of the graph’s vertices, doesn’t contain any cycles, and has the minimum possible total edge weight for the given graph.

Minimum spanning trees have been used in a wide variety of modern and historical applications. Minimum spanning trees have been historically used to calculate the minimum amount of wiring necessary to connect a neighborhood to the electric grid. Minimum spanning trees are also used as an efficient way to generate mazes. While a properly implemented Prim’s algorithm results in the minimum spanning tree regardless of which node is chosen as the starting vertex, the underlying implementation of the algorithm can vary depending on the data structures chosen. In the following sections, the efficiency of two different implementations of Prim’s algorithm will be tested.

I have written two versions of the same program to generate the minimum spanning tree from a graph file. I have implemented the programs in Java. The major difference between the two programs is that in PrimSTL I have written the program and the algorithm using Java’s standard library functions wherever possible. In the other program, PrimDS, I forwent any Java standard library data structures in lieu of my own custom data structure implementations. I will experiment to determine the differences, if any, in the execution speed of these two algorithms.

Implementation

The first consideration one must make in implementing any graph algorithm is how the graph will be represented. The two most common representations are the adjacency matrix and the adjacency list. My implementation makes use of an adjacency list. In an adjacency list, the graph is stored as a list of vertices. Each index of the array, contains a list of all adjacent vertices. The figure below shows how an undirected graph is translated into an adjacency list.

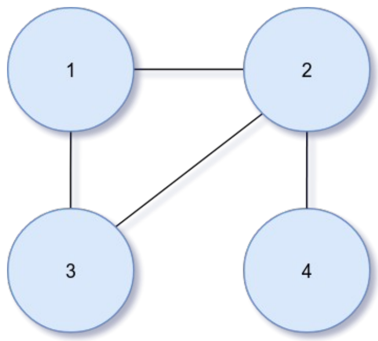


Figure 2: Simple undirected graph

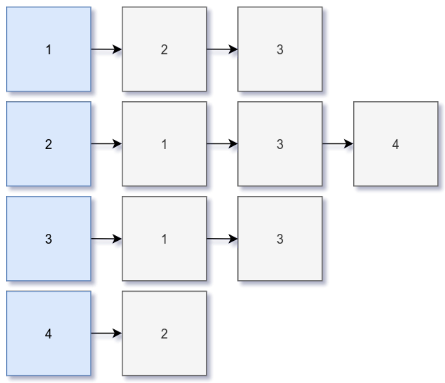


Figure 1: Adjacency list representation of the undirected graph.

As you can see, each vertex in the array has its own array containing all adjacent vertices. The adjacency list representation of the graph was selected because it makes it easy to enqueue alll adjacent edges of a specific vertex. You simply locate the vertex in the array and retrieve the array of adjacent vertices stored there. Since, I never had to access individual vertices in the adjacent array, I stored the adjacent vertices using a LinkedList. This allowed my to quickly retrieve all adjacent vertices.

The next choices one must make are algorithm specific. In order to understand these choices, the algorithm must first be understood. The implementation of Prim's algorithm that is used in both the PrimDS and PrimSTL programs is as follows:

1. *Choose a source vertex and add it to an empty array (minimum spanning tree).*

2. *Get all edges that are adjacent to the source vertex and add them to a queue.*
3. *Loop until the queue is empty.*
    1. *Remove the least cost edge from the queue.*
    2. *If the vertex it connects to is not yet in the minimum spanning tree, add it and enqueue all edges that are adjacent to the new vertex.*
    3. *If the vertex it connects to is already in the spanning tree, ignore it.*
4. *Once the loop is finished, you have the Minimum Spanning Tree.*

The above algorithm necessitates the choice of a few different data structures. I will discuss the choices I made in my implementations. The set of vertices (MST) is implemented as a normal Java array of integers. An array was chosen because the length of the MST is always the same length and the total number of vertices in the graph and is known once the graph is loaded as an adjacency matrix before the algorithm is run.

The queue is used to determine the lowest cost edge, so naturally a priority queue is the most logical choice. Where the priority queue gets interesting is in the choice of the underlying data structure that sorts the elements by the lowest weight edges. My implementation of the priority queue makes use of a binary min heap. The binary heap maintains a heap property where the edge with the lowest weight is always located at the root node. When a new node is inserted, the priority queue “heaps up” until the heap property is restored. Once the root node is removed (lowest cost edge removed), the priority queue “heaps down” until the heap property is restored.

A priority queue backed by a binary min heap is fast and efficient. Removing the minimum edge from the queue has a time complexity of O(1) and inserting a new edge into the priority queue has a time complexity of O(log n). Another advantage of the priority queue backed by a binary heap is that you can easily use an array to store values instead of a binary tree. Using a clever set of rules, you can use a node's index in the array to determine its parent node, left child node, and right child nodes as easily as you would if you had an actual binary tree implementation. The formulas to determine the indices are as follows:

Right Child: $2i + 2$

Left Child: $2i + 1$

Parent: $i – 1 / 2$

In the PrimDS program, I use my own priority queue backed by a binary min tree called EdgePriorityQueue that makes use of the above array implementation. In the PrimSTL program I simply use the PriorityQueue class from Java's standard library. It is important to note that behind the scenes, Java's PriorityQueue is also backed by a binary heap.

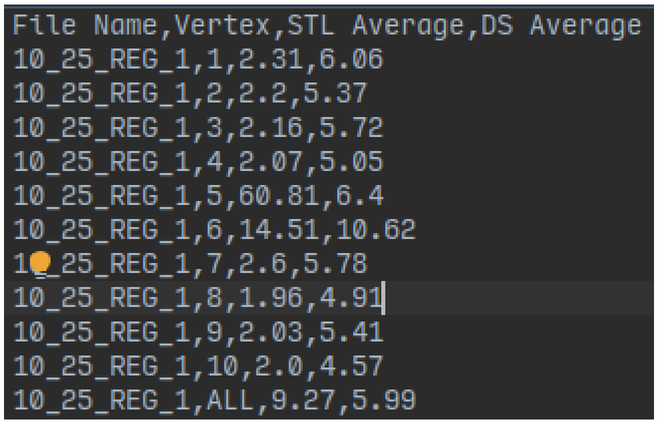
Testing

Both implementations of Prim's algorithm keep track of their execution time in microseconds. Once the graph is read from the input file and translated into an adjacency matrix object, the algorithm function is invoked. The function makes use of Java's System.nanoTime() function to get the current system time with nanosecond accuracy. The algorithm function saves the system time when the array to hold the MST is instantiated but before the source vertex is added. Once the algorithm has finished calculating the minimum spanning tree, the current system time is taken again.The finish time is then subtracted from the start time to get the total execution time in nanoseconds. To convert the nanoseconds to microseconds, the time is simply divided by 1000. The time in microseconds is printed to the terminal and returned as a Java long type from the function.

This basic implementation works well for small scale testing, but in order to get accurate results the algorithms should be run many times in order to calculate the average speeds. To test the algorithms at scale, you need to use some sort of test runner to run the tests by itself many different times. One's first idea may be to write a simple script that executes the Java program from the command line. This would not be a bad idea if you were using a different language, but Java requires your tests to be conducted slightly differently. Java makes use of a Virtual Machine (VM), this virtual machine is what allows the same Java code to run on many different devices. The downside of having a VM is that it takes some time to warm up before it runs at maximum efficiency. If the tests were run one at a time, the VM would never have time to warm up before shutting down and starting over again. To test Java programs it is recommended to create a "driver" in Java that creates the required objects and runs the tests. Since the program doesn't stop and start for each test causing excessive VM start and shutdown overhead, the resulting test times are much lower. In order to test the two implementations at scale, I wrote a custom test driver to run the two algorithm implementations many times and save the results.

The driver class receives the generated graphs output directory, the number of iterations to run, and a random seed as command line arguments. Once the driver starts, it generates four different graphs. There are two fully connected graphs with 5 and 10 vertices and two non-fully connected graphs with 5 and 10 vertices. The non-fully connected graphs use the random seed to generate each vertex with the same random number of connection. These graph files are saved to the directory supplied as a command line argument. After the graph files are generated, the testing starts.

I have written a class called AverageBenchmark that runs the tests for a set number of iterations and saves the results to a comma separated values (CSV) file. AverageBenchmark also simplifies testing by calculating each algorithm's averages before writing to the CSV file. This allows me to generate charts without having to crunch all the numbers of each test manually. The class calculates average execution speed of each algorithm when using each vertex as the source. It then takes all those averages and computes the average algorithm speed for the entire graph. The results are all outputted to a formatted CSV file that can then be imported into Excel to generate visuals.



```
File Name,Vertex,STL Average,DS Average
10_25_REG_1,1,2.31,6.06
10_25_REG_1,2,2.2,5.37
10_25_REG_1,3,2.16,5.72
10_25_REG_1,4,2.07,5.05
10_25_REG_1,5,60.81,6.4
10_25_REG_1,6,14.51,10.62
10_25_REG_1,7,2.6,5.78
10_25_REG_1,8,1.96,4.91
10_25_REG_1,9,2.03,5.41
10_25_REG_1,10,2.0,4.57
10_25_REG_1,ALL,9.27,5.99
```

Figure 3: Sample CSV file format for a non-fully connected graph with 10 vertices and 1 connection per vertex using a random seed of 25 over100 test iterations.

Implementing my own test driver to run the algorithms and calculate the benchmarks allowed me to run many more test iterations because I didn’t have to worry about manually calculating the averages.

When running a benchmark, one must take into consideration the system on which the test will be run. Running a benchmark on one’s own computer may lead to many inaccuracies because there are other processes unrelated to the benchmark that are competing for CPU time. For this algorithmic analysis I ran all the benchmarks on an isolated testing machine called “mcgonagall”. This machine doesn’t have any other users or background processes that may skew results. At the time of testing, I was the only user logged in to the system. Therefore, I believe the tests are about as accurate as possible and far more accurate than if I were to run the tests on my local machine.

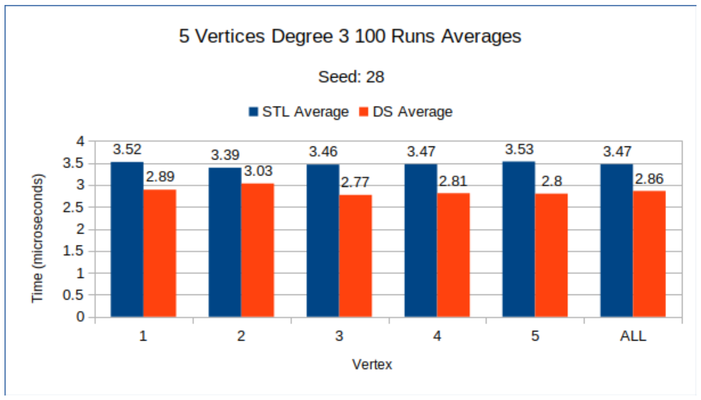
Results

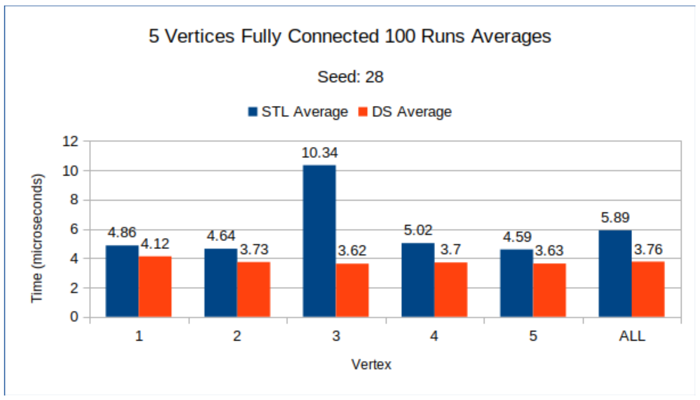
I benchmarked both implementations of the algorithm with the following parameters:

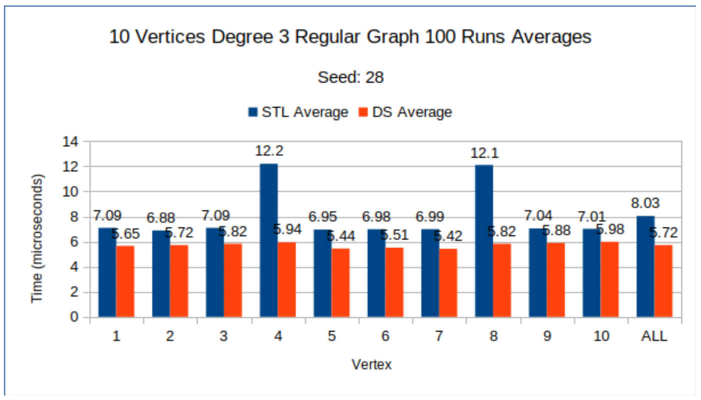
- Random Seed: 28
- Graphs
    - Fully Connected: 5 and 10 vertices
    - Non-fully Connected: 5 and 10 vertices
- Iterations: 100, 1,000, 10,000

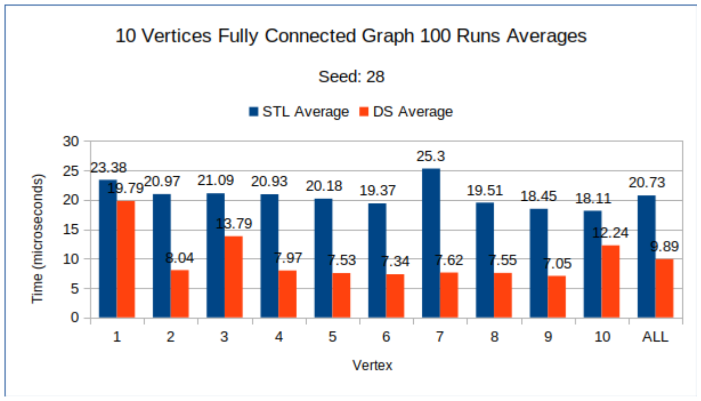
The following charts display the results for each implementation and each trial run of iterations.

100 Runs

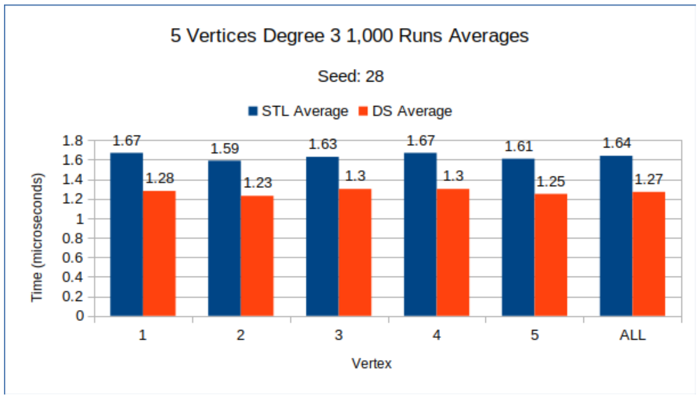


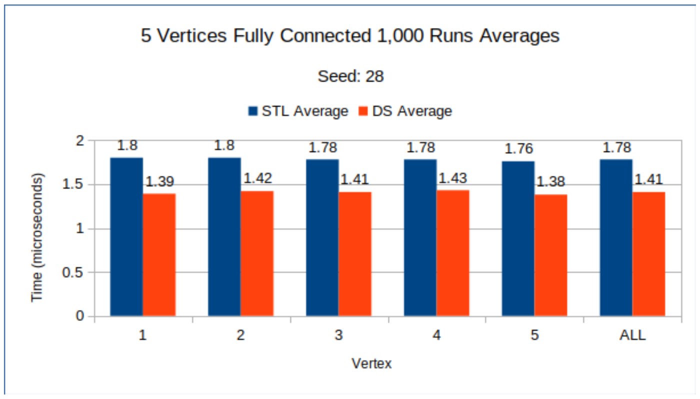


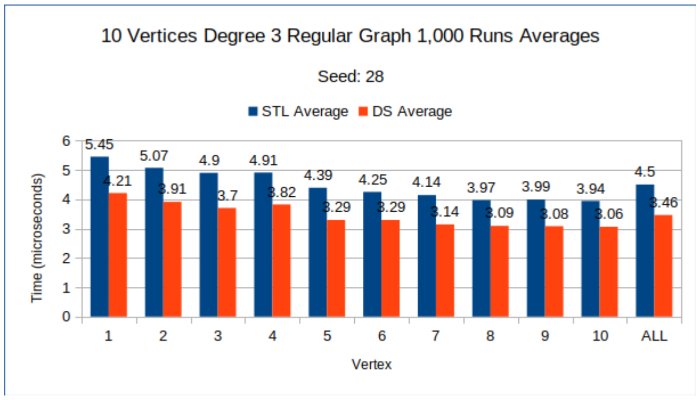


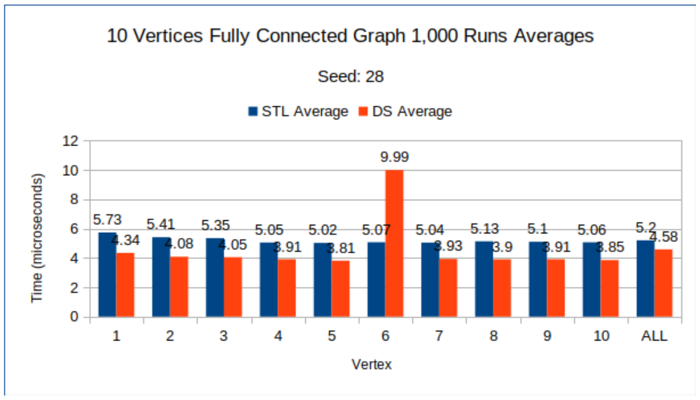


1,000 Runs

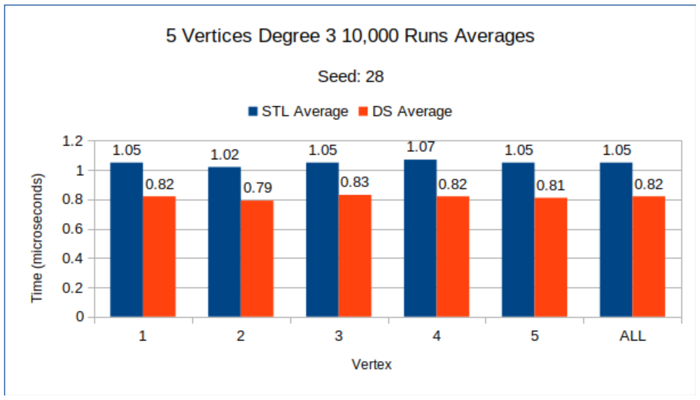


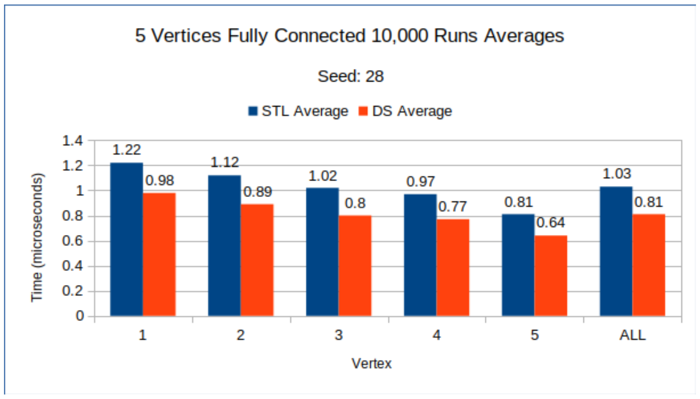


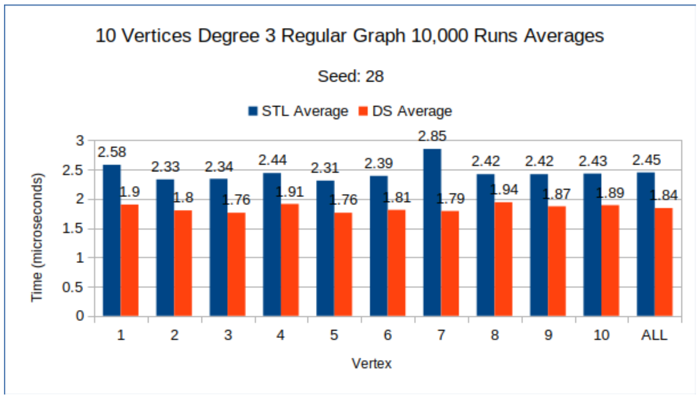


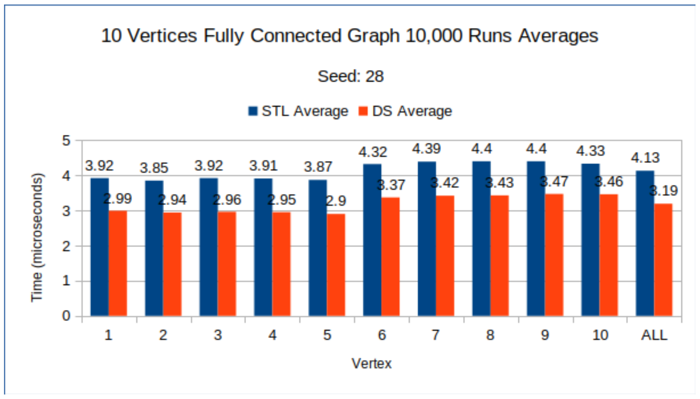


10,000 Runs

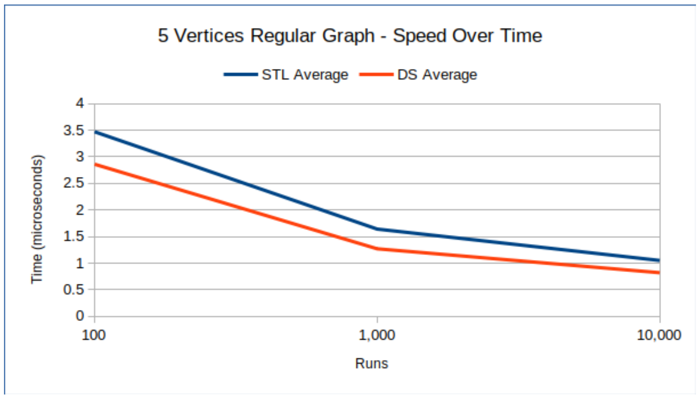


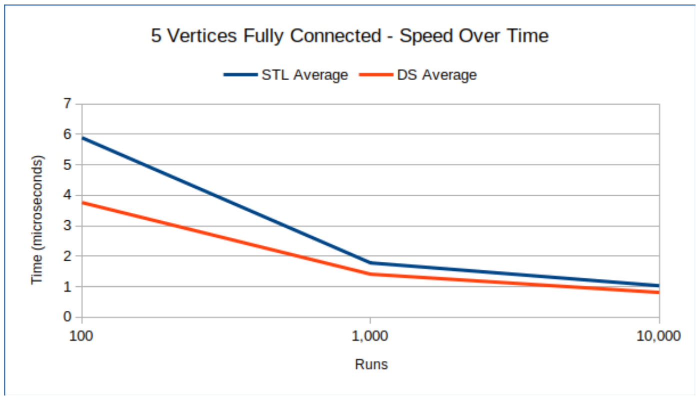


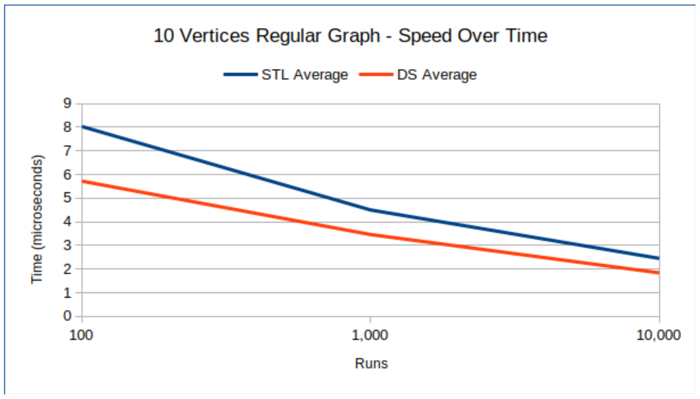


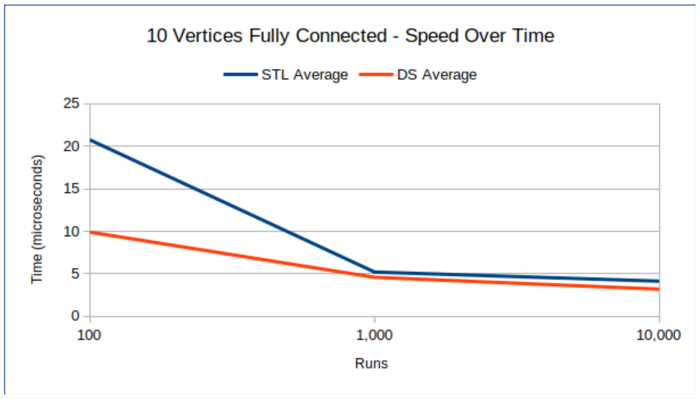


Average Algorithm Speed Over Increased Iterations









Conclusions

Citations