Abstract

This paper analyzes two implementations of Prim's minimum spanning tree algorithm to determine what, if any, differences there are between the speed and efficiency of the algorithm when implemented with Java 8 data structures versus custom data structures. To answer this question, the performance of both implementations is measured and compared. Testing graphs of various sizes and types, the results show that the custom data structures version of Prim's algorithm is consistently faster than the implementation that uses Java's standard library's data structures. When implementing Prim's algorithm one must keep these discrepancies in mind in order to maximize performance.

Introduction

Prim's Minimum Spanning Tree algorithm is an algorithm that is used to determine the shortest path between all of the 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 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 source 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 using Java 8. The only difference between the two programs is that in PrimSTL I have written the algorithm using Java's standard library data structures wherever possible. In PrimDS, I forwent any Java standard library data structures in lieu of my own custom data structure implementations. This paper will analyse the difference, if any, in the efficiency and execution speed of these two implementations.

Implementation

The first consideration one must make in implementing any graph algorithm is how the graph itself will be represented. The two most common data structure 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 vertex of the graph occupies an index in the array. At each index (vertex), there is another array containing all vertices that are adjacent to the vertex given at the index. The figure below shows how an undirected graph is translated into an adjacency list.

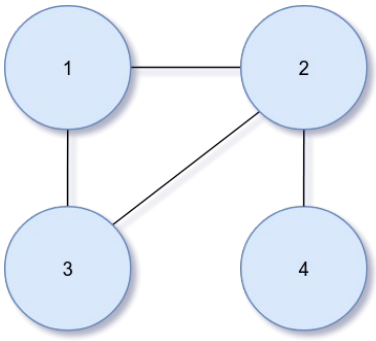


Figure 1: Simple undirected graph

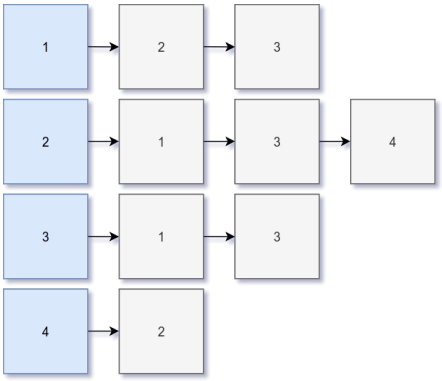


Figure 2: Adjacency list representation of the undirected graph.

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

all adjacent edges of a specific vertex. You simply locate the vertex in the array and retrieve the array of adjacent vertices stored there. Since, you never need to access individual adjacent vertices in Prim's algorithm, I stored them using a LinkedList object. Linked lists also have the advantage of being non-contiguous and not having a size constraint. Each vertex may have a different number of adjacent vertices.

The next data structure 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 both of 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 equal to the number of vertices in the graph and the total number of vertices in the graph is known once the adjacency matrix is created.

The queue is used to determine the lowest cost edge currently available, so naturally a priority queue is the most logical data structure choice. Where the priority queue gets interesting

is in the choice of the underlying structure that sorts the elements by the lowest weight edges. My custom implementation of the priority queue for PrimDS 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 tree implementation. The formulas to determine these 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. The PriorityQueue contains Edge objects so I had to supply a Comparator that tells the PriorityQueue how to compare Edge objects to get the least cost Edge. It is important to note that behind the scenes, Java's PriorityQueue is also backed by a binary heap.

Testing

Both Java 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() method to get the current system time with nanosecond accuracy. The algorithm function saves the starting system time once the array to hold the resulting minimum spanning tree is instantiated but before the source vertex is added. Once the algorithm has finished calculating the minimum spanning tree, the 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, each algorithm should be run many times to get a an average speed. To test the algorithms at scale, you need to use some sort of test runner to run the tests many different times. One's first idea may be to write a simple script that executes the Java program from the command line via the "java [program name]" command. This would not be a bad idea if you were using a different language like C or C++, but Java requires your tests to be conducted slightly differently for better accuracy.

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 which is basically a separate class

that creates the required objects and runs the algorithms. Since the program doesn't stop and start for each test, the resulting tests run much faster due to the lack of startup and shutdown overhead. In order to test the two implementations at scale, I wrote a custom test driver to run the two algorithm implementations many times and store the results.

The driver class receives the a file directory that the graphs should be created in, 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 a randome degree of adjacent vertices. These graph files are saved to the designated graph directory. After the graph files are generated, the algorithm testing begins.

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. This runs the tests for any file found in the specified directory, in this case that directory contains the generated graph files. AverageBenchmark also simplifies testing by calculating each the averages for each algorithm before writing the test results to a CSV file. This allows me to generate charts without having to crunch all the numbers of each test manually. AverageBenchmark runs both implementations on each vertex of the given graph (sets the vertex as the source) and calculates the average algorithm speed for each vertex. It then takes the per vertex 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 over 100 test iterations.

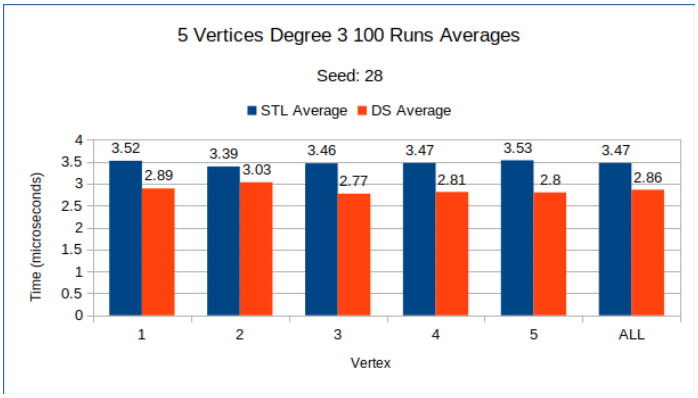
Implementing my own test driver to run the algorithms and calculate the benchmarks allowed me to run a very large amount of test iterations because I didn't have to worry about manually calculating the averages.

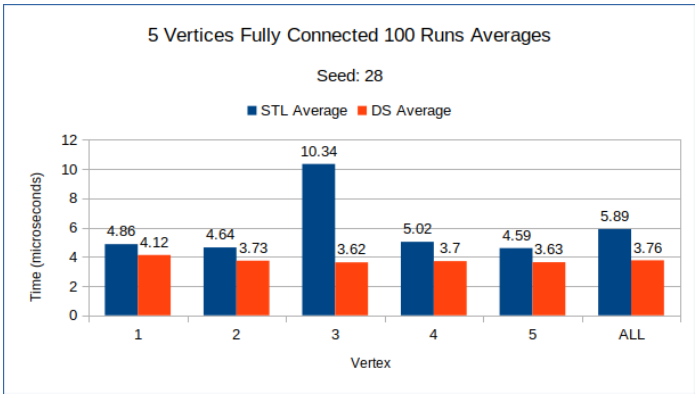
One final consideration must be made. When running a benchmark, the system that the test will be run is very important to the accuracy and validity of the test. Running a benchmark on one's own computer may lead to many inaccuracies because there are processes unrelated to the benchmark that are constantly 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 running that may skew results. At the time of testing, I was the only user logged in to the system. Therefore, I believe the test results are about as accurate as possible and far more accurate than if I were to run the tests on my local machine.

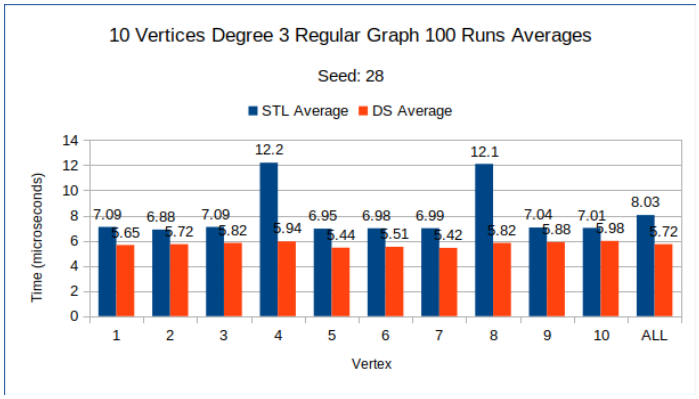
Results

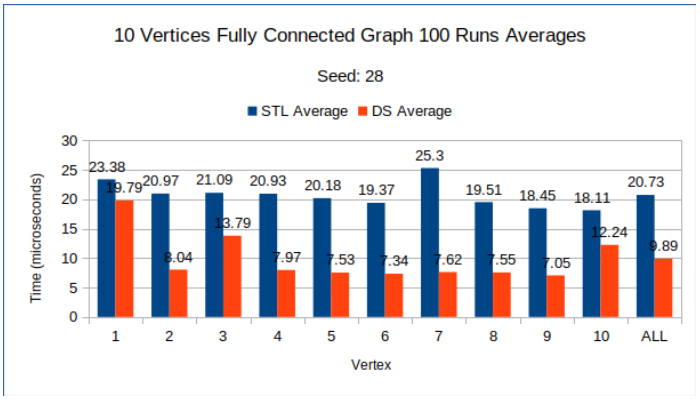
I benchmarked both implementations of the algorithm three different times. In each benchmark I used a random seed of 28. This keeps all "random" values the same across iterations and makes the results reproducible by others who use the same seed. Each benchmark tested both algorithms on 4 different graphs. It was run on both fully-connected and regular graphs with 5 and 10 vertices. In the case of the regular graphs, each vertex had a degree of 3. The degree of each vertex was derived from the random number generator using the given seed (28). Lastly, the three different benchmarks used different numbers of iterations. The benchmark was run three times with 100 runs, 1,000 runs, and 10,000 runs. The resulting test results were saved CSV files which were imported into Excel and used to generate visuals. The following charts display the results for each algorithm implementation and benchmark run. The "STL Average" is using Java's data structures and "DS Average" is using the custom data structures.

Benchmark 1 - 100 Runs



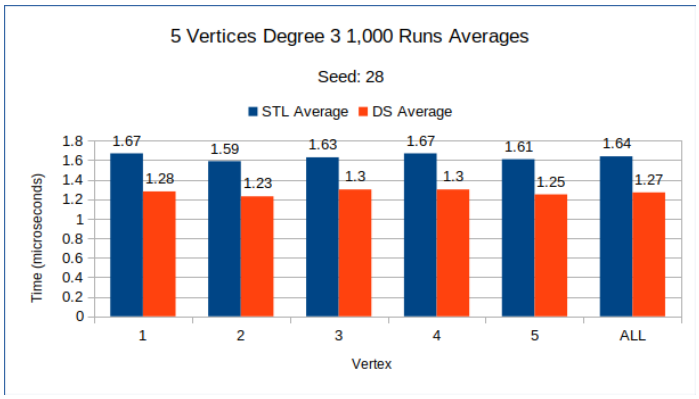


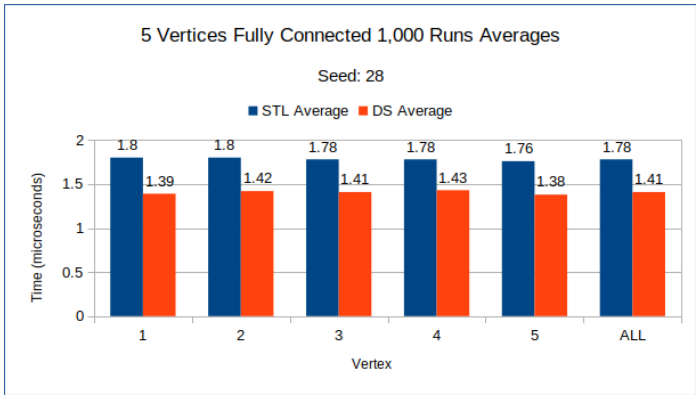


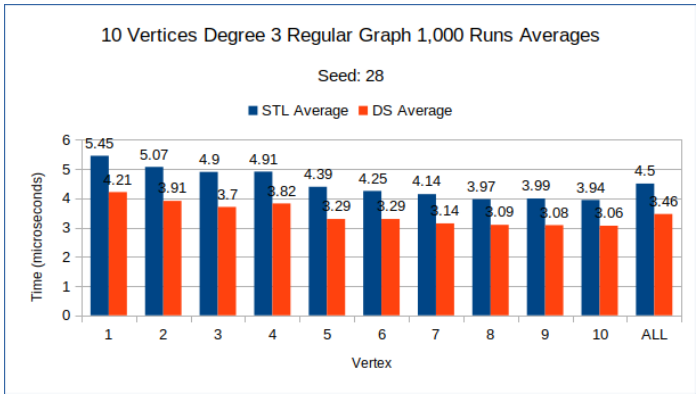


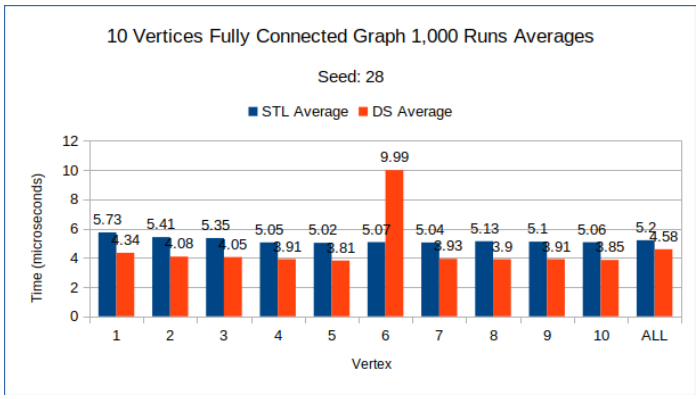
Running each graph only 100 times, you can clearly see that the custom implementation of Prim's algorithm (DS Average) written with my own priority queue is consistently faster than the implementation that uses Java's standard data structures. The 10 vertices graphs are much slower than the 5 vertices graphs and there is a large difference in speed between the fully connected and degree 3 graphs for both 5 vertices and 10 vertices. The most surprising result of this benchmark is how much faster the DS implementation is over the STL implementation for the 10 vertices fully connected graph. On this benchmark, that is where there seems to be the largest discrepancy between the two Prim's implementations. Also, on each graph there are some vertices that run much slower for no discernible reason. I consider these to be outliers as the corresponding opposite algorithm implementation did not show any noticeable slowdown.

Benchmark 2 - 1,000 Runs



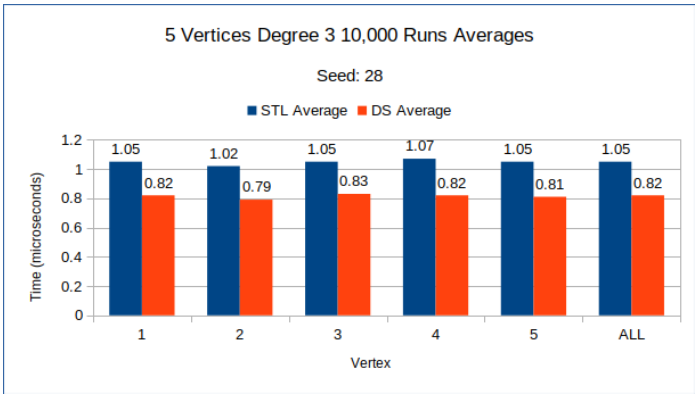


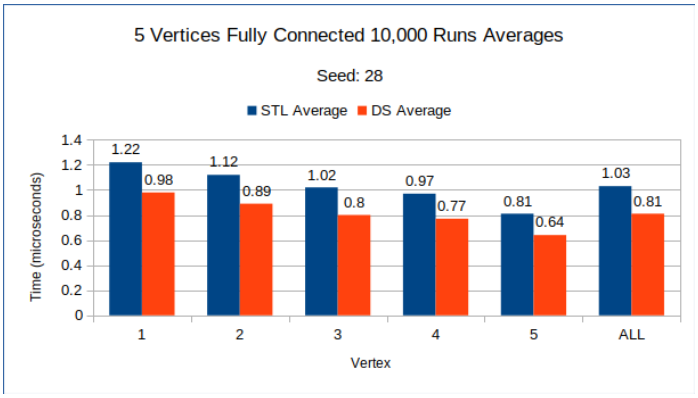


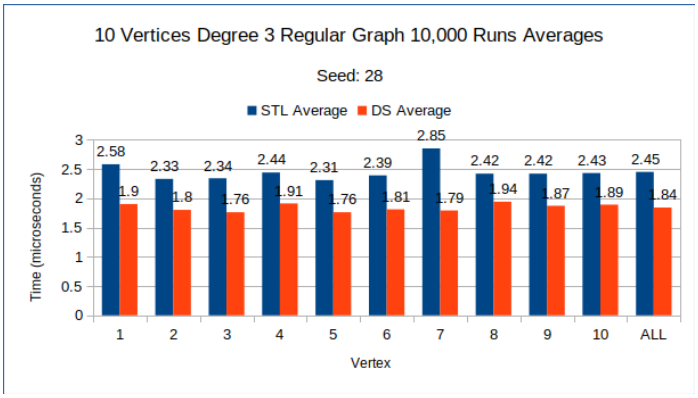


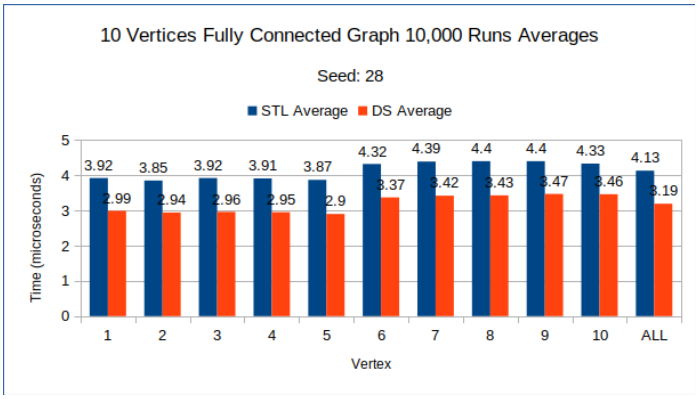
Running each graph 1,000 times, the results remain much the same as benchmark 1.The custom DS implementation performed better than the STL implementation in every test except for one. In the 10 vertices fully connected graph results, there is a single outlier that is opposite the outliers from the previous benchmark; in this case the DS implementation is much slower. The most interesting difference between this benchmark and the previous benchmark of 100 iterations is how much faster the algorithm's speed has become for both implementations. The full graph averages (ALL vertex) have sped up and run about 2 times faster compared to the previous benchmark.

Benchmark 3 - 10,000 Runs



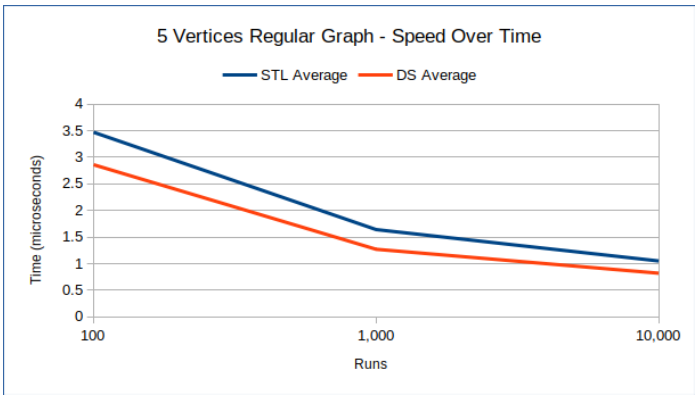


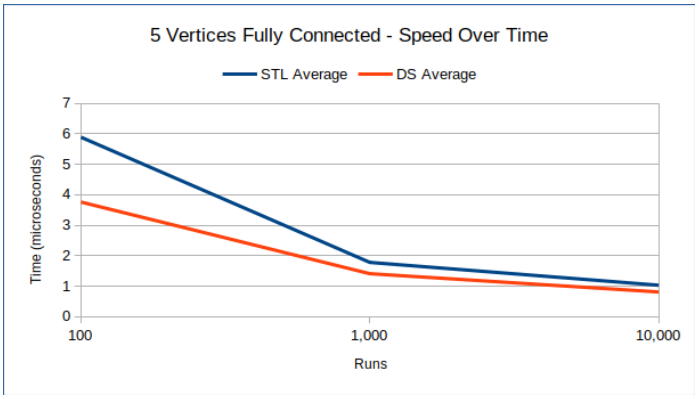


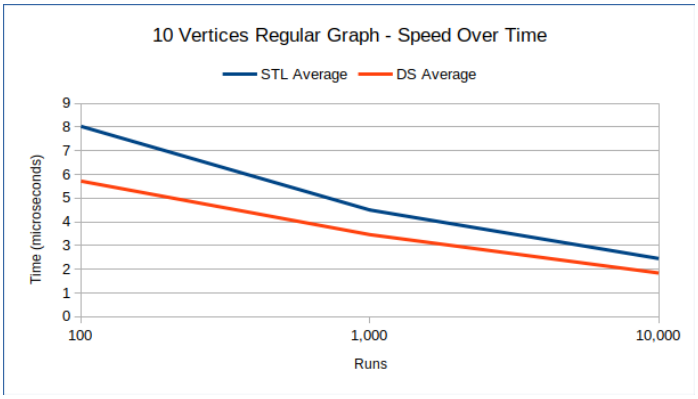


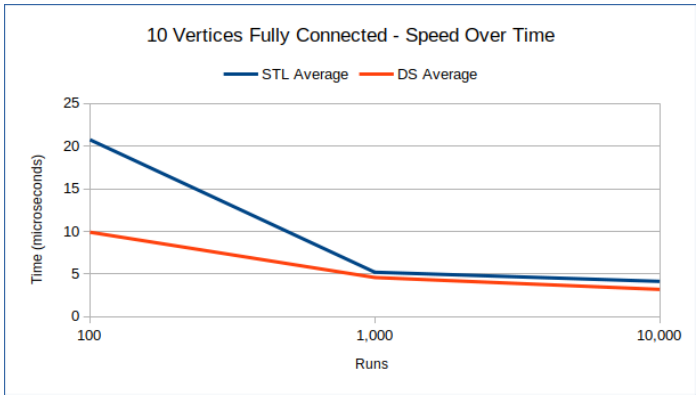
Finally, running each graph 10,000 times shows much the same as benchmark 2. In this benchmark, there are no outliers and the DS implementation is consistently faster just like the previous benchmarks. Interestingly, the algorithm speed is even faster with 10,000 runs than the 1,000 runs in benchmark 2.

Average Algorithm Speed Over Iterations









Perhaps the most interesting finding of the benchmark tests was just how much faster both algorithms sped up as the number of iterations increased. I created these graphs to visually see just how much faster the algorithm runs as the number of iterations increased. There is a very large change in algorithm speed as the iterations goes from 100 runs to 1,000 runs. Going from 1,000 runs to 10,000 runs, the algorithms continue to speed up but not as drastically. You can also see that as the runs increase, the gap between the speeds of both implementations gets smaller and smaller.

To make sense of the benchmark results I had to do a little bit of research. I believe that the drastic increase in speed as the number of iterations increased was a side effect Java's JIT compilation. JIT stands for "just in time" and it is the method that Java uses to compile bytecode

into native instructions for the specific device the code is being run on. In order to speed up compilation times, the JIT compiler is only run once the program is started and it compiles the code "just in time" for it to be run. When one writes a java class and compiles it, it is converted to ".class" files which is Java's bytecode. It must be compiled into bytecode so that you can run it on any system with Java installed. Unfortunately, computers cannot make use of bytecode so it must be compiled into native machine code. Compiling all of the code at runtime leads to slow execution speed, so JIT is used to only compile the code that will be run. On small scale tests, the JIT compilation time is very long in relation to the number of tests run, so the averages are skewed. As more tests are performed, the effect that the initial compilation to machine code has is reduced.

As you can see from the results, my own data structures implementation was consistently faster throughout all benchmarks and tests. This is interesting as my first thought would be that the "standard" implementations would be heavily tuned and optimized by very smart people to be the best possible implementation. On second thought I can understand why my own implementation was faster. All the standard library data structures are designed to be generic. They have to work with whatever type or object the programmer needs. As a result this overhead slows them down. In my own data structure implementation of a priority queue, it was tuned specifically to hold Edge class objects. Using Java's PriorityQueue class you must supply a Comparator object in order to make it work with your own classes and this adds overhead. Also, my own EdgePriorityQueue class only has three methods; add(), removeLeast(), and checkEmpty(). Looking at the Java 8 documentation there are 13 methods. For these reasons I believe that the custom data structures implementation of Prim's algorithm was faster than the Java standard library implementation of Prim's algorithm.

Conclusion

Prim's algorithm is an efficient and easy to understand way to generate a minimum spanning tree of a weighted and undirected graph. After testing two implementations of Prim's algorithm, one with Java's standard data structures and one without, I can say that it remains fairly efficient no matter what type of undirected graph it is run on. I can also say that if maximizing performance is of utmost importance, you should implement your own customized data structures. The program written with custom data structures consistently performed better than the program using Java's standard library. One final thing to keep in mind when using Java to write high performance code, is the number of iterations. You must keep in mind that Java requires some warm up time and that the more iterations you run, the better your programs will perform.

Source Code

You can view the code for both implementations as well as the testing benchmarks here:

https://github.com/evan-buss/CSC402-Project2

Citations

https://aboullaite.me/understanding-jit-compiler-just-in-time-compiler/

https://visualgo.net/en/mst

https://www.programiz.com/dsa/prim-algorithm

https://archive.org/details/bstj36-6-1389