Programming Assignment 1:

Minimum Spanning Trees for Random Complete Graphs

Justin Gonzalez and Carlos Robles

February 29, 2020

1 Implementation

1.1 Algorithm Choice

For this assignment, we chose to implement Prim's algorithm for finding a minimum spanning tree. Let $m = |E|$ and $n = |V|$. As we have seen in class, the run time for Prim's algorithm is $O(m \times \text{insert} + n \times \text{deletemin})$. If we implement Prim's algorithm with a binary heap, each insert and deletemin operation is $O(\log n)$ and results in a total run time of $O(m \log n + n \log n) = O(m \log n)$. Since we are working with complete graphs, the number of edges in our graph is $m = \binom{n}{2} = \frac{n(n-1)}{2} = O(n^2)$. Thus, Prim's algorithm with a binary heap will result in a run time of $O(n^2 \log n)$. If we implement Prim's algorithm with a list, each insert operation is $O(1)$ and each deletemin operation is $O(n)$. This results in a run time of $O(m + n^2) = O(2n^2) = O(n^2)$. Therefore, Prim's algorithm with a list implementation is faster for complete graphs.

In comparison, Kruskal's algorithm has a run time of $O(\text{sort}(m) + m \times \text{find} + n \times \text{union})$. As we have shown in class, this evaluates to a total run time of $O(m \log m + (m + n) \log^* n)$ which is equal to $O(m \log m)$ as $\log^* n = o(\log m)$. Since $m = O(n^2)$, this results in a run time of $O(n^2 \log(n^2))$ which is slower than our $O(n^2)$ algorithm using Prim's algorithm with a list implementation. Therefore Prim's algorithm with a list implementation is the fastest of the given algorithms for finding the minimum spanning tree on a complete graph.

1.2 Time Complexity

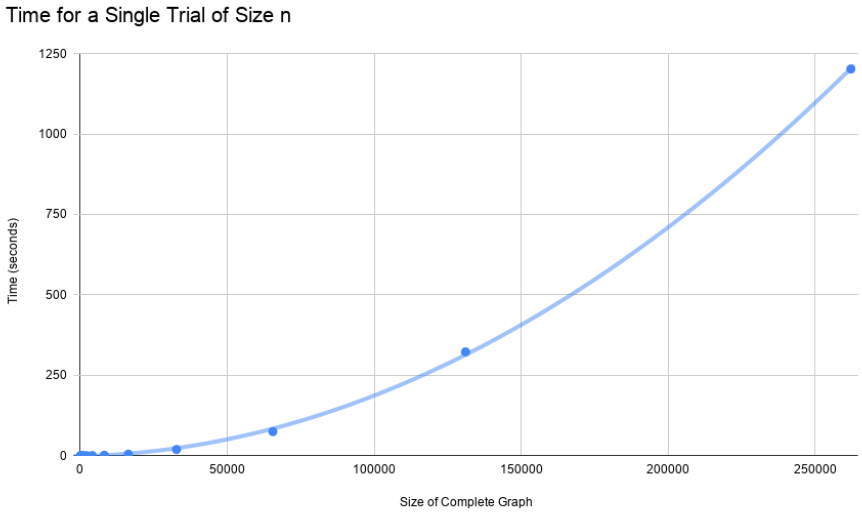

Figure 1: Completion time for a single trial of size n and dimension 2

As before mentioned, Prim's algorithm with a list implementation has an asymptotic run time of $O(n^2)$. In practice, our algorithm behaves as we would expect with our running time growing quadratically. When we ran our algorithm for all $n \in \{128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144\}$ and found the average MST size across 5 trials for each dimension $d \in \{0, 2, 3, 4\}$, our program took 9 hours to run. On any given dimension, running our algorithm 5 times took about 1 hour and 40 minutes for $n = 262144$, 25 minutes for $n = 131072$, 6 minutes for $n = 65536$, 1.5 minutes for $n = 32768$, and so on. As we can see, as our input size doubles, our run time quadruples, thus indicating quadratic growth.

1.3 Space Complexity

One of the main challenges we came across in implementing our algorithm was figuring out how to efficiently store our random complete graphs. Both of the traditional representations of a graph (matrix and adjacency list) took up $O(n^2)$ space as we are storing a complete graph with $O(n^2)$ edges. For our large values of n, this would take up tens of gigabytes of storage and cause our system to

run out of space. To work around this, we decided that for dimensions 2 and higher, we would store the random coordinates associated with each vertex in an array. Whenever we need the weight of edge (v, w) we would simply run our $O(1)$ algorithm for computing the Euclidean distance between vertex v and vertex w. This works since by the definition of a complete graph we know that each vertex is connected to every other vertex. (We do not run into the issue where $w = v$ since v is in S and w must be in $V - S$ by the Cut Property.) Storing just the coordinates for each vertex results in $O(dn)$ space, which for our finite dimensions d is $O(n)$. In the case where $d = 0$, we actually do not store any information about the vertices or edges and just generate a random number in the interval $[0, 1]$ whenever we need the weight of the edge (v, w).

1.4 Randomness

To generate random edge weights for dimension 0 and random coordinates for dimensions 2, 3, and 4, we used Java's `java.util.Random` class. This is a pseudo-random number generator that returns a number (in our case in the interval $[0, 1]$) when given an input seed of 48 bits. When no input seed is given (as in our case), the seed is created using system nano time (https://www.journaldev.com/17111/java-random). Like other pseudo-random number generators, the numbers produced are not truly random as they are determined by the input seed using a deterministic algorithm. Since the seed is only 48 bits, Java's `java.util.Random` class can be cracked with fewer than 2^{48} guesses and should not be trusted for anything that depends on randomness for security, but for the purposes of our tests, it works well enough to help us generate our average approximations.

2 Results

2.1 Average Minimum Spanning Tree Sizes

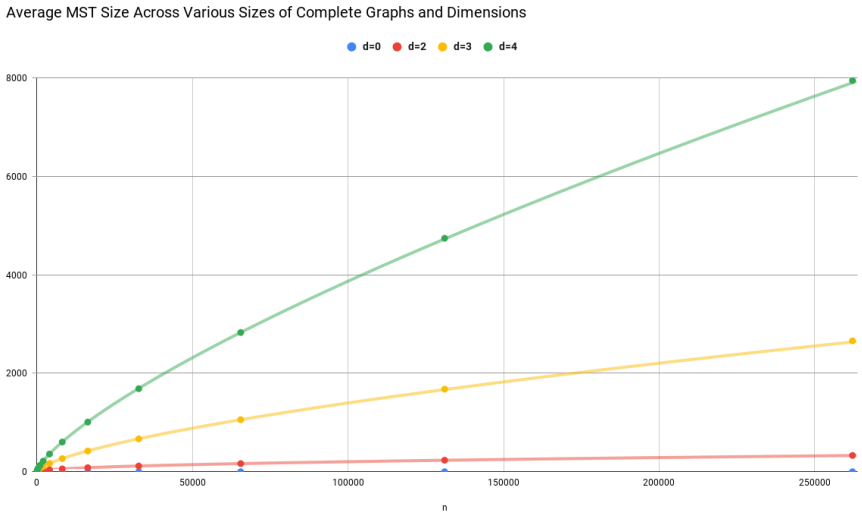

Figure 2: Average MST total weight for 0, 2, 3 and 4 dimensions

n	d=0	d=2	d=3	d=4
128	1.1552	7.6550	17.4523	27.9861
256	1.1962	10.5477	27.6275	46.3497
512	1.1947	14.9072	43.4551	77.8754
1024	1.1790	21.0339	68.2414	130.7181
2048	1.2037	29.6318	107.2267	216.8369
4096	1.2127	41.6852	169.2815	360.6239
8192	1.2087	58.9501	267.1752	603.6150
16384	1.2048	83.1522	422.1522	1009.1916
32768	1.2040	117.6035	667.8714	1687.7400
65536	1.2028	166.0760	1058.3898	2828.4914
131072	1.2028	234.7090	1677.6793	4742.4568
262144	1.2021	331.6873	2659.1776	7947.9732

Table 1: Average MST total weight for 0, 2, 3 and 4 dimensions

2.2 Approximating a Function for Average MST Size

By graphing the average minimum spanning trees produced by our algorithm, we were able to write a function $f(n)$ to accurately describe the relationship between input size, dimension, and expected MST size. The simplest function we created to approximate our algorithm's data was

$$f(n) = \begin{cases} 1.2 & d = 0 \\ \frac{2}{3}n^{(1-\frac{1}{d})} & d \geq 2 \end{cases}$$

where d is the dimension and n is the number of vertices in our graph.

This function works rather well. In order to test more closely the viability of our modeling function to match the MST weight data, we ran an R^2 statistical test. The results are as follows:

Dimension	R^2
2	0.998
3	0.999
4	0.987

Table 2: R^2 values for our MST weight modeling function

Based off of these results, as well as simply checking the closeness of values, we could see that our function matched the MST weight growth quite accurately.

2.3 Discussion of Findings

Looking at the raw results, some observations we were surprised by, and others made intuitive sense. The first thing that made intuitive sense was that average MST weight would increase with dimension, simply by the nature that maximum edge weight is equal to $\sqrt{d}$ which increases as d increases.

We did not expect to find that $\frac{2}{3}n^{1-\frac{1}{d}}$ would match the MST growth function, and it did not come up as a prediction. However, as we analyzed the function, multiple elements of the function make intuitive sense.

First, it seems intuitive that there would be some kind of growth with respect to n rather than decay or asymptotic stability. We see in our function that

$f(n)$ initially grows quickly and the growth rate decreases as n increases due to the $(1 - \frac{1}{d})$ term. This makes sense since adding an edge to a graph with fewer edges is likely to increase the MST size more drastically than adding an edge to a graph with more edges. (A graph with more edges is more likely to have more edges that are relatively "light".) Also, since we are working with complete graphs, increase the size of a graph with k vertices to $k + 1$ vertices adds k edges. This gives our algorithm more edges to choose from, increasing the likelihood of choosing lighter edges.

As the dimension increases, it also makes sense that it would approach linear growth as it closes in on $2/3n^{1+0} = 2/3n$. This makes sense as we would expect the difference in growth rates between $d = 10001$ and $d = 10000$ to be rather minimal.

While that part of the function was quite effective in approximating the dimensions 2, 3, and 4, it is quite easy to see that that function simply does not work for $d = 0$, as it divides by 0 in the exponent. We turned our function into a piecewise function, and used the trend in the results to approximate it. As our dimension 0 MST increased in vertices, the MST weight was functionally the same as you can see in the table below:

n	dim 0 MST weight
128	1.1552
256	1.1962
512	1.1947
1024	1.1790
2048	1.2037
4096	1.2127
8192	1.2087
16384	1.2048
32768	1.2040
65536	1.2028
131072	1.2028
262144	1.2021

Table 3: Average MST total weight for 0 dimensions

From the table, we can see that MST total weight functionally stops increasing after 256 edges, instead it fluctuates closely around 1.2. We found this to be

reasonable, given that edge weights were uniformly distributed. Given that the number of edges grew at $\frac{n(n-1)}{2}$, while the number of edges required for an MST grew at $n-1$, it makes sense that the MST could choose $n-1$ necessary edges from the extremely lightweight minima of the uniformly distributed edge weights, such that additional edges functionally added no weight to the MST. As for why it arrives at around 1.2 specifically, our original hypothesis used order statistics of the uniform distribution, to find the sum of the expected weights of the $n-1$ lightest edges given n vertices, and we arrived at the formula

$$f(n) = \frac{(n-1) \cdot n}{n \cdot (n-1) + 2}$$

which results in convergence towards 1. This was empirically shown to be wrong, however this equation assumed the uniform distribution was valid in our case, but we now think it may be the case that as n increases, by the Central Limit Theorem, we may be required to assume that the distribution changes such that slightly larger edges are included, and produce this value of 1.2 on average.