CS 124 Programming Assignment 1: Spring 2024

Your name(s) (up to two): Andy Martinez

Collaborators: (You shouldn't have any collaborators but the up-to-two of you, but tell us if you did.)

No. of late days used on previous psets: 4
No. of late days used after including this pset: 6

Homework is due Wednesday Feb. 21 at 11:59pm ET. You are allowed up to **twelve** (college)/**forty** (extension school) late days through the semester, but the number of late days you take on each assignment must be a nonnegative integer at most **two** (college)/**four** (extension school).

Overview: The purpose of this assignment is to experience some of the problems involved with implementing an algorithm (in this case, a minimum spanning tree algorithm) in practice. As an added benefit, we will explore how minimum spanning trees behave in random graphs.

Assignment: You may work in groups of two, or by yourself. Both partners will receive the same grade and turn in a single joint report. (You should submit a single copy of the report, with both of you listed as submitters, in Gradescope.)

We recommend using a common programming language such as Java, C, or C++. If you use a more obscure language, that is fine, but if there are errors that are correspondingly harder to find it may cause your grade to be lower. We might advise you not to use a scripting language like Python, although many students successfully do the assignment in Python.

For any algorithms or data structures taught in CS 124 that you wish to use, you must write your own code. For instance, you may not use external libraries for heaps, priority queues, or finding MSTs outright, but you may use external libraries for, e.g., binary search trees, hashing (e.g. built-in Python dictionaries), or routine operations on arrays.

We will be considering *complete, undirected* graphs. A graph with n vertices is complete if all $\binom{n}{2}$ pairs of vertices are edges in the graph.

Consider the following types of graphs:

- Complete graphs on n vertices, where the weight of each edge is a real number chosen uniformly at random on $[0,1]$.
- Complete graphs on n vertices, where the vertices are points chosen uniformly at random inside the unit square. (That is, the points are (x,y), with x and y each a real number chosen uniformly at random from $[0,1]$.) The weight of an edge is just the Euclidean distance between its endpoints.
- Complete graphs on n vertices, where the vertices are points chosen uniformly at random inside the unit cube (3 dimensions) and hypercube (4 dimensions). As with the unit square case above, the weight of an edge is just the Euclidean distance between its endpoints.

Your goal in this Programming Assignment is to determine, in each of the three cases above, how the expected (average) weight of the minimum spanning tree (not an edge, the whole MST) grows as a function of n.

We give further guidelines on how to get to this goal in the rest of this handout. This will require implementing an MST algorithm, as well as procedures that generate the appropriate random graphs. You may implement any MST algorithm (or algorithms!) you wish; however, we suggest you choose carefully.

For each type of graph, you must choose several values of n to test. For each value of n, you must run your code on several randomly chosen instances of the same size n, and compute the average value for your runs. Plot your values vs. n, and interpret your results by giving a simple function $f(n)$ that describes your plot. For example, your answer might be $f(n) = \log n$, $f(n) = 1.5\sqrt{n}$, or $f(n) = \frac{2n}{\log n}$. Try to make your answer as accurate as possible; this includes determining the constant factors as well as you can. On the other hand, please try to make sure your answer seems reasonable.

Code setup:

So that we may test your code ourselves as necessary, we prefer that your code accepts the following command line form:

```
./randmst 0 numpoints numtrials dimension
```

Accepting that command line form is optional. If your code doesn't and we need to test your code, we may need to contact you.

The flag 0 is meant to provide you some flexibility; you may use other values for your own testing, debugging, or extensions. The value numpoints is n, the number of points; the value numtrials is the number of runs to be done; the value dimension gives the dimension. (Use dimension 2 for the square, and 3 and 4 for cube and hypercube, respectively; use dimension 0 for the case where weights are assigned randomly. Notice that dimension 1 is just not that interesting, and that "dimension 0" is not actually the 0-dimensional version of the 2-, 3-, and 4-dimensional cases.) The output for the above command line should be the following:

```
average numpoints numtrials dimension
```

where average is the average minimum spanning tree weight over the trials.

It is convenient for us in grading to be able to run the programs without any special per-student attention. The following three instructions make that easier for us, but don't spend more than a few minutes trying to satisfy them—we can contact you to ask about seeing the code run on your machine if necessary.

- If possible, for compatibility reasons, the code should run on a Unix/Linux system, even if you code on another system.
- We expect the code as described above in its own file(s) separate from your results/writeup.
- The code should compile with make; no instructions for humans. That is, the command "make randmst" should produce an executable from your directory. You may need to read up on makefiles to make this happen.

Note: you should test your program – design tests to make sure your program is working, for example by checking it on some small examples (or find other tests of your choosing). You don't need to put your tests in your writeup; that is for you.

What to hand in:

Besides *submitting a copy of the code you created*, your group should hand in a single well organized and clearly written report describing your results. For the first part of the assignment, this report must contain the following quantitative results (for each graph type):

- A table listing the average tree size for several values of n. (A *graph* is insufficient, although you can have that too; we need to see the actual numbers.)
- A description of your guess for the function $f(n)$.

Run your program for $n = 128$; 256; 512; 1024; 2048; 4096; 8192; 16384; 32768; 65536; 131072; 262144; and larger values, if your program runs fast enough. (Having your code handle up to at least $n = 262144$ vertices is one of the assignment requirements; however, handling n up to 131072 will result in only losing 1-2 points. Providing results only for smaller n will hurt your score on the assignment.) Run each value of n at least five times and take the average. (Make sure your experiments use independently generated randomness!)

In addition, you are expected to discuss your experiments in more depth. This discussion should reflect what you have learned from this assignment. You should, at a minimum, discuss the asymptotic runtime of your algorithm(s) and the correctness of any modifications from the standard algorithms presented in class; otherwise, the actual issues you choose to discuss are up to you. Here are some possible suggestions for the second part:

- Which algorithm did you use, and why?
- Are the growth rates (the $f(n)$) surprising? Can you come up with an explanation for them?
- How long does it take your algorithm to run? Does this make sense? Do you notice things like the cache size of your computer having an effect?
- Did you have any interesting experiences with the random number generator? Do you trust it?

Your grade will be based primarily on the correctness of your program and your discussion of the experiments. Other considerations will include the size of n your program can handle. Please do a careful job of solid writing in your writeup. Length will not earn you a higher grade, but clear descriptions of what you did, why you did it, and what you learned by doing it will go far.

Hints:

To handle large n, you may want to consider simplifying the graph. For example, for the graphs in this assignment, the minimum spanning tree is extremely unlikely to use any edge of weight greater than $k(n)$, for some function $k(n)$. We can estimate $k(n)$ using small values of n, and then try to throw away edges of weight larger than $k(n)$ as we increase the input size. Notice that throwing away too many edges may cause problems. Why will throwing away edges in this manner never lead to a situation where the program returns the wrong tree?

You may invent any other techniques you like, as long as they give the same results as a non-optimized program. Be sure to explain any techniques you use as part of your discussion and attempt to justify why they should give the same results as a non-optimized program!

CS 124 Programming Assignment 1 Report:

Algorithm Choice/(Theory/Explanation)
For the programming set, I decided on using Kruskal's algorithm for one reason—simplicity. When comparing the most efficient implementations of Kruskal's and Prim's their overall time complexities evaluate to be approximately the same. Thus, other considerations such as the simplicity of your code should come into play. In this case, Kruskal's algorithm becomes borderline trivial if the Union-Find data structure is implemented properly. The lecture notes also give comprehensive pseudo-code for the required Union-Find methods. With that being said, there are some downsides to choosing Kruskal's algorithm. The first is that I needed to modify the graph implementation outlined in class. During lecture, we were told that modeling a graph is usally done using an adjacency list or matrix, but I ended up using neither. Since Kruskal's algorithm really only cares about edge weights I created an edge list. I did so using a C++ vector where each element was a tuple. The first element in the tuple was the edge weight and the second was a tuple of the two nodes being added. Note, the assignment specified that the graph should be a complete graph. Therefore, to conserve memory I only added the edge once to this edge list. One downside of using a vector to implement this edge list is that this implementation is memory intensive. I had to stop at n = 32768 because any larger graphs would cause my computer to crash. Additionally, since the coordinates of these nodes only matter when calculating the edge weights, I created a list numToNode which essentially mapped the coordinates of these nodes to unique, sequential ID numbers. This made the implementation of Union-Find far easier. In Union-Find, I used a vector instead of a tree to represent these sets like the slides suggest. This is for a couple reasons, the first is that having pointers to each node is done automatically since each node's slot matches their node Number. Additionally, having the pointer behavior can be replicated by changing the value at the i-th slot. For instance, the i-th slot should contain a pointer to the parent of the node with node number i, this pointing behavior is easily represented by placing the parent's node Number into the i-th slot. One down side is that vectors occupy contiguous slots of memory, again adding to the memory inefficiencies of my program. Which poses the question: **How do I make my program more memory efficient?**
My first idea would sacrifice readability of the code but instead of having Kruskal's as a separate function, I could just copy the code directly into the main function. This could help with the memory issue because I pass in an edge list as input into Kruskal's. This is an issue because doing so create a copy of the edge list, effectively doubling the memory overhead for the sake of readability. Additionally, I could try to move away from a vector and instead use a static sized array. This could help because vectors over allocate slots because they expect us to append to the end of them. Whereas a normal array would stay the same size and would be more memory efficient. Finally, I could possibly modify the types within these tuples and vectors. I use doubles instead of floats. If that extra precision isn't needed then converting the doubles into floats could do wonders for the memory allocation.

Runtime Analysis:
The overall time complexity of my entire program is $O(n^2)$. This is not due to Kruskal's but because of the construction of the complete graph. Making the graph requires connecting n nodes with n other nodes, resulting in a time complexity of $O(n^2)$. However, ignoring the graph construction and focusing only on my implementation of Kruskal's, we get a much smaller asymptotic. First, I start by sorting my edge list using C++'s sorting algorithm. This takes $O(E\log(E))$ where E represents the number of edges. We can represent this in terms of nodes because we know that the graph is a complete one. We thus get $O(E\log(E)) == O(n^2\log(n))$. Next, I instantiate my Union-Find object, this takes $O(n)$ because I must create a vector for each node in the graph. I then iterate over the edge list which has $O(n^2)$ elements and for each iteration I run $FIND$ and $UNION$ so the whole for-loop takes $O(n^2 \cdot \log(n))$. I finally return my result which is the total weight of the MST. With this, we see that the sorting of the edge list

and the for-loop both dominate my algorithm's runtime resulting in my Kruskal's implementation taking $O(E \cdot \log(E)) == O(n^2 \cdot \log(n))$.

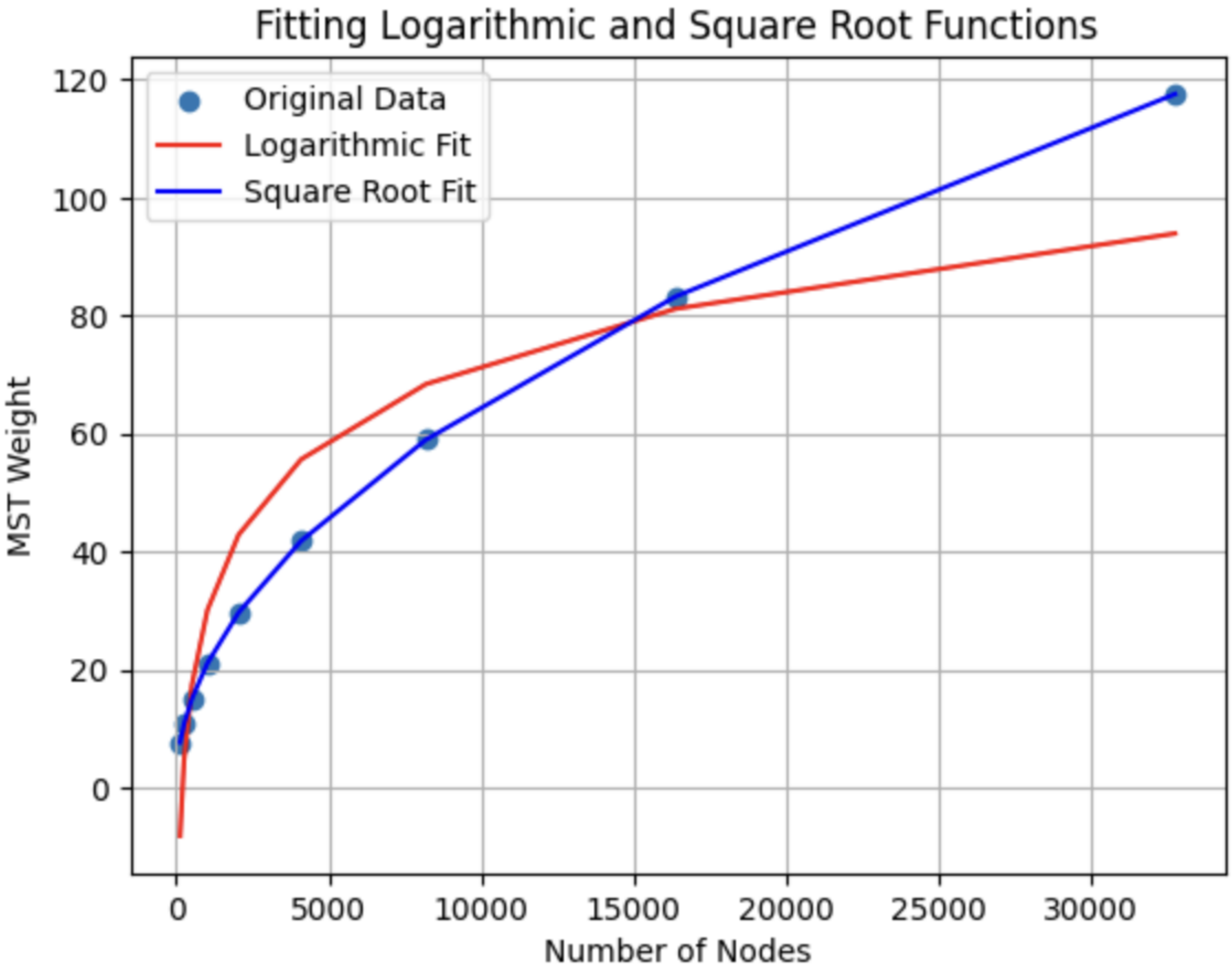
Graph Size vs MST Weight Table:

Average MST Weight Table:

Graph Size:	0-D	2-D	3-D	4-D
128	1.20774	7.66499	17.5731	28.3416
256	1.21137	10.6782	27.4032	46.9181
512	1.18518	15.072	43.2327	77.4946
1024	1.22149	21.0423	67.8713	130.554
2048	1.20363	29.7364	107.491	216.763
4096	1.19442	41.852	168.7	362.332
8192	1.20497	59.0318	267.476	603.46
16384	1.2065	83.1906	421.55	1011.44
32768	1.19672	117.5938	667.542	1684.32

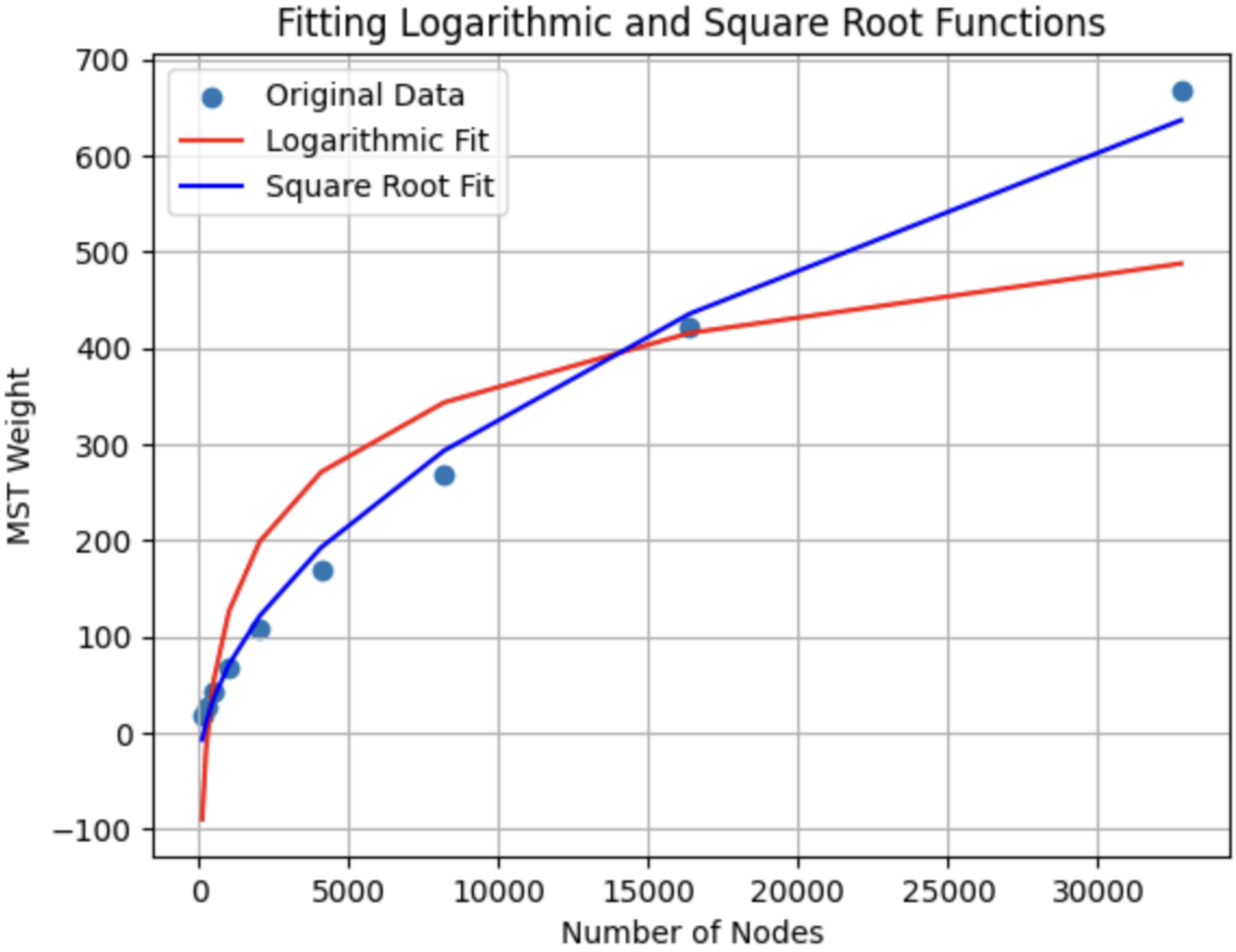
Growth Rate Analysis f(n):

I will start by pointing out that there is likely an error in my 0-D implementation. For the 0-D case it seems that the weight of the MST hovers at around a constant 1.2 value. Thus, $f(n) = 1.2 + \epsilon$ for the 0-D case, based on my data. The ϵ is meant to encapsulate any random error. See graph below for the 2-D approximation. **2-D Approximation Based on Fitting log and sqrt models:**

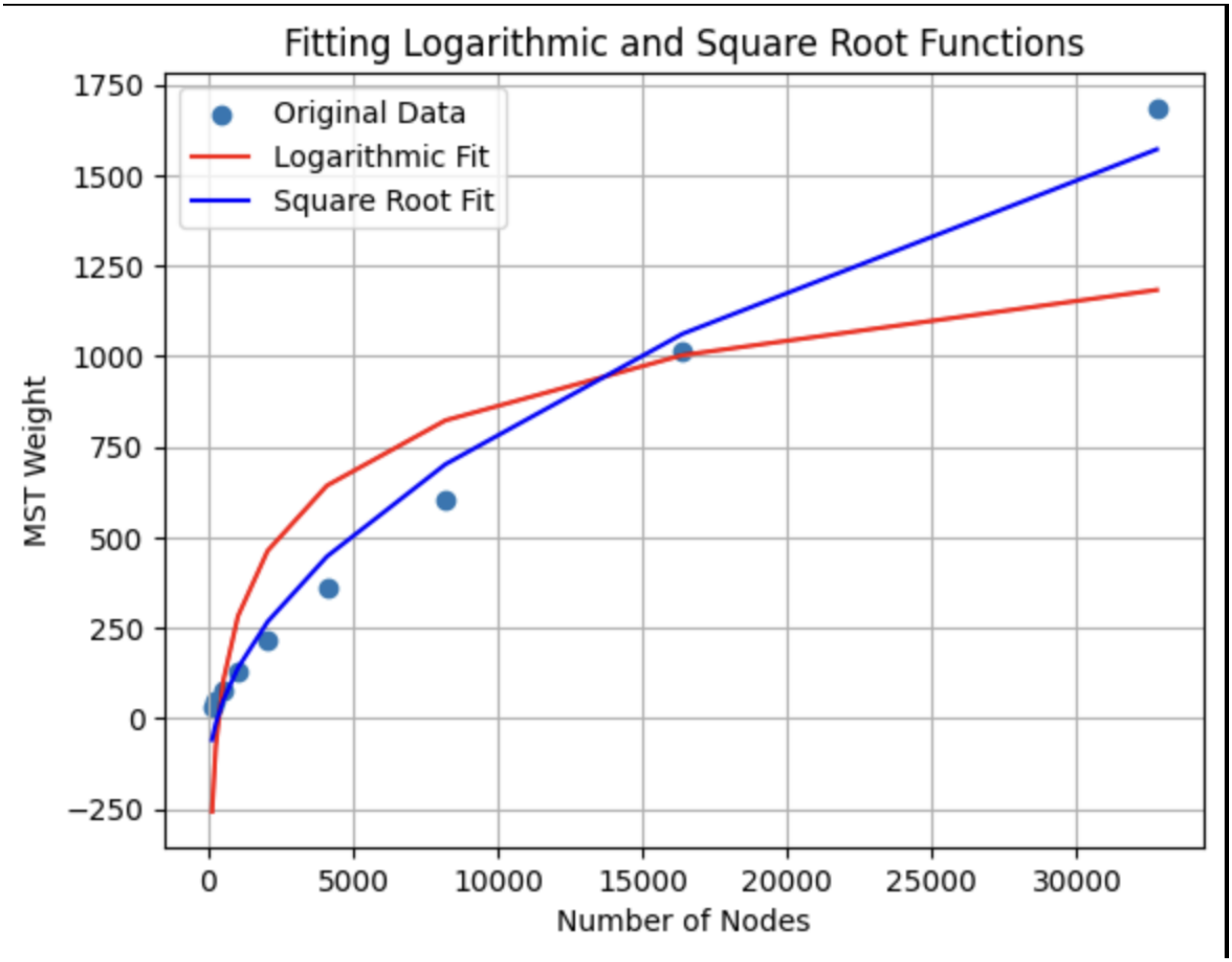


Based on this data, it seems that $f(n) = 0.8 \cdot \sqrt{0.6x} + 0.35$.

3-D Approximation Based on Fitting log and sqrt models:



Based on this data, it seems that $f(n) = 1.9 \cdot \sqrt{3.73x} - 50$.



Based on this data, it seems that $f(n) = 4.6 \cdot \sqrt{4.2x} - 168$.

Reaction to f(n):

These values surprised me because upon looking at the plotted data it seemed clear that the relationship was either log or a root of some kind. However, based on my knowledge that Kruskal's algorithm runs in $E\log(E)$ I was thinking a log model would fit the data best. Nevertheless, it seems that a square root model was a better fit to the data for all cases except for the 0-D case. This might have something to do with the distance formula being used in calculating the edge weights.

Time Analysis

Time for Kruskal's Table:

Graph Size:	**0-D**	**2-D**	**3-D**	**4-D**
128	0.001003091	0.0009351582	0.0008842428	0.0009361848
256	0.003555608	0.00328044	0.003283676	0.003324192
512	0.01163719	0.010171612	0.010194742	0.10035748
1024	0.04	0.0391	0.0392	0.03888
2048	0.169	0.17	0.171	0.1689
4096	0.738	0.7324	0.733	0.7343
8192	3.17	3.177	3.199	3.192
16384	13.879	13.766	13.749	13.675
32768	76.72	75.189	81.36	76.86142

Looking at my values, we will see that the time is relatively consistent with respect to the different dimensions of graphs. This makes sense because the addition of an additional coordinate or coordinates should only add a multiplicative factor to the runtime. A multiplicative factor that is likely negligible for large graphs.

Random Number Generator:

I do seem to trust the Random Number generator. This is because through out all my trials there did seem to be a bit of randomness to the edge weights and the total MST weight. The only things that makes me doubt the random number generator are my data for 0-D. I am still unsure why the overall weight of the MST did not increase and instead plateaued immediately.