CS 124 Programming Assignment 1: Spring 2022

Your name: Burak Ufuktepe

Quantitative Results

The average tree size for several values of *n* is given in Table 1. For each graph, we ran each value of *n* at least 5 times and we computed an average MST weight for each n.

Table 1 – Average MST Weight for Different Values of *n*

n	**Average MST Weight**			
	0D Graph	**2D Graph**	**3D Graph**	**4D Graph**
128	1.21	7.59	17.38	28.67
256	1.19	10.77	27.32	47.76
512	1.20	14.86	43.13	78.09
1024	1.20	21.00	68.13	130.04
2048	1.20	29.68	107.02	216.54
4096	1.20	41.80	169.60	361.61
8192	1.20	59.04	266.52	602.54
16384	1.20	83.19	422.40	1009.92
32768	1.20	117.49	668.26	1689.21
65536	1.20	166.00	1058.01	2827.50
131072	1.20	234.59	1677.05	4741.28
262144	1.20	331.83	2657.54	7951.42

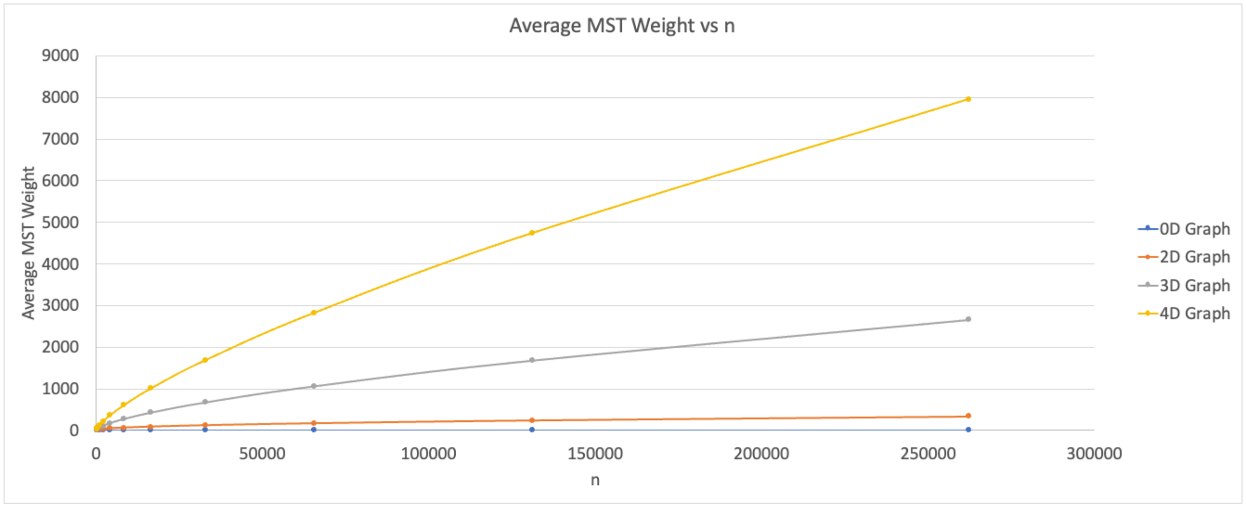

Figure 1 – Average MST Weight vs n

Estimated Functions

In general, we note that the estimated function is a constant function for 0-dimensional graphs, but for multidimensional graphs they are power functions. Growth rates are discussed more in depth in Growth Rate for the 0D Graph and Growth Rates for the 2D, 3D, and 4D Graphs sections.

0-Dimensional Graph

For the 0-dimensional graph, our estimate for the function is f(n) = 1.20.

Table 2 – Average MST Weight and f(n) for the 0-Dimensional Graph

n	**Average MST Weight**	**f(n)**
128	1.21	1.20
256	1.19	1.20
512	1.20	1.20
1024	1.20	1.20
2048	1.20	1.20
4096	1.20	1.20
8192	1.20	1.20
16384	1.20	1.20
32768	1.20	1.20
65536	1.20	1.20
131072	1.20	1.20
262144	1.20	1.20

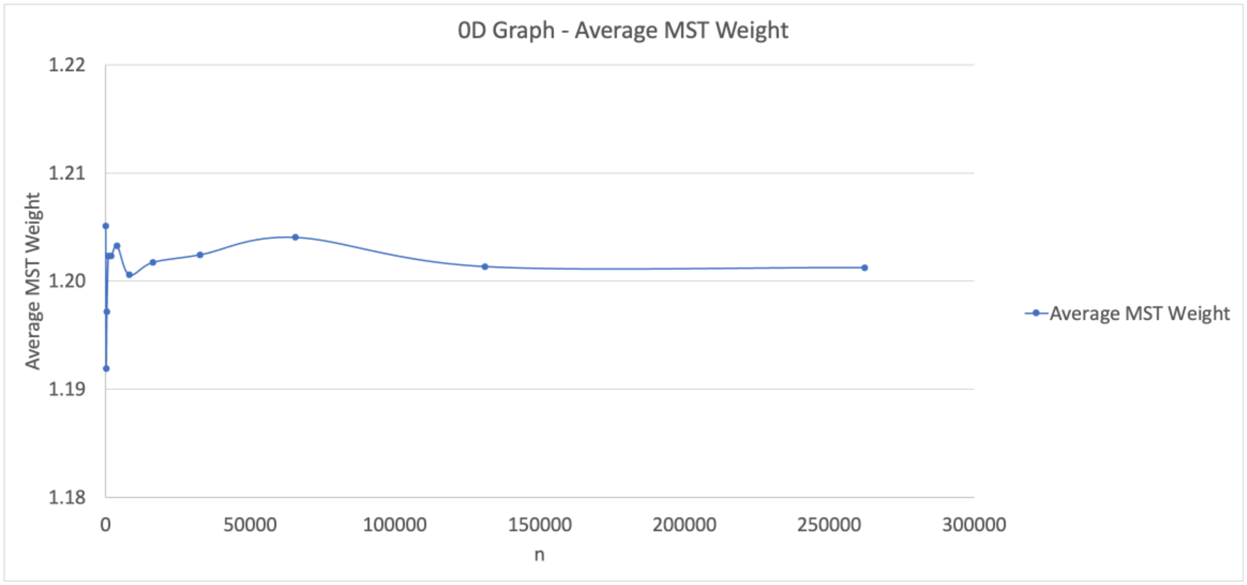

Figure 2 - Average MST Weight for the 0-Dimensional Graph

2-Dimensional Graph

For the 2-dimensional graph, our estimate for the function is *$f(n) = 0.65n^{0.5}$*.

Table 3 – Average MST Weight and f(n) for the 2-Dimensional Graph

n	Average MST Weight	f(n)
128	7.59	7.35
256	10.77	10.40
512	14.86	14.71
1024	21.00	20.80
2048	29.68	29.42
4096	41.80	41.60
8192	59.04	58.83
16384	83.19	83.20
32768	117.49	117.66
65536	166.00	166.40
131072	234.59	235.33
262144	331.83	332.80

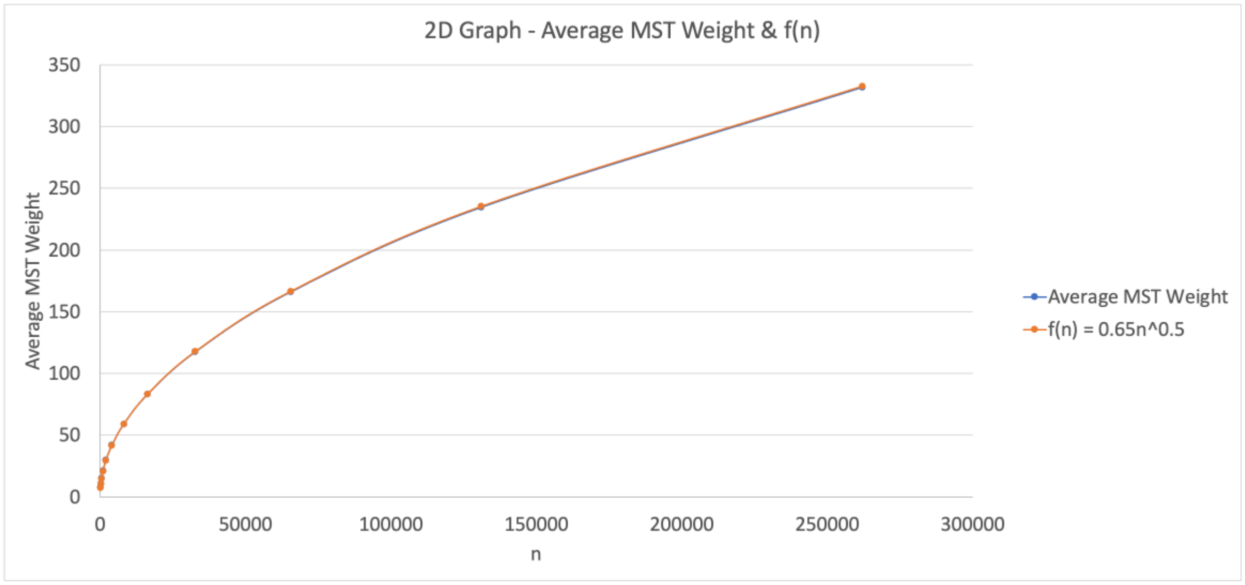

Figure 3 - Average MST Weight and f(n) for the 2-Dimensional Graph

3-Dimensional Graph

For the 3-dimensional graph, our estimate for the function is $f(n) = 0.62n^{0.67}$.

Table 4 – Average MST Weight and f(n) for the 3-Dimensional Graph

n	Average MST Weight	f(n)
128	17.38	16.00
256	27.32	25.46
512	43.13	40.51
1024	68.13	64.46
2048	107.02	102.56
4096	169.60	163.18
8192	266.52	259.63
16384	422.40	413.10
32768	668.26	657.27
65536	1058.01	1045.76
131072	1677.05	1663.89
262144	2657.54	2647.36

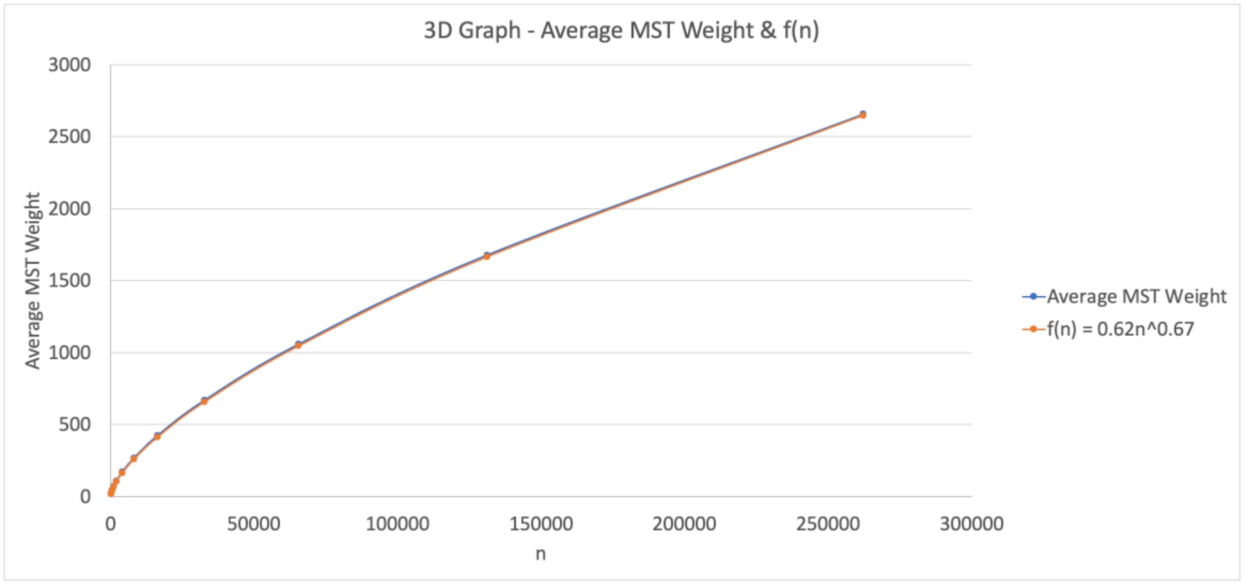

Figure 4 - Average MST Weight and f(n) for the 3-Dimensional Graph

4-Dimensional Graph

For the 4-dimensional graph, our estimate for the function is $f(n) = 0.69n^{0.75}$.

Table 5 – Average MST Weight and f(n) for the 4-Dimensional Graph

n	Average MST Weight	f(n)
128	28.67	26.26
256	47.76	44.16
512	78.09	74.27
1024	130.04	124.90
2048	216.54	210.06
4096	361.61	353.28
8192	602.54	594.14
16384	1009.92	999.23
32768	1689.21	1680.49
65536	2827.50	2826.24
131072	4741.28	4753.15
262144	7951.42	7993.81

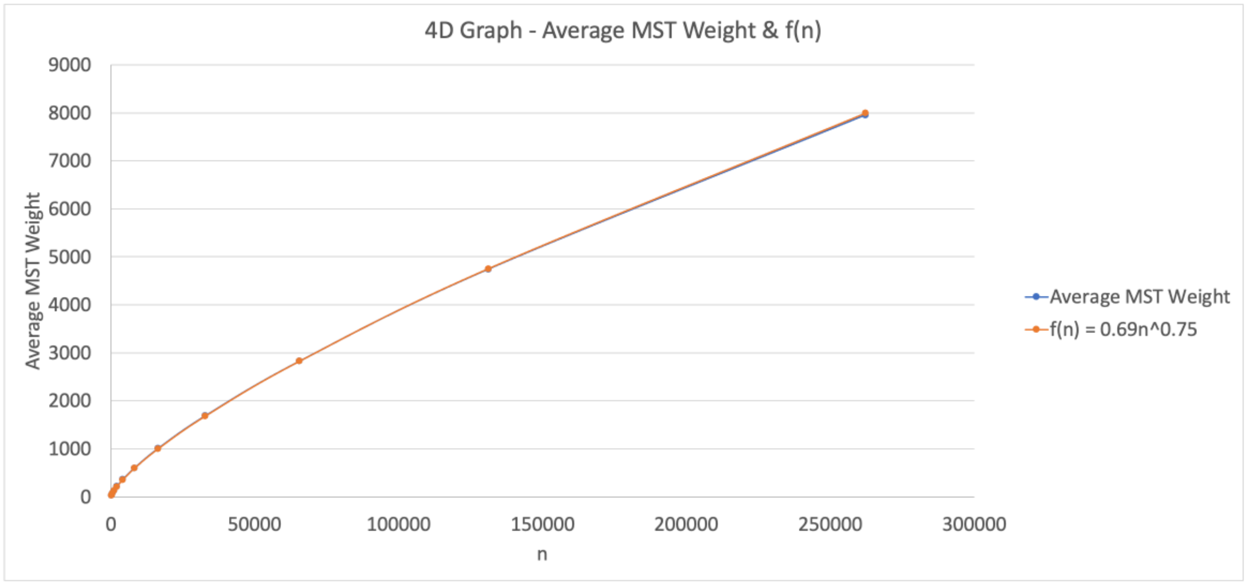

Figure 5 - Average MST Weight and f(n) for the 4-Dimensional Graph

Discussion

Due to its simplicity in terms of implementation, we used Kruskal's algorithm although we also implemented Prim's algorithm using a binary heap just to compare their runtimes.

We note that upon using reasoning of simply comparing Kruskal and Prim's algorithm, Prim's is known to run faster when we have a dense graph, and therefore should be the choice algorithm in theory – Prim's runs in O(m log(n)) time if it is implemented using a binary heap, whereas Kruskal's runs in O(m log(m)) time since we need to sort the edges. However, since both algorithms run in less than a minute even when n = 262144, we decided to use Kruskal's algorithm due to its simplicity.

The runtime of Kruskal's algorithm is bounded by the runtime of sorting the edges which can be done in O(m·log(m)) time where m is the number of edges. In order to reduce the runtime, we decided to simplify the graph by reducing the number of edges. Using small values of n, we estimated a function k(n) to determine a threshold for edge weights for each n. While building the graph, we threw away edges of weight larger than k(n).

In order to estimate a function k(n) we ran our program for n = 128, 256, 512, 1024, 2048, 4096, 8192, and 16384. We ran each value of n five times and determined the maximum edge weight. Based on the maximum edge weights we estimated a function k(n) for each graph so that k(n) gives a larger value than the corresponding edge weight. While building the graph we threw away edges that had weights greater than k(n). Throwing away edges in this manner never lead to a situation where the program returns the wrong tree. We can prove this claim as follows:

Proof. Let *G* = (*V, E*) be the complete graph, *E′* be the set of edges that are kept, and *E′′* be the set of edges that are thrown away. So, we have *E* = *E′* ∪ *E′′* where *E′* and *E′′* are disjoint sets. Note that the weight of any edge in *E′* is strictly less than the weight of any edge in *E′′*.

Furthermore, let *G′* = (*V, E′*) be the graph that we feed into Kruskal's algorithm. If our program generates a minimum spanning tree, that means the input graph was connected. This is because the while loop of Kruskal's algorithm terminates only if the number of edges in the tree reaches *n*-1. If the input graph is disconnected, then the while loop will never terminate and we will eventually get an index error when we try to get the next edge after exhausting all the edges of the input graph.

We claim that each edge in the *minimum spanning tree* of *G* must also be in *E′*. We prove this using **proof by contradiction**. Suppose that there is an MST *T* that has an edge *e* such that *e* ∈ *E′′*. Removing this edge from *T* separates *T* into two disjoint sets of vertices, *S* and *V* – *S*. Since we know G′ is connected, there must be an edge *e′* ∈ *E′* that connects the vertices *S* to *V* – *S*. Adding *e′* to the tree, we will end up with a new tree *T′*. Since *weight*(*e′*) < *weight*(*e*), the weight of *T′* must be less than *T*, contradicting the fact that *T* was a minimum spanning tree. Therefore, we conclude that each edge in the *minimum spanning tree* of G must also be in *E′*. ■

The maximum edge weight plots and the estimated k(n) for each graph are shown in the following sections.

k(n) for the 0D Graph

For the 0-dimensional graph, we used *k(n) = 20/n* as a threshold for the maximum edge weight.

Table 6 - Max Edge Weight and k(n) for the 0-Dimensional Graph

n	Max Edge Weight	k(n)
128	0.0596	0.1563
256	0.0308	0.0781
512	0.0157	0.0391
1024	0.0072	0.0195
2048	0.0042	0.0098
4096	0.0024	0.0049
8192	0.0014	0.0024
16384	0.0007	0.0012

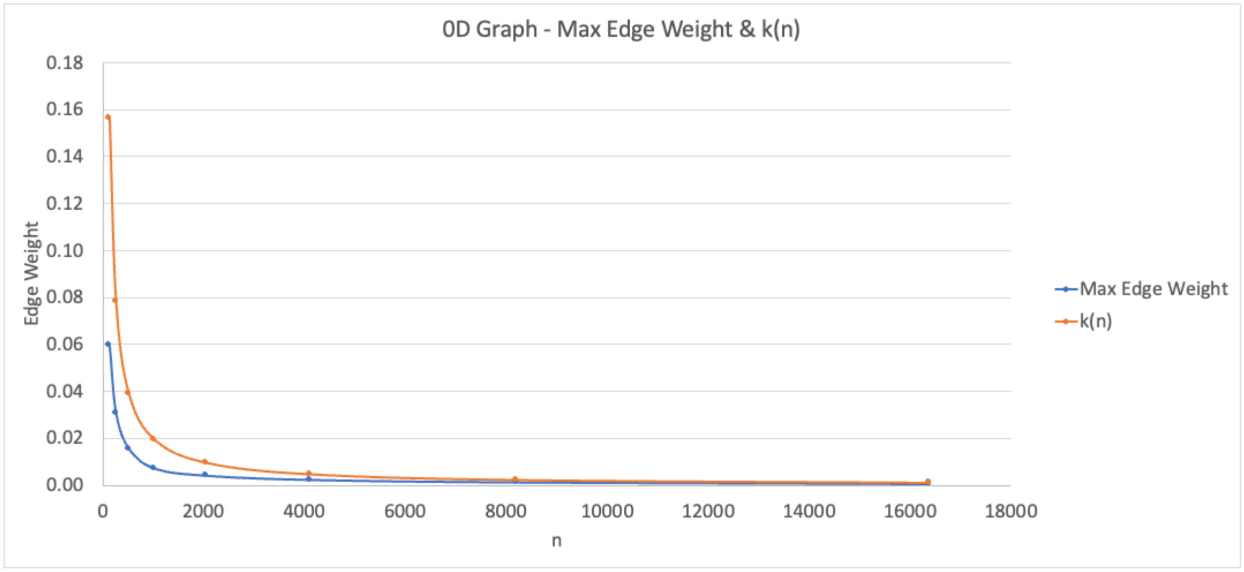

Figure 6 – Max Edge Weight and k(n) for the 0-Dimensional Graph

k(n) for the 2D Graph

For the 2-dimensional graph, we used *k(n) = 1.5/n*$^{0.4}$ as a threshold for the maximum edge weight.

Table 7 - Max Edge Weight and k(n) for the 2-Dimensional Graph

n	Max Edge Weight	k(n)
128	0.1666	0.2154
256	0.1320	0.1632
512	0.0800	0.1237
1024	0.0614	0.0938
2048	0.0405	0.0710
4096	0.0319	0.0538
8192	0.0242	0.0408
16384	0.0192	0.0309

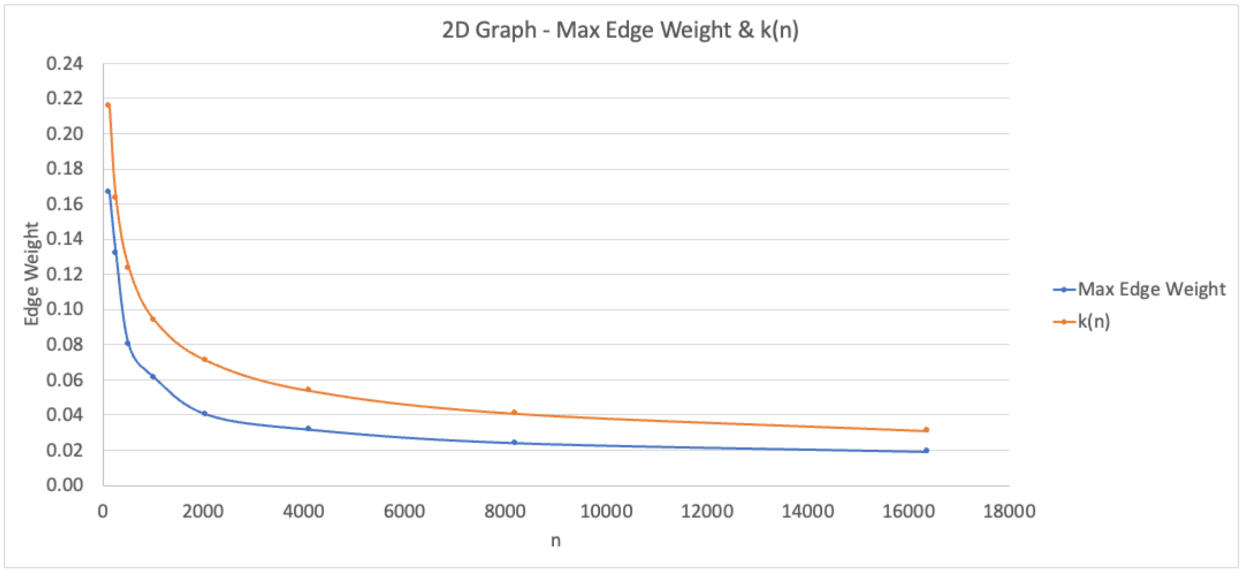

Figure 7 - Max Edge Weight and k(n) for the 2-Dimensional Graph

k(n) for the 3D Graph

For the 3-dimensional graph, we used $k(n) = 1.8/n^{0.3}$ as a threshold for the maximum edge weight.

Table 8 - Max Edge Weight and k(n) for the 3-Dimensional Graph

n	Max Edge Weight	k(n)
128	0.3246	0.4199
256	0.2851	0.3410
512	0.1729	0.2770
1024	0.1466	0.2250
2048	0.1277	0.1828
4096	0.0978	0.1484
8192	0.0790	0.1206
16384	0.0658	0.0979

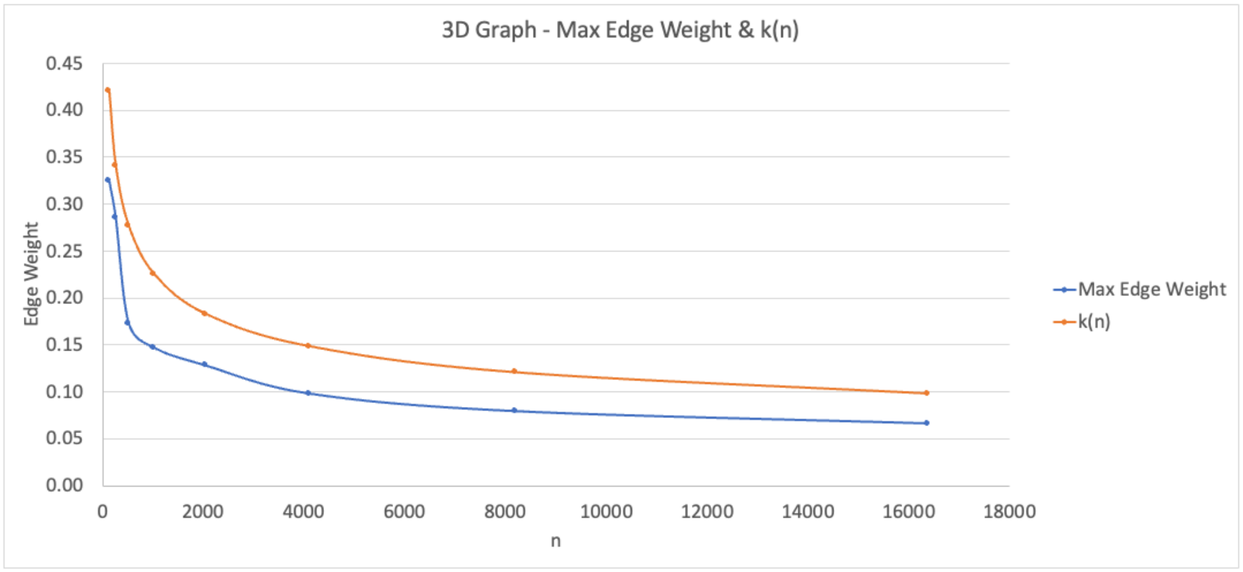

Figure 8 - Max Edge Weight and k(n) for the 3-Dimensional Graph

k(n) for the 4D Graph

For the 4-dimensional graph, we used *k(n) = 2/n*$^{0.25}$ as a threshold for the maximum edge weight.

Table 9 - Max Edge Weight and k(n) for the 4-Dimensional Graph

n	**Max Edge Weight**	**k(n)**
128	0.4917	0.5946
256	0.3417	0.5000
512	0.2934	0.4204
1024	0.2544	0.3536
2048	0.2330	0.2973
4096	0.1802	0.2500
8192	0.1640	0.2102
16384	0.1350	0.1768

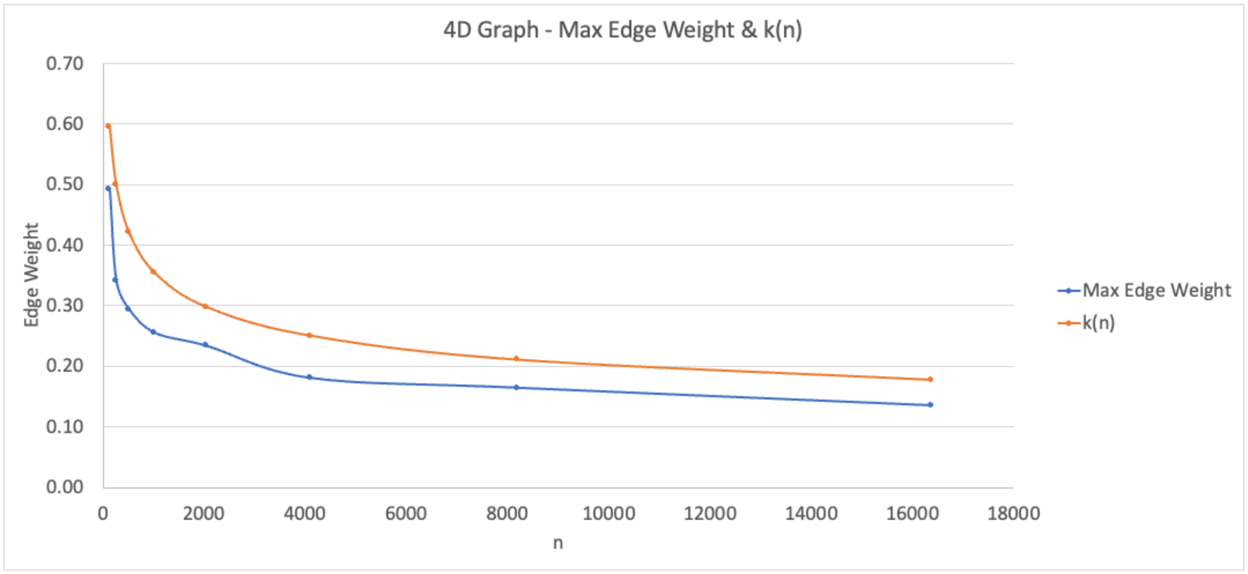

Figure 9 - Max Edge Weight and k(n) for the 4-Dimensional Graph

Growth Rate for the 0D Graph

The growth rate for the 0-dimensional graph was very interesting. At first, we were not able to come up with a function for f(n) because we observed oscillations for small values of *n* as shown in Figure 10.

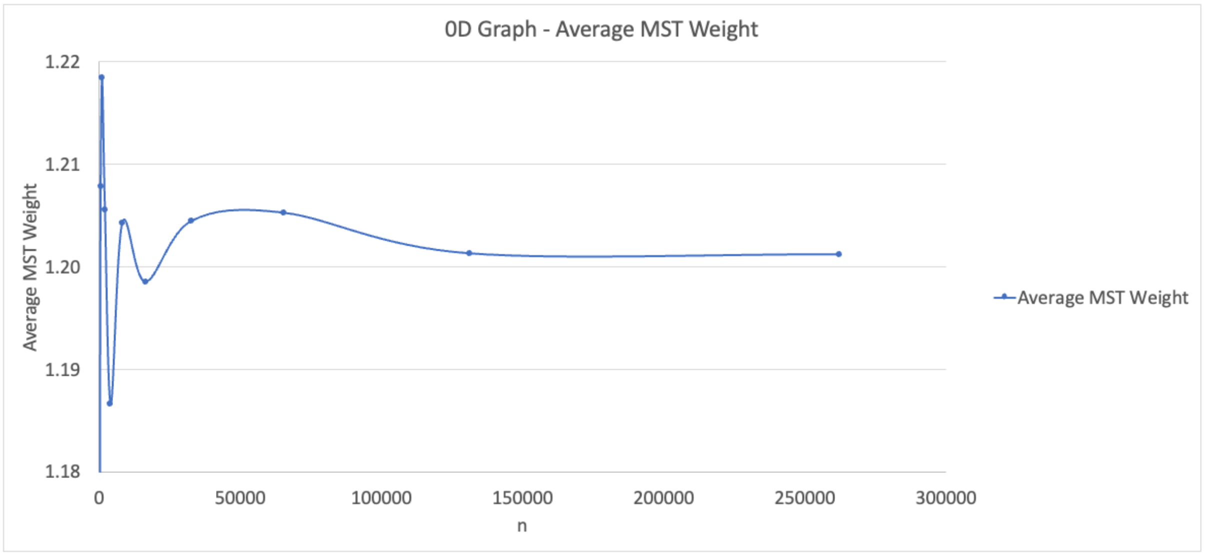

Figure 10 - Average MST Weight After 5 Iterations

We suspected that we needed to run more iterations for small values of n to get a better uniform distribution for random numbers. The number of iterations that we performed for each value of n is given in Table 10. As shown in Figure 2, after increasing the number of iterations, we observed that as n approaches infinity, the average MST weight converges to 1.20. This is equal to the Riemann zeta function at 3, which was very surprising. Apparently, this fact was shown by Alan M. Frieze in his paper “On the Value of a Random Minimum Spanning Tree” in 1983.

Table 10 – Number of Iterations Performed

n	Number of Iterations
128	120
256	120
512	120
1024	120
2048	120
4096	120
8192	40
16384	15
32768	12
65536	10
131072	5
262144	5

Growth Rates for the 2D, 3D, and 4D Graphs

We observed that as we increase the dimension of the graph and keep the same n value the average weight of the MST also increases. This makes sense because for the 2D graph the edge weights vary between 0 and 1.4, for the 3D graph they vary between 0 and 1.7, and for the 4D graph the edge weights vary between 0 and 2. In other words, if we use the same number of vertices but increase the space, the average weight of the MST also increases.

Furthermore, we see that the 4D graph has the largest slope whereas the 2D graph has the smallest slope. This is because each time we double *n*, the average weight of a single edge reduces by about 17% for the 4D graph and about 30% for the 2D graph. So, each time we double *n*, the average MST weight increases by a factor of (1 - 0.17) * 2 = 1.66 for the 4D graph, whereas for the 2D graph the average MST weight increases by a factor of (1 - 0.3) * 2 = 1.4. Hence, as we increase the dimension of the graph, the average MST weight grows faster.

We also observed that there is a correlation between the dimension of the graph and the expected weight of the MST. Below are our estimated functions for each graph:

2D Graph: $f(n) = 0.65n^{0.5}$

3D Graph: $f(n) = 0.62n^{0.67}$

4D Graph: $f(n) = 0.69n^{0.75}$

We can see that $f(n)$ is approximately $c \cdot n^{(d-1)/d}$ where c is a constant that is estimated from empirical evidence and d is the dimension of the graph. This was also determined by Steele, J. Michael (1988), "Growth rates of Euclidean minimum spanning trees with power weighted edges".

Runtime

After throwing away edges, we observed that running Kruskal's algorithm takes less than 10 seconds even for the 4D Graph when *n* = 262144. However, we noticed that building the graph dominates the runtime. The total runtime for each value of *n* is given in Table 11 (the values are rounded).

All runs were performed using the same laptop (10 Core M1 Pro Apple CPU, 32 GB RAM).

To speed up the process, we used Python's multiprocessing module utilizing 5 CPU cores.

Table 11 – Total Runtime for each Value of *n*

n	**Total Runtime in Seconds**			
	0D Graph	**2D Graph**	**3D Graph**	**4D Graph**
128	0	0	0	0
256	0	0	0	0
512	0	0	0	0
1024	0	0	0	0
2048	0	1	1	1
4096	1	3	4	4
8192	5	10	14	14
16384	19	41	57	58
32768	75	164	234	235
65536	303	671	911	950
131072	1187	2684	3715	3856
262144	4901	11222	23809	16378