Programming Assignment 1

Jennifer Guo and Ayana Yaegashi

February 23, 2022

1 Introduction

In this report we will discuss the results of our experiments as we examine how the average weight of a minimum spanning tree grows with respect to the number of vertices. We worked only with complete graphs of n vertices in 0, 2, 3, and 4 dimensions. We used C++ to implement Kruskal's algorithm to calculate the average weight of minimum spanning trees for these randomly generated complete graphs.

2 Implementation

2.1 Generating Random Complete Graphs

First, we had to generate random complete graphs on 0, 2, 3, and 4 dimensions. We decided to represent graphs as an array of edge objects that we defined ourselves. We defined an edge struct with 3 properties: an integer representing the outgoing vertex, an integer representing the incoming vertex, and a floating point number representing the weight of the edge.

Next, we wrote a function `createGraph` that would populate each graph by adding new edge objects to the array of edges as well as assign an edge weight depending on the dimension of the graph. We used C++'s built-in `rand()` function, normalized to return floating points between 0 and 1 inclusive, to generate weights. For the 0th dimension, we simply assigned each edge weight the value of our normalized random number generator function. For the 2nd, 3rd, and 4th dimension, we used our normalized random number generator function to generate dimensional (x, y, z, etc) values for the positions of vertices. Then, we used the distance formula to compute the weights of the edges. Additionally, to ensure we generated truly random graphs, we reseeded the `rand()` function at the start of every trial.

2.2 Choosing Kruskal's Algorithm & Implementing Union Find

Now that we had procedures to generate random graphs, we needed an algorithm to find the minimum spanning tree of a graph. We chose to implement Kruskal's algorithm over Prim's. Kruskal's algorithm iterates through the edges of a graph in sorted order of weight, greedily selecting the lightest edge and adding it to the MST so long as at least one vertex of the edge is not already in the MST. We were interested in implementing Kruskal's algorithm for a couple of reasons. First, we wanted to implement a Union Find data structure after learning about it in lecture. Asymptotically, Prim's algorithm is slightly faster if we would have implemented the priority queue as a sorted list since this would come out to $O(m) = O(n^2)$. On the other hand, Kruskal's asymptotic running time is $O(m \cdot log^*(n)) = O(n^2 \cdot log^*(n))$. In addition, Kruskal's algorithm requires sorting the list of edges by weight (as we did using MergeSort) which is an additional $O(m \log m) = O(n^2 \log(n))$. However, what stood out as interesting to us about Kruskal's algorithm was the ability to optimize and improve on this runtime by coming up with a threshold function and discarding edges with weights above a certain threshold. Doing so would allow us to sort $o(n^2)$ edges and improve on the expected runtime of our program.

We implemented the Union Find data structure as a tree where each element in the set is a "Node" with "value", "parent", and "rank" properties. Union Find has five methods: `MAKESET`, `FIND`, `LINK`, `UNION`, and finally `DESTROY`. Initially, each vertex in the graph is initialized to its own set, such that its "root" is itself. This is done in `MAKESET`. In the `FIND` method, we return the root of a Node. We can tell if two Nodes are in the same set if they share the same root. In addition, in order to get the $log^*(n)$ factor in our runtime, we implemented path compression. In other words, when we call `FIND(x)` on some Node x, we also set x's immediate parent to its root so that future `FIND(x)` calls will only require $O(1)$ time (so long as the root is not merged into another set). As discussed in lecture, `UNION(x,y)` calls `LINK(FIND(x),FIND(y))`. Lastly, we added the method `DESTROY` (which was not discussed in Lecture 7) in order to aid us in managing memory. Once we no longer need a node (at the end of the program) we call `DESTROY` on that node to free the memory.

2.3 Optimizing Algorithm Efficiency

In sum, we were able to generate random complete graphs as a function of the number of vertices and the dimension. We then ran Kruskal's algorithm on these random graphs for a certain number of trials to find the average minimum spanning tree weight. However, as we increased the number of vertices n, we ran into segmentation faults for $n > 800$. To mitigate the issue, we moved the storage of our Union Find sets to the heap (using malloc) instead of the stack. We also moved the random generated graphs (represented as an array of edge objects) onto the heap. This helped us reach values of n up to 1000, but we clearly could do better.

We realized that in graphs with a large number of vertices, Kruskal's algorithm does not need to look at the vast majority of these edges. This is because each graph of n vertices has $\binom{n}{2} \approx n^2$ edges. Meanwhile, a minimum spanning tree of a graph of n vertices has only $n-1$ edges to connect all vertices. Because Kruskal's algorithm looks at edges in sorted order, edges with a weight above a certain threshold will likely never appear in the minimum spanning tree. By graphing the maximum weight edge appearing in the minimum spanning tree of graphs with n nodes on varying dimensions, we found a threshold function $k(n)$ that would return the weight in which we can start throwing away edges for a certain graph.

We were able to find a different threshold function $k(n)$ for each dimension $d = 0, 2, 3, 4$. We found the following equations that modeled the average maximum weight edge added to each minimum spanning tree as a function of n.

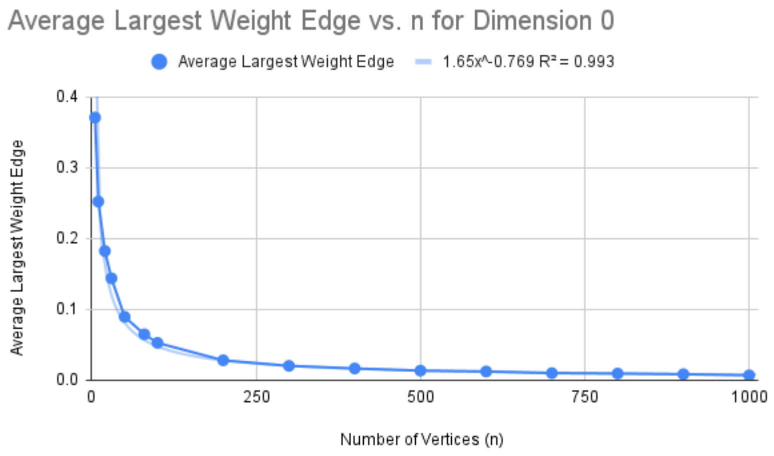


Figure 1: $k(n) = 1.65n^{-0.769}$ for Dimension 0

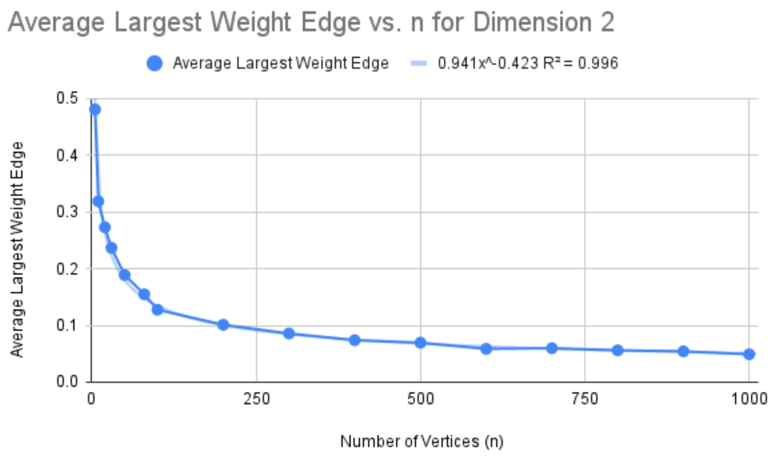


Figure 2: $k(n) = 0.941n^{-0.423}$ for Dimension 2

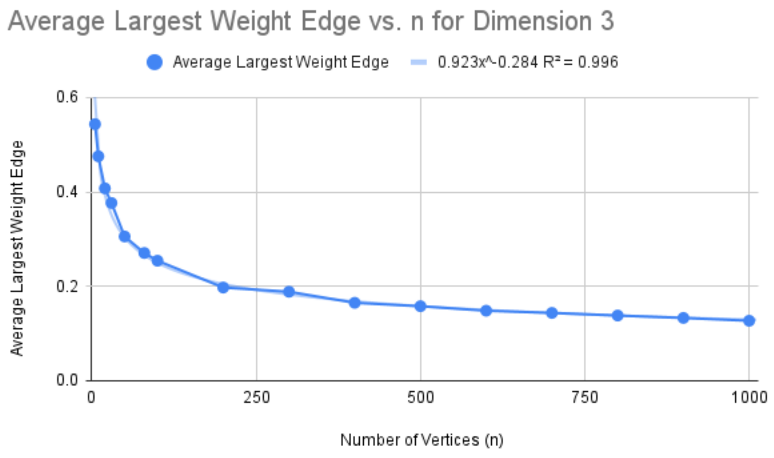


Figure 3: $k(n) = 0.923n^{-0.284}$ for Dimension 3

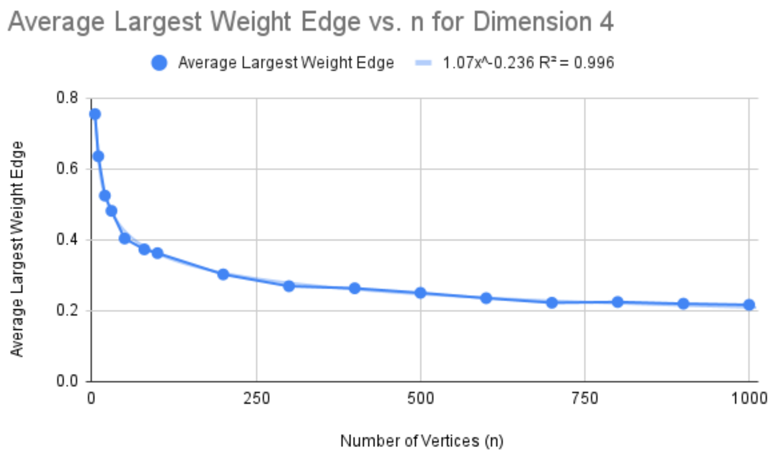


Figure 4: $k(n) = 1.07n^{-0.236}$ for Dimension 4

To test the validity of our optimization, after our program constructed a minimum spanning tree we added an $O(n)$ check to verify if the tree was missing any vertices. At first we noticed that our trees were consistently missing one or two vertices, even for low n values. Intuitively we understood why we were losing vertices: for dimensions 2 for example, if most vertices' coordinates are clustered in one area of the unit square and then one is a bit removed, then it's Euclidean distance to most other vertices will be relatively far and most if not all of its edges may be discarded by our optimization.

To address this, we first realized that for values of $n < 1000$, having a threshold function didn't have a big impact on the running time or our machines' ability to handle the required memory. Furthermore, for these low n values the distribution of edge weights was more sporadic making it more likely we would miss a vertex even with a higher threshold. As a result, we decided not to use a threshold function for $n < 1000$. For larger n, we addressed the issue by introducing a safety buffer to the threshold function and experimentally tested different values such that the vast majority of the time we were missing 0 vertices in the final minimum spanning tree and only began to losing vertices for $n > 50,000$.

Due to running time and space considerations, we were only able to output the maximum weight edge for graphs with $n < 1000$. As a result, we discovered that the above estimations for $k(n)$ work best for $n < 50,000$. Extrapolating $k(n)$ for values $n > 50,000$ we found that we were still keeping too many edges for $n = 65,536$ and $n = 131,072$. As a result, we experimentally found lower threshold values such that we were getting rid of enough edges for our program to run, while keeping enough edges to be able to construct a valid minimum spanning tree (or one losing very few vertices in comparison to n). This was the result of our experimentation:

Values of n	**Dimension**	**k(n)**
$50,000 < n \leq 70,000$ (i.e. $n = 65,536$)	0	0.0002
$50,000 < n \leq 70,000$ (i.e. $n = 65,536$)	2	0.008
$50,000 < n \leq 70,000$ (i.e. $n = 65,536$)	3	0.04
$50,000 < n \leq 70,000$ (i.e. $n = 65,536$)	4	0.08
$70,000 < n \leq 150,000$ (i.e. $n = 131,072$)	0	0.00007
$70,000 < n \leq 150,000$ (i.e. $n = 131,072$)	2	0.005
$70,000 < n \leq 150,000$ (i.e. $n = 131,072$)	3	0.027
$70,000 < n \leq 150,000$ (i.e. $n = 131,072$)	4	0.07

We thus implemented our `findThreshold` function as a piece-wise function. Then, while generating random graphs we wouldn't add edges with randomly generated weights above the threshold we found for a specific n and d to the graph. The result was that our array of edges was much smaller than $O(n^2)$ and we were able to reduce the significant sorting overhead of Kruskal's so that our program was much faster. With accurate thresholds, we were now able to generate graphs with n up to 131,072.

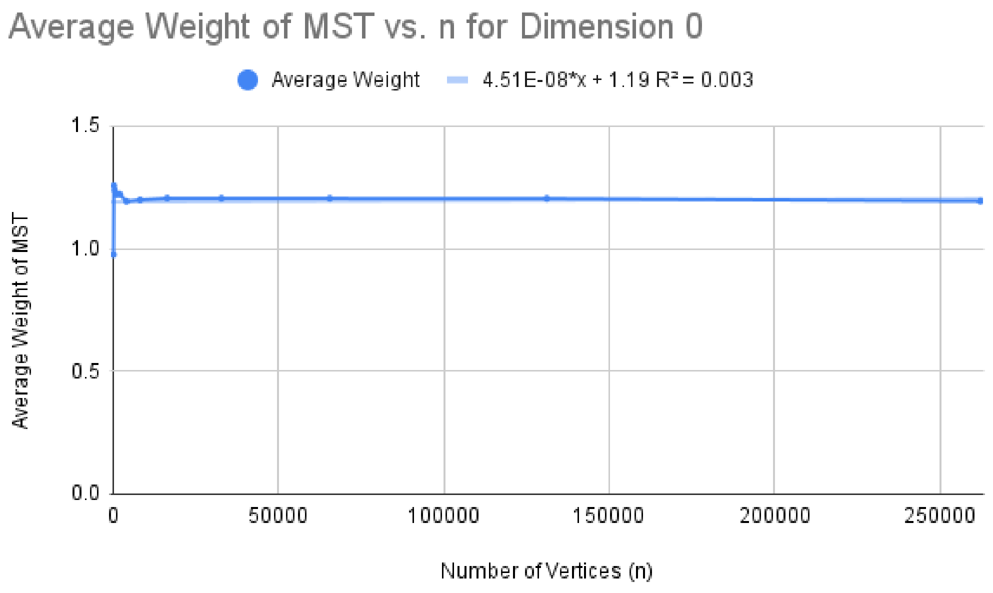
3 Results

For each dimension and each value of n, we ran 5 trials and took the average size of the MST our program generated.

3.1 Dimension 0

Number of Vertices (n)	Number of Trials	Average MST Size
128	5	0.974841
256	5	1.25651
512	5	1.23691
1024	5	1.21987
2048	5	1.22093
4096	5	1.19217
8192	5	1.19808
16384	5	1.20465
32768	5	1.20394
65536	5	1.20378
131072	5	1.20127
262144	5	1.19304

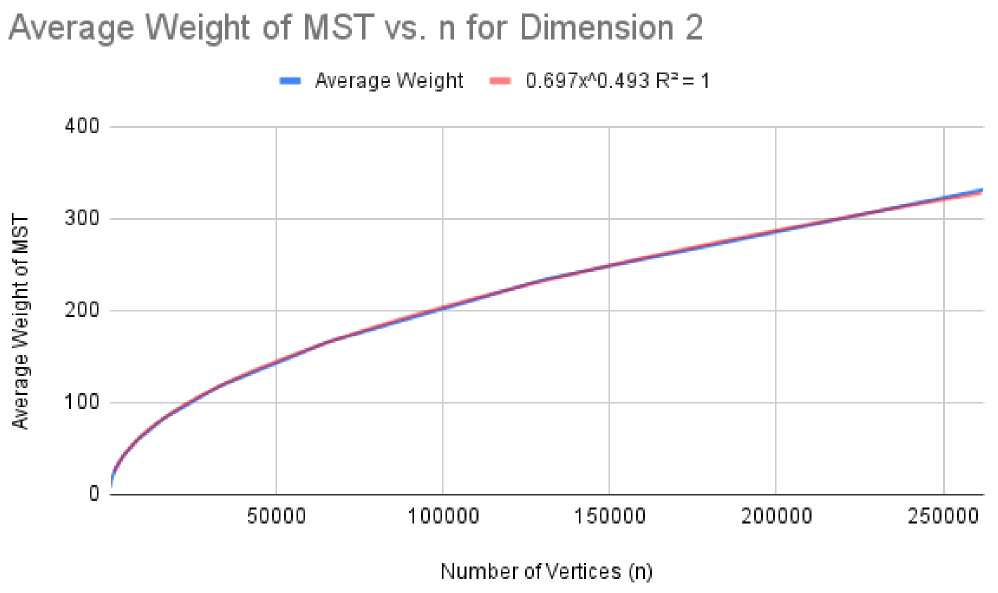
We graphed n against the average weight of the corresponding minimum spanning tree, and found the following trend. We guess that $f(n) = 1.2$. While this is a bit different from the line of best fit generated from our data that can be seen in the graph below, we believe that the coefficient applied to x is primarily the result of noisy data from small values of n and for large n the trend is that of a constant function.



3.2 Dimension 2

Number of Vertices (n)	Number of Trials	Average MST Size
128	5	7.72424
256	5	10.8236
512	5	15.1549
1024	5	21.1074
2048	5	29.8641
4096	5	41.5819
8192	5	58.8632
16384	5	83.2242
32768	5	117.487
65536	5	165.96
131072	5	234.464
262144	2	331.642

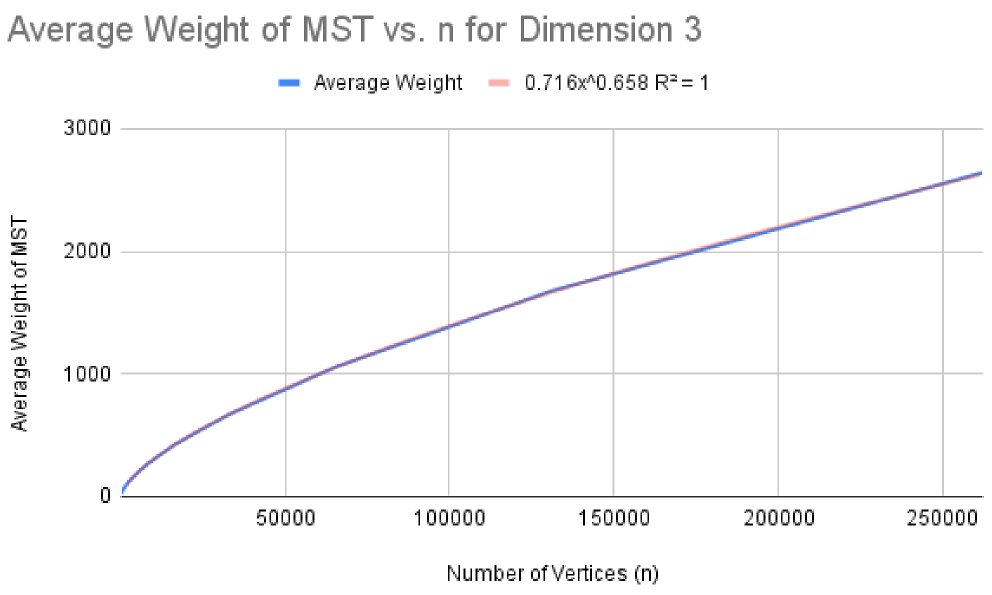
After graphing the average MST sizes against n, we saw the following trend. For dimension 2, our guess is $f(n) = 0.697x^{0.493}$.



3.3 Dimension 3

Number of Vertices (n)	Number of Trials	Average MST Size
128	5	17.766
256	5	27.4806
512	5	42.8833
1024	5	68.5715
2048	5	107.134
4096	5	169.364
8192	5	266.737
16384	5	423.254
32768	5	667.554
65536	5	1058.828
131072	5	1676.858
262144	1	2641.2

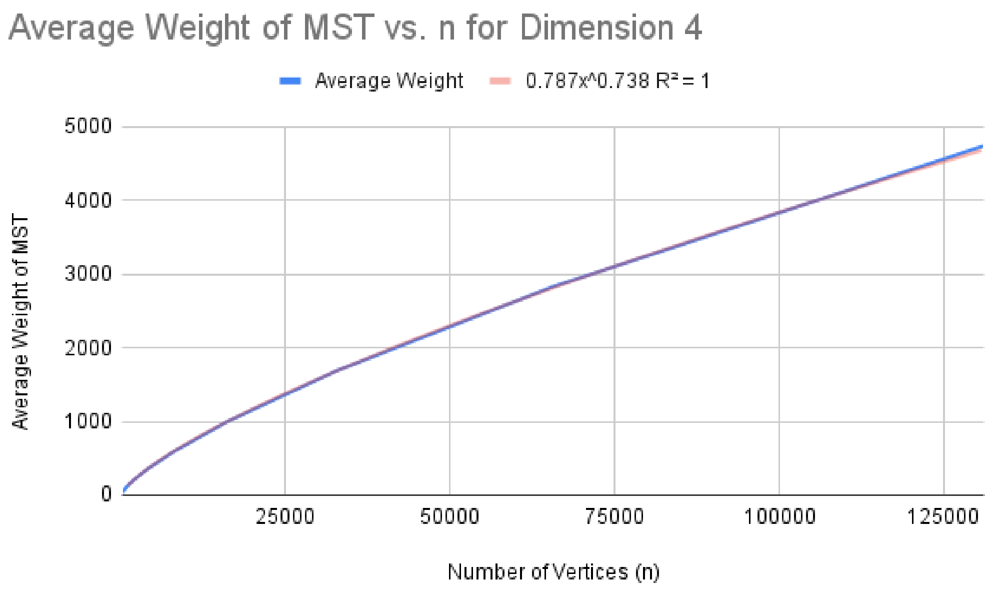
We graphed the average MST sizes against n, and guessed for dimension 3 that $f(n) = 0.716n^{0.658}$.



3.4 Dimension 4

Number of Vertices (n)	Number of Trials	Average MST Size
128	5	29.0745
256	5	46.3492
512	5	77.9911
1024	5	130.997
2048	5	215.953
4096	5	360.032
8192	5	601.375
16384	5	1009.23
32768	5	1687.6
65536	5	2826.01
131072	5	4739.0516

We graphed the average MST sizes against n, and our guess for dimension 4 is $f(n) = 0.787n^{0.738}$.



3.5 Impact of Threshold Function

In order to assess the impact of the threshold function on our results, we also kept track of the number of vertices missing from the minimum spanning trees our program produced. The table below summarizes these results.

n	Avg. Missing Vertices (Dimension 0)	Avg. Missing Vertices (Dimension 2)	Avg. Missing Vertices (Dimension 3)	Avg. Missing Vertices (Dimension 4)
128	0	0	0	0
256	0	0	0	0
512	0	0	0	0
1024	0	0	0	0
2048	0	0	1.8	6
4096	0	0.6	1.8	6
8192	0	0.4	1.6	8.4
16384	0	0.4	1	9.2
32768	0	0.6	1.2	9.8
65536	0	0.2	0.8	16.8
131072	15.2	3.8	17.2	11.6
262144	400	0	658	n/a

3.6 Experimental Runtime for Dimension 0

We measured the runtime of the two portions of our program: the time it took to run Kruskal's algorithm to generate the MST and the time it took to construct the graph data structure.

n	**Kruskal's Time (ms)**	**Construct Graph Time (ms)**	**% of Total Time Spent Constructing Graph**
128	1.71	1.61	15.83%
256	6.75	3.40	9.14%
512	31.09	51.03	24.71%
1024	135.45	38.48	5.38%
2048	2.37	77.28	86.72%
4096	6.45	254.25	88.75%
8192	14.34	862.97	92.33%
16384	34.48	3323.08	95.07%
32768	86.31	12904.95	96.76%
65536	129.578	50154.92	98.72%

4 Discussion

4.1 Interpreting f(n)

In our results, we found that for dimensions 2, 3, and 4, our guess for $f(n)$ was some power function, as summarized in the table below.

Dimension	**f(n)**
0	f(n) = 1.2
2	$f(n) = 0.701n^{0.492}$
3	$f(n) = 0.718n^{0.657}$
4	$f(n) = 0.787n^{0.738}$

At first we were surprised by our results because they implied that for very large n, $f(n)$ eventually converges for Dimension 0. With Dimension 0, the average weight of MST tends to fluctuate with small numbers of vertices, but then converges to around 1.2. We believe it makes sense for the weight of the MST to eventually converge because as the number of vertices n get bigger, the number of edges in the complete graph increases by n^2. However, the number of edges that can be added to the MST remain at $n-1$. So, with large n, there are exponentially more edges distributed across $[0,1]$ being generated, but a fewer proportion of these edges can be added to the MST. Therefore, as n increases, the average weight of an edge added to the MST decreases. This is seen in Figure 1 in section 2.3, where the curve approaches 0 for large n. The average maximum weight edge added to an MST decreases as n gets bigger. Thus, we postulate that weight of the MST converges because the increasing number of edges $n-1$ added to the MST is counterbalanced by the decreasing average weight of these edges.

As for higher dimensions (dimensions 2, 3, 4), we don't believe the average weight of the MST converges as the number of vertices gets bigger. The main reason we believed the average weight of the MST converged for dimension 0 is because as the number of edges added to the MST increases, this is counterbalanced by the decreasing average weight of these edges. However, as can be seen by Figures 2, 3, and 4 in section 2.3, the decreasing average weight of the edges gets higher and higher as the dimension increases. This can be thought of as less and less counterweight to the increasing number of edges added to the MST as n approaches infinity. Because there is less of a counterweight as the dimensions increase, the average weight of the MST also converges to infinity at a faster rate for higher dimensions as well, explaining why as dimension increase, the exponents of n increase as well.

An additional note is that we were not able to run 5 trials on $n = 262144$ for dimensions 2, 3, and 4. At first, we were not able to run $n = 262144$ for any dimension when we used MergeSort or QuickSort as our primary overhead for Kruskal's because we kept running into segmentation fault errors. We theorized that this must be because MergeSort and Quicsksort respectively have $O(n)$ and (expected) $O(\log n)$ space complexities, so we switched our sorting mechanism to InsertionSort for a constant space complexity, with the tradeoff of a much larger runtime. This resolved our segmentation fault errors and we were able to run trials on $n = 262144$ for all dimensions. However, the trials took a very long time. For example, for $n = 262144$ on the 4th dimension, 1 trial took upwards of 12 hours. Therefore, we were only able to include a few trials for $n = 262144$ for dimensions 2, 3, and 4 due to time constraints.

4.2 Impact of Threshold Function

As summarized in section 3.5, the number of vertices missing from the minimum spanning graphs produced by our program is minimal, averaging below 1 for dimensions 0 and 2 up to $n = 65,536$. In general, the number of missing vertices increases with n and dimension. This is because with higher values of n and higher dimensions (in particular dimensions 3 and 4), the graphs became larger and individual edges became more complex which required more memory usage. As a result, we needed to implement tighter thresholds in order for our program to run. Note however that as n increases, the largest expected weight of an edge also decreases so the incremental difference of having one more vertex in the minimum spanning tree is unlikely to have a large impact on the overall minimum spanning tree's size. Furthermore, we found that the number of vertices missing from each graph was less than 0.3% of n. Thus, we felt confident that even in cases where our minimum spanning trees were missing a couple of vertices for graphs with large n and dimensions, our program was able to produce accurate and meaningful estimates of the average minimum spanning tree's size. Note that with $n = 262144$, our threshold values were less consistent than for $n = 131072$ and below since each trial took so long to run, it was more difficult to optimize the threshold values.

4.3 Runtime Analysis

Additionally, we were able to collect some data on the experimental runtime of our program. We noticed quickly that the bulk of our runtime was in creating the graph data structures, while running

Kruskal's Algorithm to find the MST all took less than 400 milliseconds for each trial. For example, our highest runtime to find the MST was for a graph of 131072 nodes in the 4th dimension, and finding the MST took a blazingly fast 354ms. Table 3.5 clearly illustrates how the bulk of the total runtime of our program is dominated by the time spent constructing the graph. This is less so for smaller values of n, but definitely a consideration for large values of n ($n \geq 2048$).

As for a more detailed runtime analysis of our implementation of Kruskal's algorithm, it's difficult for us to say much more. Because we implemented a $k(n)$ threshold and threw out edges above the threshold while creating each graph, it became the case that the number of edges Kruskal's had to go through were no longer proportional to n as n increases. This explains a few discrepancies in Table 3.5, such as how the runtime of Kruskal's takes 135.45 ms for graphs with 1024 vertices on the 0th dimension, but 2.37 ms for graphs with 2048 vertices on the 0th dimension. One would expect the overall runtime of Kruskal's to increase as the number of nodes n increases, and this is generally the case. However, there are a few discrepancies in our data which we believe are explained by the fact that we removed edges above a certain threshold, so the case with 1024 vertices may actually be dealing with more sorting more edges (the overhead of Kruskal's run time) than the case with 2048 vertices, depending on our $k(n)$ function. A more accurate analysis of runtime would involve us recording precisely how many edges were added to the graph after removing those above the threshold, and comparing that number to the actual runtime of Kruskal's. If we had more time to do this experiment again, we would be careful to take this into consideration.