CS 124 Programming Assignment 1: Spring 2024

Noah Chung[1] and Anaïs Killian[1]

[1]Harvard University, Class of 2026.

1 Abstract

In this experimental assignment, our goal was to determine, in each of the types of graphs handled, how the expected weight of the Minimum Spanning Tree (MST) grows as a function of n vertices. We had four main different types of graphs (zero, two, three, and four dimensions), each of which could take in an inputted n vertices.

1.1 Programming Language Used

We decided to use C++ to implement our code. This was due to two primary factors: 1) our most proficient programming language, Python, is much too slow for the graphs needed in this programming assignment, and 2) C++ has nice extended functionality to C that we believe makes it a little bit easier to code in.

1.2 Minimum Spanning Tree Algorithm Used

We ultimately decided to use Prim's algorithm over Kruskal's algorithm. The primary reasons for choosing do to this were as follows:

- **Efficiency in Dense Graphs:** Prim's algorithm tends to perform better in dense graphs, where the number of edges is close to the maximum possible n choose 2 edges. On the other hand, Kruskal's sorts through all of the edges in the list then goes through them again to check if the edge is part of the minimum spanning tree (MST) or not. Thus Kruskal's can take more time with more edges.
- **Ease of Implementation with Priority Queue**: Prim's algorithm may be simpler to implement since we use a priority queue, which we implemented using a binary heap. The key operation in Prim's algorithm involves selecting the minimum-weight edge connected to the growing minimum spanning tree, whereas Kruskal's algorithm involves sorting edges by weight, which can be slightly more complex to implement.
- **Space Complexity**: Prim's algorithm typically has a lower space complexity compared to Kruskal's algorithm, since Kruskal's algorithm requires storing the entire edge set, which can be larger than the adjacency matrix representation of the graph in dense graphs.

Overall, while both Prim's and Kruskal's algorithms are effective for finding minimum spanning trees, we thought Prim's would better fit our requirements in this Progset.

We did realize afterwards, however, that Kruskal's may have been a better algorithm because of our pruning mechanism. If our pruning algorithm was efficient enough, then our graph would not have been that dense, which means that Kruskal's algorithm could have been more efficient as the runtime is mostly dominated by the sorting aspect of the edges, which is, of course, based on the number of edges.

2 Quantitative Summary Summary

2.1 Average Tree Size

We assessed our average tree output throughout each each value of n stated in the abstract, and we plotted these values of n with the associated average minimum spanning tree and dimension used in the chart below.

Table 1: Average MST Results

Nodes	Trials	Dimension 0	Dimension 2	Dimension 3	Dimension 4
128	5	1.13519	7.60024	17.7462	28.1105
256	5	1.16087	10.7727	27.3951	47.7977
512	5	1.18509	14.9987	42.955	78.0151
1024	5	1.20066	21.2557	67.9947	129.552
2048	5	1.19177	29.4788	107.369	216.469
4096	5	1.20139	41.8249	169.889	361.619
8192	5	1.19739	59.0091	267.399	602.536
16384	5	1.19884	83.176	421.59	1008.97
32768	5	1.19906	117.674	668.613	1688.41
65536	5	1.20674	166.123	1058.41	2829.58
131072	5	1.20432	234.747	1676.97	4738.14
262144	5	1.19898	331.53	2656.91	7952.43

Below, we show the graphs representing the average tree size as a function of n, per dimension:

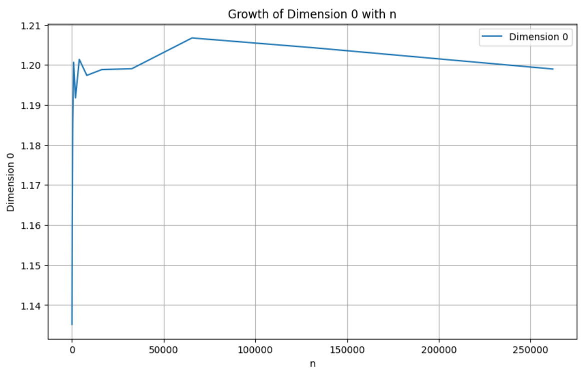


Fig. 1: Plotted graph for Dimension 0

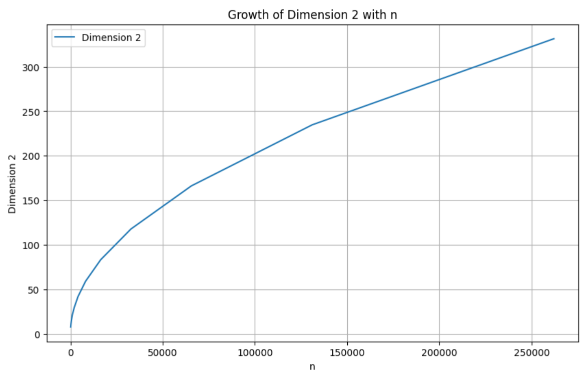


Fig. 2: Plotted graph for Dimension 2

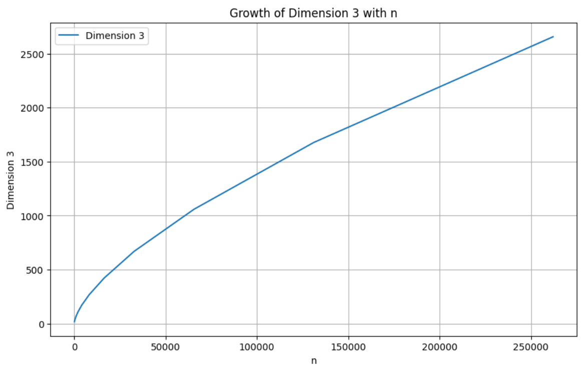


Fig. 3: Plotted graph for Dimension 3

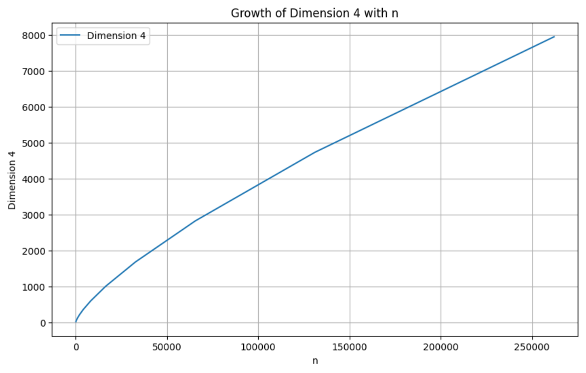


Fig. 4: Plotted graph for Dimension 4

Some interesting observations we had is that dimensions 2, 3, and 4 all grow relatively similar, whereas dimension 0 seems to have more of a constant, where the average MST jumps significantly when going from small values of n to larger values of n, and then becomes constant.

2.2 Guess for the Function $f(n)$

Our calculated estimates for $f(n)$ were calculated by experimentally determining the best fit for each of the graphs. We created a python script that would determine the best function fit based on the function $y = ax^b$. The code is provided below:

```python
import numpy as np
from scipy.optimize import curve_fit

# We gave the data
nodes = np.array([128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072,
    262144])
dimension_0 = np.array([1.13519, 1.16087, 1.18509, 1.20066, 1.19177, 1.20139, 1.19739,
    1.19884, 1.19906, 1.20674, 1.20432, 1.19898])
dimension_2 = np.array([7.60024, 10.7727, 14.9987, 21.2557, 29.4788, 41.8249, 59.0091,
    83.176, 117.674, 166.123, 234.747, 331.53])
dimension_3 = np.array([17.7462, 27.3951, 42.955, 67.9947, 107.369, 169.889, 267.399,
    421.59, 668.613, 1058.41, 1676.97, 2656.91])
dimension_4 = np.array([28.1105, 47.7977, 78.0151, 129.552, 216.469, 361.619, 602.536,
    1008.97, 1688.41, 2829.58, 4738.14, 7952.43])

# Define the function to fit (y = ax^b)
def func(x, a, b):
    return a * np.power(x, b)

# Fit the function for each dimension
p0 = (1, 1)  # Initial guess for parameters
params_0, _ = curve_fit(func, nodes, dimension_0, p0=p0)
params_2, _ = curve_fit(func, nodes, dimension_2, p0=p0)
params_3, _ = curve_fit(func, nodes, dimension_3, p0=p0)
params_4, _ = curve_fit(func, nodes, dimension_4, p0=p0)

print("Dimension 0: y = {:.6f}x^({:.6f})".format(params_0[0], params_0[1]))
print("Dimension 2: y = {:.6f}x^({:.6f})".format(params_2[0], params_2[1]))
print("Dimension 3: y = {:.6f}x^({:.6f})".format(params_3[0], params_3[1]))
print("Dimension 4: y = {:.6f}x^({:.6f})".format(params_4[0], params_4[1]))
```

As a result, we got this estimation for each dimension:

Dimension	Equation
0	$f(n) = 1.136817n^{0.005270}$
2	$f(n) = 0.6494399n^{0.497924}$
3	$f(n) = 0.650883n^{0.663237}$
4	$f(n) = 0.690598n^{0.745086}$

These functions aim to model the average weight of the MST as a function of n, and because of the drastic difference between each of the average MST weights modeled in the graphs due to their *dimensions*, we decided to find the equation of best fit also as a function of the dimension.

Similarly to our observation above, we noticed that the zero dimension graph had a strikingly different equation than the other functions. For the zero dimensional graph, n is only raised to the 0.00527.

We believe this is likely the case because the two, three, and four dimension graphs have distances between their nodes based on where the nodes are placed, and thus the distance function is needed to determine the weight of the edges. On the other hand, in the zero dimension case, the distance between the vertices are completely arbitrary and due to the random number that is generated, and thus there is no distance calculation needed.

2.3 Detailed Observations

We noticed that the exponent in the equation $y = ax^b$ roughly is of the form $\frac{(d-1)}{d}$, where d represents the number of dimensions. Furthermore, we noted that in the zero dimensional case, the graph is roughly constant at $x = 1.20145$. Therefore, we rewrote our estimated equations as follows:

Dimension	Equation
0	$f(n) = 1.20145$
2	$f(n) = 0.6494399n^{1/2}$
3	$f(n) = 0.650883n^{2/3}$
4	$f(n) = 0.690598n^{3/4}$

We also replotted the points, and the dashed line in the plots below represents the function of best fit that we found for each dimension as a function of n:

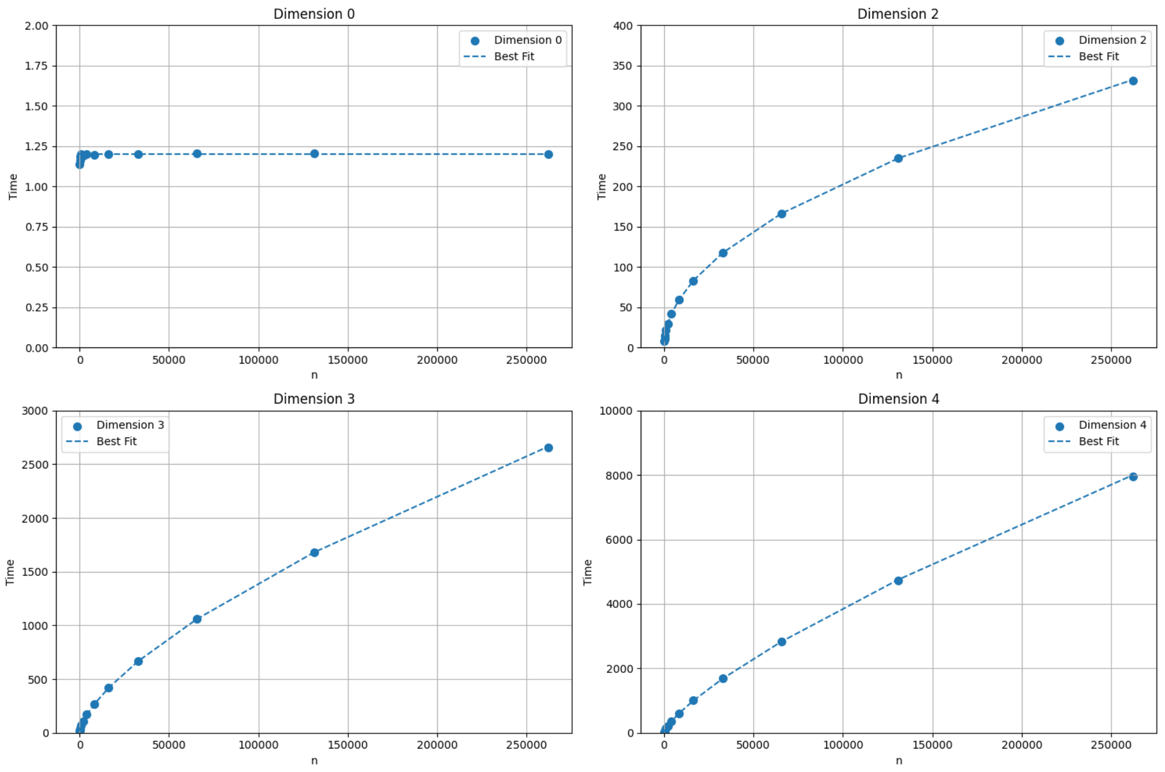


Fig. 5: Points against estimated function of best fit

As can be seen to the plots above, the (x, y) coordinates for the average minimum spanning tree size truly closely align with the best fit lines found.

2.4 Asymptotic Analysis of MST Size

Through our analysis, we can conclude how each of the graphs behave asymptotically.

First, we have that dimension 0 is constant, converging to 1.20145. This makes sense because we add edges with random length, and so as n goes up in size, the range that the smallest edges can take on decreases. That being said, the size of the MST can not really decrease or increase much further.

Then, we found that dimension 2 is proportional to $n^{1/2}$. We noticed that the MST weight increases as a function of n and there is no bound. Furthermore, dimension 3 is proportional to $n^{2/3}$ and dimension 4 is proportional to $n^{3/4}$ (where, for all of these, n is the number of nodes). We found this convergence property to be quite intriguing. In particular, as n tends to infinity, for dimensions 2, 3, 4, $f(n)$ still grows without bound since $\lim_{n\to\infty} n^{(d-1)/d} = \infty$.

The reason for this unbounded growth lies in the increasing complexity of the graphs as the dimensionality increases. In higher dimensions, there are exponentially more possible connections between nodes, leading to longer minimum spanning trees. This phenomenon further contributes to the unbounded growth of the MST weight since there are fewer constraints on edge lengths, allowing the MST to expand more as the number of n goes up.

2.5 Were our results surprising?

Yes, our results were surprising. Before starting the progset, we predicted that the dimension wouldn't make that much of a difference on MST size because if you have enough points then they will all be fairly close together. We learned through our analysis, however, that this wasn't the case. Another aspect we thought was surprising was how increasing the number of nodes caused such a drastic change in the MST size for dimensions 2 through 4. We figured that increasing the number of nodes wouldn't change that much because more points would mean that the points were closer together overall, so the corresponding edges would be smaller. Clearly, the number of points did have a larger affect on the MST than any shortening of edges that may have been in between.

3 Experimental Analysis

3.1 Runtime Analysis

Our runtime is primarily dependent on two factors, the generation of the graph and the running of Prim's algorithm, which also involves the sorting time of our binary heap. The generation of our graph runs in $O(n^2)$ because there are n nodes and there need to be n edges drawn (in the 2 through 4 dim case, this also involves a comparison). Prim's algorithm with a binary heap runs in $O(m \log n)$ as proved in lecture, where m is the number of edges. Under the hood, inserting and extracting from a binary heap is $O(\log n)$, which does not add to the runtime because it has lower time complexity. Since the upper bound for the number of edges is n^2, this gives us $O(n^2 \log n)$ so our total runtime is $O(n^2 + n^2 \log n) = O(n^2 \log n) \approx O(n^2)$, as we prune the number of edges enough so that the $\log n$ isn't factored in as much.

3.2 Asymptotic Time in Seconds of the Algorithm(s)

We also generated a table based on the runtimes for each dimension based on increasing node size, which is as follows. The runtimes do scale pretty similarly, and any differences are most likely due to the extra calculations (squares, powers) that are incurred as dimensions increase, as well as the efficiency of our pruning algorithm for each particular dimension.

Table 2: Time Results for Different Dimensions

Nodes	Dimension 0	Dimension 2	Dimension 3	Dimension 4
128	0.004	0.005	0.004	0.003
256	0.01	0.01	0.009	0.008
512	0.026	0.028	0.027	0.025
1024	0.019	0.016	0.018	0.095
2048	0.069	0.054	0.065	0.083
4096	0.259	0.201	0.245	0.317
8192	0.98	0.758	0.921	1.228
16384	3.794	2.933	3.538	4.788
32768	14.555	11.04	13.771	18.842
65536	57.153	44.541	55.548	76.508
131072	227.699	176.359	220.608	303.448
262144	910.419	698.878	889.723	1212.05

The time results demonstrate that as n increases, run time to generate the graph and find the MST increases. Indeed, we believe that the factor that dominated the time in seconds to find the MST was actually *generating* the graph in the first place, whereas, due to our pruning, the finding of the MST would run more quickly. This is because the pruning aspect helped Prim's algorithm avoid looping over large edges that would not be included in the MST anyway.

We also found that the time it takes to run our algorithm increases as the number of dimensions increases. In addition, the time it takes to actually do the math to calculate the distances between vertices also takes quite some time, which is why we ended up generating our own recursive square root function to save time.

3.3 Correctness of any Modifications from the Standard Algorithms Presented in Class

We used Prim's algorithm without any modifications in our implementation of this programming set, so no new correctness proofs need to be conducted. Our binary heap was implemented in the standard way with one correction. Instead of storing integers it stored tuples that held one float and one int. The values being compared for swapping were the floats, as these were the edge weights. The integers held the destination node. We utilized a minHeap, which is the specified binary heap for Prim's algorithm.

In terms of the correctness of the trials, we can first look at how our algorithm generates the graphs. The graphs are generated based on dimension. If the input dimension was 0, then we simply generate an edge weight randomly from 0 to 1 from every vertex to every other vertex. If the input dimension is non-zero, we generate the appropriate amount of coordinates randomly from 0 to 1. WLOG, say our input dimension was 2. Then we'd generate an x, y coordinate randomly for all n nodes. Once that is done, every vertex would be compared and an edge would be given a weight equal to the distance between the two nodes using the distance formula. This would appropriately output the correct edge weights that are specified in the progset description. Finally, we would generate a graph and run Prim's algorithm once for every number of trials, adding the minWeight to a value we called sum. Once that was done, we found the average minWeight by dividing by the number of trials.

3.4 Correctness of Our Prunning Algorithm

The assignment description recommended finding a $k(n)$ function for each dimensionality of graph type and letting this function guide whether or not to add an edge. The idea is that we do not add edges to our graph that have a close to a zero probabiliy of being in the final Minimum Spanning Tree.

To determine what this $k(n)$ actually is, we ran 5 trials for each value of n of $n = 128, 256, 512, 1024, 2048, 4096$. We then found the output produced from our execution and plotting that in a graph, using the software Desmos. This plot, for each dimension, demonstrates the largest edge found in the Minimum Spanning Tree.

In order to find the maximum edge along the way, we had to modify our Prim's algorithm and our main to be able to find the largest edge encountered. This was done like follows:

```
double prim(vector<vector<pair<float, int>>> adj, int n, float &largestEdge)
{
  BinaryHeap heap;
  vector<float> dist(n, 1e9);
  vector<bool> visited(n, false);
  double totalWeight = 0.0;

  largestEdge = 0.0; // Initialize largestEdge to 0

  visited[0] = true;
  dist[0] = 0;

  for (const auto &edge : adj[0])
  {
    heap.insert(edge);
  }

  while (!heap.empty())
  {
    auto minEdge = heap.extractMin();
    float minWeight = get<0>(minEdge);
    int to = get<1>(minEdge);

    if (visited[to])
      continue;
    totalWeight += minWeight;

    largestEdge = max(largestEdge, minWeight); // Update largestEdge if needed


    visited[to] = true;
    for (const auto &edge : adj[to])
    {
      int new_to = get<1>(edge);
      float new_weight = get<0>(edge);
      if (dist[new_to] > new_weight && !visited[new_to])
      {
        dist[new_to] = new_weight;
        heap.insert(edge);
      }
    }
    for (const auto &edge : adj[to])
    {
      int new_to = get<1>(edge);
      float new_weight = get<0>(edge);
      if (!visited[new_to])
      {
        dist[new_to] = new_weight;
        heap.insert(edge);
      }
    }
  }

  return totalWeight;
}
```

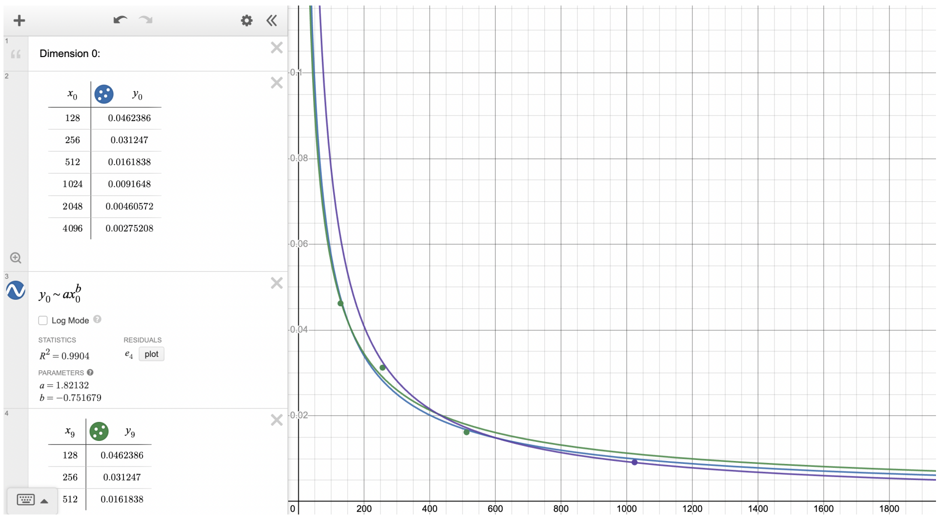
Below, we demonstrate our graphs that plot the largest edge found in the MST as a function of number of vertices, which are represented with Desmos for each dimension used (0, 2, 3, 4). Our pruning method involved a bit of experimental trial and error using Desmos. We added a table which represents this highest edge weight included in the MST from $n = 128$ to $n = 4096$ with 5 trials for each dimension.

Then we created a function of best fit of the form $y = ax^b$ where x is the number of vertices (or edges is the 0 dimensional case) and y is the largest edge weight in the MST. Then we plotted another function that is always larger than the function of best fit, such that we will only prune edges that will never be in the MST.

If we produced a pruning algorithm that was incorrect, then this MST would have fewer than $n-1$ edges total. In our code, we check to ensure that the final MST produced has $n-1$ edges and if not, then we rerun that trial to ensure that that trial is accounted for. If there are NOT fewer than $n-1$ edges in our MST, then we know that we obtained the proper MST, because the MST will always include the edges with the smallest weight possible such that all n vertices are included.

However, we believe that the probability of having fewer than $n-1$ edges in our MST is so improbable that a trial will likely never have to be rerun. We empirically tested our different $y = ax^b$ cutoff/ threshold function to ensure that no edges would ever be left out that should be included.

Additionally, we show that if, after pruning, there exists MST, the algorithm finds the MST. This is because our algorithm runs exactly how the typical Prim's algorithm runs (more details in the modifications section) and thus if there exists an MST, it will use the shortest possible edges in order to obtain the MST, and since the only edges remaining are the edges that could only possibly be in the MST, then this MST will be found.


Dimension 0:

x_0	y_0
128	0.0462386
256	0.031247
512	0.0161838
1024	0.0091648
2048	0.00460572
4096	0.00275208

$y_0 \sim a x_0^b$

Log Mode

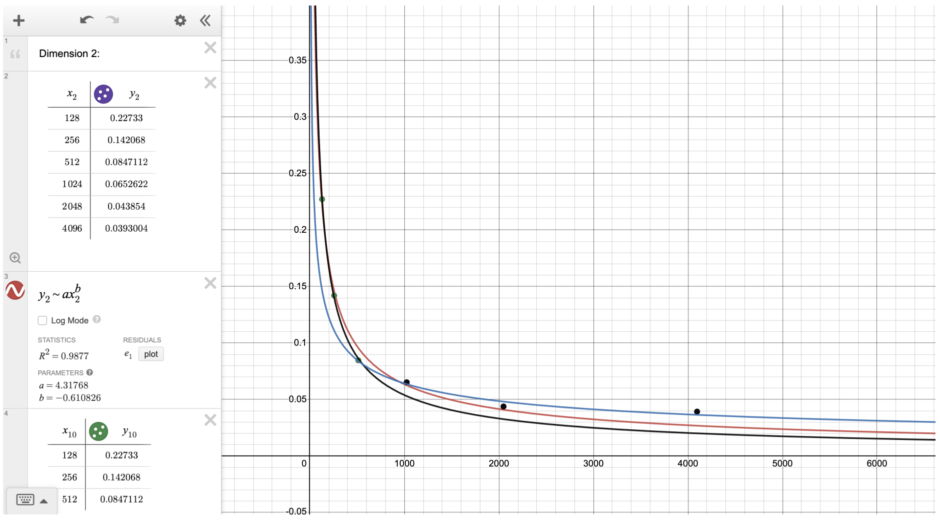
STATISTICS
$R^2 = 0.9904$

RESIDUALS
e_4 plot

PARAMETERS
$a = 1.82132$
$b = -0.751679$

x_9	y_9
128	0.0462386
256	0.031247
512	0.0161838

Fig. 6: Desmos fit for Dimension 0


Dimension 2:

x_2	y_2
128	0.22733
256	0.142068
512	0.0847112
1024	0.0652622
2048	0.043854
4096	0.0393004

$y_2 \sim a x_2^b$

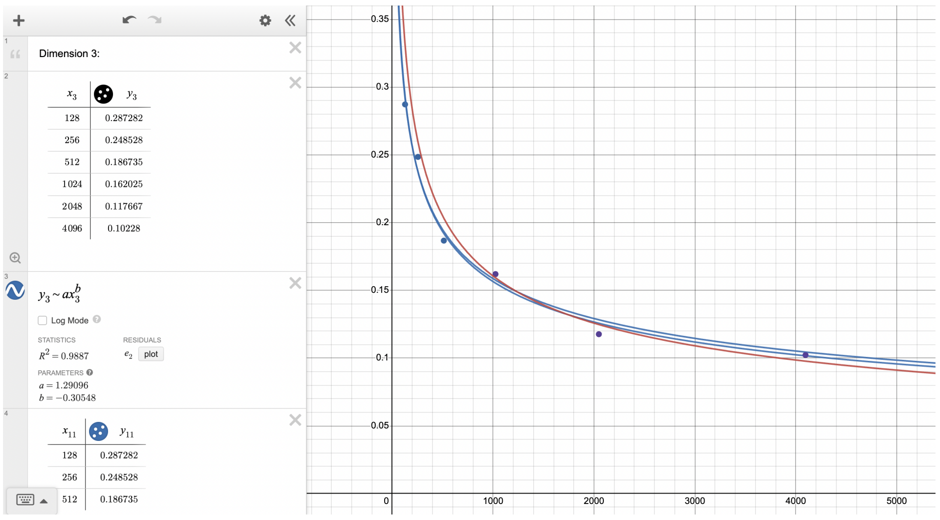
Log Mode

STATISTICS: $R^2 = 0.9877$ RESIDUALS: e_1 plot

PARAMETERS: $a = 4.31768$, $b = -0.610826$

x_{10}	y_{10}
128	0.22733
256	0.142068
512	0.0847112

Fig. 7: Desmos fit for Dimension 2


Dimension 3:

x_3	y_3
128	0.287282
256	0.248528
512	0.186735
1024	0.162025
2048	0.117667
4096	0.10228

$y_3 \sim a x_3^b$

Log Mode

STATISTICS: $R^2 = 0.9887$ RESIDUALS: e_2 plot

PARAMETERS: $a = 1.29096$, $b = -0.30548$

x_{11}	y_{11}
128	0.287282
256	0.248528
512	0.186735

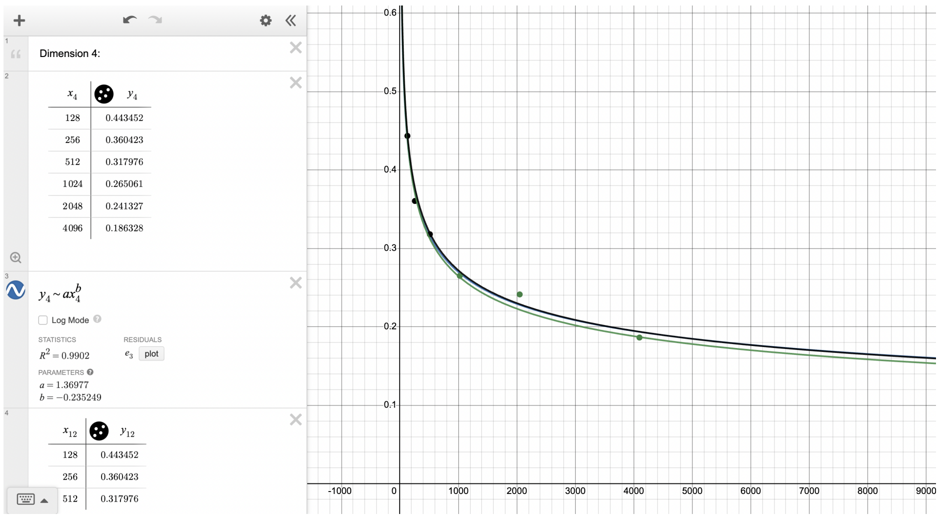
Fig. 8: Desmos fit for Dimension 3

3.5 Computer Cache Size Effect

Before we pruned our tree before running our Prim's algorithm, Noah found that his computer would run out of memory around when $n = 32768$, especially if he had other apps running in the background. After optimization, our algorithm never caused my computer to run out of memory, even with other apps running.

3.6 Random Number Generator Experience

The random number generator we used was mt19937, which is "A Mersenne Twister pseudo-random generator of 32-bit numbers with a state size of 19937 bits." Our research shows that the distribution of numbers generated are uniform from 0 to 1. From our personal usage experience, our numbers seemed to be pretty random as they followed the LLN in that the MST always converged to the same values for the same number of nodes each time. We also only had a single seed in our code, rather than multiple seeds. Once a PRNG is seeded, it will produce the same sequence of numbers every time if given the same seed. This property is useful for debugging and reproducibility because it allows us to recreate the same sequence of random numbers.



Dimension 4:

x_4	y_4
128	0.443452
256	0.360423
512	0.317976
1024	0.265061
2048	0.241327
4096	0.186328

$y_4 \sim a x_4^b$

Log Mode

STATISTICS

$R^2 = 0.9902$

RESIDUALS

e_3 plot

PARAMETERS

$a = 1.36977$

$b = -0.235249$

x_{12}	y_{12}
128	0.443452
256	0.360423
512	0.317976

Fig. 9: Desmos fit for Dimension 4

4 Conclusion

In conclusion, our exploration utilizing Prim's algorithm for finding the minimum spanning trees of complete undirected graphs, combined with the analysis of edge weights under four distinct conditions, has shown really interesting and reproducible results. By varying the dimensionality of the graphs and observing how the length weights of the MSTs evolve with the number of vertices, we have found surprising patterns that can be helpful for real-world applications of reducing problems down to finding the MST. That is, by extrapolating this scaling behavior, we can make informed predictions about the lengths of MSTs for very large graphs, helping in the estimation of computational resources required for analyzing such structures.

Our empirical findings suggest that the relationship between the number of vertices included (which is n) and the length of the MST ($f(n)$) can be characterized by a power-law function. Specifically, for the d-dimensional case, we propose the functional form $f(n) = an^{(d-1)/d}$ as a good fit to our experimental data. We showed that a serves as a scaling constant.

Overall, our study contributes to a deeper understanding of the behavior of MSTs in various complete graphs and provides a good framework for further theoretical investigations and practical applications including network design and computational geometry.

4.1 Acknowledgements

We would like to thank Ed for the opportunity to complete this Progset. We would also like to thank the CS 124 TFs for accommodating us during their lovely office hours. Special thanks to the professors for teaching us what an algorithm is.