CS 1240 Programming Assignment 1

Rafael Jacobsen and Ipek Eryilmaz

February 2025

Contents

1 Introduction

In this programming assignment we investigated the expected value of the weight of the minimal spanning trees (MSTs) for three different types of graphs: complete graphs with uniformly sampled weights, hypercube graphs, and complete graphs with edge weights sampled from Euclidean distances within a d-dimensional hypercube.

This report aims to trace the iterative improvements we have made to our implementation (see section 2) as well as offer some mathematical intuition for our results (see section 3). As we approached this question from a computational point of view, the bulk of the report will concern the optimization of our implementation and justification of the decisions we made.

2 Implementation

In this section, we will discuss our implementation of the MST algorithm, including the choices of algorithms and data structures used. We will also discuss adjacent topics such as the generation of graphs and graph representations. Finally, we will aim to establish an asymptotic runtime for our implementation and compare this with the data on our runtimes.

2.1 Graph Generation and Representations

Our first implementation for generating the graphs used an $n \times n$ matrix to store the values of the weights. We set the edge weights by iterating over the row and column number corresponding to the upper triangle of the matrix and sampling the uniform pseudo-random distribution for each entry.

This implementation not only failed to use hardware optimizations for vector operations, or Python library functions for manipulating arrays, but was also not the natural choice of graph representation for a later modification to our implementation. Specifically, even if we optimized generation (e.g. by generating an nxn matrix of uniform random variables using library functions, transposing, etc.) throwing away high-weight edges made adjacency lists a more natural choice for our representation.

Therefore, in our final iteration of the implementation, we generate the graph directly as an adjacency list with the appropriate cutoff applied. Furthermore, we use the numpy libraries matrix optimization to try to mitigate the inherent slowdown of python for loops. Although cutoff implementations could introduce the possibility of skewing the expected MST weights if one is not careful, we argue in Section 2.3 that this is not an issue in our implementation.

2.2 Runtimes

For the complete graph, we first generate n^2 uniform random variables in the form of an $n \times n$ matrix of numbers. We then loop over two indices corresponding to the upper triangle of the matrix (not including the diagonal) that are less than the cutoff and insert each such edge in two places into the adjacency list. Therefore, the runtime of this functions will be the time required to generate the random matrix, plus the time required to loop over the indices and insert (which will be proportional to $c\frac{n(n-1)}{2}$ where c is the cutoff), i.e. $O(n^2)$.

For the hypercube graph, we first generate a random matrix of size $n \times \lceil \log n \rceil$. We then loop over each of the $\frac{n\lceil \log n \rceil}{2}$ edges to insert them into the adjacency list. Then, our runtime is $O(n \log n)$.

For the d-dimensional unit hypercube graph, we first generate an $n \times d$ random matrix, then we iterate over each of the $\frac{n(n-1)}{2}$ edges to compute the edge weight using the Euclidean distance formula. Therefore, we again have a runtime of $O(n^2)$ despite sampling fewer uniform random variables for $d < n$.

2.3 Choice of MST

As stated during lecture on February 12, with n defined to be the number of vertices and m as the number of edges, the runtime for Prim's algorithm using binary heaps (which we use in our implementation) is

$$O((m+n)\log n)$$

while the runtime for Kruskal's algorithm is

$$O(m \log m)$$

which is just the runtime for sorting a list of m elements. At the beginning of the assignment, when we were not removing edges from the graph, the number of edges for a complete graph was

$$m = \frac{n(n-1)}{2}$$

Therefore, for Prim's

$$O\left(\frac{n(n+1)}{2}\log n\right) = O(n^2 \log n)$$

For Kruskal's

$$O\left(\frac{n(n-1)}{2}\log\left(\frac{n(n-1)}{2}\right)\right) = O(n^2 \log n)$$

Therefore, asymptotically, have the same runtime.

For the hypercube graph, we connect a vertex to another vertex if the difference between their values is 2^i for some $i \in [k]$ where $k = \lfloor \log n \rfloor$. Therefore, for a hypercube of $n = 2^k$

vertices, we know that any given vertex will be connected to $k-1$ many vertices; therefore, there will be

$$\frac{2^k(k-1)}{2} = \frac{n(\log n - 1)}{2}$$

many edges for a graph with n vertices. Therefore, Prim's will run in time

$$O\left((n\log n)\log n\right) = O(n(\log n)^2)O\left(n\frac{(\log n)^2}{\log\log n}\right)$$

while Kruskal's will run in time

$$O\left(n\log n\log(n\log n)\right) = O(n(\log n)^2 + n\log n\log\log n) = O(n(\log n)^2)$$

Observe that the asymptotic runtimes given above are for when the graph is represented by an adjacency list rather than an adjacency matrix (as is the assumption for all graph algorithms we have looked at so far during lecture). Although this does not make a difference for the complete graph, for the hypercube graph, if we were not applyiung cutoffs and using an adjacency matrix the runtime would be dominated asymptotically by the $O(n^2)$ time required to iterate through the nxn matrix (although the actual runtimes could still be different).

One important factor for our preference of Prim's algorithm was that the runtime can be improved by the use of d-ary heaps or Fibonacci heaps. As covered in lecture on February 24, although there are very efficient data structures (e.g. UnionFind) to implement Kruskal's algorithm, the need to sort the vertices is a significant bottleneck for the runtime, especially in the case of dense graphs such as complete graphs.

For the case of the complete graph, we would expect applying the same cutoff for different values of n to keep the asymptotic behaviour the same and scale them by c. However, in our implementation, we chose to change the cutoff based on the maximum value of an edge in the MST for a graph on n edges. Therefore, we expect to get a runtime that is possibly less than those established above, with a more strict bound only possible based on our cutoff function, which we discuss in Section 2.4.2.

2.4 Removing Edges

Having observed through the values outputted by the naive implementations of Prim's algorithm that for large n, the maximum weights of the edges used in the minimum spanning tree were consistently small. Therefore, we decided to remove higher valued edges from our graph representation. We will first discuss below that establishing a cutoff results in either the minimum spanning tree or a non-spanning tree. We will then discuss the empirical cutoffs we have established and some considerations in establishing those factors. We will finally argue that this modification does not skew our expected values for the total weights of the MSTs.

2.4.1 Correctness

We aim to show that removing edges e of weight $w(e) < c$ may result in no spanning tree existing due to the graph not being connected, but that it will never result in a minimum spanning tree with a higher total weight. Having shown this, the correctness of our modified algorithm follows from the correctness of Prim's.

Let $G = (V, E)$ denote the graph with all edges present and $G' = (V, E')$ denote the graph with the edges of weight larger than some cutoff c removed. Let $T = (W, F)$ be an MST of G. If $\forall e \in F$, $w(e) < c$ then all such $e \in E'$ also, i.e. T remains a subgraph of G'. Since T was a spanning tree of G, $W = V$; therefore, T is a spanning tree of G' also. Since no edges were added and none of the edge weights decreased, T will still be the minimum of the spanning trees of G'. Therefore, the fact that T (or another MST) will be outputted by Prim's follows from the correctness of the algorithm as an MST finding algorithm.

Assume, then, that there exists $e_1, \ldots, e_k \in F$ for some values of i such that $e_i \notin E'$. Then, $X = (W, F \setminus \{e_i, \ldots, e_k\})$ will be a forest that is a subgraph of both T and X. Observe that since there were no cycles in T, there will be no cycles in X and $|W| = |F| - k - 1$. Since one property of the Proposition 1 in Lecture 6 holds and the other doesn't, the third shouldn't hold as well (for otherwise we would get a contradiction with the Proposition). Therefore, we know that X is not connected. Take one connected component of X. Assume for the sake of contradiction that there exists a spanning tree in G'. Since this connected component is a subgraph of a tree in G, by the cut property, we know that it can be made into an MST by choosing an edge crossing out of X. However, if there is such an edge in G', then it should have a weight less than c, which implies that this should have been an edge in T as well due to T being a minimum spanning tree. Therefore, we conclude that there exists to spanning tree in G'. Therefore, Prim's will not have a set of edges of size $|V| - 1$, i.e. the algorithm will return no result and not an incorrect result.

2.4.2 Establishing a Cutoff

Having implemented Prim's correctness with a cutoff, we ran the naive implementation of Prim's algorithm on a number of smaller size graphs and returned the array d of edge weights, and found the maximum edge weights for a number of trials. We then plotted the maximum edge weight vs number of vertices as a log-log graph and found the line of best fit, which gave us a line of the form

$$\log(w) = m \log(n) + b$$

where w denotes the weight of the maximum edge. Thus, we obtained w as a function of n with two fit parameters b and m.:

$$w = e^b \cdot n^m$$

Having found an expression for the maximum weight edge of the MST, we multiplied the value of b by 1.5, which we concluded to be a reasonable tradeoff between the reduction in

time due to the decreased number of edges and the increase in time when one has to repeat the process of finding the MST for the given number of trials.

To ensure that the invalid result returned wouldn't skew our results, we sampled a new group of the given trial number each time one of the MST finds resulted in an invalid answer. As this is a slow process, we have chosen to not attempt a stricter cutoff.

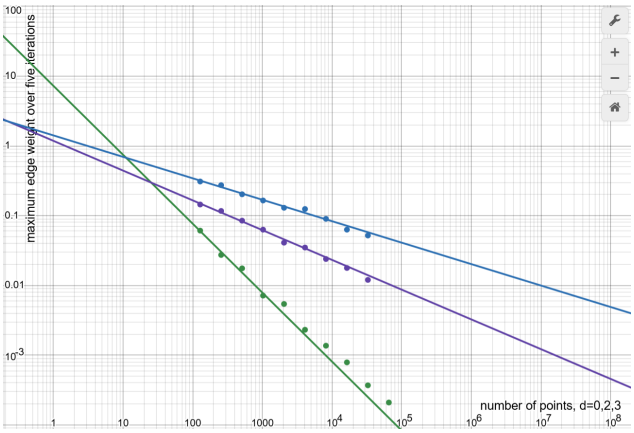

Figure 1: Desmos log-log regression for $d = 0, 2, 3$ cases. We calculated the optimal cutoff in this way. (green is $d = 0$, purple is $d = 2$, and blue is $d = 3$.

2.5 Priority Queue Implementation

For the implementation of the priority queue used in Prim's algorithm, we use a binary heap utilizing an array as the underlying structure. Although binary heap does not have as low asymptotic runtimes as some alternatives, the ease of implementation and the simplicity of the underlying data structures make it a good choice for our implementation.

3 Results

3.1 Expected Values and Runtime by Graph Type

In order to get information on higher values of n, we ran the python script on the fas-rc cluster with 512 gigabytes of RAM.

- Complete Graph on n Vertices

n	Runtime (s)	MST Size	Max Edge Weight
128	0.0522	1.249	0.06175
256	0.0032	1.181	0.02743
512	0.0094	1.213	0.01764
1024	0.0262	1.198	0.00716
2048	0.0826	1.212	0.00543
4096	0.2726	1.198	0.00232
8192	0.952	1.201	0.00137
16384	3.4334	1.21	0.00079
32768	13.0352	1.204	0.00037
65536	49.1216	1.204	0.00021
131072	196.392	1.199	0.00014

- Hypercube Graph on n Vertices

n	Runtime (s)	MST Size	Max Edge Weight
128	0.0018	11.908	0.49766
256	0.0036	20.465	0.42875
512	0.0082	37.337	0.34997
1024	0.019	66.904	0.37263
2048	0.043	120.603	0.42338
4096	0.095	219.342	0.3896
8192	0.2114	400.786	0.3705
16384	0.4914	739.85	0.34672
32768	1.162	1378.851	0.34161
65536	2.733	2571.156	0.34207
131072	6.152	4829.011	0.34253
262144	13.7148	9105.825	0.34347
524288	30.618	17180.693	0.37604
1048576	70.0574	32636.383	0.37524
2097152	157.5524	62056.137	0.33728
4194304	356.8576	118241.349	0.35806

- Unit Hypercube on n Vertices
  - $d = 2$

n	Runtime (s)	MST Size	Max Edge Weight
128	0.045	7.441	0.16987
256	0.0056	10.674	0.11251
512	0.0136	14.909	0.09153
1024	0.0356	21.142	0.06496
2048	0.099	29.531	0.050073
4096	0.2934	41.83	0.03283
8192	0.929	58.882	0.02362
16384	3.212	83.217	0.017
32768	11.7804	117.524	0.0133
65536	44.3276	166.015	0.00953
131072	170.4852	234.565	0.00658

– $d = 3$

n	Runtime (s)	MST Size	Max Edge Weight
128	0.003	17.544	0.36147
256	0.0062	27.837	0.23935
512	0.0156	43.873	0.18878
1024	0.0404	67.587	0.18253
2048	0.1092	107.18	0.13004
4096	0.3184	168.835	0.10537
8192	1.031	267.641	0.07842
16384	3.5486	422.348	0.0641
32768	12.922	668.574	0.05629
65536	48.4706	1058.898	0.04515
131072	185.0334	1677.875	0.03558

– $d = 4$

n	Runtime (s)	MST Size	Max Edge Weight
128	0.0028	28.826	0.48523
256	0.0062	47.494	0.37741
512	0.0306	78.54	0.30371
1024	0.0402	129.906	0.27939
2048	0.1104	217.034	0.21579
4096	0.3222	361.37	0.1877
8192	1.0336	602.645	0.16136
16384	3.6824	1009.414	0.14467
32768	13.6602	1690.661	0.11454
65536	52.2214	2828.685	0.09339
131072	200.1762	4740.124	0.08799

3.2 Discussion of Runtime and Expected Value

3.2.1 Graph Generation

For the complete graph on n vertices, the time required to generate the graph without a cutoff is plotted in Figure 2 along with a quadratic fit. This provides reasonable support for our argument that the runtime of graph generation would be $O(n^2)$. The quadratic fit gives the fit parameters that result in the following curve

$$T(n) = 6.3 \cdot 10^{-7} n^2 - 6.4 \cdot 10^{-5} n + 3.2 \cdot 10^{-2}$$

where the time is given in seconds.

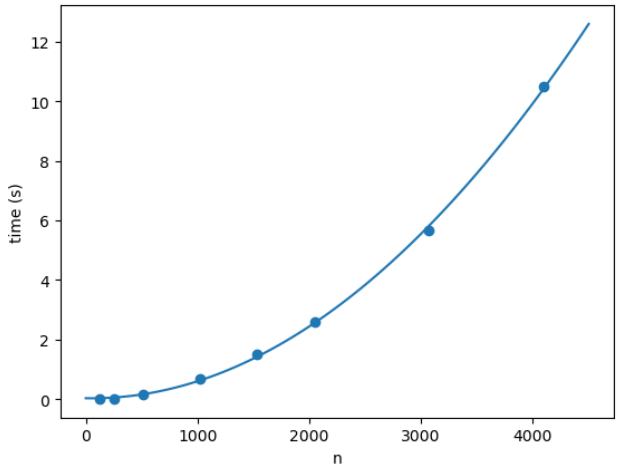

Figure 2: Runtime to Generate K_n

The corresponding plot for a cutoff of 0.1 being applied for all n is given in 3 where the fitted curve is

$$T(n) = 8 \cdot 10^{-8} n^2 - 1.6 \cdot 10^{-5} n + 1.5 \cdot 10^{-2}$$

Observe that a cutoff of 0.1, reduced the runtime by approximately $\frac{1}{8}$, which is reasonable considering that the runtime corresponding to sampling the uniform random variables is the same in both cases; however, the number of adjacency list insertions is less compared to the case without the cutoff. This also shows that the part of the function that is more significant in terms of runtime is the for loop and insertions rather than generating random variables.

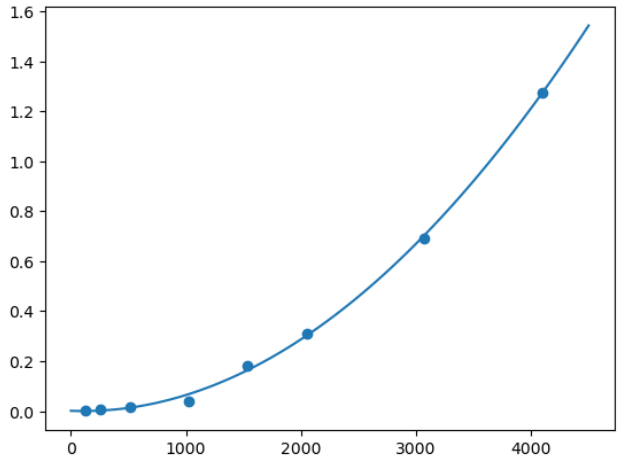

Figure 3: Runtime to Generate K_n with Cutoff 0.1

Another plot of interest is the time required to generate unit hypercube graphs of different dimensions. When we don't apply a cutoff, we get similar runtimes for dimensions 2 to 4, as shown in Figure 4, and it can be reasonable concluded that the runtime is quadratic in n. The fact that the runtimes are approximately the same shows that sampling the uniform distribution for the extra dimension and the extra substraction and squaring is not a significant factor for the runtime.

Applying a conservative cutoff of 0.5 for all dimensions and number of vertices, we obtain Figure 5. Observe that in this case, the growth of the runtime again appears to be quadratic. The fact that the most decrease in runtime happens in the hypercube graph with dimension $d = 4$ is consistent with the fact that the average edge weights for a higher dimensional hypercube will be higher; therefore, applying the same cutoff for all values of d will result in more edges being removed for higher d, which was reflected in our cutoff calculations as well (higher cutoffs for higher d). We can also show this by calculating the expectation of the square of the Euclidean distance between two points inside a d-dimensional hypercube as

$$E\left(\sum_{i=1}^{d}(x_i - x_i')^2\right) = \sum_{i=1}^{d}\left[E(x_i^2 + x_i'^2) - 2E(x_i x_i')\right] = \frac{2d}{3} - \frac{2d}{4} = \frac{d}{6}$$

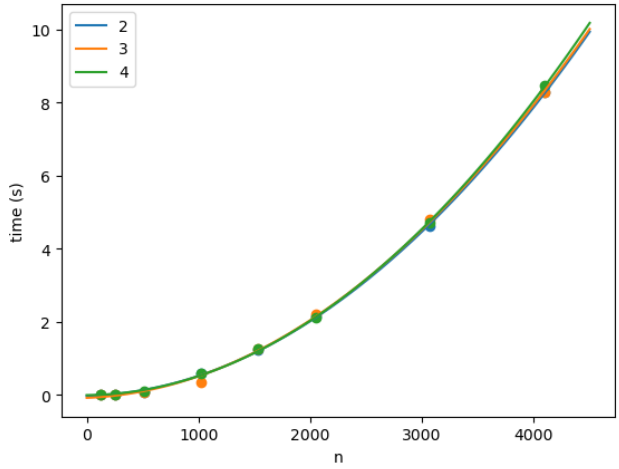

Figure 4: Runtimes for Unit Hypercube, No Cutoff

where for the second to last equality we use the fact that the positions are i.i.d. uniform random variables.

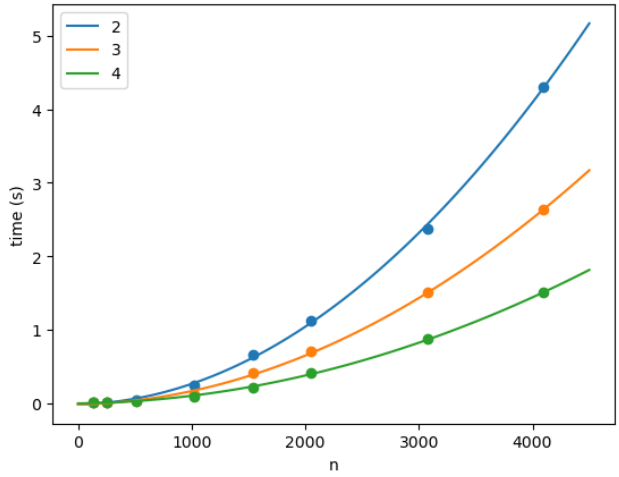

Figure 5: Runtimes for Unit Hypercube With Cutoff 0.5

3.2.2 Runtime Summary

Having thoroughly discussed the runtime for graph generation and MST algorithm in the sections before, we now give the runtimes with empirical constants:

- Complete Graph on n Vertices

The plot for the runtime is given in Figure 6 where a quadratic fit for approximating the runtime is reasonable due to the combination of the theoretical bound we have established of $O(n^2 \log n)$ and the n dependent cutoff we use. This will also be the case of Unit Hypercube Graphs. Therefore,

$$T(n) \approx 1.588 \cdot 10^{-8} n^2 - 3.982 \cdot 10^{-5} + 6.023 \cdot 10^{-2}$$

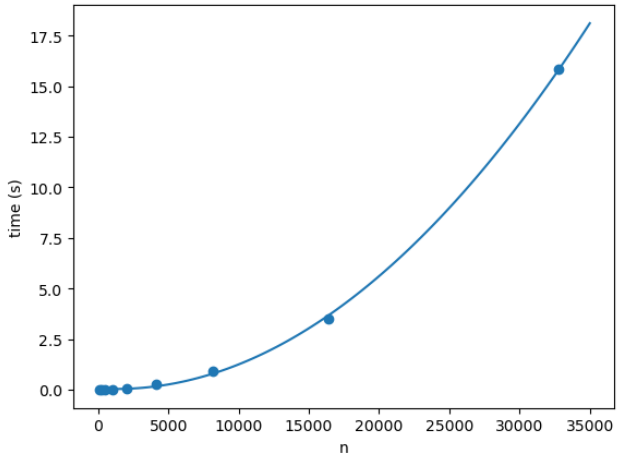

Figure 6: Complete Graph Runtime

- Hypercube Graph on n Vertices

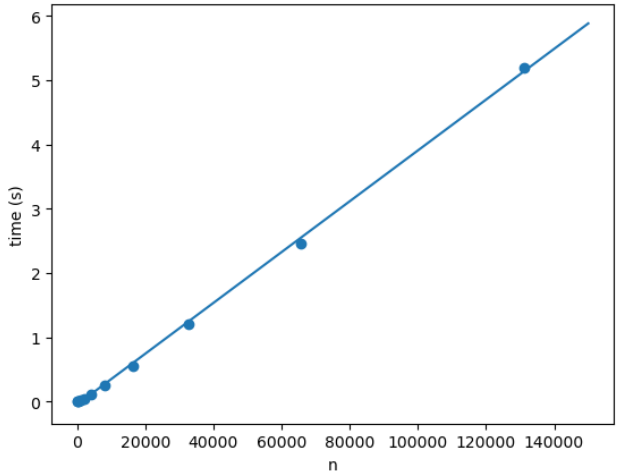

Figure 7: Hypercube Graph Runtime

The plot for the runtime of the overall algorithm for a hypercube graph on n vertices is given in Figure 7, where we have drawn a best fit line (linear). This is empirically justfied in the range we are looking at since a polyfit of degree 2 gives a coefficient for the the quadratic term that is about $8 \cdot 10^{-7}$ times the linear term and similarly a low value for the cubic. Therefore, the empirical runtime we obtain is approximated by

$$T(n) \approx 3.952 \cdot 10^{-5} n - 4.458 \cdot 10^{-2}$$

Since our hypothesis for the asymptotic behaviour was $O\left(n \frac{(\log n)^2}{\log \log n}\right)$, taking into account also the cutoff that becomes smaller (excludes a greater fraction of edges) as n increases, this is a reasonable result.

- Unit Hypercube Graphs with $d = 2, 3, 4$

  The plot for the runtime of the algorithm for a unit hypercube graphs with $2, 3$, and 4 dimensions are given in Figures 8 with the polynomial fits given by

$$T_2(n) \approx 7.939 \cdot 10^{-9} n^2 + 3.704 \cdot 10^{-5} n - 1.099 \cdot 10^{-2}$$

$$T_3(n) \approx 8.157 \cdot 10^{-9} n^2 + 4.898 \cdot 10^{-5} n - 7.308 \cdot 10^{-3}$$

$$T_4(n) \approx 8.246 \cdot 10^{-9} n^2 + 5.377 \cdot 10^{-5} n + 5.650 \cdot 10^{-2}$$

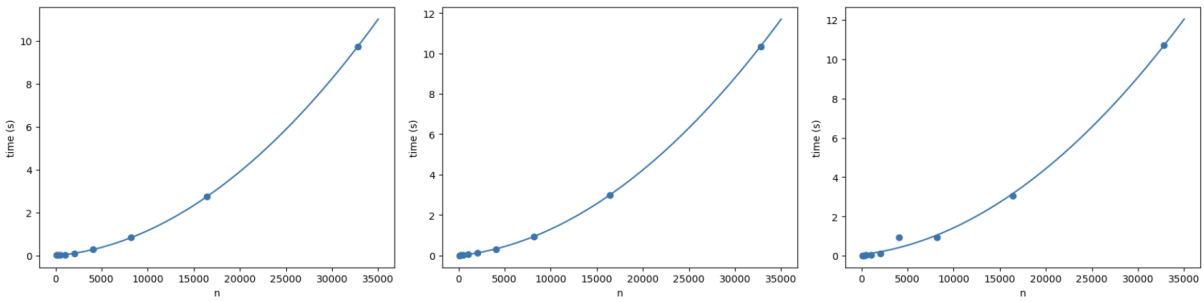

Figure 8: Unit Hypercube Graph Runtimes for $d = 2, 3, 4$

3.3 MST average weights

Performing a regression complete graph on n vertices, we can immediately see that the MST size converges to a constant value of approximately 1.20.

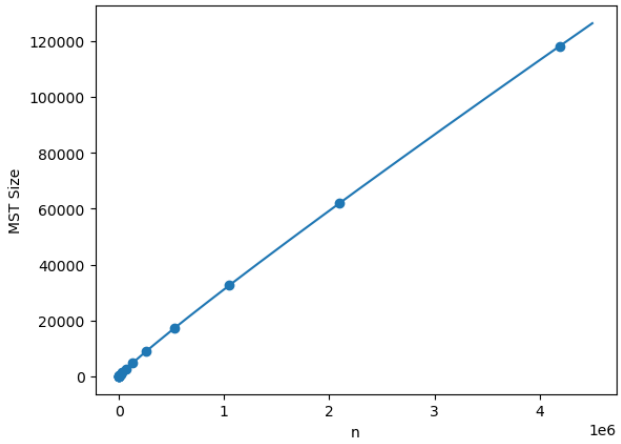

Figure 9: Average MST Size for Hypercube

The hypercube graph (see Figure 9), gives us a fit of

$$\text{size}(n) \approx 0.62063503 \cdot \frac{n}{\log_2 n}$$

The unit hypercube MSTs reveal nonlinear regressions, as seen in Figure 10. Specifically, doing a curve fit of $\text{size}(n) = c \cdot n^p$ gave regressions of

$$\text{size}(n) \approx 0.6643 \cdot n^{0.4978}$$

$$\text{size}(n) \approx 0.6807 \cdot n^{0.6628}$$

$$\text{size}(n) \approx 0.7426 \cdot n^{0.7435}$$

These exponents can be approximated as $n^{1/2}, n^{2/3}$, and $n^{3/4}$ respectively. Therefore, for the unit hypercube we can conclude that size grows

$$O(n^{1-\frac{1}{d}})$$

.

3.3.1 Intuition behind MST average weights

The unit hypercube is has the most immediately obvious explanation. If we divide the total weight of the MST we can get the average weight of each edge to be proportional to

$$\frac{n^{1-1/d}}{n} = n^{-1/d} = \frac{1}{\sqrt[d]{n}}$$

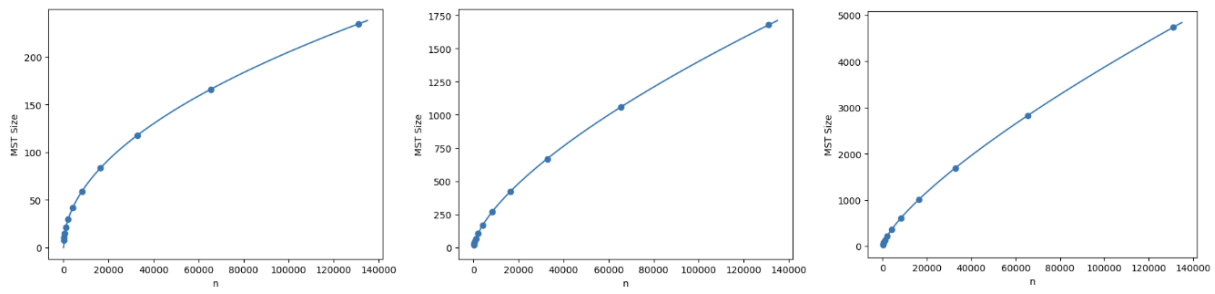

Figure 10: Unit Hypercube Graph MST

This makes intuitive sense because the average distance between points in a unit hypercube scales with $1/\sqrt[d]{n}$. We can visualize this by imagining a uniform distribution: in a line segment, the distance between adjacent points is clealy $1/n$. In a square, the points are laid out in a lattice with $1/\sqrt{n}$ points in each row so the distance is $1/\sqrt{n}$. In higher dimensions, the lattice is the same so each generalized row has $\sqrt[d]{n}$ points and the average distance between them is $1/\sqrt[d]{n}$. We can apply this same logic to the other two graphs. For $d = 1$, the hypercube connection law means that each vertex has approximately $\log_2 n$ connections. Therefore, the minimum connection scales with $1/\log_2 n$ and the average total connection is proportional to $n/\log_2 n$, which fits our regression. Finally, the complete graph has n connections per vertex, so each connection's minimum weight scales with $1/n$ and the total weight is therefore constant.

4 Conclusion

Using Prim's algorithm modified to exclude edges with weights above a certain cutoff we concluded convergence to a constant for complete graphs , $\frac{n}{\log n}$ scaling for hypercube graphs and $n^{\frac{d-1}{d}}$ scaling for unit hypercube graphs on n vertices for the expected value of minimal spanning tree (MST) sizes. We have also analyzed the improvements to runtime due to different changes to the implementation of Prim's algorithm and established the correctness of these modifications.