1 Averaged Weighted Size

For the functional form of the average MST, we know that the main difference between the dimensional spaces is the calculation of the euclidean distances for computing the weights. Additionally, we know that both pruning and neighborhood-based edge creation are used for Graphs 0 and 2-4, respectively, while Graph 1 is already fairly sparse given the conditions for creating an edge. Thus, we predict that the functional forms for the graphs are roughly $f(n) = nlog(n)$ as its a functional form less than n^2 but greater than a linear amount of n.

Size of Graph (n)	Graph 0	Graph 1
128	1.2193175572651571	12.678771123540447
	0.005901	0.002966
256	1.1659143511776604	20.937775502976276
	0.018242	0.007334
512	1.1882662866071587	37.12533063466131
	0.069599	0.017329
1024	1.2114116267656638	67.41202784186837
	0.254284	0.035296
2048	1.194320434742082	120.61447522056241
	0.954904	0.078916
4096	1.18936797099135	216.9607239391418
	3.646637	0.176946
8192	1.2022845548065175	400.20806820963264
	14.399205	0.416302
16384	1.2071318591470308	742.6632262281073
	59.323117	0.934806
32768	1.200614506527518	1374.6198662742522
	230.261832	2.289345
65536	NA	2577.603907877497
	NA	4.970963
131072	NA	4843.744798771024
	NA	11.034139
262144	NA	9099.654962892058
	NA	25.339428

Table 1: Average Weights and Runtimes for graphs 0 and 1. The values in the top row are the average MST weights while the values in the bottom row are the runtimes (seconds). *numtrials* was set to 5.

2 Size Relationship

Let n represent the size of the graph and $f(n)$ represent the average MST weight outputted by our algorithm. We determine that the asymptotic relationship for determining the average weight of the MST for a graph of n vertices is:

1. **Graph 0:** $f(n) = 2.67 \cdot 10^{-7}n + 1.2$

Size of Graph (n)	Graph 2	Graph 3	Graph 4
128	7.5346	17.336937721697744	27.949965015731713
	0.007938	0.017434	0.043903
256	10.739404535712765	27.039291767035177	47.15981605042733
	0.016598	0.042990	0.091383
512	15.05669001182218	43.01387322787632	78.3629085656989
	0.032352	0.075919	0.191444
1024	21.225648293854604	68.07955692888619	129.5287229853655
	0.068803	0.171383	0.410275
2048	29.68962662319401	106.9199259920108	216.81919923755044
	0.139622	0.329487	0.847547
4096	41.87586553557348	168.90071310972607	362.30733311930214
	0.291958	0.698360	1.842308
8192	59.09599910687589	266.91956495173235	603.7881072886106
	0.578324	1.474851	3.673733
16384	83.31777505116521	423.17159155060756	1007.526858188998
	1.243870	3.155555	8.414373
32768	117.59681130473446	668.6261506522414	1688.1976890390554
	2.554254	7.114254	19.684680

Table 2: Average Weights and Runtimes. The values in the top row are the average MST weights, while the values in the bottom row are the runtimes (seconds). *numtrials* was set to 5.

2. **Graph 1:** $f(n) = 0.0348 \cdot n + 102$
3. **Graph 2:** $f(n) = 0.689 \cdot n^{0.494}$
4. **Graph 3:** $f(n) = 0.703 \cdot n^{0.659}$
5. **Graph 4:** $f(n) = 0.781 \cdot n^{0.738}$

3 Determining Our Pruning Function $k(n)$

To determine the best $k(n)$, we guessed numerous, plausible functional forms and compared their results to the true value of the MST. To determine the true value to compare our guesses to, we adjusted our

Size n	True Value	$\frac{2log(n)}{n}$	$\frac{1}{n}$	$\frac{n}{log(n)}$
128	1.187928586511511	1.177084871518836	0.2410766528804492	1.210604688659769
256	1.2163819277018701	1.2088550064318289	0.24303445144277822	1.1949408779768003
512	1.2210730249586108	1.201454186321192	0.25288782382761893	1.1973635840005166
1024	1.205388527948638	1.1978712804439644	0.2570627914026218	1.201145025216167
2048	1.1944594621832298	1.2053618646409647	0.249634239056564	1.2057997741593798
4096	1.204469984994753	1.1951991286920598	0.25036094071540327	1.2035528827847484

Table 3: Average Weights determined for Graph 0 by different functions for the pruning function

pruning threshold such that all edges would be considered within the completed graph. From there, we

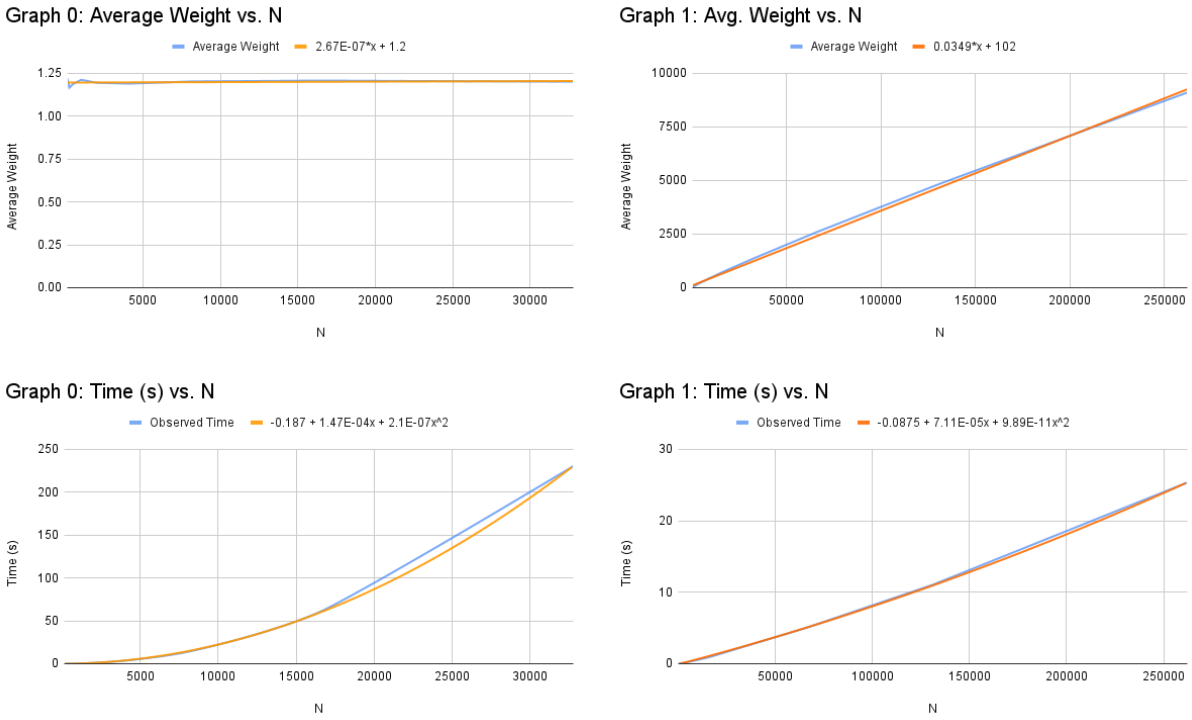

Figure 1: Function Plots: Average MST Weight and Time to Run for Graph Types 0 & 1

determined the average MST weight with our 5 trials a total of 3 times and took the average of the 3. We took a similar approach for each of our guesses for the functional form of our pruning function $k(n)$ until we reached a plausible solution.

Based on our experiments and values, our equation of $k(n) = \frac{2log(n)}{n}$ works well in pruning our 0-dimensional graph.

4 Algorithm Design

Our algorithm for computing the minimum spanning tree (MST) was developed by integrating several key components into a single cohesive solution. We implemented Kruskal's algorithm, which sorts all the edges by weight and then iteratively adds the smallest edge that does not create a cycle, stopping when $n-1$ edges have been added since any spanning tree of n vertices must have exactly $n-1$ edges. To efficiently detect cycles, we built a disjoint-set union data structure, incorporating path compression and union by rank to optimize the find and union operations, closely following the algorithms presented in class.

For graph generation, we considered three cases. In Graph 0, representing complete graphs with random edge weights uniformly distributed in $[0, 1]$, we applied a pruning function $k(n) = \frac{2\log(n)}{n}$ to generate only those edges with weights below this threshold (refer to Section 3); this is justified by the fact that MSTs tend to consist of low-weight edges. In Graph 1, the hypercube graph, each vertex is connected to vertices at distances corresponding to powers of two, with edge weights again generated uniformly at random; no additional pruning is necessary here. For the geometric graphs (Graphs 2, 3, and 4) where vertices are randomly placed in a d-dimensional unit hypercube and edge weights are determined

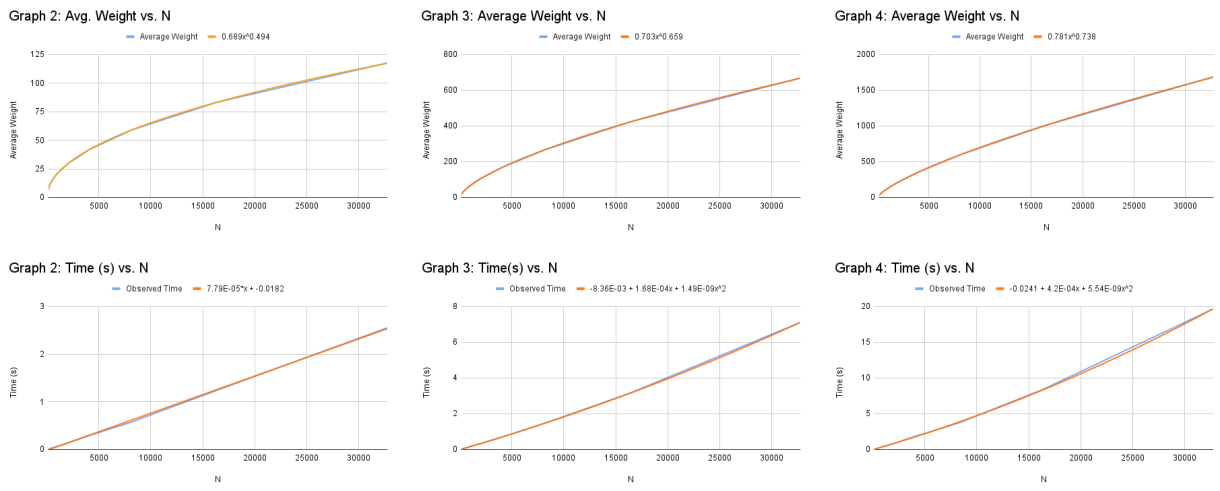

Figure 2: Function Plots: Average MST Weight and Time to Run for Graph Types 2 - 4

by Euclidean distances, we implemented a grid bucketing strategy. This approach divides the space into a grid with approximately $n^{1/d}$ cells per dimension, meaning each cell has a side length of roughly $n^{-1/d}$, ensuring that each cell contains about one point on average. We then generate edges only between points in the same or adjacent cells. This essentially serves as a pruning function and dramatically reduces the number of edges considered from $O(n^2)$, discarding all edges not so likely to appear in the MST.

This development process was iterative: we started with a straightforward implementation, identified performance bottlenecks for large n, and then introduced these optimizations. Extensive testing confirmed that our approach correctly computes the MST and scales efficiently for large graphs.

5 Runtime Analysis

For the hypercube and n-dimensional graph types, our algorithm achieves an optimized runtime of roughly $O(n)$ as we only consider the neighbors of a vertex for building the MST. As per lecture, we know that the runtime for Kruskal's algorithm is $O(mlog^*n)$. Thus, we will determine the runtime of our adjusted algorithm by considering the runtime of our graph creation steps.

Case 1: Graph 0
Since we implement a pruning function to limit the number of edges considered, we just need to check for roughly each pair of vertices whether the randomly determined weight value in [0,1] is less than or equal to the threshold value determined by our pruning function of $k(n) = \frac{2 \cdot log(n)}{n}$. Thus, our runtime for creating Graph 0 remains $O(n^2)$ as we perform a constant number of operations for roughly every pair of vertices.

Case 2: Hypercube
Since no pruning occurs for the hypercube case, we know that our runtime for constructing this graph is roughly $O(nlog(n))$ as we roughly traverse each value of 2^i that is less than or equal to n, loop through each vertex within our graph, and create an edge of random weight in [0,1] between, say, vertices a and b if and only if $|a - b| = 2^i$ for some i.

Case 3: d-dimensional Graphs
To simplify the graph for Kruskal's algorithm, we only consider vertices near within the neighborhood of

a vertex. We first determine the number of cells per side to be $n^{1/d}$, $n^{(1/d)^d} = n$ cells in the d-dimensional space. Next, we partition the d-dimensional space into cells, each with a size of $\frac{1}{side} = \frac{1}{n^{(1/d)}}$. Afterwards, we track where each cell is via cell map. Each of these steps takes $O(n)$ time as there are n cells to create and store in our dictionary. For each cell in our cell map, we only consider creating edges within the cell and between its neighbors. Since vertex values are randomly determined via d-dimensional values in [0,1], we know that the expected number of vertices per pair is $\frac{\text{number of vertices}}{\text{number of cells}} = \frac{n}{n} = 1$. Next, we determine weight values for each edge within the same or adjacent cell by computing the euclidean distance, which takes $O(d)$. Ultimately, the runtime for this step is a matter of considering the number of possible edges in the d-dimensional space alongside the probability of creating an edge with a vertex in the same cell or a neighboring cell. Since the outer cell space obtained when comparing a vertex to its neighboring vertices is roughly $(3 \cdot \text{side})^d$ with side $= n^{(1/d)}$, we know that the overall runtime for our graph construction of d-dimensions is $O(n^2) \cdot (\frac{3}{n^{(1/d)}})^d = O(n \cdot 3^d) = O(n)$.

6 Correctness

We claim that if the spanning tree exists after pruning the initial graph, the algorithm will find the MST. We will prove this via the proof from lecture on Kruskal's algorithm. We know that Kruskal returns the MST, if Kruskal processes the graph and returns a spanning tree, we know that it is the spanning tree.

Next, we must show that our graph constructions still show that our MST building algorithm is correct. We know that Kruskal's algorithm works by only considering edges below a certain threshold. We will demonstrate that the thresholds leveraged in our algorithm maintain this.

Case 1: Graph 0
When constructing a MST for a complete graph on n vertices where the weight of each edge is a real number chosen uniformly at random on [0,1], we know that our pruning function of $k(n) = \frac{2 \cdot log(n)}{n}$ allows us to find the correct MST while accounting for graphs of large n. Since the MST algorithm returns a tree connecting all vertices via the lowest-weighted edges, we know that it's highly unlikely that edges with large weights will be incorporated into the MST. Thus, it's plausible to build a MST by constructing a graph that only considers weights below a certain threshold. Additionally, this pruning function should scale with the size of the graph as the runtime of the algorithm can be optimized by pruning more edges in a larger graph relative to a smaller graph. Our experimentation with different functions for our pruning function suggests that $k(n) = \frac{2 \cdot log(n)}{n}$ preserves enough information for a MST to be built if it exists for a given graph. Thus, Kruskal's algorithm will properly work for a graph of this construction.

Case 2: Graph 1
As per the explanation from the runtime analysis, no pruning occurs for the hypercube graph, so Kruskal's algorithm will properly work for a graph of this construction.

Case 2: Graphs 2-4
Drawing from the explanation from the runtime analysis, we know that the euclidean distance between a vertex and an edge of consideration is roughly at most the length of a cell's side. This is only roughly so given the literal corner case of an edge being created at the diagonal corners of a cell. However, our threshold of the side of a cell is still valid in enabling our implmeentation of Kruskal's algorithm to run.

Given the correctness and runtime of Kruskal's algorithm as long as our graph construction processes, we have demonstrated that our algorithm is correct and runs in optimized runtimes as per the functional forms derived in section 2 of our write up.