An Analysis of Kruskal's Algorithm in 4 Dimensions using $k(n)$

Gabriel Mehra

February 22, 2023

1 Introduction

The following is an explanation and analysis of the code submitted to find a Minimum Weight Spanning Tree (MST) graphs of size n of the following sizes: {128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144}. We will have 4 sets of results to represent our 1-Dimension (randomly generated weights), 2-Dimensional (randomly generate node coordinates that determine the distance-weights), 3-Dimensional (same as 2D but with 3 coordinates), and 4-Dimensional (same as 2D and 3D but with 4 coordinates) shape areas. We also have calculated runtimes for finding each of these MST's. We will analyze how both the runtimes and the MST weights change (or don't change) with n and they also might change between dimensions.

Usage: To run the code, there are four C++ files with "1D", "2D", "3D", and "4D" in the titles. First, simply type "make" and then the name of the each of the files. Then you're all set to run any four of them.

2 Classes

The code makes use of two different classes. The first is to define the DSU (Union-Find) abstract data structure, and the second is to define what we want our Graph to look like.

2.1 DSU / Union-Find Class

In order to run an algorithm that finds the MST of a graph (ultimately, Kruskal's algorithm), we make use of the Union-Find abstract data structure that was introduced in class. We have arrays named *parent* and *rank*. The purpose of the *parent* array is to keep track of which nodes are the parent of the other nodes. The purpose of the *rank* array is to create a hierarchy of subtrees so that we know which subtree is larger for when we do our union by rank optimization.

2.1.1 find()

The purpose of this function is to find the parent of a given node. We want to do this because, when running Kruskal's algorithm, we need to be able to check whether or not adding a specific edge will create a cycle in the graph. Therefore, if the candidate edge does not connect two nodes that are contained within the same set, then we are safe to add it to the MST. find() is used with path compression to find the root parent of the given node.

2.1.2 Path Compression

Our implementation of find() uses Path Comporession to make the process of finding the root parent of a given node much quicker. When a child node has many levels parents to get to it's root parent node, it will point not only the child node, but also all nodes in between directly to that root parent node. This way, any time any of the child node or in-between parent nodes get find() called on them, it removes the in between steps that it once did t find the root parent node.

2.1.3 link() and union_()

These two functions work together to add new edges to the MST. Once a candidate edge has been checked to not create a cycle (by using find), it gets "Unioned" into the MST. Using Union by Rank logic, we add the edge into the MST.

2.1.4 Union by Rank

This concept is used in the link() function in our implementation. Instead of blindly adding the node into the DSU, we instead add the node in such a way that the *rank* depth of the entire tree stays as small as possible. This way, we can make lookups of nodes much quicker as well as save space. With union by rank, we always let the smaller tree be the child of the larger tree. This results in the total depth of the tree being smaller.

2.2 Graph Class

In order to find the MST, we need some way of actually storing the values of edges and nodes somewhere. Therefore, we make use of a Graph class that, for the 1D implementation, contains a vector of edges containing edge weight, and two node values representing the nodes it connects (in the 2D, 3D, and 4D implementations, the Graph class also contains a vector of nodes with coordinates dictating the edge lengths).

2.2.1 addEdge(double w, double v1, double v2) & addNode(double x_coord, double y_coord)

These functions add an Edge to the *edges* vector and a Node to the *nodes* vector respectively (*nodes* is not used in 1D). *edges* is a vector of vectors where each inner vector represents an individual edge of the graph (and each edge contains a randomly generated value for weight and the two nodes it connects).

2.2.2 getWeight(double x1, double y1, double x2, double y2)

This function (not used in 1D) simply takes in randomly generated coordinates for each node and calculates the distance between them using the distance formula. It is written slightly differently depending on the dimension. For 2D, this function uses the 2D distance formula (with two coordinates)to calculate an edge's weight (distance). As you might imagine, for 3D and 4D it does the same thing but for 3 and 4 coordinates respectively.

2.2.3 kruskal(int n)

This is the heart of our program. To answer the classic question: Yes, we have indeed chosen to run Kruskal's algorithm instead of Prim's.

Question: *Why did we decide to go with this algorithm?*

The core reason is that it has to do with what I like to refer to as out "cutoff function" for adding (or not adding) edges. This is a topic I will touch on later, but essentially if we use the cutoff function, it ends up being such that Kruskal's algorithm runs much faster than Prim's can.

Implementation:
In our implementation of Kruskal's algorithm, the class structure we have setup for both DSU's and Graph's comes together. Kruskal's algorithm begins by sorting the edges. We chose to do this by using the built-in C++ sorting function. This process can be done in $O(n \log n)$ time, and as it turns out, is asymptotically the heaviest part of our algorithm. Then we create our DSU that we built earlier. Then we iterate through each edge in *edges*, adding edges only when they don't have the same parent (using *.find()* to check that they don't create a cycle). If they don't create a cycle, then we *.union()* them into the MST and add their weight to a running total weight of our MST. Then, we return our weight once we have completed the edge looping.

Efficiencies:
In this implementation there are two central efficiencies that make the Kruskal's algorithm run faster.

1. Stopping (early) after exactly $n-1$ unions: Before iterating over the edges, we have created a counter *count* to keep track of the exact number of edges added to the MST so far. There is an *if* statement in *kruskal()* that checks, before we add each edge, whether or not if $n-1$ edges have already been added to the MST. If there have been, the loop immediately returns the current total weight and ends. We can be sure that we have found an MST in this case because, assuming all other parts of the algorithm are correct, there must be exactly $n-1$ edges in an MST. Therefore, if we have already unioned $n-1$ times, then there must be exactly $n-1$ edges, meaning we have an MST and don't need to continue iterating over the edges.

2. Cutoff function: As previously mentioned, we are making use of what I call a "cutoff" function, or $k(n)$, to limit the number of edges in our Graph. This functionality actually cannot

be seen within our *kruskal()* function, but its effects are felt within *kruskal()*. In the *main()* function, we have a nested *for* loop that generates $\binom{n}{2}$ edges. Within the nested *for* loop, right before we call our *addEdge()* function to generate an edge and append it to the *edges* vector, we have an *if* statement that only allows some edges through. The math is slightly different depending on which dimension file you are looking at, but the idea is the same across all of them. The idea is that some edges have a statistically improbable chance of being a part of the MST because their weight is so high. Recall that MST's find the *minimum* weight spanning tree, which means that we are taking the smallest edges possible to create a tree that touches every vertex. Therefore, we can carefully choose a function (depending on implementation and dimension) that prevents edges from being added to the graph in the first place if their weight is above a certain threshold.

Therefore, the question arises: How do you know what a good cutoff function $k(n)$ is for a given dimension?

A reasonable method of estimating a good cutoff function (and ultimately the method I ended up settling on) is to run the finished algorithm on several smaller values of n and calculate the highest value edge that gets added to the MST. For values of n from 128 to 4096, I decided to generate 5 MST's and select the highest weight edge between the five of them. Then I plotted these points (n, highest weight edge) on a graph for all four dimensions and used the online Desmos graphing calculator to find a line the best fits them. Note: To be safe, I made sure the line of "best fit" for the data points was a little above our data points. This way, we can still cut out plenty of edges while having some wiggle room in case of outlier edges sneaking through. I have included screenshots of what the $k(n)$ functions that I settle on looked like in Desmos along with the data points. Note: some data points might seem slightly "out of place". This is because I selected the max edge weight between 5 MST's, which can result in outlier max weight edges making their way on to the graph. After I got these sample data points, I realized in hindsight that it may have been more effective to get the average max weight over several trials and plot that instead. However, the strategy I ended up using still works and it is arguably safer (less likely to result in the algorithm unintentionally not finding an MST, since getting take the max over 5 gives more "wiggle-room"). Here are the aforementioned images of $k(n)$, which will launch us into a deeper analysis of our results:

3 Analysis:

3.1 $k(n)$ analysis:

Figure 1: **1 Dimension:** $k(n) = \frac{1}{n^{0.74}}$

Figure 2: **2 Dimensions:** $k(n) = \frac{1.4}{n^{0.43}}$

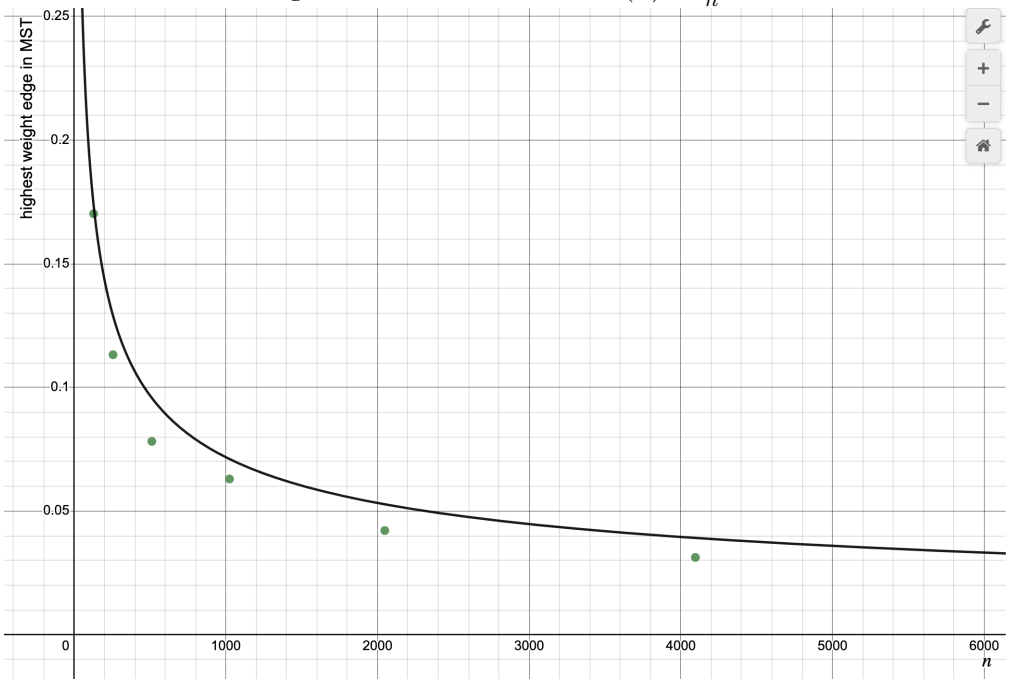


Figure 3: **3 Dimensions:** $k(n) = \frac{1.23}{n^{0.27}}$

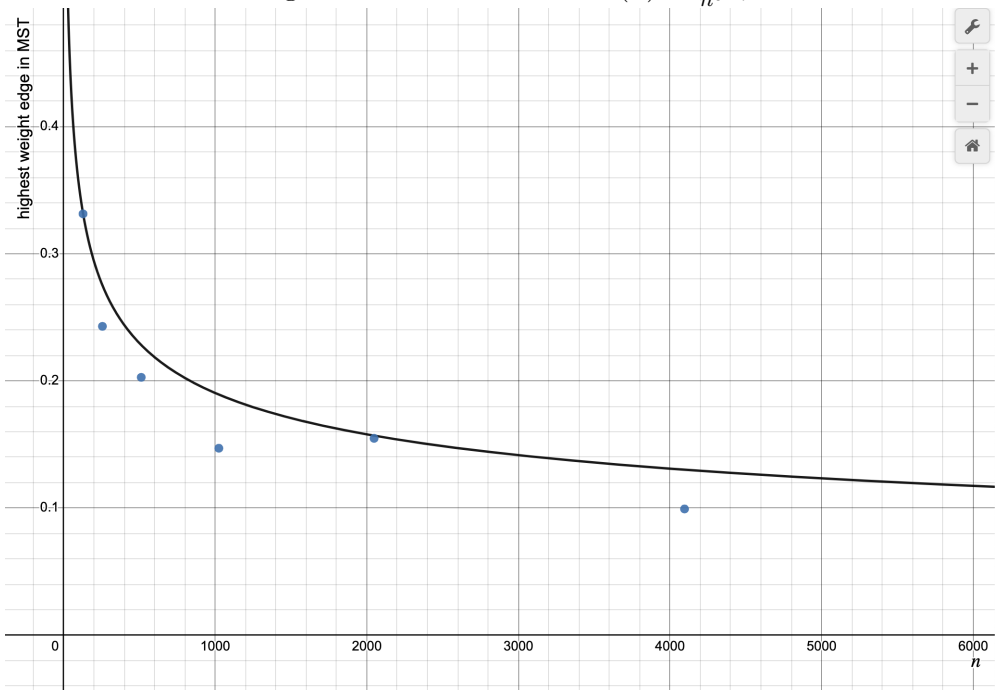
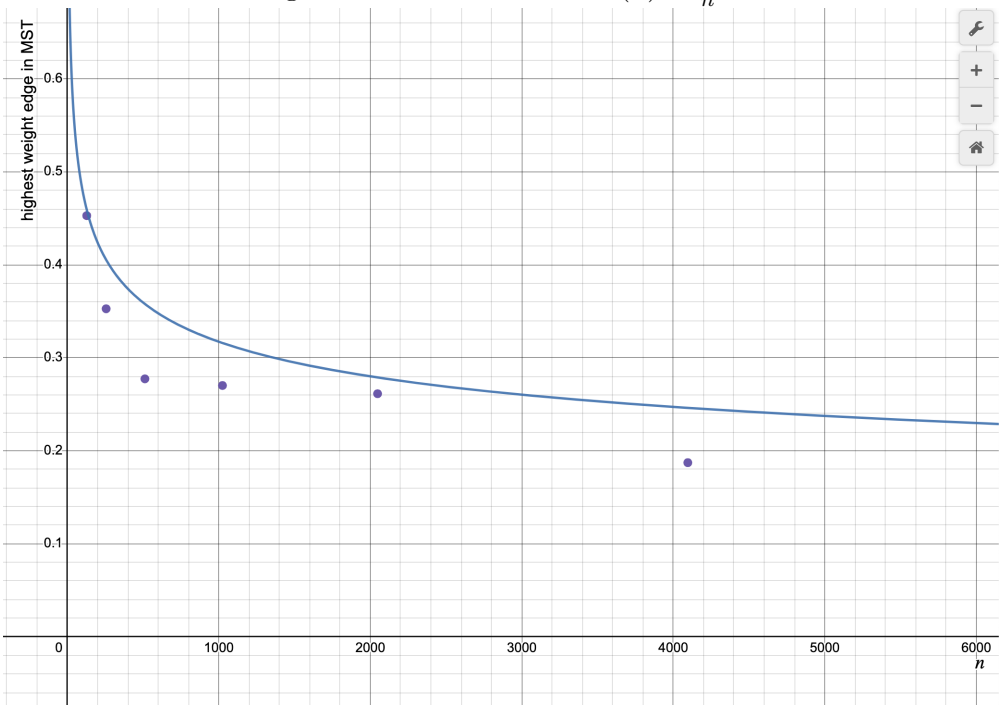


Figure 4: **4 Dimensions:** $k(n) = \frac{1.1}{n^{0.18}}$

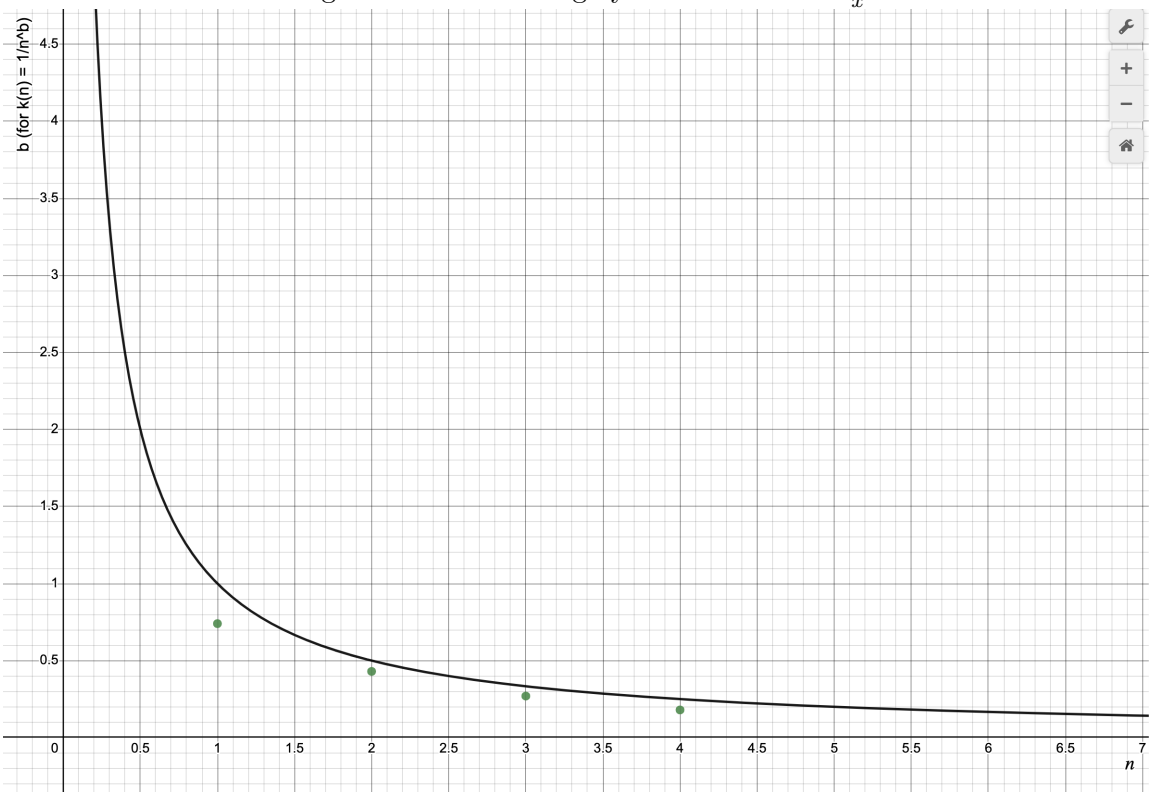


In analyzing these graphs and how they change between dimensions, we can see a pattern. Let all our cutoff functions be of the form $k(n) = \frac{1}{n^b}$. (We are letting the numerator be a constant value of 1 because the other numerator constants used that I used for my $k(n)$ did not affect the end behavior of the function and did not have any particular reason for being chosen other than ensuring they were slightly above all data points I gathered, for low values of n in particular). We can estimate that the below data:

d (value of dimension)	**b value (in $\frac{1}{n^b}$)**
1	0.74
2	0.43
3	0.27
4	0.18

Now, we can graph these points and notice that as $n \to \infty$, then $b \to 0$, which means that $k(n) \to 1$:

Figure 5: b vs. n: roughly models function $\frac{1}{x}$



3.2 $f(n)$ and runtime analysis:

After creating a model cutting out edges, it was time to actually run all four dimensions on the set of n values. For each value of n in each dimension, 5 MST's were found, and the average MST weight and average runtime were recorded. Here are the results:

1D:

n:	weight of MST (average over 5 trials):	runtime (seconds, average over 5 trials):
128	1.04703	0.0013928
256	1.10009	0.005951
512	1.18028	0.0213014
1024	1.19908	0.072916
2048	1.18369	0.26788
4096	1.20454	0.984102
8192	1.21036	3.77824
16384	1.19519	15.11968
32768	1.20215	59.4224
65536	1.20045	230.674
131072	1.20528	908.948
262144	1.2033	3491.36

2D:

n:	weight of MST (average over 5 trials):	runtime (seconds, average over 5 trials):
128	7.64441	0.0035914
256	10.8324	0.012621
512	14.9909	0.0361776
1024	21.0406	0.1126816
2048	29.6748	0.370924
4096	41.7837	1.25012
8192	58.8032	4.61898
16384	83.2416	17.54386
32768	117.435	67.6462
65536	166.073	259.472
131072	234.705	1016.848
262144	331.599	3926.54

3D:

n:	weight of MST (average over 5 trials):	runtime (seconds, average over 5 trials):
128	17.7775	0.0041828
256	27.8669	0.0156234
512	43.3552	0.0527388
1024	67.7156	0.1656042
2048	107.306	0.517198
4096	169.014	1.797796
8192	267.145	6.79208
16384	422.408	25.0376
32768	668.366	95.895
65536	1058.78	371.746
131072	1677.31	1417.926
262144	2658.69	5460.58

4D:

n:	weight of MST (average over 5 trials):	runtime (seconds, average over 5 trials):
128	28.1783	0.0045574
256	47.2167	0.0192334
512	78.2097	0.0658244
1024	130.394	0.200512
2048	217.28	0.669566
4096	362.801	2.44168
8192	603.34	9.01892
16384	1009.83	33.2486
32768	1689.6	125.7698
65536	2826.82	486.088
131072	4741.74	1826.966
262144	7951.79	6984.86

Let's now graph $f(n)$ and runtime both against n for each dimension:

Figure 6: **1 Dimension:** $f(n)$ (in red) and runtime (in blue)

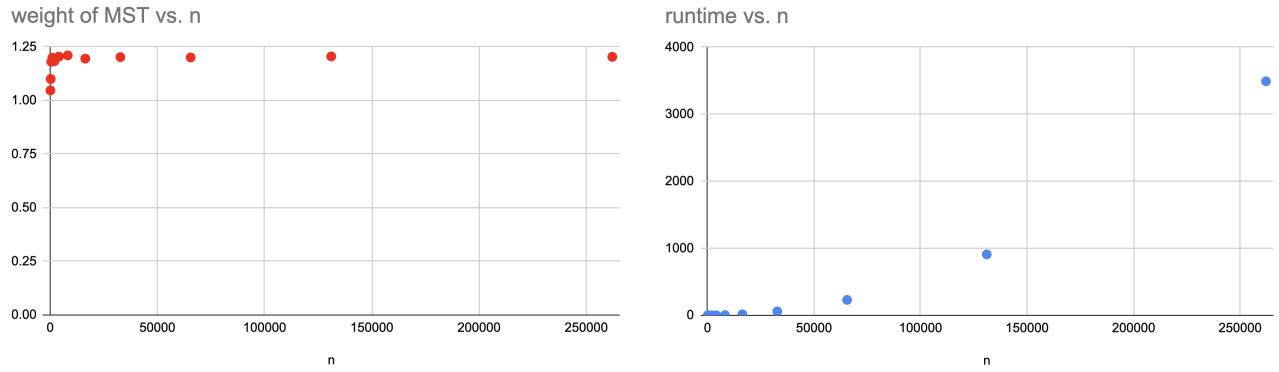


Figure 7: **2 Dimensions:** $f(n)$ (in red) and runtime (in blue)

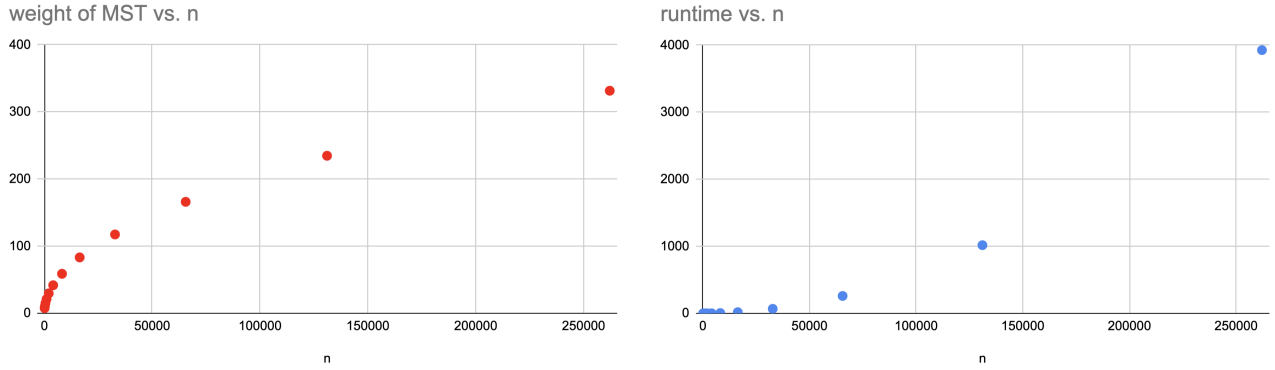


Figure 8: **3 Dimensions:** $f(n)$ (in red) and runtime (in blue)

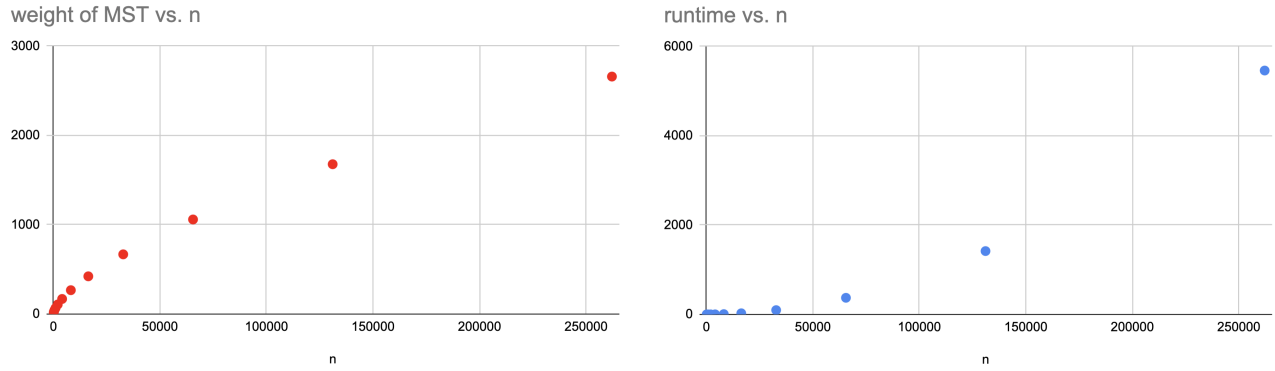
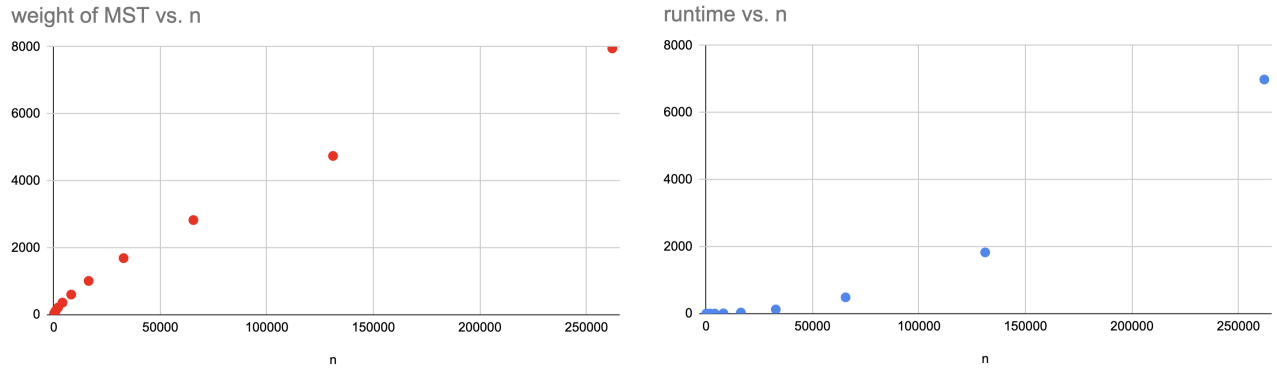


Figure 9: **4 Dimensions:** $f(n)$ (in red) and runtime (in blue)



We can see in the above graphs that $f(n)$ stays asymptotically constant ≈ 1.2 in 1 Dimension, but can be modeled with $f(n) = n^{\frac{1}{a}}$ in 2, 3, and 4 dimensions, where $a > 1$.

2 Dimensions: $a \approx 2.16$
3 Dimensions: $a \approx 1.58$
4 Dimensions: $a \approx 1.39$

For runtime, let's let the $r(n)$ be the function's runtime. We can see in the above graphs that, for all four dimensions tested, the runtime of our algorithm can be modeled with $r(n) = jn^2$, where $0 < j < 1$.

1 Dimension: $j \approx 5 \cdot 10^{-8}$
2 Dimensions: $j \approx 5.7 \cdot 10^{-8}$
3 Dimensions: $j \approx 8 \cdot 10^{-8}$
4 Dimensions: $j \approx 1 \cdot 10^{-7}$

In general, we can say that our weight of the MST $f(n)$ increases such that $f(n) = n^{\frac{1}{a}}$ where $a > 1$, and our runtime increases quadratically (with a tiny constant out front).

3.3 Discussion of randomness:

In this section of our analysis, we will discuss why we chose to use the Mersenne Twister pseudo-random generator over the standard, built-in C++ *rand()* library.

The built-in C++ *rand()* library is far from complex. And, as can be easily found on Wikipedia, or as Stephan T. Lavavej details in his Microsoft lecture from 2013 enumerating the problems with *rand()*, this function does not produce random numbers well. Under the hood, it only generates numbers in the range [0, 32767], which is incredibly small. On top of this, *rand()* uses what's known

as a Linear Congruential Generator, which is essentially a multiplication operation and a bit shift. The operations done to generate these numbers is very bare-bones, and if you were to plot 3D and 4D points using this function, as we are exactly doing in our algorithm, the points tend to line up on planes and hyperplanes. This is bad, because we need the values to be as random as possible. Also, if you run *rand()* for a long enough period of time, it will start repeating long sequences of numbers. These are only a few of the non-random qualities of *rand*() that make it an unwise deicison to use, especially for large problems such as finding the MST of a graph with 262144 nodes.

This is why we have elected to use the Mersenne Twister pseudo-random generator instead. The function *mt19937()* makes use of an engine that comes from a famous paper. It allows us to make use of (among other distributions) a uniform real distribution along with a *random_device*. We can pass our *uniform_real_distribution* through the random device and we get random numbers far more random than *rand()*. For our case, we can give the distribution the bounds (0.0, 1.0) to generate doubles in this range. The Mersenne Twister library is also incredibly fast, which is good since it won't negatively affect our runtime.

3.4 Discussion of Kruskal's vs. Prim's:

Now that we have a more clear understanding of our implementation of $k(n)$, our "cutoff function", I can discuss in more detail why I chose to use Kruskal's algorithm over Prim's. When making use of $k(n)$ to limit the number of edges, Kruskal's algorithm is simply more time efficient. This is because, in general, Prim's algorithm is faster when there are a large amount of edges and Kruskal's algorithm is faster when there are a small amount of edges. By removing a significant percentage of edges using our $k(n)$ strategy (the vast majority of edges!) the total number of edges becomes small again, making Kruskal's (with $k(n)$) ultimately the faster algorithm.