Programming Assignment 1

Name(s): Aneesh Muppidi and Eric Li

Aneesh

No. of late days used on previous psets: 2

No. of late days used after including this pset: 4

Eric

No. of late days used on previous psets: 4

No. of late days used after including this pset: 6

All of the following results ran with the listed values in the chart (vertices and dimensions) and with the number of trials set at 5. The output average total MST weight and average runtime (in seconds) are listed below as well.

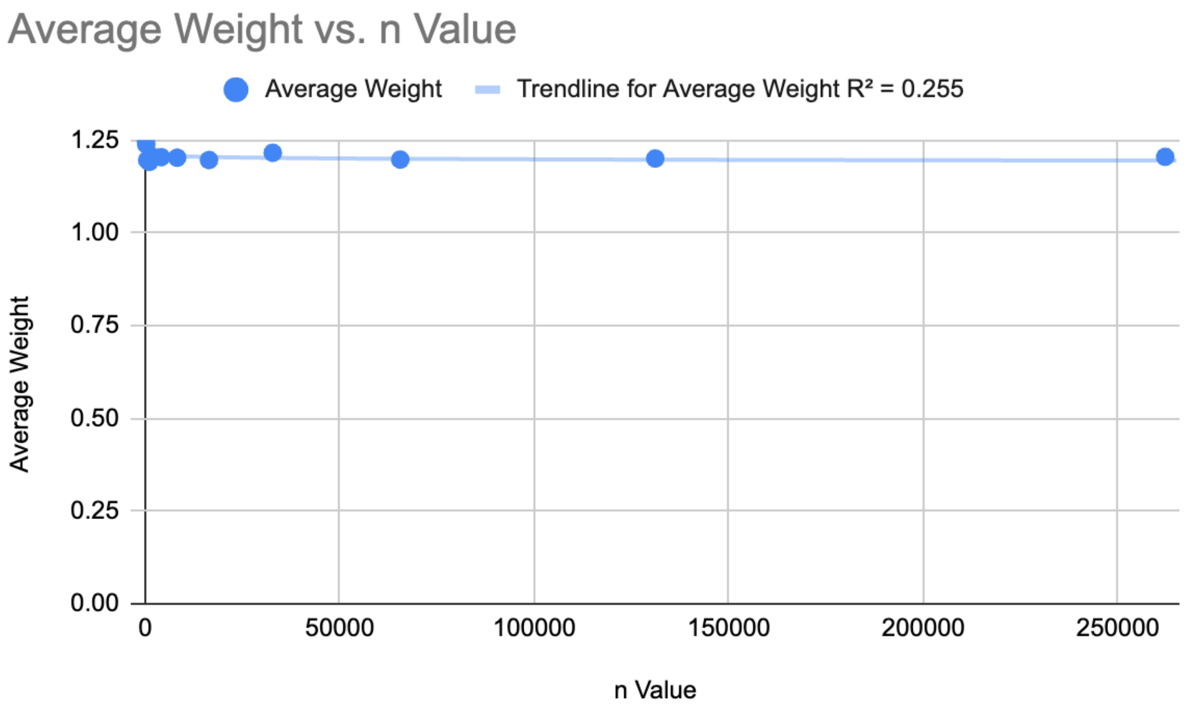
Algorithm

We implemented Prim's Algorithm using a binary heap with certain optimizations which we will go further into depth later on.

Dimension 0

Dim = 0	NumTrials = 5		
n Value	**k(n) - value**	**Average MST weight**	**Average Time (Seconds) of Each Trial**
128	0.08	1.24902424916432000	0.00247144699096679
256	0.04	1.23845899627423000	0.00696897506713867
512	0.02	1.19643041848469000	0.02110939025878900
1024	0.008	1.19168307833124000	0.06347961425781250

2048	0.007	1.20472268097921000	0.26228098869323700
4096	0.004	1.20447817248810000	0.96682200431823700
8192	0.002	1.20314717993438000	3.55013499259948000
16384	0.001	1.19680883840479000	14.60701298710000000
32768	0.0008	1.21638396365123000	54.09196591380000000
65536	0.0005	1.19791702369032000	211.78990316390000000
131072	0.0003	1.20092835606369000	842.50426602360000000
262144	0.00008	1.20541996311716000	5460.08874797821000000



We see that for dimension 0, there is no "growth" in our average weight as our n value increases. We will discuss this point. More precisely, we know that when a complete undirected graph is given with n vertices, and the weight of the edges (which are generated randomly on a range of

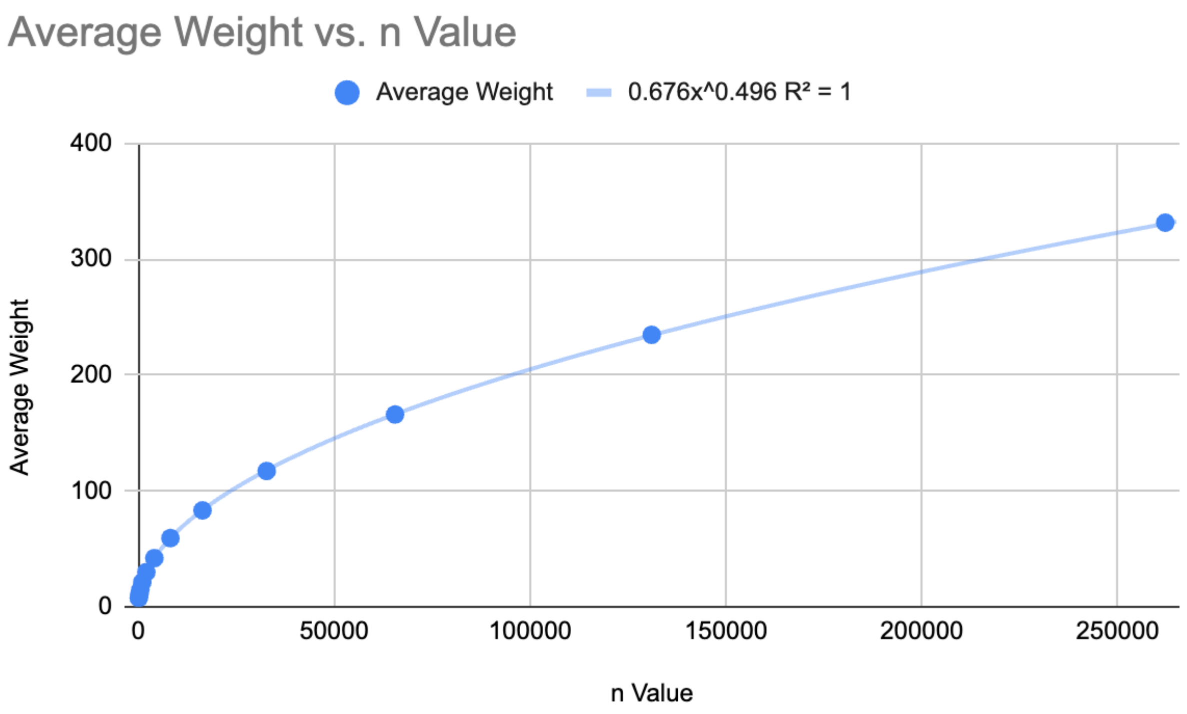
$[0,\ 1]$ have a "continuous distribution function whose derivative at zero is $D > 0$"[1], the expected weight doesn't actually grow as a function of n (as in $f(n)$). What it is instead is that it is bounded by a constant which, as $\lim_{n \to \infty}$, goes to $\zeta(3)/D$ (in this, ζ is the Riemann Zeta Function). For this constant, $\zeta(3)$ is known as the Apéry's constant which has a value of 1.20205690315959428540[2]. We can see that our Average weight vs. n Value is around 1.22 as given by the equation through the trendline equation listed at the top of our graph. This adheres pretty closely to the Apéry's constant for randomly generated edge weights where instead of the total weight being related to the number of points, it is bound to this unique constant.

Dimension 2

Dim = 2	NumTrials = 5		
n Value	k(n) - value	Average MST weight	Average Time (Seconds) of Each Trial
128	0.2	7.57177670019158000	0.00712180137634277
256	0.14	10.56220261798100000	0.01951003074645990
512	0.12	14.70341779841300000	0.03878402709960930
1024	0.1	21.27992095721470000	0.13075208663940400
2048	0.08	29.93613136543220000	0.46348524093627900
4096	0.06	41.97763214975360000	1.57901287078857000
8192	0.04	59.29648885023700000	5.23794317245483000

[1] "3.2.1 Cumulative Distribution Function." *Introduction to Probability, Statistics, and Random Processes*, https://www.probabilitycourse.com/chapter3/3_2_1_cdf.php. Accessed 24 February 2023.
[2] "Apéry's Constant -- from Wolfram MathWorld." *Wolfram MathWorld*, https://mathworld.wolfram.com/AperysConstant.html. Accessed 24 February 2023.

16384	0.0205	83.22655493720120000	16.31844782830000000
32768	0.015	117.18566761641800000	67.81180524830000000
65536	0.009	165.93266593174300000	241.89503693580000000
131072	0.008	234.67713184420800000	966.24508094790000000
262144	0.006	331.49554404458800000	6933.96871089935000000



For the second dimension, we see that the trendline expression for the data in our graph is $0.676n^{0.496}$. We generated this expression by using a power series regression (provided to us in Google Sheets), with a perfect fit of $R^2 = 1$. To generalize into a function, we round n's exponent to the nearest tenth so we can simplify our equation as such: $0.676n^{0.5} = 0.676\sqrt{n}$. This is done by using the fractional exponent to radical form rule.

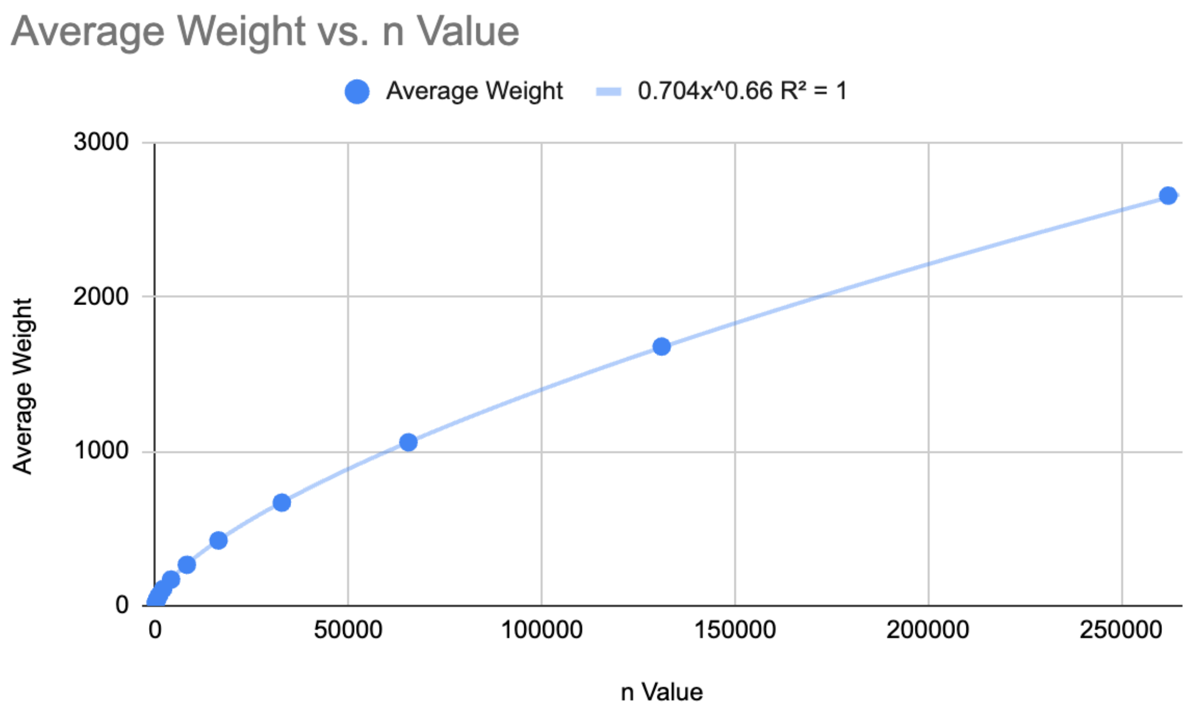
So, we can see that a general function that encapsulates the relationship between the n-value and average weight in dimension 2 is given by $f(n) = 0.676\sqrt{n}$.

As a discussion point, we see that as n increases, the average MST weight also increases. Our intuitive explanation for this is that in 2-dimensions, the vertices are more spread out; thus, since we have more vertices (which are spread out), we may have more paths, and thus, more weights we need to add to our MST. Thus our total MST weight will be greater as there are more vertices in the graph.

Dimension 3

Dim = 3	**NumTrials = 5**		
n Value	**Pruning Value (k(n))**	**Average MST weight**	**Average Time (Seconds) of Each Trial**
128	0.36	17.37105373534730000	0.00956487655639648
256	0.28	26.67183366103640000	0.02533507347106930
512	0.22	43.82610657495020000	0.06709408760070800
1024	0.18	68.83332726467690000	0.20295810699462800
2048	0.15	108.28138185451600000	0.69963288307189900
4096	0.11	169.72777835148900000	2.26843786239624000
8192	0.08	265.22909040812600000	8.32475495338440000
16384	0.06	422.85362653552500000	18.95484399800000000
32768	0.05	667.73744404025300000	74.82380628590000000

65536	0.045	1058.20146337635000000	302.19348192210000000
131072	0.035	1677.77766711805000000	1184.22231101989000000
262144	0.02	2654.78784273804000000	7880.88369894027000000



For the third dimension, we see that the trendline expression for the data in our graph is $0.704n^{0.66}$. We generated this expression by using a power series regression (provided to us in Google Sheets), with a perfect fit of $R^2 = 1$. To generalize into a function, we rewrite n and it's exponent as a root so we can simplify our equation as such: $0.704n^{0.66} = 0.704\sqrt[3]{n^2}$. This is done by using the fractional exponent to radical form rule.

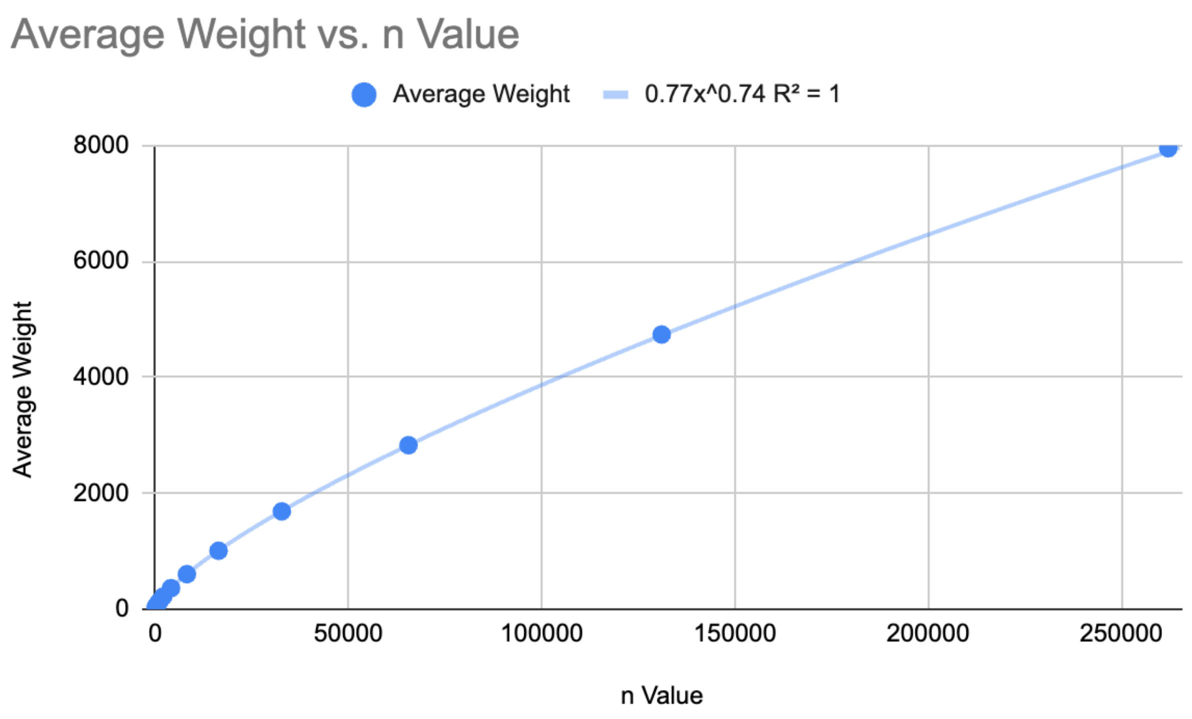
So, we can see that a general function that encapsulates the relationship between the n-value and average weight in dimension 3 is given by $f(n) = 0.704\sqrt[3]{n^2}$.

As a discussion point, we see that as n increases, the average MST weight also increases. Our intuitive explanation for this is that in 3-dimensions, the vertices are more spread out, specifically in comparison to dimension 1 and dimension 2 (because 3-dimensions adds a new plane for our vertices to exist on); thus, since we have more vertices (which are more spread out), we may have more paths, and thus, more weights we need to add to our MST. Thus our total MST weight will be greater as there are more 3d vertices in the graph.

Dimension 4

Dim = 4	**NumTrials = 5**		
n Value	**k(n) - value**	**Average MST weight**	**Average Time (Seconds) of Each Trial**
128	0.46	27.76195124955890000	0.00919508934020996
256	0.4	47.47071873184050000	0.02836298942565910
512	0.3	77.99629385561960000	0.06203508377075190
1024	0.3	130.09423530082500000	0.25494003295898400
2048	0.2	216.45271082430800000	0.69203209877014100
4096	0.2	362.68792839709700000	2.84127807617187000
8192	0.19	603.96661692306200000	11.02535200119010000
16384	0.14	1007.62871682792000000	26.45822572708130000
32768	0.13	1687.76407196386000000	106.89353203770000000
65536	0.13	2828.52997324676000000	426.63466095920000000
131072	0.099	4738.35949871510000000	1531.54516291620000000

262144	0.08	7950.44875027352000000	9405.81409120559000000



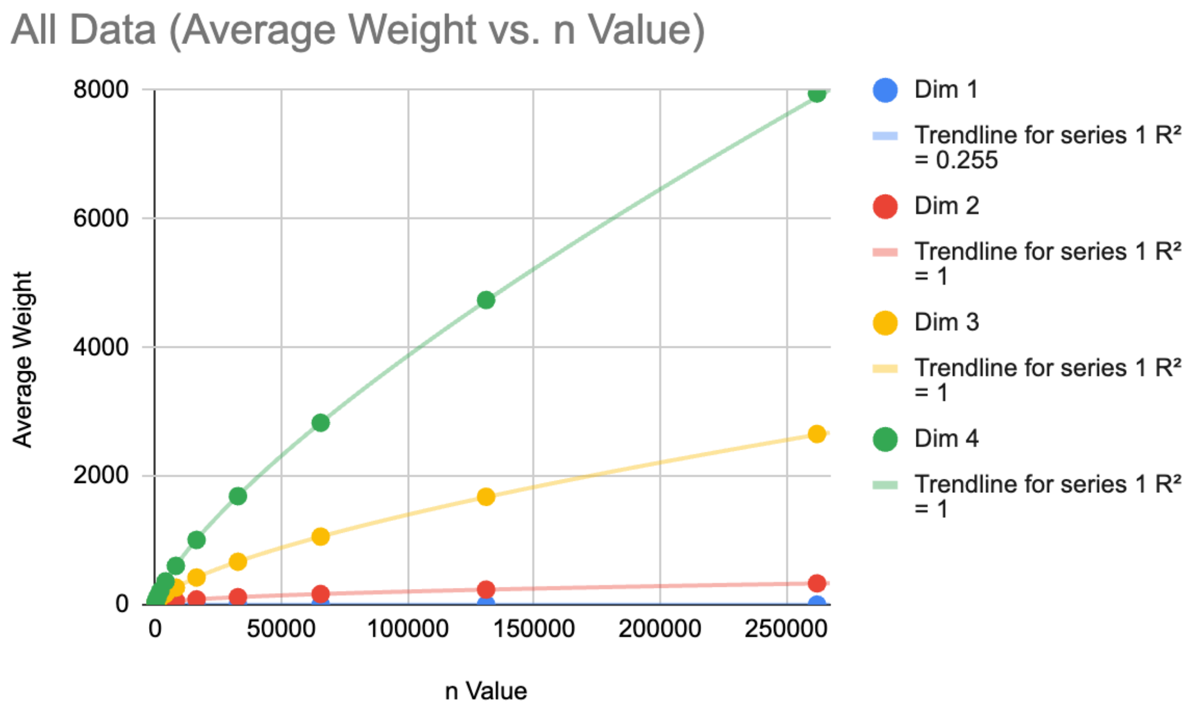
For the fourth dimension, we see that the trendline expression for the data in our graph is $0.77n^{0.74}$. We generated this expression by using a power series regression (provided to us in Google Sheets), with a perfect fit of $R^2 = 1$. To generalize into a function, we will round n's exponent to the nearest five-hundredth (difference of 0.01): $0.77n^{0.75} = 0.77\sqrt[4]{n^3}$. This is done by using the fractional exponent to radical form rule.

So, we can see that a general function that encapsulates the relationship between the n-value and average weight in dimension 3 is given by $f(n) = 0.77\sqrt[4]{n^3}$.

As a discussion point, we see that as n increases, the average MST total weight also increases. Our intuitive explanation for this is that in 4-dimensions, the vertices are more spread out, specifically in comparison to dimension 1, dimension 2, dimension 3, (because 4-dimensions adds a new plane for our vertices to exist on); thus, since we have more vertices (which are more spread out), we may have more paths, and thus, more weights we need to add to our MST. Thus our total MST weight will be greater as there are more 4d vertices in the graph.

<u>All Data</u>

Total	**(Average MST Weight)**			
N Value	**Dim 1**	**Dim 2**	**Dim 3**	**Dim 4**
128	1.249024249	7.5717767	17.37105374	27.7619512495589
256	1.238458996	10.56220262	26.67183366	47.4707187318405
512	1.196430418	14.7034178	43.82610657	77.9962938556196
1024	1.191683078	21.27992096	68.83332726	130.0942353008250
2048	1.204722681	29.93613137	108.2813819	216.4527108243080
4096	1.204478172	41.97763215	169.7277784	362.6879283970970
8192	1.20314718	59.29648885	265.2290904	603.9666169230620
16384	1.196808838	83.22655494	422.8536265	1007.6287168279200
32768	1.216383964	117.1856676	667.737444	1687.7640719638600
65536	1.197917024	165.9326659	1058.201463	2828.5299732467600
131072	1.200928356	234.6771318	1677.777667	4738.3594987151000
262144	1.205419963	331.495544	2654.787843	7950.4487502735200



Taking our dimensions (excluding 0 since it is a special case and follows the explanation listed under the dimension 0 section) into account and their derived functions, we can formulate a general formula for average weight vs. n-value that accounts for all dimensions. Listed below are all of our functions:

$$f(n) = 0.676\sqrt{n}$$

$$f(n) = 0.704\sqrt[3]{n^2}$$

$$f(n) = 0.770\sqrt[4]{n^3}$$

It is clear that these three functions share similarities. They all have the general structuring of having a constant value c multiplied by n raised to a value all under some root. From our tested dimensions, the c value appears to be around (average of provided c values) 0.716, however, we see that it grows as the dimension increase so we can assume that the constant c grows with some proportion to dimension. More specifically, our guess for the relationship in the root is the dim-root of n raised to $(dim - 1)$:

$$f(n) = c\sqrt[d]{n^{(d-1)}}$$

This was not what we anticipated since initially our hypothesis was that the average weight would grow exponentially, however, this derived function makes more sense after analyzing the graphs which fell under the category of a root graph. We can see that as the dimension grows for a set n value, the weight will also grow and as we increase dimension, the distance and the probabilities of larger distances between two vertices will increase. This makes sense with our data since as we increase dimensions for any of the given values of n, we see an increase in average weight (weight).

Algorithm

As mentioned earlier, we decided to implement Prim's using a binary heap. One of the reasons we both chose to implement Prim's algorithm is because we were both more comfortable with the concepts of a heap (from CS120) than the union-find structure (for Kruskal's). Moreover, after doing some research online, we found that Prim's algorithm is supposedly faster for denser graphs than Kruskals – and since we are working with complete graphs, our edges will be maximal for our graphs, so it only makes sense to implement Prim's over Kruskal's for this specific problem.

We initially implemented a list as our priority queue structure for Prim's. That is, we stored our edges in a list and then traversed that list to find the minimum edge. We found our algorithm was running too slow. Then, we researched what a fibonacci heap was. One drawback we found about the fibonacci is that it uses a lot more memory than other heaps. This was not ideal, especially because we implemented our algorithm in Python, meaning we can not easily make memory optimizations. Still, we tried implementing the heap. But, after many trials, we could not implement the fibonacci heap (mainly due to the conceptual complexity of the algorithm). We then decided to implement a standard binary heap that was slightly modified to heapify our Vertex object structure. We looked into this lecture among other online resources for a refresher on how to think about implementing binary heaps. If we had more time, we would explore a slightly faster structure, d-ary heaps. One consideration about using that structure is that we would need to figure out an optimal d-value. But again, we did not have much time to explore that option.

Algorithm Asymptotic Runtime

To generate a graph (for dim = 2,3,4), we need to generate a random coordinate for each vertex. We loop V times to generate a random coordinate, where V = n = the number of vertices. This takes O(V) time.

We know from Lecture 6 Notes that Prim's algorithm has almost exactly the same runtime and pseudocode as Dijkstra's algorithm. We can, thus, use lecture 5 notes, to see that Prim's would also run in $O(|V| \times deletemin\ +\ |E| \times insert)$. We know that with our binary heap implementation that both deletemin and insert operations run in $O(log|V|)$. We further know that the runtime for a binary heap, |V|×deletemin+|E|×insert takes $O(|E|log|V|)$.

Thus, Prim's alone with a binary heap implementation takes $O(|E|log|V|)$.

Program Runtime (not asymptotic) and Effect of Cache Size

To see the actual runtime (second) of our algorithm, refer to the charts above for each dimension and associated n value. We can see that as our n value grows, so does our runtime (seconds). This is intuitive since as there are more vertices, we need to traverse more "adjacent" vertices, we need to store more vertices in our heap, we need to check if certain conditions meet for more vertices, and we need to calculate the euclidean distance for more vertices. We also see that as dimension increases, the runtime for a set value n also increases, especially for the large n values. This can be attributed to increased small time consuming operations as dimension increases (such as generating and assigning random weights, calculating distances between two vertices with more coordinates as dimension increases, heapifying vertices, etc) per iteration, however, we are iterating tens and hundreds of thousands of times for the large n values, which sums up to a significant runtime.

In terms of effects of cache size, when we ran our initial program using a list as a priority queue, we ran into the problem of running out of memory which did not allow us to compute n values larger than a certain threshold. One of our strategies to deal with this problem was to only run one program at once as well as quit out of any other applications (so that our memory was only being used by our program). After implementing a binary heap and implementing our mentioned optimizations, our cache size significantly decreased!

Pruning

In order to optimize our algorithm, we implemented a pruning technique in order to cut down on the number of edgers we add to our priority queue as well as the number of edges we visit. In order to do this, we ran our algorithm on various n values and displayed all of the edge weights that were selected to be in the MST. We then searched for the heaviest weighted edge in each trial for each given n and took an average of the heaviest weighted edge (done for each dimension). We then set our $k(n)$ value to be the value (for the specific dimension) used for the generation of our perimeter in our dimensional k-bound filter optimization where all edge weight values outside of this perimeter are disregarded. (More on this in the optimization section).

Random Number Generator

We did not have any interesting experiences regarding the random number generator. For all of our trials, the randomness of the edge weights fit within the bounds that we set them to and they seemed to be, indeed, random. We used the Python $random$ library, which has been around, worked on, and updated, for many years, so we do not see a reason as to why we shouldn't trust the effectiveness of it, however, it is difficult to prove that something is truly and holistically random. Moreover, as we tested multiple experiments more than once, we saw that our total average MST costs for the same n and same dimension would vary slightly differently from experiment to experiment, and even trial to trial within the same experiment. This ensured us that the edges and vertices were being randomly generated.

Optimizations

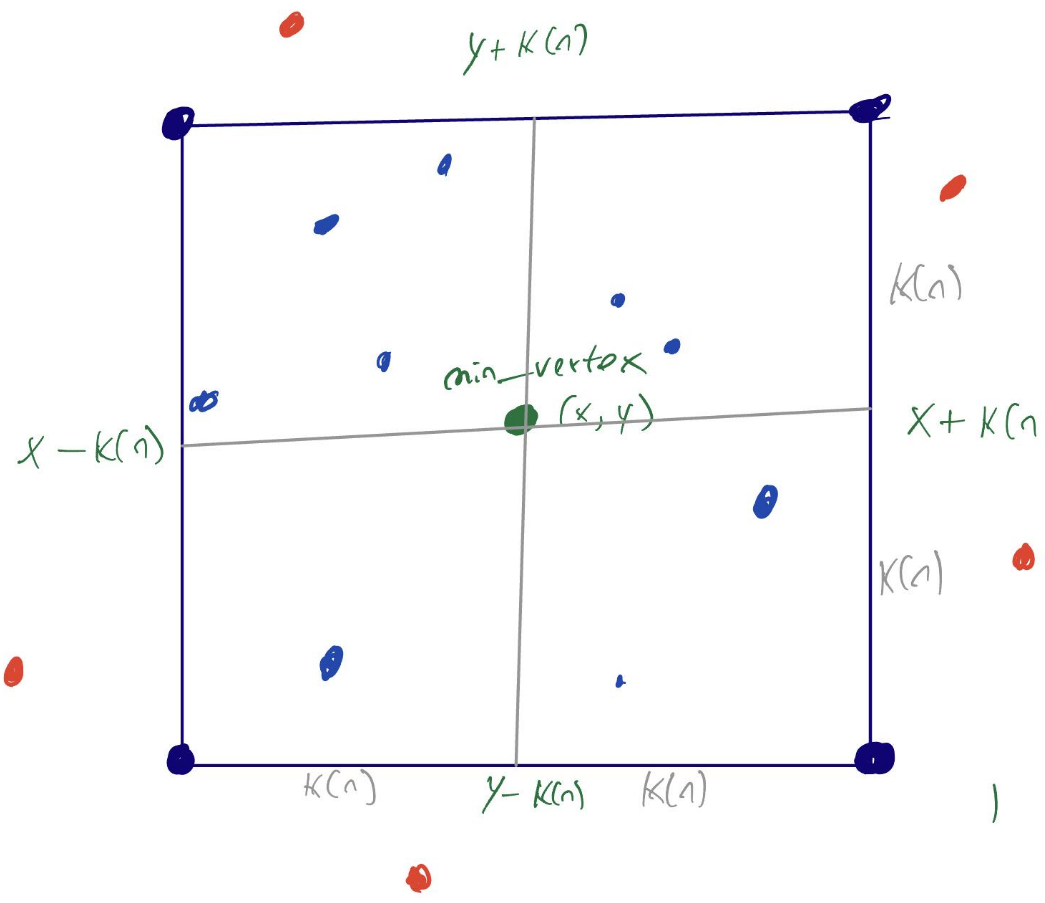
At first, we calculated the euclidean distance between every edge for the graph before running Prims. Our first optimization was to generate edge weights as we traversed the neighbors of our extractedMin node from our priority queue, but to only generate those edge weights if we hadn't added the vertex to our MST.

Our second point of optimization was how we were storing the vertices we added to our MST. We initially had them stored in a dictionary. Initializing this dictionary required looping over every vertex and then setting dict[vertex] = false. However, look up (to see if a vertex is in our MST or not) was still O(1). We optimized this by storing them in a set. With a set, we didn't initially have to initialize it with all of the vertices. We simply added a vertex to our set if it belonged to our MST, which is O(1). What is nice about storing it in the set is that look up for the vertex is also O(1).

Our third optimization was only adding an edge to our priority queue if the edge weight was less than or equal to our $k(n)$ value. Thus, if an edge weight was above our $k(n)$ value, we would **"throw it out."**

Our fourth optimization asked the question, *can we **throw out** edges without calculating the edge weight using the euclidean distance formula?* This would save us time since our euclidean distance function requires squaring and square roots. To do this, we introduced a k-bound filter. Every time we extracted the min edge/vertex (let us denote this vertex as min_vertex) from our priority queue and added it to our MST, we would then traverse its neighbors to see if we should add its neighboring edges to our queue. When we traverse the neighbors, before we calculate the edge weights and see if they are below or above our $k(n)$ value, we introduce a quick filter. The filter constructs boundaries for each plane in our dimension based on the coordinates of our min_vertex. This is done by adding $k(n)$ to each coordinate of our min_vertex to create the upper boundary and subtracting $k(n)$ from each coordinate of our min_vertex to create the lower boundary . We then do a quick set of if statements to check if the coordinates of the neighbor vertices are above the upper boundaries or below the lower boundaries, if they are, we throw them out. If they are not, then they may be within $k(n)$-distance of our min_vertex. We then calculate the weight of these edges and then add them to our priority queue if they are, for sure, within our $k(n)$-values. The intuition for this optimization is that we only want vertices that are within $k(n)$-distance of our min_vertex. Thus, we just need to check if any of the neighbor vertex's coordinates are outside of $k(n)$-distance + each coordinate of min_vertex, because that would mean the vertex has a distance from min_vertex that is greater than $k(n)$-distance.

To provide clarity, we will show an example in dimension 2.



In this example, the blue vertices may be within $k(n)$-distance of min_vertex, but we certainly know that the red vertices are outside of min_vertex because their respective x coordinate is greater than x + $k(n)$-distance or less than x - $k(n)$-distance or because their y coordinate is greater than y + $k(n)$-distance or less than y - $k(n)$-distance. Thus, in this example, it only takes 4 if statements to check whether a vertex is within $k(n)$-distance of min_vertex, and we can throw away that vertex without ever having to calculate its actual euclidean distance from min_vertex.