CS 124 Programming Assignment 1

Qiang Fei, Jordan Turley

February 29, 2020

1 Introduction

In this assignment, we are tasked with generating complete graphs on n vertices and finding the average weight of a minimum spanning tree. A complete graph on n vertices has $\binom{n}{2}$ edges, and we generate random edge weights by one of two processes. First, we can simply generate the edge weight as Uniform(0, 1). Second, for each vertex, we generate vertices as random points in a unit square, cube, or hyper-cube, and calculate the edge weight as the Euclidean distance between the vertices.

2 Algorithm

We implemented both Prim's and Kruskal's algorithms in C++. We can compare these two using order notation. Prim's algorithm, using a binary heap, is $O(|E| \log |V|)$, and Kruskal's algorithm is $O(|E| \log |E|)$. Since $|E| = \binom{n}{2}$ is $O(n^2)$, Kruskal's algorithm will theoretically take twice as much time as Prim's, since $\log(n^2) = 2 \log n$, even though technically the algorithms are the same in terms of order notation, since Kruskal's algorithm is $O(|E| \log |E|) = O(|E| \log |V|^2) = O(2|E| \log |V|) = O(|E| \log |V|)$.

We also wanted to look at a real world comparison of the two. We compared the runtimes, in seconds, for several values of n, which is shown in Figure 1. We run the algorithms on the complete graph with $n = 4096$ vertices for ten trials. Note that this is *not* using the graph simplification methods discussed below.

n	**Prim's Algorithm**	**Kruskal's Algorithm**
128	0.004	0.005
256	0.006	0.01
512	0.014	0.027
1024	0.044	0.093
2048	0.175	0.346
4096	0.776	1.502
8192	3.209	6.367
16384	13.69	26.91

Figure 1: Timing comparison between Prim and Kruskal, in seconds

We see that our earlier calculation is almost exactly true in practice. Kruskal's algorithm takes almost exactly twice as much time as Prim's algorithm, so we will use Prim's algorithm in the rest of this assignment. Prim's algorithm also used less memory in our implementation, as for Kruskal's algorithm, we had to keep a separate list of edges to sort.

2.1 Graph Representation

We represented our graph using adjacency lists. This allowed us to only use as much memory as we had the number of edges, as an adjacency matrix would still use memory even if there wasn't an edge. Since

we are dealing with complete graphs where there are edges between all vertices, this should not make much difference, but as we increase the number of vertices and start discarding edges, we can save a lot of memory by using adjacency lists instead of an adjacency matrix. If we decide to throw away an edge in the adjacency matrix, as discussed below, we would still use memory by storing something like a -1 or a null value, but in adjacency lists, the edge simply does not exist, so there is no memory used by edges not in the graph.

The above allowed us to save lots of memory even when using a very large number of vertices. When using $n = 2^{18}$, we needed less than 500MB of memory to represent the graph and find the MST. In the interest of time, we did not try any larger values of n, but I am confident that n would have to become much larger before it was a problem regards to memory, as long as the bounding functions we defined above continued to hold.

2.2 Graph Simplification

With this algorithm, we would like to be able to find the minimum spanning tree of a complete graph with a large number of vertices, i.e. up to 2^{18} in this case, but in the real would we might care about millions or billions of vertices. There are $\binom{n}{2}$ edges in a complete graph, so a graph with 2^{18} vertices would contain about 34 billion edges. If, for example, each edge used 4 bytes of memory, the enitre graph would take over 100 GB of memory to store, which is not feasible.

To remedy this, we observed that the minimum spanning tree does not depend on any of the other higher-weight edges in the graph, so it wouldn't matter if we hadn't even added them to begin with. We observed how the maximum edge weight in the trees changed as we added more vertices and edges, and formulated the formulas below as a function of the number of vertices in the graph, as a maximum bound to the edge weight to include in the graph. Throwing away edges greater than this bound would not affect the minimum spanning tree, and would not consume memory. It is important to note that if this bound is too low, we will throw away too many edges and the graph will not be connected, so a minimum spanning tree will not exist. This was easy to detect in our output, as the tree's weight would be 3e+10 and the maximum edge weight 1e+10, and in theory both of these would be infinite.

Graph Type	**Edge Weight Upper Bound**
Uniform	$k(n) = 32/n$
2-Dimensional	$k(n) = 4/n^{0.5}$
3-Dimensional	$k(n) = 2.5/n^{0.33}$
4-Dimensional	$k(n) = 2.1/n^{0.25}$

Note that these are simply patterns that we observed for $n = 128, 256, \ldots 262144$ and may or may not hold for higher values of n.

2.3 Runtime

We ran the graph algorithm on a MacBook Pro with an Intel i7 processor. Before adding the bounding function to discard high-weight edges, we could only run up to $n = 8192$, both due to time and space contraints. However, after discarding high-weight edges, we were able to run even the higher numbers of n in a reasonable amount of time. Below are the runtimes to find the minimuim spanning tree for one trial given for each type of graph.

n	**Uniform**	**2D**	**3D**	**4D**
128	0.004	0.005	0.004	0.004
256	0.006	0.006	0.005	0.005
512	0.007	0.008	0.007	0.007
1024	0.015	0.013	0.013	0.013
2048	0.038	0.022	0.026	0.028
4096	0.104	0.059	0.064	0.071
8192	0.338	0.146	0.164	0.19
16384	1.279	0.439	0.527	0.62
32768	4.835	1.539	1.881	2.145
65536	19.552	5.789	6.751	7.675
131072	73.332	20.366	27.188	30.936
262144	281.436	78.514	105.231	119.095

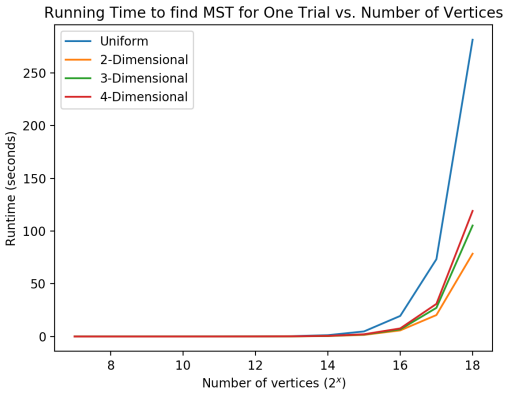


Figure 2: Comparison of runtime between methods of graph generation, in seconds

Note, since these are a single sample, there may be a lot of variability from one run to another. We kept the running environment constant, keeping the laptop plugged into power and not using other applications.

We were actually surprised at how low the runtimes were. We could run the program in about or under a second up to $n = 16384$ vertices, and even going up to the maximum of $n = 262144$ vertices only took a few minutes, so you could run this for a few hours and get several trials. We were a bit surprised that the slowest was the uniform edge weight case, as it seems like we have to do more calculations in the other cases, but we hypothesized that the random number generation was the bottleneck, which we do a lot less of on the latter three cases. In these cases, we only have to generate $2|V|$, $3|V|$, or $4|V|$ random numbers, but in the first case we have to generate $\binom{n}{2}$ random numbers. We have to calculate the Euclidean distance in the latter cases, but this should be quick and the compiler likely does some loop unrolling to further optimize this.

We didn't try running the program on any other systems (other than nice.fas.harvard.edu just to ensure it compiled on that system), so we did not observe any differences in cache size or any problems relating to this.

2.4 Random Number Generation

We did not observe any problems in generating random numbers. We used the Mersenne Twister engine (`mt19937` in C++) to generate numbers from a Uniform(0, 1) distribution. We seeded our random generator with the current time to ensure as much randomness as we could get.

3 Results

3.1 Edge Weights ∼ Uniform(0, 1)

n	**Trials**	**Average Tree Weight**
128	1000	1.20017
256	1000	1.19803
512	1000	1.19936
1024	1000	1.20319
2048	500	1.20159
4096	100	1.2021
8192	50	1.20305
16384	10	1.19711
32768	5	1.19828
65536	5	1.20216
131072	5	1.20366
262144	5	1.20179

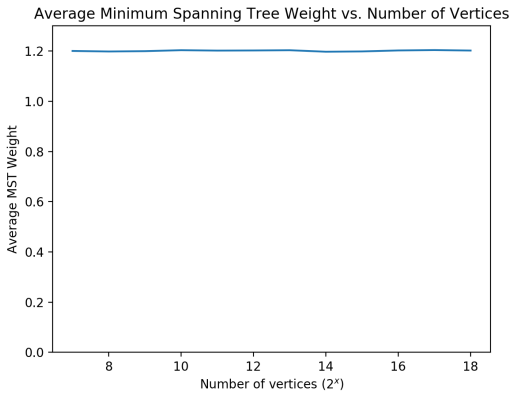


Figure 3: Average MST Weight for Uniform Edge Weights

In the uniform edge weight case, we see that changing the number of vertices does not seem to change the average tree weight. The average tree weight seems to be constant at 1.2 as we change n. In some ways this is surprising, but in others it is not. At first glance, it seems like as we add more edges the weight of the tree would have to increase. However, as we add more edges, there are more changes for a Uniform(0, 1) to be a very small value, so it seems like the increase in the number of edges and the increased number of small edge values perfectly cancel each other out to give the answer of 1.2. We could not find an exact mathematical solution or reasoning for this, but one could likely be made using the order statistics of a Uniform random variable.

3.2 2-Dimensional Points

n	**Trials**	**Average Tree Weight**
128	1000	7.61478
256	1000	10.6673
512	1000	14.9676
1024	1000	21.0504
2048	500	29.6333
4096	100	41.7634
8192	50	58.9493
16384	10	83.1285
32768	5	117.52
65536	5	166.085
131072	5	234.632
262144	5	331.666

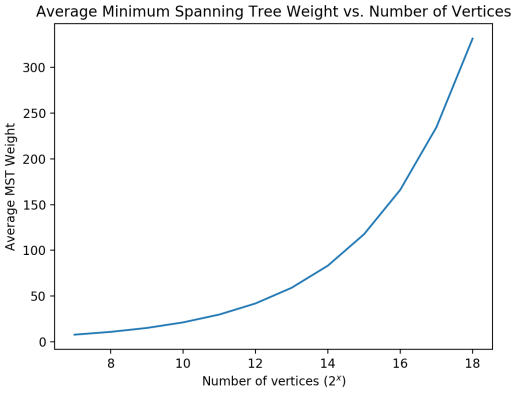


Figure 4: Average MST Weight for 2-Dimensional Graphs

In this case, we see more of what we might expect. As we increase the number of vertices and edges, the average weight of the tree goes up.

3.3 3-Dimensional Points

n	Trials	Average Tree Weight
128	1000	17.617
256	1000	27.5943
512	1000	43.2932
1024	1000	68.1003
2048	500	107.295
4096	100	169.214
8192	50	267.267
16384	10	422.103
32768	5	668.412
65536	5	1058.05
131072	5	1677.77
262144	5	2657.34

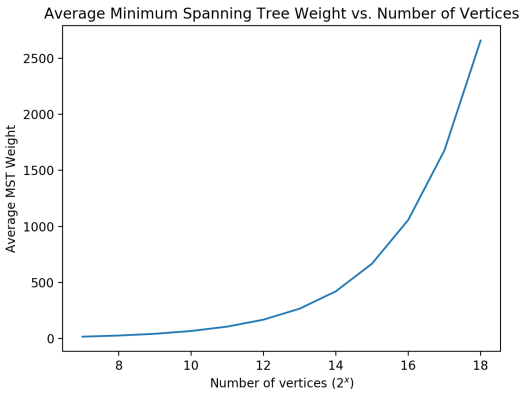


Figure 5: Average MST Weight for 3-Dimensional Graphs

Again, we see that the graph weight increases as a function of the number of vertices. We also see that as we go from 2 dimensions to 3, the average tree weight increases as well. This is likely due to the points being able to spread out more. The maximum distance between two points in a 2D square with edge length 1 is $\sqrt{2}$, but in a 3D box the maximum distance is $\sqrt{3}$.

3.4 4-Dimensional Points

n	Trials	Average Tree Weight
128	1000	28.4653
256	1000	47.1759
512	1000	78.2615
1024	1000	130.104
2048	500	216.606
4096	100	361.038
8192	50	603.39
16384	10	1008.89
32768	5	1689.69
65536	5	2828.88
131072	5	4740.22
262144	5	7951

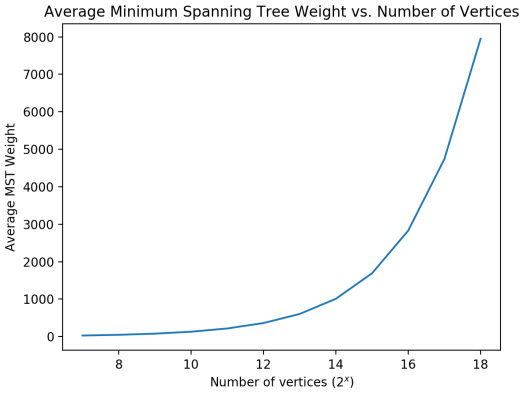


Figure 6: Average MST Weight for 4-Dimensional Graphs

Again, we see that as we go from 3D to 4D, the average tree weight increases even more. Again, the maximum distance between two points in 3D is $\sqrt{3}$, but in 4D it is $\sqrt{4} = 2$. We can see that the functional form of the three looks identical, as the graphs look almost exactly the same except for the y-axis scale. This is helpful for our analysis of the functions below.

3.5 Estimation of Functions Approximating Plots

While the results for the dimension one case seems uniform and independent of change in n, we quickly decide to estimate it by a constant function. At the same time, we find that the relationship between n and weights are nonlinear when all other dimensions, but always monotonically increasing. After trying several

combinations of basic functions, we find that the points can all be approximated by a function of form

$$f(n) = \frac{n^b}{\log(n)}$$

for some b in range $[0, 1]$. By testing different values, we decide our best estimates for functions at each dimension, as listed below:

Dimension 0 :

$$f_1(n) = 1.2$$

Dimension 2 :

$$f_2(n) = \frac{n^{0.55}}{\log(n)}$$

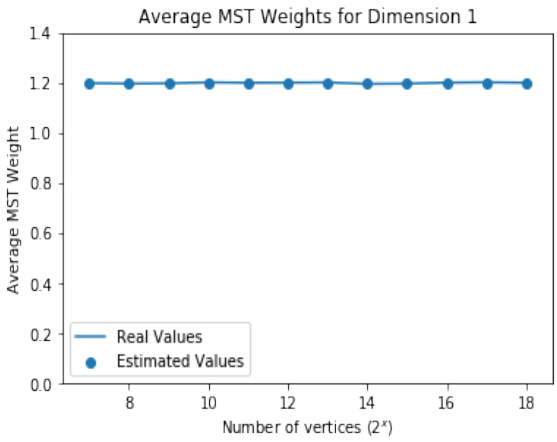
Dimension 3 :

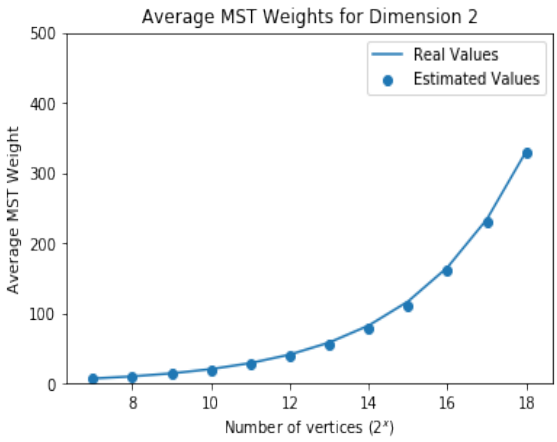
$$f_3(n) = \frac{n^{0.72}}{\log(n)}$$

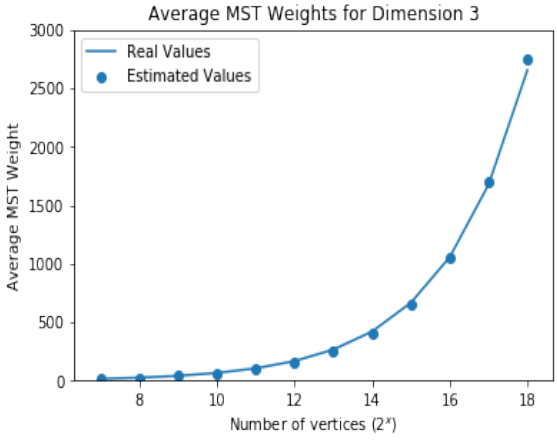
Dimension 4 :

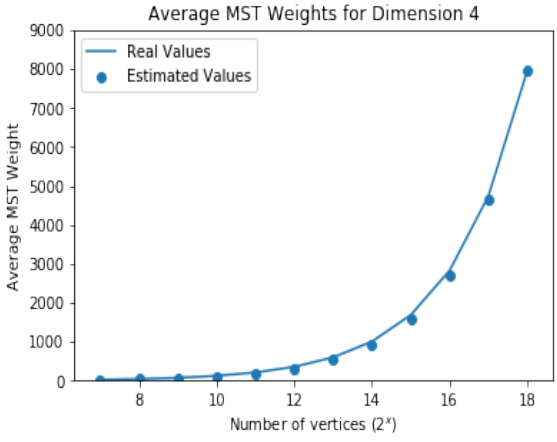
$$f_4(n) = \frac{n^{0.805}}{\log(n)}$$

The graphs showing approximations are listed below, where the lines show the values calculated by codes and the points are our estimates.









3.6 Discussion on our Estimation Functions

While the estimates of our functions well approximate the shapes of lines formed by the points, we are not sure whether they truly explain the pattern of these points so that these functions still work with even larger n. We know that these weights should be calculated as the expected value of some function of n and d for dimension of points, correlated to the formula for the uniform distribution from which we draw our values, but we are not so sure how to derive this true function. Thus we searched online and see that there are some established formula for calculating weights of this kind of minimum spanning trees.

From results established in [1], we know that the expected size of the Euclidean MST for large numbers of points for large n can be approximated by:

$$c(d)n^{\frac{d-1}{d}} \int_{R^d} f(x)^{\frac{d-1}{d}} dx$$

Where f is the density of the probability function for picking points, d represents dimension and $c(d)$ is some constant. As in our case, we pick points from uniform distribution, the integral in the function will very much yields a result of some constant value, so that the general formula can be simplified as

$$f(n) = c \times n^{\frac{d-1}{d}}$$

Based on this result, we substitute d with appropriate dimensions and approximate c (which is some constant value) by fitting the function to the points we have. (Notice that the formula at dimension 1 will yields a constant function, which should thus be the same function as what we just guessed.) After rounds of testing, we get the following functions:

Dimension 0 :

$$f_1(n) = 1.2$$

Dimension 2 :

$$f_2(n) = 0.66 \times n^{\frac{1}{2}}$$

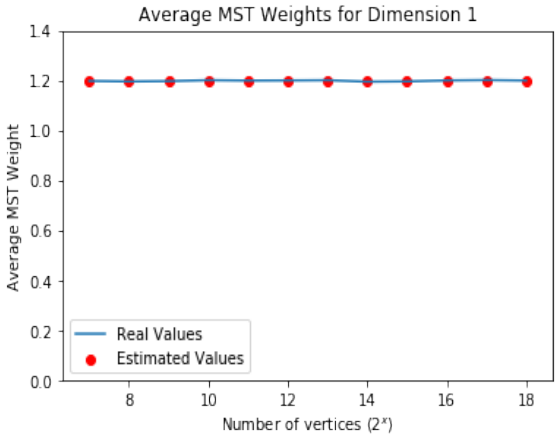
Dimension 3 :

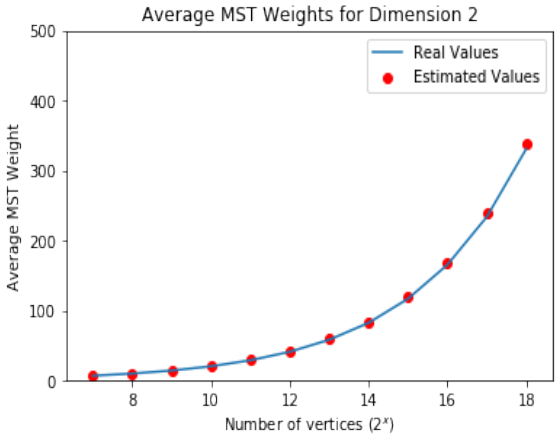
$$f_3(n) = 0.66 \times n^{\frac{2}{3}}$$

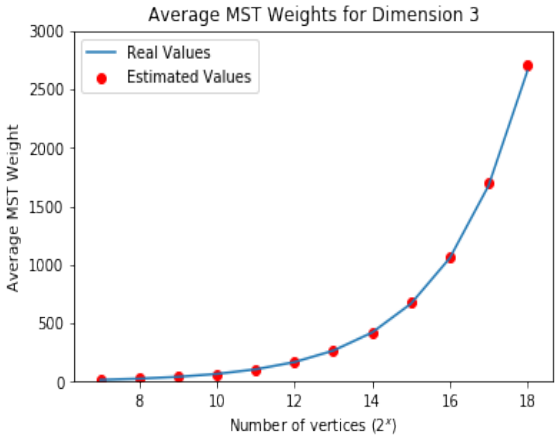
Dimension 4 :

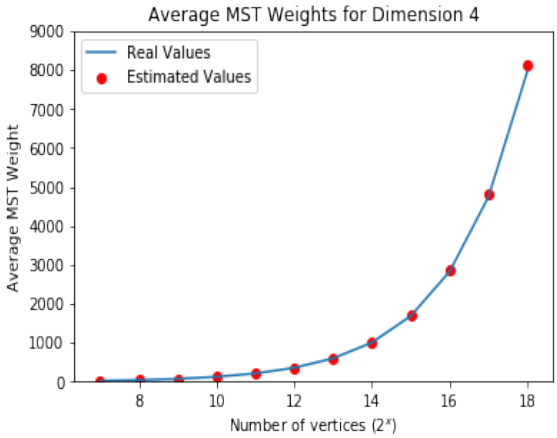
$$f_4(n) = 0.7 \times n^{\frac{3}{4}}$$

The graphs showing approximations are listed below, where the lines show the values calculated by codes and the red points are our estimates. These functions again make good approximation of the points generated from codes, but we have better confidence in the correctness of these functions.









4 Conclusion

One of the most important things to take away from this project is the ability to think about simplifying a problem when it becomes too big. A lot of people would have stopped when the graph was too big and took too much memory, but by looking at some results and figuring out how we can discard data in memory that is basically wasted, as it has no impact on the final answer. By doing this, we were able to take this program from using at least 100GB of memory to less than 1GB. With a bit more analysis, we could probably have used even less memory.

Skills like this are especially useful in the computer science industry. Imagine trying to run an algorithm on a graph of Facebook's users and friends. Facebook has well over one billion users, so to make the algorithm finish in any reasonable amount of time and consume a reasonable amount of memory, you must simplify the graph in some way. Granted, Facebook has computers and clusters of computers with enormous amounts of memory and processing power, but even then, simplifying the graph could take the runtime of an algorithm from weeks or months to hours.

In addition, it is useful to implement algorithms like this. These are classical graph algorithms, and implementing them, along with implementing a priority queue or a disjoint set data structure, is useful and important for any computer science student to have done at some point in their career. It's easy to find a library that can do this in only a few lines of code, and these are useful because it is a waste of time to reinvent the wheel, but implementing these on ones own allows a person to think about the same challenges that someone implementing the library would go through and see what goes on behind the scenes, and may allow a new person to make further efficiency improvements. In addition, depending on the application, one may need to make changes to the algorithm that a library does not allow for, requiring the individual to implement the algorithm themselves.

We were confident that Prim's algorithm would be faster than Kruskal's, but we wanted to implement both anyway and see real-world comparisons of the two, along with having the experience of implementing both and seeing the challenges that went with each. It is also useful to think about things like random number generation and how random these numbers actually are. If the numbers are not random, then we could get results that look one way, but the correct results should look a different way.

References

[1] Steele, M. J. Growth rates of euclidean minimum spanning trees with power weighted edges. *The Annals of Probability 16*, 4 (1998), 1767–1787.