CS124 Programming Project 1

Sean Anderson Ty

February 2021

1 Introduction

We perform a simulation to estimate the average minimum spanning tree weight for randomly generated graphs in four different cases: randomly uniformly generated edge weights and randomly uniformly generated points in the unit (hyper)cube in $\mathbb{R}^d$ for $d = 2, 3, 4$, where Euclidean distance is used as the metric, as per the project requirements. In this project, Java was the programming language of choice.

2 Results and Hypotheses

For convenience, we let n be the number of vertices in the graph we are simulating, d be the dimension, where a dimension of zero denotes the random edge weights case, and t the number of repetitions per simulation. The following are the results of a simulation performed for powers of 2 from $n = 128$ to $n = 262144$ (we also included $n = 2^{19}$). For each n, $t = 5$ repetitions were performed when simulating the average. We also included the running time of each simulation (for each n and dimension d) as empirical evidence of our run time claims later.

2.1 Zero dimensions

A summary of the results can be found in the table below:

n	Average Tree Weight (W)	Run Time
128	1.2324330885943162	5ms
256	1.2089755312823676	6ms
512	1.1881442112666172	20ms
1024	1.2118850217497408	79ms
2048	1.2132380250178643	282ms
4096	1.2023122230864531	1191ms
8192	1.2010440150883888	4816ms
16384	1.198781461682008	20083ms
32768	1.201752654754719	81578ms
65536	1.1992144308365655	339887ms
131072	1.2048110821944444	1370196ms
262144	1.2006058478303374	5235954ms
524288	1.2015392474523334	22443834ms

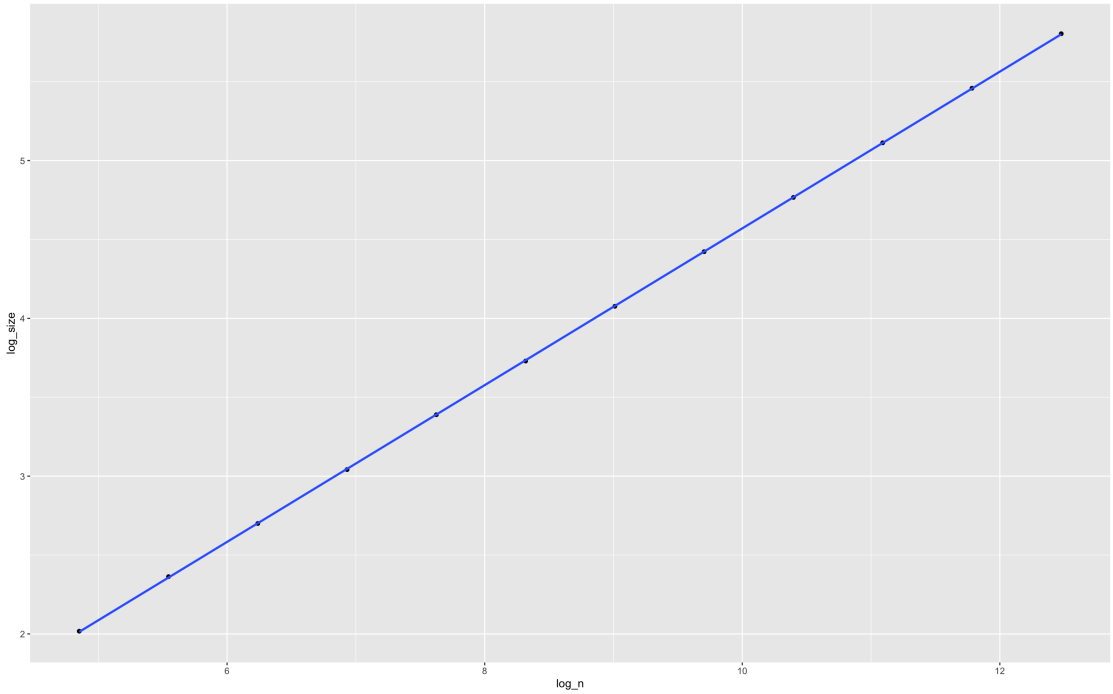
We notice that the average tree weight is roughly constant across n. Because of this, we hypothesize that $f(n) = c$ where $c \approx 1.2$, with some simulations farther than the hypothesized value likely due to the small number of trials. We also see that for large n, the run time roughly quadruples as n doubles.

2.2 Two Dimensions

A summary of the results can be found in the table below:

n	Average Tree Weight (W)	Run Time
128	7.516460256262485	5ms
256	10.623036539185577	2ms
512	14.879713065048335	9ms
1024	20.940261848258732	29ms
2048	29.64191517906581	74ms
4096	41.66164762863426	293ms
8192	58.95329189055026	2087ms
16384	83.32405729061047	7352ms
32768	117.51191232913493	35156ms
65536	166.11135715842465	138704ms
131072	234.73306297973264	570794ms
262144	331.64343032340355	3287510ms
524288	466.9602734973945	14270040ms

To determine $f(n)$, we attempted to fit a linear regression model with n as a predictor and the average weight as the response. We log-transformed the variables and fit a model in R (see Appendix for code):



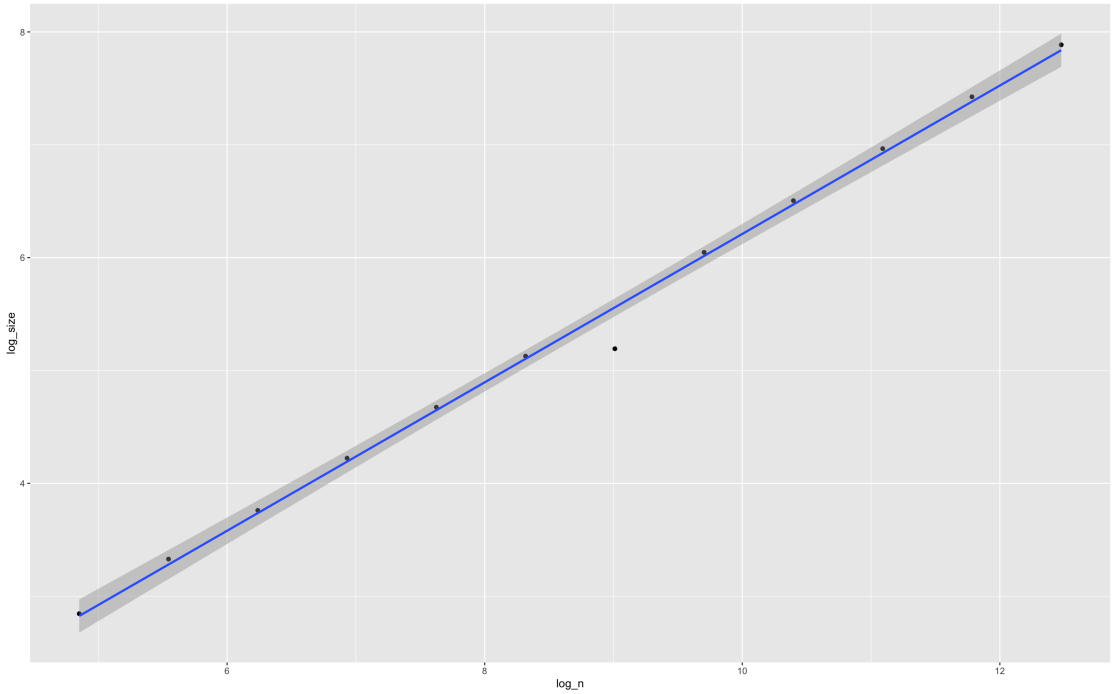
The graph suggests a linear relationship, which is what we want. We also obtained the linear approximation $\log f(n) \approx 0.49675 \log n - 0.39722$, where logs here are taken base e. This suggests that $f(n) \approx 0.672 n^{0.497}$ after exponentiating both sides. So, we hypothesize that $f(n) = cn^{\frac{1}{2}}$ for $d = 2$, where $c \approx 0.67$. We chose 0.5 as the exponent because it is cleaner and looks like a good approximation, since it is close to the obtained exponent. Just as before, the run times in the table quadruple for large n as n doubles.

2.3 Three Dimensions

A summary of the results can be found in the table below:

n	Average Tree Weight (W)	Run Time
128	17.195937382389552	6ms
256	27.896946634597885	3ms
512	42.955909024586035	13ms
1024	68.22215554683945	21ms
2048	107.16347377873038	81ms
4096	168.45594262217293	330ms
8192	179.94186521029488	1478ms
16384	422.5846497011231	8918ms
32768	668.1563888314165	38786ms
65536	1059.5967056179493	165999ms
131072	1677.1564185199975	722307ms
262144	2659.1928318217415	2848498ms
524288	3999.5283121945973	11525236ms

To determine $f(n)$, we again fit a linear regression exactly like before. This yields the following graph:



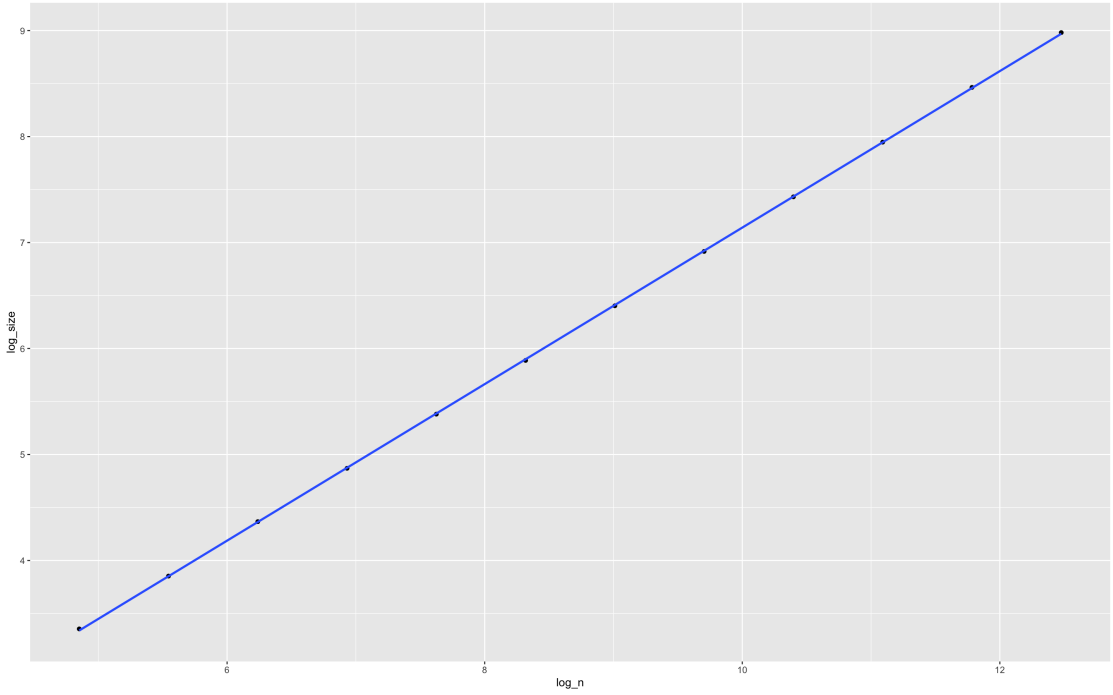
The graph suggests a linear relationship, which is again nice. We also obtain that $\log f(n) \approx 0.65753 \log n - 0.36563$, whence we get $f(n) \approx 0.694 n^{0.65753}$. So, we hypothesize that $f(n) = cn^{\frac{2}{3}}$ for $d = 3$, where $c \approx 0.69$. We rounded the exponent to roughly $\frac{2}{3}$ because it is "nicer" and close to the obtained exponent, and follows the same pattern as the previous case. Just as before, the run time roughly quadruples as n doubles, for large n.

2.4 Four Dimensions

A summary of the results can be found in the table below:

n	Average Tree Weight (W)	Run Time
128	28.622370079845297	6ms
256	47.121818951173395	3ms
512	78.74374835008923	8ms
1024	130.28369598482954	32ms
2048	217.186122620099	86ms
4096	360.5901247491421	317ms
8192	603.859740743497	1490ms
16384	1008.6451957736433	7798ms
32768	1687.7719595620172	35468ms
65536	2828.1775100954655	156713ms
131072	4740.192205102716	699238ms
262144	7950.495723351189	3085422ms
524288	13116.692394319734	12353798ms

To determine $f(n)$, we again fit a linear regression exactly like before. This yields the following graph:



The graph suggests a linear relationship, which is again nice. We also obtain that $\log f(n) \approx 0.73846 \log n - 0.24375$, whence we get $f(n) \approx 0.784 n^{0.73846}$. So, we hypothesize that $f(n) = cn^{\frac{3}{4}}$ for $d = 3$, where $c \approx 0.78$. We rounded the exponent to roughly $\frac{3}{4}$ because it is "nicer" and close to the obtained exponent, and follows the same pattern as the previous case. Just as before, the run time roughly quadruples as n doubles, for large n.

2.5 Generalizing to Higher Dimensions

The patterns for $f(n)$ suggest that for $d \geq 2$, we have $f(n) = cn^{\frac{d-1}{d}}$, where c is a constant that depends on d and d the number of dimensions. We also hypothesize that $0 \leq c \leq 1$ and is increasing with d (perhaps asymptotically approaching 1), because in all our models the intercept term is negative and are decreasing in absolute value as d increases.

3 Discussion

3.1 Algorithm Analysis

We used Prim's algorithm over Kruskal's due to space issues. This is because storing all edges of Kruskal's would take $O(n^2)$ space while Prim's would only require $O(n)$, unless we used a probabilistic algorithm and considered only edges of sufficiently small weights. Prim's was therefore more intuitive to implement, and we require only linear space as we can generate the edges when needed instead of storing all of them.

We modify the traditional implementation of Prim's by using for loops instead of heaps to determine the minimum distance. For each simulation given n and d, the outermost for loop denotes the number of repetitions we perform and average. Then, in each iteration, we reset the random number generator according to the system clock to simulate randomness.[1] Then, we initialize our dist array to store the distances of each vertex to the current MST, where we define this distance to be the minimum distance between the vertex and all points in the MST. We then iterate n times, each time including the closest vertex into the MST, where in each iteration we find this closest vertex by traversing through the array of distances and adding the closest vertex into the MST. Afterwards, we update the dist array to reflect the updated distances of the vertices not in the MST.

Now we check for correctness. Since we are effectively using Prim's, it suffices to prove that our modified for loop strategy achieves the same result as the heap. We can prove this by induction on the number of vertices in the MST. Indeed, in the first step a random vertex is added and the distances are initialized to the distances of every other vertex from the added vertex, which is indeed the minimum. Now assuming the dist array does indeed contain the shortest distances to the MST, we then add the closest vertex not yet in the MST found by traversing through the dist array to the MST. By definition, this vertex must be the closest to the MST that is not yet in the MST. We then update the weights by taking the minimum of the current dist value for the vertices not in the tree, and updating it with the distance between that vertex and the newly added vertex if that distance is smaller. Therefore, by construction, the dist array still contains the shortest distances. Since the closest node in each step is added, we are indeed effectively doing Prim's, which proves correctness.

Note that this algorithm clearly must terminate. Finding the closest node takes $O(n)$ time, and so does updating the dist array. Marking a vertex as visited takes $O(1)$ time, so since we loop through this $O(n)$ times, this takes $O(n^2) = O(|V|^2)$ time. Since we must also process the edges, this takes $O(|E|) = O(|V|^2)$ time, since the getWeight method works in $O(d)$ time and we are taking d as constant. Therefore, the run time of this algorithm is $O(|V|^2)$. Notice that this matches with our tables and observations earlier, where we noted that the run time quadruples.

[1] More on this in the Random Number Generator section.

Finally, the inMST and dist arrays take $O(n)$ space, the locations array takes $O(dn) = O(n)$ space, and the rest take $O(1)$ space. Thus, the space complexity is $O(|V|)$.

3.2 Growth Rate Analysis

We were rather surprised by the result in zero dimensions, where we hypothesized that $f(n) \approx 1.2$ is constant for all n. This may be explained by the fact that the total MST weight must be at least as large as the sum of the smallest $n-1$ edge weights, since the MST has $n-1$ edges. There are $\binom{n}{2} = \frac{n(n-1)}{2}$ edges in total and they are i.i.d. uniform, so from a well-known result in statistics, the expected weight of the i^{th} edge is $\frac{i}{N}$, where $N = \frac{n(n-1)}{2}$. This implies that the sum of the $n-1$ lightest edges is $\sum_{i=1}^{n-1} \frac{i}{N} = \frac{N}{N} = 1$, which is constant! So perhaps the reason why the average MST weight is always around 1.2 is because it is always around c times the average sum of the lightest edges in the graph where $c > 1$ (from our argument earlier), and c happened to be 1.2 due to probabilities of some kind (e.g. maybe the probability that the i^{th} lighest will be in the MST for each i).

We also noticed that $f(n) = cn^{\frac{d-1}{d}}$ for $d \geq 2$ dimensions, where $0 < c \leq 1$ is a constant depending on d. This might make sense since if the number of vertices grows, there will be more edges in the MST. However, more vertices would also mean that there are likely to be smaller edges in the tree since there is more opportunity to create smaller edges, which creates a trade off. Thus, we expect that $f(n)$ is increasing with respect to n and should be $o(n)$, which checks out with what we have. Finally, we also note that for fixed n, $f(n)$ grows as d grows. This is because adding dimensions adds more $(x_i - y_i)^2$ terms in the distance formula, which are always nonnegative, thus increasing the edge weights.

3.3 Random Number Generator Discussion

For generating the random numbers, note that we only require generating $\text{Unif}(0,1)$ random variables. This can be done by the Random object's nextDouble method. To simulate "randomness" as much as possible, we set the Random object seed before each iteration through the system clock. This then generates a continuous $\text{Unif}(0,1)$ random variable, which can be used for either the edge weights or the point coordinates as seen in the source code.

A Appendix

A.1 R code for plots and linear regression

The following is the R code used to plot the observations and fit a linear regression model on the transformed data:

```
library(ggplot2)
df <- read.table("dim4.txt", sep = " ")
colnames(df) <- c("n", "average_size", "runtime")
df["log_n"] = log(df["n"])
df["log_size"] = log(df["average_size"])

ggplot(data = df, aes(x = log_n, y = log_size)) + geom_point() +
```

```
  geom_smooth(method = "lm")
lm1 <- lm(log_size~log_n, data = df)
summary(lm1)
```