Prog1 124

Motolani Oyewole and Ricardo Skewes

February 2023

1 Results

average tree lengths				
Number of points	Average Tree Length 4d	Average Tree Length 3d	Average Tree Length 2d	Average Tree Length 1d
128	6.81	4.28	1.72	0.202
256	11.12	6.72	2.67	0.204
512	18.76	10.29	3.56	0.211
1024	30.49	16.41	5.19	0.206
2048	51.83	25.352	7.14	0.202
4096	84.686	40.16	9.98	0.21
8192	140.457	63.23	14.309	0.2

1.0.1 graphs

1.1 $f(n)$

Our guess of f(n) comes from looking at how the average tree length grows with the log_2 values that we have for our x value. We first conclude that the graph looks like the function for the square root of x for 2 dimensions. In 2 dimensions the function lines up very closely with the square root function and is of the form Tree Length= $0.1576\sqrt{n}$ The correlation is very high and the points line up almost perfectly with this function.

In higher dimensions the correlation is not quite so strong. This points don't line up quite as well with the graphs, we tried putting in the cubed root, but it didn't quite give us the growth rate we were looking for so we tried the formula $x^{\frac{3}{4}}$ and we saw that it lined up very well with the graphs and the information that we had provided. But not as well at $x^{\frac{1}{2}}$ had lined up for 2 dimensions, so we then tried this same value for 4 dimensions and we saw that the correlation was again almost perfect like the square root function had been for 2 dimensions and we saw that the equation came out to be Tree length = $0.165x^{\frac{3}{4}}$.

Now we saw that this was similar in form to the 2-d case and worked just as well for the 4-d case so we reanalyzed the 3-d case with this information expecting to get a similar result and we found that this is the case when we look at the $\frac{2}{3}$ power we find that for 3 dimensions the equation must be of the form Tree length = $0.154x^{\frac{2}{3}}$. This gives us the very good correlation we were hoping for and we conclude that the general formula for the length of the tree is

$$f(n) = 0.15 * x^{\frac{d-1}{d}}$$

Where d is the number of dimensions that you are dealing with. This even fits the 1-d case because then we would have a constant regardless of x that is close to 0.15 and that is exactly what we see in the data that we collected. They were all almost exactly 0.2 regardless of the number of points

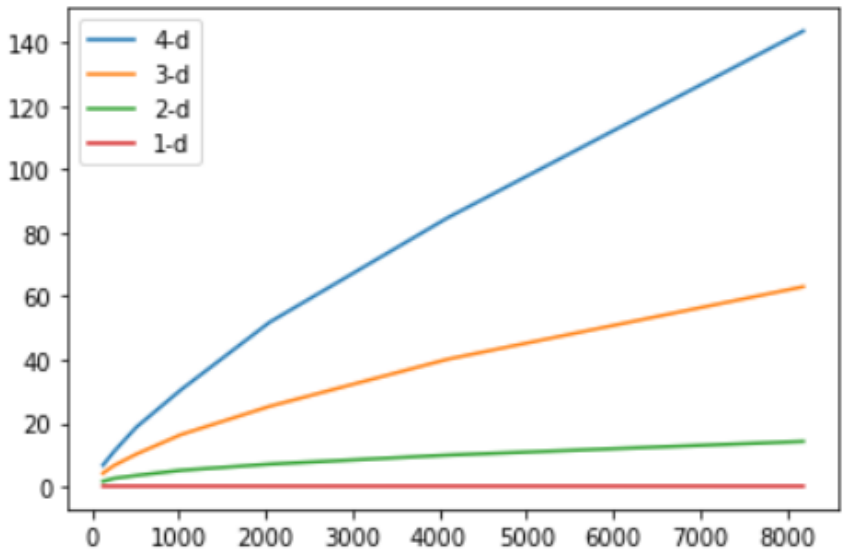

Figure 1: From this we can see that there are very different growth rates. The growth rate is clearly below linear growth for most except for in 1 dimensions where it is linear and there is no growth in average tree length. We can now try to analyze there individuallly

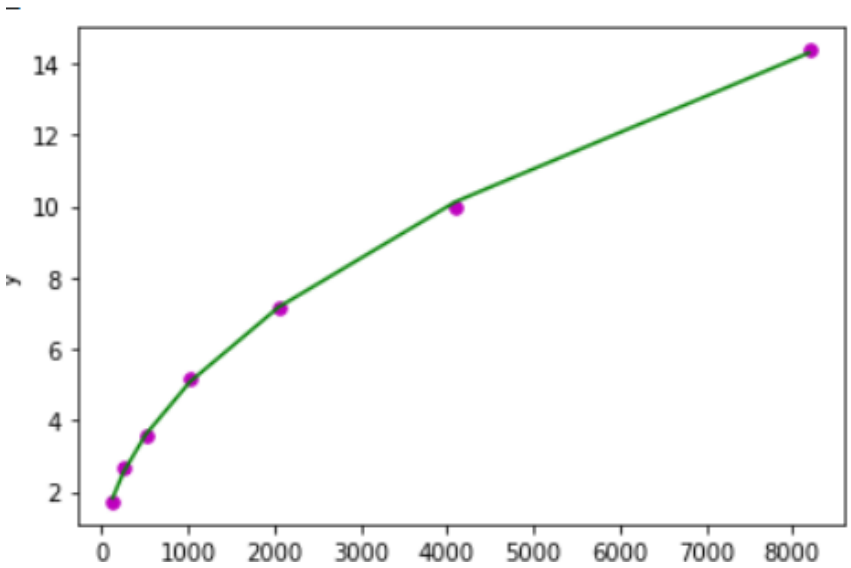

Figure 2: This shows us that there is a very clear relationship between the equation we came up with and the graph. The points line up very well visually allowing us to conclude this is the true distribution

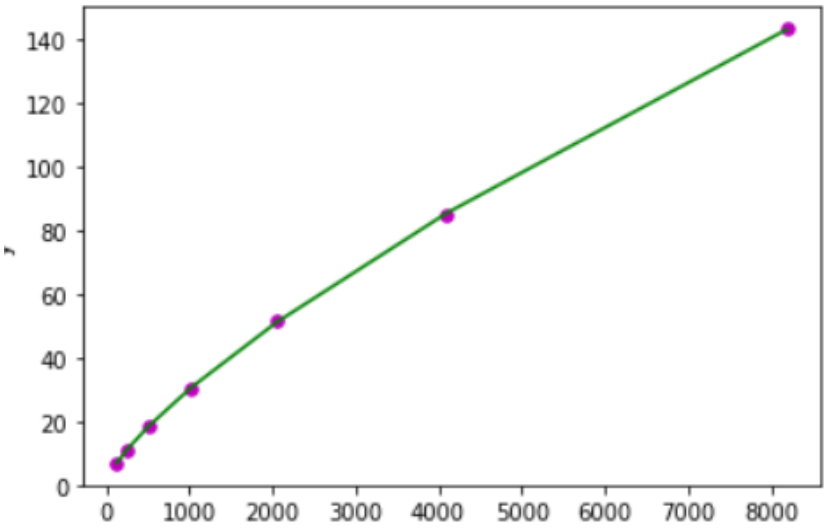

Figure 3: This shows us that there is a very clear relationship between the equation we came up with and the graph and we can see that the initial growth is less steep but it evels off more slowly

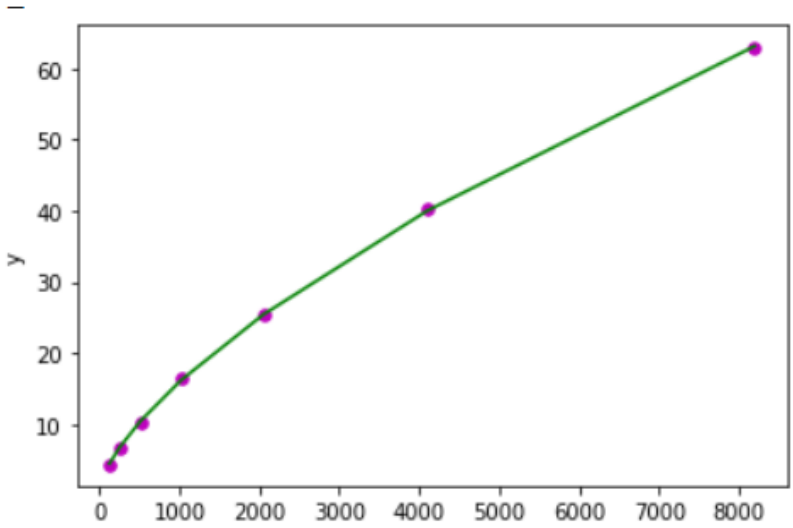

Figure 4: This also shows us that there is a very clear relationship between the equation we came up with and the graph. The graph again doesn't go up as sharply as first but levels off slower as well

2 Discussion

For our algorithm we made a slight modification to Prim's Algorithm to make it that the algorithm returned the length of the tree as opposed to the actual tree itself. This was a relatively simple part of the process, but required that we slightly tweak out pseudocode. Since we no longer cared about the points, we didn't track the 'previous' value since it didn't matter to use where the path came from as long as we knew its length. The way we counted this was by by encoding the distance of each element from the set S as being part of the structure that encoded each point. So the point had the x, y and z locations as well as the distance from the set that we calculated every time we added a new point to the set. This allowed us to be able to just add up all the distance values from all the points in the set to get out MST. We can see that this is correct because if we consider some vertex v in out MST T, then it will only get added to T when it is closer to the set S than any other point in G that still needs to be added to the space. If this happens then according to our algorithm it should have the distance that makes it closest to the set S of points we have already added to T. This will be the case because for each point we add to S we calculate the distance of every points in $V(G)$ S and if that distance is less that the distance we started off with we update the distance. Since every point starts at infinite distance this algorithm will work for from the first iteration on. So when we take out the vertex v we can take the distance that it carries with it and add it to out tree length value that we have been calculating and know that it encodes the length of the edge that connects it to the set. Then summing this up over all of the points, we can see that this will give us the tree length of the tree that we are dealing with.

One issue we came across while we were programming was finding a good way to represent the vertices and the weights of the edges between them. We ended up creating our own class of object that would represent each vector. This class of object we represented by its name and also had connected to it a list of all the elements that it has en edge to. We liked this construction because it would allow us to be able to be able to quickly see all the edges that a point . This data structure allowed us to look at each point quickly and see all the other points that it is connected to all the other points around it. We learned quickly though that we should put a limit on how large the set of points that it connects to are or else we would spend a really long time trying to parse out really big connections. This caused us to try to come up with a function to optimize this process, and we will discuss the creation of this function next.

In order to speed up our run time we tried to find a function $k(n)$ that would allow us to eliminate the unnecessary edges of our graph. To do this we began thinking about the structure of our space and the points in it. So the fist simplification that we did in the creation of this function was to think of the d-dimensional cube as being a circle. This simplification made the problem a little bit easier to think of mathematically because we know points that go from one corner of the square to another will definitely not be in the minimum spanning tree since the graph is connected. This allows us to only think about the unit hyper circle in our analysis. In addition, we can think of the spread of the points in this unit sphere. Since this is a uniform distribution we can look at the most likely spread in which they uniformly fill up the space. So let's say we starts off by looking at one point in our space s. We want to know the distance at which we will be sure that there will be other points close to our initial point s. We can re-frame this by thinking of the n points having d-dimensional circles around them that grow until they together will the whole space. This set up allows us to think about the problem as being a result that comes from sphere packing and we will know that at this critical radius at which I will call 'r', is inversely proportional to the number of points. So we can say that 1 $r^d n$, removing numerical constants for volume of a sphere we can see that this equation volume of the sphere to the volume of all the points combined. This may seem to be the end of the story but in reality we have only considered the expected case where all the points are uniformly spread throughout the space. Even though we are drawing from a uniform distribution, there is still a probability that we end up with the points being very separated from one another. In this case, if we choose a limit that is too small with $k(n)$ our algorithm may end up failing because it doesn't reach the two groups of points. So we want to choose an r that minimizes this probability as well. So let's think of the limiting case in which we have two separate clusters that have a large distance between them. In this case say we choose a ball of radius r, then the probability of another point being in its open ball can be expressed as $\frac{r^d}{1}$. so the probability that

it isn't in a given ball is $(1 - r^d)$ so the probability of it being outside of the radius of N-1 balls if, we had an outlier is $(1 - \frac{1}{N^{N-1}})$ we can round to order of magnitude and say that this probability is just $(1 - \frac{1}{N^N})$. Since we want to minimize this probability we choose r to be $\frac{1}{N^{1/d}}$ since this will give us a probability of being outside of all of out spheres of $(1 - (\frac{1}{N^{1/d}})^d) = (1 - \frac{1}{N}$ which would grow as N goes up, but this value decays much slower than using a function that decays on the order of $\frac{1}{N}$ which would give us a probability of having points outside of our balled neighborhoods of $(1 - \frac{1}{N^d}$. So we chose our $k(n) = \frac{1}{N^{\frac{1}{d}}}$

The growth rate was really surprising. I didn't think the growth rate would be so low. I expected it to grow logarithmically, but it actually grew according to fractional powers. It was hard to figure out why this might be the case. It seems like it has to do with the fact that we are taking lengths that can go in many more directions. In 1-d there is y one way in which an edge can extend so its just going from one spreading linearly out from the source poitn until it reaches the end points. On the other hand in higher dimensions, this can start to trace out area in the space similar to a fractal. This is what gives it fractional dimensional values in terms of length and how it scales with n. The pahts can begin to trace out a plane or hyper plane in the higher dimensional spaces so the distance can scale fractionally with respect to how long they are going to be.

For the run time, we only were able to run out algorithm on smaller values on n but we can see that it was an $O(n^2)$ running time because out code wast really efficient, but we could see that as we doubled the about of points it woudl take 4-6 times the amount of time to run. This may have been due to the fact that we were running the code on a machine that was already using alot of ram which slowed it down, but they scaling of runtimes from each of the point values that we used showed us this was the growth rate. We can tell it wasn't linear because the operations that we were using don't allow the problem to be done in linear time. The sorting of the heap is an $O(n)$ operation and we did it n times. In addition, we know our algorithm was of logarithmic time because the amount of time to run when varying between each n that we were given wasn't decreasing but rather increasing, so this would be an unrealistic run tiem