CS 124 Programming Assignment 1: Spring 2022

Your name(s) (up to two): Christy Jestin, Ash Ahmed

Collaborators: (You shouldn't have any collaborators but the up-to-two of you, but tell us if you did.)

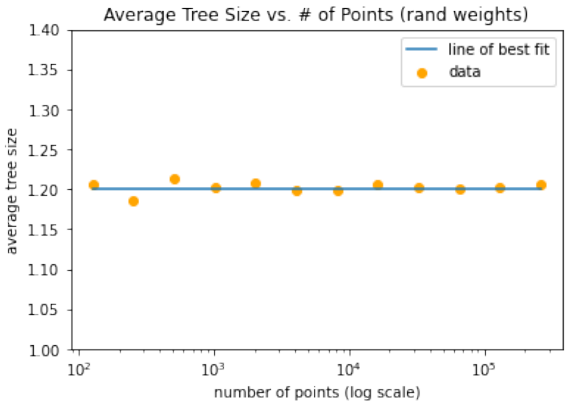
No. of late days used on previous psets: 2
No. of late days used after including this pset: 4

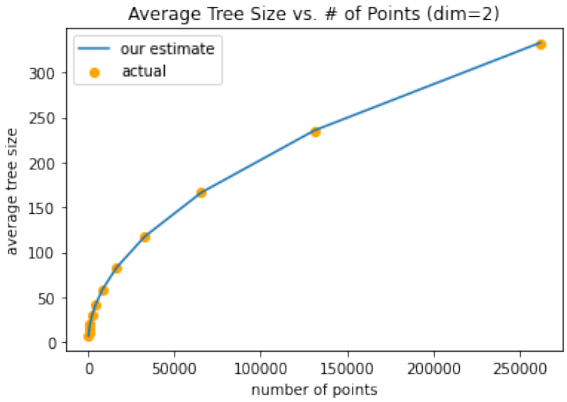
Results

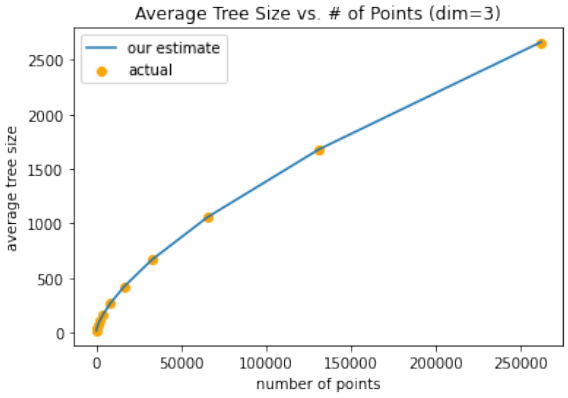
See the below table and graphs for our data (average tree size for varying values of n and dimensions). All entries in the table were the average of 5 trials.

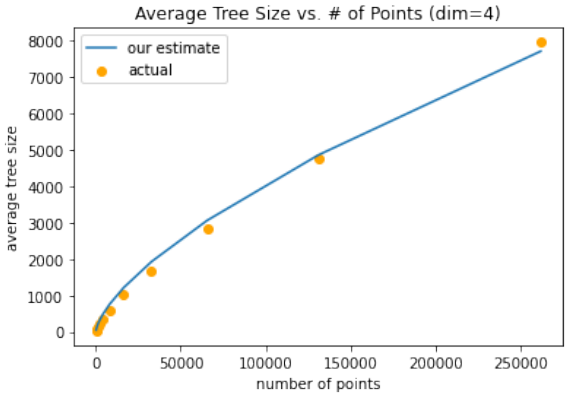
num points	rand weights	dim=2	dim=3	dim=4
128	1.205680	7.601829	17.809147	28.121256
256	1.185011	10.675033	27.693384	46.785278
512	1.212662	15.092278	43.370174	78.195328
1024	1.202702	21.049362	67.910927	129.830444
2048	1.207653	29.622177	107.219315	216.680939
4096	1.199081	41.808372	169.286438	361.006378
8192	1.199311	58.910259	267.165283	602.675293
16384	1.206522	83.051468	422.843414	1008.771118
32768	1.201661	117.433556	668.567871	1687.606201
65536	1.201136	166.102737	1058.312256	2829.706543
131072	1.202527	234.588867	1676.750732	4738.087402
262144	1.205643	331.629486	2658.213379	7950.791992

Plots with our guesses and actual data:









Our guesses for $f(n)$, used on above plots:

i. random weights: $f(n) = 1.2$

ii. 2-dimensional: $f(n) = 0.65n^{\frac{1}{2}}$

iii. 3-dimensional: $f(n) = 0.65n^{\frac{2}{3}}$

iv. 4-dimensional: $f(n) = 1.88n^{\frac{3}{4}}$

Discussion

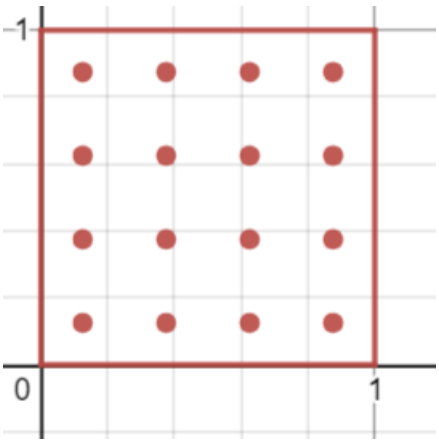
1. Which algorithm did you use, and why?

   We used Kruskal's algorithm. The complexity of both Prim's and Kruskal's algorithm is $O(m \log n)$ when you consider the time required to sort all edges for Kruskal's algorithm. Note m is the number of edges and n is the number of vertices in the graph. Prim's algorithm can be run in time $O(n^2)$ with an array implementation of a priority queue, but $O(n^2)$ is only better than $O(m \log n)$ for sufficiently dense graphs, and we weren't sure if the graph would still be dense enough after pruning. Similarly, Prim's algorithm can be run in time $O(n \log n)$ with a Fibonacci heap implementation of a priority queue, but we were worried about the difficulty of implementing a Fibonacci heap.

   We ended up choosing Kruskal's because it seemed easier to implement: the key sections of code needed were just 1) the Union-Find data structure 2) graph generation 3) Kruskal's itself. While Prim's implementation is also straightforward (very similar to Dijkstra's), we didn't want to have to implement our own heap data structure. The union find data structure ended up being quite short, and the logic was fairly straightforward to implement. All in all, Kruskal's algorithm seemed easier and more intuitive to us. Plus we'd get to work with the union find data structure which is pretty poggers.

2. Are the growth rates (the $f(n)$) surprising? Can you come up with an explanation for them?

   The growth rates were surprising in the sense that we did not expect to be so spot on with our intuition. Our logic was that since we're choosing the value of each coordinate uniformly, the points will be uniformly distributed within the d-dimensional spaces (unit square/cube/hypercube). More specifically, we'd expect the points to be spread out in a pattern resembling a grid. While this example is in 2-dimensions, the concept extends to higher dimensions.



   Suppose we have s points along a grid line. Since this applies for all dimensions, we'd have s^d points in total (in this case $s = 4$ and we have $4^2 = 16$ points in total). Thus s is about $\sqrt[d]{n}$, and each point in our shape should have another point (on average) about $\frac{1}{s}$ units away in each direction (up, down,

left, right etc.). This follows from the fact that each side has length 1, and there are s points along each grid line. Since we have $n-1$ egdes in an MST, the sum of the weights of the MST should be about

$$n \cdot \frac{1}{s} = \frac{n}{\sqrt[d]{n}} = n^{1-\frac{1}{d}}.$$

This tracks very well with the empirical data except there is a constant factor for each model. This factor likely reflects some property of the expected value of the distance between nearby points.

For the case where we randomly generated weights, we can roughly expect the MST edge going toward each vertex to be the shortest edge going toward that vertex. Since the edge weights are chosen uniformly at random, we can model the shortest edge by using the first order statistic of the uniform which has the Beta distribution with the parameters $\alpha = 1$ and $\beta = n$. The expected value of the distribution is $\frac{1}{n+1}$, so our total should be $n - 1 \cdot \frac{1}{n+1} \approx 1$. While the empirical average looks constant as n varies, its value is around 1.2 instead of 1. This may be due to the fact that the edges aren't independent in the sense that all of the edges drawn for a vertex i are shared with some other vertex j i.e. while every undirected edge is drawn independently from the uniform distribution, each edge is shared between two vertices. It's also possible that the average is 1.2 instead of 1 because we aren't actually always selecting the shortest edges from each vertex for the MST.

3. How long does it take your algorithm to run? Does this make sense? Do you notice things like the cache size of your computer having an effect?

   Our algorithm generally takes four times as long as when the number of points is doubled and all else is held equal. This is not the case when the number of points is sufficiently small. This makes sense because all of the functions have some fixed start up time. This pattern makes sense because there are $O(n^2)$ edges where n is the number of points, so the number of edges quadruple when the number of points double. Even with effective pruning, it still takes a good amount of time to calculate weights and process all edges. We didn't notice the cache size having an effect.

4. Did you have any interesting experiences with the random number generator? Do you trust it?

   The random number generator was outputting the same values when we didn't pass in the current timestamp as a seed. We trust it now after testing that it returns new values on every run.

5. Pruning strategy?

   Our pruning strategy was to use the max weight of any edge on the MSTs generated on the previous n value as a "threshold" for the maximum distances we would consider in the creation of our graph for the current n value. For example, if we were trying to run 3 dimensions with $n = 4096$, we would look at the maximum weight included in any MST generated with 3 dimensions and $n = 2048$ (which was ≈ 0.125818). Then we only considered a weight if it is less than 0.125818 while generating our graphs for $n = 4096$. This makes sense since as n increases, the points are getting denser and denser within our shape, so the distances between points is decreasing. This was also reflected in the data (before we implemented pruning) as the maximum weight edge used in an MST was always strictly decreasing.

   To do this in a more organized fashion then writing down threshold numbers and manually typing them in for a given n and *dim* value, we kept track of the max weights in a vector in unique files (had four files with the naming convention "graph-info-dim.cc"). We wrote to these files with updated vectors as we got the max weight for a given n value (see the print info function) and read from them in order to get the needed threshold for graph generation. For the highest number of points (65536 to 262144), we needed tighter thresholds for the code to run efficiently, so we also allowed the randmst

file to take in manually inputted thresholds with flag 3. We only used the manual thresholds for a handful of the higher n values and ended up automating most of the process with a Python script.