ProgSet 1

CS 124: Data Structures and Algorithms

Due: Wednesday, February 21, 2024 **Denny Cao and Ossimi Ziv**

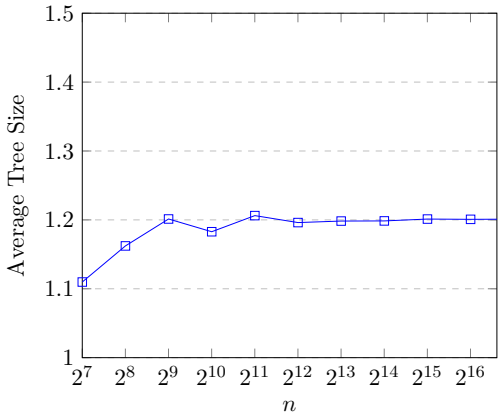
---

§1 Part 1: Quantitative Results

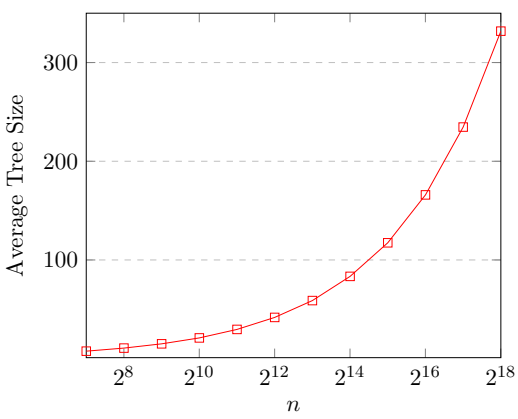
§1.1 Results

n	Dimensions			
	0	**2**	**3**	**4**
128	1.1098	7.5950	16.9045	28.4403
256	1.1622	10.7016	27.6048	47.1692
512	1.2013	15.0169	43.1509	78.0112
1024	1.1828	21.0029	67.9189	130.7465
2048	1.2062	29.6215	107.6589	216.3232
4096	1.1961	41.7762	168.9072	361.3350
8192	1.1983	58.8374	267.8582	602.5550
16384	1.1985	83.3555	422.9353	1008.3248
32768	1.2012	117.3818	668.4247	1690.6445
65536	1.2008	165.9153	1057.9618	2828.1298
131072	1.2011	234.6354	1677.4447	4741.7945
262144	1.2022	331.8225	2658.2786	7948.8444

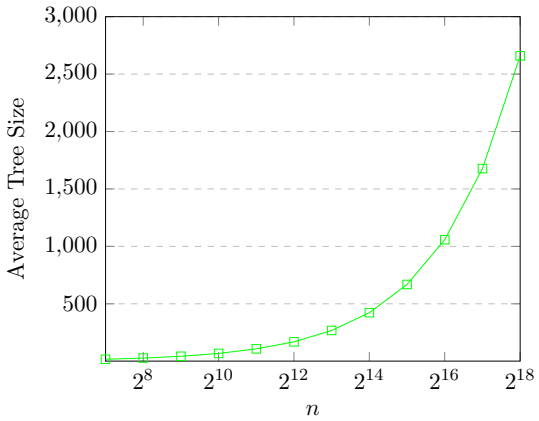
Table 1: Average Tree Size for Different Dimensions and n



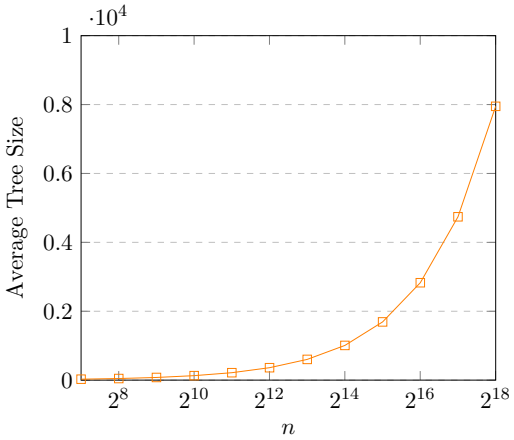
(a) Average Tree Size vs. n for Dimension 0



(b) Average Tree Size vs. n for Dimension 2



(c) Average Tree Size vs. n for Dimension 3



(d) Average Tree Size vs. n for Dimension 4

Figure 1: Average Tree Size vs. n for Different Dimensions

§1.2 Guess for $f(n)$

With all cases being random vertices generated from a uniform distribution of unit shapes (line in 0D, square in 2D, cube in 3D, hypercube in 4D). Then, the density of the graph is given by

$$\rho = \frac{n}{V}$$

Where n is the number of vertices, and V is the volume of our space in which we generate the vertices. In this case, $V = 1$, per the area/volume of our line/square/cube/hypercube. Therefore the density is just the number of vertices:

$$\rho = \frac{n}{1} = n$$

We can let v be the volume that each vertex takes up within our volume V, therefore the total volume of our vertices will be

$$V = nv$$

Since V = 1, we can solve for the volume that each vertex occupies in our dim-D cube to be

$$1 = nv$$
$$v = \frac{1}{n}$$

Finding the average distance between each of each vertex, d, can be done by finding the edge length of our dim-D cube, knowing that the volume scales exponentially as we increment the dimension:

$$v = d^{dim}$$
$$d = v^{\frac{1}{dim}}$$

Substituting the our earlier value of v, we can obtain the average value of an edge to be:

$$d = (\frac{1}{n})^{\frac{1}{dim}}$$

Thus we can conclude, that since there are $n-1$ edges in our MST, the average length of the MST will be

$$length = (n-1)(\frac{1}{n})^{\frac{1}{dim}} = n^{\frac{1-dim}{dim}} - 1(\frac{1}{n})^{\frac{1}{dim}}$$

As n gets larger, $-1(\frac{1}{n})^{\frac{1}{dim}}$ will approach 0 and the the effect of the second term will be negligible, giving us a final approximation for $f(n)$:

$$f(n) = n^{\frac{1-dim}{dim}}$$

We can also deduce and verify this expression geometrically case by case by modeling the random assortment of vertices and edges as points on their respective dimensional shapes (line, square, cube, hypercube) and calculating the average edge length:

- **0D**: Our vertices are generated uniformly between 0 and 1. In our case, we can model this geometrically as a uniform distribution of points along a unit line in 1-D. The uniform distribution of vertices can be thought of as diving the unit line into n equal segments, each a length of $1/n$, thus the average edge length in our MST will be $1/n$. Given that there are n-1 edges in an MST, the average length of our MST is $\frac{n-1}{n}$. As n gets larger, this expression will approach $\frac{n}{n}$ which results in a constant ratio of MST size in 0-D.
- **2D**: Our vertices are generated uniformly in a unit square. Akin to the 0-D/1-D case, the 2-D case can be thought of as dividing the unit square into n equal squares, each with an area of $\frac{1}{n}$ Thus, the average distance between any two vertices in this uniform distribution can be computed as the edge of such a square $= \sqrt{\frac{1}{n}}$. For the length of our MST, we have n-1 edges, thus the average length of an MST generated from a unit square will scale with $(n-1)\sqrt{\frac{1}{n}} = n^{\frac{1}{2}} - \sqrt{\frac{1}{n}}$. For larger n's this final term approaches 0 rapidly, and the expression simplifies to essentially $n^{\frac{1}{2}}$
- **3D/4D**:The same logic can be extended for our 3-D cube and our 4-D hypercube. The vertices in the cube can be thought of as creating smaller cubes with area $\frac{1}{n}$. Thus each edge in this cube has a length of $\sqrt[3]{\frac{1}{n}}$. Therefore our average MST from a 3-D cube can be approximated as $(n-1)\sqrt[3]{\frac{1}{n}} = n^{\frac{2}{3}}$ Similarly, we can extend this logic for our 4-D hypercube, giving us an average MST length of $n^{\frac{3}{4}}$.

Finalizing this formula, we can deduce that the approximation of $f(n)$ follows a pattern of $cn^{\frac{1-d}{d}}$ where d is the number of dimensions and c is a constant factor to account for arbitrary scaling. This conclusion is supported by the empirical evidence for our trials.

Our "0-D" graph (treated as 1-D in this formula) approaches a constant ratio, and the higher dimensional graphs exhibit concave up behavior on a logarithmic axis, indicating some polynomial growth, supporting our guess that $f(n)$ scales on the order of $n^{\frac{1-d}{d}}$

§1.3 Line of Best Fit Approximation of $f(n)$

Let $f_d(n)$ denote the regression model for $f(n)$ for dimension d:

- $f_0(n) = \frac{0.0213n - 1.3180}{0.0178n - 1}$
- $f_2(n) = 0.648368(n-1)(1/n)^{\frac{1}{2}}$
- $f_3(n) = 0.6497985(n-1)(1/n)^{\frac{1}{3}}$
- $f_4(n) = 0.687412(n-1)(1/n)^{\frac{1}{4}}$

§2 Part 2: Discussion

§2.1 Correctness

Proof. We determine the average MST size for complete graphs of size n by:

- Creating a graph G with n vertices in $[0,1]^d$, where d is the dimension (for 0D, $d = 1$) and adding an edge between two vertices if the distance between the head and tail of the edge (computed using `euclidean-distance` is less than a `threshold` (we will later describe how to find this threshold) and if the edge is added, sets the weight of the edge to the distance between the points.
- We run `Kruskals` on G, and keep track of the total cost of the tree by adding the weight of edges added to the MST to a variable.
- We repeat this 5 times and return the average of the total weight of edges across all 5 trials.

Assuming our implementation of Kruskals is correct, we will show that pruning edges less than `threshold` maintains the MST.

> **Claim 2.1 —** If we prune all edges e such that $w(e) >$ `threshold` and the resulting graph G' is still connected, then the cost of the MST T' of G' is the same as the cost of the MST T of G.

Proof of claim. By contradiction. Suppose that T and T' have different costs.
T' is a subgraph of G, as T' uses edges from G', which is a subgraph of G. Since V', the vertex set of G', is the same as V, the vertex set of G, and T' spans G', it follows that T' spans G.
The edges of T' are all less than or equal to `threshold`, as all edges greater than `threshold` were pruned.
Let E_T be the set of edges in T. If $\max(E_T) \leq$ `threshold`, then T is the same weight as T', as all edges in T are less than or equal to `threshold`, meaning that T is the MST of G'. If $\max(E_T) >$ `threshold`, then T is not the MST, as T' is a tree that spans G and has a cost less than T. ⨳
We reach a contradiction, and thus, the cost of the MST T' of G' is the same as the cost of the MST T of G. ■

Claim 2.2 — If we prune all edges e such that $w(e) > \sqrt[d]{\frac{\ln n}{n}}$, then it is a extremely high probability that the resulting graph G' is still connected, where d is the dimension of the graph (Note that 0D is treated as $d = 1$).

Remark 2.3. This threshold is inspired by Erdos and Renyi's in paper "On Random Graphs", where $p = \frac{\ln n}{n}$ is the threshold for connectivity for a complete graph where the p is the edge probability, as $n \to \infty$.

We set `threshold` based on the case of d:

$$\texttt{threshold} = \sqrt{d}\left(\frac{\ln n}{n}\right)$$

Proof of claim. The probability that a vertex is not connected to any other vertex in a complete graph is given by:

$$P(v \text{ is isolated}) = (1-p)^{n-1}$$

The expected number of isolated vertices in a complete graph is given by:

$$E[N] = n(1-p)^{n-1}$$

Let $p = \sqrt{d}(\frac{\ln n}{n})$. We analyze the expected number of isolated vertices in a complete graph as $n \to \infty$:

$$\lim_{n\to\infty} n(1-p)^{n-1} = \lim_{n\to\infty} n\left(1 - \sqrt{d}\left(\frac{\ln n}{n}\right)\right)^{n-1} = 0$$

The edge weights are generated by euclidean distances between points in $[0,1]^d$, where d is the dimension of the graph (Again, note for 0D, $d = 1$). The euclidean distance between two points in $[0,1]^d$ is uniformly distributed between 0 and $\sqrt{d}$, and thus, the edge weights are uniformly distributed between 0 and $\sqrt{d}$. With a threshold of $\sqrt{d}(\frac{\ln n}{n})$, we can view this as setting an edge probability p of $\frac{\sqrt{d}(\frac{\ln n}{n})}{\sqrt{d}} = \frac{\ln n}{n}$, as the CDF of the uniform distribution of edge weights is $\frac{\sqrt{d}(\frac{\ln n}{n})}{\sqrt{d}}$.

As $n \to \infty$, the probability that the graph is connected approaches 1, and thus, we can use this threshold to ensure that the graph is connected with high probability. ■

We combine these two claims to show that our algorithm is correct. If we prune all edges e such that $w(e) > \sqrt{d}\left(\frac{\ln n}{n}\right)$, then it is a extremely high probability that the resulting graph G' is still connected, and the cost of the MST T' of G' is the same as the cost of the MST T of G. Thus, our algorithm is correct. □

§2.2 Asymptotic Runtime

Claim 2.4 — The runtime of the algorithm is $O(n^2 + E \log E)$.

Creating all the vertices takes $\mathrm{O}(n^2)$ as it requires iterating through a double for loop that can be modeled as the sum of an arithmetic series from 0 to n. Thus the runtime is $\mathrm{O}(\frac{(n-1)((n-1)+1)}{2}) = \mathrm{O}(n^2)$. Within this double for loop are calls to the threshold

function — a constant math operation (O(1)); comparisons to the threshold and an append operation to an array — also O(1); and within the TwoDim, ThreeDim, and FourDim functions, a call to the euclidean distance function — also O(1).

Within our Kruskal algorithm, sorting the edges uses Java's built in timesort, which has $O(n \log(n))$, in this case n is the number of edges, so sorting has a runtime of $O(E \log(E))$. It's outer while loop iterates through all the edges, having $O(E)$

Deeper within our Kruskal algorithm, our union-find algorithm has two functions, union and find: Find runs recursively, continuously running find and checking a given array index until it reaches the parent. However, to improve runtime we cache the result of each find operation and compress the tree so each node points directly to the result of the find, essentially halving the time needed for a find, and in the best case making it nearly a constant time. This process takes $O(\log(n))$. The Union method simply references the find and performs a constant operation of adjusting the pointer, also taking $O(\log(n))$.

Combining the aforementioned runtimes and considering the largest power present in each part of the program, our full algorithm has a runtime of $O(n^2 + E \log(E))$.

§2.3 Further Discussion

Although it is possible to obtain a faster runtime of $O(M \log_{M/N} N)$ with `Prims`, as complete graphs are very dense, and even with pruning, $M >>> N$, we chose to implement `Kruskals` for its simplicity and ease of implementation without needing to implement `D-ary Heap`, opting to focus on the pruning aspect of the problem and if time permitted, to attempt a more efficient implementation using `Prims`.

We also tested an alternate threshold of

$$\texttt{threshold} = \sqrt[d]{\frac{\ln n}{n}}$$

which worked, and if time permitted, it would be really interesting to analyze if there is an even tighter bound with high probability of maintaining a connected graph.

When running the algorithm, we noticed that the system we ran the code on, had a very significant impact on the time it took the algorithm to compile. We ran the code on two different computers primarily, a Macbook with an M2 processor and a Desktop with a much more powerful processor which contributed far more than we expected to the runtime of the computer. The larger number of cores and inherent cache of the desktop proessor allowed it to compile in nearly a third of the runtime of the Macbook testing.

Another notable phenomenon we noticed was the surprisingly taxing process of printing each line of edge generation. To verify our algorithm's progress during testing, we incorporated a basic print statement to print the value of each edge that was generated. When conducting this test, we observed that the simply inclusion of the print statements, which we assumed would be a quick and cheap operation hindered the runtime of our code tremendously – changing the runtime from a few minutes without print to nearly an hour with.