Programming Assignment 1
Expected Weight of Minimum Spanning Trees

Adarsh Hiremath and Artemas Radik

February 22, 2023

1. Introduction

In this report, we examine how the expected weight of various minimum spanning trees (MSTs) grows as a function of n vertices. We consider two types of graphs: 0-dimensional random complete graphs and 2,3,4-dimensional random complete Euclidean graphs. 0-dimensional random complete graphs are complete graphs with the weight of each edge e distributed as $W_e \sim Unif(0,1)$. The 2,3,4-dimensional random complete Euclidean graphs are generated in a unit square, cube, and hypercube respectively. Unlike the 0-dimensional case, the weights for the Euclidean graphs are computed using the Euclidean distance between the vertices composing an edge.

We implemented Kruskal's Algorithm in Rust to generate and compute the expected weights of our MSTs. In the 0-dimensional case, we improved our algorithm by lazily generating the sorted list of edges for Kruskal's algorithm using the exponential distribution, implementing multithreading, and hyper-optimizing a function $k(n)$ to sparsify our graph. This resulted in an incredibly fast 0-dimensional algorithm, with the $n = 262144$ case running to completion in ~ 100 ms. We implemented similar optimizations for the 2,3,4-dimensional random complete Euclidean graphs, also achieving decent results.

2. 0-dimensional Random Complete Graphs

Kruskal's algorithm works by sorting edges by weight, repeatedly adding minimum weight edges that don't form a cycle, and stopping once the MST has $|V| - 1$ edges. This has a relatively efficient $O(m \cdot log(n))$ runtime where m and n are the edges and vertices in our graph. Generating and storing the graph, however, is far more costly. Thus, our approach must efficiently implement this portion of our algorithm.

2.1 The Naïve Approach

Consider the naïve approach from class:

- Iterate through the n vertices and append them to a vertex list, V.
- Iterate through each pair of vertices and use a random number library to generate a weight w_e for all edges e such that $w_e \in [0,1]$. Append each edge to an edge list, E.

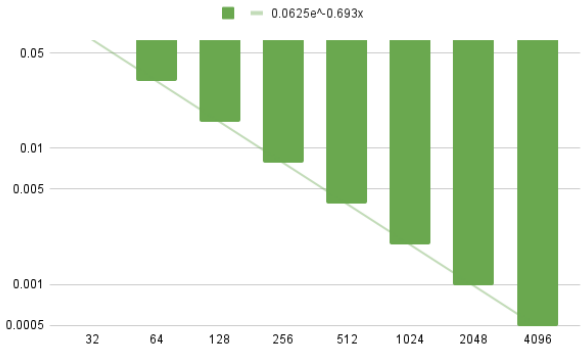
This has space complexity $O(n + \frac{n!}{2!(n-2)!})$ since we store all n vertices once and store all $\binom{n}{2}$ pairs of vertices as our edges since our graph is complete. Generating and storing every edge causes memory overflow and is inefficient because we don't use most edges in our graph to generate our MST.

2.2 Sparsifying Random Complete Graphs

Instead of storing every edge in our graph, we chose to build a sparse graph where every weight w_e in our edge list is upper bounded by some function $k(n)$. If $w_e > k(n)$, we disregard the weight. The filtering function we used for our 0-dimensional algorithm was:

$$k(n) = \frac{4}{\sqrt{n}}$$

We determined $k(n)$ by setting the filter function at $k(n) = 1$ and repetitively halving the output of $k(n)$ until the output diverged too far from the original. We repeated this process for $n = 32, 64, 128, 256, 512, 1024, 2048, 4096$. Then, using the halt values, we generated a line of best fit on a log-log plot to find our general $k(n)$ for all values of n. Our plot looks like this:



Our data for the following values of n is as follows:

n	0-dim Halt
32	0.062500
64	0.0312500
128	0.015625
256	0.007813
512	0.003906
1024	0.001953
2048	0.000977
4096	0.000488

2.3 Random Weight Generation

We initially implemented random number generation with Rust's rand crate which uses the ChaCha20 algorithm under the hood. Although rand is extremely efficient in a vacuum, it slowed down our algorithm dramatically. Instead, we opted to lazily generate the sorted list of edges for Kruskal's by sampling the exponential distribution. Recall that the exponential distribution is generally used to measure the expected time for an event to occur. Here, we use the distribution to approximate the next edge weight for our MST algorithm. The exponential distribution is memoryless, so we increment our weight at each iteration step to approximate the next element of our sorting algorithm. Since we have $\binom{n}{2}$ edges in our complete graph, our random variable is distributed as:

$$X \sim Expo\left(\lambda = \binom{n}{2}\right)$$

Note that the exponential distribution works best for large n. While our error rate is negligible for large n, our error rate is more pronounced for smaller n. An improved version of our algorithm would utilize an alternate implementation for smaller n.

2.4 Results

We integrated both the graph sparsification and random weight generation steps of our algorithm into our final algorithm. Our algorithm ran on the $n = 262144$ case in approximately 100ms, demonstrating blazing fast performance. Moreover, the total edge weight of our MST converged to approximately 1.202 which is consistent with Apery's constant. The results for our trials are as follows:

n	MST Weight	Trials	Dimension
2^5	1.2947	8	0
2^6	1.3821	8	0
2^7	1.1755	8	0
2^8	1.2195	8	0
2^9	1.2259	8	0
2^{10}	1.2153	8	0
2^{11}	1.2215	8	0
2^{12}	1.1979	8	0
2^{13}	1.1962	8	0
2^{14}	1.2083	8	0
2^{15}	1.2058	8	0
2^{16}	1.2009	8	0
2^{17}	1.2035	8	0
2^{18}	1.2019	8	0

Since our algorithm appears to converge at an average edge weight of 1.202 across all 8 trials for large n, our asymptotic bound for the expected edge weight as a function of n is approximately $f_0 = \Theta(1.202)$.

3. 2, 3, 4-dimensional Random Euclidean Graphs

Much like the 0-dimensional random complete graphs, the naïve approach to compute the average weight of the MST for the 2,3,4-dimensional random Euclidean graphs runs into similar issues with generating and storing the graph. For the Euclidean graphs, we utilize a similar approach to the 0-dimensional case but modify the filtering function and eliminate the exponential sampling component of the 0-dimensional algorithm.

3.1 Sparsifying Euclidean Graphs

The filtering function we used for our 2, 3, 4-dimensional algorithm was:

$$k(n) = \frac{2}{n^{\frac{1}{d}}}$$

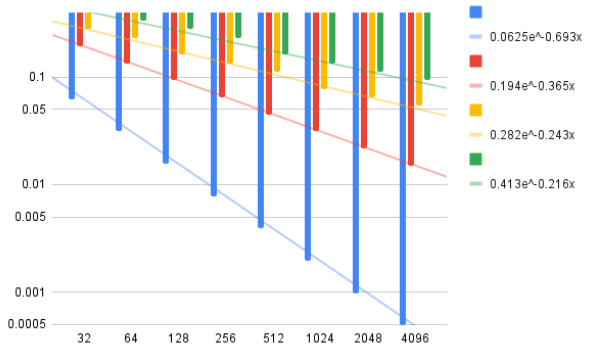
.

We determined $k(n)$ with a similar approach to the filter halving function used in the 0-dimensional case. Note that with the filtering function for the 2, 3, 4-dimensional cases, we can derive a more general

form for $k(n)$ as follows:

$$k(n) = 2n^{-\frac{1}{max(d,2)}}$$

To confirm our hypothesis for the general form of $k(n)$, we combined all the halt values for the 0, 2, 3, 4-dimensional cases into a single log-log plot which looks like this:



We can clearly see that the filtering functions for all d dimensions are asymptotically similar. This is also evident when we look at the halt values across dimensions as a function of n.

n	2-dim Halt	3-dim Halt	4-dim Halt
32	0.193807	0.279082	0.401878
64	0.062500	0.232568	0.334898
128	0.093464	0.161506	0.279082
256	0.064905	0.134588	0.232568
512	0.045073	0.112157	0.161506
1024	0.031301	0.077887	0.134588
2048	0.021737	0.064905	0.112157
4096	0.015095	0.054088	0.093464

3.2 Results

Unlike our 0-dimensional algorithm, we did not include any special optimizations for random weight generation, sticking to the Rust rand crate described above. The exponential distribution implementation only works for the 0-dimensional case since the algorithm only generates edge weights without any regard for the position of vertices relative to one another. As expected, computing the MSTs for the 2,3,4-dimensional cases was far more time-intensive than the 0-dimensional case, taking approximately 5 minutes to run the $n = 262144$ 4-dimensional case. The results for our the trials with $n = 2^{12}$ and greater is as follows:

n	MST Weight	Trials	Dimension
2^{12}	41.7678	8	2
2^{12}	169.1725	8	3
2^{12}	361.7226	8	4
2^{13}	58.9445	8	2
2^{13}	267.6399	8	3
2^{13}	603.6280	8	4
2^{14}	83.1974	8	2
2^{14}	423.2945	8	3
2^{14}	1009.2346	8	4
2^{15}	117.4428	8	2
2^{15}	669.0943	8	3
2^{15}	1689.2637	8	4
2^{16}	165.9434	8	2
2^{16}	1058.6761	8	3
2^{16}	2828.8566	8	4
2^{17}	234.6769	8	2
2^{17}	1677.2294	8	3
2^{17}	4738.7873	8	4
2^{18}	331.6701	8	2
2^{18}	2657.9564	8	3
2^{18}	7950.5301	8	4

Plotting the data and examining the asymptotic behavior yields a friendly function in terms of d dimensions and n vertices for the expected weight of our MST:

$$f_d(n) \in \Theta(n^{\frac{d-1}{d}})$$

This asymptotic behavior also aligns perfectly with the graphical representation shown in section 3.1.

4. Performance

In addition to the algorithmic optimizations we implemented in the above sections, the choice of Rust as a programming language made it easy to improve our program further for ultra-fast performance.

4.1 Multithreading

Multithreading is acheivied relatively simply via the `prelude` module of the `rayon` crate (external library). By including `use rayon::prelude::*;` at

the top of `main.rs`, we enable the `.into_par_iter` function. This function allows us to iterate through the range `0..args.num_trials` in a parallelized manner, where `args.num_trials` is the third command line parameter `numtrials`. Thus, we can utlize multiple threads to run multiple trials simultaneously, and then average them once they're all done.

Multithreading offers significant performance benefits on multicore machines, but not in every metric. To evaluate the benefits offered by multithreading we compare and contrast the performance of identical versions of our program, but where one version swaps the `.into_par_iter` function for `.into_iter`. This change effecitvely forces one version of the program to be single-threaded. To evaluate the performance of each version of the program, we utilize the Unix command `time sh ./run.sh`.

Runtime Statistics				
Version	`user`	`sys`	`real`	% CPU
Single	1339.68s	8.80s	22:32.20	99
Multi	1662.11s	6.11s	3:32.90	783

Tests were run sequentially on an Apple M1 Pro system-in-package, containing a 10-core CPU with 8 performance cores and 2 efficiency cores. This context explains the 783% CPU usage of the multi-threaded program. It is likely that the system performed some CPU-time conservations, such as keeping the efficiency cores free for processing user inputs and other basic tasks. Thus the multi-threaded program mostly scaled across the M1 Pro's 8 performance cores.

An interesting result is the 24% longer `user` time of the multi-threaded program. This suggests that significant overhead is present in the multithreading implementation, or perhaps that Unix `time` is including the time usage of hanging threads that are waiting for the final trial to complete. As expected, the `real` time of the multithreaded program is 15.5% of the singlethreaded program, which is far faster and within the bounds of expectation given our machine's core count.

4.2 Programming Challenges

Implementing the algorithm proved to be more of a syntactic challenge rather than a mathematical or logical one. Neither Artemas nor I had experience with Rust or its quirks, which increased the amount of time it took to implement the algorithm and data structures. We learned advanced Rust features such as interior mutability, generics, and traits by utilizing helpful resources such as *The Rust Programming Language* by Steve Klabnik and Carol Nichols and our friend and Rust expert Lev Kruglyak.

5. Conclusion

For our 0-dimensional case, the expected weight of the MST is:

$$f_0 = \Theta(1.202)$$

.

For the d-dimensional case (d = 2,3,4), the expected weight of the MST is:

$$f_d(n) \in \Theta(n^{\frac{d-1}{d}})$$

Overall, the inclusion of the filtering function $k(n)$ in both the 0-dimensional case and the 2,3,4-dimensional cases proved crucial in increasing the efficiency of our algorithm. For the 0-dimensional case, the lazy generation of sorted edges yielded even more incredible results.