CS 124 Programming Assignment 1: Spring 2022

Your name: Tram B. Nham

Collaborators:

No. of late days used on previous psets: 12
No. of late days used after including this pset: 16 (plus extra 7 late days due to medical emergency)

1 Results

Table 1: Average MST Tree Size By n

n	$0D$ size	$2D$ size	$3D$ size	$4D$ size
128	1.12188	7.45479	17.8173	28.4138
256	1.17835	10.6516	27.5811	46.7459
512	1.16656	14.8441	43.4598	78.253
1024	1.18559	21.0072	68.5154	129.441
2048	1.19741	29.6467	107.328	216.322
4096	1.1938	41.8027	168.942	360.906
8192	1.20043	58.8345	267.128	604.165
16384	1.20044	83.3004	422.465	1009
32768	1.2022	117.367	668.745	1689.79
65536	1.19847	166.068	1059.24	2830.38
131072	1.20249	234.697	1676.81	4743.73

Table 1 above shows the average MST tree size for $n = 2^7, 2^8, ..., 2^{17}$. For all four types of graph, the average tree size grows faster at the beginning, then gradually tapers out as n increases. However, their growth rates differ ($0D$ to $4D$ in the order of slowest to fastest growth) and I estimate each function as follows:

$$f(n)_{0D} = (\log(\log n))^{0.13}$$
$$f(n)_{2D} = 0.55 \cdot n^{0.52}$$
$$f(n)_{3D} = 0.71 \cdot n^{0.66}$$
$$f(n)_{4D} = 0.84 \cdot n^{0.73}$$

For the $0D$ graph, since the average tree size grows very slow even as n gets very large, and I found that $f(n) = o(\log(\log n))$, I estimate $f(n) = \log(\log n))^c$ for some $0 < c < 1$.

For the other three graphs, I found that $f(n) = \omega((\log n)^c)$ for $c > 1$, and $f(n) = o(n)$. Therefore, I estimate $f(n) = k \cdot n^c$ for some $k > 0$ and for some $0 < c < 1$. I adjust k and c to accomodate different growth rates for each graph type, increasing both constants for the tree type that grows faster.

2 Discussion

I used Prim's algorithm with a binary heap to find the MST tree size. I think this is better for runtime than Kruskal's algorithm because Kruskal's requires sorting the edges, which takes $O(|E|log|E|)$ time. Since our graphs are complete, we have $\frac{n(n-1)}{2}$ edges, and thus sorting alone already takes $O(n^2 \log n)$ time, plus $O(n^2 \log^* n)$ to run Union-Find operations. Meanwhile, Prim's takes $O(n^2 \log n)$ time to run with a binary heap. I also implemented a version with Kruskal's algorithm and it was indeed significantly slower than Prim's. However, as I was thinking about how to write up this discussion, I realized that it would have been better if I had used linked list instead of binary heap, since linked list would only take $O(n^2)$. This is because our graphs are very dense and the Insert operation using binary heap is more expensive ($O(\log n)$ vs. $O(1)$). I was not able to run the largest $n = 262144$ as required even after edge pruning, and I suspect this is the main reason. Unfortunately, it was too late for me to go back and change my code, so the following analysis is based on my implementation of the Prim's algorithm using binary heap.

I find the growth rates to be reasonable. Since edge weights grow faster as n increases in higher-dimensional graphs as opposed to lower-dimensional graphs, their tree sizes also grow faster, and the slope of $f(n)$ is steeper in $4D$, followed by $3D$, $2D$, and OD. Within each graph type, however, tree size growth slows down as n increases since a MST only contains exactly $n-1$ edges. Thus, even when the total number of edges in each graph increases significantly as n increases ($O(n^2)$), the MST only uses the $n-1$ smallest-weighted edges that connect all n vertices in the graph. This is also why we can throw away larger-weighted edges to improve runtime without affecting MST size.

The time it takes my algorithm to run varies from roughly 10 milliseconds to 2.5 hours, depending on input size n and graph type. Higher n and higher-dimensional graphs take longer to run. Before edge-pruning, it takes $O(n^2 \log n)$ as expected. After edge pruning, runtime is cut down significantly, but more so for $0D$ and $2D$ graphs than for $3D$ and $4D$. I estimated maximum edge weight needed for a MST as a function $k(n)$ and threw away any edges of weight larger than $k(n)$:

$$k(n)_{0D} = \frac{1}{n^{0.59}}$$
$$k(n)_{2D} = \frac{1}{(\log n^{0.25})^{2.5}}$$
$$k(n)_{3D} = \frac{1}{\log n - 1.6\log(\log n)}$$
$$k(n)_{4D} = \frac{1}{\log(\sqrt{\frac{n}{12}})}$$

I used the same strategy for estimating $f(n)$ as described above to estimate $k(n)$. I printed out the maximum weight used before edge pruning for several smaller values of n that my computer can handle in reasonable amount of time (from 128 to 16384), estimated upper bounds and lower bounds, and finally adjusted the constants. When I tried some other $k'(n) = o(k(n))$, the algorithm returned average MST size $= \infty$. This is expected since if we throw away too many edges, our graph could become disconnected and the distances from the root node to some other nodes are set to ∞. My estimate of $k(n)$ is fairly conservative to make sure the output is correct, and I have compared results before and after edge pruning using the same seed for the random number generator to generate the same graph every time. (Note that this is for testing only and the code I used to output the results in *Table 1* generates a random positive integer as the seed for the random real uniform distribution generator for every trial.) However, even when I tested a smallest k possible that still produces the correct output, the runtime did not improve much for larger n. Therefore, I think I could only improve runtime if I improve my Prim's algorithm by using a linked list rather than a binary heap. (I also think my code can be improved slightly since this is my first time coding in C++ and I'm not familiar with all the special data structures such as vectors and pointers). I also attempted to run $n = 262144$ overnight, and although my 8GB RAM computer had not crashed, I decided to interrupt the program since it could possibly take as much as 30 hours for 5 trials of the $4D$ graph based on the runtime growth rate that I observed with smaller values of n.

The results I found clearly demonstrate the properties of minimum spanning trees and graph algorithms. An important lesson I learned is that, not only does the algorithm matter, the types of data structures we choose to implement our algorithm on also affect the performance of our program a lot. We should not use the same algorithm for all types of graphs, because even when a binary heap is better than linked list in a lot of cases (when graphs are sparse), we should carefully examine the graph attributes to decide which data structure and which algorithm are optimal in our case.