CS 124 Programming Assignment 1: Spring 2023

Your name(s) (up to two): Michael Xiang, Marcos Johnson-Noya
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:
No. of late days used after including this pset:

Programming Set 1 Report:

Overview

We created an algorithm to generate complete graphs, and we implemented Prim's algorithm in order to find the MST's of our randomly generated graphs. Our algorithms were implemented in C++.

Data

Table 1: Table of average tree lengths

n	**128**	**256**	**512**	**1024**
0	1.24842	1.22573	1.16541	1.19445
2	7.78727	10.968	14.9287	21.060722
3	17.2477	27.692	42.2752	67.6973
4	28.5713	47.7896	78.2618	129.546
n	**2048**	**4096**	**8192**	**16384**
0	1.201239	1.259857	1.278965	1.941678
2	29.723211	41.791049	59.015266	83.008856
3	106.97	168.948	267.422	422.731
4	216.886	361.097	602.042	1009.18
n	**32768**	**65536**	**131072**	**262144**
0	1.221838	1.232282	1.21032	1.20047
2	117.9832	166.00305	234.454527	331.980012
3	667.989	1058.834419	1678.911324	2658.88889
4	1689.557373	2828.500532	4740.3256	7949.421

Analysis of Growth Rates

We get that average tree length for dimension 0 is approximately 1.2. For other dimensions, the function is diffierent. Using this table it seems that the average tree lengths grow exponentially as a function of $\log(n)$. Thus, it seems that the average tree length is multiple of the average tree length for the previous value n. Or rather, it seems that the function is of the form $f(n) = AB^{\log(n)-7}$. Thus, for each dimension we can use exponential regression to find $f(n)$. After using the my TI-84CE ExpReg regression calculator, we can get the functions for dimensions 2, 3, and 4.Altogether we get the following functions modeling the growth rate of average tree lengths as a function of n.

$$f_0(n) \approx 1.2$$
$$f_2(n) = 7.64498648 \cdot 1.407414916^{(\log_2(n)-7)}$$
$$f_3(n) = 17.18877365 \cdot 1.580608541^{(\log_2(n)-7)}$$
$$f_4(n) = 28.21291422 \cdot 1.668223853^{(\log_2(n)-7)}$$

The first function certainly was a surprise. Although, the number of edges grows, it seems that fact is offset by the fact that the average distance between the nodes gets smaller. If we denote the number of nodes as n and k as the average distance between nodes then we get that $n\dot{k} \approx 1.2$. The functions which represent dimensions $2, 3$, and 4 make sense. The functions which model the average length of the MST with higher dimensions increase more quickly.

Analysis of Our Algorithm

In our programming assignment, we implemented Prim's algorithm with a heap. We used Prim's because we thought that our space usage would be a big issue throughout the assignment, given that our MST algorithm needed to handle at least 262144 vertices. Since Prim's algorithm handles denser graphs better than Kruskal's algorithm ($E\log(V)$ is faster than $E\log(E)$ when $E = |V|^2$), and the graphs that we randomly generated were complete graphs, we thought that Prim's would run faster as $n \to \infty$. We did consider the effects of edge-pruning on the density of the graph, but we ultimately thought that, after edge-pruning, the graph would still be relatively dense. Thus, we went with Prim's algorithm.
The way we implemented Prim's algorithm mirrors the pseudocode from the class. We used a binary heap, so, according to the table from class, our algorithm must run in $O(|E|\log(|V|))$ time. But we did modify the algorithm by creating a "visited" array that keeps track of whether a vertex has already been accounted for within our MST or not. The reason we created this table is that we wanted a way to check whether we "over-pruned" or not, and if some vertex is not in our MST, then our algorithm over-pruned and disconnected the graph. We marked a vertex as visited if we popped it from the heap during our algorithm, which takes $O(1)$ time. In the end, when the heap no longer contains any elements, we scan linearly through the "visited" array to check whether there is some vertex that we have not visited yet. If there is an element that has not been visited, that means our algorithm did not yield an MST. This takes $O(|V|)$ time. Thus, we need to add this linear scan to our runtime, so our total runtime is

$$O(|V| + |E|\log(|V|))$$

From class, we have already proved correctness, by showing that the cut property is maintained with our greedy approach. This is clear, because from the cut property, we know that we can construct an MST with a set of vertices $X \subseteq T$, where T is an MST, if we add an edge e such that e is the minimum-weight edge from a vertex $\in X$ and a vertex $\in V(G) \setminus X$. In other words, $X \cup e \subseteq T$. And since we are using a heap to extract the minimum distance element, just like we did in class, our algorithm is correct because the heap will extract the minimum weight edge from a vertex within our growing MST X to a vertex $\in V(G) \setminus X$. The additional work required to check whether all vertices are accounted for within our MST does not actually affect how we calculate the weight of the MST and the edges we use to construct our MST, so our algorithm is effectively unmodified from class and is thus correct.

Edge Pruning

In order to keep our program running relatively fast and save memory, we chose to discard edges of weight larger than $k(n)$ as we increase the input size. In our program, we use the following code to implement this idea of pruning large edges:

```
if (g[i].size() == 2)
        {
          w = (float)1 * pow((0.91105165234), log2(g.size()) - 7);
        }
        if (g[i].size() == 3)
        {
          w = (float)1 * pow((0.83266771883), log2(g.size()) - 7);
        }
        if (g[i].size() == 4)
        {
          w = (float)1 * pow((0.79943993619), log2(g.size()) - 7);
        }
```

We do not prune edges when dimension = 0.What we're doing here is depending on the value of n we set a variable w to be the maximum weight of an edge in the graph g. Since from our $f_i(n)$ function we know that the average length of the graph is approximately k times bigger every time we double the value of n, where the value of k can be one of $1.407414916, 1.580608541, 1.668223853$ depending on the dimension of g. Thus, we can expect that the average length of each edge in a graph is approximately k^{-1} times smaller as the value of n doubles. The values of k^{-1} are $0.71052252511, 0.63266771883, 0.59943993619$. However, we know that often times, the average length of each edge on a given MST can be several standard deviations from the mean. Hence, we add 0.2 to each k^{-1}. Then, as n gets bigger we will cap the largest edge length to be $0.2 + k^{-1}$ times smaller than the previous value of n. We can say that this is safe way that still ensures we do not over prune the adjacency list because $0.2 + k^{-1}$ is several standard deviations away from the average edge length in the MST. So, we define $k(n) = (0.2 + k^{-1})^{\log(n-7)}$. This, value of $k(n)$ ensures that we don't overprune but helps with speed.