CS124: Programming Assignment 1

Julian Lee & Daniel Nguyen

February 28, 2020

1 Data & *f(n)* prediction

Table 1: Average MST Weights for n vertices

	Random Edge Weight	**2D**	**3D**	**4D**
n = *128*	1.196	7.69875	17.8329	28.4063
n = *256*	1.193	10.7314	27.7367	47.1245
n = *512*	1.196	15.0681	43.3291	77.4787
n = *1024*	1.207	20.9651	68.5996	129.511
n = *2048*	1.205	29.5796	106.792	216.425
n = *4096*	1.200	41.7839	168.958	361.772
n = *8192*	1.201	58.8899	267.281	603.001
n = *16384*	1.198	83.1787	422.378	1008.650
n = *32768*	1.208	117.339	2668.863	1689.11
n = *65536*	1.204	165.826	1056.83	2829.6
n = *131072*	1.202	234.481	1704.869	4742.045
n = *262144*	1.204	331.982	2699.86	7945.18

In general, we can observe that for a graph with n randomly generated vertices in d dimensions such that the edge weights are the Euclidean distances between the vertices, the average weight of the MST approaches $\boxed{f(n) = 0.66 * n^{\frac{d-1}{d}}}$. The coefficient for the $n^{\frac{d-1}{d}}$ giving the best fit for our empirical data has varied slightly between the dimensions - 0.69 for $d = 4$, 0.66 for $d = 3$, and 0.65 for $d = 2$.

For the case in which the weights for the edges were randomly assigned between $(0, 1)$, we made the counterintuitive observation that $f(n) \approx \boxed{1.2}$ - in other words, the average weight of the MST is independent of the number of nodes in the graph.

2 Discussion

We implemented Prim's algorithm using doubly linked lists for this assignment. The reason for this was because Prim's algorithm is known to run faster on dense graphs, while Kruskal's algorithm is known to run faster on sparse graphs. Since we deal with complete graphs in the assignment, which are the ultimate form of dense graphs, we thought that Prim's would be more time efficient. However, we ran into some minor issues with runtime; a single trial of $n = 262144$ took approximately 3 hours. We have reasonably strong evidence that our time complexity is still linear w.r.t to the graph size and specifically is $O(|E|)$, since doubling the number of vertices consistently increased the runtime by a factor of 4, and we know that $|E| \approx 0.5|V|^2$. We conjecture that perhaps our while loop executes too many elementary operations, or that running the algorithm on a virtual machine slowed it down.

One especially surprising aspect of the growth rate was the fact that $f(n)$ remains constant at 1.2 even if n gets arbitrarily large for the set of graphs with randomly weighted edges. At first, this seems counterintuitive, but we realize that the number of edges in the graph grows on the order of n^2. Given the uniform distribution of edge weights over the interval $(0, 1)$, the likely explanation for the constant trendline, then, is that the increase in the number of low-weighted candidate edges for the MST exactly cancels out the increase in the number of edges contained within MST as n grows.

We can give somewhat of a similar explanation for the function $f(n)$ for two, three, and four dimensions. We observe first of all that the growth rate is sublinear with respect to n for all values of d, which is again explained by the fact that as n grows, the density of the points increases and thus the average length of an edge decreases. However, why does the average MST weight not remain constant, as we observe in the 0-dimension case? While not a rigorous mathematical proof, let us set the threshold of edge weight ≤ 0.1 as the definition of a low-weighted edge. For zero dimensions, no matter the value of n, the proportion of low-weighted edges is always $0.1n$. However, as the dimensionality of our space grows, the maximum possible distance between points grows as well; in 2 dimensions, it is $\sqrt{2}$, in 3 dimensions $\sqrt{3}$, and so forth. Thus, if we assume a uniform distribution of distances, the proportion of low-weighted edges actually decreases with the function $\frac{0.1}{\sqrt{d}}$ as the dimensionality d increases, since there is more space for the points to occupy. This is the reason for the gradual increase in the MST weight.

The experience with the random number generator was quite interesting, as we had learned about pseudo-random generators in theory from CS121, but this was our first time implementing a program which required their use. It was particularly notable to observe that the Mersenne Twister engine which we used required a seed from the system's clock - thus we see that the number produced by the generator is not truly random, as the Mersenne Twister takes a random seed and amplifies this into a longer number which appears to be random. However, there are reasons to be somewhat skeptical about the extent to which our random number generating function truly achieves the effect of randomness, as the seed is derived from the operating system's clock. Perhaps if we were to sample two seeds in close temporal proximity, the seeds would be identical and the Mersenne Twister would output the same number.

Instead of calculating all edge weights first and throwing out all edges with weight higher than some threshold, we attempted to optimize the space complexity of our algorithm by only calculating edge weights when they were needed. In one iteration of the while loop of our implementation of Prim's, we iterated through every pair of vertices (v, w) s.t. $w \in V - S$ and v was the vertex just popped off the heap. Then, we calculated the edge weight of (v, w), and stored this in $dist[w]$ if it was lower than the existing entry. Otherwise, the edge weight was simply forgotten. Therefore, the space complexity of our algorithm was simply $O(|V|)$ instead of $O(|V|^2)$. This is guaranteed to return the same tree as if we stored all edge weights in an adjacency matrix, since our implementation does store all vertex coordinates - thus the edge weight can simply be recalculated with the same result if we need it again.