Programming Assignment 1 Report

Vanessa Chambers and Britteny Okorom-Achuonye*

(Dated: 29 February 2020)

OUR PROGRAM

***Note: our program compiles with the command "make".*

We decided on a Prim's algorithm implementation for our program. Prim's algorithm greedily builds a single tree, adding the shortest edge to the MST one at a time–this quality of prim's algorithm allowed us to calculate edge lengths adjacent to the node being explored, thus eliminating the need to store all edges at once (which saves memory). Originally, we attempted to represent a graph using an adjacency matrix, however, this implementation results in an n x n matrix, redundantly storing non existent edges–i.e. the value at position [0][0], which represents a nonexistent edge between node zero and itself–and all real edge values, twice. Because of this wasted space, which would also amplify the running-time, we decided against this implementation and instead chose to represent our graph as a list of vertices, represented by Point structs. Each Point stores x, y, z, and a values, a weight, and a boolean value "visited". Using this data structure, we did not have to calculate and store all $\binom{n}{2}$ edges at the beginning and could instead calculate them as we determined the MST. Because of this, we could immediately discard calculations of edge lengths that were larger than the already stored "weight" values of the Points (this is also why we did not need to utilize the hint, because edges considered too large were automatically thrown out under our implementation).

Additionally, the algorithm is set up so that no specific edge value is calculated more than once, avoiding wasting time recalculating the Euclidean distance between two vertices that had already been calculated; this saved some calculation time in the overall running-time. However one trade-off of our implementation was that because we did not use a complex data structure, such as a linked list or heap, the algorithm had to iterate through all n Points at every turn. Although no mathematical operations were done on Points with visited = true, nonetheless, these points were still iterated through. This is still a waste of time because rather than discarding a node that had already been visited, the algorithm uselessly checks it, gaining no new information. To improve the design of our program (omitting the need to write redundant code), we used the same algorithm, "prim" and the same Point data structure to calculate MST weights in all dimensions. This stylistic choice required us to store unnecessary coordinate values for some points–i.e. setting $x, y, z, a = 0$ for points in

* CS124, Harvard University.

the zero dimension–which wasted 4 bytes per unnecessary float stored.

ANALYSIS OF $f(n)$

For graphs in the **0 dimension**, the average MST weight, $f(n)$ converged at a value of ≈ 1.2.

$$f(n) = 1.2020569\ldots \tag{1}$$

This is particularly interesting, because it shows that the MST weight seems to approach the Apéry's constant, ≈ 1.20205. A possible explanation for this could be the fact that for a graph with zero dimensions, as we randomly determine different edge lengths, as n increases, the edge lengths are still bounded within the same interval [0,1] so the possible MST has to be bounded by some constant. Also after researching more about Apéry's constant, we discovered that [Apéry's constant] has been proven to be the bound of a randomly generated MST as n, the number of nodes, goes to infinity (1).[1]

After plotting the results of the MSTs in the 2^{nd}, 3^{rd}, and 4^{th} dimensions, we noticed, quantitatively, that these plots displayed logarithmic behavior (Figures 1, 2, 3, 4). Thus, we used this observation to guess a form for the curve that fits $f(n)$ for these higher dimensions:

$$a^{log_2 n} \cdot (b) - c \tag{2}$$

where a, b, and c are real numbers, and n is the number of vertices in the complete graph. We then tested several different values of a, b and c in this general form in order to best fit the data and derived these equations:

Two Dimensions

$$f(n) = \left(\frac{16.86}{12}\right)^{log_2 n} \cdot 0.725 - 0.5 \tag{3}$$

Three Dimensions

$$f(n) = \left(\frac{3.1552}{2}\right)^{log_2 n} \cdot 0.725 - 0.725 \tag{4}$$

Four Dimensions

$$f(n) = \left(\frac{2.0}{1.1927}\right)^{log_2 n} \cdot 0.725 - 0.725 \tag{5}$$

It is interesting to note that in the second dimension, the value a is 1.405, which is close to the value of $\sqrt{2}$, the maximum Euclidean distance between two vertices in 2D; in 3D, a is 1.5776, which is (somewhat) close to $\sqrt{3}$ (max Euclidean distance in 3D graph); in 4D, a takes the

value of 1.677, which also approaches 2 (max distance in 4D). This observation shows that a more precise f(n) function could depend in some way on the maximum Euclidean distance between two vertices in a given dimension, as well as the number of nodes.

ANALYSIS OF RUNTIME

The time complexity of our program seems to be approximately $O((n^2)*$ (DISTANCE FUNCTION RUNNING TIME)). We get the n^2 from the fact that our algorithm has for loop, which iterates through all n nodes nested within a while loop that iterates through all $n-1$ edges. The distance function calculates the euclidean distance between points during (almost) each turn through the for loop–we say almost because distance is not recalculated for Points with visited = true. The specific running-time of the distance function depends on the time complexity of the C++ $pow()$ and $sqrt()$ function, which are both $O(logn)$, where n is the value of the number in the base. Thus, the time complexity of our program is $O((nlogn)^2)$. This program works fast for small n, but, evidenced by the time complexity, large n in some dimensions has a very slow running time, possibly spending more than an hour to run (Table 1, 2, 3, 4).

ANALYSIS OF RANDOM NUMBER GENERATOR

To generate random values, we used the $rand()$ function from the std library to generate a list of random numbers, and $srand()$ to reseed the list. $srand()$ is called once in the main function, and $srand$ reseeds according to my computer's internal clock– this is useful because the time constantly changes, so calls to the rand() function should constantly change, as well. However, it is worth mentioning that because $srand()$'s reseeding method is a pseudo random number generator, and it could be unreliable in the case that the time used to reseed happens to be the same during separate runs. Despite its drawbacks, this random number generator was sufficient for our purposes, and, in addition, we performed five trials and took the average over these trials to make up for any unreliability in the random number generating process.

FIGURES AND TABLES

dimension	nodes	avg runtime (s)	avg weight
0d	128	0	1.20545
	256	0	1.20976
	512	0	1.17719
	1024	0	1.18859
	2048	0	1.17845
	4096	0.2	1.20148
	8192	0.6	1.20441
	16384	2	1.20304
	32768	9	1.20083
	65536	51	1.20225
	131072	172	1.20116
	262144	594	1.20129

Table 1: Zero Dimension Results

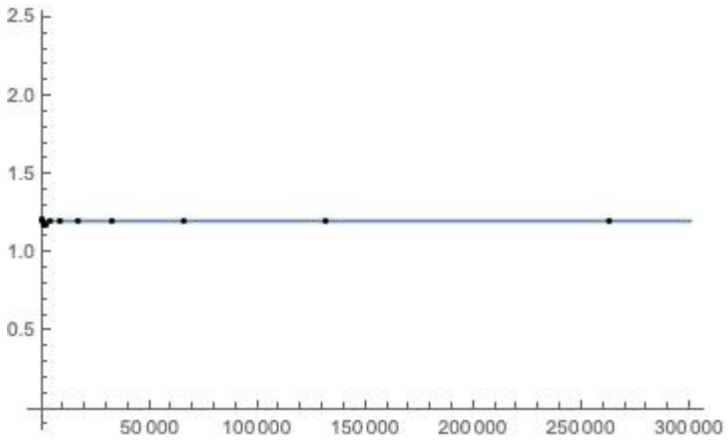

FIG. 1. Plot of n vs MST weight for 0D graph.

dimension	nodes	avg runtime (s)	avg weight
2d	128	0	7.57055
	256	0	10.7167
	512	0	15.0981
	1024	0.02	21.0522
	2048	0.2	29.6389
	4096	0.8	41.7306
	8192	3.6	58.9656
	16384	14.6	83.3139
	32768	58	117.429
	65536	239.6	166.047
	131072	981.6	234.813
	262144	4957.6	331.488

Table 1: Two Dimension Results

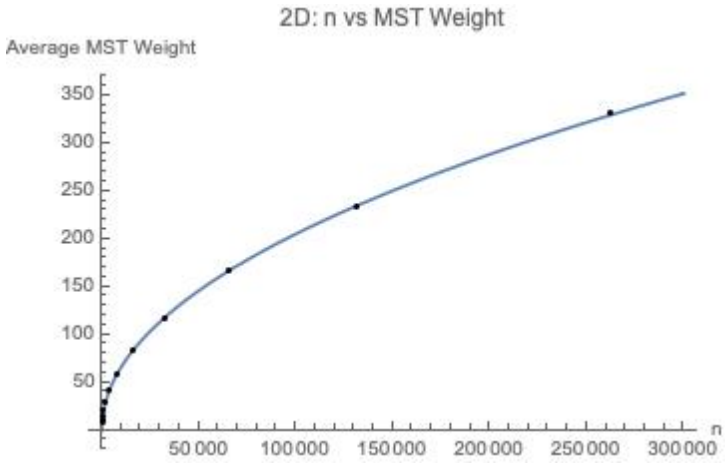

FIG. 2. Plot of n vs MST weight for a 2D graph.

dimension	nodes	avg runtime (s)	avg weight
3d	128	0	17.5275
	256	0	27.1154
	512	0	43.5512
	1024	0	67.9326
	2048	0.2	107.49
	4096	1	169.278
	8192	4.6	267.053
	16384	17.4	422.581
	32768	69.6	668.096
	65536	281.6	1059
	131072	1606.2	1677.71
	262144	5670	2657.87

Table 1: Three Dimension Results

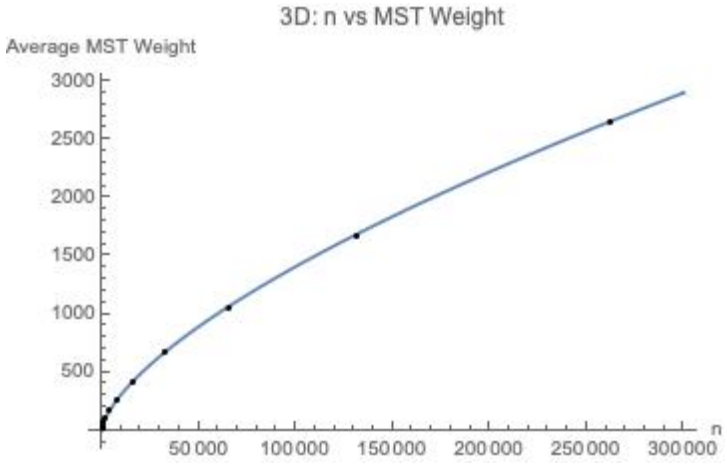

FIG. 3. Plot of n vs MST weight for a 3D graph.

dimension	nodes	avg runtime (s)	avg weight
4d	128	0	28.4664
	256	0	47.5355
	512	0.2	78.7056
	1024	0.2	130.29
	2048	0.4	216.987
	4096	1.2	361.283
	8192	5.2	602.767
	16384	21	1007.6
	32768	81.4	1690.65
	65536	326.2	2830.09
	131072	1807.2	4737.94
	262144	6001	7946.97

Table 1: Four Dimension Results

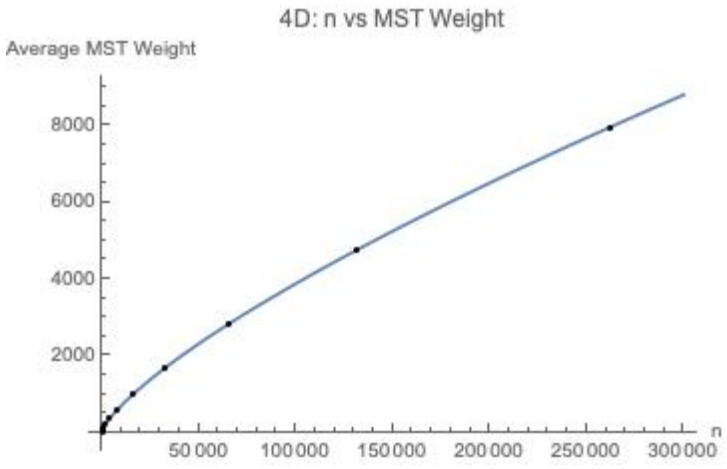

FIG. 4. Plot of n vs MST weight for a 4D graph.

[1] Frieze, A. (1985). On the value of a random minimum spanning tree problem. Discrete Applied Mathematics, 10(1), 47-56.