CS124 Programming Assignment 1

Amanda Stetz and Rucha Joshi

February 2021

No. of late days used on previous psets: 3
No. of late days used after including this pset: 5

1 Introduction

In this paper we explore the the behavior of Minimum Spanning Trees as we vary the number and dimension of vertices. We implemented Kruskal's Algorithm in C++ on adjacency list representations of undirected complete graphs, and below have provided a table and discussion of our results.

2 Results

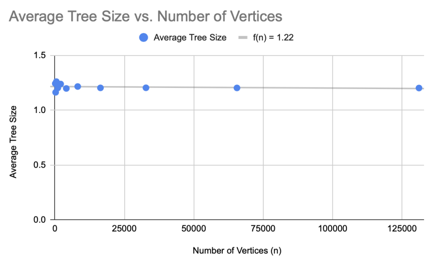
Below we can see the average tree size of a Minimum Spanning Tree when we vary the number of vertices (n) and the dimension of the graph. We chose n vertices, where vertices are points chosen randomly within a unit (0 dimensions), unit square (2 dimensions), unit cube (3 dimensions) and hypercube (4 dimensions).

Number of Vertices (n)	Dimensions	Average Tree Size
	0	1.184366
128	2	7.51193
	3	17.44916
	4	28.29784
	0	1.1626422
256	2	10.54236
	3	26.90776
	4	45.08354
	0	1.260306
512	2	14.86526
	3	43.49472
	4	78.92666

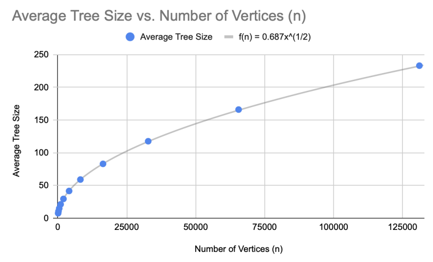
Number of Vertices (n)	Dimensions	Average Tree Size
1024	0	1.20274
	2	21.128
	3	69.2128
	4	129.843
2048	0	1.23865
	2	29.4516
	3	107.804
	4	217.893
4096	0	1.19816
	2	41.5472
	3	169.424
	4	361.831
8192	0	1.2159
	2	59.0314
	3	266.354
	4	605.33
16384	0	1.2034
	2	82.9967
	3	422.687
	4	1009.06
32768	0	1.2041
	2	117.5972
	3	667.755
	4	1688.65
65536	0	1.202927
	2	166.0208
	3	1058.974
	4	2829.18
131072	0	1.2021
	2	233.1216
	3	1643.941
	4	4674.99
262144	0	N/A
	2	N/A
	3	N/A
	4	N/A

3 Implementation

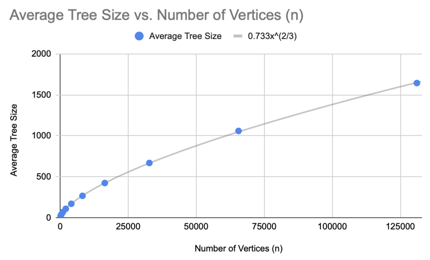
We began by creating an Edges class which consisted of vectors of edges in the graph and MST, helper functions in Kruskal's algorithm, and of course Kruskal's algorithm itself. This helped us in constructing the different graphs for each trial. Our random number generator generates a random number and fits it on a Uniform distribution from $[0, 1]$. If the user input a dimension of 0



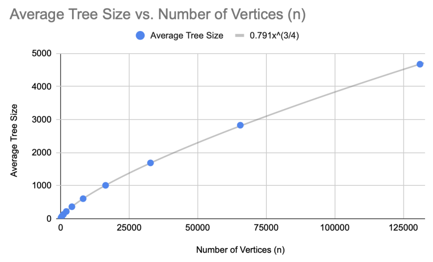
(a) Dimension 0



(b) Dimension 2



(c) Dimension 3



(d) Dimension 4

we directly create the adjacency list, discarding edges if they are greater than $k(n) = 2^{-x*\log(n)}$, an equation we experimentally derived through running tests on smaller sized graphs to determine a relationship between the maximum sized edge and the dimension of the graph. We then experimented with various values of x for each dimension. For example, in the case of 0 dimensions $x = 0.6$.

If the dimension is greater than 0, we create a $n \times d$ 2D array where n is the number of nodes and d is the number of dimensions. The edge weight is determined by finding the distance between vertices whose points are randomly generated within a unit. We run Kruskal's algorithm and determine the weight of the minimum spanning tree.

We have plotted our values of the edge weights vs n above.

Our first goal was to determine the expected weight of the minimum spanning tree as it grows as a function of n. After plotting the empirical values by n we believed that a polynomial equation of the form $a \cdot x^b$ would best fit the values we had. After fitting multiple curves on the graphs, we decided the following equations as the line of best fit for each dimension.

Dimension	$f(n)$
0	$f(n) = 1.22$
2	$f(n) = 0.687x^{\frac{1}{2}}$
3	$f(n) = 0.733x^{\frac{2}{3}}$
4	$f(n) = 0.791x^{\frac{3}{4}}$

4 Discussion

Which algorithm did you use, and why?

We first tried to use Prim's algorithm due to its lesser time complexity but faced difficulties in implementing a Fibonacci heap. Additionally, when using an adjacency matrix representation of the graph with Prim's algorithm our code could not support n greater than 512 because of the excess amount of space the matrices took up for larger values of n, ($O(n^2)$ space complexity). Thus, we pivoted to Kruskal's algorithm and an adjacency list representation of the graph due to its smaller space complexity of $O(n+m)$ (where n is the number of vertices and m the number of edges). Within Kruskal's Algorithm, we created a disjoint-set data structure which proved much easier to implement, and the algorithm's run time, $O(E \log V)$, was still quite good.

Are the growth rates surprising?
We were surprised by the growth rate converging at 1.2 for dimension 0. However we rationalized it by considering that because the number of vertices are increasing, that must mean that the distance between various vertices is decreasing since the nodes are between 0 and 1. So regardless of there being 1024 or 262144 vertices in that line because the distances are so small between vertices there will be a path of weight 1.2.

We weren't too surprised by the other dimensions because it makes sense that they would be somewhat proportional to n since in a graph of n vertices there will be $n-1$ edges in the MST.

How long does it take your algorithm to run?
For n values from 124 to 2048 our code ran almost instantaneously, however once we reached $n = 4096$ our code began to run for 1 to 3 minutes, and for subsequently larger n this run time exponentially increased. For $n = 8192$ it took about 8 minutes, for $n = 16384$ it took about an hour, $n = 32768$ about 3 hours, $n = 65536$ about 5 to 7 hours and from here only increasing with the largest n value we could achieve, 131072, running for several hours overnight.

These run times are not super surprising because of the way n was doubling with each run of the algorithm, which its elements taking up greater space and time with each trial. Once we reached 262144 our algorithm could not be completed, as after a few hours the computer we were running it on received a storage capacity error. This was again not super surprising because of the sheer size of the n value.

Do you trust the random number generator?
Yes! Our random number generator generates a uniformly-distributed integer that is not dependent on our computer's clock cycles. We found that when our RNG was seeded with the clock cycle times we would get the same edge weights in a row after running the program a few times, as the computer's clock cycles are not completely independent, so modifying it proved to produce a much better random number generator.