CS 124 Programming Assignment 1: Spring 2021

Your name(s) (up to two): Catherine Liang, Nishant Mishra

No. of late days used on previous psets: 1

No. of late days used after including this pset: 2

Table:

Average MST Size for each n:

	Dim 0	Dim 2	Dim 3	Dim 4
n=128	1.175113	7.581495	17.638308	28.33672
n=256	1.139805	10.678034	26.778301	47.782247
n=512	1.181851	14.967085	43.494598	79.674215
n=1024	1.16845	21.056606	67.95418	130.806892
n=2048	1.194503	29.551414	108.381274	216.083849
n=4096	1.196495	41.910149	169.063714	362.054314
n=8192	1.195051	58.824887	267.81638	604.511032
n=16384	1.196562	83.197649	422.902674	1008.64755
n=32768	1.200252	117.469283	669.288376	1687.631838
n=65536	1.201693	166.030158	1057.961546	2825.539945
n=131072	1.200818	234.722987	1676.857439	4741.760704
n=262144	1.20138	*	*	*

* We didn't generate these values, but we expect them to be close to 331.79, 2662.49, and 7975.39 respectively, based-off our functions in the next section.

Function fitting:

We originally inputted the data into Excel, and created a scatterplot with a power function best-fit function. We then observed that the exponent constants for Dimensions 2, 3, and 4 were very close to 1/2, 2/3, and 3/4 respectively.

This made us curious about if the functions could be improved by fixing the exponential values and adding a constant. We used scipy's curve_fit() function to generate best-fit functions with those exact exponents, since we thought these might be more accurate to the mathematically expected functions for these data. We then made plots in python (see all below).

Both sets of functions are below:

Excel-Generated Functions that describe each plot

	function $f(n)$	$R^2 value$
Case 1 (Dim 0)	1.201*	(see note)
Case 2 (Dim 2)	$0.6879n^{0.494}$	0.999987367
Case 3 (Dim 3)	$0.7107n^{0.6585}$	0.0.99996814
Case 4 (Dim 4)	$0.7996n^{0.736}$	0.999977075

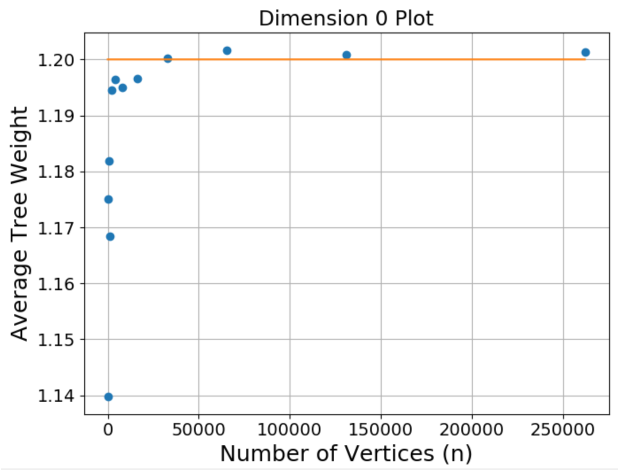
Curve-Fit Functions with Scipy

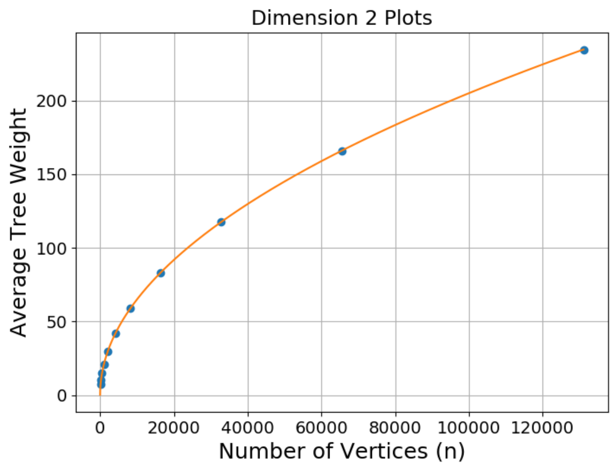
	function $f(n)$	$R^2 value$
Case 1 (Dim 0)	1.201*	(see note)
Case 2 (Dim 2)	$0.64741699n^{0.5} + 0.31051205$	0.999999
Case 3 (Dim 3)	$0.64939412n^{\frac{2}{3}} + 2.56836959$	0.999994
Case 4 (Dim 4)	$0.68778359n^{0.75} + 7.65308661$	0.99999

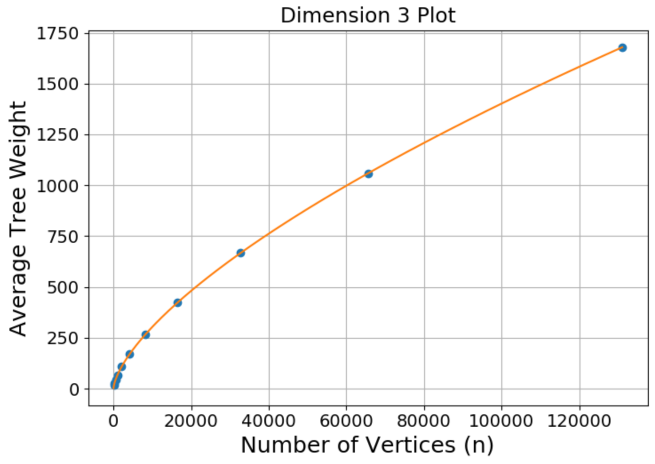
*note: For Dimension 0, the average MST size at first increases rapidly, overshooting 1.201 before it converges to 1.201. This doesn't fit a curve very well, and we have provided more detail in the f(n) discussion section below.

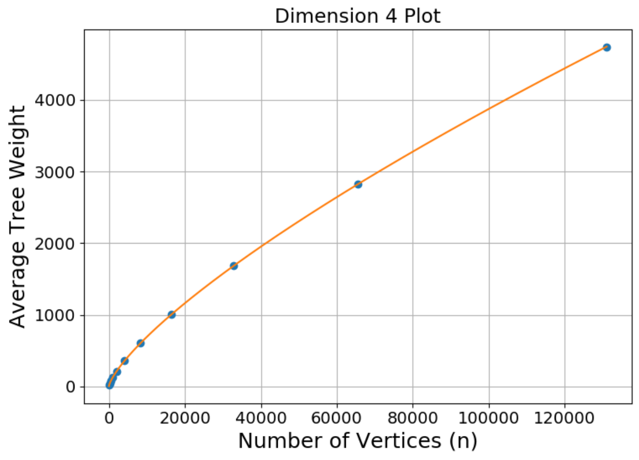
Plots of Functions and Data

Scipy curves fitted with data points on matplotlib









Discussion:

Algorithm Choice and Implementation:

Since we're working with complete graphs, which are the most dense graphs possible in terms of number of edges, we decided to use Prim's Algorithm. From a runtime standpoint, this makes a lot of sense. Prim's Algorithm, which runs in $O(|V|^2)$, is significantly faster than one of the alternatives, Kruskal's Algorithm, which runs in $O(|E|log|V|)$. The reason we can say this is because we know that there are $\binom{V}{2}$ edges, since each pair of vertices in a complete graph must have an edge between them. So we can replace $|E|$ in the run time with Kruskal's algorithm with $\frac{|V-1|*|V|}{2}$, so its $O(|V|^2 log|V|)$ when its plugged in a simplified. Clearly, this is a higher runtime than that of Prim's Algorithm, which is why we went with Prim's for our implementation.

We did this with adjacency matrices (a 2-dimensional array in C), since array indexing takes constant time, and our code loops through array columns, so we wanted to be able to access values quickly. In our running for larger n, we realized that the loop to determine the minimum weight edge forward is still slow, since it grows linearly with n. Yet this did not impact our results, so we stuck with the 2D array implementation.

We also worked to optimize our 0 dimension algorithm. Instead of generating the entire graph beforehand, we generated edge weights of the graph as the algorithm progressed. In the for loop in Prim's algorithms where we compare all the edge weights that are connected to a certain vertex, we randomly generated them each iteration, picking the smallest edge weight that came out of all the iterations to be our minimum edge weight to greedily pick. We discussed implementing something similar for other dimensions, but the optimization ended up not being necessary to actually compute the desired n for this assignment, so we use the same, unmodified version of Prim's algorithm method for those cases. We kept the optimization for the 0-dimension case however, as we had already implemented it.

Analysis and Discussion of f(n) for each dimension

We used Excel to plot and fit functions to our data.

Dimension 0 is a unique case, as it converges, for the other dimensions we obtained power functions that fit the data extremely well. From the results of our tests, we can see that for each of the non-zero dimensions, while all of the total edge weights increase with the number of vertices, they do so at a decreasing rate. A potential reason for this is that as we add more vertices in the unit spaces (whether that be a square, cube, or hyper-cube), the average distance between vertices starts to decrease. So while there are more vertices that need to be connected, the actual length of of the edges in the MST decreases.

One way to think about this is when you add one more vertex to a relatively densely-filled cube (say for Dimension 3), the average amount this vertex increases the MST size gets lower and lower, since it will be increasingly close to an edge on the MST.

<u>Dimension 0:</u>

When fitting a curve to our data from this case, we noticed that another function might fit more closely. We actually think the f(n) for this case is in the form $f(n) = 1.201(a)^{\frac{b}{n}}$ for some positive constants a and b, where $0 < a < 1$.

That is, the MST in this case increases extremely rapidly in size until about n=25000, at which point the size increases much more slowly then converges to about 1.201.

Theoretically, we think this makes sense, as when n is very large, each edge of the MST for large values of n will become smaller and smaller (e.g. possibly inversely proportional to the number of vertices). On a concrete level, we can assume this behavior from our code, since we generate n random numbers and add the smallest to the size of the MST as we build it.

Dimension 2: Within a Unit Square

The function we fitted to these data was: $0.6879n^{0.494}$ (see Function Fitting Table) We were originally very surprised at how well our data followed a power function in this case, as the R^2 value is 0.999987367, which is surprisingly high. We think this high R^2 value that we see amongst all of Dimensions 2, 3, and 4 is due to the fact that our values of n are quite large, each MST size run will be quite close to the expected value, whatever that may be. On a high level, we should see an increase in precision when we increase n.

We observe that this function takes the approximate form of $f(n) = c\sqrt{x}$, where c is a positive constant.

Dimension 3: Within a Unit Cube

The function we fitted to these data was: $0.7107n^{0.6585}$ (see Function Fitting Table). The R^2 value is similarly high to that of Dimension 2, and we observed that the function was also of the form $f(n) = an^b$, where a is a positive constant, and b is a positive constant less than 1.

Dimension 4: Within a Unit Hypercube

The function we fitted to these data was: $0.7996n^{0.736}$ (see Function Fitting Table). The R^2 value is similarly high to that of Dimension 2 and 3, and we observed that the function was yet again of the form $f(n) = an^b$, where a is a positive constant, and b is a positive constant less than 1.

Runtime Trends and Issues

Initially, when we ran our algorithm with a larger number of vertices, our code would be killed, because our arrays were using the stack. To prevent memory cache issues that originally killed our code, we used global variables to store graphs, and malloc-ed our 2D array and other useful arrays on the heap.

For dimension 0, our values are very fast, with all running within a minute except the last two values of n. The last n takes about a couple hours to finish, depending on the machine.

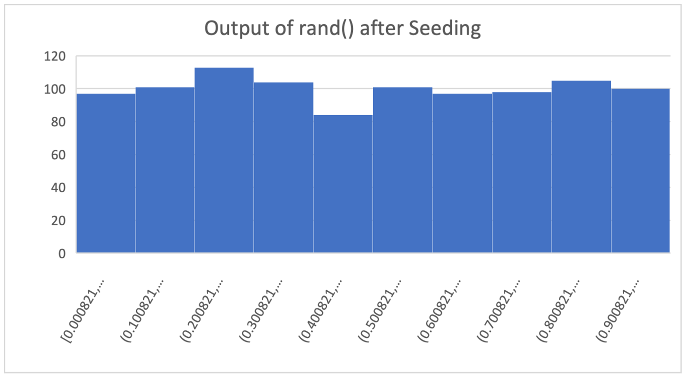
For dimensions 2, 3, and 4, our values until around 80000 are quite fast, finishing within 30 minutes. 131072 finished in a few hours, and the last value of n took overnight runs. Running on machines with larger cache sizes allowed us to avoid our code being killed.

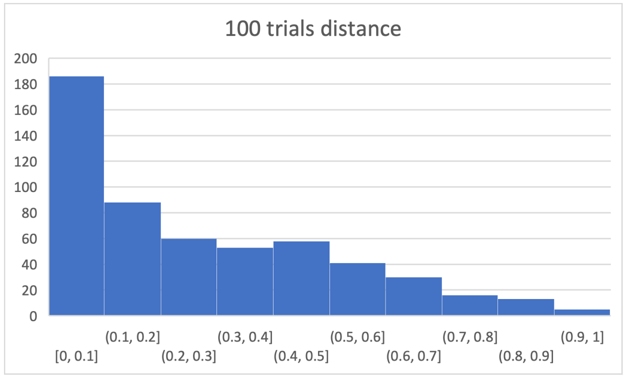
Random Number Generation

We noticed that when using srand to seed values, if srand was called repeatedly, the random number generated by rand would be nearly identical. We assume this is due to the seed (time) being a very similar number. We settled on seeding the random number generator just once.

We were initially suspicious of the randomness of our random number generator (based-off the rand method). We tested this by generating around 1000 random numbers in the range $[0, 1)$ (in practice equivalent to $[0, 1]$).

Plotting these values as a histogram in Excel, we obtained a very uniform distribution, leading us to think our rand is working correctly. We then ran our graph-generation code on these random values, and plotted the edge weights for Dimension 2. The histogram for these edge weights decreased sharply, as expected, so this also encourages the conclusion that our random is working.





With this evidence, we are sufficiently confident in the random value generator we created.