Programming Assignment 1

Harvard ID: 30940040
Harvard ID: 80943423

Abstract

The purpose of this assignment was to implement a MST algorithm, and test it on a number of different random graphs. We chose to implement Prim's Algorithm, and used a min-heap to store edge candidates. The graphs we built randomly generated vertices (or edges in the dimension 0 case) over $[0, 1]$ in dimensions 2, 3, and 4, and used then use a simple euclidean distance function to calculate edge weights. For each dimension, we explored how the expected weight of the MST grew as a function of n. For large $n > 32768$, it became more difficult to generate graphs within a reasonable amount of time and avoid space issues. We address some of the optimizations for time and space that we implemented in the discussion section below.

1 Quantitative Results

We ran our implementation of Prim's Algorithm on each dimension for the values $n = 128$ 256; 512; 1024; 2048; 4096; 8192; 16384; 32768; 65536; 131072. Below, there is a table for each dimension, along with a graph showing the growth rate of the average MST. After plotting our points, we made an educated guess about what the function, $f(n)$, of this growth rate is for each dimension.

1.1 Dimension 0

Table 1: Growth Rate of Expected Weight of MST

n	$E[w]$
8	1.156
16	1.351
32	1.212
64	1.206
128	1.222
256	1.196
512	1.207
1024	1.200
2048	1.205
4096	1.193
8192	1.204
16384	1.199
32768	1.202
65536	NaN
131072	NaN

$$\hat{f}(x) = \frac{6x}{5x + 1}$$

The most notable results in dimension 0 were that the values we're asymptotic to about 1.2. Thus, when we we're guessing our function, we estimated the 6 in the numerator and 5 in the denominator as the asymptote, and the graph did a good job representing the true growth function, $f(n)$.

1.2 Dimension 2

Table 2: Growth Rate of Expected Weight of MST

n	$E[w]$
8	1.989
16	2.640
32	3.984
64	5.419
128	7.486
256	10.584
512	14.951
1024	21.074
2048	29.575
4096	41.655
8192	58.829
16384	83.1974
32768	117.588
65536	NaN
131072	NaN

$$\hat{f}(x) = \frac{2}{3}x^{\frac{1}{2}}$$

After trying a number of functions, we came to the conclusion that the growth was best described by an exponential function where the exponent was $0 < e < 1$. It resembled $x^{\frac{1}{2}}$, and after plotting it, the curve was a good fit. We proceeded to test different values for the coefficient, until the line was fit almost perfectly.

1.3 Dimension 3

Table 3: Growth Rate of Expected Weight of MST

n	$E[w]$
8	2.657
16	4.736
32	7.617
64	11.005
128	17.727
256	27.7177
512	43.554
1024	68.2981
2048	107.567
4096	169.437
8192	267.307
16384	422.846
32768	668.743
65536	NaN
131072	NaN

$$\hat{f}(x) = \frac{5}{6}x^{0.64}$$

We noticed this curve resembled the previous curve of dimension 2, except it grew faster. Thus we began by increasing the value of the exponent, and when it seemed a close fit to the points, we tweaked the coefficient to make it a very good fit. Dimension 0 was the only unique case (since there was no euclidean distance function, there was no relationship between each generated edge). Our hypothesis is as follows: as the dimension increases, the maximum length edge increases (eg. hypotenus of hypercube is longer than that of unit square). Because more points become available at the same rate, but there is a greater amount of space they can be in, the average weight of an MST grows faster in higher dimensions because edges will have higher weight more frequently.

1.4 Dimension 4

Table 4: Growth Rate of Expected Weight of MST

n	$E[w]$
8	4.028
16	5.990
32	10.583
64	16.897
128	28.228
256	47.081
512	78.478
1024	130.136
2048	216.165
4096	361.326
8192	603.700
16384	1008.39
32768	1687.08
65536	NaN
131072	NaN

$$\hat{f}(x) = \frac{5}{6}x^{0.73}$$

The final growth function is again very similar to the ones of dimensions 2 and 3. It grew faster than the other dimensions, but resembled the same curve, so we started by slightly increasing the exponent. Then we made the line a proper fit by increasing the exponent to raise the angle of the curve.

2 Discussion

2.1 Implementation Details

We implemented two classes to implement this algorithm: Heap and Complete_Undirected. The Heap class was a Binary Min-Heap that was implemented using dynamic arrays (vectors), and the Complete_Undirected Class was implemented using an adjacency matrix. Our Heap class had three public methods: *insert*, *delete_min*, and *get_size*. Elements of our heap were stored as structs with two fields, vertex number and distance.

Our graph class had a constructor function that took two arguments, vertices (the number of vertices in the graph) and dimension. We exposed 7 public methods, *generate_graph* (generated the random edge weights), *get_vertices*, *get_dimension*, *get_graph*, *print_graph*,and *prims*. To generate randomly weighted edges in *generate_graph*, we seeded values based on the machine's clock (in milliseconds) each time a new instance of Complete_Undirected was instantiated. We implemented a private method *gen_rand* that used C++'s rand() function. In *generate_graph*, we called *gen_rand* each time we needed to generate a random edge weight in dimension 0, and in dimensions 1 to 4, we used it to generate random points on the unit square, unit cube, etc.

The *prims* method ran Prim's algorithm on our randomly generated graph, calculated its MST, and returned the weight of the MST. Prim's algorithm has a run time of $O(|V| * deletemin + |E| * insert)$, and since we used a Binary Heap with $0(log|V|)$ for both insert and delete, our implementation of Prim's had an asymptotic runtime of $O(|E|log|V|)$.

2.2 Are the growth rates (the $f(n)$) surprising? Can you come up with an explanation for them?

To find the growth rate function for each dimension, we plotted the number of vertices n against the average weights of the corresponding MSTs on an xy plot and tried fitThe growth rates for each of our dimensions were not surprising. As the dimension increased, the growth rate as a function of n (the number of vertices) became larger (a faster growing function). This can be explained by the fact that as you increase the dimension, the max possible edge weight increased as well. For dimension 0, the max edge weight was 1, and for dimensions 2 to 4 the max edge weight was $\sqrt{d}$, where d is the dimension. Since the max possible edge weight increased with higher dimensions, it makes sense that for the same number of vertices, the average weight of an MST was greater for higher dimensions.

2.3 Seeding the RNG

In seeding the random number generator, we explored three different times. Seeding before generating each edge, seeding before generating each graph, and seeding once before running randmst. Seeding before generating each edge did not make sense. Every time *srand* is called, *rand* operates off of a unique pseudorandom distribution. Thus, seeding multiple times in generating one graph breaks the pseudorandom distribution of the graph itself, and each edge is not necessarily 'random' in comparison to each other. Seeding only once in the program flat out did not work - it gave us the same value for average MST weight consistently for a given n. Since all graphs generated were on the same pseudorandom distribution, they lacked the necessary variance between trials.

2.4 Code optimizations

We optimized our random edge weight generation by taking advantage of the fact that we were generating undirected graphs. That meant that an edge $(u, v) \in E$ had the same weight as the edge (v, u), so we only had to generate a random edge weight once for every edge between two vertices (it would have been incorrect to generate two random edge weights anyway).

We also optimized the speed of our algorithm by about 25% by 'throwing out edges'. After calculating the growth functions, we had a constant time way of finding an estimate of the average weight of an edge with $\hat{f}(n)$. We divided this by n for any given graph to get it's average edge weight. From here, we used a constant multiple to get an upper bound on edge weights to throw out. This optimization worked very well because it *always* returns the right MST. Since we initialized all initial vertex distances to ∞, if edge that would have been in the MST was deleted, then the value returned is infinity. Thus, raising the multiple increases the likelihood that ∞ won't be returned, but when it is not we can be certain that it is a valid MST.

3 Final Thoughts

3.1 Makefile

We wanted to use a somewhat robust Makefile to have a good file structure. For that reason, we are keeping all of our header files in an 'include' directory, and the rest of our program in a 'src' directory. Object and executables also had respective 'obj' and 'bin' directories, but we opted to make randmst.exe out of the 'bin' directory so it could run according to the spec. One important note is that make must be run from the 'src' directory, as that is where it is present.

3.2 Potential Optimizations

In the end, our program took very long to run values greater than 32-thousand vertices. We believe this is likely because we opted to store all of our edge weights in an adjacency matrix, instead of computing them dynamically (within Prim's) and discarding the edges according to our removal function from **2.4**. This would have most definitely optimized for space. In terms of speed, we could have explored multi-threading to run a number of processes in parallel (like generating multiple edges at once). We did not know, however, if using a multi-threading approach in C++ would be allowed within the scope of the assignment.