Programming Assignment 1 Report

81238782, 60944733

February 27, 2017

1 Introduction and Methods

We developed a Java program to generate four categories of complete undirected graphs, build their minimum spanning trees (MSTs), and calculate each MST's weight. In order to generate edge weights and vertex coordinates, we used a pseudo-random number generator seeded with machine clock time from Java's Random library. We implemented a priority queue and a linked list data structure and used Prim's algorithm to construct the MST of each graph and find its weight.

2 Results

2.1 Average MST Weight for Increasing Number of Vertices

First, we analyzed how the average weight of the MSTs for each category of graphs grew as a function of the number of vertices (n). This data is shown in Table 1 and Figure 1. Category 0 graphs are complete, undirected graphs with each edge weight drawn independently from the uniform distribution between 0 and 1. Category 2-4 graphs refer to the number of dimensions d: each vertex is assigned a point in d dimensions with coordinates drawn independently from the uniform distribution between 0 and 1, and the edge weights are calculated as the Euclidean distance between them. In other words, the vertices are picked randomly from the inside of a d-dimensional unit hypercube. The data on MST weight represents the mean of 50 different randomly generated graphs for each category and value of n. The random number generator was reseeded with machine clock time for each iteration.

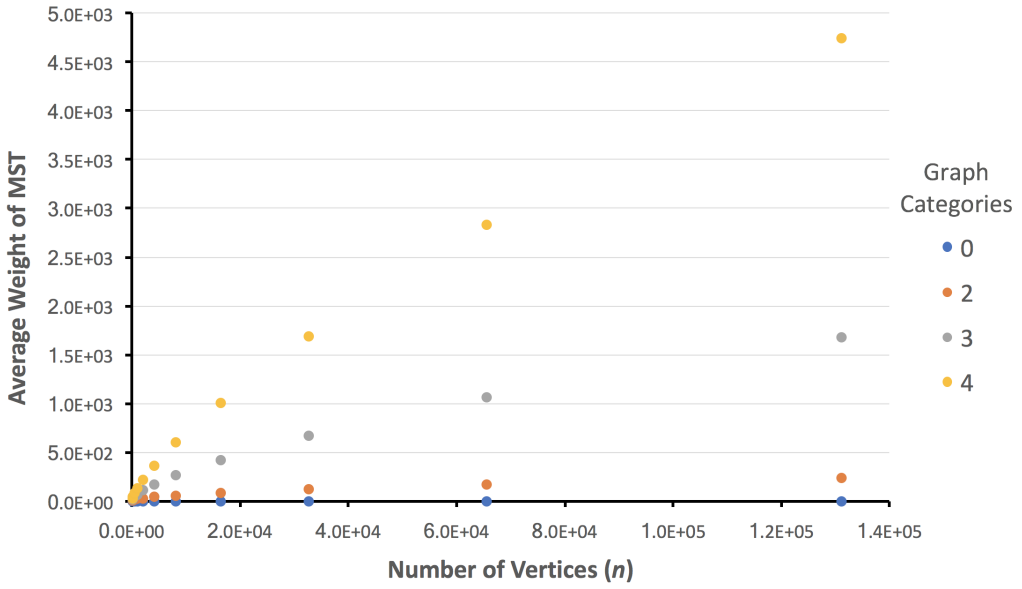

Figure 1: Mean average MST weights based on 50 repetitions for graphs with increasing number of vertices.

Table 1: Average MST weights for Graphs with n Vertices

	Graph Category			
n	0	2	3	4
128	1.19	7.62	17.6	28.46
256	1.2	10.68	27.57	47.17
512	1.2	14.98	43.34	78.38
1024	1.19	21.04	68.2	130.15
2048	1.2	29.61	107.27	216.64
4096	1.2	41.76	169.36	361.24
8192	1.2	58.96	267.27	603.62
16384	1.2	83.18	422.5	1008.75
32768	1.2	117.55	668.49	1688.18
65536	1.2	166.03	1058.54	2828.12
131072	1.2	234.59	1677.06	4740.08

2.2 Average MST Weight as a Function of n

Our hypothesized functions estimating the average MST weight based on the number of vertices in the graph are shown in Table 2. R^2 values are included to indicate how well the functions fit the empirical data. For category 0 graphs, the average MST weight did not deviate significantly

from approximately 1.2 for increasing values of n. Therefore, we hypothesize that this function is constant. (Because correlation is only defined for random variables with non-zero variance, this has no R^2 value). For category 2-4 graphs, we observed a radical-looking trend between n and average MST weight, so we squared the data, fit it to linear parameters, and back-calculated to determine square root functions of best fit.

Table 2: Estimated Functions of n for Average MST Weight

Graph Category	$F(n)$	R^2
0	1.20	N/A
2	$0.65\sqrt{n}$	1
3	$4.5\sqrt{n}$	0.98
4	$12.5\sqrt{n}$	0.96

3 Discussion

3.1 Algorithm Choice

We chose to implement Prim's algorithm because we expected challenges with sorting the edge weights for large values of n in Kruskal's algorithm. Our implementation keeps space complexity low by not storing the $O(n^2)$ edge weights in memory for category 2-4 graphs, and this technique was only possible because we were using Prim's algorithm. We also avoided probabilistic results since we did not have to develop heuristics to throw out edge weights over a given length. Thus, our algorithm always calculates the correct MST whereas in an implementation of Kruskal's algorithm with some heuristics, there is a very small, but still nonzero, probability of the incorrect MST.

In addition, we specifically chose to implement the priority queue underlying Prim's algorithm with a linked list instead of a binary tree because $|E| = \frac{|V|(|V|-1)}{2}$ in this complete graph, so the constant time insert operations combined with linear time delete-min operations will make the whole algorithm run in $O(|V|^2)$ time. For a binary heap implementation, the running time would have been $O(|V|^2 \log |V|)$.

3.2 Random Number Generator

We used the Java Random library to instantiate a Random object which generated pseudo-random numbers from the uniform distribution on the interval [0, 1). Although the assignment asked us to generate a random number on the interval [0, 1], since the probability of generating exactly 1 on a continuous distribution is 0, we did not think this would affect the results.

We instantiated the Random object seeded with machine clock time (which constantly changes) for every random graph we generated. If we had used the same seed each time, we would have generated the same random graph multiple times, resulting in identical average MST weights. In our testing of our algorithm, we never had identical MST weights, suggesting that the edge weights or vertex cooredinates were never exactly identical. Therefore, we do trust our random number generation in the sense that it will never generate the same sequence of random numbers as long as it is seeded properly. However, we trust this randomness only to the extent that pseudo-random numbers can be deemed truly random.

3.3 Running Time and Optimization

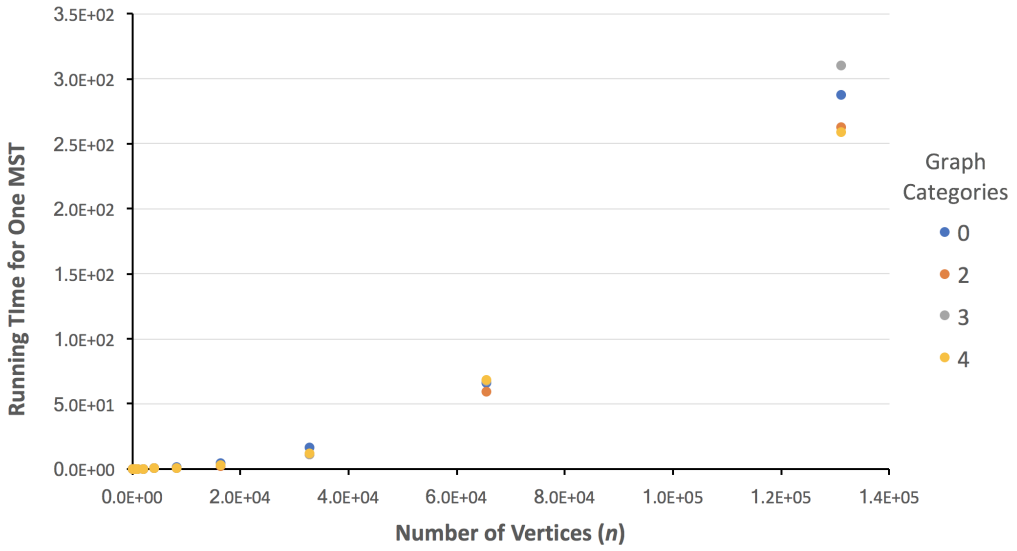

Figure 2: Running time of our implementation of Prim's algorithm for increasing numbers of vertices.

Figure 2 shows how long our program runs to calculate the average MST weight for graphs with increasing number of vertices. Because this data does not represent the mean of a sample, the numbers are not very accurate, but the general trend is that the program runs in polynomial time. This makes sense since Prim's algorithm is $O(n^2)$.

In our programming, we realized we had to develop one particular technique to reduce the running time of Prim's algorithm. Because the input graphs are complete, the priority queue fills up with all of the vertices in the graph after the first edge of the MST is identified. In order for our algorithm to truly run in $O(|V|^2)$ time, we needed to maintain not only constant insertion but also constant updating of a vertex's distance from the growing MST. To accomplish this, we kept an array of pointers to each vertex's linked node in the priority queue. This way, if we wanted to update a vertex's distance in the priority queue, we could simply go to the array, follow the pointer, and then set the linked node's value. This could be accomplished in constant time versus the linear time that would be necessary to search the linked list from the beginning. This also avoided having to instantiate new linked nodes every time the priority queue was updated. This is an example of something we learned by implementing the algorithm over theoretically analyzing it.

3.4 Growth Rate Intuition

Without knowing with certainty the explicit form of the function estimating the average MST size using the number of vertices, from the data, we observe that the average MST size increases with slowing growth rate as n approaches infinity. At first, this might seem counterintuitive since for each vertex added to a graph, the MST needs to include another edge, so the growth rate should

be constant. However, this mistakenly assumes that each edge weight added to the MST is an independent random variable. In fact, since each edge weight added is the minimum of a growing pool of available edges, its expected value is approaching 0 as n grows. Therefore, we do expect the growth rate of the function to decrease as n approaches infinity.

For category 0 graphs, we observed approximately constant MST weights even as the number of vertices approaches infinity. This zero growth rate is surprising since one would expect more vertices and edges to increase the total MST weight. We do not actually believe the average MST weight for all values of n is 1.2. (Take the simple extreme case of a graph with just $n = 2$ vertices. The maximum MST weight is the maximum single edge weight, which is 1, and the expected MST weight is $\frac{1}{2}$). We considered that maybe the function is asymptotically approaching 1.2, but then we would not have achieved empirical results with average MST weights greater than 1.2. Perhaps the average MST weight grows too slowly to observe like the $\log^* n$ or the inverse Ackermann function. We would need to run the program for much higher values of n to answer this question empirically. Overall, this is a curious result that merits further investigation.