CS 124: Programming Assignment 1

Hao Wang Chao Cheng

March 1, 2020

1 Quantitative Results

1.1 Table of Results

Average Weights				
Size	0-Dim.	2-Dim.	3-Dim.	4-Dim.
128	1.162935495	7.630529782	17.5274274	29.11715893
256	1.151544456	10.59075748	27.27618402	47.77013319
512	1.198177366	14.9716805	43.52688469	78.43835405
1024	1.225937435	21.08375776	68.02592257	129.6327967
2048	1.187548705	29.51409682	106.9611423	216.9463079
4096	1.213256023	41.70816941	169.0656925	360.8684949
8192	1.202708402	58.97468603	267.2882169	602.8943309
16384	1.201245242	83.10736216	422.7191834	1009.435431
32768	1.196438335	117.5505205	669.3618846	1687.943916
65536	1.198175487	166.0095534	1058.679153	2826.229297
131072	1.202087768	234.5978984	1677.312888	4743.438661
262144	1.202546574	331.6290273	2658.336056	7949.513048

1.2 Function Approximation

- **0-Dimensional:** $f(n) = 1.2$
  $f(n)$ appears to converge to 1.2 as n grows.

- **2-Dimensional:** $f(n) = 0.67972381n^{0.495644972} \approx 0.68n^{1/2}$
  We approximated $f(n)$ using power regression, which seemed to fit the data very well ($r = 0.999989358$). In particular, the exponent on n appears to be very close to $\frac{1}{2}$.

- **3-Dimensional:** $f(n) = 0.707834929n^{0.65910846} \approx 0.71n^{2/3}$
  We also used power regression ($r = 0.999993078$) to approximate $f(n)$. In particular, the exponent on n appears to be very close to $\frac{2}{3}$.

- **4-Dimensional:** $f(n) = 0.79385626n^{0.737172751} \approx 0.79n^{3/4}$
  We also used power regression ($r = 0.999977992$) to approximate $f(n)$. In particular, the exponent on n appears to be very close to $\frac{3}{4}$.

2 Reflection

Algorithm We chose to use Prim's algorithm for this assignment, based on the consideration of correctness, space complexity, and time complexity. One of the main advantages of using Prim rather than Kruskal was not needing to store the edges of the graph in memory. Instead of storing the edges and their weights, we simply calculated the Euclidean distance between the vertices of the edges as we processed them.

We also considered the hint given to us about throwing away some edges, but decided against using it, since it seemed like it would difficult to justify our algorithm's correctness if we did throw out edges. We found that when we kept approximately five edges for every vertex, i.e. $5n$ edges in total for n vertices, our algorithm returned the same minimal spanning tree weight as regular Kruskal (without dropping any edges) for small values of n. However, for larger n, we thought it would be very difficult to calculate and justify the probability of getting the correct MST weight

after dropping edges, and we also considered that there would be a possibility of the algorithm not finding the correct MST.

At first, we attempted to implement Kruskal and store the edges in the graph. For larger values of n, the algorithm was too slow (we left it running overnight for $n = 262144$, which was still not enough time). We then decided to throw away edges as explained above, however, this still did not reduce the running time significantly. At this point, we decided to switch to Prim, since we determined that its running time would be faster than Kruskal; we were able to run the algorithm 5 times for $n = 262144$ in roughly an hour for each case with Prim.

Both Prim and Kruskal were implemented according to the approach presented in lecture. However, for Prim, we used the indexed min-heap data structure, rather than a binary min-heap, since the index min-heap has an inverse mapping such that it takes constant time to get the index of a given node, which improves the speed of the algorithm significantly.

Running Time Another reason we chose Prim is due to the fact that it is much faster than Kruskal. For complete graphs, we have that $E = \binom{V}{2} = O(V^2)$. The time complexity of Kruskal is $O(E \log E)$, while the time complexity of Prim is $O(E \log V)$, and the difference between $\log E$ and $\log V$ in the running times is quite significant, as it makes Prim much faster than Kruskal for complete graphs.

Growth Rate The $f(n)$ grow rather quickly, on the order of $O(n^\alpha)$ for some positive α. We attribute this to the fact that, for higher dimensions, the average distance between nodes increases by definition of Euclidean distance. Each time we add a dimension, we increase the maximum possible distance between two nodes. For instance, the maximum distance between two nodes is $\sqrt{2}$ for 2D space, $\sqrt{3}$ for 3D space, and 2 for 4D space, and so on, since as we increase the number of dimensions, the points become more spread out. In addition, each added dimension adds another term to the distance calculation. Thus, increasing the dimension increases the average weight of the edges.

In particular, for the 2-dimensional, 3-dimensional, and 4-dimensional cases, we noticed that the $f(n)$ seemed to be in the form $f(n) = kn^{\frac{d-1}{d}}$, where k is a constant and d is the number of dimensions. This implies that no matter how large d is, the exponent of n grows infinitely close to but never reaches 1, meaning $f(n)$ never reaches the order of $O(n)$. For the 0-dimensional case, we noticed that the values of $f(n)$ seemed to converge to 1.2, which implies that in that case, the average weight of the MST is constant and does not depend on n.

Random Number Generation For the coordinates of the nodes of the graphs (or, in the 0-dimensional case, the weights of the edges), we used the `java.util.Random` class to generate doubles between 0.0 and 1.0 pseudo-randomly. To seed the pseudo-random generator, we used the time from the machine clock.