# CS 124 Programming Assignment 1: Spring 2022

**Katrina Brown, Suvin Sundararajan**

**No. of late days used on previous psets: 2 (Suvin), 1 (Katrina)**
**No. of late days used after including this pset: 2 (Suvin), 1 (Katrina)**

## Overview:

For Programming Assignment 1, we decided to utilize Kruskal's Union Find Algorithm to create a minimum spanning tree in C++. The greedy algorithm finds the correct MST for an arbitrary input graph. To implement the MST algorithm, we originally were programming in Python, but found that the higher overhead made initial calculations difficult over 8000 data points with a standard Kruskal Algorithm. Porting the code towards C++ yielded a significant increase in speed. We also originally implemented Prim's algorithm in C++, but found that our implementation of Kruskal was comparatively faster. This was likely due to the additional heuristics we implemented to speed up the run-time of Kruskal. Kruskal runs more quickly on sparse graphs than Prim. Because part of our strategy involved throwing out edges with weights above a certain threshold, our algorithm dealt with sparse graphs and thus benefitted from the use of Kruskal over Prim.

We had difficulty addressing the high time required to run Kruskal on input graphs with large n. To address this, we approximated $k(n)$ for each dimension, where $k(n)$ is the largest edge length likely to be present in the minimum spanning tree. To do this, we measured the maximum edge weight present in the tree across several trials for many values of n in each dimension. We then calculated the lines of best fit to approximate $k(n)$ for each dimension. When constructing the graph, if an edge weight was greater than $k(n)$, then we did not add the edge to the graph. This is justified because edges with weight greater than $k(n)$ are extremely unlikely to be used in the optimal minimum spanning tree, so throwing them out is unlikely to prevent our algorithm from constructing the correct (or very near correct) MST. We verified experimentally that this $k(n)$ heuristic did not result in significantly un-optimal minimum spanning trees, because the average weights of the minimum spanning trees produced in our trials were not statistically significantly different before or after implementing this heuristic.

This also proves that the optimization works as intended; if the new weights had been statistically significantly different, then the optimization would have been removing edges that would have decreased the weight of the calculated MST had they not been thrown out.

As a result, kruskal() ran much more quickly. For instance, this optimization reduced the runtime of kruskal() on an input graph with 32,000 vertices in 4 dimensions from 85 seconds to 3.5 milliseconds. We approximated k(n) as follows: for 0 dimensions, $k(n) = 6 \cdot n^{-0.919}$; for 2 dimensions, $k(n) = 2.6 \cdot n^{-.514}$; for 3 dimensions, $k(n) = 2 \cdot n^{-0.342}$; and for 4 dimensions: $k(n) = 1.6 \cdot n^{-0.243}$.

We implemented the Union by Rank and Path Compression heuristics for Kruskal to result in a final asymptotic runtime of $\Omega(n \cdot \log^* n)$. The runtime of $\Omega(n \cdot \log^* n)$, which is near-linear because $\log^* n$ grows extremely slowly, is consistent with the runtime we observed when running our algorithm across many values of n. For each 2x increase in the number of vertices in the graph, the time required to run kruskal() on the given graph increased by slightly over 2x. In the four-dimensional case, our algorithm took 480 milliseconds to run on a graph with 262144 vertices, approximately 220 milliseconds to run on a graph with 131072 vertices, approximately 105 milliseconds to run on a graph with 65536 vertices, and so on. We saw a similar near-linear runtime for the 0,2, and 3 dimension cases. The runtime of kruskal() was approximately the same across the 0, 2, 3, and 4 dimensional graphs, which makes sense because the format

of the input graph (with fixed edge weights) was the same regardless of the way that the edge weights were constructed. The code to generate the graphs took proportionally longer for 4 dimensions than for 3, and longer for 3 dimensions than for 2, which makes sense because a greater number of computations were required for the higher-dimensional cases.

The code to generate each random graph took longer than the code to run kruskal() on that graph. To generate a random graph with 262144 vertices took approximately 3 minutes for a 0-dimension graph, 60 seconds for a 2-dimension graph, 70 seconds for a 3-dimension graph, and 2 minutes for a 4-dimension graph.

## Table:

In the 0 dimensional case, the average MST weight does not appear to depend on the number of vertices in the graph. This may be because for increasingly large values of n, each vertex is increasingly likely to have several extremely low-cost edges to other vertices, so although the number of edges in the MST increases, the average weight of each edge selected decreases. For smaller values of n, there are fewer edges in the MST, but there are likely to be fewer edges available that have very low cost.

| F(n), 0D | MST Average Weight | F(n), 2D | MST Average Weight | F(n), 3D | MST Average Weight | F(n), 4D | MST Average Weight |
|---|---|---|---|---|---|---|---|
| 128 | 1.287471 | 128 | 7.545031 | 128 | 17.79811 | 128 | 28.26211 |
| 256 | 1.209552 | 256 | 10.75792 | 256 | 27.70182 | 256 | 47.50942 |
| 512 | 1.221583 | 512 | 14.95753 | 512 | 43.12343 | 512 | 78.08763 |
| 1024 | 1.172664 | 1024 | 20.87614 | 1024 | 67.95974 | 1024 | 130.074 |
| 2048 | 1.206965 | 2048 | 29.63265 | 2048 | 107.2285 | 2048 | 216.4085 |
| 4096 | 1.217146 | 4096 | 41.69846 | 4096 | 168.9186 | 4096 | 360.176 |
| 8192 | 1.198017 | 8192 | 58.95697 | 8192 | 267.3997 | 8192 | 603.4527 |
| 16384 | 1.202768 | 16384 | 83.2098 | 16384 | 422.5958 | 16384 | 1009.078 |
| 32768 | 1.203739 | 32768 | 117.4669 | 32768 | 668.2369 | 32768 | 1690.679 |
| 65536 | 1.204191 | 65536 | 166.0711 | 65536 | 1057.491 | 65536 | 2829.4310 |
| 131072 | 1.202361 | 131072 | 234.5761 | 131072 | 1676.541 | 131072 | 4742.0911 |
| 262144 | 1.2021 | 262144 | 331.626 | 262144 | 2657.87 | 262144 | 7952.16 |

This inverse relationship between number of vertices and average weight of each edge in the MST may explain why the average MST weight does not appear to depend significantly on the number of vertices in the graph.

In the 2, 3 and 4 dimensional cases, the expected MST weight increases as $n$ increases. As the dimension $f$ increases, the average weight of each edge in the graph increases, so the average weight of the MST increases.

## Function:

As such, we can derive a formula that models the average MST weight that best follows the experimental data. Let $W_i(n)$ denote the expected weight of the MST of a graph with $n$ vertices in the $i$-dimensional case, where $i \in \{0, 2, 3, 4\}$.

For the 0 Dimension, the function would approach $W_0(n) = 1.2$ as per our tests and doesn't require an extensive formula.

For dimensions 2-4, we can derive a generalized formula. Let $w$ represent the specific coefficient to alter the amplitude, $n$ denote the number of vertices in the graph, and $f$ denote the dimension of the MST (2,3, or 4). For dimension 2, let $w = 1.3$. For dimension 3, let $w = 1.0$. For dimension 4, let $w = 0.9$. We obtain the following equations of best fit for dimensions 2, 3, and 4.

$$W_f(n) = w \cdot \frac{f-1}{f} \cdot n^{\frac{f-1}{f}}$$

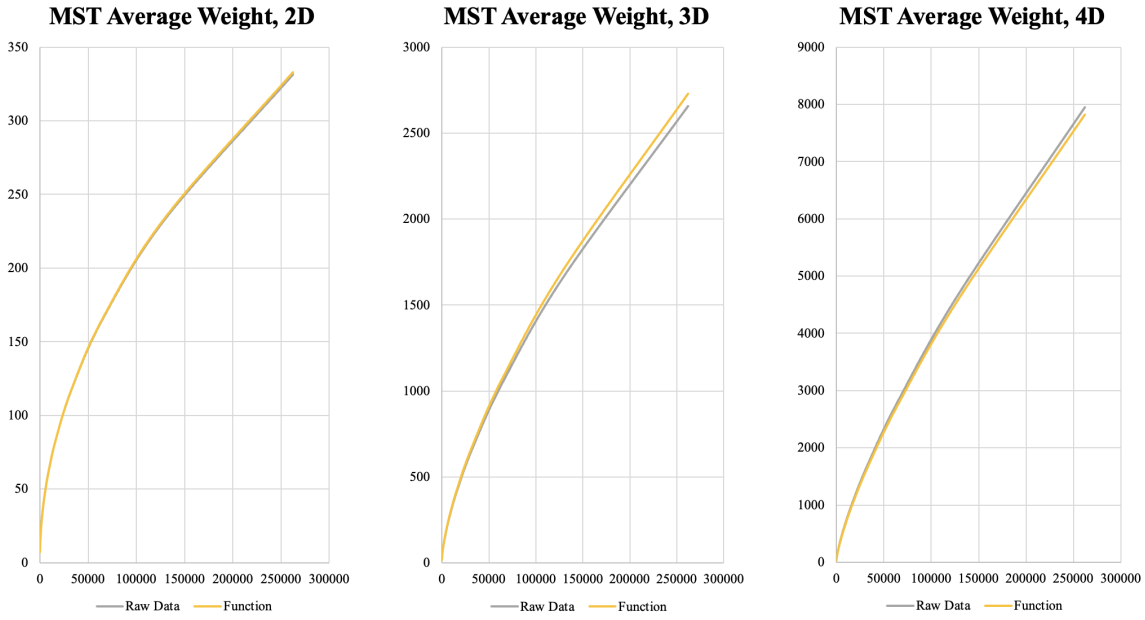
$$W_0(n) = 1.2$$

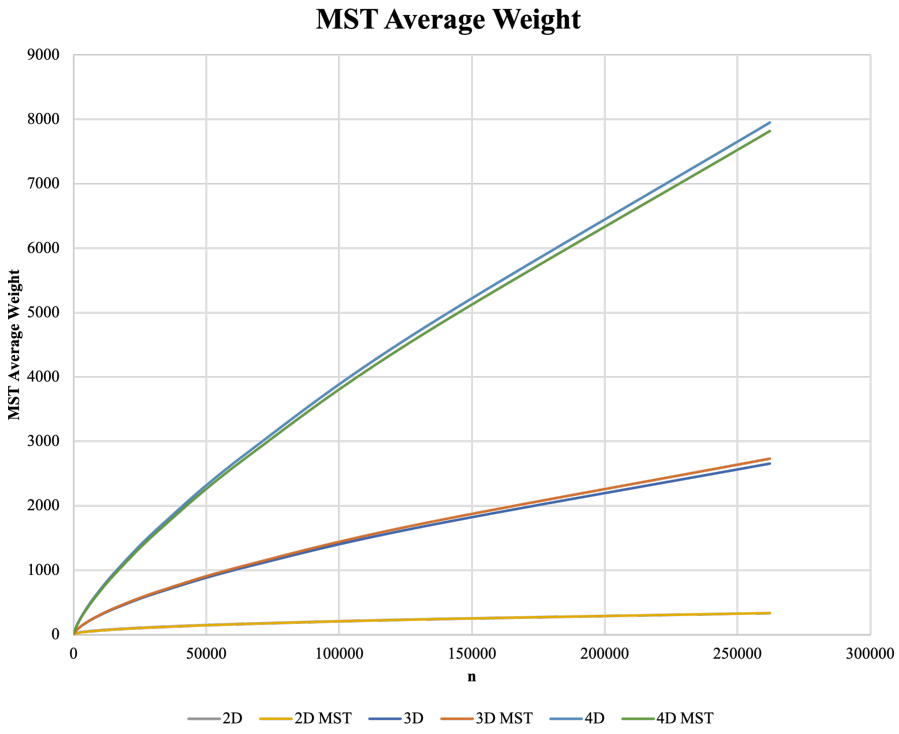
$$W_2(n) = 1.3 \cdot \frac{f-1}{f} \cdot n^{\frac{f-1}{f}} = 1.3 \cdot \frac{1}{2} \cdot n^{\frac{1}{2}} = \frac{13}{20} \cdot n^{\frac{1}{2}}$$

$$W_3(n) = \frac{f-1}{f} \cdot n^{\frac{f-1}{f}} = \frac{2}{3} \cdot n^{\frac{2}{3}}$$

$$W_4(n) = 0.9 \cdot \frac{f-1}{f} \cdot n^{\frac{f-1}{f}} = 0.9 \cdot \frac{3}{4} \cdot n^{\frac{3}{4}} = \frac{27}{40} \cdot n^{\frac{3}{4}}$$

The graphs below plot the MST weight averaged over five trials vs the number of vertices in the graph. The plots also display the approximated functions $W(n)$.






## Further Discussion:

Attempting a moving average, exponential, and logarithmic fit did not yield a satisfactory representation of the model, shown with low R values. This exponential approach model is only known to work in dimensions 0, 2 - 4, and would require further testing and validation for higher dimensions. However, based on the data provided through our tests, it is fair to say that this accurately represents the current data. This formula is not surprising; as the MST expands significantly over a large amount of data points, there are more edges that are available with a lower cost. So the function of the average weight increases by a factor of $\frac{x-1}{x}$. With an standard error rate of less than 0.05 for these functions, the function accurately models the expected weight of the MST given the number of vertices $n$ in the graph.

We did not have issues with the random number generator. We trust the random number generation. Our program successfully generates a different minimum spanning tree each time a new graph is constructed, since each time a new input graph is constructed, we reseed the "uniform real distribution" number generator using a random number produced by the rd() function.

**Github Link**:
https://github.com/SuvinSundararajan/CS124-Prog1