CS 124 Programming Assignment 1: Spring 2023

Your name(s) (up to two): Tomas Arevalo, Joanna Bai
Collaborators: N/A
No. of late days used on previous psets: (1,1)
No. of late days used after including this pset: (1,1)

1 Overview

For this assignment, we chose to implement a minimum spanning tree algorithm (Kruskal's) on undirected, complete graphs in C++. As specified in the instructions, we randomly generated n points to correspond with n vertices, where $n = 128; 256; 512; 1024; 2048; 4096; 8192; 16384; 32768; 65536; 131072;$ and 262114 to build our graphs in varying dimensions. In dimension 0, the edge weights are randomly generated for vertices that correspond directly with their indices. In dimensions 2, 3, and 4, our vertices were assigned points in a unit object and the edge weights were calculated by Euclidean distance between the points.

This write-up contains the following: (1) Our reasoning and explanation for our algorithm of choice and our optimization strategies. (2) The results of our algorithm on various quantities of n and dimensions, as well as an estimate for a function $f(n)$ to approximate the MST size for the different number of vertices. (3) We discuss our experience throughout this assignment and things we learned.

2 Reasoning

We decided to use **Kruskal's algorithm** in order to to **take advantage of the disjoint-set data structure, Union Find**. We referenced the pseudocode in the Lecture 7 notes to help build out the helper functions: **makeSet**, **find**, **link**, and **unionFind**. We called on these functions inside a separate kruskal's function to calculate whether or not an edge could be added to our MST. We also implemented a couple helper functions — distance & compare — which we used to calculate the Euclidean distance between points, and to sort the edges in order of increasing weight, respectively. The helper functions were created to help with the style and succinctness of our code.

Our program prints out the average MST weight, the maximum edge weight (from all 5 experiments), and the runtime of the program (in seconds). We used the maximum edge weight value to inform the optimizations we performed in "throwing away" certain edges. Prior to optimizing, we ran our program and noticed that our program began slowing down at $n = 8192$ so we made note that the maximum edge weight in dimension 0 where $n = 8192$ was 0.001316. Since Kruskal's is a greedy algorithm, it selects the lowest edge weight that connects one vertex to another. We concluded that no edge weights above 0.005 would be realistically included in the MST, so we optimized by discarding edge weights > 0.005 (not adding them to our edges vector altogether) when $n >= 8192$ and when dimension $= 0$. This holds especially true for larger values of n because as n grows in a complete graph, the number of edges grows, so it becomes increasingly more unlikely that an edge of weight $>= 0.005$ would be included in the MST. We followed a similar line of reasoning for calculating an upper bound to optimize with the other dimensions.

However, as we tested more and more, we realized that this threshold would not be enough leading us

to also further optimize (created tighter bounds) for values $n >= 131072$, throwing away more edges. Subsequent trials supported our previous belief that the maximum edge weight decreases as n increases. Even though we created tighter bounds, these thresholds are still conservative since we are picking the highest maximum edge weight out of all trials. Also, these values change very little between trials.

3 Results

The tables below show the output of the average MST weight over 5 trials for graphs with varying n vertices and different dimensions. Note, trials where $n >= 8192$ were optimized for runtime, and then trials where $n >= 131072$ were even further optimized for runtime.

3.1 Trials

n	Average MST Weight	Dimension
128	1.347122	0
256	1.372775	0
512	1.180944	0
1024	1.183922	0
2048	1.198779	0
4096	1.206583	0
8192	1.208591	0
16384	1.210358	0
32768	1.208401	0
65536	1.204709	0
131072	1.202395	0
262144	1.200226	0

n	Average MST Weight	Dimension
128	7.525617	2
256	10.537251	2
512	15.000311	2
1024	21.384781	2
2048	29.689566	2
4096	41.889542	2
8192	58.978371	2
16384	83.303772	2
32768	117.269875	2
65536	165.735062	2
131072	234.619171	2
262144	332.170807	2

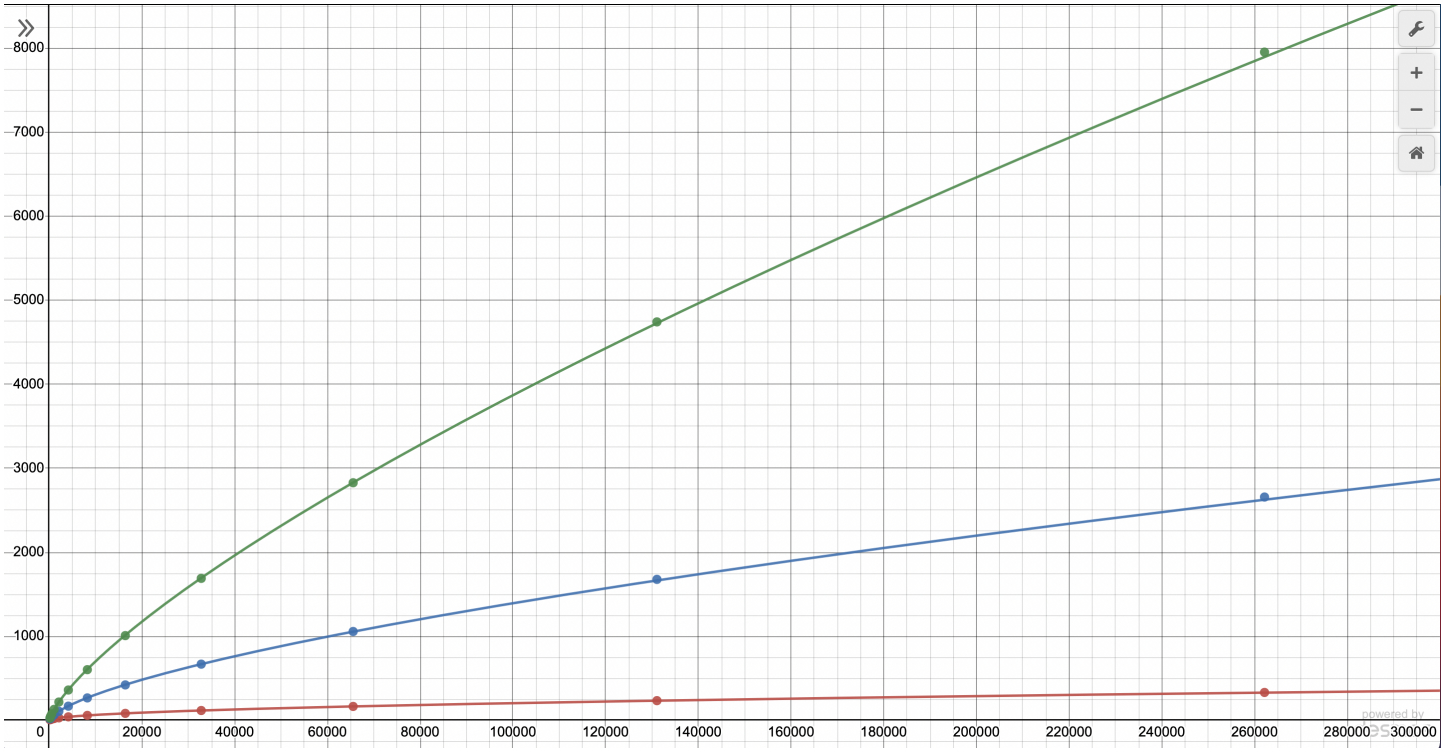
n	Average MST Weight	Dimension
128	17.956575	3
256	27.799046	3
512	43.125492	3
1024	67.324661	3
2048	107.839012	3
4096	169.122269	3
8192	267.160553	3
16384	421.886810	3
32768	668.939087	3
65536	1057.479126	3
131072	1677.787842	3
262144	2657.316406	3

n	Average MST Weight	Dimension
128	27.591467	4
256	48.061668	4
512	77.543266	4
1024	131.050003	4
2048	218.114090	4
4096	361.219025	4
8192	603.306824	4
16384	1008.552856	4
32768	1689.782593	4
65536	2827.417725	4
131072	4741.182617	4
262144	7951.481934	4

3.2 Estimates for $f(n)$

For dimension 0, the average MST weight stayed around 1.20 regardless of n. For dimensions 2-4, the plotted points appeared to model that of a power equation which is for some function $f(n)$, n is raised to some exponent. Thus, by plotting all the points in a stat plot, and using a TI-84 calculator to calculate a power regression equation, we got the following functions

For dimension 0: $f(n) = 1.20$
For dimension 2: $f(n) = 0.677366 * n^{0.496119}$
For dimension 3: $f(n) = 0.720861 * n^{0.657271}$
For dimension 4: $f(n) = 0.772327 * n^{0.739953}$



The graph above plots the average MST weights for increasing values of n (from our trials) against the $f(n)$ equations we found (X axis: n vertices, Y axis: Avg MST Weight). Dimension 2 is shown in red, Dimension 3 is shown in blue, & Dimension 4 is shown in green.

4 Reflection

One of the biggest concerns was the runtime of our program. Originally, as we began conceptualizing how to create and store the edges for our vertices, we coded in an adjacency matrix, then used a separate piece of code to return only the upper right triangle of the matrix (since we have undirected and therefore symmetric graphs). This was inefficient, albeit accurate for the smaller values of n that it loaded. At this point, with accurate code to work off of, we went back in and simplified our edges structure by only ever

generating the upper right hand triangle, foregoing ever needing to generate the adjacency matrix. This increased the space efficiency of our program and thus also improved our time efficiency as well.

Then we optimized by "throwing away edges" as we discussed in the **Reasoning** section. To recap, we threw away edges that we knew would never be in the MST by looking at what the max edge weight, in our MST, typically was. We implemented these thresholds for all dimensions and for values $n \geq 8192$ and $n \geq 131072$. This drastically improved our run time, for trials $n = 8192$. The runtime before optimizing was upwards of 30 seconds (dimension 2), but after optimizing we got runtime down to around 1 second (dimension 2).

As we ran through our trials, values of $n \geq 65536$ were slow in our system; we attempted to change the data structures in order to save space. We had originally implemented both our list of vertices and edges as a vector of vectors. However, after a little research, we decided to switch our data structures to use pairs. Our vertices are now stored in a vector of pairs of pairs and our edges are a vector of pairs where the first element is a float and our second element is another pair. Below is show the change in structure:

1. Original Vector of Vertices: $<< x_1, y_1, z_1, a_1 >, < x_2, y_2, z_2, a_2 >, ... >$
2. Original Vector of Edges: $<< v_1, v_2$, weight $>, < v_1, v_3$, weight$>, ... >$
3. Updated Vector of Vertices: $< ((x_1, y_1), (z_1, a_1)), ((x_2, y_2), (z_2, a_2)), ... >$
4. Updated Vector of Edges: $<$ (weight, (v_1, v_2)), (weight, (v_1, v_3)), ... $>$

Unfortunately, this only marginally improved our runtime, but any improvement counts with significantly large n.

We further attempted to improve our runtime by clearing our computer cache, but again, this only had a marginal effect on the runtimes of our trials. Ultimately, we got all of our trials to run in under 5 minutes, with the exception of $n = 262144$ & dimension $= 4$, which took 15 minutes and 22 seconds to run. This led us to assume that it would also work on graphs with a greater number of vertices, though one would need to change the global variable of MAXN, at the top of our program, to initialize the parent and array vectors correctly.

Note, some of the runtime improvements mentioned were calculated during the optimization process and there are instances where our final code is better optimized than mentioned, but the comparison is still valid and was helpful to our learning during this assignment.

As for the growth rates of our $f(n)$, the growth rate of our dimension 0 graphs surprised us. We weren't expecting the graph to converge to 1.20 with increasing values of n. To explain this, we reasoned it has something to do with the fact that edges were determined randomly, as opposed to being determined by Euclidean distance. As such, the generated edges are fixed between $(0.0, 1.0)$, so the distribution of edges could be more uniform than in the other dimensions? Aside from dimension 0, the behavior of the other $f(n)$ functions seemed reasonable and sensible for the specifications of our program. It makes sense that the more vertices you add, the higher the avg MST weight will be since it takes more edges to create the MST. It also makes sense that increasing the dimension increases the avg MST weight since the maximum possible distance between two vertices increases as the dimension increase. For example, in 2D the max distance between two vertices is $\sqrt{2}$, but in 3D its $\sqrt{3}$.