CS 124 Programming 1: Spring 2022

Name: Emil Liu and Jamin Liu
No. of late days used on this pset: 1

Introduction

For this assignment, we used C++ to generate graphs and identify minimum spanning trees using Kruskal's algorithm. Our experiments were ran on the four following types of graphs:

- Complete graphs on n vertices, where the weight of each edge is a real number chosen uniformly at random on [0, 1].
- Complete graphs on n vertices, where the vertices are points chosen uniformly at random inside the unit square. (That is, the points are (x, y), with x and y each a real number chosen uniformly at random from [0, 1].) The weight of an edge is just the Euclidean distance between its endpoints.
- Complete graphs on n vertices, where the vertices are points chosen uniformly at random inside the unit cube (3 dimensions) and hypercube (4 dimensions). As with the unit square case above, the weight of an edge is just the Euclidean distance between its endpoints.

Implementation

Our project implementation was divided into two major parts: generating and representing graphs of different dimensions, and identify the MSTs of those graphs.

Graph Representation and Generation

To represent our graphs, we used three structures. First, the vertex struct with the following members: four coordinate floating point values (set to zero by default) to represent our four dimensions, a parent property that stores a pointer to another vertex, and finally an integer rank property. The latter two members are to be used in Kruskal's algorithm. Second is an edge struct, which stores a floating point weight and a pair of pointers to the two vertices that the edge connects. Third is a struct for the graph, which stores a vector of all its vertices, a vector of all its edges, the dimension of the graph dim, and the number of vertices n.

To generate this graph, we first initialized n vertices, using our random number generator to produce the coordinates (depending on dim) and storing them with our vector of vertices. Next, we used a two dimensional loop to iterate through our vector of vertices and create an edge between every unique pair of vertices, either by assigning a random weight between 0 and 1 if $dim = 0$, or calculating Euclidean distance as the weight between each pair of vertices.

Kruskal's Algorithm

For our MST finding function, we used Kruskal's algorithm. Our implementation can be decomposed into two subsections, our implementation of union find, and Kruskal's algorithm itself.

First, we implemented union find, creating a makeset, find, link, and union method directly within our graph structure. To make the implementation of these functions more efficient, we stored within each vertex its parent vertex (as a pointer) and its rank within our MST in order to use both the UNION BY RANK and PATH COMPRESSION heuristics.

Next, to implement Kruskal's algorithm, we first took our vector of edges and sorted in increasing order by edge weight (using the default C++ sort function). We initialized a tree for each vertex using makeset() and iterated through our sorted vector of edges. For every iteration, we checked if the vertices of the edge belonged to the same tree by checking if both vertices had the same root using find(). If this condition was not met, we used union() which calls link() to merge the two tree to which each vertex belonged, then proceeded onto the next edge. Once all vertices are contained within a single tree, we would simply pass through each iteration till the loop terminates. The single tree we are left with is guaranteed to be a correct MST because we iterated through our edges from lowest weight to highest weight and through our in-class discussions of Kruskal's.

Design Choice

When approaching our MST algorithm, we chose to implement Kruskal's algorithm instead of Prim's algorithm, which has a similar asymptotic run time. Our decision was motivated by run time efficiency, as well as implementation convenience.

As discussed in class, Prim's algorithm, when implemented using a Fibonacci heap, has running time of $O(|V| \log |V| + E)$ whereas Kruskal's algorithm has runtime of $O(|E| \log |E|)$ if the edge set is unsorted. While for an unoptimized version of the graph, choosing Prim's algorithm would be a much better option as there are n vertices in comparison to the $\binom{n}{2}$ edges in the fully connected graph, by pruning the tree (as discussed later), we were able to greatly reduce the number edges in our graph. As such, for our pruned graph, $|E|$ was close enough to $|V|$that from a run time standpoint, choosing Prim's over Kruskal's was not greatly more efficient.

Because of implementation convenience, we chose to implement Kruskal's algorithm due to its simpler data structures. When implementing union find, we were able to integrate disjoint sets directly into our implementation of our vertex structures, making the algorithm very convenient to implement. If we had chosen to implement Prim's algorithm using a Fibonacci heap, we would have needed to create a separate and more complex Fibonacci heap structure which would have been more difficult to implement while only yielding a minor increase in efficiency.

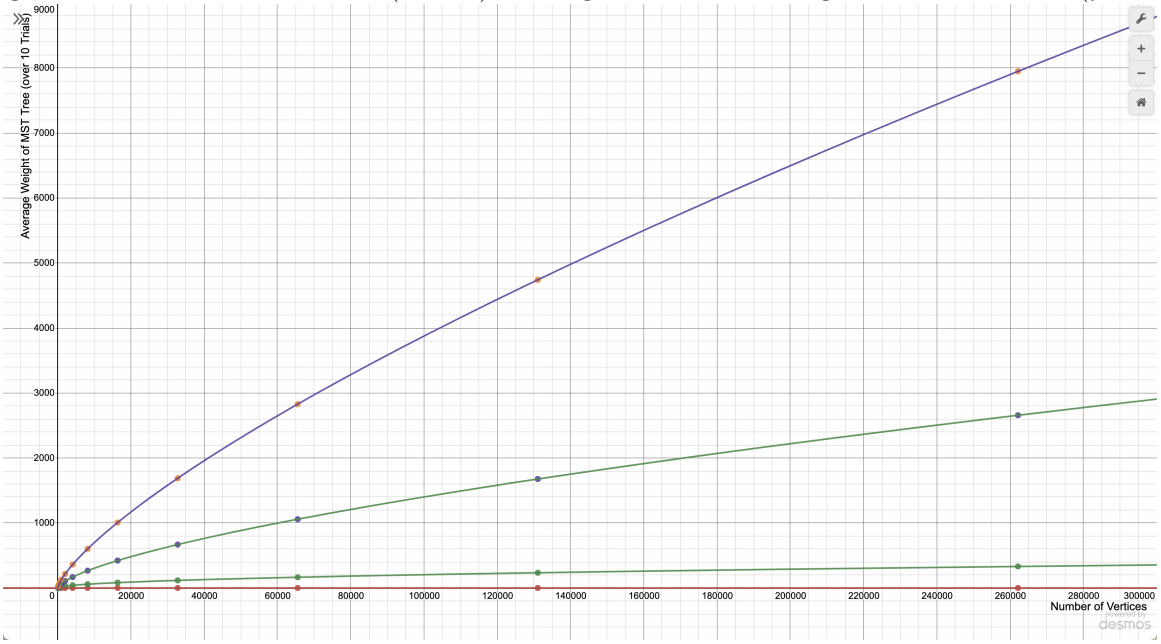
In hindsight, another option we could have explored was to implement Prim's algorithm

with a binary heap. This would have running time of $O(|E|log|V|)$ and might have been slightly more efficient than Kruskal's. It also may have been slightly easier to implement compared to Prim's algorithm with a Fibonacci heap, making a good medium between the two options we weighed.

Quantitative Results

Average Weight of MST Tree (over 10 trials)				
Number of Nodes	0-Dimensions	2-Dimensions	3-Dimensions	4-Dimensions
128	1.18669	7.64298	17.5536	28.3905
256	1.22462	10.7097	27.6163	47.2933
512	1.21076	14.9716	42.9332	78.1959
1024	1.20424	21.0737	67.8646	129.893
2048	1.1875	29.6727	107.537	216.783
4096	1.20491	41.8258	169.279	361.113
8192	1.20275	58.9062	267.346	602.409
16384	1.2009	83.1979	422.492	1008.13
32768	1.20394	117.632	668.851	1688.92
65536	1.20331	166.013	1058.27	2828.48
131072	1.20359	234.663	1676.88	4740.68
262144	1.20496	331.751	2657.47	7948.89

Figure 1: Number of Vertices (x-axis) vs Avg. MST Tree Weight Over 10 Trials (y-axis)



Color Key: Red - 0Dimension, Green - 2Dimension, Purple - 3Dimension, Orange - 4Dimension

After plotting these results and using Desmos to fit a curve to the points, we have the following functions that describes the rate of growth, $f(n)$, for the various graph dimensions:

$$\text{0-d} : y = 1.20$$
$$\text{2-d} : y = 0.65x^{0.5} + 0.33$$
$$\text{3-d} : y = 0.67x^{0.66} + 1.18$$
$$\text{4-d} : y = 0.72x^{0.75} + 2.59$$

Though the 0-Dimension $f(n)$ initially may seem surprising because it is counter-intuitively constant, when thinking more about it, it makes sense. We can think about as, if we have more vertices, we have more edges, and thus more "chances" to roll a smaller weight, and only include the smaller weights in the final MST. Therefore, the total weight of an MST tree should roughly always be the same, on average.

In the other cases, we can see that the rates of growth between the 2nd, 3rd, and 4th dimensions roughly follow the same shape of $a * x^b$, but that each higher dimension grows faster. This makes sense because for each dimension we add, given the same number of vertices, the average Euclidean distance between a pair of vertices is higher, therefore the expected size of the MST tree is higher.

Furthermore, we can make sense of the exponents of the x term by thinking about it like the following. In the three dimensional case, we can conceive of n points spread out roughly evenly in a cube because they are independent and uniformly distributed between 0 and 1. Think about a Rubix cube with points roughly at the corner's of each of its 27 smaller cubes. This means that the distance between these evenly spread out points would be $\frac{1}{\sqrt[3]{n}}$, since the cube's overall side length is 1 and each point has a closest neighbor that is 1/3 distance away from it (in the case of the Rubix cube with side length 1). Then, we have that we have $n-1$ edges, so this turns into $\frac{n-1}{\sqrt[3]{n}} \approx n^{2/3}$. This matches our 3-d result $f(n) = 0.67x^{0.66} + 1.18$ except for the constant term. We can similarly conceive of the 2-d and 4-d versions in the same way, where we have $\frac{n-1}{\sqrt[2]{n}}$ and $\frac{n-1}{\sqrt[4]{n}}$ respectively for those two cases.

Optimization:

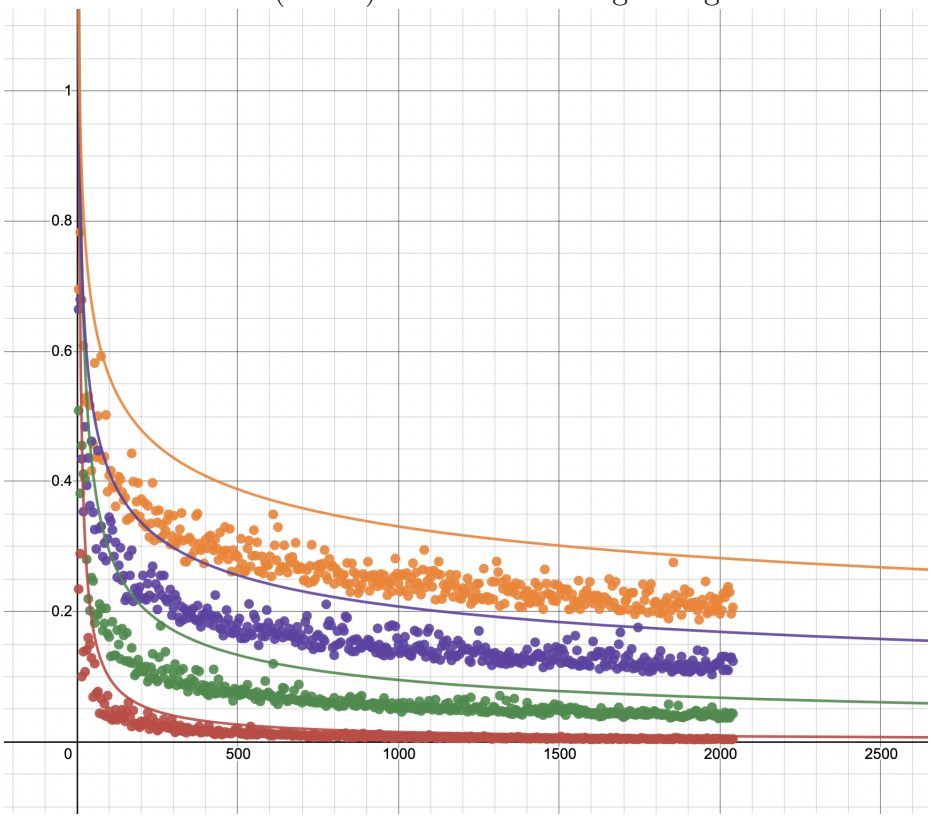
In terms of optimization, we made a couple design choices to increase the space and run time efficiency of our project. First, in our graph representation design, we decided to represent vertices within edges as pointers to store space. In doing this, we only had to store each vertex once when we initialized our vector of vertices, instead of inefficiently storing a vertex in every edge structure. Additionally, rather than create a separate structure for disjoint sets in union find, we added additional parent vertex and rank properties directly to our vertex structure to make it compatible with union find functions.

In terms of run time optimization, we used an O3 C++ tag for optimization. The O3 tag instructs the compiler to prioritize performance over size of machine code generated, leading to a higher performing but less concise version of the program in machine code. Finally, our major source of optimization which was responsible for the most run time optimization was our tree pruning.

Pruning

After running Kruskal's algorithm on the complete graphs with edges between every pair of vertices, it was clear that for large n, our algorithm would take a long time, especially over multiple trials. So, we decided to simplify the graphs by pruning, or reducing the number of edges in the graph representation before running Kruskal's. To implement pruning, we proceed by the following reasoning: the MST is very unlikely to use any edge of weight greater than $k(n)$ for some function $k(n)$. Thus, when constructing our vector of edges to run Kruskal's algorithm on, we can disregard all edge weights greater than $k(n)$ without altering the MST generated. In doing this we will have significantly reduced the number of edges we run Kruskal's algorithm on, and thus, greatly reduce our running time.

Figure 2: Number of Vertices (x-axis) vs Maximum Edge Weight Over 10 Trials (y-axis)



Color Key: Red - 0Dimension, Green - 2Dimension, Purple - 3Dimension, Orange - 4Dimension

To estimate this function $k(n)$, we ran our algorithm and found the average maximum edge weight used in the final MST over 10 trials and plotted it as a function of n, for relatively small n. We tested on n ranging from 5 to 2040 in 5 step increments. Then, we fitted a function a $k(n)$ to these data for each of the four different graph dimensions that acts as a threshold function, making sure that our function rests above all of the data points. Finally, we implemented these $k(n)$ within the graph generation step: when we found that an edge weight, e, was greater than the estimated $k(n)$ for some given number of vertices n, we simply do not add it to the graph. This was an extremely powerful tool to use because of the sheer number of edges that could be pruned. The following are our

estimated $k(n)$ for each dimension (as seen in graph as well).

$$\text{0-d} : y = 4.3x^{-0.82}$$
$$\text{2-d} : y = 2.8x^{-0.49}$$
$$\text{3-d} : y = 1.65x^{-0.30}$$
$$\text{4-d} : y = 1.62x^{-0.23}$$

Finally, to ensure that the results of our optimized algorithm are the same as in an non-optimized version, we consider the following problem. Imagine we get unlucky and we roll many consecutive high edge weights (0.990, 0.991, ...), and therefore that all these edges are pruned by our $k(n)$ function, and there exists no valid MST on our graph anymore. To catch this edge case, we simply check at the end of our Kruskal's algorithm that the length of the MST we return is the number of vertices - 1, which is a property of trees in general. If this is not the case, then we know we have pruned too much - therefore, we simply rerun the generation of all edges, this time accepting all edge weights, and rerun Kruskal's. This ensures that, in the slim chance that we do prune too much, we still return the correct answer.

Below are some results of the number of edges we pruned in the 2-Dimension case:

Number of Edges Pruned per Trial (in 2-D case) on Average over 10 trials

Number of Nodes	Edges Total	Edges Pruned
128	8128	6800.6
256	32640	29720.8
512	130816	124423
1024	523776	510438
2048	2.09613e+06	2.06809e+06
4096	8.38656e+06	8.32946e+06
8192	3.35503e+07	3.34331e+07
16384	1.3421e+08	1.33969e+08
32768	5.36855e+08	5.36364e+08
65536	2.14745e+09	2.14645e+09

Efficiency

Average Runtime Per Trial to Generate Graph and Run Kruskal's Algorithm in Microseconds (over 10 trials)				
Number of Nodes	0-Dimensions	2-Dimensions	3-Dimensions	4-Dimensions
128	363	174	150	176
256	918	403	382	439
512	2115	1054	1023	1179
1024	5068	2703	2862	3268
2048	15882	7652	8362	9493
4096	59281	24946	26700	30297
8192	231265	85833	92625	98264
16384	893106	317755	337701	344302
32768	3534376	1211214	1252497	1266720
65536	14048853	4713470	4838188	4808866
131072	56168735	18609977	19113373	18754444
262144	223853310	73538138	73990511	78171352

Overall, we found our project to be relatively efficient (we were able to generate test results in a reasonable amount of time). We can attribute this efficiency to two main sources. First is the O3 compiler tag we used when running our project, which instructed the compiler to prioritize performance over size of machine code generated. When running our tests without this tag, we found our code took many times longer to run. Our second source of increased efficiency was graph pruning, which significantly reduced the number of edges we ran our algorithm on and allowed it generate results faster.

Looking at the above table of run times, we can identify a couple patterns in our run time data. Looking at run time efficiency by dimension, it appears that dimensions 2-4 all had similar run times, with run times slightly increasing with dimension (2D runs a little faster than 3D which runs a little faster than 4D). These discrepancies, however, do not seem statistically significant and could be explained by our choice of our respective $k(n)$ pruning function for that dimension.

Looking at run time efficiency of dimension 0, however, we note that it is higher than those of other dimensions. One potential explanation for this is that the cutoff function for dimension 0 does not disregard as many edges as the cutoff function for higher dimensions. This could be because edge weights in dimension 0 are distributed across the more restrictive range, causing the cutoff function to have a proportionally higher cutoff threshold compared to other dimensions. As such, the MST function would have to iterate through more edges, leading a longer running time.

Below is the runtime of our code for 2-d graphs without O3 and with O3 optimization.

Average Runtime Per Trial to Generate Graph and Run Kruskal's Algorithm in Microseconds for 2-Dimensional Graph (over 10 trials)		
Number of Nodes	Without O3 Flag	With O3 Flag
128	772	174
256	2406	403
512	8135	1054
1024	29441	2703
2048	110648	7652
4096	429896	24946
8192	1696724	85833
16384	6687041	317755
32768	26635303	1211214
65536	107251373	4713470
131072	425433323	18609977
262144	1697685762	73538138

Randomness Function

To generate a random number between 0 and 1 inclusive, we first generated a seed using the *srand* C++ function on the current time at the start of our program, and then started generating the random number between 0 and 1 using $rand()/RAND - MAX$. Originally, we made the mistake of generating the random seed for every trial run, which turned out to cause the same random sequence to be generated when the time between trials was the same. Then, we learned about how rand and srand worked and changed the seed to only generate once at the beginning. Although it doesn't seem to be the most random number generator out there, because it's pseudo-random (meaning for the same seed, it will generate the same sequence), it is suffice for our purposes, especially when we seed it properly at the beginning of the program.