Yihua Leng, Daniel Schwartz
CS124
2/23/2021

Programming Assignment 1: Minimum Spanning Tree Average Weights

Introduction

Our goal was to determine how the average weight of an MST grows as a function of n (nodes) in an undirected graph G. We used C++ to create complete graphs with 0, 2, 3, and 4 dimensions, and then we implemented Kruskal's algorithm to find the weight of the Minimum Spanning Tree (MST). We figured the runtime of Kruskal and Prim's algorithms would be very similar: ElogE and ElogV respectively. So we used Kruskal's because we were interested in implementing a disjoint set data structure and Union Find, which neither of us have implemented before. In doing this, we may have neglected the time to build the graph in Kruskal's algorithm, which we will discuss in more detail below.

Implementation

We first needed to create a function that creates an undirected graph with random weights. This required a struct called graph that is composed of an unordered_map and a vector. The unordered_map has the key as the edge weight and the value as a vector of vertex pairs who share that weight, each edge in the form of a pair (u,v). And then the vector of the graph contains the weights of the graph.

Once we made the graph struct we tackled the case that our dimension is equal to 0 and we want a complete undirected uniformly random weight graph. We have a double for loop so that each node has an edge going to every other n-1 nodes to satisfy the complete graph requirement. To randomize the weights, we used the C++ rand() function and the srand() function. We seeded the srand() function with the current time in order to ensure our best pseudo-randomness. Then we divided this value by RAND_MAX to obtain a float value between 0 and 1. We inserted this weight into our unordered_map (graph.dist_graph) if not already there and pushed our random edge into the corresponding vector so that dist_graph[weight] = vector[(u,v),...,(x,y)]. With the new random weight, we also added this weight to the graph.edges vector.

For the other dimensions 2, 3, and 4, we created a function called create_uniform_vertices that, given a dimension, node count, and random seed, outputs (node_count X dimension) random vertices on a unit square, cube, or hypercube in the form of an array of arrays. Now that we have these vertices, we can generate a graph with them using vertex u and vertex v's Euclidean distance as the edge weight.

Now that we can create a graph given n vertices and a dimension, we can begin to implement Kruskal's algorithm. We first had to implement the disjoint set data structure, so we needed

Union Find functions. For Find, we do two things. First, we recursively call the Find function to point the node to its parent, implementing path compression. Path compression is an important optimization to minimize our tree depth. Second, we return the node's parent. For Union, we arbitrarily point parent x to parent y in the case that their ranks are equal. Otherwise, we point the tree with the lower rank towards the tree with the higher rank in order to minimize our tree depths and speed up the Find method.

Now, in our Kruskal function, we could initialize the parent and rank array and follow the steps of Kruskal's algorithm while our MST edges were less than |V| - 1 since an MST will always have |V| - 1 edges. We also had to sort the edge weights in our graph.edges vector, which we did with the C++ sort function. This sorting is important because for each edge weight, we want to add the lowest weight edges first and then search the unordered_map for that edge weight to get the edges associated with this current lowest edge weight. Then, if the parent of two nodes x and y was different then we could Union parent(x) and parent(y). An important optimization we made here was to check if our current edge weight was the same as our previous edge weight as we go through the sorted vector of edge weights. If it was, we could skip that edge weight index. This is because we only need to visit each edge weight once—for a given edge weight, we search every edge associated with it in the map.

Now that we had the algorithm in place, we timed each part of the function to see what we needed to optimize. After timing the creation of the graph, the sorting of the weights, and other parts of Kruskal's algorithm, it appeared that creating the graph was running about double the time as Kruskal's algorithm and was the main inhibitor of speed.

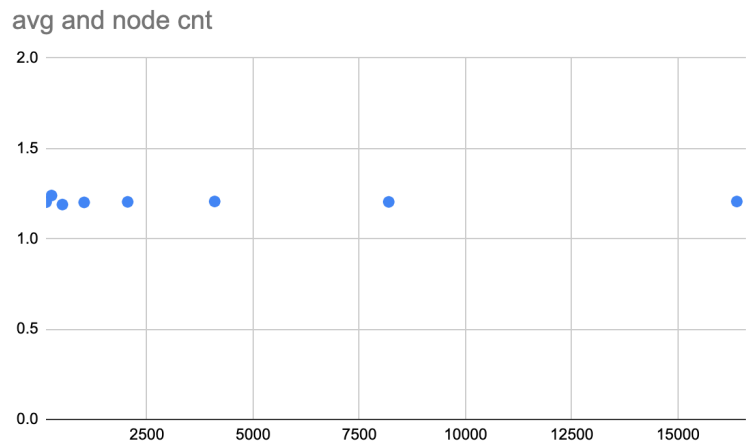
We consulted the hint that was given in the instructions to find a value k(n) that our edge weight would likely not be greater than. To do this, we created a function called cutoff_func that estimates the maximum edge weight in our MST. We noticed that the maximum MST edge weight decreased as the node count increased, but it increased as the dimension increased. Therefore, we decided to build a linear model using these two features against maximum MST edge weights which we obtained from the MSTs on 2^7 to 2^13 nodes with 1 to 4 dimensions. In the end, we had cutoff_weight = -0.02298154*log2(node_cnt) + 0.07643103*dim+0.3. This cut-off function worked pretty well except for 2^17 and 2^18 nodes on 1 dimension. This is due to the fact that our linear function was fit within the node count range 2^7 to 2^13, and 2^17 might be too far away to get the accurate estimation. However, after we manually changed the intercept to 0.35, it worked.
Now that we have this value for a given number of nodes and a dimension, we put a condition in our create_random_graph function so that an edge would only be added to graph.dist_graph if the edge weight was below this weight.

Outputs

Dimension = 0

Avg mst weight	node cnt	trails	dim
1.202621	128	5	1
1.239329	256	5	1
1.189325	512	5	1
1.20134	1024	5	1
1.20419	2048	5	1
1.206446	4096	5	1
1.204029	8192	5	1
1.206246	16384	5	1
1.200541	32768	5	1
1.202711	65536	5	1
1.202379	131072	5	1
1.205341	261244	5	1

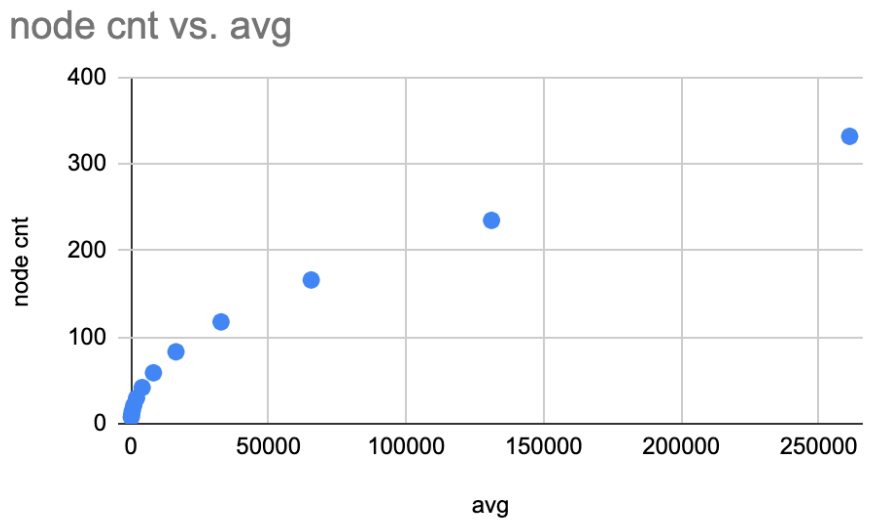
constant at 1.2



Dimension = 2

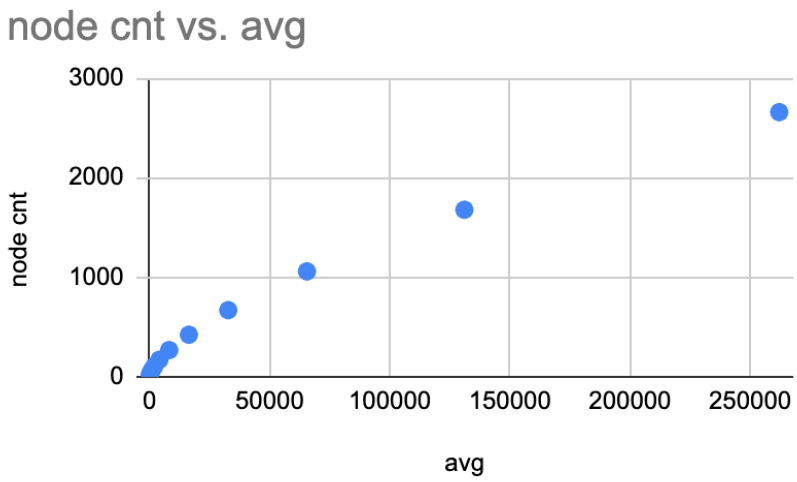
Avg mst weight	node cnt	trails	dim
7.65462	128	5	2
10.898884	256	5	2
15.067772	512	5	2
21.152227	1024	5	2
29.633859	2048	5	2

41.769656	4096	5	2
58.829882	8192	5	2
83.052014	16384	5	2
117.561844	32768	5	2
165.919409	65536	5	2
234.687828	131072	5	2
331.625912	261244	5	2



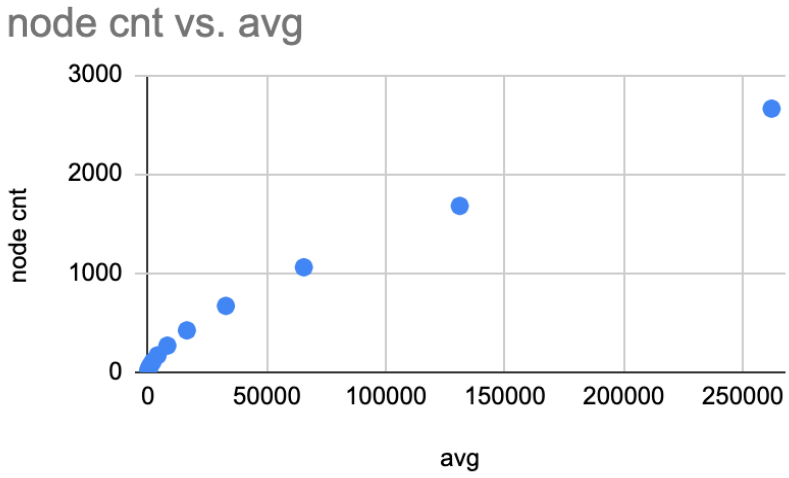
Dimension = 3

Avg mst weight	node cnt	trails	dim
17.460399	128	5	3
27.573809	256	5	3
43.212453	512	5	3
68.183257	1024	5	3
107.107274	2048	5	3
169.965845	4096	5	3
267.747077	8192	5	3
421.928815	16384	5	3
668.673187	32768	5	3
1058.199663	65536	5	3
1677.362756	131072	5	3
2658.554827	262144	5	3



Dimension = 4

Avg mst weight	node_cnt	trials	dimension
28.478515	128	5	4
46.877955	256	5	4
78.314454	512	5	4
360.87803	4096	5	4
602.194988	8192	5	4
1008.79858	16384	5	4
1688.587626	32768	5	4
2828.812337	65536	5	4
4739.968131	131072	5	4
7950.926217	262144	5	4

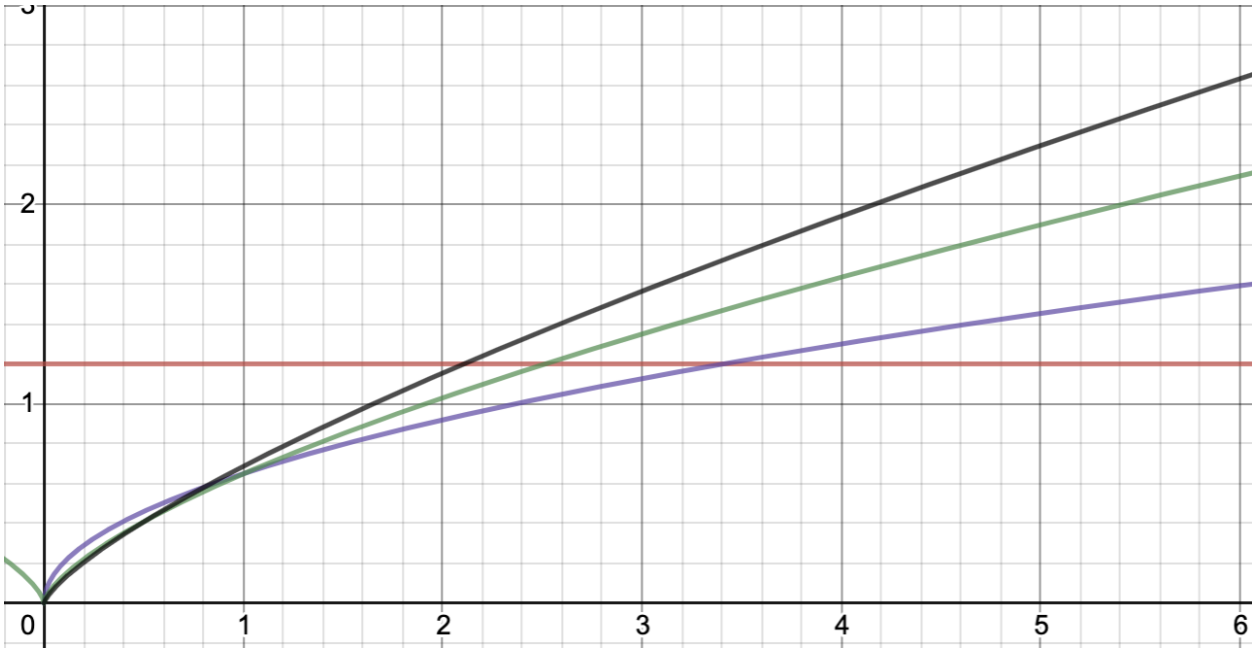


The first thing we notice when looking at these results is that as the node count increases, the average MST weight monotonically increases, except when dimension = 0. Also, as dimension increases and node count stays constant, average MST weight increases. For dimensions = 0, we were at first surprised that the average didn't increase with nodes, but giving it some thought, we realize that since we were only operating between 0 and 1, as node count increases, it is more likely that the edges between nodes will become shorter. So the number of edges is increasing while edge length is decreasing. For dimensions = 2, 3, and 4, the function is increasing because on these multidimensional figures, you have multiple dimensions to place vertices and your space is less inhibited, so random vertices will continue to add a significant amount to the total MST edge eights. This is the same reason that increasing the dimensions shallows out the curve. With more dimensions, you are inhibiting space less.

Our guess for the function f(n) varies by dimension (as you can see in the graphs above), so we have a guess for each dimension. The graphs appear to be a polynomial because it grows faster than a linear function. So we fit the curve using polynomial regression in the form of $a * (n^{b}) + c$, and we ended up with the following:

Dimension = 1	$f(n) = 1.2$
Dimension = 2	$f(n) = 0.65 * \sqrt{n} + 0.41$
Dimension = 3	$f(n) = 0.6489 * n^{2/3} + 2.757$
Dimension = 4	$f(n) = 0.6862 * n^{3/4} + 8.8130$

Graph with y intercepts at 0 to better visualize the difference in growth rates



Legend

Dimension 1

Dimension 2

Dimension 3

Dimension 4

Conclusion:

This turned out to be a pretty challenging problem in terms of optimizing runtime. Not only did we have to apply things we learned for optimization like the disjoint set data structure, but we had to apply new techniques like our max cutoff technique. It was a great learning experience in terms of realizing a program you're building can (almost) always be optimized and changed. A great example of this occurred towards the end of this project while we were eagerly waiting for the 262144 node count to execute. We realized Prim's algorithm would likely be even faster than ours because we wouldn't need to build the entire graph at the start. We could just generate edge weights as we go. And it turned out that our function for building the graph accounted for about 98% of the running time, so this would be useful. The long graph building runtime makes sense because we were building a connected graph which has a runtime of

$O(n^2)$. Another point of optimization could be making sure the weights in our edge weight vector were all unique, so we could possibly make it a set rather than a vector. This would speed up the sorting algorithm, as we only ever need the unique weight values.

The ongoing optimizations and improvements, like those above, are what makes programming exciting and what makes me optimistic for the future of algorithms and engineering in general.