CS 124 Programming Assignment 1:
Matt Tengtrakool and Nikhil Datar

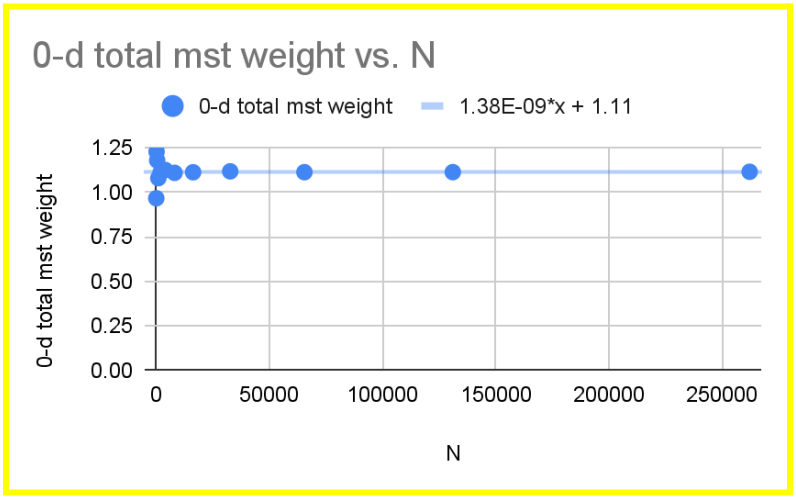
Part 1: Overall Weight of Each Coordinate type:

Total Weights				
N	0-d total mst weight	2-d total mst weight	3-d total mst weight	4-d total mst weight
128	0.966392	7.65285	17.1652	29.2981
256	1.22783	10.3561	26.8715	46.3699
512	1.17915	15.186	44.1782	77.973
1024	1.07943	20.9513	68.5436	128.146
2048	1.11185	30.015	107.676	215.794
4096	1.12489	41.6698	169.574	358.861
8192	1.1091	59.11	267.364	603.785
16384	1.11238	83.0274	423.433	1009.15
32768	1.11748	117.294	668.345	1685.43
65536	1.11275	165.823	1057.94	2832.15
131072	1.11305	234.473	1677.01	4746.92
262144	1.115	331.742	2656.37	7954.58

Function guesses:

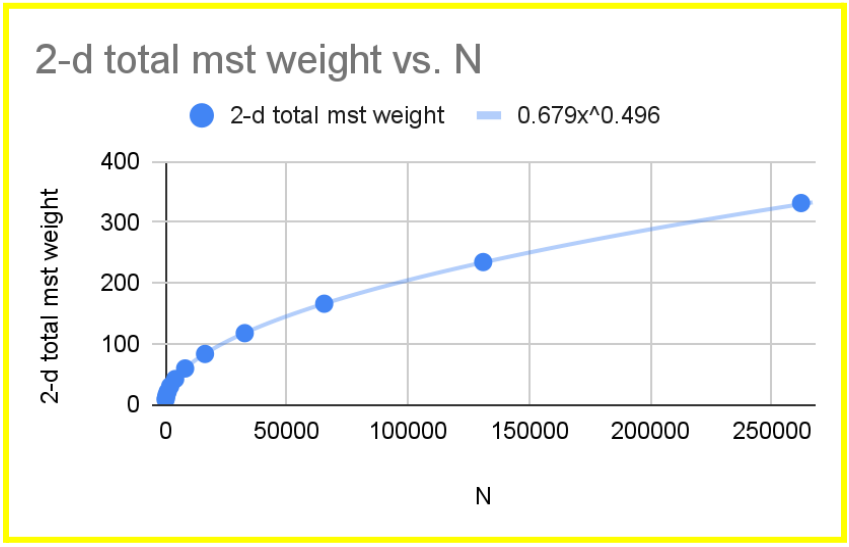
1. 0-D: 1.38E-09*x + 1.11

a.



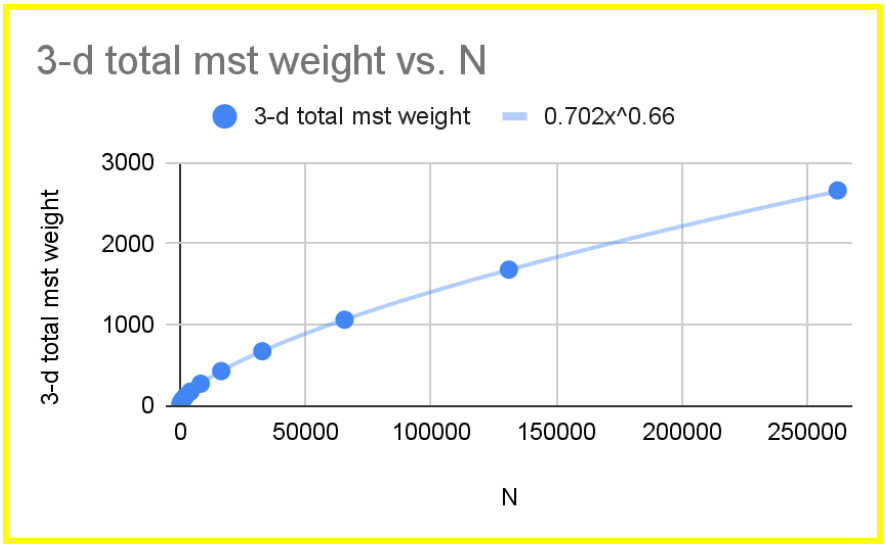
2. 2-D: 0.679x^0.496

a.



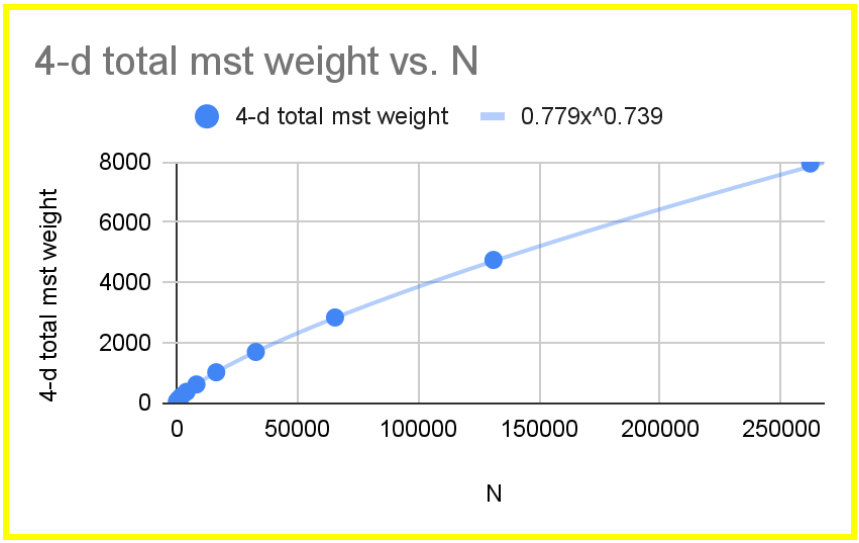
3. 3-D: 0.702x^0.66

a.



4. 4-D: 0.779x^0.739

a.



Part 2: Discussion

In order to find the overall weights in each of the scenarios, we decided to use a combination of Prim's and Kruskal's algorithm. In the lecture 6 notes, it discusses that Prim's algorithm performs well on densely connected graphs and maintains a runtime of O(log |V|) for deletes, O(log |V|) for insertion, and O(E log |V|) overall for our implementation with a binary heap, as we must consider a delete/insertion for every one of the E edges within the graph. As previously stated, we implemented the algorithm using a binary heap data structure because on the dense graphs with no edges removed, binary heap was a fair mix in terms of runtime between using a linked list or a more dispersed heap such as a d-array heap. Furthermore, the changes we had in the algorithm were simply that we represented edges as a vector of unordered maps rather than an adjacency list, thus yet they still preserve correctness of the algorithm because the overall implementation of the algorithm itself follows the procedure given in lecture 6, just with a different data structure used.

On the other hand, Kruskal's algorithm performs well on more sparse graphs, with a runtime of O (E log E), because we are sorting the edges in increasing order by the weights. Furthermore, the changes we had were the data structures we used to store the graph, priority queue, and MST information (in particular, we used a vector of unordered maps/hash tables, a disjoint set structure for the priority queue, and edge adjacency list for the MST). However, the correctness is preserved because the algorithm functions in the same manner as discussed in the lecture 6 notes, and thus can be proved similarly.

The unordered maps we used to represent the input graph where each element in the vector corresponds to a node and it stores the edges adjacent as well as the edge weights. This allows us to index edges adjacent to a node in O(1). Additionally, we used a disjoint set to check connections between components and merging sets. Most importantly, we used an edge adjacency list to store the edges in the MST which also provides us with O(1) indexing. The use

of these data structures allows the Kruskal's algorithm implementation to efficiently process large graphs and compute their minimum spanning trees.

Overall, we used Prim's to get an initial sense of the max weights chosen for each value of n, prior to edge removal because it is faster on dense graphs. However, once we decided on an optimal function for pruning via regression models, we switched over to Kruskal's algorithm, since the number of edges was reduced in the pruning process, making the graph more sparse and Kruskal's algorithm more efficient (with the number of edges reduced).

One difficult item we had to deal with in the process of shifting to higher dimensions (3rd and 4th dimensions) was the fact that we were calling several math functions (power, sqrt) hundreds of thousands to millions of times initially, to calculate directly the Euclidean distance. Instead, we eventually shifted to using an approximation for the Euclidean distance to check if a certain threshold was met. Particularly, we used the Manhattan approximation distance conversion to compute an estimate for the distance between 2 vertices solely using additions. In the case where the 2 edges met the threshold for the Manhattan distance, we then actually calculated the Euclidean distance and stored that information, severely reducing the number of mathematical functions called and speeding up the process of calculating the sum of the edge weights.

The growth rates we achieved followed expected patterns for increased total weight as dimensionality increased as well as relationally with the number of nodes.

	0-d runtime	**2-d runtime**	**3-d runtime**	**4-d runtime**
f(n) =	1.38E-09*x + 1.11	0.679x^0.496	0.702x^0.66	0.779x^0.739

For the 0-D case, the growth rate is essentially constant. This shows that the total mst weight does not depend on the number of nodes, due to the average edge weight decreasing as the number of nodes increased. For the 2-D, 3-D, 4-D they all fit a power series. Something that we found surprising was that total MST weight is not proportional to an increase in the dimensionality. As the number of dimensions in the input space increases, the number of possible configurations grows larger and there is less data clumped together leading to higher overall mst weight as dimensionality increases. Additionally, there are just overall a higher amount of vertices and edges for every increasing dimension. Another interesting observation we made is that the growth rates of the MST weights for the 2-D, 3-D, and 4-D cases are superlinear, meaning they grow faster than a linear function. This suggests that the increase in the number of vertices and edges in higher dimensions outweighs the decrease in edge weights due to the increased space available, resulting in a net increase in MST weight. This makes sense, as in a higher dimension since the vertices have more dimensions that are uniformly distributed [0, 1], the edges can have greater distance between them. For example, in the 0d setting the edges have a max distance of 1, but in the unit square setting the edges have a max distance of sqrt(2).

For each example, the runtime is summarized in the table below.

Runtimes (s)				
N	**0-d runtime**	**2-d runtime**	**3-d runtime**	**4-d runtime**
128	.135	.280	.285	.148
256	.137	.267	.256	.142
512	.150	.139	.151	.144
1024	.140	.279	.138	.142
2048	.174	.144	.153	.155
4096	.362	.143	.165	.158
8192	.400	.195	.201	.212
16384	1.242	.132	.279	.322
32768	4.073	.682	.555	.724
65536	15.403	1.499	1.719	2.283
131072	60.7	6.020	6.180	8.428
262144	242.33	22.884	23.845	32.441

This, in general, makes sense as we see that the runtime increases as the dimensionality of the vertices increases. This is because with each increase in dimensionality comes a small increase in the number of mathematical operations and function calls, especially when we run the program with the optimized flag. In particular, we fit a power series for each of the 2-D, 3-D, and 4-D dimensions for pruning, so these were more (over)fit to the exact max edge and ran faster than the estimated function for 0-D. The computer cache had an effect in that it allowed certain programs to run faster for particular values of n because the code was already compiled and data was already cached. Thus, as the size of the cache increases on the computer less memory access called are made and the data is retrieved faster.

Additionally, we had an interesting experience with the random number generator. We used some basic arithmetic in conjunction with the rand() function from the <cstdlib> library. When we did so, we noticed that the random numbers essentially created a seed automatically and would output the same values across trials. Thus, the random numbers weren't necessarily random at all times, meaning we can't fully trust the implementation of the random number generator. This had the effect of potentially allowing for overfitting to the data in terms of the optimal functions we fit for cutting edges, as we fit and tested essentially the same data across trials because the random number generator seeded itself for particular values.