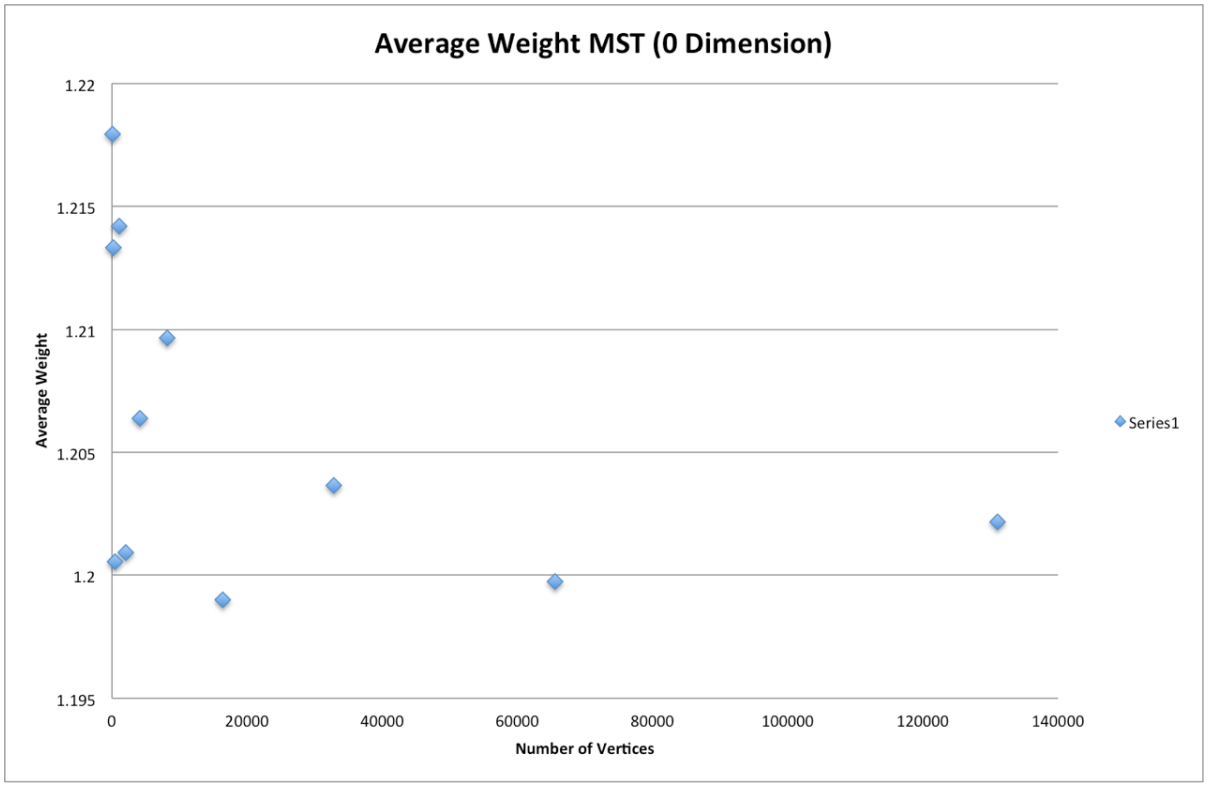
I. Data

a. Average Weight of MST (0 Dimension, 5 trials per Data Point)

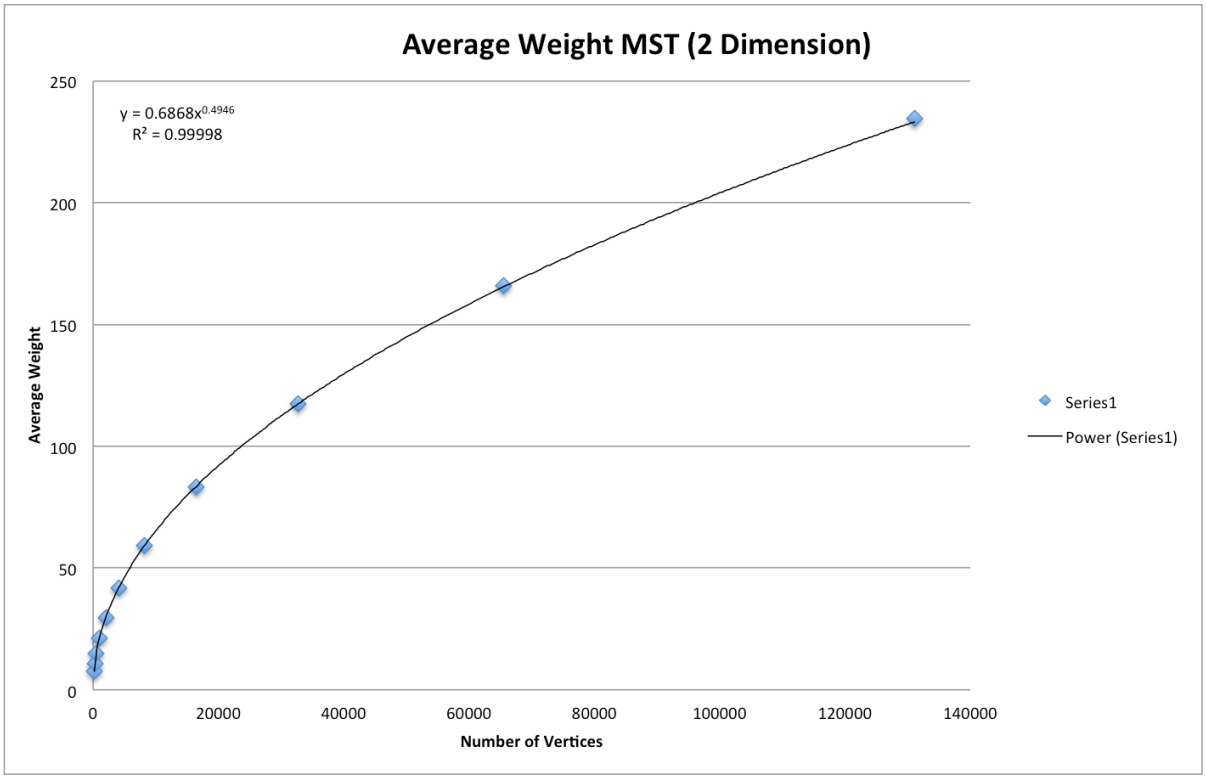
Vertices	0D - Average Weight (5 trials)
128	1.21795
256	1.21333
512	1.20055
1024	1.21421
2048	1.20092
4096	1.2064
8192	1.20967
16384	1.199
32768	1.20366
65536	1.19975
131072	1.20216



The function f(n) is of the form f(n) = k, where k ≈ 1.2

b. Average Weight of MST (2 Dimension, 5 trials per Data Point)

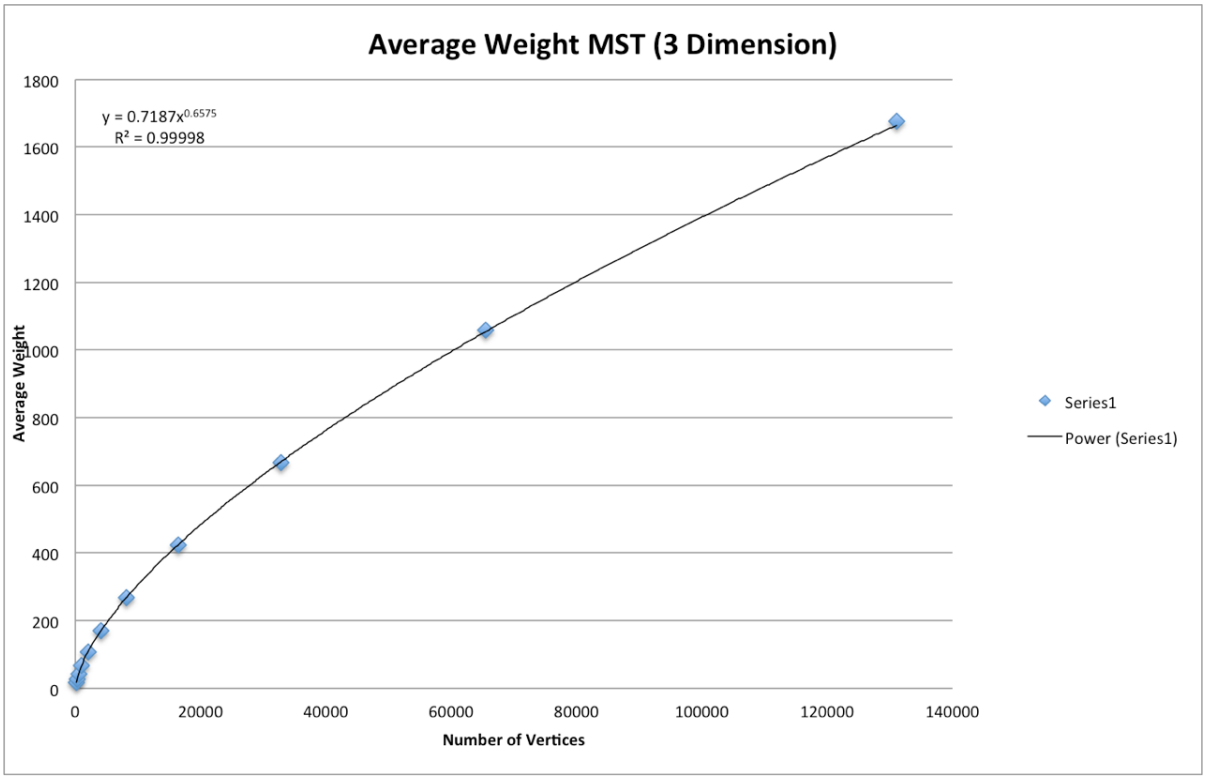
Vertices	2D - Average Weight (5 trials)
128	7.62073
256	10.6456
512	15.0999
1024	21.0682
2048	29.7036
4096	41.7452
8192	59.0762
16384	83.1878
32768	117.533
65536	165.887
131072	234.541



The function f(n) is of the form f(n) = $k\sqrt{n}$, where k ≈ 0.69

c. Average Weight of MST (3 Dimension, 5 trials per Data Point)

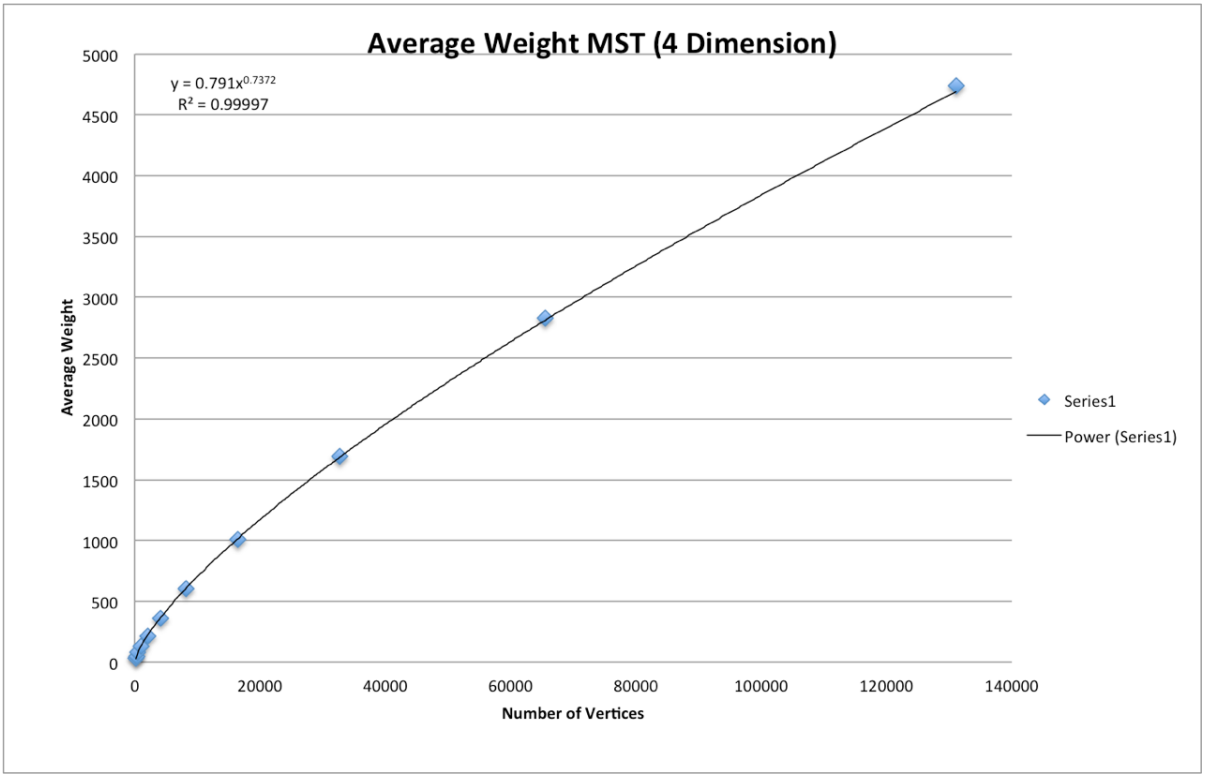
Vertices	3D - Average Weight (5 trials)
128	17.5491
256	27.8689
512	43.2417
1024	67.9718
2048	107.397
4096	170.13
8192	267.622
16384	423.11
32768	668.09
65536	1058.3
131072	1677.16



The function f(n) is of the form f(n) = $k(n^{2/3})$, where k ≈ 0.72

d. Average Weight of MST (4 Dimension, 5 trials per Data Point)

Vertices	4D - Average Weight (5 trials)
128	28.7787
256	47.2818
512	78.5219
1024	130.335
2048	215.778
4096	361.181
8192	603.062
16384	1008.85
32768	1688.86
65536	2826.63
131072	4740.86



The function f(n) is of the form $f(n) = k(n^{3/4})$, where $k \approx 0.79$

II. Discussion

a. Algorithm and Implementation

We chose to implement Prim's algorithm over Kruskal's algorithm because the former is significantly faster in a graph with lots of edges. Given that we were working with fully connected graphs in which all vertices were connected to one another, that meant we had to handle graphs $G = (V, E)$ with $E = {}_V C_2 = \frac{V(V-1)}{2}$ edges for V vertices. Prim's algorithm runs in $O(E + V\,logV) = O(V^2)$ time after substituting in our expression for E, since our implementation does not use a binary heap and only uses arrays (iterating over vertices twice). Kruskal's algorithm runs in $O(ElogV) = O(V^2 logV)$ time. Thus, we chose Prim's because it would perform better in very dense graphs, especially as the number of vertices scaled up. In addition, for memory management purposes, we did not want to implement Kruskal's algorithm because that would entail sorting and storing all of the edges, whereas our implementation of Prim's does not store all of the edges.

First, we seeded the random number generator with the current time and used the built-in rand() function. We normalized the outputs of the function to ensure they were between 0 and 1 for all four cases and cast them to floating point values.

Next, to generate graphs for the 2D, 3D, and 4D cases, we created an array $coords$ of size V of tuples of 2/3/4 floats (based on the dimension) and generated random coordinates for each vertex. We initialized an array $visited$ of booleans of size V that would track whether or not a vertex had been added to our MST, and an array $costs$ of floats that would store the key for each vertex, in this case the edge cost to add a vertex to our MST. All values in $visited$ were initialized to false and all values in $costs$ were initialized to infinity (FLT_MAX). The start node (index 0), however, is initialized to true and its cost is 0, since we must begin with one node as our first vertex in the MST.

Afterward, we run Prim's algorithm until our MST contains all the vertices/all vertices have been visited by looping V times. Within each loop, we pick the vertex with the lowest cost not yet in our MST. We add it to our MST by marking it as visited and adding its cost to the running total weight of our MST. Then we update the edge weights for all vertices adjacent to that vertex (though since this graph is fully connected, this is just iterating over all vertices in an inner loop). For each adjacent vertex that hasn't been visited (and therefore isn't in our MST), we calculate the edge weight between the two vertices, and if that value is less than the current cost of that adjacent matrix in $costs$, we update the new key value to be the calculated edge weight. For the 0D case, the edge weight was a randomly generated floating point value between 0 and 1 (since MSTs must be acyclic, we do not have to store this value and worry about recalculating it), and for the rest of the cases, the edge weight was the Euclidean distance between the two vertices' coordinates. Calculating these edge weights rather than storing all of the weights and the edges at the beginning saved us memory.

Once the outer loop over all the vertices finishes running, we now have the total weight of our MST and can average the weight per trial. To run a given number of trials, we wrapped our code in an outer loop, and we also timed each trial to get the average runtime.

Side note on running our code: We submitted a Makefile along with our code. Our code compiles and runs locally on our computers, but it appears nice.fas.harvard.edu does not support C++11 (same issue as this Piazza post). To run this code, we needed version 4.8 of g++ and the g++ flag -std=c++11 (g++-4.8 --std=c++11). In case of any issues that may arise from testing our code locally, we set up a Cloud9 workspace at https://ide.cs50.io/lisalu/cs124.

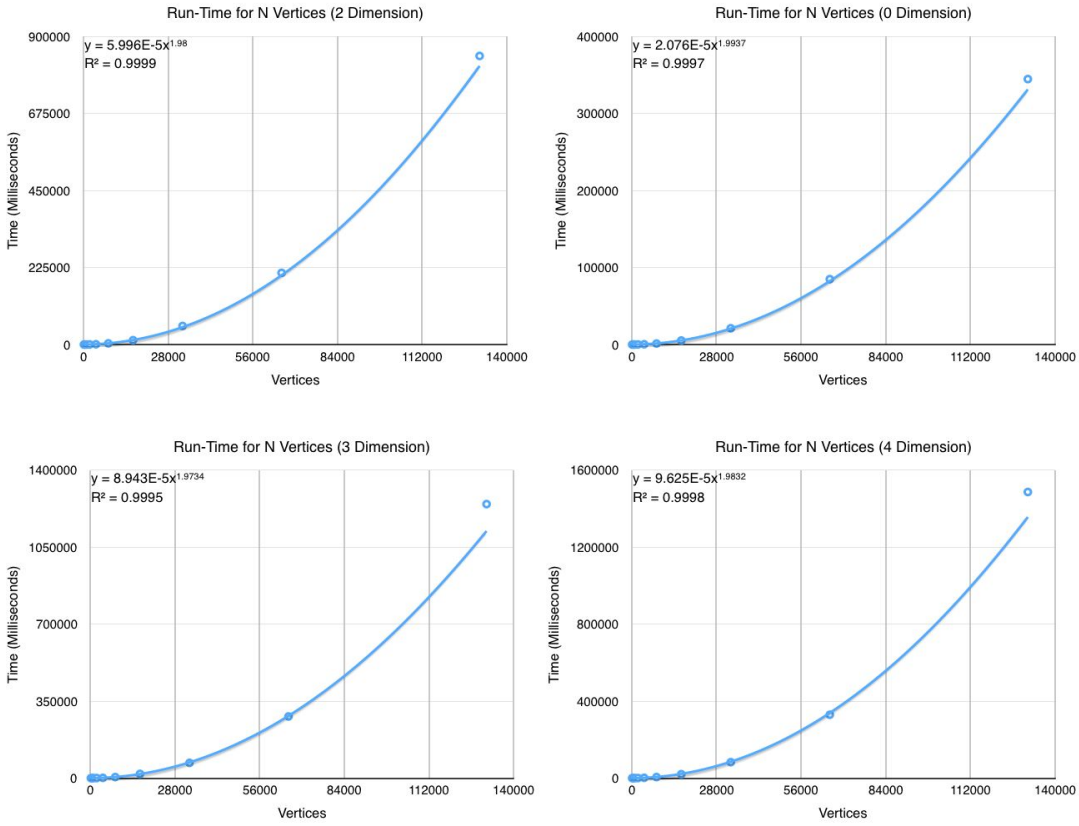
b. Growth Rates

In I. Data, we made tables and graphs of the average weight of MSTs based on five trials for each number of vertices and dimension. We then tried to determine $f(n)$ to describe the average weight of an MST for n vertices.

For the 0D case, even as the number of vertices increased by powers of 2, the average weight stayed approximately the same (around 1.2), so we guessed $f(n) = 1.2$. These results surprised us—our hypothesis as to why the values converge instead of increasing with respect to the number of vertices is that even though the number of edges is increasing, the edge lengths/weights should decrease proportionally as more vertices crowd the (0, 1) range from which we are generating random numbers. Therefore, the average weight stays around the same.

For dimensions 2, 3, and 4, we determined $f(n)$ via regression to get a best-fit curve with Microsoft Excel, all of which are listed above in I. Data. Interestingly, all the functions were of the form $f(n) = k(n^{\frac{d-1}{d}})$, where n is the number of vertices, d is the dimension, and k is a constant. It makes sense that the average edge weight of the MSTs go to infinity as the number of vertices goes to infinity for the three dimensions, the more vertices there are, the more edges there are to add to our MST. If each edge is an infinitesimally thin 1D line segment, then the area of the unit square or the volumes of the unit cube/unit hypercube will not bound the total weights of all the line segments since there will be infinitely many as you approach infinite vertices, unlike the 0D case. Why the exponents happen to be $\frac{d-1}{d}$, we are unsure of, but clearly larger dimensions entail larger edge weights simply because there is a greater range to generate random coordinates from and therefore greater Euclidean distances possible (for instance, the diagonal of a unit square is obviously less than the space diagonal of a unit cube), and as the dimension approaches infinity, the function approaches a linear one.

c. Run-Time



Above we graphed the average run-time per dimension against the number of vertices in the graph using information from the same trials in I. Data (5 trials per configuration). After fitting best-fit curves in Excel, it is clear that the time and number of vertices have an approximately quadratic relationship, which is what we expected when determining our algorithm to have time complexity $O(n^2)$ for n vertices. The largest case for 4 dimensions and 131072 vertices took, on average, 24 minutes per trial. That said, we obtained the data for the average weight of the MSTs and the run-time from trials performed on Cloud9's cloud Ubuntu workspace (which was chosen for easy collaborative coding and editing), and that has less computing power than running it on our own computers (for which it did run more quickly).

d. Random Edge Weight Generation

To fulfill the requirement of random vertices in the implementation of each graph, we used the function srand() and fed it the current time from the system clock (function: time()) to

ensure that each time the values were initialized, they were unique. We then manipulated the random number to be between 0 and 1 using RAND_MAX. We only seeded the random number generator once at the beginning of our main() function.

e. Future Directions

Our algorithm effectively handles large values of n, but it could be optimized to do this faster by storing edge weights and implementing a binary or Fibonacci heap to sort through the edges of the graph. Using a binary heap with Prim's algorithm would give us O(|E|log(|V|) and a Fibonacci heap would give us O(|E|+|V|log|V|). Either of these implementations would use more memory than our current implementation because they require storing all the edges of the complete graphs, so depending on which we were interested in optimizing (run-time or memory) our algorithm would change accordingly.

Though our implementation in C++ was not terribly slow (24 minutes on average per trial for 131072 vertices and 4 dimensions), we also could convert the code to C. We chose C++ because representing coordinates with the tuple data structure was intuitive, but coordinates could also be represented with n-dimensional arrays with one coordinate per element.