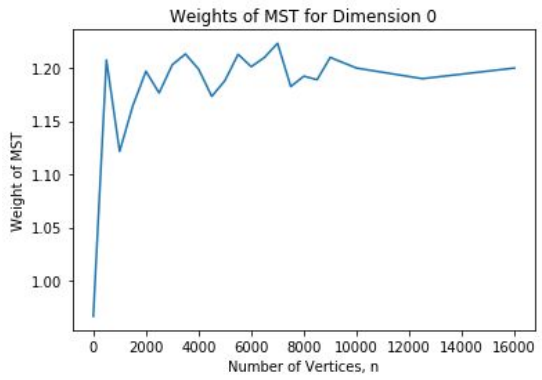
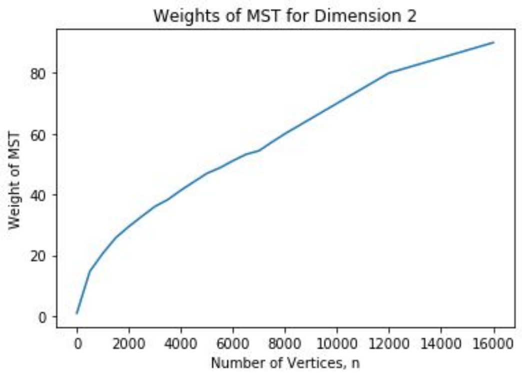
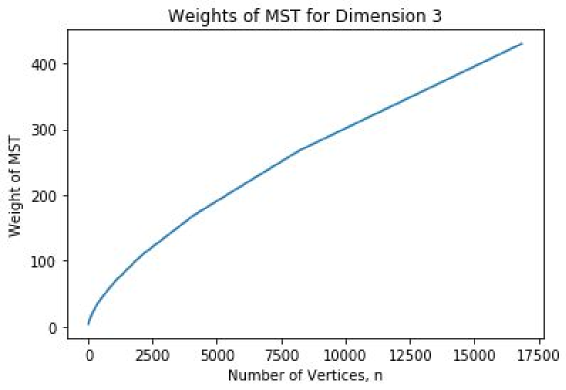
ANLY550
Programming Assignment 1
Brody Vogel, Jinghao Yan

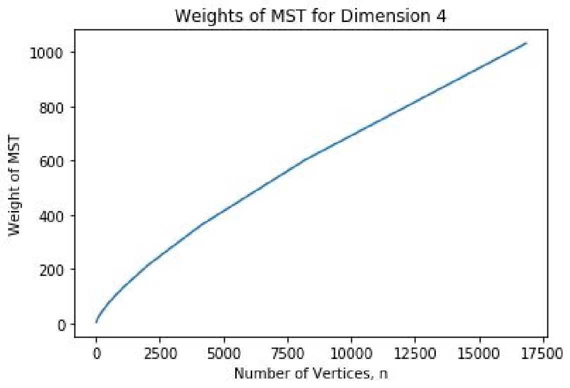
Part I - Results

First, here are plots of the average weight of Minimum Spanning Trees randomly generated according to the assignment’s specifications (we averaged the weights of five MSTs at each level):









And here is the corresponding table, with the averaged weight in the cells:

	16	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**	**16384**
0	1.11	1.14	1.30	1.22	1.18	1.16	1.21	1.20	1.21	1.19	1.20
2	2.87	4.05	5.55	7.64	10.76	15.07	21.05	29.54	41.92	60.17	90.27
3	4.33	7.31	11.38	17.51	27.99	43.43	68.08	107.65	169.16	267.29	429.70
4	5.84	10.13	17.34	28.14	46.90	78.16	129.23	216.11	360.97	602.11	1030.20

And finally, here is a table with the average runtimes (in seconds) for each dimension and graph size *n*[1]:

	16	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**	**4096**	**8192**	**16384**
0	.0008	.003	.013	.026	.144	.607	3.00	18.36	123.65	1234.43	5118.33
2	.004	.004	.013	.021	1.30	.774	5.08	35.81	302.12	1842.42	7241.84
3	.012	.031	.078	.219	.859	3.41	13.79	38.28	452.08	529.27	3606.4
4	.012	.016	.109	.297	1.10	3.87	18.06	73.31	321.20	569.78	4074.15

Part II - Discussion

Algorithm Choice

We decided to use Prim's Algorithm instead of Kruskal's to compute Minimum Spanning Trees (MSTs) because complete, undirected graphs are very dense, which means they have a lot of edges relative to their number of nodes. Prim's runtime is *O(m + nlogn)*, while Kruskal's is close to *O(mlogn + mlog*m)*, where *m* = |*E*| and *n* = |*V*|. Thus, when graphs are dense, the *m* terms dominate the runtime of the algorithms, and Prim's is more efficient with respect to *m* than Kruskal's. Thus, Prim's was a seemingly easy choice.

Implementation

There were a few interesting things we learned while building our code. In particular, storing our edges in an adjacency list and using probabilities associated with the Uniform(0,1) distribution (*U(0,1)* from here on) to throw out edges over a certain weight threshold vastly improved our runtime, and we had some strange issues with Python's *random* module (which we used as our random number generator). We'll briefly discuss both of these observations, in turn.

In our early attempts at implementing Prim's, we looked through every edge in the graph when we needed neighbors of a node. That meant we were considering all *m* edges every time we ran through Prim's *while()* loop, which made our code very slow. Thankfully, we remembered the adjacency list data structure and built a pseudo version of one into our code using Python's dictionary class. After that, each iteration of the *while()* loop found the relevant neighbor nodes in an *O(1)* call to an adjacency list look-up (dictionary index), as opposed to an *O(m)* look through all the graph's edges. This obviously sped our code up quite a lot.

[1] These were run on different computers with different amounts of RAM. Graphs with dimensions 0 and 2 were run on a computer with 8 GB of RAM, while graphs with dimensions 3 and 4 were run on a computer with 16 GB; so the runtimes listed here aren't necessarily illuminating. This is discussed in Part II.

As did the other choice we made for efficiency: throwing out edges above a certain weight. In a MST, there will be exactly *n-1* edges. Furthermore, in a complete, undirected graph like those in this assignment, each node will be an endpoint of exactly *n-1* edges (1 connecting it to each other node). Because Prim's will always try to choose one of the smallest (if it can't choose *the* smallest) edges that will allow it to include every node in the MST, and remembering that the weights of the edges are based on *U(0,1)*, we know it's incredibly unlikely that edges with weights towards the right end of the distribution will end up in the MST. Take, for example, a graph with *n* = 1000 nodes, with edges weighted as a random draw from the *U(0,1)* rounded to 4 decimal places. Based on probability theory, we'd expect, for each node, there to be around 100 edges with weights less than the 10th percentile of *U(0,1)*, .1. It seems reasonable, then, to disregard edges larger than .1 in this example. We used similar considerations to build cutoffs for various numbers of *n*, for each type of graph. We settled on the cutoffs in the code based on trial-and-error (which is the reason they may seem arbitrary). This, too, sped our code up considerably.[2]

Talking now about the problems we had with Python's *random* module, we found that, when multiplying two floating-point numbers very, very close to 0, Python will occasionally output negative or complex numbers, which is nonsensical. Scanning the available literature[3], this appears to be caused by the way Python stores long numbers. At a certain point, the multiplication estimates become unstable, and impossible outputs - like those we got - turn up. To remedy this, we rounded our decimals to 4 places whenever possible. We feared this would produce a lot of edges with weights of exactly 0, but this wasn't the case in the tests we ran. Thus, this rounding seems to solve the problem of nonsensical numbers. We ran into another issue, though, inside Prim's *while()* loop. Sometimes, our *Change()* function wouldn't correctly update the distance to a node, and so we'd get a weird error. We fixed this with a *try/except* call (which we tested by hand on small graphs to ensure accuracy), but we're still not sure exactly *why* our *Change()* function was sporadically malfunctioning. Our guess is that it has something to do with comparing decimals very close to 0 to infinity from the *math* module, but we aren't sure. Nevertheless, we think we managed to fix it.

Growth Rates

For *dim = 0*, it looks like the average weight of a MST for any moderately-sized graph is around 1.2, so our estimate is *f(n) = 1.2, n > 16*. If we assume 1.2 is close to a minimum for the weight of a MST for a moderately-sized, *dim = 0* graph, this makes some sense. As the graph grows larger, there will be more nodes to connect, but also more edges to choose from, and so the weights of edges added to the MST will get very close to 0 as the number of nodes grows. Thus, it would apparently stabilize around 1.2.

[2] Because it's just *very likely* that edges above the various cutoff weights will not end up in a corresponding MST, we built the code in such a way that Prim's will throw an error in the case that a MST has not actually been built because too many edges were thrown out.
[3] See https://docs.python.org/3/tutorial/floatingpoint.html.

For *dim* = 2, *dim* = 3, and *dim* = 4, our guess is that *f(n)* will be related to the percentiles of *U(0,1).* Remembering the way we found cutoffs by “safely” removing edges from our graphs, we can think of these as upper bounds for the largest edge weights that will be in the MST. So, for example, for a *dim* = 2 graph with 100 nodes, we can assume every edge weight in the corresponding MST will be in the interval [0, .25], where .25 is the best-guess result of our trial-and-error tests. Now, it’s unlikely that most of the edge weights in the graph will be close to that upper bound of .25. If we assume instead that the majority are in the first quartile of the just-mentioned interval (since Prim’s always tries to add the smallest edges it can), we can guess that the average edge weight in the MST is .25/4 = .0625. Thus, for higher dimensions, our guess at the function to estimate the weight of a MST, *f(n)*, is *f(n) = (c/4)n,* where *c* is the cutoff we found through trial-and-error.[4] Comparing that to our actual results:

dim = 2

	16	32	64	128	256	512	1024	2048	4096	8192	16384
Tested	2.87	4.05	5.55	7.64	10.76	15.07	21.05	29.54	41.92	60.17	90.27
f(n) = (c/4)n	N/A[5]	N/A	N/A	8	16	19.2	25.6	35.84	71.68	122.8	210.4

dim = 3

	16	32	64	128	256	512	1024	2048	4096	8192	16384
Tested	4.3	7.3	11.3	17.5	27.9	43.4	68.0	107.6	169.1	267.2	429.7
f(n) = (c/4)n	N/A	N/A	N/A	10.2	20.5	28.2	48.7	81.92	122.8	225.3	462.9

dim = 4

	16	32	64	128	256	512	1024	2048	4096	8192	16384
Tested	5.8	10.1	17.3	28.1	46.9	78.16	129.2	216.11	360.97	602.11	1030.2
f(n) = (c/4)n	N/A	N/A	N/A	16.6	33.3	44.8	81.9	133.1	225.3	409.6	819.2

It looks like our guess, on the whole, performs fairly well, although it falls apart a bit for *dim = 4* graphs. Where there are large discrepancies between our test results and our guess, we think

[4] The specific cutoffs we used for a 2-dimensional graph, for example, are: {100:.25, 500:.15, 1000:.1, 1500:.07, 5000: .06, 10000:.05}.
[5] The code ran fast enough that we didn’t find cutoffs for graphs smaller than 100 nodes.

we were too lenient or too strict with our cutoffs, or that the assumption that most of the edge weights were in the first quartile didn't hold up as well in the five test trials. With some tuning, we think we could use a close variation of *f(n) = (c/4)n* to very closely approximate our actual results.

Runtimes

Intuitively, the larger the graph, the longer it will take to find a MST. It also takes longer, it appears, to find a MST of graphs in higher dimensions. This makes sense, because it takes our code much longer to *initialize* graphs in higher dimensions. Graphs of *dim = 0* are initialized pretty quickly because all their initialization entails is building *m* combinations of the nodes and assigning them random weights. Graphs in *dim = 2, 3, 4,* though, take progressively longer to initialize because they're finding *2n, 3n,* and *4n* random numbers, building *m* combinations of those nodes, *and* computing *m* Euclidean distances. This would explain why computing MSTs in higher-dimensional graphs always took longer.

Actually, they *almost always* took longer, because we used separate computers to find the average MST weights for graphs of the 4 dimensions. For *dim = 0* and *dim = 2*, the program was run on a computer with 8 GB of RAM, while *dim = 3* and *dim = 4* were run on a computer with 16 GB of RAM. This had a huge impact on the runtimes for larger graphs. As listed in the table in Part I, it took well over an hour to compute a MST of a *dim = 0* graph with 16384 nodes on the computer with 8 GB, and well over two hours for an identical procedure on a *dim = 2* graph. With the 16 GB of RAM, though, we could compute a MST for a *dim = 4* graph with 8192 nodes in less than ten minutes (it took over a half hour to compute a similar MST of a *dim = 2* graph on the computer with 8 GB), and one for a 16384-node graph in a little over an hour (this crashed the computer with 8 GB). It therefore seems that cache size had a huge impact on the runtime of our code.

As a final note, it makes sense that calling our program on large graphs takes something of a long time to complete. The program - even in the best cases - goes through *a lot* of operations both during the initialization of a large graph and inside Prim's *while()* loop. So we don't think, for instance, an hour is a long time to find a MST for a 16384-node graph in 3 dimensions.