Matt Leifer 2/26/17

Average Weight	n	Trials	Dimension
1.303038	16	10	0
1.189757	32	10	0
1.217981	64	10	0
1.149354	128	10	0
1.182155	256	10	0
1.219053	512	10	0
1.211459	1024	10	0
1.198079	2048	10	0
1.202405	4096	10	0
1.199453	8192	5	0
1.200891	16384	5	0
1.201496	32768	5	0
1.203183	65536	5	0
1.201245	131072	5	0

Average Weight	n	Trials	Dimension
2.496839	16	10	2
3.773954	32	10	2
5.403075	64	10	2
7.611237	128	10	2
10.665314	256	10	2
14.942686	512	10	2
20.976625	1024	10	2
29.695654	2048	10	2
41.809853	4096	10	2
59.031574	8192	5	2
83.244881	16384	5	2
117.509232	32768	5	2
166.107647	65536	5	2
234.668610	131072	5	2

Average Weight	n	Trials	Dimension
4.455906	16	10	3
7.014079	32	10	3
11.012176	64	10	3
17.397980	128	10	3
27.677490	256	10	3
43.331860	512	10	3
67.915863	1024	10	3
107.336815	2048	10	3
168.968781	4096	10	3
267.144745	8192	5	3
423.622162	16384	5	3
668.000000	32768	5	3
1058.182739	65536	5	3
1677.734741	131072	5	3

Average Weight	n	Trials	Dimension
6.203696	16	10	4
10.267516	32	10	4
17.053518	64	10	4
28.464367	128	10	4
47.145477	256	10	4
78.498344	512	10	4
129.121155	1024	10	4
215.954666	2048	10	4
361.273834	4096	10	4
604.566101	8192	5	4
1008.472534	16384	5	4
1689.557373	32768	5	4
2828.068359	65536	5	4
4742.111816	131072	5	4

The function that seemed to me that approximates the weight of a minimum spanning tree with n vertices in d dimensions with is $f(n,d) = c \cdot n^{1-\frac{1}{d}}$ for some constant c. I graphed the points in a log-log graph because the growth of the functions looked polynomial and then upon playing around with what I thought was in the exponent I eventually realized that it was proportional to $n^{1-\frac{1}{d}}$ and then I did another regression to calculate c for each of the dimensions $d = \{2, 3, 4\}$. However, the minimum spanning trees of graphs with random edges (that's not in a euclidean space) seems to oscillate or approach a value around 1.202 that is largely independent of n - for smaller values the deviation from 1.202 is greater which suggests that this is some value that the weight of the minimum spanning tree will converge to as n tends to infinity. A possible reason for why this sort of minimum spanning tree approaches a constant as opposed to growing with the number of vertices is that as n increased, though there are more edges, the chances that there is an edge which is very close to a vertex becomes increasingly higher. So, in other words, the average edge length decreases even though the the total number of edges increases and for some reason these to forces 'balance' out at 1.202. The general form of $f(n)$ makes sense because as the dimension grows the maximum possible edge (and thus the length of all the edges grows) and so it makes sense that as d gets higher the threshold for size of the the maximum edge that will occur with in a minimum spanning tree should increase.

Now I'll discuss the algorithms and data structures and general procedures I used and then I'll discuss my actual results. Broadly speaking I used Prim's algorithm with a priority queue to hold the vertices to find the minimum spanning tree. At first, I stored every edge in the graph, however this became a problem for graphs which were larger than 16000 vertices because my computer ran out of memory - every time I doubled the number of vertices the number of edges, and thus the amount of space required to stroe them, quadrupled.

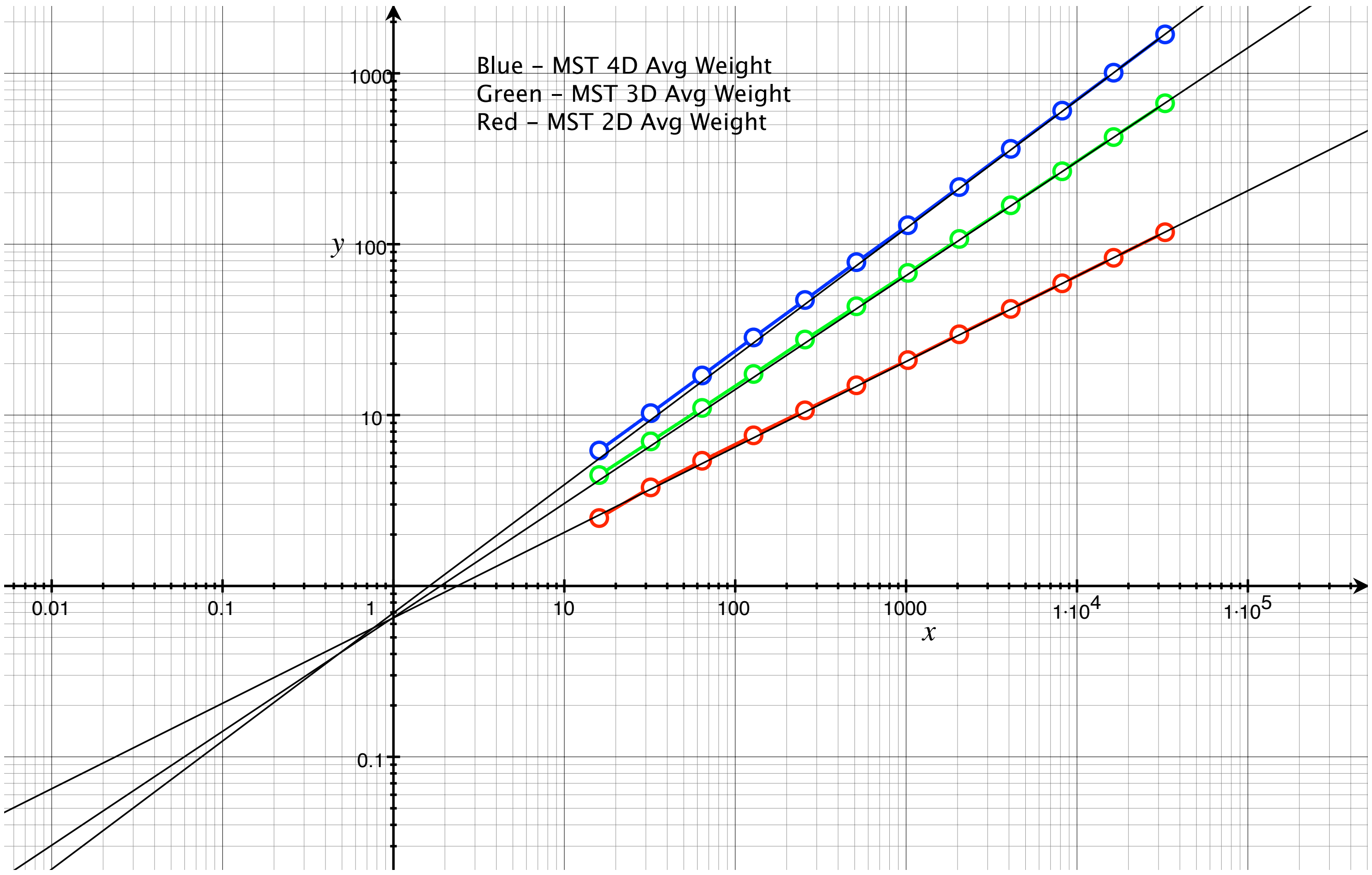
So, I looked at the hint given at the end of the problem set assignment, and derived a function that would provide an upper bound on the length of the edge I would need to store to run the program. The reason that we can throw away edges larger than a certain length is due to the greedy nature of Prim's algorithm. Since we always select the smallest edge that won't form a cycle and add that to the graph there will very likely be some edges which are so large that they will never be selected by Prim's algorithm because a shorter one can be found. Assuming we're dealing with a 'typical' graph, meaning that the points are more or less evenly distributed (which they should be because they're randomly generated), then there will be a high probability a randomly generated graph will not violate this upper bound. Intuitively, if we had 100000 vertices generated in the unit square randomly, it would be very unlikely that we would have to include use an edge that was 0.75 in length and so there's no need to store it later since it'll (likely) never be used. In practice, the bound is significantly lower than 0.75. To calculate this upper bound, I calculated the average largest edge in graphs that had between 4 and 100 vertices in dimension 2,3, and 4 and as well as the graphs with edges generated uniform at random. I guessed that the edges were a function of the form $n^{-1/b} = 1/\sqrt[b]{n}$ and then ran a regression to find out what b should be. As indicated by the graph at the end of the writeup, this threshold seems to overestimate the average largest edge seen in graphs with many vertices, but that doesn't really seem to hinder performance and if anything it may be better to have a slightly too high upper bound and thus very high success rate than tight upper bound that fails frequently.

I then coded these values into the get_threshold function in my code which calculates the maximum edge that I will store in a graph of size n. For graphs smaller than 2048 vertices, I don't calculating a threshold because there's too much variablility in the smaller graphs' edges and distribution of points. The threshold calculated for 1024 vertices fails approximately 2.5% of the time. During my testing, the threshold never caused a failure for 2048 vertices or more. When employed, the threshold excludes more then 98% of edges in my testing and even more for larger graphs. To give an idea of how much space this saves, when I didn't use the threshold my computer required about 2 gigabytes to run this program on a graph of 16384 vertices but when I use the threshold that number did not exceed 12 megabytes. Unfortunately, because edges are thrown away if they're too big according to the threshold (and are generated at runtime), if the threshold proves to be incorrect, there's nothing that can be done for this graph and the threshold will just have to be updated accordingly afterwards. Before I started using the threshold to throw away edges, the memory on my computer was used up so quickly that it became virtually unusable - 8 gigabytes of RAM would be gone within seconds.

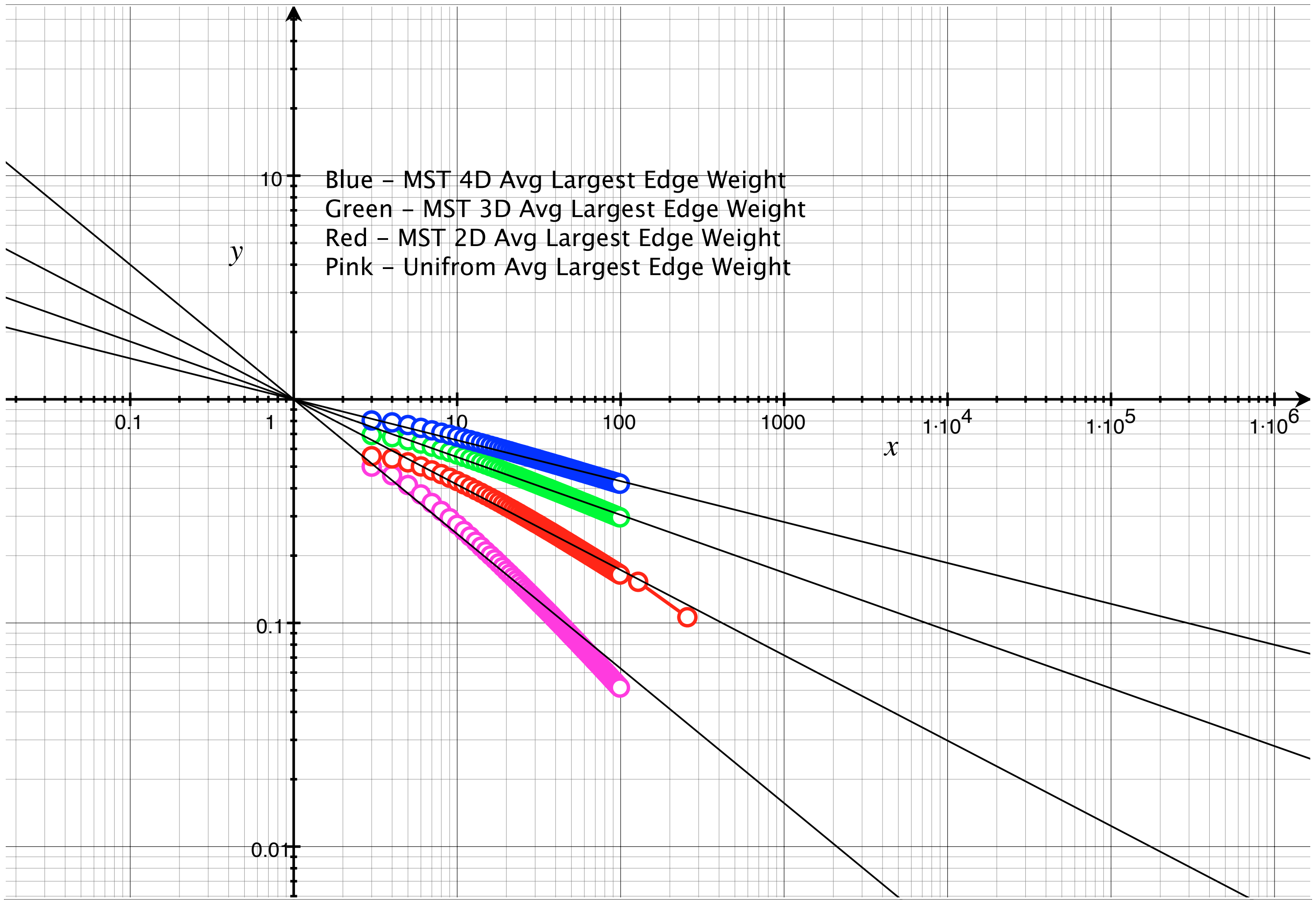
Because I knew I would be dealing with a complete graph and before I started excluding edges, I used Prim's algorithm since using a linked list to store all the potential edges will cause Prim's algorithm to run in $O(|V|^2)$ time which is better than using a heap which will run in $O(|E|log|V|) = O(|V|^2 log|V|)$ time since the graph is complete. Kruskal's algorithm would also run into similar problems because we'd have to sort the edges as they (or after they were generated) which would lead to the algorithm running in $O(|E|log|E|) = O(|V|^2 log|V|)$. Prim's algorithm therefore is a good choice when the graph is dense. However, once I figured out a function that would allow me to discard most of the edges, I changed the linked list that controlled all the edges we needed to check to a binary heap, because the graph was now very sparse and this would allow me to achieve much faster run times on the order of $|E|log|V|$. I stored the edges using adjacency lists because that would allow me to add only as many edges as I needed to added without allocating extra memory that using a static array would require. Moreover, there's no need for constant time access to all the elements in an adjaceny list because we just have to traverse the adjacency lists in Prim's algorithm. When I used the linked list, the time it took for the program to run appeared to quadratically in the number vertices, which makes sense given that this is an $O(V^2)$ algorithm as I implemented it. For example, when run on 4096, 8192, 16384, and 32768 vertices for the uniform edge graph (dimension = 0), the times it took for the algorithm to complete were 0.353175, 1.413877, 7.053407, 34.673977. Each time as the number of vertices doubled, the total time it took to run increased by a factor somewhere between 4 and 5. The reason it does not exactly quadruple is because we have to account for the time it takes for the edges to be generated and, for the examples in euclidean space, the time it takes to generate all the vertices as well. The use of a binary heap made the process so much faster. To calculate the minimum spanning tree on 65536 vertices in two dimensions with a linked list took approximately an hour, but with the binary heap, it took a little over two minutes. The most time consuming process (once I have a binary heap) is the time it takes to generate the edges which is still $O(V^2)$ since it's a complete graph - of those two minutes with the 65536 vertices MST, only about 3 seconds were spent using Prim's algorithm. When $n = 131072$, it takes between 9 and 10 minutes to generate all the edges in the graph and calculate the minimum spanning tree for that graph.

The random number generator originally gave me problems because I didn't realize that if I initialized it more than once within the same second it would generate the exact same set of numbers, because it's a pseudorandom number generator. I ran into this problem because at first I initialized the PNRG at the start of every trial and for the graphs with a very small number of vertices all these trials can take less than a second to run and so I actually generated the same graph each time. The solution was to instantiate it once before any trials had been run and not update until after a second had passed.

Lastly, here's a quick overview of the code in randmst.c itself. The flag (argv[0]) controls how much is printed by the program as it runs, the higher the number, the more gets printed for debugging and other information. I start off by initializing an array, edges[], of adjacency lists, and if the dimension is 2,3, or 4, an array points[] to store all the coordinates of the points in the graph. Coordidinates and edges are then generated using the rand() function from <random.h> and any edge greater than the predetermined threshold is not stored. Once the edges have been calculated, they're passed to MSTFind which runs Prim's algorithm to calculate the Minimum Spanning Tree for the graph and returns the weight of that MST. There's also code below that which handles inserting, creating, and deleting elements from various different data structures. The adjacency lists are unordered linked lists, the queue is a priority queue that is a sorted linked list - I didn't end up using any of this code, but I wrote it before I figured out how to exclude edges appropriately and a binary heap - and lastly, there's some code which manages the construction and insertion and deletion in a binary heap. The very last section is a test graph that we used Professor Mitzenmacher used in class - it's MST has weight 22 - and I used that to test my MST function. I have also attached two files that can be opened in the Mac application Grapher which I used to calculate the Regressions.



Blue – MST 4D Avg Weight
Green – MST 3D Avg Weight
Red – MST 2D Avg Weight
y
1000
100
10
0.1
x
0.01
0.1
1
10
100
1000
1·10^4
1·10^5


Blue – MST 4D Avg Largest Edge Weight
Green – MST 3D Avg Largest Edge Weight
Red – MST 2D Avg Largest Edge Weight
Pink – Unifrom Avg Largest Edge Weight
y
x
10
0.1
0.01
0.1
1
10
100
1000
1·10^4
1·10^5
1·10^6