Programming Assignment 1

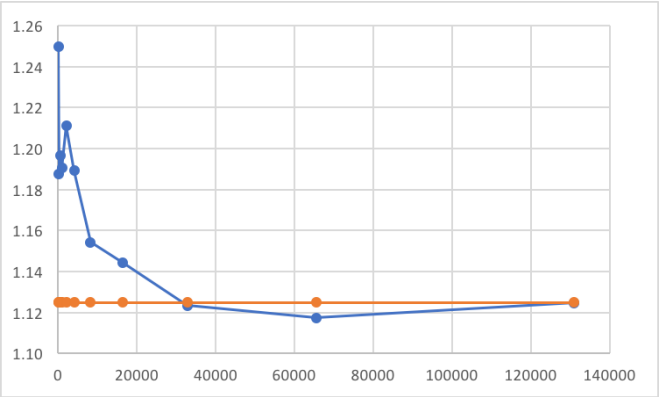
February 2017

60951527

Basic Graph

n	MST	f(n)	% Error
128	1.25	1.125	9.97%
256	1.19	1.125	5.25%
512	1.20	1.125	5.99%
1024	1.19	1.125	5.50%
2048	1.21	1.125	7.10%
4096	1.19	1.125	5.40%
8192	1.15	1.125	2.54%
16384	1.14	1.125	1.69%
32768	1.12	1.125	0.14%
65536	1.12	1.125	0.68%
131072	1.12	1.125	0.03%

$f(n) = 1.125$

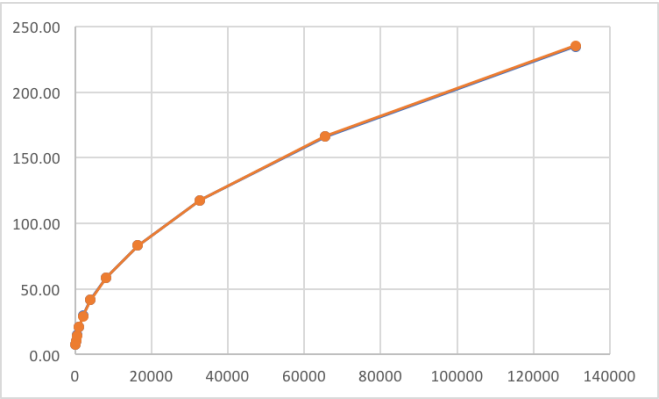


My estimate for the average weight of an MST for the "Basic" graph structure (uniform random weights on edges) is $\mathbf{O(1)}$. As you can see from the results, the number of vertices has little effect on the MST weight. This is a curious phenomenon, but I have some intuition as to why it happens. In the Euclidean graphs, if two vertices are close by then they are likely far away from some third vertex. However, vertices in the '0-Dimension' graph are not restricted by spatial properties - so they can be arbitrarily close or far from any other vertex, regardless of their existing relationships. Therefore, adding vertices in a complete graph of this form may increase the number of edges in the MST, but it also increases the probability of there being a very small-weighted edge connecting the vertex to the MST. In that way, each added vertex has a smaller and smaller impact on the weight of the tree, because it has thousands of chances to get a near-zero weight connecting edge.

Square Graph

n	MST	f(n)	% Error
128	7.65	7.35	3.85%
256	10.58	10.40	1.74%
512	14.90	14.71	1.28%
1024	21.16	20.80	1.72%
2048	29.60	29.42	0.62%
4096	41.84	41.60	0.58%
8192	58.84	58.83	0.02%
16384	83.29	83.20	0.11%
32768	117.48	117.66	0.16%
65536	166.03	166.40	0.23%
131072	234.71	235.33	0.26%

$f(n) = (0.65)n^{1/2}$

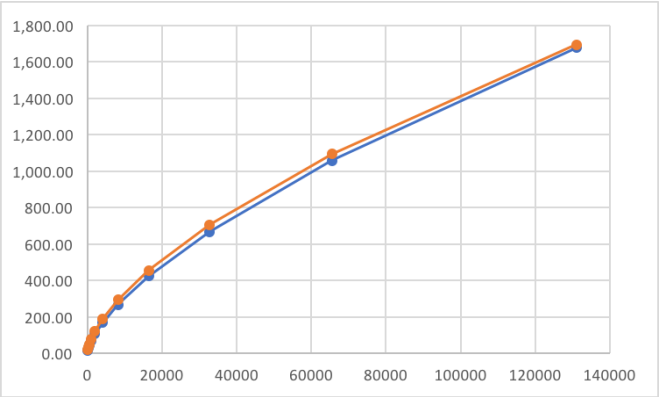


My estimate for the average weight of an MST of a "Square" graph structure (uniform random locations in 2D space) is $\mathbf{O(\sqrt{n})}$. This makes sense in that we're going to be bounded by the average distance - as I mentioned above, because the points are evenly distributed around the unit square, when two vertices are close to one another it means that they are likely far from some third vertex on the other side of the map. This means that you won't have the trend towards zero incremental weight as seen in the Basic graphs, though saturation still means we are expanding at a reduced rate ($\sqrt{n}$).

Cube Graph

n	MST	f(n)	% Error
128	17.41	21.36	22.67%
256	27.61	33.07	19.77%
512	43.26	51.21	18.38%
1024	68.04	79.30	16.55%
2048	106.96	122.81	14.81%
4096	169.03	190.17	12.51%
8192	267.98	294.49	9.89%
16384	422.61	456.04	7.91%
32768	668.09	706.21	5.71%
65536	1058.25	1093.61	3.34%
131072	1677.53	1693.52	0.95%

$\mathbf{f(n) = n^{\log_2(3)} = n^{0.63}}$

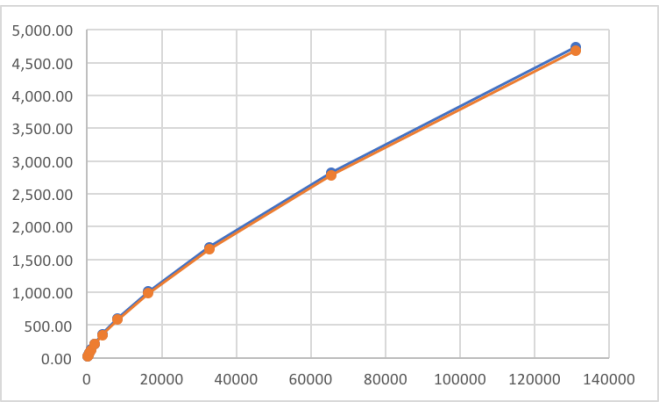


My estimate for the average weight of an MST of a "Cube" graph structure (uniform random locations in 2D space) is $\mathbf{O(n^{\log_2(3)})}$. This is basically $O(n^{2/3})$, which shows the pattern that's emerging here. As the dimensionality increases, the order of the weight of the average MST increases, with an upper bound.

HyperCube Graph

n	MST	f(n)	% Error
128.00	28.65	25.88	9.67%
256.00	47.33	43.52	8.04%
512.00	78.38	73.19	6.63%
1024.00	130.12	123.09	5.40%
2048.00	217.66	207.02	4.89%
4096.00	360.76	348.16	3.49%
8192.00	602.98	585.53	2.89%
16384.00	1009.33	984.75	2.44%
32768.00	1688.27	1656.14	1.90%
65536.00	2826.53	2785.28	1.46%
131072.00	4739.95	4684.26	1.17%

$\mathbf{f(n) = (0.68)n^{(3/4)}}$



My estimate for the average weight of an MST of a "Cube" graph structure (uniform random locations in 2D space) is $\mathbf{O(n^{3/4})}$. This shows the pattern: for an n-dimension average MST, the weight appears to be $O((n-1)/n)$ (with $n = 0$ being $O(1)$). As a sanity check, we can see that this will hold for $n = 1$: a number line. Basically, the MST will just be the length of the line - each vertex connects to its closest left and right partners until we reach the full width, giving us $O(0/1) = O(1)$.

Discussion:

Kruskal's: I decided to use Kruskal's algorithm for this problem for a number of reasons. First, I find it to be more interesting and perhaps a bit more challenging. While the algorithm itself isn't necessarily difficult, creating and testing the union find data structure took a bit of thought.

Growth Rates: At first I was a bit surprised by the growth rate of the Basic graph. However after thinking it through (as explained up above) it makes sense - this is a complete graph, so the more vertices you add, the more likely it becomes that you will get a connection weight close to 0 for each new vertex. Otherwise, I think the average weights weren't super surprising. It is also nice to see the pattern emerging, since it gives a more intuitive understanding of the relationship between dimensionality and average MST weight.

Runtime: There are a few factors that mess with the runtime of the algorithm. At best, I was able to calculate the whole data set (6 trials for every n for each graph type) in just over 19 minutes. I think that is quite good, given the size of the problem. I'm running on a Intel Core i7-4980HQ CPU @ 2.80GHz (Overclocks to 4.0GHz), which has 4 physical and 8 logical cores. By multithreading the trials, I was able to run about 6x faster (Though I'm sure I could've taken a look at pset3 q4 and parallelism vs. concurrency to better distribute the processes). Furthermore, by compiling with -Ofast, I was able to reduce the runtime by another 5x.

Pruning, Iterative Deepening: For the staff-ready operation (which deals with specific trials rather than full-blast calculation), the 131,072-vertex Hypercube graph calculation runs in about 38 seconds and eats approximately 118MB of RAM. Part of why it is able to operate so speedily is the aggression with which I am able to prune edges. I employed an iterative deepening strategy, which will re-run a trial of Kruskal's with a looser bound on the maximum considered edge weight in the case of failure. This means that even if my k(n) is a bit too aggressive, the algorithm is still correct. Because I double the bound in the case of a failed run, it doesn't change the order of the worst-case runtime.

k(n): Ultimately I doctored k(n) to make sure that the iterative deepening step happens very infrequently, because it does add a constant factor to the runtime. Also, because edges have to be pruned at creation (rather than just Union-Find insertion) for the Basic/0-Dimension graph, it does have a small chance of failure. Overall though, the effect is faster practical runtime without loss of correctness.

Random Numbers, Optimization: I found the random number generator to operate fine. Testing the average edge weight (not MST weight) on the standard graph showed that we were getting good random numbers. I would have liked to spend some time researching speed vs. accuracy tradeoffs for various functions, since the generator is called very frequently. However, when profiling the runtime of the program, it was the edge weight calculation of the Euclidean graphs that ended up taking up the bulk of the runtime (aside from the sort). I would definitely be interested in researching whether there is a faster way to calculate or estimate distance - perhaps in the same vein as the fast-inverse square root algorithm made famous by John Carmack. However in the interest of staying in line with course policy I haven't looked into it yet.

Closing Remarks: Overall, I found this to be a stimulating and enjoyable exercise. While the goal is correctness, I would definitely enjoy a "Big-Board" style algorithm competition to see how different implementations line up. Regardless, I'm glad to have the opportunity to implement some of the data structures and algorithms from class - though I'm enjoying the written problem sets I definitely prefer the more practical aspects of programming.

Sources:

cs.fsu.edu/~myers/cop3330/notes/dma.html

en.cppreference.com/w/cpp/container/vector

en.cppreference.com/w/cpp/memory/unique_ptr

cplusplus.com/forum/beginner/12409/

cplusplus.com/reference/vector/vector/

en.cppreference.com/w/cpp/numeric/random/uniform_real_distribution

en.cppreference.com/w/cpp/numeric/random

stackoverflow.com/questions/1074474/should-i-use-double-or-float

acodersjourney.com/2016/05/top-10-dumb-mistakes-avoid-c-11-smart-pointers/

stackoverflow.com/questions/333443/c-object-instantiation

quora.com/How-do-I-make-my-own-header-file-in-C-and-C++

cplusplus.com/forum/articles/10627/

cs.bu.edu/teaching/cpp/writing-makefiles/

stackoverflow.com/q/2408038

cs.umd.edu/class/fall2002/cmsc214/Tutorial/makefile.html

stackoverflow.com/q/9322253

stackoverflow.com/q/9322253

cplusplus.com/doc/tutorial/operators/

stackoverflow.com/q/2940367

cplusplus.com/doc/tutorial/arrays/