CS124 Programming Assignment: Ben Anandappa and Kyle Kwong

RESULTS

Dimension = 0

n	$f(n)$
16	1.287811
32	1.566054
64	1.497568
128	1.408702
256	1.51272
512	1.491951
1024	1.518965
2048	1.521151
4096	1.51782
8192	1.207898
16384	1.438853
32768	1.490328
65536	1.500143

Excluding the outliers f(16) and f(8192), the average of f(n) is 1.5003. The function g(n) = 1.5003 is a good approximation of f(n).

Dimension = 2

n	$f(n)$	$n^{1/2}$	$f(n)/n^{1/2}$	$0.7614 * n^{1/2}$
16	2.788859	4	0.69721475	3.0456
32	4.301632	5.656854249	0.760428289	4.307128826
64	6.168682	8	0.77108525	6.0912
128	8.884378	11.3137085	0.785275491	8.614257651
256	12.442919	16	0.777682438	12.1824
512	17.561265	22.627417	0.776105598	17.2285153
1024	24.772576	32	0.774143	24.3648
2048	34.409645	45.254834	0.760352916	34.4570306
4096	48.564183	64	0.758815359	48.7296
8192	68.625559	90.50966799	0.758212471	68.91406121
16384	96.942668	128	0.757364594	97.4592
32768	136.844034	181.019336	0.755963628	137.8281224
65536	196.010885	256	0.76566752	194.9184

The average of $f(n)/n^{1/2}$ is 0.7614, so as we can see from the table, we can approximate f(n) as g(n) = $0.7614 * n^{1/2}$. We figured this out by looking a graph of f(n) vs n and noticing that it looks very similar to a square root graph.

Dimension = 3

n	$f(n)$	$n^{2/3}$	$f(n)/n^{2/3}$	$0.766 * n^{2/3}$
16	4.743462	6.349604208	0.747048453	4.863796823
32	7.820685	10.0793684	0.775910225	7.720796194
64	12.842705	16	0.802669063	12.256
128	20.19819	25.39841683	0.795253898	19.45518729
256	31.599015	40.3174736	0.783754838	30.88318478
512	49.245631	64	0.769462984	49.024
1024	77.736074	101.5936673	0.765166531	77.82074917
2048	122.8152	161.2698944	0.761550694	123.5327391
4096	193.923037	256	0.757511863	196.096
8192	305.780897	406.3746693	0.752460525	311.2829967
16384	483.547002	645.0795775	0.749592793	494.1309564
32768	765.19601	1024	0.747261729	784.384
65536	1211.697603	1625.498677	0.745431307	1245.131987

By graphing it, we saw that f(n) grew slower than linear but faster than $n^{1/2}$ so we guessed $n^{2/3}$. The average of $f(n)/n^{2/3}$ is 0.766, and we can see that g(n) = 0.766 * $n^{2/3}$ is a good approximation of f(n)

Dimension = 4

n	$f(n)$	$n^{3/4}$	$f(n)/n^{3/4}$	$0.8123 * n^{3/4}$
16	5.980992	8	0.747624	6.4984
32	11.602174	13.45434264	0.862336742	10.92896253
64	19.671795	22.627417	0.869378728	18.38025083
128	31.514738	38.05462768	0.828144694	30.91177406
256	53.1891	64	0.831079688	51.9872
512	89.476336	107.6347412	0.831296058	87.43170024
1024	148.533202	181.019336	0.820537768	147.0420066
2048	246.27943	304.4370214	0.808966757	247.2941925
4096	410.523811	512	0.801804318	415.8976
8192	685.939253	861.0779292	0.796605313	699.4536019
16384	1145.321887	1448.154688	0.790883665	1176.336053
32768	1917.845048	2435.496172	0.787455579	1978.35354
65536	3210.352736	4096	0.783777523	3327.1808

By similar logic, we saw that f(n) is slower than linear but faster than $n^{2/3}$ so we guessed $n^{3/4}$. The average of $f(n)/n^{3/4}$ is 0.8123, and we can see that g(n) = 0.8123 * $n^{3/4}$ is a good approximation of f(n)

Discussion

We decided to use Prim's algorithm because in a complete graph of n vertices, there are significantly more edges than vertices (since each of the n vertices has $n-1$ edges). The amortized time of Prim's algorithm is faster than Kruskal's if implemented with a Fibonacci heap for graphs whose number of edges is significantly larger than the number of vertices, so we decided to use Prim's algorithm. We also found that Prim's algorithm was easier to implement, since it is very similar to Djikstra's algorithm, and we already learned how to implement binary heaps in class.

The growth rates of $f(n)$ are fairly surprising; for the 0-dimension case, we found that a constant function described the $f(n)$ fairly well, despite the fact that intuitively, the weight of the MST should be growing as the number of vertices grows. However, this may be explained by the fact that as the number of vertices grows, the number of edges grows exponentially, giving us a higher probability of finding extremely small edges with which to build our MST. For the higher dimension cases ($d \geq 2$), we found that although $f(n)$ grows more slowly than a linear function, $f(n)$ appears to follow a trend such that for dimension d, the function that models the average weight of an n-vertex graph is of the form $n^{(\frac{d-1}{d})}$. In other words, as the number of dimensions increases, it seems to approach a linear model.

The longest amount of time that it took for our program to run was 17 minutes for $numpoints = 65536$, $numtrials = 5$, and $dimension = 4$. We found that as we doubled the number of vertices, the amount of time it would take to run increased exponentially. For example, while running our program for 2048 points in two dimensions only took 1 second, running our program for 4096 points in two dimension took 3 seconds. This is most likely do the fact that as we double the number of vertices, the number of edges we have to check and filter increases exponentially. We also found that as we increased the dimension, the amount of time for our program also increased, which is due to the fact the higher the dimension, the more calculations required to find the weight of the edge between two vertices. We did notice that the cache size of our computer had an effect on how quickly our code ran; one of our computers have a cache that was twice the other's, and it ran the same command significantly faster (not twice as quickly, but still faster).

We simply used Java's built in Math.random() function generate random numbers between 0 and 1. We decided not to seed it, since by default, the random number generator will seed it with something like the system time anyway. If we seeded it with the same number every time, then we would generate the same "random" numbers every time and get the same MST weights each time, so we decided against seeding the random number generator ourselves. While there were probably ways we could have improved our random number generator to get enough more "random" numbers (using an established random number generator algorithm, such as Mersenne Twister), we believed that Math.random() would give us random enough numbers.

Because we began getting memory errors for large numbers of *numpoints* if we kept every edge, we began keeping track of the largest edge weight in our resulting MST for the runs that did not lead to memory errors. We noticed that as the number of vertices in the graph increased, the largest edge in the resulting MST strictly decreased. This meant that when adding edges to our adjacency list of a graph with n vertices, we could use the value of the largest edge in a MST of a graph with $\frac{n}{2}$ vertices as an upper bound for even including those edges in our adjacency list. So, to cut down on the number of edges we actually stored in our adjacency list and therefore in our computers' memory, we had a series of statements such that if a graph has more than n nodes, then we only add the edge if the edge weight is greater than the largest edge weight in an MST of $\frac{n}{2}$ (we hard-coded these edge lengths using our empirical data). Our technique should give the same results as a non-optimized program because as the number of vertices increases, there are many more edges whose weights are more or less random, which means that the more edges there are from a given vertex, the more likely that there will be an edge with an extremely small

weight that can be used in our MST, leading to a smaller maximum edge weight in the MST. Since we know edge lengths greater than the max edge weight in the MST will never be used in the MST, then we can filter out all of the edges from our adjacency list whose weight is greater than that max edge weight we found.