CS 124: Programming PSET 1

10904501 & 80944012

2/27/17

1 Algorithm Selection

We chose to use Prim's algorithm as Prim's algorithm performs better than Kruskal's algorithm when dealing with a dense graph, which is the case since we are dealing with a complete graph. We note that the running time of Prim's algorithm, if we use d-heaps, is $O(mlog_{m/n}n)$. Whereas Kruskal's algorithm runs in $O(mlog^*n + mlogm)$ assuming that the edges are not initially given in sorted order (which is the case here).

2 Graph Representation

2.1 Adjacency Matrix Implementation

We decided to use an adjacency matrix over an adjacency list as when dealing with a complete graph an adjacency list takes the same space as that of the matrix $O(n^2)$. Additionally, an adjacency matrix has the advantage of an $O(1)$ look-up for edge weights as versed to an $O(n)$ look-up time in an adjacency list. However, we discuss some advantages of the adjacency list when factoring in edge deletions by calculating $k(n)$.

We first note that the creation of an adjacency matrix takes $O(n^2)$ (using the same amount of space as well) since there are n^2 cells in the matrix. When implementing the matrix, we noted that the elements running down the diagonal (vertexes that match to themselves) could be left untouched. Additionally, we could further make use of symmetry in an undirected graph such that when we set the [i][j] position in the matrix (where i and j denote vertexes), we could set the [j][i] position in the matrix at the same time. We note that this helps to reduce the number of iterations we have to make through the matrix to $\frac{n^2-n}{2}$. While this is still $O(n^2)$, such constant time optimizations can certainly make a difference.

Note that we later found out that using an adjacency matrix, we quickly ran out of memory starting at an n value of 2048. To fix this, we took advantage of the fact that we could store each vertex's position in d, where d denotes the number of dimensions, n length arrays. In Prim's algorithm we could then compute the edge weight only when we needed it, thus we reduced our space costs to $O(n)$ as versed to $O(n^2)$ as we essentially got rid of storing the matrix at all. This works well as we took advantage of the fact that we would not have to re-compute the edges from or to a vertex we already visited; thus each edge was only computed once.

2.2 Adjacency List Implementation & k(n) Discussion

We implemented an adjacency list as a vector of lists. The lists were of type node, which is a struct that has a float weight and int index. For every node, the weight is the weight of the edge leading to the index node. Every line in the vector contains a list of some of these nodes. The index of the line is just the node that the list of adjacent nodes connects to.

Advantages of this method are that the size of the list is dynamic so that we can remove edges and save in size. An adjacency matrix took quadratic space, which was around 17 bil bytes of memory at high n's.

The disadvantage of this method is that we lose random access while looking over the list for each node.

We can find k(n), or the maximum weight that a MST for a tree with n vertices will have, by just finding the largest edge in some of the cases.

For d = 0:
k(128) = 0.05 k(256) = 0.03 k(512) = 0.02 k(1024) = 0.01 k(2048) = 0.005 k(4096) = 0.0025 k(8192) = 0.00125

For d = 2:
k(128) = 0.17 k(256) = 0.15 k(512) = 0.09 k(1024) = 0.077 k(2048) = 0.05 k(4096) = 0.04 k(8192) = 0.025

This just seems to be around 10 times more than k(n) for d = 0 for large enough values of k.

For d = 3:
k(128) = 0.35 k(256) = 0.26 k(512) = 0.19 k(1024) = 0.15 k(2048) = 0.125 k(4096) = 0.01 k(8192) = 0.085

For d = 4:
k(128) = 0.5 k(256) = 0.4 k(512) = 0.35 k(1024) = 0.25 k(2048) = 0.2 k(4096) = 0.2 k(8192) = 0.15 k(16384) = 0.14 k(32768) = 0.11

It seems like the k(n) function is largest for d = 4, so as a conservative overestimate, we can use that data to find an accurate function for what edges we can discard.

After plotting the values for k(n) for d = 4, we find a function that is conservative overestimate for all of the values. This means that we can be sure that none of the edges we delete could have been used in the MST. $k(n) = 0.5 * e^{(-1/70400)*n}$ This function assumes that n will be between 128 and 132000.

This assumes that the number of edges we delete will save us enough space such that we will not run out of memory. When we were storing an adjacency matrix we ran out of space when we had to store a matrix of size $n = 2048$.

While we save time our main focus was on correctness, adding this would not change the results of our MST values reported below.

FILIP: Also you should note that the times will vary on machine so we will need to generalize this. There could also be difference in results as well bc of different machines, etc.

3 Prim's Algorithm

3.1 Heap Implementation

We note that the number of edges in a complete graph can be found by applying the general formula of n^{n-2}, where n denotes the number of vertexes in the graph. From lecture, a natural choice for $d = \frac{|E|}{|V|}$; i.e. $d = \frac{n^{n-2}}{n}$. By using d-heaps, we will obtain a running time for Prim's algorithm of $O(mlog_{m/n}n)$ time. While the use of a Fibonnaci heap would offer improved time bounds, we chose to stick with d-heaps for simplicity.

We used a standard array to store the d-heap values, taking advantage of arithmetic to access an element's children or parents, for simplicity and efficiency purposes. Additionally, we implemented each method iteratively as versed to recursively to save on space costs. While inserting, we noted that we would need to handle a vertex change as versed to a fresh vertex insert differently. To deal with this, we used an array to track the position of that vertex in the heap (-1 if it was not). This allows for a change operation to be done in $O(1)$ time. While sacrificing space, we save considerably on time as this is indeed more efficient than having to traverse the entire heap each time to find what position the vertex is in.

Note: when running Prim's algorithm we noticed a bug in determining the optimal value of d. Namely, for large values of n (computing $\frac{n^{n-2}}{n}$ and also finding the Kth child in some cases), we would face integer overflow. To deal with this we created a d-heap of $n/16$ if $n = 131072$ and $n/8$ for values of n less than 131072, ensuring that we would not run into integer overflow. While we do not achieve our desired running time of $O(mlog_{m/n}n)$, we still achieve a running time of $O(mlogn)$.

3.2 Set Implementation

While Prim's algorithm relies on a set, we made use of a fixed size array that stores in position i the ith vertex, denoting whether that vertex had been included in our set S or not. The cost of using $O(n)$ space was well worth it since look-ups and set-unions can now be done in $O(1)$ time.

4 Results

4.1 Table of Average Weight of MST with n Vertices and Dimension d

n	*trials*	*d=0*	*sec-0*	*d=2*	*sec-2*	*d=3*	*sec-3*	*d=4*	*sec-4*
16	5000	1.1613	0.075886	2.70831	0.137965	4.51605	0.130334	6.14025	0.179705
32	5000	1.18445	0.177623	3.86289	0.402742	7.16151	0.417654	10.316	0.655128
64	5000	1.19356	0.528856	5.43166	1.41798	11.2455	1.50397	17.166	2.15686
128	1000	1.20222	0.340681	7.61926	0.964226	17.6058	1.04328	28.4835	1.44943
256	200	1.20306	0.228937	10.6576	0.770303	27.5798	0.824984	47.1438	1.20693
512	200	1.19535	0.875126	14.9728	3.13928	43.3816	3.3114	78.1309	4.16728
1024	100	1.1965	1.80612	21.0525	7.01304	68.1942	6.90976	130.043	8.69116
2048	25	1.19861	2.13062	29.6363	8.69082	107.309	7.21514	216.671	9.06341
4096	10	1.19471	3.41653	41.8752	16.7337	169.332	12.3352	361.383	14.8331
8192	5	1.2013	7.18896	58.9849	36.3993	267.715	26.4521	603.916	30.5629
16384	5	1.20119	28.5013	83.1703	186.647	422.792	116.658	1007.94	135.965
32768	5	1.20259	125.27	117.501	980.42	668.647	538.804	1690.28	580.045
65536	5	1.20236	508.181	165.893	5499.49	1058.54	2421.86	2827.15	2245.38
131072	5	1.20128	963.277	234.69	15317.8	1676.49	8830.29	4741.68	7814.21

For clarity, note that time is represented in seconds and each observation contains the total time taken to run m trials. To find the time per trial, divide the total time by m.

4.2 f(n) Estimation

For the following dimensions we can fit these values to a graph and estimate $f(n)$. We see what shape the graph has and do some binary searching for the functions and constants.

For Dimension 0, we see that the average weight of the MST quickly approaches 1.2. Thus, we can estimate $f(n) = 1.2$.

For Dimension 2, we can estimate $f(n) = 0.65 * n^{1/2}$.

For Dimension 3, we can estimate $f(n) = (0.65) * n^{2/3}$.

For Dimension 4, we can estimate $f(n) = 0.685 * n^{3/4}$.

The growth rates were initially surprising. However, as we progressed and plotted the the values with n on a graph, we saw that the dimension change impacts the growth rate in a pattern. As we can see, the 2 dimension f(n) is with a power of 1/2, the 3 dimension f(n) is with power 2/3 and 4 dimension f(n) is with power 3/4. This has to do with the increase in the possible length of any edge with the increase of d.

5 Discussion

5.1 Algorithm Run Time

We can very clearly see that the algorithm running time increases exponentially as n increases, which follows from the Order Notation we calculated earlier. Additionally, we see that higher dimensions take more time to finish for smaller values of n. One potential explanation is the number of computations necessary to compute the Euclidean distance, which increases with dimensions. However, as n increases it seems that lower dimensions take longer to compute (with the exception of $d = 0$, which took less time overall). Thus, there may be some more interesting things at work in relation to the dimension of the problem. Overall, however, we see that the run times grow exponentially, which makes sense given the Order Notation we had determined before.

5.2 Random Number Generator

When working with random numbers, it is important to keep in mind that a computer is deterministic and uses the same algorithm to generate a random number based on the seed (thus finding true randomness can be difficult). A random number generator produces the same random numbers within a small amount of time because it seeds from the clock-time on the machine that the program is being run on. On a large enough number of trials, the clock-time also changes so we do get actual random numbers - thus it is important to use a random seed in order to generate random numbers.

5.3 Cache Size

The size of the cache on our computers certainly played an important role. We found that we had difficulty initially handling large values of n as we would quickly run out of memory. Thus, we had to make adjustments as mentioned above to ensure we were using less memory and the program would finish. It was also interesting to note that the run-time differed per computer - computers with more memory than the other would run faster.