CS124 Coding Project 1 Write-up

Luke Bailey, Matthew Karle

February 2020

1 Introduction

The problem of determining minimum spanning trees within a graph is a long studied issue of tremendous pertinence to computer scientists, mathematicians, and any others who would benefit from the analysis of complicated network systems. Many algorithms have arisen to address the minimum spanning tree problem in a more optimal fashion, all with respective benefits and trade-offs. In this coding assignment we sought not to generate another such algorithm, but to examine one's tendencies on randomized graph inputs of increasing scale. To do so, we implemented Prim's Algorithm and randomly generated graphs in various dimensions as per the coding project specification. We conducted the coding project entirely in C and eschewed established high-level data structures in favor of recreating our own. For each dimension, we ran a number of trials of the program for graphs with 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, and 262144 vertices. By running multiple trials, we found the average total weight of the MST for the above graph sizes and dimensions. The number of trials we ran depended on the running time of the program for the specific input size and the dimension it was running in (because as the input size and dimension increased, running time increased); however, for every graph size we ran 5 or more trials.

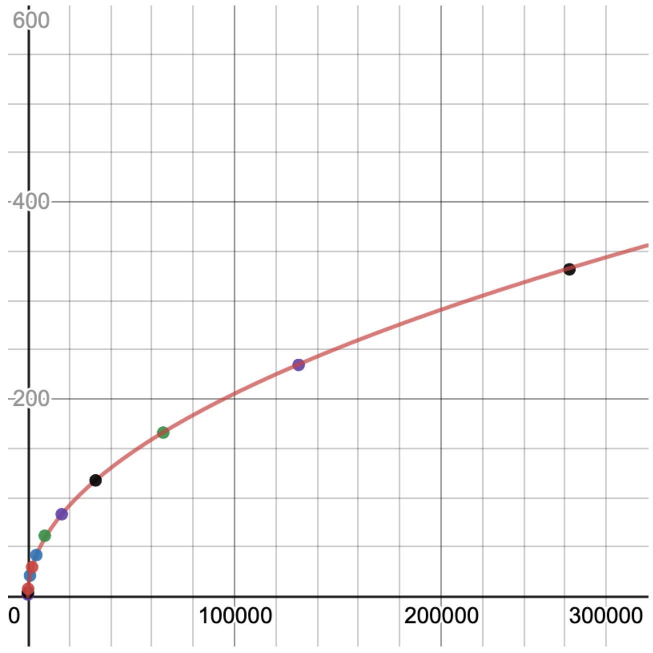
2 Results

The following table shows the average total MST weight for graphs of various sizes and in various dimensions.

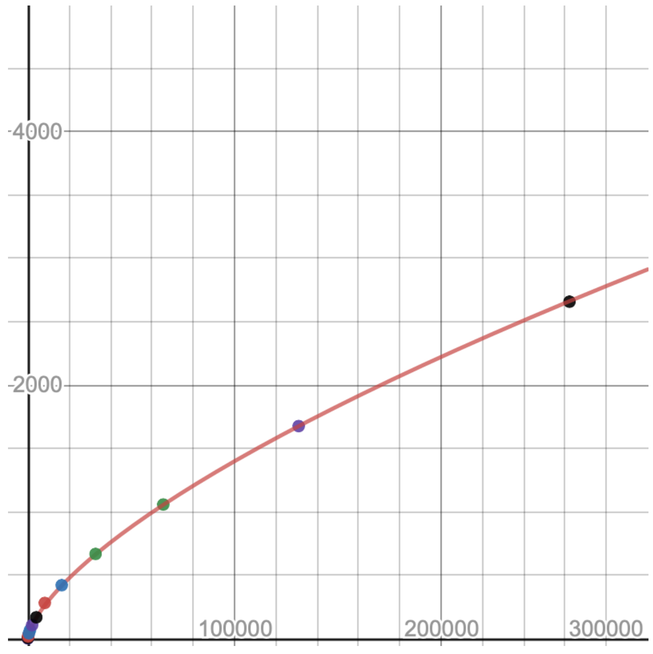
	Dimension			
Vertices	0	2	3	4
128	1.180803	7.614967	17.612462	28.43935
256	1.201191	10.672225	27.60131	47.126434
512	1.201794	14.977388	43.311386	78.224012
1024	1.201486	21.047236	68.099553	130.044431
2048	1.201426	29.637109	107.257958	216.576966
4096	1.201644	41.776056	169.221486	361.241733
8192	1.201765	61.434428	282.498118	642.18612
16384	1.201866	83.212015	422.506417	1008.836426
32768	1.197493	117.502275	668.873177	1689.61262
65536	1.197922	166.009873	1058.653831	2828.074297
131072	1.213603	234.675024	1676.958158	4740.921672
262144	1.17194	331.569952	2657.767641	7950.26359

From these results, we graphed the data to try and form an equation $f(n)$, where $f(n)$ is the average weight of the MST in a given dimension, and n is the number of vertices in the graph. For the case when the weights of edges were all assigned randomly (dimension 0) we did not graph the values because it is fairly clear from the data that $f(n) = 1.2$. For the other dimensions we found the following approximations that fit our data very well:

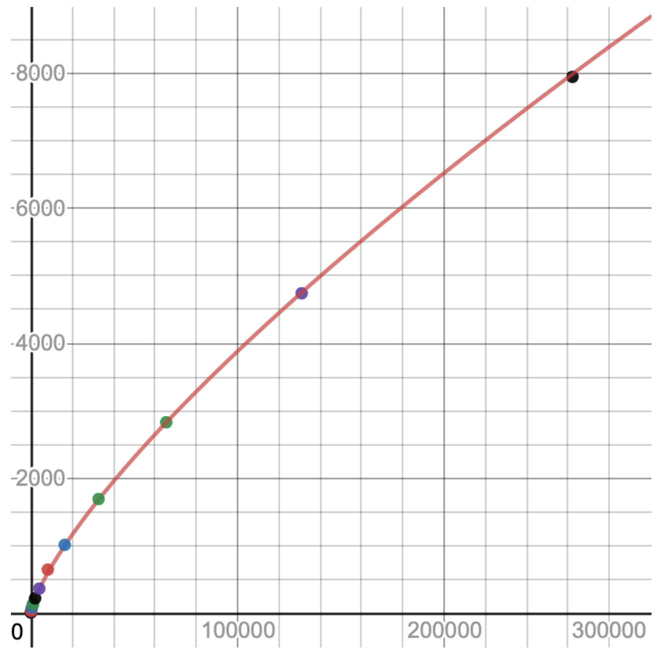
$$2\ dimensions:\ \ f(n) = 0.65n^{\frac{1}{2}}$$



$$3\ dimensions:\ \ f(n) = 0.65n^{\frac{2}{3}}$$



$$4\ dimensions:\ \ f(n) = 0.69n^{\frac{3}{4}}$$



We can see that from the above, if the dimension being considered is d (excluding the dimension 0 case), $f(n) = c \cdot n^{\frac{d-1}{d}}$ where c is some value. Our data suggests c could be some constant value roughly equal to 0.663 (the average of $6.5, 6.5$ and 6.9) but we cannot claim it is always constant or that $f(n) = c \cdot n^{\frac{d-1}{d}}$ holds for higher dimensions.

3 Project Discussion

3.1 Choice of algorithm

We decided to use Prim's algorithm over Kruskal's for a number of reasons. Firstly, after some research online, there was consensus that Prim's performs faster than Kruskal's for denser graphs. Seeing as our algorithm would be running on a completely dense graph, Prim's seemed like a better choice. With later consideration, we recognized the graph might be made more sparse by the strategic discarding of edges. However, we were not sure of to what extent this would be the case, and felt that Prim's remained a valid approach. Moreover, as we were interested in implementing a Heap in C in particular because of its frequent appearance in many algorithms of interest, we decided to continue to pursue Prim's Algorithm.

3.2 Implementation

In an effort to best understand the inner workings of our algorithms, we were determined to implement Prim's in relatively low-level C with minimal assistance from existing structures or libraries. From a programming perspective, this began with the implementation of the priority queue data structure integral to Prim's very functionality - the min-heap. In the name of an uncluttered and readable Main file, we moved the large number of heap helper functions and definitions into another file, Heap.c. I first designed the heap from my understanding of it in section and lecture using an array of integers. Though this choice would later result in a lengthy refactor to accommodate more complicated "nodes" as heap elements, it was a useful translation of learned theory into practice. These nodes were the next step in producing our program. It became important for us to decide exactly what needed to be tracked and what could be omitted to conserve space. A "known distance" value was quickly recognized as essential, as well as an ID of some sort to uniquely identify a given node. For fast lookups of set membership, we also had the node structures store boolean values of whether the node was in the heap or in the MST. A notable omission was any storing of a previous node. We decided that since our concern was only in the weight of the tree and not its actual elements, we could save some space by simply ignoring this typical feature in Prim's algorithm. This might have been a useful feature to have had in testing, but ultimately did not impact our production of results. The node structures would also go on to have "heapIdx" and "coords" properties, the motivation and details of which are described in detail below and in section 3.6. With nodes and the heap implemented, Prim's itself came quickly and naturally - almost appearing identical to the pseudocode. Testing revealed greater complications that had to be dealt with, however. I had to return to my heap implementation to handle "updating" nodes, without actually inserting duplicates. Though the nature of my data structure with pointers to nodes allowed these to be updated without traversing the heap, the subsequent manipulation of the heap to accommodate the changed node did not come readily. To do so I added in additional tracking of a node's index within the heap, so I could perform a "heapify-up" operation on the changed node when it was reduced to accurately send it up the heap as needed. We would encounter and debug many other issues in this approach as we tested (see 3.6 "What we learned"), but this largely reflects our initial decision making process regarding the implementation of Prim's algorithm. The last remaining major portion of the programming fell on randomization.

3.3 Random Number Generation

For our random number generation we use the rand() and srand() functions in stdlib.h. After some online research, we quickly became aware that rand() would produce the same sets of pseudo-random numbers on each running of the program unless it was seeded with different values. This can be done using srand(time(0)) at the beginning of the program. time(0) returns the time in seconds since January 1st 1970, and srand(time(0)) seeds rand() with this value. This means that as long as every time the program is run to collect data, 1 second has passed since the previous program ran, then the seeding will generate new pseudo random numbers. We ensured when we collected data that this was satisfied. I believe that the only way for our random number generation to work incorrectly would be if we incorrectly seeded it on each running of the program. Since we ensured the time between the start of the running of consecutive programs was more than a second, I believe the seeds used were all different and thus I trust the pseudo random numbers that were generated. It is worth noting, however, that rand() is a fast pseudo random number generating algorithm and thus does not create numbers that are as random as other algorithms. We accepted this downside when writing our program as we did not believe it would skew our results by a significant amount.

3.4 Explanation for growth rates of $f(n)$

In the following explanation I will propose an intuitive explanation for the trend of $f(n)$ that we found from our data. From the analysis of our data we know the following, where c_1, c_2, c_3 and c_4 are constants:

$$Random\ Weight : f(n) = c_1$$
$$Dim\ 2 : f(n) = c_2 \cdot n^{\frac{1}{2}}$$
$$Dim\ 3 : f(n) = c_3 \cdot n^{\frac{2}{3}}$$
$$Dim\ 4 : f(n) = c_4 \cdot n^{\frac{3}{4}}$$

I have removed the exact value of the constant terms that we found because they are irrelevant for the intuition I am proposing. I shall first deal with the random weight case as this is different in nature to the rest (because all weights are random numbers between 1 and 0 as opposed to euclidean distances). Because the weights of all the edges in the graph take a uniform random distribution, as more nodes are added, and thus more edges are added, the lowest possible distance between nodes should decrease. This suggests that a close to constant MST weight could be accurate.

For the higher dimensional cases, we can see that the growth rate of the weight of an MST with respect to the number of vertices to be spanned n increases as dimension increases (as higher dimension cases have higher exponents of n). If we consider the average distance between two points with random coordinates within a unit square and unit cube, it follows that the average distance should be larger in the the unit cube because in the Pythagorean formula there is another squared term to be added (the difference in the z coordinates of the two points). This logic can be applied again to compare the average distance between two random points in a unit cube and unit hyper-cube and conclude that the average would be greater in the hyper-cube. Thus it seems intuitively correct that if we increase the number of vertices in a graph in a higher dimension, the weight of the MST formed in the higher dimension would increase at a faster rate than the MST formed in the lower dimension. Thus we believe the values of $f(n)$ that we found for the trees in different dimensions makes some sense.

3.5 Algorithm running time

We found the following approximate running time for our algorithm in different dimensions and sizes of input. These values would vary from computer to computer but they allow us to draw some conclusions about our algorithm and comparison between cases. Our algorithm ran incredibly quickly for cases of of $n < 8192$ so we did not consider these cases.

	Dimension			
Vertices	0	2	3	4
8192	0.98	1.15	1.36	1.51
16384	1.87	3.56	4.53	5.41
32768	6.83	13.10	16.71	20.45
65536	26.20	58.92	65.54	72.08
131072	106.80	218.95	265.02	318.50
262144	426.24	917.5	1082.74	1310.72

The above running times would seem to make sense if we consider the work the computer has to do when running our program. Notice in particular that when the input size doubles, in almost all cases, the running time close to quadruples, suggesting an asymptotic running time of roughly $O(n^2)$. This is slower than the regular running time of Prim's algorithm that is $O(m \cdot log_{\frac{m}{n}}(n))$. This difference is slightly puzzling; however, it is most likely due to the fact that we are generating the edges of the graph when needed as the algorithm runs. This includes using functions such as rand() and sqrt() whose running times we do not know. Overall however we were very pleased with our running times. We never had to discard edges because our program ran fast enough to compute all the data that we needed in an appropriate amount of time. We were pleased it only took around 22 minutes to run the 4 dimensional case when $n = 262144$ and we were even able to run it on higher cases of n. As is discussed in more depth later in the next section, we believe one of the big reasons we achieved such fast running times was good use of pointers that avoided unnecessary copying of large data structures.

3.6 What we learned from the assignment

This coding assignment was an excellent learning experience for my partner and I. Neither of us had ever had to write a program that would scale to such large input sizes and so we learned a lot from embarking on such a project. Below we have outlined some of the major things we learned.

We conducted our coding assignment in C. The main reason we did this is because it is the language that together we know best. Additionally, we believed we would be aided by its very low level nature. We had both conducted coding projects in C before, however nothing that had to handle an input size as large as our program did for this assignment. This led us to run into issues and solve issues that have changed the way we think about coding problems and solutions. For example, we began by storing the random graph in an adjacency matrix. We wrote all the code for the project including the entire Prim's algorithm implementation and began testing. Our algorithm worked brilliantly for low values of n, however as n increased, we noticed the algorithm ran slower and slower until finally it failed to run to completion and the program was killed. Upon inspection of the load on our computers when the program was running, we quickly discovered that our computers ram and disk space was being filled by the program when it was constructing the adjacency matrix. This was obviously going to happen if we had considered the size of n that the program was going to have to run for and the corresponding amount of memory needed to store a matrix of size $n \cdot n$. From this we learned a crucial lesson, consideration of the cases under which a program will run must be taken into account during the design phase, before you start coding a solution that very well may (and in our case did) break down at some point. We solved the above problem by only generating and storing the required edge weights of the edges connected to the MST and thus the only edges that Prim's algorithm needs to consider at any given step.

Another excellent learning experience was the importance of the use of pointers. In this assignment, because the data structures we were using were so large for large cases of n (the heap for example), we were aware that providing such data structure as an input to a function would require the copying of that data structure. This could slow down the running time of our program massively, and so we were fairly meticulous in the use of passing pointers to structures as opposed to structures themselves into functions. This problem of decreased running time associated with passing large structures to functions was something neither of us had encountered or had to consider previously because none of our coding projects had to work with such large structures and amounts of data.

The experience with random number generation was new to us as well. Although we were aware computers can only produce pseudo random numbers, we were not aware of the details and nuances of seeding of random number generators and the things that have to be taken into account when using them. This is discussed in a previous section.

Testing our code more rigorously was another new experience for us. We quickly learned that testing the overall algorithm was of little use if it was not working, and instead it is prudent to test individual functions, then groups of functions etc. For example, we had a tiny error in one of our Heap functions that caused it to form invalid heaps (when a child was smaller than a parent) in certain circumstances but not others. We began by testing our complete program and found that it was finding incorrect MSTs. It then took several hours to find exactly where the fault was. If instead we had started by testing individual heap functions and collections of heap functions (possibly earlier on in the coding process), we would have found the fault far quicker.

Another insidious problem that took tremendous debugging efforts and reshaped our understanding of coding was the result of references to local pointers. We discovered that some values, such as the size of the heap, were changing inexplicably to something entirely different, even between functions when nothing else was called. Seeing as no code that we had written was explicitly manipulating these values, we were mystified by their changes. After a great deal of time just reaching the certainty that we were not ourselves accidentally changing those values, we conducted some reading online to discover that we had stored pointers to local variables within our heap struct, such that the compiler did not recognize an issue, but would also inappropriately use the memory at that address. It was a simple fix but a very difficult problem to discover and understand.

We also learned a great deal about data collection and analysis. To collect the data we needed, we placed appropriate print statements in the program. We then wrote a shell script that would run the program for all the inputs types we needed (varying sizes of n, trials and dimensions) and ran this shell script on our computers overnight and during a day. It was an interesting learning experience to have to plan the exact data we needed and how we were to collect it. We knew it would take sizable time to collect such large amounts of data and so had to plan appropriate deadlines so we left ourselves enough time to complete the coding, collect the data, analyse it and conduct the write-up.