CS 124 Programming Assignment 1: Spring 2023

Your name(s) (up to two): Gabriel Chiong

Collaborators: None

No. of late days used on previous psets: 0
No. of late days used after including this pset: 0

This document contains the accompanying write-up to my implementation of a Minimum Spanning Tree (MST) algorithm for complete, undirected, and random graphs. The goal of this experiment was to investigate how the average weight of the MST grows as a function of n. The experiment was carried out on four different variations of the complete, undirected, and random graph:

- Complete graphs on n vertices, where the weight of each edge is a real number chosen uniformly at random in the interval $[0, 1]$.
- Complete graph on n vertices where the vertices are points chosen uniformly at random inside a unit square (the weight of an edge is the Euclidean distance between two x and y coordinates).
- Complete graph on n vertices, where the vertices are points chosen uniformly at random inside a unit cube (3 dimensions), and edge weights determined by the Euclidean distance between two vertices.
- Complete graph on n vertices, where the vertices are points chosen uniformly at random inside a hypercube (4 dimensions), and edge weights determined by the Euclidean distance between two vertices.

Overview

The complete implementation for this experiment is separated into the main program (`randmst.cpp`), and the three sets of header and implementation files (`random_graph.h`, `random_graph.cpp`, `prims.h`, `prims.cpp`, `min_heap.h`, and `min_heap.cpp`). There is also a bash script (`time.sh`) to automate the collection of running times and average MST weights for a number of trials in each dimension, a `Makefile` to automate the compilation of these different files, and a suite of unit tests to verify correctness of my implementation on some toy examples (`tests.cpp`). The instructions to execute the program can be found in the file `README.md`.

We have decided to use an adjacency list to represent the graph. Although a complete graph has $\Theta(n^2)$ edges which makes adjacency list and adjacency matrix asymptotically (both time and space-wise) equal for the purposes of our implementation, an optimization which will be described in a later section will provide a performance enhancement for the adjacency list data structure. To generate random real numbers over a uniform distribution, I decided to use the Mersenne Twister pseudorandom number generator over more conventional options from the C++ library, for reasons which will also be described in a later section.

My implementation of Prim's MST algorithm uses a binary heap to determine the next vertex to be explored. One small space optimization I made to my binary heap implementation was to allow an "eager" `insert(key, vertex)` operation, which essentially decreases the priority (distance) of a vertex without re-inserting a separate copy with a lower key. To do this, I maintained a collection of pointers to element positions in the heap. It is possible to obtain better run times using a Fibonacci Heap, but through profiling my program, the bottleneck was not my implementation of Prim's MST algorithm, but constructing the graph data structure. Therefore, I decided against spending time to further optimize Prim's MST algorithm and my binary heap implementation in favor of other improvements.

Graph Pruning

Other than the minor optimization to the binary heap's `insert(key, vertex)` operation, the most effective method of optimizing this program is the pruning of edges from the complete tree. The idea is that when constructing the complete, undirected, random graph, we will not include edges with weight beyond a certain threshold, $k(n)$, where the threshold is a function of the number of vertices n. The crux of this optimization is to choose a $k(n)$ such that the total MST weight is no different than when without pruning the graph. Since a MST is a tree which only includes edges of minimal weight, a complete graph will have many redundant edges of higher weights which are effectively unused. These higher-weight edges can be discarded without affecting the total MST weight.

There are $\binom{n}{2} = \Theta(n^2)$ edges in a complete graph, so assuming that an edge is represented by 16 bytes, a graph with $262,144$ vertices would require in excess of one terabyte to store. This is simply impractical for some modern computers, and can also be impossible to store on older hard disk drives. Even if we were to simplify our representation of edges to a single floating point type of four bytes, it would still take around 275 gigabytes to represent the same graph. Graph pruning would not improve this for adjacency matrices, so it can be challenging to use for complete graphs as n increases to large numbers. This is also the reason why I have decided to use an adjacency list representation.

To determine the appropriate $k(n)$ to use, we run our program to find the average MST weight of complete graphs with small values of n. We also keep a measure of the average edge weight, and maximum edge weight in the graph, for different dimensions $0, 2, 3, 4$. We recorded the time taken to run our program on a complete graph to compare the magnitude of runtime improvement once we include our optimization, as described in a later section. Since we included all $\Theta(n^2)$ edges for this part of the investigation, we could only check these values for graphs with $n \leq 8192$. The following results were obtained by averaging the results over five trials, to four decimal places.

Dimension	n	Time (s)	Avg. Edge Weight	Max. Edge Weight	Avg. MST Weight
0	128	0.0015	0.4981	0.9999	1.1008
	256	0.0035	0.4989	0.9999	1.1371
	512	0.0099	0.5002	0.9999	1.1984
	1024	0.067	0.5000	1.0000	1.2315
	2048	0.342	0.4999	1.0000	1.2253
	4096	1.427	0.5000	1.0000	1.1904
	8192	6.495	0.4999	1.0000	1.2047
2	128	0.0006	0.5337	1.3218	7.5850
	256	0.0021	0.5127	1.3051	10.6616
	512	0.0085	0.5242	1.3665	15.0130
	1024	0.0677	0.5209	1.3749	20.9991
	2048	0.3291	0.5209	1.3858	29.7674
	4096	1.3982	0.5228	1.3975	41.7826
	8192	6.3833	0.5021	1.3951	58.9967
3	128	0.0006	0.6622	1.4830	17.6765
	256	0.0021	0.6688	1.5245	27.7794
	512	0.0094	0.6633	1.5763	43.7326
	1024	0.0670	0.6626	1.5700	68.0662
	2048	0.3239	0.6627	1.6391	107.1110
	4096	1.3738	0.6643	1.6563	169.8030
	8192	6.4295	0.5519	1.6738	266.8760

4	128	0.0006	0.7758	1.5875	28.4530
	256	0.0021	0.7795	1.6902	46.9175
	512	0.0082	0.7763	1.7069	78.1144
	1024	0.0647	0.7778	1.7484	129.7120
	2048	0.3268	0.7780	1.7845	217.3800
	4096	1.3669	0.7782	1.8377	361.4590
	8192	6.3237	0.6484	1.8502	603.4870

Table 1: Results without graph pruning.

Armed with the knowledge of these results, we then employ a trial and error approach to estimate appropriate values of $k(n)$ by discarding edges of weights greater than $k(n)$, when constructing a graph. Our implementation throws a runtime error if Prim's MST algorithm was unable to reach all vertices, which signals if we might have been too liberal in our discarding of edges. At each stage, we compare the average MST weight found with the complete graph of all edges to ensure that the average MST weight stayed the same. On a side note, the average edge weight for the 0-dimension graph is roughly 0.5 across all values of n trialed, which is a positive sign for our choice of pseudorandom number generator (since the 0-dimension case is when the edge weight is chosen uniformly at random over the real numbers from 0 to 1). The following table shows the final functions $k(n)$ we will use for our graph simplification.

Dimension	$k(n)$
0	$8n^{-0.8} - 0.00025$
2	$2.1n^{-0.45}$
3	$2.3n^{-0.3} - 0.013$
4	$2.8n^{-0.276}$

Table 2: Threshold functions for a given n on each dimension.

Other Implementation Details

The two most popular algorithms for finding a MST are Prim's and Kruskal's algorithms. Prim's has a time complexity of $O(|E| \log |V|)$ when paired with a binary heap, which is $O(|V|^2 \log |V|)$ for a complete graph with $|V|^2$ edges. Kruskal's algorithm has a time complexity of $O(|E| \log |E|) = O(|V|^2 \log |V|^2) = O(|V|^2 \log |V|)$ for a complete graph, with a factor of 2 hidden by the big-O notation. When recording run times in practice, it may be prudent to also consider constant factors. Although they have equal asymptotic run times for a complete graph, when used on graphs with more edges than vertices (i.e. $|V| \leq |E|$), Prim's algorithm should have a better practical performance, since it is less dependent on the number of edges. It is for this reason that we have chosen to use Prim's algorithm for our implementation.

To continue our discussion on the benefits of the Mersenne Twister pseudorandom number generator over the more well-known `std::rand()`, we note the relatively good "uniform" results obtained in the previous section for the average edge weight in a 0-dimension graph (approximately 0.5). Although the details are not included in this write-up, when swapping out the Mersenne Twister generator for something based on `std::rand()`, we found a skew towards lower numbers. In addition to this benefit provided by the Mersenne Twister for our use case, reading the documentation for `std::rand()` also revealed its other disadvantages - low randomness of lower-order bits, a short period, and the low value of `RAND_MAX` (the range of values at which it generates random numbers). We have therefore chosen to use the Mersenne Twister generator for our implementation.

Results and Analysis

The following section presents the results of our program on the different dimensions and number of vertices of the graph. Note that these results include the graph pruning technique to speed up the simulation. We begin with a description of the machine used to compile and run the program.

Machine Specifications

The following results were obtained using a MacBook Pro with the Apple Silicon M1 chip and 16GB of memory. Even though the Apple Silicon M1 chip is based on the ARM architecture, our code was compiled to target a x86-64 instruction set. Although we have found through some trial and error that it is possible to obtain faster running times when compiled directly for the ARM instruction set, we have decided to continue with compiling for a x86-64 system due to it being more ubiquitous, and therefore having a higher chance of obtaining more reproducible results outside of our own machine.

Running Times with Optimizations

With the optimizations provided in our previous section, we now provide the running times for different values of n and dimensions for each graph. The results below were averaged over five trials, and are presented in seconds, to four decimal places. There is a clear improvement in time taken when compared to Table 1. Where a graph of size $n = 8192$ used to take 6 seconds without optimizations (for all dimensions), it now takes less than 1 second. In fact, apart from the 0-dimension graph which took 167.6 seconds on a graph with $262,144$ vertices, all other dimensions with $262,144$ vertices took under one minute.

	Dimension			
n	0	2	3	4
128	0.0007	0.0002	0.0004	0.0010
256	0.0012	0.0004	0.0007	0.0018
512	0.0024	0.0009	0.0017	0.0027
1024	0.0051	0.0021	0.0039	0.0046
2048	0.0145	0.0054	0.0096	0.0114
4096	0.0497	0.0157	0.0259	0.0328
8192	0.1884	0.0511	0.0789	0.0831
16384	0.7372	0.1840	0.2510	0.2540
32768	2.8408	0.6953	0.8724	0.8393
65536	11.2875	2.7424	3.0340	3.0319
131072	43.5603	10.6255	11.2835	11.5163
262144	167.6000	41.4627	42.6170	42.9668

Table 3: Running times with optimizations, in seconds.

From the results of this table, it appears that the 0-dimension graph is the slowest of the four types of graphs. We hypothesize that it is more about implementation detail, rather than being specific to the algorithm or graph structure. Since the edges of the $2, 3, 4$-dimension graphs are calculated as a Euclidean distance, the random graph vertex point values are first generated in a vector and this allows the C++ compiler more freedom to optimize. However, the 0-dimension graph is constructed by simply generating random weights and assigning it to each edge in sequence, which limits the optimization freedom of the C++ compiler. A graph of the running times are provided below for reference.

A polynomial of degree two appears to fit the data quite well, which correlates with our hypothetical runtime, since we require that each pair of vertices be considered for placing an edge. Even though an

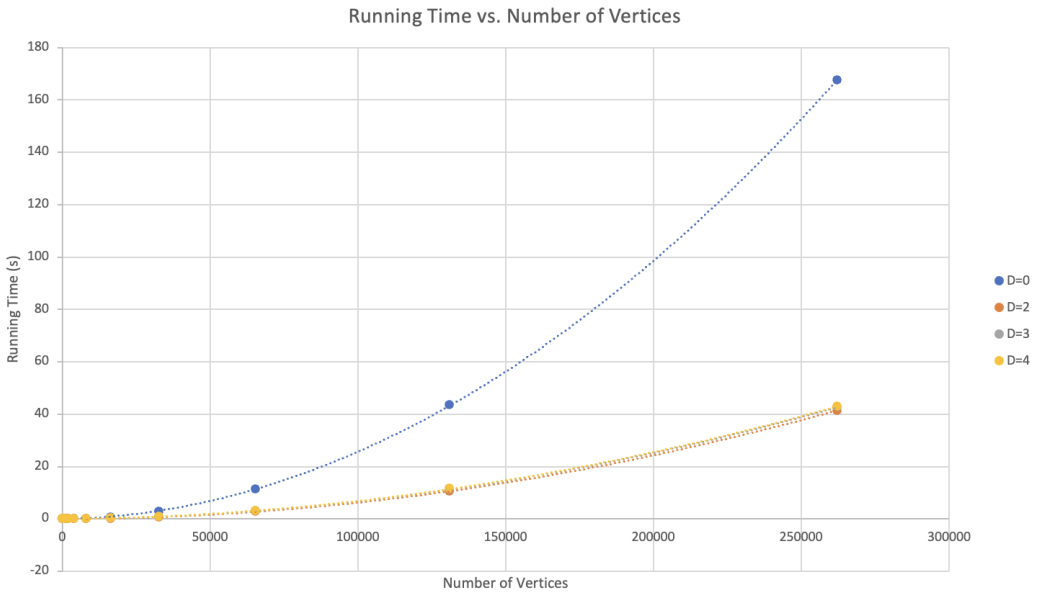

Figure 1: Running time versus the number of vertices, and a polynomial trendline of degree two.

edge might not be placed in the final form of the graph due to our pruning of higher-weight edges, the construction of the graph still requires that each edge between a pair of vertices be considered. This dominates the running time of Prim's Algorithm, which is log-linear, and we have a useful sanity check for our implementation.

0-Dimension Results

The results of the experiment for the 0-dimension graph are presented in the table below. Each value of n was trialed five times and the reported result is the average over the five trials.

n	Average MST Weight
128	1.37112
256	1.20854
512	1.21125
1024	1.20013
2048	1.20768
4096	1.20211
8192	1.21539
16384	1.19972
32768	1.20407
65536	1.20386
131072	1.20199
262144	1.20237

Table 4: The average dimension-0 MST weight for given n, averaged over five trials.

From these results, it is clear that the average weight of the MST does not appear to change as n increases, for 0-dimension graphs. Furthermore, the weight of the MST is very similar to the results from a graph without pruning the higher-weight edges in Table 1, so we have good confirmation that our function

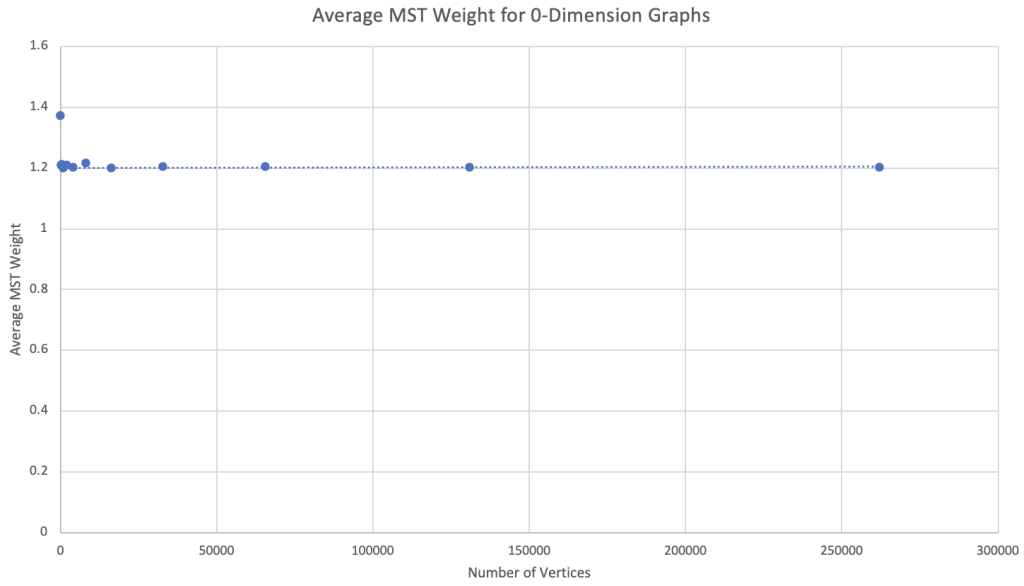

Figure 2: Average dimension-0 MST weight versus the number of vertices.

$k(n)$ for the 0-dimension case has not been overly rash in discarding edges. Using the graph of results in Figure 2, we can confidently suggest that the total MST weight as a function of n in the 0-dimension case can be modeled by $f(n) = 1.2$ when n grows large. We believe that a constant function $f(n)$ represents the best fit for the results since there is minimal change in $f(n)$ as n increases. Figure 2 confirms our observations from the table of results.

It does appear strange that the MST weight would not increase with an increasing number of vertices, and this could be a topic for further investigation outside the scope of this experiment. However, one could imagine that with an increasing number of vertices, there are a larger amount of very small edge weights that are retained (while the higher-weighted ones are discarded anyways) and this contributes to maintaining a constant MST weight, even as n increases.

2-Dimension Results

The results of the experiment for the 2-dimension graph are presented in the table below. Each value of n was trialed five times and the reported result is the average over the five trials. From the results in Table 5, the average MST weight appears to increase with increasing n, for 2-dimension graphs. Also, the average MST weights appear to be quite similar to the results without optimization from Table 1, so it seems that our choice of $k(n)$ for the 2-dimension case has not been overly rash in discarding edges.

We used a trial and error approach in conjunction with some suggestions from Microsoft Excel to fit different trendlines onto the results we have obtained, and the best fitting function to model the total MST weight as a function of n was $f(n) = 0.65\sqrt{n}$. This trendline is shown in Figure 3. From this, we can observe that the asymptotic growth rate of the total MST weight $f(n)$ is sublinear in the number of vertices. This result is more in line with our expectations that the weight of the MST would increase with the number of vertices, since the number of edges in the graph also increase. Also, we expect that dimension-2 graphs would have higher edge weights compared to dimension-0 graphs since dimension-2 edge weights are chosen in the interval $[0, \sqrt{2}]$. As expected, when compared with the MST weights of the 0-dimension graph, the MST weights for the 2-dimension graph are larger, with the smallest being approximately 7.65 (when $n = 128$), compared to the constant value of 1.2 in the 0-dimension graph.

n	Average MST Weight
128	7.6534
256	10.7343
512	15.0747
1024	21.0580
2048	29.6319
4096	41.9484
8192	59.0455
16384	83.2291
32768	117.405
65536	166.036
131072	234.686
262144	331.716

Table 5: The average dimension-2 MST weight for given n, averaged over five trials.

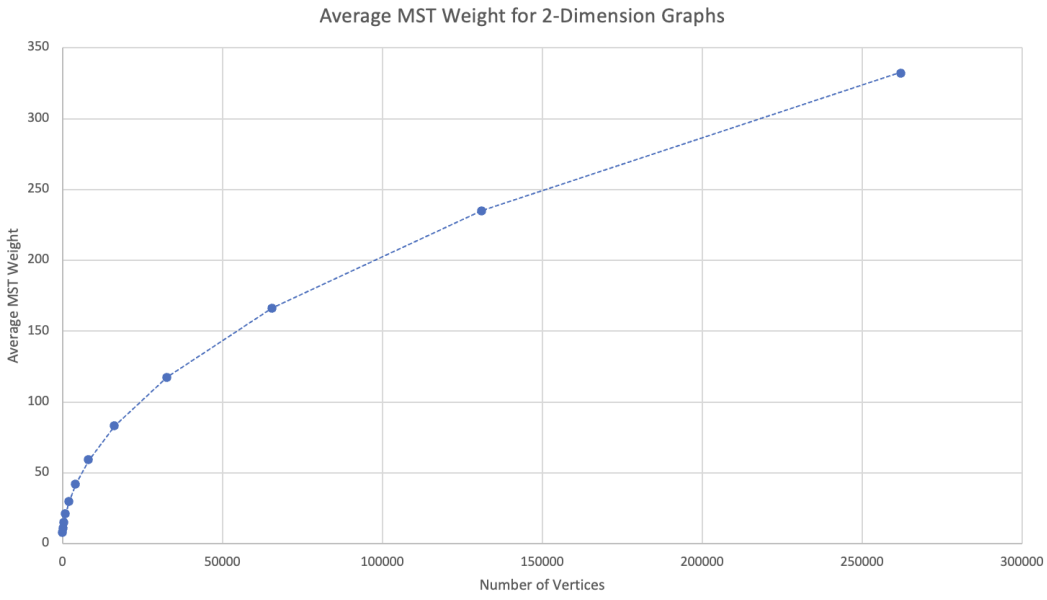

Figure 3: Average dimension-2 MST weight versus the number of vertices.

3-Dimension Results

The results of the experiment for the 3-dimension graph are presented in the table below. Each value of n was trialed five times and the reported result is the average over the five trials.

n	Average MST Weight
128	17.3588
256	27.7157
512	43.1013
1024	67.9202
2048	107.492
4096	168.859
8192	267.254
16384	422.801
32768	669.084
65536	1058.58
131072	1677.7
262144	2657.73

Table 6: The average dimension-3 MST weight for given n, averaged over five trials.

From the results in Table 6, the average MST weight appears to increase with increasing n. Also, the average MST weights appear to be quite similar to the results without optimization from Table 1, so it seems that our choice of $k(n)$ for the 3-dimension case has not been overly rash in discarding edges.

Again, we used a trial and error approach in conjunction with suggestions from Microsoft Excel to fit different trendlines onto the results, and the best fitting function to model the total MST weight as a function of n was $f(n) = 0.7n^{0.66}$. This trendline is shown in Figure 4. From this, we can observe that the asymptotic growth rate of the total MST weight $f(n)$ is sublinear in the number of vertices. This result is in line with our expectation for the same reasons outlined in the section for the 2-dimension case.

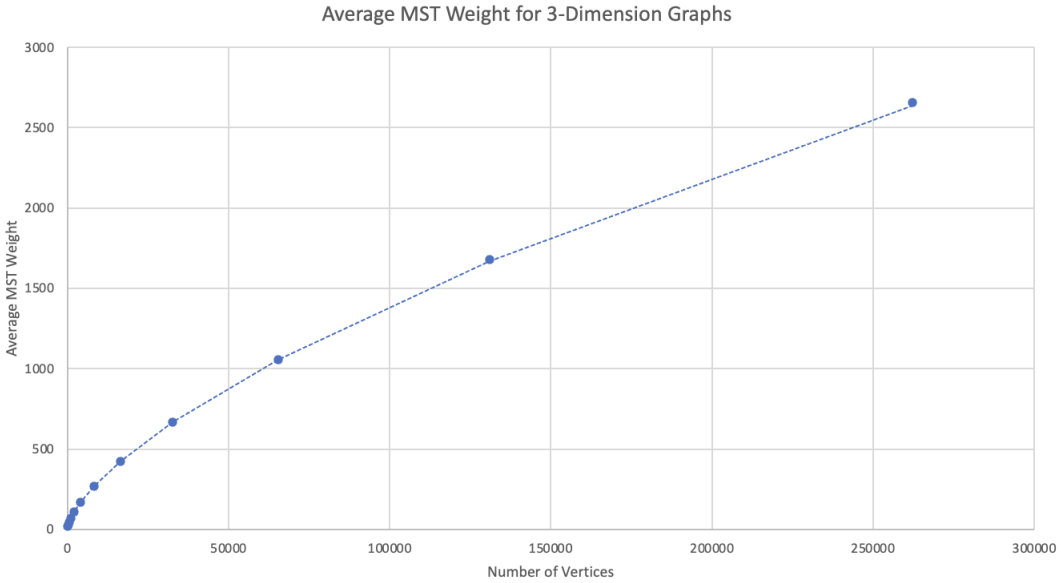

Figure 4: Average dimension-3 MST weight versus the number of vertices.

4-Dimension Results

The results of the experiment for the 4-dimension graph are presented in the table below. Each value of n was trialed five times and the reported result is the average over the five trials.

n	Average MST Weight
128	28.2784
256	47.2223
512	77.8771
1024	130.696
2048	216.834
4096	361.491
8192	602.84
16384	1008.47
32768	1688.12
65536	2829.55
131072	4742.31
262144	7948.04

Table 7: The average dimension-4 MST weight for given n, averaged over five trials.

From the results in Table 7, the average MST weight appears to increase with increasing n. Also, the average MST weights appear to be quite similar to the results without optimization from Table 1, so it seems that our choice of $k(n)$ for the 4-dimension case has not been overly rash in discarding edges.

Again, we used a trial and error approach in conjunction with suggestions from Microsoft Excel to fit different trendlines onto the results, and the best fitting function to model the total MST weight as a function of n was $f(n) = 0.78n^{0.74}$. This trendline is shown in Figure 5. From this, we can observe that the asymptotic growth rate of the total MST weight $f(n)$ is sublinear in the number of vertices. This result is in line with our expectation for the same reasons outlined in the section for the 3-dimension case.

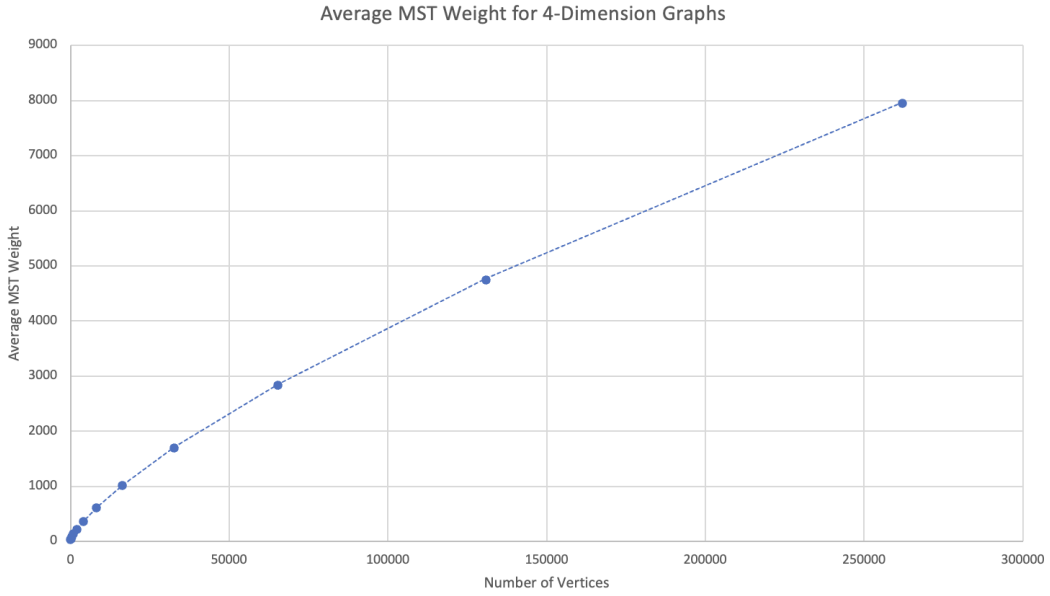

Figure 5: Average dimension-4 MST weight versus the number of vertices.

Conclusion

In summary, we have found the following functions as a suitable way to model the MST weight of a graph as the number of vertices increase. For 0-dimension random graphs, the function $f(n) = 1.2$ appears to be a good fit for our results. For 2-dimension random graphs, the function $f(n) = 0.65\sqrt{n}$ seems to be a good fit for our results. For 3-dimension random graphs, the function $f(n) = 0.7n^{0.66}$ appears to be a good fit for our results. For 4-dimension random graphs, the function $f(n) = 0.78n^{0.74}$ seems to be a good fit for our results. In general, for dimensions that are greater than or equal to two, it is observed that functions of the form $f(n) = an^b$, for some real numbers $0 \leq a, b \leq 1$, appear to be a good fit for our results. Furthermore, it appears that a, b appear to grow with the dimension of the graph.

From our results, the running time of the program seems to be dominated by the time taken to construct the graph. Our implementation of Prim's Algorithm and its accompanying priority queue were (predictably) swallowed by the time taken to construct the graph. This was also observed by profiling our running program for various values of n and dimensions. It was interesting to implement a variation of the priority queue that allowed for `insert(key, vertex)` to change priorities of elements that are already in the heap, based on the suggestion in the lecture notes to maintain a collection of pointers to element positions in the heap.

This experiment has also taught us an important lesson in using pseudorandom number generators, something we have previously taken for granted. It seems like they are not all created equal, and generators like the Mersenne Twister should be the preferred choice over the classical `std::rand()`. As described in previous sections, using the `std::rand()`-based engine resulted in a skew towards lower numbers, while the Mersenne Twister engine provided a fairer distribution of numbers.

A further optimization that we would have been keen to implement if we had more time, is to parallelize the edge construction of the random graph. We have found through profiling that this was the bottleneck in terms of running time, so targeting this area for improvement would have been an interesting exercise to pursue. We carried out a minimal spike to investigate the benefits of using multithreading to construct the adjacency list of the graph, and found approximately 2x improvement in running time when compiled to an ARM instruction set. Interestingly, the same code compiled to a x86-64 instruction set saw slightly increased running times (granted, the implementation might not have been perfect). Since we had decided to orientate this experiment towards the x86-64 system for reasons outlined in the previous sections, we did not pursue this interesting line of enquiry further.