ANLY 550 Programming Assignment 2 Report

Mengtong Zhang, Yunjia Zeng

April 15, 2019

Abstract

The goal of this assignment is to find appropriate functions $f(n)$ to model the weight of the minimum spanning tree (MST) of random, complete graphs of n edges in: 0-dimension, 2-dimension (unit square), 3-dimension (unit cube), and 4-dimension. We implemented a random graph generator and used Prim's algorithm to produce the MST of those graphs. Our results show that for 0-dimension graph, the total weight of MST is a constant. And for 2-dimension, 3-dimension and 4-dimension graphs, the total weight of MST is approximately a polynomial function of node number.

Quantitative Results

Results of MST Weight

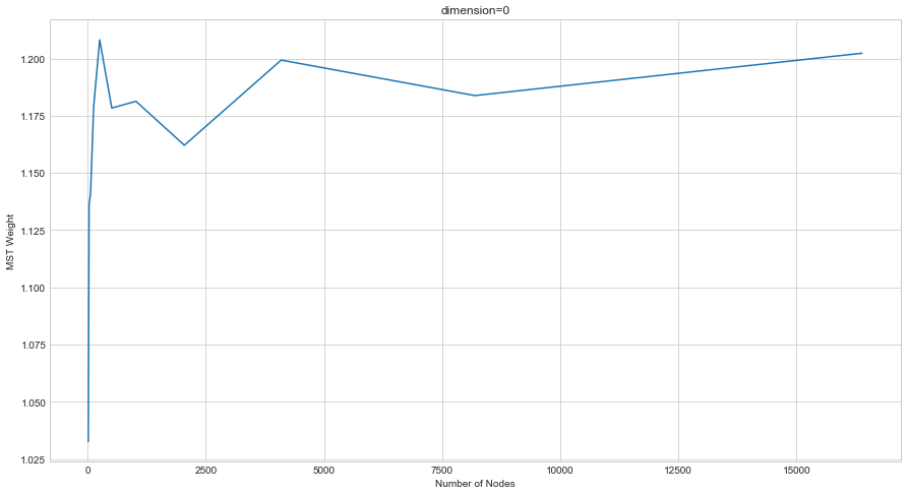
We ran our program a series of times on different values of n and d, following the guidance of the assignment description for testing the powers of two. The results are recorded in the following table.

Table 1: Average Weight of MST with n Vertices and d-dimension

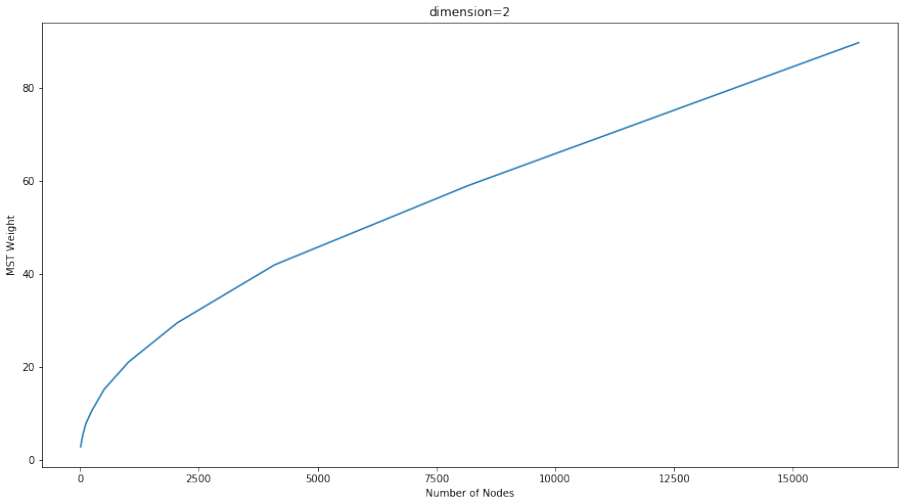
n	**trails**	**d = 0**	**d = 2**	**d = 3**	**d = 4**
16	1000	1.03264	2.7525	4.8219	6.1919
32	1000	1.13594	3.6911	6.8844	10.4589
64	1000	1.14144	5.4180	11.1707	17.2867
128	500	1.17878	7.7993	17.6633	28.2978
256	200	1.20814	10.5818	27.5700	46.9136
512	200	1.17837	15.1131	43.1396	78.2168
1024	100	1.18130	20.9779	68.3132	129.3008
2048	50	1.16210	29.3978	107.8429	217.1148
4096	10	1.1993	41.8688	169.6389	360.8317
8192	5	1.1838	59.0073	266.8234	602.1123
16384	5	1.1635	89.6512	430.2560	1009.8489

The goal is to find a function $f(n)$ which could model the average weight of the MST given n. We noticed from the above table that the function must depend on both n and d. Therefore, we plot the data in above table by average MST versus n, with specific d, as follows.

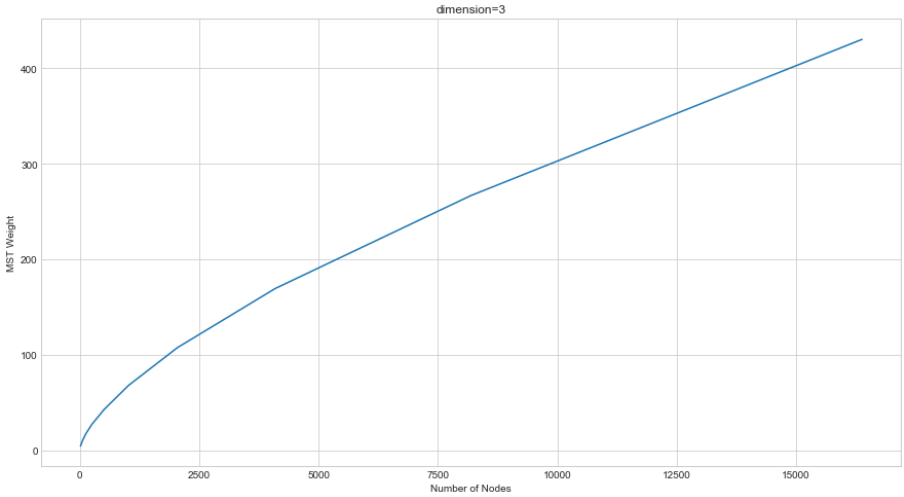
0-dimension: Simple Graphs



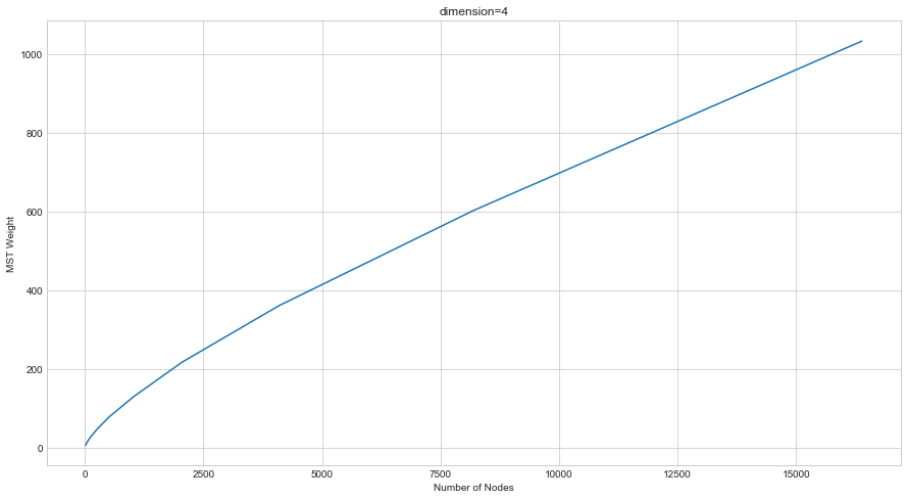
2-dimension: Graphs in Unit Square



3-dimension: Graphs in Unit Cube



4-dimension: Graphs in Hypercube



f(n) Estimation

In this section, we fit the values in the previous quantitative results section to estimate $f(n)$. In the graph above, we immediately notice that $f(n)$ depends on both n and d. For $d = 0$, if we ignore some noise at small values of n, it is around 1.2. For $d = 2$, $d = 3$ and $d = 4$, the figure looks like a polynomial form. Thus, we assume that for $n \neq 0$,

$$f(n) = an^b.$$

We did not include a constant intercept because the figure will definitely start from (0,0). Notice that under this form,

$$log(f(n)) = blog(n) + log(a)$$

It is reasonable to fit a linear regression model between observational $log(f(n))$ and n. The estimated parameters are:

Table 2: k(n) for d = 4

	d=2	**d = 3**	**d = 4**
a	0.4839	0.6527	0.7572
b	0.6546	0.6855	0.7230

- For graphs with vertices in 0-dimension, the average weight of the MST with different n quickly converges to 1.2, therefore we estimate $f(n) = 1.2$.

- For graphs with vertices in 2-dimension, we estimate $f(n) = 0.65n^{1/2}$.

- For graphs with vertices in 3-dimension, we estimate $f(n) = 0.68n^{2/3}$.

- For graphs with vertices in 4-dimension, we estimate $f(n) = 0.7^{3/4}$.

Discussion

Algorithm Selection

We choose to use Prim's algorithm rather than Kruskal's since it performs better when dealing with a dense graph, as we are working with complete graphs for this assignment. The Prim's algorithm carries the runtime of $O(m \log_{m/n} n)$, where as Kruskal's algorithm runs in $O(m \log n + m \log m)$. Note that the $\log n$ term denotes the number of times to iterate the log base 2 function on a number down to a constant smaller or equals 1 and the $m \log m$ term denotes the time to sort the edges.

Prim's Algorithm

Prim's algorithm is an example of greedy algorithm which grows and MST out of a source node s. At all time-steps a set X will consist of a tree rooted at s and S will be the set of nodes reachable from s via X. For this assignment, we use priority queue implementation which carried out by binary heap for Prim's implementation. Since Prim's algorithm relies on a set, we use an array to store the vertices that has been chosen to be included in the MST. Here we provide the pseudo-code of Prim's algorithm:

Algorithm 1 Prim's Algorithm

```
1: procedure PRIM(G(V, E), s)
2:     dist[v] ← infinity for all v in V, v not equals to s
3:     prev[v] ← NULL for all v in V
4:     dist[s] ← 0
5:     priority queue H ← {(v, dist[v]}
6: loop:
7:     while H is not none: do
8:         V ← delete-min(H).
9:         X ← X ∪ {(v, prev[v]}.
10:        for (v,w) in E and w not in S: do
11:            if dist[w] > weight(v,w): then
12:                dist[w] ← weight(v,w)
13:                dist[w] ← v
14:                insert(H, w, dist[w])
```

Graph Representation

When we are dealing with extremely large graphs, we find that it is necessary to use an adjacency list to store information of edges of the graph. In this way, every time we run $Prim's$ algorithm, we no longer need to loop through the whole graph to find the neighbors of each nodes. Rather, we only need to look up the neighbors of certain node. In this assignment, the adjacency list is implemented as a dictionary of lists. Each key in the dictionary represents a node, and the value is a list of the key's neighbors and the weight of edge between them.

The advantage of using a adjacency list as graph representation is that the size of the list is dy-

namically allocated so that we can remove edges and save in matrix size. The adjacency matrix can take quadratic space as big as 20+ billion bytes memory as we increase the size of n.

Optimization

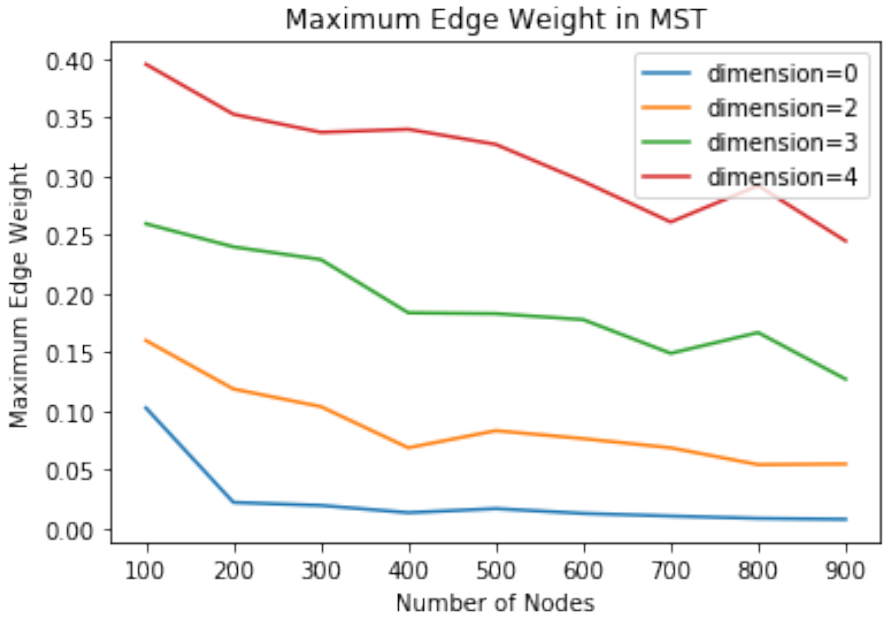
k(n) Estimation

To handle large n, we would like to simplify our graph by removing some large edges that will not be included in the minimum spanning tree. For example, for the graphs in this assignment, the minimum spanning tree is extremely unlikely to use any edge of weight greater than $k(n)$, for some function $k(n)$, where n is the number of nodes in the graph.

To find $k(n)$, we first find the largest edge of MST for small n from 100 to 900. The table and figure below shows the results of largest weight for graph of n from 100 to 900.

Table 3: k(n) by Finding the Largest Edge

n	**d = 0**	**d = 2**	**d = 3**	**d = 4**
100	0.1025	0.1600	0.2595	0.3954
200	0.0220	0.1187	0.2398	0.3528
300	0.0194	0.1037	0.2290	0.3374
400	0.0133	0.0686	0.1836	0.3400
500	0.0167	0.0832	0.1829	0.3271
600	0.0126	0.0764	0.1778	0.2956
700	0.0104	0.0686	0.1491	0.2610
800	0.0084	0.0543	0.1668	0.2919
900	0.0076	0.0548	0.1271	0.2448



Obviously, as dimension increases, the maximum edge weight in MST will increase. In order to have a uniform formula for $k(n)$ for all dimensions, we will use dimension=4 to fit $k(n)$ so that $k(n)$ will be large enough to greater than the maximum weight under all dimensions. Based on the shape of $k(n)$ in the plot below, we assume $k(n) = ae^{-bn}$.
To fit $k(n)$, we collect the maximum weight for different number of nodes which are greater than 1000. They are:

Table 4: k(n) for d = 4

k(n)	**d = 4**
k(2048)	0.2096
k(4096)	0.1819
k(8192)	0.1425
k(16384)	0.1296

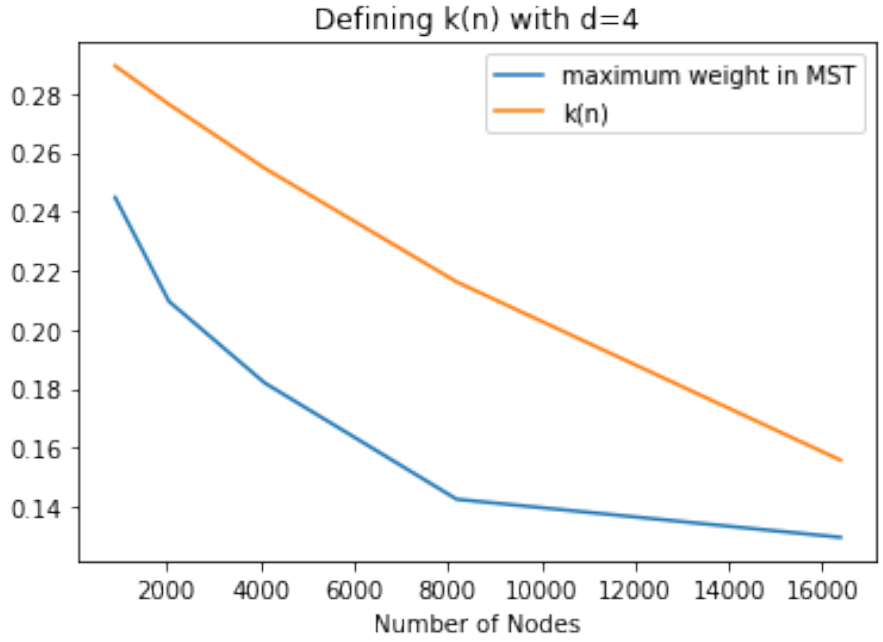
Notice that under the assumption of $k(n)$'s form,

$$log(k(n)) = -bn + log(a)$$

It is reasonable to fit a linear regression of $log(k(n))$ and n. We slightly change the estimated parameters and make $k(n)$ larger to have a safe margin between $k(n)$ and maximum weight, finally we have:

$$k(n) = 0.30e^{0.00004n}$$

We draw a picture with fitted $k(n)$ and the maximum weight in MST for different number of nodes. And we find that when n is between 2000 and 16000, $k(n)$ is above the line of maximum weight. So we can safely use $k(n)$ to cut off heavy edges. Note that we only need to simplify graph with dimension greater than 2000 because the original algorithm is already fast enough to deal with smaller dimensions.



Improvement of Runtime

By throwing the edges bigger than $k(n)$ before constructing MST, we find the efficiency of our algorithm is improved significantly. To illustrate this improvement, we have tried to apply the original algorithm and modified algorithm to $n = 2048, 4096, 8192$ and recorded the runtime.

Table 5: Average running time for finding MST with n Vertices and d-dimension comparison

n,d	Runtime without simplification(s)	Runtime with simplification(s)
2048,0	87.5421	20.5590
2048,2	86.6386	17.1486
2048,3	90.1300	11.7404
2048,4	72.8699	10.8961
4096,0	768.9689	120.4562
4096,2	677.9416	85.1316
4096,3	710.3090	49.4463
4096,4	551.1962	37.3365
8192,0	5423.3233	426.8563
8192,2	5389.6541	418.8919
8192,3	3902.2739	203.2087
8192,4	3023.1248	151.1882

It is obvious that throwing away large weight edges help us improve the efficiency of algorithm significantly. And as the graph becomes more complicated, the improvement in run time becomes more significant.

Memory and Cache Size

For this assignment, the cache on our computers plays a very important role, and it mainly decide the elapsed time of the program. There is certainly difficulty initially when we were handling large n as we would quickly run out of memory. The optimization of picking a threshold weight, i.e. finding $k(n)$, allows us to store fewer edge weights. The optimization helps not only the program running time but also the space complexity. While we have computers with different specification, the one with more memory run faster than the other one.

Growth Rate

Based on the estimation of $f(n)$, we can see that the growth rate of MST weight for dimension0 is 0 because $f(n)$ is a constant for dimension=0. That is probably because even the number of nodes and edges in the graph increases exponentially, the number of extremely light edges available to the algorithm to include in an MST also increases exponentially.
For dimension=2,3,4, we find that $f(n)$ is following a form that

$$f(n) = an^{\frac{d-1}{d}}$$

Furthermore, the polynomial coefficient a is increasing as we increase d. We are not sure if this form holds for higher values of d, if it does, then as $d \to \infty$, $f(n) \to lim_{n\to\infty} a * n.$, which means $f(n)$ will linearly dependent on d. We can safely say the growth rate of MST weight will increase if we increase the dimension of graph, and will approach to a linear growth rate as d approaches to infinity.

Algorithm Run Time

It is very clear that the algorithm run time increases exponentially as n increases, which follows the function that we have derived earlier. We also observed that the higher dimensions will take more time to finish even for smaller values of n. This may due to the number of computations necessary to compute the Euclidean distance, which increases with dimensions. However, the lower dimensions take longer time to compute as n increases. Overall, the run time grow exponentially, which is the same as the order notation that we have determined previously. The run time recorded in the following table are in seconds.

Table 6: Average running time for finding MST with n Vertices and d-dimension

n	**trails**	**d = 0**	**d = 2**	**d = 3**	**d = 4**
16	1000	0.0008	0.0006	0.0012	0.0013
32	1000	0.0027	0.0022	0.0041	0.0047
64	1000	0.0134	0.0089	0.0151	0.0167
128	500	0.0490	0.0404	0.0640	0.0647
256	200	0.2057	0.1972	0.2683	0.2694
512	200	1.1863	1.1239	1.5199	1.4012
1024	100	3.7916	3.3854	2.9299	2.6845
2048	50	20.5590	17.1486	11.7407	10.8961
4096	10	120.4562	85.1316	49.4463	37.3365
8192	5	426.8523	418.8919	203.2087	151.1882
16384	5	3818.2495	2377.8009	2543.4425	2532.1354

While recording the running time of our program, we find that the running time is fluctuating after n grows greater than 4096. The variance of the running time was large, when we ran our program on $n = 16384$, the difference of time recorded could have more than 600 seconds difference. The average running time that we recorded in the table above for $n = 16384$ has somehow low reference value. We definitely need more trails to adjust the average running time of the program over large n, but due to the nature of Python this programming language, the run time is incompatible with programming languages like C or C++. The elapsed time for a single trail for $n = 16384$ is for about an hour, to run it for more trails is too time consuming.