ANLY 550 Structure&Algorithm
Spring 2018
Dr. Justin Thaler
Yigao Li and Tao Yu

Programming Assignment 1 Report

- **Objective:**

For this programming assignment, we will be constructing minimum spanning tree within a complete and undirected graph *G(V,E)* in Python 3. Spanning tree is a subset of edges that is acyclic and connects all of the nodes, and minimum spanning tree (MST) is the one with the least sum of edge weights. During this assignment, our goal is to discover the function that describes a relationship between the number of nodes in *G* and the expected weight of the MST, as well as the time complexity of MST algorithm.

- **Procedure:**

To explore the relationship, we will assign different weight to the edges with different number of nodes. The weights in the first trial will be a random number selected between 0 to 1. Therefore we will call it the 0-dimension trial. In the second trial we will assign the weight of each edge as the geodesic distance between two nodes that are being connected with this edge. Therefore we call it 2-dimension trail. Then we will discuss the situations where the weight equals to the geodesic distance between the nodes in 3-dimensional and 4-dimensional space. We will be using adjacency matrix to represent the weight of each nodes since it suits the situation perfectly. A[i][j](which equals A[j][i] because the graph is undirected) represents the weight of the edge between i and j when $i,j \in V$. Since the graph is complete and undirected, the diagonal entry will all be 0 and all other entry will be non-zero.

- **Selecting Algorithm to Build MST**

There are 2 algorithms that are commonly used in finding MST, Prim's algorithm and Kruskal's algorithm. Because we will randomly generate complete, undirected graphs, the number of edges, $|E|$, is much more than the number of nodes, $|V|$. For example, if $|V| = n$, for a complete graph, $|E| = \frac{1}{2}(n^2 - n)$. Therefore we are working with a very dense graph. The time complexity for both algorithms is $O(|E|log|V|) = O(n^2log(n))$. However, Kruskal's algorithm deals with all edges by sorting and Prim's only works on edges that connect to subgraph rooted at source node, so it is less efficient with dense graph. Since we are going to explore MST on the densest undirected graph, we choose to implement Prim's algorithm with Min heap.

- **List of Tree Size vs. Number of Nodes(*n*)**

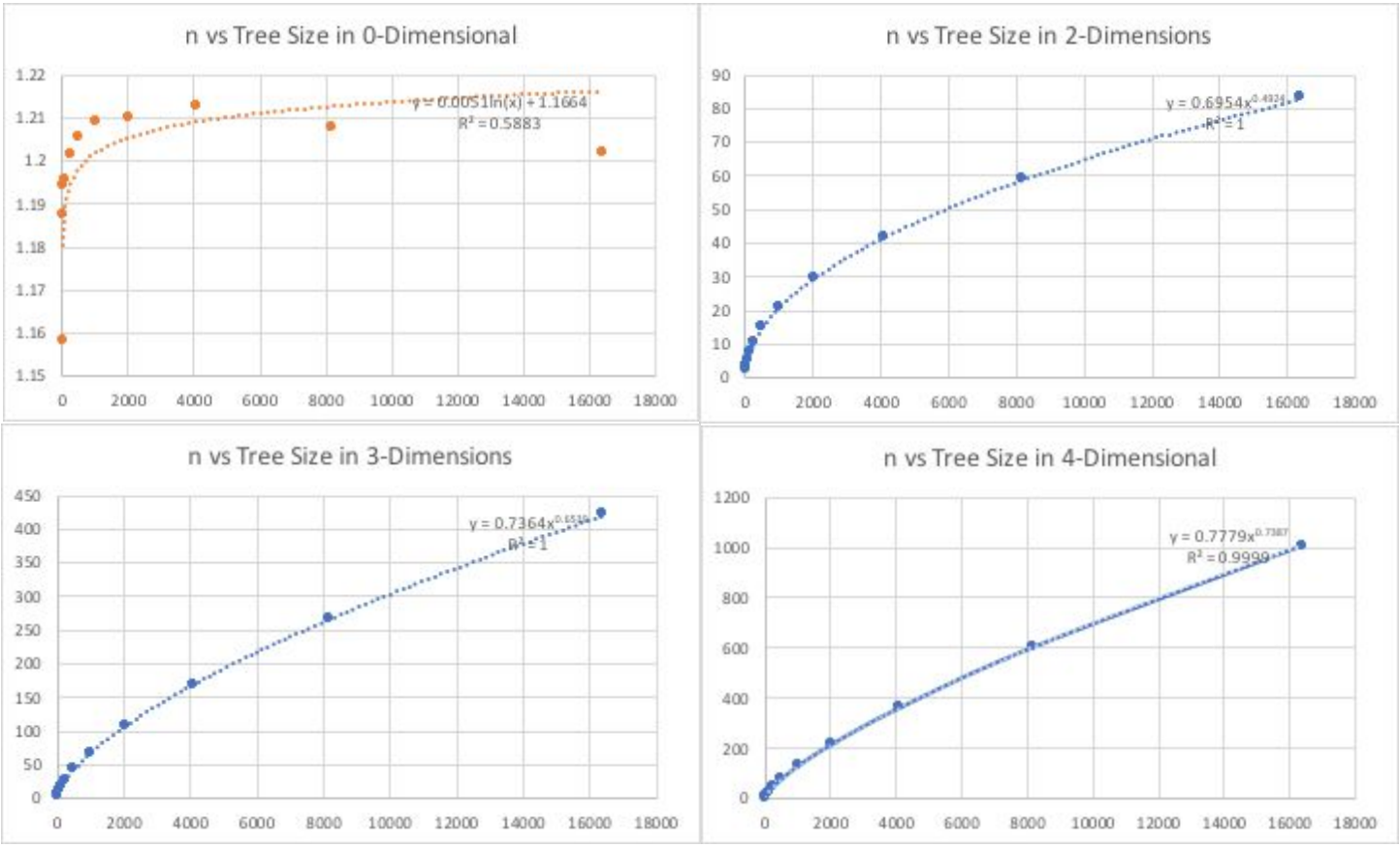
Below is a table of Python outputs about average weight of MST with respect to *n*-dimension graph with power-of-2 number of vertices.

Number of nodes	Tree size(0-dim)	Tree size(2-dim)	Tree size(3-dim)	Tree size(4-dim)
16	1.158156309019	2.708685565476	4.502107335986	5.75404119
32	1.187235199622	3.858370760699	7.155687677755	10.4951435
64	1.19402986547	5.43226204818	11.23495169199	16.8364394
128	1.195603567318	7.612851288692	17.58356104344	28.2831654
256	1.201545540559	10.61224897095	27.53509654972	47.0133492
512	1.197977822327	14.97084066919	43.26399885976	77.8091093
1024	1.18882964	20.92414273302	67.98945210356	130.412336
2048	1.192676057909	29.53738584934	107.2416805388	216.487695
4096	1.212609112232	41.72484696555	168.9285256036	360.696096
8192	1.207490338781	59.02704131872	267.7086290151	603.534958
16384	1.201936382364	83.27140915818	423.0836584198	1007.89859

- **Estimating** $f(n)$

To explore the relationship between number of nodes and tree size, we plot the list above into a scatter plot.

ANLY 550 Structure&Algorithm
Spring 2018
Dr. Justin Thaler
Yigao Li and Tao Yu



After fitting with linear, logistic, exponential, and polynomial equations, we found that power equation gave us the largest R^2 value and the best fit(except for 0-dimensional). With the number of nodes as the x-axis and tree size as the y-axis, The equations we got are:

$y = 0.0051ln(x) + 1.1664$ in 0-dimension.

$y = 0.6954x^{0.4924}$ in 2-dimensions.

$y = 0.7364x^{0.6539}$ in 3-dimensions.

$y = 0.7779x^{0.7384}$ in 4-dimensions.

When dimension = 0, as n gets greater and greater, the tree size fluctuate around 1.2. Considering there are fluctuations and errors in the equations, our estimation for the equations are:

$f(n) = 1.2$ in 0-dimension.

$f(n) = 0.7n^{0.5}$ in 2-dimensions.

$f(n) = 0.75n^{0.7}$ in 3-dimensions.

$f(n) = 0.8n^{0.75}$ in 4-dimensions.

- **Growth Rate Analysis**

The growth rate is surprising to us. First, it grows with a power, which means the number of nodes changes could affect the MST size in a power law distribution. Secondly, the increasing rate is slowing down as the number of dimensions increases: When we have 2-dimensions, the power is 0.5, and in 3-dimensions the power is 0.7, which increases the power of 0.2. And in 4-dimensions the power is 0.75, which only increases 0.05. Therefore we can make a reasonable assumption that in 5-dimensions space the power increase would be even less than 0.05. This could be due to the reason that as dimension increases, the length of each edge increases because we are introducing more variables to the graph. For example, at 2-dimensional space, the weight of an edge between node $i(x_1, y_1)$ and $j(x_2, y_2)$ is $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ as x_1, x_2, y_1, y_2 are all random number between 0 and 1. And in 3-dimensional space, the weight of an edge between $i(x_1, y_1, z_1)$ and $j(x_2, y_2, z_2)$ is $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$ as $x_1, x_2, y_1, y_2, z_1, z_2$ are all random number between 0 and 1. The weight(which is also the distance) is expected to be increased as the dimension increases. We will verify this assumption in the next section.

- **List of Max edge vs. number of nodes(*n*)**

The list below shows the max edge length (weight) of the MST with different number of dimensions and different number of nodes. We can clearly see that as the dimension increases, the max weight increases as well. This proves our guess above is true.

Number of nodes	Max edge(0-d)	Max edge(2-d)	Max edge(3-d)	Max edge(4-d)
16	0.151531856385	0.354060785881	0.465580383974	0.583350290791
32	0.075827638832	0.281853899564	0.447771256774	0.496376566904
64	0.09944168349	0.220281129642	0.323478441319	0.479516627529

128	0.040499094282	0.154815797294	0.238905285061	0.431879292132
256	0.023495122701	0.117184054658	0.230284420679	0.342178561309
512	0.012509345081	0.073100605189	0.197602525428	0.28993610536
1024	0.007066624186	0.053325971829	0.127659281185	0.230883425309

- **Run Time and Cache Size**

By running the code, we can see the running time increases exponentially as *n* increases. This could be due to the size of the graph increases and the number of edges that need to be computed increases quadratically. At the same time, as the number of dimensions increases, the running time increases as well. This can be explained by the number of computation increases as the dimension increases.

The Cache size influence the running time. With a small memory space, it takes longer to run the same *n*. With *n* greater than 8192, my computer would stop working (system restarted) because we ran out of memory space with such a large *n*. Therefore it is important to have an algorithm that takes less memory space.

To improve our original method by reducing the memory space it takes, we decide to store less edges by throwing away the edges with extremely large weights since they will not show up in the MST. From the previous section, we find out that as *n* increases, the maximum edge in MST will always decrease. This is a useful fact since we can set an upper bound of edge weight to reduce edges, memory usage, and runtime. We define *k(n)* to be the maximum weight in MST as a function of *n*. *k(n)* is the largest for 4-dimension. For conservative reason, we reserve weights less than 0.2 in the graph for *n* larger than 4096.

- **Random Number Generator**

The “random.uniform” command generate random number by the actual time on the computer. So the random numbers are actually not “random”. When we have small n, the “random” weights are not random because the weights are picked in a very short period of time. The program runs so fast that the numbers being generated as “random numbers” are the same with each other. When we are working with a large n, for example, *n*=8192, which takes my computer 5 minutes to find the MST, the weights are more random because the random numbers computer generate depend on different period of time.