Programming Assignment 1

Bin Yan

# Experiment results

I have simulated the average size of minimum spanning tree of four types of complete undirected graphs for number of vertices . I have run 40 trials for each graph when . The results for dimension =1, 2, 3 and 4 are listed in Table 1. The results are also plotted in Figure 1, Figure 2(a), Figure 3(a) and Figure 4(a).

Table Average size of minimum spanning tree for four types of graphs

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| numpoints | d = 1 | d=2 | d=3 | d=4 | numtrials |
| 16 | 1.193 | 2.64 | 4.38 | 6.15 | 40 |
| 32 | 1.156 | 3.83 | 7.09 | 10.22 | 40 |
| 64 | 1.192 | 5.40 | 11.21 | 17.06 | 40 |
| 128 | 1.169 | 7.67 | 17.57 | 28.57 | 40 |
| 256 | 1.191 | 10.69 | 27.57 | 47.12 | 40 |
| 512 | 1.195 | 15.05 | 43.36 | 78.19 | 40 |
| 1024 | 1.202 | 21.12 | 68.03 | 129.96 | 40 |
| 2048 | 1.196 | 29.68 | 107.16 | 216.42 | 20 |
| 4096 | 1.202 | 41.78 | 169.32 | 361.10 | 20 |
| 8192 | 1.204 | 59.01 | 267.31 | 602.56 | 20 |
| 16384 | 1.202 | 83.20 | 422.26 | 1008.71 | 20 |
| 32768 | 1.206 | 117.39 | 669.25 | 1688.58 | 5 |
| 65536 | 1.199 | 165.96 | 1058.60 | 2827.48 | 5 |
| 131072 | 1.202 | 234.62 | 1677.85 | 4739.94 | 5 |
| 262144 | 1.203 | 331.60 | 2658.32 | 7952.42 | 5 |

d=1

Figure MST size vs. number of vertices, dimension = 1

d=2

Figure (a) MST size vs. number of vertices n (b) MST size vs. number of vertices , dimension = 2

d=3

Figure (a) MST size vs. number of vertices n (b) MST size vs. number of vertices , dimension = 3

d=4

Figure (a) MST size vs. number of vertices n (b) MST size vs. number of vertices , dimension = 4

# Minimum spanning tree size

As shown in in Figure 1, Figure 2(b), Figure 3(b) and Figure 4(b),

where is the number of vertices in the graph, and the is the dimension of the graph type.

Therefore, my guess of is,

where and are constants.

# Discussion

## Prim vs. Kruskal

I use Prim’s algorithm. In particular, I use adjacency matrix to represent a graph in Prim’s algorithm.

For Prim’s algorithm, if using adjacency matrix searching, the running time is . If using binary heap and adjacency list, the running time is , where is the number of the vertices and is the number of edges. For Kruskal’s algorithm, the running time is . Since we are dealing with complete graph, . Therefore, I choose Prim’s algorithm and adjacency matrix to represent the graph. In this particular case, heap will not help. I simply use two nested for loops which takes .

## Memory problem when is large

When ( corresponds to ) grows larger than 30,000, memory issue becomes a challenge. If we simply use an adjacency matrix, we need sizeof(float) GB. My laptop memory is 4 GB. It still can handle as I tested, but already takes very long. To solve this problem, I define three types of graphs in C++, AdjacencyMatrixGraph, HashGraph and EuclideanGraph.

AdjacencyMatrixGraph is used for one dimensional random graph when . Instead of storing weights, we only allocate sizeof(float) memory to store edge weights. This is because edge weight of is equal to edge weight of . In addition, there is no edge . When getting and setting the edge weights, we need to define our own function to deal with index (see the code in AdjacencyMatrixGraph class/struct as follows).

AdjacencyMatrixGraph (**int** \_num\_vertices) : Graph (\_num\_vertices){

*// allocate memory in constructor*  
 edge\_weights = (**float**\*)malloc(**sizeof**(**float**) \* (num\_vertices - 1) \* num\_vertices / 2);  
 **...**

}

**float** getEdgeWeights (**int** i, **int** j) {  
 **if** (i > j) {  
 **return** edge\_weights[(i-1) \* i / 2 + j];  
 } **else if** (j > i) {  
 **return** edge\_weights[(j-1) \* j / 2 + i];  
 } **else** {  
 **return** std::numeric\_limits<**float**>::infinity();  
 }  
  
}  
  
**void** setEdgeWeights (**long long int** i, **long long int** j, **float** weight) {  
 **if** (i > j) {  
 edge\_weights[(i-1) \* i / 2 + j] = weight;  
 } **else if** (j > i) {  
 edge\_weights[(j-1) \* j / 2 + i] = weight;  
 }  
}

For one dimensional graph when , we use hash function to generate random edge weights. Therefore, we do not need any memory to store edge weights. For each trial, we assign a random seed to the hash function. Within each trial, we use the hash function to get the edge weights. (See class HashGraph in the main.cpp.) For other dimensional graphs, we constructed a EuclideanGraph class/struct, which stores the vertices coordinates generated randomly. Every time when we need to get the weight of an edge, we read these two vertices from memory and calculate the edge weight. Therefore, we only need memory sizeof(float) . As a result, the number of vertices limited by memory for our code is

For this assignment, we tested up to 262144. It takes less than an hour to run 5 trials.

## Growth rate of

When , is constant. is the sum of first smallest edge weights among the values randomly generated between . The average value of the first elements might be order of . Then the MST tree size is , where is a constant.

grows at the rate of .