# Randomized Algorithms
# Lab Assignment

*Ali Arabyarmohammadi*
*January 2021*

**An Exploratory Assignment on Minimum Spanning Trees**
*Mitzenmatcher-Upfal book - Assignment 10.6.*

## Abstract

The goal is to define a function $f(n)$ to calculate the total weight of the MST[1] of a complete graph with $\binom{n}{2}$ edges which weigh of each edge has been chosen uniformly at random on [0,1]. I used **C** to implement Kruskal's algorithm to find a minimum spanning tree for a weighted graph. To implement the random graph generator, I used C's built-in *rand()* function and seeded it with a value of my machine's local time.

## Choice of MST Algorithm

There are two choices for choosing an algorithm for the graphs that are weighted, connected, and undirected: **Kruskal's algorithm** and **Prim's algorithm**.

I decided to choose Kruskal's algorithm because I realize that its runtime performance would be much faster than Prim's algorithm for sparse graphs. The barrier will be sorting all the edges with Kruskal's algorithm, but I would simply take the shortest of the remaining ones after that. I figured out that the cost of sorting edges that should be done in Kruskal's algorithm would be much lower than the repeated searching we would have to do with Prim's algorithm. If I were to use Prim's algorithm, I would have to look at all vertices linked to the MST, and it would be very time consuming to loop repeatedly through all the edges. Therefore, I finally ended up using Kruskal's algorithm.

## Implementation

Constructing a random graph generator (RGG) was the first task. (File: graph.c).
To produce this random value, I seeded C's random number generator with the current microsecond time and used C's built-in $rand()$ function. Of course, $rand()$ produces a random integer between $0$ and $RAND\_MAX$, and so I converted them to floats and normalized so the values would fall between 0 and 1. (Function: generateGraphAtRandom)

[1] Minimun Spanning Tree

Then I needed some way of representing an edge. I defined a new *edge struct* that contained two integers representing the vertices of the edge (u and v) and a float representing the weight. Considering this, finishing the RGG was simple: just iterate over all possible combinations of edges and create a *struct* with the suitable endpoints and random weight for each one.

**Kruskal algorithm pseudocode** *(located in MST.c file)*

```
KRUSKAL(G):
A = ∅
For each vertex v ∈ G.V:
    MAKE-SET(v)
For each edge (u, v) ∈ G.E ordered by increasing order by weight(u, v):
    if FIND-SET(u) ≠ FIND-SET(v):
    A = A ∪ {(u, v)}
    UNION(u, v)
return A
```

## Optimization and Memory

I simplified the graph by dropping heavy edges unlikely to be included by Kruskal's algorithm in the MST in order to handle a large value of $n$. By running the program for $n$ equal to powers 2 through 8192 and logging the heaviest edge included in the MST each time, I obtained the concept of what makes a heavy edge. Based on these results, I chose a threshold weight, which I then used to determine whether an edge should be included in the array sorted during each run by Kruskal's algorithm. This sorting procedure is the slowest part of Kruskal's algorithm, and decreasing the number of edges is needed to sort consequently accelerates Kruskal's execution time. Throwing the edges away in this way will never lead to a circumstance where the program returns the wrong tree because we are truncating the array that Kruskal's pulls edges from at a point beyond that needed by the algorithm to create an MST.
This means we are throwing out the edges we know will never be used. In this way, the optimized algorithm must give the same output as a non-optimized one.

Before this hard cutoff optimization for edge weight, my laptop would run out of memory to store the edge lists for generated graphs. The optimization allows the code to store fewer edge weights. By dynamically resizing the edge list array as necessary when building up a graph, we can use significantly less memory. Therefore, the optimization benefits not only in running time but also space complexity.

## Weight Results

The following data shows the average tree size over five trials for several n performing Kruskal's algorithms. It appears that the average weight of the MST is around 1.2 and the best fit function $f(n) \approx 1.2$.

*Table 1: Average Weight of MST vs. n Over 5 Trials*

| n | trials | Average MST Weight |
|---|---|---|
| **16** | 5 | 1.227888 |
| **32** | 5 | 1.156426 |
| **64** | 5 | 1.173687 |
| **128** | 5 | 1.149626 |
| **256** | 5 | 1.214557 |
| **512** | 5 | 1.183582 |
| **1024** | 5 | 1.212469 |
| **2048** | 5 | 1.168531 |
| **4096** | 5 | 1.145232 |
| **8192** | 5 | 1.177607 |
| **16384** | 5 | 1.212063 |
| **32768** | 5 | 1.179531 |
| **65536** | 5 | 1.203361 |

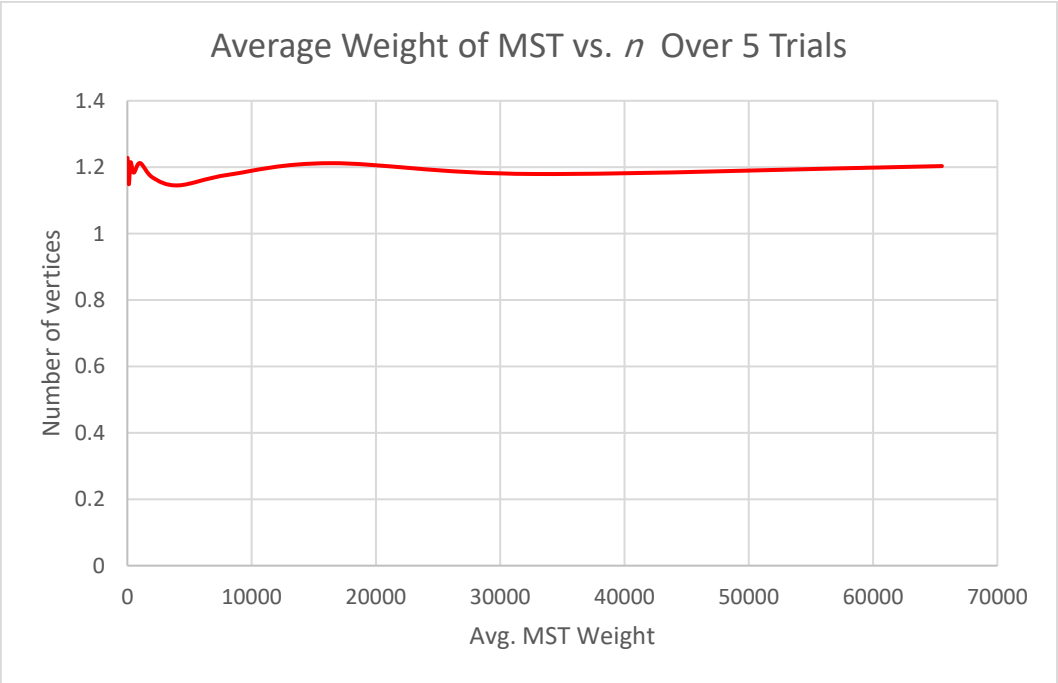


*Figure 1: Average Weight of MST vs. n Over 5 Trials*

As we see in the following table and the chart, the maximum edge weight of MST decreases significantly by increasing the value $n$.

*Table 2: Maximum Edge Weight of MST*

| n | Maximum Edge Weight |
|---|---|
| **16** | 0.26496 |
| **32** | 0.18210 |
| **64** | 0.08915 |
| **128** | 0.05133 |
| **256** | 0.02887 |
| **512** | 0.01245 |
| **1024** | 0.00925 |
| **2048** | 0.00415 |
| **4096** | 0.00345 |
| **8192** | 0.00137 |
| **16384** | 0.00067 |
| **32768** | 0.00034 |
| **65536** | 0.00018 |

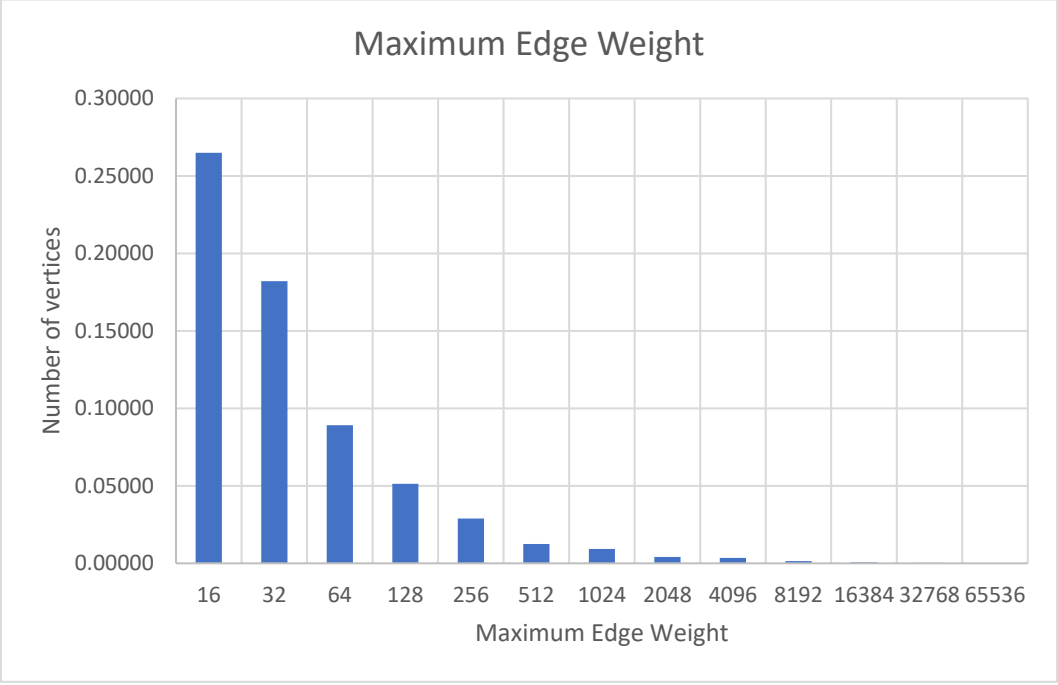


*Figure 2: Maximum Edge Weight of MST*

## Execution Time Result

I measured each execution time over five trials in milliseconds for different numbers of $n$ and detailed them in the table below. In this measurement, I did not include the time spent performing graph generation. I just consider the execution time of Kruskal's algorithm. I used a hard cutoff optimization for edge weight in the implementation, and its effect can be seen in the $n \geq 10000$. For values of $n$ in the range $[2, 10000)$, the running time is of the form $O(n\ log\ n)$. Besides, for values of $n$ in the range $[10000; \infty)$, the execution time is also of the form $O(n\ log\ n)$. The change that occurs at $n = 10000$ is a direct consequence of the cutoff optimization for edge weight I implemented. The optimization starts at values of $n \geq 10000$. During the execution time, I did not notice the cache size of my computers affecting the runtime.

*Table 3: Execution time over five trials in milliseconds*

| n | trials | Execution time (ms) |
|---|---|---|
| **16** | 5 | 0.00 |
| **32** | 5 | 0.20 |
| **64** | 5 | 1.00 |
| **128** | 5 | 3.00 |
| **256** | 5 | 14.40 |
| **512** | 5 | 77.00 |
| **1024** | 5 | 268.20 |
| **2048** | 5 | 1107.40 |
| **4096** | 5 | 4567.30 |
| **8192** | 5 | 5376.20 |
| **16384** | 5 | 29.40 |
| **32768** | 5 | 124.50 |
| **65536** | 5 | 519.7 |

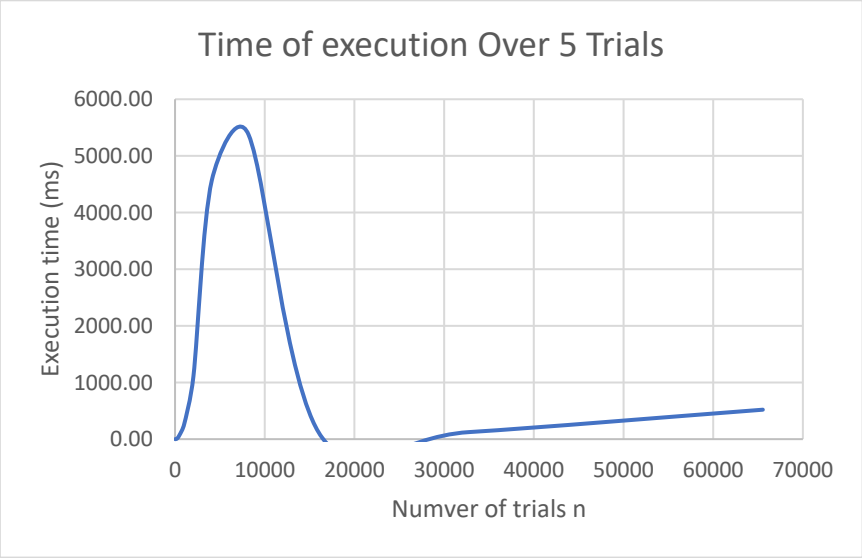


*Figure 3: Execution time of Koruskal's algorithm over five trials (ms)*

## Discussion

I mentioned some issues and discussed the implementation of the algorithm before. Now I am going to discuss how MST behaves in random graphs.

One observation I made was in regards to the RNG. At first, I seeded the $rand()$ function at the beginning of creating a complete graph. I decided to seed more often halfway through coding. Oddly, this resulted in very distinct outcomes. I saw that seeding more often did not result in convergence to 1.2. During coding using a single seeding, I frequently tested for various values and somewhat consistently got an average weight of 1.2. As I seeded more often, I saw more differences in the data, which made me uncertain whether the results were reliable. Therefore, I returned to seeding once, and the final function results became promising again. (Now is located before calling *generateGraphAtRandom()* function inside the *main()* function in the file MST.c)

Before revealing the results, I thought that the average edge length in the MST must decrease proportionally by increasing $n$. I was surprised by the results from the data. I saw that the average weight numbers converge at 1.2. This is because we are choosing edge lengths randomly on the interval from 0 to 1, and so the range of the smallest $n - 1$ edges decreases as $n$ increases. The MST does not necessarily include the smallest $n - 1$ edges, but the facts remain true that the smallest proportion of edges decreases.
I gathered a bit of data on the algorithm's runtime during this process. I did not have sufficient information to carry out a comprehensive analysis, but one of the clear findings is that the running time increases dramatically as n increases. (When we have more edges). Of course, the run time decreased significantly after implementing the cutoff optimization. An analysis of the time spent revealed that most of the time was spent creating the complete graph and not performing the MST.