RA Project - Exploratory Assignment on Minimum Spanning Trees

Dinis Moreira

16th January 2021

1 Introduction

This is a resolution of the assignment from chapter 10.6 of the Mitzenmatcher-Upfal book "Probability and Computing - Randomized Algorithms and Probabilistic Analysis"

The focus of this project is about the Minimum Spanning Tree of a complete, undirected graph with n nodes and $\binom{n}{2}$ edges, where each edge has a random weight, a real number chosen uniformly between [0,1]. The main goal is to estimate how the expected weight of the minimum spanning tree grows as a function of n for such graphs.

This project was implemented using Python v3.8.5

2 Generating appropriate random graphs

In order to generate complete graphs two classes of objects where created, predictably Nodes and Edges.

Each Node is comprised of an integer id, and a list of connected edges

```
class Node:

    def __init__(self, id):
        self.id = id
        self.connectedEdges = []
```

Figure 1: Node Class

Each Edge is comprised of two nodes, an "originalNode" and an "otherNode", and a float weight.

With these Node and Edge classes we can now generate a Graph, constituted of a list of Nodes and a list of Edges.

Generating the list of nodes is quite straightforward, we just generate each node with and incremental id until we reach the desired number of nodes. The

```python
class Edge:

    def __init__(self, originalNode, otherNode, weight):
        self.originalNode = originalNode
        self.otherNode = otherNode
        self.weight = weight
```

Figure 2: Edge Class

```python
class Graph:

    def __init__(self, nodesNum, seed, excludingEdges):
        self.nodes = []
        self.edges = []
```

Figure 3: Graph Class

connected edges of each node will be appended next while we generate each edge.

For the edges, as we are generating a complete graph where each node has an undirected edge connecting to each other node, we will generate $n(n-1)/2$ edges. As we do not want edges leading to the same node and we do not want reversed edges, meaning that if we already have generated an edge $e(i, j)$ we do not want to generate the same reversed edge $e(j, i)$. We basically iterate through n^2 combinations of nodes and we only generate the edges where the index of the first node is lower than the one on the second node the edge is connecting to. Attributing each edge a randomly generated weight between $[0, 1]$

Meanwhile we are also appending each edge to the respective "connectedEdges" list of the correct original node.

Optionally, as we will discuss in section 5 of this report, edges that were generated with a weight higher than a predetermined value may not be added to the graph in the first place, as they were deemed redundant.

3 Minimum Spanning Tree algorithm

For finding the minimum spanning tree of the complete graphs, Prim's algorithm was chosen as our best option. Because complete graphs are very dense when it comes to edges, this algorithm is more efficient than Kruskal's algorithm, that usually explores every edge in the graph log(n) times.

3.1 Time complexity of the Algorithm

In this implementation, Prim's algorithm time complexity is $O(n^2)$. For every time a new node is reached, that node's connected edges are added to the "availableEdges" list so that they can be selected next, and they are also removed from the original graph along with the node itself, as there is no use for

them in the next iteration, that node has already been reached. This process is repeated until all the nodes have been reached, so the time complexity is $O(n + n - 1 + ... + 2 + 1) = O(n^2)$

4 Seeded random number generator

In order to achieve a repeatable graph with weighted edges generated at random, the *random.py* library was used. It implements pseudo-random number generators for various distributions, in this case we are interested in an uniform distribution of real numbers between $[0, 1]$. Python uses the Mersenne Twister, which is by far the most widely used general-purpose pseudorandom number generator, producing 53-bit precision floats, and it has a large period of $2^{19937} - 1$. The Mersenne Twister is one of the most extensively tested random number generators in existence, giving us confidence on the actual randomness and distribution of the numbers generated.

Throughout testing of this algorithm and the analysis of the generated random numbers no evident anomalies were found, the distribution seems to be correct and similar seeds do not produce similar results.

5 Excluding heavy edges

5.1 Why throwing away heavy edges does not affect the final result

Because we are working with a complete graph, where every node has an edge connecting to every other node, any transition between 2 nodes is possible in order to achieve the MST. Because every node has n edges this means that, according to the uniform distribution of the random weight of the edges, every node should be reachable with an edge weighing less than $k(n)$

We can take advantage of this repeated random sampling with the Monte Carlo method and deduce that every edge weighing more than that will never be used for the MST, so the final result of the MST is not affected.

5.2 Finding $k(n)$

The approach taken in order to exclude heavy unnecessary edges was to find a function $k(n)$ where any edge with a weight above $k(n)$ would be extremely unlikely to be part of the minimum spanning tree. In order to find $k(n)$ several runs of the algorithm were run for some lower values of n, namely $n = 16, 32, 64, 128, 254, 512, 1024$, for each value of n the program was run at least 10 times and for each result the weight of the largest edge in the MST was recorded. We can see the results in the following table, and on the graph of figure 4, the weight of the heaviest edge recorded across multiple runs for each value of n:

n	Biggest registered weight of an edge used in the MST
16	0,376248343342903
32	0,207476464853699
64	0,113125607847262
128	0,0695882513714249
256	0,0281809153992951
512	0,0131664799275629
1024	0,0095604419769727

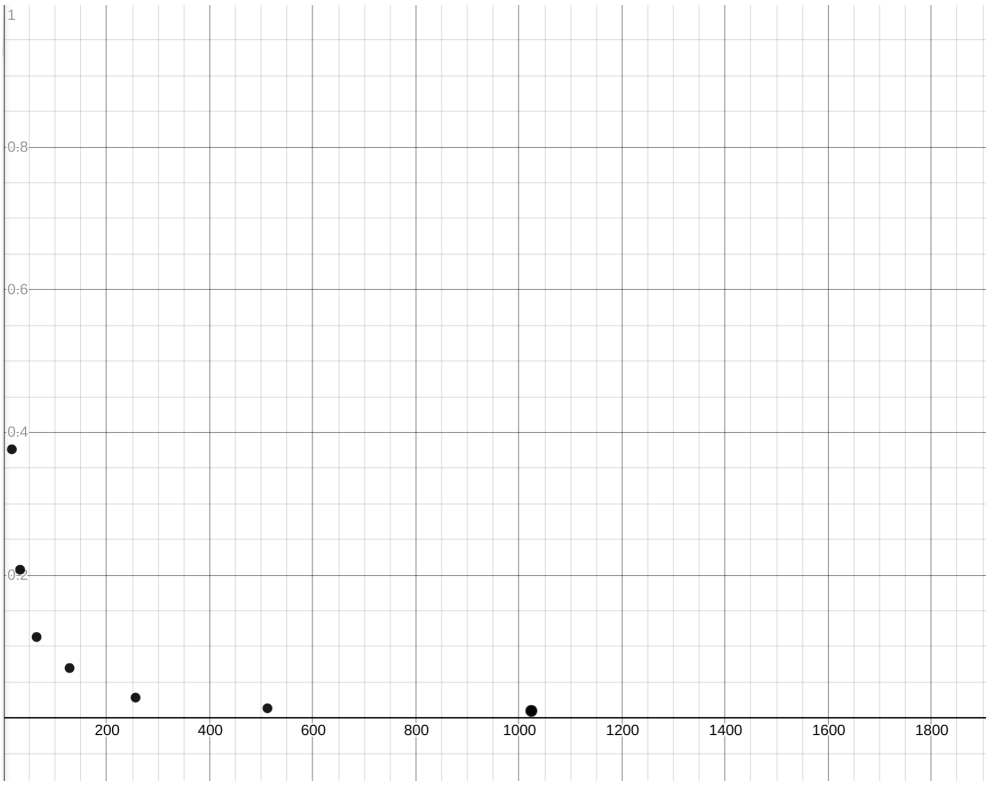

Figure 4: Graph representing the biggest registered weight of edges used in the MST across multiple experiments for each value of n, tested without removal of any edges. The vertical axis represents the weight of the edges and the horizontal axis the value of n

From these values we can deduce that a good approximation for $k(n)$ would be a reciprocal type function equal to $1/(k \cdot x)$

There is no "correct" value of k, a good estimate for the function $k(n)$ which ensures a very high probability of achieving the correct MST is $k(n) = 1/(0,075 \cdot x)$

For values of k slightly bigger we get get smaller values for $k(n)$ and we start to increase the risk of excluding edges that would otherwise be used in the MST. For values of k slightly smaller we get bigger values for $k(n)$ and the number of edges that is not excluded decreases, meaning that it takes more time to iterate through more edges that could be used in the MST.

$k = 0,075$ was used in all the recorded results, including the initial 35 tests concluded, with $n = 16, 32, 64, 128, 254, 512, 1024$, 5 times each, where this approach was tested alongside another repetition of the algorithm with the complete graph without excluded edges, generated with the same seed. 100% of these tests achieved the exact same MST, and much faster, as we can observe in the following table and in figure 5:

n	$k(n)$	Biggest registered weight of an edge used in the MST	Average weight of the biggest edge of the MST	Standard Deviation of the weight of the biggest edge of the MST	Average Total Weight of the MST	Standard Deviation of the Total Weight of the MST	Average Time with all edges	Average Time excluding edges with weight above $k(n)$
16	0,83333	0,24273	0,19406	0,03528	1,13960	0,27091	0,00059	0,00050
32	0,41667	0,18579	0,13371	0,04623	1,25055	0,17379	0,00375	0,00156
64	0,20833	0,11170	0,07375	0,02430	1,12588	0,14725	0,02754	0,00477
128	0,10417	0,05490	0,04235	0,00773	1,22384	0,06618	0,15336	0,01878
256	0,05208	0,03160	0,02404	0,00608	1,20845	0,05284	1,33953	0,06370
512	0,02604	0,01554	0,01304	0,00149	1,23724	0,03604	16,7641	0,21556
1024	0,01302	0,00920	0,00719	0,00124	1,21093	0,02975	154,452	0,90940
2048	0,00651	0,00409	0,00366	0,00032	1,19655	0,02353	-	3,52057
4096	0,00326	0,00253	0,00222	0,00023	1,19989	0,01984	-	20,7812
8192	0,00163	0,00153	0,00135	0,00012	1,20504	0,00726	-	101,177
16384	0,00081	0,00076	0,00060	0,00008	1,19047	0,00960	-	493,293

As we can see, despite the low amount of experiments, 5 for each value of n, $k(n)$ consistently stays above the biggest registered weight of an edge used in the MST for every value of n, with a margin of about 7% for even the larges values of n, and of at least 20% when compared to the Average weight of the biggest edge of the MST. This gives us great confidence as to whether the generated MST is actually the minimal one, for the experiments that could not be run without removing some edges.

We can also see that both the standard deviation of the weight of the biggest edge of the MST and of the total weight of the MST tend to decrease as n gets bigger.

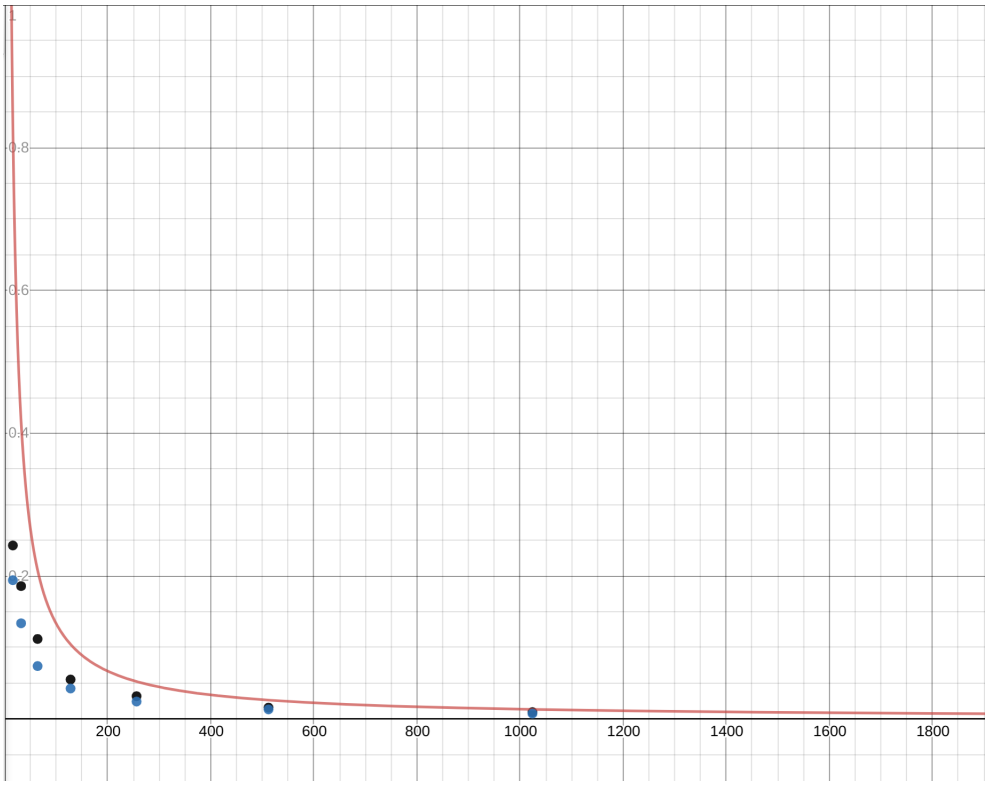

Figure 5: Graph representing $k(n) = 1/(0,075 \cdot x)$ on red, alongside the biggest registered weight of edges used in the MST across multiple experiments for each value of n on the black dots, and the average value for the edge with the biggest weight used in the MST, on the blue dots. The vertical axis represents the weight of the edges and the horizontal axis the value of n

6 Expected weight of the minimum spanning tree

We can conclude that the expected weight of the MST does not change with different values of n, always revolving around $1, 2$ in total. This happens because when we introduce more nodes in the graph, we are also introducing more edges from where these nodes can be reached, and therefore the expected minimum weighted edge for reaching each of these nodes decreases, balancing the total weight of the MST across the range of n.