# DA2440 - Advanced Algorithms
# Sublinear Algorithms: Minimum Spanning Forest

Submission: 5079443
Attempted Grade: B

Pedro Campos
Kirk Easterson
Filip Hörnsten
Manuel Rickli

December 6, 2019

## 1 The Algorithm

The algorithm we chose to implement is a modified version of the Minimum Spanning Tree (MST) algorithm described in the Czumaj paper [2]. We will begin by discussing the implementation of MST. Start with the case where the maximum weight $W = 2$. We start by finding the weight of the minimum spanning trees of the subgraph with only edges equal to 1. Let this graph $G^{(1)} = (V, E^{(1)}$ where $E^{(1)}$ is the set of all edges in $G$ with weight up to 1. One should expect $G^{(1)}$ to be composed of many connected components. Let this number of connected components be $c^{(1)}$. Since there are $c^{(1)}$ number of connected components in $G^{(1)}$, there must be $(c^{(1)} - 1)$ edges of weight 2 and $(n - 1 - (c^{(1)} - 1))$ edges of weight 1 in $G$. This implies that the weight of the MST in $G$ is:

$$1 \cdot (n - 1 - (c^{(1)} - 1)) + 2 \cdot (c^{(1)} - 1) = n - 2 + c^{(1)} = n - W + c^{(1)}$$

Equation 1 demonstrates generalization of this equation [2]. Algorithm 1 implements equation 1 to approximate the weight of the minimum spanning tree [2].

$$MST = n - W + \sum_{i=1}^{W-1} c^{(i)} \tag{1}$$

**Algorithm 1:** MST Approximation

**Input:** $G, \varepsilon$

**for** $i \leftarrow 1$ **to** $(W - 1)$ **do**
  $\hat{c}^{(i)}$ = approximator for $c^{(i)}$
**end**

**Output:** $\widetilde{MST} = n - W + \sum_{i=1}^{W-1} \hat{c}^{(i)}$

Next an approximator for $c^{(i)}$ must be implemented. 1. Algorithm 2 can be used [2][1].

The next step is to generalize equation 1 to Minimum Spanning Forest (MSF). To accomplish this, one can add edges of weight $(W + 1)$ to connect all components in $c^{(W)}$. Then one must calculate the weight of the MST of this modified graph. The added weight is then subtracted from this MST weight. The value returned will be equal to the weight of the minimum spanning forest. This is however the ideal case - to achieve sub-linear performance the $\hat{c}^{(i)}$ approximation must be used. This is simple to implement, for MSF is very similar to the last step of MST. The last step of MST assumes that the edges connecting all components in $c^{(W-1)}$ are of weight W, so it multiplies $W \cdot (c^{(W-1)})$. Thus, if we run MST with maximum weight $(W + 1)$ then we will get the weight of the modified spanning tree. The weight that must be subtracted can be determined with $(c^{(W)} - 1) \cdot (W + 1)$.

**Algorithm 2:** Approximate Number of Connected Components

**Input:** $G, s$

uniformly choose $s = \left(\frac{1}{\varepsilon^2}\right)$ vertices $u_1, u_2, ..., u_s$ at random

**for** $i \leftarrow 1$ **to** $(s)$ **do**

choose $X$ such that $Pr[X \geq k] = \frac{1}{k}$

run BFS from $u_i$ until one of two cases:

1. the entire connected component has been explored
2. $X$ vertices have been explored

**if** *BFS stopped in case (1)* **then**

$b_i = 1$

**else**

$b_i = 0$

**end**

**end**

**Output:** $\hat{c} = \frac{n}{s} \sum_{i=1}^{s} b_i$

Thus, the weight of a MSF is as shown in equation 2. Algorithm 3 demonstrates an implementation of this formula.

$$n - (W+1) + \sum_{i=1}^{W-1+1} c^{(i)} - (c^{(W)} - 1) \cdot (W+1)$$

$$n - W - 1 + \sum_{i=1}^{W-1} c^{(i)} + c^{(W)} - (c^{(W)} \cdot W) - c^{(W)} + W + 1$$

$$MSF = n - (c^{(W)} \cdot W) + \sum_{i=1}^{W-1} c^{(i)} \tag{2}$$

**Algorithm 3:** MSF Approximation

**Input:** $G, \varepsilon$

**for** $i \leftarrow 1$ **to** $W$ **do**

$\hat{c}^{(i)} =$ approximator for $c^{(i)}$

**end**

**Output:** $\widetilde{MSF} = n - (\hat{c}^{(W)} \cdot W) + \sum_{i=1}^{W-1} \hat{c}^{(i)}$

Algorithm 3 will also work for an MST, for the value of $c^{(W)}$ would be 1. In this case it will be identical to equation 1. The value of $\hat{c}^{(W)}$ will not always be equal to 1, but it will return a similar result to algorithm 1. It is not as precise as MST but the generalization works fo rboth cases.

# 2 The Development Process

Python was our choice of language for this project. We began with the example file from Canvas. The only part used from this example was the method for communicating with Kattis. The next step was to read and implement the algorithms in the Czumaj paper [2]. We had implemented the basic *ApproxMSTWeight(G,ε)* and *ApproxConnectedComps(G,s)* algorithms, but we had to write our own *BFS* method. A basic *BFS* implementation was used but we modified the return value to return the appropriate $b_i$ for the BFS search of that node. The Czumaj paper does not suggest a value for $s$ in *ApproxConnectedComps(G,s)*, so we used the value suggested in the Chazelle paper.[1].

At this point, we should have been able to complete the sublinear MST problem group. Our implementation was inconsistently passing about 4 out of the 6 test cases. Most of the errors were

"time-limit exceeded", not "wrong answer". This led us to believe that our implementation of sub-trees in the *BFS* was inefficient. We also thought that our program was querying Kattis too often. To combat this problem we redesigned how our program handled node processing. We started by modifying the `getNeighbors(node)` function. A list of tuples is inefficient when looking for neighbors on an edge of a specific length. This was changed to a hashmap where the value is the weight of the edge and the value is an array of node ID's. In Python terms, it is a dict that maps an int to a list. This improved time performance when searching through sub-trees. The program receives a list of the appropriate neighbor nodes without having to sort through which neighbors are valid or not. To avoid querying Kattis too often, each hashmap is stored in a global neighbor hashmap where the key is the node id and the value is the neighbor hashmap. In Python terms, it is a dict that maps an int to a neighbor dict.

These optimizations allowed us to pass all test cases for MST. The next step was to generalize to Minimum Spanning Forest (MSF). This appeared to be similar to the last few steps in the MST algorithm. If $\hat{c}^{(W-1)}$ is not 1 then we can run MST on the forest but with maximum weight $W+1$, then subtract all edges of weight $W+1$. The number of edges of $W+1$ can be approximated by multiplying $\hat{c}^{(W)}$ and $(W+1)$. Unfortunately $\hat{c}^{(W-1)}$ does not always return 1. We tried to find a threshold for $\hat{c}^{(W-1)}$ in which the MSF algorithm should run, but we discovered that the MSF algorithm can be generalized for every case. The result is within error for MST and passes all cases. So we simply modified the return statement of *ApproxMSTWeight(G,$\varepsilon$)* to compensate for this.

At this point our best Kattis submission was about 49.944, but it was not reaching 50. We kept modifying some parameters to allow for larger and smaller sample sizes but we were unable to pass test group 6, which is random tests. To pass the final test group, we decided that we must implement a deterministic algorithm for edge cases where the number of nodes in the graph is smaller than the sample size. This led us to implement Kruskal's algorithm [3]. After reading about the algorithm in papers and on websites, we studied a Python implementation from a Github repository [4] for inspiration and created our own implementation to work with our algorithm. Implementing Kruskal's algorithm allowed us to pass all of test group 6. But now we were failing test group 5, which was testing how the algorithm scales with large values of $W$. Our program was compensating correctly for this before, but we couldn't figure out why it was no longer working. We tried tuning parameters such as the sample size $s$ to scale according to $W$ but were unsuccessful in making it work while still passing all the other test groups. At this point we had achieved a score of 50 and decided that this was where we would conclude our work. We believe there to be a way to accommodate for large $W$ but we did not discover how.

# References


[1] Bernard Chazelle, Ronitt Rubinfeld, and Luca Trevisan. Approximating the minimum spanning tree weight in sublinear time. *SIAM Journal on computing*, 34(6):1370–1379, 2005.

[2] Artur Czumaj and Christian Sohler. Property testing. chapter Sublinear-time Algorithms, pages 41–64. Springer-Verlag, Berlin, Heidelberg, 2010.

[3] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[4] msAzhar. kruskal.py. https://gist.github.com/msAzhar/b51d7dc21c939c53209d5a2dda3fc8ad, 2015.