Sublinear Algorithms: Minimum Spanning Forest

DD2440 Advanced Algorithms

Kristin Johansson Evegård
evegard@kth.se

Lukas Frösslund
lukasfro@kth.se

December 2019

Kattis submission ID 1: 5049551
Kattis submission ID 2: 5079608

1 Introduction

The task of this project was to implement a sublinear time algorithm with an approximate solution to the Minimum Spanning Forest (MSF) problem. To achieve this, an adaptation of a sublinear time algorithm for the Minimum Spanning tree (MST) problem presented by Chazelle et al.[1] and further theorized and modified by Czumaj and Sohler[2] was implemented.

This adaptation was applied in different alterations with varying success in terms of performance on it's test cases on Kattis, but all variations relied on the same basic concept. We treated edges of maximum weights as the edges that would connect an MSF into an MST, and altered the algorithm in [2] by first calculating the approximated number of trees in the forest and then subtracting the product of this approximated amount with the maximum edge weight, essentially eliminating these edges from the calculation.

1.1 Minimum Spanning Tree (MST)

The main idea of the sublinear time algorithm for the MST problem is to approximate amounts of connected components in subgraphs of G, with a randomized approach. The result is an approximation with a relative error of at most ε.

The algorithm assumes an undirected, connected and weighted graph $G = (V, E)$ with a maximum vertex degree d ($d \geq 1$) and integer edge weights ranging

$\{1, 2, ..., W\}$. The approximated weight of it's MST can be calculated as:

$$\widetilde{MST} = V - W + \sum_{i=1}^{W-1} \hat{c}^{(i)}$$

$\hat{c}^{(i)}$ is calculated as the approximated amount of connected components with edge weights at most i, thus $G^{(i)} = (V, E^{(i)})$ represents the subgraph of G with edge weights at most i.

Pseudo codes for the algorithm can be found in section 4.1.

1.2 MST to MSF

For the MST problem, the assumption is that G is a connected graph, which means that $\hat{c}^{(W)} = 1$, because the connected component of edge weights at most W will be the complete graph G. For the MSF problem, a connected graph can no longer be assumed (undirected and weighted graph still applies). Taking that into consideration, we tweak the algorithm to subtract the product of the approximated number of trees $|T|$ and the maximum edge weight W.

$$\widetilde{MSF} = V - W \cdot |T| + \sum_{i=1}^{W-1} \hat{c}^{(i)}$$

Because edges of weight W bounds connected components of edge weights 1 to $W - 1$ into a tree, $|T|$ can be approximated as $\hat{c}^{(W)}$ for unconnected graphs. As stated, the value for a connected graph will be 1, hence the tweak of the algorithm won't affect the result for connected graphs.

2 Method

2.1 Work progress

The initial work consisted of studying [2], which was provided to us in the assignment. Subsequently, we also studied [1] to get a further understanding of the sublinear algorithm for MST. We studied the proofs and the provided pseudo code, which was helpful to implement our code.

We first implemented functions for the sublinear MST algorithm. The initial rendition of the program had a few issues though. Primarily, we struggled with determining the value of X, the maximum number of vertices allowed to be explored in each BFS. According to pseudo code in [2], X should be chosen such that $Pr[X \geq k] = 1/k$. In the beginning, this made little sense to us and hence X was treated as a constant throughout the whole run.

We had also made a misinterpretation of the inner workings of the algorithm,

where we had wrongly assumed that only edge weights with the exact current weight i would be considered in the BFS, and not all edge weights at most i, which would be the correct interpretation.

Later, we realized that we could set $X = 1/random(0, 1)$ because there is an equivalence between $X \geq k$ and $1/X \leq 1/k$, combined with the fact that a number R, uniformly randomly generated from [0, 1] gives $Pr[R \leq p] = p$. We also corrected our misinterpretation regarding the BFS implementation.

We then began adapting the program to work for MSF and not only for MST. This was initially a stumbling block in our process, but once we figured out how we could represent the approximated number of trees in the graph, the adaptation was quite simple.

By now we were satisfied about our progression, but still identified a few flaws in our program, chiefly in how we treated the sample size S as a fixed value. This was originally done due to convenience and lack of knowledge in how we should interpret the sample size. At this point we started to elaborate on that though, with the aim of improving performance. We had a hunch that it's dynamic value would incorporate ε, and after studying [1], we saw that a sample of size $1/\varepsilon^2$ had been used in their experiments. Thus, we emulated this in our program which positively affected performance in certain test cases, but didn't give an overall lift in performance across all test cases.

After this point, our approach became more experimental. Both in terms of the value of S, and also by experimenting with applying different ceilings to the value of X. Earlier, we had limited it to a maximum value of 100, but now we tried different values as well as trying not to put any limit. However, we are still unsure on whether limiting this value gave any effect because we saw no consistent results.

In terms of S, we started looking at the complexity of the algorithm presented in [2], and based our approach on the time bound of it. The $1/\varepsilon^2$ was drawn from [1], but because our algorithm more closely resembled the algorithm in [2], we figured that an improvement could be made. We tried several approaches, but all were based on incorporating W in the calculation. W/ε^2, W^2/ε^2 and W^3/ε^2 were all tested to varying success. We noticed that different test groups responded differently depending on the choice of S, which led to the final iteration of the program where we tried to alter the value of S not only between different tests, but also during a test. Our thinking with this was to balance the scale of exploring enough nodes to get an adequate approximation with not exploring too many due to time constraints.

2.2 Implementation of Algorithm

In the description of the task it was mentioned that the actual time required by the algorithm did not matter. Since it then was not a problem to choose a slower language, we chose to write the program in Python because it is a programming language we were already familiar with.

We used the provided template code as a base for our program. For the implementation of the Breadth First Search (BFS) we used a queue which was implemented as a list. To avoid loops in the BFS, we created a visited list to keep track of all the visited nodes. The BFS function takes a randomly sampled node as an input, together with a limit for the maximum number of explored nodes and the current maximum weight we are looking at. As described in the template code, the BFS will break and return a specific value when either the whole connected component has been explored or if we have reached the maximum limit for the number of nodes explored, i.e. the maximum number of iterations.

3 Conclusions

3.1 Results

Two Kattis submission ID's have been provided, the first one (5049551) from a stage in the work progress where the sample size S was still set to a fixed value, and the other submission (5079608) from the final iteration of the program, where we had taken an experimental approach regarding the value of S. This was the submission that reached the highest score.

The reason for providing two submissions is primarily because while our experimentation with the sample size clearly had a positive effect, it's still a bit unclear on why it had this effect. As mentioned before, we theorized around the notion of changing the value of S during a test due to find a balance between time and a good enough approximation. Our thinking was to first increase the sample size after a certain weight, for this submission $W/2$, due to more need of a thorough search once components grew larger to achieve an adequate approximation. We then lowered value of S after $W/1.5$, after noticing that the time limit was exceeded in many test cases. In the end, our theories netted a positive effect in performance.

3.2 Proof of Correctness

In this section, we are going to provide a proof of why our algorithm returns an MSF. A formal, mathematical proof on how an approximation of connected components of at most edge weights i, with edge weights ranging from 1 to $W - 1$ can give an approximation of an MST is provided in [2], hence we will

not reproduce this proof and instead focus on our modification of this algorithm and why it is correct.

To summarize the sublinear MST algorithm, it's basis is expected values for the minimum number of edges of at most weight i that connects a maximum number of components. This value for weight i is estimated to be $(c^{(i)} - 1)$. For the MST problem, the graph will always be connected. Thus, this expected value for $i = W$ will always be zero, because the amount of connected components $c^{(W)}$ will be equal to 1 for a connected graph. This is why we never consider edge weights of weight W for the MST problem.

For the MSF problem, the situation is different. We now also consider unconnected graphs, for which we have more than one connected component when considering edge weights of weight W. We have previously also concluded the approximated number of trees $|T|$ to be $\hat{c}^{(W)}$. If we look at the formula to approximate the weight of a MST, the only modification we need to do is to subtract the product of the approximated number of trees $|T|$ and the maximum edge weight W, instead of just W. Thus, the formula for approximating the weight of an MSF will be

$$\widetilde{MSF} = V - W|T| + \sum_{i=1}^{W-1} \hat{c}^{(i)} \qquad (5)$$

It is also easy to see that the algorithm will look the same as before for connected graphs, since the modification in terms of the subtracted product would then result in $W \cdot |T| = W \cdot 1 = W$.

3.3 Time complexity

The time complexity for the template code we have used as a base for our MSF algorothm is $(\frac{W}{\varepsilon})^3 \cdot \log n$. However, since we have done a bit of a modification to be able to transform MST to MSF, the time complexity has changed. For MSF we run the algorithm from 1 to W, instead of 1 to $W - 1$. We must therefore add a factor W to the time complexity, which gives us

$$W \cdot (\frac{W}{\varepsilon})^3 \cdot \log n = \frac{W^4}{\varepsilon^2} \cdot \log n$$

4 Appendix

4.1 Pseudo Code

Pseudo code for approx_MSF_weight, approx_connected_comps and BFS.

4.1.1 approx_MSF_weight:

Input: Sample size S and an initial value for $\hat{c}$
Output: The estimated weight of the minimum spanning forest

for $i = 1$ to W-1 **do**
    $\hat{c}$ += approx_connected_comps(S, i)
end for
number_of_trees = approx_connected_comps(S, W)
return N - (W · number_of_trees) + $\hat{c}$

4.1.2 approx_connected_comps:

Input: Sample size S and the current maximum weight
Output: $\hat{c}$: an estimation of the number of connected components of the graph

for $i = 0$ to S **do**
    choose X according to Pr[X k] = 1/k
    choose a random node u
    b += BFS(u, X, current weight)
end for
return N/S · b

4.1.3 BFS:

Input: Randomly sampled node u, X and the current maximum weight

Output: 1 or 0

Create a queue and a visited list
Append u to the queue and the visited list

while *queue* and $iterations < X$ **do**
    Pop a node from the queue
    Get neighbours
    for $neighbours, weight$ in $neighbors$ **do**
        if $wight \leq current\ weight$ **then**
            Append the neighbour to the queue and the visited list
        else
            Continue
        end if
    end for
return

4.2 References

[1] Chazelle et, al. Approximating the minimum spanning tree weight in sublinear time
https://epubs.siam.org/doi/pdf/10.1137/S0097539702403244

[2] Czumaj, Sohler: Sublinear time algorithms
http://www.wisdom.weizmann.ac.il/ oded/PTW/sublin.pdf