Implementing Euler Tour Trees in C

Isabella Kang,[*] Emily Liu,[†] and Srijon Mukherjee[‡]

MIT Department of Electrical Engineering and Computer Science

(Dated: May 20, 2021)

I. Introduction

This is a C implementation of Euler Tour trees (ETTs), using splay trees as the underlying tree data structure. We implement the standard operations of `FindRoot`(v), `Cut`(v), `Link`(u, v), and `Connectivity`(v, w), as well as `SubtreeAggregate`(v) for `min`, `max`, `sum`, and `size`. We also implemented a few extra operations for modifying the tree: `pointUpdate(`u`, new_value)` and `subtreeUpdate(`u`, increment_amount)`.

II. User Guide

The main files implementing the data structure are `euler_tour_tree.h`, `euler_tour_tree.c`, `splay_tree.h`, and `splay_tree.c`. The `euler_tour_tree.*` files implement the ETT operations described above. The `splay_tree.*` files implement the typical splay tree operations and also support range queries for augmentations.

Other files are used in testing and benchmarking, which is described in section IV.

III. Design

The splay tree structure is a recursive tree data structure based on a `Node` struct. Each `Node` has a parent `Node` and left and right child `Node`s. It also stores a key and value, and has additional fields for augmentations. To implement the key operation `splay`(v) on some `Node` v, we first must implement the helper methods `right_rotate` and `left_rotate`, which are simply a series of pointer manipulations.

[*] ikang@mit.edu

[†] emiliu@mit.edu

[‡] srijonm@mit.edu

The `EulerTourTree` struct stores the size of the set, n, and a length-$2n$ array of `Node` pointers called `visits`. For simplicity, we assign the names of the nodes to be integers from 0 to $n-1$, and we store the pointers to the first and last visits (splay tree `Node`s) to node i in entries $2i$ and $2i+1$ of `visits`. We can then implement operations on the disjoint trees as described in Lecture 20, using the (implicitly stored) splay trees as the underlying representation.

III.1. Augmentations and Lazy Propagation

We augment our splay tree nodes with subtree values to enable `subtreeAggregate`. Augmentations include min, max, sum, and size of the subtree rooted at each node. The augmentations are stored in a struct and are recalculated from the children values using the function `augment_node`, which is called after every rotation or other structural change.

We also support setting the value of a vertex (`pointUpdate`) or incrementing the value of every vertex in a subtree (`subtreeUpdate`). For `subtreeUpdate`, we use lazy propagation for which we add a lazy value to the augmentations struct and use a function called `propagate` to push the updates down the tree. The values need to be propagated before every rotation so that every node we encounter has fully updated values.

We need to be careful of the fact that each vertex in the represented tree has multiple visits in the splay trees. To avoid double counting vertices, we only associate the value with the first visit. This in turn requires keeping track of which visit is the first one. We do this using a flag `is_start` stored in each splay tree node. This needs to be updated as we perform link and cut operations.

Currently, we have augmentations that calculate subtree minimum, maximum, sum, and size. These can be turned on or off using preprocessor logic by commenting out relevant definitions in augmentations.h. Moreover, additional augmentations can be added by updating the Augmentations struct and implementing their calculation/propagation rules in the `augment_node` and `propagate` functions.

IV. Testing and Benchmarking

IV.1. Testing

To test our code, we compared the results we get from using the ETT with brute-force results we get from using a direct access array that stores each node's parent and following parent pointers when testing for connectivity, as well as setting or removing parents when we perform a link or cut. For subtree queries (such as a `min` or `max`) we perform a DFS from our starting node to find all nodes in the subtree.

IV.2. Benchmarking

To benchmark our ETT's operations, we generate ETTs with various sizes for n, where each n is in the range $N = [100, 200, \dots, 900, 1000, 2000, \dots, 10000]$ and for each value $n_i \in N$, we perform $10n_i$ operations (randomly chosen from `Connectivity`, `Link`, `Cut`, `SubtreeAggregate`, `PointUpdate`, and `SubtreeUpdate`, where there is a $\frac{4}{9}$ probability that `Link` is chosen, and a uniform probability of choosing all other operations), and take the average runtime over all operations. The results are shown in Figure 1, plotted against the logarithm of n; we can see a linear relationship between these two quantities, suggesting that the runtime per operation scales proportionally to $\log n$, as expected from theory.

Because the generation of successful links took a potentially linear amount of time, which would make the benchmarking times slower than expected, we generated random operations and wrote only the successful operations to an .in file, then read in the operations from that file when we were benchmarking.

The ρ^2 statistic for the line of best fit is 0.976, suggesting that the linear regression predictions approximate the actual values well.

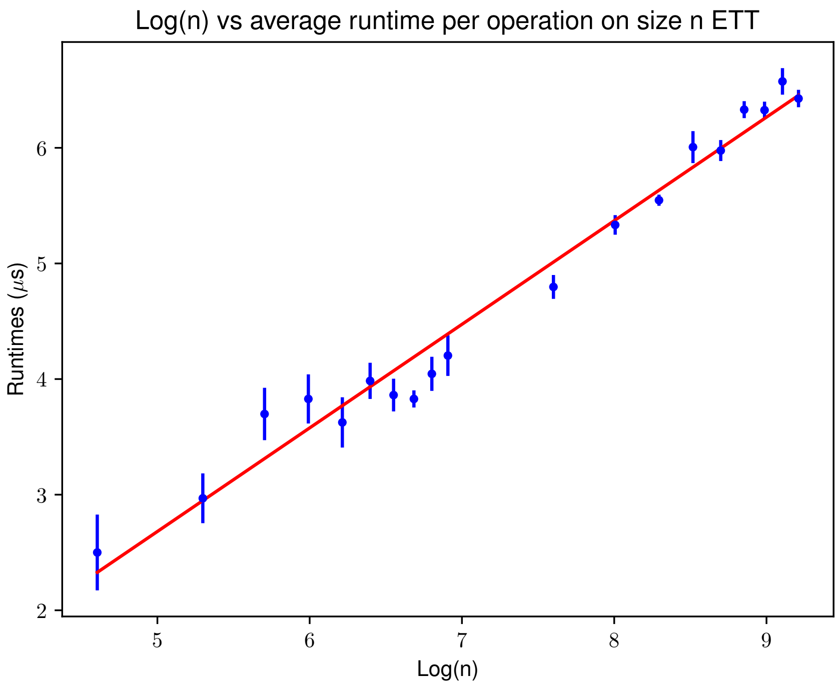

FIG. 1. A plot of the runtimes as a function of $\log(n)$, where n is the size of the ETT

V. Challenges

We spent some time trying to solve the puzzle of implementing the `evert`(v) operation. We found some issues with the accepted solution (and went to office hours to confirm this) and had difficulty solving it on our own. Professor Erik Demaine later posted an update with a fix for implementing `evert`, but for time and complexity reasons we eventually decided not to include `evert`.

The first time we tried to benchmark our ETT implementation, we found that because our implementation of `link` made a random link between two nodes, we could not create a link for a node that already had a parent. Therefore, we had to spend a potentially linear amount of time searching for a node without a parent, which would make our bench marking times inaccurate. Our solution was to generate successful operations (without timing) and write them to an external file. Then we would read from the external file and perform the listed operations while timing them. This ensured that the time was recorded for only successful operations.

VI. Conclusion and Future Work

This project was a great way to gain a deeper understanding of Euler Tour trees and splay trees. It was also interesting to add additional functionalities not present in the original set of standard operations (such as `pointUpdate` and `subtree_update`), and to speed up their times by using a lazy method of updating.

Since we weren't able to implement `evert` this time, one thing to try in the future would be to rewrite the ETT representation to support that operation by adding a representation that stores the edges used in the ETT in addition to the nodes.