`Filter_Clusters` improvements

October 15, 2018

1 Preliminaries

Define:

- $T[u]$ for some tree T and some node u to be the subtree of T rooted at u
- $\Lambda(T)$ for some tree T to be the leaf set of T
- Trees T_A and T_B on which the procedure `Filter_Clusters` will be run
- Centroid path of T_A to be π
- Set of side trees created when π is removed from T_A to be $\sigma(\pi)$

2 Bottlenecks

Note that there are two bottlenecks in `Filter_Clusters`:

- Step 3 preprocesses T_B in $O(n\ log\ n)$ time and then uses $\sum_{\tau \in \sigma(\pi)} O(|\Lambda(\tau)|log(|\Lambda(\tau)|))$ time to construct the subtrees.
- Step 6 involves making $O(n)$ add/remove operations on a BT of size $O(n)$, costing $O(n\ log\ n)$

3 Step 3

First, to avoid preprocessing T_B for RMQ queries within each recursive call, simply do it once at the beginning. We will now discuss how this original tree can be used to answer RMQ queries in recursive calls.

Note that the RMQ structure can give us a handle to the node which has the max weight between two nodes. When constructing $T_B||\Lambda(\tau)$ we add some special nodes z to $T_B|\Lambda(\tau)$ which represent a node with max weight between some two nodes. These nodes should keep handles to the node in the original tree that they identify with. In recursive calls, if asked to construct a special node where one or both of the endpoints is/are special node(s), use the handle to the node in the original tree to make this query. That is, all RMQ queries are made against the original preprocessed tree.

Constructing each $T_B|\Lambda(\tau)$ costs total $O(n)$ time, as mentioned in the paper. From each of these, constructing $T_B||\Lambda(\tau)$ will then cost $O(|\Lambda(\tau)|)$ time each since each tree contains $O(|\Lambda(\tau)|)$ edges. Also, marking spoiled nodes can be done in $O(|\Lambda(\tau)|)$ time per tree, by doing a bottom up traversal of the tree and counting the size of the leaf set.

Thus step 3 can be completed in $O(n)$ time where n is the size of the subproblem, along with a single preprocessing step at the start of the `Filter_Clusters` subroutine, costing $O(n\ log\ n)$ time, where n is the size of the leaf set.

4 Step 6

The key idea here is noticing that the weights being stored in the BT are all bounded by k. Thus we can utilise a more efficient data structure (specifically a vEB tree) to store these weights.

Recall that a vEB tree can carry out findMax operations in $O(1)$ time and insert and delete operations in $O(log\ log\ m)$ time where m is the largest possible integer being stored in the tree. In our case, m is $O(k)$.

Note that while a BT can store duplicate keys, a vEB tree is not capable of this. To fix this, maintain an array of size k where each element represents the number of nodes in the vEB tree that have the given weight. When inserting into our DS, increment the relevant counter by one, and only insert into the vEB tree if the counter was 0. Similarly, when deleting, decrement the counter, and only delete from vEB tree if counter goes to 0. Then each operation is maintained at $O(log\ log\ k)$. (Will have to initialize a new array of size k each time?? Can a vEB tree store duplicates if we store an integer at each point instead of a 0/1? Will that cost extra time? Alternatively, just sort the weights initially using counting sort, then map them to numbers in $[1, n]$)

Then Step 6 takes $O(n\ log\ log\ k)$ time. However, we can do a little better.

The idea here is that there are at most n nodes in a tree and at most k weight values, so we could remap the weight values into numbers in $[1, min(n, k)]$. This can be done by counting sorting all the weights in all the trees initially, while keeping track of which tree they came from. Construct one array per tree, each of size $min(n, k)$, which will then store all the weights of that tree. Then do a left to right pass over the sorted array, moving each weight to a new slot in the respective array. Then in each array, we can remap each weight to its index, and use these remapped values as keys for the BT. Note that these remapped values retain the ordering, and the original weights can be recovered easily by indexing into this array. (Note that this now also avoids the issue of duplicate keys in the vEB)

Since the largest value we're inserting into the vEB is now $min(n, k)$, Step 6 takes $O(n\ log\ log\ min(n, k))$ time.

5 Overall analysis

$$T(n) = O(n\ log\ log\ min(n, k)) + \sum_{\tau \in \sigma(\pi)} T(|\Lambda(\tau)|)$$

Since there are $O(log\ n)$ recursion levels,

$$T(n) = O(n\ (log\ n)\ (log\ log\ min(n, k)))$$