Maintaining Information in Fully-Dynamic Trees with Top Trees

Jaehyun Koo (koosaga)

February 22, 2021

Introduction

Introduction

Welcome to the `#project_tcs` CS Theory Study Group **Season 2**!

Group members (5 people)

1. koosaga (me)
2. Ce Jin
3. Aeren
4. siroo fanclub
5. I am worried

Iterations

1. Season 1: 2020.07.23 - 2021.01.09
2. Season 2: 2020.02.22 - 2022.05.17 (Tentative)

Introduction

Tentative schedule

- Week 1 (2/22) - Maintaining Information in Fully-Dynamic Trees with Top Trees (Host: koosaga)
- Week 2 (3/1) - Deterministic Approximation for Submodular Maximization over a Matroid in Nearly Linear Time (Host: leejseo)
- Week 3 (3/8) - Minimum Cuts in Near-Linear Time (Host: TAMREF)
- Week 4 (3/15) - Sylvester-Gallai type theorems for quadratic polynomials (Host: Aeren)
- Week 5 (3/22) - A Fine-Grained Perspective on Approximating Subset Sum and Partition (Host: Ce Jin)
- Week 6 (3/29) - Expander Decomposition and Pruning: Faster, Stronger, and Simpler. (Host: koosaga)

Introduction

Tentative schedule

- Week 7 (4/26) - Generalized Sorting with Predictions (Host: leejseo)
- Week 8 (5/3) - Three-in-a-Tree in Near Linear Time (Host: TAMREF)
- Week 9 (5/10) - Approximating APSP without Scaling: Equivalence of Approximate Min-Plus and Exact Min-Max (Host: Ce Jin)
- Week 10 (5/17) - Pseudorandom Generators for Group Products (Host: Aeren)

Preliminaries

Before we start

The original paper, *Maintaining Information in Fully-Dynamic Trees with Top Trees* is not really a good tutorial to the Top Trees. Not only outdated, I think it will end the lecture in 5 minutes.

I refered *Self-Adjusting Top Trees* by Renato F. Werneck, and his dissertation *Design and Analysis of Data Structures for Dynamic Trees* heavily.

First, I will discuss the internal structure of Top trees and the implementation.

Second, I will discuss the application of Top trees: What can it compute, and which information it can aggregate.

Before we start

Three well-known tree DS techniques in Competitive Programming:

Heavy Light Decomposition decomposes the tree into several chain of lines. It can handle path query, but not subtree query.

Euler Tour flattens the tree according to the Euler tour traversal of the tree. It can handle subtree query, but not path query.

Dynamic Tree DP is a modification of HLD that supports subtree query: For each vertex, it maintains a data structure to aggregate information from the light edges. *Probably not all with agree with the name, but at least that's how we call it in Korea.*

Before we start

Link Cut Tree (aka ST-tree) is a dynamic version of HLD, which supports edge insertion, deletion and path queries.

Euler Tour Tree (aka ET-tree) is a dynamic version of Euler Tour, which supports edge insertion, deletion and subtree queries.

Before we start

Link Cut Tree (aka ST-tree) is a dynamic version of HLD, which supports edge insertion, deletion and path queries.

Euler Tour Tree (aka ET-tree) is a dynamic version of Euler Tour, which supports edge insertion, deletion and subtree queries.

Top Tree did not started as a generalization of Dynamic Tree DP, but from the more complicated concept of *Topology Tree*.

In 2006, Werneck provided a simplified description of top trees (*rake-compress*) and provided a connection between LCT and top trees. This description is compatible with what we know as *Dynamic Tree DP*.

Top tree supports edge insertion, deletion, path, subtree queries.

Let's go!

Construction

Top Trees

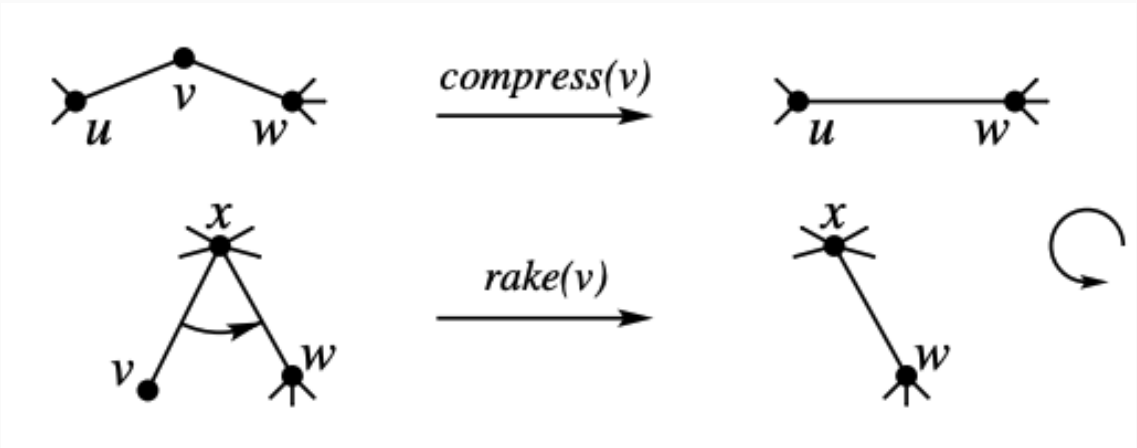
In this problem, we assume that the edges are *ordered* cyclically.

Consider the following two operation that reduces the vertices of tree.

$compress(v)$: *smooth out* the degree two vertex v.

$rake(v)$: Take an leaf vertex v and merge it to a cyclically adjacent vertex, per the edge ordering of parent of v.

$rake$ operations are always possible if $|E| \geq 2$.



Top Trees

Top Tree is a rooted binary forest of **clusters**, where each cluster is either a

1. **Compress** of two child cluster.
2. **Rake** of one child cluster to another.
3. **Base cluster** (leaf) that corresponds to an original edges in forest.

In the end, the root cluster corresponds to some path of the tree.

Top Trees

Top Tree is a rooted binary forest of **clusters**, where each cluster is either a

1. **Compress** of two child cluster.
2. **Rake** of one child cluster to another.
3. **Base cluster** (leaf) that corresponds to an original edges in forest.

In the end, the root cluster corresponds to some path of the tree.

The user can call the following three public functions:

$link(v, w)$: Add edge (v, w).

$cut(v, w)$: Remove edge (v, w).

$expose(v, w)$: Ensure that the root cluster is an edge with endpoint v, w.

We can view the top tree as a *binary search tree* of the tree.

Top Trees: Construction

We want to use the top tree as a *binary search tree* of the tree.

To do this, top trees should have small depths, preferably of $O(\log n)$.

It is possible to construct a top tree of depth $4 \log n$: There is a way to remove at least $1/6$ fraction of nodes with edge-independent rake and compress operations.

But here we are using the **self-balancing** strategy.

Top Trees: Construction

We don't care if the initial depth is large. We just **believe in splay.**

This is similar to link-cut trees: It works like HLD, but we never compute the HLD explicitly. The underlying tree just tends to work like HLD as we query it.

Thus, the data structure does not guarantee worst-case complexity, but only amortized complexity.

Top Trees: Construction

We don't care if the initial depth is large. We just **believe in splay.**

This is similar to link-cut trees: It works like HLD, but we never compute the HLD explicitly. The underlying tree just tends to work like HLD as we query it.

Thus, the data structure does not guarantee worst-case complexity, but only amortized complexity.

It is worth noticing that all top tree operations can work in worst-case $O(\log n)$ complexity. This is also true for link-cut trees.

In this presentation, we won't discuss this worst-case improvement.

We will also skip all the proofs for its complexity. Nothing is really radical, and for most of the part you can just believe in splay.

Top Trees: Construction

We don't care if the initial depth is large. We just **believe in splay.**

This is similar to link-cut trees: It works like HLD, but we never compute the HLD explicitly. The underlying tree just tends to work like HLD as we query it.

Thus, the data structure does not guarantee worst-case complexity, but only amortized complexity.

Top Trees: Construction

We don't care if the initial depth is large. We just **believe in splay.**

This is similar to link-cut trees: It works like HLD, but we never compute the HLD explicitly. The underlying tree just tends to work like HLD as we query it.

Thus, the data structure does not guarantee worst-case complexity, but only amortized complexity.

It is worth noticing that all top tree operations can work in worst-case $O(\log n)$ complexity. This is also true for link-cut trees.

In this presentation, we won't discuss this worst-case improvement.

We will also skip all the proofs for its complexity. Nothing is really radical, and for most of the part you can just believe in splay.

Top Trees: Construction

Now, let's initialize a top tree. *Of course we can simply add* $n-1$ *edges, but let's do it for an exercise*.

Take a leaf node z and make it rooted from z.

Then take any path decomposition of tree: Every path starts from the leaf and ends at another path or the root.

Path decomposition is a partition of edges.

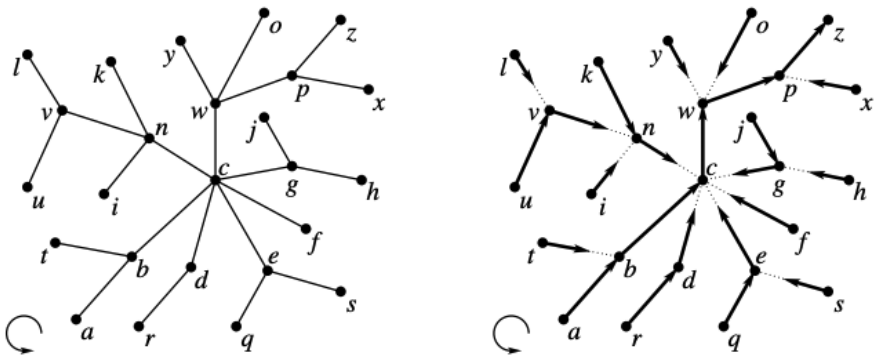


Figure 11: Example: Original tree and directed version (rooted at z and partitioned).

Top Trees: Construction

Let's recurse from the root path.

There is a list of paths that ends at the vertices of root path r. Those path x, and the path y that ends at x, and the path z that ends at y... induces a subtree.

In this sense, each edges incident to root path forms a single subtree.

Recursively merge them into clusters.

Top Trees: Construction

Now we have some clusters hanging in the path p.

Note that we care the order of the edges: In the picture, two subtrees A, B are sort of *disconnected*, and they can't be directly merged without ignoring edge (v, w).

So, **rake** the clusters left to the path, and **rake** the cluster right to the path. In the end we reach the situation like the figure.

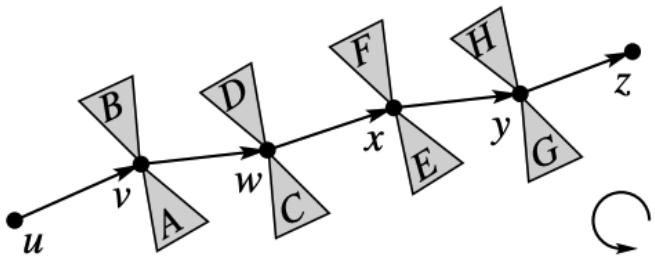


Figure 2: A unit tree rooted at z.

Top Trees: Construction

Suppose that we don't have any raked clusters. Then the subtree is simply a path: A collection of base clusters.

We can compress them in binary tree fashion to form a cluster.

We will do similarly here, but note that we have to rake.

Intuitively, we can rake everything at the beginning, but here we will rake *right before* the compression. Rake is performed only if it blocks the compression.

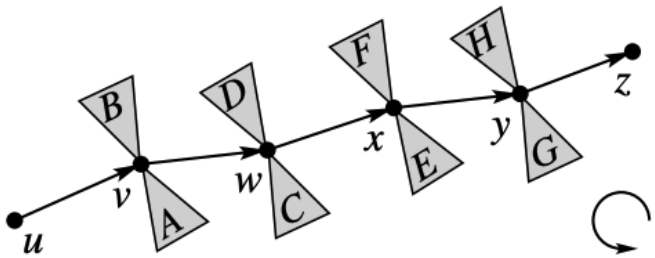


Figure 2: A unit tree rooted at z.

Top Trees: Construction

I think the important takeaway is not the exact way how the operation is performed (like the detail on raking) but on how the construction of top trees are similar to Dynamic Tree DP technique.

Basically, Dynamic Tree DP aggregates subtree information by maintaining a data structure on the light edges for each vertices.

Here, the data structure corresponds for the *rake* operation.

And the default data structure on each HLD corresponds to a *compress* operation.

Note that there are easier way to implement top trees!

Top Trees: Construction

I think the important takeaway is not the exact way how the operation is performed (like the detail on raking) but on how the construction of top trees are similar to Dynamic Tree DP technique.

Basically, Dynamic Tree DP aggregates subtree information by maintaining a data structure on the light edges for each vertices.

Here, the data structure corresponds for the *rake* operation.

And the default data structure on each HLD corresponds to a *compress* operation.

Note that there are easier way to implement top trees!

Also there are technical issues on storing vertex information, which we are not going through here (Brainless way is to create auxiliary edge for each vertex).

Updates

Handles

We will define the handle N_v of the vertex v.

If the degree of v is at least two, then N_v is the node representing $compress(v)$.

Otherwise, N_v is the topmost non-rake node that have v as an endpoint.

If you think the *rake* process as a tree, then it's a leaf of that tree.

If v is a root, then N_v is the root of the entire top tree.

Other approach is to define a separate nodes for each vertices. I emphasize that the following method is not a only possible implementation.

Splay, Splice

Now we introduce a basic tool: *splay* and *splice*.

Splay is a process of escalating a node as a root without changing the currently represented tree. By casing upon the node type splaying can be *well* handled.

Splice is akin to the *access* operation of LCT. Let $p_1, p_2, \ldots, p_k$ be a sequence of internal paths that starts from root and leads to v. Splice cuts the path p_{k-1} into two, per the point where it is incident to p_k, and give the higher half to the p_k. In the picture below z is a root.

Repeated *splice* operation is identical to access operation in LCT.

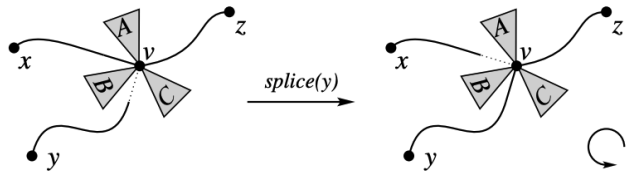


Figure 6: Splice: $y \cdots v \cdots z$ replaces $x \cdots v \cdots z$ as the exposed path.

Soft Expose

Soft Expose is a way to extract the path v, w. We denote it as $soft\_expose(v, w)$

If $v = w$ or v, w is isolated, do nothing. If v, w are in different component splay the respective handles.

Otherwise, N_w is splayed, and then N_v is brought right down to the root, while still maintaining that N_w is a root.

This makes the cluster representing the path v, w to exist, and also be very close (depth 2) at the root.

For analogy, consider how you perform range updates at splay trees.

It can be implemented by using $splay, splice$ *very well*.

Hard Expose

After the soft expose, the root node represents some path that contains path v, w.

Node representing path v, w is in depth 2, and it's combined with at most two compress operation.

Hard expose temporarily replaces this compress operation to **rake** operation, and make the root as path v, w.

This breaks some invariant, so we should undo it after the operations are done.

Link and Cut

Cut is almost fully implemented with soft expose.

Soft-expose the edge v, w to cut, then the base cluster have depth 2.

Remove the edge, and detach the parent vertex (which is the handle N_v).

Now both root lacks the right child: Find the immediate predecessor / successor of the removed edge in the tree per the cyclic order. Attach it.

For Link, do this in the opposite order: Let (a, v), (b, w) be a successor of (v, w) in both forest. Soft expose them, empty the right child, and connect.

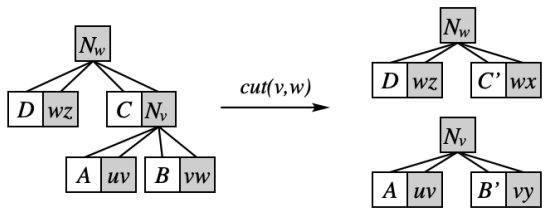


Figure 8: Cutting edge (v, w).

Interface

Local Interface

Problem 1. We can maintain the sum of all weight of edge for each component.

Proof. For each cluster, maintain a value $sum(c)$ which stores the corresponding edge weight in base clusters, and the sum of two child cluster in others.

This is a trivial problem.

Here, we don't have to care the cluster type.

Local Interface

Problem 2. Given a path v, w, we can compute the maximum weight edge in path in $O(\log n)$ time.

Proof. If you expose the path v, w, then the edges inside the paths are compressed, and the edges outside the paths are raked.

Maintain a value $pathMax$, which takes the maximum in compress cluster, and ignores the raked child in rake cluster.

Local Interface

Problem 2. Given a path v, w, we can compute the maximum weight edge in path in $O(\log n)$ time.

Proof. If you expose the path v, w, then the edges inside the paths are compressed, and the edges outside the paths are raked.

Maintain a value $pathMax$, which takes the maximum in compress cluster, and ignores the raked child in rake cluster.

Problem 3. Given a path v, w, we can compute the maximum weight edge **NOT** in path in $O(\log n)$ time.

Proof. Maintain a value $ignoredMax$ which collected all ignored edges.

Local Interface

Problem 2. Given a path v, w, we can compute the maximum weight edge in path in $O(\log n)$ time.

Proof. If you expose the path v, w, then the edges inside the paths are compressed, and the edges outside the paths are raked.

Maintain a value $pathMax$, which takes the maximum in compress cluster, and ignores the raked child in rake cluster.

Problem 3. Given a path v, w, we can compute the maximum weight edge **NOT** in path in $O(\log n)$ time.

Proof. Maintain a value $ignoredMax$ which collected all ignored edges.

For both problem, you can use the sum instead of max, to compute the length of path.

Local Interface

Problem 4. Given a path v, w, we can add a constant in path, or compute the maximum weight edge in path in $O(\log n)$ time.

Proof. Now lazy propagation comes in play, but the principle is identical. Compress cluster propagates to both child. Rake cluster propagates to the original child.

Local Interface

Problem 5. We can maintain the diameter in $O(\log n)$ time.

Proof. Maintain the 1) diameter, 2) path length, 3) furthest point from each endpoint of path.

In both case, two clusters have a **center** vertex which are common to both cluster. With information 3, we can trace the new diameter that goes across two clusters.

It's easy to maintain info 2 and 3.

Local Interface

Problem 6. We can mark, unmark, and find the nearest marked vertex in $O(\log n)$ time.

Proof. Let's assume that there is an auxiliary edges for each vertex (it's position does not matter).

For each edge, maintain the closest marked edge for each endpoint, and the path length.

Both can be well merged in two types of cluster merge.

For query, you can simply expose the auxiliary edge for each vertex. Mark sets the distance 0, Unmark sets the distance ∞.

Non-local interface

Until this point, we have only discussed with *local* property. A property is local if it can be aggregated from the subtrees.

Consider a binary tree over an array. Range sum is a local property.

However, finding a k-th element over an array, is not a local property. The child's k-th element have nothing to do with it's parent.

In a binary search tree context, we solve this by descending down the tree from top to bottom. We want to do the same for the top tree.

If we can do this, we can solve many interesting problems: For example, we can compute the level ancestor, or maintain the centroid of the tree.

Non-local interface

Let's take a look at the following lemmas:

Lemma (Center). Let T be a tree, and A, B a neighboring cluster where $A \cap B = \{c\}$ and $A \cup B = T$. If $max\_dist(A, c) \geq max\_dist(B, c)$, where $max\_dist(A, w)$ is the maximum distance between any node $v \in A$ and w, then A contains all centers.

Lemma (Centroid). Let T, A, B, c same as above, If $|A| \geq |B|$, then A contains a centroid of T.

Note that the things are bit different for level ancestor and LCA: But they are actually easier, since you can just expose the path and search down in the compressed cluster, like the usual BST. I will leave this as an exercise.

Non-local interface

We can abstract such kind of property into the following: Given the root cluster, we have a $choice$ function that decides which part the desired vertex or edge lies in.

With the choice function, we bisect the candidate set of the answer C.

Non-local interface

We can abstract such kind of property into the following: Given the root cluster, we have a $choice$ function that decides which part the desired vertex or edge lies in.

With the choice function, we bisect the candidate set of the answer C.

In the beginning. $C = T$. At first, it is easy to reduce them by half.

To move on for subsequent iteration, we modify the top tree so that the child of $C = (A, B)$ lies in the different child for root cluster $R = (A', B')$.

The verdict of $choice(R)$ decides whether $C \leftarrow A$ or $C \leftarrow B$.

I don't know how this modification is implemented (nobody mentions it...) but this is one possible abstraction.

Concluding Remark

Concluding Remark

Andrew He (*ecnerwal*) shared his implementation of top trees. If you want to implement top trees, it will be a very helpful resource. `https://github.com/ecnerwala/cp-book/blob/master/src/top_tree.hpp`

I'm sorry for not demonstrating how to do subtree queries. It is sad that I couldn't implement this before doing the lecture.

Might see you in future Data Structure Stream!