A Basic Exploration of Dynamic Trees

Translation compiled by Kirito Feng

Zhiao Huang

CTSC 2014

Contents

Summary

One OI topic is dynamic trees. Algorithms for solving such problems include heavy-light decomposition, centroid decomposition, and the link/cut tree data structure, are commonly known and studied. At the same time, methods for maintaining subtree information using the tree's DFS order or an Euler tour tree are also well known. This article will summarize a few relevant problems and propose some original solutions. At the end, this article will introduce two extensions of the link/cut tree: top tree and link/cut cactus, which are used to solve related problems.

§1 Dynamic Tree Problems

Dynamic Trees refers to the problem of dynamically maintaining relevant information on a tree. But sometimes when it comes to dynamic trees, it refers specifically to the LCT data structure.

In typical dynamic tree problems, you may encounter the following operations:

1. Add or delete an edge to maintain the structure of the tree.
2. Perform path operations to maintain the weights of each node.
3. Perform path queries on the tree.
4. Modify the weight of a subtree, and propagate this change to all nodes in said subtree.
5. Perform subtree queries on the tree.

§2 Older Dynamic Tree Problems

There have been a variety of tree-related topics in algorithm competitions. These are typically solved by using another tree-like structure, such as the balanced binary search trees.

In some older problems, the tree's structure is more or less static. In these cases, the difficulty of the problem is typically from maintaining the values of the nodes efficiently, rather than from maintaining the structure of the tree.

§2.1 Heavy-Light Decomposition

The core of the heavy-light decomposition algorithm is categorizing the edges of the tree into light and heavy edges according to certain rules. The chain formed by heavy edges is maintained using an appropriate data structure. You can prove that for any node, the path from it to the root can be divided into $\mathcal{O}(\log N)$ heavy and $\mathcal{O}(\log N)$ light edges. Thus you can solve each query in $\mathcal{O}(\log N \times f(N))$, where $f(N)$ is the time complexity to query the data structure.

§2.2 Centroid Decomposition

Centroid decomposition separates the tree into subtrees based on the centroid, thus reducing the size of the tree by half each time.

Centroid decomposition is typically used to solve problems related to either paths or vertices. In simple problems, it is usually possible to iterate through the relevant path

information by considering the LCA of the path's two endpoints. Each iteration only needs to consider the nodes in its subtree, as only these nodes can have an LCA equal to the centroid.

§3 Link/Cut Trees

§3.1 What is a LCT

An LCT is a dynamic tree that represents chains of nodes, which are maintained with the Splay operation. Using the access operation of the dynamic tree, you can remove the path from a given node to the root, and replace it with the path from another node to the root. By using lazy propagation with Splay, the root of the tree can be changed. The LCA can be found by using two access operations.

Using amortized analysis, we can see that a single operation takes amortized $\mathcal{O}(\log N)$. This allows for quite a few problems to be solved.

§3.2 Maintaining Edge And Vertex Information

For some problems, you will need to maintain information on both edges and vertices.

One such method is to create a dummy node u' for each edge (u, v), and maintain the edges (u, u') and (u', v).

> **Remark.** From Tarjan and Slator's original paper, I suspect you can also can also convert the node u to an edge (u, u'), and then query these.
>
> Make all connections to u', and maintain $depth(u) < depth(u')$. If you have to perform a reverse operation, swap u and u''s pointers, and when performing access, always check that you are connecting to u' instead of u.

Another method involves denoting each edge as a pair of edges, and maintaining these when you evert the tree.

> **Remark.** I'm not sure how well-known this technique is, but I have never heard of it before. As such, here goes my best guess as to what this means.
>
> For each node u, maintain the edges p, u and u, c, where p is the parent of u, and c is the preferred child. The evert stays the same, and the only difference is when you perform an access on a reversed tree. In that case, you should check that you are connecting to the higher of the two edges. This can be done in $\mathcal{O}(1)$ time by maintaining the two edges' pointers in an array.
>
> This implies that you should splay every tree before converting virtual edges to real edges, as then you know which edge is lower in the tree.

§3.3 How to use the LCT

Consider one of the simplest problem:

1. You can either add or delete an edge
2. You can add a constant to all numbers on a given path
3. Query the weights of all edges along a given path

Consider how to implement these operations with a dynamic tree:

1. An edge can be added by rooting the tree at one of the new endpoints, and adding a virtual edge to the other endpoint
2. An edge can be deleted by rerooting the tree to one of the endpoints, and splaying the other endpoint to the root of the splay tree. By deleting the edge from the root of the Splay tree, you have removed the edge
3. Set one of the endpoints of the path to the root of the real tree, and splay the other endpoint so that it becomes the root of the virtual tree. This splay tree now represents the desired path, and all lazy and query operations can be trivially maintained.

Any problem about maintaining certain aggregates over subarrays that can be solved with balanced binary search trees can be extended to trees using LCTs.

§3.4 Simple Applications of Link/Cut Trees

§3.4.1 Obvious Dynamic Tree Problems

Some problems can be solved with a direct application of link/cut trees. One such problem is BZOJ 3091 城市旅行.

This problem requires you to add and delete edges, lazily adding a delta to all nodes on a path, as well as querying for the subpath with the largest sum, and for the sum of any given path.

The tree's structure can easily be maintained with a link/cut tree, and the queries can be solved by maintaining the maximum prefix, suffix sums, as well as the total sum of each segment. The maintenance of these values is a classic problem, and as such we shall not go into any further detail.

§3.4.2 Problems With Hidden Tree Structure

There are many problems that do not explicitly tell the problem solver that there is a tree structure, but once it is found, it can be maintained using a dynamic tree. One sample problem is IOI 2011 P5 Elephants, where the observation that each node has a unique parent suggests a tree structure, which can be maintained using a dynamic tree.

§3.5 Applications of LCT in Graph Theory Problems

Trees are an important tool for describing graphs. The flexibility of dynamic trees makes it possible to solve some graph problems.

§3.5.1 Dynamic MST (Adding Edges Only)

Finding the minimum spanning tree is a classic problem. However, dynamically maintaining the minimum spanning tree is a rather difficult problem.

If we use divide-and-conquer, we can obtain a time complexity of $\mathcal{O}(M \log M)$. However, this algorithm is offline, and the online algorithm has both a nasty implementation and poor time complexity.

> **Remark.** I believe there's an $\mathcal{O}(M \log^2 M)$ randomized online algorithm for this problem...

If we only allow for the addition of new edges, we can solve this problem online using a dynamic tree. By applying the cycle property each time we process a new edge, we can find the minimum spanning tree. To do so, we only need to support querying the maximum edge weight along a given path, which is a classical operation that can be supported in $\mathcal{O}(\log N)$ time.

§3.5.2 Minimal Path Difference Query

Given a weighted, undirected graph, find a path from S to T such that the difference between the maximum and minimum edge weight is minimized.

This problem can be solved using dynamic trees: Sort by edges, enumerate the edge from the smallest to the largest, as the problem can be reduced to finding a path where the maximum minimum weight in edges that are less than the current maximum value. Maximizing the minimum weight is actually a subset of the minimum bottleneck spanning tree problem, moreover a minimum spanning tree is always a minimum bottleneck spanning tree, therefore the previous method can be applied to solve the problem.

§3.5.3 Determining if a Dynamic Graph is Bipartite

Given a graph, you can add and remove edges. After each operation, determine if the graph is bipartite or not. You can process offline.

This problem is similar to maintaining the dynamic connectivity of a graph with divide and conquer. As such, we will not focus on the divide and conquer component as much, and instead focus on the dynamic tree's operations.

For each edge, we can process its deletion time offline, and construct a maximum spanning tree of these edges. When a cycle is formed, check if this is an even or an odd cycle. If the cycle is odd, then add the new edge to a set. When an edge is deleted, remove it from the set. The graph is bipartite iff the set is empty. The correctness of this algorithm is not obvious, but it is possible to prove.

> **Remark.** I believe this is related to the Edmond's Blossom algorithm...

Naturally, these operations can all be maintained with a dynamic tree.

§4 DFS Tour and Euler Tour Trees

At this point, it should be noted that most of the previous dynamic tree problems are related to maintaining information on chains. These problems can typically be solved with LCTs, as they themselves maintain information on chains. However, this leads to difficulty when trying to solve problems related to subtree queries.

This is where another powerful tool comes in: the DFS traversal order. It should be noted that if the nodes are written out in DFS order, then subtrees map to contiguous intervals, and these intervals are either disjoint or completely contained within each other; there are no intervals that partially overlap. These properties allow you to maintain the DFS tour using a variety of data structures.

§4.1 Simple Subtree Query Problems

Many problems involve obvious subtree queries, and these can be solved easily by considering the DFS ordering of the tree.

§4.1.1 Sample Problem - Tree

Problem Source: 2012 Chinese national team training

The problem effectively asks you to support the following operations on a tree:

1. Reroot the tree
2. Change the value of a vertex
3. Query the minimum value in a given subtree

After a simple analysis, it can be found that supporting the queries with an LCT is not wise. Even though there is a operation that easily changes the root, the structure of the tree itself is static.

This motivates us to consider the DFS tour of the tree. You can maintain the updates and queries using a segment tree over the Euler tour.

To support the rerooting operation, we consider two cases. Let us root the tree arbitrarily initially. Consider a query for the subtree minimum of node u, with root v. If v is not in u's subtree, then u's subtree is not changed, and it can be queried directly. Otherwise, let w be the child of u such that w is an ancestor of v. Then we would query for the complement of the subtree of w.

§4.2 Maintaining the DFS Tour With a BBST

To obtain the DFS tour of a tree, you must first DFS the tree. What happens if the structure of the tree changes due to inserting or deleting edges?

If you fix the root, inserting and deleting a node corresponds to deleting a number from a dynamic array, which can be maintained with a balanced binary search tree.

Similarly, inserting and deleting a subtree corresponds to either inserting or deleting a range of values, which can be maintained with either a splay tree, or a split-merge treap.

Thus combining the DFS tour with a balanced binary search tree allows for many simple problems to be solved.

> **Remark.** Yes, this is just a crippled Euler Tour tree. Amazing.

§4.2.1 Mashmokh's Designed Problem

Problem source: Codeforces Round 240 Div1 E

The problem asks you to support the following operations on a tree:

1. Find the distance between nodes u and v
2. Disconnect node u from its father and connect it to its hth ancestor
3. Find the last vertex in the DFS tour with distance k from the root

This problem does not lend itself easily to LCTs, thus we can instead consider the DFS tour of the tree.

> **Remark.** The editorial given in the paper is basically the same as the one on CF, so I'll use the CF one instead.

Firstly, we store the DFS tour with 1 and -1s: We store 1 when we enter a node's subtree, and -1 when we leave it. We maintain this with a balanced binary search tree, and for each node, we store the minimum and maximum sum corresponding to its range.

The distance between two nodes is equal to $\text{depth}(u) + \text{depth}(v) - 2\,\text{depth}(\text{LCA}(u, v))$. We can easily query the LCA by finding the node of minimal depth along the segment from u to v.

To find the hth ancestor, we find first node of depth h that appears after the -1 corresponding to node u. We can then cut the range corresponding to u's subtree, and insert it before the node of depth h. This can be done by binary searching as each element is either 1 or -1, thus the sums are all consecutive.

To find the last node with depth k, you can binary search for it, using a similar technique as above.

§4.2.2 Persistence and DFS Tour

As we can maintain the DFS tour with a balanced binary search tree, and balanced binary search trees can be made persistent, we can maintain the persistent DFS tour. This leads to the following problem.

Problem source: Mao Xiao

Given a tree rooted at node 0, you are asked to support the following operations:

1 Add a node with weight 0 to the tree

2 Increase the value of all nodes in a given subtree by some value c

3 Query for the weight of single node

4 Query for the weight of a given subtree

5 Revert to a previous revision of the tree

> **Joke.** What an *organic* usage of persistence!

All of the query and update operations are very classical, and can be easily supported with a balanced binary search tree. However, the problem becomes a lot harder if persistence is required.

Persistence cannot be directly applied, as you have to maintain aggregates. If a node u is updated, then all of its parents in the BBST will have their aggregates modified too. Thus we have to update how we access these parents too. This can be maintained by using a persistent array. As you access u in the BBST, you have to update the pointers to all of the parent nodes along the path. If we use a treap, then you only have to access an expected $\mathcal{O}(\log N)$ nodes, and maintaining the persistent array with a segment tree takes $\mathcal{O}(\log N)$ per update. Thus you have $\mathcal{O}(\log^2 N)$ per query.

§4.3 The Euler Tour

The following sections will introduce the Euler Tour Tree.

§4.3.1 What is the Euler Tour?

Let T be an undirected tree. For each edge, split it into two directed edges, and chose an arbitrary node S as the starting point for the DFS. This is the Euler tour of T. Note that S can be any arbitrary node because the Euler tour forms a cycle.

§4.3.2 Euler Tour Sequence

An **Euler tour sequence** is a sequence formed by consecutive directed edges in the Euler Tour.

§4.3.3 Euler Tour Tree

An Euler tour tree is a binary search tree that maintains the Euler tour sequence. This is generally implemented with either a splay tree or a treap. The Euler tour focuses on maintaining the information of the whole tree. Unlike with the DFS tour, it maintains the information of all subtrees with an arbitrary root. Thus it can be used to implement the root-changing operation operation which can be quite difficult to implement with the DFS tour.

§4.3.4 Some Implementation Details

You only need to support two operations: adding and removing an edge.

(a) Adding an edge (connecting two trees): You can see this as merging two cycles. Split one of the cycles, and then insert the other one, and then merge the two cycles together.

(b) Deleting an edge (disconnecting two trees): It can be seen as splitting a cycle into two cycles. This operation is not very difficult.

§4.4 Simple Applications of ETT

Given a graph, you can add or delete any edge. After each operation, return the connectivity of the graph.

The implementation of this question is somewhat complex, as such, the following is only a brief description.

Because any two vertices can be connected, it is not necessarily true that the graph will be a forest. Additionally, simply maintaining a forest will fail to account for edge deletions.

Thus, we can use divide and conquer to solve this problem: we assign a label of e_i to each edge, so that the size of connected component of any edge with a label less than or equal to some constant e is at most $2e$. It can be seen that $\max(e) = \log N$.

> **Remark.** I suspect this means assign $\{1, 1, 2, 2, 2, 2, 4, 4, 4, 4, 4, 4, 4, 4, \ldots\}$. The 2 is likely because each undirected edge is converted into 2 directed edges.

For each e, we maintain an ETT with the minimum spanning tree of all edges that are have labels less than or equal to e.

To add an edge, assign it the label $\lfloor \log N \rfloor$, and add it to the correct minimum spanning tree.

The difficulty is in deleting an edge. If the edge is not in the minimum spanning tree, it can be deleted right away. This operation takes $\mathcal{O}(\log N)$ time.

§5 Some Extensions of LCT

The LCT is not only limited to chain operations, but can also be used to solve some interesting problems. The following two examples illustrate some interesting applications of LCT.

§5.1 LCT and Subtree Maintenance

In the previous section, subtree queries were answered using the DFS tour and ETT, but does that mean these problems are unsolvable with LCT?

For example, is it possible to solve 4.1.1 with LCT? It turns out the answer is yes!

> **Remark.** The following is an expanded version of the translated solution that goes into more implementation details.

Consider an arbitrary node v. Let A_v be the maximum of the maximum value of all nodes in its subtree that are **not** on its preferred path, and the value of v. We can query the maximum value of the subtree of v by splaying v to the root of its splay tree and returning the maximum A of v and its right subtree.

We can handle updates as follows. Let every node contain a heap with the pair (v_i, i). If a node w in u's subtree is updated, we call `access` on v. If a virtual edge is converted to a real edge, then we need to update the heap. Let the endpoints of the virtual edge be p and c, such that the virtual edge is $p \to c$. We keep popping p's queue until its head is a pair (v_j, j) such that the value of j is equal to v_j. Then we insert the pair (v_w, w). Additionally, we update A_p, and then splay this value to the root, updating the tree aggregates.

The rerooting operation does not require any special processing. When the root is changed, the only nodes that are affected are the ones along the preferred path: none of the A values are affected.

§5.2 LCT and Reversing Paths

The following problem uses an LCT to implement reversing all nodes along a given path:

§5.2.1 Lord

Problem Source: 2012 Chinese national team training

You are effectively asked to support the following operations:

1. Query the sum of all nodes in y's subtree if the tree were rooted at x.
2. Reverse the path from x to y.
3. Change the value of node x to y.
4. Connect x and y.
5. With x as the root, disconnect y and its father.

While this question has subtree queries, because of the path-flipping operation, we cannot use an ETT. Instead, we must consider using an LCT.

We use the technique introduced in the previous problem: each node maintains the answer for all nodes in its subtree that are not on its preferred path. In this case, we additionally use a splay tree within each splay tree in the LCT to support revering the paths. When a path is reversed, we just set a lazy flag on the LCT. When a virtual edge $p \to c$ is converted to a real edge, we detach the subtree that corresponds the p's old right child, and append c's splay tree.

While the time complexity may appear to be $\mathcal{O}(Q \log^2 N)$, the splay operations are in fact equivalent and will not break the amortized analysis. Thus the final time complexity is $\mathcal{O}(Q \log N)$.

§6 sone1

§6.1 Problem Statement

There are a group of gods running around sxyz. They require you to maintain a tree:

1. Add or change the weight of all nodes in a specified subtree.
2. Add or change the weight of all nodes along a specified path.
3. Query the maximum, minimum, or sum of all nodes in a specified subtree.
4. Query the maximum, minimum, or sum of all nodes along a specified path.
5. Add and remove edges.
6. Reroot the tree.

§6.2 ETT Solution

This problem requires operations on chains, which is normally impossible with an ETT. At the same time, once subtree operations are involved, using an LCT becomes significantly harder.

However, it is still possible to solve this problem using a combination of both. We set the first child of each node in the ETT to the preferred child of the LCT. As each subtree is a contiguous, non-overlapping interval, reordering the children of a given node can be done in $\mathcal{O}(\log N)$.

When we call the `access` operation on the LCT, some virtual edges will turn into real edges and vice versa. Thus each time this happens, we must reorder the children in the ETT.

As both the ETT and LCT support a root-changing operation, this can be implemented without much difficulty.

Thus each operation will take $\mathcal{O}(log^2 N)$ time, as we need to modify the ETT $\mathcal{O}(\log N)$ times.

Next we consider the update and query operations. Because the preferred child is always the first child, a path corresponds to a single, contiguous interval. Thus all of the required update and query operations can be implemented as simple interval updates and queries on an ETT, backed by a BBST.

§6.3 Extending the LCT

The vanilla LCT cannot perform subtree updates and queries. While the inclusion of these operations has made the problem significantly harder, it is still possible to solve this problem with LCT in $\mathcal{O}(Q \log N)$.

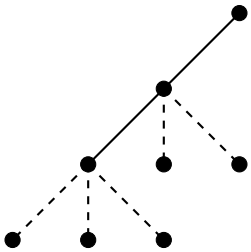
§6.3.1 A Simple Trick

As we either add or set the values of several nodes, we can rewrite our lazy flag as $ax + b$, for the sake of convenience.

§6.3.2 A Visualization

Obviously, directly applying the vanilla LCT results in great difficulty when maintaining subtree information.

Let us consider a chain:



Solid lines correspond to real edges, and dashed lines correspond to virtual edges. Points are ordered by depth.

§6.3.3 Merging Aggregate Values

Because the chain formed by real edges is maintained by a splay tree, any path-based aggregates are very easy to answer.

Next, we need to consider the virtual edges. Because we need to maintain subtree aggregates, we need to merge the aggregates of each child connected by a virtual edge. We consider what information needs to be maintained for each node:

1. Chain: for each chain, we maintain the aggregates for each node
2. Tree: the aggregates over all chains connected to the current node via virtual edges
3. All: the aggregate of `Chain` and `Tree`

Note that the chain information can easily be maintained with a splay tree.

The problem is how to update the `tree` values. Brute force is clearly too slow, so we instead need to modify the LCT.

§6.3.4 AAA Tree

The AAA tree exists for this purpose: each virtual node connected to a given node is instead placed into a balanced binary tree to maintain it. In order to support updates, we introduce **inner nodes** to form a binary tree. This is effectively equivalent to binarizing the tree, which introduces no more than $\mathcal{O}(N)$ dummy nodes.

Each chain will obviously only belong to a single AAA tree, and the AAA tree will maintain every chain attached with a virtual edge.

With the AAA tree, you will need two basic operations:

1. Add: add a node to the AAA tree, inserting an inner node if needed.
2. Del: remove a node from the AAA tree, deleting an inner node if needed.

§6.3.5 AAA Tree and The Access Operation

We now consider the most important operation: `access`.

It can be found that we can use the `add` and `del` operations to implement access.

Within the loop, we preform 3 operations repeated:

1. Find the father of the current chain
2. Turn a real edge into a virtual edge: this is just a `add` operation
3. Turn a virtual edge into a real edge: this is just a `del` operation

This makes it possible to maintain aggregates in the LCT.

§6.3.6 Maintaining Aggregates

To update aggregates, we need to use two types of lazy flags:

The first type is `treeflag`, which is used to maintain lazy updates for the subtree, excluding the nodes on the preferred path. The second is `chainflag`, used to maintain the lazy tag for the preferred path.

Propagation of `chainflag` follows standard LCT conventions.

The implementation of `treeflag` is more nuanced, and involves two cases:

1. When propagating to children within the splay tree, simply pass the lazy flag along to the children, with no modification.
2. When propagating within the AAA tree, we split the lazy flag into `chainflag` and `treeflag`. In this case, we *do* apply the lazy flag, and pass them along to all of the attached chains.

§6.3.7 Implementation Details

How do we implement this? At first glance, this appears to be very complicated. However, this is not that bad. We simply extend the `node` class of the LCT to have 4 children instead of 2. Additionally, we maintain a `type` flag to indicate whether a node is an **inner** node or not, and also a flag indicating if a node is connected within the LCT or the AAA tree.

§6.3.8 Time Complexity

A preliminary analysis would give us an upper bound of $\mathcal{O}(\log^2 N)$ per operation. However, it is possible to prove that this is indeed amortized $\mathcal{O}(\log N)$ per operation. While there is a constant factor of 96, in practice, this data structure is actually quite fast.

§6.3.9 Name?

This is sometimes referred to as a Self-Adjusting Top Tree, however, you can just view it as an extended LCT, which, in the author's opinion, is easier.

§6.4 Rake and Compress

> **Note.** This section is based on the Wikipedia article, rather than the paper, as the paper does not go into much detail.

§7 Dynamic Cactus Problems

This section is devoted to discussions problems on a dynamically changing cactus graph.

§7.1 Dynamic Cactus Problems

A **cactus graph** is a graph where every edge belongs to at most 1 simple cycle.

While older problems focused on path updates and queries, true dynamic cactus problems require you to maintain a dynamically changing graph.