Implementing Dynamic LCA

Jonathan Conroy

1 Introduction

Given a tree, the *lowest common ancestor* (LCA) problem asks us to maintain a data structure that can support queries of the following form: given two nodes, what is the deepest node in the tree that is an ancestor of both nodes?

If the tree is static, there is a relatively straightforward data structure that can answer queries in $O(1)$ time with only $O(n)$ preprocessing time and space. The idea is to reduce from the RMQ ("range minimum query") problem. In RMQ, we preprocess static array and, given two query indices, the goal is to find the the lowest value in the array between those two indices. Glossing over a few details, RMQ can be solved in $O(n \log n)$ space by precomputing solutions to all queries with lengths that are powers of 2, and indirection with lookup tables can reduce this to $O(n)$ space/preprocessing time. We can reduce LCA to RMQ with an Euler tour.

If the tree is dynamic, the problem becomes more difficult. In 2005, Cole and Hariharan showed that it is possible to maintain a tree that could answer LCA queries in $O(1)$ time while also supporting constant time insertion, deletion of leaves, and deletion of nodes with exactly one child [1]. This paper is complicated, but it builds on a simpler data structure proposed by Gabow [2]. Gabow shows that it is possible to answer LCA queries while also supporting *amortized* constant time leaf insertion. Gabow also shows that it is possible to support amortized $O(\alpha(m, n))$ linking of subtrees, where α denotes the inverse Ackermann function, n is the number of nodes involved, and m is the number of LCA/link operations - interting a leaf is a special case of this operation.

The project will explore a limited version of Gabow's dynamic LCA data structure. Specifically, we implement $O(1)$ LCA queries on a dynamic tree with amortized $O(\log n)$ leaf insertion and $O(n)$ space. The next section of this writeup will briefly describe the data structure and the decisions made impementing it; the next section will describe how the code can be run; the last section will discuss testing and performance.

2 Gabow's Data Structure

Gabow introduces a data structure that can answer LCA queries in constant time on a *static* tree with $O(n \log n)$ space and preprocessing time. He then shows how to maintain

this data structure in amortized $O(\log^2 n)$ time per insertion. One application of indirection reduces this to $O(n)$ space and $O(\log n)$ amortized insertion. A second application of indirection reduces insertion to $O(1)$ amortized time, though that was not explored in this project.

2.1 Static Preprocessing

A tree is represented with the `ExpensiveTreeNode` class (named so because insertions on this sort of tree will be expensive - $O(\log^2 n)$ amortized). A method is provided for adding nodes without performing any preprocessing. After construction, a `preprocess()` procedure can be called on the root of the tree to prepare it for LCA queries.

The first idea of Gabow's data structure is to partition the tree according to a heavy-light decomposition. A node is "heavy" if its subtree size is greater than half the subtree size of its parent; it said to be the "apex" of a path if it is not a heavy child. The tree then compressed so that the parent of each vertex is the first proper ancestor that is an apex in the uncompressed tree.

The compressed tree and uncompressed tree are both represented by the `ExpensiveTreeNode` class. This class explicitly maintains both the `uncompressedChildren` and `uncompressedParent`, as well as the compressed `children` and `parent` of the node. This allows us to easily translate between the compressed and uncompressed representation, which we do frequently. Further, maintaining seperate variables makes it obvious which operations occur on the compressed tree and what operations occur on the original.

The second idea in Gabow's data structure is to mantain a "fat preorder numbering". Essentially, means that each node is assigned an interval that satisfies certain nice properties. For example, all children of a node must be represented by disjoint intervals that fit within the interval of the node itself. (Notice that this means that the interval grows as the size of the subtree rooted at the node increases.) In addition, there must be some "buffer space" on either end of the interval, which no intervals can occupy. Gabow shows that the LCA of two nodes x and y must have an interval with length greater than $|start(x) - start(y)|$ (where $start(z)$ denotes the first integer in z's interval). In fact, the LCA is either the first ancestor of x that satisfies this property, or it is the parent of the first ancestor of x that satisfies this property.

To help answer LCA queries, each node stores a precomputed "ancestor table", where the i-th entry stores the last ancestor whose interval is smaller than β^i (for some parameter β determined by the fat preodering). Each ansestor table has size $O(\log n)$. Gabow shows that (with some case analysis) these tables are sufficient to determine the LCA in the compressed tree in constant time. In the implementation, these tables are represented by `vector`s containing `ExpensiveTreeNode*` pointers - this allows them to grow dynamically, which will be important later on.

Note that the fat preordering intervals can grow quite large - Gabow proposes parameters

such that the root is assigned an interval that has length of $5 \cdot n^4$. The endpoints of intervals are represented as 64-bit integers (specifically, a `long long int`), so this puts a limit on the size of trees that can be processed in this way. This means that trees are capped at about $35,000$ elements - which is rather restrictive for practical purposes. This could be solved by representing intervals with a BigInteger. For proof-of-concept testing, though, 64 bits was sufficient.

The last thing to note about the static case is that we can compute "characteristic ancestors" in constant time. The "characteristic ancestors" of two nodes x and y consist of the LCA, as well as the child of the LCA that is an ancestor of x and a child of the LCA that is an ancestor of y. These characteristic ancestors are used to convert the LCA in the compressed tree into the LCA in the uncompressed tree. They are also used when applying indirection.

2.2 Making it Dynamic

The static data structure can be extended to support the `add_leaf` operation. When a leaf is added, it is initially added as part of a new path in the compressed tree (ie. it is treated as an apex, regardless of whether or not it is actually a heavy child). Note that this insertion changes the subtree sizes: the subtree sizes that were used to assign the original fat preordering are no longer valid. Thus, we have to check if the parent interval has "room" to fit in the the new leaf interval. Specifically, we travel up the (compressed) tree to find the last interval whose subtree size has grown "too large" relative to the subtree size used to assign its fat preordering numbers. We reorganize the subtree headed by this node, replacing it with the heavy-light path compressed tree and assigning new intervals.

To implement this, the `ExpensiveTreeNode` maintains both the original `subtreeSize` used to assign its fat preordering, as well as a dynamically updated size (appropriately named `dynamicSubtreeSize`). The reorganization occurs in the `recompress()` function. During a reorganization of a subtree rooted at node x, new intervals are assigned based on the largest endpoint of the buffered intervals of the compressed parent of x. As such, a `largestChildEndBuffer` augmentation is maintained at each node.

2.3 Indirection

The dynamic data structure described above takes $O(n \log n)$ space and has amortized $O(\log^2 n)$ insertions. We can reduce these requirements by applying a layer of indirection. The `ExpensiveTreeNode` data structure will be used to maintain a "summary" data structure with $O(n/\log n)$ nodes. The entire data structure will be stored in a new class, `MultilevelTreeNode`.

Nodes in `MultilevelTreeNode` are partitioned into what Gabow calls "2-subtrees" of size at most $O(\log n)$. (Gabow refers to the "summary" tree as T_1 and the full tree as T_2, which explains the name.) In the implementation, each node holds a pointer to the root of the 2-subtree it belongs to. This root stores information about the entire subtree, such as the

`twoSubtreeSize` and a pointer to the corresponding `summaryNode` (these values are not maintained for non-root subtrees). Each "full" 2-subtree has an associated `ExpensiveTreeNode` node in the summary tree.

To answer an LCA query of x and y, the summary data structure is used to find the 2-subtree containing the LCA, as well as the lowest node in that subtree that are ancestors of x (resp. y). This is done with case analysis on the characteristic ancestors. To answer an LCA query within a 2-subtree, we maintain a bitstring `ancestorWord` for each node z such that the i-th bit is 1 if and only if the i-th node in the subtree is an ancestor of z. The root of each subtree mantains a vector `intToSubtreeNode` that gives a correspondence between integers and nodes in the subtree. The LCA between two nodes within the same subtree can then be found by examining the `ancestorWord`s of each.

Each `ancestorWord` can be stored in a single RAM word because it has size $O(\log n)$. In fact, the reason we wanted to partition the full tree into subtrees of size $O(\log n)$ is precisely because we know that we can compute LCA queries in constant time on these small subtrees. In the dynamic case, we do not know n in advance. The easiest way to resolve this is to set the maximum size of the 2-subtree equal to the number of bits in a RAM word (in this implementation, we use an `unsigned int` with 64 bits) - this will be at least $\Omega(\log n)$, and it only improves performance if it goes over.

3 Code Usage

The `MultilevelTreeNode` class should be used to construct a tree supporting dynamic LCA. A node can be created with the constructor `MultilevelTreeNode(std::string id)`, where the input is some string data stored by the node. A tree is simply represented by the root node. Given two pointers to `MultilevelTreeNode`s, we can use the `MultilevelTreeNode::lca` method to determine their LCA in constant time. See the snippet below for a minimal example:

```
#include "lcaMultilevel.hpp"
#include <iostream>

int main(){
    MultilevelTreeNode* root = new MultilevelTreeNode("root");
    MultilevelTreeNode* node1 = new MultilevelTreeNode("node1");
    MultilevelTreeNode* node2 = new MultilevelTreeNode("node2");
    MultilevelTreeNode* node3 = new MultilevelTreeNode("node3");

    root->add_leaf(node1);
    root->add_leaf(node2);
    node2->add_leaf(node3);

    root->print();
```

```
    MultilevelTreeNode* lcaNode = MultilevelTreeNode::lca(node1, node3);
    std::cout << "LCA of node1 and node3: " << lcaNode->data << std::endl;

    root->deleteNode();

    return 0;
}
```

3.1 Testing Correctness

A naive $O(n)$ LCA method was implemented to test correctness of the $O(1)$ implementation. The naive method moves up the tree from x and from y and records a vector of the nodes it sees along the way, and then it compares the paths of x and y to determine the deepest common node.

Prüfer sequences were used to generate random testing trees. A Prüfer sequence of a tree with n vertices is a list of $n-2$ numbers that fully capture its structure. Random sequences of integers were generated and translated into trees: see `https://en.wikipedia.org/wiki/Pr%C3%BCfer_sequence` for the pseudocode that inspired this approach. To construct a sequence of `add_leaf` operations from a random tree, random leaves were iteratively chosen from the tree, removed, and recorded. Note that constructing random trees in this way is rather slow, but testing with random trees provides compelling evidence for correctness.

1000 random trees of 1000 nodes were generated, and 1000 random LCA queries were run. Smaller tests on trees of size 10000 were also run. This is implemented in `test.cpp`. In all cases, the naive implementation matched the LCA implementation of preprocessed static trees, incrementally-built dynamic trees, and incrementally-built multilevel dynamic trees.

3.2 Testing Efficiency

Four different approaches were compared.

1. The **naive** approach. An `ExpensiveTreeNode` tree was build using the `addaddLeafNoPreprocessing` method - $O(n)$ space and preprocessing time - and queries were run with the naive $O(n)$algorithm.

2. The **static** approach. An `ExpensiveTreeNode` tree was built using the `addaddLeafNoPreprocessing` method, and then the `preprocess()` method was called. Construction should take $O(n \log n)$ time and space; queries should take $O(1)$ time.

3. The **expensive dynamic** approach. An `ExpensiveTreeNode` tree was built using the `add_leaf` method. Construction should take $O(n \log^2 n)$ time and $O(n \log n)$ space; queries should take $O(1)$ time.

Approach	**Construction (ms)**	**Query (μs)**
Naive	2.59	2.90
Static	62.88	0.65
Expensive Dynamic	703.18	0.80
Multilevel Dynamic	3.81	0.49

Table 1: Results of efficiency tests

4. The **multilevel dynamic** approach. A `MultilevelTreeNode` tree was built using the `add_leaf` method. Construction should take $O(n \log n)$ time and $O(n)$ space; queries should take $O(1)$ time.

100 random trees with 10,000 nodes were generated. Preprocessing was run 10 times for each, and the average time was recorded. 1,000 random LCA queries were run, and the average time was recorded. The results can be found in Table 1.

There are a few interesting things going on here. First, note that both "static" and "expensive dynamic" have fast query times relative to the naive approach but have have much higher construction times. This is as expected. More suprizing is just how well the multilevel dynamic approach performs. The construction time is blazingly fast - despite being theoretically $O(n \log n)$, it performs comparably to the $O(n)$ construction for the naive tree. The extreme speedup is partly due to the fact that 2-subtree sizes were set to be 64 (the size of a RAM word) as opposed to being $\log n$ (which is approximately 13 in this case), as discussed in section 2.3.

The query times for all three variants of Gabow's algorithm were significantly faster than the naive algorithm. It is again suprising that the multilevel approach is faster than both "static" and "expensive dynamic." This can likely be understood by considering the cache - on a small tree, random LCA queries are more likely to cause cache hits and on a larger tree.

Overall, Gabow's approach performed extremely well - there was a significant improvement in LCA runtime at very little cost while preprocessing.

References

[1] Richard Cole and Ramesh Hariharan. Dynamic lca queries on trees. *SIAM Journal on Computing*, 34(4):894–923, 2005.

[2] Harold N. Gabow. A data structure for nearest common ancestors with linking. *CoRR*, abs/1611.07055, 2016.