Winter School 2003, St. Petersburg

The Least Common Ancestor Problem

Stefan Kiefer

January 30, 2003

Abstract

The *Least Common Ancestor Problem* can be used to solve many other algorithmic problems on trees. It is shown that an efficient *and* simple solution is possible.

1 Introduction

One of the most fundamental algorithmic problems on trees is how to find the *Least Common Ancestor (LCA,* also known as *Lowest Common Ancestor* or *Nearest Common Ancestor)* of a pair of nodes. The LCA of nodes u and v in a tree is the shared ancestor of u and v that is located farthest from the root.[1] More formally, the LCA Problem is stated as follows: Given a rooted tree T, how can T be preprocessed to answer LCA queries quickly for any pair of nodes? Thus, one must optimize both the preprocessing time and the query time.

Finding LCAs in trees arises in a number of applications. For example, it arises in computing maximum weight matchings in graphs, in computing longest common extensions of strings, finding maximal palindromes in strings and matching patterns with k-mismatches. The tree involved in many of these applications is a *Suffix Tree*.[2] It is also proved useful in *bounded tree-width* algorithms[3][Als02]. Furthermore, finding LCAs has some relevance in the context of computational biology [Gus97].

Due to the many applications, the LCA problem has been studied intensively. In [Tar84], Harel and Tarjan showed the surprising result that LCA queries can be answered in constant time after only linear preprocessing of the tree T. In [Sch88], Schieber and Vishkin introduced a new and simpler algorithm. Nevertheless, algorithms for the LCA problem were still hard to implement.

By examining and sequentializing a highly parallelizable algorithm [Ber89], the authors of [Ben00] presented a simple sequential algorithm with the same optimal performance as the

[1] One can also think of the *youngest* common ancestor.

[2] This data structure is the subject of another talk at this Winter School.

[3] that are also subject of another talk at this Winter School

previously known algorithms. This algorithm is also presented in this paper.

2 Definitions

2.1 The Least Common Ancestor (LCA) problem

Structure to Preprocess: A rooted tree T having n nodes.

Query: For nodes u and v of tree T, query $\text{LCA}_T(u, v)$ returns the least common ancestor of u and v in T, that is, it returns the node furthest from root that is an ancestor of both u and v. (When the context is clear, the subscript T on the LCA is dropped.)

The *Range Minimum Query (RMQ) Problem*, which seems quite different from the LCA problem, is, in fact, intimately linked.

2.2 The Range Minimum Query (RMQ) problem

Structure to Preprocess: A length n array A of numbers.

Query: For indices i and j between 1 and n, query $\text{RMQ}_A(i, j)$ returns the index of the smallest element in the subarray $A[i \ldots j]$. (When the context is clear, the subscript A on the RMQ is dropped.)

In order to simplify the description of algorithms that have both preprocessing and query complexity, the following notation is used: If an algorithm has preprocessing time $f(n)$ and query time $g(n)$, we will say that the algorithm has complexity $\langle f(n), g(n) \rangle$.

3 A linear reduction from LCA to RMQ

The purpose of this section is to show that a LCA problem can be efficiently solved by transforming it into a RMQ problem and solving the latter.

As mentioned before, our goal is to have an algorithm with linear preprocessing time. That suggests doing a tree traversal. We will in fact perform a depth first search (DFS) on the tree, giving us the so called *Euler Tour*.[4] An example is shown in the figures 1 and 2. The resulting Euler Tour can be stored in an array E of length $2n - 1$ as suggested by table 1.

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
array E	1	2	5	2	6	2	7	8	7	9	7	2	1	3	1	4	1

Table 1: The Euler Tour stored in an array

[4]The Euler Tour of T is the sequence of nodes we obtain if we write down the label of each node each time it is visited during a DFS. The array of the Euler tour has length $2n - 1$ because we start at the root and subsequently output a node each time we traverse an edge. We traverse each of the $n - 1$ edges twice, once in each direction.

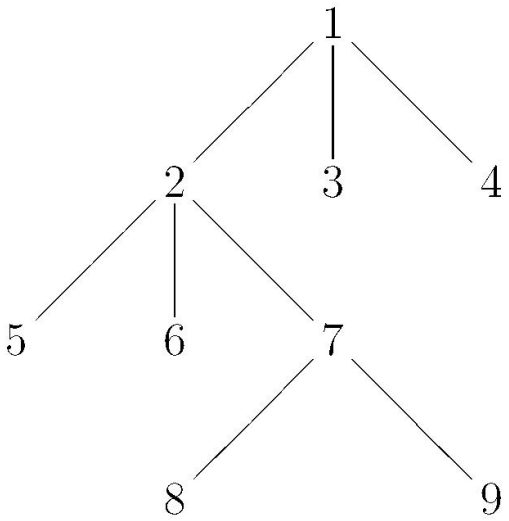

Figure 1: An example tree

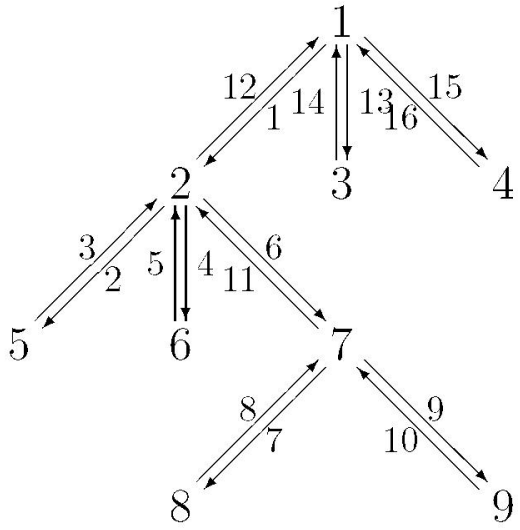

Figure 2: An Euler Tour

Observation 3.1 *The LCA of nodes u and v is the shallowest node encountered between the visits to u and to v during a depth first search traversal of T.*

Observation 3.1 suggests storing the "level information" in an additional array L of length $2n-1$ (Table 2).

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
array E	1	2	5	2	6	2	7	8	7	9	7	2	1	3	1	4	1
array L	0	1	2	1	2	1	2	3	2	3	2	1	0	1	0	1	0

Table 2: The Euler Tour with additional level information

Furthermore, we can compute the first occurrences of all nodes in an Euler Tour. We store these first occurrences in an array R of length n (Table 3).[5]

node	1	2	3	4	5	6	7	8	9
array R	1	2	14	16	3	5	7	8	10

Table 3: The first occurrences

Note that all three arrays can be filled during a DFS. That is, the arrays can be built up in linear time.

Theorem 3.2 *If there is an $\langle O(n), O(1)\rangle$-solution for RMQ, then there is an $\langle O(n), O(1)\rangle$-solution for LCA.*

Proof: The reduction is as follows. Given the tree T, compute the arrays E, L, and R as described above. Then preprocess L for RMQ (in linear time, following the assumption). It is now claimed that $\text{LCA}_T(u,v) = E[\text{RMQ}_L(R[u], R[v])]$. To see that, observe the following facts:

- The nodes in the Euler Tour between the first visits to u and to v are $E[R[u], \ldots, R[v]]$ (or $E[R[v], \ldots, R[u]]$).
- The shallowest node in this subtour is at index $\text{RMQ}_L(R[u], R[v])$, since $L[i]$ stores the level of the node at $E[i]$, and the RMQ will thus report the position of the node with minimum level. (Recall Observation 3.1.)
- The node at this position is $E[\text{RMQ}_L(R[u], R[v])]$, which is thus the output of $\text{LCA}_T(u,v)$.

To calculate the query time observe that an LCA query in this reduction uses one RMQ query in L (which is $O(1)$ according to the assumption) and three array references at $O(1)$ time each. Therefore we have a $\langle O(n), O(1)\rangle$-solution for LCA. □

From now on, we focus only on RMQ solutions. We will consider solutions to the general RMQ problem as well as to an important restricted case suggested by the array L. In array L from the above reduction adjacent elements differ by $+1$ or -1. We obtain this ± 1

[5] In fact, we do not necessarily need the *first* occurrences, but *any* occurrence. But we consider the first occurrences for the sake of concreteness.

restriction because, for any two adjacent elements in an Euler tour, one is always the parent of the other, and so their levels differ by exactly one.[6] Thus, we consider the ± 1 RMQ problem as a special case.

4 RMQ Algorithms

4.1 A Simple Solution for RMQ

RMQ has a solution with complexity $\langle O(n^2), O(1)\rangle$: build a table storing answers to all of the n^2 possible queries. To achieve $O(n^2)$ preprocessing rather than the $O(n^3)$ naive preprocessing, we apply a trivial dynamic program. Notice that answering an RMQ query now requires just one array lookup.

4.2 A Faster RMQ Algorithm

It is now shown that a more intelligent dynamic programming leads to a $\langle O(n \log n), O(1)\rangle$-solution for (general) RMQ.

The idea is to precompute each query whose length is a power of two. That is, for every i between 1 and n and every j between 1 and $\log n$, we find the minimum element in the block starting at i and having length 2^j, that is, we compute $M[i, j] = m$, such that $A[m] = \min A[i, \ldots, i + 2^j - 1]$. Table M therefore has size $O(n \log n)$, and we fill it in time $O(n \log n)$ by using dynamic programming. Specifically, we find the minimum in a block of size 2^j by comparing the two minima of its two constituent blocks of size 2^{j-1}. More formally, $M[i, j] = M[i, j - 1]$ if $A[M[i, j - 1]] \leq A[M[i + 2^{j-1}, j - 1]]$ and $M[i, j] = M[i + 2^{j-1}, j - 1]$ otherwise.

How do we use these blocks to compute an arbitrary $\textsc{rmq}(i, j)$? We select two overlapping blocks that entirely cover the subrange: let 2^k be the size of the largest block that fits into the range from i to j, that is let $k = \lfloor \log(j - i) \rfloor$. Then $\textsc{rmq}(i, j)$ can be computed by comparing the minima of the following two blocks: i to $i + 2^k - 1$ ($M(i, k)$) and $j - 2^k + 1$ to j ($M(j - 2^k + 1, k)$). These values have already been computed, so we can find the RMQ in constant time. Figure 3 shows an example for $i = 10$ and $j = 15$.

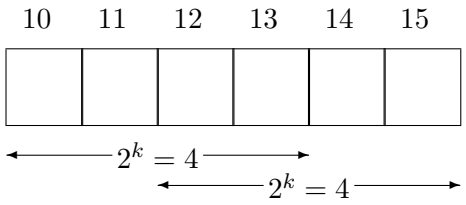

Figure 3: An example how overlapping blocks can be used

This gives the *Sparse Table (ST)* algorithm for RMQ, with complexity $\langle O(n \log n), O(1)\rangle$, if we regard the log-operation together with the floor-operation as a constant time operation. These can be seen together as the problem of finding the most significant bit of a word.

[6] Consider, for example, again Table 2.

Below, we will use the ST algorithm to build an even faster algorithm for the ± 1 RMQ problem.

4.3 An $\langle O(n), O(1)\rangle$-Algorithm for ± 1 RMQ

Suppose we have an array A with the ± 1 restriction. We will use a table-lookup technique to precompute answers on small subarrays, thus removing the log factor from the preprocessing. To this end, partition the array A into blocks of size $\frac{\log n}{2}$. Define an array $A'[1, \ldots, 2n/\log n]$, where $A'[i]$ is the minimum element in the ith block of A. Define an equal size array B, where $B[i]$ is a position in the ith block in which value $A'[i]$ occurs. Recall that RMQ queries return the position of the minimum and that the LCA to RMQ reduction uses the position of the minimum, rather than the minimum itself. Thus we will use array B to keep track of where the minima in A' came from.

The ST algorithm runs on array A' in time $\langle O(n), O(1)\rangle$. Having preprocessed A' for RMQ, consider how we answer any query $\text{RMQ}(i, j)$ in A. The indices i and j might be in the same block, so we have to preprocess each block to answer RMQ queries. If $i < j$ are in different blocks, we can answer the query $\text{RMQ}(i, j)$ as follows. First compute the values:

1. The minimum from i forward to the end of its block.
2. The minimum of all the blocks in between (between i's block and j's block).
3. The minimum from the beginning of j's block to j.

The query will return the position of the minimum of the three values computed. The second minimum is found in constant time by an RMQ on A', which has been preprocessed using the ST algorithm. But, we need to know how to answer range minimum queries inside blocks to compute the first and third minima, and thus to finish off the algorithm. Thus, the in-block queries are needed whether i and j are in the same block or not.

Therefore, we focus now only on in-block RMQs. If we simply performed RMQ preprocessing on each block, we would spend too much time in preprocessing. If two blocks were identical, then we could share their preprocessing. However, it is too much to hope for that blocks would be so repeated. The following observation establishes a much stronger shared-preprocessing property.

Observation 4.1 *If two arrays $X[1, \ldots, k]$ and $Y[1, \ldots, k]$ differ by some fixed value at each position, that is, there is a c such that $X[i] = Y[i] + c$ for every i, then all RMQ answers will be the same for X and Y. In this case, we can use the same preprocessing for both arrays.*

Thus, we can *normalize* a block by subtracting its initial offset from every element. The ± 1 property is now used to show that there are very few kinds of normalized blocks.

Lemma 4.2 *There are $O(\sqrt{n})$ kinds of normalized blocks.*

Proof: Adjacent elements in normalized blocks differ by $+1$ or -1. Thus, normalized blocks are specified by a ± 1 vector of length $\frac{\log n}{2} - 1$. There are $2^{\frac{\log n}{2}-1} \in O(\sqrt{n})$ such vectors. $\square$

We are now basically done. We create $O(\sqrt{n})$ tables, one for each possible normalized block. In each table, we put all $\left(\frac{\log n}{2}\right)^2 \in O(\log^2 n)$ answers to all in-block queries.[7] This gives a total of $O(\sqrt{n}\log^2 n)$ total preprocessing of normalized block tables, and $O(1)$ query time. Finally, compute, for each block in A, which normalized block table it should use for its RMQ queries. Thus, each in-block RMQ query takes a single table-lookup.

Overall, the total space and preprocessing used for normalized block tables and A' tables is $O(n)$ and the total query time is $O(1)$.

5 Summary

We started out by a reduction from the LCA problem to the RMQ problem, but with the key observation that the reduction actually leads to a ± 1 RMQ problem.

A trivial $\langle O(n^2), O(1)\rangle$ table-lookup algorithm for RMQ was given, and it was shown how to sparsify the table to get a $\langle O(n\log n), O(1)\rangle$ table-lookup algorithm. We used this latter algorithm on a smaller summary array A' and needed only to process small blocks to finish the algorithm. Finally we noticed that most of these blocks are the same, from the point of view of the RMQ problem, by using the ± 1 assumption given by the original reduction.

6 The End of the Story?

In section 3 it was shown that the LCA problem can be solved by solving a restricted case of RMQ. But the relationship between these two problems is even stronger. Actually, the authors of [Ben00] showed that the general RMQ problem can be solved by reducing to the LCA problem, i. e. by building up a tree! This method leads to an $\langle O(n), O(1)\rangle$ algorithm for general RMQ as well. (See appendix for details of this construction.) Thus, to solve a general RMQ problem, one would convert it to an LCA problem and then back to a ± 1 RMQ problem.

Therefore, the LCA problem and the RMQ problem are in some sense equivalent.

A Appendix: A Fast Algorithm for RMQ

It is now shown that the general RMQ problem can be reduced to LCA:

Theorem A.1 *If there is an $\langle O(n), O(1)\rangle$-solution for LCA (and there is!), then there is an $\langle O(n), O(1)\rangle$-solution for general RMQ.*

Proof: Let $A[1, \ldots, n]$ be the input array. It is shown that the $O(n)$ term in the preprocessing comes from the time needed to build the Cartesian Tree of A and the $O(1)$ term in the query comes from the time needed to convert the LCA answer on this tree to an RMQ answer on A.

The Cartesian Tree of an array is defined as follows. The root of a Cartesian Tee is the minimum element of A, and the root is labeled with the position of this minimum. Removing

[7] Note that the simple algorithm from 4.1 could be used here.

the root element from A splits the array into two pieces. The left and right children of the root are the recursively constructed Cartesian trees of the left and right subarrays, respectively.

A Cartesian Tree can be built in linear time as follows. Suppose C_i is the Cartesian tree of $A[1, \ldots, i]$. To build C_{i+1}, we notice that node $i+1$ will belong to the rightmost path of C_{i+1}, so we climb up the rightmost path of C_i until finding the position where $i+1$ belongs. Each comparison either adds an element to the rightmost path or removes one, and each node can only join the rightmost path and leave it once. Thus the total time to build C_n is $O(n)$.

The proprocessing for A for RMQ is as follows.

- Construct the Cartesian Tree C of A (in linear time).
- Preprocess C for LCA (in linear time, as seen before).

It is now claimed that for every query: $\textsc{rmq}_A(i, j) = \textsc{lca}_C(i, j)$.

To see this, consider the least common ancestor, k, of i and j in the Cartesian Tree C. In the recursive description of a Cartesian Tree, k is the first node that separates i and j. Thus, in the array A, element $A[k]$ is between elements $A[i]$ and $A[j]$. Furthermore, $A[k]$ must be the smallest such element in the subarray $A[i, \ldots, j]$ since otherwise, there would be a smaller element k' in $A[i, \ldots, j]$ that would be an ancestor of k in C, and k and j would already have been separated by k'. □

References

[Tar84] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338-355, 1984.

[Sch88] B. Schieber and U. Vishkin. On finding lowest common ancestors. Simplification and parallelization. *SIAM Journal on Computing*, 17:1253-1262, 1988.

[Ber89] O. Berkman, D. Breslauer, Z. Galil, B. Schieber and U. Vishkin. Highly parallelizable problems. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing*, pages 309-319, 1989.

[Gus97] Dan Gusfield: Algorithms on strings, trees, and sequences: Computer science and computational biology. Cambridge University Press: Cambridge, 1997.

[Ben00] M. A. Bender, M. Farach-Colton. The LCA Problem Revisited. `http://www.cs.sunysb.edu/~bender/pub/lca.ps`

[Als02] S. Alstrup and C. Gavoille and H. Kaplan and T. Rauhe: Nearest common ancestors: A survey and a new distributed algorithm. `http://www.it-c.dk/research/algorithms/Kurser/AD/2002E/Uge6/nca.pdf`