Lowest Common Ancestor(LCA)

Fayssal El Moufatich

Technische Universität München
St. Petersburg

JASS 2008

1 Introduction

LCA problem is one of the most fundamental algorithmic problems on trees. It is concerned with how we can find the **Least Common Ancestor** of a pair of nodes. Over the last 3 decades, it has been intensively studied mainy because:

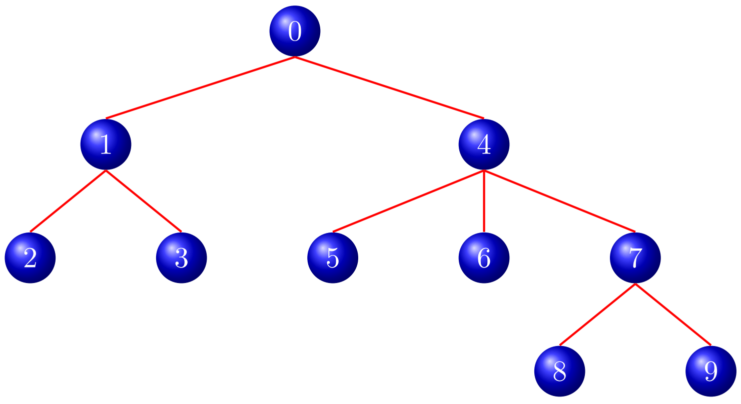
- It is inherently algorithmically beautiful.
- Fast algorithms for the LCA problem can be used to solve other algorithmic problems.

2 Definitions

Before embarking into the technicalities of the different algorithms for LCA, let us first agree on the terminology that we will be using throughout this article.
Let there be a rooted tree T(E,V). A node x ∈ T is called an ancestor of a node y ∈ T if the path from the root of T to y goes through x. Also, a node v ∈ T is called to be a common ancestor of x and y if it is an ancestor of both x and y. Hence, the Nearest/Lowest Common Ancestor, **NCA or LCA**, of two nodes x, y is the common ancestor of x and y whose distance to x (and to y) smaller than the distance to x of any common ancestor of x and y. From now on, we denote the NCA of two nodes x and y from the tree T as $nca(x, y)$. As we have already pointed out, efficiently computing NCAs has been studied extensively for the last 3 decades in both *online* and *offline* settings.

3 Example

Here follows an example for a tree. Choosing two arbitrary nodes from the tree, one is interested in finding their lowest common ancestor.



For example, the $nca(2, 5) = 0$, $nca(7, 5) = 4$, and $nca(0, 9) = 0$.

4 Applications

A procedure solving the NCA problem has been widely used by algorithms from a large spectrum of applications. To point out, LCA algorithms have been used in finding the *maximum weighted matching* in a graph, finding a *minimum spanning tree* in a graph, finding a *dominator tree* in a graph in a *directed flow-graph*, several string algorithms, *dynamic planarity testing*, *network routing*,solving various geometric problems including *range searching*, finding *evolutionary trees*, and in *bounded tree-width* algorithms as well as in many other fields.

5 Survey of Algorithms

One of the most fundamental results on computing NCAs is that of Harel and Tarjan [5], [3]. They describe a linear time algorithm to preprocess a tree and build a data structure that allows subsequent NCA queries to be answered in **constant time!**. Several simpler algorithms with essentially the same properties but better constant factors were proposed afterwards. They all use the observation that it is rather easy to solve the problem when the input tree is a complete binary tree.We will be having a short view on the properties of the Harel and Tarjan algorithm, and then we will be considering a simpler algorithm that was presented later by Farach and Bender.

6 How do we do it?

We have said in the previous section that finding the LCA of arbitrary two nodes from a completely balanced binary tree is rather simple. We proceed by first labeling the nodes by their index in an *inorder* traversal of the complete binary tree. It follows that if the tree has n nodes, each such number occupies $\ell = \lfloor \log(n) \rfloor$ bits. We assume that the LSB is the rightmost and that its index is 0. Let denote $inorder(x)$ and $inorder(y)$ to be the inorder indexes of x and y. Then, let $i = max((1), (2), (3))$ where:

1. index of the leftmost bit in which $inorder(x)$ and $inorder(y)$ differ.
2. index of the rightmost 1 in $inorder(x)$.
3. index of the rightmost 1 in $inorder(y)$.

It turns out that it can be proved by induction that:

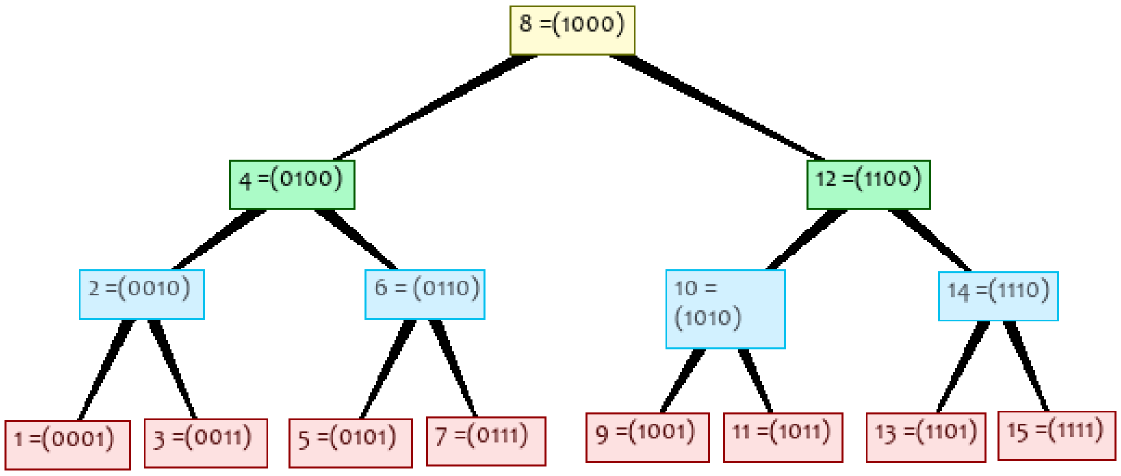
Lemma 1. *[2]the $inorder(nca(x, y))$ consists of the leftmost $\ell - i$ bits of $inorder(x)$ (or $inorder(y)$ if the max was (3)) followed by a 1 and i zeros.*

An example that illustrates the meaning of the above lemma follows.

7 Example

Before getting to the example illustrated in Figure1, it might be worth noting here that in this setting, the basic idea is construct the $inorder(nca(x, y))$ from $inorder(x)$ and $inorder(y)$ alone and without accessing the original tree or any other global data structure. This basically means that we are answering the LCA query in constant time!
Now considering the tree depicted in Figure1, let us choose to arbitrary nodes in the tree. Let us say that we choose 6 and 9, i.e. 0110 and 1001. The position of the different bit from left is 3, the index of the first 1 in 6 from right is 1 and the index of the first 1 in 9 from right is 0. Hence max(3,1,0)= 3. Hence 1000 is the lowest common ancestor of 6 and 9.

Figure 1: Inorder labeling for a completely balanced binary tree [6]



8 So what if the input tree is not a completely balanced binary tree?

In such case, one can simply do a mapping to a completely binary balanced tree. It turns out that different algorithms differ by the way they do this mapping. However, all algorithms have to use some precomputed auxilliary data structures and the labels of the nodes to compute the NCAS due the usage of the completely balanced binary tree mapping. Unfortunately, most of the algorithms for general trees do not allow to compute a unique identifier of nca(x,y) from short labels associated with x and y alone. However, one can prove the following interesting theorem:

Theorem 2. *There is a linear time algorithm that labels the n nodes of a rooted tree T with labels of length $\mathcal{O}(\log n)$ bits such that from the labels of nodes x, y in T alone, one can compute the label of nca(x, y) in constant time.*

here follows a sketch for the proof of the above theorem. However by no means this sketch is inclusive. For complete details, please refer to the related paper of Alstrup et al[**?**].

Proof. [4]

- Use lexigraphic sorting the sequence of intergers or binary strings.
- Use results from Gilbert and Moore on alphabetic coding of sequences of integers $\langle b \rangle_k (|b_i| < \log n - \log y_i + \mathcal{O}(1)$ for all i).
- use labeling along **HPs, Heavy Paths**.

□

9 NCA and Discrete Range Searching (DRS)

Gabow, Bentley and Tarjan[9] nicely observed that one-dimensional DRS problem is equivalent to NCA problem.Hence, DRS is used by most of simple NCA algorithms. here follows a definition for the *DRS problem*:

Definition 3. DRS Problem Given a sequence of real numbers x_1, x_2, $...x_n$, preprocess the sequence so that one can answer efficiently subsequent queries of the form:

given a pair of indices (i, j), what is the maximum element among x_i,...,x_j or $max(i, j)$.

DRS problem is a fundamental geometric searching problem. It turns out that DRS can be reduced to NCA by constructing a **Cartesian tree** for the sequence x_1, ..., x_n [9].

10 What is a Cartesian tree?

Definition 4. Cartesian Tree The Cartesian tree of the sequence $x_1, ..., x_n$ is a **binary tree** with n nodes each containing a number x_i and the following properties:
Let $x_j = max(x_1, ..., x_n)$

1. The root of the Cartesian tree contains x_j.
2. The left subtree of the root is a Cartesian tree for $x_1, ..., x_{j-1}$.
3. The right subtree of the root is a Cartesian tree for $x_{j+1}, ..., x_n$.

The Cartesian tree for $x_1, ..., x_n$ can be constructed in $\mathcal{O}(n)$ [Vuillemin, 1980]. The algorithm is rather simple and should be left to the curious reader to tinker with. As such, the maximum among $x_i, ..., x_j$ namely corresponds to the *NCA* of the node containing x_i and the node containing x_j.

11 What about NCA as DRS?

As stated above Gabow et al. also show how to reduce the NCA problem to the DRS problem. Given a tree, we first construct a sequence of its nodes by doing a depth first traversal. Each time we visit a node, we add it to the end of the sequence so that each node appears in the sequence as many times as its degree.[a prefix of the Euler tour of the tree]. Now, let $depth(x)$ be the depth of a node x. We replace each node x in the sequence by -$depth(x)$. Then, to compute $nca(x, y)$, we pick arbirary 2 elements x_i and x_j representing x and y, and compute the maximum among $x_i, ..., x_j$. The node corresponding to the maximum element is simply $nca(x, y)$!

12 Euler tour of a tree

Figure2 shows an example for an Eulerian tour of a tree.

13 What is the LCA of given two nodes then?

After getting the sequence of the Euler tour of the tree, the LCA of given two nodes would be simply the node of the least depth (i.e. Closest to the root) that lies between the nodes in the Euler tour. Hence, finding specific node in the tree is equivalent to finding the minimum element in the proper interval in the array of numbers. The latter problem, it turns out, can be solved by **min-range queries**.

14 Range Minimum Query (RMQ) Problem

This is the same as the DSR problem but outputs the minimum instead.

Definition 5. RMQ Problem **Structure to Preprocess:** an array of numbers of length n. **Query:** for indices i and j and n, query **RMQ(x,y)** returns the index of the smallest element in the subarray A[i...j].

Remark
As with the DSR algorithm, LCA can be reduced to an RMQ problem[1].

Figure3 shows an example for what we mean by the RMQ of two indices in the array of a sequence of numbers.

Figure 2: Euler tour of a tree and the resulting sequence[7]

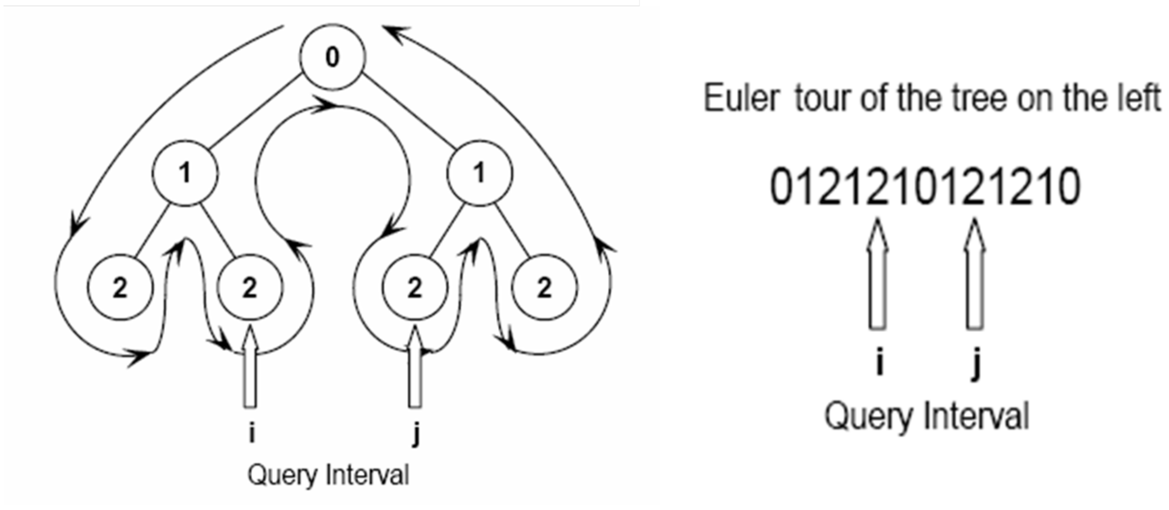
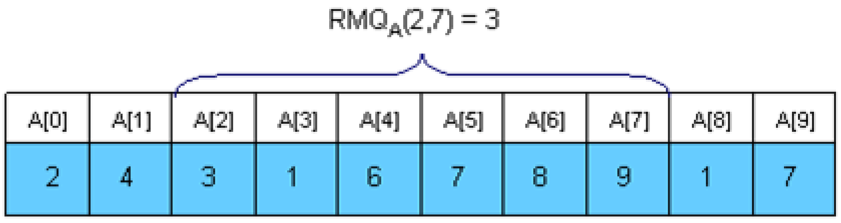


Figure 3: RMQ example for an array of numbers[8]



15 Isn't that a loop in our reduction?

At this point, the reader might be confused by the circular approach of using LCA and RMQ. Mainly, we started by reducing the range-min/DSR problem to an LCA problem, and now we want to solve LCA by reducing it or an RMQ problem. The answer is of course no! The constructed array of numbers has a special property known as ∓ 1 property. Here follows a definition of this property:

Definition 6. ∓ 1 propertyEach number differs by exactly one from its preceding number.

Hence, our reduction is a special case of the range-min query problem that can be solved without further reductions.
Our goal then is to solve the following problem:

Problem
Preprocess an array of n numbers satisfying the ∓ 1 property such that given two indices i and j in the array, determine the index of the minimum element within the given range $[i, j]$, $\mathcal{O}(1)$ time and $\mathcal{O}(n)$ space.

16 Bender-Farach Algorithm for LCA

This is an algorithm that was reengineered from existing complicated LCA algorithms. (PRAM (Parallel Random Access Machine) from Berkman et al.). It reduces the LCA problem to an RMQ problem and considers RMQ solutions rather.

16.1 Naïve Attempt

It turns out that RMQ has a simple solution with complexity $\langle \mathcal{O}(n^2), \mathcal{O}(1) \rangle$:

- Build a lookup table storing answers to all the n^2 possible queries.
- To achieve $\mathcal{O}(n^2)$ preprocessing rather than $\mathcal{O}(n^3)$, we use a trivial dynamic program.

16.2 A Faster RMQ Algorithm

In this case, the idea is to precompute each query whose length is a power of 2. In other words, for each i in $[1, n]$ and every j in $[1, \log n]$, we find the minimum of the block starting at i and having length 2^j. Expressed mathematically, this means:

$$M[i, j] = argmin_{k=i \ldots i+2^j-1} A[k] \qquad (1)$$

Table M has size $\mathcal{O}(n \log n)$. We fill it in using *dynamic programming*, which will illustrated afterwards. We find the minimum in a block of size 2^j by comparing the two minima of its constituent blocks of size 2^{j-1}. Formally speaking, we have:

$$M[i, j] = M[i, j-1] \, if \, A[M[i, j-1]] \leq A[M[i+2^{j-1}, j-1]] \qquad (2)$$

and

$$M[i, j] = M[i + 2^{j-1}, j - 1] \, otherwise \qquad (3)$$

16.3 How do we use blocks to compute an arbitrary RMQ(i,j)?

For this sake, we select 2 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, i.e. $k = \lfloor \log(j - i) \rfloor$ .. Then $RMQ(i, j)$ can be computed by comparing the minima of the 2 blocks:
i to $i + 2^k - 1$ (M(i,k)) and $j - 2^k + 1$ to j ($M(j - 2^k + 1, k)$). The values needed are all already computed values, hence we can find RMQ in constant time!

16.4 Remarks

This gives the **Sparse Table(TS)** algorithm for RMQ with complexity $\langle \mathcal{O}(n \log n), \mathcal{O}(1) \rangle$ ..

The total computation to answer an RMQ query is then 3 additions, 4 array reference and a minimum, and 2 ops: log and floor. This can be seen as problem of finding the MSB of a word. It might be also worth noting that LCA problem is shown to have $\Omega(\log\log n)$on a pointer machine by Harel and Tarjan.

16.5 An $\langle\mathcal{O}(n), \mathcal{O}(1)\rangle$ algorithm for $\mp$RMQ

Now, we want to consider even a faster algorithm for $\mp$RMQ. Suppose we have array A with $\mp$ **restriction**. We use a lookup-table to precompute answers for small subarrays, thus removing log factor from preprocessing. We proceed by partitioning A into blocks of size $\frac{\log n}{2}$. Then, we define an arrayA'[1,...,$\frac{2n}{\log n}$] where A'[i] is the minimum of the ith block of A. We also define an equal size array B where B[i] is a position in the ith block in which A[i] occurs. B is used to keep track of where the minima of A came from.
The Sparse Table algorithm runs on A in time $\langle\mathcal{O}(n), \mathcal{O}(1)\rangle$. Now, consider RMQ(i,j) in A:

- i and j can be in same block in which case we process each block to answer RMQ queries.
- in case $i < j$, then we consider the following minimums:
  - Minimum from i forward to end of its block.
  - Minimum of all blocks btw. is block and js block.
  - Minimum from beginning of js block to j.

We can see the second minimum can be found in constant time by RMQ on A since we already have in the our table.

16.6 How to answer range RMQ queries inside blocks?

It turns out that in-block queries would be needed for both the first and third values to complete algorithm. Unfortunately RMQ processing on each block would result in too much time in processing. Also, one can notice that in case we 2 blocks identical, we could share their processing, but it seems we would have too much hope that blocks would be so repeated! As a result, a different observation would be needed:

Observation
If two arrays X[1,...,k] and Y[1,...,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 Xand Y.

Given the above observation, we then proceed by normalizing a block by subtracting its initial offset from every element. Then, we use the $\mp$1 property to show there are actually very few kinds of normlized blocks. For such sake, we have the following to the rescue:

Lemma 7. *There are $\mathcal{O}(\sqrt{n})$ kinds of normalized blocks*

Proof. Adjacent elements in normalized blocks differ by +1 or -1. Thus, normalized blocks are specified by $\mp$1 vector of length $\frac{1}{2\log n} -$ 1. There are $2^{\frac{1}{2\log n}-1}$ =$\mathcal{O}(\sqrt{n})$ such vectors [1] □

At this stage, we are basically done! We Create $\mathcal{O}(\sqrt{n})$ tables, one for each possible normalized block. A total of $\mathcal{O}(\sqrt{n})\log^2 n$ total processing of normalized block tables and $O(1)$ query time would be required. Finally, we compute for each block in A which normalized block table it should use for its RMQ queries.

16.7 Wrapping up!

So we started by reducing from LCA problem to RMQ problem given reduction leads to $\mp$1RMQ problem. We gave a trivial $\langle\mathcal{O}(n^2), \mathcal{O}(1)\rangle$ time table-lookup algorithm for RMQ and show how to sparsify the table to get $\langle\mathcal{O}(n\log n), \mathcal{O}(1)\rangle$-time table-lookup algorithm. Then, we used the latter algorithm on a smaller summary array A, and then we needed only to process small blocks to finish algorithm. Finally, we noticed that most of these blocks are the same by using the $\mp$1 assumption from original reduction.(from RMQ problem point of view).

17 A Fast Algorithm for RMQ!

By now, We have $\langle\mathcal{O}(n), \mathcal{O}(1)\rangle$ $\mp$RMQ. Our goal is to the general RMQ in the same complexity. It turns out that this can be achieved by reducing the RMQ problem to an LCA problem again. Hence, to solve a general RMQ problem, one would convert it to an LCA problem and then back to $\mp$1RMQ problem.

17.1 How?

We use the results of the following lemma:

Lemma
If there is a $\langle\mathcal{O}(n), \mathcal{O}(1)\rangle$ solution for LCA, then there is a $\langle\mathcal{O}(n), \mathcal{O}(1)\rangle$ solution for RMQ.

Here $\mathcal{O}(n)$ comes from time needed to build Cartesian Tree C of A and $\mathcal{O}(1)$ comes from time needed to convert LCA to an RMQ answer on A. We can prove that:

$$RMQ_A(i,j) = LCA_C(i,j) \tag{4}$$

. By this, our **Reduction would be completed!**

18 Final Remarks

As we have seen in this article, we can solve the range-min query problem in an array of n numbers with $\mp$1 property in $\mathcal{O}(1)$ query time and $\mathcal{O}(n)$ preprocessing time. It turns out that we can extend this to $(O)(n)$ space as well. We proceed by divide the array A into m=$\frac{2n}{\log n}$ buckets, each of size k=$\frac{\log n}{2}$.
It might be worth noting that there several parallel and distributed versions for the LCA algorithm, and that might also be one the reasons for the popularity of this algorithms in other fields.

References

[1] Michael A. Bender and Martin Farach-Colton. *The LCA Problem Revisited.* 2000.

[2] B. Schieber and U. Vishkin. *On finding lowest common ancestors: Simplification and parallelization.* SIAM J. Comput., 17:12531262, 1988.

[3] D. Harel and R. E. Tarjan. *Fast algorithms for finding nearest common ancestors.* SIAM J. Comput.,13(2):338355, 1984.

[4] Stephen Alstrup, et al. *Nearest Common Ancestors: A Survey and a new Distributed Algorithm.* 2002.

[5] D. Harel. *A linear time algorithm for the lowest common ancestors problem.* In 21st Annual Symposium On Foundations of Computer Science, pages 308-319, 1980.

[6] Mohamed Ahsan Yusuf Eida. *Lowest Common Ancestor.* Web Link

[7] Erik Demaine. *Advanced Data Structures: Lecture 11.* 2003.

[8] Daniel P. *Range Minimum Query and Lowest Common Ancestor.* Web Link

[9] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. *Data structure for weighted matching and nearest common acestors with linking.* In 1st Annual ACM Symposium on Discrete Algorithms, pages 434-443, 1984.