Algorithms Analysis
Lowest Common Ancestor

Dorin-Mihai Manea

Faculty of Automatic Control and Computer Science
Politehnica University Bucharest
dorin.mihai17@gmail.com

Abstract. The paper studies the Lowest Common Ancestor (LCA) problem, in ontologies also known as the Least Common Ancestor. Furthermore, it carries an in depth analysis of the time-space complexities of two proposed algorithms for it, based on a set of specifically designed generated tests.

Keywords: Lowest Common Ancestor · Least Common Ancestor · LCA · Tarjan's off-line algorithm · off-line algorithm · Farach-Colton and Bender algorithm · RMQ · Range Minimum Querry

1 Introduction

1.1 Problem Description

Let T be a tree with N nodes, indexed from 0 to $N-1$. Additionally, there are M queries of the form (u, v) to be made. Each query will return the **Lowest Common Ancestor (LCA)** w of the nodes u and v, i.e. the node with the greatest depth in the tree that is an ancestor for both u and v. If u is the ancestor of v, then u is their LCA.

Particular interest arises whether $M \ll N$, $M \approx N$ or $M \gg N$, since these are the variables that directly impact the time complexity. The structure of the graph will also massively influence the performance of the chosen algorithms to be tested: Farach-Colton and Bender algorithm and Tarjan's off-line LCA algorithm.

1.2 Real-World Applications

An immediate use for LCA is determining the distance between two nodes in a tree (i.e. the number of edges in the shortest path connecting them):

$$d(u, w) = d(R, u) + d(R, v) - 2 * d(R, LCA),$$

where R stands for root, u, v are the two given nodes, and $d(x, y)$ denotes the distance.

One of the fundamental principles of Object-Oriented Programming (OOP) languages is inheritance, that is when a class derives from another class. The child class will inherit all the public and protected properties and methods from the parent class. Instances of classes are called objects and LCA is the natural way to resolve object dependence during the stages of compilation and execution [1].

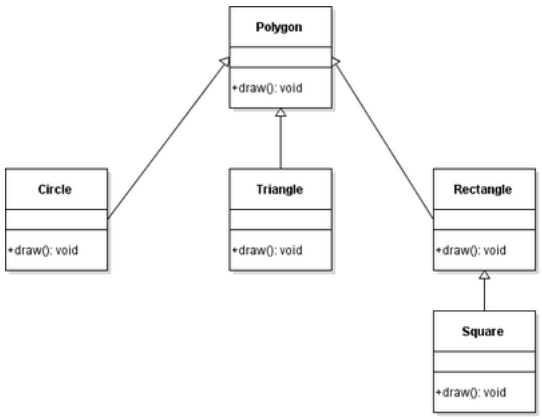

Fig. 1. OOP Inheritance [7]

A lattice is a partially ordered set (poset) in which every pair of elements has both a least upper bound and a greatest lower bound [5]. Algorithms on lattices are used to model the dynamic and static behavior of complex systems arising in distributed computing. LCA queries arise in computing the covering of maximal ideal lattices [1].

1.3 Selected solutions

Farach-Colton and Bender Algorithm

Definition 1. *Let A be a length n array of numbers. The* ***Range Minimum Query (RMQ)*** *represents the smallest element in the array.*

If an algorithm has preprocessing time $f(n)$ and query time $g(n)$, we will say that the algorithm has complexity $(f(n),\ g(n))$.

Lemma 1. *If there is an $(f(n),\ g(n))$ time solution for RMQ, then there is an $(f(2n-1)+\mathcal{O}(n),\ g(2n-1)+\mathcal{O}(1))$ time solution for LCA [1]*

Therefore, we reduce the LCA problem to the RMQ problem. We traverse all nodes of the tree with DFS and keep an array with all visited nodes and the heights of these nodes. The LCA of two nodes u and v is the node between the occurences of u and v in the tour, that has the smallest height [3].

It is worth noting that the reduced RMQ problem has a very specific form: any two adjacent elements in the array differ by exactly one (since the elements of the array are the heights of the nodes visited in order of traversal, and we either go to a descendant, in which case the next element is one larger, or go back to an ancestor, in which case the next element is one smaller). The Farach-Colton and Bender algorithm describes a solution for this specialized RMQ problem [3].

It is asymptotically optimal. We solve the given RMQ queries in $\mathcal{O}(1)$ time, while still taking only $\mathcal{O}(N)$ time for preprocessing. [3].

Tarjan's off-line LCA Algorithm The problem is solved offline, meaning that all queries are known beforehand and can therefore be answered in any order. Using this algorithm, all M queries can be answered in $\mathcal{O}(N+M)$ total time, or in $\mathcal{O}(1)$ time per query for sufficiently large M. The algorithm works by performing a single Depth-First Search (DFS) traversal of the tree, and answering a query (u, v) at node u if node v has already been visited [3].

For a fixed v, the visited nodes of the tree can be split into a set of disjoint sets, each of which is associated with an ancestor p of node v. Each set contains node v and all subtrees rooted in the children of p that are not on the path from v to the root of the tree. The set containing node u determines the LCA of u and v: the LCA is the representative of the set, which is the node on the path between v and the root of the tree. To efficiently maintain these sets, the data structure known as a Disjoint Set Union (DSU) can be used [3].

1.4 Evaluation Criteria

20-30 tests will be generated, both manually and randomly, to test for small and big input data. Also, I will take into consideration how the sizes of M and N compare to each other and affect the final result. Finally, I will look into whether the tree is binary or general.

2 Proposed Solutions

2.1 Farach-Colton and Bender Algorithm

Solution Analysis & Complexity [3]

Proposition 1. *The LCA of nodes u and v is the shallowest node encountered between the visits to u and to v during a DFS traversal of T.*

Definition 2. *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 traversed 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.

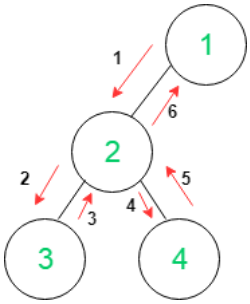

Fig. 2. Input [6]
Output:
Euler: 1 2 3 2 4 2 1
Level: 0 1 2 1 2 1 0

Proof. [3]
Therefore, the reduction to a RMQ problem proceeds as follows:

1. Let array $E[1, \ldots, 2n-1]$ store the nodes traversed in an Euler Tour of the tree T. That is, $E[i]$ is the label of the i-th node traversed in the Euler tour of T;
2. Let the level of a node be its distance from the root. Compute the Level Array $L[1, \ldots, 2n-1]$, where $L[i]$ is the level of node $E[i]$ of the Euler Tour;
3. Let the representative of a node in an Euler tour be the index of first occurrence of the node in the tour; formally, the representative of i is $min\{j : E[j] = i\}$. Compute the Representative Array $R[1, \ldots, n]$, where $R[i]$ is the representative of node i.

Each of these three steps takes $\Theta(n)$ time, yielding $\Theta(n)$ total time. To compute $LCA_T(x, y)$, we note the following:

- 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 $RMQ_L(R[u], R[v])$, 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;
- The node at this position is $E[RMQ_L(R[u], R[v])]$, which is thus the output of $LCA_T(u, v)$.

Thus, we can complete our reduction by preprocessing Level Array L for RMQ. As promised, L is an array of size $2n-1$, and building it takes time $\Theta(n)$. Thus, the total preprocessing is $f(2n-1) + \Theta(n)$. To calculate the query time observe that an LCA query in this reduction uses one RMQ query in L and three array references at $\Theta(1)$ time each. The query thus takes time $g(2n-1) + \Theta(1)$, and we have completed the proof of the reduction.

From now on, we focus only on RMQ solutions. In array L from the above reduction adjacent elements differ by $+1$ or -1. We obtain this ± 1 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. Thus, we consider the $\pm 1 - RMQ$ problem as a special case.

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

We will improve the $(\Theta(n^2),\ \Theta(1))$-time brute-force table algorithm 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 $M[i, j] = min\{A[k] : k = i\ ...\ i + 2^j - 1\}$. Table M therefore has size $\Theta(n\ log\ n)$, and we fill it in time $\Theta(n\ log\ n)$ 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 $A[M[i,\ j-1]] \leq A[M[i+2^{j-1}-1,\ j-1]]$ and $M[i,\ j] = M[i+2^{j-1}-1,\ j-1]$ otherwise.

How do we use these blocks to compute an arbitrary $RMQ(i,\ j)$? 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 $k = [log(j - i + 1)]$. Then $RMQ(i,\ j)$ 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.

This gives the **Sparse Table (ST)** algorithm for RMQ, with complexity $(\Theta(n\ log\ n),\ \Theta(1))$. The table is indexed using a $(distance,\ array\ index)$ tuple. Notice that the total computation to answer an RMQ query involves two subtractions, two 2-dimensional array references, a minimum and a truncated-log operation. The truncated-log operation does not require a table lookup in most modern processors, and can be seen as finding the most significant bit of a word. Notice that we must have at least one array indexing operation in our algorithm, since Harel and Tarjan showed that a pointer-algorithm LCA computation has a lower bound of $\Omega(log\ log\ n)$ operations.

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

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 A into blocks of size $\frac{log\ n}{2}$. (Without loss of generality assume $log\ n$ is even). Define an array $A'[1,\ \ldots,\ \frac{2n}{log\ n}]$, 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 $(\Theta(n),\ \Theta(1))$. Having preprocessed A' for RMQ, consider how we answer any query $RMQ(i,\ j)$ 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, then we can answer the query $RMQ(i,\ j)$ 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. (If i and j are not in the same block prefix minima and suffix minima suffice).

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 block 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.

Proposition 2. *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. We now use the ± 1 property to show that there are few kinds of normalized blocks.

Lemma 2. *There are $\Theta(\sqrt{n})$ 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 $(\frac{1}{2} \cdot log\ n) - 1$. There are $2^{(\frac{1}{2} \cdot log\ n)-1} = \Theta(\sqrt{n})$ such vectors.

We are now basically done. We create $\Theta(\sqrt{n})$ tables, one for each possible normalized block. In each table, we put all $(\frac{log\ n}{2})^2 = \Theta(log^2\ n)$ answers to all in-block queries. This gives a total of $\Theta(\sqrt{n}\ log^2\ n)$ total preprocessing of normalized block tables, and $\Theta(1)$ 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 normalized-block table lookup.

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

Advantages and Disadvantages [8]

Final complexity of this algorithm for our given problem is $\mathcal{O}(N\ log\ N + M)$. The disadvantage of this method is that it uses $\mathcal{O}(N\ log\ N)$ memory, which can be an impediment in certain cases.

2.2 Tarjan's off-line LCA algorithm

Solution Analysis & Complexity [3]

Definition 3. ***DSU*** *or* ***union-find****, as it is popularly called, is a data structure that categorizes objects into different sets and lets checking out if two objects belong to the same set. [9]*

Since this data structure is easy to implement it is extensively used to solve graph connectivity related problem efficiently, e.g. in the Kruskal's algorithm to avoid forming cycles. It provides almost $\mathcal{O}(1)$ time on average to perform operations such as to add new sets, to merge existing sets, and to determine whether elements are in the same set. [9]

A union-find algorithm is an algorithm that performs two useful operations on such a data structure:

- $find\_set(v)$: Returns the *leader* (representative) of the set that contains the element v. This representative is an element of its corresponding set. It is selected in each set by the data structure itself (and can change over time, namely after *union* calls). This representative can be used to check if two elements are part of the same set or not. a and b are exactly in the same set, if $find(a) == find(b)$. Otherwise they are in different sets.

- $union\_sets(a, b)$: Merges the two specified sets (the set in which the element a is located, and the set in which the element b is located). Here first we have to check if the two subsets belong to same set. If no, then we cannot perform union.

Often, it can be equipped with a constructor that organizes every object into its own set.

We will store the sets in the form of *trees*: each tree will correspond to one set. And the root of the tree will be the representative/leader of the set.

In the beginning, every element starts as a single set, therefore each vertex is its own tree. Then we combine the set containing the element 1 and the set containing the element 2. Then we combine the set containing the element 3

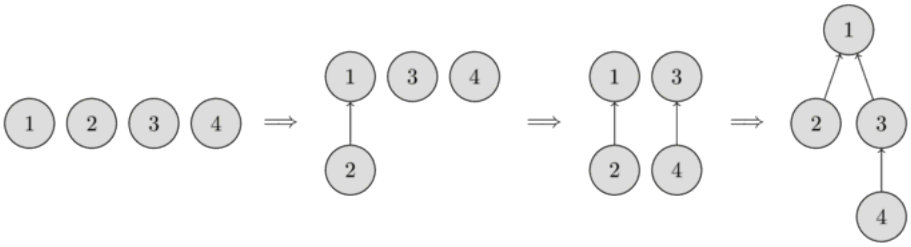

Fig. 3. Tree representations of disjoint sets [3]

and the set containing the element 4. And in the last step, we combine the set containing the element 1 and the set containing the element 3.

For the implementation this means that we will have to maintain an array *parent* that stores a reference to its immediate ancestor in the tree.

We can already write the first implementation of the DSU data structure. It will be pretty inefficient at first, but later we can improve it using two optimizations, so that it will take nearly constant time for each function call.

As we said, all the information about the sets of elements will be kept in an array *parent*.

To create a new set (operation $make\_set(v)$), we simply create a tree with root in the vertex v, meaning that it is its own ancestor.

To combine two sets (operation $union\_sets(a, b)$), we first find the representative of the set in which a is located, and the representative of the set in which b is located. If the representatives are identical, that we have nothing to do, the sets are already merged. Otherwise, we can simply specify that one of the representatives is the parent of the other representative - thereby combining the two trees.

Finally the implementation of the find representative function (operation $find\_set(v)$): we simply climb the ancestors of the vertex v until we reach the root, i.e. a vertex such that the reference to the ancestor leads to itself. This operation is easily implemented recursively.

However this implementation is inefficient. It is easy to construct an example, so that the trees degenerate into long chains. In that case each call $find\_set(v)$ can take $\mathcal{O}(n)$ time.

This is far away from the complexity that we want to have (nearly constant time). Therefore we will consider two optimizations that will allow to significantly accelerate the work.

This optimization is designed for speeding up $find\_set$.

If we call $find\_set(v)$ for some vertex v, we actually find the representative p for all vertices that we visit on the path between v and the actual representative

p. The trick is to make the paths for all those nodes shorter, by setting the parent of each visited vertex directly to p.

You can see the operation in the following image. On the left there is a tree, and on the right side there is the compressed tree after calling $find\_set(7)$, which shortens the paths for the visited nodes 7, 5, 3 and 2.

The simple implementation does what was intended: first find the representative of the set (root vertex), and then in the process of stack unwinding the visited nodes are attached directly to the representative.

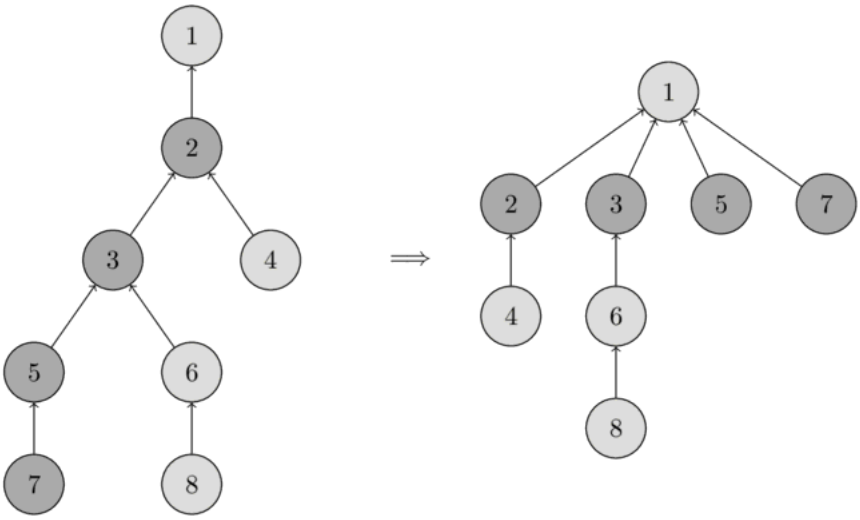

Fig. 4. [3]

This simple modification of the operation already achieves the time complexity $\mathcal{O}(\log n)$ per call on average (here without proof). There is a second modification, that will make it even faster.

In this optimization we will change the *union_set* operation. To be precise, we will change which tree gets attached to the other one. In the naive implementation the second tree always got attached to the first one. In practice that can lead to trees containing chains of length $\mathcal{O}(n)$.With this optimization we will avoid this by choosing very carefully which tree gets attached.

There are many possible heuristics that can be used. Most popular are the following two approaches: In the first approach we use the *size* of the trees as *rank*, and in the second one we use the *depth* of the tree (more precisely, the upper bound on the tree depth, because the depth will get smaller when applying path compression).

In both approaches the essence of the optimization is the same: we attach the tree with the lower rank to the one with the bigger rank. Both optimizations are equivalent in terms of time and space complexity.

As mentioned before, if we combine both optimizations - path compression with union by size / rank - we will reach nearly constant time queries. It turns out, that the final amortized time complexity is $\mathcal{O}(\alpha(n)$, where $\alpha(n)$ is the inverse *Ackermann function*, which grows very slowly. In fact it grows so slowly, that it doesn't exceed 4 for all reasonable n (approximately $n < 10^{600}$).

Amortized complexity is the total time per operation, evaluated over a sequence of multiple operations. The idea is to guarantee the total time of the entire sequence, while allowing single operations to be much slower then the amortized time. E.g. in our case a single call might take $\mathcal{O}(\log n)$ in the worst case, but if we do m such calls back to back we will end up with an average time of $\mathcal{O}(\alpha(n)$.

Also, it's worth mentioning that DSU with union by size / rank, but without path compression works in $\mathcal{O}(\log n)$ time per query.

Both union by rank and union by size require that you store additional data for each set, and maintain these values during each union operation. There exist also a randomized algorithm, that simplifies the union operation a little bit: l*inking by index*.

We assign each set a random value called the *index*, and we attach the set with the smaller index to the one with the larger one. It is likely that a bigger set will have a bigger index than the smaller set, therefore this operation is closely related to union by size. In fact it can be proven, that this operation has the same time complexity as union by size. However in practice it is slightly slower than union by size.

It's a common misconception that just flipping a coin, to decide which set we attach to the other, has the same complexity. However that's not true. Coin-flip linking combined with path compression has complexity $\Omega\left(n\frac{\log n}{\log\log n}\right)$. And in benchmarks it performs a lot worse than union by size/rank or linking by index.

Advantages and Disadvantages [8]

Final complexity of this algorithm for our given problem is $\mathcal{O}(N\ log\ N+M)$. It uses even more memory then our previous RMQ solution: $\mathcal{O}(M)$, which on certain conditions cannot fit the memory limits. Another disadvantage in certain cases is the fact that the queries are not run in order, but are solved offline, that is, it is necessary to know them in advance.

3 Evaluation

3.1 Test Construction

I have manually developed the first 17 tests. The rest are modified tests to meet the input format requirements from source [8]. They take into account the structure of the graph (binary or general), and how the variables N and M size up to each other.

4 Conclusions

We started out by showing a reduction from the Lowest Common Ancestor (LCA) problem to the Range Minimum Query (RMQ) problem, but with the key observation that the reduction actually leads to a $\pm 1RMQ$ problem.

We gave a trivial ($\Theta(n^2)$, $\Theta(1)$)-time table-lookup algorithm for RMQ, and show how to sparsify the table to get a ($\Theta(n\ log\ n)$, $\Theta(1)$)-time 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 notice 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.

In a similar fashion, we have got to the final form of DSU by implementing several optimizations to the Union and Find operations.

References

1. Author, Michael A. Bender, Author, Martín Farach-Colton, et al.: Lowest Common Ancestors in Trees and Directed Acyclic Graphs
2. Author, Thomas H. Cormen et al.: Introduction to Algorithms. 3nd edn. The MIT Press (2009)
3. https://cp-algorithms.com/graph/lca.html. Last accessed 9 Jan 2023
4. https://www.baeldung.com/cs/tree-lowest-common-ancestor. Last accessed 25 Nov 2022
5. https://calcworkshop.com/relations/lattices/. Last accessed 25 Nov 2022
6. https://www.geeksforgeeks.org/euler-tour-tree/. Last accessed 9 Jan 2023
7. https://www.dariawan.com/tutorials/java/inheritance-in-java/. Last accessed 9 Jan 2023
8. https://www.infoarena.ro/problema/lca. Last accessed 9 Jan 2023
9. https://brilliant.org/wiki/disjoint-set-data-structure/. Last accessed 9 Jan 2023