Least Common Ancestor

Xing Chen, Sicong Chen, Xuemeng Zhao

Abstract

This paper explores the Least Common Ancestor (LCA) problem in graph theory, which is essential in theoretical computer science and practical applications. Our paper delves into the naive algorithm and three better algorithms: doubling, Tarjan and RMQ-based. Each algorithm is analyzed for efficiency and suitability in complex data structures. The experiment demonstrates the enhanced query time of these three better algorithms, suggesting their utility in practical applications which require efficient data processing.

1 Introduction

The concept of least common ancestor (LCA) [1] is a key concept in the study of graph theory, especially in relation to tree data structures. The LCA in a tree is defined as the lowest node that is the common ancestor of two given nodes [2]. Not only is this a key element of theoretical computer science, it has far-reaching applications across a variety of industries.
For example, LCA is used in bioinformatics to trace the evolutionary lineage of species, providing important insights into genetic relationships and species evolution. In version control systems, LCA is critical for merge operations and conflict resolution within branch structures, ensuring data integrity and consistency. Other applications include XML document processing (LCA facilitates efficient querying and data retrieval) [3], file systems (LCA plays a key role in managing hierarchical data storage and access) and etc [4].

2 Background

The increasing need for efficient data processing in complex, large-scale systems drives the motivation to explore optimized LCA. As data becomes increasingly larger and more complex, the need for algorithms that can quickly and accurately identify LCA becomes more pressing. This is particularly relevant in fields with hierarchical, large-scale data structures, such as bioinformatics and broadly distributed systems.
Additionally, ever-changing computing challenges require continued research and development in this area. Enhancing the LCA algorithm can significantly improve system performance, data management efficiency, and resource optimization, which is of great significance for practical applications in various industries. Our research aims to contribute to the theoretical and practical advancement of this critical field of computer science by comprehensively studying and implementing various algorithms for LCA.
Through the research and learning of our team, we found that in addition to the most basic

Naive Algorithm to solve the LCA problem, there are three better algorithms to further optimize the implementation of LCA. They are doubling algorithm, Tarjan algorithm, RMQ (range min/max query) based algorithm.

3 Method

3.1 Naive Algorithm

The main idea of find the LCA is to lift up the two nodes and then get their LCA. Here is the pseudo code:

Algorithm 1 algorithm Naive(tree, root, u, v)

```
0: function FINDANCESTORS($node, ancestors$)
0:    $ancestorsList \leftarrow$ empty list
0:    while $node$ is not null do
0:       append $node$ to $ancestorsList$
0:       $node \leftarrow parent[node]$
0:    end while
0:    return $ancestorsList$
0: end function
0: function FINDLCA($u, v$)
0:    $ancestorsU \leftarrow$ FINDANCESTORS($u, ancestorsU$)
0:    $ancestorsV \leftarrow$ FINDANCESTORS($v, ancestorsV$)
0:    $LCA \leftarrow$ null
0:    for $i \leftarrow 0$ to length($ancestorsU$) $-1$ do
0:       for $j \leftarrow 0$ to length($ancestorsV$) $-1$ do
0:          if $ancestorsU[i] == ancestorsV[j]$ then
0:             $LCA \leftarrow ancestorsU[i]$
0:             break
0:          end if
0:       end for
0:    end for
0:    return $LCA$
0: end function=0
```

The time complexity of naive algorithm is $O(N)$. N represents the count of all the nodes. If the size of the tree is so large, this algorithm seems to cost a lot of time to find the LCA.

3.2 Doubling Algorithm

The doubling algorithm is based on the naive algorithm, whose idea is to lift up the two nodes to find their ancestor. But it would be a little bit slow to lift up just one step at one time, and subsequently it is what the doubling algorithm want to solve. The "doubling" means one node will move 2^k steps at one time. [5]
So we need to initialize an ancestor table to contain the ancestral information of each

node. The element $ancestor[u][k]$ of the ancestor table means: if we lift up the node u 2^k steps, which node we will reach. We can use DP(Dynamic Programming) to initialize the ancestor table and here is the recursive formula:

$$ancestor[u][k] = ancestor[ancestor[u][k-1]][k-1] \tag{1}$$

Also, we need to use DFS/BFS to initialize the depth table which contains the depth of each node.
After these preprocessing, we could begin to query the LCA. Here is the pseudo code of doubling algorithm:

Algorithm 2 algorithm Doubling(tree, root, u, v)

```
function PREPROCESS($tree, root$)
  $parent \leftarrow$ array[$nodeCount$][log($nodeCount$) + 1]
  $depth \leftarrow$ array[$nodeCount$]
  $depth[root] \leftarrow 0$
  $parent[root][0] \leftarrow root$
  DFS($tree, root$)
  for $i \leftarrow 1$ to log($nodeCount$) do
    for $node \leftarrow 1$ to $nodeCount$ do
      $parent[node][i] \leftarrow parent[parent[node][i-1]][i-1]$
    end for
  end for
end function
function DFS($tree, node$)
  for each $child$ in $tree[node]$ do
    if $child$ is not visited then
      $depth[child] \leftarrow depth[node] + 1$
      $parent[child][0] \leftarrow node$
      DFS($tree, child$)
    end if
  end for
end function
function FINDLCA($u, v$)
  if $depth[u] < depth[v]$ then
    swap($u, v$)
  end if
  for $i \leftarrow$ log($nodeCount$) downto 0 do
    if $depth[u] - 2^i \geq depth[v]$ then
      $u \leftarrow parent[u][i]$
    end if
  end for
  if $u == v$ then
    return $u$
  end if
  for $i \leftarrow$ log($nodeCount$) downto 0 do
    if $parent[u][i] \neq parent[v][i]$ then
      $u \leftarrow parent[u][i]$
      $v \leftarrow parent[v][i]$
    end if
  end for
  return $parent[u][0]$
end function=0
```

The time complexity for using DFS to save the nodes' depth is $O(N)$, the time complexity for using DP to save ancestral information is $O(N \log N)$ and the time complexity for the LCA query process is $O(\log N)$. Compared with the naive algorithm, it improves the running time.

3.3 Tarjan Algorithm

This algorithm [6] is an excellent example of combining DFS with Union-Find to achieve efficient solutions.
It uses Union-Find data structures to dynamically keep track of the ancestors as the DFS progresses through the tree. Though it is important to note, it operates in an "offline" manner, meaning that it requires all LCA queries to be known in advance before the processing begins.

Algorithm 3 UnionFind Implementation

```
0: function FIND(i)
0:   if parent[$i$] != $i$ then
0:     parent[$i$] = FIND(parent[$i$])
0:   else
0:     return parent[$i$]
0:   end if
0: end function
0: function UNION(x, y)
0:   xRoot = find(x)
0:   yRoot = find(y)
0:   if xRoot != yRoot then
0:     if rank[xRoot] < rank[yRoot] then parent[xRoot] = yRoot;
0:       if rank[xRoot] > rank[yRoot] then parent[yRoot] = xRoot;
0:       elseparent[yRoot] = xRoot; rank[xRoot] += 1;
0:       end if
0:     end if
0:
```

Algorithm 4 DFS and Answer LCA Queries

```
0: int n = // Number of nodes
0: boolean[] visited = new boolean[n];
0: int[] ancestor = new int[n];
0: UnionFind uf = new UnionFind(n);
0: function DFS(node)
0:    visited[node.id] = true;
0:    ancestor[node.id] = node.id;
0:    // Visit all children
0:    for child in node.children do
0:       if !visited[child.id] then
0:          DFS(child);
0:          uf.union(node.id, child.id);
0:          ancestor[uf.find(node.id)] = node.id;
0:       end if
0:    end for// Answer the LCA queries involving this node
0:    for query in queries do
0:       if query.contains(node)  visited[query.otherNode(node)] then
0:          print("LCA of " + query + " is " + ancestor[uf.find(query.otherNode(node).id)]);
0:       end if
0:    end for
0: end function=0
```

This algorithm has a running time complexity of O(N+Q), where N is the number of nodes and Q is the number of LCA queries. DFS traversal provides a linear O(N) component, visiting each node only once. Union-Find improves overall efficiency by processing each query in nearly constant time through path compression.

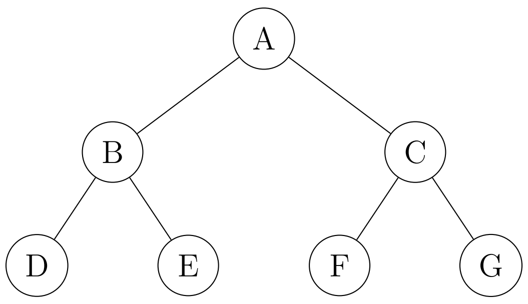
3.4 RMQ Based Algorithm

RMQ is called Range Minimum/Maximum Query. It is used to find the index of the minimum/maximum number in one array. Before introducing RMQ based algorithm [7], we need to verify two definition first. First, sparse table: here is an array A, then let $st[i][j] betheindexoftheminimumelementinA[i...i + 2^(j - 1)]$. We could use DP to initialize the sparse table, and here is the recursive formula:

$$st[i][0] = i \tag{2}$$

$$st[i][j] = \min(st[i][j-1], st[i + 2^{j-1}][j-1]) \tag{3}$$

Second, we need to know Euler Tour Tree (ETT). The Euler Tour of a tree is a technique that converts the tree structure into an array representation while traversing it in a specific way. During this traversal, nodes are visited in a depth-first order, and each node is visited twice: once upon entering (pre-order) and once upon leaving (post-order) the node. For example, ETT of the tree below is [A,B,D,B,E,B,A,C,F,C,G,C,A]



The followings are the whole process of RMQ based algorithm:

Euler Tour Traversal:

Algorithm 5 Euler Tour Traversal

```
0: procedure EulerTour(node, depth)
0:    Add node to eulerTour
0:    Mark node's entry index
0:    for each child in node.children do
0:       EulerTour(child, depth + 1)
0:       Add node to eulerTour
0:    end for
0: end procedure=0
```

Sparse Table Construction:

Algorithm 6 Sparse Table Construction

```
0: procedure BuildSparseTable
0:    Initialize sparseTable
0:    for $i$ from 0 to $n - 1$ do
0:       Initialize sparseTable[$i$][0]
0:    end for
0:    for $j$ from 1 to log_n do
0:       for $i$ from 0 to $n - (1 << j)$ do
0:          Compute sparseTable[$i$][$j$]
0:       end for
0:    end for
0: end procedure=0
```

Least Common Ancestor (LCA) Query:

Algorithm 7 Least Common Ancestor (LCA) Query

```
0: procedure FindLCA(u, v)
0:   if entryIndex[$u$] > entryIndex[$v$] then
0:     Swap($u, v$)
0:   end if
0:   Compute rangeStart and rangeEnd
0:   Compute $k$
0:   if depth[eulerTour[sparseTable[rangeStart][$k$]]] < depth[eulerTour[sparseTable[rangeEnd
     - (1 << $k$) + 1][$k$]]] then
0:     return eulerTour[sparseTable[rangeStart][$k$]]
0:   else
0:     return eulerTour[sparseTable[rangeEnd - (1 << $k$) + 1][$k$]]
0:   end if
0: end procedure=0
```

The time complexity of preprocessing for sparse table and ETT is $O(N \log N)$ and the time complexity for the LCA query is $O(1)$.

4 Experiment

Implementation can be found at https://github.com/Term-inator/LeastCommonAncestors. We applied the above algorithms to different trees. It is shown that RMQ based Algorithm and Tarjan Algorithm is much faster than the other ones.

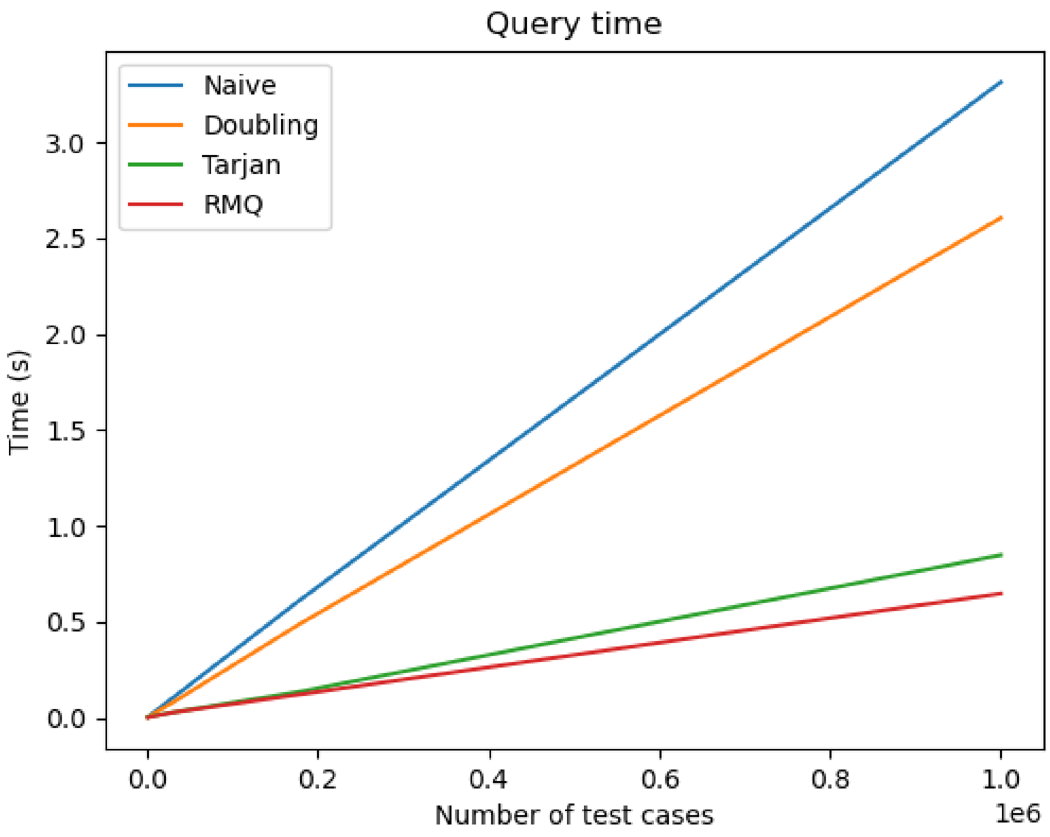


Figure 1: Experiment Result

References

[1] M. J. Atallah and D. Z. Chen, "Chapter 4 - deterministic parallel computational geometry," in *Handbook of Computational Geometry* (J.-R. Sack and J. Urrutia, eds.), pp. 155–200, Amsterdam: North-Holland, 2000.

[2] M. A. Bender and M. Farach-Colton, "The lca problem revisited," in *LATIN 2000: Theoretical Informatics: 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000 Proceedings 4*, pp. 88–94, Springer, 2000.

[3] P. R. Lambole and P. N. Chatur, "A review on xml keyword query processing," in *2017 International Conference on Innovative Mechanisms for Industry Applications (ICIMIA)*, pp. 238–241, 2017.

[4] Y. Wu and C. Fan, "A gene based semantic encoding method for subsumption testing and lca detection," in *2016 IEEE International Conference on Network Infrastructure and Digital Content (IC-NIDC)*, pp. 112–116, 2016.

[5] R. Kapralov, K. Khadiev, J. Mokut, Y. Shen, and M. Yagafarov, "Fast classical and quantum algorithms for online k-server problem on trees," 2020.

[6] B. Schieber and U. Vishkin, "On finding lowest common ancestors: Simplification and parallelization," *SIAM Journal on Computing*, vol. 17, no. 6, pp. 1253–1262, 1988.

[7] M. A. Bender, G. Pemmasani, S. Skiena, and P. Sumazin, "Finding least common ancestors in directed acyclic graphs," 2001.