Implementation of Lowest Common Ancestor to Calculate Extreme Edge's Weight in The Path Between Two Vertex of a Tree

Berto Richardo Togatorop - 13522118[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13522118@std.stei.itb.ac.id

***Abstract*— In a tree, a hierarchical structure commonly used in computer science, the Extreme edge's weight in the path between 2 arbitrary vertices can be determined. This statement gives rise to a crucial problem with broad applications in more complex contexts. There are various solutions to this problem, and the author will discuss the use of Lowest Common Ancestor (LCA) as an approach to solve this problem.**

***Keywords*—Lowest Common Ancestor, Extreme Edge's Weight, Tree.**

I. Introduction

In computer science, trees are essential hierarchical structures that are used extensively due to their computational and organizational efficiency. A crucial issue in the field of tree architectures is figuring out the weight of the minimal edge that connects two random vertices. This challenge has broad practical implications, as it can be applied to optimization issues across multiple fields. For example, in supply chain logistics, finding the path with the least amount of weight at the edge becomes crucial to reducing costs, optimizing efficiency, and expediting the movement of goods between distribution hubs.

Examine the difficulties presented by network optimization in a separate but related scenario. One of the primary concerns of any network is the effective transfer of data, and choosing the best channel has a direct impact on overall performance. Under such circumstances, determining the lowest edge weight between particular vertices in a tree becomes essential to improving network performance. Over and beyond theoretical concerns, the solutions to these kinds of problems have practical applicability in many other domains.

To solve this problem, the Lowest Common Ancestor (LCA) emerges as a powerful tool. LCA provides an elegant solution to the broader problem, offering computational efficiency and scalability. In the subsequent sections, we will explore the application of LCA and its role in resolving the problem of calculating the Extreme edge's weight in the path between two vertices within the context of a tree.

II. Fundamenthal Theorem.

A. Graph

Graph is a data structure utilized to depict relationships among discrete objects. In formal terms, a graph is defined by a tuple (V, E), where V is a non-empty set of vertices, representing vertices (or nodes), and E is a set of edges connecting pairs of vertices.

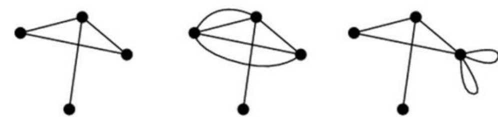
Based on edge orientation, graphs are categorized into two types:

1. Undirected graph

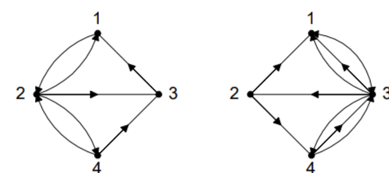
An undirected graph is characterized by edges that do not possess directional orientation.

2. Directed graph or digraph

A directed graph (or digraph) (V , E) consists of a nonempty set of vertices V and a set of directed edges (or arcs) E. Each directed edge is associated with an ordered pair of vertices. The directed edge associated with the ordered pair (u, v) is said to start at u and end at v.



(a) Undirected Graph



(b)Directed Graph

Fig.1 Directed and undirected graph (source: https://informatika.stei.itb.ac.id/~rinaldi.munir/)

A graph can have weight on each edges. This type of graph called weighted graph. The weight on each edges can represent many things based on the need of the structure.

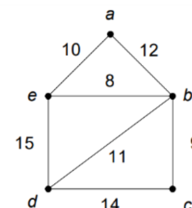


Fig.2 Weighted graph (source:

https://informatika.stei.itb.ac.id/~rinaldi.munir/)

In the realm of graphs, the terms "path" and "cycle" hold significance. A path of length n from the initial vertex v0 to the destination vertex vn within graph G can be defined as an alternating sequence of vertices and edges, denoted as v0, e1, v1, e2, v2, ..., vn-1, en, where each edge, such as e1 = (v0, v1), e2 = (v1, v2), ..., en = (vn-1, vn), represents edges in the graph G. On the other hand, a cycle is a specific type of path that commences and concludes at the same vertex, forming a closed loop within the graph and creating a continuous pattern that returns to the initial vertex. Look at Fig. 2. The sequence a-b-c forms a path, while a-b-e-a forms a cycle.

B. Tree

A tree is an undirected connected graph without cycles. The formal definition can be observed in the following theorem. Suppose G = (V, E) is a simple undirected graph with n vertices. Then, the following statements are equivalent:

1. G is a tree.
2. Every pair of vertices in G is connected by a unique path.
3. G is connected, and it has m = n – 1 edges.
4. G does not contain cycles, and it has m = n – 1 edges.
5. G does not contain cycles, and adding one edge to the graph will create exactly one cycle.
6. G is connected, and all its edges are bridges.

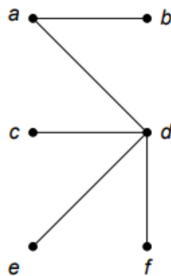


Fig.3 Tree (source: https://informatika.stei.itb.ac.id/~rinaldi.munir/)

C. Rooted Tree

A rooted tree is characterized by designating a specific vertex as the "root" and assigning directional edges to form a directed graph. As a convention, due to the commencement at the root, the directional information is typically omitted in visual representations for simplicity.

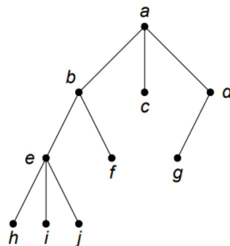


Fig.4 Rooted Tree (source: https://informatika.stei.itb.ac.id/~rinaldi.munir/)

There are some terminology in Rooted Trees :

1. Child and Parent:

In the context of rooted trees, a "child" refers to a node directly connected to another node (the parent) in the direction away from the root.

2. Path:

A "path" in a tree signifies a sequence of nodes where each consecutive pair is connected by an edge. The length of a path is the number of edges it contains.

3. Sibling:

"Siblings" are nodes that share the same parent in a rooted tree.

4. Subtree:

A "subtree" of a node includes that node and all its descendants, forming a smaller tree within the overall tree structure.

5. Degree:

The "degree" of a node refers to the number of subtree (or children) it has.

6. Leaf:

A "leaf" is a node with a degree of zero, signifying that it has no children. Leaves are often referred to as terminal nodes.

7. Internal Nodes:

"Internal nodes" are nodes with one or more children in a rooted tree. They are not leaves, as they have descendants.

8. Level:

The "level" of a node is its distance from the root, with the root itself considered at level 0.

9. Height:

The "height" or "depth" of a tree is the length of the longest path from the root to a leaf. It represents the maximum level in the tree.

D. Lowest Common Ancestor

Lowest Common Ancestor (LCA) is a fundamental concept in graph theory, particularly applicable to rooted trees. It refers to the node that represents the closest shared ancestor of two given nodes in the tree structure. The LCA provides insight into the relationship and connectivity between nodes in a tree, aiding in various algorithms and problem-solving scenarios.

In a rooted tree, the LCA is determined based on the paths from the root to the respective nodes. The LCA of nodes u and v is the deepest node that is a common ancestor of both u and v. This concept is essential for understanding relationships among nodes, calculating distances, and optimizing tree-based algorithms.

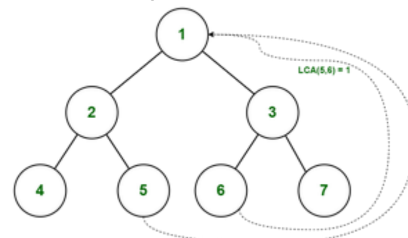


Fig.5 Lowest Common Ancestor in a Tree (source : https://www.geeksforgeeks.org/lowest-common-ancestor-binary-tree-set-1/)

Calculating the LCA can be reached by traversing the tree

efficiently, and numerous algorithms have been developed for this purpose. These algorithms ensure the effective computation of the LCA with reasonable time complexity, enhancing its utility as a valuable tool in solving problems related to tree structures.

Understanding the LCA in rooted trees lays the groundwork for resolving diverse computational challenges. From calculating paths to manipulating subtrees and assessing connectivity within the tree structure, the applications of the LCA reveal its importance in unraveling complex relationships and hierarchical arrangements inherent in rooted trees. This paper, in particular, focuses on effective methods for determining the Extreme edge weight between two nodes in a tree using the LCA.

III. IMPLEMENTATION

A. Finding the LCA using binary lifting

The provided code implements the Lowest Common Ancestor (LCA) algorithm using binary lifting for efficient retrieval of Extreme edge weights between two nodes in a tree. These are the key components :

1. Global Variables and Initialization:

The code initializes global variables including adjacency list (adj), dynamic arrays (tin, tout, up, and maxL), and integer variables (l for logarithmic height and timer for time-stamping).

```
#include <bits/stdc++.h>
#include <stdio.h>
using namespace std;

const int maxN = 1e5 + 7;

// GLOBAL VARIABLE
vector<pair<int, int>> adj[maxN];
vector<int> tin, tout;
vector<vector<int>> up, maxL;
int l, timer;
```

2. Depth-First Search (DFS) Function:

The dfs function takes the current vertex v, its parent p, and the maximum length from the root to the current vertex (maxLen).It populates the tin and tout arrays with timestamps for each vertex, and computes the binary lifting tables (up and maxL) for LCA queries.The function then recursively calls itself for each child vertex (u) in the adjacency list.

```
void dfs(int v, int p, int maxLen)
{
    tin[v] = ++timer;
    up[v][0] = p;
    maxL[v][0] = maxLen;
    for (int i = 1; i <= l; i++)
    {
        up[v][i] = up[up[v][i - 1]][i - 1];
        maxL[v][i] = max(maxL[v][i - 1],
maxL[up[v][i - 1]][i - 1]);
    }
    for (auto u : adj[v]) // u.first = vertex
u.second = weight
        if (u.first != p)
            dfs(u.first, v, u.second);
    tout[v] = ++timer;
}
```

3. isAncestor Function:

The isAncestor function checks whether a vertex u is an ancestor of vertex v based on their timestamps in the DFS traversal.

```
bool isAncestor(int u, int v)
{
    return tin[u] <= tin[v] && tout[u] >= tout[v];
}
```

4. LCA Function:

The LCA function computes the Lowest Common Ancestor of two vertices u and v.It checks if one vertex is an ancestor of the other and performs a binary lifting traversal to find the LCA efficiently.

```
int lca(int u, int v)
{
    if (isAncestor(u, v))
        return u;
    if (isAncestor(v, u))
        return v;
    for (int i = l; i >= 0; --i) // greedy
    {
        if (!isAncestor(up[u][i], v))
            u = up[u][i];
    }
    return up[u][0];
}
```

5. maxSum Function:

The maxSum function computes the maximum edge weight between two vertices u and v through their LCA.It utilizes the binary lifting tables to efficiently traverse the tree and calculate the maximum edge weight.

```
int maxSum(int u, int v)
{
    int LCA = lca(u, v);
    int maxLen = 0;
    for (int i = l; i >= 0; i--)
    {
        if (isAncestor(LCA, up[u][i]))
        {
            maxLen = max(maxLen, maxL[u][i]);
            u = up[u][i];
        }
    }
    for (int i = l; i >= 0; i--)
    {
        if (isAncestor(LCA, up[v][i]))
        {
            maxLen = max(maxLen, maxL[v][i]);
            v = up[v][i];
        }
    }
    return maxLen;
}
```

6. Preparation Function (prep):

The prep function initializes necessary variables and arrays based on the number of vertices (sz).

```
void prep(int sz)
{
    tin.resize(sz);
    tout.resize(sz);
    timer = 0;
    l = ceil(log2(sz));
    up.assign(sz, vector<int>(l + 1));
    maxL.assign(sz, vector<int>(l + 1));
}
```

7. Main Function:

The main function is where we initialize the tree verticesand add the edges to each vertices we use. After that, we perform queries and print the result.

```
int main()
{
    // Initialize the tree with 20 vertices
    prep(20);

    // Adding edges to the tree (example)
    adj[1].push_back({2, 5});  // Edge from vertex 1
to vertex 2 with weight 5

    // Perform queries
    int u, v;
    // Example Query 1
    u = 4, v = 6;
    int lcaResult = lca(u, v);
    int maxSumResult = maxSum(u, v);
    cout << "Lowest Common Ancestor of " << u << "
and " << v << ": " << lcaResult << endl;
    cout << "Maximum Edge Weight between " << u << "
and " << v << ": " << maxSumResult << endl;

    return 0;
}
```

B. Analyzing The Code

For each node in the tree, we precompute its ancestors using the Binary Lifting technique. Specifically, we store the ancestors at different levels: the ancestor above the node, the ancestor two nodes above, the ancestor four above, and so on. This information is stored in the array up, where up[i][j] represents the 2^j-th ancestor above node i, with i ranging from 1 to N and j from 0 to ceil(log(N)). The use of this array enables us to efficiently jump from any node to any ancestor above it in O(log N) time. The computation of this array is performed through a Depth-First Search (DFS) traversal of the tree. Additionally, for each node, we record the time of its first visit (when discovered during DFS) and the time when we leave it (after visiting all its children and exiting the DFS function). This information aids in determining, in constant time, whether a node is an ancestor of another node. Upon receiving a query (u, v), we can quickly check if one node is the ancestor of the other. If u is not the ancestor of v and vice versa, we climb the ancestors of u until we find the highest node that is not an ancestor of v. This is achieved by iterating through the ancestors of u from ceil(log(N)) to 0 and checking, in each iteration, whether one node is the ancestor of the other. The LCA is then determined as up[u][0], representing the smallest node among the ancestors of u that is also an ancestor of v. As a result, each LCA query can be answered in O(log N) time. This efficient approach significantly reduces the time complexity for answering LCA queries and enhances the overall performance of the tree traversal algorithm.

IV. Test Case

To assess the effectiveness of the Lowest Common Ancestor (LCA) implementation, we conducted several test cases on trees of varying sizes and structures. The goal was to evaluate the algorithm's performance in different scenarios and verify its ability to handle diverse tree configurations.

1. Balanced Tree

In the first test case, we examined the algorithm's performance on a balanced tree. This scenario aimed to assess its efficiency when dealing with a tree structure where each level has approximately the same number of nodes. The queries involved random pairs of vertices within the tree.

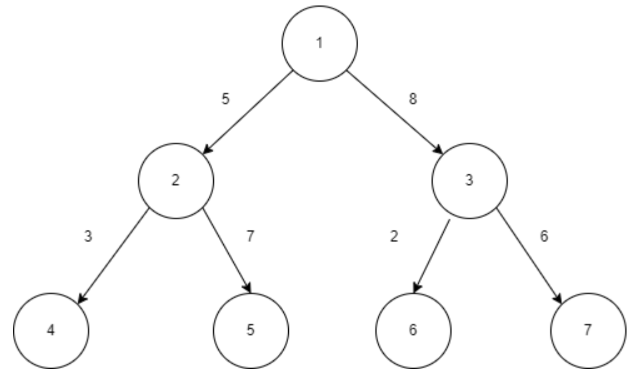


Fig.6 Balanced Tree (source: writer's archive)

In this balanced tree structure, we attempt to find the query results for vertices 4 with 6 and also 5 with 2. It is evident that the Lowest Common Ancestor (LCA) for 4 and 6 is 1, and the corresponding maximum edge weight is 8. Similarly, for the vertices 5 and 2, the LCA is identified as 2, with a maximum weight of 7 along the path. The result of the program provide below.

```
PS D:\.Programming\Matdis> ./lca
Lowest Common Ancestor of 4 and 6: 1
Maximum Edge Weight between 4 and 6: 8

Lowest Common Ancestor of 5 and 4: 2
Maximum Edge Weight between 5 and 4: 7
```

Fig.7 Test Case for balanced Tree (source: writer's archive)

2. Skewed Tree

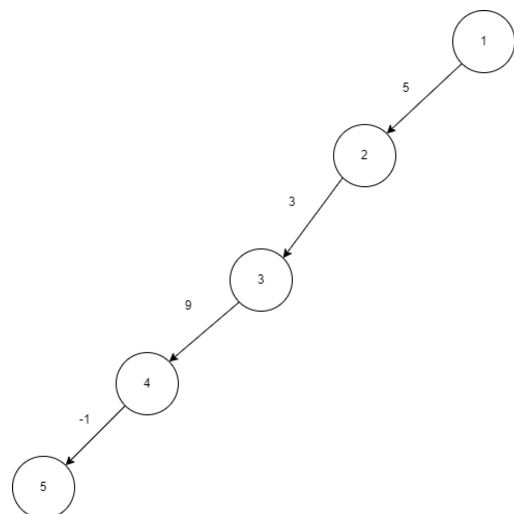


Fig.8 Skewed Left Tree (source: writer's archive)

In this skewed tree structure, we attempt to find the query results for vertices 4 with 2 and also 4 with 6. It is evident that the Lowest Common Ancestor (LCA) for 4 and 6 is 2, and the corresponding maximum edge weight is 9. Similarly, for the vertices 4 and 5, the LCA is identified as 5, with a maximum weight of -1 along the path. The result of the program provide below.

```
PS D:\.Programming\Matdis> ./lca
Lowest Common Ancestor of 4 and 2: 2
Maximum Edge Weight between 4 and 2: 9

Lowest Common Ancestor of 4 and 5: 4
Maximum Edge Weight between 4 and 5: -1
```

Fig.9 Test Case for balanced Tree (source: writer's archive)

3. Random Tree

For the third test case, we introduced a tree with a random structure, including a mix of balanced and skewed subtrees. This more unpredictable scenario aimed to mimic real-world tree structures commonly encountered in practical applications. The LCA algorithm was subjected to diverse query scenarios to assess its adaptability and reliability.

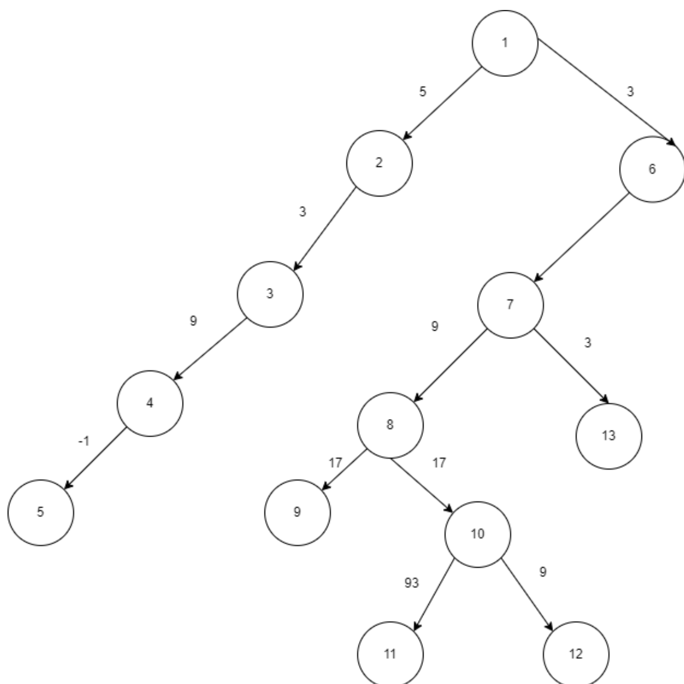


Fig.6 Random Tree (source: writer's archive)

In this random tree, we want to find the query for vertices 9 with 10, vertices 3 with 7, and also 13 and 19. The result shown in the figure below.

```
PS D:\.Programming\Matdis> ./lca
Lowest Common Ancestor of 9 and 10: 8
Maximum Edge Weight between 9 and 10: 17

Lowest Common Ancestor of 3 and 7: 1
Maximum Edge Weight between 3 and 7: 5

Lowest Common Ancestor of 13 and 9: 7
Maximum Edge Weight between 13 and 9: 17
```

Fig.11 Test Case for Random Tree (source: writer's archive)

V. Conclusion

In conclusion, this paper has explored the implementation of the Lowest Common Ancestor (LCA) algorithm in the context of a tree to efficiently calculate the extreme edge's weight along the path between two vertices. The fundamental concepts of trees, paths, and cycles in graph theory were introduced to provide a comprehensive foundation for understanding the significance of determining the minimum and maximum edge weights.

The focus of the paper was on utilizing the LCA algorithm as a powerful tool to address the problem of finding the lowest common ancestor between two arbitrary vertices in a tree that used to find the extreme weight of the edge. The algorithm's efficiency stems from its ability to reduce the time complexity of ancestor queries, crucial for optimizing various applications across different domains.

Furthermore, the implementation of the LCA algorithm with binary lifting was discussed, emphasizing its role in achieving a time complexity of O(log n) for both building the tree and answering queries. The binary lifting technique, along with the Depth-First Search (DFS) traversal, contributes to the algorithm's effectiveness in handling large trees.

V. Acknowledgment

The writer would like to thank all IF2120 lecturers especially Dr. Fariska Zakhralativa Ruskanda, S.T., M.T. as lecturer in second class of IF2120 for Discrete Mathematics, for teaching and supporting students to write these paper. I have gained much better understanding in graph, tree, and its application. I also would ike to thank Dr. Ir. Rinaldi, M.T, who provided students with plenty of resources on Discrete Mathematics at the website.

References

[1] R. Munir, "Graf Bagian 1," IF2120 Matematika Diskrit. Retrieved: December 7, 2023, from https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2023-2024/19-Graf-Bagian1-2023.pdf

[2] R. Munir, "Graf Bagian 2," IF2120 Matematika Diskrit. Retrieved: December 7, 2023, from https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2023-2024/19-Graf-Bagian2-2023.pdf

[3] R. Munir, "Graf Bagian 3," IF2120 Matematika Diskrit. Retrieved: December 7, 2023, from https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2023-2024/19-Graf-Bagian1-2023.pdf

[4] R. Munir, “Pohon Bagian 1,” IF2120 Matematika Diskrit. Retrieved: December 7, 2023, from https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2023-2024/22-Pohon-Bag1-2023.pdf
[5] R. Munir, “Pohon Bagian 2,” IF2120 Matematika Diskrit. Retrieved: December 7, 2023, from https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2023-2024/22-Pohon-Bag2-2023.pdf
[6] Geeks, G. F. (2021, July 16). LCA in a tree using Binary Lifting Technique. Retrieved December 7, 2022, from https://www.geeksforgeeks.org/lca-in-a-tree-using-binary-lifting-technique/
[7] Michael, L., & Kapoutsis, C. (2003). Lecture notes on LCAs from a 2003 MIT Data Structures course. Course by Erik Demaine, notes written by Loizos Michael and Christos Kapoutsis. Notes from 2007 offering of same course, written by Alison Cichowlas.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 3 Desember 2023

Berto Richardo Togatorop 13522118