Level Ancestor Problem simplified

Cai Qi

Seminar Advanced Algorithms and Data Structures - Student Report

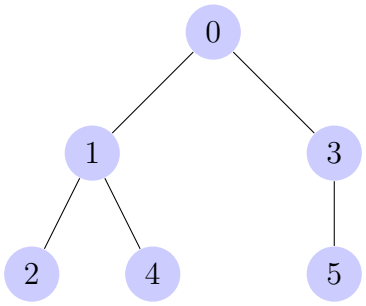
1 Introduction

Here, we will first find out what the *Level Ancestor Problem* respectively what a *Level Ancestor Query* $LA_T(u, d)$ is and define some essential terms.

Definition 1. A `rooted tree` $T = (V, E)$ is a tree which has an arbitrary vertex as the root.

Definition 2. The `depth` of a vertex u in a tree T, denoted as *depth(u)*, is the level in which it is located, starting from $depth(root) = 0$.
i.e. $depth(u) :=$ #edges on the shortest path from root to u

Example 1. with node 0 as root we have



node	0	1	2	3	4	5
depth	0	1	2	1	2	2

Definition 3. Let v be a descendant of the vertex u in a tree T and P_v the u-v-path (which is unique because T is a tree).
Then: v is the `deepest descendant` of $u \iff |P_v| \geq |P_x| \; \forall x$ is a descendant of u.

Definition 4. The `height` of a vertex u in a tree T, denoted as *height(u)*, is the number of levels we have to go through, if we want to travel from u to one of its deepest descendants (which is a leaf). Note that here, the leaves are defined to have height 1.
i.e. $height(u) :=$ #vertices on the path from u to one of its deepest descendants

e.g. for the graph in example 1 we have:

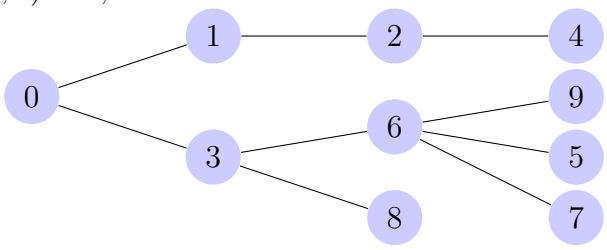
node	0	1	2	3	4	5
height	3	2	1	2	1	1

Level Ancestor Problem. Let $T = (V, E)$ be a rooted tree with $|V| = n$, $|E| = n - 1$. Find the ancestor v of a given node u at depth d.
So for an ancestor v of u with $depth(v) = d$, a `Level Ancestor Query` $LA_T(u, d)$ returns v or $undefined/false$. I.e.

$$LA_T(u, d) = \begin{cases} v & \text{,if } v \text{ is the unique ancestor of } u \text{ with } depth(v) = d \\ false & \text{,if such an ancestor doesn't exist} \end{cases}$$

Note that $LA_T(u, root) = root$ and $LA_T(u, depth(u)) = u \ \forall u \in V$.

Example 2. with 0 as root, we should have $LA_T(6, 1) = 3$, $LA_T(4, 2) = 2$, $LA_T(5, 1) = 3$, $LA_T(8, 2) = 8$, etc.



To solve this Problem we will see several algorithms which have complexity $f(n)$ for preprocessing T and $g(n)$ for the query $LA_T(u, d)$. We denote the complexity of such an algorithm as $\langle f(n),\ g(n) \rangle$.

2 Simpler Algorithms

Here, we will see 3 different algorithms to solve the Level Ancestor Problem and how they work on examples and in pseudo-code. They are called Table Algorithm, Jump-Pointer Algorithm and Ladder Algorithm. A 4th will be a combination of the last two algorithms.
For all algorithms we first do a *DFS* to assign numbers to the nodes. While doing that we also fill arrays `parent` and `depth` which are both globally accessible. $parent[i]$ tells the direct ancestor of node i and $depth[i]$ = depth of node $i = depth[\ parent[i]\ ] + 1$. Since DFS runs in linear time it won't affect the complexity of the algorithms.

2.1 Table Algorithm

As the name says, it uses a table, more precisely a look-up table which is filled using dynamic programming. Each entry $table[i][j]$ is the ancestor of node j at depth i. This results in a $n \times n$ table for a rooted tree $T = (V, E)$, because $depth(u) \leq n \ \forall u \in V$ and there are n vertices in total. The preprocessing complexity is $O(n^2)$, because we have to fill the table, whereas the complexity for the query $LA_T(u, d)$ is $O(1)$. Which means the Table Algorithm has complexity $\langle O(n^2), O(1) \rangle$.
Now let's see how the algorithm works. Remember we have the auxiliary arrays *parent* and *depth* from the DFS and $LA_T(u, depth(u)) = u$.
For every node j we start at row $i = depth[j]$, assign the entry to i and move upwards. Then each entry has the parent of the node below as its value.

Algorithm 1 TableAlgotithm

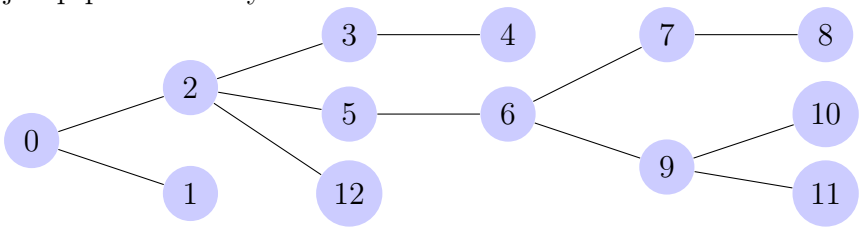
```
1:  procedure createTable                                   ▷ preprocessing
2:      initialize a n × n table with −1
3:      for i = 0, ..., n do
4:          table[depth[i]][i] ← i
5:          for j = depth[i] − 1, ..., 0 do
6:              table[j][i] ← parent[table[j+1][i]]
7:      return table
8:
9:
10: procedure LA_T(u, d)                          ▷ returns ancestor of u at depth d
11:     if table[d][u] = −1 then                  ▷ assume table is globally accessible
12:         return false
13:     return table[d][u]
```

2.2 Jump-Pointer Algorithm

This algorithm uses for each node an array `ptr` containing at most $\log_2(n)$ pointers, also called jump pointers. Each pointer points to the 2^k-th ancestor of the associated node, more precisely the k-th entry of the array associated to node u, i.e. $ptr[k]$, contains the 2^k-th ancestor.
Note that the last index of the pointer array associated to u is $\lfloor \log_2(depth(u)) \rfloor$, which means the highest ancestor of u in its array is the $2^{\lfloor \log_2(depth(u)) \rfloor}$-th ancestor.

The algorithm works as follows: First we create the pointer-arrays for every node except the root by using the precomputed parent array. u and d are given as parameters. Then we look if the right ancestor $x := LA_T(u, d)$ is in $\texttt{ptr}_u$, if *yes* return the node, if *no* we go to $ptr_u[\lfloor \log_2(depth(u) - d) \rfloor] = v$ and look in $\texttt{ptr}_v$ for x by doing the same as in $\texttt{ptr}_u$, but with remaining distance between u and x, i.e. $(depth(u) - d) - (depth(u) - depth(v))$. We repeat this process until we find x.

Example 3. Here we have a rooted tree with $root = 0$ (numeration from DFS) and its jump-pointer arrays.



ptr_1	0

ptr_2	0

ptr_3	2	0

ptr_4	3	2

ptr_5	2	0

ptr_6	5	2

ptr_7	6	5	0

ptr_8	7	6	2

ptr_9	6	5	0

ptr_{10}	9	6	2

ptr_{11}	9	6	2

ptr_{12}	2	0

Algorithm 2 JumpPointerAlgorithm

```
1: procedure createPointerArrays                                  ▷ preprocessing
2:     for all node v ≠ root do
3:         compute smallest k, s.t. ⌊depth(v) / 2^k⌋ = 0
4:         allocate array ptr of length k for u
5:         for i = 0, ..., k − 1 do
6:             ptr[i] ← 2^i-th ancestor of u
7:     return Collection of pointer-arrays                        ▷ e.g. Array of Arraylists
8:
9:
10: procedure LA_T(u, d)                                          ▷ returns ancestor of u at depth d
11:     if depth[u] < d then                                      ▷ We cannot find an ancestor of u that is below u
12:         return false
13:     else if depth[u] = d then
14:         return u
15:     δ ← depth[u] − d
16:     ancestor ← ptr_u[0]
17:     while depth[ancestor] ≠ d do
18:         if δ = 2^k, for a k ∈ ℕ then
19:             ancestor ← ptr_u[k]
20:         else
21:             idx ← ⌊log_2(δ)⌋
22:             u ← ptr_u[idx]
23:             ancestor ← ptr_u[0]
24:             δ ← δ − 2^idx
25:     return ancestor
```

In the preprocessing part we compute for every node an array with length less or equal to $\lfloor \log_2(n) \rfloor$. Which results in complexity $O(n \cdot \log_2(n))$.
For the query time complexity, we define $\delta := depth(u) - d$ for the given parameters u and d, i.e. δ is the distance we have to travel to reach the right ancestor. At the first array we can travel up to the $\lfloor \log_2(\delta) \rfloor$-th entry. This means we can travel up by $2^{\lfloor \log_2(\delta) \rfloor}$ which is at least $\delta/2$. With the second array we can travel up to at least half of the remaining distance, and so on. This results in query time complexity of $O(\log_2(n))$. In total the Jump-Pointer Algorithm has complexity $\langle O(n \cdot \log_2(n)), O(\log_2(n)) \rangle$.

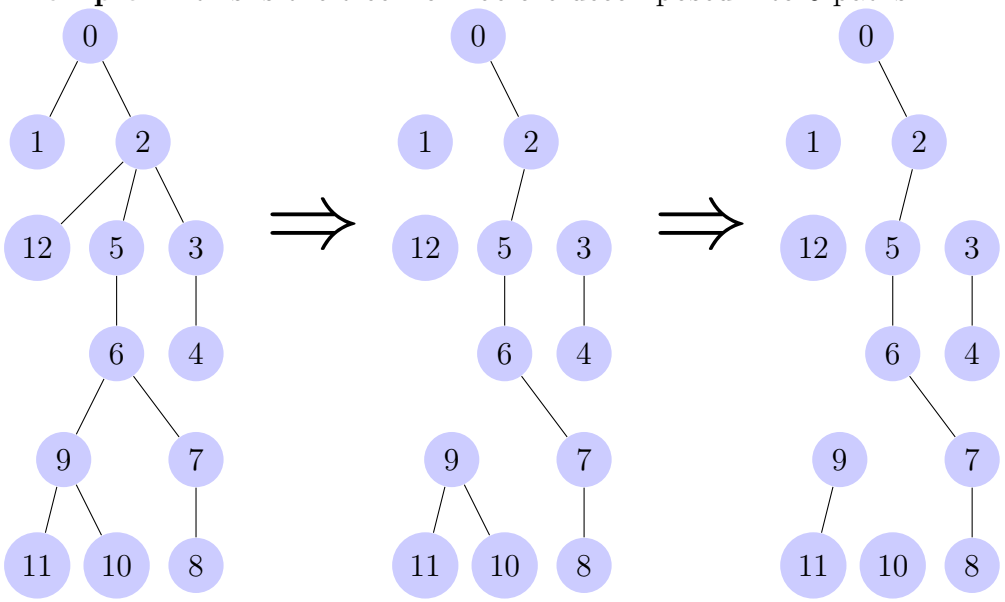
2.3 Ladder Algorithm

The idea here is to decompose the tree into paths/ladders. Suppose we have a tree that is a path $P = \{root = v_0, v_1, ..., v_{n-1}\}$ stored in an array `ladder`, then the entry `ladder[i]` is the node at depth i. This allows us to output $LA_T(u, d) = ladder[d]$ in time $O(1)$, as long as $d < depth(u)$. Now we want to use this nice property.

We decompose the tree as follows: First, we find a longest root-to-leaf path P and cut its bindings, i.e. delete all edges $e = \{u, v\}$ with $u \in P, v \notin P$ or $u \notin P, v \in P$. This decomposes the tree into P and several subtrees. We store P into an array and recurse

for each subtree, whereat we choose new roots for them. The base case is, when the subtree is only a path (note: a single node is also a path).

Example 4. this is the tree from before decomposed into 6 paths



For each path P we create a ladder array. The ladders are named A, B, C,... to not confuse them with nodes. We also create a global array `a` to keep track in which ladder at which index each node is. This step is called `long path decomposition`.

A	0	2	5	6	7	8

B	3	4

C	1

D	12

E	9	11

F	10

node	0	1	2	3	4	5	6	7	8	9	10	11	12
ladder	A_0	C_0	A_1	B_0	B_1	A_2	A_3	A_4	A_5	E_0	F_0	E_1	D_0

After the long path decomposition is done, we proceed by extending the ladders(except the one with the root at the top) up to twice its length, i.e. a ladder array with length h has now length $2h$. The additional h spaces will be filled with the h immediate ancestors of the top node by adding them to the top of the ladder, s.t. one ladder contains $2h$ nodes with a node of height $2h$ (because it is the ancestor of the $2h$ nodes in the ladder) as top and/or it contains the full path from the root to a leaf. (Assume we can have negative indices s.t. we don't have to change the pointer array, it can also be done differently.)

A	0	2	5	6	7	8

B	0	2	3	4

C	0	1

D	2	12

E	5	6	9	11

F	9	10

For a query we follow the pointer to the ladder and either the right ancestor is in the ladder or we have to jump to the ladder of the current ladder's top, and do the same there. Note that we must always calculate the remaining distance between the depth of the node at which we are and the depth of the wanted node.

Algorithm 3 LadderAlgorithm

```
1: procedure createLadder(tree T=(V,E))                          ▷ preprocessing
2:     choose longest root-leaf path P={v_0, ..., v_{h-1}}
3:     allocate array ladder of length h                          ▷ h := #nodes in P
4:     for i = 0, ..., h − 1 do
5:         ladder[i] ← v_i
6:         a[v_i]← ladder_i                                       ▷ suppose ladder is a letter
7:     allocate additional h space at top of ladder
8:     for i = −1, ..., −h do
9:         if ladder[i+1] = root then
10:            deallocate remaining not-occupied space            ▷ s.t. access of top is easy
11:            break
12:        ladder[i] ← parent[ladder[i+1]]
13:    M = {(u, v) ∈ E | u ∈ P, v ∉ P and vice versa}
14:    for all edge e ∈ M do
15:        T_e ← subgraph that is connected to P via e
16:        createLadder(T_e)
17:                ▷ e is the only edge connecting T_e and P, else there would be a cycle
18:            ▷ Note for every e ∈ M we get a T_e, meaning there will be |M| times G_e's
19:
20:
21: procedure LA_T(u, d)                                          ▷ returns ancestor of u at depth d
22:    if depth[u] < d then
23:        return false
24:    else if depth[u] = d then
25:        return u
26:    ladder_i ← a[u]                                            ▷ ladder_i corresponds to e.g. C_3
27:    while depth[ladder[top]] > d do                            ▷ a top pointer can be created easily
28:        u ← ladder[top]
29:        ladder_i ← a[u]
30:    distance ← depth[u]−d
31:    return ladder[i-distance]
```

Consider we are at a vertex of height h (remember height is the number of vertices between the vertex and its deepest descendant). Then we can travel up to the top and reach a vertex of height at least $2 \cdot h$ (except we reach the root, but in this case, the needed ancestor is certainly in the current ladder).
The worst case would be, that we start at a ladder consisting of 2 vertices, i.e. it had one vertex v before being extended by its ancestor u. v is without a doubt a leaf and has height 1. We go to the top of v's ladder and get u, $height(u) = 2$. If we jump to u's ladder and travel to its top we certainly reach a node x with $height(x) \geq 4$, since u can only be in the lower half of its ladder.
I.e. if u is the top of the ladder before it got extended and the ladder had h elements, then the ladder has currently $2h$ elements and the top node x has $height(x) = 2 \cdot height(u)$. This implies that we reach a node of height at least 2^i after

we jumped i ladders which gives us a query time complexity of $O(\log_2(n))$.

In the preprocessing part, we choose a longest root-leaf path and remove it from the tree. To find the longest root-leaf path we create a `child` array which contains for every node a child of maximal height which can be done in $O(n)$ by e.g. checking the height of the children for each node (Note it's a tree, so $\#edges = n - 1$ and the total nodes to be checked is done in $O(n)$). The `height` array can be filled e.g. by starting at a leaf, which has height 1, and going up to the last unvisited ancestor and each time we set height(current node) = height(previous node)+1. The extending is done by following the parent array.

All together it makes the Ladder Algorithm an $\langle O(n), O(\log_2(n))\rangle$ algorithm.

2.4 Ladder-JumpPointer combination

The idea is to use only one jump-pointer that transports us halfway to the needed ancestor and then climb up one ladder. Therefore we have to do both preprocessing which needs time $O(n \cdot \log_2(n))$.

For the query $LA_T(u, d) = x$ we define $\delta := depth(u) - d$, the distance between u and x. With the jump-pointer we can reach the $2^{\lfloor \log_2(\delta)\rfloor}$-th ancestor v (shown in subsection 2.3) which means that $height(v) = 2^{\lfloor \log_2(\delta)\rfloor}$ (a 2^k-th ancestor of a node has 2^k descendants) and that we can travel up at least halfway to x. From the Ladder Algorithm we know that v's ladder has at least $2 \cdot 2^{\lfloor \log_2(\delta)\rfloor}$ elements. We also know that x can be at most $2^{\lfloor \log_2(\delta)\rfloor}$ elements away from v, so x must be in v's ladder.

In total it's a $\langle O(n \cdot \log_2(n)), O(1)\rangle$ algorithm.

3 The Macro-Micro-Tree Algorithm

In this section, we want to improve the preprocessing of the Ladder-JumpPointer combination to $O(n)$. Based on the observation that we don't need jump pointers of a vertex u, if a descendant vertex v of u has jump pointers because $LA_T(u, d) = LA_T(v, d)\ \forall d \leq depth(u)$, we would intuitively say, let's only assign jump pointers to the leaves. But this only speeds up the case where the tree has $O(n/\log_2(n))$ leaves. So the idea now is to assign jump pointers to $O(n/\log_2(n))$ nodes which cover as much of the tree as possible. To make it easier, we introduce the terms `jump node, macro node, micro node`.

Definition 5. Since we do not assign jump pointers to all the vertices but few, we call the ones having jump pointers `jump nodes`. For $n = \#vertices$ in the tree, jump nodes satisfy following two properties:

1. Each jump node has at least $\log_2(n)/4$ descendants (note that the node itself is also a descendant).

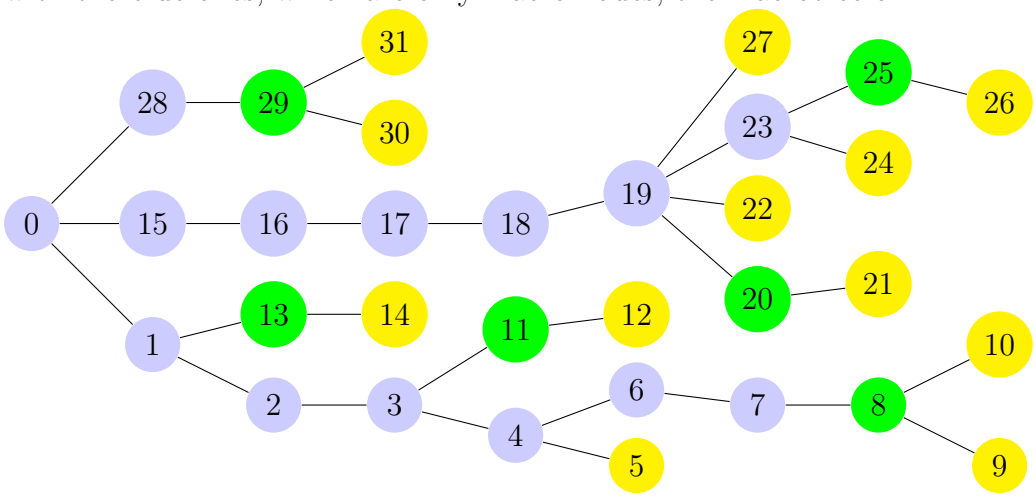
2. Every child of a jump node has fewer than $\log_2(n)/4$ descendants.

The $\log_2(n)$ is there in order to bound the number of jump nodes to $O(n/\log_2(n))$ and the factor $\frac{1}{4}$ will play a role for the performance which we will see later.

Definition 6. Any ancestor of a jump node, including the jump node itself, is defined as a `macro node`. Together they build one big `macrotree`.

Definition 7. Any node that isn't a macro node is a `micro node`. They build several connected components. We call such a connected component of micro nodes a `microtree` (a single node can also be a microtree).

Example 5. Here we have a rooted tree T with $2^5 = 32$ vertices. The green ones are the jump nodes, in total $6 \leq n/\log_2(n) = 6.4$. Each of them has at least $\log_2(n)/4 = 5/4 = 1.25$ descendants (each of them has at least 2 descendants, because we have to round up) and each child of a jump node has strictly less than 1.25 descendants (every child has 1 descendant, namely itself). The 12 yellow nodes are micro nodes building 12 microtrees. The jump nodes, which are also macro nodes, build with the blue ones, which are only macro nodes, the macrotree of T.



Claim 1. We can solve the *LevelAncestorProblem* for macro nodes in $\langle O(n), O(1)\rangle$.

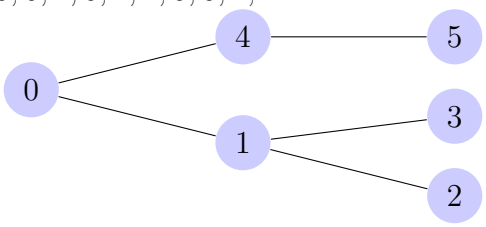
Proof. First, we have to create the (extended) ladders for the given tree T. This happens as seen before in $O(n)$. The jump nodes can be found using a `descendant` array, which contains the number of descendants for each node and can be filled similarly as the height array in subsection 2.3 at the end. Once we have them, we need to create the jump pointers. For this we use the ladders which gives us $O(\log_2(n))$ time for each jump node. Since there are only $O(n/\log_2(n))$ jump nodes, we get a time complexity of $O(n)$ for preprocessing.
For queries we create a `jumpDesc` array, which can be filled with a DFS visit. Each entry i contains one jump node which is descendant of node i. Since $LA_T(u,d) = LA_T(v,d)$ $\forall d \leq depth(u)$ and $\forall v$ is descendant of u, we can compute $LA_T(u,d)$ by computing $LA_T(jumpDesc(u),d)$ the way shown in subsection 2.4 in time $O(1)$. □

Now we know how to compute $LA_T(u,d)$ for u is a macro node. If u is a micro node we must do something else. In fact we are going to apply the Table Algorithm on each micro tree and to keep the preprocessing complexity at $O(n)$ we show that there are only $O(\sqrt{n})$ possible shapes of a microtree.

Claim 2. Microtrees come at most $\sqrt{n}$ shapes.

Proof. Consider $m = \#edges$ in a microtree, then $m < \log_2(n)/4 - 1$, because a microtree has fewer than $\log_2(n)/4$ vertices and a tree has $|E| = |V| - 1$ edges. Let an *up edge* be an edge traversed from child to parent in the DFS and a *down edge* an edge traversed from parent to child. Then a microtree's shape can be completely described by up and down edges.
Let 0 represent a down edge and 1 an up edge. A microtree's shape can now be represented as a bit-string of length $2m$ and the number of possible permutations is $2^{2m} < 2^{2(\log_2(n)/4-1)} \leq 2^{\log_2(n)/2} = \sqrt{n}$ □

Example 6. Consider this microtree and DFS-order: $0, 1, 2, 3, 4, 5$. Then the bit-string representing it would be: $0, 0, 1, 0, 1, 1, 0, 0, 1, 1$.



The Algorithm. The preprocessing for the macro nodes works as described before in proof of claim 1, additionally a set `M`:= set of all macro nodes can be created by adding all nodes having at least $log_2(n)/4$ descendants to M. The step is done while we search the jump nodes.
For micro nodes we must first build the micro trees which is done by starting several DFS', where we also set pointers to the roots of the microtrees (for the creation of tables). The DFS' will tell us which nodes are connected, i.e. which nodes build a microtree. We make for each microtree a set containing the tree's nodes. During those DFS' we construct the bit-strings that represent the microtree shapes. After a bit-string is built, we assign it to the shape, enumerate the shapes and create a table for each shape.
Since a microtree has size less than $log_2(n)/4$ and there are $O(\sqrt{n})$ shapes (claim 2), the complexity is $O(\sqrt{n} \cdot log_2(n)^2) = O(n)$.

For a query $LA_T(u, d)$ we have to determine if u is a macro or a micro node by checking if $u \in M$. If it's a macro node we can find the right ancestor in $O(1)$ described in proof of claim 1.
If u is a micro node, we look in which microtree-set it is, check up the microtree's shape (the corresponding bit-string) and search the ancestor in the corresponding table (We can create a mapping between the node numbers of the microtree and those of the shape table) in constant time, if the ancestor is in the microtree, that is $depth(root\ of\ microtree) \leq d$.
Else (i.e. $depth(root\ of\ microtree) > d$) the wanted ancestor is a macro node and outside of the microtree. In this case we can look up the parent of the microtree's root which is a macro node and proceed like before when we described the query for macro nodes.

Summing up, the total time complexity to solve the Level Ancestor Problem is $\langle O(n), O(1) \rangle$ for the Macro-Micro-Tree Algorithm.

4 Feedback on the Paper

The paper "The Level Ancestor Problem simplified" by Michael A. Bender and Martin Farach-Colton is quite well structured. It starts with an introduction telling us the structure of the paper and what the goal of each section is, then proceeds with important definitions, necessary algorithms which combined build the last algorithm having the best time complexity. However, the main focus was on the time complexity, such that the space complexity was never relevant, though it's certainly bounded by the time complexity. Also the proofs are quite poor, especially when it comes to implementation. Hardly an algorithm's implementation is well described. There are only proofs about the fact that it is possible to run the algorithms in the given time complexity, but not e.g. how it is done in pseudo-code. In spite of everything the descriptions are sufficient to understand how the algorithms work.

References

Michael A. Bender[a], Marín Farach-Colton[b], *The Level Ancestor Problem simplified*,
[a]Departement of Computer Science, State of New York at Stony Brook, NY 11794-4400, USA,
[b]Departement of Computer Science, Rutgers University, Piscataway, NJ 08854, USA
Received 15 August 2002; received in revised from 2 May 2003; accepted 22 May 2003

Wikipedia, Level Ancestor Problem,
https://en.wikipedia.org/wiki/Level_ancestor_problem, 25 Oktober 2018