AA - Second Stage - Lowest Common Ancestor

Ploscaru Alexandru

Politechnica University of Bucharest

Abstract. Testing different LCA algorithms with a complete set of tests and comparing the results between them, to better understand how to analyze algorithms.

Keywords: LCA · Binary Search Tree

1 Description of the problem

1.1 Definition

The lowest common ancestor of two nodes u and v in a tree is the lowest (deepest) node that has both nodes as descendants. The LCA algorithm resolves this problem and finds that node which is the shared ancestor between u and v that is located farthest from the root of the tree.

Throughout the evolution of algorithms, there were many tries at finding an efficient LCA algorithm, and today it can be implemented quite easily, and it can resolve the problem quite fast. The LCA problem has been studied intensively because fast algorithms for this can be used to solve other algorithmic problems.

1.2 Practical uses

In graph theory and computer science, the LCA algorithm has a few use cases besides its principle one. It can be used to determine the distance between pairs of nodes in a tree (the distance between *node1* and *node2* can be computed as the distance from the root to *node1* plus the distance from the root to *node2*, minus twice the distance from the root to their lowest common ancestor). This algorithm is deeply related to the RMQ problem, and it has practical uses for solving RMQ problems [1] [2].

In real life practical situations, the LCA algorithm can be used to find the common ancestor of two species of organisms, to find the common root or suffix of two words.

Also, I found that the LCA algorithm can be used in three-way merge algorithms, because it is an efficient and fast algorithm, and it solves the problem quite well. Another use for the LCA is found in computing dominators in flow graphs.

2 Chosen solutions

There are a few algorithms out there written for the LCA problem. I chose to compare the efficiency of three algorithms using binary searching trees.

2.1 First algorithm

The first algorithm I chose computed the LCA of two nodes by storing paths from root to the first and second nodes and comparing the paths until it finds a difference in the path. When it finds that difference, it returns the last common value in the two paths, which is the lowest common ancestor of the nodes. For the reference and more details, see [3]. This algorithm has a time complexity of $O(N)$, N being the number of nodes, for every interrogation, because the tree is traversed twice, and then path arrays are compared. In terms of space, it uses extra space for the two paths from the root of the tree to the nodes.

2.2 Second algorithm

The second algorithm uses parent pointers for the nodes, which simplifies things a bit, using the following fact: if both nodes are at the same level and if the program traverses up using the parent pointers of both nodes, the first common node in the path to root is the lowest common ancestor. First, the program has to find the depths of the nodes, compare them, move along the path to root of the deeper node until both nodes are on the same level, then it traverses up on both paths and returns the first common node. For the reference and more details, see [4]. This algorithm has a time complexity of $O(h)$, h being the height of the tree, for every interrogation, because the program only moves along the path from the node to the root and compares the nodes until it finds the common ancestor. In terms of space, it doesn't use extra space because there are no maps or arrays used, just the pointers of the data structure itself.

2.3 Third algorithm

The third algorithm uses the binary lifting technique. This algorithm uses extra space by pre-computing an array of ancestors (for each node n, the array *memo[n]* will hold every 2^j ancestor of n. Example: *memo[n][i]* contains the $2^i - th$ ancestor of node n). If any node of the two is not the ancestor of the other (that would be the LCA), the program finds a node that is the child of the LCA and returns *memo[child][0]* (which would be its parent, meaning the LCA). For the reference and more details, see [5]. This algorithm has a time complexity of $O(logN)$, N being the number of nodes, for every interrogation, but is preceded by some pre-processing for the computation of the *memo* array, which is roughly $O(NlogN)$. In terms of space, the program uses extra space for the array of ancestors, and for an extra array that keeps track of every node's level in the tree.

3 Evaluation criterias

For testing each algorithm's efficiency, I used a set of 30 complete tests that I generated myself, that include the number of nodes, each edge that generates the tree, the number of interrogations and each interrogation itself. The number of nodes and the number of interrogations are generated randomly and have a max value of 10000, because of the very long time it takes to compute more interrogations than that. For the edges I wrote a program that creates a tree from a list of shuffled numbers and return the list of edges of the tree, so I could confirm that it is a valid tree. For every interrogation, I generated two random numbers.

I think the best cases for testing would arise from using binary searching trees, because using any other type of tree would result in it being quite balanced and for testing purposes, that would not give as much depth to a tree that a binary search tree would.

These tests provide the total time of computing, the memory used by the program in every test, the time that each algorithm takes to complete one interrogation. After all the tests were over, I compared the total time of computing for each algorithm on each test, the total memory used in each test by every algorithm, and the minimum, average and maximum value of the interrogation time for each algorithm.

4 The Algorithms

4.1 First algorithm

The first algorithm I chose is explained at [3], and I took the code from the same website. It works by storing paths from root to each node, and that should result in a little more memory used, but I found out that in terms of memory, this algorithm and program uses the least from all three. It is quite fast but not the fastest algorithm available for LCA.

4.2 Second Algorithm

The second algorithm I chose is explained at [4], and I took the code from the same website. The structure that represents a node has one more element, a pointer to the parent of the node, and the LCA is computed using that pointer. This is the best algorithm in terms of time, is the fastest from all the three. In terms of memory, this algorithm shouldn't use that much memory, but it performed worse than the first, in some cases, but in others, it performed better. Overall, the first algorithm used less memory.

4.3 Third Algorithm

The third algorithm I chose is explained at [5], and I took the code from the same website. I expected this algorithm to perform better when talking about the average time of each interrogation but surprisingly, it performed the worst. In terms of memory, it uses a pre-computed array that stores the node's ancestors, and that takes up a lot of memory, and also the time it takes to compute that array and then compute each interrogation is very high, comparing to the other solutions. I chose this algorithm, LCA using binary lifting, because I found a lot of sources that say it is one of the best algorithms for this problem, but after I have done the testing, I found out the opposite is true, there are other algorithms much better than this one.

5 Evaluation of the algorithms

5.1 Creating the tests

For creating each test I used a java program that outputs each test to a file in the tests folder. First, I generated a random number of nodes, and a random number of interrogations, both with a max value of 10000, a relatively small number, but if it was larger than that, the testing would have lasted too much time, hours to days, that is the reason I dropped the max value of one million. I created a list that has numbers from 0 to the number of nodes - 1, and then I shuffled it. I created a binary search tree and I inserted every element from the shuffled list into that tree. For that tree I created a method that outputs the edge list of the tree, so I can use that to create the same tree when I input the tests into the main program. Then, for every interrogation I created two more random numbers and put them in the test file.

These tests, though relatively small, provided me valuable information and details about the algorithms I chose. This set of tests is a complete one, it tests even the special cases, like performing the LCA of a node and the same node, or two nodes very distant from each other, or a node with the root, so I found it is an exhaustive set of tests.

5.2 Technical details

I wrote the program that creates the tests, all the algorithms and the main testing program in java, because I find it a very powerful programming language and I'm quite familiar with it. It has built-in classes for the binary searching tree and binary searching tree with parent pointer, but I wanted to write the code for that myself. This programming language helped me a lot because after some research I found it is quite fast when it comes to performing such big tasks.

The java version I used to compile this project is: openjdk 17.0.2.

I ran the tests on a laptop that has AMD Ryzen 5, 16GB RAM, Windows 10 Pro, but I also checked that the tests can be ran on Linux, with an Ubuntu VM.

5.3 Interpreting the data

After running the tests, I put all the resulted data in an excel spreadsheet and analyzed it. I compared each algorithm's total time of computing and memory used for each test, and also calculated the minimum, average and maximum interrogation time for each algorithm.

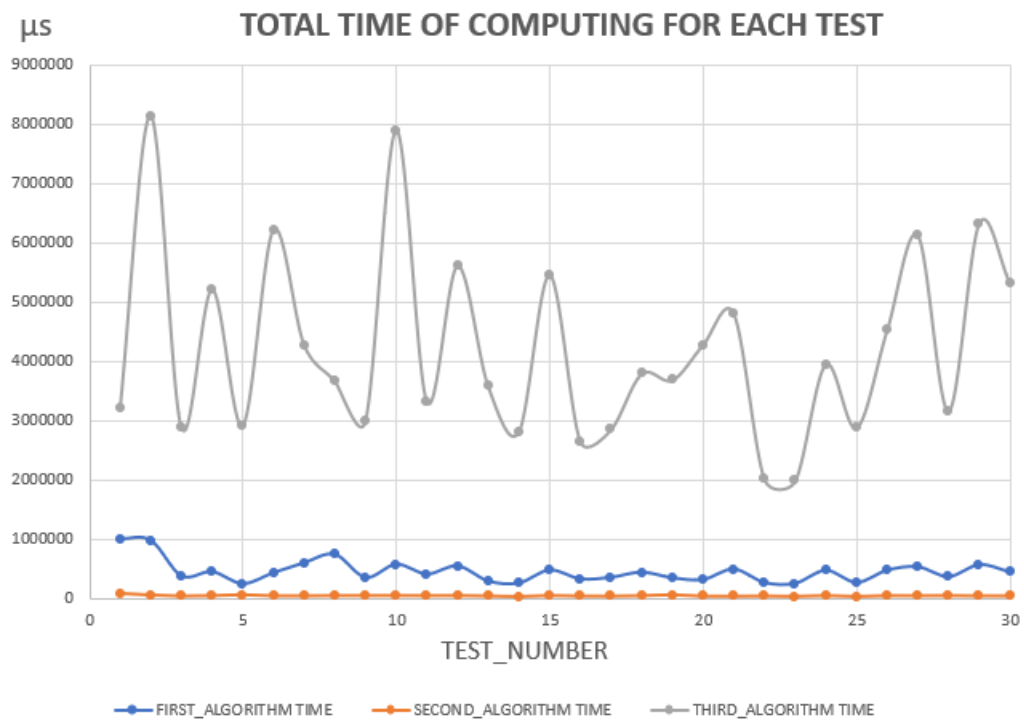
This is the total time of computing of each algorithm for every test:

µs	FIRST_ALGORITHM TIME	SECOND_ALGORITHM TIME	THIRD_ALGORITHM TIME	BEST ALGORITHM ID
Test1	996342	77892	3202389	2
Test2	971486	53205	8137449	2
Test3	374115	37316	2889235	2
Test4	443910	43235	5211339	2
Test5	241712	56696	2922268	2
Test6	426770	40639	6216188	2
Test7	594120	39951	4274305	2
Test8	735205	43792	3669182	2
Test9	346126	44276	2999223	2
Test10	568554	44983	7893782	2
Test11	403517	43217	3323010	2
Test12	541086	44619	5606720	2
Test13	286166	38891	3576973	2
Test14	253433	29937	2812950	2
Test15	471813	44726	5452018	2
Test16	322766	36666	2651966	2
Test17	348460	35629	2847778	2
Test18	435220	42620	3790091	2
Test19	339969	55682	3683432	2
Test20	311309	35254	4273571	2
Test21	487053	33883	4806800	2
Test22	260870	37954	2016135	2
Test23	246908	31985	1979108	2
Test24	466417	43871	3947544	2
Test25	261987	30897	2877271	2
Test26	468631	43331	4524680	2
Test27	524952	41785	6119909	2
Test28	359986	43909	3160159	2
Test29	562148	39874	6320729	2
Test30	437953	39383	5316976	2
BEST ALGORITHM FOR TIME COMPLEXITY				2

Each cell is the total time of computing of the test, represented in microseconds, for example, for the first test, the first algorithm completed the computation in 0.996 seconds, the second in 0.077 seconds and the third in 3.202 seconds, so clearly the second algorithm performed best.

Overall, the second algorithm has the best performance, in all the tests, because working with the parent pointer of every node takes less time than looking in an array.

Here is a visual representation of the data inside the table:



The total memory used by every algorithm for each test is represented below:

KB	FIRST_ALGORITHM MEMORY	SECOND_ALGORITHM MEMORY	THIRD_ALGORITHM MEMORY	BEST ALGORITHM ID
Test1	26804	14922	12844	3
Test2	6641	34053	28801	1
Test3	10946	19543	34315	1
Test4	5080	28855	16387	1
Test5	25818	29746	14460	3
Test6	21518	14009	22298	2
Test7	16740	23788	30070	1
Test8	11855	10348	37981	2
Test9	17703	20568	15524	3
Test10	20597	11463	28238	2
Test11	8533	23166	34441	1
Test12	24473	12460	20042	2
Test13	27535	16500	20587	2
Test14	28382	18023	18927	2
Test15	16242	32426	27970	1
Test16	18039	12418	30708	2
Test17	25989	18575	33512	2
Test18	22653	31902	42469	1
Test19	8505	16469	20645	1
Test20	7276	20674	23991	1
Test21	7600	31954	33326	1
Test22	9846	9789	31345	2
Test23	9591	10365	30346	1
Test24	10375	24227	38950	1
Test25	26543	25734	38871	2
Test26	9881	13443	22022	1
Test27	13264	28362	32689	1
Test28	9966	9886	36710	2
Test29	28230	24275	23692	3
Test30	32132	33938	31145	3
BEST ALGORITHM FOR SPACE COMPLEXITY				1

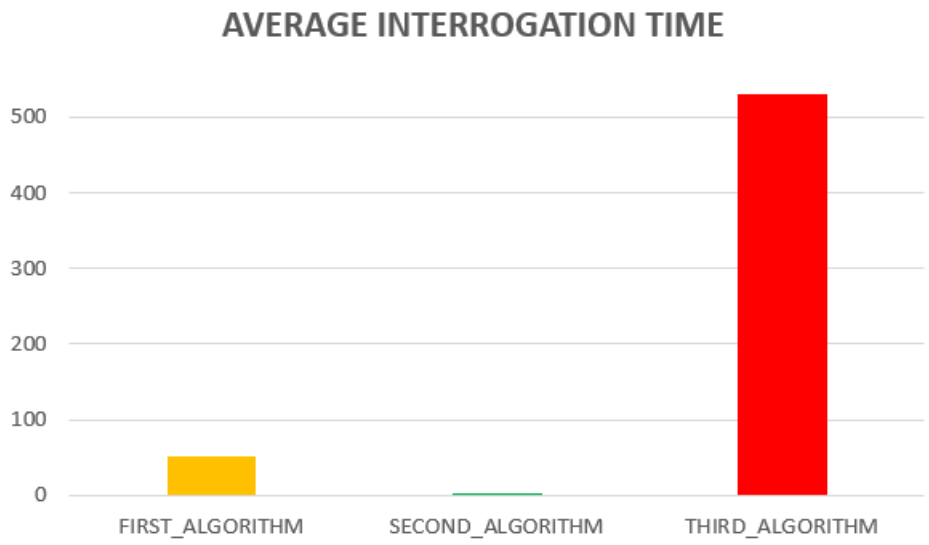
Each cell is the total memory used for each test, represented in KB, for example, for the first test, the first algorithm used 27MB of memory, the second 15MB and the third 13MB, but this is not the case for all tests because overall, the first algorithm is the best in the matter of memory usage.

The average, minimum and maximum values for the interrogation time of each algorithm are:

µs	AVERAGE	MIN	MAX
FIRST_ALGORITHM	51,98341843	0	11793
SECOND_ALGORITHM	0,745988909	0	7886
THIRD_ALGORITHM	530,7174238	1	20254

Each cell represents the time for one interrogation in microseconds, for example, the average time for the first algorithm to complete one interrogation is 0.000051 seconds, but the minimum time is 0 because some interrogations were performed in under a microsecond, same for the minimum interrogation time of the second algorithm.

Here is a visual representation of the average interrogation time:



5.4 Conclusions

After analyzing all the data I found that the second algorithm is the best, in theory and in practice, in the matter of time complexity, and the second best when talking about memory used. It is a very fast solution for the LCA problem and it solves it in a simple and elegant way.

6 Some references I found useful

- I found this website very useful, it taught me a lot about LCA algorithms: https://www.geeksforgeeks.org/lowest-common-ancestor-binary-tree-set-1/
- This article was quite useful, but it was hard to understand because of the programming language: https://www.schoolofhaskell.com/user/edwardk/online-lca
- This pdf helped me understand the uses of the LCA problem and algorithms: https://www.ics.uci.edu/~eppstein/261/BenFar-LCA-00.pdf

References

1. https://www.geeksforgeeks.org/lowest-common-ancestor-in-a-binary-tree-set-3-using-rmq/
2. urlhttps://www.ics.uci.edu/~eppstein/261/BenFar-LCA-00.pdf
3. https://www.geeksforgeeks.org/lowest-common-ancestor-binary-tree-set-1/
4. https://www.geeksforgeeks.org/lowest-common-ancestor-in-a-binary-tree-using-parent-pointer/
5. https://www.geeksforgeeks.org/lca-in-a-tree-using-binary-lifting-technique/