A Comprehensive LCA Problem Review Based on *The LCA Problem Revisited*

ZHANG Caiqi 18085481d

The Hong Kong Polytechnic University

Abstract

In this report, we discuss the Least Common Ancestor problem mainly based on the paper: *The LCA Problem Revisited*. We firstly do a brief literature review including the background, motivation, definitions, crucial results and main techniques used in this paper. Also, we give a very detailed example to illustrate the procedure of the three RMQ algorithms. Then we analyze the real experiment results of the algorithms presented in the paper. After that, we present some further discussion of the LCA problem, try to use the techniques to solve some similar problems, and point out an error of the paper.

1 Introduction

The *Least Common Ancestor (LCA)* problem is one of the most fundamental and well-studied problems on trees. Moreover, LCA problem has very strong relationship with many other famous algorithms. Because of this **background**, people are always finding better solutions of it, not only more efficient, but also more implementable. In this report, our discussion will mainly based on *The LCA Problem Revisited* [1].

The **motivation** of studying the LCA problems is also very sufficient. As we we mentioned before, it is closely related to many algorithmic problems. For example, it appears in computing maximum weight matching in graphs, in computing longest common extensions of strings, finding maximal palindromes in strings and matching patterns with k-mismatches [2]. It is also proved useful in bounded tree-width algorithms [3].

Apart from the theoretical motivation, it also has many applications in the real life. For example, the problem of computing lowest common ancestors of classes in an inheritance hierarchy arises in the implementation of object-oriented programming systems [4]. The LCA problem also finds applications in models of complex systems found in distributed computing [5] [6]. Furthermore, finding LCAs has some relevance in the context of computational biology [7].

2 Definitions

In order to make all the concepts consistent, there is some definitions we should clarrify here.

- **The *Least Common Ancestor (LCA)* problem:**

  Structure to Preprocess: A rooted tree T having n nodes.

  Query: For nodes u and v of tree T , query $\textsc{lca}_T(u, v)$ returns the least common ancestor of u and v in T.

- **The *Range Minimum Query (RMQ)* problem:**

  Structure to Preprocess: A length n array A of numbers.

  Query: For indices i and j between 1 and n, query $\textsc{rmq}_A(u, v)$ returns the index of the smallest element in the subarray $A[i...j]$.

- **Complexity $\langle$*f(n)*, *g(n)*$\rangle$:**

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

- **Euler Tour**

  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 visited 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.

2.1 A Linear Reduction from LCA to RMQ

Because that the soon-to-be-presented algorithms are mainly based on RMQ problem, we should first think how to reduce the LCA problem into RMQ problem. The preparation is as follows:

- Array $E[1, ..., 2n - 1]$: store the nodes visited in an Euler Tour of the tree T. $E[i]$ is the label of the ith node visited in the Euler tour
- Array $L[1, ..., 2n - 1]$: L[i] is the depth of node $E[i]$ of the Euler Tour of T.
- Array $R[1, ..., n]$: Define the *representative* of a node in an Euler tour to be the index of first occurrence of the node in the tour. $R[i]$ is the index of the representative of node i.

After the preparation, let us compute the $\textsc{lca}_T(u, v)$ by the following three steps.

1. The nodes in the Euler Tour between the first visits to u and to v are $E[R[u], . . . , R[v]]$ (or $E[R[v], . . . , R[u]]$).
2. The shallowest node in this subtour is at index $\textsc{rmq}_L(R[u], R[v])$, 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.
3. The node at this position is $E[\textsc{rmq}_L(R[u], R[v])]$, which is the output of $\textsc{lca}_T(u, v)$.

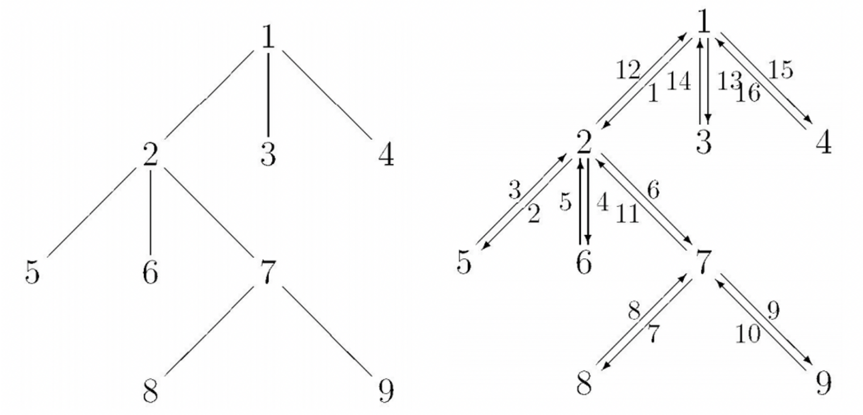


Figure 1: A nine-vertices example and its Euler Tour

2.2 A Nine-vertices LCA Example

To illustrate the above definitions clearly, we give an nine-vertices example here. Suppose that we are going to find the $\text{LCA}_T(5,9)$. Following the three steps to reduce the LCA problem to RMQ problem, we can get the following tables.

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
array E	1	2	5	2	6	2	7	8	7	9	7	2	1	3	1	4	1
array L	0	1	2	1	2	1	2	3	2	3	2	1	0	1	0	1	0

Table 1: The array E and array L for the nine-vertices example

node	1	2	3	4	5	6	7	8	9
array R	1	2	14	16	3	5	7	8	10

Table 2: The array R for the nine-vertices example

To find $\text{LCA}_T(5,9)$, we firstly check Table 2 for node 5 and 9 (in red color) and get their values in R (in blue color). Using $R[5]$ and $R[9]$, we check the Table 1 and reduce the $\text{LCA}_T(5,9)$ to $\text{RMQ}_L(3,10)$. The node is $E[\text{RMQ}_L(3,10)]$, which is 2 in this case. We can easily check in Figure 1 that the LCA of node 5 and node 9 is node 2.

Therefore, we successfully show that the reduction from LCA to RMQ. Now the most significant thing is how to solve the RMQ problem more efficiently. The paper gave us the solution.

3 A Brief Summary of: *The LCA Problem Revisited*

3.1 Summary of the Results

In the paper, it gave us three possible algorithms for the RMQ problems. We will go through them one by one. **Here we will only introduce the main ideas and give a clear example. Details can be found in the paper [1].**

3.1.1 A Naive Solution for RMQ

The naive solution is very simple, we build a table for every pair of nodes. The brute force method takes $O(n^3)$ time. However, we can reduce the time to $O(n^2)$ by using the dynamic programming. With the table, the query time is just $O(1)$. Therefore, this naive solution gives us $\langle O(n^2), O(1)\rangle$ time complexity.

3.1.2 A Faster Solution for RMQ

From the naive solution, we may find that the most time consuming part is to build a table for every two pairs of nodes. However, there is a quicker method using *Sparse Table* (*ST*) to reduce the table building time to $O(n \log n)$.

3.1.3 An ⟨O(n), O(1)⟩-Time Algorithm for ±1RMQ and RMQ

A key observation is that the array L is with±1 restriction. Suppose we have an arbitrary array A with the ±1 restriction. We partition A into blocks of size $\frac{\log n}{2}$. Define an array A'[1, . . . , $2n/\log n$], 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.

The ST algorithm runs on array A' in time ⟨O(n), O(1)⟩. Now the main problem is to answer any query $\text{RMQ}(i, j)$ in A. Because that i and j may be in the same block or not. We divide it into three cases and find that in-block RMQs is the most significant.

Actually, we do not need to calculate the ST for all the blocks, we can normalize the blocks by subtracting its initial offset. We can use the ±1 property to show that there are only $O(\sqrt{n})$ kinds of normalized blocks. The normalization will lead to the total time complexity of $O(\sqrt{n}\log^2 n)$. Another interesting result is that we can also reduce the general RMQ problem to the LCA problem by Cartesian tree and solve it in linear time.

3.2 Overview of Techniques

In the paper, there are some useful techniques which we also learnt from the COMP3011 course. Some of them are listed here in the order of their appearance order of the lectures:

- Big O notation: In the paper, the authors use the Big O notation to analyze the time complexity, which make the presentation and comparison more clear.
- Divide-and-Conquer: When dealing with the RMQ table of an array, the author used a kind of the Divide-and-Conquer thinking, which is dividing the array into blocks and calculate each one.
- Dynamic Programming: In the naive solution of the RMQ problem, the authors used the Dynamic Programming, which is much better than the brute force solution.
- Efficient Data Structures: The authors used some efficient data structures, for example the *Sparse Table* and the *Cartesian tree.*
- Reduction: One of the most crucial method in the paper is reduction. We used the reduction from LCA to RMQ, from RMQ to LCA, and from RMQ to ±1RMQ.

3.3 Revisit The Nine-vertices LCA Example

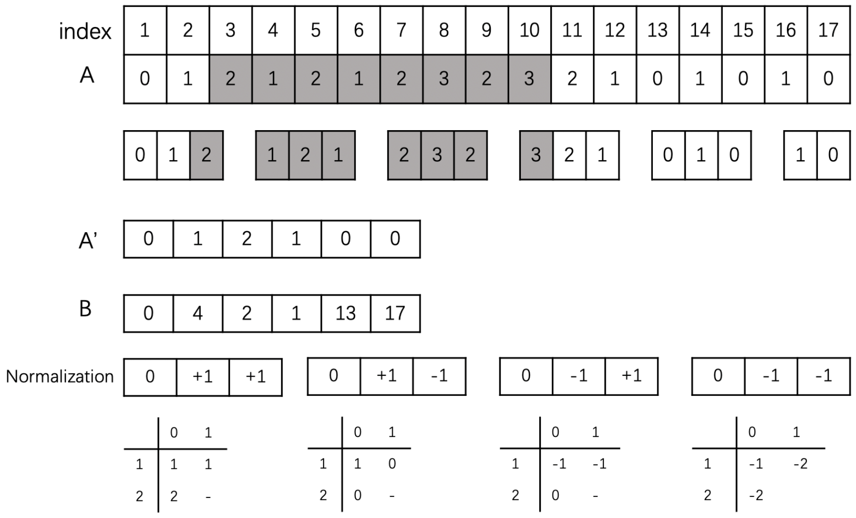
Having the three algorithms, we now try to use them on arrry L. The naive solution is very trivial, so we will omit it here. We will illustrate how to use the second and third methods.

3.3.1 The Faster RMQ Algorithm Using ST

For array L, the ST is as the Figure 2 shows. From index 3 to index 10, it is exactly of length 8, therefore they are in the third block with length 2^3. By checking the ST, we can easily find that the minimum value is 1, as shown in the blue color. The query time is $O(1)$.

	0	1	2	3	4
1	0	0	0	0	0
2	1	1	1	1	0
3	2	1	1	1	-
4	1	1	1	1	-
5	2	1	1	1	-
6	1	1	1	0	-
7	2	2	2	0	-
8	3	2	2	0	-
9	2	2	1	0	-
10	3	2	0	0	-
11	2	1	0	-	-
12	1	0	0	-	-
13	0	0	0	-	-
14	1	0	0	-	-
15	0	0	-	-	-
16	1	0	-	-	-
17	0	-	-	-	-

Figure 2: ST for array L



index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
A	0	1	2	1	2	1	2	3	2	3	2	1	0	1	0	1	0

0 1 2	1 2 1	2 3 2	3 2 1	0 1 0	1 0

A'	0	1	2	1	0	0
B	0	4	2	1	13	17

Normalization	0 +1 +1	0 +1 -1	0 -1 +1	0 -1 -1

	0	1
1	1	1
2	2	-

	0	1
1	1	0
2	0	-

	0	1
1	-1	-1
2	0	-

	0	1
1	-1	-2
2	-2	

Figure 3: Complete steps for array L of ⟨O(n), O(1)⟩-Time Algorithm

3.3.2 The ⟨O(n), O(1)⟩-Time Algorithm for ±1RMQ

As shown in Figure 3, to be consistent with the description of the third algorithm, we rename the previous array L to array A. We then divide into several blocks. The length of each block is $\lceil \frac{\log n}{2} \rceil = 3$. For array A', it stores the minimum element in the ith block of A. And in array B, B[i] is a position in the ith block in which value A'[i] occurs. After that, we list all the possible kinds of normalized blocks and together with their ST.

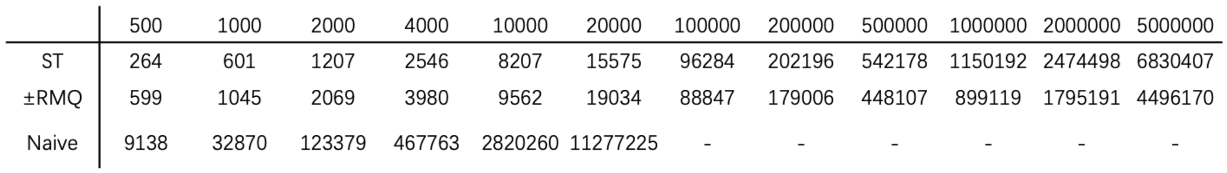
To use these tables to find the $\text{RMQ}_L(3, 10)$, we firstly find how many blocks the $L[3, 10]$ occupy. As shown in the grey color, they are in three blocks. By checking array A', we can easily get the minimum element of the second and third blocks, which are 1 and 2. For the first and last block, we normalize them to [0, +1, +1] and [0, -1, -1]. Check the STs of the normalized blocks, we can know that the minimum element of the first and forth blocks are 2 and 3. Therefore, the $\text{RMQ}_L(3, 10)$ is 1.

Because the block size is small here, the third algorithm does not seem to have a big advantage in this case. However, if the block size is big, the third algorithm will be much quicker. We will see the experiment results in the next section.

4 Experiments

To test the performance of the three algorithms in the real life, we implemented all of the three algorithms based on some online resources [8]. The results are in Figure 4.

In Figure 4, the first row represent the number of nodes, from 500 to 5,000,000. ST represents the $\langle O(n \log n),\ O(1)\rangle$ method. $\pm 1RMQ$ is the faster $\langle O(n),\ O(1)\rangle$ method. $Naive$ is the $\langle O(n^2),\ O(1)\rangle$ method. The unit of all the running time is microsecond (μs). The experiment environment is a 8GB, 64-bit machine with Linux system. Because of the hardware limitation, there is no running time result for $Naive$ method of over 20,000 nodes, as it runs out the memory very fast.



	500	1000	2000	4000	10000	20000	100000	200000	500000	1000000	2000000	5000000
ST	264	601	1207	2546	8207	15575	96284	202196	542178	1150192	2474498	6830407
±RMQ	599	1045	2069	3980	9562	19034	88847	179006	448107	899119	1795191	4496170
Naive	9138	32870	123379	467763	2820260	11277225	-	-	-	-	-	-

Figure 4: Experiment result

To make the graph easy to classify the three algorithms, we divide the results into two graphs. The Figure 5 shows the result of three algorithms up to 1000 nodes. It can be clearly found that the running time of $Naive$ solution increase very fast and the ST almost takes the same time with $\pm 1RMQ$ method. It is worth mentioning that when the number of nodes is small, ST method has better performance than ST. The reason might be that it takes some extra time for the $\pm 1RMQ$ program to call some functions from some packages or libraries to prepare the essential data structures.

Figure 6 shows the running time of ST and $\pm 1RMQ$ with large nodes numbers. When the size of the nodes increases, $\pm 1RMQ$ method becomes more efficient than the ST method, which also prove our theoretical analysis that the $\pm 1RMQ$ method is with lower time complexity.

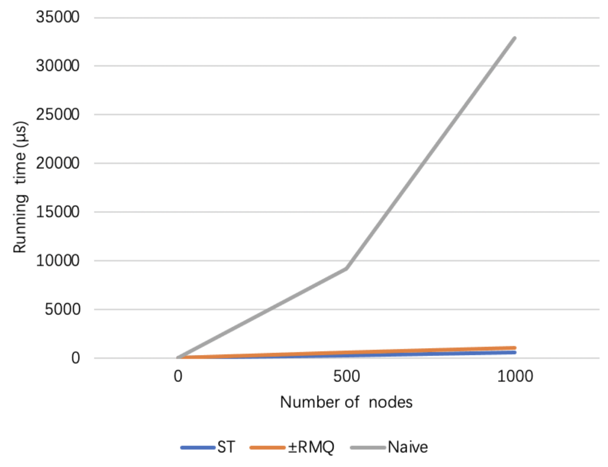


Figure 5: ST, $\pm 1RMQ$, and $Naive$ with less nodes

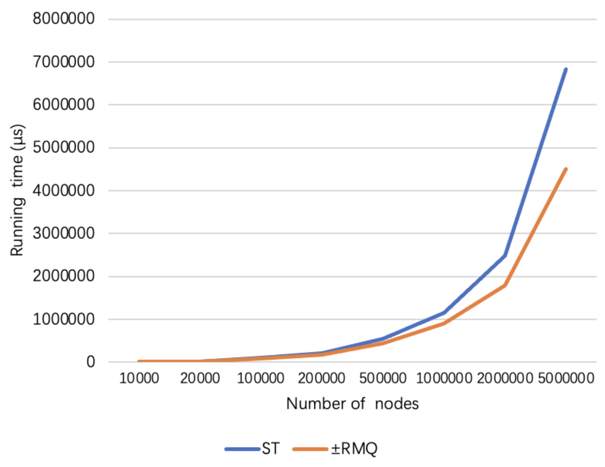


Figure 6: ST and $\pm 1RMQ$ with more nodes

5 Some Theoretical Thinking

5.1 LCA on Some Special Trees

After we get the efficient algorithms of LCA, a very natural thought is whether it can be simpilified in some special trees. Here we give two examples, which are random skewed tree and binary search tree. Although, we can still use the algorithm we presented previously, we are now want to find some more efficient algorithm.

In the random skewed tree, the LCA of two nodes is very easy to find, which is the parent of the node with smaller depth. Therefore, it takes constant time to find the LCA in random skewed tree. The time complexity is $O(1)$ and does not need auxiliary space.

In the binary search tree, it is not as easy as random skewed tree but there is still some easier solution. Create a recursive function that takes a node and the two values a and b. If the value of the current node is less (larger) than both a and b, call the recursive function for the right (left) subtree. If both the above cases are false then return the current node as LCA. For example, if we want to find $\textsc{lca}(10, 14)$, starting from the root, we firstly go to node 8 as 20 is larger than both 10 and 14, then we go to 12 as 8 is less than both two nodes. Then 12 is between 10 and 14, so we find the LCA. The time complexity is $O(h)$, h is the hight of the tree. In average, the time complexity is $O(\log n)$.

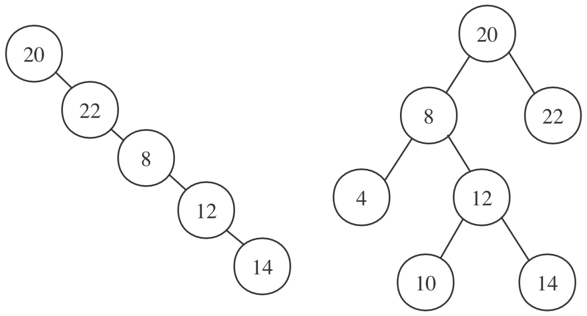


Figure 7: Random Skewed Tree and Binary Search Tree

5.2 Using LCA to solve LCP

After we get the $\langle O(n),\ O(1)\rangle$ time efficient algprithm of LCA, we may have a question: whether we can use the LCA to solve some other problem? The answer is yes. In fact, we can solve the *Longest Common Prefix (LCP)* problem in linear time using LCA [9].

During the reduction, we will use a efficient data structure named *suffix tree*, which is a compressed trie containing all the suffixes of the given text as their keys and positions in the text as their values. Let w_i and w_j be the leaves of the suffix tree of T that represent the suffixes T_i and T_j. The lowest common ancestor of w_i and w_j represents the longest common prefix of T_i and T_j. Thus $\textsc{lcp}(T_i, T_j) = \text{Depth}(\textsc{lca}(w_i, w_j))$, which can be computed in constant time using the suffix tree with LCA preprocessing.

i	SA[i]	suff$_{\mathsf{SA}[i]}$	LCP[i]
1	9	a	0
2	4	ababba	1
3	6	abba	2
4	1	abbababba	4
5	8	ba	0
6	3	bababba	2
7	5	babba	3
8	7	bba	1
9	2	bbababba	3

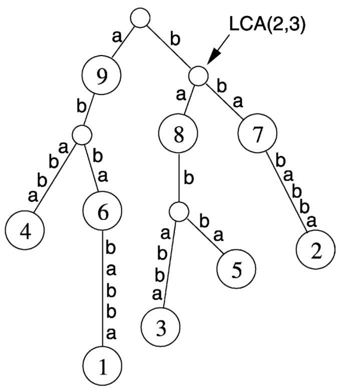


Figure 8: SA, LCP arrays, and suffix tree

5.3 Using RMQ to Solve LCE

Another very natural thought is that since RMQ is so useful to solve the LCA problem, is it possible to use RMQ to solve some similar questions? The fact is that it can also be used to provide a efficient solution of *Longest Common Extensions (LCE)* problems [10].

We just discuss the LCP problem. Actually, the $\textsc{lce}_T(i, j)$ computes the LCP of the suffixes that start at indexes i and j in T. To do this we first compute the suffix array SA, and the inverse suffix array SA^{-1}. We then compute the *LCP-array* giving the LCP of adjacent suffixes in A. Basically, SA describes the order of the suffixes of T, and LCP stores the lengths of the longest common prefixes of T that are consecutive in SA. Then prepare LCP for RMQs. And finally we get the formula, $\textsc{lce}(i, j) = \textsc{rmq}_{LCP}(SA^{-1}[i] + 1, SA^{-1}[j])$. As the RMQ problem can be solved in $O(n)$ time, the LCE problem can also be finished in $O(n)$ time.

In Figure 8, there is a concrete example of the algorithm. It shows the SA, LCP arrays, and suffix tree of the string *abbababba*. Now we show how to get $\textsc{lce}(2, 3)$. $\textsc{lce}(2, 3) = \textsc{rmq}_{LCP}(SA^{-1}[3] + 1, SA^{-1}[2]) = \textsc{rmq}_{LCP}(7, 9) = 1$. We may check the LCP of *bbababba* and *bababba* is b, so the result is correct. Noticed that it is also the depth $|b|$ of the node $\textsc{lca}(3, 2)$ in the suffix tree as indicated in the suffix tree.

5.4 An Error

In the paper, Section 3, first paragraph, the last sentence, it should be $A[M[i, j-1]] \leq A[M[i + 2^{j-1} - 1, j - 1]]$ rather than $A[M[i, j-1]] \leq M[i + 2^{j-1} - 1, j - 1]$.

Acknowledgement

I would first like to thank the lecturer of COMP3011, Professor Jesper Andreas JANSSON, whose clear explanation makes the complex algorithms easy to understand. His prompt and insightful feedback also gave me a lot of help. I really enjoy the course and enjoy the beauty of algorithms!

Appendices

References

[1] M. A. Bender and M. Farach-Colton, "The lca problem revisited," in *Latin American Symposium on Theoretical Informatics.* Springer, 2000, pp. 88–94.

[2] S. Kiefer, "Winter school 2003, st. petersburg," 2003.

[3] S. Alstrup, C. Gavoille, H. Kaplan, and T. Rauhe, "Nearest common ancestors: a survey and a new distributed algorithm," in *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, 2002, pp. 258–264.

[4] H. Aït-Kaci, R. Boyer, P. Lincoln, and R. Nasr, "Efficient implementation of lattice operations," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 11, no. 1, pp. 115–146, 1989.

[5] G.-V. Jourdan, J.-X. Rampon, and C. Jard, "Computing on-line the lattice of maximal antichains of posets," *Order*, vol. 11, no. 3, pp. 197–210, 1994.

[6] M. Morvan and L. Nourine, "Generating minimal interval extensions," *Rapport de recherche*, pp. 92–015, 1992.

[7] D. Gusfield, "Algorithms on stings, trees, and sequences: Computer science and computational biology," *Acm Sigact News*, vol. 28, no. 4, pp. 41–60, 1997.

[8] L. Walsh. rmq. [Online]. Available: https://github.com/leifwalsh/rmq.git

[9] L. Ilie, G. Navarro, and L. Tinta, "The longest common extension problem revisited and applications to approximate string searching," *Journal of Discrete Algorithms*, vol. 8, no. 4, pp. 418–428, 2010.

[10] J. Fischer and V. Heun, "Theoretical and practical improvements on the rmq-problem, with applications to lca and lce," in *Annual Symposium on Combinatorial Pattern Matching.* Springer, 2006, pp. 36–48.