Journal of Computer Science & Its Applications, June 2007, Vol. 14 No. 1

# The Left and Right Extension Problems Revisited

**Ezekiel F. Adebiyi**
Department of Computer and Information Sciences
Covenant University
PMB 1023, Ota, Nigeria.
E-mail: eadebiyi@sdsc.edu

**ABSTRACT.**

***Given a set of alphabets $\Sigma$ , a string S over $\Sigma$ , where the length of S, $|S| = n$, and two equal substrings S[i1; j1] and S[i2; j2], the right (left) extension problem is to find how far (say p) to the right (or left) can we extend S[i1; j1] and S[i2; j2] simultaneously such that the distance between S[j1 + 1; j1+p] and S[j2 +1; j2+p] is q (or the distance between S[i1-p; i1-1] and S[i2-p; i2-1] is d-q). The distances consider here is usually either Hamming or the Edit distances, where $q \in [0; d]$ and $d \geq 0$. This stringology problem finds applications in the bioinformatics problem of repeats extraction[9] and recently in the selection of oligonucleotides for microarray and PCR[2]. Microarray and PCR are both academic and industrial tools of Bioinformatics used in Life Sciences.***

***The left and right extension was first formulated in [9] and solved mainly using the longest common prefix (LCP) technique on the suffix tree. Solution of LCA for the left and right extension problems has its inherent difficulty in that it is difficult to implement and maintain[4]. Recent results that solve LCP via the range minimum query (RMQ) has been proposed[4]. We re-visited in this paper the left and right extension problem by proposing new solution procedures and show how the resulting LCP problem can be solved using the RMQ. The resulting solution is found easy to implement and maintain. Furthermore, we show how the left extension problem can be solved without building the suffix tree for the reversal version of the subject string.***

***Keywords. Stringology, Suffix tree, Longest common ancestor (LCA), Longest common prefix (LCP), Dynamic programming, Range minimum query (RMQ), Repeats, Oligonucleotides.***

## 1.0 Introduction

Formally, the stringology right and left extension problems can be concisely described as follows: Given a set of alphabets $\Sigma$, a string S over $\Sigma$, where the length of S, $|S| = n$, two equal substrings S[i1; j1] and S[i2; j2], positive constant d, the right and left

extension problems require us to compute tables $T_{left}$ and $T_{right}$ of size d + 1 such that

$T_{right}(q) = p$ s.t. $d_H(S[j1 + 1; j1 + p], S[j2 + 1; j2 + p]) = q$ and
$T_{left}(d - q) = p$ s.t. $d_H(S[i1 - p; i1 - 1]; S[i2 - p; i2 - 1]) = d - q$ (1)

for each $q \in [0; d]$, where $d_H$ is the hamming distance. Note that the original formulation by [9] wrongly included the maximum function in the equation (1) given above. For Edit distance, the right and left tables are defined as follows.

$T_{right}(q) = \{(xr,yr) | (xr,yr) \in [j1 + 1; n] \times [j2 + 1; n]$ and is maximal w.r.t $d_E(S[j1 + 1; xr], S[j1 + 1; yr]) \leq q\}$,
$T_{left}(d - q) = \{(xl,yl) | (xl,yl) \in [1; i1 - 1] \times [1; i2 - 1]$ and is maximal w.r.t $d_E(S[1; xl]; S[1; yl]) \leq d-q\}$. (2)

Matrices [j1 +1; n]x[j2 +1; n] and [1; i1 -1]_[1; i2 -1] are Edit distance dynamic programming matrices for strings (S[j1+1; n]; S[j2+1; n]) and (S[1; i1-1]; S[1; i21]). And the pair (xr, yr) is said to be maximal with respect to $d_E(S[j1+1; xr]; S[j2+ 1; yr]) = q$ if and only if

1. $d_E(S[j1 + 1; xr + 1], S[j2 + 1; yr]) > q$ if $xr < n$,
2. $d_E(S[j1 + 1; xr], S[j2 + 1; yr + 1]) > q$ if $yr < n$, and
3. $d_E(S[j1 + 1; xr + 1], S[j2 + 1; yr + 1]) > q$ if $xr < n$ and $yr < n$.

The maximality defined above must also hold for the pair (xl, yl) appropriately. Considering the simple unit measure, the Hamming distance between two strings S[1; i] and S[1; j], $\delta(i,j)$ is 0 if S[i] = S[j] and 1 otherwise. For Edit distance,

$\delta(i,j) = \min[\delta(i - 1, j) + 1, \delta(i, j - 1) + 1; \delta(i - 1, j - 1) + t(i, j)]$. (3)

where t(i,j) is 0, if S1(i) = S2(j) else 1. For Hamming distance, Tables $T_{right}$ and $T_{left}$ can be computed in O(d) time, using the suffix tree that allows the constant time computation of the length of the LCP of two substrings of S[7, 13]. Tables $T_{right}$ and $T_{left}$ under the Edit distance cost $O(d^2)$ time using the longest common prefix technique and the computation of front(d) of the DP-matrix[15]. But the ideas behind the computation of LCP using the suffix tree in constant time has been found to be unwieldy and un-implemental. The Range Minimum Query (RMQ) has been proved to provide us with a simple algorithm to compute.the LCP using the su_x tree[4]. We couple this idea with some other pattern matching techniques to design simple O(d) and $O(d^2)$ algorithms for the left and right extension problems under the Hamming and Edit distance metrics. Our resulting algorithm has been used in Adebiyi[2] and Adebiyi and Olarenwaju[3] to design efficient sequential and parallel algorithm for oligonucleotides selection.

### 1.1 Organization of the Article

This paper is structured as follows. Section 1 contains the introduction. Section 2 presents the suffix tree, the lowest common ancestor (LCA) problem and how using the using the suffix tree, the LCA problem is equivalent to the LCP problem and given two nodes, how LCP can be done in constant time. Section 2 is intended to show the reader how unwiedy and un-implemental LCP computation via the suffix tree could be. And we made attempt also in this section to simplify the implementation of this technique by presenting another look of a concise

design of the algorithms involved. In section 3, we present a new technical exposition of the RMQ problem equivalence to the LCA problem, and how the RMQ problem can be solved in constant time. Using precious sections ideas, our new solutions to the left and right extension problems is presented in section 4. Lastly, section 5 contains our conclusion.

## 2.0 Suffix tree and LCA problem

Some good materials on suffix tree can be found in Adebiyi[1] and Gusfield[6]. We give a brief description below:

A suffix tree is a lexicographically inter connected data structure, that provide efficient access to all substrings of a strings, over which it is built. This data structure can be constructed and represented in linear time and space. And this has enabled the solution of many strings problem in linear time. The construction of a suffix tree in linear time can be found in Weiner[16], McCreight[11] and Ukkonen[14]. A recent paper by Kurtz[8] discussed how an economical construction of suffix tree with respect to space can be done.

We give here description of a suffix tree for an arbitrary string x of length n over an alphabet $\Sigma$ using [1, 6]. We follow this to show how a suffix tree can be built for a set of strings.

**Definition 1** A suffix tree of a n-character string x is a rooted directed tree with exactly n leaves, numbered 1 to n. Each internal node, other than the root, has at least two children and each edge is labelled with a nonempty substring of x. No two edges out a node can have edge-labels beginning with the same characters. The key feature of the suffix tree is that for any leaf i, the label of the path from the root to leaf i exactly spells out the suffix of x that start at position i.

Note that the definition above does not guarantee the existence of a suffix tree for any string x. The problem is that if a prefix of a suffix of x matches a suffix of x, the path for the later suffix would not end at a leaf. Therefore, to guarantee the existence of a suffix tree for any string x, we place at the end of x a special symbol that is not in the alphabet $\Sigma$.We use in this paper, the symbol \$, for the termination character. Below is a suffix tree for x = GTATCTAGG. The number at the leaves indicate the starting position of the corresponding suffixes.

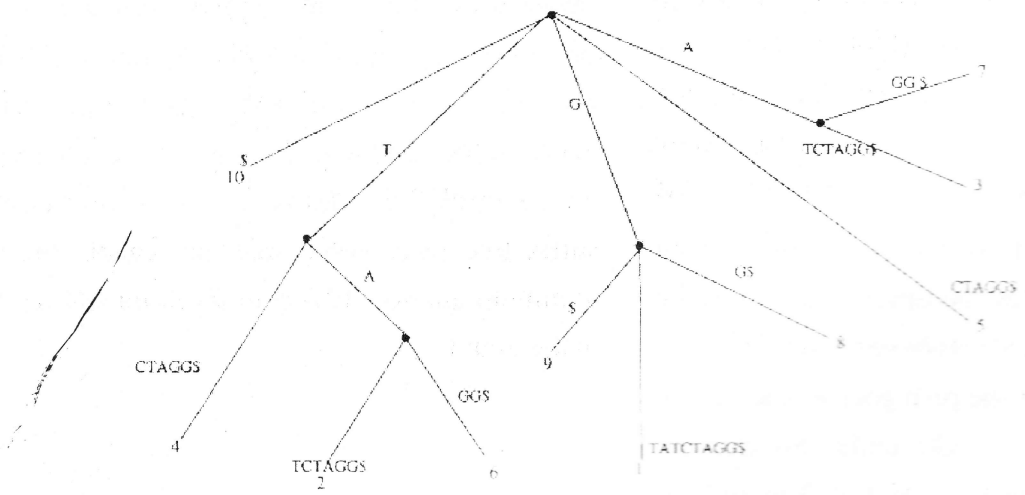


Fig. 1. The suffix tree for $x = GTATCTAGG$

To describe the lowest common ancestor problem, let us take the following definitions from Gusfield[6].

**Definition 2** Consider a rooted tree T, a node u is an ancestor of a node v if u is on the unique path from the root to v and a proper ancestor of v refers to an ancestor that is not v.

**Definition 3** In T, the lowest common ancestor (lca) of two nodes x and y is the deepest node in T that is an ancestor of both x and y.

The following amazing result that helps to solve the LCP in constant time is summarized in the theorem that follows.

**Theorem 1** After a linear amount of pre-processing of a rooted tree, the LCA of any two nodes can be found in constant time[7, 13].

This result finds application in prescribing solution for the LCP problem in that the suffix tree connects strings to the LCA problem, so that the LCA of leaves i and j identifies the LCP of suffixes i and j. We have summarized in the following the proof of Schieber and Vishkin[13] (reformulated using Gusfield[6]) for theorem 1 above.

Let B be a rooted complete binary tree with p leaves (n = 2p - 1 nodes in total), then every internal node has exactly two children and the number of edges on the path from the root to any leaf in B is $d = \log_2 p$. Furthermore, let node v of B be assigned a d + 1 bit number, called its path number, that encodes the unique path from the root to v. A 0 for the *ith* bit from the left indicates that the *ith* edges on the path goes to a left child, and a 1 indicates a right child. So that the extracted path of node v in fig. 2(a) will be encoded 0010100 and the root will be 1000000, if d = 6 and we padded out to d + 1 bits by adding a 1 to the right of the path bits followed by as many additional 0s as needed to make d + 1 bits.

Fig. 2(b) is an example of complete binary tree. Note that the nodes are numbered as they are encountered in a depth-first traversal. Let us assume that we can execute the XOR of two binary numbers of size O(log n) in constant time, shift a binary number (left or right) by up to O(log n) bits in constant time and find the position of the left-most or right-most 1-bit in a binary number in a constant. Note that the XOR of two bits is 1 if and only if the bits are different, and the XOR of two d + 1 bit numbers is obtained by independently taking the XOR of each bit of the two numbers. For example, the XOR of 00101 and 10011 is 10110. Therefore, using the XOR operation, in a complete binary tree, we can find lca(i; j), where i and j are nodes as follows, in constant time in fig. 3.

Using this algorithm, from fig. 2(b), it is easy to find in constant time that the lca(5 (101), 7 (111))= 6 (110). To apply the above results to a general rooted tree T, we need to map the nodes of T to the nodes of B, in such a way that LCA retrievals on B will provide information to solve LCA queries on T . Note that, although quite unwieldy and maybe un-implemental, a linear time pre-processing task can be perform on T to map T to B and the first of all these is traversing T in a depth-first manner, numbering the nodes in the order that they are first encountered in the traversal. This idea is also use in converting the suffix tree to a rooted tree that enable the range minimum queries. This actually complete the proof of theorem 1.

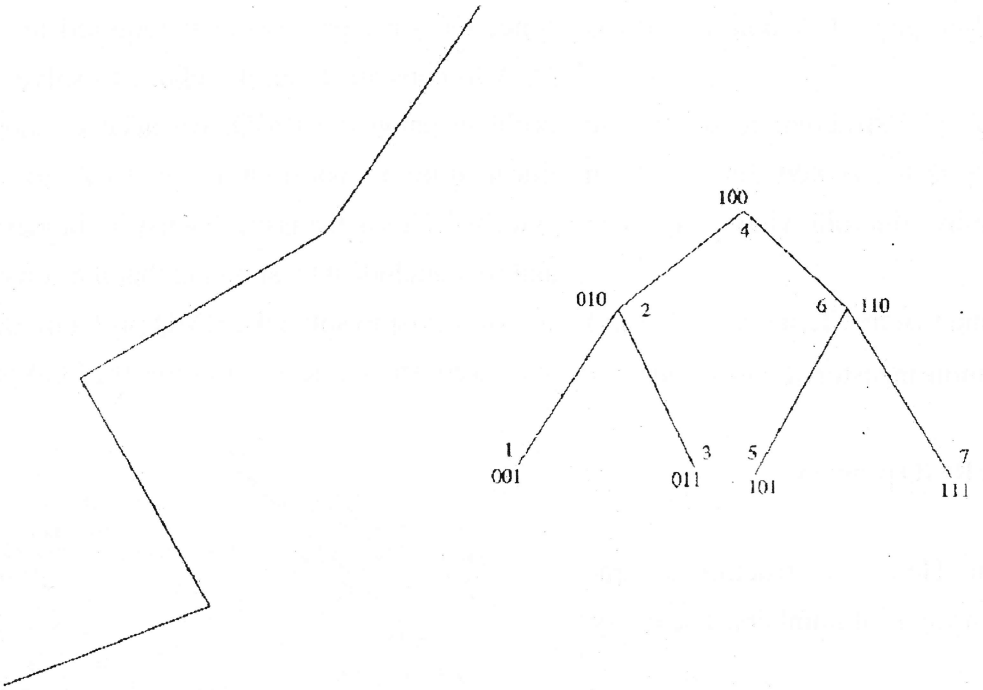


**Fig. 2.** a) The 0010 path, b) A binary tree with four leaves. The path numbers are written both in binary and in base ten.

1. BINARY-TREE-LCA($i, j$)
2. $r \leftarrow$ XOR($i, j$)
3. $k \leftarrow$ find the left-most 1-bit($r$)
4. shift $i$ right by $d + 1 - k$ places,
5. set the right most bit to a 1, and
6. shift it back left by $d + 1 - k$ places
7. return the resulting new $i$

**Fig. 3.** The constant time algorithm that find LCA in a preprocessed complete binary tree.

### 3.0 The LCA and RMQ problems

The new technical exposition given in this section is based on the work of Kiefer[10].

To show the direct application of RMQ to solving the LCA in a suffix tree (and not in a general rooted tree), we use the suffix tree of GTATCTAGG in fig. 1 in the following. Pre-processing the suffix tree as described above, that is, number the nodes in the order that they are first encountered in the traversal. The resulting suffix tree is shown in fig. 4 below.

For clarity, we define the LCA and the RMQ problems.

**The LCA problem.** The structure to query and therefore pre-process is the rooted tree T with n nodes. The LCA requires the following query to be solved.

Query: For nodes u and v of tree T, query $LCA_T(u,v)$ returns the least common ancestor of both u and v.

We now take next the RMQ problem.

**The RMQ problem**. Here, the structure to pre-process is a length n array L of numbers. The query required is

Query: For indices i and j between 1 and n, query RMQL(i,j) returns the index of the smallest element in the subarray L[i...j].

We adopt the following notation to state clearly the time needed for pre-processing and the time for performing the actual query given. We therefore write that if an algorithm has pre-processing time f(n) and query time g(n), we then write that the algorithm has the complexity <f(n), g(n)>.

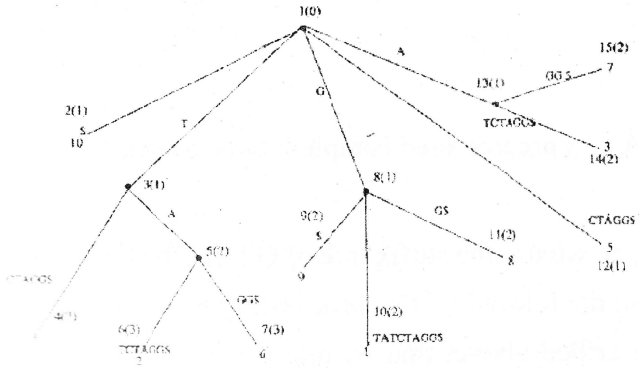

Fig. 4. A $O(n)$ preprocessed suffix tree

Since O(n) pre-processing is required to solve the LCA in constant time, therefore, to solve the LCA problem using the RMQ, we need to show that a linear time transformation of LCA to RMQ is possible. This is the issue discuss in the next section and we conclude it by showing that if a constant time solution exist to solve the RMQ problem, then there exist, a constant time then to solve the LCA problem.

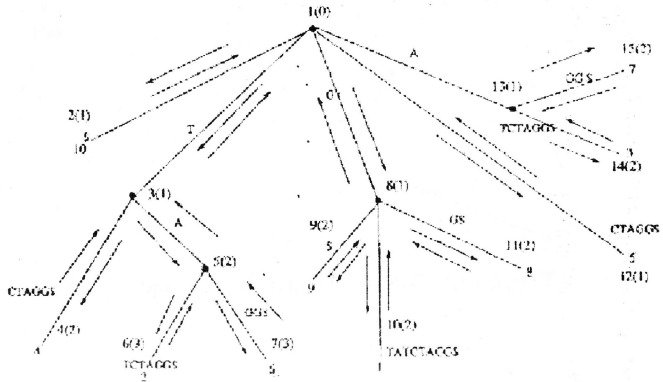

**Fig. 5.** An Euler tour of the preprocessed suffix tree.

### 3.1 A Linear Reduction from LCA to RMQ

Since the number of a node in a suffix tree is bounded from above by O(n)[5], a linear time reduction will be possible via doing some tree traversal on the tree. Thus, to have a detailed link of how the nodes are connected via ancestors, a depth first search on the tree to produce an Euler tour is more appropriate. For the suffix tree in fig. 5, this Euler tour can be shown in an array E as follows. Additionally, we store the various level, each node is, in the tree, in an array L. These levels are shown in brackets behind the Euler tour numbering.

**Table 1.** The Euler tour of the suffix tree in fig. 5 with additional level information.

| Index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| array $E$ | 1 | 2 | 1 | 3 | 4 | 3 | 5 | 6 | 5 | 7 | 5 | 3 | 1 | 8 | 9 | 1 | 2 | 1 | 2 | 1 | 0 | 1 | 0 | 1 | 2 | 1 | 2 | 1 | 0 |
| array $L$ | 0 | 1 | 0 | 1 | 2 | 1 | 2 | 3 | 2 | 3 | 2 | 1 | 0 | 1 | 2 | 1 | 2 | 1 | 2 | 1 | 0 | 1 | 0 | 1 | 2 | 1 | 2 | 1 | 0 |

Furthermore, to enable us know the specific interval to use in our attempt to use the RMQL to solve the $LCA_T$, we need to compute the first occurrences[1] of all nodes in a Euler tour. This is given in an array R of length n below.

**Table 2.** The first occurrence array for all nodes.

| node | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| array $R$ | 1 | 2 | 4 | 5 | 7 | 8 | 10 | 14 | 15 | 17 | 19 | 22 | 24 | 25 | 27 |

From the L array above, note that

**Observation 1** The LCA of nodes u and v is the shallowest node encountered between the visits to u and to v during a depth first search traversal of T .

[1] This must not be a first occurrences, it may be in-fact any occurrences

Using the above observation, and the foregone arguments, it is now easy to prove the theorem that follows.

**Theorem 3** If there is an <O(n);O(1)>-solution for RMQ, then there is an <O(n);O(1)>-solution for LCA.

**Proof.** We first show that the LCA problem can be reduce in linear time to the RMQ problem and the solution of the RMQ is the current node expected out for the LCA problem. We conclude the proof by showing how the RMQ after the O(n) pre-processing time can be solve in O(1) time.

The reduction is as follows. Given the tree T, compute the arrays E, L, and R as we have discussed above. Let assume that we can preprocess L in linear time. To deal with the reduction problem of translating the LCA to a RMQ problem. Note that the initial problem, we have is a $LCA_T(u,v)$. Now

- 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]]).
- The shallowest node in this sub-tour is at index $RMQ_L(R[u],R[v])$, 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. This is in line with observation 1 above.
- The node at this position is $E[RMQ_L(R[u],R[v])]$, which is thus the output of $LCA_T(u,v)$.

And thus, we can write that $LCA_T(u,v) = E[RMQ_L(R[u],R[v])]$. Note that the query time is simply O(1), since the LCA query in this reduction uses one RMQ query in L and three array references R[u], R[v] and $E[RMQ_L(R[u],R[v])]$ at O(1) time each. Therefore, we have a <O(n);O(1)>-solution for LCA.

For the solution to the general RMQ problem, the reader is referred to [4] and [10]. To end this summary, we will encapsulate below the solution to a special case of the RMQ problem, the ±RMQ problem. The solution to the ±RMQ problem is sufficient to apply the longest common prefix solution required to solve the left and right extensions problem revisited in this paper. This special case is as result of the +1 or -1 relationship in the adjacent elements of the array L.

### 3.2 An <O(n), O(1)>-Algorithm for ± RMQ

To represent the <O(n),O(1)>-algorithm for ±RMQ of [10], we need as subroutine a Sparse Table (ST) algorithm that was used to derive a <O(n log n),O(1)>-solution for the general RMQ. This algorithm is as follows.

The ST algorithm is based on dynamic programming. Here, we pre-compute each query whose length is a power of two and store our result in a matrix M, so that M[i; j] = m, such that L[m] = min L[I,....i+2 -1]. Note that matrix M has size O(n log n) and for every i between 1 and n and every j between 1 and log n, each entry of matrix M store the minimum element in the block starting at i and having length 2j . Matrix M is filled dynamically using the following equation:

$$M[i,j] = \begin{cases} M[i,j-1] & \text{if } L[M[i,j-1]] \le L[M[i+2^{j-1},j-1]], \\ M[i+2^{j-1},j-1] & \text{otherwise.} \end{cases} \quad (4)$$

Using M, the following algorithm (see fig. 6), the ST algorithm compute an arbitrary $RMQ_L(i,j)$ by comparing the minimums of the two blocks that cover the sub-range i to j. The two blocks i to i+2 -1 and j -2 +1 to j are obtained via the largest block of size 2k, $k = \lfloor \log(j-i) \rfloor$ that fit this sub-range.

Using the algorithm in fig. 6, we now present the required <O(n);O(1)>-algorithm for ± RMQ. Above, the ST algorithm is based on a table of size n log n. The idea of the algorithm that follows is a further application of the divide-and-conquer technique to pre-compute answers on small sub-arrays thus removing the log factor from the pre-processing. To this end, an array L' of size 2n/log n is constructed, that stores minimums of blocks of size log n/2 of L. An array L" is use to keep track of

1. Sparse Table Algorithm($L$)
2. fill Matrix $M$ using (4)
3. for arbitrary pair $(i,j)$, compute $RMQ_L(i,j)$:
4. $\quad k = \lfloor \log(j-i) \rfloor$
5. $\quad$ get $m_1 = M(i,k)$
6. $\quad$ get $m_2 = M(j-2^k+1,k)$
7. $\quad$ output $\min\{m_1, m_2\}$

**.6. The $(O(n \log n), O(1))$-algorithm for computing an arbitrary $RMQ_L(i,j)$ using (4) above.**

where each of the minima in L' came from. We name the resulting <O(n),O(1)>- algorithm Optimal-$RMQ_L$-compute(L) and our version is given below in fig. 7.

To simplify our presentation, we did not include the details as regard the use of normalization property between blocks in order to reduce the hidden constant in the linear time required to preprocess blocks for the in-blocks RMQs required in steps 9, 11, and 12. But the normalization idea used is straight-forward and can be found in [10, 4].

## 4.0 Our New Solutions for the Left and Right Extension Problems

Two basic new solutions simplify the formal solution for the left and right problems.
They are

1. the LCP solution via the RMQ, and

1. Optimal$RMQ_L$-compute($L$)
2. partition $L$ into blocks of size $\log n/2$
3. for $i = 1 \ldots 2n/\log n$, $j = 1 \ldots \log n/2$
4. compute $M[i,j]$ using (4)
5. define array $L'[1 \ldots 2n/\log n]$
6. define array $L''[1 \ldots 2n/\log n]$
7. for arbitrary pair $(i,j)$ s.t $i < j$, compute $RMQ_L(i,j)$:
8. if(check-if-same-block)
9. output($RMQ_L(i,j)$)
10. else
11. $m_1 = RMQ_L(L'[i], L[i + \frac{\log n}{2} - 1])$
12. $m_2 = \min\{L'[i], L'[j]\}$
13. $m_3 = RMQ_L(L'[j], L[j + \frac{\log n}{2} - 1])$
14. output $\min\{m_1, m_2, m_3\}$

**Fig. 7.** The $\langle O(n), O(1)\rangle$- algorithm for computing an arbitrary $RMQ_L(i,j)$ using (4) above.

2. the new simpler problem formulation to solve the right extension problem, instead of increasing the hidden constant through the building of a reversed string suffix tree.

We take turn in the following sections to present our solution to the extension problems in (1) and (2).

## 4.1 Left and Right Extension Problem under the Hamming distance

Basically, tables $T_{right}$ and $T_{left}$ can be computed in O(d) using the LCP deduced from the suffix tree for the two substrings involved. This is done as follows. For q = 0, for the right extension, we need to find the LCP of suffixes j1 +1 and j2 +1, let this be p1 and then for q = 1 (that is a mismatch is allow), we need to find the LCP (say p2) between suffixes j1 +p1 +1 and j2 +p1 +1 and this also can be done in constant time, so that the new longest common prefix will end at j1 + p1 + p2 and j2 + p1 + p2. We will continue this until we reach q = d (mismatch allowed is d). The essence of sections 2 and 3 is to show how we can simplify the solution of the LCP problem using the RMQ. Therefore, for the required LCPs in the solution prescribed above, we make use of Optimal-$RMQ_1$-compute(L) of fig. 7, saving a lot of overhead in terms of space and complexity as regard implementation.

To obtain relevant LCPs to solve the right extension problem, a naive solution required that we construct the suffix tree for the reverse-string of the subject string[6].
But we can avoid this by noting that the problem solved in $T_{left}$, can be reformulated as determining i'1, i'2, such that their LCP with Hamming difference allowed as required ends at i1, i2. In other words, determine p, such that H(x[i1 - p, i1 - 1], x[i2 - p, i2 - 1]) = q and i'1 + p = i1 and i'2 + p = i2. A multiple application of RMQ will solve this, yielding to a O(d) solution.

### 4.2 Left and Right Extension Problem under the Edit distance

The formal method prescribed the use of computing $T_{left}$ and $T_{right}$ using the front(d) of the DP-matrix[15] in O(dn) time. It is also important to note that using the spelling approach of Sagot[12], a new solution that runs in O(n) can be obtained.

And this can be improved but with much overhead by combining the front(d) of the DP-matrix technique with the LCP technique in O(d2) time. Also here, the use of the RMQ solution and an important observation that avoid building reversal string suffix tree have decrease significantly the overhead normally experienced in the $O(d^2)$ method.

### 5.0 Conclusion

The new solutions discussed in this paper has been used in the implementation of an algorithm for oligonucleotides selection in an expressed sequence tag sequences[2]. The formal solution was used in the selection of repeats in DNA sequences[9]. Our experimental experience has thus shown that true to the wide spread notion, the formal solution implementation is unwieldy and hard to implement and our new techniques using the RMQ is void of the overhead challenges, in term of memory and high hidden constant value in the complexity that are paramount in the implementation of the formal method. Finally, the technical exposition given in section 3 of this paper is new and will help in the easy implementation of the RMQ solution generally.

### Acknowledgment

We thank the anonymous referee of our paper accepted at the Intl Workshop and Conf. on new trends in the Mathematical and Computer Science with application to real world problems, June 19-23, 2006, whose careful review work necessitate the new technical exposition in this paper. This work is partial supported by the Covenant University Senate Research Grant 2004/2005. Part of this work was done, while the author was at LIRMM, France on a CNRS-NEPAD special grant.

### References

1. Adebiyi, E. F. (2002) Pattern Discovery in Biology and String Sorting: Theory and Experimentation. Shaker Publisher, Aachen, Germany.
2. Adebiyi, E. F. (2006) Using Suffix tree for efficient selection of unique oligos for large EST databases, Submitted to WABI 2006 and International Journal of Bioinformatics and Computational Biology (IJBCB)
3. Adebiyi, E. F and Oyelade, J. O. (2006) A Comparative Analysis of existing Oligonucleotides Selection Algorithms and Optimal Parallel Oligos Selection for large EST Databases (Extended Abstract). Accepted (peer reviewed) proc. of the Intl Workshop and Conf. on new trends in the Mathematical and Computer Science with application to real world problems, June 19-23, 2006.
4. Bender, M. A. and Farach-Colton, M. The LCA problem revisited, 88-94, LATIN 2000.
5. A. Blumer A. Ehrenfeucht, and D. Haussler. Average Size of Suffix trees and DAWGS. Discrete Applied Mathematics, 24, 37-45, 1989.
6. Gusfield, D. Algorithms on strings, trees and sequences. Cambridge University Press, New York, 1997.

7. Harel, D., and Tarjan, R. (1984) Fast Algorithms for Finding nearest Common Ancestors. SIAM J. Computing 13: 338-355.
8. Kurtz, S. (1999) Reducing the Space Requirement of Suffix Trees. Software-Practice and Experience, 29(13):1149-1171.
9. Kurtz, S. and Ohlebusch, E., Schleiermacher, C., Stoye, J. and Giegerich, G. Computation and Visualization of degenerate Repeats in complete Genomes. Proceeding of international conference of ISMB, 228-238, 2000.
10. Kiefer, S. (2003) The Least Common Ancestor Problem. Presented at the Winter School 2003, St. Petersburg.
11. McCreight, E. M. (1976) A Space-Economical Suffix Tree Construction Algorithm. Journal of ACM 23(2): 262-272.
12. Sagot, M-F. Spelling Approximate Repeated or Common Motifs using a Suffix tree. LNCS 1380, 111-127, 1998.
13. Schieber, B., and Vishkin, U. (1998) On Finding Lowest Common Ancestors. SIAM J. Computing 17(6): 1253-1263.
14. Ukkonen, E. (1995) On-line Construction of Suffix Trees. Algorithmica, 14:249-260.
15. Ukkonen, E. (1985) Algorithms for Approximate String Matching. Information and Control, 64:100-118.
16. Weiner, P. (1973) Linear Pattern Matching Algorithm. Proc. 14th IEEE Sym. on Switching and Automata theory, 1-11.
17. Zheng, J., Close, T., Jiang, T., and Lonardi, S. (2003) Efficient Selection of Unique and Popular Oligos for Large EST Databases. CPM 2003, LNCS 2676, 384-401.
18. Zheng, J., Close, T., Jiang, T., and Lonardi, S. (2004) Efficient Selection of Unique and Popular Oligos for large EST Databases. Bioinformatics, 20(13), 2101-2112.