Computing a Longest Common Subsequence of two strings when one of them is Run Length Encoded

SHEGUFTA BAKHT AHSAN[1]
TANAEEM M. MOOSA[2]
M. SOHEL RAHMAN[2]
SHAMPA SHAHRIYAR[1]

AℓEDA Group, Department of CSE
Bangladesh University of Engineering and Technology
[1] (plaban777,shampa077)@gmail.com
[2] (tanaeem,msrahman)@cse.buet.ac.bd

Abstract. Given two strings, the longest common subsequence (LCS) problem computes a common subsequence that has the maximum length. In this paper, we present new and efficient algorithms for solving the LCS problem for two strings one of which is run length encoded (RLE). We first present an algorithm that runs in $O(gN)$ time, where g is the length of the RLE string and N is the length of uncompressed string. Then based on the ideas of the above algorithm we present another algorithm that runs in $O(\mathcal{R}\log(\log g)+N)$ time, where $\mathcal{R}$ is the total number of ordered pairs of positions at which the two strings match. Our first algorithm matches the best algorithm in the literature for the same problem. On the other hand, for $\mathcal{R} < gN/\log(\log)g$, our second algorithm outperforms the best algorithms in the literature.

Keywords: algorithms, longest common subsequence, run length encoded strings.

(Received May 15th, 2011 / Accepted September 1st, 2011)

1 Introduction

The *longest common subsequence* (LCS) problem is a classic and well-studied problem in computer science with extensive applications in diverse areas ranging from spelling error corrections to molecular biology. For example, the task of spelling error correction is to find the dictionary entry which resembles most a given word. In order to save storage a file archive of several versions of a source program is maintained compactly by storing only the original version and the differences of subsequent versions with the previous ones. In molecular biology [19, 1], we want to compare DNA or protein sequences to learn how homologous they are. All these cases can be seen as an investigation for the *'closeness'* among strings. And an obvious measure for the closeness of strings is to find the maximum number of common symbols in them preserving the order of the symbols. This is known as the longest common subsequence of two strings.

Suppose we are given two strings $X[1..N] = X[1]X[2] \ldots X[N]$ and $Y[1..G] = Y[1]Y[2] \ldots Y[G]$. Without the loss of generality, we can assume that $N \leq G$. A subsequence $S[1..R] = S[1]S[2] \ldots S[R], 0 < R \leq N$ of X is obtained by deleting $N - R$ symbols from X. A common subsequence of two strings X and Y is a subsequence common to both X and Y. The longest common subsequence problem for two strings, is to find a common subsequence in both the strings, having the maximum possible length. We use $lcs(X, Y)$ and $r(X, Y)$ to denote a longest common subsequence of X and Y and its length, respectively.

The classic dynamic programming solution to LCS problem, invented by Wagner and Fischer [22], has

$O(NG)$ worst case running time. Masek and Paterson [15] improved this algorithm using the "Four-Russians" technique [4] to reduce the worst case running time to $O(NG/\log N)$. Since then, not much improvement in terms of N, G can be found in the literature. However, several algorithms exist with complexities depending on other parameters. For example, Myers in [17] and Nakatsu et al. in [18] presented an $O(ND)$ algorithm, where the parameter D is the simple Levenshtein distance between the two given strings [13].

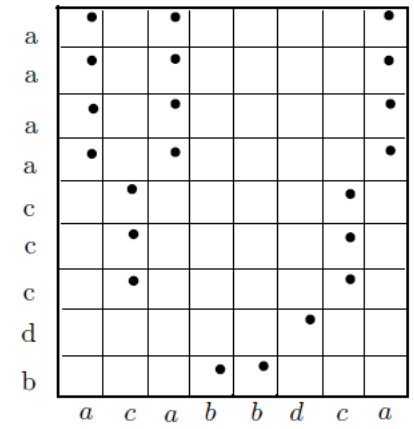
Another interesting and perhaps more relevant parameter for this problem is $\mathcal{R}$, which is the total number of ordered pairs of positions at which the two strings match. Hunt and Szymanski [8] presented an algorithm to solve the LCS problem in $O((\mathcal{R}+N)\log N)$ time. They also cited applications, where $\mathcal{R} \sim N$ and thereby claimed that for these applications the algorithm would run in $O(N \log N)$ time. Very recently, Iliopoulos and Rahman [10, 11] presented an efficient algorithm to solve the LCS problem in $O(\mathcal{R}\log(\log(N))+N)$ time.

1.1 LCS for RLE Strings

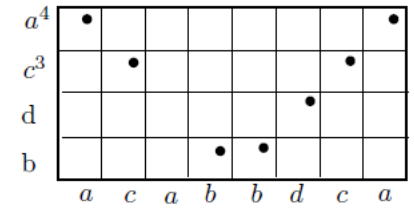
In this paper, we are interested to compute an LCS when one of the strings is run length encoded. The motivation for using compressed strings as input comes from the huge size of biological sequences. Here we will be focusing on the run-length encoded [12] strings. In a string, the maximal repeated string of characters is called a run and the number of repetitions is called the run-length. Thus, a string can be encoded more compactly by replacing a run by a single instance of the repeated character along with its run-length. Compressing a string in this way is called run-length encoding and a run-length encoded string is abbreviated as an RLE string.

In what follows, we use the following convention: if X is a (uncompressed) string, then the run length encoding of X will be denoted by $\widetilde{X}$. For example, the RLE string of $X = bdcccaaaaaa$ is $\widetilde{X} = b^1d^1c^3a^6$. Note that for $\widetilde{X}$, we define $\widetilde{X}[1] = b^1, \widetilde{X}[4] = a^6$ and so on. The notation $|X|$ is used to denote its usual meaning, i.e., the length of X; the length of the corresponding RLE string $\widetilde{X}$ is denoted by $|\widetilde{X}|$. We will use small letters to denote the length of an RLE string; whereas capital letters will be used to denote the length of an uncompressed string. For example, if $|X| = N$, then we shall use n to denote the length of $\widetilde{X}$.

Note that, the notion of a match and hence the definition of the set of matches, $\mathcal{M}$, can be extended in a natural way when one or both of the strings involved is/are run length encoded. For example, the notion of a match $(i, j) \in \mathcal{M}$, is extended when one input is an RLE string as follows: if $\widetilde{Y}[i] = a^q$ and $X[j] = a$ then we say $(i, j) \in \mathcal{M}$ and $run((i, j)) = q$. The set $\mathcal{M}$ as well as its size in two different contexts are illustrated in Figure 1. In particular, Figure 1(a) considers two normal strings and Figure 1.(b) illustrates the scenario when one of those is run length encoded. The problem we handle in this paper is formally defined as follows:



(a) The set $\mathcal{M}$ when both strings are uncompressed. Here, $R = 21$



(b) The set $\mathcal{M}$ when one of the strings is run length encoded. Here, $R = 7$

Figure 1: The set of matches $\mathcal{M}$ in two different settings. A dot in a cell indicates a match.

Problem 1. Problem LCS_RLE. *Given one uncompressed string $X[1..N] = X[1]X[2] \ldots X[N]$ and one RLE string $\widetilde{Y}[1..g] = \widetilde{Y}[1]\widetilde{Y}[2] \ldots \widetilde{Y}[g]$, we want to compute a Longest Common Subsequence (LCS) of X and $\widetilde{Y}$.*

We will use $LCS\_RLE(X, \widetilde{Y})$ to denote an LCS of X and $\widetilde{Y}$. There has been significant research on solving the LCS problem involving RLE strings in the literature. Mitchell proposed an algorithm [16] capable of computing an LCS when both the input are RLE strings. Given two RLE strings $\widetilde{X}[1..n]$ and $\widetilde{Y}[1..g]$, Mitchell's algorithm runs in $O((\mathcal{R}+g+n)\log(\mathcal{R}+g+n))$ time. Apostolico et al. [3] gave another algorithm

for solving the same problem in $O(gn\log(gn))$ time whereas the algorithm of Freschi and Bogliolo [6] runs in $O(gN + Gn - gn)$ time. Ann et al. also proposed an algorithm to compute an LCS of two run length encoded strings [2] in $O(gn + \min\{g_1, g_2\})$ where g_1, g_2 denote the number of elements in the bottom and right boundaries of the matched blocks respectively. The version of the problem where only one string is run length encoded was handled recently by Liua et al. in [14]. Here, the authors proposed an $O(gN)$ time algorithm to solve the problem.

1.2 Our Contribution

In this paper, we make an effort to solve the LCS problem efficiently when one of the input strings is run length encoded. Our main contributions are two novel efficient algorithms, namely, LCS_RLE-I and LCS_RLE-II, to solve Problem 1. In particular, we first present a novel and interesting idea to solve the problem and present an algorithm that runs in $O(gN)$ time (LCS_RLE-I). This matches the best algorithm in the literature [14] for the same problem. Subsequently, based on the ideas of our above algorithm, we present another algorithm that runs in $O(\mathcal{R}\log(\log(g)) + N)$ time (LCS_RLE-II). Clearly, for $\mathcal{R} < gN/\log(\log(g))$, our second algorithm outperforms the best algorithms in the literature. In this context, Algorithm LCS_RLE-II is an input sensitive algorithm. In many cases, the input could be such that $\mathcal{R} = o(gN)$. In such cases, our algorithm will definitely show better behaviour than the other algorithms. Also, note that, in our setting, Mitchell's algorithm would run in $O((\mathcal{R} + G + n)\log(\mathcal{R} + G + n))$ time, which clearly is worse than ours. (Notably, Mitchell's algorithm could also be used in our setting with an extra preprocessing step to compress the uncompressed string. In this case, the cost of compression must be taken into account.)

With the existence of LCS algorithms in the literature that can deal with both RLE strings, our algorithms may seem to be only theoretically interesting. However, we note the following points in favour of the practical importance of our algorithms. Firstly, in many practical instances a much smaller reference pattern is compared with a large genome. In such cases our version of the problem may turn out to be more relevant. Secondly, when we talk about comparing two RLE genomes, we often ignore the cost of compressing the two genomes. Now, if we count the cost of compression, then our algorithms (and the algorithm of Liua et al. in [14]) may turn out to be more favourable in different practical settings. Finally, our work can be seen as a building block for an efficient algorithm for the version where both strings are RLE. Indeed, we believe that combining some of the tricks of Mitchell [16] with our work, we would be able to get an algorithm for two RLE strings that runs faster than Mitchell's algorithm.

1.3 Roadmap

The rest of the paper is organized as follows. In Section 2, we present an $O(gN)$ algorithm, namely LCS_RLE-I, to solve Problem LCS_RLE. LCS_RLE-I provides the base of our second algorithm, LCS_RLE-II, described in Section 3. We achieve $O(\mathcal{R}\log\log g + N)$ running time for LCS_RLE-II. Finally, we briefly conclude in Section 4.

2 A New Algorithm

In this section, we present Algorithm LCS_RLE-I which works in $O(gN)$ time. Since our algorithm depends on some ideas of Algorithm LCS-I of [10, 11], we give a very brief overview of LCS-I in the following subsection.

2.1 Review of LCS-I

Note that, LCS-I solves the classic LCS problem for two given strings X and Y. For the ease of exposition, and to remain in line with the description of [10, 11], while reviewing LCS-I (in this section) we will assume that $|X| = |Y| = N$. Recall that, we say a pair (i, j), $1 \le i \le N, 1 \le j \le G$, defines a match, if $X[i] = Y[j]$. The set of all matches, $\mathcal{M}$, is defined as follows:

$$\mathcal{M} = \{(i, j) \mid X[i] = Y[j], 1 \le i \le N, 1 \le j \le G\}.$$

Observe that $|\mathcal{M}| = \mathcal{R}$. From the definition of LCS it is clear that if $(i, j) \in \mathcal{M}$, then we can calculate $\mathcal{T}[i, j], 1 \le i, j \le N$ by employing the Equation 1 from [20, 9].

$$\mathcal{T}[i,j] = \begin{cases} \text{Undefined} & \text{if } (i,j) \notin M, \\ 1 & \text{if } (i = 1 \text{ or } j = 1) \text{ and } (i,j) \in M \\ \max\limits_{\substack{1 \le \ell_i < i \\ 1 \le \ell_j < j \\ (\ell_i, \ell_j) \in M}} \{(\mathcal{T}[\ell_i, \ell_j])\} + 1 & \text{if } (i, j \ne 1) \text{ and } (i,j) \in M. \end{cases} \quad (1)$$

Here we have used the tabular notion $\mathcal{T}[i,j]$ to denote $r(Y[1..i], X[1..j])$. We use the notation $\mathcal{M}_i$ to denote the set of matches in Row i. Also, for the sake of better exposition we impose a numbering on the matches of a particular row from left to right as follows. If we have $\mathcal{M}_i = \{(i,j_1),(i,j_2),\ldots,(i,j_\ell)\}$, such that $1 \leq j_1 < j_2 < \ldots < j_\ell$, then, we say that $number((i,j_q)) = q$ and may refer to the match (i,j_q) as the qth match in Row i. Note that, $number((i,j_q))$ may or may not be equal to j_q.

In what follows, we assume that we are given the set $\mathcal{M}$ in the prescribed order assuming a row by row operation. LCS-I depends on the following facts, problem and results.

Fact 1. ([9, 20]) *Suppose $(i,j) \in \mathcal{M}$. Then for all $(i',j) \in \mathcal{M}, i' > i$, (resp. $(i,j') \in \mathcal{M}, j' > j$), we must have $\mathcal{T}[i',j] \geq \mathcal{T}[i,j]$ (resp. $\mathcal{T}[i,j'] \geq \mathcal{T}[i,j]$).* □

Fact 2. ([9, 20]) *The calculation of the entry $\mathcal{T}[i,j], (i,j) \in \mathcal{M}, 1 \leq i,j \leq n$, is independent of any $\mathcal{T}[\ell,q], (\ell,q) \in \mathcal{M}, \ell = i, 1 \leq q \leq N$.* □

Problem 2. Range Maxima Query Problem. *We are given an array $A = a_1 a_2 ... a_n$ of numbers. We need to preprocess A to answer the following form of queries:*
Query: *Given an interval $I = [i_s..i_e], 1 \leq i_s \leq i_e \leq n$, the goal is to find the index k (or the value $A[k]$ itself) with maximum value (ties can be broken arbitrarily, e.g. by taking the one with larger (smaller) index) $A[k]$ for $k \in I$. The query is denoted by $RMQ_A(i_s, i_e)$*

Theorem 1. ([7, 5]) *Range Maxima Query Problem can be solved in $O(n)$ preprocessing time and $O(1)$ time per query.* □

Now, assume that we are computing the match (i,j). LCS-I maintains an array H of length N, where, for the current value of $i \in [1..N]$ we have, $H[\ell] = \max_{1 \leq k < i, (k,\ell) \in \mathcal{M}}(\mathcal{T}[k,\ell]), 1 \leq \ell \leq N$. The 'max' operation, here, returns 0, if there does not exist any $(k,\ell) \in \mathcal{M}$ within the range. Now, given the updated array H, LCS-I computes $\mathcal{T}[i,j]$ by using the constant time range maxima query as follows: $\mathcal{T}[i,j] = RMQ_H(1, j-1) + 1$. Because of Fact 1, LCS-I is able to maintain the array H on the fly using another array S, of length N, as a temporary storage. After calculating $\mathcal{T}[i,j]$, such that $(i,j) \in \mathcal{M}_i$, LCS-I stores $S[j] = \mathcal{T}[i,j]$. It continues to update S (and not H) as long as the computation continues in the same row. As soon as the processing of a new row begins, it updates H with new values from S. Due to Fact 1, LCS-I does not need to reset S after H is updated (by it) for the next row. Now, for the constant time range maxima query, an $O(N)$ time preprocessing is required as soon as H is updated. But due to Fact 2, it is sufficient to perform this preprocessing once per row. So, the computational effort added for this preprocessing is $O(N^2)$ in total. Therefore, LCS-I runs in $O(N^2)$ time.

2.2 LCS_RLE-I

In this section we present our first algorithm, namely LCS_RLE-I, to solve the LCS problem when one of the strings is an RLE string. Recall that, the notion of a match $(i,j) \in \mathcal{M}$, is extended when one input is an RLE string as follows: if $\widetilde{Y}[i] = a^q$ and $X[j] = a$ then we say $(i,j) \in \mathcal{M}$ and $run((i,j)) = q$. Following the idea of [10, 11], in the LCS_RLE-I algorithm, we maintain the arrays H and S and use them exactly the same way as they are used in LCS-I. We will be using another array $\mathcal{K}$ for the efficient implementation of our algorithm and its use will be clear as we proceed.

Now consider that we have completed the computation for the matches belonging to Row $i-1$ (i.e., $\mathcal{M}_{i-1}$). Now we start Row i. Given the updated array H, assume that we are processing the match (i,j). Also assume that when the the computation of the match (i,j) would be complete, i.e. $\mathcal{T}[i,j]$ is completely computed, we would have the result of $LCS\_RLE(\widetilde{Y}'a^q, X')$, where $\widetilde{Y}'a^q$ and X' are prefixes of $\widetilde{Y}$ and X respectively. Then, clearly, the match is due to the letter a. Now, if $q = 1$, then to compute $\mathcal{T}[i,j]$, we simply need to perform: $\mathcal{T}[i,j] = RMQ_H(1, j-1) + 1$. We also need to update S array to store the new value of $\mathcal{T}[i,j]$ as the current highest value of the jth column, i.e., we perform $S[j] = \mathcal{T}[i,j]$. In what follows, we refer to the above operations as the $baseOperation$.

If $q > 1$, then, we require two steps. Firstly, we perform the $baseOperation$. Then, in the second step (referred to as the $weightOperation$), we consider q previous matches (if fewer matches are available we need to consider all of them) in Row i, including the current one. Now, note carefully that T-values for these matches have already been computed and reflected in the S array. We copy S to $\mathcal{K}$ and S array is never changed by any $weightOperation$. For Row i, we call $\mathcal{K}[k]$ to be a $match\ position$ if $(i,k) \in \mathcal{M}_i$ and $\mathcal{K}[k]$ and k are referred to as the corresponding $\mathcal{K}$-value and

$\mathcal{K}$-index (similar notations are also defined for the arrays H and S). Now, we add a $weight$ to each of the corresponding $\mathcal{K}$-values: the $weight$ is 0 for the current match, 1 for the previous match, 2 for the match before it and so on. Now observe that $\mathcal{T}[i,j]$ will be the maximum of these values. This is because the kth element of this "window" from right, corresponds to matching k a's from the run a^q with rightmost k a's from X' and then matching the remaining substring with $\widetilde{Y}'$.

We will use an array $\mathcal{K}$ to do this computation efficiently. Now recall that we are handling the match $(i,j) \in \mathcal{M}_i$. We can implement the $weightOperation$ by adding the appropriate weights at the corresponding $match\ positions$ of $\mathcal{K}$ and then performing the query $RMQ_{\mathcal{K}}(u,j)$, such that $\mathcal{K}[u]$ is a $match\ position$ (due to (i,u)) and $q = number((i,j)) - number((i,u)) + 1$. In what follows we will refer to the above range (i.e., the range $[u..j]$) as the $weighted\ query\ window$. However, in this strategy, we may need to adjust the weights every time we compute a new match since for each match the $weighted\ query\ window$ may change. However, this would be costly. In what follows, we discuss how to do this more efficiently.

Rather than adding the appropriate $weight$, for a particular row, we will add $relative\ weight$ to all the $match\ positions$ of $\mathcal{K}$. This would ensure that the position of the maximum value remains the same, although the value may not. To get the correct value, we will finally deduct the appropriate difference from the value. We do it as follows. After the $baseOperation$, we copy the array S to array $\mathcal{K}$. Then, to a match $(i,\ell), 1 \leq \ell \leq |\mathcal{M}_i|$ we add $|\mathcal{M}_i| - number((i,j)) + 1$ as the $relative\ weight$. In other words we give weight 1 to the rightmost match, 2 to the next one and so on and finally, $|\mathcal{M}_i|$ to the first match.

Now recall that we are considering the match $(i,j) \in \mathcal{M}_i$, i.e., we are computing $\mathcal{T}[i,j]$. Assume that $number((i,j)) = |\mathcal{M}_i| - k + 1$, i.e., this is the kth $match\ position$ from right. As before, we execute the query $RMQ_{\mathcal{K}}(q,j)$, such that $\mathcal{K}[u]$ is a $match\ position$ (due to (i,u)) and $q = number((i,j)) - number((i,u)) + 1$. However, this time we need to do some adjustment as follows. It is easy to realize that each of the values of the matched positions in $\mathcal{K}[u..j]$, is k higher than the actual value. So, to correct the computation we perform $T[i,j] = RMQ_{\mathcal{K}}(q,j) - k$.

The analysis of the algorithm is similar to that of LCS-I algorithm of [10, 11]. As we need to do at most two RMQ preprocessing per row, overall it will cost $O(Ng)$ time (using $O(N)$ time preprocessing algorithm). We need two RMQ queries per match which amounts to $O(\mathcal{R})$ (using constant time RMQ query) time. Note that, in the worst case $\mathcal{R} = O(Ng)$. Finally, it is easy to see that, the set $\mathcal{M}$ in the prescribed order can be computed easily in $O(Ng)$ time. Therefore, we get the following theorem.

Theorem 2. *LCS_RLE-I solves Problem LCS_RLE in $O(Ng)$ time.* □

2.3 An Illustrative Example

In this section, we will give a partial example on how the LCS_RLE-I algorithm works for $X = ABBCCCAAAA$ and $\widetilde{Y} = C^3A^3$. We assume that we have already completed processing the matches belonging to Row 1 (i.e. $\mathcal{M}_1$). After calculating Row 1, values of S array are shown in Table 1 and the $\mathcal{T}$-values of Row 1 are shown in Table 2

value	0	0	0	0	1	2	3	0	0	0	0
index	0	1	2	3	4	5	6	7	8	9	10

Table 1: S array after calculating Row 1

C^3	0	0	0	0	1	2	3	0	0	0	0
		A	B	B	C	C	C	A	A	A	A
index	0	1	2	3	4	5	6	7	8	9	10

Table 2: $\mathcal{T}$-values of Row 1 after processing all the matches of Row 1

Now we will calculate Row 2. We have, $\mathcal{M}_2 = \{(2,1),(2,7),(2,8),(2,9),(2,10)\}$. Here, we have to perform both $baseOperation$ and $weightOperation$. The values of S array After $baseOperations$ for all the matches of $\mathcal{M}_2$ are shown in Table 3

value	0	1	0	0	1	2	3	4	4	4	4
index	0	1	2	3	4	5	6	7	8	9	10

Table 3: S array after $baseOperations$ of all the matches of Row 2

To calculate $weightOperation$, we will copy the S array into the $\mathcal{K}$ array and add relative weight as shown in Table 4 and Table 5.

value	0	1	0	0	1	2	3	4	4	4	4
weight	0	5	0	0	0	0	0	4	3	2	1
index	0	1	2	3	4	5	6	7	8	9	10

Table 4: $\mathcal{K}$-Array before addition of relative weight

value	0	6	0	0	1	2	3	8	7	6	5
index	0	1	2	3	4	5	6	7	8	9	10

Table 5: $\mathcal{K}$-Array after addition of relative weight

After calculating the matches of Row 2, values of S Array and $\mathcal{T}$ Array are shown in Table 6 and Table 7 respectively.

value	0	1	0	0	1	2	3	4	5	6	6
index	0	1	2	3	4	5	6	7	8	9	10

Table 6: S array after calculating Row 2

A^3	0	1	0	0	0	0	0	4	5	6	6
C^3	0	0	0	0	1	2	3	0	0	0	0
		A	B	B	C	C	C	A	A	A	A
index	0	1	2	3	4	5	6	7	8	9	10

Table 7: $\mathcal{T}$-values of Row 2 after processing all the matches of Row 2

3 LCS_RLE-II

In this section, we use the ideas of LCS_RLE-I to present our second algorithm, LCS_RLE-II, which runs in $O(\mathcal{R}\log(\log(g)) + N)$ time. To achieve this running time, we will use an elegant data structure (referred to as the vEB tree henceforth) invented by van Emde Boas [21] that allows us to maintain a sorted list of integers in the range $[1..n]$ in $O(\log(\log(n)))$ time per insertion and deletion. In addition to that it can return $next(i)$ (successor element of i in the list) and $prev(i)$ (predecessor element of i in the list) in constant time.

We follow the same terminology and assume the same settings of Section 2 to describe LCS_RLE-II. So, assume that we are considering the match $(i, j) \in \mathcal{M}_i$ and recall that when the the computation of the match (i, j) would be complete, i.e. $\mathcal{T}[i, j]$ is completely computed, we would have the result of $LCS\_RLE(\widetilde{Y}'a^p, X')$. Note carefully that the $baseOperation$ is basically the operation required to compute a normal LCS. We can use the LCS algorithm of [10] or [11] just to do the $baseOperation$ for each match. Then, for each match we would only need $O(\log(\log(g)))$ time [11, 10], requiring a total of $O(\mathcal{R}\log(\log(g)))$ time to perform all the $baseOperations$.

Now, we focus on the $weightOperations$. Our goal is to completely avoid any RMQ preprocessing. We need to modify the $weightOperation$ as follows. We will use the vEB tree for this purpose. Recall that we want to find the maximum value of $\mathcal{K}$ in the $weighted\ query\ window$. Furthermore, note that, only the matched positions of $\mathcal{K}$ in the $weighted\ query\ window$ are important in the calculation. So instead of maintaining the array $\mathcal{K}$, we maintain a vEB tree where always the appropriate number (q in this case) of matches (with values after the addition of the relative weights) are kept. And as the computation moves from one match to the next, to maintain the appropriate $weighted\ query\ window$, only one element (corresponding to a match) is added to the vEB tree and at most one element is deleted. (We do not need to delete any element if the current $weighted\ query\ window$ has fewer matches than q.)

When we need the maximum value of the $weighted\ query\ window$, we just find the maximum from the vEB tree which can also be found in $O(\log(\log(g)))$ time (by inserting a fictitious element having infinite value and then deleting it after computing its predecessor). As we need to insert and delete constant number of elements from the vEB tree for each match, this can be done in $O(\mathcal{R}\log(\log(g)))$ time on the whole. Like before, we would need to deduct the appropriate value ($(|\mathcal{M}_i| + 1 - number((i, j))$ in this case) from the returned maximum to do the proper adjustment. Algorithm 1 presents the idea (only the computation of the values) more formally. Notably, we must maintain appropriate pointers to recover the actual LCS after the computation is done.

Finally, the computation of the set $\mathcal{M}$ in the prescribed order can be done following the preprocessing algorithm of [10, 11] which runs in $O(\mathcal{R}\log(\log(g)) + N)$ time. So, we have the following theorem.

Theorem 3. *LCS_RLE-II solves Problem LCS_RLE in* $O(\mathcal{R}\log(\log(g)) + N)$ *time.*

4 Conclusion

In this paper, we have studied the longest common subsequence problem for two strings, where one of the input strings is run length encoded. We have presented two novel algorithms, namely LCS_RLE-I and LCS_RLE-II to solve the problem. We have first presented LCS_RLE-I combining some new ideas with the techniques used in [11, 10]. LCS_RLE-I runs in $O(gN)$ time, which matches the best algorithm in the literature. Then we present an input sensitive algorithm, namely, LCS_RLE-II that runs in $O(\mathcal{R}\log(\log(g)) + N)$ time. Observe that in the worst case, $\mathcal{R} = O(gN)$ and hence the worst case running time of LCS_RLE-II is slightly

Algorithm 1 LCS_RLE-II: Computation of Row i (Only the computation of value is shown)

1: **for** each $(i, j) \in \mathcal{M}_i$ in the left to right order **do**
2: Perform $baseOperation$ and update S array accordingly
3: **end for**
{Now we perform the $weightOperation$ for every match of $\mathcal{M}_i$. Assume that each vEB element is a tuple $(value, pos)$}
4: vEBTree = null
5: **for** each $(i, j) \in \mathcal{M}_i$ in the left to right order **do**
6: $relativeWeight = |\mathcal{M}_i| - number((i, j)) + 1$
7: vebTree.Insert((S[j]+relativeWeight,j))
8: **if** $|vebTree| > q$ **then**
9: Delete the earliest inserted element from vebTree {The earliest inserted element can be efficiently found by maintaining a normal linked list between the inserted elements}
10: **end if**
11: $S[j] = vebTree.Maximum() - (|\mathcal{M}_i| + 1 - number((i, j)))$
12: $\mathcal{T}[i, j] = S[j]$
13: **end for**

worse than the best algorithm in the literature. However, in many cases $\mathcal{R} = o(gN)$, and our algorithm would show superior behaviour in these cases. In particular, if $\mathcal{R} < gN/\log(\log(g))$, LCS_RLE-II will outperform the best algorithms in the literature. Additionally, if we run Mitchell's algorithm (the best algorithm in the literature for two RLE strings) in our setting, the running time would be $O((\mathcal{R}+G+n)\log(\mathcal{R}+G+n))$, which clearly is worse than ours. Also, employing some of the insights of Mitchel [16], we believe, our work can be extended to the version where both the input are RLE strings.

Finally, all the works in the literature so far on LCS computation considering RLE strings focused only on theoretical complexity results of the devised algorithms. Theoretical improvement in these algorithms were achieved in most cases by using complex data structures (e.g., in our case, vEB tree and RMQ data structures). In practice, such algorithms, despite having better theoretical bounds, may turn out to be worse in performance. Hence, an interesting research direction could be to implement the algorithms in the literature along the new ones proposed here and to compare them against each other from a practical point of view. Notably, we have already started working in this direction and hope to present the findings in a forthcoming paper.

Acknowledments

The authors would like to thank the annonymous reviewers and the editor for constructive comments and suggestions which improved the presentation of the paper a lot. This research work constitues part of the B.Sc. Engineering thesis of Ahsan and Shahriyar under the supervision of Rahman. Moosa is currently working at Google Inc., USA.

References

[1] Altschul, S. F., Gish, W., Miller, W., Meyers, E. W., and Lipman, D. J. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[2] Ann, H.-Y., Yang, C.-B., Tseng, C.-T., and Hor, C.-Y. A fast and simple algorithm for computing the longest common subsequence of run-length encoded strings. *Inf. Process. Lett.*, 108(6):360–364, 2008.

[3] Apostolico, A., Landau, G. M., and Skiena, S. Matching for run-length encoded strings. *J. Complexity*, 15(1):4–16, 1999.

[4] Arlazarov, V., Dinic, E., Kronrod, M., and Faradzev, I. On economic construction of the transitive closure of a directed graph (english translation). *Soviet Math. Dokl.*, 11:1209–1210, 1975.

[5] Bender, M. A. and Farach-Colton, M. The lca problem revisited. In *LATIN*, pages 88–94, 2000.

[6] Freschi, V. and Bogliolo, A. Longest common subsequence between run-length-encoded strings: a new algorithm with improved parallelism. *Inf. Process. Lett.*, 90(4):167–173, 2004.

[7] Gabow, H., Bentley, J., and Tarjan, R. Scaling and related techniques for geometry problems. In *STOC*, pages 135–143, 1984.

[8] Hunt, J. W. and Szymanski, T. G. A fast algorithm for computing longest subsequences. *Commun. ACM*, 20(5):350–353, 1977.

[9] Iliopoulos, C. S. and Rahman, M. S. Algorithms for computing variants of the longest common subsequence problem. *Theor. Comput. Sci.*, 395(2-3):255–267, 2008.

[10] Iliopoulos, C. S. and Rahman, M. S. New efficient algorithms for the lcs and constrained lcs problems. *Inf. Process. Lett.*, 106(1):13–18, 2008.

[11] Iliopoulos, C. S. and Rahman, M. S. A new efficient algorithm for computing the longest common subsequence. *Theory Comput. Syst.*, 45(2):355–371, 2009.

[12] K. Sayoood, E. F. E. *Introduction to Data Compression*. Morgan Kaufmann Publishers Inc, 2000.

[13] Levenshtein, V. Binary codes capable of correcting deletions, insertions, and reversals. *Problems in Information Transmission*, 1:8–17, 1965.

[14] Liu, J. J., Wang, Y.-L., and Lee, R. C. T. Finding a longest common subsequence between a run-length-encoded string and an uncompressed string. *J. Complexity*, 24(2):173–184, 2008.

[15] Masek, W. J. and Paterson, M. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–31, 1980.

[16] Mitchell, J. A geometric shortest path problem, with application to computing a longest common subsequence in run-length encoded strings. *Technical Report Department of Applied Mathematics, SUNY Stony Brook*, 1997.

[17] Myers, E. W. An o(nd) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[18] Nakatsu, N., Kambayashi, Y., and Yajima, S. A longest common subsequence algorithm suitable for similar text strings. *Acta Inf.*, 18:171–179, 1982.

[19] Pearson, W. and Lipman, D. Improved tools for biological sequence comparison. *Proceedings of National Academy of Science, USA*, 85:2444–2448, 1988.

[20] Rahman, M. S. and Iliopoulos, C. S. Algorithms for computing variants of the longest common subsequence problem. In *ISAAC*, pages 399–408, 2006.

[21] van Emde Boas, P. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6:80–82, 1977.

[22] Wagner, R. A. and Fischer, M. J. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.