Longest Common Subsequence Problem for Run-Length-Encoded Strings

Shegufta Bakht Ahsan[a], Syeda Persia Aziz[a], M. Sohel Rahman[a]
[a] AℓEDA Group
Department of CSE, BUET, Dhaka-1000, Bangladesh
Email: {shegufta.b.ahsan, persia.aziz}@gmail.com, msrahman@cse.buet.ac.bd

***Abstract*—In this paper, we present a new and efficient algorithm for solving the Longest Common Subsequence (*LCS*) problem between two run-length-encoded (*RLE*) strings. Suppose $\hat{Y}$ and $\hat{X}$ are two *RLE* strings having length $\hat{k}$ and $\hat{\ell}$ respectively. Also assume that Y and X are the two uncompressed versions of the two *RLE* strings $\hat{Y}$ and $\hat{X}$ having length k and ℓ respectively. Then, our algorithm runs in $O((\hat{k}+\hat{\ell})+\mathcal{R}\log\log(\hat{k}\hat{\ell})+\mathcal{R}\log\log\omega)$ time, where $\omega = k+\ell$ and $\mathcal{R}$ is the total number of ordered pairs of positions at which the two *RLE* strings match. Our algorithm outperforms the best algorithms for the same problem in the literature.**

***Index Terms*—RLE, LCS, vEB Tree, Bounded Heap, Matched Block Calculation.**

I. Introduction

SUPPOSE two strings $X[1\ldots n] = X[1]X[2]\ldots X[n]$ and $Y[1\ldots n] = Y[1]Y[2]\ldots Y[n]$ are given. A subsequence $S[1\ldots r] = S[1]S[2]\ldots S[r]$, $0 < r \leq n$ of X is obtained by deleting $n-r$ symbols from X. A common subsequence of two strings X and Y, denoted *lcs(X,Y)* is a subsequence common to both X and Y. The longest common subsequence (LCS) problem for two strings is to find a common subsequence in both the strings, having the maximum possible length.

There is a string compression technique which is called run-length-encoding [1]. 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*. For example, the *RLE string* of the string $bdcccaaaaaa$ is $b^1d^1c^3a^6$. Given two *RLE* strings, the longest common subsequence problem for two *RLE* strings (*Problem LCS_2RLE*) is to find a common subsequence in both the strings, having the maximum possible length. In this paper, we present an efficient algorithm to solve *Problem LCS_2RLE*.

A. Literature Review

The problem of computing *LCS* when one or both of the strings are run-length-encoded has attracted significant attention in the literature. Freschi and Bogliolo [2] proposed an $O(k\hat{\ell}+\ell\hat{k}-\hat{\ell}\hat{k})$ time algorithm for finding the longest common subsequence between two *RLE* strings, where ℓ and k are the lengths of the original strings X and Y, respectively, and $\hat{\ell}$ and $\hat{k}$ are the numbers of *runs* in the *RLE* representations of X and Y, respectively.

An interesting and perhaps more relevant parameter for the *LCS* problem is $\mathcal{R}$, which is the total number of ordered pairs of positions where the two strings match. Efficient algorithms based on parameter $\mathcal{R}$ to solve the *LCS* problem for uncompressed strings have been proposed in the literature [3], [4]. Working with a similar goal, Mitchell proposed an $O((\mathcal{R}+\hat{k}+\hat{\ell})\log(\mathcal{R}+\hat{k}+\hat{\ell}))$ algorithm [5] capable of computing an *LCS* when both inputs are *RLE* strings. Ann et al. [6] also proposed an algorithm for solving the same problem in $O(\hat{k}\hat{\ell}+\min(p_1,p_2))$ time where p_1, p_2 denote the number of elements in the bottom and right boundaries of the *matched blocks* respectively. Apostolico et al. [7] gave another algorithm for this problem which runs in $O(\hat{k}\hat{\ell}\log(\hat{k}\hat{\ell}))$ time. To the best of our knowledge the latest work on this problem are from Sakai [8] which proposes a solution of $O(\hat{k}\hat{\ell}\log\log(\min(\hat{k},\hat{\ell},k/\hat{k},l/\hat{\ell},X)))$ time where X is the average difference between the length of a run from one input string and that of a run from the other. Notably, this is a conference paper which does not provide many proofs and details due to space constraints and we are still waiting for the journal version of the paper to be published to get all the details.

The variants of the LCS problem have also been investigated in literature. Liu et al. [9] proposed an $O(\min(k\hat{\ell},\ell\hat{k}))$ time algorithm and Ahsan et al. [10] proposed an $O(\ell+\mathcal{R}\log\log\hat{k})$ time algorithm for finding a longest common subsequence when one of the strings is *RLE* and the other is an uncompressed string. Farhana et al. [11] presented finite automata based algorithms for two recently studied variants of the classic LCS problem, namely, the Doubly-Constrained LCS (DC-LCS) and Hybrid-Constrained LCS (HC-LCS). Alam et al. [12] proposed an $O(k\ell)$ algorithm to solve the Substring Inclusion Constrained LCS problem.

B. Our Contribution

In this paper, we are interested in computing an *LCS* for two Run-Length-Encoded strings. The application of *LCS* mostly comes from Computational Molecular Biology [13], [14], where we want to compare DNA or

doi:10.4304/jcp.9.8.1769-1775

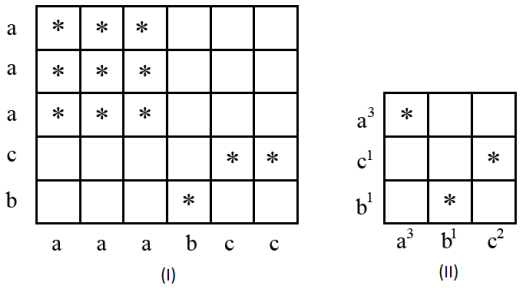

Fig. 1. Blocks containing '*' are the elements of set $\mathcal{M}$ for (I) two uncompressed strings, where $|\mathcal{M}| = \mathcal{R} = 12$, and (II) two run-length-encoded strings, where $|\mathcal{M}| = \mathcal{R} = 3$.

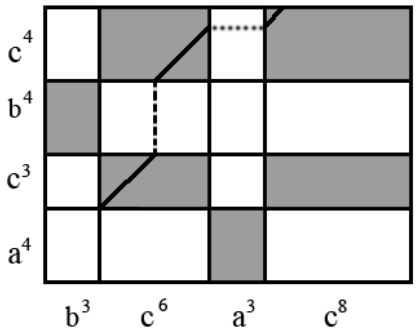

Fig. 2. A path, consisting of character 'c' and starting from the lower-left corner of $\mathcal{T}[2,2]$.

protein sequences to learn how homologous they are. The motivation of using *RLE* strings is to reduce cost in such computations because the DNA and protein sequences are mostly large and have consecutive identical characters.

We combine the idea and technique of Apostolico et al. in [7] and Iliopoulos and Rahman in [3], [15] to devise an efficient algorithm for *Problem LCS_2RLE*. Our main result is an $O((\hat{k}+\hat{\ell})+\mathcal{R}\log\log(\hat{k}\hat{\ell})+\mathcal{R}\log\log\omega)$ time algorithm, where $\omega = k + \ell$.

C. Road Map

The rest of the paper is organized as follows. In Section II, we discuss about the notation and definitions that we have used throughout this paper. In Section III, we describe our algorithm. In Section IV, we discuss about the complexity of the algorithm. Finally, we briefly conclude in Section V.

II. Preliminaries

Throughout this paper, we use the following conventions. We use $\hat{X} = \hat{X}_1\hat{X}_2\ldots\hat{X}_{\hat{\ell}}$ to denote the run-length-encoding of a string X, where $\hat{X}_j, 1 \le j \le \hat{\ell}$ is a maximal run of identical characters and $|\hat{X}_j|$ denotes the length of this run. We say that the length of the *RLE* string $\hat{X}$ is $|\hat{X}| = \hat{\ell}$. The length of string X, denoted ℓ, represents the total number of characters in X. So $\ell = |X| = \sum_{j=1}^{\hat{\ell}} |\hat{X}_j|$. We use x_j to denote the unique character comprising the run $\hat{X}_j$.

We maintain a $\hat{k}*\hat{\ell}$ matrix $\mathcal{T}$ (corresponding to the runs of characters in $\hat{Y}$ and $\hat{X}$), such that $\mathcal{T}[i,j]$ contains the value of an optimal solution between the prefixes $\hat{X}_{[1\cdots j]}$ and $\hat{Y}_{[1\cdots i]}$.

Note that, the notation $\mathcal{R}$ and $\mathcal{M}$ may be used in the context of both compressed and uncompressed strings (Fig. 1). In both case, the concept remains the same. So, $\mathcal{R}$ is the total number of ordered pairs of positions at which the characters (character blocks) of two uncompressed (compressed) strings match. More formally, for compressed strings, we say a pair $(i,j), 1 \le i \le \hat{k}, 1 \le j \le \hat{\ell}$ defines a match if $x_j = y_i$. Additionally, if $x_j = y_i = \alpha$, then we say that the match (i,j) contains or is due to the character α. The set of all matches, $\mathcal{M}$ is defined as follows:

$$\mathcal{M} = \{(i,j) \mid x_j = y_i, 1 \le i \le \hat{k}, 1 \le j \le \hat{\ell}\}$$

In Fig. 1, the blocks containing '*' is a matched block. Observe that $|\mathcal{M}| = \mathcal{R}$.

We use $COUNT^j_{\hat{X}}(\alpha)$ to denote the number of occurrences of the character α in the uncompressed version of $\hat{X}_1\hat{X}_2\cdots\hat{X}_j$, where $1 \le j \le \hat{\ell}$. For example, in string $\hat{X} = b^3c^2b^7a^5c^6b^1a^6$, $COUNT^7_{\hat{X}}(b) = 11$, because in the uncompressed version of $\hat{X}_1\hat{X}_2\cdots\hat{X}_7$, there are total $3+7+1 = 11$ occurrences of b. The tables $COUNT^j_{\hat{X}}(\alpha)$, $1 \le j \le \hat{\ell}$ and $COUNT^i_{\hat{Y}}(\alpha)$, $1 \le i \le \hat{k}$, can be computed easily in $O(\hat{k}+\hat{\ell})$ time.

Fig. 2 illustrates the matrix of blocks for the input strings $\hat{X} = b^3c^8a^3c^8$ and $\hat{Y} = a^4c^3b^4c^4$. We say that a block (i,j) is dark if the corresponding characters match, i.e. $y_i = x_j$; block (i,j) is light if $y_i \ne x_j$. Sometimes dark blocks are referred to as *matched blocks*.

If $\mathcal{T}[i,j]$ is a *matched block* consists of character α, then a path starts from the *lower-left* corner of $\mathcal{T}[i,j]$. While it traverses through block $\mathcal{T}[i,j]$, it goes at $45°$ with the bottom boundary. The path ends if it hits the *upper-right* corner of the block. Else if the path hits the *top* (*right*) boundary of the block, it goes vertically (horizontally) until it hits the next *matched block* (if any) at $\mathcal{T}[i,j']$ where $j+1 < j' \le \hat{\ell}$ ($\mathcal{T}[i',j]$ where $i+1 < i' \le \hat{k}$). After entering the *matched block* $\mathcal{T}[i,j']$ ($\mathcal{T}[i',j]$), again it traverses that block at $45°$ with the bottom boundary and when the *upper-right* or *top* or *right* corner is reached, it repeats the same procedure described above. This procedure continues until either the path hits the *upper-right* corner of a *matched block*, or the path exceeds the boundary of $\mathcal{T}$.

Fig. 2 shows an example of a path starting from the *lower-left* corner of $\mathcal{T}[2,2]$ which is a *matched block* consists of character 'c'. It traverse at $45°$ inside $\mathcal{T}[2,2]$ (solid path). While it reaches the *top* boundary of $\mathcal{T}[2,2]$, it starts traversing vertically (dashed path) until it hits the next *matched block* at $\mathcal{T}[4,2]$. Inside this *matched block*, it again traverse at $45°$ (solid path) and while it reaches the *right* boundary of $\mathcal{T}[4,2]$, it starts traversing horizontally (dotted path) until it hits the next *matched block* at $\mathcal{T}[4,4]$. It traverse at $45°$ inside $\mathcal{T}[4,4]$ (solid path) and reaches at the *top* boundary. But as this is the end of the boundary

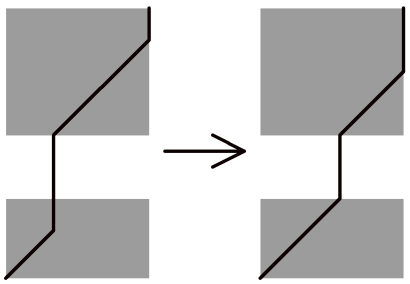

Fig. 3. Converting an arbitrary subpath into a forced subpath.

of $\mathcal{T}$, this path which was started from $\mathcal{T}[2,2]$ ends its traversal.

Apostolico et al. [7] first introduced the concept of a *corner path* which is a path entering a *matched block* only at the lower-left corner and exiting only through the upper-right corner. On the other hand, a path beginning at the lower-left corner of a *matched block* is a *forced path* if it traverses *matched blocks* by strictly diagonal moves and, whenever the right (respectively upper) side of an intermediate *matched block* is reached, it proceeds ,to the next *matched block* by a straight horizontal (respectively vertical) "leap" through the light blocks in between. Intuitively, whenever we enter a *matched block* in a lower-left corner, it will be the start of a *forced path*. The *forced path* stops when it hits either the right or the upper side and the path follows the side to the upper-right corner. Any arbitrary path can be converted to a *forced path*. The process is illustrated in Fig. 3.

We define $RANK(\alpha,i,j) = COUNT_{\hat{Y}}^{i-1}(\alpha) - COUNT_{\hat{X}}^{j-1}(\alpha)$ to be an attribute of a path starting from $\mathcal{T}[i,j]$ and containing character α. Maximum possible value of $RANK$ for any $\alpha \in \Sigma$ can be k as follows: $RANK_{max}(\alpha,\hat{k},1)$ $= COUNT_{\hat{Y}}^{\hat{k}-1}(\alpha) - COUNT_{\hat{X}}^{1-1}(\alpha) = COUNT_{\hat{Y}}^{\hat{k}-1}(\alpha) - 0 \leq k$

Similarly, minimum possible value of $RANK$ for any $\alpha \in \Sigma$ can be $-\ell$ as follows: $RANK_{min}(\alpha,1,\hat{\ell})$ $= COUNT_{\hat{Y}}^{1-1}(\alpha) - COUNT_{\hat{X}}^{\hat{\ell}-1}(\alpha) = 0 - COUNT_{\hat{X}}^{\hat{\ell}-1}(\alpha) \geq -\ell$

Clearly, all possible values of $RANK$ can be represented by the numbers within the range $[-\ell,k]$ that contains $\omega = k+\ell$ values.

III. Our Algorithm

In this Section, we present a new algorithm by extending the concept of a path based solution for *LCS* between two *RLE* strings by Apostolico et al. [7]. To improve performance, we will use the techniques used by Iliopoulos and Rahman [3], [15].

In this algorithm, our plan is to compute only those $\mathcal{T}[i,j]$ for which $(i,j) \in \mathcal{M}$. The following information is kept for each forced path that starts from the block $\mathcal{T}[i,j]$: (1) The starting location of the path, namely (i,j); (2) the character of the match, i.e. α; (3) its *initial value* v_{in}; and (4) its *RANK*. For a path, started at $\mathcal{T}[i,j]$, its *initial value* v_{in} is $\max(\mathcal{T}[i',j'])$, $1 \leq i' < i, 1 \leq j' < j$. If $i = 1$ or $j = 1$, then *initial value* $v_{in} = 0$. Naive calculation of v_{in} may require the calculation of some $\mathcal{T}[i,j]$, for which $(i,j) \notin \mathcal{M}$. But later we will present a novel algorithm to calculate the *initial value* v_{in} by considering only the *matched blocks*.

TABLE I
Calculation of the $RANK$ of paths (Fig. 4) consist of character 'a'

(i,j)	$COUNT_{\hat{Y}}^{i-1}(a)$	$COUNT_{\hat{X}}^{j-1}(a)$	$RANK(a,i,j)$
$(1,1)$	0	0	0
$(1,3)$	0	7	-7
$(1,5)$	0	17	-17
$(3,1)$	8	0	8
$(3,3)$	8	7	1
$(3,5)$	8	17	-9
$(5,1)$	16	0	16
$(5,3)$	16	7	9
$(5,5)$	16	17	-1

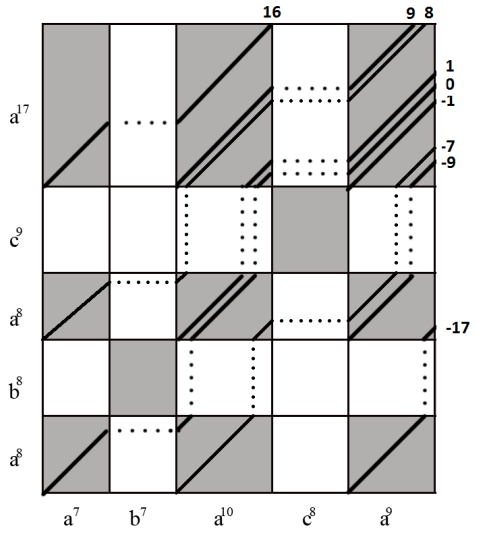

Fig. 4. Forced paths consist of same character never cross each other and obey a top-down order which is maintained by *RANK*.

It is clear from [7] that all characters which are matched on any given forced path will be identical. Also two *forced paths* which proceed on matches of different instances of the same character will never cross each other (Fig. 4). Hence, the *forced paths* consist of same character obey a top-down order which is maintained by $RANK$. So, the path, starting from $\mathcal{T}[i,j]$, and containing character α intersects any column $j' \geq j$ according to the value of $RANK(\alpha,i,j)$. For example, in Fig. 4 the *forced paths* consist of character 'a' and passing column 5 maintain an order according to their $RANK$. Similarly *forced paths* consist of character 'a' and passing row 5 maintain an order according to their $RANK$. The calculation of the $RANK$ is explained in Table I.

Now consider a forced path of α, which starts at $\mathcal{T}[i,j]$ with an *initial value* v_{in}. When this path crosses column $j' > j$, its *value* $v = v_{in} + COUNT_{\hat{X}}^{j'}(\alpha) - COUNT_{\hat{X}}^{j-1}(\alpha)$. Similarly, when this path crosses row

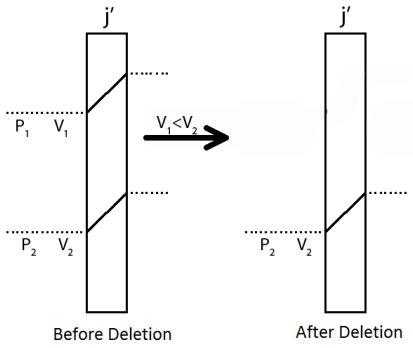

Fig. 5. Deletion of a lower valued path which has higher RANK

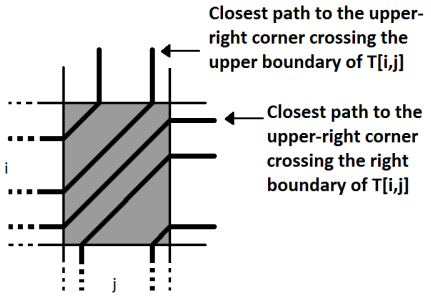

Fig. 6. Two closest paths to the upper-right corner of $\mathcal{T}[i,j]$, one crossing the upper boundary of $\mathcal{T}[i,j]$ and another crossing the right boundary of $\mathcal{T}[i,j]$ (partial view).

$i' > i$, its *value* will be $v = v_{in} + COUNT^{i'}_{\hat{Y}}(\alpha) - COUNT^{i-1}_{\hat{Y}}(\alpha)$.

So it is clear for a forced path of α which starts at (i,j) with an *initial value* v_{in} that if a column $j', j \leq j' \leq \hat{\ell}$ $(row\ i', i \leq i' \leq \hat{k})$ is given, then the *value* of the forced path that crosses this column (row) can be computed in $O(1)$ time, following $O(\hat{k}+\hat{\ell})$ time preprocessing.

$\mathcal{T}$ is an $\hat{k} * \hat{\ell}$ matrix such that $\mathcal{T}[i,j]$ contains the value of an optimal solution between prefixes $\hat{X}_{[1\cdots j]}$ and $\hat{Y}_{[1\cdots i]}$, $1 \leq j \leq \hat{\ell}$ and $1 \leq i \leq \hat{k}$. From [7], it is found that $\mathcal{T}[i,j]$ is the maximum of $\mathcal{T}[i-1,j], \mathcal{T}[i,j-1]$ and the *value* of the forced paths that cross block (i,j), including the one that starts on its *lower-left* corner. The set of forced paths can be divided into two groups. The *first group* contains all paths that cross column j below row i, while the *second group* contains all paths that cross row i on the left of column j. Our goal is to find the path with the highest score in each group, so that $\mathcal{T}[i,j]$ can be computed in $O(1)$ time.

Below, we discuss how to find the path with maximum *value* in the *first group*. Here we consider forced paths that match the character α. The *second group* and other characters can be handled similarly.

According to [7], it can be said that if we consider two forced paths P_1 and P_2 with values v'_1 and v'_2 respectively when they cross column j', and v''_1 and v''_2 respectively when they cross column j''. Then those values obey the equality: $v'_1 - v'_2 = v''_1 - v''_2$.

Therefore whenever a forced path P_1 intersects column j' higher than another forced path P_2, but the value of P_1 at j' is smaller than the value of P_2 at j', then path P_1 can be excluded from further consideration and hence can be deleted (Fig. 5). Our goal is to maintain, in order, only the paths which have higher values than the paths below them.

In order to be able to maintain the above properties we use an elegant data structure (referred to as the *vEB tree* henceforth) invented by van Emde Boas [16] that allows us to maintain a sorted list of integers in the range $[1 \ldots n]$ 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.

Here, we use $RANK$ as the key of the *vEB trees*. For each character α, two *vEB trees* are maintained, one $(vEB_{(\alpha,col)})$ maintaining the ordered list of paths matching the character α and crossing columns and the other $(vEB_{(\alpha,row)})$ maintaining the ordered list of paths matching the character α and crossing rows. These two trees will be used in dealing with all dark blocks that match α.

Here since the paths are sorted according to their $RANKs$ and $values$, it is sufficient to consider only two forced paths. These paths are the two closest paths to the upper-right corner of $\mathcal{T}[i,j]$, one that crosses the right side of $\mathcal{T}[i,j]$ and one that crosses the upper side of $\mathcal{T}[i,j]$ as shown in Fig. 6.

For each $(i,j) \in \mathcal{M}$ and containing letter α, a new record $\langle RANK, v_{in}\rangle$ for the path started from the lower left corner of $\mathcal{T}[i,j]$ is inserted in $vEB_{(\alpha,col)}$ and $vEB_{(\alpha,row)}$ according to its $RANK$. This new record is called $Record_{new}$. The current value v of the new path is calculated. The *value* v_{left} of the record $Record_{pre}$, which is the predecessor of the $Record_{new}$ of $vEB_{(\alpha,col)}$ is calculated. If $v < v_{left}$, then $Record_{new}$ is deleted, otherwise the *value* v_{right} of the record $Record_{suc}$ which is the successor of $Record_{new}$ is calculated. If $v_{right} < v$ then $Record_{suc}$ is deleted and this process continues until in $vEB_{(\alpha,col)}$, a successor of $Record_{new}$ with a greater *value* is reached. This process is described in Algorithm 2. Similar operations are applied on $vEB_{(\alpha,row)}$.

Now, $\mathcal{Z} = COUNT^{i}_{\hat{Y}}(\alpha) - COUNT^{j}_{\hat{X}}(\alpha)$. In the $vEB_{(\alpha,col)}$, the *value* C of the successor of $\mathcal{Z}$ (if any) is calculated, which is the highest score of the forced paths on column j, below row i. This procedure is described in Algorithm 3. Similarly the highest score (R) of the forced paths on row i, left to column j is calculated.

To compute the set $\mathcal{M}$ in the prescribed order (for a row by row operation) we can easily use the preprocessing of [17] which will run in $O((\hat{k}+\hat{\ell})+\mathcal{R}\log\log(\hat{k}\hat{\ell}))$ time. Now, we utilize the following facts observed in [17].

Fact 1: ([17]) 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 $\mathcal{T}[i,j] \leq \mathcal{T}[i',j]$ (resp. $\mathcal{T}[i,j] \leq \mathcal{T}[i,j']$). □

Fact 2: ([17]) The calculation of the entry $\mathcal{T}[i,j], (i,j) \in \mathcal{M}, 1 \leq i \leq \hat{k}, 1 \leq j \leq \hat{\ell}$ is independent of any $\mathcal{T}[p,q], (p,q) \in \mathcal{M}, p = i, 1 \leq q \leq \hat{k}$. □

Notably, although in [17], these facts were observed

for uncompressed strings, it is easy to realize that these hold true for compressed strings as well. We also use the *BoundedHeap* data structure of [18], that supports the following operations: $Insert(\mathcal{H}, Pos, Value, Data)$, $IncreaseValue(\mathcal{H}, Pos, Value, Data)$, $BoundedMax(\mathcal{H}, Pos)$.

The *BoundedHeap* data structure can support each of the above operations in $O(\log \log n)$ amortized time [18], where keys are drawn from the set $1, \ldots, n$. The data structure requires $O(n)$ space.

By using Fact 1, it can be said that for a path starting from $\mathcal{T}[i,j]$, $(i,j) \in \mathcal{M}$ if $i = 1$ or $j = 1$, then its *initial value* $v_{in} = 0$ else its *initial value* $v_{in} = \max_{1 \le e \le j-1}(\mathcal{T}[i-1, e])$. To compute $\mathcal{T}[i,j]$, we also have to find the value of $\mathcal{T}[i-1,j]$ and $\mathcal{T}[i,j-1]$. But it may happen that $(i-1,j) \notin \mathcal{M}$ and/or $(i,j-1) \notin \mathcal{M}$. Recall on the other hand that we only want to spend computational time on $(i,j) \in \mathcal{M}$. To handle this situation, we use an array $\mathcal{F}$ of size $\hat{\ell}$, which is initialized to zero. In each iteration, after calculating the value of $\mathcal{T}[i,j]$, $(i,j) \in \mathcal{M}$, $\mathcal{F}[j]$ is updated with the value of $\mathcal{T}[i,j]$. So, by using Fact 1, it also can be said that for any iteration $T[i,j]$, $(i,j) \in \mathcal{M}$, $\mathcal{T}[i-1,j] = \mathcal{F}[j]$ and if $j = 1$ then $\mathcal{T}[i,j-1] = 0$ else $\mathcal{T}[i,j-1] = \mathcal{F}[j-1]$. Thus by using $\mathcal{F}$, we can avoid calculating blocks $(i,j) \notin \mathcal{M}$.

Now we formally present our algorithm *LCS_2RLE_I* in Algorithm 1 which proceeds as follows. Recall that we perform a row by row operation. We always deal with two *BoundedHeap* data structures simultaneously. While considering row i, we already have the *BoundedHeap* data structure $\mathcal{H}_{i-1}$ at our hand; now we construct the *BoundedHeap* data structure $\mathcal{H}_i$. At first $\mathcal{H}_i$ is initialized to $\mathcal{H}_{i-1}$. Due to Fact 1, as soon as we compute the $\mathcal{T}$ value of a new match in a column j , we can forget about the previous matches of that column. So, as soon as we compute $\mathcal{T}[i,j]$ in row i, we insert it in $\mathcal{H}_i$ to update it for the next row, i.e. row $i+1$. And, due to Fact 2, we can use $\mathcal{H}_{i-1}$ for the computation of the $\mathcal{T}$ values of the matches in row i and do the update in $\mathcal{H}_i$ (initialized at first to $\mathcal{H}_{i-1}$) to make $\mathcal{H}_i$ ready for row $i+1$. For each match $(i,j) \in \mathcal{M}$ and $x_j = y_i = \alpha$, we insert a new record $\langle RANK, v_{in} \rangle$ in $vEB_{(\alpha,col)}$ and $vEB_{(\alpha,row)}$ for the path started from the lower left corner of $\mathcal{T}[i,j]$ as described in Algorithm 2. To find the v_{in}, we perform $BoundedMax(\mathcal{H}_{i-1}, j)$ and thus skip calculation for $(i,j) \notin \mathcal{M}$. Then we calculate the highest score (C) of the forced paths on column j, below row i as described in Algorithm 3 and the highest score (R) of the forced paths on row i, left to column j as described in Algorithm 4. Finally the value of $\mathcal{T}[i,j]$ is $\max(\mathcal{F}[j-1], \mathcal{F}[j], C, R)$. After that using the $IncreaseValue()$ operation, we update the value of $\mathcal{T}[i,j]$ in H_i. $\mathcal{F}[j]$ is also updated with the value of $\mathcal{T}[i,j]$.

IV. Algorithm Analysis

In this section we will discuss about the complexity of our proposed algorithm.

Algorithm 1 LCS_2RLE_I

1: Construct the set $\mathcal{M}$. Let $\mathcal{M}_i = \{(i,j) \in \mathcal{M},\ 1 \le j \le \hat{\ell}\},\ 1 \le i \le \hat{k}$
2: $\mathcal{H}_0 = \epsilon$
3: $\mathcal{F}[j] = 0,\ 1 \le j \le \hat{\ell}$
4: **for** $i = 1$ *to* $\hat{k}$ **do**
5: $\quad \mathcal{H}_i = \mathcal{H}_{i-1}$
6: $\quad$ **for** $each(i,j) \in \mathcal{M}_i$ **do**
7: $\quad\quad \alpha = x_j$
8: $\quad\quad$ *Step I* : Insert a new forced path in $vEB_{(\alpha,col)}$ and $vEB_{(\alpha,row)}$ according to its $RANK$, and keep the paths sorted according to their value.
9: $\quad\quad$ *Step II* : Find the highest score (C) of the forced paths on column j, below row i.
10: $\quad\quad$ *Step III* : Find the highest score (R) of the forced paths on row i, left to column j.
11: $\quad\quad$ **if** $j = 1$ **then**
12: $\quad\quad\quad temp = 0$
13: $\quad\quad$ **else**
14: $\quad\quad\quad temp = \mathcal{F}[j-1]$
15: $\quad\quad$ **end if**
16: $\quad\quad$ *Step IV* : $\mathcal{T}[i,j] = \max(temp, \mathcal{F}[j], C, R)$
17: $\quad\quad IncreaseValue(\mathcal{H}_i, j, \mathcal{T}[i,j], (i,j))$
18: $\quad\quad \mathcal{F}[j] = \mathcal{T}[i,j]$
19: $\quad$ **end for**
20: $\quad$ Delete $\mathcal{H}_{i-1}$
21: **end for**

Algorithm 2 *Step I* Inserting a new path

1: Compute $RANK(\alpha,i,j) = COUNT_{\hat{Y}}^{i-1}(\alpha) - COUNT_{\hat{X}}^{j-1}(\alpha)$.
2: $maxResult = BoundedMax(\mathcal{H}_{i-1}, j)$
3: $v_{in} = maxResult.value$
4: Insert a new record $\langle RANK, v_{in} \rangle$ for the new path into $vEB_{(\alpha,col)}$ and $vEB_{(\alpha,row)}$.
5: Compute the new record's current value v at the end of the block $\mathcal{T}[i,j]$ where $v = v_{in} + \min(|\hat{Y}_i|, |\hat{X}_j|)$
6: Compute the *value* v_{left} of the record which is the predecessor of the new record.
7: **if** $v < v_{left}$ **then**
8: $\quad$ delete the new record.
9: **else**
10: $\quad$ Compute the *value* v_{right} of the record which is the successor of the new record. If $v_{right} < v$, delete the old record. Continue until a record with a greater v_{right} is reached.
11: **end if**

Algorithm 3 *Step II* Finding the highest score of the forced paths on column j, below row i

1: Compute $\mathcal{Z} = COUNT_{\hat{Y}}^{i}(\alpha) - COUNT_{\hat{X}}^{j}(\alpha)$
2: Compute the *value* C of the successor of $\mathcal{Z}$ in the $vEB_{(\alpha,col)}$.
3: If there is a node, for which $\mathcal{Z} = RANK$, then that node is removed from the $vEB_{(\alpha,col)}$.

Algorithm 4 $Step\ III$ Finding the highest score of the forced paths on row i, left to column j

1: it is computed in an analogous way to Step II.

In our algorithm, we use $RANK$ as the key of the *vEB trees*. Recall that, this $RANK$ can be at least $-\ell$ and at most k. To represent the range $[-\ell, k]$ a total of $\log \omega$ bits are needed where $\omega = \ell + k$. Hence, for our *vEB trees*, any insertion, deletion or membership operation can be done in $O(\log\log\omega)$ time.

Precomputing the variables $LEFT$ and TOP as in precomputing $COUNT^j_{\hat{X}}(\alpha)$ and $COUNT^i_{\hat{Y}}(\alpha)$, $i \leq r < r' \leq \hat{k}$ and $j \leq c < c' \leq \hat{\ell}$ takes $O(\hat{k}+\hat{\ell})$ time. *"Algorithm Pre"* [17] requires $O((\hat{k}+\hat{\ell})+\mathcal{R}\log\log(\hat{k}\hat{\ell}))$ time. Each of the $2|\Sigma|$ *vEB trees* has at most $\mathcal{R}$ nodes, and any insertion, deletion or membership operation takes $O(\log\log\omega)$ time. The *BoundedHeap* data structure requires $O(\log\log\omega)$ for each operation.

In $Step\ I$ (Algorithm 2), the $BoundedMax()$ operation in Line 2 requires $O(\log\log\omega)$. Line 4 also requires $O(\log\log\omega)$. Line 10 requires $O((\text{number of deletions})\log\log\omega)$. Since each deleted block must previously have been inserted, the total number of deletion is $O(\mathcal{R})$. So total complexity of $Step\ I$ is $O(\mathcal{R}\log\log\omega + \log\log\omega)$.

In Algorithm 1, $Steps\ II\ and\ III$ in Line 9 and Line 10 respectively are computed in $O(\log\log\omega)$, while $Step\ IV$ in Line 16 requires $O(1)$ time.

We perform $Steps\ I\ to\ IV$ in Algorithm 1, Line 8 to Line 16 and Line 17 for the $\mathcal{R}$ *matched blocks*. Therefore, $O((\hat{k}+\hat{\ell})+\mathcal{R}\log\log(\hat{k}\hat{\ell})+\mathcal{R}\log\log\omega)$ time suffices to compute the longest common subsequence of $\hat{X}$ and $\hat{Y}$. Again, if $\hat{k} = \hat{\ell} = \hat{n}$, then $O(\hat{n}+\mathcal{R}\log\log(\hat{n}))$.

From the above discussion, it is clear that *LCS_2RLE_I* solves Problem *LCS_2RLE* in $O((\hat{k}+\hat{\ell})+\mathcal{R}\log\log(\hat{k}\hat{\ell})+\mathcal{R}\log\log\omega)$ time. Again, if $\hat{k} = \hat{\ell} = \hat{n}$, then $O(\hat{n}+\mathcal{R}\log\log(\hat{n}))$.

Table II summarizes the time complexity of the relevant algorithms mentioned in Sub-Section I-A. It is evident from Table II that, only Sakai [8]'s algorithm is comparable to our algorithm. That algorithm has to deal with the term $(\hat{k}\hat{\ell})$ whereas our algorithm uses $\mathcal{R}$ to solve the same problem. In the very worst case $\mathcal{R}$ is $O(\hat{k}\hat{\ell})$, which can be at best 50% of $(\hat{k}\hat{\ell})$. The impact of $\log\log(\dots)$ term is mostly negligible. Hence our algorithm outperforms Sakai [8]'s algorithm.

TABLE II
SUMMARY OF COMPLEXITY

Reference	Number Of RLE String	Time Complexity
This Algorithm	2	$O((\hat{k}+\hat{\ell})+\mathcal{R}\log\log(\hat{k}\hat{\ell})+\mathcal{R}\log\log\omega)$
Mitchell [5]	2	$O((\mathcal{R}+\hat{k}+\hat{\ell})\log(\mathcal{R}+\hat{k}+\hat{\ell}))$
Ann et al. [6]	2	$O(\hat{k}\hat{\ell}+\min(p_1,p_2))$
Apostolico et al. [7]	2	$O(\hat{k}\hat{\ell}\log(\hat{k}\hat{\ell}))$
Sakai [8]	2	$O(\hat{k}\hat{\ell}\log\log(\min(\hat{k},\hat{\ell},k/\hat{k},l/\hat{\ell},X)))$
Liu et al. [9]	1	$O(\min(k\hat{\ell},\ell\hat{k}))$
Ahsan et al. [10]	1	$O(\ell+\mathcal{R}\log\log\hat{k})$

V. CONCLUSION

Measuring the similarity between two strings through longest common subsequence is one of the fundamental problems in computer science. In this paper, we develop an efficient algorithm for solving the longest common subsequence problem between two run-length-encoded strings. Our algorithm runs in $O((\hat{k}+\hat{\ell})+\mathcal{R}\log\log(\hat{k}\hat{\ell})+\mathcal{R}\log\log\omega)$ time, where $\hat{k}$ and $\hat{\ell}$ are the length of run-length-encoded string $\hat{Y}$ and $\hat{X}$ respectively, $\omega = k+\ell$ where k and ℓ are the length of the uncompressed version of the two strings respectively and $\mathcal{R}$ is the total number of ordered pairs of positions at which the two run-length-encoded strings match. If $\hat{k} = \hat{\ell} = \hat{n}$, then our algorithm works in $O(\hat{n}+\mathcal{R}\log\log(\hat{n}))$ time. Mitchell proposed an $O((\mathcal{R}+\hat{k}+\hat{\ell})\log(\mathcal{R}+\hat{k}+\hat{\ell}))$ algorithm [5], Apostolico et al. [7] gave another $O(\hat{k}\hat{\ell}\log(\hat{k}\hat{\ell}))$ algorithm for this problem. Ann et al. [6] also proposed an $O(\hat{k}\hat{\ell}+\min(p_1,p_2))$ algorithm for solving the same problem, where p_1, p_2 denote the number of elements in the bottom and right boundaries of the matched blocks respectively. Our work obviously outperforms these algorithms.

Our future plan is to find a sophisticated solution for the *constrained LCS* problem [19] by applying this technique.

REFERENCES

[1] E. F. E. Khalid Sayood, *Introduction to Data Compression*. Morgan Kaufmann Publishers Inc, 2000.

[2] V. Freschi and A. Bogliolo, "Longest common subsequence between run-length-encoded strings: a new algorithm with improved parallelism," *Inf. Process. Lett.*, vol. 90, no. 4, pp. 167–173, 2004.

[3] C. S. Iliopoulos and M. S. Rahman, "A new efficient algorithm for computing the longest common subsequence," *Theory Comput. Syst.*, vol. 45, no. 2, pp. 355–371, 2009.

[4] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest subsequences." *Commun. ACM*, vol. 20, no. 5, pp. 350–353, 1977.

[5] J. S. B. Mitchell, "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.

[6] H.-Y. Ann, C.-B. Yang, C.-T. Tseng, and C.-Y. Hor, "A fast and simple algorithm for computing the longest common subsequence of run-length encoded strings," *Inf. Process. Lett.*, vol. 108, no. 6, pp. 360–364, 2008.

[7] A. Apostolico, G. M. Landau, and S. Skiena, "Matching for run-length encoded strings," *J. Complexity*, vol. 15, no. 1, pp. 4–16, 1999.

[8] Y. Sakai, "Computing the longest common subsequence of two run-length encoded strings," in *ISAAC*, 2012, pp. 197–206.

[9] J. J. Liu, Y.-L. Wang, and R. C. T. Lee, "Finding a longest common subsequence between a run-length-encoded string and an uncompressed string," *J. Complexity*, vol. 24, no. 2, pp. 173–184, 2008.

[10] M. S. R. Shegufta Bakht Ahsan, Tanaeem M. Moosa and S. Shahriyar, "Computing a longest common subsequence of two strings when one of them is run length encoded," *INFOCOMP Journal of Computer Science*, vol. 10, no. 3, pp. 48–55, 2011.

[11] E. Farhana and M. S. Rahman, "Doubly-constrained lcs and hybrid-constrained lcs problems revisited," *Inf. Process. Lett.*, vol. 112, no. 13, pp. 562–565, 2012.

[12] M. R. Alam and M. S. Rahman, "The substring inclusion constraint longest common subsequence problem can be solved in quadratic time," *J. Discrete Algorithms*, vol. 17, pp. 67–73, 2012.

[13] W. R. Pearson and D. J. Lipman, "Improved tools for biological sequence comparison," *Proceedings of National Academy of Science, USA*, vol. 85, pp. 2444–2448, 1988.
[14] S. F. Altschul, W. Gish, W. Miller, E. W. Meyers, and D. J. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403–410, 1990.
[15] C. S. Iliopoulos and M. S. Rahman, "New efficient algorithms for the lcs and constrained lcs problems," *Inf. Process. Lett.*, vol. 106, no. 1, pp. 13–18, 2008.
[16] P. V. E. Boas, "Preserving order in a forest in less than logarithmic time and linear space," *Information Processing Letters*, vol. 6, pp. 80–82, 1977.
[17] M. S. Rahman and C. S. Iliopoulos, "Algorithms for computing variants of the longest common subsequence problem." in *ISAAC*, 2006, pp. 399–408.
[18] G. S. Brodal, K. Kaligosi, I. Katriel, and M. Kutz, "Faster algorithms for computing longest common increasing subsequences," in *CPM*, 2006, pp. 330–341.
[19] Yin-Te and Tsai, "The constrained longest common subsequence problem," *Information Processing Letters*, vol. 88, no. 4, pp. 173 – 176, 2003.

Shegufta Bakht Ahsan received his B.Sc. degree from the Department of Computer Science and Engineering (CSE), Bangladesh University of Engineering and Technology (BUET), Dhaka, Bangladesh, in 2011. He then joined Samsung Research Institute Bangladesh. He is currently a PhD student of University of Illinois ar Urbana-Champaign. His research interests involve Bioinformatics and various areas of Networking and Systems.

Syeda Persia Aziz received her B.Sc. in Computer Science and Engineering (CSE) from Bangladesh University of Engineering and Technology (BUET), Dhaka, Bangladesh, in 2011. She then joined Samsung Research Institute Bangladesh as a Software Engineer. She is a graduate student of University of Illinois. Her research interests focus mainly on Networking and Bioinformatics.

M. Sohel Rahman received his B.Sc. Engg. degree from the Department of Computer Science and Engineering (CSE), Bangladesh University of Engineering and Technology (BUET), Dhaka, Bangladesh, in 2002 and the M.Sc. Engg. degree from the same department, in 2004. He received his Ph.D. degree from the Department of Computer Science, King's College London, UK in 2008. He is currently a Professor in the Department of CSE, BUET. His research interests include String and sequence algorithms, Bioinformatics, Musicolgy, Design and analysis of Algorithms, Meta-heuristics etc.