Effective Sparse Dynamic Programming Algorithms for Merged and Block Merged LCS Problems

AHM Mahfuzur Rahman[a], M. Sohel Rahman[a]

[a] AℓEDA Group, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology (BUET) Dhaka-1000, Bangladesh

***Abstract*— The longest common subsequence problem has been widely studied and used to find out the relationship between sequences. In this paper, we study the interleaving relationship between sequences, that is, we measure the relationship among three sequences, where two of those are interleaved in different ways and then the interleaved sequences are subsequently matched with the third remaining sequence, as pairwise longest common subsequence problems. Given a target sequence T and two other sequences A and B, we need to find out the LCS between $M(A,B)$ and T, where $M(A,B)$ denotes the merged-sequence which consists of A and B. We first present an improved $O((\mathcal{R}r + \mathcal{P}m)\log\log r)$ time algorithm, where we consider only the matches between sequences; here $|T| = n$, $|A| = m$ and $|B| = r$ $(m \geq r)$; $\mathcal{R}$ is the total number of ordered pairs of positions at which the two strings A and T match and $\mathcal{P}$ denotes the total number of ordered pairs of positions at which the two strings B and T match. Basing on the same idea, we also propose an improved algorithm to solve a block constrained variation of the problem. Running time of the blocked version is $O(\max\{\mathcal{R}\beta\log\log r, \mathcal{P}\alpha\log\log r\})$, where α denotes the number of blocks in A and β denotes the number of blocks in B. However, these improved algorithms do not provide best output in every cases. Therefore we finally fabricate a hybrid algorithm which utilizes the advantages of our proposed algorithms and previous state of the arts algorithms to provide best output in every possible cases, from both time efficiency and space efficiency.**

***Index Terms*— Bioinformatics, Dynamic Programming, Longest Common Subsequence, Merged Sequence, Block Constraint**

I. Introduction

The *longest common subsequence*(LCS) problem is one of the most studied problems in computer science. It is a classical approach for measuring the similarity between sequences and has extensive applications in different areas of computer science such as file comparison, spelling error correction, speech recognition and specially in computational biology and bioinformatics. Given a sequence S, a subsequence S' of S can be obtained by deleting zero or more symbols from the sequence S. Given two sequences A and B, the LCS of A and B, denoted by $LCS(A,B)$, is one of the longest sequences $\bar{S}$, where $\bar{S}$ is a subsequence of both A and B. For example, if $A = agcat$ and $B = gac$ a longest common subsequence is $LCS(A,B) = ga$. A LCS may not be unique, as in this case, there are two other $LCSs$ namely gc and ac.

A. *Literature Review*

Algorithms for finding the LCS of two or more sequences have been studied extensively in the literature [1]–[6]. Most of these algorithms are based on either traditional dynamic programming [2] or sparse dynamic programming [3], [5], [7] approach. Recent trend also includes heuristic based solution of LCS problem [8]. Wagner and Fischer [2] used the traditional dynamic programming approach to give a solution which has $O(n^2)$ worst case running time and space, assuming $|A| = |B| = n$. Masek and Paterson [9] reduced the running time to $O(n^2/\log n)$ using "Four-Russians" technique [10].

Sparse dynamic programming can also be used to solve the LCS problem; algorithms of this kind depend on the nature of inputs. For example, Hirschberg [3] gave a sparse dynamic programming algorithm having a time complexity depending on the length of LCS; if $|LCS(A,B)| = p$, $|A| = m$ and $|B| = n$, where $m \leq n$, this algorithm [3] runs in $O((m+1-p)p\log n)$ time. Clearly, when p is close to m, this algorithm takes time much less than $O(n^2)$. Another interesting and perhaps more relevant parameter for this problem is $\mathcal{R}$, where $\mathcal{R}$ is the total number of ordered pairs of positions at which the two strings match. Hunt and Szymanski [6] presented an algorithm running in $O((\mathcal{R}+n)\log n)$, assuming $|A| = |B| = n$. 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. Recently Rahman and Illipoulos [7] have proposed an $O(\mathcal{R}\log\log n + n)$ time algorithm for computing LCS.

In addition to the traditional LCS problem, considerable works on the variants of LCS have been studied in the literature [7], [11]–[19]. Recently Huang et al. [20] proposed a new variant of LCS, the *merged LCS* (referred to as MLCS) which finds out the interleaving relationship between sequences. Another version of this problem that considers block constraint is named *block merged LCS* (referred to as BMLCS).

The motivation of interleaving relationship between sequences comes from biology, where two related concepts, doubly conserved synteny (DCS) and whole genome

doi:10.4304/jcp.9.8.1743-1754

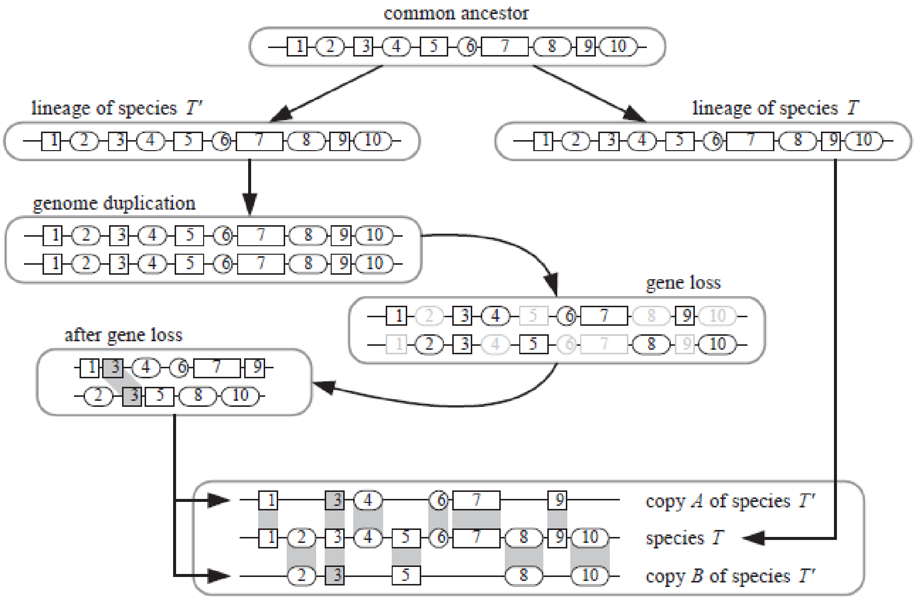


Figure 1. A simplified diagram of DCS block and WGD (borrowed from [21]).

duplication (WGD) are associated. The $synteny$ phenomenon means the order of specific genes in a chromosome is conserved over different organisms [22]–[24]. A typical example for this phenomenon can be seen between two yeast-species $Kluyveromyces\ waltii$ and $Saccharomyces\ cerevisiae$ [23]. By detecting the doubly conserved synteny (DCS) blocks of the two species, where each region of $K.\ waltii$ corresponds to two regions in $S.\ cerevisiae$, Kellis et al. [23] obtain the support for the whole-genome duplication (WGD). If we interleave two genomes from $S.cerevisiae$ and compare it with a $Kluyveromyces\ waltii$ genome, the mapping becomes evident. This resolves the controversy on the ancestry of yeast genome and can also be used to find the ancestry of other related genomes.

The MLCS problem can be used in signal comparison. For example, to identify the similarity between a complete voice (one series of signals without noise), and two incomplete voices (two series of signals sampled with different noise). A simple application of BMLCS is copy detection between texts. For example, we have some input text files and a target text file. Now, we want to compare how much of the texts in the target file are copied from the input files. Taking the words or lines in texts as blocks, we can check how much of the texts written are original to the writer and how much are copied from the input texts.

B. *Our Contribution*

We devise an algorithm for the $merged\ LCS$ problem using sparse dynamic approach [7]. Suppose that three sequences A,B and T are given as input. We use the notation $\mathcal{R}$ and $\mathcal{P}$ where $\mathcal{R}$ denotes ordered pairs of positions at which two strings A and T matches and $\mathcal{P}$ denotes the same for the pair of strings B and T. We first propose an algorithm which takes $O((\mathcal{R}r+\mathcal{P}m)\log\log r)$ time (we assume $m \geq r$) and $\theta(\max\{(\mathcal{R}+\mathcal{P}), r\})$ space for the MLCS problem. Later we formulate another algorithm for the BMLCS problem whose running time is $O(\max\{\mathcal{R}\beta\log\log r, \mathcal{P}\alpha\log\log r\})$ (α denotes the number of blocks in A and β denotes the number of blocks in B).

Huang et al. proposed traditional algorithms for MLCS and BMLCS problem [20] for three strings T, A and B. They proposed an algorithm to solve the MLCS problem in $O(nmr)$ time and space. For the BMLCS problem, they first proposed an algorithm that requires $O(n^2\gamma)$ time and space($\gamma = \max\{\alpha, \beta\}$). Later, they gave an algorithm using the concept of S-table [12], [25], [26] which takes $O(n^2 + n\gamma^2)$ space and time.

Furthermore they proposed a sparse dynamic programming approach to give a better solution of the MLCS and BMLCS problem [21]. They gave an $O(Lnr)$ time algorithm for MLCS problem which takes $O(Lmr)$ space, where $|A| = m, |B| = r, |T| = n$ and $L = |MLCS(A, B, T)|$. For the BMLCS problem, they proposed an algorithm which takes $O(L'n\gamma)$ time and $O(L'm\gamma)$ space, where $L' = |BMLCS(A, B, T)|$. The value of L and L' can be at most n.

Notably the running time of the algorithm in [21] is based on the LCS length, i.e, L and L'. On the other hand, our approach is based on the number of matches between A and T ($\mathcal{R}$) and between B and T ($\mathcal{P}$). To compare the two results, we need to investigate possible relations between the parameters and the inputs.

When the search space is sparse, that is there are insignificant number of matches ($\mathcal{R} \sim n$ and $\mathcal{P} \sim n$) and the MLCS length is considerably high

($L \sim n$), Huang's algorithm [21] gives the runtime of $O(n^2r)$ whereas our algorithm gives the runtime of only $O((nr + nm) \log\log r) = O(nm \log\log r)$ (assuming $m \geq r$) and outperforms both of the algorithms in [20] and [21] clearly. Besides, for the blocked version, Hunag's algorithm [21] gives the runtime of $O(n^2\gamma)$. But, our algorithm gives the runtime of $O(\max\{n\beta \log\log r, n\alpha \log\log r\}$. If we take $\alpha \geq \beta$, which is usually the case, then the runtime becomes $O(n\alpha \log\log r)$, which is clearly far better than the algorithms described in [20], [21].

To the contrary, when the search space is dense and the number of matches is pretty high but the MLCS length is significantly lower than n, Huang's algorithm [20], [21] will perform better than our algorithm. In the worst case, when the search space is full, $\mathcal{R}$ becomes $O(nm)$ and $\mathcal{P}$ becomes $O(nr)$. Therefore, due to the $\log\log r$ term, our algorithm will behave slightly worse than existing algorithms.

Space complexity of both our algorithms in the worst case ($\mathcal{R} = nm$ and $\mathcal{P} = nr$) is $\theta(\max\{nm, r\})$ ($m \geq r$). When search space is sparse ($\mathcal{R} \sim n$ and $\mathcal{P} \sim n$), space requirement is just $\theta(\max\{n, r\})$. The space complexity of previous best algorithms described in [21] are $O(Lmr)$ and $O(L'm\gamma)$ for the merged LCS and block merged LCS problem respectively. Therefore irrespective of the case, space complexities of our algorithms are better than the previous algorithms.

It is clearly visible that no single algorithm has the exclusive dominance in all possible cases which gives us the opportunity to work for a hybrid algorithm. We devise an algorithm which first decides the density of the search space depending on the number of matches. Then counting on that number, we take the decision to run the algorithm which will be more facilitating; this ensures our choice of best algorithm in every possible cases; the time and space complexity depends on the algorithm we choose.

The rest of the paper is organized as follows. In Section II, we present the preliminary concepts. Section III is devoted to a new sparse dynamic programming algorithm for the MLCS Problem. In Section IV, we devise a algorithm for BMLCS problem. We suggest a hybrid algorithm in Section V and finally conclude in Section VI.

II. Preliminaries

A. *Merged LCS*

Given a pair of merging sequences A and B and a target sequence T, the *merged LCS* problem, denoted by $MLCS(A, B, T)$, is to determine the LCS of T and merged sequence of A and B, denoted by $M(A, B)$. We define $A = a_1a_2 \ldots a_{|A|}$, $B = b_1b_2 \ldots b_{|B|}$ and $T = t_1t_2 \ldots t_{|T|}$.

$$\begin{aligned} MLCS(A, B, T) &= LCS(M(A, B), T) \\ &= LCS(M(a_1a_2 \ldots a_{|A|}, b_1b_2 \ldots \\ &\qquad b_{|B|}), t_1t_2 \ldots t_{|T|}) \end{aligned}$$

Note that, $M(A, B)$ is basically a set of merged sequences and $LCS(M(A, B), T)$ gives the longest sequence among all possible sequences where each of the sequence is a LCS between a sequence of the set $M(A, B)$ and T. Suppose that, we have the sequences $A = a_1a_2a_3a_4 = actt$ and $B = b_1b_2b_3 = ctg$ and the target sequence is $T = tcatcg$. To originally compute $MLCS(A, B, T)$, we have to check all $C(m + r, m)$ possible merging sequences (in this case, $C(m + r, m) = C(4 + 3, 4) = 35$). Here we show only three of all possible merging sequences within which we have an answer[1]. These sequences are: $M_1(A, B) = acctgtt = a_1b_1a_2b_2b_3a_3a_4$, $M_2(A, B) = acttctg = a_1a_2a_3a_4b_1b_2b_3$ and $M_3(A, B) = catcgtt = b_1a_1b_2a_2b_3a_3a_4$ and the output is: $MLCS(A, B, T) = LCS(M_3(A, B), T) = catcg$, whose length is 5.

B. *Block Merged LCS*

The *block merged LCS* problem arises when we impose block restriction on the sequences A and B. We denote the block merged sequence of A and B by $M^b(A, B)$ and the problem by $BMLCS(A, B, T)$. If α and β are the number of blocks in A and B respectively ,we define $A = A_1A_2 \ldots A_\alpha$ and $B = B_1B_2 \ldots B_\beta$.

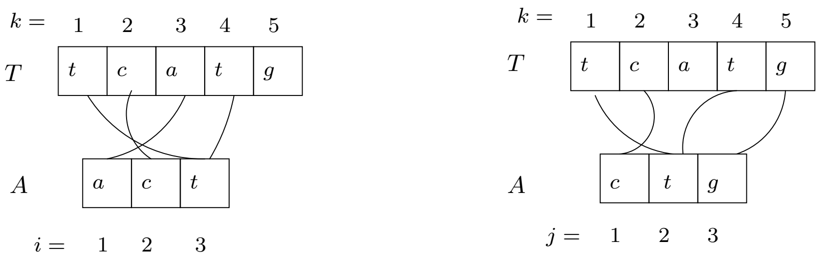
$$\begin{aligned} BMLCS(A, B, T) &= LCS(M^b(A, B), T) \\ &= LCS(M^b(A_1A_2 \ldots A_\alpha, B_1B_2 \ldots \\ &\qquad B_\beta), t_1t_2 \ldots t_{|T|}) \end{aligned}$$

Here, $M^b(A, B)$ are calculated exactly the same way it was done in case of merged sequences except that, A and B are blocked sequences. We had $A = actt$ in the previous example. Now, we divide $A = actt$ into two blocks $A_1 = ac$ and $A_2 = tt$, represented as $A = A_1A_2 = ac\#tt\#$. Note that, $'\#'$ denotes the end of the block only and is not actually a part of the string A. So, $'\#'$ will not occur in T and will not affect the output. Suppose, $B = ctg$ is also divided into two blocks $B = B_1B_2 = ct\#g\#$.

In this case, the total number of possible block merged sequences are $C(\alpha + \beta, \alpha) = C(2 + 2, 2) = 6$; the sequences are: $M^b_1(A, B) = A_1B_1B_2A_2 = acctgtt$, $M^b_2(A, B) = A_1A_2B_1B_2 = acttctg$,$M^b_3(A, B) = B_1A_1A_2B_2 = ctacttg$, $M^b_4(A, B) = A_1B_1A_2B_2 = acctttg$, $M^b_5(A, B) = B_1A_1B_2A_2 = ctacgtt$ and $M^b_6(A, B) = B_1B_2A_1A_2 = ctgactt$. By checking all merging sequences of $M^b(A, B)$, one can see that both $bLCS(M^b_2(A, B), T) = atcg$ and $bLCS(M^b_3(A, B), T) = ctcg$ are optimal solutions of

[1] The list of all possible merged sequences and their corresponding MLCS lengths are listed in Table 2 in the Appendix.

$$A = act$$
$$B = ctg$$
$$T = tcatg$$



$i = 1, A(1) = (a)$ matches at $k = 3$
$i = 2, A(2) = (c)$ matches at $k = 2$
$i = 3, A(3) = (t)$ matches at $k = 1, 4$

$j = 1, B(1) = (c)$ matches at $k = 2$
$j = 2, B(2) = (t)$ matches at $k = 1, 4$
$j = 3, B(3) = (g)$ matches at $k = 5$

$$\mathcal{M}_i^{\mathcal{A}} = \{(1,3),(2,2),(3,1),(3,4)\}$$
$$\mathcal{M}_j^{\mathcal{B}} = \{(1,2),(2,1),(2,4),(3,5)\}$$

Figure 2. Construction of $\mathcal{M}_i^{\mathcal{A}}$ and $\mathcal{M}_j^{\mathcal{B}}$ from A,B and T

$BMLCS(A, B, T)$. Note that for this case, the merging sequence $M_3(A, B) = b_1a_1b_2a_2b_3a_3a_4$ in our previous example cannot be obtained from any $M^b(A, B)$ as $M_3(A, B)$ does not conserve block information.

In this paper, we denote a_p as the pth character of the string A. For example, if $A = actt$, then $a_0 = \epsilon$ (empty string), $a_1 = a$, $a_2 = c$ etc. This is also true for the other strings B and T.

We use the following notions: a pair $(i, k), 1 \leq i \leq |A|, 1 \leq k \leq |T|$ defines a match if $a_i = t_k$; similarly $(j, k), 1 \leq j \leq |B|, 1 \leq k \leq |T|$ defines a match if $b_j = t_k$. Let,

$${\mathcal{M}_i}^A = \{(i, k) | a_i = t_k, 1 \leq i \leq |A|, 1 \leq k \leq |T|\}$$
$${\mathcal{M}_j}^B = \{(j, k) | b_j = t_k, 1 \leq j \leq |B|, 1 \leq k \leq |T|\}$$

Given A, B and T, we can calculate the matches $\mathcal{M}_i^{\mathcal{A}}$ and $\mathcal{M}_j^{\mathcal{B}}$ as shown in Figure 2. We define the set of all matches, $\mathcal{M}$, as $\mathcal{M} = \mathcal{M}_i^{\mathcal{A}} \cup \mathcal{M}_j^{\mathcal{B}}$. We also define, $|\mathcal{M}| = \mathcal{R} + \mathcal{P}$, where $\mathcal{R} = |\mathcal{M}_i^{\mathcal{A}}|$ and $\mathcal{P} = |\mathcal{M}_j^{\mathcal{B}}|$.

III. Algorithm for Merged LCS Problem

In this section, we present a new algorithm for the MLCS problem. We use the traditional dynamic programming formulation for MLCS presented in [20]. We define $L(i, j, k)$ $(0 \leq i \leq |A|,\ 0 \leq j \leq |B|,\ 1 \leq k \leq |T|)$ as follows:

$$L(i, j, k) = LCS(M(a_1a_2 \ldots a_i, b_1b_2 \ldots b_j), t_1t_2 \ldots t_k).$$

We have the following dynamic programming formulation for the MLCS problem from [20]:

$$L(i, j, k) =$$
$$\max \begin{cases} L(i-1, j, k-1) + 1 & \text{if } a_i = t_k \\ L(i, j-1, k-1) + 1 & \text{if } b_j = t_k \\ \max \begin{cases} L(i, j, k-1) \\ L(i-1, j, k) \\ L(i, j-1, k) \end{cases} & \text{if } a_i \neq t_k \text{ and } b_j \neq t_k \end{cases} \tag{1}$$

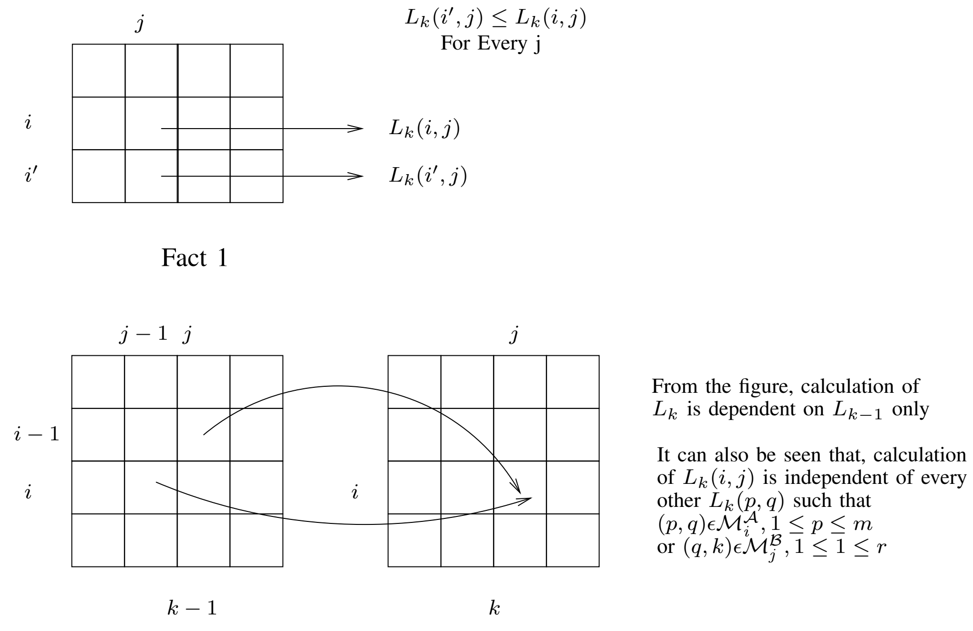
We assume $|T| = n$, $|A| = m$ and $|B| = r$. For simplicity, we can take that $n > m$ and $n > r$. We can also assume that $m \geq r$. This does not lose generality because when this is not the case, we can exchange A and B.

It is quite straightforward to give an $O(nmr)$ algorithm to implement Equation 1. Our goal is to present a new algorithm using sparse dynamic programming. From the dynamic programming concept of finding LCS we know that, the positions where there are no matches do not have any contribution to the length of LCS. This is in fact the base of sparse dynamic programming. Keeping the concept in mind, we reformulate Equation 1 as follows:

$$\nu^1_{(i,j,k)} = \max_{\substack{0 \leq \ell_i < i \\ 0 \leq \ell_j \leq j \\ 1 \leq \ell_k < k \\ (\ell_i, \ell_k), (\ell_j, \ell_k) \in \mathcal{M}}} L(\ell_i, \ell_j, \ell_k) \tag{2}$$

$$\nu^2_{(i,j,k)} = \max_{\substack{0 \leq \ell_i \leq i \\ 0 \leq \ell_j < j \\ 1 \leq \ell_k < k \\ (\ell_i, \ell_k), (\ell_j, \ell_k) \in \mathcal{M}}} L(\ell_i, \ell_j, \ell_k) \tag{3}$$

$$L(i, j, k) = \max \begin{cases} \nu^1_{(i,j,k)} + 1 & \text{if } a_i = t_k \\ \nu^2_{(i,j,k)} + 1 & \text{if } b_j = t_k \end{cases} \tag{4}$$



Fact 1

From the figure, calculation of L_k is dependent on L_{k-1} only

It can also be seen that, calculation of $L_k(i,j)$ is independent of every other $L_k(p,q)$ such that $(p,q)\epsilon\mathcal{M}_i^{\mathcal{A}}, 1 \le p \le m$ or $(q,k)\epsilon\mathcal{M}_j^{\mathcal{B}}, 1 \le 1 \le r$

Fact 2 and 3

Figure 3. Facts for MLCS Problem.

In our algorithm, referred to as Algorithm-MLCS henceforth, we use a preprocessing step requiring $O((\mathcal{R}+\mathcal{P})\log\log n+n)$ (Algorithm 1 in [1]) to calculate the set $\mathcal{M}_i^{\mathcal{A}}$ and $\mathcal{M}_j^{\mathcal{B}}$ in row by row order. Then the algorithm processes each $(i,k) \in \mathcal{M}_i^{\mathcal{A}}$ and each $(j,k) \in \mathcal{M}_j^{\mathcal{B}}$ using Equations 2 to 4. $L(i,j,k)$ can be thought of as a three dimensional matrix having dimensions $m+1, r+1$ and n. It would be more useful to visualize it as n two dimensional matrices instead of a three dimensional matrix. To highlight this view, we use the notation $L_k, 1 \le k \le n$ to denote the kth two dimensional matrix. With this new notation, we have $L(i,j,k) = L_k(i,j), 0 \le i \le m, 0 \le j \le r, 1 \le k \le n$. The implementation of the Equations 2 to 4 utilizes the following facts :

Fact 1: Suppose $(i,k) \in \mathcal{M}_i^{\mathcal{A}}$(*resp.* $(j,k) \in \mathcal{M}_j^{\mathcal{B}}$). *Then for all* $(i',j,k), i' > i$(*resp.* $(i,j',k), j' > j$), *we must have* $L_k(i',j) \ge L_k(i,j)$(*resp.* $L_k(i,j') \ge L_k(i,j)$).

While calculating the value of $L_k(i',j)$, $i' > i$, LCS length may increase but it will never decrease which provides the basis for the greater than or equal relationship, $L_k(i',j) \ge L_k(i,j)$. Fact 1 also implies that, if we calculate all the j values ($0 \le j \le r$) for the row i', then we do not need the values of row i further, where $i = i'-1, i'-2, \ldots, 0$.

Fact 2: The calculation of the matrix $L_k, 1 \le k \le n$ *is independent of any matrix* L_p *such that* $p < k-1$.

From Equations 1 to 4 , we can see that calculation of L_k is dependent only on L_k and L_{k-1}. Any L_p, where $p < k-1$ is therefore unnecessary in the calculation of L_k.

Fact 3: The calculation of a $L_k(i,j), 0 \le i \le m, 0 \le j \le r, 1 \le k \le n$ *is independent of* $L_k(p,q)$, *such that* $(p,k) \in \mathcal{M}_i^{\mathcal{A}}, 1 \le p \le m$ *or* $(q,k) \in \mathcal{M}_j^{\mathcal{B}}$, $1 \le q \le r$.

If we consider for the matches only ($\mathcal{M}_i^{\mathcal{A}}$ or $\mathcal{M}_j^{\mathcal{B}}$), then from Equations 2 to 4, we can see that the calculation of L_k considers only L_{k-1} values. L_k values were necessary in calculation of LCS in Equation 1 to propagate the solution, which in our case will be done by the assignment step of our algorithm. Figure 3 explains all the three facts in detail. Along with the above facts, we use the BoundedHeap data structure of [27], which supports the following operations:

$Insert(\mathcal{H},\ pos\ ,val\ ,data)$: Insert into the BoundedHeap $\mathcal{H}$ the position pos with value val and associated information $data$.

$IncVal(\mathcal{H},\ pos,\ val,\ data)$: If $\mathcal{H}$ does not already contain the position pos, perform $Insert(\mathcal{H},\ pos\ ,val\ ,data)$. Otherwise, set this position's value to $\max\{val, val'\}$, where val' is its previous value. Also, change the $data$ field accordingly.

$BoundedMax(\mathcal{H},\ pos)$: Return the item (with additional $data$) that has maximum value among all items in $\mathcal{H}$ with position smaller than pos. If $\mathcal{H}$ does not contain any items with position smaller than pos, return 0.

If we have more than one position where the maximum value can be found, then note that, $BoundedMax()$ returns the one having minimum among these positions, that is (i,j,k) with the minimum j value. For example, if we have elements in $\mathcal{H}_2^1$ $(k=1, i=2)$ as $(0,0,2,2,1,0)$

with indexes $0, 1, 2, 3, 4, 5$ (j) respectively then a call as $BoundedMax(\mathcal{H}_2^1, 2)$ will return $j = 2$, even if the answers could be either 2 or 3, as 2 is minimum. The following theorem from [27] presents the time complexity of the BoundedHeap data structure.

Theorem 1 ([27]): BoundedHeap data structure can support each of the above operations in $O(\log\log n)$ amortized time, where keys are drawn from the set $\{1, \ldots, n\}$. The data structure requires $O(n)$ space.

We have n $(k = 1, \ldots, n)$ two dimensional matrices in our hand. We consider the input A along the row and B along the column of each two dimensional matrices. So, each of the two dimensional matrices consists of $m + 1$ $(i = 0, \ldots, m)$ rows and in each row we have $r + 1$ $(j = 0, \ldots, r)$ elements.

Algorithm-MLCS proceeds as follows. We consider all the matches of row i for all k two dimensional matrices. We consider Row $i+1$ only when the calculation of row i is completed for all $L_k, 1 \leq k \leq n$. We need to deal with four BoundedHeap data structures $\mathcal{H}_i^k, \mathcal{H}_{i-1}^k, \mathcal{H}_i^{k-1}$ and $\mathcal{H}_{i-1}^{k-1}$. Each of the bounded heap, $\mathcal{H}_i^k$, contains values for $LCS(M(a_1, a_2 \ldots a_i, b_0), t_1 t_2 \ldots t_k)$, $LCS(M(a_1, a_2 \ldots a_i, b_0 b_1), t_1 t_2 \ldots t_k), \ldots$ $LCS(M(a_1, a_2 \ldots a_i, b_0 b_1 \ldots b_j), t_1 t_2 \ldots t_k)$ where $0 \leq j \leq r$. So, a bounded heap requires $O(r)$ space.

At first $\mathcal{H}_i^k$ is initialized to $\max\{\mathcal{H}_{i-1}^k, \mathcal{H}_i^{k-1}\}$. If we consider the match $(i, k) \in \mathcal{M}_i, 1 \leq i \leq m, 1 \leq k \leq n$, we will need $\mathcal{H}_{i-1}^{k-1}$ to calculate the value $L_k(i, j)$ as follows:

$$L_k(i, j).Val = BoundedMax(\mathcal{H}_{i-1}^{k-1}, j + 1).Val + 1$$
$$L_k(i, j).Prev = BoundedMax(\mathcal{H}_{i-1}^{k-1}, j + 1).Pos$$

Note that, we have not considered $i = 0$ for a (i, k) match. At, $i = 0$, $a_0 = \epsilon$, an empty string and k starts from 1. So, T can never contain an empty string ϵ and a match is impossible for $i = 0$. Similarly, we do not need to consider $j = 0$ for a (j, k) match. If we consider a match $(j, k) \in \mathcal{M}_j, 1 \leq j \leq r, 1 \leq k \leq n$, we will need $\mathcal{H}_i^{k-1}$ and calculate the value $L_k(i, j)$ as follows:

$$L_k(i, j).Val = BoundedMax(\mathcal{H}_i^{k-1}, j).Val + 1$$
$$L_k(i, j).Prev = BoundedMax(\mathcal{H}_i^{k-1}, j).Pos$$

We have to perform the following operation in both of the cases:

$$IncVal(\mathcal{H}_i^k, j, L_k(i, j).Val, (i, j, k))$$

To keep track of the best MLCS found so far, we use a global variable named $glMLCS$. It stores the best MLCS length found so far in $glMLCS.Val$ and the corresponding position in $glMLCS.Pos$. If the relation $glMLCS.Val < L_k(i, j).Val$ becomes true, we update the $glMLCS$ as follows:

$$glMLCS.Val = L_k(i, j).Val$$
$$glMLCS.Pos = (i, j, k)$$

Finally, when the calculation of a $\mathcal{H}_i^k, 1 \leq i \leq m, 1 \leq n \leq n$ ends, we can delete $\mathcal{H}_{i-1}^{k-1}$ as it will not be necessary in any calculation further. The algorithm is formally presented in the form of Algorithm 1

The correctness of the above procedure follows from Facts 1 to 3 and Equation 1 to Equation 4. Due to Fact 1 , as soon as we compute the L_k value of a new match in a column j, we can forget the previous matches of that column. After the computation of any $L_k(i, j)$ in Row i, we insert it into $\mathcal{H}_i^k, 1 \leq k \leq n$ to update it for the calculation of next Row $i + 1$. Due to Facts 2 and 3, we use $\mathcal{H}_{i-1}^{k-1}$ in case of (i, k) matches and $\mathcal{H}_i^{k-1}$, in case of (j, k) matches to calculate $L_k(i, j)$, where $1 \leq k \leq n$,$1 \leq i \leq m$ and $1 \leq j \leq r$.

Algorithm 1 $AlgorithmMLCS(A, B, T)$

```
 1: Construct the set $\mathcal{M}$ using Algorithm 1 of [1]. Let
    $\mathcal{M}_i^{\mathcal{A}} = (i, k)$ and $\mathcal{M}_j^{\mathcal{B}} = (j, k)$, $1 \leq i \leq m, 1 \leq j \leq r, 1 \leq k \leq n$
 2: $glMLCS.Pos = \epsilon$
 3: $glMLCS.Val = \epsilon$
 4: $\mathcal{H}_0^0 = \mathcal{H}_{-1}^0 = \mathcal{H}_{-1}^1 = \epsilon$
 5: for $i = 0$ $m$ do
 6:   for $k = 1$ $n$ do
 7:     $\mathcal{H}_i^k = \max(\mathcal{H}_i^{k-1}, \mathcal{H}_{i-1}^k)$
 8:     for each $(j, k) \in \mathcal{M}_j$ do
 9:       $max = BoundedMax(\mathcal{H}_i^{k-1}, j)$
10:       $L_k(i, j).Val = max.Val + 1$
11:       $L_k(i, j).Prev = max.Pos$
12:       $IncVal(\mathcal{H}_i^k, max.Val, (i, j, k))$
13:       if $glMLCS.Val < L_k(i, j).Val$ then
14:         $glMLCS.Val = L_k(i, j).Val$
15:         $glMLCS.Pos = (i, j, k)$
16:       end if
17:     end for
18:     for each $(i, k) \in \mathcal{M}_i$ do
19:       for $j \leftarrow 0$ $r$ do
20:         $max = BoundedMax(\mathcal{H}_{i-1}^{k-1}, j + 1)$
21:         $L_k(i, j).Val = max.Val + 1$
22:         $L_k(i, j).Prev = max.Pos$
23:         $IncVal(\mathcal{H}_i^k, max.Val, (i, j, k))$
24:         if $glMLCS.Val < L_k(i, j).Val$ then
25:           $glMLCS.Val = L_k(i, j).Val$
26:           $glMLCS.Pos = (i, j, k)$
27:         end if
28:       end for
29:     end for
30:     Delete $\mathcal{H}_{i-1}^{k-1}$
31:   end for
32: end for
33: return $glMLCS$
```

A. *Runtime Analysis*

Now, we analyze the runtime of the Algorithm 1. The first line of the algorithm, which consists the preprocessing step requires $O((\mathcal{R}+\mathcal{P})\log\log n+n)$ time to compute the set $\mathcal{M}_i^{\mathcal{A}}$ and $\mathcal{M}_j^{\mathcal{B}}$.

In the next part, we have either a $\mathcal{M}_i^{\mathcal{A}}$ or $\mathcal{M}_j^{\mathcal{B}}$ match and we need to use two operations, $BoundedMax()$ and $IncVal()$, for each of the match. The operations take amortized $O(\log\log r)$ time. We know that, there are $\mathcal{R}$ number of (i,k) matches and $\mathcal{P}$ number of (j,k) matches. Now a (j,k) match can occur for every i, $1 \leq i \leq m$. Similarly, a (i,k) match can occur for every j values, $1 \leq j \leq r$. So, the total running time is $O((\mathcal{R}r + \mathcal{P}m)\log\log r)$.

For the space complexity, to store the matches in memory, we need $\theta(\mathcal{R} + \mathcal{P})$ space. To calculate the MLCS, we need to store all the $L_k(i,j).Pos, (i,k) \in \mathcal{M}_i^{\mathcal{A}}, (j,k) \in \mathcal{M}_j^{\mathcal{B}}$ values. If we take space for one $L_k(i,j).Pos$ to be constant, then, as there are in total $(\mathcal{R}+\mathcal{P})$ matches, this process also requires $\theta(\mathcal{R}+\mathcal{P})$ space. Again, We need to keep track of four Bounded Heap data structures, requiring only $O(r)$ space. So, the total space requirement becomes $\theta(max(\mathcal{R}+\mathcal{P}), r)$.The time and space complexity of Algorithm-MLCS is summarized in the following theorem.

Theorem 2: Algorithm-MLCS runs in $O((\mathcal{R}r + \mathcal{P}m)\log\log r)$ time and $\theta(\max\{(\mathcal{R}+\mathcal{P}), r\})$ space.

If the number of matches becomes $O(n)$, the runtime reduces to $O(n(r+m)\log\log r) = O(nm\log\log r)$ $(m \geq r)$ and our algorithm works far better than the previous best runtime of $O(Lnr)$. To the contrary, the worst case value of $\mathcal{R}$ is $O(nm)$ and $\mathcal{P}$ is $O(nr)$ and consequently the worst case running time of our algorithm becomes $O(nmr\log\log r)$. A detailed performance comparison is provided in the appendix.

B. Performance Comparison

Here, we compare the performance of Algorithm-MLCS(Algorithm 1) with the previous algorithms in [20], [21] and show different cases with illustrative examples.

As provided in Theorem 2, our proposed MLCS algorithm takes $O((\mathcal{R}r + \mathcal{P}m)\log\log r)$ time and $\theta(\max\{(\mathcal{R}+\mathcal{P}), r\})$ space to calculate $MLCS(A,B,T)$. The traditional algorithm described in [20] calculates the MLCS in $O(nmr)$ time and space. Sparse Dynamic Programming based on the length of MLCS discussed in [21] needs $O(Lnr)$ time and $O(Lmr)$ space, where $L = |MLCS(A,B,T)|$. In rest of the section, we discuss different cases of the MLCS problem and show how our algorithm performs in comparison with other approaches.

The worst case of our algorithm is observed when there is only one character in our alphabet or in the input sequence. For example, if the input sequences are $A = ttt$, $B = ttt$ and $T = tttt$, we have only one character t in our input sequences. So, every position $i, 1 \leq i \leq m$ in the sequence A and every position $j, 1 \leq j \leq r$ in the sequence B matches with every position $k, 1 \leq k \leq n$ in the target sequence T. As a result, the number of (i,k) matches $\mathcal{R}$ becomes $O(nm)$ and the number of (j,k) matches $\mathcal{P}$ becomes $O(nr)$. In this case, we obtain the maximum possible values for $\mathcal{R}$ and $\mathcal{P}$.

Our run time of $O((\mathcal{R}r + \mathcal{P}m)\log\log r)$ becomes $O(nmr\log\log r)$ in the worst case; space requirement becomes $\theta(\max\{nm, r\})$ $(m \geq r)$. We now see how the other two algorithms [20], [21] works in the worst case. The traditional dynamic prgoramming algorithm proposed by Huang et al. in [20] gives the running time and space of $O(nmr)$ in the worst case. In particular, this algorithm takes $O(nmr)$ time and space in every cases. Note that, the length of the output $MLCS(A,B,T)$ is $O(n)$ in the worst case. The algorithm described in [21] for MLCS problem runs in $O(Lnr)$ time and $O(Lmr)$ space. So, this algorithm takes $O(n^2r)$ time and and $O(nmr)$ space in the worst case. This is also the worst case runtime and space of this algorithm as the worst case value of L is n. Depending on the value of n, m and r, our runtime can be better or worse than the algorithm of [21]; our space complexity is always better than both the algorithms [20], [21] though.

The runtime of Algorithm-MLCS depends on the number of matches $\mathcal{R}$ and $\mathcal{P}$. If the number of matches is not high ($R \sim n$ and $P \sim n$), the runtime reduces to $O(n(r+m)\log\log r) = O(nm\log\log r)$ $(m \geq r)$ and the space complexity reduces to $\theta(\max\{n, r\})$ and our algorithm works far better than the traditional runtime and space of $O(nmr)$.

The runtime of the algorithm discussed in [21] depends on the length of the output L. If the matches are $O(n)$ and the length of the output is $L \sim n$, then the algorithm works in $O(n^2r)$ time, but our algorithm works in $O(nm\log\log r)$ time in this case. Consequently Algorithm-MLCS clearly outperforms the sparse dynamic programming algorithm of [21] which depends on the length of the output. The space complexity of Algorithm-MLCS, which is $\theta(\max\{n, r\})$ also is much better than the space complexity of $O(nmr)$ of the algorithm described in [21].

We give an example of the best case of our algorithm here. Let the inputs for this case are $A = bba, B = tcg$ and $T = tcga$. We have only three (j,k) matches which gives $\mathcal{M}_j^{\mathcal{B}} = \{(1,1),(2,2),(3,3)\}$ and $|\mathcal{M}_j^{\mathcal{B}}| = \mathcal{P} = 3$, where $\mathcal{M}_j^{\mathcal{B}}$ is the set of matches between B and T. Again, we have only one (i,k) match so that $\mathcal{M}_i^{\mathcal{A}} = \{(3,4)\}$ and $|\mathcal{M}_i^{\mathcal{A}}| = \mathcal{R} = 1$, where $\mathcal{M}_i^{\mathcal{A}}$ is the set of matches between A and T. The output is $MLCS(A,B,T) = tcga$ and output length $|MLCS(A,B,T)| = 4$. So, output length L is $O(n)$ and the number of matches $\mathcal{R}$ and $\mathcal{P}$ are both less than n. Our algorithm works very efficiently for this kind of examples but the corresponding algorithms in [20], [21] performs considerably bad.

IV. Algorithm for Block Merged LCS Problem

In this section, we present a new algorithm for the BMLCS problem. We use the traditional dynamic programming formulation for BMLCS presented in [20]. We define $L^b(i,j,k)$ $(0 \leq i \leq m, 0 \leq j \leq r, 1 \leq k \leq n)$

as follows:

$$\begin{aligned} & L^b(i,j,k) \\ & = LCS(M^b(A_1A_2\dots A_\alpha, B_1B_2\dots B_\beta), t_1t_2\dots t_k). \end{aligned}$$

We have the following dynamic programming formulation for the BMLCS problem from [20]:

$$L^b(i,j,k) = \max \begin{cases} \max \begin{cases} L(i-1,j,k-1)+1 & \text{if } a_i = t_k \\ L(i-1,j,k) & \text{if } a_i \neq t_k \\ L(i,j-1,k-1)+1 & \text{if } b_j = t_k \\ L(i,j-1,k) & \text{if } b_j \neq t_k \\ L(i,j,k-1) & \text{if } a_i \neq t_k \text{ and } b_j \neq t_k \end{cases} & \text{if } a_i = \# \text{ and } b_j = \# \\ \max \begin{cases} L(i-1,j,k-1)+1 & \text{if } a_i = t_k \\ L(i-1,j,k) & \text{if } a_i \neq t_k \\ L(i,j,k-1) & \text{if } a_i \neq t_k \end{cases} & \text{if } a_i \neq \# \text{ and } b_j = \# \\ \max \begin{cases} L(i,j-1,k-1)+1 & \text{if } b_j = t_k \\ L(i,j-1,k) & \text{if } b_j \neq t_k \\ L(i,j,k-1) & \text{if } b_j \neq t_k \end{cases} & \text{if } a_i = \# \text{ and } b_j \neq \# \end{cases} \tag{5}$$

Note that, the condition $a_i \neq \#\ and\ b_j \neq \#$ is not considered in the equation, because it will be excluded by the following fact:

Fact 4: $L^b(i,j,k)$ *is independent of the positions* a_i *and* $b_k, 0 \leq i \leq m, 0 \leq j \leq r$ *when* $a_i \neq \{\#, \phi\}$ *and* $b_j \neq \{\#, \phi\}$, ϕ *deonotes empty string* (*when* $i = 0$ *or* $j = 0$).

Furthermore, we can exclude the part of Equation 5, when "$a_i = \#\ and\ b_j = \#$", as there is no possibility of a match here (the string T do not contain any block). In particular, the authors of [21] as well considers only three cases:

1. $(i,j) = (0,0)$
2. $a_i \notin \{\#, \phi\}$
3. $b_j \notin \{\#, \phi\}$

Additionally, as we are calculating only the matches, we can exclude all of the cases "$a_i \neq t_k$ and $b_j \neq t_k$". Therefore, we can reformulate Equation 5 with the help of Equations 2 and 3 as follows:

$$L^b(i,j,k) = \max \begin{cases} \nu^1_{(i,j,k)} + 1 & \text{if } a_i = t_k \& b_j \in \{\phi, \#\} \\ \nu^2_{(i,j,k)} + 1 & \text{if } b_j = t_k \& a_i \in \{\phi, \#\} \end{cases} \tag{6}$$

Now we can discuss our algorithm to solve BMLCS problem namely, Algorithm-BMLCS (Algorithm 2). Like Algorithm 1, we also need four *BoundedHeap* data structures here. Calculation of matches $\mathcal{M}^{\mathcal{A}}_i$ and $\mathcal{M}^{\mathcal{B}}_j$ and the initialization process remains the same. But in BMLCS, according to Fact 4 the condition $a_i \neq \#\ and\ b_j \neq \#$ never arises. Therefore, it is impossible to have the $\mathcal{M}^{\mathcal{A}}_i$ matches (pair of matches between A and T) and the $\mathcal{M}^{\mathcal{B}}_j$ matches (pair of matches between B and T) at the same iteration. When $a_i \in \{\phi, \#\}$ and $b_j \notin \{\phi, \#\}$, we consider the $(j,k) \in \mathcal{M}^{\mathcal{B}}_j$ matches and when $b_j \in \{\phi, \#\}$ and $a_i \notin \{\phi, \#\}$, we consider the $(i,k) \in \mathcal{M}^{\mathcal{A}}_i$ matches, $1 \leq i \leq m, 1 \leq j \leq r$ and $1 \leq k \leq n$. The calculation after getting a match is done similarly as we did in case of MLCS.

A. *Runtime Analysis*

To analyze the runtime of Algorithm 2, we need to consider $\mathcal{M}^{\mathcal{A}}_i, 1 \leq i \leq m$ only when $a_i \notin \{\phi, \#\}$. We calculate the change within a row i only when $b_j \in \{\phi, \#\}$ and there are $O(\beta)$ such cases in total. We need only $O(\beta)$ positions to update within a row, but for simplicity, we have taken the size of our bounded heap as $O(r)$ in spite of $O(\beta)$. For this reason, we need $O(\mathcal{R}\beta \log\log r)$ time here. Again when $a_i \in \{\phi, \#\}$, we will consider $\mathcal{M}^{\mathcal{B}}_j, 0 \leq j \leq r$. We have only $O(\alpha)$ of such a_i's. Here we will need bounded heap structure of space $O(r)$ each and the space necessity becomes $O(\mathcal{P}\alpha \log\log r)$. As both cases can not arrive at the same time, the total running time becomes $O(\max\{\mathcal{R}\beta \log\log r, \mathcal{P}\alpha \log\log r\})$.

We need to keep track of four Bounded Heap data structures, requiring only $O(r)$ space. The space complexity for the pre-calculation step remains also the same as Algorithm 1(Algorithm-MLCS). Time and space complexity of Algorithm-BMLCS is summarized in the following theorem.

Theorem 3: Algorithm-BMLCS runs in $O(\max\{\mathcal{R}\beta \log\log r, \mathcal{P}\alpha \log\log r\})$ time and in $\theta(\max\{(\mathcal{R} + \mathcal{P}), r\})$ space.

The worst case value of $\mathcal{R}$ is $O(nm)$ and $\mathcal{P}$ is $O(nr)$ and as a result, the worst case running time of our algorithm becomes $O(\max\{\beta nm \log\log r, \alpha nr \log\log r\})$ whereas the space requirement is: $\theta(\max\{nm, r\})$, $m \geq r$. If the number of matches are such that, $R \sim n$ and $P \sim n$, the runtime reduces to $O(\max\{\beta n \log\log r, \alpha n \log\log r\})$ and space requirement reduces to $\theta(\max\{n, r\})$ which is better than the previous best runtime of $O(L'n\gamma)$ and space complexity of $O(L'm\gamma)$, L or L' denoting the length of the result. A detailed performance comparison is provided in the appendix.

Algorithm 2 Algorithm-BMLCS(A,B,T)

```
Construct the set $\mathcal{M}$ using Algorithm 1 of [1]. Let $\mathcal{M}_i^{\mathcal{A}} = (i,k)$ and $\mathcal{M}_j^{\mathcal{B}} = (j,k)$, $1 \le i \le m, 1 \le j \le r, 1 \le k \le n$
$glBMLCS.Pos = \epsilon$
$glBMLCS.Val = \epsilon$
$\mathcal{H}_{\mathit{0}}^{\mathit{0}} = \mathcal{H}_{-\mathit{1}}^{\mathit{0}} = \mathcal{H}_{-\mathit{1}}^{\mathit{1}} = \epsilon$
for $i = 0$ $m$ do
  for $k = 1$ $n$ do
    $\mathcal{H}_i^k = \max(\mathcal{H}_i^{k-\mathit{1}}, \mathcal{H}_{i-\mathit{1}}^k)$
    if $a_i = \{\phi, \#\}$ and $b_j \neq \{\phi, \#\}$ then
      for each $(j,k) \in \mathcal{M}_j$ do
        $max = BoundedMax(\mathcal{H}_i^{k-\mathit{1}}, j)$
        $L_k(i,j).Val = max.Val + 1$
        $L_k(i,j).Prev = max.Pos$
        $IncVal(\mathcal{H}_i^k, max.Val, (i,j,k))$
        if $glBMLCS.Val < L_k(i,j).Val$ then
          $glBMLCS.Val = L_k(i,j).Val$
          $glBMLCS.Pos = (i,j,k)$
        end if
      end for
    else
      for each $(i,k) \in \mathcal{M}_i$ do
        for $j = 0$ $r$ do
          if $b_j = \{\phi, \#\}$ and $a_i \neq \{\phi, \#\}$ then
            $max = BoundedMax(\mathcal{H}_{i-\mathit{1}}^{k-\mathit{1}}, j+1)$
            $L_k(i,j).Val = max.Val + 1$
            $L_k(i,j).Prev = max.Pos$
            $IncVal(\mathcal{H}_i^k, max.Val, (i,j,k))$
            if $glBMLCS.Val < L_k(i,j).Val$ then
              $glBMLCS.Val = L_k(i,j).Val$
              $glBMLCS.Pos = (i,j,k)$
            end if
          end if
        end for
      end for
    end if
  end for
  Delete $\mathcal{H}_{i-\mathit{1}}^{k-\mathit{1}}$
end for
return $glBMLCS$
```

B. Performance Comparison

In this section, we compare the performance of Algorithm-BMLCS with the previous algorithms in [20], [21] and show different cases with illustrative examples.

As described in Theorem 3, Algorithm-BMLCS takes $O(\max\{\mathcal{R}\beta \log\log r, \mathcal{P}\alpha \log\log r\})$ time and $\theta(\max\{(\mathcal{R} + \mathcal{P}), r\})$ space to calculate $BMLCS(A,B,T)$. Two algorithms are described in [20]. First of them calculates the BMLCS in $O(n^2\gamma)$ time and space and the next and improved algorithm needs only $O(n^2 + n\gamma^2)$ time and space, where $\gamma = \max\{\alpha, \beta\}$. A sparse dynamic programming approach based on the length of BMLCS discussed in [21] needs $O(L'n\gamma)$ time and $O(L'm\gamma)$ space, where $L' = |BMLCS(A,B,T)|$.

The worst case of our algorithm is observed when there is only one character in the input sequence. For example, if the input sequences are $A = tt\#t\#$, $B = tt\#t\#$ and $T = tttt$, we have only one character t in our input sequences. Therefore, every position $i, 1 \le i \le m$ in the sequence A and every position $j, 1 \le j \le r$ in the sequence B matches with every position $k, 1 \le k \le n$ in the target sequence T. The number of (i,k) matches, $\mathcal{R}$ becomes $O(nm)$ and the number of (j,k) matches, $\mathcal{P}$ becomes $O(nr)$. We obtain the maximum possible values for $\mathcal{R}$ and $\mathcal{P}$ in this case.

Our run time of $O(\max\{\mathcal{R}\beta \log\log r, \mathcal{P}\alpha \log\log r\})$ in the worst case becomes $O(\max\{\beta nm \log\log r, \alpha nr \log\log r\})$; the space requirement in this case becomes $\theta(\max\{nm, r\})$, $m \ge r$. We now see how other two algorithms works in the worst case. The traditional dynamic programming algorithm proposed by Huang et al. in [20] gives the running time and space of $O(n^2\gamma)$ in the worst case and the improved algorithm [21] works in $O(n^2 + n\gamma^2)$ time and space. Note that, the length of the output $BMLCS(A,B,T)$ is $O(n)$. As the algorithm described in [21] for BMLCS problem runs in $O(L'n\gamma)$ time and $O(L'm\gamma)$ space, the algorithm takes $O(n^2\gamma)$ time and $O(nm\gamma)$ when the worst case arises. This is also the worst case runtime and space of this algorithm as the worst case value of L' is n. Depending on the value of n,m,r and γ, our algorithm may perform better or worse than this algorithm [21].

As the runtime of Algorithm-BMLCS depends mainly on the amount of matches, if the number of match between T, A and T, B is less, then our algorithm performs better. If the number of matches are such that, $R \sim n$ and $P \sim n$, the runtime reduces to $O(\max\{\beta n \log\log r, \alpha n \log\log r\})$ and space requirement reduces to $\theta(\max\{n, r\})$. Without the loss of generality, we can take that, $\alpha \ge \beta$ and the runtime becomes $O(\alpha n \log\log r)$. Our algorithm works far better than the traditional runtime of $O(n^2\gamma)$ or $O(n^2 + n\gamma^2)$ in such cases. Also, our space requirement of $\theta(\max\{n, r\})$ is much better than $O(n^2\gamma)$ or $O(n^2 + n\gamma^2)$.

The runtime of the algorithm for BMLCS problem discussed in [21] depends on the length of the output L'. If the matches are $O(n)$ and the length of the output is $L' \sim n$, then the algorithm works in $O(n^2\gamma)$ time and $O(nm\gamma)$ space. But, Algorithm-BMLCS works in $O(\alpha n \log\log r)$ in this case. As α is the number of blocks in A, it is usually very small compared to m, r or n. Also, $\gamma \ge \alpha$ and $n >> \log\log r$. So, our algorithm clearly outperforms the sparse dynamic programming of [21]. The space complexity of our algorithm, $\theta(\max\{n, r\})$ is also far better than the space complexity of $O(nm\gamma)$.

We give an example of the best case of our algorithm here. Let the inputs for this case are $A = bb\#a\#, B = tc\#g\#$ and $T = tcga$. We have only three (j,k) matches which gives $\mathcal{M}_j^{\mathcal{B}} = \{(1,1),(2,2),(3,3)\}$ and

$|\mathcal{M}_j^{\mathcal{B}}| = \mathcal{P} = 3$, where $\mathcal{M}_j^{\mathcal{B}}$ is the set of matches between B and T. Again we have only one (i, k) match so that $\mathcal{M}_i^{\mathcal{A}} = \{(3, 4)\}$ and $|\mathcal{M}_i^{\mathcal{A}}| = \mathcal{R} = 1$, where $\mathcal{M}_i^{\mathcal{A}}$ is the set of matches between A and T. The output is $BMLCS(A, B, T) = tcga$ and output length $|BMLCS(A, B, T)| = 4$. Consequently output length L is $O(n)$ and number of matches $\mathcal{R}$ and $\mathcal{P}$ are both less than n. Our algorithm works very efficiently for this type of examples where amount of matches is insignificant but the output length is about $O(n)$ whereas, the algorithms described in [20], [21] work considerably bad.

V. Hybrid Algorithm for MLCS and BMLCS

From the analysis of the algorithms proposed in this paper so far and the previous state of the art algorithms, we can say that none of the algorithms is a prominent winner in every cases. Therefore we provide a hybrid algorithm which exploits the best features from both of our algorithms and the previous algorithms. The new algorithm will first count the number of matches available for the input and target strings. Then depending on the number of matches, it will select our algorithm or the previous best algorithm. The method is described in Algorithm 3.

Algorithm 3 Hybrid-Algorithm(A,B,T,isMLCS)

```
Construct the set $\mathcal{M}$ using Algorithm 1 of [1]. Let $\mathcal{M}_i^{\mathcal{A}} = (i, k)$ and $\mathcal{M}_j^{\mathcal{B}} = (j, k)$, $1 \le i \le m, 1 \le j \le r, 1 \le k \le n$
Count the number of matches from $\mathcal{M}$
if number of matches, $\mathcal{M} \sim n$ then
    if isMLCS is true then
        run Algorithm 1.
    else
        run Algorithm 2.
    end if
else
    if isMLCS is true then
        run Algorithm 1 from [21].
    else
        run corresponding algorithm for BMLCS from [21].
    end if
end if
```

In Algorithm 3, isMLCS is a boolean variable which should be assigned true before calling the algorithm if we want to calculate the MLCS; otherwise it should be assigned false. Depending on the value of the inputs given to the hybrid-algorithm, it will calculate the LCS which should be better in performance in every cases which is not possible for the individual algorithms. The only overhead of this algorithm is to count the number of matches which can be done simply in $O((\mathcal{R} + \mathcal{P}) \log \log n + n)$ time. In the worst case the preprocessing takes about $O(n^2) \log \log n$ which is significantly less than the time necessary to run the individual algorithms.

TABLE 1. Comparison of algorithms for the merged LCS and block-merged LCS problem. Here n, m and r denote the lengths of T, A and B respectively. L or L' denotes the length of the answer and $\gamma = \max\{\alpha, \beta\}$, where α and β denotes the number of blocks in A and B respectively; $\mathcal{P}$ and $\mathcal{R}$ retain the same meaning described in Section II. Time and space complexity of Algorithm 3 is not shown here as it will equal Huang's [21] or our algorithm depending on the number of matches.

Merged LCS		
Algorithm	Type	Complexity
Huang's [20]	Time	$O(nmr)$
	Space	$O(nmr)$
Huang's [21]	Time	$O(Lnr)$
	Space	$O(Lmr)$
Our Algorithm	Time	$O((\mathcal{R}r + \mathcal{P}m) \log \log r)$
	Space	$\theta(\max\{nm, r\})$

Block Merged LCS		
Algorithm	Type	Complexity
Huang's [20]	Time	$O(n^2 + n\gamma^2)$
	Space	$O(n^2 + n\gamma^2)$
Huang's [21]	Time	$O(L'n\gamma)$
	Space	$O(L'm\gamma)$
Our Algorithm	Time	$O(\max\{\mathcal{R}\beta \log \log r,$ $\mathcal{P}\alpha \log \log r\})$
	Space	$\theta(\max\{(\mathcal{R} + \mathcal{P}), r\})$

VI. Conclusions

In this paper, we have studied the *merged LCS* problem with and without block constraint that is used to find interleaving relationship between sequences. We used a sparse dynamic programming approach to give efficient algorithm for those problems. First we developed an $O((\mathcal{R}r + \mathcal{P}m) \log \log r)$ algorithm for the Merged LCS Problem. Then we adopted that algorithm for the block constraint and gave another algorithm for the *block merged LCS* problem which runs in $O(\max\{\mathcal{R}\beta \log \log r, \mathcal{P}\alpha \log \log r\})$ time. Both of those algorithms work very efficiently when $\mathcal{R} \sim n$ and $\mathcal{P} \sim n$ but in the worst case when $\mathcal{R}$ is $O(nm)$ and $\mathcal{P}$ is $O(nr)$, the algorithms work slightly worse due to the $\log \log(r)$ term. To overcome this problem, we finally proposed a hybrid-algorithm which gives better output in every cases.

For future study, it would be interesting to see whether the approaches we have taken here to solve the MLCS and BMLCS problem can be employed to other variants of LCS problem to devise efficient algorithms. Also it might be interesting to see if more practically motivated variants of the MLCS or BMLCS problem can be introduced using the concepts of merging sequences and block constraint. Additionally for protein sequences, considering the effect of gaps in sequences, merged LCS problem can be extended to the merged-alignment problem. Furthermore we can extend the problem to the multiple merged LCS or multiple merged alignment problems for merging several sequences.

APPENDIX

TABLE 2. A total of 35 merged sequences and their corresponding MLCS results for the input sequences $A = actt$, $B = ctg$ and $T = tcatcg$

$M(A, B)$	T	$MLCS(A, B, T)$
$acttctg$	$tcatcg$	4
$actcttg$	$tcatcg$	4
$actcttg$	$tcatcg$	4
$actctgt$	$tcatcg$	4
$acctttg$	$tcatcg$	3
$acctttg$	$tcatcg$	3
$accttgt$	$tcatcg$	3
$acctttg$	$tcatcg$	3
$accttgt$	$tcatcg$	3
$acctgtt$	$tcatcg$	3
$acctttg$	$tcatcg$	3
$acctttg$	$tcatcg$	3
$accttgt$	$tcatcg$	3
$acctttg$	$tcatcg$	3
$accttgt$	$tcatcg$	3
$acctgtt$	$tcatcg$	3
$actcttg$	$tcatcg$	4
$actctgt$	$tcatcg$	4
$actcgtt$	$tcatcg$	4
$actgctt$	$tcatcg$	3
$cactttg$	$tcatcg$	4
$cactttg$	$tcatcg$	4
$cacttgt$	$tcatcg$	4
$cactttg$	$tcatcg$	4
$cacttgt$	$tcatcg$	4
$cactgtt$	$tcatcg$	4
$catcttg$	$tcatcg$	5
$catctgt$	$tcatcg$	5
$catcgtt$	$tcatcg$	5
$catgctt$	$tcatcg$	4
$ctacttg$	$tcatcg$	4
$ctactgt$	$tcatcg$	4
$ctacgtt$	$tcatcg$	4
$ctagctt$	$tcatcg$	3
$ctgactt$	$tcatcg$	3

REFERENCES

[1] M. S. Rahman and C. S. Iliopoulos, "Algorithms for computing variants of the longest common subsequence problem," in *ISAAC* (T. Asano, ed.), vol. 4288 of *Lecture Notes in Computer Science*, pp. 399–408, Springer, 2006.

[2] R. Wagner and M. Fischer, "The string-to-string correction problem," *Journal of the ACM (JACM)*, vol. 21, no. 1, pp. 168–173, 1974.

[3] D. S. Hirschberg., "Algorithms for the longest common subsequence problem," *J. ACM*, vol. 24, no. 4, pp. 664–675, 1977.

[4] C.-B. Yang and R. C. T. Lee, "Systolic algorithms for the longest common subsequence problem," *Journal of the Chinese Institute of Engineers*, vol. 10, no. 6, pp. 691–699, 1987.

[5] B. Baker and R. Giancarlo, "Sparse Dynamic Programming for longest common subsequence from fragments," *Journal of algorithms*, vol. 42, no. 2, pp. 231–254, 2002.

[6] T. G. S. J. W. Hunt, "A fast algorithm for computing longest subsequences," *Communication of ACM*, vol. 20, no. 5, pp. 350–353, 1977.

[7] 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.

[8] A. Unay and M. Guzey, "A swarm intelligence heuristic approach to longest common subsequence problem for arbitrary number of sequences," *Metabolomics*, vol. 3, no. 120, pp. 2153–0769, 2013.

[9] W. Masek and M. Paterson, "A faster algorithm computing string edit distances," *Journal of Computer and System sciences*, vol. 20, no. 1, pp. 18–31, 1980.

[10] V. Arlazarov, E. Dinic, M. Kronrod, and I. Faradzev, "On economic construction of the transitive closure of a directed graph (in russian). english translation in soviet math. dokl. 11, 1209-1210, 1975," in *Dokl. Acad. Nauk*, vol. 194, pp. 487–488, 1970.

[11] A. N. Arslan and Ömer Eğecioğlu, "Algorithms for the constrained longest common subsequence problems," *International Journal of Foundations of Computer Science*, vol. 16, no. 6, pp. 1099–1109, 2005.

[12] K. Huang, C. Yang, K. Tseng, Y. Peng, and H. Ann, "Dynamic programming algorithms for the mosaic longest common subsequence problem," *Information Processing Letters*, vol. 102, no. 2-3, pp. 99–103, 2007.

[13] C. S. Iliopoulos and M. S. Rahman, "Algorithms for computing variants of the longest common subsequence problem," *Theor. Comput. Sci.*, vol. 395, no. 2-3, pp. 255–267, 2008.

[14] W. Yingjie, L. Wang, D. Zhu, and X. Wang, "An efficient dynamic programming algorithm for the generalized lcs problem with multiple substring exclusive constraints," *Journal of Discrete Algorithms*, 2014.

[15] A. Gorbenko and V. Popov, "The longest common parameterized subsequence problem," *Applied Mathematical Sciences*, vol. 6, no. 58, pp. 2851–2855, 2012.

[16] S. R. Chowdhury, M. M. Hasan, S. Iqbal, and M. S. Rahman, "Computing a longest common palindromic subsequence," in *Combinatorial Algorithms*, pp. 219–223, Springer, 2012.

[17] 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.

[18] 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.

[19] J. M. Moosa, M. S. Rahman, and F. T. Zohora, "Computing a longest common subsequence that is almost increasing on sequences having no repeated elements," *J. Discrete Algorithms*, vol. 20, pp. 12–20, 2013.

[20] K.-S. Huang, C.-B. Yang, K.-T. Tseng, H.-Y. Ann, and Y.-H. Peng, "Efficient algorithms for finding interleaving relationship between sequences," *Inf. Process. Lett.*, vol. 105, no. 5, pp. 188–193, 2008.

[21] Y.-H. Peng, C.-B. Yang, K.-S. Huang, C.-T. Tseng, and C.-Y. Hor, "Efficient sparse dynamic programming for the merged lcs problem with block constraints," *International Journal of Innovative Computing, Information and Control*, vol. 6, April 2010.

[22] O. Jaillon, J. Aury, F. Brunet, J. Petit, N. Stange-Thomann, E. Mauceli, L. Bouneau, C. Fischer, C. Ozouf-Costaz, A. Bernot, S. Nicaud, D. Jaffe,

S. Fisher, G. Lutfalla, C. Dossat, B. Segurens, C. Dasilva, M. Salanoubat, M. Levy, N. Boudet, S. Castellano, V. Anthouard, C. Jubin, V. Castelli, M. Katinka, B. Vacherie, C. Biemont, Z. Skalli, L. Cattolico, J. Poulain, V. de Berardinis, C. Cruaud, S. Duprat, P. Brottier, J. Coutanceau, J. Gouzy, G. Parra, G. Lardier, C. Chapple, K. McKernan, P. McEwan, S. Bosak, M. Kellis, J. Volff, R. Guigo, M. Zody, J. Mesirov, K. Lindblad-Toh, B. Birren, C. Nusbaum, D. Kahn, M. Robinson-Rechavi, V. Laudet, V. Schachter, F. Quetier, W. Saurin, C. Scarpelli, P. Wincker, E. Lander, J. Weissenbach, and H. Roest Crollius, "Genome duplication in the teleost fish Tetraodon nigroviridis reveals the early vertebrate proto-karyotype," *Nature*, vol. 431, no. 7011, pp. 946–957, 2004.

[23] M. Kellis, B. Birren, and E. Lander, "Proof and evolutionary analysis of ancient genome duplication in the yeast Saccharomyces cerevisiae," *Nature*, vol. 428, no. 6983, pp. 617–624, 2004.

[24] D. Vallenet, L. Labarre, Z. Rouy, V. Barbe, S. Bocs, S. Cruveiller, A. Lajus, G. Pascal, C. Scarpelli, and C. Médigue, "MaGe: a microbial genome annotation system supported by synteny results," *Nucleic acids research*, vol. 34, no. 1, p. 53, 2006.

[25] G. M. Landau, B. Schieber, and M. Ziv-Ukelson, "Sparse lcs common substring alignment," in *CPM* (R. A. Baeza-Yates, E. Chávez, and M. Crochemore, eds.), vol. 2676 of *Lecture Notes in Computer Science*, pp. 225–236, Springer, 2003.

[26] G. Landau and M. Ziv-Ukelson, "On the Common Substring Alignment Problem," *Journal of Algorithms*, vol. 41, no. 2, pp. 338–359, 2001.

[27] G. S. Brodal, K. Kaligosi, I. Katriel, and M. Kutz, "Faster algorithms for computing longest common increasing subsequences," in *CPM* (M. Lewenstein and G. Valiente, eds.), vol. 4009 of *Lecture notes in Computer Science*, pp. 330–341, Springer, 2006.

AHM Mahfuzur Rahman received his B.Sc. Engg. degree in Computer Science and Engineering (CSE) from Bangladesh University of Engineering and Technology (BUET) in 2011. He is currently working as a senior software engineer in Samsung Research Institute, Bangladesh. His research interests are in Bioinformatics and Computational Biology, Computer Vision and Artificial Intelligence.

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, Kings 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,