Algorithms for Two Versions of LCS Problem for Indeterminate Strings*

Costas S. Iliopoulos[1,3,**], M. Sohel Rahman[1,3,***,†], and Wojciech Rytter[2,4,‡]

[1] Department of Computer Science, King's College London,
Strand, London WC2R 2LS, England
http://www.dcs.kcl.ac.uk/adg
[2] Institute of Informatics, Warsaw University, Warsaw, Poland,
[3] {sohel,csi}@dcs.kcl.ac.uk
[4] rytter@mimuw.edu.pl

Abstract. We study the complexity of the longest common subsequence (LCS) problem from a new perspective. By an indeterminate string (i-string, in short) we mean a sequence $\widetilde{X} = \widetilde{X}[1]\widetilde{X}[2]\ldots\widetilde{X}[n]$, where $\widetilde{X}[i] \subseteq \Sigma$ for each i, and Σ is a given alphabet of potentially large size. A subsequence of $\widetilde{X}$ is any usual string over Σ which is an element of the finite (but usually of exponential size) language $\widetilde{X}[i_1]\widetilde{X}[i_2]\ldots\widetilde{X}[i_p]$, where $1 \leq i_1 < i_2 < i_3 \ldots < i_p \leq n, p \geq 0$. Similarly, we define a supersequence of x. Our first version of the LCS problem is Problem ILCS: for given i-strings $\widetilde{X}$ and $\widetilde{Y}$, find their longest common subsequence. From the complexity point of view, new parameters of the input correspond to $|\Sigma|$ and maximum size ℓ of the subsets in $\widetilde{X}$ and $\widetilde{Y}$. There is also a third parameter $\mathcal{R}$, which gives a measure of similarity between $\widetilde{X}$ and $\widetilde{Y}$. The smaller the R, the lesser is the time for solving Problem ILCS. Our second version of the LCS problem is Problem CILCS (constrained ILCS): for given i-strings $\widetilde{X}$ and $\widetilde{Y}$ and a plain string Z, find the longest common subsequence of $\widetilde{X}$ and $\widetilde{Y}$ which is, at the same time, a supersequence of Z. In this paper, we present several efficient algorithms to solve both ILCS and CILCS problems. The efficiency in our algorithms are obtained in particular by using an efficient data structure for special type of range maxima queries and fast multiplication of boolean matrices.

1 Introduction

This paper deals with the classical and well-studied longest common subsequence (LCS) problem and two its variants: LCS for indeterminate strings and Con-

* Part of this research work was carried out when Costas Iliopoulos and M. Sohel Rahman were visiting Institute of Informatics, Warsaw University.
** Supported by EPSRC and Royal Society grants.
*** Supported by the Commonwealth Scholarship Commission in the UK under the Commonwealth Scholarship and Fellowship Plan (CSFP).
† On Leave from Department of CSE, BUET, Dhaka-1000, Bangladesh.
‡ Supported by the grant of the Polish Ministery of Science and Higher Education N 206 004 32/0806.

strained LCS (CLCS) problem, also for indeterminate strings. Given two strings, the LCS problem consists in computing a subsequence of maximum length common to both strings. In CLCS, the computed longest common subsequence must also be a supersequence of a third given string. The classic dynamic programming solution to the LCS problem, invented by Wagner and Fischer [16], has $O(n^2)$ worst case running time, where n is the length of the two strings. For a comprehensive comparison of the well-known algorithms for LCS problem and study of their behaviour in various application environments the readers are referred to [5]. The CLCS problem, on the other hand, has been introduced quite recently by Tsai in [14]. In [14], a dynamic programming formulation for CLCS was presented leading to a $O(pn^4)$ time algorithm to solve the problem, where p is the length of the third string which applies the constraint. Later, Chin et al. [6] and independently, Arslan and Eğecioğlu [2] presented improved CLCS algorithm with $O(nmr)$ time and space complexity. In this paper, we revisit the LCS and CLCS problems, but in a different setting: instead of standard strings, we consider indeterminate strings (i-strings, in short), where at each position the string may contain a set of characters. The motivation of our study comes from the fact that i-strings are extensively used in molecular biology to express polymorphism in DNA sequences, e.g. the polymorphism of protein coding regions caused by redundancy of the genetic code or polymorphism in binding site sequences of a family of genes. We present a number of efficient algorithms to solve the LCS and CLCS problems. We use $LCS(X, Y)$ to denote a longest common subsequence of X and Y. We denote the length of $LCS(X, Y)$ by $\mathcal{L}(X, Y)$. Given two strings $X[1..n]$ and $Y[1..n]$ and a third string $Z[1..p]$, a common subsequence S of X, Y is said to be *constrained* by Z if, and only if, Z is a subsequence of S. We use $LCS_Z(X, Y)$ to denote a longest common subsequence of X and Y that is constrained by Z. We denote the length of $LCS_Z(X, Y)$ by $\mathcal{L}_Z(X, Y)$.

Fig. 1. $|LCS(X, Y)| = 4$ and $|LCS_Z(X, Y)| = 2$.

Example 1. Suppose $X = ACTACA$, $Y = ACCAAG$ and $Z = AC$. As is evident from Fig. 1, $S_1 = CCAA$ is an $LCS(X, Y)$. However, S_1 is not an $LCS_Z(X, Y)$ because Z is not a subsequence of S_1. On the other hand, $S_2 = ACA$ is an $LCS_Z(X, Y)$. Note carefully that, in this case $\mathcal{L}_Z(X, Y) < \mathcal{L}(X, Y)$.

In this paper, we are interested in *indeterminate* strings (*i-strings*, in short). In contrast, usual strings are called here *standard* strings. A string $\widetilde{X}[1..n]$ is said to be indeterminate, if it is built over the potential $2^{|\Sigma|} - 1$ non-empty sets of letters

belonging to Σ. Each $\widetilde{X}[i], 1 \leq i \leq n$ can be thought of as a set of characters and we have $|\widetilde{X}[i]| \geq 1, 1 \leq i \leq n$. The length of the i-string $\widetilde{X}$, denoted by $|\widetilde{X}|$, is the number of sets (of characters) in it, i.e., n. In this paper, the set containing the letters A and C will be denoted by $[AC]$ and the singleton $[C]$ will be simply denoted by C for ease of reading. Also, we use the following convention: we use plain letters like X to denote normal strings. The same letter may be used to denote a i-string if written as $\widetilde{X}$. For i-strings, the notion of symbol equality is extended to single-symbol match between two (indeterminate) letters in the following way. Given two subsets $A, B \subseteq \Sigma$ we say that A matches B and write $A \approx B$ iff $A \cap B \neq \emptyset$. Note that, the relation $\approx$, referred to as the 'indeterminate equality' henceforth, is not transitive.

Example 2.
Suppose we have i-strings $\widetilde{X} = AC[CTG]TG[AC]C$ and $\widetilde{Y} = TC[AT][AT]TTC$. Here we have $\widetilde{X}[3] \approx \widetilde{Y}[3]$ because $\widetilde{X}[3] = [CTG] \cap \widetilde{Y}[3] = [AT] = T \neq \emptyset$. Similarly we have, $\widetilde{X}[3] \approx \widetilde{Y}[1]$, and also $\widetilde{X}[3] \approx \widetilde{Y}[2]$ etc.

We can extend the notion of a subsequence for i-strings in a natural way replacing the equality of symbols by the relation $\approx$ as follows. A subsequence of $\widetilde{X}$ is a plain string U over Σ which is an element of the finite (but usually of exponential size) language $\widetilde{X}[i_1]\widetilde{X}[i_2]\ldots\widetilde{X}[i_p]$, where $1 \leq i_1 < i_2 \ldots < i_p \leq n, p \geq 0$.
Similarly, we define a supersequence of $\widetilde{X}$. The notion of common and longest common subsequence for i-strings can now be extended easily. We are interested in the following two problems.

Problem "ILCS" (LCS for Indeterminate Strings). *Given* 2 *i-strings* $\widetilde{X}$ *and* $\widetilde{Y}$ *we want to compute an* $LCS(\widetilde{X}, \widetilde{Y})$.

Problem "CILCS" (CLCS for Indeterminate Strings). *Given* 2 *i-strings* $\widetilde{X}$ *and* $\widetilde{Y}$ *and another (plain) string* Z*, we want to compute an* $LCS_Z(\widetilde{X}, \widetilde{Y})$.

Example 3. Suppose, we are given the i-strings

$$\widetilde{X} = [AF]BDDAAA,\ \widetilde{Y} = [AC]BA[CD]AA[DF],\ Z = BDD$$

Figure 2 shows an $LCS(\widetilde{X}, \widetilde{Y})$ and an $LCS_Z(\widetilde{X}, \widetilde{Y})$. Note that, although $\mathcal{L}(\widetilde{X}, \widetilde{Y}) = 5$, $\mathcal{L}_Z(\widetilde{X}, \widetilde{Y}) = 4$.

$\widetilde{X}$	$[AF]$	B		D	D	A	A	A		
$\widetilde{Y}$	$[AC]$	B	A	$[CD]$			A	A	$[DF]$	
An $LCS(\widetilde{X}, \widetilde{Y})$	A	B		D			A	A		

$\widetilde{X}$	$[AF]$	B		D			D	A	A	A
$\widetilde{Y}$	$[AC]$	B	A	$[CD]$	A	A	$[DF]$			
Z		B		D			D			
An $CLCS_Z(\widetilde{X}, \widetilde{Y})$	A	B		D			D			

Fig. 2. $LCS(\widetilde{X}, \widetilde{Y})$ and $LCS_Z(\widetilde{X}, \widetilde{Y})$ of Example 3.

In what follows, for the ease of exposition, we assume that $|\widetilde{X}| = |\widetilde{Y}| = n$. But our results can be easily extended when $|\widetilde{X}| \neq |\widetilde{Y}|$.

2 Algorithm for ILCS

In this section, we devise efficient algorithms for Problem ILCS. We start with a brief review of the traditional dynamic programming technique employed to solve LCS [16] for standard strings. Here the idea is to determine the longest common subsequences for all possible prefix combinations of the input strings. The recurrence relation for extending the length of LCS for each prefix pair $(X[1..i], Y[1..j])$, i.e. $\mathcal{L}(X[1..i], Y[1..j])$, is as follows [16]:

$$\mathcal{T}[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ max(\mathcal{T}[i-1,j-1] + \delta(i,j),\ \mathcal{T}[i-1,j],\ \mathcal{T}[i,j-1]), & \text{otherwise,} \end{cases} \tag{1}$$

where $\delta(i,j) = 0$ if $X[i] = Y[j]$; otherwise $\delta(i,j) = 1$.

Here we have used the tabular notion $\mathcal{T}[i,j]$ to denote $\mathcal{L}(X[1..i], Y[1..j])$. After the table has been filled, $\mathcal{L}(X,Y)$ can be found in $\mathcal{T}[n,n]$ and $LCS(X,Y)$ can be found by backtracking from $\mathcal{T}[n,n]$ (for detail please refer to [16] or any textbook on algorithms, e.g. [8]). It is easy to see that Equation 1 can be realized easily in $O(n^2)$ time.

Interestingly, Equation 1 can be adapted to solve Problem ILCS quite easily. The only thing we need to do is to replace the equality check ('=') in Equation 1 with the indeterminate equality ('$\approx$'). However, this 'simple' change affects the running time of the algorithm because instead of the constant time equality check we need to perform intersection between two sets. To deduce the precise running time of the resulting algorithm, we make the assumption that the sets of characters for each input i-strings are given in sorted order. Under this assumption, we can perform the intersection operation in $O(|\Sigma|)$ time in the worst case, since there can be at most $|\Sigma|$ characters in a set of a i-string. So we have the following theorem.

Theorem 1. *Problem ILCS can be solved in $O(|\Sigma|n^2)$ time.*

In the rest of this section, we try to devise algorithms giving better running times than what is reported in Theorem 1. We assume that the alphabet Σ is indexed, which is the case in most practical situations. We also assume that the sets of characters for each input i-strings are given in sorted order. Recall that the latter assumption is required to get the running time of Theorem 1. To improve the running time, we plan to do some preprocessing to realize Equation 1 more efficiently. In particular, we want to preprocess the two given strings so that the indeterminate equality check can be realized in $O(1)$ time. In the next subsections, we present three different preprocessing steps and analyze the time and space complexity of the resulting algorithms.

2.1 Preprocessing 1 for ILCS

Here the idea is to first compute a table $\mathcal{I}[i,j], 1 \leq i,j \leq n$ as defined below:

$$\mathcal{I}[i,j] = \begin{cases} 1 & \text{If } \widetilde{X}[i] \bigcap \widetilde{Y}[j] \neq \emptyset \\ 0 & \text{Otherwise.} \end{cases} \tag{2}$$

It is easy to realize that, with that table $\mathcal{I}$ in our hand, we can realize Equation 1 in $O(n^2)$ time because the indeterminate equality check reduces to the constant time checking of the corresponding entry in $\mathcal{I}$-table. Now it remains to see how efficiently we can compute the table $\mathcal{I}$. Recall that, our ultimate goal is to get a overall running time better than the one reported in Theorem 1. To compute the table $\mathcal{I}$, we first encode each $\widetilde{X}[i], 1 \leq i \leq n$ and $\widetilde{Y}[j], 1 \leq j \leq n$ as a binary vector of size $|\Sigma|$ as follows. We use $\widetilde{X}_e$ and $\widetilde{Y}_e$ to denote the encodings for $\widetilde{X}$ and $\widetilde{Y}$, respectively. For all $1 \leq i \leq n$ and $c \in |\Sigma|$, the encoding for $\widetilde{X}[i]$ is defined below:

$$\widetilde{X}_e[i][c] = \begin{cases} 1 & \text{If } c \in \widetilde{X}[i] \\ 0 & \text{Otherwise.} \end{cases} \tag{3}$$

The encoding for $\widetilde{Y}$ is defined analogously. Now, we can view $\widetilde{X}_e$ and $\widetilde{Y}_e$ as two ordered lists having n binary vectors each, where each vector is of size $|\Sigma|$. And it is easy to realize that the computation of $\mathcal{I}$ reduces to the matrix multiplication of $\widetilde{X}_e$ and $\widetilde{Y}_e$. To speed-up this computation, we perform the following trick. Without loss of generality, we assume that n is divisible by $|\Sigma|$ We divide both the boolean matrices $\widetilde{X}_e$ and $\widetilde{Y}_e$ in square partitions, each partition having size $|\Sigma| \times |\Sigma|$. Now we can perform the matrix multiplication by performing square matrix multiplication of the constituent square blocks.

Next, we analyze the running time of the preprocessing discussed above. The encoding of the two input i-strings $\widetilde{X}$ and $\widetilde{Y}$ require $O(n|\Sigma|)$ time and space. For square matrix multiplication, the best known algorithm is due to Coppersmith and Winograd [7]. Their algorithm works in $O(\mathcal{N}^{2.376})$ time, where the involved matrices are of size $\mathcal{N} \times \mathcal{N}$. Now, recall that in our case the square matrices are of size $|\Sigma| \times |\Sigma|$. Also, it is easy to see that, in total, we need $(n/|\Sigma|)^2$ such computation. Therefore, the worst case computational effort required is $O((n/|\Sigma|)^2 \times |\Sigma|^{2.376}) = O(n^2|\Sigma|^{0.376})$. To sum up, the total time required to solve Problem ILCS is $O(n|\Sigma| + n^2|\Sigma|^{0.376}) + n^2) = O(n|\Sigma| + n^2|\Sigma|^{0.376}))$ in the worst case. This implies the following result.

Theorem 2. *Problem ILCS can be solved in $O(n|\Sigma| + n^2|\Sigma|^{0.376})$ time.*

Before concluding this section, we briefly review some other boolean matrix multiplication results that may be used in our algorithm. There is a simple so called "Four Russians" algorithm of Arlazarov et al. [1], which performs Boolean $\mathcal{N} \times \mathcal{N}$ matrix multiplication in $O(\mathcal{N}^3/\log \mathcal{N})$ time. This was eventually improved slightly to $O(\mathcal{N}^3/\log^{1.5} \mathcal{N})$ time by Atkinson and Santoro [3]. Rytter [13] and independently Basch, Khanna, and Motwani [4] gave an $O(\mathcal{N}^3/\log^2 \mathcal{N})$ algorithm for Boolean matrix multiplication on the $(\log n)$-word RAM. Similar result also follows from a very recent paper [17]. Therefore problem ILCS can be solved in $O(n|\Sigma| + n^2|\Sigma|/\log^2 |\Sigma|))$ time without algebraically sophisticated matrix multiplication.

2.2 Preprocessing 2 for ILCS

We first present some notations needed to discuss the next preprocessing phase. We define the term ℓ as follows:

$$\ell = \max\{|\widetilde{X}[i]|, |\widetilde{Y}[i]| \mid 1 \leq i \leq n\}.$$

Also, we say a pair $(i, j), 1 \leq i, j \leq n$ defines a match, if $\widetilde{X}[i] \approx \widetilde{Y}[j]$. The set of all matches, $\mathcal{M}$, is defined as follows:

$$\mathcal{M} = \{(i, j) \mid \widetilde{X}[i] \approx \widetilde{Y}[j], 1 \leq i, j \leq n\}.$$

We assume that $\mathcal{R} = |\mathcal{M}|$.

Algorithm 1 Computation of the Table $\mathcal{I}$

```
1: for each i, j do  I[i, j] = 0;
2: for each a ∈ Σ do
3:     L_X̃[a] := ∅; L_Ỹ[a] := ∅;
4: for i = 1 to n do
5:     for each a ∈ X̃[i] do   insert(i, L_X̃[a])
6:     for each b ∈ Ỹ[i] do   insert(i, L_Ỹ[b])
7: for each a ∈ Σ do
8:     for each i ∈ L_X̃[a] do
9:         for each j ∈ L_Ỹ[a] do   I[i, j] = 1.
```

In the algorithm we pre-compute $\mathcal{M}$, and then fill up the table $\mathcal{I}$. We proceed as follows. We construct, for each symbol $a \in \Sigma$, two separate lists, $L_{\widetilde{X}}[a]$ and $L_{\widetilde{Y}}[a]$. For each $a \in \Sigma$, $L_{\widetilde{X}}[a]$ ($L_{\widetilde{Y}}[a]$) stores the positions where a appears in $\widetilde{X}$ ($\widetilde{Y}$), if any. We have for $1 \leq i, j \leq n$

$$\mathcal{I}[i, j] \;=\; 1 \;\Leftrightarrow\; \exists\ (a \in \Sigma) \text{ such that } (i \in L_{\widetilde{X}}[a]) \text{ and } (j \in L_{\widetilde{Y}}[a])$$

The initialization of the table $\mathcal{I}$ requires $O(n^2)$ time. The constructions of the lists $L_{\widetilde{X}}[a], L_{\widetilde{Y}}[a], a \in \Sigma$ can be done in $(n\ell)$ time simply by scanning $\widetilde{X}$ and $\widetilde{Y}$ in turn. Traversing the two lists $L_X[a]$ and $L_Y[a]$ for each $a \in \Sigma$ to fill up $\mathcal{I}$ requires $O(\mathcal{R}\ell)$ time. This follows from the fact that there are in total $\mathcal{R}$ positions where we can have a match and at each such position we can have up to ℓ matches. Thus the total running time required for the preprocessing is $O(n\ell + n^2 + \mathcal{R}\ell)$.

Theorem 3. *Problem ILCS can be solved in $O(n\ell + n^2 + \mathcal{R}\ell)$ time.*

We remark that, in the worst case we have $\ell = |\Sigma|$. However, ℓ can be much smaller than $|\Sigma|$ in many cases. Also, $\mathcal{R}\ell$ and $n\ell$ are 'pessimistic' upper bounds in the sense that rarely for all $1 \leq i \leq n$, we will have $|\widetilde{X}[i]| = \ell$ ($|\widetilde{Y}[i]| = \ell$). Also, in the worst case we have $\mathcal{R} = O(n^2)$. But in many practical cases $\mathcal{R}$ turns out to be $o(n^2)$, or even $O(n)$. Another remark is that to compute an actual LCS, we will additionally require $O(n\ell)$ time in the worst case.

2.3 Preprocessing 3 for ILCS

The preprocessing of Section 2.2 can be slightly modified to devise an efficient algorithm based on the parameter $\mathcal{R}$. There exist efficient algorithms for computing LCS depending on parameter $\mathcal{R}$. The recent work of Rahman and Iliopoulos [12] presents an $O(\mathcal{R}\log\log n+n)$ algorithm (referred to as LCS-II in [12]) for computing the LCS. LCS-II computes the set $\mathcal{M}$, sorts it in a 'prescribed' order and then considers each $(i,j) \in \mathcal{M}$ and do some useful computation (instead of performing computation for all n^2 entries in the usual dynamic programming matrix). The efficient computation in LCS-II is based on the use of the famous vEB data structure invented by van Emde Boas [15] to solve a restricted dynamic version of the Range Maxima Query problem. The vEB data structure allows us to maintain a sorted list of integers in the range $[1..n]$ in $O(\log\log n)$ time per insertion and deletion. In addition to that it can return $next(i)$ (successor element of i in the list) and $prev(i)$ (predecessor element of i in the list) in constant time. On the other hand, the range maxima query (RMQ) problem is to preprocess an array of numbers to answer queries to find the maximum in a given range of the array. The use of RMQ in solving LCS problems and variants thereof was explored in [10] and later in [9, 11, 12]. In [12], the authors observed that to compute the LCS, one needs only solve a restricted but dynamic version of RMQ problem[5]. Using the vEB structure, they then solved this restricted RMQ problem in $O(\mathcal{R}\log\log n)$ time per update and per query, which leads to an overall $O(\mathcal{R}\log\log n + n)$ time algorithm for computing the LCS. Now the essential thing about LCS-II is that if $\mathcal{M}$ is computed and supplied to it, it can compute the LCS in $O(\mathcal{R}\log\log n)$ time. Based on the results of [12], we get the following theorem.

Theorem 4. *Problem ILCS can be solved in $O(\mathcal{R}\ell + \mathcal{R}\log\log n + n)$ time.*

Proof. (Sketch)
We can slightly change Algorithm 1 to compute the set $\mathcal{M}$ instead of computing the table $\mathcal{I}$. This can be done simply by replacing Step 13 of Algorithm 1 with the following statement:

$\mathcal{M} = \mathcal{M} \bigcup (i,j)$.

We also need to initialize $\mathcal{M}$ to $\emptyset$ just before the for loop of Step 9.

Now, for plain strings, $\mathcal{M}$ can be computed in $O(\mathcal{R})$ time. But, for i-strings, it requires $O(\mathcal{R}\ell)$ time, because at each match position we may have up to ℓ matches in the worst case. However, recall that our goal is to use LCS-II for which we need $\mathcal{M}$ to be in a prescribed order. This however, can be done using the same preprocessing algorithm (Algorithm Pre) used in [12]. Algorithm Pre computes $\mathcal{M}$ requiring $O(\mathcal{R}+n)$ time (for normal strings) and using the vEB structures maintain the prescribed order spending $O(\mathcal{R}\log\log n)$ time. In our case, we have already computed $\mathcal{M}$ for the degenerate strings, and hence, we use Algorithm Pre, only to maintain the orders. Therefore, in total our preprocessing

[5] Similar ideas were also utilized in [10] to solve LCS although employing a slightly different strategy and using a different data structure.

requires $O(\mathcal{R}\ell + \mathcal{R}\log\log n)$ time. Finally, once $\mathcal{M}$ (for the degenerate strings) is computed in the prescribed order, we can employ LCS-II directly to solve Probelem ILCS, requiring a further $O(\mathcal{R}\log\log n)$ time. Therefore, in total, the running time to solve Problem ILCS remains $O(\mathcal{R}\ell+\mathcal{R}\log\log n+n)$ in the worst case. □

Remark 1. LCS-II can compute the actual LCS in $O(\mathcal{L}(X,Y))$ time. However, in our adaptation of that algorithm for i-strings, we will need $O(\mathcal{L}(X,Y)\times\ell)$ time because we don't keep track of the matched character and therefore, are required to do the intersection operations to find a match. However, this can be reduced to $O(\mathcal{L}(X,Y))$ simply by keeping track of the matched character (at least one of them if there exists more) in the set $\mathcal{M}$.

3 Algorithm for CILCS

In this section, we present algorithms to solve Problem CILCS, i.e. Constrained LCS problem for i-strings. We follow the same strategy of Section 2: we use the best known dynamic programming algorithm for CLCS and try to devise an efficient algorithm for CILCS by doing some preprocessing. We use the dynamic programming formulation for CLCS presented in [2]. Extending our tabular notion from Equation 1, we use $\mathcal{T}[i,j,k], 1 \le i,j \le n, 0 \le k \le p$ to denote $\mathcal{L}_{Z[1..k]}(X[1..i], Y[1..j])$. We have the following formulation for Problem CLCS from [2].

$$\mathcal{T}[i,j,k] = \max\{\mathcal{T}'[i,j,k], \mathcal{T}''[i,j,k], \mathcal{T}[i,j-1,k], \mathcal{T}[i-1,j,k]\} \tag{4}$$

where

$$\mathcal{T}'[i,j,k] = \begin{cases} 1+\mathcal{T}[i-1,j-1,k-1] & \text{if}(k=1 \text{ or} \\ & (k>1 \text{ and } \mathcal{T}[i-1,j-1,k-1]>0)) \\ & \text{and } X[i]=Y[j]=Z[k]. \\ 0 & \text{otherwise.} \end{cases} \tag{5}$$

and

$$\mathcal{T}''[i,j,k] = \begin{cases} 1+\mathcal{T}[i-1,j-1,k] & \text{if } (k=0 \text{ or } \mathcal{T}[i-1,j-1,k]>0) \\ & \text{and } X[i]=Y[j]. \\ 0 & \text{otherwise.} \end{cases} \tag{6}$$

The following boundary conditions are assumed in Equations 4 to 6:

$$\mathcal{T}[i,0,k] = \mathcal{T}[0,j,k] = 0,\ 0 \le i,j \le n, 0 \le k \le p.$$

It is straightforward to give an $O(n^2p)$ algorithm realizing the dynamic programming formulation presented in Equations 4 to 6. Now, for CILCS, we

have to make the following changes. First of all, all equality checks of the form $X[i] = Y[j]$ have to be replaced by:

$$\widetilde{X}[i] \approx \widetilde{Y}[j]. \tag{7}$$

Here, the constant time operation is replaced by an $O(|\Sigma|)$ time operation in the worst case. On the other hand, all the triple equality checks of the form $X[i] = Y[j] = Z[k]$ have to be replaced by the check:

$$Z[k] \in \widetilde{X}[i] \bigcap \widetilde{Y}[j]. \tag{8}$$

Once again, the constant time operations are replaced by $O(|\Sigma| + \log|\Sigma|)$ time operations. So we have the following theorem.

Theorem 5. *Problem CILCS can be solved in $O(|\Sigma|n^2p)$ time.*

As before, our goal is to do some preprocessing to facilitate $O(1)$ time realization of the Check 7 and 8 and thereby improve the running time reported in Theorem 5.

3.1 Preprocessing 1 for CILCS

It is clear that we need the table $\mathcal{I}$ as defined in Section 2.1 for constant time realization of Check 7. In addition to that, to realize Check 8 in constant time, we compute two more tables $\mathcal{B}_{\widetilde{X}}[i,k]$, for $1 \leq i \leq n, 1 \leq k \leq p$ and $\mathcal{B}_{\widetilde{Y}}[j,k], 1 \leq j \leq n, 1 \leq k \leq p$, as defined below:

$$\mathcal{B}_{\widetilde{X}}[i,k] = \begin{cases} 1 & \text{If } Z[k] \in \widetilde{X}[j] \\ 0 & \text{Otherwise.} \end{cases} \tag{9}$$

$$\mathcal{B}_{\widetilde{Y}}[j,k] = \begin{cases} 1 & \text{If } Z[k] \in \widetilde{Y}[j] \\ 0 & \text{Otherwise.} \end{cases} \tag{10}$$

It is easy to realize that Check 8 evaluates to be true if, and only if, we have $\mathcal{B}_{\widetilde{X}}[i,k] = \mathcal{B}_{\widetilde{Y}}[j,k] = \mathcal{I}[i,j] = 1$. Therefore, with all three tables pre-computed, we can evaluate Check 8 in constant time. We have already discussed the construction of table $\mathcal{I}$ in Section 2.1. The other two tables can be computed exactly in the same way requiring $O(np|\Sigma|^{0.376})$ time each. Note that $p \leq n$. So we have:

Theorem 6. *Problem CILCS can be solved in $O(n|\Sigma| + n^2|\Sigma|^{0.376} + n^2p)$ time.*

Theorem 7. *Problem CILCS can be solved in $O(n|\Sigma| + n^2|\Sigma|/\log^2|\Sigma|) + n^2p)$ time.*

However, instead of applying the complex matrix multiplication algorithm to compute $\mathcal{B}_{\widetilde{X}}$ and $\mathcal{B}_{\widetilde{Y}}$ we can do it more simply by employing the set-membership check for each entry in the $n \times p$ table. This would require $O(np\log|\Sigma|)$ time to compute $\mathcal{B}_{\widetilde{X}}$ and $\mathcal{B}_{\widetilde{Y}}$. However, the overall asymptotic running time remains unimproved.

3.2 Preprocessing 2 for CILCS

In this section, we adapt the preprocessing of Section 2.2 to solve Problem CILCS. We first define the set of all 'triple' matches, $\mathcal{M}_3$, as follows:

$$\mathcal{M}_3 = \{(i,j,k) \mid \widetilde{X}[i] \approx \widetilde{Y}[j] \bigwedge Z[k] \in \widetilde{X}[i] \bigcap \widetilde{Y}[j], 1 \leq i,j \leq n, 1 \leq k \leq p\}.$$

We assume that $\mathcal{R}_3 = |\mathcal{M}_3|$. Now our goal is to compute the table $\mathcal{I}_3[i,j,k], 1 \leq i,j \leq n, 1 \leq k \leq p$ as defined below:

$$\mathcal{I}_3[i,j,k] = \begin{cases} 1 & \text{If } Z[k] \in \widetilde{X}[i] \bigcap \widetilde{Y}[j] \\ 0 & \text{Otherwise.} \end{cases} \tag{11}$$

It is easy to realize that, with table $\mathcal{I}$ and $\mathcal{I}_3$, we can evaluate, respectively, Check 7 and 8 in constant time each. And it is quite straightforward to adapt Algorithm 1 to compute $\mathcal{I}_3$ (please see Algorithm 2). All we need to do is to construct lists $L_Z[a]$, (similar to $L_{\widetilde{X}}[a]$ and $L_{\widetilde{Y}}[a]$) for each symbol $a \in \Sigma$ and incorporate it in the for loop of Step 9. The preprocessing time to compute $\mathcal{I}_3$ can be easily deduced following the analysis of Algorithm 1. To compute $\mathcal{I}_3$ we need to create $3 * |\Sigma|$ lists. These can be constructed by simply scanning $\widetilde{X}$, $\widetilde{Y}$ and Z in turn requiring in total $O(2 \times n\ell + n) = O(n\ell)$ time in the worst case. The initialization of $\mathcal{I}_3$ requires $O(n^2p)$ time. The filling up of $\mathcal{I}_3$ requires $O(\mathcal{R}_3\ell)$ time. Note that we also need to compute $\mathcal{I}$. Thus the total running time required for the preprocessing is $O(n\ell + n^2 + n^2p + \mathcal{R}\ell + \mathcal{R}_3\ell) = O(n\ell + n^2p + \ell(\mathcal{R} + \mathcal{R}_3))$. Since, we already have an n^2p component in the above running time, the total running time for the CILCS problem remains same as above.

Algorithm 2 Computation of the Table $\mathcal{I}_3$

```
1: for a ∈ Σ do
2:     Insert the positions of a in X̃ in L_X̃[a] in sorted order
3:     Insert the positions of a in Ỹ in L_Ỹ[a] in sorted order
4:     Insert the positions of a in Z in L_Z[a] in sorted order
5: for i = 1 to n do
6:     for j = 1 to n do
7:         for k = 1 to p do
8:             I[i, j, k] = 0.
9: for a ∈ Σ do
10:    for i ∈ L_X̃[a] do
11:        for j ∈ L_Ỹ[a] do
12:            for k ∈ L_Z[a] do
13:                I[i, j, k] = 1.
```

Theorem 8. *Problem CILCS can be solved in $O(n\ell + n^2p + \ell(\mathcal{R} + \mathcal{R}_3))$ time.*

We remind that the remarks in Section 2.2, regarding ℓ and $\mathcal{R}$, applies here as well. We further remark that, in the worst case we have $\mathcal{R}_3 = n^2p$. But in many practical cases $\mathcal{R}_3$ may turn out to be $o(n^2p)$. Also, since ℓ can be much smaller than $|\Sigma|$ in many cases, $\mathcal{R}_3\ell$ remains as a 'pessimistic' upper bound.

4 Conclusion

In this paper, we have studied the classic and well-studied longest common subsequence (LCS) problem and a recent variant of it namely the constrained LCS (CLCS) problem, when the inputs are indeterminate strings. In LCS, given two strings, we want to find the common subsequence having the highest length; in CLCS, in addition to that, the solution to the problem must also be a supersequence of a third given string. We have presented efficient algorithms to solve both LCS and CLCS for indeterminate strings. In particular, we have used some novel techniques to preprocess the given strings, which lets us use the corresponding DP solutions for normal string to get efficient solution for indeterminate strings. It would be interesting to see how well the presented algorithms behave in practice and compare them among themselves on the basis of their practical performance.

Acknowledgement

We would like to express our gratitude to the anonymous reviewers for their helpful comments and especially to "Reviewer 2" for pointing out that the similar techniques used in [9, 12] to solve LCS have also been used in [10] to get efficient algorithms.

References

1. V. Arlazarov, E. Dinic, M. Kronrod, and I. Faradzev. On economic construction of the transitive closure of a directed graph (english translation). *Soviet Math. Dokl.*, 11:1209–1210, 1975.
2. A. N. Arslan and Ö. Eğecioğlu. Algorithms for the constrained longest common subsequence problems. *Int. J. Found. Comput. Sci.*, 16(6):1099–1109, 2005.
3. M. D. Atkinson and N. Santoro. A practical algorithm for boolean matrix multiplication. *Inf. Process. Lett.*, 29(1):37–38, 1988.
4. J. Basch, S. Khanna, and R. Motwani. On diameter verification and boolean matrix multiplication. *Technical Report, Department of Computer Science, Stanford University*, (STAN-CS-95-1544), 1995.
5. L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *String Processing and Information Retrieval (SPIRE)*, pages 39–48. IEEE Computer Society, 2000.
6. F. Y. L. Chin, A. D. Santis, A. L. Ferrara, N. L. Ho, and S. K. Kim. A simple algorithm for the constrained sequence problems. *Inf. Process. Lett.*, 90(4):175–179, 2004.

7. D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *J. Symb. Comput.*, 9(3):251–280, 1990.
8. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press and McGraw-Hill Book Company, 1989.
9. C. S. Iliopoulos and M. S. Rahman. Algorithms for computing variants of the longest common subsequence problem. *Theoretical Computer Science*, 2007. To Appear.
10. V. Mkinen, G. Navarro, and E. Ukkonen. Transposition invariant string matching. *Journal of Algorithms*, 56:124–153, 2005.
11. M. S. Rahman and C. S. Iliopoulos. Algorithms for computing variants of the longest common subsequence problem. In T. Asano, editor, *ISAAC*, volume 4288 of *Lecture Notes in Computer Science*, pages 399–408. Springer, 2006.
12. M. S. Rahman and C. S. Iliopoulos. A new efficient algorithm for computing the longest common subsequence. In M.-Y. Kao and X.-Y. Li, editors, *AAIM*, volume 4508 of *Lecture Notes in Computer Science*, pages 82–90. Springer, 2007.
13. W. Rytter. Fast recognition of pushdown automaton and context-free languages. *Information and Control*, 67(1-3):12–22, 1985.
14. Y.-T. Tsai. The constrained longest common subsequence problem. *Inf. Process. Lett.*, 88(4):173–176, 2003.
15. P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett.*, 6:80–82, 1977.
16. R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.
17. R. Williams. Matrix-vector multiplication in sub-quadratic time (some preprocessing required). In *SODA*, pages 1–11. ACM Press, 2007.