Available online at www.sciencedirect.com
ScienceDirect
Information Processing Letters 106 (2008) 13–18
Information Processing Letters
www.elsevier.com/locate/ipl

New efficient algorithms for the LCS and constrained LCS problems ☆

Costas S. Iliopoulos [1], M. Sohel Rahman [*,2,3]

Algorithm Design Group, Department of Computer Science, King's College London, Strand, London WC2R 2LS, England, UK

Received 16 May 2007; received in revised form 18 September 2007; accepted 18 September 2007

Available online 21 September 2007

Communicated by L.A. Hemaspaandra

Abstract

In this paper, we study the classic and well-studied longest common subsequence (LCS) problem and a recent variant of it, namely the constrained LCS (CLCS) problem. In the CLCS problem, the computed LCS must also be a supersequence of a third given string. In this paper, we first present an efficient algorithm for the traditional LCS problem that runs in $O(\mathcal{R}\log\log n + n)$ time, where $\mathcal{R}$ is the total number of ordered pairs of positions at which the two strings match and n is the length of the two given strings. Then, using this algorithm, we devise an algorithm for the CLCS problem having time complexity $O(p\mathcal{R}\log\log n + n)$ in the worst case, where p is the length of the third string.

Keywords: Algorithms; Combinatorial problems; Longest common subsequence

1. Introduction

The longest common subsequence (LCS) problem is one of the classical and well-studied problems in computer science having extensive applications in diverse areas. In this paper, we study the traditional LCS problem along with an interesting and newer variant of it, namely the Constrained LCS problem (CLCS). In the CLCS problem, the computed LCS must also be a supersequence of a third string (given). This problem finds its motivation from bioinformatics: in the computation of the homology of two biological sequences it is important to take into account a common specific or putative structure [17].

The longest common subsequence problem for k strings ($k > 2$) was first shown to be NP-hard [11] and later proved to be hard to be approximated [9]. The restricted but probably the more studied problem that

☆ A preliminary version appeared in [C.S. Iliopoulos, M.S. Rahman, New effcient algorithms for LCS and constrained LCS problem, in: H. Broersma, S.S. Dantchev, M. Johnson, S. Szeider (Eds.), Algorithms and Complexity in Durham 2007—Proceedings of the Third ACiD Workshop, 17–19 September 2007, Durham, UK, ACiD, Texts in Algorithmics, vol. 9, King's College, London, ISBN 978-1-904987-55-0, 2007, pp. 83–94.].

* Corresponding author.
E-mail addresses: csi@dcs.kcl.ac.uk (C.S. Iliopoulos), sohel@dcs.kcl.ac.uk (M.S. Rahman).
URL: http://www.dcs.kcl.ac.uk/adg.

[1] Supported by EPSRC and Royal Society grants.

[2] Supported by the Commonwealth Scholarship Commission in the UK under the Commonwealth Scholarship and Fellowship Plan (CSFP).

[3] On Leave from Department of CSE, BUET, Dhaka-1000, Bangladesh.

0020-0190/$ – see front matter
doi:10.1016/j.ipl.2007.09.008

deals with two strings has been studied extensively. The classic dynamic programming solution to LCS problem, invented by Wagner and Fischer [18], has $O(n^2)$ worst case running time, where n is the length of the two strings. Masek and Paterson [12] improved this algorithm using the "Four-Russians" technique [1] to reduce the worst case running time to $O(n^2/\log n)$.[4] Since then not much improvement in terms of n can be found in the literature. However, several algorithms exist with complexities depending on other parameters. For example, Myers in [13] and Nakatsu et al. in [14] presented an $O(nD)$ algorithm, where the parameter D is the simple Levenshtein distance between the two given strings [10]. 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 [8] presented an algorithm running in $O((\mathcal{R}+n)\log n)$ time. They also cited applications where $\mathcal{R} \sim n$ and thereby claimed that for these applications the algorithm would run in $O(n \log n)$ time. Very recently, Rahman and Iliopoulos presented an improved LCS algorithm running in $O(\mathcal{R}\log\log n+n)$ time [16]. 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 [4].

The CLCS problem, on the other hand, was introduced quite recently by Tsai in [17], where an algorithm was presented solving the problem in $O(pn^4)$ time, where p is the length of the third string which applies the constraint. Later, Chin et al. [6] and independently, Arslan and Eğecioğlu [2,3] presented improved algorithms with $O(pn^2)$ time and space complexity.

In this paper, similar to the work of [8], we devise efficient algorithms for LCS and CLCS problems having running time dependent on the parameter $\mathcal{R}$. The main goal of this paper is to present an efficient algorithm to solve Problem CLCS. To do that, we first devise an efficient algorithm for the traditional LCS problem that runs in $O(\mathcal{R}\log\log n+n)$ time. Then, using this algorithm, we devise an algorithm for the CLCS problem running in $O(p\mathcal{R}\log\log n+n)$ time in the worst case. It is clear that, in the worst case, we have $\mathcal{R} = O(n^2)$. Therefore, in the worst case, due to the $\log\log n$ term, our algorithm will behave slightly worse than the existing algorithms. In particular, in the extreme cases, the running time of our LCS and CLCS algorithms could be $O(n^2 \log\log n)$ and $O(pn^2 \log\log n)$ respectively. However, it is clear that, if $\mathcal{R} < n^2/\log\log n$, our CLCS algorithm will outperform the (best) $O(pn^2)$ time algorithm in the literature. For the same upper bound of $\mathcal{R}$, our LCS algorithm would beat the classic $O(n^2)$ algorithm. Our LCS algorithm also outperforms the celebrated $O((\mathcal{R}+n)\log n)$ algorithm of Hunt and Szymanski [8]. Furthermore, it is worth-mentioning that there are large number of applications for which we have $R \sim n$. Typical of such applications include finding the longest ascending subsequence of a permutation of integers from 1 to n, finding a maximum cardinality linearly ordered subset of some finite collection of vectors in 2-space etc. (for more details see [8] and references therein). Hence, in these situations, our algorithms will exhibit significant speed-up over the existing ones.

The rest of the paper is organized as follows. In Section 2, we present the preliminary concepts. Section 3 is devoted to a new $O(\mathcal{R}\log\log n+n)$ time algorithm for the LCS problem. In Section 4, using the algorithm of Section 3, we devise a new efficient algorithm to solve the CLCS problem, which runs in $O(p\mathcal{R}\log\log n+n)$ time. Finally, we briefly conclude in Section 5.

2. Preliminaries

Suppose we are given two strings $X[1\ldots n] = X[1]\,X[2]\ldots X[n]$ and $Y[1\ldots n] = Y[1]\,Y[2]\ldots Y[n]$. A subsequence $S[1\ldots r] = [1]\,S[2]\ldots S[r]$ of X is obtained by deleting $n-r$ symbols from X. A common subsequence of two strings X and Y, denoted $CS(X,Y)$, is a subsequence common to both X and Y. The longest common subsequence of X and Y, denoted $LCS(X,Y)$, is a common subsequence of maximum length. We denote the length of $LCS(X,Y)$ by $\mathcal{L}(X,Y)$.

Problem "LCS". Given 2 strings X and Y, we want to find out an $LCS(X,Y)$.

Given two strings $X[1\ldots n]$ and $Y[1\ldots n]$ and a third string $Z[1\ldots p]$, a $CS(X,Y)$ is said to be *constrained* by Z if, and only if, Z is a subsequence of that $CS(X,Y)$. We use $CS_Z(X,Y)$ to denote a common subsequence of X and Y constrained by Z. Then, the longest common subsequence of X and Y, constrained by Z, is a $CS_Z(X,Y)$ of maximum length and is denoted by $LCS_Z(X,Y)$. We denote the length of $LCS_Z(X,Y)$ by $\mathcal{L}_Z(X,Y)$. It is easy to note that $\mathcal{L}_Z(X,Y) \leqslant \mathcal{L}(X,Y)$.

Problem "CLCS". Given 2 strings X and Y and another string Z, we want to find an $LCS_Z(X,Y)$.

[4] Employing different techniques, the same worst case bound was achieved in [7]. In particular, for most texts, the achieved time complexity in [7] is $O(hn^2/\log n)$, where $h \leqslant 1$ is the entropy of the text.

Fig. 1. $LCS(X, Y)$ and $LCS_Z(X, Y)$ of Example 1.

Example 1. Suppose $X = TCCACA$, $Y = ACCAAG$ 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 that, in this case $r_Z(X, Y) < r(X, Y)$.

In this paper, we use the following notions. We say a pair (i, j), $1 \leqslant i, j \leqslant n$, defines a match, if $X[i] = Y[j]$. The set of all matches, M, is defined as follows:

$$M = \big\{(i, j) \mid X[i] = Y[j], 1 \leqslant i, j \leqslant n\big\}$$

We define $|M| = \mathcal{R}$. In what follows, we assume that $|X| = |Y| = n$. But our results can be easily extended when $|X| \neq |Y|$.

3. A new algorithm for Problem LCS

In this section, we present a new efficient algorithm to solve Problem LCS combining an interesting data structure of [5] with the techniques of [15]. The resulting algorithm will be applied to get an efficient algorithm for Problem CLCS in Section 4. We follow the simple dynamic programming formulation of [15] and improve its running time of $O(n^2 + \mathcal{R})$ to $O(\mathcal{R} \log\log n + n)$. The dynamic programming formulation from [15] is as follows:

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

Here, we have used the tabular notion $\mathcal{T}[i, j]$ to denote $\mathcal{L}(X[1 \ldots i], Y[1 \ldots j])$. In our new Algorithm, referred to as AlgLCSNew henceforth, we use a preprocessing step (Algorithm 1 in [15]), requiring $O(\mathcal{R} \log\log n + n)$ time,[5] to calculate the set M in the 'prescribed' row by row order. Then the algorithm processes each $(i, j) \in M$ in that order using Eq. (1). The efficient implementation of Eq. (1) utilizes the following facts from [15].

[5] In [15], the running time of this preprocessing step is reported as $O(\mathcal{R} \log\log n)$ under the implicit assumption that $\mathcal{R} \geqslant n$.

Fact 1. *(See [15].) Suppose $(i, j) \in M$. Then for all (i', j), $i' > i$ (resp. (i, j'), $j' > j$) we must have $\mathcal{T}[i', j] \geqslant \mathcal{T}[i, j]$ (resp. $\mathcal{T}[i, j'] \geqslant \mathcal{T}[i, j]$).*

Fact 2. *(See [15].) The calculation of a $\mathcal{T}[i, j]$, $(i, j) \in M$, $1 \leqslant i, j \leqslant n$ is independent of any $\mathcal{T}[i, q]$, $(i, q) \in M$, $1 \leqslant q \leqslant n$.*

Along with the above two facts, we use the 'BoundedHeap' data structure of [5], that supports the following operations:

Insert($\mathcal{H}$, Pos, Value, Data): Insert into the BoundedHeap $\mathcal{H}$ the position Pos with value Value and associated information Data.

IncreaseValue($\mathcal{H}$, Pos, Value, Data): If $\mathcal{H}$ does not already contain the position Pos, perform Insert($\mathcal{H}$, Pos, Value, Data). Otherwise, set this position's value to max{Value, Value$'$}, where Value$'$ is its previous value. Also, update Data 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.

The following theorem from [5] presents the time complexity of the BoundedHeap data structure.

Theorem 1. *(See [5].) 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.*

The algorithm AlgLCSNew 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 $\mathcal{H}_{i-1}$. Assume that we are considering the match $(i, j) \in M_i$, $1 \leqslant j \leqslant n$, where $M_i = \{(i, j) \mid (i, j) \in M, 1 \leqslant j \leqslant n\}$. We calculate $\mathcal{T}[i, j]$ as follows:

$$\mathcal{T}[i, j].\text{Value} = \text{BoundedMax}(\mathcal{H}_{i-1}, j).\text{Value} + 1. \quad (2)$$

$$\mathcal{T}[i, j].\text{Prev} = \text{BoundedMax}(\mathcal{H}_{i-1}, j).\text{data}.^{3} \quad (3)$$

Then we perform the following operation:

$$\text{IncreaseValue}\big(\mathcal{H}_i, j, \mathcal{T}[i, j].\text{Value}, (i, j)\big). \quad (4)$$

The correctness of the above procedure follows from Facts 1 and 2. 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 H_i to update it for the next row, i.e. row $i+1$. And, due to Fact 2, we can use H_{i-1} for the computation of the $\mathcal{T}$-values of the matches in row i and do the update in H_i (initialized at first to H_{i-1}) to make H_i ready for row $i+1$.

Next, we analyze the running time of AlgLCSNew. The preprocessing requires $O(\mathcal{R}\log\log n + n)$ time to get the set M in the required order [15]. Then, we calculate each $(i, j) \in M$ using Eqs. (2) to (4). Note carefully that, we need to use each of the two operations, BoundedMax() and IncreaseValue(), once for each of the matches in M. Therefore, according to Theorem 1, in total, the running time is $O(\mathcal{R}\log\log n + n)$. The space requirement is as follows. The preprocessing step requires $\theta(\max\{\mathcal{R}, n\})$ space [15]. And, in the main algorithm, we only need to keep track of two BoundedHeap data structures at a time, requiring $O(n)$ space (Theorem 1). So, in total the space requirement is $\theta(\max\{\mathcal{R}, n\})$.

Theorem 2. *Problem LCS can be solved in* $O(\mathcal{R}\times \log\log n + n)$ *time requiring* $\theta(\max\{\mathcal{R}, n\})$ *space.*

Algorithm 1 formally presents the algorithm. Since we are not calculating all the entries of the table, we, off course, need to use variables to keep track of the actual LCS. We remark that, AlgLCSNew uses the same basic strategy used in [16], and has the same worst case running time achieved there. However, AlgLCSNew turns out to be conceptually simpler and easier to use in solving the CLCS problem, which is handled in the next section.

```
1: Construct the set M using Algorithm 1 of [15].
   Let M_i = {(i, j) | (i, j) ∈ M, 1 ⩽ j ⩽ n}.
2: globalLCS.Instance = ϵ
3: globalLCS.Value = ϵ
4: H_0 = ϵ
5: for i = 1 to n do
6:    H_i = H_{i-1}
7:    for each (i, j) ∈ M_i do
8:       maxresult = BoundedMax(H_{i-1}, j)
9:       T[i, j].Value = maxresult.Value + 1
10:      T[i, j].Prev = maxresult.Instance
11:      if globalLCS.Value < T[i, j].Value then
12:         globalLCS.Value = T[i, j].Value
13:         globalLCS.Instance = (i, j)
14:      end if
15:      IncreaseValue(H_i, j, T[i, j].Value, (i, j)).
16:   end for
17:   Delete H_{i-1}.
18: end for
19: return globalLCS
```

Algorithm 1. AlgLCSNew.

4. CLCS algorithm

In this section, we present a new efficient algorithm for Problem CLCS. We use the dynamic programming formulation for CLCS presented in [2,3]. Extending our tabular notion from Eq. (1), we use $\mathcal{T}[i, j, k]$, $1 \leqslant i, j \leqslant n$, $0 \leqslant k \leqslant p$ to denote $\mathcal{L}_{Z[1\ldots k]}(X[1\ldots i], Y[1\ldots j])$.[4] We have the following formulation for Problem CLCS from [2,3].

$$\mathcal{T}[i, j, k] = \max\big\{\mathcal{T}'[i, j, k], \mathcal{T}''[i, j, k], \mathcal{T}[i, j-1, k], \mathcal{T}[i-1, j, k]\big\} \quad (5)$$

where

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

and

$$\mathcal{T}''[i, j, k] = \begin{cases} 1 + \mathcal{T}[i-1, j-1, k] \\ \quad \text{if } (k = 0 \text{ or } \mathcal{T}[i-1, j-1, k] > 0) \\ \quad \text{and } X[i] = Y[j], \\ 0 \quad \text{otherwise.} \end{cases} \quad (7)$$

The following boundary conditions are assumed in Eqs. (5) to (7):

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

It is straightforward to give a $O(pn^2)$ algorithm realizing the dynamic programming formulation presented in

[3] Note that, we use the 'data' field to keep the '*address*' of the match.

[4] Recall that, $p = |Z|$.

```
1: Construct the set M using Algorithm 1 of [15].
   Let M_i = (i, j) ∈ M, 1 ⩽ j ⩽ n.
2: globalCLCS.Instance = ϵ
3: globalCLCS.Value = ϵ
4: H_0^{-1} = ϵ
5: H_0^0 = ϵ
6: for i = 1 to n do
7:    for k = 0 to p do
8:       H_i^k = H_{i-1}^k
9:       for each (i, j) ∈ M_i do
10:         max_1 = BoundedMax(H_{i-1}^{k-1}, j)
11:         max_2 = BoundedMax(H_{i-1}^k, j)
            /*First, we consider Eqs. (10) and (11)/*
12:         Val_2.Value = 0
13:         Val_2.Instance = ϵ
14:         if ((k = 0) or (max_2.Value > 0)) then
15:            Val_2 = max_2
16:         end if
            /*Now, we consider Eqs. (8) and (9)/*
17:         Val_1.Value = 0
18:         Val_1.Instance = ϵ
19:         if (X[i] = Z[k])/*This means X[i] = Y[j] = Z[k]/* then
20:            if (k = 1) or ((k > 1) and (max_1.Value > 0)) then
21:               Val_1 = max_1
22:            end if
23:         end if
            /*Finally, we consider Eq. (5)/*
24:         maxresult = max{Val_1, Val_2}
25:         T_k[i, j].Value = maxresult.Value + 1
26:         T_k[i, j].Prev = maxresult.Instance
27:         if globalLCS.Value < T_k[i, j].Value then
28:            globalLCS.Value = T_k[i, j].Value
29:            globalLCS.Instance = (i, j, k)
30:         end if
31:         IncreaseValue(H_i^k, j, T[i, j].Value, (i, j, k)).
33:      end for
33:      Delete H_{i-1}^{k-1}.
34:   end for
35: end for
36: return globalLCS
```

Algorithm 2. AlgLCSNew.

Eqs. (5) to (7). Our goal is to present a new efficient algorithm using the parameter $\mathcal{R}$. In line of Eq. (1), we can reformulate Eqs. (6) and (7) as follows:

$$\mathcal{V}_1 = \max_{\substack{1\leqslant \ell_i < i \\ 1\leqslant \ell_j < j \\ (\ell_i,\ell_j)\in M}} \{\mathcal{T}[\ell_i, \ell_j, k-1]\} \tag{8}$$

$$\mathcal{T}'[i,j,k] = \begin{cases} 1+\mathcal{V}_1 & \text{if } (k=1 \text{ or } \\ & (k>1 \text{ and } \mathcal{V}_1>0)) \\ & \text{and } X[i]=Y[j]=Z[k], \\ 0 & \text{otherwise,} \end{cases} \tag{9}$$

$$\mathcal{V}_2 = \max_{\substack{1\leqslant \ell_i < i \\ 1\leqslant \ell_j < j \\ (\ell_i,\ell_j)\in M}} \{\mathcal{T}[\ell_i, \ell_j, k]\}, \tag{10}$$

$$\mathcal{T}''[i,j,k] = \begin{cases} 1 & \text{if } (i=1 \text{ or } j=1) \\ & \text{and } (i,j)\in M, \\ 1+\mathcal{V}_2 & \text{if } (k=0 \text{ or } \mathcal{V}_2>0) \\ & \text{and } (i,j)\in M, \\ 0 & \text{otherwise.} \end{cases} \tag{11}$$

Now, our goal is to implement the Eqs. (5), (8)–(11) efficiently with the help of AlgLCSNew. In the tabular notion we used, $\mathcal{T}[i,j,k]$ can be thought of as a three dimensional matrix having dimensions n, n and $p+1$. However, it would be useful, in our algorithm, to visualize it as $p+1$ 2-dimensional matrices, instead of one 3-dimensional matrix. We slightly change the notation to highlight this view and use $\mathcal{T}_k, 0 \leqslant k \leqslant p$ to denote the kth two dimensional matrix. Note that, with this new notation we have $\mathcal{T}[i,j,k] = \mathcal{T}_k[i,j],\ 1 \leqslant i,j \leqslant n,\ 0 \leqslant k \leqslant p$. It is easy to realize that, Fact 1 still holds for each of the matrices $\mathcal{T}_k,\ 1 \leqslant k \leqslant p$. And we have the following facts, extended from Fact 2, realizing the dependency between the matrices in the CLCS computation in light of Eqs. (8) to (11).

Fact 3. *The calculation of the matrix $\mathcal{T}_k,\ 0 \leqslant k \leqslant p$, is independent of any matrix $\mathcal{T}_\ell$ such that $\ell < k-1$.*

Fact 4. *The calculation of a $\mathcal{T}_k[i,j], (i,j)\in M,\ 1 \leqslant i,j \leqslant n,\ 1 \leqslant k \leqslant p$ is independent of any $\mathcal{T}_k[i,q_1]$ and $\mathcal{T}_{k-1}[i,q_2]$ such that $(i,q_1),(i,q_2)\in M, 1 \leqslant q_1,q_2 \leqslant n$.*

The algorithm, referred to as AlgCLCSNew henceforth, proceeds as follows. We consider all the matches of row i for all $p+1$ matrices and employ a slightly extended version of AlgLCSNew (Algorithm 1). We consider row $i+1$ only when the calculation of row i is completed for all $\mathcal{T}_k, 0 \leqslant k \leqslant p$. Extending the idea of AlgLCSNew, in AlgCLCSNew, we always deal with 3 BoundedHeap data structures $\mathcal{H}_i^k, \mathcal{H}_{i-1}^k$ and $\mathcal{H}_{i-1}^{k-1}$:

- $\mathcal{H}_{i-1}^k$ is required due to Eqs. (10) and (11).
- $\mathcal{H}_{i-1}^{k-1}$ is required due to Eqs. (8) and (9).
- We update $\mathcal{H}_i^k$, which is initialized to $\mathcal{H}_{i-1}^k$ when we start row i. $\mathcal{H}_i^k$ is later used when we consider the matches of row $i+1$ of the two matrices $\mathcal{T}_k$ and $\mathcal{T}_{k+1}$.

Due to Fact 4, while computing a $\mathcal{T}_k[i,j],\ (i,j)\in M,\ 1 \leqslant i,j \leqslant n,\ 1 \leqslant k \leqslant p$, we can employ the same technique with $\mathcal{H}_i^k$, $\mathcal{H}_{i-1}^k$ and $\mathcal{H}_{i-1}^{k-1}$ used in AlgLCSNew (with $\mathcal{H}_i$ and $\mathcal{H}_{i-1}$ to compute $\mathcal{T}[i,j], (i,j)\in M, 1 \leqslant j \leqslant n$). On the other hand, courtesy to Fact 3, to

realize Eqs. (8)–(11), we need only keep track of $\mathcal{H}_{i-1}^{k-1}$ and $\mathcal{H}_{i-1}^{k}$. The steps are formally stated in Algorithm 2. In light of the analysis of AlgLCSNew, it is quite easy to see that the running time is $O(p\mathcal{R}\log\log n + n)$. The space complexity, like AlgLCSNew, is dominated by the preprocessing step of [15], because in the main algorithm, we only need to keep track of the 3 BoundedHeap data structures and the third string Z, requiring in total $O(n)$ space.

Theorem 3. *Problem CLCS can be solved in* $O(p\mathcal{R} \times \log\log n + n)$ *time requiring* $\theta(\max\{\mathcal{R}, n\})$ *space.*

5. 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. In particular, we have presented an efficient algorithm for the traditional LCS problem that runs in $O(\mathcal{R}\log\log n + n)$ time. Then, using this algorithm, we have devised an algorithm for the CLCS problem having time complexity $O(p\mathcal{R}\log\log n + n)$ in the worst case. It is clear that, in the worst case, we have $\mathcal{R} = O(n^2)$. Therefore, in the extreme cases, the running time for our LCS and CLCS algorithms could be $O(n^2\log\log n)$ and $O(pn^2\log\log n)$ respectively. However, it is clear that, if $\mathcal{R} < n^2/\log\log n$, our CLCS algorithm will outperform the best $O(pn^2)$ time algorithm in the literature. For the same upper bound of $\mathcal{R}$, our LCS algorithm would beat the classic $O(n^2)$ algorithm. Our LCS algorithm also outperforms the celebrated $O((\mathcal{R}+n)\log n)$ algorithm of [8]. Finally, there are large number of applications for which we have $R \sim n$. In these cases, we achieve a very good running time of $O(pn\log\log n)$ for the CLCS problem and $O(n\log\log n)$ for the traditional LCS problem. It would be interesting to see whether our techniques can be used to other variants of LCS problem to devise similar efficient algorithms.

Acknowledgements

The authors would like to express their gratitude to the anonymous reviewers and the editor for their helpful suggestions.

References

[1] V. Arlazarov, E. Dinic, M. Kronrod, I. Faradzev, On economic construction of the transitive closure of a directed graph, Soviet Physics—Doklady 11 (1975) 1209–1210. English translation.
[2] A.N. Arslan, Ö. Eğecioğlu, Algorithms for the constrained longest common subsequence problems, in: M. Simánek, J. Holub (Eds.), The Prague Stringology Conference, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University, 2004, pp. 24–32.
[3] A.N. Arslan, Ö. Eğecioğlu, Algorithms for the constrained longest common subsequence problems, International Journal of Foundations Computer Science 16 (6) (2005) 1099–1109.
[4] L. Bergroth, H. Hakonen, T. Raita, A survey of longest common subsequence algorithms, in: String Processing and Information Retrieval (SPIRE), IEEE Computer Society, 2000, pp. 39–48.
[5] G.S. Brodal, K. Kaligosi, I. Katriel, M. Kutz, Faster algorithms for computing longest common increasing subsequences, in: M. Lewenstein, G. Valiente (Eds.), Annual Symposium on Combinatorial Pattern Matching (CPM), in: Lecture Notes in Computer Science, vol. 4009, Springer, 2006, pp. 330–341.
[6] F.Y.L. Chin, A.D. Santis, A.L. Ferrara, N.L. Ho, S.K. Kim, A simple algorithm for the constrained sequence problems, Information Processing Letters 90 (4) (2004) 175–179.
[7] M. Crochemore, G.M. Landau, M. Ziv-Ukelson, A sub-quadratic sequence alignment algorithm for unrestricted cost matrices, in: Annual ACM–SIAM Symposium on Discrete Algorithms, 2002, pp. 679–688.
[8] J.W. Hunt, T.G. Szymanski, A fast algorithm for computing longest subsequences, Communications of the ACM 20 (5) (1977) 350–353.
[9] T. Jiang, M. Li, On the approximation of shortest common supersequences and longest common subsequences, SIAM Journal on Computing 24 (5) (1995) 1122–1139.
[10] V. Levenshtein, Binary codes capable of correcting spurious insertions and deletions of ones, Problems in Information Transmission 1 (1965) 8–17.
[11] D. Maier The complexity of some problems on subsequences and supersequences, Journal of the ACM 25 (2) (1978) 322–336.
[12] W.J. Masek, M. Paterson, A faster algorithm computing string edit distances, Journal of Computer and System Sciences 20 (1) (1980) 18–31.
[13] E.W. Myers, An O(ND) difference algorithm and its variations, Algorithmica 1 (2) (1986) 251–266.
[14] N. Nakatsu, Y. Kambayashi, S. Yajima, A longest common subsequence algorithm suitable for similar text strings, Acta Informatica 18 (1982) 171–179.
[15] M.S. Rahman, C.S. Iliopoulos, Algorithms for computing variants of the longest common subsequence problem, in: T. Asano (Ed.), ISAAC, in: Lecture Notes in Computer Science, vol. 4288, Springer, 2006, pp. 399–408.
[16] M.S. Rahman, C.S. Iliopoulos, A new efficient algorithm for computing the longest common subsequence, in: M.-Y. Kao, X.-Y. Li (Eds.), AAIM, in: Lecture Notes in Computer Science, vol. 4508, Springer, 2007, pp. 82–90.
[17] Y.-T. Tsai, The constrained longest common subsequence problem, Information Processing Letters 88 (4) (2003) 173–176.
[18] R.A. Wagner, M.J. Fischer, The string-to-string correction problem, Journal of the ACM 21 (1) (1974) 168–173.