Information Processing Letters 129 (2018) 11–15

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

A hardness result and new algorithm for the longest common palindromic subsequence problem

Shunsuke Inenaga [a,*], Heikki Hyyrö [b]

[a] Department of Informatics, Kyushu University, Japan
[b] Faculty of Natural Sciences, University of Tampere, Finland

ARTICLE INFO

Article history:
Received 16 January 2017
Accepted 29 August 2017
Available online 11 September 2017
Communicated by R. Uehara

Keywords:
Algorithms
String processing
Palindromic subsequences
Longest common subsequences
Nesting rectangles

ABSTRACT

The 2-*LCPS problem*, first introduced by Chowdhury et al. (2014) [17], asks one to compute (the length of) a longest common palindromic subsequence between two given strings A and B. We show that the 2-LCPS problem is at least as hard as the well-studied longest common subsequence problem for four strings. Then, we present a new algorithm which solves the 2-LCPS problem in $O(\sigma M^2 + n)$ time, where n denotes the length of A and B, M denotes the number of matching positions between A and B, and σ denotes the number of distinct characters occurring in both A and B. Our new algorithm is faster than Chowdhury et al.'s sparse algorithm when $\sigma = o(\log^2 n \log\log n)$.

1. Introduction

Given $k \geq 2$ string, the *longest common subsequence problem* for k strings (*k-LCS problem* for short) asks to compute (the length of) a longest string that appears as a subsequence in all the k strings. Whilst the problem is known to be NP-hard for arbitrary many strings [1], it can be solved in polynomial time for a constant number of strings (namely, when k is constant).

The 2-LCS problem that concerns two strings is the most basic, but also the most widely studied and used, form of longest common subsequence computation. Indeed, the 2-LCS problem and similar two-string variants are central topics in theoretical computer science and have applications e.g. in computational biology, spelling correction, optical character recognition and file versioning. The fundamental solution to the 2-LCS problem is based on dynamic programming [2] and takes $O(n^2)$ for two given strings of length n.[1] Using the so-called "Four Russians" technique [3], one can solve the 2-LCS problem for strings over a constant alphabet in $O(n^2/\log^2 n)$ time [4]. For a non-constant alphabet, the 2-LCS problem can be solved in $O(n^2 \log\log n/\log^2 n)$ time [5]. Despite much effort, these have remained as the best known algorithms to the 2-LCS problem, and no strongly sub-quadratic time 2-LCS algorithm is known. Moreover, the following conditional lower bound for the 2-LCS problem has been shown: For any constant $\lambda > 0$, an $O(n^{2-\lambda})$-time algorithm which solves the 2-LCS problem over an alphabet of size 7 refutes the so-called strong exponential time hypothesis (SETH) [6].

In many applications it is reasonable to incorporate additional constraints to the LCS problem (see e.g. [7–16]). Along this line of research, Chowdhury et al. [17] introduced the *longest common palindromic subsequence problem* for two strings (2-*LCPS problem* for short), which asks one to compute (the length of) a longest common subsequence

* Corresponding author.
E-mail addresses: inenaga@inf.kyushu-u.ac.jp (S. Inenaga), heikki.hyyro@uta.fi (H. Hyyrö).

[1] For simplicity, we assume that input strings are of equal length n. However, all algorithms mentioned and proposed in this paper are applicable for strings of different lengths.

http://dx.doi.org/10.1016/j.ipl.2017.08.006

between strings A and B with the additional constraint that the subsequence must be a palindrome. The problem is equivalent to finding (the length of) a longest palindrome that appears as a subsequence in both strings A and B, and is motivated for biological sequence comparison [17]. Chowdhury et al. presented two algorithms for solving the 2-LCPS problem. The first is a conventional dynamic programming algorithm that runs in $O(n^4)$ time and space. The second uses sparse dynamic programming and runs in $O(M^2 \log^2 n \log\log n + n)$ time and $O(M^2)$ space,[2] where M is the number of matching position pairs between A and B.

The contribution of this paper is two-folds: Firstly, we show a tight connection between the 2-LCPS problem and the 4-LCS problem by giving a simple linear-time reduction from the 4-LCS problem to the 2-LCPS problem. This means that the 2-LCPS problem is at least as hard as the 4-LCS problem, and thus achieving a significant improvement on the 2-LCPS problem implies a breakthrough on the well-studied 4-LCS problem, to which all existing solutions [18–22] require at least $O(n^4)$ time in the worst case. Secondly, we propose a new algorithm for the 2-LCPS problem which runs in $O(\sigma M^2 + n)$ time and uses $O(M^2 + n)$ space, where σ denotes the number of distinct characters occurring in both A and B. We remark that our new algorithm is faster than Chowdhury et al.'s sparse algorithm with $O(M^2 \log^2 n \log\log n + n)$ running time [17] when $\sigma = o(\log^2 n \log\log n)$.

2. Preliminaries

Let Σ be an *alphabet*. An element of Σ is called a *character* and that of Σ^* is called a *string*. For any string $A = a_1 a_2 \cdots a_n$ of length n, $|A|$ denotes its length, that is, $|A| = n$.

For any string $A = a_1 \cdots a_m$, let A^R denote the reverse string of A, namely, $A^R = a_m \cdots a_1$. A string P is said to be a *palindrome* iff P reads the same forward and backward, namely, $P = P^R$.

A string S is said to be a *subsequence* of another string A iff there exist increasing positions $1 \le i_1 < \cdots < i_{|S|} \le |A|$ in A such that $S = a_{i_1} \cdots a_{i_{|S|}}$. In other words, S is a subsequence of A iff S can be obtained by removing zero or more characters from A.

A string S is said to be a *common subsequence* of k strings ($k \ge 2$) iff S is a subsequence of all the k strings. S is said to be a *longest common subsequence* (*LCS*) of the k strings iff other common subsequences of the k strings are not longer than S. The problem of computing (the length of) an LCS of k strings is called the *k-LCS problem*.

A string P is said to be a *common palindromic subsequence* of k strings ($k \ge 2$) iff P is a palindrome and is a subsequence of all these k strings. P is said to be a *longest common palindromic subsequence* (*LCPS*) of the k strings iff other common palindromic subsequences of the k strings are not longer than P.

In this paper, we consider the following problem:

Problem 1 *(The 2-LCPS problem).* Given two strings A and B, compute (the length of) an LCPS of A and B.

For two strings $A = a_1 \cdots a_n$ and $B = b_1 \cdots b_n$, an ordered pair (i, j) with $1 \le i, j \le n$ is said to be a *matching position pair* between A and B iff $a_i = b_j$. Let M be the number of matching position pairs between A and B. We can compute all the matching position pairs in $O(n + M)$ time for strings A and B over integer alphabets of polynomial size in n.

3. Reduction from 4-LCS to 2-LCPS

In this section, we show that the 2-LCPS problem is at least as hard as the 4-LCS problem.

Theorem 1. *The 4-LCS problem can be reduced to the 2-LCPS problem in linear time.*

Proof. Let A, B, C, and D be four input strings for the 4-LCS problem. We wish to compute an LCS of all these four strings. For simplicity, assume $|A| = |B| = |C| = |D| = n$. We construct two strings $X = A^R Z B$ and $Y = C^R Z D$ of length $4n+1$ each, where $Z = \$^{2n+1}$ and \$ is a single character which does not appear in A, B, C, or D. Then, since Z is a common palindromic subsequence of X and Y, and since $|Z| = 2n+1$ while $|A| + |B| = |C| + |D| = 2n$, any LCPS of X and Y must be at least $2n+1$ long containing Z as a substring. This implies that the alignment for any LCPS of X and Y is enforced so that the two Z's in X and Y are fully aligned. Since any LCPS of X and Y is a palindrome, it must be of form $T^R Z T$, where T is an LCS of A, B, C, and D. Thus, we can solve the 4-LCS problem by solving the 2-LCPS problem. □

Example 1. Consider four strings $A =$ `aabbccc`, $B =$ `aabbcaa`, $C =$ `aaabccc`, and $D =$ `abcbbbb` of length 7 each. Then, an LCPS of $X =$ `cccbbaa`$\$^{15}$`aabbcaa` and $Y =$ `cccbaaa`$\$^{15}$`abcbbbb` is `cba`$\$^{15}$`abc`, which is obtained by e.g., the following alignment:

```
cccbbaa$$$$$$$$$$$$$$$aabbcaa
   |/  |||||||||||||||||//
cccbaaa$$$$$$$$$$$$$$$abcbbbb
```

Observe that `abc` is an LCS of A, B, C, and D.

4. A new algorithm for 2-LCPS

In this section, we present a new algorithm for the 2-LCPS problem.

4.1. Finding rectangles with maximum nesting depth

Our algorithm follows the approach used in the sparse dynamic programming algorithm by Chowdhury et al. [17]:

[2] The original time bound claimed in [17] is $O(M^2 \log^2 n \log\log n)$, since they assume that the matching position pairs are already computed. For given strings A and B of length n each over an integer alphabet of polynomial size in n, we can compute all matching position pairs of A and B in $O(M + n)$ time.

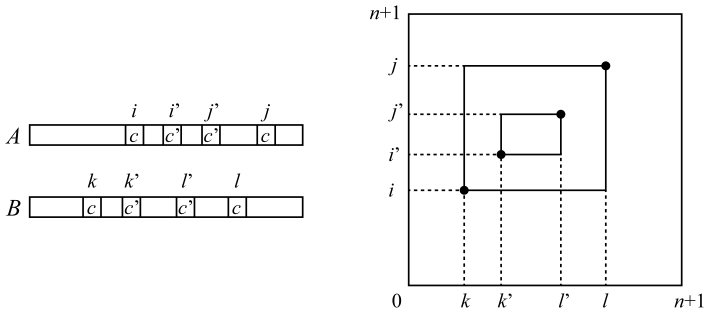

Fig. 1. Illustration for the relationship between the 2-LCPS problem and the MDNRS problem. The two nesting rectangles defined by $\langle(i,k),(j,\ell)\rangle$ and $\langle(i',k'),(j',\ell')\rangle$ correspond to a common palindromic subsequence $cc'c'c$ of A and B, where $c = c_{i,k} = c_{j,\ell}$ and $c' = c_{i',k'} = c_{j',\ell'}$.

They showed that the 2-LCPS problem can be reduced to a geometry problem called the *maximum depth nesting rectangle structures* problem (*MDNRS* problem for short), defined as follows:

Problem 2 *(The MDNRS problem).*

Input: A set of integer points (i,k) on a 2D grid, where each point is associated with a color $c \in \Sigma$. The color of a point (i,k) is denoted by $c_{i,k}$.

Output: A largest sorted list L of pairs of points, such that

1. for any $\langle(i,k),(j,\ell)\rangle \in L$, $c_{i,j} = c_{j,\ell}$, and
2. for any two adjacent elements $\langle(i,k),(j,\ell)\rangle$ and $\langle(i',k'),(j',\ell')$ in L, $i' > i$, $k' > k$, $j' < j$, and $\ell' < \ell$.

Consider two points (i,k), (j,ℓ) in the grid such that $i < j$ and $k < \ell$ (see also Fig. 1). Imagine a rectangle defined by taking (i,k) as its lower-left corner and (j,ℓ) as its upper-right corner. Clearly, this rectangle can be identified as the pair $\langle(i,k),(j,\ell)\rangle$ of points. Now, suppose that i and k are positions of one input string $A = a_1 \cdots a_m$ and j and ℓ are positions of the other input string $B = b_1 \cdots b_n$ for the 2-LCPS problem. Then, the first condition $c_{i,j} = c_{j,\ell}$ for any element in L implies that $a_i = a_j = b_k = b_\ell$, namely, i, j, k, ℓ are matching positions in A and B. Meanwhile, the second condition $i' > i$, $k' > k$, $j' < j$, and $\ell' < \ell$ implies that i', j', k', ℓ' are matching positions that are "inside" i, j, k, ℓ. Hence if we define the set of 2D points (i,k) to consist of the set of matching position pairs between A and B and then solve the MDNRS problem, the solution list L describes a set of rectangles with maximum nesting depth, and the characters that correspond to the lower-left and upper-right corner matching position pairs define an LCPS between the input strings A and B. Recall that M is the number of such pairs. As here the lower-left and upper-right corners of each rectangle corresponding to matching position pairs, the overall number of unique rectangles in this type of MDNRS problem is $O(M^2)$.

4.2. Our new algorithm

Consider the MDNRS over the set of 2D points (i,k) defined by the matching position pairs between A and B, as described above.

The basic strategy of our algorithm is to process from larger rectangles to smaller ones. Given a rectangle $R = \langle(i,k),(j,\ell)\rangle$, we locate for each character $c \in \Sigma$ a maximal sub-rectangle $\langle(i',k'),(j',\ell')\rangle$ in R that is associated to character c (namely, $c_{i',k'} = c_{j',\ell'} = c$). The following lemma is important:

Lemma 1. *For any character $c \in \Sigma$, its maximal sub-rectangle is unique (if it exists).*

Proof. Assume on the contrary that there are two distinct maximal sub-rectangles $\langle(i',k'),(j',\ell')\rangle$ and $\langle(i'',k''),(j'',\ell'')\rangle$ both of which are associated to character c. Assume w.o.l.g. that $i' > i''$, $k' < k''$, $j' < j''$ and $\ell'' > \ell'$. Then, there is a larger sub-rectangle $\langle(i'',k'),(j',\ell'')\rangle$ of R which contains both of the above rectangles, a contradiction. Hence, for any character c, a maximal sub-rectangle in R is unique if it exists. □

Lemma 1 permits us to define the following recursive algorithm for the MDNRS problem:

We begin with the initial virtual rectangle $\langle(0,0),(n+1,n+1)\rangle$. Suppose we are processing a rectangle R. For each character $c \in \Sigma$, we compute its maximal sub-rectangle R_c in R and recurse into R_c until we meet one of the following conditions:

(1) There remains only a single point in R_c,
(2) There remains no point in R_c, or
(3) R_c is already processed.

The recursion depth clearly corresponds to the rectangle nesting depth, and we associate each R with its maximum nesting depth d_R. Whenever we meet a rectangle R_c with Condition (3), we do not recurse inside R_c but simply return the already-computed maximum nesting depth d_{R_c}.

Initially, every rectangle R is marked non-processed, and it gets marked processed as soon as the recursion for R is finished and R receives its maximum nesting depth. Each already processed rectangle remains marked processed until the end of the algorithm.

Theorem 2. *Given two strings A and B of length n over an integer alphabet of polynomial size in n, we can solve the MDNRS problem (and hence the 2-LCPS problem) in $O(\sigma M^2 + n)$ time* *and $O(M^2+n)$ space, where σ denotes the number of distinct characters occurring in both A and B.*

Proof. To efficiently perform the above recursive algorithm, we conduct the following preprocessing (alphabet reduction) and construct the two following data structures.

Alphabet reduction: First, we reduce the alphabet size as follows. We radix sort the original characters in A and B, and replace each original character by its rank in the sorted order. Since the original integer alphabet is of polynomial size in n, the radix sort can be implemented with $O(1)$ number of bucket sorts, taking $O(n)$ total time. This way, we can treat A and B as strings over an alphabet $[1, 2n]$. Further, we remove all characters that occur only in A from A, and remove all characters that occur only in B from B. Let $\hat{A} = \hat{a}_1 \cdots \hat{a}_{\hat{m}}$ and $\hat{B} = \hat{b}_1 \cdots \hat{b}_{\hat{n}}$ be the resulting strings, respectively. It is clear that we can compute $\hat{A}$ and $\hat{B}$ in $O(n)$ time. The key property of the shrunk strings $\hat{A}$ and $\hat{B}$ is that since all M matching position pairs in the original strings A and B are essentially preserved in $\hat{A}$ and $\hat{B}$, it is enough to work on strings $\hat{A}$ and $\hat{B}$ to solve the original problem. If σ is the number of distinct characters occurring in *both* A and B, then $\hat{A}$ and $\hat{B}$ are strings over alphabet $[1, \sigma]$. It is clear that $\sigma \le \min\{\hat{m}, \hat{n}\} \le n$.

Data structure for finding next maximal sub-rectangles: For each character $c \in [1, \sigma]$, let $\mathcal{P}_{\hat{A},c}$ and $\mathcal{P}_{\hat{B},c}$ be the set of positions of $\hat{A}$ and $\hat{B}$ which match c, namely, $\mathcal{P}_{\hat{A},c} = \{i \mid \hat{a}_i = c, 1 \le i \le \hat{m}\}$ and $\mathcal{P}_{\hat{B},c} = \{k \mid \hat{b}_k = c, 1 \le k \le \hat{n}\}$. Then, given a rectangle R, finding the maximal sub-rectangle R_c for character c reduces to two predecessor and two successor queries on $\mathcal{P}_{\hat{A},c}$ and $\mathcal{P}_{\hat{B},c}$. We use two tables of size $\sigma \times \hat{m}$ each, which answer predecessor/successor queries on $\hat{A}$ in $O(1)$ time. Similarly, we use two tables of size $\sigma \times \hat{n}$ each, which answer predecessor/successor queries on $\hat{B}$ in $O(1)$ time. Such tables can easily be constructed in $O(\sigma(\hat{m}+\hat{n}))$ time and occupy $O(\sigma(\hat{m}+\hat{n}))$ space. Notice that for any position i in $\hat{A}$ there exists a matching position pair (i, k) for some position k in $\hat{B}$, and vice versa. Therefore, we have $\max\{\hat{m}, \hat{n}\} \le M$. Since $\sigma \le \min\{\hat{m}, \hat{n}\} \le \max\{\hat{m}, \hat{n}\}$, we have $\sigma(\hat{m}+\hat{n}) = O(M^2)$. Hence the data structure occupies $O(M^2)$ space and can be constructed in $O(M^2)$ time.

Data structure for checking already processed rectangles: To construct a space-efficient data structure for checking if a given rectangle is already processed or not, we here associate each position in $\hat{A}$ and $\hat{B}$ with the following character counts: For any position i in $\hat{A}$, let $\mathrm{cnt}_{\hat{A}}(i) = |\{i' \mid \hat{a}_{i'} = \hat{a}_i, 1 \le i' \le i\}|$ and for any position k in $\hat{B}$, let $\mathrm{cnt}_{\hat{B}}(k) = |\{k' \mid \hat{b}_{k'} = \hat{b}_k, 1 \le k' \le k\}|$. For each character $c \in [1, \sigma]$, let M_c denotes the number of matching position pairs between $\hat{A}$ and $\hat{B}$ for character c. We maintain the following table T_c of size $M_c \times M_c$: For any two matching positions pairs (i, k) and (j, ℓ) for character c (namely, $\hat{a}_i = \hat{b}_k = \hat{a}_j = \hat{b}_\ell = c$), we set $T_c[\mathrm{cnt}_{\hat{A}}(i), \mathrm{cnt}_{\hat{B}}(k), \mathrm{cnt}_{\hat{A}}(j), \mathrm{cnt}_{\hat{A}}(\ell)] = 0$ if the corresponding rectangle $\langle(i, k), (j, \ell)\rangle$ is non-processed, and set $T_c[\mathrm{cnt}_{\hat{A}}(i), \mathrm{cnt}_{\hat{B}}(k), \mathrm{cnt}_{\hat{A}}(j), \mathrm{cnt}_{\hat{A}}(\ell)] = 1$ if the corresponding rectangle is processed. Clearly, this table tells us whether a given rectangle is processed or not in $O(1)$ time. The total size for these tables is $\sum_{c \in [1,\sigma]} M_c^2 = O(M^2)$.

We are now ready to show the complexity of our recursive algorithm.

Main routine: A unique visit to a non-processed rectangle can be charged to itself. On the other hand, each distinct visit to a processed rectangle R can be charged to the corresponding rectangle which contains R as one of its maximal sub-rectangles. Since we have $O(M^2)$ rectangles, the total number of visits of the first type is $O(M^2)$. Also, since we visit at most σ maximal sub-rectangles for each of the M^2 rectangles, the total number of visits of the second type is $O(\sigma M^2)$. Using the two data structures described above, we can find each maximal sub-rectangle in $O(1)$ time and can check if it is already processed or not in $O(1)$ time. For each rectangle after recursion, it takes $O(\sigma)$ time to calculate the maximum nesting depth from all of its maximal sub-rectangles. Thus, the main routine of our algorithm takes a total of $O(\sigma M^2)$ time.

Overall, our algorithm takes $O(\sigma M^2 + n)$ time and uses $O(M^2 + n)$ space. □

5. Conclusions and further work

In this paper, we studied the problem of finding a longest common palindromic subsequence of two given strings, which is called the 2-LCPS problem. We proposed a new algorithm which solves the 2-LCPS problem in $O(\sigma M^2 + n)$ time and $O(M^2 + n)$ space, where n denotes the length of two given strings A and B, M denotes the number of matching position pairs of A and B, and σ denotes the number of distinct characters occurring in both A and B.

Since the 2-LCPS problem is at least as hard as the well-studied 4-LCS problem, and since any known solution to the 4-LCS problem takes at least $O(n^4)$ time in the worst case, it seems a big challenge to solve the 2-LCPS problem in $O(M^{2-\lambda})$ or $O(n^{4-\lambda})$ time for any constant $\lambda > 0$. This view is supported by the recent result on a conditional lowerbound for the k-LCS problem: If there exists a constant $\lambda > 0$ and an integer $k \ge 2$ such that the k-LCS problem over an alphabet of size $O(k)$ can be solved in $O(n^{k-\lambda})$ time, then the famous SETH (strong exponential time hypothesis) fails [6].

We also remark that our method should have a good expected performance. Consider two random strings A and B of length n each over an alphabet of size σ. Since roughly every σ-th character matches between A and B, we have $M = O(n^2/\sigma)$. Hence our method runs in $O(\sigma M^2 + n) = O(n^4/\sigma)$ expected time. On the other hand, the conventional dynamic programming algorithm of Chowdhury et al. [17] takes $\Theta(n^4)$ time for *any* input strings of length n each. Thus, our method achieves a σ-factor speed-up in expectation.

As an open problem, we are interested in whether the space requirement of our algorithms can be reduced, as this could be of practical importance.

References

[1] D. Maier, The complexity of some problems on subsequences and supersequences, J. ACM 25 (2) (1978) 322–336.

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

[3] V. Arlazarov, E. Dinic, M. Kronrod, I. Faradzev, On economical construction of the transitive closure of a directed graph, Sov. Math. Dokl. 11 (1970) 1209–1210.

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

[5] S. Grabowski, New tabulation and sparse dynamic programming based techniques for sequence similarity problems, Discrete Appl. Math. 212 (2016) 96–103.

[6] A. Abboud, A. Backurs, V.V. Williams, Tight hardness results for LCS and other sequence similarity measures, in: FOCS 2015, 2015, pp. 59–78.

[7] F.Y.L. Chin, A.D. Santis, A.L. Ferrara, N.L. Ho, S.K. Kim, A simple algorithm for the constrained sequence problems, Inf. Process. Lett. 90 (4) (2004) 175–179.

[8] A.N. Arslan, Regular expression constrained sequence alignment, J. Discret. Algorithms 5 (4) (2007) 647–661.

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

[10] G. Kucherov, T. Pinhas, M. Ziv-Ukelson, Regular language constrained sequence alignment revisited, J. Comput. Biol. 18 (5) (2011) 771–781.

[11] S. Deorowicz, Quadratic-time algorithm for a string constrained LCS problem, Inf. Process. Lett. 112 (11) (2012) 423–426.

[12] E. Farhana, M.S. Rahman, Doubly-constrained LCS and hybrid-constrained LCS problems revisited, Inf. Process. Lett. 112 (13) (2012) 562–565.

[13] D. Zhu, X. Wang, A simple algorithm for solving for the generalized longest common subsequence (LCS) problem with a substring exclusion constraint, Algorithms 6 (3) (2013) 485–493.

[14] E. Farhana, M.S. Rahman, Constrained sequence analysis algorithms in computational biology, Inf. Sci. 295 (2015) 247–257.

[15] D. Zhu, Y. Wu, X. Wang, An efficient algorithm for a new constrained LCS problem, in: ACIIDS 2016, 2016, pp. 261–267.

[16] D. Zhu, Y. Wu, X. Wang, An efficient dynamic programming algorithm for STR-IC-STR-EC-LCS problem, in: GPC 2016, 2016, pp. 3–17.

[17] S.R. Chowdhury, M.M. Hasan, S. Iqbal, M.S. Rahman, Computing a longest common palindromic subsequence, Fundam. Inform. 129 (4) (2014) 329–340.

[18] S.Y. Itoga, The string merging problem, BIT 21 (1) (1981) 20–30.

[19] W.J. Hsu, M.W. Du, Computing a longest common subsequence for a set of strings, BIT 24 (1) (1984) 45–59.

[20] R.W. Irving, C. Fraser, Two algorithms for the longest common subsequence of three (or more) strings, in: CPM 1992, 1992, pp. 214–229.

[21] K. Hakata, H. Imai, The longest common subsequence problem for small alphabet size between many strings, in: ISAAC 1992, 1992, pp. 469–478.

[22] Q. Wang, D. Korkin, Y. Shang, A fast multiple longest common subsequence (MLCS) algorithm, IEEE Trans. Knowl. Data Eng. 23 (3) (2011) 321–334.