Algorithms for Computing the Longest Parameterized Common Subsequence

Costas S. Iliopoulos[1,⋆], Marcin Kubica[2,⋆⋆],
M. Sohel Rahman[1,⋆⋆⋆,†], and Tomasz Waleń[2,⋆⋆]

[1] Algorithm Design Group
Department of Computer Science, Kings College London,
Strand, London WC2R 2LS, England
{csi,sohel}@dcs.kcl.ac.uk
http://www.dcs.kcl.ac.uk/adg
[2] Institute of Informatics, Warsaw University
Banacha 2, 02-097 Warszawa, Poland
{kubica,walen}@mimuw.edu.pl

Abstract. In this paper, we revisit the classic and well-studied *longest common subsequence* (LCS) problem and study some new variants, first introduced and studied by Rahman and Iliopoulos [Algorithms for Computing Variants of the Longest Common Subsequence Problem, ISAAC 2006]. Here we define a generalization of these variants, the *longest parameterized common subsequence* (LPCS) problem, and show how to solve it in $O(n^2)$ and $O(n+\mathcal{R}\log n)$ time. Furthermore, we show how to compute two variants of LCS, RELAG and RIFIG in $O(n+\mathcal{R})$ time.

1 Introduction

This paper deals with some new interesting variants of the classic and well-studied *longest common subsequence* (LCS) problem. The longest common subsequence between strings can be defined as the maximum number of common (identical) symbols between them, while preserving the order of those symbols. Therefore, the LCS problem, can be seen as an investigation for the *"closeness"* among strings. Apart from being interesting from pure theoretical point of view, the LCS problem has extensive applications in diverse areas of computer science and bioinformatics.

The LCS problem for $k > 2$ strings was first shown to be NP-hard [13] and later proved to be hard to be approximated [11]. In fact, Jiang and

⋆ Supported by EPSRC and Royal Society grants.
⋆⋆ Partially supported by the Polish Ministry of Science and Higher Education under grant N20600432/0806.
⋆⋆⋆ 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.

Li, in [11], showed that there exists a constant $\delta > 0$, such that, if LCS problem for more than 2 strings has a polynomial time approximate algorithm with performance ratio n^δ, then $P = NP$. The restricted but probably the more studied problem that deals with two strings has been studied extensively [7,8,9,14,15,16,17,19]. The classic dynamic programming solution to LCS problem (for two strings), invented by Wagner and Fischer [19], has $O(n^2)$ worst case running time, where each given string is of length n. Masek and Paterson [14] improved this algorithm using the "Four-Russians" technique [1] to reduce the worst case running time[3] to $O(n^2/\log n)$. 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 [15] and Nakatsu et al. in [17] presented an $O(nD)$ algorithm where the parameter D is the simple Levenshtein distance between the two given strings [12]. 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 [9] presented an algorithm running in $O((\mathcal{R} + n)\log n)$. They have 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. 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].

Very recently, Rahman and Iliopoulos [18,10] introduced the notion of gap-constraints in LCS and presented efficient algorithms to solve the resulting variants. The motivations and applications of their work basically come from Computational Molecular Biology and are discussed in [10]. In this paper, we revisit those variants of LCS and present improved algorithms to solve them. The results we present in this paper are summarized in the following table.

PROBLEM	INPUT	Results in [18,10]	Our Results
LPCS	X, Y, K_1, K_2 and D	$-$	$O(\min(n^2, n + \mathcal{R}\log n))$
FIG	X, Y and K	$O(n^2 + \mathcal{R}\log\log n)$	
ELAG	X, Y, K_1 and K_2	$O(n^2 + \mathcal{R}\log\log n)$	
RIFIG	X, Y and K	$O(n^2)$	$O(n + \mathcal{R})$
RELAG	X, Y, K_1 and K_2	$O(n^2 + \mathcal{R}(K_2 - K_1))$	

The rest of the paper is organized as follows. In Section 2, we present all the definitions and notations required to present the new algorithms. In

[3] Employing different techniques, the same worst case bound was achieved in [6]. In particular, for most texts, the achieved time complexity in [6] is $O(hn^2/\log n)$, where $h \leq 1$ is the entropy of the text.

Sections 3 to 5, we present new improved algorithms for all the variants discussed in this paper. Finally, we briefly conclude in Section 6.

2 Preliminaries

Suppose we are given two sequences $X[1] \dots X[n]$ and $Y[1] \dots Y[n]$. A subsequence $S[1..r] = S[1]\,S[2] \dots S[r]$ of X is obtained by deleting $[0, n-r]$ 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. In LCS problem, given two sequences, X and Y, we want to find out a longest common subsequence of X and Y.

In [18,10], Rahman and Iliopoulos introduced a number of new variants of the classical LCS problem, namely FIG, ELAG, RIFIG and RELAG problems. These new variants were due to the introduction of the notion of gap constraints in LCS problem. In this section we set up a new 'parameterized' model for the LCS problem, giving us a more general way to incorporate all the variants of it. In the rest of this section we define this new notion of parameterized common subsequence and define the variants of LCS mentioned above in light of the new framework. We remark that both the definitions of [18,10] and this paper are equivalent.

Let X and Y be sequences of length n. We will say, that the sequence C is the *parameterized common subsequence* $PCS(X,Y,K_1,K_2,D)$ (for $1 \le K_1 \le K_2 \le n$, $0 \le D \le n$) if there exist such sequences P and Q, that:

- $|C| = |P| = |Q|$; we will denote the length of these sequences by l,
- P and Q are increasing sequences of indices from 1 to n, that is: $1 \le P[i], Q[i] \le n$ (for $1 \le i \le l$), and $P[i] < P[i+1]$ and $Q[i] < Q[i+1]$ (for $1 \le i < l$),
- the sequence of elements from X indexed by P and the sequence of elements from Y indexed by Q are both equal C, that is: $C[i] = X[P[i]] = Y[Q[i]]$ (for $1 \le i \le l$),
- additionally, P and Q satisfy the following two constraints:
  - $K_1 \le P[i+1] - P[i], Q[i+1] - Q[i] \le K_2$, and
  - $|(P[i+1] - P[i]) - (Q[i+1] - Q[i])| \le D$, for $1 \le i < l$.

By $LPCS(X,Y,K_1,K_2,D)$ (*longest parameterized common subsequence*) we will denote the problem of finding the maximum length of the common

subsequence C of X and Y[4]. Now we can define the problems introduced in [18,10] using our new framework as follows.

- *FIG*(X, Y, K) (LCS problem with fixed gap) denotes the problem *LPCS*$(X, Y, 1, K, n)$,
- *ELAG*(X, Y, K_1, K_2) (LCS problem with elastic gap) denotes the problem *LPCS*(X, Y, K_1, K_2, n),
- *RIFIG*(X, Y, K) (LCS problem with rigid fixed gap) denotes the problem *LPCS*$(X, Y, 1, K, 0)$,
- *RELAG*(X, Y, K_1, K_2) (LCS problem with rigid elastic gap) denotes the problem *LPCS*$(X, Y, K_1, K_2, 0)$.

Let us denote by $\mathcal{R}$ the total number of ordered pairs of positions at which X and Y match, that is the size of the set $M = \{(i, j) : X[i] = Y[j], 1 \leq i, j \leq n\}$.

3 An $O(n^2)$ Algorithm for LPCS

The *LPCS*(X, Y, K_1, K_2, D) problem can be solved in polynomial time using dynamic programming. Let us denote by $T[i, j]$ maximum length of such a *PCS*$(X[1, \ldots, i], Y[1, \ldots, j], K_1, K_2, D)$, that ends at $X[i] = Y[j]$. Using the problem definition, we can formulate the following equation:

$$T[i, j] = \begin{cases} 0 & \text{if } X[i] \neq Y[j] \\ 1 + \max(\{0\} \cup \{T[x, y] : (x, y) \in Z_{i-K_1, j-K_1}\}) & \text{if } X[i] = Y[j] \end{cases}$$

where $Z_{i,j}$ denotes the set:

$$Z_{i,j} = \{(x, y) : 0 \leq i - x, j - y \leq K_2 - K_1, |(i - x) - (j - y)| \leq D\}$$

We will show, how to compute array T in $O(n^2)$ time using dynamic programming. But first we have to introduce an auxiliary data-structure.

3.1 Max-queue

Max-queue is a kind of priority queue that provides the maximum of the last L elements put into the queue (for a fixed L). It provides the following operations:

- `init`(Q, L) initializes Q as the empty queue and fixes the parameter L,

[4] The parameterization presented here should not be mistaken with one that can be found in the parameterized edit distance problem [2,3].

– `insert`(Q, x) inserts x into Q,
– `max`(Q) is the maximum from the last L elements put into Q (assuming, that Q is not empty).

Max-queue is implemented as a pair $Q = (q, c)$, where q is a two-linked queue of pairs, and c is a counter indexing consecutive insertions. Each element x inserted into the queue is represented by pair (i, x), where i is its index. The q contains only pairs containing these elements, that (at some moment) can be returned as answer to `max` query. These elements form a decreasing sequence. The empty queue is represented by $(\emptyset, 0)$. Insertion can be implemented as shown in Algorithm 1.

Algorithm 1: `insert`$(Q = (q, c), x)$

```
   /* Remove such pairs (i, val), that val ≤ x.                 */
1  while not empty(q) and q.tail.val ≤ x do RemoveLast(q)
2  c++
3  Enqueue(q, {index = c, val = x})
   /* Remove such pairs (i, val), that i ≤ c − L.               */
4  while q.head.index ≤ c − L do RemoveFirst(q)
```

The amortized running time of `insert` is $O(1)$. The `max` query simply returns $q.head.val$ (or 0 if the q is empty).

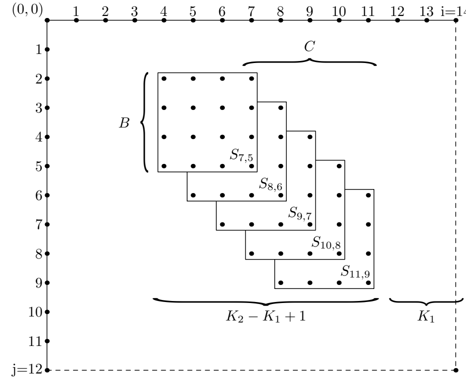

Fig. 1. Set $Z_{i-K_1, j-K_1}$, for $i = 14$, $j = 12$, $K_1 = 3$, $K_2 = 10$, and $D = 3$.

3.2 The algorithm

The set $Z_{i,j}$ has a complicated shape. It is easier to view it as a sum of squares. Let $B = \min(K_2 - K_1, D) + 1$, $C = K_2 - K_1 - B + 2$, and $S_{i,j} = \{(i - x, i - y) : 0 \leq x, y < B\}$. Then, we can define $Z_{i,j}$ as:

$$Z_{i,j} = \bigcup_{0 \leq k < C} S_{i-k,j-k}$$

To compute T, we will use three auxiliary arrays:

- $R[i, j] = \max_{k=0,\ldots,B-1} T[i - k, j]$,
- $S[i, j] = \max_{k=0,\ldots,B-1} R[i, j - k] = \max_{(x,y)\in S_{i,j}} T[x, y]$,
- $P[i, j] = \max_{k=0,\ldots,C-1} S[i - k, j - k] = \max_{(x,y)\in Z_{i,j}} T[x, y]$.

Now, $T[i, j]$ can be expressed as:

$$T[i, j] = \begin{cases} 0 & \text{if } X[i] \neq Y[j] \\ 1 + P[i - K_1, j - K_1] & \text{if } X[i] = Y[j] \end{cases}$$

We will compute all the arrays using dynamic programming, filling them row by row. We will also use max-queues to compute respective maxima — while computing elements of these arrays indexed by i and j:

- Q_R is a max-queue containing information about $T[i - B + 1 \ldots i, j]$,
- $Q_S[i]$ is a max-queue containing information about $R[i, j-B+1 \ldots j]$,
- $Q_P[i-j]$ is a max-queue containing information about $S[i, j], \ldots, S[i - C + 1, j - C + 1]$.

The value $\mathit{LPCS}(X, Y, K_1, K_2, D)$ is computed in the *GlobalMax* variable. Please note, that arrays R, S and P are introduced for the clarity of the algorithm and can be removed.

The actual longest parameterized common subsequence can be reconstructed in $O(n)$ time. Since the operations on max-queues run in $O(1)$ amortized time, total time complexity of the above algorithm is $O(n^2)$.

4 An $O(n + \mathcal{R} \log n)$ Algorithm for FIG and ELAG

For special cases, where $\mathcal{R} = o(n^2/\log n)$, we can solve ELAG (and FIG) problems more efficiently, namely in $O(n+\mathcal{R} \log n)$ running time. In order to do it, instead of computing the whole array T, we should compute only

Algorithm 2: AlgLPCS-1

```
1  Initialize R[i,j] = S[i,j] = GlobalMax = 0
2  for i = 1 to n do Init(Q_S[i], B)
3  for i = -n+1 to n-1 do Init(Q_P[i], C)
4  for j = 1 to n do
5      Init(Q_R, B)
6      for i = 1 to n do
7          if X[i] = Y[j] then
8              T[i,j] = P[i - K_1, j - K_1] + 1
9              GlobalMax = max(GlobalMax, T[i,j])
10         else
11             T[i,j] = 0
12         insert(Q_R, T[i,j]);    R[i,j] = max(Q_R)
13         insert(Q_S[i], R[i,j]);    S[i,j] = max(Q_S[i])
14         insert(Q_P[i-j], S[i,j]);    P[i,j] = max(Q_P[i-j])
```

these entries that correspond to matches from the set M. For $(i, j) \notin M$ we have $T[i, j] = 0$, and for $(i, j) \in M$ we have:

$$T[i,j] = 1 + \max\left(\{0\} \cup \left\{ \begin{array}{r} T[x,y] : (x,y) \in M, i - K_2 \le x \le i - K_1, \\ j - K_2 \le y \le j - K_1 \end{array} \right\}\right)$$

We will require data structures D and Q providing the following operations:

- *Insert*(i, j, p) — inserts element (i, j) with priority p,
- *Remove*(i, j) — removes element (i, j),
- *Priority*(i, j) — returns priority of the element (i, j), or 0 if it is not present,
- *Max*(l, r) — returns maximum priority among such elements (i, j), that $l \le i \le r$ (or 0 if there are no such elements).

We can implement the above operations in $O(\log n)$ time, using balanced search trees (such, as AVL or Red-Black trees [5]) and enriching each node with a maximum priority in the corresponding subtree.

Let $M' = \{(i - K_1, j - K_1) : (i, j) \in M\}$ and $B = K_2 - K_1 + 1$. The algorithm scans the consecutive rows of M and M'. While scanning, we keep in D information about elements from the last B rows of $T[i, j]$. Hence, when processing row j, we have:

$$D.\mathit{Max}(i - B + 1, i) = \max_{\substack{(x,y) \in M, \\ i - K_2 + K_1 \le x \le i, \\ j - K_2 + K_1 \le y \le j}} T[x, y]$$

However, instead of storing values $T[i,j]$ in an array, we store in Q pairs (i,j) (for $(i,j) \in M'$) with priorities $\max\{T[x,y] : 0 \le i-x, j-x < B\}$.

Algorithm 3: AlgELAG

```
1  Compute sets M and M' = {(i − K1, j − K1) : (i, j) ∈ M}
2  Initialize D = ∅, Q = ∅, GlobalMax = 0, B = K2 − K1 + 1
3  for j=1 to n do
4      // Remove row j − B from D.
5      for (x, y = j − B) ∈ M do  D.Remove(x, y)
6      // Insert row j into D
7      for (x, j) ∈ M do
8          Len = 1 + Q.Priority(x − K1, j − K1)
9          D.Insert(x, j, Len)
10         GlobalMax = max(GlobalMax, Len)
11     for (x, j) ∈ M' do
12         Q.Insert((x, j),  D.Max(x − B + 1, x))
```

The value $ELAG(X,Y,K_1,K_2)$ is computed in the *GlobalMax* variable. The actual longest common subsequence with elastic gap can be reconstructed in $O(n)$ time. Clearly, the overall time complexity of the above algorithm is $O(n + \mathcal{R}\log n)$.

The above algorithm can be extended to solve the LPCS problem in $O(n + \mathcal{R}\log n)$ running time.

5 An $O(n + \mathcal{R})$ Algorithm for RIFIG and RELAG

To solve the RELAG and RIFIG problems, we need to observe, that they can be reduced to $O(n)$ independent 1-dimensional problems. Since RIFIG is a special case of RELAG, for $K_1 = 1$, we will focus on the latter one. Please recall, that RELAG is equivalent to $LPCS(X,Y,K_1,K_2,0)$.

Let $T[i,j]$ denote the maximum length of such a $PCS(X[1,\ldots,i], Y[1,\ldots,j], K_1,K_2,0)$, that includes $X[i]$ and $Y[j]$. $T[i,j]$ can be computed using the following formula:

$$T[i,j] = \begin{cases} 0 & \text{if } X[i] \ne Y[j] \\ 1 + \max\{T[i-p, j-p] : K_1 \le p \le K_2\} & \text{if } X[i] = Y[j] \end{cases}$$

Let $M' = \{(i-K_1, j-K_1) : (i,j) \in M\}$, and $R[i,j] = \max\{T[i-p, j-p] : 0 \le p \le K_2 - K_1\}$. It is enough to calculate values $T[i,j]$ only for

$(i, j) \in M$, and they can be expressed as: $T[i, j] = 1 + R[i - K_1, j - K_1]$. Hence, it is enough to calculate values $R[i, j]$ only for $(i, j) \in M'$.

We will use a slightly extended version of max-queue (cf. Section 3.1). Since we process only indices $(i, j) \in M \cup M'$, we must be able to insert elements with specified indices. Let $Q = (q, c)$ be a max-queue. Operation `insert-ind`(Q, x, i) first sets the counter c to $i - 1$, and then calls `insert`(Q, x). The amortized running time of such an operation is still constant, since each element is inserted and removed once.

We will process each diagonal separately. For each $d = 1, \ldots, 2n - 1$ we scan points $(i, j) \in M \cup M'$ laying on the d-th diagonal (i.e. such that $n + i - j = d$), in order of increasing i. We will use a max-queue Q to compute values $R[i, j]$, but we will store them in a one dimensional vector $P[i]$, $P[i] = R[i, n + i - d]$. When processing $(i, j) \in M$, we can compute $T[i, j]$, as $T[i, j] = R[i - K_1, j - K_1] + 1 = P[i - K_1] + 1$. When processing $(i, j) \in M'$ we can compute $P[i]$, as $P[i] = R[i, j] = \texttt{max}(Q)$. The details are shown in Algorithm 4: AlgRELAG.

Algorithm 4: AlgRELAG

```
1  Compute sets M and M' = {(i − K1, j − K1) : (i, j) ∈ M}
2  Initialize GlobalMax = 0, P[i] = 0, for 1 ≤ i ≤ n
3  for d=1 to 2n-1 do
4      init(Q, K2 − K1 + 1) ;                      /* extended Max-Queue */
5      foreach (i, j) ∈ M ∪ M' and n + i − j = d (in order of increasing i) do
6          if (i, j) ∈ M then
7              Len = P[i − K1] + 1
8              insert-ind(Q, Len, i)
9              GlobalMax = max(GlobalMax, Len)
10         if (i, j) ∈ M' then
11             insert-ind(Q, 0, i) ;   /* phony insert, to clean up the Q */
12             P[i] = max(Q)
13     // Clean modified cells of array P
14     foreach (i, j) ∈ M' and n + i − j = d do  P[i] = 0
```

Sets M and M' can be computed and sorted in $O(n + \mathcal{R})$ time (assuming, that the alphabet is composed of polynomially bounded integer numbers). While scanning the diagonals, we have to process $|M \cup M'|$ positions, each requiring constant amortized time. Hence, the overall time complexity of the AlgRELAG is $O(n + \mathcal{R})$.

6 Conclusions

We have studied variants of the well-known LCS problem: FIG, ELAG, RIFIG and RELAG, presented in [18,10]. These problems can be seen as special cases of the more general LPCS problem, introduced here. We presented an algorithm for solving the LPCS problem in $O(n^2)$ time, that improves the previously known algorithms for FIG, ELAG and RELAG. For special cases, when $\mathcal{R} = o(n^2)$, we have also presented algorithms for RELAG and RIFIG problems running in $O(n+\mathcal{R})$ time, and for FIG and ELAG problems running in $O(n + \mathcal{R} \log n)$ time. The latter one can be extended to solve LPCS problem, without changing its running time.

References

1. V.L. Arlazarov, E.A. Dinic, M.A. Kronrod, and I.A. Faradzev. On economic construction of the transitive closure of a directed graph (english translation). *Soviet Math. Dokl.*, 11:1209–1210, 1975.
2. Brenda S. Baker. Parameterized diff. In *Symposium of Discrete Algorithms (SODA)*, pages 854–855, 1999.
3. Brenda S. Baker and Raffaele Giancarlo. Sparse dynamic programming for longest common subsequence from fragments. *Journal of Algorithms*, 42(2):231–254, 2002.
4. Lasse Bergroth, Harri Hakonen, and Timo Raita. A survey of longest common subsequence algorithms. In *String Processing and Information Retrieval (SPIRE)*, pages 39–48. IEEE Computer Society, 2000.
5. T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to algorithms.* MIT Press and McGraw Hill, Cambridge, 1992.
6. Maxime Crochemore, Gad M. Landau, and Michal Ziv-Ukelson. A sub-quadratic sequence alignment algorithm for unrestricted cost matrices. In *Symposium of Discrete Algorithms (SODA)*, pages 679–688, 2002.
7. F. Hadlock. Minimum detour methods for string or sequence comparison. *Congressus Numerantium*, 61:263–274, 1988.
8. Daniel S. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of ACM*, 24(4):664–675, 1977.
9. James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest subsequences. *Commun. ACM*, 20(5):350–353, 1977.
10. Costas S. Iliopoulos and M. Sohel Rahman. Algorithms for computing variants of the longest common subsequence problem. *Theoretical Computer Science*, page To Appear, 2007.
11. Tao Jiang and Ming Li. On the approximation of shortest common supersequences and longest common subsequences. *SIAM Journal of Computing*, 24(5):1122–1139, 1995.
12. V.I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Problems in Information Transmission*, 1:8–17, 1965.
13. David Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM*, 25(2):322–336, 1978.
14. William J. Masek and Mike Paterson. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–31, 1980.

15. Eugene W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.
16. Veli Mkinen, Gonzalo Navarro, and Esko Ukkonen. Transposition invariant string matching. *Journal of Algorithms*, 56:124–153, 2005.
17. Narao Nakatsu, Yahiko Kambayashi, and Shuzo Yajima. A longest common subsequence algorithm suitable for similar text strings. *Acta Inf.*, 18:171–179, 1982.
18. M. Sohel Rahman and Costas 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.
19. Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of ACM*, 21(1):168–173, 1974.