Information Processing Letters 113 (2013) 94–99

Contents lists available at SciVerse ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

A linear algorithm for 3-letter longest common weakly increasing subsequence ☆

Lech Duraj

Theoretical Computer Science Department, Faculty of Mathematics and Computer Science, Jagiellonian University, ul. Prof. St. Lojasiewicza 6, 30-348 Krakow, Poland

ARTICLE INFO

Article history:
Received 5 September 2012
Received in revised form 2 November 2012
Accepted 13 November 2012
Available online 12 December 2012
Communicated by A. Tarlecki

Keywords:
Algorithms
Longest common weakly increasing subsequence

ABSTRACT

The problem of finding a longest weakly increasing common subsequence (LCWIS) of two sequences is a variant of the popular longest common subsequence (LCS) problem. While there are no known methods to find LCS in truly sub-quadratic time, there are faster algorithms to compute LCWIS if the alphabet size is small enough. We present a linear-time algorithm finding LCWIS over 3-letter alphabet. Up to now, the fastest known algorithm was $O(\min\{m + n\log n, m\log\log m\})$, where $m \geqslant n$ denote lengths of the sequences.

1. Introduction

The problem of longest common weakly increasing subsequence (LCWIS) is one of the many variants of a popular longest common subsequence (LCS) problem.

We say that a sequence $X = (x_1, \ldots, x_r)$ is a subsequence of another sequence $Y = (y_1, \ldots, y_s)$, if there exist $\alpha_1 < \alpha_2 < \cdots < \alpha_r$ such that $y_{\alpha_i} = x_i$ for $i = 1, 2, \ldots, r$. The LCS problem is formulated as follows: given two sequences A, B over some alphabet Σ find the longest sequence C which is a common subsequence of A and B.

The canonical dynamic programming algorithm by Wagner and Fischer [1] solves this problem in $O(mn)$ time, where $m = |A|$, $n = |B|$. Hirschberg [2] provided a version of the same algorithm with linear space complexity. To the day, the fastest known algorithm for LCS is $O(mn/\log n)$ provided by Masek and Paterson [3]. Despite extensive studies, no further speedup has been achieved for over 30 years. Even restricting the problem to small alphabet case ($|\Sigma| = 2$) has not resulted in any faster algorithms. The question of solving LCS in $O(n^{2-\epsilon})$ time complexity is considered a very important open problem of the string algorithms theory.

In this paper we consider a natural modification of the LCS problem: we search for the common subsequences which are sorted, i.e., non-decreasing. This problem, as well as similar variants, has also been studied before, and it is known in literature as the *longest common weakly increasing subsequence* (LCWIS).

Yang, Huang and Chao [4] found a simple and clever dynamic-programming algorithm to find a longest common increasing sequence, with running time $O(mn)$. While their algorithm was designed to find a *longest common strictly increasing sequence* (LCIS), it can be easily adapted to find common sorted subsequences as well. The result was later improved by Sakai [5] by applying Hirschberg's technique and reducing space complexity to linear. As in the case of LCS, it seems hard to achieve a truly sub-quadratic complexity. Some algorithms, though, work faster under certain conditions. For example, Kutz et al. [6,7] found an algorithm for both LCIS and LCWIS that works in $O((m + n\ell)\log\log|\Sigma| + Sort_\Sigma(m))$, where ℓ denotes the output size, and $Sort_\Sigma(m)$ is the time complexity of sorting an m-element sequence over Σ. This algorithm is considerably faster for small outputs.

The authors of [6] noted that LCWIS for small alphabets behaves differently from both LCS and LCIS. Specifically,

☆ This article was supported by funding from the Jagiellonian University within the SET project. The project is co-financed by the European Union within the European Social Fund.

E-mail address: duraj@tcs.uj.edu.pl.

0020-0190/$ – see front matter
http://dx.doi.org/10.1016/j.ipl.2012.11.007

we can search for non-trivial algorithms for LCWIS of better than quadratic complexity, while no such algorithm has been found for LCS. (Note also that the problem makes little sense for strictly-increasing variant, as for small alphabets the output is also small.) A simple linear algorithm for the 2-letter case as well as two algorithms for the 3-letter case (one $O(m+n\log n)$ [6] and the other $O(m\log\log m)$ [7]) have been found up to date.

This paper improves the latter result: we show an optimal $O(m+n)$ algorithm for LCWIS for 3-letter alphabet. This improvement turns out to refine one of the ideas in [7], but we use simple list-like data structures instead of (relatively) complex van Emde-Boas priority queues, achieving better time complexity.

2. Preliminaries

Let us formally define the input and output of the algorithm:

Input: Two sequences $A, B \in \{0,1,2\}^*$.

Output: A non-decreasing sequence C such that C is a subsequence of both A and B and C is a longest such sequence.

Throughout the paper we use $m = |A|$ and $n = |B|$ as the lengths of the sequences. We denote the i-th element of sequences A and B by $A[i]$ and $B[i]$ respectively. Every common subsequence C of A and B corresponds to a matching between symbols in A and symbols in B, such that only the same symbols may be matched, and matching edges do not cross each other. Formally, we can view a common subsequence of length r as a set of pairs (i_γ, j_γ) where $\gamma = 1,2,\ldots,r$, $A[i_\gamma] = B[j_\gamma]$ and $i_1 < i_2 < \cdots < i_r$ as well as $j_1 < j_2 < \cdots < j_r$. Finding the longest matching is equivalent to LCWIS problem, as matchings can be easily constructed from common subsequences in linear time.

We start with a simple lemma, which justifies some greedy strategy for this problem:

Lemma 2.1. *Suppose the optimal LCWIS matches k 0's and l 2's. There exists an optimal LCWIS which matches the k first 0's and the l last 2's in both sequences.*

Proof. Simply replace the 0's in LCWIS by the first 0's of the sequences. This clearly cannot violate the solution correctness. Similarly, we can replace 2's with the last ones. □

Suppose that there are $z(A)$ occurrences of 0 in A and $z(B)$ in B. If $z(A) > z(B)$, we can simply delete the last $z(A) - z(B)$ zeros from A, as they do not appear in the optimal solution supplied by Lemma 2.1. Therefore we can assume without loss of generality that $z(A) = z(B)$ and denote this number by f_0. In a similar way (deleting some of the first 2's from A or B if needed) we assume that the numbers of 2's in A and B are equal and denote this number by f_2.

We define a *position* as a pair of integers (x, y) with $0 \leqslant x \leqslant m$ and $0 \leqslant y \leqslant n$. A position corresponds to a pair of (possibly empty) prefixes of A and B, though we will more often identify it with the last letters of the prefixes.

For $\alpha = (\alpha_A, \alpha_B)$ and $\beta = (\beta_A, \beta_B)$ being arbitrary positions, we use the following notions:

- We use a partial order on positions: $\alpha \leqslant \beta$ iff $\alpha_A \leqslant \beta_A$ and $\alpha_B \leqslant \beta_B$.
- For $\alpha \leqslant \beta$, we define $d(\alpha, \beta)$ as $\beta_A - \alpha_A + \beta_B - \alpha_B$.
- For $\alpha \leqslant \beta$, we use $\#_1(\alpha_A, \beta_A)$ to denote the number of 1's in $A[\alpha_A + 1 \ldots \beta_A - 1]$, and $\#_1(\alpha_B, \beta_B)$ in similar way.
- For $k = 1, 2, \ldots, f_0$, let $Z[k] = (Z[k]_A, Z[k]_B)$ be the position of k-th symbol 0 in both strings. Similarly, $T[j] = (T[j]_A, T[j]_B)$ is the position of j-th symbol 2 for $j = 1, 2, \ldots, f_2$. For convenience, we also use $Z[0] = T[0] = (0,0)$ and $Z[f_0+1] = T[f_2+1] = (m, n)$.

Fix any position $\alpha = (\alpha_A, \alpha_B)$. By a *left-matching for* α we mean a non-decreasing subsequence of $A[1 \ldots \alpha_A]$ and $B[1 \ldots \alpha_B]$ containing only 0's and 1's. We also distinguish the *k-th left-matching for α* as the one which greedily matches first k 0's and (also greedily) maximal possible number of 1's. Let $score(k, \alpha)$ denote the length of k-th left-matching. Obviously,

$$score(k, \alpha) = k + \min\{\#_1(Z[k]_A, \alpha_A), \#_1(Z[k]_B, \alpha_B)\}.$$

We also define an auxiliary function

$$surplus(k, \alpha) = \#_1(Z[k]_A, \alpha_A) - \#_1(Z[k]_B, \alpha_B).$$

This function tells us how many 1's are left unmatched between $Z[k]$ and α. Note that this value is positive if unmatched 1's are in A and negative if they are in B.

Observe that for given α and k, both *score* and *surplus* can be easily calculated in constant time if we precompute the number of 1's in every prefix of A and B.

Lemma 2.2. *The value of the optimal LCWIS is*

$$\max_{1 \leqslant j \leqslant f_2+1} \Big(f_2 + 1 - j + \max_{k:\ Z[k] \leqslant T[j]} score(k, T[j]) \Big).$$

Proof. Suppose that the optimal LCWIS matches 2 symbols starting at j-th one, which means exactly $f_2 + 1 - j$ of them (the case of $j = f_2 + 1$ corresponds to no 2's matched). Let k be the number of 0's in the solution. Clearly, $Z[k] \leqslant T[j]$. All 1's in the solution must fit between the positions $Z[k]$ and $T[j]$, and we can match no more than $\min\{\#_1(Z[k]_A, T[j]_A), \#_1(Z[k]_B, T[j]_B)\}$ of them, with greedy strategy being the optimal one. Then the length of LCWIS is equal to $score(k, T[j]) + (f_2 + 1 - j)$ for some k and j, and the formula for the best solution is obtained by taking maximum over j and k. □

3. Algorithm

From Lemma 2.2 we know that it is enough to compute for every $1 \leqslant j \leqslant f_2 + 1$ the best left-matching for position $T[j]$. Our algorithm will do exactly that: iterate over j and compute, for position $T[j]$, the maximum over k of $score(k, T[j])$ in amortized constant time. The maximum for position $T[j]$ will be calculated using the data from position $T[j-1]$.

To do this, we will maintain the set S of some left-matchings (informally, the ones that can still produce the optimal solution). Elements of S will be triples (k, s, p), where $s = score(k, T[j])$, $p = surplus(k, T[j])$, with j being the current step number.

A single step of the algorithm will consist of three subroutines:

- `Update(j)`: Calculate the score and surplus for left-matchings in S for position $T[j]$, using ones from $T[j-1]$.
- `Add(j,k)`: Process the matchings which became available since last step (all k such that $Z[k] \leqslant T[j]$ but not $Z[k] \leqslant T[j-1]$), and add them to S if necessary.
- `Check(j)`: Obtain the best left-matching for this step and compare it with the overall best solution. The solution is kept as a triple (a, b, c), corresponding to a sequence $0^a 1^b 2^c$.

Algorithm 1: 3-Letter LCWIS

```
best ← 0;
for j = 1, 2, ..., f_2 + 1 do
    /* Update(j)                                              */
    foreach (k, s, p) ∈ S do
        compute s' = score(k, T[j]) and p' = surplus(k, T[j]) from
        (k, s, p);
        replace (k, s, p) with (k, s', p');
    end
    foreach k : Z[k] ⩽ T[j] and not Z[k] ⩽ T[j − 1] do
        /* Add(j,k)                                           */
        compute s = score(k, T[j]), p = surplus(k, T[j]);
        add (k, s, p) to S if necessary;
    end
    /* Check(j)                                               */
    select (k, s, p) ∈ S such that s is maximal possible;
    if s + f_2 + 1 − j > best then
        best ← s + f_2 + 1 − j;
        solution ← (k, s − k, f_2 + 1 − j);
    end
end
return solution;
```

Up to this point, this is quite similar to the algorithm in [7], which would then use a van Emde-Boas queue for S. The key observation that allows us to use simple data structures and simultaneously reduce the complexity to linear is the following one:

Observation 3.1. *For integers k,k' and position α, if $score(k, \alpha) \geqslant score(k', \alpha)$ and either*:

- $surplus(k, \alpha) \geqslant surplus(k', \alpha) \geqslant 0$;
- *or* $surplus(k, \alpha) \leqslant surplus(k', \alpha) \leqslant 0$

then for every $\beta \geqslant \alpha$, $score(k, \beta) \geqslant score(k', \beta)$.

Proof. The two cases are symmetrical with respect to swapping A with B, therefore it is enough to prove the statement when $surplus(k, \alpha) \geqslant surplus(k', \alpha) \geqslant 0$. Let k-th left-matching at α match k `0`'s and x `1`'s, leaving y excess `1`'s of A unmatched. Let k'-th left-matching match k' `0`'s, x' `1`'s and leave y' surplus. Suppose that between α and β there are v_A ones in A and v_B in B. From the initial assumptions we have $k + x \geqslant k' + x'$ and $y \geqslant y'$. Consider two cases:

- if $v_A \geqslant v_B$, then $score(k, \beta)$ is equal to $k + x + v_B$ and $score(k', \beta) = k' + x' + v_B$, so clearly $score(k, \beta) \geqslant score(k', \beta)$;
- if $v_A < v_B$, then let $v_B - v_A = v$. After k zeros we have $x + y + v_A$ ones in A and $x + v_B = x + v_A + v$ in B. Hence, $score(k, \beta) = k + x + v_A + \min(y, v)$. In the same way we compute $score(k', \beta) = k' + x' + v_A + \min(y', v)$. As $k + x \geqslant k' + x'$ and $y \geqslant y'$, the first sum is at least the same as the second one. □

Recall that S is the set storing possible left-matchings, whose elements are triples (k, s, p). Now let S_+, S_-, S_0 be the partition of S into triples for which $p > 0$, $p < 0$ and $p = 0$, respectively. From Observation 3.1 we know that if there are triples (k, s, p) and (k', s', p') in S_+ with $s \geqslant s'$ and $p \geqslant p'$, then $s \geqslant s'$ for the whole rest of the algorithm, regardless of how these triples are updated. This allows us to simply drop (k', s', p') from S, as the corresponding left-matching would never be a candidate for the best one. Then, if we keep left-matchings of S_+ ordered strictly ascending by value of *surplus*, we can erase from S_+ every matching that has a score not strictly larger than its successors.

Therefore we choose to keep S_+ as a doubly-linked list sorted ascending by *surplus* and maintain the invariant of strictly decreasing *score*. Similarly, we order S_- by *surplus* descending (with values close to zero at the front)—the scores are then also kept strictly decreasing. The set S_0 is either empty or has one element, as we keep only the best matching with zero surplus. We denote the only triple in S_0 as $(zeros_0, score_0, 0)$. We use the following standard list operations:

- *L.front*()—returns a pointer to the first element of L;
- *L.back*()—returns a pointer to the last element of L;
- *next*(t)—returns the pointer right after t;
- *prev*(t)—returns the pointer right before t;
- *insert*(t, (k, s, p))—inserts the triple (k, s, p) after the element pointed by t;
- *push*(k, s, p)—inserts the triple (k, s, p) at the front of the list;
- *delete*(t)—removes the element pointed by t from the list;
- *zeros*(t), *score*(t), *surplus*(t)—for the triple (k, s, p) pointed by t returns k, s and p, respectively.

All these operations work in $O(1)$ time.

On S_+ we maintain the invariants $surplus(t) < surplus(next(t))$ and $score(t) > score(next(t))$ for every possible iterator t. On S_-, we maintain the invariants $surplus(t) > surplus(next(t))$ and $score(t) > score(next(t))$. We also employ one complex operation *restore*(t), which restores the score invariant on one of these lists, violated by insertion at t. We only restore violations of the type $score(t) \geqslant score(prev(t))$, assuming that other ones will never happen. The restoring is simple: we remove the violating element $prev(t)$ until either t becomes front

of the list, or until $score(t) < score(prev(t))$. It is obvious that *restore*() works in time proportional to the number of deleted elements.

It is now evident that we can find the best left-matching in constant time—it is enough to examine the front elements of S_+ and S_- and, possibly, the only one of S_0. As long as S is updated correctly, the *Check* phase is then easy to implement and produces the right answer. Let us then analyze the other two phases of the algorithm separately:

Update

The `Update` subroutine shifts the current position from $T[j-1]$ to $T[j]$ and updates scores of existing matchings in S. We will increase the position gradually, by adding characters of A and B one-by-one, in arbitrary order. In fact, only `1` symbols matter, as other ones do not change any values in S.

Procedure Update(j)

```
for i = T[j-1]_A + 1, ..., T[j]_A do
    if A[i]=1 then
        OneFromA();
end
for i = T[j-1]_B + 1, ..., T[j]_B do
    if B[i]=1 then
        OneFromB();
end
```

If a `1` from A is added, the following happens:

- All matchings in S_+ only increase their *surplus* by 1—those matchings already have excess `1`'s in A.
- For matchings in S_-, the new `1` matches to one of their the excess `1`'s, increasing their (negative) *surplus* by 1 and also increasing *score*.
- The matching in S_0 (if such one exists) increases *surplus* to 1.

These operations may result in one of the matchings of S_- having now *surplus* equal to 0—in this case, we remove it from S_- and transfer to S_0. Also, if there was a matching in S_0, it has now *surplus* equal to 1, so it should be moved to S_+. Note that the adding of a `1` symbol from B is almost identical, with roles of S_+ and S_- swapped and some signs reversed.

As these operations always affect all the elements in S_+ or all in S_-, they are possible to implement in $O(1)$ time. To do this, instead of increasing/decreasing individual values in S, we keep global modifiers $score_+$, $score_-$, $surplus_+$, $surplus_-$. Whenever we want to increase all scores in S_+, we only increase the variable $score_+$, and similarly for other global changes. It is easy to modify the functions *score*(), *surplus*(), *insert*() and *push*() to take the global modifiers into account (e.g. when inserting a triple (k, s, p) into S_+, we *de facto* insert $(k, s - score_+, p - surplus_+)$). Also, transferring a zero-surplus element is obvious, as it may only appear at the front of S_+ list. The implementation of adding a `1` symbol of A is then as follows:

Procedure OneFromA( )

```
surplus_+ ← surplus_+ + 1;
surplus_- ← surplus_- + 1;
score_- ← score_- + 1;
if S_0 ≠ ∅ then
    if score_0 > score(S_+.front()) then
        push(S_+, (zeros_0, score_0, 1));
    end
    S_0 = ∅;
end
if surplus(S_-.front()) = 0 then
    score_0 = score(S_-.front());
    zeros_0 = zeros(S_-.front());
    delete(S_-.front());
end
```

The procedure `OneFromB()` is largely identical and quite obvious, so it is omitted for shortness' sake. Both these procedures clearly take $O(1)$ time, and for every character of A and B at most one of them is invoked. Therefore the total time complexity of all `Update` calls is $O(m+n)$.

Add

The goal of the `Add` phase of step j is to insert to S all k-left-matchings with $Z[k] < T[j]$ that has not been processed before. Recall that we keep S as two sorted lists S_+ and S_- and a set S_0 of at most one element. We will be inserting triples (k, s, p) (where $s = score(k, T[j])$ and $p = surplus(k, T[j])$) into lists sorted by increasing absolute value of *surplus*. Whenever we want to do that, we have to find a right insertion place on the list by simply moving a pointer to the desired spot.

For the sake of convenience, assume that we modify both S_+ and S_- with only one common list pointer called *cursor* which can move between these lists. We also allow *cursor* to "point" at S_0. To be more specific, when *cursor* is at the front of S_+ and is moved forward, it jumps to S_0. A further forward move will result in *cursor* pointing at the front of S_-. Observe that moving between elements (k, s, p) and (k', s', p') always requires at most $|p - p'|$ moves, regardless of the signs of p and p'. This holds because every move on the list changes *surplus*(*cursor*) by at least 1.

An insertion of the triple (k, s, p) consists of:

a) Moving *cursor* to the right list, to the last spot where $|surplus(next(cursor))| \geqslant |p|$.
b) Inserting (k, s, p) at this spot or replacing the existing element with (k, s, p). We do it only if s is big enough (greater than *score* of replaced element or *score* of the next element).
c) Restoring (if needed) the invariant of decreasing scores on S_+ or S_- by a single *restore*() operation. The only possible invariant violation could appear between *cursor* and *prev*(*cursor*), so the prerequisites for *restore*() are fulfilled. It is also clear from Observation 3.1 that any removed elements would not appear in any optimal LCWIS and thus can be safely dropped.

Procedure Add(j, k)

```
s ← score(k, T[j]);
p ← surplus(k, T[j]);
/* retain cursor from previous Add()                    */
if p > 0 then
    move cursor to S+;
if p < 0 then
    move cursor to S−;
if p = 0 then
    move cursor to S0;
    if s > score0 then
        S0 = {(k, s, p)};
end
else
    move cursor to last spot with |surplus(next(cursor))| ⩾ |p|;
    if p = surplus(cursor) then
        if s > score(cursor) then
            delete(cursor);
            insert(cursor, (k, s, p));
            restore(cursor);
        end
    else
        if s > score(next(cursor)) then
            insert(cursor, (k, s, p));
            restore(cursor);
        end
end
```

The key routine is that after any insertion we keep *cursor* at the place it ended up. The next insertion (either `Add(j,k+1)` or `Add(j+1,k+1)`) will begin from this spot. As we will see in a moment, this trick ensures the linear time complexity of the algorithm.

To bound the total running time of insertions, observe first that neither b) nor c) steps can require more than $m+n$ operations. This is because there are at most $f_0 \leqslant \min\{m, n\}$ left matchings to insert in step b), and every element of S can be erased in step c) at most once. It is now enough to bound the total number of moves in a)-type steps, i.e. the number of *cursor* moves.

Fix the step number j and consider two consecutive inserted elements $(k-1, s', p')$ and (k, s, p). As stated before, during the insertion of (k, s, p) we do at most $|p - p'|$ moves, which is precisely $|surplus(k, T[j]) - surplus(k-1, T[j])|$. The special case is for the first insertion of the step j: we start moving at the spot of the last insertion from $j-1$-th step, therefore performing $|surplus(k, T[j]) - surplus(k-1, T[j-1])|$ moves at most.

We claim that:

Observation 3.2.

a) $|surplus(k, T[j]) - surplus(k-1, T[j])| \leqslant d(Z[k-1], Z[k])$;
b) $|surplus(k, T[j]) - surplus(k-1, T[j-1])| \leqslant d(T[j-1], T[j]) + d(Z[k-1], Z[k])$.

Proof. For a), observe that the difference between $surplus(k, T[j])$ and $surplus(k-1, T[j])$ comes only from `1`'s between $Z[k]$ and $Z[k-1]$. Therefore the surplus change cannot exceed the total number of characters in those two intervals.

To prove b) we need the same observation twice: the characters in $surplus(k-1, T[j-1])$ are between $Z[k-1]$ and $T[j-1]$. To see the statement, imagine that we first move the right bound from $T[j-1]$ to $T[j]$ (thus changing *surplus* by at most $d(T[j-1], T[j])$), then we move the left bound from $Z[k-1]$ to $Z[k]$, changing *surplus* by at most $d(Z[k-1], Z[k])$. □

Thus, for every inserted k-th matching, $k = 1, 2, \ldots, f_0+1$, we perform at most $d(Z[k-1], Z[k])$ moves. During first insertions of step j, for every $j = 1, 2, \ldots, f_2+1$ we perform at most additional $d(T[j-1], T[j])$ moves. The total number of moves is then bounded by:

$$\sum_{k=1}^{f_0+1} d\big(Z[k-1], Z[k]\big) + \sum_{j=1}^{f_2+1} d\big(T[j-1], T[j]\big).$$

Calculating both sums separately we obtain:

$$\begin{aligned}&\sum_{k=1}^{f_0+1} d\big(Z[k-1], Z[k]\big)\\ &\quad= \sum_{k=1}^{f_0+1} \big(Z[k]_A - Z[k-1]_A + Z[k]_B - Z[k-1]_B\big)\\ &\quad= Z[f_0+1]_A - Z[0]_A + Z[f_0+1]_B - Z[0]_B \leqslant m+n\end{aligned}$$

and similarly

$$\sum_{j=1}^{f_2+1} d\big(T[j-1], T[j]\big) \leqslant m+n.$$

Hence the total complexity of the algorithm does not exceed $O(m+n)$.

4. Final remarks

The implementation of the algorithm can be simplified—for every triple (k, s, p) in S the values s and p can be computed in constant time from k and the current position. Thus we can keep S as a sorted list of indices (without partitioning it into S_+, S_- and S_0), with *cursor* traversing this list in the same way as before. We keep an additional single pointer to the list (call it *middle*) indicating the element with $k = 0$ (or the element with the least non-negative value of k). The *middle* pointer is needed for the `Check` operation and we update it by linear search (for every `1` symbol in any of the sequences we make at most one move of *middle*). This modification was proposed by one of the paper's reviewers. We would like to thank the reviewers for this suggestion, as well as for all other comments, which greatly helped to improve the paper.

References

[1] R.A. Wagner, M.J. Fischer, The string-to-string correction problem, J. ACM 21 (1) (1974) 168–173.
[2] D.S. Hirschberg, A linear space algorithm for computing maximal common subsequences, Comm. Assoc. Comput. Mach. 18 (6) (1975) 341–343.
[3] W.J. Masek, M.S. Paterson, A faster algorithm computing string edit distances, J. Comput. System Sci. 20 (1980) 18–31.

[4] I.H. Yang, C.P. Huang, K.M. Chao, A fast algorithm for computing a longest common increasing subsequence, Inform. Process. Lett. 93 (5) (2005) 249–253.
[5] Y. Sakai, A linear space algorithm for computing a longest common increasing subsequence, Inform. Process. Lett. 99 (5) (2006) 203–207.
[6] G.S. Brodal, K. Kaligosi, I. Katriel, M. Kutz, Faster algorithms for computing longest common increasing subsequences, in: Combinatorial Pattern Matching: 17th Annual Symposium, CPM 2006, Barcelona.
[7] M. Kutz, G.S. Brodal, K. Kaligosi, I. Katriel, Faster algorithms for computing longest common increasing subsequences, J. Discrete Algorithms (2011) 314–325.