A New Efficient Algorithm for Computing the Longest Common Subsequence

Costas S. Iliopoulos and M. Sohel Rahman

Algorithm Design Group
Department of Computer Science, King's College London,
Strand, London WC2R 2LS, England
{csi, sohel}@dcs.kcl.ac.uk
http://www.dcs.kcl.ac.uk/adg

Abstract. The *longest common subsequence*(LCS) problem is a classic and well-studied problem in computer science. The LCS problem is a common task in DNA sequence analysis with many applications to genetics and molecular biology. In this paper, we present a new and efficient algorithm for solving the LCS problem for two strings. Our algorithm runs in $O(\mathcal{R} \log\log n)$ time, where $\mathcal{R}$ is the total number of ordered pairs of positions at which the two strings match.

1 Introduction

The *longest common subsequence*(LCS) problem is a classic and well-studied problem in computer science with extensive applications in diverse areas ranging from spelling error corrections to molecular biology. A subsequence of a string is obtained by deleting zero or more symbols of that string. The longest common subsequence problem for two strings, is to find a common subsequence in both strings, having maximum possible length. More formally, suppose we are given two strings $X[1..n] = X[1]X[2] \ldots X[n]$ and $Y[1..n] = Y[1]Y[2] \ldots Y[n]$. A subsequence $S[1..r] = S[1]S[2] \ldots S[r]$, $0 < r \leq n$ 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 $lcs(X, Y)$ or $LCS(X, Y)$, is a common subsequence of maximum length. We denote the length of $lcs(X, Y)$ by $r(X, Y)$. In this paper, we assume that the two given strings are of equal length. But our results can be easily extended to handle two strings of different length.

Problem "LCS". LCS Problem for 2 Strings. *Given strings X and Y, compute the Longest Common Subsequence of X and Y.*

The longest common subsequence problem for k strings ($k > 2$) was first shown to be NP-hard [14] and later proved to be hard to be approximated [11]. The restricted but, probably, the more studied problem that deals with two strings has been studied extensively [22, 17, 16, 15, 10, 9, 8]. The classic dynamic programming solution to LCS problem, invented by Wagner and Fischer [22], has $O(n^2)$ worst case running time. Masek and Paterson [15] improved this algorithm using the "Four-Russians" technique [2] to reduce the worst case running time 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 [16] 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 $\mathcal{R}$, which is the total number of ordered pairs of positions at which the two strings match. More formally, we say a pair (i, j), $1 \leq i, j \leq n$, defines a match, if $X[i] = Y[j]$. The set of all matches, M, is defined as follows:

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

Observe that $|M| = \mathcal{R}$. Hunt and Szymanski [10] presented an algorithm to solve Problem LCS 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. 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 [6].

In this paper, we revisit the much studied LCS problem for two strings and present new algorithms using some novel ideas and interesting observations. Our main result is an $O(\mathcal{R} \log \log n)$ algorithm for Problem LCS. The rest of the paper is organized as follows. In Sections 2 and 3, we present the new algorithms to solve Problem LCS. In particular, we present a new approach to solve Problem LCS in Section 2 which provides the base of our new algorithms and also report a new algorithm (LCS-I) that runs in $O(n^2 + \mathcal{R} \log \log n)$ time[1]. In Section 3, we improve the running time of LCS-I using some novel techniques. The resulting algorithm, LCS-II, runs in $O(R \log \log n)$ time. We conclude in Section 4, with some directions to future research.

[1] This running time can be improved to $O(n^2 + \mathcal{R})$.

2 A New Algorithm

In this section, we present a new algorithm for the LCS problem based on some ideas and observations introduced and employed in [19] to solve some new interesting variants of the Problem LCS. The resulting algorithm, namely LCS-I, will work in $O(n^2 + \mathcal{R}\log\log n)$ time. Note that, LCS-I is an easy extension of the algorithms presented in [19] and the main contribution of this paper is an improved algorithm, namely LCS-II, with $O(\mathcal{R}\log\log n)$ running time, to be presented in Section 3.

From the definition of LCS it is clear that, if $(i,j) \in M$, then we can calculate $\mathcal{T}[i,j], 1 \leq i,j \leq n$ by employing the following equation:

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

Here we have used the tabular notion $\mathcal{T}[i,j]$ to denote $r(X[1..i], Y[1..j])$. From Equation 1, it follows that only the entries $\mathcal{T}[i,j]$ such that $(i,j) \in M$ are useful. Therefore, we can ignore all $\mathcal{T}[i,j]$ with $(i,j) \notin M$ from the calculation. In order to do that, we need a preprocessing step to construct the set M in sorted order according to their position they would be considered in the algorithm (we consider a row by row operation). Such a preprocessing algorithm, referred to as "Algorithm Pre" in the rest of this paper, was presented in [19] which runs in $O(\mathcal{R}\log\log n)$ time. After we have computed the set M (using Algorithm Pre), we can start computing the entries $\mathcal{T}[i,j], (i,j) \in M$ according to the Equation 1. Since we are not calculating all the entries of the table, we need to use a global variable and appropriate pointers to keep track of the actual LCS. It is easy to verify that a straightforward (inefficient) implementation of Equation 1 would give us a running time of $O(\sum_{(i,j)\in M}(i-1)(j-1) + \mathcal{R}\log\log n)$. We, on the other hand, present an efficient implementation based on some interesting facts as follows.

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

Fact 2. *The calculation of the entry $\mathcal{T}[i,j], (i,j) \in M, 1 \leq i,j \leq n$ is independent of any $\mathcal{T}[\ell,q], (\ell,q) \in M, \ell = i, 1 \leq q \leq n$.* □

The idea is to avoid checking the $(i-1)(j-1)$ entries and check only $(j-1)$ (or $(i-1)$) entries instead. We maintain an array H of length n where, for $\mathcal{T}[i,j]$ we have, $H[\ell] = \max_{1<k<i,(k,\ell)\in M}(\mathcal{T}[k,\ell]), 1 \leq \ell \leq n$.

The 'max' operation, here, returns 0, if there does not exist any $(k, \ell) \in M$ within the range. Given the updated array H, we can easily perform the task by checking only the $(j-1)$ entries of H. And Fact 1 makes it easy to maintain the array H on the fly, as we proceed as follows. As usual, we proceed in a row by row manner. We use another array S, of length n, as a temporary storage. When we find an $(i, j) \in M$, after calculating $\mathcal{T}[i, j]$, we store $S[j] = \mathcal{T}[i, j]$. We continue to store in this way as long as we are in the same row. As soon as we find an $(i', j) \in M, i' > i$, i.e. we start processing a new row, we update H with new values from S.

The correctness of the above procedure follows from Fact 1 and 2. However, we don't still have the desired running time for the algorithm, because we need to check a lot of entries to implement the max operation in Equation 1. To achieve our goal, we have to be able to compute the maximum from the elements of H array in constant time. To do that we use the Range Maxima Query Problem.

Problem "RMAX". Range Maxima Query Problem. *Suppose we are given a sequence $A = a_1 a_2 ... a_n$. A Range Maxima (Minima) Query specifies an interval $I = (i_s, i_e), 1 \leq i_s \leq i_e \leq n$ and the goal is to find the index ℓ with maximum (minimum) value a_ℓ for $\ell \in I$.*

Theorem 1. *([7, 5]). The RMAX problem can be solved in $O(n)$ preprocessing time and $O(1)$ time per query.* □

So using an appropriate query on duly updated H, we can compute the max operation in Equation 1 in constant time. However there is a preprocessing time of $O(n)$ in case the array H gets updated. But since this preprocessing is needed once per row (due to Fact 2), the computational effort added is $O(n^2)$ in total. Therefore we get the following theorem.

Theorem 2. *LCS-I solve Problem LCS in $O(n^2 + \mathcal{R} \log \log n)$ time using $\theta(max(\mathcal{R}, n))$ space.* □

The outline of LCS-I is presented formally in the form of Algorithm 1. Note that we can shave off the $\log \log n$ term from the running time reported in Theorem 2 as follows. Since we have an n^2 term anyway in the running time, we do not need to compute the set M in the preprocessing step using Algorithm Pre. Instead, we consider each $\mathcal{T}[i, j], 1 \leq i, j \leq n$ and perform useful computation only when $(i, j) \in M$.

3 The Improved Algorithm

In this section, we present the main result of this paper. In particular, we improve the running time of LCS-I, as reported in Theorem 2, with some

Algorithm 1 Outline of LCS-I

```
1: Construct the set M using Algorithm Pre. Let M_i = (i, j) ∈ M, 1 ≤ j ≤ n.
2: globalLCS.Instance = ϵ
3: globalLCS.Value = 0
4: for i = 1 to n do
5:    S[i].Value = 0 {Initialize the temporary array S}
6:    S[i].Instance = ϵ
7: end for
8: for i = 1 to n do
9:    H = S{Update H for the next row}
10:   Preprocess H.Value for Range Maxima Query
11:   for each (i, j) ∈ M_i do
12:      maxindex = RMQ_H(1, j − 1){Range Maxima Query on Array H}
13:      T.Value[i, j] = H[maxindex].Value + 1
14:      T.Prev[i, j] = H[maxindex].Instance
15:      S[j].Value = T.Value[i, j]
16:      S[j].Instance = (i, j)
17:      if globalLCS.value < T.Value[i, j] then
18:         globalLCS.Value = T.Value[i, j]
19:         globalLCS.Instance = (i, j)
20:      end if
21:   end for
22: end for
23: return globalLCS
```

nontrivial modifications. The resulting Algorithm, LCS-II, will eventually run in $O(\mathcal{R} \log\log n)$ time. As is explained in the previous section, LCS-I exploits the constant time query operation (Theorem 1) of Problem RMAX. However, due to the $O(n)$ preprocessing step of RMAX, we can't eliminate the n^2 term from the running time of LCS-I. But a very important, albeit easily observable, fact is that the range maxima queries made in LCS-I is always of a special form.

Fact 3. *All the range maxima queries in Algorithm LCS-I are of the form* $RMQ(1, j),\ 0 \leq j \leq n.$ □

From Fact 3, it seems that Problem RMAX may be too general a tool to solve LCS and it seems to be worthwhile to look for a better solution exploiting the special query structure reported in Fact 3. Indeed, as we shall show that we can exploit this special structure in the query to avoid the $O(n)$ preprocessing step and hence the n^2 term from the running time reported in Theorem 2. However the price we pay is that the query time increases to $O(\log\log n)$. We present the idea as follows.

Assume that we have an array $A[1..n]$ on which we want to apply the range maxima queries. We now use an elegant data structure (referred

$$... \rightarrow (\alpha_i, x_i) \rightarrow (\alpha_j, x_j) \rightarrow (\alpha_k, x_k) \rightarrow ...$$

Fig. 1. Partial E_A with e_i, e_j, and e_k

to as vEB data structure henceforth) invented by van Emde Boas [21] that allows us to maintain a sorted list of integers in the range $[1..n]$ in $O(\log \log n)$ time per insertion and deletion. In addition to that it can return $next(i)$ (successor element of i in the list) and $prev(i)$ (predecessor element of i in the list) in constant time. We maintain a vEB data structure E_A, where each element $e_i \in E, 1 \leq i \leq |E_A|$ is a 2-tuple $(Value, Pos)$. The order in E_A is maintained according to $e_i.Pos, 1 \leq i \leq |E_A|$. Now consider 3 entries $e_i, e_j, e_k \in E_A$ such that $e_j = next(e_i), e_k = next(e_j)$. Let $e_i = (\alpha_i, x_i), e_j = (\alpha_j, x_j)$ and $e_k = (\alpha_k, x_k)$ (Figure 1). The invariant we maintain is as follows:

$$RMQ(1..x) = \alpha_i,\ prev(e_i).Pos < x \leq x_i$$

$$RMQ(1..x) = \alpha_j,\ x_i < x \leq x_j$$

$$RMQ(1..x) = \alpha_k,\ x_j < x \leq x_k$$

Assuming that we have the above data structure at our disposal, answering a query is easy as follows. Consider a query $RMQ(1..x')$. To answer this query, we just need to return the 'Value' of the element, which would be next in order, if a new element with $Pos = x'$ were inserted in E_A. So, we create an entry $e' = (null, x')$ and insert it in E_A and get the 'Value' of the $Next(e')$. Finally, we delete $e' = (null, x')$ from E_A. The only thing we need to ensure is that if there is already an entry e in E_A such that $e.Pos = x'$, e' must be placed before e in E_A. This is to ensure that $Next(e') = e$, as required. This can be easily achieved if we take 'Value' into account while preserving the order in E_A for equal values of 'Pos' and assume 'null' to be a lesser value than any other 'Value'. Note, however, that, by definition, in 'normal' state, there can be no two elements in E_A having same value for 'Pos'.The steps are formally presented in Algorithm 2.

The correctness of Algorithm 2 follows from the invariants maintained for E_A. Now it remains to show how we can maintain that invariant under update operations in the context of the Algorithm LCS-I. Recall that our goal is to get the answer of appropriate range maxima queries on the array H in Algorithm LCS-I and we operate in a row by row basis. For the sake of convenience, we use the following notation.

$$M_i = \{(i, j) | X[i] = Y[j], 1 \leq j \leq n\}$$

Algorithm 2 Steps to answer the query $RMQ(1..x')$ on array A

1: Insert $e' = (null, x')$ in E_A
2: $Result = Next(e').Value$
3: Delete e' from E_A
4: **return** Result

We start with reporting the following fact.

Fact 4. *$\mathcal{T}[i, j] = 1$, for all $(i, j) \in M_1$.* □

In cases, where $M_1 = \emptyset$ or a number of subsequent $M_i = \emptyset, i > 1$, we have the following fact.

Fact 5. *$\mathcal{T}[i, j] = 1$, for all $(i, j) \in M_i$ such that $M_k = \emptyset$, for all $1 \leq k < i$.* □

$$(0, j' - 1) \rightarrow (1, n) \rightarrow (\infty, \infty)$$

Fig. 2. Initial E_H

We initialize E_H with three elements, $e_s = (0, j' - 1)$, $e_e = (1, n)$ and $e_\infty = (\infty, \infty)$, where $(1, j') \in M_1$ and there exists no $j < j'$ such that $(1, j) \in M_1$ (Figure 2). Note that, if $M_1 = \emptyset$ then, for initialization, we have to use M_i instead of M_1 such that $M_k = \emptyset$, for all $1 \leq k < i$ (Fact 5). This initialization of E_H correctly maintains the invariants for the processing of the next row. Indeed, for the next row, we must have $RMQ(1..x) = 0$ if $x \leq j' - 1$ ($j' - 1$ is defined as above) and $RMQ(1..x) = 1$ otherwise. The last element, e_∞, is required to tackle the general cases and here we assume ∞ to be greater than any number. Now let us consider the case, where we process the subsequent rows. It is important to note that as we process a particular row i, for each $(i, j) \in M_i$, we need to update E_H; but this update is effective only for the next row, i.e. row $i + 1$. So, as we process row i we perform the update on a temporary copy and as soon as row i is completely processed we actually change the E_H to make it ready for row $i+1$. In what follows, for the sake of convenience, we denote by E_H^i the 'state' of E_H which is used to process row i.

Now consider the case that we are in row i and processing the match $(i, x' + 1) \in M_i$. It is easy to see that we need the answer of the query $RMQ(1..x')$, which can be obtained easily applying Algorithm 2 on E_H^i. So, according to LCS-I, we would compute $\mathcal{T}[i, x'+1] = RMQ(1..x')+1 =$

$$... \rightarrow (\alpha_i, x_i) \rightarrow (\alpha_j, x_j) \rightarrow (\alpha_k, x_k) \rightarrow ...$$

Fig. 3. Partial E_H^{i+1} with e_i, e_j, and e_k

$\alpha'(let)$. Now we need to consider the updating of E_H^i to get the E_H^{i+1} to be used when processing row $i+1$. We initialize E_H^{i+1} with E_H^i and for each match $(i,j) \in M_i$ we continue to update E_H^{i+1} so that we get the 'correct' E_H^{i+1} as soon as the processing of row i is finished. The update process is as follows. In what follows, we assume (without the loss of generality) that we have $e_i, e_j, e_k \in E_H^{i+1}$ such that $e_j = next(e_i), e_k = next(e_j)$ (Figure 3). Let $e_i = (\alpha_i, x_i), e_j = (\alpha_j, x_j)$ and $e_k = (\alpha_k, x_k)$. Assume, without the loss of generality, that $x_i < x' + 1 \leq x_j$. Since we have the value α' at position $x'+1$, the query $RMQ(1..x'+1)$ should return $\zeta \geq \alpha'$ when we are processing subsequent rows. So, first we check whether $RMQ(1..x'+1) \geq \alpha'$ on the current E_H^{i+1}. It is clear that if the answer is positive, we don't need to do any update at all. Otherwise, we have, at position x_j or before it (off course after x_i) a higher value α'. So we insert a new element (α_j, x') to E_H^{i+1}, because up to x' we have no change in the RMQ answers. Now we have to change $e_j = (\alpha_j, x_j)$. But this change may be influenced by $e_k = (\alpha_k, x_k)$ as follows. We have two cases.

Case 1.a: $\alpha_k = \alpha'$. In this case, we just need to delete e_j because $e_k = (\alpha_k = \alpha', x_k)$ has already taken into account the updated value α' at position $x'+1 \leq x_k$ (Figure 4).

$$... \rightarrow (\alpha_i, x_i) \rightarrow (\alpha_j, x') \rightarrow (\alpha_k = \alpha', x_k) \rightarrow ...$$

Fig. 4. Updated E_H for Case 1.a

Case 1.b: $\alpha_k > \alpha'$. In this case, $e_k.Value$ is greater than the updated value at position $x'+1 \leq x_j$. So it is clear that up to position x_j, we have α' as the highest value and hence we need to update e_j such that $e_j.value = \alpha'$ (Figure 5).

$$... \rightarrow (\alpha_i, x_i) \rightarrow (\alpha_j, x') \rightarrow (\alpha', x_j) \rightarrow (\alpha_k, x_k) \rightarrow ...$$

Fig. 5. Updated E_H for Case 1.b

So far we have discussed the algorithm without analyzing the running time. Theorem 3 below reports the running time of the Algorithm LCS-II.

Theorem 3. *LCS-II solves Problem LCS in* $O(\mathcal{R} \log\log n)$ *time.*

Proof. It is clear that for each $(i, j) \in M$ we do the following 3 steps.

1: Perform appropriate range maxima query using Algorithm 2
2: Compute $\mathcal{T}[i, j]$ from the result of Step 1
3: Update E_H based on $\mathcal{T}[i, j]$

It is easy to see that Step 1, i.e., Algorithm 2 requires $O(\log\log n)$ time. Step 2 requires $O(1)$ time. In Step 3 we perform the update. Here, we first check, using Algorithm 2, whether any update is indeed required. This, again, requires $O(\log\log n)$ time. Finally, if update is required, then we need to perform constant number of insertion and/or deletion requiring, again, $O(\log\log n)$ time. So, for each $(i, j) \in M$ the computation effort spent is $O(\log\log n)$. Therefore, Algorithm LCS-II requires $O(\mathcal{R} \log\log n)$ time to compute LCS. □

4 Conclusion

In this paper, we have studied the classic and much studied LCS problem for two strings. Problem LCS has been the focus of extensive research from both theoretical and practical point of view. Using some new ideas, we have presented an $O(\mathcal{R} \log\log n)$ time algorithm to solve the problem where $\mathcal{R}$ is the total number of ordered pairs of positions at which the two strings match. Although, $\mathcal{R} = O(n^2)$, 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 [10] and references therein). So in these situations our algorithm would exhibit an almost linear $O(n \log\log n)$ behavior. The techniques we have used to develop our algorithm is new and, we believe, of independent interest. We believe a number of interesting issues remain as candidates for future research as follows.

1. LCS problem for more than two strings have extensive applications e.g. in molecular biology. It would be interesting to see whether our techniques can be extended for LCS problems involving more than two strings or variants thereof (e.g. constrained LCS [20, 4, 3], rigid LCS [13]), motivated by practical applications in molecular biology.

2. We have implicitly presented an algorithm for the range maxima query problem. Our algorithm allows restricted dynamic updates and considers a restricted sets of queries. It would be interesting to see whether we can lift the restrictions and/or improve the query and pre-processing time. Moreover, we believe that this algorithm could be used in many other problems requiring similar sort of restricted updates and queries.

References

1. Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Meyers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.
2. 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.
3. Abdullah N. Arslan and Ömer Egecioglu. Algorithms for the constrained longest common subsequence problems. In *Stringology*, pages 24–32, 2004.
4. Abdullah N. Arslan and Ömer Egecioglu. Algorithms for the constrained longest common subsequence problems. *Int. J. Found. Comput. Sci.*, 16(6):1099–1109, 2005.
5. Michael A. Bender and Martin Farach-Colton. The lca problem revisited. In *LATIN*, pages 88–94, 2000.
6. Lasse Bergroth, Harri Hakonen, and Timo Raita. A survey of longest common subsequence algorithms. In *SPIRE*, pages 39–48, 2000.
7. H. Gabow, J. Bentley, and R. Tarjan. Scaling and related techniques for geometry problems. In *STOC*, pages 135–143, 1984.
8. F. Hadlock. Minimum detour methods for string or sequence comparison. *Congressus Numerantium*, 61:263–274, 1988.
9. Daniel S. Hirschberg. Algorithms for the longest common subsequence problem. *J. ACM*, 24(4):664–675, 1977.
10. James W. Hunt and Thomas G. Szymanski. A fast algorithm for computing longest subsequences. *Commun. ACM*, 20(5):350–353, 1977.
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. Bin Ma and Kaizhong Zhang. On the longest common rigid subsequence problem. In *CPM*, pages 11–20, 2005.
14. David Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM*, 25(2):322–336, 1978.
15. William J. Masek and Mike Paterson. A faster algorithm computing string edit distances. *J. Comput. Syst. Sci.*, 20(1):18–31, 1980.
16. Eugene W. Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.
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. W.R. Pearson and D.J. Lipman. Improved tools for biological sequence comparison. *Proceedings of National Academy of Science, USA*, 85:2444–2448, 1988.
19. M. Sohel Rahman and Costas S. Iliopoulos. Algorithms for computing variants of the longest common subsequence problem. In *ISAAC*, pages 399–408, 2006.
20. Yin-Te Tsai. The constrained longest common subsequence problem. *Inf. Process. Lett.*, 88(4):173–176, 2003.
21. P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6:80–82, 1977.
22. Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.