

Restricted LCS

Zvi Gotthilf¹, Danny Hermelin^{*2},
Gad M. Landau², and Moshe Lewenstein¹

¹ Department of Computer Science, Bar-Ilan University,
Ramat Gan 52900, Israel

`gotthiz@cs.biu.ac.il`, `moshe@cs.biu.ac.il`

² Department of Computer Science, University of Haifa,
Mount Carmel, Haifa 31905, Israel

`danny@cri.haifa.ac.il`, `landau@cs.haifa.ac.il`

Abstract. The *Longest Common Subsequence* (LCS) of two or more strings is a fundamental well-studied problem which has a wide range of applications throughout computational sciences. When the common subsequence must contain one or more *constraint strings* as subsequences, the problem becomes the *Constrained LCS* (CLCS) problem. In this paper we consider the *Restricted LCS* (RLCS) problem, where our goal is finding a longest common subsequence between two or more strings that does not contain a given set of *restriction strings* as subsequences. First we show that in case of two input strings and an arbitrary number of restriction strings the RLCS problem is NP-hard. Afterwards, we present a dynamic programming solution for RLCS and we show that this algorithm implies that RLCS is in FPT when parameterized by the total length of the restriction strings. In the last part of this paper we present two approximation algorithms for the hard variants of the problem.

1 Introduction

Given a set of strings A_1, \dots, A_m , a *common subsequence* of these strings is a string S which appears as a subsequence in each of A_1, \dots, A_m , *i.e.* it can be obtained from each A_i by deleting (possibly none) letters. The Longest Common Subsequence (LCS) problem is the problem of determining the (length of the) longest common subsequence between a given set of strings. LCS is a fundamental problem in computer science, and has therefore been thoroughly studied (see *e.g.* [1, 5, 10, 11]). The problem had also been investigated on more general structures such as trees and matrices [2], run-length encoded strings [3], and more.

The *Constrained Longest Common Subsequence* (CLCS) for is an extension of the LCS problem, where now we are given a set of constraint strings B_1, \dots, B_ℓ in addition to the set of comparison strings, and the goal is to find the longest common subsequence of the comparison strings that contains each of B_1, \dots, B_ℓ

* Supported by the Adams Fellowship of the Israel Academy of Sciences and Humanities.

as a subsequence. Quite a few results on the CLCS problem were presented recently [4, 7–9, 14].

In this paper, we consider the “opposite” extension of the LCS problem, namely the Restricted LCS (RLCS) problem:

Definition 1 (Restricted LCS). *Given m input strings A_1, \dots, A_m and ℓ restriction strings B_1, \dots, B_ℓ , the Restricted LCS (RLCS) problem is the problem of computing the (length of the) longest common subsequence of A_1, \dots, A_m that does not contain each of B_1, \dots, B_ℓ as a subsequence.*

As far as we know, this extension of the LCS problem has yet been considered. We believe that RLCS might be better suited than its counterpart CLCS for some scenarios, *e.g.* biological or data-mining applications.

1.1 Related work

One of the goals of this paper is to compare the RLCS problem with CLCS. We therefore briefly describe the state of the art of CLCS. The problem was first introduced by Tsai [14] where he presents a dynamic programming algorithm for the simplest case of two comparison strings and a single constraint string. Improved dynamic programming algorithms were proposed in [4, 7]. In [9], fast approximation algorithms were designed for this basic CLCS variant. In [8], it is proven that it is NP-hard to approximate CLCS within any factor, even in case of two input strings and an arbitrary number of constraint strings. Moreover, a factor $\frac{1}{\sqrt{n_{min}|\Sigma|}}$ approximation algorithm is presented for the case of many input strings and a single constraint, where n_{min} denotes the length of the shortest comparison string, and $|\Sigma|$ denotes the number of different letters occurring in both the comparison and constraint strings.

1.2 Our contribution

We focus on several different settings of RLCS. First, in section 3, we show that the problem is NP-hard even in the case of two comparison strings and an arbitrary number of restrictions, each of length at most 2. Afterwards, in Section 4, we present an $O(n^{m+\ell})$ dynamic programming solution for the RLCS problem, where m and ℓ respectively denote the number of comparison and restriction strings. We also show in this section that this algorithm implies that RLCS is in FPT when parameterized by the total length of the restriction strings. Finally, in Section 5, we present two simple approximation algorithms for the problem: The first having an approximation ratio of a $\frac{1}{|\Sigma|}$, and is suited for the most general variant of the RLCS problem, and the second having a ratio of $\frac{k_{min}-1}{n_{min}}$, and is relevant only for instances with a constant number of input strings. Here n_{min} and k_{min} are the lengths of the shortest input string and the shortest restriction, respectively, and $|\Sigma|$ is the number of different letters occurring in all strings of the instance. Section 2 is devoted to fixing some notation and simplifications.

2 Preliminaries

All strings considered in this paper are defined over some fixed alphabet Σ which can have arbitrary (including infinite) cardinality. For a string S , we use $|S|$ to denote the length of S . For $i \in \{1, \dots, |S|\}$, we write $S[i]$ for the letter at the i 'th position in S , and $S[[i]]$ for the i 'th prefix of S , *i.e.* $S[[i]] = S[1] \cdots S[i]$.

We will use A_1, \dots, A_m and B_1, \dots, B_ℓ to denote the set of *comparison* and *restriction* strings in our input of the RLCS problem. Typically, we will use the letter i to index the comparison strings, and the letter j to index the restriction strings. For $i \in \{1, \dots, m\}$, we write n_i for $|A_i|$, and for $j \in \{1, \dots, \ell\}$, we use k_j to denote $|B_j|$. Finally, n is used to denote the total length of the comparison strings, *i.e.* $n = \sum_{i=1}^m n_i$, and k to denote the total length of the restriction strings, *i.e.* $k = \sum_{j=1}^{\ell} k_j$.

We will make two assumptions that will not introduce any loss of generality, yet will help in simplifying matters somewhat. The first assumption is that all restriction strings have length at least 2, since if any single character appears as a restriction, we can simply delete all its occurrences from the comparison strings, and proceed without this restriction. The second assumption is that for all $i \in \{1, \dots, \ell\}$ and all $j \in \{1, \dots, \ell\}$, we have $k_j \leq n_i$, since otherwise we can remove the j 'th restriction as it will never appear in a common subsequence of A_1, \dots, A_m . Thus, $k \leq n$, and so we can think of n as the total input length.

3 Hardness Result

In this section we prove the following hardness result for RLCS:

Theorem 1. *The RLCS problem in case of two comparison strings and an arbitrary number of restrictions, each of length 2, is NP-hard.*

Note that a valid solution to RLCS can always be easily found, as opposed to the CLCS problem where it is NP-hard to determine whether a given instance has any valid solutions. In this sense, proving hardness of approximation for RLCS is somewhat more challenging in comparison to CLCS. Nevertheless, to prove Theorem 1, we deploy a reduction which is similar to the one used for CLCS in [8].

The reduction we use is from the 6-OCC-MAX-2SAT problem which is defined as follows: Given a CNF formula ϕ with clauses of size 2, and where every variable appears in at most 6 clauses, the goal is to find an assignment of the variables that maximize the number of satisfied clauses in ϕ . Berman and Karpinski [6] showed that 6-OCC-MAX-2SAT is APX-hard.

Given a 6-OCC-MAX-2SAT instance ϕ with variables x_1, \dots, x_{n_ϕ} and clauses c_1, \dots, c_ℓ , we construct an RLCS instance $I_\phi = (A_1, A_2, B_1, \dots, B_\ell)$ over the alphabet $\Sigma = \{c_1, \dots, c_\ell\} \cup \{s\}$, where s is a special *padding character*. The string A_1 is constructed as follows: For each variable x_i , if c_{i_1}, \dots, c_{i_t} are the clauses satisfied by setting $x_i = 1$, and $c_{i'_1}, \dots, c_{i'_t}$ are the clauses satisfied by

setting $x_i = 0$, we construct a pair of substrings

$$X_i = "c_{i_1}, \dots, c_{i_t} c'_{i'_1}, \dots, c'_{i'_t}" \text{ and } X'_i = "c'_{i'_1}, \dots, c'_{i'_t} c_{i_1}, \dots, c_{i_t}."$$

The comparison strings A_1 and A_2 are then constructed as

$$A_1 = X_1 s^6 X_2 s^6 \dots s^6 X_{n_\phi} \text{ and } A_2 = X'_1 s^6 X'_2 s^6 \dots s^6 X'_{n_\phi},$$

where $s^6 = 'ssssss'$. Finally, we let $B_j = "c_j c_j"$ for all $j \in \{1, \dots, \ell\}$.

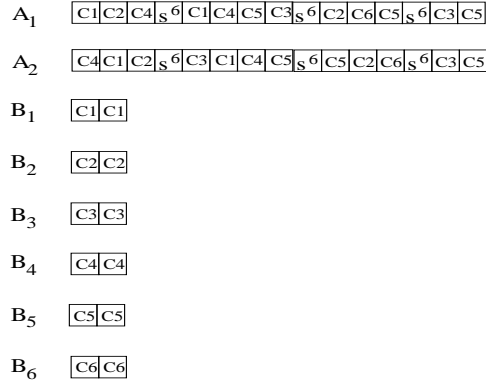


Fig. 1. An example of the instance I_ϕ constructed out of the 2-SAT formula $\phi = (x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (\bar{x}_2 \vee \bar{x}_4) \wedge (\bar{x}_1 \vee x_2) \wedge (x_2 \vee \bar{x}_3) \wedge (x_3 \vee x_4)$.

Note that the instance I_ϕ satisfies the requirements of the theorem and can be constructed in polynomial time. To complete the proof of Theorem 1, we have the following lemma:

Lemma 1. w clauses can be satisfied in ϕ iff there is an solution to I_ϕ of length $6(n_\phi - 1) + w$.

Proof. For simplicity, we assume that there are no clauses in ϕ that contains both x_i and \bar{x}_i .

(\Rightarrow) Let $\varphi : \{x_1, \dots, x_n\} \rightarrow \{0, 1\}$ be some assignment satisfying w clauses in ϕ , and for each i , $1 \leq i \leq n_\phi$, let C_i denote the clauses satisfied by $\varphi(x_i)$. We construct a valid solution to I_ϕ from left to right: Assuming we have already processed $\{C_1, \dots, C_{i-1}\}$, we append to our solution all clauses in C_i which have not been appended previously, concatenated by s^6 .

Note that since we only add clauses satisfied by $x_i = 0$ or $x_i = 1$, but never both, our solution is indeed a common subsequence of A_1 and A_2 . Furthermore, notice that every clause has exactly one occurrence in our solution, therefore it does not contain any of B_1, \dots, B_ℓ as a subsequence. Obviously, since w different

clauses are satisfied and the separator has $6(n_\phi - 1)$ occurrences, the length of our solution is $6(n_\phi - 1) + w$.

(\Leftarrow) Let S_ϕ be a solution to I_ϕ of length $6(n_\phi - 1) + w$. Since every X_i contains at most 6 letters, we can conclude that S_ϕ contains exactly $6(n_\phi - 1)$ occurrences of the padding-character s . Moreover, every clause has at most one occurrence in S_ϕ , by construction of the restriction strings B_1, \dots, B_ℓ . Now, since all padding characters are in S_ϕ , S_ϕ must contain exactly w different clauses, and for each of these clauses there is an i with the clause selected from X_i in A_1 and X'_i in A_2 . By construction of X_i and X'_i , clauses that are satisfied by $x_i = 0$ cannot be selected with clauses satisfied by $x_i = 1$. It follows that there exists an assignment to x_1, \dots, x_{n_ϕ} which satisfies w clauses of ϕ . \square

4 Exact Algorithms

In this section we present an exact algorithm for the RLCS problem. We show that the dynamic programming solution for the CLCS problem applies also for RLCS. We begin with the special case of two input strings and one restriction string, the variant we dub the *basic RLCS* problem. We then extend this algorithm to the general case of m input strings and ℓ restriction strings. We show that this generalization implies that RLCS is FPT when parameterized by the total length of the restriction strings.

For describing our dynamic-programming solution for basic RLCS, we define a dynamic-programming table DP , where the entry $DP[i_1, i_2 : j_1]$, for $(i_1, i_2) \in \{1 \dots, n_1\} \times \{1, \dots, n_2\}$ and $j_1 \in \{1, \dots, k_1\}$, will store the length of the LCS between $A_1[[i_1]]$ and $A_2[[i_2]]$ restricted by $B_1[[j_1]]$. The entry $DP[n_1, n_2 : k_1]$ will store the length of the LCS between A_1 and A_2 restricted by B_1 . The computation of $DP[i_1, i_2 : j_1]$ is given by the following recursion:

$$DP[i_1, i_2 : j_1] = \begin{cases} \max \begin{cases} DP[i_1 - 1, i_2 - 1 : j_1 - 1] + 1 \\ DP[i_1, i_2 - 1 : j_1] \\ DP_k[i_1 - 1, i_2 : j_1] \end{cases} & A_1[i_1] = A_2[i_2] = B_1[j_1], \\ DP[i_1 - 1, i_2 - 1 : j_1] + 1 & A_1[i_1] = A_2[i_2] \neq B_1[j_1], \\ \max \begin{cases} DP[i_1 - 1, i_2 : j_1] \\ DP[i_1, i_2 - 1 : j_1] \end{cases} & \text{otherwise.} \end{cases}$$

It is not difficult to see that the above recursion is correct. In particular, if $A_1[i_1]$ and $A_2[i_2]$ are not both equal to $B_1[j_1]$, then the recursion for $DP[i_1, i_2 : j_1]$ follows the standard recursion for pairwise LCS, since there is no danger in computing a solution which contains $B_1[1] \dots B_1[j_1]$. On the other hand, if $A_1[i_1] = A_2[i_2] = B_1[j_1]$, then a common subsequence of $A_1[[i_1]]$ and $A_2[[i_2]]$ ending with the letter $A_1[i_1]$ cannot contain $B_1[[j_1 - 1]]$ as a subsequence, and so its length must be equal to $DP[i - 1, i_2 - 1 : j - 1] + 1$.

We next extend the above recursion for the case of m comparison strings A_1, \dots, A_m and ℓ restriction strings B_1, \dots, B_ℓ . Again, we have a dynamic

programming table DP , indexed by tuples $(i_1, \dots, i_m) \in \{1, \dots, n_1\} \times \dots \times \{1, \dots, n_m\}$ and $(j_1, \dots, j_\ell) \in \{1, \dots, k_1\} \times \dots \times \{1, \dots, k_\ell\}$, where $DP[i_1, \dots, i_m : j_1, \dots, j_\ell]$ is equal to the length of the LCS between $A_1[[i_1]], \dots, A_m[[i_m]]$ restricted by $B_1[[j_1]], \dots, B_\ell[[j_\ell]]$.

If it is not the case that $A_1[i_1] = \dots = A_m[i_m]$, then the recursion for $DP[i_1, \dots, i_m : j_1, \dots, j_\ell]$ follows the standard recursion for LCS between m strings (*i.e.* the restriction strings can be ignored). If $A_1[i_1], \dots, A_m[i_m]$ all equal some letter σ , then we compute $DP[i_1, \dots, i_m : j_1, \dots, j_\ell]$ by:

$$DP[i_1, \dots, i_m : j_1, \dots, j_\ell] = \max \left\{ \begin{array}{l} \max \left\{ \begin{array}{l} DP[i_1 - 1, \dots, i_m : j_1, \dots, j_\ell] \\ \cdot \\ \cdot \\ DP[i_1, \dots, i_m - 1 : j_1, \dots, j_\ell] \end{array} \right. \\ DP[i_1 - 1, \dots, i_m - 1 : j_1^*, \dots, j_\ell^*] \end{array} \right.$$

Where for all $x \in \{1, \dots, m\}$, we set $j_x^* = j_x - 1$ if $B_x[j_x] = \sigma$, and otherwise $j_x^* = j_x$.

Correctness of this recursion follows from the same arguments used for the recursion for basic RLCS. Thus, since the dynamic programming table DP has $O(n^{m+\ell})$ entries, with each entry computable in constant time, we get:

Lemma 2. *RLCS can be solved in $O(n^{m+\ell})$ time.*

Let k denote the total length of the restriction strings, *i.e.* $k = \sum_{j=1}^{\ell} k_j$. Observe that the number of entries in DP can also be bounded by $O(2^k n^m)$, since the number of prefixes of the restriction strings cannot exceed 2^k . Thus, we have:

Lemma 3. *RLCS is in FPT when parameterized by the total length of the restriction strings.*

5 Approximation Algorithms

We next present two approximation algorithms for the RLCS problem. The first algorithm provides a $\frac{1}{|\Sigma|}$ approximation ratio for the case of both arbitrary number of input strings and arbitrary number of restrictions. Here Σ is the set of letters used in the instance, *i.e.* the actual alphabet of the comparison and restriction strings. Afterwards, we present an $\frac{k_{min}-1}{n_{min}}$ -approximation algorithm, where n_{min} and k_{min} are the lengths of the shortest input string and the shortest restriction, respectively. This algorithm is relevant only for the case of fixed number input strings (and arbitrary number of restrictions). Both algorithms are very simple. This situation should be compared with CLCS, where in general no approximation can be sought unless $P=NP$, and for the case of a single constraint string only a ratio of $\frac{1}{\sqrt{n_{min}|\Sigma|}}$ is known.

Algorithm I:

1. For every $s \in \Sigma$ and every $i \in \{1, \dots, m\}$, compute the number of occurrences $Occ_i(s)$ of s in A_i .
2. For every $s \in \Sigma$ compute:
 - $Occ(s)$, the minimum between $Occ_1(s), \dots, Occ_m(s)$.
 - $Cons(s)$, the length of the shortest restriction string that does not contain any symbol besides s . If no such restriction string exists, $Cons(s) = \infty$.
 - $Val(s)$, the minimum between $Occ(s)$ and $Cons(s)$.
3. Find $s \in \Sigma$ with maximal $Val(s)$ and return $s^{Val(s)}$.

The following lemma is easily established:

Lemma 4. *There is no restricted common subsequence of A_1, \dots, A_m that contains more than $Val(s)$ occurrences of any $s \in \Sigma$, and thus Algorithm I returns a $\frac{1}{|\Sigma|}$ -approximate solution.*

Our second algorithm is even simpler than the first:

Algorithm II:

1. Compute the LCS S of A_1, \dots, A_m .
2. Return the prefix of length $k_{min} - 1$ of S .

Lemma 5. *If the LCS of A_1, \dots, A_m is shorter than n_{min} , then Algorithm II finds an optimal RLCS. Otherwise, it outputs an RLCS of length $k_{min} - 1$ which yields an approximation ratio of $\frac{k_{min}-1}{n_{min}}$.*

References

1. A. V. Aho, D. S. Hirschberg and J. D. Ullman. Bounds on the Complexity of the Longest Common Subsequence Problem. *Journal of the ACM*, 23(1): 1-12, 1976.
2. A. Amir, T. Hartman, O. Kapah, B. R. Shalom and D. Tsur. Generalized LCS. *Proc. SPIRE 2007*, 50-61, 2007.
3. A. Apostolico, G. M. Landau and S. Skiena. Matching for Run-Length Encoded Strings. *Journal of Complexity*, 15(1): 4-16, 1999.
4. A. N. Arslan and Ö. Egecioglu. Algorithms For The Constrained Longest Common Subsequence Problems. *International Journal of Foundations of Computer Science*, 16(6): 1099-1109, 2005.
5. L. Bergroth, H. Hakonen and T. Raita. A Survey of Longest Common Subsequence Algorithms. *Proc. SPIRE 2000*, 39-48, 2000.
6. P. Berman and M. Karpinski. On Some Tighter Inapproximability Results. *Electronic Colloquium on Computational Complexity*, 5(29): 1998.
7. F. Y. L. Chin, A. De Santis, A. L. Ferrara, N. L. Ho and S. K. Kim. A simple algorithm for the constrained sequence problems. *Information Processing Letters*, 90(4): 175-179, 2004.
8. Z. Gotthilf, D. Hermelin and M. Lewenstein: Constrained LCS: Hardness and Approximation. *Proc. CPM 2008*, 255-262, 2008.

9. Z. Gotthilf and M. Lewenstein: Approximating Constrained LCS. *Proc. SPIRE 2007*, 164-172, 2007.
10. D. S. Hirschberg. A Linear Space Algorithm for Computing Maximal Common Subsequences. *Communications of the ACM*, 18(6): 341-343, 1975.
11. D. S. Hirschberg. Algorithms for the Longest Common Subsequence Problem. *Journal of the ACM*, 24(4): 664-675, 1977.
12. D. Maier. The Complexity of Some Problems on Subsequences and Supersequences. *Journal of the ACM*, 25(2): 322-336, 1978.
13. W. J. Masek and M. Paterson. A Faster Algorithm Computing String Edit Distances. *Journal of Computer and System Sciences*, 20(1): 18-31, 1980.
14. Y.-T. Tsai. The constrained longest common subsequence problem. *Information Processing Letters*, 88(4): 173-176, 2003.