Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

On the longest common parameterized subsequence

Orgad Keller, Tsvi Kopelowitz, Moshe Lewenstein [*]

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

ABSTRACT

The well-known problem of the longest common subsequence (LCS), of two strings of lengths n and m respectively, is $O(nm)$-time solvable and is a classical distance measure for strings. Another well-studied string comparison measure is that of parameterized matching, where two equal-length strings are a parameterized match if there exists a bijection on the alphabets such that one string matches the other under the bijection. All works associated with parameterized pattern matching present polynomial time algorithms.

There have been several attempts to accommodate parameterized matching along with other distance measures, as these turn out to be natural problems, e.g., Hamming distance, and a bounded version of edit-distance. Several algorithms have been proposed for these problems.

In this paper we consider the longest common parameterized subsequence problem which combines the LCS measure with parameterized matching. We prove that the problem is NP-hard, and then show a couple of approximation algorithms for the problem.

1. Introduction

The problem of finding the *longest common subsequence*, denoted as LCS, of two given strings is one of the classical and well-studied problems in the area of algorithms: given two strings B and C of lengths n and m respectively (throughout this paper we will assume $n \geq m$), we wish to find the longest string that is a subsequence of both B and C.

For apparent reasons, LCS is one of the most natural measures used to test the *similarity* between two strings. While this problem and its variants are interesting theoretically, they are of fundamental practical use in the areas of molecular biology and code analysis, e.g., where one wishes to test the differences between two programming language code fragments. To name only one, the well-known UNIX *diff* command applies LCS as its main tool.

The classic and well-known solution of Wagner and Fischer [19] uses dynamic programming to solve the problem in time $O(nm)$. It can be generalized to solve LCS for any *fixed* number of input strings in polynomial time. Masek and Paterson [16] improved the running time of the case where $n = m$ to $O(n^2/\log n)$, by using the "*four Russians*" technique. Other solutions – e.g., [10,18,17] – in which the running time of the solutions are dependent on different parameters besides the length of the strings, have also been provided.

While, as mentioned, the problem for any fixed number of strings can be solved in polynomial time, Maier [15] showed that LCS on an *arbitrary* number of strings is NP-hard (by applying a reduction from *vertex cover*), and later Jiang and Li [11] showed that there exists a constant $\delta > 0$ for which there is no n^{δ}-approximation algorithm for the problem, unless P = NP. Note that when the number of input strings is fixed to be 2, almost all LCS variants can be solved in polynomial time.

Another very important and interesting model for testing similarity between strings, introduced by Baker [2–5], is called *parameterized matching*, or *p-match* in short. In this model, two length-n input strings are said to p-match if (roughly, and

[*] Corresponding author.
E-mail addresses: kellero@cs.biu.ac.il (O. Keller), kopelot@cs.biu.ac.il (T. Kopelowitz), moshe@cs.biu.ac.il (M. Lewenstein).

0304-3975/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2009.09.011

will be detailed later) there exists a bijection on the alphabet symbols which maps the ith symbol of the first string to the ith symbol of the second. As the symbols of the alphabet can be, for example, programming language code tokens, this model has practical importance in testing whether two code segments are essentially the same, even when some tokens (e.g., variable names) have been globally renamed.

In parameterized pattern matching, we get a length-n text and a length-m pattern and wish to report all locations i in the text where the pattern p-matches the length-m text substring starting at location i. Extensive amount of work has been done on this problem: Amir et al. [1] showed an efficient algorithm even when the alphabet size is $O(n)$, which runs in worst-case $O(n \log \sigma)$ time, where σ is the size of the parametric alphabet. In [8] they showed how to efficiently provide an approximate solution, and in [9] they generalized the problem for the 2-dimensional case. Work was done by [13,3,4,6] to provide *parameterized text indexing* by showing how to efficiently construct a *parameterized suffix tree*. Finally, Ferragina and Grossi [7] showed how to provide for efficient parameterized text indexing even in external memory.

In parameterized pattern matching, we benefit from two facts: the first, that in each match, *consecutive* symbols of the text are compared against the *consecutive* symbols of the pattern, and the second, that in two locations where the pattern matches the text, the corresponding bijections need not be the same. It is very natural and tempting to solve the problem without using these conditions to aid us; by this, we adapt the p-match model to the LCS problem, thus defining the LCPS problem discussed in this paper. Such a setting would be very practical in the case where, for example, two code fragments – an original, and a suspected copy – are being tested for similarity after the alleged copy has been edited, besides possibly having its variable names changed. Unfortunately, we show that this problem is NP-hard. We prove this by a reduction from the problem of 3D-matching in hypergraphs [12], and then provide a couple of approximation algorithms, which yield a $\lambda\sqrt{|\text{OPT}|}$-length solution for any constant λ, where OPT is the optimal solution.

A note must be made about the similarity between LCS and *edit-distance* [14]: testing the similarity of two strings via LCS is the equivalent of doing so using edit-distance when the edit operations allowed are only insertions and deletions. Baker [5] discusses the notion of *parameterized edit-distance*, in which the operations allowed are insertions, deletions, and p-matches, where the p-match edit operation replaces a *substring* in the first string with a substring that p-matches it which appears in the second. Therefore, the aiding conditions of parameterized pattern matching still play a role there.

The rest of this paper is organized as follows: in Section 2 we provide the formal definitions of our problems and some preliminaries, including a naïve algorithm for the specific case where the parametric alphabet is small. In Section 3 we prove that the LCPS problem is NP-hard. In Sections 4 and 5 we provide an approximation for a specific case of the problem called *LCMS*, and for the general LCPS, respectively.

2. Problem definitions and preliminaries

Let $S = s_1 \dots s_n$ and $T = t_1 \dots t_n$ be strings over alphabet set $\Sigma \cup \Pi$, such that $\Sigma \cap \Pi = \emptyset$. Following the notations used in [1], we say S and T are a *parameterized match* (*p-match* for short) if there exists a bijection $f : \Pi \to \Pi$ for which, for each $i = 1, \dots, n$, it holds that:

1. if $s_i \in \Sigma$, then $s_i = t_i$.
2. if $s_i \in \Pi$, then also $t_i \in \Pi$, and $f(s_i) = t_i$.

For two strings $B = b_1 \dots b_n$ and $C = c_1 \dots c_m$ over $\Sigma \cup \Pi$, We define their *common parameterized subsequence* (CPS for short) as a pair of two ascending sequences $I = \langle i_1, \dots, i_k \rangle$ and $J = \langle j_1, \dots, j_k \rangle$ of locations in B and C respectively (i.e., $i_\ell \in \{1, \dots, n\}$ and $j_\ell \in \{1, \dots, m\}$ for each $\ell = 1, \dots, k$), such that B^I p-matches C^J, where $B^I = b_{i_1} b_{i_2} \dots b_{i_k}$ and $C^J = c_{j_1} c_{j_2} \dots c_{j_k}$.

The *longest common parameterized subsequence problem* is defined as follows:

Input: Two strings $B = b_1 \dots b_n$ and $C = c_1 \dots c_m$ over alphabet set $\Sigma \cup \Pi$, such that $\Sigma \cap \Pi = \emptyset$.
Output: A CPS of maximal length, denoted LCPS.

By CPS we will also denote the decision version of the problem, in which we ask whether two strings have a common parameterized subsequence of a specified length. The meaning will be clear from the context.

The specific case of the LCPS problem in which $\Sigma = \emptyset$ (i.e., the only alphabet is the parametric alphabet Π) is denoted the *longest common mapped subsequence* (*LCMS*) problem.

Let $\mathcal{A}$ be an algorithm (exact or approximate) for the LCPS problem. $\mathcal{A}(B, C)$ returns a pair (I, J) of sequences of indices in B and C respectively. Denote $I = \langle i_1, \dots, i_k \rangle$ and $J = \langle j_1, \dots, j_k \rangle$. We define the length of the solution $|(I, J)| = |I| = |J| = k$ and denote $|\mathcal{A}(B, C)| = |(I, J)|$.

Solving the problem for asymptotically-small Π

Theorem 1. *There exists an algorithm $\mathcal{N}$ for the LCPS problem, which solves the problem in $O(|\Pi|! \cdot nm)$ time.*

Proof. We propose the following "naïve" brute force algorithm: for each possible bijection $f : \Pi \to \Pi$, construct a new string B_f by replacing each symbol $b_i \in \Pi$ in B with $f(b_i)$, and find LCS(B_f, C) using [19]. Finally, choose the bijection f for which LCS(B_f, C) gave a maximal-length result, and recover its corresponding indices in B_f (and hence, in B) and in C. Since there are $|\Pi|!$ possible bijections from Π to Π, and [19] runs in time $O(nm)$, the running time is $O(|\Pi|! \cdot nm)$. □

Corollary 1. *If $|\Pi| = c$ for some constant c, then the LCPS problem can be solved in time $O(nm)$.*

Corollary 2. *Assume w.l.o.g. that $n \geq m$ and let c be a constant. If $|\Pi| \leq \frac{c \log n}{\log\log n}$, then the LCPS problem can be solved in time*

$$O((c \log n/ \log\log n)! \cdot nm) = O\left(2^{\frac{c\log n}{\log\log n}\log\left(\frac{c\log n}{\log\log n}\right)}nm\right) = O(n^{c+1}m)\ . \tag{1}$$

Remark 1. Note that $\mathcal{N}$ also trivially solves the LCMS problem, and therefore will be used as such later.

3. Finding the LCPS of two strings is NP-hard

We define the decision version of the LCPS problem: for two strings B and C and an integer t, we say $(B, C, t) \in$ CPS if there exists a solution (I, J) for LCPS(B, C) such that $|(I, J)| \geq t$.

Sequence graphs. A convenient way of describing the CPS restrictions is by defining the *sequence graph*: given the input strings B and C and two sequences $\langle i_1, \dots, i_k\rangle$ and $\langle j_1, \dots, j_k\rangle$ of locations in B and C respectively, a *sequence graph* is a directed planar graph $G = (V, E)$ in which the vertex set V is the set of location-specific characters of B and C, set on a grid in the following manner:

1. for each $i = 1, \dots, n$, b_i is set at grid location $(i, 1)$;
2. for each $j = 1, \dots, m$, c_j is set at grid location $(j, 0)$;

and E is defined such that there is an edge from $(i_\ell, 1)$ to $(j_\ell, 0)$ for each $\ell = 1, \dots, k$. Formally: $E = \{((i_\ell, 1), (j_\ell, 0)) \mid \ell = 1, \dots, k\}$.

Remark 2. For convenience, when we refer to some edge written as "(b_i, c_j)" or described as "the edge mapping b_i to c_j", we mean the specific edge from grid-point $(i, 1)$ to grid-point $(j, 0)$ (if such exists), and not to any other edge whose endpoints are two other grid-points labeled with the symbol b_i and the symbol c_j, respectively, which might also exist in the graph.

If a sequence graph contains some edge (b_i, c_j), we say b_i is *mapped* to c_j. Two different edges (b_i, c_j), $(b_{i'}, c_{j'})$ are said to be *intersecting* if the straight line on the plane connecting grid-point $(i, 1)$ to grid-point $(j, 0)$ (which corresponds to (b_i, c_j)) crosses the straight line connecting $(i', 1)$ to $(j', 0)$ (which corresponds to $(b_{i'}, c_{j'})$). Alternatively: if $i' \geq i$, but $j' \leq j$.

Observation 1. If the sequences $\langle i_1, \dots, i_k\rangle$ and $\langle j_1, \dots, j_k\rangle$ are both *ascending*, then the sequence graph does not contain intersecting edges.

A sequence graph is said to be a *CPS graph* if it corresponds to some CPS, i.e., to two sequences $\langle i_1, \dots, i_k\rangle$ and $\langle j_1, \dots, j_k\rangle$ which comply with the conditions described in Section 2. Notice that there is always a one-to-one correspondence between a CPS of two strings and a CPS graph.

Let X, Y, Z be three disjoint sets such that $|X| = |Y| = |Z| = n$, and let $S \subseteq X \times Y \times Z$. In the 3*D-matching* problem [12], we wish to find a subset $S' \subseteq S$ which is a *perfect matching* of X, Y, and Z, i.e., *every* element of X, Y, and Z is covered by S' exactly *once*. In the problem's decision version, denoted 3DM, when given (X, Y, Z, S), we say $(X, Y, Z, S) \in$ 3DM if there exists such a perfect matching $S' \subseteq S$. Notice that we can always assume $n < |S| < n^3$, otherwise solving the problem is trivial.

Theorem 2. *LCPS is NP-hard.*

Proof. We show that LCPS is NP-hard (or rather, that CPS $\in$ NPC) using a reduction from 3DM:

The reduction. Given the input-tuple (X, Y, Z, S) for the 3DM problem, where $|X| = |Y| = |Z| = n$ (note that in this section n denotes the size of X, Y, and Z) and $S = \{t_1, \dots, t_s\} \subseteq X \times Y \times Z$, we choose $\Sigma = \emptyset$ and $\Pi = X \cup S \cup \{*\}$. In order to construct the reduction strings properly, we first require some notation: for a specific tuple $t_i = (x, y, z)$, we denote $x(t_i) = x$, $y(t_i) = y$ and $z(t_i) = z$. For some fixed $y_i \in Y$, we define $S(y_i) = \{(x, y, z) \in S \mid y = y_i\}$, i.e., $S(y_i)$ is the set of all tuples in S having y_i as their y-coordinate. Denote $s(y_i) = |S(y_i)|$. Furthermore, assume $S(y_i) = \{t_{r_1}, \dots, t_{r_{s(y_i)}}\}$, where the sequence $\langle t_{r_1}, \dots, t_{r_{s(y_i)}}\rangle$ is $S(y_i)$ sorted in ascending order of x-coordinates. We define the blocks

$$\mathcal{B}^B_{y_i} = x(t_{r_1})x(t_{r_2}) \dots x(t_{r_{s(y_i)-1}})x(t_{r_{s(y_i)}}), \tag{2}$$

and

$$\mathcal{B}^C_{y_i} = t_{r_{s(y_i)}}t_{r_{s(y_i)-1}} \dots t_{r_2}t_{r_1}. \tag{3}$$

In other words, in $\mathcal{B}^B_{y_i}$ we list the x-coordinates of the tuples in an *ascending* order, and in $\mathcal{B}^C_{y_i}$ we list the tuples themselves (each tuple serves as a single character), only this time, in the *descending* order of their respective x-coordinates. As we shall see later, the role of $\mathcal{B}^B_{y_i}$ and $\mathcal{B}^C_{y_i}$ will be to assure that no two tuples which share the same y-coordinate value will be included in S', i.e., each y_i will be covered at most once by a tuple in S'. Finally, we define $\mathcal{B}^B_{z_i}$ and $\mathcal{B}^C_{z_i}$, using the same principle, only this time for the z-coordinates.

We now move to construct the strings, each comprised of three segments:

$$B = \overbrace{x(t_1) *^3 \ldots *^3 x(t_s)*^3}^{\text{Seg. 1}} \overbrace{\mathcal{B}^B_{y_1} *^3 \ldots *^3 \mathcal{B}^B_{y_n} *^3}^{\text{Seg. 2}} \overbrace{\mathcal{B}^B_{z_1} *^3 \ldots *^3 \mathcal{B}^B_{z_n} *^3}^{\text{Seg. 3}}, \tag{4}$$

and

$$C = \underbrace{t_1 *^3 \ldots *^3 t_s *^3}_{\text{Seg. 1}} \underbrace{\mathcal{B}^C_{y_1} *^3 \ldots *^3 \mathcal{B}^C_{y_n} *^3}_{\text{Seg. 2}} \underbrace{\mathcal{B}^C_{z_1} *^3 \ldots *^3 \mathcal{B}^C_{z_n} *^3}_{\text{Seg. 3}}. \tag{5}$$

Notice that each of the strings contains $s + 2n$ blocks of $*$ symbols – each block is of length 3 – and $3s$ non-$*$ symbols (since each tuple appears exactly once in each segment of C, and for each such single appearance, the tuple's x-coordinate appears once in the respective segment of B). We derive that $|B| = |C| = 3(s + 2n) + 3s = 6s + 6n$. Finally, we choose $t = 3s + 9n < 6s + 6n$.

Before showing that this reduction is correct, we require some definitions: for some sequence graph of B and C, we define an *$(*, *)$-type edge* as an edge whose endpoints are both $*$ symbols, and an *(x, t)-type edge* as an edge whose endpoint in B is some x-coordinate value, and whose endpoint in C is some tuple. Likewise we define an *$(x, *)$-type edge* and an *$(*, t)$-type edge*. We continue to the following claim:

Claim 1. *Assume a CPS of B and C is given, and is of length $3s + 9n$, and let f be its corresponding bijection. Then the following statements apply to the corresponding CPS graph and bijection f:*

1. $f(*) = *$.
2. *There are exactly $3n$ (x, t)-type edges, and exactly $3s + 6n$ $(*, *)$-type edges.*
3. *Every $*$ at some location i in B is mapped to its respective $*$ at location i in C.*
4. *Each segment of B contributes exactly n (x, t)-type edges. In particular, Segment 1 of B contributes n (x, t)-type edges, all of them vertical.*
5. *Each $\mathcal{B}^B_{y_i}$ (resp. $\mathcal{B}^B_{z_j}$) block contributes exactly one edge, to a symbol in $\mathcal{B}^C_{y_i}$ (resp. $\mathcal{B}^C_{z_j}$).*

Proof. We prove each item using the previous ones:

1. Assume by contradiction that $f(*) \neq *$. In this case (as shown by a very loose analysis), there are (a) at most 3 $(*, t)$-type edges (since each unique tuple t_i appears at most 3 times in C, one in each segment), (b) at most $3s$ $(x, *)$-type edges (since a unique x-coordinate appears at most s times in each segment of B), and (c) at most $3n$ (x, t)-type edges (since there are n distinct x-coordinates, each of them may be mapped to a tuple, and each unique tuple appears 3 times in C). We derive that this scenario gives us at most $3 + 3s + 3n < 3s + 9n$ edges, which contradicts the fact that the LCPS is of length $3s + 9n$. We conclude that indeed, $f(*) = *$.
2. From the last item it follows that each x-coordinate x_i is mapped by f to some tuple t_j. Since each unique tuple appears exactly 3 times in C, and there are n distinct x-coordinates, then there are at most $3n$ (x, t)-type edges. Now, since the number of $(*, *)$-type edges is bounded by $3(s+2n)$ (the number of $*$ symbols in each string), we conclude that in order to reach length $3s + 9n$, we require the number of $(*, *)$-type edges to be exactly $3(s + 2n) = 3s + 6n$, and the number of (x, t)-type edges to be exactly $3n$.
3. Since the number of $(*, *)$-type edges is $3s + 6n$, and no two edges can intersect each other (since it is a CPS graph), the only way to obtain this number of edges is by mapping *every* $*$ at some location i in B to the $*$ at the respective location i in C.
4. First of all, notice that an (x, t)-type edge emanating from a specific segment in B cannot go to other than its respective segment in C, otherwise it would result in the loss of $(*, *)$-type edges, which would contradict Item 3. In each segment of B, there are n distinct x-coordinates. In each segment of C, each unique tuple appears once. Therefore, each segment can contribute at most n (x, t)-type edges, and must contribute exactly n of those, otherwise we would not reach the target length. Finally, each non-vertical (x, t)-type edge emanating from Segment 1 of B would result in the loss of $(*, *)$-type edges. We conclude all (x, t)-type edges in Segment 1 of B are vertical and therefore go to symbols in Segment 1 of C.
5. First notice that an edge emanating from some block $\mathcal{B}^B_{y_i}$ cannot go to other than the block $\mathcal{B}^C_{y_i}$; the opposite would result in losing $(*, *)$-type edges. We proceed to show that there is at most a single edge from each block. Assume by contradiction that there are two edges from $\mathcal{B}^B_{y_i}$ to $\mathcal{B}^C_{y_i}$, and let them be (x_a, t_c) and (x_b, t_d). Assume w.l.o.g. that x_b appears right of x_a in $\mathcal{B}^B_{y_i}$. Since a unique tuple can appear at most once in $\mathcal{B}^C_{y_i}$, then obviously $t_c \neq t_d$. It follows that also $x_a \neq x_b$ (since f is a proper function). Notice that $x(t_c) = x_a$ and $x(t_d) = x_b$ (in words, both edges must be from an x-value to a tuple having this value as its x-coordinate), otherwise we would lose one of the n vertical (x, t)-type edges in Segment 1, which always map a value to a tuple having it as its x-coordinate. However, since x_b appears *right* of x_a and $x_a \neq x_b$, it follows that in $\mathcal{B}^C_{y_i}$, the tuples for which x_b is the x-coordinate appear *left* of the tuples for which x_a is the x-coordinate. In particular, t_d is left of t_c in $\mathcal{B}^C_{y_i}$. We conclude that the two edges intersect, which contradicts the fact that this is a CPS graph. The proof for $\mathcal{B}^B_{z_j}$ and $\mathcal{B}^C_{z_j}$ is similar. We have just proved that each $\mathcal{B}^B_{y_i}$ (resp. $\mathcal{B}^B_{z_j}$) block contributes at most a single edge, but since we require n edges from Segment 2 (resp. Segment 3) in order to obtain the target length, we conclude that each such block contributes exactly one edge. $\square$

It remains to show that the reduction described is correct:

Claim 2. $(X, Y, Z, S) \in$ 3DM *if and only if* $(B, C, 3s + 9n) \in$ CPS.

Proof. We prove both directions:

(only if) Given a subset $S' \subseteq S$, $|S'| = n$, which covers each element of X, Y, or Z exactly once (i.e., S' is a *perfect matching*), we determine the respective I, J sequences by describing a CPS graph: for each $i = 1, \ldots, 6s + 6n$:

1. If $b_i = c_i = *$, then map b_i to c_i.
2. Otherwise, c_i is some tuple in S. If it also holds that $c_i \in S'$, then:
   (a) If i is a location in Segment 1, map b_i to c_i.
   (b) If i is a location in Segment 2, then c_i appears as a symbol in the block $\mathcal{B}^C_{y(c_i)}$, and therefore $x(c_i)$ appears as a symbol b_j in $\mathcal{B}^B_{y(c_i)}$. Therefore, map b_j to c_i.
   (c) If i is a location in Segment 3, the argument is similar, only this time with $\mathcal{B}^C_{z(c_i)}$ and $\mathcal{B}^B_{z(c_i)}$ respectively.

Claim 3. *The above scheme yields a CPS graph and therefore a CPS of length* $3s + 9n$.

Proof. First notice that the mappings of the form $(*, *)$ actually define that $f(*) = *$ and contribute $3s + 6n$ edges. Since they are all vertical, they do not intersect with each other. Since all other edges in Segment 1 are also vertical (i.e., are of the form (b_i, c_i)), they do not intersect with the above edges or each other. In addition, since S' is a matching, each unique x_i value is mapped to a unique tuple denoted $t(x_i)$ having x_i it as its x-coordinate value. Hence it defines by this that $f(x_i) = t(x_i)$ for $i = 1, \ldots, n$. Since $|S'| = n$, we conclude that this has contributed another n edges. Finally, at each $\mathcal{B}^B_{y_i}$ block, we make a *single* mapping to a value in $\mathcal{B}^C_{y_i}$ (because S' is a matching, and all tuples in $\mathcal{B}^C_{y_i}$ share the same y-coordinate, and in addition a unique tuple can appear at most once in $\mathcal{B}^C_{y_i}$). Notice that mappings in these blocks are consistent with mappings in Segment 1, and therefore agree with the definition of f made before. The argument for $\mathcal{B}^B_{z_j}$ and $\mathcal{B}^C_{z_j}$ is similar. Finally, since each $\mathcal{B}^B_{y_i}$ or $\mathcal{B}^B_{z_j}$ block contributes a single edge, we conclude that those blocks contributed $2n$ edges all together, none of them intersects with other edges. It follows that the constructed graph is a CPS graph with $3s + 9n$ edges and therefore the claim follows. □

We thus conclude that $(B, C, 3s + 9n) \in$ CPS.

(if) Assume that $(B, C, 3s + 9n) \in$ CPS, i.e., B, C have a common parameterized subsequence of length $3s + 9n$, and consider the corresponding CPS graph and the bijection f. By Item 4 of the first claim, each (x, t)-type edge in Segment 1 is vertical and therefore agrees with the mapping of each unique x_i to a *unique* tuple t_j for which $x(t_j) = x_i$. Define $S' = \{t_j \mid \exists x_i, \quad f(x_i) = t_j\}$. Since all tuples sharing the same y-coordinate (resp. z-coordinate) appear in the same $\mathcal{B}^C_y$ (resp. $\mathcal{B}^C_z$) block, and by Item 5 such block contributes a single edge (which agrees with the mappings defined by the edges in Segment 1, since f is a bijection), we conclude each unique y-coordinate (resp. z-coordinate) is covered, and furthermore covered exactly once by S'. We conclude that S' is a perfect matching and therefore $(X, Y, Z, S) \in$ 3DM. □

3DM $\in$ NPC, CPS is trivially in NP, and the above reduction clearly can be performed in polynomial time. We therefore conclude CPS $\in$ NPC. Therefore if LCPS admits a polynomial time algorithm, then P = NP. □

4. Approximating LCMS

Recall that LCMS is the specific case of the LCPS problem where $\Sigma = \emptyset$. For a given parameter $\lambda > 0$, we provide an $O(n^{2\lambda^2+1}m)$-time algorithm, $\mathcal{A}^{\text{LCMS}}_\lambda$, for which, for two strings B and C of lengths n and m respectively, $|\mathcal{A}^{\text{LCMS}}_\lambda(B, C)| \geq \lambda\sqrt{|\text{OPT}(B, C)|}$, where OPT$(B, C)$ denotes the optimal solution.

First, some notation: for a string S, let $\Pi_S = \{a \in \Pi \mid a \text{ appears in } S\}$. Given some alphabet set $\Gamma \subseteq \Sigma \cup \Pi$, we denote by S_Γ the string S with all symbols not from Γ deleted, while, for symbols not deleted, preserving their original location in S. In other words, we keep aside each symbol in S_Γ its original location in S. We will refer to this location as the symbol's *effective* location. For our two strings B and C, let $\pi_{\min} = \min\{|\Pi_B|, |\Pi_C|\}$. Finally, let OPT$(B, C) = (I^*, J^*)$ be the optimal solution, and let $I^* = \langle i^*_1, \ldots, i^*_t\rangle$ and $J^* = \langle j^*_1, \ldots, j^*_t\rangle$. We define π^* to be the number of distinct symbols which appear in B^{I^*} (equivalently, in C^{J^*}; by the problem properties, it is the same).

Our algorithm $\mathcal{A}^{\text{LCMS}}_\lambda$ utilizes the fact that two strategies for the LCMS problem are available: for the first, notice that both $|\Pi_B| \geq \pi_{\min}$ and $|\Pi_C| \geq \pi_{\min}$ by the definition of $\pi_{\min}$. We can therefore create sequences I, J for which $|I| = |J| = \pi_{\min}$, by mapping the ℓth unique symbol which appears in B, to the ℓth unique symbol which appears in C, for $\ell = 1, \ldots, \pi_{\min}$. For the second strategy, assume we know the λ^2 symbols most frequent in B^{I^*}, and the λ^2 symbols most frequent in C^{J^*}. Running the naïve algorithm on the two strings, wherein all symbols not from the λ^2 most frequent are deleted, will yield a solution of length at least $\frac{|\text{OPT}(B,C)|}{\pi^*/\lambda^2}$ (since if we partition $\Pi_{B^{I^*}}$ to π^*/λ^2 sets, each of size λ^2, one of them must give us length of at least $\frac{|B^{I^*}|}{\pi^*/\lambda^2} = \frac{|\text{OPT}(B,C)|}{\pi^*/\lambda^2}$ when running the naïve algorithm on the strings induced by its symbols only). Since we do not know $\Pi_{B^{I^*}}$, we test every possible combination of λ^2 symbols in both strings and choose the combination yielding the maximal result. Finally, our approximation algorithm chooses the better of the two strategies.

4.1. Analysis

Theorem 3. *Given a parameter $\lambda > 0$, $\mathcal{A}_\lambda^{\text{LCMS}}$ is an $O(n^{2\lambda^2+1}m)$-time approximation algorithm for LCMS, such that $|\mathcal{A}_\lambda^{\text{LCMS}}(B, C)| \geq \lambda\sqrt{|\text{OPT}(B, C)|}$.*

Proof. We provide the approximation factor and the running-time analysis:

Approximation. From the discussion above, the algorithm returns sequences of length $\max\{\pi_{\min}, \frac{|\text{OPT}(B,C)|}{\pi^*/\lambda^2}\}$. Notice that:

$$\begin{aligned}
\lambda^2|\text{OPT}(B, C)| &= \pi^* \cdot \frac{|\text{OPT}(B, C)|}{\pi^*/\lambda^2} && (6)\\
&\leq \min\{|\Pi_B|, |\Pi_C|\} \cdot \frac{|\text{OPT}(B, C)|}{\pi^*/\lambda^2} && (7)\\
&= \pi_{\min} \cdot \frac{|\text{OPT}(B, C)|}{\pi^*/\lambda^2}, && (8)
\end{aligned}$$

where (7) is true because π^* is bounded by $\min\{|\Pi_B|, |\Pi_C|\}$ and (8) is true by definition. We therefore conclude that $\max\{\pi_{\min}, \frac{|\text{OPT}(B,C)|}{\pi^*/\lambda^2}\} \geq \lambda\sqrt{|\text{OPT}(B, C)|}$. Since $|\mathcal{A}_\lambda^{\text{LCMS}}(B, C)| = \max\{\pi_{\min}, \frac{|\text{OPT}(B,C)|}{\pi^*/\lambda^2}\}$, the approximation factor follows.

Running-time. Computing the sequences of size $pi_{\min}$ can be done efficiently by sorting both strings according to the symbols of the alphabet. Computing the second strategy can be efficiently executed by (a) leaving only one copy of each unique symbol in the two sorted strings, and (b) re-sort the sorted strings, this time using the indices as the keys by which the sorting is done. Since there are $\binom{|\Pi_B|}{\lambda^2} \leq n^{\lambda^2}$ options for Π', and $\binom{|\Pi_C|}{\lambda^2} \leq n^{\lambda^2}$ options for Π'', and running the naïve algorithm costs $O(nm)$, we conclude that the running-time is bounded by $O(n^{\lambda^2} \cdot n^{\lambda^2} \cdot nm) = O(n^{2\lambda^2+1}m)$. □

5. Approximating LCPS

For a given parameter $\lambda > 0$, we provide an $O(n^{4\lambda^2+1}m)$-time algorithm, $\mathcal{A}_\lambda^{\text{LCPS}}$, for which, for two strings B and C of lengths n and m respectively, $|\mathcal{A}_\lambda^{\text{LCPS}}(B, C)| \geq \min\{\lambda\sqrt{|\text{OPT}(B, C)|}, \frac{1}{2}|\text{OPT}(B, C)|\}$.

Note that almost all notation remains the same, except that this time, (I^*, J^*) is the solution returned by $\text{OPT}(B_\Pi, C_\Pi)$ (instead of $\text{OPT}(B, C)$, as before). Again, $I^* = \langle i_1^*, \ldots, i_t^*\rangle$ and $J^* = \langle j_1^*, \ldots, j_t^*\rangle$. π^* is defined as before to be the number of distinct symbols which appear in B^{I^*} (or equivalently, in C^{J^*}).

Our algorithm $\mathcal{A}_\lambda^{\text{LCPS}}$ utilizes the fact that this time *three* strategies for the LCPS problem are available: while the first two remain the same as before – and thus, actually work now on B_Π and C_Π – the third corresponds to B_Σ and C_Σ: we can simply run the ordinary LCS algorithm on B_Σ and C_Σ, thus obtaining a legal CPS. As before, our approximation algorithm $\mathcal{A}_\lambda^{\text{LCPS}}$ will choose the best of the three.

5.1. Analysis

Theorem 4. *Given a parameter $\lambda > 0$, $\mathcal{A}_\lambda^{\text{LCPS}}$ is an $O(n^{4\lambda^2+1}m)$-time approximation algorithm for LCPS, such that*

$$|\mathcal{A}_\lambda^{\text{LCPS}}(B, C)| \geq \min\left\{\lambda\sqrt{|\text{OPT}(B, C)|}, \frac{1}{2}|\text{OPT}(B, C)|\right\}.$$

Proof. We provide the approximation factor and the running-time analysis:

Approximation. $\sqrt{2}\lambda$ is used as the parameter when running $\mathcal{A}^{\text{LCMS}}$ on B_Π and C_Π, and therefore $\mathcal{A}_{\sqrt{2}\lambda}^{\text{LCMS}}$ returned a $\max\{\pi_{\min}, \frac{|\text{OPT}(B_\Pi,C_\Pi)|}{\pi^*/2\lambda^2}\}$-length solution. It follows that the entire $\mathcal{A}_\lambda^{\text{LCPS}}$ algorithm returned a solution of length $\max\{\pi_{\min}, \frac{|\text{OPT}(B_\Pi,C_\Pi)|}{\pi^*/2\lambda^2}, |\text{LCS}(B_\Sigma, C_\Sigma)|\}$. Notice that:

$$\begin{aligned}
2\lambda^2|\text{OPT}(B, C)| &\leq 2\lambda^2|\text{OPT}(B_\Pi, C_\Pi)| + 2\lambda^2|\text{LCS}(B_\Sigma, C_\Sigma)| && (9)\\
&= \pi^* \cdot \frac{|\text{OPT}(B_\Pi, C_\Pi)|}{\pi^*/2\lambda^2} + 2\lambda^2|\text{LCS}(B_\Sigma, C_\Sigma)| && (10)\\
&\leq \pi_{\min} \cdot \frac{|\text{OPT}(B_\Pi, C_\Pi)|}{\pi^*/2\lambda^2} + 2\lambda^2|\text{LCS}(B_\Sigma, C_\Sigma)|, && (11)
\end{aligned}$$

where (9) is true because $|\text{OPT}(B, C)| \leq |\text{OPT}(B_\Pi, C_\Pi)| + |\text{LCS}(B_\Sigma, C_\Sigma)|$ (since symbols from Π in the optimal solution cannot contribute more than $|\text{OPT}(B_\Pi, C_\Pi)|$, and likewise, symbols from Σ in the optimal solution cannot contribute more than $|\mathrm{LCS}(B_\Sigma, C_\Sigma)|$), and (11) is true due to the same explanation of (7)–(8). We conclude that $\pi_{\min} \cdot \frac{|\mathrm{OPT}(B_\Pi, C_\Pi)|}{\pi^*/2\lambda^2} + 2\lambda^2|\mathrm{LCS}(B_\Sigma, C_\Sigma)| \geq 2\lambda^2|\mathrm{OPT}(B, C)|$ and therefore

$$\max\left\{\pi_{\min} \cdot \frac{|\mathrm{OPT}(B_\Pi, C_\Pi)|}{\pi^*/2\lambda^2}, 2\lambda^2|\mathrm{LCS}(B_\Sigma, C_\Sigma)|\right\} \geq \lambda^2|\mathrm{OPT}(B, C)| \,. \tag{12}$$

We can therefore split to cases:

1. If $2\lambda^2|\mathrm{LCS}(B_\Sigma, C_\Sigma)| \geq \pi_{\min} \cdot \frac{|\mathrm{OPT}(B_\Pi, C_\Pi)|}{\pi^*/2\lambda^2}$, we get that $|\mathrm{LCS}(B_\Sigma, C_\Sigma)| \geq \frac{1}{2}|\mathrm{OPT}(B, C)|$.
2. Otherwise, $\pi_{\min} \cdot \frac{|\mathrm{OPT}(B_\Pi, C_\Pi)|}{\pi^*/2\lambda^2} > 2\lambda^2|\mathrm{LCS}(B_\Sigma, C_\Sigma)|$. Since It follows that $\pi_{\min} \cdot \frac{|\mathrm{OPT}(B_\Pi, C_\Pi)|}{\pi^*/2\lambda^2} \geq \lambda^2|\mathrm{OPT}(B, C)|$, in this case we finally conclude that $\max\{\pi_{\min}, \frac{|\mathrm{OPT}(B_\Pi, C_\Pi)|}{\pi^*/2\lambda^2}\} \geq \lambda\sqrt{|\mathrm{OPT}(B, C)|}$.

Summing up the two cases, we get:

$$\max\left\{\pi_{\min}, \frac{|\mathrm{OPT}(B_\Pi, C_\Pi)|}{\pi^*/2\lambda^2}, |\mathrm{LCS}(B_\Sigma, C_\Sigma)|\right\} \geq \min\left\{\lambda\sqrt{|\mathrm{OPT}(B, C)|}, \frac{1}{2}|\mathrm{OPT}(B, C)|\right\} \,. \tag{13}$$

Since $|\mathcal{A}_\lambda^{\mathrm{LCPS}}(B, C)| = \max\{\pi_{\min}, \frac{|\mathrm{OPT}(B_\Pi, C_\Pi)|}{\pi^*/2\lambda^2}, |\mathrm{LCS}(B_\Sigma, C_\Sigma)|\}$, the approximation factor follows.

Running-time. The running-time is dominated by the use of $\mathcal{A}_{\sqrt{2}\lambda}^{\mathrm{LCMS}}$ as a sub-procedure. Since it is executed on B_Π and C_Π with $\sqrt{2}\lambda$ as the parameter, its running-time (and therefore the running-time of the entire algorithm) is $O(n^{2(\sqrt{2}\lambda)^2+1}m) = O(n^{4\lambda^2+1}m)$. □

References

[1] A. Amir, M. Farach, S. Muthukrishnan, Alphabet dependence in parameterized matching, Inf. Process. Lett. 49 (3) (1994) 111–115.
[2] B.S. Baker, Parameterized pattern matching by boyer-moore-type algorithms, in: SODA, 1995, pp. 541–550.
[3] B.S. Baker, Parameterized pattern matching: Algorithms and applications, J. Comput. Syst. Sci. 52 (1) (1996) 28–42.
[4] B.S. Baker, Parameterized duplication in strings: Algorithms and an application to software maintenance, SIAM J. Comput. 26 (5) (1997) 1343–1362.
[5] B.S. Baker, Parameterized diff, in: SODA, 1999, pp. 854–855.
[6] R. Cole, R. Hariharan, Faster suffix tree construction with missing suffix links, In: STOC, 2000, pp. 407–415.
[7] F. Ferragina, R. Grossi, The string b-tree: A new data structure for string search in external memory and its applications, J. ACM 46 (2) (1999) 236–280.
[8] C. Hazay, M. Lewenstein, D. Sokol, Approximate parameterized matching, ACM Trans. Algorithms 3 (3) (2007).
[9] C. Hazay, M. Lewenstein, D. Tsur, Two dimensional parameterized matching, in: Combinatorial Pattern Matching, 16th Annual Symposium, CPM 2005, in: Lecture Notes in Computer Science, vol. 3537, Springer, 2005, pp. 266–279.
[10] J.W. Hunt, T.G. Szymanski, A fast algorithm for computing longest subsequences, Commun. ACM 20 (5) (1977) 350–353.
[11] T. Jiang, M. Li, On the approximation of shortest common supersequences and longest common subsequences, SIAM J. Comput. 24 (5) (1995) 1122–1139.
[12] R.M. Karp, Reducibility among combinatorial problems, in: Complexity of Computer Computations, Plenum Press, 1972, pp. 85–103.
[13] S.R. Kosaraju, Faster algorithms for the construction of parameterized suffix trees (preliminary version), in: FOCS, 1995, pp. 631–637.
[14] V. I. Levenshtein, Binary codes capable of correcting deletions, insertions, and reversals, Soviet Phys. Dokl. 10 (1966) 707–710.
[15] D. Maier, The complexity of some problems on subsequences and supersequences, J. ACM 25 (2) (1978) 322–336.
[16] W.J. Masek, M. Paterson, A faster algorithm computing string edit distances, J. Comput. Syst. Sci. 20 (1) (1980) 18–31.
[17] E.W. Myers, An $o(nd)$ difference algorithm and its variations, Algorithmica 1 (2) (1986) 251–266.
[18] N. Nakatsu, Y. Kambayashi, S. Yajima, A longest common subsequence algorithm suitable for similar text strings, Acta Inf. 18 (1982) 171–179.
[19] R.A. Wagner, M.J. Fischer, The string-to-string correction problem, J. ACM 21 (1) (1974) 168–173.