Heuristic algorithms for the Longest Filled Common Subsequence Problem

Radu Stefan Mincu
Department of Computer Science
University of Bucharest
Bucharest, Romania
mincu.radu@fmi.unibuc.ro

Alexandru Popa
Department of Computer Science
University of Bucharest
Bucharest, Romania
alexandru.popa@fmi.unibuc.ro
National Institute for Research
and Development in Informatics
Bucharest, Romania

Abstract—At CPM 2017, Castelli et al. define and study a new variant of the Longest Common Subsequence Problem, termed the Longest Filled Common Subsequence Problem (LFCS). For the LFCS problem, the input consists of two strings A and B and a multiset of characters $\mathcal{M}$. The goal is to insert the characters from $\mathcal{M}$ into the string B, thus obtaining a new string B^*, such that the Longest Common Subsequence (LCS) between A and B^* is maximized. Casteli et al. show that the problem is NP-hard and provide a 3/5-approximation algorithm for the problem.

In this paper we study the problem from the experimental point of view. We introduce, implement and test new heuristic algorithms and compare them with the approximation algorithm of Casteli et al. Moreover, we introduce an Integer Linear Program (ILP) model for the problem and we use the state of the art ILP solver, Gurobi, to obtain exact solution for moderate sized instances.

Keywords-NP-hard problem; longest common subsequence; heuristics;

I. INTRODUCTION

MOTIVATION AND PREVIOUS WORK: The Longest Common Subsequence problem (LCS) is long studied [1]–[3] and its importance is further emphasized by its a wide array of applications in diverse fields such as data compression, computational biology, text editing and comparison (notable examples include the Unix `diff` utility and plagiarism detection systems), pattern recognition.

In the field of computational biology, advances in genome sequencing techniques are bringing about the need for algorithms to be used in the analysis and reconstruction of genomic data. As a consequence, many LCS variants have been developed to interpret the DNA sequence fragments resulting from various DNA sequencing procedures for the purpose of reconstructing a genome. Among these studied problems is the *Constrained LCS problem (CLCS)* first presented by Tsai [4] which is proven to be equivalent to a particular case of *Constrained Sequence Alignment (CSA)* [5]. There are applications for the LCS problem in comparing genome data such as the exemplar model based LCS variants, namely the *Exemplar LCS (ELCS)* variants [6] and the *Repetition Free*

LCS (RFLCS) problem [7]. A generalization of the CLCS and RFLCS problems exists in the *Doubly Constrained LCS problem (DCLCS)* [8].

In CPM 2017, Castelli, Dondi, Mauro and Zoppis introduce a new LCS variant called *Longest Filled Common Subsequence (LFCS)* [9] inspired from the particulars of the *Scaffold Filling problem* in genome reconstruction [10], [11]. The LFCS problem is defined as follows:

Definition 1 (Longest Filled Common Subsequence)**.** *Let there be two strings A and B over alphabet Σ and a multiset $\mathcal{M}$ of symbols in Σ. A filling B^* of string B is defined as a sequence obtained from B by inserting a subset of the symbols from $\mathcal{M}$ into B. Find a filling B^* that maximizes the length of the longest common subsequence of strings A and B^*.*

Castelli et al. prove that LFCS is $\mathcal{APX}$-hard even when A contains at most two occurrences of each symbol in Σ [9]. They also show a 3/5-approximation algorithm and a fixed parameter algorithm where the parameter is the number of symbols from $\mathcal{M}$ inserted in the string B.

In this paper we study the problem from the experimental point of view. We introduce, implement and test new heuristic algorithms and compare them with the approximation algorithm of Casteli et al. Moreover, we give an ILP formulation for the problem and we use the state of the art ILP solver, Gurobi, to obtain exact solution for moderate sized instances.

The rest of the paper is organized as follows.

In Section II we describe how to reduce the search space for LFCS and how to obtain bounds for the solution value. Section III contains an ILP model used to obtain optimum solutions. In practice, exact methods such as ILP solving might take a long time to complete, therefore we construct heuristic algorithms in Section IV and test them on procedurally generated data. The results and the experiments are described in Section V.

II. PRELIMINARIES

A. LFCS Solution Space

We wish to reason about the solution space of the LFCS problem to offer some insights into its nature. To begin with, we may consider the extreme where $\mathcal{M} = \emptyset$. Then, the solution for LFCS is the same as the LCS value of inputs A and B. Conversely, if $\mathcal{M}$ contains A entirely, then we can just pick as a filling $B^* = AB$, and the solution for LFCS is of size $|A|$. Clearly, the difficult cases to solve are when $\mathcal{M}$ is in the middle-ground. If $\mathcal{M}$ is very small, then there are few insertions to consider. Moreover, if $\mathcal{M}$ is too extensive, then the same situation occurs. To better understand the solution space, let us consider an alternative formulation of LFCS:

Definition 2 (Longest Filled Common Subsequence (alt.)). *Let there be two strings A and B over alphabet Σ and a multiset $\mathcal{M}$ of symbols in Σ. Let A' be string A from which we have removed a subset of symbols, no more than the amount contained in $\mathcal{M}$. Find A' that maximizes the sum between $|A| - |A'|$ and the length of the longest common subsequence of strings A' and B.*

The above definition changes the focus from inserting elements from $\mathcal{M}$ into B to deleting the elements matched with $\mathcal{M}$ from A. The reason why we prefer the latter formulation is because the search space is smaller and much easier to compute than the former, while the optimum value of the problem remains the same.

We illustrate this with an example in Figure 1: abd are matched with $\mathcal{M}$, while cbda is the LCS of A' =cbcda and B =cabbdda. Alternatively, we may consider the LCS of A with one of the optimum $B^* \ni \{$abcdabbdda, abcadbbdda, abcabdbdda$\}$. Thus, it is equivalent to say that d can be inserted into three places in B or to say that d from A is matched with $\mathcal{M}$ for the purpose of solving the problem.
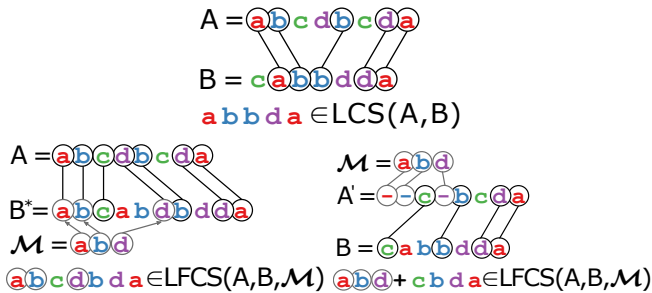

Figure 1. Examples of LCS and LFCS. On the top side is one LCS of A, B. On the bottom side, abd are matched with $\mathcal{M}$, and cbda are LCS-matched. Standard LFCS is used on the left side, while the alternative LFCS formulation is illustrated on the right side.

To obtain solutions for LFCS, we have seen that we can focus on deleting the symbols from A that are matched with $\mathcal{M}$, effectively reducing the search space. But it is possible to do even better.

Observation 1: Notice that we may only match symbols of A with as many symbols as there exist in $\mathcal{M}$.

Observation 2: Given an optimum solution for LFCS, it either has the maximum number of matched symbols with $\mathcal{M}$ or one may construct an optimum solution for LFCS that uses an extra symbol from $\mathcal{M}$ by picking a symbol present in both the LCS and in the unmatched subset of $\mathcal{M}$ and matching it with $\mathcal{M}$.

Using the above observations we may compute the search space for a given instance by counting all A'. Observations 1,2 help us to reduce the search space further by allowing us to disregard all A' that do not have the maximum number of matchings with $\mathcal{M}$. As such, in order to describe the search space, one may enumerate all possible ways to match the maximum number of symbols in $\mathcal{M}$ with the symbols of A.

Going back to the example in Figure 1, we wish to compute the size of the search space $S(A, \mathcal{M})$. We see that we can match d, a and b each in two places in A, yielding a total of 8 combinations. In the general case, we go though $\mathcal{M}$ and for each distinct symbol we calculate all the possible ways to match the occurrences in A with exactly the amount available in $\mathcal{M}$. If we do this for each symbol, we get a product of combinations. More formally:

Let $a(\sigma) = |A|_\sigma$, $m(\sigma) = \min(|\mathcal{M}|_\sigma, |A|_\sigma)$. Then:

$$S(A, \mathcal{M}) = \prod_{\sigma \in \Sigma} \binom{a(\sigma)}{m(\sigma)} \tag{1}$$

The ability to compute the size of the search space becomes important when we consider heuristic algorithms based on randomization. For example, if the search space is reasonably small, we may obtain an optimum solution by repeatedly evaluating random solutions.

B. Bounds for the LFCS Solution

As a lower bound for the LFCS problem is first given by the LCS of the A, B inputs. For a tighter and quite practical lower bound we have used in our experiments the 3/5-approximation algorithm from [9].

When considering an upper bound for the LFCS solution, one quick method is to sum the value of LCS(A,B) with $|A \cap \mathcal{M}|$ (as a multiset intersection). In other words, the value of the solution cannot exceed the case where the symbols in A are maximally matched with $\mathcal{M}$ and the sets of symbols matched with $\mathcal{M}$ and by LCS are disjoint.

III. ILP MODEL FOR THE LFCS PROBLEM

In this section we show an ILP formulation for the LFCS problem. Let $n = |A|$ and $m = |B|$. We define a variable x_{ij} for every $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, m\}$. We have $x_{ij} = 1$ if $a_i = b_j$ and these two charcters belong to the LFCS solution. We also define a variable y_i for every

$i \in \{1, \ldots, n\}$ which is equal to 1 if a_i is matched with a character from $\mathcal{M}$. The full ILP model follows:

$$\max \sum_{i=1}^{n} y_i + \sum_{i=1}^{n} \sum_{j=1}^{m} x_{ij} \text{ subject to:} \qquad (2a)$$

$$x_{ij} + x_{kl} \leq 1, \ \forall (i < k) \wedge (j > l) \qquad (2b)$$

$$y_i + \sum_{j=1}^{m} x_{ij} \leq 1, \ \forall i \in \{1, \ldots, n\} \qquad (2c)$$

$$\sum_{i=1}^{n} x_{ij} \leq 1, \ \forall j \in \{1, \ldots, m\} \qquad (2d)$$

$$\sum_{i=1, a_i=\sigma}^{n} y_i \leq |\mathcal{M}|_\sigma, \ \forall \sigma \in \mathcal{M} \qquad (2e)$$

$$x_{ij}, y_i \in \{0, 1\} \qquad (2f)$$

Explanation for the constraints in the ILP:

1) Constraint 2b ensures that two pairs of characters that match are non-crossing.
2) Constraint 2c ensures that each character from A may only be matched with a character from $\mathcal{M}$ or a single character from B.
3) Constraint 2d ensures that each character from B may only be matched with a single character from A.
4) Constraint 2e ensures that the characters from A that are matched with $\mathcal{M}$ do not exceed the total amount available in $\mathcal{M}$ for that particular character.

IV. HEURISTIC ALGORITHMS FOR THE LFCS PROBLEM

In this section we describe heuristics for obtaining solutions for LFCS in a practical setting.

A. Uniformly Sampled Solutions

A straightforward method to approach the problem is to solve uniformly random solutions in the search space of a given instance. The solutions are determined by selecting a random combination of symbols from A to match with $\mathcal{M}$, such that A is maximally matched with $\mathcal{M}$. This can be achieved procedurally by uniform sampling of the index-set of each symbol in the alphabet. Subsequently, an LCS is performed on the remainder A' and B and the value of the LFCS solution will be LCS(A',B) + $|A \cap \mathcal{M}|$. For our experiments we have chosen to generate a constant 10000 random solutions. For more consistency, it may be advisable for the sample count to be a fraction of the search space (as computed in Section II).

B. Local Search Algorithm

We consider an algorithm inspired from local search techniques and greedy hillclimbing. We start with A as an initial solution and wish to construct A' by matching symbols of A with $\mathcal{M}$.

At each iteration of the algorithm we wish to match with $\mathcal{M}$ (i.e. delete) as many as k symbols, located on
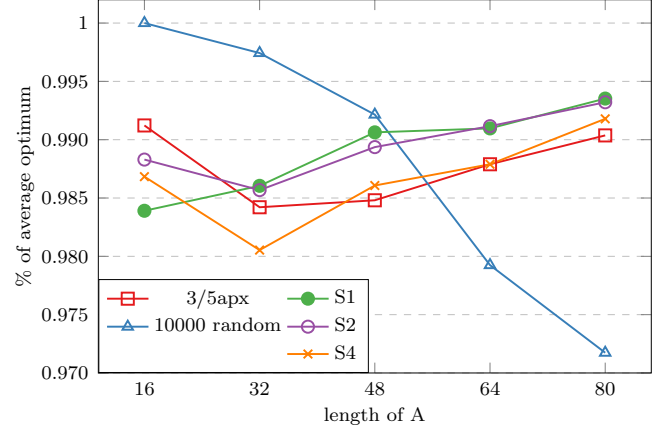

Figure 2. Average of the solutions returned by the algorithms. All values are divided by the average optimum. The alphabet of the instances is of size 1/8 the length of A.

Figure 3. Average of the solutions returned by the algorithms. All values are divided by the average optimum. The alphabet of the instances is of size 1/4 the length of A.
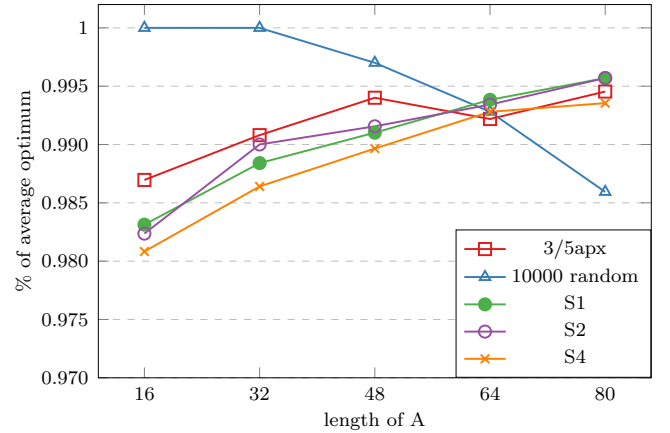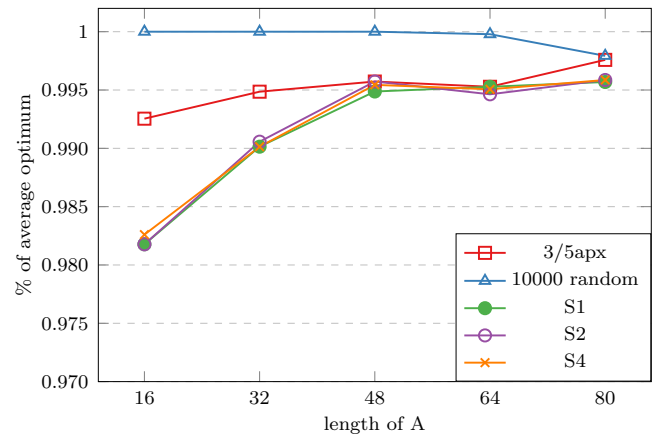

Figure 4. Average of the solutions returned by the algorithms. All values are divided by the average optimum. The alphabet of the instances is of size 1/2 the length of A.

k consecutive positions in A, such that the sum of the number of matched (i.e. deleted) symbols and the LCS of the unmatched remainder of A and B is the maximum. The best solution becomes the new incumbent solution. The algorithm terminates when we fail to match at least 1 new symbol.

For each incumbent solution I, the neighborhood to be explored is composed of all strings I' such that I' is obtained by deleting symbols within any window of length k from I. Here, *window* means exactly k consecutive positions in I. In other words, we start with a left-to-right sliding window of length k and attempt all 2^k possible ways to match the symbols, if the amount in $\mathcal{M}$ is sufficient. If $|A| = n$, we compute at most $2^k(n-k)$ LCS-s at each iteration.

We dub our algorithm Sk, where the parameter k is the length of the window.

V. Experimental results

We run our algorithms and the 3/5-approximation from [9] on a selection of instances that are randomly and procedurally generated. We find the optimum solution for each instance using an implementation of our ILP model in Gurobi/Python.

To obtain each $(A, B, \mathcal{M})$ instance we first generate a uniform string for A. To make B, we first copy A and iterate through it, changing symbols with a probability of 50%. The change applied is uniformly chosen among duplication, deletion and substitution with a different symbol. We then randomly split B into segments no longer than $|A|/8$ and discard $>30\%$ of them. The remainder is the final B string, while the discarded symbols are put into $\mathcal{M}$.

For our tests we select $n = |A|$ to be 16, 32, 48, 64, 80, while the alphabet size is $\frac{n}{8}, \frac{n}{4}, \frac{n}{2}$. For each combination of n and $|\Sigma|$, we generate 100 instances.

The results are showcased in Table I and Figures 2, 3, 4. All the algorithms arrive within 97% of the optimum. We can observe that the 3/5-approximation algorithm is quite good in the general case, but there are instances where our local search algorithm Sk outperforms the 3/5-approximation algorithm. This is noticeable for small alphabets and in Figure 2, where all Sk perform better for lengths 48, 64, 80. As expected, random sampling for solutions shines when the solution space is smaller, which is the case for Figure 4.

Because the choices of algorithm Sk are not reversible, the algorithm is prone to get stuck in local optima as k increases. To improve the clarity of the figures, we limited our selection to S1, S2, S4, which are among the best tested.

VI. Conclusions and future work

In this paper study the LFCS problem introduced by Casteli et al. from an experimental point of view. We give ILP formulations for the problem and we use the state of the art ILP solver, Gurobi, to obtain exact solution for moderate sized instances. We introduce, implement and

Table I
TABLE ILLUSTRATING HOW MANY TIMES EACH ALGORITHM REACHES AN OPTIMUM SOLUTION IN OUR EXPERIMENTS ON 100 INSTANCES (EACH CASE). THE FIRST COLUMN DESCRIBES THE ALPHABET SIZE, THEN FOLLOWS THE LENGTH OF THE A INPUT STRINGS AND THEN ARE HOW MANY OPTIMA EACH ALGORITHM FINDS.

Σ	n	3/5apx	rand	S1	S2	S4
$\frac{n}{8}$	16	88	100	78	84	82
	32	66	93	68	65	56
	48	53	70	68	63	54
	64	50	22	57	60	51
	80	53	9	65	62	57
$\frac{n}{4}$	16	83	100	78	77	77
	32	77	100	72	75	68
	48	80	89	71	71	70
	64	70	73	73	72	69
	80	71	36	78	79	67
$\frac{n}{2}$	16	91	100	79	80	80
	32	89	100	77	78	77
	48	85	100	82	86	85
	64	79	99	79	76	77
	80	87	90	76	77	76

test new heuristic algorithms and compare them with the approximation algorithm of Casteli et al..

A natural open question is to find a better than 3/5 approximation algorithm for the LFCS problem.

References

[1] A. Apostolico, *String Editing and Longest Common Subsequences*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 361–398.

[2] L. Bergroth, H. Hakonen, and T. Raita, "A survey of longest common subsequence algorithms," in *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000*, 2000, pp. 39–48.

[3] D. S. Hirschberg, "Algorithms for the longest common subsequence problem," *Journal of the ACM (JACM)*, vol. 24, no. 4, pp. 664–675, 1977.

[4] Y.-T. Tsai, "The constrained longest common subsequence problem," *Information Processing Letters*, vol. 88, no. 4, pp. 173 – 176, 2003.

[5] F. Y. Chin, A. D. Santis, A. L. Ferrara, N. Ho, and S. Kim, "A simple algorithm for the constrained sequence problems," *Information Processing Letters*, vol. 90, no. 4, pp. 175 – 179, 2004.

[6] P. Bonizzoni, G. Della Vedova, R. Dondi, G. Fertin, R. Rizzi, and S. Vialette, "Exemplar longest common subsequence," *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, vol. 4, no. 4, pp. 535–543, Oct. 2007.

[7] S. S. Adi, M. D. Braga, C. G. Fernandes, C. E. Ferreira, F. V. Martinez, M.-F. Sagot, M. A. Stefanes, C. Tjandraatmadja, and Y. Wakabayashi, "Repetition-free longest common subsequence," *Discrete Applied Mathematics*, vol. 158, no. 12, pp. 1315 – 1324, 2010, traces from LAGOS07 IV Latin American Algorithms, Graphs, and Optimization Symposium Puerto Varas - 2007.

[8] P. Bonizzoni, G. D. Vedova, R. Dondi, and Y. Pirola, "Variants of constrained longest common subsequence," *Information Processing Letters*, vol. 110, no. 20, pp. 877 – 881, 2010.

[9] M. Castelli, R. Dondi, G. Mauri, and I. Zoppis, "The Longest Filled Common Subsequence Problem," in *28th Annual Symposium on Combinatorial Pattern Matching (CPM 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), J. Kärkkäinen, J. Radoszewski, and W. Rytter, Eds., vol. 78. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 14:1–14:13.

[10] A. Muñoz, C. Zheng, Q. Zhu, V. A. Albert, S. Rounsley, and D. Sankoff, "Scaffold filling, contig fusion and comparative gene order inference," *BMC Bioinformatics*, vol. 11, no. 1, p. 304, Jun 2010.

[11] L. Bulteau, A. P. Carrieri, and R. Dondi, "Fixed-parameter algorithms for scaffold filling," *Theoretical Computer Science*, vol. 568, pp. 72 – 83, 2015.