A Heuristic Approach for Solving the Longest Common Square Subsequence Problem

Marko Djukanovic[1], Günther R. Raidl[1], and Christian Blum[2]

[1]Institute of Logic and Computation, TU Wien, Vienna, Austria,
[2] Artificial Intelligence Research Institute (IIIA-CSIC),
Campus UAB, Bellaterra, Spain
{djukanovic,raidl}@ac.tuwien.ac.at,
christian.blum@iiia.csic.es

Abstract. The longest common square subsequence (LCSqS) problem, a variant of the longest common subsequence (LCS) problem, aims at finding a subsequence common to all input strings that is, at the same time, a square subsequence. So far the LCSqS was solved only for two input strings. We present a heuristic approach, based on randomized local search and a hybrid of variable neighborhood search and beam search, to solve the LCSqS for an arbitrary set of input strings. The beam search makes use of a novel heuristic estimation of the approximated expected length of a LCS to guide the search.

Keywords: Longest common subsequence problem; Reduced variable neighborhood search; Beam search; hybrid metaheuristic; expected value.

1 Introduction

A *string* s is a finite sequence of symbols from a finite alphabet Σ. The length of a string s, denoted by $|s|$, is defined as the number of symbols in s. Strings are data structures for storing words or even complete texts. In the field of bioinformatics, strings are used to represent DNA and protein sequences. As a consequence, many computational problems in bioinformatics are phrased in terms of so-called string problems. These problems usually present measures of similarities for different molecular structures. Each string obtained from a string s by removing zero or more characters is a *subseqence* of s. A fundamental measure of similarity among molecules is the length of the *longest common subsequence* (LCS): given an arbitrary set of m input strings, $S = \{s_1, \ldots, s_m\}$, we aim at finding the longest subsequence that is common to all input strings [7]. The classical variant of the LCS for $m = 2$ has been studied for almost 50 years. The general LCS for $m \geq 2$ has been tackled for the first time by Huang et al. [4], and later by Blum et al. [1], Mousavi and Tabataba [9], and others.
Recently, *the longest common square subsequence* (LCSqS) problem, a variant of the LCS, was proposed by Inoue et al. [5]. It requires that the resulting LCS is at the same time a square subsequence. A string s is a *square* iff $s = s' \cdot s' = s'^2$, for some string s', where "$\cdot$" denotes the string concatenation. The length of

the LCSqS can be seen as a measure of similarity between disjunctive parts of each of the compared molecules. Therefore, it can give more insight into the internal similarity of the compared molecules than when just considering a LCS. Moreover, the information about those parts of the molecules that are similar to each other is obtained by identifying a LCSqS. Inoue et al. [5] proved that the LCSqS problem is $\mathcal{NP}$-hard for an arbitrary set of input strings and proposed two approaches for the case of two input strings: (1) a *Dynamic Programming* (DP) approach running in $O(n^6)$ time (n denotes the length of the largest input string), and (2) a sparse DP-based approach, which makes use of a special geometric data structure. It can be proven that, if m is fixed, the LCSqS is polynomially solvable by DP in $O(n^{3m})$ time which is not practical already for small input sizes. To the best of our knowledge, no algorithm has yet been proposed for solving the LCSqS problem for an arbitrary number $m \geq 2$ of input strings. The main contributions of this paper are as follow:

- A transformation of the LCSqS problem to a series of the standard LCS problems is described.
- An approach based on a randomized local search and a hybrid of *a Reduced Variable Neighborhood Search* (RVNS) [8] and *a Beam Search* (BS) are proposed for solving the general LCSqS problem.
- An approximation for the expected length of a LCS is derived and incorporated into the BS framework to guide its search.

Organization of the Paper. The paper is organized as follows. Section 2 describes basic solution approaches for the LCS known from the literature. Section 3 gives a basic reduction from the LCSqS to the LCS problem and two approaches to solve the LCSqS. Section 4 presents computational results, and Section 5 outlines some research questions and directions for future work.

2 Solution Approaches for the LCS Problem

For $i \leq j$, let $s[i,j] = s[i] \cdots s[j]$ be a continuous (sub)string of a string s which starts from index i and ends at index j. If $i > j$, $s[i,j]$ is the empty string ε. For $p^{\mathrm{L}} \in \mathbb{N}^m$, which is called the left position vector, $S[p^{\mathrm{L}}] := \{s_i[p^{\mathrm{L}}_i, |s_i|] \mid i = 1, \ldots, m\}$ denotes the set of the remaining parts of the input strings of S w.r.t. p^{L}.

The Best-Next Heuristic (BNH) for the LCS was proposed by Huang et al. [4]. This heuristic starts with an empty partial LCS solution $s^{\mathrm{p}} = \varepsilon$ which is then iteratively extended by a feasible letter. If there exists more than one candidate to extend s^{p}, the decision which one to choose is made by a *greedy* heuristic, calculating for each of the candidate letters a greedy value. The BNH works in detail as follows. We initialize s^{p} to the empty string ε, and the left pointers p^{L} to $(1, \ldots, 1)$, indicating that the complete input strings are still relevant for finding extensions of s^{p}. Each letter $a \in \Sigma$ which appears at least once in all strings from $S[p^{\mathrm{L}}]$ is considered as a *feasible* candidate to extend s^{p}. Let us denote by $p^{\mathrm{L}}_{a,i}$

the position of the first occurrence of letter a in $s_i[p^{\mathrm{L}}_i, |s_i|]$. Among the feasible letters, *dominated* ones may occur. We say that letter a *dominates* letter b iff $p^{\mathrm{L}}_{a,i} \leq p^{\mathrm{L}}_{b,i}$, $\forall i = 1, \dots, m$. Non-dominated letters are preferred, since the choice of a dominated letter will lead to a suboptimal solution. Denote a set of feasible, non-dominated letters by $\Sigma^{\mathrm{nd}}_{p^{\mathrm{L}}} \subseteq \Sigma$. The letter given by

$$a^* = \arg \min_{a \in \Sigma^{\mathrm{nd}}_{p^{\mathrm{L}}}} \left(g(p^{\mathrm{L}}, a) = \sum_{i=1}^{m} \frac{p^{\mathrm{L}}_{a,i} - p^{\mathrm{L}}_i}{|s_i| - p^{\mathrm{L}}_i + 1} \right)$$

is chosen to extend s^{p}, and we update $s^{\mathrm{p}} := s^{\mathrm{p}} \cdot a^*$, and $p^{\mathrm{L}}_i := p^{\mathrm{L}}_{a^*,i} + 1$, $i = 1, \dots, m$, accordingly. We repeat the steps with the new s^{p} and p^{L} until $\Sigma^{\mathrm{nd}}_{p^{\mathrm{L}}} = \emptyset$ is met, constructing a complete greedy solution s^{p}. Let us label this procedure by BNH(S), taking an instance S as input and returning the derived greedy solution.

Beam Search is known as an incomplete tree search algorithm which expands nodes in a breadth-first search manner. It maintains a collection of nodes called *beam.* The $\beta > 0$ most promising nodes of these expansions are further used to create the beam of the next level. This step is repeated level by level until the beam is empty. We will consider several ways to determine the most promising nodes to be kept at each step in Section 2.1. A BS for the LCS has been proposed by Blum et al. [1]. Each node v of the LCS is defined by a left position vector $p^{\mathrm{L},v}$ which corresponds to the set $S[p^{\mathrm{L},v}]$ relevant for further extension of v, and an l^v-value, denoting the length of the partial solution represented by the node. Initially, the beam contains the (root) node $r := ((1, \dots, 1), 0)$. In order to expand a node v, the corresponding set $S[p^{\mathrm{L},v}]$ is used to find all *feasible*, non-dominated letters $\Sigma^{\mathrm{nd}}_{p^{\mathrm{L},v}}$, and for each $a \in \Sigma^{\mathrm{nd}}_{p^{\mathrm{L},v}}$, the positions $p^{\mathrm{L},v}_{a,i}$ are determined, and further used to create all successor nodes $v' = (p^{\mathrm{L},v}_a + 1, l^v + 1)$ of v, where $p^{\mathrm{L},v}_a + 1 = \{p^{\mathrm{L},v}_{a,i} + 1 \,|\, i = 1, \dots, m\}$. If $\Sigma^{\mathrm{nd}}_{p^{\mathrm{L},v}} = \emptyset$, a *complete* node has been reached. If the l^v-value of a complete node is greater than the length of the current incumbent solution, we derive the respective solution and store it as the new incumbent. We emphasize that directly storing partial solutions within the nodes is not necessary. For any node of the search tree, the respective partial solution can be derived in a backward manner by iteratively identifying predecessors in which the l^v-values always decrease by one. Let us denote this procedure by BS(S, β), taking a set S and a beam size β as input and returning the best solution found by the BS execution.

2.1 Estimating the Length of the LCS Problem

For each node of the search tree, an upper bound on the number of letters that might further be added—i.e., the length of a LCS of $S[p^{\mathrm{L},v}]$—is given by $\mathrm{UB}(v) = \mathrm{UB}(S[p^{\mathrm{L},v}]) = \sum_{i=1}^{m} c_a$, where $c_a = \min\{|s_i[p^{\mathrm{L},v}_i, |s_i|]|_a \,|\, i = 1, \dots, m\}$, and $|s|_a$ is the number of occurrences of letter a in s; see [10]. Unfortunately, this upper bound is not tight in practice.

We develop here a novel estimation based on the approximated expected length for uniform random strings. Mousavi and Tabataba [9] came up with a recursion which calculates the probability that a specific string s of length k is a subsequence of a uniform random string t of length q in the form of a table $\mathcal{P}(|s|,|t|) = \mathcal{P}(k,q)$. Let us assume that the following holds: (1) all strings from S are mutually independent and (2) for any sequence of length k over Σ, the event that the sequence is a subsequence common to all strings in S is independent of the corresponding events for other sequences. By making use of the table $\mathcal{P}$ and some basic laws from probability theory, we can derive the estimation for the expected length as

$$\mathrm{EX}(v) = \mathrm{EX}(S[p^{\mathrm{L},v}]) = l_{\max} - \sum_{k=1}^{l_{\max}} \left(1 - \prod_{i=1}^{m} \mathcal{P}(k, |s_i| - p_i^{\mathrm{L},v} + 1)\right)^{|\Sigma|^k}, \quad (1)$$

where $l_{\max} = \min_{i=1,\ldots,m}\{|s_i| - p_i^{\mathrm{L},v} + 1\}$. EX provides, in practice, a much better approximation than the afore-mentioned upper bound UB or the heuristic from [9] which is also of limited use since it cannot be used to compare nodes from different levels of the search tree. Formula (1) is numerically calculated by decomposing the power $|\Sigma|^k = \underbrace{|\Sigma|^p \cdots |\Sigma|^p}_{\lfloor k/p \rfloor} \cdot |\Sigma|^{(k \mod p)}$, since intermediate values would otherwise be too large for a commonly used floating point arithmetic. Moreover, the calculation of (1) can be run in $O(m \log(n))$ on average by determining $v_k = \left(1 - \prod_{i=1}^{m} \mathcal{P}(k, |s_i| - p_i^{\mathrm{R},v} + 1)\right)^{|\Sigma|^k} \in (\epsilon, 1-\epsilon)$ using the divide-and-conquer principle exploiting the fact that $\{v_k\}_{k=1}^{l_{\max}}$ is a monotonic sequence. We set $\epsilon = 10^{-10}$ in our implementation. If the product which appears in (1) is close to zero, it can cause stability issues resolved by replacing v_k by an approximation derived from the Taylor expansion $(1-x)^\alpha \approx 1 - \alpha x + \binom{\alpha}{2}x^2 + o(x^2)$ which approximates v_k well. This estimation was developed by following the same idea for the palindromic case of the LCS problem; see [3] for more details.

3 Algorithms for Solving the LCSqS Problem

Let us denote by $\mathbb{P} := \{(q_1,\ldots,q_m) : 1 \le q_i \le |s_i|\} \subset \mathbb{N}^m$ all possibilities for partitioning the strings from S each one into two consecutive substrings. For each $q \in \mathbb{P}$, we define the left and right partitions of S by $S^{\mathrm{L},q} = \{s_1[1,q_1],\ldots,s_m[1,q_m]\}$ and $S^{\mathrm{R},q} = \{s_1[q_1{+}1,|s_1|],\ldots,s_m[q_m{+}1,|s_m|]\}$, respectively. Let $S^q := S^{\mathrm{L},q} \cup S^{\mathrm{R},q}$ be the joint set of these partitions. Finding an optimal solution s^*_{lcsqs} to the LCSqS problem can then be done as follows. First, an optimal LCS $s^*_{\mathrm{lcs},q}$ must be derived for all S^q, $q \in \mathbb{P}$. Let $s^*_{\mathrm{lcs}} = \arg\max\{|s^*_{\mathrm{lcs},q}| : q \in \mathbb{P}\}$ Then, $s^*_{\mathrm{lcsqs}} = s^*_{\mathrm{lcs}} \cdot s^*_{\mathrm{lcs}}$. Unfortunately, the LCS problem is already $\mathcal{NP}$–hard [7], and the size of $\mathbb{P}$ grows exponentially with the instance size. This approach is, therefore, not practical. However, we will make use of this decomposition approach in a heuristic way as shown in the following.

3.1 Randomized Local Search Approach

In this section we adapt and iterate BNH in order to derive approximate LCSqS solutions in the sense of a randomized local search (RLS).
We start with the $q = \left(\lfloor\frac{|s_1|}{2}\rfloor, \ldots, \lfloor\frac{|s_m|}{2}\rfloor\right)$ and execute BNH on the corresponding set S^q to produce an initial approximate LCSqS solution $s_{\mathrm{lcsqs}} = (\mathrm{BNH}(S^q))^2$. At each iteration, q is perturbed by adding to each q_i, $i = 1, \ldots, m$, a random offset sampled from the discretized normal distribution $\lceil\mathcal{N}(0, \sigma^2)\rceil$ with a probability $destr \in (0, 1)$, where the standard deviation is a parameter of the algorithm. BNH is applied to the resulting string set S^q for producing a new solution. A better solution is always accepted as new incumbent solution s_{lcsqs}. The whole process is iterated until a time limit $t_{\max}$ is exceeded. Note that if s_{lcsqs} is the current incumbent, only values in $\{|s_{\mathrm{lcsqs}}|/2 + 1, \ldots, |s_i| - |s_{\mathrm{lcsqs}}|/2 - 1\}$ for q_i can lead to better solutions. We therefore iterate the random sampling of each q_i until a value in this range is obtained.

3.2 RVNS&BS Approach

As an alternative to the RLS described above we consider a variable neighborhood search approach [8]. More precisely, we use a version of the VNS with no local search method included, known as *Reduced* VNS (RVNS).
For a current vector $q \in \mathbb{P}$, we define a move in the k-th neighborhood, $k = 1, \ldots, m$, by perturbing exactly k randomly chosen positions as above by adding a discretized normally distributed sampled random offset. Again, we take care not to choose meaningless small or larges values. We then evaluate q by the following 3-step process. We first calculate $ub_q = 2 \cdot \mathrm{UB}(S^q)$, and if $ub_q \leq |s_{\mathrm{lcsqs}}|$, q cannot yield an improved incumbent solution and q is discarded. Otherwise, we perform a fast evaluation of q by applying BNH which yields a solution $s = (\mathrm{BNH}(S^{q'}))^2$. If $|s| > \alpha \cdot |s_{\mathrm{lcsqs}}|$, where $\alpha \in (0, 1)$ is a threshold parameter of the algorithm, we consider q promising and further execute BS on S^q, yielding solution $s^{\mathrm{bs}} = (\mathrm{BS}(S^{q'}, \beta))^2$. Again, the incumbent solution s_{lcsqs} is updated by any obtained better solution. If an improvement has been achieved, the RVNS&BS always continues with the first neighborhood, i.e. $k := 1$, otherwise with the next neighborhood, i.e. $k := k + 1$ until $k = m + 1$ in which case k is reset to 1. To improve the performance, we store all partitionings evaluated by BS, together with their evaluations, in a hash map and retrieve these values in case the corresponding partitionings are re-encountered.

4 Computational Experiments

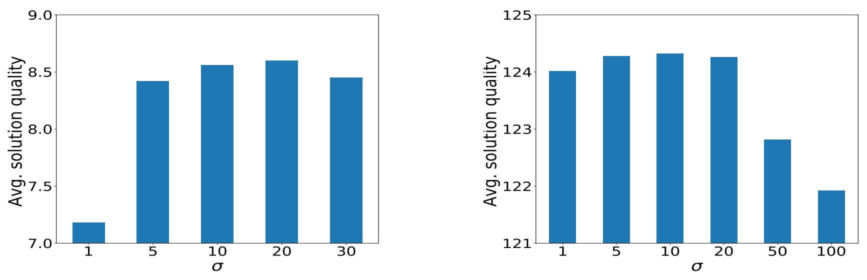
The algorithms are implemented in C++ and all experiments are performed on a single core of an Intel Xeon E5-2640 with 2.40GHz and 8 GB of memory.
We used the set of benchmark instances provided in [2] for the LCS problem. This instance set consists of ten randomly generated instances for each combination of the number of input strings $m \in \{10, 50, 100, 150, 200\}$, the length of the input

Table 1: Selected results for the LCSqS problem.

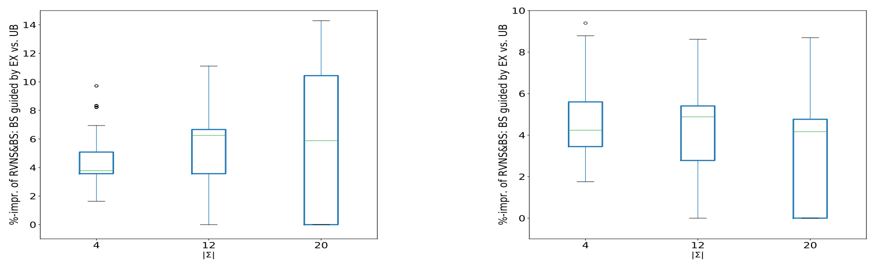
n	m	$\|\Sigma\|$	**RVNS&BS**		**RLS&BS**		**RVNS&Dive**		**RLS**	
			$\|\overline{s}\|$	$\bar{t}_{\text{best}}[s]$	$\|\overline{s}\|$	$\bar{t}_{\text{best}}[s]$	$\|\overline{s}\|$	$\bar{t}_{\text{best}}[s]$	$\|\overline{s}\|$	$\bar{t}_{\text{best}}[s]$
100	10	4	**27.08**	67.71	26.54	44.94	26.96	51.20	26.42	34.40
	10	20	3.84	0.02	**4.00**	1.66	3.96	0.05	**4.00**	4.44
	50	4	**18.54**	10.53	18.16	24.12	**18.54**	45.81	18.04	19.43
	50	20	0.20	0.01	**0.46**	4.77	0.20	0.00	0.40	0.01
	200	4	**14.00**	4.88	**14.00**	8.68	**14.00**	1.36	13.94	24.12
	200	20	**0.00**	0.05	**0.00**	0.00	**0.00**	0.00	**0.00**	0.00
500	10	4	**156.58**	143.70	156.14	146.08	149.78	160.69	149.24	110.09
	10	20	**35.78**	53.31	35.12	48.07	34.54	50.42	34.56	71.16
	50	4	**124.30**	52.66	124.12	160.39	120.32	86.33	120.12	109.37
	50	20	**21.14**	78.62	20.52	34.12	20.64	66.00	20.68	61.19
	200	4	**109.86**	152.55	108.78	102.03	106.22	66.72	104.94	90.66
	200	20	**14.48**	62.08	14.26	35.50	14.04	3.51	14.10	12.82
1000	10	4	**321.14**	206.16	320.94	193.50	304.48	186.65	304.34	161.08
	10	20	**76.84**	126.40	76.66	141.40	73.80	118.86	73.72	76.98
	50	4	**261.52**	127.81	260.82	135.14	252.94	131.88	249.84	153.18
	50	20	**49.78**	116.89	49.76	188.74	48.12	54.48	48.70	74.04
	200	4	**235.50**	213.72	234.44	202.34	230.10	135.37	222.66	145.99
	200	20	38.04	132.86	**38.12**	165.15	38.00	59.74	38.02	24.07

strings $n \in \{100, 500, 1000\}$, and the alphabet size $|\Sigma| \in \{4, 12, 20\}$. This makes a total of 450 problem instances. We apply each algorithm ten times to each instance, with a time limit of 600 CPU seconds.
From preliminary experiments we noticed that the behavior of our algorithms mostly depends on the length n of the input strings. Therefore, we tuned the algorithms separately for instances with string length 100, 500, and 1000. The *irace* tool [6] was used for this purpose. For RLS, we obtained $destr = 0.2$ and $\sigma = 5$ (for $n = 100$), $destr = 0.3$ and $\sigma = 10$ (for $n = 500$), and $destr = 0.3$ and $\sigma = 20$ (for $n = 1000$). For RVNS&BS, we obtained $\alpha = 0.9$ and $\beta = 100$ (for $n = 100$), $\alpha = 0.9$ and $\beta = 200$ (for $n = 500$), and $\alpha = 0.9$ and $\beta = 200$ (for $n = 1000$). For σ of the RVNS&BS, *irace* yielded the same values as for the RLS. Moreover, EX was preferred over UB as a guidance for BS.
We additionally include here results for RVNS&Dive, which is RVNS&BS with $\beta = 1$. In this case BS reduces to a simple greedy heuristic (or dive). This was done for checking the impact of a higher beam size. Moreover, RLS&BS refers to a version of RLS in which BNH is replaced by BS with the same beam size as in RVNS&BS. Selected results are shown in Table 1. For each of the algorithms we present the avg. solution quality and the avg. median time when the best solution was found. From the results we conclude the following:

– RVNS&BS produces solutions of significantly better quality then the other algorithms on harder instances.
– The rather high beam size is apparently useful for finding approximate solutions of higher quality.



Instance: $m = 10, n = 100, |\Sigma| = 12$. Instance: $m = 50, n = 500, |\Sigma| = 4$.
Fig. 1: The impact of parameter σ on the solution quality of RVNS&BS.



$n = 500$. $n = 1000$.
Fig. 2: Improvements of solution quality when using EX instead of UB for guiding BS in RVNS&BS.

- Concerning the computation time for harder instances, the times of the RVNS&BS are usually higher than those of the RLS. It seems harder for BNH to help to improve solution quality in later stages of the RLS than for the BS in RVNS&BS.
- From Figure 1 we can see that, for smaller instances with larger alphabet sizes, stronger jumps in the search space are in essence preferred. This is because a small number of feasible solutions is distributed over the search space, and to find them it is convenient to allow large, random jumps in the search. When n is larger, choosing to do larger jumps in the space is not a good option (see the bar plot on the right). This can be explained by the fact that already the vector q that is defined by the middle of all input strings (which are generated uniformly at random) yields a promising solution, and many promising partitions are clustered around this vector. By allowing larger jumps, we move further away from this middle vector quickly, which yields usually in weaker solutions.

Figure 2 provides box plots showing the relative differences between the quality of the solutions obtained by RVNS&BS using EX and RVNS&BS using UB ($\beta = 200$). The figure shows a clear advantage of several percent when using EX over the classical upper bound UB as search guidance.

5 Conclusions and Future Work

This article provides the first heuristic approaches to solve the LCSqS problem for an arbitrary set of input strings. We applied a reduction of the LCSqS problem to a series of standard LCS problems by introducing a partitioning of the input strings as a first-level decision. Our RVNS framework explores the space of partitionings, which are then tackled by BNH and, if promising, by BS. Hereby, BS is guided by a heuristic which approximates the expected length of a LCS. Overall, RVNS&BS yields significantly better solutions that the also proposed, simpler RLS.
In future work we want to solve smaller instances of the LCSqS problem to optimality. To achieve this, we aim for an A* search that is also based on the described reduction to the classical LCS problem.

Acknowledgments. We gratefully acknowledge the financial support of the project by the Doctoral Program "Vienna Graduate School on Computational Optimization" funded by the Austrian Science Foundation (FWF) under contract no. W1260-N35.

References

1. C. Blum, M. J. Blesa, and M. López-Ibáñez. Beam search for the longest common subsequence problem. *Computers & Operations Research*, 36(12):3178–3186, 2009.
2. C. Blum and P. Festa. *Metaheuristics for String Problems in Bio-informatics*. John Wiley & Sons, 2016.
3. M. Djukanovic, G. Raidl, and C. Blum. Anytime algorithms for the longest common palindromic subsequence problem. Technical Report AC-TR-18-012, Algorithms and Complexity Group, TU Wien, 2018.
4. K. Huang, C.-B. Yang, K.-T. Tseng, et al. Fast algorithms for finding the common subsequence of multiple sequences. In *Proceedings of the International Computer Symposium*, pages 1006–1011. IEEE press, 2004.
5. T. Inoue, S. Inenaga, H. Hyyrö, H. Bannai, and M. Takeda. Computing longest common square subsequences. In *In Proceedings of CPM 2018 – 29th Annual Symposium on Combinatorial Pattern Matching*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, Dagstuhl Publishing, 2018.
6. M. López-Ibáñez, J. Dubois-Lacoste, L. Pérez Cáceres, T. Stützle, and M. Birattari. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.
7. D. Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM (JACM)*, 25(2):322–336, 1978.
8. N. Mladenović and P. Hansen. Variable neighborhood search. *Computers & Operations research*, 24(11):1097–1100, 1997.
9. S. R. Mousavi and F. Tabataba. An improved algorithm for the longest common subsequence problem. *Computers & Operations Research*, 39(3):512–520, 2012.
10. Q. Wang, M. Pan, Y. Shang, and D. Korkin. A fast heuristic search algorithm for finding the longest common subsequence of multiple strings. In *Proceedings of AAAI 2010 – 24th AAAI Conference on Artificial Intelligence*, pages 1287–1292, 2010.