# Efficient Search Algorithms for the Restricted Longest Common Subsequence Problem

MARKO DJUKANOVIĆ, Faculty of Natural Sciences and Mathematics, University of Banja Luka, Bosnia and Herzegovina
ALEKSANDAR KARTELJ, Faculty of Mathematics, University of Belgrade, Serbia
TOME EFTIMOV, Computer Systems Department, Jožef Stefan Institute, Slovenia
JAUME REIXACH, Artificial Intelligence Research Institute (IIIA-CSIC), Campus of the UAB, Spain
CHRISTIAN BLUM, Artificial Intelligence Research Institute (IIIA-CSIC), Campus of the UAB, Spain

**ACM Reference Format:**
Marko Djukanović, Aleksandar Kartelj, Tome Eftimov, Jaume Reixach, and Christian Blum. 2024. Efficient Search Algorithms for the Restricted Longest Common Subsequence Problem. 1, 1 (September 2024), 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## ABSTRACT

This paper deals with the restricted longest common subsequence (RLCS) problem, an extension of the well-studied longest common subsequence problem involving two sets of strings: the input strings and the restricted strings. This problem has applications in bioinformatics, particularly in identifying similarities and discovering mutual patterns and motifs among DNA, RNA, and protein molecules. We introduce a general search framework to tackle the RLCS problem. Based on this, we present an exact best-first search algorithm and a meta-heuristic Beam Search algorithm. To evaluate the effectiveness of these algorithms, we compare them with two exact algorithms and two approximate algorithms from the literature along with a greedy approach. Our experimental results show the superior performance of our proposed approaches. In particular, our exact approach outperforms the other exact methods in terms of significantly shorter computation times, often reaching an order of magnitude compared to the second-best approach. Moreover, it successfully solves all problem instances, which was not the case with the other approaches. In addition, Beam Search provides close-to-optimal solutions with remarkably short computation times.

## 1 INTRODUCTION

A string is a finite sequence of characters from a finite alphabet $\Sigma$. Strings are often used as a data structure, for example, in programming languages. Moreover, they play an important role as a model for DNA, RNA, and protein sequences. In the fields of stringology and bioinformatics, a pivotal task is to find meaningful and representative measures of structural similarity between molecular structures. Among several measures, one that has gathered significant attention from both practical and theoretical perspectives is the well-known *longest common subsequence* (LCS). In this context,

Authors' addresses: Marko Djukanović, marko.djukanovic@pmf.unibl.org, Faculty of Natural Sciences and Mathematics, University of Banja Luka, , Banja Luka, Bosnia and Herzegovina; Aleksandar Kartelj, aleksandar.kartelj@matf.bg.ac.rs, Faculty of Mathematics, University of Belgrade, , Belgrade, Serbia; Tome Eftimov, tome.eftimov@ijs.si, Computer Systems Department, Jožef Stefan Institute, Jamova cesta 39, Ljubljana, Slovenia, SI-1000; Jaume Reixach, jaume.reixach@iiia.csic.es, Artificial Intelligence Research Institute (IIIA-CSIC), Campus of the UAB, , Bellaterra, Spain; Christian Blum, christian.blum@iiia.csic.es, Artificial Intelligence Research Institute (IIIA-CSIC), Campus of the UAB, , Bellaterra, Spain.

a subsequence of a string $s$ is a string obtained by deleting zero or more symbols from $s$ without changing the order of the remaining symbols. Finding longest common subsequences has been a subject of study for over half a century. Given a set of input strings $S = \{s_1, \ldots, s_m\}$, the LCS problem aims to identify a common subsequence concerning all strings in $S$ of maximal length [4]. Apart from bioinformatics applications, this problem has shown to be useful in various fields, such as data compression and text processing [21].

From the very beginning, scientists have been concerned with the development of efficient algorithms for the LCS problem, especially for the case $m = 2$. Notable examples include algorithms based on the dynamic programming (DP) paradigm, such as the Hirschenberg algorithm, the Hunt-Szymanski algorithm, and the Apostolico-Crochemore algorithm; see [16, 3]. If $m$ is a fixed value, the LCS problem becomes polynomially solvable by DP, with a time complexity of $O(n^m)$, where $n$ is the length of the longest string in $S$. For arbitrarily large sets $S$, however, the problem is known to be $\mathcal{NP}$-hard [18]. Moreover, it was found that a time complexity of $O(n^m)$ is likely the tightest unless $\mathcal{P} = \mathcal{NP}$. Consequently, the existence of an efficient algorithm for the general LCS problem scenario seems unlikely. As a result, various heuristic and approximation algorithms have been proposed. In particular, beam-search-based approaches [11] and hybrid anytime algorithms [12] have proven to be very efficient. In parallel with the development of methods for solving the LCS problem, several practical variants of this problem have been introduced. These include the longest arc-preserving common subsequence problem [17, 5], the constrained LCS problem [22, 10], and the shortest common supersequence problem [19], among others.

In this study, we deal with the *restricted longest common subsequence* (RLCS) *problem*, originally introduced by Gotthilf et al. [14]. In addition to considering an arbitrary set of input strings $S$, the problem involves a set of restricted pattern strings $R = \{r_1, \ldots, r_k\}$. The objective is to find a longest common subsequence $s$ such that none of the restricted patterns $r_i \in R$ is contained as a subsequence of $s$. In their work, the authors show that the RLCS problem is $\mathcal{NP}$-hard even in the case of two input strings and an arbitrary number of restricted patterns. Moreover, they develop a DP approach for general values of $m$ and $k$. In this scenario, they find that RLCS is in FPT (Fixed-Parameter Tractable) when parameterized by the total length of the restricted patterns. In addition, the authors propose two approximation algorithms. The first ensures an approximation ratio of $1/|\Sigma|$ , while the second one guarantees a ratio of $(k_{\min} - 1)/n_{\min}$ , where $k_{\min}$ and $n_{\min}$ represent the lengths of the shortest strings in $R$ and $S$, respectively.

Independently of Gotthilf et al. [14], Chen and Chao [8] proposed a DP approach specifically for the RLCS problem with $m = 2$ and $k = 1$, achieving a time complexity of $O(|s_1| \cdot |s_2| \cdot |r_1|)$. For the same special case of the RLCS problem, Deorowicz and Grabowski [9] introduced two asymptotically faster algorithms than the conventional dynamic approach, with subcubic time complexities of $O(|s_1| \cdot |s_2| \cdot |r_1| / \log(|s_1|))$ and $O(|s_1| \cdot |s_2| \cdot |r_1| / \log^{\frac{3}{2}}(|s_1|))$ by utilizing well-designed internal data structures. Farhana and Rahman [13] proposed a finite automata-based approach to solve the general RLCS problem with a time complexity of $O(|\Sigma|(\mathcal{R} + m) + nm + |\Sigma|\mathcal{R}n^k)$, where $\mathcal{R} = O(n^m)$ denotes the size of the resulting automaton. The experimental results presented in that paper emphasize the superiority of the automata approach over the classical DP approach. The contributions of this paper are as follows:

(1) *An error-free DP approach.* We present a DP approach to the RLCS problem that handles an arbitrary number of input strings and restricted pattern strings. In particular, our approach avoids significant flaws identified in the DP approach from [14].

(2) *General search framework.* We design a general search framework for solving the RLCS problem, which serves as the core of an exact A$^*$ algorithm and a Beam Search approach. The search process in these methods is guided by utilizing the tightest known upper bounds for the classical LCS problem with arbitrary input strings.

(3) *In-depth comparative analysis.* We perform a thorough comparison of all seven approaches (three proposed in this paper and four from the literature) for the RLCS problem, using a comprehensive set of available instances for evaluation.

Our A$^*$ search shows a clear superiority over the other two exact approaches from the literature. It excels in terms of the number of optimally solved instances, while it provides significantly shorter runtimes, often by an order of magnitude, compared to the best exact approaches from the literature.

### 1.1 Preliminaries

The length of a string $s$ is denoted by $|s|$, whereas $s[i], 1 \leq i \leq |s|$, stands for its $i$-th character. It should be noted that—in this paper—the position of the leading character is indexed with 1. For two integers $i, j \leq |s|$, $s[i, j]$ denotes a continuous part of the string $s$ that begins with the character at position $i$ and ends with the character at position $j$. If $i = j$, the single-character string $s[i] = s[i, i]$ is given, or if $i > j$, the empty string $\varepsilon$ is assigned.

For a left position vector $p^L = (p^L_1, \ldots, p^L_m),\ 1 \leq p^L_i \leq |s_i|,\ i = 1, \ldots, m$, we denote by $S[p^L]$ the set of suffix input strings associated with the respective coordinates of this vector, i.e., $S[p^L] := \{s_i[p^L_i, |s_i|] \mid i = 1, \ldots, m\}$. Finally, we define $p^L - \mathbf{1} := (p^L_i - 1, \ldots, p^L_m - 1)$ or, more generally, for two vectors $\mathbf{p}, \mathbf{q} \in \mathbb{N}^m$, $\mathbf{p} - \mathbf{q} := (p_1 - q_1, \ldots, p_m - q_m)$.

A complete RLCS problem instance is denoted as a pair $(S, R)$ of two sets of strings, where $S$ contains the input strings and $R$ the restricted pattern strings. For two integer vectors $\mathbf{p} \in \mathbb{N}^m$ and $\mathbf{q} \in \mathbb{N}^k$, a sub-problem (sub-instance) of the initial problem instance with respect to these two (left) positional vectors is denoted by $(S[\mathbf{p}], R[\mathbf{q}])$.

The remaining sections of the work are organized as follows. Section 2 presents a DP approach for solving the RLCS problem. In particular, this section addresses and corrects the shortcomings of DP proposed in [14]. Section 3 presents a simple and naive greedy algorithm as an alternative method for solving the RLCS problem. Section 4 builds on the DP approach and derives a general search framework. In particular, we propose efficient A$^*$ and Beam Search algorithms. The practical comparison between our approaches and those in the literature is detailed in Section 5 through a thorough experimental evaluation. The paper concludes in Section 6, with possible directions for future research.

## 2 THE DP APPROACH FOR THE RLCS PROBLEM

When considering DP, it is crucial to determine whether the problem under study has the optimal substructure property. The question is if the problem can be broken down into smaller parts, such that solving these smaller subproblems leads to an optimal solution to the overall problem. Based on this concept and using the well-known DP algorithm for the LCS problem with an arbitrary number of input strings, we derive the DP approach for the RLCS problem as follows. Let $(S, R)$ be a RLCS problem instance, $\mathbf{p} \in \mathbb{N}^m$ and $\mathbf{l} \in \mathbb{N}^k$ with $1 \leq p_i \leq |s_i|$ for $i \in \{1, \ldots, m\}$ and $1 \leq l_j \leq |r_j|$ for $j \in \{1, \ldots, k\}$. We denote by $\mathrm{RLCS}[\mathbf{p}; \mathbf{l}]$, the length of a longest common subsequence of $S[\mathbf{p}] = \{s_1[1, p_1], \ldots, s_m[1, p_m]\}$ with no string from $R_l = \{r_1[1, l_1], \ldots, s_k[1, l_k]\}$ as a subsequence. We distinguish the following non-trivial cases:

**Case** 1: $s_i[p_i] = \sigma \in \Sigma$ for every $i \in \{1, \ldots, m\}$. Let us denote $J := \{j \in \{1, \ldots, k\} \mid r_j[l_j] = \sigma\}$. In the event that letter $\sigma$ does not contribute to the optimal solution of this subproblem, the relevant smaller subproblem is $(S_{\mathbf{p}-\mathbf{1}}, R_{\mathbf{l}})$ or $(S_{\mathbf{p}-\mathbf{1}}, R_{\mathbf{l}^*})$ if it does contribute, where $l^*_j = l_j - 1$ for $j \in J$ and $l^*_j = l_j$ otherwise. There are two sub-cases for this case:

(a) $\mathrm{RLCS}[\mathbf{p}; \mathbf{l}] = \mathrm{RLCS}[\mathbf{p} - \mathbf{1}; \mathbf{l}]$ if there is an index $j \in J$ such that $l_j = 1$;

(b) $\mathrm{RLCS}[\mathbf{p}; \mathbf{l}] = \max\{\mathrm{RLCS}[\mathbf{p} - \mathbf{1}; \mathbf{l}], \mathrm{RLCS}[\mathbf{p} - \mathbf{1}; \mathbf{l}^*] + 1\}$, otherwise.

**Case** 2: $s_{i_1}[p_{i_1}] \neq s_{i_2}[p_{i_2}]$ for some $i_1, i_2 \in \{1, \dots, m\}$. The same is done as in the recursion for the LCS problem. That is, the recursion is given by

$$\mathrm{RLCS}[\mathbf{p}; \mathbf{l}] = \max\{\mathrm{RLCS}[\mathbf{p} - \mathbf{e}_i; \mathbf{l}] \mid i = 1, \dots, m\}.$$

In [14], the authors did not provide a correct derivation for Case 1 (see Section 4 of the aforementioned paper). In particular, they failed to include the +1 term ("plus one") for the Case 1b and did not distinguish between the Cases 1a and 1b. This omission led to incorrect calculations in the experimental evaluations of subsequent papers from the literature, e.g. [13].

## 3 GREEDY ALGORITHM

The greedy algorithm for the RLCS problem uses a constructive approach that employs the best-next heuristic. It is an extension of the greedy approach used for the LCS problem [6]. At each step, it consists of appending the feasible letter with the best greedy value to the current partial solution. The algorithm starts with an empty solution $s^p = \varepsilon$ and processes the input strings from the far left (pointing to the characters at position 1 for strings from both sets, $S$ and $R$) towards their right endpoints. The pointers for this process are denoted as $\mathbf{p}$ and $\mathbf{l}$ for each set, respectively. Next, the set $\Sigma^{cand}$ of those letters that occur in all suffix strings $s_i[p_i, |s_i|]$ is considered. From this set, the algorithm retains only those letters which, when appended to $s^p$, do not cause a violation of the restriction that the whole $r_i$ becomes a subsequence of the extended $s^p$. We denote the filtered set with $\Sigma^{feas}$. The algorithm selects a letter $a \in \Sigma^{feas}$ with the smallest value calculated by:

$$g(\mathbf{p}, \mathbf{l}, a) = \sum_{i=1}^{m} \frac{Succ[p_i]_{i,a} - p_i}{|s_i| - p_i + 1} + \sum_{j=1}^{k} \frac{|\Sigma|}{|r_j| - l_j - I_{r_i[l_i]=a} + 1} \tag{1}$$

where $Succ[p_i]_{i,a}$ denotes the smallest position in $s_i$ greater or equal to $p_i$, at which letter $a$ appears (in this way we eliminate suboptimality at the local level of decisions) and $I$ represents the indicator function. Let us denote the best letter according to these $g$-values by $a^*$. Then, the following updates are performed: $s^p = s^p \cdot a^*$, $p_i = Succ[p_i]_{i,a^*} + 1$ and $l_j = l_j + I_{r_j[\mathbf{l}_j]=a^*}$ for every $i \in \{1, \dots, m\}$ and $j \in \{1, \dots, k\}$ respectively. The following iterations proceed in the same way until $\Sigma^{feas} = \emptyset$. Finally, the constructed solution is returned.

Note that, at each iteration, the greedy criterion (1) favors the feasible letters whose selection excludes smaller parts of the input strings from being considered, and who make the partial solution less close to having one of the restricted strings as a subsequence.

## 4 THE PROPOSED EFFICIENT SEARCH METHODS

In this section, we first describe a general search framework for the RLCS problem based on the definition of a state graph. Afterwards, we derive an exact and a heuristic search algorithm, both based on this framework.

### 4.1 The General Search Framework

The state graph is the environment of our proposed algorithms. Its inner nodes represent partial solutions while its sink nodes represent complete solutions. Moreover, edges between nodes represent extensions of partial solutions. The state graph $G = (V, E)$ of an RLCS problem instance $(S, R)$ is defined as follows.

We say that a partial solution—that is, a common subsequence $s^v$ of the strings in $S$ that does not contain any string from $R$ as as subsequence—*induces* a node $v = (p^{L,v}, l^v, u^v) \in V$ if:

- $|s^v| = u^v$
- $s^v$ is a subsequence of all $s_i[1, p_i^{L,v} - 1], i \in \{1, \ldots, m\}$ and $p_i^{L,v} - 1$ is the smallest index that satisfies this property.
- $s^v$ contains none of the prefix strings $r_j[1, l_j^v]$ as its subsequence whereas $r_j[1, l_j^v - 1], j \in \{1, \ldots, k\}$ are all included.

Additionally, there is an edge between nodes $v_1 = (p^{L,v_1}, l^{v_1}, u^{v_1})$ and $v_2 = (p^{L,v_2}, l^{v_2}, u^{v_2})$ labelled with a letter $a \in \Sigma$, denoted by $t(v_1, v_2) = a$, if:

- $u^{v_1} + 1 = u^{v_2}$
- The partial solution inducing node $v_2$ is obtained by appending the letter $a$ to the partial solution inducing node $v_1$.

Each edge of the state graph of an RLCS problem instance has weight one and (as mentioned above) a label denoting the letter used for the extension.

To extend a node $v$ and determine its successor nodes (children), it is necessary to identify the letters that can feasibly extend the partial solution $s^v$ represented by $v$. First, all letters occurring in each string from the set $S[p^{L,v}]$ are identified. Then, the letters that cause a violation of the restrictions are removed, i.e., letters that cause one of the restricted patterns $r_i \in R$ to be a subsequence of the partial solution obtained by extending $s^v$ with this letter. In addition, dominated letters are also omitted. A letter $a$ is said to *dominate* the letter $b$ (i.e., $b$ is dominated by $a$) if $Succ[p^{L,v}i]_{i,a} \leq Succ[p^{L,v}i]_{i,b}$ for every $i \in \{1, \ldots, m\}$ and $r_j[l_j^v] \notin \{a, b\}$ for all $j \in \{1, \ldots, k\}$. We denote the set of non-dominated feasible letters to extend the partial solution of a node $v$ by $\Sigma_v^{nd}$.

For a letter $a \in \Sigma_v^{nd}$, the corresponding successor node $w = (p^{L,w}, l^w, u^w)$ of $v$ is constructed as follows.

- $u^w = u^v + 1$, i.e., the partial solution of node $v$ derives the partial solution of node $w$ by appending the letter $a$ to it: $s^w = s^v \cdot a$.
- $l_j^w = l_j^v + 1$ if $r_j[l^v] = a$ or $l_j^w = l_j^v$ otherwise.
- For the (left) position vectors, $p_i^{L,w} = Succ[p_i^{L,v}]_{i,a} + 1$.

Notably, the aforementioned data structure *Succ* can be preprocessed before the construction of an RLCS state graph is started. In this way, finding suitable position vectors of a child node is addressed in time $O(m)$.

The *root* (initial) node $r = ((1, \ldots, 1), (1, \ldots, 1), 0)$ corresponds to the empty solution $s^r = \varepsilon$, which is trivially feasible and induces the complete problem instance $(S, R)$.

We say that a node $v$ is *complete* if $\Sigma_v^{nd} = \emptyset$. These are the nodes that have no child nodes (successors). Note that (partial) solutions induced by complete nodes are candidates for optimal solutions. In this context, note that optimal solutions are end-points of the longest paths from the root node $r$ to complete nodes. Since the RLCS problem is $\mathcal{NP}$-hard, generating the entire state graph is generally infeasible as its size grows exponentially with the instance size. Consequently, our algorithm proposals generate and visit nodes on the fly, making intelligent decisions to prioritize the exploration of more promising nodes, as explained in the following sections. This section concludes by showing the complete state graph of an instance in Figure 1.

### 4.2 A* Search Algorithm

A* search [15] is an exact, informed search algorithm that follows the best-first search strategy for path-finding. It is the most widely used path-finding algorithm, being of high relevance in many fields e.g., in video games, in string matching and parsing, and others. The idea of the algorithm is to always expand the most promising nodes first. To

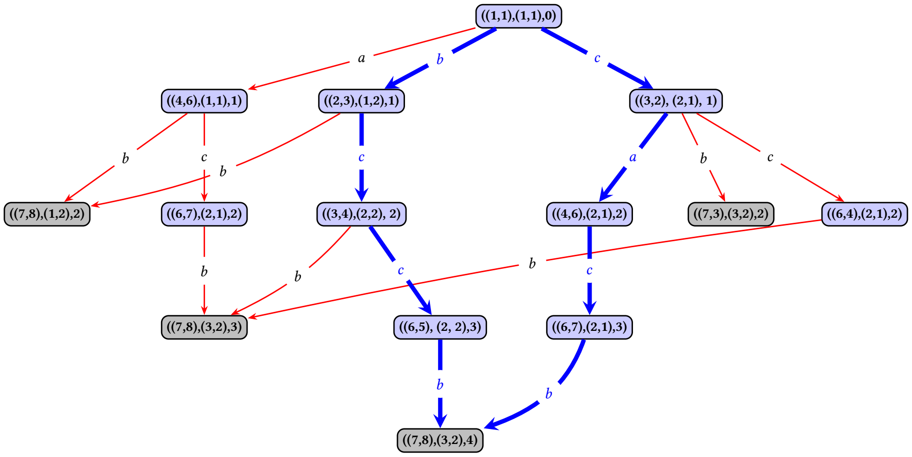

Fig. 1. Example of the full state graph in the form of a directed acyclic graph for the problem instance ($S = \{s_1 = \text{bcaacbb}, s_2 = \text{cbccacb}\}$, $R = \{\text{cbb}, \text{ba}\}$). It contains four complete nodes (light grey background). The two paths from $((7,8),(3,2),4)$ to the root node (in blue) are the longest paths in the graph. Hence, they represent two optimal solutions for this problem instance, bccb and cacb respectively.

rank the quality of nodes, a scoring function $f(v) = g(v) + h(v)$ is used. If the goal is to find longest paths, as in our case, functions $g()$ and $h()$ are defined as follows:

- $g(v)$ is the length of the longest path currently known from the root $r$ to $v$.
- $h(v)$ is a heuristic function for estimating the length of the longest path from $v$ to a complete (goal) node.

Note that A* works on a dynamically generated directed acyclic graph and in practice rarely examines all nodes. It has the advantageous ability to merge multiple nodes into one, which, as explained below, leads to considerable memory savings. To set up an efficient A* search for the RLCS problem, two important data structures are used:

- A hash map $N$ with keys of the form $(p^{L,v}, l^v)$, where the corresponding value is the longest partial solution that induces a node with these vectors, indicating the sub-instance $(S[p^{L,v}], R[l^v])$ to be solved. This data structure efficiently checks whether or not a node with the same key values has already been visited.
- A priority queue $Q$ that contains open (not yet expanded) nodes that are prioritized based on their $f$-values. This structure facilitates the efficient retrieval of the most promising node.

Next, the heuristic function $h()$ must be defined. For this purpose, we opt for using the tightest known upper bound for the LCS problem. Note that any upper bound for an LCS problem instance is also an upper bound for a corresponding RLCS problem instance obtained by adding a set $R$ of restricted strings. The upper bound we used is the minimum of two known upper bounds, denoted as $\text{UB}_1$ and $\text{UB}_2$, that is, $\text{UB} = \min\{\text{UB}_1, \text{UB}_2\}$. For detailed information, we refer to [6] and [23]. The upper bound $\text{UB}_1$ determines for every letter an upper bound on the number of times this letter is potentially included in an optimal solution and then returns the sum of these values. On the other hand, $\text{UB}_2$ repeatedly applies dynamic programming to the LCS problem of two input strings for constructing the upper bound. It is important to note that there may be multiple nodes with the same $f$-value. In such cases, ties are resolved by favoring those with a higher $u^v$ value.

The algorithm starts by initializing the root node $r$, which is then added to both $N$ and $Q$. At the same time, the best solution $s^{best}$ is initialized to the empty string $\varepsilon$. In each iteration, the most promising node $v$ from the beginning of the priority queue $Q$ is selected. If $f(v)$ is less than or equal to $|s^{best}|$, the search is terminated and the proven optimal solution $s^{best}$ is returned. If the node $v$ is complete, it is checked whether $u^v$ is greater than $|s^{best}|$. If this is the case, $s^{best}$ is chosen for inducing node $v$, which is reconstructed by traversing from $v$ back to the root node $r$ and reading the letters along the transitions. The algorithm then proceeds to the next iteration. If none of the above conditions are met, the node $v$ is expanded in all possible ways by creating its children. For each child $w$, it is checked whether $(p^{L,w}, l^w)$ is already contained in $N$. If not, $w$ is added to both $N$ and $Q$. Otherwise, it is checked whether a new best path from the root node $r$ to any node associated with $(p^{L,w}, l^w)$ has been found. In the case of a positive answer, the information in $N$ is updated and the priority of this node is changed in $Q$. In the case of a negative answer, $w$ is declared irrelevant and thus skipped in the next iteration of the algorithm.

### 4.3 Beam Search Algorithm

Beam Search (BS) [1] is a heuristic search algorithm that works in a "breadth-first-search" (BFS) manner, expanding nodes at each level, with a limitation on the number of nodes to be expanded. More precisely, up to $\beta > 0$ of the most promising nodes at each level are selected to generate the nodes of the next level. Parameter $\beta$ ensures that the size of the BS tree remains polynomial with respect to the size of the problem instance, which makes this method applicable to various complex problems. BS is widely used in fields such as packing [1], scheduling [20], and bioinformatics [7], among others.

In addition to parameter $\beta > 0$, the effectiveness of BS strongly depends on a heuristic function $h()$ used to evaluate the "quality" of each node. The choice of $h()$ is typically a problem-specific decision. For our purpose, we opt for the tightest upper bound (UB) for the LCS problem already introduced in the previous section. The BS approach for the RLCS problem works as follows. The root node $r$ is first generated and included in the beam $B$ (i.e. $B = \{r\}$) and $l_{best}$ is initialized to 0. Then, the main loop is entered. All nodes in the beam $B$ are expanded in all possible ways. The resulting child nodes are stored in $V_{ext}$, while $l_{best}$ is increased by one. The nodes from $V_{ext}$ are then sorted in descending order according to their $h()$-values. $B$ is emptied and the best $\beta > 0$ nodes are then added to it for the subsequent level. These steps are repeated as long as the beam $B$ is not empty, in which case, the algorithm is stopped, and the best RLCS solution $s_{best}$ of length $l_{best}$ is returned.

## 5 EXPERIMENTAL EVALUATION

This section presents a comprehensive experimental evaluation comparing three exact competitors: the A* search proposed in Section 4.2 and two existing approaches from the literature, namely, the automaton approach presented in [13] (denoted as Automaton) and the corrected version of DP initially proposed in [14], as provided in Section 2 (denoted as Dp). Besides the exact approaches, we also compare four heuristic methods: the Beam Search (BS) proposed in Section 4.3, the greedy approach from Section 3 (denoted as Greedy) and two approximation algorithms (denoted as Approx1 and Approx2) from the literature, proposed in [14]. As mentioned above, the source code for the Automaton approach in its original form was obtained directly from the authors of [13], along with the provided problem instances. The remaining six approaches were implemented in C++ under Ubuntu 20.0 and compiled with gcc 13.1.0 with optimization level $Ofast$. All experiments were conducted in single-threaded mode on an Intel Xeon E5-2640 with 2.40GHz and 16 GB of memory.

The experimental evaluation employs three sets of benchmark instances denoted as Random, Real and Substr-Ec. In the benchmark set Random, instances are divided into six groups based on the values of $m$ and $k$, which denote the amount of input and restricted strings respectively. Five groups comprise 10 instances each, while one group comprises a single instance, as provided by the original authors. Thus, there are a total of 51 randomly generated problem instances in this set.

For the benchmark set Real, we use four real-world instances presented in [13]. Finally, to extend the scope of our experiments, we use randomly generated instances designed for a variant of the RLCS problem, namely the substring-exclusion constrained LCS problem [2]. This set consists of two sets of 10 randomly generated instances each, resulting in a total of 20 instances. We refer to this benchmark set as Substr-Ec.

**Parameter setting.** Only Beam search (BS) requires the setting of one of its parameters, namely the beam width $\beta > 0$. After a preliminary experimental evaluation, we found that setting $\beta = 100$ leads to a favorable compromise between the quality of the final solutions and the required computation times.

**Description of the results.** Table 1 shows the numerical results for the three exact algorithms applied to the benchmark set Random. Every row presents the average results for one of the instance groups. The first four columns describe the properties of each group, specifying the number of input strings ($m$), the number of pattern strings ($k$), the length of all input strings ($|s_0|$) and the length of all pattern strings ($|r_0|$). Moreover, $\overline{|s|}$ denotes the average length of the optimal solutions for every instance group.

For each algorithm and instance group, we present the amount (*#opt*) of instances that were solved to optimality and the average runtime required for obtaining these optimal solutions. Symbol "–" denotes that the respective algorithm was not able to provide any optimal solution due to problems during runtime, such as reaching time or memory limits.

The following conclusions can be drawn from Table 1.

(1) A$^*$ search and Dp can find a provably optimal solution for all (51) problem instances within the given time and memory constraints. For the Automaton approach, this was possible for 32 problem instances, facing memory limitations for the remaining ones.

(2) In terms of runtime, the A$^*$ search emerges as the clear winner, as it shows a significantly faster performance in comparison to Dp and Automaton, often outperforming them by an order of magnitude. It achieves consistently short average runtimes, all below one second.

(3) The runtimes of Dp increase rapidly with increasing $m$, while it seems to be difficult for the Automaton approach to handle instances with larger $k$ values (memory problems are notable for $k \geq 3$).

Table 2 shows the averaged numerical results for the four heuristic approaches applied to the benchmark set Random. The values are averages over the instances within each group (rows). As in Table 1, the first four columns correspond to the instance groups. The table is then divided into four blocks, each consisting of two columns. These blocks correspond to the approaches Bs, Greedy, Approx1 and Approx2 respectively. Two values are provided for each algorithm: the average quality of the best-found solutions ($\overline{|s|}$) and the average runtime for obtaining these best solutions ($\bar{t}[s]$).

The following conclusions can be drawn from Table 2.

- The most effective heuristic approach, both in terms of solution quality and time efficiency, is Bs. It outperforms the second-best approach, Greedy, by a significant margin. With runtimes of around 0.1 seconds on average, Bs achieves a high solution quality and remains within 7% of the optimal results.

Table 1. Comparisons between the exact approaches on benchmark set RANDOM. All pattern strings in the instances of the same group are of an equal length $|r_0|$. The same also holds for the length of the input strings, which are of length $|s_0|$.

| $m$ | $k$ | $\|s_0\|$ | $\|r_0\|$ | $\overline{\|s\|}$ | A* | | DP | | AUTOMATON | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | #opt | $\bar{t}[s]$ | #opt | $\bar{t}[s]$ | #opt | $\bar{t}[s]$ |
| 2 | 1 | 200 | 3 | 65.0 | 1 | 0.0 | 1 | 0.0 | 1 | 0.3 |
| 2 | 3 | 250 | 8 | 88.2 | 10 | 0.0 | 10 | 6.8 | 1 | 2.1 |
| 2 | 4 | 250 | 6 | 87.1 | 10 | 0.0 | 10 | 18.5 | 0 | – |
| 3 | 1 | 200 | 10 | 46.0 | 10 | 0.6 | 10 | 35.3 | 10 | 1.2 |
| 3 | 2 | 200 | 3 | 43.8 | 10 | 1.0 | 10 | 44.9 | 10 | 1.9 |
| 4 | 1 | 75 | 3 | 12.9 | 10 | 0.0 | 10 | 92.4 | 10 | 0.2 |

Table 2. Comparisons between the heuristic approaches on benchmark set RANDOM.

| $m$ | $k$ | $\|s_0\|$ | $\|r_0\|$ | $\overline{\|s\|}$ | BS | | GREEDY | | APPROX1 | | APPROX2 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | $\overline{\|s\|}$ | $\bar{t}[s]$ | $\overline{\|s\|}$ | $\bar{t}[s]$ | $\overline{\|s\|}$ | $\bar{t}[s]$ | $\overline{\|s\|}$ | $\bar{t}[s]$ |
| 2 | 1 | 200 | 3 | 65.0 | 64.0 | 0.1 | 62.0 | 0.0 | 13.0 | 0.0 | 2.0 | 0.0 |
| 2 | 3 | 250 | 8 | 88.2 | 88.2 | 0.1 | 86.2 | 0.0 | 15.9 | 0.0 | 7.0 | 0.0 |
| 2 | 4 | 250 | 6 | 87.1 | 87.1 | 0.1 | 85.0 | 0.0 | 16.4 | 0.0 | 5.1 | 0.0 |
| 3 | 1 | 200 | 10 | 46.0 | 44.9 | 0.1 | 38.2 | 0.0 | 11.2 | 0.0 | 9.0 | 7.0 |
| 3 | 2 | 200 | 3 | 43.8 | 41.0 | 0.1 | 36.6 | 0.0 | 11.5 | 0.0 | 2.0 | 7.0 |
| 4 | 1 | 75 | 3 | 12.9 | 12.9 | 0.0 | 11.5 | 0.0 | 4.2 | 0.0 | 2.0 | 20.9 |

- In contrast, the two approximation algorithms APPROX1 and APPROX2 are significantly behind the rest. In particular, the computation times of APPROX2 grow rapidly with increasing values of $m$, which indicates that using DP for the LCS problem is slow for larger instances.

Table 3 shows the numerical results for all seven approaches applied to four real cases. This table is structured as follows. The first four columns show the characteristics of the problem instances, including the number of input strings ($m$), the number of restricted strings ($k$), the length pairs for the shortest and longest input strings ($n$), and the length pairs for the shortest and longest restricted pattern strings ($p$). The next seven blocks, each consisting of two columns, show the results for the aforementioned approaches. In particular, the two columns show the delivered solution quality ($|s|$) and the corresponding runtime ($t[s]$).

Table 3. Comparison between all seven approaches for benchmark set REAL.

| $m$ | $k$ | $n$ | $p$ | A* | | AUTOMATON | | DP | | APPROX1 | | APPROX2 | | BS | | GREEDY | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | $\overline{\|s\|}$ | $\bar{t}[s]$ | $\overline{\|s\|}$ | $\bar{t}[s]$ | $\overline{\|s\|}$ | $\bar{t}[s]$ | $\overline{\|s\|}$ | $\bar{t}[s]$ | $\overline{\|s\|}$ | $\bar{t}[s]$ | $\overline{\|s\|}$ | $\bar{t}[s]$ | $\overline{\|s\|}$ | $\bar{t}[s]$ |
| 4 | 2 | (141, 146) | (1, 2) | 48 | 0.1 | 48 | 3.4 | – | – | 15 | 0 | – | – | 48 | 0.1 | 27 | 0 |
| 3 | 2 | (255, 293) | (1, 4) | 74 | 0.4 | 74 | 8.7 | 74.0 | 69.7 | 29 | 0 | 0.0 | 26.9 | 72 | 0.1 | 57 | 0 |
| 5 | 4 | (98, 123) | (1, 4) | 16 | 0.1 | 16 | 2.0 | – | – | 9 | 0 | – | – | 16 | 0.0 | 15 | 0 |
| 3 | 2 | (124, 185) | (4, 5) | 37 | 0.1 | 37 | 0.5 | 37.0 | 19.3 | 10 | 0 | 3.0 | 2.8 | 32 | 0.0 | 30 | 0 |

The following conclusions can be drawn from the numerical results in Table 3.

- A* search and AUTOMATON provide optimal solutions for all four instances. A* outperforms AUTOMATON by being approx. one order of magnitude faster. Conversely, the DP approach only achieved optimal solutions for two instances and reached memory limits for the remaining two cases.

- Among the heuristic approaches, Bs stands out as the superior choice, providing optimal solutions for two out of four instances. It is closely followed by Greedy, which provides solutions of reasonable quality with exceptionally short computation times. On the other hand, both approximation algorithms, Approx1 and Approx2, provide impractical results. In particular, Approx2 does not provide any results for instances with $m \geq 4$.

Table 4 reports the numerical results for all seven approaches on the benchmark set Substr-Ec. This table is structured similarly to Table 2 with a small difference in the instance description. In particular, in addition to $m$ and $k$, instances are also described by their *index* number (there are 10 (indexed) instances in each of the two groups).

The following conclusions can be derived from Table 4.

- A* and Dp achieve optimal solutions for all 20 instances. However, A* shows again a notable advantage in terms of computation times (a difference of two orders of magnitude). The Automaton approach encounters difficulties in terms of memory usage for the instances with $k = 3$, as already observed for the Random benchmark set.
- Regarding the heuristic approaches, Bs proves to be an outstanding performer as it provides optimal solutions for all 20 problem instances with remarkably short computation times (at most 0.1 seconds). Greedy is able to derive optimal solutions for all (10) instances with $k = 2$. However, in case of the instances with $k = 3$, the performance of Greedy deteriorates as an optimal solution is only produced in 3 (out of 10) cases.
- In contrast, the other two algorithms provide extremely fast solutions, but they deviate significantly from the known optimal solutions.

Table 4. Comparison between the approaches on benchmark set Substr-Ec. The instances comprised in the group $m = 2, k = 1$ have both input strings of length 100 and pattern strings of length 40. The instances comprised in the group $m = 2, k = 3$ have both input strings of length 250 and all pattern strings of length 8.

| $m$ | $k$ | *ind.* | A* | | Automaton | | Dp | | Approx1 | | Approx2 | | Bs | | Greedy | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | $\bar{s}$ | $\bar{t}[s]$ | $\bar{s}$ | $\bar{t}[s]$ | $\bar{s}$ | $\bar{t}[s]$ | $\bar{s}$ | $\bar{t}[s]$ | $\bar{s}$ | $\bar{t}[s]$ | $\bar{s}$ | $\bar{t}[s]$ | $\bar{s}$ | $\bar{t}[s]$ |
| 2 | 1 | 0 | 47 | 0.0 | 47.0 | 0.2 | 47 | 0.1 | 7 | 0 | 39 | 0.0 | 47 | 0.1 | 47 | 0.0 |
| | | 1 | 42 | 0.0 | 42.0 | 0.2 | 42 | 0.1 | 6 | 0 | 39 | 0.0 | 42 | 0.0 | 42 | 0.0 |
| | | 2 | 44 | 0.0 | 44.0 | 0.2 | 44 | 0.1 | 6 | 0 | 39 | 0.0 | 44 | 0.1 | 44 | 0.0 |
| | | 3 | 44 | 0.0 | 44.0 | 0.2 | 44 | 0.1 | 8 | 0 | 39 | 0.0 | 44 | 0.0 | 44 | 0.0 |
| | | 4 | 46 | 0.0 | 46.0 | 0.2 | 46 | 0.1 | 7 | 0 | 39 | 0.0 | 46 | 0.0 | 46 | 0.0 |
| | | 5 | 47 | 0.0 | 47.0 | 0.2 | 47 | 0.1 | 7 | 0 | 39 | 0.0 | 47 | 0.1 | 47 | 0.0 |
| | | 6 | 45 | 0.0 | 45.0 | 0.2 | 45 | 0.1 | 7 | 0 | 39 | 0.0 | 45 | 0.0 | 45 | 0.0 |
| | | 7 | 48 | 0.0 | 48.0 | 0.2 | 48 | 0.1 | 7 | 0 | 39 | 0.0 | 48 | 0.1 | 48 | 0.0 |
| | | 8 | 43 | 0.0 | 43.0 | 0.2 | 43 | 0.1 | 8 | 0 | 39 | 0.0 | 43 | 0.0 | 43 | 0.0 |
| | | 9 | 45 | 0.0 | 45.0 | 0.2 | 45 | 0.1 | 7 | 0 | 39 | 0.0 | 45 | 0.1 | 45 | 0.0 |
| 2 | 3 | 0 | 90 | 0.0 | - | - | 90 | 7.0 | 16 | 0 | 7 | 0.0 | 90 | 0.1 | 87 | 0.0 |
| | | 1 | 84 | 0.0 | - | - | 84 | 8.1 | 14 | 0 | 7 | 0.0 | 84 | 0.1 | 83 | 0.0 |
| | | 2 | 87 | 0.0 | - | - | 87 | 6.5 | 16 | 0 | 7 | 0.0 | 87 | 0.1 | 87 | 0.0 |
| | | 3 | 91 | 0.0 | - | - | 91 | 5.6 | 16 | 0 | 7 | 0.0 | 91 | 0.1 | 90 | 0.0 |
| | | 4 | 89 | 0.0 | - | - | 89 | 7.4 | 16 | 0 | 7 | 0.0 | 89 | 0.1 | 85 | 0.0 |
| | | 5 | 87 | 0.0 | - | - | 87 | 7.5 | 15 | 0 | 7 | 0.0 | 87 | 0.1 | 83 | 0.0 |
| | | 6 | 88 | 0.0 | - | - | 88 | 6.4 | 17 | 0 | 7 | 0.0 | 88 | 0.1 | 87 | 0.0 |
| | | 7 | 91 | 0.0 | - | - | 91 | 9.5 | 17 | 0 | 7 | 0.0 | 91 | 0.1 | 91 | 0.0 |
| | | 8 | 89 | 0.0 | - | - | 89 | 5.9 | 15 | 0 | 7 | 0.0 | 89 | 0.1 | 83 | 0.0 |
| | | 9 | 86 | 0.0 | - | - | 86 | 7.0 | 17 | 0 | 7 | 0.0 | 86 | 0.1 | 86 | 0.0 |

## 5.1 Statistical Analysis

To determine statistical differences between the results of the seven competing approaches, we conducted a pairwise statistical analysis using a one-sided Wilcoxon rank-sum test, as shown in Figure 2. The null hypothesis, asserting that the first algorithm yields superior (larger) results compared to the second algorithm, was assessed at a significance level ($\alpha$) of 0.05. For instance, at the point of intersection of A* search (on the $x$-axis) and the Greedy approach (on the $y$-axis), a $p$-value resulting from a one-sided Wilcoxon rank-sum test between the outcomes of these two algorithms is provided (with a value of 1.0). This indicates substantial evidence in favor of retaining the null hypothesis over the alternative. Hence, we infer that the A* search results statistically outperform those of the Greedy approach. The solution quality achieved by A* is superior across all 75 instances compared to other competitors. In addition, remember that the running times of A* are significantly shorter than those of Dp and Automaton approaches.

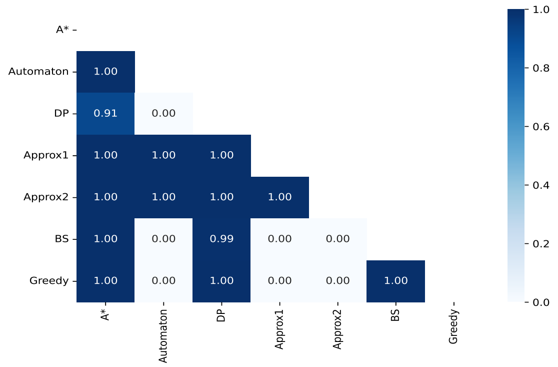

Fig. 2. Post-hoc pairwise statistical comparison between the seven competing approaches on all problem instances using the (one-sided) Wilcoxon rank-sum test.

## 6 CONCLUSIONS AND FUTURE WORK

This work has dealt with the RLCS problem, an extension of the well-known LCS problem. First, we corrected a previously proposed dynamic programming approach for the RLCS problem. Then, a comprehensive search framework based on the concept of a state graph was introduced. Using this framework, we developed both an exact A* search algorithm and a heuristic Beam Search. Our results, validated on 75 problem instances from the literature, obtained from the authors of [13], emphasize the effectiveness of the proposed methods. The exact approach showed strong performance by being the only algorithm to provide provably optimal solutions for all instances, with computation times being an order of magnitude shorter than those of the best exact competitor from the literature. In addition, the Beam Search showed promising results by providing optimal solutions for many instances employing remarkably short computation times.

For future research, it would be interesting to explore the limits of our proposed exact approach by testing it on much larger instances. Moreover, the known scalability and robustness of Beam Search present considerable potential and also deserve further investigation. Improving the heuristic guidance for Beam Search is another research direction; possible ideas include developing a probabilistic search model or incorporating machine learning techniques.

**Acknowledgments.** The research of M. Djukanović is partially supported by the Ministry for Scientific and Technological Development and Higher Education of the Republic of Srpska, B&H in the course of the bilateral research project between B&H and Slovenia entitled "Theoretical and computational aspects of some graph problems with the application to graph network information spreading" and the COST Action ROAR-NET under no. CA22137. A. Kartelj was supported by grant 451-03-47/2023-01/200104 funded by the Ministry of Science Technological Development and Innovations of the Republic of Serbia. J. Reixach and C. Blum are supported by grants TED2021-129319B-I00 and PID2022-136787NB-I00 funded by MCIN/AEI/10.13039/501100011033. The authors would like to thank the Compute Cluster Unit of the Institute of Logic and Computation at the Vienna University of Technology for providing computing resources for this research project.

## REFERENCES

[1] Akeb, H., Hifi, M., M'Hallah, R.: A beam search algorithm for the circular packing problem. Computers & Operations Research **36**(5), 1513–1528 (2009)

[2] Ann, H.Y., Yang, C.B., Tseng, C.T.: Efficient polynomial-time algorithms for the constrained lcs problem with strings exclusion. Journal of Combinatorial Optimization **28**(4), 800–813 (2014)

[3] Apostolico, A., Guerra, C.: The longest common subsequence problem revisited. Algorithmica **2**, 315–336 (1987)

[4] Bergroth, L., Hakonen, H., Raita, T.: A survey of longest common subsequence algorithms. In: Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000. pp. 39–48. IEEE (2000)

[5] Blum, C., Blesa, M.J.: A hybrid evolutionary algorithm based on solution merging for the longest arc-preserving common subsequence problem. In: 2017 IEEE Congress on Evolutionary Computation (CEC). pp. 129–136. IEEE (2017)

[6] Blum, C., Blesa, M.J., Lopez-Ibanez, M.: Beam search for the longest common subsequence problem. Computers & Operations Research **36**(12), 3178–3186 (2009)

[7] Carlson, J.M., Chakravarty, A., Gross, R.H.: Beam: a beam search algorithm for the identification of cis-regulatory elements in groups of genes. Journal of Computational Biology **13**(3), 686–701 (2006)

[8] Chen, Y.C., Chao, K.M.: On the generalized constrained longest common subsequence problems. Journal of Combinatorial Optimization **21**(3), 383–392 (2011)

[9] Deorowicz, S., Grabowski, S.: Subcubic algorithms for the sequence excluded lcs problem. In: Man-Machine Interactions 3. pp. 503–510. Springer (2014)

[10] Djukanovic, M., Berger, C., Raidl, G.R., Blum, C.: On solving a generalized constrained longest common subsequence problem. In: International Conference on Optimization and Applications. pp. 55–70. Springer (2020)

[11] Djukanovic, M., Raidl, G.R., Blum, C.: A beam search for the longest common subsequence problem guided by a novel approximate expected length calculation. In: International Conference on Machine Learning, Optimization, and Data Science. pp. 154–167. Springer (2019)

[12] Djukanovic, M., Raidl, G.R., Blum, C.: Finding longest common subsequences: New anytime A$^*$ search results. Applied Soft Computing **95**, 106499 (2020)

[13] Farhana, E., Rahman, M.S.: Constrained sequence analysis algorithms in computational biology. Information Sciences **295**, 247–257 (2015)

[14] Gotthilf, Z., Hermelin, D., Landau, G.M., Lewenstein, M.: Restricted lcs. In: International Symposium on String Processing and Information Retrieval. pp. 250–257. Springer (2010)

[15] Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE transactions on Systems Science and Cybernetics **4**(2), 100–107 (1968)

[16] Hirschberg, D.S.: Algorithms for the longest common subsequence problem. Journal of the ACM (JACM) **24**(4), 664–675 (1977)

[17] Lin, G., Chen, Z.Z., Jiang, T., Wen, J.: The longest common subsequence problem for sequences with nested arc annotations. Journal of Computer and System Sciences **65**(3), 465–480 (2002)

[18] Maier, D.: The complexity of some problems on sequences. Princeton University (1978)

[19] Mousavi, S.R., Bahri, F., Tabataba, F.S.: An enhanced beam search algorithm for the shortest common supersequence problem. Engineering Applications of Artificial Intelligence **25**(3), 457–467 (2012)

[20] Sabuncuoglu, I., Bayiz, M.: Job shop scheduling with beam search. European Journal of Operational Research **118**(2), 390–412 (1999)

[21] Storer, J.A.: Data compression: methods and theory. Computer Science Press, Inc. (1987)

[22] Tsai, Y.T.: The constrained longest common subsequence problem. Information Processing Letters **88**(4), 173–176 (2003)

[23] Wang, Q., Pan, M., Shang, Y., Korkin, D.: A fast heuristic search algorithm for finding the longest common subsequence of multiple strings. In: Proceedings of the AAAI Conference on Artificial Intelligence. vol. 24, pp. 1287–1292 (2010)