Technical Report AC-TR-2020-004
March 2020

An A* Search Algorithm for the Constrained Longest Common Subsequence Problem

Marko Djukanovic, Christoph Berger, Günther R. Raidl, and Christian Blum

Submitted to the journal Bioinformatics
www.ac.tuwien.ac.at/tr
Sequence analysis

An A* Search Algorithm for the Constrained Longest Common Subsequence Problem

Marko Djukanovic 1,*, Christoph Berger, Günther R. Raidl1, and Christian Blum 2

1 Institute of Logic and Computation, TU Wien, 1040 Vienna, Austria
2 Artificial Intelligence Research Institute (IIIA-CSIC), Campus of the UAB, Barcelona, 08193, Spain.

* To whom correspondence should be addressed.

Abstract

Motivation: The constrained longest common subsequence (CLCS) problem was introduced as a specific measure of similarity between molecules. It is a special case of the constrained sequence alignment problem and of the longest common subsequence (LCS) problem, which are both well studied problems in the scientific literature. Finding similarities between sequences plays an important role in the fields of molecular biology, gene recognition, pattern matching, text analysis, and voice recognition, among others. The CLCS problem in particular represents an interesting measure of similarity for molecules that have a putative structure in common.

Results: We propose an exact A* search algorithm for effectively solving the CLCS problem. This A* search is guided by a tight upper bound calculation for the cost-to-go for the LCS problem. Our computational study shows that on various artificial and real benchmark sets this algorithm scales better with growing instance size and requires significantly less computation time to prove optimality than earlier state-of-the-art approaches from the literature.

Availability: The source code of the project is accessible at https://www.ac.tuwien.ac.at/files/resources/software/clcs.zip, and the benchmark instances are available at https://www.ac.tuwien.ac.at/files/resources/instances/clcs/2d-clcs.zip

Contact: djukanovic@ac.tuwien.ac.at

1 Introduction

Strings are objects commonly used for representing DNA and RNA molecules. Finding similarities between molecular structures plays an important role for understanding biological processes that relate to these molecular structures. A frequently applied measure of similarity is given by considering the (length of) subsequences common to all given input strings. Hereby, a subsequence of a string $s$ is any sequence obtained by deleting zero or more characters from $s$. The well-known longest common subsequence (LCS) problem [26] has been studied for more than fifty years in the literature. Given a set of at least two input strings, we seek for a longest string that is a subsequence of all these input strings. This LCS problem has numerous applications, not only in molecular biology [22], but also in data compression [28], pattern recognition, file plagiarism checking, text editing, and voice recognition [24], to name some of the most prominent ones. Furthermore, the LCS problem is a special case of the also prominent sequence alignment problem. Aligning multiple sequences finds application in many tasks such as studying gene regulation or inferring the evolutionary relationships of genes or proteins [6].

A literature review shows that there are several well-studied variants of the LCS problem. Examples include the repetition-free longest common subsequence (RFLCS) problem [1], the longest arc-preserving common subsequence (LAPCS) problem [23], and the longest common palindromic subsequence (LCPS) problem [10]. These variants provide sequence similarity measures depending on the structural properties of the compared molecules. In this paper we study the constrained longest common subsequence (CLCS) problem [29, 2], which is defined as follows. We are given two input strings $s_1$ and $s_2$ and a so-called pattern string $P$. The goal is to find the longest common subsequence of the two input strings that includes $P$ as a subsequence. A possible application scenario of the CLCS problem concerns the identification of homology between two biological sequences which have a specific or putative structure in common [29]. A more concrete example is described in [9]. It deals with the comparison of seven RNAse sequences so that the three active-site
residues, RHR, form part of the solution. This pattern is responsible, in essence, for the main functionality of the RNase molecules such as catalyzing the degradation of RNA sequences.

1.1 Preliminaries

Before we start outlining our approach, let us introduce essential notation. By \([s]\) we denote the length of a string \(s\) over a finite alphabet \(\Sigma\), and by \(n\) we denote the length of the longer one among the two input strings \(s_1\) and \(s_2\), i.e., \(\max(|s_1|, |s_2|)\). The \(j\)-th letter of a string \(s\) is denoted by \(s[j]\), \(j = 1, \ldots, |s|\), and for \(j > |s|\) we define \(s[j] = \epsilon\), where \(\epsilon\) denotes the empty string. Moreover, we denote the contiguous subsequence—that is, the substring—of \(s\) starting at position \(j\) and ending at position \(j'\) by \(s[j:j']\), \(1 \leq j \leq j' \leq |s|\). If \(j > j'\), then \(s[j:j'] = \epsilon\). Finally, let \(|s|_a\) be the number of occurrences of letter \(a \in \Sigma\) in \(s\).

2 Related Work

The CLCS problem with two input strings \(s_1\) and \(s_2\) and a pattern string \(P\) was formally introduced by Tsa2 [29]. A first solution approach based on dynamic programming (DP), which runs in time \(O(|s_1|^2 \cdot |s_2| \cdot |P|)\), was also presented in this work. Due to its large time complexity, this algorithm has no real practical relevance. Since then, several more efficient algorithms were proposed. The most relevant ones are explained in more detail in Section 5. Chin et al. [8] proved that the CLCS problem is a special case of the constrained multiple sequence alignment (CMSA) problem. They developed an alternative DP-based approach that requires \(O(|s_1| \cdot |s_2| \cdot |P|)\) space and time. In fact, this algorithm can be regarded as the first practical algorithm for the CLCS problem. By modifying the recursion of Tsa2 [29], Arslan and Egecioglu [2] also obtained a more efficient algorithm requiring \(O(|s_1| \cdot |s_2|)\) time space. The approach of Chin et al. [8] further inspired the development of an algorithm suggested by Deorowicz [11] with a time complexity of \(O(|P| \cdot (|s_1| + L + R) + |s_2|)\), where \(L\) is the length of the LCS of the two strings and \(R\) is the number of pairs of matching positions between \(s_1\) and \(s_2\). Ideas by Hunt and Szymanski [20] were used to achieve this complexity. Some improvements of the performance of Deorowicz’s algorithm were introduced in a follow-up paper by Deorowicz and Obstoj [12] by utilizing so-called external-entry points (EEP) which were initially proposed in the context of the CMSA problem. Another approach was proposed by Iliopoulos and Rahman [21]. This algorithm has a time complexity of \(O(|P| \cdot R \cdot \log \log n + n)\). It makes use of a specialized bounded heap data structure. Ho et al. [18] proposed a method exploiting the idea that most corresponding CLCS lattice cells in a DP approach remain unchanged in two consecutive layers when \(|\Sigma|\) is small. This algorithm avoids corresponding redundant computations. To the best of our knowledge, the latest algorithm developed for the CLCS problem was proposed by Hung et al. [19]. It is based on the diagonal approach for the LCS problem by Nakatsu et al. [27]. The method requires \(O(|s_1| \cdot (n - L) \cdot |P|)\) time and \(O(|s_1| \cdot |P|)\) space, where \(L\) is the length of a CLCS. From the existing literature, the following conclusions can be drawn.

- The algorithm by Chin et al. [8] is effective for rather short input strings or when \(|\Sigma|\) is small.
- The algorithm by Deorowicz [12] can be seen as the state-of-the-art algorithm for instances with large alphabet sizes.
- The algorithm by Hung et al. [19] was shown to be one order of magnitude faster than the algorithm of Deorowicz. Speed differences are especially noticeable in the presence of a rather high similarity of the input strings (>70%) or a rather low similarity (<20%).

Moreover, we can identify the following weaknesses in the computational studies of the approaches from the literature.

- Most of the benchmark instances used in [19, 12] seem rather easy to solve. In fact, most of the compared algorithms were able to do so in a fraction of a second. This makes it difficult to make well-founded claims about the running times. Moreover, we remark that, apart from the real benchmark instances, all other benchmark instances from the literature are not publicly available.
- The comparison of the two state-of-the-art algorithms from Hung et al. and Deorowicz and Obstoj in [19] was limited to instances with a large fixed alphabet size \(|\Sigma| = 256\). Although it was shown that the algorithm of Hung et al. is an order of magnitude faster than the algorithm from [12] on these instances, the observed differences in running times may not be significant as they are mostly below 0.1 seconds.

2.1 Our Contribution

Our contribution is twofold. First, we present a novel \(A^*\) search approach for the CLCS problem. This algorithm works on a so-called state graph, which is a directed acyclic graph whose nodes represent (partial) solutions. Second, we re-implemented the leading algorithms from the literature and compare our \(A^*\) search with these on a wide and diverse set of benchmark instances which is made publicly available. By means of this comprehensive comparison we are able to make, for the first time, well-founded claims about the practical performance of the considered methods and their individual pros and cons. The obtained results in particular indicate the practical efficiency of our \(A^*\) algorithm. Running times of the \(A^*\) search are in most cases significantly lower than those of the competitors.

The remainder of this article is organized as follows. In Section 3, we first present the state graph that will serve as the environment for our \(A^*\) search. Section 4 presents the \(A^*\) search algorithm, while further details about the re-implemented competitors from the literature are given in Section 5. The experimental comparison of the \(A^*\) search to other state-of-the-art methods is detailed in Section 6. Finally, Section 7 offers conclusions and directions for future work.

3 The State Graph

In the following we introduce the state graph, whose inner nodes are (meaningful) partial solutions, sink nodes are complete solutions, and directed arcs represent (meaningful) extensions of partial solutions. Note that this state graph has similarities to the one that we already presented for the general LCS problem in [13, 14].

Henceforth, let \(S = (s_1, s_2, P, \Sigma)\) be the considered problem instance. Let \(s\) be a string over \(\Sigma\) that is a subsequence of both input strings \(s_1\) and \(s_2\). Moreover, for \(i = 1, 2\), let \(p_i^s\) be the position in \(s_i\) such that \(s_i[p_i^s - 1] = s\). The minimal string among all strings \(s_i[p_i^s, x = 1, \ldots, |s_i|]\), that contains \(s\) as a subsequence. We call \(p^s = (p_1^s, p_2^s)\) the position vector induced by \(s\). Note that, in this way, \(s\) induces a CLCS subproblem \(\mathcal{S}(p^s)\) that consists of strings \(s_1[p_1^s, |s_1|]\) and \(s_2[p_2^s, |s_2|]\). This is because \(s\) can be only be extended by potentially adding letters that appear both in \(s_1[p_1^s, |s_1|]\) and \(s_2[p_2^s, |s_2|]\). In this context, let subsuming \(P[1, k']\) of pattern string \(P\) be the maximal string among all strings \(P[1, x], x = 1, \ldots, |P|\), such that \(P[1, k']\) is a subsequence of \(s\). We then say that \(s\) is a valid (partial) solution iff \(P[k' + 1, |P|]\) is a

---

An A∗ Search Algorithm for the CLCS Problem

3

1.4 Upper Bounds for the CLCS Problem

One of the essential ingredients of an A∗ search is an admissible heuristic function for estimating the cost-to-go, i.e., in our case the length of a CLCS for any subproblem represented by a node of our state graph. In the context of a maximization problem such as the CLCS problem, a heuristic function is said to be admissible if it never underestimates the length of an optimal solution. We therefore make use here of a tight upper bound function that is said to be admissible if it never underestimates the length of an optimal solution. 

However, in order to generate only extensions of node v that correspond to feasible partial solutions, we additionally have to filter out the extensions of node v that correspond to feasible partial solutions, we additionally have to filter out those extensions that lead to subproblems whose strings do not contain the remaining part of P as a subsequence. These cases are encountered by introducing another data structure that is set up during preprocessing: Embed[i, u] stores for each si, i = 1, 2, and for each u =1, 2, . . . , |P| the right-most position x of si such that P[u, x] is a subsequence of si[x, |si|]. Thus, for each letter c ∈ Σ\{e\}, if c ≠ P[|u| + 1] and Succ[i, p[|u| + 1]], letter c cannot be used for extending a partial solution represented by v, and consequently it is removed from Succ[i, p[|u| + 1]]. An extension v′ = (p, l′, u′) is generated for each remaining letter c ∈ Σ\{e\}, where p′ = Succ[i, p[|u| + 1]] for i = 1, 2, l′ = l′ + 1 and u′ = u′ + 1 in case c = P[|u| + 1], respectively u′ = u′ otherwise.

The root node of the state graph is defined by r = (p′ = (1, 1), l′ = 0, u′ = 0). Sink nodes are all non-extensible nodes and represent complete solutions (in contrast to partial solutions). Consequently, a longest path from the root node to a sink node in the state graph represents an optimal solution to the CLCS problem. Finally, notice that the definition of the state graph does not depend on the number of input strings, and can therefore be straightforwardly extended to an arbitrary number of input strings.

An example of the state graph for problem instance (s1 = baabcdba, s2 = cbbcaadbcb, s3 = cbadbca, s4 = cbadbca, s5 = cbadbca, s6 = cbadbca, s7 = cbadbca, s8 = cbadbca) is shown in Fig. 1. The root node, for example, can only be extended by letters b and c, because letters a and d are dominated by the other two letters. Furthermore, note that node (16, 5, 4, 1) (induced by partial solution b e c b c) can only be extended by letter b. Even though letter d is not dominated by letter b, adding letter d can only lead to infeasible solutions, because any possible solution starting with b e c b c will not have P = c e d as a subsequence. Finally, the sequence of arc labels on the longest path is b e c b c b a c b c, which is therefore the (unique) optimal solution to this example problem instance.

3.1 Upper Bounds for the CLCS Problem

One of the essential ingredients of an A∗ search is an admissible heuristic function for estimating the cost-to-go, i.e., in our case the length of a CLCS for any subproblem represented by a node of our state graph. In the context of a maximization problem such as the CLCS problem, a heuristic function is said to be admissible if it never underestimates the length of an optimal solution. We therefore make use here of a tight upper bound function that was originally developed for the LCS problem [13]. Note, in this context, that any valid upper bound for an LCS problem instance is also an upper bound for a corresponding CLCS problem instance obtained by adding a pattern string P to the LCS problem instance.

Given a node v of the state graph, the LCS upper bound function proposed by Blum et al. [3] determines for each letter an upper limit on the number of its occurrences in any solution that contains the partial solution inducing v as prefix string. Summing these values over all letters from Σ, we obtain a valid upper bound on any complete solution that can
4. A* Algorithm for the CLCS Problem

A* is a so-called informed search algorithm that was originally developed by Hart et al. [16] to find shortest paths in weighted graphs. The search maintains a list of open nodes, which is initialized with the root node, and works in a best-first-search manner by expanding in each iteration a most promising open node. In order to rank open nodes, A* search makes use of a priority function \( f(v) = g(v) + h(v) \), for \( v \in V(G) \), where, \( g(v) \) denotes the length of a so far best path from the root node to \( v \), and \( h(v) \) is the heuristic estimate for the cost-to-go, i.e., the length of an optimal further path from \( v \) to a goal node. As the state graph in the case of the CLCS problem was already outlined in Section 3, it remains to be mentioned that for \( h(v) \) we will use the upper bound \( UB(v) \) from the previous section, and \( g(v) = \ell(v) \).

In order for the search process to be efficient, our implementation maintains two data structures: (1) a hash-map \( N \) storing all nodes that were encountered during the search, and (2) the open list \( Q \subseteq N \) containing all not yet expanded/treated nodes. More specifically, \( N \) is implemented as a nested data structure of sorted lists within a hash map. The position vector \( p^v \) of a node \( v \in (P, \Sigma) \) is mapped to a (linked) list storing pairs \((P, \Sigma)\). This structure allows for an efficient membership check, i.e., whether a node \( v \) is in \( N \) or \( Q \).

Note that it might occur that several nodes representing the same subproblem \( S[p^v] \) are stored, as the following example demonstrates: Consider the problem instance with input strings \( s_1 = bcbcmnb \), \( s_2 = abcabcnc \), and pattern string \( P = b \). The A* search might, at some time, encounter node \( v_1 = ((4, 4), 2, 1) \) induced by partial solution \( t_0 \) and—at some other time—it might encounter another node \( v_2 = ((4, 4), 3, 0) \) induced by partial solution \( t_{acbc} \). Even though the path from the root node to node \( v_1 \) is shorter than the path to node \( v_2 \), the former still leads to a better solution in the end (bcbcmnb in comparison to abcabcnc). As the information which of the nodes leads to an optimal solution is not known beforehand, both nodes are stored.

Finally, the open list \( Q \) is realized by a priority queue with priority values \( f(v) = \ell(v) + UB(v) \), for all \( v \in V \). In case of ties, nodes with larger \( \ell(v) \)-values are preferred. In the case of further ties, nodes with larger \( UB(v) \)-values are preferred.

The search starts by inserting the root node of the state graph into \( N \) and \( Q \). Then, at each iteration, a node \( v \) with highest priority is retrieved from \( Q \) and expanded by considering all successor nodes for \( v \in \Sigma \). If such an extensions leads to a new state, the corresponding node, denoted by \( v_{next} \), is added to \( N \) and \( Q \). Otherwise, \( v_{next} \) is compared to the nodes from set \( N_{rel} \subseteq N \) containing those nodes that represent the same subproblem \( S[p^v] \) only. Dominated nodes are identified in this way and dropped from the search process, i.e., the dominated nodes are removed from \( N \) and \( Q \). If node \( v_{next} \) is dominated by one of the nodes from \( N_{rel} \), it can simply be discarded. Otherwise, it is added to \( N \) and \( Q \). In this context, given \( v_1, v_2 \in N_{rel} \) we say that \( v_1 \) dominates \( v_2 \) if \( \ell(v_1) \geq \ell(v_2) \) and \( \ell(v_1) \geq \ell(v_2) \).

We would like to emphasize that detecting the domination in \( N_{rel} \) was, on average, slightly faster when the elements of the lists were sorted in decreasing order of their \( \ell(v) \)-values. Therefore, we used this ordering in our implementation. As the upper bound function \( UB(v) \) is admissible—that is, it never underestimates the length of an optimal solution—A* yields an optimal solution whenever the node selected for expansion is a complete node [16]. Moreover, note that \( UB(v) \) is also monotonically increasing, which means that the upper bound of any child node never overestimates the upper bound of its parent node. This implies that no re-expansion of already expanded nodes become necessary [16]. In general, A* search is known to be optimal in terms of the number of node expansions required to prove optimality w.r.t. the upper bound and the tie-breaking criterion used. A pseudocode of our A* search implementation for the CLCS problem is provided in Algorithm 1.

5 Algorithms Used for Comparison

Algorithm by Chin et al. [8]. This method is based on dynamic programming. It uses a three-dimensional matrix \( M \) to store the lengths of optimal solutions of subproblems \( S_{i,j,k} = (s_1[1..i], s_2[1..j], P[1..k], \Sigma) \) for \( i = 1, \ldots, |s_1| \), \( j = 1, \ldots, |s_2| \), \( k = 1, \ldots, |P| \). All these
values are obtained recursively on the basis of solutions to smaller subinstances for which optimal values are already known. In essence, the recursive procedure distinguishes the following cases and handles them appropriately: $s_1[i] = s_2[j] = P[k]$, $s_1[i] = s_2[j] \neq P[k]$, or $s_1[i] \neq s_2[j]$. In this way, optimal values of successor entries (representing larger subproblems) are determined in constant time.

Due to its simplicity, the algorithm is fast for problem instances of small and medium size but its performance degrades for longer sequences. In general, its time and space complexity is $O(|s_1| \cdot |s_2| \cdot |P|)$.

Algorithm by Arslan and Eğecioğlu [2]. This approach replaces the matrix used in the original dynamic programming algorithm of Tsai [29] by multiple three-dimensional matrices in order to realize some calculations of the approach of Tsai more efficiently. In particular, the recurrence used by Tsai was simplified. In the end, this results in an algorithm with the same time complexity as the algorithm of Chint et al., however with a memory requirement that is by a factor of three higher.

Algorithm by Iliopoulos and Rahman [21]. This method is based on a modification of the dynamic programming formulation from [2]. To perform the matrix calculations of each iteration efficiently, the authors make use of a so-called bounded heap data structure [5] that was realized by means of Van Emde Boas (vEB) trees [4]. This data structure allows to calculate intermediate results more efficiently in $O(\log \log n)$ time, leading to a total time complexity of $O(|P| \cdot \log \log n + n)$, where $R$ is the number of ordered pairs of positions at which input strings $s_1$ and $s_2$ match.

Algorithm by Hong et al. [19]. This method is a more recent development that is particularly suited for input strings that are highly similar. It was developed on the basis of the so-called diagonal concept for the LCS [17] initially proposed for the pairwise sequence alignment problem, for omitting those cells in the lists that do not contribute to optimal solutions.

### 6 Experimental Results
All algorithms were implemented in C++ with g++ 7.4 and the experiments were conducted in single-threaded mode on a machine with an Intel Xeon E5-2640 processor with 2.40 GHz and a memory limit of 32 GB. The maximum computation time allowed for each run was limited to one hour.

We aimed to re-implement all algorithms from the literature in the way in which they are described in the original articles as the respective code could not be obtained. In a few cases, due to a lack of sufficient details, we had to make our own specific implementation decisions. This was in particular the case for the algorithm of Biopolous and Rahman [21]. The bounded heap data structure has to be initialized for different indices, and it remains unclear how this can be done efficiently. The authors were contacted with this issue but we did not receive a response. Our implementation creates a new bounded heap for a new index by copying the content from the bounded heap of the previous index. This is the most time-demanding part of the algorithm, which is in particular noticed in the context of instances with large values of $n$. Unfortunately, the original article does not contain any computational study that could serve as a comparison but just focuses on asymptotic runtimes from a theoretical point-of-view.

We emphasize that in general, we did our best to achieve efficient re-implementations of the approaches from literature for the experimental comparison.

#### 6.1 Benchmark Instances
With the aim of creating a diverse set of problem instances, for each combination of $n \in \{100, 500, 1000\}$ (length of the input strings), $|\Sigma| \in \{4, 12, 20\}$ (alphabet size), $p' = \frac{1}{|\Sigma|^2} \in \{\frac{1}{4}, \frac{1}{16}, \frac{1}{2}, \frac{1}{4}\}$ (length of the pattern string), ten problem instances were randomly generated. This results in a total of 450 instances. The following procedure was used for generating each instance. First, a pattern string $P$ was created uniformly at random, that is, each character from $\Sigma$ has an equal chance to be chosen for each position of $P$. Second, two input strings of equal length $n$ were generated as follows. First, a pattern string $P$ was created uniformly at random from $\Sigma$. Then, characters $P[1], \ldots, P[|P|]$ are placed (in this order) from left to right at these positions. Finally, the remaining characters of each input string were set to letters chosen uniformly at random from the alphabet $\Sigma$. This procedure ensures that at least one feasible CLCS solution exists for each benchmark instance. Unfortunately, none of the artificial benchmarks from [12] and [19] were provided to us, although the respective authors were contacted with this concern.

In addition to these artificially generated instances, we use a benchmark suite from [12] based on strings representing real biological sequences².

This benchmark set is henceforth called Real. It has its origins in experimental studies on the constrained multiple sequence alignment (CMSA) problem considered in [25, 9]. Each possible pair of sequences from this data set, together with a pattern string, was used in [12] to define a problem instance for the CLCS problem. Properties of the input strings, together with their origins, are provided in Table 1. In particular, Chen et al. [9] provided four sets of strings containing RNAse sequences with lengths from 111 to 327. In contrast, set ds—containing aspartic acid protease family sequences—was provided by Li and Huang [25], also in the context of the CMSA problem. Overall, benchmark set Real consists of 121 problem instances.

Table 1. Benchmark suite Real from [12].

<table>
<thead>
<tr>
<th>data set</th>
<th>number of sequences</th>
<th>sequence length (min, med, max)</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>d0</td>
<td>7</td>
<td>(111, 124, 134)</td>
<td>20</td>
<td>[9]</td>
</tr>
<tr>
<td>d1</td>
<td>6</td>
<td>(124, 149, 185)</td>
<td>20</td>
<td>[9]</td>
</tr>
<tr>
<td>d2</td>
<td>6</td>
<td>(131, 142, 160)</td>
<td>20</td>
<td>[9]</td>
</tr>
<tr>
<td>d3</td>
<td>5</td>
<td>(189, 277, 327)</td>
<td>20</td>
<td>[9]</td>
</tr>
<tr>
<td>d4</td>
<td>6</td>
<td>(98, 114, 123)</td>
<td>20</td>
<td>[25]</td>
</tr>
</tbody>
</table>

Table 2. Instances with $p' = \frac{\Sigma}{\delta_0} = \frac{1}{n}$: Average runtimes in seconds.

<table>
<thead>
<tr>
<th>$\Sigma$</th>
<th>n</th>
<th>Chin</th>
<th>Deo</th>
<th>AE</th>
<th>IR</th>
<th>Hung</th>
<th>A*</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>100</td>
<td>0.9</td>
<td>0</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>4</td>
<td>500</td>
<td>319.3</td>
<td>0.1</td>
<td>0.1</td>
<td>0.2</td>
<td>6.5</td>
<td>0.1</td>
</tr>
<tr>
<td>4</td>
<td>1000</td>
<td>646.3</td>
<td>0.2</td>
<td>1</td>
<td>1.3</td>
<td>86.4</td>
<td>0.5</td>
</tr>
<tr>
<td>12</td>
<td>100</td>
<td>40.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>12</td>
<td>500</td>
<td>216.0</td>
<td>0.1</td>
<td>0.1</td>
<td>0.2</td>
<td>2.9</td>
<td>0.2</td>
</tr>
<tr>
<td>12</td>
<td>1000</td>
<td>435.5</td>
<td>0.3</td>
<td>0.5</td>
<td>1.4</td>
<td>39.4</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>100</td>
<td>175.5</td>
<td>&lt;0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>20</td>
<td>500</td>
<td>355.4</td>
<td>0.3</td>
<td>0.5</td>
<td>1.4</td>
<td>26.6</td>
<td>1.1</td>
</tr>
</tbody>
</table>

Table 3. Instances with $p' = \frac{\Sigma}{\delta_0} = \frac{1}{n}$: Average runtimes in seconds.

<table>
<thead>
<tr>
<th>$\Sigma$</th>
<th>n</th>
<th>Chin</th>
<th>Deo</th>
<th>AE</th>
<th>IR</th>
<th>Hung</th>
<th>A*</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>100</td>
<td>61.9</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>4</td>
<td>500</td>
<td>323.0</td>
<td>0.1</td>
<td>0.5</td>
<td>0.4</td>
<td>15.7</td>
<td>0.2</td>
</tr>
<tr>
<td>4</td>
<td>1000</td>
<td>645.9</td>
<td>0.9</td>
<td>1.8</td>
<td>3.4</td>
<td>215.5</td>
<td>1.2</td>
</tr>
<tr>
<td>12</td>
<td>100</td>
<td>41.0</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>12</td>
<td>500</td>
<td>215.3</td>
<td>0.1</td>
<td>0.2</td>
<td>0.4</td>
<td>5.3</td>
<td>0.3</td>
</tr>
<tr>
<td>12</td>
<td>1000</td>
<td>457.0</td>
<td>0.9</td>
<td>1.1</td>
<td>3.4</td>
<td>69.2</td>
<td>2.2</td>
</tr>
<tr>
<td>20</td>
<td>100</td>
<td>322</td>
<td>&lt;0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>20</td>
<td>500</td>
<td>170.9</td>
<td>0.1</td>
<td>0.2</td>
<td>0.3</td>
<td>5.3</td>
<td>0.2</td>
</tr>
<tr>
<td>20</td>
<td>1000</td>
<td>344.8</td>
<td>1</td>
<td>1.1</td>
<td>3.5</td>
<td>40.6</td>
<td>1.7</td>
</tr>
</tbody>
</table>

6.2 Results

We compare our A* search from Section 4 with our re-implementations of the following state-of-the-art algorithms from the literature.

- **Chin:** Algorithm by Chin et al. [8];
- **Deo:** Algorithm by Deorowicz [11];
- **AE:** Algorithm by Arslan and Ejegho [2];
- **IR:** Algorithm by Biapolous and Rahman [21];
- **Hung:** Algorithm by Hung et al. [19].

In general, all algorithms could find optimal solutions and prove their optimality for all instances. However, the required runtime differs sometimes substantially. Tables 2–7 show these runtimes for each re-implemented algorithm as well as our A* search in seconds averaged over each group of instances. Results for the artificial instance sets are subdivided into five different subclasses w.r.t. the value of $p'$, which determines the length of pattern string $P$. Concerning benchmark suite Real, the average running times refer to all those instances that belong to the respective data set in combination with a pattern $P$, cf. Table 7. For each instance group (line), the lowest runtimes among the competing algorithms are shown in bold font. The first two columns present the properties of the instance group, while the third column lists the average length of the optimal solutions for the respective problem instances. The following observations can be drawn from these results.

- The small instances (where $n = 100$) are easy to solve and all competitors require only a fraction of a second for doing so.

<table>
<thead>
<tr>
<th>$\Sigma$</th>
<th>n</th>
<th>Chin</th>
<th>Deo</th>
<th>AE</th>
<th>IR</th>
<th>Hung</th>
<th>A*</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>100</td>
<td>62.6</td>
<td>&lt;0.1</td>
<td>&lt;0.1</td>
<td>&lt;0.1</td>
<td>&lt;0.1</td>
<td>&lt;0.1</td>
</tr>
<tr>
<td>4</td>
<td>500</td>
<td>320.9</td>
<td>0.3</td>
<td>0.6</td>
<td>0.9</td>
<td>26.8</td>
<td>0.4</td>
</tr>
<tr>
<td>4</td>
<td>1000</td>
<td>646.4</td>
<td>1.8</td>
<td>3.5</td>
<td>9.2</td>
<td>331.2</td>
<td>3.3</td>
</tr>
<tr>
<td>12</td>
<td>100</td>
<td>40.5</td>
<td>&lt;0.1</td>
<td>&lt;0.1</td>
<td>&lt;0.1</td>
<td>&lt;0.1</td>
<td>&lt;0.1</td>
</tr>
<tr>
<td>12</td>
<td>500</td>
<td>207.1</td>
<td>0.2</td>
<td>0.3</td>
<td>0.9</td>
<td>7.3</td>
<td>0.3</td>
</tr>
<tr>
<td>12</td>
<td>1000</td>
<td>419.0</td>
<td>2.1</td>
<td>2.2</td>
<td>8.3</td>
<td>91.1</td>
<td>2.7</td>
</tr>
<tr>
<td>20</td>
<td>100</td>
<td>31.1</td>
<td>&lt;0.1</td>
<td>&lt;0.1</td>
<td>&lt;0.1</td>
<td>&lt;0.1</td>
<td>&lt;0.1</td>
</tr>
<tr>
<td>20</td>
<td>500</td>
<td>157.4</td>
<td>0.2</td>
<td>0.3</td>
<td>0.9</td>
<td>5.3</td>
<td>0.2</td>
</tr>
<tr>
<td>20</td>
<td>1000</td>
<td>317.9</td>
<td>1.3</td>
<td>2.1</td>
<td>8.4</td>
<td>68.1</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 7. Benchmark set Real: Average runtimes in seconds.

<table>
<thead>
<tr>
<th>$P$</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_0$</td>
<td>HKSH</td>
<td>0.012</td>
<td>0.013</td>
<td>0.014</td>
<td>0.017</td>
<td>0.017</td>
<td>0.018</td>
</tr>
<tr>
<td>$d_1$</td>
<td>HKSH</td>
<td>0.012</td>
<td>0.017</td>
<td>0.013</td>
<td>0.012</td>
<td>0.019</td>
<td>0.019</td>
</tr>
<tr>
<td>$d_2$</td>
<td>HKSH</td>
<td>0.011</td>
<td>0.013</td>
<td>0.017</td>
<td>0.022</td>
<td>0.024</td>
<td>0.024</td>
</tr>
<tr>
<td>$d_3$</td>
<td>HKSH</td>
<td>0.015</td>
<td>0.017</td>
<td>0.017</td>
<td>0.022</td>
<td>0.024</td>
<td>0.024</td>
</tr>
<tr>
<td>$d_4$</td>
<td>HKSH</td>
<td>0.015</td>
<td>0.017</td>
<td>0.018</td>
<td>0.022</td>
<td>0.024</td>
<td>0.024</td>
</tr>
</tbody>
</table>

The first algorithm that starts losing efficiency with growing input string length is $P$. Already starting with $n = 500$, the computation times start to grow substantially in comparison to the other approaches, which is most likely due to the complexity of the utilized data structures. We remark that our specific implementation decision concerning the initialization of the bounded heap may have a significant impact, as mentioned already in Section 5.

- **Algorithm Chin clearly outperforms Deo when $|\Sigma|$ is small.** With growing $|\Sigma|$, as already noticed in earlier studies [11], Deo becomes more efficient. In fact, the two approaches perform similarly for $|\Sigma| = 20$. The advantages of Deo over Chin are noticed in particular for higher $p'$; see Table 5.

- **Algorithm Hung generally performs better than Deo and Chin.** This confirms the conclusions from the computational study in Hung et al. [19].
An $A^*$ Search Algorithm for the CLCS Problem

- With increasing $p'$ and an increasing length of $P$, all approaches degrade in their performance, except for $A^*$ and Hung, which still remain highly efficient.
- A general conclusion for the artificial benchmark set is that $A^*$ search is in most cases about one order of magnitude faster than Hung, which is overall the second-best approach.
- Concerning the results for benchmark set bault (see Table 7), we can conclude that all algorithms only require short times as the input strings are rather short. Nevertheless we can also see here that the $A^*$ search is almost consistently fastest.
- Figure 2 shows the influence of the instance length on the algorithms’ runtimes for $|\Sigma| = 4$ and $|\Sigma| = 20$. Note that $h$ is not included here since it was obviously the slowest among the competitors. It can be noticed that the performance of $A^*$ is the only one that does not degrade much with increasing $n$.
- Figure 3 shows the influence of the length of $\Sigma$ on the algorithms’ runtimes for $n = 500$ and $n = 1000$ (in log-scale). It can be noticed again that $A^*$ does not suffer much from an increase of the length of $P$. This also holds for Hung but not the other competitors, whose performance degrades with increasing $|P|$.

Finally, we also compare the amount of work done by the algorithms in order to reach the optimal solutions. In the case of $A^*$, this amount of work is measured by the number of generated nodes of the state graph. In the case of DoCo, this refers to the number of different keys $(i, j, k)$ generated during the algorithm execution. Finally, in the case of Hung, this is measured by the amount of newly generated nodes in each $D_{i,j}$ (which corresponds to the amount of non-dominated extensions of the nodes from $D_{i−1,j−1}$). Let us call this measure the amount of created nodes for all three algorithms. This measure is shown in log-scale in Fig. 4 for the instances with $n = 500$. The $x$-axis of these graphics varies over different ratios $p' = |P|$. The curve denoted by Max (see legends) is the theoretical upper bound on the number of created nodes, which is $|x_1| \times |x_2| \times |P|$ for an instance $(x_1, x_2, P, \Sigma)$. The graphics clearly show that $A^*$ creates the fewest nodes in comparison to the other approaches. The difference becomes larger with an increasing length of $P$, which correlates with an increase in the similarity between the input strings. For those instances with strongly related input strings, the upper bound UB used in the $A^*$ search is usually tighter, which results in fewer node expansions. The amount of created nodes in $A^*$ decreases with an increasing length of $P$ after some point, because the search space becomes more restricted; see Fig. 4 and $|\Sigma| = 4$ from $p' \geq 4$ onward and $|\Sigma| = 20$ from $p' \geq 1/5$ onward.

7 Conclusions and Future Work

In future work, we plan to extend this $A^*$ search towards the general CLCS problem with an arbitrary number of input strings, which is an NP-hard problem. Moreover, we consider the $A^*$ search also a promising framework for solving related LCS problem variants such as the restricted LCS (RLCS) problem [15, 7]. For those instances where $A^*$ search might fail to prove optimality (e.g., due to exceeding a memory limit), the $A^*$ framework might be turned into an anytime algorithm [31] in order to obtain high-quality heuristic solutions already early during the search process.

**Fig. 2.** Average computation times of the algorithms for $p' = \frac{1}{n}$

**Fig. 3.** Average computation times of the algorithms for $|\Sigma| = 20$.

**Fig. 4.** Average amount of created nodes by the algorithms for $n = 500$.

Acknowledgments.

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


