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

Marko Djukanovic[a], Christoph Berger, Günther R. Raidl[a], Christian Blum[b]

[a]*Institute of Logic and Computation, TU Wien, 1040 Vienna, Austria*
[b]*Artificial Intelligence Research Institute (IIIA-CSIC), Campus of the UAB, Barcelona, 08193, Spain.*

Abstract

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. This paper proposes 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.

Keywords: longest common subsequences, constrained sequences, A* search

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 [1] 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 [2], but also in data compression [3], pattern recognition, file plagiarism checking, text editing, and voice recognition [4], 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 [5].

A literature review shows that there are several well-studied variants of the LCS problem. Examples include the *the repetition-free longest common subsequence* (RFLCS) problem [6], *the longest arc-preserving common subsequence* (LAPCS) problem [7], and the *the longest common palindromic subsequence* (LCPS) problem [8]. 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 [9, 10], 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 of 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 [9]. A more concrete example is described in [11]. It deals with the comparison of seven RNase sequences so that the three active-site residues, `HKH`, form part of the solution[1]. 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 Σ, 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 $s[j]$, $j = 1, \dots, |s|$, and for $j > |s|$ we define $s[j] = \varepsilon$, where ε 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 $s[j, j']$, $1 \le j \le j' \le |s|$. If $j > j'$, then $s[j, j'] = \varepsilon$. 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 Tsai [9]. A first solution approach based on dynamic programming (DP), which runs

[1]National Center of Biotechnology Information database, at http://www.ncbi.nlm.nih.gov.

in time $O(|s_1|^2 \cdot |s_2|^2 \cdot |P|)$, 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. [12] proved that the CLCS problem is a special case of the *constrained multiple sequence alignment* (CMSA) problem. Moreover, 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 Tsai [9], Arslan and Eğecioğlu [10] also obtained a more efficient algorithm requiring $O(|s_1| \cdot |s_2| \cdot |P|)$ time. The approach of Chin et al. [12] further inspired the development of an algorithm suggested by Deorowicz [13] with a time complexity of $O(|P| \cdot (|s_1| \cdot L + R) + |s_2|)$, 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 [14] 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 [15] 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 [16]. 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. [17] 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. [18]. It is based on the diagonal approach for the LCS problem by Nakatsu et al. [19]. The method requires $O(|P| \cdot L \cdot (n - L))$ 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. [12] is effective for rather short input strings or when $|\Sigma|$ is small.
- The algorithm by Deorowicz [15] can be seen as the state-of the-art algorithm for instances with large alphabet sizes.
- The algorithm by Hung et al. [18] 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 [18, 15] 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 [18] 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 [15] 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 [20, 21].

Henceforth, let $S = (s_1, s_2, P, \Sigma)$ be the considered problem instance. Let s be a string over Σ 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 $s_i[1, p_i^s - 1]$ is the minimal string among all strings $s_i[1, x]$, $x = 1, \ldots, |s_i|$, that contains s as a subsequence. We call $\mathbf{p}^s = (p_1^s, p_2^s)$ the *position vector* induced by s. Note that, in this way, s induces a CLCS subproblem $S[\mathbf{p}^s]$ that consists of strings $s_1[p_1^s, |s_1|]$ and $s_2[p_2^s, |s_2|]$. This is because s can 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 substring $P[1, k']$ 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 subsequence of the strings in subproblem $S[\mathbf{p}^s]$, that is, a subsequence of $s_1[p_1^s, |s_1|]$ and $s_2[p_2^s, |s_2|]$.

The state graph $G = (V, A)$ for our A* algorithm is a directed acyclic graph, which—at any moment—is only known

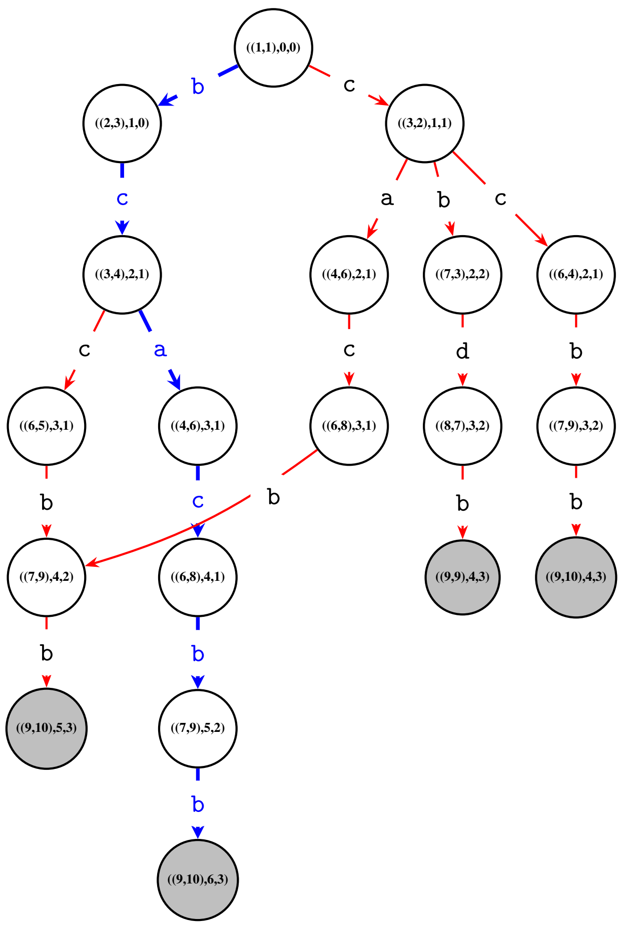


Figure 1: Example showing the full state graph for the problem instance $(\{s_1 = \texttt{bcaacbdba}, s_2 = \texttt{cbccadcbbd}\}, P = \texttt{cbb}, \Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\})$. There are four sink nodes representing non-extensible solutions (marked by light-gray color). The optimal solution is $s = \texttt{bcacbb}$ of length 6 that corresponds to the node $v = (\mathbf{p}^v = (9, 10), l^v = 6, u^v = 3)$. The longest path that corresponds to the optimal solution is displayed by means of thick arrows.

partially by our A* approach. Each node $v \in V(G)$ stores a triple $(\mathbf{p}^v, l^v, u^v)$, where $\mathbf{p}^v$ is a position vector that induces subproblem $S[\mathbf{p}^v] = (s_1[p^v_1, |s_1|], s_2[p^v_2, |s_2|], P[u^v + 1, |P|], \Sigma)$, where l^v is the length of the currently best known valid partial solution that induces $\mathbf{p}^v$, and u^v is the length of the longest prefix string of pattern string P that is contained as a subsequence in the best known partial solution that induces node v. Moreover, there is an arc $a = (v, v') \in A(G)$ with label $l(a) \in \Sigma$ between two nodes $v = (\mathbf{p}^v, l^v, u^v)$ and $v' = (\mathbf{p}^{v'}, l^{v'}, u^{v'})$ iff

- $l^{v'} = l^v + 1$ and
- Subproblem $S[\mathbf{p}^{v'}]$ is induced by the partial solution that is obtained by appending letter $l(a)$ to the partial solution that induces v.

As remarked already above, we are only interested in meaningful partial solutions, and our A* search builds the state graph on the fly. In particular, for extending a node v, the outgoing arcs—that is, the letters that may be used to extend partial solutions that induce node v—are determined as follows. First of all, these letters must appear in both strings from $S[\mathbf{p}^v]$; we call this subset of the alphabet *potential letters*. In order to find the position of the first (left-most) appearance of each potential letter in the strings from $S[\mathbf{p}^v]$ we make use of a successor data structure determined during preprocessing that allows to retrieve each position in constant time. Let this position of the first appearance of a potential letter c in string $s_i[p^v_i, |s_i|$ be $Succ[i, p^v_i, c]$, $i = 1, 2$. Moreover, a potential letter should not be taken for extending v in case it is dominated by another potential letter: We say that a letter c is *dominated* by a letter $c' \neq c$ iff $Succ[i, p^v_i, c] \geq Succ[i, p^v_i, c']$, $i = 1, 2$. Note that a dominated letter cannot lead to a better solution than when taking the letter by which it is dominated instead. Henceforth, we denote the set of non-dominated potential letters for extending a node v by $\Sigma^{\text{nd}}_v \subseteq \Sigma$. However, in order to generate only 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 s_i, $i = 1, 2$, and for each $u = 1, \ldots, |P|$ the right-most position x of s_i such that $P[u, |P|]$ is a subsequence of $s_i[x, |s_i|]$. Thus, for each letter $c \in \Sigma^{\text{nd}}_v$, if $c \neq P[u^v + 1]$ and $Succ[i, p^v_i, c] > Embed[i, u^v + 1]$, letter c cannot be used for extending a partial solution represented by v, and consequently it is removed from Σ^{nd}_v. An extension $v' = (\mathbf{p}^{v'}, l^{v'}, u^{v'})$ is generated for each remaining letter $c \in \Sigma^{\text{nd}}_v$, where $p^{v'}_i = Succ[i, p^v_i, c] + 1$ for $i = 1, 2$, $l^{v'} = l^v + 1$ and $u^{v'} = u^v + 1$ in case $c = P[u^v + 1]$, respectively $u^{v'} = u^v$ otherwise.

The *root* node of the state graph is defined by $r = (\mathbf{p}^r = (1, 1), l^r = 0, u^r = 0)$. 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 full state graph for problem instance $(\{s_1 = \texttt{bcaacbdba}, s_2 = \texttt{cbccadcbbd}\}, P = \texttt{cbb}, \Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\})$ 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 $((6, 5), 3, 1)$ (induced by partial solution bcc) 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 bccd will not have $P = \texttt{cbb}$ as a subsequence. Finally, the sequence of arc labels on the longest path is bcacbb, 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 typically tight upper bound function that was originally developed for the LCS problem [20]. 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. [22] 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 be constructed starting from v:

$$\mathrm{UB}_1(v) = \sum_{a\in\Sigma} \min\left(|s_1[p_1^v, |s_1|]|_a, |s_2[p_2^v, |s_2|]|_a\right) \tag{1}$$

This bound is efficiently calculated in $O(|\Sigma|)$ time by making use of some data structures as detailed in [21].

An alternative DP–based upper bound function was introduced by Wang et al. [23]. It makes use of the DP recursion for the LCS problem with two input strings. A *scoring matrix* M is generated where entry $M[x, y]$, $x = 1, \ldots, |s_1| + 1$, $y = 1, \ldots, |s_2| + 1$ stores the length of the LCS of $s_1[x, |s_1|]$ and $s_2[y, |s_2|]$. Thus, an upper bound for a given state graph node v is given by

$$\mathrm{UB}_2(v) = M[p_1^v, p_2^v]. \tag{2}$$

Neglecting the preprocessing step for generating M, this bound can be efficiently retrieved in constant time. As neither of the two bounds dominate the other, we use here the combination of both given by $\mathrm{UB}(v) := \min\{\mathrm{UB}_1(v), \mathrm{UB}_2(v)\}$.

4. A* Algorithm for the CLCS Problem

A* is a so-called informed search algorithm that was originally developed by Hart et al. [24] 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 $\mathrm{UB}(v)$ from the previous section, and $g(v) := l^v$.

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 $\mathbf{p}^v$ of a node $v = (\mathbf{p}^v, l^v, u^v)$ is mapped to a (linked) list storing pairs (l^v, u^v). This structure allows for an efficient membership check, i.e., whether or not a node that represents subproblem a $S[\mathbf{p}^v]$ was already encountered during the search, and a quick retrieval of the respective nodes.

Note that it might occur that several nodes representing the same subproblem $S[\mathbf{p}^v]$ are stored, as the following example demonstrates: Consider the problem instance with input strings s_1 = `bacxmnob`, s_2 = `abcxmbno`, and pattern string P = `b`. The A* search might, at some time, encounter node $v_1 = ((4, 4), 2, 1)$ induced by partial solution `bx`, and—at some other time—it might encounter another node $v_2 = ((4, 4), 3, 0)$ induced by partial solution `acx`. 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 (`bxmno` in comparison to `acxb`). 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 $f(v) = l^v + \mathrm{UB}(v)$, for all $v \in V$. In case of ties, nodes with larger l^v-values are preferred. In the case of further ties, nodes with larger u^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 $a \in \Sigma_v^{\mathrm{nd}}$). If such an extensions leads to a new state, the corresponding node, denoted by v_{ext}, is added to N and Q. Otherwise, v_{ext} is compared to the nodes from set $N_{\mathrm{rel}} \subseteq N$ containing those nodes that represent the same subproblem $S[\mathbf{p}^v]$. 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_{ext} 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 $v_1, v_2 \in N_{\mathrm{rel}}$ we say that v_1 dominates v_2 iff $l^{v_1} \geq l^{v_2} \ \wedge\ u^{v_1} \geq u^{v_2}$. 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 u^v-values. Therefore, we used this ordering in our implementation.

As the upper bound function UB() 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 [24]. Moreover, note that UB() also is *monotonic*, 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 [24]. 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.

```

N = Q = []
r = ((1,1),0,0)
N.insert(r)
Q.insert(r)
while(Q != []):
    v = Q.pop()
    Determine $\Sigma_v^{\mathrm{nd}}$
    if $\Sigma_v^{\mathrm{nd}}$ = []: # complete solution found
        return the solution corresponds to v
    else:
```

```
12          for $c$ in $\Sigma_v^{\text{nd}}$:
13              Generate child $v_{\text{ext}}$ w.r.t. char. $c$
14              $N_{\text{rel}} = N[\mathrm{p}^{v_{\text{ext}}}]$
15              for $v_{\text{rel}}$ in $N_{\text{rel}}$:
16                  if $l^{v_{\text{rel}}} \geq l^{v_{\text{ext}}}$ and $u^{v_{\text{rel}}} \geq u^{v_{\text{ext}}}$:
17                      $insert = $ false
18                      break # domination fulfilled
19                  if $l^{v_{\text{ext}}}$ >= $l^{v_{\text{rel}}}$ and $u^{v_{\text{ext}}} \geq u^{v_{rel}}$:
20                    Remove node $v_{\text{rel}}$ from $N$, $Q$
21             if $insert$: #new state is non-dominated
22               $N.insert(v_{\text{ext}})$
23               $Q.insert(v_{\text{ext}})$ # priritized acc. to UB
24  return $\varepsilon$ if no feasible solution exists
25
```

Algorithm 1: A* search for the CLCS problem.

4.1. *Time and Space Complexity of the A* Search*

In general, an upper bound for the worst-case performance of A* search is $O(b^d)$, where b is the branching factor—which, in our case, is the number of letters—and d is the length of an optimal solution. In other words, the runtime of A* search is, in general, exponential. Providing a tighter bound is often hardly possible, as the practical runtime strongly depends on the used guidance heuristic [25]. In practice, however, it frequently happens that A* search, when using a meaningful heuristic, is quite fast, even in those cases in which nothing better than the exponential worst-case run time can be proven. Therefore, respective publications typically focus more on empirically observed run times or indicate the number of expanded/visited nodes, for example, [23].

Nevertheless, it is possible to derive polynomial worst-case time and space complexity bounds for our A* search from Algorithm 1 as follows. The number of visited nodes is bounded by $O(n^2 \cdot |P|)$. Since the used upper bound function is monotonic, we can be sure that no re-expansion of already expanded nodes is necessary, which further implies that the outer while-loop of Algorithm 1 is executed at most $O(n^2 \cdot |P|)$ times. The pop() function in Line 7 of Algorithm 1 needs a constant time to retrieve the top node of Q. Afterwards, reorganizing the nodes in the priority queue Q is done in $O(\log |Q|) = O(\log(n \cdot |P|) = O(n)$ time. Determining the set of non-dominated nodes of a node v is achieved in $O(|\Sigma|^2 \cdot n)$ time by pairwise comparisons. For generating all child nodes of a node v and then checking the domination among the nodes which refer to the same subproblem (Lines 15-20), $O(|\Sigma| \cdot n \cdot \log n)$ time is required in total. Note that the factor $\log(n)$ reflects the time required to check the domination of a single node, which can be done via binary search. The code in Lines 21–23 is executes in $O(\log(n \cdot |P|)) = O(n)$ time. Overall, to execute a single iteration of the main while-loop, we need

$$O(\log n + |\Sigma| \cdot n \cdot \log n + |\Sigma|^2 \cdot n + \log(n \cdot |P|)) = \quad (3)$$

$$O(|\Sigma| \cdot n \cdot \log n + |\Sigma|^2 \cdot n) = O(n \cdot |\Sigma| \cdot (\log n + |\Sigma|)) \quad (4)$$

time. For executing the whole algorithm, the time is in

$$O(n \cdot |\Sigma| \cdot (\log n + |\Sigma|) \cdot O(n^2 \cdot |P|) = \quad (5)$$

$$O(n^3 \cdot |P| \cdot |\Sigma| \cdot (\log n + |\Sigma|)). \quad (6)$$

Since $|\Sigma|$, in practice, represents a small constant number, the time to execute our A* search is in

$$O(n^3 \cdot |P| \cdot \log n). \quad (7)$$

Concerning the space complexity of the proposed A* algorithm, the worst case corresponds to storing all nodes of the state graph, and is thus in $O(n^2 \cdot |P|)$.

5. Algorithms Used for Comparison

Algorithm by Chin et al. [12].. 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 [10].. This approach replaces the matrix used in the original dynamic programming algorithm of Tsai [9] 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 Chin et al., however with a memory requirement that is by a factor of three higher.

Algorithm by Iliopoulos and Rahman [16].. This method is based on a modification of the dynamic programming formulation from [10]. To perform the matrix calculations of each iteration efficiently, the authors make use of a so-called *bounded heap* data structure [26] that was realized by means of Van Emde Boas (vEB) trees [27]. This data structure allows to calculate intermediate results more efficiently in $O(\log \log n)$ time, leading to a total time complexity of $O(|P| \cdot R \cdot \log \log n + n)$, where R is the number of ordered pairs of positions at which input strings s_1 and s_2 match.

Algorithm by Hung et al. [18].. 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 problem by Nakatsu et al. [19]. In general it can be said that the efficiency of the algorithm grows with the length of an optimal CLCS solution. The algorithm uses a table D of dimension $|P| \times L$, where L is an upper bound for the CLCS length. Each cell $D_{i,l}$ stores a triple associated with a partial solution. At each iteration of the algorithm some of the cells are filled with information such that for any triple $(i', j, k) \in D_{i,l}$, where $i' = 1,\ldots,i$, the relation

$|\text{CLCS}(s_1[1,i'], s_2[1,j], P[1,|P|-k])| \geq l$ holds. The elements belonging to $D_{i,l}$ are determined by extending all the partial solutions from $D_{i-1,l-1}$, to which all the partial solutions of $D_{i-1,l}$ are added, and by filtering out dominated pairs. If $(i', j, 0) \in D_{i,l}$ and there is no other $(i'', j'', 0) \in D_{i,l}$ with $i' \neq i''$ and $j \neq j''$, it implies that $|\text{CLCS}(s_1[1,i'], s_2[1,j], P)| = l$. In this way an optimal solution is found for the specific subproblem.

Algorithm by Deorowicz [13].. Just like the previous approach, this algorithm is a so-called sparse approach. The matrix utilized for the calculations is processed for each level $k = 0, \ldots, |P|$ in a row-wise manner and an ordered list is maintained to store for each rank (representing the assumed length of an optimal solution) the lowest possible column number. Furthermore, a two-dimensional matrix T is used to store computed values from the current and previous levels. For each row i and column j where $s_1[i] = s_2[j]$, the list entries are recalculated. If $s_1[i] = s_2[j] \neq P[k]$, then the value for the match at (i, j) is calculated from the highest rank in the list with a column number lower than j. Otherwise, if $s_1[i] = s_2[j] = P[k]$, the value is calculated from matrix T. On completion, the highest rank in the list corresponds to the length of an optimal solution.

Improvements of Deorowicz's algorithm were introduced by Deorowicz and Obstoj [15]. They utilize so–called external–entry points (EEP) [28] 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 Iliopoulos and Rahman [16]: 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.

Table 1: Benchmark suite `Real` from [15].

data set	number of sequences	sequence length (min, med, max)	$\lvert\Sigma\rvert$	origin
$ds0$	7	(111, 124, 134)	20	[11]
$ds1$	6	(124, 149, 185)	20	[11]
$ds2$	6	(131, 142, 160)	20	[11]
$ds3$	5	(189, 277, 327)	20	[11]
$ds4$	6	(98, 114, 123)	20	[29]

6.1. Benchmark Instances

First of all, the benchmark instances are available at `https://www.ac.tuwien.ac.at/files/resources/instances/clcs/2d-clcs.zip`

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), $p' = \frac{|P|}{n} \in \left\{\frac{1}{50}, \frac{1}{20}, \frac{1}{10}, \frac{1}{4}, \frac{1}{2}\right\}$ (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 instances. First, a pattern string P was created uniformly at random, that is, each character from Σ 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, $|P|$ different positions were randomly chosen in each input string. 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 Σ. This procedure ensures that at least one feasible CLCS solution exists for each benchmark instances. Unfortunately, none of the artificial benchmarks from [15] and [18] 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 [15] based on strings representing real biological sequences[2]. This benchmark set is henceforth called `Real`. It has its origins in experimental studies on the constrained multiple sequence alignment (CMSA) problem considered in [29, 11]. Each possible pair of sequences from this data set, together with a pattern string, was used in [15] 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, Chin et al. [11] provided four sets of strings containing RNase sequences with lengths from 111 to 327. In contrast, set $ds4$—containing aspartic acid protease family sequences—was provided by Lu and Huang [29], also in the context of the CMSA problem. Overall, benchmark set `Real` consists of 121 problem instances.

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.

[2]Available at `http://sun.aei.polsl.pl/~sdeor/pub/do09-ds.zip`.

Table 2: Instances with $p' = \frac{|P|}{n} = \frac{1}{50}$: Average runtimes in seconds.

$\lvert\Sigma\rvert$	n	$\overline{\lvert s\rvert}$	`Chin`	`Deo`	`AE`	`IR`	`Hung`	`A`*
4	100	60.9	0.2	< **0.1**	< **0.1**	< **0.1**	< **0.1**	< **0.1**
4	500	319.3	< **0.1**	0.1	0.2	6.5	0.1	< **0.1**
4	1000	646.3	0.2	1	1.3	86.4	0.5	< **0.1**
12	100	40.1	< **0.1**	0.1	< **0.1**	0.1	< **0.1**	< **0.1**
12	500	216.0	< **0.1**	0.1	0.2	2.9	0.2	< **0.1**
12	1000	435.5	0.3	0.5	1.4	39.4	1	**0.1**
20	100	33.5	< **0.1**	0.1	< **0.1**	< **0.1**	< **0.1**	< **0.1**
20	500	175.7	< **0.1**	0.1	0.2	2.2	0.2	< **0.1**
20	1000	355.4	0.3	0.5	1.4	26.6	1.1	< **0.1**

Table 3: Instances with $p' = \frac{|P|}{n} = \frac{1}{20}$: Average runtimes in seconds.

$\lvert\Sigma\rvert$	n	$\overline{\lvert s\rvert}$	`Chin`	`Deo`	`AE`	`IR`	`Hung`	`A`*
4	100	61.9	0.1	< **0.1**	< **0.1**	< **0.1**	< **0.1**	< **0.1**
4	500	323.0	0.1	0.5	0.4	15.7	0.2	< **0.1**
4	1000	645.9	0.9	1.8	3.4	215.5	1.2	**0.1**
12	100	41.0	< **0.1**	0.1	< **0.1**	0.1	< **0.1**	< **0.1**
12	500	215.3	0.1	0.2	0.4	5.3	0.3	< **0.1**
12	1000	437.0	0.9	1.1	3.4	69.2	2.2	**0.2**
20	100	32.2	< **0.1**	< **0.1**	< **0.1**	< **0.1**	< **0.1**	< **0.1**
20	500	170.9	0.1	0.2	0.3	3.3	0.2	< **0.1**
20	1000	348.4	1	1.1	3.5	40.6	1.7	**0.2**

Table 4: Instances with $p' = \frac{|P|}{n} = \frac{1}{10}$: : Average runtimes in seconds.

$\lvert\Sigma\rvert$	n	$\overline{\lvert s\rvert}$	`Chin`	`Deo`	`AE`	`IR`	`Hung`	`A`*
4	100	62.6	< **0.1**	< **0.1**	< **0.1**	0.1	< **0.1**	< **0.1**
4	500	320.9	0.3	0.6	0.9	26.8	0.4	< **0.1**
4	1000	646.4	1.8	3.5	9.2	331.2	3.3	< **0.1**
12	100	40.5	< **0.1**	0.1	< **0.1**	0.1	< **0.1**	< **0.1**
12	500	207.1	0.2	0.3	0.9	7.3	0.3	< **0.1**
12	1000	419.0	2.1	2.2	8.3	91.1	2.7	**0.2**
20	100	31.1	< **0.1**	< **0.1**	< **0.1**	< **0.1**	< **0.1**	< **0.1**
20	500	157.4	0.2	0.3	0.9	5.3	0.2	< **0.1**
20	1000	317.9	1.8	2.1	8.4	68.1	2	< **0.1**

Table 5: Instances with $p' = \frac{|P|}{n} = \frac{1}{4}$: Average runtimes in seconds.

$\lvert\Sigma\rvert$	n	$\overline{\lvert s\rvert}$	`Chin`	`Deo`	`AE`	`IR`	`Hung`	`A`*
4	100	63.2	< **0.1**	< **0.1**	< **0.1**	0.1	< **0.1**	< **0.1**
4	500	320.1	0.6	1.4	2.7	34.8	0.5	< **0.1**
4	1000	642.5	5	6.6	113.6	436.6	4.5	**0.1**
12	100	39.9	< **0.1**	0.1	< **0.1**	0.1	< **0.1**	< **0.1**
12	500	203.0	0.6	0.7	3	18.7	0.3	< **0.1**
12	1000	413.2	5.3	5.7	112	213.2	3.2	< **0.1**
20	100	35.7	< **0.1**	< **0.1**	< **0.1**	0.1	< **0.1**	< **0.1**
20	500	175.5	0.6	0.6	3.3	14.4	0.3	< **0.1**
20	1000	351.1	5.2	5.9	105.4	154.8	1.8	**0.1**

Table 6: Instances with $p' = \frac{|P|}{n} = \frac{1}{2}$: Average runtimes in seconds.

$\lvert\Sigma\rvert$	n	$\overline{\lvert s\rvert}$	`Chin`	`Deo`	`AE`	`IR`	`Hung`	`A`*
4	100	63.9	< **0.1**	< **0.1**	< **0.1**	0.2	< **0.1**	< **0.1**
4	500	325.5	1.4	1.5	22.5	60.6	0.4	< **0.1**
4	1000	652.5	19.1	12.6	336.5	739.4	3.6	< **0.1**
12	100	54.6	0.1	< **0.1**	< **0.1**	0.1	< **0.1**	< **0.1**
12	500	276.5	1.4	1.4	23.9	34.2	0.2	< **0.1**
12	1000	544.3	17.8	11.3	347.5	362.2	2.4	**0.1**
20	100	53.0	< **0.1**	0.1	< **0.1**	0.1	< **0.1**	< **0.1**
20	500	264.9	1.2	1.3	21.5	30.6	0.2	< **0.1**
20	1000	524.5	18.8	11.1	341	278.8	1.5	**0.1**

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

data set	P	$\overline{\lvert s\rvert}$	`Chin`	`Deo`	`AE`	`IR`	`Hung`	`A`*
$ds0$	HKH	60.62	0.012	0.015	0.012	0.026	0.017	**0.011**
$ds1$	HKH	64.00	**0.012**	0.017	0.013	0.032	0.019	0.015
$ds1$	HKSH	63.93	**0.011**	0.021	0.017	0.033	0.017	**0.011**
$ds1$	HKSTH	63.87	0.016	0.022	0.019	0.043	0.024	**0.012**
$ds2$	HKSH	79.60	0.015	0.020	0.016	0.030	0.052	**0.012**
$ds2$	HKSTH	77.87	**0.013**	0.018	0.016	0.030	0.051	**0.013**
$ds3$	HKH	103.90	0.018	0.026	0.019	0.138	0.188	**0.014**
$ds4$	DGGG	43.87	**0.012**	0.022	0.014	0.023	0.049	**0.012**

- `Chin`: Algorithm by Chin et al. [12];
- `Deo`: Algorithm by Deorowicz [13];
- `AE`: Algorithm by Arslan and Eğecioğlu [10];
- `IR`: Algorithm by Iliopoulos and Rahman [16];
- `Hung`: Algorithm by Hung et al. [18].

The source code of this project is accessible at `https://www.ac.tuwien.ac.at/files/resources/software/clcs.zip`.

In general, all algorithms could find optimal solutions and prove their optimality for all instances. However, the required runtimes differ 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 $\overline{|s|}$ 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.
- The first algorithm that starts losing efficiency with growing input string length is `IR`. 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 [13], `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. [18].

- With increasing p' and thus 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 Real (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 IR 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 P 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 Deo, 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 $D_{i,l}$ (which corresponds to the amount of non-dominated extensions of the nodes from $D_{i-1,l-1}$). 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 $p' = \frac{|P|}{n}$. The curve denoted by Max (see legends) is the theoretical upper bound on the number of created nodes, which is $|s_1| \times |s_2| \times |P|$ for an instance (s_1, s_2, P, Σ). 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 $p' \geq \frac{1}{4}$ onward and $|\Sigma| = 20$ from $p' \geq \frac{1}{20}$ onward.

Table 8: Results for instances with different degrees of similarity (θ) of the input strings. The similarity of the input strings grows with an increasing value of θ.

θ	$\overline{\|s\|}$	Chin	A*
0.1	41.3	0.060	**0.050**
0.2	43.8	0.070	**0.050**
0.5	55.0	0.061	**0.052**
0.8	73.2	**0.050**	0.055
0.9	82.5	**0.050**	0.075

6.3. Additional Experimental Evaluation and Findings

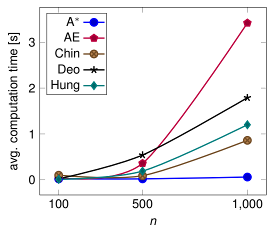
From a more practical point of view our results suggest that, the more misleading the heuristic function used by our A* for a specific problem instance is, the higher will be its running time. More specifically, the heuristic employed in our A* algorithm seems more misleading when the input strings are rather similar. In order to verify this impression, we conducted an additional set of experiments. First, we generated an additional set of problem instances with different degrees of similarity in the input strings. For example, a similarity of $\theta = 0.3$ means that, on average, 30% of the positions in the two input strings have the same character. We generated 10 problem instances with input string length $n = 100$ for each similarity degree $\theta \in \{0.1, 0.2, 0.5, 0.8, 0.9\}$ and an alphabet size of $|\Sigma| = 12$. Moreover, the same pattern string $P =$ abbbcbcbdb was used for all instances.

Running times of our A* algorithm are shown in comparison to algorithm Chin in Table 8. Results indeed confirm our observation from above. That is, when the degree of similarity is rather low, our A* search is faster (see the results for $\theta \in \{0.1, 0.2, 0.5\}$). On the other side, when the degree of similarity is rather high ($\theta \in \{0.8, 0.9\}$), Chin is faster. This is because in the case of instances with a rather high θ-value, a significant amount of time of the overall running time of A* is spent to calculate the upper bound values of the generated nodes. However, as shown in our main experimental evaluation, the A* search can be expected to outperform the competitor algorithms in most other cases, especially the harder ones.

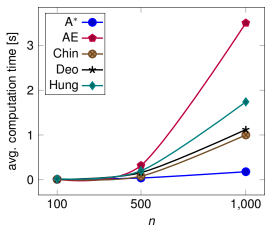
7. Conclusions and Future Work

In this paper we considered the constrained longest common subsequence (CLCS) problem. The problem is well studied in the literature, which offers algorithms based on dynamic programming as well as sparse approaches. In contrast, we presented an A* search for this problem, which is guided by tight upper bound function for the LCS problem. The effectivity of this approach was demonstrated by comparing it to several other so far leading algorithms from the literature. The A* search is able to solve all artificially generated benchmarks as well as the real benchmark instances in a fraction of a second. More specifically, the running times required by A* are about an order of magnitude smaller than those of the second-best algorithm. Interestingly, the performance of A* does not degrade much with an increase of the instance size, which is not the case for the other algorithms from the literature. We conclude that A* search is a tool that has a great potential to be used for the study of similarities between sequences. In fact, our A* search is the new state-of-the-art method for the CLCS problem.

In future work, we plan extend this A* search towards the general CLCS problem with an arbitrary number of input strings, which is an $\mathcal{NP}$–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 [30, 31]. 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 [32] in order to obtain high-quality heuristic solutions already early during the search process.

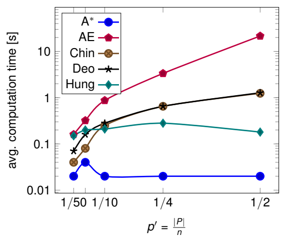


(a) $|\Sigma| = 4$

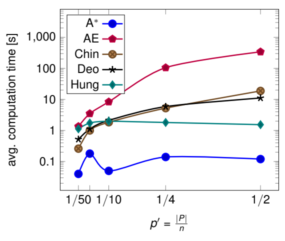


(b) $|\Sigma| = 20$

Figure 2: Average computation times of the algorithms for $p' = \frac{1}{20}$.

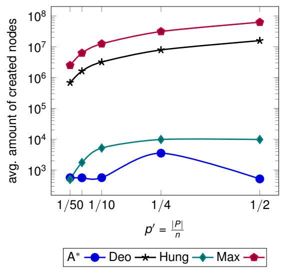


(a) $n = 500$

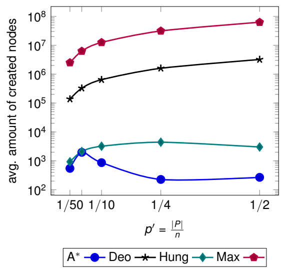


(b) $n = 1000$

Figure 3: Average computation times of the algorithms for $|\Sigma| = 20$.



(a) $|\Sigma| = 4$



(b) $|\Sigma| = 20$

Figure 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. Christian Blum acknowledges the financial support by project CI-SUSTAIN funded by the Spanish Ministry of Science and Innovation (PID2019-104156GB-I00).

[1] D. Maier, The complexity of some problems on subsequences and supersequences, Journal of the ACM 25 (2) (1978) 322–336.

[2] T. Jiang, G. Lin, B. Ma, K. Zhang, A general edit distance between RNA structures, Journal of Computational Biology 9 (2) (2002) 371–388.

[3] J. Storer, Data Compression: Methods and Theory, Computer Science Press, MD, USA, 1988.

[4] J. B. Kruskal, An overview of sequence comparison: Time warps, string edits, and macromolecules, SIAM Review 25 (2) (1983) 201–237.

[5] K.-M. Chao, L. Zhang, Sequence Comparison – Theory and Methods, Springer, London, UK, 2009.

[6] S. S. Adi, M. D. Braga, C. G. Fernandes, C. E. Ferreira, F. V. Martinez, M.-F. Sagot, M. A. Stefanes, C. Tjandraatmadja, Y. Wakabayashi, Repetition-free longest common subsequence, Discrete Applied Mathematics 158 (12) (2010) 1315–1324.

[7] T. Jiang, G.-H. Lin, B. Ma, K. Zhang, The longest common subsequence problem for arc-annotated sequences, in: Annual Symposium on Combinatorial Pattern Matching, Springer, 2000, pp. 154–165.

[8] S. R. Chowdhury, M. Hasan, S. Iqbal, M. S. Rahman, et al., Computing a longest common palindromic subsequence, Fundamenta Informaticae 129 (4) (2014) 329–340.

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

[10] A. N. Arslan, Ö. Eğecioğlu, Algorithms for the constrained longest common subsequence problems, International Journal of Foundations of Computer Science 16 (06) (2005) 1099–1109.

[11] F. Y. Chin, N. Ho, T. Lam, P. W. Wong, M. Chan, Efficient constrained multiple sequence alignment with performance guarantee, Journal of Bioinformatics and Computational Biology 3 (1) (2005) 1–8.

[12] F. Y. Chin, A. De Santis, A. L. Ferrara, N. Ho, S. Kim, A simple algorithm for the constrained sequence problems, Information Processing Letters 90 (4) (2004) 175–179.

[13] S. Deorowicz, Fast algorithm for constrained longest common subsequence problem, Theoretical and Applied Informatics 19 (2) (2007) 91–102.

[14] J. W. Hunt, T. G. Szymanski, A fast algorithm for computing longest common subsequences, Communications of the ACM 20 (5) (1977) 350–353.

[15] S. Deorowicz, J. Obstój, Constrained longest common subsequence computing algorithms in practice, Computing and Informatics 29 (3) (2012) 427–445.

[16] C. S. Iliopoulos, M. S. Rahman, New efficient algorithms for the LCS and constrained LCS problems, Information Processing Letters 106 (1) (2008) 13–18.

[17] W. C. Ho, K. S. Huang, C. B. Yang, A fast algorithm for the constrained longest common subsequence problem with small alphabet, in: Proceedings of the 34th Workshop on Combinatorial Mathematics and Computation Theory, Taichung, Taiwan, 2017, pp. 13–25.

[18] S.-H. Hung, C.-B. Yang, K.-S. Huang, A diagonal-based algorithm for the constrained longest common subsequence problem, in: Proceedings of ICS 2018 – the 23rd International Computer Symposium, Springer Singapore, 2019, pp. 425–432.

[19] N. Nakatsu, Y. Kambayashi, S. Yajima, A longest common subsequence algorithm suitable for similar text strings, Acta Informatica 18 (2) (1982) 171–179.

[20] M. Djukanovic, G. Raidl, C. Blum, A beam search for the longest common subsequence problem guided by a novel approximate expected length calculation, in: Proceedings of LOD 2019 – the 5th International Conference on Machine Learning, Optimization, and Data Science, Springer International Publishing, 2019, pp. 154–167.

[21] M. Djukanovic, G. Raidl, C. Blum, Heuristic approaches for solving the longest common squared subsequence problem, in: Proceedings of EUROCAST 2019 – the 17th International Conference on Computer Aided Systems Theory, To appear., 2019.

[22] C. Blum, M. J. Blesa, M. López-Ibáñez, Beam search for the longest common subsequence problem, Computers and Operations Research 36 (12) (2009) 3178–3186.

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

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

[25] S. Russell, P. Norvig, Artificial intelligence: a modern approach.

[26] G. S. Brodal, M. Kutz, K. Kaligosi, I. Katriel, Faster algorithms for computing longest common increasing subsequences, Journal of Discrete Algorithms 9 (4) (2011) 314–325.

[27] P. v. E. Boas, Preserving order in a forest in less than logarithmic time and linear space, Information Processing Letters 6 (3) (1977) 80–82.

[28] D. He, A. N. Arslan, A space-efficient algorithm for the constrained pairwise sequence alignment problem, Genome Informatics 16 (2) (2005) 237–246.

[29] C. L. Lu, Y. P. Huang, A memory-efficient algorithm for multiple sequence alignment with constraints, Bioinformatics 21 (1) (2005) 20–30.

[30] Z. Gotthilf, D. Hermelin, G. M. Landau, M. Lewenstein, Restricted LCS, in: Proceedings of SPIRE 2010 – the 17th International Symposium on String Processing and Information Retrieval, Springer, 2010, pp. 250–257.

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

[32] S. Zilberstein, Operational rationality through compilation of anytime algorithms, AI Magazine 16 (2) (1995) 79–79.