On Solving a Generalized Constrained Longest Common Subsequence Problem

Marko Djukanovic[1], Christoph Berger[1,2](✉), Günther R. Raidl[1](✉), and Christian Blum[2](✉)

[1] Institute of Logic and Computation, TU Wien, Vienna, Austria
{djukanovic,raidl}@ac.tuwien.ac.at, cberger03@gmail.com
[2] Artificial Intelligence Research Institute (IIIA-CSIC), Campus UAB, Bellaterra, Spain
christian.blum@iiia.csic.es

Abstract. Given a set of two input strings and a pattern string, the constrained longest common subsequence problem deals with finding a longest string that is a subsequence of both input strings and that contains the given pattern string as a subsequence. This problem has various applications, especially in computational biology. In this work we consider the $\mathcal{NP}$–hard case of the problem in which more than two input strings are given. First, we adapt an existing A* search from two input strings to an arbitrary number m of input strings ($m \geq 2$). With the aim of tackling large problem instances approximately, we additionally propose a greedy heuristic and a beam search. All three algorithms are compared to an existing approximation algorithm from the literature. Beam search turns out to be the best heuristic approach, matching almost all optimal solutions obtained by A* search for rather small instances.

Keywords: Longest common subsequences · Constrained subsequences · Beam search · A^* search

1 Introduction

Strings are commonly used to represent DNA and RNA in computational biology, and it is often necessary to obtain a measure of similarity for two or more input strings. One of the most well-known measures is calculated by the so-called *longest common subsequence* (LS) problem. Given a number of input strings, this problem asks to find a longest string that is a subsequence of all input strings. Hereby, a *subsequence* t of a string s is obtained by deleting zero or more characters from s. Apart from computational biology, the LCS problem finds also application in video segmentation [3] and text processing [13], just to name a few.

During the last three decades, several variants of the LCS problem have arisen from practice. One of these variants is the *constrained longest common subsequence* (CLCS) problem [14], which can be stated as follows. Given m input strings and a pattern string P, we seek for a longest common subsequence of the input strings that has P as a subsequence. This problem presents a useful measure of similarity when additional information concerning common structure of the input strings is known beforehand. The most studied CLCS variant is the one with only two input strings (2–CLCS); see, for example, [2,5,14]. In addition to these works from the literature, we recently proposed an A* search for the 2–CLCS problem [6] and showed that this algorithm is approximately one order of magnitude faster than other exact approaches.

In the following we consider the general variant of the CLCS problem with $m \geq 2$ input strings $S = \{s_1, \ldots, s_m\}$, henceforth denoted by m–CLCS. Note that the m–CLCS problem is $\mathcal{NP}$–hard [1]. An application of this general variant is motivated from computational biology when it is necessary to find the commonality for not just two but an arbitrary number of DNA molecules under the consideration of a specific known structure.

To the best of our knowledge, the approximation algorithm by Gotthilf et al. [9] is the only existing algorithm for solving the general m–CLCS problem so far. We first extend the general search framework and the A* search from [6] to solve the more general m–CLCS problem. For the application to large-scale instances we additionally propose two heuristic techniques: (i) a greedy heuristic that is efficient in producing reasonably good solutions within a short runtime, and (ii) a beam search (BS) which produces high-quality solutions at the cost of more time. The experimental evaluation shows that the BS is the new state-of-the-art algorithm, especially for large problem instances.

The rest of the paper is organized as follows. Section 2 describes a greedy heuristic for the m–CLCS problem. In Sect. 3 the general search framework for the m–CLCS problem is presented. Section 4 describes the A* search, and in Sect. 5 the beam search is proposed. In Sect. 6, our computational experiments are presented. Section 7 concludes this work and outlines directions for future research.

2 A Fast Heuristic for the *m*–CLCS Problem

Henceforth we denote the length of a string s over a finite alphabet Σ by $|s|$, and the length of the longest string from the set of input strings $(s_1, \ldots, s_m)$ by n, i.e., $n := \max\{|s_1|, \ldots, |s_m|\}$. The j-th letter of a string s is denoted by $s[j]$, $j = 1, \ldots, |s|$, and for $j > |s|$ we define $s[j] = \varepsilon$, 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 $s[j, j']$, $j = 1, \ldots, |s|$, $j' = j, \ldots, |s|$. If $j > j'$, then $s[j, j'] = \varepsilon$. The concatenation of a string s and a letter $c \in \Sigma$ is written as $s \cdot c$. Finally, let $|s|_c$ be the number of occurrences of letter $c \in \Sigma$ in s. We make use of two data structures created during preprocessing to allow an efficient search:

– For each $i = 1, \ldots, m$, $j = 1, \ldots, |s_i|$, and $c \in \Sigma$, $\mathit{Succ}[i, j, c]$ stores the minimal position index x such that $x \geq j \wedge s_i[x] = c$ or -1 if c does not occur in s_i from position j onward. This structure is built in $O(m \cdot n \cdot |\Sigma|)$ time.

– For each $i = 1, \ldots, m,\ u = 1, \ldots, |P|$, *Embed*$[i, u]$ stores the right-most position x of s_i such that $P[u, |P|]$ is a subsequence of $s_i[x, |s_i|]$. If no such position exists, *Embed*$[i, u] := -1$. This table is built in $O(|P| \cdot m)$ time.

In the following we present Greedy, a heuristic for the m–CLCS problem inspired by the well-known Best–Next heuristic [11] for the LCS problem. Greedy is pseudo-coded in Algorithm 1. The basic principle is straight-forward. The algorithm starts with an empty solution string $s := \epsilon$ and proceeds by appending, at each construction step, exactly one letter to s. The choice of the letter to append is done by means of a greedy function. The procedure stops once no more letters can be added. The basic data structure of the algorithm is a *position vector* $\mathbf{p}^s = (p^s_1, \ldots, p^s_m) \in \mathbb{N}^m$ which is initialized to $\mathbf{p}^s := (1, \ldots, 1)$ at the beginning. The superscript indicates that this position vector depends on the current (partial) solution s. Given $\mathbf{p}^s$, $s_i[p^s_i, |s_i|]$ for $i = 1, \ldots, m$ refer to the substrings from which letters can still be chosen for extending the current partial solution s. Moreover, the algorithm starts with a pattern position index $u := 1$. The meaning of u is that $P[u, |P|]$ is the substring of P that remains to be included as a subsequence in s. At each construction step, first, a subset $\Sigma_{\text{feas}} \subseteq \Sigma$ of letters is determined that can feasibly extend the current partial solution s, ensuring that the final outcome contains pattern P as a subsequence. More specifically, Σ_{feas} contains a letter $c \in \Sigma$ iff (i) c appears in all strings $s_i[p^s_i, |s_i|]$ and (ii) $s{\cdot}c$ can be extended towards a solution that includes pattern P. Condition (ii) is fulfilled if $u = |P| + 1$, $P[u] = c$, or $\mathit{Succ}[i, p^s_i, c] < \mathit{Embed}[i, u]$ for all $i = 1, \ldots, m$ (assuming that there is at least one feasible solution). These three cases are checked in the given order, and with the first case that evaluates to true, condition (ii) evaluates to true; otherwise, condition (ii) evaluates to false. Next, dominated letters are removed from Σ_{feas}. For two letters $c, c' \in \Sigma_{\text{feas}}$, we say that c *dominates* c' iff $\mathit{Succ}[i, p^s_i, c] \leq \mathit{Succ}[i, p^s_i, c']$ for all $i = 1, \ldots, m$. Afterwards, the remaining letters in Σ_{feas} are evaluated by the greedy function explained below, and a letter c^* that has the best greedy value is chosen and appended to s. Further, the position vector $\mathbf{p}^s$ is updated w.r.t. letter c^* by $p^s_i := \mathit{Succ}[i, p^s_i, c^*] + 1$, $i = 1, \ldots, m$. Moreover, u is increased by one if $c^* = P[u]$. These steps are repeated until $\Sigma_{\text{feas}} = \emptyset$, and the greedy solution s is returned.

The greedy function used to evaluate each letter $c \in \Sigma_{\text{feas}}$ is

$$g(\mathbf{p}^s, u, c) = \frac{1}{l_{\min}(\mathbf{p}^s, c) + \mathbb{1}_{P[u]=c}} + \sum_{i=1}^{m} \frac{\mathit{Succ}[i, p^s_i, c] - p^s_i + 1}{|s_i| - p^s_i + 1}, \qquad (1)$$

where $l_{\min}(\mathbf{p}^s, c)$ is the length of the shortest remaining part of any of the input strings when considering letter c appended to the solution string and thus consumed, i.e., $l_{\min} := \min\{|s_i| - \mathit{Succ}[i, p^s_i, c] \mid i = 1, \ldots, m\}$, and $\mathbb{1}_{P[u]=c}$ evaluates to one if $P[u] = c$ and to zero otherwise. Greedy chooses at each construction step a letter that minimizes $g()$. The first term of $g()$ penalizes letters for which the $l_{\min}$ is decreased more and which are not the next letter from $P[u]$. The second term in Eq. (1) represents the sum of the ratios of characters that are skipped (in relation to the remaining part of each input string) when extending the current solution s with letter c.

Algorithm 1. GREEDY: a heuristic for the m–CLCS problem

```
 1: Input: problem instance $(S, P, \Sigma)$
 2: Output: heuristic solution $s$
 3: $s \leftarrow \varepsilon$
 4: $p_i^s \leftarrow 1$, $i = 1, \ldots, m$
 5: $u \leftarrow 1$
 6: $\Sigma_{\text{feas}} \leftarrow$ set of feasible and non-dominated letters for extending $s$
 7: while $\Sigma_{\text{feas}} \neq \emptyset$ do
 8:     $c^* \leftarrow \arg\min\{g(\mathbf{p}^s, u, c) \mid c \in \Sigma_{\text{feas}})\}$
 9:     $s \leftarrow s \cdot c^*$
10:     for $i \leftarrow 1$ to $m$ do
11:         $p_i^s \leftarrow Succ[i, p_i^s, c^*] + 1$
12:     end for
13:     if $P[u] = c^*$ then
14:         $u \leftarrow u + 1$ // consider next letter in $P$
15:     end if
16:     $\Sigma_{\text{feas}} \leftarrow$ set of feasible and non-dominated letters for extending $s$
17: end while
18: return $s$
```

3 State Graph for the m–CLCS Problem

This section describes the state graph for the m–CLCS problem, in which paths from a dedicated root node to inner nodes correspond to (meaningful) partial solutions, paths from the root to sink nodes correspond to complete solutions, and directed arcs represent (meaningful) extensions of partial solutions. Note that the state graph for the m–CLCS problem is an extension of the state graph of the 2–CLCS problem [6].

Given an m–CLCS problem instance $I = (S, P, \Sigma)$, let s be any string over Σ that is a common subsequence of all input strings S. Such a (partial) solution s induces a position vector $\mathbf{p}^s$ in a well-defined way by assigning a value to each p_i^s, $i = 1, \ldots, m$, such that $s_i[1, p_i^s - 1]$ is the smallest string among all strings in $\{s_i[1, k] \mid k = 1, \ldots, p_i^s - 1\}$ that contains s as a subsequence. Note that these position vectors are the same ones as already defined in the context of GREEDY. In other words, s induces a subproblem $I[\mathbf{p}^s] := \{s_1[p_1^s, |s_1|], \ldots, s_m[p_m^s, |s_m|]\}$ of the original problem instance. This is because s can only be extended by adding letters that appear in all strings of $s_i[p_i^s, |s_i|]$, $i = 1, \ldots, m$. In this context, let substring $P[1, k']$ of pattern string P be the maximal string among all strings of $P[1, k]$, $k = 1, \ldots, |P|$, 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 $I[\mathbf{p}^s]$, that is, a subsequence of $s_i[p_i^s, |s_i|]$ for all $i = 1, \ldots, m$.

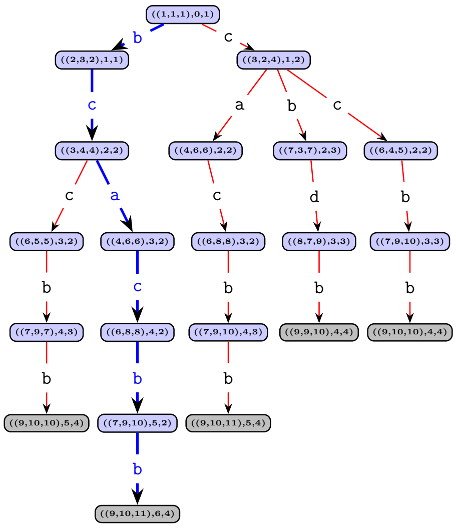


Fig. 1. State graph for the instance $(\{s_1 = \texttt{bcaacbdba}, s_2 = \texttt{cbccadcbbd}, s_3 = \texttt{bbccabcdbba}\}, P = \texttt{cbb}, \Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\})$. There are five non-extensible sink nodes (shown in gray). The longest path corresponds to the optimal solution $s = \texttt{bcacbb}$ with length six and leads to node $v = (\mathbf{p}^v = (9, 10, 11), l^v = 6, u^v = 4)$ (shown in blue). (Color figure online)

The state graph $G = (V, A)$ of our A* search is a *directed acyclic graph* where each node $v \in V$ stores a triple $(\mathbf{p}^v, l^v, u^v)$, with $\mathbf{p}^v$ being a position vector that induces subproblem $I[\mathbf{p}^v]$, l^v is the length of (any) valid partial solution (i.e., path from the root to node v) that induces $\mathbf{p}^v$, and u^v is the length of the longest prefix string of pattern P that is contained as a subsequence in any of the partial solutions that induce node v. Moreover, there is an arc $a = (v, v') \in A$ labeled with letter $c(a) \in \Sigma$ between two nodes $v = (\mathbf{p}^v, l^v, u^v)$ and $v' = (\mathbf{p}^{v'}, l^{v'}, u^{v'})$ iff (i) $l^{v'} = l^v + 1$ and (ii) subproblem $I[\mathbf{p}^{v'}]$ is induced by the partial solution that is obtained by appending letter $c(a)$ to the end of a partial solution that induces v. As mentioned above, we are only interested in meaningful partial solutions, and thus, for feasibly extending a node v, only the letters from Σ_{feas} can be chosen (see Sect. 2 for the definition of Σ_{feas}). An extension $v' = (\mathbf{p}^{v'}, l^{v'}, u^{v'})$ is therefore generated for each $c \in \Sigma_{\text{feas}}$ in the following way: $p_i^{v'} = Succ[i, p_i^v, c] + 1$ for $i = 1, \ldots, m$, $l^{v'} = l^v + 1$, and $u^{v'} = u^v + 1$ in case $c = P[u^v]$, respectively $u^{v'} = u^v$ otherwise.

The *root* node of the state graph is defined by $r = (\mathbf{p}^r = (1, \ldots, 1), l^r = 0, u^r = 1)$ and it thus represents the original problem instance. Sink nodes correspond to non-extensible states. A longest path from the root node to some sink node represents an optimal solution to the m–CLCS problem. Figure 1 shows as example the full state graph for the problem instance $(\{s_1 = \texttt{bcaacbdba}, s_2 = \texttt{cbccadcbbd}, s_3 = \texttt{bbccabcdbba}\}, P = \texttt{cbb}, \Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\})$. 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. Moreover, note that node $((6,5,5),3,2)$ (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 cannot lead to any feasible solution, because any solution starting with bccd does not have $P =$ cbb as a subsequence.

3.1 Upper Bounds

As any upper bound for the general LCS problem is also valid for the m–CLCS problem [8], we adopt the following ones from existing work on the LCS problem. Given a subproblem represented by a node v of the state graph, the upper bound proposed by Blum et al. [4] determines for each letter a limit for the number of its occurrences in any solution that can be built starting from a partial solution represented by v. The upper bound is obtained by summing these values for all letters from Σ:

$$\mathrm{UB}_1(v) = \sum_{c \in \Sigma} \min_{i=1,\ldots,m} \{|s_i[p_i^v, |s_i|]|_c\} \tag{2}$$

where $|s_i[p_i^v, |s_i|]|_c$ is the number of occurrences of letter c in $s_i[p_i^v, |s_i|]$. This bound is efficiently calculated in $O(m \cdot |\Sigma|)$ time by making use of appropriate data structures created during preprocessing; see [7] for more details.

A dynamic programming (DP) based upper bound was introduced by Wang et al. [15]. It makes use of the DP recursion for the classical LCS problem for all pairs of input strings $\{s_i, s_{i+1}\}, i = 1, \ldots, m-1$. In more detail, for each pair $S_i = \{s_i, s_{i+1}\}$, a *scoring matrix* M_i is recursively derived, where entry $M_i[x, y]$, $x = 1, \ldots, |s_i| + 1$, $y = 1, \ldots, |s_{i+1}| + 1$ stores the length of the longest common subsequence of $s_i[x, |s_i|]$ and $s_{i+1}[y, |s_{i+1}|]$. We then get the upper bound

$$\mathrm{UB}_2(v) = \min_{i=1,\ldots,m-1} M_i[p_i^v, p_{i+1}^v]. \tag{3}$$

Neglecting the preprocessing step, this bound can be calculated efficiently in $O(m)$ time. By combining the two bounds we obtain $\mathrm{UB}(v) := \min\{\mathrm{UB}_1(v), \mathrm{UB}_2(v)\}$. This bound is *admissible* for the A* search, which means that its values never underestimate the optimal value of the subproblem that corresponds to a node v. Moreover, the bound is *monotonic*, that is, the estimated upper bound of any child node is never smaller than the upper bound of the parent node. Monotonicity is an important property in A* search, because it implies that no re-expansion of already expanded nodes [6] may occur.

4 A* Search for the m–CLCS Problem

A* search [10] is a well-known technique in the field of artificial intelligence. More specifically, it is a search algorithm based on the best-first principle, explicitly suited for path-finding in large possibly weighted graphs. Moreover, it is an informed search, that is, the nodes to be further pursued are prioritized according to a function that includes a heuristic guidance component. This function is expressed as $f(v) = g(v) + h(v)$ for all nodes $v \in V$ to which a path has already been found. In our context the graph to be searched is the acyclic state graph $G = (V, A)$ introduced in the previous section. The components of the priority function $f(v)$ are:

– the length $g(v)$ of a best-so-far path from the root r to node v, and
– an estimated length $h(v)$ of a best path from node v to a sink node (known as dual bound).

The performance of A* search usually depends on the tightness of the dual bound, that is, the size of the gap between the estimation and the real cost. In our A* search, $g(v) = l^v$ and $h(v) = \mathrm{UB}(v)$, and the search utilizes the following two data structures:

1. The set of all so far created (reached) nodes N: This set is realized as a nested data structure of sorted lists within a hash map. That is, $\mathbf{p}^v$ vectors act as keys of the hash-map, each one mapping to a list that stores all pairs (l^v, u^v) of nodes $(\mathbf{p}^v, l^v, u^v)$ that induce subproblem $I[\mathbf{p}^v]$. This structure was chosen to efficiently check if a specific node was already generated during the search and to keep the memory footprint comparably small.
2. The open list Q: This is a priority queue that stores references to all not-yet-expanded nodes sorted according to non-increasing values $f(v)$. The structure is used to efficiently retrieve the most promising non-expanded node at any moment.

The search starts by adding root node $r = ((1, \ldots, 1), 0, 1)$ to N and Q. Then, at each iteration, the node with highest priority, i.e., the top node of Q, is extended in all possible ways (see Sect. 3), and any newly created node v' is stored in N and Q. If some node v' is reached via the expanded node in a better way, its f-value is updated accordingly. Moreover, it is checked if v' is dominated by some other node from $N[v'] \subseteq N$, where $N[v']$ is the set of all nodes from N representing the same subproblem $I[\mathbf{p}^{v'}]$. If this is the case, v' is discarded. In this context, given $\hat{v}, \overline{v} \in N[v']$ we say that $\hat{v}$ *dominates* $\overline{v}$ iff $l^{\hat{v}} \geq l^{\overline{v}} \wedge u^{\hat{v}} \geq u^{\overline{v}}$. In the opposite case—that is, if any $v'' \in N[v']$ is dominated by v'—node v'' is removed from $N[v'] \subseteq N$ and Q. The node expansion iterations are repeated until the top node of Q is a sink node, in which case a path from the root node to this sink node corresponds to a proven optimal solution. Such a path is retrieved by reversing the path from following each node's predecessor from the sink node to the root node. Moreover, our A* search terminates without a meaningful solution when a specified time or memory limit is exceeded.

5 Beam Search for the m–CLCS Problem

It is well known from research on other LCS variants that *beam search* (BS) is often able to produce high-quality approximate solutions in this domain [8]. For those cases in which our A* approach is not able to deliver an optimal solution in a reasonable computation time, we therefore propose the following BS approach.

Before the start of the BS procedure, Greedy is performed to obtain an initial solution s_{bsf}. This solution can be used in BS for pruning partial solutions (nodes) that provenly cannot be extended towards a solution better than s_{bsf}. Beam search maintains a set of nodes B, called the *beam*, which is initialized with the root node r at the start of the algorithm. Remember that this root node represents the empty partial solution. A single major iteration of BS consists of the following steps:

– Each node $v \in B$ is expanded in all possible ways (see the definition of the state graph) and the extensions are kept in a set V_{ext}. If any node $v \in V_{\text{ext}}$ is a complete node for which l^v is larger than $|s_{\text{bsf}}|$, the best-so-far solution s_{bsf} is updated accordingly.
– Application of function Prune($V_{\text{ext}}, \text{UB}_{\text{prune}}$) (optional): All nodes from V_{ext} whose upper bound value is no more than $|s_{\text{bsf}}|$ are removed. UB_{prune} refers to the utilized upper bound function (see Sect. 3.1 for the options).
– Application of function Filter($V_{\text{ext}}, k_{\text{best}}$) (optional): this function examines the nodes from V_{ext} and removes dominated ones. Given $v, v' \in V_{\text{ext}}$, we say in this context that v *dominates* v' iff $p_i^v \leq p_i^{v'}$, for all $i = 1, \dots, m$ $\wedge$ $u^v \geq u^{v'}$. Note that this is a generalization of the domination relation introduced in [4] for the LCS problem. Since it is time-demanding to examine the possible domination for each pair of nodes from V_{ext} if $|V_{\text{ext}}|$ is not small, the domination for each node $v \in V_{\text{ext}}$ is only checked against the best k_{best} nodes from V_{ext} w.r.t. a heuristic guidance function $h(v)$, where k_{best} is a strategy parameter. We will consider several options for $h(v)$ presented in the next section.
– Application of function Reduce(V_{ext}, β): The best at most β nodes are selected from V_{ext} to form the new beam B for the next major iteration; the *beam width* β is another strategy parameter.

These four steps are repeated until B becomes empty. Beam search is thus a kind of incomplete *breadth-first-search*.

5.1 Options for the Heuristic Guidance of BS

Different functions can be used as heuristic guidance of the BS, that is, for the function h that evaluates the heuristic goodness of any node $v = (p^{\text{L},v}, l^v, u^v) \in V$. An obvious choice is, of course, the upper bound UB from Sect. 3.1. Additionally, we consider the following three options.

5.1.1 Probability Based Heuristic

For a probability based heuristic guidance, we make use of a DP recursion from [12] for calculating the probability $\Pr(p, q)$ that any string of length p is a subsequence of a *random string* of length q. These probabilities are computed in a preprocessing step for $p, q = 0, \dots, n$. Remember, in this context that n is the length of the longest input string. Assuming independence among the input strings, the probability $\Pr(s \prec S)$ that a random string s of length p is a common subsequence of all input strings from S is $\Pr(s \prec S) = \prod_{i=1}^{m} \Pr(p, |s_i|)$. Given V_{ext} in some construction step of BS, the question is now how to choose the value p common for all nodes $v \in V_{\text{ext}}$ in order to take profit from the above formula in a sensible heuristic manner. For this purpose, we first calculate

$$p^{\min} = \min_{v \in V_{\text{ext}}} \left(|P| - u^v + 1 \right), \tag{4}$$

where P is the pattern string of the tackled m–CLCS instance. Note that the string $P[p^{\min}, |P|]$ must appear as a subsequence in all possible completions of all nodes from $v \in V_{\text{ext}}$, because pattern P must be a subsequence of any feasible solution. Based on $p^{\min}$, the value of p for all $v \in V_{\text{ext}}$ is then heuristically chosen as

$$p = p^{\min} + \min_{v \in V_{\text{ext}}} \left\lfloor \frac{\min_{i=1,\ldots,m} \{|s_i| - p_i^v + 1\} - p^{\min}}{|\Sigma|} \right\rfloor. \tag{5}$$

The intention here is, first, to let the characters from $P[p^{\min}, |P|]$ fully count, because they will—as mentioned above—appear for sure in any possible extension. This explains the first term ($p^{\min}$) in Eq. (5). The second term is justified by the fact that an optimal m–CLCS solution becomes shorter if the alphabet size becomes larger. Moreover, the solution tends to be longer for nodes v whose length of the shortest remaining string from $I[\mathbf{p}^v]$ is longer than the one of other nodes. We emphasize that this is a heuristic choice which might be improvable. If p would be zero, we set it to one in order to break ties. The final probability-based heuristic for evaluating a node $v \in V_{\text{ext}}$ is then

$$H(v) = \prod_{i=1}^{m} \Pr(p, |s_i| - p_i^v + 1), \tag{6}$$

and those nodes with a larger H–value are preferred.

5.1.2 Expected Length Based Heuristic

In [8] we derived an approximate formula for the expected length of a longest common subsequence of a set of uniform random strings. Before we extend this result to the m–CLCS problem, we state those aspects of the results from [8] that are needed for this purpose. For more information we refer the interested reader to the original article. In particular, from [8] we know that

$$\mathbb{E}[Y] = \sum_{k=1}^{l_{\min}} \mathbb{E}[Y_k], \tag{7}$$

where $l_{\min} := \min\{|s_i| \mid i = 1, \ldots, m\}$, Y is a random variable for the length of an LCS, and Y_k is, for any $k = 1, \ldots, l_{\min}$, a binary random variable indicating whether or not there is an LCS with a length of at least k. $\mathbb{E}[\cdot]$ denotes the expected value of some random variable.

In the context of the m–CLCS problem, a similar formula with the following re-definition of the binary variables is used. Y is now a random variable for the length of an LCS that has pattern string P as a subsequence, and the Y_k are binary random variables indicating whether or not there is an LCS with a length of at least k having P as a subsequence. If we assume the existence of at least one feasible solution, we get $\mathbb{E}[Y] = |P| + \sum_{k=|P|+1}^{l_{\min}} \mathbb{E}[Y_k]$.

For $k = |P|, \ldots, l_{\min}$, let T_k be the set of all possible strings of length k over alphabet Σ. Clearly, there are $|\Sigma|^k$ such strings. For each $s \in T_k$ we define the event Ev_s that s is a subsequence of all input strings from S having P as a subsequence. For simplicity, we assume the independence among events Ev_s and $\mathrm{Ev}_{s'}$, for any $s, s' \in T_k,\ s \neq s'$. With this assumption, the probability that string $s \in T_k$ is a subsequence of all input strings from S is equal to $\prod_{i=1}^{m} \Pr(|s|, |s_i|)$. Further, under the assumption that (i) s is a uniform random string and (ii) the probabilities that s is a subsequence of s_i (denoted by $\Pr(s \prec s_i)$) for $i = 1, \ldots, m$, and the probability that P is a subsequence of s (denoted by $(\Pr(P \prec s))$ are independent, it follows that the probability $P^{\mathrm{CLCS}}(s, S, P)$ that s is a common subsequence of all strings from S having pattern P as a subsequence is equal to $\Pr(|P|, k) \cdot \prod_{i=1}^{m} \Pr(k, |s_i|)$. Moreover, note that, under our assumptions, it holds that $\Pr(P \prec s') = \Pr(P \prec s'') = \Pr(|P|, k)$, for any pair of sampled strings $s', s'' \in T_k$. Therefore, it follows that

$$\begin{aligned}
\mathbb{E}[Y_k] &= 1 - \prod_{s \in T_k} \left(1 - P^{\mathrm{CLCS}}(s, S, P)\right) \\
&= 1 - \left(1 - \left(\prod_{i=1}^{m} \Pr(k, |s_i|)\right) \cdot \Pr(|P|, k)\right)^{|\Sigma|^k}. \qquad (8)
\end{aligned}$$

Using this result, the expected length of a final m–CLCS solution that includes a string inducing node $v \in V$ as a prefix can be approximated by the following (heuristic) expression:

$$\begin{aligned}
&\mathrm{EX}^{\mathrm{CLCS}}(v) \overset{7,8}{=} |P| - u^v + (l_{\min} - (|P| - u^v + 1) + 1) - \\
&\sum_{k=|P|-u^v+1}^{l_{\min}} \left(1 - \left(\prod_{i=1}^{m} \Pr(k, |s_i| - p_i^{\mathrm{L},v} + 1)\right) \cdot \Pr(|P| - u^v, k)\right)^{|\Sigma|^k} \\
&= l_{\min}^v - \sum_{k=|P|-u^v+1}^{l_{\min}^v} \left(1 - \left(\prod_{i=1}^{m} \Pr(k, |s_i| - p_i^{\mathrm{L},v} + 1)\right) \cdot \Pr(|P| - u^v, k)\right)^{|\Sigma|^k}, \qquad (9)
\end{aligned}$$

where $l_{\min}^v = \min\{|s_i| - p_i^{\mathrm{L},v} + 1 \mid i = 1, \ldots, m\}$. To calculate this value in practice, one has to take care of numerical issues, in particular the large power value $|\Sigma|^k$. We resolve it in the same way as in [8] by applying a Taylor series.

5.1.3 Pattern Ratio Heuristic

So far we have introduced three options for the heuristic function in beam search: the upper bound (Sect. 3.1), the probability based heuristic (Sect. 5.1.1) and the expected length based heuristic (Sect. 5.1.2). With the intention to test, in comparison, a much simpler measure we introduce in the following the pattern ratio heuristic that only depends on the length of the shortest string in $S[\mathbf{p}^v]$ and the length of the remaining part of the pattern string to be covered ($|P| - u^v + 1$). In fact, we might directly use the following function for estimating the goodness of any $v \in V$:

$$R(v) := \frac{\min_{i=1,\ldots,m}(|s_i| - p_i^v + 1)}{|P| - u^v + 1}. \tag{10}$$

In general, the larger $R(v)$, the more preferable should be v. However, note that the direct use of (10) generates numerous ties. In order to avoid a large number of ties, instead of $R(v)$ we use the well-known k-norm $||v||_k^k = \sum_{i=1}^m \left(\frac{|s_i| - p_i^v + 1}{|P| - u^v + 1} \right)^k$, with some $k > 0$. Again, nodes $v \in V$ with a larger $|| \cdot ||_k$-values are preferable. In our experiments, we set $k = 2$ (Euclidean norm).

6 Experimental Evaluation

All algorithms were implemented in C++ using GCC 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 maximal CPU time allowed for each run was set to 15 min, i.e., 900 s. We generated the following set of problem instances for the experimental evaluation. For each combination of the number of input strings $m \in \{10, 50, 100\}$, the length of input strings $n \in \{100, 500, 1000\}$, the alphabet size $|\Sigma| \in \{4, 20\}$ and the ratio $p' = \frac{|P|}{n} \in \left\{\frac{1}{50}, \frac{1}{20}, \frac{1}{10}, \frac{1}{4}, \frac{1}{2}\right\}$, ten instances were created, each one as follows. First, P is generated uniformly at random. Then, each string $s_i \in S$ is generated as follows. First, P is copied, that is, $s_i := P$. Then, s_i is augmented in $n - |P|$ steps by single random characters. The position for the new character is selected randomly between any two consecutive characters of s_i, at the beginning, or at the end of s_i. This procedure ensures that at least one feasible solution exists for each instance. The benchmarks are available at https://www.ac.tuwien.ac.at/files/resources/instances/m-clcs.zip. Overall, we thus created and use 900 benchmark instances.

We include the following six algorithms (resp. algorithm variants) in our comparison: (*i*) the approximation algorithm from [9] (Approx), (*ii*) Greedy from Sect. 2, and (*iii*) the four beam search configurations differing only in the heuristic guidance function. These BS versions are denoted as follows. Bs-Ub refers to BS using the upper bound, Bs-Prob refers to the use of the probability based heuristic, Bs-Ex to the use of expected length based heuristic, and BS-Pat to the use of the pattern ratio heuristic. Moreover, we include the information of how many instances of each type were solved to optimality by the exact A* search. Concerning the beam search, parameters β (the maximum number of nodes kept for the next iteration) and k_{best} (the extent of filtering) are crucial for obtaining good results. After tuning we selected $\beta = 2000$ and $k_{\mathrm{best}} = 100$. Moreover, the tuning procedure indicated that function upper bound based pruning is indeed beneficial.

Table 1. Results for instances with $p' = \frac{|P|}{n} = \frac{1}{20}$.

$\lvert\Sigma\rvert$	m	n	**Approx**		**Greedy**		Bs-Ub		**Bs-Prob**		**Bs-Ex**		Bs-Pat		**A***	
			$\overline{\lvert s\rvert}$	$\bar{t}[s]$	$\overline{\lvert s\rvert}$	$\bar{t}[s]$	$\overline{\lvert s\rvert}$	$\bar{t}[s]$	$\overline{\lvert s\rvert}$	$\bar{t}[s]$	$\overline{\lvert s\rvert}$	$\bar{t}[s]$	$\overline{\lvert s\rvert}$	$\bar{t}[s]$	#	$\bar{t}[s]$
4	10	100	21.4	<0.1	30.8	<0.1	**34.5**	19.2	**34.5**	16.8	**34.5**	21.7	33.4	25.6	3	332.8
4	10	500	119.7	<0.1	162.3	<0.1	181.7	130.1	184.2	163.7	**185.1**	179.8	173.3	192.1	0	-
4	10	1000	244.4	0.1	330.9	0.1	365.7	288.5	372.7	346.7	**374.1**	339.2	343.8	391	0	-
4	50	100	18.7	<0.1	21.3	<0.1	24.3	11.5	24.7	13.3	**24.9**	15.1	24	19.8	0	-
4	50	500	111.1	0.1	127.1	0.1	137.9	98.5	141.2	109.4	**142.2**	115.4	134.2	162.8	0	-
4	50	1000	232.7	0.5	265	0.3	281	226.4	290.1	267.6	**291.3**	289.4	273	366.4	0	-
4	100	100	17.6	<0.1	18.5	<0.1	22.3	11.6	22.4	9.6	**22.5**	13.60	21.9	19.7	0	-
4	100	500	109.4	0.2	119.5	0.2	128.9	101.2	131.9	86.2	**132.4**	119.3	126.6	156	0	-
4	100	1000	227.5	0.8	248	0.9	263.7	244.2	272.0	218.1	**273.0**	232.2	259.2	301.8	0	-
20	10	100	6	<0.1	7.1	<0.1	***7.3**	<0.1	***7.3**	<0.1	***7.3**	<0.1	***7.3**	<0.1	10	<0.1
20	10	500	30.2	<0.1	40	<0.1	46.6	16.9	**47.0**	17.5	46.3	60.0	44.7	57	10	332.1
20	10	1000	56.6	0.1	81.2	0.1	95.7	37.9	**97.8**	45.5	95.4	185.4	87.9	146.3	0	-
20	50	100	***5.0**	<0.1	***5.0**	<0.1	***5.0**	<0.1	***5.0**	<0.1	***5.0**	<0.1	***5.0**	<0.1	10	<0.1
20	50	500	26.9	0.1	28.2	0.1	***29.9**	1.8	***29.9**	1.7	***29.9**	1.3	***29.9**	1.5	10	1.2
20	50	1000	53.1	0.5	58.2	0.5	62.4	17.6	**62.7**	17	62.5	8.6	60.4	34.4	0	-
20	100	100	***5.0**	<0.1	***5.0**	<0.1	***5.0**	<0.1	***5.0**	<0.1	***5.0**	<0.1	***5.0**	<0.1	10	<0.1
20	100	500	26.1	0.2	26.4	0.2	***27.3**	0.3	***27.3**	0.2	***27.3**	0.3	***27.3**	0.3	10	0.3
20	100	1000	52	1	54.7	0.8	57.2	14	***57.3**	13.6	***57.3**	9.4	56.4	17.7	10	86.0

Table 1 reports results for the instances with $p' = \frac{1}{20}$ and Table 2 those for the instances with $p' = \frac{1}{4}$. The remaining numerical results as well as the tuning process are, due to space limitations, reported in a supplementary document that can be downloaded from https://www.ac.tuwien.ac.at/files/resources/supplementary/clcs-suppl.pdf. The first three columns of each table indicate the instance characteristics. Then, for the six competitors we provide in each table row the obtained solution quality and computation time averaged over the 10 instances with the respective characteristics. The best result of each table row is shown in bold. Finally, for A* search we provide in each table row the number of instances solved to optimality (out of 10) and the average runtime required to do so. A preceding asterisk indicates that the respective result is provenly optimal. The results allow to make the following observations.

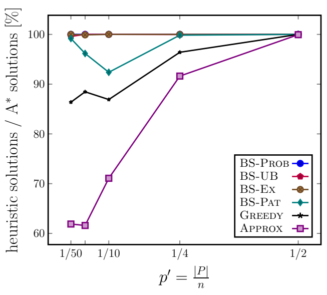
– The m–CLCS problem tends to be most difficult to solve for short pattern strings—that is, low values of $|P|$—and for small alphabet sizes. With growing $|P|$ and $|\Sigma|$, the problem becomes easier. On the one side, this is indicated by the results of the A* search. When $\frac{|P|}{n} = 1/20$ and $|\Sigma| = 4$, A* can only solve three problem instances to optimality. When moving to instances

Table 2. Results for instances with $p' = \frac{|P|}{n} = \frac{1}{4}$.

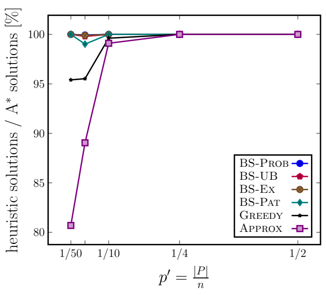
$\|\Sigma\|$	m	n	**Approx**		**Greedy**		Bs-Ub		**Bs-Prob**		**Bs-Ex**		Bs-Pat		**A***	
			$\overline{\|s\|}$	$\overline{t}[s][s]$	$\overline{\|s\|}$	$\overline{t}[s][s]$	$\overline{\|s\|}$	$\overline{t}[s][s]$	$\overline{\|s\|}$	$\overline{t}[s][s]$	$\overline{\|s\|}$	$\overline{t}[s][s]$	$\overline{\|s\|}$	$\overline{t}[s][s]$	#	$\overline{t}[s][s]$
4	10	100	28.6	<0.1	32.2	<0.1	***34.5**	1.1	***34.5**	0.9	***34.5**	1.0	***34.5**	1.5	10	0.2
4	10	500	134.3	<0.1	160.4	<0.1	179.3	45.6	**182.4**	48.8	181.1	98.0	168.6	97	1	660.8
4	10	1000	264.7	0.1	317.4	0.1	350.3	76.8	**361.7**	108	361.4	249.4	330.8	220.2	0	-
4	50	100	26.4	<0.1	26.9	<0.1	***27.5**	<0.1	***27.5**	<0.1	***27.5**	<0.1	***27.5**	<0.1	10	<0.1
4	50	500	130.1	0.1	139.5	0.1	146.2	33.6	**148.3**	28	146.3	19.9	142.7	55.9	0	-
4	50	1000	257.4	0.5	277.3	0.3	291.9	73.6	**296.4**	63.6	289.5	41.1	284.2	107.6	0	-
4	100	100	25.9	<0.1	26.2	<0.1	***26.5**	<0.1	***26.5**	<0.1	***26.5**	<0.1	***26.5**	<0.1	10	<0.1
4	100	500	128.9	0.2	135.8	0.2	140.4	24.6	**140.8**	34.8	140.3	17.4	137.3	45.9	0	-
4	100	1000	256.4	0.8	270.7	0.9	279.7	56.4	**282.5**	73.4	279.0	40.4	273.3	122	0	-
20	10	100	***25.0**	<0.1	***25.0**	<0.1	***25.0**	<0.1	***25.0**	<0.1	***25.0**	<0.1	***25.0**	<0.1	10	<0.1
20	10	500	***125.0**	<0.1	***125.0**	<0.1	***125.0**	<0.1	***125.0**	<0.1	***125.0**	<0.1	***125.0**	<0.1	10	<0.1
20	10	1000	***250.0**	0.1	***250.0**	0.1 0.1	***250.0**	0.1	***250.0**	0.1	***250.0**	0.1	***250.0**	0.1	10	0.1
20	50	100	***25.0**	<0.1	***25.0**	<0.1	***25.0**	<0.1	***25.0**	<0.1	***25.0**	<0.1	***25.0**	<0.1	10	<0.1
20	50	500	***125.0**	0.1	***125.0**	0.1	***125.0**	0.2	***125.0**	0.2	***125.0**	0.1	***125.0**	0.1	10	0.1
20	50	1000	***250.0**	0.5	***250.0**	0.5	***250.0**	0.4	***250.0**	0.5	***250.0**	0.5	***250.0**	0.5	10	0.5
20	100	100	***25.0**	<0.1	***25.0**	<0.1	***25.0**	<0.1	***25.0**	<0.1	***25.0**	<0.1	***25.0**	<0.1	10	<0.1
20	100	500	***125.0**	0.3	***125.0**	0.2	***125.0**	0.3	***125.0**	0.3	***125.0**	0.2	***125.0**	0.2	10	0.3
20	100	1000	***250.0**	1	***250.0**	0.8	***250.0**	1.1	***250.0**	1.1	***250.0**	0.8	***250.0**	1.1	10	1.0

with $|\Sigma| = 20$, A* search can already solve 70 instances to optimality. The corresponding numbers for instances with $\frac{|P|}{n} = 1/4$ are 31 (for $|\Sigma| = 4$) and 90 (for $|\Sigma| = 20$). In fact, in this last case 90 corresponds to all problem instances of this type. On the other side, the decreasing problem difficulty for growing $|P|$ and $|\Sigma|$ is also indicated by the differences between the results of the heuristic algorithms. In fact, for $\frac{|P|}{n} = 1/20$ and $|\Sigma| = 20$ all algorithms are able to solve all 90 problem instances to optimality.

- The reason for the problem difficulty to decrease with growing $|P|$ can be explained as follows. With growing $|P|$, the similarity between the input strings also grows. This results in a decrease of the search space size. Moreover, from [8] we know that EX-type guidance for BS becomes worse with a growing similarity of the input strings. And, in fact, this observation holds also in the case of the m–CLCS problem. For $\frac{|P|}{n} = 1/20$, Bs-Ex delivers in most cases better results than the other BS configurations. However, Bs-Ex seems to lose efficiency for $\frac{|P|}{n} = 1/4$ where Bs-Prob is generally the better choice.
- All our heuristic algorithms significantly improve over Approx, which is the only existing technique from the literature. This also holds for Greedy, which requires approximately the same computation time as Approx. Only when instances are easy to solve—that is, when n and m are small, and $|\Sigma|$ and $|P|$ are rather large—Approx is competitive with our algorithms.
- All versions of BS improve over Greedy. However, this comes at the price of significantly elevated computation times.
- Bs-Pat, which uses the most simplistic guidance heuristic, is clearly inferior to the other three BS variants in terms of solution quality for almost all instance types.

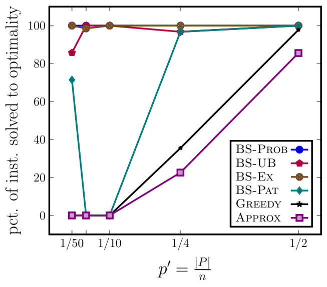


(a) $|\Sigma| = 4$

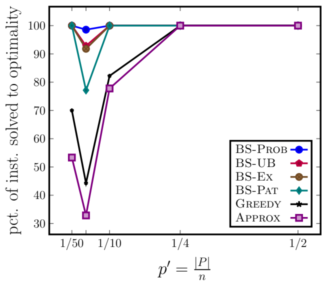


(b) $|\Sigma| = 20$

Fig. 2. Average fraction (in percent) of the length of heuristic solutions with respect to the length of the A* solutions.



(a) $|\Sigma| = 4$



(b) $|\Sigma| = 20$

Fig. 3. Percentage of instances solved to optimality.

– The results of Bs-Ub are comparable with those of Bs-Ex only when $|P|$ and n are both small. This is because the upper bound is especially tight for smaller instances.

Finally, we want to shed some more light on the comparison of the heuristic techniques with A* search. For this purpose, from now on we only consider those instances that can be solved to optimality by A* search. The two plots in Fig. 2 show for each heuristic algorithm and for each value of $\frac{|P|}{n}$ (x-axis) the average fraction (in percent) of the length of heuristic solutions in respect to the length of the A* search solutions. A dot at 100% means that the length of the heuristic solution matches the length of the optimal solution from A* search. The plots show, in particular, that Bs-Prob, Bs-Ub and Bs-Ex always reach at least a value of 98%. Complementary, the two plots in Fig. 3 show for each heuristic algorithm and for each value of $\frac{|P|}{n}$ (x-axis) the percentage of instances that were solved to optimality. Bs-Prob fails to deliver an optimal solution for only one instance of type $m = 10$, $n = 500$, $|\Sigma| = 20$, and $\frac{|P|}{n} = 1/20$.

7 Conclusions and Future Work

We tackled the generalized constrained longest common subsequence problem. First, we presented an exact A* search algorithm. Moreover, apart from a simple greedy heuristic, we also introduced four different variants of beam search that differ in the heuristic function they use for selecting the partial solutions to be further expanded in the subsequent iteration. More specifically, we considered an upper bound, a probability based heuristic, an expected length based heuristic, and a simple greedy criterion. Our approaches are compared to an approximation algorithm from the literature, the only one so far available for the problem. In general, the BS variant using the expected length calculation heuristic is best when the pattern string is rather short, while the BS variant with the probability based heuristic is leading when the pattern string is rather long. Moreover, instances become more easily solvable the longer the pattern is.

Concerning future work, the general search framework derived for the CLCS problem can be further extended towards an arbitrary number of pattern strings. We also intend to search for possibly existing real-world benchmark sets for further testing the performances of the algorithms.

Acknowledgments. This work was partially funded by the Doctoral Program Vienna Graduate School on Computational Optimization, Austrian Science Foundation Project No. W1260-N35. This work was also supported by project CI-SUSTAIN funded by the Spanish Ministry of Science and Innovation (PID2019-104156GB-I00).

References

1. Abboud, A., Backurs, A., Williams, V.V.: Tight hardness results for LCS and other sequence similarity measures. In: Proceedings of FOCS 2015 - the 56th Annual Symposium on Foundations of Computer Science, pp. 59–78. IEEE (2015)
2. Arslan, A.N., Eğecioğlu, Ö.: Algorithms for the constrained longest common subsequence problems. Int. J. Found. Comput. Sci. **16**(06), 1099–1109 (2005)
3. Bezerra, F.N.: A longest common subsequence approach to detect cut and wipe video transitions. In: Proceedings of 17th Brazilian Symposium on Computer Graphics and Image Processing, pp. 154–160. IEEE Explore (2004). https://doi.org/10.1109/SIBGRA.2004.1352956
4. Blum, C., Blesa, M.J., López-Ibáñez, M.: Beam search for the longest common subsequence problem. Comput. Oper. Res. **36**(12), 3178–3186 (2009)
5. Chin, F.Y., De Santis, A., Ferrara, A.L., Ho, N., Kim, S.: A simple algorithm for the constrained sequence problems. Inf. Process. Lett. **90**(4), 175–179 (2004)
6. Djukanovic, M., Berger, C., Raidl, G.R., Blum, C.: An A* search algorithm for the constrained longest common subsequence problem. Technical report AC-TR-20-004, Algorithms and Complexity Group, TU Wien (2020). http://www.ac.tuwien.ac.at/files/tr/ac-tr-20-004.pdf
7. Djukanovic, M., Raidl, G.R., Blum, C.: A heuristic approach for solving the longest common square subsequence problem. In: Moreno-Díaz, R., Pichler, F., Quesada-Arencibia, A. (eds.) EUROCAST 2019. LNCS, vol. 12013, pp. 429–437. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-45093-9_52

8. Djukanovic, M., Raidl, G.R., Blum, C.: A beam search for the longest common subsequence problem guided by a novel approximate expected length calculation. In: Nicosia, G., Pardalos, P., Umeton, R., Giuffrida, G., Sciacca, V. (eds.) LOD 2019. LNCS, vol. 11943, pp. 154–167. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-37599-7_14
9. Gotthilf, Z., Hermelin, D., Lewenstein, M.: Constrained LCS: hardness and approximation. In: Ferragina, P., Landau, G.M. (eds.) CPM 2008. LNCS, vol. 5029, pp. 255–262. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69068-9_24
10. Hart, P., Nilsson, N., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Syst. Sci. Cybern. **4**(2), 100–107 (1968)
11. Huang, K., Yang, C., Tseng, K.: Fast algorithms for finding the common subsequences of multiple sequences. In: Proceedings of ICS 2004 - the 3rd IEEE International Computer Symposium, pp. 1006–1011 (2004)
12. Mousavi, S.R., Tabataba, F.: An improved algorithm for the longest common subsequence problem. Comput. Oper. Res. **39**(3), 512–520 (2012)
13. Storer, J.: Data Compression: Methods and Theory. Computer Science Press, Rockville (1988)
14. Tsai, Y.T.: The constrained longest common subsequence problem. Inf. Process. Lett. **88**(4), 173–176 (2003)
15. Wang, Q., Pan, M., Shang, Y., Korkin, D.: A fast heuristic search algorithm for finding the longest common subsequence of multiple strings. In: Proceedings of the 24th AAAI Conference on Artificial Intelligence. AAAI Press (2010)