Application of A* to the Generalized Constrained Longest Common Subsequence Problem with Many Pattern Strings

Marko Djukanovic[1(✉)], Dragan Matic[1], Christian Blum[2], and Aleksandar Kartelj[3]

[1] Faculty of Natural Sciences and Mathematics, University of Banjaluka, Banjaluka, Bosnia and Herzegovina
{marko.djukanovic,dragan.matic}@pmf.unibl.org
[2] Artificial Intelligence Research Institute (IIIA-CSIC), Campus UAB, Bellaterra, Spain
christian.blum@iiia.csic.es
[3] Faculty of Mathematics, University of Belgrade, Beograd, Serbia

Abstract. This paper considers the constrained longest common subsequence problem with an arbitrary set of input strings and an arbitrary set of pattern strings as input. The problem has applications, for example, in computational biology, serving as a measure of similarity among different molecules that are characterized by common putative structures. We develop an exact A* search to solve it. Our A* search is compared to the only existing competitor from the literature, an Automaton approach. The results show that A* is very efficient for real-world benchmarks, finding provenly optimal solutions in run times that are an order of magnitude lower than the ones of the competitor. Even some of the large-scale real-world instances were solved to optimality by A* search.

1 Introduction

The longest common subsequence (LCS) problem is a prominent string problem. Given is a set of input strings $S = \{s_1, \ldots, s_m\}$, where each s_i consists of letters from a finite alphabet Σ, the goal is to find a string of maximal length that is a common subsequence of all input strings. Even though the problem is easily stated, it is challenging to solve, as it is known to be $\mathcal{NP}$–hard for an arbitrary number ($m > 1$) of input strings [20]. Solutions to LCS problems are commonly used as similarity measures in evolutionary microbiology and computational biology. Identifying LCS solutions to biological sequences (DNA, RNA, or protein sequences) plays a significant role in the field of sequence alignment and pattern discovery. LCS solutions may also serve for the discovery of structural or evolutionary relationships among the inputs sequences [22]. The related literature offers numerous variants of the original LCS problem, arising from practical applications. They are generally obtained by adding further constraints

© Springer Nature Switzerland AG 2022
M. El Yacoubi et al. (Eds.): ICPRAI 2022, LNCS 13364, pp. 53–64, 2022.
https://doi.org/10.1007/978-3-031-09282-4_5

and requirements to the original LCS problem. These variants include, but are not limited to, the longest common palindromic subsequence problem [3], the repetition–free longest common subsequence problem [1], and the arc–preserving longest common subsequence problem [17].

Another variant, the constrained longest common subsequence (CLCS) problem [24], requires a single pattern string p in addition to the set S of m input strings as input. The aim of the CLCS problem is to find a LCS s^* of all strings in S that contains p as a subsequence. This problem is $\mathcal{NP}$–hard as it includes the basic LCS problem as a special case when $p = \epsilon$, where ϵ is the empty string. In practical applications, however, considering more than one pattern string seems often to be required. As an example consider Tang et al. [23], in which `RNase` sequences were aligned with the restriction that each of the three active-site residues `His`(H), `Lyn`(K), and `His`(H) has to be a subsequence of any valid solution. Another example from the biological context concerns [21], in which a large set of real bacterial RNA sequences is considered, under the restriction that solutions must contain 15 different patterns (the so-called *contig primer structures*). Therefore, a generalized version of the CLCS problem, called GCLCS (or (m, k)–CLCS), with arbitrary large sets of input strings and pattern strings was considered in [8]. The GCLCS problem is formally defined as follows. Given a set of $m > 1$ input strings ($S = \{s_1, \ldots, s_m\}$) and a set of $k \geq 1$ pattern strings ($P = \{p_1, \ldots, p_k\}$) the task is to find a sequence s of maximum length that fulfills the following two conditions: (1) s is a subsequence of each string $s_i \in S$, and (2) each pattern $p_j \in P$ is a subsequence of s. Any sequence that fulfills these two conditions is a *feasible solution*.

Related Work. The basic LCS problem has been tackled both by a wide range of exact and approximate approaches. For a fixed number of m input strings, the LCS problem is polynomially solvable by dynamic programming (DP) [15] which runs in $O(n^m)$ time, where n is the length of the longest input string. However, with increasing n and/or m, the application of DP quickly turns unpractical. In addition to DP, various parallel exact approaches were proposed. An example is the one from [19] called FAST_LCS. This algorithm is based on the use of a so-called successors table data structure, utilizing the pruning operations to reduce the overall computational effort. Quick-DP was introduced in [25] based on a fast divide-and-conquer technique. More recently, the Top_MLCS algorithm was proposed in [18], based on a directed acyclic layered-graph model. The most recent exact approach is A* [11]. This approach outperforms the other algorithms in terms of memory consumption, running time, and the number of benchmark instances solved to optimality. However, exact approaches are still quite limited and can only solve small LCS instances (up to $m = 10$ and $n = 100$), and their main bottleneck is an excessive memory consumption. Concerning larger LCS problem instances, research has mostly been focused on heuristic approaches. Among various different metaheuristic approaches, the generalized beam search approach from [9] is currently the state-of-the-art heuristic approach.

The CLCS problem with two input strings ($m = 2$) and one pattern string ($k = 1$) was well studied over the last two decades. Several efficient exact approaches were proposed, including dynamic programming, sparse dynamic programming, and a bit-parallel algorithm [4,5,24]. As in the case of the LCS problem, the current state-of-the-art approach for the CLCS problem is A* [7]. In particular, A* has a run time which is about one order of magnitude lower than the one of the second-best approach. In [6], this A* search was adapted to the more general CLCS variant with an arbitrary number of $m \in \mathbb{N}$ input strings. The algorithm was shown to scale well from small to medium sized instances and, in some cases of a long pattern string, even to larger instances.

Finally, concerning the GCLCS problem, it is known that no approximation algorithm can exist [14]. Moreover, the GCLCS problem with $m = 2$ input strings and an arbitrary number of pattern strings is $\mathcal{NP}$-hard, which can be proven by reduction from the 3-SAT problem. This implies that the GCLCS problem, as a generalization of the latter variant of the CLCS problem, is also $\mathcal{NP}$-hard. In [8] it was even proved that finding any feasible solution to the GCLCS problem is $\mathcal{NP}$–complete. In the same paper, the authors proposed various heuristic approaches, such as a greedy search, a beam search and a hybrid of variable neighbourhood search and beam search. The efficiency of these approaches was studied along the following two lines: finding a feasible solution and finding high-quality solutions. However, concerning exact algorithms, the related literature only offers the Automaton approach from [12] which runs in $O(|\Sigma|(\mathcal{R} + m) + nm + |\Sigma|\mathcal{R}n^k)$ time complexity, where $\mathcal{R}$ is the size of the resulting subsequence automaton which, in the worst case scenario, is $O(n^m)$.

Our Contributions. Due to the success of A* for related problems and due to the lack of exact approaches for the general GCLCS problem, we develop an A* search approach that employs a problem-specific node filtering technique as one of its main features. Note that our A* differs from the A* approach for the CLCS problem with only one pattern string in several aspects: (1) the search is based on a different search framework, which implies the utilization of different data structures in order to obtain an efficient search process, and (2) it employs a problem-specific node filtering technique. The quality of our approach is analysed on a wide range of benchmark instances, in comparison to the Automaton approach from the literature.

We emphasise that A* search and the Automaton approach are built upon different methodologies. The Automaton approach is fully constructive. It builts numerous automata along the way: common subsequence automaton, an intersection automaton, and finally a maximum length automaton, from which the optimal solution can be derived. On the other hand, our A* search is an informed search that generates its nodes on the fly. It is using a heuristic rule to expand the most promising nodes at each iteration, with the hope of reaching a complete node as soon as possible. Note that in contrast to the Automaton approach, which provides useful information only when the maximum length automaton is generated, our A* search method is able to provide a dual bound on the optimal solution at each iteration.

The rest of the paper is organized as follows. In Sect. 2 we describe the state graph for the GCLCS problem. Section 3 is reserved to present our A* search approach. The experimental evaluation is provided in Sect. 4, while Sect. 5 offers conclusions and outlines future work.

2 The State Graph for GCLCS Problem

For a string s and $1 \leq x \leq y \leq |s|$, let $s[x,y] = s[x] \cdots s[y]$ be the contiguous (sub)string which starts at index x and ends at index y. If $x > y$, $s[x,y]$ is the empty string ε. By convention, the first index of a string s is always 1. By $|s|_{\mathtt{a}}$ we denote the number of occurrences of letter $\mathtt{a} \in \Sigma$ in string s. Given a position vector $\vec{\theta}$, $S[\vec{\theta}] := \{s_i[\theta_i, |s_i|] \mid i = 1, \ldots, m\}$ is the set of suffix input strings starting from the positions of $\vec{\theta}$. Similarly, given $\vec{\lambda} = (\lambda_1, \ldots, \lambda_k)$, $1 \leq \lambda_j \leq |p_j| + 1$ for $j = 1, \ldots, k$, denotes a *position vector* concerning the set of pattern strings P, and $P[\vec{\lambda}] := \{p_j[\lambda_j, |p_j|] \mid j = 1, \ldots, k\}$ is the set of pattern suffix strings starting at the positions of $\vec{\lambda}$.

Preprocessing Data Structures. We make extensive use of the following data structures constructed during preprocessing in order to establish an efficient search. For each $i = 1, \ldots, m$, $l = 1, \ldots, |s_i|$, and $c \in \Sigma$, $\mathtt{Succ}[i, x, \mathtt{c}]$ stores the minimal index y such that (1) $x \geq y$ and (2) $s_i[y] = \mathtt{c}$, that is, the position of the next occurrence of letter $\mathtt{c}$ in string s_i starting from position x. If no such letter $\mathtt{c}$ exists in s_i, then $\mathtt{Succ}[i, x, \mathtt{c}] := -1$. This data structure can be built in $O(m \cdot n \cdot |\Sigma|)$ time. Further, table $\mathtt{Embed}[i, x, j]$ for all $i = 1, \ldots, m$, $j = 1, \ldots, k$, and $x = 1, \ldots, |p_j| + 1$ stores the right-most (highest) position y of s_i such that $p_j[x, |p_j|]$ is a subsequence of $s_i[y, |s_i|]$. Note that when $x = |p_j| + 1$ it follows that $p_j[x, |p_j|] = \varepsilon$. In this case $\mathtt{Embed}[i, x, j]$ is set to $|s_i| + 1$.

The State Graph. In this section we describe the state graph for the GCLCS problem in the form of a rooted, directed, acyclic graph. In particular, the state graph consists of nodes $v = (\vec{\theta^v}, \vec{\lambda^v}, l^v)$, where

- $\vec{\theta^v}$ is a position vector regarding the input strings;
- $\vec{\lambda^v}$ is a position vector regarding the pattern strings;
- l^v is the length of a partial solution that induces node v as explained in the following.

We say that a partial solution s_v induces a node $v = (\vec{\theta^v}, \vec{\lambda^v}, l^v)$ as follows.

- Position vector $\vec{\theta^v}$ is defined such that $s_i[1, \theta^v_i - 1]$ is the shortest possible prefix string of s_i of which s_v is a subsequence, for all $i = 1, \ldots, m$.
- Position vector $\vec{\lambda^v}$ is defined such that $p_j[1, \lambda^v_j - 1]$ is the longest possible prefix string of p_j which is a subsequence of s_v, for all $j = 1, \ldots, k$.
- $l^v := |s_v|$

Note that such a node v may represent several different partial solutions. The root node of the state graph is $r = ((1, \ldots, 1), (1, \ldots, 1), 0)$, induced by the empty partial solution ε, where $0 = |\varepsilon|$. By Σ^{nd}_v we denote the set of *non-dominated letters* that can be used to extend any of the partial solutions represented by a node v. The term *extending a partial solution* refers hereby to appending a (suitable) letter to the end of the respective partial solution that induces node v. The way of deriving this set of letters (Σ^{nd}_v) is described as follows. A letter $\mathtt{c}$ belongs to the set $\Sigma_v \supseteq \Sigma^{\mathrm{nd}}_v$ iff the following two conditions are fulfilled:

1. Letter $\mathtt{c}$ appears in each of the suffix strings $s_i[\theta_i^v, |s_i|]$, $i = 1, \ldots, m$.
2. Let $I_{cov} := \{j \in \{1, \ldots, k\} \mid \lambda_j^v \leq |p_j| \wedge p_j[\lambda_j^v] = \mathtt{c}\}$ and $I_{ncov} := \{1, \ldots, k\} \setminus I_{cov}$. For all $i = 1, \ldots, m$, the following must hold:
   - For all $j \in I_{cov}$ it holds that $\theta_i + \mathtt{Succ}[i, \theta_i^v, \mathtt{c}] + 1 \leq \mathtt{Embed}[i, \lambda_j^v + 1, \mathtt{c}]$;
   - For all $j \in I_{ncov}$ it holds that $\theta_i^v + \mathtt{Succ}[i, \theta_i^v, \mathtt{c}] + 1 \leq \mathtt{Embed}[i, \lambda_j^v, \mathtt{c}]$.

Note that Condition 1 may be checked in $O(m)$ time, whereas Condition 2 may be checked in $O(km)$ time. The set Σ_v is further reduced by removing dominated letters. We say that letter $\mathtt{a} \in \Sigma$ *dominates* letter $\mathtt{b} \in \Sigma$ iff $Succ[i, \theta_i^v, \mathtt{a}] \leq Succ[i, \theta_i^v, \mathtt{b}]$, for all $i = 1, \ldots, m$. By removing dominated letters from Σ_v, we finally obtain set Σ_v^{nd}. Now, for each letter $\mathtt{c} \in \Sigma_v^{\mathrm{nd}}$, a successor node w of v is generated in the following way.

- $\theta_i^w \leftarrow \mathtt{Succ}[i, \theta_i^v, \mathtt{c}] + 1$, for all $i = 1, \ldots, m$;
- If $p_j[\lambda_j^v] = \mathtt{c}$ then $\lambda_j^w \leftarrow \lambda_j^v + 1$; $\lambda_j^w \leftarrow \lambda_j^v$ otherwise;
- $l^w \leftarrow l^v + 1$.

Moreover, a directed arc vw is added from node v to node w and is labeled with letter $\mathtt{c}$, that is, $\ell(vv') = \mathtt{c}$.

We call a node v *non-extensible* if $\Sigma_v^{\mathrm{nd}} = \emptyset$. Moreover, it is *feasible* iff $\lambda_j^v = |p_i| + 1$, for all $j = 1, \ldots, k$, which means that the respective solution contains all pattern strings as subsequences. A longest path (in terms of the number of arcs) from the root node to a non-extensible and feasible node represents an optimal solution to respective problem instance. An example of the state graph of an GCLCS problem instance is shown in Fig. 1. In general, it is infeasible to produce the whole state graph before running an algorithm. Instead, the state graph is discovered step-by-step and searched on the fly. In the next section, based on the above state graph description, we develop an A* search algorithm for the GCLCS problem.

3 A* Search for the GCLCS Problem

We make use of the previously explained state graph for defining an A* search approach. A* was introduced as a general concept by Hart et al. [16]. Since then it has been successfully applied to numerous hard optimization problems that can be phrased in terms of finding a best path in a graph. A* search is an informed search as it utilizes a heuristic as guidance. It works in a best-first-search manner by always expanding a most promising not-yet-expanded node. In order to evaluate nodes v we make use of a function $f(v) = g(v) + h(v)$ where (1) $g(v)$ represents the length of the longest path from the root node to node v; and (2) $h(v)$ represents an estimation of the length of the longest path from v to a *goal node*, a node that is feasible and non-extensible. Note that $h()$ is a dual bound. In this work we set $h() = \mathrm{UB}()$, where UB() calculates the tightest LCS upper bound known from literature; see [2,9,25] for details.

In the following, we introduce the data structures necessary to make A* an efficient graph search approach in the context of GCLCS problem. In essence, A* search maintains two sets of nodes:

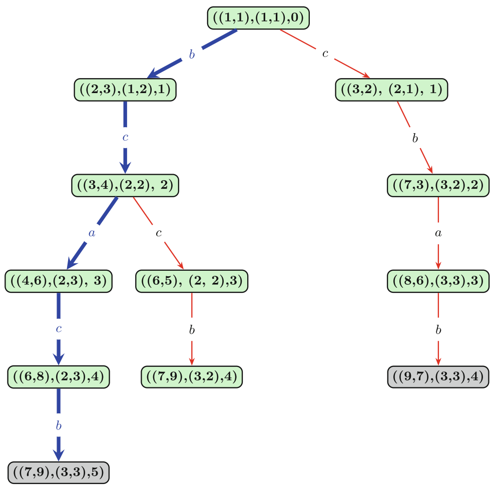

Fig. 1. The full state graph for GCLCS instance ($S = \{s_1 = \texttt{bcaacbad}, s_2 = \texttt{cbccadcb}\}$, $P = \{\texttt{cb}, \texttt{ba}\}$, $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$). It contains three non-extensible solutions. Two of them (marked by light-gray color) correspond to a feasible solution. Solution $s = \texttt{bcacb}$ is optimal. It is represented by node $v = (\vec{\theta}^v = (7, 9),\ \vec{\lambda}^v = (3, 3),\ l^v = 5)$. The corresponding path is displayed in blue. (Color figure online)

- The set of all nodes encountered during the search is realized by means of a hash map N. Each key $(\vec{\theta}^v, \vec{\lambda}^v)$ of N is mapped to the length of the currently longest path found from the root node to node v. The two position vectors together define a node v as previously described. Note again that there may exist more than one such path to the same node and, thus, the same node may represent several (partial) solutions, which helps saving memory and computational effort.
- The set of discovered, but not-yet-expanded (open) nodes $Q \subseteq N$. This set is realized by means of a priority queue whose nodes are prioritized according to their f-values; nodes with higher f-values are preferred.

With the help of these two sets, the required queries can be efficiently resolved. For example, the question if there was already a node encountered during the search defined by position vectors $\vec{\theta}$ and $\vec{\lambda}$ can be determined in constant time. Moreover, the node with the highest priority—that is, the one with the highest f-value—can be retrieved from Q in constant time. Note that after such a node-retrieval, Q must be reorganized. This process can efficiently be done in $O(\log(|Q|))$ time.

The pseudo-code of our A* search is given in Algorithm 1. Data structures N and Q are initialized with the root node r. At each iteration, the algorithm retrieves a highest-priority node v from queue Q. Then, it is first checked if v corresponds to a feasible, non-extensible solution. If so, the search is terminated since a proven optimal solution has been found. Otherwise, the upper bound UB(v) of v is calculated and—in case UB$(v) \geq l^v + \max\{|p_i| - \lambda_i^v + 1 \mid i =$

Algorithm 1. A* search for the GCLCS problem.

```
1: Data structures: $N$, the hash map containing the generated combinations of position vectors $\vec{\theta}$ and $\vec{\lambda}$; $N[\vec{\theta}, \vec{\lambda}]$ holds the length of the currently longest path for this combination; $Q$, priority queue with all open nodes
2: Create root node $r = ((1, \ldots, 1), (1, \ldots, 1), 0)$
3: $\vec{\theta}^r \leftarrow (1, \ldots, 1)$ (length $m$), $\vec{\lambda}^r \leftarrow (1, \ldots, 1)$ (length $k$)
4: $N[(\vec{\theta}^r, \vec{\lambda}^r] \leftarrow 0$
5: $Q \leftarrow \{r\}$
6: while $time \wedge memory\ limit$ are not exceeded $\wedge\ Q \neq \emptyset$ do
7:     $v \leftarrow Q.pop()$
8:     if $\mathrm{UB}(v) \geq l^v + \max\{|P_i| - \lambda_i^v + 1 \mid i = 1, \ldots, k\}$ then
9:         Determine $\Sigma_v^{\mathrm{nd}}$ (non-dominated letters)
10:        if $\Sigma_v^{\mathrm{nd}} = \emptyset$ then
11:            Derive the solution $s$ represented by $v$
12:            if $s$ is a feasible non-extensible solution then return provenly optimal solution $s$ end if
13:        else
14:            for $c \in \Sigma_v^{\mathrm{nd}}$ do
15:                Generate node $v'$ from node $v$ via extension by letter $c$
16:                if $(\vec{\theta}^{v'}, \vec{\lambda}^{v'}) \in N$ then
17:                    if $N[(\vec{\theta}^{v'}, \vec{\lambda}^{v'})] < l^{v'}$ then      // a better path found
18:                        $N[(\vec{\theta}^{v'}, \vec{\lambda}^{v'})] \leftarrow l^{v'}$
19:                        Update priority value of node $v$ in $Q$
20:                    end if
21:                else      // a new node
22:                    $f_{v'} \leftarrow l^{v'} + \mathrm{UB}(v')$
23:                    Insert $v'$ into $Q$ with priority value $f_{v'}$
24:                    Insert $v'$ into $N$
25:                end if
26:            end for
27:        end if
28:    end if
29: end while
30: return empty solution $\varepsilon$
```

$1, \ldots, k\}$—node v is expanded with all possible letters from Σ_v^{nd} as explained in Sect. 2. Further, for each generated child node v' of node v, it is checked if N already contains a corresponding key $(\vec{\theta}^{v'}, \vec{\lambda}^{v'})$. If not, v' is added to N and Q. Otherwise, it is checked if a longer path than the currently known one from the root node to node v' was discovered. If this is the case, the value of the corresponding key in N is updated accordingly, node v' is added in Q and the outdated entry is removed from Q. The above steps are repeated until either (1) an optimal solution is found as described above, or (2) Q becomes empty or (3) the memory or time limit is exceeded.

From a theoretical perspective, A* possesses some important characteristics. First, function h is *consistent*, that is, it never underestimates the length of the longest path from any node to a goal node (i.e., the optimum of the corresponding subproblem). This implies that a provenly optimal solution is found when the node retrieved from Q is a goal node. Second, the utilized upper bound UB is *monotonic*, which means that the difference in the values for any pair of parent/child nodes is never smaller that the weight of the respective arc, i.e., one in our case. As a consequence, no re-expansions of already expanded nodes are required during the search. Thus, a minimum number of node expansions is performed in order to find an optimal solution, concerning all search approaches that rely on the same state graph and the same heuristic guidance.

A^* search time complexity. The number of visited nodes is bounded by $O(n^{m+k})$. Concerning the time complexity of lines 7–28 of Algorithm 1, operation *pop*() is executed in constant time. Subsequently, a necessary rearrangement of p.q. Q is executed in $O(\log(|Q|)) = O(\log(n^{m+k})) = O((m+k)\log(n))$ time. Line 9, for generating set Σ_v^{nd}, takes $O(|\Sigma|(m+km)) + O(|\Sigma|^2 m)$ time. Lines 14–26 are executed in $O(|\Sigma|(m+k+\log(|Q|))) = O(|\Sigma|(m+k+(m+k)\log(n)))$ time. Thus, the overall time complexity of A* search is equal to $O(n^{m+k}((|\Sigma|+1)(m+k)\log(n) + |\Sigma|((|\Sigma|+k+2)m+k)))$.

4 Experimental Evaluation

For the experimental evaluation we consider, apart from our A* approach, the Automaton approach from [12]. A* was implemented in C++ using GCC 7.4 for compilation, and the experiments were conducted on a machine with an Intel Xeon E5-2640 processor with 2.40 GHz and a memory limit of 16 GB. Note that the implementation of the Automaton approach was adapted to use the same amount of memory (16 Gb) as A*. Moreover, both algorithms were tested on the same machine. The maximal CPU time allowed for each run of both algorithms was set to 180 min.

Benchmark Instances. The following three benchmark sets were used. Benchmark Random contains ten randomly generated instances per each combination of the following parameters: $m \in \{2, 5, 10\}$, $k \in \{2, 5, 10\}$, $p \in \{20, 50\}$, $|\Sigma| \in \{2, 20\}$ and $n = 100$. In total, Random consists of 360 problem instances. As a reminder, m refers to the number of input strings, k to the number of pattern strings, and $|\Sigma|$ to the alphabet size. Moreover, p refers to the fraction between n and the length of the pattern strings (all pattern strings are of equal length $n = 100$). The second benchmark set Real is composed of 12,681 bacterial 16S rRNA gene sequences. The whole set is divided into 49 classes (i.e., instances), where each class contains the sequences from one bacterial phylum. More detailed information about each class can be found in [8]. The third benchmark set (called Farhana-real) was used for the evaluation of Automaton in [12]. It consists of real-world instances generated on the basis of the NCBI database (see Table 2). This set contains 32 instances subdivided into the following four groups: `Rnase`, `Protease`, `Kinase`, and `Globin`. This division is made on the basis of a different set of pattern strings for each group. In particular,

Table 1. Results for instances from benchmark set Random

(a) Instances with $|\Sigma| = 2$

Instance group			A*			Automaton		
m	p	k	$\overline{\lvert s\rvert}$	$\overline{t}[s]$	$opt[\%]$	$\overline{\lvert s\rvert}$	$\overline{t}[s]$	$opt[\%]$
10	20	10	51.2	133.29	80	0.0	–	0
10	20	2	0.0	–	0	0.0	–	0
10	20	5	11.7	72.66	20	0.0	–	0
10	50	10	11.8	28.35	20	0.0	–	0
10	50	2	0.0	–	0	0.0	–	0
10	50	5	11.9	67.80	20	0.0	–	0
2	20	10	82.3	0.02	100	82.3	12.01	100
2	20	2	78.2	0.06	100	78.2	2.74	100
2	20	5	79.3	0.06	100	79.3	6.21	100
2	50	10	78.5	0.05	100	78.5	11.88	100
2	50	2	77.6	0.07	100	77.6	2.82	100
2	50	5	78.4	0.06	100	78.4	6.25	100
5	20	10	69.3	54.64	100	0.0	–	0
5	20	2	64.0	39.51	100	0.0	–	0
5	20	5	65.5	47.67	100	0.0	–	0
5	50	10	58.5	77.23	90	0.0	–	0
5	50	2	64.6	37.38	100	0.0	–	0
5	50	5	57.4	137.19	90	0.0	–	0

(b) Instances with $|\Sigma| = 20$

Instance group			A*			Automaton		
m	p	k	$\overline{\lvert s\rvert}$	$\overline{t}[s]$	$opt[\%]$	$\overline{\lvert s\rvert}$	$\overline{t}[s]$	$opt[\%]$
10	20	10	50.0	0.03	100	0.0	–	0
10	20	2	10.6	0.16	100	0.0	–	0
10	20	5	25.0	0.05	100	0.0	–	0
10	50	10	20.0	0.13	100	0.0	–	0
10	50	2	7.7	0.10	100	0.0	–	0
10	50	5	10.6	0.15	100	0.0	–	0
2	20	10	53.1	0.01	100	5.2	141.37	10
2	20	2	31.8	0.60	100	31.8	3.09	100
2	20	5	36.4	5.05	100	36.4	8.18	100
2	50	10	35.2	0.25	100	35.2	21.14	100
2	50	2	32.7	0.06	100	32.7	3.06	100
2	50	5	33.2	0.07	100	33.2	7.28	100
5	20	10	50.0	0.02	100	4.1	53.87	10
5	20	2	13.7	2.13	100	13.7	6.12	100
5	20	5	25.1	0.21	100	25.1	11.37	100
5	50	10	20.4	2.81	100	20.4	20.56	100
5	50	2	12.7	1.24	100	12.7	6.11	100
5	50	5	13.4	2.50	100	13.4	10.64	100

in the case of `Rnase` the set of pattern strings is {`H`, `K`, `HKSH`, `HKSTH`}; for group `Protease` it is {`P`, `L`, `DGTG`, `IIGL`}; for the group `Kinase` it is {`G`, `KEL`, `DFG`, `PEDR`}; and for group `Globin` it is {`P`, `KF`, `HGLSLH`, `LVLA`}. Apart from these four groups, there is one additional problem instance called `Input100` with $m = 100$ input strings of different lengths (ranging from 41 to 100) and having only one pattern string of length one, which is `S`.

Results for Benchmark Set Random. Tables 1 and 2 report the results of the two competitors on Random and Farhana-real benchmarks, respectively. The first block of columns shows the name of the respective instance group, together with the number of instances in that group. The following three columns are reserved for the results of A* search. The first column provides the average solution quality delivered upon termination ($\overline{t}[s]$). The second one shows the average running time, for those instances/runs for which an optimal solution could be found ($\overline{t}$[s]). Finally, the third column indicates the percentage of instances for which an optimal solution was found ($opt[\%]$). The last three table columns report the results of Automaton in the following way: the average solution quality ($\overline{|s|}$), the average running time ($\overline{t}$[s]) and the percentage of instances solved to optimality ($opt[\%]$).

Concerning instances $|\Sigma| = 2$, Table 1a allows the following observations. A* was able to solve almost all instances (118 out of 120 problem instances) with $m \leq 5$ to optimality. In contrast, the Automaton approach was successful only for the instances with $m = 2$. Concerning the instances with $m = 10$, 16 out of 60 instances were solved by A* and none by the Automaton approach. Finally, concerning the computation times for those instances for which both

Table 2. Results for the real-world benchmark set FARHANA-REAL from [12].

Instance group	#inst	A*			AUTOMATON		
		$\overline{\|s\|}$	$\bar{t}$[s]	*opt*[%]	$\overline{\|s\|}$	$\bar{t}$[s]	*opt*[%]
`Rnase`	3	68.33	0.12	100	68.33	4.78	100
`Protease`	15	55.60	0.70	100	55.60	4.71	100
`Kinase`	3	111.00	0.10	100	111.00	13.40	100
`Globin`	10	84.10	0.11	100	84.10	7.80	100
`Input100`	1	2.00	0.06	100	2.00	48.38	100

approaches were successful in finding an optimal solution, A* search is the clear winner exhibiting computation times about two orders of magnitude lower than those of AUTOMATON (around 100 times faster). The following observations can be made from Table 1b (for the instances with $|\Sigma| = 20$). First, A* was able to solve all instances (180 out of 180 problem instances) to optimality. The AUTOMATON approach was competitive on the instances with $m \leq 5$ by solving 102 out of 120 instances, but none of the instances with $m = 10$ were solved. Second, the running times of A* (limited to those instances which both algorithm could solve) appear to be much shorter than those of AUTOMATON.

Results for Benchmark Set REAL. A* was able to solve four (out of 49) real-world problem instances to proven optimality. This is quite notable since similar problems for real-world instances are rarely solved to optimality in the literature. Moreover, in three out of four cases, A* requires only a fraction of a second to prove optimality. In particular, instance Aminicenantes (result: 1365) was solved in 11.51 s, instance Atribacteria (result: 1499) in 0.12 s, instance Ignavibacteriae (result: 1354) in 0.12 s, and instance WPS-1 (result: 1358) in 0.1 s. In contrast, AUTOMATON was not able to deliver any optimal solutions since it was—in all cases—running out of time. This is because AUTOMATON was not able to finish its intermediate step of constructing the intersection automaton within the given time limit. Finally, for the instances not solved by A*, the reason of not doing so, was memory limit exceeding.

Results for Benchmark Set FARHANA-REAL*; see Table* 2. The following observations can be made. First, both algorithms were able to find optimal solutions for all 32 instances. Second, in comparison to AUTOMATON, A* required substantially less time; about an order of magnitude less. Finally, for the largest instance (*Input100*), the runtime is more than 500 times in favor of A*.

5 Conclusions and Future Work

In this paper we presented an A* approach for solving the generalized longest common subsequence problem with an arbitrary number of pattern strings. Our algorithm utilizes a problem-specific node filtering technique in order to exclude suboptimal solutions from the search. The experimental evaluation shows that A* is efficient in solving all instances with two input strings based on rather small alphabet sizes (up to four) to optimality. Moreover, A* search is also well-suited for instances with shorter input strings (up to $n = 100$), even when there is a larger number of patterns given as input. In comparison to the exact AUTOMATON approach from the literature, it turns out that A* can find proven optimal solutions in an order of magnitude faster than AUTOMATON when applied to real-world instances.

In the future work it seems promising to focus on developing tight upper bounds for the GCLCS problem and utilizing them in our A* approach. Concerning anytime algorithms based on A* [10], it would be interesting to obtain heuristic solutions in combination with dual bounds for large-sized instances (when classical A* search fails to prove optimality due to time or memory restrictions). Moreover, studying problems related to the GCLCS problem, such as the restricted LCS problem [13], might be a promising research direction.

Acknowledgements. Christian Blum was funded by project CI-SUSTAIN of the Spanish Ministry of Science and Innovation (PID2019-104156GB-I00). Dragan Matić is partially supported by Ministry for Scientific and Technological Development, Higher Education and Information Society, Government of Republic of Srpska, B&H under the Project “Development of artificial intelligence methods for solving computer biology problems”.

References

1. Adi, S.S.: Repetition-free longest common subsequence. Discr. Appl. Math. **158**(12), 1315–1324 (2010)
2. Blum, C., Blesa, M.J., López-Ibáñez, M.: Beam search for the longest common subsequence problem. Comput. Oper. Res. **36**(12), 3178–3186 (2009)
3. Chowdhury, S.R., Hasan, M., Iqbal, S., Rahman, M.S.: Computing a longest common palindromic subsequence. Fund. Inform. **129**(4), 329–340 (2014)
4. Deorowicz, S.: Bit-parallel algorithm for the constrained longest common subsequence problem. Fund. Inform. **99**(4), 409–433 (2010)
5. Deorowicz, S., Obstój, J.: Constrained longest common subsequence computing algorithms in practice. Comput. Inf. **29**(3), 427–445 (2012)
6. Djukanovic, M., Berger, C., Raidl, G.R., Blum, C.: On solving a generalized constrained longest common subsequence problem. In: Olenev, N., Evtushenko, Y., Khachay, M., Malkova, V. (eds.) OPTIMA 2020. LNCS, vol. 12422, pp. 55–70. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-62867-3_5
7. Djukanovic, M., Berger, C., Raidl, G.R., Blum, C.: An A* search algorithm for the constrained longest common subsequence problem. Inf. Process. Lett. **166**, 106041 (2021)
8. Djukanovic, M., Kartelj, A., Matic, D., Grbic, M., Blum, C., Raidl, G.: Graph search and variable neighborhood search for finding constrained longest common subsequences in artificial and real gene sequences. Technical report AC-TR-21-008 (2021)

9. 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
10. Djukanovic, M., Raidl, G.R., Blum, C.: Anytime algorithms for the longest common palindromic subsequence problem. Comput. Oper. Res. **114**, 104827 (2020)
11. Djukanovic, M., Raidl, G.R., Blum, C.: Finding longest common subsequences: new anytime A* search results. Appl. Soft Comput. **95**, 106499 (2020)
12. Farhana, E., Rahman, M.S.: Constrained sequence analysis algorithms in computational biology. Inf. Sci. **295**, 247–257 (2015)
13. Gotthilf, Z., Hermelin, D., Landau, G.M., Lewenstein, M.: Restricted LCS. In: Chavez, E., Lonardi, S. (eds.) SPIRE 2010. LNCS, vol. 6393, pp. 250–257. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16321-0_26
14. 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
15. Gusfield, D.: Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press (1997)
16. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Syst. Sci. Cybern. **4**(2), 100–107 (1968)
17. Jiang, T., Lin, G., Ma, B., Zhang, K.: The longest common subsequence problem for arc-annotated sequences. J. Discrete Algorithms **2**(2), 257–270 (2004)
18. Li, Y., Wang, Y., Zhang, Z., Wang, Y., Ma, D., Huang, J.: A novel fast and memory efficient parallel MLCS algorithm for long and large-scale sequences alignments. In: Proceedings of the 32nd International Conference on Data Engineering, ICDE 2019, pp. 1170–1181 (2016)
19. Liu, W., Chen, L.: A fast longest common subsequence algorithm for biosequences alignment. In: Li, D. (ed.) CCTA 2007. TIFIP, vol. 258, pp. 61–69. Springer, Boston (2008). https://doi.org/10.1007/978-0-387-77251-6_8
20. Maier, D.: The complexity of some problems on subsequences and supersequences. J. ACM **25**(2), 322–336 (1978)
21. Martínez-Porchas, M., Vargas-Albores, F.: An efficient strategy using k-mers to analyse 16s rRNA sequences. Heliyon **3**(7), e00370 (2017)
22. Mount, D.W.: Bioinformatics: Sequence and Genome Analysis, 2nd edn. Cold Spring Harbour Laboratory Press, Cold Spring Harbour (2004)
23. Tang, C.Y.: Constrained multiple sequence alignment tool development and its application to RNase family alignment. J. Bioinf. Comput. Biol. **01**(02), 267–287 (2003)
24. Tsai, Y.-T.: The constrained longest common subsequence problem. Inf. Process. Lett. **88**(4), 173–176 (2003)
25. Wang, Q., Korkin, D., Shang, Y.: A fast multiple longest common subsequence (MLCS) algorithm. IEEE Trans. Knowl. Data Eng. **23**(3), 321–334 (2011)