# Exact and Heuristic Approaches for the Longest Common Palindromic Subsequence Problem

Marko Djukanovic[1], Günther R. Raidl[1], and Christian Blum[2]

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

**Abstract.** The longest common palindromic subsequence (LCPS) problem requires to find a longest palindromic string that appears as subsequence in each string from a given set of input strings. The algorithms that can be found in the related literature are specific for LCPS problems with only two input strings. In contrast, in this work we consider the general case with an arbitrary number of input strings, which is NP-hard. To solve this problem we propose a fast greedy heuristic, a beam search, and an exact A$^*$ algorithm. Moreover, A$^*$ is extended by a simple diving mechanism as well as a combination with beam search in order to find good quality solutions already early in the search process. The most important findings that result from the experimental evaluation include that (1) A$^*$ is able to efficiently find proven optimal solutions for smaller problem instances, (2) the anytime behavior of A$^*$ can be significantly improved by incorporating diving or beam search, and (3) beam search is best from a purely heuristic perspective.

**Keywords:** Longest common palindromic subsequence problem; A$^*$ search; Beam search; hybrid optimization techniques

## 1 Introduction

In computer science, a *string* $s$ is defined as a finite sequence of characters from a (generally finite) alphabet $\Sigma$. An important characteristic of a string $s$ is its length, denoted by $|s|$. A string is generally used as a data type for representing and storing sequence information. Words, and even whole texts, may be stored by means of strings. Strings arise, in particular, in the field of bioinformatics, because most of the genetic instructions involved in the growth, development, functioning and reproduction of living organisms are stored in *Deoxyribonucleic acid* (DNA) molecules which can be represented by strings over the alphabet $\Sigma = \{A, C, T, G\}$. A string $s$ is called a *palindrome* if $s = s^{\text{rev}}$, where $s^{\text{rev}}$ is the reverse string of $s$; for example, `madam` is a palindrome.

Note that, given a string $s$, any string $t$ that can be obtained from $s$ by deleting zero or more characters is called a *subsequence* of $s$. Palindromic subsequences are especially interesting in the biological context. In many genetic

instructions, such as for example DNA sequences, palindromic motifs are found. In the context of a research project on genome sequencing it was discovered that many of the bases on the Y-chromosome are arranged as palindromes [17]. A palindrome structure allows the Y-chromosome to repair itself by bending over at the middle if one side is damaged. Moreover, it is believed that palindromes are also frequently found in proteins [10], but their role in the protein function is less understood. Biologists believe that identifying palindromic subsequences of DNA sequences may help to understand genomic instability [7,23]. Palindromic subsequences seem to be important for the regulation, for example, of gene activity, because they are often found close to promoters, introns and untranslated regions.

An important way for the comparison of two or more strings is to find long *common subsequences*. More specifically, given a set of $m$ non-empty strings $S = \{s_1, \ldots, s_m\}$, a common subsequence of the strings in $S$ is a subsequence that all strings in $S$ have in common. Moreover, a *longest common subsequence* of the strings in $S$ is a common subsequence of maximal length. The so-called *Longest Common Subsequence (LCS)* problem [18] is a classical optimization problem that aims at finding such a longest common subsequence of the strings in $S$. Apart from applications in computational biology [16], this problem appears, for example, in data compression [22] and the production of circuits in field programmable gate arrays [6]. Finally, a *common palindromic subsequence* of a set of strings $S$ is a common subsequence of all strings in $S$ which, at the same time, is a palindrome. For biologists it is not only of interest to identify the palindromic subsequences of an individual DNA string, for example, but it is also important to find longest common palindromic subsequences of multiple input strings in order to identify relationships among them.

### 1.1 Related Work

The LCS problem is known to be NP-hard for an arbitrary number ($m$) of input strings [18]. Note, however, that for any fixed $m$ the problem is polynomially solvable by dynamic programming [11]. Standard dynamic programming approaches require $O(n^m)$ time and space, where $n$ is the length of the longest input string. Even though this complexity can be reduced to $O(n^{m-1})$, see Bergoth et al. [1], dynamic programming becomes quickly impractical when $m$ grows. Concerning simple approximate methods, the expansion algorithm in [5] and the Best-Next heuristic [9,14] are probably the best-known techniques. A break-through both in terms of computation time and solution quality was achieved with the beam search (BS) approach described by Blum et al. [2]. Beam search is an incomplete tree search algorithm which relies on a solution construction mechanism and bounding information. More specifically, the above BS uses the construction mechanism of the Best-Next heuristic and just a simple upper bound function. Nevertheless, this algorithm was able to outperform all existing algorithms at that time. Later, Mousavi and Tabataba [20] proposed a variant of this BS with a different heuristic function and a different pruning mechanism.

The specific problem tackled in this work—that is, the longest common palindromic subsequence (LCPS) problem—has so far only been studied for the case of $m = 2$ input strings (2-LCPS). Chowdhury et al. [8] propose two different algorithms: a conventional dynamic programming with time and space complexity $O(n^4)$, and a sparse dynamic programming algorithm with time complexity $O(R^2 \log^2 n \log\log n + n)$ and space complexity $O(R^2)$, where $R$ is the number of matching position pairs between the two input strings. Furthermore, Hasan et al. [13] solved the 2-LCPS by making use of a so-called palindromic subsequence automaton (PSA). This algorithm has a time complexity of $O(n + R_1|\Sigma| + R_2|\Sigma| + n + R_1R_2|\Sigma|)$, where $R_1$ and $R_2$ denote the numbers of states of the two automata constructed for the two input strings and are bounded by $O(n^2)$. Finally, Inenaga and Hyyrö [15] present an algorithm that runs in $O(\sigma R^2 + n)$ time and uses $O(R^2 + n)$ space, where $\sigma$ denotes the number of distinct characters occurring in both of the input strings.

By reducing the general LCS problem to the LCPS problem in polynomial time, it can be shown that the LCPS problem with an arbitrary number of input strings is NP-hard. To the best of our knowledge, no algorithm has been published yet for solving this general $m$-LCPS problem, which is henceforth simply called LCPS problem. An instance of the LCPS problem is denoted by $(S, \Sigma)$, where $S$ is the set of input strings over alphabet $\Sigma$.

### 1.2 Organization of the Paper

The rest of the paper is organized as follows. In Section 2, a fast greedy heuristic is presented. Moreover, two upper bound functions are proposed. In Section 3 we present two search algorithms that operate on the same search tree: an exact A$^*$ search and a heuristic beam search. We further consider a variant of A$^*$ that has a simple diving mechanism as well as beam search embedded in order to obtain promising complete solutions already early during the search process. In this way, both A$^*$ and beam search can be used as heuristics to approach large instances. Experimental results are described in Section 4. Conclusions as well as an outlook to future work are finally provided in Section 5.

## 2 A Greedy Heuristic for the LCPS Problem

We first introduce some additional notations. As already mentioned before, let $n = \max_{s_i \in S} |s_i|$ be the maximum input string length. The $j$-th letter of a string $s$ is stated by $s[j]$, with $j = 1, \ldots, |s|$. We further denote the concatenation of two strings by operator "$\cdot$", i.e., $s_1 \cdot s_2$ is the string obtained by appending string $s_2$ to string $s_1$. Notation $s[j, j']$, $j \leq j'$, refers to the substring of $s$ starting at the $j$-th position and ending at position $j'$. The same notation refers to the empty string $\varepsilon$ if $j > j'$. Finally, let $|s|_a$ be the number of occurrences of letter $a \in \Sigma$ in string $s$, and let $|s|_A = \sum_{a \in A} |s|_a$ be the total number of occurrences of letters from set $A \subseteq \Sigma$ in string $s$.

Inspired by the well-known Best-Next heuristic for the LCS problem [9], we present in the following a constructive greedy heuristic for the LCPS problem. Henceforth, a string $s$ is called a *valid partial solution* concerning input strings $S = \{s_1, \ldots, s_m\}$, if $s \cdot s^{\text{rev}}$ or $s \cdot s[1, |s|-1]^{\text{rev}}$ is a common palindromic subsequence of the strings in $S$. The greedy heuristic starts with an empty partial solution $s = \varepsilon$ and extends, at each construction step, the current partial solution by appending exactly one letter (if possible). During the whole process, the algorithm makes use of pointers $p_i^{\text{L}} \leq p_i^{\text{R}}$ that indicate for each input string $s_i$, $i = 1, \ldots, m$, the still *relevant substring* $s_i[p_i^{\text{L}}, p_i^{\text{R}}]$ from which the letter for extending $s$ can be chosen. The choice of a letter with respect to a greedy criterion is explained below. At the start of the heuristic, i.e., when $s = \varepsilon$, the pointers are initialized to $p_i^{\text{L}} := 1$ and $p_i^{\text{R}} := |s_i|$, referring to the first, respectively, last letter of each string $s_i$, $i = 1, \ldots, m$. In other words, at each iteration the set of relevant substrings denoted by $S[p^{\text{L}}, p^{\text{R}}] = \{s_i[p_i^{\text{L}}, p_i^{\text{R}}] \mid i = 1, \ldots, m\}$ forms an LCPS subproblem, and the current partial solution $s$ is ultimately extended by appending the solution to this subproblem.

One of the questions that remain is how to determine the subset of letters from $\Sigma$ that can be used to extend a current partial solution $s$. For this purpose, let $c_a := \min_{i=1,\ldots,m} |s_i[p_i^{\text{L}}, p_i^{\text{R}}]|_a$ be the minimum number of occurrences of letter $a \in \Sigma$ in the relevant substrings with respect to $s$, and let $\Sigma_{(p^{\text{L}},p^{\text{R}})} := \{a \in \Sigma \mid c_a \geq 1\}$ be the set of letters appearing at least once in each relevant substring. In principle, any letter from $\Sigma_{(p^{\text{L}},p^{\text{R}})}$ might be used to extend $s$. However, there might be dominated letters in this set. In order to introduce the domination relation between two letters, we use the first and last positions at which each letter $a \in \Sigma_{(p^{\text{L}},p^{\text{R}})}$ appears in each relevant substring $s_i[p_i^{\text{L}}, p_i^{\text{R}}]$:

$$q_{i,a}^{\text{L}} := \min\{j = p_i^{\text{L}}, \ldots, p_i^{\text{R}} \mid s_i[j] = a\}$$
$$q_{i,a}^{\text{R}} := \max\{j = p_i^{\text{L}}, \ldots, p_i^{\text{R}} \mid s_i[j] = a\}$$

A letter $a \in \Sigma_{(p^{\text{L}},p^{\text{R}})}$ is called *dominated* if there exists a letter $b \in \Sigma_{(p^{\text{L}},p^{\text{R}})}$, $b \neq a$, such that $q_{i,b}^{\text{L}} < q_{i,a}^{\text{L}} \wedge q_{i,b}^{\text{R}} > q_{i,a}^{\text{R}}$ for $i = 1, \ldots, m$. Clearly, it is better to delay the consideration of dominated letters and select a non-dominated letter for the extension of $s$. Furthermore, letters $a \in \Sigma_{(p^{\text{L}},p^{\text{R}})}$ with $c_a = 1$, called *singletons*, should only be considered when no other letters remain in $\Sigma_{(p^{\text{L}},p^{\text{R}})}$, since only one such letter can be chosen as single middle letter in the final solution. Accordingly, let the set of all non-dominated non-singleton letters from $\Sigma_{(p^{\text{L}},p^{\text{R}})}$ with respect to $s$ be denoted by $\Sigma_{(p^{\text{L}},p^{\text{R}})}^{\text{nd}}$. Given a partial solution $s$, the selection of the letter to be appended to $s$ and the adaption of the pointers work as follows:

1. If $\Sigma_{(p^{\text{L}},p^{\text{R}})}$ is empty, the algorithm terminates with $s \cdot s^{\text{rev}}$ as resulting common palindromic subsequence, since no further extension is possible.
2. Otherwise, if $\Sigma_{(p^{\text{L}},p^{\text{R}})}^{\text{nd}}$ is empty, only singletons remain in $\Sigma_{(p^{\text{L}},p^{\text{R}})}$. The algorithm terminates with the common palindromic subsequence $s \cdot a \cdot s^{\text{rev}}$, where $a$ is the first singleton from $\Sigma_{(p^{\text{L}},p^{\text{R}})}$ in alphabetic order.

3. Otherwise, select a letter $a \in \Sigma^{\mathrm{nd}}_{(p^{\mathrm{L}},p^{\mathrm{R}})}$ that minimizes the *greedy function* $g(a, p^{\mathrm{L}}, p^{\mathrm{R}})$, which will be discussed in Section 2.1. Ties are broken randomly. Extend the current partial solution $s$ and adapt the pointers as follows:

$$s := s \cdot a \tag{1}$$

$$p^{\mathrm{L}}_i := q^{\mathrm{L}}_{i,a} + 1 \qquad i = 1, \dots, m \tag{2}$$

$$p^{\mathrm{R}}_i := q^{\mathrm{R}}_{i,a} - 1 \qquad i = 1, \dots, m \tag{3}$$

### 2.1 Greedy Function

The greedy function that is used to evaluate any possible extension $a \in \Sigma^{\mathrm{nd}}_{(p^{\mathrm{L}},p^{\mathrm{R}})}$ for a given partial solution extends the one used in [3] in a straight-forward manner. It calculates the sum of those fractions of the relevant substrings $s_i[p^{\mathrm{L}}_i, p^{\mathrm{R}}_i]$ that will be discarded from further consideration when appending $a$ as next letter to the partial solution:

$$g(a, p^{\mathrm{L}}, p^{\mathrm{R}}) := \sum_{i=1}^{m} \frac{q^{\mathrm{L}}_{i,a} - p^{\mathrm{L}}_i + p^{\mathrm{R}}_i - q^{\mathrm{R}}_{i,a}}{p^{\mathrm{R}}_i - p^{\mathrm{L}}_i + 1}. \tag{4}$$

The major advantage of this function is its simplicity, as it can be calculated in time $O(m)$. Obviously, this function also has some weaknesses: (1) it does not take into account that, when choosing a specific letter, as a result, more or less letters might be excluded from further consideration, even in cases in which the chosen letter has a good (low) greedy function value; (2) it does not take into account that of all singletons at most one can finally be selected. Instead of improving the above greedy function along these lines, and thereby increasing its time complexity, we consider it more promising—especially with the type of algorithm in mind that will be presented in the next section—to develop upper bound functions for estimating the length of an LCPS. As we will see, these bounds can also be used as alternative greedy functions to evaluate possible extensions of partial solutions.

### 2.2 Upper Bounds for the Length of an LCPS

A first upper bound for the length of any palindromic subsequences obtainable for a set of strings $S$ can be calculated by

$$\mathrm{UB}_1(S) = \left( 2 \sum_{a \in \Sigma_{(p^{\mathrm{L}},p^{\mathrm{R}})}} \left\lfloor \frac{c_a}{2} \right\rfloor \right) + \mathbb{1}_{\exists a \in \Sigma_{(p^{\mathrm{L}},p^{\mathrm{R}})} | c_a \bmod 2 = 1}. \tag{5}$$

The last term considers the fact that at most one singleton letter can be added at the end of a solution construction, with $\mathbb{1}$ denoting the unit step function that yields one iff the condition in the subscript is fulfilled, i.e., there exists a letter in $\Sigma_{(p^{\mathrm{L}},p^{\mathrm{R}})}$ with an odd value of $c_a$. Note that $\mathrm{UB}_1(S)$ can be calculated in $O(mn)$ time, considering the required re-calculation of the counters $c_a$.

A second upper bound can be derived as follows. First, each relevant substring $s_i[p_i^{\mathrm{L}}, p_i^{\mathrm{R}}]$ is reduced by deleting all letters that are not in $\Sigma_{(p^{\mathrm{L}},p^{\mathrm{R}})}$. The resulting strings are denoted by $s_i[p_i^{\mathrm{L}}, p_i^{\mathrm{R}}]^{\mathrm{red}}$, $i = 1, \ldots, m$. Then, a longest palindromic subsequence, denoted by $\mathrm{LPS}(s_i[p_i^{\mathrm{L}}, p_i^{\mathrm{R}}]^{\mathrm{red}})$, is calculated for each of these strings individually. As a longest common palindromic subsequence of all strings $S[p^{\mathrm{L}}, p^{\mathrm{R}}]^{\mathrm{red}}$ cannot be longer than any individual longest palindromic subsequence, we obtain the upper bound

$$\mathrm{UB}_2(S) = \min_{i=1,\ldots,m} |\mathrm{LPS}(s_i[p_i^{\mathrm{L}}, p_i^{\mathrm{R}}]^{\mathrm{red}})|. \tag{6}$$

A longest palindromic subsequence of a single string can be calculated by solving the LCS (longest common subsequence) problem for the problem instance that has as input strings the string itself and its reversal. This can be done by dynamic programming in $O(n^2)$ time. Masek and Paterson [19] presented a more specific and slightly faster algorithm that runs in $O(n^2/\log n)$ time.

Note that none of the two upper bounds dominates the other one. For example, for $S = \{\mathtt{abba}, \mathtt{abab}\}$ we get $\mathrm{UB}_1(S) = 4$ and $\mathrm{UB}_2(S) = 3$, whereas for $S = \{\mathtt{aba}, \mathtt{bab}\}$ we get $\mathrm{UB}_1(S) = 1$ and $\mathrm{UB}_2(S) = 3$. Therefore, it might be beneficial to consider the minimum of both bounds $\mathrm{UB}_3(S) = \min(\mathrm{UB}_1(S), \mathrm{UB}_2(S))$.

Finally, observe that our upper bound functions can also directly be applied to evaluate any partial solution $s$ with its still relevant substrings $S[p^{\mathrm{L}}, p^{\mathrm{R}}]$: While $\mathrm{UB}_x(S[p^{\mathrm{L}}, p^{\mathrm{R}}])$, for $x \in \{1, 2, 3\}$, is an upper bound for the length by which $s$ may still be extended, $|s| + \mathrm{UB}_x(S[p^{\mathrm{L}}, p^{\mathrm{R}}])$ is an upper bound for the overall length of still achievable solutions. In the greedy heuristic, our upper bound functions can therefore be used instead of $g(a, p^{\mathrm{L}}, p^{\mathrm{R}})$ by temporarily determining the updated $p^{\mathrm{L}}$ and $p^{\mathrm{R}}$ when one would accept letter $a$ and calculating $\mathrm{UB}_x(S[p^{\mathrm{L}}, p^{\mathrm{R}}])$.

## 3 Search Algorithms for the LCPS

In the following we first describe the state graph on which both A* and Beam Search will operate. This state graph is a directed acyclic multi-graph $G = (V, A)$ in which each node (state) $v = (p^{\mathrm{L},v}, p^{\mathrm{R},v}) \in V$ corresponds to a unique LCPS subproblem, i.e., a set of still relevant substrings indicated by the respective pointer vectors $p^{\mathrm{L},v}$ and $p^{\mathrm{R},v}$. Note that one such node will in general represent multiple different partial solutions in an efficient way. As an example consider $S = (\mathtt{abccdccba}, \mathtt{baccdccab})$, and partial solutions $s = \mathtt{ac}$ and $s' = \mathtt{bc}$. It holds that $p^{\mathrm{L}} = (4, 4)$ and $p^{\mathrm{R}} = (6, 6)$ in both cases, and thus, both partial solutions will be represented by a common node. Here, $s$ and $s'$ have the same length, but this need not be the case in general. Our state graph has the root node $r = ((1, \ldots, 1), (|s_1|, \ldots |s_m|))$ corresponding to the original set of input strings $S$ and representing the empty partial solution. Each node $v \in V$ stores as additional information the length $l^v$ of a so far best—i.e., longest—partial solution represented by $v$. Furthermore, each node $v \in V$ has an outgoing arc $(v, v', a) \in A$ for each valid extension of the represented partial solutions by a non-dominated non-singleton letter $a \in \Sigma_v^{\mathrm{nd}} = \Sigma_{(p^{\mathrm{L},v}, p^{\mathrm{R},v})}^{\mathrm{nd}}$.

We emphasize that it is not necessary to store actual partial solutions $s$ in the nodes. As pointed out already in Section 2, this is neither necessary for the greedy function evaluation, nor for the upper bound calculation. For any node in the graph a corresponding solution string can finally be efficiently derived in a backward manner by iteratively identifying predecessors in which the $l^v$-values always decrease by one.

### 3.1 A* Search for the LCPS Problem

A* is a widely used algorithm belonging to the class of informed search methods for finding shortest or longest paths [12]. It maintains two sets of nodes: $N$ stores all so far reached nodes, while $Q$, the set of *open nodes*, is the subset of nodes in $N$ that have not yet been *expanded*, i.e., whose outgoing arcs and respective neighbors have not yet been considered. We realize node set $N$ by means of a hash map in order to be able to efficiently find an already existing node for a state $(p^{\mathrm{L}}, p^{\mathrm{R}})$, or to determine that no respective node exists yet. Moreover, $Q$ is a priority queue in which nodes are sorted according to decreasing *priority values* $\pi(v) = l^v + \mathrm{UB}_x(S[p^{\mathrm{L},v}, p^{\mathrm{R},v}])$, where $x$ specifies the used upper bound.

The pseudo-code of our A* search is shown in Algorithm 1. It starts with the root node as unique node in $N$ and $Q$. At each step, the first node $v$ from $Q$—that is, the highest priority node—is chosen and removed from $Q$. If this node is non-extensible, it is first checked if a singleton letter can be added, and afterwards the algorithm stops. Since our priority function is *admissible*, cf. [12], we can be sure that an optimal solution has been reached. Otherwise, node $v$ is extended by considering all possible extensions from $\Sigma_v^{\mathrm{nd}}$. For each obtained new state it is checked if a respective node exists already in $N$. If this is the case, the existing node's length-value is updated in case the new path to this node represents a new longest partial solution. Otherwise, a corresponding new node is created and added to $N$ and $Q$.

Finally, we remark that both upper bound functions presented in Section 2.2, i.e., $\mathrm{UB}_1$ and $\mathrm{UB}_2$, as well as their combination $\mathrm{UB}_3$, are *monotonic* (also called *consistent*) because the upper bound value of an extension of a node is always at most as high as the upper bound value of the originating node. Due to this property we can be sure that no re-expansions of already expanded nodes will be necessary, see again [12].

*Diving in A*.* One of the main advantages of A* is the fact that the search performs in an asymptotic optimal way with respect to the applied upper bound function, requiring the least possible number of node expansions in order to find a proven optimal solution. On the downside, good approximate solutions are typically only obtained, if at all, very late in the search. To improve this situation and turn our A* into an *anytime* algorithm, which can be terminated almost arbitrarily and still yields a reasonable solution, we augment it by switching in regular intervals to a temporary greedy depth-first search until no further extensible solution is obtained. We call this extension *diving.* More specifically, diving is initiated at the very beginning and after each $\delta$ regular A* iterations,

**Algorithm 1** A* Search for the LCPS problem

```
 1: Input: an instance (S, Σ)
 2: Output: s_bsf, an optimal LCPS solution
 3: s_bsf ← ε
 4: Create root node r = ((1,...,1),(|s_1|,...|s_m|)) with l^r = 0
 5: Add r to the initially empty node set N and priority queue Q
 6: optimal ← FALSE
 7: while Q ≠ ∅ and not optimal do
 8:     Take the first node v from priority queue Q
 9:     Determine Σ_v^nd from p^{L,v} and p^{R,v}
10:     if Σ_v^nd = ∅ then
11:         Derive a partial solution s represented by v
12:         if Σ_v ≠ ∅ then
13:             Choose a singleton a ∈ Σ_v
14:             s ← s · a · s^rev
15:         else
16:             s ← s · s^rev
17:         end if
18:         s_bsf ← s
19:         optimal ← TRUE
20:     else
21:         for a ∈ Σ_v^nd do
22:             Compute state v' that results from appending a at state v
23:             if v' ∈ N then
24:                 if l^v + 1 > l^{v'} then
25:                     l^{v'} ← l^v + 1
26:                     Update entry for v' in Q with new priority value π(v')
27:                 end if
28:             else
29:                 Add new node v' with l^{v'} = l^v + 1 to N and Q
30:             end if
31:         end for
32:     end if
33:     Remove v from Q
34: end while
```

where $\delta$ is an external parameter. Starting from the first node taken from $Q$, i.e., the highest priority node, we always expand as next node a newly generated immediate successor with highest priority value. This depth-first search is performed until no further newly generated successor exists (i.e., we do not further follow any already previously created nodes). If a new best solution is obtained in this way, it is stored in $s_{\mathrm{bsf}}$, which is returned in case of an early termination due to an imposed time limit. Note that, when extending a node during diving, the same steps regarding the update of the nodes in $N$ and $Q$ are performed as in A*. An important difference is, however, that nodes expanded during diving may now require a re-expansion at a later time when a longer partial solution is found for the respective state.

### 3.2 Beam Search for the LCPS Problem

With Beam Search (BS), we further consider an alternative, purely heuristic way of searching the state graph defined at the beginning of this section. BS [21] is a breadth-first search algorithm that explicitly limits the nodes examined at each level, for example, with an explicit upper bound of their number $\beta > 0$ called the *beam width*. Before presenting our specific BS for the LCPS problem, we define a dominance relation for nodes in the state graph considered at the same level of BS: Given nodes $u, v \in V$ we say $u$ *dominates* $v$ iff $u \neq v$ and $p_i^{\mathrm{L},u} < p_i^{\mathrm{L},v} \wedge p_i^{\mathrm{R},u} \geq p_i^{\mathrm{R},v}$ for all $i = 1, \ldots, m$.

The pseudo-code of our BS is provided in Algorithm 2. The *beam* $B$—that is, the set of nodes considered at each step of the algorithm—is initialized with the root node $r$. Then, at each step, the nodes of the current beam are extended in all possible ways, dominated nodes are filtered in function RemoveDominatedEntries($V_{\text{ext}}$), and the best $\beta$ nodes with respect to their priority values are selected in function Reduce($V_{\text{ext}}, \beta$) to obtain the beam for the next iteration.

**Algorithm 2** Beam Search (BS) for the LCPS problem

1: **Input:** an instance $(S, \Sigma)$
2: **Output:** $s_{\text{bsf}}$, the best solution found
3: $s_{\text{bsf}} \leftarrow \varepsilon$
4: Create root node $r = ((1, \ldots, 1), (|s_1|, \ldots |s_m|))$
5: Beam $B \leftarrow \{r\}$
6: **while** $B$ is not empty **do**
7: $\quad V_{\text{ext}} \leftarrow \emptyset$
8: $\quad$ **for** each $v \in B$ **do**
9: $\quad\quad$ Determine $\Sigma_v^{\text{nd}}$ from $p^{\mathrm{L},v}$ and $p^{\mathrm{R},v}$
10: $\quad\quad$ **if** $\Sigma_v^{\text{nd}} = \emptyset$ **then**
11: $\quad\quad\quad$ Derive a partial solution $s$ represented by $v$
12: $\quad\quad\quad$ **if** $\Sigma_v \neq \emptyset$ **then**
13: $\quad\quad\quad\quad$ Choose a singleton $a^* \in \Sigma_v$
14: $\quad\quad\quad\quad$ $s \leftarrow s \cdot a^* \cdot s^{\text{rev}}$
15: $\quad\quad\quad$ **else**
16: $\quad\quad\quad\quad$ $s \leftarrow s \cdot s^{\text{rev}}$
17: $\quad\quad\quad$ **end if**
18: $\quad\quad\quad$ **if** $|s| > |s_{\text{bsf}}|$ **then** $s_{\text{bsf}} \leftarrow s$ **end if**
19: $\quad\quad$ **else**
20: $\quad\quad\quad$ **for** $a \in \Sigma_v^{\text{nd}}$ **do**
21: $\quad\quad\quad\quad$ Compute state $v'$ that results from appending $a$ at state $v$
22: $\quad\quad\quad\quad$ $V_{\text{ext}} \leftarrow V_{\text{ext}} \cup \{v'\}$
23: $\quad\quad\quad$ **end for**
24: $\quad\quad$ **end if**
25: $\quad$ **end for**
26: $\quad V_{\text{ext}} \leftarrow$ RemoveDominatedEntries($V_{\text{ext}}$)
27: $\quad B \leftarrow$ Reduce($V_{\text{ext}}, \beta$)
28: **end while**

### 3.3 Embedding Beam Search in A*

Instead of the simple diving described for A* in Section 3.1, we may also apply above BS embedded within A* at regular intervals, always starting with the first entry of $Q$ as the initial node in beam $B$. As in simple diving, BS skips any already earlier encountered nodes (i.e., nodes that are already in $N$ are not added to $V_{\text{ext}}$) in order to avoid ineffective re-considerations of parts of the state graph. Therefore, it might happen—just like in the case of simple diving—that the embedded BS ends without delivering any complete solution. Moreover, as in simple diving, for all considered extensions of nodes, the same steps regarding the update of the nodes in $N$ and $Q$ are performed as in A*, cf. Algorithm 1. Finally, note that with beam width $\beta = 1$ the embedded BS corresponds to simple diving.

### 3.4 Tie Breaking

While executing preliminary experiments for A*, we realized that many ties occur when ordering the nodes in the priority queue $Q$ with respect to their priorities in $\pi(v)$. To guide the search in better ways, we decided to use the length of a represented longest partial solution as a secondary decision criterion in such cases. This improved the performance significantly, but still suffered from a significant number of ties. In order to also break these, it turned out to be beneficial to additionally consider the $p$-norm, which is for a node $v$ defined as

$$||v||_p = \left( \sum_{i=1}^{m} \left| p_i^{\mathrm{R},v} - p_i^{\mathrm{L},v} \right|^p \right)^{1/p}. \quad (7)$$

Given two nodes $u \neq v$ with the same priority value and the same maximum length concerning the represented partial solutions, a node with a lower $p$-norm is finally preferred. The inspiration for making use of this norm is that the smallest still relevant substrings potentially have a higher impact on the final length of complete solutions than the larger ones. However, considering only the shortest one of the still relevant substrings—that is, applying the min norm—could be highly misleading. Therefore, a $p$ value from $(0, 1)$ appears meaningful. Following further preliminary experiments, we finally chose $p = 0.5$ for all experiments discussed in the next section.

## 4 Experimental Results

The proposed algorithms were implemented in C++ using GCC 4.7.3 and all experiments were performed as single threads on Intel Xeon E5649 CPUs with 2.53 GHz and a memory limit of 15GB.

The benchmark instances used in this work were initially introduced in [4] in the context of the LCS problem and are provided at `https://www.ac.tuwien.ac.at/wp/wp-content/uploads/LCPS_instances.zip`. This set consists for each combination of the number of input strings $m \in \{10, 50, 100, 150, 200\}$, the length of the input

**Table 1.** Comparison of BS with $UB_1$ to BS with $UB_3$ on the 150 problem instances with $|\Sigma| = 4$.

| $m$ | $n$ | BS with $UB_1$ | | BS with $UB_3$ | | $UB_1$ versus $UB_2$ | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | $\overline{\lvert s \rvert}$ | $\bar{t}[s]$ | $\overline{\lvert s \rvert}$ | $\bar{t}[s]$ | $>$ (%) | $<$ (%) | $=$ (%) | $-$ (avg) |
| | 100 | 28.1 | $< 0.0$ | **28.5** | 0.3 | 74.7 | 10.0 | 15.3 | 5.5 |
| 10 | 500 | 150.7 | 0.1 | **151.5** | 76.8 | 95.6 | 2.0 | 2.4 | 54.3 |
| | 1000 | **304.7** | 0.4 | 304.3 | 656.1 | 97.9 | 1.0 | 1.2 | 122.5 |
| | 100 | 21.2 | 0.0 | **21.4** | 1.6 | 53.3 | 25.3 | 21.4 | 1.8 |
| 50 | 500 | 125.1 | 0.4 | **125.4** | 368.1 | 93.3 | 3.4 | 3.3 | 42.6 |
| | 1000 | **256.5** | 1.3 | – | 900.0 | 100.0 | 0.0 | 0.0 | 214.7 |
| | 100 | 19.5 | 0.1 | **19.9** | 3.1 | 48.8 | 27.0 | 24.2 | 1.3 |
| 100 | 500 | 118.3 | 0.7 | **119.4** | 174.5 | 93.4 | 3.4 | 3.2 | 42.6 |
| | 1000 | **245.1** | 2.3 | – | 900.0 | 100.0 | 0.0 | 0.0 | 209.4 |
| | 100 | 18.5 | 0.1 | **18.6** | 2.9 | 39.4 | 35.8 | 24.8 | 0.4 |
| 150 | 500 | 115.7 | 1.2 | **116.5** | 887.2 | 93.1 | 3.5 | 3.4 | 39.2 |
| | 1000 | **240.9** | 3.1 | – | 900.0 | 100.0 | 0.0 | 0.0 | 211.5 |
| | 100 | 17.9 | 0.1 | **18.1** | 3.6 | 39.5 | 35.6 | 24.8 | 0.4 |
| 200 | 500 | **114.2** | 1.4 | – | 900.0 | 92.0 | 4.3 | 3.7 | 36.5 |
| | 1000 | **237.7** | 4.0 | – | 900.0 | 100.0 | 0.0 | 0.0 | 212.4 |

strings $n \in \{100, 500, 1000\}$ and the alphabet size $|\Sigma| \in \{4, 12, 20\}$ of 10 randomly generated instances, yielding a total of 450 problem instances.

### 4.1 Comparison of Upper Bound Functions

In order to study the differences and mutual benefits of the two upper bound functions from Section 2.2, BS with $\beta = 10$ was applied both using only $UB_1$ and using $UB_3$, that is, the minimum of $UB_1$ and $UB_2$. The outcome is presented in Table 1. Each row shows average results over the 10 problem instances for each combination of $m$ and $n$. The results of BS with $UB_1$ are presented in terms of the obtained average solution quality ($\overline{|s|}$) and the average required computation time ($\bar{t}[s]$) in the third and fourth table column. The corresponding results of BS with $UB_3$ are listed in the fifth and sixth table column. The best result per table row is printed bold. The "–" symbol indicates that no complete solution of length greater than zero was derived within a CPU time limit of 900 seconds since the bound calculation took already too much time.

The following can be observed. First, when it is not too costly to calculate $UB_2$, as it is always the case for the instances with $n = 100$ and mostly when $n = 500$, BS using $UB_3$ is able to outperform BS using only $UB_1$. However, the high time complexity for calculating $UB_2$—that is, $O(mn^2)$—is a major obstacle in the context of larger problem instances. Because of these limitations, we perform all further experiments for BS, $A^*$, and the hybrid using only $UB_1$.

Nevertheless, the additional four columns in Table 1 clearly indicate that the usage of $UB_2$ can be promising also for larger instances. These columns show the percentages of nodes for which $UB_2$ dominates $UB_1$ ($>$ (%)), the percentages of nodes for which $UB_1$ dominates $UB_2$ ($<$ (%)), the percentage of nodes where both bounds are the same ($=$ (%)), and the average absolute values of subtracting $UB_2$ from $UB_1$ ($-$ (avg)). Results show that $UB_2$ dominates $UB_1$ especially for long input strings. A promising idea seems to be to either limit the time for calculating $UB_2$ or to calculate this bound only for a suitably chosen subset of all nodes. However, these studies are left for future work.

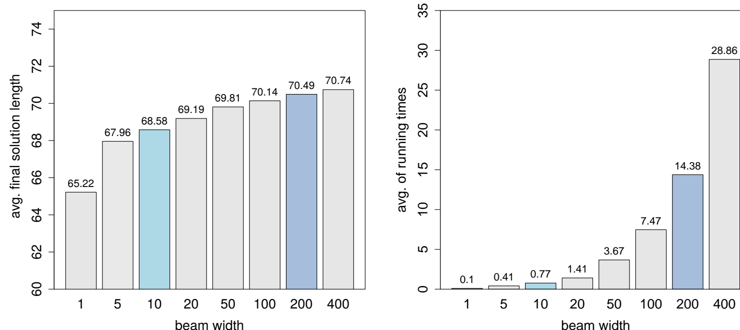


**Fig. 1.** Average final solution lengths and runtimes of BS with different beam widths $\beta$.

### 4.2 Main Results

We now compare the performance of our four solution approaches: (1) the greedy algorithm from Section 2, henceforth referred to as Greedy; (2) BS; (3) $A^*$ with simple diving, henceforth referred to as $A^*$+Dive; and (4) $A^*$ with embedded BS, henceforth referred to as $A^*$+BS.

For deciding how to choose the beam width $\beta$ in the stand-alone BS as well as in $A^*$+BS, we applied BS to each of the 450 problem instances. Average final solution lengths and runtimes are shown in Figure 1. As expected, with increasing beam width $\beta$ also the solution quality increases. However, this comes at the cost of an approximately linear increase of the runtime. Since the solution quality for $\beta = 400$ is only slightly better than that with $\beta = 200$, but the required times are about twice as large, we chose $\beta = 200$ for the standalone BS. For the embedded BS, we decided to use $\beta = 10$ due to the still relatively good results and small average runtime of only 0.77 seconds per instance.

The two variants of $A^*$ further require a setting for $\delta$, the number of regular $A^*$ iterations between diving/BS. We considered 5, 10, 50, 100, 500, and 1000 iterations and conducted preliminary experiments in a similar way as for $\beta$. Results (not shown) indicated that for $\delta = 10$, $A^*$ performs on average slightly but significantly better than with the other values. Therefore, we adopt this setting in our further tests for $A^*$+Dive and $A^*$+BS.

Results from the comparison of the four solution approaches are presented separately for instances of different alphabet sizes in Tables 2–4. Again, shown values are averages over the 10 instances of the same type, and best results from each row are printed bold. Optimal solution values (as determined by $A^*$+Dive and/or $A^*$+BS) are marked with an asterisk. For each algorithm, the table shows final average solution lengths, average runtimes, and additionally, for the algorithms $A^*$+Dive and $A^*$+BS the column $\bar{t}_{\text{best}}[s]$ which shows the average computation times at which the best found solutions were obtained. A limit of 900 seconds was imposed per run. The following observations can be made:

- By far the fastest algorithm is Greedy. However, Greedy also produces the weakest results in the comparison. Runtimes of the $A^*$ variants are generally higher, but of course these partly include proofs of optimality.
- Both $A^*$+Dive and $A^*$+BS are able to find optimal solutions for all instances with input string length $n = 100$. This corresponds to 15 out of 45 cases (table rows).

**Table 2.** Results for $|\Sigma|$=4.

| $m$ | $n$ | Greedy | | BS | | A*+Dive | | | A*+BS | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $\overline{\lvert s\rvert}$ | $\bar{t}[s]$ | $\overline{\lvert s\rvert}$ | $\bar{t}[s]$ | $\overline{\lvert s\rvert}$ | $\bar{t}[s]$ | $\bar{t}_{best}[s]$ | $\overline{\lvert s\rvert}$ | $\bar{t}[s]$ | $\bar{t}_{best}[s]$ |
| | 100 | 25.6 | < 0.1 | ***28.9** | 0.3 | ***28.9** | 13.2 | 5.6 | ***28.9** | 14.1 | 0.7 |
| 10 | 500 | 143.6 | < 0.1 | **157.4** | 2.7 | 147.5 | 900.0 | 310.8 | 156.5 | 900.0 | 222.1 |
| | 1000 | 292.6 | < 0.1 | **316.4** | 7.1 | 291.8 | 900.0 | 376.3 | 313.3 | 900.0 | 413.8 |
| | 100 | 19.4 | < 0.1 | 21.7 | 0.6 | ***21.8** | 6.6 | 1.9 | ***21.8** | 7.4 | 2.2 |
| 50 | 500 | 117.6 | < 0.1 | **128.0** | 7.8 | 123.8 | 900.0 | 148.1 | 127.8 | 900.0 | 137.6 |
| | 1000 | 251.0 | < 0.1 | **262.8** | 22.5 | 252.4 | 900.0 | 227.8 | 260.9 | 900.0 | 176.5 |
| | 100 | 18.1 | < 0.1 | 20.0 | 0.9 | ***20.1** | 8.8 | 2.0 | ***20.1** | 9.8 | 1.0 |
| 100 | 500 | 112.2 | < 0.1 | **121.4** | 13.9 | 118.4 | 900.0 | 219.5 | 121.3 | 900.0 | 203.7 |
| | 1000 | 240.3 | < 0.1 | **250.9** | 41.9 | 242.8 | 900.0 | 130.1 | 249.7 | 900.0 | 281.2 |
| | 100 | 16.3 | < 0.1 | ***19.0** | 1.2 | ***19.0** | 6.6 | 0.5 | ***19.0** | 7.6 | 0.2 |
| 150 | 500 | 108.4 | < 0.1 | 118.1 | 20.0 | 115.6 | 900.0 | 359.7 | **118.4** | 900.0 | 324.8 |
| | 1000 | 234.0 | 0.1 | **244.9** | 58.8 | 238.5 | 900.0 | 176.1 | 244.4 | 900.0 | 251.8 |
| | 100 | 16.4 | < 0.1 | 18.4 | 1.5 | ***18.5** | 8.7 | 2.1 | ***18.5** | 9.8 | 1.1 |
| 200 | 500 | 107.1 | < 0.1 | 115.9 | 25.6 | 113.8 | 900.0 | 238.0 | **116.5** | 900.0 | 336.2 |
| | 1000 | 227.2 | 0.2 | **241.4** | 77.1 | 235.9 | 900.0 | 363.2 | 241.1 | 900.0 | 171.8 |

**Table 3.** Results for $|\Sigma|$=12.

| $m$ | $n$ | Greedy | | BS | | A*+Dive | | | A*+BS | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $\overline{\lvert s\rvert}$ | $\bar{t}[s]$ | $\overline{\lvert s\rvert}$ | $\bar{t}[s]$ | $\overline{\lvert s\rvert}$ | $\bar{t}[s]$ | $\bar{t}_{best}[s]$ | $\overline{\lvert s\rvert}$ | $\bar{t}[s]$ | $\bar{t}_{best}[s]$ |
| | 100 | 8.9 | < 0.1 | ***9.6** | < 0.1 | ***9.6** | < 0.1 | < 0.1 | ***9.6** | < 0.1 | < 0.1 |
| 10 | 500 | 54.1 | < 0.1 | **60.6** | 2.7 | 57.6 | 900.0 | 273.3 | 60.3 | 900.0 | 17.8 |
| | 1000 | 113.8 | < 0.1 | **125.7** | 6.6 | 115.1 | 900.0 | 396.7 | 124.0 | 900.0 | 270.4 |
| | 100 | 4.7 | < 0.1 | ***5.6** | < 0.1 | ***5.6** | < 0.1 | < 0.1 | ***5.6** | < 0.1 | < 0.1 |
| 50 | 500 | 38.6 | < 0.1 | **43.1** | 6.1 | 41.8 | 900.0 | 124.0 | **43.1** | 900.0 | 40.1 |
| | 1000 | 83.9 | < 0.1 | **90.4** | 16.1 | 86.2 | 900.0 | 473.1 | 89.8 | 900.0 | 274.0 |
| | 100 | 3.9 | < 0.1 | ***4.6** | < 0.1 | ***4.6** | < 0.1 | < 0.1 | ***4.6** | < 0.1 | < 0.1 |
| 100 | 500 | 35.0 | < 0.1 | 38.7 | 9.8 | 37.5 | 900.0 | 102.1 | **39.0** | 900.0 | 141.7 |
| | 1000 | 77.8 | < 0.1 | **82.9** | 27.4 | 79.9 | 900.0 | 130.0 | 82.7 | 900.0 | 117.4 |
| | 100 | 3.5 | < 0.1 | ***3.8** | < 0.1 | ***3.8** | < 0.1 | < 0.1 | ***3.8** | < 0.1 | < 0.1 |
| 150 | 500 | 33.2 | < 0.1 | **37.0** | 13.7 | 36.0 | 900.0 | 61.4 | **37.0** | 900.0 | 105.7 |
| | 1000 | 72.7 | 0.1 | 79.2 | 37.2 | 77.2 | 900.0 | 211.8 | **79.4** | 900.0 | 94.0 |
| | 100 | 3.1 | < 0.1 | ***3.3** | < 0.1 | ***3.3** | < 0.1 | < 0.1 | ***3.3** | < 0.1 | < 0.1 |
| 200 | 500 | 31.3 | < 0.1 | **35.4** | 17.5 | 35.0 | 900.0 | 202.0 | 35.3 | 900.0 | 57.9 |
| | 1000 | 71.1 | 0.2 | **77.7** | 50.5 | 75.3 | 900.0 | 208.6 | 77.3 | 900.0 | 159.5 |

**Table 4.** Results for $|\Sigma|$=20.

| $m$ | $n$ | Greedy | | BS | | A*+Dive | | | A*+BS | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $\overline{\lvert s\rvert}$ | $\bar{t}[s]$ | $\overline{\lvert s\rvert}$ | $\bar{t}[s]$ | $\overline{\lvert s\rvert}$ | $\bar{t}[s]$ | $\bar{t}_{best}[s]$ | $\overline{\lvert s\rvert}$ | $\bar{t}[s]$ | $\bar{t}_{best}[s]$ |
| | 100 | 5.0 | < 0.1 | ***5.4** | < 0.1 | ***5.4** | < 0.1 | < 0.1 | ***5.4** | < 0.1 | < 0.1 |
| 10 | 500 | 32.7 | < 0.1 | 38.3 | 2.8 | 36.6 | 900.0 | 107.0 | **38.4** | 900.0 | 166.6 |
| | 1000 | 70.4 | < 0.1 | **79.3** | 6.6 | 73.7 | 900.0 | 213.7 | 78.5 | 900.0 | 163.5 |
| | 100 | 2.3 | < 0.1 | ***2.5** | < 0.1 | ***2.5** | < 0.1 | < 0.1 | ***2.5** | < 0.1 | < 0.1 |
| 50 | 500 | 21.7 | < 0.1 | **24.9** | 5.5 | 24.5 | 900.0 | 70.4 | **24.9** | 900.0 | 3.8 |
| | 1000 | 48.7 | < 0.1 | **54.3** | 15.8 | 51.6 | 900.0 | 159.1 | 53.7 | 900.0 | 115.8 |
| | 100 | ***1.3** | < 0.1 | ***1.3** | < 0.1 | ***1.3** | < 0.1 | < 0.1 | ***1.3** | < 0.1 | < 0.1 |
| 100 | 500 | 18.5 | < 0.1 | **21.9** | 9.1 | 21.0 | 900.0 | 1.5 | 21.8 | 900.0 | 62.0 |
| | 1000 | 44.0 | 0.1 | **48.7** | 26.7 | 47.0 | 900.0 | 46.5 | 48.4 | 900.0 | 40.1 |
| | 100 | ***1.1** | < 0.1 | ***1.1** | < 0.1 | ***1.1** | < 0.1 | < 0.1 | ***1.1** | < 0.1 | < 0.1 |
| 150 | 500 | 17.8 | < 0.1 | 20.5 | 11.6 | 20.1 | 900.0 | 46.9 | **20.6** | 900.0 | 165.5 |
| | 1000 | 40.1 | 0.1 | **46.0** | 37.6 | 44.9 | 900.0 | 201.4 | 45.8 | 900.0 | 81.5 |
| | 100 | ***1.1** | < 0.1 | ***1.1** | < 0.1 | ***1.1** | < 0.1 | < 0.1 | ***1.1** | < 0.1 | < 0.1 |
| 200 | 500 | 16.9 | < 0.1 | 19.1 | 14.9 | 19.0 | 900.0 | 6.3 | **19.4** | 900.0 | 79.1 |
| | 1000 | 39.8 | 0.2 | **44.7** | 46.8 | 43.1 | 900.0 | 60.3 | 44.6 | 900.0 | 213.6 |

- In most of those cases in which the $A^*$ variants cannot find optimal solutions, $A^*$+BS outperforms $A^*$+Dive. This shows the benefit of using BS as embedded heuristic as opposed to simple diving.
- In those cases where the $A^*$ versions are able to find optimal solutions and prove their optimality, BS is most of the time also able to find solutions of equal quality. However, this seems to become more difficult for BS when the alphabet size decreases. In particular, BS failed to find all optimal solutions in three out of five cases with $|\Sigma| = 4$.
- From a pure heuristic point of view, BS outperforms $A^*$+BS more and more when the length of the input strings increases. More specifically, while the results obtained by BS and the $A^*$+BS are comparable for instances with $n = 500$, BS generally outperforms $A^*$+BS for instances with $n = 1000$.

## 5 Conclusions and Future Work

We proposed different algorithms for solving the LCPS problem with an arbitrary number of strings heuristically as well as exactly. A general state graph was defined that can be searched by different strategies. With BS we provided a pure heuristic search that scales well to also large instances. With $A^*$ we provided an efficient method for solving instances with up to 200 strings of lengths up to 100 to proven optimality. Since for instances with even larger strings, $A^*$ search cannot find a complete solution in a reasonable time, it is upgraded to an anytime algorithm by embedding either the simple diving or the more advanced BS. For the instances where our hybrid algorithms do not find optimal solution, the optimality gaps between final (heuristic) solutions and the corresponding upper bounds produced by $A^*$ are not so tight. The reason for this is that $UB_1$ partly provides only rather weak bounds. Using $UB_1$ in combination with $UB_2$, i.e., $UB_3$, would clearly be beneficial from the quality point-of-view, but the larger time complexity of $UB_2$ makes this approach prohibitive for larger instances. In future work the strengthening of the upper bounds seems to be most promising. We believe that this can be achieved by applying $UB_2$ only for subproblems up to a certain size or by finding an approximation of $UB_2$ that can be calculated in a faster way. Testing the algorithms with real world instances, e.g., coming from protein, DNA and virus structure sequences, would also be interesting, since such instances may have special structures on which the algorithms might perform differently or which might be further exploited.

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

## References

1. L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In *Proceedings of SPIRE 2000 – 7th International Symposium on String Processing and Information Retrieval*, pages 39–48. IEEE press, 2000.
2. C. Blum, M. J. Blesa, and M. López-Ibáñez. Beam search for the longest common subsequence problem. *Computers & Operations Research*, 36(12):3178–3186, 2009.

3. C. Blum and P. Festa. Longest common subsequence problems. In *Metaheuristics for String Problems in Bioinformatics*, chapter 3, pages 45–60. Wiley, 2016.
4. C. Blum and G. R. Raidl. *Hybrid Metaheuristics: Powerful Tools for Optimization*. Springer, 2016.
5. P. Bonizzoni, G. Della Vedova, and G. Mauri. Experimenting an approximation algorithm for the LCS. *Discrete Applied Mathematics*, 110(1):13–24, 2001.
6. P. Brisk, A. Kaplan, and M. Sarrafzadeh. Area-efficient instruction set synthesis for reconfigurable system-on-chip design. In *Proceedings of the 41st Design Automation Conference*, pages 395–400. IEEE press, 2004.
7. C. Q. Choi. DNA palindromes found in cancer. *Genome Biology*, 6:1–3, 2005. spotlight-20050216-01.
8. S. R. Chowdhury, M. M. Hasan, S. Iqbal, and M. S. Rahman. Computing a longest common palindromic subsequence. *Fundamenta Informaticae*, 129(4):329–340, 2014.
9. C. B. Fraser. *Subsequences and Supersequences of Strings*. PhD thesis, University of Glasgow, Glasgow, UK, 1995.
10. M. Giel-Pietraszuk, M. Hoffmann, S. Dolecka, J. Rychlewski, and J. Barciszewski. Palindromes in proteins. *Journal of Protein Chemistry*, 22(2):109–113, 2003.
11. D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Computer Science and Computational Biology. Cambridge University Press, Cambridge, 1997.
12. P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.
13. M. M. Hasan, A. S. M. Sohidull Islam, M. Sohel Rahman, and A. Sen. Palindromic subsequence automata and longest common palindromic subsequence. *Mathematics in Computer Science*, 11:219–232, 2017.
14. K. Huang, C. Yang, and K. Tseng. Fast algorithms for finding the common subsequences of multiple sequences. In *Proceedings of the IEEE International Computer Symposium*, pages 1006–1011. IEEE press, 2004.
15. S. Inenaga and H. Hyyrö. A hardness result and new algorithm for the longest common palindromic subsequence problem. *Information Processing Letters*, 129:11–15, 2018. Supplement C.
16. T. Jiang, G. Lin, B. Ma, and K. Zhang. A general edit distance between RNA structures. *Journal of Computational Biology*, 9(2):371–388, 2002.
17. S. Larionov, A. Loskutov, and E. Ryadchenko. Chromosome evolution with naked eye: Palindromic context of the life origin. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 18(1), 2008.
18. D. Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM*, 25(2):322–336, 1978.
19. W. J. Masek and M. S. Paterson. A faster computing string edit distances. *Theoretical Computer and System Sciences*, 20:18–31, 1980.
20. S. R. Mousavi and F. Tabataba. An improved algorithm for the longest common subsequence problem. *Computers & Operations Research*, 39(3):512–520, 2012.
21. P. S. Ow and T. E. Morton. Filtered beam search in scheduling. *International Journal of Production Research*, 26:297–307, 1988.
22. J. Storer. *Data Compression: Methods and Theory*. Computer Science Press, MD, USA, 1988.
23. H. Tanaka, D. A. Bergstrom, M.-C. Yao, and S. J. Tapscott. Large DNA palindromes as a common form of structural chromosome aberrations in human cancers. *Human Cell*, 19(1):17–23, 2006.