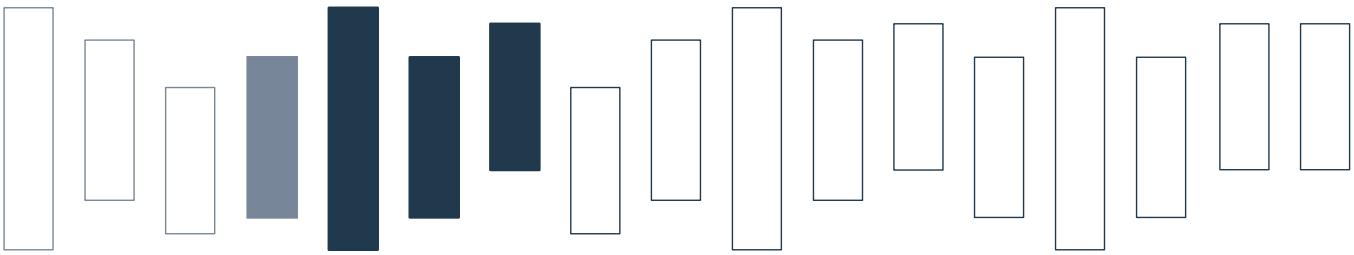
Technical Report ac-tr-18-012

December 2018

Anytime Algorithms for the Longest Common Palindromic Subsequence Problem

Marko Djukanovic, Guenther Raidl, Christian Blum



This paper submitted to the journal Computers & Operations Research

www.ac.tuwien.ac.at/tr

Anytime Algorithms 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 aims at finding a longest string that appears as a subsequence in each of a set of input strings and is a palindrome at the same time. The problem is a special variant of the well known longest common subsequence problem and has applications in particular in genomics and biology, where strings correspond to DNA or protein sequences and similarities among them shall be detected or quantified. We first present a more traditional A* search that makes use of an advanced upper bound calculation for partial solutions. This exact approach works well for instances with two input strings and, as we show in experiments, outperforms several other exact methods from the literature. However, the A* search also has natural limitations when a larger number of strings shall be considered due to the problem's complexity. To effectively deal with this case in practice, we investigate anytime A* variants, which are able to return a reasonable heuristic solution at almost any time and are expected to find better and better solutions until reaching a proven optimum when enough time given. In particular we propose a novel approach in which Anytime Column Search (ACS) is interleaved with traditional A* node expansions. The ACS iterations are guided by a new heuristic function that approximates the expected length of an LCPS in subproblems usually much better than the available upper bound calculation. This A*+ACS hybrid is able to solve small to medium-sized LCPS instances to proven optimality while returning good heuristic solutions together with upper bounds for large instances. In rigorous experimental evaluations we compare A*+ACS to several other anytime A* search variants and observe its superiority.

Keywords: longest common palindromic subsequence problem, anytime algorithms, A* search, hybrid algorithms, expected value.

1. Introduction

In computer science, a *string* s is a finite sequence consisting of symbols from a finite set Σ called alphabet. The length of a string s, denoted by $|s|$, is defined as the number

of symbols in s. Strings are generally used as a data type for storing, for example, words or even complete texts. In the field of bioinformatics, strings are used to represent DNA and protein sequences. As a consequence, many computational problems in bioinformatics can be phrased in terms of so-called string problems. A string s is called *palindrome* iff $s = s^{\text{rev}}$, where s^{rev} is the reverse string of s. Consider, for example, the palindrome `madam`. Furthermore, given a string s, any string which can be obtained from s by deleting zero or more characters is called a *subsequence* of s. A variety of applications are based on so-called string comparison measures. One of the most well-known string comparison measures in bioinformatics is the length of the *Longest Common Subsequence* (LCS) [38] of two or more input strings. More specifically, the famous LCS problem consists in finding the longest common subsequence of a set $S = \{s_1, \ldots, s_m\}$ of $m \geq 2$ input strings. Even though the LCS problem was originally defined in bioinformatics, the problem has also found applications in other areas such as data comparison and data compression [31, 44]. Well known examples where the LCS problem plays important roles are the *diff* command on Unix systems and the Git version control system.

In this work we tackle a variant of the LCS problem known as the *Longest Common Palindromic Subsequence* (LCPS) problem. Given a set S of $m \geq 2$ input strings over a finite alphabet Σ, the problem consists in finding a longest common subsequence that, at the same time, is a palindrome. Palindromes appear to be of particular interest in the biological context as palindromic motifs are frequently found in DNA sequences. A research project on genome sequencing discovered palindromic regions on the Y-chromosome [34]. These palindromic structures allow the Y-chromosome to repair itself by bending over at the middle if one side is damaged. Furthermore, biologists believe that identifying palindromic DNA subsequences may help to understand genomic instability [11, 45]. 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. There is also strong evidence that palindromes of $\gtrapprox 500$ base pairs on human chromosomes 11 and 22 have harmful consequences such as initiating chromosomal translocation, which may result in cancer or developmental defects; see for example [43]. Finally, it is believed that palindromes are frequently found in proteins [21]. However, their role in the protein function is less understood. For biologists it is of importance to find not only palindromic subsequences of an individual DNA sequence, but also the longest common palindromic subsequence of multiple strings in order to learn about the relations between these strings.

With the exception of our preliminary work [17], prior work on the LCPS problem only considered the special case with two input strings (2–LCPS) [12, 26, 29]. In particular, these studies deal with exact approaches to solve the 2–LCPS problem. The generalization to an arbitrary number of $m \geq 2$ of input strings is challenging from a theoretical as well as practical point of view. From [38] it is known that the LCS problem is $\mathcal{NP}$–hard for an arbitrary number $m \geq 2$ of input strings, and since there exists a polynomial-time reduction from the LCS problem to the LCPS problem, and vice versa [29], this implies $\mathcal{NP}$–hardness of the LCPS. Moreover, although still an open question, it is believed that the computational complexity of the LCPS for a fixed $m \geq 2$ is at least $O(n^{2m})$, where n is the length of the longest input string [1]. In a preliminary work [17], we proposed the first algorithms for solving the general LCPS problem with an arbitrary number of input strings. We introduced an exact A* search, a heuristic *Beam Search* (BS), and a hybrid of both, labeled A*+BS. The latter is a so-called *anytime* approach.

Generally speaking, an anytime algorithm is able to return a valid solution even if it is interrupted at (almost) any time, and when it keeps running it is expected to find better

and better solutions until reaching a proven optimum. The study of anytime algorithms is an important field in optimization and artificial intelligence. In many challenging practical applications exact techniques are appreciated but sometimes too time-demanding. Pure heuristics, however, are also not satisfying as they usually do not provide performance guarantees. According to Dean and Boddy [15] and Horvitz [27], who used the term anytime algorithm for the first time in literature, these kind of methods offer to choose the trade-off between solution quality against the computational requirements. The A*+BS, is—in contrast to classical A*—able to provide good heuristic solutions early during the search process, while maintaining completeness when given enough time. The algorithm switches in regular intervals between the classical A* search strategy and a BS-oriented width-limited breath-first search for quickly obtaining high-quality heuristic solutions.

This article is an extension of the above mentioned preliminary work presented at a conference [17]. We propose an advanced anytime approach based on A*—labeled A*+ACS—that differs in the following aspects from the previous A*+BS proposal from [17]:

- The A* algorithm itself is improved by a more efficient implementation, utilizing in particular a tighter upper bound calculation for partial solutions.
- Instead of standard beam search, a new mechanism for expanding nodes and finding promising heuristic solutions is used. More specifically, the beam search component of A*+BS is replaced by *Anytime Column Search* (ACS) [48].
- In addition to the above-mentioned improved upper bound calculation, a novel heuristic function is derived and used for guiding the search in the context of the ACS iterations. This function approximates the expected length of the LCPS.

Our computational results show that A*+ACS is a new state-of-the-art approach for the LCPS problem concerning the quality of the obtained heuristic solutions. It substantially improves over A*+BS not only in terms of solution quality, but also in terms of the anytime behavior and the evolution of the gaps between obtained upper bounds and the heuristic solution values. Moreover, a comparison to other A*-based anytime search variants utilizing similar ingredients such as our new upper bound calculation shows the particular benefits of the proposed A*+ACS combination. Last but not least, we also experimentally compare our pure A* approach to the 2–LCPS algorithms from the literature. This is especially interesting, since—as far as we know—no study existed yet providing a comparison of the available 2–LCPS approaches. Our results show in particular, that the proposed A* scales better than the other approaches with growing input string length.

1.1. Organization of the Work

The article is organized as follow. Section 2 surveys the existing literature concerning the LCPS. In Section 3 we describe the A* search for the LCPS problem together with two considered upper bounds calculations. The new heuristic function for evaluating partial solutions based on the approximated expected length of the LCPS is provided in Section 4. In Section 5 we present our new anytime algorithm A*+ACS. All experimental evaluations and comparisons are presented in Section 6. Finally, conclusions and an outlook on promising future work are given in Section 7. We provide also a supplementary document at `https://www.ac.tuwien.ac.at/research/problem-instances/LCPS` containing more complete results and further implementation details for algorithms to which we compare.

2. Problem-Related Literature

The general LCS problem has been tackled by means of exact and heuristic approaches during the last decades. The problem is polynomially solvable for any fixed value of m by means of *Dynamic Programming* (DP) in $O(n^m)$ time [22]. Remember that n is the length of the longest string among the m input strings. As a consequence, DP is considered not practical for larger m. An approach that systematically enumerates common subsequences has been proposed by Hsu and Du [28] and is further improved by incorporating specialized branch & bound rules [18]. Singireddy [42] proposed an integer linear programming approach based on branch & cut. Considering approximation algorithms, Jiang and Li [30] suggested a simple Long Run (LR) algorithm which finds an LCS consisting of a single letter, resulting in a $|\Sigma|$-approximation ratio. The best-next heuristic from [51] also guarantees an approximation ratio of $|\Sigma|$. However, it typically performs much better than LR considering practical solution quality. Concerning other greedy and approximation algorithms for the LCS problem, see [4, 8, 9]. Within the last ten years, many metaheuristic approaches for solving the LCS problem were proposed. Easton and Singireddy [19] described a large neighborhood search that makes internal use of the specialized branch & bound rules from [18]. This heuristic was able to outperform all existing approximation and greedy algorithms proposed at that time. At approximately the same time, Shyu and Tsai [41] proposed an ant colony optimization approach. A breakthrough in solution quality was finally achieved with BS as described by Blum et al. [6]. An extension of this BS approach was presented in [39]. This algorithm uses a heuristic function—instead of an upper bound—for guiding the search process of BS. Hybrid algorithms such as [36] and [5, 7] were able to further improve over this BS for certain subsets of the commonly used benchmark instances. The currently best approach appears to be an algorithm labeled *Chemical Reaction Optimization* from [40]. However, this algorithm has not yet been re-implemented and is therefore to be considered with care.

As mentioned before, apart from our preliminary work [17], the LCPS problem has only been studied for two input strings; that is, the 2–LCPS problem version. Chowdhury et al. [12] solved the 2–LCPS problem in two ways: (1) by using a standard dynamic programming approach that requires $O(n^4)$ time and space and (2) by using a sparse dynamic programming approach which runs in time $O(\mathcal{R}^2 \log^2 n \log\log n + n)$ and requires $O(\mathcal{R}^2)$ space, where $\mathcal{R}$ denotes the number of matching position pairs between the strings. The sparse approach uses a special data structure—a 3D balanced range search tree—for which the 2-LCPS problem instances is initially transformed into a geometric problem called Minimum Nested Depth Rectangular Structures (MNDRS). Hasan et al. [26] solved the 2–LCPS problem by deriving and using weighted finite automata which they called Common Palindromic Subsequence Automata (CPSA). The algorithm runs in time $O(n + R_1|\Sigma| + n + R_2|\Sigma| + R_1R_2|\Sigma|)$, where R_1 and R_2 denote the numbers of states of the two automata constructed for the input strings, respectively (bounded by $O(n^2)$). Finally, Inenaga and Hyyrö [29] presented an algorithm that solves the 2–LCPS by making use of certain preprocessing steps, a reduction phase and special data structures, solving the equivalent MNDRS problem; this method runs in time $O(\sigma\mathcal{R}^2 + n)$ and requires $O(\mathcal{R}^2 + n)$ space, where σ denotes the number of distinct characters that occur in both strings. The authors also proved that the 2–LCPS is computationally at least as hard as the 4–LCS. Abboud et al. [1] showed that if there exists an algorithm which solves the 2–LCPS problem in $O(n^{4-\lambda})$ time for some $\lambda > 0$, then a strong exponential time hypothesis fails. A proof is given to conclude that the LCPS for m strings is at least as hard as the LCS for twice

as many strings.

As we already pointed out, the first work considering the LCPS for an arbitrary number of input strings is our preliminary work from [17]. The proposed A* was able to solve rather small instances within a short time to optimality, but applying it to larger instances was practically infeasible due to too excessive memory and time requirements. This issue was addressed by extending the algorithm to the hybrid A*+BS, which obtains reasonable solutions also for larger instances within practical time and memory limitations.

3. A* Search for the LCPS

We start with some common notation definitions. As we already stated, let n be the maximum length of the strings in S. The j-th letter of a string s is denoted by $s[j]$, with $j = 1, \ldots, |s|$. We further state the concatenation of two strings by the 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 position j and ending at position j'; if $j > j'$, we define $s[j, j']$ to correspond to the empty string ε. Finally, let $|s|_a$ be the number of occurrences of letter $a \in \Sigma$ in string s. Henceforth, a string s is called a *(valid) partial solution* concerning input strings $S = \{s_1, \ldots, s_m\}$, if $s \cdot s^{\text{rev}}$ is a common palindromic subsequence of the strings in S.

A* search is a widely used problem solving technique belonging to the class of informed search methods, in general for finding shortest or longest paths [25]. It is based on the best-first search principle and acts on a weighted directed state graph $G = (N, A)$ with node set N and arc set A, expanding/processing always a node which is one of the most promising ones at that moment. To estimate the quality of nodes, A* makes use of a function $f(v) := g(v) + h(v),\ v \in N$, where $g(v)$ denotes the so far best known cost of a path from a dedicated root node in G to node v. Moreover, the heuristic function $h(v)$ provides an estimate for the still possible cost to reach from v a goal node. The performance of an A* algorithm mostly depends on the tightness of the heuristic function. Problem-specific aspects in order to realize an A* search for a specific problem are primarily to define (1) the state graph including the root and goal nodes and (2) the heuristic function h. This will be done in the following for the LCPS problem.

3.1. State Graph

Let $p^{\text{L}}, p^{\text{R}} \in \mathbb{N}^m$ be m-dimensional integer valued vectors such that $1 \leq p_i^{\text{L}} \leq p_i^{\text{R}} \leq |s_i|$ for all $i = 1, \ldots, m$. Given such vectors p^{L} and p^{R}, set $S[p^{\text{L}}, p^{\text{R}}] := \{s_i[p_i^{\text{L}}, p_i^{\text{R}}] \mid i = 1, \ldots, m\}$ consists of a continuous substring $s_i[p_i^{\text{L}}, p_i^{\text{R}}]$ for each input string s_i, $i = 1, \ldots, m$. Hereby, p^{L} is called the *left position vector* and p^{R} is called the *right position vector*. Moreover, $S[p^{\text{L}}, p^{\text{R}}]$ is henceforth called a *subproblem* of the original LCPS problem. For the definition of the state graph of the A* approach we only consider those subproblems that are induced by valid partial solutions. More specifically, we say that a valid partial solution s *induces* a subproblem $S[p^{\text{L}}, p^{\text{R}}]$ iff the following two conditions hold:

1. $s_i[1, p_i^{\text{L}} - 1]$ is a minimal string among all strings $s_i[1, x]$ with $1 \leq x \leq p_i^{\text{L}} - 1$ containing s as a subsequence for all $i = 1, \ldots, m$.
2. $s_i[p_i^{\text{R}} + 1, |s_i|]$ is a minimal string among all strings $s_i[x, |s_i|]$ with $p_i^{\text{R}} + 1 \leq x \leq |s_i|$ containing s^{rev} as a subsequence for all $i = 1, \ldots, m$.

In this context, note that the same subproblem may be induced by more than one valid partial solution. As an example consider $S = (\texttt{abccdccba}, \texttt{baccdccab})$, and partial solutions $s = \texttt{ac}$ and $s' = \texttt{bc}$. 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 in the state graph. Here, s and s' have the same length, but this need not be the case in general.

The state graph of our A* search is a directed *acyclic graph* $G = (V, A)$ in which each node corresponds to a unique state $v = (p^{\mathrm{L},v}, p^{\mathrm{R},v}) \in V$ and thus also to a unique LCPS subproblem $S[p^{\mathrm{L},v}, p^{\mathrm{R},v}]$. For the reason outlined above, a node v may potentially be induced by multiple valid partial solutions. The special root node $r = ((1, \ldots, 1), (|s_1|, \ldots, |s_m|)) \in V$ represents the original LCPS problem, which can be said to be induced by the empty partial solution ε.

An arc $(v, v') \in A$ leading from a node $v \in V$ to a node $v' \in V$ corresponds to the extension of a partial solution s inducing v to a partial solution s' inducing v' by one specific letter $a \in \Sigma$, i.e., $s' = s \cdot a$. Any path from the root node r to any node in V represents a valid partial solution, which is directly given by the sequence of letters associated with the arcs.

A letter $a \in \Sigma$ is called *feasible* for a node $v \in V$ if it appears at least twice in each substring of subproblem $S[p^{\mathrm{L},v}, p^{\mathrm{R},v}]$. Let $a, b \in \Sigma$ be two feasible letters with respect to node v. Moreover, let v' be the node corresponding to the subproblem induced by appending a to some feasible partial solution inducing v, and let v'' be the node induced by appending b to some feasible partial solution inducing v. We say that letter b is *dominated* by letter a (or a is dominating b) iff $p_i^{\mathrm{L},v'} \leq p_i^{\mathrm{L},v''} \wedge p_i^{\mathrm{R},v''} \leq p_i^{\mathrm{R},v'}\ \forall i = 1, \ldots, m$. Obviously, arcs that correspond to appending dominated letters do not need to be considered. Therefore, an arc (v, v') exists in our state graph G iff the letter that is used for obtaining v' from v is (1) feasible and (2) non-dominated. This subset of letters with respect to a node v is henceforth denoted by $\Sigma_v^{\mathrm{nd}} \subseteq \Sigma$. In other words, each node $v \in V$ has an outgoing arc $(v, v') \in A$ for each letter $a \in \Sigma_v^{\mathrm{nd}}$. Letters that do not appear in at least one of the substrings of $S[p^{\mathrm{L},v}, p^{\mathrm{R},v}]$ are called *infeasible letters*. Moreover, letters that appear at least once in each substring of $S[p^{\mathrm{L},v}, p^{\mathrm{R},v}]$, and exactly once for at least one $i \in \{1, \ldots, m\}$, are called *singleton letters*. Nodes $v \in V$ without any outgoing arcs are *goal nodes*.

In order to solve the LCPS problem, we are looking for a longest path in the state graph leading from the root node r to some goal node. In respect to the number of arcs, such a longest path represents a longest partial solution s. The corresponding palindromic subsequence is $s \cdot s^{rev}$, but note that we may still be able to insert, as a last step, a singleton letter $a \in \Sigma$ in the middle, yielding the longer palindrome $s \cdot a \cdot s^{\mathrm{rev}}$. Thus, we have to either find a longest partial solution that allows such an extension by a singleton letter or prove that no other equally long path with a singleton-extension exists, in which case we may then return $s \cdot s^{\mathrm{rev}}$ as LCPS. To account for this possible insertion of a final middle element and to reflect by the path lengths in our state graph the actual lengths of resulting palindromes, we assign each arc $(u, v) \in A$ a length

$$\ell(u, v) = \begin{cases} 3 & \text{if } v \text{ is a goal node and } S[p^{\mathrm{L},v}, p^{\mathrm{R},v}] \text{ contains a singleton letter} \\ 2 & \text{else.} \end{cases} \tag{1}$$

When we refer to the length of a path from now on, we therefore mean the sum of these lengths of all the arcs forming the path.

During our search—as described in detail below—nodes $v \in V$ store as additional information the length l_v of the currently longest path from r to v.[1]

[1] In this context we emphasize that it is not necessary to store actual partial solutions s with the nodes.

3.2. Upper Bounds for the Length of an LCPS

Remember that A* depends on a heuristic function $h(v)$ for estimating the still possibly length of a longest path from node $v \in V$ to some goal node, i.e., for the length of the LCPS of the subproblem represented by v. This function is generally implemented—in the context of maximization problems such as the LCPS problem—in terms of an upper bound in order to ensure *admissibility*, i.e., the completeness of the A* search. In the following we present a combined upper bound UB composed of two individual upper bounds UB_1 and UB_2, that is $\mathrm{UB}(v) = \min\{\mathrm{UB}_1(v), \mathrm{UB}_2(v)\}$ for all $v \in V$. Hereby, UB_1 is based on the UB_1 bound from [17], while UB_2 is newly developed. In contrast to [17], an appropriate pre-processing action for speeding up the calculation of UB_1 in a significant way is used.

Let us denote by c_a the minimum number of occurrences of a letter $a \in \Sigma$ in the subproblem represented by a current node $v \in V$, that is, $c_a := \min_{i=1,\dots,m} |s_i[p_i^{\mathrm{L},v}, p_i^{\mathrm{R},v}]|_a$. A simple upper bound is given by

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

The last term considers the fact that at most one singleton letter can finally 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 Σ with an odd value of c_a. In a naive fashion, $\mathrm{UB}_1(v), v \in V$ is calculated in $O(mn)$ time. However, the repeated calculation of UB_1 can be sped up by a pre-processing step that determines the number of occurrences of each letter in all postfixes of all input strings in advance. More precisely, we predetermine

$$\omega_{i,j,a} = \left| s_i[j, |s_i|] \right|_a \qquad \forall i = 1, \dots, m,\ j = 1, \dots, n+1,\ a \in \Sigma. \tag{3}$$

During the actual A* search, c_a can then be efficiently determined for a current state v as $c_a := \min_{i=1,\dots,m}(\omega_{i,p_i^{\mathrm{L},v},a} - \omega_{i,p_i^{\mathrm{R},v}+1,a})$, and UB_1 can be calculated in $O(m|\Sigma|)$ time.

Although the second upper bound UB_2 from [17] is comparably tight, it was judged to be impractical due to being computationally too expensive. Therefore, in this article we propose an alternative UB_2 bound, which is based on the standard DP procedure for calculating the LCS of two input strings; see, for example, [50]. More specifically, this algorithm for determining the LCS of two strings s_i and s_j consists in filling a $(|s_i| + 1) \times (|s_j| + 1)$ matrix M_{ij}, whose entries $\mathrm{M}_{ij}[x, y]$ finally correspond to the lengths of the longest common subsequence for $s_i[x, \dots, |s_i|]$ and $s_j[y, \dots, |s_j|]$ with $x = 1, \dots |s_i|+1,\ y = 1, \dots, |s_j| + 1$. Hereby, all entries with $x = |s_i| + 1$ or $y = |s_j| + 1$ are set to zero. The content of all other entries is determined with the following recursive formula:

$$M_{ij}[x-1, y-1] = \begin{cases} M_{ij}[x, y] + 1, & \text{if } s_i[x] = s_j[y] \\ \max\{M_{ij}[x, y-1], M_{ij}[x-1, y]\}, & \text{otherwise.} \end{cases}$$

What we call the complete upper bound $\mathrm{UB}_2^{\mathrm{comp}}(v)$ for a node $v \in V$—that is, for the subproblem $S[p^{\mathrm{L},v}, p^{\mathrm{R},v}]$ of still relevant substrings—can now be computed as

$$\mathrm{UB}_2^{\mathrm{comp}}(v) := \min_{1 \le i < j \le m} \left(M_{ij}[p_i^{\mathrm{L},v}, p_j^{\mathrm{L},v}] - M_{ij}[p_i^{\mathrm{R},v} + 1, p_j^{\mathrm{R},v} + 1] \right). \tag{4}$$

For any node in the graph the longest path to it and the respective partial solution can finally be efficiently derived in a backward manner by iteratively identifying predecessors in which the l_v-values always decrease by two when l_v is even or by three if l_v is odd.

$\mathrm{UB}_2^{\mathrm{comp}}$ is, for the following reasons, indeed an upper bound for the length of a LCPS of $S[p^{\mathrm{L},v}, p^{\mathrm{R},v}]$. Let $S' = \{s'_1, \ldots, s'_m\}$ and $S'' = \{s''_1, \ldots, s''_m\}$ be two sets, each one containing m strings. Moreover, let $C = \{s'_1 \cdot s''_1, \ldots, s'_m \cdot s''_m\}$. Obviously it holds that $\mathrm{LCS}(S')+\mathrm{LCS}(S'') \leq \mathrm{LCS}(C)$, where $\mathrm{LCS}(S')$ denotes the length of the LCS of the strings in S', etc. This immediately implies that $\mathrm{LCS}(S') \leq \mathrm{LCS}(C) - \mathrm{LCS}(S'')$. Moreover, it holds that the length of the LCPS for S' cannot exceed $\mathrm{LCS}(S')$. However, since the number of strings in S can be large, we replace the complete upper bound $\mathrm{UB}_2^{\mathrm{comp}}$ with the following final definition of UB_2, which is a faster approximation of $\mathrm{UB}_2^{\mathrm{comp}}$:

$$\mathrm{UB}_2(v) := \min_{i=1,\ldots,m-1} \left(M_{i,i+1}[p_i^{\mathrm{L},v}, p_{i+1}^{\mathrm{L},v}] - M_{i,i+1}[p_i^{\mathrm{R},v}+1, p_{i+1}^{\mathrm{R},v}+1] \right). \tag{5}$$

Note that for the efficient evaluation of (5), we have to determine the matrices $M_{i,i+1}$, $i = 1, \ldots, m-1$ in a preprocessing step, which can be achieved in $O(mn^2)$ time. We can then calculate $\mathrm{UB}_2(v)$ for any node $v \in V$ in $O(m)$ time.

3.3. Details of the A* Search for LCPS

A* 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 successors 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 $(p^{\mathrm{L},v}, p^{\mathrm{R},v})$, or to determine that no respective node exists yet. The set of open nodes Q is realized by means of a heap in which the nodes are partially sorted according to the priority function $f(v) = g(v) + h(v) := l_v + \mathrm{UB}(S[p^{\mathrm{L},v}, p^{\mathrm{R},v}])$. In case of ties, nodes with larger l_v are preferred. In case of further ties, they are broken by considering the distance between the positions $p^{\mathrm{L},v}$ and $p^{\mathrm{R},v}$ as measured by means of the k-norm, for some $k > 0$ being a parameter of the algorithm.

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 a goal node. In this case the algorithm derives the actual partial solution corresponding to the longest path to v and returns with the resulting palindrome. Note that if a singleton letter remains in $S[p^{\mathrm{L},v}, p^{\mathrm{R},v}]$, it is added as middle letter. Since our priority function is *admissible*, cf. [25], we can be sure that an optimal solution has been reached. Otherwise, node v is extended by considering each possible extension $a \in \Sigma_v^{\mathrm{nd}}$. Corresponding arc costs $\ell(v, v')$ are normally two to account for letter a being added twice in the final palindrome, but three in case only singleton letters remain in $S[p^{\mathrm{L},v'}, p^{\mathrm{R},v'}]$ to additionally account for a respective final middle letter. For each obtained new state it is checked if a respective node exists already in N. If this is not the case, a corresponding new node is added to N and Q. Otherwise, the existing node's length-value $l_{v'}$ is updated in case the new path via v represents a new longest partial solution.

Finally, note that all proposed upper bound functions presented in Section 3.2 have the property of being *monotonic* (also called *consistent*), because the upper bound values of child nodes are always at most as high as the upper bound values of their parents. Due to monotonicity we can be sure that no re-expansions of already expanded nodes will be necessary [25].

Deriving the left and the right position vectors and thus the state for each successor v' of a node v is the computationally most expensive step during the process of *expanding a node*

Algorithm 1 A* Search for the LCPS Problem

Input: an instance (S, Σ)
Output: an optimal LCPS solution
create root node $r = ((1, \ldots, 1), (|s_1|, \ldots |s_m|))$ with $l_r = 0$
add r to the initially empty node set N and the set of open nodes Q
loop
    pop a node v with largest priority $f(v)$ from open nodes Q
    determine Σ_v^{nd} from $p^{\mathrm{L},v}$ and $p^{\mathrm{R},v}$
    if $\Sigma_v^{\mathrm{nd}} = \emptyset$ **then**
        // goal node reached
        $s \leftarrow$ partial solution corresponding to a longest path from r to v
        if $S[p^{\mathrm{L},v}, p^{\mathrm{R},v}]$ contains a singleton letter a **then**
            return palindrome $s \cdot a \cdot s^{\mathrm{rev}}$
        else
            return palindrome $s \cdot s^{\mathrm{rev}}$
        end if
    else
        // consider successors
        for $a \in \Sigma_v^{\mathrm{nd}}$ **do**
            compute node v' that results from appending a at node v
            if $S[p^{\mathrm{L},v'}, p^{\mathrm{R},v'}]$ contains only singleton letters **then**
                $\ell(v, v') \leftarrow 3$
            else
                $\ell(v, v') \leftarrow 2$
            end if
            if $v' \notin N$ **then**
                add new node v' with $l_{v'} = l_v + \ell(v, v')$ to N and Q
            else if $l_v + \ell(v, v') > l_{v'}$ **then** // a better path to v'
                $l_{v'} \leftarrow l_v + \ell(v, v')$
                update entry for v' in Q with new priority value $f(v') = l_{v'} + \mathrm{UB}(v')$
            end if
        end for
    end if
end loop

v. More specifically, for each string s_i, $i = 1, \ldots, m$, $p_i^{\mathrm{L},v'}$ (respectively $p_i^{\mathrm{R},v'}$) of a child node v' of v, given by expanding a valid partial solution represented by v by means of a letter $a \in \Sigma$, is determined as the position of the first occurrence of a in string $s_i[p_i^{\mathrm{L},v}, p_i^{\mathrm{R},v}]$ (respectively, the last occurrence of a in string $s_i[p_i^{\mathrm{L},v}, p_i^{\mathrm{R},v}]$). Finding these positions can be done efficiently by establishing during pre-processing a successor (predecessor) data structure as follows. The successor structure contains a value $\mathrm{Succ}[i, j, a]$ for each $i = 1, \ldots, m$, $j = 1, \ldots, n$, and $a \in \Sigma$ corresponding to the minimal position $p > j$ such that $s_i[p] = a$. If there is no such position, the special value $n + 1$ is used. The predecessor structure stores a value $\mathrm{Pred}[i, j, a]$ for all $i = 1, \ldots, m$, $j = 1, \ldots, n$, and $a \in \Sigma$ corresponding to the maximal position $p < j$ such that $s_i[p] = a$. In case of no such position, zero is used here. Both structures can be built in $O(mn|\Sigma|)$ time, and by using it all successors of a node can be derived in $O(m|\Sigma|)$ time.

Remember that in Section 3.1 we introduced a dominance relation between two letters

a and b for extending a node v. Dominated letters are clearly sub-optimal choices and therefore we avoid their further consideration. This pruning according to dominance may be generalized: When a new state $S[p^{\mathrm{L},v}, p^{\mathrm{R},v}]$ is obtained, we can check if there is some other already considered node $v' \in N$ for which $p_i^{\mathrm{L},v'} \leq p_i^{\mathrm{L},v} \wedge p_i^{\mathrm{R},v} \leq p_i^{\mathrm{R},v'}\ \forall i = 1, \ldots, m$ holds and for which $l_{v'} \geq l_v$. Such a node v' would dominate v in the sense that v cannot lead to any better solution, and consequently, we can omit node v and the arc leading to it from any further consideration. Unfortunately, this generalized dominance check requires $O(|N|m)$ time. In practical experiments with our A* search, it turned out that the introduced overhead is substantial and can be dramatic especially for longer runs when $|N|$ becomes large. Usually the gained reductions in the number of avoided nodes cannot outweigh this disadvantage. Therefore, we stay here with the simple dominance checks among the successors of a node.

In the rest of the paper we investigate variants of search algorithms that build upon the presented A* search. In particular, we aim for a better anytime behavior, i.e., providing a first heuristic solution soon and continuously improving it over time.

4. Approximating the Expected Length of a LCPS for Random Strings

In prior work on beam search algorithms for the LCS problem, Mousavi and Tabataba [39] noticed that the LCS problem instances generally used in the related literature have properties close to those of random instances. That is, the probability for a letter $a \in \Sigma$ to appear at the i-th position of any of the input strings is (nearly) equal for all letters from Σ. Based on this observation they derived a heuristic function for guiding their beam search approach, which led to a new state-of-the-art performance at that time. In other words, they discovered that their heuristic function guides beam search much better than the available upper bound functions. However, since their heuristic function is not a proper upper bound, it cannot be used to prove optimality. In the following we first revisit the heuristic function from [39] in the context of the LCPS problem and then build upon it by deriving an approximation for the expected length of a LCPS for random strings. This function will later be used in combination with the previous upper bound in order to be able to find good heuristic solutions quickly (due to using the heuristic function) but to possibly prove optimality as well.

Mousavi and Tabataba came up with the following recursion which calculates the probability that a specific string s of length k is a subsequence of a string t of length q, where t is generated uniformly at random: $\Pr[s \prec t] = \mathrm{P}(|s|, |t|)$ with

$$\mathrm{P}(k,q) = \begin{cases} 0 & \text{if } k > q \\ 1 & \text{if } k = 0 \\ \frac{1}{|\Sigma|} \cdot \mathrm{P}(k-1, q-1) + \frac{|\Sigma|-1}{|\Sigma|} \cdot \mathrm{P}(k, q-1) & \text{else.} \end{cases} \tag{6}$$

All probabilities $\mathrm{P}(k,q)$ for $k, p = 0, \ldots, n$ can be calculated and stored in $O(n^2)$ time during preprocessing. Let us now consider a node $v \in V$ from our state graph. Given P, we can calculate the probability that the remaining subproblem $S[p^{\mathrm{L},v}, p^{\mathrm{R},v}]$ contains a specific palindrome of length k by

$$\Pr(k, S[p^{\mathrm{L},v}, p^{\mathrm{R},v}]) = \prod_{i=1}^{m} \mathrm{P}(k, |s_i[p_i^{\mathrm{L},v}, p_i^{\mathrm{R},v}]|) = \prod_{i=1}^{m} \mathrm{P}(k, p_i^{\mathrm{R},v} - p_i^{\mathrm{L},v} + 1). \tag{7}$$

In fact, Mousavi and Tabataba directly used these probabilities as heuristic function $h(v)$ to rank in their beam search all successor nodes at a current level for selecting the most promising ones and filtering out the rest. Obviously, higher values of $h(v)$ are preferred in this ranking. For parameter k, they used at each level of the beam search the same value, which they determined from the set of all nodes to be compared (V_{ext}) simply by

$$k := \min_{v \in V_{\text{ext}},\ i=1,\ldots,m} \left(p_i^{\mathrm{R},v} - p_i^{\mathrm{L},v} + 1 \right).$$

While this approach can be meaningful in the context of a standard beam search, it cannot be easily adopted in a more general search like A*, where at each iteration a node has to be evaluated efficiently in relation to the potentially huge set of all open nodes with different distances to the root. Most importantly, there will not be a single meaningful value for k, and it would not make sense to repeatedly re-evaluate all open nodes for changing values of k.

Instead, we strive to approximate the real expected length of a LCPS for a set of strings $S = \{s_1, \ldots, s_m\}$ under the assumption that the input strings are mutually independent uniform random strings. Note that we will then use this approximation to evaluate a subproblem $S[p^{\mathrm{L},v}, p^{\mathrm{R},v}]$ represented by a node $v \in V$ in an alternative way. Concerning related work, Chvátal and Sankoff [13] considered the expected length of the LCS of two random sequences of length n over an alphabet Σ. The authors derived explicit formulas for small n, and lower and upper bounds for the so-called Chvátal–Sankoff constants $\gamma_{|\Sigma|}$, for $|\Sigma| > 1$, defined as the limits of the ratios between the expected length and n, as n increases towards infinity. These constants are still not known so far, but Dančik and Paterson [14] improved the upper bounds for γ_2 based on the theory of Markov chains. Dixon [16] considered the case for two binary strings of different lengths. He conjectured an approximate upper bound for the expected length under certain additional conditions. Znamenskij [54] came up with the hypothesis of an accurate formula for the expected length for the case of two random strings of different length and an arbitrary alphabet. An empirical indication for the correctness of this hypothesis is given, and numerical experiments showed the precision of the formula with a high accuracy. A proof for Sankoff and Mainville's conjecture about the convergence of $\gamma_{|\Sigma|}$ as $|\Sigma|$ tends toward infinity can be found in [33]. We are not aware of any previous work on the expected length of a LCS for more than two random strings or of a LCPS for random strings.

Let X be the random variable corresponding to the length of a LCPS for a set S of randomly generated input strings. Clearly, X can never be larger than the length of the shortest string in S, which we denote by $l_{\max} = \min_{i=1,\ldots,m} |s_i|$. The expected length of an LCPS can be expressed as $\mathbb{E}[X] = \sum_{l=1}^{l_{\max}} l \cdot \Pr[X = l]$ with $\Pr[X = l]$ denoting the probability that this length is l. Furthermore, let $Y_l \in \{0, 1\}$ be the random variable indicating if the strings from S have a common palindromic subsequence of length l, $l \geq 0$. Observe that the existence of a palindromic subsequence of size $l > 1$ always implies the existence of palindromic subsequences of size $l' = 0, \ldots, l-1$, since any palindromic subsequence of length l can be trivially reduced to length $l-1$ by removing an element from the middle. Therefore, it holds that $\Pr[X = l] = \mathbb{E}[Y_l] - \mathbb{E}[Y_{l+1}]$ for $l = 0, \ldots, l_{\max}$, i.e., the probability that there exists a palindromic subsequence of size l but no longer one. We obtain

$$\mathbb{E}[X] = \sum_{l=1}^{l_{\max}} l \cdot (\mathbb{E}[Y_l] - \mathbb{E}[Y_{l+1}]) = \sum_{l=1}^{l_{\max}} \mathbb{E}[Y_l]. \tag{8}$$

In order to approximate $\mathbb{E}[Y_l]$, we first observe that—for an alphabet of size $|\Sigma|$—there are $|\Sigma|^{\lceil l/2 \rceil}$ different palindromes of length l. This is because the first half and the possible middle element can be assigned any letters from Σ and the second half must be equal to the reverted first half. Because of (6), the probability that a specific palindrome s of length l is a subsequence of all strings in S is equal to $\Pr[s \prec S] = \prod_{i=1}^{m} \mathrm{P}(l, |s_i|)$. In the following we make the simplifying assumption that for each palindrome of length l the event of appearing as common subsequence of S is independent of the events of the other palindromes. Clearly, this does not entirely hold in reality and we introduce an error, but it simplifies our considerations to a level with which we can deal further. The probability that S has any common palindromic subsequence of length $l \in \mathbb{N}$ can then be approximately expressed as

$$\tilde{\mathbb{E}}[Y_l] = 1 - (1 - \Pr[s \prec S])^{|\Sigma|^{\lceil l/2 \rceil}} = 1 - \left(1 - \prod_{i=1}^{m} \mathrm{P}(l, |s_i|)\right)^{|\Sigma|^{\lceil l/2 \rceil}}, \tag{9}$$

i.e., the inverse probability of the case that none of the $|\Sigma|^{\lceil l/2 \rceil}$ palindromes of length l is a common subsequence of S. Ultimately, we obtain the approximate expected length of the LCPS

$$\tilde{\mathbb{E}}[X] = l_{\max} - \sum_{l=1}^{l_{\max}} \left(1 - \prod_{i=1}^{m} \mathrm{P}(l, |s_i|)\right)^{|\Sigma|^{\lceil l/2 \rceil}}. \tag{10}$$

To illustrate the error introduced by the assumed independence, we consider the following special cases.

- Let $S = \{s_1\}$ and $l = |s_1|$. At most one of the $|\Sigma|^{\lceil l/2 \rceil}$ different palindromes of length l can be a subsequence of s_1 since s_1 has to correspond to it. Our calculation yields

  $$\tilde{\mathbb{E}}[Y_l] = 1 - (1 - 1/|\Sigma|^l)^{|\Sigma|^{\lceil l/2 \rceil}},$$

  while the correct value corresponds to the probability that s_1 is palindromic, which is

  $$\tilde{\mathbb{E}}[Y_l] = \frac{|\Sigma|^{\lceil l/2 \rceil}}{|\Sigma|^l} = 1/|\Sigma|^{\lfloor l/2 \rfloor}.$$

- Let $S = \{s_1\}$ with $|s_1| \geq 1$ and $l = 1$. We have that $\tilde{\mathbb{E}}[Y_l] = 1 - (1 - 1)^{|\Sigma|} = 1$, and this corresponds to the correct probability for $\mathrm{E}[Y_l]$.

- Let $S = \{s_1, \ldots, s_m\}$ with $l = |s_1| = \ldots = |s_m|$.

  $$\tilde{\mathbb{E}}[Y_l] = 1 - \left(1 - \frac{1}{|\Sigma|^{lm}}\right)^{|\Sigma|^{\lceil l/2 \rceil}}$$

  while

  $$\mathbb{E}[Y_l] = \frac{1}{|\Sigma|^{l(m-1)+\lfloor l/2 \rfloor}} = \frac{1}{|\Sigma|^{\lfloor l \cdot (2m-1)/2 \rfloor}}.$$

4.1. Numerically Stable and Efficient Calculation of $\tilde{\mathbb{E}}[X]$

Calculating $\tilde{\mathbb{E}}[X]$ directly according to equation (10) is in practice hardly possible due to the extremely large power values one obtains for not so small string lengths l. Classical double precision floating point arithmetic is insufficient for strings with already more than about 40 letters. However, the term from the sum on the right-hand side of (10) can be decomposed to

$$\left(1-\prod_{i=1}^{m}\mathrm{P}(l,|s_i|)\right)^{|\Sigma|^{\lceil l/2\rceil}} = \left(\underbrace{\left(\dots\left(1-\prod_{i=1}^{m}\mathrm{P}(l,|s_i|)\right)^{|\Sigma|^p}\cdots\right)^{|\Sigma|^p}}_{\lfloor\lceil l/2\rceil/p\rfloor \text{ times}}\right)^{|\Sigma|^{\lceil l/2\rceil \bmod p}} \tag{11}$$

for $p \in \mathbb{N}_{>0}$. We may use $p = 25$, for example, which will yield small enough values for all intermediate results when using classical double precision floating point arithmetic.

While this decomposition avoids overflows there are other issues when $\prod_{i=1}^{m} P(l,|s_i|)$ becomes small due to cancellation effects in the limited precision of classical floating point arithmetic. In our implementation, we specifically check if $\prod_{i=1}^{m} P(l,|s_i|) < 10^{-10}$ and handle this case in the following different way.

To ease the further considerations, let us define $x := \prod_{i=1}^{m} P(l,|s_i|)$ and $\alpha := |\Sigma|^{\lceil l/2\rceil}$; we now have to calculate $(1-x)^\alpha$. The numerically problematic situation occurs when x is close to zero and α is large. To resolve this issue, we make use of the fact that $\ln(1-x)/x$ can be well approximated for small x by taking the first two terms of the Taylor series expansion at $x = 0$, which is $-1 - x/2 - o(x)$. This yields

$$(1-x)^\alpha = e^{\alpha\cdot\ln(1-x)} = e^{\alpha x\cdot\frac{\ln(1-x)}{x}} \approx e^{\alpha x\cdot(-1-\frac{x}{2})}. \tag{12}$$

Here, however, the product αx may still be numerically problematic to calculate, in fact already the calculation of $\alpha = |\Sigma|^{\lceil l/2\rceil}$ alone may already exceed the limits of a classical double precision floating point arithmetic. We therefore rewrite

$$\alpha x = e^{\ln(\alpha x)} = e^{\lceil\frac{l}{2}\rceil\cdot\ln|\Sigma|+\ln(x)} \tag{13}$$

and check if already $\lceil\frac{l}{2}\rceil\cdot\ln|\Sigma|+\ln(x)$ is so large that the overall result will be negligibly small. More specifically, in our implementation we check for $\lceil\frac{l}{2}\rceil\cdot\ln|\Sigma|+\ln(x) > 300$, in which case $(1-x)^\alpha < e^{-e^{300}}$, and we therefore return zero as result.

Otherwise, we determine $\tilde{\alpha} := \alpha x\cdot(-1-\frac{x}{2})$. If $\tilde{\alpha}$ is close to zero (realized in our implementation by $|\tilde{\alpha}| < 10^{-15}\}$) we can use the fact that $1 - e^{\tilde{\alpha}} \approx -\tilde{\alpha}$ and consequently approximate (12) well by returning $1+\tilde{\alpha}$.

Last but not least, in the remaining case we consider $\tilde{\alpha}$ to be in a reasonable range so that we can calculate $e^{\tilde{\alpha}}$ in a numerically stable way and return this value as approximate result of $(1-x)^\alpha$.

Summarizing the above, whenever $x \le 10^{-10}$ we calculate

$$(1-x)^\alpha \approx \begin{cases} 0 & \text{if } \lceil\frac{l}{2}\rceil\cdot\ln|\Sigma|+\ln(x) > 300 \\ 1+\tilde{\alpha} & \text{if } \lceil\frac{l}{2}\rceil\cdot\ln|\Sigma|+\ln(x) \le 300 \wedge |\tilde{\alpha}| < 10^{-15} \\ e^{\tilde{\alpha}} & \text{else,} \end{cases} \tag{14}$$

and for $x \leq 10^{-10}$, we can calculate $(1-x)^{\alpha}$ safely by applying the decomposition rule (11).

In order to determine the approximate expected LCPS length $\tilde{\mathbb{E}}[X]$ according to (10), the terms $(1 - \tilde{\mathbb{E}}[Y_l])$ must be calculated for $l = 1, \ldots, l_{\max}$, which requires $O(mn)$ time. In the case of larger n, this would be inefficient and be a bottleneck of our whole approach to solve the LCPS problem. In order to reduce this complexity, we interpolate the values for most l by means of a divide-and-conquer scheme. This approach exploits the fact that the sequence of values $\{\tilde{\mathbb{E}}[Y_l]\}_{l=1,\ldots,l_{\max}}$ is monotonically decreasing with values in the interval $[0, 1]$. The approach starts by defining the artificial border values $\tilde{\mathbb{E}}[Y_0] := 1$ and $\tilde{\mathbb{E}}[Y_{l_{\max}+1}] := 0$ and setting $l = 0$ and $l' = l_{\max} + 1$. Then it applies the following recursive principle: If $l + 1 < l'$, we know the values for $\tilde{\mathbb{E}}[Y_l]$ and $\tilde{\mathbb{E}}[Y_{l'}]$ but not yet some lying inbetween. In this case, if $\tilde{\mathbb{E}}[Y_l] - \tilde{\mathbb{E}}[Y_{l'}] \leq \varepsilon$ for some sufficiently small ε ($\varepsilon = 10^{-6}$ in our implementation), we determine $\tilde{\mathbb{E}}[Y_{l''}]$ for $l'' = l + 1, \ldots, l' - 1$ by linear interpolation between $\tilde{\mathbb{E}}[Y_l]$ and $\tilde{\mathbb{E}}[Y_{l'}]$. Otherwise, we calculate the middle value $\tilde{\mathbb{E}}[Y_{\lceil (l+l')/2 \rceil}]$ according to our approximation above and recursively call the procedure for $\{\tilde{\mathbb{E}}[Y_l], \ldots, \tilde{\mathbb{E}}[Y_{\lceil (l+l')/2 \rceil}]\}$ and $\{\tilde{\mathbb{E}}[Y_{\lceil (l+l')/2 \rceil}], \ldots, \tilde{\mathbb{E}}[Y_{l'}]\}$.

Finally, recall that each node $v \in N$ of our state graph represents a subproblem $S[p^{\mathrm{L},v}, p^{\mathrm{R},v}]$, and we can determine corresponding approximate expected LCPS lengths according to (10) and the above described stable and efficient calculation method for these:

$$\mathrm{EX}(v) = \sum_{l=1}^{l_{\max}} 1 - \left(1 - \prod_{i=1}^{m} \mathrm{P}(l, p_i^{\mathrm{R},v} - p_i^{\mathrm{L},v} + 1)\right)^{|\Sigma|^{\left\lceil \frac{l}{2} \right\rceil}} . \tag{15}$$

Note that $\mathrm{EX}(v)$, in contrast to the upper bound functions from the previous section, does not possess the property of being admissible in the context of A* search.

5. A Novel Anytime Algorithm for the LCPS

Classical A* search is targeted towards finding a proven optimal solution in the least number of expanded nodes, but in general it yields no meaningful or particularly promising heuristic solution before it terminates when the target node is selected for expansion. In [17], we improved the anytime behavior of our A* algorithm for the LCPS in the following way. The hybrid A*+BS algorithm embeds a standard beam search into the A* search framework such that, after performing a number of regular A* iterations, the search strategy repeatedly is switched to a BS starting from the node with the highest priority value from Q. Note that the new nodes discovered by the BS applications are also incorporated into set N and the priority queue Q, and expanded nodes are removed from Q. However, while this A*+BS version from [17] exclusively considered new nodes, the A*+BS version that we consider for the experimental evaluation of this paper also considers already encountered nodes—that is, nodes which are already present in Q—as candidates for the beam of the next step in a BS run. We experimentally confirmed that this change results in a significant performance improvement.

Nevertheless, after an intensive study of the A*+BS algorithm, the following shortcomings of A*+BS were detected:

1. Even though our new upper bound UB is tighter than the UB_1 bound from [17], it is still far from being a tight bound. Therefore, in case of larger instances the nodes with the highest priority values in Q—that is, those nodes that are used to initialize the BS runs—are generally close to the root node of the search tree and the chance that they are promising starting nodes for BS is rather low.
2. The embedded BS does not ensure that the most promising nodes from each level of the state graph are included in the beam corresponding to this level, as only extensions of the starting node of each BS application are considered.

It was observed that these problems lead to the following behavior. The solution quality of A*+BS at a certain time can often be significantly exceeded by a single BS run whose beam width is chosen such that its computation time is comparable. While the pure BS is no anytime algorithm and does not provide any lower bound, this observation nevertheless indicates room for improvement. Moreover, applying a rather large beam width in A*+BS leads to finding good heuristic solutions early, but afterwards these solutions are hardly improved. On the other side, applying a rather small beam width leads to initial heuristic solutions of lower quality which are improved over time, without, however, reaching the final solution quality of A*+BS when using a rather large beam width.

Therefore, we propose here the following potential improvements of A*+BS. First, we exchange the standard BS component with a beam search version known as *Anytime Column Search* (ACS), proposed by Vadlamudi et al. [48]. The most interesting feature of ACS is that it expands the most promising open nodes at each level of the state graph. Moreover, we exchange the use of the upper bound for guiding ACS with the approximation of the expected length of a LCPS as derived in the last section.

When consulting the related literature, we can find several attempts at improving the anytime performance of A* algorithms as well as several attempts at using beam search related algorithms in an anytime fashion. For completeness, we provide a short summary of these approaches before outlining our A*+ACS approach.

Concerning A* approaches, Hansen et al. [24] and Hansen and Zhou [23] proposed *Anytime Weighted* A* which makes use of weighting the heuristic function by a constant factor $w > 0$, i.e, $f(v) := g(v) + w \times h(v)$, in order to achieve a quick convergence to a heuristic and usually sub-optimal solution. The authors showed that an obtained solution is a w-approximation if heuristic h is admissible. A generalization of this idea, called *Anytime Restricted* A* (ARA*), was presented in [35]. The main idea is to exchange the constant weight w of Anytime Weighted A* with a linearly decreasing sequence of weights, one for each algorithm iteration. The value of the initial weight has—in general—a significant impact on the convergence of ARA*. Due to the fact that choosing appropriate weights in ARA* is a problem specific task, Berg et al. [49] proposed *Anytime Non-Parametric* A* (ANA*), eliminating the ad-hoc parameters involved in ARA* by adapting the greediness of the search as path quality improves. Aine et al. [3] proposed *Anytime Window* A* (AWA*), in which the nodes from the open list within one of the levels of depth from a range defined by the window size are expanded, converging to a sub-optimal solution at each iteration. The window size is adapted at each iteration to produce improved solutions. A memory-bounded version of AWA* was proposed by Vadlamudi et al. [46].

On the other side, the literature offers beam search based algorithms, extended to be anytime algorithms. Most of these algorithms work on the principle of initially performing a single beam search to get reasonably good, suboptimal (heuristic) solutions, and

then initializing the beam of subsequent BS runs with nodes which were pruned in previous iterations (see [52, 53], for example). However, the literature does not provide a work offering a comprehensive comparison of these algorithms. Recently, Vadlamudi [47] presented *Anytime Pack Search* (APS), showing that it outperforms anytime algorithms such as ANA* and AWA*. This study considers problems from three different domains. APS maintains a global priority queue Q. At each iteration, the β most promising nodes from Q are picked and used as initial beam for the current run of beam search. Note that the nodes for the initial beam may be from different levels of the search tree. When performing the beam search at each iteration, the pruned nodes are being added to Q. In the same paper, the authors proposed a version of APS, called *Anytime Progressive Pack Search* (APPS), which aims at improving the anytime behavior of APS. This is done by increasing the size of the initial beam (β) dynamically during the search process by means of a step size parameter each time when no better solution has been found. Otherwise, the beam size is reset to the initial value β. Experimental results show that APPS can indeed achieve a better anytime behavior than APS.

5.1. A*+ACS Algorithm

As indicated before, ACS is an iterative algorithm that, at each major iteration, expands nodes with the highest priority values at each level of the state graph [48]. In order to do so, the algorithm interprets the so far investigated parts of the state graph in a layered fashion, where level $j \geq 0$ contains any node $v \in N$ having depth j, i.e., can be reached from the root node r via j so far known arcs but not more. In our context of the LCPS, level j thus contains the nodes for which corresponding partial solutions with up to j letters are known. If a node is updated during the search process because a longer partial solution—represented by a longer path to this node—is found, the node will change to the respective higher level. In contrast to the classical A* search, ACS maintains an individual priority list Q_l of open nodes for each level $j = 0, \ldots, j_{\max}$, where $j_{\max}$ is an upper bound for the depth of nodes; in our implementation we chose $j_{\max} = \lfloor \mathrm{UB}(r)/2 \rfloor$. Initially, Q_0 contains the root node and all other priority queues are empty. Each iteration of ACS considers all the levels $j = 0, \ldots, j_{\max}$ with non-empty queues Q_i in turn and expands from each β nodes (or less if Q_j becomes empty). ACS terminates with an optimal solution only when all priority lists become empty. However, ACS finds at least one complete solution at each major iteration, which favors our goal of producing heuristic solutions as soon as possible.

We now embed this ACS in our A* by interleaving classical A* iterations with ACS iterations. A pseudocode of this A*+ACS is presented in Algorithms 2 and 3. The main algorithmic framework is that of A*. However, initially and after every batch of $\delta > 0$ iterations of A*, the algorithm executes one iteration of ACS. Algorithm 2 maintains in s_{best} the so far best found complete solution. Each A* iteration still expands a node v from the global open list Q having the largest priority value $f(v)$. In this way the whole approach maintains the completeness property of the classical A* search and $\max_{v \in Q} f(v)$ provided by the top element of Q always is an upper bound for the optimum solution value. In contrast, the level-wise priority queues Q_j, $j = 0, \ldots, j_{\max}$, of ACS make use of the approximate expected value function EX, cf. (15), for prioritizing the nodes. As we already pointed out, this guidance function is not a bound and therefore cannot be used for proving optimality. However, we expect it to lead the construction of heuristic solutions in substantially better ways. Note that changes in Q (removals and additions) must be

Algorithm 2 A*+ACS for the LCPS Problem

Input: an instance (S, Σ)
Output: best found LCPS solution s_{best}
Parameters: ACS column width β, number of A* iterations inbetween ACS δ
create root node $r = ((1, \ldots, 1), (|s_1|, \ldots |s_m|))$ with $l_r = 0$
add r to the initially empty node set N and the global set of open nodes Q
initialize per-level priority queues $Q_0 = \{r\}$ and $Q_i = \emptyset$, $j = 1, \ldots, j_{\max}$
optimal $\leftarrow$ **false**
$s_{\text{best}} \leftarrow \varepsilon$
loop
    // perform an ACS iteration of width β
    for $j \leftarrow 0, \ldots, j_{\max}$ **do**
        $b \leftarrow 0$
        while $Q_j \neq \emptyset \wedge b < \beta$ **do**
            // select and expand next node at level j
            pop a node v with the largest EX(v)-value from Q_j
            remove v also from Q
            ExpandNode(v)
            if *optimal* $\vee$ *time or memory limit reached* **then**
                return so far best solution s_{best}
            end if
            $b \leftarrow b + 1$
        end while
    end for
    // perform δ normal A* iteration
    loop δ times
        pop a node v with largest priority $f(v)$ from Q
        remove v also from $Q_{\lfloor l_v/2 \rfloor}$
        ExpandNode(v)
        if *optimal* $\vee$ *time or memory limit reached* **then**
            return so far best solution s_{best}
        end if
    end loop
end loop

reflected by corresponding changes in the priority queues Q_j, and vice versa. To do this efficiently, we augment in our implementation the heap data structures for the priority queues by corresponding hash tables, which enable a direct lookup of the priority queue entries for given nodes. The actual expansion of a node, which is identical for the ACS as well as the A* iterations, is separately shown in Algorithm 3. It follows the principles already known from Algorithm 1. When a goal node is reached it is checked if it yields a new best solution and s_{best} is updated in this case. At its end, Algorithm 3 always checks if the length of the so far best solution is larger than or equal to the current maximum f-value of Q, in which case the flag *optimal* is set to true and the main algorithm terminates with the proven optimal solution s_{best}. Moreover, A*+ACS also terminates when reaching a specified time or memory limit, in which case it returns the best complete solution found up to this point.

In summary, the ACS iterations augment the classical A* iterations in order to find promis-

Algorithm 3 ExpandNode(v)

Input: Node v to be expanded
Uses/updates: V, N, Q, $Q_1, \ldots, Q_{l_{\max}}$, s_{best}, *optimal*
determine Σ_v^{nd} from $p^{\text{L},v}$ and $p^{\text{R},v}$
if $\Sigma_v^{\text{nd}} = \emptyset$ **then**
    // goal node reached
    if $|s_{\text{best}}| < f(v)$ **then**
        $s \leftarrow$ partial solution corresponding to a longest path from r to v
        if $S[p^{\text{L},v}, p^{\text{R},v}]$ contains a singleton letter a **then**
            $s_{\text{best}} \leftarrow s \cdot a \cdot s^{\text{rev}}$
        else
            $s_{\text{best}} \leftarrow s \cdot s^{\text{rev}}$
        end if
    end if
else
    // consider successors
    for $a \in \Sigma_v^{\text{nd}}$ **do**
        compute node v' that results from appending a at node v
        if $S[p^{\text{L},v'}, p^{\text{R},v'}]$ contains only singleton letters **then**
            $\ell(v, v') \leftarrow 3$
        else
            $\ell(v, v') \leftarrow 2$
        end if
        if $v' \notin N$ **then**
            calculate $\text{EX}(v')$ and $f(v')$
            add new node v' with $l_{v'} = l_v + \ell(v, v')$ to N, Q, and $Q_{\lfloor l_{v'}/2 \rfloor}$
        else if $l_{v'} < l_v + \ell(v, v')$ **then** // a better path to v'
            remove v' from $Q_{\lfloor l_{v'}/2 \rfloor}$
            $l_{v'} \leftarrow l_v + \ell(v, v')$
            update entry for v' in Q with new $f(v')$
            add v' in $Q_{\lfloor l_{v'}/2 \rfloor}$ with $E(v')$
        end if
    end for
end if
if $|s_{\text{best}}| \geq$ maximum f-value of nodes in Q **then**
    optimal $\leftarrow$ **true**
end if

ing heuristic solutions soon and possibly also improve them continuously over time. This counter-balances the pure best-first strategy of A*. The number of A* iterations δ between the executions of the ACS iterations as well as ACS's width parameter β control the balance between providing good heuristic solutions and improving the upper bound over time.

6. Experimental Results

All proposed algorithms as well as algorithms considered in the following for comparison were implemented in C++ using GCC 4.7.3. All experiments were performed on a cluster of machines with Intel Xeon E5649 CPUs with 2.53 GHz and a memory limit of 15GB in single-threaded mode. The maximum computation time allowed for each run was limited to 15 minutes, i.e., 900 seconds.

The following algorithms are considered in this section: (1) A*+BS is the A*/beam search hybrid proposed in [17] but improved by the advanced upper bound calculation from Section 3, (2) the A*+ACS algorithm proposed in this work, (3) the anytime-A* variants APS and APPS from [47] which we implemented for comparison reasons, and (4) a stand-alone ACS algorithm—henceforth labeled ACS-ub—using the upper bound UB for prioritizing the nodes. This last algorithm targeted towards primarily getting good heuristic solutions is studied for comparison purposes in order to get an impression on the impact of the novel heuristic guidance function EX() from Equation (15) in our A*+ACS.

We also would like to point out that, during experimentation, we noticed that the original APPS performed significantly worse with respect to the obtained solution quality when the beam width was set back to the initial value each time a new incumbent was found. Therefore, our implementation applies a purely progressive increase of the beam width after each BS run.

All considered algorithms will be evaluated by the obtained solution quality and by the *percentage gap*, which is calculated at any time point $t > 0$ as $gap(t) := \frac{f^*(t) - s_{\text{best}}(t)}{f^*(t)} \cdot 100\%$, where $s_{\text{best}}\,(t)$ denotes the value of the best found solution and $f^*(t)$ the f-value of top node of Q (representing an upper bound) at time t. In case of ACS-ub, this upper bound is calculated as $f^*(t) := \max_{i=0,\ldots,j_{\max}}\{f(u_i) | \, Q_i \neq \emptyset \ \wedge u_i \text{ is the top node of } Q_i \text{ at the time } t\}$.

6.1. Benchmark Instances

We used a set of benchmark instances that were originally provided in [7] for the LCS problem. This instance set consists of ten randomly generated instances for each combination of the number of input strings $m \in \{10, 50, 100, 150, 200\}$, the length of input strings $n \in \{100, 500, 1000\}$, and the alphabet size $|\Sigma| \in \{4, 12, 20\}$. This makes a total of 450 problem instances. In general, the results of our algorithms will be provided as averages over the ten instances of each combination. In order to compare the algorithms concerning the 2–LCPS problem, we generated a new set of larger instances by using the instance generator from [7]. More specifically, for each combination of $|\Sigma| \in \{4, 12, 20\}$ and $n \in \{100, 200, 300, 400, 500\}$ we created ten instances yielding a total of 150 2–LCPS instances. These benchmark instances are provided at `https://www.ac.tuwien.ac.at/research/problem-instances/LCPS`.

6.2. Tuning of the Algorithms

In order to ensure a fair comparison, we employed the tuning tool irace [37] for deriving well-working parameter values for all five considered algorithms. A*+BS has the following parameters: (δ) the number of A* iterations performed after each BS run, (β) the beam width, and (k) the parameter for the k-norm used in tie-breaking. APS has the following

Table 1: Tuning results concerning solution quality.

(a) A*+BS

$\|\Sigma\|$	δ	β	k
4	1	10000	0.5
12	10	2000	0.2
20	20	1000	0.1

(b) APS

$\|\Sigma\|$	*pack*	k
4	10000	0.1
12	10000	0.2
20	5000	0.1

(c) APPS

$\|\Sigma\|$	*pack*	*step*	k
4	10000	10	0.2
12	10000	10	0.2
20	5000	5	0.1

(d) ACS-ub

$\|\Sigma\|$	β	k
4	20	0.2
12	20	0.5
20	100	1.0

(e) A*+ACS

$\|\Sigma\|$	δ	β	k
4	100	10	1.0
12	50	10	1.0
20	100	20	0.1

parameters: (*pack*) the number of nodes taken from the top of the queue in order to form the initial beam, and (k). APPS has the same parameters as APS and in addition (*step*) the amount of increase applied to *pack* after each BS run. Next, ACS-ub involves parameter (β), which is the number of expansions at each level of the state graph, and (k). Finally, A*+ACS has the parameters: (δ) the number of A* iterations performed after applying an iteration of ACS, (β) the number of expansion allowed at the same level of ACS, and (k).

The irace tool was applied separately for each algorithm and for each alphabet size. Analyzing preliminary experiments, we found that the size of the alphabet has more influence on the behavior of the algorithms than the lengths of the input strings and their number. In order to obtain tuning instances, we generated for each $|\Sigma|$ one random instance for each combination of m and n. This makes a total of 15 tuning instances for each alphabet size, and 45 tuning instances in total. The tuning process for each alphabet size was given a budget of 1000 runs and each run was limited by a run time limit of 900 seconds and a memory limit of 15 GB.

6.2.1. Tuning for Solution Quality

The first set of tuning experiments was aimed at tuning the algorithm performance with respect to solution quality, that is, for obtaining the best possible solution quality at the end of a run. In the following we present the parameter value domains used during tuning as well as the best configurations for each algorithm as determined by irace. Note that meaningful ranges for the domains were determined by preliminary experiments.

For parameter k we considered {0.1, 0.2, 0.5, 1.0, 2.0} for all algorithms. The domain of parameter β in A*+BS and parameter *pack* in APS and APPS was {1, 50, 100, 500, 1000, 2000, 5000, 10000, 20000}. In contrast, in the context of ACS-ub and A*+ ACS parameter β was given domain $\{1, 5, 10, 20, 50, 100\}$. Finally, we considered $\delta \in \{1, 5, 10, 20, 50, 100, 1000\}$ for both A*+BS and A*+ ACS, and *step* $\in \{1, 5, 10, 50, 100, 200, 500\}$ for APPS. The best configurations as determined by irace are provided in Table 1.

Table 2: Tuning results concerning small gaps.

(a) A*+BS

$\|\Sigma\|$	δ	β	k
4	20000	500	1.0
12	10000	1000	1.0
20	10000	500	0.5

(b) APS

$\|\Sigma\|$	*pack*	k
4	20000	1.0
12	20000	1.0
20	10000	1.0

(c) APPS

$\|\Sigma\|$	*pack*	*step*	k
4	20000	500	1.0
12	10000	1000	1.0
20	20000	500	1.0

(d) ACS-ub

$\|\Sigma\|$	β	k
4	10	0.2
12	1	0.1
20	1	0.5

(e) A*+ACS

$\|\Sigma\|$	δ	β	k
4	5000	20	1.0
12	10000	10	1.0
20	5000	10	1.0

6.2.2. Tuning for Small Gaps

The tuning experiments from the previous section were repeated with the aim of obtaining small gaps, thus, considering in addition to the final solution quality also the respective upper bounds. Naturally we expect for this case other parameter settings to be ideal, in particular those putting more emphasize on classical A* search iterations. The parameter domains were chosen as in the previous subsection, with the exception of the δ parameter in the case of A*+BS and A*+ACS and the *step* parameter in APPS. These were chosen as $\delta \in \{1, 100, 500, 1000, 5000, 10000, 20000, 50000\}$ and $step \in \{1, 10, 50, 100, 500, 1000, 5000\}$. The best configurations as determined by irace are provided in Table 2. Indeed, it can be observed that the resulting values in particular for parameter δ, the number of classical A* iterations, increase significantly when tuning for small gaps.

6.3. Numerical Results and Comparison

Table 3 shows average results of the algorithms over all instance groups with the parameter settings obtained by tuning for solution quality, while Table 4 shows the results with the settings obtained when targeting small gaps. Note that we excluded the results of APPS here as it turned out that they are very similar to those of APS.

Each of these tables consists of three sub-tables, one per alphabet size, and they have the following format. The first two columns indicate the type of problem instances considered in terms of n and m. Remember that the considered benchmark set consists of ten problem instances per combination of $|\Sigma|$, n and m. Consequently, each table row provides the results of the four considered algorithms (A*+BS, APS, ACS-ub, and A*+ACS) as averages over the ten respective instances. For each algorithm, column $\overline{|s|}$ lists the average final solution quality, column $\overline{\text{t}_{\text{best}}}$[s] the average time (in seconds) at which the best solution of a run was found, column $\overline{\text{t}}$[s] the overall average runtime, and, finally, column $\overline{\text{gap}}$[%] the average gap in percent. Note that the overall run time of an algorithm can only be smaller than 900 seconds (the run time limit) when a proven optimal solution is found, or in case the memory limit of 15 GB is reached before the run time limit. The former happens for all algorithms in the context of all instances with $n = 100$, thus all considered algorithms are able to prove optimality for all these instances. The latter happens, for example, in

Table 3: Average results of the algorithms when tuned for solution quality.

(a) $|\Sigma| = 4$.

m	n	A*+BS				APS				ACS-ub				A* + ACS			
		$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\bar{t}$[s]	$\overline{gap}$[%]	$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\bar{t}$[s]	$\overline{gap}$[%]	$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\bar{t}$[s]	$\overline{gap}$[%]	$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\bar{t}$[s]	$\overline{gap}$[%]
10	100	**28.9**	1.2	5.3	0.0	**28.9**	1.0	1.8	0.0	**28.9**	< 0.1	2.4	0.0	**28.9**	< 0.1	1.9	0.0
	500	159.9	13.7	368.7	45.4	159.9	9.5	251.0	44.4	161.2	130.1	415.5	43.5	**162.3**	61.2	573.6	42.6
	1000	323.1	41.0	278.8	47.63	323.3	56.3	269.1	47.3	326.2	146.3	301.3	46.6	**330.7**	256.1	532.1	45.7
50	100	**21.8**	1.3	1.3	0.0	**21.8**	1.0	1.0	0.0	**21.8**	< 0.1	0.9	0.0	**21.8**	< 0.1	0.6	0.0
	500	130.5	25.2	468.2	54.1	130.5	18.2	346.4	53.1	131.3	111.9	490.2	52.5	**132.7**	135.0	555.4	51.4
	1000	267.9	177.8	576.6	56.0	267.8	83.0	507.1	55.7	268.9	314.7	683.1	55.5	**273.0**	92.7	722.1	54.5
100	100	**20.1**	1.7	1.7	0.0	**20.1**	1.2	1.2	0.0	**20.1**	< 0.1	1.2	0.0	**20.1**	< 0.1	0.8	0.0
	500	123.5	53.5	616.2	56.3	123.5	64.3	459.4	55.4	124.1	107.3	651.4	54.9	**125.1**	86.0	688.4	54.0
	1000	254.8	123.0	686.2	58.0	254.8	127.9	563.0	57.7	255.4	324.0	744.1	57.6	**259.8**	199.8	765.8	56.6
150	100	**19.0**	2.1	2.2	0.0	**19.0**	1.4	1.4	0.0	**19.0**	< 0.1	0.9	0.0	**19.0**	< 0.1	0.5	0.0
	500	120.3	128.1	792.7	57.2	120.2	51.8	528.6	56.4	120.7	118.1	736.7	56.1	**121.7**	67.2	723.8	55.1
	1000	249.0	170.3	750.3	58.8	249.0	155.8	580.1	58.6	249.8	337.8	780.8	58.4	**253.3**	192.7	752.6	57.6
200	100	**18.5**	1.8	1.8	0.0	**18.5**	1.7	1.7	0.0	**18.5**	0.1	1.1	0.0	**18.5**	< 0.1	0.7	0.0
	500	118.0	99.1	843.3	57.9	118.0	74.5	601.5	57.2	118.4	216.6	852.4	56.9	**119.5**	25.6	778.4	55.9
	1000	245.0	287.2	859.9	59.4	244.8	196.2	671.8	59.3	245.4	275.0	884.5	59.1	**249.4**	238.2	840.9	58.2

(b) $|\Sigma| = 12$.

m	n	A*+BS				APS				ACS-ub				A* + ACS			
		$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\bar{t}$[s]	$\overline{gap}$[%]	$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\bar{t}$[s]	$\overline{gap}$[%]	$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\bar{t}$[s]	$\overline{gap}$[%]	$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\bar{t}$[s]	$\overline{gap}$[%]
10	100	**9.6**	< 0.1	< 0.1	0.0	**9.6**	< 0.1	< 0.1	0.0	**9.6**	< 0.1	< 0.1	0.0	**9.6**	< 0.1	< 0.1	0.0
	500	61.5	3.5	453.0	65.1	61.8	15.5	371.4	62.0	62.3	77.1	400.9	61.3	**62.4**	26.4	649.7	60.6
	1000	126.7	6.3	291.3	68.2	127.2	28.2	281.9	67.1	128.7	89.1	309.3	66.6	**130.5**	115.5	531.3	65.9
50	100	**5.6**	< 0.1	< 0.1	0.0	**5.6**	< 0.1	< 0.1	0.0	**5.6**	< 0.1	< 0.1	0.0	**5.6**	< 0.1	< 0.1	0.0
	500	43.3	5.7	555.7	73.8	43.6	25.2	548.0	70.6	43.8	41.4	580.8	70.4	**44.3**	96.0	617.5	69.3
	1000	91.1	21.5	568.8	76.4	91.7	92.7	714.8	75.3	92.4	246.2	608.9	75.0	**93.7**	70.2	649.8	74.4
100	100	**4.6**	< 0.1	< 0.1	0.0	**4.6**	< 0.1	< 0.1	0.0	**4.6**	< 0.1	< 0.1	0.0	**4.6**	< 0.1	< 0.1	0.0
	500	39.0	19.2	798.0	75.8	39.0	46.9	829.4	72.8	39.1	35.8	856.4	72.4	**39.6**	88.0	798.4	71.5
	1000	83.9	66.2	879.1	78.0	84.1	147.3	812.9	77.1	84.7	291.2	891.6	76.9	**85.9**	78.6	871.2	76.3
150	100	**3.8**	< 0.1	< 0.1	0.0	**3.8**	< 0.1	< 0.1	0.0	**3.8**	< 0.1	< 0.1	0.0	**3.8**	< 0.1	< 0.1	0.0
	500	37.0	22.6	900.0	76.8	37.1	68.8	900.0	73.9	37.2	52.8	900.0	73.7	**37.6**	22.6	881.7	72.8
	1000	80.3	44.4	900.0	78.8	80.6	209.3	900.0	78.0	81.0	212.7	900.0	77.9	**82.2**	51.9	900.0	77.2
200	100	**3.3**	< 0.1	< 0.1	0.0	**3.3**	< 0.1	< 0.1	0.0	**3.3**	< 0.1	< 0.1	0.0	**3.3**	< 0.1	< 0.1	0.0
	500	35.8	60.6	900.0	77.3	**36.0**	90.8	900.0	74.4	**36.0**	31.8	900.0	75.0	**36.0**	0.6	900.0	73.7
	1000	78.2	152.1	900.0	79.2	78.4	297.7	900.0	78.5	78.7	273.0	900.0	78.4	**80.0**	126.8	900.0	77.8

(c) $|\Sigma| = 20$.

m	n	A*+BS				APS				ACS-ub				A* + ACS			
		$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\bar{t}$[s]	$\overline{gap}$[%]	$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\bar{t}$[s]	$\overline{gap}$[%]	$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\bar{t}$[s]	$\overline{gap}$[%]	$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\bar{t}$[s]	$\overline{gap}$[%]
10	100	**5.4**	< 0.1	< 0.1	0.0	**5.4**	< 0.1	< 0.1	0.0	**5.4**	< 0.1	< 0.1	0.0	**5.4**	< 0.1	< 0.1	0.0
	500	38.7	9.7	351.4	69.5	38.6	7.7	523.2	65.2	**38.9**	5.8	319.1	64.6	**38.9**	1.4	801.9	63.8
	1000	79.7	60.3	353.8	74.2	79.9	20.3	500.2	72.9	80.9	75.6	386.2	72.5	**81.8**	133.4	688.0	71.8
50	100	**2.5**	< 0.1	< 0.1	0.0	**2.5**	< 0.1	< 0.1	0.0	**2.5**	< 0.1	< 0.1	0.0	**2.5**	< 0.1	< 0.1	0.0
	500	25.0	4.3	750.3	76.8	25.0	15.0	900.0	71.2	**25.1**	23.2	740.1	72.2	**25.1**	74.9	858.7	71.0
	1000	54.4	44.9	768.7	81.5	54.6	46.8	888.0	80.2	55.0	101.8	728.0	80.0	**55.6**	42.6	881.8	79.5
100	100	**1.3**	< 0.1	< 0.1	0.0	**1.3**	< 0.1	< 0.1	0.0	**1.3**	< 0.1	< 0.1	0.0	**1.3**	< 0.1	< 0.1	0.0
	500	21.8	8.1	900.0	78.6	21.9	26.0	900.0	73.2	**22.1**	56.3	893.6	73.5	**22.1**	5.9	900.0	72.7
	1000	48.9	62.8	900.0	83.2	48.9	89.9	900.0	82.1	49.1	71.8	899.3	81.9	**50.1**	110.1	900.0	81.2
150	100	**1.1**	< 0.1	< 0.1	0.0	**1.1**	< 0.1	< 0.1	0.0	**1.1**	< 0.1	< 0.1	0.0	**1.1**	< 0.1	< 0.1	0.0
	500	20.6	8.3	900.0	79.4	20.9	46.5	900.0	74.2	**21.0**	43.0	900.0	73.7	**21.0**	8.7	900.0	73.3
	1000	46.3	99.6	900.0	84.0	46.6	202.8	900.0	82.8	46.8	139.4	900.0	82.7	**47.2**	124.9	900.0	82.2
200	100	**1.1**	< 0.1	< 0.1	0.0	**1.1**	< 0.1	< 0.1	0.0	**1.1**	< 0.1	< 0.1	0.0	**1.1**	< 0.1	< 0.1	0.0
	500	19.5	74.0	900.0	80.1	19.5	49.0	900.0	74.8	19.9	65.6	900.0	73.2	**20.0**	91.3	900.0	73.5
	1000	44.8	56.2	900.0	84.4	45.0	190.6	900.0	83.4	45.0	42.7	900.0	83.3	**45.7**	175.4	900.0	82.7

the case of the ten instances with $|\Sigma| = 4$, $m = 10$ and $n = 1000$ (see Table 3a). Note that the best result in each table row is marked bold.

The following main observations can be made concerning these results.

- As mentioned already above, all algorithms are able to solve the problem instances with $n = 100$ to optimality, this within a fraction of a second. Therefore, in what follows, we will focus on the instances with $n \in \{500, 1000\}$.
- A*+ACS outperforms all other algorithms in terms of solution quality, both when tuned for solution quality and when tuned for minimizing the gap. In order to

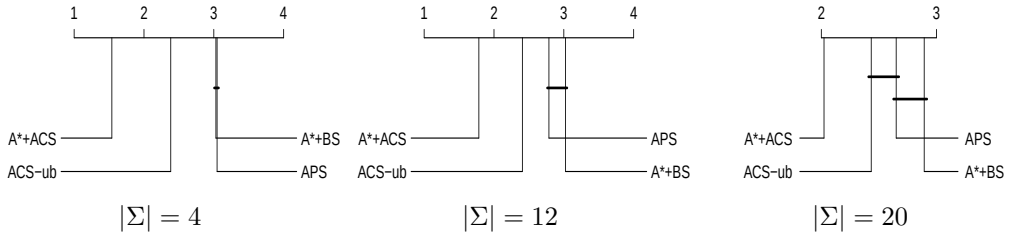


Figure 1: Critical difference plots concerning the results of the algorithms tuned for solution quality. The benchmark instances are split according to alphabet size.

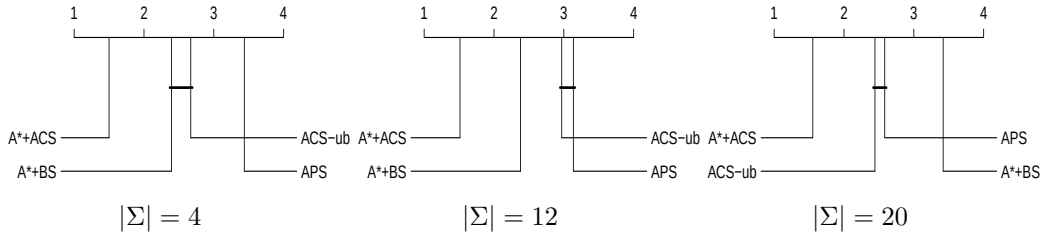


Figure 2: Critical difference plots concerning the results of the algorithms tuned for small gaps. The benchmark instances are split according to alphabet size.

confirm this statistically—at least for the case of tuning for solution quality—we performed Friedman's tests simultaneously considering all four algorithms for the subsets of the benchmark set with different alphabet sizes.[2] Given that in all cases the test rejected the hypothesis that the algorithms perform equally, pairwise comparisons were performed using the Nemenyi post-hoc test [20]. Obtained results are shown in Figure 1 by means of so-called critical difference plots. In short, each algorithm is positioned in the segment according to its average ranking concerning the considered subset of instances. Then, the critical difference (CD) is computed for a significance level of 0.05 and the performance of those algorithms that have a difference lower than CD are regarded as equal—that is, no difference of statistical significance can be detected. This is indicated in the figures by horizontal bars joining the respective algorithms. The figures show that, for each alphabet size, A*+ACS outperforms the other three algorithms with statistical significance.

- Furthermore, it can be observed that A*+ACS outperforms the other three algorithms also concerning the gap. Again, this holds both when tuned for solution quality and when tuned for minimizing the gap. The corresponding critical difference plots—concerning the results obtained after tuning for minimizing the gap—are shown in Figure 2. They confirm that A*+ACS outperforms the other algorithms with statistical significance. As all algorithms make use of the same upper bound function, the difference in gaps must be attributed to the fact that A*+ACS produces significantly better primal solutions than the other algorithms.

- Concerning the remaining three algorithms, it can be observed that A*+BS is generally the weakest algorithm with respect to solution quality. However, this algorithm usually provides better gaps. This is with the exception of instances with $|\Sigma| = 20$ where A*+BS also exhibits the weakest performance regarding the gaps. The best algorithm among A*+BS, APS and ACS-ub concerning solution quality is ACS-ub.

[2] All these tests and the resulting plots were generated using R's **scmamp** package [10].

Table 4: Average results of the algorithms when tuned for small gaps.

(a) $|\Sigma| = 4$.

m	n	**A*+BS**				**APS**				**ACS-ub**				**A* + ACS**			
		$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]	$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]	$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]	$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]
	100	28.9	0.1	1.3	**0.0**	28.9	1.6	1.9	**0.0**	28.9	0.0	2.6	**0.0**	28.9	0.1	2.2	**0.0**
10	500	158.6	28.4	287.7	43.5	160.3	27.4	335.9	44.2	161.1	144.7	486.8	43.4	162.0	67.7	614.2	**42.0**
	1000	320.2	132.6	351.0	47.2	324.2	36.2	200.6	47.1	326.2	233.4	526.9	46.6	330.0	143.3	542.3	**45.4**
	100	21.8	0.2	0.7	**0.0**	21.8	1.2	1.2	**0.0**	21.8	0.0	0.8	**0.0**	21.8	0.1	0.6	**0.0**
50	500	129.4	48.6	476.0	52.2	130.9	37.7	341.7	52.9	131.3	147.9	676.8	52.3	132.3	93.7	551.2	**50.9**
	1000	266.2	156.0	605.5	55.4	268.2	123.9	492.4	55.6	268.9	327.3	747.1	55.4	273.0	217.4	560.3	**54.1**
	100	20.1	0.3	0.9	**0.0**	20.1	1.4	1.4	**0.0**	20.1	0.0	1.1	**0.0**	20.1	0.1	0.8	**0.0**
100	500	122.6	55.0	606.9	54.4	123.7	69.2	438.4	55.2	124.1	139.1	852.3	54.8	124.9	55.3	607.2	**53.3**
	1000	253.8	141.7	675.9	57.4	255.2	195.5	527.0	57.6	255.6	469.6	889.5	57.4	259.5	226.7	672.4	**56.2**
	100	19.0	0.3	0.7	**0.0**	19.0	1.6	1.6	**0.0**	19.0	0.0	0.7	**0.0**	19.0	0.1	0.7	**0.0**
150	500	119.3	24.8	719.1	55.6	120.5	104.7	500.9	56.3	120.7	161.8	894.1	55.9	121.4	25.3	848.4	**54.7**
	1000	247.9	28.4	743.7	58.3	249.4	481.3	830.0	58.5	249.3	214.9	900.0	58.4	253.0	142.8	720.1	**57.2**
	100	18.5	0.4	0.8	**0.0**	18.5	1.6	1.6	**0.0**	18.5	0.1	1.0	**0.0**	18.5	0.1	0.7	**0.0**
200	500	117.3	47.7	810.6	56.2	118.0	199.4	672.2	57.1	118.4	267.5	900.0	56.8	119.5	140.8	768.1	**55.2**
	1000	244.3	93.1	831.8	58.8	245.0	377.7	614.5	59.2	245.4	365.6	900.0	59.1	249.4	349.7	802.3	**57.8**

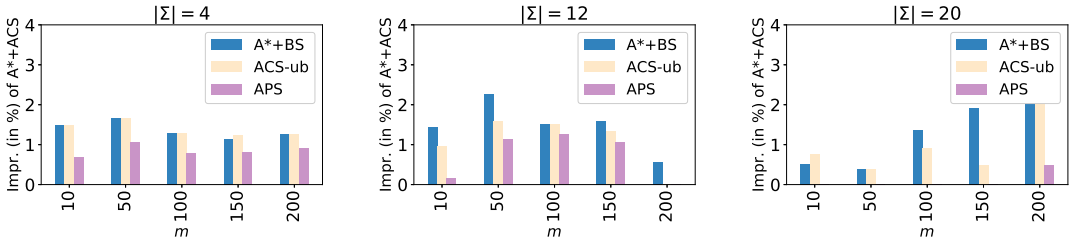
(b) $|\Sigma| = 12$.

m	n	**A*+BS**				**APS**				**ACS-ub**				**A* + ACS**			
		$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]	$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]	$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]	$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]
	100	9.6	< 0.1	< 0.1	**0.0**	9.6	< 0.1	< 0.1	**0.0**	9.6	< 0.1	< 0.1	**0.0**	9.6	< 0.1	< 0.1	**0.0**
10	500	61.4	1.2	270.2	60.8	61.8	32.2	402.4	61.9	62.3	68.1	332.5	61.3	62.2	20.0	601.2	**59.5**
	1000	125.7	3.3	338.6	67.0	128.1	70.6	353.7	66.8	128.7	98.2	332.3	66.6	130.0	151.0	592.1	**65.3**
	100	5.6	< 0.1	< 0.1	**0.0**	5.6	< 0.1	< 0.1	**0.0**	5.6	< 0.1	< 0.1	**0.0**	5.6	< 0.1	< 0.1	**0.0**
50	500	43.1	2.5	426.9	69.7	43.7	38.2	418.7	70.5	43.8	44.1	574.6	70.4	44.1	80.1	621.3	**68.5**
	1000	91.0	76.2	658.4	75.0	91.9	153.1	622.3	75.2	92.3	249.5	736.4	75.1	93.2	74.0	685.5	**74.0**
	100	4.6	< 0.1	< 0.1	**0.0**	4.6	< 0.1	< 0.1	**0.0**	4.6	< 0.1	< 0.1	**0.0**	4.6	< 0.1	< 0.1	**0.0**
100	500	38.9	4.3	656.2	71.6	39.0	72.4	640.3	72.7	39.1	35.4	804.6	72.6	39.2	35.8	840.1	**71.0**
	1000	83.6	17.5	759.3	76.8	84.3	248.6	696.7	77.0	84.7	287.7	854.4	76.9	85.4	143.9	896.2	**75.9**
	100	3.8	< 0.1	< 0.1	**0.0**	3.8	< 0.1	< 0.1	**0.0**	3.8	< 0.1	< 0.1	**0.0**	3.8	< 0.1	< 0.1	**0.0**
150	500	37.0	6.5	784.6	72.7	37.1	107.3	762.7	73.7	37.2	49.3	892.7	73.7	37.3	146.8	891.5	**72.1**
	1000	80.1	18.4	839.8	77.7	80.9	329.1	756.2	77.8	81.0	202.6	900.0	77.9	81.7	171.5	900.0	**76.9**
	100	3.3	< 0.1	< 0.1	**0.0**	3.3	< 0.1	< 0.1	**0.0**	3.3	< 0.1	< 0.1	**0.0**	3.3	< 0.1	< 0.1	**0.0**
200	500	35.6	8.4	867.2	73.5	36.0	140.2	850.0	74.2	36.0	18.1	900.0	74.5	36.0	20.3	900.0	**72.9**
	1000	77.9	26.3	900.0	78.2	78.6	469.9	867.9	78.4	78.8	343.4	900.0	78.3	79.4	117.8	900.0	**77.5**

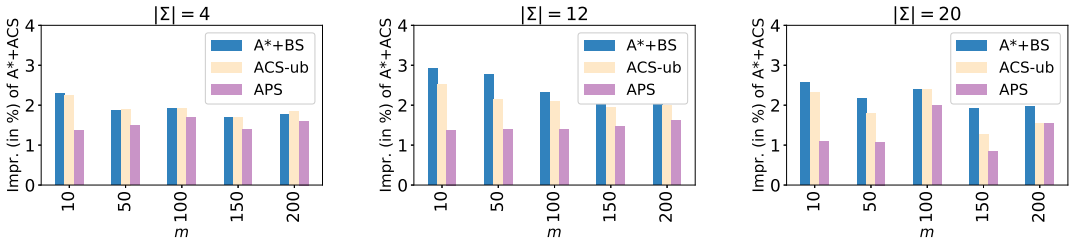
(c) $|\Sigma| = 20$.

m	n	**A*+BS**				**APS**				**ACS-ub**				**A* + ACS**			
		$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]	$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]	$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]	$\overline{\lvert s\rvert}$	$\overline{t_{best}}$[s]	$\overline{t}$[s]	$\overline{gap}$[%]
	100	5.4	< 0.1	< 0.1	**0.0**	5.4	< 0.1	< 0.1	**0.0**	5.4	< 0.1	< 0.1	**0.0**	5.4	< 0.1	< 0.1	**0.0**
10	500	34.4	39.4	457.4	66.8	38.7	15.2	493.4	64.9	38.9	8.2	532.6	64.1	38.9	17.8	581.5	**62.4**
	1000	70.3	96.9	403.1	75.1	80.2	35.3	374.9	72.8	80.9	157.0	580.5	72.2	81.3	115.6	700.3	**71.2**
	100	2.5	< 0.1	< 0.1	**0.0**	2.5	< 0.1	< 0.1	**0.0**	2.5	< 0.1	< 0.1	**0.0**	2.5	< 0.1	< 0.1	**0.0**
50	500	23.2	44.8	749.8	72.2	25.0	22.3	873.8	70.7	25.1	38.6	900.0	71.8	25.0	6.7	735.4	**70.2**
	1000	50.2	155.7	786.3	80.9	54.8	87.8	654.5	80.2	55.0	115.4	900.0	79.9	55.5	124.3	825.6	**79.0**
	100	1.3	2.9	< 0.1	**0.0**	1.3	2.9	< 0.1	**0.0**	1.3	2.9	< 0.1	**0.0**	1.3	2.9	< 0.1	**0.0**
100	500	20.4	151.5	900.0	73.7	21.9	46.9	900.0	72.9	22.1	68.5	900.0	73.6	21.9	24.1	900.0	**71.6**
	1000	45.6	197.9	900.0	82.5	49.0	139.7	900.0	81.9	49.0	32.0	900.0	81.8	49.7	73.6	900.0	**80.9**
	100	1.1	4.2	< 0.1	**0.0**	1.1	2.9	< 0.1	**0.0**	1.1	2.9	< 0.1	**0.0**	1.1	2.9	< 0.1	**0.0**
150	500	19.0	10.0	900.0	74.8	20.9	60.9	900.0	73.4	21.0	79.8	900.0	72.8	20.9	41.7	900.0	**72.1**
	1000	43.2	114.9	900.0	83.3	46.7	244.5	900.0	82.7	46.7	100.4	900.0	82.7	47.0	47.2	900.0	**81.8**
	100	1.1	3.9	< 0.1	**0.0**	1.1	4.7	< 0.1	**0.0**	1.1	4.7	< 0.1	**0.0**	1.1	4.7	< 0.1	**0.0**
200	500	18.1	52.1	900.0	76.2	19.7	104.0	900.0	74.7	19.9	112.9	900.0	73.9	19.8	288.6	900.0	**73.4**
	1000	42.2	341.8	900.0	83.6	45.0	280.5	900.0	83.3	45.0	48.9	900.0	83.3	45.1	36.6	900.0	**82.6**

The most important information from the result tables is additionally shown in a graphical way in Figure 3, which depicts the average improvements (in percent) in terms of solution quality of A*+ACS over the other algorithms. There is one figure for each combination of $|\Sigma|$ and n. The same information—that is, the average improvement in percent—concerning the gap is shown in Figure 4.

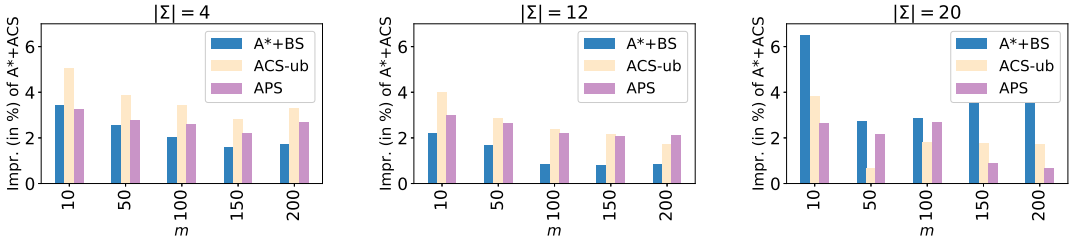


(a) Instances with $n = 500$.

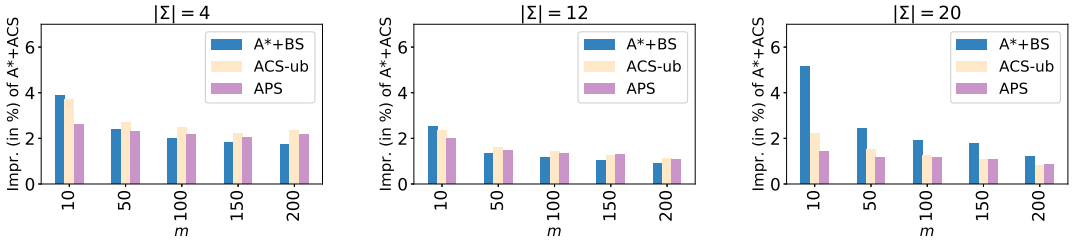


(b) Instances with $n = 1000$.

Figure 3: Improvements in solution quality of A*+ACS over the other algorithms.



(a) Instances with $n = 500$.



(b) Instances with $n = 1000$.

Figure 4: Improvements in the gap of A*+ACS over the other algorithms.

6.4. The Anytime Performance of the Algorithms

As stated above, apart from the solution quality and the optimality gap finally obtained by the algorithms when the time limit is reached, another important aspect of their behavior concerns the anytime performance. In order to visualize the anytime performance, we plot the evolution of the solution quality over time (averaged over ten problem instances of the same type). Figure 5 shows these plots for the representative case with $m = 50$ and $n = 1000$ considering all three alphabet sizes. In addition to the line plots for the average

behavior, boxplots indicating the variance are shown every 200 seconds.[3] The evolution of the obtained average gaps over time are shown in the same way in Figure 6. Note that information is only plotted concerning complete—in the sense of non-expandable—solutions. This is the reason why, for instance, the anytime line plot of APS in the three graphics of Figure 5 does not start at time zero.

The following observations can be made with respect to the anytime plots on solution quality.

- A*+ACS outperforms all other approaches during all stages of the search process. That is, A*+ACS finds better solutions than the other algorithms already very early during the search process. Moreover, A*+ACS does not seem to suffer as much from early stagnation as A*+BS. This boost of solution quality can primarily be attributed to the incorporation of the new approximate expected length calculation (15) as heuristic function, which turns out to be a much better guidance than classical upper bounds. A direct indication for this is obtained when comparing the anytime plots of A*+ACS and ACS-ub (which does not make use of the approximate expected length function).
- APS and A*+BS, which both make use of embedded BS runs in order to find good heuristic solutions, show a similar anytime behavior considering solution quality. However, a rather large beam size (β) is required in order to obtain the best possible solution quality at the end of a run. This fact is obviously negative for the anytime performance of the algorithms, as they perform very few major iterations. The role of the A* iterations is almost irrelevant for A*+BS.
- ACS-ub not only outperforms A*+BS and APS concerning the final solution quality, it also shows an improved anytime performance when comparing with the ones of APS and A*+BS.

When considering the anytime plots concerning the evolution of the gaps, the following can be observed:

- A*+ACS produces significantly better gaps when compared to those of the three other algorithms, over the whole run-time of the algorithms. This means that the significantly increased number of A* iterations (when compared to the parameter setting aimed for solution quality; see Tables 1e and 2e) pays off for A*+ACS. However, it is also interesting to remark that, even with the parameter setting aimed for minimizing the gaps, the algorithm still provides a performance concerning solution quality that outperforms all other approaches.
- Even though A*+BS uses a number of A* iterations that is one order of magnitude larger than the one used by A*+ACS, this does not really pay off for the algorithm. In the case of the instances with $|\Sigma| = 20$, for example, A*+BS shows, by far, the worst anytime performance in the comparison.
- ACS-ub and APS show a similar anytime performance concerning the evolution of the gaps.
- Generally, A*+ACS shows a very good balance between ensuring good gaps and providing high quality heuristic solutions. This can be explained by the use of both, an improved upper bound function and a strong guidance by our approximate heuristic, see (15).

[3] We provide the complete set of graphics, concerning all combinations of n and m, as supplementary material under `https://www.ac.tuwien.ac.at/research/problem-instances/LCPS`.

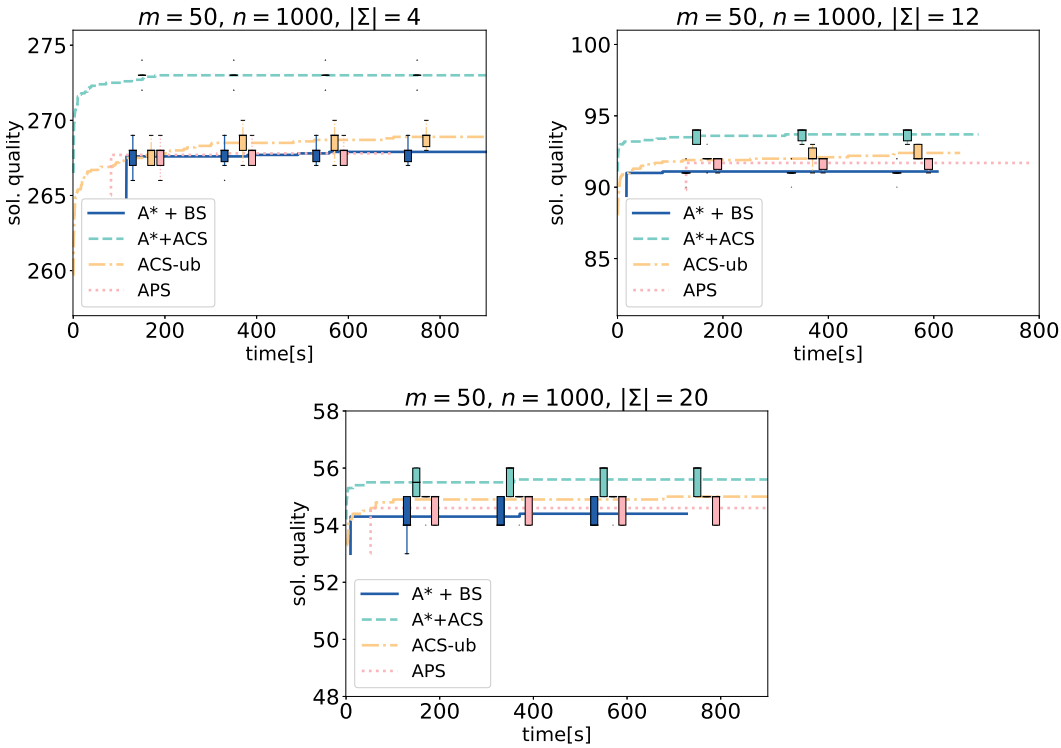


Figure 5: Evolution of solution quality over time for the exemplary case with $m = 50$ and $n = 1000$. The algorithms were tuned for solution quality.

6.5. Computational Study for the 2–LCPS Problem

As mentioned in Section 2, the existing works from the literature on the LCPS problem are primarily consider exact algorithms for the problem variant with two input strings ($m = 2$), that is, the 2–LCPS problem. We decided to implement all these approaches, that is, the DP and the MNDRS approaches from [12] and the CPSA approach from [26]; see also Section 2. However, as some of these approaches were only described from a theoretical point of view in the original papers, we sometimes had to make our own design decisions for what concerns, for example, suitable concrete data structures. A detailed description of our implementations can be found in the supplementary material under `https://www.ac.tuwien.ac.at/research/problem-instances/LCPS`. In addition to these three approaches we tested a basic *Constraint Programming* (CP) model, detailed in Appendix A in conjunction with MiniZinc 2.1.5 and its Gecode backbone solver.

Finally, we will also compare to our pure A* search as described in Section 3.3. Compared to the hybrid A* approaches, pure A* can be expected to require less node expansions to prove optimality.

The five approaches now considered were applied once with a computation time limit of 900 seconds and a memory limit of 15 GB to the 150 2–LCPS instances described in Section 6.1. Results are shown in Table 5, which lists for each instance group (n, $|\Sigma|$) consisting of ten instances and for each approach the number of instances the method was able to solve to proven optimality ($\#opt$), the number of instances for which the method was terminated either due to exceeding the time limit ($\#te$) or the memory limit ($\#me$), and the average computation times of all successful runs (or "–" if no run could

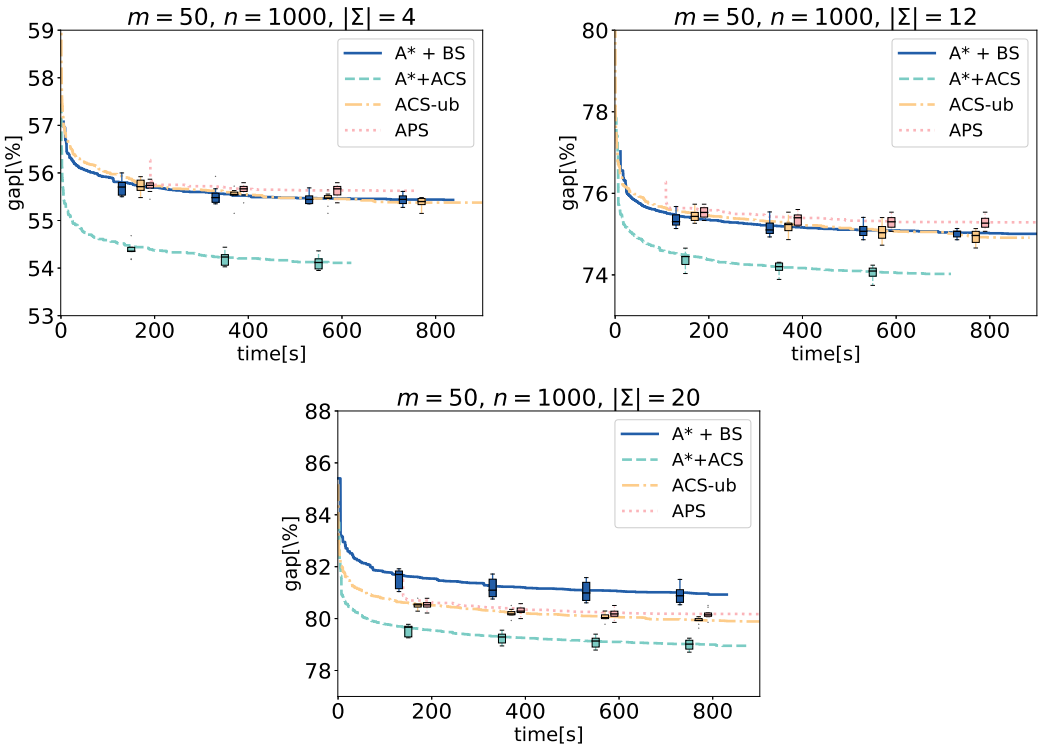


Figure 6: Evolution of the optimality gaps over time for the exemplary case with $m = 50$ and $n = 1000$. The algorithms were tuned for minimizing the gap.

Table 5: Results for the 2–LCPS instances.

n	$\|\Sigma\|$	A*				MNDRS				CPSA				DP				CP			
		#opt	*#te*	*#me*	$\bar{t}[s]$	*#opt*	*#te*	*#me*	$\bar{t}[s]$	*#opt*	*#te*	*#me*	$\bar{t}[s]$	*#opt*	*#te*	*#me*	$\bar{t}[s]$	*#opt*	*#te*	*#me*	$\bar{t}[s]$
100	4	10	0	0	0.2	10	0	0	0.4	10	0	0	0.4	10	0	0	1.1	10	0	0	15.8
	12	10	0	0	0.2	10	0	0	< 0.1	10	0	0	0.2	10	0	0	1.1	10	0	0	4.9
	20	10	0	0	0.2	10	0	0	< 0.1	10	0	0	0.2	10	0	0	1.1	10	0	0	1.3
200	4	10	0	0	0.6	10	0	0	8.5	10	0	0	27.4	10	0	0	25.6	0	10	0	–
	12	10	0	0	0.3	10	0	0	0.2	10	0	0	2.7	10	0	0	20.7	0	10	0	–
	20	10	0	0	0.2	10	0	0	< 0.1	10	0	0	0.8	10	0	0	13.7	0	10	0	–
300	4	10	0	0	5.2	10	0	0	45.7	10	0	0	431.3	10	0	0	61.5	0	10	0	–
	12	10	0	0	0.3	10	0	0	1.4	10	0	0	22.1	10	0	0	60.9	0	10	0	–
	20	10	0	0	0.2	10	0	0	< 0.1	10	0	0	5.9	10	0	0	54.0	0	10	0	–
400	4	10	0	0	26.6	9	0	1	158.9	0	10	0	–	0	0	10	–	0	10	0	–
	12	10	0	0	7.9	10	0	0	7.6	10	0	0	154.0	0	0	10	–	0	10	0	–
	20	10	0	0	2.9	10	0	0	1.6	10	0	0	31.1	0	0	10	–	0	10	0	–
500	4	10	0	0	64.9	0	0	10	–	0	10	0	–	0	0	10	–	0	10	0	–
	12	10	0	0	24.3	10	0	0	17.8	10	0	0	745.0	0	0	10	–	0	10	0	–
	20	10	0	0	9.8	10	0	0	4.9	10	0	0	108.2	0	0	10	–	0	10	0	–

prove optimality). The average computation times are also provided in graphical form in Figure 7. Note that the y-axis of these plots uses a logarithmic scaling. Moreover, cases in which the memory limit was exceeded are marked with a black asterisk, while cases in which the maximum allowed computation time was exceeded are marked with ">900".

The obtained results allow to draw the following conclusions:

- Our A* approach is the only algorithm that can find an optimal solution and prove its optimality within the time limit and respecting the imposed memory constraint over all considered instances.
- MNDRS is the second-best algorithm, only starting to fail for instances with $|\Sigma| = 4$ and $n \in \{400, 500\}$. In particular, in those cases MNDRS fails due to exceeding the

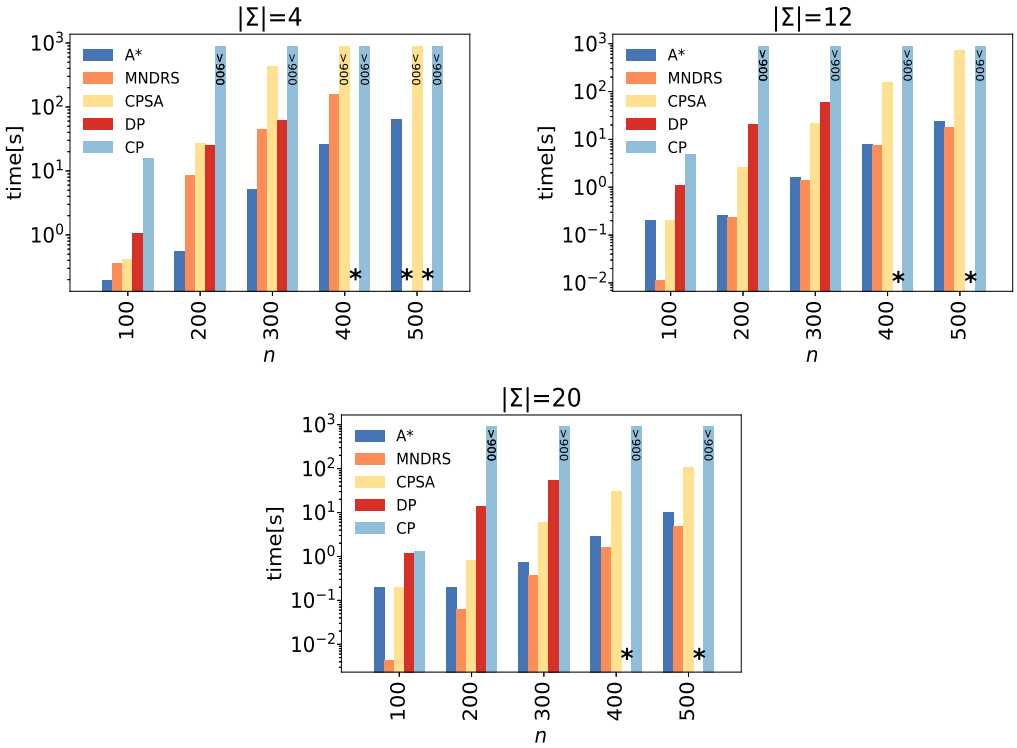


Figure 7: Average computation times of the algorithms for 2–LCPS.

memory limit with the data structures it requires.

- CPSA fails for the same problem instance types as MNDRS. However, in contrast to MNDRS it fails due to exceeding the computation time limit.
- DP is only able to solve problem instances up to $n = 300$. Starting from $n = 400$, the algorithm fails due to the memory limit.
- The CP approach is clearly the weakest one in the comparison. This approach is only able to solve the problem instances with $n = 100$. In all other cases, CP fails due to reaching the computation time limit. In fact, starting from $n = 400$, CP is not able to provide any solution within the allowed computation time.
- Concerning the computation time requirements, we can observe that A* and MNDRS—when able to solve an instance—are the fastest approaches. A* has advantages in the context of instances with $|\Sigma| = 4$. For instances with $|\Sigma| = 12$ both algorithms require comparable times, and for $|\Sigma| = 12$ MDRS is on average slightly faster. On the other side, CP is by far the most time consuming approach.

Finally, we would like to stress again that A* is the only algorithm which was able to solve all instances to optimality, respecting the time and memory limits. This confirms that the way of merging the nodes in the state graph of the A* search has a crucial impact on reducing the algorithms' memory consumption.

7. Conclusion and Future Work

We considered the LCPS problem and studied a variety of algorithms for it. For exactly solving the problem, the A* search from [17] was extended in particular by a more effective upper bound calculation. This algorithm is able to reliably solve LCPS instances with two input strings to proven optimality in short time, even when the strings have lengths up to 500 letters. None of the other solution approaches we considered performed equally consistent.

When it comes to solve the LCPS for a larger number of strings, however, the classical A* search clearly also has its limits due to the complexity of the problem. In this case, anytime algorithms are particularly interesting for practice, as they deliver promising heuristic solutions almost immediately, can be expected to continuously improve on them, and still retain the chance of finishing with proven optimality when the time allows. The first algorithm of this kind for the LCPS was the hybrid A*+BS proposed in our preliminary work [17], which embeds beam search in the A* framework. As a weakness we recognized that calling the beam search more frequently with lower beam width is usually substantially less effective than calling it only fewer times with larger beam width. Unfortunately, this property stays in contrast to the goals of an anytime approach, where we expect to obtain improved solutions more continuously.

With A*+ACS, we provide now a clearly superior approach. It replaces the beam search with iterations of anytime column search, which expands nodes at all levels more uniformly and therefore leads to a more continuous improvement. Most importantly, we also introduced a novel heuristic function that represents an approximation for the expected length of the LCPS. Using this function as guidance within the ACS iterations instead of the classical upper bound calculation leads to substantially better heuristic solutions. In order to still obtain quality guarantees and optimality proofs when time allows, classical A* iterations still rely on the (improved) upper bound calculation.

Different parameter settings are suitable for A*+ACS whether one aims just on pure heuristic performance or when considering also upper bounds and wanting to minimize optimality gaps. We performed detailed parameter tuning for both cases using irace, and the obtained settings also provide a solid basis for reasonable choices when confronted with new instances. Our computational evaluation and comparison of different A*-based anytime approaches for the LCPS clearly showed the benefits and superiority of the new A*+ACS w.r.t. final solution quality, final remaining optimality gap, as well as the overall anytime-behavior.

In future work it would be interesting to develop a parallel variant of A*+ACS. In fact, performing classical A* node expansions as well as ACS iterations in parallel while maintaining the priority queues as shared resources would be quite natural. While we focused here on the LCPS, there are also the classical LCS and other problem variants like the repetition-free LCS [2] and the arc-annotated LCS [32], for which the proposed approaches seems to be promising. Actually, A*+ACS represents a novel more general anytime search framework that can easily be adapted to other problems, providing that suitable upper bound and heuristic guidance functions can be defined.

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 nr. W1260-N35.

References

[1] A. Abboud, A. Backurs, and V. V. Williams. Tight hardness results for LCS and other sequence similarity measures. In *Proceedings of FOCS 2015 – 56th Annual Symposium on Foundations of Computer Science*, pages 59–78. IEEE press, 2015.

[2] S. S. Adi, M. D. Braga, C. G. Fernandes, C. E. Ferreira, F. V. Martinez, M.-F. Sagot, M. A. Stefanes, C. Tjandraatmadja, and Y. Wakabayashi. Repetition-free longest common subsequence. *Discrete Applied Mathematics*, 158(12):1315–1324, 2010. Traces from LAGOS'07 – IV Latin American Algorithms, Graphs, and Optimization Symposium Puerto Varas - 2007.

[3] S. Aine, P. P. Chakrabarti, and R. Kumar. AWA* - A window constrained anytime heuristic search algorithm. In M. M. Veloso, editor, *Proceedings of IJCAI 2007 – Proceedings of the 20th International Joint Conference on Artificial Intelligence*, pages 2250–2255, 2007.

[4] 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.

[5] C. Blum. Beam–ACO for the longest common subsequence problem. In *Proceedings of CEC 2010 – the IEEE Congress on Evolutionary Computation*, pages 1–8, 2010.

[6] 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.

[7] C. Blum and P. Festa. Longest common subsequence problems. In *Metaheuristics for String Problems in Bio-informatics*, chapter 3, pages 45–60. John Wiley & Sons, 2016.

[8] P. Bonizzoni, G. Della Vedova, and G. Mauri. Experimenting an approximation algorithm for the LCS. *Discrete Applied Mathematics*, 110(1):13–24, 2001.

[9] P. Brisk, A. Kaplan, M. Sarrafzadeh, and M. Sarrafzadeh. Area-efficient instruction set synthesis for reconfigurable system-on-chip designs. In *Proceedings of DAC '04 – 41st Annual Design Automation Conference*, pages 395–400. ACM, 2004.

[10] B. Calvo and G. Santafe. scmamp: Statistical comparison of multiple algorithms in multiple problems. *The R Journal*, 8(1):248–256, 2016.

[11] C. Q. Choi. DNA palindromes found in cancer. *Genome Biology*, 6(1):6:spotlight–20050216–01, 2005.

[12] 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.

[13] V. Chvátal and D. Sankoff. Longest common subsequences of two random sequences. *Journal of Applied Probability*, 12(2):306–315, 1975.

[14] V. Dančik and M. Paterson. Upper bounds for the expected length of a longest common subsequence of two binary sequences. *Random Structures & Algorithms*, 6(4):449–458, 1995.

[15] T. Dean and M. Boddy. An analysis of time-dependent planning. In *Proceedings of AAAI'88 – 7th AAAI National Conference on Artificial Intelligence*, pages 49–54. AAAI Press, 1988.

[16] J. D. Dixon. Longest common subsequences in binary sequences. *ArXiv e-prints*, 2013.

[17] M. Djukanovic, G. Raidl, and C. Blum. Exact and heuristic approaches for the longest common palindromic subsequence problem. In *Proceedings of LION12 – 12th International Conference on Learning and Intelligent Optimization*, Lecture Notes in Computer Science. Springer, 2018. In press.

[18] T. Easton and A. Singireddy. A specialized branching and fathoming technique for the longest common subsequence problem. *International Journal of Operations Research (Taichung)*, 2:98–104, 2006.

[19] T. Easton and A. Singireddy. A large neighborhood search heuristic for the longest common subsequence problem. *Journal of Heuristics*, 14(3):271–283, 2008.

[20] S. García and F. Herrera. An extension on "statistical comparisons of classifiers over multiple data sets" for all pairwise comparisons. *Journal of Machine Learning Research*, 9:2677 – 2694, 2008.

[21] M. Giel-Pietraszuk, M. Hoffmann, S. Dolecka, J. Rychlewski, and J. Barciszewski. Palindromes in proteins. *Journal of Protein Chemistry*, 22(2):109–113, 2003.

[22] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Computer Science and Computational Biology. Cambridge University Press, Cambridge, 1997.

[23] E. A. Hansen and R. Zhou. Anytime heuristic search. *Journal of Artificial Intelligence Research*, 28(1):267–297, 2007.

[24] E. A. Hansen, S. Zilberstein, and V. A. Danilchenko. Anytime heuristic search: First results. Technical report, University of Massachusetts, Amherst, MA, USA, 1997.

[25] 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.

[26] M. M. Hasan, A. S. M. S. Islam, M. S. Rahman, and A. Sen. Palindromic subsequence automata and longest common palindromic subsequence. *Mathematics in Computer Science*, 11(2):219–232, 2017.

[27] E. J. Horvitz. Reasoning about beliefs and actions under computational resource constraints. In *Proceedings of UAI'87 – 3rd Conference on Uncertainty in Artificial Intelligence*, pages 429–447. AUAI Press, 1987.

[28] W. J. Hsu and M. W. Du. Computing a longest common subsequence for a set of strings. *BIT Numerical Mathematics*, 24:45–59, 1984.

[29] S. Inenaga and H. Hyyrö. A hardness result and new algorithm for the longest common palindromic subsequence problem. *Information Processing Letters*, 129(C):11–15, 2018.

[30] T. Jiang and M. Li. On the approximation of shortest common supersequences and longest common subsequences. *SIAM Journal on Computing*, 24(5):1122–1139, 1995.

[31] 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.

[32] T. Jiang, G. Lin, B. Ma, and K. Zhang. The longest common subsequence problem for arc-annotated sequences. *Journal of Discrete Algorithms*, 2(2):257–270, 2004.

[33] M. Kiwi, M. Loebl, and J. Matoušek. Expected length of the longest common subsequence for large alphabets. *Advances in Mathematics*, 197(2):480–498, 2005.

[34] 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):013105, 2008.

[35] M. Likhachev, G. J. Gordon, and S. Thrun. ARA*: Anytime A* with provable bounds on sub-optimality. In *Proceedings of NIPS'04 – 17th International Conference on Neural Information Processing Systems*, pages 767–774, 2004.

[36] M. Lozano and C. Blum. A hybrid metaheuristic for the longest common subsequence problem. In *Proceedings of HM'10 – 7th International Conference on Hybrid metaheuristics*, pages 1–15. Springer-Verlag, 2010.

[37] M. López-Ibáñez, J. Dubois-Lacoste, L. P. Cáceres, M. Birattari, and T. Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

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

[39] S. R. Mousavi and F. Tabataba. An improved algorithm for the longest common subsequence problem. *Computers & Operations Research*, 39(3):512–520, 2012.

[40] M. Rafiqul Islam, C. M. K. Saifullah, Z. Tasnim Asha, and R. Ahamed. Chemical reaction optimization for solving longest common subsequence problem for multiple string. *Soft Computing*, pages 1–25, 2018. In press.

[41] S. J. Shyu and C.-Y. Tsai. Finding the longest common subsequence for multiple biological sequences by ant colony optimization. *Computers & Operations Research*, 36(1):73–91, 2009.

[42] Singireddy A. Solving the longest common subsequence problem in bioinformatics. Master thesis, Industrial and Manufacturing Systems Engineering, Kansas State University, Manhattan, KS, 2007.

[43] G. R. Smith. Meeting DNA palindromes head-to-head. *Genes & development*, 22(19):2612–2620, 2008.

[44] J. A. Storer. *Data compression: methods and theory*. Computer Science Press, Inc., 1987.

[45] H. Tanaka, D. A. Bergstrom, M. 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.

[46] S. G. Vadlamudi, S. Aine, and P. P. Chakrabarti. MAWA* – A Memory–Bounded Anytime Heuristic–Search Algorithm. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 41(3):725–735, 2011.

[47] S. G. Vadlamudi, S. Aine, and P. P. Chakrabarti. Anytime pack search. *Natural Computing*, 15(3):395–414, 2016.

[48] S. G. Vadlamudi, P. Gaurav, S. Aine, and P. P. Chakrabarti. Anytime column search. In *Proceedings of AI 2012 – 25th Australasian joint conference on Advances in Artificial Intelligence*, pages 254–265. Springer-Verlag, 2012.

[49] J. van den Berg, R. Shah, A. Huang, and K. Y. Goldberg. Anytime nonparametric A*. In *Proceedings of AAAI 2011 – 25th AAAI Conference on Artificial Intelligence, San Francisco, California, USA, August 7-11, 2011*, 2011.

[50] Q. Wang, M. Pan, Y. Shang, and D. Korkin. A fast heuristic search algorithm for finding the longest common subsequence of multiple strings. In *Proceedings of AAAI 2010 – 24th AAAI Conference on Artificial Intelligence*, 2010.

[51] I.-H. Yang, C.-P. Huang, and K.-M. Chao. A fast algorithm for computing a longest common increasing subsequence. *Information Processing Letters*, 93(5):249–253, 2005.

[52] W. Zhang. Complete anytime beam search. In *Proceedings of AAAI '98/IAAI '98 – the 15th National/10th Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, pages 425–430. AAAI Press, 1998.

[53] R. Zhou and E. A. Hansen. Beam-stack search: Integrating backtracking with beam search. In *Proceedings of ICAPS'05 – 15th International Conference on International Conference on Automated Planning and Scheduling*, pages 90–98. AAAI Press, 2005.

[54] S. V. Znamenskij. Approximation of the longest common subsequence length for two long random strings. *Program Systems: Theory and Applications*, 7(4):347–358, 2016.

Appendices

A. Constraint Programming model for the LCPS

For comparison purposes, we consider the following basic CP model for the LCPS problem, which was implemented in MiniZinc 2.1.5. Comparing FlatZinc, Chuffed and Gecode as backbone solvers, we found Gecode to usually work best for this model.

The model uses the following variables. Let $r \in \{1, ..., l\}$, with $l = \min\{|s_i| \,|\, i = 1, \ldots, m\}$ denote the length of the solution string, which shall be maximized. Decision variable $T_{i,j} \in \{1, \ldots, |s_i|\}$ represents the index of the solution string's j-th letter in the i-th input string, for $i = 1, \ldots, m$ and $j = 1, \ldots, r$.

The LCPS is now expressed as follows.

$$\max\ r \tag{16}$$

$$T_{i,j} < T_{i,j+1} \qquad \forall i = 1, \ldots, m,\ j = 1, \ldots, r-1 \tag{17}$$

$$s_i[T_{i,j}] = s_{i+1}[T_{i+1,j}] \qquad \forall i = 1, \ldots, m-1,\ \forall j = 1, \ldots, r \tag{18}$$

$$s_1[T_{1,j}] = s_1[T_{1,r-j+1}] \qquad \forall\ j = 1, \ldots, \lfloor r/2 \rfloor \tag{19}$$

Constraints (17) ensure that the sequence of indices $T_{i,1}, \ldots, T_{i,r}$ is strongly monotonically increasing for each input string s_i. Constraints (18) guarantee that for each $j = 1, \ldots, r$,

the letter at position $T_{i,j}$ in string s_i is the same over all $i = 1, \ldots, m$. Last but not least, constraints (19) guarantee that the solution is palindromic. Preliminary experiments indicated in this respect that stating these constraints redundantly for all input strings speeds up the solving process.