Applied Soft Computing Journal 95 (2020) 106499

Contents lists available at ScienceDirect

Applied Soft Computing Journal

journal homepage: www.elsevier.com/locate/asoc

Finding Longest Common Subsequences: New anytime A* search results

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

[a] Institute of Logic and Computation, TU Wien, Vienna, Austria
[b] Artificial Intelligence Research Institute (IIIA-CSIC), Campus UAB, Bellaterra, Spain

ARTICLE INFO

Article history:
Received 27 November 2019
Received in revised form 28 April 2020
Accepted 19 June 2020
Available online 26 June 2020

Keywords:
Longest common subsequence problem
Hybrid metaheuristic
A* search
Beam search
Anytime column search

ABSTRACT

The Longest Common Subsequence (LCS) problem aims at finding a longest string that is a subsequence of each string from a given set of input strings. This problem has applications, in particular, in the context of bioinformatics, where strings represent DNA or protein sequences. Existing approaches include numerous heuristics, but only a few exact approaches, limited to rather small problem instances. Adopting various aspects from leading heuristics for the LCS, we first propose an exact A* search approach, which performs well in comparison to earlier exact approaches in the context of small instances. On the basis of A* search we then develop two hybrid A*–based algorithms in which classical A* iterations are alternated with beam search and anytime column search, respectively. A key feature to guide the heuristic search in these approaches is the usage of an approximate expected length calculation for the LCS of uniform random strings. Even for large problem instances these anytime A* variants yield reasonable solutions early during the search and improve on them over time. Moreover, they terminate with proven optimality if enough time and memory is given. Furthermore, they yield upper bounds and, thus, quality guarantees when terminated early. We comprehensively evaluate the proposed methods using most of the available benchmark sets from the literature and compare to the current state-of-the-art methods. In particular, our algorithms are able to obtain new best results for 82 out of 117 instance groups. Moreover, in most cases they also provide significantly smaller optimality gaps than other anytime algorithms.

1. Introduction

In computer science strings are widely used for representing sequence information. Words and longer texts are naturally represented by means of strings, and in the field of bioinformatics, DNA, RNA and protein sequences, for example, play particularly important roles. We formally define a *string* s as a finite sequence of $|s|$ characters from a finite alphabet Σ. A frequently occurring necessity is to detect similarities between several strings in order to derive relationships and possibly predict different aspects of a set of strings. A *subsequence* of a string s is any sequence obtained by removing arbitrary characters from s. A natural and common way to compare two or more strings is studying their *common subsequences*. More specifically, given a set of m input strings $S = \{s_1, \ldots, s_m\}$, the *Longest Common Subsequence* (LCS) problem [1] aims at finding a subsequence of maximal length which is common for all the strings in S.

As mentioned above, the length of the LCS of two or more input strings is a popular similarity measure in computational biology. More generally, there is a large range of real-world applications in which it is necessary to compute a measure of similarity between two or more sequences, and the requirements are sometimes different. Depending on the application, these sequences may encode biological information (such as, for example, in DNA or RNA strings), sentences, whole texts, or time series (including video signals and speech sequences). Well-known similarity measures in the context of computational biology include, besides the LCS length, the *Levenshtein* distance which calculates the minimum number of single-character edits (insertions, deletions or substitutions) required to change one sequence into the other. Another example is the *Damerau–Levenshtein* distance [2] which adds transpositions to the three edit operations that are already considered in the Levenshtein distance. Finally, it is also worth to mention the *Canberra* distance (used, for example, to analyze the gut microbiome in different disease states), and the *Google* distance [3]. Well-known similarity measures for sentences and/or texts include metrics such as the *Euclidean*, the *Manhattan* and the *Minkovski* distance [4]. The *soft cosine measure* [5] considers similarities between pairs of features, and the *Jaccard* similarity [6]

* Corresponding author.
E-mail addresses: djukanovic@ac.tuwien.ac.at (M. Djukanovic), raidl@ac.tuwien.ac.at (G.R. Raidl), christian.blum@iiia.csic.es (C. Blum).

https://doi.org/10.1016/j.asoc.2020.106499
1568-4946/

is defined as the size of the intersection divided by size of the union of two sets. Finally, well-known measures of similarity for time series include *Dynamic Time Warping* (DTW) [7], the *matrix-based Euclidean* distance (GMED), and *matrix-based dynamic time warping* (GMDTW) [8], among others. Recently, many approaches from the field of deep learning and machine learning have been developed to derive measures of similarities that take the semantic meaning of the compared sentences into account. These include *deep architecture Match-SRNN* [9] that utilizes a spatial recurrent neural network to generate the global interaction between two sentences, the *Word Order Similarity* [10] which is defined as the normalized difference of word order between two sentences, and the *Latent Semantic Analysis* (LSA) [11]. However, in this work we focus on the efficient calculation of the LCS measure. Apart from applications in computational biology [12], the necessity to calculate this measure arises, for example, in data compression [13,14], text editing [15], the production of circuits in field programmable gate arrays [16], and file comparison (used in the `Unix` command *diff*) [17].

For fixed m polynomial algorithms based on dynamic programming (DP) are known [18] to solve the LCS problem. Standard dynamic programming approaches run in $O(n^m)$ time, where n denotes the length of the longest input string. These exact methods become quickly impractical when m grows and n is not small. For a general number of input strings m the LCS problem is known to be $\mathcal{NP}$-hard [1]. In practice, heuristic techniques are typically used for larger m and n. The Expansion algorithm and the Best-Next heuristic [19,20] are well known simple and fast construction heuristics, respectively. Substantially better solutions can usually be obtained by more advanced search strategies and metaheuristics. Among these are in particular many approaches that are based on *Beam Search* (BS), see e.g., [21–25], and they differ in various important details such as the heuristic guidance, the branching mechanism, and the filtering.

In our recent work [26], we proposed a general BS framework for the LCS that unifies all the heuristic state-of-the-art approaches from the literature in the sense that each one can be expressed by respective configuration settings. Moreover, a novel heuristic guidance function was proposed, which approximates the expected length of a LCS for random strings. In a comprehensive experimental comparison previous methods have been compared and a new state-of-the-art BS variant was determined, which dominates the other approaches on most of the available benchmark instances. The mentioned new heuristic guidance function turned hereby out to play a crucial role.

Concerning exact approaches for the LCS problem, an integer linear programming model has been considered in [27]. It is, however, not competitive as it cannot be applied to any of the commonly used benchmark instances due to too many variables and constraints in the model. Dynamic programming approaches are reasonable for small m and small n, but they also quickly run out of memory for larger instances and then typically return only weak solutions, if at all. Chen et al. [28] proposed the parallel FAST_LCS search algorithm, which is based on producing a special successors table to obtain all the identical pairs and their levels. Successor nodes are derived in parallel. Pruning operations are utilized to reduce the computational effort. While the algorithm is effective for a small number of input strings, it also struggles for larger m. Wang et al. [24] proposed another parallel algorithm called QUICK-DP, which is based on the dominant point approach and employs a fast divide-and-conquer technique to compute the dominant points. More recently, Li et al. [29] suggested the Top_MLCS algorithm, which is based on a directed acyclic layered-graph model (called irredundant common subsequence graph) and parallel topological sorting strategies used to filter out paths representing suboptimal solutions. Moreover, the authors showed that the earlier dominant-point-based algorithms do not scale well to larger LCS instances, and Top_MLCS significantly outperforms them. In addition to the sequential Top_MLCS, also a parallel variant was proposed. Another parallel space efficient algorithm based on a graph model, called the Leveled-DAG, was described by Peng and Wang [30]. It eliminates all the nodes in the layered graph that do not contribute to the construction of the LCS, and thus keeps only the nodes from the current level and some previously generated ones. In the experimental comparison, Leveled-DAG and Top_MLCS solved the same number of benchmark instances to proven optimality, but Leveled-DAG consumed less memory.

Despite these recent advances, solving practically relevant instances to proven optimality remains a substantial challenge in terms of memory and computation time, even when utilizing many parallel threads. The existing exact methods are therefore frequently not applicable in practice. As a compromise between classical exact techniques and pure heuristic approaches, *anytime algorithms* have been proposed [31,32]. An anytime algorithm is supposed to fulfill the following properties: (1) It is, in principle, complete in the sense that it terminates with a proven optimal solution when enough time and memory is provided; (2) it can be terminated at almost any time and then returns a solution of reasonable quality; and (3) the solution quality improves with the given time.

Anytime algorithms thus offer to choose the trade-off between solution quality and computational requirements. Concerning the LCS problem, two anytime approaches have been proposed in the literature so far: Pro-MLCS [33] and SA-MLCS [34]. Both algorithms are based on the dominant point method [35], which features a special distance measure `dist` for heuristic guidance and a specific multi-dimensional data structure for checking the dominance relation of already explored nodes during the search. Algorithm Pro-MLCS iteratively extends a fixed number of nodes at each level in a level-by-level manner and is similar to anytime column search [36], which we will consider in more detail in Section 4.2. On the other side, SA-MLCS applies an iterative beam widening strategy in successive iterations to reduce space requirements. It differs from Pro-MLCS in the data structures utilized to maintain open nodes. A specific priority queue is realized for SA-MLCS which stores those nodes whose children have not all been expanded, further exploited in the algorithm to make use of the search information from previous iterations to improve efficiency of the SA-MLCS. Last but not least, [34] describes another memory bounded variant of SA-MLCS, called SLA-MLCS. A weakness of all these approaches is that they are not able to provide an upper bound on the solution quality and therefore no quality guarantee in case of early termination. Moreover, neither in [33] nor in [34] enough details are provided concerning the multi-dimensional data structure for checking dominance. This made it, unfortunately, impossible to re-implement the algorithms with all their details, and source code is not provided by the authors. However, in the experimental section of this work we consider the distance measure `dist` as an alternative heuristic guidance and we also build upon anytime column search.

Our contributions are as follows. We first propose an exact A* search for the LCS problem, which is derived from components and settings that proved already useful in heuristic BS variants as determined in our earlier studies [25,26]. This A* search is shown to be effective for small instances, but as one may expect it has serious scalability issues similar to other exact methods in terms of time and memory requirements when considering larger instances. We therefore extend this A* search by applying two alternative hybrid search strategies from [25], turning the original A* search into effective anytime algorithms for finding an LCS. Both follow the idea of interleaving traditional A* search

iterations with heuristic search – either BS or anytime column search [36] – and they are labeled A* + BS and A* + ACS, respectively. The A* framework ensures completeness and provides upper bounds at any time, while the embedded heuristic search iterations rely on the heuristic guidance function from [26,37] and are responsible for producing a first approximate solution quickly and improving it over time. Most importantly, the heuristic search iterations also operate on the list of open nodes of A* search in order to avoid redundant node expansions.

Although we employ, from a conceptual point of view, the same hybrid search strategies as in [25], we want to emphasize the significant differences between the adaptation to the longest common palindromic subsequence (LCPS) problem in [25] and the adaptation to the LCS problem presented in this paper. Note, for example, that the best exact algorithms for the LCS problem when considering two input strings ($m = 2$) require $O(n^2)$ of time, while the best exact algorithm for the LCPS problem requires $O(n^4)$ time. This already hints that both problems are structurally quite different from each other. These differences lead to the following differences in the adaptation of the algorithmic concepts to both problems:

- The search spaces of the two problems (in terms of the definition of the A* nodes) differ. This is due to the fact that in the LCS problem solutions are generated from left to right, while in the LCPS problem a solution construction starts from the left and from the right at the same time.
- The upper bounds utilized for the two problems are different.
- The expected length calculation heuristics (EX) for guiding the tree search techniques differ, even though similar ideas are used for their derivation.
- Last but not least, the implementations differ in additional details. For example, in this paper we make use of an efficient way of filtering the dominated nodes in BS and ACS iterations (i.e., *the restricted filtering*). This was not possible in [25].

In the comprehensive experimental study of this paper we evaluate the proposed approaches on various LCS benchmark sets from the literature and compare to the so far best methods' results. Earlier computational studies always considered a subset of the available benchmark sets. Concerning proven optimality, our A* search is able to solve 106 instances from the literature, which exceeds the number of solved instances by Top_MLCS by six. Moreover, in most cases our A* search is faster than Top_MLCS. For the remaining instances that cannot be solved to optimality, A*+ACS turns out to be the now leading method for most benchmark sets in respect to final solution quality. Moreover, optimality gaps are considered for larger LCS instances for the first time, and those obtained by A* + ACS are shown to be significantly better than the ones of the other considered approaches on many occasions. Most remarkably, A* + ACS was able to achieve new best known solutions for 82 different LCS instance groups, which corresponds to ≈70% of all the considered instance groups.

The rest of this article is organized as follows. Section 2 gives an overview on essential previous work and definitions required for the A* search for the LCS problem and its anytime variants. The A* search framework is then presented in Section 3, while Section 4 provides the details of the A* + ACS and A* + BS anytime algorithms. Section 5 comprises the whole computational study. Conclusions and ideas for future work are given in Section 6.

2. Previous work

In this section we summarize those aspects of previous work that are needed for understanding the anytime algorithms proposed in this work. Most of this material was already covered in a more detailed way in [26], where we introduced a general BS framework for the LCS problem. Beam Search is a well known incomplete tree search method that works in a limited *Breadth-First Search* (BFS) manner. At each step, it maintains a set of nodes – called the *beam* – from the same level of the search tree. Note, in this context, that each node of the tree corresponds to a partial solution and a respective remaining subproblem, while the leaf nodes correspond to non-extensible solutions. The nodes of the current beam are expanded at each step, generating a set of nodes from the next level of the search tree. This set of extensions is denoted by V_{ext}. Among the nodes from V_{ext}, the $\beta > 0$ most promising nodes are selected and form the beam of the next step, where β is a strategy parameter called *beam width*. Beam Search keeps repeating these steps until the beam is empty, i.e., no further expansions are possible. The initial beam consists just of the root node of the search tree, which corresponds to the empty partial solution. The result of BS is the best solution in the final beam. The general BS framework from [26] covers all BS-based approaches from the literature known to us. In the following, after introducing some notations and additional concepts, two important topics are addressed: (1) the state graph on which both the BS approaches from the literature and the A* search proposed in this work operate, and (2) upper bound functions and a heuristic guidance function. Both upper bound functions and/or heuristic guidance functions are used in BS approaches for the selection of the nodes that are kept as the next iteration's beam.

2.1. Notations and concepts

Let n be the maximum length of the m strings in $S = \{s_1, \ldots, s_m\}$, i.e., $n = \max_{i=1,\ldots,m} |s_i|$. Furthermore, let $s[j]$, $j = 1, \ldots, |s|$, be the jth letter of a string s, and let $s_1 \cdot s_2$ denote the concatenation obtained by appending string s_2 to string s_1. Moreover, $s[j, j']$, $j \leq j'$, denotes the continuous subsequence of s starting at position j and ending at position j'; if $j > j'$, $s[j, j']$ is the empty string denoted by 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* to S, if and only if s is a subsequence of each string in S, that is, a *common subsequence* of S.

Any subproblem of S is defined on the basis of a so-called *left position vector* $p^{\text{L}} \in \mathbb{N}^m$, with $1 \leq p_i^{\text{L}} \leq |s_i|$ for $i = 1, \ldots, m$. In particular, for a given p^{L}, subproblem $S[p^{\text{L}}]$ concerns the substrings $s_i[p^{\text{L}}, |s_i|]$ for all $i = 1, \ldots, m$. In other words, $S[p^{\text{L}}]$ contains the right part of each string from S starting from the position indicated in the left position vector p^{L}. Note that the original problem S can be denoted by $S[p^{\text{L}} = (1, \ldots, 1)]$. Given a (partial) solution s to S—that is, a string s that is a common subsequence of S—a subproblem $S[p^{\text{L}}]$ is induced by defining p^{L} in the following way. For each $i = 1, \ldots, m$, p_i^{L} is determined such that $s_i[1, p_i^{\text{L}} - 1]$ is the minimal-length string among all substrings $s_i[1, p]$, $p = 1, \ldots, |s_i|$, that contain s as a subsequence. For example, given $S = \{\texttt{abcaac}, \texttt{acbaba}\}$ and the partial solution $s = \texttt{aca}$, the induced subproblem $S[p^{\text{L}}]$ is defined by left position vector $p^{\text{L}} = (5, 5)$. Note that there is potentially more than one partial solution inducing the same subproblem, respectively, the same left position vector. In the example above, partial solution $s' = \texttt{aba}$, for example, induces the same subproblem and the same left position vector as partial solution $s = \texttt{aca}$. Moreover, partial solutions inducing the same subproblem and the same left position vector may have different lengths. Considering again the example from above, substring $s'' = \texttt{aa}$ induces the same subproblem and the same left position vector as s and s'.

2.2. State graph for the LCS problem

The state graph that is used by all BS variants known in the literature so far, and which will also be used by the A* search proposed in this paper, is a *directed acyclic* graph $G = (V, A)$, where a node $v = (p^{\mathrm{L},v}, l_v) \in V$ represents the set of partial solutions that (1) have the same length l_v and that (2) induce the same subproblem denoted by $S[p^{\mathrm{L},v}]$ and left partition vector $p^{\mathrm{L},v}$. An arc $a = (v_1, v_2) \in A$ between two nodes $v_1 \neq v_2 \in V$—carrying label $\ell(a) \in \Sigma$—exists, if and only if the following two conditions are fulfilled:

1. $l_{v_2} = l_{v_1} + 1$
2. The partial solution inducing v_2 is produced by appending $\ell(a)$ to the partial solution inducing v_1.

The root node r of G corresponds to the original problem S, which is induced by the empty partial solution denoted by ε. In technical terms, $r = ((1, \ldots, 1), 0)$. In order to derive the successor nodes of a node $v \in V$, we first determine the subset $\Sigma_v \subseteq \Sigma$ of letters that can be used to feasibly extend the partial solutions represented by v. Obviously, Σ_v consists of all letters $a \in \Sigma$ that appear at least once in each string of $S[p^{\mathrm{L},v}]$. For each letter $a \in \Sigma_v$, the position of the first occurrence of a in $s_i[p_i^{\mathrm{L},v}, |s_i|]$ is denoted by $p_{i,a}^{\mathrm{L},v}, i = 1, \ldots, m$. Set Σ_v may frequently be reduced by identifying *dominated* letters: We say that letter $a \in \Sigma_v$ *dominates* letter $b \in \Sigma_v$ if and only if $p_{i,a}^{\mathrm{L},v} \leq p_{i,b}^{\mathrm{L},v}$ for all $i = 1, \ldots, m$. Dominated letters can safely be ignored since they always lead to suboptimal solutions. Let $\Sigma_v^{\mathrm{nd}} \subseteq \Sigma_v$ be the subset of those letters that are non-dominated. Graph G contains for each letter $a \in \Sigma_v^{\mathrm{nd}}$ a successor node $v' = (p^{\mathrm{L},v'}, l_v+1)$ of v, where $p_i^{\mathrm{L},v'} = p_{i,a}^{\mathrm{L},v'} + 1, i = 1, \ldots, m$. A node v that has no successor node – that is, when $\Sigma_v^{\mathrm{nd}} = \emptyset$ – is called a *non-extensible* node. Now, note that any path from the root node r to any node in $v \in V$ represents the feasible partial solution obtained by collecting and concatenating the labels of the traversed arcs.[1] Any path from r to a non-extensible node represents a common subsequence of S that cannot be further extended, and any longest path from r to a non-extensible node represents an optimal solution to problem instance S.

2.3. Upper bounds

The BS approaches from the literature make use of different upper bounds for the lengths of LCS (sub-)problems in order to select promising nodes for the beam of the next step. We consider here the most successful bounds. The upper bound from Blum et al. [21] was derived by tightening the bound from Fraiser [19]. Given a node v representing the left position vector $p^{\mathrm{L},v}$ and the corresponding subproblem $S[p^{\mathrm{L},v}]$, this bound calculates the minimal number of occurrences of each letter over all the strings of subproblem $S[p^{\mathrm{L},v}]$ and returns the sum of these, i.e.,

$$\mathrm{UB}_1(v) = \mathrm{UB}_1(p^{\mathrm{L},v}) = \sum_{a \in \Sigma} \min_{i=1,\ldots,m} |s_i[\, p_i^{\mathrm{L},v}, |s_i|\,]|_a. \tag{1}$$

This bound can be efficiently determined in $O(m \cdot |\Sigma|)$ time when using a suitable data structure that is initialized in preprocessing (for more details see [25]).

A DP-based upper bound was introduced by Wang et al. [24], making use of the LCS calculation for two input strings. For each pair $\{s_i, s_{i+1}\} \subseteq S, i = 1, \ldots, m-1$, a so-called scoring matrix M_i is constructed in a pre-processing step, where an entry $M_i[p, q]$, with $p = 1, \ldots, |s_i|$ and $q = 1, \ldots, |s_{i+1}|$, stores the length of the LCS of strings $s_i[p, |s_i|]$ and $s_{i+1}[q, |s_{i+1}|]$. Given a node v representing the left position vector $p^{\mathrm{L},v}$ and the corresponding subproblem $S[p^{\mathrm{L},v}]$, the upper bound is then calculated as

$$\mathrm{UB}_2(v) = \mathrm{UB}_2(p^{\mathrm{L},v}) = \min_{i=1,\ldots,m-1} M_i[p_i^{\mathrm{L},v}, p_{i+1}^{\mathrm{L},v}]. \tag{2}$$

Not considering the pre-processing step, this bound can be calculated in $O(m)$ time. As in general neither UB_1 dominates UB_2 nor does UB_2 dominate UB_1, it makes sense to also consider the combined bound $\mathrm{UB}(v) = \min\{\mathrm{UB}_1(v), \mathrm{UB}_2(v)\}$.

2.4. Approximate expected length calculation of an LCS

Some of the BS approaches make use of a heuristic guidance function instead of an upper bound for the selection of the nodes that form the beam of the next iteration. In the following we briefly describe the one that we introduced in [26]. This heuristic guidance function is based on a DP recursion by Mousavi and Tabataba [22], which calculates the probability that any string of length p is a subsequence of a *uniform* random string of length q, for $0 \leq p, q \leq n$. Let us assume that these probabilities are stored in a matrix P with elements $\mathrm{P}[p, q] \in [0, 1],\ 0 \leq p, q \leq n$.

The heuristic guidance function makes two strong assumptions: (1) the strings from S are all uniform random strings, and (2) the input strings are independent from each other. For each sequence x of length k over alphabet Σ we denote by Ev_x the event that x is a common subsequence of the input strings from S. For any two sequences x and y, $x \neq y$, we additionally make the simplifying assumption that the events Ev_x and Ev_y are independent (which does not hold in reality). By applying some basic laws from probability theory, the heuristic guidance function which—given a node v representing the left position vector $p^{\mathrm{L},v}$ and the corresponding subproblem $S[p^{\mathrm{L},v}]$—approximates the expected length of a LCS in the corresponding subproblem can be stated as

$$\mathrm{EX}(v) = l_{\min} - \sum_{k=1}^{l_{\min}} \left(1 - \prod_{i=1}^{m} \mathrm{P}(k, |s_i| - p_i^{\mathrm{L},v} + 1)\right)^{|\Sigma|^k}, \tag{3}$$

where $l_{\min} = \min\{|s_i| - p_i^{\mathrm{L},v} + 1 \mid i = 1, \ldots, m\}$. An extensive computational study presented in [26] has shown that this heuristic guidance function has a significant impact on finding high-quality solutions for many instances. The general conclusion was that its use in BS approaches is generally preferred over the use of upper bound functions in the context of benchmark instances in which the input strings are quasi-independent.

3. A* search framework

As mentioned before, our focus in this work is on the development of A*-based anytime algorithms for the LCS problem. A* search [38] is a well-known exact technique widely used in path-finding and planning. It belongs to the class of informed search methods, employing a *best-first search* strategy. Our A* search for the LCS problem operates on the state graph G as defined in Section 2.2. At each iteration the most promising not-yet-processed/expanded (*open*) node is expanded. To this end, each node v reached is evaluated by a priority function $f(v) := g(v) + h(v)$, where $g(v)$ denotes the cost of the best path from the root node r to v and $h(v)$ is a heuristic function that estimates the *cost-to-go*, the cost of the best path from v to a *goal* node. In the context of the LCS, the cost of a path refers to its length, and a path is better the longer it is. Furthermore, any non-extensible node of the state graph represents a goal node. Good candidates

[1] We emphasize that it is not necessary to store actual partial solutions s in the nodes. A longest path to any node in the graph starting from the root node and the respective partial solution can be efficiently derived in a backward manner by iteratively identifying a predecessor in which the l_v-value always decreases by one.

for $h(\cdot)$ in the context of the LCS are upper bound functions, such as the ones discussed in Section 2.3. They never underestimate the length of the best/longest path to a goal node and are called *admissible* in the terminology of A* search. Using an admissible heuristic function guarantees that an optimal solution is found when a goal node is finally selected for expansion and the search terminates. Moreover, the proposed upper bounds are *monotonic*, i.e., the upper bound of any extension of a node is always at most as high as the upper bound of the original node. This property guarantees that re-evaluations of already expanded nodes are never necessary. To efficiently retrieve the node with highest priority in each iteration, A* search maintains all open nodes in a priority queue Q. Additionally, our A* search maintains a hash map N with left position vectors $p^{L,v}$ as keys mapping to the respective l_v-values. By this data structure, we can efficiently recognize already reached left position vectors.

A* search starts with the initialization of Q, that is, $Q = \{r\}$. At each iteration, it expands the top node of Q by generating the respective successor nodes. If its left position vector is not already present in N, a successor node is added to N and to Q. If, on the other side, the successor's left position vector is already in N, it is checked if the new path to v is longer than the already known one. If this is the case, the l_v-value of v is updated correspondingly, and the priority of v (used for its ranking in Q) is adapted accordingly. The algorithm keeps expanding the top nodes of Q until optimality is reached by selecting a goal node or either the memory limit or a time limit is exceeded. One potential problem is that Q typically contains many nodes with the same priority value. These ties are broken by prioritizing those nodes which are farther away from the root node, i.e., the ones with higher l_v values. Remaining ties are broken with the help of a k-norm of the remaining string lengths, i.e., each node is evaluated by $\kappa(v) = \left(\sum_{i=1}^{m}(|s_i| - p_i^{L,v} + 1)^k\right)^{\frac{1}{k}}$. These $\kappa(v)$-values can be seen as a rough heuristic indicator for the cost-to-go, nodes with larger values are expected to be more promising. We used $k = 0.5$ in our implementation. A pseudo-code of our A* search for the LCS problem is given in Algorithm 1. Note that an alternative A* algorithm was proposed in [24]. However, this simpler algorithm uses just the weaker upper bound function UB_2 to guide the search, does not consider tie breaking, and has a larger memory footprint.

Algorithm 1 A* for the LCS problem.

```
N: hash map for all reached left position vectors with the lengths of the longest paths; Q: priority queue with all open nodes
p^{L,r} ← (1, ..., 1)
r ← (p^{L,r}, 0)
N[p^{L,r}] = 0
Q = {r}
while time and memory limit not exceeded and Q is not empty do
    v ← pop the top node from Q
    Σ_v^nd ← non-dominated feasible letters concerning sub-problem S[p^{L,v}]
    if Σ_v^nd = ∅ then      // v is a goal node
        return optimal solution s_lcs retrieved from v
    else
        for all a ∈ Σ_v^nd do      // expand v
            p_i^{L,v'} ← p_{i,a}^{L,v'} + 1, i = 1, ..., m
            l_{v'} ← l_v + 1
            if p^{L,v'} ∈ N then
                if N[p^{L,v'}] < l_{v'} then      // a better path to the node was found
                    N[p^{L,v}] ← l_{v'}
                    update priority value of node v in Q
                end if
            else      // a new node
                f_{v'} ← l_{v'} + UB(v')
                add v' of the priority f_{v'} to Q
                add v' to N
            end if
        end for
    end if
end while
return empty solution ε
```

4. Anytime algorithms for the LCS problem

When faced with large-size problem instances of hard optimization problems, pure exact approaches such as DP or A* search frequently reach their limits. Moreover, if not given enough time (or space) to terminate, these algorithms are not able to provide sub-optimal solutions of reasonable quality. Therefore, the optimization community has, at some point, started to improve such algorithms by adding mechanisms that allow them to be terminated early and nevertheless provide feasible solutions of reasonable quality. The phrase *anytime algorithms* was used for the first time in the literature by Dean and Boddy [39,40] in the middle of the 80's, referring to a class of algorithms that is able to find an initial approximate solution quickly and then improves upon it over time, until optimality is finally guaranteed if enough time is given. These algorithms are nowadays widely used in planning and intelligent systems and domains such as real-time diagnosis and repair, mobile robot control, and signal interpretation. In these domains, solutions must frequently be obtained rather quickly but sometimes more time can be spent [31,32]. Anytime algorithms are flexible in terms of time and resources used. They are designed to offer a selectable trade-off between solution quality and computational requirements.

Two important groups of anytime algorithms are those based on BS and on A* search, respectively. The BS-based anytime algorithms are generally characterized by repeated applications of BS (with a reduced beam width) in which the initial beam of a subsequent iteration is, in some way, based on nodes that were discovered but not processed in earlier iterations. One of the first approaches of that kind was proposed by Zhang [41]. This algorithm was able to reach high-quality solutions earlier than standard BS, in addition to being able to produce optimal solutions if given enough computational resources. Another example is *beam stack search* [42], which integrates systematic backtracking within BS. On the other side, a lot of work has been dedicated to the development of A*-based anytime approaches. Hansen and Zhou [43] presented *anytime weighted* A*. The main idea of this algorithm is to change the evaluation function to $f(v) = g(v) + w \cdot h(v)$, where $w \geq 1$ is a weight parameter. In general, a larger weight will yield a shorter run and cruder solution as the heuristic guidance is inflated, while smaller weights will yield better solutions at the cost of longer runtimes. Thus, the search is typically iteratively performed with decreasing weights. Likhachev et al. [44] proposed the *Anytime Repairing* A* (ARA*) algorithm, which extends anytime weighted A* by reusing the results of an iteration in the subsequent one; it is therefore significantly more efficient. In order to get rid of the sensitive weight parameter, *Anytime Non-parametric* A* (ANA*) was proposed by Berg et al. [45]; here, the greediness of the search is adapted as the quality of the solutions found improves. Aine et al. [46] suggested *Anytime Window* A* (AWA*), where they set a range

for the levels from which the nodes of the corresponding open lists are only allowed to be expanded and by which they enforce convergence to a sub-optimal solution at each iteration. This range (i.e., window) is adapted at each iteration to produce an improved solution. A memory bounded variant of AWA*, called MAWA*, was proposed in [47].

Later, Vadlamundi et al. [36] described *Anytime Column Search* (ACS). Here, the nodes of the state graph are organized into layers. The algorithm maintains a separate priority queue for each of these layers. At each iteration of ACS, which is performed in a level-by-level manner, up to β of the best nodes of each level of the state graph are expanded. An algorithm based on a similar idea was presented by Kao et al. [48]. We will build upon ACS in Section 4.2, where we will also introduce it in more detail. More recently, Vadlamundi et al. [49] proposed *Anytime Pack Search* (APS), which maintains a container of *pack* > 0 nodes – where *pack* is a parameter of the algorithm – and a priority queue Q of not-yet-expanded nodes. At each iteration, standalone BS runs are successively performed for which the initial beam is composed of the first *pack* nodes from the top of priority queue Q, until Q becomes empty. The authors showed APS to be a state-of-the-art approach for three different problem domains [49]. Therefore, we decided to consider APS for comparison purposes also in the current work for finding a LCS.

The two new anytime algorithms that we present in this work for the LCS problem are based on the A* framework from Section 3. Our main idea is to embed efficient heuristic approaches into the A* framework which are repeatedly executed inbetween regular A* iterations. Our A* anytime variants – apart from providing excellent solutions – are able to return proven gaps at almost any time when terminated prematurely.

4.1. A* + BS approach

Since BS approaches are the state-of-the-art heuristic techniques for the LCS problem but A* search is more promising when it comes to solving smaller instances to proven optimality, it seems sensible to combine A* search with BS into an anytime search method denoted by A* + BS. We presented the basic idea of such a hybrid search strategy originally in [25], where it was applied to the longest common palindromic subsequence problem. At the start of A* + BS, a run of BS with small width is performed for which the beam is initialized with the root node r. This initial BS run takes place to obtain a first reasonable approximate solution (and thus a primal bounds) rather quickly. Then the algorithm proceeds by iteratively applying the following scheme. First, δ traditional iterations of A* search are performed, with $\delta > 0$ being a strategy parameter. Second, a BS run is applied in which the first beam is initialized with the top node of Q. The algorithm stops once optimality is proven or a memory limit, respectively time limit, is exceeded. To avoid redundant recalculations, all the embedded BS calls and the A* search act on the same search tree. All non-expanded nodes encountered during a BS run are used to update the hash map N and are inserted into the priority queue Q (if not already there). Moreover, if a new best path to some node is encountered within any BS iteration, an update of the corresponding node in N is performed by changing the key to the new l_v-value, and the node is then added into the corresponding beam, that is, the nodes which were already encountered during the search are allowed to be added into V_{ext}.

A pseudo-code for A* + BS is provided in Algorithm 2. Parameters $\beta > 0$ (beam width of BS) and $\delta > 0$ (frequency of BS applications) control the balance between BS and classical A* search iterations and thus the emphasis on improving the primal bound versus the dual bound, respectively. Beam search makes use of a function `Filter`$(V_{\text{ext}}, k_{\text{filter}})$ for filtering dominated successor nodes at each step. This procedure works as follows. Up

Algorithm 2 A* + BS for the LCS problem.

1: N: hash map for all reached left position vectors with the lengths of the longest paths; Q: priority queue of not yet expanded nodes; $\beta > 0$: beam width; $\delta > 0$: number of consecutive A* iterations; $k_{\text{filter}} \geq 0$: extent of filtering
2: $s_{\text{best}} \leftarrow \varepsilon$
3: $p^{\text{L},r} \leftarrow (1, \ldots, 1)$
4: $r \leftarrow (p^{\text{L},r}, 0)$
5: $N[p^{\text{L},r}] = 0$
6: $Q \leftarrow \{r\}$
7: $opt \leftarrow false$
8: **while** not *opt* **and** neither *memory limit* nor *time limit* exceeded **do**
9: $B \leftarrow$ pop the β top nodes from Q
10: **while** $B \neq \emptyset$ **do**
11: // perform BS:
12: **for** all $v \in B$ **do**
13: `ExpandNode`(v) // see Algorithm 3
14: store respective children of v in V_{ext}
15: **end for**
16: `Filter`$(V_{\text{ext}}, k_{\text{filter}})$ // filter dominated nodes from V_{ext}
17: $B \leftarrow$ Reduce(V_{ext}, β)
18: **end while**
19: $iter \leftarrow 0$
20: **while** $iter < \delta$ **and** neither *memory limit* nor *time limit* exceeded **do**
21: // perform A* iteration:
22: $v \leftarrow$ get top node from Q
23: remove v from Q
24: `ExpandNode`(v) // see Algorithm 3
25: $iter \leftarrow iter + 1$
26: **end while**
27: **end while**
28: return s_{best}

to k_{filter} of the most promising nodes are selected from V_{ext} as a reference set. Then, all other nodes from V_{ext} that are dominated by at least one of these reference solutions are removed from V_{ext}. If $k_{\text{filter}} = 0$, no filtering is applied. Moreover, the employed BS uses the upper bound UB from Section 2.3 in order to choose up to β nodes for the beam of the next step. Finally, note that the A* search framework ensures completeness of the A* + BS algorithm and provides proven gaps at any time.

The procedure `ExpandNode` for the expansion of a node and updating the respective data structures is provided in Algorithm 3 (for now, disregard the lines marked to be relevant only for A* + ACS). If the node to be expanded is a goal node, it is checked if it yields a new best solution. If this is the case, s_{best} is updated accordingly. Moreover, if the length of the so-far best solution s_{best} is greater or equal to the f–value of the top node in Q, the flag *opt* is set to *true*, meaning that the search terminates with proven optimality of s_{best}.

4.2. A* + ACS approach

As mentioned above, each BS run in A* + BS starts from the current top node of Q. This means that each BS run only deals with extensions of this single node, and consequently the search space is rather restricted. In particular, many other highly promising nodes at different levels of the state graph may have already been identified, but they are ignored. In order to deal with this potential short-coming, we developed an alternative approach in the line of [25] in which BS runs are exchanged by

Algorithm 3 ExpandNode(v).

1: **Input:** a node v to be expanded; a flag parameter
2: **Uses resp. updates:** s_{lcs}, N, Q and if called from A* + ACS, Q_j, $j = 0, \ldots, j_{\max}$;
3: **if** $\Sigma_v^{\text{nd}} = \emptyset$ **then** // v is a *complete* node
4: $s \leftarrow$ derive the non-extensible solution corresponding to v
5: **if** $|s_{\text{lcs}}| < |s|$ **then** // update best sol.
6: $s_{\text{lcs}} \leftarrow s$
7: **end if**
8: **else**
9: **for** all $a \in \Sigma_v^{\text{nd}}$ **do** // expand v
10: $p_i^{\text{L},v'} \leftarrow p_{i,a}^{\text{L},v'} + 1, i = 1, \ldots, m$
11: $l_{v'} \leftarrow l_v + 1$
12: **if** $p^{\text{L},v'} \in N$ **then**
13: **if** $N[p^{\text{L},v'}] < l_{v'}$ **then** // a better path to the node encountered
14: $N[p^{\text{L},v'}] \leftarrow l_{v'}$
15: update priority of the corresponding node in Q;
16: **if** called from A* + ACS **then**
17: move node v' from Q_{l_v} to $Q_{l_{v'}}$
18: **end if**
19: **end if**
20: **else** // create new node
21: add v' to N
22: $f_{v'} \leftarrow l_{v'} + \text{UB}(v')$
23: add v' with priority $f_{v'}$ to Q
24: **if** called from A* + ACS **then**
25: $e_{v'} \leftarrow \text{EX}(v')$
26: add v' with priority $e_{v'}$ to $Q_{l_{v'}}$
27: **end if**
28: **end if**
29: **end for**
30: **end if**
31: **if** $|s_{\text{lcs}}| \geq \max_{v \in Q} f(v)$ **then**
32: $opt \leftarrow true$
33: **end if**

major iterations of the above already mentioned *Anytime Column Search* (ACS) [36]; this hybrid approach is henceforth labeled A* + ACS.

Anytime column search is an iterative algorithm which maintains for each level j of the state graph a priority queue Q_j that stores – in the context of the LCS problem – all open nodes v with $l_v = j$, $j = 0, \ldots, j_{\max}$, $j_{\max} = \text{UB}(r)$. Initially, Q_0 contains the root node r and the other priority queues are empty. Each *major iteration* of ACS considers all levels $j = 0, \ldots, j_{\max}$ with non-empty queues Q_j in turn, and expands β nodes (or less if Q_j is shorter. The procedure terminates with an optimal solution once all priority queues are empty. Note that ACS in general finds heuristic solutions very quickly since each major iteration identifies usually at least one non-extensible heuristic solution.

The main idea for combining A* with ACS consists again in interleaving classical A* iterations with major ACS iterations. Hereby, A* keeps working on the basis of priority list Q and the priority function that utilizes the upper bound function $\text{UB}(v)$. In this way, the whole approach will maintain the completeness of classical A* search and $\max_{v \in Q} f(v)$ always is a true upper bound for the optimal solution value. In contrast to Q, the heuristic guidance function EX from Section 2.4 is used as sorting criterion for the nodes in the level-specific ACS-queues Q_j. Remember that EX is usually a more promising guidance to find good heuristic solutions, but as it is no valid upper bound, it cannot be used for proving optimality. Moreover, note that changes made in

Algorithm 4 A* + ACS for the LCS problem.

1: N: hash map for all reached left position vectors with the lengths of the longest paths; Q: priority queue of not yet expanded nodes; Q_j: priority queues maintained for each level j of the state graph; $\beta > 0$: a beam width; $\delta > 0$: amount of consecutive A* iterations; $k_{\text{filter}} \geq 0$: extent of filtering
2: $s_{\text{best}} \leftarrow \varepsilon$
3: $p^{\text{L},r} \leftarrow (1, \ldots, 1)$
4: $r \leftarrow (p^{\text{L},r}, 0)$
5: $N[p^{\text{L},r}] = 0$
6: $Q \leftarrow \{r\}$; $Q_0 \leftarrow \{r\}$
7: $opt \leftarrow false$
8: **while** not opt **and** neither *memory limit* nor *time limit* exceeded **do**
9: $lev \leftarrow 0$
10: **while** $lev < j_{\max}$ **do**
11: // perform ACS iteration:
12: $b \leftarrow 0$
13: $V_{\text{ext}} \leftarrow \emptyset$
14: **while** $Q_{lev} \neq \emptyset$ **and** $b < \beta$ **do**
15: $v \leftarrow$ get the top node from Q_{lev}
16: remove v from Q_{lev} and Q
17: ExpandNode(v) // see Algorithm 3
18: Store respective children of v in V_{ext} // keep track of nodes for filtering
19: $b \leftarrow b + 1$
20: **end while**
21: Filter(V_{ext}, k_{filter}) // filter dominated nodes from V_{ext}
22: $lev \leftarrow lev + 1$
23: **end while**
24: $iter \leftarrow 0$
25: **while** $iter < \delta$ **and** neither *memory limit* nor *time limit* exceeded **do**
26: // perform A* iteration:
27: $v \leftarrow$ top node from Q
28: remove v from Q and Q_{l_v}
29: ExpandNode(v) // see Algorithm 3
30: $iter \leftarrow iter + 1$
31: **end while**
32: **end while**
33: return s_{best}

priority queue Q must be accompanied by corresponding changes in priority queues Q_j and vice versa. To enable a direct lookup of priority queue entries for a given node, we make use of the corresponding hash map N.

The pseudo-code of the A* + ACS is presented in Algorithm 4. Note that at each entry of the main while loop (lines 8–32), the algorithm first executes one major iteration of ACS (lines 10–31) and afterwards δ classical A* iterations (lines 22–28). Note that, just like A* + BS, the algorithm potentially makes use of filtering when case $k_{\text{filter}} > 0$ during the major iterations of ACS (line 21). The only difference is that nodes removed from V_{ext} due to filtering are not only removed from N and Q but also from the corresponding queue Q_j. Parameters β and δ play the same role as in A* + BS, namely, controlling the balance between finding good heuristic solutions and improving the dual bound over time. Finally, A* + ACS terminates either with a proven optimal solution, or once the memory limit or the time limit is exceeded, returning the best non-extensible solution found up to this point.

5. Experimental evaluation

In the following we first provide a summary of the algorithms that are considered for the experimental evaluation. These are our two anytime algorithms (1) A* + BS and (2) A* + ACS, (3) the APS algorithm from [49], which is one of the state-of-the-art anytime variants from literature that we implemented for comparison purposes, and (4) A* + ACS-dist which is the variation of A* + ACS in which the heuristic guidance function EX is replaced by the dist(·) estimation from Pro-MLCS [33] and SA-MLCS [34]. Unfortunately, we were not able to do a full comparison to Pro-MLCS and SA-MLCS as the codes could not be obtained from the authors and the description of the special multi-dimensional tree data structure for determining dominated solutions is insufficient for a re-implementation. However, A* + ACS-dist without the classical A* iterations in (i.e., when setting $\delta = 0$) almost corresponds to Pro-MLCS except that instead of the multi-dimensional data structure from [33], Filter(·, ·) is used for filtering dominated solutions.

All algorithms were implemented in C++ and the experiments were conducted in single-threaded mode on a machine with an Intel Xeon E5–2640 processor with 2.4 GHz allowing a maximum of 32 GB of memory. The maximum computation time for each run was limited to 900 s. The considered algorithms were evaluated by the quality of the best solutions they provided and by the *percentage gaps*, which are calculated at time $t > 0$ as $gap(t) := \frac{\text{ub}(t) - |s_{\text{best}}(t)|}{\text{ub}(t)} \cdot 100\%$, where $s_{\text{best}}(t)$ denotes the best solution found up to time t and $\text{ub}(t)$ denotes the upper bound on the length of an optimal solution obtained from the f-value of the top node in Q at time t (or the optimal solution value when already available).

5.1. Benchmark instances

The related literature on the LCS problem offers six public benchmark sets for the LCS problem. We used all of them for the experimental evaluation here. The BL benchmark [27] consists of 450 problem instances grouped by different values for the number of input strings (m), the maximum length of the input strings (n), and the alphabet size ($|\Sigma|$). For each combination of m, n, and $|\Sigma|$ the set offers ten instances generated uniformly at random. Furthermore, Rat and Virus are two benchmarks with a biological background, consisting of 20 instances each. Random is another rather small benchmark set consisting of 20 randomly generated instances. The latter three benchmark sets were introduced in [50]. Moreover, benchmark set ES, which was introduced in [51], is a large set with 600 instances grouped by different combinations of values for m, n, and $|\Sigma|$, where each group includes 50 instances. Last but not least, benchmark set BB—introduced in [52]—consists of 800 artificially created instances. These were generated in a way such that input strings have a larger similarity to each other than to random strings: Given m, $|\Sigma|$, and an additional parameter $l > 0$, first a base string of length l was generated uniformly at random over alphabet Σ. Then, each of the m strings of the instance were derived by passing over the base string and removing each character with a probability of 10%. Thus, an input string of an instance with $l = 1000$ is, on average, of length 900. We only considered the 80 largest instances from this set, based on $m \in \{10, 100\}$ and $|\Sigma| \in \{4, 8, 16, 32\}$ and $l = 1000$. There are ten instances for each combination of m and $|\Sigma|$. Note that the instances in this last set are the only ones with a clear relation between the input strings, that is, where input strings are not independent of each other.

5.2. Tuning of the algorithms' parameters

In order to ensure a fair comparison, the parameters of all considered algorithms were tuned by irace [53]. This tuning took place under the same conditions (computation time limit: 900 s; memory limit: 32 GB) as later the final experimental evaluation. After conducting some preliminary experiments, we decided to use the following domains for the values of the parameters for the tuning:

- $\delta \in \{0, 1, 10, 50, 100, 500, 1000, 5000, 10000, 20000, 50000\}$,
- $k_{\text{filter}} \in \{0, 1, 10, 50, 100, 500, 1000, 5000, 10000, +\infty\}$,
- $\beta \in \{1, 50, 100, 500, 1000, 5000, 10000, 20000\}$,

Since the parameter *pack* of APS refers to the beam width of that algorithm, we chose the same domain for *pack* and β. As we expected potentially stronger differences in suitable settings for the dependent instances BB and the quasi-independent other instances, we decided to apply tuning for these two instance categories separately. We used 40 additional randomly generated instances for the tuning process aimed for quasi-independent instances. The budget of irace was set to 5000 optimization runs in this case. On the other side, we generated 20 additional dependent instances for tuning purposes, in the same way as reported in [52]. The budget of irace was set to 1000 optimization runs in this second case. In addition to the separation concerning the instance type – quasi-independent versus dependent – we applied for each instance type two tuning runs with different aims. One of these tuning runs aimed for final solution quality, and the other one for small dual gaps. The results of these four tuning runs are reported in the four sub-tables of Table 1.

Concerning the tuning results for the quasi-independent instances, note that a higher beam size β is necessary when aiming for solution quality. On the other hand, when we focus on small dual gaps, the amount of A* iterations (δ) has to be significantly increased for all algorithms in comparison. This result appears conclusive when considering that classical A* iterations are primarily important for improving the dual bound. Concerning the tuning results for the dependent instances, we can also notice the requirement of a higher value for δ when aiming for small gaps. Rather interesting is the large value required for β in the case of A* + BS and APS when aiming for solution quality. In contrast, the tuning procedure has yielded a lower beam size for algorithms A* + ACS and A* + ACS-dist.

5.3. Experimental evaluation: Exact Solving with classical A* search

Initial tests indicated that our classical A* search is only meaningfully applicable to the smallest instances of the benchmark sets, that is, the instances with string length $n = 100$ from set BL. The corresponding results can be found in Table 2, in which we compare the proposed A* approach to the exact solver Top_MLCS [29]. In our comparison we made use of the original implementation of Top_MLCS provided by the authors.[2] We remark that Top_MLCS can effectively exploit a parallel hardware architecture, but we performed it in single threaded mode in order to ensure a fair comparison with our A* approach. Besides the instance characteristics, Table 2 lists average solution lengths $\overline{|s|}$, average times $\bar{t}$ in seconds until proven optimality has been reached, and the number of instances that could be solved to optimality #opt (out of ten per line) for both approaches.

From Table 2 it can be observed that all problem instances with $|\Sigma| \geq 12$ are solved – by both algorithms – to proven

[2] The source code of Top_MLCS can be found at https://github.com/dxslin/mlcs.

Table 1
Tuning results.

(a) Tuning for solution quality on quasi-independent instances.

	Algorithm			
Parameter	A* + BS	A* + ACS	A* + ACS-dist	APS
δ	50	1	100	–
β	500	500	100	–
k_{filter}	1	1	0	0
pack	–	–	–	500

(b) Tuning for small gaps on quasi-independent instances.

	Algorithm			
Parameter	A* + BS	A* + ACS	A* + ACS-dist	APS
δ	10000	1000	500	–
β	500	1	1	–
k_{filter}	100	0	0	100
pack	–	–	–	500

(c) Tuning for solution quality on dependent instances.

	Algorithm			
Parameter	A* + BS	A* + ACS	A* + ACS-dist	APS
δ	500	500	100	–
β	1000	1	1	–
k_{filter}	1000	1	0	1000
pack	–	–	–	1000

(d) Tuning for small gaps on non-independent instances.

	Algorithm			
Parameter	A* + BS	A* + ACS	A* + ACS-dist	APS
δ	5000	20000	20000	–
β	1000	50	100	–
k_{filter}	1000	100	100	5000
pack	–	–	–	1000

optimality, and runtimes are typically only a fraction of a second. However, A* needs significantly less time especially for the instances with $|\Sigma| = 12$. Additionally, our A* approach solved six (out of ten) instances with $|\Sigma| = 4$ and $m = 10$ to proven optimality,[3] while Top_MLCS was not able to do so due to running out of memory. None of the instances with $|\Sigma| = 4$ and $m \geq 50$ could be solved to optimality by the two algorithms due to the memory limit.

In summary, A* is able to solve 106 instances from the literature to proven optimality. At this point we would like to stress that the mixed integer linear programming solver CPLEX in version 12.9 applied to the LCS model from [27] could not solve any of these instances due to a too large number of variables and constraints.

5.4. Experimental evaluation: Anytime algorithms

In contrast to the classical A* search, the anytime algorithms studied in this work are able to yield meaningful results on all problem instances. Remember that we aim to compare A* + BS and A* + ACS with APS and A* + ACS-dist. Additional reason why the Pro-MLCS [33] and SA-MLCS [34] are not considered in this comparison is because they are not designed for providing gaps upon premature termination. Moreover, we would like to emphasize that – as observed in [26] – the main factor for obtaining high quality solutions is the heuristic guidance function. For this reason we study algorithm A* + ACS-dist which makes use of the heuristic guidance function $\texttt{dist}(v) = \sum_{i=1}^{m} p_i^{L,v}$ from Pro-MLCS and SA-MLCS. As already mentioned, when setting $\delta = 0$ in A* + ACS-dist, we get reasonably close to the original Pro-MLCS algorithm.

[3] All ten instances with $m = 10$, $n = 100$, $|\Sigma| = 4$, could be solved by A* when increasing the memory limit to 40 GB.

Table 2
Classical A* search: average results for benchmark BL, $n = 100$.

		A*			Top_MLCS		
m	$\lvert\Sigma\rvert$	$\overline{\lvert s\rvert}$	$\bar{t}[s]$	#opt	$\overline{\lvert s\rvert}$	$\bar{t}[s]$	#opt
10	4	20.5	428.33	6	0.0	–	0
	12	12.7	1.73	10	12.7	5.2	10
	20	7.9	0.08	10	7.9	0.28	10
50	4	0.0	–	0	0.0	–	0
	12	6.9	0.17	10	6.9	0.46	10
	20	3.0	0.06	10	3.0	0.08	10
100	4	0.0	–	0	0.0	–	0
	12	5.2	0.08	10	5.2	0.23	10
	20	2.1	0.07	10	2.1	0.08	10
150	4	0.0	–	0	0.0	–	0
	12	4.7	0.07	10	4.7	0.16	10
	20	1.9	0.08	10	1.9	0.08	10
200	4	0.0	–	0	0.0	–	0
	12	4.1	0.07	10	4.1	0.18	10
	20	1.1	0.06	10	1.1	0.11	10

In the following we report on results both concerning the obtained (average) solution quality and (average) gaps. For improving the readability result tables for benchmark sets Rat, Virus, ES, and BB are given in the main text, whereas the tables for Random and BL are provided in Appendix A. More specifically, the results concerning solution quality of the first four data sets can be found in Tables 3–6, while the corresponding results concerning the gaps are presented in Tables 7–10. The first three table columns indicate the characteristics of the considered sub-groups of the benchmark sets in terms of $|\Sigma|$, m, and n. Subsequently, the results of the four algorithms are presented. Each of these four blocks consists of four columns listing the following information: the average solution quality ($\overline{|s|}$), the average gaps ($\overline{gap}\,[\%]$), the average time at which the best solution was found ($\overline{t_{\text{best}}}\,[s]$), and the average total runtime ($\bar{t}\,[s]$). The tables showing the results with the parameter settings aiming for solution quality have an additional column labeled *lit. best* that reports the best-known result from the literature for the respective instance, or instance group (without considering the results from the current work). Asterisks in the solution quality column indicate that the best-known result from the literature was beaten. The best result concerning the comparison of the four algorithms considered in this work is always indicated in boldface. Note that in tables presenting results obtained with parameter settings aiming for solution quality, this concerns the columns on the average solution quality. While in tables presenting results obtained with parameter settings aiming for small gaps, this concerns the columns listing the average gaps.

A study of the numerical results allows to draw the following conclusions:

- A* + ACS generally outperforms the three competitors in terms of solution quality in the context of instances with quasi-independent input strings (that is, benchmark sets Rat, Virus, Es, Random and BL). The only exception is benchmark set BB, in which instances consist of dependent input strings. The reason for this behavior is clearly that the heuristic guidance function EX() works in general very well for instances with quasi-independent input strings, while it tends to mislead for instances with related input strings. Observe in Table 6 that the performance of A* + ACS and of A* + ACS-dist strongly decreases, especially when the instances consist of many input strings ($m = 100$). A more visual presentation of the results is provided in Figs. 5–8 in Appendix B, where the improvement of A* + ACS over the three competitors is shows in percent.

Table 3
Rat benchmark. Results when aiming for solution quality.

			A* + BS				A* + ACS				APS				A* + ACS-dist				*lit. best*
$\|\Sigma\|$	m	n	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{best}}$[s]	$\bar{t}$[s]	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{best}}$[s]	$\bar{t}$[s]	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{best}}$[s]	$\bar{t}$[s]	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{best}}$[s]	$\bar{t}$[s]	$\overline{\|s\|}$
4	10	600	197	41.7	2.15	685.6	*__206__	39.4	130.6	900.0	197	41.0	866.9	900.0	204	39.5	98.2	731.2	205
4	15	600	180	47.8	11.6	735.3	*__189__	45.5	740.1	900.0	181	47.5	130.3	900.0	186	45.6	75.4	603.1	185
4	20	600	166	43.3	29.5	900.0	*__174__	41.2	12.3	900.0	167	42.2	420.9	900.0	171	41.0	71.7	776.0	172
4	25	600	166	51.3	74.4	684.2	*__173__	49.4	38.3	900.0	166	50.4	212.3	900.0	170	49.4	57.6	642.5	170
4	40	600	152	50.0	570.7	900.0	*__154__	49.7	32.8	900.0	151	49.8	183.0	900.0	150	50.3	6.5	755.4	153
4	60	600	148	55.6	186.4	900.0	*__154__	54.0	510.3	900.0	148	55.3	129.1	900.0	151	54.4	384.7	893.9	152
4	80	600	136	52.3	190.8	900.0	*__144__	49.8	427.9	900.0	137	50.9	308.0	900.0	126	55.0	0.6	754.5	142
4	100	600	134	52.0	180.9	900.0	*__139__	50.7	458.7	900.0	134	51.6	31.9	900.0	132	52.5	421.7	809.7	137
4	150	600	123	44.3	29.9	900.0	*__131__	41.0	39.2	900.0	124	43.6	89.8	900.0	110	50.2	848.4	900.0	129
4	200	600	121	46.9	20.8	900.0	*__126__	45.0	288.0	900.0	121	46.8	23.8	900.0	105	53.7	821.4	900.0	123
20	10	600	69	63.1	5.4	900.0	*__72__	61.3	136.7	900.0	69	61.9	5.0	900.0	*__72__	59.8	172.7	900.0	71
20	15	600	61	66.8	5.8	900.0	__63__	65.9	3.8	900.0	61	65.5	44.3	900.1	__63__	64.2	536.0	900.0	__63__
20	20	600	52	68.9	6.4	900.0	*__55__	68.2	7.1	900.0	53	67.5	66.8	900.0	52	68.3	11.2	900.0	54
20	25	600	50	71.4	7.5	900.0	__52__	70.6	3.4	900.0	51	70.0	34.1	900.0	__52__	69.4	53.9	900.0	__52__
20	40	600	49	72.6	185.3	900.1	*__50__	72.1	138.6	900.0	49	71.7	11.8	900.0	47	72.5	685.8	900.0	49
20	60	600	46	73.7	138.6	900.0	__47__	73.0	11.5	900.0	46	72.8	15.6	900.0	46	72.3	690.3	900.2	__47__
20	80	600	__44__	71.6	367.0	900.1	__44__	70.5	132.5	900.0	__44__	70.1	145.4	900.3	42	71.4	638.4	900.0	__44__
20	100	600	39	75.8	280.6	900.2	__40__	75.3	6.5	900.0	39	74.5	254.7	900.2	38	74.8	141.9	905.8	__40__
20	150	600	37	76.0	30.3	900.3	*__38__	75.5	21.4	900.0	37	74.8	30.8	900.0	37	74.4	844.5	900.0	37
20	200	600	33	75.7	137.1	900.2	*__35__	74.6	144.7	900.0	34	73.0	499.2	900.2	33	73.6	104.0	900.0	34

Table 4
Virus benchmark. Results when aiming for solution quality.

			A* + BS				A* + ACS				APS				A* + ACS-dist				*lit. best*
$\|\Sigma\|$	m	n	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{best}}$[s]	$\bar{t}$[s]	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{best}}$[s]	$\bar{t}$[s]	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{best}}$[s]	$\bar{t}$[s]	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{best}}$[s]	$\bar{t}$[s]	$\overline{\|s\|}$
4	10	600	223	39.9	879.4	900.4	*__228__	38.2	80.8	900.0	222	39.3	19.2	826.6	221	39.6	394.2	900.0	227
4	15	600	200	45.2	3.7	900.0	*__206__	43.7	92.5	900.0	200	44.6	527.2	900.0	201	44.5	578.8	629.3	205
4	20	600	185	45.4	276.2	900.0	*__194__	42.9	327.6	900.0	185	45.1	61.0	900.0	183	45.9	190.8	609.0	192
4	25	600	190	46.8	185.8	900.0	*__196__	45.3	128.2	900.0	190	46.3	13.3	856.9	190	46.3	341.5	697.2	194
4	40	600	167	51.3	265.6	900.2	*__174__	49.6	264.0	900.0	167	50.7	191.4	900.0	152	55.2	246.4	678.0	170
4	60	600	162	52.9	185.0	900.0	*__168__	51.3	49.8	900.0	162	52.4	74.5	900.0	152	55.3	342.0	729.2	166
4	80	600	156	54.1	9.9	900.1	__163__	52.3	61.2	900.0	157	53.4	39.6	900.0	137	59.2	407.3	793.4	__163__
4	100	600	153	55.0	74.7	900.0	*__160__	53.1	71.5	900.0	153	54.5	79.7	900.0	136	59.5	636.2	872.6	158
4	150	600	152	54.9	19.7	900.0	*__157__	53.7	40.3	900.0	152	54.6	20.9	900.0	137	59.0	238.9	790.7	156
4	200	600	149	55.5	26.4	900.1	*__156__	53.6	582.5	900.0	150	54.8	602.6	900.0	133	59.9	310.3	897.3	154
20	10	600	74	60.8	132.2	900.0	__77__	59.3	14.6	900.0	75	59.0	189.1	900.0	76	58.2	26.7	900.0	77
20	15	600	62	66.7	7.4	900.0	__64__	65.8	4.0	900.0	63	65.0	32.4	900.0	__64__	64.2	127.7	900.0	64
20	20	600	58	69.1	7.7	900.1	*__61__	67.6	28.9	900.0	59	67.8	258.7	900.0	*__61__	66.5	852.6	900.0	60
20	25	600	53	70.4	7.4	900.1	*__56__	68.9	82.8	900.0	54	68.8	119.9	900.0	55	68.0	37.0	900.0	55
20	40	600	49	72.9	40.0	900.0	*__51__	71.8	110.4	902.3	49	71.7	5.1	900.0	49	71.8	118.1	900.0	50
20	60	600	47	73.4	312.9	900.0	__48__	73.0	6.1	900.0	47	72.2	7.0	900.0	47	72.4	837.4	900.0	__48__
20	80	600	45	74.6	744.7	900.2	__46__	74.0	7.1	900.0	45	73.4	8.8	900.1	45	73.4	683.6	900.0	__46__
20	100	600	44	75.0	97.2	900.1	__45__	74.6	8.9	900.0	44	74.3	134.1	900.1	44	74.0	880.7	900.0	__45__
20	150	600	45	75.1	42.6	900.6	*__46__	74.6	27.7	900.0	45	74.4	48.8	900.3	44	74.7	257.4	900.1	45
20	200	600	43	76.0	60.3	900.2	__44__	75.1	44.8	900.0	43	75.1	65.0	900.5	43	74.7	110.7	900.1	__44__

Table 5
ES benchmark. Results when aiming for solution quality (averaged over 50 instances per row).

			A* + BS				A* + ACS				APS				A* + ACS-dist				*lit. best*
m	n	$\|\Sigma\|$	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{best}}$[s]	$\bar{t}$[s]	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{best}}$[s]	$\bar{t}$[s]	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{best}}$[s]	$\bar{t}$[s]	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{best}}$[s]	$\bar{t}$[s]	$\overline{\|s\|}$
10	1000	2	604.7	23.5	350.8	866.7	*__618.9__	21.7	323.2	900.4	603.6	23.3	254.7	873.3	615.4	21.8	234.7	760.8	615.1
10	1000	10	195.7	57.9	285.9	891.0	*__205.0__	55.9	251.3	900.7	195.5	57.5	294.8	888.8	204.1	55.6	230.7	771.9	203.1
50	1000	2	526.6	33.0	287.3	897.5	*__540.9__	31.2	302.2	900.0	526.9	32.6	264.1	887.4	532.8	31.9	301.2	696.2	538.2
50	1000	10	131.0	71.3	219.88	900.2	*__137.5__	69.9	158.1	900.0	131.2	71.0	137.5	900.2	134.5	70.2	321.1	867.3	136.3
100	1000	2	509.1	35.1	250.8	900.1	*__522.1__	33.4	324.6	900.0	509.4	34.8	283.8	900.0	512.5	34.4	274.5	781.6	519.8
100	1000	10	118.8	73.9	217.7	900.2	*__124.1__	72.7	121.0	900.0	118.9	73.6	175.9	900.1	120.5	73.1	356.0	900.0	123.3
10	2500	25	224.5	72.1	276.4	900.0	235.0	70.7	419.5	900.4	223.9	72.0	263.5	900.0	*__236.6__	70.4	374.8	897.3	235.2
50	2500	25	132.1	83.3	217.1	900.1	*__140.4__	82.3	239.8	900.0	132.6	83.1	212.8	900.3	136.5	82.6	368.1	900.0	139.5
100	2500	25	116.8	85.2	268.2	900.6	*__123.4__	88.1	223.6	900.0	117.0	85.1	350.8	900.7	118.6	84.8	352.7	900.1	122.9
10	5000	100	137.5	84.0	338.2s	900.4	*__145.7__	84.7	434.3	900.2	136.8	84.1	392.8	901.1	144.6	83.1	340.6	900.1	144.9
50	5000	100	67.4	92.0	355.8	902.8	*__72.0__	97.6	286.1	900.1	67.6	91.9	432.1	902.7	69.6	91.9	330.6	900.7	71.9
100	5000	100	57.1	93.1	584.4	906.6	*__60.8__	97.4	515.7	900.1	57.1	93.1	601.9	905.8	57.9	93.6	382.3	901.5	60.7

- In order to check the statistical significance of differences, Friedman's test was performed simultaneously for all four anytime approaches. 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 [54]. The outcome is shown in Fig. 1 by means of so-called critical difference plots, one for each benchmark

Table 6
BB benchmark. Results when aiming for solution quality (averages over ten instances per row).

			A* + BS				A* + ACS				APS				A* + ACS-dist				*lit. best*
$\|\Sigma\|$	m	n	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\bar{t}[s]$	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\bar{t}[s]$	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\bar{t}[s]$	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\bar{t}[s]$	$\overline{\|s\|}$
2	10	1000	675.1	16.4	64.9	900.0	676.6	16.5	347.1	900.0	675.7	15.6	57.1	900.0	*__676.7__	16.4	152.0	885.4	676.5
2	100	1000	561.3	31.2	260.3	900.0	547.1	32.6	497.6	900.0	*__563.6__	30.6	264.3	900.0	486.7	40.0	464.9	870.0	560.7
4	10	1000	545.2	30.3	13.0	900.0	*__545.5__	30.7	204.9	900.0	545.2	29.4	13.4	900.0	*__545.5__	30.5	85.6	798.4	545.4
4	100	1000	389.4	51.1	209.8	900.4	344.3	56.4	503.4	900.0	*__390.2__	50.9	362.7	900.0	273.6	65.4	291.2	881.0	388.8
8	10	1000	__462.7__	39.0	17.0	900.0	__462.7__	39.9	68.4	900.0	__462.7__	38.0	19.2	900.0	__462.7__	39.7	16.9	827.2	__462.7__
8	100	1000	273.1	65.1	143.7	900.1	223.7	71.1	631.7	900.1	*__273.4__	65.0	179.8	900.1	164.7	78.7	408.7	900.0	272.1
24	10	1000	__385.6__	42.0	43.8	900.0	__385.6__	47.0	33.8	900.0	__385.6__	40.5	35.5	900.0	__385.6__	46.8	8.5	900.0	__385.6__
24	100	1000	149.4	79.5	138.0	900.4	117.0	83.5	636.9	900.3	149.4	79.5	145.2	900.4	83.8	88.2	550.2	900.0	__149.5__

Table 7
Rat benchmark. Results when aiming for small gaps.

			A* + BS				A* + ACS				APS				A* + ACS-dist			
$\|\Sigma\|$	m	n	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\bar{t}[s]$	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\bar{t}[s]$	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\bar{t}[s]$	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\bar{t}[s]$
4	10	600	198	40.5	354.2	741.0	206	__38.0__	468.2	900.0	198	40.7	454.1	900.0	204	38.6	315.1	683.5
4	15	600	180	46.9	3.3	900.0	187	__44.5__	211.8	900.0	181	46.9	7.3	908.2	186	44.6	419.7	713.7
4	20	600	166	42.2	13.9	900.0	173	__39.5__	384.1	900.0	167	42.2	123.9	902.4	170	40.6	176.4	716.3
4	25	600	165	50.5	116.8	900.0	173	__47.4__	430.3	900.0	166	50.4	336.5	900.0	170	48.3	393.9	769.8
4	40	600	150	50.0	121.3	900.0	154	__48.1__	258.3	900.0	151	49.8	18.5	900.0	151	49.2	11.8	771.6
4	60	600	149	54.6	8.1	900.0	153	__53.1__	215.1	900.0	149	55.0	7.5	900.1	149	54.3	217.8	748.4
4	80	600	136	50.7	205.8	900.0	143	__47.6__	33.8	900.0	137	50.9	359.9	900.0	127	53.5	11.9	755.2
4	100	600	133	51.6	144.6	900.0	138	__49.6__	11.8	900.0	134	51.6	37.5	900.0	127	53.6	332.5	900.0
4	150	600	123	44.1	250.9	900.1	131	__40.2__	519.7	900.0	124	43.6	104.4	900.0	105	52.1	826.2	900.0
4	200	600	121	46.5	22.8	900.0	124	__44.9__	17.2	900.1	121	46.7	24.4	900.2	102	54.7	124.8	900.0
20	10	600	70	59.8	7.0	900.0	71	59.0	20.9	900.1	70	60.5	7.2	900.0	71	__58.7__	234.4	900.0
20	15	600	60	65.1	7.8	900.0	63	__62.9__	5.9	900.1	61	65.3	23.1	900.1	62	63.3	84.6	884.8
20	20	600	53	66.9	358.5	900.1	55	__65.2__	196.4	900.1	54	67.1	48.4	900.1	52	66.9	114.8	900.0
20	25	600	50	69.7	8.2	900.0	52	68.3	15.0	911.1	51	70.0	42.9	900.2	52	__68.1__	583.1	900.1
20	40	600	48	71.6	12.6	900.3	49	__70.3__	137.1	900.0	49	71.8	567.4	900.2	45	72.6	75.4	900.0
20	60	600	45	72.7	18.9	900.4	47	__70.3__	346.6	900.4	47	72.2	652.4	900.2	45	71.5	509.9	900.1
20	80	600	44	69.4	642.3	900.4	43	__69.1__	63.4	900.3	44	70.5	65.4	900.2	40	71.2	243.8	900.1
20	100	600	38	74.3	24.1	900.0	40	__71.8__	175.4	900.3	39	74.5	216.8	900.6	37	73.8	431.1	900.0
20	150	600	37	73.2	31.1	900.7	37	__71.5__	89.5	900.3	37	74.8	31.3	900.5	34	74.0	612.1	900.2
20	200	600	32	77.1	37.6	900.0	34	__70.2__	28.1	900.6	35	72.0	433.3	900.1	31	72.8	152.6	900.2

Table 8
Virus benchmark. Results when aiming for small gaps.

			A* + BS				A* + ACS				APS				A* + ACS-dist			
$\|\Sigma\|$	m	n	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\bar{t}[s]$	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\bar{t}[s]$	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\bar{t}[s]$	$\overline{\|s\|}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\bar{t}[s]$
4	10	600	222	38.7	3.0	910.7	228	__36.8__	637.5	900.0	224	38.1	20.7	826.2	221	38.4	550.5	649.1
4	15	600	200	44.4	48.3	900.0	206	__42.5__	21.9	900.0	201	44.2	702.0	746.8	200	44.1	333.0	579.2
4	20	600	185	44.9	406.2	900.0	192	__42.7__	246.2	900.0	186	44.8	139.6	900.0	183	45.4	583.8	592.5
4	25	600	189	46.3	4.8	903.8	196	__44.2__	850.5	900.0	190	46.3	21.5	900.0	188	46.3	263.9	575.7
4	40	600	166	50.7	13.5	900.0	173	__48.4__	866.7	900.0	167	50.9	846.3	900.0	152	54.6	338.5	609.3
4	60	600	162	51.9	42.7	900.0	168	__49.9__	645.2	900.0	162	52.4	85.5	900.0	150	55.2	472.9	757.9
4	80	600	157	53.1	12.3	900.0	163	__50.8__	114.6	900.0	157	53.4	11.1	900.0	134	59.5	327.0	861.3
4	100	600	153	54.2	14.0	900.0	160	__51.5__	806.0	900.0	153	54.5	14.5	900.1	133	59.7	715.9	900.0
4	150	600	152	54.4	143.6	900.0	157	__52.6__	415.1	900.1	152	54.6	802.9	900.1	136	58.9	725.0	900.1
4	200	600	150	54.7	645.8	900.0	155	__52.7__	415.1	900.2	150	54.8	729.9	900.0	132	59.8	151.7	900.0
20	10	600	74	58.4	27.7	900.0	77	__56.5__	320.2	900.0	75	58.8	111.8	900.1	76	56.8	255.7	900.0
20	15	600	62	64.8	8.2	900.1	64	__62.6__	3.1	900.0	63	65.0	30.6	900.1	64	__62.6__	851.2	900.0
20	20	600	58	67.6	9.0	900.0	60	65.9	33.8	900.0	59	67.8	19.0	900.1	60	__65.7__	57.5	900.0
20	25	600	53	68.5	9.7	900.0	55	66.7	26.1	901.4	53	69.4	9.6	900.0	55	__66.5__	690.9	900.1
20	40	600	49	71.3	579.5	900.0	50	__70.1__	52.2	900.3	49	72.0	25.7	900.2	48	71.1	162.7	900.1
20	60	600	47	72.0	18.5	900.0	48	__70.4__	107.1	900.2	47	72.5	18.1	900.4	45	72.2	316.6	900.1
20	80	600	45	73.1	283.9	900.0	46	__71.4__	22.2	900.3	45	73.7	67.1	900.2	44	72.7	736.1	900.0
20	100	600	43	74.1	27.3	900.0	45	__72.0__	190.9	900.4	44	74.3	113.1	900.2	43	73.3	653.6	900.2
20	150	600	44	76.0	49.4	900.0	45	__72.7__	48.8	900.0	45	74.6	185.4	900.4	43	73.9	229.3	900.3
20	200	600	43	75.8	65.0	900.0	43	__73.3__	105.9	900.4	43	75.0	64.9	900.3	42	73.8	236.9	900.1

set. In short, each algorithm is positioned in the horizontal segment according to its average ranking concerning the considered set 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 considered as equal—that is, no difference of statistical significance can be detected. This is indicated in the graphics by horizontal bars joining the respective algorithms. Fig. 1 shows that A* + ACS produces significantly better results concerning solution quality for benchmark sets Rat, Virus, and BL. The differences observed for benchmark sets Random and ES are statistically not significant (despite the fact that A* + ACS produces new

Table 9
ES benchmark. Results when aiming for small gaps (averages over 50 instances per row).

			A* + BS				A* + ACS				APS				A* + ACS-dist			
n	m	$\lvert\Sigma\rvert$	$\overline{\lvert s\rvert}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\lvert s\rvert}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\lvert s\rvert}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\lvert s\rvert}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$
1000	10	2	606.4	22.8	177.3	881.0	618.1	**21.2**	427.4	900.0	607.5	22.7	165.9	881.7	614.9	21.6	269.3	818.2
1000	10	10	196.9	56.8	205.9	900.0	204.2	**54.9**	283.2	900.1	198.0	56.9	205.9	900.0	203.5	55.0	284.6	707.5
1000	50	2	526.4	32.6	300.6	892.9	540.3	**30.6**	377.0	900.0	526.6	32.7	267.5	900.0	532.6	31.6	349.2	799.5
1000	50	10	130.6	70.9	160.1	900.0	137.1	**69.1**	294.6	900.3	131.3	71.0	206.1	900.1	133.7	69.8	293.7	836.6
1000	100	2	508.9	34.8	265.1	900.0	521.6	**32.9**	336.9	900.1	509.4	34.8	297.5	900.0	512.0	34.1	440.7	875.2
1000	100	10	118.6	73.4	112.4	900.1	123.7	**71.9**	287.4	900.2	119.0	73.6	191.8	900.1	119.6	72.8	378.4	900.0
2500	10	25	226.6	71.5	179.5	900.6	231.5	70.6	576.4	900.1	227.5	71.5	244.3	900.1	235.2	**70.1**	443.9	900.0
2500	50	25	131.9	83.3	181.1	900.4	139.5	**81.9**	388.9	900.3	132.4	83.2	261.9	900.4	135.3	82.5	424.6	900.3
2500	100	25	116.5	85.2	221.0	900.6	122.7	**84.0**	360.3	900.5	117.0	85.1	362.6	900.7	117.6	84.7	405.0	900.2
5000	10	100	138.9	83.8	327.7	901.0	143.4	**82.9**	643.8	900.8	139.3	83.8	414.2	900.9	143.0	83.0	466.8	900.3
5000	50	100	67.3	92.0	337.6	902.4	71.0	**91.3**	470.8	903.5	67.5	92.0	411.2	902.6	68.3	91.6	536.0	902.0
5000	100	100	57.0	93.1	575.9	900.0	59.6	**92.6**	488.6	907.3	57.0	93.1	626.2	905.7	56.2	93.0	518.6	903.4

Table 10
BB benchmark. Results when aiming for small gaps (averaged over ten instances per row).

			A* + BS				A* + ACS				APS				A* + ACS-dist			
$\lvert\Sigma\rvert$	m	n	$\overline{\lvert s\rvert}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\lvert s\rvert}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\lvert s\rvert}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$	$\overline{\lvert s\rvert}$	$\overline{gap}[\%]$	$\overline{t_{\text{best}}}[s]$	$\overline{t}[s]$
2	10	1000	676.7	16.6	155.8	900.3	675.4	**16.2**	18.1	900.0	674.6	**16.2**	115.0	900.0	676.7	16.5	63.9	884.6
2	100	1000	547.4	32.6	428.3	900.5	561.8	31.0	357.3	900.0	563.2	**30.9**	557.2	900.0	486.5	40.1	398.6	864.0
4	10	1000	545.5	30.7	98.4	901.0	545.2	**30.0**	14.4	900.0	545.2	**30.0**	60.4	900.0	545.5	30.6	42.1	853.6
4	100	1000	346.5	56.1	507.0	901.0	389.2	**50.9**	186.7	900.0	389.4	51.1	400.2	900.0	270.2	65.8	487.3	859.6
8	10	1000	462.7	39.8	15.6	900.5	462.7	38.7	18.8	900.0	462.7	**38.6**	89.6	900.0	462.7	39.6	7.5	900.2
8	100	1000	224.1	71.0	524.3	901.9	273.0	**65.1**	106.8	900.1	272.9	**65.1**	388.1	900.1	160.9	79.1	575.0	901.4
24	10	1000	385.6	46.3	1.5	901.1	385.6	41.6	34.1	900.0	385.6	**41.2**	122.8	900.0	385.6	46.0	1.7	900.8
24	100	1000	120.9	82.9	657.5	908.4	149.4	**79.5**	138.6	900.4	149.3	**79.5**	580.9	900.6	80.6	88.6	606.8	910.1

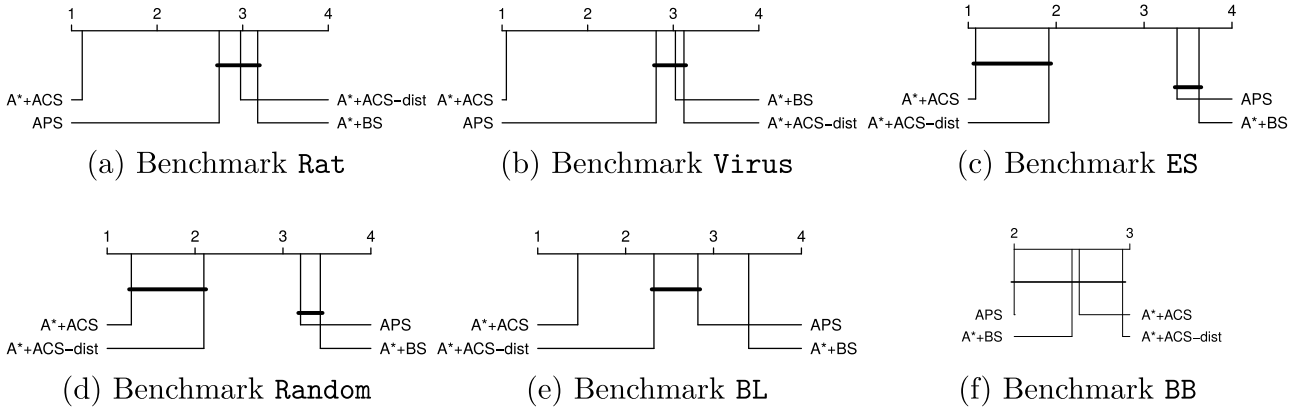


Fig. 1. Critical difference plots concerning solution quality.

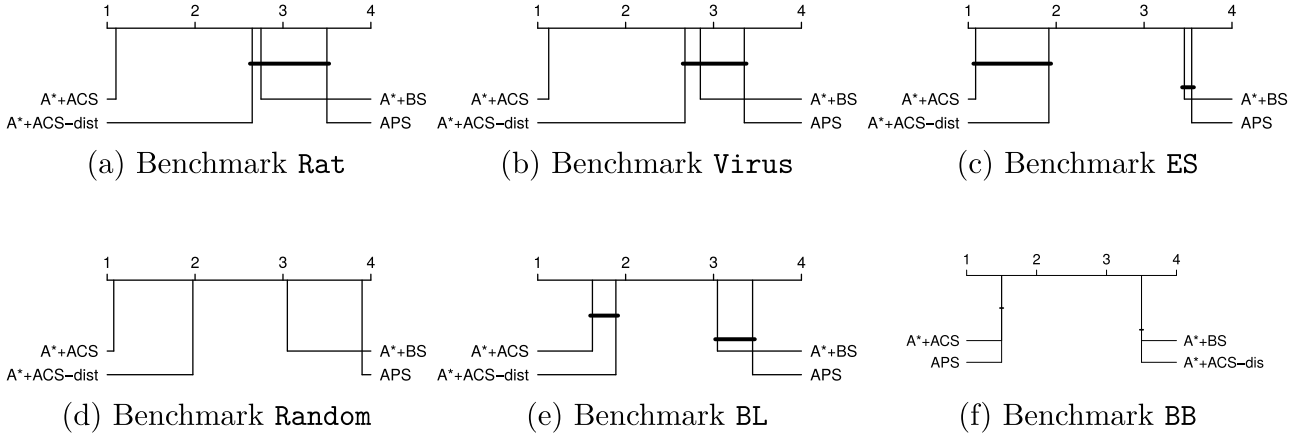


Fig. 2. Critical difference plots concerning the obtained gaps.

state-of-the-art results in 13 out of 20, and in 10 out of 12 cases, respectively).

- Just like classical A*, both A* + ACS-dist and A* + ACS are able to prove optimality for the instances of benchmark set BL with $n = 100$ and $|\Sigma| \geq 12$. This is indicated by entries with value 0.0 in columns with heading $\overline{gap}$ [%] (see Table 12). However, as expected, more computation time is needed than by A*.
- A* + ACS does not only beat the competitors we considered here. It performs also very favorably in comparison to purely heuristic state-of-the-art approaches from the literature. This can be observed by comparing the performance of A* + ACS with the last columns in Tables 3–6 and Tables 11–12 which contain the so far best known results from the literature.[4] Overall A* + ACS was able to obtain new best-known results in 82 out of 117 cases (table rows).
- For what concerns the performance of the four algorithms with respect to the produced gaps – see Tables 7–10 and Tables 13–14 (from Appendix A) – it can be observed that – also in this case – A* + ACS generally outperforms the competing algorithms. This is with the exception of benchmark set BB, where no clear tendency can be identified. The statistical significance of this conclusion is tested in the same way as done for the case of aiming for solution quality. The corresponding critical difference plots are shown in Fig. 2. Moreover, the improvements of A* + ACS over the competitors are graphically shown in Figs. 9–12 (Appendix B).
- Nevertheless, observe that A* + ACS-dist often produces better final gaps than A* + ACS for instances with a low number of input string. This is the case, for example, for instances with $n = 10$ from benchmark set BL; see Table 14 and Fig. 9. A possible explanation for this behavior is as follows. On the one hand, the heuristic guidance function dist(·) performs rather well for instances with a low number of input strings (that is, a low m-value), which means that A* + ACS-dist will not have a major disadvantage with respect to A* + ACS in those cases. On the other hand, dist(·) requires less computation time than the EX function used by A* + ACS. This implies that A* + ACS-dist is able to perform more node expansions than A* + ACS within the allowed computation time, which leads to better upper bounds.

5.5. Comparison of the algorithms' anytime behavior

So far we have only studied the final results of the algorithms for what concerns solution quality and gaps. However, in the context of anytime algorithms, another important aspect to take into account is their anytime behavior. In order to visualize the anytime behavior of the algorithms, we plot the evolution of the solution quality, respectively the gaps, over time (either averaged over all problem instances of the same specifications, or averaged over multiple runs for single problem instances, depending on the benchmark set). The plots concerning solution quality are shown, for seven representative cases, in Fig. 3, while the ones concerning the gaps are shown, for the same seven cases, in Fig. 4. In addition to the curves showing the average behavior, these graphics also contain boxplots – shown every 200 seconds – indicating the variability of the algorithm performance.

The following observations can be made concerning the anytime plots on solution quality:

- A* + ACS generally finds solutions of higher quality than the other algorithms in early stages of the search process. The main reason for this is clearly the heuristic guidance function EX() which is utilized in A* + ACS.
- Notice also that A* + ACS is able to find improving solutions more frequently than A* + BS or APS. For these latter algorithms it seems much harder to find improving solutions at later stages of the search process. Even though A* + ACS-dist can be said to generally outperform APS and A* + BS, it cannot match the performance of A* + ACS. It can also be observed that the compared algorithms find improving solutions in general more frequently when the alphabet size is rather small.
- APS and A* + BS, which make both use of an embedded BS to find heuristic solutions, show a similar evolution of solution quality over time. It is noticeable that a rather large beam size β is required to achieve the best possible anytime performance within the given computation time.

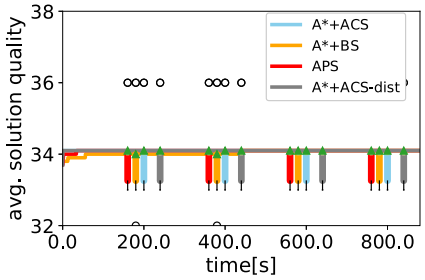
Concerning the anytime performance of the algorithms with respect to the gaps we can make the following observations:

- For the smallest ones of the considered instances—that is, the instances from benchmark set BL with $n = 100$—A* + BS shows the best evolution of the obtained gaps (see Fig. 4(a)). This is for the following two reasons: (1) the parameter values identified by our tuning process allow a significant amount of A* iterations, which is crucial for obtaining a favorable evolution of the gaps, and (2) near-optimal solutions are easily obtained for these instances by any of the algorithms.
- Concerning the remaining medium-size and large-size instances, A* + ACS shows a better anytime performance concerning the gaps for the Virus, Rat, Random, and ES benchmarks; see Figs. 4(b)–4(f). This is because the ACS-iterations, even with a rather low value of β, are still able to find rather high-quality primal solutions, while a significantly increased number of A*-iterations (in comparison to the parameter setting used when aiming for solution quality) provides improved upper bounds. In this sense, A* + ACS is an algorithm that is much better balanced than the competitors.
- In the case of small alphabet sizes, A* + ACS-dist is not able to keep up with the performances of the other algorithms (see Figs. 4(c) and 4(d)). Mainly responsible for this is the heuristic function dist(·), which provides a weaker guidance than in particular EX for finding good primal bounds, especially in the case of small alphabet sizes.
- APS and A* + BS show a similar behavior concerning the evolution of the average gaps over time. The necessity of working with a large beam width (β) hinders the evolution of the gap since the search is mainly focused on improving solution quality and less on improving the upper bound.

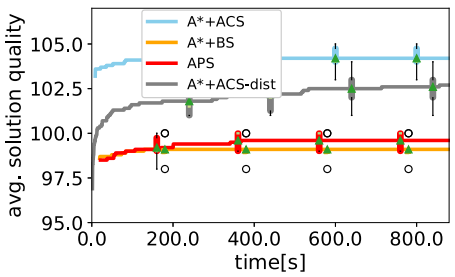
6. Conclusions and future work

We presented an exact A* algorithm for the LCS problem based on the general search framework for the problem proposed in our earlier study, which combines features of various other heuristic techniques. This A* search makes use of the combination of two previously known upper bound functions for the length of the LCS and is able to solve instances of up to $n = 100$ and $|\Sigma| \geq 12$ to proven optimality (106 instances from the literature are solved to optimality), most of them in a fraction of a second. For larger or more complex instances, however, the exact A* search soon either runs out of memory or requires substantially more time. Therefore, we combined A* search with either BS or ACS by interleaving

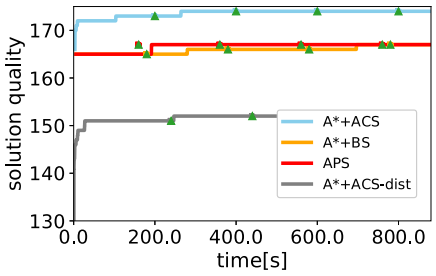
[4] As the best result for a specific group of instances from the literature we took the maximum average solution quality among the reported averages (if any) from [21–23,26,27]. Most of best results so far are from BS-EX [26].



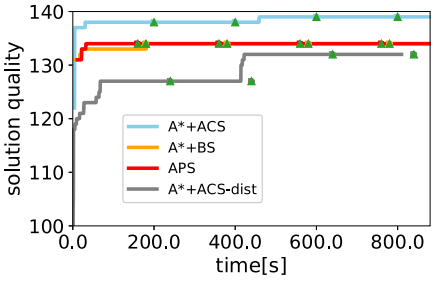
(a) Benchmark set BL: 10 instances with $|\Sigma| = 4, m = 10, n = 100$.



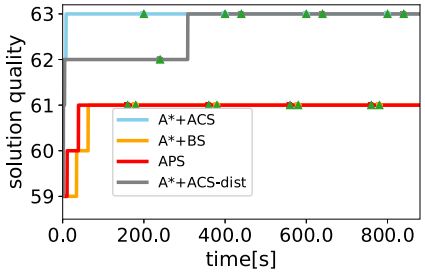
(b) Benchmark set BL: 10 instances with $|\Sigma| = 12, m = 50, n = 1000$.



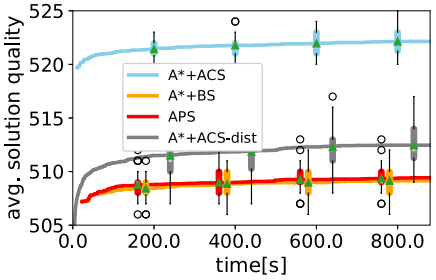
(c) Benchmark set Virus: instance $|\Sigma| = 4$, $m = 40, n = 600$.



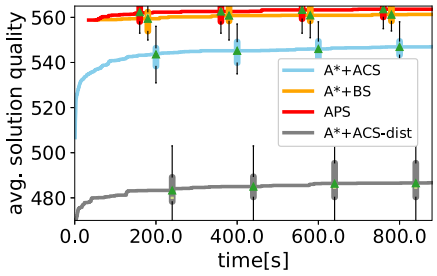
(d) Benchmark set Rat: instance $|\Sigma| = 4$, $m = 100, n = 600$.



(e) Benchmark set Random: instance $|\Sigma| = 20$, $m = 10, n = 600$.



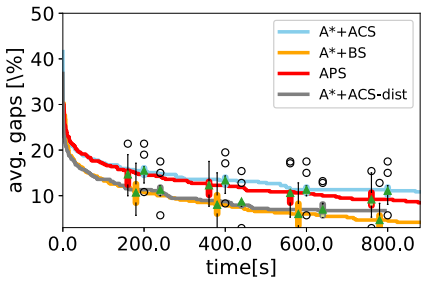
(f) Benchmark set ES: 50 instances with $|\Sigma| = 2, m = 100, n = 1000$.



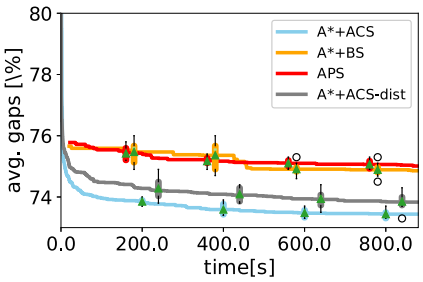
(g) Benchmark set BB: 10 instances with $|\Sigma| = 2, m = 100, n = 1000$.

Fig. 3. Comparison of the algorithms' anytime behavior concerning solution quality.

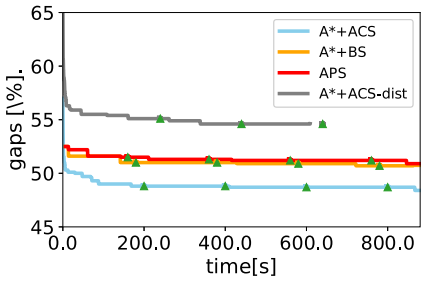
traditional A* iterations with BS runs of small width or single iterations of ACS, respectively. Note that we did this combination in a way that avoids redundant expansions of the same nodes, i.e., the methods act on a shared list of open nodes. These anytime algorithms, denoted by A* + ACS and A* + BS either run until optimality is proven or they are terminated prematurely, in which case a solution of promising quality is returned in combination with an upper bound. To the best of our knowledge, we report proven optimality gaps for larger LCS instances for the first time ever in the literature. Our two anytime algorithms were compared to the well known *Anytime Pack Search* (APS) and a variant of A* + ACS employing the `dist` heuristic as guidance. All the parameters of the algorithms were tuned w.r.t. both, solution quality and small gaps by using `irace`. Our computational study showed that A* + ACS performs in most cases significantly better than the other algorithms concerning solution quality. New best



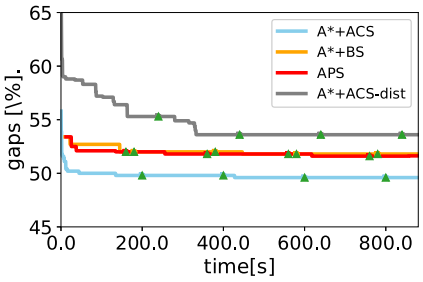
(a) Benchmark set BL: 10 instances with $|\Sigma| = 4, m = 10, n = 100$.



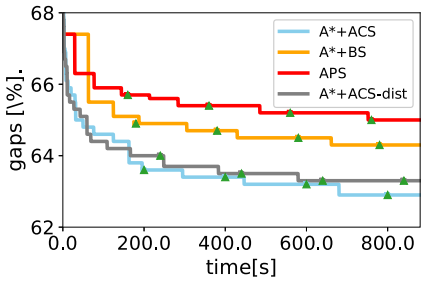
(b) Benchmark set BL: 10 instances with $|\Sigma| = 12, m = 50, n = 1000$.



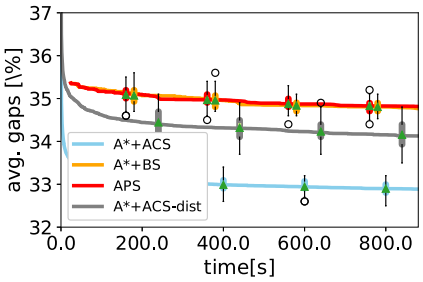
(c) Benchmark set Virus: instance $|\Sigma| = 4$, $m = 40, n = 600$.



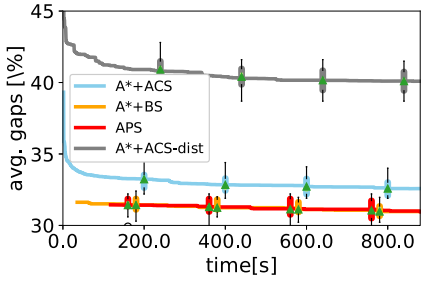
(d) Benchmark set Rat: instance $|\Sigma| = 4$, $m = 100, n = 600$.



(e) Benchmark set Random: instance $|\Sigma| = 20$, $m = 10, n = 600$.



(f) Benchmark set ES: 50 instances with $|\Sigma| = 2, m = 100, n = 1000$.



(g) Benchmark set BB: 10 instances with $|\Sigma| = 2, m = 100, n = 1000$.

Fig. 4. Comparison of the algorithms' anytime behavior concerning gaps.

solutions where found by A* + ACS for 82 different LCS instance groups from the literature (≈ 70% of all instance groups from the literature), and for the remaining groups, the so far best known results were matched by A* + ACS in most cases. Also concerning optimality gaps, A* + ACS outperforms the other approaches in most cases or is on par with them. Last but not least, A* + ACS usually provides a better anytime behavior in the sense that it earlier produces better results, and more frequently improves on them over time. Responsible for the success of A* + ACS is the careful selection and combination of strategies and components that proved already successful or promising in earlier works such as pure heuristic beam searches. The most important aspect is that we use, on the one hand, the upper bound UB for steering the classical A* search iterations and, on the other hand, the separate heuristic function EX for guiding the ACS iterations. While UB is required to obtain upper bounds and finally prove optimality,

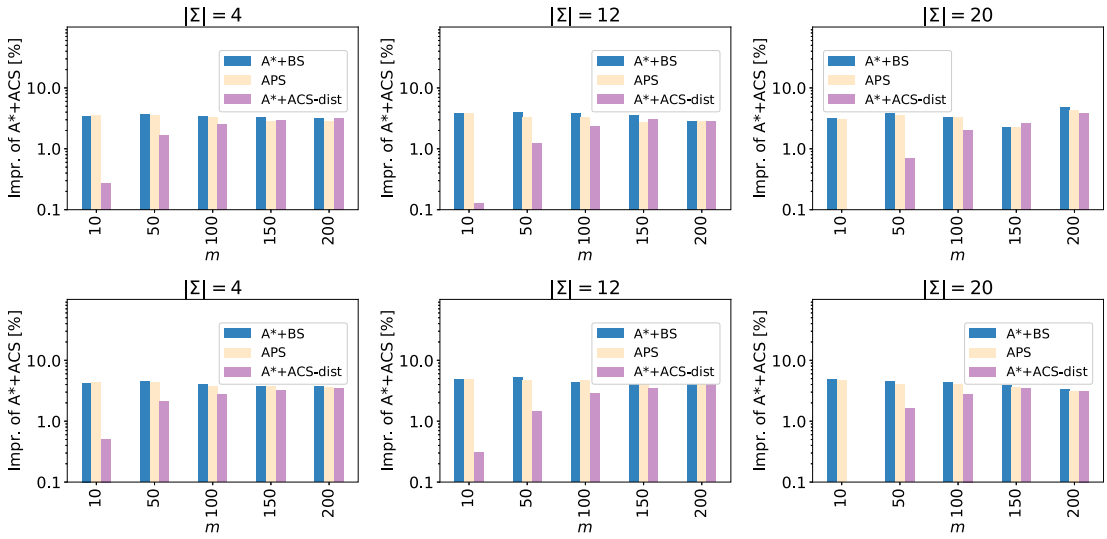


Fig. 5. Improvement of A* + ACS over the competitors in terms of solution quality (in %) for benchmark set BL. First row: instances with $n = 500$. Second row: instances with $n = 1000$.

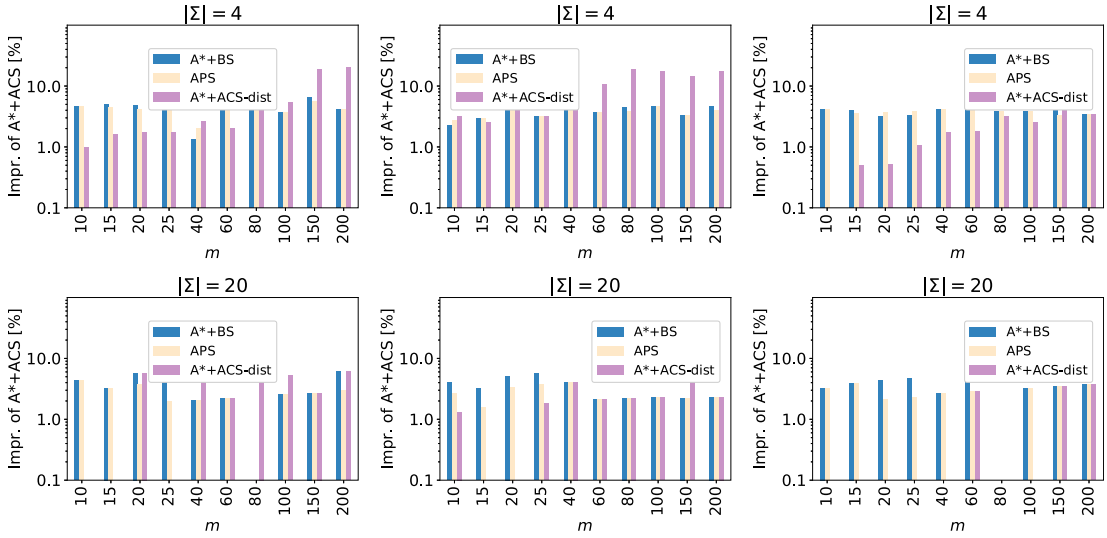


Fig. 6. Improvement of A* + ACS over the competitors in terms of solution quality (in %) for benchmark sets `Rat` (first column of graphs), `Virus` (second column of graphics) and `Random` (last column of graphics).

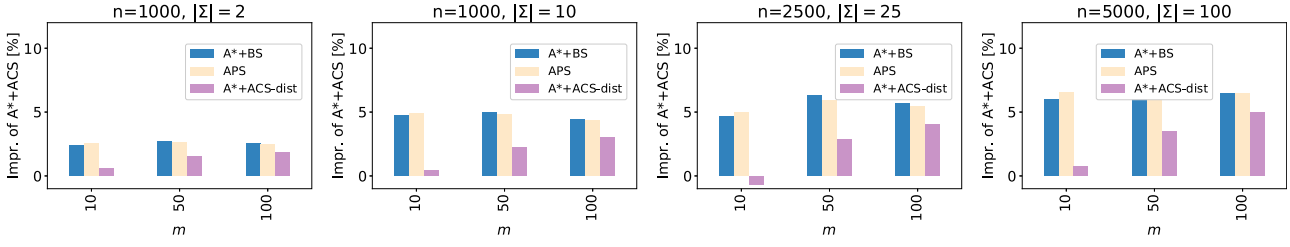


Fig. 7. Improvement of A* + ACS over the competitors in terms of solution quality (in %) for benchmark set ES.

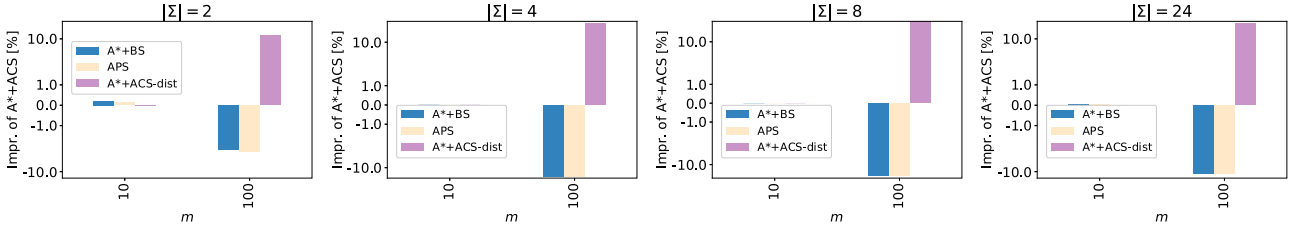


Fig. 8. Improvement of A* + ACS over the competitors in terms of solution quality (in %) for benchmark set BB.

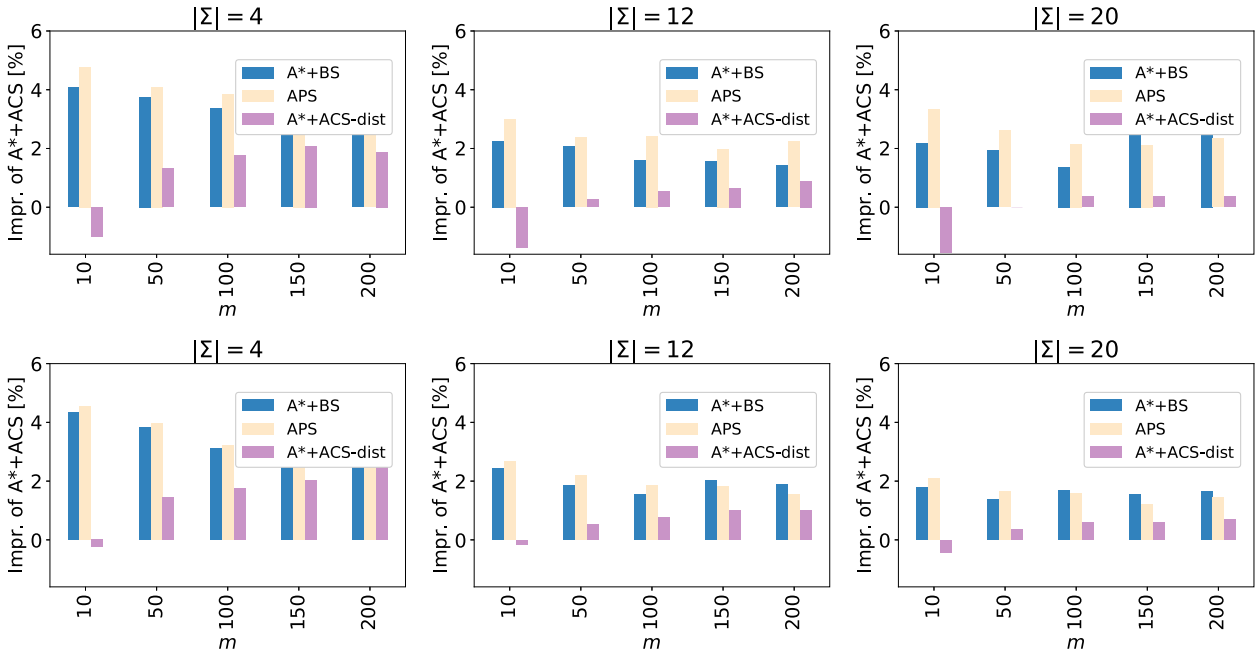


Fig. 9. Improvement of A* + ACS over the competitors in terms of gaps (in %) for benchmark set BL. First row: instances with $n = 500$. Second row: instances with $n = 1000$.

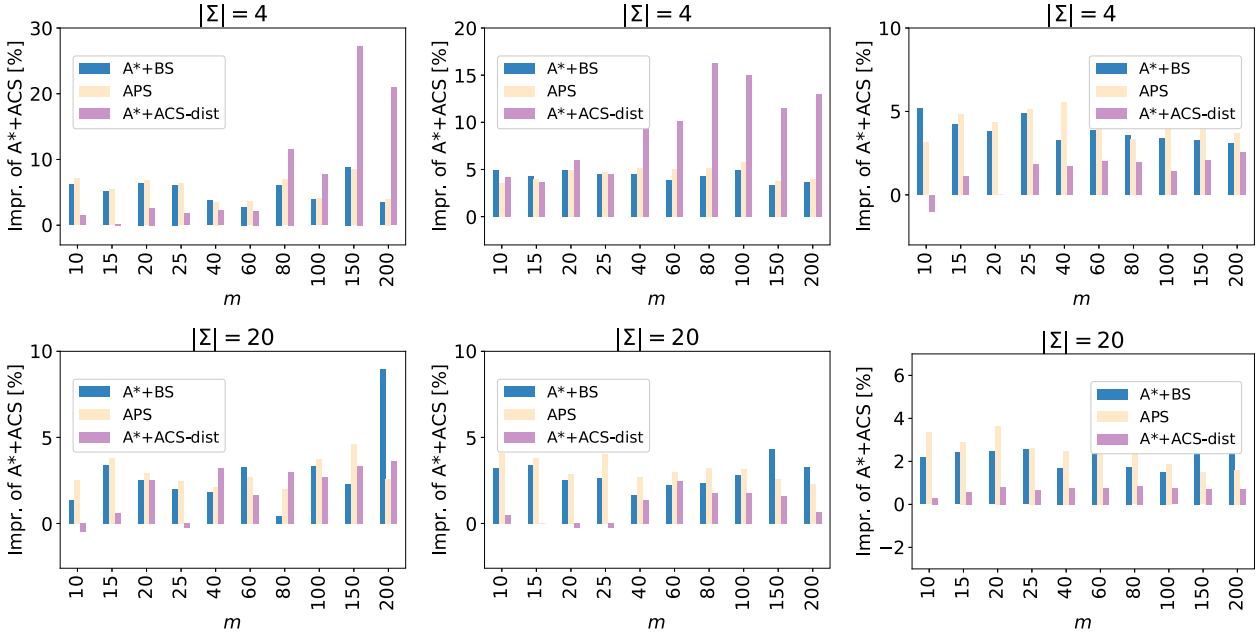


Fig. 10. Improvement of A* + ACS over the competitors in terms of gaps (in %) for benchmark sets Rat (first column of graphs), Virus (second column of graphics) and Random (last column of graphics).

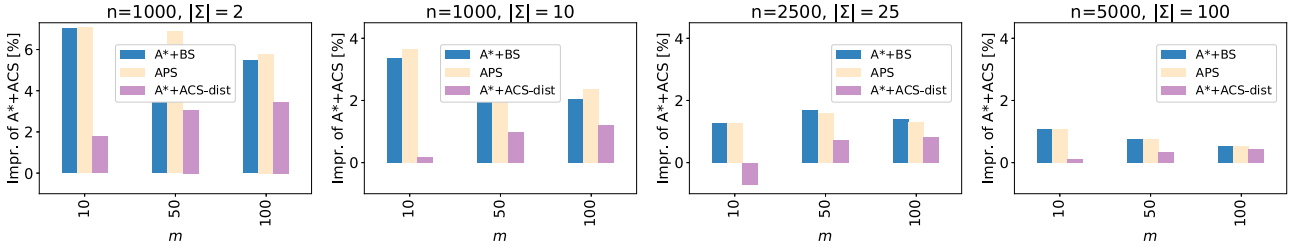


Fig. 11. Improvement of A* + ACS over the competitors in terms of gaps (in %) for benchmark set ES.

Table 11
Random benchmark. Results when aiming for solution quality.

			A* + BS				A* + ACS				APS				A* + ACS-dist				*lit. best*
$\|\Sigma\|$	m	n	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}$[$s$]	$\bar{t}$[s]	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}$[$s$]	$\bar{t}$[s]	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}$[$s$]	$\bar{t}$[s]	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}$[$s$]	$\bar{t}$[s]	$\overline{\|s\|}$
4	10	600	214	41.8	2.52	900.0	*223	39.2	548.2	900.0	214	41.0	2.2	900.0	*223	38.6	173.6	681.1	221
4	15	600	198	46.5	35.8	624.91	*206	44.5	16.0	900.0	199	45.6	892.9	900.0	205	44.1	140.4	596.2	204
4	20	600	189	48.4	687.7	900.0	*195	46.7	130.0	900.0	188	48.1	245.0	900.0	194	46.6	158.7	686.9	193
4	25	600	183	49.9	357.7	900.0	*189	48.2	26.7	900.0	182	49.4	7.7	900.0	187	48.2	110.4	627.1	187
4	40	600	170	53.4	234.6	796.0	*177	51.5	457.5	900.0	170	52.9	110.3	900.0	174	51.7	585.8	718.9	175
4	60	600	162	55.2	8.6	900.0	*169	53.3	275.9	900.0	162	54.7	7.9	900.0	166	53.6	448.3	705.0	168
4	80	600	158	56.5	112.4	900.0	*164	54.8	337.6	900.0	158	56.1	67.3	900.0	159	55.7	9.1	765.3	163
4	100	600	155	57.4	170.6	900.0	*161	55.6	735.4	900.0	155	56.8	70.9	900.0	157	56.3	24.1	832.1	159
4	150	600	149	58.9	19.7	900.0	*155	57.2	487.5	900.0	150	58.0	71.8	900.0	149	58.4	20.1	900.1	153
4	200	600	147	59.1	135.3	900.0	*152	57.8	130.5	900.0	147	58.8	54.0	900.0	147	58.7	430.0	900.0	151
20	10	600	61	66.9	62.6	900.0	**63**	65.4	8.1	900.0	61	65.7	38.7	900.0	**63**	64.2	315.5	900.0	**63**
20	15	600	51	72.3	347.2	900.0	**53**	71.0	4.4	900.0	51	71.3	5.8	900.0	**53**	70.1	201.5	900.0	**53**
20	20	600	46	74.6	57.0	900.0	**48**	73.5	3.6	900.0	47	73.1	804.5	900.0	**48**	72.4	160.4	900.0	**48**
20	25	600	43	76.0	7.8	900.1	*45	74.7	7.1	900.0	44	74.6	187.0	900.1	*45	73.7	342.2	900.0	44
20	40	600	38	78.5	9.8	900.1	**39**	77.8	4.3	900.0	38	77.8	11.4	900.1	**39**	76.9	778.8	900.6	**39**
20	60	600	34	80.6	23.0	900.1	*36	79.2	14.5	900.0	35	79.2	283.5	900.0	35	78.9	10.2	900.0	35
20	80	600	**33**	80.8	15.7	900.3	**33**	80.7	5.9	900.0	**33**	80.1	20.0	900.1	**33**	79.9	495.3	900.0	**33**
20	100	600	31	82.0	23.8	900.0	**32**	81.4	7.6	900.0	31	81.3	21.3	900.1	**32**	80.4	237.4	900.3	**32**
20	150	600	29	83.0	413.5	900.6	*30	82.4	733.9	900.0	29	82.4	817.7	900.2	29	82.1	817.7	900.1	29
20	200	600	27	84.0	37.7	901.2	**28**	83.3	14.1	900.0	27	83.1	19.7	900.1	27	83.1	19.7	900.1	**28**

Table 12
BL benchmark. Results when aiming for solution quality (averages over ten instances per row).

			A* + BS				A* + ACS				APS				A* + ACS-dist				*lit. best*
m	n	$\|\Sigma\|$	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}$[$s$]	$\bar{t}$[s]	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}$[$s$]	$\bar{t}$[s]	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}$[$s$]	$\bar{t}$[s]	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}$[$s$]	$\bar{t}$[s]	$\overline{\|s\|}$
10	100	4	**34.1**	51.4	20.1	882.1	**34.1**	15.1	0.3	900.2	**34.1**	10.9	4.0	900.0	**34.1**	9.9	0.1	800.7	**34.1**
10	100	12	**12.7**	0.0	0.2	5.2	**12.7**	0.0	0.2	5.2	**12.7**	0.0	0.7	7.1	**12.7**	0.0	0.0	4.2	**12.7**
10	100	20	**7.9**	0.0	0.1	0.1	**7.9**	0.0	0.1	0.1	**7.9**	0.0	0.1	0.1	**7.9**	0.0	0.0	0.1	**7.9**
10	500	4	179.9	40.9	312.0	900.0	*186.0	38.8	109.5	900.0	179.6	40.2	221.6	792.7	179.6	38.2	52.8	736.9	184.1
10	500	12	76.4	60.2	134.8	900.0	*79.3	58.5	19.4	900.0	76.4	59.1	123.5	900.0	76.4	57.3	57.2	795.6	78.7
10	500	20	49.7	66.7	87.4	900.0	*51.3	65.2	48.2	900.0	49.8	65.2	104.8	900.0	49.8	63.7	4.2	900.0	51.1
10	1000	4	362.6	42.4	209.7	900.0	*378.0	40.0	369.7	901.1	362.1	42.1	200.9	900.0	362.1	40.0	288.7	686.6	374.6
10	1000	12	156.2	62.2	214.4	900.4	*163.7	60.4	143.4	900.0	156.2	61.7	229.8	885.6	156.2	60.0	226.5	818.5	162.0
10	1000	20	102.4	68.9	120.8	900.0	*107.4	67.3	134.1	900.0	102.6	68.3	238.0	900.2	102.6	66.7	294.8	891.2	106.5
50	100	4	**24.2**	32.9	18.7	884.1	**24.2**	29.2	0.3	893.3	**24.2**	24.3	2.3	900.0	**24.2**	23.6	61.3	754.5	**24.2**
50	100	12	**6.9**	0.0	0.1	0.3	**6.9**	0.0	0.1	0.2	**6.9**	0.0	0.2	0.2	**6.9**	0.0	0.1	0.2	**6.9**
50	100	20	**3.0**	0.0	0.0	0.0	**3.0**	0.0	0.0	0.0	**3.0**	0.0	0.0	0.0	**3.0**	0.0	0.0	0.0	**3.0**
50	500	4	136.9	54.1	63.6	892.8	*142.0	52.4	238.3	897.3	137.2	53.3	137.0	877.7	137.2	52.5	301.7	703.0	141.0
50	500	12	47.8	74.0	190.0	900.0	*49.7	72.8	152.6	900.0	48.1	73.0	92.5	900.1	48.1	72.2	172.2	893.6	49.2
50	500	20	28.2	79.3	54.3	900.1	**29.3**	78.3	3.6	900.0	28.3	78.3	31.0	900.1	28.3	77.2	133.7	901.0	**29.3**
50	1000	4	278.6	55.3	162.5	900.0	*291.0	53.4	348.6	900.0	279.0	55.0	190.2	900.0	279.0	54.0	271.7	713.3	288.6
50	1000	12	99.1	75.4	79.0	900.3	*104.2	74.2	72.8	900.0	99.6	75.0	157.8	900.1	99.6	74.1	358.2	896.1	103.5
50	1000	20	60.5	81.0	114.5	900.1	*63.2	80.1	52.6	900.0	60.8	80.4	190.2	900.2	60.8	79.9	307.1	901.7	62.5
100	100	4	21.9	37.3	226.2	900.0	*22.1	32.5	1.1	895.5	*22.1	28.1	12.2	900.0	*22.1	24.9	63.1	815.0	22.0
100	100	12	**5.2**	0.0	0.1	0.1	**5.2**	0.0	0.1	0.1	**5.2**	0.0	0.1	0.1	**5.2**	0.0	0.0	0.1	**5.2**
100	100	20	**2.1**	0.0	0.0	0.0	**2.1**	0.0	0.0	0.0	**2.1**	0.0	0.0	0.0	**2.1**	0.0	0.0	0.0	**2.1**
100	500	4	127.6	57.9	135.2	900.0	*131.9	55.6	74.6	900.0	127.7	56.5	86.0	900.0	127.7	56.1	358.6	868.4	130.8
100	500	12	41.8	76.8	41.0	900.1	*43.4	75.9	52.0	900.2	42.0	75.8	17.8	900.2	42.0	75.4	136.2	900.1	43.1
100	500	20	24.2	81.6	16.1	900.4	*25.0	80.9	7.1	900.0	24.2	80.8	17.3	900.4	24.2	80.1	107.9	900.0	24.9
100	1000	4	261.8	57.9	135.3	900.0	*272.4	56.2	291.9	900.0	262.6	57.5	317.2	900.1	262.6	57.1	257.0	817.3	270.6
100	1000	12	89.2	77.8	106.3	900.2	*93.1	76.8	67.3	900.0	89.0	77.6	93.3	900.2	89.0	77.0	295.2	900.0	92.4
100	1000	20	52.8	83.2	50.4	900.2	*55.1	82.5	61.1	900.6	53.0	82.9	124.1	900.5	53.0	82.5	216.9	901.6	54.7
150	100	4	20.3	37.4	23.4	900.0	*20.8	30.6	2.9	899.1	20.7	26.3	117.9	900.0	20.7	22.4	38.5	826.7	20.5
150	100	12	**4.7**	0.0	0.0	0.1	**4.7**	0.0	0.0	0.0	**4.7**	0.0	0.1	0.1	**4.7**	0.0	0.0	0.0	**4.7**
150	100	20	**1.9**	0.0	0.0	0.0	**1.9**	0.0	0.0	0.0	**1.9**	0.0	0.0	0.0	**1.9**	0.0	0.0	0.0	**1.9**
150	500	4	123.5	58.4	180.3	900.0	*127.5	56.9	150.9	899.7	124.0	57.7	313.6	900.0	124.0	57.6	314.1	884.5	126.4
150	500	12	39.5	77.9	113.9	900.2	*40.9	77.1	24.1	900.0	39.8	77.1	184.7	900.3	39.8	76.9	141.0	900.0	40.4
150	500	20	22.5	82.8	31.4	900.5	**23.0**	82.3	7.4	900.0	22.5	81.7	27.9	900.8	22.5	81.5	73.3	900.1	**23.0**
150	1000	4	254.6	59.0	365.0	900.1	*264.0	57.5	245.6	900.0	254.5	58.8	311.1	900.1	254.5	58.6	295.5	864.8	262.8
150	1000	12	84.5	79.0	105.5	900.3	*88.1	78.0	44.4	900.0	84.6	78.6	101.7	900.3	84.6	78.3	296.4	900.0	87.7
150	1000	20	49.7	84.2	98.6	900.4	*51.6	83.5	86.8	900.0	49.8	83.8	114.8	900.9	49.8	83.6	280.8	900.1	51.2
200	100	4	19.9	37.9	5.7	900.0	*20.1	32.8	9.0	898.2	*20.1	28.2	58.8	900.1	*20.1	24.6	17.0	861.6	19.9
200	100	12	**4.1**	0.0	0.0	0.0	**4.1**	0.0	0.0	0.0	**4.1**	0.0	0.1	0.1	**4.1**	0.0	0.0	0.0	**4.1**
200	100	20	**1.1**	0.0	0.0	0.0	**1.1**	0.0	0.0	0.0	**1.1**	0.0	0.0	0.0	**1.1**	0.0	0.0	0.0	**1.1**
200	500	4	121.0	59.3	93.0	900.0	*124.8	57.9	188.1	900.0	121.3	58.6	126.0	900.1	121.3	58.6	258.6	880.1	123.7
200	500	12	38.0	78.7	41.0	900.4	*39.1	77.9	57.1	900.0	38.0	78.0	31.2	900.3	38.0	77.7	82.9	900.0	39.0
200	500	20	21.0	83.7	28.0	900.5	*22.0	82.9	41.4	900.2	21.1	82.9	107.5	901.0	21.1	82.3	116.4	900.1	21.8
200	1000	4	249.6	59.8	283.5	900.1	*258.8	58.3	170.0	900.0	249.8	59.6	235.8	900.1	249.8	59.4	448.5	893.6	257.6
200	1000	12	81.8	79.5	244.9	900.2	*85.2	78.6	59.5	900.0	81.9	79.2	205.1	900.4	81.9	79.1	205.4	900.0	84.8
200	1000	20	47.8	84.6	305.7	900.4	*49.4	84.1	93.4	900.0	47.9	84.5	190.7	900.6	47.9	84.2	309.5	900.1	49.1

Table 13
Random benchmark. Results when aiming for small gaps.

			A* + BS				A* + ACS				APS				A* + ACS-dist			
$\|\Sigma\|$	m	n	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}[s]$	$\bar{t}[s]$	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}[s]$	$\bar{t}[s]$	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}[s]$	$\bar{t}[s]$	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}[s]$	$\bar{t}[s]$
4	10	600	215	40.4	3.4	900.0	222	38.3	100.0	900.0	219	39.5	568.6	900.0	223	**37.9**	236.1	633.2
4	15	600	200	45.2	146.8	900.0	206	**43.3**	167.7	900.0	200	45.4	74.6	900.0	204	43.8	64.4	586.4
4	20	600	188	47.6	209.3	749.4	194	**45.8**	158.7	900.0	189	47.8	731.3	917.0	194	**45.8**	75.0	887.5
4	25	600	182	49.3	397.2	900.0	189	**46.9**	235.2	900.0	183	49.3	189.7	900.0	186	47.8	7.3	578.1
4	40	600	171	52.1	57.8	812.6	176	**50.4**	42.8	900.0	169	53.2	34.9	900.0	173	51.3	627.6	640.6
4	60	600	162	54.5	167.6	900.0	168	**52.4**	471.3	900.1	163	54.6	462.0	900.0	164	53.5	56.2	628.7
4	80	600	157	56.0	108.8	900.1	163	**54.0**	115.6	900.1	159	55.8	115.6	900.0	159	55.1	65.0	725.5
4	100	600	155	56.6	170.2	900.0	160	**54.7**	168.1	900.0	155	56.9	84.4	900.0	157	55.5	118.0	900.0
4	150	600	149	58.1	20.5	900.1	154	**56.2**	301.2	900.0	149	58.4	20.1	900.0	150	57.4	173.2	790.8
4	200	600	147	58.5	115.3	900.1	151	**56.7**	37.8	900.0	147	58.8	54.8	900.0	146	58.2	221.0	900.0
20	10	600	61	64.3	2.8	900.0	63	**62.9**	162.8	900.1	62	65.0	29.2	900.1	62	63.1	59.3	890.1
20	15	600	51	70.5	7.7	900.1	53	**68.8**	86.5	900.0	52	70.8	63.9	900.0	52	69.2	47.9	814.7
20	20	600	46	73.1	7.3	900.1	48	**71.3**	79.9	900.0	46	73.9	8.5	900.1	47	71.9	8.0	855.2
20	25	600	43	74.6	9.1	900.0	45	**72.7**	47.0	900.0	44	74.6	440.7	900.2	44	73.2	7.4	868.9
20	40	600	38	77.2	9.9	900.0	39	**75.9**	29.4	900.0	38	77.8	12.4	900.3	38	76.5	142.2	900.0
20	60	600	34	79.4	434.1	900.2	36	**77.4**	592.2	900.2	35	79.2	293.9	900.3	35	78.0	179.8	900.0
20	80	600	32	80.2	15.8	900.4	33	**78.8**	27.9	900.2	32	80.7	21.4	900.2	32	79.5	88.7	900.2
20	100	600	31	80.7	24.9	900.5	32	**79.5**	56.2	900.2	31	81.0	10.2	900.2	31	80.1	37.5	900.1
20	150	600	28	83.9	36.3	900.0	29	**81.2**	41.1	900.0	29	82.4	74.3	900.9	28	81.8	26.2	900.2
20	200	600	27	84.3	43.2	900.5	28	**81.6**	20.9	900.0	28	82.9	856.6	900.7	27	82.2	387.0	900.3

Table 14
BL benchmark. Results when aiming for small gaps (averages over ten instances per row).

			A* + BS				A* + ACS				APS				A* + ACS-dist			
m	n	$\|\Sigma\|$	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}[s]$	$\bar{t}[s]$	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}[s]$	$\bar{t}[s]$	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}[s]$	$\bar{t}[s]$	$\overline{\|s\|}$	$\overline{gap}$[%]	$\overline{t_{\text{best}}}[s]$	$\bar{t}[s]$
10	100	4	34.0	**4.2**	0.3	797.6	34.1	10.8	1.8	900.0	34.1	8.4	0.6	898.9	34.1	6.7	1.4	645.0
10	100	12	12.7	**0.0**	0.3	2.7	12.7	**0.0**	1.6	8.2	12.7	**0.0**	0.5	4.1	12.7	**0.0**	0.3	2.4
10	100	20	7.9	**0.0**	0.1	0.1	7.9	**0.0**	0.1	0.1	7.9	**0.0**	0.1	0.1	7.9	**0.0**	0.0	0.0
10	500	4	180.7	39.3	113.1	901.3	185.4	37.7	337.6	900.0	181.2	39.5	28.2	900.0	185.3	**37.3**	160.8	656.6
10	500	12	76.7	57.9	24.1	900.0	79.0	56.6	216.1	900.1	77.2	58.3	180.3	900.0	79.1	**55.8**	163.0	795.5
10	500	20	49.6	64.1	5.9	900.0	51.2	62.7	205.8	900.1	50.1	64.8	16.1	900.1	51.3	**61.7**	72.4	900.0
10	1000	4	365.5	41.4	168.4	900.0	376.2	39.6	385.8	900.0	366.5	41.4	77.4	900.0	375.6	**39.5**	229.3	700.6
10	1000	12	157.1	61.1	113.5	900.0	162.7	59.6	273.3	900.1	158.2	61.2	135.2	900.0	162.1	**59.5**	111.2	742.0
10	1000	20	103.4	67.5	38.4	900.0	106.6	66.3	265.9	900.2	104.1	67.7	83.7	900.1	106.4	**66.0**	240.6	890.9
50	100	4	23.9	21.1	0.8	900.0	24.2	18.7	9.6	900.0	24.2	25.3	2.6	900.0	24.1	**18.0**	106.5	748.2
50	100	12	6.9	**0.0**	0.2	0.3	6.9	**0.0**	0.3	0.5	6.9	**0.0**	0.3	0.4	6.9	**0.0**	0.1	0.1
50	100	20	3.0	**0.0**	0.0	0.0	3.0	**0.0**	0.0	0.0	3.0	**0.0**	0.0	0.0	3.0	**0.0**	0.0	0.0
50	500	4	136.5	53.3	110.9	900.0	141.3	**51.3**	142.3	901.0	137.2	53.4	101.3	900.0	138.8	52.0	256.1	712.6
50	500	12	47.4	72.8	8.3	900.1	49.2	**71.3**	152.7	900.2	48.1	73.0	137.3	900.0	48.4	71.5	130.8	808.5
50	500	20	28.1	77.7	20.2	900.1	29.3	**76.2**	121.1	900.3	28.4	78.2	33.8	900.3	28.7	**76.2**	94.7	900.5
50	1000	4	278.3	55.0	157.3	900.0	289.8	**52.9**	410.9	900.0	278.9	55.0	96.1	900.0	284.2	53.7	332.7	756.7
50	1000	12	99.0	74.8	104.1	900.1	103.7	**73.4**	288.9	900.2	99.6	75.0	205.8	900.1	101.7	73.8	378.8	874.5
50	1000	20	60.0	80.4	98.4	900.2	62.6	**79.3**	134.3	900.4	60.7	80.6	132.0	900.3	61.3	79.6	320.7	901.0
100	100	4	21.6	24.6	1.3	900.0	22.0	20.4	16.1	900.1	22.1	28.1	12.9	900.0	22.0	**17.4**	80.6	813.9
100	100	12	5.2	**0.0**	0.1	0.1	5.2	**0.0**	0.0	0.1	5.2	**0.0**	0.2	0.2	5.2	**0.0**	0.0	0.0
100	100	20	2.1	**0.0**	0.0	0.0	2.1	**0.0**	0.0	0.0	2.1	**0.0**	0.0	0.0	2.1	**0.0**	0.0	0.0
100	500	4	127.3	56.3	96.3	900.0	131.4	**54.4**	250.1	900.0	127.7	56.5	91.2	900.0	128.0	55.4	242.5	857.0
100	500	12	41.8	75.4	88.1	900.1	43.2	**74.2**	185.7	900.2	42.0	76.0	30.5	900.2	42.0	74.6	131.8	900.0
100	500	20	24.2	80.1	17.0	900.2	24.8	**79.0**	156.8	900.6	24.2	80.7	14.8	900.4	24.0	79.3	152.9	900.1
100	1000	4	261.9	57.5	206.9	900.0	271.4	**55.7**	312.8	900.1	262.6	57.5	354.9	900.0	264.7	56.7	521.1	890.2
100	1000	12	88.9	77.3	53.2	900.1	92.7	**76.1**	363.4	900.3	89.1	77.5	186.5	900.2	89.7	76.7	368.2	900.0
100	1000	20	52.8	83.0	103.0	900.2	54.8	**81.6**	310.1	900.6	53.0	82.9	126.8	900.2	52.9	82.1	309.9	900.1
150	100	4	20.0	22.0	1.6	900.1	20.6	18.1	17.9	900.0	20.7	26.8	123.5	900.1	20.6	**15.1**	182.6	834.9
150	100	12	4.7	**0.0**	0.1	0.1	4.7	**0.0**	0.0	0.0	4.7	**0.0**	0.1	0.1	4.7	**0.0**	0.0	0.0
150	100	20	1.9	**0.0**	0.0	0.0	1.9	**0.0**	0.0	0.0	1.9	**0.0**	0.0	0.0	1.9	**0.0**	0.0	0.0
150	500	4	123.4	57.5	254.8	900.0	127.0	**55.8**	229.7	900.1	123.9	57.7	251.4	900.0	123.2	57.0	364.8	848.4
150	500	12	39.3	76.8	21.3	900.2	40.5	**75.6**	95.5	900.4	39.8	77.1	205.9	900.3	39.3	76.1	124.2	900.1
150	500	20	22.4	82.5	25.8	900.2	22.9	**80.3**	324.6	901.0	22.5	82.0	36.4	900.9	22.1	80.6	142.3	900.2
150	1000	4	254.1	58.7	141.1	900.0	263.3	**57.0**	340.2	900.1	254.6	58.8	293.6	900.1	255.0	58.2	284.5	872.9
150	1000	12	84.4	78.8	159.2	900.7	87.7	**77.2**	294.5	900.4	84.6	78.6	106.5	900.3	84.5	78.0	204.9	900.1
150	1000	20	49.7	84.1	101.7	900.6	51.0	**82.8**	155.6	901.0	49.8	83.8	117.6	900.6	49.0	83.3	309.8	900.3
200	100	4	19.7	23.9	2.3	900.0	20.0	20.2	47.5	900.1	20.1	26.8	56.4	900.1	19.8	**17.8**	134.5	878.7
200	100	12	4.1	**0.0**	0.1	0.1	4.1	**0.0**	0.0	0.1	4.1	**0.0**	0.1	0.1	4.1	**0.0**	0.0	0.0
200	100	20	1.1	**0.0**	0.0	0.0	1.1	**0.0**	0.0	0.0	1.1	**0.0**	0.0	0.0	1.1	**0.0**	0.0	0.0
200	500	4	121.0	58.4	190.1	900.1	124.0	**56.9**	187.4	900.1	121.3	58.6	133.5	900.1	120.3	58.0	186.4	880.4
200	500	12	38.0	77.4	50.6	900.2	38.9	**76.3**	84.2	900.4	38.0	78.0	31.9	900.5	37.5	77.0	214.4	900.1
200	500	20	21.0	83.5	28.0	900.1	21.7	**81.1**	256.0	901.3	21.1	83.0	111.8	901.1	20.9	81.4	288.6	900.2
200	1000	4	249.3	59.6	124.0	900.0	258.1	**57.7**	431.9	900.2	249.8	59.6	242.5	900.1	249.0	59.2	436.3	899.3
200	1000	12	81.7	79.5	130.9	900.2	84.6	**78.0**	311.7	900.7	81.9	79.2	187.4	900.4	81.1	78.8	332.9	900.1
200	1000	20	47.5	84.7	106.7	900.5	49.0	**83.3**	221.1	900.8	47.9	84.5	203.9	900.7	46.9	83.9	298.5	900.3

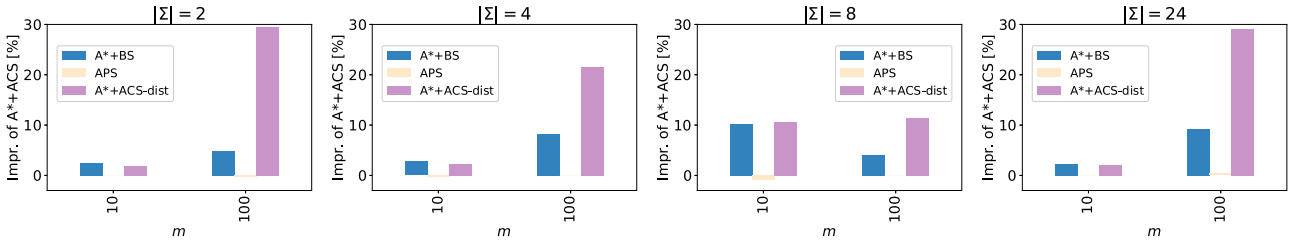


Fig. 12. Improvement of A* + ACS over the competitors in terms of gaps (in %) for benchmark set BB.

EX approximates the expected LCS length for unrelated random strings and is, for most of the considered benchmark instances, very well suited to lead ACS to promising heuristic solutions. The benefits of EX diminish, however, when instances with strongly related strings are considered, as for example in benchmark set BB. There, EX tends to become a disadvantage.

In future work it may be interesting to consider an approach that analyzes instances in a pre-processing phase in order to determine the most promising strategies and parameters to actually use. For example, one may roughly check the relatedness of the strings in order to decide whether or not EX shall be applied. Moreover, it would be good to find a more efficient way of filtering dominated solutions; this may make filtering actually beneficial also in those cases in which it did not pay off so far, especially within A* + ACS. Last but not least, coming up with an effective parallel implementation of our algorithms would be of high practical interest. Finally, note that A* + ACS is also a promising starting point for algorithms to solve other variants of the LCS problem such as the constrained LCS problem [55], the repetition-free LCS [56], the doubly-constrained LCS [57], and the restricted LCS [58].

CRediT authorship contribution statement

Marko Djukanovic: Methodology, Software, Writing - original draft. **Günther R. Raidl:** Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

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

Appendix A. Additional numerical results

See Tables 12–14.

Appendix B. Improvements of A* + ACS over other approaches

See Figs. 5–12.

References

[1] D. Maier, The complexity of some problems on subsequences and supersequences, J. ACM 25 (2) (1978) 322–336.
[2] A.S. Lhoussain, G. Hicham, Y. Abdellah, Adaptating the Levenshtein distance to contextual spelling correction, Int. J. Adv. Comput. Sci. Appl. 12 (1) (2015) 127–133.
[3] A. Zielezinski, S. Vinga, J. Almeida, W.M. Karlowski, Alignment-free sequence comparison: benefits, applications, and tools, Genome Biol. 18 (1) (2017) 186.
[4] K. Rieck, P. Laskov, K.-R. Müller, Efficient algorithms for similarity measures over sequential data: A look beyond kernels, in: Proccedings of DAGM 2006 – The 28th Joint Pattern Recognition Symposium, Springer, 2006, pp. 374–383.
[5] G. Sidorov, A. Gelbukh, H. Gómez-Adorno, D. Pinto, Soft similarity and soft cosine measure: Similarity of features in vector space model, Comput. Sist. 18 (3) (2014) 491–504.
[6] S. Kosub, A note on the triangle inequality for the jaccard distance, Pattern Recognit. Lett. 120 (2019) 36–38.
[7] L. Rabiner, A. Rosenberg, S. Levinson, Considerations in dynamic time warping algorithms for discrete word recognition, IEEE Trans. Acoust. Speech Signal Process. 26 (6) (1978) 575–582.
[8] Y. Ye, J. Jiang, B. Ge, Y. Dou, K. Yang, Similarity measures for time series data classification using grid representation and matrix distance, Knowl. Inf. Syst. 60 (2) (2019) 1105–1134.
[9] S. Wan, Y. Lan, J. Xu, J. Guo, L. Pang, X. Cheng, Match-SRNN: Modeling the recursive matching structure with spatial RNN, in: Proceedings of IJCAI'16 – The 25th International Joint Conference on Artificial Intelligence, AAAI Press, 2016, pp. 2922–2928.
[10] A. Islam, D. Inkpen, Semantic text similarity using corpus-based word similarity and string similarity, ACM Trans. Knowl. Discov. Data 2 (2) (2008) 1–25.
[11] T.K. Landauer, P.W. Foltz, D. Laham, An introduction to latent semantic analysis, Discourse Process. 25 (2–3) (1998) 259–284.
[12] T. Jiang, G. Lin, B. Ma, K. Zhang, A general edit distance between RNA structures, J. Comput. Biol. 9 (2) (2002) 371–388.
[13] J. Storer, Data Compression: Methods and Theory, Computer Science Press, MD, USA, 1988.
[14] R. Beal, T. Afrin, A. Farheen, D. Adjeroh, A new algorithm for "the LCS problem" with application in compressing genome resequencing data, BMC Genomics 17 (4) (2016) 544.
[15] J.B. Kruskal, An overview of sequence comparison: Time warps, string edits, and macromolecules, SIAM Rev. 25 (2) (1983) 201–237.
[16] P. Brisk, A. Kaplan, M. Sarrafzadeh, Area-efficient instruction set synthesis for reconfigurable system-on-chip design, in: Proceedings of DAC 2004 – The 41st Design Automation Conference, IEEE press, 2004, pp. 395–400.
[17] L. Bergroth, H. Hakonen, T. Raita, A survey of longest common subsequence algorithms, in: Proceedings of SPIRE 2000 – The 7th International Symposium on String Processing and Information Retrieval, IEEE, 2000, pp. 39–48.
[18] D. Gusfield, Algorithms on Strings, Trees, and Sequences, Computer Science and Computational Biology, Cambridge University Press, 1997.
[19] C.B. Fraser, Subsequences and Supersequences of Strings (Ph.D. thesis), University of Glasgow, Glasgow, UK, 1995.
[20] K. Huang, C. Yang, K. Tseng, Fast algorithms for finding the common subsequences of multiple sequences, in: Proceedings of ICS 2004 – The 9th International Computer Symposium, IEEE Press, 2004.
[21] C. Blum, M.J. Blesa, M. López-Ibáñez, Beam search for the longest common subsequence problem, Comput. Oper. Res. 36 (12) (2009) 3178–3186.
[22] S.R. Mousavi, F. Tabataba, An improved algorithm for the longest common subsequence problem, Comput. Oper. Res. 39 (3) (2012) 512–520.
[23] F.S. Tabataba, S.R. Mousavi, A hyper-heuristic for the longest common subsequence problem, Comput. Biol. Chem. 36 (2012) 42–54.
[24] Q. Wang, D. Korkin, Y. Shang, A fast multiple longest common subsequence (MLCS) algorithm, IEEE Trans. Knowl. Data Eng. 23 (3) (2011) 321–334.

[25] M. Djukanovic, G.R. Raidl, C. Blum, Anytime algorithms for the longest common palindromic subsequence problem, Comput. Oper. Res. 114 (2020) 104827.
[26] M. Djukanovic, G. Raidl, C. Blum, A beam search for the longest common subsequence problem guided by a novel approximate expected length calculation, in: Proceedings of LOD 2019 – The 5th International Conference on Machine Learning, Optimization, and Data Science, in: LNCS, Springer, 2019, in press.
[27] C. Blum, P. Festa, Longest common subsequence problems, in: Metaheuristics for String Problems in Bioinformatics, Wiley, 2016, pp. 45–60, chapter 3.
[28] H.-T. Chan, C.-B. Yang, Y.-H. Peng, The generalized definitions of the two-dimensional largest common substructure problems, in: Proceedings of the 33rd Workshop on Combinatorial Mathematics and Computation Theory, National Taiwan University, Department of Mathematics, 2016, pp. 1–12.
[29] Y. Li, Y. Wang, Z. Zhang, Y. Wang, D. Ma, J. Huang, A novel fast and memory efficient parallel MLCS algorithm for long and large-scale sequences alignments, in: IEEE 32nd International Conference on Data Engineering, 2016, pp. 1170–1181.
[30] Z. Peng, Y. Wang, A novel efficient graph model for the multiple longest common subsequences (MLCS) problem, Front. Genet. 8 (2017) 104.
[31] S. Zilberstein, Using anytime algorithms in intelligent systems, AI Mag. 17 (3) (1996) 73.
[32] S. Zilberstein, Operational rationality through compilation of anytime algorithms, AI Mag. 16 (2) (1995) 79.
[33] J. Yang, Y. Xu, G. Sun, Y. Shang, A new progressive algorithm for a multiple longest common subsequences problem and its efficient parallelization, IEEE Trans. Parallel Distrib. Syst. 24 (5) (2013) 862–870.
[34] J. Yang, Y. Xu, Y. Shang, G. Chen, A space-bounded anytime algorithm for the multiple longest common subsequence problem, IEEE Trans. Knowl. Data Eng. 26 (11) (2014) 2599–2609.
[35] Q. Wang, D. Korkin, Y. Shang, Efficient dominant point algorithms for the multiple longest common subsequence, MLCS problem, in: Proceedings of IJCAI'09 – The 25th International Joint Conference on Artificial Intelligence, 2009, pp. 1494–1499.
[36] S.G. Vadlamudi, P. Gaurav, S. Aine, P.P. Chakrabarti, Anytime column search, in: Proceedings of AI'12 – The 25th Australasian Joint Conference on Artificial Intelligence, Springer, 2012, pp. 254–265.
[37] M. Djukanovic, G. Raidl, C. Blum, Heuristic approaches for solving the longest common squared subsequence problem, in: Proceedings of EUROCAST 2019 – The 17th International Conference on Computer Aided Systems Theory, in: LNCS, Springer, 2019, in press.
[38] P. Hart, N. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Trans. Syst. Sci. Cybern. 4 (2) (1968) 100–107.
[39] T.L. Dean, Intractability and time-dependent planning, in: Proceedings of the 1986 Workshop on Reasoning About Actions & Plans, 1986, pp. 245–266.
[40] T.L. Dean, M.S. Boddy, An analysis of time-dependent planning, in: AAAI, vol. 88, 1988, pp. 49–54.
[41] W. Zhang, Complete anytime beam search, in: Proceedings of IAAI '98 – The 20th Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence, American Association for Artificial Intelligence, Menlo Park, CA, USA, 1998, pp. 425–430.
[42] R. Zhou, E.A. Hansen, Beam-stack search: Integrating backtracking with beam search, in: Proceedings of ICAPS 2005 – The 15th International Conference on Automated Planning and Scheduling, AAAI Press, 2005, pp. 90–98.
[43] E.A. Hansen, R. Zhou, Anytime heuristic search, J. Artificial Intelligence Res. 28 (2007) 267–297.
[44] M. Likhachev, G.J. Gordon, S. Thrun, ARA*: Anytime A* with provable bounds on sub-optimality, in: Advances in Neural Information Processing Systems, 2004, pp. 767–774.
[45] J. Van Den Berg, R. Shah, A. Huang, K. Goldberg, Anytime nonparametric A*, in: Proceedings of AAAI'11 – The 25th Conference on Artificial Intelligence, 2011.
[46] S. Aine, P. Chakrabarti, R. Kumar, AWA* – A window constrained anytime heuristic search algorithm, in: Proceedings of IJCAI'07 – The 12th International Joint Conference on Artificial Intelligence, 2007, pp. 2250–2255.
[47] S.G. Vadlamudi, S. Aine, P.P. Chakrabarti, MAWA* – A memory-bounded anytime heuristic-search algorithm, IEEE Trans. Syst. Man Cybern. B 41 (3) (2010) 725–735.
[48] G.K. Kao, E.C. Sewell, S.H. Jacobson, A branch, bound, and remember algorithm for the $1|r_i|\sum t_i$ scheduling problem, J. Sched. 12 (2) (2009) 163–175.
[49] S.G. Vadlamudi, S. Aine, P.P. Chakrabarti, Anytime pack search, Nat. Comput. 15 (3) (2016) 395–414.
[50] S.J. Shyu, C.-Y. Tsai, Finding the longest common subsequence for multiple biological sequences by ant colony optimization, Comput. Oper. Res. 36 (1) (2009) 73–91.
[51] T. Easton, A. Singireddy, A large neighborhood search heuristic for the longest common subsequence problem, J. Heuristics 14 (3) (2008) 271–283.
[52] C. Blum, M.J. Blesa, Probabilistic beam search for the longest common subsequence problem, in: T. Stützle, M. Birratari, H.H. Hoos (Eds.), Proceedings of SLS 2007 – The 1st International on Engineering Stochastic Local Search Algorithms, in: LNCS, vol. 4638, Springer, 2007, pp. 150–161.
[53] M. López-Ibáñez, J. Dubois-Lacoste, L.P. Cáceres, T. Stützle, M. Birattari, The irace package: Iterated racing for automatic algorithm configuration, Oper. Res. Perspect. 3 (2016) 43–58.
[54] J. Demšar, Statistical comparisons of classifiers over multiple data sets, J. Mach. Learn. Res. 7 (2006) 1–30.
[55] Z. Gotthilf, D. Hermelin, M. Lewenstein, Constrained LCS: Hardness and approximation, in: Proceedings of CPM 2008 – The 19th Annual Symposium on Combinatorial Pattern Matching, in: LNCS, vol. 5029, Springer, 2008, pp. 255–262.
[56] S.S. Adi, M. Iia D.V. Braga, C.G. Fernandes, C.E. Ferreira, F.V. Martinez, M.-F. Sagot, M.A. Stefanes, C. Tjandraatmadja, Y. Wakabayashi, Repetition-free longest common subsequence, Discrete Appl. Math. 158 (12) (2010) 1315–1324.
[57] P. Bonizzoni, G. Della Vedova, R. Dondi, Y. Pirola, Variants of constrained longest common subsequence, Inform. Process. Lett. 110 (20) (2010) 877–881.
[58] Z. Gotthilf, D. Hermelin, G.M. Landau, M. Lewenstein, Restricted LCS, in: Proceedings of SPIRE 2010 – The 17th International Symposium on String Processing and Information Retrieval, in: LNCS, vol. 6394, Springer, 2010, pp. 250–257.