A Beam Search for the Longest Common Subsequence Problem Guided by a Novel Approximate Expected Length Calculation

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

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

Abstract. The longest common subsequence problem (LCS) aims at finding a longest string that appears as subsequence in each of a given set of input strings. This is a well known $\mathcal{NP}$-hard problem which has been tackled by many heuristic approaches. Among them, the best performing ones are based on beam search (BS) but differ significantly in various aspects. In this paper we compare the existing BS-based approaches by using a common BS framework making the differences more explicit. Furthermore, we derive a novel heuristic function to guide BS, which approximates the expected length of an LCS of random strings. In a rigorous experimental evaluation we compare all BS-based methods from the literature and investigate the impact of our new heuristic guidance. Results show in particular that our novel heuristic guidance leads frequently to significantly better solutions. New best solutions are obtained for a wide range of the existing benchmark instances.
Keywords: string problems; expected value; beam search.

1 Introduction

We define a *string* s as a finite sequence of $|s|$ characters from a finite alphabet Σ. Strings are widely used for representing sequence information. Words, and even whole texts, are naturally stored by means of strings. In the field of bioinformatics, DNA and protein sequences, for example, play particularly important roles. A frequently occurring necessity is to detect similarities between several strings in order to derive relationships and possibly predict diverse aspects of some strings. A *sequence* of a string s is any sequence obtained by removing an arbitrary number of characters from s. A natural and common way to compare two or more strings is to find *common subsequences*. More specifically, given a set of m input strings $S = \{s_1, \ldots, s_m\}$, the *longest common subsequence* (LCS) problem [16] aims at finding a subsequence of maximal length which is common for all the strings in S. Apart from applications in computational biology [13], this problem appears, for example, also in data compression [19,1], text editing [14], and the production of circuits in field programmable gate arrays [6].

The LCS problem is $\mathcal{NP}$-hard for an arbitrary number (m) of input strings [16]. However, for fixed m polynomial algorithms based on dynamic programming (DP) are known [10]. Standard dynamic programming approaches run in $O(n^m)$ time, where n is the length of the longest input string. This means that these exact methods become quickly impractical when m grows and n is not small. In practice, heuristics optimization techniques are therefore frequently used. Concerning simple approximate methods, the expansion algorithm [5] and the Best-Next heuristic [9,11] are well known construction heuristics.

A break-through in terms of both, computation time and solution quality was achieved by the *Beam Search* (BS) of Blum et al. [3]. This algorithm is an incomplete tree search which relies on a solution construction mechanism based on the Best-Next heuristic and exploits bounding information using a simple upper bound function to prune non promising solutions. The algorithm was able to outperform all existing algorithms at the time of its presentation. Later, Wang et al. [21] proposed a fast A*-based heuristic utilizing a new DP-based upper bound function. Mousavi and Tabataba [17] proposed a variant of the BS which uses a probability-based heuristic and a different pruning mechanism. Moreover, Tabataba et al. [20] suggested a hyper-heuristic approach which makes use of two different heuristics and applies a beam search with low beam width first to make the decision about which heuristic to use in a successive BS with higher beam width. This approach is currently state-of-the-art for the LCS problem.

Recently, a *chemical reaction optimization* [12] was also proposed for the LCS and the authors claimed to achieve new best results for some of the benchmark instances. We gave our best to re-implement their approach but were not successful due to many ambiguities and mistakes found within the algorithm's description and open questions that could not be resolved from the paper. The authors of the paper also were not able to provide us the original implementation of their approach or enough clarification. Therefore, we exclude this algorithm from further consideration in our experimental comparison in this article.

In conclusion, the currently best performing heuristic approaches to solve also large LCS problem instances are thus based on BS guided by different heuristics and incorporate different pruning mechanisms to omit nodes that likely lead to weaker suboptimal solutions. More detailed conclusions are, however, difficult as the experimental studies in the literature are limited and partly questionable: On the one hand, in [20,3] mistakes in earlier works have been reported, whose impact on the final solution quality are not known. On the other hand, the algorithms have partly been tested on different LCS instance sets and/or the methods were implemented in different programming languages and the experiments were performed on different machines.

In our work we re-implemented all the mentioned BS-based methods from the literature with their specific heuristics and pruning mechanisms within a common BS framework in order to rigorously compare the methods on the same sets of existing benchmark instances. Furthermore, we derive a novel heuristic function for guiding BS that computes an approximate expected length of an LCS. This function is derived from our previous work done for the *palindromic* version of the LCS problem [7]). The experimental evaluation shows that this

heuristic is for most of the benchmark sets a significantly better guidance than the other so far used heuristic functions, and we obtain new best known results on several occasions. An exception is only one benchmark set were the input strings are artificially created in a strongly correlated way.

The rest of the paper is organized as follows. Section 2 describes the state graph to be searched for solving the LCS problem. In Section 3 we present our united BS framework which covers all the considered BS-based algorithms from the literature. We further propose our novel heuristic function for approximating the expected length for the LCS here. Section 4 reports the main results of our rigorous experimental comparison. We finally conclude with Section 5 where also some directions for promising future work are sketched.

2 State Graph

We start this section by introducing some common notation. Let n, as already stated, be the maximum length of the strings in S. The j-th letter of a string s is denoted by $s[j]$, $j = 1, \ldots, |s|$, and let $s_1 \cdot s_2$ denote the concatenation obtained by appending string s_2 to string s_1. By $s[j, j']$, $j \leq j'$, we denote the continuous subsequence of s starting at position j and ending at position j'; if $j > j'$, $s[j, j']$ is the empty string ε. Finally, let $|s|_a$ be the number of occurrences of letter $a \in \Sigma$ in string s. Henceforth, a string s is called a (valid) *partial solution* concerning input strings $S = \{s_1, \ldots, s_m\}$, if s is a common subsequence of the strings in S.

Let $p^{\mathrm{L}} \in \mathbb{N}^m$ be an integer valued vector such that $1 \leq p_i^{\mathrm{L}} \leq |s_i|$, for $i = 1, \ldots, m$. Given such vector p^{L}, the set $S[p^{\mathrm{L}}] := \{s_i[p_i^{\mathrm{L}}, |s_i|] \mid i = 1, \ldots, m\}$ consists of each original string's continuous subsequence from the position given in p^{L} up to the end. We call vector p^{L} *left position vector*, and it represents the LCS subproblem on the strings $S[p^{\mathrm{L}}]$. Note that the original problem can be denoted by $S[(1, \ldots, 1)]$. We are, however, not interested in all possible subproblems but just those which are *induced* by (meaningful) partial solutions. A partial solution s induces the subproblem $S[p^{\mathrm{L}}]$ for which

- $s_i[1, p_i^{\mathrm{L}} - 1]$ is the minimal string among strings $s_i[1, x], x = 1, \ldots, p^{\mathrm{L}} - 1$ such that it contains s as a subsequence, for all $i = 1, \ldots, m$.

For example, if $S = \{\texttt{abcbacb}, \texttt{accbbaa}\}$, then for the partial solution $s = \texttt{acb}$ the induced subproblem is represented by $p^{\mathrm{L}} = (5, 5)$.

The state graph of the LCS problem is a directed acyclic graph $G = (V, A)$, where a node $v \in V$ is represented by the respective left position vector $p^{\mathrm{L},v}$ and by the length l_v of the partial solution which induces the corresponding subproblem $S[p^{\mathrm{L},v}]$, i.e. $v = (p^{\mathrm{L},v}, l_v)$. An arc $a = (v_1, v_2) \in A$ with a label $\ell(a) \in \Sigma$ exists if l_{v_2} is by one larger than l_{v_1} and if the partial solution obtained by appending letter $\ell(a)$ to the partial solution of v_1 induces the subproblem $S[p^{\mathrm{L},v_2}]$. The root node r of the state graph G corresponds to the original problem, which can be said to be induced by the empty partial solution ε, i.e., $r = ((1, \ldots, 1), 0)$. In order to derive the successor nodes of a node $v \in V$ in G, we first determine

all the letters by which partial solutions inducing $S[p^{\mathrm{L},v}]$ can be feasibly extended, i.e., all letters $a \in \Sigma$ appearing at least once in each string in $S[p^{\mathrm{L},v}]$. Let Σ_v denote this set of feasible extension letters w.r.t. node 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 $p_{i,a}^{\mathrm{L},v}$, $i = 1, \ldots, m$. We can in general reduce Σ_v by disregarding *dominated* letters. We say letter $a \in \Sigma_v$ dominates letter $b \in \Sigma_v$ iff $p_{i,a}^{\mathrm{L},v} \leq p_{i,b}^{\mathrm{L},v}$ for all $i = 1, \ldots, m$. Dominated letters can safely be ignored since they always lead to suboptimal solutions as they "skip" some other letter that can be used before. Let Σ_v^{nd} be the obtained set of feasible and non-dominated letters for node v. For each letter $a \in \Sigma_v^{\mathrm{nd}}$ we derive a successor node $v' = (p^{\mathrm{L},v'}, l_v + 1)$, where $p_i^{\mathrm{L},v'} = p_{i,a}^{\mathrm{L},v'} + 1$, $i = 1, \ldots, m$. Each node that receives no successor node, i.e., where $\Sigma_v^{\mathrm{nd}} = \emptyset$, is called a *complete* node. Now, note that any path from the root node r to any node in V represents the feasible partial solution given by the sequence of labels of the traversed arcs. Any path from r to a complete node represents a common subsequence of S that cannot be further extended, and an LCS is therefore given by a longest path from r to any complete node.

3 Beam Search Framework

In the literature for the LCS problem, the so far leading algorithms to approach larger instances heuristically are all based on Beam Search (BS). This essentially is an incomplete tree search which expands nodes in a breadth-first search manner. A collection of nodes, called the *beam*, is maintained. Initially, the beam contains just the root node r. In each major iteration, BS expands all nodes of the beam in order to obtain the respective successor nodes at the next level. From those the $\beta > 0$ most promising nodes are selected to become the beam for the next iteration, where β is a strategy parameter called beam width. This expansion and selection steps are repeated level by level until the beam becomes empty. We will consider several ways to determine the most promising nodes to be kept at each step of BS in Section 3.1. The BS returns the partial solution of a complete node with the largest l_v value discovered during the search. The main difference among BS approaches from the literature are the heuristic functions used to evaluate LCS nodes for the selection of the beam and pruning mechanisms to recognize and discard dominated nodes. A general BS framework for the LCS is shown in Algorithm 1.

ExtendAndEvaluate(B, h) derives and collects the successor nodes of all $v \in B$ and evaluates them by heuristic h, generating the set of extension nodes V_{ext} ordered according to non-increasing h-values. Prune$(V_{\mathrm{ext}}, ub_{\mathrm{prune}})$ optionally removes any dominated node v for which $l_v + ub_{\mathrm{prune}}(v) \leq |s_{\mathrm{lcs}}|$, where $ub_{\mathrm{prune}}(\cdot)$ is an upper bound function for the number of letters that may possibly still be appended, or in other words an upper bound for the LCS of the corresponding remaining subproblem, and $|s_{\mathrm{lcs}}|$ is the length of the so far best solution. Filter$(V_{\mathrm{ext}}, k_{\mathrm{best}})$ is another optional step. It removes nodes corresponding to dominated letters as defined in Section 2, but in a possibly restricted way controlled by parameter k_{best} in order to limit the spent computing effort. More concretely, the dominance relationship is checked for each node $v \in V_{\mathrm{ext}}$ against

Algorithm 1 BS framework for the LCS problem

Input: an instance (S, Σ), heuristic function h to evaluate nodes; upper bound function ub_{prune} to prune nodes; parameter k_{best} to filter nodes (non-dominance relation check); β: beam size (and others depending on the specific algorithm)
Output: a feasible LCS solution
$B \leftarrow \{r\}$
$s_{\text{lcs}} \leftarrow \varepsilon$
while $B \neq \emptyset$ **do**
    $V_{\text{ext}} \leftarrow \text{ExtendAndEvaluate}(B, h)$
    update s_{lcs} if a complete node v with a new largest l_v value reached
    $V_{\text{ext}} \leftarrow \text{Prune}(V_{\text{ext}}, ub_{\text{prune}})$ // optional
    $V_{\text{ext}} \leftarrow \text{Filter}(V_{\text{ext}}, k_{\text{best}})$ // optional
    $B \leftarrow \text{Reduce}(V_{\text{ext}}, \beta)$
end while
return s_{lcs}

the k_{best} most promising nodes from V_{ext}. Last but not least, $\text{Reduce}(V_{\text{ext}}, \beta)$ returns the new beam consisting of the β best ranked nodes in V_{ext}.

3.1 Functions for Evaluating Nodes

In the literature, several different functions are used for evaluating and pruning nodes, i.e., for h and ub_{prune}. In the following we summarize them.

Fraser [9] used as upper bound on the number of letters that might be further added to a partial solution leading to a node v—or in other words the length of an LCS of the induced remaining subproblem $S[p^{\text{L},v}]$–by $\text{UB}_{\min}(v) = \text{UB}_{\min}(S[p^{\text{L}}]) = \min_{i=1,\ldots,m}(|s_i| - p_i^{\text{L},v} + 1)$.

Blum et al. [3] suggested the upper bound $\text{UB}_1(v) = \text{UB}_1(S[p^{\text{L},v}]) = \sum_{a \in \Sigma} c_a$, with $c_a = \min_{i=1,\ldots,m} |s_i[p_i^{\text{L},v}, |s_i|]|_a$, which dominates $\text{UB}_{\min}$. While UB_1 is efficiently calculated using smart preprocessing in $O(m \cdot |\Sigma|)$, it is still a rather weak bound. The same authors proposed the following ranking function $\text{Rank}(v)$ to use for heuristic function h. When expanding a node v, all the successors v' of v are ranked either by $\text{UB}_{\min}(v')$ or by $g(v, v') = \left(\sum_{i=1}^{m} \frac{p_i^{\text{L},v'} - p_i^{\text{L},v} - 1}{|s_i| - p_i^{\text{L},v}} \right)^{-1}$. If v' has the largest $\text{UB}_{\min}(v')$ (or $g(v, v')$) value among all the successors of v, it receives rank 1, the successor with the second largest value among the successors receives rank 2, etc. The overall value $\text{Rank}(v)$ is obtained by summarizing all the ranks along the path from the root node to the node corresponding to the partial solution. Finally, the nodes in V_{ext} are sorted according to non-increasing values $\text{Rank}(v)$ (i.e., smaller values preferable here).

Wang et al. [21] proposed a DP-based upper bound using the LCS for two input strings, given by

$$\text{UB}_2(v) = \text{UB}_2(S[p^{\text{L},v}]) = \min_{i=1,\ldots,m-1} |\text{LCS}(s_i[p_i^{\text{L},v}, |s_i|], s_{i+1}[p_{i+1}^{\text{L},v}, |s_{i+1}|])|.$$

Even if not so obvious, UB_2 can be calculated efficiently in time $O(m)$ by creating appropriate data structures in preprocessing. By combining the

two upper bounds we obtain the so far tightest known bound $\mathrm{UB}(v) = \min(\mathrm{UB}_1(v), \mathrm{UB}_2(v))$ that can still efficiently be calculated in $O(m \cdot |\Sigma|)$ time. This bound will serve in Prune() of our BS framework, since it can filter more non-promising nodes than when UB_1 or UB_2 are just individually applied.

Mousavi and Tabataba [17,20] proposed two heuristic guidances. The first estimation is derived by assuming that all input strings are uniformly at random generated and that they are mutually independent. The authors derived a recursion which determines the probability $\mathcal{P}(p, q)$ that a uniform random string of length p is a subsequence of a string of length q. These probabilities can be calculated during preprocessing and are stored in a matrix. For some fixed k, using the assumption that the input strings are independent, each node is evaluated by $\mathrm{H}(v) = \mathrm{H}(S[p^{\mathrm{L},v}]) = \prod_{i=1}^{m} \mathcal{P}(k, |s_i| - p_i^{\mathrm{L},v} + 1)$. This corresponds to the probability that a partial solution represented by v can be extended by k letters. The value of k is heuristically chosen as $k := \max\left(1, \left\lfloor \frac{1}{|\Sigma|} \cdot \min_{v \in V_{\mathrm{ext}},\, i=1,\ldots,m}(|s_i| - p_i^{\mathrm{L},v} + 1) \right\rfloor\right)$. The second heuristic estimation, the so called *power* heuristic, is proposed as follows:

$$\mathrm{Pow}(v) = \mathrm{Pow}(S[p^{\mathrm{L},v}]) = \left(\prod_{i=1}^{m} (|s_i| - p_i^{\mathrm{L},v} + 1) \right)^{q} \cdot \mathrm{UB}_{\min}(v),\ q \in [0, 1).$$

It can be seen as a generalized form of $\mathrm{UB}_{\min}$. The authors argue to use smaller values for q in case of larger m and specifically set $q = a \times \exp(-b \cdot m) + c$, where $a, b, c \geq 0$ are then instance-independent parameters of the algorithm.

3.2 A Heuristic Estimation of the Expected Length of an LCS

By making use of the matrix of probabilities $\mathcal{P}(p, q)$ from [17] for a uniform random string of length p to be a subsequence of a random string of length q and some basic laws from probability theory, we derive an approximation for the expected length of a LCS of $S[p^{\mathrm{L},v}]$ as follows. Let Y be the random variable which corresponds to the length of an LCS for a set S of uniformly at random generated strings. This value cannot be larger than the length of the shortest string in S, denoted by $l_{\max} = \min_{i=1,\ldots,m}(|s_i| - p_i^{\mathrm{L},v} + 1)$. Let us enumerate all sequences of length k over alphabet Σ; trivially, there are $|\Sigma|^k$ such sequences. We now make the simplifying assumption that for any sequence of length k over Σ, the event that the sequence is a common subsequence of all strings in S is independent of the corresponding events for other sequences. Let $Y_k \in \{0, 1\}$, $k = 0, \ldots, l_{\max}$, be a binary random variable which denotes the event that S has a common subsequence of length at least k. If S has a common subsequence of length $k + 1$, this implies that it has a common subsequence of length k as well. Consequently, we get that $\Pr[Y = k] = \mathbb{E}[Y_k] - \mathbb{E}[Y_{k+1}]$ and

$$\mathbb{E}[Y] = \sum_{k=1}^{l_{\max}} k \cdot \Pr[Y = k] = \sum_{k=1}^{l_{\max}} k \cdot (\mathbb{E}[Y_k] - \mathbb{E}[Y_{k+1}]) = \sum_{k=1}^{l_{\max}} \mathbb{E}[Y_k]. \qquad (1)$$

The probability that the input strings of S have no common subsequence of length k is equal to $1 - \mathbb{E}[Y_k]$. Due to our assumption of independence, this

probability is equal to $(1 - \prod_{i=1}^{m} \mathcal{P}(k, |s_i|))^{|\Sigma|^k}$. Finally, for any node $v \in V$, we obtain the approximate expected length

$$\mathrm{EX}(v) = \mathrm{EX}(S[p^{\mathrm{L},v}]) = \sum_{k=1}^{l_{\max}} \left(1 - \left(1 - \prod_{i=1}^{m} \mathcal{P}(k, |s_i| - p_i^{\mathrm{L},v} + 1)\right)^{|\Sigma|^k}\right). \quad (2)$$

Since a direct numerical calculation would yield too large intermediate values for a common floating point arithmetic when k is large, we use the decomposition

$$A^{|\Sigma|^k} = \left(\underbrace{\left(\cdots (A)^{|\Sigma|^p} \cdots\right)^{|\Sigma|^p}}_{\lfloor k/p \rfloor}\right)^{|\Sigma|^{(k \bmod p)}}$$

for any expression A with $p = 20$. Moreover, the calculation of (2) can in practice be efficiently done in $O(m \log(n))$ time by exploiting the fact that the terms under the sum present a decreasing sequence of values within $[0, 1]$ and many values are very close to zero or one: We apply a divide-and-conquer principle for detecting the values $\left(1 - \prod_{i=1}^{m} \mathcal{P}(k, |s_i| - p_i^{\mathrm{L},v} + 1)\right)^{|\Sigma|^k} \in (\epsilon, 1 - \epsilon)$ with the threshold $\epsilon = 10^{-6}$. Furthermore, note that if the product which appears in (2) is close to zero, this might cause further numerical issues. These are resolved by replacing the term $\left(1 - \prod_{i=1}^{m} \mathcal{P}(k, |s_i| - p_i^{\mathrm{L},v} + 1)\right)^{|\Sigma|^k}$ with an approximation derived from the Taylor expansion of $(1 - x)^{\alpha}$; details are described in [7].

3.3 Expressing Existing Approaches in Terms of our Framework

All BS-related approaches from the literature (see Section 1) can be defined as follows within our framework from Algorithm 1.

BS by Blum et al. [3]. The heuristic function h is set to $h = \mathrm{Rank}^{-1}$. Function $\mathrm{Prune}(V_{\mathrm{ext}}, ub_{\mathrm{prune}})$ uses $ub_{\mathrm{prune}} = \mathrm{UB}_1$. Moreover, all nodes that are not among the $\mu \cdot \beta$ most promising nodes from V_{ext} are pruned. Hereby, $\mu \geq 1$ is an algorithm-specific parameter. Finally, with a setting of $k_{\mathrm{best}} \geq \beta \cdot |V_{\mathrm{ext}}|$ in function $\mathrm{Filter}(V_{\mathrm{ext}}, k_{\mathrm{best}})$, the original algorithm is obtained. Instead of testing this original algorithm, we study here an improved version that uses $ub_{\mathrm{prune}} = \mathrm{UB}$. Moreover, during tuning (see below) we will consider also values for k_{best} such that $k_{\mathrm{best}} < \beta \cdot |V_{\mathrm{ext}}|$. The resulting method is henceforth denoted by Bs-Blum.

Heuristic by Wang [21]. $h = \mathrm{UB}_2$ is used as heuristic function. Moreover, a priority queue to store extensions is used instead of the standard vector structure used in the implementation of other algorithms. Function $\mathrm{Prune}(V_{\mathrm{ext}}, ub_{\mathrm{prune}})$ removes all those nodes from V_{ext} whose h-values deviate more than $W \geq 0$ units from the priority value of the most promising node of V_{ext}. Filtering is not used. Instead of $h = \mathrm{UB}_2$ (as in the original algorithm) we use here $h = \mathrm{UB}$, which significantly improves the algorithm henceforth denoted by Bs-Wang.

BS approaches by Mousavi and Tabataba [17,20]. The first approach, denoted by Bs-H, uses $h = \mathrm{H}$, whereas in the second one, denoted by Bs-Pow, $h = \mathrm{Pow}$ is used. No pruning is done. Finally, a restricted filtering ($k_{\mathrm{best}} > 0$) is applied.

The Hyper-heuristic approach by Mousavi and Tabataba [20]. This approach, henceforth labeled Hh, combines heuristic functions H and Pow as follows. First, Bs-H and Bs-Pow are both executed using a low beam width $\beta_h > 0$. Based on the outcome of these two executions, either Bs-H or Bs-Pow will be selected as the final method executed with a beam width $\beta \gg \beta_h$. The result is the best solution found in both phases.

4 Experimental Evaluation

The presented BS framework was implemented in C++ and all experiments were performed in single-threaded mode on an Intel Xeon E5-2640 with 2.40GHz and 16 GB of memory. In addition to the five approaches from the literature as detailed in Section 3.3, we test our own approach, labeled Bs-Ex, which uses $h = \mathrm{EX}$, no pruning, and a restricted filtering.

The related literature offers six different benchmark sets for the LCS problem. The `ES` benchmark, introduced by Easton and Singireddy [8], consists of 600 instances of different sizes in terms of the number and the length of the input strings, and in terms of the alphabet size. A second benchmark consists of three groups of 20 instances each: `Random`, `Rat` and `Virus`. It was introduced by Shyu and Tsai [18] for testing their ant colony optimization algorithm. Hereby, `Rat` and `Virus` consist of sequences from rat and virus genomes. The `BB` benchmark of 80 instances was generated by Blum and Blesa [2] in a way such that a large similarity between the input strings exists. Finally, the `BL` instance set [4] consists of 450 problem instances that were generated uniformly at random.

The final solution quality produced by any of the considered BS methods is largely determined by the beam size parameter β. We decided to test all algorithms with a setting aiming for a low computation time ($\beta = 50$) and with a second setting aiming for a high solution quality ($\beta = 600$). The first setting is henceforth called the *low-time* setting, and the second one the *high-quality* setting. Note that when using the same value of β, the considered algorithms expand a comparable number of nodes. The remaining parameters of the algorithms are tuned by *irace* [15] for the *high-quality* setting. Separate tuning runs with a budget of 5000 algorithm applications are performed for benchmark instances in which the input strings have a random character (`ES`, `Random`, `Rat`, `Virus`, `BL`),[1] and for the structured instances from set `BB`. 30 training instances are used for the first tuning run and 20 for the second one.

The outcome reported by *irace* for random instances is as follows. Bs-Blum makes use of function $g(.,.)$ within $h = \mathrm{Rank}^{-1}$. Moreover, $\mu = 4.0$ and $k_{\mathrm{best}} = 5$ are used. Bs-Wang uses $W = 10$. For Bs-H we obtain $k_{\mathrm{best}} = 50$, for Bs-Pow

[1] Note that even instance sets `Rat` and `Virus` contain sequences that are close to random strings.

we get $k_{\text{best}} = 100$, $a = 1.677$, $b = 0.054$, and $c = 0.074$. Finally, HH uses $\beta_h = 50$, and for BS-EX we get $k_{\text{best}} = 100$.

For the structured instances from set BB *irace* reports the following. BS-BLUM makes use of UB_{min} within $h = \text{Rank}^{-1}$. Moreover, it uses $\mu = 4.0$ and $k_{\text{best}} = 1000$. BS-WANG uses $W = 10$, BS-H needs $k_{\text{best}} = 100$, and BS-POW requires $k_{\text{best}} = 100$, $a = 1.823$, $b = 0.112$, and $c = 0.014$. Finally, HH uses $\beta_h = 50$ and for BS-EX $k_{\text{best}} = 100$. At this point we want to emphasize that we made sure that the five re-implemented competitor algorithms obtain equivalent (and often even better) results on all benchmark sets than those reported in the original papers.

We now proceed to study the numerical results presented in Tables 1–5. In each table, the first three columns describe the respective instances in terms of the alphabet size ($|\Sigma|$), the number of input strings (n), and the maximum string length (m). Columns 4–8 report the results obtained with the *low-time* setting, while columns 9–13 report on the results of the *high-quality* setting. The first three columns of both blocks provide the results of the best performing algorithm among the five competitors from the literature. Listed are for each instance (or instance group) the (average) solution length, the respective (average) computation time, and the algorithm that achieved this result. The last two columns of both blocks present the (average) solution length and the (average) computation time of our new BS-EX. The overall best result of each comparison is indicated in bold font, and an asterisk indicates that this result is better than the so-far best known one from the literature. These results allow to make the following observations.

- Concerning the *low-time* setting of the algorithms, the approaches from the literature compare as follows. BS-H and BS-POW seem to outperform the other approaches in the context of benchmarks `Rat` and `Virus`, with BS-BLUM and BS-WANG gaining some terrain when moving towards the alphabet size of $|\Sigma| = 20$. Furthermore, HH and—to some extent—BS-H dominate the remaining approaches in the context of benchmarks `ES` and `BL`. Concerning the structured instances from set `BB` the picture is not so clear. Here, the oldest BS approach (BS-BLUM) is able to win over the other approaches in three out of seven cases.
- The results obtained by BS-EX with the *low-time* setting are comparable to the best results obtained by the methods from the literature. More specifically, BS-EX produces comparable results for `Virus` and `Rat` and is able to outperform the other approaches in the context of `ES` and `BL`. As could be expected, for the `BB` instance set, in which the input strings have a strong relation to each other, the EX guiding function cannot successfully guide the search. This is because EX assumes the input strings to be random strings, that is, to be independent of each other.
- Concerning the results obtained with the *high-quality* setting, the comparison of the algorithms from the literature can be summarized as follows. For all benchmark sets (apart from `BB`) the best performance is shown by BS-H and/or BS-POW. For benchmark set `BB` the picture is, again, not so clear, with BS-BLUM gaining some terrain.

Table 1. Results on benchmark set `Rat`.

			low-time, literature			*low-time*, Bs-Ex		*high-quality*, literature			*high-quality*, Bs-Ex	
$\|\Sigma\|$	n	m	$\|\overline{s}_{\text{best}}\|$	$\bar{t}_{\text{best}}$	Algo.	$\|\overline{s}_{\text{best}}\|$	$\bar{t}[s]$	$\|\overline{s}_{\text{best}}\|$	$\bar{t}_{\text{best}}$	Algo.	$\|\overline{s}_{\text{best}}\|$	$\bar{t}[s]$
4	600	10	**201**	0.09	Bs-Pow	198	0.22	204	1.18	Bs-Pow	*__205__	3.09
4	600	15	**182**	0.10	Bs-Pow	**182**	0.18	184	0.62	Bs-H	*__185__	2.65
4	600	20	**169**	0.05	Bs-Pow	168	0.15	170	0.94	Bs-Pow	*__172__	2.25
4	600	25	166	0.12	Bs-Pow	**167**	0.18	168	1.01	Bs-Pow	*__170__	2.71
4	600	40	**151**	0.04	Bs-H	146	0.15	150	1.02	Bs-Pow	**152**	1.81
4	600	60	149	0.10	Bs-Pow	**150**	0.17	151	1.16	Bs-Pow	*__152__	2.27
4	600	80	**137**	0.05	Bs-H	**137**	0.17	139	0.67	Bs-H	*__142__	2.47
4	600	100	**133**	0.07	Bs-Pow	131	0.14	135	0.47	Bs-H	*__137__	2.50
4	600	150	125	0.06	Bs-H	**127**	0.13	126	0.91	Bs-Pow	*__129__	1.97
4	600	200	**121**	0.09	Bs-Pow	**121**	0.17	*__123__	0.70	Bs-Pow	*__123__	2.65
20	600	10	**70**	0.09	Bs-H	**70**	0.37	*__71__	1.86	Bs-H	*__71__	3.44
20	600	15	61	0.15	Bs-Pow	**62**	0.28	62	1.40	Bs-H	*__63__	2.55
20	600	20	**53**	0.12	Bs-Pow	**53**	0.20	**54**	1.15	Bs-H	**54**	2.45
20	600	25	**50**	0.22	Bs-Wang	**50**	0.21	51	1.09	Bs-H	*__52__	2.94
20	600	40	**48**	0.09	Bs-H	47	0.19	**49**	1.15	Bs-Blum	**49**	2.97
20	600	60	**46**	0.09	Bs-H	**46**	0.20	**47**	1.61	Bs-Pow	46	2.42
20	600	80	**43**	0.18	Bs-Blum	41	0.21	*__44__	1.14	Bs-H	43	2.64
20	600	100	**38**	0.11	Bs-Pow	**38**	0.23	39	0.96	Bs-H	*__40__	2.54
20	600	150	**36**	0.32	Bs-Blum	**36**	0.14	**37**	5.11	Bs-Wang	**37**	2.03
20	600	200	**34**	0.10	Bs-Pow	**34**	0.18	**34**	2.62	Bs-Blum	**34**	2.74

- Bs-Ex with the *high-quality* setting outperforms the other approaches from the literature on all benchmark sets except for `BB`, the latter again due to the strong correlation of the strings. In fact, in 48 out of 67 cases (concerning benchmarks `Rat`, `Virus`, `ES` and `BL`) Bs-Ex is able to obtain a new best-known result. Moreover, in most of the remaining cases, the obtained result is equal to the so-far best known one.
- Concerning the run times of the approaches, the calculation of EX is done in $O(m \log n)$ time and, therefore, is a bit more expensive when compared to the simpler UB, H, or Pow. However, this is not a significant issue since almost all runs completed within rather short times of usually a few seconds up to less than two minutes.
- We performed Wilcoxon signed-rank tests with an error level of 5% to check the significance of differences in the results of the approaches. These indicate that the solutions of the *high-quality* Bs-Ex are in the expected case indeed significantly better than those obtained from the *high-quality* state-of-the-art approaches from the literature for all except for the `Virus` benchmark, where no conclusion can be drawn, and the `BB` benchmark, where the Bs-Ex results are significantly worse due to the strong relationship among the sequences.

Overall, the numerical results clearly show that EX is a better guidance for BS than the heuristics and upper bounds used in former work, as long as (near-) independence among the input strings is given. Finally, note that the result for `Random` and for the instances with $|\Sigma| > 4$ of set `BL` are not provided due to page limitations. Full results can be found at `https://www.ac.tuwien.ac.at/files/resources/instances/LCS/LCS-report.zip`.

Table 2. Results on benchmark set `Virus`.

			low-time, literature			*low-time*, Bs-Ex		*high-quality*, literature			*high-quality*, Bs-Ex	
$\|\Sigma\|$	n	m	$\|\overline{s}_{\text{best}}\|$	$\overline{t}_{\text{best}}$	Algo.	$\|\overline{s}_{\text{best}}\|$	$\overline{t}[s]$	$\|\overline{s}_{\text{best}}\|$	$\overline{t}_{\text{best}}$	Algo.	$\|\overline{s}_{\text{best}}\|$	$\overline{t}[s]$
4	600	10	**225**	0.04	Bs-H	223	0.21	226	0.68	Bs-H	***227**	2.88
4	600	15	200	0.04	Bs-H	**201**	0.23	204	0.71	Bs-H	***205**	2.24
4	600	20	186	0.05	Bs-H	**188**	0.18	190	0.69	Bs-H	***192**	2.69
4	600	25	**191**	0.06	Bs-H	**191**	0.20	***194**	0.68	Bs-H	***194**	2.20
4	600	40	165	0.04	Bs-H	**167**	0.17	***170**	1.21	Bs-Pow	***170**	2.24
4	600	60	**163**	0.04	Bs-H	162	0.27	***166**	0.69	Bs-H	***166**	2.38
4	600	80	157	0.04	Bs-H	**158**	0.19	159	0.72	Bs-H	***163**	2.70
4	600	100	153	0.07	Bs-H	**156**	0.19	**158**	0.90	Bs-H	**158**	2.31
4	600	150	**154**	0.06	Bs-H	**154**	0.22	**156**	0.66	Bs-H	**156**	2.37
4	600	200	**153**	0.09	Bs-H	152	0.39	***155**	1.22	Bs-H	154	2.63
20	600	10	**75**	0.15	Bs-Pow	74	0.28	***77**	2.38	Bs-Pow	76	2.86
20	600	15	**63**	0.16	Bs-Pow	**63**	0.24	***64**	1.57	Bs-H	***64**	2.91
20	600	20	**59**	0.13	Bs-H	**59**	0.29	**60**	1.58	Bs-H	**60**	2.68
20	600	25	**55**	0.11	Bs-Pow	54	0.20	**55**	1.10	Bs-H	**55**	2.65
20	600	40	**49**	0.08	Bs-H	**49**	0.20	***50**	0.85	Bs-H	***50**	2.85
20	600	60	**47**	0.16	Bs-Pow	46	0.19	47	1.43	Bs-Blum	***48**	3.34
20	600	80	44	0.18	Bs-Blum	**46**	0.30	**46**	1.39	Bs-H	**46**	2.60
20	600	100	**44**	0.14	Bs-H	**44**	0.27	44	2.04	Bs-Blum	***45**	2.33
20	600	150	**45**	0.11	Bs-H	**45**	0.24	**45**	2.94	Bs-Blum	**45**	2.75
20	600	200	**43**	0.17	Bs-H	**43**	0.28	**44**	1.69	Bs-H	43	3.17

Table 3. Results on benchmark set `ES` (averaged over 50 instances per row).

			low-time, literature			*low-time*, Bs-Ex		*high-quality*, literature			*high-quality*, Bs-Ex	
$\|\Sigma\|$	n	m	$\|\overline{s}_{\text{best}}\|$	$\overline{t}_{\text{best}}$	Algo.	$\|\overline{s}_{\text{best}}\|$	$\overline{t}[s]$	$\|\overline{s}_{\text{best}}\|$	$\overline{t}_{\text{best}}$	Algo.	$\|\overline{s}_{\text{best}}\|$	$\overline{t}[s]$
2	1000	10	608.52	0.31	Hh	**609.80**	0.39	614.2	1.42	Bs-Pow	***615.06**	4.43
2	1000	50	533.16	0.33	Hh	**535.02**	0.42	536.46	1.05	Bs-H	***538.24**	4.43
2	1000	100	515.94	0.11	Bs-H	**517.38**	0.46	518.56	1.33	Bs-H	***519.84**	4.82
10	1000	10	199.10	0.53	Hh	**199.38**	0.47	202.72	2.52	Bs-Pow	***203.10**	5.64
10	1000	50	133.86	0.46	Hh	**134.74**	0.35	135.52	2.12	Bs-Pow	***136.32**	3.94
10	1000	100	121.28	0.50	Hh	**122.10**	0.40	122.40	1.50	Bs-H	***123.32**	4.32
25	2500	10	**230.28**	2.33	Hh	223.00	1.57	***235.22**	10.45	Bs-Pow	231.12	19.10
25	2500	50	136.6	1.69	Hh	**137.90**	1.24	138.56	7.23	Bs-Pow	***139.50**	14.51
25	2500	100	120.3	1.74	Hh	**121.74**	1.32	121.62	7.29	Bs-Pow	***122.88**	15.97
100	5000	10	**141.86**	16.12	Hh	139.82	6.98	***144.90**	75.88	Bs-Pow	144.18	91.87
100	5000	50	70.28	9.16	Hh	**71.08**	4.79	71.32	39.11	Bs-Pow	***71.94**	53.54
100	5000	100	59.2	8.71	Hh	**60.04**	4.75	60.06	36.03	Bs-Pow	***60.66**	53.67

Table 4. Results on benchmark `BL` (averaged over 10 instances per row, $|\Sigma| = 4$).

			low-time, literature			*low-time*, Bs-Ex		*high-quality*, literature			*high-quality*, Bs-Ex	
$\|\Sigma\|$	n	m	$\|\overline{s}_{\text{best}}\|$	$\overline{t}_{\text{best}}$	Algo.	$\|\overline{s}_{\text{best}}\|$	$\overline{t}[s]$	$\|\overline{s}_{\text{best}}\|$	$\overline{t}_{\text{best}}$	Algo.	$\|\overline{s}_{\text{best}}\|$	$\overline{t}[s]$
4	100	10	**34.0**	0.01	Bs-Pow	**34.0**	0.02	***34.1**	0.14	Bs-Pow	***34.1**	0.39
4	100	50	23.7	0.03	Hh	**23.8**	0.02	24.1	0.08	Bs-H	***24.2**	0.30
4	100	100	**21.5**	0.03	Hh	**21.5**	0.02	***22.0**	0.23	Bs-Wang	***22.0**	0.27
4	100	150	**20.2**	0.03	Hh	**20.2**	0.02	***20.5**	0.12	Bs-Pow	***20.5**	0.31
4	100	200	**19.8**	0.01	Bs-H	19.5	0.02	***19.9**	0.14	Bs-H	***19.9**	0.31
4	500	10	**182.0**	0.24	Hh	181.2	0.25	***184.1**	1.03	Bs-Pow	184.0	2.41
4	500	50	138.6	0.21	Hh	**139.1**	0.19	140.1	0.90	Bs-Pow	***141.0**	2.13
4	500	100	129.2	0.06	Bs-H	**129.7**	0.18	130.2	1.01	Bs-Pow	***130.8**	2.10
4	500	150	124.7	0.07	Bs-H	**125.5**	0.19	125.9	0.79	Bs-H	***126.4**	2.38
4	500	200	122.6	0.07	Bs-H	**123.0**	0.22	123.2	0.83	Bs-H	***123.7**	2.61
4	1000	10	368.3	0.35	Hh	**368.5**	0.42	373.2	1.80	Bs-Pow	***374.6**	5.22
4	1000	50	284.2	0.36	Hh	**286.2**	0.35	287.0	1.69	Bs-Pow	***288.6**	4.43
4	1000	100	267.5	0.11	Bs-H	**268.8**	0.41	269.5	1.36	Bs-H	***270.6**	4.56
4	1000	150	259.5	0.14	Bs-H	**261.2**	0.47	261.5	1.38	Bs-H	***262.8**	5.30
4	1000	200	254.9	0.17	Bs-H	**256.0**	0.52	256.5	1.81	Bs-H	***257.6**	6.31

Table 5. Results on benchmark set BB (averaged over 10 instances per row).

			low-time, literature			*low-time*, Bs-Ex		*high-quality*, literature			*high-quality*, Bs-Ex	
$\|\Sigma\|$	n	m	$\|\overline{s}_{\text{best}}\|$	$\overline{t}_{\text{best}}$	Algo.	$\|\overline{s}_{\text{best}}\|$	$\overline{t}[s]$	$\|\overline{s}_{\text{best}}\|$	$\overline{t}_{\text{best}}$	Algo.	$\|\overline{s}_{\text{best}}\|$	$\overline{t}[s]$
2	1000	10	**662.9**	0.33	Hh	635.1	0.44	***676.5**	1.16	Bs-H	673.5	5.49
2	1000	100	**551.0**	0.54	Hh	525.1	0.50	***560.7**	2.10	Bs-Pow	536.6	6.05
4	1000	10	**537.8**	0.43	Hh	453.0	0.48	***545.4**	1.73	Bs-H	545.2	6.24
4	1000	100	**371.2**	0.24	Bs-Pow	318.6	0.53	***388.8**	2.86	Bs-Pow	329.5	5.85
8	1000	10	**462.6**	0.27	Bs-Blum	338.8	0.53	***462.7**	7.93	Bs-Blum	***462.7**	7.90
8	1000	100	**260.9**	0.87	Bs-Blum	198.0	0.67	***272.1**	18.43	Bs-Blum	210.6	8.00
24	1000	10	**385.6**	0.67	Bs-Blum	**385.6**	1.04	**385.6**	13.14	Bs-Blum	**385.6**	16.24
24	1000	100	**147.0**	0.66	Bs-Pow	95.8	0.98	***149.5**	8.01	Bs-Pow	113.3	12.45

5 Conclusions and Future Work

This paper presents a beam search (BS) framework for the longest common subsequence (LCS) problem that covers all BS-related approaches that were proposed so-far in the literature. These approaches are currently state-of-the-art for the LCS problem. A second contribution consists of a new heuristic function for BS that is based on approximating the expected length of an LCS, assuming that the input strings are randomly created and independent of each other. Our experimental evaluation showed that our new heuristic guides the search in a much better way than the heuristics and upper bounds from the literature. In particular, we were able to produce new best-known results in 48 out of 67 cases.

On the other side, the experimental evaluation has shown that the new guiding function does not work so well in the context of instances in which the input strings are strongly correlated. This, however, comes with no surprise. In future work we therefore aim at combining our new heuristic with the ones available in the literature in order to exploit the advantages of each of them. For example, it could be considered to evaluate the extensions of the nodes in the current beam by applying several heuristics and keep those that are ranked promising by any of the heuristics. Alternatively, algorithm selection techniques might be applied in order to chose the most promising heuristic based on features of the problem instance. Moreover, note that the general search framework proposed in this paper also naturally extends towards exact search algorithms for the LCS, such as an A* algorithm, for example. Solving small instances to proven optimality and returning proven optimality gaps for larger instances is also our future goal.

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. Moreover, Christian Blum acknowledges the support of LOGISTAR, a proyect from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 769142.

References

1. R. Beal, T. Afrin, A. Farheen, and D. Adjeroh. A new algorithm for "the LCS problem" with application in compressing genome resequencing data. *BMC Genomics*, 17(4):544, 2016.

2. C. Blum and M. J. Blesa. Probabilistic beam search for the longest common subsequence problem. In T. Stützle, M. Birratari, and H. H. Hoos, editors, *Proceedings of SLS 2007 – First International Workshop on Engineering Stochastic Local Search Algorithms*, volume 4638 of *LNCS*, pages 150–161. Springer, 2007.
3. C. Blum, M. J. Blesa, and M. López-Ibáñez. Beam search for the longest common subsequence problem. *Computers & Operations Research*, 36(12):3178–3186, 2009.
4. C. Blum and P. Festa. Longest common subsequence problems. In *Metaheuristics for String Problems in Bioinformatics*, chapter 3, pages 45–60. Wiley, 2016.
5. P. Bonizzoni, G. Della Vedova, and G. Mauri. Experimenting an approximation algorithm for the LCS. *Discrete Applied Mathematics*, 110(1):13–24, 2001.
6. P. Brisk, A. Kaplan, and M. Sarrafzadeh. Area-efficient instruction set synthesis for reconfigurable system-on-chip design. In *Proceedings of the 41st Design Automation Conference*, pages 395–400. IEEE press, 2004.
7. M. Djukanovic, G. Raidl, and C. Blum. Anytime algorithms for the longest common palindromic subsequence problem. Technical Report AC-TR-18-012, TU Wien, Vienna, Austria, 2018.
8. T. Easton and A. Singireddy. A large neighborhood search heuristic for the longest common subsequence problem. *Journal of Heuristics*, 14(3):271–283, 2008.
9. C. B. Fraser. *Subsequences and Supersequences of Strings*. PhD thesis, University of Glasgow, Glasgow, UK, 1995.
10. D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Computer Science and Computational Biology. Cambridge University Press, Cambridge, 1997.
11. K. Huang, C. Yang, and K. Tseng. Fast algorithms for finding the common subsequences of multiple sequences. In *Proceedings of the IEEE International Computer Symposium*, pages 1006–1011. IEEE press, 2004.
12. M. R. Islam, C. M. K. Saifullah, Z. T. Asha, and R. Ahamed. Chemical reaction optimization for solving longest common subsequence problem for multiple string. *Soft Computing*, 2018. In press.
13. T. Jiang, G. Lin, B. Ma, and K. Zhang. A general edit distance between RNA structures. *Journal of Computational Biology*, 9(2):371–388, 2002.
14. J. B. Kruskal. An overview of sequence comparison: Time warps, string edits, and macromolecules. *SIAM review*, 25(2):201–237, 1983.
15. M. López-Ibáñez, J. Dubois-Lacoste, L. Pérez Cáceres, T. Stützle, and M. Birattari. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.
16. D. Maier. The complexity of some problems on subsequences and supersequences. *Journal of the ACM*, 25(2):322–336, 1978.
17. S. R. Mousavi and F. Tabataba. An improved algorithm for the longest common subsequence problem. *Computers & Operations Research*, 39(3):512–520, 2012.
18. S. J. Shyu and C.-Y. Tsai. Finding the longest common subsequence for multiple biological sequences by ant colony optimization. *Computers & Operations Research*, 36(1):73–91, 2009.
19. J. Storer. *Data Compression: Methods and Theory*. Computer Science Press, MD, USA, 1988.
20. F. S. Tabataba and S. R. Mousavi. A hyper-heuristic for the longest common subsequence problem. *Computational Biology and Chemistry*, 36:42–54, 2012.
21. Q. Wang, D. Korkin, and Y. Shang. A fast multiple longest common subsequence (mlcs) algorithm. *IEEE Transactions on Knowledge and Data Engineering*, 23(3):321–334, 2011.