Technical Report AC-TR-2021-008

April 2021/December 2021 (revised)

Graph Search and Variable Neighborhood Search for Finding Constrained Longest Common Subsequences in Artificial and Real Gene Sequences

Marko Djukanovic, Aleksandar Kartelj, Dragan Matić, Milana Grbić, Christian Blum, and Günther R. Raidl

Graph Search and Variable Neighborhood Search for Finding Constrained Longest Common Subsequences in Artificial and Real Gene Sequences

Marko Djukanovic[a,c], Aleksandar Kartelj[b], Dragan Matić[c], Milana Grbić[c], Christian Blum[d], Günther R. Raidl[a]

[a] *Institute of Logic and Computation, TU Wien, Austria*
[b] *University of Belgrade, Faculty of Mathematics, Serbia*
[c] *Faculty of Natural Science and Mathematics, University of Banja Luka, Bosnia and Herzegovina*
[d] *Artificial Intelligence Research Institute (IIIA-CSIC), Campus UAB, Bellaterra, Spain*

Abstract

We consider the constrained longest common subsequence problem with an arbitrary set of input strings as well as an arbitrary set of pattern strings. This problem has applications, for example, in computational biology where it serves as a measure of similarity for sets of molecules with putative structures in common. We contribute in several ways. First, it is formally proven that finding a feasible solution of arbitrary length is, in general, $\mathcal{NP}$-complete. Second, we propose several heuristic approaches: a greedy algorithm, a beam search aiming for feasibility, a variable neighborhood search, and a hybrid of the latter two approaches. An exhaustive experimental study shows the effectivity and differences of the proposed approaches in respect to finding a feasible solution, finding high-quality solutions, and runtime for both, artificial and real–world instance sets. The latter ones are generated from a set of 12681 bacteria 16S rRNA gene sequences and consider 15 primer contigs as pattern strings.

1. Introduction

The longest common subsequence (LCS) problem asks for a common subsequence of maximal length of a set of input strings $S = \{s_1, \ldots, s_m\}$, where each s_i consists of letters from the same finite set Σ. This problem is $\mathcal{NP}$–hard for an arbitrary number $m > 1$ of input strings, as it has been shown by a reduction from the maximum clique problem [27]. The length of the LCS of two or more input strings is a similarity measure widely applied in evolutionary microbiology. The identification of common subsequences of sequences of biological origin (such as DNA, RNA, or protein sequences) is an essential step in sequence alignment and in pattern discovery. The existence of long common subsequences may be a consequence of functional, structural, or evolutionary relationships among the considered inputs [31]. Nowadays, there exist many variants of the original LCS problem. Most of them are obtained by adding further constraints and requirements to the original problem. Many of them were studied intensively over the last two decades. Examples include the longest common palindromic subsequence problem [4], the repetition–free longest common subsequence problem [1], and the arc–preserving longest common subsequence problem [22].

In this work we consider a generalized variant of the constrained longest common subsequence (CLCS) problem [38], which can be stated as follows. In addition to the m input strings from S, this problem requires a so-called pattern string p as input. The goal of the CLCS problem is to find a longest common subsequence of all strings in S which—at the same time—contains pattern p as a subsequence. This problem is also $\mathcal{NP}$–hard as it includes the basic LCS problem as a special case in which the pattern string is an empty string. In practice, it is interesting to also be able to consider more than one pattern string that a solution string has to contain as subsequences. In this paper we therefore focus on the generalized CLCS problem, which is stated as follows. Given a set of $m > 1$ input strings $S = \{s_1, ..., s_m\}$ and a set of $k \geq 1$ pattern strings $P = \{p_1, ..., p_k\}$, the task is to find a string s of maximum length that fulfills the following two conditions:

1. String s is a subsequence of each string $s_i \in S$, and
2. string s contains each $p_j \in P$ as subsequence.

Note that any string s that fulfills these two conditions is henceforth called a *feasible solution*. Moreover, a feasible solution s is called *non-extensible*, if appending any letter $a \in \Sigma$ would turn s into an infeasible solution. In the context of this paper, we use the notation (m, k)–CLCS to refer to this generalized CLCS problem, with sets S and P of arbitrary sizes $m > 1$ and $k \geq 1$, respectively.

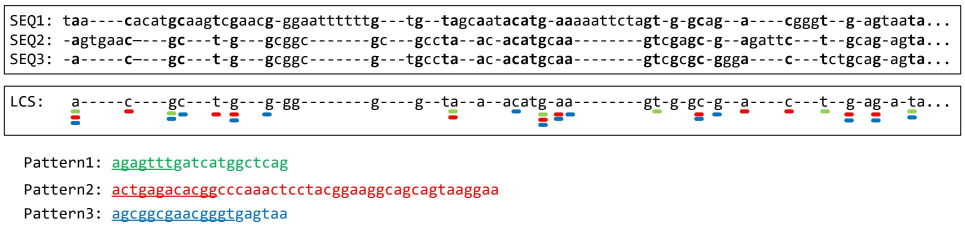
1.1. Example

Figure 1 presents a (partially shown) example for the (m, k)–CLCS problem. The three input strings of this example (SEQ1, SEQ2, and SEQ3) are RNA sequences related to the *Elusimicrobia* bacterial phylum, with lengths 1395, 1424 and 1396, respectively. Moreover, three pattern strings are considered, shown in green, red, and blue. The solution string, as computed by one of our approaches presented later in the paper, is shown as “LCS” in the middle of the figure. The “ -” symbols in the three input sequences and in the solution string are introduced in order to be able to visually align matched letters vertically. In the LCS, letters matched with certain pattern strings are underlined with the respective colors. The input strings of this example belong to a larger class of 165 bacterial rRNA gene sequences, used later for the experimentation in Section 4.7.

From Figure 1 one can already see that there exists a certain similarity between the sequences, which can be expected, since the chosen sequences belong to the same phylum. Although we display only the beginning of each sequence, it can be observed that many characters from the three patterns are already matched. In later parts of the input sequences one can find even significantly larger similarities. To illustrate this, Figure 2 shows input sequence segments and the corresponding part of the LCS that start at positions 815, 828, and 830, respectively.

1.2. Related Work

Concerning the basic LCS problem, many approaches—especially heuristics—were proposed in the last decades. However, we first review the existing exact approaches. First, note that—for a fixed value of m the LCS problem is solvable in polynomial time by dynamic programming (DP) [17]. The basic version of this algorithm runs in $O(n^m)$ time, where n is the length of the longest input string. However, with growing n and/or m, DP becomes quickly unpractical. Concerning parallel exact approaches, Liu et al. [25] proposed the parallel FAST_LCS search algorithm which is based on the use of a special data structure called successors table. Pruning operations are utilized to reduce the computational effort. Wang et

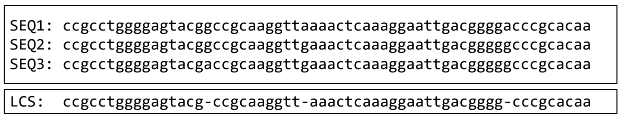


```
SEQ1: taa----cacatgcaagtcgaacg-ggaatttttg---tg--tagcaatacatg-aaaaattctagt-g-gcag--a----cgggt--g-agtaata...
SEQ2: -agtgaac—---gc---t-g---gcggc-------gc---gccta--ac-acatgcaa-------gtcgagc-g--agattc---t--gcag-agta...
SEQ3: -a-----c—---gc---t-g---gcggc-------g---tgccta--ac-acatgcaa-------gtcgcgc-ggga----c---tctgcag-agta...

LCS:   a-----c----gc---t-g---g-gg-------g----g--ta--a--acatg-aa-------gt-g-gc-g--a----c---t--g-ag-a-ta...

  Pattern1: agagtttgatcatggctcag
  Pattern2: actgagacacggcccaaactcctacggaaggcagcagtaaggaa
  Pattern3: agcggcgaacgggtgagtaa
```

Figure 1: Example for the (m, k)–CLCS problem with three rRNA sequences and three pattern strings. Only the starting segments of the input strings are shown.



```
SEQ1: ccgcctggggagtacggccgcaaggttaaaactcaaaggaattgacggggacccgcacaa
SEQ2: ccgcctggggagtacggccgcaaggttgaaactcaaaggaattgacgggggcccgcacaa
SEQ3: ccgcctggggagtacgaccgcaaggttgaaactcaaaggaattgacgggggcccgcacaa

LCS:  ccgcctggggagtacg-ccgcaaggtt-aaactcaaaggaattgacgggg-cccgcacaa
```

Figure 2: Another segment of the same problem instance and the corresponding part of the solution.

al. [39] proposed another parallel algorithm called QUICK-DP based on the dominant point approach employing a fast divide-and-conquer technique in order to compute the dominant points. More recently, the so-called TOP_MLCS algorithm was suggested by Li et al. [23], based on a directed acyclic layered-graph model. The latest exact approach, by Djukanovic et al. [11], is an A* search. It was shown that this algorithm performs best in terms of memory consumption, running time, and the number of benchmark instances solved to optimality. Nevertheless, exact approaches can generally only solve small LCS instances up to $m = 10$ and $n = 100$, and their main issue is typically an excessive memory consumption.

In the context of larger LCS problem instances, researchers have therefore focused on the development of heuristic approaches. The Best-Next heuristic [21], for example, is a well known simple and fast construction heuristics. Among various different metaheuristic approaches, the beam search variant described by Djukanovic et al. [9] has been shown to be the current state-of-the-art. This algorithm uses a special filtering method and a sophisticated search guidance based on an approximation of the expected length of an optimal solution. Concerinig the size of the largest LCS instances used in the literature which is successfully tackled by the state-of-the-art methaheuristic is the one with $m = 100$ and $n = 5000$.

Concerning the CLCS problem, the version on two input strings and one pattern string ($(2, 1)$–CLCS) was intensively studied over the last decade. Various efficient exact approaches were proposed [38, 6, 5]. Recently, the $(2, 1)$–CLCS problem was tackled with an A* search [8] which can be considered the new state-of-the-art exact method for various real and artificial benchmark sets. The A* could solve the $(2, 1)$–CLCS instances up to $n = 1000$. In [7], this A* search was adapted to the more general $(m, 1)$–CLCS case with $m \in \mathbb{N}$, but as one may expect, this approach only scales well to small and medium sized instances up to $m = 10$ and $n = 100$ with small alphabet size $|\Sigma|$ and some larger instances with up to $m = 10$ and $n = 1000$ when the pattern string is long w.r.t. n. Therefore, the authors further describe a greedy heuristic and a beam search approach to tackle large instances heuristically. The beam search is guided by an approximate expected length calculation in the spirit of [9] and was shown to be highly efficient in the case of rather short pattern strings. For longer pattern strings, a beam search with a probability-based guidance mechanism was observed to perform significantly better. This beam search technique was able to sucessfully tackle the instances with up to $m = 100$ and $n = 1000$ regardless of the sizes of $|\Sigma|$ and the input pattern.

Concerning the generalized (m, k)–CLCS problem, we are only aware of the exact algorithm based on an Automaton approach from Farhana and Sohel [13]. Unfortunately, this approach is quite limited for a practical use. In general, it is only efficient when either the input sequences are short or the number of input sequences is low. Otherwise, the amount of necessary memory to store all the states of the intersection automaton is prohibitive. Other known results for the (m, k)–CLCS problem include a proof that no approximation algorithm can exist for this problem when k may be arbitrarily large; see [15]. The $(2, k)$-CLCS problem with an arbitrary number of pattern strings is $\mathcal{NP}$-hard, which can be proven by means of a reduction from 3-SAT. This already implies that the (m, k)–CLCS problem, as a generalization of the $(2, k)$–CLCS problem, is also $\mathcal{NP}$-hard. Finally, we want to point out that the (m, k)–CLCS problem also is closely related to the shortest common supersequence problem [24] and its variant which includes constraints [12, 28].

1.3. Our Contributions

First of all, we provide a formal proof that already finding any feasible solution to the (m, k)–CLCS problem is $\mathcal{NP}$-complete. As already finding any feasible solution can be challenging in practice, we put separate effort to this task by proposing dedicated algorithms with the aim of finding a feasible solution rather that maximizing the solution length. More specifically, we present a greedy method and two different versions of beam search (BS). The way in which these approaches extend partial solutions aims at deriving at least one feasible solution. In contrast to our earlier work on a BS proposed for solving the $(m, 1)$–CLCS problem [8], the two new variants of BS work on the basis of a different search framework. Moreover, their search is guided by novel search guidance functions that are developed as significant extensions of those already known for the $(m, 1)$–CLCS problem. In addition, a restricted version of BS makes use of cutting off those extensions which have a lower chance to lead towards feasible solutions. We emphasize that feasibility was not an issue in our earlier work [8].

We then propose a variable neighborhood search (VNS) that utilizes two different neighborhood functions for local search. Moreover, we combine the previously developed BS with VNS by using the solution from the BS as initial solution of the VNS. In order to establish an efficient VNS, a partial calculation of the fitness function is employed by means of non-trivial data structures. As we will show in the experimental evaluation, VNS is generally able to significantly improve over those initial solution. The practical effectivity of our approaches is analyzed on a wide range of artificially generated benchmark instances.

In practical applications, the possibility to consider more than one pattern string in the CLCS problem seems to be particularly useful. For example, Tang et al. [37] consider the task of aligning RNase sequences, requiring that each of the three active-site residues His(H), Lyn(K), and His(H) is present in the resulting alignment. More generally, it is reasonable to assume that the requirement of the appearance of a larger number of patterns in an LCS solution can help biologists to gain meaningful insights in the context of particular biological structures. Therefore, in addition to the experiments on artificial instances as mentioned above, we perform a case study and apply the proposed algorithms on a set of real bacterial RNA sequences and a specific set of patterns based on *contig primer structures* [29]. As it will be shown in Section 4.7, obtained results indicate a high similarity between the considered sequences, even if many patterns and many sequences are considered. A biological interpretation of the obtained results is provided at the end of the experimental section.

The rest of the paper is organized as follows. In Section 1.2 an overview of the existing literature is given for some variants of the considered problem as well as closely related problems. In Section 3, three heuristic approaches are proposed: a greedy heuristic, a beam search, and a variable neighborhood search. In Section 4 we present an exhaustive experimental evaluation of the proposed methods. Section 5 draws conclusions and outlines ideas for future work.

2. Hardness of Finding a Feasible Solution

As the (m, k)–CLCS problem is a generalization of the classical LCS problem with an arbitrary number of strings, the (m, k)–CLCS problem is also NP-hard. Moreover, even finding any feasible solution is $\mathcal{NP}$–complete, what is shown in the following theorem.

Theorem 1. *The decision problem of whether or not a given (m, k)–CLCS problem instance has a feasible solution is $\mathcal{NP}$–complete for arbitrary m, $k \geq 2$, and $|\Sigma| \geq 2$.*

Proof. Our proof builds upon results from Blin et al. [2] for the classical LCS problem with arbitrary many strings. We utilize a reduction from the decision variant of the independent set problem asking if a simple undirected graph $\bar{G} = (\bar{V}, \bar{E})$ with node set $\bar{V} = \{1, \ldots, \bar{n}\}$ and edge set $\bar{E} = \{e_1, \ldots, e_{\bar{m}}\} \subseteq \bar{V}^2$ has a subset of exactly $\bar{k}$ nodes $\bar{I} \subseteq \bar{V}$ so that no pair of these nodes is connected by an edge. An instance of this independent set problem is mapped to an (m, k)–CLCS problem instance on the binary alphabet $\Sigma = \{0, 1\}$ as follows. Let us denote repeated occurrences of some letter or substring s by the power function s^i, e.g., $0^3 = 000$ or $(01)^2 = 0101$. Set S includes the single string

$$s_{\bar{m}+1} = (0^{\bar{n}}1)^{\bar{n}} \tag{1}$$

of length $\bar{n}^2 + \bar{n}$ and for each edge $e_i = \{u, v\} \in \bar{E}$, $u < v$, the string

$$s_i = (0^{\bar{n}}1)^{u-1}0^{\bar{n}}(0^{\bar{n}}1)^{v-u}0^{\bar{n}}(0^{\bar{n}}1)^{\bar{n}-v} \tag{2}$$

of length $\bar{n}^2 + 2\bar{n} - 2$. The pattern string set P includes string $0^{\bar{n}}$ and string $1^{\bar{k}}$ and thus enforces a solution to contain at least $\bar{n}$ zeros and $\bar{k}$ ones. Clearly, this mapping from an independent set instance to a corresponding (m, k)–CLCS problem instance can be done in polynomial time.

Now, we rely on the fact that an independent set of size $\bar{k}$ exists for graph $\bar{G}$ if and only if the strings in S have a subsequence of length $\bar{n}^2 + \bar{k}$, which was shown by Blin et al. [2] in their Proposition 1. More specifically, if such a subsequence of length $\bar{n}^2 + \bar{k}$ exists, it has to contain $\bar{n}^2$ zeros and the positions of the ones indicate a selection of nodes corresponding to an independent set of size $\bar{k}$, as also argued in [2]. In case of our (m, k)–CLCS problem, we have the additional two pattern strings P that enforce the appearance of at least $\bar{n}^2$ zeros and $\bar{k}$ ones in a feasible solution. Therefore, an independent set of size $\bar{k}$ exists for graph $\bar{G}$ if and only if the corresponding (m, k)–CLCS problem instance has a feasible solution. As the transformation can be done in polynomial time and the solution can be checked in polynomial time, this concludes our proof. □

3. Algorithmic Approaches

Let us first introduce notations commonly used throughout the paper. Let $n_{\max} := \max_{i=1,\ldots,m} |s_i|$ and $n_{\min} := \min_{i=1,\ldots,m} |s_i|$. For a string s and $1 \leq i \leq j \leq |s|$, let $s[x, y] = s[x] \cdots s[y]$ be the continuous (sub)string which starts at index x and ends at index y. If $x > y$, $s[x, y]$ is the empty string ε. Note that the first index of a string s is one. By $|s|_a$ we denote the number of occurrences of letter $a \in \Sigma$ in string s. By $\vec{\theta} = (\theta_1, \ldots, \theta_m)$, $1 \leq \theta_i \leq |s_i| + 1$ for all $i = 1, \ldots, m$, we denote a so-called *position vector* with respect to the input strings in S; each θ_i refers to either a specific position in string s_i or at (one position after) the end of the string. Given a position vector $\vec{\theta}$, $S[\vec{\theta}] := \{s_i[\theta_i, |s_i|] \mid i = 1, \ldots, m\}$ refers to the respective suffix strings from the indicated positions onward. Similarly, $\vec{\lambda} = (\lambda_1, \ldots, \lambda_k)$, $1 \leq \lambda_j \leq |p_j| + 1$ for $j = 1, \ldots, k$, denotes a *position vector* on the set of pattern strings P, and $P[\vec{\lambda}] := \{p_j[\lambda_j, |p_j|] \mid j = 1, \ldots, k\}$ refers to the suffix pattern strings from the positions indicated by $\vec{\lambda}$ onward.

Moreover, we make extensive use of the following table constructed during preprocessing. For each $i = 1, \ldots, m$, $l = 1, \ldots, |s_i|$, and $c \in \Sigma$, $\texttt{Succ}[i, x, c]$ holds the minimal index y such that (i) $x \geq y$ and (ii) $s_i[y] = c$, i.e., the position of the next occurrence of letter c in string s_i from position x onward. If letter c does not appear in s_i starting from position l, we set $\texttt{Succ}[i, x, c] := -1$. This data structure is built in $O(m \cdot n \cdot |\Sigma|)$ time.

Last but not least, table $\texttt{Embed}[i, x, j]$ for all $i = 1, \ldots, m$, $j = 1, \ldots, k$, and $x = 1, \ldots, |p_j| + 1$ stores the right-most (largest) position y of s_i such that $p_j[x, |p_j|]$ is a subsequence of $s_i[y, |s_i|]$. Note that when $x = |p_j| + 1$ it follows that $p_j[x, |p_j|] = \varepsilon$. In this case $\texttt{Embed}[i, x, j]$ is set to $|s_i| + 1$. In contrast to Succ, Embed does not need to be pre-computed on for the Greedy heuristic explained within the next section, where the needed values are only derived on demand and stored in a hash table.

3.1. Greedy Heuristic

In the following we develop a greedy heuristic in the style of the well-known Best–Next heuristic [21] for the LCS problem. This algorithm starts with an empty partial solution $s = \varepsilon$. At each construction step, exactly one letter is appended to s. There are two possibilities for algorithm's outcome: (i) s which is a feasible non-extensible solution or (ii) an empty solution ε since it is detected that s cannot be extended into a feasible non-extensible solution. At the start of the algorithm, the position vector $\vec{\theta}$ is initialized to the all-ones vector $\vec{1}$ of length m, while the cover position vector $\vec{\lambda}$ is initialized to the all-ones vector $\vec{1}$ of length k.

Then, at each construction step, the following is done. First, the set of feasible extensions $\Sigma_s \subseteq \Sigma$ of the current partial solution s is determined. Hereby, a letter c is part of Σ_s, if and only if the following two conditions are fulfilled:

- Condition 1: Letter c appears at least once in each of the prefix strings $s_i[\theta_i, |s_i|]$, $i = 1, \ldots, m$.
- Condition 2: After appending c to s, the resulting partial solution can still be feasibly extended. In this context, let $I_c := \{j \in \{1, \ldots, k\} \mid \lambda_j \leq |p_j| \text{ and } p_j[\lambda_j] = c\}$ and $\overline{I_c} := \{1, \ldots, k\} \setminus I_c$. This condition holds if, for all $i = 1, \ldots, m$, the following is true:
  - For all $j \in I_c$ it holds that $\theta_i + \texttt{Succ}[i, \theta_i, c] + 1 \leq \texttt{Embed}[i, \lambda_j + 1, c]$
  - For all $j \in \overline{I_c}$ it holds that $\theta_i + \texttt{Succ}[i, \theta_i, c] + 1 \leq \texttt{Embed}[i, \lambda_j, c]$

Moreover, we further reduce set Σ_s by removing dominated letters. In this context, we say that letter a *dominates* b if and only if $Succ[i, \theta_i, a] \leq Succ[i, \theta_i, b]$, for all $i = 1, ..., m$. The subset of Σ_s consisting only of non-dominated letters is henceforth denoted by Σ_s^{nd}. Finally, note that this greedy heuristic is developed with the primary aim of providing a feasible solution. One of the measures to support this aim is the use of `Embed`. The second measure is to further reduce set Σ_s^{nd} to only those letters $c \in \Sigma_s^{\text{nd}}$ such that $p_j[\lambda_j] = c$ for at least one $j \in \{1, \ldots, k\}$. This resulting set is henceforth called $\Sigma_s^{\text{nd,str}}$, where "str" stands for *strong*. Given a partial solution s, if $\Sigma_s^{\text{nd,str}} \neq \emptyset$, a letter is chosen from $\Sigma_s^{\text{nd,str}}$ that minimizes the greedy function

$$g(\vec{\theta}, c) = \sum_{i=1}^{m} \frac{\texttt{Succ}[i, \theta_i, c] - \theta_i + 1}{|s_i| - \theta_i + 1}. \tag{3}$$

Otherwise, with the same greedy function a letter is chosen from Σ_s^{nd}. In case $\Sigma_s^{\text{nd}} = \emptyset$, the solution construction stops either because s is a feasible non-extensible solution, or because s cannot be extended into a feasible non-extensible solution. Note that the greedy function (3) is well known from the above mentioned Best–Next heuristic for the standard LCS problem.

Let c^* be the letter chosen at the current construction step. After appending c^* to s—that is, $s \leftarrow s \cdot c^*$—the following updates are performed:

- $\theta_i \leftarrow \theta_i + \texttt{Succ}[i, \theta_i, c^*] + 1$
- If $p_j[\lambda_j] = c^*$ then $\lambda_j \leftarrow \lambda_j + 1$

Finally, note that $O(k \cdot m \cdot n)$ time is required to check if a letter satisfies Condition 2 above, that is, the condition that checks if a certain extension of the current partial solution leads to a (partial) solution that can be further extended in a feasible way. This is the cost for deriving the needed entries of table `Embed`. We remark that when solving the real-world instances used in our experimental studies, this runtime approximates just $O(mn)$ due to the instance characteristics. However, remember that the proposed greedy heuristic cannot guarantee the construction of a feasible solution. Consider, for example, problem instance $S = \{\texttt{abbba}, \texttt{babb}\}$, $P = \{\texttt{bb}, \texttt{a}\}$, and $\Sigma = \{\texttt{a}, \texttt{b}\}$. During the first construction step of the greedy heuristic letters `a` and `b` are the candidates to extend the initially empty solution. Their greedy heuristic values are equal. However, choosing `b` leads to an infeasible solution, as a second and third `b` would be chosen according to our greedy function and finally pattern string `a` cannot be (feasibly) extended anymore.

A pseudocode of our greedy heuristic is given in Algorithm 1 in the document on supplementary material.

3.2. Defining a State Graph

For the development of more advanced graph search techniques, we define in the following a state graph for the (m, k)–CLCS problem in the form of a rooted, directed, acyclic graph. Note that this definition will be based on a simplified way for extending partial solution as done in Greedy. In particular, we will make use of the values in the pre-computed `Embed` structure. As a consequence, Condition 2 of the above definition of the set of letters that can be used to extend a partial solution is here directly checked with help of the pre-processed `Embed` structure.

The principal component of our state graph are nodes of the form $v = (\vec{\theta}^v, \vec{\lambda}^v, l^v)$, where $\vec{\theta}^v$ is a position vector on the input strings, $\vec{\lambda}^v$ a position vector on the pattern strings, and l^v

is the length of a partial solution represented by node v. Note that such a node may represent several partial solutions. As it is, in general, infeasible to produce the whole state graph before running an algorithm, the state graph is partially discovered and searched on the fly. That is, a partial solution s induces a node $v = (\vec{\theta}^v, \vec{\lambda}^v, l^v)$ in the following way.

- Position vector $\vec{\theta}^v$ is defined such that $s_i[1, \theta_i^v - 1]$ is the shortest possible prefix string of s_i of which s is a subsequence.
- Position vector $\vec{\lambda}^v$ is defined such that $p_j[1, \lambda_j^v - 1]$ is the longest possible prefix string of p_j which is a subsequence of s.
- Length $l^v \leftarrow |s|$

The root node of the state graph is $r = ((1, \ldots, 1), (1, \ldots, 1), 0)$, induced by the empty partial solution ε. In the following, by Σ_v^{nd} we denote the set of non-dominated letters that can be used to extend any solution represented by a node v. The way of deriving this set is, as mentioned above, exactly the same as the one of deriving set Σ_s^{nd} in the context of Greedy. For each letter $c \in \Sigma_v^{\text{nd}}$, a successor node w of v is generated as follows:

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

Moreover, a directed arc vw is added from node v to node w and is labeled with the letter c, i.e., $\ell(vv') = c$. Note that no weights are associated with our arcs as each arc corresponds to an extension of a partial solution by exactly one letter.

A node v is called non-extensible if $\Sigma_v^{\text{nd}} = \emptyset$. Moreover, it is called *feasible* iff $\lambda_j^v = |p_i| + 1$, for all $j = 1, \ldots, k$, i.e., all pattern strings are covered. A longest path (in terms of the number of arcs) from the root node to a feasible non-extensible node represents an optimal solution of the (m, k)–CLCS instance. An example of the state graph can be seen in Figure 3[1].

In the next two sections, a few graph search algorithms based on the introduced state graph are described.

3.3. Beam Search for the (m, k)–CLCS Problem

Beam search (BS) is a well-known breath-first-search (BFS) heuristic [33]. On the basis of the state graph introduced in the previous section, we devised the following BS variant for the (m, k)–CLCS problem; see also Algorithm 1. Apart from a problem instance (S, P, Σ), the algorithm takes as input a so-called *beam width* β, a filtering parameter $k_{\text{best}} \geq 0$, and a boolean control variable `restricted` $\in \{true, false\}$, which lets the user choose between a restricted version of BS that aims at obtaining any feasible solution with higher chances (when `restricted` is set to *true*) and the standard BS targeted toward the overall optimization problem. The former BS version is henceforth called restricted-BS and the latter basic-BS.

[1]Note that it is not necessary to store actual partial solutions in the nodes. To any node v a path of length l^v from the root node can be efficiently derived in a backward manner by iteratively following a predecessor for which the corresponding l^v-value decreases by one.

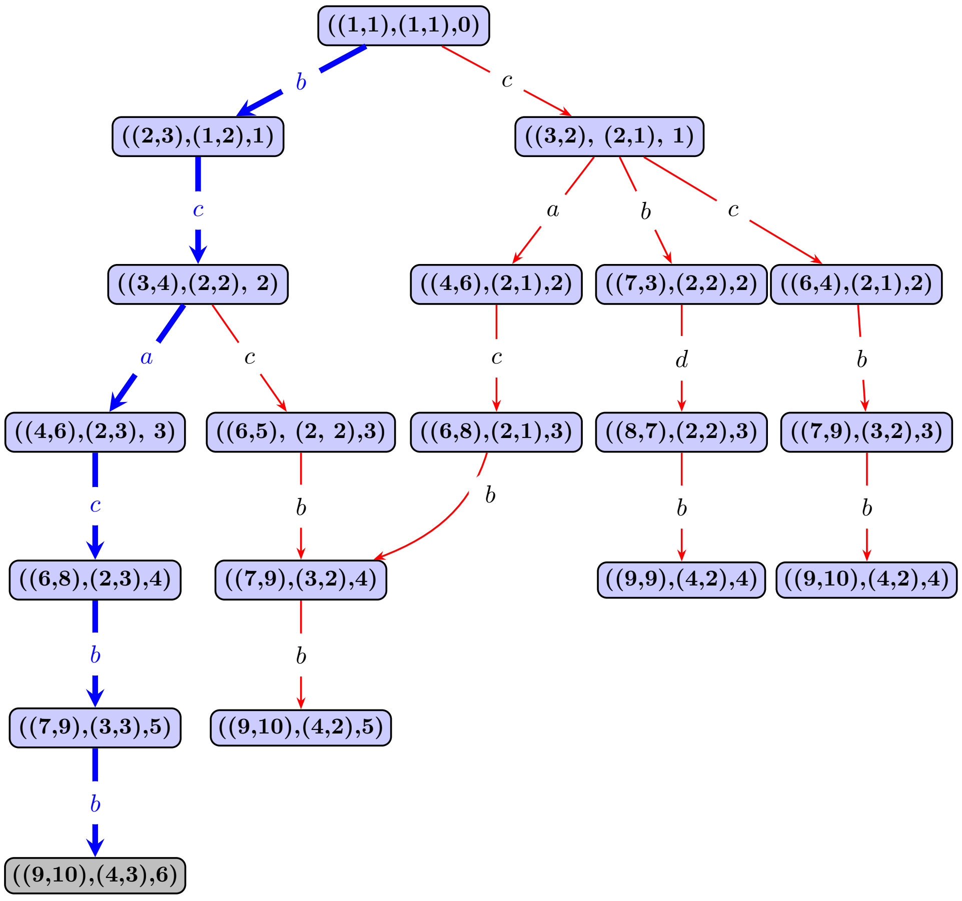


Figure 3: Example showing the full state graph in form of a directed acyclic graph for problem instance ($S = \{s_1 = \texttt{bcaacbdba}, s_2 = \texttt{cbccadcbbd}\}$, $P = \{\texttt{cbb}, \texttt{ba}\}$, $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$). It contains four non-extensible solutions. However, only one of them (marked by light-gray color) corresponds to a feasible solution $s = \texttt{bcacbb}$, which is therefore also the optimal solution. The solution is represented by node $v = (\vec{\theta}^v = (9, 10),\ \vec{\lambda}^v = (4, 3),\ l^v = 6)$. The corresponding path in the search space is displayed in blue.

Both versions start by initializing the best-found solution s_{bsf} to the empty string, that is, $s_{\text{bsf}} \leftarrow \varepsilon$. Afterwards, the beam B, which contains the currently selected nodes, is initialized to the root node of the state graph (see lines 3 and 4 of the pseudo-code). Then, while B is not empty, the following is done at each algorithm iteration. For each $v \in B$, first, the set of non-dominated letters (Σ_v^{nd}) that can be used to extend solutions represented by v is determined. Note that we henceforth call a letter $c \in \Sigma_v^{\text{nd}}$ that also belongs to $\Sigma_v^{\text{nd,str}} \subseteq \Sigma_v^{\text{nd}}$ a *strong extension* of v. If Σ_v^{nd} is empty, the solution s that is represented by v is derived and, if s is a feasible solution, the best-found solution s_{bsf} is updated with s, if appropriate. Otherwise—that is, if $\Sigma_v^{\text{nd}} \neq \emptyset$ —a node v' is generated that corresponds to the extension of v by c. Note that, in case the algorithm is run in the restricted mode with `restricted` $=$ *true*, this is only done if letter a is a strong extension. Node v' is then added to a set V_{ext} which collects all the extensions of the current beam B. Finally, as V_{ext} might contain dominated nodes, because domination was so-far only checked w.r.t. extensions of the same node v, dominated nodes are removed in function `RemoveDominatedNodes`$(V_{\text{ext}}, k_{\text{best}})$. However, in order to decrease the computational burden of this operation, domination of nodes is only checked w.r.t. the best k_{best} nodes from V_{ext} w.r.t. their so-called h-values. Candidates of for the heuristic function h are discussed in the next two subsections. Finally, the best β nodes with respect to their h-values are selected in function Reduce$(V_{\text{ext}}, \beta, h)$ and form the beam B of the next iteration.

Algorithm 1 Beam Search (BS) for the (m, k)–CLCS problem

```
1: **Input:** an instance $(S, P, \Sigma)$, beam width $\beta$, heuristic function $h$, filtering parameter
   $k_{\text{best}} \geq 0$, restricted $\in \{true, false\}$
2: $s_{\text{bsf}} \leftarrow \varepsilon$
3: Create root node $r \leftarrow ((1, \ldots, 1), (1, \ldots, 1), 0)$
4: $B \leftarrow \{r\}$ ($B$ is called the *beam*)
5: **while** $B$ is not empty **do**
6:     $V_{\text{ext}} \leftarrow \emptyset$
7:     **for** each $v \in B$ **do**
8:         Determine $\Sigma_v^{\text{nd}}$
9:         **if** $\Sigma_v^{\text{nd}} = \emptyset$ **then**
10:            Derive the solution $s$ represented by $v$
11:            **if** $s$ is a feasible solution **and** $|s| > |s_{\text{bsf}}|$ **then**
12:                $s_{\text{bsf}} \leftarrow s$
13:            **end if**
14:        **else**
15:            **for** $c \in \Sigma_v^{\text{nd}}$ **do**
16:                **if** not restricted **or** (restricted **and** letter $c$ is a *strong extension*) **then**
17:                    Derive node $v'$ from node $v$ by extension with letter $c$
18:                    $V_{\text{ext}} \leftarrow V_{\text{ext}} \cup \{v'\}$
19:                **end if**
20:            **end for**
21:        **end if**
22:    **end for**
23:    $V_{\text{ext}} \leftarrow$ RemoveDominatedNodes$(V_{\text{ext}}, k_{\text{best}})$
24:    $B \leftarrow$ Reduce$(V_{\text{ext}}, \beta, h)$
25: **end while**
26: **Output:** best solution found $s_{\text{bsf}}$ (may be $\varepsilon$, which means: no feasible solution found)
```

3.3.1. Upper Bounds

The most common way for reducing set V_{ext} in function `Reduce`$(V_{\text{ext}}, \beta, h)$ of the BS algorithm is to use an upper bound calculation as h-value of a node $v \in V_{\text{ext}}$. Function Reduce$(V_{\text{ext}}, \beta, h)$ then chooses those β nodes from V_{ext} that have the highest h-values. For implementing this option, we consider a combination of the most successful upper bounds that were developed for the standard LCS problem in the literature. The upper bound UB_1 is derived by Blum et al. [3].

The second upper bound $\text{UB}_2()$ is based on dynamic programming (DP) and was introduced by Wang et al. [39].

As, in general, neither of the two bounds UB_1 and UB_2 dominates the other one, we

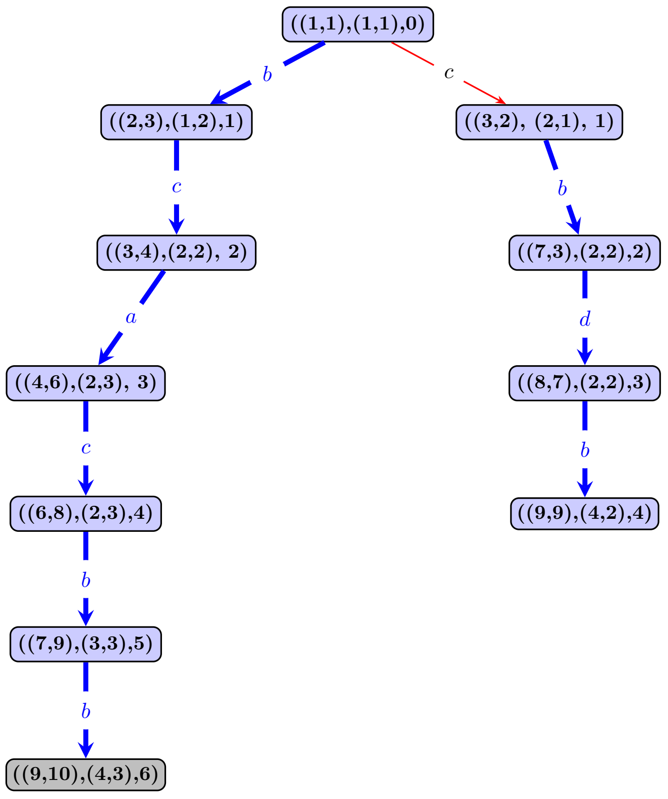


Figure 4: Example showing the restricted state graph for the problem instance ($S = \{s_1 = \texttt{bcaacbdba}, s_2 = \texttt{cbccadcbbd}\}, P = \{\texttt{cbb}, \texttt{ba}\}, \Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$). Note that in this case it becomes much easier to find a feasible solution (which is here also equal to the optimum).

consider the combination of both, i.e.,

$$\mathrm{UB}(v) = \min\{\mathrm{UB}_1(v), \mathrm{UB}_2(v)\}. \tag{4}$$

More details about these two upper bounds can be found in Section 2 of the document on supplementary material.

3.3.2. Heuristic Guidance

In this section we propose a probability–based heuristic guidance function as an alternative to the upper bound described above. Note that this function is an extension of the one developed for the $(m, 1)$–CLCS problem in [7].

In particular, we make use of a DP recursion from [32] for calculating the probability $\Pr(r, q)$ that any string of length r is a subsequence of a *random string* of length q. These probabilities are computed in a preprocessing step for $r, q = 0, \ldots, n$. By assuming independence among the input strings, the probability $\Pr(s \prec S)$ that a random string s of length r is a common subsequence of all input strings is

$$\Pr(s \prec S) = \prod_{i=1}^{m} \Pr(r, |s_i|). \tag{5}$$

In order to make use of this probability in the case of the (m, k)–CLCS problem, we make the simplifying assumption that each such string s is extensible towards a feasible, non-extensible

(m,k)–CLCS solution (that is, s has at least one feasible completion). Given V_{ext} in some construction step of BS, the question is how to choose the value of r common to all nodes $v \in V_{\text{ext}}$ in order to take profit from the above formula in a reasonable way. For this purpose, we first calculate

$$p_j^{\min} = \min_{v \in V_{\text{ext}}} \left(|p_j| - \lambda_j^v + 1\right), j = 1, \ldots, k. \tag{6}$$

Due to our assumption suffixes $p_j[p_j^{\min}, |p_j|]$, $j = 1, \ldots, k$, must appear as subsequences in at least one possible completion of any of the nodes from $v \in V_{\text{ext}}$. In practice, it may happen that any number of extensions in the range from $\max_{j=1,\ldots,k}\{p_j^{\min}\}$ to $\sum_{j=1}^{k} p_j^{\min}$ may be required in order to cover suffixes $p_j[p_j^{\min}, |p_j|]$, $j = 1, \ldots, k$, and which are counted for safe common extensions (i.e., the extensions which must occur somewhere in the search) of the partial solution of any node $v \in V_{\text{ext}}$. Since in practice this number tends towards $p^{\min} = \sum_{j=1}^{k} p_j^{\min}$, we heuristically choose the r-value as

$$r = p^{\min} + \min_{v \in V_{\text{ext}}} \left\lfloor \frac{\min_{i=1,\ldots,m}\{|s_i| - \theta_i^v + 1\} - p^{\min}}{|\Sigma|} \right\rfloor. \tag{7}$$

The intention here is, first, to let the characters from $p_j[p_j^{\min}, |p_j|]$, $j = 1, \ldots, k$, fully count, because they will have to appear for sure in any possible extension due to our assumption. This explains the first term ($p^{\min}$) in Eq. (7). The second term is motivated by the fact that an optimal (m,k)–CLCS solution becomes shorter if the alphabet size becomes larger. Moreover, the solution tends to be longer for nodes v whose length of the shortest remaining string from $S[\vec{\theta}^v]$ is longer than the one of other nodes. We emphasize that this is a heuristic choice which might be improvable. If this calculation for r results in zero, it is set to one for breaking ties. The final probability-based heuristic for evaluating a node $v \in V_{\text{ext}}$ is

$$H(v) = \prod_{i=1}^{m} \Pr(r, |s_i| - \theta_i^v + 1), \tag{8}$$

and those nodes with a larger H-value are preferred.

3.4. Variable Neighborhood Search

Variable neighborhood search (VNS) is a well known metaheuristic that was proposed by Mladenović and Hansen [30]. Its main idea is the systematic change of neighborhood structures in order to escape from local optima. In the past two decades, VNS has been proven to be a very effective metaheuristic, by means of numerous applications to complex optimization problems (see for example [18, 19] and references therein). However, before outlining our VNS method, we first introduce a fine-grained fitness function that evaluates both feasible and infeasible solutions. This is important because our VNS is not restricted to work in the feasible part of the search space. The VNS-related fitness $F(s)$ of a solution s is calculated as

$$F(s) := \begin{cases} \sum\limits_{s_i \in S} (|s| - |\text{LCS}(s, s_i)|) + \sum\limits_{p_j \in P} (|p_j| - |\text{LCS}(p_j, s)|) & \text{if } s \text{ is infeasible,} \\ \dfrac{n_{\min} - |s|}{n_{\min} + 1} & \text{if } s \text{ is feasible.} \end{cases} \tag{9}$$

Hereby, $\text{LCS}(s, s')$ refers to the longest common subsequence between strings s and s', computed by dynamic programming. Note that for any feasible solution s it holds (a) that

$|s| - |\mathrm{LCS}(s, s_i)| = 0$ for all $i = 1, \ldots, m$ and (b) that $|p_j| - |\mathrm{LCS}(p_j, s)| = 0$, for all $j = 1, \ldots, k$. This is because a feasible solution s must be a subsequence of each input string s_i, and each pattern string p_j must be a subsequence of s. Moreover, for any infeasible solution s it holds that some of these terms are greater than zero. In fact, we could say that—in such a case—$F(s)$ counts the number of feasibility violations on a per character basis. Finally, if a solution s is feasible, $\dfrac{n_{\min} - |s|}{n_{\min} + 1}$ evaluates to a value smaller than one. Therefore, in case a solution s is infeasible, it holds that $F(s) \geq 1$, and $F(s) < 1$ otherwise. Moreover, for any two feasible solution s and s' with $|s| > |s'|$ it holds that $F(s) < F(s')$. The VNS algorithm therefore aims to find a solution s that minimizes function F. Until a feasible solution is found, the algorithm will not aim to add new characters just for the sake of increasing the solution length. Instead, it will be focused on reaching feasibility as soon as possible, either by adding or by removing characters. Once feasibility is reached, the fitness function will drive the algorithm to increase the solution length.

In the literature one can find different VNS variants. The variant adopted in this paper is shown in Algorithm 2 and works as follows. It starts with an initial solution s_{init} which is provided as input to the algorithm; note that s_{init} might be the empty string ε. Then, at each iteration, first a so-called shaking procedure is applied to a copy (s) of the best-so-far solution s_{bsf}. If s is feasible $(F(s) < 1)$, procedure $\mathsf{Shaking\_Delete}(s, \kappa)$ is applied; see line 11. Otherwise, the alternative shaking procedure $\mathsf{Shaking\_Change}(s, \kappa)$ is used (line 13). The strength of the respective shaking procedure—which is determined by a parameter κ with a lower limit $\kappa_{\min}$ and an upper limit $\kappa_{\max}$—depends hereby on the success of the previous iteration. In general, the diversification of the search process increases with an increasing shaking strength. After shaking, local search is applied to the solutions obtained from shaking; see line 16. The aim of this local search procedure is to systematically examine the quality of solutions within a smaller neighborhood of the solution suggested by shaking. Finally, if the new solution s is better than s_{bsf} $(F(s) < F(s_{\text{bsf}}))$, s_{bsf} is set to s and the shaking strength is set to the lower limit $\kappa_{\min}$. Moreover, the variable $\mathrm{it}_{\text{impr}}$ that stores the last iteration at which s_{bsf} was improved, is set to the current iteration index called "it". In addition, if $F(s) = F(s_{\text{bsf}})$, s_{bsf} is updated with s with a probability p_{move}. However, counter $\mathrm{it}_{\text{impr}}$ is obviously not updated in this case. Otherwise—that is, if s_{bsf} is not updated with s—the shaking strength for the next iteration is increased by one in line 22. If κ, however, surpasses the maximum shaking strength $\kappa_{\max}$, it is set back to $\kappa_{\min}$. Finally, if s_{bsf} is a feasible solution, the algorithm returns s_{bsf}. Otherwise, the empty string is returned.

Termination of the algorithm is controlled by three parameters: (i) $\mathrm{it}_{\max}$: the maximal number of allowed iterations; (ii) it_{rep}: the maximal number of allowed iterations without finding an improved solution, and (iii) $t_{\max}$: the maximal execution time (in CPU seconds).

The algorithm terminates whenever one of the three stopping conditions is fulfilled. Shaking, local search, and ways of making the algorithm more efficient are outlined in the subsequent sections.

3.4.1. Shaking

Shaking is realized in dependence on the feasibility of the respective solution.

$\mathsf{Shaking\_Delete}(s, \kappa)$: This procedure is applied if s is feasible. It consists in randomly remov-

Algorithm 2 VNS for the (m, k)–CLCS problem.

```
1: Input: $(S, \mathcal{P}, \Sigma)$, problem instance; $\kappa_{\min}$, minimal neighborhood size; $\kappa_{\max}$, maximal neighborhood size; $\mathrm{it}_{\max}$, maximal number of iterations; $\mathrm{it}_{\mathrm{rep}}$, maximal number of iterations without improving (current) best solution; $t_{\max}$, maximal execution time; $p_{\mathrm{move}}$, probability of accepting a solution of the same quality; $s_{\mathrm{init}}$, initial solution (may be empty).
2: $s_{\mathrm{bsf}} \leftarrow s_{\mathrm{init}}$
3: if $s_{\mathrm{bsf}} = \varepsilon$ then LocalSearch$(s_{\mathrm{bsf}}, S, P, \Sigma)$ endif
4: $\mathrm{it} \leftarrow 1$
5: $\mathrm{it}_{\mathrm{impr}} \leftarrow 1$
6: $\kappa \leftarrow \kappa_{\min}$
7: while $\mathrm{it} \leq \mathrm{it}_{\max}$ and $(\mathrm{it} - \mathrm{it}_{\mathrm{impr}}) < \mathrm{it}_{\mathrm{rep}}$ and time limit $t_{\max}$ not reached do
8:     $\mathrm{it} \leftarrow \mathrm{it} + 1$
9:     $s \leftarrow s_{\mathrm{bsf}}$     // copy best-so-far solution
10:    if $F(s) < 1$ then     // solution is feasible
11:        Shaking_Delete$(s, \kappa)$
12:    else
13:        Shaking_Change$(s, \kappa)$
14:    end if
15:    $r_{\mathrm{num}} \leftarrow$ uniform random number $\in [0, 1]$
16:    LocalSearch$(s, S, P, \Sigma)$
17:    if $F(s) < F(s_{\mathrm{bsf}})$ or $(F(s) = F(s_{\mathrm{bsf}})$ and $r_{\mathrm{num}} \leq p_{\mathrm{move}})$ then
18:        $s_{\mathrm{bsf}} \leftarrow s$
19:        if $F(s) < F(s_{\mathrm{bsf}})$ then $\mathrm{it}_{\mathrm{impr}} \leftarrow it$ end if
20:        $\kappa \leftarrow \kappa_{\min}$
21:    else
22:        $\kappa \leftarrow \kappa + 1$
23:    end if
24:    if $\kappa > \kappa_{\max}$ then $\kappa \leftarrow \kappa_{\min}$ end if
25: end while
26: if $F(s_{\mathrm{bsf}}) > 1$ then $s_{\mathrm{bsf}} \leftarrow \varepsilon$ end if
27: output: $s_{\mathrm{bsf}}$ (either a feasible solution or the empty string)
```

ing κ letters from s in order to move away from the current solution. Especially for larger values of κ, this will tend to make solution s infeasible. The hope is that later, during local search, s will be turned into an improved feasible solution.

Shaking_Change(s, κ): This procedure is applied if s is infeasible. It selects κ random positions in s and changes the letters at these positions to randomly chosen letters from Σ.

The first shaking variant is likely to produce a higher level of perturbation, i.e., it is suitable when feasibility is already achieved and when it is therefore rather safe to diversify the search. The second variant is more cautious and aims at reaching feasibility.

Algorithm 3 Function LocalSearch(s, S, P, Σ) of Algorithm 2

```
1:  Input: $s$, solution; $(S, P, \Sigma)$, problem instance
2:  improved ← $true$
3:  while improved do
4:      while improved and $F(s) \geq 1$ do      // only done if $s$ is infeasible
5:          $s' \leftarrow$ Change-Based-LS$(s, S, P, \Sigma)$
6:          if $F(s') < F(s)$ then $s \leftarrow s'$ else improved ← $false$ end if
7:      end while
8:      improved ← $false$
9:      $s' \leftarrow$ Insert-Based-LS$(s, S, P, \Sigma)$
10:     if $F(s') < F(s)$ then
11:         $s \leftarrow s'$
12:         improved ← $true$
13:     end if
14: end while
```

3.4.2. Local Search

Local search (LS) is done by combining two first improvement strategies with a time complexity of $O(|s| \cdot |\Sigma|)$ per LS iteration. The details of LS are given in Algorithm 3.

Change-based local search (see line 5) is executed while the current solution is infeasible. Once the solution becomes feasible or in the case this local search cannot make an improvement, insertion-based local search (see line 9) is performed until no further improvement can be achieved. In the following these two types of local search are described.

Change-Based-LS(s, S, P, Σ): First, the letters from Σ are stored in a pre-defined order in a string σ. Moreover, a string σ^* is generated by copying σ and by appending a dummy letter $*$. The goal is to find a pair of indices (x, y), where $x = 1, \ldots, |s|$ and $y = 1, \ldots, |\sigma^*|$ such that s is improved by exchanging the letter at position x of s by the letter $\sigma^*[y]$. In case $\sigma^*[y] = *$, this means that the letter at position x of s is deleted from s. Once the first pair of such indices is found, the solution resulting from the corresponding change is immediately returned as a result. Otherwise, the input solution s is returned. In order to avoid a bias that might arise from the order in which the positions of s and the letters are considered, the search for an improving pair of indices starts from a random position and from a random letter.

Insert-Based-LS(s, S, P, Σ): The goal of this local search is to find a pair of indices (x, y), where $x = 1, \ldots, |s| + 1$ and $y = 1, \ldots, |\sigma|$ such that s is improved by inserting letter $\sigma[y]$ before position x in s. Again, the resulting solution is returned immediately, once the first pair of such indices is found. Otherwise, the input solution s is returned. Moreover, as in the case of Change-Based-LS(s, S, P, Σ), the search for an improving pair of indices starts from a random position for insertion and from a random letter.

Pseudocodes of Change-Based-LS and Insert-Based-LS procedures are shown in Algorithm 2 and 3 in the document on supplementary material.

3.4.3. Making the VNS More Efficient

The computational bottleneck of our VNS algorithm is the calculation of the fitness $F(s)$ of a solution s as described in (9). This is because $m+k$ applications of dynamic programming for the calculation of the LCS between s and the inputs strings, respectively between s and the pattern strings, are required for the calculation of $F(s)$. In order to reduce the computational burden, we developed an incremental way of calculating $F(s')$ from $F(s)$ in the case s' is obtained from s by an insertion, a change, or a deletion of a single letter. This incremental evaluation is only applied within the two local search procedures Most importantly, this incremental evaluation works without the application of dynamic programming. It is based on a series of internal data structures that must be initialized, updated, and maintained. On the downside, the resulting fitness value is only an approximation in some cases, occasionally overestimating the real fitness function value. Therefore, whenever an improved solution is found, the real value of $F()$, and the content of the internal data structures, is calculated based on dynamic programming. Due to the large amount of technical details in the description of the aforementioned data structures and the adaptive fitness value calculation, we refer the interested reader to the source code at `https://github.com/kartelj/gclcs_public` and restrict ourselves here to the presentation of three illustrative examples.

Example 1. Given input strings $s_1 = \texttt{aabcaabaad}$, $s_2 = \texttt{aaabcabada}$, patterns $p_1 = \texttt{bcabaa}$, $p_2 = \texttt{aabbaa}$ and a solution string $s = \texttt{abccada}$, the goal is to find a character to be inserted into s and to find a position at which this character is inserted into s such that the fitness function is improved. For this purpose, first the *rightmost* LCS for each pair of strings in $\{(s_1, s), (s_2, s), (p_1, s), (p_2, s)\}$ must be determined. For example, the rightmost LCS for the pair (s_1, s) is constructed based on a dynamic programming table DP as follows. The starting cell is the cell $DP[i, j]$ with $i \leftarrow |s|$ and $j \leftarrow |s_1|$. Then, the following is done in an iterative way until either $i = 0$ or $j = 0$:

1. If $s[i] = s_1[j]$, include the character as a part of the LCS and reduce both i and j by one;

2. Otherwise, compare the values stored in $DP[i-1, j]$ and $DP[i, j-1]$ and move to the cell with the greater value or to $DP[i-1, j]$ in case of equal values.

The rightmost LCS of (s_1, s) is `abcad`[2]. Similarly, the rightmost LCS of (s_2, s) is `abcada`, the rightmost LCS of (p_1, s) is `bcaa`, and the rightmost LCS of (p_2, s) is `abaa`.

The next step consist in the determination of the left and right mappings of each of the obtained LCSs to the corresponding input string, respectively pattern. The left mapping of a string s to a string s' is hereby done as follows. First, the position x_1 of the first occurrence of letter $s[1]$ in s' is determined. This is the position to which $s[1]$ is then mapped. Subsequently, the position x_2 of the first occurrence of letter $s[2]$ in $s'[x_1 + 1, |s'|]$ is determined. This is the position to which $s[2]$ is mapped. This process is continued until each letter of s is mapped. The right mapping is obtained by repeating the same process, but in the opposite direction, i.e., both s and s' are traversed backwards. Concerning the left and right mapping of the corresponding LCSs to the input strings, respectively pattern, *middle regions* are non-empty substrings bounded from the left by the left mapping of the x-th LCS letter (including the

[2]See Figure 1 in the supplementary material document for an illustration.

boundary) and bounded from the right by the right mapping of the x-th LCS letter (excluding the boundary). Left and right mappings, along with all middle regions for all input strings and patterns are shown in Table 1.

Let us consider only the mapping to s_1, together with the corresponding middle regions, and let us exemplary check what will happen when s = `abccada` is extended to s^{new} = `abcc`$\boxed{\texttt{b}}$`ada`. Obviously, letter `b` will fall into the middle region `aaba`. The crucial point here is that it is safe to insert any character found inside the middle region. This is demonstrated in Table 2. Notice that `abc` can be mapped to s_1 according to the left part of the left mapping ($\boxed{\texttt{abc}}$`ad`) and `ad` according to the right part of the right mapping (`abc`$\boxed{\texttt{ad}}$) and neither of these mappings does consider the middle region of the input string s_1 = `aabc`$\boxed{\texttt{aaba}}$`ad`.

Table 1: Middle regions (shown with a light-gray background) for solution s = `abccada` w.r.t. input strings and patterns.

s_1	$\boxed{a}$	a	b	c	a	a	b	a	a	d
left mapping	a		b	c	a					d
right mapping		a	b	c					a	d
s_2	$\boxed{a}$	a	a	b	c	a	b	a	d	a
left mapping	a			b	c	a			d	a
right mapping			a	b	c			a	d	a
$p1$	b	c	a	b	a	a				
left mapping	b	c	a		a					
right mapping	b	c			a	a				
$p2$	$\boxed{a}$	a	$\boxed{b}$	b	a	a				
left mapping	a		b		a	a				
right mapping		a		b	a	a				

Table 2: Solution s^{new} = `abcc`$\boxed{\texttt{b}}$`ada` w.r.t. input strings and patterns after insert.

s_1	a	a	b	c	a	a	$\boxed{b}$	a	a	d
left part	a		b	c						
middle part							$\boxed{b}$			
right part									a	d
s_2	a	a	a	b	c	a	$\boxed{b}$	a	d	a
left part	a			b	c					
middle part							$\boxed{b}$			
right part								a	d	a
$p1$	b	c	a	$\boxed{b}$	a	a				
left part	b	c								
middle part				$\boxed{b}$						
right part					a	a				
$p2$	a	a	$\boxed{b}$	b	a	a				
left part	a									
middle part			$\boxed{b}$							
right part				b	a	a				

The next step is to check whether a fitness function value improvement can be achieved by inserting a certain character at a certain position x in the current solution s, obtained by shifting the character currently found at the x-th position, together with all subsequent characters, to the right. In order to quantify the total effect of this character insertion on the fitness function value, it must be considered how this separately influences the LCSs between the new solution string and all input strings, respectively patterns. This is done in the same manner as previously exemplified for s_1. There are several possible improvements that can be performed according to Table 1. Inserting character a at the first position ($s^{\text{new}} = \boxed{\texttt{a}}\texttt{abccada}$) would increase the LCS w.r.t. s_1, s_2, and p_2. Therefore, the total effect will be a reduction of the number of feasibility violations by one. Note that the number of feasibility violations w.r.t. s_1 and s_2 will remain unchanged since they are calculated as $|s^{\text{new}}| - |\text{LCS}(s_i, s^{\text{new}})|$ (the LCSs are increased, but s^{new} as well). Also, there will be no effect when it comes to p_1, i.e., the number of feasibility violations will remain the same. The second option would be to insert character a at the fifth position ($s^{\text{new}} = \texttt{abcc}\boxed{\texttt{a}}\texttt{ada}$). This will in total decrease the number of feasibility violations by one, only that now the LCS w.r.t. p_1 will be improved instead of the one w.r.t. p_2. Finally, the insertion of character b at the fifth position ($s^{\text{new}} = \texttt{abcc}\boxed{\texttt{b}}\texttt{ada}$) proves to be the best option, since this decreases the total number of feasibility violations by two (see Table 2).

The reason why the presented approach enables a fast calculation of the LCSs after the insertion of a single character is because there is no need to perform dynamic programming calculations for each character/position probing. By maintaining appropriate data structures concerning the region mappings, one can check quickly whether a certain region is a candidate for an insertion and, if this is the case, the region is scanned in linear time.

When it comes to single character edit/delete operations, we will uniquely denote them as a *change* operation, since deletion can be viewed as an edit operation in which the previous character is replaced with the empty string ε. Obviously, deletion is focused on reducing the number of feasibility violations with respect to the input strings, while a change operation may, in general, reduce the number of feasibility violations with respect to both input strings and patterns (9). The procedure for checking the suitability of a change operation is similar to the one described above. However, the newly calculated LCS lengths will not always be correct, i.e., the LCSs obtained in this way may be shorter than the real ones. This means that the fitness function value obtained in this way is an upper bound for the exact fitness function value. Obtaining an upper bound of the new exact fitness function value quickly is nevertheless useful because, if it is better (lower) than the previous known exact fitness function value, the new exact fitness function value must also be better than the old one.

Example 2. Given the same set of input strings, patterns, and the solution string as in Example 1, the goal is to determine efficiently a lower bound of the exact LCS lengths after a change operation. The procedure is very similar to the one for insertion operations described above. First, the middle regions between left and right mappings are determined and subsequently there is character/position probing. The only difference in comparison to the insertion scenario is that the solution string is not changed by inserting characters at certain positions but by changing already existing ones. For example, changing $s = \texttt{abc}\boxed{\texttt{c}}\texttt{ada}$ to $s^{\text{new}} = \texttt{abc}\boxed{\texttt{b}}\texttt{ada}$ would lead to a feasibility violations decrease w.r.t. all input strings and all patterns—four in total. In this particular case, all efficiently calculated LCS lengths will match the exact ones.

Example 3. Finally, we want to show a case in which the efficient LCS calculation does not provide the exact LCS lengths but only lower bounds. One of the situations in which this happens is when changing characters that are included in the selected LCS. For example, the change $s = \texttt{abcca}\boxed{\texttt{d}}\texttt{a}$ to $s^{\text{new}} = \texttt{abcca}\boxed{\texttt{a}}\texttt{a}$ is never considered in the efficient LCS calculation w.r.t. s_1 and s_2, since `d` is not part of a middle region. However, notice that changing it to character `a` would produce $\text{LCS}(s_1, s^{\text{new}}) = \texttt{abcaaa}$, which has length six, which is an improvement by one.

4. Experimental Evaluation

We include the following five algorithms (resp. algorithm variants) in our experimental evaluation:

i. The Greedy procedure from Section 3.1 (Greedy);

ii. The basic variant of the BS in which all letters from Σ_v^{nd} are used as extensions, denoted as basic-BS;

iii. The restricted version of BS targeted to maximize the performance in finding any feasible solution, denoted as restricted-Bs;

iv. The variable neighborhood search from Section 3.4 denoted as VNS;

v. A hybrid BS&VNS in which restricted-BS is performed first to find an initial solution which is then further optimized by the VNS.

All algorithms were implemented in C++ using GCC 7.4, and the experiments were conducted in single-threaded mode on a machine with an Intel Xeon E5–2640 processor with 2.40 GHz and a memory limit of 16 GB for `BS` and restricted-Bs, and 4Gb for VNS and BS&VNS. The maximal CPU time allowed for each run and each of our algorithm was set to 20 minutes, i.e., 1200 seconds. The public repository with the source code of our implementations, instances and raw results can be found at `https://github.com/kartelj/gclcs_public`.

4.1. Benchmark Instances

We consider two different sets of benchmark instances for the experiments, a fully random set and a real-world instance set. These are described in the next sections.

4.1.1. Random Benchmark Set

Properties of each instance set include the number of strings m, the length of the input strings n (all input string are of the same length), the number of pattern strings k, the alphabet size $|\Sigma|$, and the ratio p between n and the length of the pattern strings (all pattern strings have the same length). The following procedure has been used to generate this random type of benchmark instances:

1. Pattern strings $P_1, \ldots, P_k$ are generated uniformly at random, all with length $\lfloor n/p \rfloor$ $(p \cdot k < n)$.

2. Each input string $s_i, i = 1, \ldots, m$, of length n is generated as follows in order to ensure the existence of a feasible solution:
   - For each pattern string p_j, $j = 1, \ldots, k$, $\lfloor n/p \rfloor$ positions of s_i are randomly selected from $\{(j-1) \cdot k, \ldots, j \cdot k\}$ without replacement.
   - These selected positions in s_i are filled with the letters of pattern string p_j in the respective order.
   - All remaining positions in s_i are filled with letters selected uniformly at random from alphabet Σ.

For all combinations of $m \in \{2, 5, 10\}$, $n \in \{100, 500, 1000\}$, $k \in \{2, 5, 10\}$, $p = \frac{n}{|P_0|} \in \{20, 50\}$, and $|\Sigma| \in \{2, 4, 20\}$ ten independent instances were generated, which gives in total 1620 instances. We call this set Random.

4.1.2. Real-World Benchmark Set

This set uses 12,681 bacterial 16S rRNA gene sequences. The whole set is divided into 49 classes, where each class contains the sequences from one bacterial phylum. The cardinality of these classes varies. Ten classes contain only one sequence, so they are omitted here. Some classes contain only few sequences, while four classes contain thousands and more sequences. More detailed information about the cardinality of each class is provided in Table 4 of the document on supplementary material. This dataset is taken from the website that provides the *Mothur* software package, one of the most widely used tools for analyzing 16S rRNA gene sequence data [35].

As pattern strings, we use continuous sequences of molecules, called "primer contigs", which are generated by the assembly of all of the primers reported for conserved regions of a sequence. A more detailed description of this benchmark set is provided in Section 4.7 in conjunction with the analysis of the results. This set of instances is labeled by Real.

4.2. Parameter Tuning of the Algorithms

We tuned the parameters of basic-BS, restricted-BS, and VNS with the automated parameter configuration tool *irace* [26].

The outcome of the tuning experiments and the configurations of our algorithms' parameters used for the experimental evaluation can be found in Section 1.4 of the document on supplementary material.

4.3. Results of Methods with Focus on Feasibility on Random *Instances*

In this section, we first focus only on finding any feasible solutions to our Random benchmark instances. Consequently, we compare Greedy, restricted-BS, and VNS, the latter two with the parameter settings determined by the tuning described in the previous section. While Greedy and restricted-BS were performed once on each benchmark instance, VNS was performed ten times due to its stochastic nature.

Tables 3 – 4 show the results of the aforementioned three algorithms for $|\Sigma| \in \{4, 20\}$, respectively[3]. In each table, the first four columns indicate the instance characteristics followed by the three blocks reporting the results for each algorithm in the form of the average

[3]Additional results concerning the instances for $|\Sigma| = 2$ are given by Tables 1–2 in the document on supplementary material.

solution length ($\overline{|s|}$) of found feasible solutions, the average runtime ($\bar{t}$[s]) and the percentage of instances for which at least one feasible solution was found (*feas*[%]). For VNS, average objective values of the best solutions among the ten runs per each instance is additionally reported by column $\overline{|s^{\max}|}$. Missing values indicate that an algorithm could not find a feasible solution in any of its runs. Best percentages of feasible solutions found are highlighted in bold font.

In order to check the statistical significance of observed performance differences of our algorithms, we perform a statistical test as follows. Friedman's test was executed on the results of all algorithms that are grouped w.r.t. alphabet size $|\Sigma|$. Given that, if the test rejected the hypothesis that the results of the compared algorithms on the same group of instances are (statistically) equal, pairwise comparisons are performed using the Nemenyi post-hoc test [34]. The outcome is shown by means of so-called critical difference plots, for each group of results. In short, each algorithm is positioned in the horizontal segment according to its average ranking concerning the considered group 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 smaller than the 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. Due to completeness, each CD plot in this paper shows a comparison between all five algorithms. The plots concerning this section are given by Figure 6(a)–(b) and the algorithms are evaluated according to number of instances for which a feasible solution is successfully found. The following conclusions can be drawn.

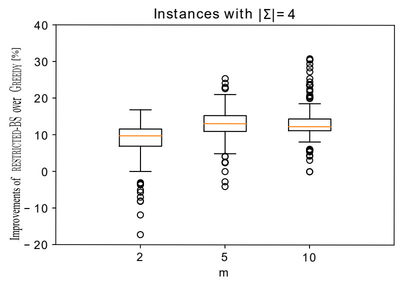
- For the instances with $|\Sigma| = 4$ (see Table 3) RESTRICTED-BS can obtain feasible solutions for all these instances. This also holds for VNS with a few exceptions (eight out of 5400 runs) and for GREEDY (nine out of 540 runs), see also Figure 7. Concerning solution quality, VNS was able to outperform the other approaches in particular on the instances with smaller n. On most of the other instances, RESTRICTED-BS yielded longer solutions than the other two approaches. For example, relative improvements of RESTRICTED-BS over GREEDY for those instances where both algorithms could find a feasible solution, are shown in Figure 5(a) and range from 10% to 13%. Further, the quality of solutions obtained by VNS are better in average than the quality of solutions obtained by GREEDY.

- The instances with $|\Sigma| = 20$ (see Table 4) turn out to be the most challenging ones in respect to finding any feasible solution. For fixed n, when the alphabet size increases, the corresponding search space gets more restricted concerning the number of feasible versus infeasible nodes. RESTRICTED-BS can nevertheless deal relatively well with these instances: for 22 out of 48 instance groups a feasible solution could be identified for all instances, and for just one instance group no feasible solution could be found at all. In comparison to the other two algorithms, this difference is substantial. In few cases GREEDY could find more feasible solutions than VNS, but not significantly more (see respective plots in Figure 8). The differences w.r.t. the number of instances for which a feasible solution was found are most pronounced for $m = 10$, where VNS was successful in just around 4.2% of all runs, GREEDY in around 25% (54 out of 180 runs) but RESTRICTED-BS in around 80% (151 out of 180 runs). Concerning solution quality, average percentage improvements of RESTRICTED-BS over GREEDY are displayed in Figure 5(b). However, better solutions could not be found in all cases. Differences

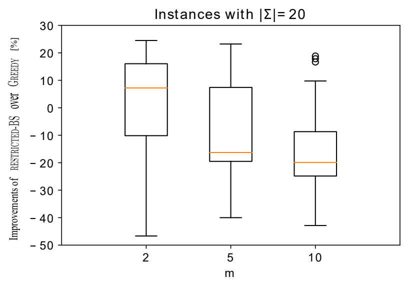
Instance				Greedy			restricted-Bs			VNS			
m	n	p	k	$\overline{\lvert s\rvert}$	$\overline{t}$[s]	feas[%]	$\overline{\lvert s\rvert}$	$\overline{t}$[s]	feas[%]	$\overline{\lvert s^{\max}\rvert}$	$\overline{\lvert s\rvert}$	$\overline{t}$[s]	feas[%]
	100	20	2	49.90	0.10	**100**	54.00	0.16	**100**	63.00	62.45	2.09	**100**
	100	50	2	55.60	0.12	**100**	58.50	0.23	**100**	62.70	62.44	2.09	**100**
	100	20	5	54.30	0.08	**100**	56.40	0.91	**100**	63.20	62.81	2.34	**100**
	100	50	5	55.40	0.10	**100**	59.60	0.26	**100**	62.90	62.42	2.31	**100**
	100	20	10	63.50	0.09	**100**	61.90	2.72	**100**	67.70	67.50	3.01	99
	100	50	10	56.80	0.10	**100**	61.00	0.35	**100**	63.20	62.84	2.94	**100**
	500	20	2	259.50	0.09	**100**	293.00	30.41	**100**	315.10	306.53	49.41	**100**
	500	50	2	276.90	0.14	**100**	306.30	32.61	**100**	309.70	302.95	44.34	**100**
	500	20	5	270.40	0.11	**100**	305.20	35.15	**100**	314.90	305.77	59.39	**100**
2	500	50	5	279.20	0.09	**100**	313.20	35.33	**100**	313.50	304.46	53.55	**100**
	500	20	10	299.50	0.11	**100**	324.00	37.73	**100**	325.20	314.58	70.81	**100**
	500	50	10	289.70	0.09	**100**	318.50	35.69	**100**	314.30	306.25	65.71	**100**
	1000	20	2	511.10	0.29	**100**	587.10	76.22	**100**	614.30	602.60	233.99	**100**
	1000	50	2	555.00	0.19	**100**	624.40	84.69	**100**	619.60	605.29	254.74	**100**
	1000	20	5	545.60	0.33	**100**	617.50	83.72	**100**	621.10	606.27	287.96	**100**
	1000	50	5	560.60	0.30	**100**	629.60	80.28	**100**	615.70	602.91	271.09	**100**
	1000	20	10	597.50	0.36	**100**	657.60	97.72	**100**	640.70	620.82	355.24	**100**
	1000	50	10	573.20	0.24	**100**	637.60	101.18	**100**	618.10	605.92	382.46	**100**
	100	20	2	35.80	0.10	**100**	39.80	4.66	**100**	42.70	41.61	1.72	**100**
	100	50	2	36.50	0.09	**100**	40.10	5.67	**100**	41.70	40.46	2.09	**100**
	100	20	5	37.90	0.10	**100**	41.40	5.43	**100**	44.30	43.14	1.85	99
	100	50	5	36.50	0.07	**100**	40.70	5.68	**100**	42.60	41.29	2.06	**100**
	100	20	10	46.50	0.10	80	50.00	6.86	**100**	53.00	52.79	2.66	99
	100	50	10	37.30	0.08	**100**	42.00	6.14	**100**	43.60	42.62	2.34	**100**
	500	20	2	179.90	0.12	**100**	209.10	39.45	**100**	212.60	207.05	78.62	**100**
	500	50	2	187.70	0.09	**100**	217.40	47.38	**100**	211.40	206.98	68.26	**100**
5	500	20	5	190.10	0.11	**100**	221.20	46.08	**100**	219.30	212.15	83.12	**100**
	500	50	5	191.30	0.10	**100**	218.80	45.09	**100**	212.60	206.15	76.22	**100**
	500	20	10	208.40	0.12	**100**	253.20	54.96	**100**	253.50	234.68	112.12	**100**
	500	50	10	194.70	0.11	**100**	223.50	53.95	**100**	214.60	209.48	84.56	**100**
	1000	20	2	365.50	0.34	**100**	423.00	87.94	**100**	423.10	415.60	384.62	**100**
	1000	50	2	382.60	0.24	**100**	439.40	95.50	**100**	422.10	413.48	386.03	**100**
	1000	20	5	374.70	0.33	**100**	442.50	92.51	**100**	429.40	419.63	407.46	**100**
	1000	50	5	385.90	0.23	**100**	446.20	102.24	**100**	423.10	414.22	411.43	**100**
	1000	20	10	401.60	0.30	**100**	506.60	114.83	**100**	473.70	444.53	512.43	**100**
	1000	50	10	392.80	0.29	**100**	451.30	103.43	**100**	425.70	415.47	442.96	**100**
	100	20	2	29.11	0.11	90	32.80	4.28	**100**	34.50	33.20	1.82	**100**
	100	50	2	28.70	0.09	**100**	33.20	5.46	**100**	33.80	32.69	2.26	**100**
	100	20	5	32.62	0.12	80	35.10	5.19	**100**	37.40	35.89	1.94	98
	100	50	5	29.30	0.09	**100**	33.60	5.25	**100**	34.60	33.49	2.10	**100**
	100	20	10	41.38	0.09	80	45.90	7.07	**100**	50.20	50.17	2.98	97
	100	50	10	31.44	0.08	90	34.90	5.79	**100**	36.00	34.98	2.47	**100**
	500	20	2	155.20	0.13	**100**	175.50	38.22	**100**	172.90	168.42	95.43	**100**
	500	50	2	159.00	0.13	**100**	179.50	44.20	**100**	171.60	167.98	99.90	**100**
10	500	20	5	161.30	0.12	90	185.90	42.89	**100**	180.00	174.68	106.31	**100**
	500	50	5	160.20	0.12	**100**	182.60	44.24	**100**	172.70	169.06	103.08	**100**
	500	20	10	186.40	0.12	**100**	235.40	55.07	**100**	251.00	246.14	170.76	**100**
	500	50	10	164.50	0.12	**100**	188.50	47.65	**100**	176.80	171.78	111.34	**100**
	1000	20	2	310.70	0.32	**100**	357.90	81.04	**100**	347.60	340.25	541.96	**100**
	1000	50	2	324.10	0.35	**100**	367.50	88.58	**100**	346.30	339.25	532.82	**100**
	1000	20	5	320.70	0.51	**100**	373.20	90.20	**100**	353.50	346.28	571.40	**100**
	1000	50	5	328.30	0.32	**100**	372.70	89.02	**100**	348.40	340.76	515.36	**100**
	1000	20	10	348.60	0.39	**100**	478.40	110.21	**100**	497.90	420.76	864.54	**100**
	1000	50	10	334.40	0.29	**100**	381.80	93.83	**100**	351.30	334.81	556.73	**100**

Table 3: Results of the algorithms with focus on finding any feasible solution; benchmark set Random, $|\Sigma| = 4$.

are largest on the instances with $|\Sigma| = 20$ and larger m, where Greedy was able to occasionally outperform the results of restricted-BS. However, this happens on a rather small amount of instances (the common number of instances for which the two algorithms find a feasible solution is 120; in 66 cases Greedy delivers a better solution) in comparison to the size of the whole benchmark set Random. The reason for this behavior on this subset of instances lies in our probabilistic guidance of the beam search, which focuses on finding any feasible but not so much on solution quality. Here, it rarely occurs that each node can be expanded towards a feasible solution.

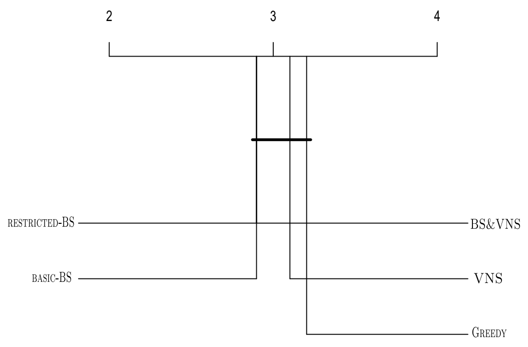


(a) $|\Sigma| = 4$.

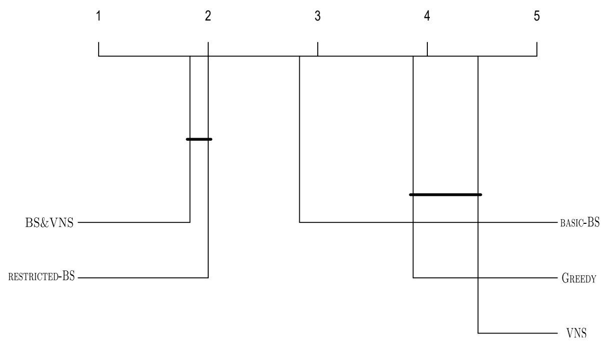


(b) $|\Sigma| = 20$.

Figure 5: Improvements of restricted-BS over Greedy: benchmark set Random.



(a) $|\Sigma| = 4$



(b) $|\Sigma| = 20$

Figure 6: CD plots that compare finding a feasible solution: benchmark set Random.

Instance				Greedy			restricted-Bs			VNS			
m	n	p	k	$\overline{\lvert s\rvert}$	$\bar{t}$[s]	feas[%]	$\overline{\lvert s\rvert}$	$\bar{t}$[s]	feas[%]	$\overline{\lvert s^{\max}\rvert}$	$\overline{\lvert s\rvert}$	$\bar{t}$[s]	feas[%]
	100	20	2	20.00	0.09	20	16.75	0.04	**40**	30.75	30.75	1.22	5
	100	50	2	25.30	0.11	**100**	23.89	0.03	90	32.70	31.99	1.19	98
	100	20	5	-	-	0	24.89	0.87	**90**		-	-	0
	100	50	5	30.00	0.06	20	19.71	0.05	**70**	33.22	32.97	1.71	34
	100	20	10	53.00	0.06	10	42.90	15.14	**100**	-	-	-	0
	100	50	10	31.50	0.06	20	22.00	3.23	**30**	35.00	35.00	6.59	1
	500	20	2	-	-	0	72 .00	4.96	**30**	-	-	-	0
	500	50	2	100.12	0.11	80	111.60	5.60	**100**	163.00	161.30	34.48	7
	500	20	5	-	-	0	120.00	58.97	**10**	-	-	-	0
2	500	50	5	101.00	0.14	10	115.30	25.97	**100**	-	-	-	0
	500	20	10	-	-	0	209.43	130.80	**70**	-	-	-	0
	500	50	10	-	-	0	130.30	64.98	**100**	-	-	-	0
	1000	20	2	-	-	0	146.00	34.18	**10**	-	-	-	0
	1000	50	2	202.33	0.22	90	236.40	54.77	**100**	-	-	-	0
	1000	20	5	-	-	0	231.50	85.87	**20**	-	-	-	0
	1000	50	5	-	-	0	237.80	73.79	**100**	-	-	-	0
	1000	20	10	-	-	0	415.89	285.97	**90**	-	-	-	0
	1000	50	10	249.00	0.21	10	257.90	118.99	**100**	-	-	-	0
	100	20	2	14.00	0.11	10	10.25	0.07	**80**	15.00	15.00	2.10	3
	100	50	2	10.00	0.08	40	8.75	0.13	40	12.70	12.42	1.10	**73**
	100	20	5	25.00	0.06	30	22.90	0.22	**100**	-	-	-	0
	100	50	5	14.00	0.05	10	10.57	0.14	**70**	13.33	13.33	2.21	6
	100	20	10	49.67	0.06	90	41.40	13.70	**100**	-	-	-	0
	100	50	10	21.00	0.14	10	17.78	1.72	**90**	-	-	-	0
	500	20	2	-	-	0	-	-	0	-	-	-	0
	500	50	2	49.00	0.12	40	53.90	32.80	**100**	64.00	64.00	24.65	1
5	500	20	5	-	-	0	114.60	67.76	**100**	-	-	-	0
	500	50	5	-	-	50	57.67	34.79	**90**	-	-	-	0
	500	20	10	249.50	0.14	20	205.22	151.49	**90**	-	-	-	0
	500	50	10	-	-	0	83.57	67.90	**70**	-	-	-	0
	1000	20	2	-	-	0	98.50	57.62	**20**	-	-	-	0
	1000	50	2	94.00	0.24	30	114.10	102.53	**100**	-	-	-	0
	1000	20	5	-	-	0	226.70	134.75	**100**	-	-	-	0
	1000	50	5	-	-	0	119.20	80.90	**100**	-	-	-	0
	1000	20	10	499.00	0.34	20	411.67	339.50	**90**	-	-	-	0
	1000	50	10	-	-	0	166.17	132.29	**60**	-	-	-	0
	100	20	2	10.40	0.06	50	9.80	0.01	**100**	11.50	11.5	1.46	2
	100	50	2	6	0.05	20	5.71	0.08	**70**	7.70	7.48	1.36	63
	100	20	5	25	0.08	80	22.30	0.15	**100**	-	-	-	0
	100	50	5	-	-	0	9.55	0.03	**90**	9.67	9.13	1.36	7
	100	20	10	50	0.09	**100**	39.80	13.55	50	-	-	-	0
	100	50	10	19.83	0.06	60	16.70	0.59	**100**	20.00	20.00	6.97	3
	500	20	2	-	-	0	49.25	22.29	**80**	-	-	-	0
	500	50	2	37	0.07	10	39.80	28.57	**100**	-	-	-	0
10	500	20	5	125	0.13	10	112.30	72.37	**100**	-	-	-	0
	500	50	5	-	-	0	47.20	29.87	**50**	-	-	-	0
	500	20	10	250	0.14	**100**	206.87	183.60	80	-	-	-	0
	500	50	10	-	-	0	83.29	75.95	**70**	-	-	-	0
	1000	20	2	-	-	0	97.70	67.75	**100**	-	-	-	0
	1000	50	2	69	0.25	20	84.90	66.44	**100**	-	-	-	0
	1000	20	5	-	-	0	227.10	180.77	**100**	-	-	-	0
	1000	50	5	-	-	0	94.25	69.41	**40**	-	-	-	0
	1000	20	10	500	0.58	**90**	409.78	313.57	**90**	-	-	-	0
	1000	50	10	-	-	0	164.78	120.27	**90**	-	-	-	0

Table 4: Results of the algorithms with focus on finding any feasible solution; benchmark set Random, $|\Sigma| = 20$.

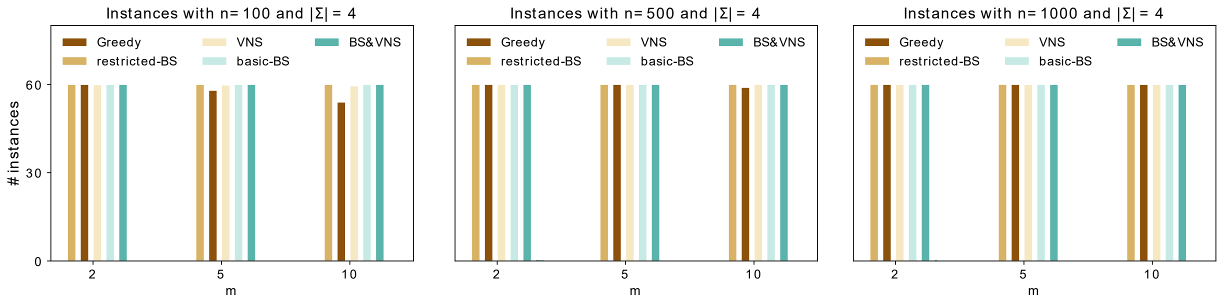
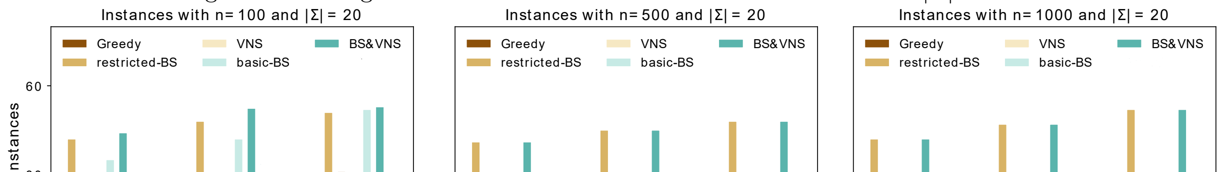


Figure 7: Finding a feasible solution on benchmark set Random for $|\Sigma| = 4$.



4.4. Results of Methods with Focus on Feasibility on REAL *Instances*

In this section, we focus on finding any feasible solutions to our REAL benchmark instances. The same algorithm are compared as those in Section 4.3. The numerical results are reported in Table 5. Concerning the statistical difference w.r.t. finding a feasible solution, the corresponding CD plot is provided in Figure 11. Due to completeness, all five approaches are shown in the plot, as also used in the latter discussion. The following conclusion can be drawn.

- VNS was able to find a feasible solution to all 40 instances, RESTRICTED-BS to 37. For GREEDY, it was possible for 39 cases, see Figure 9.
- Concerning final solution quality, VNS delivers best solutions in 37 out of 40 cases, whereas RESTRICTED-BS does it in just 3 cases. However, running times of RESTRICTED-BS are much lower than those of the VNS approach. GREEDY is clearly obtaining the worst results among the other competitors, see Figure 10 to check the improvements of RESTRICTED-BS over GREEDY, which is in average around 30%. For statistical difference w.r.t. the number of instances for which respective algorithms find a feasible solution, see Figure 11.
- VNS seems to benefit strongly from the distribution of the instances from benchmark REAL, that is, the high correlation between input strings where, in general, it gets easier to find a feasible solution than in unrelated random strings.

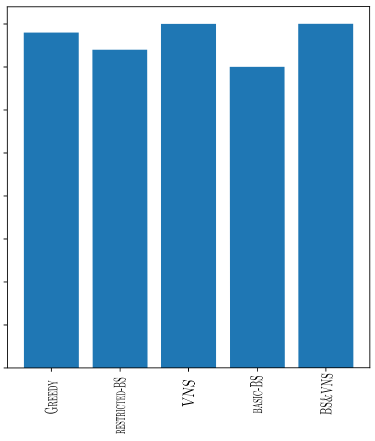


Figure 9: Feasible solution findings on benchmark REAL.

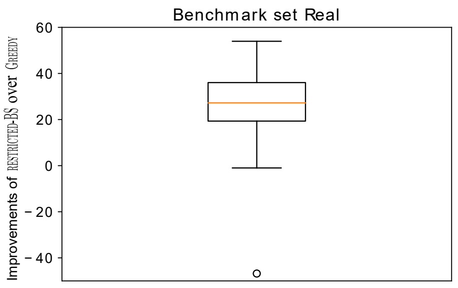


Figure 10: Improvements of RESTRICTED-BS over GREEDY: benchmark REAL.

4.5. Results of Methods with Focus on High-Quality Solutions on RANDOM *Instances*

In this section, we focus on finding high-quality solutions on benchmark set RANDOM. Consequently, we compare BASIC-BS and BS&VNS. These algorithms use the parameter settings determined by the tuning described in Section 4.2. While BASIC-BS was performed once on each benchmark instance, BS&VNS was performed ten times due to its stochastic nature. Tables 6 – 7 present the respective results for $|\Sigma| \in \{4, 20\}$[4]. Again, the average

[4]Complete results for the instances with $|\Sigma| = 2$ are provided in Table 1–2 in the document on supplementary material.

Instance	GREEDY		RESTRICTED-BS		VNS		
	$\overline{\lvert s\rvert}$	$\bar{t}$[s]	$\overline{\lvert s\rvert}$	$\bar{t}$[s]	$\overline{\lvert s^{\max}\rvert}$	avg $\overline{\lvert s\rvert}$	$\bar{t}$[s]
Acidobacteria	168	3.88	217	108.17	**265**	260.20	1200.54
Actinobacteria	345	23.67	235	1200.23	**382**	337.20	1202.43
Aminicenantes	730	0.12	1336	57.01	**1358**	1209.20	1200.03
Aquificae	432	0.33	585	48.45	**750**	646.80	1200.08
Armatimonadetes	358	0.91	490	92.37	**553**	510.30	1200.34
Atribacteria	1247	0.07	1413	26.21	**1499**	1499.00	1200.03
Bacteroidetes	321	10.86	382	357.35	**401**	391.30	1201.41
BRC1	459	0.23	666	81.67	**724**	619.70	1200.02
Candidatus-Saccharibacteria	441	0.22	597	28.89	**667**	621.60	1200.06
Chlamydiae	489	0.15	761	99.86	**935**	843.40	1200.03
Chlorobi	489	0.17	736	31.97	**803**	747.00	1200.03
Chloroflexi	297	0.46	376	61.50	**513**	500.80	1200.09
Chrysiogenetes	743	0.12	1234	115.02	**1146**	1026.10	1200.02
Cyano Chloroplast	346	2.13	76	157.62	**531**	502.90	1200.45
Deferribacteres	527	0.10	789	98.83	**969**	812.10	1200.03
Deinococcus-Thermus	428	0.58	601	128.64	**704**	623.40	1200.21
Dictyoglomi	1293	0.14	1461	107.59	**1522**	1519.20	1200.02
Elusimicrobia	686	0.12	**1115**	102.11	1042	908.00	1200.02
Fibrobacteres	638	0.13	**1099**	107.65	1068	934.70	1200.02
Firmicutes	**126**	25.54	-	-	122	110.80	1200.90
Fusobacteria	373	0.39	591	102.32	**634**	608.40	1200.11
Ignavibacteriae	778	0.34	1320	101.00	**1323**	1158.80	1200.03
Latescibacteria	457	0.16	690	83.10	**694**	610.40	1200.03
Lentisphaerae	645	0.18	**1088**	105.42	961	884.50	1200.01
Microgenomates	319	0.11	426	56.04	**455**	440.00	1200.02
Nitrospinae	1237	0.14	1389	105.03	**1431**	1431.00	1200.03
Nitrospirae	495	0.19	787	92.69	**966**	855.20	1200.03
Parcubacteria	312	0.48	433	67.27	**564**	483.50	1200.04
Planctomycetes	442	0.29	623	99.15	**747**	631.00	1200.07
Poribacteria	252	0.24	316	39.47	**423**	414.30	1200.02
Proteobacteria	165	31.34	-	-	**324**	266.10	1203.38
Spirochaetes	393	0.52	471	105.28	**569**	534.10	1200.16
SR1	454	0.16	986	113.83	**987**	837.80	1200.03
Synergistetes	440	0.36	604	31.20	**778**	661.20	1200.07
Tenericutes	365	1.16	498	63.11	**521**	507.50	1200.30
Thermodesulfobacteria	554	0.16	1187	44.00	**1192**	1005.60	1200.03
Thermotogae	460	0.41	**667**	113.86	621	588.00	1200.05
Verrucomicrobia	127	0.38	174	22.33	**247**	208.20	1159.13
WPS-1	1097	0.12	1299	53.05	**1301**	1155.00	1200.02
All	-	-	-	-	**113**	102.60	1205.95

Table 5: Results on the real instance set with focus on finding any feasible solution.

solution quality ($\overline{|s|}$), the average running time ($\bar{t}$[s]), and the percentage of instances for which the algorithm found a feasible solution (*feas*[%]) are listed for each algorithm on each instance group. For BS&VNS, the average solution quality of the maximum solutions (over 10 runs) ($\overline{|s^{\max}|}$) and the averaged solution quality (of 10 runs) ($\bar{t}$[s]) is additionally reported. Concerning the statistical difference w.r.t. the obtained solution quality, the corresponding CD plots are shown by Figures 13(a)–(b). For the sake of completion, all five algorithms are compared and shown in the plots. For VNS and BS&VNS, the average solution quality of the maximum solutions are taken in this comparison, i.e., the results from column ($\overline{|s^{\max}|}$). The following conclusions can be drawn from these results:

- For the instances with $|\Sigma| = 4$ (see Table 6) we observe that in case of small n ($n = 100$),

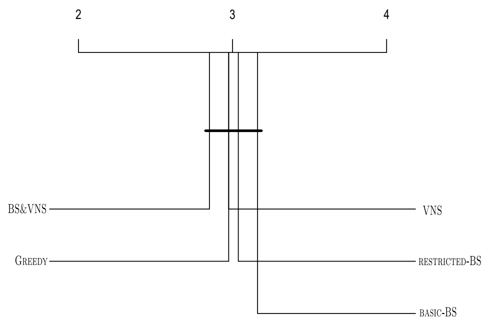
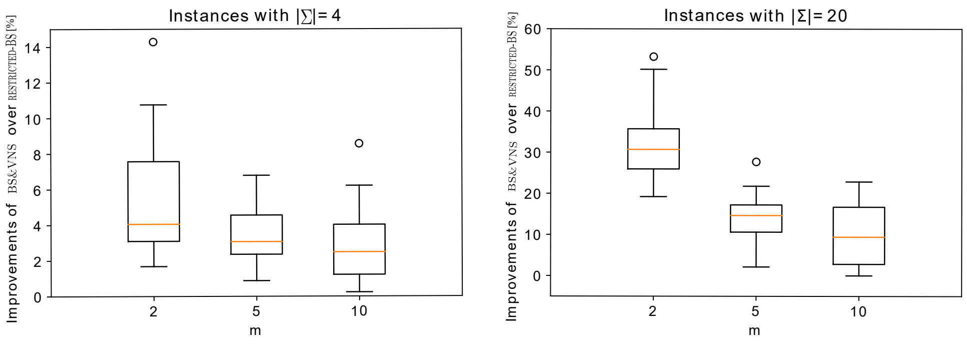


Figure 11: CD plot that compare finding a feasible solution: benchmark set Real.



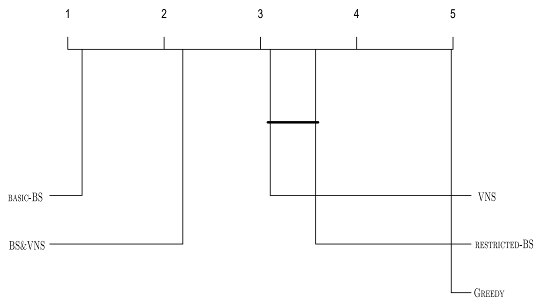
(a) $|\Sigma| = 4$.

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

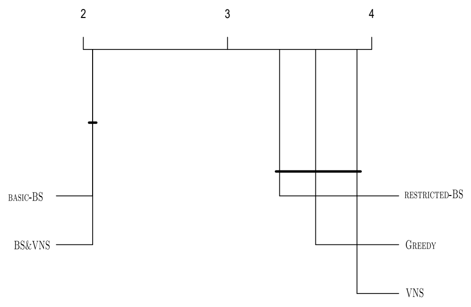
Figure 12: Improvements of BS&VNS over restricted-BS: benchmark set Random.

the obtained results of basic-BS and BS&VNS outperform those of the other approaches. BS&VNS improves the (initial) solutions of restricted-BS by about three percent on average, see Figure 12(a). Looking at the broader perspective, basic-BS performs best and earns significant advantages over BS&VNS, see Figure 13(a). The largest differences are pronounced in case of the instances with larger n, and especially for those with $n = 1000$, where basic-BS obtains substantially better results than BS&VNS within much lower computation times.

- For the instances with $|\Sigma| = 20$ (see Table 7) it can be noticed that the results of BS&VNS are significantly better than those of restricted-BS where average improvements lie above 10%, see Figure 12(b). Concerning the task of finding at least feasible solution, basic-BS was successful for 254 out 540 instances. Note that for restricted-BS it was possible in 416 cases. The hybrid BS&VNS found a feasible solution in 420 out of 540 instances. The solution is of the similar quality than the respective solution of BS&VNS, and difference is small and statistically not relevant; see Figure 13(b). On those instances where it gets hard to find a feasible solution, the best option thus is to apply restricted-BS for obtaining a solution which is then further improved by VNS.



(a) $|\Sigma| = 4$



(b) $|\Sigma| = 20$

Figure 13: CD plots comparing solution quality: benchmark set Random.

Instance				BASIC-BS				BS&VNS		
m	n	p	k	$\overline{\lvert s\rvert}$	$\bar{t}$[s]	feas[%]	$\overline{\lvert s_{\max}\rvert}$	$\overline{\lvert s\rvert}$	$\bar{t}$[s]	feas[%]
	100	20	2	63.00	0.53	100	63.00	62.38	1.77	100
	100	50	2	62.70	0.07	100	62.70	62.69	1.71	100
	100	20	5	63.20	1.54	100	63.20	63.1	2.01	100
	100	50	5	63.00	0.13	100	62.90	62.76	2.03	100
	100	20	10	67.70	1.51	100	67.60	67.50	2.59	100
	100	50	10	63.20	0.42	100	63.20	63.11	2.48	100
	500	20	2	322.40	11.66	100	319.50	318.43	30.51	100
	500	50	2	319.00	11.78	100	318.70	318.36	24.64	100
	500	20	5	323.50	11.72	100	322.10	321.71	35.21	100
2	500	50	5	322.70	13.27	100	322.00	321.76	29.85	100
	500	20	10	337.70	14.74	100	336.10	334.55	42.78	100
	500	50	10	325.10	12.74	100	324.10	323.26	37.85	100
	1000	20	2	645.60	22.79	100	637.20	634.42	172.48	100
	1000	50	2	648.70	23.62	100	646.70	645.14	130.15	100
	1000	20	5	649.90	30.02	100	644.70	643.12	165.64	100
	1000	50	5	646.20	27.69	100	642.30	641.62	138.95	100
	1000	20	10	680.60	29.94	100	677.80	676.46	220.80	100
	1000	50	10	649.90	28.28	100	648.50	648.07	163.12	100
	100	20	2	42.90	4.36	100	42.70	42.39	1.41	100
	100	50	2	41.90	3.72	100	41.40	41.27	1.55	100
	100	20	5	44.40	4.38	100	43.90	43.44	1.35	100
	100	50	5	42.70	4.07	100	41.80	41.71	1.55	100
	100	20	10	53.00	5.10	100	53.00	52.97	2.10	100
	100	50	10	43.90	4.87	100	43.30	43.09	1.85	100
	500	20	2	223.90	23.31	100	220.60	219.02	47.19	100
	500	50	2	224.30	27.63	100	223.70	223.01	42.70	100
	500	20	5	231.40	24.26	100	228.70	227.24	51.67	100
5	500	50	5	223.80	24.61	100	222.30	221.71	42.77	100
	500	20	10	263.20	27.19	100	263.00	262.19	67.57	100
	500	50	10	227.30	24.80	100	226.60	225.73	51.26	100
	1000	20	2	452.00	48.38	100	444.50	441.81	270.77	100
	1000	50	2	451.50	44.68	100	449.60	448.00	207.94	100
	1000	20	5	463.00	39.02	100	455.90	453.67	295.42	100
	1000	50	5	454.10	44.43	100	451.70	450.62	220.61	100
	1000	20	10	526.00	65.81	100	524.70	522.51	317.21	100
	1000	50	10	458.30	56.00	100	455.30	454.29	221.03	100
	100	20	2	35.00	3.50	100	34.30	34.03	1.29	100
	100	50	2	34.30	3.50	100	33.70	33.51	1.70	100
	100	20	5	37.70	3.30	100	37.20	36.49	1.45	100
	100	50	5	35.00	3.71	100	34.50	33.96	1.57	100
	100	20	10	50.20	6.52	100	50.20	50.20	2.59	100
	100	50	10	36.50	3.64	100	35.90	33.54	1.77	100
	500	20	2	184.50	19.52	100	180.90	179.58	58.32	100
	500	50	2	184.00	22.05	100	182.50	181.84	54.48	100
	500	20	5	194.40	20.62	100	190.40	188.66	73.56	100
10	500	50	5	185.90	21.13	100	184.30	183.66	56.95	100
	500	20	10	251.00	42.46	100	251.00	250.9	89.32	100
	500	50	10	191.10	21.10	100	189.40	188.79	61.40	100
	1000	20	2	374.80	44.58	100	368.90	366.79	396.10	100
	1000	50	2	374.50	33.75	100	371.70	370.75	311.03	100
	1000	20	5	389.40	33.20	100	381.10	378.91	391.75	100
	1000	50	5	377.50	45.55	100	374.50	373.67	287.50	100
	1000	20	10	502.10	87.60	100	502.10	501.73	469.44	100
	1000	50	10	386.10	46.11	100	382.70	382.44	284.51	100

Table 6: Results of algorithms whose performances are maximized towards finding high-quality solutions on random sets for $|\Sigma| = 4$.

Instance				BASIC-BS				BS&VNS		
m	n	p	k	$\overline{\lvert s\rvert}$	$\bar{t}$[s]	feas[%]	$\lvert s^{\max}_{\text{best}}\rvert$	$\overline{\lvert s\rvert}$	$\bar{t}$[s]	feas[%]
	100	20	2	33.75	0.75	40	32.33	32.22	0.97	43
	100	50	2	32.14	0.22	70	32.70	32.20	0.91	99
	100	20	5	38.50	3.43	40	33.33	31.58	0.77	90
	100	50	5	33.20	2.46	50	33.00	32.94	1.04	79
	100	20	10	53.10	7.12	100	53.10	52.55	1.26	100
	100	50	10	35.60	4.80	50	34.67	34.37	0.91	30
	500	20	2	-	-	0	154.00	150.07	15.17	30
	500	50	2	175.67	30.04	60	167.00	164.61	19.50	100
2	500	20	5	-	-	0	176.00	171.60	17.09	10
	500	50	5	179.00	39.26	50	169.30	166.43	18.88	100
	500	20	10	263.83	58.61	60	263.43	256.31	17.63	70
	500	50	10	189.67	49.59	30	177.70	174.09	16.64	100
	1000	20	2	-	-	0	293.00	289.60	52.80	10
	1000	50	2	357.33	45.16	90	335.70	330.72	98.92	100
	1000	20	5	-	-	0	345.00	308.35	55.75	20
	1000	50	5	358	77.40	50	334.00	329.72	71.05	100
	1000	20	10	533	128.62	20	516.56	497.97	47.61	90
	1000	50	10	364.67	91.46	30	347.10	343.04	64.46	100
	100	20	2	14.50	2.32	40	10.50	10.41	0.43	80
	100	50	2	13.60	1.74	50	12.10	11.96	0.81	84
	100	20	5	25.10	4.71	100	23.40	23.25	0.59	100
	100	50	5	14.67	2.88	30	12.00	11.93	0.50	70
	100	20	10	50.00	7.67	100	50.00	49.62	1.44	100
	100	50	10	20.40	3.87	100	19.89	19.48	0.65	90
	500	20	2	-	-	0	-	-	-	0
	500	50	2	79.00	118.46	10	68.90	66.61	7.95	100
5	500	20	5	125.89	110.45	90	123.70	117.83	7.53	100
	500	50	5	-	-	0	68.78	67.27	8.07	90
	500	20	10	250.00	86.39	100	250.00	247.68	18.57	90
	500	50	10	102.80	89.27	100	99.86	97.50	12.72	70
	1000	20	2	-	-	0	113.00	99.95	9.19	20
	1000	50	2	-	-	0	139.90	137.06	36.15	100
	1000	20	5	251.10	202.33	100	242.50	231.50	19.61	100
	1000	50	5	-	-	0	135.10	132.71	33.43	100
	1000	20	10	500.00	215.65	100	497.33	478.23	54.15	90
	1000	50	10	206.00	193.75	60	194.67	188.72	41.49	60
	100	20	2	10.67	0.70	90	9.80	9.80	0.46	100
	100	50	2	8.33	0.22	30	7.40	6.91	0.85	89
	100	20	5	25.00	4.03	100	22.60	22.46	0.62	100
	100	50	5	10.60	0.71	100	9.56	9.56	0.45	90
	100	20	10	50.00	13.29	100	50.00	49.80	1.71	50
	100	50	10	20.00	2.50	100	18.00	17.88	0.66	100
	500	20	2	-	-	0	49.50	49.30	3.59	80
10	500	50	2	-	-	0	43.50	42.62	5.14	100
	500	20	5	125.00	246.70	100	124.60	116.42	12.60	100
	500	50	5	-	-	0	52.00	49.48	6.83	50
	500	20	10	250.00	145.06	100	250.00	247.26	27.41	80
	500	50	10	100.00	164.45	100	99.86	98.51	13.79	70
	1000	20	2	-	-	0	99.00	97.83	12.49	100
	1000	50	2	-	-	0	93.90	92.30	30.17	100
	1000	20	5	250.00	733.16	100	246.70	233.25	34.85	100
	1000	50	5	-	-	0	105.25	100.50	32.79	40
	1000	20	10	500.00	333.85	100	500.00	484.83	89.76	90
	1000	50	10	200.00	388.69	100	197.89	190.51	54.38	90

Table 7: Results of algorithms whose performances are maximized towards finding feasible solutions on random sets for $|\Sigma| = 20$.

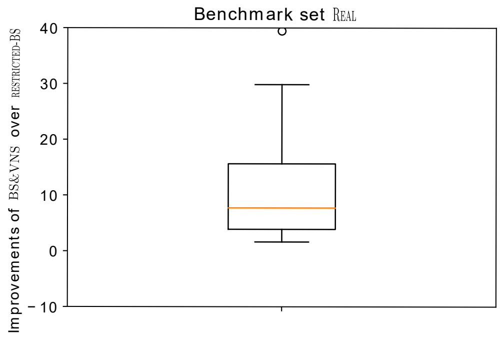


Figure 14: Finding a feasible solution on benchmark set REAL.

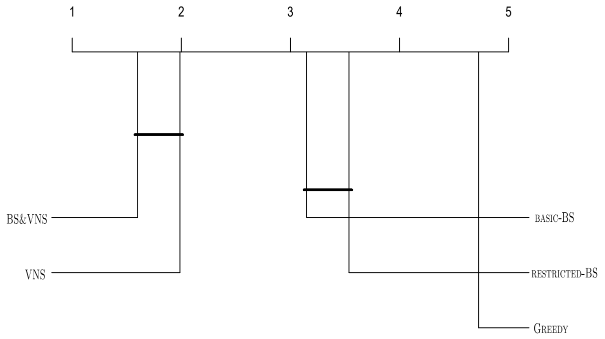


Figure 15: CD plots comparing solution quality: benchmark set REAL

4.6. Results of Methods with Focus on High-Quality Solutions on REAL *Instances*

In this section, we focus on finding high-quality solutions on benchmark set REAL. The results are shown in Table 8. The statistical differences w.r.t. the quality of obtained solutions is shown by the CD plot in Figure 15. Again, due to completeness, all five competitor algorithms are compared and drawn in the plot. The following conclusions may be drawn for this instance set.

- It is obvious that the BS&VNS hybrid works extremely well, in particular our VNS is able to improve the solution passed by RESTRICTED-BS; the average percentage improvement is about nine percent, see Figure 14. The results of BS&VNS clearly outperforms the results of BASIC-BS in 36 out of 40 cases. Overall, the observed differences are statistically significant, see Figure 15. Note that BASIC-BS could not deliver any feasible solution for five instances (the largest ones). For the largest instance ALL, which has 12681 input strings, VNS is the only algorithm able to produce a feasible solution.

- In overall, the results on benchmark set REAL indicate the high effectivity of VNS and its combination with BS. The reason why BS&VNS fully outperforms BASIC-BS (which was not usually the case on the RANDOM instances) lie in high similarity among the input sequences. It turns out that for VNS it gets easy to find local optima of reasonable quality, and, afterwards, by a carefully tuned amount of diversification, to jump from one local optima to another ones quickly to get new incumbents. Hereby, the implemented incremental evaluation plays a crucial role in exploitation, keeping runtimes reasonably low.

4.7. Analysis of the Results Obtained on the Biological Data

In this section we analyze in from a more biological perspective the results obtained on a class of bacterial 16S rRNA gene sequences, which is commonly used in a wide range of computational in silico experiments. This 16S rRNA gene is a core component of the 30S small subunit of prokaryotes and it is a marker and fingerprint for bacterial identification, i.e., its sequence differs among bacteria [36]. It is also used for phylogeny building because of its slow rate of evolution. Ten conserved (C) regions that are separated by variable (V) regions are contained in this molecule [29]. Each region contains a certain number of primers, which are short single-stranded nucleic acid sequences utilized by all living organisms in the

Instance	BASIC-BS		BS&VNS		
	$\overline{\lvert s\rvert}$	$\overline{t}$[s]	$\overline{\lvert s^{\max}\rvert}$	$\overline{\lvert s\rvert}$	$\overline{t}$[s]
Acidobacteria	-	-	**259**	253.70	1200.30
Actinobacteria	**406**	1039.95	388	381.10	1204.93
Aminicenantes	**1365**	16.65	1364	1364.00	1200.02
Aquificae	574	25.48	**652**	645.40	1200.09
Armatimonadetes	475	91.56	**516**	510.30	1200.21
Atribacteria	**1499**	5.27	**1499**	1499.00	1200.03
Bacteroidetes	375	822.18	**424**	416.90	1203.01
BRC1	635	36.67	**693**	682.50	1200.02
Candidatus-Saccharibacteria	561	30.19	**634**	622.70	1200.05
Chlamydiae	829	54.93	**840**	821.40	1200.03
Chlorobi	755	45.01	**852**	830.60	1200.04
Chloroflexi	351	34.55	**533**	495.10	1200.12
Chrysiogenetes	1139	58.36	**1267**	1266.20	1200.03
Cyano Chloroplast	457	143.52	**532**	524.90	1200.40
Deferribacteres	789	51.90	**802**	800.10	1200.06
Deinococcus-Thermus	640	109.36	**706**	654.30	1200.37
Dictyoglomi	**1522**	7.89	**1522**	1522.00	1200.02
Elusimicrobia	**1141**	67.34	**1141**	1141.00	1200.03
Fibrobacteres	1122	69.01	**1123**	1123.00	1200.03
Firmicutes	-	-	**122**	108.40	1201.21
Fusobacteria	547	61.61	**658**	651.20	1200.17
Ignavibacteriae	1353	24.69	**1354**	1354.00	1200.02
Latescibacteria	**768**	56.89	724	720.00	1200.03
Lentisphaerae	**1112**	69.93	1110	1108.30	1200.02
Microgenomates	432	25.44	**473**	456.50	1200.02
Nitrospinae	**1431**	5.31	**1431**	1431.00	1200.02
Nitrospirae	806	44.23	**809**	805.80	1200.03
Parcubacteria	426	34.96	**557**	538.70	1200.06
Planctomycetes	590	45.15	**648**	644.60	1200.06
Poribacteria	329	21.52	**441**	432.70	1200.04
Proteobacteria	-	-	**256**	206.90	1204.75
Spirochaetes	463	85.10	**557**	530.60	1200.21
SR1	980	97.43	**1026**	1022.90	1200.05
Synergistetes	634	18.23	**671**	643.10	1200.06
Tenericutes	501	174.55	**530**	526.80	1200.51
Thermodesulfobacteria	890	50.70	**1241**	1241.00	1200.04
Thermotogae	606	24.57	**744**	736.10	1200.23
Verrucomicrobia	-	-	**248**	245.90	1200.13
WPS-1	**1358**	11.53	1356	1356.00	1200.02
All	-	-	**107**	101.40	1209.39

Table 8: Results on instance set REAL for the algorithms whose performance is maximized towards finding best possible CLCS solutions.

initiation of DNA synthesis. In the mentioned article [29], continuous sequences, called *primer contigs*, are generated by the assembly of all of the primers reported for each conserved region. Such configuration represents a basis for further analysis of 16S rRNA sequences (see [29] for further information).

Established contigs were used for evaluation of the conservation degree of the conserved regions separating the hypervariable regions of the 16S rRNA gene in [16]. More precisely, for every contig and every gene from the considered databases, the longest common subsequence is calculated. The percentage of conservation of the considered contig in the considered gene was calculated by dividing the length of the longest common subsequence by the length of the contig. The obtained results show that the average conservation is not more than 60%.

In our research, we start from the assumption that each primer contig is contained (as a subsequence) in bacterial 16S rRNA gene sequences, and we seek the longest common subsequence which also contains all the primer contigs. Such a sequence should contain information of the common conserved parts of the starting sequences and could be useful for further research.

For the purpose of our tests, we adapted a set of 16S rRNA gene sequences and utilized a set of primer contigs from 16S rRNA gene as patterns generating a CLCS problem. Both types of sequences, 16S rRNA gene sequences and primer contigs sequences, contain some of the degenerate bases R, Y, M, K, S, W, H, D, B, V, and N. These degenerate bases can be replaced with corresponding amino acids [16]. For each of them exist two, three, or four possible amino acids. For example, M can be replaced with A or C. In our research, we used only one variant of these replacements. More precisely, we made the following replacements: R with A, Y with C, M with A, K with G, S with G, W with A, H with A, D with G, B with G, V with G and N with A. More about these replacements can be found in [16].

The first goal is to find the longest common subsequence that contains all the considered primer contigs for each phylum. The second and more challenging goal is to find the longest common subsequence also containing all the considered primer contigs, which include all sequences from the 12681 known bacterial 16S rRNA gene sequences.

The list of primer contigs used in this research and the systematized list of bacterial phyla together with the results obtained by the proposed methods, are given by Tables 3 and 4 in the document on supplementary material. The third column contains the total number of sequences in the considered phylum. For the sake of deeper insight into the obtained results, we included the next three columns containing the information about the minimum, average, and maximum length of a sequence in the phylum, respectively. The last two columns contain the best result obtained among all the proposed methods (the length of the CLCS $|s|$, containing all contig patterns as subsequences) and the percentage calculated as a ratio between the length of the CLCS and length of the shortest sequence in the considered phylum.

From Table 4 of the document on supplementary material several conclusions can be drawn about the considered sequences and the conserved parts of the sequence in different bacterial phyla. If a phylum contains a small number of bacterial sequences (see the lines numbered by 3, 6, 17, 22, 26 and 39), the length of the CLCS is more than 90% of the minimal length in the phylum. This high percentage can be simply explained by the fact that sequences of the bacteria belonging to the same phylum are very similar. The largest set is Proteobacteria (line #31), containing 4543 sequences, with the shortest sequence of 471 nucleotides. The CLCS for this phylum is rather long (324 nucleotides), which is about 68% of the length of the shortest sequence. This is not the case with other phyla containing many sequences, (see for example lines 2, 7, and 20), where we see that the length of the CLCS is rather small (less than 40%). The last row contains information about the length of the CLCS obtained under the entire set of 12681 sequences. From the feasibility of the result, one can conclude that all 12681 bacterial 16S rRNA gene sequences in the dataset indeed contain each primer contig as a subsequence. This result can be also considered as a specific success of the proposed methods, since the starting set of sequences is very large. Finally, this fact also confirms that there is a high level of similarity among all the sequences in the set, since the best reported result of the CLCS has length 113, which is about 35% of the length of the shortest sequence.

5. Conclusions and Future Work

We presented several heuristic approaches to solve the constrained longest common subsequence problem with arbitrary many input strings as well as arbitrary many pattern strings ((m,k)–CLCS). In contrast to many other variants of the LCS problem, this problem is particularly challenging since already the decision problem of finding any feasible string is $\mathcal{NP}$–hard, as formally proved here, and can also be challenging in practice. With this aspect in mind, we first designed methods with the particular focus of finding any feasible solution, disregarding the objective of finding a longest possible string to a large extent. Our restricted BS variant excelled in this respect, in particular, on randomly created benchmark instance set. On the contrary, when also considering the string length and aiming for high quality solutions, the basic BS approach, whose search is guided by a specially designed probability-based heuristic, works best for smaller alphabet size (up to four). For larger alphabets (of 20), the VNS turned out to be a better choice, especially when applied in combination with the restricted BS for obtaining an initial feasible solution. The incremental evaluation of solutions in the local search neighborhoods plays a particularly important role to achieve the high effectiveness in the VNS. On the considered real-world benchmark sets we observed significantly different results, primarily because the input strings are strongly related. Here, the hybrid VNS is the only method which could find feasible solutions in all cases. Also, best solutions are delivered by the proposed VNS in combination with the restricted BS.

In future work it makes sense to concentrate on developing more sophisticated methods to find first feasible solutions for those instances where the proposed methods have not been so successful. This is especially the case for our tight random instances on a larger alphabet and rather short input strings in comparison to the pattern string lengths. In order to possibly further improve the BS with focus on solution quality, considering alternative search guidance functions is also a promising next step. For example, one may consider an approximate expected length calculation in the spirit as it was already shown to be successful for some other LCS problem variants [9]. Last but not least, considering anytime variants of A* search [10, 11] would also be interesting for obtaining heuristic solutions in combination with dual bounds on larger instances where the classical A* search cannot be applied anymore and the complete search must be stopped early. Also, studying a complementary problem of the (m,k)–CLCS problem, so called the restricted LCS problem [14] where we aim at finding an LCS between a set of input strings that does not contain any string from a given set of restriction strings as its subsequences, is a promising research direction to consider.

Acknowledgments

This research was partially supported by Ministry for Scientific and Technological Development, Higher Education and Information Society, Government of Republic of Srpska, B&H under the Project "Development of artificial intelligence methods for solving computer biology problems", project no. 19.032/-961-24/19. The first author was funded by the Doctoral Program Vienna Graduate School on Computational Optimization (VGSCO), Austrian Science Foundation, project no. W1260-N35. Christian Blum was funded by project CI-SUSTAIN of the Spanish Ministry of Science and Innovation (PID2019-104156GB-I00).

References

[1] S. S. Adi, M. D. Braga, C. G. Fernandes, C. E. Ferreira, F. V. Martinez, M.-F. Sagot, M. A. Stefanes, C. Tjandraatmadja, and Y. Wakabayashi. Repetition-free longest common subsequence. *Discrete Applied Mathematics*, 158(12):1315–1324, 2010.

[2] G. Blin, L. Bulteau, M. Jiang, P. J. Tejada, and S. Vialette. Hardness of longest common subsequence for sequences with bounded run-lengths. In *Proceedings of CPM 2012 – the 21st Annual Symposium on Combinatorial Pattern Matching*, volume 7354 of *Lecture Notes in Computer Science*, pages 138–148, Berlin, Heidelberg, 2012. Springer.

[3] C. Blum, M. J. Blesa, and M. López-Ibáñez. Beam search for the longest common subsequence problem. *Computers and Operations Research*, 36(12):3178–3186, 2009.

[4] S. R. Chowdhury, M. Hasan, S. Iqbal, and M. S. Rahman. Computing a longest common palindromic subsequence. *Fundamenta Informaticae*, 129(4):329–340, 2014.

[5] S. Deorowicz. Bit-parallel algorithm for the constrained longest common subsequence problem. *Fundamenta Informaticae*, 99(4):409–433, 2010.

[6] S. Deorowicz and J. Obstój. Constrained longest common subsequence computing algorithms in practice. *Computing and Informatics*, 29(3):427–445, 2012.

[7] M. Djukanovic, C. Berger, G. R. Raidl, and C. Blum. On solving a generalized constrained longest common subsequence problem. In *Proceedings of OPTIMA 20 – XI International Conference Optimization and Applications*, volume 12422 of *LNCS*, pages 55–70. Springer, 2020.

[8] M. Djukanovic, C. Berger, G. R. Raidl, and C. Blum. An A* search algorithm for the constrained longest common subsequence problem. *Information Processing Letters*, page 106041, 2021.

[9] M. Djukanovic, G. R. Raidl, and C. Blum. A beam search for the longest common subsequence problem guided by a novel approximate expected length calculation. In G. Nicosia, P. Pardalos, R. Umeton, G. Giuffrida, and V. Sciacca, editors, *Machine Learning, Optimization, and Data Science*, pages 154–167, Cham, 2019. Springer International Publishing.

[10] M. Djukanovic, G. R. Raidl, and C. Blum. Anytime algorithms for the longest common palindromic subsequence problem. *Computers & Operations Research*, 114:104827, 2020.

[11] M. Djukanovic, G. R. Raidl, and C. Blum. Finding longest common subsequences: New anytime A* search results. *Applied Soft Computing*, 95:106499, 2020.

[12] R. Dondi. The constrained shortest common supersequence problem. *Journal of Discrete Algorithms*, 21:11 – 17, 2013.

[13] E. Farhana and M. S. Rahman. Constrained sequence analysis algorithms in computational biology. *Information Sciences*, 295:247–257, 2015.

[14] Z. Gotthilf, D. Hermelin, G. M. Landau, and M. Lewenstein. Restricted LCS. In *Proceedings of SPIRE 2010 – the 9th International Symposium on String Processing and Information Retrieval*, pages 250–257. Springer, 2010.

[15] Z. Gotthilf, D. Hermelin, and M. Lewenstein. Constrained lcs: hardness and approximation. In *Proceedings of CPM2008 – the 17th Annual Symposium on Combinatorial Pattern Matching*, pages 255–262. Springer, 2008.

[16] O. Gursoy and M. Can. On the accuracy of the 16S-rRNA gene conserved regions. *Southeast Europe Journal of Soft Computing*, 8(1), 2019.

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

[18] P. Hansen, N. Mladenović, and J. A. M. Pérez. Variable neighbourhood search: methods and applications. *4OR*, 6(4):319–360, 2008.

[19] P. Hansen, N. Mladenović, and J. A. M. Pérez. Variable neighbourhood search: methods and applications. *Annals of Operations Research*, 175(1):367–407, 2010.

[20] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[21] K. Huang, C. Yang, and K. Tseng. Fast algorithms for finding the common subsequences of multiple sequences. In *Proceedings of ICS 2004 – the 3rd IEEE International Computer Symposium*, pages 1006–1011, 2004.

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

[23] Y. Li, Y. Wang, Z. Zhang, Y. Wang, D. Ma, and J. Huang. A novel fast and memory efficient parallel MLCS algorithm for long and large-scale sequences alignments. In *Proceedings of ICDE2019 – the 32nd International Conference on Data Engineering*, pages 1170–1181, 2016.

[24] J. Liu, Y. Wang, and Y. Chiu. Constrained longest common subsequences with run-length-encoded strings. *The Computer Journal*, 58(5):1074–1084, 2015.

[25] W. Liu and L. Chen. A fast longest common subsequence algorithm for biosequences alignment. In D. Li, editor, *Proceedings of CCTA 2007 – the 1st Computer And Computing Technologies In Agriculture, Volume I*, pages 61–69, Boston, MA, 2008. Springer US.

[26] M. López-Ibáñez, L. P. Cáceres, J. Dubois-Lacoste, T. Stützle, and M. Birattari. The irace package: User guide. In *Technical Report TR/IRIDIA/2016-004*. IRIDIA, Université Libre de Bruxelles, Belgium, 2016.

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

[28] K. Mangal and R. Kumar. A recursive algorithm for generalized constraint SCS problem. *National Academy Science Letters*, 39(4):273–276, 2016.

[29] M. Martínez-Porchas and F. Vargas-Albores. An efficient strategy using k-mers to analyse 16s rRNA sequences. *Heliyon*, 3(7):e00370, 2017.

[30] N. Mladenović and P. Hansen. Variable neighborhood search. *Computers & Operations Research*, 24(11):1097 – 1100, 1997.

[31] D. W. Mount. Sequence and genome analysis. *Bioinformatics: Cold Spring Harbour Laboratory Press: Cold Spring Harbour*, 2, 2004.

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

[33] P. S. Ow and T. E. Morton. Filtered beam search in scheduling. *The International Journal of Production Research*, 26(1):35–62, 1988.

[34] T. Pohlert. The pairwise multiple comparison of mean ranks package (pmcmr). *R package*, 27(2019):9, 2014.

[35] P. D. Schloss, S. L. Westcott, T. Ryabin, J. R. Hall, M. Hartmann, E. B. Hollister, R. A. Lesniewski, B. B. Oakley, D. H. Parks, C. J. Robinson, J. W. Sahl, B. Stres, G. G. Thallinger, D. J. Van Horn, and C. F. Weber. Introducing mothur: Open-source, platform-independent, community-supported software for describing and comparing microbial communities. *Applied and Environmental Microbiology*, 75(23):7537–7541, 2009.

[36] E. Stackebrandt and B. M. Goebel. Taxonomic note: a place for DNA-DNA reassociation and 16S rRNA sequence analysis in the present species definition in bacteriology. *International Journal of Systematic and Evolutionary Microbiology*, 44(4):846–849, 1994.

[37] C. Y. Tang, C. L. Lu, M. D.-T. Chang, Y.-T. Tsai, Y.-J. Sun, K.-M. Chao, J.-M. Chang, Y.-H. Chiou, C.-M. Wu, H.-T. Chang, et al. Constrained multiple sequence alignment tool development and its application to rnase family alignment. *Journal of Bioinformatics and Computational Biology*, 1(02):267–287, 2003.

[38] Y.-T. Tsai. The constrained longest common subsequence problem. *Information Processing Letters*, 88(4):173–176, 2003.

[39] 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.