Construct, Merge, Solve and Adapt: Application to the Repetition-Free Longest Common Subsequence Problem

Christian Blum[1,2](✉) and Maria J. Blesa[3]

[1] Department of Computer Science and Artificial Intelligence,
University of the Basque Country, San Sebastian, Spain
christian.c.blum@gmail.com
[2] IKERBASQUE, Basque Foundation for Science, Bilbao, Spain
[3] ALBCOM Research Group, Computer Science Department,
Universitat Politécnica de Catalunya - BarcelonaTech, Barcelona, Spain
mjblesa@cs.upc.edu

Abstract. In this paper we present the application of a recently proposed, general, algorithm for combinatorial optimization to the repetition-free longest common subsequence problem. The applied algorithm, which is labelled Construct, Merge, Solve & Adapt, generates sub-instances based on merging the solution components found in randomly constructed solutions. These sub-instances are subsequently solved by means of an exact solver. Moreover, the considered sub-instances are dynamically changing due to adding new solution components at each iteration, and removing existing solution components on the basis of indicators about their usefulness. The results of applying this algorithm to the repetition-free longest common subsequence problem show that the algorithm generally outperforms competing approaches from the literature. Moreover, they show that the algorithm is competitive with CPLEX for small and medium size problem instances, whereas it outperforms CPLEX for larger problem instances.

Keywords: Hybrid algorithm · Combining metaheuristics with ILP solvers · Repetition-free longest common subsequence problem

1 Introduction

The problem that is often encountered when applying exact solvers to combinatorial optimization problems is that they are not applicable to problem instances

This work was supported by project TIN2012-37930-C02-02 (Spanish Ministry for Economy and Competitiveness, FEDER funds from the European Union) and project SGR 2014-1034 (AGAUR, Generalitat de Catalunya). Additionally, Christian Blum acknowledges support from IKERBASQUE. Our experiments have been executed in the High Performance Computing environment managed by RDlab (http://rdlab.cs.upc.edu) and we would like to thank them for their support.

F. Chicano et al. (Eds.): EvoCOP 2016, LNCS 9595, pp. 46–57, 2016.
DOI: 10.1007/978-3-319-30698-8_4

of realistic sizes. However, for smaller problem instances, exact solvers are often surprisingly efficient. This is because the operations research community has invested a lot of time, effort and expertise into the development of exact solvers. Prime examples are general-purpose mathematical programming solvers such as CPLEX and Gurobi. Therefore, recent research efforts have focused on ways in which efficient exact solvers can be used within heuristic frameworks even in the context of large problem instances. One of the most recent examples of these research efforts is an algorithm labelled Construct, Merge, Solve & Adapt (CMSA) [1,2]. This algorithm works as follows. At each iteration, solutions to the tackled problem instance are generated in a probabilistic way. The solution components found in these solutions are then added to a sub-instance of the original problem instance. Subsequently, an exact solver such as, for example, CPLEX is used to solve the sub-instance to optimality. Moreover, the algorithm is equipped with a mechanism for deleting seemingly useless solution components from the sub-instance. This is done such that the sub-instance has a moderate size and can be solved rather quickly to optimality.

In this work we apply the CMSA algorithm to the so-called repetition-free longest common subsequence problem [3]. This problem, which is NP-hard, is a special case of the well-known longest common subsequence problem. The repetition-free longest common subsequence problem seems to be well-suited for being solved with CMSA, because the standard integer linear programming (ILP) model for the problem can only be solved to optimality in the context of rather small problem instance. Both the number of variables and constraints in this model (which is outlined later in Sect. 2) are exponential in the input parameters of the problem. The obtained results show that, indeed, the application of CMSA obtains state-of-the-art results, especially in the context of large problem instances.

The remaining part of the paper is organized as follows. In Sect. 2 we provide a technical description of the repetition-free longest common subsequence problem. Moreover, we describe the standard ILP model for this problem. Next, in Sect. 3, the application of CMSA to the tackled problem is outlined. Finally, Sect. 4 provides an extensive experimental evaluation and Sect. 5 offers a discussion and an outlook to future work.

2 Repetition-Free Longest Common Subsequence Problem

The longest common subsequence (LCS) problem is a string problem with numerous applications, for example, in computational biology [4–6]. A problem instance (S, Σ) consists of a set $S = \{s_1, s_2, \ldots, s_n\}$ of n input strings over a finite alphabet Σ. The goal consists in finding the longest possible subsequence of all strings in S. A string t is a subsequence of a string s, if t can be produced from s by deleting zero or more characters. For example, *dga* can be produced from *adagtta* by deleting the first two occurrences of letter a and the two occurrences of letter t. Apart from applications in computational biology, the LCS problem finds applications, for example, in data compression and file comparison [7,8]. Moreover, note that the LCS problem was shown to be NP-hard [9] for an arbitrary number n of input strings.

In this work we consider a restricted version of the LCS problem, the so-called repetition-free longest common subsequence (RFLCS) problem. Given exactly two input strings x and y over a finite alphabet Σ, the goal is to find a longest common subsequence with the additional restriction that no letter may appear more than once. This problem was introduced in [3] as a comparison measure for two sequences of biological origin. In the same paper, the authors proposed three heuristics for solving this problem. Other algorithms from the literature for solving the RFLCS problem include a Beam-ACO approach [10] and an evolutionary algorithm [11]. Among these techniques, the Beam-ACO approach can be regarded as the current state-of-the-art method.

The RFLCS problem can be stated in terms of an integer linear program (ILP) in the following way. First, let us denote the length of x by l_x and the length of y by l_y. Furthermore, the positions in x and y are numbered from 1 to l_x, respectively from 1 to l_y. The letter at position i of x is denoted by $x[i]$, and the letter at position j of y is denoted by $y[j]$. The set Z of binary variables that is required for the ILP model is composed as follows. For each combination of $i = 1, \ldots, l_x$ and $j = 1, \ldots, l_y$ such that $x[i] = y[j]$, set Z contains a binary variable $z_{i,j}$. Moreover, we say that two variables $z_{i,j}$ and $z_{k,l}$ are *in conflict*, if and only if either $i < k$ and $j > l$ or $i > k$ and $j < l$. Finally, for each letter $a \in \Sigma$, set $Z_a \subset Z$ contains all variables $z_{i,j}$ such that $x[i] = y[j] = a$. The RFLCS problem can then be rephrased as the problem of selecting a maximal number of non-conflicting variables from Z provided that, among all variables representing a letter $a \in \Sigma$, at most one variable is chosen. Given these notations, the ILP is stated as follows.

$$\mathbf{max} \sum_{z_{i,j} \in Z} z_{i,j} \tag{1}$$

subject to:

$$\sum_{z_{i,j} \in Z_a} z_{i,j} \leq 1 \quad \text{for } a \in \Sigma \tag{2}$$

$$z_{i,j} + z_{k,l} \leq 1 \quad \text{for all } z_{i,j} \text{ and } z_{k,l} \text{ being in conflict} \tag{3}$$

$$z_{i,j} \in \{0, 1\} \quad \text{for } z_{i,j} \in Z \tag{4}$$

Hereby, constraints (2) ensure that each letter from the alphabet is chosen at most once, and constraints (3) ensure that selected variables are not in conflict.

3 Application of CMSA to the RFLCS Problem

The application of CMSA to the RFLCS problem is pseudo-coded in Algorithm 1. Note that, in the context of this algorithm, solutions to the problem and sub-instances are both subsets of the complete set Z of variables. If a solution S

Algorithm 1. CMSA for the RFLCS problem

```
 1: input: strings $x$ and $y$ over alphabet $\Sigma$, values for parameters $n_a$ and $\mathrm{age}_{\max}$
 2: $S_{\mathrm{bsf}} :=$ NULL, $Z_{\mathrm{sub}} := \emptyset$
 3: $\mathrm{age}[z_{i,j}] := 0$ for all $z_{i,j} \in Z$
 4: while CPU time limit not reached do
 5:    for $i = 1, \ldots, n_a$ do
 6:       $S :=$ ProbabilisticSolutionConstruction($Z$)
 7:       for all $z_{i,j} \in S$ and $z_{i,j} \notin Z_{\mathrm{sub}}$ do
 8:          $\mathrm{age}[z_{i,j}] := 0$
 9:          $Z_{\mathrm{sub}} := Z_{\mathrm{sub}} \cup \{z_{i,j}\}$
10:       end for
11:    end for
12:    $S'_{\mathrm{opt}} :=$ ApplyILPSolver($Z_{\mathrm{sub}}$)
13:    if $|S'_{\mathrm{opt}}| > |S_{\mathrm{bsf}}|$ then $S_{\mathrm{bsf}} := S'_{\mathrm{opt}}$
14:    Adapt($Z_{\mathrm{sub}}$, $S'_{\mathrm{opt}}$, $\mathrm{age}_{\max}$)
15: end while
16: output: $S_{\mathrm{bsf}}$
```

contains a variable $z_{i,j}$, this means that this variable must be given value one in order to produce the corresponding solution. The main loop of CMSA is executed while the CPU time limit is not reached. It consists of the following actions. First, the best-so-far solution S_{bsf} is initialized to NULL, and the restricted problem instance (Z_{sub}) to the empty set. Then, at each iteration a number of n_a solutions is probabilistically constructed in function ProbabilisticSolutionConstruction(Z) in line 6 of Algorithm 1. The variables contained in these solutions are added to Z_{sub}. The age of a newly added variable $z_{i,j}$ ($\mathrm{age}[z_{i,j}]$) is set to 0. After the construction of n_a solutions, an ILP solver is applied to find the best solution S'_{opt} in the sub-instance Z_{sub} (see function ApplyILPSolver(Z_{sub}) in line 12 of Algorithm 1). In case S'_{opt} is better than the current best-so-far solution S_{bsf}, solution S'_{opt} is stored as the new best-so-far solution (line 13). Next, sub-instance Z_{sub} is adapted, based on solution S'_{opt} and on the age values of the variables. This is done in function Adapt(Z_{sub}, S'_{opt}, $\mathrm{age}_{\max}$) in line 14 as follows. First, the age of each variable in Z_{sub} is increased by one, and, subsequently, the age of each variable in $S'_{\mathrm{opt}} \subseteq Z_{\mathrm{sub}}$ is re-initialized to zero. Finally, those solution components from Z_{sub} whose age has reached the maximum component age ($\mathrm{age}_{\max}$) are deleted from Z_{sub}. The motivation behind the aging mechanism is that variables which never appear in an optimal solution of Z_{sub} should be removed from Z_{sub} after a while, because they simply slow down the ILP solver. On the other side, components which appear in optimal solutions seem to be useful and should therefore remain in Z_{sub}.

In the following we will describe in detail the remaining component of the algorithm: the probabilistic construction of solutions in function ProbabilisticSolutionConstruction(Z). Such a solution construction starts with an empty solution $S = \emptyset$, and the first step consists in generating the set of variables that serve as options to be added to S. More specifically, the initial set C is generated in order to contain for each letter $a \in \Sigma$ the variable $z_{i,j} \in Z_a$ (if any) such that $i < k$ and $j < l$, $\forall z_{k,l} \in Z_a$. Moreover, options $z_{i,j} \in C$ are given a weight value $w(z_{i,j}) := \frac{i}{l_x} + \frac{j}{l_y}$, which is a known greedy function for longest common subsequence problems. At each construction step, exactly one variable is chosen from C and added to S. For doing so, first, a value r is chosen uniformly at random from $[0, 1]$. In case $r \leq d_{\text{rate}}$, where d_{rate} is a parameter of the algorithm, the variable $z_{i,j} \in C$ with the smallest weight value is deterministically chosen. Otherwise, a candidate list $L \subseteq C$ of size $\min\{l_{\text{size}}, |C|\}$ containing the options with the lowest weight values is generated and exactly one variable $z_{i,j} \in L$ is then chosen uniformly at random and added to S. Note that l_{size} is another parameter of the solution construction process. Finally, the set of options C for the next construction step is generated. This is done such that C only contains variables that represent letters that are not already represented by one of the variables in S. Moreover, being $z_{i,j}$ the last variable that was added to S, C contains for each non-represented letter $a \in \Sigma$ the variable $z_{r,s} \in Z_a$ (if any) with the lowest weight value $w(z_{r,s})$ calculated as $w(z_{r,s}) := \frac{r-i}{l_x-i} + \frac{s-j}{l_y-j}$. The construction of a complete (valid) solution is finished when the set of options is empty.

4 Experimental Evaluation

We implemented the proposed algorithm in ANSI C++ using GCC 4.7.3, without the use of any external libraries. The ILP models, both the ones of the original RFLCS instances and the ones of sub-instances within CMSA, were solved with IBM ILOG CPLEX v12.1 in one-threaded mode. The experimental evaluation has been performed on a cluster of PCs with Intel(R) Xeon(R) CPU 5670 CPUs of 12 nuclei of 2933 MHz and at least 40 Gigabytes of RAM. In the following we first describe the set of benchmark instances that we generated to test the CMSA algorithm. Then, we describe the tuning experiments that were performed in order to determine a proper setting for the parameters. Finally, an exhaustive experimental evaluation is presented.

4.1 Problem Instances

Two sets of problem instances were adopted from [10]. These sets were generated with the same procedure as described in [3]. The first set (henceforth called Set1) consists for each combination of input sequence length $n \in \{32, 64, 128, 256, 512\}$ and alphabet size $|\Sigma| \in \{n/8, n/4, 3n/8, n/2, 5n/8, 3n/4, 7n/8\}$ of exactly 10 problem instances. The second set of instances (henceforth called Set2) is generated on the basis of alphabet sizes $|\Sigma| \in \{4, 8, 16, 32, 64\}$ and the maximal repetition of each letter $rep \in \{3, 4, 5, 6, 7, 8\}$ in each input string. For each combination of $|\Sigma|$ and rep this instance set consists of 10 randomly generated problem instances. In addition, we generated an extension of Set2 consisting of larger problem instances. More specifically, we generated for each combination of $|\Sigma| \in \{128, 256\}$ and $rep \in \{3, 4, 5, 6, 7, 8\}$ ten problem instances. All the results to be shown in the forthcoming sections are averages over the 10 problem instances of each type.

4.2 Tuning of CMSA

There are several parameters involved in Cmsa for which well-working values must be found: (n_a) the number of solution constructions per iteration, ($\text{age}_{\max}$) the maximum allowed age of solution components, (d_{rate}) the determinism rate, (l_{size}) the candidate list size, and ($t_{\max}$) the maximum time in seconds allowed for CPLEX per application to a sub-instance. The last parameter is necessary, because even when applied to reduced problem instances, CPLEX might still need too much computation time for solving such sub-instances to optimality. In any case, CPLEX always returns the best feasible solution found within the given computation time.

We decided to make use of the automatic configuration tool irace[12] for the tuning of the five parameters. In fact, irace was applied to tune Cmsa separately for each alphabet size, which—after initial experiments—seems to have more influence on the behavior of the algorithm than the length of the input strings. In the context of Set1 we randomly generated two tuning instances for each combination of string length and alphabet size, whereas for Set2 (and its extension) we randomly generated two tuning instances for each combination of alphabet size and number of repetitions.

The tuning process for each alphabet size was given a budget of 200 runs of Cmsa, where each run was given a computation time limit of $l_x/10$ CPU seconds for instances of Set1 (remember that for instances of Set1 it holds that $l_x = l_y$) and $(|\Sigma| * reps)/10$ CPU seconds for instances of Set2 and its extension. Finally, the following parameter value ranges were chosen concerning the five parameters of Cmsa:

- $n_a \in \{10, 30, 50\}$
- $\text{age}_{\max} \in \{1, 5, 10, \mathit{inf}\}$, where *inf* means that solution components are never removed from Z_{sub}.
- $d_{\text{rate}} \in \{0.0, 0.3, 0.5, 0.7, 0.9\}$, where a value of 0.0 means that the selection of the next variable to be added to the partial solution under construction is always done randomly from the candidate list, while a value of 0.9 means that solution constructions are nearly deterministic.
- $l_{\text{size}} \in \{3, 5, 10\}$
- $t_{\max} \in \{0.5, 1.0, 5.0\}$ (in seconds) for instances of Set1 and Set2, and $t_{\max} \in \{1.0, 10.0, 100.0\}$ for the instances of the extension of Set2.

The tuning runs with iraceproduced the configurations of Cmsa as shown in Table 1.

4.3 Experimental Results

Three algorithms were included in the comparison. Beam-ACO is currently a state-of-the-art method for the RFLCS problem [10], CPLEX refers to the application of CPLEX to the complete problem instances, and CMSA is the algorithm

Table 1. Results of tuning CMSA with irace.

(a) Tuning results for the seven alphabet sizes of Set1.

$\lvert\Sigma\rvert$	n_a	$\text{age}_{\max}$	d_{rate}	l_{size}	$t_{\max}$
$n/8$	30	5	0.7	5	0.5
$n/4$	10	1	0.7	10	1.0
$3n/8$	30	1	0.3	10	5.0
$n/2$	50	1	0.3	3	5.0
$5n/8$	30	5	0.7	10	5.0
$3n/4$	30	5	0.5	5	5.0
$7n/8$	30	5	0.0	10	5.0

(b) Tuning results for the seven alphabet sizes of Set2 and its extension.

$\lvert\Sigma\rvert$	n_a	$\text{age}_{\max}$	d_{rate}	l_{size}	$t_{\max}$
4	10	*inf*	0.9	10	1.0
8	10	10	0.9	5	0.5
16	50	*inf*	0.7	3	0.5
32	50	*inf*	0.5	10	5.0
64	10	10	0.5	5	1.0
128	30	1	0.5	5	10.0
256	10	1	0.7	3	10.0

proposed in this work. The results of Beam-ACO for the instances of Set1 and Set2 were taken from [10], where Beam-ACO was applied once to each problem instance with a computation time limit of 5 CPU seconds per run, a beam width of 10, and a determinism rate of 0.9. Note that the low computation time limit of 5 CPU seconds was adopted in [10], because Beam-ACO always produced its best results during the first seconds of a run. For the application to the larger problem instances that were generated as an extension of Set2 Beam-ACO was applied with the same parameter values for beam with and determinism rate, but with the same computation time limit as CMSA. In particular, CMSA was applied to each problem instance with a computation time limit of $l_x/10$ CPU seconds for instances of Set1 (remember that for instances of Set1 it holds that $l_x = l_y$) and $(|\Sigma| * reps)/10$ CPU seconds for instances of Set2 and its extension. The stand-alone application of CPLEX to each problem instances was given more computation time, namely, 600 CPU seconds for each run, regardless of the instance/alphabet size. Moreover, a memory limit of 2 GB were used for each application of CPLEX.

The numerical results are presented in Table 2 concerning Set1, Table 3 concerning Set2, and Table 4 concerning the extension of Set2. Each table row presents the results averaged over 10 problem instances of the same type. The results of Beam-ACO and CMSA are provided in two columns each. The first one (with heading **result**) provides the result of the corresponding algorithm averaged over 10 problem instances, while the second column (with heading **time (s)**) provides the average computation time necessary for finding the corresponding solutions. The same information is given for CPLEX. However, in this case we also provide the average optimality gaps (in percent), that is, the average gaps between the upper bounds and the values of the best solutions when stopping a run.

The results allow to make the following observations:

– First of all, CPLEX is able to provide optimal solutions for all instances of 29 out of 35 instance types (that is, table rows) concerning Set1, and for all instances of 27 out of 30 instance types concerning Set2. This means, on one side, that the instances of these two benchmark sets are, in their majority,

Table 2. Experimental results concerning the instances of Set1.

$\|\Sigma\|$	n	Beam-ACO		CPLEX			CMSA	
		result	**time (s)**	**result**	**time (s)**	**gap (%)**	**result**	**time (s)**
$n/8$	32	4.0	< 0.1	4.0	0.1	0.0	4.0	< 0.1
	64	8.0	< 0.1	8.0	0.8	0.0	8.0	< 0.1
	128	16.0	< 0.1	16.0	9.6	0.0	16.0	< 0.1
	256	31.9	< 0.1	n.a.	n.a.	n.a.	31.8	0.1
	512	62.3	1.8	n.a.	n.a.	n.a.	60.4	13.3
$n/4$	32	7.9	< 0.1	7.9	0.1	0.0	7.9	< 0.1
	64	14.3	< 0.1	14.4	0.3	0.0	14.4	< 0.1
	128	25.3	0.2	25.9	17.2	0.0	25.7	1.1
	256	42.4	0.7	41.6	495.1	8.6	42.6	3.6
	512	68.0	0.8	n.a.	n.a.	n.a.	68.7	7.1
$3n/8$	32	8.7	< 0.1	9.0	< 0.1	0.0	9.0	< 0.1
	64	14.4	< 0.1	14.8	0.2	0.0	14.8	< 0.1
	128	25.1	< 0.1	25.3	3.1	0.0	25.3	< 0.1
	256	39.7	0.2	40.1	133.5	0.0	40.1	1.5
	512	59.4	1.3	7.0	36.2	> 100.0	59.5	3.2
$n/2$	32	8.8	< 0.1	8.8	< 0.1	0.0	8.8	< 0.1
	64	14.5	< 0.1	14.6	0.1	0.0	14.6	< 0.1
	128	23.4	< 0.1	23.4	1.0	0.0	23.3	< 0.1
	256	34.1	0.2	34.3	30.5	0.0	34.1	0.3
	512	53.1	0.6	14.9	207.5	> 100.0	53.1	5.9
$5n/8$	32	7.9	< 0.1	7.9	< 0.1	0.0	7.9	< 0.1
	64	13.7	< 0.1	13.7	< 0.1	0.0	13.7	< 0.1
	128	21.1	< 0.1	21.1	0.5	0.0	21.1	< 0.1
	256	31.1	0.2	31.2	10.4	0.0	31.2	1.6
	512	47.8	0.3	47.9	308.3	0.0	47.8	2.9
$3n/4$	32	7.8	< 0.1	7.8	< 0.1	0.0	7.8	< 0.1
	64	13.1	< 0.1	13.3	< 0.1	0.0	13.3	< 0.1
	128	19.1	< 0.1	19.1	0.2	0.0	19.1	< 0.1
	256	30.0	< 0.1	30.1	4.3	0.0	30.1	1.3
	512	44.7	0.5	44.8	115.5	0.0	44.8	1.3
$7n/8$	32	7.6	< 0.1	7.6	< 0.1	0.0	7.6	< 0.1
	64	12.2	< 0.1	12.2	< 0.1	0.0	12.2	< 0.1
	128	18.5	< 0.1	18.5	0.2	0.0	18.5	< 0.1
	256	27.2	< 0.1	27.2	2.2	0.0	27.1	0.1
	512	40.7	0.3	40.9	59.4	0.0	40.8	4.3

not very difficult to be solved. On the other side, there seems to be a kind of phase transition between instances that can be solved to optimality quite easily, and instances that are difficult to be solved. In three out of six instance types of Set1 which CPLEX cannot solve to optimality within the allocated CPU time, the allocated memory is not sufficient, and for other two instance types the average optimality gap is greater than 100 %.

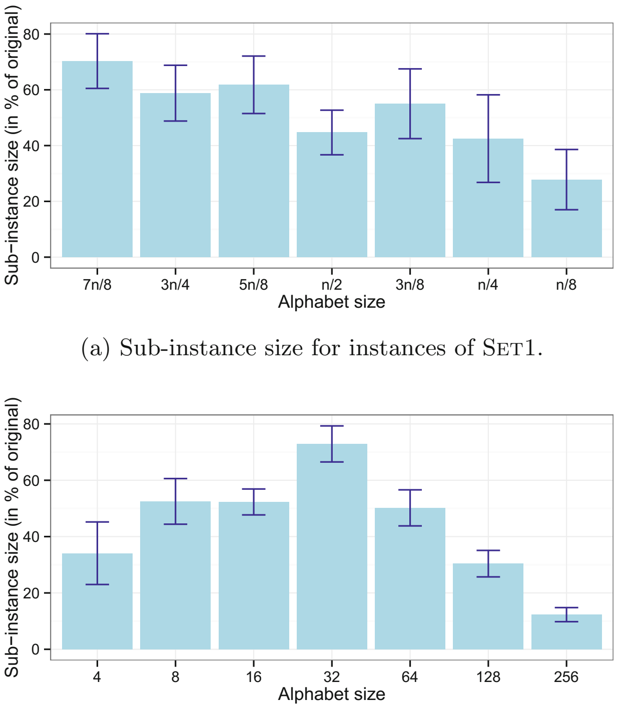
Table 3. Experimental results concerning the instances of Set2.

$\|\Sigma\|$	n	Beam-ACO		CPLEX			CMSA	
		result	**time (s)**	**result**	**time (s)**	**gap (%)**	**result**	**time (s)**
4	3	3.4	< 0.1	3.4	< 0.1	0.0	3.4	< 0.1
	4	3.8	< 0.1	3.8	< 0.1	0.0	3.8	< 0.1
	5	3.8	< 0.1	3.8	< 0.1	0.0	3.8	< 0.1
	6	3.8	< 0.1	3.8	< 0.1	0.0	3.8	< 0.1
	7	3.9	< 0.1	3.9	< 0.1	0.0	3.9	< 0.1
	8	4.0	< 0.1	4.0	< 0.1	0.0	4.0	< 0.1
8	3	5.9	< 0.1	5.9	< 0.1	0.0	5.9	< 0.1
	4	6.7	< 0.1	6.7	< 0.1	0.0	6.7	< 0.1
	5	6.8	< 0.1	7.0	< 0.1	0.0	7.0	< 0.1
	6	7.3	< 0.1	7.3	< 0.1	0.0	7.3	< 0.1
	7	7.6	< 0.1	7.7	< 0.1	0.0	7.7	< 0.1
	8	7.5	< 0.1	7.5	< 0.1	0.0	7.5	< 0.1
16	3	9.6	< 0.1	9.6	< 0.1	0.0	9.6	< 0.1
	4	11.1	< 0.1	11.1	< 0.1	0.0	10.9	< 0.1
	5	13.7	0.2	13.8	< 0.1	0.0	13.6	0.2
	6	13.0	< 0.1	13.2	0.1	0.0	13.1	< 0.1
	7	14.5	< 0.1	14.7	0.3	0.0	14.7	< 0.1
	8	14.7	< 0.1	15.2	0.6	0.0	15.1	0.3
32	3	16.1	< 0.1	16.1	< 0.1	0.0	16.1	< 0.1
	4	19.2	< 0.1	19.2	0.4	0.0	19.2	< 0.1
	5	20.6	0.1	20.9	1.3	0.0	20.9	< 0.1
	6	24.0	0.5	24.4	5.8	0.0	24.4	0.2
	7	24.9	< 0.1	25.8	9.4	0.0	25.8	2.8
	8	26.8	0.4	27.4	32.2	0.0	27.4	1.5
64	3	24.8	< 0.1	24.9	1.8	0.0	24.9	0.3
	4	30.1	0.1	30.3	8.7	0.0	30.3	0.9
	5	34.5	0.2	34.8	70.5	0.0	34.7	1.8
	6	38.4	0.4	38.8	231.4	1.7	39.0	8.4
	7	42.3	0.4	42.8	435.7	5.2	44.0	6.0
	8	45.1	0.9	35.4	413.1	53.1	45.7	17.0

- Concerning Set1, both Beam-ACO and CMSA provide (near-)optimal solutions and both outperform CPLEX once the average optimality gaps start to increase. However, no clear trend about the superiority of CMSA over Beam-ACO (or the other way around) is noticeable.
- Concerning Set2, the performance of Beam-ACO and CMSA is comparable for instances of alphabet sizes $|\Sigma| \in \{4, 8, 16\}$, both providing (near-)optimal solutions. However, starting from alphabet size $|\Sigma| = 32$, CMSA outperforms Beam-ACO. This becomes even more clear in the case of the extension of Set2, consisting of larger problem instances. In the context of these instances, CMSA outperforms both CPLEX and Beam-ACO.

Table 4. Experimental results for larger problem instances.

$\lvert\Sigma\rvert$	n	Beam-ACO		CPLEX			CMSA	
		result	**time (s)**	**result**	**time (s)**	**gap (%)**	**result**	**time (s)**
128	3	38.3	< 0.1	38.4	50.1	0.0	38.4	0.1
	4	44.3	< 0.1	45.3	296.1	0.0	45.2	3.8
	5	52.6	1.8	23.3	85.8	> 100.0	53.7	1.3
	6	58.6	< 0.1	18.1	78.3	> 100.0	61.2	9.2
	7	66.3	5.3	n.a.	n.a.	n.a.	68.7	26.0
	8	73.7	6.9	n.a.	n.a.	n.a.	75.8	32.1
256	3	53.6	< 0.1	7.10	102.2	> 100.0	53.6	0.7
	4	66.8	0.9	0.10	143.2	> 100.0	67.0	12.1
	5	79.4	0.6	n.a.	n.a.	n.a.	81.0	7.6
	6	90.2	22.1	n.a.	n.a.	n.a.	92.1	35.0
	7	99.4	16.5	n.a.	n.a.	n.a.	102.2	47.7
	8	109.0	23.1	n.a.	n.a.	n.a.	111.3	62.7



(a) Sub-instance size for instances of Set1.

(b) Sub-instance size for instances of Set2and larger instances.

Fig. 1. Graphical presentation of the sizes of the sub-instances in percent with respect to the size of the original problem instances.

Finally, we studied the (average) size of the sub-instances that are generated (and maintained) within CMSA in comparison to the size of the original problem instances. These sub-instance sizes are provided in a graphical way in Fig. 1a for instances of Set1, and in Fig. 1b for instances of Set2 and its extension. Note that these graphics show the sub-instance sizes averaged over all instances of the same alphabet size. In both cases, the x-axis ranges from small alphabet size (left) to large alphabet sizes (right). Interestingly, when the alphabet size is rather small, the tackled sub-instances in CMSA are rather large (up to ≈70 % of the size of the original problem instances). With growing alphabet size, the size of the tackled sub-instances decreases. This is more clearly visible in the context of instances of Set1. However, this trend also becomes clear starting from alphabet size 32 in the context of instances of Set2. The reason for this trend is as follows. As CPLEX is very efficient for problem instances based on rather small alphabet sizes, the parameter values of CMSA are chosen during the tuning process of iracesuch that the sub-instance sizes become quite large. On the contrary, with growing alphabet size, the parameter values chosen during tuning lead to smaller sub-instances, simply because CPLEX is not so efficient anymore when applied to sub-instances that are not much smaller than the original problem instances.

5 Discussion and Future Work

CMSA is a new, general, algorithm for combinatorial optimization which is based on a simple, but apparently successful, idea: the generation of sub-instances based on merging the solution components found in randomly constructed solutions, and their subsequent solution by means of an exact solver. Moreover, the considered sub-instances are dynamically changing due to adding new solution components at each iteration, and removing existing solution components on the basis of indicators about their usefulness. In this work, the CMSA algorithm has been applied to the repetition-free longest common subsequence problem. The general picture of the results, in comparison to CPLEX, is similar to the one observed in earlier applications of CMSA to the minimum common string partition problem and a minimum weight arborescence problem in [1]. CMSA is generally competitive with CPLEX for small to medium size problem instances, whereas it outperforms CPLEX with growing problem instances size. In our opinion, this algorithm is quite appealing, especially for the following reasons:

- CMSA can be applied to any problem for which a constructive heuristic and an exact solver are known.
- In comparison to other metaheuristics, CMSA can generally be implemented with few lines of code.
- When using an ILP solver for solving sub-instances, CMSA allows to make use of the valuable operations research expertise that has gone into the development of the ILP solver, without the need of being an expert in operations research.

Finally, note that the idea behind CMSA is similar, in some sense, to the basic idea of large neighborhood search (LNS) [13]. However, while exact solvers in LNS are used to search the best solution in a large neighborhood of the current solution which is generally obtained by a partial destruction of the current solution, exact solvers in the context of CMSA are applied to sub-instances of the original problem instances.

Concerning future work, we first plan to extend the conducted experimental study to even larger problem instances. Second, we intent to study the incorporation of potentially valuable knowledge about, for example, the reduced costs of variables, in order to develop a more sophisticated—and hopefully more effective—mechanism for the removal of variables from the considered sub-instances.

References

1. Blum, C., Pinacho, P., López-Ibáñez, M., Lozano, J.A.: Construct, merge, solve and adapt a new general algorithm for combinatorial optimization. Comput. Oper. Res. **68**, 75–88 (2016)
2. Blum, C., Calvo, B.: A matheuristic for the minimum weight rooted arborescence problem. J. Heuristics **21**(4), 479–499 (2015)
3. Adi, S.S., Braga, M.D.V., Fernandes, C.G., Ferreira, C.E., Martinez, F.V., Sagot, M.F., Stefanes, M.A., Tjandraatmadja, C., Wakabayashi, Y.: Repetition-free longest common subsquence. Discrete Appl. Math. **158**, 1315–1324 (2010)
4. Gusfield, D.: Algorithms on Strings, Trees, and Sequences. Computer Science and Computational Biology. Cambridge University Press, Cambridge (1997)
5. Smith, T., Waterman, M.: Identification of common molecular subsequences. J. Mol. Biol. **147**(1), 195–197 (1981)
6. Jiang, T., Lin, G., Ma, B., Zhang, K.: A general edit distance between RNA structures. J. Comput. Biol. **9**(2), 371–388 (2002)
7. Storer, J.: Data Compression: Methods and Theory. Computer Science Press, MD (1988)
8. Aho, A., Hopcroft, J., Ullman, J.: Data Structures and Algorithms. Addison-Wesley, Reading (1983)
9. Maier, D.: The complexity of some problems on subsequences and supersequences. J. ACM **25**, 322–336 (1978)
10. Blum, C., Blesa, M.J., Calvo, B.: Beam-ACO for the repetition-free longest common subsequence problem. In: Legrand, P., Corsini, M.-M., Hao, J.-K., Monmarché, N., Lutton, E., Schoenauer, M. (eds.) EA 2013. LNCS, vol. 8752, pp. 79–90. Springer, Heidelberg (2014)
11. Castelli, M., Beretta, S., Vanneschi, L.: A hybrid genetic algorithm for the repetition free longest common subsequence problem. Oper. Res. Lett. **41**(6), 644–649 (2013)
12. López-Ibáñez, M., Dubois-Lacoste, J., Stützle, T., Birattari, M.: The irace package, iterated race for automatic algorithm configuration. Technical report TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, Belgium (2011)
13. Pisinger, D., Ropke, S.: Large neighborhood search. In: Gendreau, M., Potvin, J.Y. (eds.) Handbook of Metaheuristics. International Series in Operations Research and Management Science, vol. 146, pp. 399–419. Springer, US (2010)