Construct, Merge, Solve and Adapt: Application to Unbalanced Minimum Common String Partition

Christian Blum[1,2(✉)]

[1] Department of Computer Science and Artificial Intelligence, University of the Basque Country UPV/EHU, San Sebastian, Spain
christian.c.blum@gmail.com
[2] IKERBASQUE, Basque Foundation for Science, Bilbao, Spain

Abstract. In this paper we present the application of a recently proposed, general, algorithm for combinatorial optimization to the unbalanced minimum common string partition problem. The algorithm, which is labelled Construct, Merge, Solve & Adapt, works on sub-instances of the tackled problem instances. At each iteration, the incumbent sub-instance is modified by adding solution components found in probabilistically constructed solutions to the tackled problem instance. Moreover, the incumbent sub-instance is solved to optimality (if possible) by means of an integer linear programming solver. Finally, seemingly unuseful solution components are removed from the incumbent sub-instance based on an ageing mechanism. The results obtained for the unbalanced minimum common string partition problem indicate that the proposed algorithm outperforms a greedy approach. Moreover, they show that the algorithm is competitive with CPLEX for problem instances of small and medium size, whereas it outperforms CPLEX for larger problem instances.

1 Introduction

The drawback of exact solvers in the context of combinatorial optimization problems is often that they are not applicable to problem instances of realistic sizes. When small problem instances are considered, however, exact solvers are often extremely efficient. This is because a considerable amount of time, effort and expertise has gone into the development of exact solvers. As examples consider general-purpose integer linear programming solvers such as CPLEX and Gurobi. Having this in mind, recent research efforts focused on ways of making use of

This work was supported by project TIN2012-37930-C02-02 (Spanish Ministry for Economy and Competitiveness, FEDER funds from the European Union). Additionally, we acknowledge 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.

M.J. Blesa et al. (Eds.): HM 2016, LNCS 9668, pp. 17–31, 2016.
DOI: 10.1007/978-3-319-39636-1_2

exact solvers within heuristic frameworks even in the context of large problem instances. A recently proposed algorithm labelled Construct, Merge, Solve & Adapt (CMSA) [1,3] falls into this line of research. The 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 an incumbent sub-instance of the original problem instance. Subsequently, an exact solver such as, for example, CPLEX is used to solve the incumbent sub-instance to optimality. Moreover, the algorithm makes use of a mechanism for deleting seemingly useless solution components from the incumbent sub-instance. This is done in order to avoid that these solution components slow down the exact solver when applied to the sub-instance.

In this work we apply the CMSA algorithm to the unbalanced minimum common string partition problem (UMCSP) [4]. This problem, which is NP-hard, is a generalization of the well-known minimum common string partition problem (MCSP) [5]. The UMCSP seems to be well-suited for being tackled with CMSA, because the integer linear programming (ILP) model that we present in this work (see Sect. 2) contains an exponential number of binary variables and can, therefore, only be solved to optimality in the context of problem instances of small and medium size. 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 unbalanced minimum common string partition problem. Moreover, we describe the first ILP model for this problem. Next, in Sect. 4, the application of CMSA to the tackled problem is outlined. Finally, Sect. 5 provides an extensive experimental evaluation and Sect. 6 offers a discussion and an outlook to future work.

2 Unbalanced Minimum Common String Partition

The UMCSP problem can technically be described as follows. Given is an input string s^1 of length n_1 and an input string s^2 of length n_2, both over the same finite alphabet Σ. A valid solution to the UMCSP problem is obtained by partitioning s^1 into a set P_1 of non-overlapping substrings, and s^2 into a set P_2 of non-overlapping substrings, such that exists a set S with $S \subseteq P_1$ and $S \subseteq P_2$ and no letter $a \in \Sigma$ is simultaneously present in a string $x \in P_1 \backslash S$ and a string $y \in P_2 \backslash S$. Henceforth, given P_1 and P_2 let us denote the largest subset S such that the above-mentioned condition holds by S^*. The objective function value of a solution (P_1, P_2) is then $|S^*|$. The goal consists in finding a solution (P_1, P_2) such that $|S^*|$ is minimal.

Consider the following example. Given are DNA sequences $s^1 = \mathbf{AAGACTG}$ and $s^2 = \mathbf{TACTAG}$. A trivial valid solution can be obtained by partitioning both strings into substrings of length 1, that is, $P_1 = \{\mathbf{A}, \mathbf{A}, \mathbf{A}, \mathbf{C}, \mathbf{T}, \mathbf{G}, \mathbf{G}\}$ and $P_2 = \{\mathbf{A}, \mathbf{A}, \mathbf{C}, \mathbf{T}, \mathbf{T}, \mathbf{G}\}$. In this case, $S^* = \{\mathbf{A}, \mathbf{A}, \mathbf{C}, \mathbf{T}, \mathbf{G}\}$, and the

objective function value is $|S^*| = 5$. However, the optimal solution, with objective function value 2, is $P_1 = \{\mathbf{ACT}, \mathbf{AG}, \mathbf{A}, \mathbf{G}\}$, $P_2 = \{\mathbf{ACT}, \mathbf{AG}, \mathbf{T}\}$ and $S^* = \{\mathbf{ACT}, \mathbf{AG}\}$.

Note that the UMCSP problem [4] is a generalization of the well-known minimum common string partition (MCSP) problem, which was introduced in [5] due to its relation to genome rearrangement. In fact, the MCSP problem is obtained in case the input strings s^1 and s^2 are related, that is, in case all letters appear the same number of times in s^1 and in s^2. The MCSP problem was shown to be NP-hard even in very restrictive cases [8]. Therefore, the more general UMCSP problem is also NP-hard. In contrast to the MCSP, the UMCSP has not been tackled yet by means of heuristics or metaheuristics. The only existing algorithm is a fixed-parameter approximation algorithm described in [4]. The more specific MCSP problem has been tackled by a greedy heuristic [9], an ant colony optimization approach [6,7], and probabilistic tree search [2]. Finally, the application of the CMSA algorithm to the MCSP problem (see [3]) is currently the state-of-the-art algorithm for this problem.

3 An ILP Model for the UMCSP Problem

In order to derive an ILP model for the UMCSP problem, we introduce in the following the *common block* concept, which allows to re-phrase the problem in a different way. A *common block* b_i concerning input strings s^1 and s^2 is denoted as a triple (t_i, k_i^1, k_i^2) where t_i is a string which can be found starting at position $1 \leq k_i^1 \leq n_1$ in string s^1 and starting at position $1 \leq k_i^2 \leq n_2$ in string s^2. Let $B = \{b_1, \ldots, b_m\}$ be the arbitrarily ordered set of all possible common blocks of s^1 and s^2. Moreover, given a string t over alphabet Σ, $n(t, a)$ denotes the number of occurrences of letter $a \in \Sigma$ in string t. Specifically, $n(s^1, a)$, respectively $n(s^2, a)$, are the number of occurrences of letter $a \in \Sigma$ in input string s^1, respectively s^2. Given the definition of B, a subset S of B corresponds to a valid solution to the UMCSP problem iff the following conditions hold:

1. $\sum_{b_i \in S} n(t_i, a) = \min\{n(s^1, a), n(s^2, a)\}$ for all $a \in \Sigma$. In other words, the sum of the occurrences of a letter $a \in \Sigma$ in the common blocks present in S must be equal to the minimum number of occurrences of letter a in s^1 and s^2.
2. For any two common blocks $b_i, b_j \in S$ it holds that their corresponding strings neither overlap in s^1 nor in s^2.

With these definitions we can state the following ILP model for the UMCSP problem, which uses for each common block $b_i \in B$ a binary variable x_i indicating its selection in the solution. In other words, if $x_i = 1$, the corresponding common block b_i is selected for the solution, and if $x_i = 0$, common block b_i is not selected.

$$\min \quad \sum_{i=1}^{m} x_i \tag{1}$$

$$\text{s.t.} \quad \sum_{i \in \{1,\ldots,m | k_i^1 \leq j < k_i^1 + |t_i|\}} x_i = 1 \qquad \text{for } j = 1, \ldots, n_1 \tag{2}$$

$$\sum_{i \in \{1,\dots,m | k_i^2 \leq j < k_i^2 + |t_i|\}} x_i = 1 \qquad \text{for } j = 1, \dots, n_2 \qquad (3)$$

$$\sum_{i=1}^{m} n(t_i, a) x_i = \min\{n(s^1, a), n(s^2, a)\} \qquad \text{for } a \in \Sigma \qquad (4)$$

$$x_i \in \{0, 1\} \qquad \text{for } i = 1, \dots, m$$

The objective function (1) minimizes the number of selected common blocks. Equation (2) ensure that the strings corresponding to the selected common blocks do not overlap with respect to s^1, and Eq. (3) ensure the same with respect to s^2. Finally, Eq. (4) ensure that the number of occurrences of each letter in the selected strings is equal to the minimum number of occurrences of this letter in s^1 and s^2.

4 Application of CMSA to the UMCSP Problem

The (CMSA) algorithm, whose pseudo-code is given in Algorithm 1, works as follows. It maintains an incumbent sub-instance B', which is a subset of the complete set B of common blocks. Moreover, each common block $b_i \in B$ has a non-negative age value denoted by $age[b_i]$. In an initialization step, the best-so-far solution S_{bsf} is set to NULL, indicating that no such solution exists yet, and the sub-instance B' is initilized to the empty set. Then, at each iteration a number of n_{a} solutions is probabilistically generated, see function ProbabilisticSolutionGeneration(B) in line 6 of Algorithm 1. The common blocks found in these solutions are added to B' and their age is re-initilalized to 0. Afterwards, an ILP solver—we used CPLEX—is applied to solve sub-instance B', if possible within the given CPU time limit, to optimality; see function ApplyExactSolver(B') in line 12 of Algorithm 1. If S'_{opt} is better than the current best-so-far solution S_{bsf}, solution S'_{opt} replaces the best-so-far solution (line 13). Next, sub-instance B' is adapted, based on solution S'_{opt} and on the age values of the common blocks. This is done in function Adapt(B', S'_{opt}, $age_{\max}$) in line 14. In the following we outline the functions of the algorithm in more detail.

Function ProbabilisticSolutionGeneration(B): Henceforth we call $S \subset B$ a valid partial solution if the substrings corresponding to the common blocks in S do not overlap neither concerning s^1 nor concerning s^2. Furthermore, let set $Ext(S) \subset B \backslash S$ denote the set of common blocks that may be used in order to extend S such that the result is again a valid (partial) solution. Note that when $Ext(S) = \emptyset$, S corresponds to a complete solution. Given these definitions, a simple greedy heuristic—which is an extension of the greedy heuristic from [9] for the MCSP problem—starts with the empty partial solution $S := \emptyset$ and chooses at each step from $Ext(S)$ the common block with the longest substring. This greedy heuristic will henceforth be called GREEDY.

In function ProbabilisticSolutionGeneration(B) of line 6 of Algorithm 1 we make use of the following probabilistic version of GREEDY for generating solutions to the tackled problem instance. More specifically, the construction of a

Algorithm 1. CMSA for the UMCSP problem

given: set B corresponding to the tackled problem instance, values for parameters n_{a} and $age_{\max}$
$S_{\text{bsf}} := \text{NULL}$; $B' := \emptyset$
$age[b_i] := 0$ for all $b_i \in B$
while CPU time limit not reached **do**
  for $i = 1, \ldots, n_{\text{a}}$ **do**
    $S :=$ ProbabilisticSolutionGeneration(B)
    for all $b_i \in S$ **and** $b_i \notin B'$ **do**
      $age[b_i] := 0$
      $B' := B' \cup \{b_i\}$
    end for
  end for
  $S'_{\text{opt}} :=$ ApplyExactSolver(B')
  if $|S'_{\text{opt}}| < |S_{\text{bsf}}|$ **then** $S_{\text{bsf}} := S'_{\text{opt}}$
  Adapt(B', S'_{opt}, $age_{\max}$)
end while
return s_{bsf}

solution (see Algorithm 2) starts with the empty partial solution $S := \emptyset$. At each construction step, a solution component b_i from $Ext(S)$ is chosen and added to S. This is done until S is a complete solution, that is, until $|Ext(S)| = 0$. The choice of b_i is done as follows. First, a value $\delta \in [0, 1)$ is chosen uniformly at random. In case $\delta \leq d_{\text{rate}}$, b_i is chosen such that $|t_i| \geq |t_j|$ for all $b_j \in Ext(S)$, that is, one of the common blocks whose substring is of maximal size is chosen. Otherwise, a candidate list L containing the (at most) l_{size} longest common blocks from $Ext(S)$ is built, and b_i is chosen from L uniformly at random. In other words, the greediness of this procedure depends on the pre-determined values of d_{rate} (determinism rate) and l_{size} (candidate list size). Both are input parameters of the algorithm.

Function ApplyExactSolver(B'): In this function, CPLEX is applied to the ILP model outlined in Sect. 3 for solving sub-instance B'. This is achieved by replacing all occurrences of B in this ILP model with B', and by replacing m with $|B'|$.

Function Adapt(B', S'_{opt}, $age_{\max}$): First, the age of each common block in $B' \backslash S'_{\text{opt}}$ is incremented while the age of each common block in $S'_{\text{opt}} \subseteq B'$ is re-initialized to zero. Then, those common blocks from B' whose age has reached the maximum component age ($age_{\max}$) are deleted from B'. The motivation behind the aging mechanism is that common blocks which never appear in the solutions of B' returned by the exact solver should be removed from B' after some time, because they would otherwise slow down the exact solver on the long term. In contrast, common blocks which appear in the solutions returned by the exact solver seem to be useful and should therefore remain in B'.

Algorithm 2. Function ProbabilisticSolutionGeneration(B)

given: B, d_{rate}, l_{size}
$S := \emptyset$
while $|Ext(S)| > 0$ **do**
  choose a random number $\delta \in [0, 1]$
  if $\delta \leq d_{\text{rate}}$ **then**
    choose b_i such that $|t_i| \geq |t_j|$ for all $b_j \in Ext(S)$
    $S := S \cup \{b_i\}$
  else
    let $L \subseteq Ext(S)$ contain the (at most) l_{size} longest common blocks from $Ext(S)$
    choose b_i from L uniformly at random
    $S := S \cup \{b_i\}$
  end if
end while
return complete solution S

5 Experimental Evaluation

Three different solution methods are compared in the following. The first on is Greedy, the simple, deterministic, greedy algorithm mentioned in Sect. 4 in the context of probabilistically generating solutions to the UMCSP problem. The second one is the CMSA algorithm, henceforth denoted by Cmsa. And the third one is the application of IBM ILOG CPLEX v12.1 to the original problem instances, labelled Cplex. The solution methods were implemented in ANSI C++ using GCC 4.7.3. Both in the context of Cmsa and Cplex, CPLEX was used in one-threaded mode. The experimental evaluation was performed on a cluster of PCs with Intel(R) Xeon(R) CPU 5670 CPUs of 12 nuclei of 2933 MHz and at least 40 Gb of RAM. Note that the fixed-parameter approximation algorithm described in [4] was not included in the comparison because, according to the authors of this work, the algorithm is only applicable to very small problem instances.

In the following we first describe the set of benchmark instances that we generated to test the considered solution methods. Then, we describe the tuning experiments that were performed in order to determine a proper setting for the parameters of Cmsa. Finally, an exhaustive experimental evaluation is presented.

5.1 Problem Instances

For the comparison of the three considered solution methods we generated a set of 600 benchmark instances. In more detail, this benchmark set consists of 10 randomly generated instances for each combination of the *base-length* $n \in \{200, 400, \ldots, 1800, 2000\}$, the alphabet size $|\Sigma| \in \{4, 12\}$, and a so-called *length-difference* $ld \in \{0, 10, 20\}$. In the context of all instances, each letter of Σ has the same probability to appear at any of the positions of input strings s^1 and s^2.

Given a value for the base-length n and the length-difference ld, the length of s^1 is determined as $n + \lfloor (ld \cdot n)/100 \rfloor$ and the length of s^2 as $n - \lfloor (ld \cdot n)/100 \rfloor$. In other words, ld refers to the length difference between s^1 and s^2 (in percent) given a certain base-lenth n.

5.2 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, (age_max) the maximum allowed age of common blocks, (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 made use of the automatic configuration tool irace [10] for the tuning of the five parameters. In fact, irace was applied to tune CMSA separately for instances of each *base-length*, which—after initial experiments—seemed to be the parameter with most influence on the algorithm performance. For each of the 10 considered base-length values, 12 tuning instances were randomly generated: two for each of the six combinations of Σ and ld. The tuning process for each alphabet size was given a budget of 1000 runs of CMSA, where each run was given a computation time limit of 3600 CPU seconds. Finally, the following parameter value ranges were chosen concerning the five parameters of CMSA:

- $n_a \in \{10, 30, 50\}$.
- $\mathrm{age}_\mathrm{max} \in \{1, 5, 10, inf\}$, where inf means that no common block is ever removed from sub-instance B'.

Table 1. Results of tuning CMSA with irace.

n	n_a	age_max	d_rate	l_size	t_max
200	50	10	0.0	10	480
400	50	10	0.0	10	120
600	50	10	0.0	10	240
800	50	5	0.5	10	120
1000	50	10	0.7	10	60
1200	50	5	0.5	10	120
1400	50	10	0.9	10	480
1600	50	5	0.9	10	480
1800	50	5	0.9	10	480
2000	50	10	0.9	10	480

- $d_{\text{rate}} \in \{0.0, 0.3, 0.5, 0.7, 0.9\}$, where a value of 0.0 means that the selection of the next common block 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.
- $l_{\text{size}} \in \{3, 5, 10\}$.
- $t_{\text{max}} \in \{60, 120, 240, 480\}$ (in seconds).

The tuning runs with irace produced the configurations of CMSA as shown in Table 1. The most important tendencies that can be observed are the following ones. First, with growing base-length, the greediness of the solution construction grows, as indicated by the increasing value of d_{rate}. Second, the number of solution constructions per iteration is always high. Third, the time limit for CPLEX does not play any role for smaller instances. However, for larger instances the time limit of 480 seconds is consistently chosen.

5.3 Experimental Results

The numerical results are presented in Table 2 concerning all instances with $|\Sigma| = 4$, and in Table 3 concerning all instances with $|\Sigma| = 12$. Each table row presents the results averaged over 10 problem instances of the same type. For each of the three solution methods in the comparison we provide (at least) the following two columns. The first one (with heading **mean**) provides the average values of the best solutions obtained over 10 problem instances, while the second column (with heading **time**) provides the average computation time (in seconds) necessary for finding the corresponding solutions. In the case of CPLEX, this column provides two values in the form X/Y, where X corresponds to the (average) time at which CPLEX was able to find the first valid solution, and Y to the (average) time at which CPLEX found the best solution within 3600 CPU seconds. An additional column with heading **gap** provides—in the case of CPLEX—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. A third additional column in the case of CMSA (with heading **size** (%)) provides the average size of the sub-instances considered in CMSA in percent of the original problem instance sizes, that is, the sizes of the complete sets B of common blocks. Finally, note that the best result for each table row is marked by a gray background and the last row of each table provides averages over the whole table. Moreover, the numerical results are presented graphically in Figs. 1 and 2 in terms of the improvement of CMSA over CPLEX and GREEDY (in percent).

The results allow to make the following observations:

- Concerning the application of CPLEX to the original problem instances, the alphabet size has a strong influence on the problem difficulty. For instances with $|\Sigma| = 4$, CPLEX is only able to provide feasible solutions within 3600 CPU seconds for input strings of lengths up to 800. When $|\Sigma| = 12$, CPLEX provides feasible solutions for input strings of lengths up to 1600 (for values of $ld \in \{0, 10\}$). When $ld = 20$ CPLEX is even able to provide feasible solutions for all problem instances.

Table 2. Results for the instances with $|\Sigma| = 4$.

(a) Results for instances with $ld = 0$.

n	Greedy		Cplex			Cmsa		
	mean	**time**	**mean**	**time**	**gap**	**mean**	**time**	**size (%)**
200	67.0	< 1.0	55.3	4/21	0.0	55.3	90.1	30.7
400	119.4	< 1.0	98.7	118/1445	2.1	99.4	1878.3	14.7
600	172.8	< 1.0	146.0	556/1865	6.7	145.7	2317.5	9.3
800	222.5	< 1.0	189.1	2136/3525	8.1	190.8	1837.3	5.0
1000	271.7	1.6	n.a	n.a.	n.a.	235.1	1320.3	6.1
1200	314.3	2.0	n.a	n.a.	n.a.	274.1	1837.9	3.5
1400	368.5	3.7	n.a	n.a.	n.a.	320.4	2455.8	2.6
1600	413.2	4.9	n.a	n.a.	n.a.	358.3	2875.8	2.0
1800	450.5	6.7	n.a	n.a.	n.a.	401.6	2802.1	1.7
2000	504.8	9.3	n.a	n.a.	n.a.	453.4	2166.7	1.5
avg.								

(b) Results for instances with $ld = 10$.

n	Greedy		Cplex			Cmsa		
	mean	**time**	**mean**	**time**	**gap**	**mean**	**time**	**size (%)**
200	61.0	< 1.0	52.1	3/5	0.0	52.1	88.0	29.0
400	108.4	< 1.0	90.3	102/675	0.0	91.2	801.0	13.0
600	151.4	< 1.0	126.3	548/3018	2.5	127.3	1500.4	7.8
800	192.6	< 1.0	164.2	2038/3583	4.6	164.6	1513.2	3.6
1000	232.8	1.5	n.a.	n.a.	n.a.	198.3	2504.4	3.5
1200	269.8	1.9	n.a.	n.a.	n.a.	231.7	2334.2	2.2
1400	313.1	3.5	n.a.	n.a.	n.a.	267.6	2170.0	1.8
1600	346.6	4.6	n.a.	n.a.	n.a.	301.3	3114.6	1.3
1800	383.3	6.4	n.a.	n.a.	n.a.	330.9	2652.8	1.1
2000	423.1	8.8	n.a.	n.a.	n.a.	364.7	2512.3	1.0
avg.								

(c) Results for instances with $ld = 20$.

n	Greedy		Cplex			Cmsa		
	mean	**time**	**mean**	**time**	**gap**	**mean**	**time**	**size (%)**
200	51.8	< 1.0	44.8	3/3	0.0	44.8	241.9	23.8
400	89.4	< 1.0	77.4	86/90	0.0	77.6	251.6	9.9
600	127.9	< 1.0	108.5	467/634	0.0	109.3	716.8	5.7
800	159.5	< 1.0	135.8	1941/2583	0.2	138.0	1009.4	2.8
1000	197.9	1.4	n.a.	n.a.	n.a.	169.6	1220.1	2.7
1200	229.9	1.7	n.a.	n.a.	n.a.	198.9	1659.6	1.7
1400	262.1	3.1	n.a.	n.a.	n.a.	229.2	2052.5	1.7
1600	294.4	4.1	n.a.	n.a.	n.a.	255.4	1830.2	1.2
1800	327.6	5.9	n.a.	n.a.	n.a.	284.5	2620.1	1.0
2000	359.0	8.1	n.a.	n.a.	n.a.	313.0	2160.0	1.0
avg.								

Table 3. Results for the instances with $|\Sigma| = 12$.

(a) Results for instances with $ld = 0$.

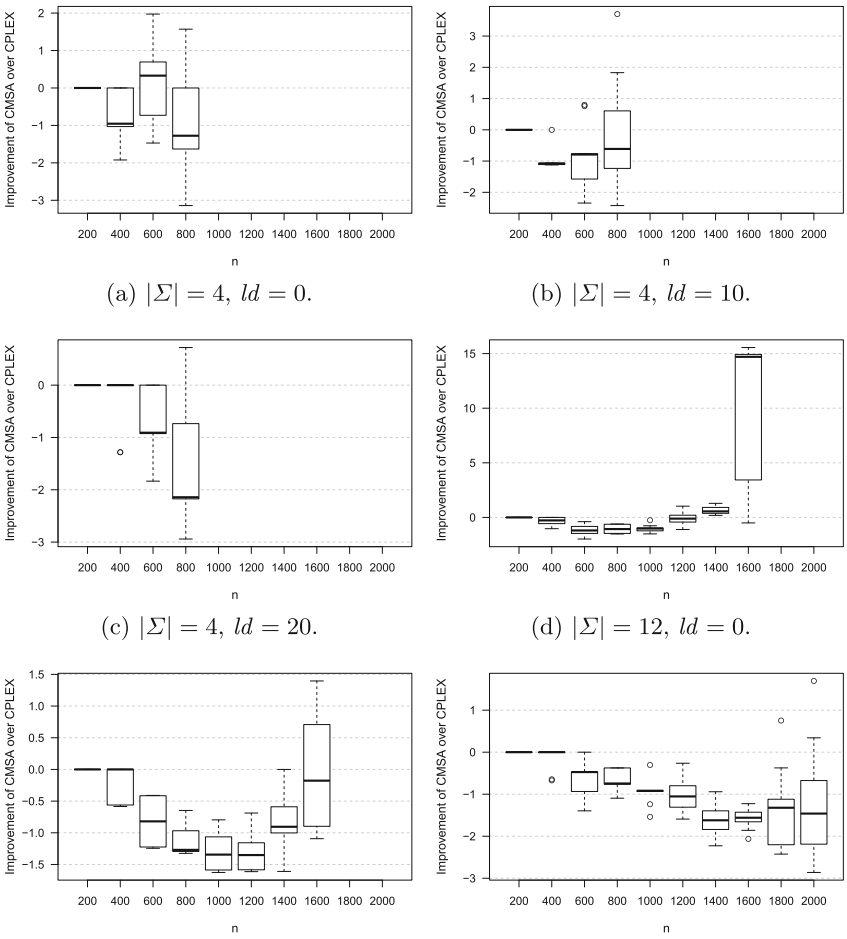
n	Greedy		Cplex			Cmsa		
	mean	**time**	**mean**	**time**	**gap**	**mean**	**time**	**size (%)**
200	107.5	< 1.0	98.1	0/0	0.0	98.1	0.3	52.0
400	204.7	< 1.0	181.2	4/7	0.0	181.8	791.5	34.0
600	290.2	< 1.0	251.8	28/599	0.0	254.9	1710.8	23.2
800	379.5	< 1.0	328.1	134/1338	1.1	331.6	1020.8	18.3
1000	467.3	< 1.0	399.5	305/2413	1.6	403.7	1682.6	18.3
1200	537.5	1.0	468.7	672/2800	2.9	469.3	2057.3	12.7
1400	624.7	2.1	543.2	1089/2641	3.6	539.7	2108.7	10.3
1600	706.0	2.6	672.2	2193/2661	12.0	610.9	1875.5	8.7
1800	794.2	3.6	n.a.	n.a.	n.a.	694.2	2300.9	8.0
2000	876.3	5.1	n.a.	n.a.	n.a.	758.1	2356.0	7.4
avg.								

(b) Results for instances with $ld = 10$.

n	Greedy		Cplex			Cmsa		
	mean	**time**	**mean**	**time**	**gap**	**mean**	**time**	**size (%)**
200	104.2	< 1.0	95.9	0/0	0.0	95.9	0.9	54.2
400	196.5	< 1.0	173.5	5/7	0.0	173.8	733.1	32.7
600	274.2	< 1.0	240.1	34/119	0.0	242.0	1275.3	22.7
800	354.1	< 1.0	304.9	109/903	0.2	308.3	605.0	16.0
1000	427.3	< 1.0	369.0	303/2035	0.3	373.9	1332.0	15.2
1200	502.6	1.0	433.3	550/2457	0.7	439.0	1615.6	10.8
1400	570.1	1.7	494.4	1011/2249	1.3	498.6	1146.7	9.4
1600	646.8	2.4	561.4	1738/3071	2.1	561.8	1493.3	7.1
1800	712.7	3.6	n.a.	n.a.	n.a.	621.0	1571.2	6.2
2000	776.5	4.3	n.a.	n.a.	n.a.	678.2	2079.4	5.6
avg.								

(c) Results for instances with $ld = 20$.

n	Greedy		Cplex			Cmsa		
	mean	**time**	**mean**	**time**	**gap**	**mean**	**time**	**size (%)**
200	93.9	< 1.0	86.0	0/0	0.0	86.0	0.8	51.0
400	170.2	< 1.0	151.4	4/5	0.0	151.6	101.7	29.1
600	237.4	< 1.0	210.8	28/29	0.0	212.1	688.4	18.1
800	301.3	< 1.0	267.2	99/102	0.0	269.1	741.8	12.1
1000	365.6	< 1.0	323.1	250/277	0.0	326.2	425.6	11.6
1200	426.3	< 1.0	374.0	568/591	0.0	377.7	772.9	8.2
1400	484.4	1.4	426.6	997/1142	0.0	433.6	609.1	7.3
1600	542.0	2.0	477.1	1331/1640	0.2	484.7	792.4	5.5
1800	598.3	3.1	528.1	2043/2387	0.5	535.2	698.2	5.0
2000	663.2	4.1	680.5	2971/3132	7.1	587.7	518.5	5.2
avg.								



(a) $|\Sigma| = 4,\ ld = 0.$ (b) $|\Sigma| = 4,\ ld = 10.$

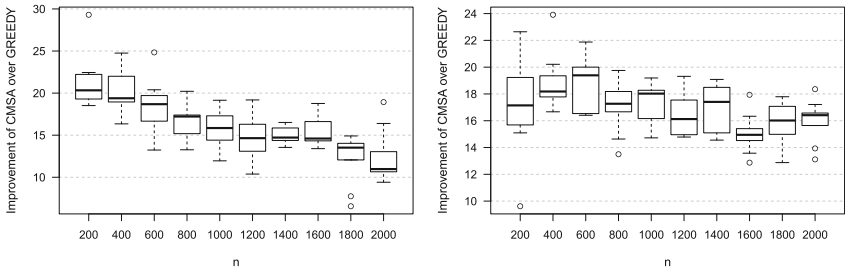
(c) $|\Sigma| = 4,\ ld = 20.$ (d) $|\Sigma| = 12,\ ld = 0.$

(e) $|\Sigma| = 12,\ ld = 10.$ (f) $|\Sigma| = 12,\ ld = 20.$

Fig. 1. Improvement of CMSA over CPLEX (in percent). Note that when boxes are missing, CPLEX was not able to provide feasible solutions within the allowed computation time.

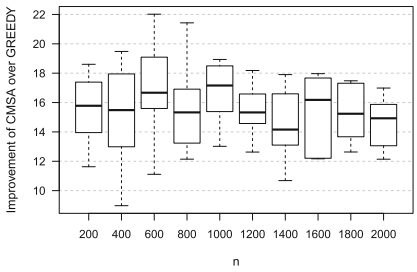
– In contrast to CPLEX, CMSA is able to provide feasible solutions for all problem instances. Moreover, CMSA outperforms GREEDY in all cases. In those cases in which CPLEX is able to provide feasible (or even optimal) solutions, CMSA is either competitive, or not much worse than CPLEX. In particular, CMSA is never more than 3 % worse than CPLEX.

In summary, we can state that CMSA is competitive with the application of CPLEX to the original ILP model when the size of the input instances is rather small. The larger the size of the input instances, and the smaller the alphabet size, the greater is—in general—the advantage of CMSA over the other algorithms.

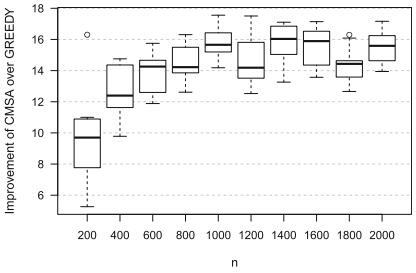


(a) $|\Sigma| = 4$, $ld = 0$.

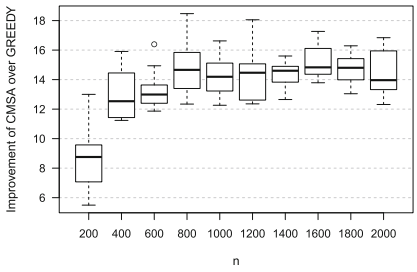
(b) $|\Sigma| = 4$, $ld = 10$.



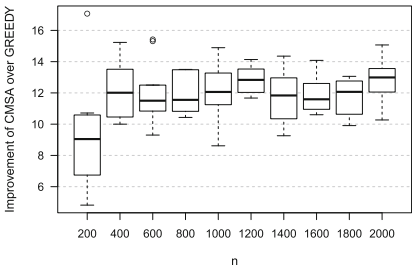
(c) $|\Sigma| = 4$, $ld = 20$.



(d) $|\Sigma| = 12$, $ld = 0$.



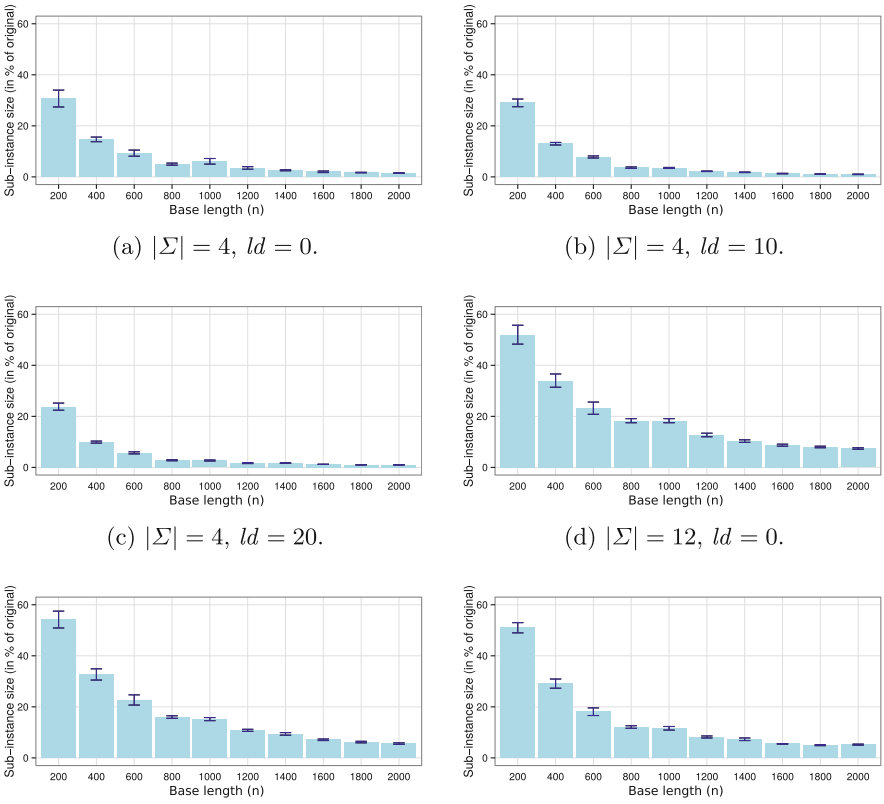
(e) $|\Sigma| = 12$, $ld = 10$.



(f) $|\Sigma| = 12$, $ld = 20$.

Fig. 2. Improvement of CMSA over GREEDY (in percent).

Finally, we also present the 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. 3. Note that these graphics show the sub-instance sizes averaged over all instances of the same alphabet size and the same value for ld. In all cases, the x-axis ranges from instances with a small base-length (n) at the left, to instance with a large base-length at the right. Interestingly, when the base-length is rather small, the tackled sub-instances in CMSA are rather large (up to $\approx 55\,\%$ of the size of the original problem instances). With growing base-length, the size of the tackled sub-instances decreases. The reason for this trend is as follows. As CPLEX is very efficient for problem instances created with rather small



(a) $|\Sigma| = 4,\ ld = 0.$ (b) $|\Sigma| = 4,\ ld = 10.$

(c) $|\Sigma| = 4,\ ld = 20.$ (d) $|\Sigma| = 12,\ ld = 0.$

(e) $|\Sigma| = 12,\ ld = 10.$ (f) $|\Sigma| = 12,\ ld = 20.$

Fig. 3. Graphical presentation of the sizes of the sub-instances in percent with respect to the size of the original problem instances.

base-lengths, the parameter values of CMSA are chosen during the tuning process of irace such that the sub-instance sizes become quite large. On the contrary, with growing base-length, 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.

6 Discussion and Future Work

CMSA is a recently proposed, general, algorithm for combinatorial optimization. The algorithm is based on a simple, but seemingly successful, idea: (1) the creation of sub-instances based on merging the solution components found in randomly constructed solutions, and (2) the subsequent solution of these sub-instances by means of an exact solver. In this process, the considered sub-instances undergo dynamic changes caused by 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 was applied to the unbalanced minimum common string partition problem. The nature of the obtained 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 [3]. CMSA is generally competitive with—or not much worse than—CPLEX for small to medium size problem instances, whereas it outperforms CPLEX with growing problem instances size. In the opinion of the authors, this algorithm is quite appealing, especially for the following reasons:

- Given a constructive heuristic and an exact solver, CMSA can be applied to any combinatorial optimization problem.
- Compared to other metaheuristics, the implementation of CMSA is rather simple and involves only a few lines of code in addition to the heuristic and the exact solver.

Finally, it is important to observe that the idea behind CMSA is related, in some sense, to the basic idea of large neighborhood search (LNS) [11]. However, while LNS uses exact solvers for searching the best solution in a large neighborhood of the current solution, 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 plan to indentify the respective strengths and weaknesses of LNS and CMSA.

References

1. Blum, C., Calvo, B.: A matheuristic for the minimum weight rooted arborescence problem. J. Heuristics **21**(4), 479–499 (2015)
2. Blum, C., Lozano, J.A., Pinacho Davidson, P.: Iterative probabilistic tree search for the minimum common string partition problem. In: Blesa, M.J., Blum, C., Voß, S. (eds.) HM 2014. LNCS, vol. 8457, pp. 145–154. Springer, Heidelberg (2014)
3. Blum, C., Pinacho, P., López-Ibáñez, M., Lozano, J.A.: Construct, Merge, Solve & Adapt: A new general algorithm for combinatorial optimization. Comput. Oper. Res. **68**, 75–88 (2016)
4. Bulteau, L., Fertin, G., Komusiewicz, C., Rusu, I.: A fixed-parameter algorithm for minimum common string partition with few duplications. In: Darling, A., Stoye, J. (eds.) WABI 2013. LNCS, vol. 8126, pp. 244–258. Springer, Heidelberg (2013)
5. Chen, X., Zheng, J., Fu, Z., Nan, P., Zhong, Y., Lonardi, S., Jiang, T.: Computing the assignment of orthologous genes via genome rearrangement. In: Proceedings of the Asia Pacific Bioinformatics Conference 2005, pp. 363–378 (2005)
6. Ferdous, S.M., Sohel Rahman, M.: Solving the minimum common string partition problem with the help of ants. In: Tan, Y., Shi, Y., Mo, H. (eds.) ICSI 2013, Part I. LNCS, vol. 7928, pp. 306–313. Springer, Heidelberg (2013)
7. Ferdous, S.M., Sohel Rahman, M.: A MAX-MIN ant colony system for minimum common string partition problem. *CoRR*, abs/1401.4539 (2014). http://arxiv.org/abs/1401.4539

8. Goldstein, A., Kolman, P., Zheng, J.: Minimum common string partition problem: hardness and approximations. In: Fleischer, R., Trippen, G. (eds.) ISAAC 2004. LNCS, vol. 3341, pp. 484–495. Springer, Heidelberg (2004)
9. He, D.: A novel greedy algorithm for the minimum common string partition problem. In: Măndoiu, I.I., Zelikovsky, A. (eds.) ISBRA 2007. LNCS (LNBI), vol. 4463, pp. 441–452. Springer, Heidelberg (2007)
10. 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)
11. Pisinger, D., Ropke, S.: Large neighborhood search. In: Gendreau, M., Potvin, J.-Y. (eds.) Handbook of Metaheuristics. International Series in Operations Research & Management Science, vol. 146, pp. 399–419. Springer, US (2010)

http://www.springer.com/978-3-319-39635-4

Hybrid Metaheuristics
10th International Workshop, HM 2016, Plymouth, UK, June 8-10, 2016, Proceedings
Blesa Aguilera, M.J.; Blum, C.; Cangelosi, A.; Cutello, V.; DI NUOVO, A.; Pavone, M.; Talbi, E.-G. (Eds.)
2016, XII, 223 p. 70 illus., Softcover
ISBN: 978-3-319-39635-4