Construct, Merge, Solve & Adapt: A New General Algorithm For Combinatorial Optimization

Christian Blum[a,b], Pedro Pinacho[a,c], Manuel López-Ibáñez[d], José A. Lozano[a]

[a]*Department of Computer Science and Artificial Intelligence, University of the Basque Country UPV/EHU, San Sebastian, Spain*
[b]*IKERBASQUE, Basque Foundation for Science, Bilbao, Spain*
[c]*Escuela de Informática, Universidad Santo Tomás, Concepción, Chile*
[d]*Alliance Manchester Business School, University of Manchester, UK*

Abstract

This paper describes a general hybrid metaheuristic for combinatorial optimization labelled Construct, Merge, Solve & Adapt. The proposed algorithm is a specific instantiation of a framework known from the literature as Generate-And-Solve, which is based on the following general idea. First, generate a reduced sub-instance of the original problem instance, in a way such that a solution to the sub-instance is also a solution to the original problem instance. Second, apply an exact solver to the reduced sub-instance in order to obtain a (possibly) high quality solution to the original problem instance. And third, make use of the results of the exact solver as feedback for the next algorithm iteration. The minimum common string partition problem and the minimum covering arborescence problem are chosen as test cases in order to demonstrate the application of the proposed algorithm. The obtained results show that the algorithm is competitive with the exact solver for small to medium size problem instances, while it significantly outperforms the exact solver for larger problem instances.

Keywords: Metaheuristics, Exact Solver, Hybrid Algorithms, Minimum Common String Partition, Minimum Covering Arborescence

1. Introduction

In this paper we introduce a general algorithm for combinatorial optimization labelled Construct, Merge, Solve & Adapt (CMSA). The proposed algorithm belongs to the class of hybrid metaheuristics [1, 2, 3, 4], which are algorithms that combine components of different techniques for optimization. Examples are combinations of metaheuristics with dynamic programming, constraint programming, and branch & bound. In particular, the proposed algorithm is based on the following general idea. Imagine it were possible to identify a substantially reduced sub-instance of a given problem instance such that the sub-instance contains high-quality solutions to the original problem instance. This would allow applying an exact technique—such as, for example, a mathematical programming solver—with little computational effort to the reduced sub-instance in order to obtain a high-quality solution to the original problem instance. This is for the following reason. For many combinatorial optimization problems the field of mathematical programming—and integer linear programming (ILP) in particular—provides powerful tools; for a comprehensive introduction into this area see, for example, [5]. ILP-solvers are in general based on a tree search framework but further include the solution of linear programming relaxations of a given ILP model for the problem at hand (besides primal heuristics) in order to obtain lower and upper bounds. To tighten these bounds, various kinds of additional inequalities are typically dynamically identified and added as cutting planes to the ILP-model, yielding a branch & cut algorithm. Frequently, such ILP approaches are highly effective for small to medium sized instances of hard problems, even though they often do not scale well enough to large instances relevant in practice. Therefore, in those cases in which a problem instance can be sufficiently reduced, a mathematical programming solver might be very efficient in solving

Email addresses: `christian.blum@ehu.es` (Christian Blum), `ppinacho@santotomas.cl` (Pedro Pinacho), `manuel.lopez-ibanez@manchester.ac.uk` (Manuel López-Ibáñez), `ja.lozano@ehu.es` (José A. Lozano)

the reduced problem instance.

1.1. Related Work

The general idea described above is present in several works from the literature. For example, it is the underlying idea of the general algorithm framework known as Generate-And-Solve (GS) [6, 7, 8, 9]. In fact, our algorithm can be seen as an instantiation of this framework. The GS framework decomposes the original optimization problem into two conceptually different levels. One of the two levels makes use of a component called *Solver of Reduced Instances* (SRI), in which an exact method is applied to sub-instances of the original problem instance that maintain the conceptual structure of the original instance, that is, any solution to the sub-instance is also a solution to the original instance. At the other level, a metaheuristic component deals with the problem of generating sub-instances that contain high quality solutions. In GS, the metaheuristic component is called *Generator of Reduced Instances* (GRI). Feedback is provided from the SRI component to the GRI component, for example, by means of the objective function value of the best solution found in a sub-instance. This feedback serves for guiding the search process of the GRI component.

Even though most existing applications of the GS framework are in the context of cutting, packing and loading problems—see, for example, [7, 8, 9, 10, 11]—other successful applications include the ones to configuration problems arising in wireless networks [12, 13, 14]. Moreover, it is interesting to note that the applications of GS published to date generate sub-instances in the GRI component using either evolutionary algorithms [10, 14] or simulated annealing [11, 13]. Finally, note that in [10] the authors introduced a so-called density control operator in order to control the size of the generated sub-instances. This mechanism can be seen as an additional way of providing feedback from the SRI component to the GRI component.

Apart from the GS framework, the idea of solving reduced problem instances to optimality has also been explored in earlier works. In [15, 16], for example, the authors tackle the classical traveling salesman problem (TSP) by means of a two-phase approach. The first phase consists in generating a bunch of high-quality TSP solutions using a metaheuristic. These solutions are then merged, resulting in a reduced problem instance, which is then solved to optimality by means of an exact solver. In [17] the authors present the following approach for the prize-collecting Steiner tree problem. First, the given problem instance is reduced in such a way that it still contains the optimal solution to the original problem instance. Then, a memetic algorithm is applied to this reduced problem instance. Finally, a mathematical programming solver is applied to find the best solution to the problem instance obtained by merging all solutions of the first and the last population of the memetic algorithm. Massen et al. [18, 19] use an ant colony optimization algorithm to generate a large number of feasible routes for a vehicle routing problem with feasibility constraints, then apply an exact solver to a relaxed set-partitioning problem in order to select a subset of the routes. This subset is used to bias the generation of new routes in the next iteration.

Finally, note that a first, specific, application of the general algorithm proposed in this work has been published in [20] in the context of the minimum weight arborescence problem.

1.2. Contribution of this Work

Even though—as outlined above—there is important related work in the literature, the idea of iteratively solving reduced problem instances to optimality has not yet been explored in an exhaustive manner. In this work we introduce a generally applicable algorithm labelled Construct, Merge, Solve & Adapt (CMSA) for tackling combinatorial optimization problems. The algorithm can be seen as a specific instantiation of the GS framework. It is designed to take profit from ILP solvers such as CPLEX even in the context of large problem instances to which these solvers can not be applied directly. In particular, the main feature of the algorithm is the generation of sub-instances of the original problem instance by repeated probabilistic solution constructions, and the application of an ILP solver to the generated sub-instances. Hereby, the way of generating sub-instances by merging the solution components found in probalistically constructed solutions distinguishes our algorithm from other instantiations of the GS framework from the literature. This feature is actually quite appealing, because our algorithm can easily be applied to any problem for which (1) a constructive heuristic and (2) an exact solver are known.

We consider two test cases for the proposed algorithm: (1) the *minimum common string partition* (MCSP) problem [21], and a *minimum covering arborescence* (MCA) problem, which is an extension of the problem tackled in [20].

For both problems, ILP solvers such as CPLEX are very effective in solving small to medium size problem instances. However, their performance deteriorates (1) in the context of the MCSP problem when the length of the input strings exceeds 600, and (2) in the context of the MCA problem when the number of nodes of the input graph exceeds 1000. We will show that the CMSA algorithm is a new state-of-the-art algorithm for the MCSP problem, especially for benchmark instances for which the application of CPLEX to the original ILP model is not feasible. In the context of the MCA problem we will show that our algorithm is able to match the performance of CPLEX for small and medium size problem instances. Moreover, when large size instances are tackled, the algorithm significantly outperforms a greedy approach.

1.3. Organization of the Paper

The remaining part of the paper is organized as follows. The CMSA algorithm is outlined in general terms in Section 2. The application of this algorithm to the minimum common string partition problem is described in Section 3, whereas its application to the minimum covering arborescence problem is outlined in Section 4. An extensive experimental evaluation is provided in Section 5. Finally, in Section 6 we provide conclusions and an outlook to future work.

2. Construct, Merge, Solve & Adapt

In the following we assume that, given a problem instance $\mathcal{I}$ to a generic problem $\mathcal{P}$, set C represents the set of all possible components of which solutions to the problem instance are composed. C is henceforth called the complete set of solution components with respect to $\mathcal{I}$. Note that, given an integer linear (or non-linear) programming model for problem $\mathcal{P}$, a generic way of defining the set of solution components is to say that each combination of a variable with one of its values is a solution component. Moreover, in the context of this work a valid solution S to $\mathcal{I}$ is represented as a subset of the solution components C, that is, $S \subseteq C$. Finally, set $C' \subseteq C$ contains the solution components that belong to a restricted problem instance, that is, a sub-instance of $\mathcal{I}$. For simplicity reasons, C' will henceforth be called a sub-instance. Imagine, for example, the input graph in case of the TSP. The set of all edges can be regarded as the set of all possible solution components C. Moreover, the edges belonging to a tour S—that is, a valid solution—form the set of solution components that are contained in S.

The Construct, Merge, Solve & Adapt (CMSA) algorithm works roughly as follows. At each iteration, the algorithm deals with the incumbent sub-instance C'. Initially this sub-instance is empty. The first step of each iteration consists in *generating* a number of feasible solutions to the original problem instance $\mathcal{I}$ in a probabilistic way. In the second step, the solution components involved in these solutions are added to C' and an exact solver is applied in order to *solve* C' to optimality. The third step consists in *adapting* sub-instance C' by removing some of the solution components guided by an aging mechanism. In other words, the CMSA algorithm is applicable to any problem for which (1) a way of (probabilistically) generating solutions can be found and (2) a strategy for solving the problem to optimality is known.

In the following we describe the CMSA algorithm, which is pseudo-coded in Algorithm 1, in more detail. The main loop of the proposed algorithm 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 (C') to the empty set. Then, at each iteration a number of n_a solutions is probabilistically generated (see function ProbabilisticSolution-Generation(C) in line 6 of Algorithm 1). The components of all these solutions are added to set C'. The age of a newly added component c (age[c]) is set to 0. After the construction of n_a solutions, an exact solver is applied to find the best solution S'_{opt} in the restricted problem instance C' (see function ApplyExactSolver(C') 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 C' is adapted, based on solution S'_{opt} and on the age values of the solution components. This is done in function Adapt(C', S'_{opt}, age$_{\text{max}}$) in line 14 as follows. First, the age of each solution component in C' is increased by one, and, subsequently, the age of each solution component in $S'_{\text{opt}} \subseteq C'$ is re-initialized to zero. Finally, those solution components from C' whose age has reached the maximum component age (age$_{\text{max}}$) are deleted from C'. The motivation behind the aging mechanism is that components which never appear in an optimal solution of C' should be removed from C' after a while, because they slow down the exact solver. On the other side, components which appear in optimal solutions seem to be useful and should therefore remain in C'. In general, the average size

Algorithm 1 Construct, Merge, Solve & Adapt (CMSA)

```
 1: input: problem instance $\mathcal{I}$, values for parameters $n_a$ and $\text{age}_{\max}$
 2: $S_{\text{bsf}}$ := NULL, $C'$ := $\emptyset$
 3: age[$c$] := 0 for all $c \in C$
 4: while CPU time limit not reached do
 5:     for $i = 1, \ldots, n_a$ do
 6:         $S$ := ProbabilisticSolutionGeneration($C$)
 7:         for all $c \in S$ and $c \notin C'$ do
 8:             age[$c$] := 0
 9:             $C'$ := $C' \cup \{c\}$
10:         end for
11:     end for
12:     $S'_{\text{opt}}$ := ApplyExactSolver($C'$)
13:     if $S'_{\text{opt}}$ is better than $S_{\text{bsf}}$ then $S_{\text{bsf}}$ := $S'_{\text{opt}}$
14:     Adapt($C'$, $S'_{\text{opt}}$, $\text{age}_{\max}$)
15: end while
16: output: $S_{\text{bsf}}$
```

of set C' depends on the parameter values. For example, the higher the value of $\text{age}_{\max}$, the higher the average size of C' during a run of the algorithm. In summary, the behavior of the general CMSA algorithm depends on the values of two parameters: the number of solution construction per iteration (n_a) and the maximum allowed age ($\text{age}_{\max}$) of solution components. Moreover, as long as the mechanism for probabilistically generating solutions has a non-zero probability for generating an optimal solution, the probability to find an optimal solution converges to one with a growing computation time limit. This completes the general description of the algorithm.

3. Application to the MCSP Problem

The MCSP problem is an NP-hard string problem from the bioinformatics field. String problems are very common in bioinformatics. This family of problems includes, among others, string consensus problems such as the far-from most string problem [22, 23], the longest common subsequence problem and its variants [24, 25], and alignment problems [26]. These problems are often computationally very hard, if not even NP-hard [27].

The MCSP problem can technically be described as follows. Given are two input strings s^1 and s^2 of length n over a finite alphabet Σ. The two strings are *related*, which means that each letter appears the same number of times in each of them. This definition implies that s^1 and s^2 have the same length n. A valid solution to the MCSP 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 $P_1 = P_2$. Moreover, the goal is to find a valid solution such that $|P_1| = |P_2|$ is minimal. Consider the following example. Given are DNA sequences s^1 = **AGACTG** and s^2 = **ACTAGG**. As **A** and **G** appear twice in both input strings, and **C** and **T** appear once, the two strings are related. A trivial valid solution can be obtained by partitioning both strings into substrings of length 1, that is, $P_1 = P_2 = \{\mathbf{A}, \mathbf{A}, \mathbf{C}, \mathbf{T}, \mathbf{G}, \mathbf{G}\}$. The objective function value of this solution is 6. However, the optimal solution, with objective function value 3, is $P_1 = P_2 = \{\mathbf{ACT}, \mathbf{AG}, \mathbf{G}\}$.

The MCSP problem was introduced by Chen et al. [21] due to its relation to genome rearrangement. More specifically, it has applications in biological questions such as: May a given DNA string possibly be obtained by rearrangements of another DNA string? The general problem has been shown to be NP-hard even in very restrictive cases [28]. Approximation algorithms are described, for example, in [29]. Recently, Goldstein and Lewenstein [30] proposed a greedy algorithm for the MCSP problem that runs in $O(n)$ time. He [31] introduced a greedy algorithm with the aim of obtaining better average results. To our knowledge, the only metaheuristic approaches that have been proposed in the related literature for the MCSP problem are (1) the $\mathcal{MAX}$-$\mathcal{MIN}$ Ant System by Ferdous and Sohel Rahman [32, 33] and (2) the probabilistic tree search algorithm by Blum et al. [34]. In these works the proposed algorithm is applied to a range of artificial and real DNA instances from [32]. Finally, the first ILP model for the

Algorithm 2 Probabilistic Solution Generation (MCSP problem)

```
 1: input: $s^1$, $s^2$, $d_{\text{rate}}$, $l_{\text{size}}$
 2: $S := \emptyset$
 3: while $Ext(S) \neq \emptyset$ do
 4:    Choose a random number $\delta \in [0, 1]$
 5:    if $\delta \leq d_{\text{rate}}$ then
 6:       Choose $c^*$ such that $|t_{c^*}| \geq |t_c|$ for all $c \in Ext(S)$
 7:       $S := S \cup \{c^*\}$
 8:    else
 9:       Let $L \subseteq Ext(S)$ contain the (at most) $l_{\text{size}}$ longest common blocks from $Ext(S)$
10:       Choose $c^*$ uniformly at random from $L$
11:       $S := S \cup \{c^*\}$
12:    end if
13: end while
14: output: The complete (valid) solution $S$
```

MCSP problem, together with an ILP-based heuristic, was proposed in [35].

The remainder of this section describes the application of the CMSA algorithm presented in the previous section to the MCSP. For this purpose we define the set C of solution components and the structure of *valid subsets* of C as follows. Henceforth, a *common block* c_i of 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 $1 \leq k_i^1 \leq n$ in string s^1 and starting at position $1 \leq k_i^2 \leq n$ in string s^2. Moreover, let $C = \{c_1, \ldots, c_m\}$ be the arbitrarily ordered set of all possible common blocks of s^1 and s^2, i.e., C is the set of all solution components. Given the definition of C, a subset S of C is called a valid subset iff the following conditions hold:

1. $\sum_{c_i \in S} |t_i| \leq n$, that is, the sum of the length of the strings corresponding to the common blocks in S is smaller or equal to the length of the input strings.
2. For any two common blocks $c_i, c_j \in S$ it holds that their corresponding strings neither overlap in s^1 nor in s^2.

Given a valid subset $S \subset C$, set $Ext(S) \subset C \setminus S$ denotes the set of common blocks that may be used in order to extend S such that the result is again a valid subset. Note that in case $Ext(S) = \emptyset$ it necessarily holds that $\sum_{c_i \in S} |t_i| = n$. In this case S is a valid subset which corresponds to a complete (valid) solution to the problem.

3.1. *Probabilistic Solution Generation*

Next we describe the implementation of function ProbabilisticSolutionGeneration(C) in line 6 of Algorithm 1. The construction of a complete (valid) solution (see Algorithm 2) starts with the empty subset $S := \emptyset$. At each construction step, a solution component c^* from $Ext(S)$ is chosen and added to S. This is done until $Ext(S) = \emptyset$. The choice of c^* is done as follows. First, a value $\delta \in [0, 1]$ is chosen uniformly at random. In case $\delta \leq d_{\text{rate}}$, c^* is chosen such that $|t_{c^*}| \geq |t_c|$ for all $c \in Ext(S)$, that is, one of the common blocks whose substring is of maximal size is chosen. Otherwise, a candidate list L containing the l_{size} longest common blocks from $Ext(S)$ is built, and c^* is chosen from L uniformly at random (ties are broken randomly). In case the number of remaining blocks in $Ext(S)$ is lower than l_{size}, all the blocks are selected. 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.

3.1.1. *Solving Reduced Sub-Instances*

The last component of Algorithm 1 which remains to be described is the implementation of function ApplyExact-Solver(C') in line 12. In the case of the MCSP problem we make use of the ILP model proposed in [35] and the ILP solver CPLEX for solving it. The model for the complete set C of solution components can be described as follows. First, two binary $m \times n$ matrices M^1 and M^2 are defined. In both matrices, row $1 \leq i \leq m$ corresponds to common block $c_i \in C$. Moreover, a column $1 \leq j \leq n$ corresponds to position j in input string s^1, respectively s^2. In general, the entries of matrix M^1 are set to zero. However, in each row i, the positions that string t_i (of common block c_i) occupies in input string s^1 are set to one. Correspondingly, the entries of matrix M^2 are set to zero, apart from the fact that in each row i the positions occupied by string t_i in input string s^2 are set to one. Henceforth, the position (i, j) of a matrix M is denoted by $M_{i,j}$. Finally, we introduce for each common block $c_i \in C$ a binary variable x_i. With these definitions the MCSP can be expressed in terms of the following ILP model.

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

subject to:

$$\sum_{i=1}^{m} M^1_{i,j} \cdot x_i = 1 \quad \text{for } j = 1, \ldots, n \tag{2}$$

$$\sum_{i=1}^{m} M^2_{i,j} \cdot x_i = 1 \quad \text{for } j = 1, \ldots, n \tag{3}$$

$$x_i \in \{0, 1\} \quad \text{for } i = 1, \ldots, m$$

The objective function minimizes the number of selected common blocks. Constraints (2) make sure that the strings corresponding to the selected common blocks do not overlap in input string s^1, while constraints (3) make sure that the strings corresponding to the selected common blocks do not overlap in input string s^2. The condition that the length of the strings corresponding to the selected common blocks is equal to n is implicitly obtained from these two constraint sets.

As an example, let us consider the small problem instance that was mentioned at the start of Section 3. The complete set of common blocks (C), as induced by input strings s^1 = **AGACTG** and s^2 = **ACTAGG**, is as follows:

$$C = \begin{Bmatrix} c_1 = (\mathbf{ACT}, 3, 1) \\ c_2 = (\mathbf{AG}, 1, 4) \\ c_3 = (\mathbf{AC}, 3, 1) \\ c_4 = (\mathbf{CT}, 4, 2) \\ c_5 = (\mathbf{A}, 1, 1) \\ c_6 = (\mathbf{A}, 1, 4) \\ c_7 = (\mathbf{A}, 3, 1) \\ c_8 = (\mathbf{A}, 3, 4) \\ c_9 = (\mathbf{C}, 4, 2) \\ c_{10} = (\mathbf{T}, 5, 3) \\ c_{11} = (\mathbf{G}, 2, 5) \\ c_{12} = (\mathbf{G}, 2, 6) \\ c_{13} = (\mathbf{G}, 6, 5) \\ c_{14} = (\mathbf{G}, 6, 6) \end{Bmatrix}$$

Given set C, matrices M^1 and M^2 are the following ones:

$$M^1 = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad M^2 = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

The optimal solution to this instance is $S^* = \{c_1, c_2, c_{14}\}$. It can easily be verified that this solution respects constraints (2) and (3) of the ILP model.

Note that this ILP model can also be solved for any subset C' of C. This is achieved by replacing all occurrences of C with C', and by replacing m with $|C'|$. The solution of such an ILP corresponds to a feasible solution to the original problem instance as long as C' contains at least one feasible solution to the original problem instance. Due to the way in which C' is generated (see Section 3.1) this condition is fulfilled.

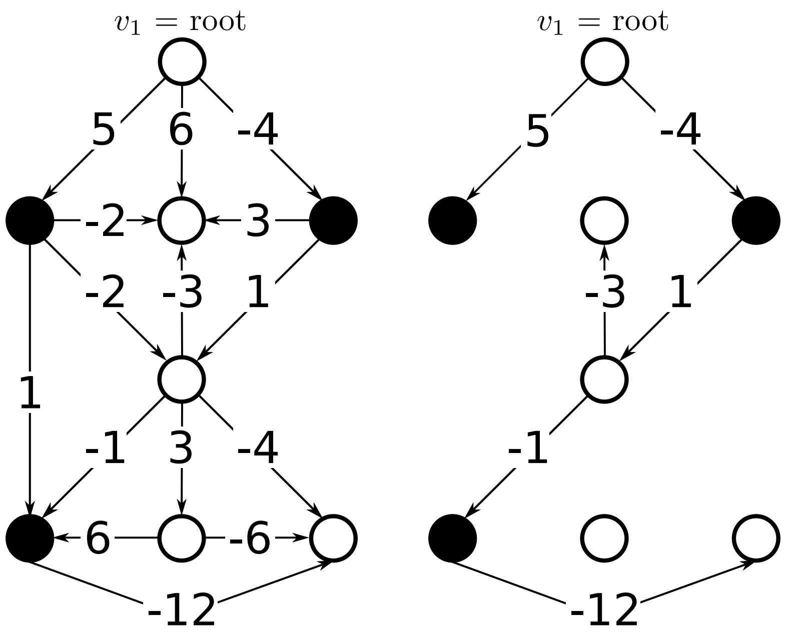
4. Application to the MCA Problem

The MCA problem considered in this section belongs to the family of *minimum weight rooted arborescence* (MWRA) problems [36]. In this type of problem we are given a directed (acyclic) graph with integer weights on the arcs. In some of these problems the weight values might be restricted to be positive, while in other problems positive and negative weights are allowed. Valid solutions to such a problem correspond to subgraphs of the input graph that are arborescences rooted in the pre-defined root node. In this context, a rooted arborescence is a directed, rooted (not necessarily spanning) tree in which all arcs point away from the root node (see [37]). The goal is to find, among all valid solutions, one with minimal weight. Hereby, the weight of an arborescence is defined as the sum of the weights of its arcs. These type of problems have applications, for example, in computer vision and in multistage production planning.

The specific problem tackled in this work—henceforth called *minimum covering arborescence* (MCA) problem—is an extension of the MWRA problem considered in [20] and the minimum covering arborescence problem described on page 535 of [38]. The MCA problem is formally defined as follows. Given is a directed acyclic graph (DAG) denoted by $G = (V, A)$. Hereby, $V = \{v_1, \ldots, v_n\}$ is the set of n nodes and $A \subseteq \{(i, j) \mid i \neq j \in V\}$ is a set of m directed arcs. Without loss of generality it is assumed that v_1 is the designated root node. Each arc $a \in A$ has assigned an integer weight $w(a) \in \mathbb{Z}$. Moreover, a pre-defined subset $X \subseteq V$ of the nodes of the input graph must be included in a valid solution. Any arborescence $T = (V(T), A(T))$—where $V(T) \subseteq V$ is the node set of T and $A(T) \subseteq A$ is the arc set of T—rooted in v_1 with $X \subseteq V(T)$ is a valid solution to the problem. Let $\mathcal{A}$ be the set of all such arborescences. The objective function value (that is, the weight) $f(T)$ of an arborescence $T \in \mathcal{A}$ is defined as follows:

$$f(T) := \sum_{a \in A(T)} w(a) \ . \tag{4}$$

The goal of the MCA problem is to find an arborescence $T^* \in \mathcal{A}$ such that the weight of T^* is smaller or equal to the weight of any arborescence in $\mathcal{A}$. In other words, the goal is to minimize objective function $f(\cdot)$. An example of the MCA problem is shown in Figure 1. As the problem version in which $X = \emptyset$ is already NP-hard [20], the more general problem in which $X \neq \emptyset$ is also NP-hard. An example for the MCA problem is shown in Figure 1.



(a) Example input graph. (b) Opt. solution of value -14.

Figure 1: (a) shows an input graph with eight nodes and 15 arcs. The uppermost node is the root node v_1. Moreover, the nodes colored in black form set X, that is, they must be included in any valid solution. (b) shows the optimal solution with value -14.

The remainder of this section describes the application of the CMSA algorithm to the MCA problem. For this purpose we define the set of solution components and the structure of valid subsets of the complete set of solution components as follows. First, in the case of the MCA problem, the complete set of solution components corresponds to the set A of arcs of the input graph, that is, $C := A$. However, for the sake of maintaing the readability of the following description of the algorithm components we continue to use notation A instead of C. Second, a subset S of A is called a valid subset iff $T = (V(S), S)$ is an arborescence of the input graph G rooted in v_1. Hereby, $V(S) \subseteq V$ refers to the subset of nodes that is obtained by joining all the heads and tails of the arcs in S. Given a valid subset $S \subset A$, $Ext(S) \subset A \setminus S$ refers to all arcs that can be added to S such that the result is again a valid subset. More in detail, $Ext(S) := \{a = (v_i, v_j) \in A \mid v_i \in V(S), v_j \in V \setminus V(S)\}$. In the special case of $S = \emptyset$, $Ext(S) := \text{Out}(v_1)$, where $\text{Out}(v)$, given $v \in V$, denotes the set of outgoing arcs of v, that is, the set of arcs that have v as *tail*. In the same way, $\text{In}(v)$ denotes the set of incoming arcs of v, that is, the set of arcs that have v as *head*. Finally, a valid subset corresponds to a (valid) solution to the problem in case $X \subseteq V(S)$.

4.1. Probabilistic Solution Generation

Next, the implementation of function ProbabilisticSolutionGeneration(C) in line 6 of Algorithm 1 is described. The pseudo-code of this procedure is outlined in Algorithm 3. Starting from the root node v_1, at each step an arc—that is, a solution component—is chosen from set $\hat{A}$ (see lines 3 and 7 of Algorithm 3). For the choice of the first solution component in line 3, $\hat{A}$ is defined as the set of outgoing arcs of the root node v_1. For all further construction steps, $\hat{A}$ is defined as $Ext(S)$. However, instead of considering the whole set of arcs connecting one of the nodes of the current arborescence with one of the remaininig nodes, function Reduce($\hat{A}$) is applied before choosing one of the arcs from $\hat{A}$ (see line 8). This function chooses from each set $\{(v_j, v_i) \mid v_j \in V(S)\} \subseteq \hat{A}$, for all $v_i \in V \setminus V(S)$, the arc with minimal weight. The chosen arc remains in $\hat{A}$, while the other ones are deleted. In other words, if a node $v_i \in V \setminus V(S)$ may be connected via several arcs with the current arborescence $T = (V(S), S)$, only the arc with minimal weight is considered. Finally, the process of constructing a solution finishes when $\hat{A} = \emptyset$, that is, when all nodes are already included in the constructed arborescence. In principle, the construction process could already be stopped once all

Algorithm 3 Probabilistic Solution Generation (MCA problem)

```
1:  input: a DAG $G = (V, A)$ with root node $v_1$, $d_{\min}$, $d_{\max}$
2:  $S := \emptyset$
3:  $\hat{A} :=$ Out$(v_1)$
4:  while $\hat{A} \neq \emptyset$ do
5:      $a^* :=$ Choose$(\hat{A}, d_{\min}, d_{\max})$
6:      $S := S \cup \{a^*\}$
7:      $\hat{A} := Ext(S)$
8:      $\hat{A} :=$ Reduce$(\hat{A})$
9:  end while
10: output: valid subset $S$ which induces arborescence $T = (V(S), S)$
```

nodes from X are included in the constructed arborescence. However, experimental tests have shown that generating spanning arborescences leads, overall, to better results.

The choice of an arc from $\hat{A}$ is done in function Choose($\hat{A}$, $d_{\min}$, $d_{\max}$)—see line 5 of the pseudo-code—based on heuristic information. The heuristic information $\eta(a)$ of an arc $a \in \hat{A}$—which will be used below in Eq. (7)—is computed as follows. First, let

$$w_{\max} := \max\{w(a) \mid a \in A\}. \tag{5}$$

Based on this maximal weight of all arcs in G, the heuristic information is defined as follows:

$$\eta(a) := w_{\max} + 1 - w(a) \tag{6}$$

In this way, the heuristic information of all arcs is a positive integer number. Moreover, the arc with minimal weight has the highest heuristic value.

Given the current valid subset S—corresponding to arboresence $T = (V(S), S)$—and the non-empty set of arcs $\hat{A}$ that may be used for extending S, the probability for choosing arc $a \in \hat{A}$ is defined as follows:

$$\mathbf{p}(a \mid S) := \frac{\eta(a)}{\sum_{a' \in \hat{A}} \eta(a')} \tag{7}$$

At the start of each arborescence construction, a so-called *determinism rate* δ is chosen uniformly at random from $[d_{\min}, d_{\max}]$, where $0 \leq d_{\min} \leq d_{\max} \leq 1$. The chosen value for δ is then used during the arborescence construction as follows. At each construction step, first, a value $r \in [0, 1]$ is chosen uniformly at random. Second, in case $r \leq \delta$, the arc $a^* \in \hat{A}$ with the maximum probability is deterministically chosen, that is: $a^* := \text{argmax}_{a \in \hat{A}}\{\mathbf{p}(a \mid S)\}$. Otherwise, that is, when $r > \delta$, arc $a^* \in \hat{A}$ is chosen probabilistically according to the probability values.

4.1.1. Solving Reduced Sub-Instances

The last component of Algorithm 1 which remains to be described is the implementation of function ApplyExact-Solver(C') in line 12. In the case of the MCA problem we make use of the following ILP model, which is a slight modification of models that can be found in [20, 39, 40] for related problems. The model works on an augmented graph $G^+ = (V^+ := V \cup \{v_0\}, A^+ := A \cup \{(v_0, v_1)\})$, where v_0 is an additional dummy node and (v_0, v_1) is a dummy arc connecting v_0 with the root node v_1. The weight $w(v_0, v_1)$ of arc (v_0, v_1) is zero. Henceforth, let $Pred(a) \in A^+$ denote for each $a \in A$ the set of predecessor arcs, that is, the set of arcs pointing to the tail of arc a. The ILP model works on a set of binary variables which contains for each arc $a \in A^+$ a binary variable $x_a \in \{0, 1\}$. The ILP itself can then be stated as follows.

$$\textbf{min} \sum_{a \in A} w(a) \cdot x_a \tag{8}$$

subject to:

$$\sum_{a \in In(v_i)} x_a \leq 1 \quad \text{for } v_i \in V \setminus (X \cup \{v_1\}) \tag{9}$$

$$\sum_{a \in In(v_i)} x_a = 1 \quad \text{for } v_i \in X \tag{10}$$

$$x_a - \sum_{a' \in Pred(a)} x_{a'} \leq 0 \quad \text{for } a \in A \tag{11}$$

$$x_{(v_0,v_1)} = 1 \tag{12}$$

$$x_a \in \{0,1\} \quad \text{for } a \in A^+$$

Hereby, constraints (9) ensure that for each node $v_i \in V \setminus (X \cup \{v_1\})$ (that is, all nodes of the original graph without the nodes from X and the root node) at most one incoming arc is chosen to form part of the arborescence. For all nodes in X, constraints (10) make sure that exactly one incoming arc is chosen. Constraints (11) ensure that if an arc a from the original graph is chosen for the arborescence, then also one predecessor arc of the tail of a must be chosen for the arborescence. Finally, constraint (12) forces the arborescence to start in dummy arc (v_0, v_1), which means that v_1 is forced to be the root node of the arborescence in the original graph G.

This ILP model can also be solved for any subgraph G' of G which is, itself, a DAG with root node v_1. Note that set C' (see Algorithm 1) in case of the MCA problem induces such a subgraph. The optimal solution to such a *reduced* ILP corresponds to a feasible solution to the original problem instance as long as G' contains at least one feasible solution to the original problem instance. Due to the way in which C' is generated (see Section 4.1) this condition is fulfilled.

5. Experimental Evaluation

The proposed applications of CMSA to the MCSP problem and the MCA problem were implemented in ANSI C++ using GCC 4.7.3 for compiling the software. Moreover, both the complete ILP models and the reduced ILP models within CMSA were solved with IBM ILOG CPLEX 12.1. The experimental evaluation was conducted on a cluster of 32 PCs with Intel(R) Xeon(R) X5660 CPUs with 2 cores at 2.8 GHz and 48 Gigabytes of RAM.

5.1. Experiments Concerning the MCSP Problem

The following algorithms were considered for the comparison: Greedy, the greedy approach from [31]; TreSea, the probabilistic tree search approach from [34]; $\text{Ilp}_{\text{compl}}$, the application of CPLEX to the complete ILP for each considered problem instance; HeurIlp, the application of an ILP-based heuristic from [35];[1] and Cmsa, our proposed CMSA approach. Moreover, in the context of the existing benchmark instances from the literature a comparison to the ant colony optimization approach from [32, 33] (labelled Aco) is also included.

5.1.1. Benchmark Instances

Both existing as well as new benchmark instances were used for the experimental evaluation. As a first benchmark set we chose the one that was introduced by Ferdous and Sohel Rahman [32] for the experimental evaluation of their ant colony optimization approach. This set contains, in total, 30 artificial instances and 15 real-life instances consisting of DNA sequences, that is, the size of the alphabet is four. Remember, in this context, that each problem instance consists of two related input strings. Moreover, the benchmark set is divided into four subsets of instances. The first subset (henceforth labelled Group1) consists of 10 artificial instances in which the input strings are maximally of

[1] HeurIlp has a parameter l that needs to be given a value. In our experiments we chose $l = \min\{5, l_{\max}\}$, where $l_{\max}$ denotes the length of the longest common block. This is following a suggestion of the authors of [35].

length 200. The second set (Group2) consists of 10 artificial instances with input string lengths between 201 and 400. In the third set (Group3) the input strings of the 10 artificial instances have lengths between 401 and 600. Finally, the fourth set (Real) consists of 15 real-life instances of various lengths between 200 and 600.

The second benchmark set that we used is new. It consists of 20 randomly generated instances for each combination of $n \in \{200, 400, \ldots, 1800, 2000\}$, where n is the length of the input strings, and alphabet size $|\Sigma| \in \{4, 12, 20\}$. 10 of these instances are generated with an equal probability for each letter of the alphabet. More specifically, the probability for each letter $l \in \Sigma$ to appear at a certain position of the input strings is $\frac{1}{|\Sigma|}$. The resulting set of 300 benchmark instances of this type are labelled Linear. The other 10 instances per combination of n and $|\Sigma|$ are generated with a probability for each letter $l \in \Sigma$ to appear at a certain position of the input strings of $l / \sum_{i=1}^{|\Sigma|} i$. The resulting set of 300 benchmark instances of this second type are labelled Skewed.

5.1.2. Tuning of Cmsa *and* TreSea

Cmsa has several parameters 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 [41] for the tuning of the five parameters. In fact, irace was applied to tune Cmsa separately for each instance size from $\{200, 400, \ldots, 1800, 2000\}$. For each of these 10 different instance sizes we generated 12 *training instances* for tuning: two instances of type Linear and two instances of type Skewed for each alphabet size from $\{4, 12, 20\}$. The tuning process for each instance 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:

- $n_a \in \{10, 30, 50\}$
- $\text{age}_{\max} \in \{1, 5, 10, \mathit{inf}\}$, where *inf* means that solution components are never removed from C'.
- $d_{\text{rate}} \in \{0.0, 0.5, 0.9\}$, where a value of 0.0 means that the selection of solution component c^* (see line 6 of Algorithm 2) is always done randomly from the candidate list, while a value of 0.9 means that solution constructions are nearly deterministic.
- $l_{\text{size}} \in \{3, 5, 10\}$
- $t_{\max} \in \{60, 120, 240, 480\}$ (in seconds)

The 10 applications of irace produced the 10 configurations of Cmsa shown in Table 1a. The following trends can be observed. First of all, with growing instance size, more time ($t_{\max}$) should be given to individual applications of CPLEX to sub-instances of the original problem instance. Second, irrespective of the instance size, candidate list sizes (l_{size}) smaller than five seem to be too restrictive. Third, also irrespective of the instance size, less than 30 solution constructions per iteration (n_a) seem to be insufficient. Presumably, when only few solution constructions per iteration are performed, the resulting change in the corresponding sub-instances is not large enough and, therefore, some applications of CPLEX result in wasted computation time. Finally, considering the obtained values of d_{rate} for instance sizes from 200 to 1600, the trend is that with growing instance size the degree of greediness in the solution construction should grow. However, the settings of d_{rate} for $n \in \{1800, 2000\}$ is not in accordance with this observation.

In addition to tuning experiments for Cmsa, we also performed tuning experiments for TreSea. In fact, TreSea constructs solutions in the same way in which they are constructed in Cmsa. The parameters involved in TreSea are, therefore, d_{rate} and l_{size}. For the tuning of TreSea we used the same training instances, the same budget of 1000 runs, and the same parameter value ranges as for the tuning of Cmsa. The obtained parameter values per instance size are displayed in Table 1b.

Table 1: Parameter settings produced by irace for the 10 different instance sizes.

(a) Tuning results for Cmsa

n	n_a	age_{max}	d_{rate}	l_{size}	t_{max}
200	50	*inf*	0.0	10	60
400	50	10	0.0	10	60
600	50	10	0.5	10	60
800	50	10	0.5	10	240
1000	50	10	0.9	10	480
1200	50	10	0.9	10	480
1400	50	*inf*	0.9	5	480
1600	50	5	0.9	10	480
1800	30	10	0.5	5	480
2000	50	10	0.0	10	480

(b) Tuning results for TreSea

n	d_{rate}	l_{size}
200	0.9	5
400	0.9	3
600	0.9	10
800	0.5	5
1000	0.5	5
1200	0.5	3
1400	0.5	5
1600	0.0	5
1800	0.0	10
2000	0.0	10

5.1.3. *Results*

In the following we present the experimental results for the two benchmark data sets described in Section 5.1.1, which are different from the data sets used for tuning the algorithms.

The first benchmark set, as outlined above, consists of four subsets of instances labelled Group1, Group2, Group3, and Real. The results for these groups of instances are shown in the four Tables 2a–2d. The structure of these four tables is as follows. The first column provides the instance identifiers. The second column contains the results of Greedy. The third column provides the value of the best solution found in four independent runs per problem instance (with a CPU time limit of 7200 seconds per run) by Aco; results are taken from [32, 33]. The fourth column contains the value of the best solution found in 10 independent runs per problem instance (with a CPU limit of 3600 seconds per run) by TreSea. The next three table columns are dedicated to the presentation of the results provided by solving the complete ILP model $\text{Ilp}_{\text{compl}}$. The first one of these columns provides the value of the best solution found within 3600 CPU seconds. The second column provides the computation time (in seconds). In case of having solved the corresponding problem to optimality, this column only displays one value indicating the time needed by CPLEX to solve the problem. Otherwise, this column provides two values in the form X/Y, where X corresponds to the time at which CPLEX was able to find the first valid solution, and Y corresponds to the time at which CPLEX found the best solution within 3600 CPU seconds. Finally, the third one of the columns dedicated to $\text{Ilp}_{\text{compl}}$ shows the optimality gap, which refers to the gap between the value of the best valid solution and the current lower bound at the time of stopping a run. The next two columns display the results of the ILP-based, deterministic heuristic HeurIlp. The first column contains the results, and the second column the computation time. Finally, the last three columns of each table are dedicated to the presentation of the results obtained by Cmsa. The first column provides the value of the best solutions found by Cmsa within 3600 CPU seconds. The second column provides the average (mean) results over 10 independent runs per problem instance. The last column indicates the average time needed by Cmsa to find the best solution of a run. The best result for each problem instance is marked by a grey background and the last row of each table provides averages over the whole table.

Analyzing the results it can be observed that the results of Cmsa are very similar to the ones of applying CPLEX to $\text{Ilp}_{\text{compl}}$. In fact, the application of the non-parametric Wilcoxon test for all four instance subsets did not reveal differences of statistical significance between both techniques (for an α-value of 0.05). In comparison to the other techniques (Greedy, Aco, TreSea and HeurIlp) both Cmsa and the application of CPLEX to $\text{Ilp}_{\text{compl}}$ significantly outperform the competitors.

As described in Section 5.1.1, the second benchmark set which was specifically generated for this paper, consists of 300 instances of type Linear and another 300 instances of type Skewed. The results for instances of type Linear are presented in the three Tables 3a–3c, in terms of one table per alphabet size. In contrast to the first benchmark set, for which the probabilistic algorithms such as TreSea and Cmsa were applied for 10 independent runs, results for instances of type Linear and Skewed are presented in these tables in terms of averages over 10 random instances

Table 2: Results for the instances of the first benchmark set (consisting of Group1, Group2, Group3 and Real).

(a) Results for Group1.

id	Greedy	Aco	TreSea	Ilp$_{\text{compl}}$			HeurIlp		Cmsa		
	value	value	value	value	time	gap	value	time	best	mean	time
1	46	42	42	41	1	0.0%	42	< 1	41	41.0	2
2	54	51	48	47	3	0.0%	48	< 1	47	47.0	6
3	60	55	55	52	30	0.0%	54	< 1	52	52.0	298
4	46	43	43	41	2	0.0%	43	< 1	41	41.0	23
5	44	43	41	40	1	0.0%	43	< 1	40	40.0	2
6	48	42	41	40	3	0.0%	41	< 1	40	40.0	2
7	64	60	59	55	38	0.0%	59	< 1	56	56.0	29
8	47	47	45	43	3	0.0%	44	< 1	43	43.0	1027
9	42	45	43	42	2	0.0%	48	< 1	42	42.0	28
10	63	59	58	54	50	0.0%	58	< 1	54	54.0	36
avg.	51.4	48.7	47.5	45.5	13.3	0.0%	48.0	< 1	45.6	45.6	145

(b) Results for Group2.

id	Greedy	Aco	TreSea	Ilp$_{\text{compl}}$			HeurIlp		Cmsa		
	value	value	value	value	time	gap	value	time	best	mean	time
1	118	113	111	98	66/1969	2.9%	108	3	101	101.2	2045
2	121	118	114	106	129/1032	7.5%	111	2	104	104.6	1677
3	114	111	107	97	55/1216	2.7%	105	3	97	97.1	1883
4	116	115	110	102	63/949	4.9%	111	3	102	102.5	1187
5	132	132	127	116	146/3299	6.7%	125	4	117	117.8	1581
6	107	105	102	93	56/1419	5.6%	101	< 1	94	95.4	1587
7	106	98	95	88	41/2776	6.0%	96	2	88	89.0	2103
8	122	118	114	104	101/2980	5.1%	116	2	103	105.2	1858
9	123	119	113	104	81/1630	5.2%	112	2	104	104.9	2010
10	102	101	97	89	32/1458	3.6%	94	3	89	89.8	1550
avg.	116.1	113.0	109.0	99.7	77/1873	5.0%	107.9	2	99.9	100.8	1748

(c) Results for Group3.

id	Greedy	Aco	TreSea	Ilp$_{\text{compl}}$			HeurIlp		Cmsa		
	value	value	value	value	time	gap	value	time	best	mean	time
1	181	177	171	155	733/1398	7.5%	173	5	157	157.9	1842
2	173	175	168	155	553/869	7.7%	165	9	156	157.5	1702
3	195	187	185	166	746/2183	8.5%	180	6	166	167.3	1805
4	191	184	179	159	731/1200	6.9%	171	15	160	161.8	2057
5	174	171	162	150	485/886	9.7%	164	4	149	151.1	1224
6	169	160	162	147	399/764	9.1%	155	4	148	149.3	2027
7	171	167	159	149	524/990	9.8%	160	4	146	147.8	2265
8	185	175	170	151	492/3584	6.7%	166	7	153	154.2	1790
9	174	172	169	158	571/1186	10.9%	169	5	154	155.3	2468
10	171	167	160	148	547/1446	9.1%	160	4	148	149.0	1768
avg.	178.4	173.5	168.5	153.8	578/1451	8.6%	166.3	6	153.7	155.1	1895

(d) Results for Real.

id	Greedy	Aco	TreSea	Ilp$_{\text{compl}}$			HeurIlp		Cmsa		
	value	value	value	value	time	gap	value	time	best	mean	time
1	93	87	86	78	972	0.0%	85	< 1	78	78.9	1192
2	160	155	153	139	432/752	9.2%	150	3	138	140.0	1960
3	119	116	113	104	125/3580	5.6%	112	2	103	104.7	1126
4	171	164	156	144	577/1730	6.5%	158	6	143	143.7	2037
5	172	171	166	150	778/2509	7.9%	161	5	151	152.9	1557
6	153	145	143	128	257/3578	6.5%	139	3	126	127.6	1469
7	135	140	131	121	359/2187	6.9%	132	2	122	122.7	1657
8	133	130	128	116	275/3365	6.8%	123	3	118	118.4	1576
9	149	146	142	131	399/613	8.8%	139	2	130	130.7	1790
10	151	148	143	131	311/1771	7.2%	144	3	131	131.7	1500
11	124	124	120	110	205/3711	4.8%	122	2	111	111.9	1658
12	143	137	138	126	299/793	9.8%	136	2	126	127.5	1903
13	180	180	172	156	784/1130	7.1%	171	5	158	158.6	2066
14	150	147	146	134	370/2456	9.7%	147	6	133	134.0	1789
15	157	160	152	139	560/1762	7.7%	148	3	141	141.7	1424
avg.	146	143.3	139.3	127.1	409/2131	7.0%	137.8	3	127.3	128.3	1647

Table 3: Results for the instances of set Linear.

(a) Results for instances with $\Sigma = 4$.

n	Greedy	TreSea	Ilp$_{\text{compl}}$			HeurIlp		Cmsa	
	mean	**mean**	**mean**	**time**	**gap**	**mean**	**time**	**mean**	**time**
200	75.0	68.7	63.5	4/104	0.0%	69.0	< 1	63.7	608
400	133.4	126.1	115.7	108/2081	6.8%	124.3	3	116.4	1381
600	183.7	177.5	162.2	513/1789	9.4%	174.1	8	162.9	1918
800	241.1	232.7	246.8	1671/1671	23.8%	229.1	17	212.4	2434
1000	287.0	280.4	n/a	n/a	n/a	277.2	1095	256.9	2623
1200	333.8	330.4	n/a	n/a	n/a	324.8	1803	303.3	2369
1400	385.5	378.9	n/a	n/a	n/a	373.1	1807	351.0	2452
1600	432.3	427.1	n/a	n/a	n/a	416.7	1811	400.6	1973
1800	477.4	474.2	n/a	n/a	n/a	464.4	1813	445.4	2194
2000	521.6	520.7	n/a	n/a	n/a	512.7	1820	494.0	1744
avg.	307.1	301.7	n/a	n/a	n/a	296.5	1018	280.7	1969

(b) Results for instances with $\Sigma = 12$.

n	Greedy	TreSea	Ilp$_{\text{compl}}$			HeurIlp		Cmsa	
	mean	**mean**	**mean**	**time**	**gap**	**mean**	**time**	**mean**	**time**
200	127.3	122.1	119.2	1/1	0.0	123.0	< 1	119.2	22
400	228.9	223.5	208.9	7/51	0.0	215.7	6	209.4	892
600	322.2	318.7	291	47/1277	0.9	296.2	691	293.8	1433
800	411.4	408.1	368.7	147/2405	1.6	373.9	1546	373.2	1484
1000	499.2	494.9	453.4	395/2084	3.8	452.0	1802	449.9	2651
1200	586.0	585.6	536.6	784/3188	4.7	542.4	1803	531.0	2318
1400	666.0	664.6	684.1	1667/1667	15.8	653.3	1864	606.9	2467
1600	754.4	754.6	773.5	2648/2648	16.0	749.7	2045	694.8	2392
1800	827.3	833.0	n/a	n/a	n/a	850.7	3301	773.6	1484
2000	913.5	916.2	n/a	n/a	n/a	939.6	5052	849.6	2967
avg.	533.6	532.1	n/a	n/a	n/a	519.7	1811	490.1	1811

(c) Results for instances with $\Sigma = 20$.

n	Greedy	TreSea	Ilp$_{\text{compl}}$			HeurIlp		Cmsa	
	mean	**mean**	**mean**	**time**	**gap**	**mean**	**time**	**mean**	**time**
200	149.2	146.6	146.2	1/1	0.0%	146.4	< 1	146.2	2
400	274.5	268.8	261.5	2/2	0.0%	263.8	< 1	261.9	80
600	389.2	383.5	362.3	10/15	0.0%	369.3	4	366.6	364
800	495.8	492.3	456.1	43/700	0.0%	464.7	121	463.1	804
1000	600.6	597.5	547.1	125/1737	0.6%	562.5	205	555.0	529
1200	706.1	707.8	642.2	296/2732	1.3%	658.8	415	648.5	1372
1400	801.1	804.0	737.9	559/2314	3.1%	745.7	812	737.7	2334
1600	899.8	903.1	861.3	966/2885	6.6%	872.6	1015	825.7	2251
1800	996.8	1000.1	1012.9	1559/1845	12.6%	994.4	1336	917.6	2437
2000	1097.8	1102.6	1136.0	2349/2349	14.4%	1120.7	1773	1024.9	2924
avg.	641.1	640.6	616.35	591/1458	3.9%	619.9	568	594.4	1310

of the same characteristics. Each algorithm included in the comparison was applied exactly once to each problem instance. Note that in addition to different alphabet sizes ($|\Sigma| \in \{4, 12, 20\}$) this second benchmark set also contains much larger instances than the first benchmark set (input strings with a length of up to $n = 2000$).

The analysis of the results permits to draw the following conclusions:

- Surprisingly, hardly any differences can be observed in the relative performance of the algorithms for instances of type Linear and instances of type Skewed. Therefore, all the following statements hold both for Linear and Skewed instances.

- Concerning the application of CPLEX to Ilp$_{\text{compl}}$, the alphabet size has a strong influence on the problem difficulty. A value of "n/a" denotes that CPLEX was not able to find a feasible solution within 3600 CPU seconds. For instances with $|\Sigma| = 4$, CPLEX is only able to provide feasible solutions for input strings of length up to 800, both in the context of instances Linear and Skewed. When $|\Sigma| = 12$, CPLEX can provide feasible solutions for input strings of length up to 1600 (Linear), respectively 1400 (Skewed). However, starting from $n = 1000$ CPLEX is not competitive with Cmsa anymore. Finally, even though CPLEX can provide feasible solutions for all instance sizes concerning the instances with $|\Sigma| = 20$, starting from $n = 1400$ the results are inferior to the ones of Cmsa.

Table 4: Results for the instances of set Skewed.

(a) Results for instances with Σ = 4.

n	Greedy	TreSea	Ilp$_{\text{compl}}$			HeurIlp		Cmsa	
	mean	**mean**	**mean**	**time**	**gap**	**mean**	**time**	**mean**	**time**
200	68.7	62.8	57.4	10/217	0.0%	64.6	< 1	57.5	903
400	120.3	115.0	105.3	168/1330	7.6%	116.5	3	105.1	1314
600	170.6	163.8	149.7	938/2193	10.1%	165.2	38	150.4	1500
800	219.8	213.3	224	2600/2600	22.9%	211.7	1334	196.5	2303
1000	268.6	261.7	n/a	n/a	n/a	260.1	1798	240.2	2692
1200	313.8	309.0	n/a	n/a	n/a	302.1	1807	285	2785
1400	358.7	352.2	n/a	n/a	n/a	346.0	1801	327.6	2888
1600	400.9	397.9	n/a	n/a	n/a	394.4	1807	376.0	2171
1800	440.6	442.1	n/a	n/a	n/a	431.7	1808	417.7	2162
2000	485.0	481.2	n/a	n/a	n/a	468.9	1814	470.2	1222
avg.	284.7	279.9	n/a	n/a	n/a	276.1	1221	262.6	1994

(b) Results for instances with Σ = 12.

n	Greedy	TreSea	Ilp$_{\text{compl}}$			HeurIlp		Cmsa	
	mean	**mean**	**mean**	**time**	**gap**	**mean**	**time**	**mean**	**time**
200	117.9	112.7	108.5	1/1	0.0%	112.7	< 1	108.6	12
400	216.1	208.5	193.4	10/136	0.0%	197.6	32	194.3	1002
600	304.8	301.7	274.5	71/1081	1.2%	277.9	650	277.2	1711
800	389.3	385.4	347.0	248/2725	2.3%	348.8	1533	351.0	2177
1000	471.6	468.9	429.4	650/2582	4.9%	428.7	1805	424.4	2648
1200	551.1	549.9	559.4	1351/1804	14.9%	535.0	1686	500.1	2597
1400	625.7	626.3	645.1	2693/2693	16.7%	638.4	1879	570.0	2962
1600	705.6	706.4	n/a	n/a	n/a	715.1	2981	643.8	2434
1800	788.4	788.9	n/a	n/a	n/a	810.1	4689	723.3	2329
2000	857.8	858.0	n/a	n/a	n/a	879.9	6072	797.3	2805
avg.	502.8	500.7	n/a	n/a	n/a	494.4	2133	459.0	2068

(c) Results for instances with Σ = 20.

n	Greedy	TreSea	Ilp$_{\text{compl}}$			HeurIlp		Cmsa	
	mean	**mean**	**mean**	**time**	**gap**	**mean**	**time**	**mean**	**time**
200	140.4	135.9	134.7	1/1	0.0%	136.5	< 1	134.7	8
400	255.5	251.3	240.3	3/4	0.0%	246.1	2	240.6	1080
600	366.8	361.2	336.1	19/101	0.0%	344.6	15	341.1	764
800	466.3	462.7	424.4	80/1119	0.2%	429.9	442	429.8	753
1000	567.6	566.6	514.7	202/2253	0.9%	525.0	1130	520.9	1121
1200	661.8	662.4	604.2	469/2064	2.1%	608.2	1633	605.7	1869
1400	762.3	760.7	694.4	719/2511	2.8%	696.1	1837	693.2	1743
1600	851.2	855.2	863.3	1378/1828	12.3%	838.9	1804	780.4	2681
1800	948.7	948.8	969.8	1774/1976	13.9%	964.7	1713	870.2	2815
2000	1034.3	1037.7	1061.6	2589/2589	14.4%	1066.6	2547	967.1	2978
avg.	605.5	604.3	584.4	723/1844	4.7%	585.7	1112	558.4	1581

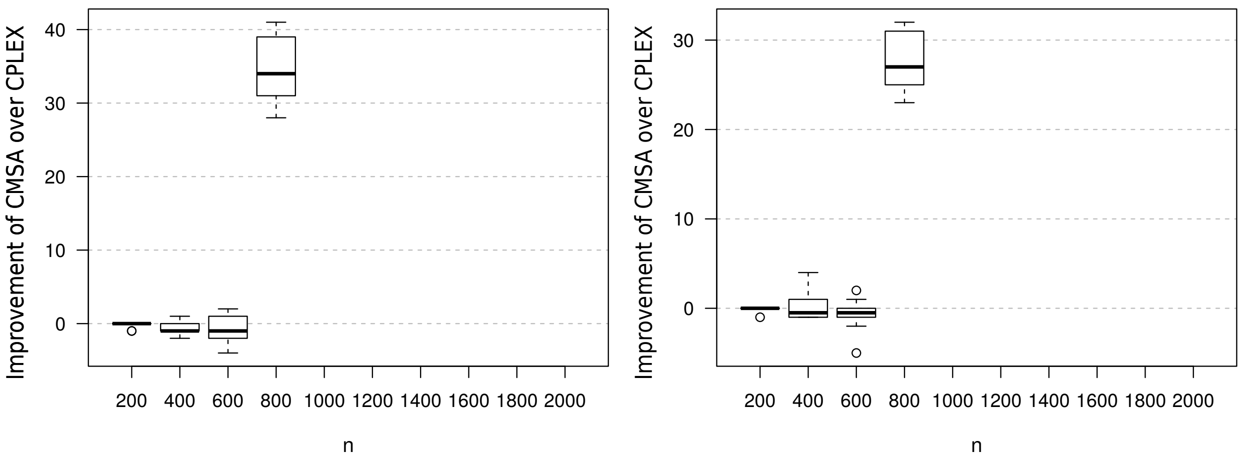
- For instances smaller than those for which Cmsa outperforms CPLEX, the differences between the results of Cmsa and the ones of applying CPLEX to Ilp$_{\text{compl}}$ are, again, very small.

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. Moreover, the larger the size of the input instances, and the smaller the alphabet size, the greater is the advantage of Cmsa over the other algorithms. The validity of these statements can be conveniently observed in the graphics of Figure 2.

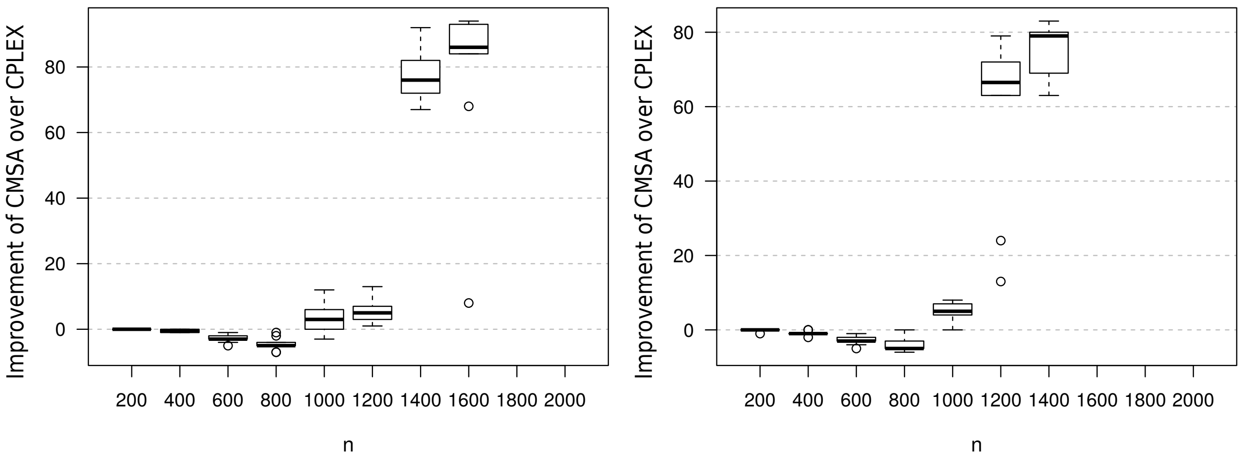
Finally, we also provide information about the average sizes of the sub-instances tackled within Cmsa, in comparison to the sizes of the original problem instances. In particular, the average sizes of the sub-instances are shown in Figure 3 in percent of the original problem instance sizes. For example, in the case $|\Sigma| = 4$, Linear, and input strings of length $n = 200$, the considered average size of the tackled sub-instances within Cmsa is approximately 58% of the size of the original instances. It can be observed that this percentage is getting smaller and smaller with growing size of the original instances. This is why CPLEX can either solve the sub-instances to optimality or provide nearly-optimal solutions in little computation time, even in the context of large original problem instances.

5.2. *Experiments Concerning the MCA Problem*

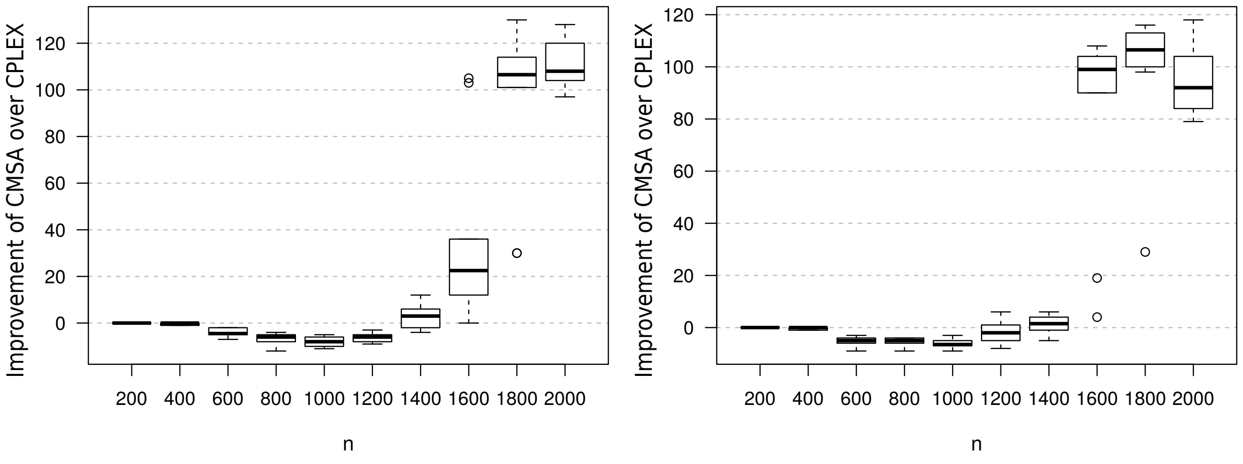
The following algorithms were considered for the comparison:



(a) Results for Σ = 4, Linear (left), Skewed (Right)



(b) Results for Σ = 12, Linear (left), Skewed (right)



(c) Results for Σ = 20, Linear (left), Skewed (right)

Figure 2: Differences between the results of Cmsa and the ones obtained by applying CPLEX to $\text{Ilp}_{\text{compl}}$ concerning the 600 instances of the second benchmark set. Each box shows these differences for the corresponding 10 instances. Note that negative values indicate that CPLEX obtained a better result than Cmsa.

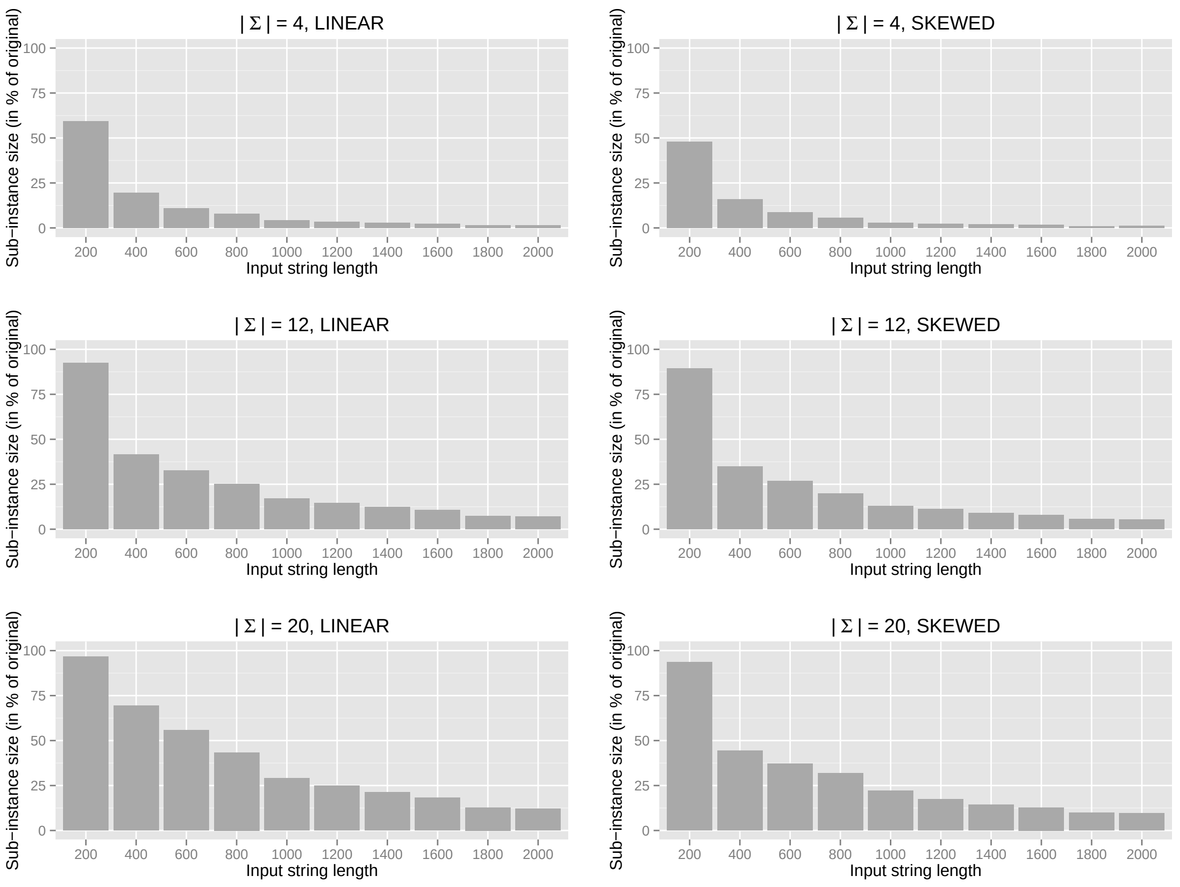


Figure 3: Average sizes of the sub-instances tackled within Cmsa concerning the 600 instances of the second benchmark set. In particular, sub-instance sizes are shown in percent of the original instances. For example, in the case $|\Sigma| = 4$, Linear, and $n = 200$, the considered average size of the tackled sub-instances within Cmsa is approx. 58% of the size of the original instances.

1. PGreedy: this is a cut-down version of Cmsa in which solutions are probabilistically generated, while other algorithmic components such as the ageing mechanism and the application of the ILP solver to reduced instances are not used. Remember that in the Cmsa implementation for the MCA problem, solutions are generated that are not extensible, that is, the generated solutions cover all reachable nodes. This implies that, during the construction process, once all nodes from X are included in the current arborescence, other complete (and valid) solutions are encountered. For each solution construction, PGreedy returns the best solution encountered in the process of generating a non-extensible solution.
2. Ilp$_{\text{compl}}$: the application of CPLEX to the complete ILP for each considered problem instance.
3. Cmsa: the proposed CMSA approach.

5.2.1. Benchmark Instances

A diverse set of benchmark instances was generated in the following way. Each benchmark instance is characterized by three different parameters. First, the size n (number of nodes) of each generated DAG is taken from $\{500, 1000, 5000\}$. In the process of randomly generating a DAG $G = (V, A)$ with n nodes, the arc probability $\mathbf{p}_{\text{arc}}$ is used to determine for each possible arc a pointing from a node $v_i \in V$ to another node $v_j \in V \setminus \{v_i\}$—where $i < j$—if a is added to A or not. Three different arc probabilities were considered: $\mathbf{p}_{\text{arc}} \in \{0.1, 0.3, 0.5\}$. Finally, the third parameter determines the size of set X. For this purpose we used a parameter *perc*, which refers to the percentage of the number of nodes of the respective DAG, that is, in case *perc* = 20%, for example, set X contains 20% of the nodes of the respective DAG. Note that the nodes in X are randomly selected from the set of reachable nodes. Values for *perc* were chosen from $\{1\%, 10\%, 20\%\}$.

Finally, note that the arc weights for all problem instances were chosen in the following way. In 99% of all arcs, a weight is chosen uniformly at random from $\{0, \ldots, 1000\}$. In the remaining cases, a negative arc weight is chosen from $\{-1000, \ldots, -1\}$. This was done in this way, because initial experiments indicated that a low percentage of arcs with negative weights leads to more difficult problem instances. For each possible combination of values for n, $\mathbf{p}_{\text{arc}}$, and *perc*, 10 problem instances were randomly generated. This makes a total of 270 problem instances.

5.2.2. *Tuning of* CMSA *and* PGREEDY

CMSA has several parameters 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, and $d_{\min}$—respectively $d_{\max}$—which determine the degree of greediness which is employed during the process of constructing a non-extensible arborescence. Note that, in the case of the MCA problem, parameter $t_{\max}$—the maximum time in seconds allowed for CPLEX per application to a sub-instance—was not subject to parameter tuning. This is because, in all cases, applications of CPLEX to sub-instances used very few CPU seconds. Therefore, we used a problem instance independent value of 50 for $t_{\max}$

As in the case of the MCSP problem, we make use of the automatic configuration tool irace [41] for the tuning of the three parameters. More specifically, irace was applied to tune CMSA separately for each combination of n and $\mathbf{p}_{\text{arc}}$. For each of these 9 combinations we randomly generated 12 *training instances*: four instances for each possible value of *perc*. The tuning process for each instance size was given a budget of 1000 runs of CMSA, where each run was given a computation time limit of $n/2$ CPU seconds. Finally, the following parameter value ranges were chosen concerning the three 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 the sub-instance.
- $(d_{\min}, d_{\max}) \in \{(0.0, 0.0), (0.5, 0.5), (0.9, 0.9), (0.0, 0.5), (0.5, 0.9), (0.0, 0.9)\}$

The 9 applications of irace produced the configurations of CMSA as shown in Table 5a. The following trends can be observed. First of all, the desired number of solution constructions per iteration seems to decrease with increasing instance size (in terms of the number of nodes). The same trend can be observed for the values of parameter $\text{age}_{\max}$, whose desired value tends to decrease with increasing instance size. Concerning the greedyness of the solution constructions process, rather low greedyness seems to be indicated. This is with the exception of the instances with $n = 500$ and $\mathbf{p}_{\text{arc}} = 0.5$ for which the obtained values for $d_{\min}$ and $d_{\max}$ are 0.5, respectively 0.9.

In addition to tuning experiments for CMSA, we also performed tuning experiments for PGREEDY. As PGREEDY constructs solutions in the same way in which they are constructed in CMSA, the parameters involved in PGREEDY are $d_{\min}$ and $d_{\max}$. For the tuning of PGREEDY we used the same training instances and the same parameter value combinations as for the tuning of CMSA. The obtained parameter values per combination of n and $\mathbf{p}_{\text{arc}}$ are displayed in Table 5b.

5.2.3. *Results*

In the following we present the experimental results for the benchmark set described in Section 5.2.1. The results are shown in the three Tables 6a–6c. Note that each table row provides average results over 10 problem instances, and that each considered algorithm was applied exactly once to each problem instance. The layout of the three tables is as follows. The first column provides the size of the input graphs in terms of the number of nodes, whereas the second column indicates the graph density in terms of the edge probability used to generate the corresponding graphs. The third and fourth column provide the results and computation times of PGREEDY. The next three table columns are dedicated to the presentation of the results provided by solving the complete ILP model $\text{ILP}_{\text{compl}}$ described in Section 4.1.1. The first one of these columns provides the value of the best solution found within $n/2$ CPU seconds, where n is the number of nodes of the respective graphs. The second column provides the computation time (in seconds) needed to solve the problems to optimality (if possible). Finally, the third one of the columns dedicated to $\text{ILP}_{\text{compl}}$ shows the

Table 5: Parameter settings produced by irace for the 9 different combinations of n and $\mathbf{p}_{\text{arc}}$.

(a) Tuning results for Cmsa

$(n, \mathbf{p}_{\text{arc}})$	n_a	$\text{age}_{\max}$	$(d_{\min}, d_{\max})$
(500, 0.1)	50	10	(0.0, 0.0)
(500, 0.3)	30	10	(0.5, 0.5)
(500, 0.5)	30	5	(0.5, 0.9)
(1000, 0.1)	10	5	(0.5, 0.5)
(1000, 0.3)	30	1	(0.5, 0.5)
(1000, 0.5)	10	1	(0.5, 0.5)
(5000, 0.1)	10	5	(0.0, 0.5)
(5000, 0.3)	10	1	(0.0, 0.5)
(5000, 0.5)	10	5	(0.0, 0.5)

(b) Tuning results for PGreedy

$(n, \mathbf{p}_{\text{arc}})$	$(d_{\min}, d_{\max})$
(500, 0.1)	(0.9, 0.9)
(500, 0.3)	(0.5, 0.9)
(500, 0.5)	(0.9, 0.9)
(1000, 0.1)	(0.5, 0.9)
(1000, 0.3)	(0.5, 0.9)
(1000, 0.5)	(0.0, 0.5)
(5000, 0.1)	(0.0, 0.5)
(5000, 0.3)	(0.0, 0.5)
(5000, 0.5)	(0.0, 0.5)

optimality gap, which refers to the gap between the value of the best valid solution and the current lower bound at the time of stopping a run. The last two columns display the results obtained by Cmsa. The first one of these columns provides the results and the second one the average time needed by Cmsa to obtain these results. The best-performing algorithm for each table row is marked by a grey background.

The analysis of the results permits to draw the following conclusions:

- Concerning the application of CPLEX to $\text{Ilp}_{\text{compl}}$, the size of the input graphs has—as expected—a strong influence on the problem difficulty. In fact, CPLEX was able to solve all problem instances with $n \in \{500, 1000\}$ to optimality. In contrast, CPLEX was not even able to come up with a feasible solution within the allowed computation time in the case of input graphs with $n = 5000$. The percentage of nodes that must be included in a solution (*perc*) apparently has no influence on CPLEX. With growing value of *perc*, CPLEX seems even faster in solving the corresponding problem instances.
- For input graphs with $n \in \{500, 1000\}$ Cmsa is nearly always able to provide optimal solutions, and is, therefore, competitive with Cplex. With growing graph density, Cmsa is considerably faster than Cplex. For instances with $n = 5000$, Cmsa outperforms both Cplex, which is not able to provide feasible solutions, and the probabilistic greedy algorithm PGreedy.

Again, as in the case of the MCSP problem, we also provide in the case of the MCA problem information about the average sizes of the sub-instances tackled within Cmsa, in comparison to the sizes of the original problem instances. These average sub-instance sizes are shown in Figure 4 in percent of the original problem instance sizes. More in detail, the 270 benchmark instances are categorized into nine different subsets concerning the number of nodes and the density of the graph. A notation X-Y is used, where X refers to the number of nodes of the graphs, that is, $X \in \{500, 1000, 5000\}$, and Y refers to low, medium and high density, that is, $Y \in \{L, M, H\}$. For example, in the case 500-H, that is, graphs with 500 nodes of high density, the considered average size of the tackled sub-instances within Cmsa is approximately 22% of the size of the original instances. It can be observed that this percentage is getting smaller and smaller with growing size of the input graphs and growing density. This is why CPLEX can either solve the sub-instances to optimality or provide nearly-optimal solutions in little computation time, even in the context of large original problem instances.

6. Conclusions and Future Work

In this paper we introduced a new, generally applicable, algorithm for solving combinatorial optimization problems. The algorithm is an instantiation of the Generate-and-Solve framework from the literature. It is based on the general idea of generating solutions in a probabilistic way, solving the sub-instances of the original problem instance that result from merging the solution components contained in the generated solutions to optimality, and adapting these

Table 6: Results for the MCA problem instances.

(a) Results for instances with *perc* = 1%.

n	$\mathbf{p}_{\mathrm{arc}}$	PGreedy		Ilp$_{\mathrm{compl}}$			Cmsa	
		mean	**time**	**mean**	**time**	**gap**	**mean**	**time**
	0.1	10011.7	95.4	-940.1	0.4	0.0	-940.1	7.6
500	0.3	2429.6	123.9	-13450.8	3.5	0.0	-13450.8	2.6
	0.5	-6244.6	151.8	-28030.8	12.8	0.0	-28030.8	4.7
	0.1	33128.9	272.4	-15263.9	4.5	0.0	-15251.2	63.0
1000	0.3	-7501.4	235.4	-62414.8	42.3	0.0	-62414.5	86.4
	0.5	-42531.7	292.9	-106522.3	152.9	0.0	-106522.3	58.4
	0.1	-114469.8	1073.0	n/a	n/a	n/a	-530515.0	1572.8
5000	0.3	-757318.8	1167.1	n/a	n/a	n/a	-1380184.7	938.1
	0.5	-1203516.6	1716.8	n/a	n/a	n/a	-1959379.6	365.6

(b) Results for instances with *perc* = 10%.

n	$\mathbf{p}_{\mathrm{arc}}$	PGreedy		Ilp$_{\mathrm{compl}}$			Cmsa	
		mean	**time**	**mean**	**time**	**gap**	**mean**	**time**
	0.1	47753.9	100.1	5648.3	0.3	0.0	5653.6	29.8
500	0.3	12247.8	153.9	-11338.3	3.6	0.0	-11338.3	3.3
	0.5	6.8	142.9	-26982.4	11.4	0.0	-26982.4	11.2
	0.1	62650.5	130.1	-9115.1	3.5	0.0	-9025.9	119.3
1000	0.3	1152.3	248.4	-61065.8	38.5	0.0	-61065.8	51.2
	0.5	-39504.9	205.3	-105633.0	145.6	0.0	-105633.0	51.5
	0.1	-104899.7	1760.7	n/a	n/a	n/a	-526539.8	1922.3
5000	0.3	-758425.1	853.0	n/a	n/a	n/a	-1379684.4	861.9
	0.5	-1203092.2	1239.2	n/a	n/a	n/a	-1959193.4	247.2

(c) Results for instances with *perc* = 20%.

n	$\mathbf{p}_{\mathrm{arc}}$	PGreedy		Ilp$_{\mathrm{compl}}$			Cmsa	
		mean	**time**	**mean**	**time**	**gap**	**mean**	**time**
	0.1	51730.7	139.2	10887.8	0.3	0.0	10899.6	41.9
500	0.3	14726.6	126.4	-9801.0	3.2	0.0	-9801.0	3.2
	0.5	2258.7	104.4	-25827.6	11.1	0.0	-25827.6	32.0
	0.1	66223.5	237.3	-4228.4	2.9	0.0	-4190.0	168.7
1000	0.3	4079.5	258.1	-59319.6	36.4	0.0	-59319.6	73.2
	0.5	-39905.1	218.8	-104945.8	136.7	0.0	-104941.1	57.1
	0.1	-105791.8	1304.2	n/a	n/a	n/a	-522990.6	1831.9
5000	0.3	-751317.6	1768.9	n/a	n/a	n/a	-1379042.0	754.9
	0.5	-1204557.1	967.2	n/a	n/a	n/a	-1959042.5	399.5

sub-instances based on an aging mechanism. The proposed algorithm has been applied to two NP-hard combinatorial optimization problems—the minimum common string partition problem and the minimum covering arborescence problem—as test cases. The results have shown that the proposed algorithm is a state-of-the-art method for these problems, especially, for what concerns rather large problem instances.

In future work we will consider the following two lines of research. First, we would like to apply the algorithm to other types of combinatorial optimization problems such as, for example, permutation problems or scheduling problems. Second, we plan to study alternatives for the aging mechanism applied in this work. This is because the aging mechanism results in a binary decision whether a solution component is considered or not. It would be interesting to investigate more fine-grained mechanisms that take into account the quality of the solutions or interactions between solution components.

Acknowledgments. C. Blum was supported by project TIN2012-37930-02 of the Spanish Government. In addition, support is acknowledged from IKERBASQUE (Basque Foundation for Science). J. A. Lozano was partially supported by the IT609-13 program (Basque Government) and project TIN2013-41272P (Spanish Ministry of Science and Innovation).

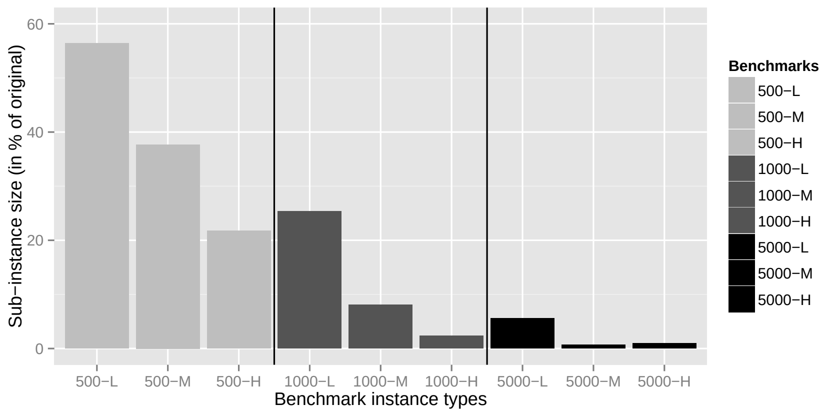


Figure 4: Average sizes of the sub-instances tackled within Cmsa concerning the 270 instances of the benchmark set, categorized into nine different subsets (see the text for a more detailed description). Sub-instance sizes are shown in percent of the original instances. For example, in the case 500-H, that is, graphs with 500 nodes of high density, the considered average size of the tackled sub-instances within Cmsa is approx. 22% of the size of the original instances.

References

[1] C. Blum, J. Puchinger, G. Raidl, A. Roli, Hybrid metaheuristics in combinatorial optimization: A survey, Applied Soft Computing 11 (6) (2011) 4135–4151.

[2] E.-G. Talbi (Ed.), Hybrid Metaheuristics, No. 434 in Studies in Computational Intelligence, Springer Verlag, Berlin, Germany, 2013.

[3] G. R. Raidl, Decomposition based hybrid metaheuristics, European Journal of Operational Research 244 (1) (2015) 66–76.

[4] M. A. Boschetti, V. Maniezzo, M. Roffilli, A. Bolufé Röhler, Matheuristics: Optimization, simulation and control, in: M. J. Blesa, C. Blum, L. Di Gaspero, A. Roli, M. Sampels, A. Schaerf (Eds.), Proceedings of HM 2009 – Sixth International Workshop on Hybrid Metaheuristics, Vol. 5818 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, pp. 171–177.

[5] L. A. Wolsey, Integer Programming, Wiley-Interscience, Hoboken, NJ, 1998.

[6] N. V. Nepomuceno, Combinação de metaheurísticas e programação linear inteira: uma metodologia híbrida aplicada ao problema de carregamento de contêineres, Ph.D. thesis, MSc Thesis, University of Fortaleza (2006).

[7] N. V. Nepomuceno, P. R. Pinheiro, A. L. V. Coelho, Combining metaheuristics and integer linear programming: A hybrid methodology applied to the container loading problem, in: Proceedings of the XX Congreso da Sociedade Brasileira de Computação, Concurso de Teses e Dissertações, 2007, pp. 2028–2032.

[8] N. V. Nepomuceno, P. R. Pinheiro, A. L. V. Coelho, Tackling the container loading problem: A hybrid approach based on integer linear programming and genetic algorithms, in: C. Cotta, J. van Hemert (Eds.), Proceedings of EvoCOP 2007 – 7th European Conference on Evolutionary Computation in Combinatorial Optimization, Vol. 4446 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 154–165.

[9] N. Nepomuceno, P. Pinheiro, A. L. V. Coelho, Recent Advances in Evolutionary Computation for Combinatorial Optimization, Vol. 153 of Studies in Computational Intelligence, Springer Verlag, Berlin, Germany, 2008, Ch. A Hybrid Optimization Framework for Cutting and Packing Problems, pp. 87–99.

[10] P. R. Pinheiro, A. L. V. Coelho, A. B. de Aguiar, T. O. Bonates, On the concept of density control and its application to a hybrid optimization framework: Investigation into cutting problems, Computers & Industrial Engineering 61 (3) (2011) 463–472.

[11] R. D. Saraiva, N. V. Nepomuceno, P. R. Pinheiro, The generate-and-solve framework revisited: Generating by simulated annealing, in: M. Middendorf, C. Blum (Eds.), Evolutionary Computation in Combinatorial Optimization, Vol. 7832 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 262–273.

[12] D. Coudert, N. V. Nepomuceno, I. Tahiri, Energy saving in fixed wireless broadband networks, in: J. Pahl, T. Reiners, S. Voß (Eds.), Proceedings of INOC 2011 – 5th International Conference on Network Optimization, Vol. 6701 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 484–489.

[13] D. Coudert, N. Nepomuceno, H. Rivano, Power-efficient radio configuration in fixed broadband wireless networks, Computer Communications 33 (8) (2010) 898–906.

[14] P. R. Pinheiro, A. L. V. Coelho, A. B. de Aguiar, A. de Menezes Sobreira Neto, Towards aid by generate and solve methodology: application in the problem of coverage and connectivity in wireless sensor networks, International Journal of Distributed Sensor Networks 2012, article ID 790459.

[15] D. Applegate, R. Bixby, V. Chvátal, W. Cook, Finding tours in the TSP, Tech. rep., Forschungsinstitut für Diskrete Mathematik, University of Bonn, Germany (1999).

[16] W. Cook, P. Seymour, Tour merging via branch-decomposition, INFORMS Journal on Computing 15 (3) (2003) 233–248.

[17] G. W. Klau, I. Ljubić, A. Moser, P. Mutzel, P. Neuner, U. Pferschy, G. Raidl, R. Weiskircher, Combining a memetic algorithm with integer programming to solve the prize-collecting Steiner tree problem, in: Proceedings of GECCO 2004 – Genetic and Evolutionary Computation Conference, Vol. 3102 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2004, pp. 1304–1315.

[18] F. Massen, Y. Deville, P. Hentenryck, Pheromone-based heuristic column generation for vehicle routing problems with black box feasibility, in: N. Beldiceanu, N. Jussien, É. Pinson (Eds.), Proceedings of CP-AI-OR 2012 — Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems, Vol. 7298 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 260–274.

[19] F. Massen, M. López-Ibáñez, T. Stützle, Y. Deville, Experimental analysis of pheromone-based heuristic column generation using irace, in: M. J. Blesa, C. Blum, P. Festa, A. Roli, M. Sampels (Eds.), Proceedings of HM 2013 – International Workshop on Hybrid Metaheuristics, Vol. 7919 of Lecture Notes in Computer Science, Springer, 2013, pp. 92–106.

[20] C. Blum, B. Calvo, A matheuristic for the minimum weight rooted arborescence problem, Journal of Heuristics 21 (4) (2015) 479–499.

[21] X. Chen, J. Zheng, Z. Fu, P. Nan, Y. Zhong, S. Lonardi, T. Jiang, Computing the assignment of orthologous genes via genome rearrangement, in: Proceedings of the Asia Pacific Bioinformatics Conference 2005, 2005, pp. 363–378.

[22] S. Mousavi, M. Babaie, M. Montazerian, An improved heuristic for the far from most strings problem, Journal of Heuristics 18 (2012) 239–262.

[23] C. Meneses, C. Oliveira, P. Pardalos, Optimization techniques for string selection and comparison problems in genomics, IEEE Engineering in Medicine and Biology Magazine 24 (3) (2005) 81–87.

[24] W. J. Hsu, M. W. Du, Computing a longest common subsequence for a set of strings, BIT Numerical Mathematics 24 (1) (1984) 45–59.

[25] T. Smith, M. Waterman, Identification of common molecular subsequences, Journal of Molecular Biology 147 (1) (1981) 195–197.

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

[27] M. R. Garey, D. S. Johnson, Computers and intractability; a guide to the theory of *NP*-completeness, W. H. Freeman, 1979.

[28] A. Goldstein, P. Kolman, J. Zheng, Minimum common string partition problem: Hardness and approximations, in: R. Fleischer, G. Trippen (Eds.), Proceedings of ISAAC 2004 – 15th International Symposium on Algorithms and Computation, Vol. 3341 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2005, pp. 484–495.

[29] P. Kolman, T. Waleń, Reversal distance for strings with duplicates: Linear time approximation using hitting set, in: T. Erlebach, C. Kaklamanis (Eds.), Proceedings of WAOA 2007 – 4th International Workshop on Approximation and Online Algorithms, Vol. 4368 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 279–289.

[30] I. Goldstein, M. Lewenstein, Quick greedy computation for minimum common string partitions, in: R. Giancarlo, G. Manzini (Eds.), Proceedings of CPM 2011 – 22nd Annual Symposium on Combinatorial Pattern Matching, Vol. 6661 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2011, pp. 273–284.

[31] D. He, A novel greedy algorithm for the minimum common string partition problem, in: I. Mandoiu, A. Zelikovsky (Eds.), Proceedings of ISBRA 2007 – Third International Symposium on Bioinformatics Research and Applications, Vol. 4463 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 441–452.

[32] S. M. Ferdous, M. Sohel Rahman, Solving the minimum common string partition problem with the help of ants, in: Y. Tan, Y. Shi, H. Mo (Eds.), Proceedings of ICSI 2013 – 4th International Conference on Advances in Swarm Intelligence, Vol. 7928 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, pp. 306–313.

[33] S. M. Ferdous, M. Sohel Rahman, A MAX-MIN ant colony system for minimum common string partition problem, CoRR abs/1401.4539, `http://arxiv.org/abs/1401.4539`.

[34] C. Blum, J. A. Lozano, P. Pinacho Davidson, Iterative probabilistic tree search for the minimum common string partition problem, in: M. J. Blesa, C. Blum, S. Voss (Eds.), Proceedings of HM 20104– 9th International Workshop on Hybrid Metaheuristics, Vol. 8457 of Lecture Notes in Computer Science, Springer Verlag, Berlin, Germany, 2014, pp. 154–154.

[35] C. Blum, J. A. Lozano, P. Pinacho Davidson, Mathematical programming strategies for solving the minimum common string partition problem, European Journal of Operational Research 242 (3) (2015) 769–777.

[36] V. Venkata Rao, R. Sridharan, The minimum weight rooted arborescence problem: Weights on arcs case, Tech. rep., Indian Institute of Management Ahmedabad, Research and Publication Department (1992).

[37] W. T. Tutte, Graph Theory, Cambridge University Press, Cambridge, UK, 2001.

[38] J. Bang-Jensen, G. Z. Gutin, Digraphs: theory, algorithms and applications, Springer Science & Business Media, 2008.

[39] V. Venkata Rao, R. Sridharan, Minimum-weight rooted not-necessarily-spanning arborescence problem, Networks 39 (2) (2002) 77–87.

[40] C. Duhamel, L. Gouveia, P. Moura, M. Souza, Models and heuristics for a minimum arborescence problem, Networks 51 (1) (2008) 34–47.

[41] M. López-Ibáñez, J. Dubois-Lacoste, T. Stützle, M. Birattari, The irace package, iterated race for automatic algorithm configuration, Tech. Rep. TR/IRIDIA/2011-004, IRIDIA, Université Libre de Bruxelles, Belgium (2011).