International Journal of Computational Intelligence Systems (2024) 17:109
https://doi.org/10.1007/s44196-024-00488-7

RESEARCH ARTICLE

Extension of CMSA with a Learning Mechanism: Application to the Far from Most String Problem

Pedro Pinacho-Davidson[1] · Christian Blum[2] · M. Angélica Pinninghoff[1] · Ricardo Contreras[3]

Received: 5 December 2023 / Accepted: 26 March 2024

Abstract
One of the problems with exact techniques for solving combinatorial optimization problems is that they tend to run into problems with growing problem instance size. Nevertheless, they might still be very usefully employed, even in the context of large problem instances, as a sub-ordinate method within so-called hybrid metaheuristics. "Construct, Merge, Solve and Adapt" (CMSA) is a hybrid metaheuristic technique that allows the application of exact methods to large-scale problem instances through intelligent instance reduction. However, CMSA does not make use of an explicit learning mechanism. In this work, an algorithm called LEARN_CMSA is presented for the application to the far from most string problem (FFMSP), which is an NP-hard combinatorial optimization problem from the field of string consensus problems. LEARN_CMSA results from hybridization between CMSA and a population-based algorithm. By means of this hybridization, explicit learning is introduced to CMSA. Even though the FFMSP is a well-studied problem, LEARN_CMSA achieves superior performance when compared to current state-of-the-art solvers.

Keywords Hybrid metaheuristics · Population-based algorithm · Optimization · String consensus problems

Abbreviations

ILP	– Integer linear programming;
CMSA	– Construct, Merge, Solve & Adapt;
FFMSP	– Far from most string problem;
ACO	– Ant colony optimization;
BA	– Bacteria algorithm;

✉ Christian Blum
christian.blum@iiia.csic.es

Pedro Pinacho-Davidson
ppinacho@udec.cl

M. Angélica Pinninghoff
mpinning@udec.cl

Ricardo Contreras
ricardo.contreras@edu.uai.cl

[1] Department of Computer Science, Faculty of Engineering, Universidad de Concepción, Edmundo Larenas 219, Concepción 4070409, Chile

[2] Artificial Intelligence Research Institute (IIIA-CSIC), Campus of the UAB, Bellaterra 08193, Spain

[3] Faculty of Sciences and Engineering, Universidad Adolfo Ibáñez, Diagonal Las Torres 2640, Peñalolén, Santiago de Chile 7910000, Chile

1 Introduction

In recent decades, different algorithmic approaches have been introduced for solving combinatorial optimization problems to find the best or satisfactory solutions. Generally, we distinguish between exact approaches that derive an optimal solution in bounded computation time, and heuristic approaches that provide sub-optimal solutions within more practical computation time limits. Especially in those cases in which large-scale problem instances must be solved, heuristic or metaheuristic search strategies are often used instead of exact techniques. Metaheuristic algorithms include methods such as evolutionary algorithms, ant colony optimization and tabu search, just to name a few. Depending on the type of optimization problem, a whole range of different techniques can be used in the process of finding a good enough solution. Nowadays, many of these approaches take advantage of the joint use of exact and heuristic techniques. Such approaches are often called *hybrid metaheuristics* [1, 2] or *matheuristics* [3].

The hybrid metaheuristic framework CMSA (Construct, Merge, Solve & Adapt) was introduced by [4] for solving combinatorial optimization problems. The concept behind CMSA is to still profit from the power of exact solvers, even

Published online: 29 April 2024

for the application to large problem instances. In particular, CMSA solves—at each iteration—a sub-instance of the original problem instance. All these sub-instances are generated by merging different solutions constructed during the algorithm run. Moreover, sub-instances are usually expressed in terms of integer linear programming (ILP) models and solved by black-box ILP solvers such as CPLEX or Gurobi. Applications of CMSA have shown remarkably good results for some interesting problems. Among these problems, we find, for example, *minimum common string partition* [5], which is a problem in the field of bioinformatics. Other examples include *project scheduling* [6], the *maximum happy vertices problem* [7], and the problem of *test case generation for software checking* [8].

The present proposal focuses on the development of a new variant of CMSA based on the hybridization with a population-based metaheuristic, with the goal of adding a learning component to standard CMSA. The usefulness of this proposal will be shown by the application to the so-called *far from most string problem* (FFMSP) [9], which is an NP-hard combinatorial optimization problem. This optimization problem forms part of a category of string-related problems known as sequence consensus problems. In these problems, a finite set of sequences is provided, and the goal is to identify their consensus—namely, a novel sequence that effectively encapsulates the essence of all the provided sequences. Various objectives, some potentially conflicting, can be contemplated within the realm of sequence consensus problems. Examples of such objectives are outlined in the following [10].

- **Closest string problem (CSP):** Find a sequence whose total distance from all input sequences is minimal
- **Farthest string problem (FSP):** Find a sequence that maximizes its overall distance from all input sequences.
- **Far from most string problem (FFMSP):** Find a sequence far from most of the input sequences, where "most" is defined by a fixed threshold.

As mentioned above, to solve the FFMSP problem, we develop a new CMSA variant that incorporates learning into the solution construction process. In the case of this hybrid approach, the learning component will result from the hybridization with a bacterial algorithm (BA), which is an evolutionary algorithm in which the crossover operator is inspired by processes observed in bacteria. Note that any population-based approach could have been used to add learning to CMSA. Moreover, note that the BA algorithm will be described in metaphor-free language in this work.

In general, BAs are inspired by the survival of a bacterial population, and how a population of bacteria evolves and develops a resistance to antibiotics. Note that when bacteria develop such resistance, they are no longer affected by exposure to antibiotics. From the viewpoint of bacteria, developing such a resistance is, of course, desirable. From a medical point of view, however, this may result in severe problems for humans. This is due to the fact that infections triggered by resistant microorganisms frequently do not respond to standard treatments, leading to prolonged illness and an increased likelihood of mortality. In simpler terms, antibiotic resistance refers to a form of drug resistance wherein a microorganism can withstand exposure to an antibiotic [11]. Recent research concerning the exploitation of bacteria behavior has already resulted in the application to a group formation problem in the context of designing students' activities. The core idea is that students collaborate in a group to improve their academic performance [12].

1.1 Contribution and Paper Outline

In this work, we present an algorithmic proposal that belongs to the category of hybrid algorithms based on problem instance reduction [1]. In particular, we augment the CMSA algorithm with a learning mechanism by combining it with a population-based metaheuristic. The proposed hybrid algorithm, called LEARN_CMSA, is applied to the notoriously difficult far from most string problem. In the section on the experimental evaluation, we not only compare LEARN_CMSA to its components (CMSA and the bacterial algorithm), but we also compare to the current state of the art for the far from most string problem. Our results show that LEARN_CMSA outperforms its algorithmic components and that it compares very favorably with the current state of the art.

The remainder of this paper is organized as follows. Section 2 introduces the far from most string problem. Section 3 first introduces the pure variants of both CMSA and the bacterial algorithm, before discussing the hybridization of CMSA with the bacterial algorithm. Section 4 outlines the experimental design and the obtained results, and finally, Sect. 5 offers conclusions derived from this research and an outlook to future work.

2 The Far from Most String Problem

A problem instance of the far from most string problem (FFMSP) is represented as (Ω, t), where set $\Omega = s_1, \ldots, s_n$ contains n input strings over a finite alphabet Σ. Each input string s_i in Ω has a length of m, i.e., $|s_i| = m$ for all $s_i \in \Omega$. Additionally, a fixed threshold value $0 < t < m$ is provided. In subsequent discussions, $s_i[j]$ denotes the jth character of a string s_i. The *Hamming distance* between two strings $s_i \neq s_j \in \Omega$ of equal length, expressed as $d_H(s_i, s_j)$, is defined as the count of positions where corresponding char-

acters in the two strings differ. In other words:

$$d_H(s_i, s_j) = \left|\left\{k \in \{1, \ldots, m\} \mid s_i[k] \neq s_j[k]\right\}\right|. \quad (1)$$

Note that all possible strings of length m over alphabet Σ are valid FFMSP solutions. Given such a string s, its objective function value $f_{\text{orig}}(s)$ is computed as follows:

$$f_{\text{orig}}(s) := |\{s_i \in \Omega \mid d_H(s, s_i) \geq t\}|. \quad (2)$$

This implies that the objective function value of a solution or string s is determined by the count of input strings for which the Hamming distance with s is greater than or equal to the threshold value t.

In addition to the technical problem description, we provide an integer linear programming (ILP) model of the FFMSP. This is because CMSA and LEARN_CMSA internally use an ILP solver for the purpose of solving sub-instances of the considered problem instance. In particular, we provide below the description of the ILP model from [10]. This model makes use of two sets of binary variables. The first one of these sets comprises a variable $x_{j,a}$ for every combination of a position $j = 1, \ldots, m$ in a potential solution and a character $a \in \Sigma$. The second set encompasses binary variables y_i corresponding to each input string $s_i \in \Omega$ $(i = 1, \ldots, n)$. The ILP model can be formulated as follows.

$$\textbf{max} \sum_{i=1}^{n} y_i \quad (3)$$

subject to:

$$\sum_{a \in \Sigma} x_{j,a} = 1 \quad \forall\, j \in \{1, \ldots, m\} \quad (4)$$

$$\sum_{j=1}^{m} x_{j,s_i[j]} \leq m - t \cdot y_i \quad \forall\, i \in \{1, \ldots, n\}$$

$$x_{j,a},\, y_i \in \{0, 1\} \quad (5)$$

Observe that constraints (4) guarantee each position j in a solution string is occupied by exactly one letter from Σ. Furthermore, constraints (5) force a variable y_i to take the value zero in case the Hamming distance of the solution string (defined by the x-variables) to input string $s_i \in \Omega$ is below threshold t.

2.1 Existing Work on the FFMSP

The FFMSP can be regarded a well-studied problem. In 2003, for example, [13] proved that approximating the FFMSP within a polynomial factor is NP-hard for sequences over an alphabet Σ with $|\Sigma| \geq 3$. Given the computational complexity of the problem, the research community has primarily concentrated on heuristic and metaheuristic approaches. The initial proposal consisted of a greedy heuristic with the subsequent application of local search [14]. In fact, approaches based on randomized solution construction—in particular, metaheuristics known as greedy randomized adaptive search procedures (GRASP)—have enjoyed popularity for the application to the FFMSP problem; see [15–19]. The newest one of these GRASP approaches [19] is, in fact, a hybrid technique that combines GRASP both with variable neighborhood search (VNS) and path relinking.

Apart from the GRASP proposals, the literature on the FFMSP also offers evolutionary algorithms (EAs) such as the one from [16]. This algorithm features a diversity preservation mechanism to augment its exploration ability. The second EA proposal from [20] is a memetic algorithm that makes use of local search to improve the generated solutions. This algorithm proposal takes profit from earlier work using GRASP mechanisms for the construction of solutions. Ant colony optimization (ACO) is another bioinspired metaheuristic that has been applied already twice to the FFMSP. The first application from [10] hybridizes an ACO algorithm with the application of the ILP solver CPLEX for a possible improvement of the output of ACO. In contrast, in [21] the authors propose the application of a very recent ACO variant, known as *negative learning ACO*, to the FFMSP. Finally, for the sake of completeness, it is worth mentioning the beam search approach from [9]. Currently, the negative learning ACO approach from [21] and the memetic algorithm from [20] can be regarded as the state-of-the-art approaches for solving the FFMSP.

2.2 Augmented Objective Functions

The FFMSP poses a significant challenge not only for exact techniques but also for metaheuristics, primarily due to the limited range of possible objective function values. In fact, for an instance with n input strings the set of possible objective function values is $\{0, \ldots, n\}$. Because of this, the search space of an FFMSP problem instance encompasses broad plateaus, leading to situations where similar solutions frequently share identical objective function values. For a metaheuristic, this means that the search space often offers little (or no) guidance regarding the question of how to keep moving and exploring during the search process. This results in the fact that metaheuristics often get trapped on such plateaus. In light of the previously discussed reasons, [21] tested four different augmented objective functions in addition to the original objective function. In this work, we will consider the most successful options. It is crucial to note, though, that these alternative functions can solely be employed for all solution evaluations/comparisons occurring within the CMSA variants and in the bacterial algorithm. CPLEX, which is utilized in CMSA and LEARN_CMSA for

solving sub-instances at every iteration, continues to rely on the original objective function.

[17] proposed the first alternative objective function, which is denoted as $f_{\text{mou}}()$. The purpose of this function is to generate a search landscape with fewer plateaus and a decreased number of local optima. It evaluates a solution by considering the *likelihood* of it leading to improved solutions through a relatively small number of local search moves. It is important to note that if $f_{\text{orig}}(s) > f_{\text{orig}}(s')$ for two valid solutions s and s', then $f_{\text{mou}}(s) > f_{\text{mou}}(s')$ also holds. Consequently, $f_{\text{mou}}()$ can function independently. Due to the complexity of $f_{\text{mou}}()$ and the limited space available for its description, we direct interested readers to the original publication [17] or to [20], where the authors implemented $f_{\text{mou}}()$ in the context of their memetic algorithm.

[10] introduced a second alternative objective function, which we will refer to as $f_{\text{blu}}()$. This function is a lexicographic objective function that primarily employs the original objective function as its first criterion. The second criterion utilizes the following function:

$$h(s) := \sum_{\{s_i \in \Omega | d_H(s,s_i) \geq t\}} d_H(s,_s i) + \max_{\{s_i \in \Omega | d_H(s,s_i) < t\}} \{d_H(s, s_i)\}. \quad (6)$$

In more straightforward language, $h(s)$ adds up the Hamming distances between s and the input strings $s_i \in \Omega$ in those cases in which the Hamming distance is at least t. It also takes into account the maximum Hamming distance between s and the input strings $s_i \in \Omega$ where the Hamming distance is less than t. The original objective function and $h()$ are then integrated using a lexicographic approach:

$$f_{\text{blu}}(s) > f_{\text{blu}}(s') \textbf{iff} f_{\text{orig}}(s) > f_{\text{orig}}(s') \textbf{or} (f_{\text{orig}}(s) = f_{\text{orig}}(s') \textbf{and} h(s) > h(s')). \quad (7)$$

For valid solutions s and s' to the problem, the rationale behind $h()$ can be described as follows: a higher value of $h(s)$ corresponds to a lower likelihood that minor modifications in s will result in a reduction of the original objective function. In addition to the two functions mentioned earlier, we also examine a simplified variant of $f_{\text{blu}}()$ that employs function $h'()$ rather than $h()$:

$$h'(s) := \max_{\{s_i \in \Omega | d_H(s,s_i) < t\}} \{d_H(s, s_i)\}. \quad (8)$$

It is important to note that $h'(s)$, unlike $h(s)$, focuses solely on the maximum Hamming distance between s and the input strings $s_i \in \Omega$ where the Hamming distance is less than t.

Algorithm 1 CMSA for the FFMSP

1: **input:** problem instance (Ω, t), complete set C of solution components
2: **input:** CMSA parameter values for $\text{age}_{\max}$, n_{a}, t_{solver}
3: $S_{\text{bsf}} := \emptyset$
4: $C' := \emptyset$
5: $age[c_{j,a}] := 0$ for all $c_{j,a} \in C$
6: **while** CPU time limit not reached **do**
7: **for** $i = 1, \ldots, n_{\text{a}}$ **do**
8: $S :=$ ConstructSolution()
9: **for** all $c_{j,a} \in S$ **and** $c_{j,a} \notin C'$ **do**
10: $age[c_{j,a}] := 0$
11: $C' \leftarrow C' \cup \{c_{j,a}\}$
12: **end for**
13: **end for**
14: $S'_{\text{opt}} \leftarrow$ ApplyExactSolver(C', t_{solver})
15: **if** $f(S'_{\text{opt}}) > f(S_{\text{bsf}})$ **then** $S_{\text{bsf}} := S'_{\text{opt}}$
16: Adapt($C', S'_{\text{opt}}, \text{age}_{\max}$)
17: **end while**
18: **output:** S_{bsf}

The resultant simplified lexicographic function is from now on denoted as $f_{\text{sim}}()$.

Lastly, [21] also considered a combined objective function involving three criteria of the ones already presented.

3 The Proposed Algorithms

In the following, we first describe the pure variants of CMSA and the population-based metaheuristic, before the developed hybrid technique (LEARN_CMSA) is presented.

3.1 CMSA for the FFMSP

CMSA algorithms assemble solutions from a finite set C of so-called solution components. In the case of our CMSA algorithm for the FFMSP, each combination of a position j in the solution string (where $j = 1, \ldots, m$) and a letter $a \in \Sigma$ is a solution component $c_{j,a}$. That is, $C := \{c_{j,a} \mid j = 1, \ldots, m \text{ and } a \in \Sigma\}$. Any feasible solution S is a subset of C such that for each position $j = 1, \ldots, m$, S contains exactly one of the solution components from $C_j := \{c_{j,a} \mid a \in \Sigma\}$. Similarly, the sub-instance C' is always a subset of C. Note that, in the following, $f_{\text{orig}}(S) := f_{\text{orig}}(s)$, where s is the solution string which is derived in a well-defined way from the solution components in S. Moreover, let $f_{\text{orig}}(\emptyset) := 0$.

At the start of the CMSA algorithm, the best-so-far solution (S_{bsf}) and the sub-instance (C') are initialized to the empty set; see lines 3 and 4. Moreover, the so-called age value $age[c_{j,a}]$ of each solution component $c_{j,a} \in C$ is initialized to zero (see line 5). Then, at each iteration, n_{a} valid solutions are probabilistically generated in the *construct-step* of CMSA; see line 8. This is done in function ConstrucSolution() which is described below. After constructing a solution

S, all those solution components from S that do not yet form part of the sub-instance C' are added to C', and their age values are set to zero; see lines 9–12. This step of CMSA is known as the *merge-step*. Next, in the *solve-step* of CMSA, an ILP solver (in our case, CPLEX) is applied to sub-instance C' with a computation time limit of t_{solver} CPU seconds. The ILP model corresponding to sub-instance C' is obtained by adding the following constraints to the ILP model from Sect. 2:

$$x_{j,a} = 0 \quad \forall c_{j,a} \in C \setminus C'. \tag{9}$$

In other words, position-letter combinations that are not found in C'—that is, position-letter combinations whose corresponding solution component does not form part of C'—are not allowed in the ILP model. The best solution returned by CPLEX in the given computation time limit is henceforth called S'_{opt}. As a last step, the *adapt-step* of CMSA is carried out in function Adapt(C', S'_{opt}, age_{max}). This function implements a mechanism to eliminate seemingly irrelevant solution components from the sub-instance C' at each algorithm iteration. Specifically, it involves three steps: first, incrementing the age values of all vertices in C'; second, resetting the age values of all vertices in S'_{opt} to zero; and third, removing the vertices with age values exceeding age_{max} from C'. The output of CMSA, after reaching the overall computation time limit, is solution S_{bsf} (respectively its string variant).

3.2 Probabilistic Solution Construction for FFMSP

The CMSA algorithm outlined above requires a way to probabilistically generate valid solutions (in function ConstructSolution() of Algorithm 1). For this purpose, we implemented the following solution construction procedure which is pseudo-coded in Algorithm 2. Note that this is a variant of the heuristic proposed by [14]. The construction of a solution starts by determining the first position j of the solution string to which we will assign a letter. This is done in function DetermineStartingPosition() of line 4 as follows. Let $\text{occ}(i, a)$ be the number of occurrences of letter $a \in \Sigma$ at position i of all input strings, that is:

$$\text{occ}(i, a) := |\{s_k \in \Omega \mid s_k[i] = a\}|. \tag{10}$$

Furthermore, let $\text{min\_occ}(i)$ be defined as the minimal number of occurrences of any letter at position i of all input strings, that is:

$$min\_occ(i) := \min\{occ(i, a) \mid a \in Sigma\}. \tag{11}$$

Then, j (the starting position for solution construction) is determined in function DetermineStartingPosition() of

Algorithm 2 Randomized heuristic for the FFMSP

```
1: input: a problem instance (Ω, t)
2: input: parameter value for d_rate
3: S := ∅
4: j := DetermineStartingPosition()
5: k := 0
6: for k < m do
7:   Draw r ∈ [0, 1] uniformly at random
8:   if r ≤ d_rate then
9:     if ∃ a ∈ Σ s.t. f_partial(S ∪ {c_{j,a}}) > f_partial(S) then
10:      S := S ∪ {c_{j,a}}
11:    else
12:      Let b ∈ Σ s.t. occ(j, b) ≤ occ(j, t) for all t ∈ Σ
13:      S := S ∪ {c_{j,b}}
14:    end if
15:  else
16:    Draw a letter a ∈ Σ uniformly at random
17:    S := S ∪ {c_{j,a}}
18:  end if
19:  j := j + 1
20:  if j > m then j := 1 end if
21:  k := k + 1
22: end for
23: output: S
```

line 4 as follows:

$$j := argmin\{\text{min\_occ}(i) \mid i = 1, \ldots, n\}. \tag{12}$$

In the case of ties, we take—among all tied options—the one with the lowest index. Starting from j, the randomized heuristic from Algorithm 2 then assigns a letter to all positions in sequential order. When the end of the solution string is reached, j is set to one; see lines 19 and 20. In this context, note that a solution S is assembled as a set of solution components. However, assigning a solution component $c_{j,a}$ to S is exactly the same as assigning the letter a to position j of the solution S in string form. The choice of a letter for the current position j is done as follows. First, a random number r is drawn uniformly at random from [0, 1]. In case $r > d_{\text{rate}}$—where d_{rate} is an algorithm parameter called the determinism rate—a letter from Σ is chosen uniformly at random and assigned to j. Otherwise, it is first determined if a letter $a \in \Sigma$ exists such that the objective function value of the extended partial solution $S \cup \{c_{j,a}\}$—that is, $f_{\text{partial}}(S \cup \{c_{j,a}\})$—is better than the objective function value of the partial solution S; see line 9. Hereby, function $f_{\text{partial}}()$ is exactly the same as the original objective function $f_{\text{orig}}()$, just that it only considers those positions of the partial solution to be evaluated that are already occupied with a letter. If such a letter $a \in \Sigma$ exists, it is deterministically chosen and assigned to position j; see line 10. Otherwise, we deterministically choose letter $b \in \Sigma$ such that $\text{occ}(j, b) \leq \text{occ}(j, z)$ for all $z \in \Sigma$ and assign it to position j of S; see line 13. The solution construction procedure finishes once all positions of S have an assigned letter.

3.3 The Population-Based Metaheuristic for the FFMSP

With the aim of introducing learning into the solution construction mechanism of CMSA, in this paper, we present a hybridization of CMSA with a specific type of bacterial algorithm (BA) inspired by mechanisms bacteria have developed to fight antibiotics. This type of algorithm was first described in [12, 22]. In this work, we present an adaptation of this algorithm for solving the FFMSP, before describing its hybridization with CMSA. Before we delve into the algorithm description, the interested reader should note however that we believe that nearly any population-based metaheuristic can be used for the same purpose. Moreover, being aware of the abuse that natural phenomena have suffered in recent years in the context of so-called new nature-inspired optimization algorithms, the BA algorithm will be described in metaphor-free language.

Bacteria, classified as microorganisms, are tiny life forms similar to viruses, algae, fungi, and protozoa. These unicellular organisms, existing at a microscopic scale, have the ability to thrive in diverse environments such as oceans, land, space, and even the human intestine. The interaction between humans and bacteria is intricate; at times, bacterial behavior proves beneficial or even essential to human well-being, while at other times, it can lead to harmful diseases and health issues. Since penicillin was discovered in 1929 by Alexander Fleming, antibiotics have been important in the treatment of diseases caused by bacteria and other microorganisms. However, a serious problem is that, when frequently exposed to the same type of antibiotics, bacteria develop defense mechanisms to neutralize the action of antibiotics. This key mechanism for bacteria survival is achieved through communication with other members of the population and can be understood as a collaborative mechanism based on transferring DNA among bacteria. In this way, stronger bacteria may transfer their characteristics to weaker bacteria which can acquire the capability to resist the common enemy: the antibiotic.

A relevant difference between superior organisms and bacteria is the mechanism for reproduction and recombination of genetic material. In fact, populations of superior organisms vary genetically in a vertical process, that is, offspring is created as part of a new generation through sexual interaction between parents. On the other side, genetic variation in bacteria populations may happen through a horizontal process in which genetic material is transferred among individuals, without requiring the creation of a new individual. Therefore, in the context of bacteria, it seems more natural to talk about donors and receptors instead of parents and offspring. In fact, reproduction in bacteria is achieved by means of a cell division, a replication, which implies a bacteria generation containing exactly the same genetic material.

Algorithm 3 Bacteria Algorithm (BA) for the FFMSP

1: **input:** a problem instance (Ω, t)
2: **input:** parameter values for p_{size}, pr_{heur}, d_{rate}, pr_{mut}, pr_{reg}
3: $S_{\text{bsf}} := \emptyset$
4: $P :=$ GenerateInitialPopulation$(p_{\text{size}}, pr_{\text{heur}}, d_{\text{rate}})$
5: **while** CPU time limit not reached **do**
6: $S_{\text{ib}} := \text{argmin}\{f(S) \mid S \in P\}$
7: **if** $f(S_{\text{ib}}) > f(S_{\text{bsf}})$ **then** $S_{\text{bsf}} := S_{\text{ib}}$
CONJUGATION PHASE
8: $S_{\text{level}} :=$ DetermineSeparationLevel(P)
9: $(P_{\text{donor}}, P_{\text{receptor}}) :=$ Classification(S_{level}, P)
10: $P_{\text{receptor}} :=$Conjugation$(P_{\text{donor}}, P_{\text{receptor}}, pr_{\text{mut}})$
11: $P := P_{\text{donor}} \cup P_{\text{receptor}}$
REGENERATION PHASE
12: $S_{\text{level}} :=$ DetermineSeparationLevel(P)
13: $(P_{\text{donor}}, P_{\text{receptor}}) :=$ Classification(S_{level}, P)
14: $P_{\text{receptor}} :=$ Regeneration$(P_{\text{donor}}, pr_{\text{reg}})$
15: $P := P_{\text{donor}} \cup P_{\text{receptor}}$
16: **end while**
17: **output:** S_{bsf}

Eventually, this process may be affected by a mistake in the replication process or by the influence of an external agent, a mutagen.

As previously indicated, the emergence of antibiotic resistance poses a significant concern for humans. Nevertheless, for bacteria, it signifies an evolutionary advantage that enhances their ability to survive. Viewed in this light, this particular bacterial behavior serves as a valuable source of inspiration for optimization processes, as demonstrated in [12, 22]. This is especially true for the *horizontal transfer* of DNA material, referring to the sharing of genetic material within the local community, i.e., among neighbors belonging to the same generation. In the following, we show how these principles were applied to solving the FFMSP.

The pseudo-code of the bacterial algorithm (BA) is provided in Algorithm 3. Apart from a problem instance (Ω, t), the algorithm requires the following five parameters as input (see lines 1 and 2):

1. Population size (p_{size})
2. Rate of initial solutions generated by the probabilistic heuristic (pr_{heur})
3. Determinism rate used by the probabilistic heuristic (d_{rate})
4. Probability of mutation during the conjugation phase (pr_{mut})
5. Probability of mutation during the regeneration phase (pr_{reg})

At the start of the algorithm, the best-so-far solution (S_{bsf}) is set to the empty set. Then, the initial population of solutions of size p_{size} (where each solution corresponds to a bacterium) is generated in function GenerateInitialPopulation$(p_{\text{size}}, pr_{\text{heur}}, d_{\text{rate}})$. Hereby, with probability pr_{heur}, a solution is generated by means of the randomized heuris-

tic from Algorithm 2. Otherwise, the solution is generated uniformly at random. The first action of each iteration consists of the determination of the iteration-best solution S_{ib} from the current population (see line 6) to update the best-so-far solution if necessary (line 7). Then, the two main procedures—conjugation and regeneration—of the BA are executed. Both procedures start in the same way (see lines 8 and 9, respectively lines 12 and 13). In particular, the current population P is divided into two parts: donor solutions (P_{donor}) and receptor solutions (P_{receptor}). For this purpose, first, a separator level (S_{level}) is determined as follows in function DetermineSeparatorLevel(P). Two pairs of solutions—say, (S_i, S_j) and (S_k, S_l)—are chosen uniformly at random from the current population P. Then, the best solution from each pair is selected. Let $S_1 := \text{argmax}\{f(S_i), f(S_j)\}$ and $S_2 := \text{argmax}\{f(S_k), f(S_l)\}$. Furthermore, let $S_{\min}$ be the lower quality solution between S_1 and S_2, that is, $S_{\min} := \text{argmin}\{f(S_1), f(S_2)\}$. Then S_{level} is defined as $f(S_{\min})$. Thus, a low-cost procedure is used to choose a solution with a fitness value close to the population's median. S_{level} is then used to divide the current population into a sub-population P_{donor} of donor solutions and a sub-population P_{receptor} of recipient solutions. Hereby, the solutions in P_{receptor} have an objective function value below S_{level}, and donor solutions have an objective function value of at least S_{level}.

In the *conjugation step* of BA, all receptor solutions from P_{receptor} receive a random piece of genetic material from some randomly chosen donor solution from P_{donor}. This piece of genetic material corresponds to a contiguous section of arbitrary size extracted from the donor solution. As in nature, this operation may suffer a corruption in the genetic transcription (mutation). Thus, each letter of the transferred sub-string may be changed to any other letter with a probability pr_{mut}.

In contrast, in the *regeneration step* of BA, after classifying the members of the current population into donors P_{donor} and receptors P_{receptor}, all solutions from P_{receptor} are exchanged with clones of randomly chosen donor solutions after applying mutation with probability pr_{reg} to each position.

These steps are iterated until a computation time limit is reached. After termination, the best-so-far solution S_{bsf} is provided as output.

3.4 The LEARN_CMSA Algorithm

The pseudo-code of the hybridization between CMSA and BA is provided in Algorithm 4. In addition to the CMSA and BA parameters, as outlined before, it takes the following two parameters, which regulate the interplay between CMSA and BA, as input:

Algorithm 4 LEARN_CMSA for the FFMSP

1: **input:** problem instance (Ω, t), complete set C of solution components
2: **input:** values for CMSA parameters $\text{age}_{\max}$, n_{a}, t_{solver}
3: **input:** values for BA parameters p_{size}, pr_{heur}, d_{rate}, pr_{mut}, pr_{reg}
4: **input:** values for CMSA/BA interplay parameters b_{iter}, r_{inject}
5: $S_{\text{bsf}} := \emptyset$
6: $C' := \emptyset$
7: $age[c_{j,a}] := 0$ for all $c_{j,a} \in C$
8: $P :=$ GenerateInitialPopulation(p_{size}, pr_{heur}, d_{rate})
9: **while** CPU time limit not reached **do**
10: $P :=$ Execute_BA_Algorithm(P, b_{iter}, p_{size}, pr_{heur}, d_{rate}, pr_{mut}, pr_{reg})
11: $T :=$ Extract_From(P, n_{a})
12: **for** all $S \in T$ **do**
13: **for** all $c_{j,a} \in S$ **and** $c_{j,a} \notin C'$ **do**
14: $age[c_{j,a}] := 0$
15: $C' \leftarrow C' \cup \{c_{j,a}\}$
16: **end for**
17: **end for**
18: $S'_{\text{opt}} \leftarrow$ ApplyExactSolver(C', t_{solver})
19: **if** $f(S'_{\text{opt}}) > f(S_{\text{bsf}})$ **then** $S_{\text{bsf}} := S'_{\text{opt}}$
20: Adapt(C', S'_{opt}, $\text{age}_{\max}$)
21: $P :=$ InjectSolverSolution(P, S_{bsf}, r_{inject})
22: **end while**
23: **output:** S_{bsf}

1. b_{iter}: number of BA iterations executed in function Execute_BA_Algorithm(P, b_{iter}, p_{size}, pr_{heur}, d_{rate}, pr_{mut}, pr_{reg}) at each CMSA iteration; see line 10.
2. r_{inject}: the rate of injection of the solution returned by the ILP solver (S_{bsf}) into the current BA population in function InjectSolverSolution(P, S_{bsf}, r_{inject}); see line 21.

The differences to the pure CMSA algorithm from Sect. 3.1 are as follows. First, in addition to the initialization of CMSA in lines 5–7, the initial population P of the BA algorithm is generated in line 8 in the same way as explained before in the context of the BA algorithm in Sect. 3.3. Next, at the beginning of each CMSA iteration function Execute_BA_Algorithm(P, b_{iter}, p_{size}, pr_{heur}, d_{rate}, pr_{mut}, pr_{reg}) executes b_{iter} iterations of the BA algorithm exactly in the same way as explained in Sect. 3.3. This function returns the current BA population P as output. Then, function Extract_From(P, n_{a}) (see line 11) extracts exactly n_{a} solutions from the current BA population P and stores them in set T. In particular, T contains the best solution from P, in addition to $(n_{\text{a}} - 1)$ randomly selected donor solutions from P. Note that, for this purpose, the separator level (S_{level}) is determined and the population P is divided into donors and receptors, as outlined in Sect. 3.3. The solutions from T are then added to the current sub-instance C' of CMSA, replacing the randomized solution construction procedure of standard CMSA. Finally, at the end of each LEARN_CMSA iteration, the solution S_{bsf} returned by the solver after being applied to the current sub-instance C' is used to replace $\lfloor r_{\text{inject}} \cdot |P_{\text{receptor}}| \rfloor$

receptor solutions from P. This is done in function InjectSolverSolution(P, S_{bsf}, r_{inject}); see line 21.

Note that in this way of hybridizing CMSA and BA, both memory mechanisms—the sub-instance C' of CMSA and the population P of BA—influence each other. In particular, a set of donor solutions from P is added to C' at each iteration, while CMSA influences BA by injecting S_{bsf} into the BA population P.

4 Experimental Evaluation

CMSA, BA, and LEARN_CMSA were implemented in C++ using GCC 11.3.0 for compiling the software. Experiments were all performed on a cluster (Luthier) of the Engineering Faculty of the University of Concepción, Chile, in single-threaded mode. Luthier is composed of 30 computing nodes (servers). All nodes have an Intel®CPU Xeon®E3-1270 v6 at 3.8 GHz with 64 GB RAM. Moreover, all ILP models were solver with IBM ILOG CPLEX version 20.1.

In this section, we first provide an overview of the benchmark datasets used. Next, we detail the parameter tuning procedure employed to optimize the configuration of BA, CMSA and LEARN_CMSA. Finally, we present the numerical results.

4.1 Benchmark Sets

Various benchmark instance sets for the FFMSP have been introduced by different authors, and for this study, we adopt the following set. [19] employed a collection of instances comprising 100 problem instances with randomly generated input strings over alphabet $\Sigma = \{\texttt{A}, \texttt{C}, \texttt{T}, \texttt{G}\}$ for each combination of $n \in \{100, 200, 300, 400\}$ and $m \in \{200, 600, 800\}$. This set, totaling 1200 problem instances, is referred to as `Ferone` from hereon. It is worth noting that all instances were previously solved in other studies with thresholds $t \in \{0.75m, 0.8m, 0.85m\}$. A subset of these instances—specifically those with $n \in \{100, 200\}$—was already utilized in earlier publications [10, 18].

A set of problem instances with specifications akin to those introduced by [19] was presented in [20]. However, in contrast to the 100 random instances per combination of n and m in the former, the set by [20] comprises only five random instances per combination. Furthermore, this set is limited to $n \in \{100, 200\}$. The collection of random instances from [20] is hereinafter referred to as `Gallardo`.

It is important to highlight that in this study, we address all the outlined problem instances for $t \in \{0.8m, 0.85m\}$. The threshold $t = 0.75m$ was excluded from consideration due to our observation (similar to findings in prior works) that problems resulting from this threshold are easily solved optimally. Also, we perform a comparison only in the range of $n \in \{100, 200\}$ to maintain a complete comparison between all algorithms from the state-of-the-art.

4.1.1 New Datasets

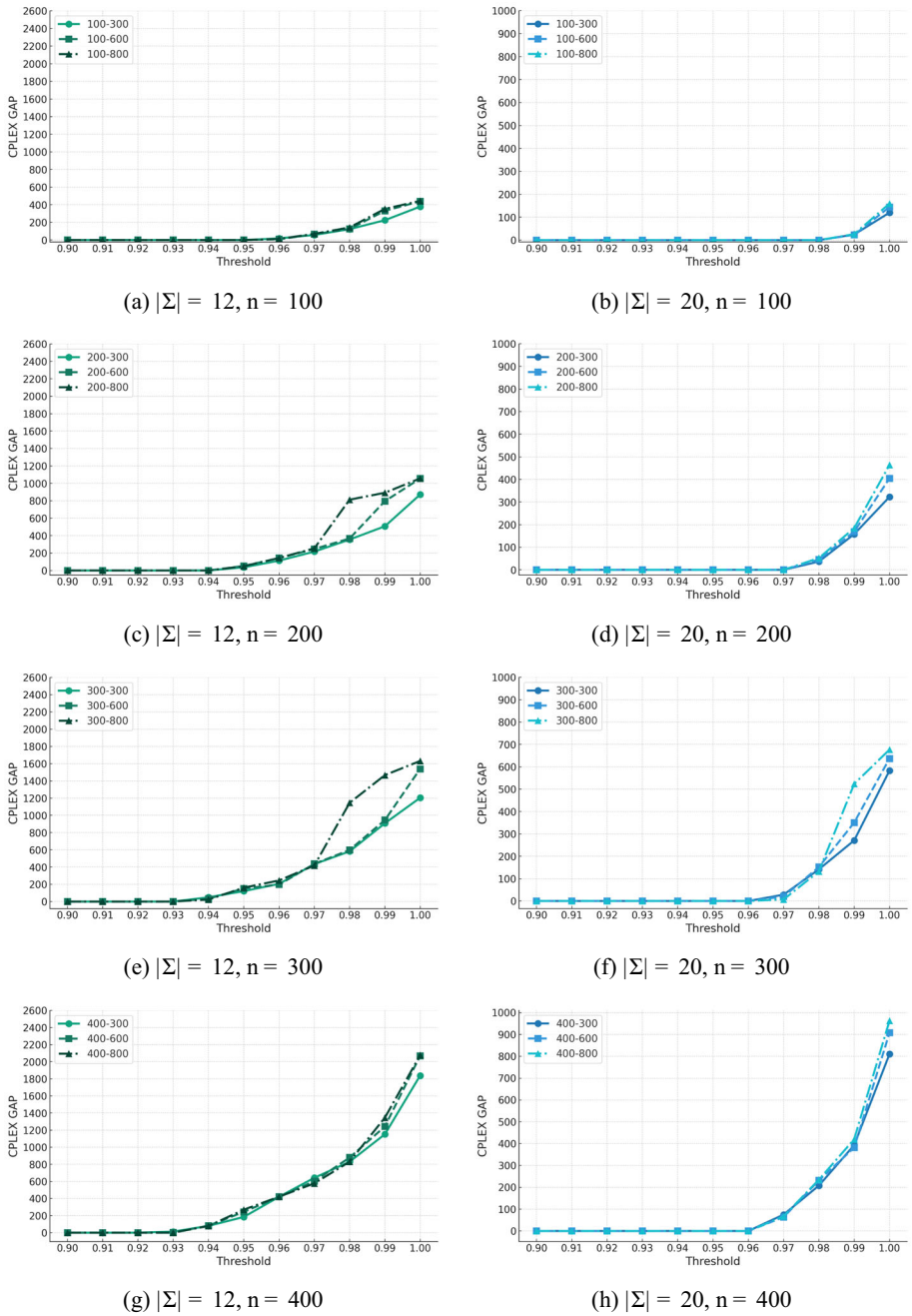
We further plan to investigate the performance of our algorithms on problem instances of different alphabet sizes, specifically focusing on sizes 12 and 20. Suitable threshold values were determined based on an analysis of the optimality gaps produced by CPLEX after a computation time of 600 s per instance. This served as an indicator of the problem's computational complexity and ensured that the chosen thresholds were high enough to produce challenging problem instances.

Threshold determination was performed for $\Sigma = \{12, 20\}$, using thresholds t spanning from 0.80 to 1.0 in increments of 0.01. Test instances were defined by combinations of $n = \{100, 200, 300, 400\}$ and $m = \{300, 600, 800\}$. For each parameter combination (Σ, n, m), we randomly selected one instance and solved it using CPLEX for every specified threshold. The outcomes of these experiments (in terms of the optimality gaps) can be found in Fig. 1. Note that, in the legends, the first integer after the keyword 'cplex' indicates the value of n and the second one the value of m.

Based on the observed behavior, we have established threshold values for the 12-character alphabet at $t = (0.95m, 0.97m, 1.0m)$. For the 20-character alphabet, the thresholds are set at $t = (0.98m, 0.99m, 1.0m)$. We chose a higher starting value for the 20-character set, because the figures clearly show that the larger alphabet presents a simpler problem for CPLEX. Finally, the new dataset for $\Sigma = \{12, 20\}$ is composed of 100 problem instances for each combination of $|\Sigma| \in \{12, 20\}$, $n \in \{100, 200, 300, 400\}$ and $m \in \{300, 600, 800\}$, which makes a total of 2400 problem instances.

4.2 Algorithm Tuning

The adjustment of the parameter values of BA, CMSA, and LEARN_CMSA was performed with the tuning tool `irace` [23]. During preliminary experiments, we noticed changes concerning the parameter value requirements of the tested algorithms for different threshold values. Another important factor of a problem instance is n, the number of input strings. Therefore, we decided to tune parameters separately for small problem instances ($n = \{100, 200\}$) and large problem instances ($n = \{300, 400\}$). In this way, we perform 16 different tuning processes for each algorithm, one for each combination of $|\Sigma|$, n (small vs. large), and t. As tuning instances, we use the first (out of 100 instances) for each case from the Ferone dataset (for $|\Sigma| = 4$), respectively from our new datasets (for $|\Sigma| \in \{12, 20\}$).



(a) $|\Sigma| = 12$, n = 100 (b) $|\Sigma| = 20$, n = 100

(c) $|\Sigma| = 12$, n = 200 (d) $|\Sigma| = 20$, n = 200

(e) $|\Sigma| = 12$, n = 300 (f) $|\Sigma| = 20$, n = 300

(g) $|\Sigma| = 12$, n = 400 (h) $|\Sigma| = 20$, n = 400

Fig. 1 Study of the evolution of the optimality gaps produced by CPLEX for a range of different threshold values (x-axis)

The algorithm parameters and the corresponding domains are detailed in Tables 1a, b, and 2. Additionally, `irace` was allocated a budget of 5000 runs, each run with a time limit of 600 s. The parameter configurations identified by `irace` are presented in the respective tables.

Analysis of the LEARN_CMSA *tuning results.* In Table 2, we can observe the tuning results obtained with `irace` for LEARN_CMSA. Considering the case with $|\Sigma| = 4$, there is a significant difference in behavior between $t = 0.8m$ and $t = 0.85m$. At $t = 0.8m$, a collaborative interaction between BA and CPLEX can be noticed. Such an interaction is characterized by the allocation of shorter computation times to the ILP solver (t_{solver}), a lower maximum age for solution components (age_{max}), and a fewer number of extracted solutions from BA for feeding the sub-instance of LEARN_CMSA (n_{a}). This limits the introduction of new solution components per iteration, thereby maintaining a more controlled sub-instance size. Furthermore, BA seems to play an important role as it works with higher quality initial solutions (see the high value for heuristic initialization pr_{heur}) and with high levels of determinism (d). This is not the case with $t = 0.85m$, where the process seems to be more of a pipeline between BA and

Table 1 Parameters of CMSA (a) and BA (b) together with their domains considered for the tuning process and chosen values for all datasets

(a) Parameter tuning of CMSA

	CMSA	$\|\Sigma\| = 4$				$\|\Sigma\| = 12$						$\|\Sigma\| = 20$					
		$n = \{100, 200\}$		$n = \{300, 400\}$		$n = \{100, 200\}$			$n = \{300, 400\}$			$n = \{100, 200\}$			$n = \{300, 400\}$		
Parameter	Considered domain	$t = 0.8$	$t = 0.85$	$t = 0.8$	$t = 0.85$	$t = 0.95$	$t = 0.97$	$t = 1.0$	$t = 0.95$	$t = 0.97$	$t = 1.0$	$t = 0.98$	$t = 0.99$	$t = 1.0$	$t = 0.98$	$t = 0.99$	$t = 1.0$
$\text{Age}_{\max}$	{1, 2, 3, 5, 10, 50, 1000}	50	50	50	1000	1000	1	10	50	1000	3	1	50	10	2	3	3
n_a	{1, 2, 3, 4, 5, 10, 20, 50, 100}	50	100	20	50	50	100	50	100	50	50	100	100	100	100	100	100
t_{solver}	{1, 3, 5, 10, 20, 50, 100, 200, 300, 400, 500}	500	500	500	400	500	400	400	500	500	500	500	500	200	500	500	500
d	[0.0, 1.0]	0.6	0.73	0.18	0.61	0.3	0	0.49	0.32	0.01	0.19	0.3	0.09	0.19	0.51	0.09	0.03
o_f	{0, 1, 2}	0	1	1	0	1	1	2	0	2	2	1	2	2	1	2	0
CPL_{stop}	{0, 1}	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0

(b) Parameter tuning of BA

	BA	$\|\Sigma\| = 4$				$\|\Sigma\| = 12$						$\|\Sigma\| = 20$					
		$n = \{100, 200\}$		$n = \{300, 400\}$		$n = \{100, 200\}$			$n = \{300, 400\}$			$n = \{100, 200\}$			$n = \{300, 400\}$		
Parameter	Considered domain	$t = 0.8$	$t = 0.85$	$t = 0.8$	$t = 0.85$	$t = 0.95$	$t = 0.97$	$t = 1.0$	$t = 0.95$	$t = 0.97$	$t = 1.0$	$t = 0.98$	$t = 0.99$	$t = 1.0$	$t = 0.98$	$t = 0.99$	$t = 1.0$
p_{size}	{10, 100, 500, 1000, 2000, 5000, 10000}	1000	10000	100	10000	100000	5000	10000	500	500	10000	10000	2000	10000	500	5000	10000
d_{rate}	[0.0, 1.0]	0.99	0.85	0.64	0.81	0.99	0.98	0.82	1.0	0.92	0.81	1.0	0.99	0.94	1.0	0.96	0.89
pr_{reg}	[0.0, 1.0]	0.01	0.02	0.01	0.02	0.01	0.01	0.71	0.01	0.02	0.5	0.01	0.01	0.58	0.01	0.01	0.53
pr_{mut}	[0.0, 1.0]	0.74	0.39	0.52	0.86	0.75	0.53	0.66	0.79	0.33	0.54	0.28	0.43	0.32	0.9	0.65	0.1
pr_{heur}	[0.0, 1.0]	0.72	0.38	0.16	0.83	0.67	0.9	0.78	0.78	0.65	0.91	0.85	0.26	0.61	0.7	0.75	0.53
o_f	{0, 1, 2}	1	1	1	1	1	1	1	1	1	2	1	1	1	1	1	1

Table 2 Algorithm parameters for LEARN_CMSA and their domains considered for the tuning process with `irace` for all datasets

		$\|\Sigma\| = 4$				$\|\Sigma\| = 12$						$\|\Sigma\| = 20$					
		$n = \{100, 200\}$		$n = \{300, 400\}$		$n = \{100, 200\}$			$n = \{300, 400\}$			$n = \{100, 200\}$			$n = \{300, 400\}$		
Parameter	Considered domain	$t = 0.80$	$t = 0.85$	$t = 0.80$	$t = 0.85$	$t = 0.95$	$t = 0.97$	$t = 1.0$	$t = 0.95$	$t = 0.97$	$t = 1.0$	$t = 0.98$	$t = 0.99$	$t = 1.0$	$t = 0.98$	$t = 0.99$	$t = 1.0$
$\text{Age}_{\max}$	{1, 2, 3, 5, 10, 50, 1000}	1	50	1	50	1	5	2	3	3	5	3	5	50	50	10	50
n_{a}	{1, 2, 3, 4, 5, 10, 20, 50, 100}	4	20	5	20	100	100	100	2	100	100	100	100	100	100	100	100
t_{solver}	{1, 3, 5, 10, 20, 50, 100, 200, 300, 400, 500}	2	400	1	400	10	500	20	1	500	400	500	500	500	500	500	50
b_{iter}	{1, 2, 3, 5, 10, 30, 50, 75, 100, 200, 500, 1000}	1	1000	2	5	1	500	3	2	30	1	1	2	50	50	10	50
r_{inject}	[0.0, 1.0]	0.02	0.58	0.03	0.7	0.14	0.46	0.28	0.02	0.4	0.97	0.47	0.76	0.75	0.29	0.21	0.53
p_{size}	{10, 100, 500, 1000, 2000, 5000, 10000}	500	1000	100	500	1000	1000	500	100	1000	1000	5000	10000	1000	2000	2000	100
d	[0.0, 1.0]	0.71	0.74	0.84	0.34	0.79	0.66	0.51	0.26	0.56	0.4	0.59	0.54	0.92	0.85	0.13	1
pr_{reg}	[0.0, 1.0]	0.35	0.49	0.11	0.71	0.82	0.78	0.7	0.48	0.61	0.43	0.6	0.79	0.37	0.97	0.46	0.16
pr_{mut}	[0.0, 1.0]	0.02	0.16	0.31	0.13	0.33	0.86	0.05	0.75	0.04	0.79	0.71	0.75	0.92	0.87	0.96	0.69
pr_{heur}	[0.0, 1.0]	0.83	0.08	0.84	0.18	0.39	0.76	0.46	0.81	0.03	0.87	0.11	0.04	0.86	0.09	0.21	0.18
o_f	{0, 1, 2}	1	2	1	2	1	2	2	1	1	0	1	0	1	1	0	1
CPL_{stop}	{0, 1}	1	0	0	0	0	0	1	0	0	1	0	0	1	0	0	1

CPLEX, evident from the high computation time allocated to CPLEX and a larger number of solutions (n_{a}) extracted for inclusion in the sub-instance. This results in larger and more complex sub-instances which demand longer execution times by the solver.

Concerning $|\Sigma| = 12$, the analysis is less obvious. In some cases where the CPLEX time is limited (($t = 0.95m$, small), ($t = 0.97m$, small), and ($t = 1.0m$, large)), the allocated number of solutions extracted from BA (n_{a}) does not seem consistent, resulting in potentially large instances ($t = 0.95m$, small). Also, these cases are not accompanied with a high determinism for BA (see the values for d and pr_{heur}), unlike what was observed for $Sigma = 4$. This could be attributed to the fact that, with a growing alphabet size, the problem seems to become easier, allowing `irace` more flexibility for the parameter value selection. In the pipeline scenario, the parameters appear clearer, especially when solver times are high (($t = 0.97m$, small), ($t = 0.97m$, large), ($t = 1.0m$, large)), resorting to extracting the maximum number of solutions from BA (n_{a}) at each iteration.

Finally, concerning $|\Sigma| = 20$, it also appears that a pipeline approach is preferred with extended solver times (t_{solver}) and the maximum number of solutions extracted from BA (n_{a}), particularly in small cases, and for ($t = 0.98m$, large) and ($t = 0.99m$, large). An exception is observed for ($t = 1.0m$, large), which features a more limited computation time for the solver, even though it deals with a potentially larger sub-instance (maximum value for n_{a}).

Upon examining the tuning results concerning the chosen objective functions, it is evident that there is a tendency to utilize the enhanced functions $o_f = 1$ (f_{blu}) and $o_f = 2$ (f_{sim}) in over 80% of the configurations. This aligns with the previous assertion that these functions enhance the performance of the search process compared to the original objective function (f_{orig}). Hereby, note that f_{blu} is the most frequently used objective function (56%). Another interesting observation is that in cases where the CPLEX time is shorter, leading to greater collaboration between BA and CPLEX within the framework, the percentage of solution integration from CPLEX (r_{inject}) is lower over BA than in cases operating in a pipeline manner. This results in a reduced influence of CPLEX solutions on the bacteria population.

Analysis of the CMSA *tuning results.* In Table 1a, the parameter values selected by `irace` for CMSA are presented. The prevailing configuration across these instances is marked by important computation time allocations to CPLEX, coupled with a considerable number of heuristic solutions generated at each iteration. These solutions exhibit low levels of determinism, causing a notable variability in the components generated. Consequently, this leads to the creation of large sub-instances that demand extensive computational time in a pipeline-oriented setup of the approach. It is also noteworthy that the enhanced objective functions

Table 3 Numerical results concerning instances from the `Ferone` set

n	m	t	CPLEX	HYACO	GRASP	$\text{ACO}^{+}_{\text{neg}}$	LEARN_CMSA	CMSA	BA
	300	$0.8m = 240$	70.77	77.84	76.26	84.07	85.24	71.08	69.53
100	600	$0.8m = 480$	72.11	72.97	77.53	88.12	89.25	72.65	69.09
	800	$0.8m = 640$	72.41	70.94	82.17	89.23	90.48	72.69	67.34
	300	$0.8m = 240$	87.40	104.17	94.71	107.41	113.63	89.91	85.09
200	600	$0.8m = 480$	80.30	85.02	80.94	102.42	110.4	86.23	76.33
	800	$0.8m = 640$	76.38	77.95	85.71	97.67	106.62	80.68	63.89
	300	$0.8m = 240$	89.42	n.a	112.83	120.16	125.53	102.55	71.45
300	600	$0.8m = 480$	69.74	n.a	83.12	104.95	119.23	86.77	75.14
	800	$0.8m = 640$	66.51	n.a	90.26	92.88	114.53	82.77	56.68
	300	$0.8m = 240$	92.8	n.a	119.32	129.29	137.39	109.66	78.43
400	600	$0.8m = 480$	50.72	n.a	85.99	103.86	125.36	70.08	81.00
	800	$0.8m = 640$	48.67	n.a	92.86	89.05	111.58	61.19	62.15
	300	$0.85m = 255$	25.43	28.30	29.54	30.53	30.9	25.56	10.45
100	600	$0.85m = 510$	23.84	22.82	27.47	27.53	27.98	23.85	1.36
	800	$0.85m = 680$	24.10	21.66	26.54	26.61	26.82	24.33	0.57
	300	$0.85m = 255$	23.19	28.59	30.37	32.32	33.04	22.04	8.42
200	600	$0.85m = 510$	22.14	21.90	26.35	27.31	27.95	22.91	0.56
	800	$0.85m = 680$	22.29	20.40	24.42	25.83	26.78	23.24	0.06
	300	$0.85m = 255$	20.92	n.a	31.83	31.33	33.33	21.26	17.05
300	600	$0.85m = 510$	21.87	n.a	24.95	24.68	28.39	22.09	0.61
	800	$0.85m = 680$	22.5	n.a	23.53	20.68	26.88	22.46	0.06
	300	$0.85m = 255$	20.98	n.a	32.78	31.55	33.76	21.32	16.06
400	600	$0.85m = 510$	19.67	n.a	24.56	24.80	28.27	17.61	0.39
	800	$0.85m = 680$	20.53	n.a	22.82	15.86	26.71	18.53	0.02

are more frequently utilized than their original counterparts, albeit at a marginally lower rate than in LEARN_CMSA (by a difference of 0.75%).

Analysis of the BA tuning results. Table 1b presents the parameter values selected by `irace` for BA. Notably, each configuration tends to exhibit high—sometimes even reaching the maximum—population sizes (p_{size}). The recommended configurations predominantly resort to heuristic initialization, showcasing medium-to-high determinism levels (d_{rate}). Within its operational framework, the BA is characterized by employing elevated mutation rates for its recombination operator (pr_{mut}), while maintaining very low mutation rates during regeneration phases (pr_{reg}). A salient observation is BA's consistent preference for enhanced objective functions, eschewing the original altogether. This preference is rationalized by the potential inadequacy of the original function to differentiate between solutions. It might happen, for example, that in a population of random individuals, all have an original objective function value of zero. Again, function f_{blu} ($o_f = 1$) emerges as the primary choice, being utilized in 93% of cases.

4.3 Results

The LEARN_CMSA, CMSA and BA algorithms were applied precisely once to each of the 1,200 problem instances from the Ferone dataset (Table 3) and 10 times to each of the 60 problem instances from the Gallardo dataset (Table 4). These choices are due to the way in which existing algorithms from the literature were evaluated on these datasets. A computational time constraint of 600 CPU seconds was imposed for all executions across both datasets. The tables present the results in terms of average objective function values. For the Ferone dataset, the displayed value represents the average objective value across problem instances sharing the same (n, m) combination. In contrast, for the Gallardo dataset, the value signifies the average result over the 10 repeated runs for all available instances.

In the case of the Ferone dataset, the results of LEARN_CMSA, CMSA and BA are compared with CPLEX (standalone and with the same computation limit as LEARN_CMSA, CMSA and BA), a hybrid ACO algorithm called HYACO from [10], the best performing of six variants of GRASP from [19], and the current state-of-the-art algorithm called $\text{ACO}^{+}_{\text{neg}}$ from [24]. GRASP variants use a time limit of 30 CPU seconds. This

Table 4 Numerical results for the instances from the `Gallardo` set

n	m	t	CPLEX	MA	$\text{GRASP}_{\text{fer}}$	$\text{GRASP}_{\text{mou}}$	$\text{ACO}^{+}_{\text{neg}}$	LEARN_CMSA
	300	$0.8m = 240$	69.82	84.82	80.78	70.99	83.14	84.12
100	600	$0.8m = 480$	72.16	87.08	79.12	70.83	86.90	88.36
	800	$0.8m = 640$	71.7	89.90	79.52	71.08	89.80	90.95
	300	$0.8m = 240$	88.22	109.58	105.85	83.04	106.00	113.36
200	600	$0.8m = 480$	81.4	101.23	88.95	80.90	102.48	110.89
	800	$0.8m = 640$	79.2	93.92	80.09	79.77	97.40	106.31
	300	$0.85m = 255$	24.98	32.58	18.41	30.10	30.50	31.02
100	600	$0.85m = 510$	23.6	28.76	4.89	25.36	27.38	28.08
	800	$0.85m = 680$	23.1	27.96	2.58	24.33	26.74	27.08
	300	$0.85m = 255$	24.14	34.49	14.85	32.69	32.32	32.42
200	600	$0.85m = 510$	22.38	26.17	2.26	25.54	27.10	28.4
	800	$0.85m = 680$	22.36	25.61	0.60	23.71	26.00	26.74
		AVERAGE	50.29	61.84	46.49	51.53		63.98

might seem unfair as LEARN_CMSA, CMSA and BA were allowed 600 CPU seconds per run. However, the low time limit of 30 CPU seconds was chosen by the authors of [19] because GRASP did not take advantage from longer running times. It is important to note that LEARN_CMSA outperforms the competing algorithms in all problem configurations within the Ferone dataset. Not only does LEARN_CMSA demonstrate superior performance compared to other heuristic approaches, but it also surpasses the standalone CPLEX component, the traditional implementation of CMSA, and the BA. In other words, LEARN_CMSA clearly seems to profit from the synergy between CMSA and BA, which are the algorithmic components of LEARN_CMSA.

In the case of the Gallardo dataset, again a computation time limit of 600 s per run were employed. Table 4 offers a comparison of our algorithms (LEARN_CMSA, CMSA and BA) with the standalone-application of the CPLEX component, a Memetic Algorithm (MA) from [20], Ferone's GRASP proposal [18] ($\text{GRASP}_{\text{fer}}$), Mousavi's GRASP proposal [17] ($\text{GRASP}_{\text{mou}}$), and the $\text{ACO}^{+}_{\text{neg}}$ approach from [24] ($\text{ACO}^{+}_{\text{neg}}$). Based on the experimental results, the following observations can be made:

- In main terms, the relative comparison between the algorithms allows very similar conclusions as in the case of the Ferone dataset. LEARN_CMSA is significantly better than the MA for threshold value $t = 0.80m$.
- When $t = 0.85m$, LEARN_CMSA results competitive with respect to the MA, being better than this last one in large instances of the problem.

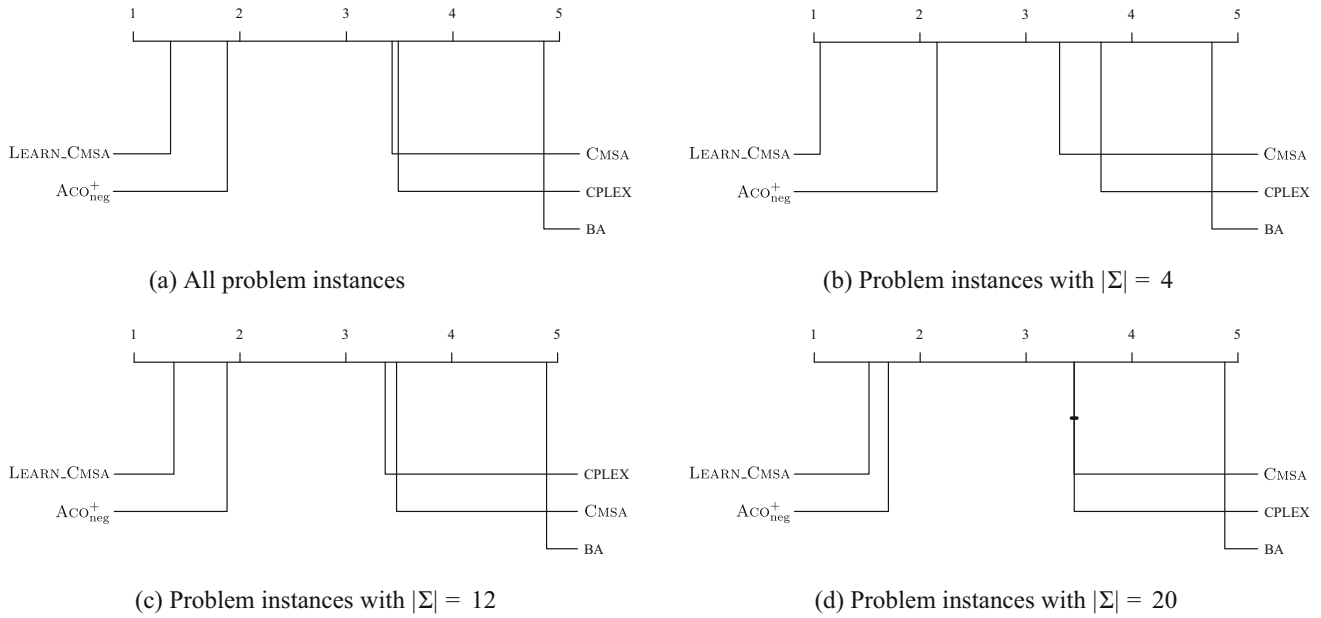
Note that the average performance of the LEARN_CMSA algorithm is better than the one of all other algorithms for the Gallardo dataset.

Finally, Table 5 presents the numerical results of our methods (LEARN_CMSA, CMSA and BA), CPLEX and $\text{ACO}^{+}_{\text{neg}}$ for our new dataset featuring instances on alphabets of sizes 12 and 20. The table clearly shows that LEARN_CMSA outperforms not only $\text{ACO}^{+}_{\text{neg}}$—which is, as mentioned before, one of the current state-of-the-art approaches—but also the individual algorithm components (CMSA and BA) of LEARN_CMSA. Nevertheless, $\text{ACO}^{+}_{\text{neg}}$ exhibits its competitiveness in the context of specific scenarios including, for example, ($n = 200, t = 0.95m, \Sigma = 12$) and ($n = 300, t = 1.0m, \Sigma = 20$). Nonetheless, on average, LEARN_CMSA is superior to all considered algorithms for both alphabet sizes. CMSA beats CPLEX by a narrow margin and falls short when juxtaposed with $\text{ACO}^{+}_{\text{neg}}$ and LEARN_CMSA. Importantly, BA's performance is suboptimal when operating in a standalone manner. In summary, it can again be observed that the synergies between CMSA and BA are exploited very well in LEARN_CMSA.

For a statistical validation of the results, we aimed to test for performance differences among the algorithms across the encompassed datasets. For this validation, we included the results of $\text{ACO}^{+}_{\text{neg}}$ (the strongest competitor of LEARN_CMSA), standalone CPLEX, LEARN_CMSA, and the individual components of LEARN_CMSA (CMSA, BA). Initially, a simultaneous comparative analysis was conducted via the Friedman test. Following the rejection of the null hypothesis postulating equivalent performance across all algorithms, pair-wise assessments were undertaken through the Nemenyi post hoc test [25]. The findings are illustrated in Fig. 2, via critical difference (CD) plots. These plots spatially position each method based on its mean ranking across the considered instance (sub)set. The CD for a significance level of 0.05 is computed, with algorithmic performances deemed statistically indistinguishable if differing less than the CD, as depicted by the horizontal bars connecting the algorithms in

Table 5 Numerical results for our new dataset of instances with $|\Sigma| \in \{12, 20\}$

n	m	$\Sigma = 12$						$\Sigma = 20$					
		t	CPLEX	$\text{ACO}^{+}_{\text{neg}}$	LEARN_CMSA	CMSA	BA	t	CPLEX	$\text{ACO}^{+}_{\text{neg}}$	LEARN_CMSA	CMSA	BA
	300	0.95m = 285	100	100	100	100	96.32	0.98m = 294	100	100	100	100	98.06
100	600	0.95m = 570	100	100	100	100	98.32	0.98m = 588	100	100	100	100	99.03
	800	0.95m = 760	100	100	100	100	98.67	0.98m = 784	100	100	100	100	99.38
	300	0.97m = 291	61.25	68.54	69.73	57.45	47.53	0.99m = 297	81.05	84.38	85.25	81.22	62.62
100	600	0.97m = 582	61.47	65.6	66.37	59.09	27.6	0.99m = 594	80.25	82.77	83.5	80.63	44.20
	800	0.97m = 776	59.40	64.35	64.6	59.7	19.49	0.99m = 792	79.21	81.87	82.42	80.04	35.26
	300	m = 300	19.54	21.67	22.11	19.1	0	m = 300	42.89	48.23	49.17	42.36	3.95
100	600	m = 600	17.20	18.97	18.97	17.16	0	m = 600	37.77	42.36	42.14	37.68	0
	800	m = 800	16.48	17.88	17.98	16.45	0	m = 800	35.66	40.59	39.46	36.24	0
	300	0.95m = 285	133.28	151.2	148.35	139.17	131.99	0.98m = 294	136.04	156.13	155.74	138.5	125.56
200	600	0.95m = 570	131.17	151.66	146.68	137.8	110.11	0.98m = 588	132.54	147.85	151.29	136.27	91.30
	800	0.95m = 760	130.25	150.3	147.38	132.65	101.43	0.98m = 784	129.86	143.66	149.58	128.68	78.27
	300	0.97m = 291	64.98	77.77	80.2	54.33	45.53	0.99m = 297	76.87	98.15	99.29	73.13	59.30
200	600	0.97m = 582	58.70	69.24	70.54	55.54	14.33	0.99m = 594	67.39	87.83	91.13	72.41	18.67
	800	0.97m = 776	56.41	67.4	67.67	56.13	5.36	0.99m = 792	45.73	86.02	88.16	71.72	8.41
	300	m = 300	19.54	21.67	22.34	19.21	0	m = 300	43.85	51.02	51.42	42.93	1.92
200	600	m = 600	17.14	18.3	18.9	17.25	0	m = 600	36.58	43.38	41.74	37.39	0
	800	m = 800	16.34	17.31	17.96	16.21	0	m = 800	35.13	40.46	39.37	35.41	0
	300	0.95m = 285	139.32	168.79	183.89	154.42	140.26	0.98m = 294	128.86	168.62	175.23	136.7	132.22
300	600	0.95m = 570	120.38	154.83	183.58	146.95	108.21	0.98m = 588	114.54	156.4	162.49	107.12	82.83
	800	0.95m = 760	115.54	152.79	176.46	109.1	88.46	0.98m = 784	112.93	153.4	158.89	110.17	58.97
	300	0.97m = 291	54.70	81.69	73.29	52.24	44.78	0.99m = 297	75.24	104.73	98.47	68.7	52.07
300	600	0.97m = 582	54.63	71.12	73.61	53.71	7.46	0.99m = 594	64.41	91.14	92.65	69.23	13.09
	800	0.97m = 776	55.70	67.36	69.39	55.21	1.36	0.99m = 792	36.95	86.64	88.82	68.98	3.38
	300	m = 300	19.36	21.96	22.67	19.07	0	m = 300	41.22	52.04	50.53	42.9	1.52
300	600	m = 600	17.11	18.1	18.99	16.94	0	m = 600	36.31	41.72	40.47	37.47	0
	800	m = 800	16.53	17.23	17.88	15.96	0	m = 800	34.81	38.96	38.54	35.67	0
	300	0.95m = 285	136.08	178.66	205.19	161.45	148.98	0.98m = 294	124.84	179.92	187.53	137.98	136.51
400	600	0.95m = 570	116.12	158.4	193.4	113.63	101.74	0.98m = 588	109.96	164.88	169.39	102.85	73.46
	800	0.95m = 760	114.30	153.3	170.05	106.31	76.54	0.98m = 784	102.02	160.19	160.61	107.28	49.39
	300	0.97m = 291	51.95	83.08	69.02	50.87	44.34	0.99m = 297	75.17	108.36	101.49	67.41	48.70
400	600	0.97m = 582	54.73	69.15	74.31	53.36	5.2	0.99m = 594	66.71	93.42	95.22	67.97	7.77
	800	0.97m = 776	55.94	65.87	70.69	55	0	0.99m = 792	61.43	88.03	90.27	67.6	1.11
	300	m = 300	19.17	21.86	22.74	18.72	0	m = 300	41.49	52.53	51.37	42.75	1.41
400	600	m = 600	17.24	18.26	18.98	16.54	0	m = 600	36.6	41.84	60.66	36.96	0
	800	m = 800	16.64	17.23	17.92	15.98	0	m = 800	33.68	39.15	38.46	34.39	0
AVERAGE			64.96	77.82	81.72	65.91	43.44		73.83	93.24	94.74	76.02	41.34



(a) All problem instances

(b) Problem instances with $|\Sigma| = 4$

(c) Problem instances with $|\Sigma| = 12$

(d) Problem instances with $|\Sigma| = 20$

Fig. 2 Critical difference (CD) plots. The four plots show the results for different (sub-)sets of problem instances

the graphical representation. All tests and plots were created using the R scmamp package by Calvo and Santafé (2016), accessible at https://github.com/b0rxa/scmamp.

The CD plots substantiate LEARN_CMSA's dominance over competing algorithms across all datasets, with $\text{ACO}^{+}_{\text{neg}}$ trailing as a significant secondary. CMSA and CPLEX are closely matched, yet CMSA displays superiority in a comprehensive assessment of instances (as depicted in Fig. 2a). It is important to note that the differences between CMSA and CPLEX is greatest in the context of small-size alphabets ($|\Sigma| = 4$; see Fig. 2b), reduces considerably with $|\Sigma| = 12$ (Fig. 2c), and disappears for $|\Sigma| = 20$ (Fig. 2d). Note that this suggests that, with an increasing alphabet size, the FFMSP seems to become easier to solve. The graphical analysis also confirms the non-competitiveness of the standalone BA variant.

5 Conclusions and Future Work

In this paper, we have proposed a hybrid variant of the CMSA algorithm, based on a combination with a population-based metaheuristic (bacterial algorithm, BA) whose population provides, at each iteration of CMSA, the solutions that are merged into the current sub-instance of CMSA. In turn, the solution provided by the ILP solver when solving the current sub-instance of CMSA is fed back into the population of the BA. Therefore, it can be said that the proposed LEARN_CMSA algorithm is based on two memory mechanisms: (1) the sub-instance of CMSA and (2) the population of BA. Both algorithms—CMSA and BA—employ a mutual interaction to improve each other's search process. The proposed algorithm is applied to the so-called far from most string problem, an NP-hard problem from the field of bioinformatics. The obtained results show, first, that LEARN_CMSA performs significantly better than its two algorithm components (CMSA and BA). Second, our results show that LEARN_CMSA is a new state-of-the-art approach for solving the far from most strings problem.

However, our new LEARN_CMSA approach certainly also has limitations. The first limitation concerns an elevated number of algorithm parameters, which results from this new algorithm being designed as a combination of two standalone approaches, each one coming with its own set of parameters. A second possible limitation is that the sub-ordinate optimization technique which is used within LEARN_CMSA—BA, in the case of this paper—must work harmonically with the outer CMSA approach. When an optimization problem different to the far from most string problem is considered, BA might not work well for this purpose, and finding a suitable technique might not be straightforward in all cases.

In future work, we plan to apply the same mechanism to other NP-hard combinatorial optimization problems to show the universal use of the proposed technique. Moreover, we plan to replace the BA mechanism by other population-based metaheuristics to show the generality of our algorithmic proposal.

Acknowledgements We acknowledge administrative and technical support by the Spanish National Research Council (CSIC, Spain) and by the University of Concepción (Chile).

Author Contributions Methodology, P.P-D., C.B., R.C., and M.A.P.; programming, P.P-D.; writing—original draft, P.P-D., C.B., and R.C.; writing—review and editing, C.B. and M.A.P.; data curation, P.P-D.; supervision, C.B. and R.C.; validation, M.A.P. All authors have read and agreed to the published version of the manuscript.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. P. Pinacho-Davidson acknowledges financial support from FONDECYT through grant number 11230359. C. Blum was supported by grants TED2021-129319B-I00 and PID2022-136787NB-I00 funded by MCIN/AEI/10.13039/501100011033.

Data Availability For detailed results, interested parties can request information from the corresponding author. The same applies to all the problem instances utilized in this study.

Declarations

Conflict of interest The authors declare no conflicts of interest. The funders played no part in the study's design, data collection, analysis, interpretation, manuscript writing, or decision to publish the results.

References

1. Blum, C., Raidl, G.R.: Hybrid Metaheuristics. Powerful Tools for Optimization. Springer, Switzerland (2016)
2. Raidl, G.R., Puchinger, J., Blum, C.: In: Gendreau, M., Potvin, J.-Y. (eds.) Metaheuristic Hybrids. Springer, Cham (2019)
3. Boschetti, M.A., Maniezzo, V., Roffilli, M., Bolufé Röhler, A.: Matheuristics: Optimization, simulation and control. In: Blesa, M.J., Blum, C., Di Gaspero, L., Roli, A., Sampels, M., Schaerf, A. (eds.) Proceedings of HM 2009 – 6th International Workshop on Hybrid Metaheuristics. Lecture Notes in Computer Science, vol. 5818, pp. 171–177. Springer, Berlin, Heidelberg (2009)
4. Blum, C., Pinacho, P., López-Ibánez, M., Lozano, J.A.: Construct, merge, solve & adapt: A new general algorithm for combinatorial optimization. Comput. Oper. Res. **68**, 75–88 (2016)
5. Blum, C.: Construct, Merge, Solve and Adapt: Application to unbalanced minimum common string partition. In: Blesa, M.J., Blum, C., Cangelosi, A., Cutello, V., Di Nuovo, A., Pavone, M., Talbi, E.-G. (eds.) Proceedings of HM 2016 – 10th International Workshop on Hybrid Metaheuristics. Lecture Notes in Computer Science, vol. 9668, pp. 17–31. Springer, Cham (2016)
6. Thiruvady, D., Blum, C., Ernst, A.T.: Maximising the net present value of project schedules using CMSA and parallel ACO. In: Blesa, M.J., Blum, C., Gambini Santos, H., Pinacho-Davidson, P., Campo, J. (eds.) Proceedings of HM 2019 – 11th International Workshop on Hybrid Metaheuristics. Lecture Notes in Computer Science, vol. 11299, pp. 16–30. Springer, Cham (2019)
7. Lewis, R., Thiruvady, D., Morgan, K.: Finding happiness: An analysis of the maximum happy vertices problem. Comput. Oper. Res. **103**, 265–276 (2019)
8. Ferrer, J., Chicano, F., Ortega-Toro, J.A.: CMSA algorithm for solving the prioritized pairwise test data generation problem in software product lines. J. Heuristics **27**, 229–249 (2021)
9. Mousavi, S.R.: A hybridization of constructive beam search with local search for far from most strings problem. Int. J. Comput. Inform. Eng. **4**(8), 1200–1208 (2010)
10. Blum, C., Festa, P.: A hybrid ant colony optimization algorithm for the far from most string problem. In: Proceedings of EvoCOP 2014 – European Conference on Evolutionary Computation in Combinatorial Optimization, pp. 1–12. Springer, Berlin, Heidelberg (2014)
11. Odonkor, S.T., Addo, K.K.: Bacteria resistance to antibiotics: Recent trends and challenges. Int. J. Biol. Med. Res. **2**(4), 1204–1210 (2011)
12. Ricardo Contreras, A., Valentina Hernández, P., Pinacho-Davidson, P., Angélica Pinninghoff J., M.: A bacteria-based metaheuristic as a tool for group formation. In: Proceedings of IWINAC 2022 – 9th International Work-Conference on the Interplay Between Natural and Artificial Computation. Lecture Notes in Computer Science, vol. 13259, pp. 443–451. Springer, Berlin, Heidelberg (2022)
13. Lanctot, J., Li, M., Ma, B., Wang, S., Zhang, L.: Distinguishing string selection problems. Inform. Comput. **185**(1), 41–55 (2003)
14. Meneses, C.N., Oliveira, C.A., Pardalos, P.M.: Optimization techniques for string selection and comparison problems in genomics. IEEE Eng. Med. Biol. Magaz. **24**(3), 81–87 (2005)
15. Festa, P.: On some optimization problems in mulecolar biology. Math. Biosci. **207**(2), 219–234 (2007)
16. Festa, P., Pardalos, P.M.: Efficient solutions for the far from most string problem. Ann. Oper. Res. **196**(1), 663–682 (2012)
17. Mousavi, S.R., Babaie, M., Montazerian, M.: An improved heuristic for the far from most strings problem. J. Heuristics **18**, 239–262 (2012)
18. Ferone, D., Festa, P., Resende, M.G.C.: Hybrid metaheuristics for the far from most string problem. In: Blesa, M.J., Blum, C., Festa, P., Roli, A., Sampels, M. (eds.) 8th International Workshop on Hybrid Metaheuristics. Lecture Notes in Computer Science, vol. 7919, pp. 174–188. Springer, Berlin, Heidelberg (2013)
19. Ferone, D., Festa, P., Resende, M.G.C.: Hybridizations of grasp with path relinking for the far from most string problem. Int. Trans. Oper. Res. **23**(3), 481–506 (2016)
20. Gallardo, J.E., Cotta, C.: A GRASP-based memetic algorithm with path relinking for the far from most string problem. Eng. Appl. Artif. Intell. **41**, 183–194 (2015)
21. Blum, C., Pinacho-Davidson, P.: Application of negative learning ant colony optimization to the far from most string problem. In: Proceedings of EvoCOP – European Conference on Evolutionary Computation in Combinatorial Optimization. Lecture Notes in Computer Science, vol. 13987, pp. 82–97. Springer, Cham (2023)
22. Pinninghoff J, M.A., Orellana M, J., Contreras A, R.: Bacterial resistance algorithm. an application to CVRP. In: Proceedings of IWINAC 2019 – 8th International Work-Conference on the Interplay Between Natural and Artificial Computation. Lecture Notes in Computer Science, vol. 11487, pp. 204–211. Springer, Cham (2019)
23. López-Ibáñez, M., Dubois-Lacoste, J., Cáceres, L.P., Birattari, M., Stützle, T.: The irace package: Iterated racing for automatic algorithm configuration. Oper. Res. Perspectiv. **3**, 43–58 (2016)
24. Blum, C., Pinacho-Davidson, P.: Application of negative learning ant colony optimization to the far from most string problem. In: Pérez Cáceres, L., Stützle, T. (eds.) Evolutionary Computation in Combinatorial Optimization, pp. 82–97. Springer, Cham (2023)
25. García, S., Herrera, F.: An extension on "statistical comparisons of classifiers over multiple data sets" for all pairwise comparisons. J. Mach. Learn. Res. **9**, 2677–2694 (2008)