AIMS Mathematics, 10(7): 15539–15587.
DOI:10.3934/math.2025697
Received: 25 February 2025
Revised: 28 May 2025
Accepted: 04 June 2025
Published: 08 July 2025

https://www.aimspress.com/journal/Math

Research article

Computational complexity of swarm-based algorithms: a detailed analysis

María-Luisa Pérez-Delgado[1,2,*] **and Jesús-Ángel Román-Gallego**[1,2]

[1] Department of Computer Science and Automatics, Universidad de Salamanca, Escuela Politécnica Superior de Zamora, Av. Requejo, 33, Zamora 49022, Spain

[2] CIMET Research Group (https://cimet.usal.es)

* **Correspondence:** Email: mlperez@usal.es; Tel: +34980545000.

Abstract: In recent years, swarm-based algorithms have been applied to numerous optimization problems. These algorithms use a set or population of solutions that are updated in an iterative process to obtain an approximate solution to the problem. Many articles use these methods to solve complex problems, but do not include information about how time-consuming the methods are. On the other hand, the literature on swarm-based algorithms does not usually include the analysis of computational complexity of the algorithms. The structure of these algorithms makes them time-consuming, so it is essential to know that cost to assess whether it is appropriate to apply them. This article aims to fill the gap by showing a detailed analysis of the computational complexity of a set of 10 popular swarm-based algorithms (particle swarm optimization, shuffled-frog leaping algorithm, artificial bee colony, firefly algorithm, gravitational search, cuckoo search, bat algorithm, grey wolf optimization, chicken swarm optimization, and whale optimization). The operations associated with each method are described using a homogeneous notation, and then the computational complexity is analyzed. Furthermore, the methods are applied to 20 problems, and statistical tests are performed on the results. Although the algorithms have a common basic structure, it is observed that the computational cost is not the same for all of them. Furthermore, the algorithms that consume the most time are not always the ones that generate the best results, so it is advisable to take this information into account before choosing a specific algorithm to solve a complex problem.

Keywords: swarm-based method; computational complexity; optimization; heuristic
Mathematics Subject Classification: 68-04, 68Q17, 68Q25, 68T20

1. Introduction

Let us consider an optimization problem with a solution space of dimension r and an objective function $f(x)$ associated with it. A solution method should find the value $x* = (x_1, ..., x_r)$ that obtains

the best value for the objective function. That is, the solution to a maximization problem is the value $x*$ that produces the largest $f(x*)$ value; conversely, for a minimization problem, the solution is the value of $x*$ that yields the smallest $f(x*)$ value. As the complexity of the optimization problem or the size of the solution space increases, the optimization problem becomes more difficult to solve.

Swarm-based methods are suitable for solving problems that are too complex to be solved by traditional algorithms. They are widely used to solve NP-complete optimization problems [1–3]. These algorithms are inspired by the collective behavior observed in some animals, so that each individual independently can only perform very simple tasks, but when working in a group, they can solve more complex problems.

In general, a swarm-based algorithm considers a population of individuals that represent feasible solutions to a problem. Initially, each individual is usually associated with a random solution from the problem solution space. The best solution from this initial set is taken as the best initial global solution found by the swarm. The algorithm applies an iterative process to try to improve the solution represented by the individuals. If at the end of an iteration a solution is found that improves the global solution found so far, this value is updated. When the iterative process concludes, the solution to the problem is the best of the solutions that have been found throughout the iterations.

Although all swarm algorithms share a common basic operation scheme, they differ both in the operations that update the feasible solutions and in the algorithm parameters. Each swarm algorithm uses specific equations to update the solution associated with each individual. Some equations apply directly to the current solution set, but others require additional calculations. On the other hand, each method uses a different number of parameters. Although values have been proposed for the parameters that generate good results for many problems, it is advisable to adjust these parameters for the problem of interest, which will probably improve the results. Obviously, the more parameters a method includes, the more effort the parameter-tuning operation requires.

Numerous swarm-based algorithms have been proposed in recent years [4, 5]. They have been applied in different domains and problems, including cluster analysis, parameter optimization, data mining, image processing, signal processing, and scheduling problems. The reader can consult [5] for general applications of swarm algorithms, and also other articles with applications of specific algorithms that mimicking bats [6], particles [7], cuckoos [8], frogs [9], and wolves [10].

Heuristic methods are useful for solving complex problems. However, it is important to take into account the computational cost of the applied method to assess its usefulness. Obviously, the method that can generate a good solution with the least time consumption is more appropriate. In many cases the literature describing the application of approximate methods to solve complex problems does not include information on the time taken to obtain the solution. This also occurs in the literature related to swarm-based algorithms. However, when comparing various solution methods, it is important not only to compare the results obtained but also the time used to obtain them.

The goal of this article is to compare the time consumption of several popular swarm-based methods. On the one hand, the computational complexity of each method is analyzed. On the other hand, the algorithms are applied to a set of 20 test problems in order to compare both the execution time and the quality of the generated solution. The comparison of the results will allow us to verify if there are significant differences among the methods.

To analyze the algorithms, the article begins by describing the fundamental operations of all of them. This description uses a notation that allows the identification of operations common to all methods and

simplifies the analysis. A detailed description of the equations used and the associated variables is also included. The goal is to help other researchers understand and use the algorithms, which is difficult when reading articles that present some algorithms due to the general description of some operations.

This article analyzes 10 swarm-based methods: particle swarm optimization (PSO) [11], shuffled-frog leaping algorithm (SFLA) [12], artificial bee colony (ABC) [13, 14], firefly algorithm (FA) [15], gravitational search algorithm (GSA) [16], cuckoo search (CS) [17], bat algorithm (BA) [18], grey wolf optimization (GWO) [19], chicken swarm optimization (CHO) [20], and whale optimization algorithm (WOA) [21]. The set of selected methods includes some of the most popular swarm-based methods, along with others that have been proposed more recently.

The rest of this article is organized as follows. First, Section 2 describes the selected swarm-based methods. Next, Section 3 analyzes the computational complexity of each method. Then, Section 4 includes the experimental results. Section 5 shows a brief reflection on the trade-offs between speed and accuracy in swarm-based algorithms. Finally, Section 6 contains the conclusions of the article.

2. Swarm-based algorithms

There are several key elements when working with a swarm-based method: the objective function of the problem to be solved, the size of the search space, the population, the fitness function, and the solution found by the swarm.

These algorithms attempt to solve an optimization problem that has an objective function associated with it. The function must be maximized or minimized, depending on the problem selected. Let $f(x)$ denote the objective function that is computed for a value x, where x is a vector of r components, $x = (x_1, ..., x_r)$, which represents a feasible solution to the problem, and r is the dimension of the search space. The method applied to solve the optimization problem tries to find the best of the feasible solutions, that is, the one that maximizes (or minimizes) the value of the objective function.

Swarm-based methods use a population of size N that represents a set of feasible solutions. The individuals in the population evolve over time to try to find increasingly better solutions. In general, these algorithms perform a predetermined number of iterations, T. The position of the individual i represents the feasible solution $x_i = (x_{i1}, ..., x_{ir})$, with $i = 1, ..., N$. Since the position of the individuals evolves over the iterations, $x_i(t)$ denotes the position of individual i at iteration t, and $x_i(0)$ denotes the initial position considered before applying the iterative process.

In general, the components of the solution vectors of a problem can only take values in a range. Let lb and ub denote the vectors of size r that determine the minimum and maximum values, respectively, of the components of a position vector. When an algorithm applies some operation to update the current solution associated with an individual, the value of each component of the new solution must fit within the valid range (that is, if the value of component j is greater than ub_j, it is replaced by that maximum; on the contrary, if it is smaller than lb_j, it is replaced by said minimum). The adjustment operation is not explicitly included in the pseudocode of the algorithms described below in order to simplify the description. However, it must be taken into account that it is a necessary operation after calculating a new position. It should be noted that when the set of initial solutions of an algorithm is defined, random values are usually taken in the valid interval, so it is not necessary to apply the aforementioned adjustment.

Swarm-based methods use the objective function to determine the quality or fitness of the solution

associated with an individual. Let $fitness(x)$ denote the function that computes the quality of a feasible solution x. In some cases the fitness is calculated by directly applying the objective function ($fitness = f$), but in others a more general calculation is used that includes the objective function ($fitness \approx f$).

The solution with the best fitness of the set of solutions associated with the individuals during the iterative process defines the solution to the problem found by the swarm algorithm. Let $gbest(t)$ denote the global best solution found by the swarm until iteration t and fit_{gb} the fitness of this solution. The value $gbest(0)$ is the first one associated with this variable. As iterations of the algorithm are carried out, it is checked if a better solution than the one stored in this variable has been found. If in the iteration t of the algorithm the fitness of the solution associated with any of the individuals in the population is better than fit_{gb}, the position and fitness of said individual are used to define the new values of the variables $gbest(t)$ and fit_{gb}, respectively. When the algorithm concludes, the solution to the problem defined by the swarm is the one stored as the global best solution.

Calculating the objective function associated with some problems can be time-consuming. Therefore, it may be useful to store the fitness of each individual, which avoids repeating the calculation of this value each time it is used in some operation. Let fit denote the vector of N elements, $fit = (fit_1, ..., fit_N)$, that stores the fitness of the current position of each individual, where fit_i is the fitness of the current position of individual i, which is computed as $fit_i = fitness(x_i(t))$.

Table 1 summarizes the notation described in this block.

Table 1. Notation.

Variable	Description
$f(x)$	objective function of an optimization problem
$x = (x_1, ..., x_r)$	a feasible solution to the optimization problem
$lb = (lb_1, ..., lb_r)$	the minimum values of the components of a solution vector
$ub = (ub_1, ..., ub_r)$	the maximum values of the components of a solution vector
r	dimension of the search space of the optimization problem
N	size of the population used by a swarm-based method
T	predetermined number of iterations of a swarm-based method
t	swarm algorithm iteration counter, with $t = 1, ..., T$
$x_i = (x_{i1}, ..., x_{ir})$	position of the individual i, with $i = 1, ..., N$
$x_i(t)$	position of individual i at iteration t
$x_i(0)$	initial position of individual i (before applying the iterative process)
$fitness(x)$	function to compute the fitness of the feasible solution x
$gbest(t)$	the best solution found by the swarm until iteration t (vector of size r)
fit_{gb}	fitness of $gbest(t)$, $fit_{gb} = fitness(gbest(t))$
fit_i	fitness of the current position of individual i, $fit_i = fitness(x_i(t))$
$fit = (fit_1, ..., fit_N)$	fitness of the current position of each individual in the swarm

The following subsections describe the operations of various swarm algorithms. Some of the operations are identical. Therefore, if any of said operations has already been described in a section, it is not described again in subsequent sections.

The description of the methods below considers that the objective is to solve a minimization problem. Adapting the operations to solve a maximization problem is very simple.

2.1. Particle swarm optimization algorithm

The PSO algorithm was proposed by Kennedy and Eberhart in 1995 based on the movement of a flock of birds [11]. In this case, a swarm of particles is used to solve a problem, where each particle represents a feasible solution. The particles move in the search space to try to improve the solution they represent.

This algorithm stores three values for each particle i: its current position $x_i(t)$, its current velocity $v_i(t)$, and its personal best position $pbest_i(t)$. The current position is the solution to the problem that the particle represents in the current iteration. The personal best position is the position with the best fitness that the particle has occupied up to the current iteration. Therefore, the global best solution found by the swarm until the current iteration, $gbest(t)$, is the personal best position with the best fitness.

As indicated in the previous section, the current position of each particle must take values in the interval defined by ub and lb. In general, the components of the velocity vector also take values in an interval to limit their effect on updating the position of the particle. Let max_v and min_v denote the vectors of size r that define the upper and lower limits, respectively, for the velocity.

To reduce the number of times a fitness must be calculated, it is advisable to use a vector to store the fitness of the personal best position of each particle. Let $fit_{pb} = (fit_{pb\,1}, ..., fit_{pb\,N})$ denote the vector that stores this information, where $fit_{pb\,i}$ represents the fitness of the current value stored in $pbest_i(t)$.

The basic steps of PSO are defined in Algorithm 1. The initial operations define the initial swarm. These operations select initial values for the position and velocity of each particle ($x_i(0)$, $v_i(0)$, for $i = 1, ..., N$). In general, random values are selected belonging to the predefined interval for position and velocity. However, it is also possible to set the initial velocity equal to zero. Then, the fitness of the particles is computed, $(fit_1, ..., fit_N)$.

Algorithm 1 PSO

1: Define the initial position of the particles
2: Define the initial velocity of the particles
3: Compute the fitness of each particle
4: Define the initial personal best position of the particles and its fitness
5: Select the initial best solution of the swarm, $gbest(0)$, with fitness fit_{gb}
6: **for** t=1 to T **do**
7: Update the velocity of each particle i by Eq (1)
8: Update the position of each particle i by Eq (2)
9: Compute the fitness of each particle
10: Update the personal best position of each particle i, $pbest_i(t)$
11: Update the best solution of the swarm, $gbest(t)$, with fitness fit_{gb}
12: **end for**

After this, the initial position of each particle is stored as the initial personal best position of the particle ($pbest_i(0) = x_i(0)$, for $i = 1, ..., N$), and the fitness of the current position of the particles is copied to define the initial value of fit_{pb} ($fit_{pb\,i} = fit_i$, for $i = 1, ..., N$). This information is used to select the initial global best solution, $gbest(0)$. The particle k whose personal best solution has the minimum fitness ($fit_{pb\,k}$) determines the value stored as the initial global best solution: $gbest(0) =$

$pbest_k(0)$, and its fitness: $fit_{gb} = fit_{pb\,k}$.

An iterative process is then applied to update the current set of solutions associated with the particles. Each iteration updates the velocity and position of each particle i by Eqs (1) and (2), respectively. The product $\otimes$ represents the Hadamard operation. The parameters ω, c_1, and c_2 used in Eq (1) are nonnegative real values that determine the relative influence of each term, while ϵ_1 and ϵ_2 are two vectors with r random values in $[0, 1]$. The parameter c_1 determines the effect of the previous experience of the particle on its movement, while c_2 determines the effect of the previous experience of the swarm.

$$v_i(t) = \omega v_i(t-1) + c_1\epsilon_1 \otimes (pbest_i(t-1) - x_i(t-1)) + c_2\epsilon_2 \otimes (gbest(t-1) - x_i(t-1)). \tag{1}$$

$$x_i(t) = x_i(t-1) + v_i(t). \tag{2}$$

It can be observed that the movement of each particle is influenced both by its previous personal best position and by the best position found so far by the set of particles in the swarm.

It should be noted that after applying Eq (1) to update the velocity of a particle, it is necessary to adjust the components of the vector to the valid interval defined by max_v and min_v (similarly as described for adjusting positions). After making this adjustment, Eq (2) can be applied to update the position of the particle. After this second calculation, it is necessary to adjust the position vector to the interval defined by ub and lb.

The following operation calculates the fitness of the new positions associated with the particles. Based on this information, the best local and global positions are updated if necessary. If the new position of a particle improves its current local best position, this value is updated. That is, if the fitness of $x_i(t)$ is better than the fitness of $pbest_i(t-1)$, ($fit_i < fit_{pb\,i}$), then $pbest_i(t) = x_i(t)$ and $fit_{pb\,i} = fit_i$. It should be noted that if in the current iteration the particle i has not found a better position, its local best position is not modified ($pbest_i(t)$ is the same as $pbest_i(t-1)$).

Finally, if the local best position of a particle improves the current global best position, this value is also updated, and the associated fitness is stored in fit_{gb}.

2.2. *Shuffled-frog leaping algorithm*

This algorithm was proposed by Eusuff and Lansey, based on the behavior of a group of frogs that search for the place that has the maximum amount of food [12].

This method uses a set of N frogs whose positions represent solutions to a problem. An iterative process is applied to improve the solution obtained by the swarm. To do this, in each iteration the frogs of the population are separated into several subsets, called memeplexes, and an iterative process is applied that tries to improve the position of the worst frog in each memeplex. Next, the frogs of all the memeplexes are grouped into a single population, and the same process is repeated until completing T iterations. A peculiarity of this algorithm is that it tries to improve the worst frog instead of considering the best one to apply said improvement. Algorithm 2 shows the main steps of this method.

Algorithm 2 SFLA

Define the initial position of the frogs
Compute the fitness of each frog
Sort the frogs by increasing fitness
Set the initial best solution of the swarm (Set $gbest(0) = x_1(0)$ and $fit_{gb} = fit_1$)
for t=1 to T **do**
  Define m memeplexes
  for each memeplex **do**
    for $j = 1$ to J_{max} **do**
      Identify the best (b) and worst (w) frogs in the memeplex
      Compute x' by Eqs (3) and (4)
      Compute the fitness of x', $fitness_{x'}$
      if ($fitness_{x'} < fit_w$) **then**
        $x_w = x'$, $fit_w = fitness_{x'}$
      else
        Compute x' by Eqs (3) and (5)
        Compute the fitness of x', $fitness_{x'}$
        if ($fitness_{x'} < fit_w$) **then**
          $x_w = x'$, $fit_w = fitness_{x'}$
        else
          Assign random values to x_w and compute fit_w
        end if
      end if
    end for
  end for
  Sort the frogs by increasing fitness
  Update the best solution of the swarm, $gbest(t)$, with fitness fit_{gb}
end for

In the same way as in other swarm-based algorithms, the individuals are initially placed at random positions in the solution space. Then, the fitness of all these positions is computed. After this, the frogs are sorted by increasing fitness value. The last operation of the initial block defines the initial global best solution, $gbest(0)$, which takes the position of the first frog in the sorted list, $x_1(0)$, since fit_1 is the minimum value in the vector fit.

Each iteration of the algorithm starts by defining m memeplexes. To do this, the sorted list of frogs is considered, and successive frogs from that list are assigned to successive memeplexes. The frog i from the list is assigned to the memeplex $[1 + ((i-1)\ MOD\ m)]$, where MOD represents the remainder of the integer division.

After defining the memeplexes, each one is processed independently. This processing consists of trying to improve the worst frog in the memeplex. The first operation identifies the worst and the best frogs in the memeplex, whose current positions are denoted x_w and x_b, respectively, for simplicity. The next operation computes a candidate position, x', for the worst frog in the memeplex by applying Eq (3), where D is computed by Eq (4) and ϵ_1 is a random value between 0 and 1. Before using the

value of D in Eq (3), it fits the range $[-D_{max}, D_{max}]$, where D_{max} is a parameter that determines the maximum amount of change allowed in the position of a frog.

$$x' = x_w + D. \tag{3}$$

$$D = \epsilon_1(x_b - x_w). \tag{4}$$

If the candidate position is better than the current worst frog position (if the fitness of the candidate position, denoted $fitness_{x'}$, is better than fit_w), x' becomes the new worst frog position. Otherwise, a new candidate position x' is calculated using Eqs (3) and (5), where ϵ_2 is a random value between 0 and 1 and D is set to the range $[-D_{max}, D_{max}]$ as described in the previous case.

$$D = \epsilon_2(gbest(t-1) - x_w). \tag{5}$$

If the second candidate position is better than the current worst frog position, x' becomes the new worst frog position; otherwise, a new random position is assigned to the worst frog in the memeplex.

The operations that try to improve the worst frog in a memeplex are applied a fixed number of iterations (J_{max}). Once all memeplexes have been processed, all frogs are considered to be part of a single group to perform the remaining operations in the iteration. Before starting a new iteration of the algorithm, it is necessary to reorder the frogs in the population so that $x_1(t)$ is the best frog in the current population. Also, if said frog has a position that improves the value stored in $gbest$, $x_1(t)$ and fit_1 are used to update the variables $gbest$ and fit_{gb}, respectively.

This algorithm does not require recalculating the fitness of each frog in each iteration, since the position of all the frogs is not updated. It is really only necessary to recalculate the fitness of the worst frog that has been modified in each case. To do this, when the value of x_w is updated, the fitness of this new position can also be stored. It should be noted that the fitness of the worst frog in the memeplex is updated every time its position is updated, as the worst frog can be different in each improvement iteration.

2.3. Artificial bee colony algorithm

The ABC algorithm was proposed by Karaboga, inspired by the behavior of honeybees when searching for food [13, 14].

The bees search for flowers to bring their nectar to the nest. The quality of a flower is determined by the amount of nectar it contains. When a bee finds a flower with food, it informs the other bees in the swarm so that they can also exploit that flower. However, the nectar of a flower decreases as it is visited by bees, so the flower loses quality over time. After a certain time, the flower no longer provides food and is abandoned by the bees, which will look for new flowers. There are different types of bees in a hive, which are assigned different tasks. Each employed bee bring to the hive the nectar from the set of flowers that are currently being exploited and informs the onlooker of their findings. The onlooker bees wait in the hive for information from other bees about the flowers they have found and use this information to decide which flower to move to. On the other hand, the scout bees search for new flowers at random.

In the ABC algorithm, the solutions to the problem are represented by flowers or food sources. The fitness of the flower represents the quality or nectar amount of the flower. In this case, each flower i

has an associated variable that indicates the exhaustion of the flower, $limit_i$. The variable $limit_i$ is set to 0 when the flower begins to be exploited by a bee, and its value increases progressively. When said value reaches a preset limit denoted L, the flower is considered to be exhausted and must be replaced by another. In addition, the method considers the three types of bees described above. Algorithm 3 shows the steps of ABC.

Algorithm 3 ABC

```
Define the initial position of the food sources
Set limit_i = 0, for each food source i
Compute the fitness of each food source
Select the initial best solution of the swarm, gbest(0), with fitness fit_gb
for t=1 to T do
    for each employed bee i do
        Compute a candidate source x' by Eq (6)
        Compute the fitness of x', fitness_x'
        if (fitness_x' < fit_i) then
            Set x_i(t) = x', limit_i = 0, fit_i = fitness_x'
        else
            Set limit_i = limit_i + 1
        end if
    end for
    Select a food source x_i for each onlooker bee based on the probability p_i computed by Eq (7)
    for each onlooker bee j do
        Apply to flower i selected for bee j the same operations described for employed bees
    end for
    Replace each food source x_i whose value limit_i > L
    Update the best solution of the swarm, gbest(t), with fitness fit_gb
end for
```

Initially, a random position is assigned to each of the N flowers in the set. In addition, the variable $limit_i$ associated with each flower in the current set is initialized. The initial phase of the algorithm is completed by calculating the fitness of all the flowers and taking the one with the best fitness as the initial solution of the algorithm. Next, the iterative phase begins, in which the operations of the three types of bees are simulated.

First of all, the operations of the employed bees are applied. There is one such bee associated with each of the flowers in the current set. The bee i, which is associated with the flower i, looks for another flower that may have more nectar than the current one. To do this, a position is calculated for that candidate flower, x', applying Eq (6), where k is the index of another flower in the current set other than i and ϵ_i is a vector of random values in $[-1, 1]$.

$$x' = x_i(t-1) + \epsilon_i \otimes (x_i(t-1) - x_k(t-1)). \tag{6}$$

If the fitness of the candidate flower is better than the fitness of the flower i, the bee leaves said flower and goes to the candidate (that is, the position of the candidate flower replaces the position of

the flower i). When the candidate flower replaces the current flower i, the variable $limit_i$ is reset, and the fitness of the new position is also stored; otherwise, the value of $limit_i$ is increased by 1.

The operations of the onlooker bees are then applied. In general, the number of onlooker bees is equal to the number of employed bees. These operations are similar to those of the employed bees, although in this case the onlooker bee j does not have a flower assigned to it but has to choose a flower i from the current set. Therefore, the bee j first chooses a random flower i and then applies the enhancement operations to the flower i. The selection probability of flower i is calculated by Eq (7).

$$p_i = \frac{fit_i}{\sum_{k=1}^{N} fit_k}. \tag{7}$$

Finally, the operations of the scout bees are applied. This consists of checking the $limit_i$ value associated with each flower i of the current set. If said value exceeds the threshold L, the current position of the flower i is replaced with a new random position, and the corresponding variable $limit_i$ is set to zero. Additionally, the fitness of the new flower position is calculated.

After applying the operations of the three types of bees, it is checked if any of the flowers in the current set is better than the solution stored as best so far ($gbest$), in which case said solution must be updated.

2.4. Firefly algorithm

The firefly algorithm is based on the flashing patterns of fireflies [15]. Fireflies are attracted to the light emitted by other nearby fireflies. The degree of attraction of one firefly over another depends on the light it emits (its brightness) and the distance that separates both individuals. On the other hand, the intensity of light that an observer perceives from a light source depends on the distance between the observer and the source and also on the light absorption coefficient of the medium.

FA considers several simplifications:

- Fireflies are considered unisex, so a firefly is attracted to others regardless of their sex.
- The attractiveness of a firefly is proportional to its brightness, and both values decrease as distance increases. Therefore, each firefly will move towards the ones brighter than it. Since the brightest firefly is not attracted to any companions, it will move randomly.
- The brightness of a firefly is determined by the objective function of the problem to be solved.

The algorithm considers a set of N fireflies and the position of each firefly represents a feasible solution. In this algorithm the term "brightness" is generally used instead of "fitness". When considering a minimization problem, the brightest firefly is the one with the lowest fitness. The brighter a firefly is, the better the solution it represents. Algorithm 4 shows the main operations of FA.

The first operation sets random initial positions for the fireflies in the swarm. Then, the fitness of the fireflies is computed. This information is used to sort the fireflies by decreasing brightness. The first firefly in this sorted list is the best firefly in the initial swarm, so its position ($x_1(0)$) defines the initial value for the global solution of the algorithm, $gbest(0)$.

The iterative process of this algorithm moves the fireflies towards brighter individuals. This allows fireflies that represent worse solutions to move towards more promising areas of the solution space. This algorithm uses two different equations to update the position of the fireflies: one of them is applied to the brightest firefly, and the other is applied to the rest of the fireflies.

Algorithm 4 FA

Define the initial position of the fireflies
Compute the fitness of each firefly
Sort the fireflies by decreasing brightness
Set the initial best solution of the swarm (Set $gbest(0) = x_1(0)$ and $fit_{gb} = fit_1$)
for t=1 to T **do**
  for $i = N$ to 2 **do**
    Update $x_i(t)$ by Eq (9)
  end for
  Update $x_1(t)$ by Eq (8)
  Compute the fitness of each firefly
  Sort the fireflies by decreasing brightness
  Update the best solution of the swarm, $gbest(t)$, with fitness fit_{gb}
end for

As previously indicated, the firefly 1 in the sorted list is the brightest. Since it is not attracted to any other firefly, it moves randomly around its current position, applying Eq (8), where ϵ_1 is a vector with r random values in the interval $[-0.5, 0.5]$ and $\alpha \in [0, 1]$ is the randomization parameter of the algorithm.

$$x_1(t) = x_1(t-1) + \alpha\,\epsilon_1. \tag{8}$$

To update firefly i in the sorted list, with $i = 2, 3, ..., N$, it must be taken into account that the fireflies that are before it in the list are brighter, so they attract it. However, the individuals that come after i in the list are less bright, so they do not attract firefly i. Therefore, the equation that updates the position of this firefly (Eq (9)) includes a summation with the contribution of each firefly brighter than i. In this case, ϵ_2 and α have the same characteristics described for Eq (8). On the other hand, $\beta(r_{ij})$ represents the attractiveness of the firefly j, and it can be computed by Eqs (10) or (11), where β_0 is the attractiveness at distance 0, γ is a parameter that represents the light absorption coefficient of the medium ($\gamma \geq 0$), and r_{ij} is the distance between the fireflies i and j. The author of the method proposed both options to compute $\beta(r_{ij})$ since the second can be computed faster [15].

$$x_i(t) = \sum_{j=1}^{i-1} \beta(r_{ij})(x_j(t-1) - x_i(t-1)) + \alpha\,\epsilon_2. \tag{9}$$

$$\beta(r_{ij}) = \beta_0 \exp(-\gamma\, r_{ij}^2). \tag{10}$$

$$\beta(r_{ij}) = \frac{\beta_0}{1 + \gamma r_{ij}^2}. \tag{11}$$

Once all the fireflies have moved, the fitness of the new positions is calculated. The fireflies are then sorted again by brightness. Finally, the global best solution is updated if its current fitness is worse than the fitness of the first firefly in the sorted list.

2.5. Gravitational search algorithm

This algorithm, proposed by Rashedi et al. in 2009, is inspired by the law of gravity and mass interactions [16].

Newton's law of universal gravitation states that each particle in the universe attracts every other particle with a gravitational force that is directly proportional to the product of their masses and inversely proportional to the square of the distance between them. On the other hand, Newton's second law indicates that when a force is applied to a particle, its acceleration depends only on the applied force and the mass of the particle. Equation (12) defines the gravitational force that acts on particle i by particle j, where MP_i is the passive gravitational mass of particle i, MA_j is the active gravitational mass of particle j, R_{ij} is the distance between the particles, and $G(t)$ is the gravitational constant at time t. The gravitational constant is not taken as a fixed value but depends on the actual age of the universe due to the effect of decreasing gravity. Equation (13) defines the acceleration of particle i, where MI_i is the inertial mass of the particle, and F_i is the net force (the sum of the forces) acting on particle i.

$$F_{ij} = G(t)\frac{MP_i MA_j}{R_{ij}^2}. \tag{12}$$

$$a_i = \frac{F_i}{MI_i}. \tag{13}$$

GSA uses a set of N objects that interact according to Newton's laws of gravity and motion. The position of each object represents a solution to the problem. The mass of an object is calculated based on the fitness of its position, and the objects with more mass represent better solutions. The gravitational force causes all objects to attract each other, which causes a movement of all of them. The attraction increases when considering larger and closer objects. The movement brings the objects closer to the object with the greatest mass. The heavy masses move more slowly than lighter ones; this guarantees the exploitation process of the algorithm.

Algorithm 5 GSA

1: Define the initial position of the objects
2: Define the initial velocity of the objects
3: Compute the fitness of each object
4: Select the initial best solution of the swarm, $gbest(0)$, with fitness fit_{gb}
5: **for** t=1 to T **do**
6: Update G by Eq (14)
7: Compute the mass of each object i by Eq (15)
8: Compute the total force that acts on each object i by Eq (17)
9: Compute the acceleration of each object i by Eq (19)
10: Update the velocity of each object i by Eq (20)
11: Update the position of each object i by Eq (21)
12: Compute the fitness of each object
13: Update the best solution of the swarm, $gbest(t)$, with fitness fit_{gb}
14: **end for**

Algorithm 5 shows the main steps of GSA. The object i has a position $x_i(t)$, a velocity $v_i(t)$, an acceleration $a_i(t)$, and a mass $M_i(t)$. Although the authors of the method considered three masses for each object (active mass, passive mass, and inertial mass), they proposed giving them all the same value [16]. Therefore, to simplify the description of the algorithm, in this section it is considered that each object has a single mass associated with it.

Initially, each object is assigned a random position within the solution space, and the components of the velocity vectors are given a value of zero. The fitness of each object is then calculated, and the position of the object with the best fitness is taken as the best initial solution ($gbest(0)$).

An iterative process is then carried out that moves all objects in the swarm. Each iteration begins by calculating a new value for the gravitational constant by Eq (14), where $G0$ is the initial value considered for the gravitational constant and α is a parameter.

$$G(t) = G_0 \exp(-\alpha\, t/T). \tag{14}$$

The mass of all the objects is then calculated. Equation (15) generates the mass of the object i, where the value $m_i(t)$ is calculated by Eq (16). In this equation, fit_w and fit_b represent the worst and best fitness, respectively, of the current set of objects.

$$M_i(t) = \frac{m_i(t)}{\sum_{j=1}^{N} m_j(t)}. \tag{15}$$

$$m_i(t) = \frac{fit_i - fit_w}{fit_b - fit_w}. \tag{16}$$

Once the mass of each object has been calculated, the forces acting on the objects can be calculated.

The total force that acts on object i, $F_i(t)$, is a randomly weighted sum of all the forces acting on the object. This value is computed by Eq (17), where ϵ_i is a vector of size r that contains random numbers in the interval [0, 1], and $F_{ij}(t)$ is the force acting on object i from object j at time t.

$$F_i(t) = \sum_{j=1, j\neq i}^{N} \epsilon_j \otimes F_{ij}(t). \tag{17}$$

$F_{ij}(t)$ is computed by Eq (18), where ε is a small constant introduced to avoid division by zero, and $R_{ij}(t)$ is the Euclidean distance between objects i and j. Some variants of the algorithm use the value of $R_{ij}(t)$ raised to a power p, which is a parameter of the algorithm; however, the initial version of the algorithm does not use this option.

$$F_{ij}(t) = G(t)\frac{M_i(t)M_j(t)}{R_{ij}(t) + \varepsilon}(x_j(t) - x_i(t)). \tag{18}$$

Once the mass and total force have been calculated, the acceleration of each object i is calculated by Eq (19),

$$a_i(t) = \frac{F_i(t)}{M_i(t)}. \tag{19}$$

After this, the velocity of each object i is calculated by Eq (20), where ϵ_i is a vector of size r that contains random values in the interval $[0, 1]$. Finally, the new position of each object is computed by Eq (21).

$$v_i(t) = v_i(t-1) \otimes \epsilon_i + a_i(t). \tag{20}$$

$$x_i(t) = x_i(t-1) + v_i(t). \tag{21}$$

To complete an iteration of the algorithm, the fitness of the new positions of the objects is calculated, and the best solution found so far ($gbest(t)$) is updated if necessary.

2.6. Cuckoo search algorithm

The CS algorithm was proposed by Yang and Deb based on the mechanism used by cuckoos to lay their eggs [17].

Cuckoos lay their eggs in the nests of other birds for them to incubate. There is a chance that the host bird will detect these eggs, in which case it may throw them away or abandon the nest. If the bird does not detect the cuckoo eggs, it continues incubating them until new cuckoos hatch.

The CS algorithm imitates the reproductive behavior of cuckoos, considering the following simplifications:

- Each cuckoo can only lay one egg in the nest it chooses at random.
- The number of nests does not change and is considered constant.
- If an egg has been laid in the nest before and it is of higher quality than the egg laid later, the lineage continues from the high-quality egg.
- The host bird recognizes the egg that does not belong to it with a probability in $[0, 1]$. In that case, the solution associated with that nest is discarded and a new one is defined. The mutation probability value $p_0 \in [0, 1]$ is a parameter of the algorithm used to determine if the host bird recognizes the egg.

Therefore, to solve the problem, the algorithm considers a set of N nests. Each nest contains an egg and represents a solution to the problem. To generate new solutions, this algorithm uses Lévy flights, which provide a random walk. Equation (22) performs a Lévy flight to generate a new solution x_i' for cuckoo i. The parameter $\alpha > 0$ is the step size, whose value depends on the scale of the problem being solved. $Levy(\lambda)$ represents a probability value obtained from a Lévy distribution (Eq (23)), which has infinite variance with infinite mean.

$$x_i' = x_i(t-1) + \alpha \otimes Levy(\lambda). \tag{22}$$

$$Levy(\lambda) \sim u = t^{-\lambda}, (1 < \lambda \leq 3). \tag{23}$$

Algorithm 6 shows the main steps of CS. To define the operations of this algorithm, the Matlab implementation published by the author of the method has been taken into account [22], since the original article includes fewer details.

Algorithm 6 CS

Define the initial population of host nests
Compute the fitness of each nest
Select the initial best solution of the swarm, $gbest(0)$, with fitness fit_{gb}
for t=1 to T **do**
  Compute a candidate solution x'_i for each cuckoo i by Eq (26)
  Compute the fitness of each candidate solution x'_i, $fitness_{x'_i}$
  for i=1 to N **do**
    Select a random nest j
    if $fitness_{x'_i} < fit_j$ **then**
      Set $x_j(t) = x'_i$, $fit_j = fitness_{x'_i}$
    end if
  end for
  Sort the individuals by decreasing fitness
  for i = 1 to w **do**
    if $\varepsilon > pa$ **then**
      Compute a candidate solution x' for nest i by Eq (27)
      Compute the fitness of x', $fitness_{x'}$
      if $fitness_{x'} < fit_i$ **then**
        Set $x_i(t) = x'$, $fit_i = fitness_{x'}$
      end if
    end if
  end for
  Update the best solution of the swarm, $gbest(t)$, with fitness fit_{gb}
end for

The first operation of the algorithm associates random initial positions with the N nests. Once the fitness of each solution is calculated, the best global solution of the algorithm, $gbest(0)$, and its fitness, fit_{gb}, are determined.

An iterative process is then applied that includes three basic operations: updating the set of solutions through Lévy flights, abandoning the worst nests (solutions), and updating the best global solution of the swarm.

First, Lévy flights are applied to try to improve the solutions of the current set. The Matlab implementation of this algorithm available at [22] applies the method proposed by Mantegna to generate random numbers according to a symmetric Lévy stable distribution [23]. The method first computes $u = \sigma\epsilon_1$ and $v = \epsilon_1$, where ϵ_1 and ϵ_2 are random values drawn from a standard normal distribution. The value of σ is computed by Eq (24), where Γ represents the gamma function and β is the scale factor, which is a parameter of the algorithm.

$$\sigma = \left(\frac{\Gamma(1+\beta)\sin(\pi\beta/2)}{\Gamma(\frac{1+\beta}{2})\beta 2^{(\beta-1)/2}} \right)^{1/\beta}. \tag{24}$$

Then, the *stepsize* is computed by Eq (25), and the candidate position for the current cuckoo i is

computed by Eq (26), where ϵ_3 is a random value that fits a standard normal distribution.

$$stepsize = 0.01\left(\frac{u}{|v|}\right)^{1/\beta}(x_i(t-1) - gbest). \tag{25}$$

$$x_i' = x_i(t-1) + stepsize\ \epsilon_3. \tag{26}$$

Once the N candidate solutions have been defined, their fitness is calculated. Next, an iterative process is carried out that compares the candidate solutions with the solutions of the current set to try to improve them. In each iteration a nest j is randomly chosen from the current set. If the candidate solution x_i' is better than the one associated with nest j, the candidate solution replaces the current solution associated with nest j.

The discovery of eggs by host birds is simulated below. To do this, the solutions are sorted by decreasing fitness, which allows the worst solutions to be identified. The worst w solutions are then evaluated to determine if they are discarded and replaced by others. To decide whether to abandon a nest, a random number ε between 0 and 1 is generated. If this number is greater than the parameter pa of the algorithm, a candidate solution is generated for the nest. To do this, two solutions are randomly selected from the current set, which we denote a and b, and Eq (27) is applied, where ϵ_4 is a random value drawn from the uniform distribution in the interval [0, 1]. If the candidate solution is better than the current position of nest i, the candidate position is taken as the new position of the nest.

$$x' = x_i(t) + \epsilon_4(x_a(t) - x_b(t)). \tag{27}$$

Before completing an iteration of the algorithm, it is checked if any of the solutions of the current population is better than the one stored in $gbest$, in which case said solution and its fitness are used to define the new values of $gbest$ and fit_{gb}, respectively.

2.7. Bat algorithm

BA was proposed by Yang in 2010, inspired by the echolocation used by bats [18].

Bats use echolocation to detect prey and avoid obstacles. They emit a very strong pulse of sound and listen for the echo that is generated. They can detect the distance and orientation of the target, the type of prey, and even the speed at which the prey is moving.

The algorithm uses a population of N bats whose positions are elements of the search space. Bats fly around the search space to find prey that represent solutions to the problem. Each bat i has 5 associated variables that evolve over time: position $x_i(t)$, velocity $v_i(t)$, frequency $fr_i(t)$, volume $A_i(t)$, and pulse emission index $r_i(t)$. The frequency takes values in the interval $[f_{min}, f_{max}]$. The volume initially takes a positive value, A_0, and decreases as the iterations of the algorithm progress. The pulse emission index is a value in the interval [0, 1]; it is possible to give an initial value r_0 for all bats or take different values of this interval for each one.

The operations of this method are summarized in Algorithm 7. In this case, the initial operations not only need to assign a random initial position $x_i(0)$ to each bat i but also require initial values for $v_i(0)$, $A_i(0)$, and $r_i(0)$. The initial velocity is usually set to zero. As previously indicated, the parameters A_0 and r_0 are used to define $A_i(0)$ and $r_i(0)$, respectively. After this, the fitness of the positions of all

the bats is calculated and the best of these positions is taken as the initial solution found by the swarm, $g_{best}(0)$.

Algorithm 7 BA

```
Define the initial position of the bats
Define the initial velocity of the bats
Define the initial volume and pulse of the bats
Compute the fitness of each bat
Select the initial best solution of the swarm, gbest(0), with fitness fit_gb
for t=1 to T do
    for each bat i do
        Compute the frequency f_i(t) by Eq (28)
        Update the velocity v_i(t) by Eq (29)
        Compute a candidate position x' by Eq (30)
        if (r_i(t − 1) < ε_1) then
            Select a solution x_j among the best solutions
            Compute a candidate position x' by Eq (31)
        end if
        Compute the fitness of x', fitness_x'
        if (A_i(t − 1) > ε_2) and (fit_i > fitness_x') then
            Set x_i(t) = x', fit_i = fitness_x'
            Update r_i(t) by Eq (32) and A_i(t) by Eq (33)
        end if
    end for
    Update the best solution of the swarm, gbest(t), with fitness fit_gb
end for
```

An iterative process then begins in which new solutions are generated and an attempt is made to improve the position of the bats. To generate new solutions, the frequency and velocity of each bat i are updated by applying Eqs (28) and (29), respectively, where β is a vector of random values in $[0, 1]$ taken from a uniform distribution. Then, a candidate position x' is computed by Eq (30).

$$f_i(t) = f_{min} + (f_{max} - f_{min}) \otimes \beta. \tag{28}$$

$$v_i(t) = v_i(t-1) + (x_i(t-1) - g_{best}(t-1)) \otimes f_i(t). \tag{29}$$

$$x' = x_i(t-1) + v_i(t). \tag{30}$$

The pulse of bat i is then used to determine if a new candidate position should be calculated for that bat. If the pulse of the bat is smaller than a random value $\epsilon_1 \in [0, 1]$, a new candidate position is generated for said bat by Eq (31), where ϵ_3 is a random number in $[-1, 1]$, while M is the average of the $A_i(t-1)$ values of all bats. This candidate position is in the environment of another solution, x_j, chosen

randomly among the best solutions. Although this position can be selected among the best in the current population, it is usual to consider the best global position found by the algorithm ($g_{best}(t-1)$).

$$x' = x_j(t-1) + \epsilon_3 M. \tag{31}$$

If the current volume of the bat exceeds a random value $\epsilon_2 \in [0, 1]$ and also its current position is worse than the candidate position, the position of said bat is updated. In this case, its pulse also increases (Eq (32)) and its volume decreases (Eq (33)). α and γ are constant values such that $0 < \alpha < 1$ and $\gamma > 0$. In the simplest case, the author of the method proposes to make $\alpha = \gamma$ and use $\alpha = \gamma = 0.9$ [18].

$$r_i(t) = r_i(0)(1 - \exp(-\gamma t)). \tag{32}$$

$$A_i(t) = A_i(t-1)\alpha. \tag{33}$$

Before concluding each iteration of the algorithm, the fitness of the current positions of the bats is evaluated to check if a better solution has been found than the one stored in *gbest* so far.

2.8. Grey wolf optimization algorithm

This algorithm was proposed by Mirjalili et al. in 2014, imitating the hunting mechanism of the grey wolf [19]. Grey wolves hunt in groups. There is a clear hierarchy within the group, which determines the role of each individual. The GWO algorithm tries to mimic the leadership hierarchy and hunting method of these wolves.

Algorithm 8 shows the basic operations of GWO. It solves an optimization problem by using a population of N wolves. The position of each wolf represents a solution to the problem, and the three wolves with the best fitness are used to guide the movement of the swarm. To simplify the notation, the three best wolves are denoted α, β and δ, respectively, and their current positions are denoted $x_\alpha(t)$, $x_\beta(t)$, and $x_\delta(t)$.

Algorithm 8 GWO

1: Define the initial position of the wolves
2: Compute the fitness of each wolf
3: Identify the α, β and δ wolves
4: Set the initial best solution of the swarm (Set $gbest(0) = x_\alpha(0)$, with fitness $fit_{gb} = fit_\alpha$)
5: Set $a = a_0$
6: **for** $t = 1$ to T **do**
7: Update the positions of each wolf i by Eq (34)
8: Compute the fitness of each wolf
9: Identify the α, β and δ wolves
10: Update the best solution of the swarm, $gbest(t)$, with fitness fit_{gb}
11: Update a by Eq (43)
12: **end for**

The initial operation selects random initial positions for the individuals in the swarm. Then, the fitness of all these positions is computed, and this information is used to identify the best three wolves. The initial position of the α wolf defines the initial value for the global best solution found by this algorithm.

The position of each wolf is updated at each iteration of this algorithm, taking into account the position of the three best wolves. The new position of the wolf i at iteration t, $x_i(t)$, is computed by Eq (34), where the vectors $X1$, $X2$, and $X3$ are computed by Eqs (35), (36), and (37), respectively.

$$x_i(t) = \frac{X1 + X2 + X3}{3}. \tag{34}$$

$$X1 = x_\alpha(t-1) - A_1 \otimes D_\alpha. \tag{35}$$

$$X2 = x_\beta(t-1) - A_2 \otimes D_\beta. \tag{36}$$

$$X3 = x_\delta(t-1) - A_3 \otimes D_\delta. \tag{37}$$

D_α, D_β, and D_δ are computed by Eqs (38), (39), and (40), respectively, where the operator $|\cdot|$ indicates that the absolute value of each component of the resulting vector is taken.

$$D_\alpha = |C_1 \otimes x_\alpha(t-1) - x_i(t-1)|. \tag{38}$$

$$D_\beta = |C_2 \otimes x_\beta(t-1) - x_i(t-1)|. \tag{39}$$

$$D_\delta = |C_3 \otimes x_\delta(t-1) - x_i(t-1)|. \tag{40}$$

A_1, A_2, and A_3 are computed by Eq (41), where a is a parameter of the algorithm.

$$A = 2\, a\, \epsilon_1 - a. \tag{41}$$

C_1, C_2, and C_3 are computed by applying Eq (42).

$$C = 2\epsilon_2. \tag{42}$$

The variables ϵ_1 and ϵ_2 used in Eqs (41) and (42) are vectors whose r components take random values from the interval $[0, 1]$.

The next step of the algorithm computes the fitness of the new positions. The three best wolves in the swarm are identified below. If the position of the α wolf in the current iteration improves the global best solution stored so far, $gbest$ is updated to store the position of the current α wolf, and fit_{gb} stores its fitness.

The last operation of each iteration updates the value of the a parameter. This parameter is initialized before the iterative process starts and is updated over the course of iteration. Mirjalili proposed setting the initial value of a equal to 2 ($a_0 = 2$) and then applying Eq (43) to update the value so as the parameter decreases linearly from 2 to 0 over the course of iterations [19].

$$a = a_0\left(1 - \frac{t}{T}\right). \tag{43}$$

2.9. Chicken swarm optimization algorithm

This algorithm was proposed by Meng et al. in 2014 [20]. It is inspired by the behavior of a group of chickens when eating. The algorithm simulates the search for food, taking into account that there are 3 types of individuals in the group: roosters, hens, and chicks, among whom a hierarchical relationship is established. The algorithm imitates the behavior of each type of individual, taking into account the hierarchical order.

A population of N individuals is considered, the position of each representing a solution to the problem. All individuals search for food in the solution space of the problem to be solved, updating their position over time. There are three subsets of individuals: the roosters are the individuals with the best fitness, the chicks are the individuals with the worst fitness, and the hens are the rest of the individuals. Each of the roosters is the leader of a group. Additionally, each chick has a mother hen.

Algorithm 9 shows the main steps of the method.

Algorithm 9 CHO

Define the initial position of the individuals
Compute the fitness of each individual
Sort the individuals by increasing fitness
Define the type of each individual
Define the relationship between hens and roosters
Define the relationship between chicks and mothers
Set the initial best solution of the swarm (Set $gbest(0) = x_1(0)$ and $fit_{gb} = f_1$)
for t=1 to T **do**
  if ($t \mod G$) = 0 **then**
    Sort the individuals by increasing fitness
    Define the type of each individual
    Define the relationship between hens and roosters
    Define the relationship between chicks and mothers
  end if
  Compute a candidate position x'_i for each individual by Eqs (44), (46) or (49) (based on its type)
  for each individual i **do**
    Compute the fitness of x'_i, $fitness_{x'}$
    if $fitness_{x'} < fit_i$ **then**
      Set $x_i(t) = x'_i$, $fit_i = fitness_{x'}$
    end if
  end for
  Update the best solution of the swarm, $gbest(t)$, with fitness fit_{gb}
end for

As with other swarm algorithms, the first operation consists of defining an initial position for each individual in the swarm, and then the fitness of all positions is calculated. Before starting the iteration stage of the algorithm, the relationship between the different individuals in the group must be evaluated. This requires performing several operations. The individuals are first sorted by fitness. Then, the

best R individuals are taken as roosters, the worst C as chicks, and the remaining H as hens, so that $N = R + C + H$. The next operation is to determine the relationship between hens and roosters. To this end, each hen is assigned a randomly chosen rooster, whereby the hen is associated with the group headed by that rooster. The last operation is to determine the relationship between chicks and mothers. To perform this operation, M of the H hens are randomly chosen as mothers. Then, each chick randomly chooses among said hens and is associated with the group to which its mother belongs.

The type of each individual in the swarm and the relationship among the individuals are periodically updated based on the parameter G.

To complete the initial stage, the solution with the best fitness of the sorted list (the first one) is used to define the best initial solution found by the algorithm.

The iterative process updates the positions of the individuals, taking into account the type of each individual and the relationships that exist among the individuals. To update the position of the individual i, a new candidate position x_i' is calculated. If the fitness of said position is better than the fitness of the current position of the individual, it moves to the new position.

The candidate position for the rooster i is computed by Eq (44). In this case, ϵ_1 is a vector of size r that includes random values of a Gaussian distribution with mean 0 and standard deviation σ^2, computed by Eq (45), where k is a random rooster from the group of roosters (with $k \neq i$) and ε is a very small constant (to avoid a division by zero error).

$$x_i' = x_i(t-1) \otimes (1 + \epsilon_1). \tag{44}$$

$$\sigma^2 = \begin{cases} 1 & if\ fit_i < fit_k \\ \exp\left(\frac{fit_k - fit_i}{|fit_i| + \varepsilon}\right) & otherwise \end{cases}. \tag{45}$$

The candidate position for the hen i is computed by Eq (46), where ϵ_2 and ϵ_3 are vectors with random values of a uniform distribution in the interval $[0, 1]$, $r1$ is the index of the rooster of the group to which the hen belongs, and $r2$ is the index of a rooster or a hen (other than $r1$) taken at random. The values of s_1 and s_2 are computed by Eqs (47) and (48), respectively, where ε is a very small constant (to avoid a division by zero error).

$$x_i' = x_i(t-1) + s_1\epsilon_2 \otimes (x_{r1}(t-1) - x_i(t-1)) + s_2\epsilon_3 \otimes (x_{r2}(t-1) - x_i(t-1)). \tag{46}$$

$$s_1 = \exp\left(\frac{fit_i - fit_{r1}}{|fit_i| + \varepsilon}\right). \tag{47}$$

$$s_2 = \exp(fit_{r2} - fit_i). \tag{48}$$

The candidate position for the chick i is computed by Eq (49), where m denotes the index of the mother hen of the chick and P is a parameter with a value in the interval $(0, 2)$.

$$x_i' = x_i(t-1) + (x_m(t-1) - x_i(t-1))\, P. \tag{49}$$

Once the individuals' update process is completed, the fitness of their new positions is calculated. If any of these positions is better than the global solution stored so far, said solution is updated.

2.10. Whale optimization algorithm

This algorithm, proposed by Mirjalili and Lewis, mimics the hunting behavior of humpback whales [21]. The algorithm imitates three basic behaviors of humpback whales: the movement towards the best individual to pursue the prey, the movement towards random individuals to search for new prey, and the use of a spiral to simulate the bubble net attack mechanism of the whales.

Humpback whales swim around their prey in a shrinking circle along a spiral path. To simulate this behavior, the algorithm assumes that when updating the position of a whale, there is the same probability of choosing between the shrinking encircling mechanism and the spiral model. A random value $p \in [0, 1]$ is used to choose between the two options.

Algorithm 10 defines the main steps of WOA. Initially, all the whales in the swarm are assigned random positions. The fitness of these positions is then calculated, and the one with the lowest fitness is taken to define the best initial solution, $g_{best}(0)$.

Algorithm 10 WOA

```
Define the initial position of the whales
Compute the fitness of each whale
Select the initial best solution of the swarm, gbest(0), with fitness fit_gb
Set a = a_0
for t=1 to T do
    for each whale i do
        if p < 0.5 then
            Update the position of the whale i by Eq (50)
        else
            Update the position of the whale i by Eq (55)
        end if
    end for
    Compute the fitness of each whale
    Update the best solution of the swarm, gbest(t), with fitness fit_gb
    Update a by Eq (43)
end for
```

The algorithm uses the parameter a, similar to the one described in the case of the GWO algorithm, whose value reduces over time and allows a whale's approximation to the best global solution to become increasingly greater. As described by the authors of the method, this parameter can be initialized and updated in the same way described for the GWO algorithm, so it is not repeated here.

The algorithm applies an iterative process that moves all whales in the swarm. Each whale updates its position either with respect to a randomly chosen whale or with respect to the best global solution obtained so far, $g_{best}(t-1)$. To update the position of a whale i, a random value $p \in [0, 1]$ is generated that determines the behavior of the whale to be simulated.

When $p < 0.5$, the whale can apply two different behaviors: one of them brings it closer to the best solution found so far (the whale encircles the current prey), while the other brings it closer to a random solution (the whale moves towards another possible prey). In this case, the 2-norm of A is used to determine the solution considered to update the position of the whale i. If $\|A\|_2 < 1$, the position of the

global best solution obtained by the swarm is considered. On the other hand, if $\|A\|_2 \geq 1$, the position of a random whale, denoted x_{rand}, is used to update the position of whale i.

$$x_i(t) = \begin{cases} gbest(t-1) - A \otimes D & if \ \|A\|_2 < 1 \\ x_{rand} - A \otimes D_{rand} & if \ \|A\|_2 \geq 1 \end{cases}. \tag{50}$$

A, D, D_{rand}, and C are vectors computed by Eqs (51), (52), (53), and (54), respectively. The operator $|\cdot|$ indicates that the absolute value of each component of the resulting vector is taken. Finally, the variables ϵ_1 and ϵ_2 are vectors whose r components take random values from the interval [0, 1].

$$A = 2\ a\ \epsilon_1 - a. \tag{51}$$

$$D = |C \otimes gbest(t-1) - x_i(t-1)|. \tag{52}$$

$$D_{rand} = |C \otimes x_{rand} - x_i(t-1)|. \tag{53}$$

$$C = 2\ \epsilon_2. \tag{54}$$

When $p \geq 0.5$, the spiral motion behavior is simulated. The position of the whale is updated by Eq (55), where D' is a vector computed by Eq (56), b is a constant used to define the shape of the logarithmic spiral (it is usually set to 1), and l is a random number in $[-1, 1]$.

$$x_i(t) = D' e^{bl} cos(2\pi l) + gbest(t-1). \tag{55}$$

$$D' = |gbest(t-1) - x_i(t-1)|. \tag{56}$$

Once all the whales have been processed, the fitness of their new positions is calculated. This information is used to check whether the best solution stored so far should be updated. The value of a is updated by applying Eq (43) before concluding the current iteration of the algorithm.

3. Computational complexity

The time complexity of each algorithm can be defined by analyzing the complexity of its operations.

Since there is a set of operations that are common to all the algorithms, they are analyzed in the initial subsection before going on to analyze each algorithm independently.

3.1. Operations common to all methods

Table 2 summarizes the complexity of the operations discussed in this section. The common operations are as follows:

- Define the initial position of the individuals in the swarm (op1). This operation defines r values for the position vector of N individuals. The complexity is $O(rN)$.

- Compute the fitness of a feasible solution (op2). The objective function of the problem to be solved determines the complexity of calculating the fitness of a solution. However, for a problem defined in a space of dimension r, it is generally necessary to evaluate the r components of the solution vector. Therefore, the complexity of this operation is $O(r)$ at least, but it can be much greater.
  Let us analyze a simple example. Several swarm-based algorithms have been used for color quantization, including ABC [24], SFLA [25], PSO [26], CS and FA [27], and some other algorithms [28]. When swarm algorithms are applied to color quantization, the objective function considered is usually the mean squared error (MSE), and the solution vector includes $r = 3c$ components, where c is the number of colors in the quantized palette that defines the solution to the problem considering the RGB color space. In this case, to calculate the fitness of a solution (a palette), it is necessary to process the pixels of the original image and the quantized image that would be generated with said palette. If these images include P pixels, the calculation of the MSE error has a complexity of $O(rP)$, with P being a much larger value than r.
  Therefore, to make a more general representation, we consider that the complexity of this operation is $O(F)$, where $O(F) \geq O(r)$ and F is determined by the problem to be solved.

- Compute the fitness of all individuals in the population (op3): This requires computing the fitness N times. Therefore, the complexity is $O(FN)$

- Determine the initial best solution of the swarm, $gbest(0)$, with fitness fit_{gb}: once the best individual from the initial population has been identified, its information is used to define the initial global solution. There are two different cases for the first operation:
  - For the methods that do not sort the individuals, this operation requires checking the N fitness values to decide which is the best; so, this operation is $O(N)$.
  - The methods that sort the individuals by fitness (SFLA, FA, GWO, CHO) can directly select the value of $gbest(0)$, since it is at one end of the sorted list. This requires a constant time, $O(1)$.

  Regarding the second operation, it is necessary to copy r values into $gbest$ and store the fitness of the selected position as the initial value of fit_{gb}. The complexity of this operation is $O(r)$.
  Therefore, the overall complexity of the operation is $O(\max\{r, N\})$ for algorithms that apply sorting (op4) and $O(r)$ for those that do not apply sorting (op5).

- Update the best solution of the swarm, $gbest(t)$, with fitness fit_{gb}. In this case, 2 operations are carried out: first, the individual with the best solution within the current population is determined; then, if that individual's solution improves on the one currently stored as the global best, it is taken as the new global solution (otherwise the value is unchanged).
  Regarding the first operation, the two cases discussed in the previous point are distinguished:
  - For methods that do not sort the individuals, the operation that identifies said individual has complexity $O(N)$, since N individuals must be evaluated.
  - On the contrary, for methods that sort the individuals, identifying the individual that should be used to update $gbest(t)$ requires constant time ($O(1)$).

  The second operation is similar to the one described in the previous point, but in this case it is only executed if a condition is met, so the complexity is between $\Omega(1)$ and $O(r)$.

In short, for algorithms that do not sort individuals (op6), the complexity is between $\Omega(N)$ and $O(\max\{r, N\})$. For algorithms that sort individuals (op7), the complexity is between $\Omega(1)$ and $O(r)$.

- Adjust a new position to the valid range (op8). After calculating the new position of an individual, the r components of the new vector must be clipped to a predefined interval. This operation has complexity $O(r)$. This complexity is less than or equal to the complexity associated with the operation that updates the individual's position. Therefore, since the position is first recalculated and then adjusted to the valid interval, the step that includes both operations has the complexity associated with the operation that recalculates the position. For this reason, the rest of this section does not explicitly refer to the second operation.
  It should be noted that this adjustment is not required when the initial positions of the population are defined, since said positions are generated within the valid interval.
- Sort the individuals by fitness (op9). This operation sorts the individuals by increasing fitness and generates a list with the indexes of the individuals ordered by fitness. It is possible to apply a sorting method that has complexity $O(N \log(N))$, such as the mergesort method [29].

Table 2. Complexity of operations common to all algorithms.

	Description	Complexity
op1	Define the initial position of the N individuals	$O(rN)$
op2	Compute the fitness of a feasible solution or individual	$O(F)$ $(\geq O(r))$
op3	Compute the fitness of the N individuals	$O(FN)$ $(\geq O(rN))$
op4	Select the initial best solution of the swarm (without previous sorting)	$O(\max\{r, N\})$
op5	Set the initial best solution of the swarm (with previous sorting)	$O(r)$
op6	Update the best solution of the swarm (without previous sorting)	$[\Omega(N), O(\max\{r, N\})]$
op7	Update the best solution of the swarm (with previous sorting)	$[\Omega(1), O(r)]$
op8	Adjust a new position to the valid interval	$O(r)$
op9	Sort by fitness	$O(N \log(N))$

3.2. PSO complexity

Table 3 shows the computational complexity of each operation of the PSO algorithm. The first column refers to the line number that appears in the pseudocode of the algorithm. The operations that were already discussed in the previous section and that appear in Table 2 are identified with the code associated with each of them in the first column of said table (op1, op2, ...).

The operation that defines the initial velocity of the particles is similar to the one that defines the initial position, so they have the same complexity.

The operation that defines the initial personal best position of each particle simply copies N vectors of size r and their associated fitness, so the complexity is $O(rN)$. On the other hand, the operation that updates the personal best position of a particle i is only applied if a condition is met (if $fit_i < fit_{pb\,i}$). Therefore, the complexity of this update when considering the N particles is $\Omega(N)$ in the best case and $O(rN)$ in the worst case.

The equations that update the velocity and position of a particle (Eqs (1) and (2), respectively) define new values for the r components of both vectors, so the complexity of these updates is $O(r)$.

Therefore, the complexity of the previous operations when applied to the N particles is $O(rN)$.

Taking into account the T iterations of the iterative stage (which includes steps 7 to 11), the complexity of this block is $O(TFN)$, and this is the overall complexity of PSO. The largest of the three values will determine the final complexity in each particular case.

Table 3. PSO complexity.

Line		Description	Complexity
1	op1	Define the initial position of the particles	$O(rN)$
2		Define the initial velocity of the particles	$O(rN)$
3, 9	op3	Compute the fitness of each particle	$O(FN)$
4		Define the initial personal best position of the particles and its fitness	$O(rN)$
5	op4	Select the initial best solution of the swarm	$O(\max\{r, N\})$
7		Update the velocity of each particle	$O(rN)$
8		Update the position of each particle	$O(rN)$
10		Update the personal best position of each particle	$[\Omega(N), O(rN)]$
11	op6	Update the best solution of the swarm	$[\Omega(N), O(\max\{r, N\})]$
Overall: $O(TFN)$			

3.3. SFLA complexity

Table 4 shows the computational complexity of each operation of SFLA.

Table 4. SFLA complexity.

Line		Description	Complexity
1	op1	Define the initial position of the frogs	$O(rN)$
2	op3	Compute the fitness of each frog	$O(FN)$
3, 25	op9	Sort the frogs by increasing fitness	$O(N\log(N))$
4	op5	Set the initial best solution of the swarm	$O(r)$
6		Define m memeplexes	$O(N)$
9		Identify the best and worst frogs in a memeplex	$O(N)$
10, 15		Compute a candidate position x' for the worst frog	$O(r)$
11, 16	op2	Compute the fitness of a candidate position	$O(F)$
13, 18		Update the position and fitness of the worst frog	$O(r)$
20		Assign random values to x_w and compute its fitness	$O(F)$
26	op6	Update the best solution of the swarm	$[\Omega(N), O(\max\{r, N\})]$
Overall: $\max\{O(TN\log(N)), O(TN\,J_{max}\,\max\{F, N\})\}$			

The operation that defines the m memeplexes can be performed by assigning each individual the number of the memeplex to which it belongs. This operation requires processing the N individuals in the population, so its complexity is $O(N)$.

To identify the best and worst frog in a memeplex, it is necessary to analyze all the frogs in that memeplex, i.e., N/m frogs. The complexity of this operation is $O(N)$, since the constant m is discarded for the calculation of the complexity.

The operation that computes a candidate position x' (using Eq (3) combined with Eqs (4) or (5)) defines the r components of a position vector, so that its complexity is $O(r)$. The same complexity is associated with the operation that updates the position and fitness of the worst frog, given that it copies a vector of size r and a fitness value.

On the other hand, the operation that assigns random values to the position of the worst frog first assigns values to the r components of the position vector and then calculates its fitness. Therefore, the complexity of this operation is conditioned by the complexity of the fitness calculation ($O(F)$).

The complexity of the block of operations that applies $Jmax$ iterations to improve a memeplex (lines 8 to 23) is determined by the complexity of two operations: the operation that calculates the fitness of a position and the operation that identifies the best and worst frog of a memeplex. Therefore, the complexity of this block is $O(J_{max} \max\{N, F\})$. The complexity of the operation applied to all memeplexes is $O(N J_{max} \max\{F, N\})$. Since $m < N$ but the values are proportional, the value of N is considered when calculating the complexity.

The overall complexity of the algorithm is conditioned by two of the operations included in the main loop: the operation that sorts the frogs ($O(N \log(N))$) and the block of operations that processes each memeplex $O(N J_{max} \max\{F, N\})$. Therefore, the total complexity will be the maximum between the values $O(TN \log(N))$ and $O(TN J_{max} \max\{F, N\})$.

3.4. ABC complexity

Table 5 shows the computational complexity of each operation of the ABC algorithm.

The operation that sets to zero the *limit* variable associated with each of the N flowers has complexity $O(N)$.

The set of operations applied by the employed bees is analyzed below (lines 6 to 14). The operation that computes a candidate food source has complexity $O(r)$, since it defines the r values of a position vector. After computing the fitness of this candidate ($O(F)$), the obtained value determines the operation to be performed (lines 9–13). If the candidate source is better than the source i, the candidate source is used to update the source i, which has complexity $O(r)$, since the r values of the position vector must be copied; in the other case, only the value $limit_i$ must be updated, which has complexity $\Omega(1)$. This indicates that the complexity of the if-else construction defined in lines 9–13 is $\Omega(1)$ in the best case and $O(r)$ in the worst case. Therefore, the complexity of the operations applied to update an employed bee depends on the complexity of the fitness calculation, so the complexity of the operation applied to the N employed bees is $O(FN)$.

The operations of the onlooker bees are similar to the previous ones but require previously selecting the flower that will be associated with each bee.

To select a flower for each onlooker bee, the roulette wheel selection method can be applied [30]. This selection requires performing several successive operations that are described below, along with their complexity. First, it is necessary to calculate the probability of the N flowers using Eq (7), but considering $1/fit_i$ instead of fit_i, given that we want to apply the selection method to a minimization problem ($O(N)$). The probabilities are then ordered by increasing value ($O(N \log(N))$). After this, the cumulative probability CP is calculated for each of the N elements (that is, $CP_1 = p_1$ and $CP_i = p_i + CP_{(i-1)}$, for $i = 2, ..., N$), ($O(N)$). As a result of this operation, a vector of N elements in the interval $[0, 1]$ is obtained, which defines N subintervals. At this time a flower can be chosen for each of the N onlooker bees. To select a flower for bee j, a random number $a \in [0, 1]$ is generated, and the vector

CP is traversed from its first element, comparing a with the elements. The traversal concludes when the first element CP_i that is less than or equal to a is found. Then, flower i is selected for bee j ($\Omega(1)$ in the best case and $O(N)$ in the worst case). If this last operation is performed by applying binary search instead of sequential search, the complexity is reduced to $O(\log(N))$. Therefore, the complexity for the operation applied to the N onlooker bees is $O(N\log(N))$ in the worst case. In summary, the complexity of the selection operation by the roulette-wheel method is $O(N\log(N))$.

The operation that updates the exhausted food sources analyzes the N sources, although it does not always update all of them. When a flower must be updated, r new values in the range $[lb, ub]$ are generated for this flower, and then the fitness of the new position is computed. Therefore, the complexity of the operation is $\Omega(N)$ in the best case and $O(FN)$ in the worst case.

The global complexity of the algorithm is conditioned by two operations of the main loop: the operation that calculates the fitness and the one that selects the flowers for the onlooker bees. Therefore, the value is the maximum between $O(TN\log(N))$ and $O(TFN)$.

Table 5. ABC complexity.

Line		Description	Complexity
1	op1	Define the initial position of the food sources	$O(rN)$
2		Set $limit_i = 0$, for each food source i	$O(N)$
3	op3	Compute the fitness of each food source	$O(FN)$
4	op4	Select the initial best solution of the swarm	$O(\max\{r, N\})$
7		Compute a candidate source x'	$O(r)$
8	op2	Compute the fitness of x'	$O(F)$
9–13		If-else of employed bees operations	$[\Omega(1), O(r)]$
15		Select a food source for each onlooker bee	$O(N\log(N))$
17		Apply to flower i selected for bee j operations of employed bees	$O(F)$
19		Replace each food source x_i whose value $limit_i > L$	$[\Omega(N), O(FN)]$
20	op6	Update the best solution of the swarm	$[\Omega(N), O(\max\{r, N\})]$
Overall: $\max\{O(TN\log(N)), O(TFN)\}$			

3.5. FA complexity

Table 6 shows the computational complexity of each operation of FA.

The operation that updates the position of the brightest firefly has complexity $O(r)$, since it requires computing new values for the r components of the position vector. On the other hand, the operation that updates the position of the other $N-1$ fireflies has complexity $O(rN^2)$.

It is possible to take advantage of the order in the data to perform the second calculation. After sorting the data, the brightest firefly (the best) is at the beginning of the ordered sequence, and the least bright (the worst) is at the opposite end. Therefore, a loop can be used that runs through the ordered sequence of individuals, starting at the end (from $i = N$ to 2). In each iteration of said loop, a second loop is executed that runs through all the fireflies that are brighter than the firefly i (that is, from 1 to $i-1$). These loops allow updating the r components of the position vector of each firefly except the brightest. Therefore, the operations performed in these loops have compexity $O(rN^2)$.

The operations with the greatest complexity in the main loop of FA are the one that calculates the fitness of all the individuals ($O(FN)$) and the one that updates all the fireflies except the brightest one ($O(rN^2)$). Since the iterative process is applied T times, the overall complexity of the algorithm is the maximum between $O(TrN^2)$ and $O(TFN)$.

Table 6. FA complexity.

Line		Description	Complexity
1	op1	Define the initial position of the fireflies	$O(rN)$
2, 10	op3	Compute the fitness of each firefly	$O(FN)$
3, 11	op9	Sort the fireflies	$O(N\log(N))$
4	op5	Set the initial best solution of the swarm	$O(r)$
6–8		Update the position of $N-1$ fireflies	$O(rN^2)$
9		Update the position of the best firefly	$O(r)$
12	op7	Update the best solution of the swarm	$[\Omega(1), O(r)]$
Overall: $\max\{O(TrN^2), O(TFN)\}$			

3.6. GSA complexity

Table 7 shows the complexity of the operations included in GSA.

Table 7. GSA complexity.

Line		Description	Complexity
1	op1	Define the initial position of the objects	$O(rN)$
2		Define the initial velocity of the objects	$O(rN)$
3, 12	op3	Compute the fitness of each object	$O(FN)$
4	op4	Select the initial best solution of the swarm	$O(\max\{r, N\})$
2		Update G	$O(1)$
7		Compute the mass of each object	$O(N)$
8		Compute the total force that acts on each object	$O(rN^2)$
9		Compute the acceleration of each object	$O(rN)$
10		Update the velocity of each object	$O(rN)$
11		Update the position of each object	$O(rN)$
13	op6	Update the best solution of the swarm	$[\Omega(N), O(\max\{r, N\})]$
Overall: $\max\{O(TrN^2), O(TFN)\}$			

The operation that calculates the initial velocity of the objects has the same complexity as that which calculates the initial position ($O(rN)$).

The time needed to calculate the value of G is constant ($O(1)$).

To calculate the mass of the N objects, Eq (16) is applied N times. Then, the N values obtained are added, and each one is divided by the sum, as defined in Eq (15). Therefore, the computational complexity of the mass calculation is $O(N)$.

To calculate the total force acting on an object (Eq (17)), the remaining objects must be taken into account. The operation defines the r components of the force vector for an object. Therefore, the complexity of this operation applied to calculate the total force for each of the N objects is $O(rN^2)$.

To update the acceleration, position, and velocity of the N objects, the r components of the vectors of each object must be defined, so the complexity of each of these operations is $O(rN)$.

The two operations that determine the complexity of the main loop of the algorithm are the one that calculates the fitness of all the objects ($O(FN)$) and the one that calculates their total force ($O(rN^2)$). As a result, it is obtained that the complexity of the GSA method is the greater of the following two values: $O(TFN)$ and $O(TrN^2)$.

3.7. CS complexity

Table 8 shows the complexity of the operations included in CS algorithm.

Table 8. CS complexity.

Line		Description	Complexity
1	op1	Define the initial position of the host nests	$O(rN)$
2	op3	Compute the fitness of each nest	$O(FN)$
3	op4	Select the initial best solution of the swarm	$O(\max\{r, N\})$
5		Compute a candidate solution for each cuckoo by Eq (26)	$O(rN)$
6		Compute the fitness of each solution x_i'	$O(FN)$
8		Select a random nest j	$O(1)$
9–11, 18–20		Update a solution and its fitness if it is better than the current solution	$[\Omega(1), O(r)]$
13	op9	Sort the individuals by decreasing fitness	$O(N\log(N))$
16		Compute a candidate solution x' for nest i by Eq (27)	$O(r)$
17	op2	Compute the fitness of a solution	$O(F)$
23	op6	Update the best solution of the swarm	$[\Omega(N), O(\max\{r, N\})]$
Overall: $\max\{O(TFN), O(TN\log(N))\}$			

When using Eq (26) to compute a candidate solution for a cuckoo, the r components of a position vector are defined. Therefore, the complexity of this operation applied to the N individuals is $O(rN)$. After calculating the N candidate positions, the fitness of all of them is calculated. This operation has the same complexity as op3, since it is applied to the same number of solutions.

The complexity of calculating a new position using Eq (27) is the same as when Eq (26) is applied for an individual.

There are two blocks of operations that allow updating a solution from the current set if a better one has been found (lines 9–11 and 18–20). The operations are only executed if the condition is met, so the minimum complexity is $\Omega(1)$. If the condition is met, a vector of r components and the associated fitness are copied, which has a complexity of $O(r)$. Therefore, the complexity of both blocks varies between $\Omega(1)$ and $O(r)$.

The for loop included in lines 7 to 12 determines whether the candidate solutions calculated with Eq (26) replace randomly chosen nests. Since the time required to select a random nest is constant ($O(1)$), the complexity of this loop is determined by the if construction. Therefore, the complexity of the loop is between $\Omega(N)$ and $O(rN)$.

The next for loop (lines 14 to 22) is executed w times, with $w < N$. The operations of each iteration are executed only if a condition is met, so the complexity of the operations performed in each iteration is $\Omega(1)$ if the condition is not met and $O(F)$ if it is met. Therefore, the complexity of the for loop is between $\Omega(N)$ and $O(FN)$. It should be noted that to compute the complexity, N is considered instead of w because w is proportional to N.

The overall complexity of the algorithm is determined by the operations of the main loop. Taking all of the above into account, the overall complexity is the larger of $O(TFN)$ and $O(TN\log(N))$.

3.8. BA complexity

Table 9 shows the complexity of the operations included in BA.

Table 9. BA complexity.

Line		Description	Complexity
1	op1	Define the initial position of the bats	$O(rN)$
2		Define the initial velocity of the bats	$O(rN)$
3		Define the initial volume and pulse for each bat	$O(N)$
4	op3	Compute the fitness of each bat	$O(FN)$
5	op4	Select the initial best solution of the swarm	$O(\max\{r, N\})$
8		Compute the frequency of a bat	$O(r)$
9		Update the velocity of a bat	$O(r)$
10		Compute a candidate position x' by Eq (30)	$O(r)$
11– 14		Compute another candidate position x' by Eq (31) if condition is true	$[\Omega(1), O(r)]$
15	op2	Compute the fitness of x'	$O(F)$
16 –19		Update x_i, r_i and A_i if condition is true	$[\Omega(1), O(r)]$
21	op6	Update the best solution of the swarm	$[\Omega(N), O(\max\{r, N\})]$
		Overall: $O(TFN)$	

The operation that defines the initial velocity of the bats sets to 0 the r elements of the vector for each of the N individuals, so it has the same complexity ($O(rN)$) as op1.

To define the initial volume and pulse of the bats, it is necessary to store a value in each of the N positions of both vectors. This operation has complexity $O(N)$.

To calculate the frequency of a bat, r values must be calculated, so the complexity of the operation is $O(r)$. Updating the velocity of a bat and calculating a candidate position for that bat have the same complexity as the previous operation.

If the pulse of the bat is less than a random value in the interval $[0, 1]$, another candidate position must be calculated (lines 11 to 14). The calculation of the second candidate position has the same complexity as the calculation of the first ($O(r)$). To calculate this new position by Eq (31), the position of another bat, x_j, is used. Typically the global best position (*gbest*) is used, which does not require additional operations (it is possible to choose a bat from the current set, which has complexity $O(N)$, but this case is not considered here). Therefore, the total complexity of the operations included in lines 11 to 14 is $\Omega(1)$ in the best case (when the condition is not met) and $O(r)$ in the worst case.

When the volume of the bat is greater than a random value in the interval $[0, 1]$ and the candidate

position is better than the current position of this bat, the position, fitness, volume, and pulse of said bat are updated (lines 16–19). The complexity of these operations is $\Omega(1)$ in the best case and $O(r)$ as a maximum.

The operations in lines 8 to 19 are applied to each bat, so the complexity of these operations is conditioned by the complexity of the operation that computes the fitness. Therefore, the complexity of this block is $O(FN)$. Since these operations are included in the main loop that is executed T times, the overall complexity of the algorithm is $O(TFN)$.

3.9. GWO complexity

Table 10 shows the analysis of GWO.

Table 10. GWO complexity.

Line		Description	Complexity
1	op1	Define the initial position of the wolves	$O(rN)$
2, 8	op3	Compute the fitness of each wolf	$O(FN)$
3, 9		Identify the α, β and δ wolves	$O(N)$
4	op5	Set the initial best solution of the swarm	$O(r)$
5		Set $a = a_0$	$O(1)$
7		Update the position of each wolf	$O(rN)$
10	op7	Update the best solution of the swarm	$[\Omega(1), O(r)]$
11		Update a by Eq (43)	$O(1)$
Overall: $O(TFN)$			

The operation that identifies the index of the three best wolves in the group has complexity $O(N)$, since all individuals must be evaluated to identify the α, β, and δ wolves.

To update the position of the N wolves, it is necessary to calculate a new value for each of the r components of the position vector of each wolf. Therefore, this operation has complexity $O(rN)$.

The time required to set the initial value of a and to update a is constant.

Taking into account that operations 7 to 11 are part of the iterative process of the algorithm, the total complexity of the GWO method is $O(TFN)$.

3.10. CHO complexity

Table 11 shows the complexity of the operations associated with the CHO algorithm.

The operation that defines the type of the N individuals has complexity $O(N)$.

The operation that associates a rooster with each hen chooses the rooster randomly, so it has complexity $O(H)$ for the set of H hens.

The operation that associates a mother with each of the C chicks first selects M of the H hens as mothers (with $M \leq H$) and then assigns a random mother to each of the C chicks. Therefore, the complexity of this operation depends on the number of hens and chicks considered, so it is the maximum between $O(H)$ and $O(C)$.

The operation that calculates a candidate position for an individual applies a different equation for each type of individual. However, the operation that calculates this new position has complexity $O(r)$ for all three types of individuals, since it requires updating the r components of the position

vector of the individual. Therefore, the operation that calculates a candidate position for each of the N individuals has complexity $O(rN)$.

When the candidate position is better than the current position of an individual, the candidate is taken as the new position of said individual (lines 18 to 20). Since the update is not performed if the condition is not met, the complexity of this operation is between $\Omega(1)$ and $O(r)$. Therefore, the loop that processes all individuals (lines 16-21) has complexity $O(FN)$, since it is conditioned by the complexity of the operation that calculates the fitness of all individuals.

The overall complexity of the algorithm is determined by the complexity of the main loop operations. This loop includes two operations that determine the final complexity: the calculation of the fitness of all individuals and the ordering of the individuals. As a result, it is obtained that the complexity of the CHO algorithm is the maximum between $O(TNlog(N))$ and $O(TFN)$.

It can be seen that the ordering is executed if a condition is met. This condition depends on G, which is a value proportional to T. Therefore, to determine the complexity associated with the sorting operation, T is considered instead of G.

Table 11. CHO complexity.

Line		Description	Complexity
1	op1	Define the initial position of the individuals	$O(rN)$
2	op3	Compute the fitness of each individual	$O(FN)$
3, 10	op9	Sort the individuals by increasing fitness	$O(N\log(N))$
4, 11		Define the type of each individual	$O(N)$
5, 12		Define the relationship between hens and roosters	$O(H)$
6, 13		Define the relationship between chicks and mothers	$O(\max\{H, C\})$
7	op5	Set the initial best solution of the swarm	$O(r)$
15		Compute a candidate position for each individual	$O(rN)$
17	op2	Compute the fitness of the candidate position	$O(F)$
18–20		Update the current position of the individual if necessary	$[\Omega(1), O(r)]$
22	op6	Update the best solution of the swarm	$[\Omega(N), O(\max\{r, N\})]$
Overall: $\max\{O(TNlog(N)), O(TFN)\}$			

3.11. WOA complexity

Table 12 shows the computational complexity of each operation of WOA.

The time required to set the initial value of a and to update a is constant.

The operations that update the position of the whales (lines 6–12) have complexity $O(rN)$. This set of operations calculates a new position for each of the N particles. The calculation of each position involves defining the r components of the position vector.

Taking into account the iterative process that this method applies to improve the positions of the whales, the overall complexity of the algorithm is $O(TFN)$. It depends on the complexity of the operation that computes the fitness of the individuals.

Table 12. WOA complexity.

Line		Description	Complexity
1	op1	Define the initial position of the whales	$O(rN)$
2, 13	op3	Compute the fitness of each whale	$O(FN)$
3	op4	Select the initial best solution of the swarm	$O(\max\{r, N\})$
4		Set $a = a_0$	$O(1)$
6–12		Update the position of the whales (by Eqs (50) or (55))	$O(rN)$
14	op6	Update the best solution of the swarm	$[\Omega(N), O(\max\{r, N\})]$
15		Update a by Eq (43)	$O(1)$
Overall: $O(TFN)$			

3.12. Comparative analysis

This last subsection compares the algorithms, taking into account the computational complexity, the number of parameters, and the variables required to apply the operations of each algorithm.

Four groups of algorithms with different computational complexity have been identified:

- BA, GWO, PSO, and WOA – $O(TFN)$.
- ABC, CS, and CHO – $\max\{O(TFN), O(TN\log(N))\}$.
- FA, and GSA – $\max\{O(TFN), O(TN^2r)\}$.
- SFLA – $\max\{O(TNJ_{max}\max\{N, F\}), O(TN\log(N))\}$.

It can be observed that BA, GWO, PSO, and WOA have the lowest computational complexity.

The number of parameters of a method is an important factor since, in general, it is necessary to tune the parameters to obtain the best possible result for a specific problem. Therefore, methods that include fewer parameters require less work to tune them.

All analyzed methods use a common set of parameters: population size (N), number of iterations (T), size of a solution vector (r), and lower and upper bounds, for a solution vector (lb and ub, respectively). Table 13 summarizes the specific parameters of each method. In the case of the CHO algorithm, C is not included as a parameter, since it can be calculated from N, R, and H. The size of the solution vector is determined by the problem to which the swarm algorithm is applied, but the rest of the parameters can be adjusted when applying the algorithm.

It is clearly observed that GWO and ABC are the methods with fewer parameters, while BA is the one that includes the most. In the case of the ABC algorithm, a single parameter is considered because the number of employed bees and onlooker bees is taken equal to the number of flowers, as proposed by the author of the method [14], although some variants of the algorithm consider a number of bees different from the number of flowers.

The memory space required by an algorithm is another important issue. Table 14 shows the main variables used by all the analyzed algorithms. This table does not include the parameters already described in Table 13 or the counters used in the loops. The variables have been divided into two categories: those that are associated with each individual in the swarm and those that are associated with the population. In cases where a specific name was not given to the variables when defining the algorithms, a description is included.

Table 13. Specific parameters of each swarm-based method.

Method	Parameters	Type
PSO	ω : inertia weight	real
	c_1 : cognitive component weight	real
	c_2 : social component weight	real
	min_v : minimum for the values of a velocity vector	vector of r reals
	max_v : maximum for the values of a velocity vector	vector of r reals
SFLA	m : number of memeplexes	integer
	D_{max} : limit in the change allowed in the position of a frog	real
	J_{max} : number of improvement iterations for a memeplex	integer
ABC	L : maximum number of attempts to improve a food source	integer
FA	β_0 : attractiveness at distance 0	real
	γ : light absorption coefficient	real
	α : randomization parameter	real
GSA	G_0 : initial value of the gravitational constant	real
	α : parameter used to update the gravitational constant	real
CS	pa : mutation probability	real
	w : number of the worst nests evaluated for mutation	integer
	β : scale factor to compute Lévy flights	real
BA	f_{min} : minimum frequency value	real
	f_{max} : maximum frequency value	real
	A_0 : initial volume	real
	R_0 : initial pulse emission index	real
	α : constant used to update the volume	real
	γ : constant used to update the pulse emission index	real
GWO	a_0 : initial value to compute a	real
CHO	R : number of roosters	integer
	H : number of hens	integer
	M : number of mothers	integer
	G : parameter to decide when to update relationships	integer
	P : parameter used to update the position of a chick	real
WOA	a_0 : initial value to compute a	real
	b : parameter to define the shape of the logarithmic spiral	real

Table 14. Variables of each method (common to all of them and additional for several methods. FA does not use extra variables).

Variables for the swarm		
Method	Variable	Storage space
all	$gbest(t)$	vector of size r – reals
	fit_{gb}	real
ABC	index of the source selected for each onlooker bee	vector of size N – integers
CHO	C	integer
	relation hen-rooster	vector of size H – integers
	relation chick-hen	vector of size C – integers
	type of each individual	vector of size N – integers
GSA	G	real
SFLA	memeplex of each frog	vector of size N – integers
CS	σ	real
WOA	a	real
GWO	a	real
	α, β, δ	3 integer values
Variables for each individual i (for $i = 1, ..., N$)		
Method	Variable	Storage space
all	$x_i(t)$	vector of size r – reals
	fit_i	vector of size r – reals
ABC	$limit_i$	integer
CHO	x'_i	vector of size r – reals
GSA	$v_i(t)$	vector of size r – reals
	$F_i(t)$	vector of size r – reals
	$F_{ij}(t)$ $(j = 1, ..., N, i \neq j)$	vector of zise r – reals
	$a_i(t)$	vector of size r – reals
	$M_i(t)$	real
	$m_i(t)$	real
PSO	$v_i(t)$	vector of size r – reals
	$pbest_i(t)$	vector of size r – reals
	$fit_{pb\,i}$	real
BA	$v_i(t)$	vector of size r – reals
	$A_i(t)$	real
	$r_i(t)$	real

PSO, FA, and BA do not require additional variables associated with the swarm. When applying the CS algorithm, a variable can be used to store the value of σ calculated with Eq (24), so this operation will only be performed once. The operation of ABC that selects a food source x_i for each onlooker bee (line 15 of Algorithm 3) should store the index of the food source selected for each bee. As already indicated, in the CHO method, the number of chicks, C, is calculated from the parameters N, R, and H. In this case, the type of each individual must also be identified (this can be done by storing an integer value that represents each of the 3 possible types). In addition, for each chick the index of its mother

must be stored, and for each hen the index of the rooster that has been chosen as leader must be stored. In SFLA, a vector of N integers can be used to store the number of the memeplex to which each frog is associated.

CS, FA, GWO, SFLA, and WOA do not require additional variables associated with each individual in the swarm. ABC and CHO only require one additional variable for each individual, but the CHO variable requires more storage space (it is a vector to store a candidate position for the individual). Clearly, GSA is the method that requires the most memory space to store the additional data associated with each individual.

When analyzing the storage space required by an algorithm, the treatment applied to the fitness function must be taken into account. Some code implementations calculate the fitness function whenever necessary, and the calculated value is not stored. That is, if the same value must be used in several operations, it must be recalculated. This can deeply influence the execution time of the algorithm when applied to problems in which the fitness calculation has a high computational cost. An example of this situation was described in section 3.1 for the case of swarm-based methods applied to solve the color quantization problem. In similar cases, it is recommended to store the fitness of the individuals to avoid unnecessary new calculations. For this reason, Table 14 shows the memory space required to store these values. Obviously, storing the values will improve the execution time of the algorithms.

4. Experimental results

This section describes the results of the experiments carried out. The swarm-based methods were coded using the Python programming language. Table 15 shows the values used for the specific parameters of each algorithm. The values used are those proposed by the authors and those that have generated good results when applying the algorithms to multiple problems. On the other hand, since N, r, and T are the parameters considered fundamental during the analysis of the computational complexity of the algorithms, several values are considered for each of these parameters: $\{25, 50, 100\}$.

Table 15. Control parameters for the swarm-based methods used in the experiments.

Algorithm	Control parameters
PSO	$c_1 = c_2 = 1.49$, $\omega = 0.72$, $min_v = lb$, $max_v = ub$
SFLA	$m = 1 + \lceil 0.1N \rceil$, $D_{max} = ub/2$, $J_{max} = \lceil T/2 \rceil$
ABC	$L = \lceil T/4 \rceil$
FA	$\beta_0 = 1$, $\gamma = 0.1$, $\alpha = 1$
CS	$pa = 0.25$, $w = N/2$, $\beta = 1.5$,
GSA	$G_0 = 100$, $\alpha = 20$
BA	$f_{min} = 0$, $f_{max} = 100$, $A_0 = 1$, $R_0 = 0.5$, $\alpha = 0.9$, $\gamma = 0.9$
GWO	$a_0 = 2$
CHO	$R = \lceil 0.2N \rceil$, $H = \lceil 0.6N \rceil$, $C = N - R - H$, $M = 0.1N$, $G = 10$, $P = 0.5$
WOA	$a_0 = 2$, $b = 1$

The methods were applied to a set of 20 objective functions commonly used in the literature, shown in Table 16 (functions f1 to f10 are unimodal, while the remaining functions are multimodal). The

table also shows the lower and upper bounds for the components of the solution vectors. The optimum value of all these functions is 0. More information about the functions can be obtained in [31].

The tests were carried out on a PC with a Windows 10 Pro operating system, 16 Gb of RAM, and an Intel Core i3-4160 3.60GHz CPU. Twenty independent tests of each algorithm were performed for each objective function and each of the 27 possible combinations of the values of N, r, and T.

Table 16. Functions used for the tests and hypercube in which each function is evaluated ($x_i \in [lb_i, ub_i]$, for $i = 1, ..., r$).

Name	Equation	$[lb_i, ub_i]$
Brown	$f1(x) = \sum_{i=1}^{r-1}(x_i^2)^{(x_{i+1}^2+1)} + (x_{i+1}^2)^{(x_i^2+1)}$	$[-1, 4]$
Chung Reynolds	$f2(x) = \left(\sum_{i=1}^{r} x_i^2\right)^2$	$[-100, 100]$
Dixon and Price	$f3(x) = (x_1 - 1)^2 + \sum_{i=2}^{r} i(2x_i^2 - x_{i-1})^2$	$[-10, 10]$
Quartic	$f4(x) = \sum_{i=1}^{r} i\, x_i^4 + random[0, 1)$	$[-1.28, 1.28]$
Rosenbrock	$f5(x) = \sum_{i=1}^{r-1}[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$	$[-5, 10]$
Rotated hyper-ellipsoid	$f6(x) = \sum_{i=1}^{r}\sum_{j=1}^{i} x_j^2$	$[-100, 100]$
Step	$f7(x) = \sum_{i=1}^{r}(\lfloor x_i \rfloor + 0.5)^2$	$[-100, 100]$
Sphere	$f8(x) = \sum_{i=1}^{r} x_i^2$	$[-100, 100]$
Sum of different powers	$f9(x) = \sum_{i=1}^{r} \lvert x_i \rvert^{i+1}$	$[-10, 10]$
Sum of squares	$f10(x) = \sum_{i=1}^{r} i\, x_i^2$	$[-10, 10]$
Ackley	$f11(x) = -20\exp\left(-0.2\sqrt{\frac{1}{r}\sum_{i=1}^{r} x_i^2}\right)$	
	$-\exp\left(\frac{1}{r}\sum_{i=1}^{r}\cos(2\pi x_i)\right) + 20 + \exp(1)$	$[-32, 32]$
Alpine 1	$f12(x) = \sum_{i=1}^{r} \lvert x_i \sin(x_i) + 0.1x_i \rvert$	$[-10, 10]$
Csendes	$f13(x) = \sum_{i=1}^{r} x_i^6\left(2 + \sin\frac{1}{x_i}\right)$	$[-1, 1]$
Drop-wave	$f14 = 1 - [1 + \cos(12\sqrt{\sum_{i=1}^{r} x_i^2})]/[0.5\sum_{i=1}^{r} x_i^2 + 2]$	$[-5.12, 5.12]$
Griewank	$f15(x) = 1 + \frac{1}{4000}\sum_{i=1}^{r} x_i^2 - \prod_{i=1}^{r}\cos\left(\frac{x_i}{\sqrt{i}}\right)$	$[-100, 100]$
Levy	$f16 = \sin^2(\pi w_1) + \sum_{i=1}^{r-1}\left[(w_i - 1)^2(1 + 10\sin^2(\pi w_i + 1))\right]$	$[-10, 10]$
	$+[(w_r - 1)^2(1 + \sin^2(2\pi w_r)]$	
	where $w_i = 1 + (x_i - 1)/4$, $i = 1, ..., r$	
Rastrigin	$f17(x) = 10r + \sum_{i=1}^{r}[x_i^2 - 10\cos(2\pi x_i)]$	$[-5.12, 5.12]$
Salomon	$f18(x) = 1 - \cos\left(2\pi\sqrt{\sum_{i=1}^{r} x_i^2}\right) + 0.1\sqrt{\sum_{i=1}^{r} x_i^2}$	$[-100, 100]$
Schwefel	$f19(x) = 418.9829r - \sum_{i=1}^{r} x_i \sin(\sqrt{\lvert x_i \rvert})$	$[-500, 500]$
Zakharov	$f20(x) = \sum_{i=1}^{r} x_i^2 + (\sum_{i=1}^{r} 0.5i\, x_i)^2 + (\sum_{i=1}^{r} 0.5i\, x_i)^4$	$[-5, 10]$

Figure 1 shows the average execution time of each method. Each subfigure shows the average values obtained for the 27 combinations of the parameters N, r, and T. To facilitate interpretation, cases with the same value of N have been joined by a line. It is clearly observed that the execution time of all the algorithms increases with the values of the parameters r, N, and T.

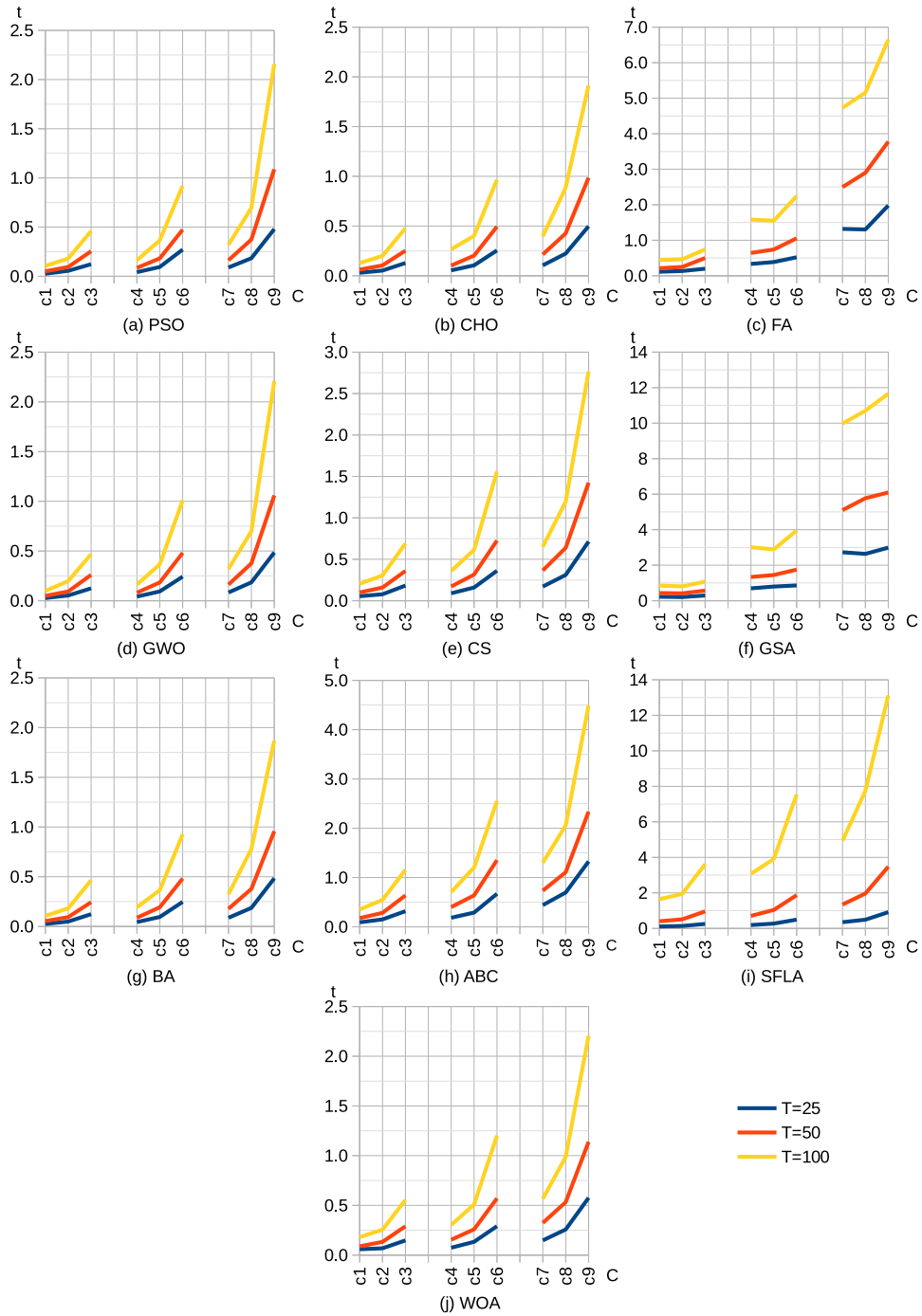


Figure 1. Average execution time (seconds). C: combination of parameters [c1: (N = 25, r = 25), c2: (N = 25, r = 50), c3: (N = 25, r = 100), c4: (N = 50, r = 25), c5: (N = 50, r = 50), c6: (N = 50, r = 100), c7: (N = 100, r = 25), c8: (N = 100, r = 50), c9: (N = 100, r = 100)].

Considering the 27 cases represented, PSO is the best overall method, as it is the fastest method in 13 cases and the second fastest in 7 cases. After PSO, GWO and BA are the best methods. The 14 cases for which PSO is not the best method correspond to GWO (8 cases) or BA (6 cases). Although

GWO and BA have similar times in many cases, BA is always among the top 3 methods, but GWO is the fourth or fifth method for 3 cases with $r = 100$.

CHO is the fourth method in most cases (21), although it is the second in 4 cases and the third in 2 cases. WOA is the fifth method in 25 cases, while CS is the sixth in 26 cases. Certainly, WOA and CS define a point that clearly separates the other methods into two groups: PSO, GWO, BA, and CHO are the fastest, while GSA, SFLA, FA, and ABC are the slowest.

GSA and SFLA are the slowest methods. GSA is the slowest method in 16 cases, while SFLA is the slowest in the other 10 cases. GSA is worse than SFLA for the 9 cases corresponding to $T = 25$, but it is better in 7 of the cases corresponding to $T = 100$. FA is faster than GSA in all cases. It is also faster than SFLA for all cases considering $T = 100$, but only for 6 cases with $T = 50$ and 2 cases with $T = 25$. ABC is faster than FA in 18 out of 27 cases (including the 9 cases with $N = 100$).

Although the computational complexity of FA and GSA is the same, the figures show that GSA requires more execution time. In this case it must be taken into account that updating the positions of the individuals requires many more calculations in the case of GSA (requires prior calculation of mass, total force, acceleration, and velocity).

The computational complexity of CS, ABC, and CHO is also the same, but CHO is clearly faster than the other two methods. It must be taken into account that each iteration of ABC calculates at least $2N$ candidate positions, while CHO calculates N. CS is halfway between both methods, as it calculates between N and $N + w$ candidate positions (with $w < N$). Furthermore, in all 3 cases the fitness of each of the candidate positions is calculated. On the other hand, although all 3 methods apply a sorting operation, CHO does not do so in all iterations of the algorithm.

The case of GWO and WOA is significant. Despite having the same computational complexity, a very similar structure, and even some similar equations, it is observed that WOA is slower. It must be taken into account that WOA calculates an exponential and a cosine in Eq (55), which are more expensive operations for the processor than those used by GWO.

Figure 1 clearly shows that the execution time of the algorithms increases with the values of N, r, and T. To facilitate the analysis of the effect of these parameters, Figure 2 shows the grouped results considering each of the 3 parameters independently. It is clearly observed that the subfigure that groups results by value of r obtains better times for SFLA, FA, and GSA than the other two parameters when the value 100 is considered. On the contrary, for the rest of the methods, worse times are obtained in that case. Certainly, SFLA, FA, and GSA are the methods that show the greatest differences in the 3 subfigures.

The execution time of SFLA increases more with the value of T because this parameter also determines the number of improvement iterations of each memeplex ($J_{max} = T/2$). On the other hand, N conditions the number of memeplexes defined in the tests (4 memeplexes for $N = 25$, 6 memeplexes for $N = 50$, and 11 memeplexes for $N = 100$). Increasing memeplexes increases the number of times the process that applies J_{max} improvement iterations must be applied.

In the case of FA, the subfigure that shows the results for the 3 values of N is the one that shows the greatest differences among the 3 cases represented. The value of N influences the time needed to classify the fireflies, which is a time-consuming operation in this algorithm. Although the value of T determines the number of times the sorting operation should be applied, it is observed that this parameter has less influence.

It is observed that the execution time of GSA varies very little with r but a lot with N. It must

be taken into account that the parameter N significantly influences the calculation time of Eqs (17) and (18). The second parameter that most influences these calculations is the number of times it must be performed (T).

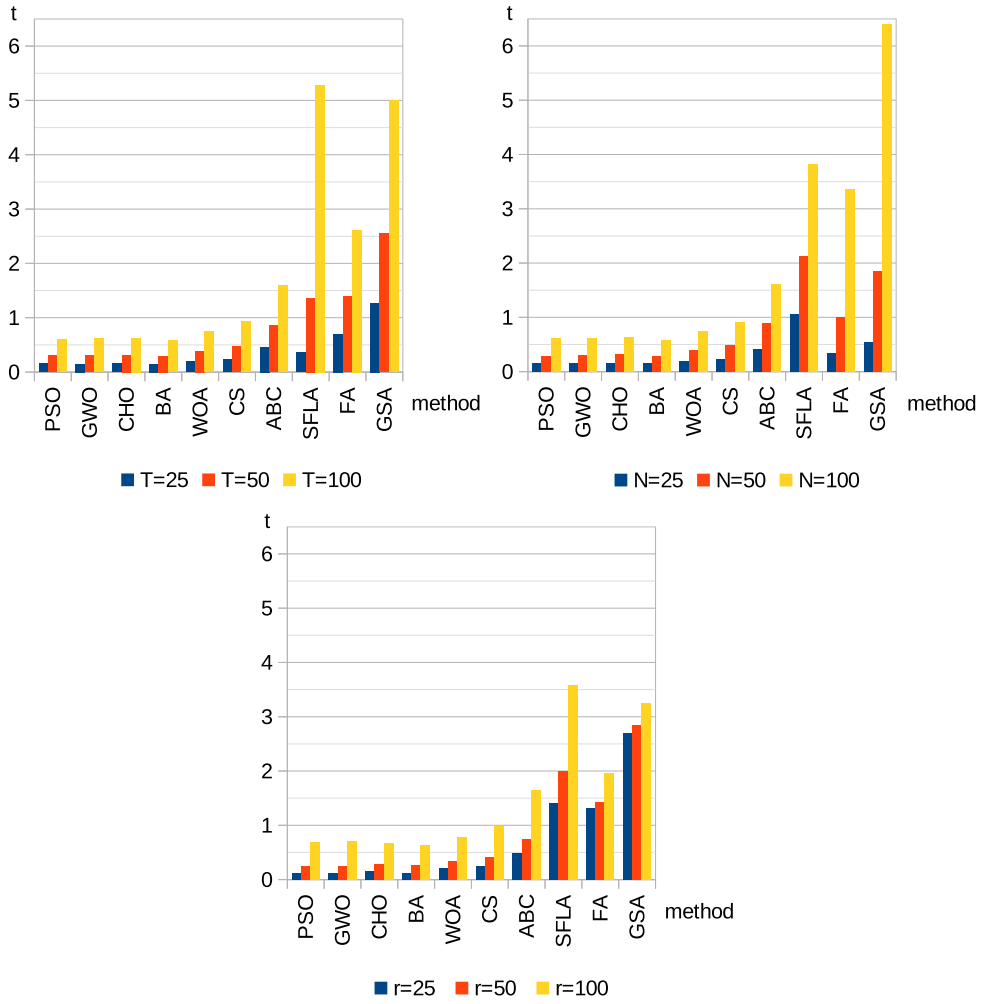


Figure 2. Average execution time (seconds) of each method for each value of T, N, and r.

Although GSA and FA exhibit similar theoretical computational complexities, GSA consistently demonstrates longer execution times in practice. This discrepancy can be attributed to the internal computational overhead associated with its dynamics. Unlike FA, GSA requires multiple intermediary calculations per agent per iteration, including mass determination, total gravitational force, acceleration, velocity, and position updates. These operations are particularly sensitive to the population size N, as each agent interacts with all others, leading to a high volume of pairwise computations. The reliance of GSA on cumulative force aggregation and acceleration updates introduces additional processing steps not present in FA. In GSA, each agent interacts with all others, requiring the computation of pairwise distances and force vectors in each iteration, which becomes increasingly costly with larger populations and problem dimensions. In contrast, FA typically limits interactions to brighter neighbors, reducing the number of computations per iteration. The results also

show that the execution time of GSA scales more strongly with N and T, highlighting its intensive force calculation steps as major contributors to its inefficiency. While FA limits its interactions and uses simpler update rules, the idea behind GSA, although conceptually elegant, results in heavier computational demands. These factors collectively contribute to the longer runtime of GSA despite its algorithmic complexity comparable to that of FA and make GSA the slowest method in most of the analyzed scenarios.

The number of times the objective function is evaluated is an important issue to determine the time consumption of an algorithm. PSO, GWO, FA, CHO, BA, WOA, and GSA perform N evaluations per iteration, but the other 3 methods perform more evaluations. ABC evaluates the fitness twice for each food source ($2N$ evaluations). However, scout bee operations make it necessary to calculate the fitness when a new source replaces an exhausted source. The average number of evaluations per iteration carried out in the tests was $(52, 104, 207)$ for $N = 25, 50, 100$, respectively. On the other hand, CSA evaluates the fitness at least once for each individual but may perform a second evaluation depending on a random value. The average number of evaluations carried out in the tests was $(34, 69, 138)$ for $N = 25, 50, 100$, respectively. It is clearly observed that SFLA is the method that performs the most fitness calculations, since the average results obtained in this case were $(95, 189, 346)$, for $N = 25, 50, 100$ respectively. In this algorithm, each iteration evaluates the fitness function between mJ_{max} and $3mJ_{max}$ times.

The Friedman test was applied to determine whether the applied swarm-based method has an effect on the results of the experiments. This test compares several related samples and determines whether there are significant differences between any of the pairs of the samples. Each swarm-based method was applied to the 20 objective functions, performing 20 independent tests for the 27 combinations of values of the parameters r, N, and T. The average of each group of 20 independent tests was calculated, finally obtaining 540 data points for each method.

The Friedman test was applied to analyze two cases: the first case compares the execution time, while the second compares the value of the objective function obtained by the methods. The p-value considered to apply the test was 0.05. The test statistic obtained was 4483.933 in the first case and 3312.611 in the second case. The significance obtained in both cases was 0, so the Friedman test is significant for any p-value. Therefore, the test indicates that the obtained result (execution time and objective function value) is not the same for the ten swarm-based methods compared. This result indicates that at least two of the methods compared are significantly different, but the test does not indicate which ones. This makes it necessary to apply an additional test to determine which methods present differences. For this, the Wilcoxon post-hoc test was conducted, with the Bonferroni correction to control the probability of occurrence of a type I error. This operation requires comparing 45 pairs of swarm-based methods. Since the test was significant for most of the 45 cases that were evaluated, Table 17 only shows the cases corresponding to non-zero corrected p-values in order to reduce the table size.

It can be seen that the differences are non-significant for 5 pairs of methods when comparing execution time and for 4 pairs of methods when comparing the value of the objective function (in this case the difference between SFLA and WOA is significant). It is observed that the results related to the execution time are not associated with the same pairs of methods as those related to the value of the objective function. For all the pairs of methods not listed in Table 17, the differences are significant.

As already indicated in the previous analysis, PSO, GWO, and BA are the fastest methods. The

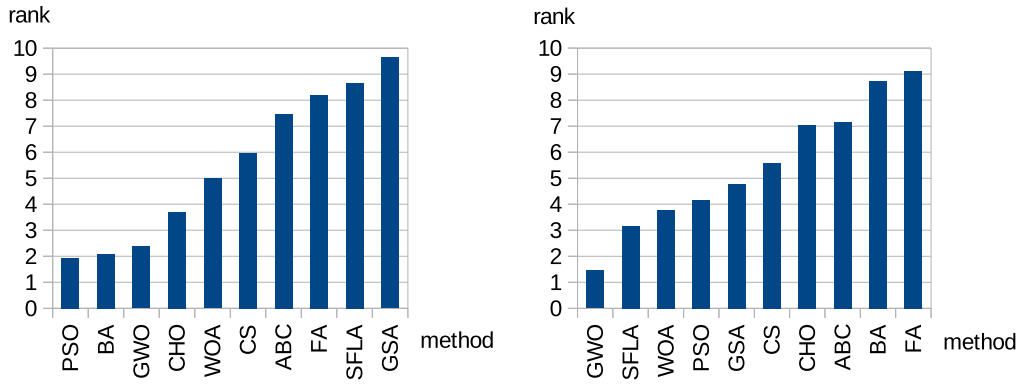
statistical test now indicates that there are no significant differences in the execution time of GWO and BA. The same goes for PSO and the other two methods.

Table 17. Corrected significance of post-hoc test (only for non-significant cases).

Execution time		Objective function value	
BA – GWO	1.000	BA – FA	1.000
PSO – BA	1.000	PSO – WOA	1.000
PSO – GWO	0.887	PSO – GSA	0.176
FA – SFLA	0.604	ABC – CHO	1.000
FA – ABC	0.050	SFLA – WOA	0.009

Although the objective of the tests is to analyze the execution time of the different methods, it is also interesting to analyze the result obtained when evaluating the objective function. The value obtained for the functions increases with the value of r, so the range of variation of the results does not allow a useful graphical representation of said results. However, a global comparison of the 10 methods can be made by ranking them based on the value obtained for the objective function.

The Kendall W test was applied to obtain a ranking of the methods. The test was applied twice, considering in one case the execution time and in the other the value of the objective function. The information generated with this test allows for a global comparison of all methods. Figure 3 shows the methods ranked based on the values of the test. In terms of execution time, PSO and BA are the fastest, while GSA is the slowest. On the other hand, GWO is the method that generates the best results, while BA and FA generate the worst results.



(a) average execution time (b) average objective function

Figure 3. Ranking of the methods.

GWO generates good results and is faster than most of the other methods. On the other hand, FA is among the slowest methods and generates the worst global results. Although SFLA is among the slowest methods, it is also among those that generate good results, which is not the case with GSA. It is also observed that the results of ABC and CHO are very similar, although CHO is faster.

PSO and GWO seem to be the best methods to have a balance between the execution time of the algorithm and the result it generates.

The ranking of execution time generates results coherent with the computational complexity defined for the algorithms. PSO, BA, GWO, and WOA have complexity $O(TFN)$ and are among the five better methods in the ranking. It should be noted that CHO is before WOA in the ranking, despite its complexity being $\max\{O(TFN), O(TN\log N)\}$. Next, CS and ABC are in the following positions in the ranking (having the same complexity as CHO). FA and GSA are in the same category of computational complexity, but it can be observed that SFLA is ranked better than GSA.

5. Accuracy versus speed in swarm-based algorithms: discussion and future work

Before concluding the article, we consider it interesting to reflect on the trade-offs between speed and accuracy in swarm-based algorithms.

The efficiency of swarm-based algorithms, measured in terms of both convergence speed and precision of the obtained solution, is a crucial aspect in their practical applications. Often, there is an inherent trade-off between these two objectives. Achieving a high-precision solution may require a larger number of iterations and, therefore, a longer computation time. Conversely, seeking rapid convergence may lead to suboptimal solutions. Several mechanisms and strategies can be employed in the design of swarm-based algorithms to manage this balance, including early termination criteria, adaptive parameters, and hybridization with other algorithms.

Implementing termination criteria that stop the algorithm before reaching a predefined maximum number of iterations can significantly reduce computation time. These criteria can be based on the detection of minimal improvement in fitness over a certain number of consecutive iterations, the achievement of a pre-established fitness threshold, or the detection of swarm convergence within a specific radius. While these criteria can accelerate the obtaining of a solution, there is a risk of premature termination that prevents complete exploration of the search space and, therefore, the identification of more accurate solutions. The choice of an appropriate termination criterion and its associated parameters (e.g., the number of iterations without improvement, the fitness threshold) directly influences the balance between speed and the quality of the final solution.

Many swarm-based algorithms incorporate parameters that control the behavior of the swarm, such as the inertia, cognitive, and social coefficients of PSO. The dynamic adjustment of these parameters during the execution of the algorithm can offer a way to balance the exploration and exploitation of the search space. For example, an initial phase with parameters that favor exploration can help identify promising regions, while a later phase with parameters that emphasize exploitation can refine the solutions found. Parameter adaptation strategies can be deterministic (based on the number of iterations or the progress of the search) or self-adaptive (based on the performance of the swarm itself). Effective adaptation can lead to faster convergence towards high-quality solutions.

Combining swarm-based algorithms with other optimization techniques, such as local search algorithms or gradient-based methods, can be another strategy to improve accuracy without excessively sacrificing speed. For example, a swarm-based algorithm can be used to identify promising regions of the search space, and then a local search algorithm can be applied to refine the solutions within those regions. The efficiency of these hybrid strategies depends on the correct integration of the different algorithms and the management of the transition between their operating phases.

Future designs of swarm-based algorithms could prioritize several specific improvements to achieve a more efficient balance between runtime efficiency and solution quality. Some of these promising

directions include parallelization, heuristic simplification, advanced self-adaptation mechanisms, and the design of problem-specific operators.

The inherently population-based nature of many swarm-based algorithms makes them ideal candidates for parallelization. The simultaneous evaluation of the fitness of multiple individuals, as well as the parallel updating of their states, can significantly reduce computation time, especially on parallel computing architectures such as GPUs or multi-core clusters. Investigating efficient strategies for workload distribution and communication between processors in the context of swarm-based algorithms could lead to much faster implementations without compromising the ability of the algorithm to explore the search space effectively. However, it is crucial to analyze the communication overhead introduced by parallelization to ensure a net gain in performance.

Some swarm-based algorithms have become quite complex with the addition of multiple operators and parameters to improve their performance on various problems. In future research, it could be beneficial to explore the simplification of the underlying heuristics. This could involve identifying the most influential components of the algorithm and eliminating or simplifying those that contribute marginally to the solution quality but significantly increase computational complexity. A more parsimonious design could not only improve runtime efficiency but also facilitate the understanding and theoretical analysis of the algorithm. However, any simplification must be done carefully to avoid a significant degradation in the exploration and exploitation capabilities of the algorithm.

While parameter adaptation strategies already exist, the development of more sophisticated and robust self-adaptation mechanisms could further improve the balance between speed and accuracy. This could include the use of machine learning or more complex feedback techniques to dynamically adjust not only the parameters but also the structure of the algorithm itself during its execution. For example, the algorithm could adapt its exploration strategy based on the diversity of the swarm or the rate of solution improvement. Designing such self-adaptive mechanisms requires a deep understanding of the dynamics of algorithms and the ability to extract useful information from the search process in real time.

The efficiency of a swarm-based algorithm often depends on the suitability of its operators (e.g., movement rules, social interactions) for the specific optimization problem. Future research could focus on the design of more specialized operators that exploit the inherent characteristics of certain classes of problems. This could involve incorporating domain-specific knowledge into the design of the algorithm, which could lead to faster convergence and higher-quality solutions for those particular problems. However, it is important to balance specialization with generality to maintain the applicability of the algorithm to a wider range of problems.

Exploring these and other future directions in the design of swarm-based algorithms is crucial for advancing the field and developing more efficient and effective optimization tools for a wide variety of complex applications.

6. Conclusions

Swarm-based algorithms are heuristic methods that have been applied to numerous optimization problems. Many such algorithms have been proposed over the years. They all use a population of solutions that are updated in an iterative process, differing fundamentally in the equations applied during the updating process.

This article analyzes 10 swarm-based algorithms (particle swarm optimization, shuffled-frog leaping algorithm, artificial bee colony, firefly algorithm, gravitational search, cuckoo search, bat algorithm, grey wolf optimization, chicken swarm optimization, and whale optimization). The operations performed by each algorithm have been described in detail, and the computational complexity of all of them has been analyzed. In addition to the parameters used by each method, the variables necessary to perform the operations have also been taken into account. The theoretical analysis has been complemented with experimental results obtained for a set of 20 objective functions commonly used in the literature.

It is observed that the computational complexity of most of the methods depends on the number of iterations (T), the size of the population (N), and the dimensions of the objective function of the problem to be solved (r). The first two factors can be defined before applying the algorithm, but the third is determined by the problem to be solved.

The objective function is a fundamental element, as it is used to calculate the quality or fitness of the solutions associated with the individuals in the population. In general, this calculation is performed on a vector of size r, but it has been shown that in some problems the calculation is not reduced to working on r elements but on many more. This calculation is performed many times in all algorithms and greatly influences the computational cost. Although it is possible to calculate the fitness whenever necessary, it is advisable to store its value for reuse, which reduces the execution time of the algorithms.

The computational complexity analysis carried out has separated the 10 algorithms into 4 groups. The first group includes PSO, BA, GWO, and WOA; the second includes ABC, CS, and CHO; the third includes FA and GSA; the fourth includes SFLA. The algorithms of the first group are those with the least computational complexity.

Taking into account the experimental results, we can consider that PSO and GWO are among the best methods if both the execution time and the obtained result are taken into account. In contrast, although GSA is time-consuming, it is not among the methods that generate the best results. Unlike GSA, SFLA is also time-consuming but generates very good results.

The analysis of the experimental results has allowed us to verify that some methods that have the same computational complexity have quite different execution times. This difference is justified when taking into account the specific operations performed by each algorithm.

The number of parameters of a method is another important issue, as they must be adjusted when applied to a specific problem. Therefore, when analyzing the results of the experimental tests carried out, it must be taken into account that they have been executed with a fixed set of values for all problems, so the results could improve if an adjustment were made to these parameters. It can be seen that GWO and ABC are the methods with the fewest parameters, while BA is the method with the most parameters.

In summary, when selecting a swarm algorithm to apply to a specific problem, it is advisable to take into account the computational complexity of the algorithm, how many parameters it uses, the memory necessary to store the variables, and also the specific operations it includes, since all these factors determine the efficiency of the method both in its execution time and in the result obtained.

In future research, it would be interesting to formally analyze the impact of different early termination criteria and parameter adaptation strategies on the computational complexity and the quality of the solutions obtained by swarm-based algorithms. Future designs of swarm-based algorithms could benefit from emphasizing enhancements such as parallelization and heuristic

simplification to better balance runtime efficiency with solution quality. Such strategies can make these algorithms more scalable and suitable for high-dimensional or real-time problems, marking a promising direction for future research. Furthermore, the empirical evaluation of trade-offs between speed and result quality in high-dimensional optimization problems and with complex search landscapes remains a relevant area of study. However, selecting the right trade-off is context-specific: time-sensitive applications may prioritize speed, while others can afford longer runtimes for better precision. This trade-off remains an important area for empirical and theoretical investigation.

Author contributions

M.L.P.D.: Writing–original draft, Conceptualization, Formal analysis, Methodology, Software, Validation; J.A.R.G: Writing–review and editing, Visualization.

Use of Generative-AI tools declaration

The authors declare they have not used Artificial Intelligence (AI) tools in the creation of this article.

Acknowledgments

This work was supported by the Ministry of Science and Innovation of Spain [Grant number POD2019-108883RB-C21].

Conflict of interest

The authors declare that there are no conflicts of interest.

References

1 L. Brezočnik, I. Fister, V. Podgorelec, Swarm intelligence algorithms for feature selection: a review, *Appl. Sci.*, **8** (2018), 1521. https://doi.org/10.3390/app8091521

2 A. E. Hassanien, E. Emary, *Swarm intelligence: principles, advances, and applications*, Boca Raton: CRC Press, 2018. https://doi.org/10.1201/9781315222455

3 Bilal, M. Pant, H. Zaheer, L. Garcia-Hernandez, A. Abraham, Differential evolution: a review of more than two decades of research, *Eng. Appl. Artif. Intel.*, **90** (2020), 103479. https://doi.org/10.1016/j.engappai.2020.103479

4 A. Chakraborty, A. K. Kar, Swarm intelligence: a review of algorithms, In: *Nature-inspired computing and optimization*, Cham: Springer, 2017, 475–494. https://doi.org/10.1007/978-3-319-50920-4_19

5 J. Tang, G. Liu, Q. Pan, A review on representative swarm intelligence algorithms for solving optimization problems: applications and trends, *IEEE/CAA J. Automatic. Sinica*, **8** (2021), 1627–1643. https://doi.org/10.1109/JAS.2021.1004129

6 T. Agarwal, V. Kumar, A systematic review on bat algorithm: theoretical foundation, variants, and applications, *Arch. Comput. Methods Eng.*, **29** (2022), 2707–2736. https://doi.org/10.1007/s11831-021-09673-9

7 A. G. Gad, Particle swarm optimization algorithm and its applications: a systematic review, *Arch. Comput. Methods Eng.*, **29** (2022), 2531–2561. https://doi.org/10.1007/s11831-021-09694-4

8 M. Guerrero-Luis, F. Valdez, O. Castillo, A review on the cuckoo search algorithm, In: *Fuzzy logic hybrid extensions of neural and optimization algorithms: theory and applications*, Cham: Springer, 2021, 113–124. https://doi.org/10.1007/978-3-030-68776-2_7

9 B. B. Maaroof, T. A. Rashid, J. M. Abdulla, B. A. Hassan, A. Alsadoon, M. Mohammadi, et al., Current studies and applications of shuffled frog leaping algorithm: a review, *Arch. Comput. Methods Eng.*, **29** (2022), 3459–3474. https://doi.org/10.1007/s11831-021-09707-2

10 S. N. Makhadmeh, M. A. Al-Betar, I. A. Doush, M. A. Awadallah, S. Kassaymeh, S. Mirjalili, et al., Recent advances in grey wolf optimizer, its versions and applications, *IEEE Access*, **12** (2023), 22991–23028. https://doi.org/10.1109/ACCESS.2023.3304889

11 J. Kennedy, R. Eberhart, Particle swarm optimization, In: *Proceedings of ICNN'95-international conference on neural networks*, Perth, WA, Australia, 1995, 1942–1948.

12 M. M. Eusuff, K. E. Lansey, Optimization of water distribution network design using the shuffled frog leaping algorithm, *J. Water Res. Plan. Man.*, **129** (2003), 210–225. https://doi.org/10.1061/(ASCE)0733-9496(2003)129:3(210)

13 D. Karaboga, An idea based on honey bee swarm for numerical optimization, Erciyes University, Engineering Faculty, Computer Engineering Department, 2005, Technical Report-TR06.

14 D. Karaboga, C. Ozturk, A novel clustering approach: artificial bee colony (ABC) algorithm, *Appl. Soft Comput.*, **11** (2011), 652–657. https://doi.org/10.1016/j.asoc.2009.12.025

15 X.-S. Yang, Firefly algorithms for multimodal optimization, In: *Stochastic algorithms: foundations and applications*, Heidelberg: Springer, 2009, 169–178. https://doi.org/10.1007/978-3-642-04944-6_14

16 E. Rashedi, H. Nezamabadi-Pour, S. Saryazdi, GSA: a gravitational search algorithm, *Inform. Sciences*, **179** (2009), 2232–2248. https://doi.org/10.1016/j.ins.2009.03.004

17 X.-S. Yang, S. Deb, Cuckoo search via Lévy flights, In: *2009 World congress on nature & biologically inspired computing (NaBIC)*, Coimbatore, India, 2009, 210–214. https://doi.org/10.1109/NABIC.2009.5393690

18 X.-S. Yang, A new metaheuristic bat-inspired algorithm, In: *Nature inspired cooperative strategies for optimization (NICSO 2010)*, Heidelberg: Springer, 2010, 65–74. https://doi.org/10.1007/978-3-642-12538-6_6

19 S. Mirjalili, S. M. Mirjalili, A. Lewis, Grey wolf optimizer, *Adv. Eng. Softw.*, **69** (2014), 46–61. https://doi.org/10.1016/j.advengsoft.2013.12.007

20 X. Meng, Y. Liu, X. Gao, H. Zhang, A new bio-inspired algorithm: chicken swarm optimization, In: *Advances in swarm intelligence*, Cham: Springer 2014, 86–94. https://doi.org/10.1007/978-3-319-11857-4_10

21 S. Mirjalili, A. Lewis, The whale optimization algorithm, *Adv. Eng. Softw.*, **95** (2016), 51–67. https://doi.org/10.1016/j.advengsoft.2016.01.008

22 X.-S. Yang, Cuckoo search algorithm – matlab implementation, 2020. Available from: `http://www.mathworks.in/matlabcentral/fileexchange/29809-cuckoo-search-cs-algorithm/content/cuckoo_search.m/`

23 R. N. Mantegna, Fast, accurate algorithm for numerical simulation of Lévy stable stochastic processes, *Phys. Rev. E*, **49** (1994), 4677. https://doi.org/10.1103/PhysRevE.49.4677

24 M.-L. Pérez-Delgado, The color quantization problem solved by swarm-based operations, *Appl. Intell.*, **49** (2019), 2482–2514. https://doi.org/10.1007/s10489-018-1389-6

25 M.-L. Pérez-Delgado, Color image quantization using the shuffled-frog leaping algorithm, *Eng. Appl. Artif. Intel.*, **79** (2019), 142–158. https://doi.org/10.1016/j.engappai.2019.01.002

26 M.-L. Pérez-Delgado, Color quantization with particle swarm optimization and artificial ants, *Soft Comput.*, **24** (2020), 4545–4573. https://doi.org/10.1007/s00500-019-04216-8

27 M.-L. Pérez-Delgado, M. A. Günen, A comparative study of evolutionary computation and swarm-based methods applied to color quantization, *Expert Syst. Appl.*, **231** (2023), 120666. https://doi.org/10.1016/j.eswa.2023.120666

28 M.-L. Pérez-Delgado, Recent applications of swarm-based algorithms to color quantization, In: *Recent advances on memetic algorithms and its applications in image processing*, Singapore: Springer, 2020, 93–118. https://doi.org/10.1007/978-981-15-1362-6_5

29 D. E. Knuth, Sorting by merging, In: *The art of computer programming: sorting and searching*, Addison-Wesley, 1998, 158–168.

30 A. Lipowski, D. Lipowska, Roulette-wheel selection via stochastic acceptance, *Physica A*, **391** (2012), 2193–2196. https://doi.org/10.1016/j.physa.2011.12.004

31 M. Jamil, X.-S. Yang, A literature survey of benchmark functions for global optimisation problems, *International Journal of Mathematical Modelling and Numerical Optimisation*, **4** (2013), 150–194. https://doi.org/10.1504/IJMMNO.2013.055204

AIMS Press