Symbols

Table 1: Some symbols in this work.

Symbol	Meaning
$\mathcal{U}(a,b)$	A single random uniform distributed number in $[a,b)$
r	A single random uniform distributed number in $[0,1)$. It is always randomly generated when utilized.
$Exp(a)$	A single random exponential distributed number with the scale $\lambda = a$
$\mathbf{Min}(\{a_i\}) \to a_i$	Minimization calculator that will generate one a_i that has the minimum fitness value amongst the set $\{a_i\}$.
$\mathbf{Sort}(\{a_i\}) \to \langle a_i \rangle$	Sort calculator that will generate an ordered sequence $\langle a_i \rangle$ of the ascending fitness values of a_i amongst the set $\{a_i\}$.
$\mathbf{Round}(a) \to a^{'}$	Rounding calculator that will round a value a to its nearest integer $a^{'}$.
$\mathbf{Dist}(a,b) \to D_{a,b}$	Distance calculator that will calculate the Euclidean distance by $D_{a,b} = \sqrt{(a-b)^2}$.
$\lvert\ \rvert$	Only denotes absolute value.

Chapter 1

Introduction

1.1 Motivation

The pace of designing optimization algorithms has never been stopped even since antiquity [1]. In the late stage of the 1900s, heuristic algorithms began to stand out with their impressive performance on more complicated modern optimization problems [28]. In these heuristic research fields, due to the great success of the simulated annealing optimization algorithm [28], the analogy connection between the nature and optimization methods attracts increasing attention.

The nature evolution process, as the most successful analogy [28], inspired most of the metaphor-based optimization algorithms until now. From classical genetic algorithms, particle swarm optimization to modern 'animal'-inspired optimization algorithms, from pure algorithms to their various variants, from independent algorithms to combination algorithms, these algorithms that have come or are coming are welcome to solve problems and enrich the research community, however, one issue in this researching field becomes increasingly obvious: **a great number of new nature-inspired optimization algorithms are just old heuristic algorithms in new clothes, which might already mislead the development of heuristic algorithms** [11, 12, 24, 28].

This problem frequently appears in population-based algorithms that are also well-known as swarm-based algorithms. One paper mentioned that these new modern nature-inspired algorithms are actually similar to swarm intelligence [11], in which they theoretically displayed the common components between the swarm intelligence and the nature-inspired algorithm. Some researchers also mentioned that several core components comprised

most of these metaheuristic algorithms [12], in which they provided a generalized flow chart to display these common components. Moreover, some papers pointed out that a unified representation for organizing these algorithms together, especially a mathematical representation, will help prevent a worse development of heuristic algorithms [29, 35].

Therefore, in this work, we will propose a unified framework, with primitive math knowledge, for a set of swarm-based optimization algorithms. Before proposing this unified framework, there are two principles in this work:

- *Variants of algorithms aren't considered in this work.*
  Variants are important improvements to the algorithms, however, these variants will be ignored when building up the unified framework in this work. This is because it is widely accepted that the base algorithm is the foundation of its variants, and if the foundation is suffering from issues, the variant built upon the foundation is also not solid. Moreover, theoretically speaking, any theory and any practice derived from the base can be applied to its variants, this is because the base and its variants share the same algorithm structure. Therefore, variants can be safely ignored here, and we will only focus on the base algorithms
- *Hybrid algorithms aren't considered in this work.*
  On the one side, independent studies should be the starting of the larger study area, on the other side, it must be accepted that it is very challenging to discuss independent and hybrid algorithms together. Therefore, hybrid algorithms will be also ignored here.

To build up such a unified framework, inspired by gene terminology and operations in the Evolutionary algorithm group (EA), this work will firstly build up a generic dictionary (like gene terminology and operations) in which **different expressions with the same information will be re-defined as a generic expression (see ??)**. After analyzing these algorithms on the level of components, the unified framework seems to come out by itself. The unified framework will be built upon the level of optimization progress that is **the algorithms' progresses are also able to be re-defined in a unified framework only with primitive math knowledge (see ??)**.

Therefore, for building a unified environment to challenge these chaotic new swarm-based algorithms, the work will firstly provide a generic language with which these chaotic algorithms can be discussed on the same component level in the **??**. Next, the unified framework will be proposed which can simultaneously cover these chaotic algorithms in the **??**. Furthermore, we will prove the feasibility of this unified framework by benchmark experiments in the **??** and **??**. Lastly in the Chapter 6, we will conclude our findings and give future research directions.

1.2 Overview

As the Figure 1.1 displayed, since unifying the entire nature-inspired algorithms is impossible in one work, we will start from working on seven examples, including new modern and old classical examples, after proving the feasibility of the unified framework we will build, we will discuss the possibility of this framework is also able to be employed in much more other nature-inspired algorithms.

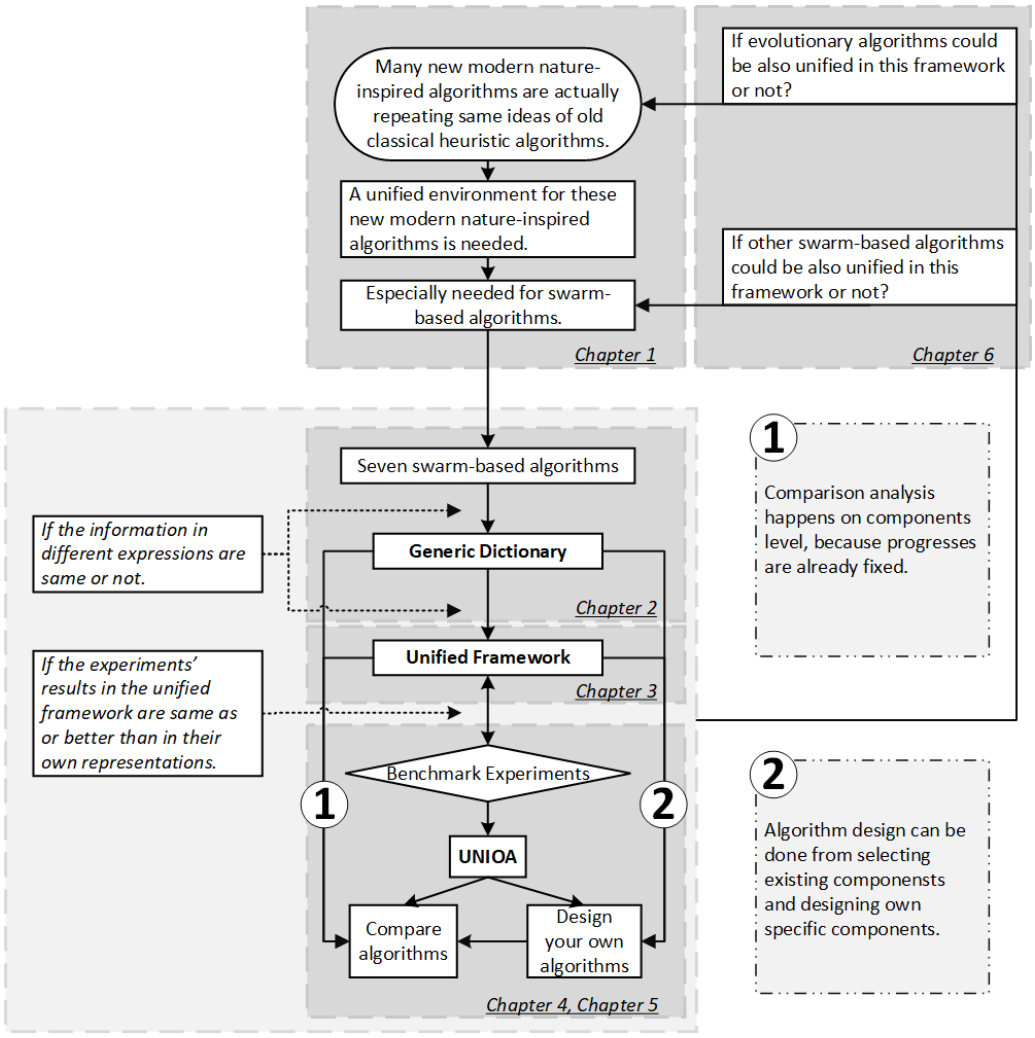

Figure 1.1: Overview.

Meanwhile, these seven examples will be selected from the Bestiary of evolutionary computation[1]. They are :

- New modern swarm-based algorithms:
  Bat-inspired algorithm [34], Grasshopper optimization algorithm [27], Crow search algorithm [2], Moth-flame optimization algorithm, Monarch butterfly optimization algorithm [31], Butterfly optimization algorithm [3].
- Old classical swarm-based algorithms:
  Particle swarm optimization [17]

This is because we hope the generic dictionary in the **??** and the unified framework in the **??** are unified enough to cover both traditional and modern nature-inspired algorithms.

[1] https://github.com/fcampelo/EC-Bestiary

Chapter 2

Unified Terminologies

2.1 Motivation

This chapter discusses the commonalities among seven selected nature-inspired algorithms on the level of their components. Specifically, in these algorithms, various terminologies of components with their meanings are categorized into unified terminologies with unified definitions.

Firstly, we want to introduce several common symbols in nature-inspired algorithms. In the nature-inspired algorithm researching field, [1] the given objective optimization problem is formulated as a math formula $f(\vec{x})$ in which every possible solution is represented as a vector $\vec{x}$. Moreover, the number of independent elements in $\vec{x}$ is usually defined as the dimension of search space for the objective optimization problem. When seeking the optimal solution to the optimization problem, algorithms will first initialize an arbitrary set of $\vec{x}$ and then optimize this set by using iterative strategies. Same as in most studies, we also denote $\vec{x}$ as $\mathbf{x}$ and call it the objective solution. Inspired by this representation, we define that any term $\mathbf{a}$ in the bold style denotes the item that is also a vector with the same dimension as the objective solution $\mathbf{x}$. Furthermore, we want to clarify that if the bold style item $\mathbf{a}$ has a subscript i, it means this item has a one-to-one correspondence with each individual $\mathbf{x}_i$. However, one item without this subscript will apply to the whole population, whether this item is in bold style or not in bold style.

Moreover, usually, the set of $\vec{x}_i$ is called the entire population, and the number of $\vec{x}_i$ in this set is called the size of the entire population. Each $\vec{x}_i$ is a possible optimal solution

[1] In our work, we only consider single-objective continuous problem optimization.

to the target optimization problem $f()$, therefore the value of $f(\vec{x})$ is called the fitness value of $\vec{x}$.

In the following sections, Section 2.2 introduces these seven algorithms in detail but doesn't include how natures are simulated to create mathematical models. Readers can find such approaches in their original published papers. In this work, the description of each algorithm is straightforward, only including the final model and its default hyper-parameter settings. Meanwhile, we also summarize what is special in each algorithm. Section 2.3 will give and discuss our unified terminologies based on these seven selected algorithms. An overview summary about this chapter is displayed in Section 2.4.

2.2 Specific Swarm-based Algorithms

2.2.1 Bat-inspired Algorithm (BA)

Description When simulating the echolocation behavior of micro-bats, for simplicity, the Bat-inspired algorithm only considered the velocity $\mathbf{v}_i$ of each bat $\mathbf{x}_i$, the echo frequency $freq$ emitted by each bat $\mathbf{x}_i$, the rate r of echo pulse, and the loudness A of echo [34]. The algorithm described here follows the published implementation code [2], and explanations with math formulas are from their published paper [34].

This algorithm starts from is (1) initializing a bat $\mathbf{x}_i$ population with size $n = 20$ in the d dimension environment and the initialization method is Eq.2.1 in which Lb/Ub is lower/upper boundary of each element in $\mathbf{x}_i$. Then, (2) velocity $\mathbf{v}_i$ of each bat $\mathbf{x}_i$ is initialized as Eq.2.2.

$$\mathbf{x}_i = \boldsymbol{\mathcal{U}}(Lb, Ub), i = 1, 2, \cdots, n \tag{2.1}$$

$$\mathbf{v}_i = \boldsymbol{\mathcal{U}}(0, 0), i = 1, 2, \cdots, n \tag{2.2}$$

After (3) evaluating fitness of each $\mathbf{x}_i$, it (4) goes to find the best individual $\mathbf{x}_*$ that is the best $\mathbf{x}_i$ the whole population has found so far. Next is (5) iteratively optimizing process in which the maximum number of iterations is $t\_max = 1000$. Under the iterative process, it firstly (6) updates A as Eq.2.3 in which $A(t = 0) = 1$ is the initial value of loundness, and $\alpha = 0.97$ is a decreasing hyper-parameter. Next, (7) r is updated as Eq.2.4 in which $r_0 = 1$ is the initial value of pulse rate and $\gamma = 0.1$ is a decreasing hyper-parameter. In this algorithm, t is the iteration counter.

$$A(t + 1) = \alpha \times A(t) \tag{2.3}$$

[2] https://uk.mathworks.com/matlabcentral/fileexchange/74768-the-standard-bat-algorithm-ba?s_tid=prof_contriblnk

$$r(t+1) = r_0 \times (1 - e^{-\gamma \times t}) \tag{2.4}$$

Next is (8) updating each $\mathbf{x}_i$ as Eq.2.5 in which $freq\_min = 0 / freq\_max = 2$ is the lower/upper boundary of $freq_i$.

$$\begin{aligned} freq_i &= \mathcal{U}(freq\_min, freq\_max) \\ \mathbf{v}_i(t+1) &= \mathbf{v}_i(t) + (\mathbf{x}_i(t) - \mathbf{x}_*) \times freq_i \\ \mathbf{x}_i(t+1) &= \mathbf{x}_i(t) + \mathbf{v}_i(t+1) \end{aligned} \tag{2.5}$$

If (9) $rand < r(t+1)$, $\mathbf{x}_i$ will be further updated as Eq.2.6 in which $\epsilon = 0.1$.

$$\mathbf{x}_i(t+1) = \mathbf{x}_*(t) + \epsilon \times \mathbf{rand} \times A(t+1) \tag{2.6}$$

After (10) fixing outliers in $\mathbf{x}_i(t+1)$ by replacing them with the boundary value, the BA model evaluates fitness of each $\mathbf{x}_i(t+1)$ again. Then (11) if the fitness of $\mathbf{x}_i(t+1)$ is better than $\mathbf{x}_i(t)$ or the situation is $rand > A$, $\mathbf{x}_i(t+1)$ will be accepted. Lastly (12) the $\mathbf{x}_*$ is updated.

Until now, one optimization round (from step (6) to step(12)) is finished, and it will iteratively execute until the stop condition meets.

Summary When modeling a Bat algorithm, there are two main processes: Initialization and Optimization. In the initialization process, besides initializing an initial population and all default parameters (including hyper-parameters and initial values), an initial velocity population is also initialized. Meanwhile, the initialization process also calculates the initial best bat. In the optimization process, two dynamically changing parameters are updated before updating the population. It is worth mentioning that the whole process of updating the population is asynchronous, including asynchronously evaluating the fitness of each bat, asynchronously judging whether the updated bat is improved, and asynchronously updating the current best bat.

Moreover, we want to clarify that in their implemented code, it is unreasonable to compare the previous best bat with a current unaccepted bat when updating the current best bat. The definition of the current best bat is finding the best bat in the current population. The unaccepted bat is not in the current population anymore. Therefore, it is more reasonable to calculate the current best bat by comparing the previous best bat with the current bat who is at this same position in the current population.

Furthermore, we want to point out that in the Initialization process, the evaluation of the population happens immediately after initializing the initial population. However, in the Optimization process, the evaluation happens immediately after generating a temporary

population. In other words, it happens in the middle of obtaining the temporary population and judging if accepting the generated individuals as the updated individuals, which means the evaluation does not happen on the final new updated population, but on a temporary updated population.

2.2.2 Grasshopper Optimization Algorithm (GOA)

Description The Grasshopper Optimization Algorithm was proposed in 2017 [27]. They designed this algorithm by studying grasshoppers' main swarming behaviors in larval and adulthood phases, such as slow movement and small steps in the larval phase, abrupt movement and long-range steps in the adulthood phase, and food source seeking in both phases. Their final model described here is from their published code [3], and explanations with math formulas are from their published paper [27].

This model starts at (1) initializing grasshoppers $\mathbf{x}_i$ with size $N = 100$ in the d dimension environment. The initialization method in their published code can be simply re-framed as Eq.2.7 in which $up/down$ is the upper/down boundary of $\mathbf{x}_i$ [4].

$$\mathbf{x}_i = \boldsymbol{\mathcal{U}}(down, up), i = 1, 2, \cdots, N \tag{2.7}$$

Next is (2) calculating the fitness of each individual $\mathbf{x}_i$ and (3) finding the best individual $\mathbf{T}$ that the whole population has found so far. The followings are iterative optimizing process in which the stop condition is current number of iteration l is smaller than the maximum number of iterations $max\_iteration = 100$ [5]. Under the iterative process, (4) the hyper-parameter c is firstly updated as Eq.2.8 in which $cMin = 0.00004/cMax = 1$ is the lower/upper boundary of c.

$$c = cMax - l \times \frac{cMax - cMin}{max\_iteration} \tag{2.8}$$

Then, the GOA model (5) normalizes the distances $\widetilde{d}$ between individuals into $[1, 4]$. The normalization method in their published code is Eq.2.9 in which d is the Euclidean distance [6] between individual pairs.

$$\widetilde{d} = 2 + d \bmod 2 \tag{2.9}$$

[3] https://seyedalimirjalili.com/goa

[4] Their published implementation code considered two boundary cases: if elements in $\mathbf{x}_i$ share the same boundary or if elements in $\mathbf{x}_i$ have different boundaries. According to the further experiments in the Chapter 5, we only consider the first case.

[5] Their published code thought the current iteration $l = 2$ when starting optimizing, however, we define the current iteration $l = 1$ when starting optimization in order to be same as other algorithms.

[6] It refers to their faster version code.

Next is (6) updating $\mathbf{x}_i$ as Eq.2.10 that is a simplified version of their published original formula. Because we only consider one case that elements in $\mathbf{x}_i$ share the same boundary, we replace ub_d/lb_d by ub/lb. We also replace x_i^d by $\mathbf{x}_i$ and $\widehat{T_d}$ by $\mathbf{T}$ in the view of vector [7].

$$\mathbf{x}_i = c \times \left(\sum_{j=1, j \neq i}^{N} c \times \frac{ub - lb}{2} \times S\left(\widetilde{d}_{i,j}\right) \times \frac{\mathbf{x}_j - \mathbf{x}_i}{d_{ij}} \right) + \mathbf{T} \tag{2.10}$$

The method of updating Eq.2.10 has a function named S in which $\widetilde{d}_{i,j}$ is the input. The S function is formulated as Eq.2.11 in which $f = 0.5$ and $l = 1.5$ are two hyper-parameters [8].

$$S(\widetilde{d}_{i,j}) = f e^{\frac{-\widetilde{d}_{i,j}}{l}} - e^{-\widetilde{d}_{i,j}} \tag{2.11}$$

After updating each $\mathbf{x}_i$, (7) outliers in $\mathbf{x}_i$ are replaced by the boundary values before (8) calculating its fitness. The last step is (9) updating the best individual $\mathbf{T}$.

Until now, one optimization round (from step(4) to step(9)) is finished, and it will iteratively execute until the stop condition meets.

Summary When modeling a GOA algorithm, there are two main processes: Initialization process and Optimization process. In the Initialization process, the algorithm has only two main tasks: initializing the initial population and finding the global best individual in the initial population. In the Optimization process, the update strategy depends on neighbors' positions.

2.2.3 Crow Search Algorithm (CSA)

Description The crow search algorithm was designed [2] in 2016. The core metaphor is crows' behaviour of hiding excess food and retrieving this stored food when needed. The author believes that these behaviours, including stealing others' food by tailing after them and fighting with theft by moving to another place instead of their real hiding place, are similar to an optimization process [2]. Their final algorithm model described here is from their published implementation code [9], and explanations with math formulas are from their published paper [2].

[7] Their published paper didn't point out whether $\widehat{\ }$ is a special operation or not.

[8] Please note, here the l is not the current iteration mentioned above.

[9] https://nl.mathworks.com/matlabcentral/fileexchange/56127-crow-search-algorithm

The algorithm starts at (1) initializing the crow population of size $N = 20$ in the d dimension environment. The initialization method they used in their implementation code can be formulated as Eq.2.12 in which l/u is the lower/upper boundary.

$$\mathbf{x}_i = \boldsymbol{\mathcal{U}}(l, u), i = 1, 2, \cdots, N \tag{2.12}$$

The next is (2) calculating the fitness of each individual $\mathbf{x}_i$, and (3) initializing the memory $\mathbf{m}_i$ for each individual $\mathbf{x}_i$ as Eq.2.13.

$$\mathbf{m}_i = \mathbf{x}_i \tag{2.13}$$

The following steps are about the iterative optimization process in which the stop condition is the maximum number of iterations $tmax = 5000$. Under the iterative process, (4) each individual $\mathbf{x}_i$ is updated as Eq.2.14 in which $fl = 2$ and $AP = 0.1$ are two hyper-parameters.

$$\mathbf{x}_i(t+1) = \begin{cases} \mathbf{x}_i(t) + rand \times fl \times (\mathbf{m}_j(t) - \mathbf{x}_i(t)) & , \quad rand > AP \\ \boldsymbol{\mathcal{U}}(u, l) & , \quad \text{o.w} \end{cases} \tag{2.14}$$

After (5) calculating the fitness of the new individual $\mathbf{x}_i(t+1)$, if (6) the new individual $\mathbf{x}_i(t+1)$ is in the case of $l \leq \mathbf{x}_{i,d}(t+1) \leq u$, the new individual $\mathbf{x}_i(t+1)$ will be accepted, if not, the individual $\mathbf{x}_i(t)$ will keep unchanged and reject to update. Lastly, (7) the memory $\mathbf{m}_i(t+1)$ will be updated as Eq.2.15.

$$\mathbf{m}_i(t+1) = \begin{cases} \mathbf{x}_i(t+1) & , \quad f(\mathbf{x}_i(t+1)) < f(\mathbf{m}_i(t)) \\ \mathbf{m}_i(t) & , \quad \text{o.w} \end{cases} \tag{2.15}$$

Until now, one optimization round (step (4) to step (7)) is finished, and it will iteratively execute until the stop condition meets.

Summary When modeling a CSA algorithm, there are two main processes: Initialization process and Optimization process. In the Initialization process, the algorithm has only two main tasks: initializing the initial population and the initial memory population that is the same as the initial population. In the Optimization process, the memory position of neighbors determines how far the individual moves.

When meeting outliers, the CSA algorithm chooses to give up the new position. In other words, as far as there is an outlier, the individual rejects to be updated.

2.2.4 Moth-flame Optimization Algorithm (MFO)

Description In 2015, the Moth-flame optimization algorithm simulated the process that moths will fly to the center of artificial light (that is also seen as flames) in the spiral path [22]. The core characteristic in MFO is that the author defined the positions of flames as the optimization orientation of moths, and each moth has its own dynamically changing flame. The final algorithm model described here is from their published implementation code [10], and explanations with math formulas are from their published paper [22].

This algorithm starts from (1) $N = 30$ initial moths population $Moth\_pos$ in a dim dimension environment. The initialization method in their published code can be formulated as Eq.2.16 in which $\mathbf{m}_i$ is each moth and ub/lb is the upper/lower boundary of elements in each moth [11].

$$\mathbf{m}_i = \boldsymbol{\mathcal{U}}(lb, ub), i = 1, 2, \cdots, N \tag{2.16}$$

Then the algorithm model starts iteratively optimizing each $\mathbf{m}_i$ in which the stop condition is there is a maximum number of iterations T. Under the iterative optimization process, first of all, (2) the number of flames is calculated as Eq.2.17 in which l is the current number of iteration. The $flame\_no$ will impact the updating positions of moths in the following steps.

$$flame\_no = \mathbf{Round}(N - l \times \frac{N-1}{T}) \tag{2.17}$$

After (3) replacing outliers in $\mathbf{x}_i$ by the boundary values, fitness of each moth is calculated. Next is (4) sorting the combination of the current population and the previous population [12] based on the ascending of their fitness, and the first $N = 30$ moths in the combination population are defined as $sorted\_population$. Then, a linearly decreasing parameter a is calculated by Eq.2.18 in which l is the current number of iteration.

$$a = -1 + l \times \frac{-1}{T} \tag{2.18}$$

Next, (5) each element j of each moth $\mathbf{m}_i$ will be updated as follows: if $i \leqslant flame\_no$, Eq.2.19; if not, Eq.2.20 in which $\mathbf{m}'_{i,j}$ denotes each newly updated element j of each moth $\mathbf{m}_i$; $b = 1$ is a constant parameter used to define the shape of spiral [22]; $t =$

[10] https://nl.mathworks.com/matlabcentral/fileexchange/52270-moth-flame-optimization-mfo-algorithm-toolbox?s_tid=srchtitle

[11] In this work, we only consider that the boundaries of all elements in each moth are same, although their published code also provided a method to deal with different boundaries.

[12] The previous population is $\emptyset$ (there is no previous population), when the current population is the initial population.

$(a-1) \times rand + 1$ is a random number related to a.

$$\begin{aligned} distance\_to\_flame &= |sorted\_population(i,j) - \mathbf{m}_{i,j}| \\ \mathbf{m}_{i,j}^{'} &= distance\_to\_flame \times e^{b \times t} \times \cos(t \times 2 \times \pi) + sorted\_population(i,j) \end{aligned} \tag{2.19}$$

$$\begin{aligned} distance\_to\_flame &= |sorted\_population(i,j) - \mathbf{m}_{i,j}| \\ \mathbf{m}_{i,j}^{'} &= distance\_to\_flame \times e^{b \times t} \times \cos(t \times 2 \times \pi) + sorted\_population(flame\_no,j) \end{aligned} \tag{2.20}$$

Until now, one optimization round (from step(2) to step(5)) is finished, and it will iteratively execute until the stop condition meets.

Summary When modeling a Moth-flame algorithm, there are two processes: Initialization process and Optimization process. In the initialization process, this algorithm only initializes an initial population with all default parameters. In the optimization process, the main convergency mechanism is to keep moving to better solutions by narrowing better choices down. Here, the better choices are selected from the top several best solutions, and the method of narrowing down is to narrow the number of best solutions. The main convergency mechanism happens at Eq.2.20.

Compared with other algorithms, this algorithm has another two different places. The first one is that there is no evaluation operator in the initialization process. The second one is that fixing outliers and evaluating are happening at the beginning of the optimization process.

2.2.5 Monarch Butterfly Optimization (MBO)

Description The Monarch Butterfly Optimization simulated the migration behavior of monarch butterflies in 2015 [31]. The core metaphor is that monarch butterflies in two different habitats evolve differently. These two habitats are mimicked by dividing the whole population into two subpopulations. The author defined that the whole population will firstly be sorted based on their fitness before dividing them into two with a ratio [31]. Then, each habitat experiences its particular evolution approach (migration operator or adjusting operator [31]). We describe their final algorithm model implemented in their published code [13], and explanations with math expressions from their published paper [31].

[13] https://nl.mathworks.com/matlabcentral/fileexchange/101400-monarch-butterfly-optimization-mbo?s_tid=srchtitle

This algorithm starts from (1) an initial population $Population$ with size $M = 50$, and the initialization method [14] in their published code can be formulated as Eq.2.21 in which $MinParValue/MaxParValue$ is the lower/upper boundary of elements in $\mathbf{x}_i$.

$$\mathbf{x}_i = \boldsymbol{\mathcal{U}}(MinParValue, MaxParValue), i = 1, 2, \cdots, M \tag{2.21}$$

After (2) evaluating each $\mathbf{x}_i$, the entire population is sorted from the best to the worst. Then (3) it starts iteratively optimizing each $\mathbf{x}_i$ in which the stop condition is the maximum number of iterations $Maxgen = 50$. Under the iterative process, (4) $Keep = 2$ best $\mathbf{x}_i$ are first kept in $chromKeep$ [15]. Next is (5) dividing the entire $Population$ into two sub-populations with a ratio $partition = \frac{5}{12}$. The two sub-populations ($Population1$, $Population2$) will be updated by two operators ('Migration operator', 'Adjusting operator') respectively. (6) In the $Population1$, if $rand \times period \leqslant partition$ in which $period = 1.2$, $\mathbf{x}_{i,k}^{t+1} = \mathbf{x}_{population1,k}^{t}$; else $\mathbf{x}_{i,k}^{t+1} = \mathbf{x}_{population2,k}^{t}$, in which k is the $k-$th element of $\mathbf{x}_i$, and $population1/population2$ is one arbitrary moth from previous $Population1/Population2$. (7) Next is fixing outliers in this newly generated $Population1$ by replacing them with the boundary value, before evaluating this newly generated $Population1$.

(8) In $Population2$, if $rand \leqslant partition$, $\mathbf{x}_{i,k}^{t+1} = \mathbf{x}_{best,k}^{t}$ in which k is the $k-$th element and $best$ is the best moth the entire population has found so far; else $\mathbf{x}_{i,k}^{t+1} = \mathbf{x}_{population2,k}^{t}$, in which k is the $k-$th element, and $population2$ is one arbitrary moth from previous $Population2$, furthermore, under this situation, if $rand > BAR$ in which $BAR = \frac{5}{12}$, $\mathbf{x}_{i,k}^{t+1}$ will be further updated as Eq.2.22 in which α is a weighting factor as Eq.2.23, and $\mathrm{d}\mathbf{x}$ is a walk step as Eq.2.24. (9) Then, outliers in the newly generated $Population2$ are fixed by replacing them with the boundary value, and next is evaluating this newly generated $Population2$.

$$\mathbf{x}_{i,k}^{t+1} = \mathbf{x}_{i,k}^{t+1} + \alpha \times (\mathrm{d}\mathbf{x}_k - 0.5) \tag{2.22}$$

$$\alpha = \frac{S_{max}}{t^2} \text{ where } S_{max} = 1 \text{ and } t \text{ is the current iteration.} \tag{2.23}$$

$$\mathrm{d}\mathbf{x} = \mathbf{Levy}(\mathbf{x}_j^t) \tag{2.24}$$

where $\mathbf{Levy}$ function in their published code can be formulated as Eq.2.25.

$$\begin{aligned} Stepsize &\sim Exp(2 \times Maxgen) \\ \mathrm{d}\mathbf{x}_k &= \sum(\underbrace{\tan(\pi \times rand), \cdots, \tan(\pi \times rand)}_{Stepsize}) \end{aligned} \tag{2.25}$$

[14] The original paper considered the situation if boundaries of elements are same or not, but in this work, we only consider the situation that boundaries of elements are same

[15] This elitism strategy stated in their published code wasn't stated in their publish paper.

Then (10) these two newly generated sub-populations are combined into one population. Next, (11) after the last $Keep = 2$ worst $\mathbf{x}_i$ will be replaced by $chromKeep$ that is mentioned in step (4), (12) the final new population will be sorted again.

Until now, one optimization round (from step (4) to step(12)) is finished, and it will iteratively execute until the stop condition meets.

Summary When modeling an MBO algorithm, there are two main processes: Initialization and Optimization. In the initialization process, besides initializing an initial population and all default parameters, the initial population is also sorted from best to worst. In the optimization process, the core idea is that the whole population is divided into two sub-populations, and they are updated differently. One sub-population is better than another one because of the previous sort operator. The elitism strategy is also a highlight of this model. It happens at the beginning and the end of the optimization process.

Furthermore, although in the initialization process, the evaluation of the population only happens once immediately after initializing the initial population, there are two evaluations in the optimization process. There are two evaluations because the whole population is updated in two methods for two sub-populations.

2.2.6 Butterfly Optimization Algorithm (BOA)

Description The Butterfly Optimization Algorithm was modeled in 2018 [3]. Butterflies' behaviors of searching for food and mates are mimicked to solve optimization problems. The algorithm model described here is from their published code [16], and explanations with math formulas are from their published paper [3].

In this algorithm, they define a dim dimensional environment in which each individual is $\mathbf{x}_i$ and the fitness of $\mathbf{x}_i$ is $f(\mathbf{x}_i)$. (1) The algorithm first initializes a population $n = 50$ of $\mathbf{x}_i$ by Eq.2.26 in which Lb/Ub is lower/upper bound [17].

$$\mathbf{x}_i = \boldsymbol{\mathcal{U}}(Lb, Ub), i = 1, 2, \cdots, n \tag{2.26}$$

[16] https://www.mathworks.com/matlabcentral/mlc-downloads/downloads/b4a529ac-c709-4752-8ae1-1d172b8968fc/67a434dc-8224-4f4e-a835-bc92c4630a73/previews/BOA.m/index.html

[17] The initialization method isn't shown in their published paper or code, but according to their published code that only said there was an 'initialization' function receiving n, dim, Ub and Lb as inputs. We can reasonably assume their initialization method is Eq.2.26.

It also (2) setups three parameters: probability switch $p = 0.8$, $power\_exponent = 0.1$ and $sensory\_modality = 0.01$. After evaluating each $\mathbf{x}_i$, it (3) calculates the best individual g^* that the whole population had found so far.

Next steps are the iterative optimization process in which the stop condition is the current iteration t is smaller than the maximum number of iterations N_iter. Under the iterative process, for each $\mathbf{x}_i$, (4) its own fragrance FP_i is first calculated as Eq.2.27 in which $f(\mathbf{x}_i)$ is the fitness value of $\mathbf{x}_i$ [18].

$$FP_i(t) = sensory\_modality \times f(\mathbf{x}_i(t))^{power\_exponent} \tag{2.27}$$

Then (5) if $rand > p$, the $\mathbf{x}_i$ will be updated as Eq.2.28, (6) if not, the $\mathbf{x}_i$ will be updated as Eq.2.29 in which $\mathbf{x}_j$ and $\mathbf{x}_k$ are two arbitrary neighbors around $\mathbf{x}_i$.

$$\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + FP_i(t) \times (rand^2 \times g^*(t) - \mathbf{x}_i(t)) \tag{2.28}$$

$$\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + FP_i(t) \times (rand^2 \times \mathbf{x}_j(t) - \mathbf{x}_k(t)) \tag{2.29}$$

Then after (7) replacing the outliers by the boundary values, (8) only if the updated $\mathbf{x}_i(t+1)$ is better than the previous $\mathbf{x}_i(t)$, the update $\mathbf{x}_i(t+1)$ will be deliveried into the next optimization round. Next steps are (9) updating g^* and (10) updating $sensory\_modality$ using Eq.2.30.

$$sensory\_modality(t+1) = sensory\_modality(t) + \frac{0.025}{sensory\_modality(t) \times N\_iter} \tag{2.30}$$

Until now, one optimization round (step(4) to step(10)) is finished, it will iteratively execute until the stop condition meets.

Summary When modeling a BOA algorithm, there are two processes: Initialization process and Optimization process. In the Initialization process, there are four tasks: initialize the initial population and evaluate them, find the global best one individual, and set up several hyper-parameters. In the optimization process, the most interesting part is the fitness joins in optimizing the population. Moreover, the square value of the random number is also a special place.

Meanwhile, the original implemented code also has the same issue as the BA algorithm (see Subsection 2.2.1) when updating the global best one individual. We think it is unreasonable to compare the previous best individual with an unaccepted one when

[18] Their published code calculated the fitness twice here. However, the published paper did not mention whether this is a necessary operator. Therefore, we ignore this duplicated evaluation when studying this algorithm.

updating the current global best individual. The definition of the current best global individual shall be the best one that the current population has found so far. Here, the current population shall be the population that has been accepted and will be delivered into the next optimization round. Therefore, it is more reasonable to calculate the current global best one by comparing the previous best one with the current accepted best one at this same position in the current population.

2.2.7 Particle Swarm Optimization (PSO)

Description Particle Swarm Optimization was firstly introduced in 1995 [17]. The main idea comes from the movements of the animal swarm. By simulating their behaviors that the swarm dynamically moves to a 'roost' [17], the extremely simple algorithm, even with a small swarm size (15 to 30 agents), boasted impressive performance on solving continuous optimization functions. In the very first paper, the position of each agent in its swarm was controlled by two velocities $\mathbf{X}$ and $\mathbf{Y}$. Moreover, each agent can remember its best position and know the global best position in its swarm. Although the very first paper also provided the algorithm model with formulas, we preferred to reference clearer published pseudocode whose algorithm model is as same as the very first paper possible. The final PSO model described in this work is from a tutorial of this method [21], and the hyper-parameter settings are from an application of this method [5].

This algorithm starts (1) from an initial swarm $\mathbf{x}_i$ with the size $N = 25$. The initialization method is formulated as Eq.2.31 in which min/max is the lower/upper boundary of $\mathbf{x}_i$.

$$\mathbf{x}_i(t=0) = \boldsymbol{\mathcal{U}}(min, max), i = 1, 2, \cdots, N \tag{2.31}$$

Then (2) each velocity $\mathbf{v}_i$ is initialized as Eq.2.32 in which $lb_{\mathbf{v}}/ub_{\mathbf{v}}$ is the boundary of velocity. Next (3) each $pbest_i$ is calculated as Eq.2.33.

$$\mathbf{v}_i = \boldsymbol{\mathcal{U}}(lb_{\mathbf{v}}, ub_{\mathbf{v}}), i = 1, 2, \cdots, N \tag{2.32}$$

$$pbest_i = \mathbf{x}_i, i = 1, 2, \cdots, N \tag{2.33}$$

After (4) calculating the fitness of each agent $\mathbf{x}_i$, the algorithm obtains (5) the best agent $gbest$ that is the whole swarm has found so far in the initial swarm. The following steps are about (6) iterative optimization process with a stopping condition. Under the iterative process, the published pseudo-code first updates each velocity $\mathbf{v}_i$ by Eq.2.34 in which $w = 0.73$ is an adjusting parameter, $c_1 = 1.49$ and $c_2 = 1.49$ is respectively the cognitive and social coefficient [21].

$$\mathbf{v}_i(t+1) = w \times \mathbf{v}_i(t) + c_1 \times rand \times (pbest_i - \mathbf{x}_i) + c_2 \times rand \times (gbest - \mathbf{x}_i) \tag{2.34}$$

After updating (7) each agent as Eq.2.35, all elements in each new agent are checked if they are feasible. The last step in one optimization round is (8) updating $pbest$ and (9) $gbest$.

$$\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + \mathbf{v}_i(t+1) \tag{2.35}$$

Until now, one optimization round (from step(6) to step(9)) is finished, and it will iteratively execute until the stop condition meets.

As a side note, the very first published paper did not directly point out the notation for searching space dimension, the stop condition, and if outliers should be dealt with or not. However, as discussed in the Section 2.1, the dimension depends on the number of elements in $\mathbf{x}_i$, therefore, it is reasonable to accept that the PSO model also needs a concept of dimension. Meanwhile, the stop condition and the method to deal with the outliers also must exist. In this work, the method of fixing outliers is the most common way that replaces outliers by boundary values of $\mathbf{x}_i$.

Summary When modeling a PSO algorithm, there are two processes: Initialization process and Optimization process. In the Initialization process, four components need to be initialized: the initial population, their velocity population, their personal best position group, the global best one position in this population. In the Optimization process, individuals are optimized one by one, which means that after updating one individual, its fitness, its $pbest$, and the current $gbest$ are then updated.

Moreover, in the Initialization process, the evaluation happens immediately after generating the initial population. In the Optimization process, the evaluation happens after generating a temporary population and before updating $pbest$ and $gbest$.

2.3 General Dictionary

2.3.1 Catalog of General Terminologies

Although these selected algorithms use different expressions, their core meanings are the same. For example, the memory position in CSA is the personal best position in PSO if we only consider the information carried by the memory position and the personal best position. Therefore, **the information carried by various expressions is the only principle to categorize the original terminologies**. As shown in Table 2.1, we find **20 general components can cover the entire components in these seven selected algorithms**. Specifically, only 20 kinds of information are used when modeling these seven algorithms. At the same time, **these 20 general components can be further**

divided into two groups: compulsory components and selective components, as shown in Figure 2.1.

In Table 2.1, ***Information 1 to 5, 13, 15, 17-20* are compulsory components**, because they must exit when modeling a nature-inspired algorithm. In other words, these components can theoretically make up a most basic nature-inspired algorithm, if we don't consider whether the algorithm performance is excellent. Meanwhile, we consider the ***Information 18* is also a compulsory component**, because firstly, selection process is an important process of natural evolution; secondly, it is reasonable to convert a process that does not use selection to using a special selection process that all individuals are delivered into the next generation round. More details are as follows:

- *Information 1, 3, 4*: the objective function $f()$ must exist to determine where the algorithm will happen. When the $f()$ is determined, the dimension n and the boundary $[lb_{\mathbf{x}}, ub_{\mathbf{x}}]$ are also determined which is because they exist in the $f()$.
- *Information 2, 5, 13, 15*: when searching possible solutions to $f()$, possible solutions $\mathbf{x}_i$ with the number M of $\mathbf{x}_i$ also must exist, because it determines the size of the searching pool in which the algorithm could find the final optimal solution to f. Furthermore, the method $Init_{\mathbf{x}}$ to initialize the initial possible solutions also must exist, because it determines where the algorithm starts to find the final optimal solution, which means there also must be a method $Opt_{\mathbf{x}}$ that can iteratively optimize the initial possible solutions.
- *Information 17, 18, 19, 20*: because these swarm-based optimization algorithms are a kind of iterative optimization heuristic algorithms in which sampling and repetition play an important role in finding optimal solutions [7], the method S related to sampling and the stop condition T must exist. Moreover, these algorithms will solve the problem in a limited searching space, there must also be a method C to deal with outliers outside the searching space.

Therefore, we conclude that f, $\mathbf{x}_i$, M, T and methods $Init_{\mathbf{x}}$, $Opt_{\mathbf{x}}$, C, S must exist in a swarm-based optimization algorithm. As shown in the left side of Figure 2.1, these compulsory components exist in all of seven selected algorithms.

In Table 2.1, ***Information 6 to 12, 14, 16* are selective components**, because not every algorithm needs these information. We find these information plays a similar role in modeling nature-inspired algorithms, and the role is that these information acts on compulsory components to improve the performance of the algorithm. **Considering the role of these information is to influence how far the newly generated individual will move, we define these selective components step-size with notation Δ.**

Meanwhile, we consider the ***Information 11* is also a selective component**, because their main role is to help to achieve a higher quality of optimization algorithm, although it exists in every algorithm. Furthermore, we further categorize these selective components as as follows:

- *Information 11*: Static step-size $\Delta : w, z^0, [lb_{\mathbf{y}}, ub_{\mathbf{y}}]$.
  Most static numerical step-size, such as w, is able to be understood as common hyper-parameters in most algorithms. They are numbers; are set before running algorithms and are unchanged when approaching algorithms. In Figure 2.1, we find there are two groups of static step-size. The first group is w that is commonly seen as hyper-parameters. The second group is the initial value of dynamic step-size (see next item •) in which the dynamic numerical step-size z needs an initial value z^0, the $\mathbf{y}$-relative dynamic vector step size $\mathbf{y}_i$ needs an initial value $[lb_{\mathbf{y}}, ub_{\mathbf{y}}]$.
- *Information 6, 7, 8, 9, 12*: Dynamic step-size $\Delta : z, \mathbf{x}_{i_p}, \mathbf{x}_g, \mathbf{x}_s, \mathbf{y}_i$.
  - *Information 12*: Dynamic numerical step-size $\Delta : z$.
    The dynamic numerical step-size z is changing over the iteration t during the optimization process.
  - Dynamic vector step-size $\Delta : \mathbf{x}_{i_p}, \mathbf{x}_g, \mathbf{x}_s, \mathbf{y}_i$.
    As its name indicates, this kind of step-size is a vector with the same dimension as $\mathbf{x}$.
    - *Information 6, 7, 8*: $\mathbf{x}$-relative vector step size $\Delta : \mathbf{x}_{i_p}, \mathbf{x}_g, \mathbf{x}_s$.
      This kind of vector can be generated from the target $\mathbf{x}$. Specifically, there is a straightforward formula between $\mathbf{x}$ and $\mathbf{x}$-relative vector step-size. For example, personal best position $\mathbf{x}_{i_p}$, global best position $\mathbf{x}_g$. The $\mathbf{x}_s$ denotes a special operator related to $\mathbf{x}$, and it is mainly customized. For example, the sorted population in MFO.
    - *Information 9*: $\mathbf{y}$-relative vector step size $\Delta : \mathbf{y}_i$.
      This kind of vector is generated without considering the target $\mathbf{x}_i$. It can be imagined as a tool that one algorithm needs it to improve algorithm's performance. The algorithm can never generate this tool by itself, besides getting it from outside of itself. In other words, it is impossible to represent this kind of vector variable by translating other existing variables in a straightforward method. Normally, this kind of vector has same size as the population of individual. For example, velocity $\mathbf{v}_i$.

For other selective components *Information 10, 14, 16*, because the dynamic step-size $\Delta : \mathbf{x}_{i_p}, \mathbf{x}_g, \mathbf{x}_s, \mathbf{y}_i$ is changing during the optimization process, there must be an

initialization status and a changing process. Therefore, we also define $Init_\Delta$ and Opt_Δ to achieve the initialization status and the changing process for dynamic step sizes, in which the $Init_\Delta$ method can also include the set-up of static step size $\Delta : w, z^0, [lb_\mathbf{y}, ub_\mathbf{y}]$. As shown in the right side of Figure 2.1, all of these step-size Δ have an initial status determined by $Init_\Delta$, and each dynamic step-size $\Delta : z, \mathbf{x}_{i_p}, \mathbf{x}_g, \mathbf{x}_s, \mathbf{y}_i$ will change by an optimization operator Opt_Δ.

2.3.2 Extension in Other Swarm-based Algorithms.

This section discusses the possibility of applying this general dictionary in any other swarm-based algorithms. First of all, all of compulsory components (see *Information 1 to 5, 13, 15, 17-20* in Table 2.1) and several selective components (see *Information 6, 7, 11, 12* in Table 2.1) are easily detected in any one swarm-based algorithms, because they are straightforward. For example, $\mathbf{x}$ is the animal, w is the hyper-parameter and z is also the hyper-parameter but z is changing. Mostly any one swarm-based algorithm will use very clear noun or terminology to point out these Information.

The place where it is most likely to confuse users is **how to detect the dynamic vector step-size: Why we think one vector step-size is a $\mathbf{x}$-relative vector not a $\mathbf{y}$-relative vector $\mathbf{y}$?** This question can be answered according to the observations found in these seven algorithms.

In most of swarm-based algorithms, besides the animal $\mathbf{x}$, another noun is also always existing. This noun could be the velocity (in BA, PSO), the memory (in CSA) or the flame (in MFO). We are not allowed to directly assign them the $\mathbf{y}$-relative vector step-size, although it is the most straightforward way. **The reason is that the information carried by the noun is the only principle used to assign them different general component name.** For example:

- The memory $\mathbf{m}_i$ in CSA is initialized by $\mathbf{x}_i$ itself as $\mathbf{m}_i(t=0) = \mathbf{x}_i(t=0)$, and then will be updated by $\mathbf{m}_i(t+1) = \mathbf{Min}\{\mathbf{m}_i(t), \mathbf{x}_i(t+1)\}$. Therefore in the first iteration, the update will happen as Eq.2.36. In other words, the $\mathbf{m}_i$ always can be represented by $\mathbf{x}_i$ all the time. Moreover, the update method to the $\mathbf{m}_i$ is same to find the $\mathbf{x}_{i_p}$, therefore, it is totally safe to replace $\mathbf{m}_i$ by $\mathbf{x}$-relative vector $\mathbf{x}_{i_p}$rather than the $\mathbf{y}$-relative vector $\mathbf{y}_i$.

$$\begin{aligned} \mathbf{m}_i(t=1) &= \mathbf{Min}\{\mathbf{m}_i(t=0), \mathbf{x}_i(t=1)\} \\ &= \mathbf{Min}\{\mathbf{x}_i(t=0), \mathbf{x}_i(t=1)\} \\ &= \mathbf{x}_i(t=0) \text{ or } \mathbf{x}_i(t=1) \end{aligned} \tag{2.36}$$

- The flame $\langle flame_i \rangle$ in MFO is initialized by $\langle \mathbf{x}_i \rangle$ itself as $\langle flame_i \rangle(t=0) = \langle \mathbf{x}_i \rangle(t=0)$, and then will be updated by $\langle flame_i \rangle(t+1) = \mathbf{Sort}(\{\mathbf{x}_i(t)\} \cup \{\mathbf{x}_i(t+1)\}), i = 1 \dots M$. Therefore, in the first iteration, the update will happen as Eq.2.37. In other words, the $\langle flame_i \rangle$ always can be represented by $\mathbf{x}$-relative vectors $\langle \mathbf{x}_i \rangle$ all the time, rather than the $\mathbf{y}$-relative vectors $\langle \mathbf{y}_i \rangle$.

$$\begin{aligned} \langle flame_i \rangle(t=1) &= \mathbf{Sort}(\langle flame_i \rangle(t=0) \cup \{\mathbf{x}_i(t=1)\}), i = 1 \dots M \\ &= \mathbf{Sort}(\langle \mathbf{x}_i \rangle(t=0) \cup \{\mathbf{x}_i(t=1)\}), i = 1 \dots M \end{aligned} \tag{2.37}$$

- The velocity in BA and PSO uses a different way to initialize itself, such as $\boldsymbol{\mathcal{U}}(lb, ub)$. Moreover, the most important point is there is no way to represent the velocity by $\mathbf{x}$-relative vector $\mathbf{x}_i$. Therefore, we prefer assigning $\mathbf{y}$-relative vector $\mathbf{y}_i$ to this kind of step-size.

After resolving the confusion between $\mathbf{x}$-relative vector step size and $\mathbf{y}$-relative vector step size, other *Information 8 to 10, 14, 16* also become clear to detect in any other swarm-based algorithms according to the previous observations.

2.4 Summary

In this chapter, we discuss the commonalities among terminologies in seven selected algorithms. We summarize their commonalities in Table 2.1 in which we conclude 20 general components can cover entire components in these algorithms. Meanwhile, we discuss the classification of these 20 general components in Figure 2.1.

In Subsection 2.3.1, we detail the classification of these 20 general components. Every algorithm must have the target problem f, the search space n and $[lb_{\mathbf{x}}, ub_{\mathbf{x}}]$, the possible target solution $\mathbf{x}_i$, the population size M, the stop condition T, the initialization method $Init_{\mathbf{x}}$ to $\mathbf{x}_i$, the optimization method $Opt_{\mathbf{x}}$ to $\mathbf{x}_i$, the method C to deal with outliers and the method S to select which individuals could be delivered into the next generation. However, selective components are more flexible, and these selective components are all related to determine how far the newly generated individual will move. Therefore, we name selective components step-size Δ. These selective components can be further categorized into static step-size $\Delta : w, z^0, [lb_{\mathbf{y}}, ub_{\mathbf{y}}]$ and dynamic step-size $\Delta : z, \mathbf{x}_{i_p}, \mathbf{x}_g, \mathbf{x}_s, \mathbf{y}_i$. What is interesting is the dynamic step-size also has two categories: dynamic numeric step-size $\Delta : z$ and dynamic vector step-size $\Delta : \mathbf{x}_{i_p}, \mathbf{x}_g, \mathbf{x}_s$ that are $\mathbf{x}$-relative vector step-size and $\Delta : \mathbf{y}_i$ that is $\mathbf{y}$-relative vector step-size.

In Subsection 2.3.2, we discuss how to extend this general dictionary into any other swarm-based algorithms. Furthermore, we resolve a confusion of the difference between $\mathbf{x}$-relative vector step size and $\mathbf{y}$-relative vector step size.

In conclusion, there is a general dictionary in which we can discuss these seven selected algorithms on the level of totally same components.

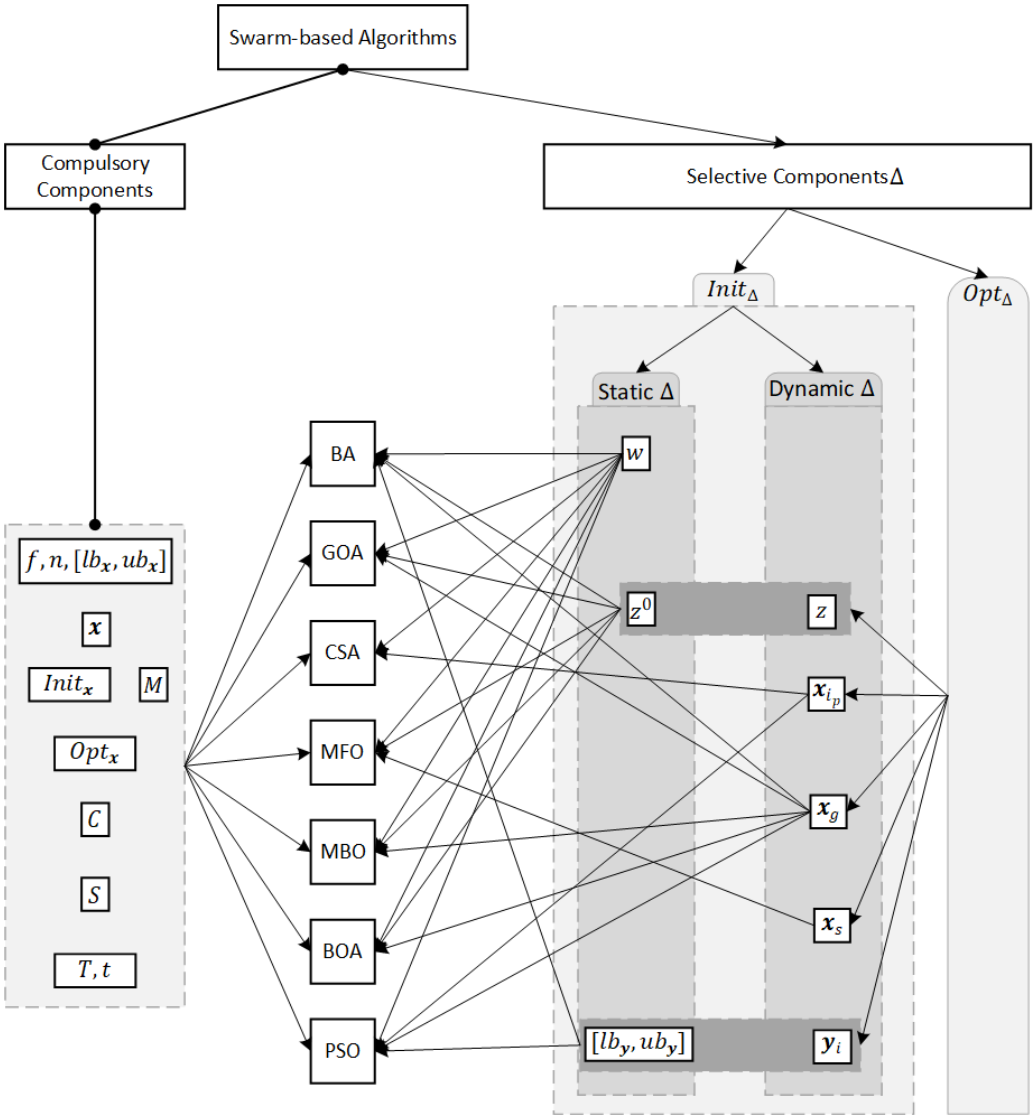

Figure 2.1: These algorithms made of the entire compulsory components and some of selective components.

Table 2.1: General dictionary: all information appearing in these seven algorithms can be categorized into 20 components. The none in gray color means one information doesn't exist in this algorithm.

Information	Different Representation	General Notation
1.The objective optimization problem.	The representations in seven algorithms are same.	$f()$: optimization function.
2.One possible objective solution.	•BA: each bat $\mathbf{x}_i$. •GOA: each grasshopper $\mathbf{x}_i$. •CSA: crow $\mathbf{x}_i$. •MFO: moth $\mathbf{m}_i$. •MBO: monarch butterfly $\mathbf{x}_i$. •BOA: butterfly $\mathbf{x}_i$. •PSO: particle $\mathbf{x}_i$.	$\mathbf{x}_i$: one objective solution.
3.The number of independent elements in $\mathbf{x}_i$.	•BA: d. •GOA: dim. •CSA: d. •MFO: dim. •MBO: k. •BOA: dim. •PSO: dimension.	n: dimension of the $f()$ searching space.
4.The boundary of each independent element in $\mathbf{x}_i$[19].	•BA: Lb/Ub. •GOA: $down/up$. •CSA: l/u. •MFO: lb/ub. •MBO: $MinParValue/MaxParValue$. •BOA: Lb/Ub. •PSO: lb/ub.	$lb_{\mathbf{x}}/ub_{\mathbf{x}}$: the lower/upper boundary of all element in $\mathbf{x}_i$.
5.The number of $\mathbf{x}_i$.	•BA : n. •GOA: N. •CSA: N. •MFO: N. •MBO: M. •BOA: n. •PSO: N.	M: population size.
6.The best $\mathbf{x}_i$ that it itself has found so far. It will change during the optimization process.	•BA :none. •BA :none. •GOA: none. •CSA: memory $\mathbf{m}_i$. •MFO: none. •MBO: none. •BOA: none. •PSO: $pbest$.	$\mathbf{x}_{i_p}$: $\mathbf{x}$-relative dynamic step size Δ.
7.The best $\mathbf{x}_i$ that the entire population has found so far. It will change during the optimization process.	•BA: $\mathbf{x}_*$. •GOA: $\mathbf{T}$. •CSA: none. •MFO: none. •MBO: $\mathbf{x}_{best}$. •BOA: g^*. •PSO: $gbest$.	$\mathbf{x}_g$: $\mathbf{x}$-relative dynamic step size Δ.

[19]Some algorithms, such as GOA, MFO, consider the boundary of every element is different, however, we only consider the boundary of every element is same in this work.

8.Some special vector step sizes are able to be represented by $\mathbf{x}_i$. It will change during the optimization process.	•BA: none. •GOA: none. •CSA: none. •MFO: sorted population. •MBO: none. •BOA: none. •PSO: none.	$\mathbf{x}_s$: $\mathbf{x}$-relative dynamic step size Δ.
9.Some special vector step sizes are not able to be represented by $\mathbf{x}_i$. It will change during the optimization process.	•BA:velocity $\mathbf{v}_i$. •GOA: none. •CSA: none. •MFO: none. •MBO: none. •BOA: none. •PSO: velocity $\mathbf{v}_i$.	$\mathbf{y}_i$: assisting step size Δ.
10.The boundary of each independent element in $\mathbf{y}_i$.	•BA: Eq.2.2. •GOA: none. •CSA: none. •MFO: none. •MBO: none. •BOA: none. •PSO: Eq.2.32.	$lb_{\mathbf{y}}/ub_{\mathbf{y}}$: the lower/upper boundary of $\mathbf{y}_i$.
11.step sizes that are set before running algorithm. They are unchanged during the optimization process.	•BA: echo frequency interval $[freq\_min, freq\_max]$, 3 decreasing factors ϵ, α, γ. •GOA: attractive intensity f, attractive length l. •CSA: awareness probability AP, flight length fl. •MFO: spiral shape b. •MBO: $Keep$, $partition$, $period$, BAR, S_{max}. •BOA: switch probability p, $power\_exponent$, $sensory\_modality$. •PSO: adjusting parameter w, cognitive coefficient c_1, social coefficient c_2.	w: step size Δ.
12.step sizes whose initial values are set before running algorithm. They will change by math formulas during the optimization process.	•BA: loudness A, pulse rate $rate$. •GOA: coefficient c. •CSA: none. •MFO: decreasing weight tt, threshold $Fame\_no$. •MBO: weight α. •BOA: sensory_modality $sensory\_modality$. •PSO: none.	z: step size Δ.
13.How to initialize $\mathbf{x}_i$ with size M in the n dimension environment.	•BA: Eq.2.1. •GOA: Eq.2.7. •CSA: Eq.2.12. •MFO: Eq.2.16. •MBO: Eq.2.21. •BOA: Eq.2.26. •PSO: Eq.2.31.	$Init_{\mathbf{x}}$: initialization method on $\mathbf{x}$.

14.How to initialize dynamic vector step sizes.	•BA: Eq.2.2,step(4)(6)(7). •GOA: step(3),Eq.2.8. •CSA: Eq.2.13. •MFO: Eq.2.17,Eq.2.18,step(4). •MBO: step(6),Eq.2.23. •BOA: step(3),Eq.2.27. •PSO: Eq.2.32,Eq.2.33,step(5).	$Init_\Delta$: initialization method on Δ.
15.How to update $\mathbf{x}_i$.	•BA: Eq.2.5,Eq.2.6. •GOA: Eq.2.10. •CSA: Eq.2.14. •MFO: Eq.2.19,Eq.2.20. •MBO: step(5)(6),Eq.2.22. •BOA: Eq.2.28,Eq.2.29. •PSO: Eq.2.35.	$Opt_\mathbf{x}$: optimization method on $\mathbf{x}$.
16.How to update dynamic step sizes.	•BA: Eq.2.3,Eq.2.4,Eq.2.5. •GOA: Eq.2.8,step(9). •CSA: Eq.2.15. •MFO: Eq.2.17,Eq.2.18. •MBO: Eq.2.23,step(5). •BOA: step(3),Eq.2.30. •PSO: Eq.2.3.•PSO: Eq.2.35,sep(7)(8).	Opt_Δ: optimization method on Δ.
17.How to deal with outliers in $\mathbf{x}_i$.	•BA: step(10). •GOA: step(7). •CSA: step(6). •MFO: step(6). •MBO: step(7). •BOA: step(7). •PSO: common method.	C: clip outliers in $\mathbf{x}_i$.
18.How to decide if the new generated $\mathbf{x}_i$ would be accepted.	•BA: step(11). •GOA: none. •CSA: none. •MFO: none. •MBO: step(8). •BOA: step(8). •PSO: none.	S: selection method on new generated population.
19.The iteration counter.	•BA: t. •GOA: l. •CSA: t. •MFO: t. •MBO: t. •BOA: t. •PSO: current iteration.	t: current iteration.
20.The maximum number of iterations.	•BA: t_max. •GOA: $max\_iteration$. •CSA: $tmax$. •MFO: T. •MBO: $Naxgen$. •BOA: N_iter. •PSO: stop condition.	T: the budget.

Chapter 3

Unified Procedure

3.1 Introduction

This chapter discusses the commonality among seven selected algorithms on the level of their procedures for approaching optimization. In other words, we discuss how these 20 unified components make up a valid algorithm. Specifically, we find that these seven selected algorithms can arrange their components in the same order.

In Chapter 2, these algorithms are able to be composed of 20**Huilin: not 20, shall be 8 functions** components, and these components can be categorized into compulsory and selective components according to their properties. This chapter discusses the category of these components according to their instructions because instructions determine their positions when constructing a valid nature-inspired algorithm [25]. As shown in Table 3.1, we find that **a valid nature-inspired algorithm requires only eight instructions to implement**. This valid algorithm model can be defined in 8-tuple as Eq.4.1 in which f, $Init_{\mathbf{x}}$, $Opt_{\mathbf{x}}$, C, S, $Init_{\Delta}$, Opt_{Δ} represent several kinds of operation strategy, and T represents a kind of stop strategy.

$$NIOA = (f, Init_{\mathbf{x}}, Opt_{\mathbf{x}}, C, T, S, Init_{\Delta}, Opt_{\Delta}) \tag{3.1}$$

We explain these eight instructions as follows:

- *Index.1*: 'Evaluation' instruction f:
  'Evaluation' is needed to measure the quality of solutions the algorithm finds to the problem, when modeling a valid nature-inspired algorithm.

- *Index.2*: 'Initialize population' instruction $Init_{\mathbf{x}}$:
  'Initialize population' provides a set of possible solutions, and a nature-inspired algorithm will improve this set and find the optimal one in this set.
- *Index.3*: 'Initialize step-size' instruction $Init_{\Delta}$:
  'Initialize step-size' helps the nature-inspired algorithm to improve the quality of possible solutions efficiently.
- *Index.4*: 'Update population' instruction $Opt_{\mathbf{x}}$:
  'Update population' is the strategy the algorithm used to improve the quality of possible solutions to the problem.
- *Index.5*: 'Update step-size' instruction Opt_{Δ}:
  If the step-size used by the algorithm is dynamically changing, 'Update step-size' is needed to implement the mechanism of changing.
- *Index.6*: 'Outliers treatment' instruction C:
  'Outliers treatment' deals with every outlier when the nature-inspired algorithm updates the population.
- *Index.7*: 'Selection' instruction S:
  'Selection' determines how many updated individuals will be delivered into the next generation round, when the nature-inspired algorithm iteratively updates the population.
- *Index.8*: 'Stop strategy' instruction T:
  'Stop strategy' determines when the iterative optimization process stops.

Huilin: index to tuple

Table 3.1: The category of these 20 components, according to their functionalities. **Huilin: explai the number in the cell**

Index	Component		Functionality	
Index.1	1.f		evaluation	
Index.2	2.$\mathbf{x}_i$ 3.n 4.$lb_{\mathbf{x}}/ub_{\mathbf{x}}$ 5.M 13.$Init_{\mathbf{x}}$		initialize the initial population.	
		6.$\mathbf{x}_{i_p}$ 7.$\mathbf{x}_g$ 8.$\mathbf{x}_s$	initialize $\mathbf{x}$-relative dynamic step-size Δ.	
Index.3	14.$Init_{\Delta}$		initialize step-size Δ.	

continued

. . . continued

Index	Component		Functionality	
		9.$\mathbf{y}_i$ 10.$lb_{\mathbf{y}}/ub_{\mathbf{y}}$	initialize $\mathbf{y}$-relative dynamic step-size Δ.	
		11.w	initialize static step-size Δ.	
		12.z	initialize dynamic numeric step-size Δ.	
Index.4	15.$Opt_{\mathbf{x}}$		update the population.	
Index.5	16.Opt_{Δ}		update step-size Δ.	
Index.6	17.C		outliers treatment.	
Index.7	18.S		deliver selected individuals into next generation.	
Index.8	19.t 20.T		iteration counter, and the maximum number of generation	

?? discusses whether the eight tuples are ordered at the same position in these seven selected algorithms. The overview summary about this chapter is displayed in **??**. 1. all calculation between dimensions are converted into vector.

2. not mention where is the static Δ, because no matter where it is calculated, they can move to the initial step. Because it doesn't matter where it is setup.

3. focus on where they put these 20 components. In this section, we detail how we extract the general flow of optimization process. As summarized in Chapter 2, Algorithm.1 to Algorithm.1 give more details **only** about their optimization flows, but we describe the flow by using general terminologies in Table 2.1.

3.2 Unified Procedure

In this section, firstly, we detail each algorithm's original position of each tuple. Secondly, we discuss the possibility of arranging the eight tuples in the same position in these seven algorithms.

Algorithm 1 original positions of eight tuples in BA

$t \leftarrow 0$ ▷ *iteration counter*
Tuple.2 $Init_{\mathbf{x}}$ ▷ initialize population
Tuple.1 f ▷ evaluation
Tuple.3 $Init_{\Delta:w}$ ▷ initialize w-relative step-size Δ
Tuple.3 $Init_{\Delta:\mathbf{y}}$ ▷ initialize $\mathbf{y}$-relative step-size Δ
Tuple.3 $Init_{\Delta:\mathbf{x}}$ ▷ initialize $\mathbf{x}$-relative step-size Δ
while *Tuple.8* T **do** ▷ stop strategy
    Tuple.3 $Init_{\Delta:z}$ and *Tuple.5* $Opt_{\Delta:z}$ ▷ initialize and update z-relative step-size Δ
    Tuple.5 $Opt_{\Delta:\mathbf{y}}$ ▷ update $\mathbf{y}$-relative step-size Δ
    Tuple.4 $Opt_{\mathbf{x}}$ ▷ update population
    Tuple.6 C ▷ outliers treatment
    Tuple.1 f ▷ evaluation
    Tuple.7 S ▷ selection
    Tuple.5 $Opt_{\Delta:\mathbf{x}}$ ▷ update $\mathbf{x}$-relative step-size Δ
    $t \leftarrow t + 1$
end while

Algorithm 2 original positions of eight tuples in GOA

$t \leftarrow 0$ ▷ *iteration counter*
Tuple.2 $Init_{\mathbf{x}}$ ▷ initialize population
Tuple.1 f ▷ evaluation
Tuple.3 $Init_{\Delta:w}$ ▷ initialize w-relative step-size Δ
Tuple.3 $Init_{\Delta:\mathbf{x}}$ ▷ initialize $\mathbf{x}$-relative step-size Δ
while *Tuple.8* T **do** ▷ stop strategy
    Tuple.3 $Init_{\Delta:z}$ and *Tuple.5* $Opt_{\Delta:z}$ ▷ initialize and update z-relative step-size Δ
    Tuple.4 $Opt_{\mathbf{x}}$ ▷ update population
    Tuple.6 C ▷ outliers treatment
    Tuple.1 f ▷ evaluation
    Tuple.5 $Opt_{\Delta:\mathbf{x}}$ ▷ update $\mathbf{x}$-relative step-size Δ
    $t \leftarrow t + 1$
end while

Algorithm 3 original positions of eight tuples in CSA

$t \leftarrow 0$ ▷ *iteration counter*
Tuple.2 $Init_{\mathbf{x}}$ ▷ initialize population
Tuple.1 f ▷ evaluation
Tuple.3 $Init_{\Delta:w}$ ▷ initialize w-relative step-size Δ
Tuple.3 $Init_{\Delta:\mathbf{x}}$ ▷ initialize $\mathbf{x}$-relative step-size Δ
while *Tuple.8* T **do** ▷ stop strategy
    Tuple.4 $Opt_{\mathbf{x}}$ ▷ update population
    Tuple.6 C ▷ outliers treatment
    Tuple.1 f ▷ evaluation
    Tuple.5 $Opt_{\Delta:\mathbf{x}}$ ▷ update $\mathbf{x}$-relative step-size Δ
    $t \leftarrow t + 1$
end while

Algorithm 4 original positions of eight tuples in MFO

$t \leftarrow 0$ ▷ *iteration counter*
Tuple.2 $Init_{\mathbf{x}}$ ▷ initialize population
Tuple.3 $Init_{\Delta:w}$ ▷ initialize w-relative step-size Δ
while *Tuple.8* T **do** ▷ stop strategy
    Tuple.3 $Init_{\Delta:z}$ and *Tuple.5* $Opt_{\Delta:z}$ ▷ initialize and update z-relative step-size Δ
    Tuple.6 C ▷ outliers treatment
    Tuple.1 f ▷ evaluation
    Tuple.3 $Init_{\Delta:\mathbf{x}}$ and *Tuple.5* $Opt_{\Delta:\mathbf{x}}$ ▷ initialize and update $\mathbf{x}$-relative step-size Δ
    Tuple.4 $Opt_{\mathbf{x}}$ ▷ update population
    $t \leftarrow t + 1$
end while

Algorithm 5 original positions of eight tuples in MBO

1: $t \leftarrow 0$ ▷ *iteration counter*
2: *Tuple.2* $Init_{\mathbf{x}}$ ▷ initialize population
3: *Tuple.1* f ▷ evaluation
4: *Tuple.3* $Init_{\Delta:w}$ ▷ initialize w-relative step-size Δ
5: *Tuple.3* $Init_{\Delta:\mathbf{x}}$ ▷ initialize $\mathbf{x}$-relative step-size Δ
6: **while** *Tuple.8* T **do** ▷ stop strategy
7: *Tuple.4* $Opt_{\mathbf{x}}$ ▷ update population
8: *Tuple.1* f ▷ evaluation
9: *Tuple.4* $Opt_{\mathbf{x}}$ ▷ update population
10: *Tuple.1* f ▷ evaluation
11: *Tuple.7* S ▷ selection
12: *Tuple.5* $Opt_{\Delta:\mathbf{x}}$ ▷ update $\mathbf{x}$-relative step-size Δ
13: $t \leftarrow t + 1$
14: **end while**

Algorithm 6 original positions of eight tuples in BOA

1: $t \leftarrow 0$ ▷ *iteration counter*
2: *Tuple.2* $Init_{\mathbf{x}}$ ▷ initialize population
3: *Tuple.1* f ▷ evaluation
4: *Tuple.3* $Init_{\Delta:z}$ ▷ initialize z-relative step-size Δ
5: *Tuple.3* $Init_{\Delta:\mathbf{x}}$ ▷ initialize $\mathbf{x}$-relative step-size Δ
6: **while** T **do** ▷ stop strategy
7: *Tuple.4* $Opt_{\mathbf{x}}$ ▷ update population
8: *Tuple.6* C ▷ outliers treatment
9: *Tuple.1* f ▷ evaluation
10: *Tuple.7* S ▷ selection
11: *Tuple.5* $Opt_{\Delta:\mathbf{x}}$ ▷ update $\mathbf{x}$-relative step-size Δ
12: *Tuple.5* $Opt_{\Delta:z}$ ▷ update z-relative step-size Δ
13: $t \leftarrow t + 1$
14: **end while**

Algorithm 7 original positions of eight tuples in PSO

1: $t \leftarrow 0$ ▷ *iteration counter*
2: *Tuple.2* $Init_{\mathbf{x}}$ ▷ initialize population
3: *Tuple.1* f ▷ evaluation
4: *Tuple.3* $Init_{\Delta:w}$ ▷ initialize w-relative step-size Δ
5: *Tuple.3* $Init_{\Delta:\mathbf{y}}$ ▷ initialize $\mathbf{y}$-relative step-size Δ
6: *Tuple.3* $Init_{\Delta:\mathbf{x}}$ ▷ initialize $\mathbf{x}$-relative step-size Δ
7: **while** T **do** ▷ stop strategy
8: *Tuple.5* $Opt_{\Delta:\mathbf{y}}$ ▷ update $\mathbf{y}$-relative step-size Δ
9: *Tuple.4* $Opt_{\mathbf{x}}$ ▷ update population
10: *Tuple.6* C ▷ outliers treatment
11: *Tuple.1* f ▷ evaluation
12: *Tuple.5* $Opt_{\Delta:\mathbf{x}}$ ▷ update $\mathbf{x}$-relative step-size Δ
13: $t \leftarrow t + 1$
14: **end while**

3.3 Summary

Chapter 4

Unified Framework

4.1 Unified Nature-Inspired Optimization Algorithm

In this chapter, we provide an outline of nature-inspired optimization algorithms. This framework is able to cover all seven different algorithms mentioned in Chapter 2. As in Table 2.1, all components of algorithms can be represented by 20 components whose notations are only basic mathematical notations. This general outline can be defined in 10-tuple as Eq.4.1 in which f, $Init_{\mathbf{x}}$, $Opt_{\mathbf{x}}$, C, S, $Init_{\Delta}$, Opt_{Δ} are functional equations, M is a natural number, $\mathbf{x}$ is a vector and T is logical statement.

$$NIOA = (f, \mathbf{x}, M, Init_{\mathbf{x}}, Opt_{\mathbf{x}}, C, T, S, Init_{\Delta}, Opt_{\Delta}) \tag{4.1}$$

where

(1) Objective problem: f.
f represents the math programmed actual optimization problem. f also provides the information about the searching space, such as the dimension n, the constraint $[lb_{\mathbf{x}}, ub_{\mathbf{x}}]$. It must exist.

(2) Objective solution: $\mathbf{x}$.
$\mathbf{x}$ is one possible optimal solution obtained from the algorithm. It must exist.

(3) Population size: M.
The population size M is a very important component in nature-inspired algorithms [26]. It must exist.

(4) Objective solution initialization method: $Init_{\mathbf{x}}(n, M, [lb_{\mathbf{x}}, ub_{\mathbf{x}}])$.
The Eq.4.2 is the first step in all swarm-based algorithms. It must exist.

$$Init_{\mathbf{x}}(n, M, [lb_{\mathbf{x}}, ub_{\mathbf{x}}]) \coloneqq \boldsymbol{\mathcal{U}}(lb_{\mathbf{x}}, ub_{\mathbf{x}}), i = 1, 2, \ldots, M \tag{4.2}$$

(5) Objective solution optimization method: $Opt_{\mathbf{x}}$.
The optimization method is an important feature that determines the optimization ability of swarm-based algorithms. It is different in different algorithms. It must exist.

(6) Dealing with outliters method: C.
In these seven algorithms, there are two methods to deal with outliers in $\mathbf{x}_i$ after obtaining new generated $\hat{\mathbf{x}}_i$[1]. The Eq.4.3 is commonly used in most algorithms, for example, except the CSA model, other algorithms in this work all use the Eq.4.3 to deal with outliers. In this work, the CSA model especially uses the Eq.4.4. It must exist[2].

$$\mathbf{x}_{i,n}^{\text{fixed}}(t+1) = \begin{cases} ub_x & , \quad \mathbf{x}_{i,n}(t+1) > ub_{\mathbf{x}} \\ \mathbf{x}_{i,n}(t+1) & , \quad \text{o.w} \\ lb_x & , \quad \mathbf{x}_{i,n}(t+1) < lb_{\mathbf{x}} \end{cases} \tag{4.3}$$

$$\mathbf{x}_{i,n}^{\text{fixed}}(t+1) = \begin{cases} \mathbf{x}_{i,n}(t) & , \quad \mathbf{x}_{i,n}(t+1) < lb_{\mathbf{x}} \text{ or } \mathbf{x}_{i,n}(t+1) > ub_{\mathbf{x}} \\ \mathbf{x}_{i,n}(t+1) & , \quad \text{o.w} \end{cases} \tag{4.4}$$

(7) Stop condition: T.
The T determines if the iterative optimization process would stop or not. It must exist.

(8) Selection method: S.
The method Eq.4.5 determines if the new generated $\hat{\mathbf{x}}_i(t+1)$ would be accepted and would go to the next optimization round. This method differs from different special conditions. It might exist. As a side note, some algorithms, such as

[1] In this work, $\hat{\mathbf{x}}(t+1)$ is the new solutions generated before Selection S. However, $\mathbf{x}(t+1)$ is the new solutions generated after Selection S, which is also the final new updated solutions after one complete optimizarion round.

[2] Some algorithms didn't mention C in their published paper or code, however, it is acceptable that outliers are invalid and must be dealt with.

MBO, might use very specific method to select from the new generated population, therefore, such methods might not be able to be involved in the Eq.4.5.

$$\mathbf{x}_i(t+1) = \begin{cases} \hat{\mathbf{x}}_i(t+1), & \text{special conditions} \\ \mathbf{x}_i(t), & \text{o.w} \end{cases} \tag{4.5}$$

(9) step-sizes initialization method: $Init_{\Delta}$.
step-sizes Δ are selective factors that play a crucial role in $Opt_{\mathbf{x}}$. There are three types of step-sizes as we studied in the Table 2.1 and they might exist or not.

(a) Static numerical step-size: $w, z^0, [lb_{\mathbf{y}}, ub_{\mathbf{y}}]$.
This kind of step-size is fixed after setting-up at the beginning. The w is commonly used in most algorithms and its common name is hyper-parameter. The z^0 and $[lb_{\mathbf{y}}, ub_{\mathbf{y}}]$ are respectively initial values of dynamic numerical step-sizes and the assisting vector step-size.

(b) Dynamic step-size: z, $\mathbf{x}_{i_p}$, $\mathbf{x}_g$, $\mathbf{x}_s$, $\mathbf{y}_i$.

i. Dynamic numerical step-size: z.
The z is a kind of hyper-parameter that will be changing as the iteration t during the optimization process. Its initialization method is varying.

ii. Dynamic vector step-size: $\mathbf{x}_{i_p}$, $\mathbf{x}_g$, $\mathbf{x}_s$, $\mathbf{y}_i$.

A. $\mathbf{x}$-relative vector step-size: $\mathbf{x}_{i_p}$, $\mathbf{x}_g$, $\mathbf{x}_s$.

- the $\mathbf{x}_{i_p}$ is the best $\mathbf{x}_i$ that each $\mathbf{x}_i$ has found so far. Its initialization method is fixed.

$$\mathbf{x}_{i_p}(t=0) = \mathbf{x}_i(t=0), i = 1 \dots M \tag{4.6}$$

- the $\mathbf{x}_g$ is the best $\mathbf{x}_i$ that the whole population has found so far. Its initialization method is fixed.

$$\mathbf{x}_g(t=0) = \textbf{Min}(\{\mathbf{x}_i(t=0)\}), i = 1 \dots M \tag{4.7}$$

- the $\mathbf{x}_s$ denotes one kind of special vector step-size that has a strong connection with the $\mathbf{x}$. It depends on different algorithms, so it is varying.

B. assisting vector step-size: $\mathbf{y}_i$.
In this work, there is only one kind of initialization method as Eq.4.8. In this work, its initialization method is fixed, however, it could be customized in the future.

$$\mathbf{y}_i(t=0) = \boldsymbol{\mathcal{U}}(lb_{\mathbf{y}}, ub_{\mathbf{y}}), i = 1 \dots M \tag{4.8}$$

(10) step-sizes optimization method: Opt_{Δ}.
As mentioned above, dynamic step-size will be changing during the optimization process, therefore, the Opt_{Δ} is the method to determine how they are changing during the optimization process. Mostly, it differs in different algorithms, however, for $\mathbf{x}_{i_p}$ and $\mathbf{x}_g$ whose initialization method is fixed, their optimization method is also fixed respectively as Eq.4.8 and Eq.4.10. It might exist or not.

$$\mathbf{x}_{i_p}(t+1) = \textbf{Min}(\{\mathbf{x}_{i_p}(t), \mathbf{x}_i(t+1)\}) \tag{4.9}$$

$$\mathbf{x}_g(t+1) = \textbf{Min}(\mathbf{x}_g(t) \cup \{\mathbf{x}_i(t+1)\}), i = 1 \dots M \tag{4.10}$$

and pseudo-code is Algorithm.8. The Section 4.1 will only give very intuitive findings in these seven algorithms based on the UNIOA framework. Later the Section 4.3 will conclude the UNIOA framework with more details after detailing how to re-frame each algorithm in the Section 4.2.

Algorithm 8 General Nature-Inspired Optimization Algorithm

$t \leftarrow 0$
$\mathbf{X}(t=0) \leftarrow Init_{\mathbf{x}}(n, M, [lb_{\mathbf{x}}, ub_{\mathbf{x}}])$ ▷ Initialization
$F(t=0) \leftarrow f(\mathbf{X}(t=0))$ ▷ Evaluation
$\{\Delta(t=0)\} \leftarrow Init_{\Delta}(\mathbf{X}(t=0), w, t)$
while T **do**
    $\{\mathbf{Y}(t+1) \leftarrow Opt_{\Delta:\mathbf{y}}(\Delta(t), w, t), \emptyset\}$ ▷ Optimization
    $\hat{\mathbf{X}}(t+1) \leftarrow Opt_{\mathbf{x}}(\mathbf{X}(t), \{\mathbf{Y}(t+1), \emptyset\}, \Delta^*(t), w, t, C)$
    $F(t+1) \leftarrow f(\hat{\mathbf{X}}(t+1))$ ▷ Evaluation
    $\mathbf{X}(t+1) \leftarrow S(\mathbf{X}(t), \hat{\mathbf{X}}(t+1), \{z(t), w, \emptyset\})$ ▷ Selection
    $\{\Delta^*(t+1)\} \leftarrow Opt_{\Delta^*}(\Delta^*(t), w, t+1)$
    $t \leftarrow t+1$
end while

The math representation Eq.4.1 displays **what components are used to construct these seven algorithms**, then, the pseudo-code Algorithm.8 will display **how these**

components are constructed to build up these seven algorithms, which is in the view of algorithms' optimization progress. As the Algorithm.8 described, every algorithm will start from the initialization process in which the population must be evaluated after being initialized, moreover, different algorithms will utilize $Init_{\Delta}$ differently. In the optimization process, before meeting the stop condition, some algorithms might firstly optimize the assisting vector step-size $\mathbf{y}_i$ before updating $\mathbf{x}_i$. After dealing with outliers in the new generated $\hat{\mathbf{x}}_i(t+1)$, $\hat{\mathbf{x}}_i(t+1)$ must be evaluated again before deciding if the new generated $\hat{\mathbf{x}}_i(t+1)$ will be accepted by using the selection method. Lastly before moving to the next optimization round, other dynamic step-sizes will be updated in some algorithms.

4.2 Re-framed Nature-inspired Algorithms

4.2.1 Re-framed BA

Description As discussed in the Table 2.1, the BA model utilized three kinds of dynamic step-sizes and several static step-sizes to achieve the optimization on the objective solutions. These three dynamic step-sizes are Δ: $\mathbf{y}_i$, $\mathbf{x}_g$ and z.

The re-framed BA model, in the initialization process, will specially initialize $\mathbf{y}_i$ as Eq.4.11, $\mathbf{x}_g$ as Eq.4.7, z_1 as Eq.4.12, z_2 as Eq.4.13. The static step-sizes will be also set at this moment.

$$\mathbf{y}_i(t=0) = \boldsymbol{\mathcal{U}}(lb_{\mathbf{y}}, ub_{\mathbf{y}}), i = 1 \dots M \tag{4.11}$$

$$z_1(t=0) = z_1^0 \times (1 - e^{-w_1 \times t}) \tag{4.12}$$

$$z_2(t=0) = z_2^0 \times w_2 \tag{4.13}$$

where $[lb_{\mathbf{y}}, ub_{\mathbf{y}}] = [0,0]$, $z_1^0 = 1$, $z_2^0 = 1$, $w_1 = 0.1$ and $w_2 = 0.97$.

Next in the optimization process, before the stop condition T meets, the re-framed BA model will firstly generate new $\mathbf{y}_i(t+1)$ as Eq.4.14, then generate $\hat{\mathbf{x}}_i(t+1)$ as Eq.4.15 with the help of dealing with outliers C as Eq.4.3.

$$\mathbf{y_i}(t+1) = \mathbf{y_i}(t) + \mathcal{U}(lb_{w_4}, ub_{w_4}) \times (\mathbf{x}_i(t) - \mathbf{x}_g(t)) \tag{4.14}$$

$$\hat{\mathbf{x}}_i(t+1) = \begin{cases} \mathbf{x}_g(t) + w_3 \times \boldsymbol{\mathcal{N}}(\mathbf{0}, \mathbf{1}) \times z_2(t), & r < z_1(t) \\ \mathbf{x}_i(t) + \mathbf{y}_i(t+1), & \text{o.w} \end{cases} \tag{4.15}$$

where $[lb^{w_4}, ub^{w_4}] = [0,2]$,$w_3 = 0.1$.

Lastly after evaluating the fitness of $\hat{\mathbf{x}}_i(t+1)$, the special condition used in the selection method is $r > z_2 \cap f(\hat{\mathbf{x}}_i(t+1)) < f(\mathbf{x}_i(t))$. At this moment, the re-framed BA model has obtained the final generated new solutions after one round optimization. And finally other dynamic step-sizes ($\mathbf{x}_g$, z_1, z_2) are going to be updated ($\mathbf{x}_g$ as Eq.4.10, z_1 as Eq.4.16, z_2 as Eq.4.17) before the next optimization round.

$$z_1(t+1) = z_1^0 \times (1 - e^{-w_1 \times t}) \tag{4.16}$$

$$z_2(t+1) = z_2(t) \times w_2 \tag{4.17}$$

Till now, all symbols that are identified after the 'where' statement are involved in the static step-sizes.

Algorithm 9 Bat Algorithm

1: $t \leftarrow 0$
2: $Init_{\mathbf{x}}, M$ ▷ Initialization Process
3: f ▷ Evaluation
4: $Init_{\Delta}: \mathbf{y}_i, \mathbf{x}_g, z_1, z_2,$
5: **while** termination criteria are not met **do**
6: $Opt_{\Delta} \rightarrow \mathbf{y}_i(t+1)$ ▷ Optimization Process
7: $Opt_{\mathbf{x}}, C \rightarrow \hat{\mathbf{x}}_i(t+1)$
8: $f(\hat{\mathbf{x}}_i(t+1))$ ▷ Evaluation
9: $S \rightarrow \mathbf{x}_i(t+1)$ ▷ Selection
10: $t \leftarrow t+1$
11: $Opt_{\Delta} \rightarrow \mathbf{x}_g(t+1), z_1(t+1), z_2(t+1)$
12: **end while**

4.2.2 Re-framed GOA

As discussed in the Table 2.1, the GOA model utilized two kinds of dynamic step-sizes and several static step-sizes to achieve the optimization on the objective solutions. These two dynamic step-sizes are Δ: $\mathbf{x}_g$ and z.

The re-framed GOA model, in the initialization process, will specially initialize $\mathbf{x}_g$ as Eq.4.7, z as Eq.4.18. The static step-sizes will also be set at this moment.

$$z(t=0) = ub^z - t \times (\frac{ub^z - lb^z}{T}) \tag{4.18}$$

Next in the optimization process, before the stop condition T meets, the re-framed GOA model will firstly directly generate $\hat{\mathbf{x}}_i(t+1)$ as Eq.4.19 with the help of dealing with

outliers C as Eq.4.3.

$$\widetilde{D}_{i,j}(t) = 2 + \textbf{Dist}(\mathbf{x}_i(t), \mathbf{x}_j(t)) \bmod 2$$
$$\hat{\mathbf{x}}_i(t+1) = z(t) \times (\sum_{j=1, j\neq i}^{M} z(t) \times \frac{ub_{\mathbf{x}} - lb_{\mathbf{x}}}{2} \times (w_1 \times e^{\frac{-\widetilde{D}_{i,j}(t)}{w_2}} - e^{-\widetilde{D}_{i,j}(t)}) \times \frac{\mathbf{x}_i(t) - \mathbf{x}_j(t)}{\textbf{Dist}(\mathbf{x}_i(t), \mathbf{x}_j(t))}) + \mathbf{x}_g \quad 4.19$$

where $[ub^z, lb^z] = [0.00004, 1]$,$w_1 = 0.5$,$w_2 = 1.5$.

Lastly after evaluating the fitness of $\hat{\mathbf{x}}_i(t+1)$, the special condition used in the selection method is $\forall$. At this moment, the re-framed GOA model has obtained the final generated new solutions after one round optimization. And finally other dynamic step-sizes ($\mathbf{x}_g$, z) are going to be updated ($\mathbf{x}_g$ as Eq.4.10, z as Eq.4.20) before starting the next optimization round.

$$z(t+1) = ub_z - (t+1) \times (\frac{ub_z - lb_z}{T}) \quad 4.20$$

Till now, all symbols that are identified after the 'where' state are involved in the static step-sizes.

Algorithm 10 Grasshopper Optimization Algorithm

1: $t \leftarrow 0$
2: $Init_{\mathbf{x}}, M$ ▷ Initialization Process
3: f ▷ Evaluation
4: $Init_{\Delta}$: $\mathbf{x}_g, z$
5: **while** termination criteria are not met **do**
6: $Opt_{\mathbf{x}}, C \rightarrow \hat{\mathbf{x}}_i(t+1)$ ▷ Optimization Process
7: $f(\hat{\mathbf{x}}_i(t+1))$ ▷ Evaluation
8: $S \rightarrow \mathbf{x}_i(t+1)$ ▷ Selection
9: $t \leftarrow t+1$
10: $Opt_{\Delta} \rightarrow \mathbf{x}_g(t+1), z(t+1)$
11: **end while**

4.2.3 Re-framed CSA

As discussed in the Table 2.1, the CSA model utilized one kind of dynamic step-sizes and several static step-sizes to achieve the optimization on the objective solutions. The only one dynamic step-size is $\mathbf{x}_{i_p}$.

The re-framed CSA model, in the initialization process, will specially initialize $\mathbf{x}_{i_p}$ as Eq.4.6. The static step-sizes will be also set at this moment.

Next in the optimization process, before the stop condition T meets, the re-framed CSA model will directly firstly generate $\hat{\mathbf{x}}_i(t+1)$ as Eq.4.21 with the help of dealing with outliers C as Eq.4.4.

$$\hat{\mathbf{x}}_i(t+1) = \begin{cases} \mathbf{x}_i(t) + rand \times w_2 \times (\mathbf{x}_{j_p}(t) - \mathbf{x}_i(t)) & , \quad r > w_1 \\ \boldsymbol{\mathcal{U}}(lb_{\mathbf{x}}, ub_{\mathbf{x}}) & , \quad \text{o.w} \end{cases} \tag{4.21}$$

where $w_1 = 0.1$, $w_2 = 2$. Moreover, the $\mathbf{x}_{j_p}$ is any one neighbor around the $\mathbf{x}_{i_p}$.

Lastly after evaluating the fitness of $\hat{\mathbf{x}}_i(t+1)$, the special condition used in the selection method is $f(\hat{\mathbf{x}}_i(t+1)) < f(\mathbf{x}_i(t))$. At this moment, the re-framed CSA model has obtained the final generated new solutions after one round optimization. And finally the dynamic step-size is going to be updated before starting the next optimization round, in which the update method to $\mathbf{x}_{i_p}$ is as Eq.4.9.

Till now, all symbols that are identified after the 'where' statement are involved in the static step-sizes.

Algorithm 11 Crow Search Algorithm

$t \leftarrow 0$
$Init_{\mathbf{x}}, M$ ▷ Initialization Process
f ▷ Evaluation
$Init_{\Delta}$: $\mathbf{x}_{i_p}$
while termination criteria are not met **do**
    $Opt_{\mathbf{x}}, C \rightarrow \hat{\mathbf{x}}_i(t+1)$ ▷ Optimization Process
    $f(\hat{\mathbf{x}}_i(t+1))$ ▷ Evaluation
    $S \rightarrow \mathbf{x}_i(t+1)$ ▷ Selection
    $t \leftarrow t+1$
    $Opt_{\Delta} \rightarrow \mathbf{x}_{i_p}(t+1)$
end while

4.2.4 Re-framed MFO

Description As observed in the Table 2.1, in the re-framed MFO model, there are four critical components used to optimize the initial population. They are one dynamic vector step-size: $\mathbf{x}_s$ that is related to $\mathbf{x}$, and two dynamic numerical step-sizes: z_1, z_2.

In the initialization process, the re-framed MFO model initializes the initial population as Eq.4.2, and step-sizes: $\mathbf{x}_s$ as Eq.4.22, z_{1_i} as Eq.4.23, z_2 as Eq.4.24. Meanwhile, one

static step-size w is also set up.

$$\langle \mathbf{x}_i(t=0) \rangle = \mathbf{Sort}(\{\mathbf{x}_i(t=0)\}), i = 1 \dots M \tag{4.22}$$

$$z_{1_i}(t=0) = rand \times (-2 - \frac{t}{T}) + 1 \tag{4.23}$$

$$z_2(t=0) = \mathbf{Round}(M - t \times \frac{M-1}{T}) \tag{4.24}$$

Next in the optimization, before the stop condition T meets, the re-framed MFO model directly generates the $\hat{\mathbf{x}}_i$ as Eq.4.25, and outliers occurred in $\hat{\mathbf{x}}_i$ will be fixed immediately as Eq.4.3.

$$\hat{\mathbf{x}}_i(t+1) = \begin{cases} (\mathbf{x}_{s_i}(t) - \mathbf{x}_i(t)) \times e^{w \times z_{1_i}(t)} \times \cos(2\pi \times z_{1_i}(t)) + \mathbf{x}_{s_i}(t), & i \leq z_2(t) \\ (\mathbf{x}_{s_{z_2(t)}}(t) - \mathbf{x}_i(t)) \times e^{w \times z_{1_i}(t)} \times \cos(2\pi \times z_{1_i}(t)) + \mathbf{x}_{s_{z_2(t)}}(t), & \text{o.w.} \end{cases} \tag{4.25}$$

where $w = 1$.

Lastly, after evaluating the fitness of $\hat{\mathbf{x}}_i$, no matter if the newly generated $\hat{\mathbf{x}}_i(t+1)$ performs better than the previous $\mathbf{x}_i(t)$, the newly generated $\hat{\mathbf{x}}_i(t+1)$ will always be delivered into the next optimization round. After judging whether the newly generated $\hat{\mathbf{x}}_i(t+1)$ are eligible to enter the next optimization round, other step-sizes ($\mathbf{x}_s$, z_{1_i}, z_2) will be updated ($\mathbf{x}_s$ as Eq.4.26, z_{1_i} as Eq.4.27, z_2 as Eq.4.28).

$$\langle \mathbf{x}_i(t+1) \rangle = \mathbf{Sort}(\{\mathbf{x}_i(t)\} \cup \{\mathbf{x}_i(t+1)\}), i = 1 \dots M \tag{4.26}$$

$$z_{1_i}(t+1) = rand \times (-2 - \frac{t+1}{T}) + 1 \tag{4.27}$$

$$z_2(t+1) = \mathbf{Round}(M - (t+1) \times \frac{M-1}{T}) \tag{4.28}$$

Summary The pesudocode of the re-framed MFO algorithm is Algorithm.12

Algorithm 12 Re-framed MFO with population size M; search space $n, [lb_{\mathbf{x}}, ub_{\mathbf{x}}]$; stop condition T; initialization method $Init_{\mathbf{x}}$, optimization method $Opt_{\mathbf{x}}$, treatment C of outliers, and selection S to objective solutions; initialization method $Init_{\Delta}$ and optimization method Opt_{Δ} to step-size Δ.

$t \leftarrow 0$
$\mathbf{X}(t) \leftarrow Init_{\mathbf{x}}(n, M, [lb_{\mathbf{x}}, ub_{\mathbf{x}}])$ as Eq.4.2 ▷ initialize initial population
$F(t) \leftarrow f(\mathbf{X}(t))$ ▷ evaluate
$w \leftarrow Init_{\Delta:w}(w)$ ▷ initialize static numeric step-size w
$\mathbf{X}_s(t) \leftarrow Init_{\Delta:\mathbf{x}_s}(\mathbf{X}(t))$ as Eq.4.22 ▷ initialize dynamic $\mathbf{x}$-relative vector step-size $\mathbf{x}_s$
$z_1(t) \leftarrow Init_{\Delta:z_1}(t, T)$ as Eq.4.23 ▷ initialize dynamic numeric step-size z_1
$z_2(t) \leftarrow Init_{\Delta:z_2}(t, M, T)$ as Eq.4.24 ▷ initialize dynamic numeric step-size z_2
while $\top$ **do**
    $\hat{\mathbf{X}}(t+1) \leftarrow Opt_{\mathbf{x}}(\mathbf{X}(t), \mathbf{X}_s(t), z_1(t), z_2(t), w)$ as Eq.4.26 ▷ generate temporarily updated population
    $\hat{\mathbf{X}}(t+1) \leftarrow C(\hat{\mathbf{X}}(t+1))$ as Eq.4.3 ▷ treatment to outliers
    $F(t+1) \leftarrow f(\hat{\mathbf{X}}_i(t+1))$ ▷ evaluate
    $\mathbf{X}(t+1) \leftarrow S(\mathbf{X}_i(t), \hat{\mathbf{X}}_i(t+1))$ ▷ select and generate finally updated population
    $\mathbf{X}_s(t+1) \leftarrow Init_{\Delta:\mathbf{x}_s}(\mathbf{X}(t), \mathbf{X}(t+1))$ as Eq.4.26 ▷ update dynamic $\mathbf{x}$-relative vector step-size $\mathbf{x}_s$
    $z_1(t+1) \leftarrow Opt_{\Delta:z_1}(t+1, T)$ as Eq.4.27 ▷ update dynamic numeric step-size z_1
    $z_2(t+1) \leftarrow Opt_{\Delta:z_2}(t+1, M, T)$ as Eq.4.28 ▷ update dynamic numeric step-size z_2
    $t \leftarrow t+1$
end while

4.2.5 Re-framed MBO

As discussed in the Table 2.1, the MBO model utilized two kinds if dynamic step-sizes and several static step-sizes to achieve the optimization on the objective solutions. These two dynamic step-sizes are Δ: $\mathbf{x}_g$ and z.

The re-framed MBO model, in the initialization process, will specially initialize $\mathbf{x}_g$ as Eq.4.7, z as Eq.4.29.

$$z(t=0) = \frac{w_4}{t^2} \tag{4.29}$$

where $w_4 = 1$. Next in the optimization process, before the stop condition T meets, the re-framed MBO model firstly directly generate $\hat{\mathbf{x}}_i(t+1)$ as Eq.4.30 with the help of

dealing with outliers C with the help of dealing with outliers C as Eq.4.3.

$$\begin{aligned}
\langle \mathbf{x}_i(t) \rangle &= \textbf{Sort}(\{\mathbf{x}_i(t)\}), i = 1 \dots M \\
{}^{strong}\hat{\mathbf{x}}_{i,n}(t+1) &= \begin{cases} \mathbf{x}_{j,n}(t) \in \langle \mathbf{x}_i(t) \rangle, j \in [1, M^{'}] & , \quad r \times w_2 \leqslant w_1 \\ \mathbf{x}_{j,n}(t) \in \langle \mathbf{x}_i(t) \rangle, j \in (M^{'}, M] & , \quad \text{o.w.} \end{cases} \\
M^{'} &= \lceil w1 \times M \rceil \\
{}^{weak}\hat{\mathbf{x}}_{i,n}(t+1) &= \begin{cases} \mathbf{x}_{g,n}(t), r \geqslant w_1 \\ \begin{cases} \mathbf{x}_{j,n}(t) + z(t) \times \left(\textbf{Lévy}_{i,n} - 0.5 \right), j \in (M^{'}, M], r > w_3 \\ \mathbf{x}_{j,n}(t) \in \langle \mathbf{x}_i(t) \rangle, j \in (M^{'}, M], \text{o.w.} \end{cases} & , \text{o.w.} \end{cases} \\
\{\hat{\mathbf{x}}_i(t+1)\} &= \{{}^{strong}\hat{\mathbf{x}}_i(t+1)\} \cup \{{}^{weak}\hat{\mathbf{x}}_i(t+1)\}
\end{aligned} \tag{4.30}$$

where $w_1 = \frac{5}{12}$, $w_2 = 1.2$, $w_3 = \frac{5}{12}$ and $\textbf{Lévy}_{i,n} = \textbf{Lévy}(d, n, T)$ with $d \sim Exp(2 \times T)$.

Lastly after evaluating the fitness of $\hat{\mathbf{x}}_i(t+1)$, the special selection method is as Eq.4.31. At this moment, the re-framed MBO model has obtained the final generated new solutions after one round optimization. And finally the dynamic step-sizes ($\mathbf{x}_g$, z) are going to be updated before starting the next optimization round, in which the update method to $\mathbf{x}_g$ is as Eq.4.10, the update method to z is as Eq.4.32.

$$\begin{aligned}
&\langle \hat{\mathbf{x}}_i(t+1) \rangle = \textbf{Sort}(\{\hat{\mathbf{x}}_i(t+1)\}), i = 1 \dots M \\
&\mathbf{x}_i(t+1) \in \{\langle \hat{\mathbf{x}}_i(t+1) \rangle, i = 1 \dots M - w_5\} \cup \{\langle \mathbf{x}_i(t) \rangle, i = 1 \dots w_5\}
\end{aligned} \tag{4.31}$$

where $w_5 = 2$.

$$z(t+1) = \frac{w_4}{(t+1)^2} \tag{4.32}$$

Till now, all symbols that are identified the 'where' statement are involved in the static step-sizes.

Algorithm 13 Monarch Butterfly Optimization Algorithm

$t \leftarrow 0$
$Init_{\mathbf{x}}, M$ ▷ Initialization Process
f ▷ Evaluation
$Init_{\Delta}$: $\mathbf{x}_g, z$
while termination criteria are not met **do**
    $Opt_{\mathbf{x}}, C \rightarrow \hat{\mathbf{x}}_i(t+1)$ ▷ Optimization Process
    $f(\hat{\mathbf{x}}_i(t+1))$ ▷ Evaluation
    $S \rightarrow \mathbf{x}_i(t+1)$ ▷ Selection
    $t \leftarrow t+1$
    $Opt_{\Delta} \rightarrow \mathbf{x}_g(t+1), z(t+1)$
end while

4.2.6 Re-framed BOA

As discussed in the Table 2.1, the BOA model utilized two kinds if dynamic step-sizes and several static step-sizes to achieve the optimization on the objective solutions. These two dynamic step-sizes are Δ: $\mathbf{x}_g$ and z.

The re-framed BOA model, in the initialization process, will specially initialize $\mathbf{x}_g$ as Eq.4.7, z as Eq.4.33.

$$z(t=0) = z^0 \tag{4.33}$$

where $z^0 = 0.01$.

Next in the optimization process, before the stop condition T meets, the re-framed BOA model will firstly firstly directly generate $\hat{\mathbf{x}}_i(t+1)$ as Eq.4.34 with the help of dealing with outliers C with the help of dealing with outliers C as Eq.4.3.

$$\hat{\mathbf{x}}_i(t+1) = \begin{cases} \mathbf{x}_i(t) + (r^2 \times \mathbf{x}_g(t) - \mathbf{x}_i(t)) \times z_1(t) \times f\left(\mathbf{x}_i(t)\right)^{w_1} & , \quad r > w_2 \\ \mathbf{x}_i(t) + (r^2 \times \mathbf{x}_j(t) - \mathbf{x}_k(t)) \times z_1(t) \times f\left(\mathbf{x}_i(t)\right)^{w_1} & , \quad \text{o.w} \end{cases} \tag{4.34}$$

where $w_1 = 0.1$, $w_2 = 0.8$. Moreover, $\mathbf{x}_j$ and $\mathbf{x}_k$ are any two neighbors around $\mathbf{x}_i$.

Lastly after evaluating the fitness of $\hat{\mathbf{x}}_i(t+1)$, the special condition used in the selection method is $f(\hat{\mathbf{x}}_i(t+1)) < f(\mathbf{x}_i(t))$. At this moment, the re-framed BOA model has obtained the final generated new solutions after one round optimization. And finally the dynamic step-sizes are going to be updated before starting the next optimization round,

in which the update method to $\mathbf{x}_g$ is as Eq.4.10, the update method to z is as Eq.4.35.

$$z(t+1) = z(t) + \frac{0.025}{z(t) \times T} \tag{4.35}$$

Till now, all symbols that are identified after the 'where' statement are involved in the static step-sizes.

Algorithm 14 Butterfly Optimization Algorithm

1: $t \leftarrow 0$
2: $Init_{\mathbf{x}}, M$ ▷ Initialization Process
3: f ▷ Evaluation
4: $Init_{\Delta}$: $\mathbf{x}_g, z$
5: **while** termination criteria are not met **do**
6: $Opt_{\mathbf{x}}, C \rightarrow \hat{\mathbf{x}}_i(t+1)$ ▷ Optimization Process
7: $f(\hat{\mathbf{x}}_i(t+1))$ ▷ Evaluation
8: $S \rightarrow \mathbf{x}_i(t+1)$ ▷ Selection
9: $t \leftarrow t+1$
10: $Opt_{\Delta} \rightarrow \mathbf{x}_g(t+1), z(t+1)$
11: **end while**

4.2.7 Re-framed PSO

Description As discussed in the Table 2.1, the PSO model utilized three kinds of dynamic step-sizes and several static step-sizes to achieve the optimization on the objective solutions. These two dynamic step-sizes are Δ: $\mathbf{y}_i$, $\mathbf{x}_{i_p}$, $\mathbf{x}_g$.

The re-framed PSO model, in the initialization process, will specially initialize $\mathbf{y}_i$ as Eq.4.8, $\mathbf{x}_{i_p}$ as Eq.4.6, $\mathbf{x}_g$ as Eq.4.7. The static step-sizes will be also set at this moment.

Next in the optimization, before the stop condition T meets, the re-framed PSO model will firtsly generate the $\mathbf{y}_i$ as Eq.4.36, then generate $\hat{\mathbf{x}}_i(t+1)$ as Eq.4.37 with the help of dealing with outliers C with the help of dealing with outliers C as Eq.4.3.

$$\mathbf{y}_i(t+1) = w_1 \times \mathbf{y}_i(t) + \mathcal{U}(0, w_2) \times (\mathbf{x}_{i_p}(t) - \mathbf{x}_i(t)) + \mathcal{U}(0, w_3) \times (\mathbf{x}_g(t) - \mathbf{x}_i(t)) \tag{4.36}$$

where $w_1 = 0.73$, $w_2 = 1.49$, $w_2 = 1.49$.

$$\hat{\mathbf{x}}_i(t+1) = \mathbf{x}_i(t) + \mathbf{y}_i(t+1) \tag{4.37}$$

Lastly after evaluating the fitness of $\hat{\mathbf{x}}_i$, the special condition used in the selection method is $\forall$. At this moment, the re-framed PSO model has obtained the final generated new

solutions after one round optimization. And finally dynamic step-sizes ($\mathbf{x}_{i_p}$, $\mathbf{x}_g$) are going to be updated ($\mathbf{x}_{i_p}$ as Eq.4.6, $\mathbf{x}_g$ as Eq.4.10) before starting the next optimization round.

Till now, all symbols that are identified after the 'where' state are involved in the static step-sizes.

Algorithm 15 Particle Swarm Optimization

1: $t \leftarrow 0$
2: $Init_{\mathbf{x}}, M$ ▷ Initialization Process
3: f ▷ Evaluation
4: $Init_{\Delta}$: $\mathbf{y}_i, \mathbf{x}_{i_p}, \mathbf{x}_g$
5: **while** termination criteria are not met **do**
6: $\quad Opt_{\Delta} \rightarrow \mathbf{y}_i(t+1)$ ▷ Optimization Process
7: $\quad Opt_{\mathbf{x}}, C \rightarrow \hat{\mathbf{x}}_i(t+1)$
8: $\quad f(\hat{\mathbf{x}}_i(t+1))$ ▷ Evaluation
9: $\quad S \rightarrow \mathbf{x}_i(t+1)$ ▷ Selection
10: $\quad t \leftarrow t+1$
11: $\quad Opt_{\Delta} \rightarrow \mathbf{x}_{i_p}(t+1), \mathbf{x}_g(t+1)$
12: **end while**

Summary

4.3 Conclusion

Re-framed psuedo-codes[3] in the Section 4.2 show that **these seven algorithms can share a complete same model template Algorithm.8, no matter how different their original models are**. The Section 4.3 will firstly describe this unified framework in the high level overview, before detailing these broad observations.

In the high level overview, the Figure 4.1 illustrates that in the unified model template, there are two processes (Initialization Process and Optimization Process). One process has only one evaluation f. The f in the Initialization Process happens immediately after $Init_{\mathbf{x}}, M$, which is similar to that the f in the Optimization Process happens immediately after $Opt_{\mathbf{x}}, C$, because both of them generate a population. The main update strategy happens in the $Opt_{\mathbf{x}}, C$ in the Optimization Process, and algorithms are mainly distinguished to each other here. There are two main aspects (Δ, S) that will

[3]BA: Algorithm.9. GOA: Algorithm.10. CSA: Algorithm.11. MFO: Algorithm.12. MBO: Algorithm.20. BOA: Algorithm.14. PSO: Algorithm.15.

seriously affect the quality of optimization in the $Opt_{\mathbf{x}}$, in which the Δ impact is the *Link-1 to 3,5 to 16*, the S impact is the *Link-4,17*. Sometimes, some algorithms, such as BOA, MFO, will also utilize f fitness in their $Opt_{\mathbf{x}}$ method, as the *Link-18*.

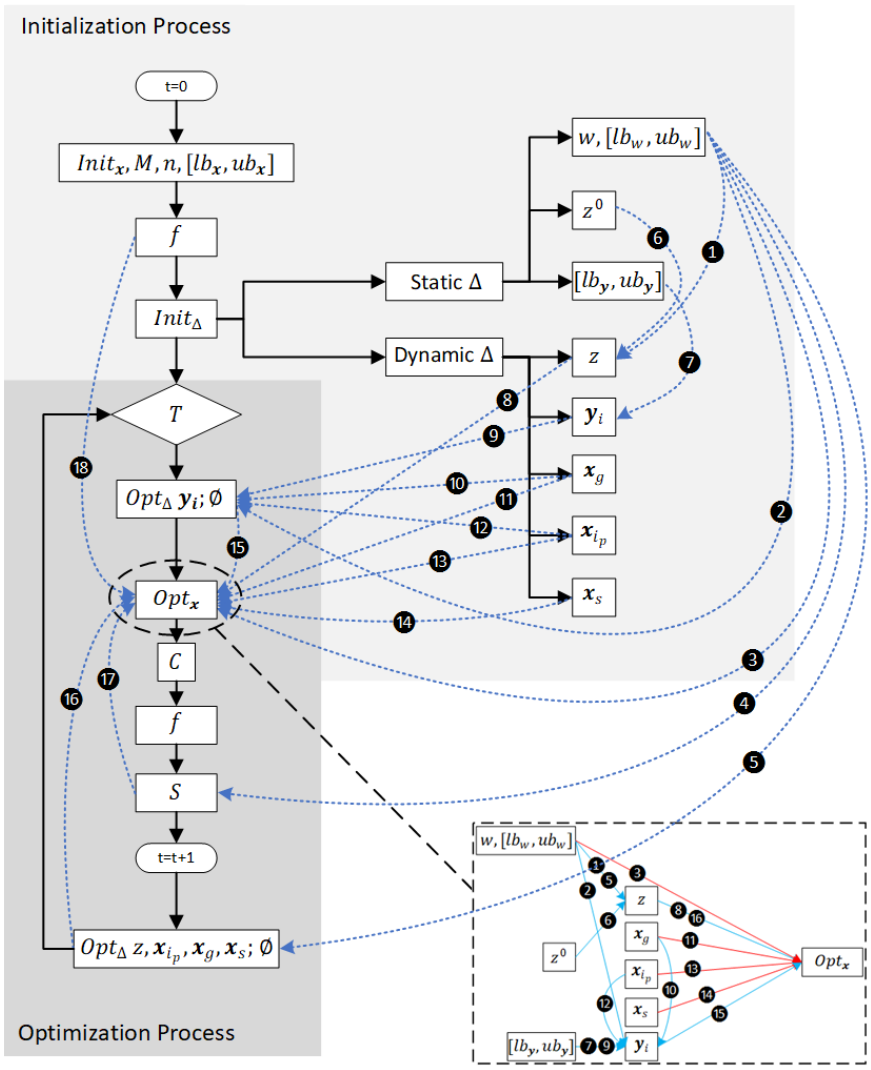

Figure 4.1: Unified framework with relationships amongst their components

The Figure 4.1, especially the small sub-picture in the bottom right corner, also illustrates that different algorithms will prepare different step-sizes Δ to participate in their optimization strategy $Opt_{\mathbf{x}}$. **Some participation is direct (bright red links)**, such as *Link-3,*

11, 13, 14 in which some static Δ: $w, [lb_w, ub_w]$(commonly known as hyper-parameters) and some dynamic Δ: $\mathbf{x}_{i_p}, \mathbf{x}_g, \mathbf{x}_s$, are directly added in the $Opt_{\mathbf{x}}$. **Some participation is indirect (bright blue links)**, such as *Link-1, 5, 2, 6, 7, 9* in which these static Δ indirectly impact $Opt_{\mathbf{x}}$ by impacting some dynamic Δ: $z, \mathbf{y}$. Meanwhile, besides direct impact, dynamic Δ: $\mathbf{x}_{i_p}, \mathbf{x}_g$ can also impact $Opt_{\mathbf{x}}$ by impacting the dynamic Δ: $\mathbf{y}$, such as *Link-10, 12*.

Moreover, besides the Δ impact, the selection method S will also impact the $Opt_{\mathbf{x}}$, such as *Link-4, 17* in which static Δ firstly impacts the selection method, then the selection method impacts $Opt_{\mathbf{x}}$.

In the minor details level, the Table 4.1 illustrates the entire methods existed in these seven algorithms for each components in the unified framework, and points out whether these methods are allowed to be customized in the future. For example, $Init_{\mathbf{x}}, M$, f, $Init_{\Delta}$: $\mathbf{x}_{i_p}$, $\mathbf{x}_g$ and Opt_{Δ}: $\mathbf{x}_{i_p}$, $\mathbf{x}_g$ are not allowed to be customized, and they have to be kept same in any one of these seven algorithms. However, other components have much freedom in designing a customized method, such as $Init_{\Delta}$: $\mathbf{y}_i$, $\mathbf{x}_s$, z, w, Opt_{Δ}: $\mathbf{y}_i$, $\mathbf{x}_s$, z, $Opt_{\mathbf{x}}$, C and S, in which C has two options $C1$, $C2$, and S has four options $S1$, $S2$, $S3$, $S4$.

Until now, we can conclude, as the Table 4.2 displayed, at least in these seven algorithms, **algorithms are different because of the types of components they utilized and the design of these components, but the total types of components that can be selected and the order in which the components are arranged are totally same**.

A general utilization of this conclusion is to design an auto-designer for algorithms. For example, there are many open Python libraries [4, 6, 9, 13, 23, 30] for designing evolutionary and genetic algorithms (EA), in which various 'crossover', 'mutation' and 'selection' are freely combined with each other to generate new algorithms. However, this researching road might not be such smooth in the UNIOA for swarm-based algorithms, which means **it might be not such intuitive to select several existing methods to combine a new algorithm in the UNIOA**. This is because swarm-based algorithms have more complicated mechanisms compared to EA, for example, the gene operations are limited but social activities in the real world are very various. In more details, in the most well-known EA common framework, the population will walk through 'crossover', 'mutation', 'selection'. One interesting observation to note here is that *you definitely will obtain a complete updated population even only using one of them, if ignoring the convergence*. However, in the swarm-based algorithm, it is hard to organized such a similar framework. For example, in the Table 4.1, in the *Index-4-*$Init_{\Delta}$*-*z, there are five different methods, however, it doesn't mean the method for BA is able to replace the method for GOA.

One reason is the number of z is different, another reason is the positive impact of z for BA might be negative impact for GOA in the $Opt_{\mathbf{x}}$. How to use *Index-4*, *Index-6* and *Index-11* only depend on how the *Index-7*-$Opt_{\mathbf{x}}$ is designed.

Therefore, compared to freely packaging a new algorithm by selecting from existing methods in the EA framework, **the UNIOA emphasizes creating new methods from source**, which means the $Opt_{\mathbf{x}}$ is the beginning point to select or create other accessories. Moreover, compared to the rich selection pool in the EA framework, **the UNIOA emphasizes creating new $Opt_{\mathbf{x}}$ and its accessories** which means learning ideas from existing methods is more important than directly using existing methods.

Moreover, because C and S meet the condition that *you definitely will obtain a complete updated population even only using one of them, if ignoring the convergence*, $C1$, $C2$ and $S1$, $S2$, $S3$, $S4$ are also allowed to be freely utilized in any one algorithm. The details about these methods are detailed as follows:

- Two options $C1$, $C2$ to deal with outliers.

$$\mathbf{x}_{i,n}^{\text{fixed}}(t+1) = \begin{cases} ub_x & , \quad \mathbf{x}_{i,n}(t+1) > ub_{\mathbf{x}} \\ \mathbf{x}_{i,n}(t+1) & , \quad \text{o.w} \\ lb_x & , \quad \mathbf{x}_{i,n}(t+1) < lb_{\mathbf{x}} \end{cases} \tag{C1}$$

$$\mathbf{x}_{i,n}^{\text{fixed}}(t+1) = \begin{cases} \mathbf{x}_{i,n}(t) & , \quad \mathbf{x}_{i,n}(t+1) < lb_{\mathbf{x}} \text{ or } \mathbf{x}_{i,n}(t+1) > ub_{\mathbf{x}} \\ \mathbf{x}_{i,n}(t+1) & , \quad \text{o.w} \end{cases} \tag{C2}$$

- Four options $S1$, $S2$, $S3$, $S4$ to select updated population from temporary generated population.

$$\mathbf{x}_i(t+1) = \hat{\mathbf{x}}_i(t+1) \tag{S1}$$

$$\mathbf{x}_i(t+1) = \begin{cases} \hat{\mathbf{x}}_i(t+1) & , \quad f(\hat{\mathbf{x}}_i(t+1)) < f(\mathbf{x}_i(t)) \\ \mathbf{x}_i(t) & , \quad \text{o.w} \end{cases} \tag{S2}$$

$$\mathbf{x}_i(t+1) = \begin{cases} \hat{\mathbf{x}}_i(t+1) & , \quad f(\hat{\mathbf{x}}_i(t+1)) < f(\mathbf{x}_i(t)) \text{ or } r > z_2(t) \\ \mathbf{x}_i(t) & , \quad \text{o.w} \end{cases} \tag{S3}$$

$$\begin{aligned} &\langle \hat{\mathbf{x}}_i(t+1) \rangle = \textbf{Sort}(\{\hat{\mathbf{x}}_i(t+1)\}), i = 1 \dots M \\ &\mathbf{x}_i(t+1) \in \{\langle \hat{\mathbf{x}}_i(t+1) \rangle, i = 1 \dots M - w_5\} \cup \{\langle \mathbf{x}_i(t) \rangle, i = 1 \dots w_5\} \end{aligned} \tag{S4}$$

Table 4.1: In the unified framework, for each component, the existing methods collected in these seven algorithms and whether they could be customized in the future.

	Index	Components		Existing Method	Customization
Initialization	1	t=0			
	2	$Init_{\mathbf{x}}, M$		$\mathbf{x}_i(t=0) = \boldsymbol{\mathcal{U}}(lb_{\mathbf{x}}, ub_{\mathbf{x}}), i = 1, 2, \ldots, M$	no
	3	$f(\mathbf{x}_i(t=0)), i = 1, 2, \ldots, M$			no
	4	$Init_{\Delta}$	$\mathbf{y}_i$	$\mathbf{y}_i(t=0) = \boldsymbol{\mathcal{U}}(lb_{\mathbf{y}}, ub_{\mathbf{y}}), i = 1, 2, \ldots, M$	yes
			$\mathbf{x}_{i_p}$	$\mathbf{x}_{i_p}(t=0) = \mathbf{x}_i(t+1), i = 1, 2, \ldots, M$	no
			$\mathbf{x}_g$	$\mathbf{x}_g(t=0) = \textbf{Min}(\{\mathbf{x}_i(t=0)\}), i = 1, 2, \ldots, M$	no
			$\mathbf{x}_s$	Eq.4.22(MFO)	yes
			z	Eq.4.12, Eq.4.13(BA). Eq.4.18(GOA). Eq.4.23, Eq.4.24(MFO). Eq.4.29(MBO). Eq.4.33(BOA)	yes
			w	commonly known as hyper-parameters	yes
Optimization	5	Stop condition T			
	6	Opt_{Δ}	$\mathbf{y}_i(t+1)$	Eq.4.14(BA). Eq.4.36(PSO)	yes
	7	$Opt_{\mathbf{x}}, C \rightarrow \hat{\mathbf{x}}_i(t+1)$		Eq.4.15(BA). Eq.4.19(GOA). Eq.4.25(MFO). Eq.4.30(MBO). Eq.4.34(BOA). Eq.4.37(PSO) $C1$ Eq.4.21(CSA) $C2$	yes
	8	$f(\hat{\mathbf{x}}_i(t+1))$			no
	9	$S \rightarrow \mathbf{x}_i(t+1)$		$S1$(GOA, CSA, MFO, PSO). $S2$(BOA). $S3$(BA). $S4$(MBO)	yes
	10	t=t+1			
	11	Opt_{Δ}	$z(t+1)$	Eq.4.16, Eq.4.17(BA). Eq.4.20(GOA). Eq.4.27, Eq.4.28(MFO). Eq.4.32(MBO). Eq.4.35(BOA)	yes
			$\mathbf{x}_{i_p}(t+1)$	$\mathbf{x}_{i_p}(t+1) = \textbf{Min}(\mathbf{x}_{i_p}(t), \mathbf{x}_i(t+1)), i = 1, 2, \ldots, M$	no
			$\mathbf{x}_g(t+1)$	$\mathbf{x}_g(t+1) = \textbf{Min}(\mathbf{x}_g(t) \cup \{\mathbf{x}_i(t+1)\}), i = 1, 2, \ldots, M$	no
			$\mathbf{x}_s(t+1)$	Eq.4.26(MFO)	yes

Table 4.2: How these components in Table 2.1 are structured in these seven algorithms.

Progress	Index	Components		BA	GOA	CSA	MFO	MBO	BOA	PSO
Initialization	1	t=0								
	2	$Init_{\mathbf{x}}, M$		Eq.4.2						
	3	$f(\mathbf{x}(t))$								
	4	$Init_{\Delta}$	$\mathbf{y}_i$	Eq.4.11	×	✓	×	×	×	Eq.4.8
			$\mathbf{x}_{i_p}$	×	×	Eq.4.6	×	×	×	Eq.4.6
			$\mathbf{x}_g$	Eq.4.7	Eq.4.7	×	×	Eq.4.7	Eq.4.7	Eq.4.7
			$\mathbf{x}_s$	×	×	×	Eq.4.22	×	×	×
			z	Eq.4.12, Eq.4.13	Eq.4.18	×	Eq.4.23, Eq.4.24	Eq.4.29	Eq.4.33	×
			w	✓	✓	✓	✓	✓	✓	✓
Optimization	5	Stop condition T								
	6	Opt_{Δ}	$\mathbf{y}_i(t+1)$	Eq.4.14	×	×	×	×	×	Eq.4.36
	7	$Opt_{\mathbf{x}}, C \rightarrow \hat{\mathbf{x}}_i(t+1)$		Eq.4.15, $C1$	Eq.4.19, $C1$	Eq.4.21, $C2$	Eq.4.25, $C1$	Eq.4.30, $C1$	Eq.4.34, $C1$	Eq.4.37, $C1$
	8	$f(\hat{\mathbf{x}}_i(t+1))$								
	9	$S \rightarrow \mathbf{x}_i(t+1)$		$S3$	$S1$	$S1$	$S1$	$S4$	$S2$	$S1$
	10	t=t+1								
	11	Opt_{Δ}	$z(t+1)$	Eq.4.16, Eq.4.17	Eq.4.20	×	Eq.4.27, Eq.4.28	Eq.4.32	Eq.4.35	×
			$\mathbf{x}_{i_p}(t+1)$	×	×	Eq.4.9	×	×	×	Eq.4.9
			$\mathbf{x}_g(t+1)$	Eq.4.10	Eq.4.10	×	×	Eq.4.10	Eq.4.10	Eq.4.10
			$\mathbf{x}_s(t+1)$	×	×	×	Eq.4.26	×	×	×

Chapter 5

Experimental Setup of Benchmark Study

5.1 Motivation for experiments

As discussed in previous chapters, we show that these seven selected nature-inspired algorithms can be rewritten in terms of the unified framework (see Chapter 4). However, **to demonstrate the reliability of the unified framework, theoretical evidence alone is not enough, and empirical evidence is also essential**. Therefore, this chapter introduces our experiment plans to provide empirical evidence.

We designed the experiment as follows:

(1) **Research question**: We want to know if the unified framework we designed for these seven algorithms could adequately replace their original framework. Specifically, we want to know whether the same algorithm in two different representations behaves identically on solving a set of questions.

(2) **Define main variables**: The algorithm representation shall be the independent variable, and the performance of algorithms shall be the dependent variable.

(3) **Our hypothesis**: The unified framework we designed can correctly cover these seven algorithms. Specifically, for these seven algorithms, the performances of each algorithm between written in the unified framework and written in its original framework are the same on solving a set of questions.

(4) **Experimental treatment**: We will implement both representations of each algorithm in Python. Each algorithm in both frameworks will solve the same set of optimization problems.

(5) **Measure dependent variable**: The performances of each algorithm in both frameworks will be measured by the number of problem evaluations [14]. The comparison between each algorithm's two representations will be measured by a statistical test method.

Therefore, we need a tool —IOHprofiler [8] —that can access enough optimization problems for testing algorithms, and allows the number of problem evaluations to be the criteria of algorithms' performance. Furthermore, considering these seven algorithms are created for solving continuous optimization problems [15–19, 33, 34], we will select the BBOB problem set for benchmarking algorithms. Specifically, the BBOB set merged into the IOHprofiler framework contains 24 different single-objective continuous optimization problems searching for a minimal solution [10].

The tool IOHprofiler [8, 32] consists of an experimental part and a post-processing part. The experimental part is used to generate time-series running data, and the post-processing part will quantify the algorithm performance by analyzing these generated data. **In the experimental part**, we will manually implement each algorithm in both representations. For example, we will define the stop condition, the dimension of problems, the number of instances of each problem, and the number of runs of each instance [1]. **In the post-processing part**, we will compare the performance of each algorithm in the unified framework and its original framework by observing their ERT plots obtained from IOHanalyzer [2]. Here, because we are mostly interested in how many evaluations the algorithm will take when faced with different target values, the ERT plot visualizes how many evaluations each algorithm in each framework will take to reach a given target value, where the x-axis denotes various target values whose minimum value is 10^{-8}, and the y-axis denotes the number of evaluations called Expected Running Time (ERT) whose maximum number is $n \times 10^4$. n here is the dimension of problems.

The IOHanalyzer also provides R programming interfaces in which the area under the ECDF curve for each algorithm in each framework on solving each problem will be collected for further analysis. Here, ECDF for each problem is an aggregated Empirical Cumulative Distribution over a set of target values. The set of target values obtained from IOHanalyzer is $\{10^i \mid i = 2, 1.8, 1.6, ..., -8\}$. Furthermore, repeating experiments

[1] The instance of each continuous optimization problem is its variants by multiplicative shift and/or additive shit. Please find more details in [8].

[2] https://iohanalyzer.liacs.nl/

multiple times is essential to ensure the experimental results are scientific. Therefore, we conclude how we set up and use the IOHprofiler environment in Table 5.1.

Table 5.1: Setups and usages in IOHproflier.

Environment	Setups	Usages
IOHexperimenter	· minimum target value: 10^{-8}.	· generate running data of each problem over multiple instances and multiple runs.
	· maximum number of evaluations: $n \times 10^4$.	
	· dimensions of problems: $n = 5, 20$.	
	· the number of instances for each problem: 5.	
	· the number of runs for each instance: 5.	
IOHanalyzer	· web-based GUI	· observe ERT plots.
	· R programming interfaces.	· calculate AUCs of ECDFs.

As shown in Table 5.1, when measuring experimental results, ERT plots will describe differences of each algorithm performance in the unified framework and its original framework in the view of quality. We also need to measure the differences in the view of quantity. The AUC value of ECDF curve will measure each algorithm's performance in each framework. Precisely, each algorithm in each framework will have 24 AUC values throughout 24 optimization problems obtained in IOHexperimenter. Considering we are interested in differences between each algorithm performance in our designed unified framework and its original framework, also, both of them (the algorithm in our designed unified framework and the same algorithm in its original framework) will solve the same 24 problems, a kind of paired sample test statistical procedure will be preferred.

Lastly, as mentioned in previous paragraphs, only two measurement indicators (ERT and AUC of ECDF) will be used for analyzing experimental results. The ERT plot will visually display how the number of evaluations increases (in the y-axis) as the target value decreases (in the x-axis). In the ERT plot, one line denotes the performance of one algorithm in one framework over one dimension on one problem. The AUC values of

ECDF will quantify the probability that each algorithm in each framework successfully reaches the target value. Each AUC value denotes such a probability for one algorithm in one framework over one dimension on one problem. Here, we want to clarify that the IOHprofiler environment has its own mechanism to deal with multiple instances and runs. Simply speaking, this mechanism is a kind of average calculation, but more accurate details can be found in [8, 32].

5.2 Summary

In this chapter, we introduced the purpose of doing such practical experiments to give empirical evidence on whether these seven algorithms can safely be rewritten in our designed unified framework.

For achieving such experiments, we will test the performance of each algorithm in each framework with the help of the IOHprofiler benchmarking tool. Although the IOHprofiler provides various statistical ways to measure the performance of algorithms, we will focus on two measurements: ERT plots in the view of quality and AUC values of ECDF in the view of quantity.

Moreover, the AUC values will be further analyzed by a kind of paired sample test statistical method that will be determined in actual experiments (see Chapter 5).

Chapter 6

Experimental Results

6.1 Introduction

In Chapter 4, We point out the need to further demonstrate the reliability of the unified framework UNIOA with its general terminologies through actual experiments. We also briefly describe the experimental environment with its settings shown in Table 5.1. In this chapter, firstly, we detail how many kinds of experiments are displayed to achieve the purpose of experiments. Next, we answer the question mentioned in Section 4.2 about what kind of statistics test method will be used in this work. Lastly, we display the experimental results and give discussions.

Because we are interested in how close the performances of one algorithm are in its original framework and our unified framework, the framework structure shall be the only aspect affecting the algorithm performance and any other aspects with side effects on algorithms' performance shall be eliminated. Therefore, **the first group of experiments is to test whether syncE/asyncE and syncG/asyncG will significantly affect the algorithm performance, and significant effects shall be eliminated** [1]. The experimental results of this group are displayed in Section 5.2.

After eliminating some aspects with side effects, **the second group of experiments is to compare the performances of each algorithm in its original framework and our unified framework**. The experimental results of this group are displayed in **??**.

[1] In a variety of experimental attempts at measuring the performance of algorithms, we find these two aspects (syncE/asyncE: the evaluation method is synchronous or asynchronous. syncG/asyncG: the method of calculating the global best individual is synchronous or asynchronous.) that might significantly affect the algorithm performance. Therefore, we do experiments to make sure the influence of them.

Then, for providing convincing discussion, as discussed in Chapter 4, instead of a common two-sample test, a paired test shall be preferred. Moreover, rather than the popular paired t-test, we prefer the Wilcoxon signed-rank test with a confidence level of 95% because samples in our case are not normally distributed. ***The null hypothesis*** H_0 ***in our experiments is the difference between two algorithms' performances is on average zero for a random BBOB optimization problem.*** Here, the two algorithms mean two versions of one algorithm. For example, they could be (1) syncE bat-algorithm and asyncE bat-algorithm. (2) syncG bat-algorithm and asyncG bat-algorithm. (3) bat-algorithm in its original framework and bat-algorithm in our unified framework. Because we studied seven different algorithms, such experiments will be repeated seven times.

After observing the performances of each algorithm in its original framework and our unified framework, we are interested in the actual application of this framework. In **??**, we achieve our designed framework into a Python package named UNIOA in which various ideas of swarm-based optimization algorithms can be prototyped in the unified framework, evaluated with the help of the IOHprofiler environment, and compared with each other.

In the **??**, we test our UNIOA package to compare these seven algorithms. Therefore, **the third group of experiments is to compare the performances of algorithms in our unified framework for giving an overview understanding of these algorithms**.

Lastly, to make the discussion in the following sections clearer and easier to understand, we use 'UNIOA' to denote the algorithm in the unified framework with unified terminologies and 'ORIGINAL' (abbr. orig) to denote the algorithm model in its original framework. The chapter is concluded in **??**.

6.2 Experiments for avoiding side effects

6.2.1 Results

In the first group of experiments, we are interested in whether different ways of calculating the fitness and the global best individual will significantly impact the performance of algorithms. In many experimental attempts, we found that there are two ways to calculate fitness:

- asynchronous Evaluation (abbr. asyncE): each time updating one individual, this individual's fitness is immediately calculated.

- synchronous Evaluation (abbr. syncE): only after the entire population is updated, the fitness of each individual in the entire population is then calculated simultaneously.

There are also two ways to calculate the global best one individual $\mathbf{x}_g$:

- asynchronous $\mathbf{x}_g$ (abbr. asyncG): each time updating one individual, the $\mathbf{x}_g$ is updated by comparing with this one individual.
- synchronous $\mathbf{x}_g$ (abbr. syncG): only after the entire population is updated, the $\mathbf{x}_g$ is updated by comparing with the entire updated population.

Therefore, for avoid side effects from asyncE/syncE and asyncG/syncG, we constructed 33 sub-experiments [2] in which:

- Each UNIOA has two options: evaluation is synchronous or asynchronous, but $\mathbf{x}_g$ is always synchronous.
- Each ORIGINAL has three options: when the evaluation is synchronous, the $\mathbf{x}_g$ is synchronous, but when the evaluation is asynchronous, the $\mathbf{x}_g$ is synchronous or asynchronous.

From Figure 6.1 in which y-axis is each algorithm whose evaluation method could be synchronous or asynchronous and its corresponding AUC values obtained in IOHanalyzer are shown on x-axis, it is observed that the two distributions of 24 AUC values for syncE algorithm and asyncE algorithm are the same. Even the outliers from syncE algorithm and asyncE algorithm are also significantly near each other. Same observations can be found when dimension n is 5 and when dimension n is 20. Meanwhile, in Table 6.1, each case represents one comparison between one algorithm whose evaluation method is synchronous or is asynchronous. We find that all p values are larger than 5%. It means

[2] Generally, for the five algorithms who use $\mathbf{x}_g$, there should be $5 \times 2 \times 2 \times 2 = 40$ sub-experiments in which each algorithm has two frameworks, and each framework has two evaluation options, and each evaluation option has two $\mathbf{x}_g$ calculation options. However, we found it was impossible to calculate $\mathbf{x}_g$ asynchronously when the evaluation was synchronous. The reason is when the evaluation is already synchronous, it means the *for loop* for $\mathbf{x}_i$ already stops, and it is impossible for the $\mathbf{x}_g$ to come back into the *for loop*. Moreover, for UNIOA, when the evaluation is asynchronous, it is meaningless to set the $\mathbf{x}_g$ asynchronous also, which is because in UNIOA, other components also have to be changed when the evaluation and $\mathbf{x}_g$ are simultaneously asynchronous, then it is impossible to decide which component exactly affects the algorithm performance. Therefore, there are only $5 \times 2 \times 2 \times 2 - 5 \times 2 \times 1 \times 1 - 5 \times 1 \times 1 \times 1 = 25$ for five algorithms who use $\mathbf{x}_g$. Meanwhile, for the two algorithms who didn't use $\mathbf{x}_g$, there will be $2 \times 2 \times 2 = 8$ experiments in which each algorithm has two framework, and each framework has two evaluations. Therefore, there are total $25 + 8 = 33$ experiments. Table 7.15 and Table 7.16 lists details about how these 33 sub-experiments are constructed.

the data obtained from syncE algorithm and asyncE algorithm does not reject the hull hypothesis, no matter which framework it belongs to and no matter which dimension it is in.

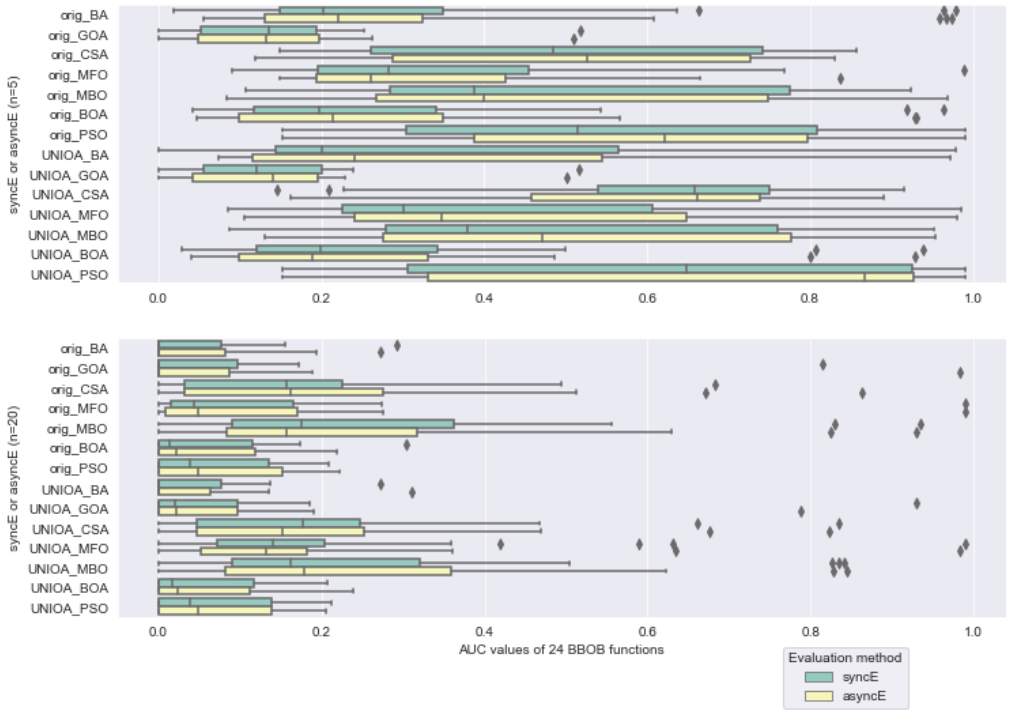

Figure 6.1: Distribution of 24 AUC values for paired algorithm performances in syncE or asyncE, when $n = 5$ or $n = 20$.

Table 6.1: P-values of Wilcoxon signed-rank test for difference on average between performances of paired algorithms in syncE or asyncE for a random optimization function, when $n = 5$ or $n = 20$.

Case	p-value (n=5)	p-value (n=20)
orig_BA_syncE_or_asyncE_syncG	0.3313349065031326	0.22886900026455015
orig_GOA_syncE_or_asyncE_syncG	0.6261109162613098	0.6673653270475619

continued . . .

. . . continued

Case	p-value (n=5)	p-value (n=20)
orig_CSA_syncE_or_asyncE	0.6475683676310555	0.8862414820514412
orig_MFO_syncE_or_asyncE	0.2530979089471155	0.8409995729722781
orig_MBO_syncE_or_asyncE_syncG	0.31731050786291415	0.17469182698689223
orig_BOA_syncE_or_asyncE_syncG	0.7750969621959847	0.2773986245500173
orig_PSO_syncE_or_asyncE_syncG	0.954431397113681	0.19449921074798193
UNIOA_BA_syncE_or_asyncE_syncG	0.24142655338204444	0.5968445170755943
UNIOA_GOA_syncE_or_asyncE_syncG	0.6465582592577419	0.6233436223982587
UNIOA_CSA_syncE_or_asyncE	0.9090113066460508	0.9430276454464167
UNIOA_MFO_syncE_or_asyncE	0.6475683676310555	0.38327738184229543
UNIOA_MBO_syncE_or_asyncE_syncG	0.265156633625956	0.24697273165906564
UNIOA_BOA_syncE_or_asyncE_syncG	0.09749059620220792	0.629275096131297
UNIOA_PSO_syncE_or_asyncE_syncG	0.954431397113681	0.4655179892460418

From Figure 6.2 in which y-axis is each algorithm whose $\mathbf{x}_g$ calculation method could be synchronous or asynchronous and its corresponding AUC values obtained in IOHanalyzer are shown on x-axis, it is observed that the two distributions of 24 AUC values for syncG algorithm and for asyncG algorithm are varying from algorithms. For example, the difference of paired performances between syncG BA and asyncG BA is significant when dimension n is 5 and when dimension n is 20, however, this kind of difference shown in Figure 6.2 is not significant in other algorithms. From Table 6.2 in which each case represents one comparison between each algorithm whose $\mathbf{x}_g$ calculation method is synchronous or is asynchronous, it is observed that when dimension n is 5, the data obtained from orig_GOA algorithm rejects the hull hypothesis which means the difference between syncG and asyncG orig_BA algorithm's performance is significant. Such same observation is also found in orig_GOA algorithm when dimension n is 20.

Considering the difference is not significant in other cases, further analysis is needed to make the role of $\mathbf{x}_g$ explicit in unsignificant cases (orig_MBO, orig_BOA, orig_PSO) compared to significant cases (orig_BA, orig_GOA). From Table 6.3 that lists the way of

each algorithm using $\mathbf{x}_g$ in ORIGINAL framework and UNIOA framework, it is observed that the effect of $\mathbf{x}_g$ is different in these algorithms, although $\mathbf{x}_g$ affects the quality of optimization in all of these algorithms. For example, $\mathbf{x}_g$ is directly added to objective solution $\mathbf{x}_i$ in BA and GOA, but in MBO, BOA and PSO, $\mathbf{x}_g$ is scaling down by multiplying a very small decimal (BOA), or by subtracting a larger value (PSO), or by reducing the probability of using $\mathbf{x}_g$ (MBO).

Therefore, it is preferred to conclude that **syncG and asyncG algorithms have different performance on average for a random optimization BBOB function in ORIGINAL framework, no matter which dimension it is in**.

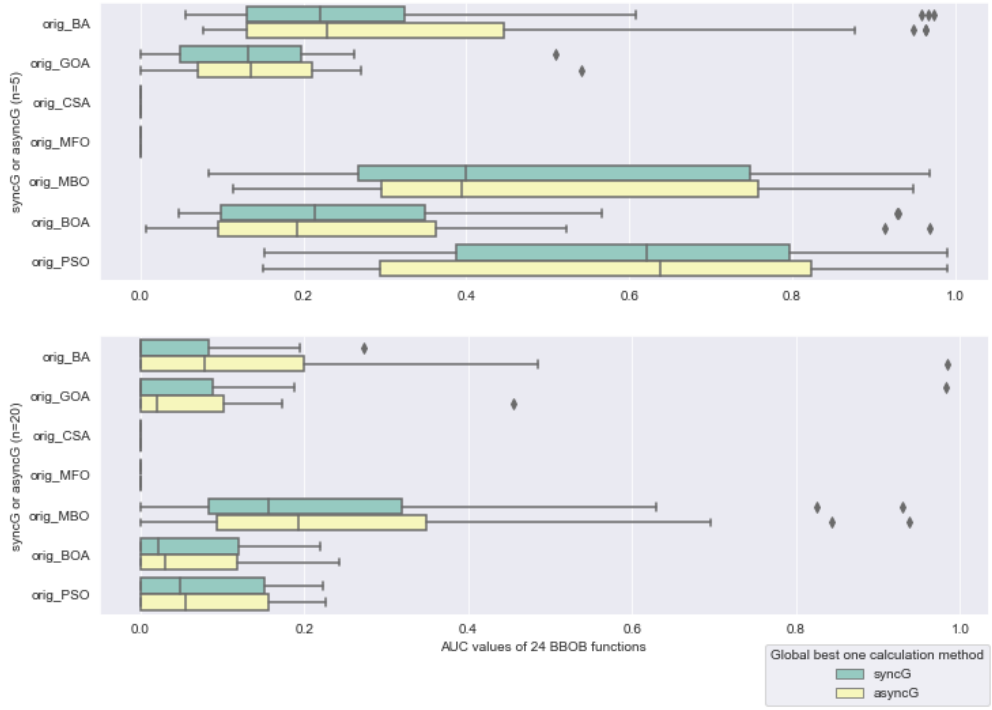

Figure 6.2: Distribution of 24 AUC values for paired algorithm performances in syncG or asyncG, when $n = 5$ or $n = 20$.

Table 6.2: P-values of Wilcoxon signed-rank test for difference on average between performances of paired algorithms in syncG or asyncG for a random optimization function, when $n = 5$ or $n = 20$.

Case	p-value (n=5)	p-value (n=20)
orig_BA_asyncE_syncG_or_asyncG	0.22459133607647186	0.004638292309835263
orig_GOA_asyncE_syncG_or_asyncG	0.0042199644589382525	0.17318594569671153
CSA	NO USE $\mathbf{x}_g$	
MFO	NO USE $\mathbf{x}_g$	
orig_MBO_asyncE_syncG_or_asyncG	0.19854279368666194	0.07411601304083099
orig_BOA_asyncE_syncG_or_asyncG	0.8192020334011836	0.2773986245500173
orig_PSO_asyncE_syncG_or_asyncG	0.44045294529422474	0.964388671614557

Table 6.3: How each algorithm uses $\mathbf{x}_g$ in ORIGINAL framework and UNIOA framework.

H_0	Algorithm	$\mathbf{x}_g$ role in ORIGINAL	$\mathbf{x}_g$ role in UNIOA
reject	BA	Eq.2.5, Eq.2.6	Eq.4.15
reject	GOA	Eq.2.10	Eq.4.19
-	CSA	NO USE $\mathbf{x}_g$	
-	MFO	NO USE $\mathbf{x}_g$	
not reject	MBO	Step (6)	Eq.4.30
not reject	BOA	Eq.2.28	Eq.4.34
not reject	PSO	Eq.2.34	Eq.4.36

6.2.2 Conclusion

According to the experimental results in Section 5.2, we can conclude that the different effects from asyncE and syncE could be ignored, but whether the way of calculating $\mathbf{x}_g$ is synchronous or asynchronous much likely has negative impacts on measuring the

performance of algorithms. In other words, when the performance of UNIOA is the same as the performance of ORIGINAL, it is much likely because different ways of calculating $\mathbf{x}_g$ force their performance to behave same, but not because the unified framework itself does not affect the algorithm performance.

Therefore, as shown in Table 6.4, we modify asyncG to syncG, but not change their ways of evaluation. Here, considering the way of calculating $\mathbf{x}_g$ in UNIOA is synchronous, we choose to change asyncG to syncG, but not change syncG to asyncG. This is because we want to avoid misleading our conclusions about whether UNIOA performs same as ORIGINAL. For example, when the performance of UNIOA is different from the performance of ORIGINAL, we hope the reason is the fault of our unified framework, but not the way of calculating $\mathbf{x}_g$.

Table 6.4: hfngnhmgh.

Algorithm	Evaluation		$\mathbf{x}_g$	
	in original code	in this work	in original code	in this work
orig_BA	asyncE	asyncE	asyncG	syncG
orig_GOA	syncE	syncE	syncG	syncG
orig_CSA	syncE	syncE	no use	no use
orig_MFO	asyncE	asyncE	no use	no use
orig_MBO	syncE	syncE	syncG	syncG
orig_BOA	asyncE	asyncE	asyncG	syncG
orig_PSO	asyncE	asyncE	asyncG	syncG

6.3 Experiments for verifying the unified framework

6.3.1 Results

In Section 5.2, we eliminate side effects from $\mathbf{x}_g$ calculation method when reproducing these algorithms in ORIGINAL framework, subsequently, it is much safer to discuss the difference between ORIGINAL and UNIOA.

From Figure 6.3 in which y-axis is each algorithm whose framework is ORIGINAL or UNIOA and its corresponding AUC values obtained in IOHanalyzer are shown on x-axis, it is observed that the two distributions of 24 AUC values for ORIGINAL algorithm and for UNIOA algorithm are the same. Even the outliers from ORIGINAL algorithm and from UNIOA algorithm are also significantly near to each other. Same observations can be found when dimension n is 5 and when dimension n is 20.

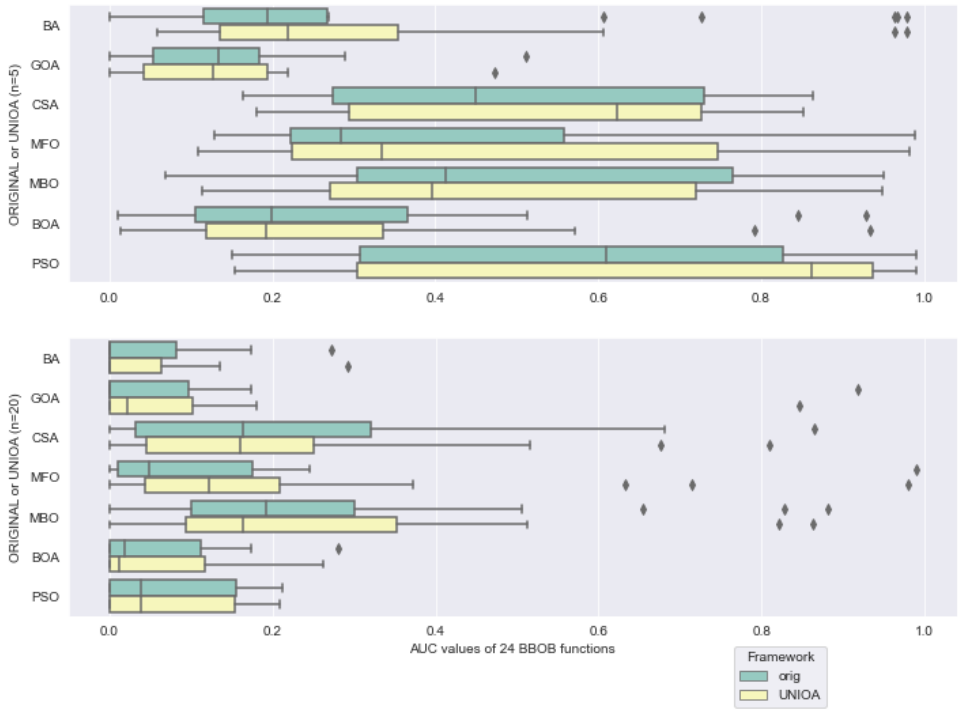

Figure 6.3: Distribution of 24 AUC values for paired algorithm performances in orig framework or UNIOA framework, when $n = 5$ or $n = 20$.

Meanwhile, from Table 6.5 in which each case represents one comparison between each algorithm whose framework is ORIGINAL or UNIOA, it is observed that when dimension n is 5, all p values are larger than 5% which means when $n = 5$, the data obtained from ORIGINAL algorithm and UNIOA algorithm does not reject the hull hypothesis, in other words, **when $n = 5$, the difference between ORIGINAL and UNIOA is not significant**.

Furthermore, in Table 6.5, it is also observed that when dimension n is 20, the difference is significant between orig_PSO and UNIOA_PSO as well as between orig_MFO and UNIOA_MFO. However, it can be acceptable if the difference direction is that UNIOA is better than ORIGINAL in our study. For example, when $n = 20$, UNIOA_PSO has higher AUC values in 16 of 24 BBOB optimization functions, and UNIOA_MFO has higher AUC values in 20 of 24 BBOB optimization functions.

Table 6.5: (1) P-values of Wilcoxon signed-rank test for difference on average between performances of paired algorithms in ORIGINAL framework or UNIOA framework for a random optimization function, when $n = 5$ or $n = 20$. (2) The number of wins orig had or UNIOA had when comparing their own AUC value throughout 24 BBOB functions.

Dimension n	Case	p-value	orig win	UNIOA win
$n = 5$	BA	0.11276741428797125	7	17
	GOA	0.0804264513256041	15	9
	CSA	0.05933346675499405	9	15
	MFO	0.17931823604537578	11	13
	MBO	0.37577150825113037	14	10
	BOA	0.8638867905449266	12	12
$n = 20$	PSO	0.006090717662464104	8	16
	BA	0.6558614582873129	3	21
	GOA	0.06569860480594888	3	21
	CSA	0.7860401269462913	12	12
	MFO	0.000191118942510799	4	20
	MBO	0.14097231038635524	15	9
	BOA	0.8327296566664774	6	18
	PSO	0.6064080161085506	6	18

6.3.2 Conclusion

According to the experimental results in **??**, we can conclude that at least in these seven algorithms, the ORIGINAL framework can be safely replaced by our unified UNIOA framework, in the point of actual algorithm performance.

For example, in Figure 6.4 showing fix-target curves, the purple line denotes the MFO in ORIGINAL, and the orange line denotes the MFO in UNIOA. We can find two lines are close to each other in most of 24 functions, which means the MFO algorithm performs same in two different frameworks when solving most of 24 optimization problems. Sometimes, the MFO in ORIGINAL needs more evaluations to reach the same target value as the MFO in UNIOA, when solving F1. Sometimes, the MFO in UNIOA can obtain much better optimization results than the MFO in ORIGINAL, when solving F2, F7, F17, F21. However, the MFO in ORIGINAL performs slightly better than the MFO in UNIOA when solving F8, F9, F15.

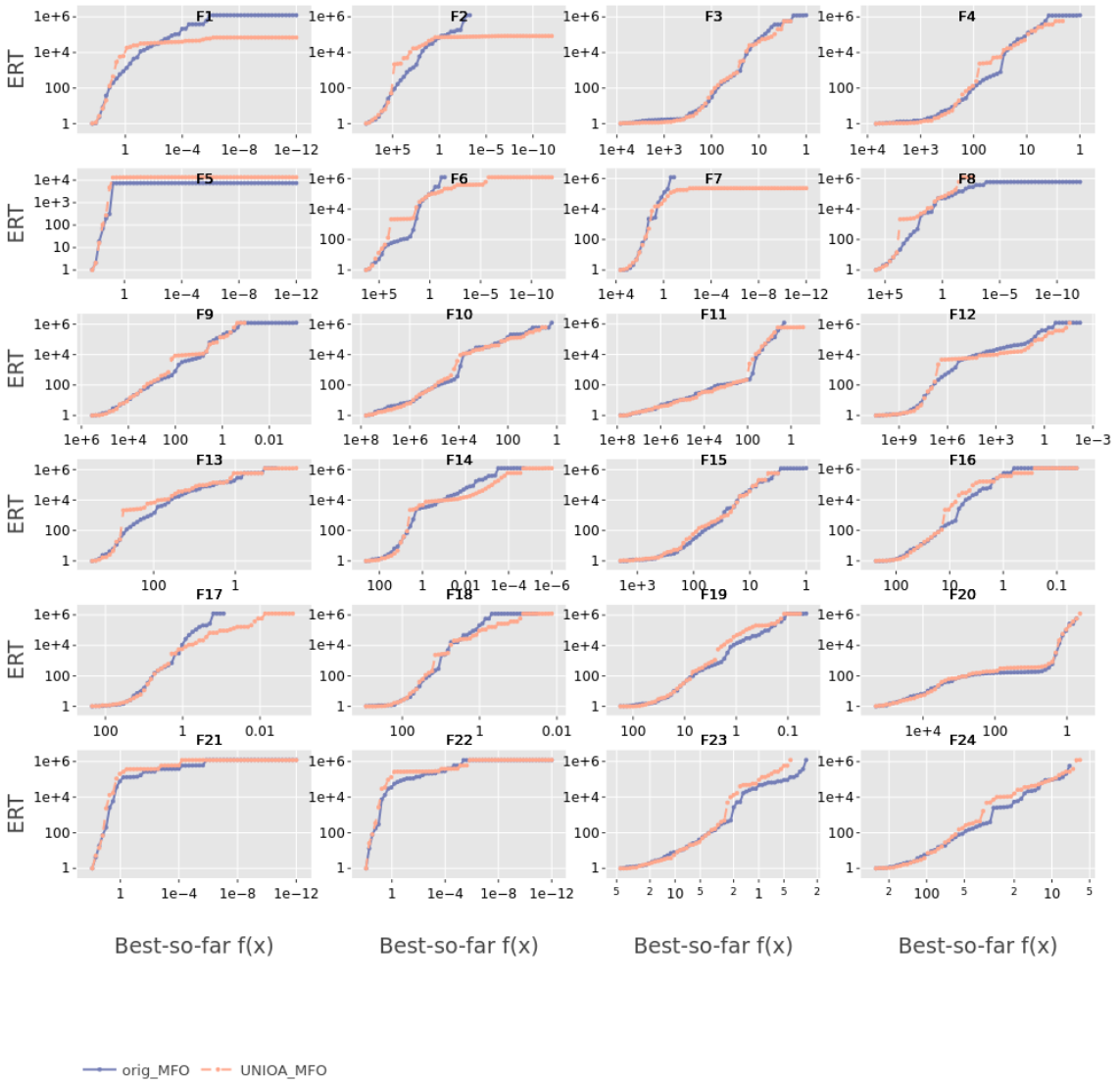

Figure 6.4: ERT values of the MFO in UNIOA and in ORIGINAL, when $n = 5$.

6.4 UNIOA Package

We can safely admit that these seven algorithms can be rewritten in our unified framework with the general dictionary. At the same time, we are interested in an actual application of this framework. Therefore, we attempt to develop a small python package named UNIOA

[3] in which users can design their iterative optimization algorithms only with primitive math knowledge and without any biological knowledge.

Moreover, in UNIOA, the customized algorithm can measure its performance or compare its performance with the other seven existing algorithms with the help of the IOHprofiler environment.

6.5 Experiments for applying the unified framework

In **??**, we study the actual application of our designed framework. The UNIOA package allows scientists without biological knowledge to design their nature-inspired algorithm with primitive math knowledge. Besides customized designing, measuring and evaluating its performance are also important characteristics of the UNIOA package.

In this section, we utilize the UNIOA package to study the performance of these seven algorithms. In Figure 6.5, when the dimension increases (from $n = 5$ to $n = 20$), we can find the probability that the algorithm will reach the target value under limited evaluations is reducing, no matter which algorithm it is.

When the dimension is 5, the PSO in UNIOA is much more likely to generate more target values in solving most of 24 BBOB problems. Meanwhile, we can also observe that the CSA, MBO and MFO in UNIOA have similar performances in solving 24 problems. The same observation can also be found between the BA and BOA in UNIOA. Lastly, it is obvious that the GOA in UNIOA performs worst compared to the other six algorithms when solving 24 problems.

Figure 6.6 also displays the performance of these seven algorithms when the dimension is 5. When solving F3 and F4, the MBO in UNIOA can generate much more optimal solutions before reaching the maximum evaluations. However, other six algorithms obtain much worse solutions, although the maximum evaluations have been finished. Furthermore, same observations can be found in solving F16, F17, F18, F19, but the CSA in UNIOA is the winner this time.

[3] https://github.com/Huilin-Li/UNIOA

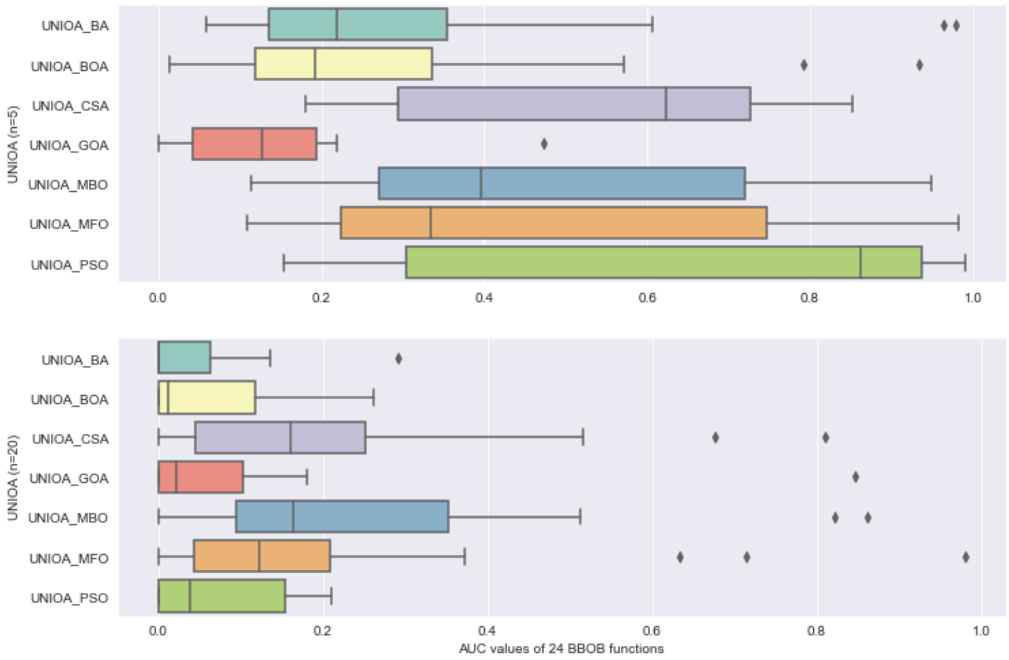

Figure 6.5: Distribution of 24 AUC values for UNIOA algorithms' performance, when $n = 5$ or $n = 20$.

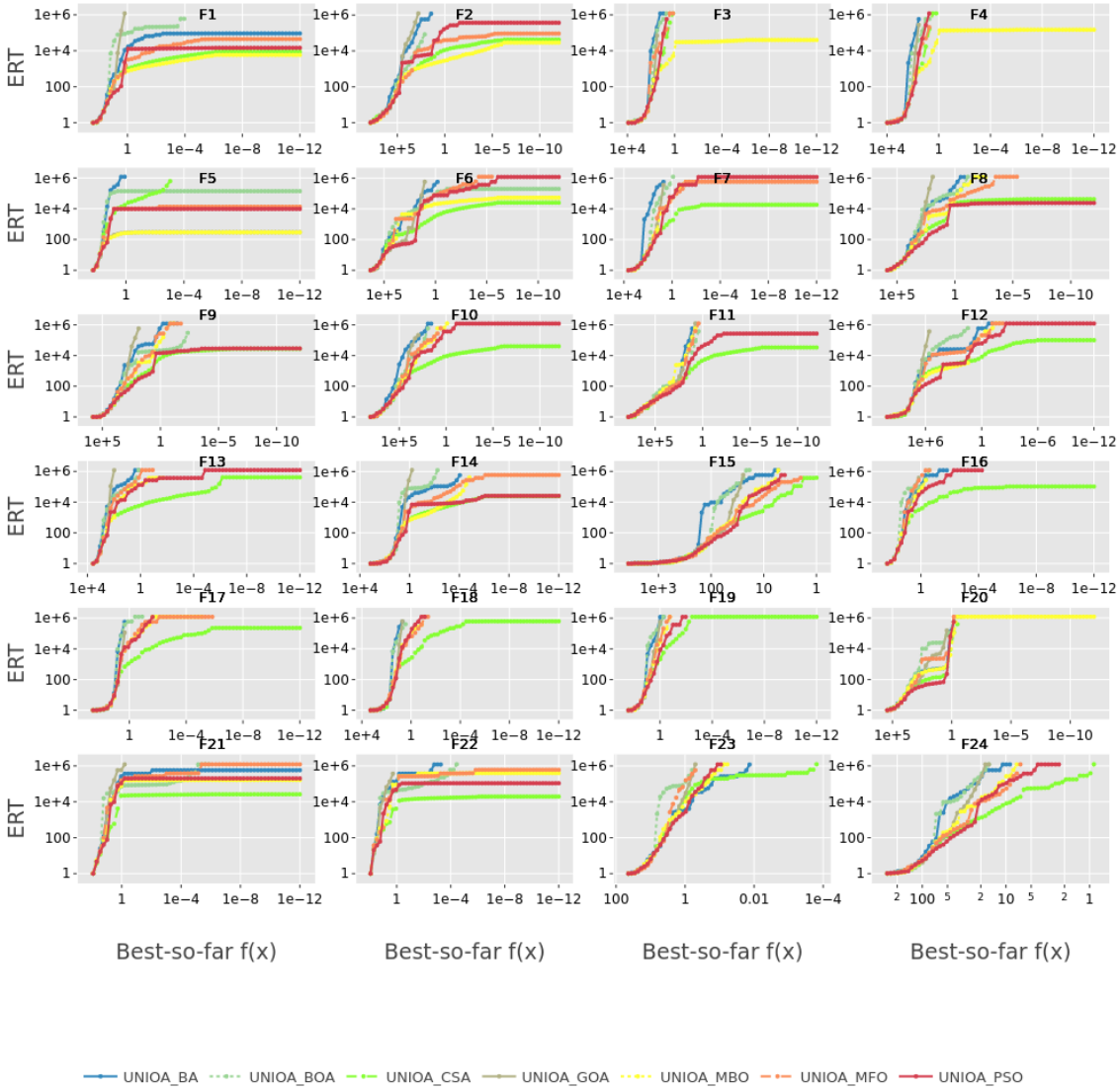

Figure 6.6: ERT values of seven algorithms in UNIOA, when $n = 5$.

6.6 Summary

The math formulas totally determine the ability of different algorithm.

Chapter 7

Summary

7.1 Conclusions in our studies

The math formulas totally determine the ability of different algorithm.

The influence of generic notations/operators could be noticed when observing the development difference between EA group and SI group. The very first clear difference is that the genetic-terminology (mutation, crossover, selection) is the only one analogy in EA but SI has various analogies (bee, ant, whale and so on). Due to generic terminologies, EA group develops well in both theoretical comparison study and auto-algorithm design **Huilin: +ref**. However, SI group seems to have stopped at simulating one natural observation into one optimization process. Recent studies on SI group began to focus on analysing SI group as a whole, such as a comparison study amongst a set of SI algorithms**Huilin: +ref2**, a platform environment in which automatically calls needed algorithms**Huilin: +ref**, a unified framework that subsumes a set of population-based algorithms [20]. Inspired by these previous studies, we hope the comparison can happen in a larger set of SI, we hope auto-design can also happen in optimization operators (just like combinations of operators in EA group), we also hope primitive math knowledge could be enough to construct such a unified framework.

In the 'Generic' environment, amongst nature-inspired algorithms, we have analysed their similarities on symbols and operators in **??**; we also have analysed their similarities on structures and their differences on operators in **??**; moreover, experiments in **??** have proved that 'Generic' can safely replace 'Specific' to a significant extent.

Meanwhile, the Section 6.2 has a taste for population size influence on nature-inspired algorithms in the 'Generic' environment. In this Chapter 7, we will continue theoretically discussing more insights in the view of algorithms' comparison.

If we accept that all nature-inspired algorithms can be divided into these mentioned components,

- does it mean people also can compare algorithms only on their update strategies? For example, comparisons only happen on 'Opt_X'.
- does it mean people in the future could develop new methods by developing these specific components? For example, if there is a more smart method to initialize population, if there is a more efficient method to update assisting variables?

7.2 Limitations in our analysis

1. more comparisons and discussions should be also given, on the math level, there could be more interesting findings.

2. no comparison with other framework ,such as Liu Bo

3. it is meaningful to discuss whether this framework is also ok in EA algorithms.

4. when measuring performance of algorithms, it would be better to also statistically test algorithm-1 and algorithm-2 over 30 runs on solving one problem. Because, if I want to test whether the performance is same on solving one problem.

5. more runs, instances can give more conviced solutions.

6.about the UNIOA demo, the ideal package should be inputs are only components, and output is a complete algorithm.

7. what is the global best one? It is the best one in the current population from current optimization round, or it is the best one in the combination of previous population and current population.

7.3 Future work

For the future research, we focus on discussing the possibility of using this 'Generic' to unify the entire nature-inspired algorithms including EA and SI.

Huilin: parallel space