NUIST Experiment Report

Course name: Embedded Systems Scripting

Experiment name:**The development of Beluga Whale Optimization in Python**

Date: 2024-12-28 Tutor:**Qu Chengzhi**

College:Waterford Class: **IoT Class1**

Name:**Jin Kaifeng** Student ID:202283890029

Name:**Huang Jiarui** Student ID:202283890036

1. Experimental division of labor

Jin is mainly responsible for testing the other three algorithms, as well as reproducing and researching the DO (Dandelion Optimizer) algorithm, and reproducing and conducting preliminary analysis of the MGO (Mountain Gazelle Optimizer) algorithm. In addition, Jin is also responsible for writing reports.

Huang is mainly responsible for the detailed research of DO(Dandelion Optimizer) algorithm , CPO(Crown Pig Optimizer) algorithm , and MGO(Mountain Gazelle Optimizer) algorithm . And he mainly complete the PPT.

In summary, they studied the characteristics of the algorithm, tested and analyzed it, and then extracted the advantages of the other three algorithms separately, and integrated them into the original BWO algorithm to improve the original algorithm.

2. Experimental content

2.1 Analysis of improvement points

(1)Best Fitness

It refers to the fitness value of the optimal solution reached by individuals in the group in the current iteration, and is an important indicator for measuring the quality

of the solution.

When evaluating model performance, 'fitness' can be considered as the 'bias value'. Fundamentally speaking, various algorithm models aim to accurately obtain the true values and predicted values, and the smaller the deviation between the two, the better. In this process, we usually use cost functions such as variance functions to measure the difference between the true value and the target value. However, practical problems are very complex and involve numerous complex functions. How to solve extreme values in a targeted manner is one of the core goals of various algorithms nowadays.

(2)Best Position

It refers to the position of the solution corresponding to the optimal fitness, and the algorithm gradually approaches the global optimal solution by continuously updating the individual's position and velocity.

2.2 Initialization Function

Added the function of recording the best fitness data in every 500 iterations (1 round) in the main function section; Added the function of selecting the best fitness data from the 20 best fitness data; Added the function of recording the total time required for running 20 rounds; Added the function of calculating the median of data.

We ran the BWO initial code 5 times and tested the function's performance as follows:

Table1. Initial BWO

Test function	The magnitude of the median	The magnitude of variance	The magnitude of best adaptation data	Average runtime per run (seconds)
F1	e^ -8	e^ -3	e^ -19	14
F2	e^ -3	e^ -2	e^ -9	13
F3	e^ -35	e^ -28	e^ -130	34
F4	e^ -5	e^ -1	e^ -10	27
F5	e^ -4	e^ -3	e^ -14	12

F6	e^ 0	e^ -1	e^ -4	16
F7	e^ -4	e^ -3	e^ -6	14
F8	e^ -3	e^ 0	e^ -30	18
F9	e^ -7	e^ -2	e^ -21	19

Hint:*Running once will present the best adaptation data after 20 iterations and 500 iterations. The test function is generally stable only in terms of runtime, and has a certain degree of randomness in other aspects, so we can only make relative comparisons. (F7, F8 are relatively stable in all four aspects)*

2.3 Analysis of the other three algorithms

We use the method of **controlling variables** to separate and replace each stage. Analyze whether other algorithms have improved the BWO algorithm through experimental and control groups. Then we will analyze the advantages and disadvantages of the algorithm based on the formulas mentioned in the paper and websites such as GitHub.

2.3.1. Dandelion Optimization Algorithm (DO)

（1）The first stage of DO replacing:

Table2. DO First Stage

Test function	The magnitude of best adaptation		Average runtime per run (seconds)	
	DO_1st	BWO	DO_1st	BWO
F1	e^ -30	e^ -19	15	14
F2	e^ -17	e^ -9	16	13
F3	e^ -20	e^ -130	49	34
F4	e^ -14	e^ -10	15	27
F5	e^ -4	e^ -14	18	12
F6	e^ -6	e^ -4	16	16
F7	e^ -5	e^ -6	18	14
F8	e^ -307 / [0.]	e^ -30	74	18

F9	e^ -20	e^ -21	18	19

After running F8 multiple times, the optimal fitness shown in the figure below is [0.], indicating that the data is already very close.

```
The total operation time is: 74.04824876785278
Best Fitness among all runs: [0.]
```

Figure1. Close to Zero

Summary:

The above method replaces the **first stage** algorithm of BWO with the **first stage** algorithm of DO. Regarding the F8 testing function, there has been an improvement in the level of optimal data adaptation, but the corresponding running time has also increased. The improvement effect of other testing functions is not significant, and even inferior to BWO (such as F5).

Analysis:

The stronger global search capability of DO in the first stage is due to its combination of dynamic adjustment, random factors, and adaptive updates based on fitness. The specific advantages are as follows:

1. Introduced a dynamic parameter adjustment mechanism

In the **first stage of DO**, multiple parameters are dynamically adjusted based on the number of iterations, such as `Yt=2*numpy.randm.rand()*(1-t/Mapos_iter)**(t/Mapos_iter)`. This formula introduces a nonlinear control factor that varies with the number of iterations, allowing the algorithm to explore the search space more fully in the early stages. However, the parameter changes in the **first stage of BWO** are relatively simple and lack control over the number of iterations, resulting in insufficient global exploration capability.

2. Adaptive update rules

The update rule of the **first stage of DO** is based on the relative values of fitness (such as `St` and `Mt`), which dynamically adjusts the search intensity according to the quality of the solution. However, the first stage of BWO lacks dynamic adjustment

based on fitness, and individual updates rely more on random selection, which may lead to the search process relying too much on initial solutions or falling into local optima.

（2）The second stage of DO replacing:

Table3. DO Second Stage

Test function	The magnitude of best adaptation		Average runtime per run (seconds)	
	DO_2ed	BWO	DO_2ed	BWO
F1	e^ -2	e^ -19	20	14
F2	e^ -2	e^ -9	21	13
F3	e^ 1	e^ -130	54	34
F4	e^ -2	e^ -10	20	27
F5	e^ 1	e^ -14	23	12
F6	e^ -1	e^ -4	20	16
F7	e^ -5	e^ -6	22	14
F8	e^ -17	e^ -30	80	18
F9	e^ -4	e^ -21	21	19

Summary:

The above method replaces the **second stage** algorithm of BWO with the **second stage** algorithm of DO. In terms of magnitude and running time, it is not as good as BWO, and the improvement effect is very poor.

Analysis:

1. Insufficient randomness design

Compared to the BWO algorithm, in the position update formula of the **second stage of DO**, all solutions tend to search near the mean `pos_mean`, which reduces population diversity and leads to premature convergence to local optima.

2. Lack of dynamic control

The dynamic parameter `alpha` only depends on the number of iterations, and its variation is fixed. The effect of Gaussian random perturbation is limited, and the

updated formula tends to be biased towards the mean region as a whole.

Hint:*In the third stage of the BWO algorithm, the process of whale fall is simulated, which reduces the number of whale groups. The closer the whale group is to the optimal position, the lower the probability of whale fall. This step aims to force the whale group to approach the optimal position. However, there are no similar steps in other algorithms, so we do not intend to perform equivalent replacement operations in the third stage.*

2.3.2. Crested Porcupine Optimizer (CPO)

（1）The first stage of CPO replacing:

Table4. CPO First Stage

Test function	The magnitude of best adaptation		Average runtime per run (seconds)	
	CPO_1st	BWO	CPO_1st	BWO
F1	e^ -24	e^ -19	17	14
F2	e^ -20	e^ -9	18	13
F3	e^ -26	e^ -130	52	34
F4	e^ -13	e^ -10	16	27
F5	e^ -2	e^ -14	19	12
F6	e^ -4	e^ -4	17	16
F7	e^-4	e^ -6	19	14
F8	[0.]	e^ -30	74	18
F9	e^ -20	e^ -21	19	19

After running F8 multiple times, the optimal fitness was [0.], indicating that the data is already very close.

Summary:

The above method replaces the **first stage** algorithm of BWO with the **first stage** algorithm of CPO. Regarding the F8 testing function, there has been a significant improvement in the level of optimal data adaptation, but the corresponding running

time has also increased. The improvement effect of other testing functions is not significant, and even inferior to BWO (such as F3).

Analysis:

1. More exploratory random factors

In the **first stage of CPO**, multiple random factors were introduced when updating positions: `numpy. random. rand (dim)` and `numpy.random.randn (dim)` were used in combination to provide uniform and normal distribution randomness, respectively, enhancing the diversity of understanding. `numpy.abs (2*numpy.random.rand()* fitness[i,:]-y)` combines `fitness[i,:]` with random factors to further enhance the exploratory ability of understanding.

The randomness in the **first stage of BWO** mainly focuses on updating a few positions, and the design of random factors is relatively simple (such as trigonometric functions and random numbers `r1`, `r2`), which cannot fully cover the search space.

2. Dynamic interaction mechanism

In the **first stage of CPO**, the center point `y` is calculated by comparing the position `pos[i,:]` with randomly selected individuals `pos[rand_index1,:]`, and then perturbation is performed based on this. This approach not only utilizes information exchange between individuals, but also introduces new search directions for the current location.

（2）The second stage of CPO replacing:

Table5. CPO Second Stage

Test function	The magnitude of best adaptation		Average runtime per run (seconds)	
	CPO_2ed	BWO	CPO_2ed	BWO
F1	e^ -3	e^ -19	24	14
F2	e^ -2	e^ -9	25	13
F3	e^ -2	e^ -130	57	34
F4	e^ -2	e^ -10	23	27

F5	e^ 1	e^ -14	25	12
F6	e^ -1	e^ -4	24	16
F7	e^ -3	e^ -6	26	14
F8	e^ -30	e^ -30	78	18
F9	e^ -4	e^ -21	26	19

Summary:

The above method replaces the **second stage** algorithm of BWO with the **second stage** algorithm of CPO. The improvements are not significant or even inferior to BWO.

Analysis:

1.The monotonicity of the update mechanism

The parameter `alpha=random.random()*(t/Mapos_iter-1)**2` only changes linearly with the number of iterations, and the dynamic adjustment mechanism is single and cannot be adjusted based on population fitness or individual differences.

2.Lack of diversity enhancement mechanism

The randomly selected individual `RJ` is only generated through a loop and does not substantially affect the update formula.

Hint:*Similar to the replacement steps of the DO algorithm mentioned above, we do not intend to perform equivalent third stage replacement operations in the CPO section.*

2.3.3. Mountain Gazelle Optimizer (MGO)

The global search and local search of MGO algorithm are performed simultaneously, which is not completely equivalent to the idea of BWO algorithm and cannot be directly replaced by brute force. So we can only try to improve the first stage of BWO by using MGO's global and local search methods.

Table6. MGO First Stage

Test function	The magnitude of best adaptation	Average runtime per run (seconds)

	MGO_1st	BWO	MGO_1st	BWO
F1	e^ -4	e^ -19	32	14
F2	e^ -2	e^ -9	33	13
F3	e^ -2	e^ -130	67	34
F4	e^ -3	e^ -10	31	27
F5	e^ -3	e^ -14	34	12
F6	e^ -3	e^ -4	32	16
F7	e^ -4	e^ -6	34	14
F8	e^ -39	e^ -30	89	18
F9	e^ -6	e^ -21	34	19

Summary:

The above method replaces the **first stage** algorithm of BWO with the **first stage** algorithm of MGO. Regarding the F8 testing function, there has been an improvement in the optimal order of data adaptation, but the effect is weak and the running time has also increased significantly. The improvement effect of the remaining test functions is not significant, and even most of them are inferior to BWO.

Analysis:

1.High computational complexity

Introduce multiple complex operations (such as `Coefficient_Vector` and multiple random perturbations)

2.Strong parameter dependency

There are many random factors and weight parameters in the updated formula that need to be continuously adjusted.

2.4 Further optimization attempts

2.4.1 BWO self-improved

Positional changes of intelligent swarm algorithm are based on random numbers. Whales may find good positions but move elsewhere in updates. Using best position

as guide can make all whales approach it. We improve position transformation in BWO's first phase using best position as guide. This maintains global search breadth and enhances local search convergence:

Table7. Self-improved

Test function	The magnitude of best adaptation		Average runtime per run (seconds)	
	Self-improved	BWO	Self-improved	BWO
F1	e^ -20	e^ -19	20	14
F2	e^ -15	e^ -9	20	13
F3	e^ -32	e^ -130	52	34
F4	e^ -12	e^ -10	18	27
F5	e^ -4	e^ -14	21	12
F6	e^ -5	e^ -4	20	16
F7	e^ -5	e^ -6	22	14
F8	e^ -250 / [0.]	e^ -30	76	18
F9	e^ -44	e^ -21	21	19

Summary:

The above methods have optimized the BWO algorithm itself.

Analysis:

The result is consistent with our hypothesis:

The position changes of intelligent swarm algorithms are generated based on random numbers and have strong randomness. In the BWO algorithm, even if some whale individuals reach an optimal position at a certain moment, due to the influence of random factors, they may move to other positions in subsequent position update iterations, which to some extent affects the convergence efficiency and optimization effect of the algorithm.

So we attempted to improve the position transformation in the first stage of BWO (i.e. swimming exploration stage) to use optimal position guidance, which not only

maintains the breadth of global search but also enhances the convergence of local search.

2.4.2 Other

We have represented the original main method as a class and incorporated OOP concepts such as encapsulation, inheritance, and polymorphism. In addition, we have added regular expressions, exception handling, code introspection, and other usage to make the structure and content of the code more complete, improving its readability and robustness.

3. Experimental result

Based on the above content, we have replaced and integrated various methods, resulting in two solutions:

（1）BWO Self-improved, DO first stage and CPO second stage replacing:

Although the individual performance of the second stage of CPO was not satisfactory, we want to try to integrate multiple stages to test whether it can enhance:

Table8. Self, DO and CPO

Test function	The magnitude of best adaptation		Average runtime per run (seconds)	
	Self_DO1_CPO2	BWO	Self_DO1_CPO2	BWO
F1	e^ -2	e^ -19	21	14
F2	e^ -2	e^ -9	22	13
F3	e^-2	e^ -130	60	34
F4	e^-2	e^ -10	21	27
F5	e^1	e^ -14	22	12
F6	e^0	e^ -4	21	16
F7	e^-3	e^ -6	22	14
F8	e^-23	e^ -30	77	18
F9	e^-5	e^ -21	22	19

Summary:

Despite multiple stages of replacement, the results were extremely poor.

Analysis：

Replace the first stage (global search): It has strong global search ability, making the population evenly distributed in the function space. However, without the gradual convergence mechanism of BWO algorithm, the search scope in the subsequent development stage will be too scattered, making it difficult to effectively utilize the global optimal solution.

Replace the second stage (convergence): Lack of strong convergence ability, making it difficult for the population to quickly aggregate to the optimal solution. Random disturbances are still significant, affecting the concentration of the population towards the optimal solution, resulting in poor performance of the entire model.

（2）BWO Self-improved and DO first stage replacing:

Due to the failure of the first attempt, we came up with another method that replaces the first stage of DO with the improved best position guidance of BWO:

Table9. Self and DO First Stage

Test function	The magnitude of best adaptation		Average runtime per run (seconds)	
	Self_DO1	BWO	Self_DO1	BWO
F1	e^ -35	e^ -19	15	14
F2	e^ -14	e^ -9	16	13
F3	e^ -31	e^ -130	48	34
F4	e^ -30	e^ -10	14	27
F5	e^ -5	e^ -14	18	12
F6	e^ -8	e^ -4	15	16
F7	e^ -8	e^ -6	17	14
F8	[0.]	e^ -30	70	18
F9	e^ -30	e^ -21	17	19

Summary:

The effect is very good, and we have optimized nearly 80% of the existing 9 test functions.

Analysis:

The newly constructed algorithm model leverages the excellent global search capability of the DO algorithm and the powerful function convergence ability of the BWO algorithm. Additionally, by introducing an algorithm function guided by the best position during the global search process, our model has significantly improved in the prediction of the F8 function and has also shown performance enhancements in the testing of other functions.

The following is a schematic diagram of the convergence curves for each result. However, it should be noted that due to our limited time, we are unable to further operate, so this diagram only shows the best convergence curve for one of them:

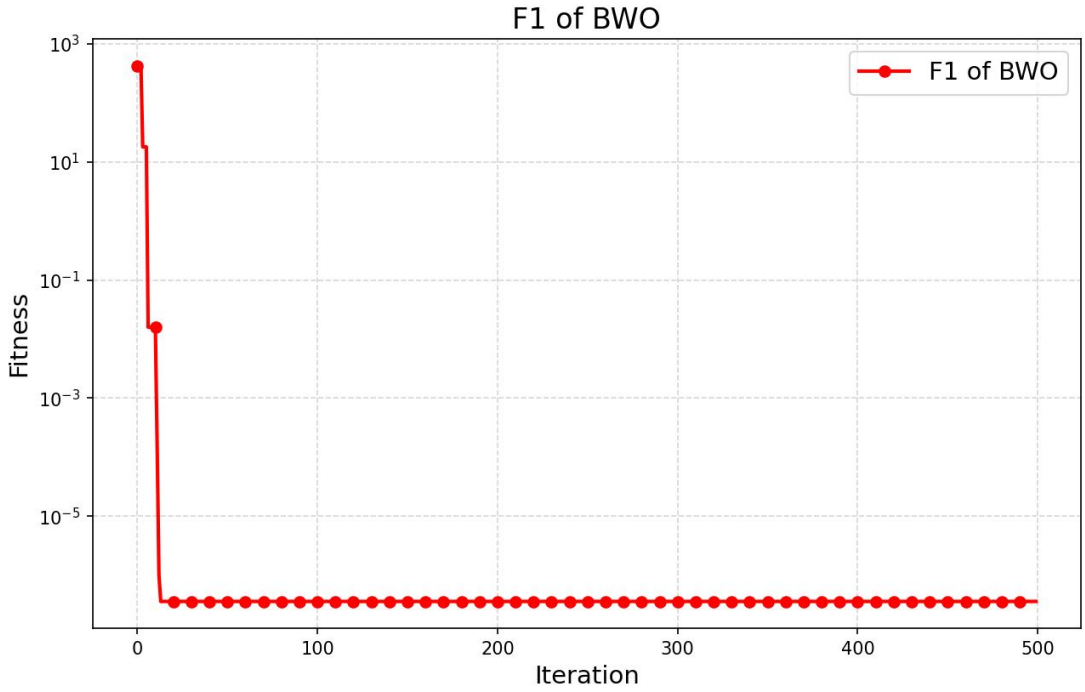

Figure2. F1

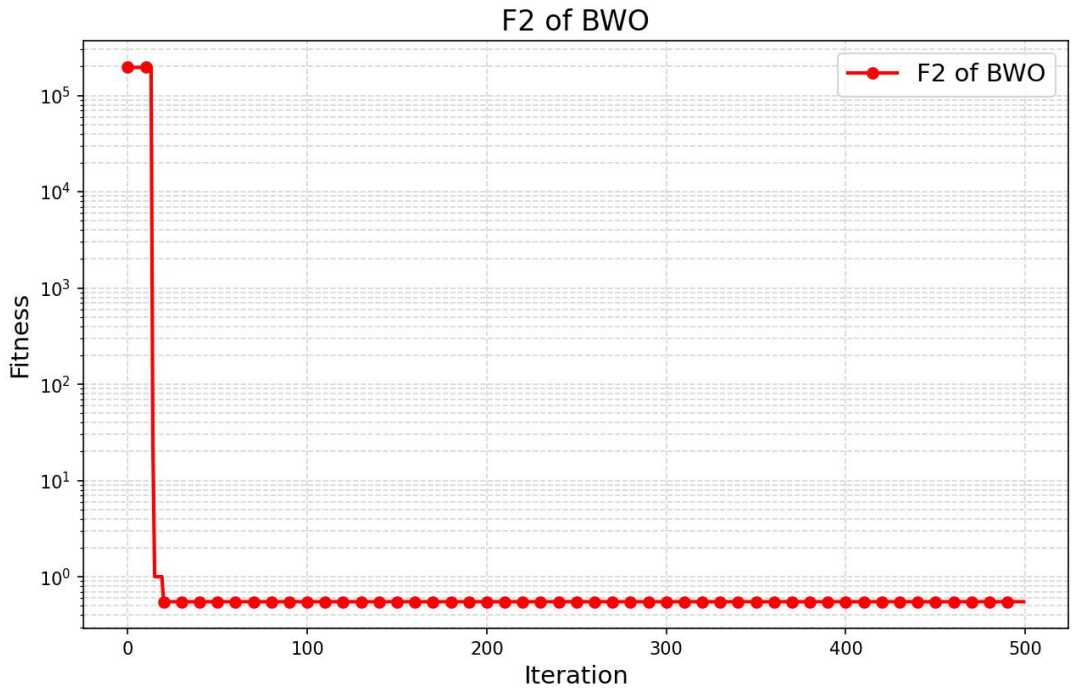

Figure3. F2

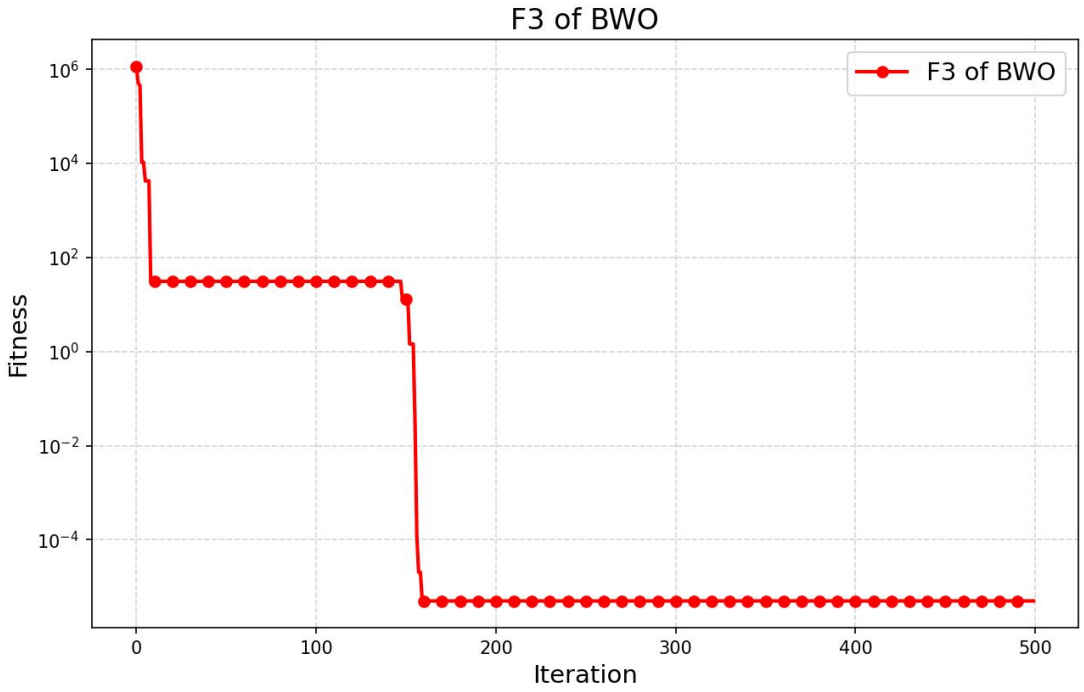

Figure4. F3

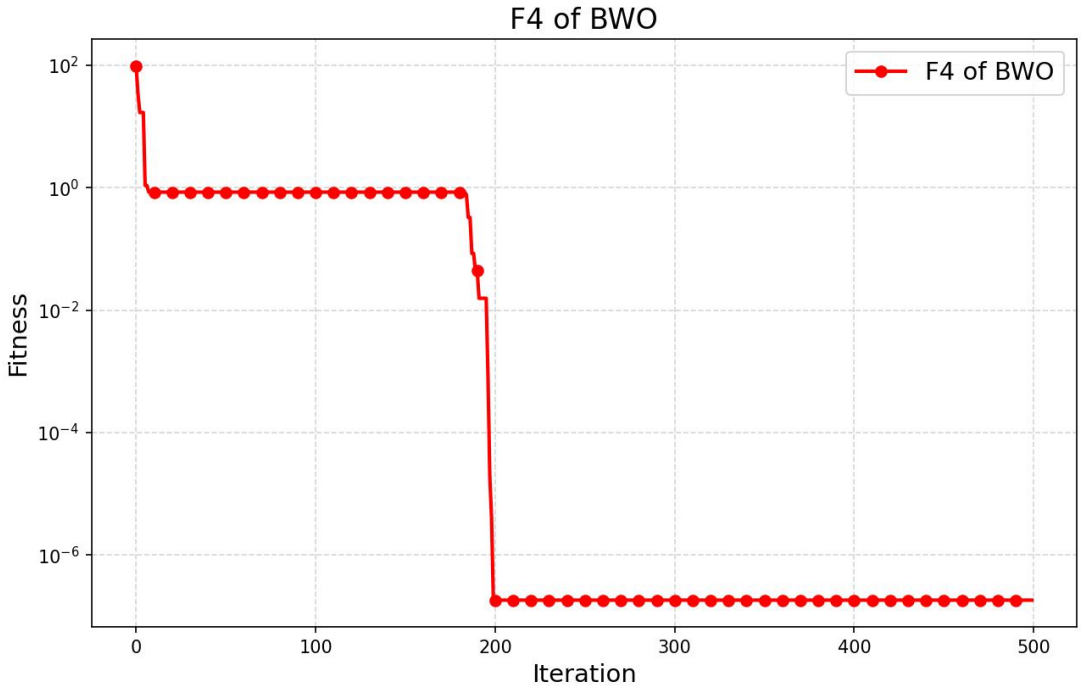

Figure5. F4

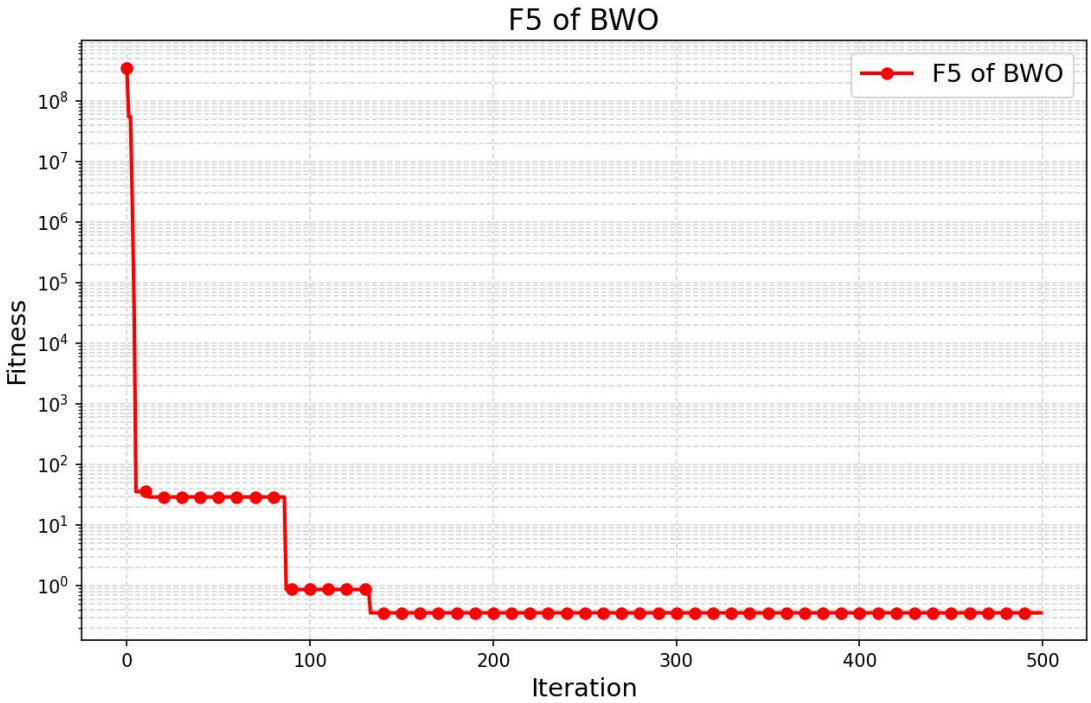

Figure6. F5

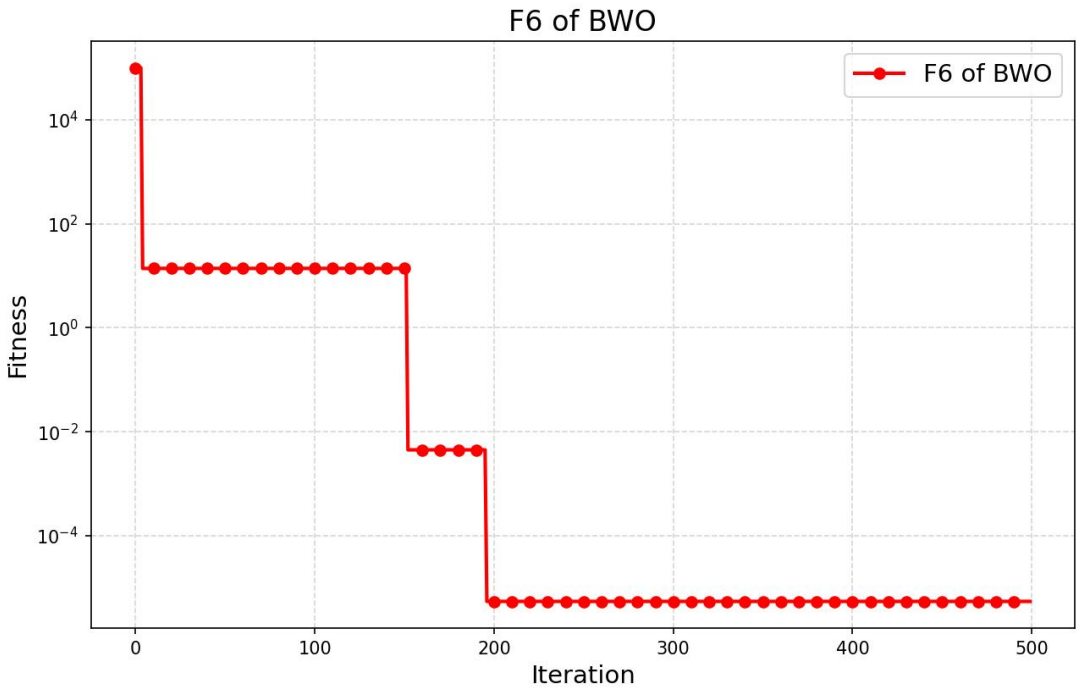

Figure7. F6

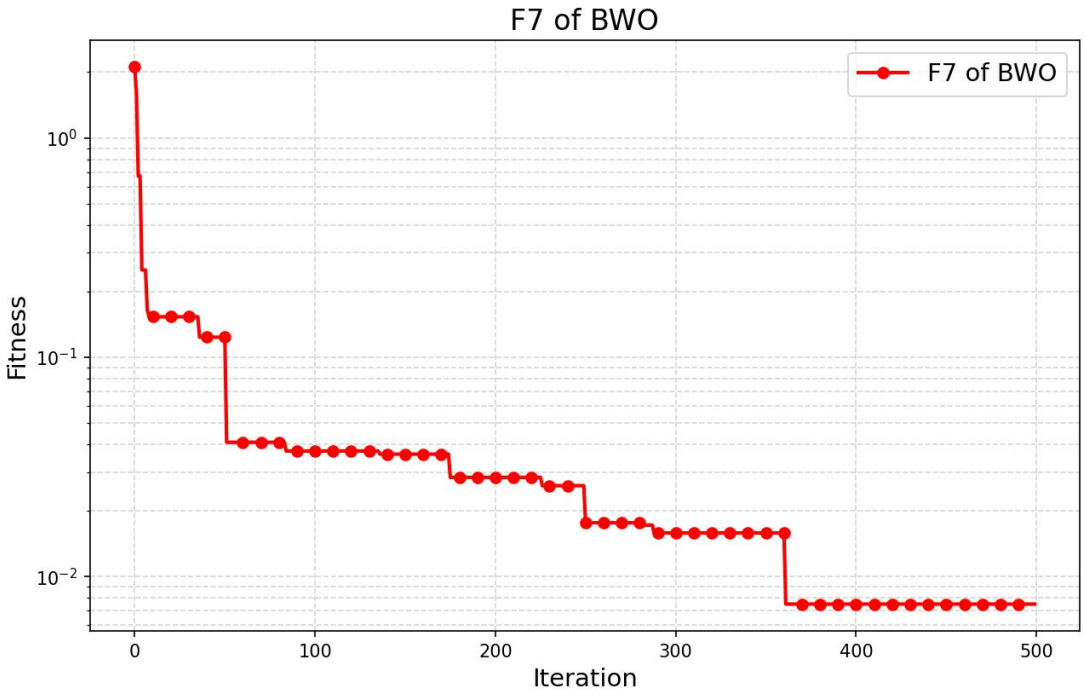

Figure8. F7

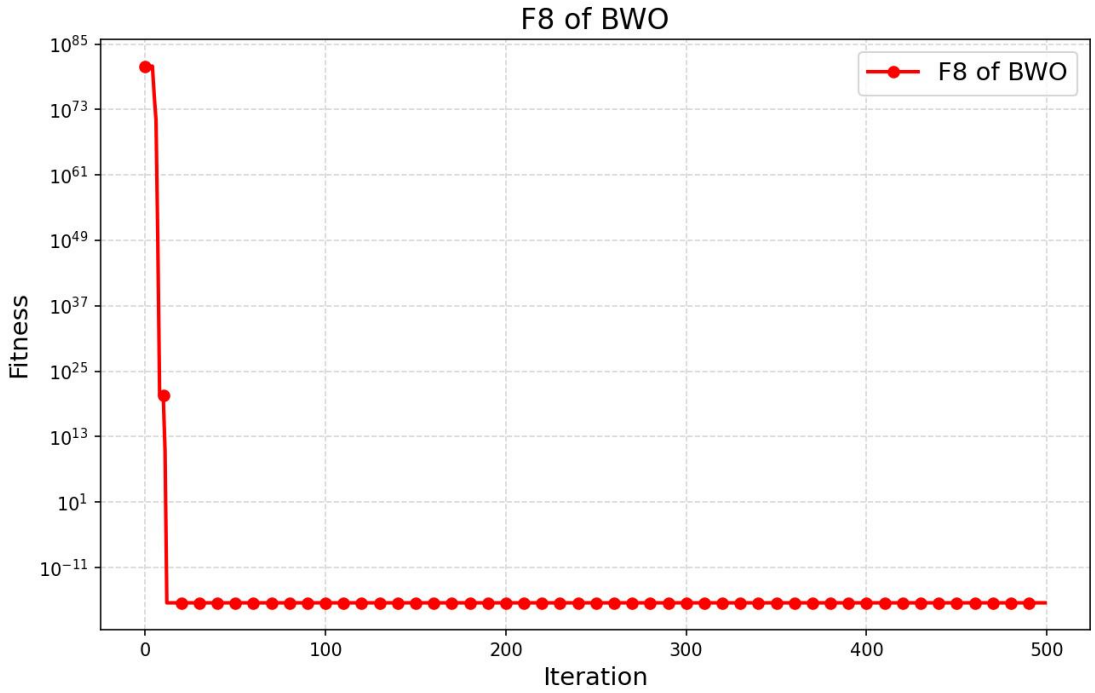

Figure9. F8

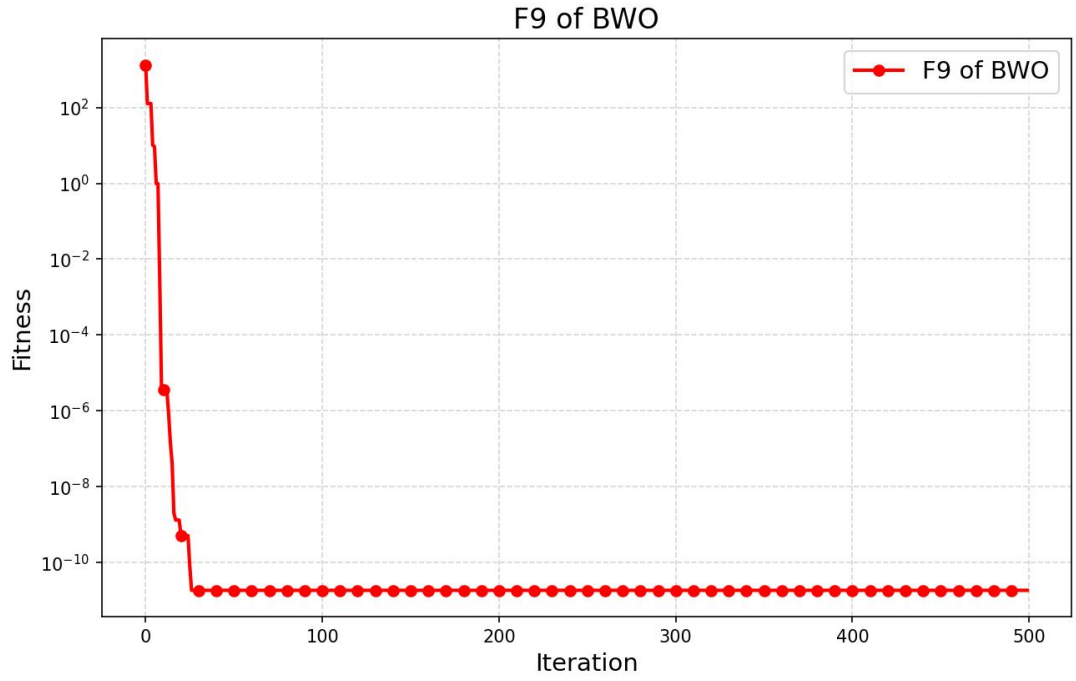

Figure10. F9

In summary, among the existing replacement methods, we have comprehensively evaluated that "BWO Self-improved and DO first stage replacing" is currently the best solution for algorithm optimization.

4. Program code

According to the third section, we adopt the 'BWO Self Improved and DO First Stage Replacing' algorithm as the final replacement optimization algorithm. The specific code content can be found in the zip file attached. Only the main algorithm part is displayed here:

```
import random
import numpy
import math
from solution import solution
import time
import copy

# population sorting
# return the fitness and order after sorting
def SortFitness(Fit):
    fitness = numpy.sort(Fit)
    indepos = numpy.argsort(Fit)
    return fitness, indepos

# adjust the population position based on the sorting results
# return the adjusted population matrix
def SortPosition(pos, indepos):
    posnew = numpy.zeros(pos.shape)
    for i in range(pos.shape[0]):
        posnew[i, :] = pos[indepos[i], :]
    return posnew

def Bounds(s, Lb, Ub):
    temp = s
    for i in range(len(s)):
        if temp[i] < Lb[i]:
            temp[i] = Lb[i]
        elif temp[i] > Ub[i]:
            temp[i] = Ub[i]

    return temp

def Levy(dim):
    beta = 1.5
    sigma = (
```

```
        math.gamma(1 + beta)
        * math.sin(math.pi * beta / 2)
        / (math.gamma((1 + beta) / 2) * beta * 2 ** ((beta - 1) / 2))
    ) ** (1 / beta)
    u = 0.01 * numpy.random.randn(dim) * sigma
    v = numpy.random.randn(dim)
    zz = numpy.power(numpy.absolute(v), (1 / beta))
    step = numpy.divide(u, zz)
    return step

def BWO(objf, lb, ub, dim, SearchAgents_no, Mapos_iter):
    # best White Whale
    xposbest = numpy.zeros([1, dim])
    fvalbest = float("inf")

    # determine whether it is a vector
    if not isinstance(lb, list):
        # vectorization
        lb = [lb for _ in range(dim)]
        ub = [ub for _ in range(dim)]
    lb = numpy.asarray(lb)
    ub = numpy.asarray(ub)

    # initialize the population
    pos = numpy.asarray(
        [pos * (ub - lb) + lb for pos in numpy.random.uniform(0, 1, (SearchAgents_no,
dim))]
    )
    Newpos = numpy.zeros(pos.shape)
    Newfitness = numpy.zeros([SearchAgents_no, 1])
    fitness = numpy.zeros([SearchAgents_no, 1])

    for i in range(SearchAgents_no):
        fitness[i] = objf(pos[i, :])
    fitness, sortIndepos = SortFitness(fitness)  # sort fitness values
    pos = SortPosition(pos, sortIndepos)  # sort population
    fvalbest = copy.copy(fitness[0])  # record the optimal fitness value
    xposbest[0, :] = copy.copy(pos[0, :])  # record the optimal solution
    # initialize convergence curve
    convergence_curve = numpy.zeros(Mapos_iter)

    # save the result
    s = solution()
```

```
    timerStart = time.time()
    s.startTime = time.strftime("%Y-%m-%d-%H-%M-%S")

    t = 0  # loop counter

    best_buffer = 0

    output =" "
    # iteration
    while t < Mapos_iter:
        Newpos = pos
        # whale landing probability
        WF = 0.1-0.05*((t+1)/Mapos_iter)
        randlist = [random.random() for _ in range(0, SearchAgents_no)]
        kk = numpy.array(randlist)
        kk *= (1 - 0.5 * (t + 1) / Mapos_iter)

        for i in range(0, SearchAgents_no):
            # balance factor
            if kk[i] > 0.5:
                # exploration stage -- swimming
                alpha = random.random() * (t / Mapos_iter - 1) ** 2
                V_x = math.cos(random.uniform(-math.pi, math.pi)) / math.exp(t /
Mapos_iter)
                V_y = math.sin(random.uniform(-math.pi, math.pi)) / math.exp(t /
Mapos_iter)
                ln_y = numpy.random.uniform(0, 1)

                q = (t-1)**2/(Mapos_iter-1)**2
                k = 1 - random.uniform(0,1) * q

                RJ = math.floor(SearchAgents_no * random.random())
                while RJ == i | RJ == SearchAgents_no:
                    RJ = math.floor(SearchAgents_no * random.random())
                    if random.uniform(0,1) < 1.5:
                        if dim <= SearchAgents_no / 5:
                            indices = numpy.arange(dim)
                            numpy.random.shuffle(indices)
                            params = [indices[0], indices[1]]
                            global_best = pos[numpy.argmin(fitness)]
                            Newpos[i, params[0]] = pos[i, params[0]] + alpha * V_x *
V_y * ln_y * (
                                        global_best[params[0]] - pos[i, params[1]])
                            Newpos[i, params[1]] = pos[i, params[1]] + alpha * V_x *
```

```
V_y * ln_y * (
                                    global_best[params[0]] - pos[i, params[1]])
                    else:
                        params = numpy.arange(dim)
                        numpy.random.shuffle(params)
                        for j in range(round(dim/2)):
                            Newpos[i, 2*j-1] = pos[i, params[2*j-1]] + alpha * V_x
* V_y * ln_y * (global_best[params[2*j-1]]-pos[2*j-1])
                            Newpos[i, 2*j] = pos[i, params[2*j]] + alpha * V_x *
V_y * ln_y * (global_best[params[2*j]]-pos[2*j])
                else:
                    if dim <= SearchAgents_no / 5:
                        indices = numpy.arange(dim)
                        numpy.random.shuffle(indices)
                        params = [indices[0], indices[1]]
                        global_best = pos[numpy.argmin(fitness)]
                        Newpos[i, params[0]] = pos[i, params[0]] * k
                        Newpos[i, params[1]] = pos[i, params[1]] * k
                    else:
                        params = numpy.arange(dim)
                        numpy.random.shuffle(params)
                        for j in range(round(dim/2)):
                            Newpos[i, 2*j-1] = pos[i, params[2*j-1]] * k
                            Newpos[i, 2*j] = pos[i, params[2*j]] * k

            else:
                # development Stage -- predator
                r3 = random.random()
                r4 = random.random()
                C1 = 2*r4*(1-(t + 1) / Mapos_iter)
                RJ = math.floor(SearchAgents_no * random.random())
                while RJ == i | RJ == SearchAgents_no:
                    RJ = math.floor(SearchAgents_no * random.random())
                LevyFlight = Levy(dim)
                Newpos[i,:] = r3 * xposbest - r4 * pos[i,:] + C1 * LevyFlight* (pos[RJ,:]
- pos[i,:])

            # end exploration and development
            # boundary constraints and updates on individuals
            Newpos[i, :] = Bounds(Newpos[i, :], lb, ub)
            Newfitness[i] = objf(Newpos[i, :])
            if Newfitness[i] < fitness[i]:
                pos[i, :] = Newpos[i, :]
                fitness[i] = Newfitness[i]
```

```
        for i in range(SearchAgents_no):
            if kk[i] <= WF:
                RJ = math.floor(SearchAgents_no * random.random())
                while RJ == i | RJ == SearchAgents_no:
                    RJ = math.floor(SearchAgents_no * random.random())
                r5 = random.random()
                r6 = random.random()
                r7 = random.random()
                C2 = 2*SearchAgents_no*WF
                stepsize2 = r7*(ub-lb)*math.exp(-C2*(t+1)/Mapos_iter)
                Newpos[i, :] = (r5*(pos[i,:]-r6*pos[RJ,:])+stepsize2)
                Newpos[i, :] = Bounds(Newpos[i, :], lb, ub)
                if Newfitness[i] < fitness[i]:
                    pos[i, :] = Newpos[i, :]
                    fitness[i] = Newfitness[i]


        # update the optimal solution
        fitness, sortIndepos = SortFitness(fitness)
        pos = SortPosition(pos, sortIndepos)
        fvalbest1 = copy.copy(fitness[0])
        xposbest1 = copy.copy(pos[0, :])
        if fvalbest1 < fvalbest:
            fvalbest = fvalbest1
            xposbest = xposbest1.copy()
            best_buffer = t

        convergence_curve[t] = fvalbest
        if t % 1 == 0:
            # string formatting
            print([
                    "At iteration {}, the best fitness is {}.".format(str(t),
str(fvalbest))  # String Formatting
            ])
            if(t==499):
                print("The best fitness of this time is: "+str(fvalbest))
                print(f"Find the optimal solution in the iteration {best_buffer}")
                output = str(fvalbest)
        t = t + 1

    timerEnd = time.time()
    s.endTime = time.time()
    s.eposecutionTime = timerEnd - timerStart
```

```
    s.convergence = convergence_curve
    s.optimizer = "BWO"
    s.objfname = objf.__name__
    s.best = fvalbest
    s.bestIndividual = xposbest
    s.bestTime = best_buffer

    return s
```

5. Conclusions

The difficulty of **Jin Kaifeng** lies in the lack of suitable methods for improving the Levy flight phase, which poses a challenge. When replacing other algorithms for the three stages of BWO, the improvement is not very significant, only having a significant effect on specific test functions.
Huang Jiarui 's difficulty lies in the belief that different algorithms handle the initial matrix differently. The biggest difference between the MGO algorithm and our original BWO algorithm is that the global and local searches of the MGO algorithm are performed simultaneously, which poses some problems for the replacement of the algorithm part. After the position transformation in the first stage, the format of the matrix is not compatible with the algorithm in the next stage. Therefore, after replacement in different stages, attention should be paid to the processing of the initialization matrix by different algorithms.

In addition, both Jin and Huang found that when functions are globally explored and converged, there is strong randomness, especially in some test functions, which often occur after multiple runs without a relatively stable optimal convergence result of the order of magnitude. We can only try our best to conduct testing and replacement.
Due to time and energy constraints, we have not fully explored all possible replacements. If possible in the future, we will further explore and optimize the algorithm.