Reimplement and Extension of CCGA

Runying Jiang
rj1u20@soton.ac.uk

Abstract: In this paper, I developed a CCGA model and tested it on four functions based on the paper published by Potter, Mitchell A and De Jong, Kenneth A [1] and tried to dig deeper into the CCGA model. In the extension part, I used the static variable grouping strategy to optimize the CCGA model and found CCGA model with only one dimension is still the best strategy in most function optimization problems, though CCGA model with static variable grouping seems behaves well on some functions like Xin-She Yang N. 2 Function.

1. Introduction of Re-implement

1.1. Brief Description

A biological model that has represented the coevolution of cooperative species is the CCGA model (Potter, Mitchell A and De Jong, Kenneth A) [1] introduced as an effective way to optimize the GA and other EA-based algorithms. This model was tested and applied to the optimization of four specific functions: Rastrigin Function, Schwefel Function, Griewangk Function, Ackley Function, Rosenbrock Function. Compared with the standard GA, CCGA-1 model performs better in the former four functions and CCGA-2 model (a variant of CCGA-1) performs as well as the standard GA. Besides, the CCGA model was illustrated to have a smaller amount of computation. It is also suggested that cooperative coevolution can be applied to a more complex structure such as a neural network and set of rules.

1.2. Model Description

The CCGA model is inspired by the theory of the coevolution algorithm. A coevolutionary algorithm is an evolutionary algorithm (or collection of evolutionary algorithms) in which the fitness of an individual depends on the relationship between that individual and other individuals [2]. If we are to extend the CCGA model to provide reasonable opportunities for the emergence of coadapted sub-components, we must address the issues of problem decomposition, inter-dependencies between sub-components, credit assignment, and the maintenance of diversity [3].

- **Problem decomposition:** The domain of function optimization is a natural decomposition of the problem. It is reasonable to decompose the function with N parameters into N sub0tasks, with each assigned to the optimization of a single variable. The individuals are represented by a binary string with 16 bits. The goal of the task is to find the global minimum.
- **Inter-dependencies between sub-components:** The complete solution is by combining all of the sub-species (N parameters in the function).
- **Credit assignment:** The credit allocation of individuals is defined according to the fitness function: $u(x) = f(x\_max) - f(x)$, where $f(x\_max)$ is the maximum value of the function domain in the lastest five epochs (scaling window with width of five).
- **Maintenance of diversity:** The evolution of each species (sub-population) is handled by a standard GA with two-point crossover and bit mutation. Meanwhile, fitness proportion with the elitist strategy is used in selecting the next generation.

1.3. Detail of Experiment

In this experiment, CCGA model is evaluated by using it to perform four function optimization tasks and the performance of CCGA is compared with the standard GA. The global minimums of all selected functions are zero.
The selected functions are as follow:

- *Rastrigin:* $f(\mathrm{x}) = 10n + \sum_{i=1}^{n}(x_i^2 - 10cos(2\pi x_i))$ (-5.12 $\leq x_i \leq$ 5.12 minimum at $f(0, \cdots, 0) = 0$ n=20)

- *Schwefel:* $f(\mathrm{x}) = \sum_{i=1}^{n}(-x_i sin(\sqrt{|x_i|})) + \alpha \cdot n(\ \alpha = 418.982887 - 500 \leq x\_i \leq 500$ minimum at $f(420.968746, \cdots, 420.968746) = 0$ n=10)

- *Griewangk:* $f(\mathrm{x}) = 1 + \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} cos(\frac{x_i}{\sqrt{i}})(-512 \leq x_i \leq 512$ minimum at $f(0, \cdots, 0) = 0$ n=10)

- *Arckley:* $f(\mathrm{x}) = -20exp(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}) - exp(\frac{1}{n}\sum_{i=1}^{n} cos(2\pi x_i)) + 20 + e$ $(\ -30 \leq x_i \leq 30$ minimum at $f(0, \cdots, 0) = 0$ n=30)

Following is the implementation details, **Table: I** shows the main characteristics of CCGA Algorithm:

Algorithm 1 CCGA

0: gen = 0
for each species s **do**
$P_s(gen) =$ randomly initialized population
end for
while termination criterion $= false$ **do**
gen = gen + 1
for each species s **do**
select $P'_s(gen)$ from $P_s(gen-1)$ based on fitness proportion
apply genetic operators to $P'_s(gen)$
evaluate each individual in $P'_s(gen)$
end for
end while=0

representation	binary string (16 bits)
population size	100
selection	fitness proportionate
fitness function	scaling window technique (width=5)
elitist strategy	single copy of best individual preserved
genetic operators	two-point crossover and bit-flip mutation
crossover rate	0.6
mutation rate	1/16
generation	steady state

Table I: Main characteristics of Algorithm

2. Re-implemented Results

As illustrated in the following figure **Figure : 1, 3, 5, 7** CCGA model performs better than the standard GA model in solving function optimization problem. Since the process of a genetic algorithm is random and the result is not steady, there is no way to get the exactly the same curve as the original paper. It seems my genetic operator is generally better, though I use the same parameters as the original paper. It is worth mentioning that the running time of my CCGA model is much longer than the standard GA model, which is inconsistent with what suggested in the original paper that the standard GA maps on all function variables while the CCGA algorithms only need to apply the mapping to a single function variable, so the computation cost of CCGA algorithm is less. I think one of the reason may be some tricks in computation acceleration with python or I write a relatively complex CCGA model.

Another issue I encountered in the experiment was the representation of the individuals. At first, I tried to do some processing on the accuracy of converting a string into a number like applying a dynamic decimal point by mutation. Guess what happened, I got an amazing result in the performance of genetic algorithm. The function gets the best individual of the global minimum in 1000 epochs.

Two corrections on the original paper: One is the formula of Schewefel Function, it should be substraction instead of add; The other one is the variable range of Griewangk should be $-512 \leq x_i \leq 512$.

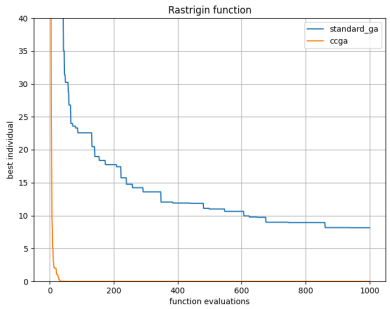

Figure 1: My experiment result on Rastrigin Function

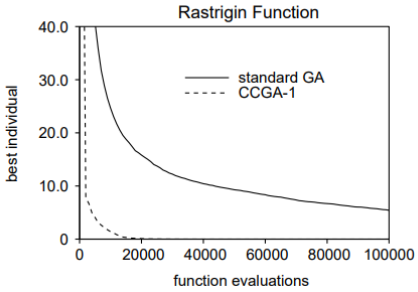

Figure 2: Original result on Rastrigin Function

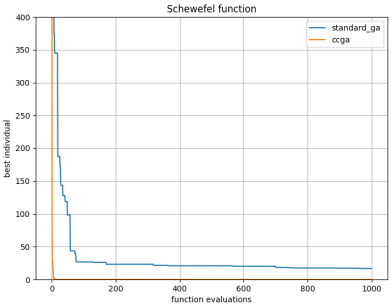

Figure 3: My experiment result on Schewefel Function

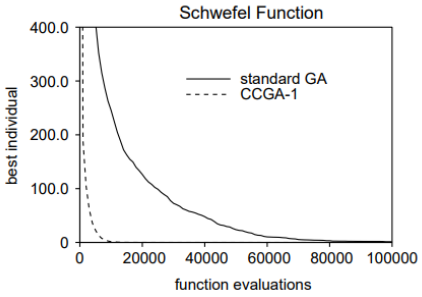

Figure 4: Original result on Schewefel Function

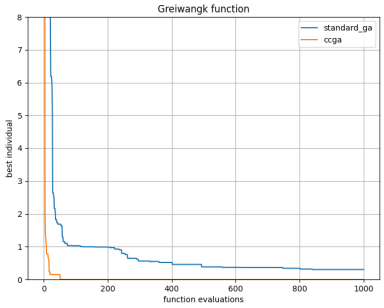

Figure 5: My experiment result on Griewangk Function

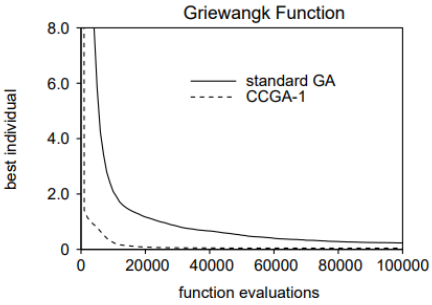

Figure 6: Original result on Griewangk Function

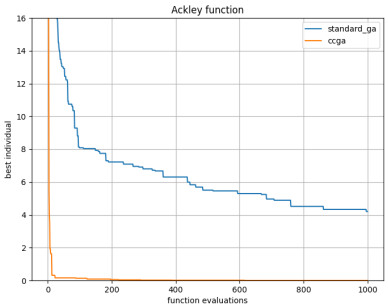

Figure 7: My experiment result on Ackley Function

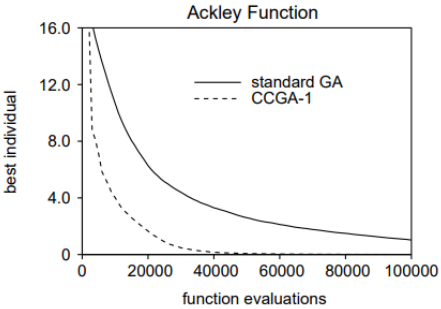

Figure 8: Original result on Ackley Function

3. Extension

3.1. Description of Extension

My research question is if variable grouping will make the CCGA model performs better on the original task and if not, on which issue variable grouping can significantly improve the effectiveness of the model. I got my idea of extension from the static decomposition model special for large-scale optimization problem [4] and random grouping strategy with fixed sub-component size proposed by Yang[5].

CCGA statically decomposes the n-dimensional optimization problem into n one-dimensional sub-problems. In the case of completely separable problems, this decomposition scheme provides a significant advantage over classic GA, because CCGA reduces the search space from n dimensions to n 1 dimensions. However, it cannot solve the indivisible problem of close interaction between decision variables and as we knew most of the problems in real life has strong linkage in parameters. So there is a need to research more in variable grouping of the CCGA model[6]

I applied CCGA with static grouping to the optimization of four original functions: Rastrigin Function, Schwefel Function, Griewangk Function, Ackley Function. For instance, I decomposed 20 parameters in Rastrigin function to 4 groups with 5 parameters so that some of the interacting variables will be kept in the same group. At first I thought that CCGA model with variable grouping would perform better, but as it is shown in **Figure: 11** the result was contrary to my assumptions though the running time of the model was halved.

Then I changed the task domain to another two functions Xin-She Yang N. 2 Function **Figure: 9**, Xin-She Yang N. 3 Function **Figure: 10** and expected the model with variable grouping will perform better since they are all non-separable functions. Also, I tested the algorithm with 20 parameters and 400 parameters separately to see the impact of dimension on the model **Figure: 13 12**. From the figure, we could see in optimizing Xin-She Yang N. 3 Function, variable grouping has a large improvement on the performance of CCGA model.

- Xin-She Yang N. 2 Function:

$$f(\mathbf{x}) = f(x_1, \ldots, x_n) = (\sum_{i=1}^{n} |x_i|) exp(-\sum_{i=1}^{n} sin(x_i^2))$$

- Xin-She Yang N. 3 Function:

$$f(\mathbf{x}) = f(x_1, \ldots, x_n) = exp(-\sum_{i=1}^{n} (x_i/\beta)^{2m}) - 2exp(-\sum_{i=1}^{n} x_i^2) \prod_{i=1}^{n} cos^2(x_i)$$

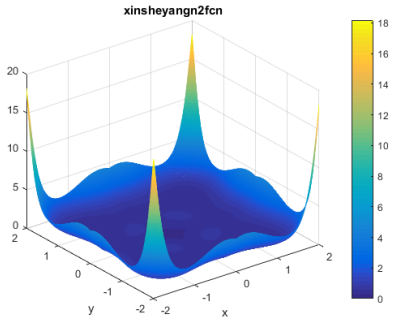

Figure 9: Landscape of Xin-She Yang N. 2 Function

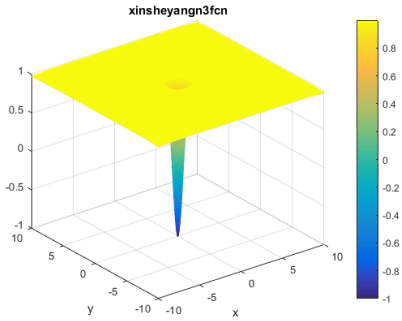

Figure 10: Landscape of Xin-She Yang N. 3 Function

4. Results of Extension

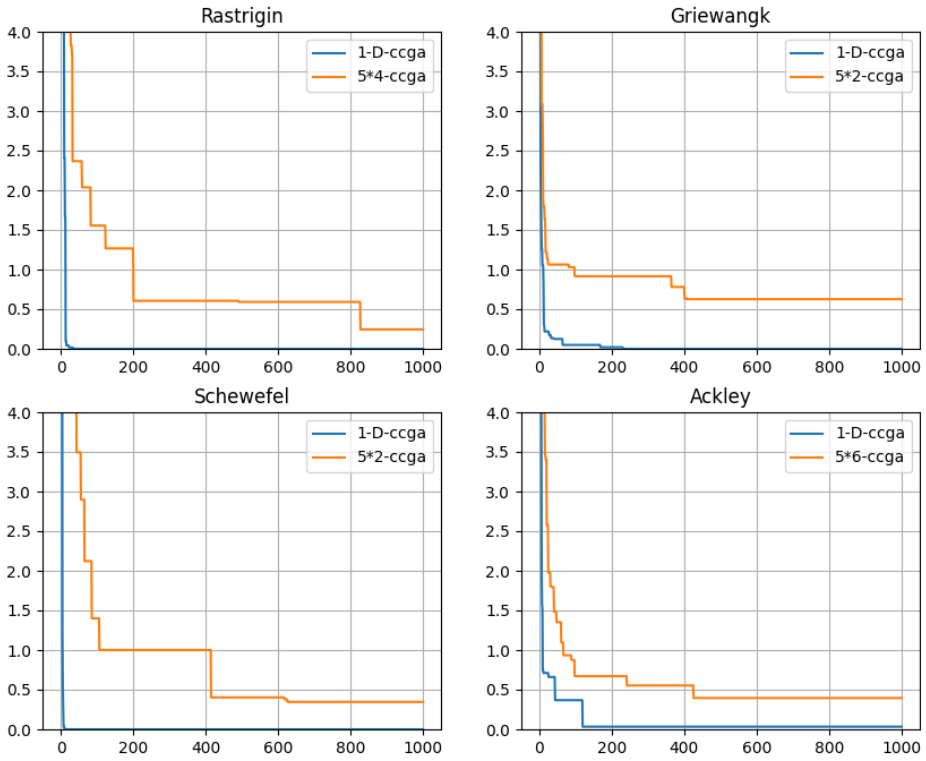

Figure 11: variable grouping performance on original task

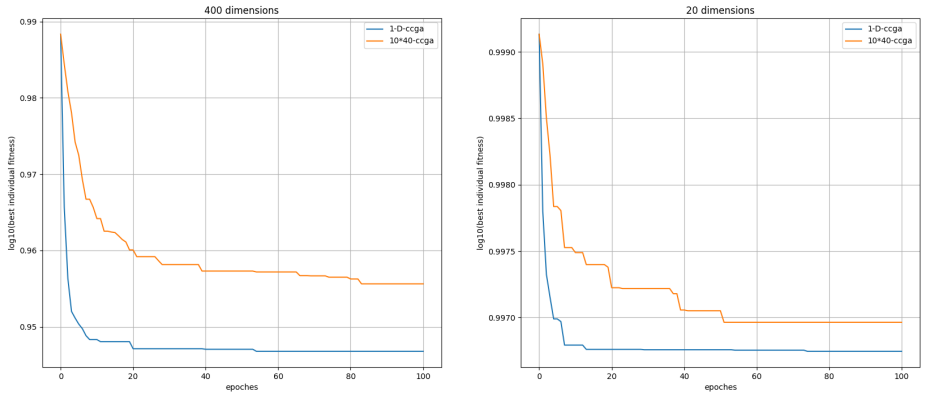

Figure 12: variable grouping performance on Xin-She Yang N. 3 Function

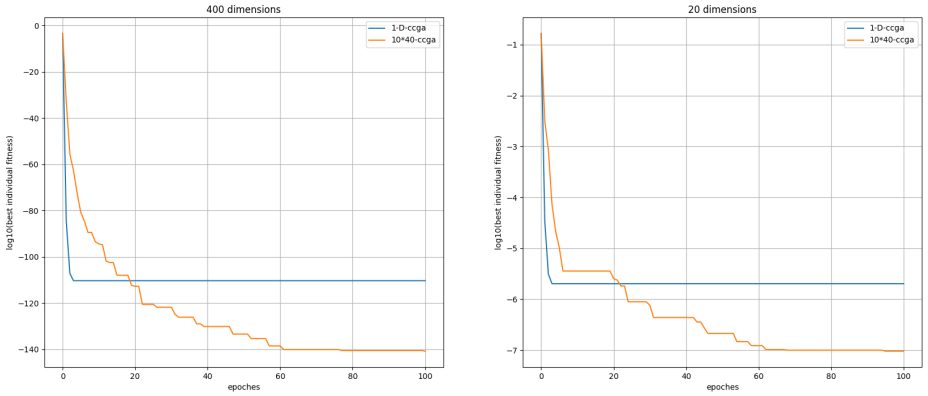

Figure 13: variable grouping performance on Xin-She Yang N. 2 Function

5. Conclusion

Although from the experimental results, I find that CCGA with variable grouping is not necessarily better at solving non-separable function optimization problems than the 1-Dimensional CCGA model. This is completely different from what I assumed before. But the advanced CCGA algorithm is guaranteed in running time, only half of the original time is needed. In addition, for some specific problems as Xin-She Yang N.2 Function, variable grouping works better.

Since Xin-She Yang N.2 Function and Xin-She Yang N.3 Function are both non-separable and there is no proof that Xin-She Yang N.2 Function has more parameter linkage than Xin-She Yang N.3 Function, the results in **Figure: 13 12** shows that variable grouping does not guarantee a better performance on non-separable functions. In most function optimization tasks, the CCGA model with only one dimension seems to be the best strategy overall, however, as I have noted before, the real-life issues are not as simple as function optimization, there are much more dimensions and parameter linkages. Many variable grouping strategies have been proposed to deal with this problem, including static variable grouping, random variable grouping, linkage learning-based variable grouping and etc. For different tasks, CCGA model with different grouping strategy should be applied.

References

[1] M. A. Potter and K. A. De Jong, "A cooperative coevolutionary approach to function optimization," in *International Conference on Parallel Problem Solving from Nature*. Springer, 1994, pp. 249–257.

[2] R. P. Wiegand, "An analysis of cooperative coevolutionary algorithms," Ph.D. dissertation, Citeseer, 2003.

[3] M. A. Potter and K. A. D. Jong, "Cooperative coevolution: An architecture for evolving coadapted subcomponents," *Evolutionary computation*, vol. 8, no. 1, pp. 1–29, 2000.

[4] Z. Cao, L. Wang, Y. Shi, X. Hei, X. Rong, Q. Jiang, and H. Li, "An effective cooperative coevolution framework integrating global and local search for large scale optimization problems," in *2015 IEEE Congress on Evolutionary Computation (CEC)*, 2015, pp. 1986–1993.

[5] Z. Yang, K. Tang, and X. Yao, "Differential evolution for high-dimensional function optimization," in *2007 IEEE Congress on Evolutionary Computation*, 2007, pp. 3523–3530.

[6] X. Ma, X. Li, Q. Zhang, K. Tang, Z. Liang, W. Xie, and Z. Zhu, "A survey on cooperative co-evolutionary algorithms," *IEEE Transactions on Evolutionary Computation*, vol. 23, no. 3, pp. 421–441, 2018.

6. Appendix

```
import numpy as np
import math
import random
from TestFunction import Rastrigin, Schewefel, Griewank, Ackley, Xin_She_Yang, Xin_She_Yang_2
import time
from tqdm import tqdm
import pandas as pd
from decimal import Decimal
from PyBenchFCN import SingleObjectiveProblem as SOP


class IndiString:
    def __init__(self, string, divide):
        self.represent = string
        self.divide_point = divide
        self.value = self.__setValue__()

    def __setValue__(self):
        num = int(self.represent[1:17], 2) / math.pow(2, self.divide_point)
        if self.represent[0] == 0:
            num = - num
        while num > max_value or num < min_value:
            num = num / 2
            self.divide_point = self.divide_point + 1
        return num

    def __getValue__(self):
        num = int(self.represent[1:17], 2) / math.pow(2, self.divide_point)
        if self.represent[0] == 0:
            num = - num
        return num


def init_pop(population_size, n):
    pop_list = []
    while len(pop_list) < population_size:
        pop_individual = []
        while len(pop_individual) < n:
            new_indi = IndiString(initStringValue(), 0)
            pop_individual.append(new_indi)
        pop_list.append(pop_individual)
    return pop_list


def initStringValue():
    seed = "01"
    sa = []
    for i in range(17):
        sa.append(seed[np.random.randint(0, 2)])
    salt = ''.join(sa)
```

```
    return salt


def get_random(key_list):
    L = len(key_list)
    i = np.random.randint(0, L)
    return key_list[i]


def crossover_mute(func, list, crossover_rate):
    individual_A = get_random(list)
    individual_B = get_random(list)
    individual_C = get_random(list)
    individual_D = get_random(list)
    parent_A = individual_A if func(individual_A) < func(individual_B) else individual_B
    parent_B = individual_C if func(individual_C) < func(individual_D) else individual_D

    child_list = []
    if random.uniform(0, 1) < crossover_rate:
        for i in range(len(parent_A)):
            str_len = min((len(parent_A[i].represent), len(parent_B[i].represent)))
            point_one = random.randint(0, str_len)
            point_two = random.randint(point_one, str_len)
            new_string = parent_A[i].represent[0:point_one] + parent_B[i].represent[point_one
                         parent_A[i].represent[point_two:]
            new_string = mutate(new_string)
            new_indi = IndiString(new_string, parent_A[i].divide_point)
            child_list.append(new_indi)
        return child_list
    else:
        child = parent_A if func(parent_A) < func(parent_B) else parent_B
        return child


def crossover_mute_ccga(func, list, crossover_rate):
    individual_A = get_random(list)
    individual_B = get_random(list)
    individual_C = get_random(list)
    individual_D = get_random(list)
    parent_A = individual_A if func(individual_A) < func(individual_B) else individual_B
    parent_B = individual_C if func(individual_C) < func(individual_D) else individual_D
    string_len = len(parent_A[0].represent)
    if random.uniform(0, 1) < crossover_rate:
        point_one = random.randint(0, string_len)
        point_two = random.randint(point_one, string_len)
        new_string = parent_A[0].represent[0:point_one] + parent_B[0].represent[point_one:poi
                     + parent_A[0].represent[point_two:]
        new_string = mutate(new_string)
        new_indi = IndiString(new_string, parent_A[0].divide_point)
        return [new_indi]
    else:
        child = parent_A if func(parent_A) < func(parent_B) else parent_B
```

```
        return child


def mutate(string):
    origin_string = string
    set_string = ''
    for char in origin_string[0:]:
        if random.uniform(0, 1) < 1 / 16:
            mute = '0' if char == '1' else '1'
            set_string = set_string + mute
        else:
            set_string = set_string + char
    return set_string


def ccga_model(pop_list, func, epoch):
    eval_list = list(map(func, pop_list))
    fitness_max = [np.min(eval_list)]
    fitness_min = [np.max(eval_list)]
    for _ in tqdm(range(epoch)):
        species_list = []
        gen_list = [pop_list[np.nanargmin(eval_list)]]
        for s in range(0, num_parameter):
            s_list = []
            s_return_list = []
            for pop in pop_list:
                s_list.append([pop[s]])
            s_eval_list = list(map(func, s_list))
            fmin = np.max(s_eval_list)
            f_list = fmin - s_eval_list
            s_list = scaling_window(f_list, s_list)
            while len(s_return_list) < len(init_list) - 1:
                child = crossover_mute_ccga(func, s_list, crossover_rate=0.6)
                s_return_list.extend(child)
            species_list.append(s_return_list)
        # re-evaluate
        gen_list.extend(list(map(list, zip(*species_list))))
        eval_list = list(map(func, gen_list))
        fitness_max.append(np.min(eval_list))
        fitness_min.append(np.max(eval_list))
        pop_list = gen_list

    return pop_list, fitness_max


# extend s*d
def ccga_svg_model(pop_list, func, epoch):
    eval_list = list(map(func, pop_list))
    fitness_max = [np.min(eval_list)]
    fitness_min = [np.max(eval_list)]
    for _ in tqdm(range(epoch)):
        species_list = []
```

```
        gen_list = [pop_list[np.nanargmin(eval_list)]]
        for s in range(0, 10):
            s_list = []
            for pop in pop_list:
                s_list.append([pop[s*10], pop[s*10+1], pop[s*10+2], pop[s*10+3], pop[s*10+4],
                               pop[s*10+5], pop[s*10+6], pop[s*10+7], pop[s*10+8], pop[s*10+9
            s_eval_list = list(map(func, s_list))
            fmin = np.max(s_eval_list)
            f_list = fmin - s_eval_list
            s_list = scaling_window(f_list, s_list)
            a_list = []
            for i in range(0, 100):
                s_return_list = []
                while len(s_return_list) < len(init_list):
                    child = crossover_mute(func, s_list, crossover_rate=0.6)
                    s_return_list.extend(child)
                a_list.append(s_return_list)
            species_list.extend(a_list)
        # re-evaluate
        gen_list.extend(list(map(list, zip(*species_list)))[0:-1])
        eval_list = list(map(func, gen_list))
        fitness_max.append(np.min(eval_list))
        fitness_min.append(np.max(eval_list))
        pop_list = gen_list

    return pop_list, fitness_max



def ccga_svgr_model(pop_list, func, epoch):
    eval_list = list(map(func, pop_list))
    fitness_max = [np.min(eval_list)]
    fitness_min = [np.max(eval_list)]
    for _ in tqdm(range(epoch)):
        species_list = []
        gen_list = [pop_list[np.nanargmin(eval_list)]]
        for s in range(0, 5):
            s_list = []
            for pop in pop_list:
                s_list.append([pop[random.randint(0, num_parameter-1)], pop[random.randint(0,
                               pop[random.randint(0, num_parameter-1)],pop[random.randint(0,
            s_eval_list = list(map(func, s_list))
            fmin = np.max(s_eval_list)
            f_list = fmin - s_eval_list
            s_list = scaling_window(f_list, s_list)
            a_list = []
            for i in range(0, 200):
                s_return_list = []
                while len(s_return_list) < len(init_list):
                    child = crossover_mute(func, s_list, crossover_rate=0.6)
                    s_return_list.extend(child)
                a_list.append(s_return_list)
```

```
            species_list.extend(a_list)
        # re-evaluate
        gen_list.extend(list(map(list, zip(*species_list)))[0:-1])
        eval_list = list(map(func, gen_list))
        fitness_max.append(np.min(eval_list))
        fitness_min.append(np.max(eval_list))
        pop_list = gen_list

    return pop_list, fitness_max



def scaling_window(fitness_list, pop_list):
    if sum(fitness_list) != 0:
        proportion = fitness_list / sum(fitness_list) * len(fitness_list)
        proportion_list = np.array(list(map(round, proportion)))
    else:
        proportion_list = np.ones(len(fitness_list))
        # pure strategy select
    select_1 = np.where(proportion_list == 1)[0]
    select_2 = np.where(proportion_list > 1)[0]
    if len(select_2) != 0:
        pop_list = [pop_list[i] for i in select_1] + 2 * [pop_list[i] for i in select_2]
    else:
        pop_list = [pop_list[i] for i in select_1]
    return pop_list


def ga_model(pop_list, func, epoch):
    eval_list = list(map(func, pop_list))
    fitness_max = [np.min(eval_list)]
    fitness_min = [np.max(eval_list)]
    for i in tqdm(range(epoch)):
        # applying scaling window
        gen_list = [pop_list[np.nanargmin(eval_list)]]
        fmin = fitness_min[0] if i < 6 else fitness_min[i - 5]
        fitness_list = fmin - eval_list
        pop_list = scaling_window(fitness_list, pop_list)
        while len(gen_list) < len(init_list):
            # two-point crossover
            child = crossover_mute(func, pop_list, crossover_rate=0.6)
            gen_list.append(child)
        # re-evaluate
        eval_list = list(map(func, gen_list))
        fitness_max.append(np.min(eval_list))
        fitness_min.append(np.max(eval_list))
        pop_list = gen_list
    return pop_list, fitness_max


if __name__ == '__main__':
    pupulation_size = 100
```

```
num_parameter = 20
min_value = -5.12
max_value = 5.12
init_list = init_pop(population_size=100, n=num_parameter)
pop_list_ccga, fitness_ccga = ccga_model(init_list, Rastrigin, 1000)
np.save('Rccga.npy', np.array(fitness_ccga))
pop_list_svg_ccga, fitness_svg_ccga = ccga_svgr_model(init_list, Rastrigin, 1000)
np.save('R_s_ccga.npy', np.array(fitness_svg_ccga))

pupulation_size = 100
num_parameter = 10
min_value = -500
max_value = 500
init_list = init_pop(population_size=100, n=num_parameter)
pop_list_ga, fitness_max= ccga_model(init_list, Schewefel, 1000)
np.save('S_ga.npy', np.array(fitness_max))
pop_list_ccga, fitness_ccga = ccga_svgr_model(init_list, Schewefel, 1000)
np.save('S_s_ccga.npy', np.array(fitness_ccga))



pupulation_size = 100
num_parameter = 30
min_value = -30
max_value = 30
init_list = init_pop(population_size=100, n=num_parameter)
pop_list_ga, fitness_max = ccga_model(init_list, Ackley, 1000)
np.save('A_ga.npy', np.array(fitness_max))
pop_list_ccga, fitness_ccga = ccga_svgr_model(init_list, Ackley, 1000)
np.save('A_s_ccga.npy', np.array(fitness_ccga))

# #
pupulation_size = 100
num_parameter = 10
min_value = -512
max_value = 512
init_list = init_pop(population_size=100, n=num_parameter)
pop_list_ga, fitness_max= ccga_model(init_list, Griewank, 1000)
np.save('G_ga.npy', np.array(fitness_max))
pop_list_ccga, fitness_ccga = ccga_svgr_model(init_list, Griewank, 1000)
np.save('G_svg_ccga.npy', np.array(fitness_ccga))
#
#


pupulation_size = 100
num_parameter = 1000
min_value = -2*np.pi
max_value = 2*np.pi
init_list = init_pop(population_size=100, n=num_parameter)
# # pop_list_ga, fitness_max= ga_model(init_list, Xin_She_Yang_2, 100)
```

```
    # # print(fitness_max)
    # # np.save('X_ga_2_400.npy', np.array(fitness_max))
    pop_list_ccga, fitness_ccga = ccga_model(init_list, Xin_She_Yang, 10)
    print(fitness_ccga)
    np.save('X_ccga_1_1000.npy', np.array(fitness_ccga))
    pop_list_ccga, fitness_ccga = ccga_svgr_model(init_list, Xin_She_Yang, 10)
    print(fitness_ccga)
    np.save('X_svg_ccga_1_1000.npy', np.array(fitness_ccga))


import numpy as np

import math


def Rastrigin(chromosome):
    fitness = 3 * len(chromosome)
    for i in range(len(chromosome)):
        fitness += chromosome[i].value ** 2 - (3 * math.cos(2 * math.pi * chromosome[i].value
    return fitness


def Schewefel(chromosome):
    alpha = 418.982887
    fitness = alpha * len(chromosome)
    for i in range(len(chromosome)):
        fitness -= chromosome[i].value * math.sin(math.sqrt(math.fabs(chromosome[i].value)))
    return fitness


def Griewank(chromosome):
    part1 = 0
    for i in range(len(chromosome)):
        part1 += chromosome[i].value ** 2
        part2 = 1
    for i in range(len(chromosome)):
        part2 *= math.cos(float(chromosome[i].value) / math.sqrt(i + 1))
    return 1 + (float(part1) / 4000.0) - float(part2)


def Ackley(chromosome):
    firstSum = 0.0
    secondSum = 0.0
    for c in chromosome:
        firstSum += c.value ** 2.0
        secondSum += math.cos(2.0 * math.pi * c.value)
    n = float(len(chromosome))
    return -20.0 * math.exp(-0.2 * math.sqrt(firstSum / n)) - math.exp(secondSum / n) + 20 +


def Xin_She_Yang(chromosome):
    firstex = 0.0
```

```
    secondex = 0.0
    multi = 0.0
    for c in chromosome:
        firstex += math.pow(c.value/15.0, 10)
        secondex += c.value ** 2
        multi = multi* (np.cos(c.value)**2)

    return math.exp(-firstex) - 2*math.exp(-secondex)*multi


def Xin_She_Yang_2(chromosome):
    firstsum = 0.0
    secondsum = 0.0
    for c in chromosome:
        firstsum += math.fabs(c.value)
        secondsum += math.sin(c.value**2)

    return  firstsum*math.exp(-secondsum)



import numpy as np
from matplotlib import pyplot as plt
import math
Rastrigin_ga_result = np.load('Rastrigin_ga.npy')
Rastrigin_ccga_result = np.load('Rastrigin_ccga.npy')
Griewank_ga_result = np.load('Griewank_ga.npy')
Griewank_ccga_result = np.load('Griewank_ccga.npy')
Schewefel_ga_result = np.load('Schewefel_ga.npy')
Schewefel_ccga_result = np.load('Schewefel_ccga.npy')
Ackley_ga_result = np.load('Ackley_ga.npy')
Ackley_ccga_result = np.load('Ackley_ccga.npy')

X_ga = np.load('X_ga_400.npy')
X_ccga = np.load('X_ccga_400.npy')
X_svg_ccga = np.load('X_svg_ccga_400.npy')

X_ga = np.load('X_ga_400.npy')
X_ccga_2 = np.load('X_ccga_2_400.npy')
X_svg_ccga_2 = np.load('X_svg_ccga_2_400.npy')
X_ccga_2_2 = np.load('X_ccga_2_20.npy')
X_svg_ccga_2_2 = np.load('X_svg_ccga_2_20.npy')

X_ccga_1 = np.load('X_ccga_1_400.npy')
X_svg_ccga_1 = np.load('X_svg_ccga_1_400.npy')
X_ccga_1_2 = np.load('X_ccga_1_20.npy')
X_svg_ccga_1_2 = np.load('X_svg_ccga_1_20.npy')



R_ccga = np.load('Rccga.npy')
R_s_ccga = np.load('R_s_ccga.npy')
```

```
S_ga = np.load('S_ga.npy')
S_s_ccga = np.load('S_s_ccga.npy')
G_ccga = np.load('G_ga.npy')
G_s_ccga = np.load('G_svg_ccga.npy')
A_ccga = np.load('A_ga.npy')
A_s_ccga = np.load('A_ccga.npy')

fig, ax = plt.subplots(2, 2, figsize=(8, 8))
# print(np.unique(Rastrigin_ga_result))
# print(np.unique(Rastrigin_ccga_result))



# x_ccga_list = list(map(math.log10, X_ccga_2))
# x_svg_list = list(map(math.log10, X_svg_ccga_2))
# x_ccga_list_2 = list(map(math.log10, X_ccga_2_2))
# x_svg_list_2 = list(map(math.log10, X_svg_ccga_2_2))
# ax[0].plot(range(len(X_ccga_2)), X_ccga_1, label='1-D-ccga')
# ax[0].plot(range(len(X_svg_ccga_2)), X_svg_ccga_1, label='10*40-ccga')
# ax[0].set_ylabel('log10(best individual fitness)')
# ax[0].set_xlabel('epoches')
# ax[0].set_title('400 dimensions')
# ax[0].legend()
# ax[0].grid()
# ax[1].plot(range(len(X_ccga_2_2)), X_ccga_1_2, label='1-D-ccga')
# ax[1].plot(range(len(X_svg_ccga_2_2)), X_svg_ccga_1_2, label='10*40-ccga')
# ax[1].set_ylabel('log10(best individual fitness)')
# ax[1].set_xlabel('epoches')
# ax[1].set_title('20 dimensions')
# ax[1].legend()
# ax[1].grid()
ax[0][0].plot(range(len(R_ccga)), R_ccga,  label='1-D-ccga')
ax[0][0].plot(range(len(R_s_ccga)), R_s_ccga, label='5*4-ccga')
ax[0][0].set_title('Rastrigin')
ax[0][0].legend()
ax[0][0].grid()
ax[0][0].set_ylim(0, 4)
ax[1][0].plot(range(len(S_ga)), S_ga,  label='1-D-ccga')
ax[1][0].plot(range(len(S_s_ccga)), S_s_ccga, label='5*2-ccga')
ax[1][0].set_title('Schewefel')
ax[1][0].set_ylim(0, 4)
ax[1][0].legend()
ax[1][0].grid()
ax[0][1].plot(range(len(G_ccga)), G_ccga, label='1-D-ccga')
ax[0][1].plot(range(len(G_s_ccga)), G_s_ccga, label='5*2-ccga')
ax[0][1].set_title('Griewangk')
ax[0][1].set_ylim(0, 4)
ax[0][1].legend()
ax[0][1].grid()
ax[1][1].plot(range(len(A_ccga)), A_ccga, label='1-D-ccga')
ax[1][1].plot(range(len(A_s_ccga)), A_s_ccga, label='5*6-ccga')
ax[1][1].set_title('Ackley')
```

```
ax[1][1].set_ylim(0, 4)
plt.show()

# ax.set_xlabel('function evaluations')
# ax.set_ylabel('best individual')
# plt.savefig('static variable')
# plt.savefig('xinshenyang_1')


# fig, ax = plt.subplots(figsize=(8, 6))
# ax.plot(range(len(G_ccga)), G_ccga, label='standard_ga')
# ax.plot(range(len(G_s_ccga)), G_s_ccga, label='ccga')
# ax.set_xlabel('function evaluations')
# ax.set_ylabel('best individual')
# ax.set_ylim(0, 50)
# ax.set_title('Schewefel function')
# plt.grid()
# plt.legend()
# plt.show()
# plt.savefig('Schewefel function')
```