9

Cuckoo Search Algorithm

Xin-She Yang
School of Science and Technology
Middlesex University, London, United Kingdom

Adam Slowik
Department of Electronics and Computer Science
Koszalin University of Technology, Koszalin, Poland

CONTENTS

9.1 Introduction

The original cuckoo search (CS) was first developed by Xin-She Yang and Suash Deb in 2009 [1], which was inspired by the brooding parasitism of some cuckoo species. CS has been used for engineering optimization [2], and has been extended to multiobjective optimization [3], and structural optimization problems [4]. In fact, the literature is rapidly expanding with a diverse range of applications [5]. One of the main advantages of cuckoo search is that it uses Lévy flights [6] that consist of a fraction of long-distance, non-local moves in addition to many local moves, which makes the algorithm much more efficient in exploring the sparse search space [5, 6]. Other variants have been developed, including the quantum-inspired CS [7], modified cuckoo search [8], random-key discrete cuckoo search [9] and others [10]. Recently, CS has been extended to a new variant of multi-species cuckoo search [11]. For a more comprehensive

review of the cuckoo search, its variants and applications, please refer to more advanced literature [5, 10].

The rest of this chapter will first introduce the main steps of the original cuckoo search, and explain in detail how to carry out Lévy flights and implementations, followed by the introduction of the demo codes in both Matlab and C++.

9.2 Original cuckoo search

Many cuckoo species engage in aggressive reproduction strategies by laying eggs in the nests of host species so as to let host species to raise their young chicks. From the evolutionary point of view, this maximizes the reproductivity probability of the cuckoo species. However, host species can also fight back by abandoning such eggs or nests with contaminated alien eggs. Thus, there is an ongoing arms race between cuckoo species and host species. Cuckoo search has been developed, based on the inspiration of such cuckoo-host characteristics.

9.2.1 Description of the cuckoo search

In essence, the main characteristics for the cuckoo search with a population of n cuckoos can be summarized as

- Each cuckoo lays one egg and dumps it into a randomly chosen host nest. An egg can be considered as a solution vector $\boldsymbol{x}$ to an optimization problem.
- The best eggs in nests will be passed on to the next generation. This means the best solutions are retained.
- The number of available nests is fixed, and each egg laid by a cuckoo may be discovered and abandoned with a probability p_a. This is equivalent to that a fraction p_a of the total population will be modified at each iteration t.

In the real-world, each nest of a host bird typically has 3 or 4 or more eggs, and each cuckoo can attack quite a few nests by laying its eggs in such nests. For simplicity, we assume that each cuckoo can only lay one egg and affect one host nest, which means that the number of eggs is equal to the number of nests and cuckoos. Thus, we can encode the location of an egg as a solution vector to an optimization problem. This way, there is no need to distinguish eggs, cuckoos and nests. Thus, we have 'egg=cuckoo=nest'.

In the original CS, there are two main equations for two different actions. One equation is

$$\boldsymbol{x}_i^{t+1} = \boldsymbol{x}_i^t + \alpha s \otimes H(p_a - \epsilon) \otimes (\boldsymbol{x}_j^t - \boldsymbol{x}_k^t), \tag{9.1}$$

where $\boldsymbol{x}_j^t$ and $\boldsymbol{x}_k^t$ are two different solution vectors, randomly selected from the population for $i = 1, 2, ..., n$. The switching condition with a probability p_a is realized by a Heaviside function (or a step function) by comparing p_a with a random number ϵ drawn from a uniform distribution in [0, 1]. In addition, s is a step size by random permutation of solution differences, and this is scaled by a scaling factor α. Furthermore, the product of two vectors is an entry-wise multiplication using $\otimes$.

The other equation is mainly for non-local random walks by Lévy flights

$$\boldsymbol{x}_i^{t+1} = \boldsymbol{x}_i^t + \alpha L(s, \beta), \tag{9.2}$$

where the step sizes are drawn from the Lévy distribution

$$L(s, \beta) \sim \frac{\beta \Gamma(\beta) \sin(\pi\beta/2)}{\pi} \frac{1}{s^{1+\beta}}, \tag{9.3}$$

which is an approximation to a Lévy distribution with an exponent $0 \leq \beta \leq 2$. The gamma function is defined by

$$\Gamma(q) = \int_0^\infty z^{q-1} e^{-u} du. \tag{9.4}$$

The notation '$\sim$' highlights the fact that this is a pseudo-random number to be drawn from the probability distribution on the right-hand side of the above equation. Strictly speaking, the Lévy step sizes should be given by the inverse integral form of the following Fourier transform [6]

$$L(s, \beta) = \frac{1}{\pi} \int_0^\infty \cos(ks) \exp[-w|k|^\beta] dk, \tag{9.5}$$

where w is a scaling parameter.

However, the realization of this integral is difficult, so we will use an efficient but simplified algorithm, called Mantegna's algorithm, in the implementation below.

9.2.2 Pseudo-code of CS

Based on the above descriptions, the main steps of the CS consist of initialization, the iteration loop, modifications of solutions by two different ways. The pseudo-code is shown in Algorithm 8.

9.2.3 Parameters in the cuckoo search

Comparing with other swarm optimization algorithms, CS has a relatively low number of parameters, which makes it easier to tune the algorithm. The three parameters are population size n, switching probability p_a and the Lévy exponent β. In our implementation here, we have used $n = 25$, $p_a = 0.25$ and $\beta = 1.5$.

Algorithm 8 Pseudo-code of the cuckoo search algorithm.

```
Define the objective function f(x)
Initialize all the parameters
Generate an initial population of n nests x_i (i = 1, 2, ..., n)
for t=1:MaxGeneration do
    Get a cuckoo/solution randomly
    Generate a solution by Lévy flights [Eq. (9.3)]
    Evaluate the new solution x_i and its fitness f_i
    Choose a nest (say, j) among n nests randomly
    if f_i < f_j then
        Replace x_j by x_i
    end if
    A fraction (p_a) of the worse nests are abandoned
    New nests/solutions are generated by Eq. (9.1)
    Update the best solution if a better solution is found
end for
Output results
```

9.3 Source code of the cuckoo search in Matlab

The implementation of the cuckoo search algorithm is straightforward using Matlab, though care should be taken when generating Lévy flights in terms of drawing random numbers.

For simplicity and to be consistent with other tutorial chapters such as the tutorial on the firefly algorithm, we use the same function benchmark $f(\boldsymbol{x})$

$$\text{minimize} \quad f(\boldsymbol{x}) = (x_1 - 1)^2 + (x_2 - 1)^2 + ... + (x_D - 1)^2, \quad x_i \in \mathbb{R}, \tag{9.6}$$

which has a global minimum of $\boldsymbol{x}_* = (1, 1, ..., 1)$. We also apply the following lower bound (Lb) and upper bound (Ub):

$$Lb = [-5, \ -5, \ ..., \ -5], \quad Ub = [+5, \ +5, \ ..., \ +5]. \tag{9.7}$$

The Matlab code for the cuckoo search algorithm here consists of five parts: initialization, the main loop, Lévy flights, simple bounds, and the objective function. The whole codes should consist of all the lines of codes in a sequential order. For ease of description, we discuss each part in sequential order.

The first part is mainly initialization of parameter values such as the population size $n = 25$ and the switching probability $pa = 0.25$. All the solutions of 25 nests or cuckoos in the initial population are evaluated so as to obtain the initial best solution. As mentioned earlier, we do not distinguish among an egg, a nest or a cuckoo and we have used here 'an egg=a cuckoo=a nest=a solution vector'.

```
function [bestnest,fmin]=cuckoo_search_demo
%% Initialization of parameters
n=25;          % Number of nests (or population size)
pa=0.25;       % Discovery rate of alien eggs/solutions
nd=15;         % Number of dimensions
Lb=-5*ones(1,nd);  % Lower bounds
Ub=5*ones(1,nd);   % Upper bounds
N_IterTotal=1000;  % Increase it to get better results

% Random initial solutions for the population with n cuckoos
for i=1:n,
nest(i,:)=Lb+(Ub-Lb).*rand(size(Lb));
end
% Get the best solution among the initial population
fitness=10^10*ones(n,1);
[fmin,bestnest,nest,fitness]=get_best_nest(nest,nest,fitness);
```

Listing 9.1
CS demo initialization.

The second part is the main part, consisting of a loop over the whole cuckoo population at each iteration. New cuckoos or solutions are generated by `get_cuckoos()` or `empty_nests()` subroutines, which will be explained below.

```
%% Starting iterations
for iter=1:N_IterTotal,
    % Generate new solutions (but keep the current best)
     new_nest=get_cuckoos(nest,bestnest,Lb,Ub);
     [fnew,best,nest,fitness]=get_best_nest(nest,new_nest,fitness);
    % Discovery and randomization
      new_nest=empty_nests(nest,Lb,Ub,pa) ;

    % Evaluate this set of solutions
      [fnew,best,nest,fitness]=get_best_nest(nest,new_nest,fitness);
    % Find the best objective so far
    if fnew<fmin,
        fmin=fnew;
        bestnest=best;
    end
end %% End of iterations
%% Display all the nests
disp(strcat('Best solution=',num2str(bestnest)));
disp(strcat('Best objective=',num2str(fmin)));
```

Listing 9.2
Main part for the cuckoo search algorithm.

The third part is about the Lévy flights and the selection of the best solution. The generation of pseudo-random numbers obeying the Lévy distribution is carried out by Mantegna's algorithm [12]

$$s = \frac{u}{|v|^{1/\beta}}, \tag{9.8}$$

where u and v are drawn from normal distributions

$$u \sim N(0, \sigma_u^2), \quad v \sim N(0, 1). \tag{9.9}$$

Here, σ^2 should be calculated by

$$\sigma = \left\{ \frac{\Gamma(1+\beta)\sin(\pi\beta/2)}{\beta\,\Gamma[(1+\beta)/2]\,2^{(\beta-1)/2}} \right\}^{1/\beta}, \tag{9.10}$$

where Γ is the standard gamma function. In practice, it is easier to use a fixed σ value, together with scaling factor such as 0.01. Thus, the Lévy flight step is replaced by

$$u \sim N(0, \sigma^2), \quad v \sim N(0, 1), \tag{9.11}$$

and

$$s = \frac{u}{|v|^{1/\beta}}, \quad \beta = 3/2, \tag{9.12}$$

which gives stepsizes for modifying $\boldsymbol{x}_i$ as

$$\text{stepsize} = 0.01(\boldsymbol{x}_i - \boldsymbol{x}_{\text{best}}) \cdot \frac{u}{|v|^{1/\beta}}. \tag{9.13}$$

```
%% Get cuckoos by ramdom walk
function nest=get_cuckoos(nest,best,Lb,Ub)
% Carry out Levy flights
n=size(nest,1);
beta=3/2;
sigma=(gamma(1+beta)*sin(pi*beta/2)/(gamma((1+beta)/2)*beta*2^((beta
    -1)/2)))^(1/beta);

for j=1:n,
    s=nest(j,:);
    %% Levy flights by Mantegna's algorithm
    u=randn(size(s))*sigma;
    v=randn(size(s));
    step=u./abs(v).^(1/beta);
    stepsize=0.01*step.*(s-best);
    s=s+stepsize.*randn(size(s));
    %% Apply simple bounds/limits
    nest(j,:)=simplebounds(s,Lb,Ub);
end

%% Find the current best nest
function [fmin,best,nest,fitness]=get_best_nest(nest,newnest,fitness)
% Evaluating all new solutions
for j=1:size(nest,1),
    fnew=fobj(newnest(j,:));
    if fnew<=fitness(j),
       fitness(j)=fnew;
       nest(j,:)=newnest(j,:);
    end
end
% Find the current best solution
[fmin,K]=min(fitness) ;
best=nest(K,:);

%% Replace some nests by constructing new solutions/nests
function new_nest=empty_nests(nest,Lb,Ub,pa)
% A fraction of worse nests are discovered with a probability pa
n=size(nest,1);
% Discovered or not -- a status vector
K=rand(size(nest))>pa;
%% Generate new solutions by biased/selective random walks
stepsize=rand*(nest(randperm(n),:)-nest(randperm(n),:));
new_nest=nest+stepsize.*K;
for j=1:size(new_nest,1)
    s=new_nest(j,:);
  new_nest(j,:)=simplebounds(s,Lb,Ub);
end
```

Listing 9.3
Lévy flights and selection of the best.

The fourth part applies the simple lower and upper bounds. This will ensure the new solution vector should be within the regular bounds.

```
% Application of simple bounds
function s=simplebounds(s,Lb,Ub)
  % Apply the lower bound
  ns_tmp=s;
  I=ns_tmp<Lb;
  ns_tmp(I)=Lb(I);

  % Apply the upper bounds
  J=ns_tmp>Ub;
  ns_tmp(J)=Ub(J);
  % Update this new move
  s=ns_tmp;
```

Listing 9.4
Simple bounds are verified for new solutions.

The fifth part is the objective function. New solutions can be evaluated by calling the objective or cost function.

```
%% Cost or objective function
function z=cost(x)
z=sum((x-1).^2); % Solutions should be (1,1,...,1)
```

Listing 9.5
The objective function.

If we run this code with a maximum number of 1000 iterations, we can get the best minimum value as $f_{\text{best}} = 3.5 \times 10^{-12}$ for 15 decision variables. If we use $D = 5$ (five variables), we can easily get about 2.7×10^{-31}.

Since such algorithms contain randomness, multiple runs are needed to get some proper statistics and statistical measures.

9.4 Source code in C++

For the CS, we have also implemented it in C++. However, as there is no standard library for Lévy flights, we have also implemented the Lévy flights in C++ code below.

```
#include <iostream>
#include <time.h>
#include <cmath>
#include <random>
#include <algorithm>
using namespace std;
float simplebounds(float s, float Lb, float Ub)
{if (s>Ub) {s=Ub;} if (s<Lb) {s=Lb;} return s;}
//A very simple approximation of gamma function
float gamma(float x)
{return sqrt(2.0*M_PI/x)*pow(x/M_E, x);}
//Definition of the objective function OF(.)
float fobj(float x[], int size_array)
{float t=0;
for(int i=0; i<size_array; i++){t=t+(x[i]-1)*(x[i]-1);}
```

```
return t;}
//Generate pseudo random numbers in the range [0, 1)
float ra(){return (float)rand()/RAND_MAX;}
int main()
{//Normal distribution generator
std::default_random_engine generator;
std::normal_distribution<double> distribution(0,1.0);
srand(time(NULL));
int n=25;        //Number of nests (or population size)
float pa=0.25;   //Discovery rate of alien eggs/solutions
int nd=15;       //Number of dimensions
float Lb[nd];    //Lower bounds
float Ub[nd];    //Upper bounds
float fitness[n];      //Fitness values
float beta=3/2, sigma, fmin, best[nd], bestnest[nd];
int K, r1, r2, r, randperm1[n], randperm2[n];
for(int i=0; i<nd; i++){Lb[i]=-5; Ub[i]=5;}
int N_IterTotal=1000; //Increase it get better results
//Random initial solutions for the population with n cuckoos
float nest[n][nd], new_nest[n][nd];
for(int i=0; i<n; i++){
    for(int j=0; j<nd; j++){
        nest[i][j]=Lb[j]+(Ub[j]-Lb[j])*ra();}}
for(int i=0; i<n; i++){
    fitness[i]=fobj(nest[i],nd);
    randperm1[i]=i; randperm2[i]=i;}
K=0; fmin=fitness[K];
for(int i=1; i<n; i++){
    if (fitness[i]<=fmin){
        fmin=fitness[i];
        K=i;}}
for(int j=0; j<nd; j++){
    best[j]=nest[K][j];}
//Starting iterations
float s, u, v, step, stepsize, fnew;
for(int iter=0; iter<N_IterTotal; iter++)
{
//Generate new solutions (but keep the current best)
sigma=pow(gamma(1+beta)*sin(M_PI*beta/2)/(gamma((1+beta)/2)*beta*pow
    (2,(beta-1)/2)),(1/beta));
for(int i=0; i<n; i++)
{for(int j=0; j<nd; j++){
    s=nest[i][j];
    //Levy flights by Mantegna's algorithm
    u=distribution(generator)*sigma;
    v=distribution(generator);
    step=u/pow(abs(v),(1/beta));
    stepsize=0.01*step*(s-best[j]);
    s=s+stepsize*distribution(generator);
    //Apply simple bounds/limits
    new_nest[i][j]=simplebounds(s, Lb[j], Ub[j]);}
    fnew=fobj(new_nest[i], nd);
    if (fnew<=fitness[i]){fitness[i]=fnew;
        for(int j=0; j<nd; j++){
            nest[i][j]=new_nest[i][j];}}}
//Get the best nest
K=min_element(fitness, fitness+nd)-fitness;
fmin=fitness[K];
for(int j=0; j<nd; j++){best[j]=nest[K][j];}
//Carry out random permutation
for(int i=0; i<n; i++){r1=rand()%n; r2=rand()%n;
        if (r1!=r2){r=randperm1[r1]; randperm1[r1]=randperm1[r2];
    randperm1[r2]=r;}
        r1=rand()%n; r2=rand()%n;
        if (r1!=r2){r=randperm2[r1]; randperm2[r1]=randperm2[r2];
    randperm2[r2]=r;}
        }
```

```
//Discovery and randomization
for(int i=0; i<n; i++)
{
    for(int j=0; j<nd; j++)
    {
        stepsize=ra()*(nest[randperm1[i]][j]-nest[randperm2[i]][j]);
        if(ra()>pa)
        {
            new_nest[i][j]=nest[i][j]+stepsize;
        }
        else
        {
            new_nest[i][j]=nest[i][j];
        }
        new_nest[i][j]=simplebounds(new_nest[i][j], Lb[j], Ub[j]);
    }
}
//Evaluate the new set of solutions
for(int i=0; i<n; i++)
{
fnew=fobj(new_nest[i], nd);
    if (fnew<=fitness[i])
    {fitness[i]=fnew;
        for(int j=0; j<nd; j++)
        {nest[i][j]=new_nest[i][j];}}}
//Get the best nest
K=min_element(fitness, fitness+nd)-fitness;
fmin=fitness[K];
for(int j=0; j<nd; j++){best[j]=nest[K][j];}
//Find the best objective so far
if (fnew<fmin)
    {fmin=fnew;
        for(int j=0; j<nd; j++)
        {bestnest[j]=best[j];}}}
//Output the best solution
cout<<"Best solution = [ ";
for(int i=0; i<nd; i++) {cout<<bestnest[i]<<" ";}
cout<<"]"<<endl;
cout<<"Best objective = "<<fmin<<endl;
getchar();
return 0;
}
```

Listing 9.6
Cuckoo search in C++.

9.5 A worked example

Let us use the cuckoo search algorithm to find the minimum of

$$f(\boldsymbol{x}) = (x_1 - 1)^2 + (x_2 - 1)^2 + (x_3 - 1)^2 + (x_4 - 1)^2 + (x_5 - 1)^2, \quad (9.14)$$

in the simple ranges of $-5 \leq x_i \leq 5$. This simple 5-variable problem has the global minimum solution $\boldsymbol{x}_{\text{best}} = [1, 1, 1, 1, 1]$.

For the purpose of algorithm demonstration, we only use $n = 3$ cuckoos. Suppose the initial population is randomly initialized and their objective

(fitness) values are as follows:

$$\begin{cases} \boldsymbol{x}_1 = \begin{pmatrix} 5.00 & -1.00 & 5.00 & 2.00 & 3.00 \end{pmatrix}, & f_1 = f(\boldsymbol{x}_1) = 41.00, \\ \boldsymbol{x}_2 = \begin{pmatrix} 3.00 & 2.00 & 4.00 & 5.00 & -5.00 \end{pmatrix}, & f_2 = f(\boldsymbol{x}_2) = 66.00, \\ \boldsymbol{x}_3 = \begin{pmatrix} 3.00 & -4.00 & 2.00 & 0.00 & -2.00 \end{pmatrix}, & f_3 = f(\boldsymbol{x}_5) = 40.00, \end{cases} \tag{9.15}$$

where we have only used two decimal places for simplicity. Obviously, the actual random numbers generated for the initial population can be any numbers in the simple bounds, but here we have used the numbers that are rounded up as the initialization; this is purely for simplicity and clarity.

Obviously, the best solution with the lowest (or best) value of the objective function is $\boldsymbol{x}_3$ with $f_3 = 40.00$ for this population at $t = 0$. Thus, we have

$$\boldsymbol{x}_{\rm best} = [3.00,\ -4.00,\ 2.00,\ 0.00,\ -2.00]. \tag{9.16}$$

At the first iteration, let us suggest that we randomly pick solution $\boldsymbol{x}_2$ for modifying it by Lévy flights. If we generate two random number vectors u and v, then suppose we get

$$\text{stepsize} = \Delta\boldsymbol{x} = [0.50,\ -0.90,\ -1.40,\ -0.70,\ 2.30], \tag{9.17}$$

then the new modified solution will be

$$\boldsymbol{x}_2^1 = \boldsymbol{x}_2^0 + \Delta\boldsymbol{x} = [3.50,\ 1.10,\ 2.60,\ 4.30,\ -2.70], \tag{9.18}$$

which gives $f(\boldsymbol{x}_2^1) = 33.40$. This solution is better than the previous solution $\boldsymbol{x}_2$ and all the previous solutions. Thus, the next call to get the best solution will give the new best solution $f_{\rm min} = 33.40$ with

$$\boldsymbol{x}_{\rm best} = [3.50,\ 1.10,\ 2.60,\ 4.30,\ -2.70]. \tag{9.19}$$

After this Lévy flight modification, we randomly select another solution, say $\boldsymbol{x}_3$ via emptying a nest. We first generate a random number and compare it with p_a to decide if we should update this solution. Suppose the result is to update the solution. Now we have to do this by random permutation of the population, and also generate a random number (say $r = 0.42$). We can get the modification step size as

$$\Delta\boldsymbol{x}_3 = 0.42 \times (\boldsymbol{x}_2^1 - \boldsymbol{x}_1) = [-0.630,\ 0.882,\ -1.008,\ 0.966,\ -2.394], \tag{9.20}$$

which gives the new solution

$$\boldsymbol{x}_3^1 = \boldsymbol{x}_3 + \Delta\boldsymbol{x}_3 = [2.370,\ -3.118,\ 0.992,\ 0.966,\ -4.394]. \tag{9.21}$$

Thus, the new objective value $f(\boldsymbol{x}_3^1) = 47.93$, which is no better than any of the solutions in the current population, so we will not update $f_{\rm min}$. However, we still replace $\boldsymbol{x}_3$ by the new solution for population diversity.

Once the above modifications over the whole population and updates are done, we move on to the next generation $t = 2$ of iterations.

Cuckoo search is very efficient, even with a small population. For example, if we use $n = 5$ (cuckoos), we can get $f_{\text{best}} = 1.34 \times 10^{-17}$ after 1000 iterations for the above example. Even for $D = 15$ and $n = 25$, we can typically get $f_{\text{best}} = 2.50 \times 10^{-12}$ after 1000 iterations.

Obviously, due to the stochastic nature, performance evaluations should be based on the statistical measures of results of multiple runs with proper parametric studies.

9.6 Conclusion

The cuckoo search algorithm can be very efficient and flexible. However, the implementation is slightly more complicated than the implementations of other algorithms due to its use of Lévy flights. The demo codes presented in this chapter provide a simple implementation to focus on the core ideas of the algorithm. It is hoped that this can form a basis for further improvements and modifications so as to solve a wide range of optimization problems in applications.

References

1. X.S. Yang, S. Deb, "Cuckoo search via Lévy flights", in *Proceedings of World Congress on Nature and Biologically Inspired Computing (NaBIC 2009)*, pp. 210-214, IEEE Publications, 2009.
2. X.S. Yang, S. Deb, "Engineering optimisation by cuckoo search", *Int. J. Mathematical Modelling and Numerical Optimisation*, 1(4): 330-343, 2010.
3. X.S. Yang, S. Deb, "Multiobjective cuckoo search for design optimization", *Computers & Operations Research*, 40(6): 1616-1624, 2013.
4. A.H. Gandomi, X.S. Yang, A. H. Alavi, "Cuckoo search algorithm: a metaheuristic approach to solve structural optimization problems", *Engineering with Computers*, 29(1): 17–35, 2013.
5. X.S. Yang, S. Deb, "Cuckoo search: recent advances and applications", *Neural Computing and Applications*, 24(1): 169-174, 2014.
6. I. Pavlyukevich, "Lévy flights, non-local search and simulated annealing", *J. Computational Physics*, 226(2): 1830-1844, 2007.

7. A. Layeb, "A novel quantum-inspired cuckoo search for knapsack problems", *Int. J. Bio-Inspired Computation*, 3(5): 297-305, 2011.
8. S. Walton, O. Hassan, K. Morgan, M.R. Brown, "Modified cuckoo search: a new gradient free optimization algorithm", *Chaos, Solitons & Fractals*, 44(9): 710-718, 2011.
9. A. Ouaarab, B. Ahiod, X.S. Yang, "Random-key cuckoo search for the travelling salesman problem", *Soft Computing*, 19(4): 1099-1106, 2015.
10. X.S. Yang, *Cuckoo Search and Firefly Algorithm: Theory and Applications*, Studies in Computational Intelligence, vol. 516, Springer, Heidelberg, 2013.
11. X.S. Yang, S. Deb, S.K. Mishra, "Multi-species cuckoo search algorithm for global optimization", *Cognitive Computation*, 10(6): 1085-1095, 2018.
12. R. N. Mantegna, "Fast, accurate algorithm for numerical simulation of Lévy stable stochastic process", *Physical Review E*, 49(5): 4677-4683, 1994.