21

Salp Swarm Algorithm: Tutorial

Essam H. Houssein
Faculty of Computers and Information
Minia University, Minya, Egypt

Ibrahim E. Mohamed
Faculty of Computers and Information
South Valley University, Luxor, Egypt

Aboul Ella Hassanien
Faculty of Computers and Information
Cairo University, Cairo, Egypt

CONTENTS

21.1 Introduction

Meta-heuristics are techniques for generating, finding or selecting heuristic partial search algorithms in computer science and optimization. This can provide a sustainable solution to a problem. The optimal solution found depends on a set of generated random parameters [1]. Meta-heuristics can indeed find suitable solutions with less mathematical potential and optimization techniques than simple heuristics or iterative methodologies by having to search for a wide range of possible solutions [2]. Swarm Intelligence (SI) algorithms are a category of meta-heuristic algorithms that mimic the collective behaviour of natural or artificial decentralized self-organized systems such as plants, animals, fish, birds, ants and other elements in our ecosystem that use the intuitive intelligence of the entire swarm, and provide solutions for a set of fundamental problems that could not be solved if the agent does not work collectively [3, 4, 5].

In [6, 7], meta-heuristics can be divided into three main classes: evolution-based, physics-based, and swarm-based methods. Swarm Intelligence (SI) meta-heuristic algorithms mimic the self-organized and collective behaviors of nature's systems. Swarm-inspired algorithms mimic the social behavior of groups of animals, birds, plants and humans. These algorithms include Pity Beetle Algorithm (PBA) [8], Emperor Penguin Optimizer (EPO) [9], Grasshopper optimisation algorithm [10], Artificial Flora (AF) [11], Grey Wolf Optimizer (GWO) [12, 13], Elephant Herding Optimization (EHO) [14] and Whale Optimization Algorithm [15, 16].

Recently, various meta-heuristic optimization algorithms have been developed to solve a wide variety of real life problems. All these algorithms are nature inspired and simulate some principle of biology, physics, ethology or swarm intelligence [17]. Surprisingly, some of them such as Genetic Algorithm (GA) [18], and Particle Swarm Optimization (PSO) [19] are fairly well-known among not only computer scientists but also scientists from different fields.

The work presented in [20] proposed a new meta-heuristic algorithm, salp swarm algorithm (SSA), influenced heavily by deep-sea swarming action salps (as shown in Figure 21.1). SSA seeks to establish a new population-based optimizer by trying to mimic the swarming behaviour of salps in the natural habitat. Several applications have been solved by SSA [21, 22]. In this chapter the modification and the mathematical model of the SSA is presented.

The remainder of this paper is structured as follows: in Section 21.2 the pseudo-code for the standard version of the SSA algorithm is presented and discussed (including a brief introduction to the traditional version of the SSA algorithm), in Sections 21.3 and 21.4 the source codes for the SSA algorithm are shown in the Matlab and C++ programming language respectively, and in Section 21.5 step by step example is presented.

21.2 Salp swarm algorithm (SSA)

21.2.1 Pseudo-code of SSA algorithm

The pseudo-code for the global version of the SSA algorithm is presented in algorithm 19.

Algorithm 19 Pseudo-code of the SSA.

```
Initialize the salp population x_i(i = 1, 2, ..., n) considering ub and lb
while (end condition is not satisfied ) do
    Calculate the fitness of each search agent (salp)
    F = the best search agent
    Update c_1
    for each salp in (X_i) do
        if (i == 1) then
            Update the position of the leading salp
        else
            Update the position of the follower salp.
        end if
    end for
    Amend the salps based on the upper and lower bounds of variables
end while
return F
```

21.2.2 Description of SSA algorithm

The pseudo-code and flowchart of SSA were presented in Algorithm 19 and Fig. 21.2. In the next section, the SSA algorithm will be discussed extensively. Before starting to discuss the SSA algorithm, the objective function $OFun(.)$ (Step 1), and all parameters of the algorithm such as c_1, c_2, c_3, number of iterations, size of population and swarm population should have initialized.

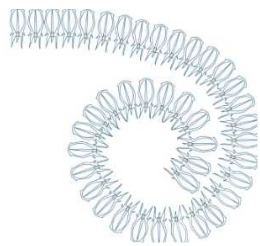

FIGURE 21.1
Demonstration of Salp's series.

Mathematically, The series Salp consists of two groups: leaders and followers. The first salp in the series of salps is called the leader salp, while the residual salps are considered followers. The first swarm (leader) directs the remaining swarm's movements [20]. Let D be the number of variables for a given problem; the positions of the Salps are denoted in a search space of D-dimension. Thus, the Salps X population consists of N swarms with a D

dimension. Hence, a matrix of $N \times D$ could be described as outlined in the equation below:

$$X_i = \begin{bmatrix} x_1^1 & x_2^1 & \dots & x_d^1 \\ x_1^2 & x_2^2 & \dots & x_d^2 \\ \vdots & \vdots & \ddots & \vdots \\ x_1^N & x_2^N & \dots & x_D^N \end{bmatrix} \tag{21.1}$$

A food source F is also thought to be the target of the swarm. The position of the leader is updated by the next equation:

$$X_j^1 = \begin{cases} F_j + c_1((ub_j - lb_j)c_2 + lb_j) & c_3 \geq 0 \\ F_j - c_1((ub_j - lb_j)c_2 + lb_j) & c_3 < 0 \end{cases} \tag{21.2}$$

where X_j^1 and F_j respectively represent leadership positions and food source positions in the j^{th} dimension. The ub_j and lb_j indicate j^{th} dimension in the upper and lower boundaries. c_2 and c_3 are two random numbers. In fact, usually, in addition to defining the step size, they govern the next position in the j^{th} dimension towards the $+\infty$ or $-\infty$. Equation 21.2 shows that the leader only updates its food source position. The coefficient c_1, the most important parameter in SSA, progressively decreases over the next iterations to balance exploration and exploitation, and is defined as follows:

$$c_1 = 2e^{-(\frac{4l}{L})^2} \tag{21.3}$$

Respectively, l and L represent the current iteration and maximum number of iterations. To update the position of the followers, the next equation is used (Newton's motion law):

$$X_j^i = \frac{X_j^i + X_j^{i-1}}{2} \tag{21.4}$$

where $i \geq 2$ and X_j^i is the location of the i^{th} follower at the j^{th} dimension.

21.3 Source code of SSA algorithm in Matlab

In Listing 21.1 the source-code for the objective function for De Jong's function 1 which will optimize by the SSA algorithm is shown. In the function $OFun(X)$, the input parameter is the D-dimensional row vector for the positions of swarm elements. The result of the $OFun(X)$ function is the minimum value. The objective function equation was formulated in equation 21.5

$$OFun(x) = \sum_{i=1}^{n} ix_i^2 \quad -5.12 \leq x_i \leq 5.12 \tag{21.5}$$

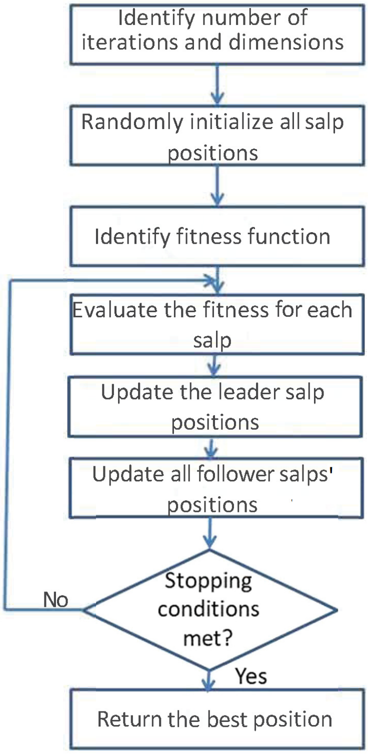

FIGURE 21.2
Flowchart of SSA.

```
%% De Jong's function 1
function [objective]=OFun(x)
[w,h]=size(x);
objective = zeros(w,1);
for i =1: w
    for j =1:h
     objective(i,1)= objective(i,1)+ (i*input(i,j)^2);
    end
end
```

Listing 21.1
Definition of objective function $OFun(x)$ in Matlab.

```
clear all
clc
SearchAgents_no=60; % Number of search agents
Max_iteration=1000; % Maximum number of iterations
% Load details of the selected benchmark function
lb=[-5.12 -5.12 -5.12 -5.12];
ub=[5.12 5.12 5.12 5.12];
dim=4;
[Best_score,Best_pos,SSA_cg_curve]=SSA(SearchAgents_no,Max_iteration,
    lb,ub,dim,OFun);
```

```
display(['The best solution obtained by SSA is ', num2str(Best_pos)]);
display(['The best optimal value of the objective function found by SSA is ', num2str(Best_score)]);
bbest=min(SSA_cg_curve);
mbest=mean(SSA_cg_curve);
wbest=max(SSA_cg_curve);
stdbest=std(SSA_cg_curve);
fprintf('\n best=%f',bbest);
fprintf('\n mean=%f',mbest);
fprintf('\n worst=%f',wbest);
fprintf('\n std. dev.=%f',stdbest);
%This function randomly initializes the position of agents in the search space.
function [Positions]=initialization(SearchAgents_no,dim,ub,lb)
   Boundary_no= size(ub,2); % number of boundaries
% If the boundaries of all variables are equal and user enters a single
% number for both ub and lb
if Boundary_no==1
    Positions=rand(SearchAgents_no,dim).*(ub-lb)+lb;
end
% If each variable has a different lb and ub
if Boundary_no>1
    for i=1:dim
        ub_i=ub(i);
        lb_i=lb(i);
        Positions(:,i)=rand(SearchAgents_no,1).*(ub_i-lb_i)+lb_i;
    end
end
function [FoodFitness,FoodPosition,Convergence_curve]=SSA(N,Max_iter,lb,ub,dim,fobj)
if size(ub,1)==1
    ub=ones(dim,1)*ub;
    lb=ones(dim,1)*lb;
end
Convergence_curve = zeros(1,Max_iter);
%Initialize the positions of salps
SalpPositions=initialization(N,dim,ub,lb);
FoodPosition=zeros(1,dim);
FoodFitness=inf;
%calculate the fitness of initial salps
for i=1:size(SalpPositions,1)
    SalpFitness(1,i)=OFun(SalpPositions(i,:));
end
[sorted_salps_fitness,sorted_indexes]=sort(SalpFitness);
for newindex=1:N
    Sorted_salps(newindex,:)=SalpPositions(sorted_indexes(newindex),:);
end
FoodPosition=Sorted_salps(1,:);
FoodFitness=sorted_salps_fitness(1);
%Main loop
l=2; % start from the second iteration since the first iteration was dedicated to calculating the fitness of salps
while l<Max_iter+1
    c1 = 2*exp(-(4*l/Max_iter)^2);
    for i=1:size(SalpPositions,1)
        SalpPositions= SalpPositions';
        if i<=N/2
            for j=1:1:dim
                c2=rand();
                c3=rand();
                if c3<0.5
                    SalpPositions(j,i)=FoodPosition(j)+c1*((ub(j)-lb(j))*c2+lb(j));
                else
```

```
                    SalpPositions(j,i)=FoodPosition(j)-c1*((ub(j)-lb(j))*c2+lb(j));
                end
            end
        elseif i>N/2 && i<N+1
            point1=SalpPositions(:,i-1);
            point2=SalpPositions(:,i);
            SalpPositions(:,i)=(point2+point1)/2;
        end
        SalpPositions= SalpPositions';
    end
    for i=1:size(SalpPositions,1)
        Tp=SalpPositions(i,:)>ub';Tm=SalpPositions(i,:)<lb';SalpPositions(i,:)=(SalpPositions(i,:).*(~(Tp+Tm)))+ub'.*Tp+lb'.*Tm;
        SalpFitness(1,i)=OFun(SalpPositions(i,:));
        if SalpFitness(1,i)<FoodFitness
            FoodPosition=SalpPositions(i,:);
            FoodFitness=SalpFitness(1,i);
        end
    end
    Convergence_curve(l)=FoodFitness;
    l = l + 1;
end
```

Listing 21.2
Source-code of the SSA in Matlab.

21.4 Source-code of SSA algorithm in C++

```
#include <iostream>
#include <math.h>
#include<ctime>
using namespace std;
float OFun(float x[], int size)
{
  float sum = 0;
  for (int i = 1; i <= size; ++i)
  {
    sum = sum + i * pow(x[i], 2);
  }
  return sum;
}
//initialization population randomly between upper and lower boundaries
float ** initialization(int SearchAgents_no, int dim, float ub[], float lb[])
{
  int Boundary_no;
  float ** Positions = new float *[SearchAgents_no];
  for (int i = 0; i < SearchAgents_no; ++i)
    Positions[i] = new float[dim];
  for (int i = 0; i < SearchAgents_no; i++)
  {
    for (int j = 0; j < dim; j++)
    {
      Positions[i][j] = rand() / (float(ub[j] - lb[j])) + lb[j];
    }
  }
```

```
  return Positions;
}
void SSA(int N, int Max_iter, float lb[], float ub[], int dim)
{
  //initialization of all used data structures in SS Algorithm
  float * Convergence_curve = new float[Max_iter]; // equal float Convergence_curve[N] where N is constant
  float * FoodPosition = new float[dim];
  float FoodFitness = 0;
  float * SalpFitness = new float[N];
  // initialization Salp Positions randomly
  float ** SalpPositions = initialization(N, dim, ub, lb);// equal float SalpPositions[M][N] where M, N are constant
                    //calculate fitness values and select best position as first iteration;
  for (int i = 0; i <N; i++)
  {
    SalpFitness[i] = OFun(SalpPositions[i], dim);
    if (SalpFitness[i] < FoodFitness || i == 0)
    {
      FoodFitness = SalpFitness[i];
      FoodPosition = SalpPositions[i];
    }
  }
  //calculate Convergence_curve for first iteration
  Convergence_curve[0] = FoodFitness;
  int l = 1;
  float c1, c2, c3;
  // Main loop
  //start from the second iteration since the first iteration was dedicated to calculating the fitness of salps
  while (l < Max_iter + 1)
  {
    c1 = 2 * exp(-pow((4 * l / Max_iter), 2));
    for (int i = 0; i < N; i++)
    {
      for (int j = 0; j < dim; j++)
      {
        //if we consider that we have N/2 leaders in the chain
        if (i <= N / 2)
        {
          srand(time(0));
          c2 = (float)(rand() % 10000) / 10000;//generate number between [0,1)
          c3 = (float)(rand() % 10000) / 10000;
          cout << c3;
          if (c3 < 0.5)
            SalpPositions[i][j] = FoodPosition[j] + c1 * ((ub[j] - lb[j])*c2 + lb[j]);
          else
            SalpPositions[i][j] = FoodPosition[j] - c1 * ((ub[j] - lb[j])*c2 + lb[j]);
        }
        else if (i > N / 2 && i < N + 1)
        {
          SalpPositions[i][j] = (SalpPositions[i][j] + SalpPositions[i - 1][j]) / 2;
        }
      }
      //evaluate current agent (swarm) and compare with best fitness value
    }
    for(int k=0; k < N;k++)
    {
      for(int cc=0;cc < dim;cc++)
      {
        if (SalpPositions[k][cc] < lb[cc])
```

```
          SalpPositions[k][cc] = lb[cc];
        if (SalpPositions[k][cc] > ub[cc])
          SalpPositions[k][cc] = ub[jcc];
      }
      SalpFitness[k] = OFun(SalpPositions[k],dim);
      if (SalpFitness[k] < FoodFitness)
      {
        FoodPosition = SalpPositions[k];
        FoodFitness = SalpFitness[k];
      }
    }
    Convergence_curve[l] = FoodFitness;
    l = l + 1;
  }
  cout << "The best solution obtained by SSA is ";
  for (int i = 0; i < dim; i++)
  {
    cout << " " << FoodPosition[i];
  }
  cout << endl << "The best optimal value of the objective function
    found by SSA is " << FoodFitness << endl;
}
int main()
{
  int SearchAgents_no = 60;//Number of search agents
  int Max_iteration = 1000;//Maximum number of iterations
              //Load details of the selected objective function
  float lb[] = { -5.12, -5.12, -5.12, -5.12 };
  float ub[] = { 5.12, 5.12, 5.12, 5.12 };
  int dim = 4;
  SSA(SearchAgents_no, Max_iteration, lb, ub, dim);
  return 0;
}
```

Listing 21.3
Source-code of SSA algorithm in C++.

21.5 Step-by-step numerical example of SSA algorithm

To demonstrate the details of the SSA algorithm, an objective function in equation 21.5 is considered. With five search agents, Let x_j^1 is the position of the leader salp and (x_j^2, x_j^3, x_j^4 and x_j^5) be positions of follower salps.

The initial positions are randomly generated within the boundaries of the design parameters and the value of objective function are shown in Table 21.1. From the first iteration in Table 21.1 it can be seen that the position of the 2^{nd} agent has the minimum value for the objective function. So it is identified as the food source $\{4.1553, -2.2682, 4.8189, -0.8012\}$.

Now for the second iteration, the value of c1(0.9076) is calcuated using equation 21.3 and the values of c2 and c3 are selected randomly. For the first agent(x_j^1), the new values of x_1^1, x_2^1, x_3^1 and x_4^1 are updated according to the equation 21.2; then the new positions of leader salp are placed on Table 21.2, and the calculation is done as shown below:

X_j^1 positions $= \left\{X_1^1,\ X_2^1,\ X_3^1,\ X_4^1\right\}$

TABLE 21.1
Initial Positions.

NO	x_1	x_2	x_3	x_4	$F(X)$	*status*
1	3.2228	-4.1212	-3.5060	-3.6671	5.311026e+01	
2	4.1553	-2.2682	4.8189	-0.8012	4.627460e+01	food source
3	-3.8197	0.4801	4.6814	4.2571	5.485881e+01	
4	4.2330	4.6849	-0.1498	2.9922	4.884174e+01	
5	1.3554	4.7605	3.0749	4.7052	5.609273e+01	

To calculate X_1^1 position let $c1 = 0.90758$, $c2 = 0.6557$ and $c3 = 0.0357$.

$X_1^1 = 4.1553 + 0.90758 * ((5.12 - (-5.12)) * 0.65574 + (-5.12))) = 5.6027$ where $c3 \leq 0.5$

To calculate X_2^1 position let $c1 = 0.90758$, $c2 = 0.8491$ and $c_3 = 0.9340$.

$X_2^1 = (-2.2682) - 0.90758 * ((5.12 - (-5.12)) * 0.84913 + (-5.12))) = -5.5128$ where $c3 \geq 0.5$

To calculate X_3^1 position let $c1 = 0.9076$, $c2 = 0.6787$ and $c_3 = 0.7577$.

$X_3^1 = 4.8189 - 0.90758 * ((5.12 - (-5.12)) * 0.67874 + (-5.12))) = 3.1578$ where $c3 \geq 0.5$

To calculate X_1^1 position let $c1 = 0.90758$, $c2 = 0.7431$ and $c_3 = 0.3922$.

$X_1^4 = (-0.80116) + 0.90758 * ((5.12 - (-5.12)) * 0.74313 + (-5.12))) = 1.4584$ where $c3 \leq 0.5$.

For the second agent(follower : x_j^2), the new values of $\{X_1^2,\ X_2^2,\ X_3^2,\ X_4^2\})$ are updated according to equation 21.4 and placed on Table 21.2 as shown below:

$x_j^2 = (x_j^2\ + x_j^1)/2$

$x_1^2 = (x_1^2 + x_1^1)/2 = (5.6027 + 4.1553)/2 = 4.8790$

$x_2^2 = (x_2^2 + x_2^1)/2 = (-5.5128 + -2.2682)/2 = -3.8905$

$x_3^2 = (x_3^2 + x_3^1)/2 = (3.1578 + 4.8189)/2 = 3.9883$

$x_4^2 = (x_4^2 + x_4^1)/2 = (1.4584 + -0.8012)/2 = 0.3286$

TABLE 21.2
After iteration 2.

NO	x_1	x_2	x_3	x_4	$F(X)$	*status*
1	5.1200	-5.1200	3.1578	1.4584	6.452732e+01	
2	4.8790	-3.8905	3.9883	0.3286	5.495549e+01	
3	0.5297	-1.7052	4.3349	2.2929	2.723661e+01	
4	2.3813	1.4898	2.0926	2.6425	1.925206e+01	food source
5	1.8683	3.1251	2.5837	3.6739	3.343010e+01	

Similarly, the new values of $x_j^3 = \{x_1^3, x_2^3, x_3^3, x_4^3\}$ based on values on Table 21.1 and Table 21.2

$$x_1^3 = (x_1^3 + x_1^2)/2 = (4.8790 + (-3.8197))/2 = 0.5297$$

$$x_2^3 = (x_2^3 + x_2^2)/2 = (-3.8905 + 0.4801)/2 = -1.7052$$

$$x_3^3 = (x_3^3 + x_3^2)/2 = (3.9883 + 4.6814)/2 = 4.3349$$

$$x_4^3 = (x_4^3 + x_4^2)/2 = (0.3286 + 4.2571)/2 = 2.2929$$

Repeat the same steps until each search agent's position is updated. Before the end of the current iteration, the new values for the position of salps should not be beyond the boundary of design variables for the welded problem. If there is a case where positions of salps exceed the limits of design variables, then the new values must be updated to the boundary of the problem.

$$\text{Before update:} \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \\ 5.6027 & -5.5128 & 3.1578 & 1.4584 \\ 4.8790 & -3.8905 & 3.9883 & 0.3286 \\ 0.5297 & -1.7052 & 4.3349 & 2.2929 \\ 2.3813 & 1.4898 & 2.0926 & 2.6425 \\ 1.8683 & 3.1251 & 2.5837 & 3.6739 \end{bmatrix}$$

$$\text{After update:} \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \\ 5.1200 & -5.1200 & 3.1578 & 1.4584 \\ 4.8790 & -3.8905 & 3.9883 & 0.3286 \\ 0.5297 & -1.7052 & 4.3349 & 2.2929 \\ 2.3813 & 1.4898 & 2.0926 & 2.6425 \\ 1.8683 & 3.1251 & 2.5837 & 3.6739 \end{bmatrix}$$

At the end of the current iteration, the fitness values for new positions are computed. Now, the values of f(x) of Table 21.2 in addition to the value of the 2^{nd} agent in Table 21.1 (last food source) are compared and the best value of f(x) is considered the new food source.

The next iterations follow the same steps until the maximum number of iterations is achieved. In the last step in the last iteration, the best solution obtained by SSA is returned as a result of the algorithm operation, and the algorithm is stopped.

21.6 Conclusion

The paper aimed to show the fundamental principles of the SSA algorithm. The algorithm works have been demonstrated. In addition, source codes were introduced in two programming languages, Matlab and C++, that could assist with the implementation of this algorithm by researchers. At last, for a clearer picture of the particular operations that take place in the SSA algorithm, the step-by-step mathematical illustration of the SSA algorithm has been described in detail. It is assumed that this section of the book will make it easier for anyone to complete the development of his modification of the SSA algorithm.

References

1. M.N. Ab Wahab, S. Nefti-Meziani, A. Atyabi. “A comprehensive review of swarm optimization algorithms”. PLoS One, vol. 10(5), pp. 1-36, 2015.
2. S. Russell, P. Norvig. “Artificial Intelligence: A Modern Approach”. Prentice Hall, 1995.
3. V. Pandiri, A. Singh. “Swarm intelligence approaches for multidepot salesmen problems with load balancing”. Applied Intelligence, vol. 44(4), pp. 849-861, 2016.
4. A.A. Ewees, M.A. Elaziz, E.H. Houssein. “Improved grasshopper optimization algorithm using opposition-based learning”. Expert Systems with Applications, vol. 112, pp. 156-172, 2018.
5. A.G. Hussien, E.H. Houssein, A.E. Hassanien. “A binary whale optimization algorithm with hyperbolic tangent fitness function for feature selection” in *Proc. of Eighth International Conference on Intelligent Computing and Information Systems (ICICIS)*, pp. 166-172, 2017.
6. S. Mirjalili, A. Lewis. “The whale optimization algorithm”. Advances in Engineering Software, vol. 95, pp. 51-67, 2016.

7. A.G. Hussien, A.E. Hassanien, E.H. Houssein, S. Bhattacharyya, M. Amin. “S-shaped binary whale optimization algorithm for feature selection” in Recent Trends in Signal and Image Processing, Advances in Intelligent Systems and Computing, vol. 727, Springer, pp. 79-87, 2019.
8. N.A. Kallioras, N.D. Lagaros, D.N. Avtzis. “Pity beetle algorithm-A new metaheuristic inspired by the behavior of bark beetles”. Advances in Engineering Software, vol. 121, pp. 147-166, 2018.
9. G. Dhiman, V. Kumar. “Emperor penguin optimizer: A bio-inspired algorithm for engineering problems”. Knowledge-Based Systems, vol. 159, pp. 20-50, 2018.
10. A. Tharwat, E.H. Houssein, M.M. Ahmed, A.E. Hassanien, T. Gabel. “MOGOA algorithm for constrained and unconstrained multi-objective optimization problems”. Applied Intelligence, vol. 48(8), pp. 2268-2283, 2018.
11. L. Cheng, X.-H. Wu, Y. Wang. “Artificial Flora (AF) optimization algorithm”. Applied Sciences, vol. 8(3), pp. 329-351, 2018.
12. S. Mirjalili, S.M. Mirjalili, A. Lewis. “Grey wolf optimizer”. Advances in Engineering Software, vol. 69, pp. 46-61, 2014.
13. A. Hamad, E.H. Houssein, A.E. Hassanien, A.A. Fahmy. “A hybrid EEG signals classification approach based on grey wolf optimizer enhanced SVMs for epileptic detection” in *Proc. of International Conference on Advanced Intelligent Systems and Informatics*, pp. 108-117, 2017.
14. A.A. Ismaeel, I.A. Elshaarawy, E.H. Houssein, F.H. Ismail, A.E. Hassanien. “Enhanced Elephant Herding Optimization for global optimization”. IEEE Access, vol. 7, pp. 34738-34752, 2019.
15. M.M. Ahmed, E.H. Houssein, A.E. Hassanien, A. Taha, E. Hassanien. “Maximizing lifetime of large-scale wireless sensor networks using multi-objective whale optimization algorithm”. Telecommunication Systems, vol. 72, pp. 243-259, 2019.
16. E.H. Houssein, A. Hamad, A.E. Hassanien, A.A. Fahmy. “Epileptic detection based on whale optimization enhanced support vector machine”. Journal of Information and Optimization Sciences, vol. 40(3), pp. 699-723, 2019.
17. A. LaTorre, S. Muelas, Santiago, J.-M. Peña. “A comprehensive comparison of large scale global optimizers”. Information Sciences, vol. 316, pp. 517-549, 2015.
18. E. Bonabeau, G. Theraulaz, M. Dorigo. “Swarm Intelligence: From Natural to Artificial Systems”. Oxford University Press, 1999.

19. E. Russell, J. Kennedy. “A new optimizer using particle swarm theory” in *Proc. of Sixth International Symposium on Micro Machine and Human Science*, pp. 39-43, 1995.
20. S. Mirjalili, A.H. Gandomi, S.Z. Mirjalili, S. Saremi, H. Faris, S.M. Mirjalili. “Salp Swarm Algorithm: A bio-inspired optimizer for engineering design problems”. Advances in Engineering Software, vol. 114, pp. 163-191, 2017.
21. H.M. Kanoosh, E.H. Houssein, M.M. Selim. “Salp swarm algorithm for node localization in wireless sensor networks”. Journal of Computer Networks and Communications, vol. 2019, Article ID 1028723, 12 pages, 2019.
22. A.G. Hussien, A.E. Hassanien, E.H. Houssein. “Swarming behaviour of salps algorithm for predicting chemical compound activities” in *Proc. of Eighth International Conference on Intelligent Computing and Information Systems (ICICIS)*, pp. 315-320, 2017.