Comparative Performance Analysis of Bat Algorithm and Bacterial Foraging Optimization Algorithm using Standard Benchmark Functions

Yazan A. Alsariera, Hammoudeh S. Alamri, Abdullah M. Nasser, Mazlina A. Majid, and Kamal Z. Zamli

Faculty of Computer Systems and Software Engineering,
Universiti Malaysia Pahang, UMP
26300 Kuantan, Pahang, Malaysia
kamalz@ump.edu.my

Abstract—**Optimization problem relates to finding the best solution from all feasible solutions. Over the last 30 years, many meta-heuristic algorithms have been developed in the literature including that of Simulated Annealing (SA), Genetic Algorithm (GA), Ant Colony Optimization (ACO), Particle Swarm Optimization (PSO), Harmony Search Algorithm (HS) to name a few. In order to help engineers make a sound decision on the selection amongst the best meta-heuristic algorithms for the problem at hand, there is a need to assess the performance of each algorithm against common case studies. Owing to the fact that they are new and much of their relative performance are still unknown (as compared to other established meta-heuristic algorithms), Bacterial Foraging Optimization Algorithm (BFO) and Bat Algorithm (BA) have been adopted for comparison using the 12 selected benchmark functions. In order to ensure fair comparison, both BFO and BA are implemented using the same data structure and the same language and running in the same platform (i.e. Microsoft Visual C# with .Net Framework 4.5). We found that BFO gives more accurate solution as compared to BA (with the same number of iterations). However, BA exhibits faster convergence rate.**

Keywords— metaheuristc optimization algorithms; meta-heuristics algorithm; bat algorithm; bacterial foraging optimization algorithm;

I. Introduction

Solving optimization problem using population based meta-heuristics algorithm has been investigated for over a few decades. As the optimization problems become more and more complex (e.g. with multi-objectives requirements), existing meta-heuristics algorithms appear not scaling well as there is no one size fit all solutions to all problems. Here, identification of new meta-heuristics algorithm is most welcome [1].

In order to help engineers make a sound decision on the selection amongst the best meta-heuristic algorithms for the problem at hand, there is a need to assess the performance of each algorithm against common case studies. Owing to the fact that they are new and much of their performance is still unknown (as compared to other established meta-heuristic algorithms), Bacterial Foraging Optimization Algorithm (BFO) [2] and Bat Algorithm (BA) [3] have been adopted for benchmark comparison using the 12 selected benchmark functions. In order to ensure fair comparison, both BFO and BA are implemented using the same data structure and the same language and running in the same platform (i.e. Microsoft Visual C# with .Net Framework 4.5). Here, our aim for making this comparison is to select the suitable meta-heuristic algorithm for solving the t-way test data generation problem.

The work described in this paper is funded by the eScience Fund: "Constraints t-way Testing Strategy with Modified Condition/Decision Coverage" from MOSTI, Malaysia.

We have developed two GUI application implementing BA and BFO algorithm called BAapp and BFOapp respectively. Here, the developed applications are based on the original pseudo code by Yang 2010 [3] and Passino 2002 [2] without any modifications.

The remainder of the paper is organized as follows. Section 2 briefly reviews BA and the experiment implementation of it application BAapp. Section 3 briefly reviews BFO and the experiment implementation of it application BFOapp. Section 4 elaborates on the results of the experiments. Section 5 provides concluding remarks and our plan for further work.

II. Bat Algorithm Implementation

A. Bat Algorithm (BA)

Bat algorithm (BA) is ta new population based meta-heuristic algorithm founded on the hunting behavior of Microbats. The algorithm has been built on the assumption that the bat is able to find its prey in complete darkness [4]. The bat position represents a possible solution of the problem. The best position of a bat to its prey indicates the quality of the solution. Here, obstacles are avoided using echolocation. In such a case, different frequencies are returned [4, 5]. Generally, the BA has three main assumptions [3, 6]:

- *All bats are using echolocation to intelligently calculate distance. They know the difference between food/prey and the surrounding environment background in magical way.*
- *Bats are flying randomly using velocity* $\mathbf{v}_i$ *at position* $\mathbf{x}_i$. *They automatically adjust emitted pulses and adjust the rate of pulse emission r [0, 1], of their echolocation frequency.*
- *Although the loudness could be different in several ways, Here, it is assumed that the loudness change from a large (positive)* $\boldsymbol{A_0}$ *to a minimum value* $\boldsymbol{A_{min}}$.

The pseudo code of the Bat algorithm can be seen in Fig. 1.

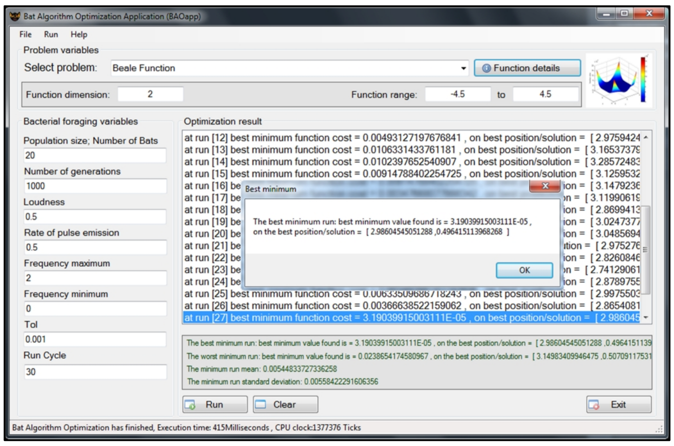
The pseudo code of the Bat algorithm

```
[1]. Objective function $f(x_i), x_i = (x_{i1}, \dots, x_{iD})^T$
[2]. Initialize the bat population $x_i$ and velocities $v_i$ for
     $i = (1, 2, \dots, Number\ of\ Bats)$
[3]. Define pulse frequency $Q_i \in [Q_{min}, Q_{max}]$
[4]. Initialize pulse rates $r_i$ and the loudness $A_i$
[5]. While $(t < T_{max})$ // number of iterations besed on No. of generation and
     No. of bats.
         Generate new solutions by adjusting frequency, and update
         velocities and location / solutions [Eq. 2 to 4]
                  $f_i = f_{min} + (f_{max} + f_{min})\beta,$        (1)
                  $v_i^{t+1} = v_i^t + (x_i^t - x_*) f_i,$        (2)
                  $x_i^{t+1} = x_i^t + v_i^{t+1},$        (3)
         if $(rand(0,1) > r_i)$
                  Select the best solution in the current population
                  Generate a local solution around the best solution
         end if
         if $(rand(0,1) < A_i and\ f(x_i) < f(x))$
                  Accept the new solutions
                  Increase $r_i$ and reduce $A_i$
         end if
                  Rank the bats and find the current best
     End while
[6]. Postprocess results and visualization
```

Fig. 1. Pseudo code of the bat algorithm (BA) according to [7].

B. *BAapp Application Implementation*

We have created three classes called *Main.cs*, *Population.cs*, and *Bats.cs* IComparable interface class, for implementing of BTapp.



Problem variables
Select problem: Beale Function
Function dimension: 2 Function range: -4.5 to 4.5

Bacterial foraging variables
Population size; Number of Bats: 20
Number of generations: 1000
Loudness: 0.5
Rate of pulse emission: 0.5
Frequency maximum: 2
Frequency minimum: 0
Tol: 0.001
Run Cycle: 30

Optimization result

```
at run [12] best minimum function cost = 0.00493127197676841 , on best position/solution =  [ 2.9759424
at run [13] best minimum function cost = 0.0106331433761181 , on best position/solution =  [ 3.16537379
at run [14] best minimum function cost = 0.0102397652540907 , on best position/solution =  [ 3.28572483
at run [15] best minimum function cost = 0.00914788402254725 , on best position/solution =  [ 3.1259532
at run [16] be                                                                        ion =  [ 3.1479236
at run [17] be                                                                         n =  [ 3.11990619
at run [18] be                                                                        ion =  [ 2.8699413
at run [19] be                                                                        ion =  [ 3.0247377
at run [20] be                                                                        ion =  [ 3.0485694
at run [21] be                                                                      ution =  [ 2.975276
at run [22] be                                                                        ion =  [ 2.8260846
at run [23] be                                                                         n =  [ 2.74129061
at run [24] be                                                                        ion =  [ 2.8789755
at run [25] best minimum function cost = 0.00633509686718243 , on best position/solution =  [ 2.9975503
at run [26] best minimum function cost = 0.00366638522159062 , on best position/solution =  [ 2.8654081
at run [27] best minimum function cost = 3.19039915003111E-05 , on best position/solution =  [ 2.986045
```

```
The best minimum run: best minimum value found is = 3.19039915003111E-05 ,
on the best position/solution =  [ 2.98604545051288 ,0.496415113968268  ]
```

```
The best minimum run: best minimum value found is = 3.19039915003111E-05 , on the best position/solution =  [ 2.98604545051288 ,0.4964151139
The worst minimum run: best minimum value found is = 0.0238654174580967 , on the best position/solution =  [ 3.14983409946475 ,0.50709117531
The minimum run mean: 0.00544833727336258
The minimum run standard deviation: 0.00558422291606356
```

```
Bat Algorithm Optimization has finished, Execution time: 415Milliseconds , CPU clock:1377376 Ticks
```

Fig. 2. Bat algorithm application screenshots

The *Main* class consists of several methods and functions as shown in Fig. 3. The attributes in the main class are defined according to the representation in Yang [3]. *initialPopulation* operation initializes the population of bats via the object of type *bats* array *(i.e.* by calling the constructor in the *Population* class). The *optimize* method controls all the execution process for the bat algorithm. Upon initialization, the Bat algorithm actual iterates from step5 to step 5 (refer to pseudo code in Fig. 1). *getPosition* is the method used to return the index of the best position of the best fitness. As the name suggests, the method *getFitness* is in charge of calculating the fitness functions. The other methods described in the main class have auxiliary purposes (i.e. to display the result and make them readable).

The main aim for the *Population* class is to initialize the attributes for the bat algorithm (i.e. through an interface object). Here, the constructor initializes a sub-object form the interface class *Bats* for all defined bats. Each sub-object in the main object *bats* is from the type *IComparable* where each object has a four attributes; *positions* array, *velocity* value, *fitness* value and *frequency* value respectively. Each row represents an interface object. This interface reduces the number of the variables, which are needed in classic implementation.

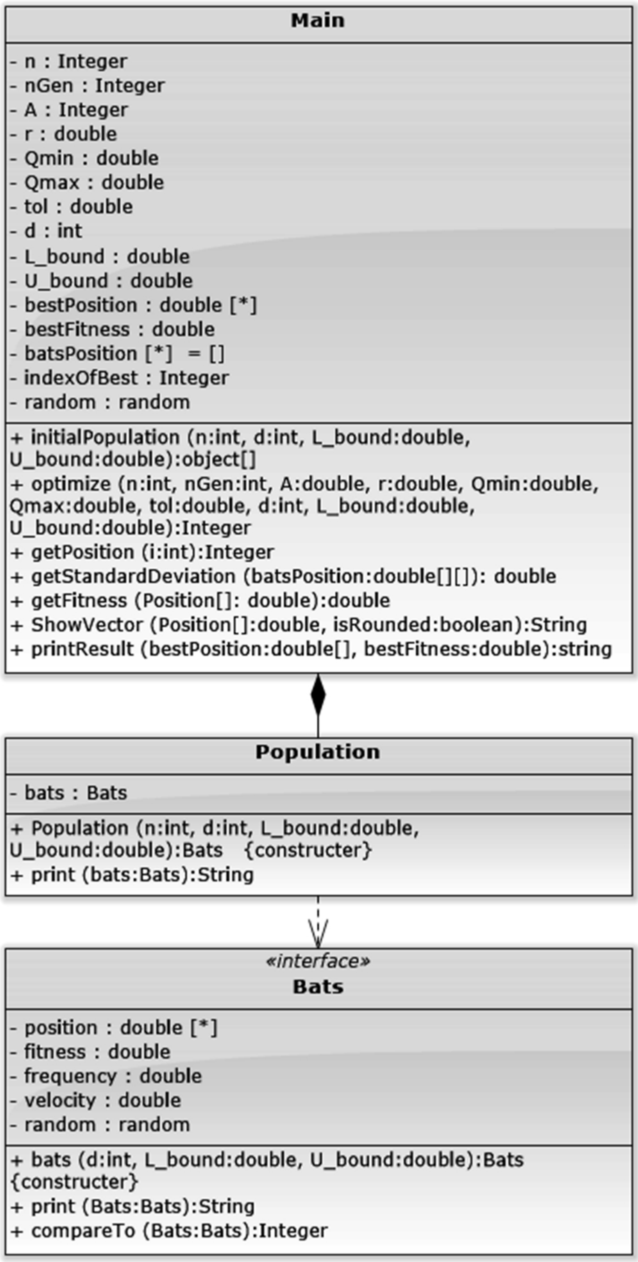

Main
- n : Integer
- nGen : Integer
- A : Integer
- r : double
- Qmin : double
- Qmax : double
- tol : double
- d : int
- L_bound : double
- U_bound : double
- bestPosition : double [*]
- bestFitness : double
- batsPosition [*] = []
- indexOfBest : Integer
- random : random
+ initialPopulation (n:int, d:int, L_bound:double, U_bound:double):object[]
+ optimize (n:int, nGen:int, A:double, r:double, Qmin:double, Qmax:double, tol:double, d:int, L_bound:double, U_bound:double):Integer
+ getPosition (i:int):Integer
+ getStandardDeviation (batsPosition:double[][]): double
+ getFitness (Position[]: double):double
+ ShowVector (Position[]:double, isRounded:boolean):String
+ printResult (bestPosition:double[], bestFitness:double):string
Population
- bats : Bats
+ Population (n:int, d:int, L_bound:double, U_bound:double):Bats {constructer}
+ print (bats:Bats):String
«interface»
Bats
- position : double [*]
- fitness : double
- frequency : double
- velocity : double
- random : random
+ bats (d:int, L_bound:double, U_bound:double):Bats {constructer}
+ print (Bats:Bats):String
+ compareTo (Bats:Bats):Integer

Fig. 3. Bat algorithm application (BAapp) class diegram

The *fitness* value for each *bat* object is calculated based on the problem domain selected in the execution. The *frequency* value and the *velocity* value are calculated using the equation

[1-3]. The best fitness from each iterative generation is stored in *bestPosition* in Array in the main class. In each iteration, the BA finds new position for each bat and compares the new positions fitness with the best position fitness. If there are improvements, the position for the new improved fitness selected as best position. Upon completion, the best solution (i.e. minimum or maximum) for the problem is found and displayed on the screen (see Fig. 2).

III. Bacterial Foraging Optimization Algorithm Implementation

A. Bacterial foraging optimization algorithm (BFO)

Bacterial foraging optimization algorithm (BFO) is inspired by the intelligence social behaviors of E-Coli bacteria. Specifically, it is based on bacteria swarming and foraging behaviors. There are five main steps: chemotaxis, swarming, reproduction, elimination and dispersal [8, 9]. These steps can be seen in the pseudo code of BFO at Fig. 4.

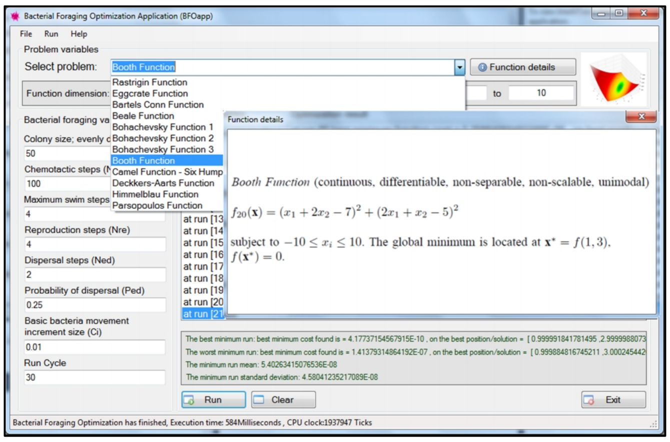
The pseudo code of Bacterial foraging optimization algorithm

```
[1] Initialize parameters $n, S, N_c, N_s, N_{re}, N_{ed}, P_{ed}, C_{(i)}$ $(i = 1,2,\dots,S), \theta i$.
[2] Elimination-dispersal loop: $l = l + 1$.
[3] Reproduction loop: $k = k + 1$.
[4] Chemotaxis loop: $j = j + 1$.
    [a]. For $i = 1, 2, \dots, S$ take a chemotactic step for bacterium i as follows.
    [b]. Compute fitness function, $J\,(i, j, k, l)$.
         Let, $J(i,j,k,l) = J(i,j,k,l) + J_{cc}\left(\theta^i(j,k,l), P(j,k,l)\right)$
         (i.e. add on the cell-to cell attractant–repellant profile to simulate the swarming behavior)
         Where, $J_{cc}$ is defined in (2).
             $x_{i,j}(0) = x_{min,j} + rand_{\ j}(0,1).\left(x_{max,j} - x_{min,j}\right)$      (4)
    [c]. Let $J_{last} = J(i,j,k,l)$ to save this value since we may find better value via a run.
    [d]. Tumble: Generate a random vector $\Delta(i) \in R^n$ with each element $\Delta_m(i), m = 1, 2, \dots, p$ a random number on [-1, 1].
    [e]. Move: Left
             $\theta(i+1,j,k) = \theta(i,j,k) + C(i)\frac{\Delta(i)}{\sqrt{\Delta^T(i)\,\Delta(i)}}$      (5)
         This results in a step of size C(i) in the direction of the tumble for bacterium i.
    [f]. Compute J(i, j + 1, k, l) with Let, J (i, j, k, l) = J (i, j, k, l) + $J_{cc}$ ($\theta^i$( j, k, l), P( j, k, l))
         Swim:
         Let m = 0 (counter for swim length).
         While m < $N_s$ (if has not climbed down too long)
                 Let $m = m + 1$.
                 If $J(i, j+1, k, l) < J_{last}$,
                 let $J_{last} = J(i, j+1, k, l)$.
                 And move left as it in equation (6).
                 And use this $\theta\ (i\ +1, j, k)$ to compute
                 the new $J\ (i, j\ +1, k, l)$ as we did in [f].
                 Else, let $m = N_s$.
         End While.
    [g]. Go to next bacterium$(i + 1)$. if $i \neq S$, do Compute fitness function as in [b] for $J\ (i, j, k, l)$. Process the next bacterium.
[5] If $j < N_c$, go to Step [3] Reproduction loop. In this case, continue chemotaxis since the life of the bacteria is not over.
[6] Reproduction:
        [a]. For the given k and l, and for each i = 1, 2, …, S,
                 $J^i_{health} = \sum_{j=1}^{N_c+1} J(i,j,k,l)$      (6)
             be the health of the bacterium i (a measure of how many nutrients it got over its lifetime and how successful it was at avoiding noxious substances). Sort bacteria and chemotactic parameters $C(i)$ in order of ascending cost $J_{health}$ (higher cost means lower health).
        [b]. The $S_r$ bacteria with the highest $J_{health}$ values die and the other $S_r$ bacteria with the best values split and the copies that are made are placed at the same location as their parent.
[7] If $k < N_{re}$ go to Step [3] Elimination-dispersal loop. In this case the number of specified reproduction steps is not reached and start the next generation in the chemotactic loop.
[8] Elimination-dispersal: For $i = 1,2\dots,S$, with probability $P_{ed}$, eliminate and disperse each bacterium, and this result in keeping the number of bacteria in the population constant. To do this, if a bacterium is eliminated, simply disperse one to a random location on the optimization domain. If $< N_{ed}$, then go to step [2]; otherwise end.
```

Fig. 4. Pseudo code of the Bacterial foraging optimization algorithm according to [10].

B. BFOapp Application Implementation

We have created three classes called *Main.cs, Colony.cs*, and *Bacterium.cs* IComparable interface class, for implementing of BFOapp.



Colony size; evenly d: 50
Chemotactic steps (N: 100
Maximum swim steps: 4
Reproduction steps (Nre): 4
Dispersal steps (Ned): 2
Probability of dispersal (Ped): 0.25
Basic bacteria movement increment size (Ci): 0.01
Run Cycle: 30

Function details

Booth Function (continuous, differentiable, non-separable, non-scalable, unimodal)

$f_{20}(\mathbf{x}) = (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2$

subject to $-10 \le x_i \le 10$. The global minimum is located at $\mathbf{x}^* = f(1,3)$, $f(\mathbf{x}^*) = 0$.

```
The best minimum run: best minimum cost found is = 4.17737154567915E-10 , on the best position/solution =  [ 0.999991841781495 ,2.9999988073
The worst minimum run: best minimum cost found is = 1.41379314864192E-07 , on the best position/solution =  [ 0.999884816745211 ,3.000245442
The minimum run mean: 5.40263415076536E-08
The minimum run standard deviation: 4.58041235217089E-08
```

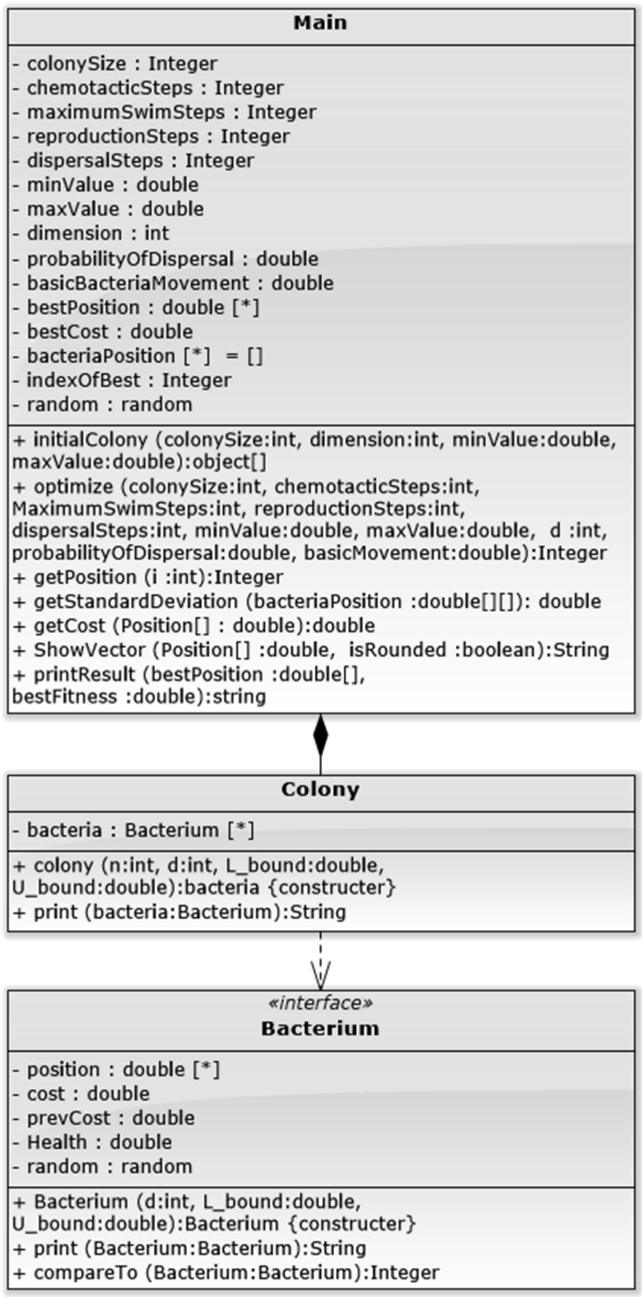
Bacterial Foraging Optimization has finished, Execution time: 584Milliseconds , CPU clock:1937947 Ticks

Fig. 5. Bacterial foraging optimization algorithm application screenshots

The Main class consists of several methods and functions (see Fig. 6). The attribute in the main class represented according to the representation in Passino [2]. *initialColony* operation initializes the colony of bacteria as object of type *Bacterium* array by calling the constructor in the colony class. The *optimize* controls all the execution process for the bacterial foraging optimization (from step [2] to step [8] in Fig. 4). The *getPosition* method returns the index of the best position of the best fitness. The method getCost is in charge with the calculation of the fitness function for each *Bacterium* position. The other methods described in the main class have auxiliary purposes (i.e. to display the result and make them readable).

The main aim for the Colony class is to initialize all the attributes for the bacterial foraging optimization algorithm (i.e. through an interface object). Here, the constructor initializes a sub-object form the interface class *Bacterium* for all defined bacteria. Each sub-object in the main bacteria object is of type IComparable. Each object has four attributes; *position* array, *Cost* value, *PreviousCost* value and *Health* value. Each row represents an interface object. This interface reduces the number of the variables, which are needed in classic implementation.

The *cost* value for each *Bacterium* object is calculated based on the problem domain selected in the execution. Here, the first generation *position* of the colony located randomly in the problem range. The *health* value and is calculated using the equation [6].



Main
- colonySize : Integer
- chemotacticSteps : Integer
- maximumSwimSteps : Integer
- reproductionSteps : Integer
- dispersalSteps : Integer
- minValue : double
- maxValue : double
- dimension : int
- probabilityOfDispersal : double
- basicBacteriaMovement : double
- bestPosition : double [*]
- bestCost : double
- bacteriaPosition [*] = []
- indexOfBest : Integer
- random : random
+ initialColony (colonySize:int, dimension:int, minValue:double, maxValue:double):object[]
+ optimize (colonySize:int, chemotacticSteps:int, MaximumSwimSteps:int, reproductionSteps:int, dispersalSteps:int, minValue:double, maxValue:double, d :int, probabilityOfDispersal:double, basicMovement:double):Integer
+ getPosition (i :int):Integer
+ getStandardDeviation (bacteriaPosition :double[][]): double
+ getCost (Position[] : double):double
+ ShowVector (Position[] :double, isRounded :boolean):String
+ printResult (bestPosition :double[], bestFitness :double):string
Colony
- bacteria : Bacterium [*]
+ colony (n:int, d:int, L_bound:double, U_bound:double):bacteria {constructer}
+ print (bacteria:Bacterium):String
«interface»
Bacterium
- position : double [*]
- cost : double
- prevCost : double
- Health : double
- random : random
+ Bacterium (d:int, L_bound:double, U_bound:double):Bacterium {constructer}
+ print (Bacterium:Bacterium):String
+ compareTo (Bacterium:Bacterium):Integer

Fig. 6. Bacterial foraging optimization application (BFOapp) class diegram

The best cost from the first generation is stored in *bestPosition* array in the main class. The main iteration of the BFO starts by looping in the number of elimination of dispersal steps, reproduction steps and chemotactic steps in sequence. In each iteration, BFO finds new position for each *bacterium* and compares the new position's cost with the previous cost. If there are improvements, the position for the new improved fitness selected as best position. Upon completion, the best solution (i.e. minimum or maximum) for the problem is found and displayed on the screen (see Fig. 5).

IV. Test Run and Results

For our test, we run the optimization tests for all the functions from BTapp and BFOapp and recorded the results. Our test is performed on Intel® Core ™ i7-3770 (3.40GHz, 3MB L3, 256KB L2, 32KB L1 cache) with 4GB of RAM on Windows 7 Operating System with Visual Studio 2013. We show the result of time performance (in milliseconds).

The Bat algorithm implemented in BTapp takes the following parameter values. The population size *nBats* = 40 (typically 10 to 40 bats [3] for the population), number of generation = 1000, loudness = 0.5, rate of pulse emission Q = 0.5 with frequency range of [0,2] and tolerance = 0.001. Notice: that we fixed the dimension of all the functions equal to 2. As for the Bacteria Foraging Algorithm (BFO) implemented in BFOapp, we set the colony size S = 50 (typically divisible by 2), chemotactic steps N_c =100, maximum swim steps N_s = 4, reproduction steps N_{re} =4, dispersal steps N_{ed} = 2, probability of dispersal P_{ed} = 0.25, and basic bacteria movement C_i = 0.01.

We have selected 12 standard benchmark functions in [11] and fixed our test with 2 diminutions for all the benchmark functions. Here, the two diminutions of the selected standard benchmark functions have been chosen carefully from the standard benchmark functions survey described in [11]. In this case, we get the best and worst solution for each function. We also calculate the mean and standard deviation for the 30 runs result (for statistical significance).

A. Test Functions

1) The Rastrigin's function:

$$f_1(x) = 10D + \sum_{i}^{D} (x_i^2 - 10\cos(2\pi x_i)) \quad (7)$$

subject to $[-5.12 \le x_i \le 5.12]$. It has global minimum located at $x^* = (0, ..., 0), f(x^*) = 0.$

2) Egg Crate function:

$$f_{2(x)} = x_1^2 + x_2^2 + 25\,(sin^2(x_1) + sin^2(x_2)) \quad (8)$$

subject to $[-5 \le x_i \le 5]$. It has global minimum located at $x^* = (0, 0), f(x^*) = 0.$

3) Bartels Conn function:

$$f_3(x) = |x_1^2 + x_2^2 + x_1x_2| + |sin\,x_1| + |cos\,x_2| \quad (9)$$

subject to $[-500 \le x_i \le 500]$. It has global minimum located at $x^* = (0, 0), f(x^*) = 1.$

4) Beale Function:

$$\begin{aligned} f_4(x) = (1.5 - x_1 + x_1x_2)^2 \\ +(2.25 - x_1 + x_1x_2^2)^2 \\ +(2.625 - x_1 + x_1x_2^3)^2 \end{aligned} \quad (10)$$

subject to $[-4.5 \le x_i \le 4.5]$. It has global minimum located at $x^* = (3, 0.5), f(x^*) = 0.$

5) *Bohachevsky Function 1:*

$$f_5\ x\ =\ x_1^2 + 2x_2^2 - 0.3\cos\ 3\pi x_1 - 0.4\cos\ 4\pi x_2\ +\ 0.7 \qquad (11)$$

subject to $[-100 \leq x_i \leq 100]$. It has global minimum located at *x^* = (0, 0), f(x^*) = 0.*

6) *Bohachevsky Function 2:*

$$f_6\ x\ =\ x_1^2 + 2x_2^2 - 0.3\cos\ 3\pi x_1 * 0.4\cos\ 4\pi x_2\ +\ 0.3 \qquad (12)$$

subject to $[-100 \leq x_i \leq 100]$. It has global minimum located at *x^* = (0, 0), f(x^*) = 0.*

7) *Bohachevsky Function 3:*

$$f_7\ x\ =\ x_1^2 + 2x_2^2 - 0.3\cos\ 3\pi x_1 +\ 4\pi x_2 + 0.3 \qquad (13)$$

subject to $[-100 \leq x_i \leq 100]$. It has global minimum located at *x^* = (0, 0), f(x^*) = 0.*

8) *Booth Function:*

$$f_8\ x\ =\ x_1 + 2x_2 - 7\ ^2 +\ \ 2x_1 + x_2 - 5\ ^2 \qquad (14)$$

subject to $[-10 \leq x_i \leq 10]$. It has global minimum located at *x^* = (1, 3), f(x^*) = 0.*

9) *Camel – Six Hump Function:*

$$f_9\ x\ =\ \ 4 - 2.1x_1^2 + \frac{x_1^2}{3}\ \ x_1^2 + x_1x_2 +\ \ 4x_2^2 - 4\ x_2^2 \qquad (15)$$

subject to $[-5 \leq x_i \leq 5]$. It has global minimum located at x^* *= ({-0.0898, 0.7126}, {0.0898, - 0.07126}), f(x^*) = -1.0316.*

10) *Deckkers-Aarts Function:*

$$f_{10}\ x\ =\ 10^5x_1^2 + x_2^2 -\ \ x_1^2 + x_2^2\ ^2 + 10^{-5}\ x_1^2 + x_2^2\ ^4 \qquad (16)$$

subject to $[-20 \leq x_i \leq 20]$. It has a global minimum located at *x^* = (0, ±15), f(x^*) = -24777.*

11) *Himmelblau Function:*

$$f_{11}\ x\ =\ \ x_1^2 + x_2 - 11\ ^2 +\ \ x_1 + x_2^2 - 7\ ^2 \qquad (17)$$

subject to $[-5 \leq x_i \leq 5]$. It has a global minimum located at *x^* = (3, 2), f(x^*) = 0.*

12) *Parsopoulos Function:*

$$f_{12}\ x\ =\ \sin\ x_1\ ^2\ \cos\ x_2\ ^2 \qquad (18)$$

subject to $[-5 \leq x_i \leq 5]$. It has 12 global minimum located at *$x^* \in$* R², *f(x^*) = 0.*

B. Result for the experiment

TABLE I. RESULT OF BAAPP

Fun.	**Result for Bat Algorithm (BTapp)**				
	Best	***Worst***	***Means***	***St.Dev***	***Time(ms)***
f_1	4.99E-04	2.29649	1.21333	6.71E-01	155
f_2	1.27E-04	1.30036	3.18E-01	3.48E-01	354
f_3	1.29393	260.301	91.7337	74.8549	252
f_4	1.18E-06	8.41E-03	3.74E-03	2.65E-03	335
f_5	8.07E-02	13.0098	5.14899	3.65217	245
f_6	-2.4E-03	17.3774	5.16764	4.47657	250
f_7	1.11E-02	26.1169	7.04784	6.65011	212
f_8	8.57E-06	2.98E-01	9.43E-02	8.28E-02	175
f_9	-1.03156	-9.5E-01	-1.00542	2.17E-02	333
f_{10}	-24775.2	-21498.6	-23801.7	909.526	585
f_{11}	1.05E-05	3.61E-01	7.45E-02	9.02E-02	301
f_{12}	1.00E-05	2.54E-02	6.54E-03	6.40E-03	175

TABLE II. RESULT OF BFOAPP

Fun.	**Result for Bacterial Foraging Optimization Algorithm (BFOapp)**				
	Best	***Worst***	***Means***	***St.Dev***	***Time(ms)***
f_1	1.69E-08	9.95E-01	5.30E-01	5.04E-01	297
f_2	1.98E-09	2.12E-06	4.41E-07	4.31E-07	712
f_3	1.00000	10862.4	3273.97	3465.19	724
f_4	2.68E-10	2.86E-07	1.01E-07	6.59E-08	777
f_5	2.42E-01	314.868	41.7957	85.5574	689
f_6	1.29E-02	314.468	30.3813	79.2000	689
f_7	4.61E-07	212.196	30.9777	54.1435	600
f_8	1.34E-10	3.33E-07	8.14E-08	8.38E-08	436
f_9	-1.03162	-1.03162	-1.03162	7.69E-08	706
f_{10}	-24776.5	-24776.5	-24776.5	1.23E-03	1395
f_{11}	8.59E-09	1.21E-06	2.99922	3.28E-07	652
f_{12}	1.72E-10	4.23E-08	1.27E-08	1.20E-08	506

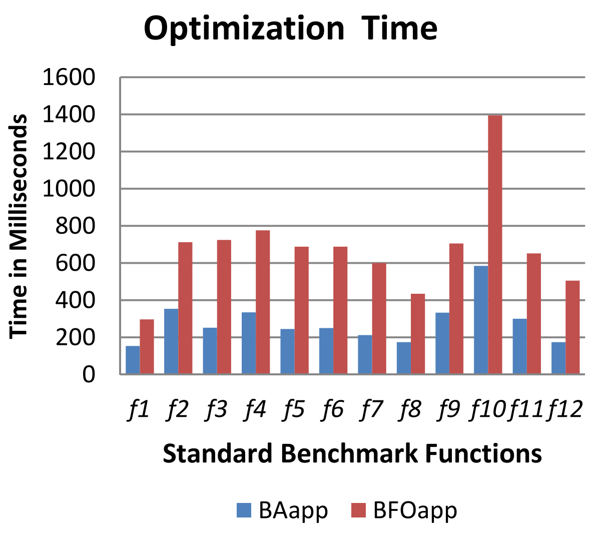


Fig. 7. Time of optimization process for the two algorithms.

V. Discussion and Conclusion

At a glance, the iterations for both BTapp and BFOapp appear different which may nullify the comparison. A counter argument suggests otherwise. Although there are parameter differences, the number of minimum iteration for both algorithms are the same. As for the Bat algorithm has two control loop the other loop is for the local search (see Fig. 1), the minimum iteration for BA = 1000 generation * 40 bats = 40000 iterations. Concerning BFO algorithm has four control loop (see Fig. 4), the minimum iteration for BFO = 100 chemotactic step * 4 reproduction step * 2 dispersal steps * 50 colony size = 40000 iterations.

In our experiments, we found that BFO gives more accurate solution as compared to BA. The result is expected as BFO also performs local search when the results is not converging well resulting in to significant increase in execution time. Expectedly, BA yield faster convergence to solution. (see Table 1 and Table 2).

Our contention is to use an optimal and fast algorithm for our t-way data test generation strategy. For this reason, we have opted for the Bat algorithm for our work [12-15]. As the scope of future work, we are currently modeling and implementing our t-way strategy using the Bat algorithm.

Acknowledgment

We would like to thank the Ministry of Science, Technology and Innovation (MOSTI), Malaysia for funding our work.

References

[1] J. Petrowski and P. Taillard, “Metaheuristics for Hard Optimization,” ed: Springer, 2006.

[2] K. M. Passino, “Biomimicry of Bacterial Foraging for Distributed optimization and control,” Control Systems, IEEE, vol. 22, pp. 52-67, 2002.

[3] X. Yang, “A new Metaheuristic Bat-inspired Algorithm,” in Nature inspired cooperative strategies for optimization (NICSO 2010), ed: Springer, 2010, pp. 65-74.

[4] G. Das, “Bat Algorithm based Softcomputing Approach to Perceive Hairline Bone Fracture in Medical X-ray Images,” International Journal of Computer Science & Engineering Technology (IJCSET), vol. 4, pp. 432-436, 04 April 2013 2013.

[5] A. Gandomi, X. Yang, A. Alavi, and S. Talatahari, “Bat Algorithm for Constrained Optimization Tasks,” Neural Computing and Applications, vol. 22, pp. 1239-1255, 2013.

[6] N. Sureja, “New Inspirations in Nature: a Survey,” IJCAIT, vol. 1, pp. 21-24, 2012.

[7] A. Hashmi, D. Gupta, Y. Upadhyay, and S. Goel, “Swarm Intelligence Based Approach For Data Clustering,” International Journal of Innovative Research & Studies (IJiRS), 2013.

[8] B. Niu, Y. Fan, H. Xiao, and B. Xue, “Bacterial Foraging based Approaches to Portfolio Optimization With Liquidity Risk,” Neurocomputing, vol. 98, pp. 90-100, 12/3/ 2012.

[9] H. Chen, B. Niu, L. Ma, W. Su, and Y. Zhu, “Bacterial Colony Foraging Optimization,” Neurocomputing, vol. 137, pp. 268-284, 8/5/ 2014.

[10] S. Yichuan and C. HanNing, “The Optimization of Cooperative Bacterial Foraging,” in Software Engineering, 2009. WCSE '09. WRI World Congress on, 2009, pp. 519-523.

[11] M. Jamil and X. Yang, “A Literature Survey of Benchmark Functions for Global Optimisation Problems,” International Journal of Mathematical Modelling and Numerical Optimisation, vol. 4, pp. 150-194, 2013.

[12] B. Ahmed and K. Zamli, “A Variable Strength Interaction Test Suites Generation Strategy using Particle Swarm Optimization,” Journal of Systems and Software, vol. 84, pp. 2171-2185, 2011.

[13] B. Ahmed, K. Zamli, and C. Lim, “Application of Particle Swarm Optimization to Uniform and Variable Strength Covering Array construction,” Applied Soft Computing, vol. 12, pp. 1330-1347, 2012.

[14] A. Alsewari and K. Zamli, “Design and Implementation of a Harmony-search-based Variable-Strength t-way Testing Strategy With Constraints Support,” Information and Software Technology, vol. 54, pp. 553-568, 2012.

[15] R. Othman, K. Zamli, and L. Nugroho, “General variable strength t-way strategy supporting flexible interactions,” Maejo International Journal of Science and Technology, vol. 6, 2012.