Artificial Intelligence 791 Assignment 3 Option 3

The Unknown Explored: The Artificial Bee Colony Algorithm

Simeon Boshoff
Division of Computer Science
Stellenbosch University
Stellenbosch, South Africa
22546510@sun.ac.za
25/05/2022

***Abstract*—This research assignment evaluates a selected meta-heuristic that was not covered by the module contents of Artificial Intelligence 791. For this purpose, the Artificial Bee Colony (ABC) algorithm is tested against the known algorithms such as Particle Swarm Optimization (PSO) and Differential Evolution (DE). After careful consideration, execution time is used to evaluate the speed at which the algorithms can solve the selected benchmark functions in 30 dimensions. Other factors such as memory use and complexity of code are also taken into account. The result is that the ABC algorithm wins at nearly every test, and can obtain solutions that are at times up to 5000% better than others.**

***Index Terms*—Unexplored, Particle Swarm Optimization, Differential Evolution, Artificial Bee Colony, Benchmark**

I. Introduction

Throughout the Artificial Intelligence 791 course, a variety of algorithms used to solve optimization problems have been properly studied. The premise of these algorithms is that in some cases, it is too complex for an objective function to be solved by using math, such as calculating the derivative and setting it equal to zero. Thus, a computationally smart algorithm is created that attempts to find the solution to a minimization or maximization problem by both exploring and exploiting the search space. During the course of Artificial Intelligence 791, meta-heuristics such as Simulated Annealing (SA), Particle Swarm Optimization (PSO), as well as some Evolutionary Algorithms (EA), and Differential Evolution (DE) have been discussed. However, there are many different meta-heuristics that can be used to solve optimization problems, which have not been covered. The purpose of this study is to choose a meta-heuristic not covered by the module, and to compare its performance with two known meta-heuristics. PSO and DE are chosen as the known meta-heuristics, and the Artificial Bee Colony (ABC) algorithm is used as the unknown. The ABC algorithm is based on the behaviour of honeybees, and is comprised of three phases, which aims to find better "food sources" (position vectors). In order to measure the quality of these meta-heuristics against each other, a race is conducted across various benchmark objective functions. Other factors such as time performance, code complexity, as well as consistency are also used in order to gauge which algorithm performs best.

II. Background

The Artificial Bee Colony (ABC) algorithm is a meta-heuristic used for numerical problem optimization and was introduced by Karaboga in 2005 [1]. The ABC algorithm is labelled as a "swarm-based" meta-heuristic, because the algorithm is based on the intelligent foraging behaviour of bees, which act in swarms. however, the real operation of the algorithm more closely resembles an evolutionary algorithm. In the ABC algorithm, there is a population of food sources (with an n-position in the search space) and the artificial "bees" modify this position over time. Instead of a honeybee being a particle, it can rather be seen as a computational agent with different operations that it executes on the food sources, in order to move them to more optimal locations. The honeybees in this algorithm can be categorized in three groups, namely Employed Bees, Onlooker Bees, and Scout Bees. The employed bees are more inclined to exploit the search space around the current positions of the food sources. The Onlooker bees use the information found by the Employed Bees, and further exploit food source positions based on fitness. Finally, if a food source is not improved over a predetermined number of iterations, it is discarded, and a new random position is chosen for a food source. Algorithm 1 shows the pseudo-code of the execution of the ABC:

Algorithm 1 Artificial Bee Colony Meta-Heuristic

```
Initialization Phase (A)
repeat
    Employed Bees Phase (B)
    Onlooker Bees Phase (C)
    Scout Bees Phase (D)
    Memorize the best solution achieved so far
until stopping condition is true
```

Variable definitions:

- i : Current dimension.
- n : Dimensions of the search space.
- m : Number of food sources.
- l_i, u_i : Lower and Upper bounds of the search space.
- $\overrightarrow{x_m}$: Food source with index m.
- $\overrightarrow{v_m}$: Candidate food source with index m.

- ϕ_{mi} : A random number $\in U[-a, a]$
- $f_m(\overrightarrow{x_m})$: Objective function value of food source position.
- $fit_m(\overrightarrow{x_m})$: Fitness of a food source position.
- p_m : Probability of a food source chosen by the Onlooker bee.
- a_m : The number of times a food source was not able to improve.
- a_{limit} : The limit of a_m for food source abandonment.

A. Initialization Phase

During the initialization phase, each food sources is assigned a random, uniformly distributed position in the search space:

$$x_{mi} = l_i + rand(0, 1) * (u_i = l_i) \tag{1}$$

This is done by the scout bee agents, and ensures that the each food source has a position vector populated by random values, governed by the upper and lower bounds of the search space in the dimension.

B. Employed Bees Phase

During this phase, employed bees search for new food sources ($\overrightarrow{v_m}$) which possess a better fitness value. This is done by randomly selecting a neighbour (k) and creating a candidate solution by combining the position in a single dimension (i) of the neighbour with the currently selected food source, governed by some random value $\phi_{mi} \in U[-a, a]$:

$$v_{mi} = x_{mi} + \phi_{mi}(x_{mi} - x_{ki}) \tag{2}$$

Following this combination, the candidate solution fitness is evaluated with the following equation (3):

$$fit_m\left(\overrightarrow{x_m}\right) = \left\{ \begin{array}{ll} \frac{1}{1+f_m\left(\overrightarrow{x_m}\right)} & \text{if } f_m\left(\overrightarrow{x_m}\right) \geq 0 \\ 1 + abs\left(f_m\left(\overrightarrow{x_m}\right)\right) & \text{if } f_m\left(\overrightarrow{x_m}\right) < 0 \end{array} \right\} \tag{3}$$

After the fitness is obtained, greedy selection is applied. Thus, if the fitness of the candidate solution is better than the fitness of the current food source, it replaces the current food source, and a_m is reset to zero. This phase is similar to Differential Evolution crossover.

C. Onlooker Bees Phase

Onlooker bees, as with scout bees, can be seen as unemployed. The employed bees share their information with the onlooker bees in the hive, after which the onlooker bees choose which food source they wish to exploit. This selection is determined with the roulette wheel selection method (Goldberg, 1989). The probability that a food source is selected by the onlooker bee, is dependent on the ratio of the fitness of a food source's fitness, with the total fitness of all food sources, shown in equation 4:

$$p_m = \frac{fit_m\left(\overrightarrow{x_m}\right)}{\sum_{m=1}^{SN} fit_m\left(\overrightarrow{x_m}\right)} \tag{4}$$

After a food source $\overrightarrow{x_m}$ is probabilistically determined, the same process is followed as in the employed phase (Equation 2), where a candidate food source is created by crossover with a random food source. Onlooker bees are more attracted to food sources with better fitness values, and ensures that the current "best" solutions are adequately exploited.

D. Scout Bees Phase

Scout bees can be categorized as bees that choose their food sources randomly. When a food source has not improved for a_m iterations and $a_m > a_{limit}$, it is determined that the food source needs to be abandoned. During the scout bee phase, food sources who need to be abandoned are replaced with random food sources as in equation (1). Thus food sources that are inherently poor, or are made poor by exploitation are replaced with random food sources, which may or may not lead to better results.

III. Implementation

This section includes the proper overview and algorithmic procedures for each of the known meta-heuristics.

A. Global Best PSO

The particle swarm optimization (PSO) algorithm is a population-based search algorithm based on the simulation of the social behavior of birds within a flock [1]. Particles are uniformly spread out throughout the search space during initialization. Particles make use of a velocity update formula, which allows particles to continuously levitate towards the best solutions. Particles converge on a point in the search space, if no better solutions than the current can be found.

Variable definitions:

- i : Current particle.
- j : Current dimension.
- v_{ij} : Current velocity of particle i in dimension j.
- w : The inertia component multiplier.
- c_1 : The cognitive component multiplier.
- c_2 : The social component multiplier.
- x_{ij} : Current position of particle i in dimension j.
- y_{ij} : Personal best position of particle i in dimension j.
- $\hat{y}_j$: Global best position in dimension j.
- r_1, r_2 : Random values $\in U[0, 1]$

The formula for velocity update is comprised out of a inertia, cognitive and social component and is show in equation (5):

$$\begin{aligned} v_{ij}(t+1) = v_{ij}(t) + c_1 r_{1j}(t)[y_{ij}(t) - x_{ij}(t)] \\ + c_2 r_{2j}(t)[\hat{y}_j(t) - x_{ij}(t)] \end{aligned} \tag{5}$$

After the velocity for $(t+1)$ is calculated, a new position for the particle is calculated, and tested to see if it is in the bounds of the search space. Equation (6) shows how this position update is done:

$$x_i(t+1) = x_i(t) + v_i(t+1) \tag{6}$$

The pseudo-code that explains how PSO execution is carried out, is show in Algorithm (2):

Algorithm 2 *gbest* PSO

Create and initialize an n_x-dimensional swarm;
repeat
  for each particle i = 1,...,n_s **do**
    //set the personal best position
    if $f(x_i) < f(y_i)$ **then**
      $y_i = x_i$;
    end if
    //set the global best position
    if $f(y_i) < f(\hat{y}$ **then**
      $\hat{y} = y_i$
    end if
  end for
  for each particle i = 1,...,n_s **do**
    update the velocity using equation (16.2);
    update the position using equation (16.1);
  end for
until *stopping condition is true*

For the purpose of this study, an explanation of convergence is omitted for only converging parameters are used.

B. Differential Evolution (DE/rand-to-best/n_v/z)

Differential evolution (DE) is a stochastic, population-based search strategy [1]. Although DE could be considered similar to evolutionary algorithms (EA), the key difference is that distance and direction is used as a guide for the search process.

Variable definitions:

- i : Current dimension.
- x_{i1} : Parent position vector.
- x_{i2}, x_{i3} : Other vectors used for trial vector creation.
- $u_i(t)$: Trial position vector.
- $\mathcal{J}$: Binomial crossover set.
- λ : Greed factor.

DE makes use of two strategic operations in order to navigate the search space, namely mutation, and crossover. Mutation is done by selecting a candidate parent x_{i1}, as well as two other parents, x_{i2} and x_{i3}. Equation (7) is used to create a trial vector:

$$u_i(t) = \gamma\hat{x}(t) + (1-\gamma)x_{i1}(t) + \beta(x_{i2}(t) - x_{i3}(t)) \quad (7)$$

Crossover is then executed, by randomly selecting dimensions in which the trial vector position is used in the candidate solution. This random selection is done via Binomial crossover, where p_r is the crossover probability. A set of dimensions in which crossover takes place is created, and then crossover takes place, shown by equation (8):

$$x'_{ij}(t) = \begin{cases} u_{ij}(t) & \text{if } j \in \mathcal{J} \\ x_{ij}(t) & \text{otherwise} \end{cases} \quad (8)$$

The pseudo-code for Binomial crossover can be found in Algorithm (3):

Algorithm 3 Differential Evolution Binomial Crossover for Selecting Crossover Points

$j^* \sim U(1, n_x)$;
$\mathcal{J} \leftarrow \mathcal{J} \cup \{j^*\}$;
for *each* $j \in 1, ..., n_x$ **do**
  if $U(0,1) < p_r$ *and* $j \neq j^*$ **then**
    $\mathcal{J} \leftarrow \mathcal{J} \cup \{j\}$;
  end if
end for

The complete DE algorithm pseudo-code, contained in Algorithm (4), illustrates the key considerations when working with this specific meta-heuristic:

Algorithm 4 General Differential Evolution Algorithm

Set the generation counter, $t = 0$
Initialize the control parameters, β and p_r
Create and initialize the population, $\mathcal{C}(0)$, of n_s individuals
while *stopping condition*(s) *not true* **do**
  for *each individual*, $x_i(t) \in \mathcal{C}(t)$ **do**
    Evaluate the fitness, $f(x_t(t))$;
    Create the trial vector, u_i by applying the mutation operator;
    Create an offspring, $x'_i(t)$, by applying the crossover operator;
    if $f(x'_i(t))$ *is better than* $f(x_i(t))$ **then**
      Add $x'_i(t)$ to $\mathcal{C}(t+1)$;
    else
      Add $x_i(t)$ to $\mathcal{C}(t+1)$;
    end if
  end for
end while
Return the individual with the best fitness as the solution;

IV. Empirical process

The primary measure of the *quality* of a meta-heuristic is determined by execution time and memory usage. The reason why *number of iterations* is not used, is because each meta-heuristic has different computational complexity during an iteration. As an example, a PSO iteration consists of updating each particle velocity as well as position, thus $2 * m_{particles} * n_{dimensions}$ operations is required. However, a DE iteration consists of executing the algorithm on only **one** parent, with slightly more computational complexity than the PSO. This creates a clear difference in what each algorithm is allowed to do during an iteration, and hence *quality at iteration n* is not used as a measurement of how good an algorithm is.

A. Benchmark functions

A diverse selection of benchmark functions is chosen to ensure that algorithm performance is tested for different strengths and weaknesses. The following table (1) explains these objective functions as well as the domains, and objective minimum function:

TABLE I
BENCHMARK FUNCTIONS

Function Name	Function Equation	Function Domain	Function Minimum
Exponential	$f(x) = -\exp \sum_{i=1}^{D} (x_i)^2$	$-1 \leq x_i \leq 1$ for all i = 1, ⋯ , d	x ∗ = f(0, ⋯ ,0) f(x ∗) = −1
Schwefel	$f(x) = 418.9829d - \sum_{i=1}^{d} x_i sin\left(\sqrt{\lvert x_i \rvert}\right)$	$-500 \leq x_i \leq 500$ for all i = 1, ⋯ , d	x^* = f(420.9687, ...,420.9687) f(x^*) = 0
Qing	$f(x) = \sum_{i=1}^{D} (x_i^2 - i)^2$	$-500 \leq x_i \leq 500$ for all i = 1, ⋯ , d	x^* = f($\pm\sqrt{i}$) f(x^*) = 0
Rosenbrock	$f(x) = \sum_{i=1}^{D-1} [100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$	$-30 \leq x_i \leq 30$ for all i = 1, ⋯ , d	x ∗ = f(1, ⋯ ,1) f(x^*) = 0
Brown	$f(x) = \sum_{i=1}^{n-1} (x_i^2)^{(x_{i+1}^2+1)} + (x_{i+1}^2)^{(x_i^2+1)}$	$-1 \leq x_i \leq 4$ for all i = 1, ⋯ , d	x ∗ = f(0, ⋯ ,0) f(x ∗) = 0

B. Quality control

To ensure that all meta-heuristics are on even footing with regard to effectiveness, all three algorithms have been implemented in Java, and make use of exactly the same data structures. One-dimensional and two-dimensional arrays are the only data structures used to store positions, fitness and previous best positions. Since no linked lists or array lists are used, any anomalies have been eliminated with regard to memory access time. The way in which measurements are recorded are also exactly the same across all three algorithms. Initialization is not taken into account.

C. Measurable variables

1) Code complexity: For a meta-heuristic to execute any form of computation, it needs to be programmed first. If an algorithm is too difficult to implement, it does not matter how well it performs, since the code written will likely have errors, resulting in sub-optimal algorithm performance. For this category, a software developer will rate the complexity of implementing the algorithms in Java, on a scale of one to ten.

2) Time performance: This is the true measure of the quality of an algorithm. Performance is measured as the best fitness of an algorithm at a given time. For the tests run, each algorithm is given 500ms to solve the problem. The method in which the quality is sampled per unit of time is noted below:

Algorithm 5 Quality capture per millisecond

```
Store start time of the algorithm (As a long integer value in milliseconds)
while algorithm is executing do
    Run iteration
    Capture new time
    while new time differs from current time do
        record current best value for time period
    end while
    Store new time as current time
end while
```

This allows the measurement of an algorithms current quality to happen in real time, after every iteration.

3) Consistency: Consistency is important for algorithm behaviour, as for extremely complicated problems, the algorithm can not be run many times, as computationally it might take too long. Therefore, it is important that an algorithm be consistent. Standard deviation is used to measure the consistency of an algorithm, as some algorithms have less stability than others.

D. Parameter values

Parameter values are chosen that are known to be good, and lead to optimal performance of the algorithm.

1) PSO parameter values: For particle swarm optimization, the following values are used and are known to converge:

- $n_{particles}$: 20
- w : 0.9
- c_1 : 0.7
- c_2 : 0.7

2) DE parameter values: For differential evolution, the following values are used and are known to produce good results:

- $n_{parents}$: 20
- $greed_{initial}$: 0
- $scale\ factor$: 0.5
- $p_{crossover}$: 0.5

The greed factor is updated as a factor of $current\ time$ divided by $total\ time$.

3) ABC parameter values: For the artificial bee colony algorithm, the following values are used and are known to produce good results:

- m = 20
- a_{limit} = $m * n$.

V. RESULTS

Results are obtained by allowing each algorithm 500 milliseconds of running time. If all algorithms have converged to a solution before this, the graph is adjusted in order to more clearly visualize the difference between the algorithms.

A. Benchmark function comparison

To illustrate the general performance of each algorithm, the mean at each time index is calculated over 100 individual runs.

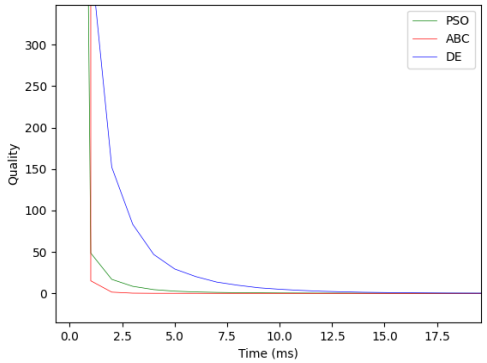

Fig. 1. Brown function. 30 dimensions. Mean Quality.

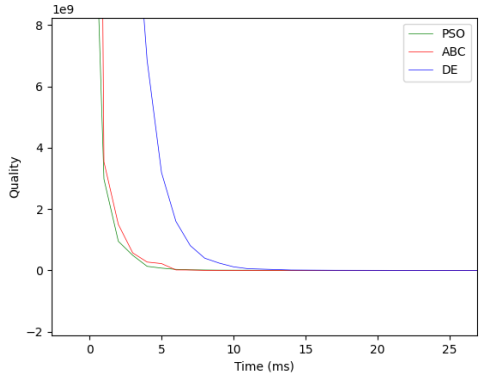

Fig. 2. Qing function. 30 dimensions. Mean Quality.

As can be seen in Figure 1 and Figure 2, for objective functions that are relatively easy to solve, ABC and PSO both excel. However, DE can be seen to lag behind slightly. This might be caused by the greed factor, which forces the algorithm to exploit current solutions instead of exploring. As soon as the greed factor increases, DE seems to do better.

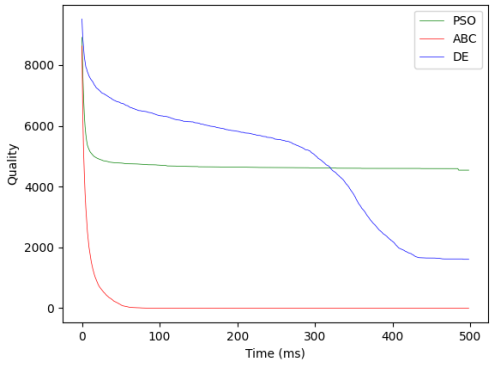

Fig. 3. Schwefel function. 30 dimensions. Mean Quality.

Figure 3 shows that when objective functions are extremely complicated, with many local minima, the ABC algorithm reigns superior, and is able to find the function minimum. In this figure, the clear impact of the greed factor in DE can be seen, as the best result obtained is exploited more and more as the time increases. PSO however, gets diverges early on, and does not find its way out of local minima.

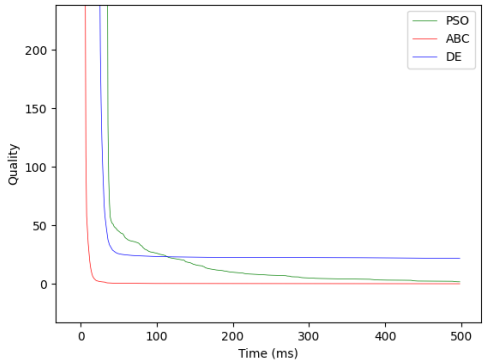

Fig. 4. Rosenbrock function. 30 dimensions. Mean Quality.

The Rosenbrock function, used to generate Figure 4, illustrates an important trade-off between PSO and DE. Since the Rosenbrock function is continuous, and requires an extreme amount of exploitation to find the global minimum, the PSO does manage to find it over time. However, DE is unable to exploit the search space well enough to find the global minimum.

B. Consistency

As discussed in the Empirical process section, an algorithm needs to be able to provide consistent results in order to be deemed superior.

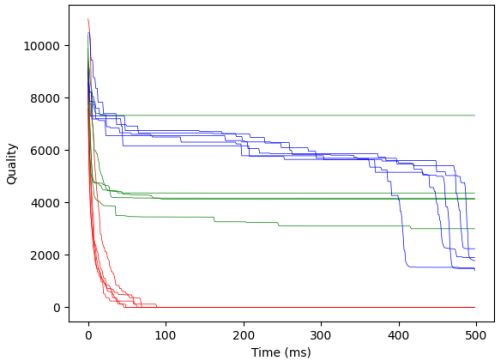

Fig. 5. Schwefel function. 30 dimensions. 5 runs.

Due to time constraints, the standard deviation was not measured, however, Figure 5 shows a clear illustration of the consistency of the selected algorithms. ABC consistently finds the same minimum, with DE having minima close to each other. PSO shows high variation in answers obtained between runs. This illustrates the dependence of the PSO on the initial particle position. Sometimes the particles are initialized at better positions than others, and therefore find better local minima.

C. Race

The race win condition is determined by two factors:

- Obtaining the best quality.
- Obtaining the best quality first.

For the purpose of the race, one relatively simple and one relatively complex benchmark function are chosen. For this study, the Qing and Rosenbrock functions are used to illustrate performance of the two algorithms. 100 independent runs are used.

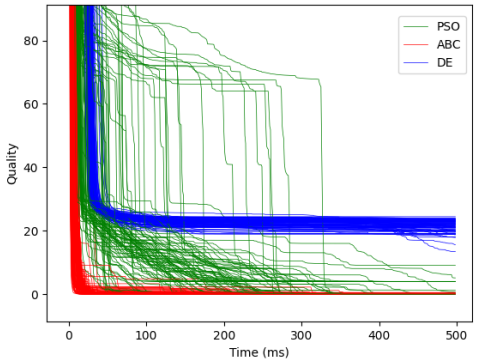

Fig. 6. Rosenbrock function. 30 dimensions. Individual runs.

Figure 6 illustrates a brief overview of the race, and shows that DE never finds the function minimum. PSO and ABC both find the minimum, with ABC being consistently faster. The results of the races are as follows:

TABLE II
ROSEBROCK FUNCTION RACE

	PSO	DE	ABC	Total
Obtained function minimum	90	0	100	100
Obtained function minimum first	0	0	100	100

TABLE III
QING FUNCTION RACE

	PSO	DE	ABC	Total
Obtained function minimum	100	93	100	100
Obtained function minimum first	5	0	95	100

Table 2 and 3 illustrate the clear superiority of the ABC algorithm, with the exception of losing the PSO in the Qing function, which shows the ability of the PSO to quickly exploit the search space. Overall, ABC is a clear winner. The DE exhibits lacking performance in these races, but is noted as being able to provide a better solution than PSO for complex objective functions.

D. Strengths and weaknesses

TABLE IV
STRENGTHS AND WEAKNESSES OF CHOSEN META-HEURISTICS

	PSO	DE	ABC
Strengths	Able to quickly exploit the search space and find the minimum, provided that the initial position of the particles is good.	Is able to find good solutions through initial exploration and later exploitation of best found solutions.	Finds the function minimum in the shortest amount of time in most cases.
Weaknesses	Can converge early to a local minimum with a bad fitness.	If the optimal minimum is not found during exploration phase, it will not be obtained in the exploitation phase.	No apparent weaknesses.

E. Code analysis

For implementing meta-heuristics in the real world, it is important that a software developer is able to understand and develop a program that executes the algorithm without bugs. To measure the algorithms next to each other, their respective code structures are analyzed and results are obtained.

TABLE V
CODE ANALYSIS TABLE

	PSO	DE	ABC
Lines of code	150	200	250
Complexity	Low	Medium	Medium
Additional Data Structures	High	Low	Medium

Table 2 shows that PSO requires the least amount of code, with low complexity and a high number of additional data structures. This is due to each particle personal best position as well as current velocity that is stored. DE requires slightly more code, with medium complexity due to the nature of the algorithm. DE only requires a positions to be stored, as well as current fitness. ABC requires the most code due to the four phases that have similarities, but are slightly different. ABC needs to store the trial count additionally.

VI. CONCLUSION

Through the various tests that the PSO, DE, and ABC algorithms have been subjected to, the ABC algorithm reigns superior. ABC exhibits fast execution, accurate results, as well as consistency. All of this with no apparent drawbacks or weaknesses. The environment in which DE and PSO excel is also explored, and the use of such algorithms are clear. DE and PSO are easier to code, and provide adequate results for relatively less complicated optimization problems. This study shows that exploring the unknown is of key importance, as there might always be a better algorithm. If no better algorithm already exists, one can always attempt to create one. Never stop improving.

REFERENCES

[1] A.P. Engelbrecht, “Computational Intelligence, An Introduction. Second Edition” University of Pretoria, South Africa, pp. 240–360, 2007.

[2] D. Karaboga, “An idea based on honey bee swarm for numerical optimization”. Erciyes University, Engineering Faculty, 2005.

[3] Niyomubyeyi, O.; Sicuaio, T.E.; Díaz González, J.I.; Pilesjö, P.; Mansourian, A. “A Comparative Study of Four Metaheuristic Algorithms, AMOSA, MOABC, MSPSO, and NSGA-II for Evacuation Planning.” Algorithms 2020, 13, 16. https://doi.org/10.3390/a13010016

[4] D. Karaboga, “Artificial bee colony algorithm.” Scholarpedia, 5(3):6915. 2010.