International Journal of Advanced Mathematical Sciences, 5 (1) (2017) 8-19

SPC

International Journal of Advanced Mathematical Sciences

Website: www.sciencepubco.com/index.php/IJAMS
doi: 10.14419/ijams.v5i1.7128
Research paper

Machine-coded genetic operators and their performances in floating-point genetic algorithms

Mehmet Hakan Satman[1*] and Emre Akadal[2]

[1]*Istanbul University, Department of Econometrics*
[2]*Istanbul University, Department of Informatics*
[*]*Corresponding author E-mail: mhsatman@istanbul.edu.tr*

Abstract

Machine-coded genetic algorithms (MCGAs) use the byte representation of floating-point numbers which are encoded in the computer memory. Use of the byte alphabet makes classical crossover operators directly applicable in the floating-point genetic algorithms. Since effect of the byte-based mutation operator depends on the location of the mutated byte, the byte-based mutation operator mimics the functionality of its binary counterpart. In this paper, we extend the MCGA by developing new type of byte-based genetic operators including a random mutation and a random dynamic mutation operator. We perform a simulation study to compare the performances of the byte-based operators with the classical *FPGA* operators using a set of test functions. The prepared software package, which is freely available for downloading, is used for the simulations. It is shown that the byte-based genetic search obtains precise results by carrying out the both *exploration* and *exploitation* tasks by discovering new fields of the search space and performing a local *fine-tuning*. It is also shown that the introduced byte-based operators improve the search capabilities of FPGAs by means of convergence rate and precision even if the decision variables are in larger domains.

Keywords: *Genetic algorithms; Optimization; Simulations*

1. Introduction

Genetic Algorithms (*GA*s) are search and optimization methods that mimic the natural selection and the principles of genetics [16, 10]. Since a *GA* perform a selection mechanism on fitness values of candidate solutions rather than the goal function itself, it does not require the goal function to be neither continuous nor differentiable. This flexibility of *GA*s always opens new research areas, for instance, in *Human-based GA*s, there is not a goal function at hand and the human judgment is used to rank candidate solutions for the selection mechanism [4, 18, 17].
Classical *GA*s perform an optimization process on an initially predefined or random population of solutions encoded using the binary alphabet. The selection operator is then applied to construct a mating-pool which is then used to re-combine candidate solutions with higher fitness values. A fitness value is a measurement of how well a candidate solution satisfies the goal function and/or a candidate solution is whether feasible or not. The generated solutions after re-combination are copied into the next generation with or without a slight modification. It is expected that the candidate solutions that have higher fitness values will then construct a better population after recombination and modification, namely, crossover and mutation, respectively.
It is natural to solve $0-1$ problems with the classical *GA*s. Since an integer can be formed up using bits that are sequenced in order, an optimization problem with integer decision variables are also natural by means of binary coding. However, representing real values is not possible because infinite number of bits are required. To cope with this issue, a sequence of moderate number of bits can be combined to map some integers to real numbers in a discreate manner. Assume that b_i is an element of a binary chromosome and Q is the mapped integer value. The corresponding real number R mapped into the range $[A,B]$ can be obtained using the formula

$$R = A + \frac{B-A}{2^L - 1} \times Q$$

where L is the length of choromosome b. With using this formula, a conversion between binary strings and real numbers in a predefined range is possible. However, this process yields another kind of problems in *GA*s, that is, as the bit strings get longer, longer number of iterations are required to obtain the global optima. Since the diversity of population is also important, candidate solutions with longer bit strings require the size of population to be higher to represent the different areas of the seach space [19, 11].
In *FPGA*s (*Floating-point genetic algorithms*), the candidate solutions are not encoded using a binary alphabet and the new solutions are recombined and altered using a different logic. The generated content is generally linear or a nonlinear function of the parents or drawn randomly in a pre-defined range. Mutation is performed by adding a single value which fallows a probability distribution with constant or dynamic parameters [20, 5]. Since the classical recombination operators are defined on bit strings, new type of genetic reproduction operators were developed for the *FPGA*.
*MCGA*s (*Machine-coded genetic algorithms*) are an other type of *GA*s which use the byte representation of real values in the computer memory. Compilers and interpreters encode numbers (includ-

Copyright © 2016 Mehmet Hakan Satman; Emre Akadal. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

ing floating-point types) using constant size byte strings using a standard. Assuming these byte strings as the *geno-type* makes the classical crossover operators applicable on the candidate solutions which are vectors of real numbers. Since the mutation operator in *MCGA*s is applied on the *geno-type* and the position of each single byte has a different effect on *pheno-type*, the byte-based mutation operator gains a similar interpretation with its binary counterpart.
It is shown that the *MCGA*s outperform some other evolutionary algorithms for some cases in a recently published study [28]. In this paper, we introduce other types of byte-based genetic operators and show their performances using a simulation study. In Section 2 we give a brief introduction for *FPGA*s. In Section 3 we show the effects of the byte-based genetic operators that are recently developed. In Section 4 we develop new kind of byte-based genetic operators for improving the genetic search. In Section 5, we give a brief introduction for the developed software for *MCGA*s. In Section 6, we perform a simulation study on a set of test functions. Finally in Section 7, we conclude.

2. Floating-point genetic algorithms

In genetic algorithms, individuals are encoded using a binary alphabet, mathematically $b_i \in \{0,1\}$, using an encoder function $r = d(b)$. The encoder function is invertible, that is, each single chromosome b can be produced using the decoder function $b = d^{-1}(r)$, where b is the binary chromosome, r is the corresponding real number such that $r \in \mathbb{R}$, $d^{-1}(.)$ is the inverse of the function $d(.)$. If a chromosome c is a vector of real values rather than binary digits, then the distinction between the encoded and the decoded values is disappeared. The genetic algorithms in which the chromosomes are directly used without encoding-decoding are called *floating-point* or *real-valued* genetic algorithms (*FPGA*s).
Classical crossover operators combine two binary chromosomes in many different ways. Since the chromosomes are real-values in *FPGA*s, binary crossover operators are not directly applicable on a real vector. As the classical mutation operator simply flips a binary value of a chromosome, it is not even possible to make an analogy for a real vector. To cope with these issues, new crossover and mutation operators are developed for *FPGA*s. Note that the other operators such as selection and elitism are not related to reproduction operators as they are only based on the fitness or cost values.
Suppose that the chromosomes A and B have real values a_1, a_2, ..., a_p and $b_1, b_2, ..., b_p$ where p is the chromosome length or the number of real values. The *Flat Crossover* [15, 25] operator produces two offspring C and D such that

$$C = (c_1, c_2, ..., c_p)$$

and

$$D = (d_1, d_2, ..., d_p)$$

where c_i and d_i are random variables which follow a Uniform distribution with parameters $\min(a_i, b_i)$ and $\max(a_i, b_i)$, and $i = 1, 2, ...p$. This operator serves a natural way of generating values in their definition ranges.
Arithmetic Crossover [20] generates two offsping C and D such that

$$c_i = \alpha a_i + (1-\alpha) b_i$$

and

$$d_i = (1-\alpha) a_i + \alpha b_i$$

where α is a random variable that follows a Uniform(0,1) distribution. α can also be selected adaptively by changing its value depending on the current number of generations. The generated variables are simply weighted means of the variables of parents. This operator also ensures the bounds of variables naturally.

BLX-α Crossover [8, 14] is an other type of crossover operators and generates the components c_i and d_i of chromosomes C and D randomly using a Uniform distribution with parameters $r_{min} - I\alpha$ and $r_{max} + I\alpha$ where $r_{min} = \min(a_i, b_i)$, $r_{max} = \max(a_i, b_i)$, and $I = r_{max} - r_{min}$. If the α parameter is selected as $\alpha = 0$ then the operator generates offspring randomly within the bounds as the *Flat Crossover* does. As the parent chromosomes A and B stay in the different areas of the solution space, the value of I increases. Since larger values of I expand the range of parameters of the Uniform distribution, the chance of obtaining diverse solutions increases. α determines the magnitude of this diversity. As the parent chromosomes get close and lay on a similar subspace of the solutions, the generated offspring get closer to parents. This operator does not ensure the boundary constraints to be satisfied.
Linear Crossover [35, 15] works in a different way as it performs an inner selection mechanism between generated offspring. The generated offspring J_1, J_2, and J_3 are defined as

$$J_1 = \frac{1}{2}A + \frac{1}{2}B$$

$$J_2 = \frac{3}{2}A - \frac{1}{2}B$$

$$J_3 = -\frac{1}{2}A + \frac{3}{2}B$$

where $J_1 = (j_{11}, j_{12}, ..., j_{1p})$, $J_2 = (j_{21}, j_{22}, ..., j_{2p})$, and $J_3 = (j_{31}, j_{32}, ..., j_{3p})$, respectively. The operator generates 3 offspring, rather than 2, and returns a vector of best two as the result. This operator may generate offspring out of the variable bounds.
Simulated Binary Crossover (SBX) simulates the single-point crossover that used in binary genetic algorithms for the real-parameters [6, 7]. Now suppose that β_i is a random variable which is defined as

$$\beta_i = \frac{d_i - c_i}{b_i - a_i}$$

and has a probability density function

$$P(\beta_i) = \begin{cases} 0.5(\eta_c + 1)\beta_i^{\eta_c} & , \beta_i \leq 1 \\ 0.5(\eta_c + 1)\frac{1}{\beta_i^{\eta_c + 2}} & , \beta_i > 1 \end{cases}$$

where η_c is a positive real number. If η_c is a small number, generated offspring will stay far away from the parents. This constant can be selected from the range of $2 \leq \eta_c \leq 5$ [5].
Now suppose that β_{q_i} is a random number that is drawn using the probability density function $P(\beta_i)$. The operator generates two offspring using formulas

$$c_i = 0.5[(1 + \beta_{q_i})a_i + (1 - \beta_{q_i})b_i]$$

and

$$d_i = 0.5[(1 - \beta_{q_i})a_i + (1 + \beta_{q_i})b_i]$$

where a_i and b_i are real parameters of parent vectors A and B, respectively. Drawing a β_{q_i} randomly using its probability density function can be performed using the *Probability Integral Transformation*. Now suppose that u is a random variable that follows a Uniform(0,1) distribution. Then the random variable w which satisfies the relation

$$u = \int_{-\infty}^{w} \varphi(x) dx$$

follows a φ distribution. Analogically,

$$\beta_{q_i} = \begin{cases} (2u)^{\frac{1}{\eta_c+1}} & ,u \leq 0.5 \\ (\frac{1}{2(1-u)})^{\frac{1}{\eta_c+1}} & ,u > 0.5 \end{cases}$$

follows a $P(\beta_i)$ distribution. Using this cumulative distribution function, randomly simulated β_{q_i} values can be obtained after drawing random numbers from the Uniform distribution.

Unfair Average Crossover [21] generates offspring C and D with components c_i and d_i using the formulas

$$c_i = \begin{cases} (1+\alpha)a_i - \alpha b_i & ,i = 1,...,j \\ -\alpha a_i + (1+\alpha)b_i & ,i = j+1,...,n \end{cases}$$

and

$$d_i = \begin{cases} (1-\alpha)a_i + \alpha b_i & ,i = 1,...,j \\ \alpha a_i + (1-\alpha)b_i & ,i = j+1,...,n \end{cases}$$

where $1 \leq j \leq n$ is the randomly selected cut-point, α is a random variable that follows a Uniform(0,0.5) distribution. Since some weights are negative, this operator does not ensure the boundary constraints of variables to be satisfied.

As it is shown above, crossover operators generate offspring that random, linear or non-linear functions of parents, numerically. The crossover operators developed for *FPGA*s, do not mimic their binary counterparts as they are not performed on the basic elements of numbers, namely *geno-type*. Even though the *FPGA* recombination operators perform successful searches in many optimization problems, there is a need for some enhanced operators, at least in some cases. [13] stressed that the main problem in *FPGA*s is the balance of *exploration* and *exploitation* and as a consequence, the problem of premature convergence. In addition to the selection of right amount of population size, maximum number of iterations, and number of elitist solutions; it is vital to select the right combination of genetic recombination operators which perform the tasks of *exploration* and *exploitation*.

3. Machine-coded genetic operators

Natural coding of human DNA consists of an alphabet § with four digits or elements, that is, $\S = \{A,T,C,G\}$. In an optimization problem, if the decision variables have only two values, the natural alphabet is $b = \{0,1\}$. For instance, in the *Knapsack* problem [23, 22], chromosomes are naturally bit strings and each single bit value is mapped to a decision variable. When the decision variables are in type of integer, more than one bits are combined to form up the value of an integer variable using encoder and decoder functions. Finally, presentation of real numbers is even possible in a discrete manner, but not continuously. Binary coding in classical genetic algorithms is the natural way of presenting variables and the classical reproduction operators such as crossover and mutation perform sensible tasks.

*FPGA*s are also genetic algorithms in which candidate solutions are not encoded using an alphabet and genetic operators are performed directly on them. Dropping encoding-decoding scheme makes the classical reproduction operators useless. Special reproduction operators developed for FPGAs combine parents linearly or non-linearly using arithmetic operations. However, real values in computer memory are already stored using byte values in a discrete manner but with a great level of precision which can be satisfied using longer chromosomes in classical *GA*s. The classical crossover operator can be directly applied on the byte representation of real values. Classical mutation operator can also be simulated with its byte-based counterpart.

Suppose that a variable with the type of `double` in C++ language has the value of 1234.56789. This value is encoded in computer memory using the byte values in type of `unsigned char`

$$b_i = (231,198,244,132,69,74,147,64)$$

using the encoding standard *IEEE 754 - (IEEE Standard for Floating-Point Arithmetic)* [9, 31] where $0 \leq b_i \leq 255$ for $i = 1,2,...,8$. This coding scheme makes classical crossover operator applicable as the number is represented in an elementary way. Now suppose that an other `double`-typed value of 612347676.566 is encoded using the same standard as

$$c_i = (176,114,72,142,215,63,194,65)$$

where $0 \leq c_i \leq 255$ for $i = 1,2,...,8$. Now the single-point, two-point and uniform crossover operators are directly applicable on b_i and c_i as in the binary genetic algorithms. By selecting a random cut-point $k = 4$, the one-point crossover generates two offspring d_i and e_i as

$$d_i = (231,198,244,132,215,63,194,65)$$

$$e_i = (176,114,72,142,69,74,147,64)$$

which are the byte-representations of `double`-typed values 612347657.9123 and 1234.5679, respectively. As it is shown above, the offspring d_i has a value very close to parent c_i. Addition to this, e_i is almost equal to b_i with a slight change. Table 1 shows all possible offspring values that one-point crossover operator can generate. Note that, since the number of generated offspring is limited with the number of cut points, the *uniform* crossover can be selected for obtaining maximum number of unique offspring.

Table 1: One-point byte-crossover on two parents

Cut-point	Offspring_1	Offspring_2
1	612347676.5660	1234.5679
2	612347676.5686	1234.5679
3	612347677.9123	1234.5679
4	612347657.9123	1234.5679
5	612272905.9123	1234.7105
6	613714697.9123	1231.9605
7	80908641.2390	9343.6840

As mentioned above, byte-crossover operator serves a natural way of applying binary crossover operators on the real-valued chromosomes. The byte-mutation operator works in a way that similar to its binary counterpart. First, the operator selects a byte b_r randomly with the probability of mutation P_m. Then a random value u is selected using a Uniform(0,1) distribution. If $u < 0.5$ then the byte value b_r is increased by 1, otherwise, it is decreased by 1 [28]. If the mutated byte value b_r^* is greater than the maximum byte value 255, then it is altered as $b_r^* = 0$. Similarly, if the mutated value is smaller than the minimum byte value 0, then is is altered as $b_r^* = 255$.

Table 2 shows the differential effect of a mutated byte on the value of 1234.56789. In the first row of Table 2, the most left byte is increased by 1 and a very small change is observed on the corresponding value. In the second row, it can be shown that the effect of the second byte is more clear but it is still small. After changing 7th byte value from 147 to 148 the integer part of the corresponding value changes from 1234 to 1298. Finally, a change on the most right byte has the maximum effect and the resulted value has an integer part of 80908641. The byte-mutation operator performs a similar task when it is compared to binary mutation operator as the difference between the original and the mutated value depends on the location of the mutated gene. The byte-coded mutation operator also performs local fine-tuning operations well in a wide range.

Table 2: Effect of the byte-coded mutation operator

b_i^*	Value
(**232**, 198, 244, 132, 69, 74, 147, 64)	1234.5678900000026135240
(231, **199**, 244, 132, 69, 74, 147, 64)	1234.5678900005824163964
(231, 198, **245**, 132, 69, 74, 147, 64)	1234.5678900149011951725 7
(231, 198, 244, **133**, 69, 74, 147, 64)	1234.5678938146972996037 2
(231, 198, 244, 132, **70**, 74, 147, 64)	1234.5688665625000339787 2
(231, 198, 244, 132, 69, **75**, 147, 64)	1234.8178900000000339787 2
(231, 198, 244, 132, 69, 74, **148**, 64)	1298.5678900000000339787 2
(231, 198, 244, 132, 69, 74, 147, **65**)	80908641.2390400022268295
(231, 198, 244, 132, 69, 74, 147, 64)	1234.5678900000000339787 2
(231, 198, 244, 132, 69, 74, 147, **63**)	0.0188380110168457036 4
(231, 198, 244, 132, 69, 74, **146**, 64)	1170.5678900000000339787 2
(231, 198, 244, 132, 69, **73**, 147, 64)	1234.3178900000000339787 2
(231, 198, 244, 132, **68**, 74, 147, 64)	1234.5669134375000339787 2
(231, 198, 244, **131**, 69, 74, 147, 64)	1234.5678861853027683537 2
(231, 198, **243**, 132, 69, 74, 147, 64)	1234.5678899850988727848 7
(231, **197**, 244, 132, 69, 74, 147, 64)	1234.5678899999418263178 1
(**230**, 198, 244, 132, 69, 74, 147, 64)	1234.5678900005801426596

4. Improving the Search Capabilities

The byte based crossover and mutation operators that are mentioned in Section 3 outperform some other optimization techiques on some test functions as in reported in [28]. Addition to one-point crossover operator, the other well-known crossover operators such as two-point crossover and uniform crossover can be implemented in a way similar to their binary counterparts.

Despite the crossover operator can be directly applied on the byte-representations of candidate solutions, the byte-based mutation operator can be applied in several ways. In the previously reported and first case, a randomly selected byte is altered by adding a $+1$ or -1 with probabilty $1/2$. This type of mutation changes the real value of chromosomes depending on the location of the mutated byte but more generations are required to achive a part of a solution space because the amount of alteration is constant and almost always equal to 1.

In the second case, a randomly selected byte can be drawn randomly in the range of $0-255$, which is the natural range of byte representation. This operator serves a different approach to the binary mutation and can mimic the behaviour as the effect of mutation depends on the location of the mutated byte. As a result of this, the tasks of searching different areas of the solution space and performing the local *fine-tuning* is carried out by the operator in a more effective way when it is compared to the original byte-based mutation operator.

The two-variables *Easom* function is defined as

$$f(x,y) = -\cos(x)\cos(y)\exp\Big(-\big((x-\pi)^2+(y-\pi)^2\big)\Big)$$

for $-100 \leq x,y \leq 100$ and used to test the performances of some evolutionary algorithms [2, 3, 27]. The function has a global minimum of -1 for $x=\pi$ and $y=\pi$. Altough the function is smooth and has only two variables, many optimization methods including *Hooke-Jeeves* and *Nelder-Mead* fail to find the global minimum despite using many starting points. For a good starting point, for instance $(3.14, 3.14)$, *Nelder-Mead* reports

$$x = 3.141591488522633923$$

and

$$y = 3.141592522488581451$$

for tolerance of 10^{-10} and maximum number of iterations 10^6 where $\pi = 3.141592653589793116$ with the same number of digits.

A genetic algorithm with byte-based uniform crossover and byte-based random mutation reports the solution as [1]

$$x = 3.1415926516056060791$$

and

$$y = 3.1415926665067672729$$

which is more precise than the classical optimization method. *MCGA* operators, unlike the reproduction operators in evolutionary algoritms in general, obtain a solution close to the real solution without using an external optimization techinique in terms of hybridization. In Figure 1, the best and the average fitness values of the genetic search is plotted by generations. It is shown that the byte-based operators perform the tasks of *exploration* and *exploitation*. The algorithm suddenly discovers a better solution in *Generation* ≈ 50 and performs a local *fine-tuning* using this solution to obtain a solution closer to the real solution in following generations. Finally, the reported $f(x,y)$ value is almost equal to its minimum for the precise estimates of x and y.

In Figure 1, it is also shown that the average fitness does not converge to the best solution by iterations. It is directly related to the mutation probability, that is, algorithm always alters solutions to *get* closer to the real solution by means of *exploration*. Alternatively, the mutation probability can be kept higher in very early stages of the genetic search and shrinked by generations and finally set to zero in last iteration. Now suppose that the mutation probability at generation t is defined as

$$P_m^{(t)} := P_m - \frac{t}{T} \times P_m$$

where P_m is the initial mutation probability, t is the current number of generations, and T is the maximum number of generations. Using this formula, the mutation probability for the initial population is set to P_m whereas it is 0 in the last generation when $t=T$. If the initial mutation probability P_m is set to a higher value, then the task of *exploration* is more prominent than the task of *exploitation* because the algorithm tends to jump many different areas of the seach space frequently, however, as the t increases by time, the $P_m(t)$ also decreases and the task of *exploitation* becomes more prominent and the population converges to the real solution. In Figure 2, convergence of the average fitness values is shown when the dynamic mutation probabilities are used instead. It is also shown in Figure 2

[1] Maximum number of generation = 500, population size = 200, selection method = Tournament selection, mutation probability = 0.20, crossover probability = 1.0, number of elitist solutions = 2.

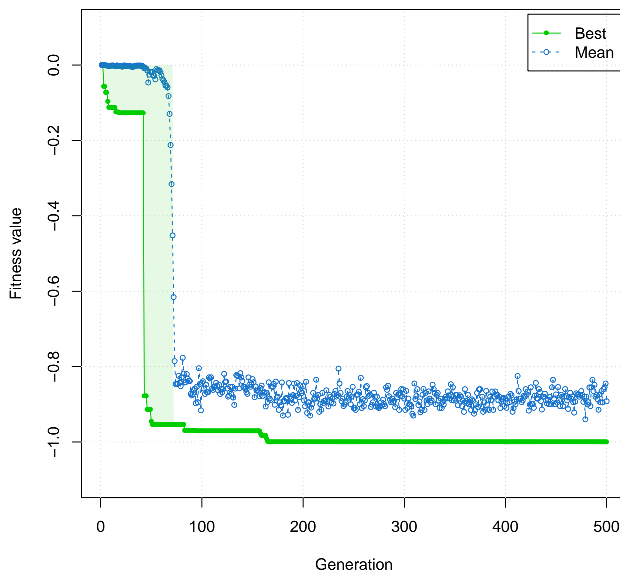

Figure 1: Minimum of Easom function by generations

that as the mutation probability decreases by iterations, the average fitness approaches to the best fitness, that is, not only a single candidate solution but the whole population converges.

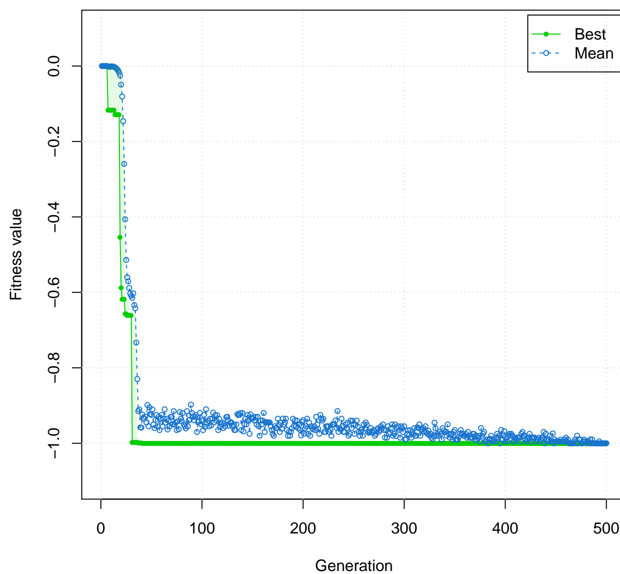

Figure 2: Minimum of Easom function by generations with dynamic mutation probabilities

The $n-$variable Rastrigin function is defined as

$$f(\vec{x}) = \sum_{i=1}^{n} x_i^2 - 10\cos(2\pi x_i) + 10$$

for

$$-5.12 \leq x_i \in \vec{x} \leq 5.12$$

and it is used in the simulation study in Section 6. The function has many local optima and it is a hard job to find to global minimum $f(\vec{x}) = 0$ for $\vec{x} = [0,0,...,0]^T$ even in a narrower domain. Since the original domain is $[-5.12, 5.12]$ for all x_i in $\vec{x}$, the function has finite number of local optima. When the domain is changed to $(-\infty, \infty)$, the function has infinite number of local optima and the problem of finding the global minimum gets more difficult. Table 3 summarizes the results of a mini simulation with a limited genetic search. The number of maximum iterations is set to 500 so the genetic search is prevented from finding the global minimum in later generations for comparison. The population size is set to 100 for all operators. This configuration is limited for such a function like Rastrigin with 50 variables with domains $[-10^{30}, 10^{30}]$, because, a moderate size of population and a limited number of generations may not be enough to represent different fields of the search space and lucky crossover and mutation operations may never come to pass to achieve the global minimum. In Table 3, it is shown that the winners are byte-based (uniform) crossover, linear crossover, and unfair-average crossover when the succeeding criteria are both the achievement the global minimum and time efficiency. The genetic search with byte-based uniform crossover is run with the dynamic random mutation operator, whereas the others run with the random uniform mutation. Note that the values that are reported in Table 3 are machine and implementation dependent, that is, the order of magnitude of algorithms can differ from the time efficiency when the operators are implemented in different algorithms and run in different machines. Byte-based crossover and mutation operators also perform a bound check which can take additional time.

5. Prepared Software

We prepared an `R` [24] package, namely `mcga`, and it is freely available for downloading on *CRAN*[2]. The package includes functions for byte-based genetic operators which can co-operate with the function `ga` of package `GA` [30]. `mcga` implements the byte-based genetic operators

- `byte_crossover`,
- `byte_crossover_1p`,
- `byte_crossover_2p`,
- `byte_mutation`,
- `byte_mutation_dynamic`,
- `byte_mutation_random`,
- `byte_mutation_random_dynamic`

for uniform crossover, one-point crossover, two-point crossover, byte-based ± 1 mutation, byte-based ± 1 mutation with dynamic mutation probabilities, byte-based random mutation, and byte-based random mutation with dynamic mutation probabilities, respectively. In `ga` function of package `GA`, user-defined crossover and mutation arguments can be set using the argument names `crossover=` and `mutation=`, respectively. For instance, a genetic search with uniform crossover and dynamic mutation requires an `R` function call like

```
R> GA::ga(type = "real-valued", fitness = f,
+     crossover = byte_crossover,
+     mutation = byte_mutation_dynamic,
+     pcrossover = 0.80,
+     pmutation = 0.30,
+     maxiter = 300,
+     popSize = 100,
+     min = rep(-100, p),
+     max = rep(100, p)
+ )
```

[2]The web page for the package is https://cran.r-project.org/web/packages/mcga/index.html and it can be installed and loaded by the standard `R` functions `install.packages` and `require`.

Table 3: 50−variables Rastrigin with domain $x_i \in (-10^{30}, 10^{30})$

	Byte	SBX	Flat	Arithmetic	BLX	Linear	Unfair
Fitness	0	$> 10^{56}$	$> 10^{57}$	$> 10^{57}$	$> 10^{56}$	0	0
Time	13.23	19.11	28.41	12.20	34.59	10.44	10.8
Time%	100	144.44	214.73	92.21	261.45	78.91	81.63

where `"real-valued"` is the string for type of the decision variables, `f` is the goal function to be maximized, `byte_crossover` is the desired crossover function,
`byte_mutation_dynamic` is the desired mutation function, `0.80` is the crossover probability, `0.30` is the first value of the mutation probability which will be then decreased by iterations, `300` is the maximum number of iterations or generations, `100` is the population size, `rep(-100, p)` is the vector of lower bounds of the p-decision variables, and `rep(100, p)` is the vector of upper bounds of the p-decision variables, respectively. The byte-based reproduction operators listed above wraps all of the low-level byte operations that are written in the `C++` language. As a consequence, the goal function to be optimized is a simple `R` function which takes a real vector as an argument. The defined recombination operators generate offspring when they are called during the optimization process and hide the low-level details from the user. This type of design also makes possible to develop and implement new kind of byte-based operators, possibly, in a future work.

6. Simulation study

We prepare a simulation study to reveal the performances of the byte-based crossover and mutation operators with the recently developed genetic operators that are developed for the *FPGA*. The simulation suite consists on the well-known test functions that are used for comparing the performances of the optimization algorithms and they are listed in Table 4 [12, 32, 34, 26, 1, 29, 33]. Since the test functions include *summation* and *production* operators, the number of parameters p is variable and we set to $p = 25$, $p = 50$, and $p = 75$. By using these scales, it is planned to measure the performances of the operators in different dimensions of the search spaces by means of tendency of failure respect to an increase on the number of parameters. Note that the genetic search tends to converge the global minimum of 0 of all functions in the test suite when the population size and the maximum number of generations are increased. We set the population size to 100 and the maximum number of generations to 200 for all configurations to make the comparison more clear, that is, a genetic search with different crossover and mutation operators has a different convergency rate in early stages of the optimization process. The crossover and the mutation probabilities are set to 0.80 and 0.10, respectively. In each single generation, the best 5 solutions are directly copied into the next population without any ruining. Simulations are performed using the `R` software [24] with packages `GA` [30] and `mcga` [28] are installed. The simulation results are shown in Table 5, Table 6, and Table 7 for $p = 25$, $p = 50$, and $p = 75$, respectively.

Table 5 summarizes the results for $p = 25$. In Table 5, functions are listed in the column 1. For each single row, results are reported for the byte-based mutation operator and the *FPGA* mutation operator which is based on adding some random values on randomly selected variables. Columns $3-9$ shows the results for different kind of crossover operators. In Table 5, it is shown that in a genetic search with byte-based mutation and crossover operators, the average of the minimum value of Ackley function is reported as 0, that is, performing genetic search on the Ackley function 250 times, we obtain 0 in average. It is also shown in the Table 5 that the byte-based crossover and the *FPGA* mutation operator results a higher average value 1.294. Similarly, a genetic search with the byte-based operators result 0 in average, whereas the minimum is reported as 177.261 by the byte-based crossover and the *FPGA* mutation operator for the function Bohacevsky. The linear crossover operator is the winner and it works well with either standard and byte-based mutation operators. Finally, in total of $14 \times 7 = 98$ cases, the byte-based mutation operator exposes a better or equal performance in 85 cases when different kind of crossover operators are used. When the byte-based crossover and the byte-based mutation operators are combined together, the byte-based operators have higher or equal performance with the winner for the functions Ackley, Bohacevsky, Bolzman, Hyper-ellipsoid, Maxmod, Multimod, Rastrigin, Schaffer, Schwefel, Sphere, and Sumsquares. The byte-based operators are out-performed by the other operators in a single case in which the Levy function is used.

Table 6 shows the simulation results for $p = 50$ and follows the same logic in representing the simulation results for $p = 25$. The byte-based mutation operator exposes a better or equal performance in 77 out of 98 cases when different kind of crossover operators are used. The evarage of genetic searches have better or equal performances of the winner for the functions Ackley, Bohacevsky, Holzman, Hyperellipsoid, Multimod, Rastrigin, Schewel, Sphere, and Sumsuares. Differently, the results of the byte-based operators are the worst cases for none of the functions.

Table 7 shows the simulation results for $p = 75$ and follows the same logic in representing the simulation results as in the Tables 5 and 6. The byte-based mutation operator exposes a better or equal performance in 79 out of 98 cases when different kind of crossover operators are used. The evarage of genetic searches have better or equal performances of the winner for the functions Ackley, Hyperellipsoid, Multimod, Rastrigin, Schewel, Sphere, and Sumsuares. In none of cases, the byte-based operators demonstrate the worst performance.

The byte-based mutation operator used in simulations is the original mutation operator reported in [28] and increases or decreases the value of a randomly selected byte by 1. As a result of using a higher cardinality of an alphabet, in some cases, more iterations are needed to achieve the correct byte value by mutations. Alternatively, random mutation can be used instead. As mentioned before, random byte-based mutation operator alters a randomly selected byte in the pre-defined range of $0-255$. Again, as a result of using a higher cardinality of an alphabet, the mutation probability is generally set to higher values, at least in early generations of the genetic search. The byte-based dynamic mutation starts its job for a given mutation probability at generation 0 and reduces the amount of this value by generations and finally sets it to 0 in the final generation and performs the alteration by changing byte values by 1. Similary, byte-based random dynamic mutation applies the random mutation operator by decreasing the mutation probability by iterations. To reveal the effects of the dynamic mutation probabilities, we perform an other simulation study with the same set of functions but this time for only the byte-based operators. Table 8 and Table 9 summarize the results.

In Table 8, the results of the byte-based mutation operators are reported for $p = 25$, $p = 50$, and $p = 75$ for the set of functions. The mutation probabilities are set to 0.10, 0.95, and 0.30 for random, dynamic and random dynamic mutation operators, respectively. Note that the mutation probabilites for dynamic operators are initial and they are reduced by iterations and finally set to 0 at last iteration. The values are average of 250 iterations. It is shown that in Table 8 that the dynamic byte-based operators have better performances for only the Griewank function and the performance does not change in 11 out of 14 cases for $p = 25$. For $p = 50$, the results are different. Dynamic mutation operators have better performances in 5 out of 14 cases and the performance does not change in 8 out of 14 cases. Summarizing these, the performance is better or does not change in 13 out of 14 cases. For the function Levy, original byte-based mutation has a better performance but the difference is small. We have similar results when the number of parameters p is 75. Dynamic mutation operators have better performances in 8 out of 14 cases. Addition to this, the performance does not change in 5 out of 14 cases. The function Levy has the different status again and the dynamic mutation operators obtain nearly same results with a small and less performance.

Note that there is not a general tumb-of-rule of determining the mutation probability and the initially selected probability can dramatically change the performances. In Table 9, the results of the same simulation study but this time with the mutation probabilites of 0.1, 0.5, and 0.5 for random, dynamic and random dynamic mutation operators are showed, respectively. It is shown in Table 9 that the smaller dynamic initial mutation probabilities have not better performances on the selected set of test functions. This results are also affected by the maximum number of iterations. When the

Table 4: Test Functions Used in Simulations

Function	Definition	Domain
Ackley	$20\exp(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2})-$ $\exp(\frac{1}{n}\sum_{i=1}^{n} cos(2\pi x_i))+20+e$	$-30 \leq x_i \leq 30$
Bohachevksy	$\sum_{i=1}^{n}(x_i^2+2x_{i+1}^2-0.3\cos(3\pi x_i)$ $-0.4\cos(4\pi x_{i+1})+0.7)$	$-50 \leq x_i \leq 50$
Griewank	$1+\frac{1}{4000}\sum_{i=1}^{n} x_i^2-\prod_{i=1}^{n} cos(\frac{x_i}{\sqrt{i}})$	$-600 \leq x_i \leq 600$
Holzman	$\sum_{i=1}^{n} i x_i^4$	$-10 \leq x_i \leq 10$
Hyperellipsoid	$\sum_{i=1}^{n} i^i + x_i^2$	$-5.12 \leq x_i \leq 5.12$
Levy	$\sin^2(\pi y_0)+$ $\sum_{i=0}^{n-2}(y_i-1)^2(1+10\sin^2(\pi y_i+1))+$ $(y_{n-1}-1)^2(1+\sin^2(2\pi x_{n-1}))$	$y_i = 1+\frac{x_i-1}{4}$ $-10 \leq x_i \leq 10$
Maxmod	$max(\|x_i\|)$	$-10 \leq x_i \leq 10$
Multimod	$\sum_{i=1}^{n}\|x_i\|\prod_{i=1}^{n}\|x_i\|$	$-10 \leq x_i \leq 10$
Rastrigin	$\sum_{i=1}^{n} x_i^2-10\cos(2\pi x_i)+10$	$-5.12 \leq x_i \leq 5.12$
Rosenbrock	$\sum_{i=2}^{n} 100(x_i-x_{i-1}^2)^2+(1+x_{i-1})^2$	$-10 \leq x_i \leq 10$
Schaffer	$\sum_{1}^{n-1}((x_i^2+x_{i+1}^2)^{1/4}$ $\sin(50(x_i^2+x_{i+1}^2)^{1/10}))^2+1$	$-100 \leq x_i \leq 100$
Schwefel	$\sum_{i=1}^{n}\{\sum_{j=1}^{j<i} x_i\}^2$	$-10 \leq x \leq 10$
Sphere	$\sum_{i=1}^{n} x_i^2$	$-10 \leq x_i \leq 10$
Sumsquares	$\sum_{i=0}^{n-1} i x_i^2$	$-10 \leq x \leq 10$

Table 5: Mean fitness values for $p = 25$

Function	Mutation	Byte	SBX	Flat	Arithmetic	BLX	Linear	Unfair
Ackley	Byte	0	0.048	1.584	1.557	1.164	0	0.008
	Real	1.294	6.278	3.61	4.259	1.651	0	0
Bohachevsky	Byte	0	1.438	22.124	23.042	54.459	0	0.049
	Real	177.261	289.668	64.494	92.817	63.803	0	0.001
Griewank	Byte	0.002	0.405	1.112	1.127	1.49	0	0.011
	Real	3.876	4.43	1.607	2.008	1.582	0	0.004
Holzman	Byte	0	0.031	0.103	0.118	25.411	0	0
	Real	0	52.716	1.861	2.634	85.328	0	0
Hyperellipsoid	Byte	0	0.013	0.358	0.404	1.935	0	0
	Real	0	12.953	2.05	3.128	2.632	0	0
Levy	Byte	2.453	1.833	0.679	0.873	0.726	0	2.001
	Real	1.271	5.062	0.453	0.689	0.754	0	1.808
Maxmod	Byte	0	0.799	0.475	0.516	0.566	0	0.004
	Real	1.514	1.281	0.91	0.988	0.92	0	0
Multimod	Byte	0	0	0	0	0	0	0
	Real	0	0	0	0	0.45	0	0
Rastrigin	Byte	0	0.043	6.806	5.14	13.913	0	0.013
	Real	0	57.656	14.109	17.189	24.749	0	0
Rosenbrock	Byte	24.969	54.933	59.048	61.115	580.811	0	23.393
	Real	59.324	816.942	161.611	190.214	1544.641	0	23.092
Schaffer	Byte	0	2.842	29.849	27.783	18.711	0.001	0.147
	Real	49.428	137.788	49.009	56.881	17.963	0.001	0.033
Schwefel	Byte	0	0.817	14.167	15.643	101.578	0	0.001
	Real	0	655.942	80.038	129.418	147.615	0	0
Sphere	Byte	0	0.005	0.126	0.141	0.533	0	0
	Real	0	3.851	0.679	1.073	0.661	0	0
Sumsquares	Byte	0	0.048	1.423	1.526	7.392	0	0
	Real	0	51.571	7.18	11.572	10.233	0	0

Table 6: Mean fitness values for $p = 50$

Function	Mutation	Byte	SBX	Flat	Arithmetic	BLX	Linear	Unfair
Ackley	Byte	0	1.894	3.797	3.857	4.094	0	0.102
	Real	6.121	6.894	5.919	6.643	4.272	0	0
Bohachevsky	Byte	0	128.977	173.791	198.861	634.252	0	0.531
	Real	1715.242	862.481	603.124	792.846	535.86	0	0.003
Griewank	Byte	0.053	2.431	2.749	2.999	8.232	0	0.13
	Real	22.532	11.236	7.919	10.358	7.183	0	0.008
Holzman	Byte	0	80.115	9.939	12.707	5673.944	0	0.001
	Real	452.34	413.445	116.911	155.7	31541.24	0	0.001
Hyperellipsoid	Byte	0	8.367	10.493	12.735	53.307	0	0.004
	Real	0.058	81.879	45.693	59.439	49.24	0	0
Levy	Byte	4.861	5.284	2.379	2.752	5.373	0	4.354
	Real	4.412	53.691	2.687	3.283	8.74	0	4.043
Maxmod	Byte	1.518	1.305	1.053	1.102	2.308	0	0.043
	Real	4.233	1.656	1.608	1.626	4.595	0	0
Multimod	Byte	0	0	0	0	0	0	0
	Real	0	0	0	0	> 1000000	0	0
Rastrigin	Byte	0	8.502	55.884	49.377	106.133	0	0.304
	Real	0	281.093	62.923	75.051	113.491	0	0
Rosenbrock	Byte	53.352	881.317	367.65	396.98	29782.03	0	48.796
	Real	3772.472	2548.67	1369.221	1631.746	150504.3	0	48.539
Schaffer	Byte	0.113	41.752	112.384	112.842	90.65	0.002	0.711
	Real	166.896	391.971	142.656	154.521	81.009	0	0.055
Schwefel	Byte	0	591.174	837.454	990.897	5605.509	0	0.042
	Real	150.019	9187.583	4083.564	6103.616	5623.049	0	0.002
Sphere	Byte	0	1.485	1.916	2.191	8.015	0	0.01
	Real	0.187	11.29	7.696	10.358	6.807	0	0
Sumsquares	Byte	0	32.996	40.417	46.195	208.287	0	0.013
	Real	4.723	299.023	169.286	234.566	188.943	0	0

Table 7: Mean fitness values for $p = 75$

Function	Mutation	Byte	SBX	Flat	Arithmetic	BLX	Linear	Unfair
Ackley	Byte	0	4.309	5.184	5.463	6.202	0	0.195
	Real	8.912	6.802	7.307	7.775	6.544	0	0
Bohachevsky	Byte	5.116	747.24	589.389	697.32	2360.041	0	1.348
	Real	5111.868	1308.588	1690.507	2049.271	2178.298	0	0.009
Griewank	Byte	0.9	9.409	7.417	8.524	28.747	0	0.277
	Real	63.822	16.846	20.864	25.33	26.622	0	0.013
Holzman	Byte	0.249	523.824	89.572	115.241	60087.81	0	0.014
	Real	9394.847	1179.416	874.926	920.62	447819.4	0	0.018
Hyperellipsoid	Byte	0	76.486	56.233	68.987	289.971	0	0.019
	Real	0.872	184.472	197.084	239.135	289.815	0	0
Levy	Byte	7.178	16.63	4.484	5.135	20.172	0	6.679
	Real	9.835	128.98	6.378	7.606	41.243	0	6.269
Maxmod	Byte	3.931	1.604	1.499	1.507	4.108	0	0.114
	Real	5.875	1.871	2.007	1.968	7.12	0	0.001
Multimod	Byte	0	0	0	0	0	0	0
	Real	0	0	0	0	> 1000000	0	0
Rastrigin	Byte	0	54.623	144.001	135.201	275.679	0	0.831
	Real	0.049	585.346	141.157	171.444	290.118	0	0.027
Rosenbrock	Byte	109.211	3182.507	1150.206	1283.993	213157.8	0	77.112
	Real	31932.39	4260.689	4576.93	4772.071	1453959	0	74.04
Schaffer	Byte	0.86	129.83	214.884	221.946	203.911	0.003	1.058
	Real	311.443	647.206	246.625	259.942	206.619	0	0.069
Schwefel	Byte	0	9430.678	7382.519	8708.908	44800.49	0	0.285
	Real	8449.909	33311.05	28855.81	37370.13	46206.6	0	0.011
Sphere	Byte	0	9.33	7.159	8.639	30.998	0	0.043
	Real	5.48	17.949	22.211	26.552	28.889	0	0
Sumsquares	Byte	0	288.37	221.715	260.394	1135.356	0	0.051
	Real	237.674	717.93	739.906	907.269	1120.843	0	0

Table 8: pmutation=0.1, 0.95, 0.3, respectively

p	Functions	random	dynamic	random_dynamic
25	Ackley	0	0	0
	Bohachevsky	0	0	0
	Griewank	0.007	0	0
	Holzman	0	0	0
	Hyperellipsoid	0	0	0
	Levy	1.306	2.007	1.426
	Maxmod	0	0	0
	Multimod	0	0	0
	Rastrigin	0	0	0
	Rosenbrock	23.674	23.867	23.611
	Schaffer	0	0	0
	Schwefel	0	0	0
	Sphere	0	0	0
	Sumsquares	0	0	0
50	Ackley	0	0	0
	Bohachevsky	2.029	0	0
	Griewank	0.795	0	0.003
	Holzman	0	0	0
	Hyperellipsoid	0	0	0
	Levy	3.713	4.369	3.77
	Maxmod	1.555	0	0
	Multimod	0	0	0
	Rastrigin	0	0	0
	Rosenbrock	54.439	48.939	48.826
	Schaffer	0.001	0	0
	Schwefel	0	0	0
	Sphere	0	0	0
	Sumsquares	0	0	0
75	Ackley	0.278	0	0
	Bohachevsky	181.757	0	0.057
	Griewank	2.232	0	0.271
	Holzman	0.684	0	0
	Hyperellipsoid	0	0	0
	Levy	6.21	6.791	6.239
	Maxmod	4.253	0	0.003
	Multimod	0	0	0
	Rastrigin	0	0	0
	Rosenbrock	97.479	73.973	73.911
	Schaffer	0.385	0	0
	Schwefel	0.351	0	0.003
	Sphere	0	0	0
	Sumsquares	0	0	0

Table 9: pmutation=0.1, 0.5, 0.5, respectively

p	Function	random	dynamic	random_dynamic
25	Ackley	0	0	0
	Bohachevsky	0	0	0
	Griewank	0.004	0	0
	Holzman	0	0	0
	Hyperellipsoid	0	0	0
	Levy	1.299	2.037	1.723
	Maxmod	0	0	0
	Multimod	0	0	0
	Rastrigin	0	0	0
	Rosenbrock	24.676	23.929	23.759
	Schaffer	0	0	0
	Schwefel	0	0	0
	Sphere	0	0	0
	Sumsquares	0	0	0
50	Ackley	0	0	0
	Bohachevsky	2.737	0	0.138
	Griewank	0.772	0	0.147
	Holzman	0	0	0
	Hyperellipsoid	0	0	0
	Levy	3.687	4.477	4.349
	Maxmod	1.6	0	0
	Multimod	0	0	0
	Rastrigin	0	0	0
	Rosenbrock	52.056	48.976	48.944
	Schaffer	0	0	0
	Schwefel	0	0	0.001
	Sphere	0	0	0
	Sumsquares	0	0	0
75	Ackley	0.28	0	0.246
	Bohachevsky	193.02	0	113.008
	Griewank	2.238	0	1.76
	Holzman	0.01	0	17.148
	Hyperellipsoid	0	0	0.17
	Levy	6.211	6.853	6.955
	Maxmod	4.223	0	1.277
	Multimod	0	0	0
	Rastrigin	0	0	0.019
	Rosenbrock	109.46	73.991	109.732
	Schaffer	0.481	0	0.156
	Schwefel	0.006	0	34.061
	Sphere	0	0	0.003
	Sumsquares	0	0	0.405

maximum number of iterations is set to a higher integer, the amount of decrease on the mutation probability is small and the effect of the mutation operator is higher at the early stages of the genetic search. However, as the maximum number of generations is small, the decrease on the mutation probability increases and the algorithm suddenly enters a *fine-tuning* stage. Finally, the *fine-tuned* solution is probably a local optimum because the algorithm may not discover the different parts of the search space. Determining a good start for the dynamic mutation probability can be the subject of a further research.

7. Conclusion

Machine-coded genetic operators are applied on the byte representation of candidate solutions on the computer memory. Byte-based crossover operators recombines real vectors to generate offspring on a way similar to their binary counterpars using the *geno-type*, namely, bytes. The original byte-based mutation operator also mimics its binary counterpart by altering a single byte by ± 1. In this paper, we introduce the random byte-based mutation operator in which a single byte is altered randomly in the range of a byte in computer memory. We also introduce the dynamic byte-based mutation operators in which the mutation probability is dynamically decreased by iterations to perform the tasks of *exploration* and *exploitation* in a more efficient way. Byte-based dynamic mutation operators search the different areas of the search space when the mutation probability is high and the task is turned into performing a *local fine-tuning* by decreasing the mutation probability by iterations. Since the effect of the byte-based mutation operators depends on the location of the mutated byte, a *long jump* or a *local fine-tuning* can be carried out without setting a user-defined option such as

a parameter of a probability density function. When a moderate size of population is used and the maximum number of iterations is enough to converge, byte-based operators obtain very precise results even with the decision variables with wider domains because of the constant-sized and compact representation of real numbers in range of roughly $(-10^{308}, 10^{308})$ with 8-bytes long `double` type.
A simulation study is performed on a set of well-known test functions that are used for comparing performances of evolutionary optimization algorithms in recent studies. Simulations are ran on the `R` system with `GA` and `mcga` packages installed. The package `mcga` implements the byte-based operators and freely avaliable for downloading. Results of the simulation study show that the byte-based operators performs well when they are combined for reproduction tasks. The performance increases when the initial mutation probability is set to a higher value and then dynamically decreased, and finally it is set to 0 at last generation.
The developed byte-based operators are based on the `double` type which is 8 bytes long. Alternatively 4 bytes long `float` and 16 bytes long `long double` data types can be used for representing floating-point numbers. Performances of developed operators on these data types are not investigated and this can be a subject of a further study.

References

[1] Ernesto P Adorio and U Diliman. Mvf-multivariate test functions library in c for unconstrained global optimization. *Quezon City, Metro Manila, Philippines*, 2005.

[2] M Montaz Ali, Charoenchai Khompatraporn, and Zelda B Zabinsky. A numerical evaluation of several stochastic algorithms on selected continuous global optimization test problems. *Journal of Global Optimization*, 31(4):635–672, 2005.

[3] Rachid Chelouah and Patrick Siarry. Tabu search applied to global optimization. *European journal of operational research*, 123(2):256–270, 2000.

[4] Sung-Bae Cho and Joo-Young Lee. A human-oriented image retrieval system using interactive genetic algorithm. *Systems, Man and Cybernetics, Part A: systems and humans, IEEE Transactions on*, 32(3):452–458, 2002.

[5] Kalyanmoy Deb. *Multi-objective optimization using evolutionary algorithms*, volume 16. John Wiley & Sons, 2001.

[6] Kalyanmoy Deb and Ram Bhushan Agrawal. Simulated binary crossover for continuous search space. *Complex systems*, 9(2):115–148, 1995.

[7] Kalyanmoy Deb and A Kumar. Real-coded genetic algorithms with simulated-binary crossover: Studies on multi-modal and multi-objective problems. *Complex systems*, 9(6):431–454, 1995.

[8] Larry J Eshelman. chapter real-coded genetic algorithms and interval-schemata. *Foundations of genetic algorithms*, 2:187–202, 1993.

[9] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48, 1991.

[10] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[11] John J Grefenstette. Optimization of control parameters for genetic algorithms. *Systems, Man and Cybernetics, IEEE Transactions on*, 16(1):122–128, 1986.

[12] Nikolaus Hansen and Stefan Kern. Evaluating the cma evolution strategy on multimodal test functions. In *Parallel problem solving from nature-PPSN VIII*, pages 282–291. Springer, 2004.

[13] Francisco Herrera and Manuel Lozano. Gradual distributed real-coded genetic algorithms. *Evolutionary Computation, IEEE Transactions on*, 4(1):43–63, 2000.

[14] Francisco Herrera, Manuel Lozano, and Ana M Sánchez. A taxonomy for the crossover operator for real-coded genetic algorithms: An experimental study. *International Journal of Intelligent Systems*, 18(3):309–338, 2003.

[15] Francisco Herrera, Manuel Lozano, and Jose L. Verdegay. Tackling real-coded genetic algorithms: Operators and tools for behavioural analysis. *Artificial intelligence review*, 12(4):265–319, 1998.

[16] John H Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. U Michigan Press, 1975.

[17] Hee-Su Kim and Sung-Bae Cho. Application of interactive genetic algorithm to fashion design. *Engineering applications of artificial intelligence*, 13(6):635–644, 2000.

[18] Alex Kosorukoff. Human based genetic algorithm. In *Systems, Man, and Cybernetics, 2001 IEEE International Conference on*, volume 5, pages 3464–3469. IEEE, 2001.

[19] Fernando G Lobo and David E Goldberg. The parameter-less genetic algorithm in practice. *Information Sciences*, 167(1):217–232, 2004.

[20] Zbigniew Michalewicz. *Genetic algorithms+ data structures= evolution programs*. Springer Science & Business Media, 2013.

[21] Tatsuya Nomura and Tsutomu Miyoshi. Numerical coding and unfair average crossover in ga for fuzzy rule extraction in dynamic environments. In *Fuzzy Logic, Neural Networks, and Evolutionary Computation*, pages 55–72. Springer, 1995.

[22] Anne L Olsen. Penalty functions and the knapsack problem. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pages 554–558. IEEE, 1994.

[23] Petr Pospichal, Josef Schwarz, and Jiri Jaros. Parallel genetic algorithm solving 0/1 knapsack problem running on the gpu. In *16th International Conference on Soft Computing MENDEL*, volume 2010, pages 64–70, 2010.

[24] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2016.

[25] Nicholas J Radcliffe. Equivalence class analysis of genetic algorithms. *Complex systems*, 5(2):183–205, 1991.

[26] Shahryar Rahnamayan, Hamid R Tizhoosh, and Magdy Salama. Opposition-based differential evolution. *Evolutionary Computation, IEEE Transactions on*, 12(1):64–79, 2008.

[27] Shahryar Rahnamayan, Hamid R Tizhoosh, and Magdy MA Salama. A novel population initialization method for accelerating evolutionary algorithms. *Computers & Mathematics with Applications*, 53(10):1605–1614, 2007.

[28] Mehmet Hakan Satman. Machine coded genetic algorithms for real parameter optimization problems. *Gazi University Journal of Science*, 26(1):85–95, 2013.

[29] Mehmet Hakan Satman and Emre Akadal. Arima forecasting as a genetic inheritance operator in floating-point genetic algorithms. *Journal of Mathematical and Computational Science*, 6(3):360, 2016.

[30] Luca Scrucca. Ga: A package for genetic algorithms in r. *Journal of Statistical Software*, 53(1):1–37, 2013.

[31] D. Stevenson. A proposed standard for binary floating-point arithmetic. *Computer*, 14(3):51–62, March 1981.

[32] Rainer Storn and Kenneth Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359, 1997.

[33] Aram Ter-Sarkisov and Stephen Marsland. K-bit-swap: a new operator for real-coded evolutionary algorithms. *Soft Computing*, pages 1–10, 2016.

[34] Jakob Vesterstrøm and Rene Thomsen. A comparative study of differential evolution, particle swarm optimization, and evolutionary algorithms on numerical benchmark problems. In *Evolutionary Computation, 2004. CEC2004. Congress on*, volume 2, pages 1980–1987. IEEE, 2004.

[35] Alden H Wright. Genetic algorithms for real parameter optimization. *Foundations of genetic algorithms*, 1:205–218, 1991.