mcga: R Implementation of the Machine-coded Genetic Algorithms

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

[1] *Istanbul University, Department of Econometrics, Istanbul, Turkey*

[2] *Istanbul University, Department of Informatics, Istanbul, Turkey*

Research Paper

Abstract

Genetic Algorithms (*GA*s) are global optimization and search algorithms that mimic the natural selection and genetic processes. Floating-point *GA*s (*FPGA*s) are other type of *GA*s which directly operate on real-vectors without requiring a *geno-type – pheno-type* distinction and encoding-decoding processes. However, the classical crossover and mutation operators are not directly applicable on the real-vectors. As a result of this, new types of genetic operators are developed for *FPGA*s. Machine-coded *GA*s (*MCGA*s) apply byte-based genetic operators on the byte representations of the candidate solutions. This natural encoding scheme makes classical crossover operators applicable on the real-vectors. Addition to this, *MCGA*s report more precise results in larger domains of decision variables. Mutation operation on the byte representations of variables have also similar effects with its binary counterpart. The `R` package `mcga` defines *plug-in* versions of byte-based operators that can be integrated with the recently developed function `ga` in package `GA`. Low level utility functions are written in `C++` and wrapped with the `Rcpp` and `.Call` interface of `R`. Advantages and disadvantages of using byte-based operators are discussed and demonstrated on some univariate and multivariate optimization problems.

Keywords: genetic algorithms; real-valued optimization; R

2010 Mathematics Subject Classification: 65K10; 93B40; 80M50

1 Introduction

Genetic algorithms (*GA*s) are search and optimization methods that mimic the natural selection and principals of genetics. Since a *GA* does not require the objective function to be continuous and

**Corresponding author: E-mail: mhsatman@istanbul.edu.tr*

differentiable, it is directly applicable in several optimization problems. This property makes the *GA* a standard tool on a wide range of optimization problems with real, integer or binary decision variables. Providing the user-defined fitness function, many parts of a *GA* do not require any modifications as *GAs* are problem independent.

The simple *GA* (*SGA*) is founded by [14] and developed by his students. Invention of *SGA* indisputably opens new branches on the subject of global optimization. In some point of view, *SGA*s or *GA*s in general, are not the sub-branches of artificial intelligence as they search the global optima of a function in a way similar to *Newton*'s or *Hooke-Jeeve*'s methods [15] in a narrow sense. In detail, *SGA*s work with neither numbers nor derivatives and the working principle of *SGA* is *symbolic* rather than *numeric*. With this property, a *SGA* implicitly performs a *symbolic* pattern-matching process which is based on the consequences of the *Schemata* theorem. Distinction of the properties of being an optimization method and a pattern-matching tool is more blurry in *Genetic Programming* which is the general form of *Artificial Feed-forward Neural Networks* in which the inputs, outputs, network weights, number of layers, and type of the activation functions are determined within a genetic search [28, 3].

SGA encodes and decodes candidate solutions using the binary coding scheme, selects parents for recombination using the Roulette wheel selection or proportional selection, combines parents to generate offspring using the one-point crossover operator and flips some bits randomly for creating small or big changes using the mutation operator [9]. The Schemata Theorem proves that an increment is expected on the average fitness of a population of candidate solutions by iterations. Note that, the population size and other parameters of genetic operators should be selected carefully to ensure diversity of population, otherwise, a *GA* can stuck on a local optimum and a further improvement may never take place as it is expected mathematically [26].

Floating-Point (Real-Valued) Genetic Algorithms (*FPGA*s) are other members of the *GA* family [29]. In *FPGA*s, candidate solutions are not encoded-decoded using a binary alphabet and genetic operators are directly applied on vectors of real numbers. Since the classical one-point, two-point and uniform crossover operators are defined on the bit strings, new type of operators are developed for the *FPGA*s. The binary mutation operator also loses its functionality on real vectors as flipping a real number does not make sense. Crossover operators that developed for *FPGA*s generate offspring by combining parents numerically. The mutation operation is applied on real elements of vectors by adding some random values using a constant distribution or values depending on the current number of generations [18, 5, 4, 11, 12].

When decision variables of an optimization problem are binary, the binary coding scheme serves a natural way of representing variables as the alphabet $\alpha\beta = \{A, T, C, G\}$ represents the human DNA. If a variable is in type of integer, more than one bits can be combined to form up its value in a wide range. Representing real numbers in binary format is also possible in a discrete manner. Machine-coded genetic algorithms (*MCGA*s), uses the byte representations of candidate solutions as they are stored in the computer memory [22, 24]. Real numbers are stored using fixed-length bytes in the computer memory using a standard [27] and this coding scheme can be considered as a natural way of representing real numbers as chromosomes in genetic algorithms. The classical crossover operators are directly applicable on the byte arrays without any modification. Suppose that two `double` typed values C_1 and C_2 are stored in the computer memory as

$$C_1 = [c_{11}, c_{12}, ..., c_{1s}]$$

and

$$C_2 = [c_{21}, c_{22}, ..., c_{2s}]$$

where $0 \leq c_{ij} \leq 255$ is the jth byte of the ith chromosome, s is the `sizeof(double)` which is replaced with byte-length of a `double` typed variable by a `C` or a `C++` compiler, $i = 1, 2$ and $j = 1, 2, ..., s$. Note that if a candidate solution consists on p variables then it is encoded using `p * sizeof(double)`

bytes. In this form, the classical one-point, two-point or uniform crossover operators are directly applicable on C_1 and C_2.

The mutation operator in *MCGA* is also applied on the byte representations on the candidate solutions. If a randomly selected byte is subject to be mutated, then the byte value is increased or decreased by 1 with the probability of $1/2$. If the mutated byte has a negative value then it is replaced by 255. Similarly, if the mutated byte has value larger then 255, it is replaced by 0. Since the effect of this mutation operator depends on the location of the mutated byte, it has a similar behaviour with its classical counterpart.

In this paper, the `R` [20] package `mcga` [24] is introduced with examples. The aim of writing the package `mcga` is to make the recently published algorithm available to researcher for those use `R` in their research area. In Section 2, we introduce the package and its components. In Section 3, we show how a performance improvement can be achieved by using the byte-based genetic operators. In Section 4, we discuss some advantages and disadvantages of using the byte-based operators instead of their *FPGA* counterparts. Finally, in Section 5, we conclude.

2 The `mcga` package

The recently developed `R` package `GA` implements many types of selection, crossover and mutation operators in an object-oriented way using `S4` classes [25]. This type of design makes integrating new type of genetic operators possible without modifying the other parts of the code. The `mcga` package implements the byte-based crossover operators with functions `byte_crossover`, `byte_crossover_1p`, `byte_crossover_2p` and byte-based mutation operators with functions `byte_mutation`, `byte_mutation_dynamic`, `byte_mutation_random`, and `byte_mutation_random_dynamic` which can be directly integrated with the package `GA`. These functions call other `C++` functions to perform byte operations and they are wrapped to interact with the `R` functions. Since `R` has a useful interface to interact with external libraries, it is possible to call code that written in other languages such as `C`, `C++`, `Java` and `Fortran`. `mcga` calls external functions that are written in `C++` using the `Rcpp` package [6]. `Rcpp` simplifies using the `.Call` interface and reduces the development time dramatically with its clean interface and easy wrapping of the lower-level `R` internal code. `mcga` performs low-level byte operations painlessly thanks to the `.Call` interface of `R` and the package `Rcpp`.

Note that the package `mcga` hosts some `C++` based utility functions for developing further functionality and they are called by the byte-based genetic operator functions. These functions do not assume a machine specific real-number representation system including the byte-length and byte-order of type `double`. Utility functions are mostly based on type conversions of floating-point numbers and byte arrays. Before the version 3.0, the older versions of `mcga` reported in [22] uses the `.Call` interface directly, without using the `Rcpp` package, and only implements the original byte-based operators. The version 3.0 implements the original and some new byte-based operators (random and dynamic mutations, other types of crossover operators, etc) as *plug-in*s which are integrated with the package `GA`. This implementation serves a modular use of byte-based operators on the recently developed package `GA`, permits combining the byte-based operators with the other *FPGA* operators, allows comparisons of results, etc. Altough the implemented functions of the older version is available in the package, the researchers are encouraged to use new ones as described in the remainder of this paper.

2.1 Utility functions

Firstly, the `SizeOfDouble` function reports the byte-length of a `double` typed value in computer memory as shown below:

```
R> SizeOfDouble()
```

```
[1] 8
```

Now, suppose that we define two candidate solutions as shown below:

```
R> cand1 <- c(6.7665, 9.8787766)
R> cand2 <- c(21.343434, 15.455439912)
```

The variables of vectors `cand1` and `cand2` are represented in the computer memory as can be shown using the function `DoubleVectorToBytes`:

```
R> bytes1 <- DoubleVectorToBytes(cand1)
R> bytes2 <- DoubleVectorToBytes(cand2)
R> bytes1

[1]  55 137  65  96 229  16  27  64 235 250 170   1 239 193  35  64

R> bytes2

 [1] 159  85 102  74 235  87  53  64 142 170 142 107  47 233  46  64
```

These byte arrays can be converted back to the corresponding real numbers using the utility function `ByteVectorToDoubles`:

```
R> ByteVectorToDoubles(bytes1)

[1] 6.766500 9.878777

R> ByteVectorToDoubles(bytes2)

[1] 21.34343 15.45544
```

Note that it seems the transformation does not result the same real numbers because of the printing functionality of `R`. It can be proved that the results are the same using the `format` function:

```
R> format(ByteVectorToDoubles(bytes2), digits = 14)

[1] "21.343434000" "15.455439912"
```

Besides the vector versions of conversion functions, `DoubleToBytes` and `BytesToDouble` utility functions can be used for converting a `double` typed value to byte vectors and converting a byte vector to a single `double` typed value, respectively:

```
R> format(BytesToDouble(DoubleToBytes(pi)), digits = 20)

[1] "3.141592653589793116"
```

2.2 Genetic operators

It is known that the one-point crossover operator in *SGA*s simply cut two candidate solutions using a cut-point k such that $1 \leq k \leq s-1$ where s is the chromosome length. The `OnePointCrossOver` function takes two byte arrays and a cut-point as arguments and returns a `list` object of generated offspring:

```
R> off <- OnePointCrossOver(bytes1, bytes2, cutpoint = 6)
R> off

[[1]]
 [1]  55 137  65  96 229  16  53  64 142 170 142 107  47 233  46  64

[[2]]
 [1] 159  85 102  74 235  87  27  64 235 250 170   1 239 193  35  64
```

The byte vectors of generated offspring can be converted back to real vectors as shown below:

```
R> ByteVectorToDoubles(off[[1]])

[1] 21.06600 15.45544

R> ByteVectorToDoubles(off[[2]])

[1] 6.835858 9.878777
```

Similarly, the `R` wrapper function `TwoPointCrossOver` applies the two-point crossover operator on byte representations of chromosomes using 2 cut-points. In the example below, cut-points are selected as 4 and 8:

```
R> off <- TwoPointCrossOver(bytes1, bytes2, cutpoint1 = 4, cutpoint2 = 8)
R> off

[[1]]
 [1]  55 137  65  96 235  87  53  64 235 250 170   1 239 193  35  64

[[2]]
 [1] 159  85 102  74 229  16  27  64 142 170 142 107  47 233  46  64

R> ByteVectorToDoubles(off[[1]])

[1] 21.343435  9.878777

R> ByteVectorToDoubles(off[[2]])

[1]  6.76650 15.45544
```

Uniform crossover in *MCGA*s draws a random number using a Uniform distribution with parameters 0 and 1 for each single byte. If the selected number is less than $1/2$ then the current byte of offspring is set from the first parent, otherwise, the current byte value is set from the second parent. The Uniform crossover operator of `mcga` is defined in the `UniformCrossOver` function and it is applied on the byte representations of candidate solutions:

```
R> set.seed(123)
R> off <- UniformCrossOver(bytes1, bytes2)
R> off

[[1]]
 [1]  55  85  65  74 235  16  53  64 142 250 142   1  47 233  35  64

[[2]]
 [1] 159 137 102  96 229  87  27  64 235 170 170 107 239 193  46  64

R> ByteVectorToDoubles(off[[1]])

[1] 21.066090  9.955437

> ByteVectorToDoubles(off[[2]])

[1]  6.835836 15.378780
```

Note that the package `mcga` includes some other helper functions for applying byte-based crossover operators on the double vectors without manual byte-conversation. These functions are defined as

- `OnePointCrossOverOnDoublesUsingBytes(d1, d2, cutpoint)`,
- `TwoPointCrossOverOnDoublesUsingBytes(d1, d2, cutpoint1, cutpoint2)`,
- `UniformCrossOverOnDoublesUsingBytes(d1, d2)`;

where `d1` and `d2` are vectors of real numbers, `cutpoint1` is the cut-point for one-point crossover, `cutpoint1` and `cutpoint2` are the cut-points for two-point crossover, respectively. In the code below, real vectors `cand1` and `cand2` are crossed-over using the byte-based uniform crossover operator:

```
R> set.seed(123)
R> UniformCrossOverOnDoublesUsingBytes(cand1, cand2)

[[1]]
[1] 21.066090  9.955437

[[2]]
[1]  6.835836 15.378780
```

`ByteCodeMutation` randomly changes bytes of a given vector `bytes` with the mutation probability `pmutation`:

```
R> set.seed(123)
R> bytes1

[1]  55 137  65  96 229  16  27  64 235 250 170   1 239 193  35  64

R> bytes1.new <- ByteCodeMutation(bytes = bytes1, pmutation = 0.15)
R> bytes1.new

[1]  55 137  65  96 229  15  27  64 235 250 170   1 239 193  34  64
```

```
> ByteVectorToDoubles(bytes1.new)

[1] 6.765523 9.378777
```

As shown in code above, changes on 6th and 15th bytes affects the values of vector `cand1` in different scales. As the location of mutated byte changes, the effect of the mutation changes. With this behaviour, byte-based mutation operator mimics the behaviour of the binary mutation operator, indeed, they are similar. Note that the mutation operation can be directly applied on real vectors using the utility function `ByteCodeMutationUsingDoubles`.

`ByteCodeMutationUsingDoubles` function implements the original byte-based mutation algorithm in `C++` language reported by [22]. As it is mentioned before, the original mutation operator increase or decrease a byte value with probability $1/2$ if the byte is selected for mutation. This type of mutation may require more generations or a bigger size of population to achieve the optimum. Instead of `ByteCodeMutationUsingDoubles`, `ByteCodeMutationUsingDoublesRandom` can be selected as the mutation operator for searching new areas of the search space more quickly. Differently, the function `ByteCodeMutation UsingDoublesRandom` randomly determines a value of a byte using a Uniform(0,255) distribution if the byte is selected for mutation operation. Sudden change on a byte value may cause a drastic change on the whole variable and this effect is generally larger than increasing or decreasing a byte value by 1.

In the code below, a candidate solution `cand1` is mutated using two mutation algorithms with the mutation probability of 0.25. `cand.new.1` is the mutated solution using the original mutation operator. The change on `cand.new.2` is larger as it is mutated by the random byte-based operator.

```
R> set.seed(123)
R> cand1

[1] 6.766500 9.878777

R> cand.new.1 <- ByteCodeMutationUsingDoubles(cand1, 0.25)
R> cand.new.1

[1] 6.765523 9.378777

R> set.seed(123)
R> cand.new.2 <- ByteCodeMutationUsingDoublesRandom(cand1, 0.25)
R> cand.new.2

[1] 6.812398 3.469694
```

2.3 Integration with `GA` package

Since the utility functions mentioned before are `R` wrappers, they are not generally used by the researcher. Addition to this, `mcga` defines *plug-in* versions of the genetic operators which can be integrated with the recently developed `ga` function which is defined in the package `GA`. `mcga` defines

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

- byte_mutation_random,
- byte_mutation_mutation_dynamic;

functions for byte-based versions of genetic operators which can be assigned to crossover= and mutation= parameters as arguments in GA::ga function.

The byte_crossover is defined as shown below:

```
byte_crossover <- function (object, parents, ...){
  parents <- object@population[parents, , drop = FALSE]
  ch1 <- parents[1, ]
  ch2 <- parents[2, ]

  offs <- UniformCrossOverOnDoublesUsingBytes(ch1, ch2)
  off1 <- offs[[1]]
  off2 <- offs[[2]]
  EnsureBounds(off1, object@min, object@max)
  EnsureBounds(off2, object@min, object@max)

  out <- list(children = rbind(off1, off2), fitness = rep(NA, 2))
  return(out)
}
```

The byte_crossover function takes ga-class object object and a vector of selected indices parents as arguments. Within the argument ..., the ga function can pass additional arguments. byte_crossover function simply extracts the parents and pass them to C++ based UniformCrossOverOnDoublesUsingBytes function.

The function EnsureBounds is an other C++ based function which controls the bounds of decision variables. If a variable of the generated content does not fall among the lower and the upper bounds then it is randomly determined. Since the function EnsureBounds takes its arguments as reference rather than value, the content of the passed arguments is changed. This kind of implementation causes many speed improvements by avoiding copying of big data structures. Suppose that the boundary of variables is defined as $0 \leq x_1, x_2 \leq 1$. An example of altering a candidate solution using the function EnsureBounds is shown below:

```
R> set.seed(123)
R> cand <- c(1.0, 2.0)
R> bounds.lower <- c(0, 0)
R> bounds.upper <- c(1, 1)
R> EnsureBounds(cand, bounds.lower, bounds.upper)
R> cand

[1] 1.0000000 0.2875775
```

The byte_mutation function is the implementation of the byte-based mutation operator and it is defined to interact with the function GA::ga as shown below:

By contrast, this kind of implementation is susceptible to side effects, in terms of *functional programming*. However, since the vectors are local copies, there are not any side-effects in these functions.

```
byte_mutation <- function (object, parent, ...){
  mutate <- as.vector(object@population[parent, ])
  mutate <- ByteCodeMutationUsingDoubles(mutate, object@pmutation)
  EnsureBounds(mutate, object@min, object@max)
  return(mutate)
}
```

The `byte_mutation` function takes `ga-class` object `object` and a selected index `parent` as arguments. The byte-based mutation operator is applied using the probability `object@pmutation` which is passed from the main genetic search routine. Similarly, the `EnsureBounds` function performs a bound-control using the vectors `object@min` and `object@max` which are passed from the `ga-class`-typed object. Through these implementations, the byte-based genetic operators can easly be integrated with the recently developed and tested package `GA`.

The `mcga` package also includes a wrapper method `mcga2` which encapsulates a `GA::ga` search using new default settings. The researcher can perform a *MCGA* search in two ways. In the first setting, the argument `crossover=` of `ga` can be set to one of the crossover operators `byte_crossover`, `byte_crossover_1p`, and `byte_crossover_2p` for uniform, one-point, and two-point crossover, respectively. Similarly, the argument `mutation=` of `ga` can be set to one of the mutation operators `byte_mutation`, `byte_mutation_dynamic`, `byte_mutation_random`, and `byte_mutation_mutation_dynamic` for different kind of byte-based mutation operators. In the second case, the `mcga2` function can be called directly. Note that the second case includes an unnecessary function wrapping, the first case is more convenient to use and it is faster.

3 Performances of the byte-based operators

Suppose that the n-parameters *Bohachevsky* function is defined as

```
R> Bohachevsky <- function(x){
+    mysum <- 0
+    for (i in 1:(length(x) - 1)) {
+      mysum <- mysum + (x[i] ^ 2 + 2 * (x[i + 1]) ^ 2 -
+        0.3 * cos(3 * pi * x[i]) -
+        0.4 * cos(4 * pi * x[i + 1]) + 0.7)
+    }
+    return (-mysum)
+  }
```

for $-50 \leq x_i \leq 50$ and $i = 1, 2, ..., n$. This function has a global minimum of 0 for all $x_i = 0$. Now suppose that the objective is to find the minimum of *Bohachevsky* function with $n = 50$ parameters. In the `R` definition of function, `-mysum` is returned to change the type of the optimization from minimization to maximization as the function `GA::ga` assumes.

```
R> n <- 50
R> bounds.min <- rep(-50, n)
R> bounds.max <- rep(50, n)
```

The code below performs a genetic search using the byte-based crossover and mutation operators with other arguments are kept as default.

```
R> set.seed(123)
R> time1 <- system.time(
+    result.mcga <- ga(type = "real-valued",
+      crossover = byte_crossover,
+      mutation = byte_mutation, min = bounds.min,
+      max = bounds.max, popSize = 100,
+      maxiter = 200, fitness = Bohachevsky)
+  )
R> time1

   user  system elapsed
 10.964   0.056  11.107

R> Bohachevsky(result.mcga@solution)

[1] -6.909616e-06
```

As shown in code above, the search takes 11.107 seconds in total and the reported minimum is $-6.909616e-06$ which is almost equal to 0. The same optimization process can be performed using the `GA::ga`'s standard configuration without the byte-based operators as shown below:

```
R> set.seed(123)
R> time2 <- system.time(
+    result.ga <- ga(type = "real-valued", min = bounds.min,
+      max = bounds.max, popSize = 100,
+      maxiter = 200, fitness = Bohachevsky)
+  )
R> time2

   user  system elapsed
 10.532   0.028  10.647

R> Bohachevsky(result.ga@solution)

[1] -496.8995
```

As shown above, standard `ga` call takes a little less computation time when it is compared to previous call. Addition to this, the standard configuration does not achieve the global optimum and needs more iterations to get more closer to 0. Note that, it is directly related to the default crossover and mutation operators that are selected in the `GA::ga` definition and a further work is required to compare performances of byte-based operators with other type of operators. The current `GA::ga` implementation uses the `gareal_laCrossover` and `gareal_raMutation` functions for crossover and mutation, respectively. Instead of using these functions, the `gareal_waCrossover` and `nraMutation` can be selected. A genetic search based on these operators produces the results as shown below:

```
R> set.seed(123)
R> time3 <- system.time(
+    result.ga2 <- ga(type = "real-valued", min = bounds.min,
+      max = bounds.max, popSize = 100, maxiter = 200,
+      fitness = Bohachevsky,
```

Performed on an i5 computer with 4 GBs of RAM and 64-Bits `R` installation on a Linux OS.

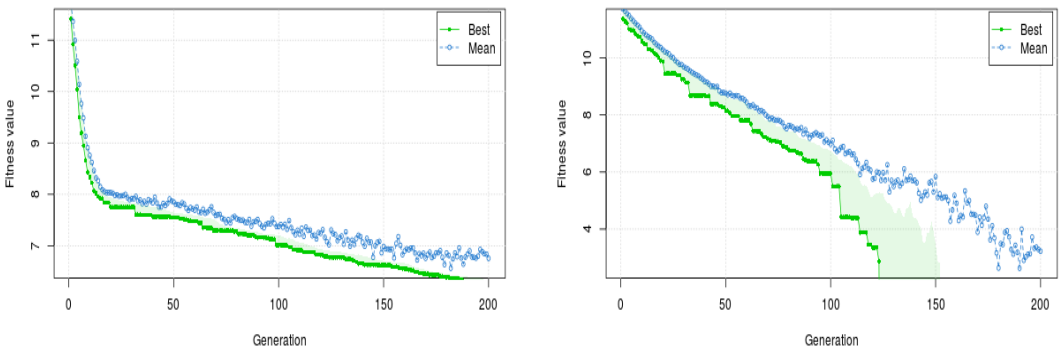
```
+       crossover = gareal_waCrossover,
+       mutation = gareal_nraMutation)
+  )
R> time3

   user  system elapsed
 10.492   0.020  10.603

R> Bohachevsky(result.ga2@solution)

[1] -1893.166
```

As shown in the code above, within a shorter elapsed time, a worse solution is reported by the function and this search needs more time to achieve the global minimum. The byte-based operators perform better on this example with a small time difference. The convergence of the *MCGA* in this example is interesting and a comparison is given in Figure 1. In Figure 1(a) the logarithm of average and best fitness values are reported for the standard `GA::ga` call by iterations. The curves are concave upwards and the convergence performance is low. In Figure 1(b) it is shown that the curves are concave downwards and the convergence performance is high for the byte-based operators. In other words, the amount of improvement is increased faster than the amount of increase of the generations, that is, byte-based search converges more quickly and needs less generations to achieve the global minimum in this example.



(a) (b)

Figure 1: Logarithm of average and best fitness values by iterations: (a) `GA::ga` search with default parameters. (b) `GA::ga` search with `byte_crossover` and `byte_mutation` options.

Table 1 shows the results of a mini-simulation study which is designed for comparing effects of some crossover and mutation operators on the 75-variables Bohachevsky function. In the simulation study, minimum of the Bohachevsky function is searched 200 times with real or byte mutation operators paired with a crossover operator. The byte-based crossover operator, Simulated Binary Crossover (SBX) [4, 5], Flat Crossover [12, 19], Arithmetic Crossover [17], BLX$-\alpha$ Crossover [7, 11], Linear

Crossover [29, 12], and Unfair Crossover [18] operators are selected for the comparison. Population size, maximum number of iterations, crossover probability, mutation probability, and number of elitism parameters are constant and set to 100, 200, 0.80, 0.10, and 5, respectively.

Table 1: Average fitness values for $p = 75$

	Byte	SBX	Flat	Arithmetic	BLX	Linear	Unfair
Byte Mutation	5.116	747.24	589.389	697.32	2360.041	0	1.348
Real Mutation	5111.868	1308.588	1690.507	2049.271	2178.298	0	0.009

It is shown in Table 1 that the byte-based mutation operator results lower or equal function values except for the Unfair crossover. GA optimizations with the Linear crossover operator is plugged obtain the global minimum whatever the mutation operator is selected. The byte-based crossover operator obtains better results than the results of SBX, Flat, Arithmetic, and BLX$-\alpha$ crossover operators. The success of the byte-based crossover operator is increased in the case of the byte-based mutation operator is selected as the mutation operator. Despite the dynamic byte-based mutation operator yields the global optimum in average, it is not included in the mini-simulation study as it is differs in terms of updating the mutation probability by generations and it is mentioned in Section 4.

4 Advantages and disadvantages of the byte-based operators

Using the byte-based genetic operators have many advantages and disadvantages which should be taken into account by the researcher. However, selecting the right population size, crossover and mutation operators and their probabilities turn disadvantages to advantages. Since the byte-based operators are applied on candidate solutions that are encoded using a higher cardinality of an alphabet, larger populations and more generations are required to ensure diversity of population and convergence. When the best solution is converged to global optimum, the byte-based operators perform a local *fine-tuning* in higher detail and precision. In this section, we discuss these issues.

4.1 Higher cardinality of the byte alphabet

In *SGA*s, the initial population should contain enough information about the search space to get rid of getting stuck on a local optimum. *GA*s that use higher cardinality of alphabets require higher population sizes to represent several building blocks in the search space [21, 10]. The main disadvantage of *MCGA*s is that the byte representation of candidate solutions uses an alphabet of $\{0, 1, 2, ..., 255\}$. Since it is hard to ensure the diversity of population with a small set of candidate solutions, population sizes are generally higher in *MCGA*s.

On the other hand, the byte-based crossover operators directly operates on the basic elements of candidate solutions as in the way that the binary crossover operators do using the *geno-type*. In *FPGA*s, the *geno-type* – *pheno-type* distinction is no longer exists but the byte-based operators are applied on the *geno-type* even they are still used in a real-valued optimization without performing encoding-decoding processes.

Byte-based mutation operator also behaves like its binary counterpart. In binary mutation, a randomly selected bit is flipped, that is, it is changed to 1 if its present value is 0 and vice versa. The effect of the binary mutation operator depends on the location of the mutated bit. In byte-based mutation, effect of a mutated byte is different and depends on the location, namely, *locus* [8]. Now suppose that the variable `bytes.pi` holds the bytes of `pi`. The code below increases each single byte value by 1 and keeps others unchanged in a loop. As in shown in the output, mutation serves a *fine-tuning* task by altering a single byte in locations 1–6. A small change in 7th and 8th bytes, results

higher changes in *pheno-type* and as a result of this, the algorithm may explore different sides of the search space with small probabilities. In this example, the probability of diving into an other area is $1/8$ and the mutation operator performs a local *fine-tuning* with the probability of $7/8$.

```
R> bytes.pi <- DoubleToBytes(pi)
R> for (i in 1:length(bytes.pi)){
+    temp.bytes <- bytes.pi
+    temp.bytes[i] <- temp.bytes[i] + 1
+    print(format(BytesToDouble(temp.bytes), digits = 20))
+  }

[1] "3.1415926535897935601"
[1] "3.1415926535899068028"
[1] "3.1415926536188969465"
[1] "3.1415926610403737129"
[1] "3.1415945609384259285"
[1] "3.142080934839793116"
[1] "3.266592653589793116"
[1] "205887.41614566068165"
```

In classical mutation operators of *FPGA*s, the amount of the mutation or the mutation probability can be changed dynamically by generations. The mutation probability can be dynamically identified using the formula

$$Pm = 0.1 - 0.05\frac{t}{T}$$

where t is the current number of generations and T is the maximum number of generations [13]. Using this formula, P_m takes values in a range of $[0.05, 0.1]$. Defining the amount of the change is an other important issue. For local *fine-tuning*, the amount of mutation can be narrowed using

$$V_j^{new} = V_j^{old} + \mathcal{N}(0, \sigma^2(t))$$

where V_j^{old} is the current value of jth variable of a candidate solution, $\mathcal{N}(\mu, \sigma^2)$ is the normal distribution function with mean μ and variance σ^2, V_j^{new} is the mutated value, t is the current number of generations and $\sigma^2(t)$ is a decreasing function of t. Altering the mutation probability has still use in *MCGA*s but the local *fine-tuning* of the amount of the mutation is handled by the byte-based mutation operator.

In `mcga`, the functions `byte_mutation_dynamic` and `byte_mutation_mutation_dynamic` dynamically reduce the mutation probability using the formula

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

where t is the current number of generations, $P_m^{(t)}$ is the mutation probability in generation t, T is the maximum number of generations and P_m is the predefined mutation probability. Since *MCGA*s use an alphabet with higher cardinality, more mutations are needed to explore several zones of the search space. Decreasing the mutation probability by iterations switches the mission of the mutation operator from exploring to *fine-tuning*.

4.2 Precision of byte-based operators

In statistics, median m simply minimizes the optimization problem

$$\min \sum_{i}^{n} |x_i - m|$$

where x is a vector of x_i, n is the number of elements, and $|a|$ is the absolute value of a. Now suppose that the vector `x` is defined as shown below:

```
R> x <- c(-1.2, 0.5, pi, 4.9, 1000.35)
R> x

[1]   -1.200000    0.500000    3.141593    4.900000   1000.350000
```

It is clear that the sorted vector `x` has a median value of `pi`. Note that the printed version of `pi` is truncated and it is represented with more digits in the computer memory. The goal function of median is defined as below:

```
R> f <- function(m, x){
+    return(-sum(abs(x - m)))
+  }
```

A genetic search with byte-based operators `byte_crossover` and `byte_mutation_dynamic` can be performed as shown below. `byte_mutation_dynamic` simply increases or decreases a byte value with 1 using the decreasing mutation probability from 0.5 to 0.0 by iterations.

```
R> set.seed(123)
R> result.mcga <- GA::ga(type = "real-valued", fitness = f,
+    min = -2, max = 1001,
+    crossover = byte_crossover,
+    mutation = byte_mutation_dynamic,
+    maxiter = 300, popSize = 100,
+    x = x, pmutation = 0.5)
R> format(result.mcga@solution, digits = 15)

[1,] "3.14159265160561"
```

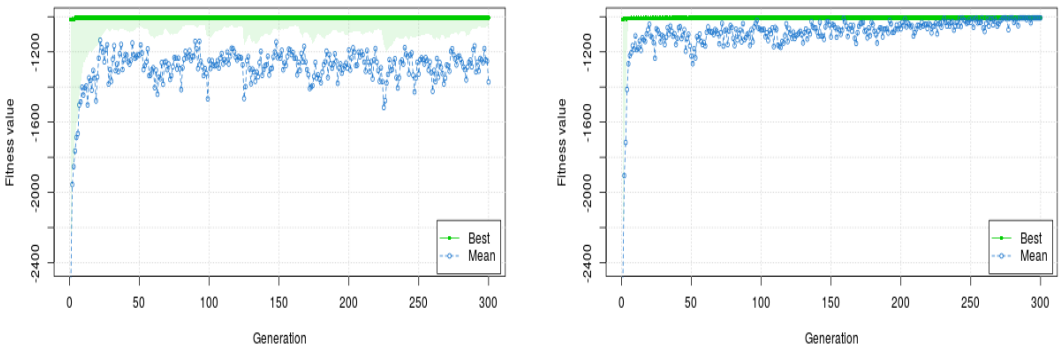
Since the `pi` value with 15 digits is 3.14159265358979, the GA search found a very precise solution up to 8 decimal digits with dynamic byte-based mutation operator. The same search with *FPGA*s can be performed using the code below:

```
R> set.seed(123)
R> result.ga <- GA::ga(type = "real-valued", fitness = f,
+    min = -2, max = 1001,
+    maxiter = 300, popSize = 100, x = x)
R> format(result.ga@solution, digits = 15)

[1,] "3.14177398383617"
```

The standard *FPGA* search reports the median as 3.14177398383617 and first 3 decimal digits are the same with the real median. In this example, the *MCGA* result is more precise as a result of *fine-tuning* capabilities of the byte-based mutation operator.

Figure 2(a) shows the average and best fitness values by iterations of `GA::ga` with standard configuration. The genetic search converges to median very quickly and average fitness increases by



(a) (b)

Figure 2: Average and best fitness values for searching median by iterations: (a) `GA::ga` search with default parameters. (b) `GA::ga` search with `byte_crossover` and `byte_mutation_dynamic` options.

iterations. In Figure 2(b) it is shown that the genetic search with `byte_crossover` and `byte_mutation_dynamic` converges quickly but the increase of average fitness still remains in further iterations. Finally, the result obtained by the byte-based operators is more precise and the median is calculated in great detail. In *FPGA*s, it is frequent that improving a result using a local search algorithm after applying a genetic search [16, 2, 1, 23]. Contrary, *MCGA*s perform a *fine-tuning* task after obtaining a good solution, that is, they accomplish both of the tasks of *exploration* and *exploitation* in great detail as a result of the byte-based representation system.

4.3 Optimization on larger domains

Since the memory of a digital computer is finite and the data is stored by using countable units, real numbers are represented in a discrete manner with a limited number of digits. Mathematically, a decision variable a can take more values in a range $[0, 10]$ than the variable b which is defined in range $[0, 5]$ if they are processed in a digital computer. If an optimization problem depends on the variable a rather than the variable b, then the search space is larger, that is, more computational effort may be required to achieve the global optimum. When a decision variable is defined as $-\infty \leq x \leq \infty$ then the search space includes many numbers depending on the compiler's or interpreter's capabilities or the system of real number representation. Whatever the range is, byte-based genetic operators use constant-length chromosomes for all problems in a specific machine with maximum level of precision.

Suppose that the `rastrigin` function is defined using the code below:

```
R> rastrigin <- function(x){
+    return (
+      -(10 * length(x) +
+      sum(x ^ 2 - 10 * cos(2 * pi * x)))
+    )
+  }
```

The number of variables in `rastrigin` function is determined by the `length(x)` and suppose that the global minimum of 50-variables version of this function is searched in the range of $-10^{30} \leq x_i \leq 10^{30}$ for $i = 1, 2, ..., 50$. The vectors `bound.min` and `bound.max` holds the values of bounds as shown below:

```
R> n <- 50
R> bound.min <- rep(-10 ^ 30, n)
R> bound.max <- rep( 10 ^ 30, n)
```

A genetic search with `byte_crossover_2p` and `byte_mutation_mutation_dynamic` operators can be performed for obtaining the global optimum of `rastrigin` using the code below:

```
R> set.seed(123)
R> result.mcga <- ga(type = "real-valued",
+    crossover = byte_crossover_2p,
+    mutation = byte_mutation_dynamic,
+    fitness = rastrigin,
+    min = bound.min,
+    max = bound.max,
+    popSize = 100,
+    maxiter = 1000,
+    pmutation = .5
+  )
R> summary(result.mcga)

+-----------------------------------+
|         Genetic Algorithm         |
+-----------------------------------+

GA settings:
Type                  =  real-valued
Population size       =  100
Number of generations =  1000
Elitism               =  5
Crossover probability =  0.8
Mutation probability  =  0.5
...
...

GA results:
Iterations             = 1000
Fitness function value = 0
Solution               =
     x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15 x16 x17 x18 x19 ...
[1,]  0  0  0  0  0  0  0  0  0   0   0   0   0   0   0   0   0   0   0 ...
     x50
[1,]   0
```

Since the `byte_mutation_mutation_dynamic` operator is applied, the mutation probability is set to 0.5 which can be considered as high. Note that the mutation probability is decreased by generations and the final value of the mutation probability is set to 0 in the last generation. In contrast to

byte_mutation_dynamic, the byte_mutation_mutation_dynamic changes byte values randomly in the range of $[0, 255]$ rather than adding ± 1 with equal probabilities. The solution is $x_i = 0.0$ for $i = 1, 2, ..., 50$ and the function takes the value of 0 which is the minimum.

Same genetic search is performed using the GA::ga function with standard operators and the same parameters of population size and maximum number of iterations. The code and the results are shown below:

```
R> set.seed(123)
R> result.ga <- ga(type = "real-valued",
+    fitness = rastrigin,
+    min = bound.min,
+    max = bound.max,
+    popSize = 100,
+    maxiter = 1000
+  )
R> summary(result.ga)

+-----------------------------------+
|         Genetic Algorithm         |
+-----------------------------------+

GA settings:
Type                  =  real-valued
Population size       =  100
Number of generations =  1000
Elitism               =  5
Crossover probability =  0.8
Mutation probability  =  0.1
...
...

GA results:
Iterations             = 1000
Fitness function value = -9.511589e+56
Solution               =
               x1           x2           x3            x4            x5
[1,] 1.330317e+27 5.266895e+27 6.227867e+27 -1.082765e+27 -3.411565e+27
                x6           x7         x8            x9           x10
[1,] -7.932537e+27 1.702512e+27 4.8688e+27 -1.647505e+27 -1.108371e+28
              x11          x12          x13          x14          x15
[1,] 3.506409e+27 3.256853e+27 7.858502e+26 -1.13104e+27 6.349995e+26
              x16           x17           x18           x19         ...
[1,] 3.112014e+27 -3.621393e+27 -5.246277e+27 -3.271371e+27         ...
               x50
[1,] -1.979981e+26
```

Since the range is very large and the function has many variables, *FPGA* with the standard configuration fails to converge and needs more iterations and/or a more scattered and larger initial population to get succeed. In contrast to standard *FPGA*, using byte-based operators has a significant effect on convergence. Note that, using the other types of byte-based operators, rather than the

`byte_crossover_2p` and the `byte_mutation_mutation_dynamic` results similar solutions and the quality of solutions does not depend on the type of operator but the type of representation of variables.

5 Conclusion

Byte-based crossover and mutation operators use the byte representations of decision variables in floating-point genetic algorithms. Using this coding scheme has many advantages. As the byte representation is the natural way of encoding and storing real numbers in computer memory, byte-based genetic operators are applied on the *geno-type* as in their classical counterparts. This feature makes classical crossover operators available for floating-point optimization. Since the effect of byte-based mutation operator depends on the location of the mutated byte, the byte-based mutation also behaves like its binary counterpart. As a result of this, byte-based mutation operator searches different sides of the search space or performs fine-tuning to adjust a solution near an optimum depending on the location of the mutated byte. Byte representation of variables are naturally map to $(-\infty, \infty)$ using a limited number of bytes with great detail and precision in a discrete manner. As a result of this, byte-based operators are suitable in the case of more precise solutions are required and/or the decision variables are defined on larger domains.

Since the byte-representation consists on a higher cardinality alphabet, the diversity of population may not be satisfied with smaller number of population sizes. Similarly, if the number of generations is not enough, a fine-tuning operation may not be performed and the solution may not converge. As a result of these, byte-based operators require more scattered initial populations on the search space for diversity and more generations to achieve the global optimum in great detail.

The `R` package `GA` provides a modular design for integrating new types of genetic operators in genetic optimization. The package `mcga` implements byte-based crossover and mutation operators of machine-coded genetic algorithms as *plug-in*s for `GA`. Byte-based genetic operators that are provided by `mcga` can be integrated with `GA::ga` function by simply setting `crossover=` and `mutation=` parameters to convenient function names as arguments. By virtue of the modular design and the utility functions, new type of byte-based genetic operators can be easly developed and integrated using the package `mcga`.

References

[1] Dan Adler. Genetic algorithms and simulated annealing: A marriage proposal. In *Neural Networks, 1993., IEEE International Conference on*, pages 1104–1109. IEEE, 1993.

[2] Thomas Bäck, David B Fogel, and Zbigniew Michalewicz. *Evolutionary Computation 1: Basic Algorithms and Operators*, volume 1. CRC Press, 2000.

[3] Wolfgang Banzhaf, Peter Nordin, Robert E Keller, and Frank D Francone. *Genetic Programming: An Introduction*, volume 270. Morgan Kaufmann San Francisco, 1998.

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

[5] 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.

[6] Dirk Eddelbuettel, Romain François, J Allaire, John Chambers, Douglas Bates, and Kevin Ushey. `Rcpp`: Seamless `R` and `C++` integration. *Journal of Statistical Software*, 40(8):1–18, 2011.

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

[8] Mitsuo Gen and Runwei Cheng. *Genetic Algorithms and Engineering Optimization*, volume 7. John Wiley & Sons, 2000.

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

[10] David E Goldberg. Real-coded genetic algorithms, virtual alphabets, and blocking. *Urbana*, 51:61801, 1990.

[11] 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.

[12] 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.

[13] Robert Hinterding, Zbigniew Michalewicz, and Agoston E Eiben. Adaptation in evolutionary computation: A survey. In *Evolutionary Computation, 1997., IEEE International Conference on*, pages 65–69. IEEE, 1997.

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

[15] Robert Hooke and To A Jeeves. “direct search” solution of numerical and statistical problems. *Journal of the ACM (JACM)*, 8(2):212–229, 1961.

[16] Shun-Fa Hwang and Rong-Song He. A hybrid real-parameter genetic algorithm for function optimization. *Advanced Engineering Informatics*, 20(1):7–21, 2006.

[17] Z Michalawicz. Genetic algorithms+ data structures= evolution programs, 1996.

[18] 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-Verlag, 1995.

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

[20] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2015.

[21] Colin R Reeves. Using genetic algorithms with small populations. In *ICGA*, volume 590, page 92, 1993.

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

[23] Mehmet Hakan Satman. Hybridization of floating-point genetic algorithms using hooke-jeeves algorithm as an intelligent mutation operator. *Journal of Mathematical and Computational Science*, 5(3):320, 2015.

[24] Mehmet Hakan Satman. *mcga: Machine Coded Genetic Algorithms for Real-valued Optimization Problems*, 2016. R package version 3.0.

[25] Luca Scrucca. GA: A package for genetic algorithms in R. *Journal of Statistical Software*, 53(4):1–37, 2013.

[26] Mandavilli Srinivas and Lalit M Patnaik. Genetic algorithms: A survey. *Computer*, 27(6):17–26, 1994.

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

[28] Ioannis P Vlahavas and Constantine D Spyropoulos. *Methods and Applications of Artificial Intelligence: Second Hellenic Conference on AI, SETN 2002 Thessaloniki, Greece, April 11–12, 2002 Proceedings*, volume 2308. Springer-Verlag, 2003.

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

©2016 Mehmet Hakan Satman & Emre Akadal; This is an Open Access article distributed under the terms of the Creative Commons Attribution License http://creativecommons.org/licenses/by/2.0, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.