Journal of Engineering Research and Applied Science
Available at www.journaleras.com
Volume 13 (1), June 2024, pp 2500-2514
ISSN 2147-3471 © 2024

An Improved Cellular Genetic Algorithm with Machine-Coded Operators for Real-Valued Optimisation Problems

Sevgi Akten Karakaya[1,*], Mehmet Hakan Satman[2]

[1] Istanbul University, Department of Informatics, Istanbul, Turkey
[2] Istanbul University, Department of Econometrics, Istanbul, Turkey

Accepted 7 April 2024

Abstract
This research introduces an enhanced cellular genetic algorithm employing machine-coded operators specifically tailored for real-valued optimization problems. The utilization of byte-based operators, designed to handle numerical data in a memory-efficient manner, distinguishes this approach. The study systematically evaluates the performance of the proposed algorithm across various test functions, including Ackley, Bohachevsky, Griewank, Holzman, Rastrigin, Rosenbrock, Schaffer, Chichinadze and Sphere. Simulation results reveal that the byte operators consistently outperform traditional counterparts, demonstrating the effectiveness of this novel approach in real-valued optimization scenarios.

Keywords: *Cellular Genetic Algorithm, Machine-Coded (Byte-Based) Operators, Real-Valued Optimization*

1. Introduction

In the realm of optimization, the fusion of evolutionary algorithms and cellular automata has yielded innovative approaches to solving complex problems. This paper introduces a novel paradigm in the form of an Improved Cellular Genetic Algorithm (ICGA) specifically tailored for addressing real-valued optimization problems. The ICGA is distinguished by its utilization of machine-coded operators, representing a departure from traditional cellular genetic algorithms that rely on predefined heuristics. In cellular genetic algorithms, which are a type of parallel genetic algorithms, selection and crossover operators are performed based on the neighborhood relationships of a candidate solution. The exploration capacity of solution space in cellular genetic algorithms is significantly higher compared to classical genetic algorithms (GAs) [1]. The proposed ICGA draws inspiration from the principles of cellular automata, incorporating an enhanced cellular structure that facilitates efficient information exchange among individuals within the population. This cellular arrangement promotes the exploration of solution spaces in a cooperative manner, enabling the algorithm to navigate complex landscapes with increased efficacy.

Real-valued optimization problems abound in diverse domains such as engineering, finance, and machine learning, demanding sophisticated solutions capable of navigating continuous solution spaces. Solving binary (0-1) problems with classical GAs is a common practice. Given that integers can be constructed using sequentially ordered bits, it is natural to extend this approach to optimization problems with integer decision variables through binary coding. However, representing real values becomes challenging due to the requirement of an infinite number of bits. To address this limitation, a finite sequence of bits can be employed to discretely map certain integers to real numbers. But as the length of the bit strings increases, a greater number of iterations is needed to achieve the global optimum [2]. 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 [3].

Machine-coded operators employ the byte representation of real values stored in computer memory. Compilers and interpreters utilize a standard to encode numbers, including floating-point types, by representing them as constant-size byte strings. Considering these byte strings as the genotype allows the application of traditional crossover and mutation operators to the candidate solutions, which are essentially vectors of real numbers [2, 4].

*Corresponding author: sevgiakten@gmail.com, sevgiakten@ogr.iu.edu.tr

Throughout this paper, we present a comprehensive analysis of the ICGA's performance across a range of benchmark real-valued optimization problems with simulation study. Comparative studies with traditional operators for real-valued problems and other state-of-the-art optimization techniques underscore the efficacy of our proposed approach. Moreover, we explore the interpretability and generalizability of the machine-coded operators, clarifying how they contribute to the algorithm's capacity to independently enhance its performance.

In summary, the Improved Cellular Genetic Algorithm with Machine-Coded Operators emerges as a promising solution for real-valued optimization challenges. By combining principles from cellular automata with machine-coded operators, our approach aims to extend the limits of optimization capabilities, providing a flexible tool for tackling the complexities of continuous solution spaces. In Section 2, we give a brief introduction for cellular genetic algorithms. In Section 3, floating-point or real-valued genetic algorithms are explained. In Section 4, machine-coded (byte) genetic operators are described. In Section 5, description of our implementation is included. In Section 6, we perform a simulation study on a set of test functions. Finally in Section 7, we conclude.

2. Cellular genetic algorithms

Genetic algorithms are algorithms that expect each generation contains individuals better than the preceding one, as guaranteed by the schema theorem [5]. These algorithms possess highly desirable features such as easy implementation, compact storage, automatic utilization of substantial relevant information encountered during operation, and robustness (efficient operation under maximum uncertainty) through genetic operators [5]. Most genetic algorithms (GAs) typically use a single population. Subsequently, the operators employed are applied to this community as a whole. However, there are also structured GAs, especially suitable for parallel applications, where the population is not centralized [1]. The use of parallel distributed populations is based on the idea that higher genetic differentiation is achieved between populations [6]. Alba and Troya [7] conducted a comparative study on different types of parallel genetic algorithms. In this study, the analyzed features included time complexity, selection pressure, schema processing rates, efficiency in finding optima, performance, acceleration, and resistance to scalability. The results generally indicated that parallel GAs are numerically superior and faster. In most cases, algorithms that use decentralized populations have facilitated better sampling of the search space. Thus, equivalent to a panmictic algorithm (a population with random mating and no biased reproduction), these algorithms have improved both the numerical behavior and the execution time. Among the structured types of GAs used as optimization methods, distributed and cellular genetic algorithms are the most popular.

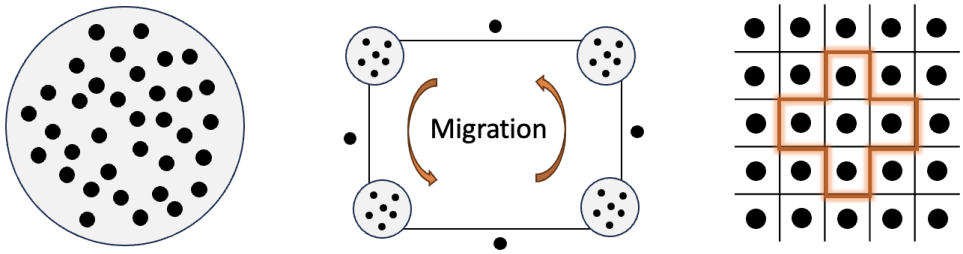
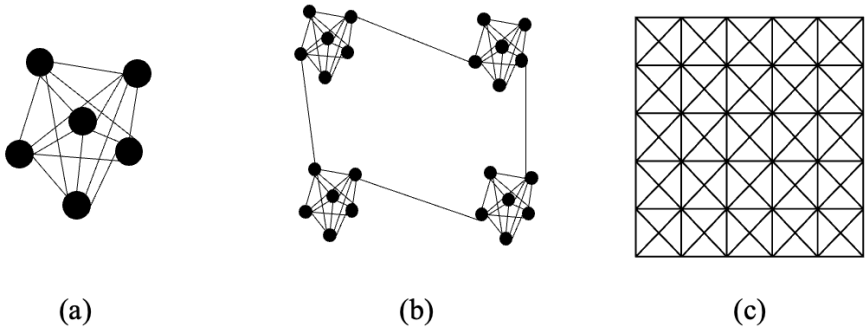


Figure 1: (a) Panmictic GA (b) Distributed GA (c) Cellular GA

In Figure 1, the structures of a standard GA and other structured types, including distributed and cellular genetic algorithms, are presented. In the standard GA, referred to as panmictic, a population with random mating and unbiased reproduction is observed. In the distributed GA, the population is divided into sections called islands, and a migration mechanism is established between these islands. In Cellular GA, individuals composing the population are arranged side by side on a two-dimensional grid, forming specific neighborhood structures. In Figure 2 below, the connections between individuals in the population for each type are depicted. In distributed GAs, the population is divided into a series of islands with distant isolation ensured. To prevent getting stuck in local optima and ensure genetic diversity within

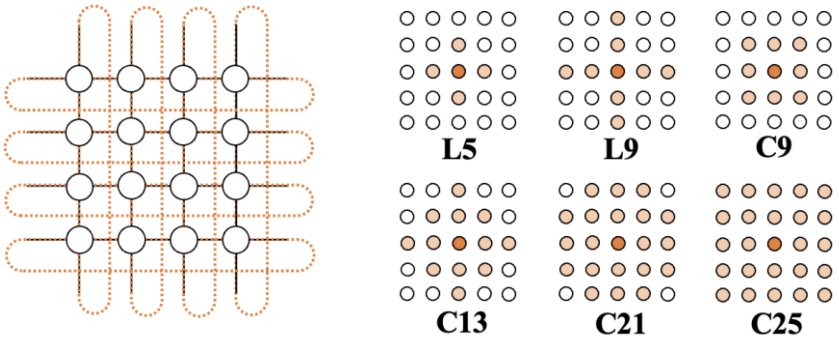
subpopulations, individuals are exchanged between islands periodically using the migration operator. In Cellular GAs, grid and neighborhood concepts are present. Such a structure implies that an individual can interact only with its nearby neighbors during the reproduction cycle. Overlapping small neighborhoods in Cellular GAs aid in exploring the search space. The gradual spread of solutions within the population through genetic operators facilitates diversification through exploration, while also achieving concentration through exploitation [1].

In cellular genetic algorithms (cGAs), where isolation is achieved through distance, the potential mates set for an individual is formed by its close neighbors. These neighborhoods, similar to the island model, create subpopulations.



(a) (b) (c)

Figure 2: (a) Panmictic GA (b) Distributed GA (c) Cellular GA



(a) (b)

Figure 3: (a) Toroidal population in cGAs and (b) the most commonly used neighborhoods.

As seen in Figure 3-a, in an cGA, the population is typically structured by positioning candidates on a two-dimensional grid. The use of this topology does not limit the scope of the obtained solutions [8]. As shown by dashed lines in Figure 3-a, individuals at the edges are connected to opposite boundaries within the same row or column. Thus, a toroidal grid in a ring shape is obtained, and all individuals have an equal number of neighbors. The grid dimensions used can be three or four-dimensional, but typically, grids are two-dimensional [9]. The distance between any two individuals on the grid is calculated using the Manhattan distance. Neighborhoods at each point on the grid are of the same size and shape, overlapping with the neighborhoods of nearby individuals. In Figure 3-b, six different neighborhood shapes frequently used in cGAs are illustrated. These neighborhood shapes are labeled as Ln and Cn. The label Ln (linear) is used for neighborhoods formed by n closest neighbors along a specific axis (north, south, east, and west), while the label Cn (compact) is used to denote neighborhoods encompassing the closer n-1 individuals in considered directions (horizontal, vertical, and diagonal) [1]. The number next to the label represents the total number of individuals in the neighborhood created by the neighbors. For example, the expression L9 signifies a neighborhood shape formed by a total of 9 individuals positioned linearly around an axis, and thus, it implies a neighborhood consisting of nine individuals. L5 (Von Neumann or NEWS neighborhood) and C9 (Moore neighborhood) are the two most commonly used neighborhoods among these. Experimental results have shown that the influence of neighbors is a significant factor in the

performance of an evolutionary algorithm. Regarding the size of the neighborhood, it has been found that a neighborhood range consisting of six, plus/minus two individuals is more efficient than the considered other dimensions in cGAs [10].

In Figure 4, the reproductive cycle of an individual in an cGA using the NEWS neighborhood is demonstrated, showing how it interacts with its neighbors. In the reproductive cycle where variation operators are applied in cGAs, individuals can only interact with their neighbors. This reproductive cycle is conducted within each individual's neighborhood. Initially, two parents are selected from among the neighbors based on certain criteria, and variation (crossover and mutation) operators are applied to them. Subsequently, if the newly created offspring have a better solution, they replace their parents.

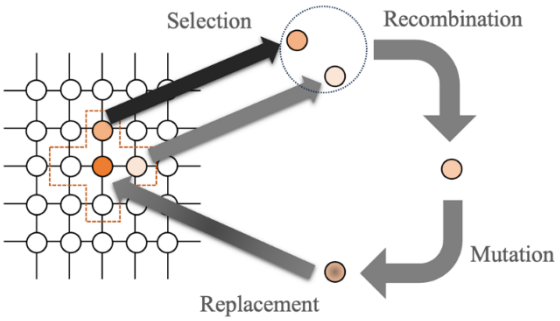


Figure 4: Reproductive cycle of an individual in cGA.

Neighborhood overlap provides an implicit migration mechanism to cGA. The smooth dissemination of the best solutions throughout the entire population ensures the long-term preservation of genetic diversity. The distribution of the best solutions within the population is a key goal of maintaining a good balance between exploration and exploitation during the search performed by cGAs. The degree of overlap between neighborhoods varies based on the size of the neighborhood. In Figure 5, the overlap degree of two different neighborhoods for the same individuals (drawn in two different shades of color) is illustrated. On the left, L5 (NEWS) neighborhood is shown, and on the right, C21 neighborhood is depicted. The individuals in darker color belong to both neighborhoods, while the other two in lighter color belong to only one neighborhood.

A cGA can be viewed as a cellular automaton (CA) with probabilistic rewrite rules. Therefore, analytical tools and existing models used in CAs can be transferred to cGAs to better understand and enhance their performance in a structured manner [11]. There are two different types of cGAs based on how the reproductive cycle is applied to individuals: synchronous and asynchronous cGAs. In synchronous cGAs, if the cycle is applied to all individuals simultaneously, the individuals creating the next generation in the population are generated simultaneously and in a synchronized manner. On the other hand, if the new individuals in the population are updated with a specific order policy and sequentially, it becomes an asynchronous cGA [12]. Schönfisch and Roos [13] conducted a comprehensive study on synchronous and asynchronous cellular automata, providing extensive insights. The behavior of an algorithm is influenced by parameters such as the update policy of individuals, the size and shape of the neighborhood forming the neighbors, and the order of individuals in asynchronous policy can also be another evaluation criterion. In asynchronous cGAs, the fundamental update policies for individuals are as follows [1]:

• Line Sweep-LS: It is the simplest method. It is achieved by updating individuals in the population row by row (1, 2, . . . , n) as shown in Figure 6.

• Fixed Random Sweep-FRS: In this case, the next cell to be updated is chosen with a uniform probability without replacement (i.e., it is not possible to select a cell twice in one generation). A constant permutation is used for all generations.

• New Random Sweep-NRS: Unlike constant random sweeping, in each generation, a new random cell permutation is used.

• Uniform Choice-UC: In this method, the next individual to be selected is randomly chosen with a uniform probability among all components of the resident population (thus, it is possible to select a cell more than once). This corresponds to a binomial distribution for the update probability.

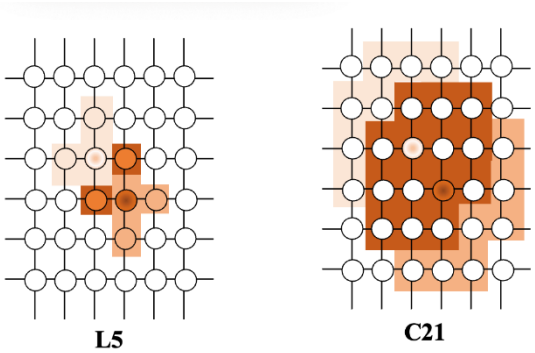


Figure 5: Larger neighborhoods result in a higher level of implicit migration.

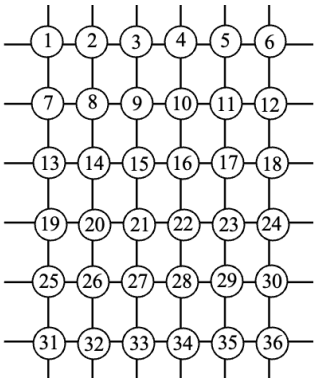


Figure 6: Numbering of individuals on a two-dimensional grid.

A time step (or a generation) is defined as the sequential update of n individuals. This corresponds to updating all individuals in the population simultaneously in synchronous cases. Other asynchronous update policies have a stochastic structure. When defining the characteristic parameters of an cGA population, the term "radius" is used for non-square grids [8, 14]. As seen in the following equation, it is assumed that the radius of a grid is equal to the distribution of n points in an ellipse centered at (x, y). This definition provides the radius value not only for characterizing the shape of the grid but also for the neighborhood shape. Although referred to as "radius," the concept of rad is more commonly used (Equation 1).

$$rad = \sqrt{\frac{\sum(x_i-x)^2+\sum(y_i-y)^2}{n}}, \quad x = \frac{\sum_{i=1}^{n} x_i}{n}, \quad y = \frac{\sum_{i=1}^{n} y_i}{n} \tag{1}$$

Sarma and Jong [8] argued that the relationship between the neighborhood shape and the grid can be measured through the ratio between radii (Equation 2). Algorithms with similar ratios can perform a similar search while using the same selection method [15].

$$ratio_{cGA} = \frac{rad_{Neighborhood}}{rad_{Grid}} \tag{2}$$

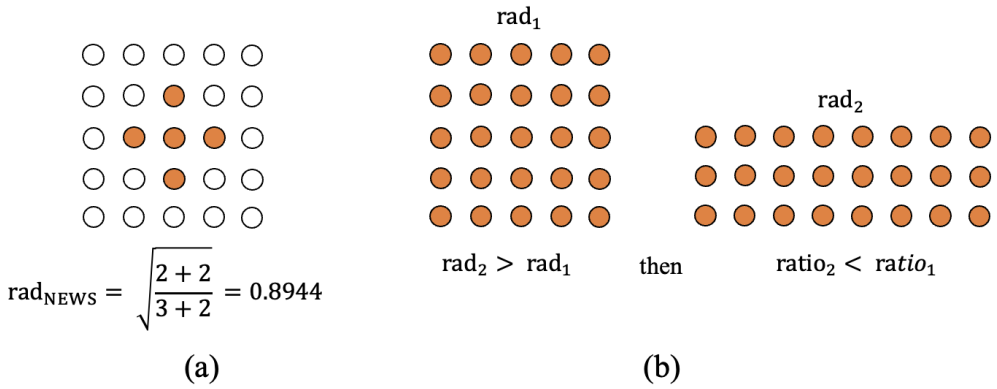


Figure 7: (a) The radius of the NEWS neighborhood can result in two different ratios for grids with (b) 5 × 5 = 25 and 3 × 8 ≈ 25 individuals, even though the number of individuals is the same.

Algorithm 1: Pseudocode for a Canonical, well-known, cGA

```
1.  proc Evolve (cga)      // Parameters of the algorithm in 'cga'
2.  GenerateInitialPopulation (cga.pop);
3.  Evaluation (cga.pop);
4.  while ! StopCondition() do
5.     for individual ← 1 to cga. cga.popSize do
6.        neighbors ← CalculateNeighborhood(cga,position(individual));
7.        parents ← Selection(neighbors);
8.        offspring ← Recombination(cga.Pc,parents);
9.        offspring ← Mutation(cga.Pm,offspring);
10.       Evaluation (offspring);
11.      Replacement(position(individual),auxiliary pop,offspring);
12.   end for
13. cga.pop ← auxiliary pop;
14. end while
15. end proc Evolve
```

To make a fair comparison between grids with different shapes, an equal number of individuals should be created. If the grid is narrow, as seen in Figure 7-b, the value of the topological radius will be higher. By keeping the size and shape of the used neighborhood constant (for example, using NEWS as in Figure 7-a), the ratio will be higher when the population grid is narrow. To reduce the value of the ratio, the intensity of global selection in the population can be decreased. This way, the exploration mechanism is encouraged, and a higher diversity is achieved within the population. This method is expected to improve the results for complex problems. Additionally, it guides the concentration of search within each neighborhood (exploitation). Algorithm 1 below presents pseudocode for a canonical, well-known, and generally accepted cGA. In this algorithm, individuals that make up the next generation's population are stored in an auxiliary population and, when completed, replace the current population with an atomic step, making it a synchronized cGA. In this model, all individuals in the population are updated simultaneously and equally. The creation of individuals is done only from the current population. In the asynchronous case, the generated children replace the individuals in the population as soon as they are selected, eliminating the need for an auxiliary population [1].

3. Floating-point or real-valued genetic algorithms

Before delving into machine-coded genetic operators, it would be helpful to first discuss floating-point genetic algorithms. Genetic algorithms that utilize chromosomes directly without the need for encoding and decoding are referred to as floating-point or real-valued genetic algorithms (FPGAs). Classical crossover operators combine two binary chromosomes in various ways, but their application is not straightforward when dealing with real-valued chromosomes in FPGAs. In addition, binary crossover operators, designed for flipping binary values, cannot be directly applied to real vectors. Similarly, the classical mutation operator, which involves flipping binary values, lacks a direct analogy for real vectors. Binary chromosomes require an increased number of bits for the representation of a more extensive solution space. On the other hand, the utilization of floating-point parameters enables the exploration of larger domains, even in cases where the limits of parameters are uncertain [16]. Nevertheless, the incorporation of crossover and mutation operations might not be immediately evident in FPGAs. The mutation operator, characterized by the flipping of a single bit, lacks meaningful applicability in the context of real numbers. Moreover, finding a method to combine two parent real vectors for the creation of new real vectors is not a straightforward task [17]. To address these challenges, crossover and mutation operators have been devised specifically for FPGAs [4]. Some of these are;

- Flat Crossover [16, 18],
- Arithmetic Crossover [19],
- BLX-a Crossover [20, 21],
- Linear Crossover [22, 16],
- Simulated Binary Crossover [23, 20],
- Unfair Average Crossover [24].

Numerous crossover methods have been proposed in the literature, and a majority of them rely on linear or non-linear combinations of real values encapsulated within floating-point chromosomes. Crossover methods designed for FPGAs exhibit comparable performance and are contingent on the specific content involved [17]. Therefore, researchers conducted a simulation study comparing crossover and mutation methods in constraint problems and found no superior method between them [21, 25].

4. Machine-coded (byte) genetic operators

In compiled computer programs, numeric data is typically stored as byte arrays in the memory. One byte consists of eight bits. Each individual bit can have a value of either zero or one. A small number of bytes are needed for storing a small-digit number, but if the number has more digits or precision is crucial, a larger number of bytes is required. This necessity is addressed through the implementation of data types in compilers and interpreters.

The storage of a numerical value in memory involves a finite number of bytes, implying that it is impossible to represent real-values with absolute precision. With an increase in the number of bytes, precision also improves, allowing for a more reasonable representation of real values [4]. The byte array depicted in Table 1 is generated through a formulation algorithm specified in IEEE 754 – IEEE Standard for Floating-Point Arithmetic [26]. Compilers typically follow this standard when converting floating-point numbers into byte arrays and vice versa. Assume f is a floating point number with a value accurate to 15 decimal points. The declaration of the variable f can be expressed as:

$$f = 12.508239423942167$$

When the variable f is converted to a byte, an 8-byte array is obtained, with each element in the array ranging from 0 to 255, as shown in the table below.

Table 1. Byte representation of vaiable *f*

	1	2	3	4	5	6	7	8
Bytes	45	189	48	245	55	4	41	64

The traditional crossover operator can be directly implemented on the byte representation of real values, and similarly the classical mutation operator can be simulated using its byte-based counterpart [2]. The byte-based mutation operator involves changing a byte by 1 instead of adding a random value. The impact of the mutation operation relies on the position of the mutated byte. This is analogous to the mutation operator used in binary-coded genetic algorithms. In machine coded mutation, each byte of a chromosome undergoes mutation with a probability of P_m. If a byte is selected for mutation, it is either increased by +1 or decreased by -1 with a probability of ½. The key benefit of employing this operator is the ability to achieve both minute and substantial changes without the need for fine-tuning optimization and operator parameters [4].

The byte based operator essentially performs a similar role to the crossover operator in GAs. In a study conducted on certain test functions, byte operators yielded better results than others [27]. It utilizes building blocks of chromosomes and assembles them instead of directly manipulating real values. Its primary advantage lies in its speed, as it does not necessitate arithmetic operations, particularly multiplication and division, which can be time-consuming [4]. In addition to the one-point crossover operator, other well-known crossover operators such as two-point crossover and uniform crossover can be implemented in a manner similar to their binary counterparts [2].

5. Designed software

Improved Cellular Genetic Algorithm with machine coded byte operators is implemented in Python, one of the most popular programming languages. It is easy to apply encoding and decoding strategy for real valued problems in this language. We developed a software named “pycell” for testing our algorithm [28]. In addition to cellular genetic algorithm, it includes functions for both classical floating point operators and byte-based genetic operators, as illustrated below [2].

- byte crossover,
- byte crossover 1p,
- byte mutation,
- byte mutation dynamic,
- byte mutation random,
- byte mutation random dynamic

Table 2 shows the crossover operators used in the pycell according to encoding types of chromosome. Arithmetic crossover, blxalpha crossover, flat crossover, linear crossover and unfair average crossover are real valued encoding types which classical recombination method for real valued optimization. Byte one point crossover and byte uniform crossover are machine coded operators for real valued problems. In contrast to the literature, byte-based crossover operators convert each allele in the chromosome into bytes and perform the crossover operation based on them.

Similarly, Table 3 illustrates the mutation operators based on encoding types of chromosome. Float uniform mutation is a classical mutation operator for real valued problems. Moreover, byte mutation, byte mutation dynamic, byte mutation random and byte mutation random dynamic are machine coded operators.

Table 2. Crossover operators according to encoding types of chromosome

Recombination	Encoding type
Arithmetic crossover	Real-valued
Blxalpha crossover	Real-valued
Flat crossover	Real-valued
Linear crossover	Real-valued
Unfair average crossover	Real-valued
Byte one point crossover	Real-valued (BYTE)
Byte uniform crossover	Real-valued (BYTE)

6. Simulation study

We conduct a simulation study to assess the efficiency of the cellular genetic algorithm with machine-coded operators in solving real-valued problems. The changeable values in the experimental design are TestFunction, KnownBestSolution, Recombination, and Mutation. The others were kept constant in all experiments. Additionally, since the cellular genetic algorithm inherently involves probabilistic calculations, each experiment was repeated 100 times.

The names of the 13 different scenarios employed in the experiment, along with the corresponding

crossover and mutation operators, are presented in Table 5.

Simulations were conducted with well-known test functions, including Ackley, Bohachevsky, Griewank, Holzman, Rastrigin, Rosenbrock, Schaffer, Chichinadze and Sphere, as outlined in Table 6 [29, 30, 31, 32, 33, 34, 35, 36, 37, 38].

Table 3. Mutation operators according to encoding types of chromosome

Mutation	Encoding type
Float uniform mutation	Real-valued
Byte mutation	Real-valued (BYTE)
Byte mutation dynamic	Real-valued (BYTE)
Byte mutation random	Real-valued (BYTE)
Byte mutation random dynamic	Real-valued (BYTE)

Table 4. Simulation parameters

Method = cga
Gen_Type = **Real-valued**
TestFunction = **Ackley**
KnownBestSolution = **0**
ChromosomeSize = 25
Selection = TournamentSelection
Recombination = **ByteUniformCrossover**
Mutation = **ByteMutation _dynamic**
Neighborhood = Linear9
Number_of_Columns = 10
Number_of_Rows = 10
Number_of_Generation = 500
Probability_of_Crossover = 90
Probability_of_Mutation = 50

Table 5. Names of the scenarios and the used operators

sn.	*Operators* / *Names*	Crossover	Mutation
1.	arithmetic_cross	ArithmeticCrossover	FloatUniformMutation
2.	blxalpha_cross	BlxalphaCrossover	FloatUniformMutation
3.	flat_cross	FlatCrossover	FloatUniformMutation
4.	linear_cross	LinearCrossover	FloatUniformMutation
5.	unfair_av_cross	UnfairAvarageCrossover	FloatUniformMutation
6.	byte_1p_cross-bytemutrand	ByteOnePointCrossover	ByteMutationRandom
7.	byte_1p_cross-bytemutrand_dynm	ByteOnePointCrossover	ByteMutationRandom dynamic
8.	byte_1p_cross-bytemut	ByteOnePointCrossover	ByteMutation
9.	byte_1p_cross-bytemut_dynm	ByteOnePointCrossover	ByteMutation dynamic
10.	byte_uniform_cross-bytemutrand	ByteUniformCrossover	ByteMutationRandom
11.	byte_uniform_cross-bytemutr_dyn	ByteUniformCrossover	ByteMutationRandom dynamic
12.	byte_uniform_cross-bytemut	ByteUniformCrossover	ByteMutation
13.	byte_uniform_cross-bytemut_dynm	ByteUniformCrossover	ByteMutation dynamic

The found best solution obtained from each of these test functions (Found_Best_Solution), the averages of the best solutions (Avg. Best_Solution), the averages of generation at which it was found (Avg. Found_at_Generation), the averages of time in seconds (Avg. Time-seconds), and the number of times it found the known best solution (How many times find the known best solution) are separately presented in each table below.

The results for the Ackley function are shown in Table 7. The global minimum for the Ackley function is 0,00. Accordingly, the best values obtained are indicated in bold in the table. Among the experiment cases for the Ackley function, “byte_1p_cross-

bytemutrand" "byte_1p_cross-bytemutrand_dynm" "byte_1p_cross-bytemut", "byte_uniform_cross-bytemutrand" and "byte_uniform_cross-bytemut" have achieved quite successful results compared to others. Among them, the one highlighted in yellow, "byte_1p_cross-bytemutrand_dynm" is the best in terms of all variables.

The results for the Bohachevsky function are presented in Table 8. The global minimum for the Bohachevsky function is also 0,00. Accordingly, the best values obtained are indicated in bold in the table. For the Bohachevsky function, the experiment cases "blxalpha_cross", "byte_1p_cross-bytemutrand", "byte_1p_cross-bytemutrand_dynm", "byte_1p_cross-bytemut", "byte_1p_cross-bytemut_dynm", "byte_uniform_cross-bytemutrand" and "byte_uniform_cross-bytemut" have achieved quite successful results compared to others. Among them, the one highlighted in yellow, "byte_1p_cross-bytemutrand_dynm" is again the best in terms of all variables.

Table 6. Test functions used in the simulation study

Function Name	Definition	Domain
Ackley	$f(x) = 20 - 20\exp(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}) + e - \exp(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i))$	$-32.768 \leq x_i \leq 32.768$
Bohachevsky	$f(x) = \sum_{i=0}^{n}(x_i^2 + 2x_{i+1}^2 - 0.3\cos(3\pi x_i) - 0.4\cos(4\pi x_{i+1}) + 0.7$	$-15 \leq x_i \leq 15$
Griewank	$f(x) = 1 + \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n}\cos(\frac{x_i}{\sqrt{i}})$	$-600 \leq x_i \leq 600$
Holzman	$f(x) = \sum_{i=1}^{n} i x_i^4$	$-10 \leq x_i \leq 10$
Rastrigin	$f(x) = 10n + \sum_{i=1}^{n}(x_i^2 - 10\cos(2\pi x_i))$	$-5.12 \leq x_i \leq 5.12$
Rosenbrock	$f(x) = \sum_{i=1}^{n-1}[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$	$-5 \leq x_i \leq 10$
Schaffer	$f(x) = \sum_{i=1}^{n-1}(x_i^2 + x_{i+1}^2)^{0,25}[sin^2(50(x_i^2 + x_{i+1}^2)^{0.1}) + 1.0]$	$-100 \leq x_i \leq 100$
Chichinadze	$f(x,y) = x^2 - 12x + 11 + 10\cos(\frac{\pi x}{2}) + 8\sin(5\pi x) - \frac{1}{\sqrt{5}} e^{-0.5(y-0.5)^2}$	$-30 \leq x,y \leq 30$
Sphere	$f(x) = \sum_{i=1}^{n} x_i^2$	$-5.12 \leq x_i \leq 5.12$

Table 7. The results of the Ackley function

Parameters / *Cases*	Best Solution Found	Avg. Best Solution	Avg.Found at Generation	Avg. Time (s)	How many times find the best solution
arithmetic_cross	4,50	7,193	295	10,16	0
blxalpha_cross	0,00	0,001	346	52,10	42
flat_cross	3,62	5,76	385	15,09	0
linear_cross	3,33	5,17	490	12,84	0
unfair_av_cross	1,47	2,61	463	10,76	0
byte_1p_cross-bytemutrand	**0,00**	**0,00**	22	38,03	**100**
byte_1p_cross-bytemutrand_dynm	**0,00**	**0,00**	**22**	**11,83**	**100**
byte_1p_cross-bytemut	**0,00**	**0,00**	22	13,44	**100**
byte_1p_cross-bytemut_dynm	**0,00**	**0,00**	24	10,26	**100**
byte_uniform_cross-bytemutrand	**0,00**	**0,00**	42	35,92	**100**

byte_uniform_cross-bytemutr_dyn	0,00	3,12	41	13,89	1
byte_uniform_cross-bytemut	**0,00**	**0,00**	40	16,33	**100**
byte_uniform_cross-bytemut_dynm	0,00	3,10	41	45,23	1

Table 8. The results of the Bohachevsky function

Parameters / *Cases*	Best Solution Found	Avg. Best Solution	Avg.Found at Generation	Avg. Time (s)	How many times find the best solution
arithmetic_cross	26,95	41,883	464	15,05	0
blxalpha_cross	**0,00**	**0,00**	229	26,44	**100**
flat_cross	22,21	38,19	410	19,41	0
linear_cross	15,62	33,26	493	17,42	0
unfair_av_cross	1,48	8,34	486	15,20	0
byte_1p_cross-bytemutrand	**0,00**	**0,00**	6	25,49	**100**
byte_1p_cross-bytemutrand_dynm	**0,00**	**0,00**	**6**	**17,42**	**100**
byte_1p_cross-bytemut	**0,00**	**0,00**	6	23,96	**100**
byte_1p_cross-bytemut_dynm	**0,00**	**0,00**	6	18,55	**100**
byte_uniform_cross-bytemutrand	**0,00**	**0,00**	34	27,13	**100**
byte_uniform_cross-bytemutr_dyn	0,00	16,40	31	22,07	23
byte_uniform_cross-bytemut	**0,00**	**0,00**	33	27,59	**100**
byte_uniform_cross-bytemut_dynm	0,00	16,66	31	21,86	22

Table 9. The results of the Griewank function

Parameters / *Cases*	Best Solution Found	Avg. Best Solution	Avg.Found at Generation	Avg. Time (s)	How many times find the best solution
arithmetic_cross	2,19	4,563	497	2,70	0
blxalpha_cross	0,00	0,002	285	4,41	19
flat_cross	1,28	2,20	497	4,34	0
linear_cross	1,48	2,56	498	4,03	0
unfair_av_cross	0,00	0,04	491	2,77	0
byte_1p_cross-bytemutrand	**0,00**	**0,00**	**69**	**3,70**	**100**
byte_1p_cross-bytemutrand_dynm	**0,00**	**0,00**	**73**	**3,36**	**100**
byte_1p_cross-bytemut	**0,00**	**0,00**	**77**	**3,88**	**100**
byte_1p_cross-bytemut_dynm	**0,00**	**0,00**	**73**	**3,38**	**100**
byte_uniform_cross-bytemutrand	0,00	0,00	94	4,96	93
byte_uniform_cross-bytemutr_dyn	2,62	4,93	88	4,80	0
byte_uniform_cross-bytemut	1,75	5,07	79	4,77	0
byte_uniform_cross-bytemut_dynm	2,02	5,14	86	4,77	0

Table 10. The results of the Holzman function

Parameters / *Cases*	Best Solution Found	Avg. Best Solution	Avg.Found at Generation	Avg. Time (s)	How many times find the best solution
arithmetic_cross	7,61	58,231	368	4,65	0
blxalpha_cross	**0,00**	**0,00**	**189**	**6,44**	**100**
flat_cross	7,77	39,62	368	6,33	0
linear_cross	2,32	16,27	499	5,90	0
unfair_av_cross	0,00	0,06	493	4,79	0
byte_1p_cross-bytemutrand	**0,00**	**0,00**	**44**	**5,67**	**100**
byte_1p_cross-bytemutrand_dynm	**0,00**	**0,00**	**44**	**4,59**	**100**
byte_1p_cross-bytemut	**0,00**	**0,00**	**44**	**5,68**	**100**
byte_1p_cross-bytemut_dynm	**0,00**	**0,00**	**44**	**4,62**	**100**
byte_uniform_cross-bytemutrand	**0,00**	**0,00**	**33**	**7,20**	**100**
byte_uniform_cross-bytemutr_dyn	0,00	6,56	31	5,96	92
byte_uniform_cross-bytemut	**0,00**	**0,00**	**32**	**7,06**	**100**
byte_uniform_cross-bytemut_dynm	0,00	6,56	32	5,97	93

The results for the Griewank function are presented in Table 9. The global minimum for the Griewank function is also 0,00. Accordingly, the best values obtained are indicated in bold in the table. For the Griewank function, the experiment cases "byte_1p_cross-bytemutrand", "byte_1p_cross-bytemutrand_dynm", "byte_1p_cross-bytemut" and "byte_1p_cross-bytemut_dynm" have achieved quite

successful results compared to others. Among them, the one highlighted in yellow, "byte_1p_cross-bytemutrand" is the best in terms of all variables.

The results for the Holzman function are presented in Table 10. The global minimum for the Holzman function is also 0,00. Accordingly, the best values obtained are indicated in bold in the table. For the Holzman function, the experiment cases "blxalpha_cross", "byte_1p_cross-bytemutrand", "byte_1p_cross-bytemutrand_dynm", "byte_1p_cross-bytemut", "byte_1p_cross-bytemut_dynm", "byte_uniform_cross-bytemutrand" and "byte_uniform_cross-bytemut" have achieved quite successful results compared to others. Among them, the one highlighted in yellow, "byte_uniform_cross-bytemut" is the best in terms of all variables.

In Table 11, the results for the Rastrigin function are presented. The global minimum of the Rastrigin function is also 0,00. Accordingly, the best values obtained are indicated in bold in the table. For the Rastrigin function, the experimental cases "byte_1p_cross-bytemutrand", "byte_1p_cross-bytemutrand_dynm", "byte_1p_cross-bytemut", "byte_1p_cross-bytemut_dynm", "byte_uniform_cross-bytemutrand", "byte_uniform_cross-bytemutr_dyn", "byte_uniform_cross-bytemut", and "byte_uniform_cross-bytemut_dynm" have achieved quite successful results compared to others. Among them, the best in terms of all variables is highlighted in yellow, and it is the condition "byte_uniform_cross-bytemutr_dyn".

In Table 12, the results for the Rosenbrock function are presented. The global minimum of the Rosenbrock function is also 0,00. However, none of the conditions have achieved an exact solution. The best in terms of all variables is highlighted in yellow, and it is the condition "blxalpha_cross".

Table 11. The results of the Rastrigin function

Parameters / *Cases*	Best Solution Found	Avg. Best Solution	Avg.Found at Generation	Avg. Time (s)	How many times find the best solution
arithmetic_cross	86,88	121,758	263	17,93	0
blxalpha_cross	19,70	41,314	466	16,23	0
flat_cross	26,71	47,52	482	15,48	0
linear_cross	102,19	130,42	299	13,58	0
unfair_av_cross	5,01	12,12	491	11,70	0
byte_1p_cross-bytemutrand	**0,00**	**0,00**	41	13,77	**100**
byte_1p_cross-bytemutrand_dynm	**0,00**	**0,00**	42	10,42	**100**
byte_1p_cross-bytemut	**0,00**	**0,00**	42	15,48	**100**
byte_1p_cross-bytemut_dynm	**0,00**	**0,00**	41	18,55	**100**
byte_uniform_cross-bytemutrand	**0,00**	**0,00**	27	18,09	**100**
byte_uniform_cross-bytemutr_dyn	0,00	**0,00**	**26**	**14,45**	**100**
byte_uniform_cross-bytemut	**0,00**	**0,00**	26	17,63	**100**
byte_uniform_cross-bytemut_dynm	0,00	**0,00**	26	47,69	**100**

Table 12. The results of the Rosenbrock function

Parameters / *Cases*	Best Solution Found	Avg. Best Solution	Avg.Found at Generation	Avg. Time (s)	How many times find the best solution
arithmetic_cross	8369,60	25023,403	463	21,76	0
blxalpha_cross	**0,02**	**1,670**	**493**	**25,34**	**0**
flat_cross	5762,62	16732,72	446	25,58	0
linear_cross	864,23	9709,19	499	25,56	0
unfair_av_cross	7,23	106,27	498	21,84	0
byte_1p_cross-bytemutrand	23,77	23,99	74	23,67	0
byte_1p_cross-bytemutrand_dynm	23,99	24,00	6	17,41	0
byte_1p_cross-bytemut	24,00	24,00	6	28,99	0
byte_1p_cross-bytemut_dynm	24,00	24,00	6	16,76	0
byte_uniform_cross-bytemutrand	24,00	44,02	50	26,48	0
byte_uniform_cross-bytemutr_dyn	24,00	764,00	43	20,99	0
byte_uniform_cross-bytemut	24,00	56,00	49	27,96	0
byte_uniform_cross-bytemut_dynm	24,00	690,03	43	23,86	0

Table 13. The results of the Schaffer function

Parameters / *Cases*	Best Solution Found	Avg. Best Solution	Avg.Found at Generation	Avg. Time (s)	How many times find the best solution
arithmetic_cross	3,95	4,93	410	5,88	0
blxalpha_cross	5,65	6,63	409	7,62	0
flat_cross	4,52	5,29	338	7,48	0
linear_cross	4,93	5,81	306	7,16	0
unfair_av_cross	3,60	4,69	425	5,94	0
byte_1p_cross-bytemutrand	**0,42**	**1,19**	**485**	**18,00**	**0**
byte_1p_cross-bytemutrand_dynm	**0,55**	**1,20**	**468**	**12,75**	**0**
byte_1p_cross-bytemut	**0,29**	**1,02**	**481**	**18,08**	**0**
byte_1p_cross-bytemut_dynm	**0,73**	**1,32**	**469**	**12,87**	**0**
byte_uniform_cross-bytemutrand	**0,33**	**1,12**	**480**	**18,64**	**0**
byte_uniform_cross-bytemutr_dyn	0,90	2,16	338	13,50	0
byte_uniform_cross-bytemut	**0,59**	**1,32**	**421**	**18,63**	**0**
byte_uniform_cross-bytemut_dynm	1,26	2,12	318	13,53	0

The results for the Schaffer function are presented in Table 13. The global minimum for the Schaffer function is also 0,00. Accordingly, the best values obtained are indicated in bold in the table. For the Schaffer function, the experiment cases “byte_1p_cross-bytemutrand”, “ byte_1p_cross-bytemutrand_dynm”, “byte_1p_cross-bytemut”, “byte_1p_cross-bytemut_dynm”, “byte_uniform_cross-bytemutrand” and “byte_uniform_cross-bytemut” have achieved quite successful results compared to others. Among them, the one highlighted in yellow, “byte_1p_cross-bytemut” is the best in terms of all variables.

Table 14. The results of the Chichinadze function

Parameters / *Cases*	Best Solution Found	Avg. Best Solution	Avg.Found at Generation	Avg. Time (s)	How many times find the best solution
arithmetic_cross	**-43,3159**	-43,3148	26	1,14	79
blxalpha_cross	**-43,3159**	**-43,3159**	**28**	**1,32**	**100**
flat_cross	**-43,3159**	-43,3158	23	1,25	98
linear_cross	**-43,3159**	-43,3159	84	1,31	98
unfair_av_cross	**-43,3159**	**-43,3159**	**31**	**1,22**	**100**
byte_1p_cross-bytemutrand	**-43,3159**	-42,8199	96	1,80	1
byte_1p_cross-bytemutrand_dynm	-43,2622	-37,1495	4	1,30	0
byte_1p_cross-bytemut	-43,2633	-40,9002	182	1,82	0
byte_1p_cross-bytemut_dynm	-43,2319	-36,2079	4	1,33	0
byte_uniform_cross-bytemutrand	**-43,3159**	-42,8523	74	1,91	24
byte_uniform_cross-bytemutr_dyn	-43,3157	-42,5146	21	1,40	0
byte_uniform_cross-bytemut	**-43,3159**	-42,9312	68	1,86	36
byte_uniform_cross-bytemut_dynm	-43,3154	-42,4968	22	1,40	0

Table 15. The results of the Sphere function

Parameters / *Cases*	Best Solution Found	Avg. Best Solution	Avg.Found at Generation	Avg. Time (s)	How many times find the best solution
arithmetic_cross	1,84	4,087	255	6,10	0
blxalpha_cross	**0,00**	**0,000**	225	12,40	97
flat_cross	1,54	2,95	238	9,96	0
linear_cross	0,94	2,41	498	8,83	0
unfair_av_cross	0,00	0,01	456	6,21	0
byte_1p_cross-bytemutrand	**0,00**	**0,00**	48	10,77	**100**
byte_1p_cross-bytemutrand_dynm	**0,00**	**0,00**	49	8,49	**100**
byte_1p_cross-bytemut	**0,00**	**0,00**	49	9,33	**100**
byte_1p_cross-bytemut_dynm	**0,00**	**0,00**	49	8,60	**100**
byte_uniform_cross-bytemutrand	**0,00**	**0,00**	26	11,65	**100**
byte_uniform_cross-bytemutr_dyn	0,00	**0,00**	25	12,48	**100**
byte_uniform_cross-bytemut	**0,00**	**0,00**	26	11,57	**100**
byte_uniform_cross-bytemut_dynm	**0,00**	**0,00**	**25**	**10,79**	**100**

In Table 14, the results for the Chichinadze function are presented. Unlike the other functions, the number of chromosomes is 2. The global minimum of the Chichinadze function is −43.3159. Accordingly, the best values obtained are indicated in bold in the table. Accordingly, the best values obtained are indicated in bold in the table. For the Chichinadze function, the conditions “arithmetic_cross”, “blxalpha_cross”, “flat_cross” and “linear_cross”, “unfair_av_cross” have achieved quite successful results compared to others. Among them, the best in terms of all variables is highlighted in yellow, and it is the condition “blxalpha_cross”.

Table 15 displays the results for the Sphere function. The global minimum of the Sphere function is also 0,00. Accordingly, the best values obtained are indicated in bold in the table. For the Sphere function, the conditions “blxalpha_cross”, “byte_1p_cross-bytemutrand” “byte_1p_cross-bytemutrand_dynm” “byte_1p_cross-bytemut” “byte_1p_cross-bytemut_dynm” “byte_uniform_cross-bytemutrand”, “byte_uniform_cross-bytemutr_dyn” and “byte_uniform_cross-bytemut” have achieved quite successful results compared to others. Among them, the best in terms of all variables is highlighted in yellow, and it is the condition “byte_uniform_cross-bytemut_dynm”.

Reviewing all the results from the experiments conducted through simulations for Ackley and Bohachevsky functions, the scenario involving "byte one-point crossover" with "byte mutation random dynamic" yields the best result. Additionally, for the Griewank function, the optimal case is “byte one-point crossover” with “byte mutation random”. For the Holzman function, the best combination is “byte uniform crossover” with “byte mutation”. Regarding the Rastrigin function, the scenario involving "byte uniform crossover" with "byte mutation random dynamic" produces the best outcome. When examining the results for the Rosenbrock and Chichinadze functions, the combination of "blxalpha crossover" and "float uniform mutation" yields the best outcome. Moreover, for the Schaffer function, the optimal case is “byte one-point crossover” with “byte mutation”. Lastly, for the Sphere function, "byte uniform crossover" with "byte mutation dynamic" is the most effective. Therefore, it can be concluded that the performance of the new cellular genetic algorithm, where byte-based operators are applied, has improved, and it is more successful than other classical operators for the implemented real-valued problems.

7. Conclusion

In conclusion, the presented study introduced an Improved Cellular Genetic Algorithm with machine-coded operators tailored for addressing real-valued optimization problems. The utilization of byte-based operators demonstrated promising results in comparison to traditional floating-point genetic algorithms. The simulation experiments across various test functions, including Ackley, Bohachevsky, Griewank, Holzman, Rastrigin, Schaffer and Sphere, revealed that specific configurations of byte-based operators outperformed others.

The cases with byte-based crossover and mutation operators, such as “byte one-point crossover" with "byte mutation random dynamic" for Ackley and Bohachevsky functions, "byte uniform crossover" with "byte mutation random dynamic" for Rastrigin and "byte uniform crossover" with "byte mutation dynamic" for Sphere, consistently yielded superior results. This implies that the application of byte-based operators in the newly designed cGA led to improved performance, showcasing better adaptability to real-valued optimization challenges compared to conventional cellular genetic algorithm operators.

The findings suggest that the proposed cGA, equipped with machine-coded byte operators, can improve the efficiency and effectiveness of solving complex real-valued optimization problems. The study creates opportunities for further exploration in the integration of byte-based operators in cellular genetic algorithms and their applicability to diverse optimization scenarios. Overall, this research contributes valuable insights into advancing cellular genetic algorithms for real-valued optimization tasks.

Acknowledgements

This research is produced from the first author’s doctoral thesis, which has been entitled “Investigation of Cellular Genetic Algorithms and Improvement of their Performance,” supervised by Prof. Dr. Mehmet Hakan SATMAN from İstanbul University, Institute of Science.

References

[1] Alba, E. and Dorronsoro B., 2008, Cellular genetic algorithms, Operations research/computer science ınterfaces series, Springer, US, ISBN: 978-0-387-77609-5.

[2] Satman, M. H. and Akadal, E., 2017, Machine-coded genetic operators and their performances in floating-point genetic algorithms, International

journal of advanced mathematical sciences, 5(1), 8-19.
[3] Fernando, G., L., and David E., Goldberg, 2004, The parameter-less genetic algorithm in practice. Information Sciences, 167(1), 217-232.
[4] Satman, M., 2013, Machine coded genetic algorithms for real parameter optimization problems. Gazi University Journal of Science, 26(1), 85-95.
[5] Holland, J. H., 1973, Genetic algorithms and the optimal allocation of trials, SIAM Journal on computing, 2 (2), 88–105.
[6] Wright, S., 1943, Isolation by distance, Genetics, 28, 114–138.
[7] Alba, E. and Troya, J. M., 2002, Improving flexibility and efficiency by adding parallelism to genetic algorithms, Statistics and computing, 12 (2), 91–114.
[8] Sarma, J. and De Jong, K., 1996, An analysis of the effects of neighborhood size and shape on local selection algorithms, Lecture notes in computer science, 236–244.
[9] Tomassini, M., 2005, Spatially structured evolutionary algorithms: Artificial evolution in space and time, Natural computing series, Springer-Verlag, Heidelberg, ISBN-13: 978-3-540-24193-5.
[10] Salto, C. And Alba, E., 2019, Cellular genetic algorithms: Understanding the behavior of using neighborhoods, Applied artificial ıntelligence, 33 (10), 863-880.
[11] Tomassini, M., 2010, Cellular evolutionary algorithms, Simulating complex systems by cellular automata, In: Hoekstra, A.G. et al. (ed.), Chapter 8, Springer-Verlag, Heidelberg, 167–191.
[12] Alba, E., Dorronsoro, B., Giacobini, M., Tomassini, M., 2006, Decentralized cellular evolutionary algorithms, Handbook of bioinspired algorithms and applications, Chapter 7, CRC Press, 103–120.
[13] Schönfisch, B., and De Roos, A., 1999, Synchronous and asynchronous updating in cellular automata. Biosystems, 51 (3), 123–143.
[14] Alba, E., and Troya, J.M., 2000, Cellular evolutionary algorithms: Evaluating the influence of ratio, Proc. of the International Conference on Parallel Problem Solving from Nature VI (PPSN-VI), In: Schoenauer, M., (ed.), Springer-Verlag, Heidelberg, 29-38.
[15] Sarma, J., and De Jong, K. A., 1997, An analysis of local selection algorithms in a spatially structured evolutionary algorithm, Proc. of the Seventh International Conference on Genetic Algorithms (ICGA), In: Bäck, T., (ed.), Morgan Kaufmann, 181–186.
[16] Herrera, F., Lozano, M., Verdegay, J.L., 1998, "Tackling Real-Coded Genetic Algorithms: Operators and Tools for Behavioural Analysis", Artificial Intelligence Review, 12: 265-319.
[17] Deb, K.,2004, "Multi-Objective Optimization using Evolutionary Algorithms", John Wiley & Sons.
[18] Radcliffe, Nicholas J., 1991, Equivalence class analysis of genetic algorithms. Complex systems, 5(2):183-205.
[19] Michalewicz, Zbigniew, 2013, Genetic algorithms+ data structures = evolution programs. Springer Science & Business Media.
[20] Eshelman, Larry J., 1993, real-coded genetic algorithms and interval- schemata. Foundations of genetic algorithms, 2:187-202.
[21] Herrera, F., Lozano, M., Sanchez, A.M., 2003, A Taxonomy for the Crossover Operator for Real-Coded Genetic Algorithms: An Experimental Study, International Journal of Intelligent Systems, 18(3): 309-338.
[22] Wright, Alden H., 1991, Genetic algorithms for real parameter optimization. Foundations of genetic algorithms, 1:205-218.
[23] Kalyanmoy D., and Kumar A., 1995, Real-coded genetic algorithms with simulated-binary crossover: Studies on multi-modal and multi- objective problems. Complex systems, 9(6):431-454.
[24] Nomura, T., and Miyoshi, T., 1995, Numerical coding and unfair average crossover in ga for fuzzy rule extraction in dynamic environ- ments, In Fuzzy Logic, Neural Networks, and Evolutionary Computation, Springer, 55-72.
[25] Elsayed, S.M., Sarker, R.A., Essam, D.L., 2011, "Multi- operator Based Evolutionary Algorithms for Solving Constrained Optimization Problems", Computers & Operations Research, 38: 1877-1896.
[26] Stevenson, D., 1981, A Proposed Standard for Binary Floating-Point Arithmetic", Draft 8.0 of IEEE Task P754, 10.1109/C-M.1981.220377, 51-62.
[27] Cortez, P, 2014, Modern optimization with R, New York: Springer, https://books.google.com.tr/books?id=D1GfoAEACAAJ
[28] Akten, S., and Satman, M. H., 2024, PYCELL, Cellular genetic algorithm in Python, Zenodo, https://doi.org/10.5281/zenodo.11472419
[29] Hansen, N., Kern S., 2004, "Evaluating the CMA Evolution Strategy on Multimodal Test Functions", Parallel Problem Solving from Nature - PPSN VIII, 282-291.
[30] Vesterstrøm, J., and Thomsen, R., 2004, A comparative study of dif- ferential evolution, particle swarm optimization, and evolutionary algo- rithms on numerical benchmark problems. In Evolutionary Computation, CEC2004, IEEE, 1980-1987.
[31] Satman, M., H., and Akadal, E., 2016, Arima forecasting as a ge- netic inheritance operator in

foating-point genetic algorithms, Journal of Mathematical and Computational Science, 6(3), 360.
[32] Satman, M., and Akadal, E., 2018, İstatistikte Optimizasyonun Yeri ve Evrimsel Optimizasyon Algoritmaları.
[33] Mishra, S., K., 2006, Some New Test Functions for Global Optimization and Performance of Repulsive Particle Swarm Method. University Library of Munich, Germany, MPRA Paper. 10.2139/ssrn.926132.
[34] Bohachevsky, I.O., Johnson, M.E., and Stein, M.L., 1986, General simulated annealing for function optimization, Technometrics, 28 (3), 209–217.
[35] Rastrigin, L., A., 1974, Systems of extremal control, Mir, Moscow.
[36] Rosenbrock, H., 1960, An automatic method for finding the greatest or least value of a function, The computer journal, 3(3), 175-184.
[37] Schaffer, J., D., 1984, Multiple objective optimization with vector evaluated genetic algorithms, In G.J.E Grefensette; J.J. Lawrence Erlbraum (eds.), Proceedings of the first international conference on genetic algorithms.
[38] Schumer, M.A., Steiglitz, K., 1968, Adaptive step size random search, IEEE transactions on automatic control, 13, (3), 270–276.