




Self-adaptation Method for Evolutionary Algorithms Based on the Selection Operator

Pavel Sherstnev^(✉) 

Artificial Intelligence Laboratory, Siberian Federal University, Krasnoyarsk, Russia
sherstpasha99@gmail.com

Abstract. Genetic algorithms are a class of effective and popular black box optimization methods that are inspired by evolutionary processes in the nature. Genetic algorithms are useful in cases where nothing is known about the optimization object except the inputs and outputs. Such an algorithm iteratively searches for a solution in the solution space based on a predefined fitness function that allows comparing different solutions. If a researcher desires to use genetic algorithms, it becomes necessary to choose genetic operators and numerical parameters of the algorithm, the choice of which may be a difficult task. Self-adaptation methods that alter the behavior of the algorithm while it is running help to deal with the task of choosing the optimal settings of the algorithm. Such methods are called methods of self-adaptation of evolutionary algorithms and are usually divided into self-tuning, which performs the tuning of numerical parameters, and self-configuring, which makes the choice of genetic operators. In recent decades, various strategies for self-adaptation of evolutionary algorithms have been actively developed, including metaheuristic algorithms, as a result of which a researcher can obtain a specialized evolutionary algorithm that solves problems from a certain class better than conventional algorithms. However, even when using the metaheuristic approach, there is a need to choose genetic operators and numerical parameters of the algorithm. Therefore, the subject of the development of self-adaptive algorithms is one of the most relevant fields in the study of evolutionary algorithms. In this paper, a new approach to the adaptation of evolutionary algorithms based on the selection of genetic operators is proposed. The method is applied to the genetic algorithm and compared with the most popular SelfCGA self-configuring approach and shows an improvement in efficiency on both real and binary optimization problems.

Keywords: Self-adaptation · Evolutionary algorithms · Optimization · Genetic algorithm · Selection

1 Introduction

Many practical problems do not have a well-developed analytical solution either because of the frequent appearance of new problems, or because of the absence of a mathematical representation. An example can be any problems from the reinforcement learning class in which there is only a reward function, but there is no mathematically described object that is required by gradient optimization methods. The way out for the researcher

in such a situation can be evolutionary algorithms, since for their work it is enough to define the function of mapping the genotype into a phenotype and a function of evaluating the solution (a fitness function). However, in practice, if a researcher wants to use evolutionary algorithms, he also needs some experience working with them, since for any evolutionary algorithm there are many different genetic operators on which the result of optimization depends. The researcher can try all the operators in order to find the best operator for a given problem, but when a new problem appears, this choice may not be optimal, and again it will be necessary to search for a good combination of operators. Therefore, various strategies for self-adaptation of evolutionary algorithms have been developed for a long time [1–3], and sometimes even automatically creating new genetic operators that would provide solutions to problems from a certain class better than conventional ones [4, 5].

However, despite the success of using metaheuristic approaches, the subject of developing methods of self-adaptation of evolutionary algorithms is important, since even in the metaheuristic approach there is a problem of choosing the optimal operators of the evolutionary algorithm, which stands at the top level. Self-adaptation methods are usually divided into two categories: self-configuring methods, which include methods that select the types of operators in the process of the algorithm; self-tuning methods, which are understood as methods for adjusting the real parameters of the algorithm, for example, the probability of mutation. As a rule, each type of evolutionary algorithm has its own set of self-adaptation methods. For example, adaptation algorithms based on the history of successful applications have been developed for the differential evolution method [6–8]. For the genetic algorithm and the genetic programming algorithm, self-configuring methods have been developed that adapt the probabilities of using genetic operators in the process of the algorithm [1] and self-tuning methods that set the behavior of the probability of mutation [2]. The method of self-configuring of evolutionary algorithms described in [1] is the most effective and utilized self-configuring method that can be used to select operators of any evolutionary algorithm. However, this method implies only the choice of operators, but not the numerical parameters of the algorithm.

In this paper, we propose a simple method of self-adaptation of both genetic operators and numerical parameters of an evolutionary algorithm, based on existing selection methods. The method is compared with SelfCGA using the genetic algorithm for solving binary and real-valued optimization problems.

2 Methods of Self-adaptation for Genetic Algorithm

2.1 Genetic Algorithm

Genetic algorithms (GA) are a family of evolutionary optimization algorithms that search for a solution in a hypercube defined by a binary string in which an individual is encoded. The goal of GA is to determine which point of a given hypercube delivers the extremum of the target function. Each of the individuals in the population is one of the points in the whole space of valid solutions. The GA cycle will be described step by step below [9].

The first population is initialized randomly uniformly throughout the search space. Then, using selection, crossover, and mutation operators, the individuals evolve. At different points in the search space, the fitness value of an individual varies.

The selection process, based on the fitness value, selects the individuals that, using crossover operators, will create the offspring for the next generation. It is usually accepted to distinguish the following variants of the selection operator: proportional selection - probability of choosing an individual is proportional to its value of the fitness function; ranked selection - probability of choosing an individual is proportional to the rank of its value of the fitness function; tournament selection - the individual with the best value of the fitness function from a randomly formed subgroup is chosen.

The individuals selected at the previous stage participate in the crossover operation. The purpose of crossover is to provide useful information to the offspring. There are three types of crossover: one-point crossover - parents' chromosomes are randomly divided into two segments, and parents exchange them; two-point crossover - parents' chromosomes are randomly divided into three segments, and parents exchange them; uniform crossover - each bit in the offspring chromosome is inherited randomly from one of the parents.

The final step is mutation. The mutation operator usually makes small random changes in the genotype, thus maintaining diversity in the population. There is only one mutation operator in GA. Only the probabilities with which this mutation is carried out differ. When the mutation operator is over, the GA cycle is repeated again, and selection operators are again applied to the offspring.

If you use GA to work with real or integer variables, each variable must be encoded in a binary form. The easiest way is to divide the number into intervals and make a relation between each interval and the binary combination. In [10] it was found that the most efficient method of encoding real variables into binary ones is Gray code, although in a study [11] it was shown when using a genetic algorithm with varying string length the most efficient way of encoding may be Rice code.

2.2 Fitness-Based Adaptation

A simple method for tuning mutation and crossover probabilities was proposed in [2]. The method is designed to both protect good solutions from destruction and explore the search space by mutating inefficient solutions. Crossover and mutation probabilities are determined for each individual separately and are calculated based on the fitness of those individuals. So, if the fitness of an individual is high, then the probability of crossover and mutation is low (for an individual with the highest fitness these probabilities are 0), but if the fitness of an individual is low, then the probability of crossover and mutation is high. This method has shown to improve the performance on a number of problems of different classes, including the traveling salesman problem and the optimization of neural network weight coefficients. In [12] this method was modified, which improved the performance of the genetic algorithm on the traveling salesman problem. In difference from the original method, for individuals with the highest fitness the probabilities of crossover and mutation are not zero, but are performed with some minimal probability. This increases the speed of convergence to the optimal solution.

2.3 Population-Level Dynamic Probabilities

This method is taken from [3] and is used to tune the probabilities of applying genetic operators (self- configuring). At the start of the algorithm start each genetic operator is assigned minimum probabilities of its application $p_{all} = 0.2/n$, where n is the number of operators of a certain type. Then for each operator the values are entered (formula 1):

$$r_i = \frac{success_i^2}{used_i}, \quad (1)$$

where $used_i$ is the number of uses of the given operator, $success_i$ is the number of successful uses of the operator when the fitness of the offspring exceeded the fitness of the parent. For each operator the probability of its use p_i is calculated by the formula 2:

$$p_i = p_{all} + \left[r_i \frac{1.0 - np_{all}}{scale} \right], \quad scale = \sum_{j=1}^n r_j. \quad (2)$$

The $success_i$ values are squared because of very low success rate of genetic operators. The scale values allow to normalize the probability values so that their sum is always equal to one. Among the shortcomings of the *PDP* method it is should be noted that the procedure for evaluating the success of an operator has some disadvantages, in particular, when comparing the succession fitness with that of the parents it is not obvious with which of the parents to compare.

2.4 Self-configuring Genetic Algorithm

The *SelfCEA* method is based on increasing the probability of choosing the operator that provided the highest average fitness on a given generation. Let z_p be the number of different operators of type k . The initial selection probability of each operator is defined as $p_i = 1/z$. The probability of an operator providing the maximum mean value of the fitness function is increased by the formula 3 [1]:

$$p_i = p_i + \frac{(z - 1)K}{zN}, \quad i = 1, 2, \dots, z, \quad (3)$$

where N is the number of generations of algorithm, K is constant controlling rate of change of probability. Probabilities of other operators are recalculated so that the total sum equals 1. The method has two parameters: K , which determines the rate of change in the probability; a threshold which is the minimum probability of using the operators. The method has shown good results in solving the problem of selecting efficient options for the spacecraft control system [13], and has also been modified with new crossover operators in [14] and [15]. In this paper, this method is applied to a genetic algorithm and denoted as *SelfCGA*.

2.5 SHAGA

This algorithm is a successful attempt to apply a strategy for real parameter adaptation of the differential evolution algorithm, called success history based parameter adaptation

(SHA) [7], to the genetic algorithm. To be able to apply this adaptation strategy, the genetic algorithm itself had to undergo several modifications that made the cycle of the genetic algorithm closer to the cycle of differential evolution. Tournament selection with size of two was used as the selection operator. Only one parent was selected for crossover and the second parent was always the current individual in the population. Crossover was performed uniformly, where the crossover probability was tuned similarly to the SHADE algorithm. The method was tested on binary and real-valued optimization problems and achieved better performance than SelfCGA [16].

3 Proposed Approach

The self-configuring approaches discussed above, such as SelfCGA and PDP, perform a selection of genetic operators based on their efficiency during the execution of the algorithm. The operators are first generated with equal probability, and then the probabilities are recalculated in favour of the more efficient ones, depending on the fitness achieved by executing these operators. Ultimately, these self-configuring methods perform the same function: they select genetic operators based on probabilities that depend on the values of the fitness functions.

SelfCGA and PDP algorithms have disadvantages when used to tune a genetic algorithm. SelfCGA has two parameters: the rate of change in probability and a threshold probability that can be assigned to an operator. For different problems and numbers of operators, these parameters must be different. In addition to this, the method is a self-configuring method, which by definition and in practice does not allow for the setting of numerical parameters such as the mutation probability. With SelfCGA, mutation probability has three values (weak, average and strong mutation), which allows you to adjust these parameters, but only to a pre-defined value.

The PDP method, on the other hand, adjusts the probabilities of applying operators not directly on the fitness basis but by creating a progeny with a higher fitness than the parent. This peculiarity creates complications because it is not clear with which parent the offspring should be compared. The method also allows only the genetic operators to be set, but not the numerical values of the algorithm parameters.

Researchers usually develop complex procedures for calculating probabilities based on fitness values, but in evolutionary algorithms there are already functions that assign a selection probability to each fitness function value from the population. These are selection operators.

If individuals were generated using different genetic operators in the previous population, a fitness value can be calculated for each individual, and the information about the type of operator that was used can be used. By having sets of fitness function values and different types of genetic operators that generated individuals with the corresponding fitness values, each time the next generation of individuals is created, the selection operator can be used to select the operator that will produce the offspring.

Let $operator_set$ be the set of all genetic operator types of a particular type. N is population size. On the first generation, when the i -th offspring is created, the genetic operator is chosen randomly from this set (formula 4):

$$Operator_i = RandomChoice(operator_set), \quad (4)$$

where *RandomChoice* is a function for selecting a random item from a set. The probability of each element being selected is equal. *Operator_i* is used to create an offspring and *Fitness_i* fitness of the offspring is associated with this operator. Once the new generation is fully formed, there is a set of values (formula 5):

$$OperList = \begin{bmatrix} Operator_1 & Fitness_1 \\ \vdots & \vdots \\ Operator_N & Fitness_N \end{bmatrix}. \quad (5)$$

Now, when creating the *i*-th offspring, the operator is selected as follows (formulas 6 and 7):

$$\text{If } rand < p_i: Operator_i = RandomChoice(operator_set) \quad (6)$$

$$\text{Else: } Operator_i = Selection(OperList) \quad (7)$$

where *rand* is a random value generated by a uniform distribution in the range 0 to 1, *p_i* is the probability with which an operator is randomly selected from the set of possible *operator_set* variants, *Selection* is a genetic selection operator that selects an individual (in this case another genetic operator from *OperList* based on the values of the fitness function).

The condition in formula 3 ensures the robustness of the adaptation process by ensuring that different operators are present in *OperList*, creating permanent competition.

Now consider the procedure for adapting a numerical parameter. Let *Left* be the minimum value of the parameter and *Right* be the maximum value of the parameter. *N* - population size. In the first generation, when the *i*-th offspring is created, the parameter is generated by a uniform distribution (formula 8):

$$Parametr_i = Uniform(Left, Right) \quad (8)$$

where *Uniform* is the function that generates a random value in the *Left* and *Right* boundaries.

Parametr_i is used to create a descendant and fitness *Fitness_i* per offspring is associated with this operator. Once the new generation is fully formed, there is a set of values (formula 9):

$$ParamList = \begin{bmatrix} Parametr_1 & Fitness_1 \\ \vdots & \vdots \\ Parametr_N & Fitness_N \end{bmatrix} \quad (9)$$

Now, when creating the *i*-th offspring, the operator is selected as follows (formulas 10 and 11):

$$\text{If } rand < p_i: Parametr_i = Uniform(Left, Right) \quad (10)$$

$$\text{Else: } Parametr_i = Selection(ParamList) \quad (11)$$

Thus, the proposed method allows to tune both the numerical parameters of the algorithm and the genetic operators, which makes it a self-adaptive method. In future work, this method will be referred to as SelfAGA. It is easy to see that there can be as many variations of this method as there are selection functions and considering that a selection function can be generated using metaheuristics, this method can be used for automated generation of self-adaptive evolutionary algorithms.

In the next section, we compare the proposed method with SelfCEA self-configuring method for solving binary and real-valued optimization problems by genetic algorithm.

4 Experimental Setup and Results

4.1 Parameters of the Genetic Algorithm

A genetic algorithm was implemented in the python programming language and then modified to match each method. The SelfCGA and SelfAGA algorithms use the same implementations of genetic operators and the other genetic algorithm process is programmed in the same way, except for the self-adaptation method. Thus the difference in algorithm performance can only be due to different adaptation methods. Table 1 below shows the algorithm parameters and operators involved in adaptation.

Table 1. Parameters of the genetic algorithm and self-adaptation methods

Name of the parameter	Value
type of crossover	one-point, two-point, uniform
type of selection	proportional, rank, tournament (5)
type of mutation (SelfCGA)	weak, average, strong
K (SelfCGA)	0.5
threshold (SelfCGA)	0.05
P_t (SelfAGA)	0.1
Selection (SelfAGA)	proportional, rank, tournament (3 and 5)

The average mutation performs a bit flip with a probability that is calculated as follows (formula 12):

$$p = \frac{1}{StringLen} \quad (12)$$

where *StringLen* is the length of the binary string. Weak and strong mutations are three times less and three times more likely, respectively.

Since in the SelfAGA method the probability is adjusted as a real parameter, the minimum value is the probability determined in the weak SelfCGA mutation, and the maximum value is the probability determined in the strong mutation. The remaining part of the paper is a comparison of SelfCGA with SelfAGA, with SelfAGA presented in four versions, distinguished by the selection function used for self-adapted parameters (proportional, rank and tournament with size of 3 and 5).

4.2 Results of Solving Binary Optimization Problems

A total of two binary problems are used: the problem “onemax”, which contains one global optimum and consists in obtaining the maximum number of 1 in the binary string. The optimum in this problem is reached when all elements of the binary string are 1.

Another problem is called “01”. The goal is to find the binary string with the maximum number of pairs 0 and 1. Unlike “onemax”, this problem has one global and one local optimum. For both problems “onemax” and “01” there are different variations, varying in the dimensionality of the problem, as well as the number of generations and population size. Table 2 below shows the parameters of the problems.

Table 2. Parameters of binary optimization problems

Problem	Number of iterations	Population size
Onemax 1000 bits	100	100
Onemax 3000 bits	100	100
Onemax 10000 bits	1000	100
«01» 1000 bits	100	100
«01» 3000 bits	100	100

Table 3 below shows the best solutions achieved by each of the self-adaptation methods, averaged over 1000 independent runs.

Table 3. Results of solving binary optimization problems.

Heading level	SelfCGA	SelfAGA (proportional)	SelfAGA (rank)	SelfAGA (tournament 3)	SelfAGA (tournament 5)
Onemax 1000 bits	891.075	899.699	900.404	905.372	914.516
Onemax 3000 bits	2204.902	2211.933	2207.496	2227.298	2264.675
Onemax 10000 bits	8444.079	8601.686	8562.07	8583.36	8662.617
«01» 1000 bits	402.348	405.309	405.238	404.676	404.942
«01» 3000 bits	1035.684	1035.135	1034.647	1035.198	1038.67

For statistical verification, the Mann-Whitney test is used with a significance level of 0.01. In the case of Null hypothesis there are no differences between observations. Table 3 shows the values in bold for the methods whose difference between the results was statistically significant compared to the SelfCGA results.

4.3 Results of Solving Real-Valued Optimization Problems

The second part of the experiments consists of solving real optimization problems, namely 10 functions from [17]. The number of iterations and the size of the iteration were chosen for each problem individually. Table 4 below shows the functions from [17] as well as the parameters under which the functions have been optimized. Only a part of the original set of functions is used due to the fact that the genetic algorithm could not find an optimal solution for a limited number of iterations (the test calculations were performed up to a number of iterations of 1000 and a generation size of 1000).

Table 4. Parameters of real-valued optimization problems.

Problem	Number of iterations	Population size
F1	30	30
F2	45	45
F4	45	45
F5	100	50
F6	1000	500
F9	100	50
F10	300	200
F12	70	50
F15	100	50
F16	1000	500

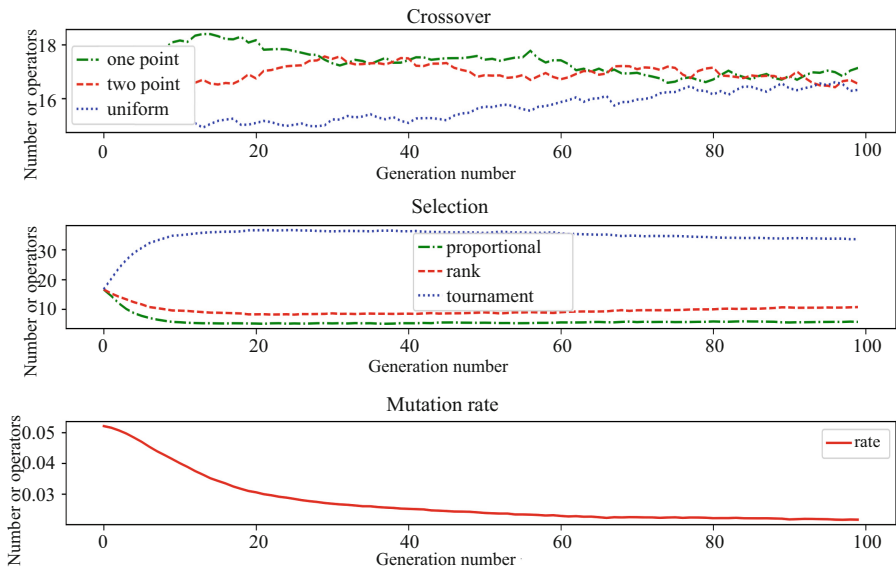
The reliability criterion for the genetic algorithm is the ratio of optimal solutions found out of all runs of the algorithm (reliability). For genotype-phenotype mapping a string of bits to real values Gray code and 16 bits per variable were used. Table 5 below shows the average reliability of algorithms averaged over 1000 runs.

The highest values achieved on a given problem are shown in bold. It can be noticed that the average reliability for all the problems solved by the new self-adaptation method surpassed the reliability achieved by the SelfCGA method. The maximum average reliability was achieved using rank selection, while using proportional selection did not significantly improve the reliability of the genetic algorithm on the given tasks. Consider as an example the process of adaptation of the genetic algorithm parameters on the F5 problem. The results are averaged over 1000 runs. Figure 1 below shows the number of various crossover operators (upper plot), selection (middle plot) and the average value of mutation probability (lower plot).

The graph shows how these values adapt over the running of the algorithm.

Table 5. Results of solving real-valued optimization problems.

Problem	SelfCGA	SelfAGA (proportional)	SelfAGA (rank)	SelfAGA (tournament 3)	SelfAGA (tournament 5)
F1	0.740	0.658	0.913	0.927	0.939
F2	0.757	0.801	0.878	0.883	0.864
F4	0.763	0.762	0.834	0.829	0.805
F5	0.621	0.468	0.637	0.601	0.568
F6	0.744	0.988	0.799	0.791	0.759
F9	0.740	0.751	0.697	0.681	0.666
F10	0.690	0.694	0.785	0.769	0.787
F12	0.630	0.548	0.747	0.732	0.724
F15	0.726	0.735	0.701	0.680	0.679
F16	0.554	0.610	0.834	0.878	0.914
mean	0.6965	0.7015	0.7825	0.7771	0.7705

**Fig. 1.** The process of adaptation of the parameters of the genetic algorithm

5 Conclusion

In this paper, a method for self-adaptation of evolutionary algorithms is proposed and tested that differs from the known methods of parameter selection and the possibility of tuning both genetic operators and numerical parameters of the algorithm. The proposed algorithm can be presented in different variants depending on the selection function that performs algorithm parameter adaptation in the process. The method is compared with the popular SelfCGA approach and shows better performance on both binary and real-valued optimization problems. Further work will be done to study the effect of pt on the efficiency of the method and investigate the performance of the algorithm as compared to other self-adaptation methods on a wider set of problems.

Acknowledgments. This work was supported by the Ministry of Science and Higher Education of the Russian Federation (Grant № 075-15-2022-1121).

References

1. Semenkina, M.: Self-adaptive evolutionary algorithms for designing information technologies for data mining. *Artif. Intell. Decis.-Making* **1**, 12–24 (2012)
2. Patnaik, L.: Adaptive probabilities of crossover and mutation in genetic algorithms. *IEEE Trans. Syst. Man Cybern.* **24**, 656–667 (1994)
3. Niehaus, J., Banzhaf, W.: Adaption of operator probabilities in genetic programming. In: Miller, J., Tomassini, M., Lanzi, P.L., Ryan, C., Tettamanzi, A.G.B., Langdon, W.B. (eds.) *EuroGP 2001*. LNCS, vol. 2038, pp. 325–336. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-45355-5_26
4. Richter, S.: The automated design of probabilistic selection methods for evolutionary algorithms. In: *Proceedings of the 2018 Genetic and Evolutionary Computation Conference Companion*, pp. 1545–1552 (2018)
5. Hong, L., Woodward, J., Özcan, E., Liu, F.: Hyper-heuristic approach: automatically designing adaptive mutation operators for evolutionary programming. *Complex Intell. Syst.* **7**, 3135–3163 (2021). <https://doi.org/10.1007/s40747-021-00507-6>
6. Zhang, J., Sanderson, A.C.: JADE: adaptive differential evolution with optional external archive. *IEEE Trans. Evol. Comput.* **13**(5), 945–958 (2009)
7. Tanabe, R., Fukunaga, A.: Success-history based parameter adaptation for differential evolution. In: *Proceedings of 2013 IEEE Congress on Evolutionary Computation*, pp. 71–78 (2013). <https://doi.org/10.1109/CEC.2013.6557555>
8. Tanabe, R., Fukunaga, A.: Improving the search performance of SHADE using linear population size reduction. In: *Proceedings of the 2014 IEEE Congress on Evolutionary Computation*, pp. 1658–1665 (2014)
9. Holland, J.: *Adaptation in Natural and Artificial Systems*. MIT Press, Cambridge (1975)
10. Whitley, L.D.: Free lunch proof for gray versus binary encoding. In: *Genetic and Evolutionary Computation Conference* (1999)
11. Panfilov, I.: Study of the performance of a genetic optimization algorithm with alternative solution representation. *Siberian Aerosp. J.* **4**(50), 68–71 (2013)
12. Han, S., Xiao, L.: An improved adaptive genetic algorithm. In: *SHS Web of Conferences*, vol. 140, pp. 5–6 (2022)

13. Semenkin, E., Semenkina, M.: Spacecrafts' control systems effective variants choice with self-configuring genetic algorithm. In: Proceedings of 9th International Conference on Informatics in Control, Automation and Robotics, pp. 84–93 (2012)
14. Semenkin, E., Semenkina, M.: Self-configuring genetic programming algorithm with modified uniform crossover. In: Proceedings of 2012 IEEE Congress on Evolutionary Computation, CEC 2012 (2012)
15. Semenkin, E., Semenkina, M.: Self-configuring genetic algorithm with modified uniform crossover operator. In: Tan, Y., Shi, Y., Ji, Z. (eds.) ICSI 2012. LNCS, vol. 7331, pp. 414–421. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30976-2_50
16. Stanovov, V., Akhmedova, S., Semenkin, E.: Genetic algorithm with success history based parameter adaptation. In: Proceedings of 11th International Conference on Evolutionary Computation Theory and Applications, pp. 180–187 (2019)
17. Suganthan, P.N., et al.: Problem definitions and evaluation criteria. In: Proceedings of CEC 2005 Special Session on Real-Parameter Optimization (2005)