ISSN 1064-2307, Journal of Computer and Systems Sciences International, 2015, Vol. 54, No. 4, pp. 525–539. © Pleiades Publishing, Ltd., 2015.
Original Russian Text © V.A. Kostenko, A.V. Frolov, 2015, published in Izvestiya Akademii Nauk. Teoriya i Sistemy Upravleniya, 2015, No. 4, pp. 24–38.

COMPUTER METHODS

Self-Learning Genetic Algorithm*

V. A. Kostenko and A. V. Frolov

Moscow State University, Moscow, 119991 Russia

e-mail: AlexFrolov1878@gmail.com

Received November 28, 2014

Abstract—In this paper, a self-learning genetic algorithm for solving combinatorial optimization problems is considered. The self-learning consists in changing the values of the probabilities of crossing and mutation in accordance with changing the value of the fitness function after operations in the next iteration of the algorithm. The results of comparing the proposed algorithm with the Holland algorithm by the problems of multiprocessor job scheduling and subset sum problem are presented.

DOI: 10.1134/S1064230715040103

INTRODUCTION

Genetic algorithms (GAs) are used for their efficiency to solve problems of the construction of optimal game strategies, configuring and training of neural networks, query optimization in databases, scheduling [1, 2], and machine learning [3], as well as to solve various tasks on graphs (coloring, the traveling salesman problem, and finding matchings), etc. However, in developing a GA for the solution of a particular task, it is necessary to solve several problems [1]. One of them is choosing the values of the probabilities of the mutation and crossover (these values have a profound effect on the efficiency of the algorithm). At present, no theoretical results that make it possible to solve this problem have been found. In most practical applications of GAs, the parameter values of the operations of crossover and mutation are chosen experimentally.

Self-learning random-search algorithms for solving problems of continuous mathematical programming were proposed by L.A. Rastrigin in the 1960s [4]. In the work, such algorithms are adapted to the relief of the target function. The basis of random-search methods is the iterative process

$$X^{k+1} = X^k + \alpha_k \frac{\varepsilon}{\|\varepsilon\|}, \quad k = 1, 2, \ldots,$$

where X^k is the vector of optimized parameters after the kth step, α_k is a certain real coefficient, and ε is an equiprobable random vector.

Self-learning algorithms for a random directed search suggest the restructuring of the probabilistic characteristics of the search, i.e., a specific purposeful action on the random vector ε. It ceases to be equiprobable and (as a result of the self-learning) acquires a certain advantage in the directions of the best steps. This is achieved by introducing a memory vector. The algorithm recurrently corrects the values of the components of this vector in each iteration according to the measure of the success or failure of a step.

The present paper suggests the introduction of the probability matrices of mutation and crossover into the classical GA and an algorithm for changing the values of the elements of these matrices according to the measure of success of applying a corresponding operation to a certain element of the solution.

1. GENETIC ALGORITHM WITH PARTICULAR PARAMETERS OF PROBABILITIES OF MUTATION AND CROSSOVER

1.1. Scheme of Algorithm

In the Holland classical GA [5], the probabilities of mutation and crossover are assigned, and the probabilities at hand serve as the probabilistic thresholds of these operations. In each iteration of the classical GA, the same values of these parameters are used. Each element S of a population represents a coded solution row X:

$$S = X, \quad X = (x_1 x_2 \ldots x_N).$$

* This work was financially supported in part by the Ministry of Education and Science of the Russian Federation, state contract no. 14.607.21.0070.

The idea of a self-learning GA consists in introducing particular parameters of the probability of mutation and crossover for each element x_i of a solution row and in the subsequent correcting of these elements in each iteration of the algorithm according to the measure of success or failure of applying a particular operation (mutation or crossover) to an element of the solution.

The matrices M_{mut} and M_{cr} of the probabilities of the mutation and crossover are $N \times P$ matrices, where N is the number of optimized parameters (usually it is identical to the number of the elements of a solution row) and P is the population size. The elements of these matrices are the probabilities of the mutation and crossover for each ith element x_i of the solution in the jth row of the population:

$$M_{mut} = (m^{mut}_{i,j})^{N,P}_{i=1,j=1}, \quad M_{cr} = (m^{cr}_{i,j})^{N,P}_{i=1,j=1}.$$

The general scheme of a self-learning GA can be represented as follows.

1. Randomly generate the original population.

2. Compute the fitness function for each row of the population.

3. Select and permute rows of the matrices of mutation and crossover.

4. Perform the operation of crossover.

4.1. Choose pairs for crossing.

4.2. For each chosen pair, do the following:

(1) perform (with the assigned probability) crossover and obtain two descendants;

(2) correct the values of the corresponding elements of the probability matrix of crossover and permute the elements of the rows corresponding to the crossed solutions;

(3) replace parents by their descendants in the population.

5. Perform the operation of mutation and correct the value of a corresponding element of the probability matrix of mutation.

6. If the stop condition is not satisfied, then go to step 2; otherwise terminate the work.

Now, we describe the basic operations of a self-learning GA.

1.2. Mutation

Denote by $X = (x_1 x_2 \ldots x_n)$ parent solution having number j in the population. For elements x_i, the limitations $A_i \le x_i \le B_i$ are assigned. Each of the values x_i is an integer. We introduce the mutation's degree S_m, which specifies the degree the values of elements of vector X are changed. The operator of mutation $\hat{X} = m(X, M_{mut}, S_m)$ generates a new solution $\hat{X} = (\hat{x}_1 \hat{x}_2 \ldots \hat{x}_n)$ according to the following scheme:

(i) for each $i = \overline{1, n}$, a random number $r_i \in (0, 1)$ is chosen in accordance with the uniform distribution law;

(ii) the new solution is found by the formula

$$\hat{x}_i = \begin{cases} ms(x_i, S_m), & r_i \le m^{mut}_{i,j}, \\ x_i \quad \text{otherwise}. \end{cases}$$

Here, $ms(x_i, S_m)$ is the function of mutation of an integer value; this function mutates particular elements of the solution vector X as follows [2]:

(1) x_i is represented as a bit string; in this case, the minimum number of bits required for this purpose is found: $b_i = \log_2 \lceil B_i - A_i \rceil$, where by A_i and B_i are denoted the lower and upper bounds of x_i, respectively, while the minimum integer greater than or equal to x is denoted by $\lceil x \rceil$;

(2) x_i is represented as

$$x_i = A_i + \sum_{k=1}^{b_i} \alpha_k 2^k,$$

where α_k is the kth bit of the string;

(3) the maximum number of mutating bits is found: $n_i = [(b_i - 1)(S_m + 0.5)]$;

(4) in accordance with the uniform distribution law, a random number l_i ($1 \le l_i \le b_i$) is taken and the bit having number l_i is inverted in the bit string: $\alpha_{l_i} = 1 - \alpha_{l_i}$; this step is repeated n_i times;

(5) the preliminary result t_i of the function of mutation is found using the mutated bit string:

$$t_i = A_i + \sum_{k=1}^{b_i} \alpha_k 2^k;$$

(6) the result is corrected, since it is necessary to fall into the interval $A_i \le \hat{x}_i \le B_i$:

$$ms(x_i, S_m) = \begin{cases} t_i + B_i - A_i, & t_i < A_i, \\ t_i, & A_i \le t_i \le B_i, \\ t_i - (B_i - A_i), & t_i > B_i. \end{cases}$$

Hence, the operator of mutation $\hat{X} = m(X, M_{mut}, S_m)$ indeed depends on two parameters, namely, M_{mut} representing the probability matrix of the mutation and $S_m > 0$ representing the degree of mutation (the greater this number, the higher the probability that the changes in elements of the parent solution will be large).

1.3. Crossover

Denote by $X = (x_1 x_2 \ldots x_n)$ and $Y = (y_1 y_2 \ldots y_n)$ parent solutions chosen for the crossover and having in the population the numbers p and q, respectively. Here, x_i and y_i are certain integer values. The parameters P_{MIN} and P_{MAX} specify the thresholds of the probability of the crossover that are responsible for various versions of the exchange fragments of rows. The operator of crossover $cr(X, Y, M_{mut}, P_{MIN}, P_{MAX})$ generates two new solutions $\hat{X} = (\hat{x}_1 \hat{x}_2 \ldots \hat{x}_n)$ and $\hat{Y} = (\hat{y}_1 \hat{y}_2 \ldots \hat{y}_n)$ according to the following scheme:

1. In accordance with the uniform distribution law, a random number k ($1 \le k \le n$) is chosen.

2. The average probabilities of crossover of the right sides of the rows are calculated:

$$p_x^{cr} = \frac{m_{k,p}^{cr} + \ldots + m_{n,p}^{cr}}{n - k + 1}, \quad p_y^{cr} = \frac{m_{k,q}^{cr} + \ldots + m_{n,q}^{cr}}{n - k + 1}.$$

3. For the parent solutions, a random number r with the uniform distribution law is chosen.

4. Elements of the new solution are found by the following formulas:

if $p_x^{cr} > P_{MAX}$ and $p_y^{cr} < P_{MIN}$, then

$$\hat{x}_i = x_i,$$

$$\hat{y}_i = \begin{cases} y_i, & i \le k, \\ x_i, & i > k; \end{cases}$$

if $p_x^{cr} < P_{MIN}$ and $p_y^{cr} > P_{MAX}$, then

$$\hat{x}_i = \begin{cases} x_i, & i \le k, \\ y_i, & i > k, \end{cases}$$

$$\hat{y}_i = y_i;$$

if $r < p_x^{cr}$ and $r < p_y^{cr}$, then

$$\hat{x}_i = \begin{cases} x_i, & i \le k, \\ y_i, & i > k, \end{cases}$$

$$\hat{y}_i = \begin{cases} y_i, & i \le k, \\ x_i, & i > k. \end{cases}$$

The operator of the crossover depends on one parameter, namely, M_{cr} representing the probability matrix of the crossover.

Since the operation of crossover permits optimized parameters x_i among strings of the population, the corresponding permutations after performing the operation must also be made in the probability matrices of mutation and crossover.

Let us describe several methods for correcting the elements of probability matrices; these methods are used in self-learning random-search algorithms [4].

1.4. Absolute Method for Correcting Probability Matrices of Mutation and Crossover

We assign the parameter δ of varying the values of elements of the probability matrices of mutation and crossover; this parameter specifies the speed of their reconstruction. The values of the elements of the probability matrices vary in the course of operation of a GA as follows (thereby the mechanism of self-learning is implemented).

If after the operation of mutation of the *i*th solution's element having number *j* in the population, the value of the fitness function decreases, then element $m_{i,j}^{mut}$ of the probability matrix of mutation (this element corresponds to the probability of mutation of this element of the solution) increases by δ; otherwise, it decreases by δ. In this case, the values of $m_{i,j}^{mut}$ must lie in the range from 0.1 to 0.9.

The elements $m_{i,j}^{cr}$ and $m_{k,l}^{cr}$ of the probability matrix of crossover, which specify the probabilities of crossover for the *i*th and *k*th solution's elements in the rows having numbers *j* and *l*, respectively, vary analogously to the case of mutation. In this case, their values must lie in the range from 0.1 to 0.9.

In contrast to the classical GA, this method assigns the values of parameters of mutation and crossover specifically for each solution's element. Self-learning is conducted in two contradictory modes: the promotion of an operation (for $\Delta F < 0$) and the punishment of an operation (for $\Delta F > 0$), where ΔF is the difference between the values of the fitness function before and after performing the operation. Both modes increase the probability of a favorable step of the self-learning GA.

1.5. Relative Method for Correcting Probability Matrices of Mutation and Crossover

In the above-considered method for correcting the values of elements of the probability matrices of mutation and crossover, with any change in the fitness function, the values change by the constant equal to δ. However, the need for correcting a particular element of the probability matrices depends primarily on the degree of change in the fitness function after the operation on the solution's element corresponding to this particular element. Therefore, it is often more reasonable to use the relative self-learning algorithm.

Let us find the changes in the values of elements of the probability matrices analogously to the case of the absolute method for correcting elements of the probability matrices; however, here instead of assigning the fixed parameter δ we use the difference $df = \bar{F} - F$ between the values of the fitness function before (F) and after ($\bar{F}$) performing a particular operation. Assume that $m_{i,j}$ is an element of the probability matrix of

mutation or crossover and this element corresponds to the ith solution's element in the row having number j. Then,

$$t = m_{i,j} + df, \quad m_{i,j} = \begin{cases} c_1, & t < c_1, \\ t, & c_1 \le t \le c_2, \\ c_2, & t > c_2. \end{cases}$$

The constants c_1 and c_2, which limit the change in the elements of the memory vector, exclude redetermining the search in the course of self-learning.

With the introduction of such dependence, the self-learning process of a GA becomes more sensitive to changes in the quality of a solution. In fact, at a small change in the fitness function, the corresponding probabilities of the operations of mutation and crossover also have minor changes, and vice versa.

1.6. Relative Method for Correcting Probability Matrices of Mutation and Crossover with Forgetting

The above-considered algorithms for correcting elements of the probability matrices memorize and store the entire previous experience of a GA. Evidently, there is no need for this. Moreover, changing the operating conditions of the GA (approaching the global extremum of an objective function or moving away from this extremum) requires sufficiently fast forgetting of information on the fitness function's changes determined before, because this information is obtained under other conditions and is now out of date. Therefore, it is reasonable to introduce forgetting into the learning algorithm.

Let us determine the changes in the values of the elements of the probability matrices of mutation and crossover analogously to the case of the relative method for correcting a memory vector. We introduce parameter s of memorizing the results of previous steps of a self-learning GA:

$$t = sy_{m_{i,j}} + df, \quad y_{m_{ij}} = \begin{cases} c_1, & t < c_1, \\ t, & c_1 \le t \le c_2, \\ c_2, & t > c_2. \end{cases}$$

Note that choosing the value of the memory parameter s defines the speed of reconstructing the probability matrices of mutation and crossover in accordance to approaching (moving away from) extremums with the space of the search. With the values of s close to one, forgetting is almost absent. At the same time, with the small values of this parameter, the probability of the determination of a local extremum increases.

1.7. Selection

The operation of selection allows generating a new population in the next iteration of the algorithm, from the strings obtained at steps 4 and 5. For determining the integer number of the descendants and distributing the remainder, the proposed algorithm uses the schemes of proportional selection and a roulette, respectively. Here, the parameter N_{best} is added to the algorithm; this parameter specifies the number of exemplars of the best string that necessarily remains in the population [2]. The scheme of performing the operation can be presented as follows:

1. At the stage of determining the fitness function, the string of the population s'_{best} with the best value of the fitness function F'_{best} is chosen; the value of the fitness function is compared with the value of the fitness function of the best string in previous iterations F_{best}. If $F'_{best} > F_{best}$ or if this is the first iteration of the algorithm, then s'_{best} is memorized as the best string $s_{best} = s'_{best}$.

2. In the population, the number of strings identical to s_{best} is found and is memorized as N_{best_exist}.

3. By the scheme of proportional selection, the number of descendants for each string is found. If the number of strings in the new population is N_{pop}, where N_{pop} is the population size, then $(N_{best} - N_{best\_exist})$ the worst strings in the population are chosen (because the N_{best_exist} available best strings will always pass to the new population by the definition of the scheme of proportional selection) and are replaced by copies of the best string s_{best} (and the operation of selection is finished); otherwise go to item 4.

4. By the roulette scheme, $N_{pop} - N_{sel1} - (N_{best} - N_{best\_exist})$ the descendants are distributed, where N_{sel1} is the number of strings obtained by the scheme of proportional selection. Then $N_{best} - N_{best\_exist}$ best strings s_{best} are added to the population; if this value is negative, such an addition does not proceed.

The solution-coding method, the fitness function, and the stop condition are assigned specifically for each problem.

2. EXAMPLES OF SELF-LEARNING GENETIC ALGORITHMS

In this section we consider GAs for solving two problems: job scheduling and subset sum problem.

2.1. Problem of Building a Schedule with Minimum Execution Time for a Fixed Number of Processors

Model of application program *H*(*PR*) [1, 6–8] is represented by an acyclic oriented marked graph whose vertices $P = p_1 \cup p_2 \cup ... \cup p_N$ are matched by processes and arcs $\prec = \{\prec_{ik} = (p_i, p_k)\}_{(i,k) \in \overline{1,N}}$ are matched by connections that specify interactions among processes from set *P*.

Each vertex has its unique number and the mark (the computational complexity), which assigns the execution time of a given process t_i. An arc is defined by the numbers of adjacent vertices and has a mark corresponding to the amount of data exchange v_{ij}.

The model of application program *H*(*PR*) is assigned as follows:

(1) by the set of processes constituting program *PR*:

$$P = \{p_i\}_{i=1}^{N};$$

(2) by a partial order on *P* (on which the acyclicity and transitivity conditions are imposed):

$$\prec = \{\prec_{ik} = (p_i, p_k)\}_{(i,k) \in \overline{1,N}};$$

(3) by the computational complexities of processes from *P*:

$$\{t_i\}_{i=1}^{N};$$

(4) by the amounts of data exchange for each connection from the set $\prec$:

$$\{v_{ik}\}_{(i,k) \in \overline{1,N}}.$$

A schedule of the program execution [1] is defined if sets of processors and operating ranges, a binding, and an order are assigned. The binding is a completely defined function on the set of processes; this function assigns the distribution of the processes among the processors. The order specifies limitations on the sequence of executing the processes and represents a partial ordering relation on the set of processors; here, this relation satisfies the acyclicity and transitivity conditions. The order relation on the set of processes distributed on the same processor is a complete-ordering relation.

A model of schedule *HP* of the program execution [1] is defined by a set of simple chains and the partial order relation $\prec_{HP}$ on set *P*:

$$HP = (\{SP_i\}_{i=\overline{1,M}}, \prec_{HP}),$$

where $\{SP_i\}_{i=\overline{1,M}}$ is a set of simple chains (branches of a parallel program). They are produced by processes distributed on the same processor (*M* is the number of processors in the computer system). In the model of schedule *HP*, the numbers of vertices, arcs, and their marks remain the same as in the model of the behavior of program *H*(*PR*).

Schedule *HP* is called *feasible* if the following requirements are met for it:

(1) each process must be assigned to a processor (in SP_i):

$$\forall p_i \in P \Rightarrow \exists m: p_i \in SP_m,$$

(2) each process must be assigned only to one processor (only in one SP_i):

$$\forall p_i \in SP_j \Rightarrow \neg\exists m \neq j: p_i \in SP_m,$$

(3) the partial order assigned in H must remain in HP:

$\prec \subset \prec_{HP}^{T}$, where $\prec_{HP}^{T}$ is a transitive closure of the relation $\prec_{HP}$,

(4) schedule HP must be open-ended, i.e., the graph of HP cannot have any circuits:

$\prec_{HP}$ is acyclic.

The problem of building a schedule with minimum execution time is formulated as follows. Assume that one assigns the model of application program $H(PR) = (P, \prec)$, the number of processors M, and the function $T = f(M, HP)$ of calculating the execution time of schedule HP for M processors. It is necessary to build a correct schedule of executing program $HP = (\{SP_i\}_{i = \overline{1, M}}, \prec_{HP})$ for the assigned number of processors M such that $\min\limits_{HP} f(M, HP)$.

In the coding of a solution as a string in a GA [9], it is necessary to code the schedule. We point out two ways of representing a schedule: the direct representation (for each process, values of the binding and order number are assigned explicitly, in the integer form) and the parametric representation (for each process, the binding and order number are assigned by values of certain parameters from which the values of the binding and order number are determined using the algorithm for restoring a schedule).

It is suggested to use the parametric representation of a schedule, and for restoring the order we propose to apply the well-known algorithm for restoring a partial order by priorities; this algorithm is described in [1, 2]. Such a way of representing a schedule allows independently changing all parameters that assign the schedule; here, schedules obtained by operating remain correct and the third requirement of the correctness of a schedule is not violated.

The coding can be performed as follows:

$$S = Y_{task} \cup Y_{prio},$$

$$Y_{task} = \bigcup_{i = 1}^{N} \langle YE \rangle_i \text{ is binding},$$

$$Y_{prio} = \bigcup_{i = 1}^{N} \langle YP \rangle_i \text{ are priorities}.$$

Here, N is the number of processes in H, $\cup$ is the operator of concatenation of strings, and Y_{task} and Y_{prio} define the schedule.

The parameter $\langle YE \rangle_i \in \overline{1, M} \subset Z$ contains the number of a processor on which the ith process is performed; i.e., parameters $\langle YE \rangle_i$ uniquely define the distribution of processes among SP (the binding). The parameters $\langle YE \rangle_i$ can take values from 1 to M.

The parameters $\langle YP \rangle_i \subset Z$ are used by the algorithm for restoring a schedule (for the given representation of the schedule it means finding the complete-ordering relation in each SP_i) as priority processes.

We define the matrices of the probabilities of crossover and mutation for this problem as follows.

1. The matrix M_{mut} of the probability of mutation is a $2N \times P$ matrix that consists of the values of the parameters of mutation for each ith parameter of the solution in the jth row:

$$M_{mut} = (m_{i,j}^{mut})_{i = 1, j = 1}^{2N, P}.$$

In particular, the row $(m_{i,5}^{mut})_{i = 1}^{2N}$ of the matrix M_{mut} corresponds to the probabilities of mutation for $2N$ parameters of the binding and the priorities of the fifth population's solution.

2. The matrix M_{cr} of the probability of crossover is a $2N \times P$ matrix that consists of the values of the parameters of crossover for each ith parameter of the solution in the jth row:

$$M_{cr} = (m_{i,j}^{cr})_{i = 1, j = 1}^{2N, P}.$$

It is clear that minimizing the downtime of processors causes minimizing the time of executing a schedule for the processors. In this respect, for the fitness function we propose a linear combination of the following criteria:

(i) the collection of the downtime coefficients of the processors of a computer system (the ratio of the downtime of the processors to their total operating time): $\{r_i\}_{i=1}^{M}$;

(ii) the execution time of a schedule: $T = f(M, HP)$.

Note that occurrences of the values $\{r_i\}_{i=1}^{M}$ and T (which have different measuring units) in the fitness function must be normalized, because for various problems they can differ several times. Each downtime coefficient $\{r_i\}_{i=1}^{M}$ is already normalized in the range [0, 1]. In order to normalize T, we multiply it by the coefficient $1/(t_1 + ... + t_N)$, where $\{t_i\}_{i=1}^{N}$ are the computational complexities of processes.

Hence, the fitness function has the form

$$F = C_1K_t + C_2K_r, \quad C_1 + C_2 = 1, \quad C_i \geq 0, \quad i = 1, 2,$$

$$K_t = 1 - \frac{T}{t_1 + \ldots + t_N},$$

$$K_r = 1 - \frac{r_1 + \ldots + r_M}{M}.$$

In this case, the fitness function is bounded above by one:

$$F \leq (C_1 + C_2)\max\{K_t, K_r\} \leq C_1 + C_2 = 1.$$

For the stop condition, we propose using a limitation on the number of iterations of the algorithm without improving; i.e., the algorithm stops operating if in I_0 steps it cannot improve the value of the fitness function in the population.

2.2. Subset Sum Problem

For a subset sum problem [10], a set of weighting coefficients w_i is specified, each being a positive number. The aim of the problem is to construct a subset of weights such that the sum of its elements is closest to the given value G:

$$\min\left|\sum_{i=1}^{N} w_i x_i - G\right|, \quad x_i \in \{0, 1\}.$$

The weighting coefficients w_i are usually defined as the values of independent random variables uniformly distributed on the interval [0, 1].

The solution row of a GA for solving this problem includes indicators $x_i \in \{0, 1\}$ of belonging to a weighting coefficient of w_i for the sought subset. Each element S of the population represents a coded solution row X:

$$S = X, \quad X = (x_1x_2\ldots x_N).$$

The matrices of the probabilities of mutation and crossover of a GA for solving the subset sum problem are $N \times P$ matrices, where N is the number of weighting coefficients $\{w_i\}_{i=1}^{N}$ and P is the population size.

As the mutation operation of a GA for solving the subset sum problem, we can use the above standard version: the operation of mutation inverts bits of the ith element of the jth string of the population with the probability $m_{i,j}^{mut}$.

The most workable scheme of crossover to solve the subset sum problem is a scheme of uniform crossover, namely, for the strings with numbers k and l independently for each number i ($1 \leq i \leq N$),

Table 1. Characteristics of tests (scheduling problem)

Number of test	K	D	M	C	Execution time for optimal schedule in cycles
1	13	[1, 10]	2	3	26
2	14	[1, 10]	3	7	13
3	19	[1, 10]	3	8	30
4	30	[1, 10]	5	6	30
5	44	[1, 10]	6	11	37
6	86	[1, 10]	11	34	40
7	573	[1, 10]	57	51	52
8	107	[1, 10]	20	0	30
9	86	[1, 10]	11	8	40
10	86	[1, 10]	11	17	40
11	86	[1, 10]	11	25	40
12	86	[1, 10]	11	34	40
13	86	[1, 10]	11	43	40
14	85	[1, 3]	11	40	12
15	86	[5, 10]	26	42	25
16	97	[3, 14]	17	15	56
17	94	[1, 15]	11	29	63
18	100	[1, 20]	11	30	86
19	100	[1, 15]	11	30	109
20	100	[1, 10]	2	41	225
21	93	[1, 10]	6	12	75
22	86	[1, 10]	11	34	40
23	81	[1, 10]	21	23	21
24	101	[1, 10]	29	20	20
25	78	[1, 10]	39	25	11

a random number r ($0 \le r < 1$) with the uniform distribution law is chosen; then elements of a new solution are found:

$$\hat{x}_i = \begin{cases} y_i, & r < m_{l,k}^{cr} \text{ and } r < m_{i,l}^{cr}, \\ x_i \text{ otherwise}, \end{cases} \quad \hat{y}_i = \begin{cases} x_i, & r < m_{i,k}^{cr} \text{ and } r < m_{i,l}^{cr}, \\ y_i \text{ otherwise}. \end{cases}$$

The fitness function F^α of the string having the number α has the form

$$F^\alpha = \left(\sum_{i=1}^{N} w_i x_i^\alpha - G \right)^2 / N.$$

As the selection operation of a GA for solving the subset sum problem, it is proposed to employ the above mixed strategy of selecting: for calculating the integer number of descendants and distributing the remainder, one uses the schemes of proportional selection and a roulette, respectively. The probability of the choice of the string having the number α is determined by the value of the fitness function F^α of this string.

For the stop condition, we use a limitation on the number of iterations of the algorithm without improving the solution; i.e., the algorithm stops operating if in I_0 steps it cannot improve the value of the fitness function in the population.

3. EXPERIMENTAL COMPARISON OF ALGORITHMS

3.1. Description of Initial Data

Experimental studies were conducted for randomly generated test data. The initial data of the experimental studies of GAs for solving the problem of building a schedule with the minimum execution time

Table 2. Characteristics of tests (constructing a subset)

Number of test	N	G
1	100	17.494
2	200	35.737
3	300	53.043
4	400	68.196
5	500	89.594
6	600	104.920
7	700	118.219
8	800	139.136
9	900	154.363
10	1000	178.294
11	1500	262.186
12	2000	346.875
13	2500	441.852
14	500	0.000
15	500	24.465
16	500	49.199
17	500	72.937
18	500	98.891
19	500	145.436
20	500	176.688
21	500	204.148
22	500	230.896
23	500	109.501
24	500	139.650
25	500	258.899

for the fixed number of processors are randomly generated tests, which represent graphs with arbitrary connections. The following characteristics of the problem and the graphs are randomly changed:

the number of processes (K);

the dispersion of the computational complexity of a process in cycles (D);

the number of processors (M);

the number of connections among processes (C).

Tests used to conduct the studies are presented in Table 1.

For the subset sum problem, a set of weights is randomly generated. The tests differ in the following characteristics:

the number of elements of the weights' set (N);

the value of the desired sum (G).

Here, elements of the set (the weighting coefficients w_i) are defined as values of independent random variables uniformly distributed on the interval [0, 1].

The tests used to conduct the studies are presented in Table 2.

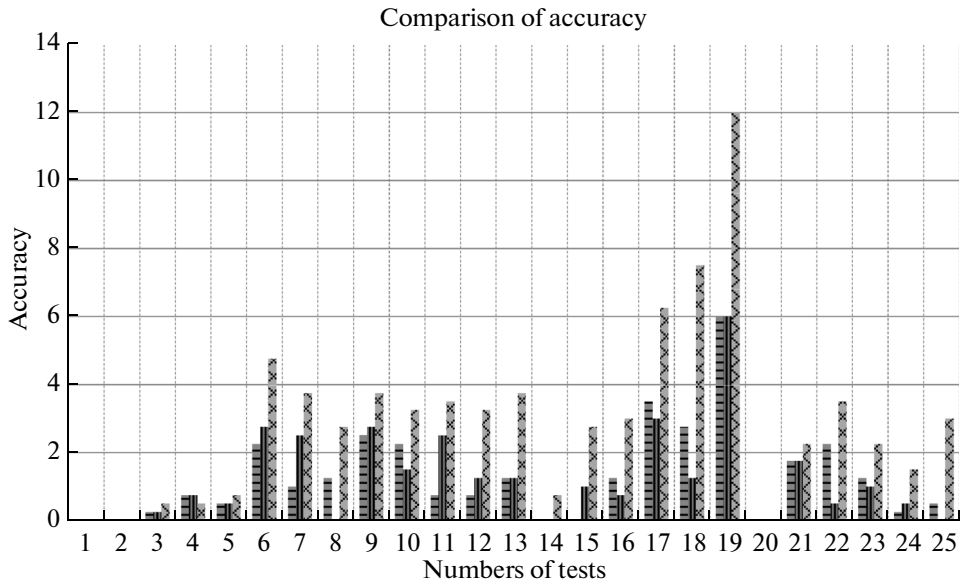

Fig. 1. Accuracy of resulting schedule.

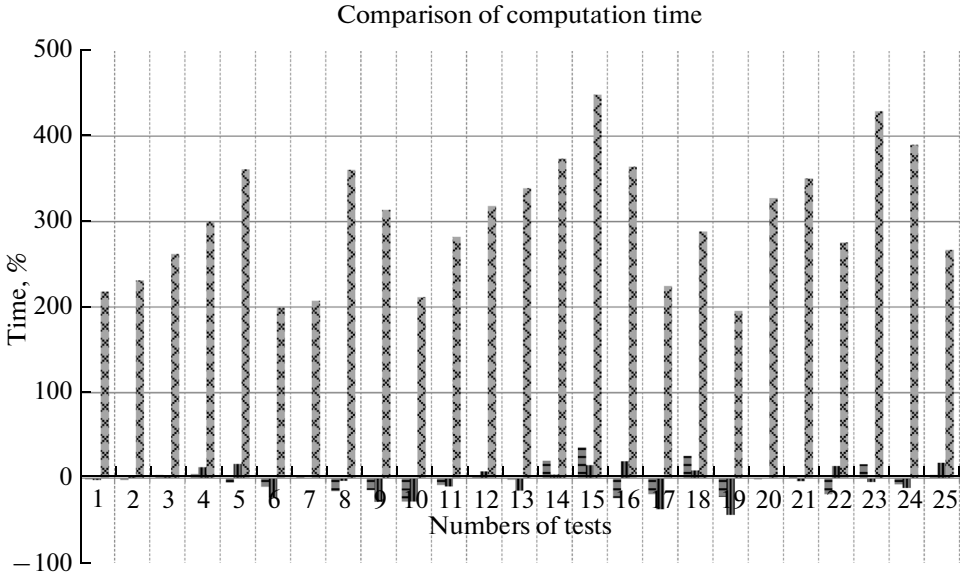

Fig. 2. Time of resulting schedule.

3.2. Scheme for Conducting Experiments

The experimental studies of the developed algorithms consist in conducting a series of 100 experiments for each considered problem. In this case, we use the tests obtained by the generator of the test data for the corresponding problem. The algorithm stops operating if in 250 iterations it cannot improve the value of the fitness function in the population. In each experiment, the following values are measured:

(1) R representing the result of the GA's operation:

for the problem of building a schedule with the minimum execution time for a fixed number of processors, it is the execution time of the obtained schedule in cycles;

for the subset sum problem, it is the absolute difference between the assigned value and the sum of the obtained subset;

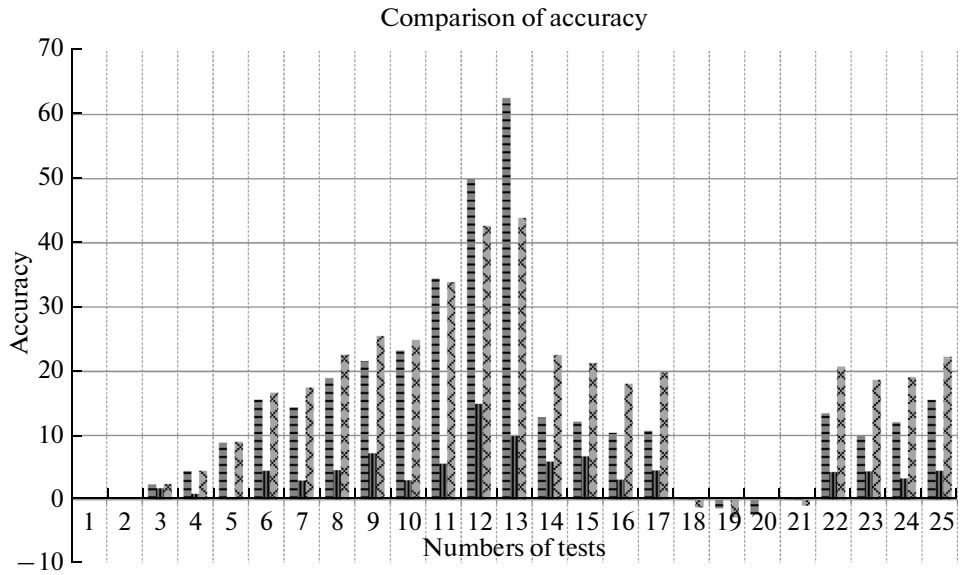

Fig. 3. Accuracy of resulting subset.

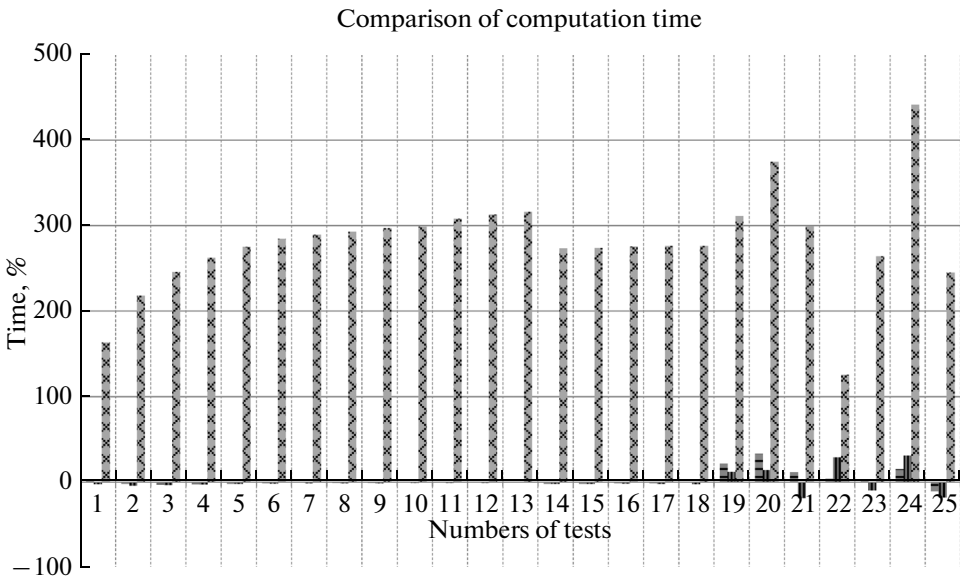

Fig. 4. Time of finding resulting subset.

(2) T representing the algorithm's operating time in seconds.

All experiments were conducted using the same computer system under the same conditions in the absence of any extraneous background problems. Hence, the measurements of the operating time of the algorithms are representative. The population size in each experiment comprised 100.

The brief description of the used computer system:

—The size of the main memory: 32 GB;

—The type of processor: Intel Itanium 2, 1.6 GHz, 8 core;

—The version of the operating system: Red Hat Enterprise Linux Server release 5.9.

All the obtained data were processed using the method of the statistical hypotheses test [11]. As a result we obtain that with a probability of at least 0.85, a GA with a memory vector solves each considered problem more efficiently than the corresponding classical algorithm. In the course of the experimental studies, the following hypotheses were tested:

1. For the problem of building a schedule with the minimum execution time for the fixed number of processors:

H^1 (the accuracy of a self-learning GA is not lower than the accuracy of the classical GA);

H^2 (the average operating time of a self-learning GA is not longer than the operating time of the classical GA).

2. For the subset sum problem:

H^3 (the accuracy of a self-learning GA is not lower than the accuracy of the classical GA);

H^4 (the average operating time of a self-learning GA is not longer than the operating time of the classical GA).

3.3. Results of the Experiments

The results of the comparative experimental studies of the self-learning GA and the Holland classical GA for the problem of building a schedule with the minimum execution time for the fixed number of processors are shown in Figs. 1 and 2. The absolute values of the results of the experimental studies are presented in the Appendix; these values are the foundation of the graphs.

The graphs demonstrate the results of operating the self-learning GA for three above-considered methods of correcting a memory vector (the results are averaged over the number of starts). For the basic results, the outcome of the classical GA is taken. The accuracy specifies the number of cycles by which the execution time of a schedule obtained by the considered algorithm was improved in comparison with the classical GA. The operating time of the algorithm specifies the increase or the decrease of the operating time of the algorithm compared to the classical GA (in percentage terms). In 100% of the cases, the accuracy of the self-learning GA is not lower than the accuracy of the classical GA, and this confirms the hypothesis H^1. The best algorithm both for the accuracy and for the execution time is the self-learning GA with the use of the relative method for correcting the probability matrices of mutation and crossover with forgetting. For this algorithm the hypothesis H^2 is true.

The results of the comparative experimental studies of the self-learning GA and the Holland classical GA for the subset sum problem are shown in Figs. 3 and 4. Absolute values of the results of the experimental studies are presented in the Appendix; these values are the foundation of the graphs.

As before, the graphs demonstrate the results of operating the self-learning GA for the three above-considered methods of correcting a memory vector (the results are averaged over the number of starts). For the basic results, the outcome of the classical GA is taken. The accuracy specifies the degree of the improvement of the solution's quality. This quality is defined by the absolute difference between the assigned value and the sum of the obtained subset. The accuracy of the self-learning GA is higher than the accuracy of the classical GA approximately in 85% of the cases. The difference in the accuracy is particularly noticeable for the experiments with the sets of a large size (the numbers of tests ranges from 10 to 13). The operating time of the self-learning GA with the use of the relative method for correcting the probability matrices of mutation and crossover with forgetting is always shorter than the operating time of the classical GA. Thus, hypotheses H^3 and H^4 are also true.

CONCLUSIONS

In this work, the self-learning GA is proposed. The experimental studies of the proposed self-learning GA and the Holland classical GA were conducted for job scheduling problem and subset sum problem. The experimental studies show the advantage of the self-learning GA in the quality of the obtained solutions and operating time.

APPENDIX

Since the number of experiments is 100, each of the 25 tests was started 4 times. Tables 3 and 4 present the results averaged over the number of starts. In these tables, the following notation is used: GA1 = the classical GA, GA2 = the self-learning GA (the method for correcting elements of memory matrices is absolute), GA3 = the self-learning GA (the method for correcting elements of memory matrices is rela-

Table 3. Problem of multiprocessor scheduling

Number of test	Result (execution time for obtained schedule in cycles)				Operating time of algorithm, s			
	GA1	GA2	GA3	GA4	GA1	GA2	GA3	GA4
1	26.00	26.00	26.00	26.00	0.512	0.519	0.521	0.161
2	13.00	13.00	13.00	13.00	0.618	0.628	0.608	0.187
3	30.50	30.25	30.25	30.00	1.075	1.035	1.040	0.297
4	31.50	30.75	30.75	31.00	3.402	3.240	3.015	0.851
5	39.75	39.25	39.25	39.00	7.927	8.325	6.801	1.720
6	50.50	48.25	47.75	45.75	23.127	25.685	29.868	7.741
7	79.75	78.75	77.25	76.00	1023.81	1009.47	1024.71	333.026
8	40.50	39.25	40.50	37.75	39.915	46.943	41.075	8.674
9	48.75	46.25	46.00	45.00	25.666	29.624	35.423	6.207
10	48.50	46.25	47.00	45.25	23.758	32.493	32.773	7.630
11	50.50	49.75	48.00	47.00	26.074	28.299	28.755	6.825
12	50.00	49.25	48.75	46.75	36.287	35.262	33.655	8.683
13	50.50	49.25	49.25	46.75	30.210	30.636	35.195	6.886
14	14.75	14.75	14.75	14.00	26.634	22.105	25.614	5.629
15	40.25	40.25	39.25	37.50	46.340	34.110	40.224	8.452
16	72.00	70.75	71.25	69.00	35.872	46.539	30.001	7.732
17	78.00	74.50	75.00	71.75	25.803	31.606	40.219	7.951
18	104.00	101.25	102.75	96.50	49.189	39.113	45.125	12.671
19	133.25	127.25	127.25	121.25	33.206	42.280	57.909	11.241
20	225.00	225.00	225.00	225.00	23.636	23.928	23.540	5.535
21	80.25	78.50	78.50	78.00	28.790	28.205	29.799	6.399
22	49.25	47.00	48.75	45.75	33.925	41.588	29.726	9.040
23	31.75	30.50	30.75	29.50	34.233	29.481	35.675	6.474
24	31.50	31.25	31.00	30.00	48.810	52.410	54.915	9.970
25	21.75	21.25	21.75	18.75	31.395	30.243	26.575	8.560

Table 4. Subset sum problem

Number of test	Result (absolute deviation from the value of the desired sum)				Operating time of algorithm, s			
	GA1	GA2	GA3	GA4	GA1	GA2	GA3	GA4
1	0.004	0.001	0.002	0.000	0.192	0.186	0.196	0.073
2	0.036	0.002	0.038	0.001	0.602	0.612	0.625	0.189
3	2.392	0.044	0.620	0.004	1.262	1.295	1.303	0.365
4	4.497	0.046	3.621	0.006	2.170	2.217	2.223	0.599
5	9.016	0.186	8.636	0.035	3.323	3.375	3.380	0.886
6	16.694	1.132	12.233	0.026	4.726	4.778	4.792	1.230
7	17.501	3.128	14.541	0.058	6.374	6.412	6.446	1.638
8	24.220	5.304	19.625	1.653	8.254	8.309	8.354	2.101
9	29.225	7.670	22.046	3.755	10.391	10.455	10.496	2.619
10	34.365	11.151	31.372	9.506	12.762	12.783	12.877	3.203
11	62.695	28.287	57.108	28.843	28.303	28.287	28.506	6.937
12	95.279	45.310	80.399	52.643	49.951	49.778	50.256	12.099
13	123.630	61.054	113.742	79.745	77.749	77.514	78.152	18.703
14	96.598	83.846	90.708	74.051	3.327	3.384	3.385	0.891
15	71.443	59.336	64.700	50.169	3.329	3.381	3.385	0.891
16	44.757	34.374	41.646	26.701	3.332	3.375	3.378	0.888
17	21.591	10.892	17.065	1.794	3.329	3.373	3.387	0.885
18	0.266	0.005	0.036	1.439	3.328	3.313	3.394	0.885
19	0.003	1.360	0.003	2.784	3.806	3.127	3.397	0.926
20	0.002	2.286	0.001	0.001	4.217	3.163	3.694	0.889
21	0.250	0.002	0.122	1.148	3.860	3.449	4.722	0.967
22	22.437	8.997	18.145	1.728	5.647	5.433	4.382	2.505
23	45.916	35.893	41.553	27.270	5.157	5.084	5.700	1.416
24	71.839	59.810	68.549	52.726	7.847	6.814	5.980	1.450
25	100.679	85.202	96.180	78.445	5.001	5.599	6.078	1.450

tive), and GA4 = the self-learning GA (the method for correcting elements of memory matrices is relative to forgetting).

REFERENCES

1. V. A. Kostenko, "Scheduling algorithms for real-time computing systems admitting simulation models," Program. Comput. Software **39**, pp. 255–267 (2013).
2. V. A. Kostenko, R. L. Smelyanskii, and A. G. Trekin, "Synthesizing structures of real-time computer systems using genetic algorithms," Program. Comput. Software **26**, pp. 281–288 (2000).
3. V. A. Kostenko and V. V. Shcherbinin, "Training methods and algorithms for recognition of nonlinearly distorted phase trajectories of dynamic systems," Opt. Memory Neural Networks (Inform. Opt.) **22** (1), pp. 8–20 (2013).
4. L. A. Rastrigin, *Statistical Methods of Search* (Nauka, Moscow, 1968) [in Russian].
5. J. N. Holland, *Adaptation in Natural and Artificial Systems* (Univ. of Michigan Press, Ann Arbor, Michigan, 1975).
6. V. A. Kostenko, "The problem of schedule construction in the joint design of hardware and software," Program. Comput. Software **28**, pp. 162–173 (2002).
7. R. Diestel, *Graph Theory, Electronic Edition* (Springer-Verlag, New York, 2005), p. 422.
8. *Computer and Job-shop Scheduling Theory*, Ed. by E. G. Coffman (Wiley, New York, 1976; Nauka, Moscow, 1984).
9. Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs* 3rd ed. (Springer, 1999).
10. M. L. Rattray, "The dynamics of a genetic algorithm under stabilizing selection," Complex Systems **9** (3), pp. 213–234 (1995).
11. V. N. Kalinina and V. F. Pankin, *Mathematical Statistics* (Vyssh. Shkola, Moscow, 1998) [in Russian].

Translated by L. Kartvelishvili