Using genetic algorithm for generating optimal data sets to automatic testing the program code

K E Serdyukov[1], T V Avdeenko[1]

[1]Novosibirsk State Technical University, Marksa avenue, 20, Novosibrsk, Russia, 630073

e-mail: tavdeenko@mail.ru

Abstract. In present paper we propose an approach to automatic generation of test data set based on application of the genetic algorithm. We consider original procedure for computation of the weights of code operations used to formulate the fitness function being the sum of these weights. Terminal objective and result of fitness function selection is maximization of code coverage by generated test data set. The idea of the genetic algorithm application approach is that first we choose the most complex branches of the program code for accounting in the fitness function. After taking the branch into account its weight is reset to zero in order to ensure maximum code coverage. By adjusting the algorithm, it is possible to ensure that the automatic test data generating algorithm finds the most distant from each other parts of the program code and, thus, the higher level of code coverage is attained. We give a detailed example illustrating the work and advantages of considered approach and suppose further improvements of the method.

1. Introduction

One of the most important stages in developing software products is testing. The terminal goals of the testing phase are compliance of the developed program with the specified requirements, ensuring right logic while data processing and, as a result, obtaining the correct final results.

Scaling of the software development stimulated the processes of creating huge software systems by diverse development teams, each of which has its own programming style and different competencies. Despite the fact that in parallel with this process there appeared programs allowing for a high level of collaborative development, control over changes and the ability to check the quality of the code, the final product does not always meet the requirements specified at the planning stage.

For this reason, the need for quality and comprehensive testing increases significantly. It is necessary not only to find errors in the code, but also logical inconsistencies. In order to test both the program as a whole and its parts as thoroughly as possible, not only a team of testers is needed, but also preparatory activities – the formation of a set of input data that would test certain parts of the program [2].

Based on the above, we can conclude that automation of testing, or at least automatic test generation, can significantly reduce not only the time but also the cost of development. There are other

advantages of that are not so obvious – a high probability of finding small errors, transparency of test development, testing simultaneously with the development of the program, etc. [1].

Testing is not a standardized process, it depends on many factors, most of which vary from one program to another. In addition, improvement of methods for automatic verification and validation of program code occur quite slowly. Development of most types of designs and templates for testing is often done manually, without use of any intelligent software. Therefore, the testing process becomes incredibly complex and time-consuming, as well as costly, if the ultimate goal is indeed the creation of high-quality software product. In such cases the testing phase can take up to 50% of the whole development time. In this regard, it seems appropriate to use methods developed in the field of artificial intelligence.

One of the most important problems to be solved at the beginning is to identify one of the most complex branches of the code. Based on the solution of this initial problem, we can further build an algorithm for finding test data set that provides coverage of the most complex branches (as many as possible) of the code. In this paper we are trying to derive a solution to this local problem of finding one the branch with most operations of the program.

Various methods have been proposed for solving the problem of automatic test generation. The paper [3] compares various methods for generating test data, including genetic algorithms, random search method, and other heuristic approaches. In [4] it is proposed to use programming based on the constraint logic programming and symbolic execution to solve this problem. In [5] constraint handling rules are used to assist in the manual verification of problem points in the code.

Some researchers use heuristic methods with the help of visualization instruments to automate the process of testing, such as data flow diagram. Studies of automation methods using this diagram have been proposed in [6-9]. In the article [6] it is proposed to additionally use genetic algorithms to determine new input test data sets based on previously used ones.

The articles [10-13] consider integrated approaches for generating test data. In [12] an approach is used that combines strategies of random search and dynamic symbolic computations. The article [13] proposes a theoretical description of a search testing strategy using a genetic algorithm. Approaches to search for local and global extremes on real programs are considered. A hybrid approach for generating test data is proposed - a memetic algorithm.

Approach in [14] uses a hybrid intelligent search algorithm to generate test data. In the proposed algorithm, the method of branch and bound and hill climbing) are used with the use of intellectual search.

Also, there is investigate of approaches for generating test data based on the machine learning [15]. The proposed approach uses the neural network structure with user-configured clustering of input data for sequential learning. There are approaches based on the meta-heuristic algorithm of the cuckoo [16].

For the convenience of generating test data, the UML diagrams are also used. [17, 18]. The articles propose to use genetic algorithms for generating triggers for UML diagrams, which will allow finding the critical path in the program. In article [19] an improved genetic algorithm-based method is proposed for selecting test data for multiple parallel paths in UML diagrams.

In addition to UML diagrams, the program can be displayed as a classification tree method developed by Grochtmann and Grimm [20]. In paper [21] discusses the problem of tree building and proposes an integrated classification tree algorithm, and in [22] the developed ADDICT prototype (abbr. AutomateD test Data generation using the Integrated Classification-Tree methodology) is investigated for an integrated approach

There are many different researches on theme of generation of test data. Most often, to solve this problem, heuristic approaches are used, since they are allowed to select data with not a complete enumeration of possible options. The approach proposed in this article is based on a genetic algorithm with a modification of the calculation of the function of adaptation, which allows to generate data based on a program code without reference to any testing and development systems. This allows to generate data directly only by specifying restrictions on the input variables.

2. Genetic algorithm

Genetic algorithm is a heuristic method, more precise, one of the types of the evolutionary algorithms, that uses the idea and terminology of evolution of the nature. Its goal is not to find the optimal and best solution, but to find one that is close enough to it. Therefore, genetic algorithm is not recommended to apply if there are already fast and well-developed optimization methods. But at the same time, the genetic algorithm perfectly shows itself in solving non-standardized tasks, problems with incomplete data or if it impossible to use other optimization methods because of the complexity of implementation or the duration of execution [23, 24].

A genetic algorithm is considered completed if a certain number of iterations are passed (it is desirable to limit the number of iterations, since the genetic algorithm works on the method of trial and error, which is quite a long process), or if the satisfactory value of the fitness function was obtained. Generally, the genetic algorithm solves the problem of maximizing or minimizing and the adequacy of each decision (chromosome) is assessed using the fitness function.

Genetic algorithm works according by the following principle:

- *Initializing.* Establishing fitness function. Forming the initial population. Classically, the initial population creating by random filling of genes in the chromosomes. However, to increase the convergence rate, the initial population can be filling in specific way, there the values can be analyzed in advance for exclusion of definitely unsuitable genes.
- *Evaluation of population.* Each of the chromosomes is evaluated by the fitness function. Based on specified requirements, chromosomes acquire a certain value in accordance with the solution of the problem [25].
- *Selection.* After each chromosome obtain its own value, the selection of the best chromosomes take place. Selection can be done by different methods, for example, take the first n chromosomes sorted by value of the fitness function, or only chromosomes with maximum value of the fitness function, etc.
- *Crossover*. The first significant difference from conventional methods and one of the most important stages of the algorithm. After selection and retrieving the suitable chromosomes to solve the problem, they crossover with each other. Randomly selected chromosomes generate new chromosomes. Crossover occurs based on the selection of a specific position in the two chromosomes and mutual replacement of parts. After filling the required number of chromosomes to create a new population, the algorithm proceeds to the next step [26].
- *Mutation.* This is also a step characteristic for GA only. In random order, a random gene can change values to a random one. The main purpose of the mutation is the same as in biology – the introduction of genetic diversity in the population. The main goal of mutation is to obtain solutions that could not be produced with existing genes. This will allow, firstly, to avoid falling into local extremes, since the mutation may allow the algorithm to go a completely different path, and secondly, to “dilute” the population in order to avoid a situation where there are only identical chromosomes in the entire population that will not move towards a global solution.

After all stages of the genetic algorithm have been completed, it is estimated whether the population has reached the desired accuracy of the solutions, or whether a certain number of populations have been reached. If these conditions have been met, the algorithm stops working. Otherwise, the cycle is repeated with the new population until the conditions are reached.

3. The test data generation with genetic algorithm

The use of genetic algorithms in the testing process makes it possible to ensure that we will determine the most complex parts of the program code in which the risks due to making mistakes are the greatest. Evaluation is executed through the use of the fitness function, the parameters of which have the different weights of each individual operation [27].

To date, many types of diagrams have been developed that allow us to represent the structure of a program code not as a set of actions, but as diagrams with a specific structure. The most widely used are diagrams (graphs) of control flows which allow representing the whole variety of ways to run a

program. The main purpose of such diagrams falls on the task of creating the program code which includes determining the program complexity, verifying the program logic, and directly writing the code. However, from the problem of generating test data point of view, this type of diagrams built on the already written program code, permits to assess the quality of the developed code and, within the scope of the task, to assess the importance, or complexity, of certain program paths.

Based on the possibility of presenting the program code in the structural form, an approach was developed through which it would be possible to evaluate the program code and to determine such a set of test data that would allow one to "walk" through the largest number of operations and the greatest number of paths. The first step in the proposed method is to consider the structural elements of the code. For ease of presentation, we can use flow diagrams to visualize the structure of the code and to understand in what way the program is executed.

Each operation of the code is assigned its own separate graph node, and as a link is the direction in which the code is executed. For example, a condition is denoted with one graph node, but two branches of the code will come out of it. Each transition between the graph nodes is assigned a certain weight depending on which part of the code the operation is in, whether any complex structural elements precede it, etc. [28].

The problem of generating the input test data consists of three subproblems:

1) Search for the input data for traversing one of the most complex code branches;
2) Elimination or reduction of the weights of operations lying on this branch of the code at the rate of subsequent branches;
3) Search for a set of test data for traversing multiple branches at once.

The limitation on the size of the input data set is established after the development stage and allows one to concentrate on certain branches in which the largest number of operations is performed.

The whole algorithm is executed cyclically. First the procedure of searching the input data for one branch is started, then the operations in this branch are excluded from further computations and the data search for one branch is started again.

Search for a single path in the program code works as follows:

- The first operation is assigned a weight, for example, 100.
- Each subsequent operation is also assigned a weight – if there are no conditions or cycles, the weight is equal to the weight of the previous operation.
- Weight of the condition is assigned in accordance with the following rule. If the condition contains only one branch (only if ...), then the weight of each operation is reduced on 0,8. If the condition is divided into several branches (if ... else ...), then the weight is divided into equivalent parts - for two branches 50 / 50, for three 33/ 33 / 33, etc.
- Weights of operations in the cycle remain the same, but can also be multiplied by a certain weight, if it is necessary to increase the significance of the cycles during testing.
- All nested restrictions are taken into account, for example, for two nested conditions, the weight of operations will be equal to 80 * 80 = 64 percent

In Figure 1 we present an example on the basis of which the algorithm was tested.

As a result of the above procedure we obtain the weights that can be used to develop test variants using genetic algorithm, that is, to estimate how much calculated weight falls on here or another branch for certain values of input parameters.

For convenience, we introduce the following notation:

X – data sets; m – population size, i.e. the number of different variants of input data values; r(i) – the weight of a single operation i; F (X) – the value of the fitness function for each data set depending on the calculated weights.

The problem is to select the maximum of the objective function, i.e.

$$F(X) = \sum_{i=0}^{m} r(\mathrm{i}) \rightarrow max \qquad (1)$$

```
0.  public static void Main(int m, int i, int n) {
1.  intsum = 0, avg = 0;
2.  if (m <i)
3.  {
4.  use_m = m;
5.  }
6.  int[] a = new int[m];
7.  Console.WriteLine("Enter the Array Elements ");
8.  for (j= 0; j< m; j++)
9.  {
10. a[j] = i;
11. if (a[j] > n)
12. {
13. a[j] = n
14. }
15. }
16. for (k= 0; k < m; k++)
17. {
18. sum += a[k];
19. }
20. avg = sum / m;
21. Console.WriteLine("Average is {0}", avg);
22. Console.ReadLine();
23. }
```

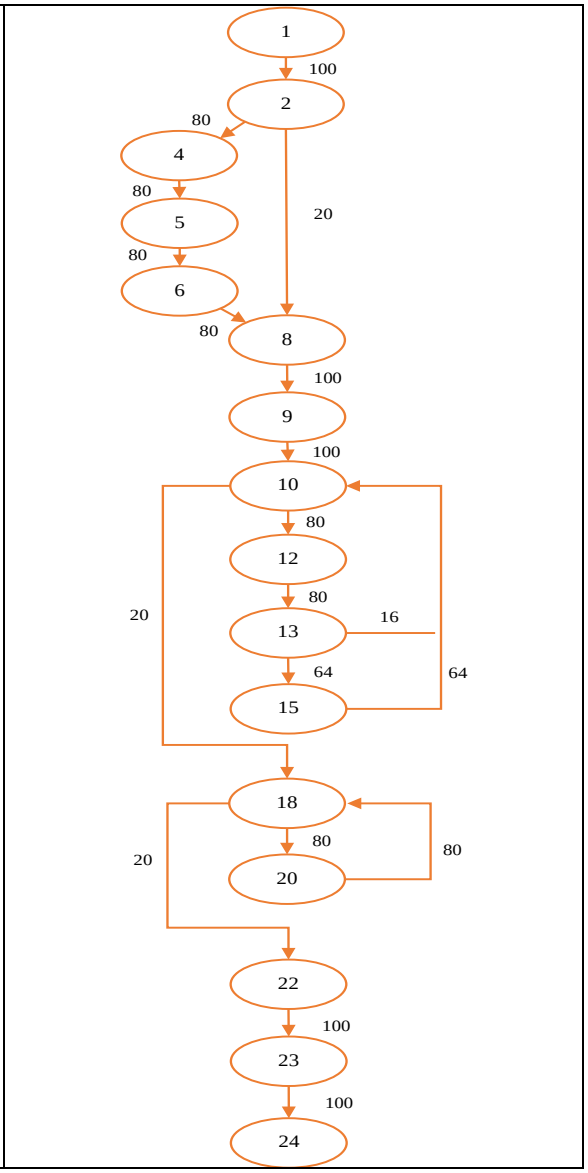

Figure 1. Program code and sequence of graph nodes for the analyzed program code.

4. Results

For a more detailed consideration of the algorithm we use simple test options and carry out each of the steps manually with a detailed description of the pro. At the initialization stage we will generate the following data sets – (10,5,12); (3,4,10); (25,30,11); (5,3,17).

Table 1 presents these sets, the calculated target value of the fitness function and the rank corresponding to the best set.

Table 1. Initial dataset options.

№	Sets X	F(X)	Rank
1	(10,5,12)	896	3
2	**(3,4,10)**	**1196**	**2**
3	**(25,30,11)**	**1308**	**1**
4	(5,3,17)	896	3

From the table we can see that the better options for the selection are 2nd and 3rd options. In order to obtain additional two new variants, their values will be mixed with a certain probability of mutation.

In the crossover stage the division of data sets occur in the first and second positions. For example, when crossing (R1, R2, R3) and (G1, G2, G3), the variables are obtained - (R1, R2, G3), (R1, G2, G3), (G1, R2, R3) and (G1, G2, R3). Parental values remain in the set-in order to keep the crossing clean, i.e. compared to the zero generation there will be added 2 new sets. Thus, in the subsequent generations, six data sets will be used. It is worth mentioning that, depending on the settings of the genetic algorithm, the parental chromosomes can be excluded from consideration.

Mutation will occur with a probability of 0,1 for the chance of changing the value from 1 to the specified value in both directions. Under these conditions, the maximum possible value added during a mutation is 5. As a result of crossing, the data sets will be obtained, shown in Table 2.

Table 2. New datasets obtained by crossing.

№	X	Y	New set	**Mutation**
1	(3,4,10)	(25,30,11)	(3,4,11)	**(3,4,13)**
2	(3,4,10)	(25,30,11)	(3,30,11)	**(3,30,11)**
3	(3,4,10)	(25,30,11)	(25,4,10)	**(25,4,10)**
4	(3,4,10)	(25,30,11)	(25,30,10)	**(25,30,10)**

As a result, two more variants will be added to the additional two parent sets - (3,4,13) and (25,30,10). Table 3 shows all the new variations of the test data set.

Table 3. The first generation of test data.

№	Sets X	F(X)	Rank	Generation
1	(3,4,10)	1196	2	0
2	(25,30,11)	1308	1	0
3	(3,4,13)	1196	2	1
4	(3,30,11)	1308	1	1
5	(25,4,10)	896	3	1
6	(25,30,10)	1308	1	1

If the two variants have the same rank, priority will be given to the option from the newer generation. In the last generation we obtained three data sets, which test the majority of the whole set of program paths - (25,30,11), (3,30,11) and (25,30,10). The first set was obtained from the first generation, so it will be excluded and there will remain only two options - (3,30,11) and (25,30,10).

Because of the small initial sample and a small code, the data sets quickly came to finding overlapping values - 30 in the second position and 10 or 11 in the third. Therefore, to continue to carry out iterations ceases to make sense - already in the next generation, the data will consist mainly of repeating sets.

For the current program code you can use test data sets obtained in the latest generation. Priority depends on rank received.

Thus, using genetic algorithms, one can find such initial test initial values, which would fully check all the paths of the program.

5. Improving the algorithm

The algorithm allows not only to evaluate which program paths will be used for certain data, but also how the data changes and whether duplicate values are preserved for the best chromosomes between populations.

This not only makes it possible to determine the initial test suite, but also, on the basis of data analysis, to draw conclusions about the presence of logic in the program that corresponds to the planned one.

Four additional tests were performed to present the results.In the algorithm, the first population is formed randomly. Certain settings were made for testing – each population contains 100 chromosomes; the total number of populations also equals 100. This will make it possible to form a sufficient number of different variants.

Table 4 presents 4 runs of the method with the first random population, two middle populations and the final one, from which the first chromosome is taken and counted as the final generated data set. For convenience, only the 5 "best" chromosomes in each population will be reflected.

Table 4. Comparison of results.

Population	Test 1	Test 2	Test 3	Test 4
0	1: 78, 23, 35	1: 97, 3, 6	1: 92, 97, 28	1: 15, 67, 26
	2: 62, 36, 95	2: 82, 77, 64	2: 38, 66, 52	2: 32, 27, 83
	3: 52, 35, 27	3: 24, 47, 57	3: 63, 76, 64	3: 37, 52, 64
	4: 17, 77, 73	4: 90, 13, 82	4: 7, 24, 56	4: 70, 49, 64
	5: 75, 9, 96	5: 81, 69, 24	5: 57, 48, 8	5: 67, 29, 94
20	1: 95, 64, 54	1: 97, 80, 4	1: 99, 13, 10	1: 99, 71, 45
	2: 95, 64, 29	2: 97, 80, 53	2: 99, 13, 11	2: 99, 71, 15
	3: 95, 64, 54	3: 97, 80, 28	3: 99, 13, 11	3: 99, 71, 3
50	1: 95, 64, 54	1: 97, 80, 29	1: 99, 13, 10	1: 99, 71, 60
	2: 95, 64, 29	2: 97, 80, 4	2: 99, 13, 11	2: 99, 71, 3
	3: 95, 64, 54	3: 97, 80, 53	3: 99, 13, 11	3: 99, 71, 3
Final (100)	**1: 95, 64, 54** **2: 95, 64, 29**	**1: 97, 80, 4** **2: 97, 80, 29**	**1: 99, 13, 10** **2: 99, 13, 11**	**1: 99, 71, 60** **2: 99, 71, 45**

At least two different final sets of test data were formed in each of the variants, in which the operations in the considered program code will have the greatest weight. In addition, there is certain patterns in the results - the first value is always the maximum (random values are limited to a maximum of 100 to increase convergence), the second value is less than the first, but more than the third.

For additional analysis, we will check how the speed of the algorithm changes depending on the genetic algorithm settings. In the graphs below it can be seeing how the duration of the program varies depending on the size of the population, i.e. the number of chromosomes, and the total number of populations. During the study of changes in the size of populations, the number of populations was equal to 100. And vice versa, during the study of the dependence on the number of populations, the number of chromosomes was equal to 100.

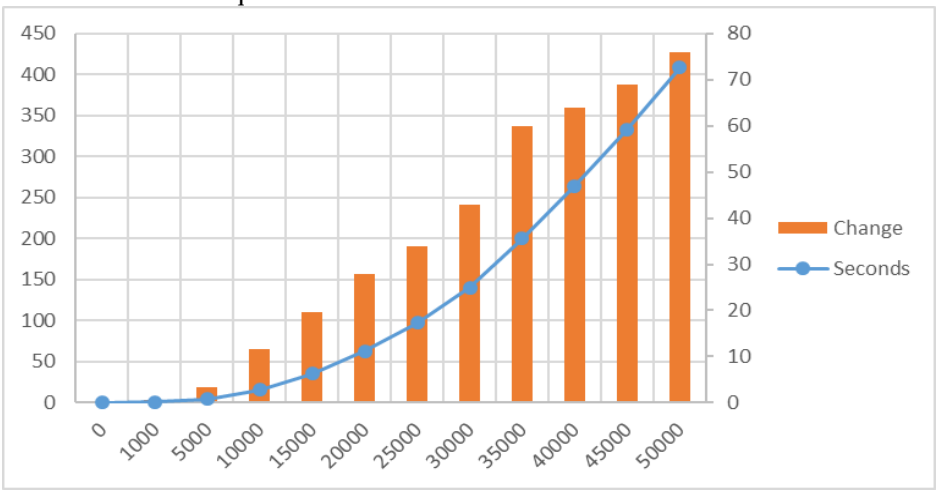

Figure 2. Dependence of execution duration on population size.

Figure 2 shows the dependence on the number of chromosomes in the population. Based on it, it can be concluded that with an increase number of chromosomes in a population, the duration of the algorithm increases significantly, and more, exponentially.

Figure 3 shows the dependence on the number of populations.

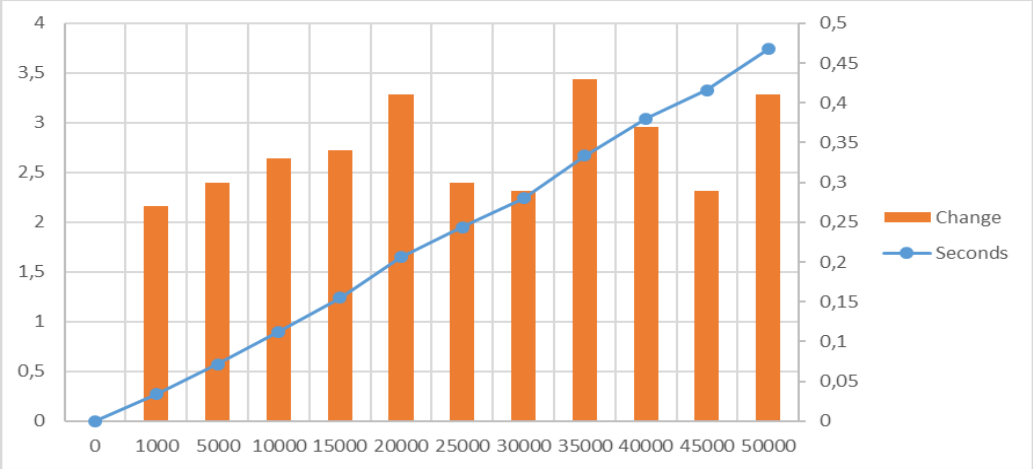

Figure3. Dependence of execution duration on the number of populations.

It becomes obvious that with a change in the number of populations, the duration of the algorithm increases, but in a linear progression. At the same time, there are noticeable fluctuations in the gain in both directions, which remain ap-proximately at the same level.

Despite the fact that in both cases total number of chromosomes remained the same, changes in the duration of work are considerably different. An increase in population size significantly increases the duration of work — the number of chromosomes in the population from 1,000 to 50,000 increased 50 times, but the duration increased 750. When the number of populations changed 50 times, the time increased only 14 times.

This is due to the fact that the most complicated operations from the point of view of loading occur in calculating the fitness function and searching for optimal chromosomes for crossing. Because the search for the best chromosomes depends on the number of chromosomes in the population, so the current search algorithm, i.e. sorting chromosomes in population from best to worst, significantly loads the power of computer systems and increases the speed of work exponentially to the size of the population.

The number of chromosomes in one population allows for a variety of options, i.e. more likely to find more suitable options. An increase in the number of populations leads to a more accurate result, but only with a large number of chromosomes. If the chromosomes are small enough, the algorithm will quickly come to one repeated value.

Based on the foregoing, the number of chromosomes in one population has the greatest influence on the result. At the same time, an increase in the number of chromosomes significantly increases the execution time. But in order to ensure the better end result, the total number of populations should also be increased with an increase in the number of chromosomes.

6. Conclusion

Evolutionary methods work in such a way as to find the best solutions in the problems that are impossible or too costly to solve using standard optimization methods. They do not always work fast or qualitatively, but they show superiority in tasks with non-standard approaches.

The method based on the genetic algorithm will automate the method of selection of input data, while significantly increasing the speed of data retrieval. The algorithm is fully automatic (with the exception of some restrictive settings), so it does not require additional testers or developers work. The resulting data set can be directly used for the testing process and, if necessary, be re-assembled at no additional cost.

In the future, it is reasonable to conduct investigations of the influence of various metrics on the final result and the amount of code coverage, in order to provide such data sets that would allow testing the code as efficiently as possible and with the maximum number of operations.

7. References

[1] Zanetti M C, Tessone CJ, Scholtes I and Schweitzer F 2014 Automated Software Remodularization Based on Move Refactoring. A Complex Systems Approach *3th international conference on Modularity* 73-83
[2] Crispin L, Gregory J 2010 *Agile Testing: A Practical Guide for Testers and Agile Teams* (Pearson Education) p 576
[3] Maragathavalli P, Anusha M, Geethamalini P and Priyadharsini S 2011 Automatic Test-Data Generation For Modified Condition/ Decision Coverage Using Genetic Algorithm *International Journal of Engineering Science and Technology* **3(2)** 1311-1318
[4] Meude C 2001 AT Gen: Automatic Test Data Generation using Constraint Logic Programming and Symbolic Execution *Software Testing Verification and Reliability*
[5] Gerlich R 2014 Automatic Test Data Generation and Model Checking with CHR *11th Workshop on Constraint Handling Rules*
[6] Moheb R 2005 Automatic Test Data Generation for Data Flow Testing Using a Genetic Algorithm *Journal of Universal Computer Science* **11(6)** 898-915
[7] Weyuker E J 1984 The complexity of data flow criteria for test data selection *Inf. Process. Lett.* **19(2)** 103-109
[8] Khamis A, Bahgat R and Abdelaziz R 2011 Automatic test data generation using data flow information *Dogus University Journal* **2** 140-153
[9] Singla S, Kumar D, Rai H M and Singla P 2011 A hybrid pso approach to automate test data generation for data flow coverage with dominance concepts *Journal of Advanced Science and Technology* **37** 15-26
[10] Luger F G 2009 *Artificial Intelligence Structures and Strategies for Complex Problem Solving* (University of New Mexico) p 679
[11] Berndt D J, Fisher J, Johnson L, Pinglikar J and Watkins A 2003 Breeding Software Test Cases with Genetic Algorithms *Proceedings of the Thirty-Sixth Hawaii International Conference on System Sciences* **36**
[12] Liu Z, Chen Z, Fang C and Shi Q 2014 Hybrid Test Data Generation / State Key Laboratory for Novel Software Technology *Companion Proceedings of the 36th International Conference on Software Engineering* 630-631
[13] Harman M, McMinn P 2010 A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search *IEEE Transactions on Software Engineering* **36(2)** 226-247
[14] Xing Y, Gong Y Z, Wang Y W and Zhang X Z 2015 A Hybrid Intelligent Search Algorithm for Automatic Test Data Generation *Mathematical Problems in Engineering* **2015** 15
[15] Paduraru C, Melemciuc M C 2018 An Automatic Test Data Generation Tool using Machine Learning *13th International Conference on Software Technologies, ICSOFT* 472-481
[16] Panda M, Sarangi P and Dash S 2015 Automatic Test Data Generation using Metaheuristic Cuckoo Search Algorithm *International Journal of Knowledge Discovery in Bioinformatics* **5(2)** 16-29
[17] Doungsa-ard C, Dahal K, Hossain A G and Suwannasart T 2007 An automatic test data generation from UML state diagram using genetic algorithm *IEEE Computer Society Press* 47-52
[18] Sabharwal S, Sibal R and Sharma C 2011 Applying Genetic Algorithm for Prioritization of Test Case Scenarios Derived from UML Diagrams *IJCSI International Journal of Computer Science* **8(3/2)**
[19] Doungsa-ard C, Dahal K, Hossain A and Suwannasart T 2008 GA-based Automatic Test Data Generation for UML State Diagrams with Parallel Paths *Advanced design and manufacture to gain a competitive edge: New manufacturing techniques and their role in improving enterprise performance* 147-156
[20] GrochtmannM, GrimmK 1993 Classification trees for partition testing. Software Testing *Verification and Reliability* **13(2)** 63-82

[21] Chen T Y, Poon P L and Tse T H 2000 An integrated classification-tree methodology for test case generation *International Journal of Software Engineering and Knowledge Engineering* **10(6)** 647-679
[22] Cain A, Chen T Y, Grant D, Poon P, Tang S and Tse T H 2004 An Automatic Test Data Generation System Based on the Integrated Classification-Tree Methodology *Software Engineering Research and Applications, Lecture Notes in Computer Science* **3026** 15
[23] Serdyukov K, Avdeenko T 2017 Investigation of the genetic algorithm possibilities for retrieving relevant cases from big data in the decision support systems *CEUR Workshop Proceedings* **1903** 36-41
[24] Yang HL, Wang C S 2008 Two stages of case-based reasoning - Integrating genetic algorithm with data mining mechanism *Expert Systems with Applications* **l 35** 262–272
[25] Mühlenbein H 1992 *How genetic algorithms really work: Mutation and hill climbing* (Parallel Problem Solving from Nature 2, North−Holland)
[26] Spears W M 1993 *Crossover or mutation? Foundations of Genetic Algorithms 2*
[27] Praveen RS, Tai-hoon K 2009 Application of Genetic Algorithm in Software Testing *International Journal of Software Engineering and Its Applications* **3(4)** 87-96
[28] Coyle L, Cunningham P 2004 Improving recommendation ranking by learning personal feature weights *Proc. 7th European Conference on Case-Based Reasoning* 560-572

Acknowledgments
The work was supported by a grant from the Ministry of Education and Science of the Russian Federation in the framework of the project part of the state task, the project № 2.2327.2017/4.6.