Method of application of the genetic algorithm for automatic generation of test data

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

[1]Novosibirsk State Technical University, K. Marks avenue 20, Novosibirsk, 630073

Abstract. Software testing has always been a time-consuming process, without obvious results but not less important than any other stage of software engineering. One of the major issues of testing is the development of initial test data that allow complete evaluate the program code and find most important parts of the program. Intelligent technologies, in particular, genetic algorithm allow to define initial data sets allowing to evaluate the values of variables for qualitative testing. This article proposes the method of possible integration of genetic algorithm with the system for generating test data.

1. Introduction

One of the most important stages in the software development is testing. The correspondence of the developed program to the specified requirements, observance of logic in the processing of data and obtaining the correct final results - all this is the testing purposes.

The processes of verification and validation of the program code are improving quite slowly. Development of most types of testing designs and templates is often done manually, without using any automation systems. Consequently, the testing process becomes incredibly complex and consuming, both in time and finance, if take it seriously. To test some programs, it may need up to 50% of the whole time.

One of the main purposes of testing is to create a test array that would ensure sufficient level of quality of the final product, by checking most of the various branches of the code. Nevertheless, the task of finding many branches of code consists of several more local tasks, the solution of which is necessary for the best possible finding of anappropriate testdata. One of the important tasks needed to be solved is the searching of most complex branch of the code. Subsequently, the task is extrapolated to find several important branches. In present paper, we solve the local problem of finding one optimal branch of the program.

Based on the foregoing, we can conclude that automation of testing, or at least the automation of test development, can significantly reduce not only the time costs, but also money. There are other advantages, not so obvious - a greater probability of finding small errors, transparency of test development, testing simultaneously with the development of the program, etc. [1]

Obviously, given the advantages of the process of the automation of testing, the researchers proposed different approaches to this process. For example, in the work [2] it was proposed an approach to automate the programming process of complex production systems in accordance with the standard IEC 61131.

Testing is not a standardized process, it depends on many factors, most of which vary from one program to another. Therefore, scientists came to a conclusion about usage of some research in the field of artificial intelligence, which would allow the development of a hybrid system for automatic finding of tests array [3,4].

The paper is organized as follows. In Section 2, the problem definition is given, and the process of distributing the weights along the code branches is described. Section 3 is devoted to the description of the genetic algorithm, its terminology and method of finding a solution. In Section 4, a detailed description of the operation of the algorithm and the results of the work are presented. Section 5 proposes an additional improvement of the algorithm and shows the results of the specially developed program. In section 6 we give conclusions on the investigation of the algorithm.

2. Genetic algorithms

Formally, the genetic algorithm is not an optimization method, at least in the terms of classical optimization. Its goal is not to find the optimal and best solution, but rather close to it. Therefore, this algorithm is not recommended to use if quick and well-developed optimization methods already exist. But at the same time, the genetic algorithm perfectly shows itself in solving non-standardized tasks, tasks with incomplete data, or those for which it is impossible to use optimization methods because of the complexity of implementation or the duration of implementation [6, 7].

The genetic algorithm is considered complete if a certain number of iterations has been passed (it is desirable to limit the number of iterations, since the genetic algorithm works on the trial-and-error basis, which is a fairly long process), or if a satisfactory fitness function has been obtained. The concept of the fitness function is one of the most important in the whole method. As a rule, the genetic algorithm solves the problems of maximization or minimization and the adequacy of each solution (chromosome) is estimated using the fitness function.

The genetic algorithm works according to the following principles:

- ***Initialization***. The fitness function is defined. An initial population is formed. In the classical theory, the initial population is formed by random filling of each gene in chromosomes. But to increase the rate of convergence of the solution, the initial population can be specified in a certain way, or random values are analyzed in advance to exclude definitely not suitable genes.
- ***Evaluation of population*** [8]. Each of the chromosomes is evaluated by the fitness function. Based on the set requirements, the chromosomes get an accurate value of how well they correspond to the problem being solved.
- ***Selection***. After each chromosome has its own fitness value, the best chromosomes are chosen. Selection can be made by different methods, for example, the first *n* chromosomes from the sorted order, or only the most suitable ones, but not less than *n*.
- ***Crossing*** [9]. The first is significantly different from conventional methods. After selection of the chromosomes that are suitable for solving the problem, they are crossed among themselves. Random chromosomes from all the "chosen" ones in a random order form new chromosome. Crossing occurs on the basis of the choice of a certain position in two chromosomes and replacement of parts of each other. After the required number of chromosomes is filled to create the population, the algorithm proceeds to the next step.
- ***Mutation***. Also a step that is characteristic only for GA. In random order, a random gene can change to random value. The main point in mutation is the same as in biology - to bring genetic diversity to the population. The mutation makes it possible to obtain solutions that could not be obtained with the available genes. Firstly, this helps to avoid getting into local extremes, since the mutation can allow the algorithm to be translated into a completely different branch and, secondly, to "dilute" the population in order to avoid the situation when only the same chromosomes will be in the whole population, which will not generally move towards a right solution.
- After all the steps have been completed, it is estimated whether the population has reached the desired accuracy or has come to limit the number of populations, and if so, the algorithm stops working. Otherwise, the cycle is repeated with the new population until conditions are reached.

3. Problem description

The usage of genetic algorithms in the testing process makes it possible to find the most complex parts of the program, in which the risks due to the assumption of errors are the greatest. Evaluation is due to the use of the fitness function whose parameters are the different weights of each individual operation [5].

By now, many kinds of diagrams have been developed that allow to represent the structure of the program not as a set of actions, but as diagrams with a certain structure. One of such types are diagrams (graphs) of control flows, allowing to represent the whole set of ways of program execution. The main purpose of such diagrams is the describe the program, i.e. to determine the complexity of the code, to check the logic and to directly determine the steps of the program. But if you look at the problem of generating test data, then such a type of diagrams, built on thealready created computer program, allows you to evaluate the quality of the developed code and, within the scope of the task, to assess the importance or complexity of certain branches of the program.

In accordance with the above it was developed a method according to which it would be possible to evaluate the program code and to determine a set of test data that would allow "to walk through" the largest number of operations. The first step is to consider the structural elements of the code. For ease of presentation, you can use the flow diagrams that were mentioned earlierto visualize the structure of the code and to understand in which way the program is executed.

Each action is assigned its own separate block (node), and the directed relationship indicates the direction to the following action. For example, a condition is awarded to one block, but two branches of code will come out of it. Each transition between blocks is assigned a certain weight, depending on which part of the code this action is in, preceded by any complex structural elements, etc.[10].

The weight of each transition is determined by the Pareto rule: 80 percent of the weight on the complex code branch, and 20 percent on the simple one. With the same value of "complexity", the separation occurs on the 50 / 50 basis. For example, if in the code there is an ordinary condition, with one branch with a positive result of the comparison, the action of this condition will account for 80 percent of the weight, and 20 percent for the branch leading to the continuation (without action under this condition). But if both branches are present in the condition, that is, if the condition is fulfilled and not fulfilled, then the weights will be distributed 50% to 50%. An example based on which the algorithm was tested is shown in Figure 1.

The assigned weights can be used to develop test array with the help of genetic algorithms, that is, to estimate how much the calculated weight falls on this or another branch for certain values of the input parameters.

For the sake of convenience, we introduce the following notations:

X – datasets;

F(X) – the fitness function value for each data set depending on the calculated weights.

The task is to maximize the objective function, i.e.

$$F(X) \rightarrow max.$$

4. Results

To test the method, four randomly generated datasets are used: (10,5,12); (3,4,10); (25,30,11); (5,3,17). Table 1 shows the datasets, the calculated fitness function value and the rank determining the best data set.

Table 1. Initial datasets.

№	Dataset 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

In this case, the second and the third variants will be used as the data sets for selection. In order to obtain additional two new variants, their values will be mixed and supplemented by some probability of mutation.

```
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. }
```

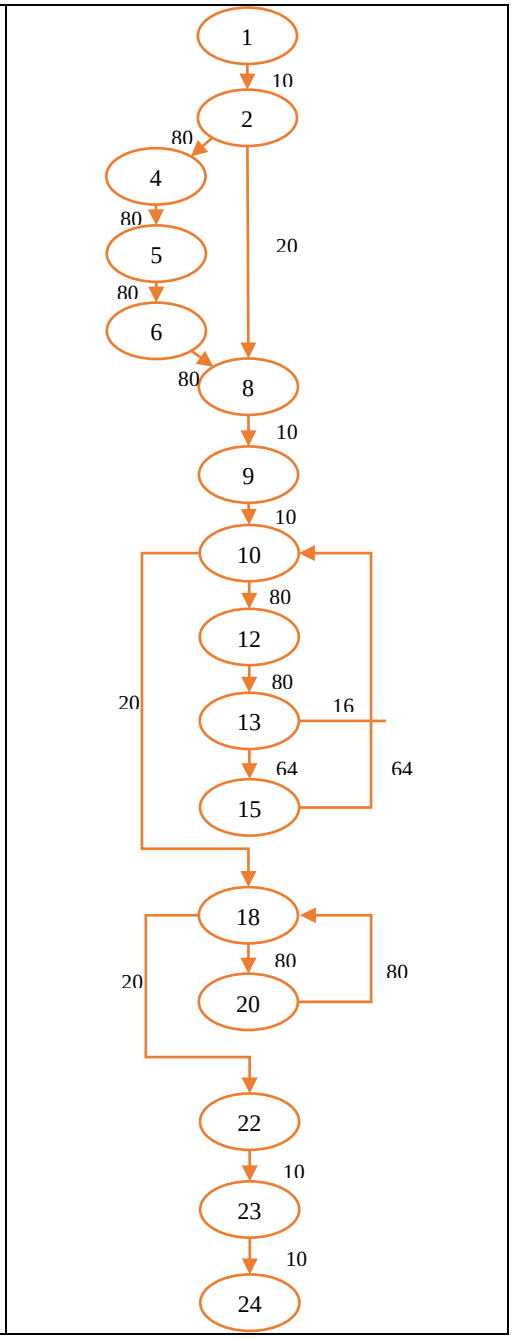

Figure 1. Program code and sequence of blocks of the analyzed program.

Division of the sets will occur at the first and second positions: when crossing (X, X, X) and (Y, Y, Y) the variables (X, X, Y), (X, Y, Y), (Y, X, X) and (Y, Y, X) will be obtained. Parental values remain to maintain the purity of the crossing, i.e. in comparison with the zero generation in the first generation the two new sets will be added. In subsequent generations, six sets of data will be stored.

The mutation will occur with a probability of 10 percent for the chance of changing the value from 1 to a given value in both directions. Under these conditions, the maximum possible additional value for the mutation is 5. As a result of crossing, the data sets presented in Table 2 will be obtained.

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

With equal ranks, priority will be given to the data sets of a newer generation. In the last generation, three sets of data were obtained, which check most of the branches of the program -

(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 as a result there will be two variants - (3,30,11) and (25,30,10).

Table 2. New data sets obtained as a result of crossing.

№	X	Y	Dataset	**Matation**
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)**

Table 3. The first generation of test data.

№	Dataset 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

Due to the small initial sample and a small size of code, the datasets quickly find intersecting values - 30 at the second position and 10 at the third one. Therefore, to continue iterating ceases to make sense - in the next generation the sets will consist mainly of repeating sets.

For this code, onecan use test datasets obtained in the last generation. Priority depends on the rank obtained.

Thus, using genetic algorithms, it is possible to find initial test values that would fully test most complex program variants depending on the assigned weights. This is testing through the maximization of the fitness functions, since the most widely describing test variant passes through those branches having the greatest weight.

As the values of chromosomes, the test sample values themselves appear. Many test cases in one iteration become a population. Using the genetic algorithm, it becomes possible to view a set of test samples to find the best initial variants.

5. Improvement of the algorithm

In accordance with the previously considered variant of using, this method is being developed to better correspondence to the real requirements. The weight is considered in accordance with the operability of the program, in other words, the more iterations the program executes, the greater weight will have the original test version. It allows you not only to estimate which branches of the program will be used for such data, but also how they change directly during the program operation and what results will be obtained.

This not only makes it possible to determine the initial test dataset, but also on the basis of the final population to draw a conclusion about changing and modification of data in accordance with the predefined logic. Four additional tests were conducted to present the results.

The first population is formed by random values. Each population contains 100 chromosomes. The total number of the population is also equal to 100. This formed a sufficient number of different variants and the best of them will be selected.

Table 4 shows the results of testing.

In each of the tests, at least two different versions of the data sets were formed, under which the program code will work most of the time and more times will take place inbranches. In addition, you can see certain patterns in the results - the first value is always maximum (random values were limited to 100), the second value is less than the first, but more than the third.

Table 4. Comparison of the 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**

6. Conclusion

The task of determining the most suitable test data is very difficult to solve with the help of standard algorithms, since the actual number of the entire array of possible values is incredibly huge, and it is not possible to select really suitable values. At the same time, automation of this process can significantly reduce the analytical burden on users involved in testing.

The use of genetic algorithms allows to compare many different data options for testing a program. The wide possibilities for improvement allow increasing the number of initial test variants, the number of generations and adding new properties, which helps to substantially increase the possibilities of finding more suitable variants. If you follow the passed graphs and reduce the weights of graphs being most often encountered in different variants, you can search for new paths that may not be available at the moment but can be as important as the most frequently encountered ones.

The generated tests can serve not only for testing the algorithm, but also for analyzing its performance. The data in its pure form already allow us to determine the laws of the program code, and with a deeper study they can allow us to draw conclusions about a possible subsequent improvement. Such an analysis can be used as a topic for further research in the field of data analysis.

In addition, it is necessary to solve several ambiguous tasks, because of which a simple assignment of weights cannot provide a qualitative search for the branches of the program:

- Transitions between different parts of programs. For example, in object-oriented programming it can be the launch of other subclasses, each of which has its own paths, the launch of other functions, and the calculation of changes in values. A lot of recommendations for programmers are related to the fact that the most difficult to understand parts are taken out in separate procedures, which makes it very difficult to find really important parts of the program.
- Loops that significantly complicate an adequate evaluation of the code, because some may occupy a huge share in the resource costs, but at the same time have a fairly simple logic. As a result, it becomes difficult to assess whether any particular cycle is an important part of the program and in fact how many iterations can be run under certain conditions.
- Insufficient code development. Often an input restriction is introduced outside the code, because of which, when using a genetic algorithm, a data set can potentially be caught, under which the program will never be executed. On the one hand, it helps to find such problem places, on the other hand, it puts a restriction directly on the algorithm.

These are not the only problematic tasks, because of which there are difficulties in correctly assessing the operability of the program. Nevertheless, their solution will allow not only to fulfill the current given goal of the proposed method, but also to obtain additional benefits from its use, such as determining non-optimal parts of the code, unnecessary operations, "littering" the code, and so on. As a result, further research will allow us to obtain new, previously undefined results.

7. References

[1] Zanetti M C, Tessone C J, 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] John K and Tiegelkamp M 2010 *Programming Industrial Automation Systems: Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids* (Springer, Berlin) p 388
[3] Luger F 2009 *Artificial Intelligence Structures and Strategies for Complex Problem Solving* (University of New Mexico) p 679
[4] 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
[5] Praveen R S and Tai-hoon K 2009 Application of Genetic Algorithm in Software Testing *International Journal of Software Engineering and Its Applications* **3(4)** 87-96
[6] Serdyukov K and 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
[7] Yang H L and Wang C S 2008 Two stages of case-based reasoning - Integrating genetic algorithm with data mining mechanism *Expert Systems with Applications* **35** 262-272
[8] Mühlenbein H 1992 How genetic algorithms really work: Mutation and hillclimbing *Parallel Problem Solving from Nature* **2**
[9] Spears W M 1993 Crossover or mutation? *Foundations of Genetic Algorithms* **2**
[10] Coyle L and Cunningham P 2004 Improving recommendation ranking by learning personal feature weights *Proc. 7th European Conference on Case-Based Reasoning* 560-572

Acknowledgments
The work is supported by a grant from the Ministry of Education and Science of the Russian Federation within of the project № 2.2327.2017/4.6.