The Study of the Sequential Inclusion of Paths in the Analysis of Program Code for the Task of Selecting Input Test Data

Konstantin E. Serdyukov[0000-0003-4162-1295] and Tatiana V. Avdeenko[0000-0002-8614-5934]

Novosibirsk State Technical University, K. Marks avenue 20, 630073 Novosibirsk, Russia
zores@live.ru

Abstract. The article proposes the results of a study evaluating one and many paths of program code, the work is not finished yet. To solve the problem of generating data sets, it is proposed to use a genetic algorithm for determining the complexity of program code for many paths. The proposed method allows to generate data for multiple code paths by sequentially eliminating operations to achieve maximum possible code coverage. For each generated data set the operations on the path are excluded from following generation, which allows generating new sets along other paths. The article describes the results of this method with a description of the data generation method and analysis of the generated data sets for many paths.

Keywords: genetic algorithm, test data, control flow graph, fitness function, testing process, program code, optimization method, test data generation, code coverage

1 Review of Existing Researches of Testing Automation and Selection of Test Data

Software engineering is a comprehensive, systematic approach to the development and maintenance of software. When developing programs, the following stages are most often distinguished - analysis, design, programming and testing. At the analysis stage, software requirements are determined and documentation is performed. At the design stage, the appearance of the program is detailed, its internal functionality is determined, the product structure is developed, and requirements for subsequent testing are introduced. Writing the source code of a program in one of the programming languages is done at the programming stage.

One of the most important steps in developing software products is testing. The important goals of testing are the compliance of the developed program with the specified requirements, adherence to logic in the data processing processes and obtaining correct final results. Therefore, for testing it is very important to generate input data, on the basis of which the program will be checked for errors and compliance with specified requirements. To assess the quality of the input data, the code coverage indicator is used, that is, what percentage of the entire program can the test suites "cover". It is determined by the ratio of the tested operations to the total number of operations in the code.

Some software code testing processes are improving quite slowly. The development of most types of test scenarios is most often done manually, without the use of any automation systems. Because of this, the testing process becomes incredibly complex and costly both in time and in finances, if you approach it in all seriousness. Up to 50% of all-time costs can be spent testing some programs.

One of the main goals of testing is to create a test suite that would ensure a sufficient level of quality of the final product by checking most of the various paths of the program code, i.e. would provide its maximum coverage. Nevertheless, the task of finding many paths itself consists of several subtasks, the solution of which is necessary to find a high-quality data set. One of the local problems that can be solved to find a test sets is to determine one of the most complex code paths.

For the most part, validation and verification of software products is difficult to optimize. It is especially difficult to automate the generation of test data, which for the most part is done manually.

However, there are many studies using non-standard algorithms to solve the automation problem. For example, in [1], it is proposed to use a constraint-based algorithm for the Mort system, which uses error testing to find input test data. Test data is selected in such a way as to determine the presence or absence of certain errors.

Quite often, genetic algorithms are used in one way or another to solve this problem. The article [2] compares different methods for generating test data, including genetic algorithms, random search, and other heuristic methods.

In [3], to solve the problem, it is proposed to use constraint logic programming and symbolic execution. In [4], the Constraint Handling Rules (CHR) are used to assist in the manual verification of problem areas in the program.

Some researchers use heuristic methods with help of visualization instruments to automate the testing process, such as a data flow diagram. Studies of automation methods using this diagram have been proposed in the articles. [5, 6, 7, 8]. In article [5] it is proposed to additionally use genetic algorithms with to determine new input test data sets based on previously used ones.

The articles [9, 10] consider hybrid automated systems for generating test data. In [9], an approach is used that combines strategies of random strategy, dynamic symbolic execution and search-based strategy. The article [10] 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 [11] 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 [12]. The proposed approach uses the neural network structure with user-configured clustering of input data for sequential learning.

To generate test data, a novelty search approach can also be applied. In article [13] is proposed to use this approach to view large input data spaces and compares it with the approaches based on the genetic algorithm.

The possibilities of generating test data for testing web services are also being investigated, for example, in the WDSL specification [14].

For the convenience of generating test data, the UML diagrams are also used. [15, 16]. 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 [17] 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 [18]. In paper [19] discusses the problem of tree building and proposes an integrated classification tree algorithm, and in [20] 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. This article proposes a comparison of different methods for evaluating code complexity for generating test data. The article is structured as follows. The second section describes the problem to be solved. In section 3 there is comparing of the different code evaluation methods and introduction of a new method for assessing code complexity. Section 4 proposes the results of the operation of the input data generation algorithm using the introduced code estimation method.

2 Problem Description

Using genetic algorithms in the testing process allows us to find the most complex parts of the program in which the risks due to errors are greatest. Evaluation is due to the use of the fitness function, the parameters of which are the weights of each passable operation. Definition of weights, i.e. the complexity of the program code occurs due to various metrics used depending on the requirements for the sets.

The task of finding input test data consists of three subtasks:

1. Search for input data for passing along one of the most complex code paths. Difficulty is determined by the metric chosen to evaluate the code.
2. The exclusion or reduction of the weights of operations on the path for which the data were selected, based on the fitness function for other paths.
3. Generation of input test data for many paths of program code.

The limit on the number of sets of input data is established after the development phase and will allow you to concentrate on certain paths.

The whole algorithm is performed cyclically - the procedure for searching for input data for one branch is started, after which operations in this branch are excluded from further calculations and the data search for one branch is started again.

3 Methods of Defining Software Complexity

For the convenience of defining methods for calculating complexity we use the term metrics, which is used in programming to determine the complexity or quality of the written program code. There are a fairly large number of metrics divided roughly into the following classes:

- quantitative - focused on calculating the number of operations, conditions, cycles, function calls, etc.;
- complexity of the program control flow - evaluates the program code according to the constructed control graph;
- complexity of the data control flow - an assessment is made of the use of data, in other words, variables, and how they interact in the program;
- object-oriented - focused on special features of object-oriented programming, such as classes, inheritance, etc.

To determine the complexity of the program code the easiest to calculate and most common metrics from the quantitative class are used. This is not only because the fact that they are quite easy to understand and use in any type of program, but also to the fact that such metrics are much easier to analyze. Comparing the calculated values, you can understand what kind of data goes along more complex paths, how much this path is more difficult than the others. Such an analysis may not be entirely accurate, but at the same time it allows using of numerical values to compare different path of the program code.

The easiest metric in terms of evaluating program code is SLOC (abbr. Source Lines of Code) - the number of lines of code. This metric takes into account only the total number of lines of code in the program, which makes it very easy to understand. In our case, the number of lines refers to the number of commands, and not the physical number of lines.

The ABC metric, or Fitzpatrick metric, is a measure that is calculated by three different variables ABC = <n_a, n_b, n_c>. First variable n_a (abbr. Assignment) allocated to lines of code that are responsible for assigning a certain value to variables, for example, int number = 1. Variable n_b (abbr. Branch) responsible for the use of functions or procedures, i.e operators, that work out of sight of the current program code. Variable n_c (abbr. Condition) counts the number of logical operators such as conditions and loops. The metric value is calculated as the square root of the sum of the squared values n_a, n_b and n_c.

$$\mathrm{F} = \sqrt{\mathrm{n}_a{}^2 + \mathrm{n}_b{}^2 + \mathrm{n}_c{}^2} \tag{1}$$

It is notable that one line of code can be taken into account in different indicators, for example, when assigning a variable the value of a certain function (double number = Math.Pow (2, 3), assigning the variable number value 2 to the power 3 is taken into consideration both in n_a and n_b). The disadvantages of this metric include the possible return of a zero value in some parts of the code.

The Jilb metric is aimed at determining the complexity of program code depending on the number of conditional operators. This metric is useful for determining the complexity of program code, both for writing and for understanding:

$$\mathrm{F} = cl/n, \quad (2)$$

where cl is number of conditional operators, n is lines of the code.

In the developed method for calculating the exponent cl, all logical operators in the code are taken into account, that include both cycles and conditions. Nevertheless, it is possible to use this metric to determine the saturation of the code only with certain operators (either loops or conditions).

The last metric is a modified SLOC metric using the rules for assigning and changing weights to operations depending on the degree of nesting.

In the current implementation, weight reduction of 80% is applied for each nesting. The algorithm can be described in more detail as follows:

- The first operation in the program code is assigned a weight, standardly, of 100 units;
- Each subsequent operation is assigned a weight by logic - if there are no conditions or cycles, then the weight is taken equal to the previous operand;
- Conditions determine the weight in accordance with the rule - if the condition contains only one branch (only if ...), then the weight of each nested operand is reduced by 80%. If the condition is divided into several branches (if ... else ...), then the weight is divided into equivalent parts - for two paths 50% / 50%, for three 33% / 33% / 33%, etc.;
- All nested weight changes are multiplied, for example, for operators nested in two conditions, the weight of operations will be calculated as 80% * 80% = 64%.

The amount of weight reduction is customizable and can be used to set priorities when selecting data and, for example, when dividing paths do not go into insignificant paths with many synonymous operations. But you can specify the distribution of the weights of the operators and more than 100%, that is, each nested operand will increase its weight. An example showing a method for determining the weight of groups of operations is presented in Figure 1.

For testing purpose, the program used to show weight assignment have limited number of path and weight for every operation can be shown. The program that is used for generating test data sets is written in such way that weight for every operation assign automatically, so it can be used to generate data for much complex code with many conditions.

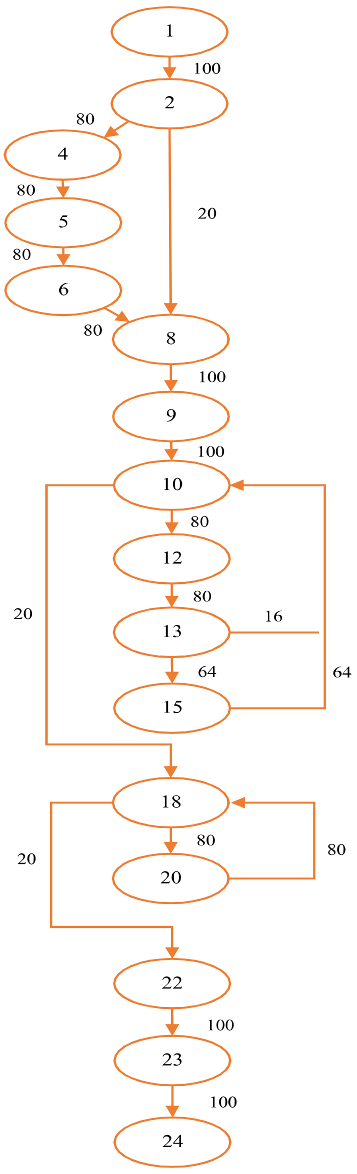

Fig. 1. Example of the weight assigment

Assigned weights can be used to develop test cases using genetic algorithms, that is, to evaluate how much calculated weight falls on this or that branch for certain values of input parameters.

For convenience, we introduce the following notation:

X - data sets;

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

The goal is to maximize the objective function, i.e. F (X) → *max*.

4 Selection of Test Data Sets for One and Many Paths

To select data for many paths, firstly it is necessary to show how the method works on one path. Modified SLOC metric is used for assessing complexity, the description of which is given above.

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 1 presents 4 runs of the method for the program on the Figure 1 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 shown.

Table 1. Different runs of the test data set for one path

Population	Run 1	Run 2	Run 3	Run 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	*1: 95, 64, 54*	*1: 97, 80, 4*	*1: 99, 13, 10*	*1: 99, 71, 60*
	2: 95, 64, 29	2: 97, 80, 29	2: 99, 13, 11	2: 99, 71, 45

As it can be seeing, at least 2 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 this program code test data sets obtained in the latest generation can be used as initial data sets to testing.

So, using genetic algorithms, can be found such initial test values, which would fully check all the variants of the program depending on the assigned weights. This is verified through the maximization of fitness functions, since the most widely describing test variant passes through those branches in which the largest number of operations with high weights.

As mentioned earlier, in order to select data for many paths of the program code, operations that has already been selected could be sequentially excluded. Due to the fact that the program in Figure 1 has rather few paths, it is not well suited to present the results of test data selection for many paths. Therefore, for the next test, another program is used with 70 operations and much more conditions and cycles.

To confirm the operability of the method, some of the paths of the program code are preliminarily enclosed in conditions that cannot be achieved. As a result, 8 operations cannot be executed, so code coverage cannot be 100%. Table 2 presents the results of running the method and it can be clearly seen that these paths are not achieved when selecting data.

Table 2. Test data sets for many program paths

Iteration	Data Set	Function	Code coverage
1	(77, 18, 99)	164 100	41% (29/70)
2	(60, 41, 61)	26 400	53% (37/70)
3	(5, 8, 56)	18 882	74% (52/70)
4	(40, 30, 55)	9 900	79% (55/70)
5	(99, 73, 73)	12 000	86% (60/70)
6	(84, 22, 64)	5 000	**89% (62/70)**
7	(48, 72, 44)	0	89% (62/70)
8	(36, 2, 36)	0	89% (62/70)

For the first 6 iterations of operation, the algorithm went over all possible code paths. In total, code coverage was 89%, with the exception of those operations that are on impassable paths. You may notice that the fitness function is evenly reduced, with the exception of the 5th iteration - this is due to the fact that the initial data is randomly selected which is one of the problem of genetic algorithm, there is solution not always tend to convergence to the best. For this reason, the chances of getting into certain paths of the program code are not uniform, nevertheless, all the paths of the program code were passed and the data was selected.

Due to the fact that the data is initially randomly selected, the chance of reaching a certain path can be quite small due to possible restrictions. This problem can be solved in several ways, for example, to limit the boundaries of the selected values due to the introduction of domains or increase the chance of mutation. An increase in the chance of a mutation will allow branching out due to a greater number of combinations, but may adversely affect the overall convergence rate of the method.

5 Conclusion

The selection of input test data is an important task to ensure high quality testing. This is due to the fact that what parts of the code the tester can verify depends on the input data. Therefore, ensuring maximum coverage of the code is one of the most important goals when selecting test data.

In the proposed method, there are two tools for providing maximum coverage. Different metrics have different functionality and can lead to the selection of data for different paths. Each metric has a different way for determining fitness functions and can be used to fulfill different testing requirements. If the metric is ineffective for the tested program code, it can be changed to another.

There is still much room for further research in this area. Starting from the implementation of other common metrics and ending with the development of a hybrid metric that can be flexibly configured to solve specific problems.

Achieving maximum coverage is ensured through the use of a genetic algorithm. Despite the fact that the method works quite efficiently, there is a lot of room for improvement. The selection of test data for one path quickly came to one solution and most of the generation wasted. Therefore, in this part, the introduction of additional restrictions is necessary, since the efficiency of the algorithm as a whole directly depends on this.

When selecting data for multiple paths, it is also important to propose additional methods. In particular, it is planned to introduce an adaptive mutation, which depends on how high a level of coverage has already been achieved and whether data with a zero value of the fitness function , but not with a maximum coverage, have been obtained.

Acknowledgments

The reported study was funded by RFBR, project number 19-37-90156.
The research is supported by Ministry of Science and Higher Education of Russian Federation (project No. FSUN-2020-0009).

References

1. DeMillo, R.A., Offutt, J.A.: Constraint-Based Automatic Test Data Generation. IEEE Transactions on Software Engineering, 17 (9), pp. 900–910 (1991)
2. Maragathavalli, P., Anusha, M., Geethamalini, P., Priyadharsini, S.: Automatic Test-Data Generation For Modified Condition/ Decision Coverage Using Genetic Algorithm. International Journal of Engineering Science and Technology, 3 (2), pp. 1311–1318 (2011)
3. Meudec, C.: ATGen: Automatic Test Data Generation using Constraint Logic Programming and Symbolic Execution. Software Testing Verification and Reliability (2001)
4. Gerlich, R.: Automatic Test Data Generation and Model Checking with CHR. 11th Workshop on Constraint Handling Rules (2014)

5. Girgis, M. R.: Automatic Test Data Generation for Data Flow Testing Using a Genetic Algorithm. Journal of Universal Computer Science, 11(6), pp. 898-915 (2005)
6. Weyuker, E. J.: The complexity of data flow criteria for test data selection. Inf. Process. Lett., 19(2), pp. 103–109 (1984)
7. Khamis, A., Bahgat, R., Abdelaziz, R.: Automatic test data generation using data flow information. Dogus University Journal, 2, pp. 140–153 (2011)
8. Singla, S., Kumar, D., Rai, H. M., Singla, P.: A hybrid pso approach to automate test data generation for data flow coverage with dominance concepts. Journal of Advanced Science and Technology, 37, pp. 15–26 (2011)
9. Liu, Z., Chen, Z., Fang C., Shi, Q. Hybrid Test Data Generation / State Key Laboratory for Novel Software Technology. ICSE Companion 2014 Companion Proceedings of the 36th International Conference on Software Engineering, pp. 630-631 (2014)
10. Harman, M., McMinn, P.: A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search. IEEE Transactions on Software Engineering, 36(2), pp. 226-247 (2010)
11. Xing, Y., Gong, Y., Wang, Y., Zhang, X. :A Hybrid Intelligent Search Algorithm for Automatic Test Data Generation. Mathematical Problems in Engineering, (2015)
12. Paduraru, C., Melemciuc, M. C.: An Automatic Test Data Generation Tool using Machine Learning, 13th International Conference on Software Technologies (ICSOFT), pp. 472-481 (2018).
13. Boussaa, M., Barais, O., Sunyé, G., Baudry, B.: A Novelty Search Approach for Automatic Test Data Generation. 8th International Workshop on Search-Based Software Testing (2015)
14. Lopez, M., Ferreiro, H., Castro, L. M.: A DSL for Web Services Automatic Test Data Generation. 25th International Symposium on Implementation and Application of Functional Languages (2013)
15. Doungsa-ard ,C., Dahal, K., Hossain, A. G., Suwannasart, T.: An automatic test data generation from UML state diagram using genetic algorithm. IEEE Computer Society Press, pp. 47-52 (2007)
16. Sabharwal, S., Sibal, R., Sharma, C.: Applying Genetic Algorithm for Prioritization of Test Case Scenarios Derived from UML Diagrams. IJCSI International Journal of Computer Science Issues, 8(3(2)) (2011)
17. Doungsa-ard, C., Dahal, K., Hossain, A., Suwannasart, T.: GA-based Automatic Test Data Generation for UML State Diagrams with Parallel Paths. Part of book Advanced design and manufacture to gain a competitive edge: New manufacturing techniques and their role in improving enterprise performance, pp. 147-156 (2008)
18. Grochtmann, M., Grimm, K.: Classification trees for partition testing. Software Testing. Verification and Reliability, 3(2), pp. 63–82 (1993)
19. Chen, T.Y., Poon, P.L., Tse, T.H.: An integrated classification-tree methodology for test case generation" International Journal of Software Engineering and Knowledge Engineering, 10(6), pp. 647–679 (2000)
20. Cain, A., Chen, T.Y., Gran,t D., Poon, P., Tang, S., Tse, T.H.: An Automatic Test Data Generation System Based on the Integrated Classification-Tree Methodology. Software Engineering Research and Applications, Lecture Notes in Computer Science, 3026 (2004).