applied sciences

MDPI

Article

Automated Test Data Generation Based on a Genetic Algorithm with Maximum Code Coverage and Population Diversity

Tatiana Avdeenko * and Konstantin Serdyukov

Applied Mathematics & Computer Science Department, Novosibirsk State Technical University, 20 Karl Marx av., 630073 Novosibirsk, Russia; zores@live.ru

* Correspondence: tavdeenko@mail.ru

Abstract: In the present paper, we investigate an approach to intelligent support of the software white-box testing process based on an evolutionary paradigm. As a part of this approach, we solve the urgent problem of automated generation of the optimal set of test data that provides maximum statement coverage of the code when it is used in the testing process. We propose the formulation of a fitness function containing two terms, and, accordingly, two versions for implementing genetic algorithms (GA). The first term of the fitness function is responsible for the complexity of the code statements executed on the path generated by the current individual test case (current set of statements). The second term formulates the maximum possible difference between the current set of statements and the set of statements covered by the remaining test cases in the population. Using only the first term does not make it possible to obtain 100 percent statement coverage by generated test cases in one population, and therefore implies repeated launch of the GA with changed weights of the code statements which requires recompiling the code under the test. By using both terms of the proposed fitness function, we obtain maximum statement coverage and population diversity in one launch of the GA. Optimal relation between the two terms of fitness function was obtained for two very different programs under testing.

Keywords: genetic algorithm; test data generation; fitness function

Citation: Avdeenko, T.; Serdyukov, K. Automated Test Data Generation Based on a Genetic Algorithm with Maximum Code Coverage and Population Diversity. *Appl. Sci.* **2021**, *11*, 4673. https://doi.org/10.3390/app11104673

Academic Editor: Evgeny Nikulchev

Received: 30 April 2021
Accepted: 18 May 2021
Published: 20 May 2021

1. Introduction

Classic software engineering lifecycles include such important stages as requirements engineering, design of the software architecture, implementation (or coding), testing and maintenance. In this sequence, the software testing is a process of investigation of the software product aimed at checking the correspondence between actual behavior of the program code and its expected behavior on a special set of tests (the so-called test cases) selected in a certain way. The testing stage is a very costly one, taking up to 40–60 percent of the total software development time.

The goal of testing is to ensure accordance of the developed program with the specified requirements, compliance with logic while data processing, and obtaining of the correct final results. There are two major testing techniques allowing the checking of the software under testing (SUT) for errors with different levels of access to the code. These approaches are black-box testing and white-box testing [1]. The black-box testing considers the software as a "black box" investigating functionality without seeing the source code. Its objective is to find out when the input-output behavior of the SUT does not agree with its specification. It is also called functional or specification-based testing.

On the contrary, the white-box testing examines the internal code structure and behavior of the SUT by execution of the source code. The tester chooses the SUT's inputs to exercise paths through the code and determine the appropriate outputs. It is also called program-based or structural testing. Test data are derived from the SUT's input domains. In some cases, the test data is already available, but in most cases, it is required to be generated.

Test data generation is a complex and time-consuming process which needs a lot of effort and a large budget. Therefore, automation of this process, at least partially, is an urgent research problem, the solution of which could improve the efficiency of the software testing. One of the goals of the automatic test data generation is to create such a multitude of test data that would ensure a sufficient level of quality of the final product by checking most of the various code paths, i.e., to provide maximum code coverage to satisfy some criteria (for example, statement or branch coverage).

In paper [2], three types of test data generators were indicated: pathwise generators, data specification generators, and random generators. Data specification generators mainly refer to the black-box testing strategy while pathwise test data generators refer to the white-box testing. Random test data generation [3] is the simplest blind strategy that can be applied both for the black- and white-box testing. It is now used in its original form mainly as a lowest point for evaluating the quality of more efficient methods. However, there are several fairly successful improvements to this method based on the use of more advanced statistical modeling. Therefore, paper [4] suggests the improvement of the classic random method in the form of a random walk operator, which turned out to be very effective even in comparison with more sophisticated optimization methods. Another strategy for developing the random method is to use advanced statistical models. For example, in [5], a specially built Markov model is used to test the reliability of Unmanned Aircraft Systems, as they are of vital importance in practical applications. The experiments showed that the proposed method could reduce the redundancy of test data while ensuring the coverage ratio.

Black-box test data generation from software specification prepares test cases for software developers before the code development phase [6,7]. A combination between black-box and white-box testing is sometimes called gray-box testing. Gray-box testing measures the quality of test data using the software specification as in black-box testing, but also uses the internal information from the behavior of the software specification as in white-box testing.

The most advanced requirements specification model suitable for the gray box testing is the UML diagram. There are some publications based on UML. For example, in [8,9] it is proposed to use genetic algorithms to generate triggers for the UML diagrams, which allow finding the critical path in the program. Paper [10] proposes an improved method based on a genetic algorithm to select test data for many parallel paths in UML diagrams. In addition to UML diagrams, the specification can be displayed in the form of Classification-Tree Method [11]. The problem of constructing the trees was considered and an integrated classification tree algorithm was proposed [12] and the developed ADDICT prototype (AutomateD test Data generation using the Integrated Classification-Tree methodology) for an integrated approach was studied [13].

Having test cases before coding helps the developers to control their code to conform to the specification. However, the use of the already written code in the white-box testing allows one to build a more perfect test suite, which simultaneously possesses both the property of non-redundancy of the set of cases and optimal code coverage. Historically, the first approach to pathwise test data generation was a static method of the code analysis based on the use of symbolic evaluation considered in papers [14–18]. Symbolic evaluation involves executing a program using symbolic values of variables instead of actual values. After a path is selected, symbolic evaluation is used to generate a system of path constraints, i.e., a set of equalities and inequalities on the SUT's input variables. This system of path constraints must be solvable for the path to be traversed. A number of algorithms have been used for the inequality solution. There are different approaches to solving the system of constraints. Consequently, in [19], a set of tools (collectively called Godzilla) that automatically generates constraints and solves them to create test cases for unit and module testing was implemented. Ref. [20] proposed using Constraint Logic Programming and symbolic execution to solve the problem. In [21], Constraint Handling Rules are used to help in manual verification of problem areas in the computer program.

However, there are several problems connected with static approach. The first one is, of course, the problem of the complexity of symbolic computations, which limits the SUT size. The second problem is related to an array element determination when the index of an array depends on input values.

So, currently, the most effective and commonly used method for automatic test data generation in white-box testing is a dynamic approach which is based on actual execution of the SUT, optimization methods, and dynamic data flow analysis. Test data are developed using actual values of input variables. When the program is executed for some input variables, the program execution data flow is monitored. One of the first execution-oriented approaches was the chaining method [22], which used data dependency analysis to guide the search process. Dependency analysis automatically forms a sequence of statements that is to be executed prior to the execution of the current statement.

Studies of execution-oriented techniques using Data Flow diagrams were carried out in papers [23,24]. Some of the researchers suggest using hybrid approaches. For example, an approach proposed in [25] combines Random Strategy, Dynamic Symbolic Execution, and Search-Based Strategy. Ref. [26] proposed a hybrid approach based on the Memetic Algorithm for generating test data. Ref. [27] compared different methods for generating test data, including genetic algorithms, random search, and other heuristic methods.

As for optimization methods, currently, evolutionary approaches have proven a powerful tool for test data generation. A large number of publications on this topic investigated the use of a genetic algorithm (GA) to solve the problem of generating test data; see, for example, [27–30]. Note, however, that most often, when using GA, the researchers limit themselves to the so-called path-oriented generation approach, which does not provide the generator with a possibility of selecting among a set of paths, as in the goal-oriented generation approach, but focus only on one specific path [31]. As a result of running the GA, from the final generation of test cases individuals are selected that are the most fitted to traversing a given path. This can be either a path specified by the researcher, or the most complex path of the SUT. If the goal is to form a population of test cases that provides full coverage of all paths or at least some of the most complex paths of the program, then the classical GA is not suitable, since it drifts towards the most fitted (for example, the most difficult) path. That is, in the process of passing from one generation to another, all individuals will be more and more similar to each other. This problem can be solved by running the GA multiple times [32]. However, this approach cannot be called efficient, since the optimization problem is solved separately for each SUT path.

Researchers are trying to solve the indicated problem of the GA drift using other evolutionary methods. In [33], the simulated repulsion algorithm based on particle systems was proposed for the automatic generation of diversity-oriented test sets (DOTS) obtained by taking random test sets and iteratively improving their diversity. Another means of increasing the diversity of the test cases is the combination of the GA with the Particle Swarm Optimization (PSO) method.

Reference [34] compared two computational techniques, GA and PSO. It was noticed that PSO produces more distinct test cases than GA, but PSO can solve only discrete problems while the GA can solve both discrete and continuous ones. In [35], the GPSMA (Genetic-Particle Swarm Mixed Algorithm) was proposed using the individual's update mode to replace the mutation operation in the GA on the basis of population division. In [36], on the basis of the classical genetic algorithm, the algorithm divided the population into "families," influencing the convergence efficiency by crossover in family, keeping the diversity of the population by crossover between families. In [37], an approach for coupling-based integration testing of object-oriented programs using PSO was proposed.

With a certain success of hybrid evolutionary approaches to increase the diversity of the population, the problem of automating the test suite generation is far from a final solution. It should be noted that the majority of works use a simple formulation of fitness function that expresses the degree of coverage of an individual path with a test case. At the same time, in most papers, the mathematical form of the fitness function is not defined at

all, while this is a very important aspect of GAs. Due to the formulation of fitness function, it is possible to increase the diversity of the population, see, for example, [38,39].

In the present paper, we use the idea of an Improved Genetic Algorithm for Multiple Paths Automatic Software Test Case Generation proposed in [38], where an additional term responsible for the greatest diversity of the population, along with a term responsible for the complexity of each path, is included into the fitness function. We investigated this approach, identified its shortcomings, and proposed an improved form of the fitness function, as well as changes into the GA, allowing us to achieve a more uniform increase in the percentage of code coverage. Our research confirmed greater effectiveness of the proposed approach compared to the original version.

The paper is organized as follows. Section 1 introduces the problem and gives the literature review. Section 2 discusses theoretical issues of the research, including the mathematical description of the proposed method. In Section 3, we describe experimental results for two different SUT. Section 4 provides the discussion of the results.

2. Theoretical Background

2.1. Basic Concepts

In this paper, we use a dynamic approach to automatic test data generation, which is based on actual execution of a SUT, dynamic data flow analysis, and fitness function optimization. Test data is developed using actual values of input variables. When the program is executed on some input data, the program execution flow is monitored.

A control flow graph (CFG) of the program under testing is a directed graph $CFG = (V,\ R,\ v0,\ vE)$, where V is a set of graph nodes, R is a subset of cartesian product $V \times V$ determining a binary relation on V (a set of graph edges), $v0$ and vE are, respectively, unique entry and unique exit nodes $v0 \in V,\ vE \in V$.

For the sake of simplicity, we restrict our analysis to a subset of structured C-like programming language constructs, namely: sequencing statements (for example, assignments), conditional statements (*if-then-else*), cycle statements (*for* and *while*). A node in V corresponds to the smallest single-entry, single-exist executable part of a statement in P that cannot be further decomposed. A single node corresponds to an assignment statement, input or output statement, or the <expression> part of an *if-then-else* or *while* statement. An edge $(v_i,\ v_j) \in R$ corresponds to a possible transfer of control from the node v_i to the node v_j. The edge $(v_i,\ v_j)$ is called a branch. Each branch in the CFG can be labeled by a predicate, describing the conditions under which the branch will be traversed.

An input variable of SUT is a variable $var_i,\ i = \overline{1, N}$, which appears in an input statement, e.g., $read(var_i)$, or it is an input parameter of a procedure. Input variables may be of different types, e.g., integer, real, Boolean, etc. Let $(var_1,\ var_2,\ \ldots,\ var_N)$ be a vector of input variables of the SUT. The domain D_i of input variable var_i is a set of all values which var_i can hold. By the domain D of the SUT we mean a cartesian product $D = D_1 \times D_2 \times \ldots \times D_N$, where each D_i is the domain of input variable var_i, i.e., a set of all values that var_i can hold.

Thus, a SUT input is a single point in the N-dimensional input space D. A path P in a CFG is a sequence of nodes $P = \langle v0,\ v_{i_1},\ \ldots,\ v_{i_k} \ldots,\ vE \rangle$, such that $v0, vE, v_{i_k} \in V$, $\left(v_{i_k},\ v_{i_{(k+1)}}\right) \in R$. A path is feasible if there exists a SUT input for which the path is traversed during program execution, otherwise the path is infeasible.

The maximum goal of automated test data generation problem is to find the so-called test case, being a set of the SUT inputs $\{x_1,\ x_2,\ \ldots,\ x_m\},\ x_i \in D$, on which the whole variety of feasible SUT paths will be traversed. This problem can be solved gradually. One can initially formulate a partial purpose of finding one SUT input on which a given path P will be traversed. By sequentially choosing all possible SUT paths (or a subset of the most critical paths), one can iteratively find a test case that provides the greatest path coverage.

In order to solve the problem of finding a SUT input for a given path P, various optimization methods can be used. For example, in [29], the GA is successfully applied. In this case, as a fitness function it is reasonable to use a weight function that assigns nonzero

weights for those nodes or branches of CFG along which the considered path P passes, and zero values for those nodes or branches along which path does not pass. For example, in [29], to solve the above problem, a function is used of the following form

$$F(x) = \sum_{j=1}^{n(x)} w_j(x), \quad (1)$$

where $w_j(x)$ are the nonzero weights corresponding to the path of the CFG generated by the input value $x \in D$; $n(x)$ is the number of statements (or branches) covered by the path.

The application of the above approach to the search for a test case that covers the vast majority of graph paths is not optimal, since it involves multiple runs of the GA without taking into account the previous results. At the same time, the use of a different formulation of fitness function in the GA (introduction of additional terms into (1)) could make it possible to take care not only of the coverage of a specific CFG path, but also of the diversity of paths in the population, which allows hope for obtaining diverse test-cases that ensure maximum coverage of the most complex subsets of paths in one GA run. The idea of such a promising approach was proposed in [38]. However, the form of the fitness function presented in [38], in addition to containing uncertainties and inaccuracies, is unstable for various tested code. In the present paper, we propose a significant development and refinement of this approach, and also provide results on the study of its effectiveness for large dimensional SUT.

2.2. Genetic Algorithm for Test Data Generation

Genetic Algorithm borrows its idea and terminology from the biological world. In such a way, it uses different representations for potential solutions referred to as chromosomes, genetic operators such as crossover and mutation used to generate new offspring solutions, and also selection and evaluation mechanisms derived from nature.

With regard to the problem considered here, a set of generated test data, which best contributes to the software testing process, can serve as potential solutions. Depending on the values of the input variables supplied to the SUT input, the code execution process can follow various paths determined by the sequence of statements, among which there can be both linear statements following one after another, conditional statements (if-then-else), and loops (while, for), leading to branching of the computations. It is the latter that ultimately determines the variety of paths of the SUT execution.

In this paper, we assume that input variables $var_1, var_2, \ldots, var_N$ of the SUT can take their values from the continuous domains $D_1, D_2, \ldots, D_N$. Therefore, it is reasonable to use continuous (real-valued) GA (unlike the binary GA), where the values of input variables are the genes included into the chromosomes that determine potential solutions to the problem of generating input test data. Denoting chromosomes as x_i, we determine a test data population, consisting of m individuals each containing N genes (values of the input variables)

$$\{x_1, x_2, \ldots, x_m\}, \text{ where } x_i = \left[var_1^i, var_2^i, \ldots, var_N^i\right] \quad (2)$$

The genetic algorithm for test data generation includes the following main stages:

1. Initialization. The initial population is formed randomly, taking into account constraints on the values of input variables. The volume m of the initial population is selected based on the size of the SUT (namely, the number of feasible paths). It should be noted that the results of the initialization stage can be used to compare the method proposed in this work with the simplest random test data generation.
2. Population evaluation. Each of the chromosomes is evaluated by a fitness function. In the following sections, we define and explore a fitness function based on the statement coverage of SUT. The proposed fitness function consists of two terms, the first of

which is responsible for the coverage of the current path. Another term is responsible for the variety of paths in one population.

3. Selection. The best 20 percent of chromosomes are selected for the next generation directly; the remaining 80 percent of chromosomes are obtained as a result of crossover.
4. Crossover. Among 80 percent of the offspring obtained as a result of crossover, 50 percent is obtained by randomly crossing 20 percent of the best chromosomes of the previous generation with each other. The remaining 50 percent is obtained by randomly crossing all the chromosomes of the previous generation with each other.
5. Crossover occurs through the random choice of a constant $\beta_l \in [0, 1]$ for each $l = \overline{1, N}$ and subsequent blending where a single l-th offspring gene comes from a linear combination of l-th genes of the two random chromosomes from the parents' pool [40]:

$$var_l^{offspring} = \beta_l \cdot var_l^{mother} + (1 - \beta_l) \cdot var_l^{father},\ l = \overline{1, N}.$$

6. Mutation. With a given mutation probability 0.05 the l-th gene, $l = \overline{1, N}$, can change its value randomly within the domain D_l. The main goal of mutations is to obtain solutions that could not be obtained from the existing genes.
7. Forming the test cases as a pool of elite chromosomes. In each generation, chromosomes are chosen into the pool of elite chromosomes sequentially, in the order of their fitness, i.e., the ability to cover the most complex SUT paths. The next chromosome is added (or not) if it provides (or not) additional code coverage compared to the existing pool of elite chromosomes.

After all the stages have been carried out, it is assessed whether the test case has reached the desired fitness, or has come to a limit on the number of generations M.

2.3. Criteria for SUT Quality

For almost any realistic program system, an exhaustive set of test cases (the so-called test suite) contains an infinite number of test cases, so that this test suite can never be executed. Therefore, selection of the finite test suite from the infinite one is an important task. An adequate criterion to solve this task is to maximize the chance of detecting an error or a fault in the SUT while minimizing the cost of executing the test suite [41].

A simple test adequacy criterion could require, for example, that each statement in the SUT should be executed at least once when the code is tested. The methodologies that use such criteria are usually called coverage analyses, because certain paths of the source code are to be covered by the test cases.

There is a hierarchy of increasingly complex coverage criteria defining levels of coverage. At the top of the hierarchy is the multiple condition coverage level, requiring researchers to ensure that every permutation of values for the Boolean variables in every condition executes at least once. At the bottom of the hierarchy is function coverage, which requires only that every function be called once under the test suite.

Somewhere between these extreme levels are statement, branch, and path coverage. Path coverage is the most complete of all coverage criteria. Only if every possible path in the SUT is executed could the path coverage achieve 100%, which is usually impossible and impractical. Statement or branch coverage is an acceptable alternative.

Statement coverage is based on counting the statements of the SUT that will be performed when the code is run by a particular test case, compared to the total number of operations. The different code paths are determined by conditions and loops. We can say that if a condition or a loop is executed, then all statements within these paths will be performed. Therefore, the purpose of statement coverage is to execute as many statements of the SUT as possible in comparison with the total number of statements:

$$Statement\ Coverage = \frac{Number\ of\ executed\ statements}{Total\ number\ of\ statements} * 100\% \quad (3)$$

This coverage approach is also called C0-coverage and is relatively weak. A stronger coverage criterion is branch coverage, also called C1-coverage, defined as follows:

$$Branch\ Coverage = \frac{Number\ of\ executed\ branches}{Total\ number\ of\ branches} * 100\%$$

Branch coverage value will always be lower than the statement coverage. However, the difference between these criteria lies more in the ignoring of unloaded branches (if without else) in conditional statements, which is not particularly important. Therefore, without loss of generality, we will restrict ourselves to the statement coverage criterion (3). Thus, in this research, we consider statement coverage as the quality criterion for the generated test case population (2).

Therefore, the purpose of the statement coverage criterion is to execute as many statements of the program as possible in comparison with the total number of statements. Thus, the GA fitness function for a certain chromosome x_i has to be formulated to take into account the statement coverage requirement. That is, the test case corresponding to the most fitted chromosome traverses the most loaded (more complex) path containing as many SUT statements as possible.

A complementary approach to the formulation of the fitness function is the requirement for maximum coverage not only by one test case, but also by multiple test cases at once (preferably 100% coverage by the chromosome population). The latter leads to the possibility of obtaining the final solution in one GA run, and can be heuristically provided by the inclusion of special terms into the fitness function ensuring as much variety of individuals x_i and distance between them in the population (2) as possible.

2.4. Fitness Function for Maximum Statement Coverage and Population Diversity

In this section, we will formulate the fitness function of the genetic algorithm in such a way that maximizes statement coverage by both individual test cases and the whole test cases population.

The first step of white-box testing is to translate the source code into a Control Flow Graph (CFG). In the CFG, the statements are represented as nodes (boxes) and the con-trol flows between the statements are represented as edges. Denote the vector of nodes of the CFG by $\{s_1, s_2, \ldots, s_n\}$, where s_j is a separate node of CFG (one or more statements of the code). Note that the order of execution of separate nodes v_{I} may differ depending on various input data, since the program code contains conditional statements when computations are branched along several paths. Thus, different initial data of the program lead to traversing along different paths of the CFP, ensuring the execution of only quite specific (not all) statements of the program. Let us denote $g(x_i)$ a vector that is an indicator of the coverage of the graph nodes by a path initiated by a specific set of the test case x_i:

$$g(x_i) = (g_1(x_i), g_2(x_i), \ldots, g_{\mathrm{n}}(x_i))$$

where

$$g_j(x_i) = \begin{cases} 1, \ if\ path\ initiated\ by\ test\ case\ x_i\ traverses\ through\ the\ node\ s_j\ ; \\ 0, \ otherwise \end{cases}$$

Assigning weights to edges of CFG, we can take into account the fact that different paths of executing program code have different complexity. More weights are assigned to statements that are critical, being part of the more error-prone paths. Following the procedure proposed in [29] an initial credit is taken (for example, 100 or 10), if CFG is dense. i.e., large numbers of statements are there, than initial credit should be taken as 100 and if CFG is sparse (small code) then it can be taken as 10. At each node of CFG, the incoming credit (the sum of the weights of all the incoming edges) is divided and distributed to all the outgoing edges of the node. For the conditional and loops statements, we have used an 80–20 rule: 80 percent of weight of the incoming credit is given to loops and branches and

the remaining 20 percent of the incoming credit is given to the edges in sequential path. If we encounter a conditional statement with two or more conditions, fulfillment of each one leads to the execution of certain following statements, the weight of such a conditional statement is divided by the number of outgoing edges.

Let us denote by $(w_1, w_2, \ldots, w_n)$ a vector of weights assigned to all statements in accordance with the above-described method. Then the fitness function for the individual chromosome x_i can be formulated as follows

$$F_1(x_i) = \sum_{j=1}^{n} w_j g_j(x_i) \tag{4}$$

Indeed, the higher the sum of weighted statements covered by a path initiated with the test case x_i, the more fit is the chromosome x_i.

On the other hand, the use of Formula (4) for the fitness function will lead to a situation where the most adapted and capable of reproduction will always be individuals that lead to the most complex pieces of the code, to the detriment of the diversity of individuals in the population, since the population aspect in this formula not taken into account. As a result of using the GA with such a fitness function, we get the fittest individuals, however, if we evaluate the fitness of the resulting population as a whole, it will not provide maximum code coverage, since the chromosomes of the population will generate very similar paths.

To ensure a greater diversity of the population, it is necessary to introduce into the fitness function a term that gives preference to chromosomes that provide the greatest possible distance from each other—all paths that are generated by test cases of the population's chromosomes.

In order to calculate the j-th similarity coefficient $sim_j(x_{i_1}, x_{i_2})$ of two chromosomes x_{i_1} and x_{i_2} we compare if the node v_j of the CFG is covered or uncovered by both paths initiated by these two test cases

$$sim_j(x_{i_1}, x_{i_2}) = g_j(x_{i_1}) \oplus g_j(x_{i_2}),\ j = \overline{1,n}.$$

The more matching bits are there between the two paths, the greater is the similarity value between the chromosomes. The following formula takes into account weights of corresponding CFG nodes:

$$sim(x_{i_1}, x_{i_2}) = \sum_{j=1}^{n} w_j \cdot sim_j(x_{i_1}, x_{i_2})$$

The value of similarity between the chromosome x_i and the rest of the chromosomes in the population is calculated as

$$f_{sim}(x_i) = \frac{1}{(m-1)} \sum_{\substack{s=1 \\ s \neq i}}^{m} sim(x_s, x_i)$$

Now we can determine the maximum value of path similarity in the whole population

$$\overline{f_{sim}} = \max_{i=\overline{1,m}} f_{sim}(x_i)$$

Thus, we can formulate the term of fitness function responsible for the diversity of paths in a population. It is

$$F_2(x_i) = \overline{f_{sim}} - f_{sim}(x_i) \tag{5}$$

Thus, the fitness function for the chromosome x_i is calculated by the formula

$$F(x_i) = F_1(x_i) + k \cdot F_2(x_i) \tag{6}$$

where $F_1(x_i)$ and $F_2(x_i)$ are defined by Formulas (4) and (5). The first term $F_1(x_i)$ determines the complexity of the path initialized by the chromosome x_i, and the second term $F_2(x_i)$ determines the remoteness of this path from other paths in the population. The constant k determines relation between the two terms. The best value of k will be empirically determined in the next section.

3. Experiment Results

To investigate the GA's ability to work with the proposed fitness function (6), we used two examples of SUT, namely, SUT1 and SUT2, which both were specially designed for the research.

3.1. Experiment Results for SUT1

The first SUT1 was designed to test the data generation method. It contains six conditional statements and three cycles, thus allowing to define a sufficient number of different paths of the program code. Appendix A shows a CFG of this code. FOR 1 is the main cycle of the program and contains most of the statements and conditions, so many operations will be executed multiple times. Conditions IF 1, IF 2, and IF 3 are checked sequentially and require different test cases to fulfill. Condition IF 6 will only be achieved if both IF 4 and IF 5 are true and cycle WHILE 1 is completed. The code has different approaches to representing conditions, so the proposed method will generate data under different circumstances.

In the course of our research, we examined two versions of GA, using the Formula (6) to calculate the fitness function for the number $m = 100$ of individuals in the population. In the first version, we assigned $k = 0$ in Formula (6). Since for this version complete code coverage was never achieved in one population with a single run of GA, each application of it was used to find one best-fit chromosome. Furthermore, the statements covered by the graph path initiated by the test case found in the previous GA launch received zero weights, and the process of searching for the next best-fit chromosome continued in a similar way. The results of computational experiments obtained with this method are presented in Table 1. A total of six experiments with different number of generations and population size were carried out, in each we received different number of test cases that completely covered all the statements of the program code. However, these solutions were not obtained in an optimal way, since we had to run the GA four times or more consecutively (single path at a time), achieving with each new test case more and more coverage. Moreover, there may be test cases that do not increase code coverage at all (such test cases are italicized in Table 1). The statement coverage indicators of the graph nodes, colored green, correspond to the newly covered nodes with the current test case.

Table 2 shows the results of studying the GA work with a nonzero value $k = 10$ in the Formula (6) for the fitness function (i.e., with the presence of both terms), with different M and m. In the six experiments, it only possible to achieve 100% coverage with sufficient number of $m = M = 20$. Therefore, the improvement of the form of fitness function formulation and careful selection of its parameters remains topical.

Table 1. Results for the multiple launch GA for SUT1.

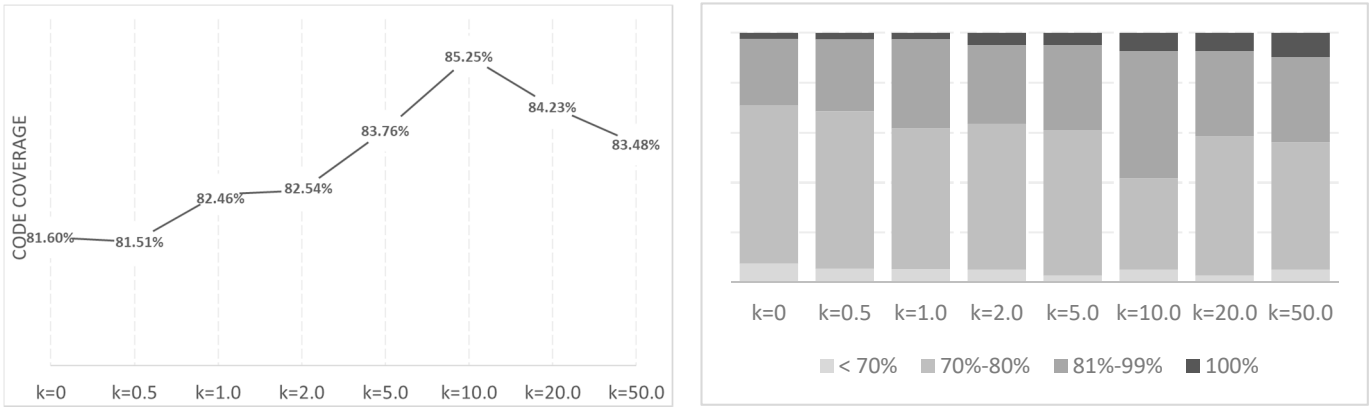
M Generations	m Chromosomes	Test Suite, x	Coverage, %	Statement Coverage Indicator $g(x)$
10	5	(6, 91, 20)	27%	11111111111110000000000000000000000000000000000000000001
		(86, 41, 98)	58%	11111111111000000000000000001111111111111111100000000001
		(29, 99, 83)	*58%*	*11111111111111000000000000000000000000000000000000000001*
		(51, 48, 3)	*58%*	*11111111111111000000000000000000000000000000000000000001*
		(60, 9, 60)	72%	11111111111000111111110000000000000000000000000000000001
		(2, 42, 80)	100%	11111111111000000000001111111111111000000000111111111 1
	10	(61, 80, 70)	52%	11111111111000000000000000011111111111111111000000000 1
		(60, 15, 55)	67%	11111111111000111111110000000000000000000000000000000001
		(57, 16, 74)	72%	11111111111111000000000000000000000000000000000000000001
		(3, 54, 85)	100%	11111111111000000000001111111111111000000000111111111 1
	20	(72, 40, 96)	52%	11111111111000000000000000011111111111111111000000000 1
		(3, 72, 69)	80%	11111111111000000000001111111111111000000000111111111 1
		(38, 83, 81)	85%	11111111111111000000000000000000000000000000000000000001
		(60, 49, 35)	100%	11111111111000111111110000000000000000000000000000000001
20	5	(67, 35, 84)	52%	11111111111000000000000000011111111111111111000000000 1
		(98, 92, 93)	58%	11111111111111000000000000000000000000000000000000000001
		(93, 13, 78)	*58%*	*11111111111000000000000000000000000000000000000000000000*
		(60, 30, 15)	72%	11111111111000111111110000000000000000000000000000000001
		(3, 30, 72)	100%	11111111111000000000001111111111111000000000111111111 1
	10	(69, 85, 72)	52%	11111111111000000000000000011111111111111111000000000 1
		(60, 41, 8)	67%	11111111111000111111110000000000000000000000000000000001
		(2, 69, 64)	94%	11111111111000000000001111111111111000000000111111111 1
		(7, 90, 41)	100%	11111111111111000000000000000000000000000000000000000001
	20	(83, 40, 98)	52%	11111111111000000000000000011111111111111111000000000 1
		(60, 7, 44)	67%	11111111111000111111110000000000000000000000000000000001
		(54, 6, 65)	72%	11111111111111000000000000000000000000000000000000000001
		(4, 11, 94)	100%	11111111111000000000001111111111111000000000111111111 1

Note: The statement coverage indicators of the graph nodes, colored green, correspond to the newly covered nodes with the current test case. Moreover, there may be test cases that do not increase code coverage at all (such test cases are italicized in Table 1).

Table 2. Results for the single launch GA for SUT1.

Generations	Chromosomes	Test Cases x	Coverage, %	Statement Coverage Indicator $g(x)$
10	5	(60, 41, 43)	72%	11111111111000111111110000000000000000000000000000000001
		(70, 81, 86)		11111111111000000000000000011111111111111111110000000001
		(28, 27, 67)		11111111111111000000000000000000000000000000000000000001
	10	(30, 5, 8)	72%	11111111111111000000000000000000000000000000000000000001
		(83, 39, 84)		11111111111000000000000000011111111111111111110000000001
		(60, 51, 67)		11111111111000111111110000000000000000000000000000000001
	20	(61, 88, 59)	85%	11111111111000000000000000000000000000000000000000000000
		(52, 81, 42)		11111111111111000000000000000000000000000000000000000001
		(62, 89, 80)		11111111111000000000000000011111111111111111110000000001
		(2, 49, 74)		11111111111000000000001111111111111100000000011111111 1
20	5	(33, 17, 94)	72%	11111111111111000000000000000000000000000000000000000001
		(67, 8, 95)		11111111111000000000000000011111111111111111110000000001
		(60, 33, 105)		11111111111000111111110000000000000000000000000000000001
	10	(65, 49, 90)	85%	11111111111000000000000000011111111111111111110000000001
		(3, 1, 57)		11111111111000000000001111111111111100000000011111111 1
		(37, 2, 57)		11111111111111000000000000000000000000000000000000000001
	20	(37, 17, 87)	100%	11111111111111000000000000000000000000000000000000000001
		(0, 70, 86)		11111111111000000000001111111111111100000000011111111 1
		(78, 0, 90)		11111111111000000000000000011111111111111111110000000001
		(60, 41, 71)		11111111111000111111110000000000000000000000000000000001

To study the effect of the constant k in the relation (6) to the results of GA runs, the values of the population size and the number of generations were specially selected, $m = M = 35$ for which complete 100% statement coverage of the SUT1 presented in Appendix A was achieved quite rarely. Figure 1a shows the average value of the achieved statement coverage for various values of k from 0 to 50, calculated from 40 GA independent runs, each time carried out with new random initial populations $m = 35$ and maximum $M = 35$ generations.



(**a**) Average value (**b**) Distribution

Figure 1. Achieved statement coverage of final population depending on k ($m = M = 35$) for SUT1.

Thus, we see a non-linear dependence of the average coverage on the value of k, which determines the ratio between the terms $F1$ and $F2$ in the expression for the fitness function. First, as k grows, the statement coverage increases, reaching its maximum for $k = 10$. After that, the value of the fitness function begins to decline, as excessive attention begins to be paid to $F2$, which is responsible for the diversity of paths in the population at the expense of the coverage for each path.

It is also interesting to consider the distribution of different degrees of code coverage, as shown in Figure 1b. As you can see, for a larger number of k, the largest share of coverage falls into the group from 70% to 80%. Only for $k = 10$, the largest share falls into the 81–99% group. Further, it can be noted that for $k = 20$ and $k = 50$, in general, the distribution worsens compared to $k = 10$ (although the share of variants with 100% coverage is higher, the share with low coverage also increases).

Therefore, there is a certain value of k (in our case, it is $k = 10$), at which a balance is achieved between the terms F_1 and F_2 in the sum (6). If F_1 outweighs, then the resulting population is not diverse enough to cover the entire code. If F_2 outweighs, then excessive attention is paid to the diversity of the population to the detriment of the quality of coverage of each individual.

3.2. Experiment Results for SUT2

As a SUT2, a specially generated program with a large number of operations and conditions was used. Appendix B shows the CFG of only a part of the program. The main functionality is found in the functions that are not presented here due to space limitations. In addition to a larger number of operations and functions in SUT2, a hierarchy of conditions is introduced to test the performance of the proposed method for automatic generation of test data.

The dependence of the code coverage on k for different number of chromosomes in the population for SUT2 is shown in Figure 2. To represent a different percent (not always 100%) of coverage, it was decided to use only 75 generations. The trend of the code coverage versus k for SUT2 is very similar to that for SUT1 shown in Figure 1a. The best coverage value is achieved at $k = 10$, even though the tested codes SUT1 and SUT2

differ significantly in the number and complexity of statements. Thus, the value $k = 10$ can presumably be recommended as an initial one for other SUTs as well.

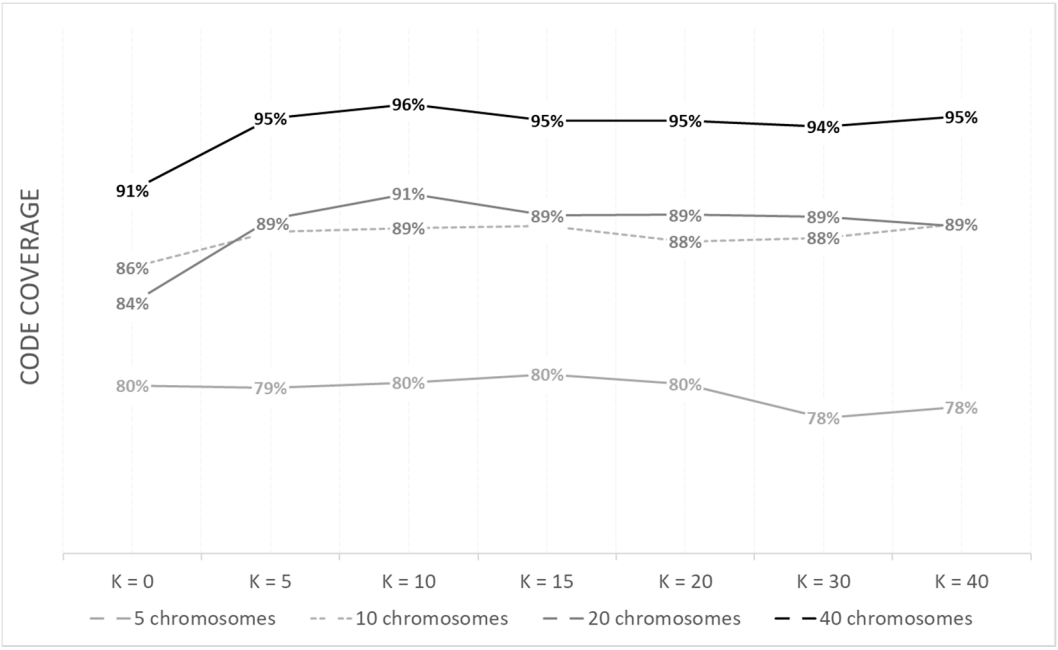


Figure 2. Achieved statement coverage of final population depending on k for SUT2.

For the SUT2, research of the dependence of the average value of statement coverage on the number of generations varying from 1 to 700 were carried out. The average coverage value was calculated from 10 random GA launches with a fitness function calculated by Formula (6) with $k = 10$. The results of calculations are presented for different population sizes (m = 5, 10, 15, 20, and 40), and it is assumed that it does not change from generation to generation. Figure 3 clearly shows how much the convergence rate of the algorithm differs when using a different number of chromosomes in the population.

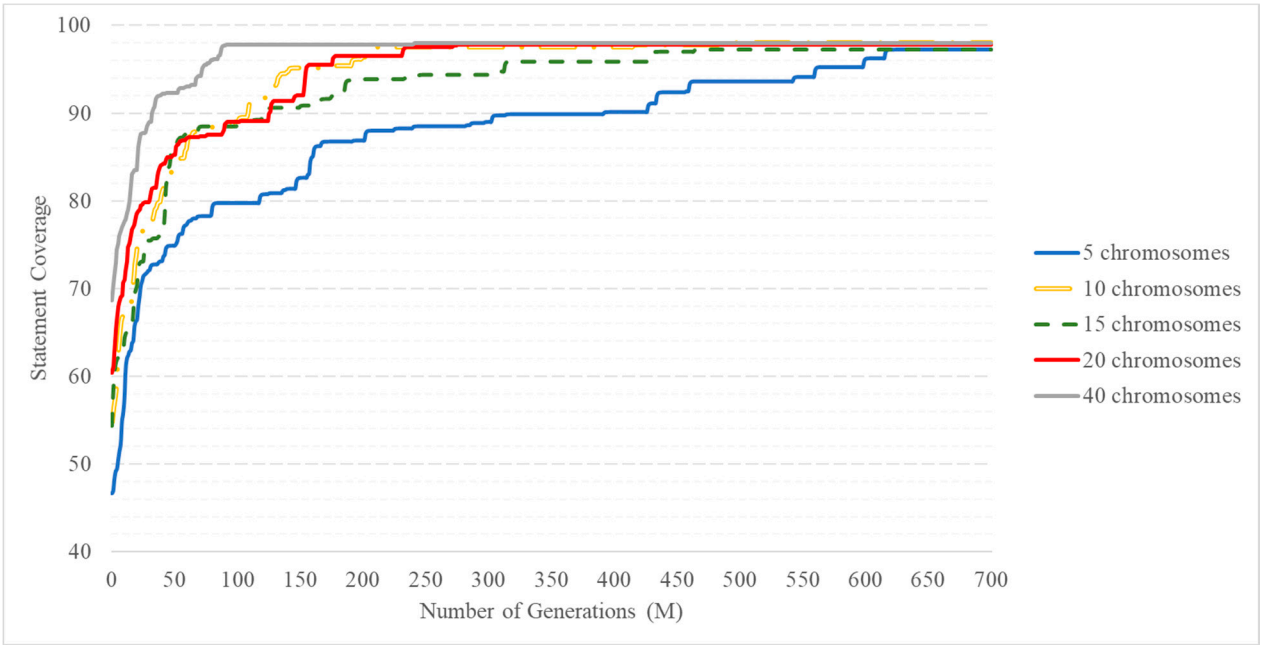


Figure 3. Comparison of the algorithm's convergence speed with different population size.

First of all, it should be noted that in all cases the proposed algorithm stably ensures the accumulation of a pool of elite chromosomes, providing almost complete code coverage (at least 98% on average). The difference between the results obtained for different population sizes mainly consists in different rates of convergence to complete code coverage when $m = 5$, the lowest speed is achieved and the final coverage value is the lowest of the considered variants. An algorithm with 5 chromosomes in the population accumulates a pool of elite chromosomes for an extremely long time (700 generations), which provides almost complete coverage in about 700 generations.

An algorithm of test data generation with a population size of 10, 15, and 20 chromosomes shows somewhat better results. It reaches the maximum coverage value at ~270–450 generations, i.e., the convergence rate is much higher. Note that a value of 20 is an upper bound for the number of different paths providing complete coverage for the SUT2.

As expected, an increase in the number of chromosomes in a population significantly increases the convergence rate of the algorithm, i.e., the rate of increase in the degree of code coverage in the pool of elite chromosomes. This is achieved by providing more sample diversity. The algorithm achieved the best values of coverage and convergence rate for 40 chromosomes in the population. The maximum value of statement coverage is reached within 70–80 generations.

For the proposed code, as mentioned above, the number of generated variants is approximately 15–20. That is, for better generation of test data, it is necessary to have at least 2 times more chromosomes than the required number of test cases required for complete code coverage. A further increase in the number of chromosomes in the population will lead to an increase in the convergence rate, but negatively affect the overall speed of the method, because it leads to an exponential increase in the amount of computation.

4. Discussion

A traditional genetic algorithm, which is used to solve various optimization problems, turns out to be promising for solving the problem of the automated search for test cases to cover a certain path of the analyzed software under testing. If the problem arises of finding the minimum set of test cases covering the entire SUT, then an ordinary GA in this case solves this problem in a non-optimal way, spending a large amount of resources for real software of large dimensions.

The method proposed in the present paper permits the generation of test data with maximum code coverage based on the special formulation of fitness function consisting of two terms: the complexity of the source code path for a particular test case and the achievement of the greatest diversity of paths covered with different test cases in one population. This allows obtaining test cases for many different code paths within a single launch of the genetic algorithm. Obtaining similar optimal relation between the two terms in the fitness function (6) for two very different SUT1 and SUT2 is also very promising and makes it possible to suggest an acceptable compromise relation for an arbitrary SUT.

In the future, it seems promising to increase the convergence rate by considering the option of dynamically changing the ratio between both terms of the fitness function in the process of approaching an optimal solution. Integration of GAs and other evolutionary approaches with use of the new formulation of the fitness function also seems promising for the problem of automatic generation of test data.

Author Contributions: Conceptualization, T.A.; methodology, T.A.; software, K.S.; validation, T.A., K.S.; formal analysis, T.A.; investigation, T.A., K.S.; resources, K.S.; data curation, K.S.; writing—original draft preparation, T.A., K.S.; writing—review and editing, T.A.; visualization, K.S.; supervision, T.A.; project administration, T.A.; funding acquisition, T.A., K.S. All authors have read and agreed to the published version of the manuscript.

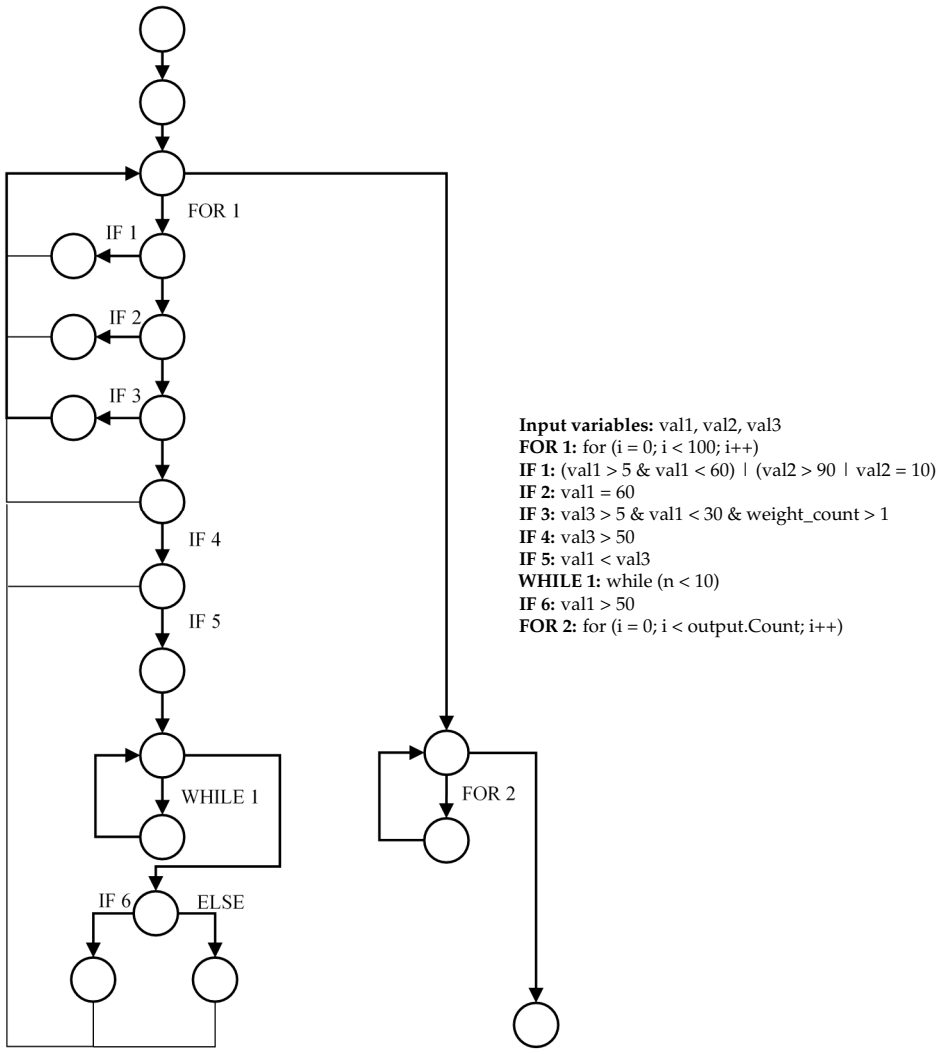
Funding: The research is supported by Ministry of Science and Higher Education of Russian Federation (project No. FSUN-2020-0009). The research is supported RFBR, project № 19-37-90156.

Institutional Review Board Statement: Not applicable.

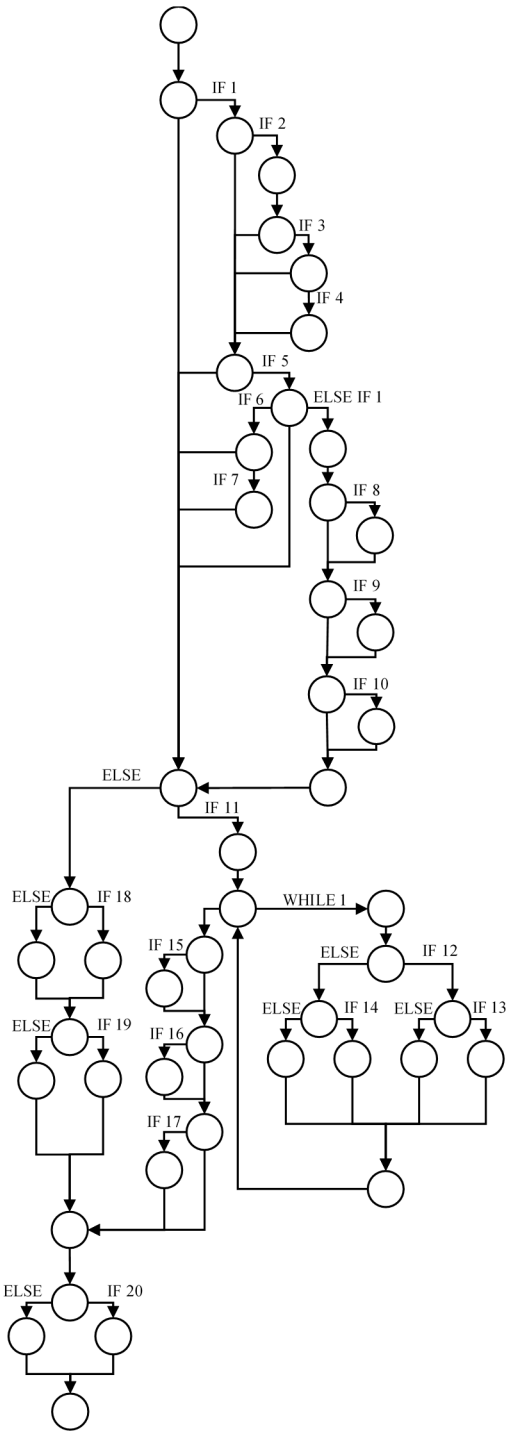
Informed Consent Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

Appendix A. Control Flow Graph for the SUT1



Appendix B. Control Flow Graph for the SUT2



Input variables: executionNumber, val1, val2, val3, val4, val5
IF 1: IsPalindromeNumber(val1) || IsPalindromeNumber(val2) || IsPalindromeNumber(val3) || IsPalindromeNumber(val4) || IsPalindromeNumber(val5)
IF 2: val1 == Largest4 Number (val2, val3, val4, val5)
IF 3: val2 <= val4/2
IF 4: IsOddNumber(val2) && IsOddNumber(val4)
IF 5: IsPrimeNumber(val1) && IsPrimeNumber(val3) && IsPrimeNumber(val5)
IF 6: NumberMoreThan(val1, val3) && NumberMoreThan(val1, val5))
IF 7: val1 < val2 && val1 < val4
ELSE IF 1: NumberLessThan(val3, val5)
IF 8: val1 > 0
IF 9: val3 > 0
IF 10: val5 > 0)
IF 11: val2 > val3 && val1 < val5
WHILE 1: checkDouble <= max)
IF 12: val1 < val3)
IF 13: checkDouble % 2 == 0)
IF 14: IsAmicableNumbers(val1, val3)
IF 15: val3 < val2
IF 16: val1 < val5
IF 17: val3 < val2 && val1 < val5
IF 18: Power(val3, 2) < val2
IF 19: Power(val1, 2) < val5
IF 20: executionNumber < 100)

References

1. Spillner, A.; Linz, T.; Schaefer, H. *Software Testing Foundations: A Study Guide for the Certified Tester Exam*, 4th ed.; Rocky Nook: San Rafael, CA, USA, 2011.
2. Korel, B. Automated software test data generation. *IEEE Trans. Softw. Eng.* **1990**, *16*, 870–879. [CrossRef]
3. Bird, D.; Munoz, C.C. Automatic generation of random self-checking test cases. *IBM Syst. J.* **1983**, *22*, 229–245. [CrossRef]
4. Xuan, J.; Jiang, H.; Ren, Z.; Hu, Y.; Luo, Z. A random walk based algorithm for structural test case generation. In Proceedings of the 2nd International Conference on Software Engineering and Data Mining, Chengdu, China, 23–25 June 2010; pp. 583–588.
5. Sui, J.; Gong, Y.; Jin, D.; Wang, Y. Statistical testing data generation for UAS. *IOP Conf. Ser. Mater. Sci. Eng.* **2020**, *715*, 7. [CrossRef]

6. Bauer, J.; Finger, A. Test plan generation using formal grammars. In Proceedings of the 4th International Conference on Software Engineering, Munich, Germany, 17–19 September 1979; pp. 425–432.
7. Cunning, S.J.; Rozenblit, J. Test scenario generation from a structured requirements specification. In Proceedings of the ECBS'99, IEEE Conference and Workshop on Engineering of Computer-Based Systems, Nashville, TN, USA, 7–12 March 1999; pp. 166–172.
8. 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: Washington, DC, USA, 2007; pp. 47–52.
9. Sabharwal, S.; Sibal, R.; Sharma, C. Applying Genetic Algorithm for Prioritization of Test Case Scenarios Derived from UML Diagrams. *IJCSI Int. J. Comput. Sci. Issues* **2011**, *8*, 433–444.
10. Doungsa-ard, C.; Dahal, K.; Hossain, A.; Suwannasart, T. GA-based Automatic Test Data Generation for UML State Diagrams with Parallel Paths. In *Advanced Design and Manufacture to Gain a Competitive Edge: New Manufacturing Techniques and Their Role in Improving Enterprise Performance*; Springer: London, UK, 2008; pp. 147–156.
11. Grochtmann, M.; Grimm, K. Classification trees for partition testing. *Softw. Test. Verif. Reliab.* **1993**, *3*, 63–82. [CrossRef]
12. Chen, T.Y.; Poon, P.L.; Tse, T.H. An integrated classification-tree methodology for test case generation. *Int. J. Softw. Eng. Knowl. Eng.* **2000**, *10*, 647–679. [CrossRef]
13. Cain, A.; Chen, T.Y.; Grant, D.; Poon, P.L.; Tang, S.F.; Tse, T.H. An Automatic Test Data Generation System Based on the Integrated Classification-Tree Methodology. In *Software Engineering Research and Applications, Lecture Notes in Computer Science*; Springer: Berlin/Heidelberg, Germany, 2004; Volume 3026.
14. Bicevskis, J.; Borzovs, J.; Straujums, U.; Zarins, A.; Miller, E. SMOTL-A system to construct samples for data processing pro- gram debugging. *IEEE Trans. Softw. Eng.* **1979**, *SE-5*, 60–66. [CrossRef]
15. Boyer, R.; Elspas, B.; Levitt, K. SELECT-A formal system for testing and debugging programs by symbolic execution. *Sigplan Not.* **1975**, *10*, 234–245. [CrossRef]
16. Clarke, L. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.* **1976**, *SE-2*, 215–222. [CrossRef]
17. Howden, W. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Trans. Softw. Eng.* **1977**, *SE-4*, 266–278. [CrossRef]
18. Ramamoorthy, C.V.; Ho, S.F.; Chen, W.T. On the automated generation of program test data. *IEEE Trans. Softw. Eng.* **1976**, *SE-2*, 293–300. [CrossRef]
19. Richard, A.D.; Jefferson, A.O. Constraint-Based Automatic Test Data Generation. *IEEE Trans. Softw. Eng.* **1991**, *17*, 900–910.
20. Meudec, C. ATGen: Automatic Test Data Generation using Constraint Logic Programming and Symbolic Execution. *Softw. Test. Verif. Reliab.* **2001**, *11*, 81–96. [CrossRef]
21. Gerlich, R. Automatic Test Data Generation and Model Checking with CHR. *arXiv* **2014**, arXiv:1406.2122.
22. Ferguson, R.; Korel, B. The Chaining Approach for Software Test Data Generation. *ACM Trans. Softw. Eng. Methodol.* **1996**, *5*, 63–86. [CrossRef]
23. Girgis, M.R. Automatic Test Data Generation for Data Flow Testing Using a Genetic Algorithm. *J. Univers. Comput. Sci.* **2005**, *11*, 898–915.
24. Khamis, A.; Bahgat, R.; Abdelaziz, R. Automatic test data generation using data flow information. *Dogus Univ. J.* **2011**, *2*, 140–153. [CrossRef]
25. Liu, Z.; Chen, Z.; Fang, C.; Shi, Q. Hybrid Test Data Generation. State Key Laboratory for Novel Software Technology. In Proceedings of the ICSE Companion 2014 Companion Proceedings of the 36th International Conference on Software Engineering, Hyderabad, India, 31 May–7 June 2014; pp. 630–631.
26. Harman, M.; McMinn, P. A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search. *IEEE Trans. Softw. Eng.* **2010**, *36*, 226–247. [CrossRef]
27. Maragathavalli, P.; Anusha, M.; Geethamalini, P.; Priyadharsini, S. Automatic Test-Data Generation for Modified Condition. Decision Coverage Using Genetic Algorithm. *Int. J. Eng. Sci. Technol.* **2011**, *3*, 1311–1318.
28. Sharma, A.; Patani, R.; Aggarwal, A. Software Testing Using Genetic Algorithms. *Int. J. Comput. Sci. Eng. Surv.* **2016**, *7*, 21–33. [CrossRef]
29. Praveen, R.S.; Tai-hoon, K. Application of Genetic Algorithm in Software Testing. *Int. J. Softw. Eng. Appl.* **2009**, *3*, 87–96.
30. Berndt, D.J.; Watkins, A. Investigating the Performance of Genetic Algorithm-Based Software Test Case Generation. In Proceedings of the Eighth IEEE International Symposium on High Assurance Systems Engineering (HASE'04), Tampa, FL, USA, 25–26 March 2004; pp. 261–262.
31. Kumar, M.; Chaudhary, J. Reviewing Automatic Test Data Generation. *Int. J. Eng. Sci. Comput.* **2017**, *7*, 11432–11435.
32. Serdyukov, K.; Avdeenko, T. Researching of methods for assessing the complexity of program code when generating input test data. *CEUR Workshop Proc.* **2020**, *2667*, 299–304.
33. Bueno, P.M.; Wong, W.E.; Jino, M. Automatic test data generation using particle systems. In Proceedings of the SAC '08: Proceedings of the 2008 ACM symposium on Applied computing, Fortaleza, Brazil, 16–20 March 2008; pp. 809–814.
34. Dixit, S.; Tomar, P. Applying Computational Intelligence in Software Testing. *J. Artif. Intell. Res. Adv.* **2015**, *2*, 5.
35. Li, K.; Zhang, Z.; Kou, J. Breeding Software Test Data with Genetic-Particle Swarm Mixed Algorithm. *J. Comput.* **2010**, *5*, 258–265. [CrossRef]

36. Ding, R.; Feng, X.; Li, S.; Dong, H. Automatic generation of software test data based on hybrid particle swarm genetic algorithm. In Proceedings of the 2012 IEEE Symposium on Electrical & Electronics Engineering (EEESYM), Kuala Lumpur, Malaysia, 24–27 June 2012; pp. 670–673. [CrossRef]
37. Khan, S.A.; Nadeem, A. Automated Test Data Generation for Coupling Based Integration Testing of Object Oriented Programs Using Particle Swarm Optimization (PSO). In *Genetic and Evolutionary Computing. Advances in Intelligent Systems and Computing*; Pan, J.S., Krömer, P., Snášel, V., Eds.; Springer: Cham, Switzerland, 2014; Volume 238.
38. Zhu, E.; Yao, C.; Ma, Z.; Liu, F. Study of an Improved Genetic Algorithm for Multiple Paths Automatic Software Test Case Generation. In *Advances in Swarm Intelligence*; Springer: Cham, Switzerland, 2017; pp. 402–408.
39. Mirhosseini, S.M.; Haghighi, H. A Search-Based Test Data Generation Method for Concurrent Programs. *Int. J. Comput. Intell. Syst.* **2020**, *13*, 1161–1175. [CrossRef]
40. Haupt, R.L.; Haupt, S.E. *Practical Genetic Algorithms*, 2nd ed.; John Wiley & Sons, Inc.: Hoboken, NJ, USA, 1998.
41. Tretmans, G.J.; Brinksma, H. TorX: Automated Model-Based Testing. In Proceedings of the First European Conference on Model-Driven Software Engineering, Nuremberg, Germany, 11–12 December 2003; pp. 31–43.