Automatic Test Case Generation using Genetic Algorithm with Antirandom Population

Kulvinder SINGH¹, Iqbal KAUR² and Rakesh KUMAR³

¹University Institute of Engg. & Technology, Kurukshetra University Kurukshetra, Haryana, India
²Department of Mathematics, Govt. P. G. College, Sector-14, Karnal Haryana, India
³Department of Computer Science and Applications, Kurukshetra University Kurukshetra, Haryana, India

Abstract: Reliability is a most important quality attribute of software which is to be achieved thorough software testing, the most effort consuming activity, success of which depends upon the adequacy of the data and all the criteria discussed in the literature are incomplete. There is the need of automatic test case generation so that the cost of testing can be reduced. In this research paper, Genetic Algorithm is used to automatically generate the test cases using the antirandom testing technique and the results are verified using n-versions of the module. Mutation adequacy is used to verify the completeness. The results of the Genetic Algorithm with random population and hamming code are compared and improvements have been observed over random and hamming code testing.

Keywords: Anti-random testing, Genetic algorithm, N-Version Programming, Hamming Code, Random testing

1. INTRODUCTION

Several approaches had been proposed for test case generation, mainly random, path-oriented, goal-oriented, and some intelligent approaches. Random techniques find out test cases based on assumptions regarding the fault distribution. Path-oriented techniques normally used control flow information to identify a set of paths to be covered and generated the appropriate test cases for these paths. These techniques are further classified as static and dynamic. Static techniques frequently based on symbolic execution, whereas dynamic techniques obtained the necessary data by executing the program under test. Goal-oriented techniques generate test cases covering a selected goal such as a statement, branch, or path taken. Intelligent techniques or automated test case generation rely on complex computations to generate test cases.

The objective of the testing is to reveal the errors in the software and testing is the process of executing a program with the intent of finding errors [1]. If errors are not identified by the testing its meaning is that test set used is not adequate enough. Activities are carried out to evaluate the attributes of a program and verifying that it meets its required results [2]. Effectively detecting the failures using the limited resource is a challenging task. Identifying faults in software is easier said than done because software is not continuous, so testing boundary values as suggested in Boundary Value Analysis or selecting test cases using criteria such as path coverage are not ample to assure correctness and exhaustive testing is infeasible. Thing are further complicated by the dynamic nature of programs. If a failure occurs during preliminary testing and the code is changed, then behavior of software on pre-error test cases that it passed before can no longer is guaranteed. So testing should be restarted.

Software is being used now in mission critical situations where failure is simply intolerable. From the point of view of a software development organization also, delivering products with defects results in loss of goodwill. Thus, the only alternative is to do it right the first time, before delivering the product to the customer [4]. In this paper, a technique of automatic test case generation using Genetic Algorithm (GA) with not taking the initial population as randomly but with the specific criteria known as gray code or hamming code has been purposed and the result were compared with random testing. Section 2 deals with importance of testing and test adequacy criteria including mutation adequacy. Section 3 gives a brief overview of n-version testing and section 4 cover the issues in automatic test cases generation using random and anti random testing. In Section 5, GA was discussed and section 6 covers the proposed technique followed by section 7 in which results were analyzed.

* Corresponding author: kshanda@rediffmail.com
2. AUTOMATIC TESTING AND ITS IMPORTANCE

The software system should be reliable, available, safe and secure. To achieve these objectives a number of techniques are being used such as fault avoidance, fault tolerance, fault removal, and fault evasion etc. Testing is an integral part for fault removal and is usually performed for the following purposes: (a) Quality assurance, (b) For Verification & Validation (V&V): Testing is used as a tool in the V&V process. Testers can make claims based on interpretations of the testing results whether the product works under certain situations or not. Tests with the purpose of validating the product works are named clean tests. The drawbacks are that it can only validate that the software works for the specified test cases. A finite number of tests can not validate that the software works for all situations. On the contrary, only one failed test is sufficient enough to show that the software does not work. Dirty tests refer to the tests aiming at breaking the software and software must have sufficient exception handling capabilities to survive a significant level of dirty tests. (c) For reliability estimation [5]: Software reliability has important relations with many aspects of software, including the structure, and the amount of testing it has been subjected to. Based on an operational profile (an estimate of the relative frequency of use of various inputs to the program [5]), testing can serve as a statistical sampling method to gain failure data for reliability estimation. So testing is an important activity in the software development but it’s a time consuming process and sometimes consume more than 50% of the total efforts required for development. Cost of software testing can be significantly reduced if the testing process can be automated.

2.1. Mutation Testing and Effectiveness of Test Cases

The quality of the test cases, effectiveness and adequacy, depends upon their capability to detect the faults. Some adequacy criteria discussed in the literature are statement coverage, branch coverage, path coverage, loop coverage etc but studies reveal that no criterion is effective enough to identify all the bugs except exhaustive testing which is theoretically and practically not possible. Mutation testing has been established as a powerful approach to evaluate test cases and for comparing different testing strategies. Empirical studies show that the generated mutants provide a good indication of the fault detection ability of a test suite [6]. Mutation testing is an approach to verify the effectiveness of the test designed cases and has been proved successful with some limitations.

2.2. Mutation Adequacy

The mutation method is a fault-based testing strategy that measures the adequacy of testing by examining whether the test set used in testing can reveal certain types of faults. The core of a mutation testing is a set of operators that modifies the source code to inject a fault to produce mutant i.e. modified program. A mutant is said to be killed relative to a test data set, if at least one test case generates different results between the mutant and the implementation. Otherwise, the mutant is live. If no test case can kill a mutant, then it is either equivalent of the original implementation or a new test case needs to be generated to kill the live mutant, a method of enhancing a test data set. The adequacy of a test data set is measured by a mutation score (MS), which is the percentage of non-equivalent mutants killed by the test data. The mutation score for a set of test cases is:

\[ \text{Mutation Score} = 100 \times \frac{D}{(N - E)}; \]

Where \( D \) = Dead mutants, \( N \) = Number of mutants, and \( E \) = Number of equivalent mutants.

A set of test cases is mutation adequate if its mutation score is 100%.

3. N-VERSION TESTING

In N-version programming, the software module is implemented in a number of different versions by different teams, using common specification. These versions are executed in parallel. There outputs are compared using a voting system and inconsistent outputs are rejected. At least three versions of the module should be available. The assumption is that it is unlikely that different teams will make the same design or programming errors. [7, 8] describes this approach as fault avoidance. In Back-to-back testing, using lessons learned from N-version programming, [9] and [10] have suggested that that N version of the software be developed even when only a single version will be used. Test cases designed using other testing techniques are provided as test input to each version and their outputs can be compared by automatic tools. In case of the differences in the output, each of the versions is analyzed to identify the fault. This method depends on the basis that all the versions have been developed independently so if any version fails that will fail independently. In this paper total three different version of the same sorting programs were prepared independently from the same specification of the software and then subjected to thousand of test cases. In this research paper we use two types of test cases: one is totally random
numbers, and second one is using the concept of the Genetic Algorithm. Both type of the test case are given to the N-version software after placing the mutant in any one version of the software.

4. AUTOMATIC TEST CASE GENERATION

4.1. Random Testing

Random Testing (RT) randomly selects test cases from the input domain [1, 11]. The advantages of RT are (a) low cost, (b) generation of test cases automatically, (c) generation of test cases in the absence of the software specification and source code, and (d) randomness into the testing process. Such randomness’s can best reflect the chaos of system operational environment; as a result, RT can detect certain failures unable to be revealed by deterministic approaches. All these advantages make RT irreplaceable in industry for revealing software failures [12, 13, 14, 15, 16, 17, 18, 19, 20, 21]. Although there are limitations also such as (a) a large number of event sequences that are not legal & not executable are generated wasting valuable resources and (b) the test designer has no control over choice of event sequences. Random testing selects arbitrarily test data from the input domain & then these test data are applied to the program under test. The automatic production of random test data, drawn from uniform distribution, should be the default method by which other systems should be judged, [22]. The random generation of tests identifies members of the sub-domains arbitrarily, with a homogeneous probability, which is related to the cardinality of the sub-domains. Under these circumstances, the chances of testing a function, whose sub-domain has a low cardinality with regard to the domain as a whole, is much reduced. A random number generator generates the test data with no use of feedback from previous tests. The tests are passed to the procedure under test, in the hope that all branches will be traversed [23].

4.2. Antirandom Testing

4.2.1. Antirandom testing

In the antirandom software testing, in the previous researches, the test cases are generated by using the hamming or Cartesian distance techniques. However, in this research paper researcher used the Gray code for generating the test cases. A antirandom number generator generates the test data with use of feedback from previous tests. The tests are passed to the procedure under test, in the hope that it detects the maximum errors.

**Algorithm for Antirandom Test case Generation**

Consider a \( n \) bit binary number \( Bin \) [n-1:0] with \( I \) representing the index of the binary number. Let \( Gray \) [n-1:0] be the equivalent Gray code.

1. For \( I=n-1 \),

   \[ Gray \ [n-1] = Bin \ [n-1] \] [the most significant bit (MSB) of the Gray code is same as the MSB of original binary number.]

2. For \( I=n-2 \) to 0,

   \[ Gray[i]= Bin[i+1] \oplus Bin[i] \]

\([ I^th \] bit of the Gray code is the exclusive-OR (XOR) of \( I^th \) of the bit of the binary number and \((I+1)^{th}\) of the bit of the binary number.]

In fact, a “Gray code” almost always refers to a binary-reflected Gray code. On the other hand, mathematicians have discovered other kinds of Gray codes. For example, n-ary Gray code, Balanced Gray code, Beckett-Gray code etc.

n-ary Gray code is the specialized type of Gray codes other than the binary-reflected Gray code, also known as a non-Boolean Gray code. As the name entails this Gray code uses non-Boolean values in its encodings. The \( (n, k) \)-Gray code is the n-ary Gray code with \( k \) digits.\[12\] For example, the sequence of elements in the \((3, 2)\)-Gray code is: \{00, 01, 02, 12, 10, 11, 21, 22, 20\}. Moreover, \((n, k)\)-Gray code may be constructed recursively, as the binary-reflected Gray code, or may be constructed iteratively.

5. GENETIC ALGORITHM

GA is a search technique used to find solutions to optimization and search problems. GAs represents a class of adaptive search techniques & procedures based on the processes of natural genetics & Darwin’s principal of the survival of the fittest. There is a randomized exchange of structured information among a population of artificial chromosomes. When GAs are used to solve optimizations problems, good results are obtained surprisingly quickly. A problem is defined as maximization of a function of the kind \( f(x_1, x_2, ... x_m) \) where \( (x_1, x_2, ..., x_m) \) are variables which have to be adjusted towards a global optimum. Three basic operators responsible for GA are (a) selection, (b) crossover & (c) mutation. Crossover performs recombination of different solutions to ensure that the genetic information of a child life is made up of the genes from each parent. GAs may be differentiated from more conventional techniques as (a) in GA a representation for the sample population must be derived, (b) GAs manipulates directly the encoded
representation of variables, rather than manipulation of the variables themselves, (c) GAs use stochastic rather than deterministic operators, (d) GAs search blindly by sampling & ignoring all information except the outcome of the sample, (e) GAs search from a population of points rather than from a single point, thus reducing the probability of being stuck at a local optimum, which make them suitable for parallel processing. In the context of S/W testing, the basic idea is to search the domain for input variables which satisfy the goal of testing. With the above defined, GA is defined as follows:

\[
\text{Find } b \text{ such that } \varphi(b) = \max(\varphi(h))
\]

\[
\text{return}(b)
\]

\text{end}\_proc

6. PROPOSED FRAMEWORK USING GA, MUTATION TESTING AND BACK-TO-BACK TESTING

Using GA is one proposed way to test application [33]. This method generates test cases based on the theory that good test coverage can be attained by simulating a novice user who would follow a more random path while an expert user of a system will follow a predictable path through an application ignoring many possible system states that would never be achieved. So it’s more desirable to create test suites that simulate novice usage because they will test more. The difficulty lies in generating test suites that simulate ‘novice’ system usage. Novice paths through the system are not random paths. First, a novice user will learn over time and generally won’t make the same mistakes repeatedly and secondly, a novice user is following a plan and probably has some domain or system knowledge. The proposed framework for automatic test case generation using GA, mutation testing and back to back testing is as under:

Write the module \( V \) // i.e. the module to be tested
Generate N versions of \( V \) i.e. \( V_1...V_n \) // for back to back testing
\( P:= \)Generate Test Cases using structural and functional testing techniques
//Generate a test set of n test cases
\( \text{FAIL}:=\)FALSE // initialize the variable
While (Not (terminating condition)) do
{
Generate the mutant of \( V \)
While (~\text{FAIL})
{
Generate the next generation S (of size n) from P using Genetic Algorithm.
Perform back to back testing
If failure
{\n\text{FAIL}:=\)TRUE
Then
add the test case killing the mutant to the population P
}
}
}
End.

Figure 1: Illustrate the Genetic Algorithm Steps

**Procedure** \text{GA}(\varphi, \theta, n, r, m)

// \( \varphi \) is the fitness function for ranking individuals
// \( \theta \) is the fitness threshold, which is used to determine when to halt
// \( n \) is the population size in each generation (e.g., 100)
// \( r \) is the fraction of the population generated by crossover (e.g., 0.6)
// \( m \) is the mutation rate (e.g., 0.001)
\( P:=\) generate n individuals at antirandom
// initial generation is generated antirandom using Gray code or Hamming code
\text{while} \( \max(\varphi(h)) < \theta \) \text{do}
//define the next generation S (also of size n)
Reproduction step: Probabilistically select \((1-r)n\) individuals of P and add them to S, where the probability of selecting individual \( h_i \) is \( \text{Prob}(h_i) = \varphi(h_i)/\Sigma(\varphi(h_j)) \)
Crossover step: Probabilistically select \( r\times n/2 \) pairs of individuals from P according to \( \text{Prob}(h_i) \)
\text{foreach} pair \((h_1, h_2)\), produce two offspring by applying the crossover operator and add these offspring to S
Mutate step: Choose m% of S and randomly invert one bit in each
\( P := S \)
\text{end}\_while
In order to make the experiment realistic, an attempt was made to choose an application that would normally be a candidate for the inclusion of fault tolerance. The problem that was selected for programming is a simple and realistic data structure sorting system. The N version program read some data that represents as test cases an array (integer or float). The outputs from the N-version software are compared to check the correctness of the system. To check the efficiency of N-version system and the completeness of the test set, tests are performed by introducing the mutant in the software. This program was originally written in MATLAB, and the program has been subjected to several thousands test cases. The figure 2 shows the block diagram for proposed approach of test case generation.

Assumptions made are as under:

1. **Encoding**: Direct value encoding can be used in problems where some more complicated values such as real numbers are used. In the value encoding, every chromosome is a sequence of some values. Values can be anything connected to the problem, such as (real) numbers, chars or any objects.

2. **Selection**: From a population of individuals, we wish to give the fitter individuals a better chance to survive to the next generation. We do not want to use the simple criterion “keep the best n individuals.” It turns out nature does not kill all the unfit genes. They usually become recessive for a long period of time. But then they may mutate to something useful. Therefore, there is a tradeoff for better individuals and diversity. The initial population is taken not randomly but using the hamming code or gray code as antirandom population. The individuals are selected according to Rank selection criteria. Rank selection ranks the population first and then every chromosome receives fitness value determined by this ranking. The worst will have the fitness 1, the second worst 2 etc. and the best will have fitness N (number of chromosomes in population).

3. **Crossover**: Two point crossover - two crossover points are selected, binary string from the beginning of the chromosome to the first crossover point is copied from the first parent, the part from the first to the second crossover point is copied from the other parent and the rest is copied from the first parent again.

4. **Mutation**: Randomly change one or more digits in the string representing an individual.

7. **RESULTS**

Fault propagation spreads the faulty result in a problem to the output and causes a failure of the program. It can be revealed by an execution of the program. A fault may be any occurrence of program in any particular version that causes that version to fail when that software is executed on some test case. The numbers of faults found in the individual versions is shown in Table 1. All of these faults have been found and corrected. Many of the faults were unique to individual versions but several occurred in more than version.

The following graphs depict the comparison approach of random vs. Genetic Algorithm.

8. **CONCLUSION**

Software testing is an optimization problem with the objective that the efforts consumed should be minimized and the number of faults detected should be maximized. Software testing is considered most effort consuming activity in the software development. Although a number of testing techniques and adequacy criteria have been suggested.
in the literature but it has been observed that no technique/criteria is sufficient enough to ensure the delivery of fault free software consequential to the need of automatic test case generation to minimize the cost of testing. As discussed the techniques like random and anti-random testing techniques have shown the good results. The proposed technique using GA with antirandom population and employing n-version testing has shown the average 20% and maximum 40% improvement over the random test case generation Although the cost incurred in producing N versions of the same module will be large but by using the technique judiciously in those modules only where a high level of reliability is required, the benefits accrued override the cost incurred.

Table 1

<table>
<thead>
<tr>
<th>Sr. No</th>
<th>Mutant</th>
<th>No. of mutant killed per No. of test cases applied</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>GA with Random population</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>OFF</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>RRF</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>ASF</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>LNF</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>LRF</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>MEF</td>
<td>3</td>
</tr>
</tbody>
</table>

Graph 1: Compare the Numbers of Mutant Kill per 10 test Cases

Graph 2: Compare the Numbers of Mutant kill per 50 Test Cases

REFERENCES


