Pragmatic Substantiation Of An Efficient Test Data Generation Algorithm Based On Adequacy Testing Criteria

I.Surya Praba¹, Assoc.Prof, Dept of CSE, ipsurya17@gmail.com
Institute of Aeronautical Engineering College, Hyderabad-43

DR.V.Rama Krishna², Prof, Dept of CSE,
KL UNIVERSITY, Vijayawada.

Abstract

The length and the complexity of the software are rising day by day. This rising complexity has increased the demand for techniques that can generate test data effectively. Test data generation techniques selects from the input domain of the program, those input values that satisfies a pre-defined testing criteria. In this paper, we propose a new test data generation algorithm. Our algorithm generates test data using adequacy based testing criteria that aims to generate an adequate test data set by using the concept of mutation analysis. In general, mutation analysis is applied after the test data is generated. But, our algorithm applies mutation analysis at the time of generating test data only rather than applying it after the generation of test data.

Incorporation of mutation analysis at the time of test data generation leads to the generation of test data that is itself adequate and hence we need not check for its adequacy after its generation. We also make use of genetic algorithms that explore the complete domain of the program to generate near-global optimum test data set. In order to analyze our algorithm, we evaluate it using fifty real time programs written in C language. The program set contains programs ranging from 35 to 350 lines of source code and includes from very basic to very complex programs. We compare our algorithm with path testing and condition testing techniques (that uses reliability based testing criteria) for these fifty programs in two categories viz. number of generated test cases and the time taken to generate test cases.

The results suggest that our adequacy based algorithm is better than the reliability based path testing and condition testing techniques in both of these categories. Thus this algorithm may significantly reduce the time of test data generation.

Keywords-Software Testing, Genetic Algorithms, Adequacy based testing criteria, Reliability based testing criteria, Mutation Analysis.

1. INTRODUCTION

Software testing is defined as a process of executing a program with the intent of finding the errors [1]. It is an investigation that is conducted to provide stakeholders the information about the quality of the product/service under test [2]. Time is one of the most important factors to be considered while performing software testing. Testing accounts for nearly 50% of the total development time and cost of the software [1]. We generally have only limited amount of time to test and it is very essential to perform testing within the time bounds in order to ensure timely delivery of the software and to face competition in the software market.

Exhaustive testing is never possible because it would be an extremely time consuming task. Thus, to test within the time bounds, it is necessary to limit the process of testing. For this, the concept of testing criteria is used. In this context, testing can be defined as a process of generating test data that satisfies pre-set testing criteria. As soon as the testing criterion is satisfied, we can stop testing. Software testing is essentially defined as the combination of software verification and validation testing. Verification testing involves testing the
intermediate work products that are produced during the process of software development.

This type of testing includes reviews, walkthroughs, inspections etc. Validation testing involves testing the final end product. It involves functional (black box) and structural (white box) testing. Functional testing involves testing the functionality of the system in terms of input output relationship. It is also known as specification based testing that test the software product using software specification. This type of testing involves equivalence class testing [3], random testing [3] etc. Structural testing deals with testing the structure of the system based on the information about the source code of that system. Structural test criteria are broadly classified into two categories viz. reliability based testing criteria and adequacy based testing criteria [4].

Reliability based testing criteria focuses upon generating test cases that are reliable. A test case set is said to be reliable if its execution ensures that the program is correct on all its inputs [4]. These testing criteria are used to show the correctness of the program in terms of achieving some coverage such as path coverage [5, 6, 7, 8, 9], branch/condition coverage [10, 11, 12] etc. Adequacy based testing criteria is used to show the adequacy of the test cases in terms of whether or not they can identify faults in the program. A test case set is adequate if it causes all the incorrect versions of the program to fail to execute successfully [4]. The difference between reliability based and adequacy based testing criteria [4] is that reliability based testing criteria verifies the correctness of the program relative to a pre-defined testing criteria but does not take into account the identification of faults in the program.

Thus, even after the test data is generated, the tester has no assurance of whether or not there are any faults in the program. Contrarily, adequacy based testing criteria verifies the adequacy of the test data. This informs the tester about whether or not there are any faults in the program, so that the quality of the program can be fostered by focusing upon these faults.

In this paper, we propose a test data generation algorithm that generates test data using adequacy based testing criteria and genetic algorithms. The basic idea of this algorithm was given in our previous work [13]. In this paper, we mainly focus on providing the algorithm in a formalized manner and on evaluating the algorithm by comparing it with other test data generation techniques. The aim is to prove the effectiveness of our proposed algorithm based on adequacy based testing criteria. Our algorithm applies mutation analysis to generate an adequate test data set.

In general mutation analysis is applied after the test data is generated. But, in our algorithm, we apply mutation analysis at the time of test data generation only. This has two significant benefits: First, the generated test data set is guaranteed to be adequate. Hence, mutation analysis need not be applied after the test data is generated. Second, the total time consumed to generate adequate test data involves only the time to generate test data and not the time to check its adequacy. This is so because the generated test data is assured to be adequate. This has an advantage over other techniques that are based on reliability based testing criteria as the total time in these techniques is the sum of both the time taken to generate test data and the time taken to check its adequacy. We use genetic algorithms (GA) because of their ability to exhaustively search the input domain of the program to provide near global optimum solutions. The use of genetic algorithms in the field of software testing is facilitated by the fact that the software testing problem can be formulated as a search optimization problem. For instance, we can state our testing problem as a problem of searching through the input domain of the program, for those input values that satisfies an adequacy testing criteria. We use GA optimization tool of MATLAB as our genetic algorithm software. The complete details of GA can be obtained from [14].

In order to validate our algorithm, we apply it on fifty real time programs written in C language and compare it with path testing and condition testing techniques (based on reliability testing criteria) for these programs in two categories viz. number of generated test cases and time taken to generate test cases. We have chosen path testing and condition testing technique because these two are
profundely used in the field of software testing [5, 6, 7, 8, 9, 10, 11, 12]. In fact, path testing technique alone can detect almost 65% of the errors in the program [15]. The experimental results suggests that there is a significant reduction in number of test cases and significant savings in time (taken to generate test cases) in our adequacy based algorithm over reliability based path testing and condition testing techniques.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 presents the proposed algorithm. Section 4 presents analytical evaluation and comparison. Section 5 describes the empirical data collection. Section 6 presents the research procedure. Section 7 presents the analysis results. Section 8 summarizes the analysis results. Section 9 concludes the paper and presents the future work.

2. RELATED WORK

Different researchers have focused upon different testing criteria while performing testing. Duran J.W, Ntafos S.C focused upon ‘random testing technique’ to generate test case values. In their work, they compared random testing with partition testing and evaluated the random testing process using various coverage criteria such as segment coverage, branch coverage, etc [3]. DeMillo R, Offutt A.J focused on generating test case values using ‘adequacy based testing criteria’ [4].

Clarke, L focused upon ‘path coverage testing criteria’ to generate the test data. His idea was to select some target paths, execute those paths symbolically, identify constraints, and then generate the test cases such that the identified constraints are satisfied [6]. Korel B focused on ‘path coverage testing criteria’ to generate software test data. He proposed a dynamic path testing technique that generates test cases by executing the program with different possible test case values [7]. Mansour N, Salame M focused on ‘path coverage testing criteria’ for generating software test data using hamming distance as a fitness function [8].

Srivastava P.R, Kim T focused on ‘path coverage testing criteria’ and proposed a technique for generating test cases using genetic algorithm [9]. This technique focused on emphasizing the critical paths during testing. In [9], Srivastava and Kim assigned weights to the edges of the control flow graph. The fitness function was calculated by taking the sum of the weights of all the edges of a particular path. The path with maximum fitness function value is the most critical. Michael et al. used ‘branch coverage testing criteria’ and proposed a technique for automated test data generation using genetic algorithms [12]. Wegener et al. focused on ‘structural test coverage criteria’ to generate test data by using evolutionary approaches like genetic algorithms. In their work, they considered all the test coverage criteria and provided an effective classification of these criteria [16].

Lin J.C, Yeh P.L focused upon ‘path testing criteria’ for automatic test data generation using genetic algorithms [17]. Xanthakis discussed about search based software test data generation using heuristic search procedures such as genetic algorithms [18]. Dahal K, Hossain A focused on UML based software specifications to generate test data using genetic algorithm. In their technique, they used ‘transition coverage test criteria’ that identifies the number of transitions fired on receiving events [19].

Michael identified the application of heuristic search techniques in solving various software engineering problems. This is facilitated by the fact that many of the software engineering problems can be formulated as the search optimization problems [20]. Ghiduk et al. focused upon ‘du paths coverage testing criteria’ to generate test case values. In their work, they focused upon generating the dominance tree from the control flow graph of the program [21]. Rauf A, Anwar S focused on ‘GUI based test criteria’ to generate test data using genetic algorithms. GUI applications are event driven. In this work, the sequence of events represented the candidate test case values and the number of paths followed, out of the total number of paths was used as a fitness function [22].

McMinn P [23], Harman M, Wegener J [24] discussed about search based software test data generation. Ahmed M.A, Hermadi I used path coverage criteria to generate test data using genetic
algorithm [25]. Bouchachia A proposed immune genetic algorithms by incorporating immune operators to the traditional genetic algorithms. In his work, he used ‘Condition coverage testing criteria’ and introduced an additional re-selection operator in addition to the three genetic operators’ viz. reproduction, crossover and mutation [26]. Shen et al. proposed GATS algorithm that is a hybrid scheme of genetic algorithm and tabu search, to generate test data. In his work, he focused upon ‘Function coverage testing criteria’ [27]. Harman M focused on search based software engineering for automated test data generation [28].

Malhotra et al. focused on test data generation using machine learning techniques for the object oriented softwares [31]. We can derive a framework for the classification of testing criteria from the above presented related work. This framework was proposed in [13] and is shown in Fig. 1.

![Testing Criteria Classification Framework](image)

This figure and the above mentioned related works shows that numerous works have been done on reliability based testing criteria but very less work has been done on adequacy based testing criteria. All the related works except for [4] have focused upon generating test data using reliability based testing criteria only. But, in this paper, we use adequacy based testing criteria for test data generation. We also use genetic algorithms in order to add its benefits in our algorithm. This paper shows the effectiveness of adequacy based testing criteria over reliability based testing criteria by comparing both on fifty real time programs.

### 3. PROPOSED ALGORITHM

In this section, we present and explain the proposed algorithm for test data generation. The basic idea of this algorithm was given in our previous paper [13]. However, in this work, we formally provide algorithm for test data generation and validate it with related techniques based on results obtained from fifty different programs written in C language. With respect to mutation analysis, the test data can be either adequate or inadequate. It is adequate, if it can identify faults in the program and inadequate, if it cannot identify faults in the program. The main idea of our algorithm is to separate the adequate test cases from the inadequate ones at the time of test data generation only and thus preventing inadequate test cases from even being generated. Broadly, the algorithm follows three basic steps.

These are mentioned below:

1. Identify mutants from the program.
2. Generate constraints between original program and mutants, such that the solutions of these constraints guarantee the killing of mutants.
3. Solve constraints by using genetic algorithms. These solutions are the desired test case values.

#### 3.1. Proposed Test Data Generation Algorithm

**1: Initialization:**
  a) Identify program mutants.
  b) Build mutation table. Initialize the status field of all mutants to the value ‘unprocessed’.
  c) Make initial settings of the genetic algorithm software.

**2: Repeat** steps 3 to 7 **until** the value of status field for all mutants is not ‘processed’:

...
3: Select an unprocessed mutant from the mutation table.

4: Generate a constraint between original statement and mutated statement of the selected mutant.

5: Construct a fitness function from the generated constraint.

6: Input the fitness function to the genetic algorithm software and generate test case values. Update the status field of the corresponding mutant to the value ‘processed’.

7: Check if the generated test cases could also kill other mutants in the mutation table. If yes THEN
7.1: Update the status field of these mutants to ‘processed’. END IF

8: End repeat until
We explain the steps of the above mentioned algorithm in detail below.

3.2. Mutant Identification

Mutants are identified by following certain rules. Hutchins et al. described the procedure used by Siemens to generate mutants [29] - we paraphrase that description here. The Siemens procedure involves identifying mutants by manually seeding faults in the programs. It involves incorporating (mostly) the single line changes in the program to generate mutants. The same procedure was also used by Rothermel et al., in order to induce mutants in their proposed regression test selection technique [30]. Following the Siemens procedure, we create and follow the following rules for mutant identification as also given in our previous work [13]:

1. Only first order mutants are generated. First order mutants are mutants that contain a single change. In general, only first order mutants are sufficient and are used in testing practically. Second and higher order mutants (that contain multiple changes) make it difficult to manage the mutants, thus adding to complexity. Thus, only first order mutants are generated.

2. In general, there are no limits on the number of mutants that can be generated. To circumvent this problem, we restrict the domain of mutation operators. We generate mutants by applying mutation operators from this domain only. The domain of mutation operators that we use are:

**Operand Replacement Operator:**
Replace a single operand with another operand or a constant.

E.g. if(x>y){} □original statement
if(5>y){} □mutated statement generated by replacing x by constant 5

**Expression Modification Operator:**
Replace an operator with some other operator or insert new operator.

E.g. if(x==y){} □original statement
if(x>=y){} □mutated statement generated by replacing == by >=

**Statement Modification Operator:**
Delete the entire if-else statement.
Replace a line by a return statement, etc.

The mutants, once identified, are maintained in a mutation table. The mutation table is used to keep track of all the identified mutants. The table contains certain fields’ viz. Mutant_Id, Parent statement, Mutated Statement and Status [13]. This is shown in Table 1.

<table>
<thead>
<tr>
<th>Mutant_Id</th>
<th>Parent Statement</th>
<th>Mutated Statement</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mi: Statement Number</td>
<td>x&gt;y</td>
<td>x&lt;y</td>
<td>unprocessed/processed</td>
</tr>
</tbody>
</table>

Table 1  Mutation Table

- **Mutant_Id** is a representative icon for describing the mutants. It is a symbol Mi followed by the statement number that is mutated. The symbol and statement number are separated by a colon (:).

- **Parent statement** is a statement in the original program that we target to change.
3.3. Constraint Generation

The constraint generation is an important step in our algorithm. The constraints are generated between the original unmodified statement of the program and the modified statement of the mutant. The constraints are generated such that the solutions of these constraints assure the killing of mutants. These solutions represent the desired test case values. An important property to be kept in mind while generating constraints is that the constraints should be generated such that they cause the mutants to follow a different execution path than the original program, after the execution of the mutated statement. The complete details of constraint generation can be obtained from [13].

3.4. Fitness Function Construction

After the constraints are generated, we need to solve these constraints to generate the test cases. The constraints are solved using genetic algorithms. Genetic algorithms are heuristic search techniques that explore the complete domain of the program to provide near-global optimum solutions. Genetic algorithms require using fitness function. The fitness function is a measure of fitness of an input value with respect to a globally optimum solution. Higher fitness function value means the corresponding input is more close to the globally optimum solution and hence is considered to be more fit than the input with comparatively less fitness function value. Construction of fitness function is a very important step. An efficiently constructed fitness function can cause genetic algorithm to work better. In our algorithm, we construct fitness function from the generated constraints using an appropriate fitness function concept. The concept that we use in our algorithm is same as that proposed in [5]. This concept is based on computing the difference between the values of program variables (depending upon the condition) such that the difference is minimized. The input values for which the value of fitness function is minimum represent the most optimized test case values. The complete details of fitness function construction can be obtained from [13].

3.5. Initial Settings of Genetic Algorithm Software

We have used the same initial settings as that decided in [13]. We present the same for the sake of convenience in Table 2.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genetic Algorithm Software</td>
<td>GA Optimization Tool of MATLAB</td>
</tr>
<tr>
<td>Selection Technique</td>
<td>Roulette Wheel</td>
</tr>
<tr>
<td>Cross over Rate</td>
<td>80% (or 0.8)</td>
</tr>
<tr>
<td>Mutation Rate</td>
<td>0.01</td>
</tr>
<tr>
<td>Representation Scheme</td>
<td>Double Vector</td>
</tr>
<tr>
<td>Cross over Technique</td>
<td>Single Point Cross Over</td>
</tr>
<tr>
<td>Initial Population Size</td>
<td>100</td>
</tr>
<tr>
<td>Maximum no. of Generations</td>
<td>100</td>
</tr>
</tbody>
</table>

Table 2: Genetic Algorithm Software Setting

4. ANALYTICAL EVALUATION AND COMPARISON

Software test data generation is a very crucial and important task and it can be done in a variety of ways. Some of the test data generation techniques are based on generating test data on the basis of program specifications (known as functional testing) including equivalence class testing, random testing, etc., while others are based on generating test data on the basis of information about the code of the program (known as structural testing) including path testing, branch testing, du-path testing, etc.

Our algorithm generates test data on the basis of source code of the program and the modified version of the program known as mutant. Our test data generation algorithm is basically based on the adequacy test criteria that incorporate mutation analysis at the time of test data generation only. This leads to the generation of test cases that are adequate. This has an advantage over other test data generation techniques that first generates test
data and then applies mutation analysis to check its adequacy.

We compare our algorithm with two other testing techniques: path testing and condition testing. We have chosen these two techniques as they are profoundly used in the field of software testing [5, 6, 7, 8, 9, 10, 11, 12]. Path testing involves testing the paths (especially the critical paths) of the program. It covers almost all the components of the program including statements, conditions, loops etc. In fact path testing alone can detect almost 65% of the errors in the program [15].

Condition testing involves testing each condition for both its true and false counterparts for at least once. The idea is to show the advantage of adequacy based proposed algorithm over reliability based path testing and condition testing techniques.

To discuss the advantage of our algorithm over other techniques, we develop an analytical framework. This framework evaluates our algorithm and compares it with other techniques in two categories: time taken to generate test cases, and number of generated test cases.

**Time taken to generate test cases:** Time is one of the most crucial factors in any phase of the software development life cycle. Software testing is also no exception. It alone accounts for nearly 50% of the total development cost and time of the software. Due to the growing competition in the software market and the need to deliver the superior quality software quickly, we generally have only limited amount of time to generate test cases for testing software. Thus, it is very important to generate good quality test cases in a reasonable amount of time. In this category, we evaluate our algorithm by computing the percentage savings in time (taken to generate test cases) achieved in our algorithm over path testing and condition testing technique. The notations for computing percentage savings in time are given in Table 3.

In Table 3, M=P=C i.e. number of mutants, paths and conditions considered for any program under test is kept to the same. The values for percentage savings in time are computed as:

**For our algorithm vs. path testing technique:**

\[
\text{% savings in time} = \frac{P_a - M_a}{P_a} \times 100
\]

(1)

**For our algorithm vs. condition testing technique:**

\[
\text{% savings in time} = \frac{C_a - M_a}{C_a} \times 100
\]

(2)

**Number of generated test cases:** The number of test cases that are generated in any testing technique is one of the important means to ensure the effectiveness of that testing technique. No. of generated test cases is directly related to the amount of time required to execute those test cases. Furthermore, we cannot test endlessly as we generally have limited testing resources (cost and time) and we have to test within those bounds only. Thus to meet time bounds, it is very important that a test data generation technique should generate a
reasonable number of test cases and it is quite obvious to say that a technique that generates less number of good test cases is more effective than a technique that generates comparatively more number of test cases.

We evaluate our algorithm in this category by comparing the number of test cases that are generated in our algorithm and comparing it with number of test cases that are generated in path testing and condition testing technique. We also compute the percentage reduction achieved (in number of test cases) in our algorithm over path and condition testing technique.

The values for percentage reduction are calculated as:

For our algorithm vs. path testing technique:

\[
\text{% reduction in no. of test cases} = \frac{\text{no. of test cases in path testing technique} - \text{no of test cases in our algorithm}}{\text{no. of test cases in path testing technique}}
\]  

(3)

For our algorithm vs. condition testing technique:

\[
\text{% reduction in no of test cases} = \frac{\text{no.of test cases in condition testing technique} - \text{no of test cases in our algorithm}}{\text{no.of test cases in condition testing technique}}
\]  

(4)

5. PRAGMATIC DATA COLLECTION

Evaluating the performance of any technique requires selecting certain subject programs which forms the basis for evaluation. To evaluate the performance of our proposed algorithm and to compare it with other techniques, we have selected fifty real time programs written in C language from the sources mentioned in [32]. The subject programs we have chosen are described in Table 4. The programs range from 35 to 350 lines of source code.

We have selected a large program base that contains programs ranging from very basic such as computing the grade of student, finding the biggest of three numbers to very complex such as implementing the binary search tree and finding the intersection of two linked lists. We have chosen a diversified range of programs including mathematical problems such as finding roots of quadratic equation, triangle classification problem, computing the median of the triangle; general logical problems such as checking for the armstrong number, magic number, palindrome number; business problem such as payroll system, commission problem, credit risk analysis; data structures such as linked list, sorting (insertion sort, selection sort, bubble sort, merge sort, heap sort, quick sort, shell sort), searching (linear search, binary search) etc. All the programs are written in standard C language that makes it easier to work with these programs.
6. RESEARCH PROCEDURE

In this section, we describe the procedure that we have used to evaluate the performance of our proposed algorithm and to compare our algorithm with other test data generation techniques.

We have compared our algorithm with path testing technique and condition testing technique and all three are based on genetic algorithm. The path testing technique is described in [5] and the condition testing technique is described in [10].

The research procedure for the evaluation is described below: In our algorithm, for each subject program, we identify some mutants. After that we identify the constraints and generate fitness function for each mutant using the procedure described in section 3. In the path testing technique, we identify some paths using the control flow graph of the program and follow the approach described in [5] to generate the fitness functions.

In the condition testing technique, we select some conditions and follow the approach described in [10] to generate the fitness functions. For any subject program under consideration, the number of mutants identified, number of paths identified and the number of conditions selected must be the same. The number of mutants, paths and conditions selected may vary from one program to another depending upon its size and complexity.

Once the fitness functions for mutants (in our algorithm), paths (in path testing technique), and conditions (in condition testing technique) are generated, we input these fitness functions one by one to the GA optimization tool of the MATLAB and generate the test case values. For each of the mutants, paths and conditions, we generate and record the test case values 10 times and then average the result for better analysis.

7. ANALYSIS RESULTS

In this section, we describe the experimental analysis results in detail as an evidence to show the advantage of our proposed algorithm over other techniques. The results are based on fifty real time C programs. As mentioned in section 4, we evaluate our algorithm in terms of number of generated test cases and percentage savings obtained in time taken to generate test cases. Section 7.1 presents the results for number of test cases. Section 7.2 presents the results for percentage savings in time taken to generate test cases.

7.1 Results for Number of Test Cases
We compute the number of test cases in our algorithm, path testing technique and condition testing technique. We also calculate the values of percentage reduction achieved in number of test cases in our algorithm over path testing and condition testing technique. These values are computed using equations 3 and 4. Section 7.1.1 presents the results for our proposed algorithm vs. path testing technique. Section 7.1.2 presents the results for our proposed algorithm vs. condition testing technique.

### 7.1.1 Proposed Algorithm vs. Path Testing Technique

Table 5 presents the results for the number of generated test cases and the percentage reduction achieved in number of test cases in our algorithm over path testing technique.

<table>
<thead>
<tr>
<th>S. No.</th>
<th>Program</th>
<th>No. of test cases in our algorithm</th>
<th>No. of test cases in path testing technique</th>
<th>% reduction in no. of test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SP01</td>
<td>6</td>
<td>12</td>
<td>50%</td>
</tr>
<tr>
<td>2</td>
<td>SP02</td>
<td>5</td>
<td>5</td>
<td>0%</td>
</tr>
<tr>
<td>3</td>
<td>SP03</td>
<td>5</td>
<td>8</td>
<td>37.5%</td>
</tr>
<tr>
<td>4</td>
<td>SP04</td>
<td>2</td>
<td>4</td>
<td>50%</td>
</tr>
<tr>
<td>5</td>
<td>SP05</td>
<td>3</td>
<td>4</td>
<td>25%</td>
</tr>
<tr>
<td>6</td>
<td>SP06</td>
<td>4</td>
<td>7</td>
<td>57.14%</td>
</tr>
<tr>
<td>7</td>
<td>SP07</td>
<td>2</td>
<td>4</td>
<td>50%</td>
</tr>
<tr>
<td>8</td>
<td>SP08</td>
<td>2</td>
<td>4</td>
<td>50%</td>
</tr>
<tr>
<td>9</td>
<td>SP09</td>
<td>1</td>
<td>2</td>
<td>50%</td>
</tr>
<tr>
<td>10</td>
<td>SP10</td>
<td>2</td>
<td>4</td>
<td>50%</td>
</tr>
<tr>
<td>11</td>
<td>SP11</td>
<td>3</td>
<td>4</td>
<td>25%</td>
</tr>
<tr>
<td>12</td>
<td>SP12</td>
<td>3</td>
<td>3</td>
<td>0%</td>
</tr>
<tr>
<td>13</td>
<td>SP13</td>
<td>2</td>
<td>2</td>
<td>0%</td>
</tr>
<tr>
<td>14</td>
<td>SP14</td>
<td>3</td>
<td>3</td>
<td>0%</td>
</tr>
<tr>
<td>15</td>
<td>SP15</td>
<td>4</td>
<td>8</td>
<td>50%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S. No.</th>
<th>Program</th>
<th>No. of test cases in our algorithm</th>
<th>No. of test cases in path testing technique</th>
<th>% reduction in no. of test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>SP16</td>
<td>1</td>
<td>3</td>
<td>66.66%</td>
</tr>
<tr>
<td>17</td>
<td>SP17</td>
<td>4</td>
<td>5</td>
<td>20%</td>
</tr>
<tr>
<td>18</td>
<td>SP18</td>
<td>3</td>
<td>4</td>
<td>25%</td>
</tr>
<tr>
<td>19</td>
<td>SP19</td>
<td>8</td>
<td>9</td>
<td>11.11%</td>
</tr>
<tr>
<td>20</td>
<td>SP20</td>
<td>5</td>
<td>11</td>
<td>54.54%</td>
</tr>
<tr>
<td>21</td>
<td>SP21</td>
<td>6</td>
<td>7</td>
<td>14.33%</td>
</tr>
<tr>
<td>22</td>
<td>SP22</td>
<td>6</td>
<td>7</td>
<td>14.33%</td>
</tr>
<tr>
<td>23</td>
<td>SP23</td>
<td>4</td>
<td>5</td>
<td>20%</td>
</tr>
<tr>
<td>24</td>
<td>SP24</td>
<td>11</td>
<td>19</td>
<td>42.11%</td>
</tr>
<tr>
<td>25</td>
<td>SP25</td>
<td>17</td>
<td>29</td>
<td>41.33%</td>
</tr>
<tr>
<td>36</td>
<td>SP36</td>
<td>5</td>
<td>6</td>
<td>15.62%</td>
</tr>
</tbody>
</table>

Table 5 Number of Test Cases In Our Algorithm And Path Testing Technique
The values in Table 5 shows that the number of test cases that are generated is less in our algorithm as compared to path testing technique for most of the programs with the exception of some where it is equal in our algorithm as compared to path testing technique. For e.g. the value 0 in row 14 of Table 5 signify that the number of test cases is equal in our algorithm as compared to path testing technique for the Armstrong number program. **Positive value of percentage reduction indicates that number of test cases is less in our algorithm. Zero value indicates that the number of test cases is same in both the techniques.**

The bar chart depicting the number of test cases in our algorithm and path testing technique is shown in Fig. 2.

![Figure 2. Chart showing number of test cases in our algorithm and path testing technique](image2)

It is clear from Fig. 2 that the number of generated test cases is less in our algorithm as compared to path testing technique.

The bar chart depicting the percentage reduction in number of test cases achieved in our algorithm over path testing technique is shown in Fig. 3.

![Figure 3. Chart showing %reduction in our algorithm over path testing technique](image3)

From Table 5 and Fig. 3, we observe that there is 11.11% to 88.52% reduction in number of test cases in our algorithm over path testing technique.

### 7.1.2 Proposed Algorithm vs. Condition Testing Technique

Table 6 presents the results for the number of generated test cases and percentage reduction achieved in number of test cases in our algorithm over condition testing technique.

<table>
<thead>
<tr>
<th>S. No</th>
<th>Program</th>
<th>No. of test cases in our algorithm</th>
<th>No. of test cases in condition testing technique</th>
<th>%reduction in no. of test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SP1</td>
<td>8</td>
<td>19</td>
<td>57.89%</td>
</tr>
<tr>
<td>2</td>
<td>SP2</td>
<td>5</td>
<td>7</td>
<td>42.86%</td>
</tr>
<tr>
<td>3</td>
<td>SP3</td>
<td>5</td>
<td>13</td>
<td>61.54%</td>
</tr>
<tr>
<td>4</td>
<td>SP4</td>
<td>2</td>
<td>6</td>
<td>66.67%</td>
</tr>
<tr>
<td>5</td>
<td>SP5</td>
<td>3</td>
<td>6</td>
<td>50%</td>
</tr>
<tr>
<td>6</td>
<td>SP6</td>
<td>3</td>
<td>7</td>
<td>57.14%</td>
</tr>
<tr>
<td>7</td>
<td>SP7</td>
<td>2</td>
<td>3</td>
<td>75%</td>
</tr>
<tr>
<td>8</td>
<td>SP8</td>
<td>2</td>
<td>3</td>
<td>75%</td>
</tr>
<tr>
<td>9</td>
<td>SP9</td>
<td>1</td>
<td>3</td>
<td>66.67%</td>
</tr>
<tr>
<td>10</td>
<td>SP10</td>
<td>2</td>
<td>7</td>
<td>71.43%</td>
</tr>
</tbody>
</table>
TABLE 6 NUMBER OF TEST CASES IN OUR ALGORITHM AND CONDITION TESTING TECHNIQUE

The values in Table 6 shows that the number of test cases that are generated is less in our algorithm as compared to condition testing technique for most of the programs with the exception of some where it is equal in both the techniques. For e.g. the value 0% in row 38 of Table 6 signifies that the number of test cases is equal in our algorithm as well as condition testing technique for the merge sort program.

The bar chart depicting the number of test cases in our algorithm and condition testing technique is shown in Fig. 4.

FIGURE 4. CHART SHOWING NUMBER OF TEST CASES IN OUR ALGORITHM AND CONDITION TESTING TECHNIQUE

It is clear from Fig. 4 that the number of generated test cases is less in our algorithm as compared to condition testing technique.
The bar chart depicting the percentage reduction in number of test cases achieved in our algorithm over condition testing technique is shown in Fig. 5.

![Bar Chart](image.png)

The values in Table 7 indicate that there is a significant saving in time taken to generate test cases in our algorithm over other techniques for most of the programs with the exception of few where no savings are obtained and yet another few where savings are obtained in other techniques and not in our algorithm.

The value 66.66% in row 20, column 4 of Table 7 indicates that for the binary search program, we generated test cases for only one mutant out of the 3 identified mutants in our algorithm whereas we had to generate test cases for all the 3 identified paths in path testing technique. Thus, there is a 66.66% saving in time in our algorithm over path testing technique (refer to section 4 for the equation to calculate % savings).

This occurred because for the binary search program, the test case that we generated for one mutant also killed other two mutants in our algorithm, but in path testing technique, a test case generated for one path did not cover any other path, thus we had to generate test cases for all the three paths. The value nil in row 21, column 4 and 5 of Table 7 indicates that there is no saving in terms of time i.e. we generated test cases for all the 5 mutants, paths and conditions.

The value -50% in row 19, column 5 of Table 7 indicates that there is a 50% savings in time taken to generate test cases. But negative sign indicates that savings is in other direction i.e. 50% savings is obtained in condition testing technique and not in our algorithm. The bar charts depicting the percentage savings in time taken to generate test cases in our algorithm over path testing technique and condition testing technique are shown in Fig. 6 and Fig. 7 respectively.

![Chart 6](image6.png)

Figure 6. Chart showing % savings in time in our algorithm over path testing technique

![Chart 7](image7.png)

Figure 7. Chart showing % savings in time in our algorithm over condition testing technique
From Table 7, Fig. 6 and Fig. 7, we observe that there is 20% to 66.66% savings in time (taken to generate test cases) in our algorithm over path testing and condition testing technique.

8. SUMMARY OF ANALYSIS RESULTS AND LIMITATIONS OF THE PROPOSED ALGORITHM

The results obtained from the analysis of the proposed algorithm can be summarized as follows:
- Our algorithm can reduce the time taken to generate test cases. This has benefits as time is one of the very important factors to be considered while performing testing and it is very important to test efficiently in as much minimum time as possible.
- Our algorithm generates less number of test cases as compared to other test data generation techniques.
- There exist some programs for which the proposed algorithm offers little in terms of savings.
- The effectiveness of test data generation algorithms depends upon certain factors such as program size, program structure, type of mutants introduced etc. Our algorithm has the following limitations:
  - We validate our algorithm on programs that accepts numerical inputs (integer, float, double etc.). Validation using non numeric inputs (characters, strings, etc.) is not yet considered in this work.
  - In this work, we focus on using efficient testing criteria to generate test cases. But we use the same fitness function concept as that used in other test data generation techniques.

9. CONCLUSIONS AND FUTURE WORK

In this paper, we propose a test data generation algorithm based on adequacy based testing criteria. We also use genetic algorithms because of their ability to provide near global optimum solutions. The main aim is to compare the adequacy based testing criteria with the reliability based testing criteria and to show the effectiveness of former over the latter. In order to do this, we evaluate our algorithm using fifty real time programs written in C language. We compare our algorithm with path testing and condition testing techniques for these fifty programs, in two categories viz. number of test cases and time taken to generate test cases.

The results have been shown with the appropriate tables and charts. These results shows that there is a significant reduction (11.11% - 88.52%) in number of test cases in our algorithm over the path testing technique and significant reduction (10% - 87.80%) in number of test cases in our algorithm over the condition testing technique. There is also significant savings (20% - 66.66%) in time taken to generate test cases in our algorithm over the path testing and condition testing techniques. Thus, the experimental analysis suggests that our algorithm (based on adequacy testing criteria) is better than the path testing and condition testing techniques (based on reliability testing criteria) in terms of both number of generated test cases and the time taken to generate test cases.

Software practitioners can use our algorithm to generate an adequate test data set in comparatively less amount of time than other test data generation techniques that are based on reliability based testing criteria. There are certain directions for future work in this area. First, while the analysis results shown in this work are encouraging, further analysis would be useful and would add to the strength of the proposed algorithm.

Second, the algorithm generates test data for the programs that accepts numerical input (integer, float, double etc.). Future research should consider applying the algorithm on non numeric inputs such as string, characters, etc. Third, while our adequacy based algorithm generates test data in comparatively lesser amount of time than other reliability techniques, future work should focus on further improving the time taken to generate test cases by considering different fitness functions.

10. REFERENCES


ABOUT THE AUTHORS

1.I.L.Surya Prabha presently working as Associate professor in the department of Computer Science and Engineering at Institute of Aeronautical Engg College ,RR Dist.Hyderabad-43 of Andhra pradesh State.INDIA,received her M.TECH.degree in Computer Science & Engineering from JNTU Hyderabad-India. and her area of interest is Software Testing and Quality & Reliability and also Pursuing Her P.hD CSE in KL University. Vijayawada.

2.DR.V.Rama Krishna presently working as Professor in the department of Computer Science and Engineering at K L University, Vijayawada. His area of interest is Software Testing and Quality & Reliability