ISSN 2347-4289

Genemutant: Test Suite Adequacy Check For Path Coverage Testing Based On Mutating Test Suite Using Genetic Algorithm

Dr Namita Gupta

Computer Science and Engineering Department, Maharaja Agrasen Institute of Technology, Rohini, Delhi, India
Email: namita@mait.ac.in

ABSTRACT: Code coverage is a measure used to describe the degree to which the source code of a program is tested by a particular test suite. A program with high code coverage has been more thoroughly tested and has a lower chance of containing software bugs than a program with low code coverage. Many different metrics can be used to calculate code coverage like statement coverage, decision coverage, condition coverage, path coverage etc. Path coverage ensures that every independent path in the program should be executed at least once by the give test suite. The proposed technique check the adequacy of given test suite and design new test cases (if required) by mutating the existing test cases, for path coverage testing based on genetic algorithm using XNOR fitness function.

Keywords : Genetic algorithm, Mutation testing, Path testing

1. INTRODUCTION

Software testing is done to detect the faults in the given source code. Test cases are developed to test the software. Static and dynamic testing techniques are available to test the software. Static testing techniques analyze the target software without actually executing the code to identify the errors. Dynamic testing executes the code using given test cases to detect presence of faults. *Mutation testing* also called *Mutation analysis* or *Program mutation* is the testing technique, used to evaluate the quality of existing software test suite and design new test cases. Mutation testing involves modifying a program's source code. Each mutated version is called a *mutant* and tests detect and reject mutants if the behavior of the original version of the source code differs from the mutant. This is called *killing* the mutant. Test suites are measured by the percentage of mutants that they kill. New tests can be designed to kill additional mutants. The purpose is to help the tester develop effective tests or locate weaknesses in the given test data used for the program or in sections of the code that are seldom or never accessed during execution. Mutants are created based on *mutation.* Many mutation operators have been explored by researchers. Here are some examples of mutation operators for imperative languages:

- Statement deletion.
- Replace boolean expression with *true or false*.
- Replace arithmetic operator with another, e.g. + with *, - and /.
- Replace relational operator with another, e.g. > with >=, == and <=.
- Replace variable with another variable declared in the same scope (variable types must be compatible).

Adequacy of the given test suite is evaluated through mutation score defined as:

mutation score
= number of mutants killed / total number of mutants (1)

Although powerful, Mutation Testing is complicated and time-consuming to perform without an automated tool. Efficiency of the technique depends on the number of mutants created. Few mutants may miss some areas of programs to test if the corresponding test case is missing in the give test suite. Too many mutants increase overhead and technique complexity. To check the adequacy of test suite efficiently and to design new test cases, a new technique is proposed. Instead of modifying the source code to create new mutants, test case is mutated i.e., an input value of the single variable in the given test case is changed to satisfy the given condition at the predicate node. The proposed algorithm is based on Path testing. The objective of path testing is to ensure that each possible independent path through the program is executed at least once. An independent program path is one that traverses at least one new edge in the flow graph. In program terms, this means exercising one or more new conditions. Both the true and false branches of all conditions must be executed. The starting point for path testing is a program flow graph. A flow graph consists of nodes representing decisions and edges showing flow of control. The flow graph is constructed by replacing program control statements by equivalent diagrams. Each branch in a conditional statement (if-then-else or case) is shown as a separate path. An arrow looping back to the condition node denotes a loop. Each statement in the program graph is represented as a separate node where the node number corresponds to the line number in the program. To reduce the complexity of program graph, control flow graph is created where sequential nodes are replaced by single node. Control graph contain following different types of nodes:

i. Starting Node / Initial Node
ii. Terminating Node / Stop Node / End Node
iii. Sequential Node
iv. Predicate/Decision Node
v. Junction Node

Path testing ensures that every path through a program has been executed at least once. If all of these paths are executed we can be sure that every statement in the method has been executed at least once and that every branch has been exercised for true and false conditions. The number of tests that are required to ensure that all paths through the program are exercised is the same as the cyclomatic complexity of the code fragment that is being tested. To evaluate the adequacy of the given test suite and to design new test cases, our proposed algorithm uses genetic algorithm discussed below.

1.1 GENETIC ALGORITHM

Genetic algorithm starts with initial pool of chromosomes and attempts to improve the set by evolution [1][2]. It typically has five parts:

1) a representation of a chromosome - . A chromosome can be a binary string or a more elaborate data structure.
2) an initial pool of chromosomes - It can be randomly produced or manually created.
3) a fitness function - The fitness function measures the suitability of a chromosome to meet a specified objective i.e., a chromosome is fitter if it corresponds to greater coverage.
4) a selection function - The selection function decides which chromosomes will participate in the evolution stage of the genetic algorithm made up by the crossover and mutation operators.
5) a crossover operator and a mutation operator - The crossover operator exchanges genes from two chromosomes and creates two new chromosomes. The mutation operator changes a gene in a chromosome and creates one new chromosome.

Basic algorithm for a Genetic Algorithm [3] follows well-defined steps:

```
initialize (population)
evaluate (population)
while (stopping condition not satisfied) do
{
        selection (population)
        crossover (population)
        mutate (population)
        evaluate (population)
}
```

The algorithm will iterate until the population has evolved to form a solution to the problem (sufficient test cases), or until a maximum number of iterations have taken place (suggesting that a solution is not going to be found given the resources available). In this paper, we discussed the proposed GENEmutant algorithm for path testing. To evaluate the effectiveness of given test suite and identify the section of codes not covered by the given test cases, proposed method applies genetic algorithm based on f(XNOR) fitness function.

2. RELATED WORK

Many researchers have proposed the application of genetic algorithm in different areas of software testing. Riccardo and Langdon [4] described two forms of crossover, one-point crossover and the new strict one-point crossover. Strict one-point crossover, behaves exactly like one-point crossover except that the crossover point can be located only in the parts of the two trees which have exactly the same structure (i.e. the same functions in the nodes encountered traversing the trees from the root node). Sean and Lee [5] showed that crossover is better over mutation given the right parameter settings (primarily larger population sizes). It is observed that mutation is more successful in smaller populations, and crossover is more successful in larger populations. Leonardo [6] proposed a fitness function that incorporates the three basic conditions required by a test case to kill a given mutant of some subject program. Method uses genetic algorithm to search for test cases that satisfy the reachability condition. The proposed fitness function has been implemented, together with a genetic algorithm and mutation analysis tool. Tzung-pei et al. [7] proposed a new genetic algorithm, the dynamic genetic algorithm (DGA). DGA simultaneously uses more than one crossover and mutation operators to generate the next generation. In the next generation, the ratios of crossover and mutation change along with the evaluation results of the respective offspring. Maria and Silvia [8] designed GPTesT, a testing tool based on Genetic Programming. Fault-based testing criteria generally derive test data using a set of mutant operators to produce alternatives that differ from the program under testing by a simple modification. GPTesT uses a set of alternatives genetically derived, which allow the test of interactions between faults. GPTesT implements two test procedures respectively for guiding the selection and evaluation of test data sets. Wen-Yang et al. [9] showed that the probabilities of crossover and mutation are critical to the success of genetic algorithms. A generic scheme for adapting the crossover and mutation probabilities in response to the evaluation results of the respective offspring in the next generation is proposed. Abdelaziz et al. [10] presented a new general technique that combines the concept of spanning set with a genetic algorithm to automatically generate test data for spanning set coverage. Lawrence and Colin [11] present a semantically driven mutation (SDM) technique which is used to improve the mutation operation in genetic programming (GP). The SDM algorithm has been developed based on semantic analysis of the changes caused by the mutation operator. The SDM algorithm works to improve performance by not allowing mutated programs to be produced when they are behaviorally equivalent to the original program, but it also avoid returning to sections of the search space that have effectively already been traversed. William et al. [12] used a multi objective Pareto optimal genetic programming approach to explore the relationship between mutant syntax and mutant semantics with respect to given test sets. The GP algorithm evolves mutant programs according to two fitness functions: semantic difference and syntactic difference. Syntactic distance sums the number of changes weighted by the actual difference. Semantic distance is measured as the number of test cases for which a mutant and original program behave differently. Wang [13] proposed a genetic algorithm based test case prioritization algorithm. Prenal and Kale. [14] discussed the use of genetic algorithms to automatically generate test cases for path testing. Each iteration of the genetic algorithms generates a generation of individuals. But sometimes, solution derived may be trapped around a local optimum and as a result fail to locate required global optimum. Timo [15] analysed the importance of mutation in genetic programming, and reveal new insights into the behavior of mutation-based genetic programming algorithms. Chayanika et al. [16] discussed the application of genetic algorithm in different areas of software testing like white box testing (data flow testing, path testing), black box testing (mutation testing, regression testingGUI testing) etc. Authors of the paper found that by using Genetic Algorithm, the results and the performance of testing can be improved. To conclude, the use of a genetic algorithm to search for test cases that satisfy the reachability condition is not new. The attempt to incorporate necessity and sufficiency conditions is more innovative. In the proposed method, given subject program is not mutated but test cases are mutated and crossover using genetic algorithm based on the selection criteria as determined by the proposed fitness function to search the test cases satisfying path testing. The structure of the paper is organized as follows. In Section 3, the pro-

posed algorithm is discussed; case studies based on proposed method are discussed in section 4. We conclude in section 5.

3. PROPOSED ALGORITHM - GENEMUTANT

For the given source code, cyclomatic complexity technique is used to find all possible independent paths. Test suite is then used to identify the paths covered by the given test. Each independent path is represented as a binary sequence showing 1 if node is present in the path and 0 if node is absent. Thus, each path P_i is represented as a binary sequence containing nodes [n_1 n_2…… n_k], where k is the total no. of nodes in flow graph. Likewise, for each given test case T_j, identify the binary sequence of nodes appearing in the path traversed by it and represent the same as [n_1 n_2 …… n_k]. Next step is to select the test case for each independent path from the given test pool and design new test case(s) for non-traversed path(s). Genetic algorithm is applied to select/design the test cases. Steps for the genetic algorithm are:

1. Fitness Function

Compute $R_{ji} = \forall_{ij} [T_j \text{ XNOR } P_i]$ (2)
such that,

function

$$f(XNOR) = \begin{cases} 1, & t_{jl} \text{ and } p_{il} \text{ both are 0 or 1} \\ 0, & t_{jl} \text{ and } p_{il} \text{ contain unmatch values} \end{cases} \quad (3)$$

i.e., (0 XNOR 1) is 0
(1 XNOR 0) is 0
(0 XNOR 0) is 1
(1 XNOR 1) is 1

A test case T_j is fitted for path P_i if it contains all 1's in resultant row after applying $f(XNOR)$ between them. To avoid redundancy among test cases and minimize the number of test cases, the following rules should be followed:

Rule 1 : If more than one test case traverses the same path, then randomly select single test case for final test suite.

Rule 2 : If a test case traverses more than two independent paths, then preference should be given to such test case over others while selecting the test case for Rule 1.

2. Selection

The selection procedure selects individuals of the current population for development of the next generation. Various alternative methods exist but all follow the idea that the fittest have a greater chance of survival. Selection chooses the chromosomes to be recombined and mutated out of this initial population. [n.gupta] Identify the non-zero predicate nodes n_p in the non-traversed path nodes sequence. If, the immediate next node n_x is not a predicate node then,

$$\text{If } n_x = \begin{cases} 0, & \text{predicate condition } n_p \text{ should be false} \\ 1, & \text{predicate condition } n_p \text{ should be true} \end{cases} \quad (4)$$

If n_x is a predicate node then,

$$\text{If } n_x = \begin{cases} 0, & \text{predicate condition } n_p \text{ should be false} \\ 1, & \text{predicate condition } n_p \text{ should be true} \end{cases} \quad (5)$$

Check the condition at the predicate nodes in the non-traversed path. Match within the selected chromosomes (test case) that satisfy the give condition(s). Test cases that match maximum number of Boolean values of predicate nodes in the non-traversed path are selected to have their solution passed onto the next generation.

3. Crossover

Selected chromosomes are crossover to create next generation. It combines two chromosomes (parents) to produce a new chromosome (child). The new chromosome takes the best characteristics from each of the parents. One point, two points, uniform, arithmetic are few crossover operators [9]. In GENEmutant algorithm, arithmetic operator is used. Arithmetic operator linearly combines two parent chromosome vectors to produce two new children according to the following equations:

$$\text{Child}_1 = a * \text{parent}_1 + (1-a) * \text{parent}_2 \quad (6)$$
$$\text{Child}_2 = (1-a) * \text{parent}_1 + a * \text{parent}_2 \quad (7)$$

where a is a random weighting factor between 0 and 1, chosen before each crossover operation. Take a = 0.5 initially and then increase or decrease its value based on the results obtained.

4. Mutation

Even though the crossover produces a large range of solutions, it may lead the evaluation function towards local maxima. The purpose of mutation in Genetic algorithm is preserving and introducing diversity. For different genome types, different mutation types are available [9] like flip bit, Gaussian, insert, swap, inversion, scramble mutation etc. In GENEmutant algorithm, if the predicate condition is not satisfied by the selected chromosomes after multiple crossover iterations, then existing chromosome is mutated based on the predicate condition.

4. CASE STUDY

In this preliminary study, we used our proposed algorithm on three moderately-sized programs.

EXAMPLE 1

Consider the source code given below

```
start    int isabsequal(int x, int y)
         {
1        if (x==y)
2        return 1;
3        else  if (x==-4)
4        return -1;
5        Else
6        return 0;
stop     }
```

Given source code is represented as control flow graph shown as figure 1 below.

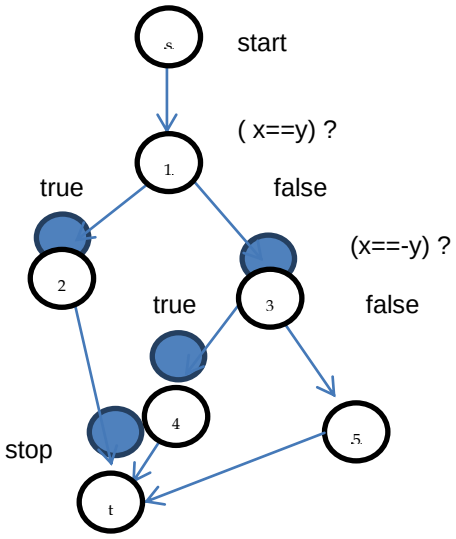

Figure 1: *Control flow graph of example 1*

Node No.	Test Condition at predicate node
n_1	x == y
n_3	x == -4

Table 1 : *Shows the predicate nodes in the flow graph and the condition at each predicate node*

On applying cyclomatic complexity technique,

No. of independent paths = no of predicate nodes + 1
= 2+1 = 3

Independent paths are

(Path 01)P_1= [start n_1 n_2 stop]
(Path 02)P_2= [start n_1 n_3 n_4 stop]
(Path 03)P_3= [start n_1 n_3 n_5 n_6 Stop]

There is a total of 6 nodes from n_1 to n_6 (excluding the Start and Stop node). Each path is represented as a binary sequence of nodes [n_1 n_2 n_3 n_4 n_5 n_6] showing 1 against traversed node and 0 against untraversed node in corresponding path.

Thus,

Path P_1 is equivalent to [1 1 0 0 0 0]
Path P_2 is equivalent to [1 0 1 1 0 0]
Path P_3 is equivalent to [1 0 1 0 1 1]

Path No.	Path Node Binary sequence	Predicate node(s) in given path	Test Condition at predicate node	Boolean value at predicate node
P_1	[n_1 n_2] [1 1 0 0 0 0]	n_1	x == y	True
P_2	[n_1 n_3 n_4] [1 0 1 1 0 0]	n_1	x == y	False
		n_3	x == -4	True
P_3	[n_1 n_3 n_5 n_6] [1 0 1 0 1 1]	n_1	x == y	False
		n_3	x == -4	False

Table 2: *Shows the details about predicate nodes appearing in each independent path*

Given input test cases
T_1: {x=5, y=3}
T_2: {x=9, y=9}

Test case No.	Test Case	Predicate node(s) in flow graph	Test Condition at predicate node	Boolean value at predicate node
T_1	{x=5, y=3}	n_1	x == y	False
		n_3	x == -4	False
T_2	{x=9, y=9}	n_1	x == y	True
		n_3	x == -4	False

Table 3: *Shows the Boolean value of predicate nodes for the given test case(s)*

Each test case T_j, is represented as a binary sequence of nodes appearing in the path traversed by it. Thus, each test case is represented as a binary sequence containing [n_1 n_2 n_3 n_4 n_5] by traversing the flow graph (Figure 1) and using the information given in table 1 and 3.

Test T_1 is equivalent to [1 0 1 0 1 1]

Test T_2 is equivalent to [1 1 0 0 0 0]

Compute $R_{ji}= \forall_{ij}$ [T_j XNOR P_i], i.e.,

T_1XNOR P_1=[1 0 1 0 1 1] XNOR [1 1 0 0 0 0]= [1 0 0 1 0 0]

T_1XNOR P_2=[1 0 1 0 1 1] XNOR [1 0 1 1 0 0]= [1 1 1 0 0 0]

T_1 XNOR P_3=[1 0 1 0 1 1] XNOR [1 0 1 0 1 1] = [1 1 1 1 11]

T_2 XNOR P_1= [1 1 0 0 0 0] XNOR [1 1 0 0 0 0] = [1 1 1 1 11]

T_2 XNOR P_2=[1 1 0 0 0 0] XNOR [1 0 1 1 0 0]= [1 0 0 0 1 1]

T_2 XNOR P_3=[1 1 0 0 0 0] XNOR [1 0 1 0 1 1]= [1 0 0 1 0 0]

Now, a test case T_j is selected corresponding to path P_i that contains all 1's in resultant row.

So, in final test suite, test case **T_1** is selected corresponding to path **P_3** and test case**T_2** is selected corresponding to path **P_1.** It is analyzed that there is no test case corresponding to path **P_2**. Next step is to design new test case from the given test suite. Genetic algorithm is applied to design the new test case for path P_2. Steps for the genetic algorithm are:

1. Selection

Test cases are selected from the given population (test suite) that matches maximum number of boolean values of predicate nodes in the non-traversed path P_2.

Using Tables 2 and 3, following observation is recorded:

Non-traversed path(s)	Test case(s)	Predicate nodes in non-traversed path	Required test condition at non-traversed path predicate nodes	Test condition at predicate nodes as satisfied by test case	Number of matched conditions
P_2	T_1	n_1	False	False	1
		n_3	True	False	
	T_2	n_1	False	True	0
		n_3	True	False	

Table 4: *Shows test case matching to missed path*

As test case T_1 matches one test condition of P_2 non-traversed path, so only test case T_1 is selected for designing new test case for path P_2. Now, next step is to mutate the test case so that it will satisfy all the test conditions of non-traversed path.

2. Mutation

Test case T_1{x=5, y=3} matches the required predicate condition at node n_1(x==y), but violates the predicate condition at node n_3 (x==-4). If the predicate condition is not satisfied by the selected chromosomes, then existing chromosome is mutated based on the predicate condition. Hence chromosome T_1 is mutated as {x=-4, y=3}. Now both the conditions at the predicate nodes are satisfied and mutated test case successfully traverse path P_2. Hence, three test cases T_1 : {x=5, y=3} for path P_3, T_2 : {x=9, y=9} for path P_1 and new test case T_3 : {x=-4, y=3} for path P_2 are required to test all the independent paths of the above problem.

EXAMPLE 2

Consider the source code given below

```
start    int product (int exp)
         {
1    int i = exp;
2    int j = 1;
3    while ( i > 0 ) {
4    j = 2*j;
a.   5    i--;    }
5    return j;
       stop      }
```

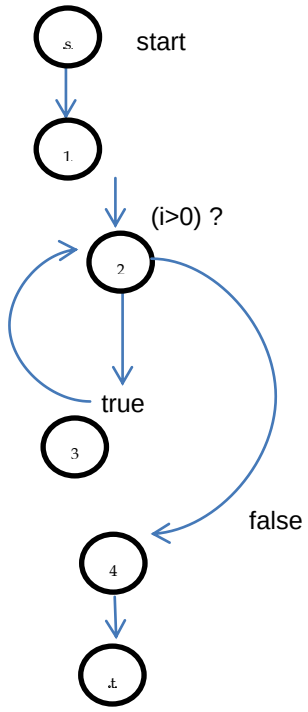

Figure 2: *Control flow graph of example 2*

Node No.	Test Condition at predicate node
n_2	i > 0 (initially i = exp)

Table 5 : *Shows the predicate node(s) in the flow graph and the condition at predicate node*

On applying cyclomatic complexity technique,

No. of independent paths
= no of predicate nodes + 1
= 1+1 = 2

Independent paths are
(Path 01) P_1= [start 1 2 4 stop]
(Path 02) P_2= [start 1 2 3 ((2 3)) 4 stop]
((2 3)) shows n iterations of loop.

There is a total of 4 nodes from n_1 to n_4 (excluding the Start and Stop node). Each path is represented as a binary sequence of nodes [n_1 n_2 n_3 n_4] showing 1 against traversed node and 0 against untraversed node in corresponding path. Thus,

Path P_1 is equivalent to [1 1 0 1]
Path P_2 is equivalent to [1 1 1 1]

Path No.	Path Node Binary sequence	Predicate node(s) in given path	Test Condition at predicate node	Boolean value at predicate node
P_1	[n_1 n_2 n_4] [1 1 0 1]	n_1	i > 0 (initially i = exp)	False
P_2	[n_1 n_2 n_3 n_4] [1 1 1 1]	n_1	i > 0 (initially i = exp)	True

Table 6: *Shows the details about predicate nodes appearing in each independent path*

Given input test cases

T_1: {exp=0}
T_2: {exp=1}
T_3: {exp=-5}

Each test case T_j, is represented as a binary sequence of nodes appearing in the path traversed by it. Thus, each test case is represented as a binary sequence containing [n_1 n_2 n_3 n_4].

Test case No.	Test Case	Predicate node(s) in flow graph	Test Condition at predicate node	Boolean value at predicate node
T_1	{exp=0}	n_1	i > 0 (initially i = exp)	False
T_2	{exp=1}	n_1	i > 0 (initially i = exp)	True
T_3	{exp=-5}	n_1	i > 0 (initially i = exp)	False

Table 7: *Shows the Boolean value of predicate nodes for the given test case(s)*

Each test case T_j, is represented as a binary sequence of nodes appearing in the path traversed by it. Thus, each test case is represented as a binary sequence containing [n_1 n_2 n_3 n_4] by traversing the flow graph (Figure 2) and using the information given in table 5 and 7.

Test T_1 is equivalent to [1 1 0 1]
Test T_2 is equivalent to [1 1 1 1]
Test T_3 is equivalent to [1 1 0 1]

Compute $R_{ji}= \forall_{ij}$ [T_j XNOR P_i], i.e.,

T_1XNOR P_1=[1 1 0 1] XNOR [1 1 0 1] = [1 1 1 1]
T_1XNOR P_2= [1 1 0 1] XNOR [1 1 1 1] = [1 1 0 1]
T_2 XNOR P_1= [1 1 1 1]XNOR [1 1 0 1] = [1 1 0 1]
T_2 XNOR P_2=[1 1 1 1] XNOR [1 1 1 1] = [1 1 1 1]
T_3 XNOR P_1=[1 1 0 1] XNOR [1 1 0 1] = [1 1 1 1]
T_3 XNOR P_2= [1 1 0 1] XNOR [1 1 1 1] = [1 1 0 1]

Now, a test case T_j is selected corresponding to path P_i that contains all 1's in resultant row. So, in final test suite, test case **T_1** and **T_3** is selected corresponding to path **P_1** and test case**T_2** is selected corresponding to path **P_2.** It is analyzed that there is two test cases corresponding to path **P_1.** According to Rule 1 either of the two test cases **T_1** or **T_3** is selected in final test suite to minimize the size of final test suite. Hence, two test cases T_1 : {exp=0} for path P_1, T_2 : {exp=1} for path P_2 are required to test all the independent paths of the above problem.

EXAMPLE 3

Consider the source code given below

```
start    int largest(int x, int y, int z)
         {
1        if (x > y)
2        if (x > z)
3        return x;
4        else
5        return z;
6        else if (y > z)
7        return y;
8        else
9        return z;

stop     }
```

Given source code is represented as control flow graph as shown in figure 3.

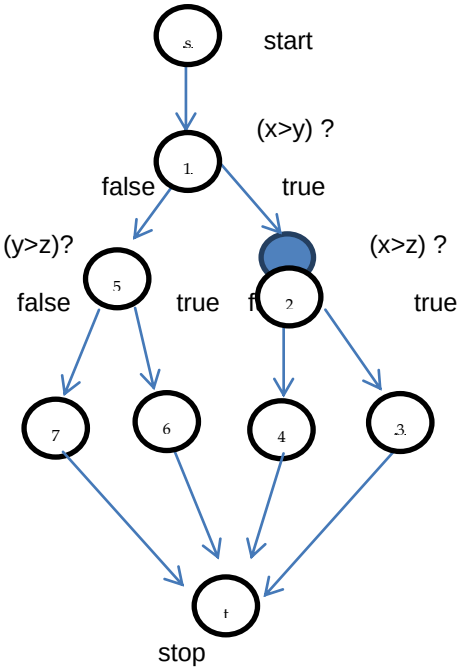

Figure 3: *Control flow graph of example 3*

Node No.	Test Condition at predicate node
n_1	x > y
n_2	x > z
n_5	y > z

Table 8: *Shows the predicate nodes in the flow graph and the condition at each predicate node*

On applying cyclomatic complexity technique,

No. of independent paths = no of predicate nodes + 1
= 3+1 = 4

Independent paths are

(Path 01)P_1= [start n_1 n_2 n_3 stop]
(Path 02)P_2= [start n_1 n_2 n_4 stop]
(Path 03)P_3= [start n_1 n_5 n_6 Stop]
(Path 04)P_4= [start n_1 n_5 n_7 Stop]

There is a total of 7 nodes from n_1 to n_7 (excluding the Start and Stop node). Each path is represented as a binary sequence of nodes [n_1 n_2 n_3 n_4 n_5 n_6 n_7] showing 1 against tra-

versed node and 0 against untraversed node in corresponding path.

Thus,

Path P_1 is equivalent to [1 1 1 0 0 0 0]
Path P_2 is equivalent to [1 0 0 1 0 0 0]
Path P_3 is equivalent to [1 0 0 0 1 1 0]
Path P_4 is equivalent to [1 0 0 0 0 0 1]

Path No.	Path Node Binary sequence	Predicate node(s) in given path	Test Condition at predicate node	Boolean value at predicate node
P_1	[n_1 n_2 n_3] [1 1 1 0 0 0 0]	n_1	x > y	True
		n_2	x > z	True
P_2	[n_1 n_2 n_4] [1 0 0 1 0 0 0]	n_1	x > y	True
		n_2	x > z	False
P_3	[n_1 n_5 n_6] [1 0 0 0 1 1 0]	n_1	x > y	False
		n_5	y > z	True
P_4	[n_1 n_5 n_7] [1 0 0 0 1 0 1]	n_1	x > y	False
		n_5	y > z	False

Table 9: *Shows the details about predicate nodes appearing in each independent path*

Given input test cases
T_1: {x=5, y=2, z=4}
T_2: {x=2, y=7, z=9}
T_3: {x=2, y=7, z=4}
T_4: {x=2, y=7, z=1}

Test case No.	Test Case	Predicate node(s) in flow graph	Test Condition at predicate node	Boolean value at predicate node
T_1	{x=5, y=2, z=4}	n_1	x > y	True
		n_2	x > z	True
		n_5	y > z	False
T_2	{x=2, y=7, z=9}	n_1	x > y	False
		n_2	x > z	False
		n_5	y > z	False
T_3	{x=2, y=7, z=4}	n_1	x > y	False
		n_2	x > z	False
		n_5	y > z	True
T_4	{x=2, y=7, z=1}	n_1	x > y	False
		n_2	x > z	True
		n_5	y > z	True

Table 10: *Shows the Boolean value of predicate nodes for the given test case(s)*

Each test case T_j, is represented as a binary sequence of nodes appearing in the path traversed by it. Thus, each test case is represented as a binary sequence containing [n_1 n_2 n_3 n_4 n_5 n_6 n_7] by traversing the flow graph (Figure 3) and using the information given in table 8 and 10.

Test T_1 is equivalent to [1 1 1 0 1 0 1]
Test T_2 is equivalent to [1 0 0 0 1 0 1]
Test T_3 is equivalent to [1 0 0 0 1 1 0]
Test T_4 is equivalent to [1 0 0 0 1 1 0]

Compute $R_{ji}= \forall_{ij}$ [T_j XNOR P_i], i.e.,

T_1XNOR P_1
=[1 1 1 0 0 0 0] XNOR [1 1 1 0 0 0 0]=[1 1 1 1 1 1 1]

T_1XNOR P_2
= [1 1 1 0 0 0 0] XNOR [1 0 0 1 0 0 0]=[1 0 0 0 1 1 1]

T_1 XNOR P_3
= [1 1 1 0 0 0 0] XNOR [1 0 0 0 1 1 0] = [1 0 0 1 0 0 1]

T_1 XNOR P_4
= [1 1 1 0 0 0 0] XNOR [1 0 0 0 1 0 1] = [1 0 0 1 0 1 0]

T_2 XNOR P_1
= [1 0 0 0 1 0 1] XNOR [1 1 1 0 0 0 0] = [1 1 1 1 1 1 1]

T_2 XNOR P_2
= [1 0 0 0 1 0 1] XNOR [1 0 0 1 0 0 0] = [1 1 1 0 0 1 0]

T_2 XNOR P_3
= [1 0 0 0 1 0 1] XNOR [1 0 0 0 1 1 0] = [1 1 1 1 1 0 0]

T_2 XNOR P_4
= [1 0 0 0 1 0 1] XNOR [1 0 0 0 1 0 1] = [1 1 1 1 1 1 1]

T_3 XNOR P_1
= [1 0 0 0 1 1 0] XNOR [1 1 1 0 0 0 0] = [1 0 0 1 0 0 1]

T_3 XNOR P_2
= [1 0 0 0 1 1 0] XNOR [1 0 0 1 0 0 0] = [1 1 1 0 0 0 1]

T_3 XNOR P_3
= [1 0 0 0 1 1 0] XNOR [1 0 0 0 1 1 0]= [1 1 1 1 1 1 1]

T_3 XNOR P_4
= [1 0 0 0 1 1 0] XNOR [1 0 0 0 1 0 1] = [1 1 1 1 1 0 0]

T_4 XNOR P_1
= [1 0 0 0 1 1 0] XNOR [1 1 1 0 0 0 0] = [1 0 0 1 0 0 1]

T_4 XNOR P_2
= [1 0 0 0 1 1 0] XNOR [1 0 0 1 0 0 0] = [1 1 1 0 0 0 1]

T_4 XNOR P_3
= [1 0 0 0 1 1 0] XNOR [1 0 0 0 1 1 0] = [1 1 1 1 1 1 1]

T_4 XNOR P_4
= [1 0 0 0 1 1 0] XNOR [1 0 0 0 1 0 1] = [1 1 1 1 1 0 0]

Now, a test case T_j is selected corresponding to path P_i that contains all 1's in resultant row. So, in final test suite, test case **T_1** is selected corresponding to path **P_1** and test case**T_2** is selected corresponding to path **P_4** and test case **T_3 or T_3** can be selected corresponding to path **P_3**. It is analyzed that there is no test case corresponding to path **P_2.** Next step is to design new test case from the given test suite for non-traversed path

P_2. Genetic algorithm is applied to design the new test case for path P_2. Steps for the genetic algorithm are:

1. Selection

Test cases are selected from the given population (test suite) that matches maximum number of boolean values of predicate nodes in the non-traversed path P_2.

Using Tables 2 and 3, following observation is recorded:

Non-traversed path(s)	Test case(s)	Predicate nodes in non-traversed path	Required test condition at non-traversed path predicate nodes	Test condition at predicate nodes as satisfied by test case	Number of matched conditions
P_2	T_1	n_1	True	True	1
		n_2	False	True	
	T_2	n_1	True	False	1
		n_2	False	False	
	T_3	n_1	True	False	1
		n_2	False	False	
	T_4	n_1	True	False	0
		n_2	False	True	

Table 11: *Shows test case matching to missed path*

As test cases T_1 , T_2 and T_3 matches single test condition each of P_2 non-traversed path, so T_1 , T_2 and T_3 test cases are selected for designing new test case for path P_2. Now, next step is to mutate the test cases so that mutant test case will satisfy all the test conditions of non-traversed path .

2. Fitness function

Compute $R_i = \forall_i [T_j \text{ XNOR } P_2]$, i=1,2,3
Select the test case T_i corresponding to path P_2 that contains all 1's in resultant row.

3. CrossOver

Test case T_1{ x=5, y=2, z=4} matches the required predicate condition (true) at node n_1 (x > y), test case T_2{ x=2, y=7, z=9} and T_3 {x=2, y=7, z=4}matches the required predicate condition (false) at node n_2 (x > z). Since both the predicate conditions n_1 and n_2 are not being satisfied by any single selected chromosome, hence existing chromosomes T_1 and T_2 or T_3 are crossover to create new chromosome satisfying both the required predicate conditions.

T_1: {x=5, y=2, z=4}
T_2: {x=2, y=7, z=9}

Predicate condition at node n_1 = (x > y) and at node n_2 = (x > z). Since gene x is common in both conditions, it is selected for single arithmetic crossover. Initially, α is chosen as 0.5 randomly.

$Child_1$ is $\{\alpha.x_1 + [1- \alpha].x_2, y_1, z_1\}$
$Child_2$ is $\{[1- \alpha]. x_1 + \alpha.x_2, y_2, z_2\}$

$Child_1$ is {x=3.5,y=2,z=4}
$Child_2$ is {x=3.5,y=7,z=9}

Non-traversed path(s)	Test case(s)	Predicate nodes in non-traversed path	Required test condition at non-traversed path predicate nodes	Test condition at predicate nodes as satisfied by test case	Number of matched conditions
P_2	$child_1$	n_1	True	True	2
		n_2	False	False	
	$child_2$	n_1	True	False	1
		n_2	False	False	

Table 12: *Shows new test case matching missed path*

Since both the predicate conditions n_1 and n_2 are being satisfied by single child chromosome $child_1$, so test case **$child_1$** is selected corresponding to path **P_2** in final test suite. Hence, test cases T_1 : {x=5, y=2, z=4} for path P_1, T_2 : {x=2, y=7, z=9} for path P_4 , T_3 : { x=2, y=7, z=4} for path P_3 and new test case $child_1$: { x=3.5,y=2,z=4} for path P_2 are required to test all the independent paths of the above problem.

5. CONCLUSION AND FUTURE SCOPE

In the paper, Genetic algorithm has been used to search the input domain of the subject program for suitable test cases. Guidance is provided by the fitness function which assigns a non-negative value to each candidate input test case. A test case that matches the maximum predicate nodes in the path is selected for the next generation. But, there are certain limitations of the proposed method.

- Testing all the paths does not mean that all bugs in a program are found. Bugs may be due to missing statements in the code, so there are no paths to execute.
- Some bugs are related to the order in which code segments are executed.
- Also it is practically impossible to test all program paths (e.g., loops).
- Nested conditions in the path are handled in simplified way.

In future, efforts will be made to overcome the above mentioned limitations and hence improve the proposed algorithm. Also more case studies containing complex source code will be considered to measure the efficiency of the proposed method.

REFERENCES

[1] N.K. Gupta and M.K. Rohil, "Using Genetic Algorithm For Unit Testing Of Object Oriented Software", Proceedings of the International Conference on Emerging Trends in Engineering and Technology, 16-18 July 2008, pp. 308-313,.

[2] M. Mitchell, An Introduction to Genetic Algorithms, MIT press, 1996.

[3] Praveen Ranjan Srivastava and Tai-hoon Kim, "Application of Genetic Algorithm in Software Testing", International Journal of Software Engineering and Its Applications , vol. 3, no. 4, October 2009, pp. 87-95.

[4] RICCARDO POLI AND W.B. LANGDON, "GENETIC PROGRAMMING WITH ONE-POINT CROSSOVER AND POINT MUTATION", SOFT COMPUTING IN ENGINEERING DESIGN AND MANUFACTURING, 1997, PP. 180-189.

[5] Sean Luke and Lee Spector, "A Revised Comparison of Crossover and Mutation in Genetic Programming", Proceedings of the Second Annual Conference on Genetic Programming 1997, 1998, pp. 240-248.

[6] Leonardo Bottaci, "A Genetic Algorithm Fitness Function for Mutation Testing", Proceedings of the first International Workshop on Software Engineering using Metaheuristic Innovative Algorithms, Toronto, Ontario, Canada, May 2001.

[7] Tzung-pei Hong , Hong-shung Wang , Wen-yang Lin and Wen-yuan Lee, "Evolution of appropriate crossover and mutation operators in a genetic process", Applied Intelligence, Vol. 16, 2002, pp. 7-17.

[8] MARIA CLÁUDIA FIGUEIREDO PEREIRA EMER, AND SILVIA REGINA VERGILIO, "GPTEST: A TESTING TOOL BASED ON GENETIC PROGRAMMING", PROCEEDINGS OF THE GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE (GECCO 2002), SEPTEMBER 2002, PP. 1343-1350.

[9] Wen-Yang Lin, Wen-Yung Lee and Tzung-Pei Hong, "Adapting Crossover and Mutation Rates in Genetic Algorithms", Journal Of Information Science And Engineering, Vol. 19, 2003, pp. 889-903.

[10] Abdelaziz M. Khamis, Moheb R. Girgis and Ahmed S. Ghiduk, "Automatic Software Test Data Generation for Spanning Sets Coverage Using Genetic Algorithms", Computing and Informatics, vol. 26, no. 4, 2007, pp. 383–401.

[11] LAWRENCE BEADLE AND COLIN G JOHNSON, "SEMANTICALLY DRIVEN MUTATION IN GENETIC PROGRAMMING", PROCEEDINGS OF THE IEEE CONGRESS ON EVOLUTIONARY COMPUTATION (CEC 2009), 2009, PP. 1336-1342.

[12] WILLIAM B. LANGDON, MARK HARMAN AND YUE JIA, "MULTI OBJECTIVE HIGHER ORDER MUTATION TESTING WITH GENETIC PROGRAMMING", TESTING: ACADEMIC AND INDUSTRIAL CONFERENCE - PRACTICE AND RESEARCH TECHNIQUES, 2009 (TAIC PART '09.), 4-6 SEPT. 2009, PP., 21-29.

[13] WANG JUN , ZHUANG YAN AND JIANYUN CHEN, "TEST CASE PRIORITIZATION TECHNIQUE BASED ON GENETIC ALGORITHM", PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON INTERNET COMPUTING & INFORMATION SERVICES (ICICIS), 17-18 SEPT. 2011, PP. 173 – 175.

[14] PRENAL B. NIRPAL AND K.V. KALE, "USING GENETIC ALGORITHM FOR AUTOMATED EFFICIENT SOFTWARE TEST CASE GENERATION FOR PATH TESTING", INTERNATIONAL JOURNAL OF ADVANCED NETWORKING AND APPLICATIONS, VOL. 2, NO. 6, 2011, PP. 911-915.

[15] TIMO KÖTZING, ANDREW M. SUTTON, FRANK NEUMANN AND UNA-MAY O'REILLY, "THE MAX PROBLEM REVISITED: THE IMPORTANCE OF MUTATION IN GENETIC PROGRAMMING", PROCEEDINGS OF THE GENETIC AND EVOLUTIONARY COMPUTATION (GECCO'12), JULY 7–11, 2012, PP. 1333-1340.

[16] Chayanika Sharma, Sangeeta Sabharwal, Ritu Sibal, "A Survey on Software Testing Techniques using Genetic Algorithm", IJCSI International Journal of Computer Science Issues, vol. 10, issue 1, no 1, January 2013, pp. 391-393.